UNIVERSIDAD DE SANTIAGO DE COMPOSTELA
FACULTAD DE FÍSICA
DEPARTAMENTO DE ELECTRÓNICA Y COMPUTACIÓN

TESIS DOCTORAL

Técnicas de compilación para la paralelización de códigos irregulares

Presentada por:

David Expósito Singh

Octubre 2003

Dr. **Francisco Fernández Rivera**,
Profesor Titular de Arquitectura y Tecnología de Computadores de la Universidad de Santiago de Compostela.

Dra. **María Martín Santamaría**,
Profesora Titular de Arquitectura y Tecnología de Computadores de la Universidad de A Coruña.

CERTIFICAN:

Que la memoria titulada "**Técnicas de compilación para la paralelización de códigos irregulares**", ha sido realizada por D. **David Expósito Singh** bajo nuestra dirección en el Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela y concluye la Tesis que presenta para optar al grado de Doctor en Ciencias Físicas.

Santiago, 16 de Octubre de 2003

Dr. Francisco Fernández Rivera
Codirector de la tesis

Dra. María Martín Santamaría
Codirectora de la tesis

Fdo. Dr. **Diego Cabello Ferrer**,
Director del Departamento de
Electrónica y Computación.

A mis padres
y a mis hermanos

Agradecimientos

Quiero dar mi más sincero agradecimiento a todas las personas que han contribuido a la realización de esta tesis tanto en el ámbito académico como en el personal.

El primer lugar quiero dar las gracias a mis codirectores, los profesores D. Francisco Fernández Rivera y Dña. María Martín Santamaría, por toda la ayuda que me han prestado. Ellos me dieron la oportunidad de iniciar mis estudios de doctorado, y me han ofrecido siempre todo su apoyo y colaboración. Quiero agradecerles muy especialmente todo el tiempo y esfuerzo que me han dedicado, así como la confianza que han depositado en mí.

También quiero expresar mis agradecimientos a todos los miembros del Departamento de Electrónica y Computación, y en particular a aquellos que pertenecen, o han pertenecido, al Grupo de Arquitectura de Computadores. Con ellos he compartido muchos momentos "mágicos" que han hecho que esta etapa de mi vida haya sido tan gratificante.

Gracias especialmente a aquellos con los que en algún momento he compartido despacho: Álvaro, Antonio, Dora, Inma, Juanjo, Marcos, Patricia, Pichel y Roberto. Todos ellos me ofrecieron su amistad, y me ayudaron en los momentos más difíciles.

A Antonio le debo mi iniciación (o casi debería decir inicialización) en la gestión de redes de computadores. Él fue la primera persona del grupo que conocí y la que despertó en mí el interés por la programación de sistemas distribuidos. Roberto ha hecho que mi vida fuera mucho más amena, y me ha dado ideas que resultaron de gran utilidad para mi trabajo. A Juan le tengo que agradecer, ante todo, su compañía a lo largo de todos estos años. Se puede decir que él estaba allí desde el primer momento, desde aquella tarde de julio. Juntos hemos pasado muchas aventuras. Lo más sorprendente es que siempre me ha sabido ayudar, independientemente del tipo de problema que hubiera tenido, y en parte, gracias a su ayuda esta tesis ha podido finalizarse.

Quiero agradecer también al profesor Michael O'Boyle del Institute for Computing

Systems Architecture de la Universidad de Edimburgo, por su interés mostrado en mi trabajo, así como por su asesoramiento en distintos tópicos relacionados con la paralelización de códigos irregulares.

Quiero darles las gracias a todos mis amigos que me ofrecieron su compañía y me prestaron su ayuda cuando más la necesitaba. En especial a Vladimir, por su colaboración en el diseño gráfico de esta tesis. A Emilio, con quien compartí grandes momentos en Escocia, y me ofreció la infraestructura necesaria para mantenerme comunicado y poder realizar buena parte de mi trabajo de investigación. A Álvaro, por todos los grandes momentos que hemos pasado juntos y por su carácter optimista que siempre ha sido una fuente de motivación para mí. A Olalla, por su amistad, aprecio y simpatía durante todo este tiempo. A Silvia, por su compañía a lo largo de estos años, y por todas las risas y sustos que hemos compartido.

Quiero agradecer al EPCC (Edinburgh Parallel Computing Centre), al CSC (Centro de Supercomputación Complutense) y al CESGA (Centro de Supercomputación de Galicia) por darme acceso a sus recursos de computación distribuida.

A la Xunta de Galicia por la beca predoctoral que he disfrutado durante estos años. Igualmente, al CICYT por el soporte económico dado por el proyecto TIC2001-3694-C02, a la Secretaria Xeral de I+D de Galicia por la financiación del proyecto PGIDT99-PXI20602B, al Fondo Europeo de Desarrollo Regional (FEDER) por el proyecto 1FD97-0118-C02 y al TRACS European Community Access to Research Infrastructure por el proyecto HPRI-CT-1999-00026.

A mi familia, por todo el apoyo que siempre me han dado. Ellos han tenido la paciencia necesaria para que yo pudiera realizar mi trabajo de la forma más cómoda y despreocupada.

Xa para rematar, debo engadir nestes agradecementos unhas verbas a Paula. A ela débolle moitas cousas. A parte da sua amizade e compañía, ela foi e é pra min unha referencia en canto a capacidade de traballo e esforzo persoal. Con ela compartín unha infinidade de momentos únicos, e a sua vitalidade fixo que eu poidera acadar moitas das metas propostas nesta tese.

"*Y no dejamos de preguntarnos,*
una y otra vez,
Hasta que un puñado de tierra
Nos calla la boca ...
Pero ¿es eso una respuesta?"

Heinrich Heine
Lazarus

Índice General

Índice de Tablas

Índice de Figuras

Notación

Símbolo	**Significado**
$\mathcal{A}_i$	Conjunto de entradas de a pertenecientes a la partición i-ésima
a	Vector de accesos
$\mathcal{C}$	Curva *Freeman chain-code*
CC	Camino crítico de un lazo irregular
col	Vector de distribución por columnas de una matriz dispersa
$\mathcal{CT}$	Conjunto de carga de trabajo
$\mathcal{D}$	Secuencia de puntos dominantes de la curva $\mathcal{C}$
$\mathcal{E}^u$, $\mathcal{E}^l$	Secuencia de puntos asociadas a la envolvente superior e inferior a $\mathcal{C}$
$\mathcal{E}_f^u$, $\mathcal{E}_f^l$	Curva digital que determina la envolvente superior e inferior a $\mathcal{C}$
$\mathcal{G}_i$	Conjunto de entradas de g pertenecientes a la partición i-ésima
g	Vector de guarda
IARD	*Irregular Access Region Descriptor* de un vector de indirección
L	Tamaño de partición de a
l	Vector de accesos mínimos en una clasificación por *slices*
$\mathcal{LIM}$	Conjunto de particionamiento de a
lim_i^{inf}, lim_i^{sup}	Límite inferior y superior asociados a la i-ésima partición de a
N_a	Número de entradas del vector a
$N_\mathcal{C}$	Número de elementos de la curva $\mathcal{C}$
$N_\mathcal{D}$	Número de elementos del conjunto $\mathcal{D}$
$N_\mathcal{E}$	Número de elementos de los conjuntos $\mathcal{E}^u$ y $\mathcal{E}^l$
N_p	Número de procesadores
N_{pt}	Número total de particiones de a
N_S	Número de *slices*
$N_{\mathcal{SL}}$	Número de secciones lineales

Símbolo	**Significado**
N_{stmt}	Número de estamentos de un lazo irregular
N_x	Número de entradas del vector de indirección x
p^{max}	Punto de mayor ordenada de un segmento de $\mathcal{C}$
row	Vector de índice de filas de una matriz dispersa
$\{s^1, s^2, s^3, s^4\}_i$	Conjunto de índices de *slices* asociados a la partición i-ésima de a
S_k	Conjunto de entradas del vector de indirección del *slice* número k
$\mathcal{SL}$	Sección lineal del patrón de acceso
$STMT^k$	k-ésima familia de estamentos de un lazo irregular
$stmt$	Estamento de un lazo irregular
$ticket$	*Ticket table*
TT_i	Tabla de tangencia asociada al segmento i-ésimo de $\mathcal{C}$
u	Vector de accesos máximos en una clasificación por *slices*
val	Vector de valores de una matriz dispersa
W	Coste del lazo sintético
x	Vector de indirección
x^{fin}	Vector de indirección reordenado
$\mathcal{X}_{s^a}^{s^b}$	Conjunto de entradas del x comprendidas entre los *slices* s^a y s^b
α, β, γ	Coste computacional asociado a cada región del patrón de acceso
κ	Densidad de entradas de x entorno al límite superior de una partición
λ	Anchura media de la banda del patrón de acceso
υ_p	Potencia de cálculo del procesador p
ρ	Vector densidad de una clasificación por *slices*
ρ_{ini}	Vector densidad acumulada

Prefacio

Los sistemas de computación actuales presentan una capacidad de explotación del paralelismo cada vez mayor. Podemos encontrar en el mercado una amplia gama de arquitecturas paralelas que ofrecen una solución competitiva a las demandas de computación existentes. Este paralelismo se explota en todos los niveles que conforman la arquitectura de un computador. Así pues, si consideramos las nuevas tecnologías de diseño de microprocesadores, existe una tendencia a integrar en un mismo circuito un número cada vez mayor de unidades de procesamiento. Si consideramos la organización interna de un computador también existe una tendencia a aumentar el número de procesadores. Si consideramos conjuntos de ordenadores interconectados a través de una red, vemos que también la computación distribuida está siendo objeto de intensos trabajos de investigación.

Debido a que la inmensa mayoría del software existente ha sido diseñado para sistemas con un único procesador, surge la necesidad de adaptar su funcionamiento a un entorno de ejecución paralela. Este proceso resulta extremadamente costoso y en la actualidad se realiza tanto de forma manual, como a través de herramientas de paralelización automática cuya eficiencia es todavía limitada. Los avances conseguidos en este campo se producen a una velocidad menor que la introducción de nuevas tecnologías paralelas. Dicho con otras palabras, cada vez tenemos una cantidad creciente de recursos que explotar y una mayor dificultad para hacerlo. Por este motivo cada vez resulta más crítico la introducción de nuevas propuestas que permitan explotar más eficientemente el paralelismo de los programas.

Por otra parte, la introducción de nuevos procesos de fabricación de circuitos integrados hace que la velocidad de cálculo de los procesadores aumente a un ritmo superior a la velocidad de acceso a memoria. Esto origina un paulatino desequilibrio entre ambos factores que hace que en la actualidad la latencia en los accesos a memoria suponga uno de los principales cuellos de botella en el rendimiento de un computador. Así pues, también es necesario el desarrollo de técnicas que permitan minimizar este efecto, explotando de forma eficiente los niveles inferiores de la jerarquía de memoria del sistema.

Un número significativo de aplicaciones reales contiene estructuras de código irregular. Los códigos irregulares se caracterizan por tener un mal comportamiento en los dos tópicos que hemos comentado anteriormente. Por una parte presentan una gran dificultad de análisis, lo que impide extraer el paralelismo de una forma eficiente. Por otra, estos códigos muestran en un gran número de casos una incorrecta explotación de la jerarquía de memoria, lo que limita su eficiencia.

La temática general en la que se enmarca esta tesis es el desarrollo de nuevas estrategias que permitan aumentar el rendimiento en la ejecución de códigos irregulares. Hemos particularizado nuestro estudio a sistemas multiprocesadores, donde es necesario hacer un especial hincapié en los dos factores que determinan de forma más importante la eficiencia de un programa: la extracción eficiente del paralelismo y la explotación, también eficiente, de la jerarquía de memoria. Debido a la estructura de esta clase de códigos, parte del proceso de análisis debe ser realizado durante la ejecución del programa. Por este motivo, un factor adicional que es necesario considerar lo supone el diseño de nuevas rutinas de análisis que minimicen su impacto sobre la ejecución del programa.

Las principales contribuciones realizadas en esta tesis son las siguientes:

- El desarrollo de una estrategia que permite caracterizar el conjunto de accesos a memoria realizados por un código irregular. Esta caracterización es posteriormente utilizada por distintas técnicas de optimización, permitiendo reducir su coste computacional.
- Una nueva técnica de análisis de dependencias con la que podemos determinar la no existencia de conflictos de acceso a memoria de dos códigos irregulares.
- Un conjunto de técnicas de reestructuración tanto del código irregular como de los datos asociados. Estas técnicas están destinadas a extraer el paralelismo y explotar la jerarquía de memoria del sistema.
- Un esquema de organización de las distintas propuestas en un árbol de decisión que permite determinar la solución más eficiente en función de las características del código considerado.

Los distintos trabajos elaborados en el marco de esta tesis han contribuido al desarrollo de las siguientes publicaciones: [96, 97, 127, 128, 129, 130, 131, 132, 133].

Capítulo 1

Introducción

Desde el desarrollo del primer ordenador digital programable, de forma pareja al aumento del poder de cálculo y complejidad de estos sistemas, se han incrementado los requisitos computacionales de las aplicaciones ejecutadas en los mismos. Dichos requisitos se deben a dos factores: la carga computacional del programa (evaluada en número de instrucciones) y la cantidad de memoria requerida (evaluada en número de bits). A lo largo de los años, se han desarrollado técnicas de programación cada vez más complejas que demandan una mayor cantidad de recursos. Dentro de este contexto destacamos el desarrollo de mecanismos de acceso a memoria más eficientes, como es el caso del acceso a estructuras de datos irregulares.

Un código irregular se caracteriza por emplear una estructura de acceso compleja con la cual, en muchas situaciones, se logra un aumento del rendimiento del programa mediante una reducción del volumen de datos almacenados. En la actualidad no existe uniformidad de criterios para calificar a una aplicación como irregular. En términos generales, existen dos vertientes distintas para realizar esta clasificación. La primera de ellas se basa en las características del conjunto de posiciones de memoria accedidas a lo largo de la ejecución del programa. De este modo, una aplicación irregular (como contrapartida a una regular) accede a posiciones de memoria no estructuradas que pueden variar dinámicamente durante la ejecución del programa. En la segunda vertiente, el criterio de clasificación está basado en el tipo de mecanismo de indireccionamiento a memoria empleado. Se dice que una aplicación es irregular si contiene estructuras de indireccionamiento que impiden determinar, en tiempo de compilación, el conjunto de accesos a memoria realizados por la aplicación. Ejemplos de estos accesos son los punteros, los vectores de indirección cuyo contenido es desconocido en tiempo de compilación, o el uso de funciones externas

empleadas para direccionar la memoria y cuya estructura tampoco puede ser determinada.

En la actualidad, una gran número de aplicaciones hacen uso de estructuras de almacenamiento irregular. En este sentido, y situándonos en el contexto de la computación de alto rendimiento, resulta necesario que estas aplicaciones sean capaces de explotar, con la mayor eficiencia posible, la arquitectura concreta del sistema en el que son ejecutadas. Usualmente, las aplicaciones se han desarrollado y se siguen desarrollando como programas secuenciales, que son posteriormente modificados y adaptados para permitir su ejecución en un sistema multiprocesador. Desafortunadamente, la amplia variedad de arquitecturas y modelos de paralelización existentes hace necesario que este proceso se realice de forma específica para cada sistema. En muchos casos este proceso se debe realizar manualmente, acarreando importantes costes materiales que pueden hacer inviable el proceso de reestructuración del programa. Con el fin de reducir estos costes se han desarrollado herramientas de paralelización automática que permiten automatizar de forma parcial, o total, este proceso. El funcionamiento de estas herramientas se basa en la identificación y caracterización de las distintas partes de código que comprenden el programa y en la realización de un correcto análisis de dependencias. Para poder realizar dicho análisis es necesario conocer el conjunto de posiciones de memoria accedidas a lo largo de la ejecución del programa. En el caso de los códigos regulares, los accesos a memoria están especificados en la mayor parte de los casos mediante expresiones simples y fácilmente analizables, permitiendo realizar este proceso en tiempo de compilación. Este hecho permite alcanzar una elevada precisión en el análisis de dependencias facilitando el desarrollo de soluciones eficientes para realizar su paralelización de forma automática.

En el caso de los códigos irregulares, los mecanismos de acceso empleados tienen una elevada complejidad y son desconocidos en tiempo de compilación, haciendo que el análisis de dependencias no pueda ser realizado en esta etapa. Ese hecho dificulta enormemente el proceso de paralelización automática, y hace que las herramientas actuales no puedan ofrecer una solución eficiente para la mayor parte de los casos. Otro factor que limita de manera importante el proceso de extracción de paralelismo de un código irregular es la amplia variedad de estructuras existentes. Dado que su definición sólo hace referencia al modo de direccionamiento de los datos, pueden existir códigos irregulares con distinto número de indirecciones, con distintos tipos de dependencias y con mecanismos de acceso diferentes. Cada uno de los cuales necesita un tratamiento específico por parte de la herramienta de análisis, lo que hace que en muchos de los casos, su paralelización quede inabordada. Las estructuras de código irregular suelen aparecer con frecuencia en las regiones más críticas del programa. Debido a este hecho, desde el punto de vista de su adecuación a sistemas multiprocesador se ha originado la necesidad de abordar la paralelización automática de

este tipo de códigos.

Es en este punto donde situamos el contexto en el cual se inició esta tesis. El objetivo propuesto consiste en la optimización de códigos irregulares en el marco de la computación de alto rendimiento. Entendemos por "optimización" al conjunto de procesos que permiten aumentar la eficiencia del programa. Ejemplos de estos procesos son la reestructuración del código secuencial, la mejora en la localidad o la paralelización del programa. En este contexto hemos considerado necesario introducir nuevas propuestas para el análisis y caracterización de códigos irregulares que permitan realizar el proceso de optimización del modo más eficiente. El fin último de nuestro trabajo es la integración de las distintas propuestas en una herramienta de paralelización automática.

En las siguientes secciones describiremos de forma más detallada cada uno de los distintos tópicos que hemos perfilado hasta este momento. Comenzaremos introduciendo las diferentes arquitecturas existentes y los distintos modelos de programación paralela. A continuación definiremos de forma más precisa el tipo de código irregular que hemos considerado, así como sus principales características. Finalmente, mostraremos un esquema general de las contribuciones realizadas en esta tesis.

1.1 Arquitecturas paralelas

Partiendo de la evolución histórica en la arquitectura de los sistemas paralelos, podemos considerar una clasificación en cuatro grandes categorías:

- Multiprocesadores vectoriales.
- Sistemas de memoria distribuida.
- Sistemas de memoria compartida.
- Sistemas altamente distribuidos.

Las arquitecturas vectoriales disponen de unidades aritméticas altamente segmentadas orientas al procesamiento de grandes matrices de datos. Un ejemplo clásico de sistema multiprocesador basado en esta arquitectura fue el Cray X-MP de Cray Research, el cual inició una larga familia de sistemas vectoriales paralelos que finalizó con las series T-90 de Cray. Para conseguir una alta eficiencia, estos sistemas requieren que el código disponga de un basto número de operaciones libres de dependencias. Dichas operaciones deben mostrar una alta regularidad, tanto en su indireccionamiento en memoria como en el tipo

de operación aritmética realizada. Sólo en estos casos el código puede ser eficientemente ejecutado de forma segmentada y paralela. Esta restricción impone una importante limitación en el empleo de este tipo de sistemas, haciendo que su uso resulte eficiente para un conjunto limitado de aplicaciones. A pesar de haber perdido popularidad, hoy en día podemos encontrar en el mercado diferentes sistemas basados en esta arquitectura. Ejemplo de los mismos son las series VPP de Fujitsu, las Series S802 de Hitachi y las series SX de NEC.

Un sistema de memoria distribuida consiste en un conjunto de nodos unidos por una red de interconexión. Cada nodo consta de una memoria local y de uno o varios procesadores. Cada procesador puede acceder a su memoria local, mientras que el único modo de acceso a una memoria remota es mediante el envío de mensajes a través de la red de interconexión. Existen distintas topologías de redes de interconexión, algunas de las cuales pueden alcanzar una alta escalabilidad, permitiendo el desarrollo de sistemas con un ingente número de nodos. Como contraposición, es necesario pagar el precio de una alta latencia en los accesos a posiciones de memoria remotas, lo cual supone el principal factor limitante del rendimiento de estos sistemas. Existen en el mercado un gran número de modelos basados en esta arquitectura. Como ejemplos más representativos podemos citar el sistema T3D de Cray Research, los Connection Machines CM-5 y CM-5E de Thinking Machines Corporation, el SP-2 de IBM y el Sandia de Intel.

En la tercera categoría de nuestra clasificación se sitúan los sistemas de memoria compartida. Estos sistemas constan de una serie de procesadores y bancos de memoria unidos por una red de interconexión. La principal característica de esta clase de sistemas radica en la incorporación de mecanismos (típicamente gestionados por el sistema operativo) que permiten que cada procesador pueda acceder al espacio global de memoria sin la comunicación explícita de mensajes. Dependiendo de la topología concreta de la red de conexión distinguimos, dentro de esta categoría, dos tipos de arquitecturas: la UMA (*Uniform Memory Access*) y la NUMA (*Non-Uniform Memory Access*). En la primera de ellas la latencia de acceso a memoria tiene un valor constante que no depende ni del procesador que realiza el acceso ni del banco de memoria accedido. Los sistemas UMA suelen estar organizados mediante un bus de interconexión que conecta los procesadores y los bancos de memoria. Esta estructura se caracteriza por su relativa simplicidad frente a otros esquemas de conexión más complejos. El principal inconveniente de este tipo de arquitecturas es su baja escalabilidad, debido a lo cual sólo alcanza una buena eficiencia con un reducido número de procesadores. Ejemplos de estos sistemas son el Power Challenge de Silicon Graphics, los Y-MP y C90 de Cray y las familias HPC4500 de SUN. En una arquitectura UMA el esquema de distribución de los datos no tiene un impacto significativo

en el rendimiento del sistema. Considerando la jerarquía de memoria de las arquitecturas actuales, el coste de acceso a memoria cache es significativamente inferior al de memoria principal. Por este motivo, la obtención de una alta localidad en los accesos a nivel de memoria cache supone un factor crítico en el rendimiento de esta clase de sistemas.

La segunda gran familia de sistemas de memoria compartida son las arquitecturas NUMA. Estas emplean redes de interconexión con topologías más complejas, en las que tanto los procesadores como los bancos de memoria están distribuidos. La memoria global del sistema, a pesar de poder ser accedida por cualquier procesador, presenta una latencia variable que depende de la posición relativa al procesador que realiza el acceso. Una manera de reducir la latencia y aumentar la escalabilidad consiste en la distribución de los bancos de memoria en nodos siguiendo una estructura similar a los sistemas de memoria distribuida no compartida. De este modo, la red de interconexión une distintos nodos los cuales constan de uno o varios procesadores y una porción de la memoria física global del sistema.

La principal característica de una arquitectura NUMA es la incorporación de mecanismos *hardware* para gestionar los accesos a memoria remota y para mantener la coherencia de la información del mismo modo que sucede en las arquitecturas UMA. Algunos sistemas, denominados *Cache-Coherent NUMA* (CC-NUMA), mantienen la coherencia en todos los niveles de la jerarquía de memoria. En este tipo de sistemas, la obtención de una alta localidad en los accesos tiene una gran influencia en su rendimiento. A diferencia de las arquitecturas UMA, donde sólo es necesario explotar la memoria cache, en un sistema NUMA también es importante maximizar la localidad en los accesos a memoria principal. Como ejemplos de arquitecturas NUMA podemos citar la familia Origin de Silicon Graphics, el T3E de Cray y el HP Exemplar. En la actualidad el uso de estas arquitecturas está muy diversificado y gozan de una gran popularidad debido a que obtienen un correcto equilibro entre escalabilidad y facilidad de programación.

Una última categoría de nuestro esquema de clasificación lo compone los sistemas altamente distribuidos. En esta clase de sistemas se prima la modularidad de cada elemento a costa de unas prestaciones inferiores en la red de interconexión. Recientemente, el empleo de esta clase de sistemas está experimentando un auge y su diseño y optimización está siendo objeto de intensos estudios de investigación. Los aspectos claves de este tipo de arquitectura son el establecimiento de un proceso de control del sistema distribuido. Control que debe realizar una distribución eficiente de las computaciones en función de las características potencialmente heterogéneas de esta arquitectura.

El tipo de sistema paralelo ejerce un papel determinante en los mecanismos de optimización utilizados. Nuestras propuestas están enmarcadas en el empleo de sistemas paralelos de memoria compartida. Existen distintos motivos para realizar esta elección: el primero de ellos es que debido a su amplia difusión y popularidad, entendemos que las aportaciones realizadas en esta tesis suponen un mayor beneficio a la comunidad científica. El segundo motivo es la disponibilidad de herramientas de paralelización automática orientadas a esta arquitectura en las que nuestras propuestas pueden ser integradas. Un último motivo que hemos encontrado para el empleo de esta clase de sistemas es su disponibilidad, tanto en el propio departamento, como en otros centros de supercomputación a los que tenemos acceso.

1.2 Modelos de programación paralela

En la actualidad existen tres modelos ampliamente difundidos de programación paralela,

- Modelo de programación por pase de mensajes.
- Modelo de programación *Data Parallel*.
- Modelo de programación sobre memoria compartida.

El modelo de programación por pase de mensajes fue introducido para la paralelización de programas en sistemas distribuidos. En la actualidad, este modelo tiene un uso extendido en un gran número de plataformas, siendo también soportado por la mayor parte de las arquitecturas de memoria compartida. Este modelo de programación se basa en la comunicación de los procesadores implicados en la ejecución del programa mediante el envío y la recepción de mensajes. Típicamente, el esquema de programación utilizado es el de *Simple Programa Múltiple Dato* (SPMD) en el que se deben establecer de forma explícita los distintos tipos de operaciones de comunicación y sincronización entre los procesadores, así como la distribución de los datos. Estas operaciones suelen estar contenidas en una librería de rutinas de comunicación. Como ejemplos de estas librerías podemos citar la *Parallel Virtual Machine* [46] (PVM), la cual fue la primera librería en ser ampliamente utilizada. Posteriormente se ha introducido la *Message Passing Interface* [48] (MPI), la cual se ha establecido como un estándar de comunicación por pase de mensajes. Otros ejemplos son las primitivas *put* y *get* [102] empleadas en sistemas de memoria compartida. La programación mediante este modelo se realiza de forma manual, en función de

las características del programa y del tipo de arquitectura utilizada. Este hecho resta versatilidad y flexibilidad a esta propuesta, introduciendo una importante limitación para su empleo.

El modelo de programación *Data Parallel* supera esta limitación haciendo que los procesos de comunicación y sincronización sean transparentes al usuario. De esta forma, el programador únicamente debe establecer la distribución de los datos sobre un espacio de memoria que se muestra como global. Posteriormente, en la etapa de compilación se genera de forma automática el programa SPMD específico para cada arquitectura. Dicho programa contiene todas las rutinas de comunicación y sincronización necesarias para llevar a cabo la distribución de los datos y la ejecución del programa paralelo. Ejemplos de modelos *data-parallel* son el Fortran-D [66], el Vienna Fortran [25] y el High Performance Fortran [82] (HPF). Este último modelo goza de una gran difusión, pudiéndose utilizar en la práctica totalidad de las plataformas existentes. A pesar de ello, en los últimos años han surgido críticas por la limitada eficiencia de su empleo en la paralelización de programas.

Los modelos de programación sobre sistemas de memoria compartida se basan en el uso de directivas de paralelización mediante las cuales se realizan las operaciones de distribución de los datos y de sincronización entre los procesadores. Dado que un sistema de memoria compartida facilita que cada procesador tenga acceso al espacio global de memoria, desaparece la necesidad de comunicar explícitamente los datos. Mediante este modelo, además de permitir realizar la distribución de los diferentes datos, se puede determinar la granularidad del paralelismo utilizado (iteraciones de un lazo, secciones de código, etc.) y especificar la carga de trabajo asignada a cada procesador. La programación se realiza, tanto a través de directivas de alto nivel que especifican el grado de paralelismo del programa, como de operaciones de sincronización de bajo nivel que determinan un orden adecuado de ejecución y de acceso a los datos. Inicialmente, se establecieron dos modelos de programación ampliamente difundidos: el Parallel Computing Forum [43] (PCF) y OpenMP [24]. Adicionalmente, existía un variado número de librerías de memoria compartida asociadas a cada sistema particular. Ejemplos de las mismas son el SUNMP [112] y las directivas MP de Silicon Graphics [92]. Estas librerías ofrecían un conjunto de directivas específicas a la arquitectura del sistema particular, las cuales permitían generar, en dichas arquitectura, una versión paralela que por lo general era eficiente. El inconveniente de estas directivas específicas era la pérdida de portabilidad en el programa paralelo. En la actualidad, el OpenMP se ha convertido en el modelo de programación estándar de sistemas de memoria compartida. Su uso está ampliamente difundido, y es el modelo adoptado por la mayor parte de los compiladores.

OpenMP es una especificación destinada a expresar el paralelismo en sistemas de memoria compartida mediante implementación portable para Fortran, C y C++. El OpenMP ofrece un conjunto de directivas de compilación que permiten establecer regiones paralelas en el programa, realizar distintos tipos de sincronizaciones, y dar soporte para la compartición o privatización de los datos. Adicionalmente, contiene una librería de rutinas que permiten controlar el entorno de ejecución del programa y dan soporte para la realización accesos atómicos a posiciones de memoria. Finalmente, este modelo de programación permite especificar, mediante variables de entorno, ciertos parámetros asociados a la ejecución paralela.

En el trabajo desarrollado en esta tesis hemos adoptado el modelo de programación sobre memoria compartida, dado que es la opción más adecuada para el tipo de arquitectura utilizada. Más concretamente, y debido a su amplia difusión y versatilidad, hemos empleado OpenMP sobre programas escritos en Fortran 77. La elección de este lenguaje de programación se debe a dos motivos. Por una parte el Fortran 77 es el lenguaje de programación utilizado por la mayor parte de los códigos de pruebas que hemos considerado. Esto se debe a que es el lenguaje de programación más utilizado en aplicaciones en las que prima el carácter computacional. El segundo motivo radica en que también es el lenguaje empleado por otros autores para implementar sus técnicas de optimización, lo cual nos va a facilitar la realización de comparativas con las propuestas que nosotros vamos a realizar.

1.3 Herramientas de paralelización automática

Actualmente existen distintas herramientas destinadas a realizar de forma automática la optimización y paralelización de programas. Ejemplo de dichas herramientas son los compiladores académicos Polaris, SUIF y PIPS, y el compilador comercial MIPSpro. En esta sección describimos sus principales características.

Comenzaremos nuestra descripción con el compilador **Polaris** [18], desarrollado en la Universidad de Illinois. Esta herramienta utiliza como entrada un código escrito en Fortran 77 que convierte, después de realizar distintos procesos de análisis y optimización [41], en un código paralelo orientado a una arquitectura específica. En la actualidad, este compilador genera códigos para arquitecturas de memoria compartida UMA como el Power Challenge de Silicon Graphics o el Convex C3, arquitecturas de memoria compartida NUMA como la familia Origin de Silicon Graphics o el Cray T3E y arquitecturas de memoria distribuida como el Cray T3D.

Las principales características de este compilador son su capacidad de realizar un

análisis simbólico del programa, mediante el cual puede identificar operaciones de reducción tanto escalares como de histograma, caracterizar las variables de inducción obteniendo su expresión analítica, llevar a cabo la eliminación de código muerto, aplicar técnicas de *in-linning*, privatizar variables [139], etc. Además, incorpora distintos mecanismos de análisis de dependencias que pueden ser extendidos a un análisis interprocedural [117]. Adicionalmente, esta herramienta permite introducir en el programa de entrada una serie de directivas para indicar, de forma explícita, el grado de paralelismo de diferentes partes del programa. Una importante característica de Polaris es que permite acceso a su representación interna [41], lo cual sirve de infraestructura para implementar y evaluar nuevas técnicas de análisis y optimización de códigos.

El rendimiento de esta herramienta ha sido evaluado de forma exhaustiva empleando conjuntos de programas de prueba como el *Perfect Benchmark* y el SPEC95fp [38]. Los resultados obtenidos muestran su eficiencia cuando se aplica sobre códigos regulares. En el caso de los códigos irregulares, la eficiencia de Polaris es más limitada, restringiéndose a la identificación y paralelización de operaciones de reducción. Esta última operación se lleva a cabo mediante la técnica de *array expansion* [42, 38]. En este sentido, uno de los desafíos existentes en la actualidad es la incorporación de nuevas propuestas en este campo.

El compilador SUIF [55] (*Stanford University Intermediate Format2*), desarrollado en la Universidad de Stanford, ofrece una plataforma para la investigación y de desarrollo de nuevas técnicas de optimización de programas. En su diseño se ha primado la facilidad de empleo y la capacidad de modificar y ampliar su contenido de un modo simple. Esta herramienta es capaz de analizar programas escritos en lenguaje C o Fortran, devolviendo, en ambos casos, un código paralelo SPMD expresado en lenguaje C.

Esta herramienta incluye técnicas de análisis de dependencias y análisis simbólico, permitiendo extraer paralelismo en el código y mejorar la localidad tanto del programa secuencial como del paralelo. Una de sus principales bazas es su importante mecanismo de análisis interprocedural. La herramienta SUIF incorpora distintas técnicas para identificar y paralelizar operaciones de reducción, tanto regulares como irregulares. En este último caso emplea, como método de paralelización, la técnica de *replicated buffers* [145], la cual se describirá posteriormente. SUIF permite la portabilidad del código paralelo generado a plataformas de memoria compartida de Silicon Graphics, el sistema multiprocesador DASH de Stanford [89] y el Kendall Square Research KSR1.

El funcionamiento del compilador SUIF se basa en la representación del programa fuente en un lenguaje intermedio, el cual es analizado en etapas sucesivas por los dife-

rentes módulos que comprenden el compilador. Este hecho otorga de gran flexibilidad al proceso de compilación, permitiendo controlar y evaluar cada una de las etapas del mismo. Sin embargo, la estructura desacoplada de los diferentes módulos implica un uso intensivo de operaciones de acceso a disco, lo cual resulta altamente ineficiente. En la actualidad, se está desarrollado una nueva versión de esta herramienta [88] que resuelve esta limitación mediante una nueva estructura interna, mucho más modular y eficiente. Esta nueva versión, denominada SUIF2, incorpora nuevos mecanismos para adquirir un mayor número de estructuras de código y para realizar un análisis del programa mucho más amplio. Adicionalmente, existen otras propuestas [91] basadas en este compilador para permitir al usuario interactuar de un modo eficiente en el proceso de paralelización.

Otra herramienta académica de paralelización automática es el compilador PIPS [71]. El rango de aplicación es en este caso más limitado, restringiéndose al análisis y optimización de programas de procesamiento de señales. El PIPS emplea, como lenguaje de programación, el Fortran 77, y genera códigos paralelos con las directivas OpenMP, PVM o MPI. Dentro de sus principales características, destacamos la privatización de matrices o escalares, la optimización de código a nivel de lazo y la paralelización.

Dentro del campo de las herramientas comerciales destacamos el paralelizador MIPSpro [47]. Esta herramienta está incluida en los compiladores de C, C++, Fortran 77 y Fortran 90 desarrollados por Silicon Graphics. Como principales características, el MIPSpro realiza análisis de dependencias interprocedural, optimización a nivel lazo y paralelización de código. A pesar de estas funcionalidades, esta herramienta es incapaz de extraer paralelismo de forma automática en secciones de código irregulares.

Basándonos en las características de las herramientas existentes, podemos concluir que el campo de la paralelización automática de códigos irregulares está muy poco desarrollado. En el mejor de los casos, el análisis y la paralelización de lazos irregulares se realiza mediante la identificación de las secciones de código irregular, y la aplicación de técnicas estándar de paralelización, que no siempre obtienen un buen rendimiento. Este hecho da muy poca flexibilidad al proceso de análisis, y restringe de forma importante el número de estructuras que pueden ser consideradas. Una solución propuesta en algunas herramientas consiste en la introducción de directivas que indican de forma expresa la existencia de dependencias de datos. Aunque de este modo es posible paralelizar un mayor conjunto de códigos irregulares, este proceso no dista mucho de la paralelización manual, y su empleo no resulta del todo claro. Por todos estos motivos, entendemos que nuestra propuesta debe estar orientada a su inclusión en una herramienta de paralelización automática que permita superar estas carencias.

1.4 Códigos irregulares

Debido a la generalidad de su definición y a la gran variedad de códigos irregulares existentes, plantear la optimización de toda la familia de códigos irregulares resulta un problema inabordable en el marco de una tesis. Por este motivo, es necesario restringir el conjunto de los códigos que pueden ser considerados. Específicamente, nos vamos a centrar en la optimización de códigos con accesos a posiciones de memoria dados por vectores de indirección. Nótese que esta definición sigue siendo muy amplia, ya que no restringe el número de indirecciones, el tipo de accesos realizados, ni la estructura concreta del código. Vamos a asumir que el contenido de estas indirecciones se desconoce en tiempo de compilación. En caso contrario, nuestras propuestas seguirían siendo válidas, pero tendrían que competir con otras técnicas estáticas de paralelización que potencialmente son más eficientes. Por simplicidad, vamos a asumir que los accesos irregulares se realizan en el cuerpo de un lazo. En cualquier caso, esta es la situación más común de los códigos irregulares que hemos encontrado en la literatura.

Para abordar la paralelización del código irregular, es necesario realizar un análisis de dependencias que permita aplicar una estrategia específica de paralelización. Dos estamentos tienen una dependencia de datos si no pueden ser ejecutados simultáneamente debido a un conflicto en el acceso a una misma variable. En términos generales, y asumiendo que un cierto estamento `S1` es ejecutado antes que otro estamento `S2`, se pueden producir tres tipos de dependencias de datos.

1. **Dependencias verdaderas o de flujo:** se producen cuando `S2` lee una posición de memoria que es escrita por `S1`.
2. **Dependencias de salida:** se producen cuando `S1` y `S2` escriben sobre la misma posición de memoria.
3. **Falsas dependencias o antidependencias:** surgen cuando `S2` escribe una posición de memoria que es leída por `S1`.

Como se discutirá con posterioridad, las estrategias de paralelización que podemos aplicar van a depender de la estructura concreta que presente el código irregular. Por este motivo, es necesario realizar un proceso de clasificación que permita caracterizar las distintas estructuras de código irregular. En este sentido, existen distintos criterios de clasificación de códigos irregulares. Atendiendo a los objetivos planteados, en nuestro caso es necesario aplicar un esquema de clasificación basado en la compatibilidad con las distintas técnicas de optimización propuestas. El esquema que presentamos se fundamenta

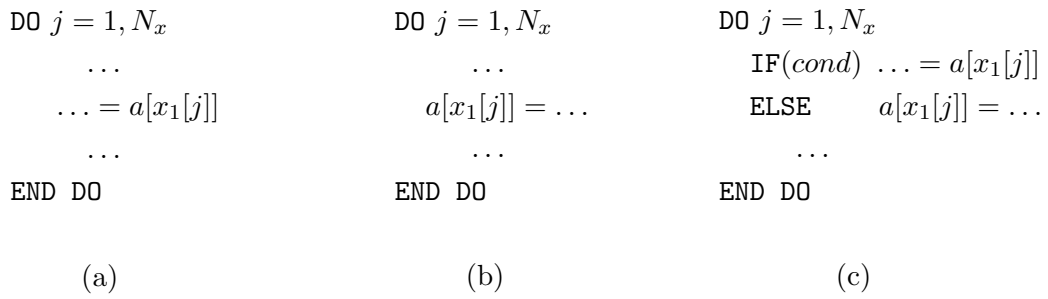


Figura 1.1: Ejemplos de lazos irregulares con una indirección: (a) lectura mediante una indirección, (b) escritura mediante una indirección, (c) acceso genérico de lectura o escritura.

en un único parámetro, que es el número de accesos irregulares existentes en el cuerpo del lazo. De este modo, distinguimos los siguientes casos:

- **Lazos con un acceso irregular.** La Figura 1.1 muestra diversos ejemplos de lazos irregulares con esta estructura. En estos ejemplos no hay más accesos sobre a que los indicados. En todos los casos tenemos un lazo de N_x iteraciones en el que se accede al vector a por medio del vector de indirección x_1. En el caso de la Figura 1.1(a) tenemos un ejemplo de accesos mediante lecturas. Como ejemplo de este tipo de estructuras podemos citar rutinas de álgebra matricial dispersa como el producto matriz dispersa vector, y rutinas pertenecientes a programas de prueba como la rutina *integer sort* del NAS *benchmark*. Asumiendo que el acceso irregular es la única fuente de dependencias, tendremos que el código de la Figura 1.1(a) es totalmente paralelo. En el caso del ejemplo de la Figura 1.1(b), tendremos la existencia de riesgo de dependencias de salidas asociadas al acceso irregular. Ejemplos de aplicaciones con esta estructura se pueden encontrar, entre otros, en rutinas de álgebra matricial. Una situación más arbitraria se ilustra en la Figura 1.1(c), para la cual, en función del valor lógico de *cond* se pueden realizar accesos tanto de lectura como de escritura. En el caso de que la variable *cond* tome valores distintos a lo largo de la ejecución del lazo, tendremos la posibilidad de existencia de cualquier tipo de dependencias entre iteraciones distintas.

- **Lazos con varios accesos irregulares.** Otro tipo de códigos irregulares es el mostrado en la Figura 1.2(a). En esta figura se ilustra un código irregular con N_{stmt} indirecciones que realizan accesos arbitrarios tanto de lectura como de escritura.

```
DO j = 1, N_x
    ...
  a[x_1[j]] = ...
  ... = a[x_2[j]]
  a[x_{N_stmt}[j]] = ...
    ...
END DO
```

(a)

```
DO j = 1, N_x
    ...
  a[x_1[j]] = a[x_1[j]] ⊗ ...
  a[x_2[j]] = a[x_2[j]] ⊗ ...
  a[x_{N_stmt}[j]] = a[x_{N_stmt}[j]] ⊗ ...
    ...
END DO
```

(b)

Figura 1.2: Ejemplos de lazo irregulares con varias indirecciones: (a) asignación de escritura y lectura, (b) asignación mediante operaciones de reducción.

En función del contenido de estos vectores de indirección existirá cualquier tipo de dependencias de datos, no sólo entre iteraciones distintas, sino también dentro de una misma iteración. El análisis de esta clase de lazos es más complejo que el del apartado anterior, dado que es necesario estudiar la interacción entre vectores de indirección diferentes. Nuevamente tendremos casos particulares: no habrá dependencias si todos accesos irregulares son de lectura, o bien sólo tendremos dependencias de salida en el caso de que sean únicamente operaciones de escritura.

Un caso particular de lazos con varios accesos son las **operaciones de reducción**, las cuales aparecen tan frecuentemente en códigos científicos que merecen un tratamiento especial. Este tipo de operación se define del siguiente modo:

Definición 1.4.1 Un estamento de la forma $a = a \otimes \ldots$ es una operación de reducción si $\otimes$ es un operador asociativo y conmutativo. □

Definición 1.4.2 Denominamos **reducción escalar** a las operaciones de reducción de la forma $a = a \otimes \ldots$ en donde a es un escalar. Por otra banda, denotamos **reducción de histograma** a aquellas con estructura $a[i] = a[i] \otimes \ldots$ donde a es una matriz. □

Un ejemplo de lazo irregular con operaciones de reducción se muestra en la Figura 1.2(b). Considerando este tipo de lazos, es necesario establecer requisitos adicionales para garantizar que un reordenamiento de sus iteraciones origine el mismo resultado que el lazo original. Concretamente, es necesario establecer dos nuevas condiciones:

- Todas las operaciones de reducción deben utilizar el mismo operador. En caso contrario, aunque cada sentencia individual sea una reducción, no lo es el conjunto de accesos realizados en cada iteración, dado que no tienen por qué cumplir ni la propiedad asociativa ni la conmutativa.
- Aquellas variables sobre las que se realiza la operación de reducción no pueden ser ni leídas ni escritas por otra operación distinta a la de una reducción. La aparición de alguno de estos tipos de operaciones hace que el conjunto de accesos no sea conmutativo.

En la siguiente sección describimos los distintos objetivos que nos hemos marcado para la realización de esta tesis. Adicionalmente, introducimos las principales contribuciones de nuestro trabajo.

1.5 Contribución y organización de esta tesis

El objetivo fundamental de esta tesis es el desarrollo de nuevas estrategias de optimización de códigos irregulares en computadores de alto rendimiento. Específicamente, hemos particularizando nuestro estudio a sistemas multiprocesadores de memoria compartida. En este contexto, las distintas propuestas realizadas están encaminadas a superar las dos principales limitaciones existentes: la extracción del máximo paralelismo existente en el código y la explotación de la jerarquía de memoria del sistema paralelo. El fin último de nuestro trabajo es la integración de las distintas propuestas realizadas en una herramienta de compilación y paralelización automática.

La revisión del trabajo previo desarrollado en este campo nos permitió definir de forma más clara la estructura de nuestra propuesta. Basándonos en esta revisión, establecimos una serie de requisitos que perfilan nuestra contribución.

1. La compleja estructura de indireccionamiento existente en los códigos irregulares hace que su análisis en tiempo de compilación resulte generalmente inviable. Por este motivo, las distintas propuestas realizadas deben estar diseñadas para operar durante la ejecución del programa.
2. El empleo de estrategias de paralelización no especulativas permite analizar, con una mayor precisión, las características del programa considerado. Una mayor precisión en el análisis implica, potencialmente, una mayor capacidad de optimización por parte de nuestra herramienta. Por este motivo, hemos adoptado la estrategia basada en el paradigma del inspector-ejecutor [33].

3. Debido a que el proceso de optimización se realiza en tiempo de ejecución, el coste asociado al inspector supone un factor crítico en la eficiencia de una herramienta de optimización. Dicho coste debe ser minimizado permitiendo, además, el reuso del inspector tanto en el uso recurrente de una misma estrategia, como en su aplicación a distintos tipos de estrategias de optimización.

4. Debido a la gran variedad de códigos irregulares existentes, es necesario establecer un sistema de clasificación que permita determinar las estrategias más eficientes para cada clase de código.

5. Es necesario realizar un análisis comparativo con el resto de propuestas existentes. Dicho análisis debe permitir establecer criterios para la elección de la técnica más eficiente en cada problema particular.

Partiendo de estos requisitos hemos desarrollado un conjunto de propuestas para la optimización de códigos irregulares. La Figura 1.3 ilustra un esquema general de nuestra contribución. Nuestras propuestas abarcan distintos tópicos, como son la caracterización, el análisis de dependencias, la paralelización y la mejora en la localidad de este tipo de códigos. Cada uno de estos tópicos es introducido a continuación de un modo más preciso, mientras que en capítulos posteriores se abordan y validan en gran detalle.

- **Caracterización del patrón de acceso a memoria.**
  El primer paso realizado fue el desarrollo de una nueva estrategia destinada a la reducción de la complejidad de la etapa de inspección. Típicamente, esta etapa realiza un análisis del patrón de acceso a memoria del código irregular sujeto a estudio. Combinando la información obtenida con las características estructurales del programa (las cuales son suministradas por el compilador), el inspector determina las dependencias existentes, y realiza los distintos tipos de reestructuraciones necesarias para aumentar la eficiencia del programa. Por norma general, la rutina de inspección tiene asociada un coste computacional significativo, el cual debe minimizarse. Nuestra propuesta está encaminada a alcanzar este objetivo mediante la segmentación del inspector en dos etapas. La primera es la encargada de llevar a cabo la caracterización y análisis del patrón de acceso a memoria asociado a un vector de indirección. Esto se consigue mediante un nuevo esquema de caracterización denominado **representación IARD**. Dicha representación tiene un bajo coste en su construcción y en la memoria que precisa. Esta representación es empleada por la segunda etapa del inspector, que realiza el proceso particular de optimización del código irregular. Debido a su enorme versatilidad, la representación IARD puede ser empleada por un gran número de procesos de optimización.

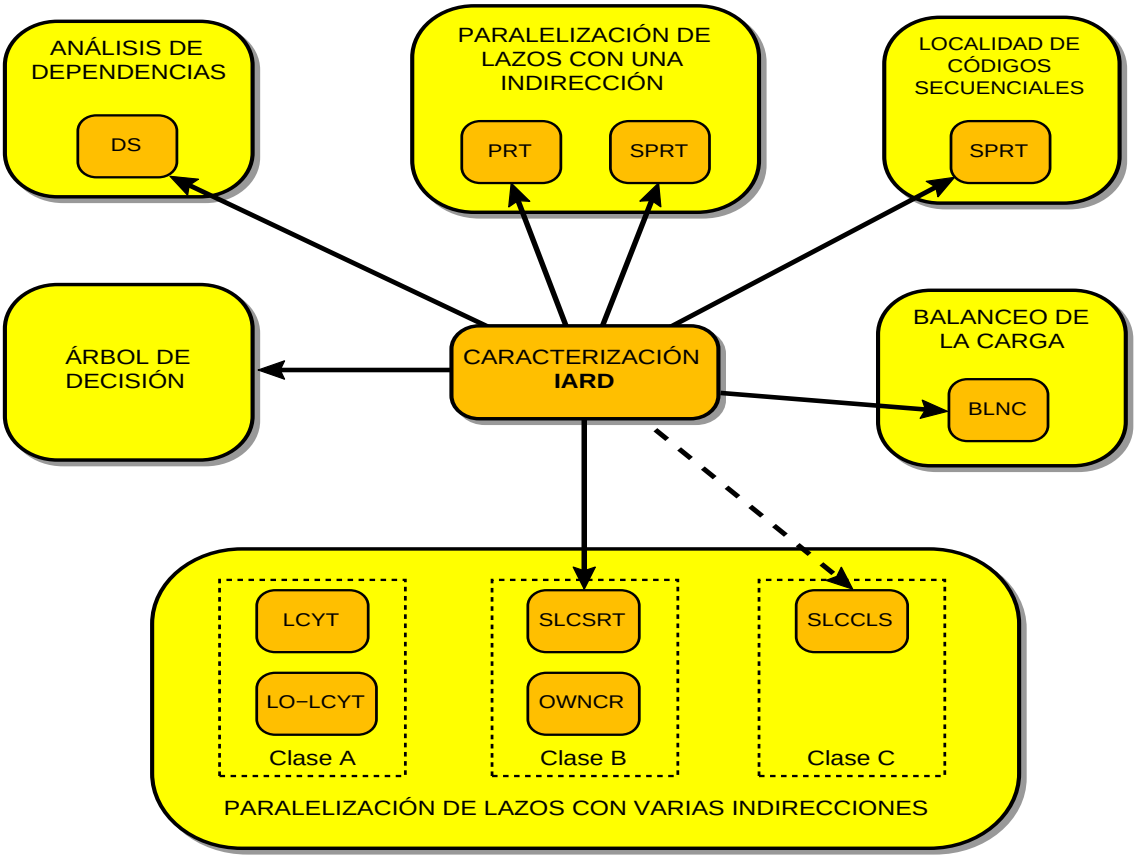


Figura 1.3: Estructura general de nuestra contribución.

Hemos desarrollado distintos procedimientos de análisis y optimización que hacen uso de esta representación, la cual les permite reducir de forma significativa su coste computacional. Adicionalmente, hemos dotado a esta caracterización de mecanismos de transformación que permiten adaptarla, con un bajo coste, a diversos esquemas de indireccionamiento. Las ventajas de esta nueva estrategia son diversas: por una parte permite realizar de forma desacoplada parte de la etapa de inspección, lo cual otorga a nuestra propuesta de una mayor flexibilidad. En el caso de que una indirección se emplee por distintas secciones del código, los inspectores asociados a cada una de ellas pueden acceder a la misma caracterización reduciendo, de este modo, el coste global del proceso. Finalmente, permite su reuso en el caso de que una sección de código se ejecute múltiples veces sin cambiar su patrón de acceso, pero que, debido a la naturaleza del sistema, el inspector deba ser múltiplemente aplicado.

- **Análisis de dependencias.**
  Hemos desarrollado una nueva estrategia de análisis de dependencias basada en el acceso a la representación IARD. Dicha estrategia, denominada ***algoritmo de División en Secciones lineales*** (DS), determina la existencia de accesos a posiciones de memoria comunes entre dos secciones de código irregular. Empleando los resultados de este algoritmo se pueden detectar dependencias de datos entre ambas regiones y, consiguientemente, determinar si estas pueden ser ejecutadas de forma concurrente.

- **Paralelización de lazos irregulares con una indirección.**
  La caracterización IARD también fue aplicada de forma exitosa para la paralelización de estos tipos de códigos. En este contexto introducimos dos nuevas técnicas de paralelización denominadas ***Private Region Technique*** (PRT) y ***Sorted Private Region Technique*** (SPRT). Ambas propuestas realizan un reordenamiento de los vectores de indirección, para obtener, a lo largo de la ejecución paralela, una alta localidad en los accesos, tanto de lectura como de escritura. Ambas propuestas presentan la propiedad de preservar el mismo orden en los accesos que el existente en el código original. Esto permite aplicarlas no sólo a códigos con reducciones irregulares[1], sino también a cualquier tipo de operación de lectura, escritura, y accesos combinados como los mostrados en el ejemplo de la Figura 1.1(c).

- **Mejora en la localidad de códigos secuenciales.**
  La importante mejora en la localidad obtenida con la estrategia SPRT nos impulsó a adaptar su empleo sobre códigos secuenciales. Mediante una modificación del esquema de distribución de esta estrategia, conseguimos explotar eficientemente la jerarquía de memoria del sistema obteniendo un aumento significativo de la eficiencia en la ejecución secuencial del programa.

- **Balanceo de la carga.**
  El siguiente paso realizado en esta tesis consistió en el diseño de nuevas estrategias de balanceo de carga computacional que pueden ser aplicadas sobre distintas estrategias de paralelización. En este marco se sitúa el **algoritmo Balanceo** (BLNC). Este algoritmo emplea la caracterización IARD para realizar un reparto eficiente de la carga computacional, y puede ser aplicado tanto a las técnicas de paralelización propuestas por otros autores, como a las estrategias PRT y SPRT.

[1] Notar que en una operación de reducción realiza un acceso de lectura y de escritura empleando la misma indirección. Sin embargo, y debido a que ambas posiciones de memoria coinciden, este tipo de operación puede clasificarse dentro de esta categoría.

- **Paralelización de lazos con varias indirecciones.**
  Dada la gran variedad de estructuras irregulares existentes, no existe una única estrategia de optimización que permita abordar eficientemente todas las situaciones. Por este motivo, realizamos una clasificación de las estructuras irregulares que más frecuentemente podemos encontrar, y para cada una de estas clases desarrollamos distintas estrategias de optimización. Específicamente, hemos considerado tres tipos de estructuras: lazos con accesos genéricos (tanto lecturas como escrituras) para los que los estamentos que conforman el cuerpo del lazo no pueden ser ejecutados fuera del orden (que denotamos como clase A), lazos con accesos genéricos cuyos estamentos pueden ser ejecutados fuera del orden (denotados como clase B) y lazos con reducciones irregulares (clase C).

  - **Clase A:** en este contexto presentamos dos nuevas estrategias de paralelización, denominadas ***Local* CYT** (`LCYT`) y ***Low Overhead Local* CYT** (`LO-LCYT`) que explotan tanto el paralelismo existente en el lazo irregular como la localidad en los accesos. Mediante ambas propuestas se consigue un equilibro entre el coste del proceso de inspección y el rendimiento del código paralelo. De forma conjunta, hemos establecido mecanismos de decisión que permiten determinar la estrategia más eficiente para cada problema concreto.
  - **Clase B:** los códigos irregulares pertenecientes a esta categoría pueden ser abordados utilizando otro tipo de técnicas de optimización. Hemos desarrollado dos estrategias, denominadas ***Owner Compute Rule*** (`OWNCR`) y ***Slice Sort*** (`SLCSRT`) que permiten obtener mayores grados de paralelismo y localidad. En particular, la estrategia `SLCSRT` se basa en un reordenamiento de los vectores de indirección e incluye una técnica propia para el balanceo de la carga computacional.
  - **Clase C:** en esta categoría explotamos la propiedad conmutativa y asociativa de la operación de reducción para llevar a cabo la optimización de estos códigos mediante un cambio en el orden de los accesos a memoria. En este contexto presentamos la estrategia ***Slice Classification*** (`SLCCLS`), que obtiene resultados competitivos en un amplio espectro de situaciones.

  En esta clase de códigos, la aplicación de la representación IARD resulta mucho más compleja. Para algunos casos, y debido a la naturaleza de su estructura interna, el empleo de la representación IARD resulta inabordable. Este el caso de las estrategias `LCYT` y `LO-LCYT`. En otros casos, como en la `SLCCLS`, hemos establecido las bases para un uso futuro de esta representación por parte del inspector, lo cual implicaría una

disminución significativa en su coste. Por este motivo empleamos, en la Figura 1.3, una línea discontinua para establecer vínculos entre ambas propuestas.

- **Construcción de un árbol de decisión**
  En el contexto del diseño de una herramienta de paralelización automática es requisito imprescindible disponer de un esquema de organización que permita determinar, para cada tipo de problema, la técnica más eficiente. Con este fin, y basándonos en las distintas pruebas comparativas realizadas a lo largo de esta memoria, hemos establecido un árbol de decisión. Dicho árbol tiene en cuenta un gran número de factores como son la naturaleza del código irregular y las características del patrón de acceso. Nuevamente, hemos establecido mecanismos para que dicha etapa utilice la información recogida en la representación IARD.

Todas estas propuestas, con la excepción de la estrategia `OWNCR`, utilizan el esquema de paralelización basado en el inspector-ejecutor. Cada una de ellas incorpora algoritmos específicos tanto de inspección, como de ejecución, adaptados a las características del código considerado.

La estructura de esta memoria es la siguiente: en el Capítulo 2 se introduce el proceso de creación de la caracterización IARD; en el Capítulo 3 se muestra la aplicación de esta caracterización a un nuevo método de análisis de dependencias; el Capítulo 4 aborda otras aplicaciones inmediatas de esta representación, las cuales comprenden la optimización de códigos con una indirección y el balanceo de la carga. En el Capítulo 5 se aborda la paralelización de códigos con varias indirecciones asociadas a accesos tanto de lectura como de escritura. El Capítulo 6 presenta nuestra propuesta de paralelización de códigos con varias operaciones de reducción irregular y varias indirecciones. Las distintas validaciones de eficiencia de todas nuestras propuestas se utilizan para establecer, en el Capítulo 7, el árbol de decisión que permite determinar el tipo de estrategia más adecuada para cada problema concreto. Finalmente, se enumeran las principales conclusiones derivadas de esta tesis.

Capítulo 2

Caracterización del patrón de acceso a memoria

El desarrollo de procedimientos de caracterización de los accesos a memoria es un factor crítico en el diseño de una herramienta de paralelización automática. En el caso de los códigos irregulares, esta caracterización debe realizarse en tiempo de ejecución, por lo que su coste computacional debe tenerse en cuenta a la hora de evaluar el rendimiento del código paralelo. De este modo, en el marco de nuestro trabajo de investigación, el primer objetivo que nos hemos planteado es el desarrollo de una técnica de caracterización aplicada a códigos irregulares. Antes de comenzar su desarrollo, hemos establecido diversas metas que debemos alcanzar: nuestra propuesta debe tener asociado un bajo coste computacional, tiene que poder ser aplicada sobre códigos irregulares lo más genéricos posibles, debe ser lo suficientemente flexible como para permitir realizar un análisis de dependencias y, con el fin de reducir el coste del proceso de análisis, también debe poder ser aplicada en otras técnicas de paralelización y optimización.

En este capítulo introducimos una nueva técnica de caracterización del conjunto de posiciones de memoria accedidas en la ejecución del código irregular. Mediante el empleo de dicha caracterización podemos alcanzar, de un modo eficiente, cada uno de las metas anteriores. Este capítulo se corresponde, dentro del esquema general mostrado en la Figura 1.3, al bloque denominado Caracterización IARD. El trabajo realizado en el mismo ha contribuido al desarrollo de las siguientes publicaciones: "The envelope of a digital curve based on dominant points" presentada en la *International Conference of Discrete Geometry for Computer Imagery* en diciembre del 2000 [127] y "Run-time Characterization of Irregular Accesses Applied to Parallelization of Irregular Reductions" presentada en el

Workshop on High Performance Scientific and Engineering computing with Applications in conjunction with the International Conference on Parallel Processing en septiembre del 2001 [128].

Comenzamos el capítulo por una descripción general de nuestra propuesta y por una revisión bibliográfica de trabajos relacionados. Posteriormente introducimos la estructura del proceso de caracterización y realizamos un estudio de su eficiencia. En los capítulos 3 y 4 describimos aplicaciones de dicha caracterización, las cuales incluyen técnicas de análisis de dependencias, de paralelización y de optimización de códigos irregulares. Finalmente, en los capítulos 5 y 6, generalizaremos el empleo de nuestra representación a un mayor número de situaciones.

2.1 Introducción

El proceso de análisis de un código irregular implica la determinación durante la ejecución del mismo, del conjunto de valores tomados en cada una de las variables que lo conforman. En base a esta información, se puede obtener el flujo de datos del programa, los distintos tipos de dependencias de datos existentes, y el grado de paralelismo que presenta. En este contexto resulta de gran utilidad el empleo, por parte de las distintas rutinas de análisis, de una caracterización que represente el conjunto de posiciones de memoria accedidas durante la ejecución del programa. Las ventajas de esta estrategia son diversas: en caso de existir distintos procedimientos (análisis de dependencias, extracción de paralelismo, etc.), y mediante el empleo de una misma representación, no es necesario evaluar múltiples veces el conjunto de posiciones de memoria accedidas. Adicionalmente, tampoco es necesario realizar un análisis exhaustivo del flujo de datos del programa, lo cual se traduce en una reducción importante tanto del coste computacional del proceso de análisis, como de sus requisitos de almacenamiento.

En este trabajo presentamos un esquema de caracterización del conjunto de posiciones de memoria accedidas por un código irregular. Adicionalmente, nuestra propuesta incluye procedimientos para la detección y almacenamiento de las dependencias de datos asociados a los accesos irregulares. Inicialmente, y con el fin de simplificar el planteamiento del problema, vamos a considerar códigos irregulares con un único vector de indirección. Posteriormente evaluaremos la capacidad de extensión de nuestra caracterización a situaciones más complejas.

Con el fin de contextualizar nuestra propuesta, la Figura 2.1 muestra ejemplos del tipo de códigos irregulares considerados en este trabajo. Podemos apreciar que el conjunto de

```
S1: DO j = 1, N1
        a[x[j]] = ...
    END DO

S2: DO j = 1, N2
        a[2 * x[j] + 10] = ...
    END DO

S3: DO j = 1, N3
        a[x[2 * j]] = ...
    END DO
```

Figura 2.1: Ejemplos de accesos por medio de una indirección.

accesos depende tanto de los valores almacenados en la indirección como del modo en que se accede a la misma. Un concepto clave es el patrón de acceso a memoria, que definimos a continuación.

Definición 2.1.1 Denotamos por **patrón de acceso a memoria de un código irregular** al conjunto ordenado de accesos realizados por dicho código a través de sus indirecciones. □

De este modo, en el ejemplo de la Figura 2.1, podemos apreciar que aunque el vector de indirección es el mismo en todos los lazos, el patrón de acceso a memoria de cada uno es diferente.

A la hora de realizar una caracterización del flujo de datos, existen dos alternativas: o bien se realiza una caracterización independiente para cada uno de los lazos, o bien se hace una caracterización única que permita ser aplicada (es decir, transformada) para cada situación particular. Las ventajas de esta última alternativa son diversas, por una parte el coste del proceso de análisis se ve reducido, dado que únicamente es necesario caracterizar una única indirección (frente a la caracterización de cada uno de los patrones de acceso). Por otra parte, asumiendo una representación convenientemente compacta y eficiente, la particularización del patrón de accesos para cada situación puede hacerse con un coste mínimo empleando la caracterización. Como principal desventaja hay que destacar que, debido a que nuestra caracterización no contiene información exacta de los accesos realizados, puede existir una pérdida en la precisión cuando se particulariza a situaciones concretas.

En base a estas consideraciones, hemos optado por la segunda alternativa. Es decir, en vez de caracterizar el patrón de acceso de cada sección particular del código, vamos a

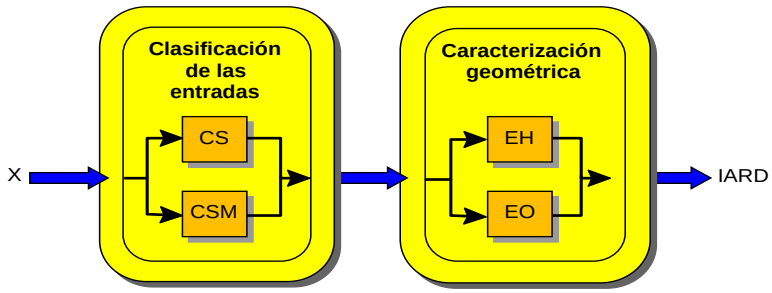


Figura 2.2: Esquema de caracterización de una indirección.

realizar una única caracterización del patrón de indirección. La definición de este nuevo concepto es la siguiente.

Definición 2.1.2 Denotamos por **patrón de acceso a memoria de una indirección** o por **patrón de indirección** al conjunto ordenado de accesos realizados por la indirección. El orden de estos accesos es el originado al recorrer todas las entradas de la indirección de forma ordenada, empezando por aquella con un índice menor. □

Algunas ventajas de esta elección son la posibilidad de dedicar más recursos al proceso de caracterización (dado que hay que realizarlo menos veces), la obtención de un resultado más general (dado que debe ser adaptado a cada situación) y el desarrollo de una notación más portable que puede ser usada en un mayor número de aplicaciones.

Denominamos *Irregular Access Region Descriptor* (IARD) a nuestra propuesta de caracterización de patrones de indirección. Esta representación debe construirse en tiempo de ejecución, una vez conocido el contenido del vector de indirección analizado. La Figura 2.2 muestra las dos etapas que conforman la estructura de nuestra propuesta. La primera de ellas, descrita en la Sección 2.3, clasifica las entradas del vector de indirección en conjuntos denominados *slices*. La representación de estos *slices* en un espacio bidimensional conforma la representación geométrica del patrón de indirección. Una segunda etapa, introducida en la Sección 2.4, analiza dicha representación geométrica, extrayendo sus principales características topológicas. El objetivo de esta segunda etapa es la reducción de los costes de almacenamiento mediante una representación reducida del patrón de indirección que debe mantener sus principales características geométricas. Hemos desarrollado dos algoritmos en cada una de estas secciones. Así pues, en la etapa de clasificación presentamos los algoritmos `CS` y `CSM`, mientras que en la etapa de caracterización geométrica tenemos los algoritmos `EH` y `EO`. Las características de cada uno de ellos, y el criterio de elección empleado se comentarán a lo largo de este capítulo.

De este modo, cada uno de los vectores de indirección existentes en el código irregular son caracterizados mediante una representación IARD. Posteriormente, en la Sección 2.5, se describe el proceso de obtención del patrón de acceso a memoria de una sección de código a partir de la representación IARD y de cierta información contextual.

Una vez planteada la estrategia que vamos a seguir en nuestra propuesta, es preciso especificar restricciones y marcar objetivos para formalizarla. Estos se enumeran a continuación.

- Los costes de generación y almacenamiento de la caracterización deben ser lo más reducidos posible.
- La exactitud de las distintas etapas del proceso de análisis será mayor cuanto más precisa sea la caracterización del patrón de acceso a memoria. Sin embargo, un aumento en su precisión implica, por norma general, un aumento del coste del procesamiento y almacenamiento. Existen, por lo tanto, dos características opuestas (precisión de la representación frente a velocidad de procesamiento de la misma) que deben ser convenientemente equilibradas.
- La representación debe clasificar los accesos de acuerdo con el orden de acceso impuesto por la indirección. Es decir, la representación debe realizarse sobre el patrón de indirección. Consideremos el ejemplo de la Figura 2.1. Asumiendo que N_1 es el número de elementos del vector de indirección, el patrón de acceso asociado al lazo `S1` se corresponde al patrón de indirección, por lo que este lazo está directamente caracterizado por la representación IARD de x. Sin embargo, deben definirse reglas que permitan transformar y aplicar dicha representación a códigos cuyos accesos no son sobre elementos consecutivos. Este es el caso de los lazos `S2` y `S3`.
- El coste del proceso de transformación de nuestra representación a otros esquemas de acceso debe ser el menor posible. Esta restricción implica realizar dicha transformación utilizando únicamente la información contenida en la representación, y sin tener que analizar de nuevo el contenido del vector de indirección.

2.2 Trabajo previo

Esta revisión bibliográfica se centra en dos tópicos que, a pesar de pertenecer a disciplinas distintas, son aunados en nuestra propuesta. Por una parte, en la Sección 2.2.1 estudiamos las distintas técnicas de caracterización de los patrones de acceso existentes en

```
DO i = 1, 3
  DO j = 1, 10
    a[20 * i + j] = ...
  END DO
END DO
```

Figura 2.3: Ejemplo de lazo regular doblemente anidado.

la literatura. En esta revisión encontramos trabajos e ideas que contribuyeron al desarrollo de nuestra propuesta. En la Sección 2.2.2 profundizamos en las técnicas de caracterización geométrica de curvas digitales, y analizamos su viabilidad en su aplicación a nuestra representación.

2.2.1 Caracterización de los patrones de acceso

Es este trabajo hemos intentado establecer una analogía con otros trabajos que realizan la caracterización de los accesos a memoria de códigos regulares. Por este motivo, vamos a remontar nuestra revisión bibliográfica a los primeros trabajos en los que se abordó este tópico. A diferencia de los códigos irregulares, el flujo de datos de un código regular puede conocerse en tiempo de compilación, permitiendo realizar en esta etapa la caracterización del patrón de acceso. Uno de los primeros trabajos en el que se aborda la caracterización de los accesos a memoria es el de Triolet, et al. [137]. En él se proponen técnicas de caracterización en los accesos a matrices con el fin de realizar un análisis interprocedural de dependencias. Posteriormente, la precisión de la representación y del análisis de dependencias fue mejorada en herramientas de paralelización automática como el PIPS [29] y SUIF [3, 56]. Esta última herramienta adoptó los *Regular Section Descriptors* [63] para representar las regiones de acceso a una matriz.

Otros ejemplo de representación son el *Data Access Descriptor* (DAD) propuesto por Balasundaram y Kennedy [8], y la notación de tripletes propuesta por Tu y Padua [138, 139]. Esta última notación caracteriza los accesos sobre una matriz de forma exacta para un gran número de situaciones. Sin embargo, cuando el mecanismo de acceso a la matriz es complejo, la notación de tripletes pierde precisión, haciendo más inexacto (y por tanto, ineficiente) el análisis de dependencias.

Existen otras representaciones matemáticas para expresar el patrón de acceso a memoria asociado a un lazo regular. Ejemplos de las mismas son las caracterizaciones mediante familias de hiperplanos [72] o expresiones matriciales [5]. En [49] se introduce una nueva

representación que permite contemplar estructuras de código más complejas, como pueden ser las sentencias condicionales. En [116] se aborda la caracterización de códigos regulares con accesos complejos, como son los realizados a través de múltiples variables de inducción o múltiples funciones lineales del índice del lazo.

Ejemplos de aplicaciones basadas en caracterizaciones del patrón de acceso los tenemos en [146], donde es aplicada para realizar transformaciones en la estructura del código con el fin de maximizar el paralelismo y la localidad en los accesos, y en [85], donde se aplican modelos de caracterización del patrón de acceso para paralelizar códigos regulares y optimizar las comunicaciones.

En este trabajo nos hemos inspirado en una mejora de esta última propuesta denominada *Linear Memory Access Descriptor* (LMAD). Esta caracterización fue inicialmente introducida en [104] y posteriormente mejorada en [67, 105, 106] y describe, para un código concreto, la secuencia de accesos realizados sobre una determinada matriz. Típicamente la caracterización LMAD está destinada a lazos con múltiple nivel de anidamiento, para los que el acceso a la matriz viene determinado como una combinación lineal de los índices de dichos lazos. El descriptor LMAD almacena, para cada índice, tres características denominadas *stride*, *span* y *offset*. El *stride* se define como la separación entre entradas de la matriz accedidas de modo consecutivo, el *span* representa la diferencia entre la primera y última entrada accedida, y finalmente, el *offset* se define como la mínima posición de la matriz accedida durante la ejecución del código considerado. La notación LMAD, para un lazo genérico de d índices, es del siguiente modo:

$$\mathcal{LMAD}^{\delta_1,\delta_2,\ldots\delta_d}_{\sigma_1,\sigma_2,\ldots\sigma_d} + \tau \tag{2.1}$$

En donde la dupla $\{\delta_i, \sigma_i\}$ hace referencia, respectivamente, al *stride* y *span* del i-ésimo índice, y τ representa el *offset*. Un ejemplo de su uso se puede ver en la Figura 2.3, en el que la matriz a es accedida mediante una combinación lineal de los dos índices del lazo. Para este ejemplo, la representación LMAD resultante tiene la siguiente forma:

$$\mathcal{LMAD}^{1,60}_{10,20} + 0 \tag{2.2}$$

En donde $\{\delta_1, \sigma_1\} = \{1, 10\}$ está asociado al índice j y $\{\delta_2, \sigma_2\} = \{60, 20\}$ al índice i. Dado que este último índice está multiplicado por una constante, el valor del *stride* y del *span* son adecuadamente modificados. Otras situaciones más complejas, como por ejemplo lazos triangulares, o lazos que contienen expresiones no afines, también pueden ser caracterizadas mediante esta representación. Adicionalmente, se han definido reglas que permiten determinar si dos representaciones diferentes de la misma matriz acceden a elementos comunes. De este modo es posible determinar la existencia de dependencias

entre las mismas. Este análisis se realiza empleando únicamente la representación LMAD, lo cual hace que el coste del proceso sea reducido.

La caracterización LMAD fue diseñada para realizar la paralelización automática de un código regular. Para llevar a cabo esta tarea es necesario obtener información adicional sobre el contexto en el que se realiza el acceso. Como ejemplo de dicha información se puede citar el tipo de operación que se realiza (lectura, escritura, reducción), la dimensión de la matriz, la monotonía en los accesos, etc. La estructura resultante tras añadir esta información adicional a la caracterización LMAD se denomina *Access Region Descriptor (ARD)*. Utilizando esta estructura se pueden realizar de modo eficiente diversas operaciones sobre el programa. Ejemplos de dichas operaciones son el desarrollo de un procedimiento de análisis interprocedural de dependencias denominado *Access Region Test (ART)* [68] y la técnica presentada por Navarro, et al. [101] que permite realizar una distribución de las iteraciones y de los datos en un entorno de ejecución paralela, maximizando la localidad en los accesos a memoria y minimizando el coste de las comunicaciones.

La caracterización ARD explota la regularidad en los accesos mostrada por ciertos códigos, realizando una descripción de los mismos con un bajo coste y un alto grado de precisión. Los códigos irregulares que consideramos no verifican esta propiedad, dado que la indirección puede tomar valores arbitrarios que no guardan relación alguna entre sí. Este hecho, unido a que el análisis del flujo de datos debe realizarse en tiempo de ejecución, hace que las propuestas anteriores no puedan ser generalizadas a códigos irregulares.

En el contexto de la paralelización automática de reducciones irregulares, Yu y Rauchwerger [152] proponen un esquema de caracterización del patrón de acceso. Este esquema está destinado a aportar información acerca del entorno de ejecución del código irregular. Posteriormente, en [151] los mismos autores enriquecen su caracterización con nuevos parámetros que permiten extraer ciertas propiedades del patrón de acceso. Mediante esta información los autores establecen un árbol de decisión que les permite determinar la técnica más eficiente para cada tipo de escenario. La caracterización realizada por estos autores resulta adecuada para la elaboración de un árbol de decisión. Sin embargo, consideramos que no es suficiente para poder realizar una caracterización precisa del patrón de acceso que se ajuste a nuestros requisitos.

En [15, 16] se propone una estrategia para la caracterización de patrones asociados a matrices. Dicha estrategia permite detectar y caracterizar distintas estructuras bandeadas, con un agrupamiento en bloques y patrones de acceso compuestos por regiones densas. En [142] se presenta otra propuesta para el análisis de patrones de acceso orientada a la reestructuración del código en función de las características de dicho patrón. En [64, 65]

se desarrolla un modelado analítico de la localidad asociada a la ejecución de códigos irregulares. El tipo de modelo que se plantea introduce medidas del grado de localidad en los accesos irregulares tanto en sistemas monoprocesador como multiprocesador. En base a este modelo se desarrollan procesos de optimización de localidad por métodos heurísticos que originan una mayor eficiencia en la gestión de los niveles de la jerarquía de memoria.

En todos estos trabajos, no encontramos una solución adecuada para los objetivos que hemos planteado. Esto se debe a diversos motivos: por una parte, las caracterizaciones no permiten ser utilizadas de forma simple por distintas técnicas (detección de dependencias, paralelización, balanceo de la carga, etc.) que hemos desarrollado. Por otro lado, no son lo suficientemente flexibles como para transformar la representación a otros esquemas de acceso. Finalmente, en algunos casos, su coste computacional resulta demasiado elevado para su empleo eficiente durante la ejecución del programa.

El proceso de clasificación que proponemos está inspirado en los primeros trabajos de paralelización de lazos irregulares parcialmente paralelos [154, 98]. Este tipo de lazos se caracterizan por presentar dependencias verdaderas entre iteraciones, siendo necesaria una adecuada política de ejecución paralela, que debe mantener la corrección del resultado. En este trabajo se propone una clasificación de las iteraciones del lazo en conjuntos que pueden ser ejecutados concurrentemente y en un orden arbitrario sin originar dependencias de datos. En lo concerniente a nuestra propuesta, este es el concepto básico que hemos adoptado para realizar nuestra representación. En vez de considerar una clasificación por iteraciones del lazo, clasificamos entradas del vector de indirección en conjuntos que presentan una serie de propiedades. Dichas propiedades se describen de forma detallada en la Sección 2.3, en la que también describimos la implementación concreta de esta técnica. Cabe destacar que el algoritmo de clasificación que hemos desarrollado no tiene relación con el realizado en [98, 154], salvo por el hecho de que estos trabajos nos han servido como un punto de partida desde el que poder desarrollar nuestra representación.

2.2.2 Caracterización geométrica del patrón de acceso

La segunda etapa de nuestra propuesta consiste en la caracterización geométrica del patrón de acceso de una indirección. Partiendo de una representación bidimensional de las entradas del vector de indirección (obtenida en la primera etapa), deseamos extraer las principales características geométricas de la misma.

Como veremos, nuestro objetivo puede ser interpretado como una caracterización de una curva digital sujeta a una serie de restricciones, o bien puede clasificarse como un proceso de optimización sobre un polígono de un gran número de elementos. De cada uno

de estos grupos existe un gran número de trabajos de los que hemos seleccionado los más representativos y afines a nuestros objetivos.

Así pues, en la primera familia existen distintas heurísticas que permiten hacer, bajo una serie de restricciones, una aproximación a un contorno por medio de un conjunto de segmentos. Todas estas técnicas son costosas en términos computacionales, con una complejidad que típicamente varía entre $\mathcal{O}(n^2)$ y $\mathcal{O}(n^3)$, siendo n el número de elementos de la curva. Un estudio comparativo de diferentes propuesta puede verse en el trabajo presentado por Rosin [121]. Existen otras propuestas con una complejidad menor y, consiguientemente, con un menor tiempo de procesamiento. Como contrapartida, estas propuestas no aseguran la obtención de una solución óptima. Ejemplo de la mismas es [6], en donde la curva digital es aproximada por una cadena poligonal, utilizando como criterio la minimización del área comprendida entre la curva original y la aproximada. Una propuesta alternativa es la realizada por Debled y Reveilles [35], en la cual se emplean líneas *navie* para realizar la segmentación de la curva digital. En [28] se presenta un método de complejidad lineal para la aproximación poligonal de una curva digital dada. Todas estas propuestas utilizan como criterio de calidad la similitud y proximidad con la curva original. Sin embargo, nosotros imponemos como restricción que la curva poligonal acote a la original, restricción que no es exigida en ninguna de las propuestas anteriores.

Dentro del segundo grupo, el problema de la obtención del mínimo polígono convexo contenido en otro polígono convexo fue eficientemente resuelto por Aggarwal y Park [2]. En este trabajo, los autores aplican técnicas de optimización basadas en búsqueda de matrices monótonas. Otros trabajos en este área son, entre otros, la obtención del mínimo triángulo que contiene a un polígono convexo [103], del mínimo polígono inscrito de k lados [1], del mínimo polígono equiangular con k lados [36], del mínimo polígono inscrito con ángulos especificados [100], y del mínimo paralelogramo circunscrito [40]. En [21], Boyce presenta un método en el que, dados n puntos, se encuentra el menor polígono de k lados con vértices en los puntos y que circunscribe a la curva digital. Eppstein et al [39], presentan un método similar, de naturaleza iterativa. En él, en función del resultado obtenido para un polígono con $k-1$ lados, se obtiene un nuevo polígono de k lados con menor área. La aplicación a nuestro problema se puede realizar asumiendo que nuestra curva está compuesta por una serie de n polígonos. Sin embargo, dada la alta complejidad de estas técnicas y el valor elevado que tendría n en nuestro caso, su eficiencia resulta extremadamente pobre.

Como resultado de esta revisión bibliográfica no hemos encontrado ningún trabajo que se adapte completamente a todos los requisitos que hemos impuesto. Dentro del primer grupo, las técnicas más afines no aseguran una aproximación por exceso y defecto de la

curva original, lo cual resulta inaceptable para nuestros fines. Las propuestas del segundo grupo, tampoco cumplen esta meta, dado que o bien son poco flexibles, o bien están más ideadas para curvas poligonales y no para una secuencia de puntos muy próximos en el espacio. Este hecho, nos ha motivado al desarrollo de algoritmos propios que resuelvan la problemática planteada de un modo eficiente.

2.3 Clasificación de las entradas de una indirección

Como punto de partida de este trabajo, hemos estudiado las propiedades existentes en diversos patrones de indirección procedentes de aplicaciones reales. Estos patrones de acceso fueron extraídos de librerías de matrices dispersas como la *Harwell Boeing sparse matrix collection* [37] o la *University of Florida Sparse Matrix Collection* [34]. Una segunda fuente de procedencia de estos patrones son aplicaciones de simulación por elementos o diferencias finitas [45] y algoritmos de simulación de n-cuerpos [108].

En todos los casos considerados, pudimos constatar la existencia de ciertos niveles de localidad en los accesos a memoria dados por las indirecciones. Esta localidad se manifiesta en que entradas consecutivas acceden, en mayor o menor grado, sobre posiciones próximas de memoria. Nuestra propuesta explota esta propiedad agrupando las entradas de la indirección en conjuntos. De este modo, en lugar de caracterizar individualmente cada entrada del vector de indirección, se considera la región de acceso de cada conjunto. Si el grado de localidad es suficientemente elevado, la región de acceso asociada a las entradas de cada conjunto podrá ser caracterizada con gran precisión. Como se verá en la Sección 2.5.3, este hecho nos va a permitir reducir considerablemente los costes de análisis y procesamiento de la indirección irregular.

Tal y como se comentó con anterioridad, vamos a realizar la caracterización del patrón de indirección que, en el caso particular del lazo ilustrado en la Figura 2.4, coincide con su patrón de acceso. Vamos a denotar por x al vector de indirección y por a al vector accedido. Denominamos N_x y N_a a sus respectivas dimensiones. Inicialmente nos vamos a restringir a un código como el de la figura en el que los accesos sobre a son operaciones de escritura y suponen la única fuente posible de dependencias. Como punto de partida, nuestra propuesta analiza los elementos del vector del indirección y los agrupa en conjuntos denominados *slices*. Una definición formal de los mismos es la siguiente.

Definición 2.3.1 Un ***slice*** se define como cualquier serie de entradas consecutivas del vector de indirección cuyos accesos a memoria son diferentes. Más formalmente, denotado S_k al k-ésimo

```
DO j = 1, N_x
    a[x[j]] = ...
END DO
```

Figura 2.4: Ejemplo de lazo considerado.

slice, se verifica lo siguiente:

$$S_k = [i_k, i_{k+1}), \qquad 1 \leq k \leq N_S \tag{2.3}$$

Tal que,

$$\forall \ i, j \in S_k, \ \ i \neq j, \quad \longrightarrow \quad x[i] \neq x[j] \tag{2.4}$$

En donde N_S es el número total de *slices*, $i_1 = 1$ y $i_{N_S+1} = N_x + 1$. □

Una representación por *slices* consiste en una clasificación de las entradas de la indirección. La definición anterior nos indica qué propiedades deben tener las entradas pertenecientes al mismo *slice*. Sin embargo, es necesario especificar la relación entre las entradas asociadas a *slices* diferentes. Una propiedad importante que debe verificar esta representación es la de preservar el orden de accesos de la indirección. Esta restricción se refleja en la siguiente definición.

Definición 2.3.2 Una **representación incremental por *slices*** de una indirección ha de cumplir las siguientes condiciones:

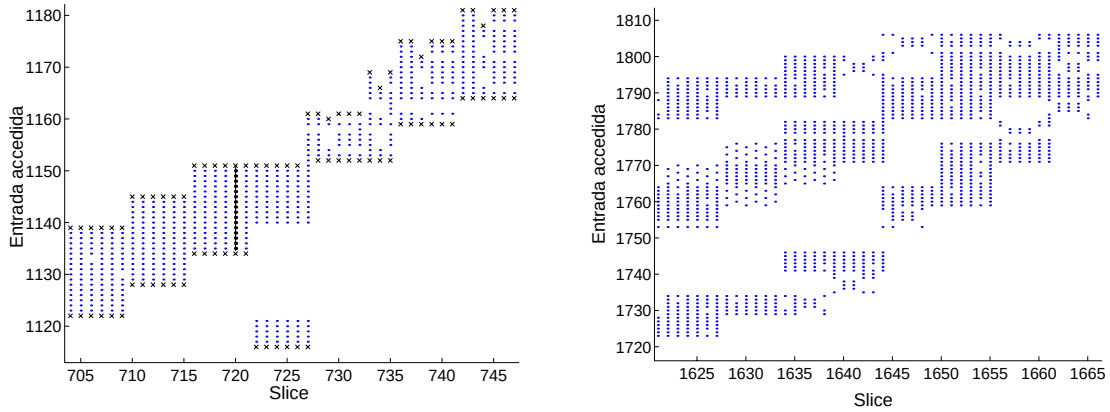
1. Sea $i \in S_k$ y $j \in S_{k'}$ dos entradas de la indirección pertenecientes a *slices* diferentes. Si $k < k'$ entonces debe verificar que $i < j$.
2. Sea S_k un *slice* genérico, se debe verificar que $i_k < i_{k+1}$. Es decir, cada *slice* debe contener al menos una entrada del vector de indirección.

□

En base a esta definición podemos enunciar la siguiente propiedad.

Propiedad 2.3.1 Para una representación incremental de una indirección con N_S *slices* y N_x entradas, se verifica que $1 \leq N_S \leq N_x$.

DEMOSTRACIÓN: La demostración de esta propiedad está implícita en la Definición 2.3.2. De acuerdo con su segundo apartado, cada *slice* contiene al menos una entrada del vector de indirección, mientras que el número máximo de entradas por *slice* está acotado superiormente por el número de entradas de la indirección. Por otra parte, es necesario como mínimo un *slice* para realizar la clasificación. De este modo se debe verificar que $1 \leq N_S \leq N_x$. □



(a) (b)

Figura 2.5: Representación por *slices*.

La Figura 2.5 muestra ejemplos de esta representación para dos patrones de acceso. Cada columna de las gráficas representa el conjunto de elementos pertenecientes a cada *slice*. El valor de cada elemento (representado en el eje de ordenadas) se corresponde con el índice de la entrada de a accedida. A modo de ejemplo, en la Figura 2.5(a) los elementos de la indirección pertenecientes al *slice* 720 (marcados en negrita en la figura) realizan accesos sobre el intervalo $[1134, 1151]$ de entradas de a.

Mediante esta representación, se agrupan y caracterizan los accesos del patrón de indirección. En la Figura 2.5, se puede apreciar la existencia de localidad en los accesos realizados en cada *slice*. Experimentalmente, hemos podido comprobar que esta situación se da para un gran número de patrones de acceso reales. Adicionalmente, en el caso de patrones que no verifiquen esta propiedad, y mediante el empleo de técnicas de reordenamiento [70, 31, 94], se puede conseguir patrones de acceso con mayor localidad.

2.3.1 Algoritmo de clasificación por *slices*

Retornando a la línea argumental iniciada en este capítulo, nuestro objetivo consiste en la explotación de la localidad en los patrones de acceso, intentando reducir el coste de la representación. De este modo, para cada *slice* únicamente vamos a almacenar tres magnitudes. Estas son, el número de entradas que tiene asociado, la posición máxima, y la posición mínima de a accedidas en el mismo. Vamos a representar estas tres magnitudes mediante tres vectores que denotaremos, respectivamente, como ρ, u y l. Cada entrada de los mismos contendrá los valores asociados a cada *slice*. Una definición precisa de estas magnitudes es la siguiente.

Definición 2.3.3 Los vectores ρ, u y l tienen N_S elementos, y se definen del siguiente modo:

$$u[k] = max(x[j] \quad \forall j \in S_k) \tag{2.5}$$

$$l[k] = min(x[j] \quad \forall j \in S_k) \tag{2.6}$$

$$\rho[k] = i_{k+1} - i_k \tag{2.7}$$

En donde las funciones max y min obtienen, respectivamente, el valor máximo y mínimo del conjunto de valores que verifican las condición impuesta en el argumento. □

La Figura 2.5(a) denota, mediante el símbolo "×", los valores almacenados en u y l para cada *slice*. Se puede observar que estas magnitudes marcan una cota superior e inferior del patrón de acceso. Mediante esta simplificación, los accesos concretos realizados en cada *slice* son desconocidos, lo cual introduce cierta incertidumbre en nuestra representación. Sin embargo, y como veremos con posterioridad, esta información es suficiente para abordar de un modo eficiente la caracterización del patrón de acceso de una indirección.

El pseudocódigo del algoritmo de clasificación por *slices* (`CS`) se ilustra en la Figura 2.6. El algoritmo emplea un vector denominado $flag$, con N_a elementos. Este vector es utilizado para almacenar el *slice* asociado al último acceso sobre cada entrada de a, y al comienzo del algoritmo este vector se inicializa a un valor nulo. Un factor clave de esta propuesta es que el vector de indirección se procesa ordenadamente, de modo que las entradas son clasificadas respetando el orden existente en la indirección. Cada vez que se analiza una de estas entradas, se comprueba el valor asociado en el vector $flag$. Si este valor es inferior al del *slice* actual (dado por la variable s), la entrada no va a producir conflictos dentro del *slice* considerado, verificando la condición de la Ecuación 2.4 y almacenando su valor en un *buffer* local. Este *buffer* contiene todas las posiciones de memoria accedidas dentro del *slice* considerado.

La Figura 2.7 muestra un ejemplo de esta clasificación para un vector de indirección dado por $x = \{2, 1, 2, 3, 3, 4, 6, 3, 1\}$ para $N_x = 9$ y $N_a = 6$. La Figura 2.7(a) muestra el contenido del vector $flag$ conforme se va procesando la indirección. Una entrada vacía del vector $flag$ significa que tiene un valor nulo. Cuando la segunda entrada es procesada, el valor almacenado en el vector $flag$ es inferior al del *slice* actual (dado por la variable s), lo que hace que dicha entrada pase a almacenar el valor del *slice* actual.

En el caso de que el valor de la entrada del vector $flag$ sea el mismo que el del *slice* actual, se realiza un aumento del contador s. De este modo, la entrada considerada pasa a pertenecer al siguiente *slice*. En el ejemplo considerado, este hecho ocurre cuando se procesa la tercera entrada del vector de indirección. El valor almacenado $flag[2]$ es 1, coincidiendo con el valor de la variable s. Para esta situación, el algoritmo `CS` actualiza,

```
Algoritmo CS
entrada
        x: vector de indirección
salida
        {u⃗, l⃗, ρ⃗}: conjunto de cotas superiores, cotas inferiores y densidades
inicio del algoritmo
         s = 1
         flag[1 : N_a] = 0
         buffer = ∅
         DO  j = 1, N_x
             IF (flag[x[j]] = s)
                 {u[s], l[s], ρ[s]} ⟸ caracteriza_slice(buffer)
                 buffer = ∅
                 s = s + 1
             END IF
             buffer ⟵ x[j]
             flag[x[j]] = s
         END DO
fin del algoritmo
```

Figura 2.6: Algoritmo de clasificación por *slices* (CS).

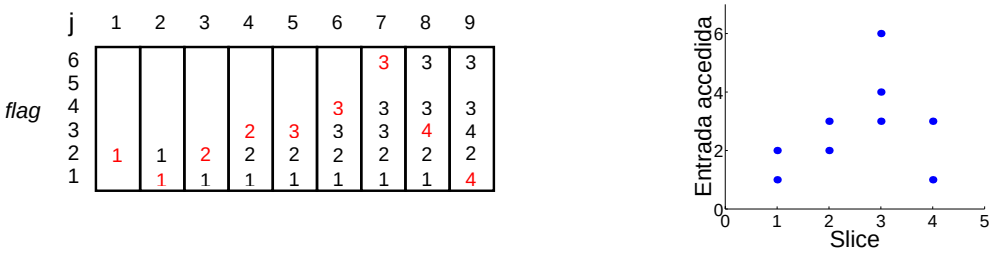
mediante la función *caracteriza_slice*, las entradas correspondientes a los vectores ρ, u y l. El funcionamiento de esta función es simple: únicamente debe analizar los accesos almacenados en el *buffer*, obteniendo el valor de acceso máximo, mínimo y el número de entradas almacenadas en dicho *buffer*.

En el ejemplo de la Figura 2.7, el número de *slices* producidos es 4. El valor de los vectores resultantes es: $u = \{2, 3, 6, 3\}$, $l = \{1, 2, 3, 1\}$, $\rho = \{2, 2, 3, 2\}$. La Figura 2.7(b) muestra el patrón de acceso obtenido mediante nuestra representación. En base a la estructura del algoritmo propuesto, podemos enunciar la siguiente propiedad.

Propiedad 2.3.2 La representación realizada por el algoritmo CS verifica la siguiente relación:

$$\exists\, i \in S_k,\ j \in S_{k-1} \,/\, x[i] = x[j] \quad \forall k \in (1, N_S] \tag{2.8}$$

DEMOSTRACIÓN: Dada la estructura de este algoritmo, el cambio de *slice* únicamente se produce cuando la entrada de la indirección considerada realiza un acceso sobre una posición previamente accedida por otra entrada del mismo *slice*. De este modo, la relación enunciada debe cumplirse



(a) (b)

Figura 2.7: Ejemplo del funcionamiento del algoritmo `CS`: (a) evolución del vector $flag$, (b) representación del patrón de indirección.

```
DO j = 1, Nx
    a[x[j]] = a[x[j]] ⊗ ...
END DO
```

(a)

```
DO i = 1, NS
    DO j = col[i], col[i+1] − 1
        a[row[j]] = a[row[j]] + b[i] ∗ val[j]
    END DO
END DO
```

(b)

Figura 2.8: Ejemplos de códigos irregulares: (a) operación con un vector de indirección, (b) operación sobre una matriz dispersa.

para al menos una entrada de cada *slice*. □

De ahora en adelante, denominaremos al vector ρ como **vector densidad**. De este modo, y en una primera aproximación, la caracterización del vector de indirección requiere $3N_S$ elementos. De acuerdo con la Propiedad 2.3.1, a lo sumo nuestra caracterización tendrá un coste de almacenamiento de $3N_x$ entradas, lo cual resulta prohibitivo. Sin embargo, como se podrá ver con posterioridad, habitualmente se verifica que $N_S \ll N_x$, por lo que el coste de almacenamiento real será significativamente inferior al máximo coste teórico.

2.3.2 Algoritmo de clasificación por *slices* modificado

Los códigos irregulares que frecuentemente aparecen en librerías matemáticas, como *Sparse Kit* [122] o BLAS [17, 27], pueden ser agrupados en dos categorías: los que

únicamente contienen vectores de indirección, y los que operan sobre matrices dispersas. La Figura 2.8(a) muestra una estructura irregular perteneciente a la primera categoría. Ejemplos de esta categoría son las reducciones irregulares o las rutinas de permutación. En ambos casos el contenido del lazo es un acceso irregular por medio de un vector de indirección. Un ejemplo del segundo grupo es la rutina que realiza el producto matriz-dispersa vector de la Figura 2.8(b). Asumiendo un formato de almacenamiento por columnas [13] (formato CCS), el patrón de acceso de la matriz dispersa hace uso de los vectores *val*, *col* y *row*. El vector *val* contiene los valores numéricos asociados a cada valor no nulo de la matriz, el vector *row* almacena el número de fila de cada uno de los elementos no nulos, y el vector *col* almacena el índice de la primera entrada de cada columna. Para códigos irregulares pertenecientes a esta última categoría, resulta mucho más útil hacer uso de una nueva representación de los accesos a memoria en la que cada *slice* se corresponde con una columna de la matriz dispersa. Con el fin de adaptar nuestra representación a este nuevo formato, es necesario realizar modificaciones al algoritmo de clasificación por *slices* comentado en la sección anterior. Este nuevo algoritmo, denominado *clasificación por* slices *modificado* (`CSM`), se ilustra en la Figura 2.9. Se ha introducido un nuevo criterio de clasificación: en vez de utilizar el vector $flag$ para detectar los accesos repetidos sobre la misma posición de memoria, ahora identificamos el primer elemento de cada columna de la matriz dispersa (línea etiquetada como `L1`), el cual determina el cambio de *slice*.

Dado que el formato CCS no permite dos accesos en la misma posición (mismo valor del vector *row* dentro de la misma columna), entonces esta nueva representación cumple el requisito impuesto por la Definición 2.3.1. La única diferencia con el resultado obtenido en el algoritmo original es que el uso de esta nueva propuesta da lugar a un mayor número de *slices*, dado que ya no se verifica la Propiedad 2.3.2.

2.3.3 Resultados experimentales

Hemos evaluado la eficiencia de los algoritmos `CS` y `CSM` con vectores de indirección obtenidos de la librería Harwell-Boeing [37]. Esta librería contiene un conjunto de matrices dispersas extraídas de diversas aplicaciones. Como punto de partida, hemos utilizado el formato de almacenamiento *Column Compress Storage* (CCS). De este modo, la matriz queda representada por tres vectores denominados *col*, *val* y *row*. De estos vectores, hemos tomado al vector *row* como vector de indirección. Este vector aparece habitualmente en códigos científicos irregulares como vector de indirección, lo que nos permite evaluar nuestra propuesta con patrones de acceso reales.

La Tabla 2.1 muestra las principales características de los patrones de acceso y el

```
Algoritmo CSM
entrada
        x: vector de indirección
        col: índice de columna
salida
        {u, l, ρ}: conjunto de cotas superiores, cotas inferiores y densidades
inicio del algoritmo
        s = 1
        buffer = ∅
         DO  j = 1, N_x
L1           IF (j = col[s + 1])
                 {u[s], l[s], ρ[s]} ⟸ caracteriza_slice(buffer)
                 s = s + 1
                 buffer = ∅
             END IF
            buffer ⟵ x[j]
         END DO
fin del algoritmo
```

Figura 2.9: Algoritmo de clasificación por *slices* modificado (`CSM`).

espacio de almacenamiento requerido por el resultado del algoritmo de clasificación. Los parámetros mostrados en la tabla son los siguientes:

- N_a: tamaño mínimo del vector a (ver Figura 2.4). Este valor representa el menor número de elementos del vector a para poder ser correctamente accedido por medio de la indirección. Más formalmente, esta cantidad se define como $N_a = max(x) - min(x) + 1$, donde las funciones max y min devuelven, respectivamente, el elemento máximo y mínimo almacenados en x.
- N_x: número de entradas del vector de indirección.
- N_S: número de *slices* obtenidos mediante el algoritmo de clasificación.
- ΔM_{slices}: cantidad de memoria requerida para almacenar el resultado del algoritmo de clasificación (vectores u, l y ρ).

Se puede apreciar como el coste de almacenamiento de ambas caracterizaciones es

mucho menor que el coste de almacenamiento del vector de indirección (N_x). Nótese que la diferencia es de hasta dos órdenes de magnitud. También se destaca que la diferencia en el número de *slices* obtenido por ambas propuestas es reducida. La máxima reducción del número de *slices* se obtiene con la matriz *nasasrb* (15% de reducción empleando el algoritmo `CS`), mientras que para la matriz *s3dkq4m2* esta reducción es prácticamente nula. Esta diferencia en el comportamiento del algoritmo `CS` se debe a la distribución particular de las entradas en el patrón de acceso, que favorecen en mayor o menor medida la reducción en el número de *slices*.

Los resultados obtenidos con estas indirecciones corroboran la existencia de regiones con alta localidad en los accesos. En la siguiente sección presentamos propuestas para la simplificación de nuestra caracterización. Mediante esta simplificación podemos reducir aún más su coste de almacenamiento.

2.4 Caracterización geométrica de la indirección

El objetivo planteado es el desarrollo de un algoritmo que permita simplificar la caracterización del patrón de acceso hecha por los algoritmos `CS` y `CSM`. Vamos a considerar dicho patrón de acceso como una representación geométrica en un espacio bidimensional, centrándonos únicamente en la representación espacial de los vectores u y l. El proceso de caracterización está sujeto a una serie restricciones que enumeramos a continuación.

- Deseamos que la nueva representación acote el patrón de accesos de la indirección. De este modo, deseamos obtener una aproximación conservadora de los accesos, la cual nos va a permitir realizar un análisis preciso y fiable del patrón de indirección. En términos geométricos, esta restricción se traduce en que el resultado debe representar una nueva secuencia de puntos que se aproxime por exceso (en caso del vector u), o

Matriz	N_a	N_x	`CS`		`CSM`	
			N_S	ΔM_{slices}	N_S	ΔM_{slices}
3dtube	45332	**3213618**	43029	**129087**	45330	**135990**
bcsstk14	1806	**63453**	1665	**4995**	1806	**5418**
bcsstk17	10973	**428650**	9466	**28398**	10973	**32919**
bcsstk29	13992	**619488**	12989	**51956**	13994	**41982**
nasasrb	54872	**2677324**	47476	**142426**	54870	**164610**
s3dkq4m2	90451	**4820892**	90299	**270894**	90449	**271347**

Tabla 2.1: Eficiencia de los algoritmos `CS` y `CSM`.

por defecto (en caso del vector l), a la secuencia original.

- Exigimos que el resultado sea lo más preciso posible. Es decir, la secuencia caracterizada debe ser lo más parecida (y próxima) posible a la original.

- El resultado de la caracterización debe tener el menor coste de procesamiento posible. Esto se traduce en una representación simple y con un menor número de elementos que la secuencia original.

Vamos a asociar a cada entrada de los vectores u y l un punto en el espacio. Específicamente, la k-ésima entrada del vector u tendrá asociada el punto espacial de coordenadas $\{k, u[k]\}$. Realizando un planteamiento análogo con el vector l obtendremos las dos secuencias de puntos discretas que deseamos caracterizar.

La organización de esta sección es la siguiente: en Sección 2.4.1 establecemos en primer lugar las bases matemáticas de nuestro proceso de caracterización geométrica. Posteriormente, en las secciones 2.4.1 y 2.4.1 presentamos dos propuestas distintas que realizan esta tarea. Finalmente, en la Sección 2.4.2 realizamos el análisis comparativo del rendimiento de cada una de ellas.

2.4.1 Algoritmo de caracterización geométrica

Nuestra solución se basa en la aproximación de los vectores u y l a una curva *Freeman chain-code* que denotamos por $\mathcal{C}$. Una vez construida esta curva, buscamos obtener una nueva curva que conforme la envolvente a $\mathcal{C}$ y que sea lo más próxima posible. A continuación, introducimos una seria de conceptos que resultarán de gran utilidad para el planteamiento y desarrollo de nuestra propuesta.

Definición 2.4.1 Definimos una curva digital $\mathcal{C}$ como ***Freeman chain-code*** a la dada por una secuencia de $N_{\mathcal{C}}$ puntos con coordenadas cartesianas enteras. Más formalmente, esta curva se define por la siguiente expresión:

$$\mathcal{C} = \{p_j = (p_{x_j}, p_{y_j}),\ j = 1, ...N_{\mathcal{C}}\}, \quad p_{x_j} \in \mathbb{Z}, \quad p_{y_j} \in \mathbb{Z} \tag{2.9}$$

Adicionalmente estos puntos deben verificar que:

1. El punto p_{j+1} es un vecino de p_j, $\forall j < N_{\mathcal{C}}$, y $p_{N_{\mathcal{C}}}$ es vecino de p_1.
2. Dos elementos p_j y p_k son vecinos si $p_j \neq p_k$, y $|p_{x_j} - p_{x_k}| \leq 1$ y $|p_{y_j} - p_{y_k}| \leq 1$.

□

Para que los vectores u y l puedan ser aproximados como curvas de *Freeman chain-code* es necesario añadir puntos intermedios que hagan verificar la condición de vecindad exigida en su definición. Este proceso se realiza mediante la interpolación de entradas consecutivas. Supongamos, por ejemplo, que $N_S = 2$ y $u = \{1, 4\}$. Originalmente, el conjunto de puntos asociado a este intervalo es $\{(1,1),(2,4)\}$, el cual no cumple los requisitos necesarios para ser una curva *Freeman chain-code*. De este modo, es necesario añadir los puntos $(2,2)$ y $(2,3)$, que han sido calculados mediante una interpolación lineal. La curva digital resultante es $\mathcal{C}^u = \{(1,1),(2,2),(2,3),(2,4)\}$. La unión de las curvas $\mathcal{C}^u$ y $\mathcal{C}^l$ origina una curva digital cerrada, que verifica la condición de ser *Freeman chain-code*.

Definición 2.4.2 Denominaremos $\mathcal{E}$ a una secuencia de $N_\mathcal{E}$ puntos que viene especificada por la siguiente relación.

$$\mathcal{E} = \{e_i = (e_{x_i}, e_{y_i}),\ i = 1, ...N_\mathcal{E}\}, \quad e_{x_i} \in \mathbb{Z}, \quad e_{y_i} \in \mathbb{Z} \tag{2.10}$$

Donde $N_\mathcal{E} \ll N_\mathcal{C}$. La curva digital asociada a $\mathcal{E}$, que denominaremos $\mathcal{E}_f$, será aquella formada por la unión mediante segmentos de puntos consecutivos de $\mathcal{E}$. Esta curva está definida como un conjunto de funciones lineales de la siguiente forma.

$$\mathcal{E}_f = \{\{x_i, f_i(x)\},\ e_{x_i} \leq x < e_{x_{i+1}}\ \}\ \ i = 1, ...N_\mathcal{E} - 1 \tag{2.11}$$

Tal que f_i es la recta que une el punto e_i con el punto e_{i+1}. □

Denotamos respectivamente $I(\mathcal{C})$ e $I(\mathcal{E}_f)$ como el conjunto de elementos inscritos dentro de las curvas $\mathcal{C}$ y $\mathcal{E}_f$. La Figura 2.10 muestra un ejemplo de estos conceptos.

Definición 2.4.3 La secuencia de puntos $\mathcal{E}$ se dice una **envolvente** de la curva $\mathcal{C}$ si su curva digital asociada, denotada como $\mathcal{E}_f$, envuelve completamente a la curva $\mathcal{C}$. Es decir, si $I(\mathcal{C}) \subset I(\mathcal{E}_f)$. □

Definición 2.4.4 Dada una secuencia de puntos $\mathcal{E}$ que envuelve a la curva $\mathcal{C}$, definimos como **área de error** a la cardinalidad del conjunto $\parallel I(\mathcal{E}_f) - I(\mathcal{C}) \parallel$. □

El área de error representa el número de elementos existentes entre las curvas $\mathcal{C}$ y $\mathcal{E}_f$. Mientras menor sea ese valor, mejor será la aproximación en la curva envolvente.

En base a este planteamiento, nuestra meta se puede establecer como la resolución del siguiente problema de optimización: dada una curva *Freeman chain-code* $\mathcal{C}$, se busca la secuencia de puntos $\mathcal{E}$ cuya curva digital asociada, denotada como $\mathcal{E}_f$, conforme la envolvente de $\mathcal{C}$ con el menor área de error.

A continuación vamos a describir nuestra propuesta para la resolución de este problema. Como punto de partida, y con el fin de simplificar el proceso de caracterización, vamos a

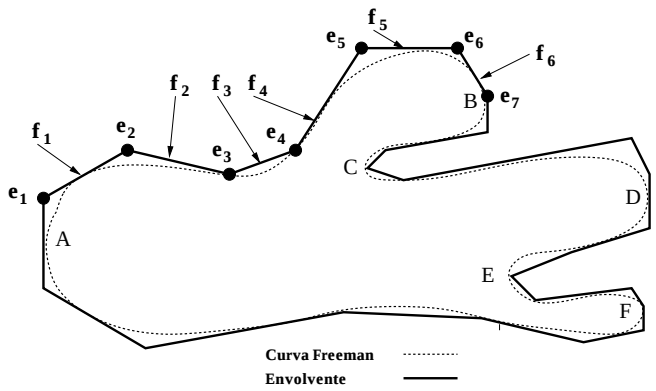


Figura 2.10: Ejemplo de envolvente a una curva.

introducir un nuevo conjunto de puntos denominado conjunto $\mathcal{D}$. De este modo, definimos $\mathcal{D}$ como un subconjunto de $N_{\mathcal{D}}$ puntos representativos de $\mathcal{C}$. Más formalmente:

$$\mathcal{D} = \{d_i = (d_{x_i}, d_{y_i}),\ i = 1, ...N_{\mathcal{D}}\}, \quad d_i \in \mathcal{C} \tag{2.12}$$

Este subconjunto debe ofrecer la máxima información acerca de la forma de la curva $\mathcal{C}$. En [7] se demuestra que los puntos más representativos de una curva coinciden con los que presentan la mayor curvatura. Estos puntos se denominan **puntos dominantes**. Los $N_{\mathcal{D}}$ puntos dominantes más característicos de la curva $\mathcal{C}$ van ser los integrantes del conjunto $\mathcal{D}$. En base a este conjunto construiremos, posteriormente, nuestro conjunto solución $\mathcal{E}$. Nótese que los elementos de $\mathcal{D}$ no tiene por qué formar parte de la envolvente, dado que la curva digital asociada a $\mathcal{D}$ puede intersecar a la curva $\mathcal{C}$.

A continuación vamos a presentar dos técnicas diferentes para la obtención de la envolvente. Ambas realizan modificaciones a la posición de los puntos de $\mathcal{D}$ hasta que su curva digital asociada conforme la envolvente. La primera de nuestras propuestas estima la mejor solución en términos de minimización del área de error (algoritmo `EO`), mientras que la segunda proporciona un método heurístico que, aunque no garantiza el mejor resultado, sí ofrece resultados próximos al óptimo con menores costes computacionales y de almacenamiento de memoria (algoritmo `EH`).

El hecho de que la topología de $\mathcal{C}$ sea arbitraria dificulta enormemente el desarrollo de una estrategia de caracterización que abarque todo el rango de situaciones. Por este motivo, hemos realizado un planteamiento diferente: en vez de abordar todas las situaciones posibles, vamos a dividir $\mathcal{C}$ en tramos que presenten una serie de características. Sobre cada uno de estos tramos, obtendremos la envolvente local al mismo. Cuando todos

los tramos hayan sido procesados, la solución global será el conjunto de contribuciones de cada uno de ellos.

De este modo, vamos a imponer la siguiente restricción a la curva $\mathcal{C}$: los valores de p_{x_j} son monótonos respecto al índice j. Es decir, $p_{x_{j+1}} - p_{x_j} \in \{0, 1\}\ \forall j$, o bien $p_{x_{j+1}} - p_{x_j} \in \{0, -1\}\ \forall j$. Por simplicidad, vamos a considerar únicamente curvas de la primera clase. Dado que ambas aproximaciones son equivalentes, nuestra propuesta es también válida para la segunda clase. En el caso de que la curva digital no verifique esta propiedad, siempre puede ser dividida en tramos que sí lo hagan. En el ejemplo de la Figura 2.10, la curva puede ser particionada en 6 tramos denotados como: AB, BC, CD, DE, EF y FA. Estos intervalos pueden ser analizados de forma independiente, obteniendo, como solución final, la unión de las soluciones parciales de cada uno de los intervalos.

Cálculo y procesamiento de los puntos dominantes

Para realizar la selección de los elementos de $\mathcal{D}$ (es decir, de los puntos dominantes) hemos empleado el algoritmo de Kankanhalli [73]. Este algoritmo es una mejora del algoritmo de Teh-Chin [135] en la que los costes computacionales se reducen de manera significativa. Del mismo modo que en el algoritmo de Teh-Chin, la propuesta de Kankanhalli no requiere ningún parámetro de entrada, obteniendo los punto dominantes en función única de las características topológicas de la curva. El algoritmo realiza un proceso iterativo, en el que en cada paso se obtiene un nuevo punto dominante. El orden de procesamiento de estos puntos se realiza en función de su relevancia[1], obteniendo en aquellas iteraciones los puntos dominantes más representativos. Este hecho nos ha permitido modificar el algoritmo, introduciendo una nueva rutina de finalización. Esta rutina detiene el algoritmo cuando se alcanza un número preestablecido de puntos dominantes, cuyo valor concreto es especificado por el usuario. Dicho número es el único parámetro de entrada de nuestro algoritmo modificado. En caso de no especificarse ningún valor, nuestra propuesta funciona del mismo modo que el algoritmo original.

El siguiente paso a realizar consiste en la extracción de las características geométricas de la curva $\mathcal{C}$ en base al conjunto $\mathcal{D}$. Nuestra propuesta divide la curva digital en tramos comprendidos por elementos consecutivos de $\mathcal{D}$. Cada uno de estos tramos es analizado de forma independiente, simplificando, de este modo, el planteamiento de nuestro problema. En esta sección vamos a describir el proceso de análisis del i-ésimo tramo de $\mathcal{C}$ que

[1]El algoritmo de Kankanhalli clasifica los puntos dominantes en función del grado de curvatura de la región en la que se encuentran. De este modo, se dice que un punto dominante es más relevante que otro si su región de $\mathcal{C}$ asociada tiene una mayor curvatura.

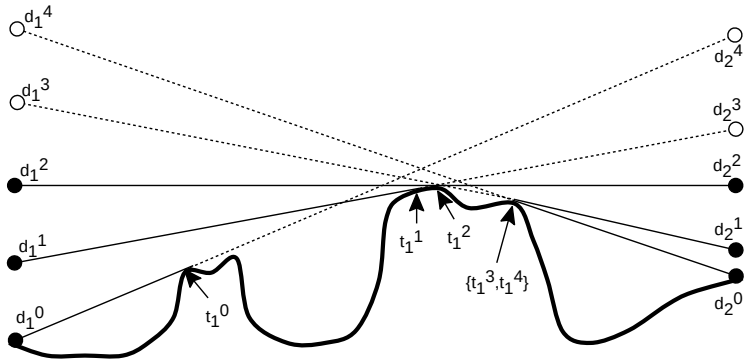


Figura 2.11: Puntos asociados a un segmento de $\mathcal{C}$.

denotaremos como $\mathcal{C}_i$, el cual está comprendido entre los puntos dominantes d_i y d_{i+1}. El conjunto de intervalos $\mathcal{C}_i$ verifica la siguiente relación.

$$\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \ldots \mathcal{C}_{N_\mathcal{D}-1} \tag{2.13}$$

Los algoritmos que proponemos modifican la posición de los puntos dominantes obteniendo, a partir de cada uno, un conjunto de elementos. Con el fin de especificar las propiedades de estos elementos, vamos a introducir en nuestro modelo de representación geométrica una serie de definiciones.

Definición 2.4.5 Definimos el punto d_i^k como una modificación de la posición original del punto $d_i = (d_{x_i}, d_{y_i})$, que guarda la siguiente relación: $d_i^k = (d_{x_i}, d_{y_i} + k)$ con $k \in \mathbb{N}$. □

Definición 2.4.6 Definimos el **punto de tangencia** de un punto d_i^k como el punto perteneciente a la intersección de la curva $\mathcal{C}$ con la recta que parte de d_i^k y que es tangente a $\mathcal{C}$ en el tramo comprendido entre d_i y d_{i+1}. Denotamos a dicho punto de tangencia como t_i^k. □

Definición 2.4.7 Definimos **punto asociado** a un punto d_i^k, y lo denotamos por $d_{i+1}^{k'}$, como el punto d_{i+1}^k más cercano a la recta que une al punto d_i^k con su punto de tangencia t_i^k. □

Un ejemplo de estos puntos se puede ver en la Figura 2.11. En este caso tenemos que el punto asociado a d_1^0 es d_2^4, el de d_1^1 es d_2^3, etc. Notar que el mínimo valor posible de k es 0, mientras que el valor máximo está determinado por el punto asociado d_2^0. Notar además que cuando $k = 0$ el punto está sobre la curva.

Para cada punto de tangencia t_i^k existe un triplete $\{d_i^k, t_i^k, d_{i+1}^{k'}\}$ que define el segmento tangente a $\mathcal{C}$ en el intervalo considerado. Conociendo dos puntos de este triplete (siendo

uno de ellos el punto de tangencia), y mediante la extrapolación de la recta que los une, se puede obtener el tercer elemento del conjunto.

El conjunto de posibles tripletes es organizado en una tabla de datos. Un ejemplo de la misma es la mostrada en la Tabla 2.2. Esta estructura nos permite almacenar la caracterización del tramo de $\mathcal{C}_i$ en términos de segmentos tangentes al mismo. En el caso de la Tabla 2.2, los valores mostrados son los correspondientes a los de la Figura 2.11. Denominaremos a esta estructura de almacenamiento **tabla de tangencia** y se denota por el símbolo TT. Esta tabla está definida para cada tramo $\mathcal{C}_i$, y tiene una columna (denotada como TT^k) por cada valor posible de d_i^k. En cada una de estas columnas se almacena, junto al valor de d_i^k, el valor del punto de tangencia y del punto $d_{i+1}^{k'}$ asociado. Adicionalmente existen dos campos denominados *error* y *enlace* que serán comentados con posterioridad. Esta tabla es la que nos va a permitir elegir los puntos más adecuados para la caracterización de la curva mediante la envolvente. En el siguiente apartado describimos el proceso de creación de una tabla de tangencia.

Determinación de los puntos de tangencia

Antes de introducir la estructura del algoritmo, es necesario enunciar dos propiedades que nos van a permitir obtener los puntos de tangencia de la curva.

Propiedad 2.4.1 Sean p_j y $p_{j'}$ dos puntos de C_i, con $j < j'$. Si el punto p_j es el único punto de tangencia en el tramo comprendido entre d_i y $p_{j'}$, es decir, $t_i^k = p_j \ \ \forall k$, entonces en este tramo el punto $p_{j'}$ es un punto de tangencia $\forall k \geq h$. El parámetro h se obtiene mediante la siguiente ecuación,

$$h = \frac{a^{0\to k} - a^{0\to k'}}{p_{x_{j'}} - p_{x_j}}(p_{x_{j'}} - d_{x_i})(p_{x_j} - d_{x_i}) \tag{2.14}$$

Siendo $a^{0\to j}$ y $a^{0\to j'}$ las pendientes de las rectas que unen, respectivamente, d_i^0 con p_j y d_i^0 con $p_{j'}$.

TT_i^0	TT_i^1	TT_i^2	TT_i^3	TT_i^4
d_i^0	d_i^1	d_i^2	d_i^3	d_i^4
t_i^0	t_i^1	t_i^2	t_i^3	t_i^4
d_{i+1}^4	d_{i+1}^3	d_{i+1}^2	d_{i+1}^1	d_{i+1}^0
$error_i^0$	$error_i^1$	$error_i^2$	$error_i^3$	$error_i^4$
$enlace_i^0$	$enlace_i^1$	$enlace_i^2$	$enlace_i^3$	$enlace_i^4$

Tabla 2.2: Estructura de datos asociada a un segmento de $\mathcal{C}$.

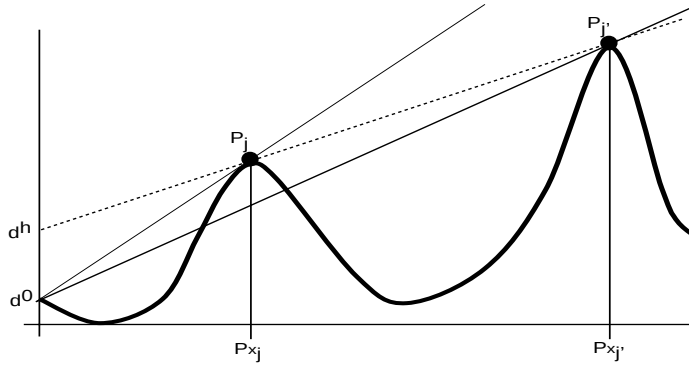


Figura 2.12: Determinación del valor del parámetro h.

DEMOSTRACIÓN: Consideremos la situación general mostrada en la Figura 2.12. En ella se muestra el punto p_j tangente a todos los d_i^k con $k < h$. Para valores de $k \geq h$, el punto asociado a todos los d_i^k pasa a ser el punto p'_j.

Sea $a^{k \to j}$ la pendiente de la recta que une d_i^k con p_j. Entonces, para un valor genérico de k esta pendiente se puede expresar en función de $a^{0 \to j}$ de acuerdo con la siguiente relación.

$$a^{k \to j} = a^{0 \to j} - \frac{k}{p_{x_j} - d_{x_i}} \tag{2.15}$$

Se puede apreciar que el valor h es aquel que verifica que $a^{h \to j} = a^{h \to j'}$. Sustituyendo la Ecuación 2.15 en esta igualdad, la Ecuación 2.14 puede ser fácilmente obtenida. □

Propiedad 2.4.2 Dado un punto $p_j \in \mathcal{C}_i$ y la pendiente $a^{0 \to j}$ asociada con el punto d_i^0. Si el valor de esta pendiente es mayor que la obtenida para los puntos del intervalo con un valor de abscisa menor, es decir, si se verifica la siguiente expresión,

$$a^{0 \to j} > a^{0 \to j'} \ , \quad \forall j' \ / \ p_{x_j} < p_{x_{j'}}, \quad p_{j'} \in \mathcal{C}_i \tag{2.16}$$

Entonces p_j es el punto de tangencia en el tramo de la curva comprendido entre d_i y p_j para todos los elementos d_i^k.

DEMOSTRACIÓN: Para todo $p_{j'}$ se verifica que si $j' < j$ entonces $d_{x_i} \leq p_{x_{j'}} \leq p_{x_j}$. Dado que $a_{j'} < a_j$, de la Ecuación 2.14 se deduce que $h < 0$. Es decir, $p_{j'}$ no puede ser punto tangente, y p_j es el único punto tangente asociado a d_i^k para todo valor de k. □

Nuestra propuesta requiere conocer el punto $p^{max} = (p_x^{max}, p_y^{max}) \in \mathcal{C}_i$, el cual representa el elemento del tramo con mayor valor de ordenada. Este punto puede ser fácilmente

obtenido, sin un coste adicional, en el algoritmo de cálculo de los puntos dominantes. Existe un valor máximo de k (k^{max}) a partir del cual no se obtienen nuevos valores de d_i^k. Este límite viene especificado en la siguiente definición.

Definición 2.4.8 Definimos el valor k^{max} para un intervalo $\mathcal{C}_i$ como aquel valor de k para el que la ordenada del punto d_i coincide con la del punto p^{max}. Más formalmente, k^{max} tiene el valor que verifica la siguiente relación:

$$d_{y_i} + k^{max} = p_y^{max} \tag{2.17}$$

□

Basándonos en estas propiedades hemos desarrollado un algoritmo que obtiene los puntos de tangencia para cada punto d_i^k. Este algoritmo, denominado `PNT_TNG`, se ilustra en la Figura 2.13. Nuestra propuesta analiza individualmente cada intervalo de $\mathcal{C}$ comprendido entre dos puntos dominantes. El análisis de cada segmento $\mathcal{C}_i$ se divide en dos problemas equivalentes. El primero de ellos es la obtención de los puntos de tangencia (t_i^k) para diferentes valores de d_i^k en el segmento de la curva comprendido entre d_i y p^{max}. El segundo problema consiste en la determinación de los valores $t_i^{k'}$ para distintos valores de d_{i+1}^k en el resto del segmento, es decir, en el tramo comprendido entre p^{max} y d_{i+1}. Hay que destacar que, aunque el problema ha sido dividido en dos partes, y dado que ambas mitades son disjuntas, solamente se realiza un único acceso a cada punto de $\mathcal{C}_i$. Si la curva tiene más de un valor con ordenada máxima (en términos geométricos se dice que $\mathcal{C}_i$ presenta dos o más crestas de igual altura), entonces únicamente es necesario considerar los intervalos comprendidos entre d_i con el punto p^{max} de menor abscisa y d_{i+1} con el punto p^{max} de mayor abscisa, siendo irrelevantes el resto de los elementos intermedios del tramo. El algoritmo mostrado en la Figura 2.13 obtiene los puntos de tangencia de $\mathcal{C}_i$ entre los puntos d_i y p^{max}. La resolución del otro intervalo es equivalente, siendo únicamente necesario analizar los puntos comprendidos entre d_{i+1} y p^{max}.

El funcionamiento del algoritmo `PNT_TNG` es el siguiente: inicialmente, el lazo externo (etiquetado como L1 en la figura) recorre todos los puntos del tramo considerado de $\mathcal{C}_i$. En cada uno de ellos, se evalúa la Propiedad 2.4.2. Para realizar esta comprobación únicamente es necesario almacenar, de entre todos los puntos de $\mathcal{C}_i$ procesados, el valor de la pendiente máxima. De acuerdo con esta propiedad, y dado que los puntos del tramo son procesados de forma ordenada, el punto p_j que tenga una pendiente máxima se convierte en el punto de tangencia para todos los posibles valores de k. En la figura esta operación se realiza en el bloque etiquetado como L2.

Si el punto considerado no tiene asociado un valor máximo de pendiente, de acuerdo con la Propiedad 2.4.2, se utiliza la función *incremento* (ver L3) para obtener el valor

```
Algoritmo PNT_TNG
entrada
        {C_i, p^max}: intervalo C_i evaluado entre d_i y p^max
salida
        Lista de puntos {d_i^k, t_i^k}
inicio del algoritmo
            k^max = p_y^max − d_{y_i}
            p_{y_tmp} = −∞
            a^max = −∞

L1          DO  p_j = d_i, d_{i+1}   p_j ∈ C
L2              IF  a^{0→j} ≥ a^max
                        t_i^k = p_j   ∀ k,
                        a^max = a^{0→j}
                        p_{y_tmp} = p_{y_j}
                    ELSE
L3                      h ⟸ ⌈incremento(d_i^0, t_i^0, p_j)⌉
                        temp = t_i^0
                        k = h
                        WHILE k ≤ k^max
L4                          IF  t_i^k = temp
L5                              t_i^k = p_j
                                k = k + 1
                            ELSE
L6                              h ⟸ ⌈incremento(d_i^0, t_i^k, p_j)⌉
                                temp = t_i^k
                                k = h
                            END IF
                        END WHILE
                    END IF
            END DO
fin del algoritmo
```

Figura 2.13: Pseudocódigo del algoritmo PNT_TNG.

```
Algoritmo TBL_TNG
entrada
        C_i: intervalo de C entre d_i y d_{i+1}
        p^max: elemento de C_i con ordenada máxima
salida
        TT_i^k = {d_i^k, t_i^k, d_{i+1}^{k'}}:    Estructura de niveles
inicio del algoritmo
        {d_i^k, t_i^k} <== PNT_TNG(d_i, p^max, C_i)
        {t_i^k, d_{i+1}^{k'}} <== PNT_TNG(p^max, d_{i+1}, C_i)
        TT_i <== extrapola({d_i^k, t_i^k}, {t_i^k, d_{i+1}^{k'}})
fin del algoritmo
```

Figura 2.14: Pseudocódigo del algoritmo `TBL_TNG`.

de k respecto al cual el punto considerado se convierte en un punto de tangencia. Esta función únicamente utiliza la Ecuación 2.14 para obtener h. Posteriormente, en la sección etiquetada como L5, se actualizan los puntos de tangencia asociados a cada punto d_i^k que verifiquen que $h \leq k \leq h^{max}$.

El lazo etiquetado como L6 se emplea para resolver una situación particular. Cuando alguna de las entradas de t_i^k con $k \geq h$ está asociada a otro punto tangente (es decir, existe otro punto tangente entre el p_j considerado y el primer punto de tangencia t_i^0), es necesario evaluar de nuevo la Equación 2.14 entre p_j y este nuevo punto. Consideremos, por ejemplo, la curva de la Figura 2.11. En esta figura las distancias entre dos valores consecutivos de t_i^k han sido exageradas con el propósito de simplificar su apreciación. Supongamos que hemos analizado todos los puntos de la curva en el intervalo $[d_i, t_i^2]$. La pendiente de la línea que une d_i^0 con t_i^2 es menor que la máxima pendiente (asociada al punto t_i^0). Por lo tanto, es necesario evaluar de nuevo la Expresión 2.14 para el conjunto $\{d_i^0, t_i^0, t_i^2\}$, resultando un valor de $h = 1$. Por medio de la condición L4, se comprueba la existencia de otro punto de tangencia entre t_i^0 y t_i^2. En el caso del ejemplo de la figura, $t_1^k \neq tmp$ ya que $t_1^1 \neq t_1^0$. De este modo, es necesario reevaluar la Expresión 2.14 para $\{d_i^0, t_i^1, t_i^2\}$, lo cual devuelve como resultado $h = 2$. A continuación, el índice k toma este nuevo valor y el registro tmp almacena el nuevo punto de tangencia. En el ejemplo de la figura, $k = 2$ y $tmp = t_1^1$. En el caso de encontrar un nuevo punto de tangencia distinto al almacenado en tmp este procedimiento se volvería a repetir.

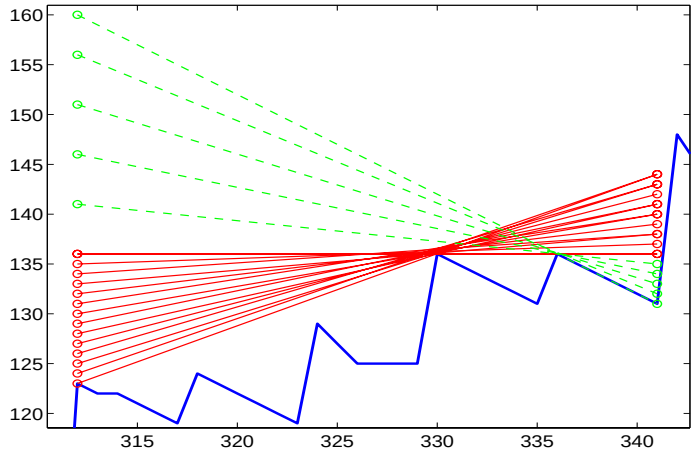


Figura 2.15: Representación gráfica de los campos d_i y d_{i+1} de una tabla de tangencia.

Aplicando el mismo algoritmo al tramo comprendido entre p^{max} y d_{i+1}, y recorriendo los elementos desde d_{i+1} hasta p^{max}, se obtiene la posición de los puntos de tangencia para distintos $d_{i+1}^{k'}$. Del mismo modo que en el ejemplo mostrado en la Figura 2.11, combinando ambos resultados y extrapolando las rectas tangentes, se obtiene la posición de todos los puntos d_i^k y $d_{i+1}^{k'}$ mayores que p^{max}. En esta figura, las líneas discontinuas y los puntos huecos denotan los valores obtenidos por extrapolación.

La Figura 2.14 muestra un esquema del método general para elaborar la tabla de tangencia de un segmento $\mathcal{C}_i$. El algoritmo, denominado `TBL_TNG`, emplea la rutina `PNT_TNG` para procesar cada uno de los subintervalos de $\mathcal{C}_i$. Una vez obtenidos los valores, la función *extrapola* obtiene, mediante la extrapolación de la recta tangente, el resto de los campos de la tabla de tangencia.

La Figura 2.15 muestra un ejemplo real de la información almacenada en la tabla de tangencia. En esta figura, las entradas obtenidas variando el punto d_i se representan con líneas continuas, mientras que se utilizan líneas discontinuas para representar las obtenidas modificando la posición del punto d_{i+1}. Es necesario destacar que los valores obtenidos por extrapolación son todos aquellos con ordenada mayor que $p_{y^{max}} = 136$. Notar que los valores extrapolados no tienen por qué ser distintos. Por ejemplo, en la Figura 2.15 todos los puntos d_i^k con una ordenada en el intervalo $d_{y_i}^k \in [136, 140]$ tienen asociado el mismo punto d_{i+1}. Concretamente, este punto es el que tiene ordenada de valor $d_{y_{i+1}}^k = 136$. De esta manera, únicamente interesará almacenar el punto $d_{y_i}^k = 136$ dado que será el más cercano a la curva.

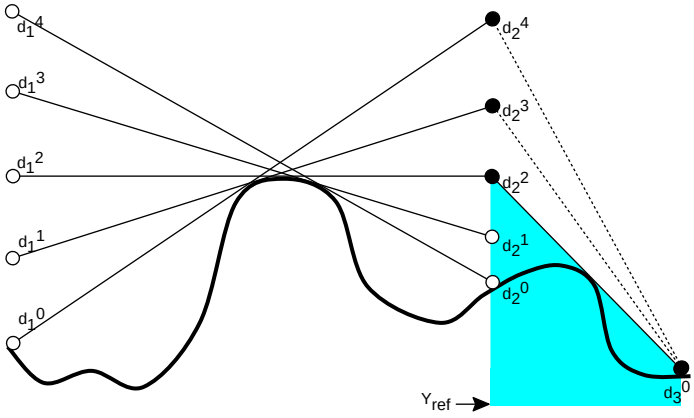


Figura 2.16: Uniones posibles de d_3^0 con d_2.

Un vez descrito el funcionamiento de este algoritmo, vamos a introducir las dos propuestas para la obtención de la envolvente. Como característica destacable hay que indicar que ambas propuestas hacen uso del algoritmo `TBL_TNG` para la generación de las tablas de tangencia, y emplean únicamente su información sin tener que volver a procesar los puntos de $\mathcal{C}$.

Algoritmo de cálculo de la envolvente óptima

Dada una curva digital $\mathcal{C}$ y un subconjunto de elementos $\mathcal{D}$, nuestra propuesta realiza una búsqueda exhaustiva de todas las posibles envolventes hasta obtener aquella que minimiza el área de error. Los elementos que conforman la envolvente óptima son almacenados en el conjunto solución $\mathcal{E}$.

El algoritmo propuesto evalúa, en cada tramo $\mathcal{C}_i$, todos los posibles segmentos tangentes al mismo. Para cada uno de estos segmentos se analizan las posibles conexiones con los segmentos del tramo previo $\mathcal{C}_{i-1}$. En la Figura 2.16 se muestra un ejemplo de este procedimiento. En esta figura, se representan dos tramos consecutivos de $\mathcal{C}$, comprendidos entre los puntos $\{d_1, d_2, d_3\}$. Se puede apreciar que el punto d_3^0 puede ser unido con el d_2^2, de modo que el segmento sea tangente a $\mathcal{C}$. Sin embargo, este punto puede ser también unido a d_2^3 o a d_2^4, satisfaciendo la condición de formar una envolvente[2]. Más aún, el resultado óptimo puede ser la unión de d_3^0 con d_2^3 o d_2^4, dado que, aunque localmente no

[2]Notar que la Definición 2.4.3 es muy general y no exige que la envolvente sea tangente a la curva.

exista un área mínima de error, esta magnitud puede minimizarse en términos globales.

Un esquema general del algoritmo es el siguiente: para cada segmento $\mathcal{C}_i$, se obtienen todos los posibles segmentos tangentes, almacenando el resultado en la tabla de tangencia TT_i. Dado que cada uno de los segmentos es almacenado en una columna de la tabla de tangencia, la unión de segmentos se puede representar como una lista enlazada entre columnas de tablas de tangencia consecutivas. Un ejemplo de este esquema de almacenamiento se puede ver en la Figura 2.16. Supongamos que la envolvente óptima está conformada por los segmentos que unen d_1^2 con d_2^2 y d_2^2 con d_3^0. Vamos a denotar a dichos segmentos como $\overline{d_1^2 d_2^2}$ y $\overline{d_2^2 d_3^0}$, respectivamente. La unión de dos segmentos se puede representar como un enlace entre el campo TT_1^2 (primera tabla, tercera columna) con el campo TT_2^0 (segunda tabla, primera columna).

Para incluir este esquema de almacenamiento es necesario almacenar en la tabla de tangencia dos nuevos elementos denominados *error* y *enlace*. El primero de ellos almacena el área de error acumulada en el segmento considerado, mientras que el campo *enlace* almacena la columna de la tabla previa con la que el segmento actual está enlazado.

La Figura 2.17 ilustra el pseudocódigo de nuestro algoritmo denominado *envolvente óptima* (`EO`). Esta propuesta consta de dos partes denominadas *Forward propagation* y *Backward propagation*. Como primer paso a realizar, empleamos el algoritmo de Kankanhalli [73] para determinar el conjunto $\mathcal{D}$ de puntos dominantes. A continuación, en la *Forward propagation*, se procesan consecutivamente los distintos tramos de $\mathcal{C}$. Para cada uno de ellos, el algoritmo elabora la tabla de tangencia asociada. A continuación, para cada uno de los segmentos almacenados en la tabla, la función *calcula_error* determina su área de error asociada. Esta área está comprendida, dentro del tramo $\mathcal{C}_i$ considerado, entre el segmento procesado y la recta de ecuación $y = y_{ref}$. En donde el valor y_{ref} verifica la siguiente condición:

$$y_{ref} < p_{y_j} \quad \forall p_j \in \mathcal{C} \tag{2.18}$$

Es decir, y_{ref} es un valor menor que la ordenada de cualquier punto de la curva $\mathcal{C}$. En la Figura 2.16, se representa el área de error para el segmento $\overline{d_2^2 d_3^0}$. El valor de este área se corresponde con el número de elementos del conjunto de error más una constante. De este modo, utilizando el mismo valor de y_{ref} con todos los intervalos, podemos realizar una clasificación de los mismos en función de su área de error.

Una vez calculadas todas las áreas de error de un mismo intervalo, para cada $d_{i+1}^{k_1} \in \mathcal{C}_i$, se genera, mediante la función *añade_candidatos*, el conjunto $\mathcal{S}_i^{k_1}$, el cual se define del siguiente modo.

Definición 2.4.9 Para un punto $d_{i+1}^{k_1} \in \mathcal{C}_i$, $\mathcal{S}_i^{k_1}$ representa el conjunto de puntos $d_i^{k_2} \in \mathcal{C}_{i-1}$ con

```
Algoritmo EO
entrada
        C: curva digital
        N_D: número de puntos dominantes
salida
        E: Conjunto de puntos que conforman la envolvente óptima
inicio del algoritmo
     D = kankanhalli(C, N_D)

     % Forward propagation
     TT_1 <== TBL_TNG(d_1, d_2, p^max, C_1)
     DO i=2, N_D-1
        TT_i <== TBL_TNG(d_i, d_{i+1}, p^max, C_i)
         DO k = 1, k^max
            local_area <== calcula_error(d_i^k, d_{i+1}^{k'}, y_ref)
            S <== añade_candidatos(TT_{i-1}, d_{i+1}^{k'})
            IF(S ≠ ∅)
               {TT_i^k.enlace, TT_i^k.error} <== selecciona_menor_unión(S, d_{i+1}^{k'})
            ELSE
               {TT_i^k.enlace, TT_i^k.error} <==añade_nueva_entrada(TT_{i-1})
            END IF
L1          previo = TT_i^k.enlace
L2          TT_i^k.error = TT_i^k.error + TT_{i-1}^{previo}
            END DO
        END DO

     % Backward propagation
     k <== selecciona_menor_error(TT_{m-1})
     e_m = TT_{m-1}^k.d_{i+1}^{k'}
      DO i = m - 2, 1
         k = TT_i^k.enlace
         e_{i+1} = TT_i^k.d_{i+1}^{k'}
      END DO
     e_1 = TT_1^k.d_i^k
fin del algoritmo
```

Figura 2.17: Algoritmo *envolvente óptima* EO.

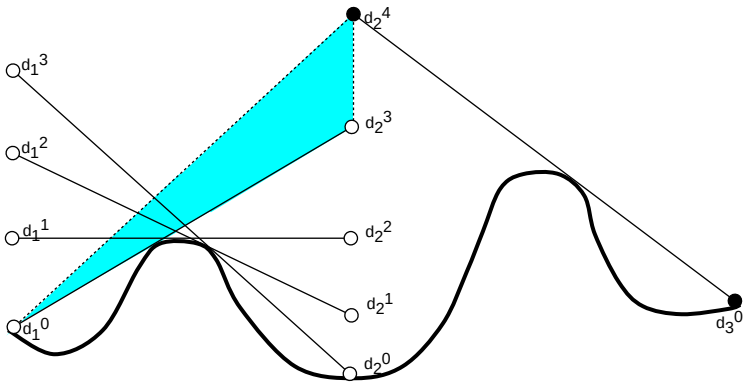


Figura 2.18: Nuevo nivel en d_2 debido al punto d_3^0.

los que el punto considerado puede ser unido por medio de un segmento sin cortar a $\mathcal{C}$. Más formalmente,

$$d_i^{k_2} \in \mathcal{S}_i^{k_1} \quad \Longleftrightarrow \quad \overline{d_i^{k_1} d_{i+1}^{k_2}} \cap \mathcal{C}_i = \emptyset \tag{2.19}$$

□

Un ejemplo se puede ver en la Figura 2.16. Para el punto d_3^0 tendremos que $\mathcal{S} = \{d_2^2, d_2^3, d_2^4\}$. La generación de este conjunto es muy simple, para un punto $d_{i+1}^{k_1}$ dado, únicamente hay que comparar la ordenada de su punto d_i^k asociado (almacenado en la misma columna de la tabla TT_i) con la ordenada de los puntos $d_i^{k_2} \in \mathcal{C}_{i-1}$. Si estas ordenadas son mayores, entonces los puntos pertenecen a $\mathcal{S}$. En base a este ejemplo, podemos establecer un criterio de selección de los elementos de $\mathcal{S}$.

$$d_i^{k_2} \in \mathcal{S}_i^{k_1} \iff d_{y_i}^{k_2} \geq d_{y_i}^{k_1} \tag{2.20}$$

El siguiente paso lo realiza la función *selecciona_menor_unión*. Para cada segmento formado por el punto considerado ($d_{i+1}^{k'}$) y los elementos de $\mathcal{S}_i^k$, se calcula el área de error asociada. Por ejemplo, considerando en la Figura 2.14 el segmento $\overline{d_3^0 d_2^2}$, el área total de error es la de la región sombreada. Sin embargo, considerando la unión $\overline{d_3^0 d_2^4}$, el error asociado se corresponde a la suma del área sombreada con el área del triángulo de vértices d_3^0, d_2^2, y d_2^4. Mediante esta función se evalúan los errores de todos los posibles enlaces y se selecciona aquel que lo minimiza. Una vez conocidas estas variables, los campos *enlace* y *error* de TT_i^k son actualizados. En la Figura 2.17 denotamos como $TT_i^k.enlace$ y $TT_i^k.error$ a la actualización de un campo concreto de esta tabla.

De este modo, para cada columna de la tabla TT_i, el campo *error* almacena el error asociado al segmento $\mathcal{C}_i$ considerado. Posteriormente, en las líneas etiquetadas como `L1`

y L2, añadimos a dicho campo el error de la etapa anterior. Mediante un procesamiento secuencial y ordenado de los distintos tramos de $\mathcal{C}$, podemos obtener el área de error global asociada a cada posible envolvente.

Una situación que es necesario contemplar es aquella en la que no hay ningún elemento en el tramo previo de $\mathcal{C}$ que pueda unirse con el actual. La Figura 2.18 muestra un ejemplo de este caso para el punto d_3^0. Inicialmente, el valor de máxima abscisa del tramo anterior es el punto d_2^3. Dado que el segmento $\overline{d_3^0 d_2^3}$ corta a $\mathcal{C}$, es necesario crear una nueva columna ($k = 4$) en la tabla de tangencia anterior de modo que d_2^4 pueda ser unida con el punto d_3^0. El proceso de creación es inmediato y lo realiza la función *añade_nueva_entrada*. El resto de los campos de esta nueva entrada son los mismos que los que tiene la entrada inmediatamente anterior. En la Figura 2.18, la entrada anterior sería el punto d_2^3, por lo que los campos d_i y t_i asociados a d_2^4 son los mismos que el de d_2^3. La única excepción es el área de error, dado que es necesario considerar el aumento producido. Este aumento viene dado por el área del triángulo cuyos vértices son el nuevo punto d_2^4, el punto anterior de la tabla d_2^3, y su d_i común (d_1^0). En el caso de la Figura 2.18, el incremento de área de error aparece sombreada.

El número de combinaciones posibles entre segmentos crece exponencialmente con el número de tramos. Sin embargo, la condición de minimización del área de error reduce significativamente el número de situaciones que debemos considerar. Este hecho lo podemos enunciar en la siguiente propiedad.

Propiedad 2.4.3 Sea $\{d_i^k\}$ el conjunto de posibles localizaciones del punto d_i, y sea $\mathcal{P}$ el conjunto total de puntos considerados por el algoritmo para formar la envolvente. Más formalmente,

$$\mathcal{P} = \{\{d_1^k\} \bigcup \{d_2^k\} \bigcup \ldots \{d_m^k\}\} \quad \forall k \tag{2.21}$$

Entonces para cualquier j tal que $1 \leq j \leq m$, si $e_j \in \mathcal{E}$ se verifica que $e_j \in \mathcal{P}$.

DEMOSTRACIÓN: Consideremos la situación en la que el punto $d_i^k \notin \mathcal{P}$ pero $d_i^k \in \mathcal{E}$. Entonces, de acuerdo con la definición de $\mathcal{P}$, el punto d_i^k no tiene una tangente asociada a la curva $\mathcal{C}$. Sin embargo, en este caso $\exists d_i^{k'} \in \mathcal{P}$, $k' < k$ que tiene un segmento tangente a $\mathcal{C}$. De este modo, y dado que el algoritmo procesa los puntos de forma ordenada, la envolvente puede emplear el punto $d_i^{k'}$ en vez del d_i^k ya que $k' < k$, reduciendo, de esta forma, el área de error. Por consiguiente, la envolvente óptima nunca puede incluir al punto d_i^k. □

Cuando todos los intervalos han sido procesados, la envolvente óptima es aquella que tiene, en su última entrada, el menor valor de error acumulado. Esto es realizado por la función *selecciona_menor_error*, la cual selecciona la columna k con menor área de error de la última tabla de tangencia. Finalmente, en la etapa de *Backward propagation* se

recorren en sentido opuesto los tramos de $\mathcal{C}$, y mediante el campo *enlace*, se van obteniendo los puntos de la envolvente óptima $\mathcal{E}$.

Aunque este algoritmo obtiene la solución óptima en términos de minimización del área de error, su empleo en situaciones reales puede resultar altamente ineficiente, dado su elevado coste computacional. Hemos identificado dos fuentes principales de ineficiencia en el rendimiento de este algoritmo. La primera de ellas hace referencia a consideraciones de requisitos de memoria. Dado que la curva es recorrida dos veces, es necesario almacenar las tablas de tangencia de cada tramo. Esto implica almacenar $N_{\mathcal{D}} - 1$ tablas, lo cual significa un considerable volumen de memoria. Por otra parte, considerando los costes computacionales, todos los puntos de cada nivel deben ser procesados con el fin de evaluar todas las posibles envolventes a la curva. Estos factores conllevan a que su empleo resulte prohibitivo en un entorno de análisis en tiempo de ejecución. Todos estos tópicos han motivado el desarrollo de una nueva técnica que obtenga una solución próxima a la óptima con un bajo coste computacional. Nuestra propuesta está basada en un método heurístico, y es descrita a continuación.

Técnica heurística para la obtención de la envolvente

Nuestra propuesta ofrece un método heurístico *greedy* que consigue una reducción tanto en los costes computacionales, como en los costes de almacenamiento. Del mismo modo que en el algoritmo anterior, la curva $\mathcal{C}$ es dividida en tramos $\mathcal{C}_i$ que son procesados secuencialmente. La principal diferencia entre ambas propuestas radica en el hecho que cada tramo de $\mathcal{C}$ es recorrido una única vez, obteniendo en ese momento la contribución final al resultado y minimizando localmente el área de error. De este modo, y dado que no existe una etapa de *Backward propagation*, no es necesario almacenar las tablas de tangencia de todos los tramos de la curva.

El pseudocódigo de nuestra propuesta, denominada `EH`, se muestra en la Figura 2.19. Para cada punto dominante d_i únicamente se consideran sus dos inmediatos vecinos d_{i-1} y d_{i+1}. El algoritmo evalúa el efecto de aumentar la ordenada del punto d_i. Tal y como se aprecia en la Figura 2.20, este aumento implica un decremento en la ordenada de sus dos vecinos. De acuerdo con esta figura, el efecto de modificar la posición del punto d_i implica un cambio del área de error que puede ser aproximado por la siguiente relación.

$$\Delta \acute{a}rea\_de\_error = (A2 + A3) - (A0 + A1 + A4) \tag{2.22}$$

En donde An representa el n-ésimo triángulo mostrado en la Figura 2.20. La primera contribución (positiva) representa un aumento del área de error debida al aumento de la

```
Algoritmo EH
entrada
        C: curva digital
        N_D: número de puntos dominantes
salida
        E: Conjunto de puntos que conforman la envolvente óptima
inicio del algoritmo
        D = kankanhalli(C, N_D)
        TT_1 ⇐= TBL_TNG(d_1, d_2, p^max, C_1)
        DO i = 2, m − 1
            TT_i ⇐= TBL_TNG(d_i, d_{i+1}, p^max, C_i)
            Δerror ⇐= evaluar_área_de_error(d_{y_i} + 1, TT_{i−1}, TT_i)
            WHILE (Δerror < 0)
                    d_{y_i} = d_{y_i} + 1
                    Δerror ⇐= evaluar_área_de_error(d_{y_i} + 1, TT_{i−1}, TT_i)
            END WHILE
            {e_{i−1}, e_i, e_{i+1}} ⇐= actualiza_puntos(d_{y_i}, TT_{i−1}, TT_i)
        END DO
fin del algoritmo
```

Figura 2.19: Algoritmo *envolvente heurística* EH.

ordenada de d_i, mientas que la segunda (negativa) está asociada a la disminución de la ordenada de los puntos d_{i-1} y d_{i+1}.

El algoritmo procede del siguiente modo: para cada par de tramos consecutivos, mediante la función TBL_TNG se generan sus correspondientes tablas de tangencia. A continuación, y utilizando la información de estas tablas, se aumenta en una posición la ordenada de d_i, obteniendo el valor de las nuevas posiciones de d_{i-1} y d_{i+1}. Una vez conocidas todas estas posiciones, se obtiene, mediante la función *evaluar_área_de_error*, el incremento del área de error debido al desplazamiento de d_i. Este proceso se repite mientras este valor sea negativo, obteniendo, de este modo, una minimización local del área de error.

La Expresión 2.22 puede ser simplificada teniendo en cuenta que el área de cada triángulo se puede expresar en función del valor de la abscisa de cada uno de los pun-

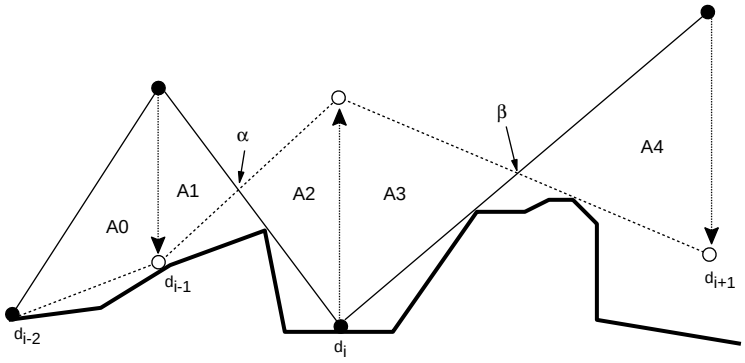


Figura 2.20: Efecto del incremento en y_i.

tos. De este modo, la expresión se puede reescribir de la siguiente manera.

$$(x_\beta - x_\alpha)(d_{x_i} - x_\alpha)(x_\beta - d_{x_i}) < (x_\alpha - d_{x_{i-1}})(x_\alpha - d_{x_{i-2}})(x_\beta - d_{x_i}) - (d_{x_{i+1}} - x_\beta)^2(d_{x_i} - x_\alpha) \tag{2.23}$$

Esta desigualdad se verifica sólo si $\Delta \acute{a}rea\_de\_error < 0$. Los valores x_α y x_β son las abscisas de los puntos α y β. Estos puntos, marcados en la Figura 2.20, están situados en la intersección de los segmentos asociados a la posición inicial y final del punto d_i. Nótese que, cuando los puntos d_{i-1} y d_{i+1} están situados sobre la curva, y d_i es modificado, entonces d_{i-1} y d_{i+1} no cambian de posición. Para esta situación x_α y x_β coincidirán, respectivamente, con la abscisa de d_{i-1} y d_{i+1}.

Mediante el empleo de la Expresión 2.23 se puede determinar de un manera eficiente si se ha producido una reducción local en el área de error. La función *evaluar_área_de_error* realiza esta comprobación.

La siguiente propiedad nos permite establecer un criterio de finalización.

Propiedad 2.4.4 Cuando para un valor d_{y_i} dado, la desigualdad dada por la Expresión 2.23 deja de verificarse, entonces, para el resto de puntos con una abscisa mayor que la de d_{y_i}, la Expresión 2.23 tampoco se verificará.

DEMOSTRACIÓN: Sea $d_i^{k_1}$ el primer punto para el que la Expresión 2.23 no se cumple, y sean $x_\alpha^{k_1}$ y $x_\beta^{k_1}$ sus valores asociados. Para el resto de los puntos $d_i^{k_2}$, con $k_2 > k_1$ se cumple que $x_\alpha^{k_2} \leq x_\alpha^{k_1}$ y $x_\beta^{k_2} \geq x_\beta^{k_1}$. Teniendo en cuenta estas desigualdades y dado que $d_{x_{i-2}}, d_{x_{i-1}}, d_{x_i}, d_{x_{i+1}}$ son constantes, se puede comprobar que el primer término siempre aumentará o mantendrá su valor, mientras que el segundo disminuirá o mantendrá su valor. De este modo, $\forall k_2 > k_1$ la Expresión 2.23 tampoco se verificará. □

En base a esta propiedad podemos concluir que, partiendo del último valor d_i que minimiza el área local de error, no existe ningún otro con mayor ordenada que reduzca más aún el área local de error. Una vez minimizada este área, mediante la función *actualiza_puntos*, la posición de los puntos $\{e_{i-1}, e_i, e_{i+1}\}$ es actualizada, pasando a considerar el siguiente intervalo de puntos. Siendo más específicos, se evalúa el incremento de la ordenada del punto d_{i+1}, teniendo como vecinos los puntos d_i y d_{i+2}. Nótese que en el nuevo intervalo únicamente es necesario operar sobre las tablas de tangencia TT_i y TT_{i+1}, pudiendo ser eliminada la tabla TT_{i-1}. El algoritmo finaliza cuando el último tramo de $\mathcal{C}$ es procesado, obteniendo en el conjunto $\mathcal{E}$ el resultado de la envolvente.

La siguiente sección evalúa, en términos de rendimiento computacional y calidad del resultado, la eficiencia de los dos algoritmos propuestos.

2.4.2 Evaluación del rendimiento

En esta sección realizamos dos tipos de consideraciones. Las primeras son referentes a una evaluación teórica del rendimiento de nuestras propuestas, mientras que las segundas muestran distintos resultados experimentales obtenidos con las mismas.

Resultados teóricos

La complejidad del algoritmo `EO` es del orden:

$$\mathcal{O}_{\texttt{EO}} = N_{\mathcal{C}} h + N_{\mathcal{D}} h^2 \tag{2.24}$$

Donde $N_{\mathcal{C}}$ el es número de puntos de la curva, $N_{\mathcal{D}}$ el número de puntos dominantes y h es el número medio de entradas[3] en las tablas de tangencia. El coste de almacenamiento de esta propuesta (denotado como $\Delta M_{\texttt{EO}}$) representa número medio de entradas que es necesario almacenar en la ejecución del algoritmo. Dicho coste viene dado por:

$$\Delta M_{\texttt{EO}} = 5h(N_{\mathcal{D}} - 1) \tag{2.25}$$

Donde el factor 5 es el número de campos por cada entrada de la tabla y $(N_{\mathcal{D}} - 1)$ se corresponde al número de tablas que es necesario almacenar. Por otra parte, la complejidad del algoritmo `EH` es, en el peor de los casos, dada por la siguiente relación:

$$\mathcal{O}_{\texttt{EH}} = N_{\mathcal{D}} h \tag{2.26}$$

[3]En nuestro esquema de almacenamiento, una entrada de la tabla de tangencia representa a una columna de la misma. Cada columna contiene a su vez varios campos, sin embargo, debido a que este número es un valor constante, no aparece reflejado en la Expresión 2.24.

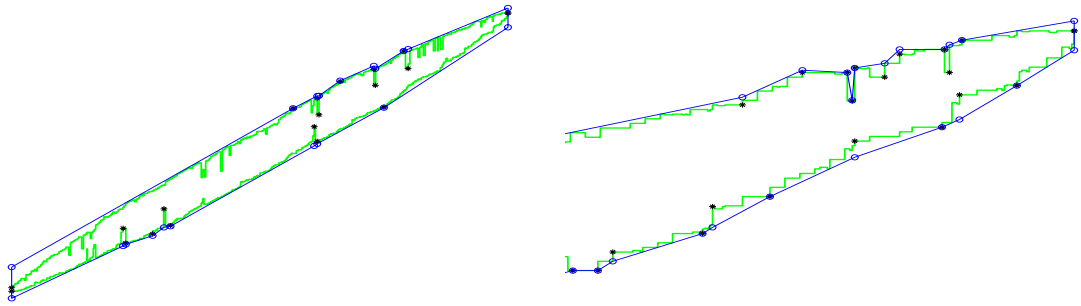


Figura 2.21: Curvas asociadas a la matriz *bcsstk14*.

El coste de almacenamiento del algoritmo `EH` es menor que el del algoritmo `EO`, dado que sólo hay que almacenar dos tablas de tangencia. Este coste se puede aproximar mediante la siguiente relación.

$$\Delta M_{\texttt{EH}} \simeq 5h2 \tag{2.27}$$

Hay que destacar que, en la práctica, la complejidad de la propuesta heurística es mucho menor que la mostrada, ya que en cada nivel, el proceso de análisis se detiene cuando el área de error es minimizada, no siendo necesario considerar (a diferencia de algoritmo óptimo) todas las posibles posiciones de los puntos d_i^k. Dado que hemos utilizado el algoritmo de Kankanhalli, es necesario añadir a ambas propuestas su coste computacional de complejidad dada por:

$$\mathcal{O}_{Kankanhalli} = N_{\mathcal{D}} N_{\mathcal{C}} \tag{2.28}$$

Resultados experimentales

Experimentalmente hemos evaluado el rendimiento de ambas propuestas para distintas curvas extraídas de los vectores u y l generadas por el algoritmo `CSM`. Cada punto de estos vectores tiene asociado un valor de abscisa igual al número de *slice*, y un valor de ordenada igual al número de entrada accedida por la indirección (que en un formato de almacenamiento CCS se corresponde al número de fila del elemento). Utilizamos matrices de la librería Harwell-Boeing [37], en concreto las matrices *bcsstk14* y *bcsstk17* que generaron tras el proceso de interpolación curvas digitales con 11773 y 69405 puntos, respectivamente. Estas dos matrices fueron seleccionadas porque sus curvas digitales asociadas presentan regiones con características muy variadas, que incluyen zonas muy abruptas y zonas con transiciones suaves.

La Figura 2.21 muestra, para la matriz *bcsstk14*, las curvas u y l junto con sus respecti-

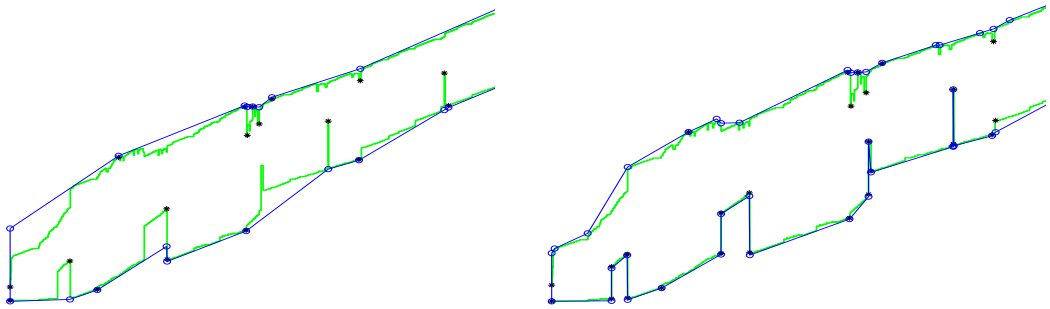
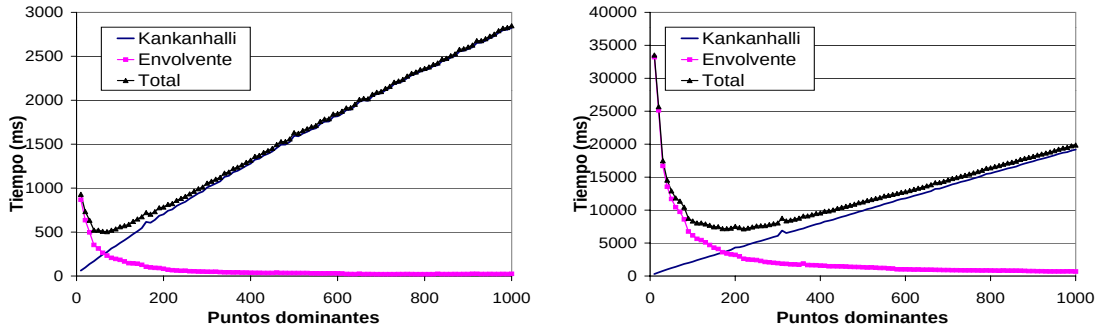


Figura 2.22: Curvas asociadas a la matriz *bcsstk17*.

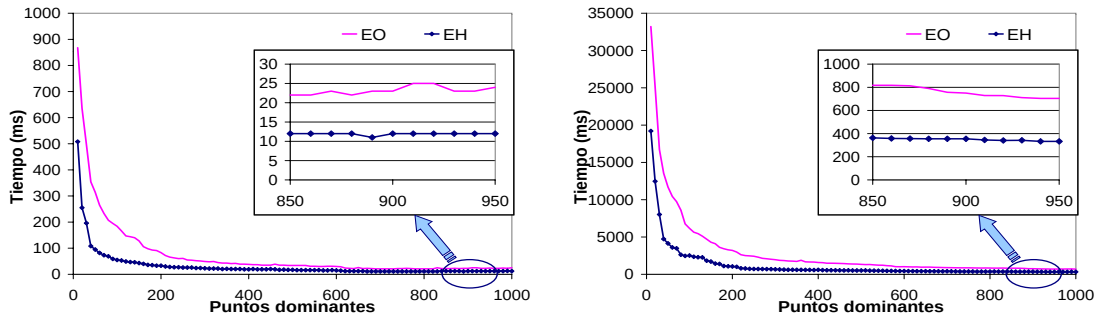


(a) *bcsstk14* (b) *bcsstk17*

Figura 2.23: Tiempos de ejecución del algoritmo `EO`.

vas envolventes. En esta representación se utilizaron 20 puntos dominantes. En la misma figura se muestra un detalle de la representación empleando 200 puntos dominantes. En estas figuras, los símbolos "$*$" y "$\circ$" representan, respectivamente, los puntos de los conjuntos $\mathcal{D}$ y $\mathcal{E}$, es decir, el conjunto de puntos dominantes de la curva digital y el conjunto de puntos que conforman la envolvente a dicha curva. En todos los casos, se utilizó el algoritmo óptimo para obtener la curva envolvente. La Figura 2.22 muestra, para distinto número de puntos dominantes, secciones de la envolvente asociadas a la matriz *bcsstk17*. Concretamente se utilizaron 100 y 200 puntos dominantes.

La Figura 2.23 representa el tiempo de ejecución del algoritmo óptimo en función del número de puntos dominantes. Las medidas fueron realizadas en un sistema SUN Enterprise 250. Este tiempo de ejecución aparece desglosado en el coste del proceso de extracción

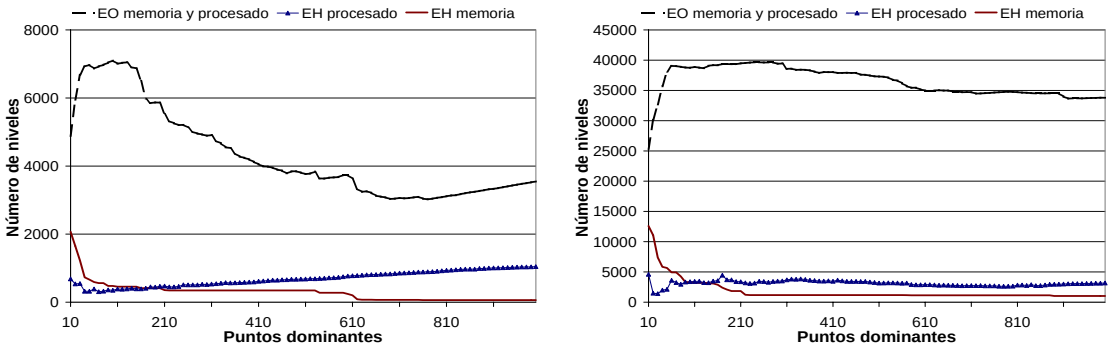


(a) *bcsstk14* (b) *bcsstk17*

Figura 2.24: Tiempos de ejecución del cálculo de la envolvente para ambas propuestas.

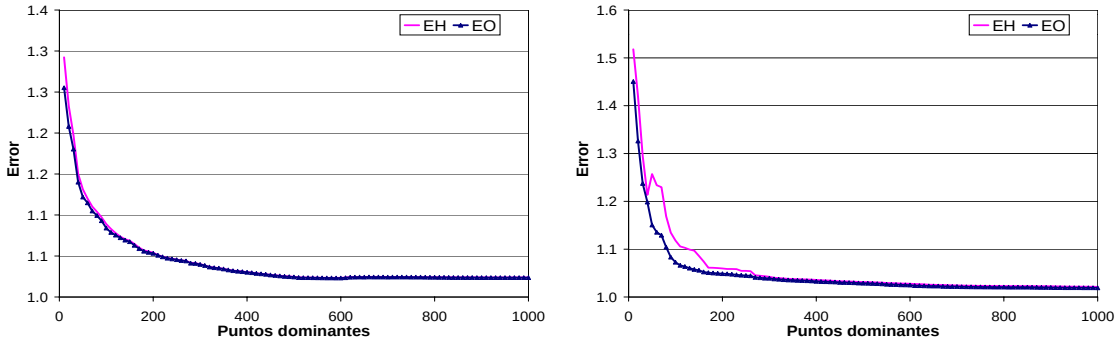
de los puntos dominantes (algoritmo de Kankanhalli) y el del proceso determinación de la envolvente. Así mismo, también aparece representado el tiempo total de ejecución, el cual es la suma de las dos etapas anteriores. Se puede apreciar cómo el coste del algoritmo de Kankanhalli guarda una dependencia aproximadamente lineal con el número de puntos dominantes. Por otro lado, conforme aumenta este número, el tiempo del proceso de cálculo de la envolvente decrece. La explicación radica en el hecho que, aunque el número de tramos (y por lo tanto, de tablas de tangencia) aumenta con el de puntos dominantes, el número medio de niveles por cada tramo (h) disminuye en una razón superior. Esta disminución es debida a que cuando la longitud de cada tramo (dada por $d_{x_{i+1}} - d_{x_i}$) disminuye, también lo hace el número de niveles que hay que considerar.

Una comparativa entre el algoritmo óptimo (`EO`) y el heurístico (`EH`) se puede ver en la Figura 2.24. En ambos casos, y para facilitar la comparación, no se ha tenido en cuenta el coste del algoritmo de Kankanhalli. Se puede apreciar la fuerte reducción de tiempo de ejecución cuando se emplea el algoritmo heurístico. Este algoritmo es, en algunos casos, hasta cinco veces más rápido que el óptimo. La Figura 2.25 muestra la máxima cantidad de entradas (denotadas como *niveles*) de las tablas de tangencia que son almacenadas y procesadas. En el algoritmo óptimo es necesario procesar todas ellas, por lo que el número de entradas almacenadas y procesadas coincide. Notar que, conforme aumenta el número de puntos dominantes, estas magnitudes pueden llegar a disminuir dado que el número de entradas de las tablas (parámetro h) decrece. Tal y como refleja la Figura 2.25, el número total de entradas procesadas por el algoritmo heurístico es mucho menor que el del óptimo. Adicionalmente, y dado que únicamente se debe almacenar de manera



(a) *bcsstk14* (b) *bcsstk17*

Figura 2.25: Eficiencia de ambas propuestas.

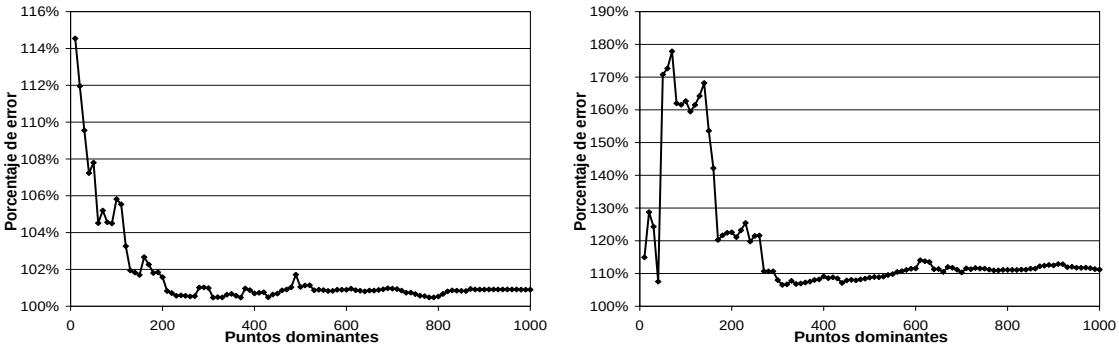


(a) *bcsstk14* (b) *bcsstk17*

Figura 2.26: Factores de error.

simultánea dos estructuras de niveles, los costes de memoria empleada por el algoritmo heurístico también son mucho más reducidos que para el otro caso.

La Figura 2.26 muestra una comparativa de ambas propuestas en términos de área de error. Hemos normalizado estos resultados de acuerdo con el número de puntos interiores a $\mathcal{C}$. De este modo, las gráficas representan, para cada una de las propuestas, la relación $\| I(\mathcal{E}_f) \| / \| I(\mathcal{C}) \|$. Notar que las curvas digitales u y l están próximas entre sí, por lo que la influencia de $I(\mathcal{C})$ no resulta muy elevada. Adicionalmente, la Figura 2.27 muestra la



(a) *bcsstk14* (b) *bcsstk17*

Figura 2.27: Porcentaje de diferencia de errores.

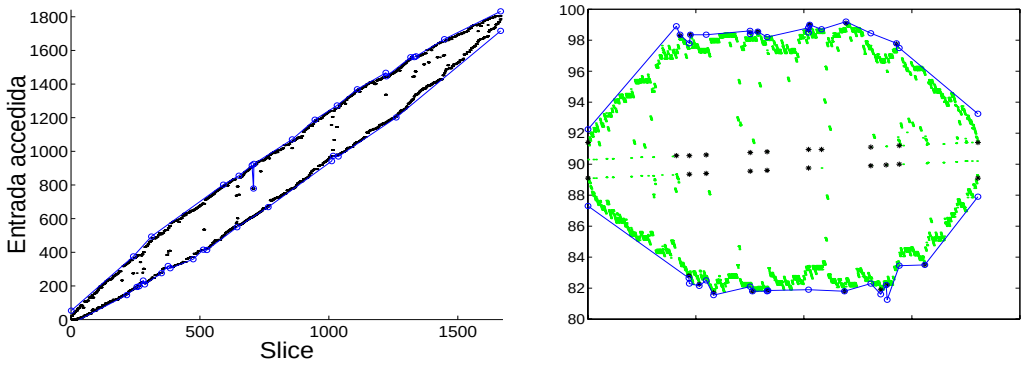
relación entre los errores de ambas propuestas, la cual se obtiene por la siguiente expresión:

$$\%error = 100\frac{\parallel I(\mathcal{E}_f^{EH}) \parallel}{\parallel I(\mathcal{E}_f^{EO}) \parallel} \tag{2.29}$$

Esta relación es similar para la mayor parte de las situaciones, especialmente cuando el número de puntos dominantes es elevado. Los peores resultados fueron obtenidos para la matriz *bcsstk17*, dado que sus curvas digitales asociadas son menos suaves, siendo necesario un gran número de puntos dominantes para alcanzar los mismos niveles de calidad que en el caso óptimo.

En base a este análisis comparativo de ambas propuestas hemos decidido utilizar el algoritmo heurístico como herramienta de caracterización del patrón de acceso. Los motivos de esta elección son variados. Por una parte, el resultado obtenido mediante esta técnica siempre verifica la única condición estricta impuesta al formular el problema: la curva resultado acota a la curva origen. Por otra parte la eficiencia de esta propuesta es muy superior a la del algoritmo óptimo, tanto en términos de coste computacional como de consumo de memoria. Además, la calidad del resultado es aceptable en el marco del dominio de aplicación, ya que la obtención de un área de error mínima no supone un factor crítico. Adicionalmente, y considerando las figuras 2.26 y 2.27, este error puede ser fácilmente reducido mediante el aumento del número de puntos dominantes.

Un último aspecto que vamos a considerar es la influencia de la geometría del patrón de acceso en el rendimiento del algoritmo `EH`. En términos de coste computacional, el rendimiento de nuestra propuesta mejora significativamente si el valor de la pendiente del



(a) Original (b) Transformado

Figura 2.28: Ejemplo de operación por desplazamiento por *slice*.

patrón de acceso es próximo a cero. Esto se puede conseguir modificando los vectores u y l mediante una operación que denominamos desplazamiento por *slice*.

Definición 2.4.10 Definimos un **desplazamiento por *slice*** sobre cada una de las entradas de u o l como la suma de un valor igual al producto de una constante c multiplicada por el índice del *slice* k que dicha entrada tiene asociada.

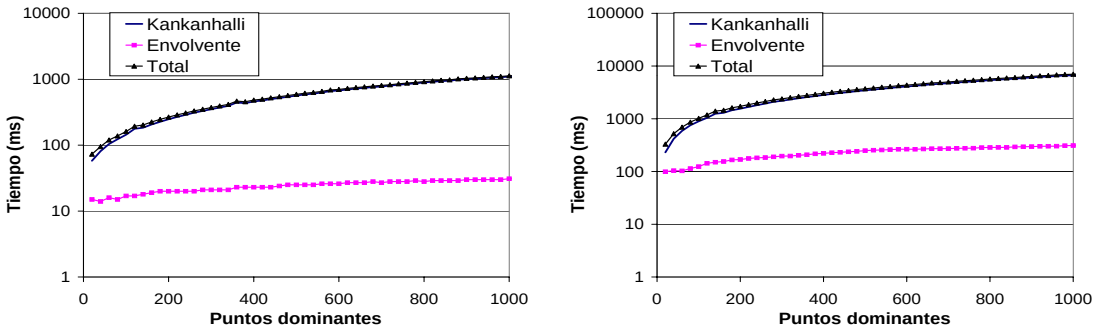
$$u[k] \Longrightarrow u[k] + c * k \qquad 1 \leq k \leq N_S \tag{2.30}$$

$$l[k] \Longrightarrow l[k] + c * k \qquad 1 \leq k \leq N_S \tag{2.31}$$

□

Mediante esta operación, es posible modificar la tendencia del patrón de acceso. Un ejemplo de esta transformación se puede ver en la Figura 2.28. La Figura 2.28(a) representa el patrón de partida correspondiente a la matriz *bcsstk14*. Considerando la curva/envolvente superior de la misma, el primer y último elemento de la representación original tienen de coordenadas $(1, 23)$ y $(1666, 1805)$, respectivamente. El valor de la constante c es igual a la pendiente del patrón de acceso invertida de signo. De este modo, un valor aproximado de la pendiente del patrón de acceso es: $a = (1805 - 23)/(1666 - 1) = 1.07$ *entradas/slice*. Para compensar esta pendiente, en nuestra transformación de desplazamiento por *slice* hemos tomado el valor de $c = -a = -1.07$. De este modo, el patrón de acceso asociado presenta una pendiente aproximadamente nula, tal y como se muestra en la Figura 2.28(b).

Experimentalmente, hemos integrado una rutina que obtiene de forma automática el valor de c en función del valor medio de la cota superior e inferior como valor aproximado



(a) *bcsstk14* (b) *bcsstk17*

Figura 2.29: Tiempos de ejecución del algoritmo `EH` con un patrón de acceso cuya pendiente asociada es próxima a cero.

de la pendiente. Así pues, el valor medio inicial m_{ini}, viene dado por $m_{ini} = (u[1] + l[1])/2$, mientras que el final, m_{fin}, será $m_{fin} = (u[N_S] + l[N_S])/2$. De este modo, el valor aproximado de la pendiente del patrón de acceso es $a = (m_{fin} - m_{ini})/N_S$. Para realizar la transformación únicamente es necesario tomar un valor de $c = -a$ y aplicar las transformaciones mostradas en las expresiones 2.30 y 2.31.

Los nuevos valores de u y l se corresponden a un patrón sobre el que es aplicado algoritmo `EH` obteniéndose una mejora en su rendimiento. Para generar las envolventes del patrón de acceso original únicamente hay que aplicar sobre $\mathcal{E}^u$ y $\mathcal{E}^l$ una nueva transformación de desplazamiento por *slice*. El nuevo valor de la constante c' viene dado por $c' = -c$. Es decir,

$$e_{y_i} \Longrightarrow e_{y_i} + c' e_{x_i}, \quad i = 1, \ldots N_{\mathcal{E}} \tag{2.32}$$

Aplicando el algoritmo `EH` sobre esta nueva representación, los tiempos de ejecución cambian significativamente. La Figura 2.29 muestra, en escala logarítmica, los valores obtenidos para un número variable de puntos dominantes. Nuevamente el tiempo de ejecución aparece desglosado en el coste del algoritmo de Kankanhalli y el coste del cálculo de la envolvente. Este último valor puede ser comparado con el mostrado en la Figura 2.24. Se puede apreciar una significativa reducción en el tiempo de cálculo de la envolvente cuando el número de puntos dominantes es reducido. Notar que, cuando este número aumenta, el tiempo de cálculo de la envolvente es menor si se utiliza el patrón de acceso original.

Podemos concluir que la estrategia más eficiente para la obtención de la envolvente

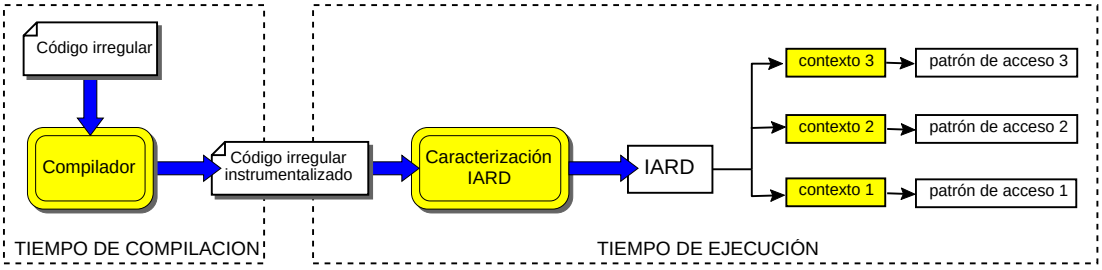


Figura 2.30: Procedimiento de caracterización de una indirección.

es mediante el empleo del algoritmo `EH` con un número de puntos dominantes lo más reducido posible. Adicionalmente, y dado que usualmente vamos a trabajar con pocos puntos dominantes, resulta más rentable aplicar, sobre el patrón de acceso, una operación de desplazamiento por *slice*. La siguiente sección describe el proceso de aplicación de esta propuesta a la caracterización de un patrón de acceso dado por una indirección.

2.5 Caracterización del patrón de acceso

En esta sección describimos el proceso de integración entre la propuesta de clasificación con el modelo geométrico descrito en la sección previa. Adicionalmente, mostramos resultados de su aplicación sobre patrones de acceso reales y planteamos técnicas de generalización a esquemas de indireccionamiento más complejos.

2.5.1 Obtención de la representación IARD

El esquema de caracterización automática del patrón de acceso de un código irregular implica un análisis tanto en tiempo de compilación como en tiempo de ejecución. La Figura 2.30 muestra un esquema general de esta propuesta. En tiempo de compilación se identifican los bloques de código irregulares del programa y las indirecciones existentes en las mismas. Para cada una de estos bloques, se extrae y almacena toda la posible información acerca del contexto de ejecución que tiene asociado. Como ejemplo de dicha información podemos citar el modo de acceso a la indirección, el tipo de operación realizada, el número de iteraciones del lazo, etc. En esta misma fase, el siguiente paso a realizar consiste en la identificación de las puntos del programa en las que estas indirecciones son inicializadas o sufren modificaciones. Estas partes del código son instrumentalizadas mediante la introducción de llamadas a las rutinas de caracterización. Posteriormente, en tiempo de ejecución, estas rutinas analizan cada una de las indirecciones y obtienen

```
DO j = 1000, N_x − 500
   a[x[j]] = ...
END DO
```

```
DO j = 1, N_x
   a[2x[j] + 200] = ...
END DO
```

Figura 2.31: Ejemplo de estructuras irregulares.

su representación IARD. El proceso de análisis se divide, a su vez, en dos etapas. En la primera de ellas se aplica el algoritmo CS o el CSM, obteniendo como resultado, el conjunto de vectores $\{u, l, \rho\}$ de N_S entradas. Tal y como se comentó en la Sección 2.3.2, el criterio de elección de un algoritmo u otro depende de las características del código irregular. La segunda etapa hace uso del algoritmo EH para obtener los conjuntos $\mathcal{E}^u$ y $\mathcal{E}^l$ que conforman la envolvente superior e inferior al patrón de indirección. De este modo, la representación del patrón de acceso de una indirección se puede realizar considerando el conjunto $\{\mathcal{E}^u, \mathcal{E}^l, \rho\}$ de $2N_{\mathcal{E}} + N_S$ entradas. Denominamos a dicho conjunto ***Irregular Access Region Descriptor*** (IARD) de la indirección. Retomando el diagrama estructural de nuestra propuesta mostrado en la Figura 2.30, a partir de un vector de indirección, se puede obtener su representación IARD. El único parámetro externo que debemos considerar es el que selecciona el tipo de técnica de clasificación por *slices* (algoritmo CS o CSM) a aplicar.

De igual forma, el patrón de acceso de cada bloque del código irregular puede derivarse a partir de la representación IARD. Este proceso consiste en transformar la representación IARD empleando la información almacenada sobre el contexto. En la siguiente sección se describe de forma detalla este proceso de transformación. De este modo, mediante el empleo de una única caracterización IARD se puede obtener el patrón de acceso de distintas regiones que operan con el mismo vector de indirección. Un última propiedad de esta representación que vamos a explotar es la capacidad de transformación a otros esquemas de acceso a la indirección. En la siguiente sección se describe la generalización de la representación IARD de modo que contemple esquemas de acceso irregulares más complejos.

2.5.2 Transformación de la representación IARD

De acuerdo con la estrategia inicialmente planteada, la representación IARD caracteriza el patrón de indirección asumiendo un acceso sobre entradas consecutivas del vector de indirección. Sin embargo, podemos encontrar situaciones en las que el vector de indirección no es accedido de esta forma. Este es el caso de los códigos mostrados en la Figura 2.31.

En el primero de ellos, únicamente se recorre parte del espacio de indirecciones, por lo que el patrón de acceso será más reducido que el original. En el segundo lazo, el patrón de acceso viene determinado por una combinación lineal del vector de indirección original. Del mismo modo que en el caso anterior, el patrón de acceso es diferente al contemplado en la representación original de la indirección. Las características particulares de cada lazo son almacenadas en su contexto y forman parte de su representación IARD. Nuestra meta es utilizar esta información para obtener la representación correspondiente al patrón del lazo considerado. Esta nueva representación debe cumplir los requisitos impuestos en la Definición 2.4.3, y la elaboración de la misma sólo debe emplear la información almacenada en la representación IARD original.

En general, hemos clasificado en tres grupos las posibles situaciones que se pueden dar en un lazo irregular con una única indirección y un único nivel de anidamiento. Denominamos a cada grupo como clase A, B y C. A continuación enumeramos las características de cada uno de ellos.

- **Clase A**. El límite superior o inferior del lazo es diferente del considerado en la representación IARD. Más formalmente, si el índice j comienza en el lazo con un valor j_{ini} y acaba con otro valor j_{fin}, entonces un lazo es Tipo A si $j_{ini} \neq 1$ y/o $j_{fin} \neq N_x$.
- **Clase B**. El mecanismo de acceso sobre el vector a es diferente al inicialmente considerado. Dentro de esta categoría estamos considerando accesos del tipo $a[kx[j]]$, $a[x[j]+k]$, $a[x[j]+j]$, etc. Donde k es una constante y j el índice del lazo.
- **Clase C**. Los accesos sobre el vector x no se realizan sobre entradas consecutivas del mismo. Ejemplos de esta categoría son accesos del tipo $a[x[kj]]$, siendo k una constante.

Estas clasificaciones no son excluyentes entre sí, dado que un lazo puede pertenecer simultáneamente a varias categorías. En este caso, el mecanismo de caracterización es la combinación de las transformaciones impuestas en cada una de ellas. A continuación describimos la transformación que debe realizarse sobre la representación IARD para cada una de estas clases.

Transformación de la representación para lazos clase A

Vamos a introducir dos nuevos elementos en el contexto de la representación IARD. Estos nuevos elementos, denotados como $slice_{ini}$ y $slice_{fin}$ son utilizados para acotar el

primer y último *slice* accedido por el lazo considerado. Consideremos el vector **densidad acumulada**, denotado como ρ_{ini} y definido como:

$$\rho_{ini}[i] = 1 + \sum_{s=1}^{i-1} \rho[s] \qquad 1 \leq i \leq (N_S + 1) \tag{2.33}$$

Asumiendo que en índice del lazo j está acotado entre los valores $j_{ini} \leq j \leq j_{fin}$, podemos enunciar la siguiente definición.

Definición 2.5.1 Los parámetros $slice_{ini}$ y $slice_{fin}$ son el mayor valor entero que verifican las siguientes relaciones: $\rho_{ini}[slice_{ini}] \leq j_{ini}$ y $\rho_{ini}[slice_{fin}] \leq j_{fin}$. □

De este modo, realizamos una aproximación conservadora, garantizando que el primer y último acceso pertenece, respectivamente, al primer y último *slice* de nuestra representación. El error cometido es proporcional al número de entradas de cada *slice* y al número total de estos. Experimentalmente hemos podido comprobar que con patrones de acceso reales cada *slice* tiene unas pocas decenas de entradas, por lo que si $j_{fin} - j_{ini} \gg 0$ el error cometido en la aproximación será reducido.

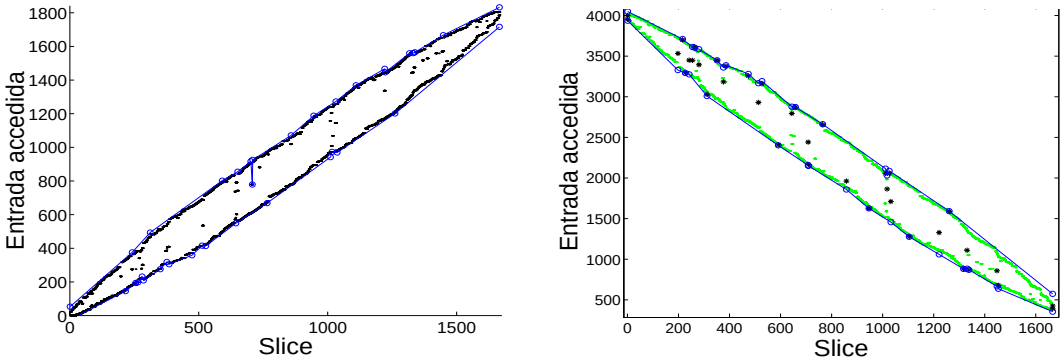
En esta clase de transformación, el contenido de las curvas $\mathcal{E}^u$ y $\mathcal{E}^l$ asociadas a la nueva representación es el mismo que el de la caracterización original. En el caso del vector ρ, este conserva el mismo valor en sus entradas[4] con la excepción de aquellas pertenecientes a los *slices* $slice_{ini}$ y $slice_{fin}$. Para estos casos, el número de accesos a memoria que se realizan vienen dados por las siguientes expresiones:

$$\rho[slice_{ini}] = \rho_{ini}[slice_{ini} + 1] - j_{ini} \qquad \rho[slice_{fin}] = \rho_{ini}[slice_{fin} + 1] - j_{fin} \tag{2.34}$$

Transformación de la representación para lazos clase B

Dentro de esta categoría existe un gran número de posibles tipos de accesos sobre el vector de indirección. La Tabla 2.3 trata de clasificar las situaciones más comunes. La primera de ellas se corresponde con una **operación de desplazamiento**: todos los accesos son desplazados una cantidad constante. Esta operación aparece frecuentemente en códigos irregulares en donde la matriz es accedida de distinta forma en distintas regiones. Un ejemplo típico son las matrices que contienen bloques de datos diferentes. En estos casos las indirecciones "apuntan" a posiciones absolutas, por lo que es necesario desplazarlas al bloque considerado. La transformación de una representación IARD para este tipo de lazos es simple. Únicamente hay que sumar el valor de la constante c a la ordenada de los puntos de $\mathcal{E}^u$ y $\mathcal{E}^l$, tal y como se muestra en la Tabla 2.3.

[4] La única diferencia es que ahora se accede a un menor número de ellas.



(a) Original (b) $c_1 = -2$ y $c_2 = 4000$

Figura 2.32: Ejemplos de transformación en el patrón de acceso.

En los accesos denominados como de **operación de multiplicación**, cada entrada de la indirección se multiplica por una constante que denominamos c. Como regla general, tal y como se ilustra en la Tabla 2.3, la transformación en la representación IARD consiste en multiplicar por dicha constante la ordenada de cada punto de la envolvente.

Las dos situaciones anteriores pueden aparecer formando el denominado **acceso mixto**. En este caso, la transformación es la combinación de las dos anteriores. La Figura 2.32(a) muestra el patrón de acceso original para la matriz *bcsstk14*, mientras que la Figura 2.32(b) muestra el patrón de acceso para un acceso mixto con $c_1 = -2$ y $c_2 = 4000$. Para este caso, la envolvente al patrón fue derivada aplicando las reglas de transformación mostradas en la Tabla 2.3.

Finalmente, el último caso contemplado es el **acceso genérico**, el cual viene dado por una función arbitraria que denominamos f. Un ejemplo de esta función es $f(j) = j$ o $f(j) = y[j]$ siendo y otro vector de indirección. En este caso no es posible una extrapolación directa entre el patrón almacenado en el IARD y el patrón de acceso realizado por el

Nombre	Acceso	Restricciones	Transformación
Desplazamiento	$a[x[j] + c]$	c constante, $c \in \mathbb{N}$	$e_{y_i} = e_{y_i} + c$
Multiplicación	$a[c * x[j]]$	c constante, $c \in \mathbb{R}$	$e_{y_i} = c * e_{y_i}$
Mixta	$a[c_1 * x[j] + c_2]$	c_1, c_2 constantes, $c_1 \in \mathbb{R}, c_2 \in \mathbb{N}$	$e_{y_i} = c_1 * e_{y_i} + c_2$
Acceso genérico	$a[x[j] + f(\ldots)]$	f función arbitraria, $f \in \mathbb{N}$	—

Tabla 2.3: Ejemplo de accesos para un lazo tipo B.

lazo. Para estas situaciones es necesario rehacer la caracterización del lazo particular, considerando una nueva indirección del tipo $x'[j] = x[j] + f(\ldots)$, la cual contempla el patrón de acceso real.

Transformación de la representación para lazos clase C

En este grupo enmarcamos aquellos códigos que no acceden a entradas consecutivas del vector de indirección. En nuestro caso asumimos que el acceso es mediante una constante que modifica el índice de x, sin embargo esta expresión es equivalente a asumir un *stride* en el lazo principal y un acceso simple (únicamente mediante el índice j) sobre el vector de indirección.

Para esta clase de lazos, para cada *slice* únicamente se accede a una fracción de los elementos que contiene. De este modo, si $k = 2$ el número de accesos se reduce a la mitad, si $k = 3$ a un tercio, etc. Considerando la representación IARD, esto se traduce en una reducción del contenido del vector densidad, mientras que los vectores $\mathcal{E}^u$ y $\mathcal{E}^l$ no sufren modificaciones.

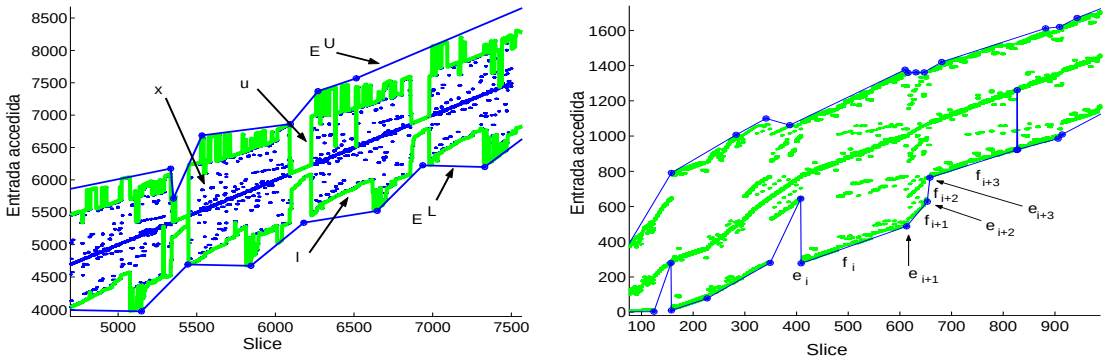
2.5.3 Evaluación del Rendimiento

Con el fin de evaluar el rendimiento de nuestra propuesta, la hemos la hemos aplicado a la caracterización de vectores de indirección extraídos de la librería Harwell-Boeing. De forma más concreta, hemos utilizado un formato de almacenamiento por columnas [13], empleando, como vector de indirección, el vector *row*. Como estrategia de clasificación de las entradas, hemos utilizado el algoritmo CS.

Los resultados obtenidos con nuestra propuesta aparecen en la Tabla 2.4. Parte de los parámetros ilustrados en esta tabla fueron introducidos en la Sección 2.3.3. Adicional-

Matriz	N_a	N_x	N_S	ΔM_{slices}	$N_{\mathcal{D}}$	ΔM_{IARD}
3dtube	45332	**3213618**	43029	129084	2	**43036**
bcsstk14	1806	**63453**	1665	4995	19	**1741**
bcsstk17	10973	**428650**	9466	28398	9	**9510**
bcsstk29	13992	**619488**	12989	51956	10	**13029**
nasasrb	54872	**2677324**	47476	142426	11	**47515**
s3dkq4m2	90451	**4820892**	90299	270894	3	**90310**

Tabla 2.4: Eficiencia de la representación IARD.



(a) Matriz *bcsstk*29 (b) Matriz *bcsstk*17

Figura 2.33: Representación por *slices*.

mente, mostramos dos nuevos parámetros asociados a la representación IARD:

- $N_{\mathcal{D}}$: Número de puntos dominantes.
- ΔM_{IARD}: Número de entradas requeridas para almacenar la caracterización IARD.

Hay que destacar que en todos los ejemplos considerados utilizamos el mismo número de puntos dominantes en la envolvente superior e inferior. Adicionalmente, no consideramos el coste asociado al almacenamiento del contexto, dado que su coste es mucho menor que el coste del resto de la caracterización. Se puede apreciar que $N_{\mathcal{D}} \ll N_S \ll N_x$ y $\Delta M_{\mathsf{IARD}} \ll \Delta M_{slices}$ para todos los casos.

En la Figura 2.33(a) se muestra un tramo de una representación por *slices* correspondiente al patrón de de la matriz *bcsstk*29. En dicha figura aparecen representados con puntos oscuros cada uno de los accesos de la indirección y con un tono más claro los que pertenecen a los vectores u o l. Se puede apreciar cómo estos puntos conforman dos curvas irregulares que acotan el patrón de acceso. En esta misma figura se representan las curvas $\mathcal{E}_f^u$ y $\mathcal{E}_f^l$ asociadas a cada uno de los vectores. Una descripción más detallada de estas curvas se muestra en la Figura 2.33(b). En ella se representa, una sección de un patrón de acceso de la matriz *bcsstk*17, junto con las curvas $\mathcal{E}_f^u$ y $\mathcal{E}_f^l$ derivadas. Para la curva $\mathcal{E}_f^l$ se muestran los puntos e_i y segmentos f_i que la conforman.

La Figura 2.34 muestra los elementos de u y l, y las curvas $\mathcal{E}^u$ y $\mathcal{E}^l$ asociadas al patrón de acceso de cada una de estas indirecciones. En todos los casos las curvas envolventes fueron obtenidas sobre el patrón de acceso transformado. A pesar de que el proceso de caracterización se realiza de forma automática, el número de puntos dominantes puede

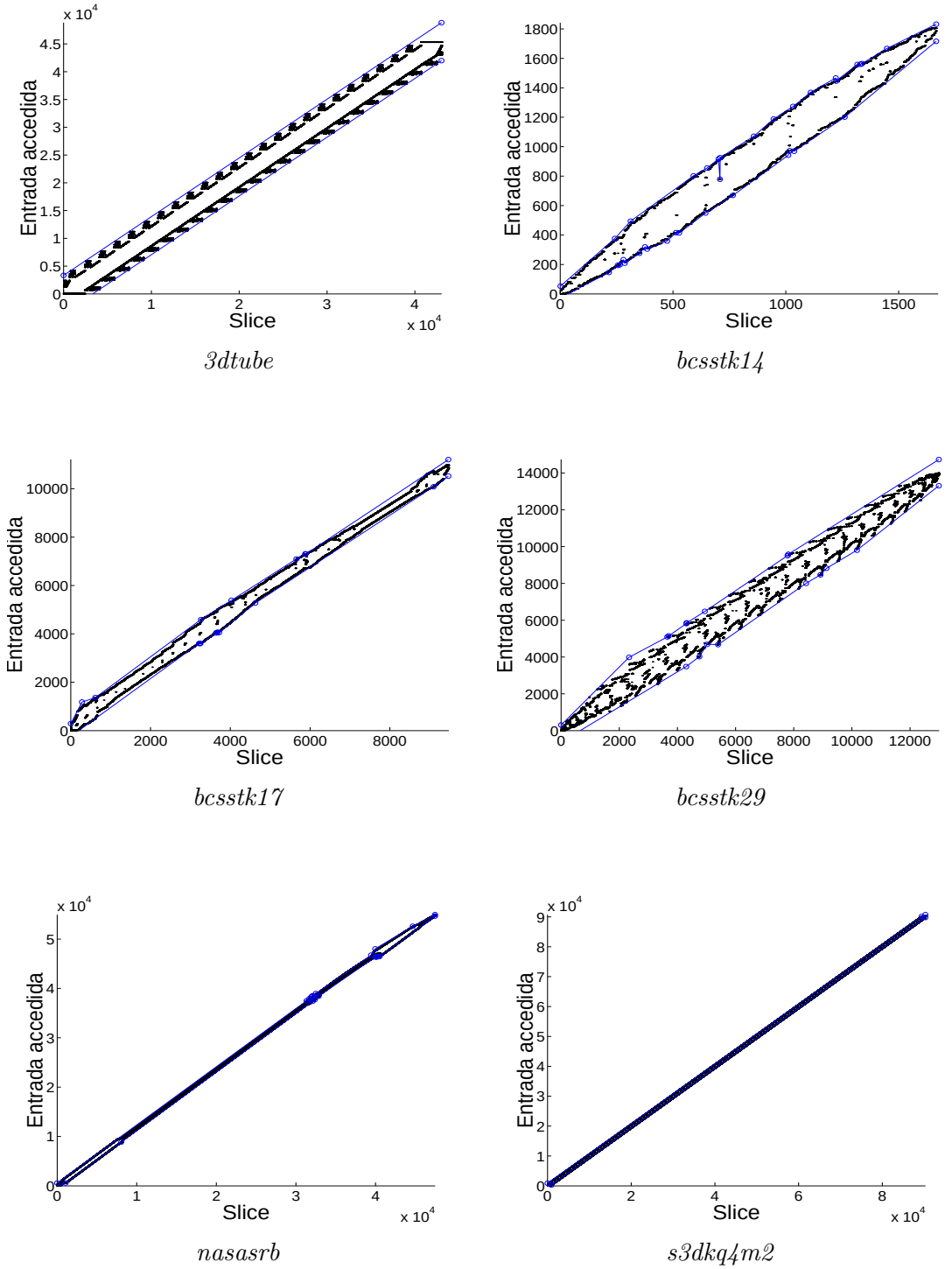


Figura 2.34: Patrón de acceso y representación IARD de diversas indirecciones.

ser muy elevado. Esto se debe a que en caso de no recibir ningún parámetro de entrada, el algoritmo `EH` genera tantos puntos dominantes como cambios de curvatura tiene el contorno. Experimentalmente, el patrón de acceso es muy abrupto, lo que origina un gran número de puntos dominantes. Empleando la transformación de desplazamiento por *slice*, los mejores rendimientos de nuestra propuesta se obtienen para un reducido número de puntos dominantes. De este modo, para cada una de las matrices elegimos un valor arbitrario (aunque reducido) de puntos dominantes. El único factor que influye en la elección de este número son las características del patrón de acceso expresadas en términos de suavidad de las curvas asociadas a su contorno. Para algunas matrices, estas curvas no presentan grandes irregularidades, por lo que basta un número muy pequeño de puntos dominantes. Este es el caso de la matriz *s3dkq4m2*. En otras situaciones con contornos más irregulares, se requiere un mayor número de puntos para obtener una caracterización precisa. Este es el caso de las matrices *bcsstk14*, *bcsstk17*, *bcsstk29* y *nasasrb*. Finalmente, existe una tercera situación en donde el contorno resulta tan irregular que harían falta un gran número de puntos dominantes para obtener una caracterización totalmente precisa. Un ejemplo es la matriz *3dtube*. En este caso, decidimos utilizar únicamente dos puntos dominantes, ya que aunque el área de error resulte muy elevada, la caracterización logra extraer las características más importantes del patrón de acceso.

La Tabla 2.5 muestra el tiempo de cálculo de la representación IARD expresado en ms. Como plataforma se utilizó el sistema Sun Enterprise 250. El código está escrito en C y se utilizó como compilador el WorkShop `cc` v4.2. En la tabla aparecen desglosados los tiempos de ejecución de cada una de las etapas. Se puede apreciar como la complejidad del algoritmo `CS` depende casi completamente del número de entradas del vector de indirección. Por contra, la caracterización geométrica guarda una relación más irregular. Dado que estamos empleando el algoritmo heurístico, resulta muy difícil establecer *a priori* el coste del mismo, ya que este depende básicamente de la estructura topológica del contorno. El coste del algoritmo `EH` puede reducirse a aproximadamente la mitad teniendo en cuenta que el proceso de obtención de las envolventes superior e inferior es independiente y pueden ser calculadas en paralelo en un sistema multiprocesador.

	3dtube	*bcsstk14*	*bcsstk17*	*bcsstk29*	*nasasrb*	*s3dkq4m2*
`CS`	556	10	73	106	454	814
$\texttt{EH}^u$	1116	49	117	241	841	704
$\texttt{EH}^l$	1192	35	118	244	617	708
Total	2865	94	308	591	1912	2226

Tabla 2.5: Tiempo de cálculo (ms) de la representación IARD.

Una vez descrito el proceso de generación de la representación IARD, nos centramos en describir las distintas aplicaciones a las que la podemos destinar. En los siguientes capítulos introducimos diversas propuestas que, mediante el empleo de esta representación, realizan diversos procesos de optimización del rendimiento de códigos irregulares.

Capítulo 3

Detección de dependencias

En el capítulo previo se introdujo nuestra propuesta de caracterización de una indirección, describiendo los mecanismos necesarios tanto para generar la representación IARD, como para obtener a partir de la misma, el patrón de acceso de un código irregular. En este capítulo presentamos una técnica de análisis de dependencias basada en la representación IARD. Este capítulo se corresponde, en el esquema general mostrado en la Figura 1.3, al bloque denominado Análisis de dependencias. En particular, abordamos el problema de analizar y comparar el conjunto de posiciones de memoria accedidas por dos bloques de código irregular. En base a este análisis es posible establecer el conjunto de posiciones de memoria susceptibles de presentar dependencias de datos.

Durante la ejecución de un programa, el acceso múltiple a una misma posición de memoria origina dependencias de datos. Tal y como se comentó en la Sección 1.4 del Capítulo 1, para cada par de accesos sobre una misma posición se pueden originar tres tipos diferentes de dependencias: dependencias verdaderas, dependencias de salida y antidependencias. El concepto de dependencia de datos en un lazo puede ser extendido al producido entre distintas secciones de código. En este caso tendremos distintas secciones de código que acceden a las mismas posiciones de memoria, lo cual acarrea la existencia de dependencias entre ambas secciones de código. El número y tipo de dependencias que es necesario contemplar depende de varios factores, como la existencia de solape entre posiciones de memoria accedidas por ambas secciones de código, el tipo de accesos realizados y la granularidad del paralelismo extraído.

Consideremos el código de la Figura 3.1, en donde se muestran dos secciones de código correspondientes a dos lazos irregulares etiquetados como `S1` y `S2`. El primero de ellos realiza un acceso de escritura sobre la matriz a por medio de la indirección x_1, mientras

```
S1: DO j = 1, N_x1
        a[x1[j]] = ...
    END DO
    ...
S2: DO j = 1, N_x2
        ... = a[x2[j]]
    END DO
```

Figura 3.1: Ejemplos de secciones de código irregular.

que el segundo accede a la misma matriz por medio de x_2 mediante operaciones de lectura. Un paralelismo de grano suficientemente grueso puede hacer que ambos lazos se asignen al mismo procesador, de modo que el resto de los procesadores del sistema queden implicados en la ejecución de otras secciones diferentes de código. En este caso la ejecución de ambos lazos es completamente secuencial, por lo que no existe riesgo de dependencias de datos. En una situación diferente, considerando una granularidad suficientemente fina, la extracción del paralelismo se realiza a nivel de lazo. En el ejemplo de la figura, inicialmente las iteraciones del lazo de la sección `S1` son ejecutadas en paralelo, a continuación se realizaría una operación de sincronización y posteriormente se ejecutaría en paralelo el lazo de `S2`. De este modo, ambos lazos se ejecutan de forma consecutiva, por lo que de nuevo se preservan las posibles dependencias de datos. Un tercer escenario se daría con la ejecución simultánea (en paralelo) de ambos lazos. Esta es la situación vamos a considerar en este capítulo. Para este escenario, existe riesgo de no preservar las dependencias entre ambas regiones, por lo que resulta necesario aplicar un procedimiento de detección y análisis de dependencias. Este proceso se debe realizar en tiempo de ejecución, por lo que el coste computacional que tiene asociado supone un factor crítico en la eficiencia del programa paralelo.

La organización de este capítulo es la siguiente: en la Sección 3.1 se contextualiza el problema que estamos abordando, y se realiza una revisión bibliográfica de otros trabajos relacionados con la detección automática de dependencias. La aplicación de la representación IARD para el análisis de dependencias de códigos irregulares se aborda en la Sección 3.2, en donde también se introducen diversos conceptos que permiten especificar los requisitos que debe cumplir nuestra propuesta para realizar el análisis de dependencias. Dicha propuesta, denominada algoritmo `DS`, es presentada en la Sección 3.3 y evaluada en la Sección 3.4.

3.1 Técnicas de detección de dependencias de datos

Las primeras técnicas de análisis de dependencias se restringen a códigos regulares. Una de las primeras propuestas es la librería Omega [113, 114, 78] que contiene procedimientos que permiten realizar diversos procesos de análisis de dependencias y de transformaciones en el código. Algunas herramientas de paralelización automática, como el compilador Polaris, incluyen técnicas propias de análisis de dependencias. En concreto, Polaris emplea el *range test* [19, 20], el cual es una extensión del análisis simbólico del *Triangular Banerjee Inequalities test* [10]. Mediante el *range test* se puede obtener las dependencias de datos existentes entre dos regiones de código regular que presentan un esquema de acceso simple[1]. Esta estrategia es, para un gran número de situaciones, más exacta que la ofrecida por librería Omega. Sin embargo, el *range test* presenta dos inconvenientes: es incapaz de analizar estructuras de indireccionamientos complejas (aún tratándose de códigos regulares), y no puede realizar un análisis de dependencias interprocedural. Con el fin de superar estas carencias, en [67] se propone un esquema, denominado *Memory Classification Analysis* (MCA) en el que se desarrolla un entorno para la detección de dependencias en lazos regulares. Este esquema establece una clasificación de los accesos a memoria realizados por cada sección del código y aplica reglas para detectar las dependencias que se pueden originar. El rango de aplicación del MCA es muy amplio y abarca desde el análisis de dependencias entre distintas secciones de código, hasta el producido en la ejecución de las iteraciones de un lazo. Adicionalmente, esta propuesta da soporte para la realización de un análisis interprocedural de dependencias. En el caso de lazos regulares, el MCA emplea el *Access Region Descriptor* ARD (previamente introducido en la Sección 2.2) para caracterizar los accesos a memoria y el *Access Region Test* (ART) [68] para el análisis de dependencias. Mediante el *Access Region Test* las entradas de memoria accedidas por varias secciones del programa son detectadas y se derivan las posibles dependencias de datos que tienen asociadas. Esta comprobación consiste, básicamente, en la realización de comparaciones entre los descriptores LMAD asociados a las distintas regiones del código. En dichas comparaciones se evalúan las regiones de memoria asociadas a cada representación, obteniendo el conjunto de entradas compartidas por ambas.

Todas estas propuestas están orientas al análisis de dependencias de códigos regulares. Dicho análisis se realiza en tiempo de compilación y conlleva, en algunos casos, un importante tiempo de cálculo. Nuestra propuesta aborda el análisis de dependencias de códigos irregulares. Debido a las características de estos códigos, es necesario realizar el análisis

[1] Típicamente, las regiones consisten en lazos con múltiple nivel de anidamiento y cuyos accesos son una combinación lineal de los índices de los lazos.

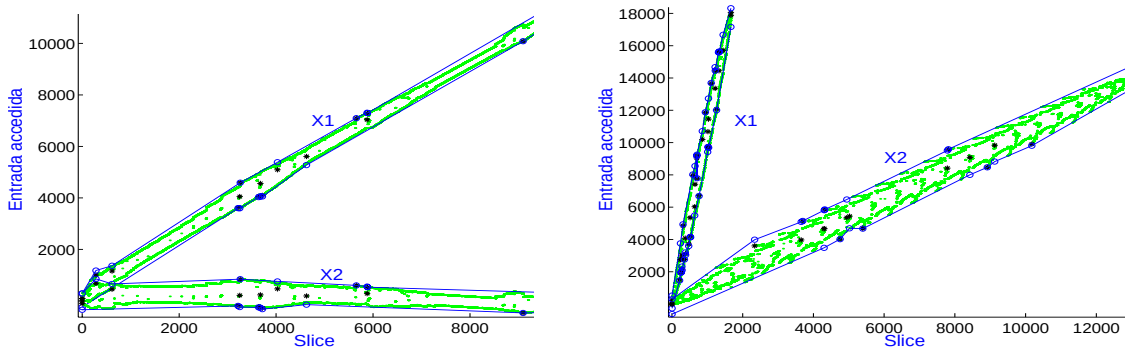


Figura 3.2: Ejemplos de cálculo de la región de solape entre dos representaciones IARD.

en tiempo de ejecución, siendo uno de los requisitos críticos la minimización del tiempo de análisis.

El problema se puede plantear como el establecimiento de un criterio de detección de la región de solape entre dos representaciones IARD. Inicialmente, consideraremos un código como el mostrado en la Figura 3.1, en donde todas las posibles dependencias de datos entre `S1` y `S2` están asociadas a los accesos a la matriz a por medio de los vectores x_1 y x_2. En un entorno de paralelización automática, la información acerca del tipo de acceso realizado en cada región se obtiene a partir de la información del contexto. Aunque la generalización a otro tipo de situaciones resulta inmediata, en este ejemplo la sección `S1` realiza una escritura y la `S2` una lectura. Por lo tanto, si ambos vectores tienen al menos un valor común, existirá una dependencia verdadera entre ambos lazos.

3.2 Región de solape entre dos representaciones IARD

Nuestro objetivo se puede plantear como la resolución del siguiente problema: dadas dos representaciones IARD asociadas a los patrones de acceso de dos secciones de código irregular, deseamos obtener el conjunto de posiciones de memoria en las que puede existir accesos comunes. Un ejemplo clarificador se puede ver en la Figura 3.2(a), en la que se representa el patrón de acceso asociado a dos indirecciones x_1 y x_2. La primera indirección fue generada en base a la matriz *bcsstk17* de la librería Harwell-Boeing. La segunda de ellas, asociada a x_2, fue generada aplicando sobre x_1 una operación de desplazamiento por *slice*. Para este ejemplo, resulta sencillo obtener una estimación de la región de solape entre ambas indirecciones: basándonos en la representación de la figura, se puede

derivar que todas las entradas de x_1 pertenecientes a un *slice* con un valor superior a 1000 no originarán dependencias con x_2, dado que acceden a un intervalo de a sobre el que x_2 no realiza ningún acceso. Este intervalo se corresponde con el conjunto de entradas $(1000, 11000]$. Del mismo modo, para aquellas entradas de x_1 pertenecientes a un *slice* inferior a 1000, existe un riesgo de "solape" con cualquier entrada de x_2. Visto de otro modo, en el intervalo $[1, 1000]$ del vector a hay riesgo de que existan dependencias verdaderas.

Nótese que este análisis ha sido realizado de forma cualitativa, considerando únicamente la representación geométrica del patrón de acceso. Consideremos ahora las representaciones mostradas en el ejemplo de la Figura 3.2(b). La primera de ellas, asociada a x_1, fue obtenida a partir de la matriz *bcsstk14* realizando una operación de multiplicación por un factor 10. La indirección asociada a x_2 fue generada en base al vector de filas de la matriz *bcsstk29*. Ambas matrices pertenecen a la librería Harwell-Boeing. Se puede apreciar que cada *slice* de x_1 comparte la región de accesos con un número significativo de los *slices* asociados a x_2. Adicionalmente, y dado que el número de *slices* de ambas representaciones es diferente, la comparación basada en la superposición geométrica de las representaciones no resulta inmediata, siendo ahora mucho más difícil determinar la región de solape entre ambas representaciones. Este hecho motiva el desarrollo de un método general para la determinación de las regiones de solape entre dos indirecciones.

El conocimiento de las entradas de a sobre las que existen dependencias, o de los índices de x_1 y x_2 asociados al mismo acceso implica, o bien el análisis exhaustivo de ambos vectores de indirección, o el diseño de un nuevo mecanismo de caracterización. Se podría, por ejemplo, almacenar en un nuevo vector auxiliar de dimensión N_a el número de accesos sobre cada entrada del vector a_i. Esto se realizaría en el momento de aplicar nuestro esquema de caracterización. Con esta nueva estructura podrían determinarse aquellas entradas del vector a que originan dependencias. Inicialmente descartamos esta solución inmediata, dado que tiene asociado un cierto coste y complejidad computacional. Adicionalmente, esta técnica no es capaz de acotar las entradas de los vectores de indirección que pueden originar conflictos, lo cual supone una importante limitación.

En esta sección, únicamente vamos a considerar una propuesta que realiza la clasificación de las regiones del patrón de acceso en dos categorías: las libres de dependencias y aquellas en las que puede existir riesgo de solape. Más concretamente, nuestra propuesta debe ajustarse a los siguientes requisitos:

1. Debe utilizar únicamente la información almacenada en la caracterización IARD. Concretamente, debe operar con las envolventes superiores e inferiores con el fin de

reducir al máximo el coste computacional del proceso de análisis.

2. Debe determinar si existe o no posibilidad de solape entre dos representaciones IARD.

3. En el caso de existir la posibilidad de una región de solape, debe acotar con la máxima precisión el conjunto de entradas de ambas indirecciones que pueden originar dependencias.

4. En el caso de existir riesgo de dependencias, debe especificar la clase de dependencia que puede existir.

En base a estos requisitos hemos desarrollado un algoritmo eficiente para la determinación de la región de solape. Dicho algoritmo es descrito de forma detallada en la siguiente sección.

3.3 Algoritmo para la determinación de la región de solape

Nuestra propuesta recibe como argumento de entrada las representaciones IARD asociadas a las dos secciones de código sobre las que se quiere determinar la presencia de dependencias de datos. Vamos a denotar por S1 y S2 a cada una de las secciones, y por $\mathsf{IARD}_{\mathsf{S1}}$ y $\mathsf{IARD}_{\mathsf{S2}}$ a sus respectivas caracterizaciones. Con el fin de explotar la estructura de dicha caracterización, nuestra propuesta divide los patrones de acceso en intervalos de *slices* denominados secciones lineales. Una definición de las mismas se enuncia a continuación.

Definición 3.3.1 Dada una representación IARD, definimos la i-ésima **sección lineal del patrón de acceso**, y la denotamos por $\mathcal{SL}_i$, como al intervalo de *slices* $[s_i^{min}, s_i^{max}]$ tal que:

1. No existe ningún punto perteneciente a la envolvente superior o inferior que posea una abscisa dentro del intervalo $[s_i^{min}, s_i^{max}]$.

   Más formalmente: $\nexists e_i \in \{\mathcal{E}^u, \mathcal{E}^l\} \ / \ e_{x_i} \in (s_i^{min}, s_i^{max})$.

2. Dos secciones lineales consecutivas comparten el primer y último *slice* de sus respectivos intervalos. Además, este *slice* compartido coincide con el valor de la abscisa de un elemento de $\mathcal{E}^u$ o $\mathcal{E}^l$.

   Más formalmente: $s_i^{min} = s_{i-1}^{max}$ y $\exists e_i \in \{\mathcal{E}^u, \mathcal{E}^l\} \ / \ e_{x_i} = s_i^{min}$.

3. Sea $N_{\mathcal{SL}}$ el número total de secciones lineales de patrón de accesos. Entonces se debe verificar que: $s_1^{min} = 1$ y $s_{N_{\mathcal{SL}}}^{max} = N_S$.

□

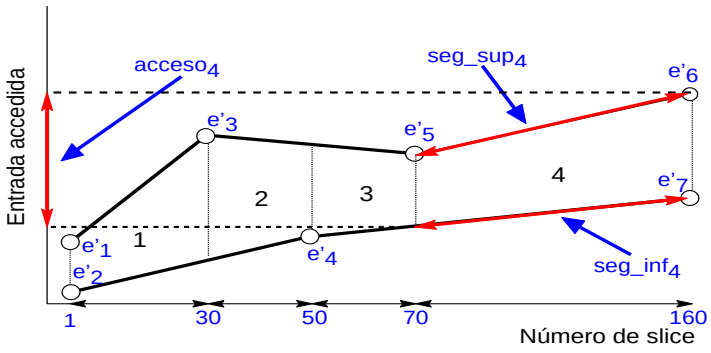


Figura 3.3: Descomposición en secciones lineales de una representación IARD.

El primer apartado de esta definición asegura que, en el interior de una sección lineal, la envolvente superior e inferior del patrón de accesos está siempre comprendida por un segmento. El segundo y tercer apartado de la definición aseguran que el conjunto de secciones lineales abarca el intervalo completo de *slices* de la representación IARD. A continuación, en la siguiente definición, introducimos conceptos que resultarán de gran utilidad a la hora de formalizar el algoritmo.

Definición 3.3.2 Dada una sección lineal $\mathcal{SL}_i^{S1}$, asociada a una representación IARD_{S1}. Definimos los siguientes conceptos:

- **Segmento superior de la sección lineal**, denotado $seg\_sup_i^{S1}$. Es el segmento que acota superiormente el patrón de acceso. La ecuación del mismo coincide con la del segmento f_i asociado a la envolvente superior $\mathcal{E}^u$. Su cálculo es muy sencillo, dado que está determinado por la ecuación de la recta que une los elementos de $\mathcal{E}^u$ entre los que la sección lineal está definida.
- **Segmento inferior de la sección lineal**, denotado $seg\_inf_i^{S1}$. Es el segmento que acota inferiormente el patrón de acceso. Del mismo modo que en el caso anterior, la ecuación de este segmento se obtiene trivialmente en función de los puntos asociados a la envolvente inferior.
- **Intervalo de accesos de la sección lineal**, denotado $acceso_i^{S1}$. Se corresponde con el intervalo de posiciones de memoria accedidas por la sección lineal. La determinación de este intervalo es inmediata, dado que basta con evaluar, respectivamente, el valor máximo y mínimo del segmento superior e inferior.

□

La Figura 3.3 muestra un ejemplo de estos conceptos. En este ejemplo, el patrón de accesos está dividido en cuatro secciones lineales, las cuales aparecen etiquetadas en la

figura como 1, 2, 3 y 4. En esta figura se representa el segmento superior e inferior y el intervalo de accesos para la última de estas secciones. Nótese que el intervalo de accesos corresponde a un conjunto de posiciones de a, por lo que aparece indicado sobre el eje de ordenadas.

En base a estas definiciones hemos desarrollado un algoritmo (algoritmo DS) que encuentra las regiones de solape entre dos representaciones IARD. El algoritmo divide cada una de las caracterizaciones IARD en secciones lineales. Empleando las propiedades de dichas secciones, se realiza una comparación entre cada una de las secciones de la primera caracterización con todas las secciones de la segunda. Con el fin de reducir el coste del proceso, la comparación se realiza utilizando únicamente la información contenida en las envolventes $\mathcal{E}^u$ y $\mathcal{E}^l$ de la caracterización. El pseudocódigo del algoritmo se ilustra en la Figura 3.4.

El algoritmo DS parte inicialmente de las representaciones IARD de las regiones S1 y S2 que se desean evaluar. El primer paso consiste en la extracción de las secciones lineales de cada una de las representaciones. Denotamos por $\mathcal{SL}^{\mathsf{S1}}$ y $\mathcal{SL}^{\mathsf{S2}}$ al conjunto de secciones lineales asociado, respectivamente, a $\mathsf{IARD}_{\mathsf{S1}}$ y $\mathsf{IARD}_{\mathsf{S2}}$. La elaboración de este conjunto es realizado por la rutina *obtiene_*$\mathcal{SL}$. El funcionamiento de esta rutina es simple: la función recibe como argumento la representación IARD considerada y accede a los campos $\mathcal{E}^u$ y $\mathcal{E}^l$. Asumiendo que el número de elementos de la envolvente superior e inferior coinciden, la rutina extrae los puntos $e_i^u \in \mathcal{E}^u$ y $e_i^l \in \mathcal{E}^l$, $1 \leq i \leq N_{\mathcal{E}}$. Cada uno de estos elementos consta de dos coordenadas, la abscisa que se corresponde al valor de *slice* y la ordenada que está asociada al valor de la entrada accedida. El algoritmo *obtiene_*$\mathcal{SL}$ realiza una copia de estos puntos en un nuevo conjunto, cuyos elementos son los elementos anteriores ordenados de forma creciente en función del valor de su abscisa. Denotaremos a dicho conjunto como $\{e'_j,\ 1 \leq j \leq 2N_{\mathcal{E}}\}$. La siguiente propiedad muestra una técnica simple para la obtención de las secciones lineales del patrón de acceso a partir de este conjunto.

Propiedad 3.3.1 Los elementos e'_j determinan los límites de las secciones lineales del patrón de acceso. Más concretamente, se verifica que:

$$s_j^{min} = e'_{x_j} \qquad \forall j \in [1, 2N_{\mathcal{E}} - 1] \tag{3.1}$$

$$s_j^{max} = e'_{x_{j+1}} \qquad \forall j \in [1, 2N_{\mathcal{E}} - 1] \tag{3.2}$$

DEMOSTRACIÓN: La demostración de esta propiedad se va a realizar comprobando si el conjunto de puntos e'_j verifica cada uno de los criterios utilizados en la Definición 3.3.1.

1. Dado que este conjunto representa un ordenamiento de los puntos de acuerdo con su índice

```
Algoritmo DS
entrada
        {IARD_S1,IARD_S2}: caracterizaciones IARD de las regiones S1 y S2
salida
        {[s1, s2]i,j, [s3, s4]j,i}: intervalos de slices de la región de solape
inicio del algoritmo
        SL^S1 ← obtiene_SL(IARD_S1)
        SL^S2 ← obtiene_SL(IARD_S2)
         DO EACH  SL_i ∈ SL^S1
            DO EACH  SL'_j ∈ SL^S2
L1              t1 = check(acceso_i^S1 ⋂ acceso_j^S2 = ∅)
                IF(t1 = cierto)
L2                  {[s1, s2]i,j, [s3, s4]j,i} = {∅, ∅}
                ELSE
L3                  lim_sup = max(acceso_i^S1 ⋂ acceso_j^S2)
L4                  lim_inf = min(acceso_i^S1 ⋂ acceso_j^S2)
                    P = intersección(SL_i, lim_sup, lim_inf)
                    P' = intersección(SL'_j, lim_sup, lim_inf)
                    [s1, s2]i,j = [⌊min(P)⌋, ⌈max(P)⌉]
                    [s3, s4]j,i = [⌊min(P')⌋, ⌈max(P')⌉]
                END IF
            END DO
         END DO
fin del algoritmo
```

Figura 3.4: Algoritmo de determinación de solape (algoritmo DS) entre dos representaciones IARD.

de *slice*(es decir, su ordenada), se asegura la no existencia de un punto de la envolvente entre dos valores consecutivos de e'_j.

2. Trivialmente, de acuerdo con su propia definición, se verifica que:
   $s_i^{min} = s_{i-1}^{max} \quad \forall i \in [2, 2N_{\mathcal{E}} - 1]$.

3. Dado que tanto el primero como el último punto de $\mathcal{E}^u$ y $\mathcal{E}^l$ tienen abscisas con valor respectivo de 1 y N_S, estos puntos formarán parte de los primeros y últimos elementos de e'_j, verificándose que: $s_1^{min} = 1$ y $s_{N_{S\mathcal{L}}}^{max} = N_S$.

□

	$i=1$	$i=2$	$i=3$	$i=4$
s_i^{min}	1	30	50	70
s_i^{max}	30	50	70	160

Tabla 3.1: Valores de s_i^{min} y s_i^{max} asociados al ejemplo de la Figura 3.3.

En el ejemplo de la Figura 3.3 se incluye el conjunto reordenado e'_j de puntos pertenecientes a las envolventes. Adicionalmente, aparece representado el índice de *slice* correspondiente a cada uno de estos puntos. La Tabla 3.1 contiene una lista de los valores de s_i^{min} y s_i^{max} asociados a cada sección lineal para este ejemplo.

Una vez determinadas las secciones lineales de las representaciones, el algoritmo `DS` comprueba la existencia de solape entre cada sección lineal de `S1` con cada una de las secciones lineales de `S2`. La detección se realiza entre cada par de secciones lineales que denominaremos $\mathcal{SL}_i$ y $\mathcal{SL}'_j$, asociadas respectivamente a `S1` y `S2`. El resultado devuelto por la comprobación de solape consiste en dos intervalos de *slices*, denominados $[s_1, s_2]_{i,j}$ y $[s_3, s_4]_{j,i}$. El primer intervalo $[s_1, s_2]_{i,j}$ hace referencia al conjunto de *slices* de $\mathcal{SL}_i$ que comparten la misma región de acceso con $\mathcal{SL}'_j$. De modo recíproco, el intervalo $[s_3, s_4]_{j,i}$ almacena el intervalo de *slices* de $\mathcal{SL}'_j$ que originan solape con $\mathcal{SL}_i$.

Para cada par de secciones lineales $\mathcal{SL}_i$ y $\mathcal{SL}'_j$, se realiza la detección de solape entre los intervalos de acceso de ambas representaciones (línea etiquetada como `L1` en el pseudocódigo). El resultado de la comprobación es almacenado en la variable lógica t_1 de modo que, en caso de no existir solape, la variable contendrá el valor `CIERTO`. En este caso, el resultado del algoritmo serán los intervalos $[s_1, s_2]_{i,j}$ y $[s_3, s_4]_{j,i}$ vacíos (etiqueta `L2` del pseudocódigo), lo cual indica que no existen, entre ambas secciones lineales, ningún *slice* que origine solape. La Figura 3.5(a) muestra un ejemplo de esta situación. Hay que destacar que la localización de los segmentos asociados a cada sección lineal no tienen por qué coincidir en el valor de su abscisa, aunque por simplicidad ocurra así en la Figura 3.5. Este hecho no afecta a nuestro algoritmo, dado que únicamente comparamos los intervalos de acceso, los cuales están situados sobre el eje de ordenadas, común a ambas representaciones.

En el caso de que la variable t_1 tenga un valor *falso*, tendremos una intersección no nula entre ambos intervalos de acceso. Un ejemplo se puede ver en la Figura 3.5(b). Es necesario identificar las regiones de solape de cada representación mediante un proceso que consiste en tres pasos, descritos a continuación.

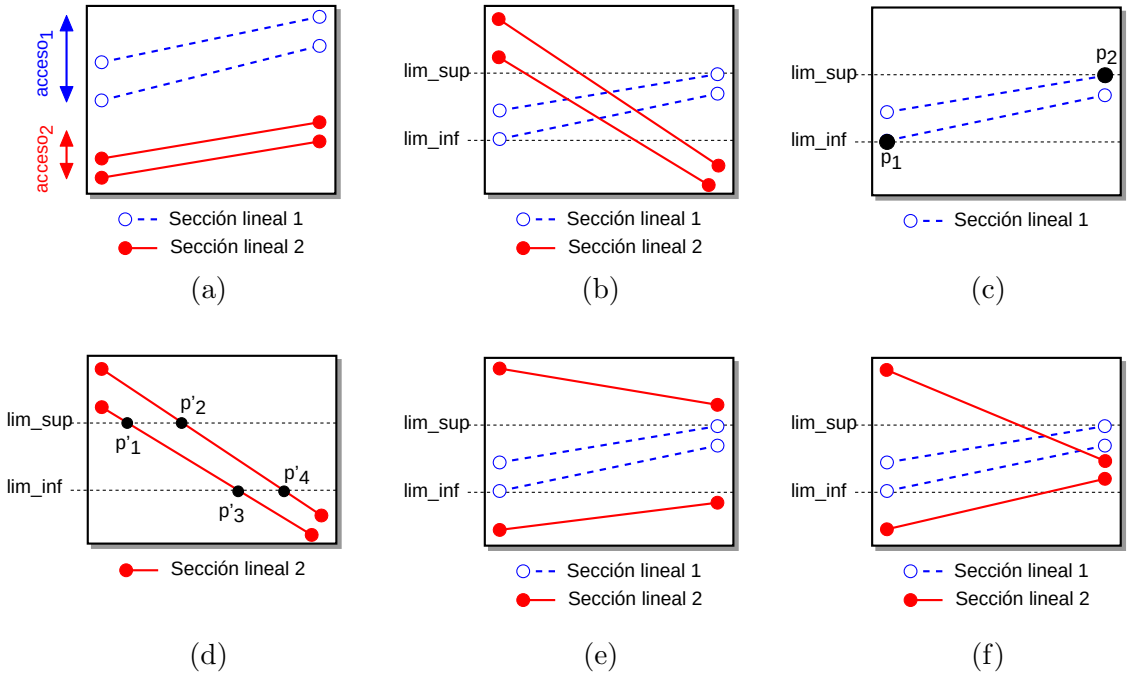


Figura 3.5: Ejemplos de cálculo de la región de solape entre dos representaciones IARD.

- **Paso 1:** determinación de la mayor y menor posición de memoria accedida en la región de solape. Dicha región, viene dada por la intersección de los intervalos de acceso asociados a ambas secciones lineales. Empleando la notación introducida en la Definión 3.3.2, la región de solape viene dada por $acceso_i^{S1} \bigcap acceso_j^{S2}$. La mayor y menor entrada de este intervalo son almacenadas, respectivamente, en las variables lim_sup y lim_inf (etiquetas `L3` y `L4` del pseudocódigo). Estos dos límites aparecen representados para el ejemplo de la Figura 3.5(b).

- **Paso 2:** obtención del *slice* en que se produce la intersección de las rectas $y = lim\_sup$ e $y = lim\_inf$ con los segmentos superiores e inferiores de ambas representaciones. Esta operación es realizada por la función *intersección*, la cual opera únicamente con las ecuaciones paramétricas de los segmentos. El resultado obtenido, expresado como un conjunto de *slices* asociados a los puntos de intersección, es almacenado en los conjuntos $\mathcal{P}$ y $\mathcal{P}'$. En las figuras 3.5(c) y 3.5(d) se ilustra este proceso. En este caso se tendría $\mathcal{P} = \{p_1, p_2\}$ para la sección lineal 1 y $\mathcal{P}' = \{p'_1, p'_2, p'_3, p'_4\}$ para la sección lineal 2.

  Existen ciertas situaciones particulares que debemos tener en cuenta.

- Consideremos la Figura 3.5(e). En este caso, las rectas $y = lim\_sup$ e $y = lim\_inf$ no intersecan los segmentos de la segunda sección lineal. El valor devuelto por la función *intersección* consiste en los *slices* que acotan a dicha sección. En el caso del ejemplo considerado, el resultado es el conjunto de *slices* extremales $\mathcal{P}' = \{s_i^{min}, s_i^{max}\}$.

- Otra situación particular se ilustra en la Figura 3.5(f). En este caso, y para la segunda sección lineal, cada una de las rectas interseca cada uno de los segmentos en un punto intermedio. Adicionalmente, para los extremos del segmento superior e inferior o bien contienen (en el extremo izquierdo) o bien están contenidos (en el extremo derecho) en el intervalo de solape. En este caso, nuevamente hay que considerar los *slices* extremales de modo que, para la segunda sección lineal, el conjunto $\mathcal{P}'$ contendrá el primer y último *slice* (*slices* extremales) de la segunda sección lineal.

- **Paso 3:** el último paso a realizar consiste en la determinación, dentro de los conjuntos $\mathcal{P}$ y $\mathcal{P}'$, de los *slices* con máximo y mínimo valor. El duplete de *slices* extraídos del conjunto $\mathcal{P}$ acotan la región de solape de la sección lineal $\mathcal{SL}_i$. Los valores almacenados en este duplete son número reales, dado que fueron obtenidos mediante la intersección de dos rectas. Así pues, es necesario realizar un redondeo por exceso o por defecto en función de que se desee obtener el valor máximo o mínimo, respectivamente. El mismo argumento puede hacerse para el conjunto $\mathcal{P}'$ asociado a la sección lineal $\mathcal{SL}'_j$. A modo de ejemplo, para las secciones lineales mostradas en la Figura 3.5(b), el resultado final es: $[s_1, s_2]_{i,j} = [p_{x_1}, p_{x_2}]$ para la Figura 3.5(c) y $[s_3, s_4]_{j,i} = [p'_{x_1}, p'_{x_4}]$ para la Figura 3.5(d). Notar que hemos extraído las abscisas de los puntos p extremales, dado que el resultado final se corresponde a un índice de *slice*.

Aplicando este esquema sobre todas las secciones lineales, el algoritmo `DS` obtiene los conjuntos de *slices* $\{[s_1, s_2]_{i,j}, [s_3, s_4]_{j,i}\}$ entre los que puede existir solape. Posteriormente, en una segunda etapa, y analizando la información del contexto asociada a cada una de las regiones, se determina el tipo de acceso (operación de lectura o escritura) que cada región realiza sobre el intervalo de posiciones de memoria considerado. En función del tipo y orden de los accesos fácilmente se pueden establecer los tipos de dependencias existentes. En la siguiente sección se analiza, desde un punto de vista teórico, la precisión y el rendimiento de nuestra propuesta.

3.4 Consideraciones de eficiencia

La primera consideración que es necesario realizar atañe a la precisión de nuestra propuesta. El empleo de la representación IARD asegura que el conjunto de accesos realizados por la indirección están acotados, para cada sección lineal, por un segmento superior y un segmento inferior. Este hecho garantiza la obtención de un resultado conservador en el sentido de que siempre se detecta un posible riesgo de dependencias. Consideremos el ejemplo de la Figura 3.5(d), dado que los segmentos acotan el patrón de accesos, se tiene que todas las entradas del vector de indirección que acceden dentro del intervalo $[lim\_sup, lim\_inf]$ lo hacen dentro del intervalo de *slices* dados por $[p'_{x_1}, p'_{x_4}]$. De este modo, la precisión del algoritmo DS se puede establecer en dos apartados:

1. Está garantizada la no existencia de dependencias de datos en las regiones del patrón de acceso clasificadas como libres de dependencias.

2. No está garantizada la existencia de dependencias en las regiones del patrón de acceso en las que se detecta riesgo de solape. La precisión del análisis dependerá tanto de la precisión en el trazo de las envolventes, como en la disposición particular de las entradas del vector de indirección.

Hay que destacar el hecho de que, en el caso de ser necesaria la determinación exacta de las entradas de los vectores de indirección que originan dependencias, mediante nuestra propuesta, y analizando el conjunto de entradas pertenecientes a los intervalos $\{[s_1, s_2]_{i,j}, [s_3, s_4]_{j,i}\}$, se puede reducir en gran medida el coste del proceso de búsqueda y comparación.

Un segundo aspecto que también es necesario considerar es la complejidad de esta propuesta. El coste del algoritmo DS puede dividirse en el coste de la función *obtiene_*$\mathcal{SL}$ y el coste del resto del algoritmo. Dado que en cada representación IARD los elementos están ordenados de menor a mayor dentro de cada envolvente (conjunto $\mathcal{E}^u$ y $\mathcal{E}^l$), la complejidad de esta función es:

$$\mathcal{O}_{obtiene\_\mathcal{SL}} = 2N_{\mathcal{E}} \tag{3.3}$$

Donde $N_{\mathcal{E}}$ es el número de elementos de los conjuntos $\mathcal{E}^u$ y $\mathcal{E}^l$. Por otra parte, y dado que las funciones *max*, *min* e *intersección* tienen una complejidad lineal, la complejidad del algoritmo DS es:

$$\mathcal{O}_{\mathtt{DS}} = 4N_{\mathcal{E}} + (2N_{\mathcal{E}})^2 \approx 4N_{\mathcal{E}}^2 \tag{3.4}$$

Hay que destacar que $N_{\mathcal{E}}$ usualmente no es muy elevado, y sus valores suelen variar de unas pocas unidades a algunas decenas. Además, el análisis del vector de indirección completo

tiene una complejidad en el caso más favorable de $\mathcal{O}(N_x)$. Este valor suele ser elevado, por lo que $\mathcal{O}_{\texttt{DS}} \ll N_x$.

Capítulo 4

Optimización de códigos irregulares con una indirección

En este capítulo presentamos distintas técnicas destinadas a obtener una ejecución eficiente de códigos irregulares. En particular, nos vamos a restringir al caso de lazos irregulares que contienen una única indirección. Los distintos procesos de optimización que vamos a presentar abarcan desde la optimización del código secuencial, hasta la paralelización automática del mismo, pasando por métodos de mejora en la localidad y de balanceo de la carga.

Las distintas propuestas que hemos desarrollado hacen uso en mayor o menor grado de la representación IARD. En algunos casos, esta representación forma parte de núcleo del algoritmo, teniendo un papel imprescindible para su correcto funcionamiento. En otros casos, únicamente se utiliza como una herramienta auxiliar que permite simplificar el coste computacional del algoritmo.

El trabajo desarrollado en este capítulo ha derivado en la siguiente serie de publicaciones: "Run-time Characterization of Irregular Accesses Applied to Parallelization of Irregular Reductions" presentada en el *Workshop on High Performance Scientific and Engineering computing with Applications in conjunction with the International Conference on Parallel Processing* en septiembre del 2001 [128]; "A run-time framework for parallelizing loops with irregular accesses" presentada en el *6th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* en marzo del 2002 [131]; "Improving load balance of the owner-computes rule in irregular reductions" presentada en el *6th World Multiconference on Systemics, Cybernetics and Informatics* en julio del 2002 [129] y "Paralelización de lazos con accesos irregulares" presentada en las *XIII*

Jornadas de Paralelismo en septiembre del 2002 [130].

La organización de este capítulo es la siguiente: en la Sección 4.1 realizamos una revisión de las técnicas existentes de paralelización, haciendo un especial hincapié en la usadas en nuestros estudios comparativos. Posteriormente, en la Sección 4.2 realizamos dos propuestas destinadas a la paralelización de códigos irregulares. Estas propuestas se corresponden, dentro del esquema general de la Figura 1.3, a las estrategias `PRT` y `SPRT` del módulo Paralelización de lazos con una indirección. Ambas propuestas están destinadas a la extracción eficiente del paralelismo de lazo y a la obtención de una alta localidad en los accesos. En esta misma sección se presentan resultados que avalan su eficiencia. La Sección 4.3 adapta la técnica `SPRT` a códigos secuenciales, proponiendo un nuevo esquema de acceso que permite explotar la jerarquía de memoria del sistema. Esta propuesta se corresponde al módulo Localidad de códigos secuenciales de la Figura 1.3. Finalmente, en la Sección 4.4 presentamos una nueva estrategia destinada a obtener una distribución balanceada de la carga computacional. Esta propuesta, denominada `BLNC` en el módulo Balanceo de la carga de la Figura 1.3, puede ser aplicada tanto a las estrategias `PRT` y `SPRT` como a otras técnicas de paralelización de códigos irregulares.

4.1 Trabajo previo

Hemos clasificado en dos familias los trabajos relacionados con la paralelización de códigos irregulares. Por una parte, tenemos un conjunto de librerías destinadas a la paralización de códigos irregulares que, sin llegar a ser una herramienta de paralelización automática, dan soporte para la realización de distintas técnicas de optimización y paralelización. Estas librerías comprenden un conjunto de primitivas que explotan la arquitectura del sistema utilizado, diversas técnicas de optimización y un entorno de interacción con el usuario.

En un segundo grupo incluimos diversas técnicas destinadas a la paralelización de ciertas clases de códigos irregulares sobre arquitecturas concretas. Dentro de esta categoría se encuentran las técnicas de paralelización más recientes y competitivas.

4.1.1 Librerías para la paralelización de códigos irregulares

Las primeras librerías destinadas a dar soporte en la paralelización de un programa fueron diseñadas para tratar únicamente códigos regulares. El compilador **Kali** [84, 83] fue la primera herramienta en soportar estructuras de código tanto regulares como irregu-

lares sobre sistemas de memoria distribuida. Mediante el empleo de esta herramienta, la memoria física se ofrece al usuario como un espacio virtual de memoria compartida. De este modo, el usuario debe realizar las distribución de los datos sobre el espacio virtual, e indicar de forma manual los lazos que desea paralelizar. Esta herramienta soporta distintos esquemas de distribución de datos, como pueden ser por bloques, cíclicos e irregulares. Posteriormente, en tiempo de ejecución, el compilador introduce un inspector que realiza las operaciones de distribución y comunicación necesarias para ejecutar en paralelo las secciones de código especificadas.

Otra propuesta pionera en dar soporte para la distribución irregular de matrices es el **ARF** [123]. Este compilador emplea las primitivas PARTI [32] para manejar estructuras de código dispersas y no estructuradas. Adicionalmente contempla distribuciones arbitrarias de los datos y contiene un conjunto de técnicas de optimización destinadas a estructuras de código concretas. Esta propuesta aplica un esquema basado en el esquema inspector-ejecutor, y su principal aportación consiste en su capacidad de aplicar las primitivas PARTI de forma totalmente automática.

La librería PARTI fue posteriormente sustituida por la librería **CHAOS** [147, 125], la cual permite abordar un mayor número de situaciones. Esta librería está orientada a sistemas de memoria distribuida, siendo una de sus principales características la capacidad para paralelizar códigos irregulares en los que los patrones de acceso se modifican a lo largo de la ejecución del programa. En [69] se propone una nueva estrategia de paralelización de códigos irregulares basada en CHAOS que aborda la paralelización de lazos irregulares con varias indirecciones [22, 144]. La librería CHAOS resulta relevante debido a su gran popularidad y a que muestra un ejemplo de esquema de paralelización sobre un sistema de memoria distribuida. Por este motivo, y con el ánimo de establecer una comparativa con otras estrategias orientadas a sistemas de memoria compartida, vamos a describir de forma más detallada las fases de las que consta.

1. *Particionamiento de los datos.* En esta fase se establece el particionamiento de los datos asociados al código considerado. La librería **CHAOS** admite distribuciones estándares de los datos, así como distintas heurísticas de distribución específicas para problemas concretos.

2. *Distribución de los datos.* En esta fase se genera el esquema de comunicaciones necesario para llevar a cabo la distribución de los datos.

3. *Particionamiento de las iteraciones.* El grano empleado por CHAOS tiene un tamaño de iteración. En esta fase se asignan las distintas iteraciones a los procesadores. La

estrategia de distribución empleada se basa en una variante de la regla del propietario, de modo que la asignación de las iteraciones se realiza en función del tipo de distribución de los datos empleada.

4. *Distribución de las iteraciones.* Una vez determinada la partición de las iteraciones y de los datos, se generan listas que contienen los procesadores propietarios de cada iteración y los procesadores propietarios de cada dato distribuido.

5. *Rutina de inspección.* En tiempo de ejecución, un inspector analiza el contenido de las indirecciones y genera el esquema de comunicaciones necesario para realizar la ejecución paralela del lazo. Este proceso depende de la distribución de los datos e iteraciones [99].

6. *Rutina de ejecución.* Esta etapa realiza la ejecución paralela del lazo empleando la información generada en las etapas previas.

La aplicación de una estrategia equivalente sobre un sistema de memoria compartida resulta mucho más simple. Mediante directivas de paralelización[1] la fase de particionamiento de los datos e iteraciones se puede especificar de un modo simple. Por otra parte, la distribución de los datos e iteraciones se realiza automáticamente y de forma transparente al usuario. Finalmente, el proceso de inspección es mucho más sencillo, ya que no resulta necesario evaluar el propietario de cada dato o iteración con el fin de generar comunicaciones.

La principal causa de ineficiencia de la estrategia basada en CHAOS lo supone el coste asociado al proceso de análisis y generación del esquema de comunicación. En muchos casos, es necesario amortizar dicho coste mediante un reuso del inspector. Esta propuesta original está sujeta a diversas mejoras. En [110] se representa el espacio de accesos a memoria como un grafo sobre el que se aplican diversos algoritmos de particionamiento. Mediante esta estrategia se consigue una mejora en la localidad y una reducción en el número de comunicaciones. En este mismo contexto se sitúa la librería PILAR [87]. Esta librería da soporte a la paralelización de códigos irregulares mediante la estrategia del inspector-ejecutor. En [87] la librería se integra en el compilador PARADIGM [9], permitiendo abordar la paralelización automática de códigos irregulares. Esta librería hereda la estructura de la librería CHAOS, enriqueciéndola con nuevas características. Algunos ejemplos de las mismas son su capacidad de combinar el análisis de dependencias con

[1] La librería OpenMP no da soporte a la distribución de los datos. Sin embargo, otras librerías (como las suministradas con los compiladores de Silicon Graphics) incluyen directivas para aplicar distintos esquemas de distribución de datos y de iteraciones. Estas directivas ofrecen una gran flexibilidad en su empleo, ya que pueden ser utilizadas de forma conjunta con las directivas de OpenMP.

información de la estructura del código, el empleo de rutinas de comunicación MPI [48], la obtención de mejores esquemas de comunicación y la reducción en el coste del proceso de análisis.

Otro ejemplo de librería destinada a sistemas distribuidos es la **TreadMarks** [4, 95]. **TreadMarks** da soporte a la paralelización de códigos irregulares sin necesidad de hacer uso de un inspector, e incorpora distintas técnicas [23, 76] que permiten a los procesadores compartir los datos a lo largo de la ejecución del programa, sin la existencia de conflictos en los accesos. Por otra parte, esta librería se encarga, de modo transparente al usuario, de la distribución de los datos y las comunicaciones entre procesadores. Esta librería ofrece una abstracción del sistema de memoria ofreciendo al programador el mismo entorno de trabajo que el de las arquitecturas de memoria compartida. El empleo de esta librería favorece la portabilidad de programas entre arquitecturas de memoria distribuida y arquitectura de memoria compartida.

Finalmente, en [44] se presenta una librería denominada **RAPID** orientada a la paralelización de códigos irregulares y basada en la técnica del inspector-ejecutor. Dicha librería parte de una división del código en secciones denominadas *tasks*, las cuales definen el tamaño de grano utilizado. Esta librería está integrada en un sistema que analiza el patrón de acceso a memoria (que puede ser irregular) y genera, en base al mismo, un grafo de dependencias entre *tasks*. Empleando la información de este grafo, se realiza la distribución de *tasks*, eliminando tanto las falsas dependencias como las dependencias de salida, y planificando la ejecución paralela del código.

4.1.2 Técnicas de paralelización de códigos irregulares

Existen diversos trabajos que abordan la paralelización de códigos irregulares. Estas propuestas pueden ser agrupadas en tres grandes grupos: aquellas basadas en el uso de primitivas de sincronización, aquellas basadas en la privatización de la matriz de reducción y aquellas basadas en la estrategia del inspector-ejecutor.

Por lo general, las distintas propuestas estás orientadas a la paralelización de reducciones irregulares. Existen dos causas fundamentales. La primera se debe a que este tipo operación aparece con frecuencia en un gran número de aplicaciones. La segunda causa se debe a las propiedades conmutativas y asociativas de la operación de reducción. Gracias a las mismas pueden aplicarse técnicas de paralelización mucho más agresivas que evitan la existencia de dependencias de datos.

A lo largo de esta sección vamos a introducir distintas técnicas que serán aplicadas a

```
DO j = 1, N_x
   a[x[j]] = a[x[j]]  ⊗  ...
END DO
```

Figura 4.1: Ejemplos de código irregular con una única indirección.

la paralelización del código mostrado en la Figura 4.1. Este código contiene un lazo de N_x iteraciones en el que el vector a, de N_a entradas, es accedido por la indirección x mediante una reducción irregular. Denotamos como N_p el número de procesadores empleados en la paralelización del lazo. En las siguientes secciones describiremos las distintas propuestas de cada uno de los grupos anteriormente enumerados.

Técnicas basadas en el uso de primitivas de sincronización

Comúnmente, existen a disposición del programador librerías con primitivas de sincronización. Estas primitivas pueden ofrecerse por el propio sistema operativo, por el lenguaje de programación, o (como situación más común) estar incluidas en librerías específicas de paralelización. Los ejemplo más comunes de estas primitivas son las barreras, los *locks* y los accesos atómicos [30].

En función del tipo de primitiva de sincronización empleada, el lazo de la Figura 4.1 puede ser paralelizado mediante dos estrategias. En la Figura 4.2(a) se ilustra un ejemplo empleando *locks*. El lazo original se ejecuta ahora en paralelo, realizando una distribución de las iteraciones (que asumiremos por bloques) entre los procesadores. Cada procesador, realiza una operación tipo *lock* sobre la entrada que va a acceder. En caso de que esta entrada haya sido previamente bloqueada por otro *thread*, el procesador quedará en estado de espera hasta que dicha entrada sea liberada. Mediante el empleo de esta operación se asegura la no existencia de conflictos en los accesos sobre la entrada de a.

Otro ejemplo de paralelización se muestra en la Figura 4.2(b). El concepto básico de funcionamiento es el mismo que en el caso anterior. La única diferencia es que ahora se garantiza la exclusión haciendo que cada acceso irregular se realice de forma atómica. Por norma general, las técnicas de paralelización basadas en el uso de primitivas de sincronización sólo son efectivas cuando el paralelismo es de grano grueso. La principal ventaja de estas técnicas es su capacidad de generar el código paralelo de un modo simple. Además, este código paralelo tiene unos costes de almacenamiento reducidos. Por coste de almacenamiento de un código paralelo entendemos la memoria extra requerida por dicho código respecto al programa secuencial. El principal inconveniente de esta técnica radica en el

```
SHARED a[1 : N_a]

DOALL j = 1, N_x
   lock(a[x[j]])
   a[x[j]] = a[x[j]]  ⊗  ...
   unlock(a[x[j]])
END DOALL
```

(a)

```
SHARED a[1 : N_a]

DOALL j = 1, N_x
   start_atomic
    a[x[j]] = a[x[j]]  ⊗  ...
   end_atomic
END DOALL
```

(b)

Figura 4.2: Paralelización mediante primitivas de sincronización: (a) empleo de *locks*, (b) empleo de operaciones atómicas.

coste del proceso de inicialización y sincronización. Este coste puede originar retardos comparables al tiempo de ejecución de las sentencias que conforman el cuerpo del lazo, lo que hace que su empleo resulte inviable para este tipo de situaciones.

Técnicas basadas en la privatización del vector de reducción

Dentro de las propuestas que no emplean un inspector se encuentran, como las opciones más populares, ***array expansion*** [42, 38] y ***replicated buffers*** [145]. La primera está implementada en el compilador paralelizador Polaris [18], mientras que *replicated buffers* es utilizada por el compilador Suif [55]. Ambas estrategias se basan en la privatización de las variables sobre las que se realizan escrituras. Una privatización de una variable consiste en su replicación sobre los distintos procesadores implicados en la ejecución paralela del lazo, de modo que cada uno accede sobre su copia privada. Una variable puede ser privatizada si se asegura la corrección del resultado. Es decir, si se asegura que el resultado obtenido por el programa paralelo coincide con el obtenido por el secuencial. De este modo, una variable puede ser privatizada [139] cuando toda lectura realizada sobre la misma conlleva una posterior escritura en la misma posición[2]. En particular, las variables o vectores sobre los que se realiza una operación de reducción pueden ser privatizados, dado que dicha operación es asociativa y conmutativa y además consta de una lectura previa a una escritura sobre el mismo elemento.

[2]Esto no es estrictamente cierto para algunos tipos de operaciones, como, por ejemplo, las reducciones. En estos casos se pueden asumir pequeñas discrepancias entre los valores numéricos del resultado paralelo y secuencial, las cuales se deben a la existencia de una precisión finita de cálculo por parte del sistema.

```
SHARED a[1 : N_a]
PRIVATE b[1 : N_a]

S1    PARALLEL DO i = 1, N_a
          b[i] = 0
      END PARALLEL DO

S2    DOALL p = 1, N_p
          DO j = ini[p], ini[p + 1] − 1
              b[x[j]] = b[x[j]]  ⊗  ...
          END DO
      END DOALL

S3    PARALLEL DO i = 1, N_a
          lock(a[i])
          a[i] = a[i]  ⊗  b[i]
          unlock(a[i])
      END PARALLEL DO
```

(a)

```
SHARED a[1 : N_a], b[1 : N_a, 1 : N_p]
SHARED ini[1 : N_p + 1]

      DOALL p = 1, N_p
S1        DO i = 1, N_a
              b[i, p] = 0
          END DO

S2        DO j = ini[p], ini[p + 1] − 1
              b[x[j], p] = b[x[j], p]  ⊗  ...
          END DO
      END DOALL

S3    DOALL i = 1, N_a
          DO p = 1, N_p
              a[i] = a[i]  ⊗  b[i, p]
          END DO
      END DOALL
```

(b)

Figura 4.3: Paralelización mediante privatización: (a) *replicated buffers*, (b) *array expansion*.

La Figura 4.3(a) muestra el código paralelo resultante de aplicar la técnica de *replicated buffers*. En esta figura introducimos la sentencia `PARALLEL DO` para indicar que todas las iteraciones del lazo son ejecutas en paralelo por todos los procesadores. La matriz a es privatizada, generándose N_p copias de la misma que denominaremos b. La ejecución paralela consta de tres etapas. En la primera de ellas (etiquetada como `S1` en la figura) cada procesador inicializa su copia privada al elemento neutro de la operación de reducción, este proceso se realiza en paralelo. En la segunda etapa (etiquetada como `S2`) cada procesador p ejecuta el intervalo de iteraciones $[ini[p], ini[p+1])$ y acumula en su copia privada su contribución parcial al resultado. El vector ini contiene los límites del particionamiento por bloques del espacio de iteraciones. Finalmente, en la tercera etapa (etiquetada como

`S3`) la matriz resultado es actualizada en base a las contribuciones parciales. Con el fin de evitar conflictos en las escrituras, estos accesos se realizan empleando primitivas de sincronización, que, en este ejemplo concreto son *locks*. El empleo de esta propuesta es adecuado en el contexto de sistemas de memoria distribuida, aunque también puede ser utilizada en sistemas de memoria compartida.

El código paralelo correspondiente a la técnica de *array expansion* se muestra en la Figura 4.3(b). En vez de realizarse una copia privada del vector de reducción, este se expande en una nueva matriz que tiene una dimensión adicional con tantos elementos como número de procesadores. Esta matriz se declara como compartida. El código paralelo consta nuevamente de tres etapas. En la primera de ellas (etiquetada como `S1`), cada procesador inicializa al elemento neutro de la operación de reducción los elementos de la matriz que le corresponde. En una segunda etapa (`S2`), cada procesador ejecuta el intervalo de iteraciones del lazo original que tiene asignadas. Finalmente, en la tercera etapa (`S3`), y explotando el hecho de que la matriz b es globalmente compartida, cada procesador actualiza una porción del vector original con las contribuciones de todos los procesadores. Dada su estructura, esta propuesta está orientada al empleo en sistemas de memoria compartida. El coste de almacenamiento del ejecutor (denotado como $\Delta M^{ejecutor}$) viene dado por la siguiente expresión:

$$\Delta M^{ejecutor}_{array\ expansion} = N_p N_a \tag{4.1}$$

Este coste de almacenamiento es el mismo que en la técnica de *replicated buffers*.

En términos de rendimiento, la principal diferencia existente entre ambas propuestas es que *array expansion* no emplea primitivas de sincronización. Este hecho introduce cierto grado de serialización en la ejecución paralela del lazo en el caso de *replicated buffers*. Por este motivo el rendimiento del código paralelo de *array expansion* resulta frecuentemente superior al de *replicated buffers*. La principal carencia de ambas propuestas es el elevado coste de memoria y de comunicaciones que acarrean. Estos costes aumentan conforme aumenta el número de procesadores. Adicionalmente, ambas técnicas presentan una baja capacidad de aprovechamiento de la jerarquía de memoria ya que no explotan la localidad en los accesos sobre a.

La técnica de ***replicated buffers with links*** [152] supone una mejora de las propuestas anteriores. El objetivo básico de esta propuesta consiste en la minimización del número de accesos realizados en la última etapa de la técnica de *replicated buffers*. Tal y como se comentó con anterioridad, esta última etapa tiene asociada un elevado coste computacional y es el principal factor que limita la eficiencia de esta propuesta. Los autores proponen duplicar el tamaño de los vectores privados introduciendo, para cada elemento,

un nuevo campo denominado *link*. El vector b se puede entender como una matriz en donde cada una de sus filas es una entrada de a replicada sobre los procesadores, y cada una de sus columnas se corresponde a una copia de a asociada a un procesador. El campo *link* se emplea para enlazar todos los accesos realizados sobre la misma fila de b, permitiendo que el proceso de comunicación final únicamente acceda a las entradas de b que están ocupadas. Adicionalmente, es necesario un nuevo vector de N_a entradas, que los autores denominan *head*, y que se emplea para marcar la primera entrada cubierta de cada fila de b. Esta propuesta está orientada a situaciones en las que los patrones de acceso tienen alta dispersión de las entradas, es decir, la relación N_x/N_a es reducida, lo que indica que existen muchas entradas de los vectores privados b que no están cubiertas. En estos casos, la aplicación de esta propuesta permite reducir de forma significativa el coste de las comunicaciones. Por contra, esta propuesta presenta dos desventajas. La primera de ellas es que el coste de almacenamiento aumenta respecto a *array expansion*. Específicamente, este coste viene dado por:

$$\Delta M_{links}^{ejecutor} = N_a(2N_p + 1) \tag{4.2}$$

La segunda desventaja consiste en que es necesaria una fase previa de inspección para generar los campos *link* y el vector *head*. Este inspector introduce un importante coste computacional si se compara con las técnicas anteriores, que no utilizan un inspector.

Con el fin de reducir los costes de almacenamiento, en [152] se propone una nueva estrategia, denominada ***selective privatization***, en la que únicamente se replican los elementos de a accedidos por varios procesadores. Esta estrategia consta de un inspector que divide el espacio de accesos sobre a en accesos exclusivos y compartidos. Los accesos exclusivos se corresponden a los que acceden, durante la ejecución paralela del lazo, a una entrada de a que no vuelve a ser utilizada. Este tipo de accesos no originan conflictos, por lo que pueden ser directamente realizados sobre a. Por otra parte, los accesos compartidos son los que se producen más de una vez sobre la misma posición de memoria, pudiendo originar conflictos. Por este motivo deben ser realizados sobre copias privadas a cada procesador. El tratamiento de los accesos compartidos es, en esencia, idéntico al de la técnica anterior: el vector *head* se emplea para indicar aquellos elementos de a que han sido replicados, y el campo *link* enlaza cada uno de ellos. En una etapa final de comunicación, a es actualizada mediante operaciones de sincronización con el contenido de estas copias. Esta técnica implica una clasificación de las entradas del vector de indirección en compartidas y exclusivas. El principal inconveniente de esta propuesta es el coste asociado, tanto a la rutina de inspección, como a procesamiento de los accesos privados. Otro inconveniente es el coste de almacenamiento, que en algunos casos puede resultar elevado. De forma más

concreta, este coste viene dado por la siguiente relación,

$$\Delta M^{ejecutor}_{selective\ privatization} = 2N_pN_b + N_a \tag{4.3}$$

Donde el primer término representa el coste de almacenamiento asociado a los accesos compartidos, siendo N_b el número de filas de la matriz b (que ahora es menor que a). El segundo término tiene en cuenta el coste de almacenamiento del vector *head*. Nótese que esta técnica realiza una copia del vector de indirección, pero dado que el vector de indirección original no se emplea durante la ejecución paralela del lazo, su coste no se refleja en la expresión anterior. Esta técnica únicamente es eficiente cuando la relación N_x/N_a es reducida, dado que es en estos casos cuando existe una menor proporción de entrada privadas.

Otra propuesta es la denominada ***sparse reductions with privatization in hash tables*** [152]. En vez de emplear, como hace la *selective privatization*, una copia privada de a en cada procesador, esta propuesta utiliza *hash tables* para almacenar los accesos clasificados como compartidos. Esta técnica consta de dos fases: en la primera el lazo paralelo es ejecutado, almacenando todos los accesos en *hash tables*. De forma simultánea a su ejecución, los accesos realizados en cada iteración se identifican y clasifican como compartidos o exclusivos. De este modo, cuando el lazo paralelo vuelve a ejecutarse (asumiendo reuso del inspector), únicamente se almacenan en las *hash tables* aquellos accesos clasificados como compartidos. En esta segunda fase, el vector de indirección se modifica de forma similar a la técnica anterior, permitiendo que los accesos exclusivos se almacenen directamente sobre a. Dado que esta escritura tiene un coste mucho menor que el empleo de *hash tables*, se consigue un aumento en la eficiencia del código paralelo. La siguiente relación muestra el coste asociado a esta propuesta:

$$\Delta M^{ejecutor}_{hash\ tables} = N_pN_{hash} \tag{4.4}$$

Donde N_{hash} representa el número de entradas de la *hash table*, que está replicada sobre todos los procesadores.

Finalmente, la Figura 4.4 muestra la versión paralela del código considerado mediante la aplicación de la regla del propietario. Esta técnica se basa en la distribución de las entradas de a en N_p particiones. Se puede emplear cualquier criterio para realizar dicha distribución, aunque típicamente se emplea un esquema de particionamiento por bloques. Esta técnica se basa en la privatización de las entradas de a mediante la asignación a un procesador de cada una de las particiones. De este modo, durante la ejecución paralela del lazo, cada procesador accede a un espacio privado de entradas de a.

En el de código de la Figura 4.4 forzamos esta privatización mediante la introducción de una sentencia condicional que depende del valor lógico devuelto por la función *own*. Esta

```
SHARED a[1 : N_a]

DOALL j = 1, N_x
    IF(own(x[j]))    a[x[j]] = a[x[j]]   ⊗   ...
END DOALL
```

Figura 4.4: Paralelización mediante la aplicación de regla del propietario mediante sentencias condicionales.

función recibe como argumento el índice de la entrada, y devuelve un valor lógico `CIERTO` si dicha entrada es local al procesador que está ejecutando la sentencia. Esta estrategia no requiere de una rutina de inspección, siendo fácilmente aplicable a lazos irregulares con una indirección. El coste de la función *own* es reducido, e incluso puede ser reemplazado por una máscara, o por una expresión analítica. Adicionalmente, el coste de memoria de esta técnica es nulo, ya que no requiere de ninguna estructura de almacenamiento adicional. Como principales desventajas cabe destacar que cada procesador debe recorrer el espacio completo de iteraciones del lazo, ejecutando para cada una de ellas la sentencia condicional. Ambos hechos suponen una importante limitación, especialmente cuando la carga de trabajo del lazo es reducida. En estos casos el rendimiento del código paralelo experimenta un importante descenso al aumentar el número de procesadores.

Técnicas basadas en el inspector-ejecutor

A continuación vamos a describir dos técnicas, denominadas LOCALWRITE [58, 60, 61] y DWA-LIP [51, 52], basadas en el empleo de una rutina de inspección. En esta sección vamos a introducir su aplicación a códigos irregulares con una indirección. Posteriormente, en el Capítulo 6 discutiremos su adecuación a códigos con un mayor número de indirecciones.

La Figura 4.5 muestra el inspector y el ejecutor de la estrategia LOCALWRITE. Esta estrategia se basa en la aplicación de la regla del propietario, reemplazando la sentencia condicional por un inspector que realiza la distribución de las iteraciones. En la etapa de inspección, y de forma paralela, cada procesador recorre el espacio completo de iteraciones verificando si el acceso realizado sobre a es local. En caso afirmativo el contenido de la indirección se almacena en el vector y_p, empleando la variable $cont_p$ como contador. Ambos datos (y_p y $cont_p$) son privados a cada procesador. Al finalizar la etapa de inspección cada

```
SHARED a[1 : N_a]
LOCAL  cont_p, y_p

DOALL  p = 1, N_p
    DO  j = 1, N_x
        cont_p = 0
        IF(own(x[j]))
          y_p[cont_p] = j
          cont_p + +
        END IF
    END DO
END DOALL
```

(a)

```
SHARED a[1 : N_a]
LOCAL  cont_p, y_p

DOALL  p = 1, N_p
    DO  k = 1, cont_p
        a[y_p[k]] = a[y_p[k]]  ⊗  ...
    END DO
END DOALL
```

(b)

Figura 4.5: Paralelización mediante la técnica de LOCALWRITE: (a) inspector, (b) ejecutor.

procesador p tiene en y_p las iteraciones que debe procesar, y en $cont_p$ el número total de ellas. La técnica de LOCALWRITE maximiza la localidad en los accesos sobre a y en la lectura de y_p. Nótese que, a pesar de replicar el tamaño del vector de indirección, el volumen de datos accedido por el ejecutor paralelo es muy próximo al del algoritmo secuencial. Específicamente el coste de memoria asociado al ejecutor se debe únicamente a las N_p copias de la variable *cont*.

$$\Delta M^{ejecutor}_{LOCALWRITE} = N_p \tag{4.5}$$

La Figura 4.6 muestra el esquema de inspector-ejecutor de la estrategia DWA-LIP. En esta figura mostramos la idea básica de esta propuesta, la cual se describirá de forma más detallada en el Capítulo 6. Esta estrategia se basa también en la aplicación de la regla del propietario sobre la matriz a. Sin embargo, el mecanismo de indireccionamiento empleado es diferente al del caso anterior. En vez de almacenar en la rutina de inspección el contenido del vector de indirección, se guarda el índice de la iteración en la que se realiza el acceso. Posteriormente, en la etapa de ejecución, cada procesador recorre aquellas iteraciones que acceden a la partición de a que tiene asignada. La principal diferencia entre ambas propuestas es un mayor espacio de almacenamiento, ya que es necesario mantener el vector

```
SHARED a[1 : N_a]

DOALL p = 1, N_p
    DO j = 1, N_x
       cont_p = 0
       IF(own(x[j]))
         next_p[cont_p] = j
         cont_p ++
       END IF
    END DO
END DOALL
```

(a)

```
SHARED a[1 : N_a]

DOALL p = 1, N_p
    DO k = 0, cont_p
       j = next_p[k]
       a[x[j]] = a[x[j]]  ⊗  ...
    END DO
END DOALL
```

(b)

Figura 4.6: Paralelización mediante la técnica de DWA-LIP: (a) inspector, (b) ejecutor.

de indirección. El coste de almacenamiento de esta técnica es:

$$\Delta M_{DWA-LIP}^{ejecutor} = N_p + N_x \tag{4.6}$$

Una vez finalizada la revisión del trabajo previo vamos a presentar dos nuevas propuestas que abordan la paralelización de este tipo de lazos irregulares. Su eficiencia será comparada con algunas de las técnicas que hemos descrito en esta sección

4.2 Paralelización de lazos irregulares

El primer campo de aplicación que vamos a abordar es la paralelización de lazos irregulares. En esta sección vamos a considerar lazos en los que las dependencias son únicamente debidas a los accesos realizados por el vector de indirección. Ejemplos de este tipo de códigos son mostrados en la Figura 4.7. El código mostrado en la Figura 4.7(a) ha sido extraído del lazo `DO 50` de la rutina *rperm* del *Sparse Kit* [122]. Esta rutina realiza la permutación del vector b, escribiendo el resultado en un nuevo vector a. Cada iteración del lazo realiza un acceso de escritura sobre el vector a. Bajo esta condición, las únicas dependencias que se pueden originar son dependencias de salida asociadas a las operaciones de escritura. En el caso de la Figura 4.7(b), en cada iteración del lazo se acumula un valor sobre una entrada de a. Este tipo de estructura aparece con frecuencia en rutinas de álgebra matricial dispersa, como, por ejemplo, el producto matriz dispersa vector, el cual

```
DO j = 1, N_x
   a[x[j] + 1] = b[j − 1] − b[j]
END DO
```

(a)

```
DO j = 1, N_x
   a[x[j]] = a[x[j]] ⊙ ...
END DO
```

(b)

Figura 4.7: Códigos irregulares: (a) Rutina *rperm*, (b) Ejemplo de lazo con operación de escritura y lectura por medio de una indirección.

```
DO i = 1, N_S
     DOALL j = ρ_ini[i], ρ_ini[i + 1] − 1
           a[x[j]] = ...
     END DOALL
END DO
```

Figura 4.8: Ejemplo de lazo irregular paralelo, con ejecución guiada por *slices*.

a su vez es empleado por distintos tipos de resolutores numéricos [13]. En la situación que vamos a considerar el operador "$\odot$" es un operador genérico que no tiene por qué ser asociativo, y por consiguiente, la operación no tiene que ser una operación de reducción. Para este ejemplo, pueden existir dependencias verdaderas entre iteraciones distintas del lazo.

Mediante la representación IARD, los lazos irregulares de las figuras 4.7(a) y 4.7(b) se pueden reescribir del modo mostrado en la Figura 4.8. En donde el lazo externo recorre los *slices* asociados al patrón de indirección, mientras que el interno recorre las entradas de cada uno de los *slices*. El vector ρ_{ini}, introducido en la Sección 2.5.2, devuelve la posición de primer elemento de cada *slice*. De acuerdo con la Definición 2.3.1, los accesos pertenecientes a un mismo *slice* se realizan sobre posiciones diferentes de memoria, por lo que el lazo interno de la Figura 4.8 es completamente paralelo. En el caso del ejemplo de la Figura 4.7(a), y dado que x es un vector de permutación, todas sus entradas tienen valores distintos, de modo que la representación IARD del mismo consta de un único *slice*. De este modo, y mediante el uso de nuestra caracterización, el lazo es correctamente identificado como totalmente paralelo.

Mediante el empleo de esta estrategia de paralelización, las dependencias de datos son respetadas, asegurando la validez del resultado. Sin embargo, tiene como inconveniente que en una situación general puede existir un número elevado de *slices* lo que conlleva

a un aumento del número de operaciones de sincronización. Dicho aumento implica una pérdida de eficiencia en la ejecución paralela del código, dando lugar a una degradación de su rendimiento. Este hecho ha motivado el desarrollo de nuevas estrategias que alcancen una mayor eficiencia en la ejecución paralela del código. En esta sección presentamos dos propuestas de paralelización de códigos irregulares denominadas *Private Region Technique* (`PRT`) y *Sorted Private Region Technique* (`SPRT`). Ambas propuestas utilizan una reordenación parcial del vector de indirección con el fin de reducir los costes de sincronización entre los procesadores. Vamos a asumir que el sistema multiprocesador tiene suficiente memoria física o memoria virtual para almacenar el contenido del vector de indirección original. En esta situación, y dado que el ejecutor únicamente va a emplear el vector de indirección reordenado, este proceso no va a tener ningún impacto sobre la eficiencia del ejecutor paralelo. Es decir, el coste de memoria del ejecutor asociado al reemplazo del vector de indirección original por el reordenado es nulo. De este modo, nuestro objetivo es realizar una reordenación de las computaciones con el fin de aumentar la localidad en los accesos a la matriz a. Adicionalmente, también vamos a reordenar las estructuras de datos con el fin de aumentar la eficiencia del ejecutor y mejorar la localidad en los accesos al vector de indirección. En las siguientes secciones realizamos una descripción detallada de cada una de nuestras propuestas.

4.2.1 *Private Region Technique*

En esta sección se presenta una técnica para la paralelización de lazos irregulares con una única indirección. La Figura 4.9 ilustra la estructura de nuestra propuesta que denominamos *Private Region Technique* (`PRT`). Nuestra propuesta está basada en la aplicación de la regla del propietario sobre la matriz accedida por la indirección. El inspector recibe como argumento el vector de indirección, su representación IARD asociada e información acerca del esquema de distribución de los datos. Durante su ejecución, la rutina de inspección invoca al algoritmo de particionamiento (algoritmo `PT`), el cual determina el conjunto de entradas del vector de indirección asignadas a cada una de las particiones (particiones que están a su vez asignadas a cada uno de los procesadores). Posteriormente, el inspector realiza la distribución efectiva de las entradas del vector de indirección, generando un nuevo vector que denominamos x^{fin}. En una última etapa, el ejecutor emplea el vector x^{fin} junto con información auxiliar para realizar la ejecución paralela del lazo.

Una vez realizada la descripción general del proceso, vamos a introducir cada uno de estos módulos, comenzando por la rutina de particionamiento.

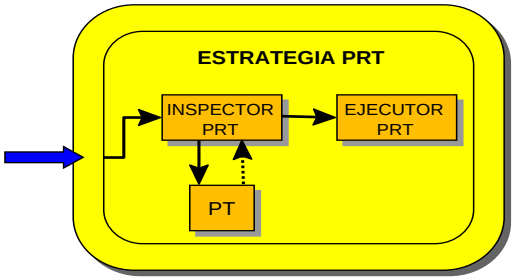


Figura 4.9: Estructura de la estrategia `PRT`.

Algoritmo particionador (`PT`)

El algoritmo de particionamiento recibe dos elementos de entrada: la caracterización IARD del lazo irregular considerado, y el conjunto de particionamiento, denotado como $\mathcal{LIM}$. Dado que nuestra propuesta se basa en la aplicación de la regla del propietario sobre la matriz a, la distribución de las entradas del vector de indirección dependerá únicamente del modo en que esté distribuida esta matriz sobre los procesadores. En este trabajo vamos a asumir que la matriz a tiene una única dimensión sobre la que se realiza una distribución por bloques de tamaño genérico. La información acerca del particionamiento de esta matriz (posición y tamaño de los bloques) está contenida en el conjunto $\mathcal{LIM}$, el cual se define del siguiente modo.

Definición 4.2.1 Sea N_{pt} el número total de particiones aplicadas sobre a. Sean lim_i^{inf} y lim_i^{sup} el límite inferior y superior que tiene asociada la i-ésima partición, denotamos **la partición** $\mathcal{A}_i$ al i-ésimo bloque en el que la matriz a ha sido particionada. Más formalmente, para cada partición $i = 1, N_{pt}$, tenemos que:

$$\mathcal{A}_i = [lim_i^{inf}, lim_i^{sup}) \subset [1, N_a] \tag{4.7}$$

□

Definición 4.2.2 El **conjunto de particionamiento** $\mathcal{LIM}$, contiene las particiones de a que definen cada bloque en el que la matriz considerada es particionada. Más formalmente,

$$\mathcal{LIM} = \{\mathcal{A}_i \qquad i = 1, N_{pt}\} \tag{4.8}$$

□

El resultado ofrecido por nuestro particionador consiste en el conjunto de *slices* asociados a cada uno de los bloques en los que la matriz es particionada. Un ejemplo se puede ver en

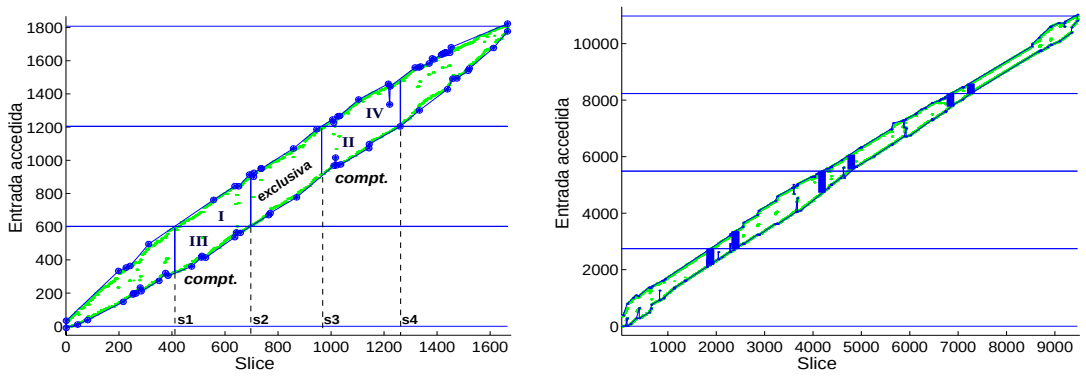


Figura 4.10: Esquemas de particionamiento: (a) particionamiento con $N_{pt} = 3$ para la matriz *bcsstk14*, (b) particionamiento con $N_{pt} = 4$ para la matriz *bcsstk17*.

la Figura 4.10(a), donde se considera un particionamiento en tres bloques de igual tamaño en la que: $\mathcal{LIM} = \{\mathcal{A}_1 = [1, 603), \mathcal{A}_2 = [603, 1205), \mathcal{A}_3 = [1205, 1806)\}$.

Cada bloque $\mathcal{A}_i$ tiene asociado cuatro *slices*, que denotaremos como s_i^1, s_i^2, s_i^3 y s_i^4. Estos *slices* delimitan los puntos en los que las rectas $y = lim_i^{inf}$ e $y = lim_i^{sup}$ cortan a la envolvente superior e inferior del patrón de acceso. La Figura 4.10(a) representa como s^1, s^2, s^3 y s^4 los *slices* asociados a la partición $[603, 1205)$. Otro ejemplo se muestra en la Figura 4.10(b), en donde se representa el patrón de acceso de la matriz *bcsstk17* particionado en cuatro bloques con sus correspondientes *slices* asociados. En este caso, los *slices* se destacan dentro del patrón de acceso como franjas gruesas.

El pseudocódigo de nuestra propuesta de particionador, denominada algoritmo `PT`, se ilustra en la Figura 4.11. El funcionamiento del mismo es el siguiente: para cada partición, y mediante la función *interseca*, se evalúan los puntos de corte entre las rectas asociadas a los límites inferior y superior de la partición con las curvas $\mathcal{E}^u$ y $\mathcal{E}^l$. Dada la representación mediante segmentos de ambas envolventes, únicamente hay que considerar aquellos que cortan a las rectas asociadas a los límites, y obtener, para cada uno de ellos, el valor de la abscisa del punto de corte. En un caso genérico las rectas cortarán a las envolventes en varios puntos. La Tabla 4.1 establece el criterio de selección para obtener el *slice* asociado, donde por ejemplo $lim_i^{sup} \cap \mathcal{E}^l$ representa el conjunto de puntos asociados a la intersección de la recta superior con la envolvente inferior. Las funciones max_x y min_x seleccionan, sobre todo el conjunto de puntos, aquel con un mayor y menor valor de abscisa, respectivamente.

```
Algoritmo PT
entrada
        IARD: caracterización IARD del lazo irregular
        LIM: conjunto de particionamiento
salida
        {s^1, s^2, s^3, s^4}_i   i = 1, N_pt: conjunto de slices asociados a cada partición
inicio del algoritmo
        DO  i = 1, N_pt
            {s^1, s^2, s^3, s^4}_i = interseca(IARD, lim_i^inf, lim_i^sup)
        END DO
fin del algoritmo
```

Figura 4.11: Algoritmo de particionamiento (PT).

Dado que el resultado devuelto por estas funciones son números reales, es necesario convertirlos en enteros mediante un redondeo que puede ser o bien por exceso o bien por defecto. El empleo del tipo de redondeo depende de que el primer corte de la recta que marca el límite de la partición sea con la envolvente superior o con la inferior. La Tabla 4.2 muestra las reglas de redondeo para los distintos tipos de cortes entre el límite superior o inferior con cada una de las envolventes. Por ejemplo, en los dos casos de la Figura 4.10, el primer corte se produce con la envolvente superior.

A continuación introducimos un nuevo concepto que va a permitir caracterizar el conjunto de entradas del vector de indirección pertenecientes a cada sección del patrón de acceso asociado a una partición.

slice	s_i^1	s_i^2	s_i^3	s_i^4
Intersección	$min_x(lim_i^{sup} \cap \mathcal{E}^u)$	$max_x(lim_i^{sup} \cap \mathcal{E}^l)$	$min_x(lim_i^{inf} \cap \mathcal{E}^u)$	$max_x(lim_i^{inf} \cap \mathcal{E}^l)$

Tabla 4.1: Criterio de selección para la obtención de los *slices* asociados a una partición.

Primer corte	$lim_i^{sup} \cap \mathcal{E}^u$	$lim_i^{sup} \cap \mathcal{E}^l$	$lim_i^{inf} \cap \mathcal{E}^u$	$lim_i^{inf} \cap \mathcal{E}^l$
$\mathcal{E}^u$	$\lfloor \cdot \rfloor$	$\lceil \cdot \rceil$	$\lfloor \cdot \rfloor$	$\lceil \cdot \rceil$
$\mathcal{E}^l$	$\lceil \cdot \rceil$	$\lfloor \cdot \rfloor$	$\lceil \cdot \rceil$	$\lfloor \cdot \rfloor$

Tabla 4.2: Tipos de redondeo para la función *interseca*.

Definición 4.2.3 Sea IARD la representación asociada a un vector de indirección x, y sean s^a y s^b dos *slices* cualesquiera de dicha representación con $s^a < s^b$, entonces definimos $\mathcal{X}_{s^a}^{s^b}$ como el conjunto de entradas del vector de indirección x comprendidas entre los *slices* s^a y s^b. Más formalmente,

$$\mathcal{X}_{s^a}^{s^b} = \{x[i] \ / \ i \in [\rho_{ini}(s^a),\ \rho_{ini}(s^b + 1))\} \tag{4.9}$$

□

De acuerdo con esta definición, podemos enunciar la siguiente propiedad.

Propiedad 4.2.1 Dada una representación IARD y un conjunto de particionamiento $\mathcal{LIM}$. Si $\{s^1, s^2, s^3, s^4\}$ es el conjunto de *slices* asociado a una partición $\mathcal{A}$ genérica derivada de $\mathcal{LIM}$, entonces se verifica que:

1. En el conjunto $\mathcal{X}_{s^2+1}^{s^3-1}$, denominado **región exclusiva**, toda entrada de la indirección realiza un acceso dentro de la partición considerada. Es decir, $\forall x[i] \in \mathcal{X}_{s^2+1}^{s^3-1} \longrightarrow x[i] \in \mathcal{A}$.
2. En los conjuntos $\mathcal{X}_{s^1}^{s^2}$ y $\mathcal{X}_{s^3}^{s^4}$, denominados **regiones compartidas**, no es posible garantizar que todas las entradas del vector de indirección accedan a a en la partición $\mathcal{A}$.
3. Cualquier acceso que se realiza dentro de $\mathcal{A}$, o bien pertenece a la región exclusiva, o bien a una de las regiones compartidas. Más formalmente, $\forall x[i] \in \mathcal{A}_i \longrightarrow x[i] \in \mathcal{X}_{s^1}^{s^4}$.

DEMOSTRACIÓN: La prueba de estas propiedades es directa teniendo en cuenta que las envolventes acotan superior e inferiormente al patrón de indirección. □

De este modo, tal y como se muestra en el ejemplo de la Figura 4.10(a), la región exclusiva abarca todas las entradas de x pertenecientes al intervalo de *slices* comprendido entre $(702, 970)$, mientras que las regiones compartidas comprenden los intervalos de *slices*$[418, 702]$ y $[970, 1283]$. Nótese que en la segunda partición todos los accesos de la región exclusiva se realizan dentro del bloque considerado (intervalo $[603, 1205)$ de entradas de a).

Las siguientes secciones describen el proceso de reordenamiento y de generación del código paralelo. En este proceso únicamente se emplean los conjuntos $\{s^1, s^2, s^3, s^4\}_i$ asociados a cada partición, y la representación IARD de la indirección.

Ejecutor `PRT`

Antes de describir nuestra propuesta de reordenamiento, comenzaremos introduciendo el modelo de código paralelo utilizado. Partiendo de un código secuencial como el mostrado en la Figura 4.7(b), la versión paralelizada mediante nuestra estrategia es de la forma

```
DOALL  p = 1, N_pt
    % Primera región compartida
     DO  l=s_p^1,s_p^2
        DO i=ρ_ini(s),corte_p(s − s_p^1)
                a[x[i]]= a[x[i]] ⊙ ...
                { ... }
        END DO
     END DO

    % Región exclusiva
     DO  i=ρ_ini(s_p^2 + 1),ρ_ini(s_p^3) − 1
                a[x[i]]= a[x[i]] ⊙ ...
                { ... }
     END DO

    % Segunda región compartida
     DO  l=s_p^3,s_p^4
        DO  i=corte_{p+1}(s − s_p^3) + 1,ρ_ini(s + 1) − 1
                a[x[i]]= a[x[i]] ⊙ ...
                { ... }
        END DO
     END DO
END DO
```

Figura 4.12: Código ejecutor de la estrategia PRT.

mostrada en la Figura 4.12. De este modo, el código original aparece ahora reemplazado por tres lazos que representan cada una de las regiones asociadas a cada partición. Como lazo más externo aparece un lazo totalmente paralelo que recorre cada una de las particiones de a. Para cada una de estas particiones el lazo correspondiente a la región exclusiva recorre todas las entradas de dicha región dado que, de acuerdo con la Propiedad 4.2.1, todas ellas acceden al intervalo de la partición considerada. Este no es el caso para las regiones compartidas, en las que es necesario utilizar un vector auxiliar, denominado *corte*, para discernir aquellas entradas de la indirección que operan sobre la partición considerada de aquellas que no lo hacen. En la Figura 4.10(a) se representan como I y II, las zonas de las regiones compartidas en las que los accesos se realizan sobre la partición considerada (que en este caso es la partición central).

El vector *corte* es generado por el algoritmo inspector PRT. La siguiente definición

formaliza las propiedades exigidas a dicho vector.

Definición 4.2.4 Dado una partición $\mathcal{A}_i$ del vector a y una representación IARD del vector de indirección, definimos el vector $corte_i$ como aquel que almacena, para cada *slice* perteneciente a la región comprendida entre los *slices* s^1 y s^2, la última entrada del vector de indirección que accede en la partición. De este modo, $\forall l \in [s^1, s^2]$ se debe verificar que:

$$\forall j \in [\rho_{ini}(s), corte_i[s - s^1]] \quad \Longrightarrow \quad x[j] \in \mathcal{A}_i \tag{4.10}$$

$$\forall j \in (corte_i[s - s^1], \rho_{ini}(s + 1)) \quad \Longrightarrow \quad x[j] \notin \mathcal{A}_i \tag{4.11}$$

□

De modo implícito esta definición asume que en cada partición las entradas pertenecientes a un *slice* pueden dividirse en dos conjuntos: aquellos que acceden sobre la partición y aquellos que no lo hacen. Además, también asumimos que las entradas correspondientes al primer conjunto ocupan posiciones contiguas en memoria.

A modo de ejemplo vamos a suponer que los accesos realizados dentro de cada *slice* se realizan en sentido decreciente. Es decir, para un *slice* S_i genérico, si $j \in S_i$ y $k \in S_i$ con $j < k$, entonces $x[j] > x[k]$. Para esta situación particular, la Figura 4.13 muestra un ejemplo de las entradas almacenadas en el vector *corte*. En este ejemplo, estamos considerando una región compartida con $s^1 = 391$ y $s^2 = 397$. Las entradas de la indirección pertenecientes a la zona I (perteneciente a $\mathcal{A}_i$) y zona III (no perteneciente a $\mathcal{A}_i$) son etiquetadas, respectivamente, con los símbolos " +" y " ∘". Para este ejemplo, el vector *corte* tiene 7 elementos, denotados en la figura con el símbolo "∗". Se puede apreciar que, para este tipo de distribución, el vector *corte* establece la separación entre los accesos pertenecientes y no pertenecientes a la partición considerada.

Inspector `PRT`

En un caso general, las entradas contenidas en cada *slice* no guardan un orden establecido. Para abordar esta nueva situación es necesario aplicar un algoritmo de permutación que agrupa las entradas de cada *slice* en dos conjuntos: aquellos que realizan accesos dentro del $\mathcal{A}_i$, y aquellos que no lo hacen. La Figura 4.14 muestra el pseudocódigo del algoritmo que realiza este reordenamiento de la indirección y genera el vector de *corte*. Este algoritmo, denominado inspector *Private Region Technique* (inspector `PRT`), recibe como datos de entrada la indirección irregular y la caracterización IARD de la región del código candidata a ser paralelizada. Adicionalmente utiliza el conjunto de particionamiento para determinar las regiones compartidas y exclusivas del patrón de acceso. El primer

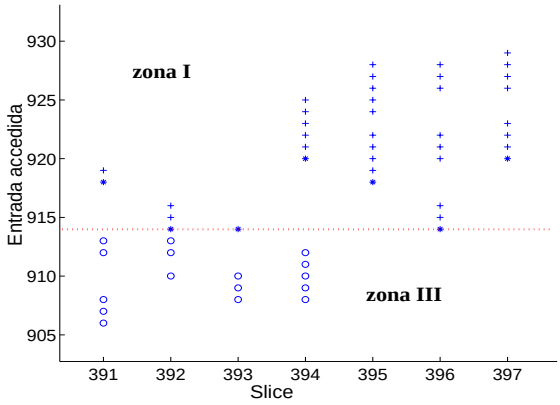


Figura 4.13: Organización de las entradas para una región compartida.

paso consiste en la determinación de los *slices* asociados a cada una de las particiones de a. Esta tarea es realizada por la rutina `PT` mostrada en la Figura 4.11. A continuación, el algoritmo realiza el reordenamiento de la indirección y genera el vector *corte*. Para ello analiza cada una de las particiones de a, y procesa secuencialmente los *slices* pertenecientes a la primera región compartida. Dicha región está comprendida entre los *slices* s_1 y s_2 asociados al límite inferior de cada partición.

Vamos a detallar el proceso de reordenamiento de las entradas de cada *slice* utilizando el ejemplo de la Figura 4.15, en el que se pretende reordenar seis entradas pertenecientes a un mismo *slice*. Inicialmente (columna (a) de la figura y etiquetas `L1` y `L2` del pseudocódigo) los punteros *ini* y fin marcan la primera y última entrada del *slice* considerado. El puntero fin es decrementado hasta apuntar a una entrada que accede dentro de la zona I (columna (b) de la figura y etiqueta `L3` del pseudocódigo). Una vez alcanzada esta posición, el puntero *ini* es incrementado hasta apuntar a un acceso dentro de la zona III (columna (c) de la figura y etiqueta `L4` del pseudocódigo). El siguiente paso a realizar consiste en la permutación de ambos accesos. Esta tarea es realizada mediante la función *intercambia* (columna (d) de la figura y etiqueta `L5` del pseudocódigo).

Una vez realizada esta permutación, el proceso se vuelve a repetir hasta que el valor del puntero *ini* sea mayor o igual que el de fin. Cuando ambos punteros marcan a la misma entrada de x, el proceso se detiene, devolviendo su posición como elemento del vector *corte* (columna (e) de la figura y etiqueta `L6` del pseudocódigo). Nótese que en un caso general las entradas de cada *slice* no están ordenadas, sino que simplemente son agrupadas en función de la partición a la que acceden.

A continuación, vamos a estimar la complejidad de nuestra propuesta. Se puede afirmar

```
Algoritmo inspector PRT
entrada
        IARD: caracterizaciones IARD del lazo irregular
        x: vector de indirección
        LIM: conjunto de particionamiento
salida
       x^fin: vector de indirección reordenado
       corte_i   i = 1, N_pt: conjunto de vectores de corte
       {s^1, s^2, s^3, s^4}_i    i = 1, N_pt: conjunto de slices asociados a cada partición
inicio del algoritmo
        {s^1, s^2, s^3, s^4}_i = PT(IARD,LIM)

        DO  i = 2, N_pt
           DO l=s^1_i, s^2_i
L1            ini = ρ_ini(s)
L2            fin = ρ_ini(s + 1) − 1
L3            WHILE  (x[fin] ∉ A_i & fin ≥ ini) fin − −
              WHILE  (fin > ini)
L4                  WHILE (x[ini] ∈ A_i & fin > ini) ini + +
                    IF (fin > ini)
L5                     intercambia(x[ini], x[fin])
                       fin − −
                       WHILE  (x[fin] ∉ A_i & fin > ini) fin − −
                    END IF
              END WHILE
L6            corte_p[s − s^p1] = fin
           END DO
        END DO
fin del algoritmo
```

Figura 4.14: Pseudocódigo del inspector PRT.

que la complejidad del algoritmo PT es reducida, dado que únicamente utiliza la información de la caracterización IARD, y las operaciones realizadas son simples. La complejidad de este algoritmo es la siguiente:

$$\mathcal{O}_{\mathtt{PT}} = (N_{pt} - 1)2N_{\mathcal{E}} \tag{4.12}$$

El primer término representa el número de cortes que hay que evaluar, mientras que el

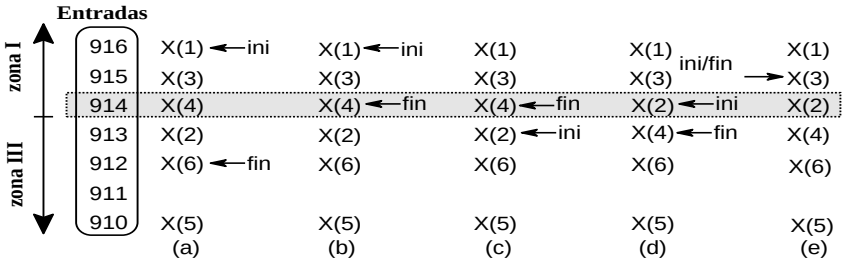


Figura 4.15: Reordenamiento de las entradas de un *slice* por medio del algoritmo `PRT`.

segundo hace referencia al número de segmentos de las envolventes superior e inferior que es necesario considerar. Por simplicidad, en los ejemplos utilizados asumimos que ambas envolventes tienen el mismo número de segmentos.

La complejidad total del algoritmo `PRT` viene dada por la suma de las contribuciones del algoritmo `PT` y las del proceso de reordenación.

$$\mathcal{O}_{\texttt{PRT}} = \mathcal{O}_{\texttt{PT}} + N_{pt} N_x^{shared} = (N_{pt} - 1) 2 N_{\mathcal{E}} + N_{pt} N_x^{shared} \tag{4.13}$$

Donde N_x^{shared} es el número medio de entradas del vector de indirección pertenecientes a la región compartida determinada por el límite inferior de la partición. Más formalmente,

$$N_x^{shared} = \frac{\sum_{i=2}^{N_{pt}} \mathcal{X}_{s1_i}^{s2_i}}{N_{pt} - 1} \tag{4.14}$$

El coste de almacenamiento del algoritmo `PRT` es otro factor que debemos considerar. Vamos a distinguir dos clases de coste de almacenamiento: el asociado al inspector y el asociado al ejecutor. El **coste de memoria asociado al inspector** representa la memoria requerida por el proceso de paralelización automática. En este factor incluimos toda la información necesaria para realizar una correcta ejecución del lazo paralelo y para retomar con posterioridad la ejecución del resto del programa. Por otra parte, el **coste de memoria del ejecutor** se define como la diferencia entre la memoria requerida por el código original y el ejecutor paralelo. El coste de almacenamiento del inspector viene dado por la siguiente expresión:

$$\Delta M_{\texttt{PRT}}^{inspector} = N_x + \sum_{i=2}^{N_{pt}} (s_i^4 - s_i^1) + 4 N_{pt} \tag{4.15}$$

Donde el primer término tiene en cuenta el coste de almacenamiento del vector reordenado, el segundo representa el coste asociado a los vectores *corte* y el último término representa los índices de *slices* asociados a cada partición. A esta contribución es necesario añadir el

coste de almacenamiento de la caracterización IARD. Cabe destacar una situación particular que aparece cuando las entradas del vector de indirección están ordenadas de forma monótona en cada *slice*. Es decir, los accesos de cada *slice* se realizan en sentido creciente (como en el caso de la Figura 4.13) o decreciente. En este caso no es necesario reordenar las entradas del vector de indirección, lo que permite una significativa reducción en los costes de almacenamiento.

El coste de almacenamiento asociado al ejecutor es mucho menor, ya que este únicamente emplea el vector de indirección reordenado, sin tener que acceder al original. Específicamente, este coste se debe a los vector *corte* y a los *slices* que delimitan cada partición.

$$\Delta M_{\texttt{PRT}}^{ejecutor} = \sum_{i=2}^{N_{pt}} (s_i^4 - s_i^1) + 4N_{pt} \tag{4.16}$$

Una propiedad interesante del algoritmo PRT es que las regiones exclusivas no son procesadas, por lo que, para patrones de acceso bandeados y un número no muy alto de particiones, la complejidad total se ve reducida de forma importante. Otra ventaja de nuestra propuesta es el desacoplo entre la etapa de caracterización IARD y la etapa de inspección. Este último aspecto será estudiado de forma más detallada en la siguiente sección, donde realizamos un estudio empírico de su eficiencia.

Análisis de eficiencia

El algoritmo PRT fue evaluado en un sistema multiprocesador de memoria compartida Sun HPC4500. Este sistema tiene una arquitectura UMA y consta de 8 procesadores UltraSparc II a 400MHz. El rendimiento de nuestra propuesta fue comparado con las técnicas de paralelización *array expansion* y DWA-LIP. La elección de estas técnicas de paralelización se debe a varios motivos: del mismo modo que nuestra propuesta, ambas están orientadas a su empleo sobre arquitecturas de memoria compartida. La técnica de *array expansion* está ampliamente difundida, es empleada en la actualidad por herramientas de paralelización automática como Polaris, y supone un punto de referencia para realizar estudios comparativos de rendimiento. La elección de la técnica DWA-LIP se debe a la afinidad que guarda con nuestra propuesta, dado que también se basa en la aplicación de regla del propietario y explota la localidad en los accesos. Un último motivo de la elección de estas técnicas radica en que son complementarias, en el sentido que presentan un mejor rendimiento para patrones de acceso con distintas características. Por este motivo, en esta sección evaluamos el impacto de la estructura del patrón de acceso en la eficiencia del código paralelo.

Los códigos paralelos fueron escritos en Fortran 77 y programados utilizando las directivas de memoria compartida de Sun. El compilador empleado es el WorkShop `f77` v4.2. Como fuente de vectores de indirección se emplearon matrices dispersas provenientes de la *Harwell Boeing sparse matrix collection* [37] y de la *University of Florida Sparse Matrix Collection* [34]. La Tabla 2.4, introducida en el Capítulo 2, muestra las principales características de estas indirecciones así como de la representación IARD asociada. En ese mismo capítulo, la Figura 2.34 muestra, para estas indirecciones, la representación del patrón de acceso y de la caracterización IARD.

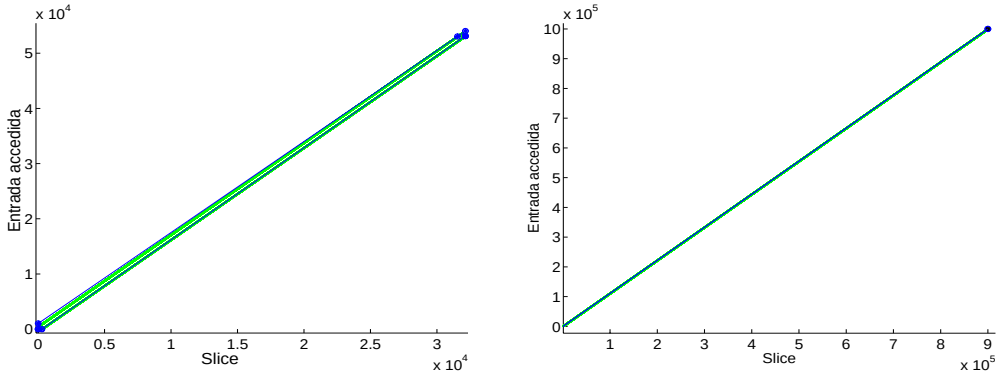
Con el fin de ampliar el número de escenarios considerados, hemos evaluado dos nuevos vectores de indirección. El primero de ellos, denominado *struct*3, fue extraído de la *Harwell Boeing sparse matrix collection*, mientras que el segundo, denominado *diag_block*, fue generado sintéticamente y se corresponde a una matriz diagonal compuesta por bloques densos, con un tamaño de bloque de 10×10 entradas. Con estos nuevos vectores pretendemos evaluar la eficiencia de nuestra propuesta para estructuras muy bandeadas. La Figura 4.16 ilustra el patrón de acceso y la representación IARD para estos dos nuevos casos, mientras que la Tabla 4.3 muestra sus principales características. Nótese que para estos casos se consigue (del mismo modo que en las otras indirecciones) una importante reducción en el coste de almacenamiento de la representación IARD (ΔM_{IARD}) respecto al coste de almacenamiento asociado al vector de indirección. (N_x).

El esquema de distribución que hemos utilizado consiste en una distribución por bloques del vector a. En esta sección no vamos a abordar el análisis del balanceo de carga computacional, por lo que todos los bloques empleados tienen el mismo tamaño.

La Tabla 4.4 muestra los tiempos de ejecución secuencial de proceso de inspección asociado a las estrategias `PRT` y DWA-LIP. La técnica de *array expansion* no necesita inspector. En el caso de nuestra propuesta, los valores de tiempo aparecen desglosados entre el coste del proceso de elaboración de la caracterización IARD y el del propio inspector `PRT`. El primero de ellos es dividido a su vez en el coste del algoritmo de *Clasificación por Slices* (`CS`) y el del algoritmo de obtención de la *Envolvente Heurística* (`EH`). Por otra banda, el coste del algoritmo (`PRT`) también aparece dividido entre el coste del *particionador* (`PT`) y

Matriz	N_a	N_x	N_S	ΔM_{slices}	$N_{\mathcal{D}}$	ΔM_{IARD}
struct3	53570	**1173694**	32142	**96425**	4	**32158**
diag–block	1000000	**10000000**	900000	**2700000**	2	**900008**

Tabla 4.3: Características de las matrices *struct*3 y *diag_block*.



struct3 *diag_block*

Figura 4.16: Patrón de acceso y representación IARD de diversas indirecciones.

el del resto del algoritmo (PRT-PT).

En esta tabla se puede apreciar que el coste del inspector PRT es mucho menor que el coste del inspector DWA-LIP, el cual no utiliza un caracterizador sino que realiza un análisis completo sobre el patrón de indirección. Se puede apreciar que el tiempo de caracterización del vector de indirección es la parte más costosa de nuestra propuesta. Dentro de esta etapa, y con el fin de reducir este coste, realizamos la paralelización del código original del algoritmo EH empleando la librería de pase de mensajes MPI. De este modo, se consigue cierta reducción en el coste de esta etapa mediante el procesamiento paralelo

Matriz	DWA–LIP⋆ (*ms*)	IARD (*ms*)		Algoritmo PRT (*ms*)		TOTAL (*ms*)
		CS	EH	PT	PRT-PT	
s3dkq4m2	1116	575	644	0.039	8.9	**1228**
struct3	269	182	193	0.047	2.2	**378**
nasasrb	620	317	728	0.042	6.0	**1051**
3dtube	748	367	995	0.166	58.5	**1447**
diag–block	2340	2855	1693	0.049	0.1	**4548**
bcsstk17	94	48	167	0.048	3.5	**218**
bcsstk14	14	7	30	0.068	1.0	**38**

⋆Versión secuencial

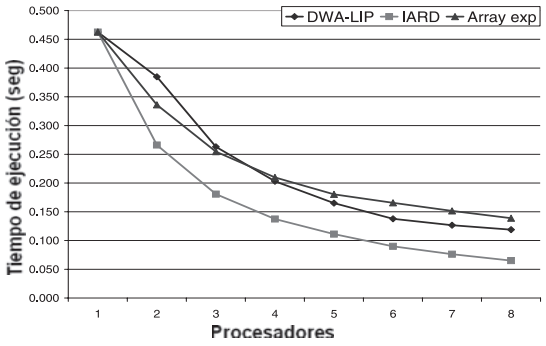
Tabla 4.4: Coste de la representación IARD y del inspector de la estrategia PRT en términos del tiempo de ejecución para una partición de 4 bloques.

de la envolvente superior e inferior. Los valores mostrados en la tabla se corresponden a las obtenidas con dicha versión paralela. La última columna muestra el tiempo total de ejecución de nuestra propuesta.

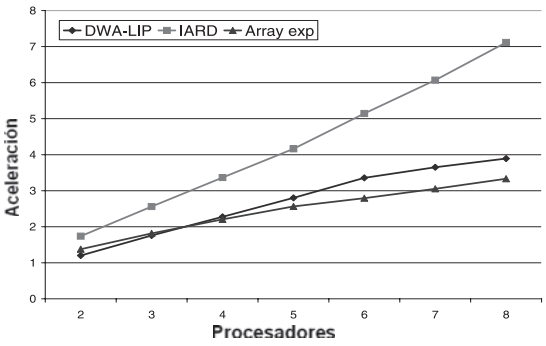
En un entorno de caracterización y paralelización automática, el análisis y procesado de la representación supone un factor crítico en el proceso de inspección. En un importante número de aplicaciones, el vector de indirección o bien es inicializado al comienzo del programa, o bien se crea o modifica en puntos alejados temporalmente del momento en que el código irregular es ejecutado. Explotando esta separación temporal entre creación y uso de la indirección, y haciendo que el proceso de caracterización se realice en paralelo con la ejecución del programa, su coste puede llegar a ocultarse. De este modo, para estas situaciones sería posible disponer de la representación IARD antes de alcanzar la región candidata a ser paralelizada. Esto no ocurre para los inspectores DWA-LIP y `PRT` en el caso de que el número de procesadores disponibles, o su carga computacional, variase a lo largo de la ejecución del programa. Es estos casos sería necesario ejecutar las rutinas de inspección de estas técnicas antes de la ejecución paralela del lazo.

Las figuras 4.17, 4.18 y 4.19 muestran, para distinto número de procesadores, los tiempos de ejecución y las aceleraciones obtenidas para el ejecutor utilizando la estrategia `PRT`. Además se representan los resultados obtenidos para un código paralelizado mediante la técnica *array expansion* y DWA-LIP. Notar que, excepto para la matriz *bcsstk14*, los mejores resultados son los conseguidos mediante el empleo de nuestra propuesta.

Las ventajas del algoritmo *PRT* respecto a otras propuestas son diversas: por un lado, realizando una comparativa con el algoritmo DWA-LIP, ambas propuestas aplican la

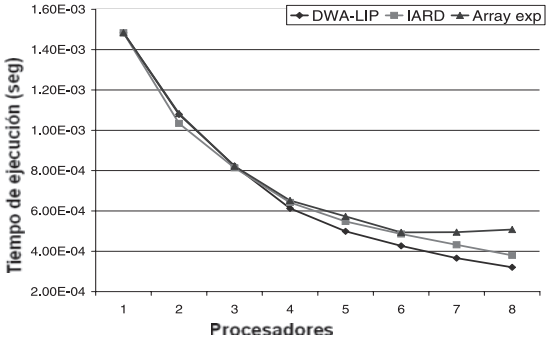


(a) Tiempo de ejecución para *diag-block*

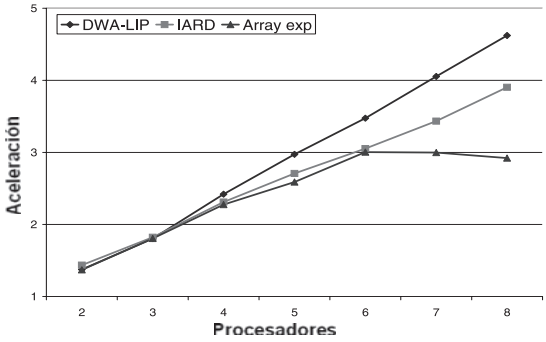


(b) Aceleración para *diag-block*

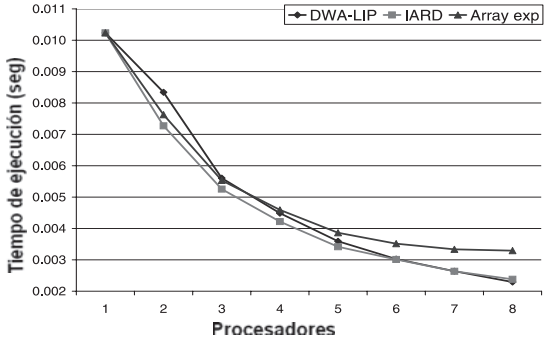
Figura 4.17: Rendimiento obtenido para la matriz *diag-block*.



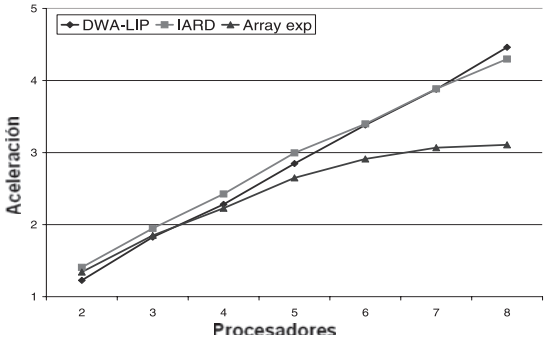
(a) Tiempo de ejecución para *bcsstk14*



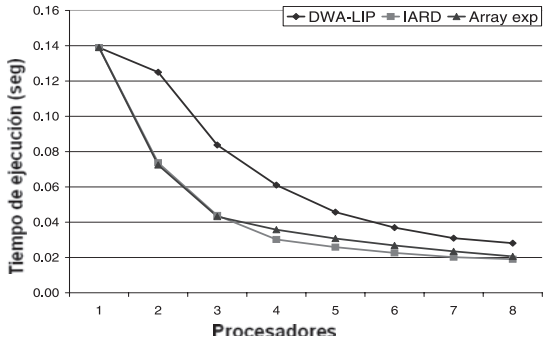
(b) Aceleración para *bcsstk14*



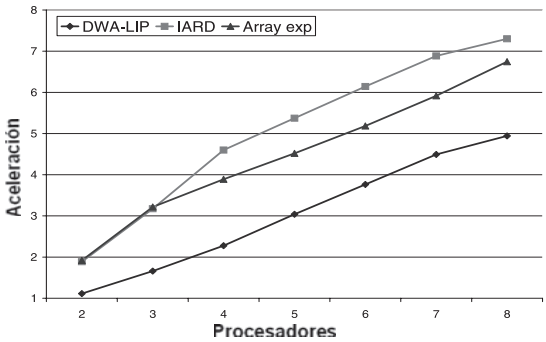
(c) Tiempo de ejecución para *bcsstk17*



(d) Aceleración para *bcsstk17*

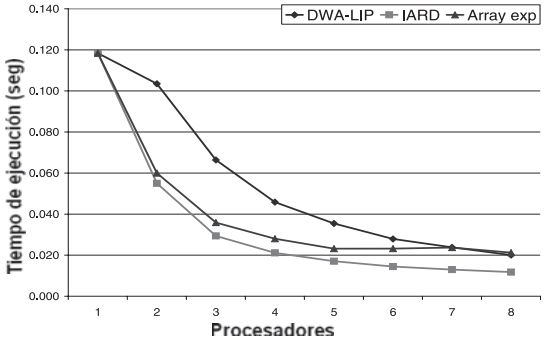


(e) Tiempo de ejecución para *3dtube*

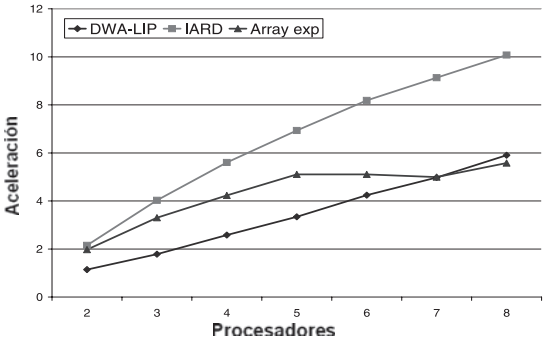


(f) Aceleración para *3dtube*

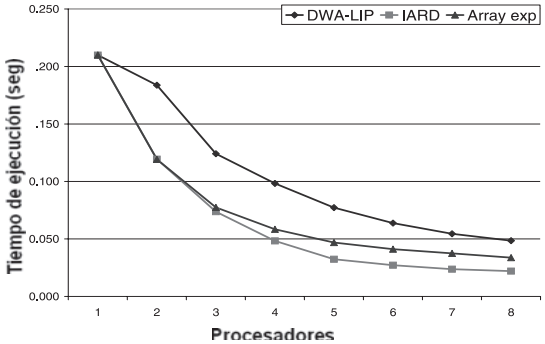
Figura 4.18: Rendimiento obtenido para las matrices *bcsstk14*, *bcsstk17* y *3dtube*.



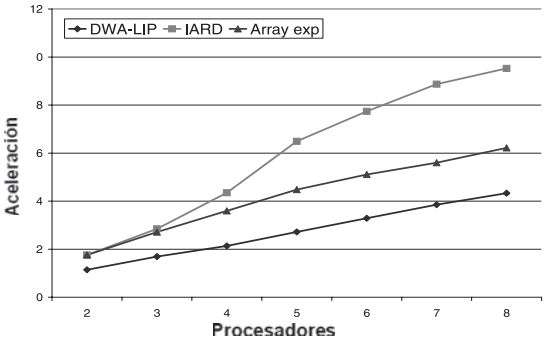
(a) Tiempo de ejecución para *nasasrb*



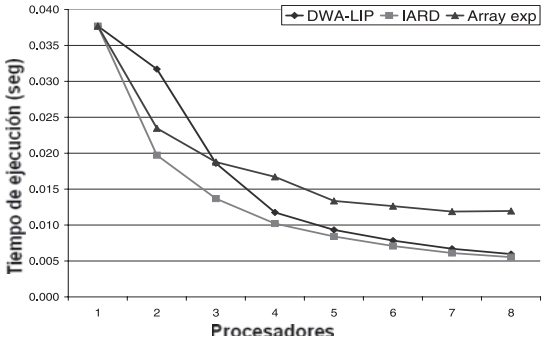
(b) Aceleración para *nasasrb*



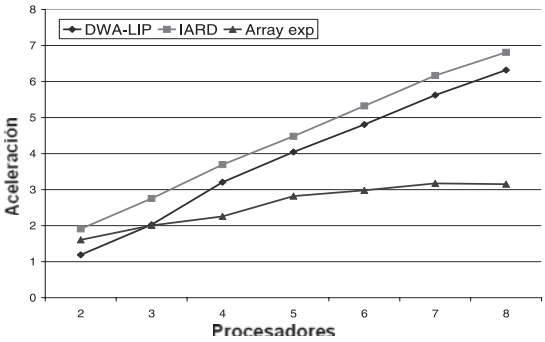
(c) Tiempo de ejecución para *s3dkq4m2*



(d) Aceleración para *s3dkq4m2*



(e) Tiempo de ejecución para *struct3*



(f) Aceleración para *struct3*

Figura 4.19: Rendimiento obtenido para las matrices *nasasrb*, *s3dkq4m2* y *struct3*.

regla del propietario sobre el vector accedido por la indirección, consiguiendo una mejora en la localidad de los accesos. Sin embargo, nuestra propuesta logra reducir los costes de almacenamiento del ejecutor (ver Expresión 4.16) respecto a la técnica DWA-LIP (ver Expresión 4.6). Experimentalmente, para el conjunto de datos considerado se tiene que $\Delta M^{ejecutor}_{DWA-LIP} \gg \Delta M^{ejecutor}_{\texttt{PRT}}$. La importante reducción de estos costes de almacenamiento implica una disminución en los accesos a la memoria y una mejor explotación de la jerarquía de memoria del sistema.

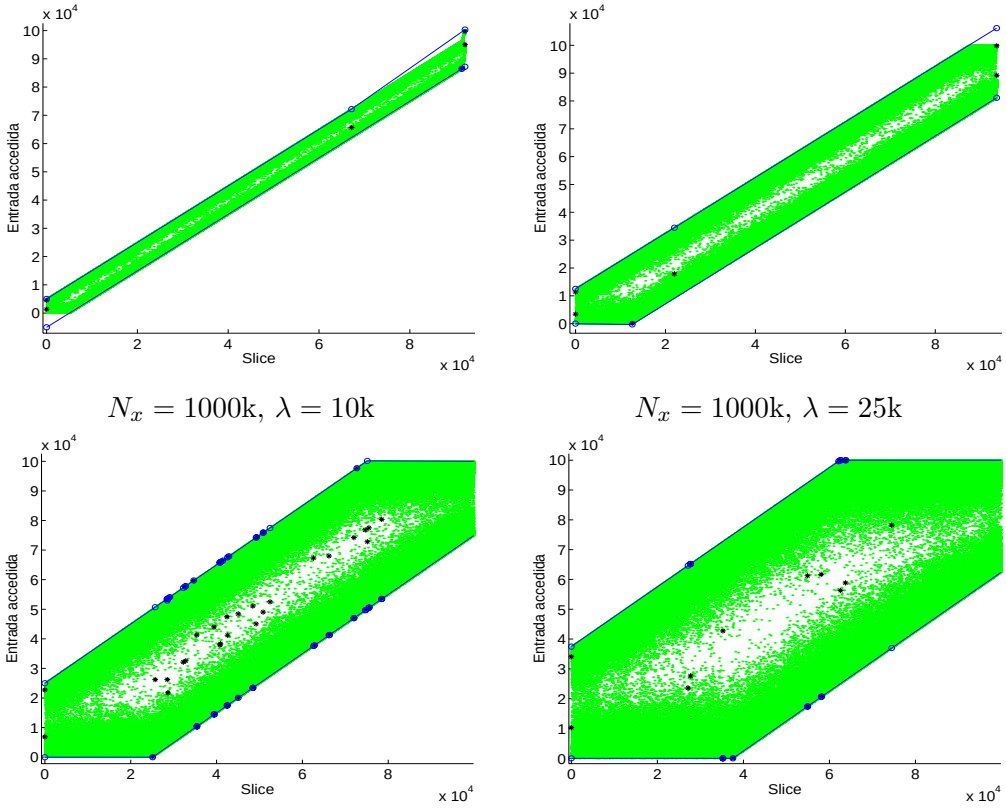
A modo de ejemplo, asumiendo una matriz no bandeada (o con una banda que representa todo el patrón de acceso) tendremos que $s^1_i = 1$ y $s^4_i = N_S \ \ \forall i$. Esta es la situación más desfavorable para nuestra propuesta, y aún así, para las matrices que hemos utilizado, hace falta un número de particiones entre 37 y 74 para superar el coste de almacenamiento de la técnica DWA-LIP.

En la Figura 4.18 se puede apreciar cómo la técnica DWA-LIP supera a nuestra propuesta para la matriz *bcsstk14*. Este hecho se puede explicar en base a que la matriz *bcsstk14* tiene un reducido número de entradas, lo que hace que el efecto anteriormente descrito no ejerza una gran influencia sobre el rendimiento del programa paralelo.

Realizando la comparativa entre nuestra propuesta y la técnica *array expansion*, el algoritmo `PRT` presenta dos grandes ventajas. Por una parte, no es necesario replicar el vector a, y por otra, tampoco es necesario realizar una operación de comunicación final para recopilar las contribuciones parciales de cada procesador. Ambas mejoras implican una importante reducción tanto en los costes de almacenamiento como de comunicaciones.

Una desventaja de la técnica *PRT* es el coste de almacenamiento de los $N_{pt} - 1$ vectores *corte*, dado por la Expresión 4.16. El coste de almacenamiento de la técnica *array expansion* está asociado a la replicación del vector a. Específicamente viene dado por la Expresión 4.1. Por norma general, se verifica que $N_S < N_a$, de modo que el máximo coste de almacenamiento de nuestra propuesta sigue siendo inferior al de *array expansion*. Adicionalmente, para un patrón de acceso bandeado, se tiene que $s^4_i - s^1_i < N_S$, por lo que esta diferencia se verá todavía más acentuada.

Con el fin de cuantificar el impacto de cada uno de estos factores (anchura de la banda y número de entradas), hemos generado una familia de matrices sintéticas parametrizadas. Todas ellas tienen el mismo número de filas, mientras que el número de entradas no nulas y la anchura de la banda están parametrizados. Vamos a denotar por λ a la anchura de la banda, expresada en número de *slices*. Para todos los casos, el número de filas y columnas de la matriz original es de 100k y la distribución de las entradas dentro de la banda es aleatoria y homogénea. Por *homogénea* entendemos que la distribución de



$N_x = 1000\text{k},\ \lambda = 10\text{k}$ $N_x = 1000\text{k},\ \lambda = 25\text{k}$

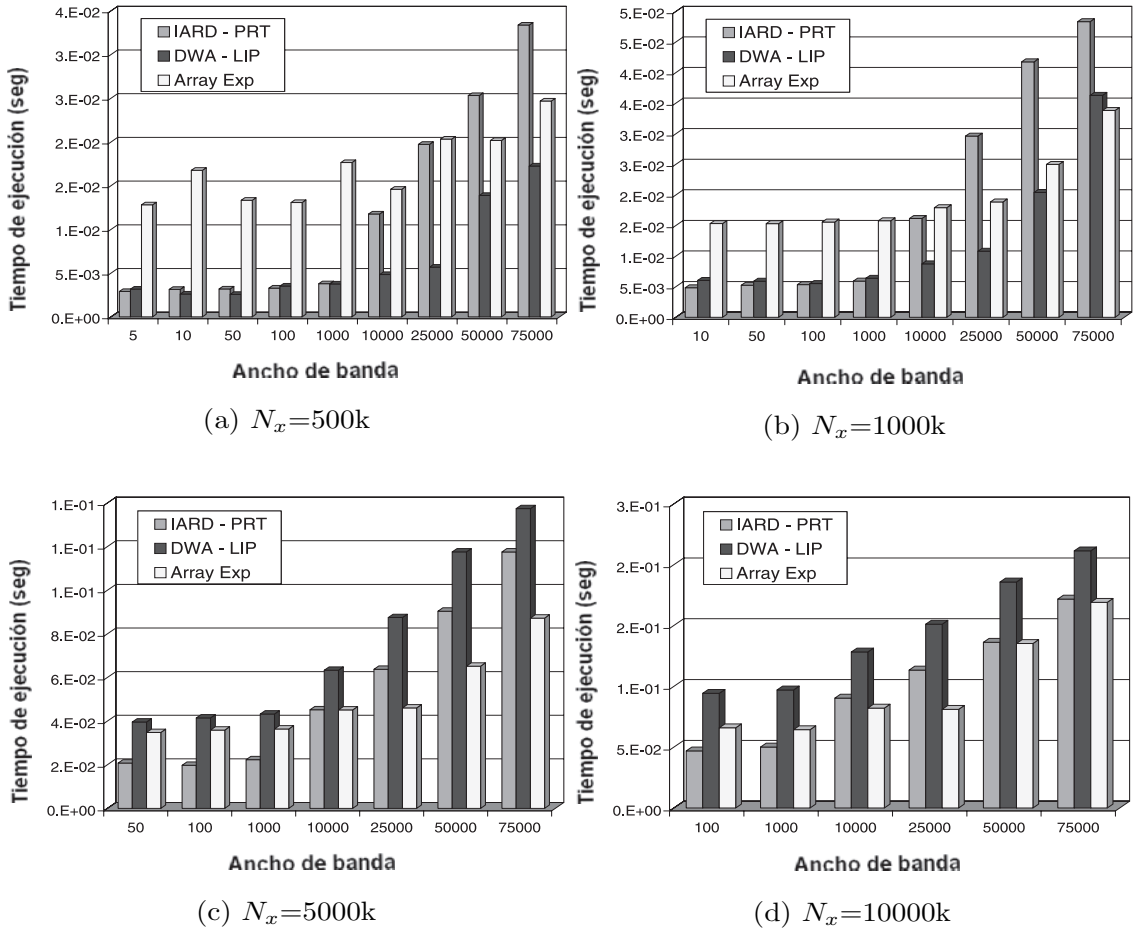
$N_x = 1000\text{k},\ \lambda = 50\text{k}$ $N_x = 1000\text{k},\ \lambda = 75\text{k}$

Figura 4.20: Representación IARD de matrices sinteticas.

las entradas es uniforme y tiene aproximadamente la misma densidad en cada *slice*. La Figura 4.20 muestra, para los ejemplos más representativos de estas matrices, las entradas máximas y mínimas de cada *slice* (vectores u y l), los puntos dominantes (conjunto $\mathcal{D}$) y las envolventes ($\mathcal{E}^u$ y $\mathcal{E}^l$). En la figura se puede apreciar la existencia de una estructura bandeada y la poca suavidad en el patrón de acceso que hace que el borde de la banda aparezca difuminado.

Bajo este conjunto de datos de entrada hemos evaluado el rendimiento de las tres estrategias. Los resultados obtenidos para una ejecución con 8 procesadores se muestran en la Figura 4.21. La Tabla 4.5 representa, para todo el rango considerado de N_x y λ, la estrategia más eficiente en términos de tiempo de ejecución del ejecutor paralelo.

Se puede apreciar cómo la estrategia `PRT` obtiene los mejores resultados en todo el rango de valores de N_x para bandas de tamaño medio y reducido. Esto es debido a que



(a) N_x=500k (b) N_x=1000k

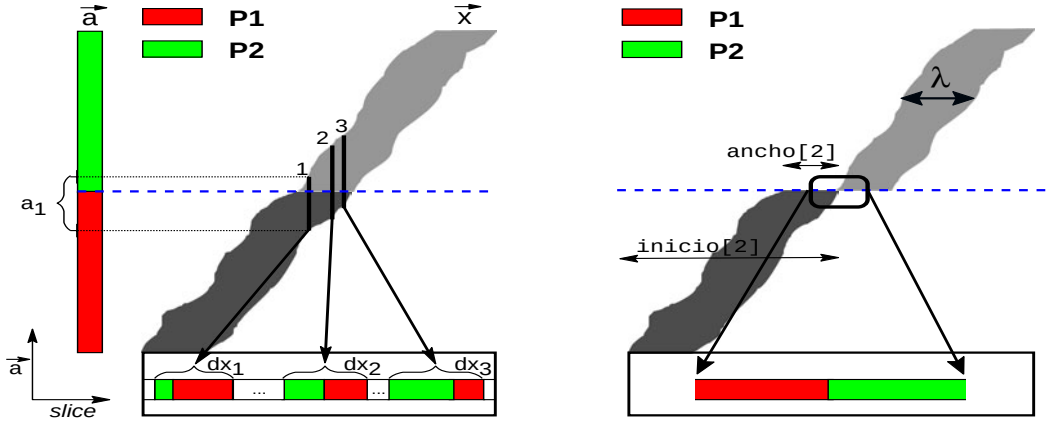
(c) N_x=5000k (d) N_x=10000k

Figura 4.21: Rendimiento para las matrices sintéticas para $N_p = 8$.

N_X	Ancho de banda (λ)								
	5	10	50	100	1000	10000	25000	50000	75000
10000k	–	–	–	♣	♣	■	■	■	■
5000k	–	–	♣	♣	♣	♣	■	■	■
1000k	–	♣	♣	♣	♣	♦	♦	♦	■
500k	♣♦	♣♦	♣♦	♣♦	♣♦	♦	♦	♦	♦

♣ PRT, ♦ DWA-LIP, ■ *array expansion.*

Tabla 4.5: Técnica más eficiente para distintos valores de N_x y λ para $N_p = 8$.



(a) (b)

Figura 4.22: Distribución de entradas de un patrón de acceso: (a) patrón de acceso original, (b) patrón de acceso reordenado.

para anchuras de banda no muy grandes el coste de almacenamiento de nuestra propuesta es pequeño. Por otra parte, la estrategia DWA-LIP obtiene los mejores resultados para matrices muy dispersas y con un número reducido de entradas. El rendimiento de esta propuesta decrece fuertemente cuando aumenta el número de entradas del vector de indirección. Finalmente, la técnica *array expansion* resulta la más eficiente cuando el número de entradas no nulas y el ancho de banda son elevados. Estos dos parámetros no afectan de modo importante al rendimiento de la técnica *array expansion*, mientras que sí lo hace (y de forma negativa) para las técnicas DWA-LIP y `PRT`.

4.2.2 *Sorted Private Region Technique*

Nuestra segunda propuesta surge ante la necesidad de obtener un código paralelo cuya eficiencia no dependa de parámetros asociados a las características del vector de indirección. Analizando el rendimiento del algoritmo `PRT`, hemos identificado dos fuentes principales de ineficiencia. La primera de ellas, se debe al coste de almacenamiento de los vectores $corte_i$. El empleo, por parte del ejecutor, de estos vectores introduce un coste de cálculo adicional y un aumento en el volumen de datos accedido por cada procesador.

La segunda fuente de ineficiencia es debida a la baja localidad en los accesos al vector de indirección. En una ejecución paralela del código de la Figura 4.7(a), y mediante el empleo de la técnica `PRT`, la matriz a se divide en N_{pt} bloques. Cada uno de estos bloques es asignado a un procesador, obteniendo una alta localidad en los accesos a la

matriz a durante la ejecución del código paralelo. Consideremos ahora la Figura 4.22(a), la cual muestra un ejemplo de dichos accesos para una partición de a en dos bloques de igual tamaño asignados a los procesadores $P1$ y $P2$. Adicionalmente, en el patrón de acceso aparecen destacadas las regiones que, de acuerdo con la regla del propietario, acceden sobre cada uno de estos bloques. En esta figura asumimos que el patrón de acceso presenta una estructura bandeada. A pesar de esta particularización que hemos realizado, los resultados y conclusiones que obtendremos pueden ser generalizados a patrones de acceso no bandeados.

Dado que las entradas del patrón están clasificadas por *slices*, el valor de abscisa de cada punto se corresponde al índice de *slice* que tiene asociado, mientras que su ordenada corresponde a la entrada de a accedida. A modo de ejemplo, los elementos del vector de indirección asociados al *slice* "1" (destacado en la figura) accederán al intervalo de entradas a_1 de a. En la parte inferior de la figura, se muestran los intervalos de entradas asociadas a tres *slices* de la representación. Por ejemplo, el intervalo dx_2 denota las entradas de x que pertenecen al *slice* "2". Si asumimos, para las entradas de cada *slice*, un orden de almacenamiento monótono decreciente, entonces la distribución de las mismas será tal y como se muestra en la figura. Se puede apreciar que la primera parte de las entradas están asociadas a la partición 2, mientras que el resto quedan asociadas a la partición 1.

De este modo, y dado que los *slices* están almacenados en memoria de modo consecutivo, la aplicación de la técnica `PRT` da lugar a una baja localidad en los accesos a las entradas del vector de indirección situadas en las regiones compartidas.

Con el fin de reducir o eliminar ambas causas de ineficiencia, proponemos un reordenamiento más agresivo del vector de indirección. Específicamente, nuestra propuesta realiza el reordenamiento de las entradas de x asociadas a cada partición, de modo que aquellas que realizan accesos sobre el mismo bloque de a están dispuestas en memoria de forma contigua. De este modo, es posible obtener una alta localidad tanto en escrituras de a como en lecturas de x. El efecto de la reordenación del patrón de accesos que proponemos se ilustra en la Figura 4.22(b). Se puede apreciar cómo el número de *slices* aumenta, dado que es necesario diferenciar los accesos dentro de cada bloque. Mediante esta nueva estrategia, cada uno de los *slices* es analizado y dividido en tantos otros como número de particiones interseque. Específicamente, si se desea particionar a en N_{pt} bloques, y el patrón de acceso tiene un anchura de banda constante de λ *slices*, entonces el aumento del número de *slices* viene dado por la siguiente expresión.

$$\Delta N_S = (N_{pt} - 1)\lambda \tag{4.17}$$

La Figura 4.23 muestra un esquema estructurado de la estrategia `SPRT`. En este trabajo

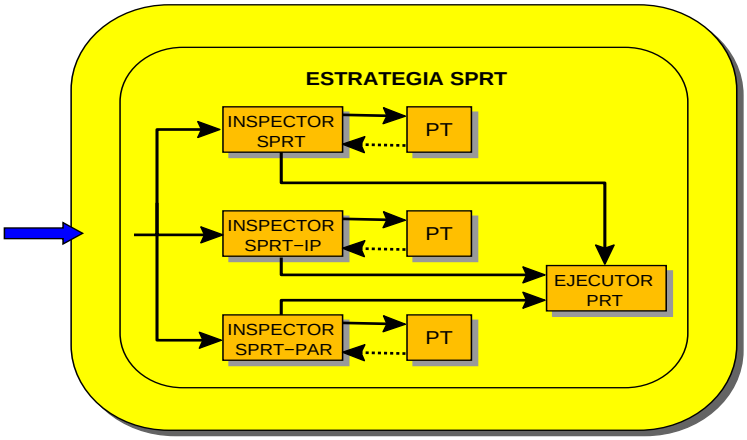


Figura 4.23: Estructura de la estrategia `SPRT`.

presentamos tres algoritmos de inspección. El primero de ellos (inspector `SPRT`) realiza una replicación del vector de indirección, generando una versión reordenada sobre la copia de la indirección original. La segunda propuesta (inspector `SPRT-IP`) realiza la reordenación *in-place* del vector de indirección. La tercera propuesta (inspector `SPRT-PAR`) es una versión paralela del inspector `SPRT`. Todos estos algoritmos invocan a la rutina `PT` para determinar la distribución de las entradas de la indirección sobre cada una de las particiones. En una última etapa de nuestra propuesta, presentamos un ejecutor común a todos estos inspectores. Esta rutina emplea el vector de indirección reordenado para ejecutar en paralelo el lazo original. A continuación mostramos una descripción detallada de cada una de las propuestas de inspector que hemos enumerado.

Inspector `SPRT`

Vamos a denominar a esta nueva técnica de reordenamiento como inspector *Sorted Private Region Technique* (`SPRT`). El inspector `SPRT` consta de dos fases. En la primera se realiza una detección de los *slices* asociados a cada partición, determinándose las regiones exclusivas y compartidas asociadas a cada una de ellas. En la segunda fase se realiza el reordenamiento de las entradas del vector de indirección pertenecientes a las regiones compartidas.

El pseudocódigo del inspector `SPRT` se muestra en la Figura 4.24. El algoritmo recibe como entrada el vector de indirección, su representación IARD y el conjunto de particionamiento. El resultado devuelto consiste en un nuevo vector de indirección (reordenado) y un conjunto de *slices* asociado a cada partición.

```
Algoritmo inspector SPRT
entrada
        x: vector de indirección
        IARD: representación IARD de x
        LIM: conjunto de particionamiento
salida
        x_fin: vector de indireccion
        {s^1, s^2, s^3, s^4}_i  i = 1, N_pt: conjunto de slices asociados a cada partición
inicio del algoritmo
        % Primera fase
           {s^1, s^2, s^3, s^4}_i = PT(IARD, LIM)

        % Segunda fase
           cont = 1
           DO p = 1, N_pt
              DO j = ρ_ini[s_p^1], ρ_ini[s_p^2] − 1                  % región compartida
                 IF(x[j] ∈ A_i)      x_fin[cont++] = x[j]
              END DO
              DO j = ρ_ini[s_p^2], ρ_ini[s_p^3] − 1                  % región exclusiva
                 x_fin[cont++] = x[j]
              END DO
              DO j = ρ_ini[s_p^3], ρ_ini[s_p^4] − 1                  % región compartida
                 IF(x[j] ∈ A_i)      x_fin[cont++] = x[j]
              END DO
           END DO
fin del algoritmo
```

Figura 4.24: Algoritmo inspector *Sorted Private Region Technique* (SPRT).

El lazo principal del algoritmo procesa las diferentes particiones de forma ordenada. La primera fase utiliza el algoritmo PT (Figura 4.11). Para obtener los conjuntos de *slices* $\{s^1, s^2, s^3, s^4\}_i$ que especifican las regiones exclusivas y compartidas asociadas a cada partición $\mathcal{A}_i$.

En la segunda fase de nuestra propuesta, se recorren todas las entradas de x susceptibles de realizar un acceso sobre el bloque considerado. Para un bloque i, estas entradas están comprendidas en el intervalo de *slices* $[s_i^1, s_i^4]$. Sobre cada una de estas entradas se

verifica si realiza el acceso sobre la partición considerada. En caso afirmativo la entrada se almacenada en el nuevo vector de indirección que denominamos x_{fin}. Procesando secuencialmente cada una de las particiones, y dado que las entradas de x se analizan de forma consecutiva, el algoritmo `SPRT` almacena los valores sobre el vector x_{fin} de forma ordenada. Esto implica que la última entrada de x asociada a la partición $\mathcal{A}_i$ es contigua a la primera entrada asociada a la partición $\mathcal{A}_{i+1}$.

Desde el punto de vista de la eficiencia del inspector, una de las principales ventajas de este algoritmo es la alta localidad en los accesos, tanto en la lectura del vector de indirección x, como en la generación del vector resultante x_{fin}, ya que en ambos casos los accesos son sobre elementos consecutivos de los vectores. La complejidad que tiene asociada el algoritmo `SPRT` puede dividirse en las contribuciones debidas a cada una de las fases. De este modo, la complejidad asociada a la primera de ellas coincide con la asociada al algoritmo de particionamiento `PT` (Expresión 4.12). El coste asociado a la segunda fase depende tanto del número de particiones (N_{pt}), como del número de entradas del vector de indirección asociadas a cada una de estas particiones. Específicamente tenemos que el coste neto de esta propuesta es el siguiente:

$$\mathcal{O}_{\texttt{SPRT}} = (N_{pt} - 1)2N_{\mathcal{E}} + \sum_{p=1}^{N_{pt}} (|\mathcal{X}_{s_p^2+1}^{s_p^3-1}| + |\mathcal{X}_{s_p^1}^{s_p^2}| + |\mathcal{X}_{s_p^3}^{s_p^4}|) \tag{4.18}$$

En donde $|\cdot|$ representa al operador cardinalidad. El primer término del sumatorio es el número de entradas de las regiones exclusivas, mientras que el segundo y tercero hacen referencia al conjunto de entradas de las regiones compartidas. Una estimación de estos valores puede hacerse bajo las siguientes aproximaciones.

1. El patrón de acceso puede aproximarse por una banda. Dicha banda esta compuesta por dos rectas separadas por una anchura λ.
2. La distribución de las entradas dentro de la banda es homogénea. Es decir, dentro de la banda no existen zonas con mayor densidad de entradas y están uniformemente distribuida.

Realizando la primera aproximación, el área total de la banda (A_{band}) y el área de la región compartida (A_{comp}) se puede expresar en función del número total de *slices* (longitud de la banda), número de entradas de a (altura de la banda) y anchura del patrón de acceso. En función de estos parámetros, las áreas se pueden aproximar por las siguientes dos expresiones:

$$A_{band} = \lambda \frac{N_a}{N_S}(N_S - \frac{\lambda}{4}) \tag{4.19}$$

$$A_{comp} = \frac{N_a}{N_S}\lambda^2 \tag{4.20}$$

Asumiendo la segunda aproximación, tendremos que:

$$\sum_{p=1}^{N_{pt}}(|\mathcal{X}_{s_p^1}^{s_p^2}| + |\mathcal{X}_{s_p^3}^{s_p^4}|) \simeq 2(N_{pt}-1)N_x\frac{A_{comp}}{A_{band}} \tag{4.21}$$

Donde estamos teniendo en cuenta que la primera y última partición únicamente tienen asociadas una región compartida. El conjunto de entradas exclusivas es, a lo sumo, el número de elementos del vector de indirección. Más formalmente,

$$\sum_{p=1}^{N_{pt}}(|\mathcal{X}_{s_p^2+1}^{s_p^3-1}|) \leq N_x \tag{4.22}$$

Utilizando los resultados de la Ecuación 4.18, obtenemos como complejidad total del algoritmo `SPRT`:

$$O_{\texttt{SPRT}} = (N_{pt}-1)2N_{\mathcal{E}}+N_x+2(N_{pt}-1)N_x\frac{\lambda}{N_s-\lambda/4} \simeq N_x(1+2(N_{pt}-1)\frac{\lambda}{N_s-\lambda/4}) \tag{4.23}$$

Un patrón de acceso genérico siempre puede ser acotado por dos líneas paralelas sin más que asumir dos puntos dominantes en cada envolvente del IARD. Si el patrón de acceso tiene un contorno suficientemente bandeado, y el número de entradas por *slice* no presenta fuertes cambios[3], entonces la Expresion 4.23 puede ser aplicada con suficiente precisión.

Nótese que si $\lambda \ll N_S$, $N_{pt}N_{\mathcal{D}} \ll N_x$ y $N_a \simeq N_S$, entonces la Expresión 4.23 se puede aproximar por $\mathcal{O}_{\texttt{SPRT}} \simeq 2N_x$.

El coste de almacenamiento del inspector `SPRT`, denotado por $\Delta M_{\texttt{SPRT}}^{inspector}$, es elevado. Esto es debido a que es necesario replicar el vector de indirección x y generar el conjunto de *slices* asociados a cada partición. Específicamente,

$$\Delta M_{\texttt{SPRT}}^{inspector} = N_x + 4N_{pt} \tag{4.24}$$

Usualmente el número de entradas del vector de indirección es mucho mayor que el número de particiones, por lo que $\Delta M_{\texttt{SPRT}}^{inspector} \simeq N_x$.

Inspector `SPRT-IP`

Nuestra segunda propuesta, denominada *Sorted Private Region Technique in Place* (`SPRT-IP`), obtiene el mismo resultado que en el caso previo reordenando el vector de

[3]Típicamente, esta propiedad se verifica por un gran número de patrones de acceso, incluidos los que hemos utilizado en la evaluación de nuestra propuesta.

indirección. El pseudocódigo del algoritmo `SPRT-IP` se muestra en la Figura 4.25. Del mismo modo que en el caso anterior, los argumentos de entrada son el vector de indirección, su representación IARD y el conjunto de particionamiento.

El primer paso consiste en la obtención del conjunto de *slices* asociados a cada partición, los cuales se calculan empleando nuevamente el algoritmo `PT`. A continuación, y por medio del algoritmo inspector `PRT` (Figura 4.14), se realiza un reordenamiento parcial de las entradas de los *slices* situados en las regiones compartidas. Para cada *slice*, sus entradas se agrupan en tantos conjuntos como número de bloques son accedidos por el mismo. Dentro de cada conjunto, las entradas están dispuestas en memoria de modo contiguo, tal y como se describió con anterioridad. Adicionalmente, el algoritmo `PRT` genera el conjunto de vectores $corte_i$, con $1 < i < N_{pt}$.

La segunda fase consiste en una permutación de las entradas del vector de indirección. Los distintos elementos de x se recorren ordenadamente, y se evalúa su posición en el vector reordenado. Para cada uno de ellos se comprueba si los accesos se realizan dentro de la partición considerada ($\mathcal{A}_i$). En caso afirmativo (línea etiquetada como `L1`) la entrada actual del vector de indirección no es modificada, pasándose a evaluar la siguiente. En caso negativo, es necesario mover la entrada a la posición que le corresponde en el vector reordenado. Esta posición se almacena en la variable temporal k', la cual se puede expresar en función de tres contribuciones denotadas por las funciones $C1$, $C2$ y $C3$, las cuales describimos a continuación.

Vamos a utilizar la Figura 4.26 para ilustrar las distintas contribuciones que es necesario considerar para obtener la nueva posición para una entrada del vector de indirección. En esta figura se representa el patrón de acceso extraído de la matriz *3dtube*, el cual ha sido particionado en dos bloques de distinto tamaño. Concretamente, tenemos que $lim_1^{sup} = 20\text{k}$ y $lim_2^{sup} = 45\text{k}$. Las regiones destacadas en la figura como C1, C2 y C3 representan el conjunto de entradas que hay que considerar para evaluar la nueva posición de la entrada $x[i]$.

La contribución de la función $C1$ se corresponde al número de entradas existentes en la región etiquetada como "C1". Si denotamos por Δx_i al conjunto de entradas del vector x que acceden en el bloque $\mathcal{A}_i$, entonces,

$$C1(i) = \sum_{j=1}^{i-1} |\Delta x_j| \tag{4.25}$$

Este valor se almacena en un vector que contiene el número de entradas de cada una de las particiones. La obtención de este vector se realiza de modo simultáneo a la generación de los vectores de *corte*, y no tiene un coste añadido.

```
Algoritmo inspector SPRT-IP
entrada
        x: vector de indirección
        IARD: representación IARD de x
        LIM: conjunto de particionamiento
salida
        x: vector de indirección original reordenado
        {s^1, s^2, s^3, s^4}_i  i = 1, N_pt: conjunto de slices asociados a cada partición
inicio del algoritmo

        % Primera fase
        {x, corte_i, s^1, s^2, s^3, s^4} ⇐ PRT(IARD, x, LIM)

        % Segunda fase
        j = ρ_ini[s_1^1]
        i = 1
         WHILE(j ≤ N_x)
L1              IF(x[j] ∈ A_i)    j++
                ELSE
                    k = j
                    DO
                      k' = C1(i) + C2(k, corte_p) + C3(k, corte_p)
                      intercambia(x[k], x[k'])
                      k = k'
                    WHILE(j ≠ k')
                END IF
L2              IF(j ≥ ρ_ini[s_i^2 + 1])
                  i++
                  j = ρ_ini[s_i^1]
                END IF
          END WHILE
fin del algoritmo
```

Figura 4.25: Algoritmo inspector *Sorted Private Region Technique in Place* (SPRT-IP).

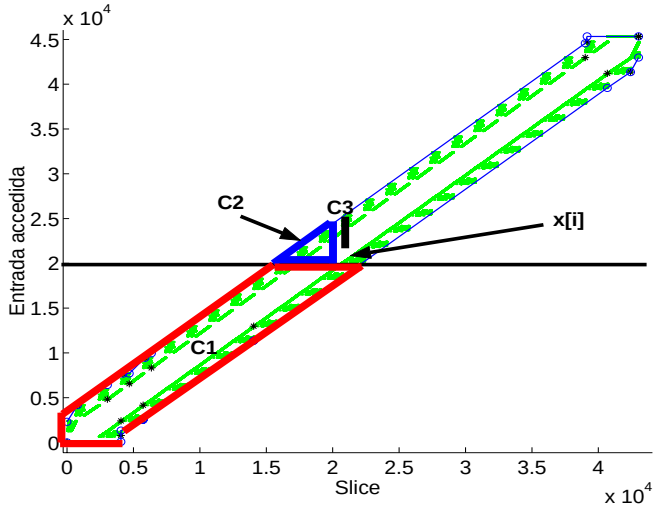


Figura 4.26: Ejemplo de las contribuciones a la nueva posición de cada punto.

La segunda contribución, $C2$, obtiene el número de entradas existentes en todos los *slices* de la partición al que el elemento pertenece y que son previos al mismo. La Figura 4.26 muestra estas entradas como las situadas dentro de la región etiquetada como "C2". Para cada elemento, es necesario obtener el *slice* al que pertenece, conociendo únicamente la partición en la que se encuentra (dada por la rutina anterior) y, por lo tanto, conociendo el intervalo de *slices* $[s_i^1, s_i^2]$ en el que el acceso $x[i]$ aparece. Para este tipo de problemas, hemos utilizado la estrategia de búsqueda binaria [81] dado su reducido tiempo de procesamiento.

La tercera de las contribuciones, $C3$, obtiene el puesto que ocupa el elemento dentro del *slice* considerado. Nuevamente, la Figura 4.26 ilustra esta contribución como el intervalo de entradas "C3". La obtención de este valor se realiza obteniendo la diferencia entre el índice de la entrada considerada y el índice de la primera entrada perteneciente al mismo *slice* que accede dentro del bloque considerado. El valor de esta entrada está almacenado en el vector $corte_p$.

Una vez obtenida la nueva posición se procede al intercambio de los valores mediante la función *intercambia*. El elemento procesado pasa a ocupar la posición k' mientras que el elemento que ocupaba dicha posición pasa a ser el evaluado por el algoritmo. De este modo, se vuelve a repetir el mismo proceso para este nuevo punto con el fin de obtener su posición final en el vector reordenado. Este proceso continúa hasta encontrar una entrada situada en el mismo hueco de la primera que originó el proceso de permutación (posición

almacenada en el índice j). Una vez detectada esta situación, el proceso de intercambio se detiene y se vuelve a iniciar la fase de comprobación: partiendo de la última entrada mal ubicada (entrada j) se vuelve a repetir el proceso anteriormente descrito.

De acuerdo con la Propiedad 4.2.1, todos los elementos de la indirección pertenecientes a la región exclusiva únicamente acceden sobre la partición considerada, por lo que no es necesario reordenarlos. De este modo, el algoritmo `SPRT-IP` únicamente analiza aquellas entradas de la indirección pertenecientes a regiones compartidas (línea etiquetada como `L2` en la Figura 4.25).

Considerando la complejidad de la propuesta, vamos a evaluar de nuevo la complejidad de cada una de las partes que lo componen. La complejidad de la primera fase viene dada por la Expresión 4.13. Adicionalmente, la complejidad de la función $C1$ viene dada por la necesidad de recorrer un vector con N_{pt} entradas,

$$\mathcal{O}_{C1} = N_{pt} \tag{4.26}$$

Por otra banda, la complejidad máxima de la función $C2$ se corresponde a la del algoritmo de búsqueda binaria utilizado. Esta viene dada por la siguiente expresión:

$$\mathcal{O}_{C2} = log_2(s_i^2 - s_i^1) - 1 \simeq log_2(\lambda) - 1 \tag{4.27}$$

Finalmente, la última de las funciones necesita analizar, a lo sumo, $N_{pt} - 1$ vectores *corte* antes de encontrar el puesto del elemento dentro del *slice*. De este modo, esta función tiene una complejidad:

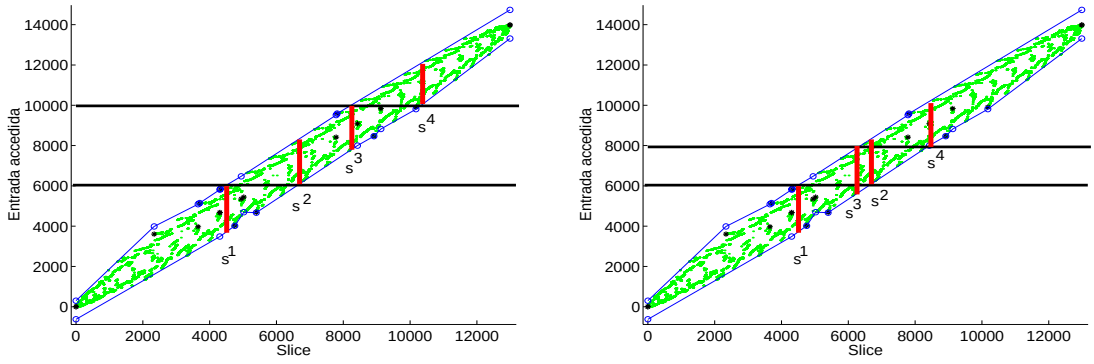
$$\mathcal{O}_{C3} = N_{pt} - 1 \tag{4.28}$$

Notar que estas rutinas de cálculo son ejecutadas tantas veces como entradas existen en las regiones compartidas asociadas a cada corte. De este modo, la complejidad del algoritmo `SPRT-IP` se puede expresar en función de cada uno de sus bloques funcionales:

$$\mathcal{O}_{\texttt{SPRT-IP}} = \mathcal{O}_{\texttt{PRT}} + \sum_{p=1}^{N_{pt}} (|\mathcal{X}_{s_p^1}^{s_p^2}|)(\mathcal{O}_{C1} + \mathcal{O}_{C2} + \mathcal{O}_{C3}) \tag{4.29}$$

La expresión anterior puede simplificarse asumiendo un patrón de acceso bandeado con una anchura λ y una distribución homogénea de las entradas dentro de la banda. Notar que si $\lambda \ll N_S$, $N_{pt}N_{\mathcal{D}} \ll N_x$ y $N_a \simeq N_S$, la expresión anterior se puede simplificar, obteniendo esta nueva relación:

$$\mathcal{O}_{\texttt{SPRT-IP}} \simeq N_x(N_{pt} - 1)\frac{\lambda}{N_S - \lambda/4}(2N_{pt} + log_2(\lambda)) \tag{4.30}$$



(a) (b)

Figura 4.27: Distintos particionamientos para la representación IARD extraída de la matriz *bcsstk29*.

El algoritmo `SPRT-IP` presenta como principal desventaja respecto al `SPRT`, una complejidad proporcional al cuadrado del número de particiones. Adicionalmente, y debido a la existencia de operaciones de permutación, la localidad en los accesos a x durante el proceso de reordenamiento es reducida. Una de las principales ventajas de este algoritmo es la no necesidad de procesar todo el vector de indirección, dado que únicamente es necesario operar sobre sus regiones compartidas. Si el patrón de acceso tiene una banda estrecha, o si el número de particiones es reducido, el tiempo de ejecución puede ser inferior al del algoritmo `SPRT`. Finalmente, y dado que el algoritmo `SPRT-IP` se aplica sobre la indirección, su coste de almacenamiento (dado por $\Delta M^{inspector}_{\texttt{SPRT-IP}}$) es mucho menor que el del algoritmo `SPRT`. Este coste viene dado por:

$$\Delta M^{inspector}_{\texttt{SPRT-IP}} = 4N_{pt} \tag{4.31}$$

Inspector paralelo `SPRT-PAR`

En esta sección únicamente vamos a abordar la paralelización del algoritmo `SPRT`. El algoritmo `SPRT-IP` no va a ser considerado debido a que su organización interna hace muy difícil el desarrollo de una implementación paralela.

Consideremos la Figura 4.27(a), la cual muestra la representación IARD de la matriz *bcsstk29* particionada en tres bloques. La aplicación del algoritmo `SPRT` sobre este patrón de accesos implica reordenar únicamente las dos regiones compartidas definidas por la intersección de las rectas $y = lim_1^{sup} = 6000$ y $y = lim_2^{sup} = 10000$ con la envolvente. Estas

regiones están comprendidas, respectivamente, en los intervalos de *slices* $[s^1, s^2]$ y $[s^3, s^4]$. En el caso de la Figura 4.27(a), tenemos que $s^3 > s^2$ lo que implica que las regiones compartidas asociadas a cada límite son regiones inconexas. Más formalmente, $\mathcal{X}_{s^1}^{s^2} \cap \mathcal{X}_{s^3}^{s^4} = \emptyset$. La aplicación del algoritmo `SPRT` sobre un patrón de acceso en el que todas las regiones compartidas son inconexas permite la paralelización inmediata del proceso. Nuestra propuesta reordena las entradas de cada región compartida de modo que la posición final de las entradas sigue perteneciendo a la misma región compartida. De este modo, y dado que se pueden reordenar las entradas dentro de cada región compartida de forma independiente, es posible realizar este proceso en paralelo. El desarrollo de una versión paralela del algoritmo `SPRT` únicamente implica reemplazar el lazo DO de la línea etiquetada como `L2` por un lazo DOALL. Del mismo modo, las entradas de las regiones exclusivas no deben ser reordenadas, sino que son directamente copiadas en el vector reordenado, ocupando la misma posición que tenían en el vector de indirección original. La única modificación relevante, consiste en el valor inicial de la variable *cont* (etiqueta `L1`, Figura 4.28). En la versión paralela del algoritmo esta variable tendrá un valor diferente para cada procesador que se corresponde con la posición de comienzo del primer elemento considerado. Este valor será la suma de todas las entradas del vector de indirección pertenecientes a regiones compartidas y exclusivas de particiones anteriores. Más formalmente,

$$cont_p = \sum_{i=1}^{p-1} (\mathcal{X}_{s_i^1}^{s_i^2} + \mathcal{X}_{s_i^2+1}^{s_i^3-1}) \tag{4.32}$$

En el caso de la Figura 4.27(a), esta expresión puede obtenerse fácilmente a través del vector de densidades acumuladas,

$$cont_p = \sum_{i=1}^{p-1} ((\rho_{ini}[s_i^2] - \rho_{ini}[s_i^1] + 1) + (\rho_{ini}[s_i^3 - 1] - \rho_{ini}[s_i^2 + 1] + 1)) \tag{4.33}$$

En aplicaciones reales, este esquema de particionamiento no aparece en todos los casos. Consideremos ahora el patrón de acceso mostrado en la Figura 4.27(b). En este caso $lim_1^{sup} = 6000$ y $y = lim_2^{sup} = 8000$, lo que origina que $s^3 < s^2$. Las regiones compartidas primera y segunda se solapan, por lo que $\mathcal{X}_{s^1}^{s^2} \cap \mathcal{X}_{s^3}^{s^4} \neq \emptyset$ y $\mathcal{X}_{s^2+1}^{s^3-1} = \emptyset$. La estrategia de paralelización previa no puede aplicarse a este esquema de particionamiento, dado que las expresiones 4.32 y 4.33 ya no resultan válidas. Para esquemas de particionamiento como este, en el que las regiones compartidas se solapan, es necesario utilizar otra estrategia de paralelización. En nuestro caso hemos adoptado la técnica de *array splitting and merging* [93]. Mediante esta técnica podemos adaptar nuestra estrategia a situaciones en las que existe solape entre las regiones compartidas. El algoritmo paralelo resultante, que denominamos *Sorted Private Region Technique* paralelo (`SPRT-PAR`), se muestra en

la Figura 4.28. En esta propuesta se han añadido dos nuevas fases de cálculo a las ya existentes. En la segunda fase, nuevamente el lazo que recorre las particiones se ejecuta en paralelo, pero ahora cada procesador almacena en un vector compartido (x_{tmp}) la sección del vector que ha reordenado. Adicionalmente, en la misma fase, cada procesador almacena en un contador privado (mi_cont) el número de entradas que procesa, es decir, el número de entradas de las regiones compartidas que acceden a la partición considerada. En el ejemplo mostrado en la Figura 4.28, estamos asumiendo la no existencia de regiones exclusivas, por lo que fusionamos en un solo lazo el intervalo de entradas.

En una tercera etapa, el número de entradas reordenadas por cada procesador es comunicado globalmente y acumulado en un vector global denominado *desplazamiento*, el cual contiene para su entrada i-ésima, el número de accesos asociados a todas las particiones previas. Finalmente, en la cuarta fase son copiadas las contribuciones de cada procesador sobre el resultado. Mediante el empleo del vector *desplazamiento*, se asegura que cada procesador escribe sobre regiones diferentes del resultado.

El rendimiento del algoritmo `SPRT-PAR` depende, en gran medida, del grado de solape existente entre las regiones compartidas. Cada procesador p, debe analizar $|\mathcal{X}_{s_p^1}^{s_p^4}|$ entradas. Este valor depende fuertemente de λ, la anchura de banda, de modo que conforme la esta anchura aumenta, el número de computaciones replicadas también crece. Dicha replicación disminuye la eficiencia del inspector paralelo.

Por otra parte, el coste de la tercera y cuarta fase es constante dado el alto desacople entre los procesadores. Estas fases tienen un coste por procesador aproximadamente igual al número de entradas del vector de indirección que acceden a la partición considerada. Asumiendo una distribución homogénea, el coste asociado al algoritmo paralelo viene dado por:

$$\mathcal{O}_{\texttt{SPRT-PAR}} = \mathcal{O}_{\texttt{PT}} + |\mathcal{X}_{s_p^1}^{s_p^4}| + N_{pt} + \frac{N_x}{N_{pt}} \tag{4.34}$$

Ejecutor `SPRT`

Como punto de partida vamos a considerar un acceso irregular genérico x. Aplicando el algoritmo de reordenamiento `SPRT` sobre el vector de indirección, obtendremos el nuevo vector reordenado x_{fin}. En la Figura 4.22(b) podemos apreciar que tras reordenar el vector de indirección el número de columnas del patrón de acceso experimenta un aumento. El ejecutor paralelo que proponemos se muestra en la Figura 4.29. Para poder acceder a las entradas del vector de indirección reordenado se utiliza el vector col_{fin}, el cual se corresponde al vector de densidad acumulada asociado a x_{fin}. Dicho en otras palabras, el

```
Algoritmo inspector SPR-PAR
entrada
        x: vector de indirección
        IARD: representación IARD de x
        LIM: conjunto de particionamiento
salida
        x_fin: vector de indireccion
        {s^1, s^2, s^3, s^4}_i  i = 1, N_pt: conjunto de slices asociados a cada partición
inicio del algoritmo
        % Primera fase
           {s^1, s^2, s^3, s^4}_i = PT(IARD, LIM)

        % Segunda fase
           DOALL p = 1, N_pt
              mi_cont[p] = 1
              DO j = ρ_ini[s_p^1], ρ_ini[s_p^4] − 1
                 IF (x[j] ∈ A_i)      x_tmp[mi_cont[p] + +, p] = x[j]
              END DO
           END DOALL

        % Tercera fase
           DO p = 2, N_pt
              desplazamiento[i] = desplazamiento[i − 1] + mi_cont[i − 1]
           END DO

        % Cuarta fase
           DOALL p = 1, N_pt
              DO j = 1, mi_cont[p]
                 x_fin[desplazamiento[i] + j] = x_tmp[j, p]
              END DO
           END DOALL
fin del algoritmo
```

Figura 4.28: Algoritmo inspector *Sorted Private Region Technique* paralelo (SPRT-PAR).

```
DOALL p = 1, N_P
   DO i = inicio[p], inicio[p+1] - 1
      DO j = col_fin[i], col_fin[i+1] - 1
         ... a[x_fin[j]] ...
         { ... }
      END DO
   END DO
END DOALL
```

Figura 4.29: Código ejecutor de la estrategia SPRT.

intervalo $[col_{fin}[s], col_{fin}[s+1])$ delimita el conjunto de entradas del vector x_{fin} pertenecientes al *slice* s. El número de elementos del vector col_{fin} (parámetro N_{col}) es la suma del número de *slices* accedidos por cada procesador. Más formalmente,

$$N_{col} = \sum_{i=1}^{N_p} s_i^4 - s_i^1 \tag{4.35}$$

Este vector se genera de forma simple por el propio inspector (ya sea el SPRT, el SPRT-IP o el SPRT-PAR). Supongamos que procesamos la entrada $x[j]$ del vector de indirección original, la cual pertenece al *slice* s. Supongamos además que esta entrada se asigna a la partición p y se almacena en $x_{fin}[j']$. Entonces el número de *slice* s', al que pertenece esta entrada reordenada viene dado por la siguiente expresión:

$$s' = s + \sum_{i=1}^{p} (s_i^2 - s_i^1) \tag{4.36}$$

Es decir, cada partición i introduce un aumento de $s_i^2 - s_i^1$ en el número de *slices* de la nueva distribución. En el caso de la primera partición, su contribución es nula, ya que $s_1^2 = s_1^1$ por tratarse del límite inferior del patrón de acceso. La Figura 4.22(b) muestra un ejemplo en el que el patrón de acceso ha sido dividido en dos particiones. En este caso el aumento del número de *slices* se debe a la anchura de la banda en franja central de corte. Nótese que estos cálculos se realizan empleando únicamente la información del IARD.

De este modo, cada vez que la rutina de inspección procesa una entrada del vector de indirección, es posible determinar el número de *slice* asociado a la nueva distribución mediante el empleo de la Expresión 4.36. Una vez conocido este número de *slice*, el vector col_{fin} puede ser convenientemente actualizado.

Utilizando dicho vector en conjunción con la información almacenada en la caracterización IARD, podemos desarrollar la versión paralela del código mostrada en la Figura 4.29.

El lazo externo divide el espacio de trabajo en tantos bloques como número de particiones existentes. Mediante la aplicación de la regla del propietario asumimos que cada una de estas particiones está asociada a un procesador.

El vector *inicio* almacena en su entrada p-ésima la posición del primer *slice* de x_{fin} que accede a la partición $\mathcal{A}_p$. Esta posición viene dada por la suma del número de *slices* asociados a todas las particiones anteriores. Expresado de un modo más formal, los valores de este vector se pueden obtener en función de los *slices* de corte mediante la siguiente expresión:

$$inicio[p] = 1 + \sum_{i=1}^{p-1} s_i^4 - s_i^1 \tag{4.37}$$

Adicionalmente, se verifica que $inicio[1] = 1$ y $inicio[N_p+1] = N_{col}+1$. La Figura 4.22(b) muestra un ejemplo de la posición del primer *slice* asociado a la segunda partición. Esta posición se corresponde con el valor almacenado en $inicio[2]$.

Debido a que el vector de indirección está reordenado, cada procesador realiza todos sus accesos sobre elementos consecutivos y regiones disjuntas de x. Así pues, mediante el uso de la técnica `SPRT` se logra aumentar significativamente la localidad en los accesos al vector de indirección, manteniendo la localidad en las escrituras en a. Como veremos posteriormente, esto se va a traducir en una reducción importante de los tiempos de ejecución del programa.

Análisis de eficiencia

En esta sección realizamos una evaluación del rendimiento obtenido mediante nuestras propuestas y el alcanzado con las técnicas *array expansion* y DWA-LIP. Como plataforma de medida, hemos utilizado un sistema multiprocesador de memoria compartida SUN HPC4500 con 10 procesadores UltraSPARC II a 400 MHz. El código utilizado fue escrito y compilado en Fortran 77 con el compilador WorkShop `f77` v4.2. Los códigos paralelos fueron programados utilizando las directivas de memoria compartida.

Respecto a la estrategia LOCALWRITE, aunque no disponemos de resultados experimentales, si podemos exponer algunas consideraciones teóricas. Por una parte, nuestro algoritmo de inspección es mucho más eficiente, dado que únicamente considera, gracias al empleo de la representación IARD, aquellas regiones asociadas a cada una de las particiones. En contraposición con las copias privadas del vector de indirección que son generadas por la técnica de LOCALWRITE, nuestra propuesta crea un vector de indirección reordenado, que es compartido por todos los procesadores. Esta opción deja abierta la posibilidad

```
DO j = 1, Nx
      a[x[j]] = a[x[j]] ⊗ ...
END DO
```

(a)

```
DOALL p = 1, NP
   DO i = inicio[p], inicio[p+1] - 1
      DO j = col_fin[i], col_fin[i+1] - 1
         a[x_fin[j]] = a[x_fin[j]] ⊗ ...
      END DO
   END DO
END DOALL
```

(b)

Figura 4.30: Reducción irregular: (a) versión secuencial, (b) código paralelo aplicando la estrategia IARD–SPRT.

de que el resto del programa haga uso del mismo reemplazando al vector de indirección original, lo que disminuiría los costes de almacenamiento del inspector. Finalmente, y como veremos a continuación, nuestra propuesta da soporte a la paralelización de códigos irregulares que hacen uso de esquemas de almacenamiento matricial, permitiendo preservar la estructura del patrón de acceso original.

En este trabajo hemos utilizado tres modelos de código para evaluar la eficiencia de nuestras propuestas: la reducción irregular, el producto matriz dispersa vector y la transposición de una matriz dispersa. Las reducciones irregulares son el núcleo de una gran número de aplicaciones comerciales. El producto matriz dispersa-vector, denotado como *spmxv*, aparece en la resolución de sistemas lineales mediante resolutores iterativos, mientras que la operación de transposición es un ejemplo de código que opera con estructuras matriciales dispersas. Esta última rutina se corresponde al DO6 de la rutina *csrcsc*2 de la Sparse Kit Collection [122]. El pseudocódigo de la versión secuencial y paralela de estos programas de prueba se ilustra, respectivamente, en las figuras 4.30, 4.31 y 4.32. Estos códigos se pueden agrupar en dos categorías: los que operan con una única indirección (como la reducción irregular), y las rutinas *spmxv* y *csrcsc2*, las cuales operan sobre una matriz dispersa. Las estrategias de paralelización para cada categoría presentan pequeñas diferencias respecto al procedimiento de elaboración de la representación IARD, las cuales se describirán a continuación.

La primera categoría representa los códigos irregulares en los que el esquema de acceso viene dado por un vector de indirección. Para estos casos el código se puede paralelizar mediante una representación IARD del vector de indirección basada en el algoritmo CS,

```
DO i = 1, N_S
   DO j = col[i], col[i+1] - 1
      k = row[j]
      a[k] = a[k] + b[i] * val[j]
   END DO
END DO
```

(a)

```
DOALL p = 1, N_P
   DO i = inicio[p], inicio[p+1] - 1
      DO j = col_fin[i], col_fin[i+1] - 1
         k = row_fin[j]
         a[k] = a[k] + b[i-ancho[p]] * val_fin[j]
      END DO
   END DO
END DOALL
```

(b)

Figura 4.31: Rutina *spmxv*: (a) versión secuencial, (b) versión paralela mediante la estrategia IARD–SPRT.

```
DO i = 1, N_S
   DO j = col[i], col[i+1] - 1
      k = row[j]
      next = col[k]
      val'[next] = val[j]
      row'[next] = i
      col'[k] = next + 1
   END DO
END DO
```

(a)

```
DOALL p = 1, N_P
   DO i = inicio[p], inicio[p+1] - 1
      DO j = col_fin[i], col_fin[i+1] - 1
         k = row_fin[j]
         next = col_fin[k]
         val'[next] = val_fin[j]
         row'[next] = i - ancho[p]
         col'[k] = next + 1
      END DO
   END DO
END DOALL
```

(b)

Figura 4.32: Rutina *csrcsc2*: (a) versión secuencial, (b) versión paralela mediante la estrategia IARD–SPRT.

introducido en la Sección 2.3.1[4]. La Figura 4.30(b) muestra un código paralelo perteneciente a esta categoría. En este caso se puede reducir el nivel de anidamiento fusionando los dos lazos más internos. Esta transformación puede ser aplicada debido a que las entradas del vector de indirección asignada a cada procesador ocupan posiciones consecutivas de memoria, por lo que únicamente es necesario indicar la primera y última entrada ac-

[4]Los códigos pertenecientes a la primera categoría también pueden ser paralelizados con una representación IARD que emplee el algoritmo CSM. La única diferencia es que el aumento del número de *slices* puede originar una leve disminución del rendimiento del inspector y del ejecutor.

cedida. Siendo más precisos, cada procesador p, debe considerar el intervalo de entradas $[col_{fin}[inicio[p]], col_{fin}[inicio[p+1]] - 1]$ del vector de indirección.

La estrategia de paralelización utilizada para la segunda categoría varía sensiblemente respecto a la primera. En los ejemplos considerados tenemos un formato de almacenamiento por columnas (formato CCS [13]). En dicho formato se emplean los vectores *val*, *col* y *row* para almacenar la estructura de la matriz dispersa. Con el fin de preservar esta información, la representación IARD debe utilizar el algoritmo `CSM` introducido en la Sección 2.3.2. Adicionalmente, para los códigos de la segunda categoría es necesario tener en cuenta que la indirección, tras ser reordenada, sufre cambios en su distribución por columnas. Las figuras 4.31(b) y 4.32(b) muestran el código paralelo correspondiente a las rutinas *spmxv* y *csrcsc*2. Del mismo modo que en el caso anterior, se ha incorporado un lazo externo que divide el espacio de trabajo en N_{pt} particiones. En el código original el vector *col* se encarga de clasificar por columnas las entradas de la indirección. El ejecutor paralelo que proponemos reemplaza el vector *col* original por el vector col_{fin} que representa el vector de densidad acumulada correspondiente a la nueva distribución. Tal y como se comentó con anterioridad, este vector tiene un mayor número de elementos, ya que el patrón de acceso reordenado experimenta un aumento del número de *slices*.

Con el fin de corregir este efecto se utiliza el vector *ancho*. Específicamente, la entrada i-ésima de *ancho* contiene el número de *slices* existentes en la intersección del límite inferior de la i-ésima partición con el patrón de indirección. De este modo, los valores almacenados en este vector vienen dados por:

$$ancho[p] = \sum_{i=1}^{p} s_i^2 - s_i^1 \tag{4.38}$$

Del mismo modo que en la primera categoría, cada procesador accede a elementos contiguos y regiones disjuntas del vector de indirección. Sin embargo, el coste de almacenamiento se incrementa respecto al primer caso, dado que ahora es necesario almacenar un mayor número de entradas en el vector col_{fin}. Si la anchura de la banda es elevada, o si se emplea un gran número de procesadores, este coste de memoria puede ser mucho mayor que el de la primera categoría y resulta comparable al de la estrategia `PRT`.

En esta segunda categoría también resulta necesario modificar los algoritmos `SPRT` y `SPRT-IP` para reordenar, además del vector *row* (que representa nuestro vector de indirección), el vector *val*. El reordenamiento de este último vector es el mismo que el realizado en la indirección, y se puede hacer de modo simultáneo al de esta. Nótese que la rutina *csrcsc2* realiza operaciones no asociativas, la cual no es compatible con el empleo de la técnica *array expansion*.

El coste de almacenamiento del inspector SPRT extendido a matrices dispersas aumenta, debido a la necesidad de reordenar el vector *val*. Específicamente, tenemos un coste dado por la siguiente expresión:

$$\Delta M^{inspector}_{\texttt{SPRT}} = 2N_x + N_{col} + 4N_{pt} \tag{4.39}$$

En donde el primer término refleja el coste de almacenamiento de los vectores row_{fin} y val_{fin}, el segundo representa el tamaño del vector col_{fin} (dado por la Expresión 4.35), y el último es el coste asociado a los *slices* que delimitan cada partición.

Usualmente, el número de entradas del vector de indirección es mucho mayor que el número de *slices*, por lo que $\Delta M^{inspector}_{\texttt{SPRT}} \simeq 2N_x$.

Por otra parte, y dado que el algoritmo SPRT-IP realiza la reordenación de ambos vectores (*row* y *val*), su coste de almacenamiento es de nuevo mucho más reducido. El término N_S descuenta el coste de almacenamiento asociado al vector *col*, el cual ya no es necesario conservar.

$$\Delta M^{inspector}_{\texttt{SPRT-IP}} = N_{col} - N_S + 4N_{pt} \tag{4.40}$$

Como datos de entrada, hemos utilizado los mismos patrones de acceso que los empleados en la estrategia PRT y cuyas características se muestran en las tablas 2.4 y 4.3. Para el caso de los códigos *spmxv* y *csrcsc*2, fue necesario operar con la representación IARD basada en el algoritmo CSM. La Tabla 4.6 muestra el número de *slices* (N_S), el número de puntos dominantes ($N_{\mathcal{D}}$), y el tiempo de ejecución empleado en la generación de la nueva representación IARD. El cálculo de la envolvente superior e inferior se ha realizado en paralelo.

Las tablas 4.7 y 4.8 muestran, para cada uno de los códigos de prueba, el tiempo de ejecución secuencial y el tiempo de ejecución paralelo con 8 particiones. Asumimos un particionamiento del vector a en bloques de igual tamaño. Se puede apreciar cómo

	Matriz							
	3dtube	*af23560*	*bcsstk14*	*bcsstk17*	*bcsstk29*	*nasasrb*	*s3dkq4m2*	*struct3*
N_S	45330	23562	1806	10973	13994	54870	90449	53570
$N_{\mathcal{D}}$	2	10	19	9	10	11	3	4
IARD (ms)	1352	428	37	215	289	1045	1219	375

Tabla 4.6: Características del conjunto de matrices dispersas y tiempo de caracterización IARD (en ms) empleando el algoritmo CSM.

la estrategia SPRT resulta la más competitiva en todas las situaciones, incluida ahora la matriz *bcsstk14* con la que el algoritmo PRT era superado por la propuesta DWA-LIP.

La mayor desventaja que presenta la técnica PRT es la degradación de su rendimiento cuando la anchura del patrón de acceso (λ) resulta elevada. Con el fin de analizar la influencia de este parámetro en el rendimiento hemos utilizado la familia de matrices sintéticas

Código	Estrategia	Matriz			
		3dtube	*af23560*	*bcsstk14*	*bcsstk17*
reducción irregular	Secuencial	125.2	12.0	1.4	10.2
	DWA-LIP	28.1	2.8	**0.3**	2.3
	Array Exp.	20.6	4.6	0.5	3.3
	PRT	19.0	2.1	0.4	2.4
	SPRT	**12.5**	**1.8**	**0.3**	**1.9**
spmxv	Secuencial	287.9	36.6	2.6	26.8
	Array Exp.	37.0	7.6	0.6	4.1
	PRT	42.3	3.7	0.5	3.2
	SPRT	**33.4**	**3.1**	**0.4**	**2.9**
*csrcsc*2	Secuencial	458.4	58.9	2.8	51.6
	PRT	88.4	5.5	0.5	4.2
	SPRT	**79.7**	**5.1**	**0.4**	**3.5**

Tabla 4.7: Tiempos de ejecución del ejecutor para 8 procesadores (en *ms*).

Código	Estrategia	Matriz			
		bcsstk29	*nasasrb*	*s3dkq4m2*	*struct3*
reducción irregular	Secuencial	14.7	103.6	185.0	35.0
	DWA-LIP	3.5	20.0	48.5	6.0
	Array Exp.	4.5	21.2	33.7	12.0
	PRT	19.0	11.7	22.0	5.5
	SPRT	**2.4**	**10.7**	**18.5**	**4.9**
spmxv	Secuencial	51.3	234.7	418.3	106.7
	Array Exp.	6.9	32.3	62.4	17.2
	PRT	8.4	28.6	56.9	13.1
	SPRT	**5.2**	**26.5**	**53.1**	**11.8**
*csrcsc*2	Secuencial	77.4	350.0	619.7	157.2
	PRT	11.5	65.1	**122.9**	19.6
	SPRT	**9.2**	**63.5**	126.4	**17.9**

Tabla 4.8: Tiempos de ejecución del ejecutor para 8 procesadores (en *ms*).

mostrada en la Figura 4.20. Estas matrices se caracterizan mediante dos parámetros: el número de entradas no nulas (N_x) y la anchura de la banda (λ). Los valores del primer parámetro varían entre 500k y 10M, mientras que λ tiene un rango de 5 a 75k. El número de columnas es el mismo en todos los casos y tiene un valor de 100k. Todas estas matrices fueron caracterizadas utilizando el algoritmo `CSM`, por lo que se verifica que el número de *slices* coincide con el de columnas de la matriz dispersa. Para el caso de las matrices utilizadas, N_S es igual a 10^5 en todos los casos. El rendimiento del algoritmo `CSM` es muy próximo al obtenido en el `CS`. Las tablas 4.9 4.10 y 4.11 muestran el tiempo de ejecución de nuestro ejecutor paralelo para $N_p = 10$ con los patrones de acceso sintéticos. Notar que nuestra propuesta resuelve el inconveniente del algoritmo `PRT` y obtiene los mejores resultados para la práctica totalidad de los casos con independencia de la anchura de la banda y el número de entradas de la indirección.

El otro aspecto que es necesario considerar lo supone el coste asociado al inspector. Las tablas 4.13 y 4.12 muestran el tiempo de ejecución del algoritmo `SPRT` para distinto número de particiones. En todos los casos se utilizó una versión secuencial del inspector. Con el fin de comparar nuestra rutina de inspección con las de otras propuestas, la tablas 4.13

N_x	Estrategia	*reducción irregular*			
		λ=1k	λ=10k	λ=25k	λ=50k
10M	DWA–LIP	85.4	140.5	211.8	316.8
	Array Exp.	56.1	67.1	72.7	73.1
	PRT	34.6	69.1	91.1	117.7
	SPRT	**33.6**	**44.2**	**48.9**	**50.6**
5M	DWA–LIP	37.8	68.7	108.7	173.2
	Array Exp.	32.7	39.4	40.7	44.9
	PRT	18.9	34.7	55.1	81.9
	SPRT	**17.5**	**23.1**	**24.4**	**25.4**
1M	DWA–LIP	6.0	7.9	9.4	19.6
	Array Exp.	14.3	16.3	18.2	20.7
	PRT	5.0	13.6	23.1	35.8
	SPRT	**4.8**	**5.9**	**7.1**	**7.7**
0.5M	DWA–LIP	3.6	6.2	10.5	14.9
	Array Exp.	12.1	13.3	16.7	18.2
	PRT	3.3	13.5	20.6	25.3
	SPRT	**3.0**	**4.0**	**4.7**	**5.0**

Tabla 4.9: Tiempo de ejecución con 10 procesadores para matrices sintéticas (en *ms*) para la reducción irregular.

y 4.12 muestran el coste asociado al inspector de la estrategia DWA-LIP cuando se realizan 4 particiones. Dado que el algoritmo `SPRT` utiliza la información del IARD con el fin de determinar las regiones compartidas y exclusivas del patrón de acceso, hemos dividido los resultados en dos contribuciones. La primera de ellas refleja el coste de la rutina `PT`, mientras que en la segunda se representa el coste del resto del proceso de reordenamiento. Nótese que el tiempo de ejecución de la primera etapa es despreciable frente al de la segunda, y que el coste de esta no presenta una dependencia lineal con el número de particiones. Nótese además que esta dependencia es más acusada cuando la anchura de la banda es elevada (como es el caso de las matrices *3dtube* y *bcsstk29*). El algoritmo `SPRT-IP`

N_x	Estrategia	*spmxv*			
		λ=1k	λ=10k	λ=25k	λ=50k
10M	Array Exp.	**103.5**	126.3	124.3	134.3
	PRT	107.0	142.6	170.6	210.1
	SPRT	105.1	**117.2**	**122.7**	**131.3**
5M	Array Exp.	57.8	61.0	67.0	72.6
	PRT	46.6	80.0	107.6	142.8
	SPRT	**44.4**	**52.1**	**55.9**	**66.1**
1M	Array Exp.	20.1	22.0	24.2	23.1
	PRT	11.4	22.5	45.6	68.5
	SPRT	**10.5**	**11.2**	**14.2**	**15.9**
0.5M	Array Exp.	15.5	16.5	18.1	21.6
	PRT	7.5	18.8	31.6	47.4
	SPRT	**7.0**	**8.4**	**9.8**	**8.7**

Tabla 4.10: Tiempo de ejecución con 10 procesadores para matrices sintéticas (en *ms*) para el producto matriz dispersa vector.

N_x	Estrategia	*csrcsc2*			
		λ=1k	λ=10k	λ=25k	λ=50k
10M	PRT	334.9	528.7	585.4	668.3
	SPRT	**332.7**	**485.8**	**532.1**	**645.6**
5M	PRT	**132.9**	**257.2**	299.4	345.9
	SPRT	133.4	227.2	**243.1**	**255.3**
1M	PRT	**20.6**	52.4	68.1	98.5
	SPRT	21.9	**37.2**	**40.1**	**35.1**
0.5M	PRT	8.0	22.6	37.1	57.3
	SPRT	**7.7**	**12.4**	**19.8**	**17.5**

Tabla 4.11: Tiempo de ejecución con 10 procesadores para matrices sintéticas (en *ms*) para la transposición de una matriz dispersa.

obtiene resultados competitivos cuando la anchura de la banda y el número de particiones es reducido. Cuando alguno de estos parámetros aumenta, su rendimiento experimenta un fuerte retroceso. Por este motivo no hemos incluido los tiempos de ejecución de este algoritmo para más de cuatro particiones. Se puede apreciar cómo el tiempo de ejecución de la propuesta DWA-LIP es superior a la del algoritmo `SPRT`. Finalmente, hay que destacar que la técnica *array expansion* no está incluida en esta tabla porque no utiliza inspector.

Estrategia	N_P	Matriz							
		3dtube		*af23560*		*bcsstk14*		*bcsstk17*	
DWA–LIP⋆	4	748		110		14		94	
SPRT	2	0.03	328	0.03	44	0.03	5	0.03	40
SPRT-IP		0.03	1.5k	0.03	22	0.03	13	0.03	43
SPRT-no-IARD		169	325	22	45	2	4	23	39
SPRT	4	0.04	366	0.04	43	0.02	4	0.04	41
SPRT-IP		0.04	3.3k	0.04	42	0.02	34	0.04	110
SPRT-no-IARD		398	349	45	43	5	3	45	38
SPRT	8	0.03	448	0.03	46	0.03	5	0.04	46
SPRT-no-IARD		790	413	86	46	11	5	82	43
SPRT	100	0.13	2.0k	0.16	96	0.17	34	0.20	163
SPRT-no-IARD		9.3k	1.6k	977	95	127	30	889	116

⋆Version secuencial

Tabla 4.12: Tiempo de ejecución de los inspectores DWA-LIP y `SPRT` (en *ms*).

Estrategia	N_P	Matriz							
		bcsstk29		*nasasrb*		*s3dkq4m2*		*struct3*	
DWA–LIP⋆	4	140		620		1116		269	
SPRT	2	0.03	55	0.04	261	0.04	456	0.03	111
SPRT-IP		0.03	128	0.04	64	0.04	277	0.03	141
SPRT-no-IARD		28	53	144	264	255	464	68	108
SPRT	4	0.03	64	0.04	263	0.03	464	0.02	112
SPRT-IP		0.03	312	0.04	201	0.03	556	0.02	187
SPRT-no-IARD		58	60	314	258	591	468	140	108
SPRT	8	0.03	81	0.04	274	0.03	483	0.03	116
SPRT-no-IARD		111	74	603	267	1151	474	263	115
SPRT	100	0.18	359	0.18	489	0.13	802	0.14	207
SPRT-no-IARD		1.3k	287	7.0k	420	13.3k	745	3.0k	204

⋆Version secuencial

Tabla 4.13: Tiempo de ejecución de los inspectores DWA-LIP y `SPRT` (en *ms*).

El uso de la caracterización IARD reduce el número de entradas del vector de indirección que deben ser analizadas por el algoritmo SPRT. Sin embargo, presenta como desventaja una imprecisión en el análisis debido al uso de la envolvente. Esta imprecisión surge por el hecho que la envolvente no se adapta completamente al patrón de acceso, por lo que el tamaño de las regiones compartidas puede ser superior al tamaño real. Este hecho origina un aumento del tiempo de ejecución de la rutina SPRT, dado que algunas entradas son analizadas de forma innecesaria.

Hemos comparado la eficiencia de nuestra propuesta con la de un algoritmo de reordenamiento que no hace uso de la información del IARD. Este algoritmo consiste en una modificación del algoritmo *interseca* en el que los *slices* asociados a cada partición se obtienen únicamente en base a la información contenida en la indirección, sin operar con las curvas envolventes. Denominamos a este algoritmo como *interseca*2 y su pseudocódigo se muestra en la Figura 4.33.

El algoritmo recorre ordenadamente las columnas del patrón de acceso. Para cada una de las particiones $\mathcal{A}_i$ se almacena el valor del primer y último *slice* que contiene un acceso sobre el bloque considerado. Dado que la estructura del patrón de acceso es desconocida, es necesario recorrer, para cada partición, todas las entradas de la indirección. La complejidad de esta propuesta está acotada tal y como se muestra en la siguiente expresión. El valor exacto de dicha complejidad depende de la estructura del patrón de acceso, y muy especialmente, de la anchura de la banda.

$$(N_{pt} - 1)N_x \leq \mathcal{O}_{interseca2} \leq N_S(N_{pt} - 1)N_x \tag{4.41}$$

El empleo de esta propuesta conduce a dos comportamientos distintos. Por una parte, la complejidad de la rutina *interseca*2 es superior a la de *interseca*, aumentando el tiempo de procesamiento. Por otra parte, el resto de la rutina SPRT obtiene una información más precisa acerca del patrón de acceso, consiguiendo reducir su coste de análisis. Hemos evaluado ambos efectos midiendo de forma separada el tiempo de ejecución de cada una de estas rutinas. Los resultados obtenidos con esta versión, denominada SPRT-no-IARD, se muestran en las tablas 4.13 y 4.12. Analizando los valores de la tabla se puede destacar el fuerte incremento del tiempo de ejecución experimentado por la rutina *interseca*2. También se puede apreciar la ligera disminución del coste del resto de la rutina SPRT-no-IARD. Estos resultados prueban la alta precisión conseguida con la caracterización IARD y el ahorro conseguido con su uso en el análisis de la indirección. Es importante destacar que ambos algoritmos (rutina SPRT y SPRT-no-IARD) originan el mismo vector de indirección reordenado. Es decir, a pesar de trabajar sobre un conjunto de datos diferente, ambas rutinas aseguran la obtención del resultado correcto.

```
Algoritmo interseca2
entrada
        x: vector de indirección
        N_pt: número de particiones
salida
      {s^1, s^2, s^3, s^4}_i   i = 1, N_pt: conjunto de slices asociados a cada partición
inicio del algoritmo
         DO i = 1, N_pt
           DO l = s_i^1[i],N_S
             DO j = ρ_ini[s], ρ_ini[s+1] − 1
               IF(x[j] ∈ A_i  &   s_i^1 = ∅)      s_i^1 = l
               IF(x[j] ∈ A_i)                     s_i^4 = l
             END DO
           END DO
           IF(i > 1)
             s_i^2 = s_{i−1}^4
             s_{i−1}^3 = s_i^1
           END IF
         END DO
fin del algoritmo
```

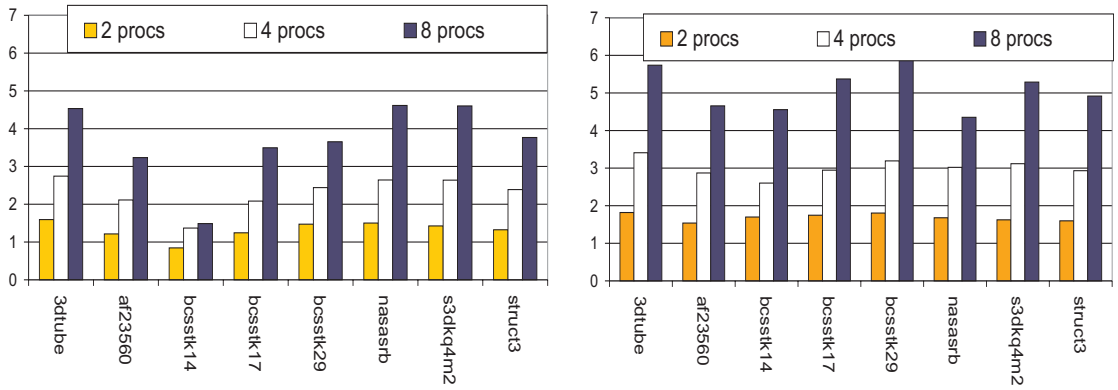
Figura 4.33: Algoritmo *interseca*2.

La Figura 4.34 muestra las aceleraciones obtenidas con el algoritmo `SPRT-PAR` para 2, 4 y 8 procesadores cuando se realizan 8 y 100 particiones.

Respecto al coste de almacenamiento asociado al inspector, es necesario distinguir si abordamos la paralelización de códigos con una única indirección o con una matriz dispersa. El segundo caso supone la situación más costosa en términos de requisitos de almacenamiento. En esta situación el coste de nuestra propuesta viene dada por:

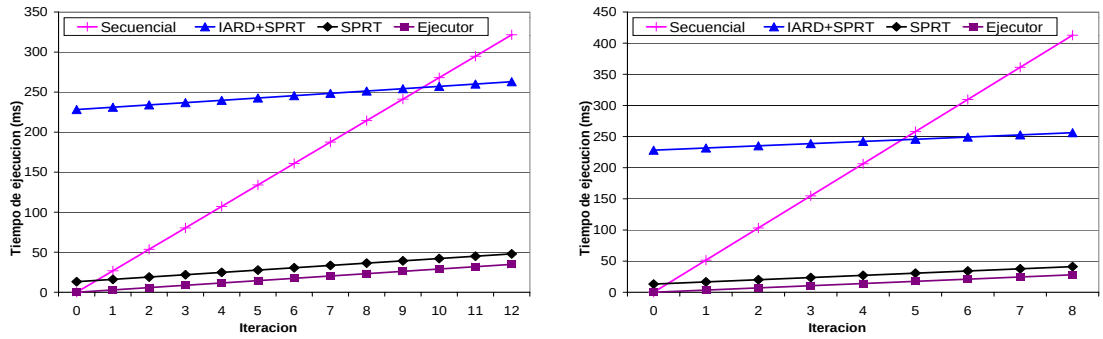
$$\Delta M_{\texttt{SPRT}}^{ejecutor} = N_{col} + 2N_{pt} + 1 \tag{4.42}$$

Nótese que, de acuerdo con la Expresión 4.35, el valor de N_{col} es dependiente tanto del número de procesadores como de la anchura media del patrón de acceso. En el conjunto de datos que hemos considerado se verifica que $N_{col} \ll N_x$, por lo que el coste asociado al ejecutor DWA-LIP (Expresión 4.6) es superior al de nuestra propuesta. El coste asociado al ejecutor de la técnica *array expansion* se muestra en la Expresión 4.1. Si la matriz es ban-



(a) (b)

Figura 4.34: Aceleraciones obtenidas para el algoritmo SPRT paralelo: (a) 8 particiones, (b) 100 particiones.



(a) (b)

Figura 4.35: Tiempo de ejecución (en *ms*) para la matriz *bcsstk17*: (a) rutina *spmxv*, (b) rutina *csrcsc2*.

deada, entonces se verifica que $N_{col} < (N_a N_p)$ por lo que $\Delta M^{ejecutor}_{\text{SPRT}} < \Delta M^{ejecutor}_{array\ expansion}$.

Un último estudio que hemos realizado se centra en los costes asociados a las diferentes etapas de nuestra propuesta. En muchas aplicaciones, el lazo candidato a ser paralelizado es ejecutado múltiples veces sin que el patrón de acceso a memoria sufra modificaciones. Para estos casos, la información ofrecida por la caracterización IARD puede ser reusada.

La Figura 4.35 muestra, para un número de iteraciones variable, y para 8 procesadores, el tiempo de ejecución acumulado de los siguientes códigos.

- *Secuencial:* esta medida se corresponde al tiempo de ejecución de la rutina secuencial. Este es un valor constante a lo largo de las iteraciones, por lo que el tiempo acumulado para la i-ésima iteración es $i * t_{secuencial}$.
- *IARD-SPRT:* representa el tiempo de ejecución de nuestro ejecutor paralelo, sumado al tiempo de ejecución de la rutina de cálculo de la representación IARD y al coste de la rutina de inspección (algoritmo SPRT).
- *SPRT:* representa el tiempo de ejecución de nuestro ejecutor paralelo y del algoritmo inspector SPRT.
- *Ejecutor:* se corresponde al tiempo de ejecución del ejecutor paralelo.

En todos los casos estamos asumiendo un reuso de la información en la caracterización IARD y de los resultados obtenidos por el inspector. Observando ambas figuras se puede apreciar como, a pesar de que el coste de la caracterización IARD es significativo, este se compensado en unas pocas iteraciones por medio del reuso de la información del inspector. Este efecto se puede apreciar con un mayor detalle en la Tabla 4.14, la cual muestra el **umbral de iteraciones** para un ejecutor paralelo con 8 procesadores. Este umbral se define como el mínimo número de iteraciones en las que el tiempo acumulado del ejecutor paralelo sumado al tiempo del inspector supera el tiempo acumulado del código secuencial. Se puede apreciar cómo el coste asociado al algoritmo SPRT es superado en unas pocas iteraciones. Este número de iteraciones aumenta cuando se considera el coste asociado a la caracterización IARD. Sin embargo, el nivel de reuso requerido no es muy alto (de unas pocas decenas de iteraciones), lo que hace factible nuestra propuesta.

Matriz	*reducción irregular*		*smpxv*		*csrcsc2*	
	IARD+SPRT	SPRT	IARD+SPRT	SPRT	IARD+SPRT	SPRT
3dtube	13	1	6	1	4	1
af235	44	2	14	1	9	1
bcsstk14	37	4	19	2	17	2
bcsstk17	28	2	10	1	5	1
bcsstk29	26	2	7	1	5	1
nasasrb	12	1	6	1	4	1
s3dkq4m2	8	1	4	1	3	1
struct3	14	2	5	1	3	1

Tabla 4.14: Umbral (en iteraciones) para sobrepasar el rendimiento del código secuencial.

La estrategia SPRT no sólo se puede aplicar a la paralelización de códigos irregulares, sino también a la mejora de la localidad de códigos secuenciales. La siguiente sección aborda la adecuación de esta estrategia en este nuevo contexto.

4.3 Mejora en la localidad de códigos secuenciales

El grado de explotación de la jerarquía de memoria y de la localidad en los accesos tiene un fuerte impacto en el rendimiento de un programa. Situándonos en el contexto de nuestra línea de investigación, en esta sección nos vamos a centrar en la problemática de la mejora de la localidad en códigos irregulares, particularizando nuestro estudio a códigos secuenciales.

Dado un lazo irregular, existen, fundamentalmente, dos alternativas para explotar la jerarquía de memoria del sistema. La primera de ellas consiste en el cambio del orden en el que las iteraciones son ejecutadas. Con esto se busca ejecutar de una forma próxima en el tiempo aquellas iteraciones que accedan a las mismas posiciones de memoria aumentando, de este modo, la localidad temporal en los accesos. Esta opción puede tener como inconveniente una baja explotación de la localidad espacial. Por ejemplo, considerando cualquiera de los códigos irregulares mostrados en la Figura 4.7, un aumento en la localidad temporal implica ejecutar de forma contigua aquellas iteraciones que accedan sobre las mismas entradas de la matriz a. Sin embargo, en un caso genérico este reordenamiento implicaría recorrer entradas no contiguas del vector x, dando lugar a una baja localidad en los accesos.

La segunda alternativa resuelve este problema mediante el reordenamiento de los datos (en nuestro caso, del vector de indirección). De este modo, los elementos son dispuestos de acuerdo con el nuevo orden de las iteraciones, preservando además la localidad espacial. Esta idea sirvió de motivación para el desarrollo del algoritmo SPRT, el cual está originalmente diseñado para la paralelización de lazos irregulares. En esta sección vamos a abordar la aplicación del mismo algoritmo para mejorar la localidad en códigos secuenciales.

4.3.1 Adaptación del algoritmo SPRT a códigos secuenciales

La adaptación de nuestra propuesta no implica la realización modificaciones en la estructura del algoritmo SPRT, sino que únicamente hay que considerar un tamaño de partición menor. Por este motivo hay que modificar el conjunto de particionamiento

(conjunto $\mathcal{LIM}$) de modo que en vez de realizar tantas particiones como número de procesadores, emplee un tamaño de partición constante y especificado por el usuario. Denominamos L al tamaño de partición. Trivialmente, este parámetro se relaciona con el número de particiones mediante la expresión $N_{pt} = \lceil \frac{N_a}{L} \rceil$.

Dado que nuestro objetivo es explotar todas las características del espacio de memoria, y dentro de esta la memoria cache primaria supone el nivel de jerarquía más reducido (exceptuando los registros del procesador), en este trabajo proponemos como tamaño de las particiones múltiplos enteros del número de palabras que se almacenan en una línea de cache primaria.

4.3.2 Análisis de eficiencia

El rendimiento de nuestra propuesta va a depender de dos factores. Por una parte tenemos el grado de eficiencia del código optimizado, y por otra tenemos el coste computacional del inspector. En las siguientes secciones realizamos una descripción del entorno de trabajo que hemos empleado, junto con una evaluación de estos dos factores.

Entorno de evaluación

Con el fin de evaluar la eficiencia del algoritmo `SPRT` sobre códigos secuenciales, hemos utilizado dos arquitecturas diferentes. Estas son, una Silicon Graphics Origin 200 y una SUN Enterprise 250. Las principales características de estas arquitecturas se describen en el Apéndice A.

Como conjunto de programas de evaluación hemos utilizado dos rutinas irregulares: la reducción irregular y la rutina de transposición de una matriz dispersa. El código secuencial y paralelo de cada una de ellas se muestra en las figuras 4.30 y 4.32. El código secuencial optimizado del ejecutor coincide en su estructura con el código paralelizado mediante la estrategia IARD-SPRT. La única diferencia existente consiste en el reemplazo del lazo DOALL externo por el de un DO secuencial. Los códigos están escritos en Fortran 77 y compilados con MIPSpro `f77` v7.2.1 en el caso de la Origin y `f77` WorkShop 4.2 en el caso de la Enterprise. En ambos casos se ha utilizado la opción de compilación "`-aling128`" con el fin de evitar falsa comparición de líneas cache.

Como vectores de indirección hemos utilizado la familia de matrices sintéticas mostrada en la Figura 4.20. Específicamente, hemos restringido nuestro análisis a aquellas que tienen una anchura de banda λ comprendida entre 100 y 50k entradas, y un número de entradas

no nulas N_x de 500k, 1000k y 5000k. El número de columnas es en todos los casos de 100k y coincide con el número de *slices*.

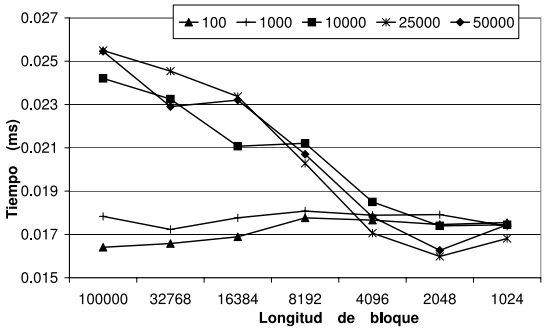
Eficiencia del ejecutor

Las figuras 4.36 y 4.37 representan el tiempo de ejecución obtenido con el código optimizado respeto a distintos tamaños de partición. Esta última magnitud aparece denotada en las figuras como *longitud de bloque*. En cada una de las gráficas, cada valor de λ está representado como una serie distinta. En el esquema que hemos utilizado, un tamaño de partición igual a 100k corresponde al programa secuencial de referencia. Este consiste en el uso del código original (figuras 4.30(a) y 4.32(a)) junto con el empleo de la indirección no reordenada. Comparando ambas arquitecturas podemos apreciar que el sistema Enterprise obtiene una mejor eficiencia en todos los casos, siendo esta mucho más acentuada para la rutina *csrcsc*2 cuando el ancho de banda es elevado y no se aplica un particionamiento.

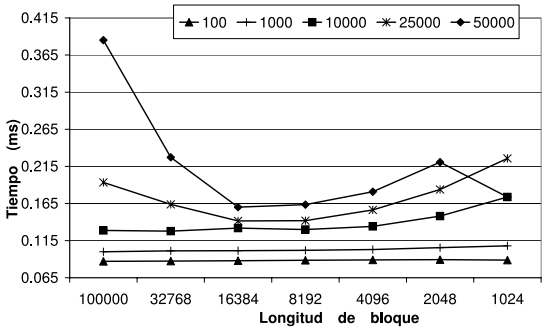
En la figura se puede apreciar que para ciertos tamaños de particionamiento existe una reducción significativa en el tiempo de ejecución del lazo de prueba. Esta reducción es más importante para matrices con gran anchura en la banda y con mayor número de entradas. Cabe destacar la fuerte mejora obtenida para la rutina *csrcsc*2 para la matriz con $N_x = 5M$. Mediante el empleo de nuestra propuesta se consigue un rendimiento similar para cada tamaño de matriz, con independencia de la anchura de la banda. En estos casos, los tiempos se ejecución son similares en ambas arquitecturas.

Con el fin de identificar las causas de esta mejora en el rendimiento del programa, hemos utilizado los contadores *hardware* existentes en el procesador MIPS R10000 [153] para medir distintos eventos durante la ejecución. Concretamente, hemos medido el número de fallos de TLB y el reuso de líneas en la cache primaria y secundaria.

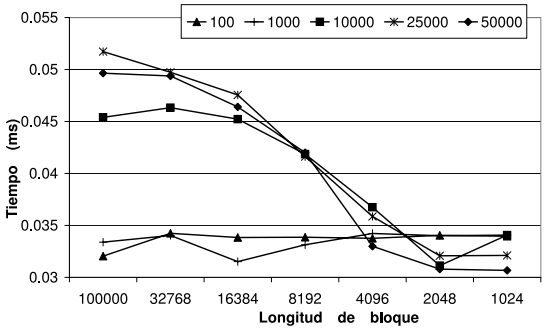
Experimentalmente, no hemos obtenido ninguna reducción significativa en el número de fallos de TLB para la rutina de reducción irregular. Esto es debido al reducido volumen de datos que utiliza, el cual puede ser íntegramente mantenido en la TLB. Esta situación no se da para la rutina *csrcsc*2, la cual utiliza, además del vector de indirección, los vectores *val* y *col*. La rutina *csrcsc*2 emplea un volumen de datos mucho mayor que el utilizado en la reducción irregular. De modo más concreto, y considerando únicamente los vectores de N_x entradas, la rutina *csrcsc*2 requiere hasta cuatro veces más espacio de almacenamiento que la reducción irregular, dado que tiene que almacenar los vectores *row* y *val* junto con las copias asociadas a su transposición. En la Figura 4.38 se muestra el número de fallos TLB para esta rutina. Se puede apreciar que para valores de λ elevados, el número de fallos experimenta una fuerte reducción.



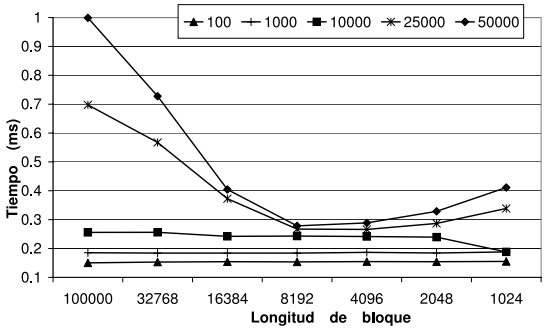
(a) rutina reducción irregular, $N_x = 500k$



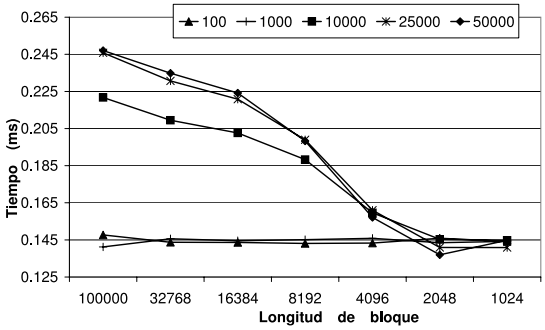
(b) rutina $csrcsc2, N_x = 500k$



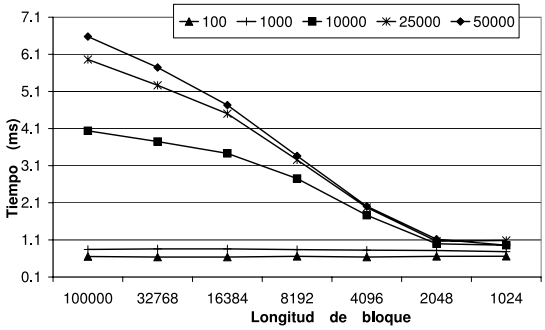
(c) rutina reducción irregular, $N_x = 1M$



(d) rutina $csrcsc2, N_x = 1M$

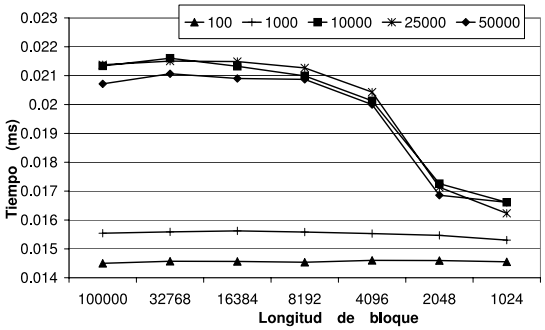


(e) rutina reducción irregular, $N_x = 5M$

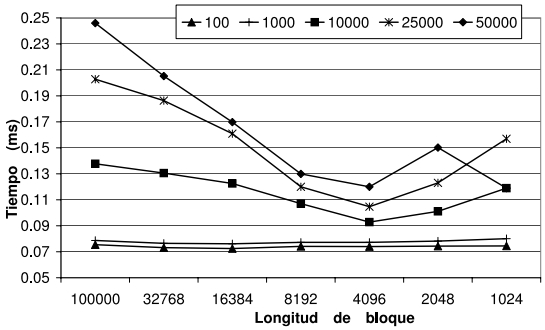


(f) rutina $csrcsc2, N_x = 5M$

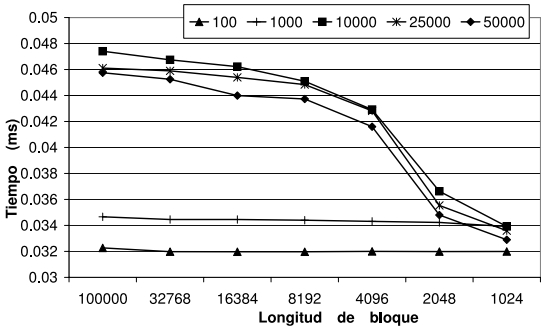
Figura 4.36: Tiempos de ejecución para el MIPS R10000.



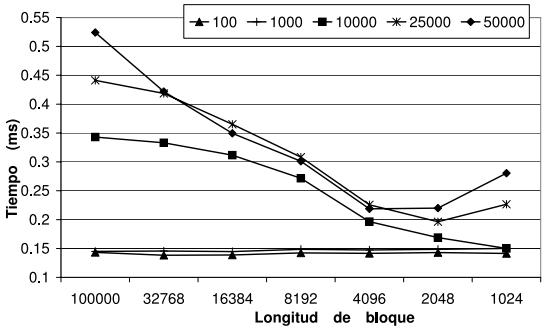
(a) rutina reducción irregular, $N_x = 500k$



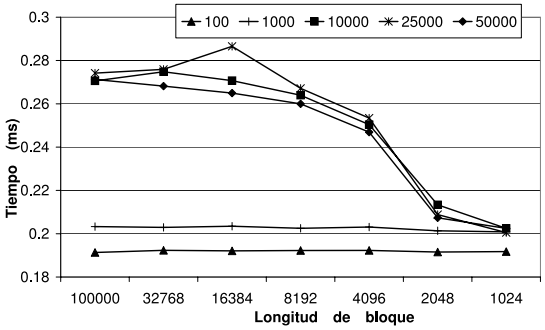
(b) rutina $csrcsc2, N_x = 500k$



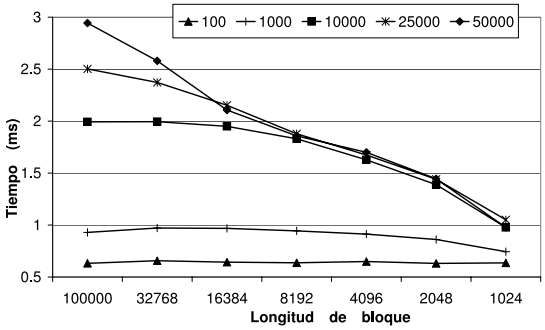
(c) rutina reducción irregular, $N_x = 1M$



(d) rutina $csrcsc2, N_x = 1M$

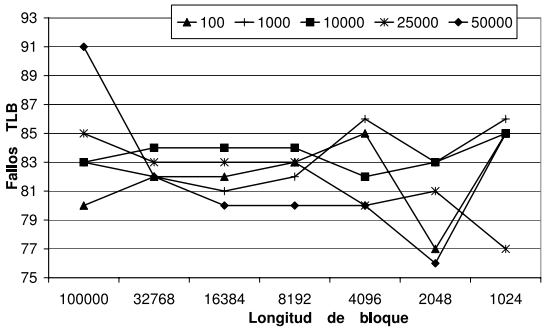


(e) rutina reducción irregular, $N_x = 5M$

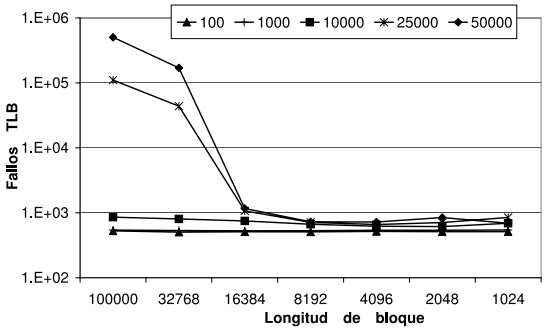


(f) rutina $csrcsc2, N_x = 5M$

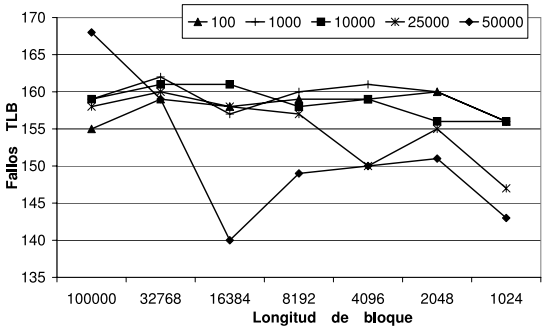
Figura 4.37: Tiempos de ejecución para el UltraSPARC II.



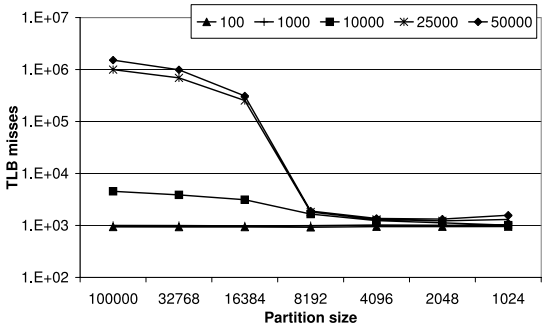
(a) rutina reducción irregular, $N_x = 500k$



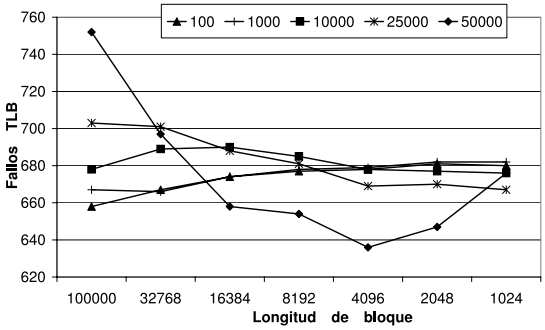
(b) rutina $csrcsc2, N_x = 500k$



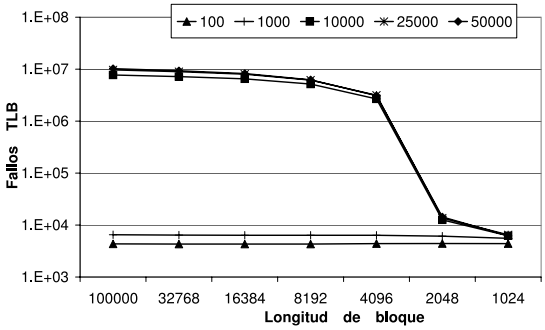
(c) rutina reducción irregular, $N_x = 1M$



(d) rutina $csrcsc2, N_x = 1M$



(e) rutina reducción irregular, $N_x = 5M$



(f) rutina $csrcsc2, N_x = 5M$

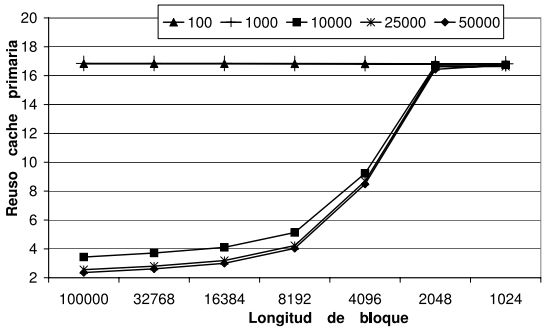
Figura 4.38: Fallos TLB para el MIPS R10000.

Considerando ahora la eficiencia obtenida con nuestra propuesta a nivel de memoria cache, la Figura 4.39 muestra el reuso de líneas de cache primaria para la rutina de reducción irregular. Se puede apreciar que, en los casos en los que la anchura de la banda es superior a $\lambda = 1000$, existe un fuerte incremento en el reuso cuando el tamaño de partición disminuye. Este incremento en el reuso ocurre para todo el rango de N_x, siendo más significativo con las matrices de mayor número de entradas. Nótese además que para un tamaño de bloque de 2048 todas las matrices alcanzan el mismo nivel de reuso, el cual coincide con aquellas que tienen una anchura de banda menor o igual de $\lambda = 1000$.

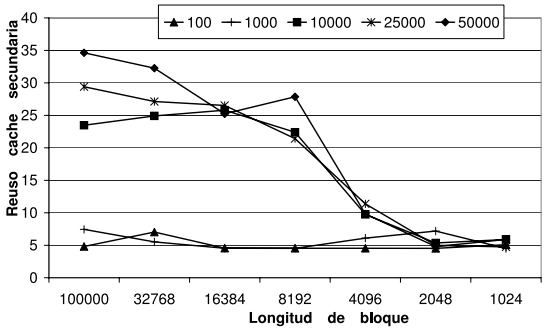
Cuando evaluamos el nivel de reuso existente en la cache secundaria, el resultado obtenido es el complementario al caso anterior. La Figura 4.39 muestra el grado de reuso para esta memoria y para el mismo código de prueba. Apreciamos una disminución del mismo cuando el tamaño de la partición disminuye. De forma análoga al caso anterior, las matrices con λ inferior o igual a 1000 mantienen un nivel constante de reuso. Sin embargo, en este caso el reuso tiene unos niveles mínimos de entorno a 5 en todos los casos.

La explicación de este fenómeno se haya en las figuras 4.22(a) y 4.22(b). Cuando la anchura de la banda es reducida, los accesos al vector a se realizan sobre un conjunto pequeño de entradas (denotado como a_1 en la figura). Adicionalmente, y dada la estructura bandeada del patrón de acceso, el procesado de *slices* consecutivos implica accesos repetidos sobre casi el mismo intervalo de entradas de a. De este modo, para valores de λ reducidos, se alcanza un alto grado de localidad espacial y temporal en los accesos a a. Cuando esto ocurre, existe un alto grado de reuso en cache primaria. Por otra parte, cuando λ toma valores suficientemente altos, el conjunto de entradas de a accedidas dentro del mismo *slice* aumenta. Si el volumen de datos accedidos supera la capacidad de almacenamiento de la memoria cache primaria, esta memoria es incapaz de mantener toda la información, lo cual conlleva una pérdida de localidad temporal. En estos casos, es en la memoria cache secundaria en donde se realiza el reuso. En el sistema Origin 200, el tamaño de la memoria cache secundaria junto con el mecanismo de precarga es suficiente para explotar a este nivel toda la localidad de los accesos, por lo que este fenómeno tiene poca repercusión sobre los accesos a memoria principal.

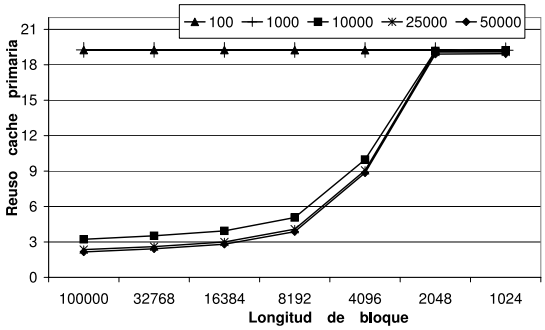
Teniendo en cuenta la estructura del lazo irregular (Figura 4.30) la localidad en los accesos únicamente se puede deber al reuso de las líneas cache que almacenan los valores de a. En el resto de los vectores ($inicio$, col_{fin} y x_{fin}) cada entrada se lee una única vez, y siempre se accede sobre elementos consecutivos. De este modo, en el mejor de los casos, cada línea que almacena entradas de uno de estos vectores se reusará tantas veces como número de entradas contenga. Adicionalmente, el nivel de reuso asociado a estos tres vectores no depende del tamaño de la partición, ya que el número y forma de acceso



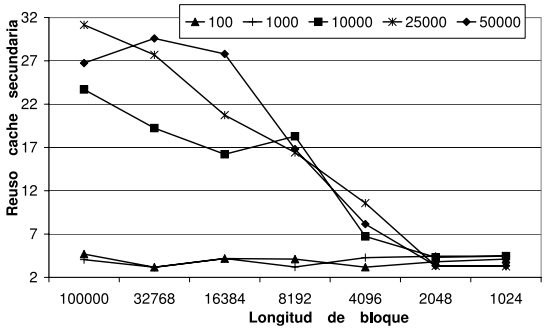
(a) cache L1, $N_x = 500k$



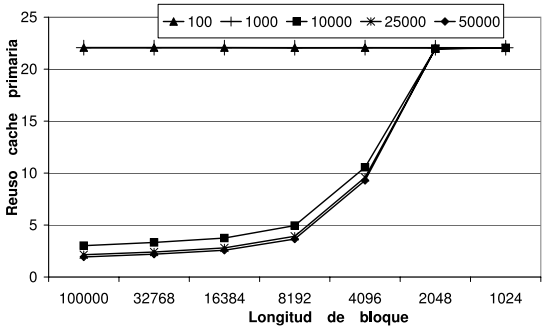
(b) cache L2, $N_x = 500k$



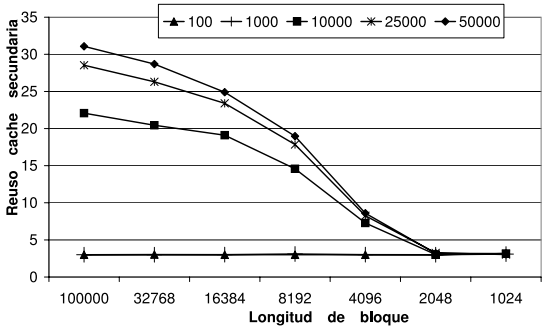
(c) cache L1, $N_x = 1M$



(d) cache L2, $N_x = 1M$



(e) cache L2, $N_x = 5M$



(f) cache L2, $N_x = 5M$

Figura 4.39: Nivel de reuso para la reducción irregular en el MIPS R10000.

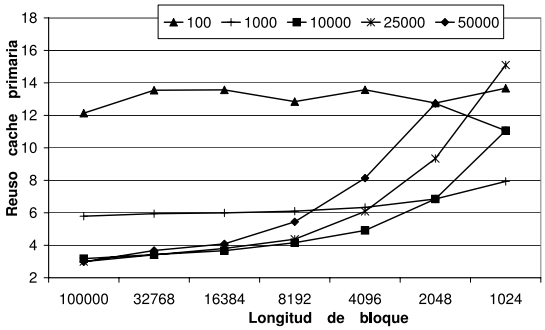
no guarda dependencia con este parámetro.

Mediante nuestra propuesta, para una matriz dada con una valor genérico de λ, obtenemos, mediante el algoritmo `SPRT`, un patrón de acceso que tiene las mismas propiedades que las de una matriz de ancho de banda λ'. Siendo λ' el tamaño de partición que hemos aplicado sobre a. De este modo, y con un valor de λ' adecuado, obtenemos niveles de reuso próximos a los alcanzados con anchura de banda reducida y, consiguientemente, una importante mejora en la eficiencia del programa secuencial.

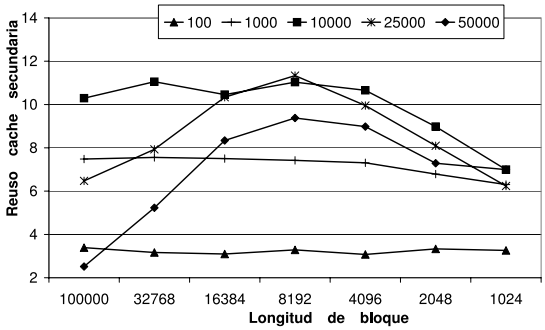
En el caso de la rutina *csrcsc*2, su comportamiento a nivel de memoria cache experimenta ciertos cambios respecto al caso anterior. La Figura 4.40 representa el nivel de reuso en memoria cache primaria. Del mismo modo que con la reducción irregular, cuando la anchura de banda es reducida existe un alto grado de reuso. Igualmente, cuando este nivel de reuso no es elevado, podemos aumentarlo mediante la aplicación de un particionamiento suficientemente reducido. La primera diferencia surge en el momento de evaluar la magnitud del aumento de reuso con el grado de particionamiento. En este caso, no se alcanza un nivel límite de reuso (como sucede para la reducción irregular) y esta cantidad depende en gran medida del número de entradas de la matriz.

La segunda diferencia aparece al considerar el grado de reuso en cache secundaria (Figura 4.40). Si la anchura de banda es suficientemente reducida, el nivel de reuso no experimenta grandes cambios, dado que este se realiza en cache primaria. Sin embargo, cuando λ es suficientemente elevado, el comportamiento resulta totalmente diferente al del caso anterior: al reducir el tamaño de partición, el nivel de reuso aumenta tanto en la memoria cache secundaria como en la primaria. Este hecho explica la fuerte reducción en los tiempos de ejecución de esta rutina.

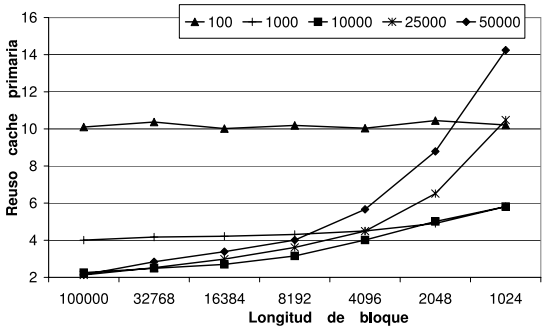
Este comportamiento se debe al aumento del volumen de datos accedido por el algoritmo *csrcsc*2 respecto a la reducción irregular. Considerando el código paralelo de la Figura 4.32, la mejora en la localidad de los accesos se realiza en el vector col_{fin} en operaciones de lectura y col' en operaciones de escritura. Por otra parte, los accesos sobre los vectores val_{fin}, v', row_{fin} y row' se realizan siempre sobre entradas distintas. De este modo, tenemos un esquema de acceso similar al de la rutina de reducción irregular, pero con un volumen de datos doble respecto a accesos con alto grado de reuso, y cuádruple respecto a accesos con un bajo grado de reuso. Este aumento del volumen de datos hace que un incremento en la localidad de los accesos implique una mejora en la localidad de ambos niveles de memoria cache. De hecho, mediante el empleo de un tamaño de partición reducido, se consigue también un aumento en la localidad en los accesos a val_{fin}, v', row_{fin} y row'. Esto es debido a que, cuando el tamaño de partición disminuye, los



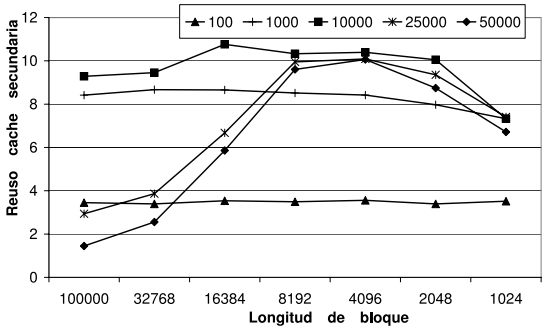
(a) cache L1, $N_x = 500k$



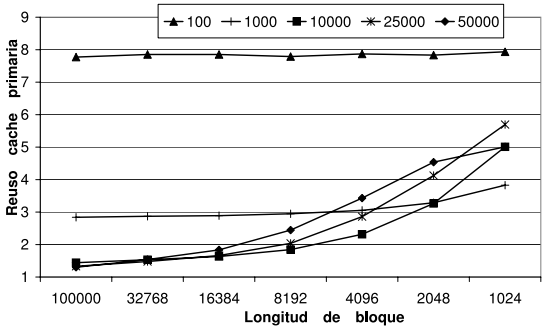
(b) cache L2, $N_x = 500k$



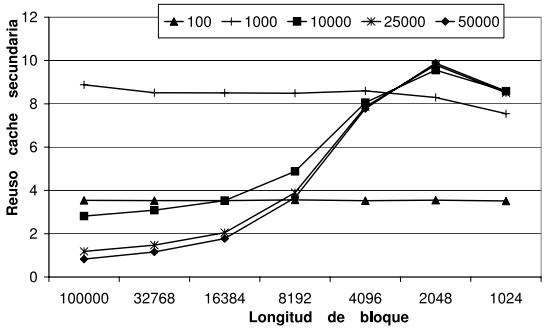
(c) cache L1, $N_x = 1M$



(d) cache L2, $N_x = 1M$



(e) cache L2, $N_x = 5M$



(f) cache L2, $N_x = 5M$

Figura 4.40: Nivel de reuso para la rutina *csrcsc*2 en el MIPS R10000.

valores accedidos a col_{fin} tiene una mayor localidad espacial y temporal. Es decir, tiende a acceder de forma repetida sobre elementos contiguos de este vector. Como los valores almacenados en este vector son monótonos crecientes, esta mayor localidad en col_{fin} implica obtener valores de la variable *next* más próximos entre sí, lo cual implica a su vez un aumento en la localidad espacial en los accesos sobre val_{fin}, v', row_{fin} y row'.

Eficiencia del inspector

La Tabla 4.15 ilustra los tiempo de procesamiento de la representación IARD y de nuestro inspector para dos tamaños representativos de particionamiento. Concretamente, elegimos unos tamaños de bloque de 2048 y 4096 entradas. La plataforma empleada es una Silicon Graphics Origin 200. En esta tabla, el coste del inspector aparece desglosado en el coste de la caracterización IARD, la rutina *interseca* (primera entrada de cada celda) y el resto del algoritmo SPRT (segunda entrada de cada celda).

Nuevamente, destacamos el hecho de que la caracterización IARD supone la parte más costosa de nuestra propuesta, aunque este coste puede ser ocultado a través de su reuso. Adicionalmente, el empleo de la caracterización IARD hace que la etapa de inspección tenga un menor coste. Con el fin de evaluar este efecto, para los tamaños de partición considerada, hemos vuelto a utilizar el algoritmo modificado SPRT que emplea la rutina *interseca*2. Los resultados obtenidos para una partición de 2048 entradas se muestran en la última fila de la Tabla 4.15. Nuevamente podemos apreciar que el coste de esta última

N_x	Estrategia	Ancho de banda									
		λ =100		λ =1000		λ =10000		λ =25000		λ =50000	
0.5M	IARD	425		539		983		2460		3328	
	$\|\mathcal{A}\| = 2048$	0.085	69	0.098	89	0.127	251	0.219	492	0.214	972
	$\|\mathcal{A}\| = 4096$	0.063	68	0.071	78	0.089	167	0.140	291	0.143	557
1M	IARD	518		593		1158		1986		2610	
	$\|\mathcal{A}\| = 2048$	0.097	115	0.098	144	0.097	393	0.183	875	0.170	1555
	$\|\mathcal{A}\| = 4096$	0.071	113	0.071	127	0.075	263	0.118	511	0.114	866
5M	IARD	907		950		1773		3176		4751	
	$\|\mathcal{A}\| = 2048$	0.098	482	0.102	574	0.106	1893	0.106	3634	0.100	6143
	$\|\mathcal{A}\| = 4096$	0.077	476	0.078	512	0.075	1231	0.080	3482	0.079	2115
0.5M	SPRT-no-IARD 2048	937	67	953	85	1062	227	1231	484	1479	960
1M		1566	112	1593	141	1775	384	2034	861	2457	1565
5M		6285	482	6306	566	7101	1870	8341	3578	10060	5958

Tabla 4.15: Tiempo de ejecución del inspector para distintos particionamientos (ms).

rutina es muy grande en comparación con la rutina *interseca*, basada en la información del IARD. También es necesario destacar que la reducción en el tiempo de ejecución del resto de la rutina `SPRT` (debida a una determinación más exacta de las regiones compartidas) es muy pequeño, lo que prueba la precisión obtenida mediante nuestra caracterización.

4.4 Mejora del balanceo de la carga

Como punto de partida para abordar este problema, nos centraremos en el desarrollo de una estrategia de balanceo de la carga aplicada a un lazo irregular como el mostrado en la Figura 4.30(a). Aunque en esta figura únicamente se representa una reducción irregular, nuestra propuesta es también aplicable a lazos en los que el cuerpo consta de un acceso genérico, que puede ser tanto de lectura, como escritura o acumulación.

Tal y como se comentó en la Sección 4.1, la paralelización de este tipo de lazos mediante la regla del propietario se puede realizar empleando un condicional que verifique la región sobre la que se realiza el acceso. La Figura 4.41(a) muestra la reducción irregular paralelizada mediante esta estrategia. Hemos aplicado la regla del propietario sobre los accesos de a asumiendo una distribución de las entradas mediante bloques, los cuales se denotan por $\mathcal{A}_p$. El lazo de prueba contiene además un conjunto de operaciones que suponen una carga de trabajo extra (denotado como *carga* en la figura). Vamos a asumir que estas operaciones no tienen asociadas ninguna dependencia de datos, ni tampoco tienen significativa influencia sobre la cantidad de memoria consumida.

Esta estrategia de paralelización asigna las distintas iteraciones en función de la distribución del vector a. Esta política de distribución tiene como ventaja el empleo de una rutina de inspección con un coste reducido, dado que únicamente hay que distribuir la iteraciones del lazo y el vector de indirección sobre los procesadores. Uno de sus principales inconvenientes es el mal comportamiento del código paralelo bajo patrones de acceso mal balanceados, dado que en estos casos una distribución regular de a origina una distribución desbalanceada de las computaciones. Una solución a esta problemática fue propuesta por los autores en [62]. En este trabajo se propone el empleo de algoritmos de particionamiento de grafos para distribuir la carga de trabajo en función de las características del patrón de acceso. De este modo, se aumenta la eficiencia del código paralelo, a costa de un incremento significativo del coste del inspector.

Tal y como se ha comentado con anterioridad, el coste asociado al inspector tiene una importancia crítica a la hora de establecer la viabilidad del empleo de una estrategia de paralelización. Este hecho ha motivado al desarrollo de una nueva estrategia de balanceo

```
DOALL p = 1, N_pt
   DO j = 1, N_x
      IF (x[j] ∈ A_p)
         a[x[j]] = a[x[j]] ⊗ ...
         carga()
      END IF
   END DO
END DOALL
```

(a)

```
DOALL i = 1, N_pt
      % Región compartida
        DO j = ρ_ini[s_i^1], ρ_ini[s_i^2 + 1] − 1
           IF  x[j] ∈ A_i
               a[x[j]] = ...
               { ... }
           END IF
        END DO
      % Región exclusiva
        DO j = ρ_ini[s_i^2 + 1], ρ_ini[s_{i+1}^3] − 1
           a[x[j]] = ...
           { ... }
        END DO
      % Región compartida
        DO j = ρ_ini[s_{i+1}^3], ρ_ini[s_{i+1}^4 + 1] − 1
           IF  x[j] ∈ A_i
               a[x[j]] = ...
               { ... }
           END IF
        END DO
END DOALL
```

(b)

Figura 4.41: Reducción irregular: (a) Código paralelo utilizando la regla del propietario, (b) Código paralelo aplicando la regla del propietario junto con la información del IARD.

de la carga que emplee la información almacenada en el IARD con el fin de acelerar el proceso de cálculo. Inicialmente, esta nueva estrategia fue concebida para su empleo en un código paralelizado mediante la regla del propietario. Con el propósito de seguir la misma línea de desarrollo, vamos a introducir nuestra propuesta en el marco de este contexto. Posteriormente, generalizaremos esta estrategia a técnicas de paralelización como la PRT y la SPRT.

4.4.1 Balanceo de la carga en la regla del propietario

El código paralelo mostrado en la Figura 4.41(a) puede ser optimizado si se dispone de información acerca de la estructura del patrón de accesos. Concretamente, a partir de su representación IARD, podemos desarrollar una versión paralela más eficiente. La Figura 4.41(b) muestra dicha versión, la cual introduce dos importantes mejoras respecto a la estrategia original. La primera de estas mejoras es que cada procesador no debe analizar todo el espacio de iteraciones del lazo original, sino que se debe centrar únicamente en el conjunto de iteraciones en los que pueden existir accesos sobre la partición que tiene asignada. Estas regiones son el conjunto de iteraciones pertenecientes tanto a las regiones compartidas como a la región exclusiva asociada a dicha partición. Empleando la notación introducida en la Definición 4.2.3, el conjunto de entradas analizado por cada procesador es $\mathcal{X}_{s^1}^{s^4}$, donde s^1 y s^4 son los mayores y menores *slices* asociados a la partición. Ambos valores pueden ser obtenidos mediante la función *interseca*. La segunda mejora del código optimizado consiste en la eliminación de las rutinas condicionales para la región exclusiva. De acuerdo con la Propiedad 4.2.1, todas las entradas pertenecientes a esta región acceden dentro del intervalo considerado, por lo que resulta innecesario el empleo de una rutina de comprobación. Este no es el caso de las regiones compartidas, para las cuales la rutina condicional sigue siendo necesaria.

A continuación vamos a evaluar la carga asociada a la ejecución de cada una de estas regiones. Con el fin de tener un ejemplo simplificado, vamos a asumir una distribución de la matriz a en tres bloques de igual tamaño para un patrón de acceso como el mostrado en la Figura 4.10(a). Considerando el procesador asociado a la partición central, distinguimos tres regiones con diferente carga de trabajo:

1. **Regiones compartidas I y II.** Todas las entradas del vector de indirección pertenecientes a estas regiones tienen asociado el coste de la ejecución de la rutina de comprobación más el del cuerpo del lazo. Vamos a asumir para esta región un coste temporal de procesamiento de valor α.

2. **Regiones compartidas III y IV.** Las entradas existentes en estas regiones no realizan accesos dentro del bloque considerado, sin embargo tienen que ser analizadas mediante la rutina condicional. De este modo, vamos a asignarle un coste de valor β, el cual se corresponde al coste de la rutina condicional.

3. **Región exclusiva.** En este caso, todos los accesos se realizan sin aplicar la rutina de comprobación, por lo que su coste temporal asociado, denominado γ, únicamente será el del cuerpo del lazo.

En base a esta diferenciación, el coste computacional C_p asociado a la ejecución del la p-ésima partición del lazo puede modelarse mediante la siguiente expresión.

$$C_p = (\alpha(Z_p^I + Z_p^{II}) + \beta(Z_p^{III} + Z_p^{IV}) + \gamma Z_p^{excl}) * \upsilon_p \tag{4.43}$$

Donde Z_i^j es el número de entradas del vector de indirección que acceden a la región j de la i-ésima partición. De este modo, se verifica que:

$$Z^I + Z^{II} = |\mathcal{X}_{s^1}^{s^2}| \tag{4.44}$$

$$Z^{III} + Z^{IV} = |\mathcal{X}_{s^3}^{s^4}| \tag{4.45}$$

Los parámetros α, β y γ se pueden determinar de distintas formas. Una posibilidad es obtenerlos en tiempo de compilación, identificando el coste de cada una de las regiones y asignándoles un valor preestablecido. De este modo, los parámetros se podrían expresar en número de operaciones enteras o de punto flotante. Otra posibilidad consiste en determinar su valor en tiempo ejecución. Durante la ejecución del programa, y antes de alcanzar la rutina de inspección, se podría ejecutar una versión instrumentalizada del lazo, en la cual se podría medir el coste asociado a cada una de las regiones. Para este caso, los parámetros α, β y γ se expresarían en unidades de tiempo. Con independencia de la fuente de obtención de estas magnitudes, su valor puede ser incluido en el contexto de ejecución del lazo, estando a disposición del algoritmo de balanceo de la carga.

El parámetro υ_p representa el poder de cálculo asociado al procesador p[5]. Este parámetro puede tomar distintos valores en el caso de sistemas paralelos heterogéneos, o en aquellos en los que cada procesador está ejecutando adicionalmente otras tareas con distinta carga de trabajo. La estimación del parámetro υ_p puede hacerse en tiempo de ejecución mediante una monitorización del estado del sistema paralelo y de su carga de trabajo.

La determinación de los valores de Z^I, Z^{II}, Z^{III} y Z^{IV} puede hacerse empleando distintas estrategias. La primera de ellas consiste en un análisis del vector de indirección en cada una de las regiones compartidas y el recuento de las entradas de cada una de las subregiones. Hemos descartado esta alternativa debido al enorme coste de procesamiento que conlleva.

Otra alternativa consiste en la elaboración de un histograma de accesos de a. En él, se almacena el número de accesos sobre cada entrada de dicho vector. De este modo, únicamente habría que considerar la partición asociada a cada una de las subregiones y sumar las entradas del histograma. Esta opción nos permitiría obtener un resultado

[5]En este caso, estamos asumiendo que la partición p-ésima es ejecutada por el p-ésimo procesador.

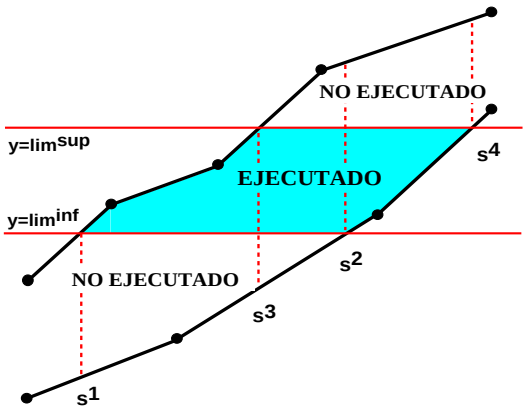


Figura 4.42: Representación de solape entre regiones compartidas.

preciso, sin embargo, la hemos descartado porque implicaría un aumento del coste de procesamiento y de almacenamiento del inspector.

Nuestra elección es la tercera alternativa, la cual consiste en la aproximación del número de entradas en función del área del patrón de accesos. Para que esta aproximación sea válida, la distribución de las entradas del patrón debe ser *localmente homogénea*. Por localmente homogéneas entendemos que no experimenten fuertes cambios dentro de cada región compartida. Bajo esta aproximación los valores de Z^I, Z^{II}, Z^{III} y Z^{IV} pueden expresarse mediante las siguientes relaciones:

$$Z_p^I \simeq |\mathcal{X}_{s^1}^{s^2}| * \frac{A^I}{A^I + A^{III}} \qquad \text{y} \qquad Z_p^{III} \simeq |\mathcal{X}_{s^1}^{s^2}| - Z_p^I \tag{4.46}$$

$$Z_p^{II} \simeq |\mathcal{X}_{s^3}^{s^4}| * \frac{A^{II}}{A^{II} + A^{IV}} \qquad \text{y} \qquad Z_p^{IV} \simeq |\mathcal{X}_{s^3}^{s^4}| - Z_p^{II} \tag{4.47}$$

En donde A representa el área de cada una de las subregiones. Una situación particular ocurre cuando $s^3 < s^2$. Este es el caso mostrado en la Figura 4.42. Para esta situación, la región exclusiva no contiene ninguna entrada del vector de indirección, pudiéndose aproximar la función de coste por la siguiente expresión.

$$C_p = (\alpha * Z_{ejecutado} + \beta * Z_{no\ ejecutado}) * \upsilon_p \tag{4.48}$$

Donde $Z_{ejecutado}$ y $Z_{no\ ejecutado}$ contienen las entradas de las regiones mostradas en la Figura 4.42, y nuevamente pueden expresarse en función del área del patrón de acceso.

El cálculo de todas estas área se puede realizar empleando únicamente la representación IARD de la indirección. El proceso es el siguiente: el patrón de indirección se divide en secciones lineales mediante el algoritmo *obtiene_SL* descrito en el Capítulo 3. A continuación, se obtiene el valor del área mediante la integración analítica de la superficie

comprendida en la partición de cada sección lineal. Debido a la estructura simple de las secciones lineales, este proceso de cálculo se puede realizar con un coste despreciable. Esta información es utilizada por el algoritmo de particionamiento, cuya estructura se describe en la siguiente sección.

Algoritmo de particionamiento

A grandes rasgos, el funcionamiento del algoritmo particionador es el siguiente: inicialmente se parte de una distribución por bloques del vector a. A continuación, evalúa, mediante la Expresion 4.43, el coste computacional asociado a cada una de ellas. En caso de existir desbalanceo de carga, modifica el tamaño de las mismas mediante un proceso iterativo. El mecanismo de convergencia es adaptativo y utiliza la caracterización IARD con el fin de acelerar el proceso de análisis.

El conjunto $\mathcal{LIM}$, introducido en la Definición 4.2.2, establece los límites de cada partición de a. Un particionamiento concreto de este vector tiene asociado un conjunto de cargas de trabajo $\mathcal{CT}$. Este conjunto se define del siguiente modo.

Definición 4.4.1 Definimos el **conjunto de carga de trabajo** $\mathcal{CT}$ de la partición $\mathcal{LIM}$, como el conjunto de todas las cargas de trabajo asociadas a cada una de las particiones de $\mathcal{LIM}$.

$$\mathcal{CT} = \{C_1, C_2, \ldots C_{N_{pt}}\} \tag{4.49}$$

Donde C_i es el coste asociado a la partición $\mathcal{A}_i$, dado por la Expresión 4.43. □

De este modo, nuestro problema se puede plantear como la búsqueda de un particionamiento óptimo, que denominaremos $\mathcal{LIM}_{óptimo}$ que obtenga el mejor balanceo de la carga. Si denominamos $\mathcal{CT}_{óptimo}$ como el conjunto de carga de trabajo asociado a $\mathcal{LIM}_{óptimo}$, y $c_{max} = max(\mathcal{CT}_{óptimo})$ como el máximo valor de carga del conjunto, entonces el balanceo de carga será el óptimo cuando c_{max} tome el mínimo valor posible. La búsqueda del balanceo de carga óptimo entre todas las posibles particiones es un problema NP completo. Debido a que el tiempo de cálculo del particionador es un factor crítico, hemos desarrollado un algoritmo heurístico que no analiza el conjunto completo de posibles soluciones, sino que únicamente considera aquellas más próximas al resultado óptimo.

La Figura 4.43 muestra el pseudocódigo de nuestra propuesta, que denominamos algoritmo **balanceador de carga** (algoritmo `BLNC`). Vamos a asumir que el algoritmo `BLNC` parte de una distribución por bloques de igual tamaño. Sin embargo, este no es un requisito y el algoritmo tendría el mismo funcionamiento para otras distribuciones iniciales de a. Los argumentos de entrada son la representación IARD de la indirección, el conjunto de particionamiento, el conjunto de parámetros α, β y γ, y el poder de cálculo de

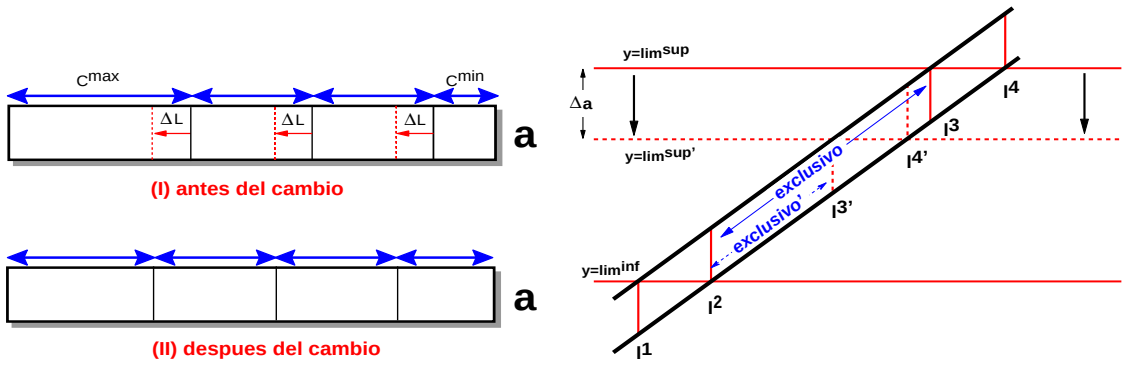
```
Algoritmo BLNC
entrada
```
IARD: representación IARD de x
$\{\alpha, \beta, \gamma\}$: coste computacional de cada región.
$\Upsilon = \{\upsilon_i, \ \ 1 \leq i \leq N_P\}$: poder de cálculo de cada procesador
r_{limit}: umbral de convergencia.
$\mathcal{LIM}$: conjunto de particionamiento inicial.
```
salida
```
$\mathcal{LIM}$: conjunto de particionamiento balanceado.

```
inicio del algoritmo
```
$\mathcal{CT} \Longleftarrow evaluar\_carga(\alpha, \beta, \gamma, \Upsilon, \mathcal{LIM}, \{E^U, E^L, \rho\})$
$C^{max} = max(\mathcal{CT})$
$C^{min} = min(\mathcal{CT})$
$r = C^{min}/C^{max}$
```
WHILE
```
$(r < r_{limit})$
$\kappa \Longleftarrow evaluar\_densidad(\mathcal{LIM}, C^{max}, \{E^U, E^L, \rho\})$
$\mathcal{A} = (C^{max} - C^{min})/(2 * \kappa * \gamma)$
$\Psi \Longleftarrow cambio\_particion(\mathcal{LIM}, \mathcal{A})$
$C_\Psi \Longleftarrow evaluar\_carga(\alpha, \beta, \gamma, \Upsilon, \mathcal{LIM}, \{E^U, E^L, \rho\})$
$C^{max} = max(\mathcal{CT})$
$C^{min} = min(\mathcal{CT})$
$r = C^{min}/C^{max}$
```
END WHILE
fin del algoritmo
```

Figura 4.43: Algoritmo para balanceo de la carga (`BLNC`).

cada procesador. Mediante el parámetro r_{limite} establecemos un límite en el proceso de convergencia.

El funcionamiento del algoritmo `BLNC` es el siguiente: la función *evaluar_carga* obtiene el conjunto $\mathcal{CT}$ asociado al conjunto de particionamiento. Esta función únicamente aplica la Expresión 4.43 utilizando las aproximaciones dadas por las ecuaciones 4.46 y 4.47. A continuación se obtienen los valores extremales C^{max} y C^{min} del conjunto $\mathcal{CT}$. En base a estos valores, se comprueba el nivel de desbalanceo existente. Hemos utilizado como criterio de desbalanceo el cociente de estas dos magnitudes $r = \frac{C^{min}}{C^{max}}$, de modo que



(a) Modificación de una partición (b) Aproximación de banda homogénea

Figura 4.44: Esquemas de partición empleando la representación IARD.

$0 \leq r \leq 1$. Mediante el parámetro r_{limite} se establece un valor máximo de desbalanceo. En el caso de que $r \geq r_{limite}$ se entiende que el desbalanceo es suficientemente bueno y el proceso se detiene. En caso contrario, los límites de las particiones se modifican mediante la función *cambio_particion*.

Esta función desplaza todos los límites de las particiones situadas entre aquella con máximo valor de carga (C^{max}) y la que tiene un valor mínimo (C^{min}). La Figura 4.44(a) muestra un ejemplo de este proceso. Se puede apreciar como únicamente las particiones asociadas a C^{max} y C^{min} modifican el tamaño de su partición. Concretamente, la primera de ellas (aquella con carga máxima) se ve reducida en ΔL entradas, mientras que la partición que origina una carga mínima aumenta sus entradas en la misma cantidad. El número de entradas de las particiones intermedias no se ve modificado, cambiando únicamente sus límites.

El proceso de determinación del valor de ΔL se describe utilizando el ejemplo de la Figura 4.44(b). Esta figura representa una sección de la envolvente, la cual contiene una partición del vector a inicialmente comprendida entre las rectas $y = lim^{inf}$ e $y = lim^{sup}$. El límite superior de esta partición es modificado pasando a ser el dado por la recta $y = lim^{sup'}$. La Figura 4.44(b) muestra el efecto de cambiar este límite. Se puede apreciar que, para un valor reducido de ΔL, y asumiendo una distribución homogénea de las entradas, las regiones compartidas no experimentan una reducción en el número de entradas, dado que sus dimensiones se mantienen.

En la figura se puede apreciar que es la región exclusiva la que experimenta la reducción en el número de entradas. Vamos a asumir, sin pérdida de generalidad, que el poder de cálculo de cada procesador es el mismo. De este modo, el incremento de carga

computacional puede aproximarse por la siguiente expresión:

$$\Delta C = C^{fin} - C^{ini} \simeq C^{fin}_{excl} - C^{ini}_{excl} = (Z^{fin}_{excl} - Z^{ini}_{excl}) * \gamma \equiv \Delta Z_{excl} * \gamma \tag{4.50}$$

Si definimos C^{dif} como la máxima diferencia de carga para un conjunto de particionamiento, entonces, para obtener un buen balanceo de carga, nuestra propuesta reduce la partición con carga C^{max} en una cantidad $C^{dif}/2$. La misma cantidad es añadida a la partición con menor carga computacional. Es decir, $\frac{C^{dif}}{2} = C^{max} - C^{min}$.

Introducimos el parámetro κ como la densidad de entradas entorno al límite superior de la partición. Dicho de otra forma, κ contiene el número de accesos realizados sobre una entrada de a cercana a lim^{sup}. Este parámetro es obtenido mediante la función *evaluar_densidad*. Utilizando esta variable, el valor de ΔZ_{excl} puede ser aproximado por la siguiente expresión.

$$\Delta Z_{excl} \simeq \kappa * \Delta L \tag{4.51}$$

Dado que $C^{dif}/2 \simeq \Delta Z_{excl} * \gamma$, el número de entradas desplazadas es:

$$\Delta L \simeq \frac{C^{dif}}{2 * \kappa * \gamma} \tag{4.52}$$

El algoritmo `BLNC` emplea la función $modifica\_partición$ para desplazar las entradas una cantidad ΔL. Una vez obtenido un nuevo conjunto de particionamiento, se vuelve a evaluar las particiones con mayor y menor carga, y en el caso de que su cociente siga siendo inferior a r_{limit} se vuelve a repetir el proceso anterior.

Experimentalmente, se obtiene que el valor de ΔL decrece monótonamente. En las primeras iteraciones, la distribución suele estar muy desbalanceada, por lo que los valores obtenidos para ΔL suelen ser muy elevados. En las últimas iteraciones la distribución del vector a suele ser próxima a la óptima, haciendo que ΔL sea reducido. En una situación real, la corrección realizada por la operación de desplazamiento tiene asociada un cierto grado de error, que causa que la reducción en el desbalanceo no sea la mejor. Sin embargo, y dada la naturaleza adaptativa del proceso, este error es detectado y reducido en las siguientes iteraciones del algoritmo.

Análisis de eficiencia

Hemos evaluado las particiones obtenidas con nuestra propuesta sobre un sistema multiprocesador Silicon Graphics Origin200 utilizando el compilador MIPSpro `f77` v7.2.1. La Tabla 4.16 muestra el tiempo de ejecución de nuestro particionador para diferentes matrices y valores de α, β y γ. Como matrices de entrada hemos empleado las *bcsstk14*,

bcsstk17 y *bcsstk29* de la *Harwell Boeing sparse matrix collection* y la matriz sintética *diag_block*. Sus características se pueden consultar en las tablas 2.4 y 4.3. El tiempo de ejecución es proporcional al número de elementos de la representación IARD y a las características del patrón de acceso. Nótese que este tiempo de ejecución no depende del número de *slices*. Para las matrices de prueba utilizadas, el tiempo de ejecución del código paralelo es de varias decenas de ms, siendo este un valor muy superior al de nuestro particionador (que es del orden de cientos de μs). Este hecho hace que nuestra propuesta resulte muy adecuada en entornos para los que es necesario redistribuir con cierta frecuencia la carga computacional sin que el patrón de acceso sufra modificaciones.

Con el fin de evaluar el impacto del empleo de la representación IARD en la eficiencia de nuestro particionador, hemos desarrollado una versión que analiza el patrón de accesos sin acceder a la información de la envolvente. En su lugar, esta versión utiliza la representación de los vectores u y l (introducidas en el Capítulo 2) obteniendo una información mucho más precisa acerca del contorno del patrón de acceso. Las medidas asociadas a este caso aparecen marcadas con el símbolo "$*$" y se muestra en la última columna de la tabla. El fuerte incremento en el tiempo de ejecución, es debido a una mayor complejidad del proceso de análisis, dado que los conjuntos u y l tienen muchas más entradas que los conjuntos $\mathcal{E}^u$ y $\mathcal{E}^l$. Para evaluar la eficiencia del balanceo de la carga obtenido, hemos utilizado el lazo de la Figura 4.41(a) con una carga consistente en operaciones en punto flotante que varían entre 1 y 30. El código paralelo fue escrito en Fortran 77 utilizando las directivas OpenMP y el compilador MIPSpro `f77` v7.2.1. Los resultados obtenidos para tres procesadores se muestran en la Tabla 4.17. Esta tabla representa el porcentaje de mejora en el tiempo de ejecución respecto a dos casos:

- LHS: se corresponde a una distribución de la matriz a por bloques de igual tamaño.
- BLK: consiste en una distribución por bloques del vector de indirección. Esta situa-

$\{\alpha,\beta,\gamma\}$	$\{1,0,1\}$	$\{2,1,1\}$	$\{3,1,2\}$	$\{4,1,3\}$	$\{5,1,4\}$	$\{5,1,4\}^*$
bcsstk14	176 μs	234 μs	313 μs	238 μs	238 μs	7432 μs
bcsstk17	276 μs	264 μs	260 μs	199 μs	262 μs	49735 μs
bcsstk29	59 μs	558 μs	444 μs	441 μs	256 μs	87506 μs
diag_block	75 μs	391 μs	507 μs	654 μs	396 μs	25387 μs

*Ejecutado sin la información IARD.

Tabla 4.16: Tiempo de ejecución del particionador para 3 particiones.

ción se da para $\alpha = 1$, $\beta = 0$ y $\gamma = 1$.

Mediante nuestra estrategia, el tiempo de ejecución se reduce significantemente en relación con una distribución por bloques de la matriz a. Comparando los resultados con los obtenidos mediante una distribución por bloques de x, nuestra propuesta obtiene, para un gran número de casos, los mejores resultados cuando β es distinto de cero, lo cual equivale a considerar el coste de las rutinas condicionales.

Otro factor que hemos evaluado en este trabajo es la calidad del resultado en términos de balanceo de carga computacional. Con el fin de obtener una medida de referencia de este parámetro, hemos desarrollado un algoritmo que obtiene la mejor distribución de la carga en términos dados por la Expresión 4.43. Este algoritmo realiza una búsqueda exhaustiva sobre todos los posibles conjuntos de particionamiento. Para cada uno de ellos, evalúa el número exacto de entradas de x existentes en cada región, obteniendo la carga computacional exacta. La obtención de estos valores implica el análisis directo sobre el vector de indirección que, como se ha visto anteriormente, resulta muy costoso. Las tablas 4.18 y 4.19 muestran, para una distribución en tres particiones, los valores obtenidos en las rectas asociadas a la segunda partición. Más específicamente, estas tablas representan los valores de lim_2^{inf} y lim_2^{sup} para $\alpha = 5$, $\beta = 1$ y $\gamma = 4$. El parámetro r' se define como la relación entre la carga máxima obtenida con ambas estrategias. Es decir, $r' = \frac{C_{max}^{\texttt{BLNC}}}{C_{max}^{óptimo}}$. Se puede apreciar la gran precisión obtenida mediante nuestra propuesta. Debido a la estructura por bloques del patrón sintético, la solución obtenida por el algoritmo heurístico coincide con la óptima.

4.4.2 Balanceo de la carga en las estrategias `PRT` y `SPRT`

Los algoritmos de balanceo de carga presentados en la sección previa pueden adaptarse para permitir su empleo en otras estrategias de paralelización. En esta sección vamos a

Cuerpo de lazo		*bcsstk14*		*bcsstk17*		*bcsstk29*		*diag_block*	
FLOPs	$\{\alpha, \beta, \gamma\}$	LHS	BLK	LHS	BLK	LHS	BLK	LHS	BLK
1	$\{3, 1, 2\}$	14.6%	11.6%	11.2%	5.1%	11.5%	11.5%	37.2%	34.0%
5	$\{4, 1, 3\}$	11.8%	8.3%	10.2%	3.2%	8.0%	8.0%	25.7%	31.6%
10	$\{5, 1, 4\}$	8.6%	4.5%	8.6%	0.7%	4.8%	4.8%	16.0%	20.0%
20	$\{5, 1, 4\}$	5.8%	1.4%	8.4%	0.0%	2.0%	2.1%	16.9%	16.7%
30	$\{5, 1, 4\}$	4.8%	0.3%	8.5%	-1.6%	0.7%	0.8%	16.8%	5.8%

Tabla 4.17: Relación de mejora para la reducción irregular con $N_{pt} = 3$.

abordar su uso en las estrategias PRT y SPRT.

La Figura 4.12 muestra el ejecutor paralelo correspondiente a la estrategia PRT para el caso de reducciones irregulares. Se puede apreciar que las entradas de la indirección pertenecientes a la región exclusiva tienen un tratamiento diferente a las pertenecientes a las regiones compartidas. Dentro de estas últimas, únicamente se consideran las entradas que acceden dentro de la partición considerada. Por lo tanto, tenemos que: $\beta = 0$, $\alpha \neq 0$ y $\gamma \neq 0$, con $\alpha \neq \gamma$. Los valores concretos de los parámetros α y γ dependen de las características de la arquitectura del sistema.

La Figura 4.30(b) ilustra el ejecutor paralelo de la estrategia SPRT. En esta figura se puede apreciar que todos los elementos del vector de indirección que acceden a la partición considerada son procesados de la misma forma. Es decir, debido a la naturaleza de esta propuesta, no se establece una distinción entre regiones compartidas y exclusivas, siendo procesadas todas las entradas como exclusivas. De este modo, tendremos que $\alpha \neq 0$, $\beta = 0$ y $\gamma \neq 0$, con $\alpha = \gamma$.

En este contexto, el nivel de balanceo de la carga viene determinado por la distribución de los accesos realizados por la indirección. Esta distribución se puede representar como un histograma de accesos sobre a. Todos los patrones de acceso que hemos utilizado en este capítulo tienen asociado un histograma de acceso homogéneo. En estas situaciones, un particionamiento por bloques de a origina una distribución de las computaciones relativamente homogénea, que no da lugar a grandes desbalanceos de carga.

Con el fin de evaluar el algoritmo BLNC para situaciones en las que existan fuertes

Matriz	*bcsstk14*			*bcsstk17*		
	lim_2^{inf}	lim_2^{sup}	r	lim_2^{inf}	lim_2^{sup}	r
Optimal	644	1158	1.000	4191	7349	1.000
IARD	647	1167	1.006	4186	7333	1.005

Tabla 4.18: Comparación entre la precisión del resultado óptimo y heurístico.

Matriz	*bcsstk29*			*diag_block*		
	lim_2^{inf}	lim_2^{sup}	r	lim_2^{inf}	lim_2^{sup}	r
Optimal	4674	9037	1.000	375	750	1.000
IARD	4737	9081	1.011	375	750	1.000

Tabla 4.19: Comparación entre la precisión del resultado óptimo y heurístico.

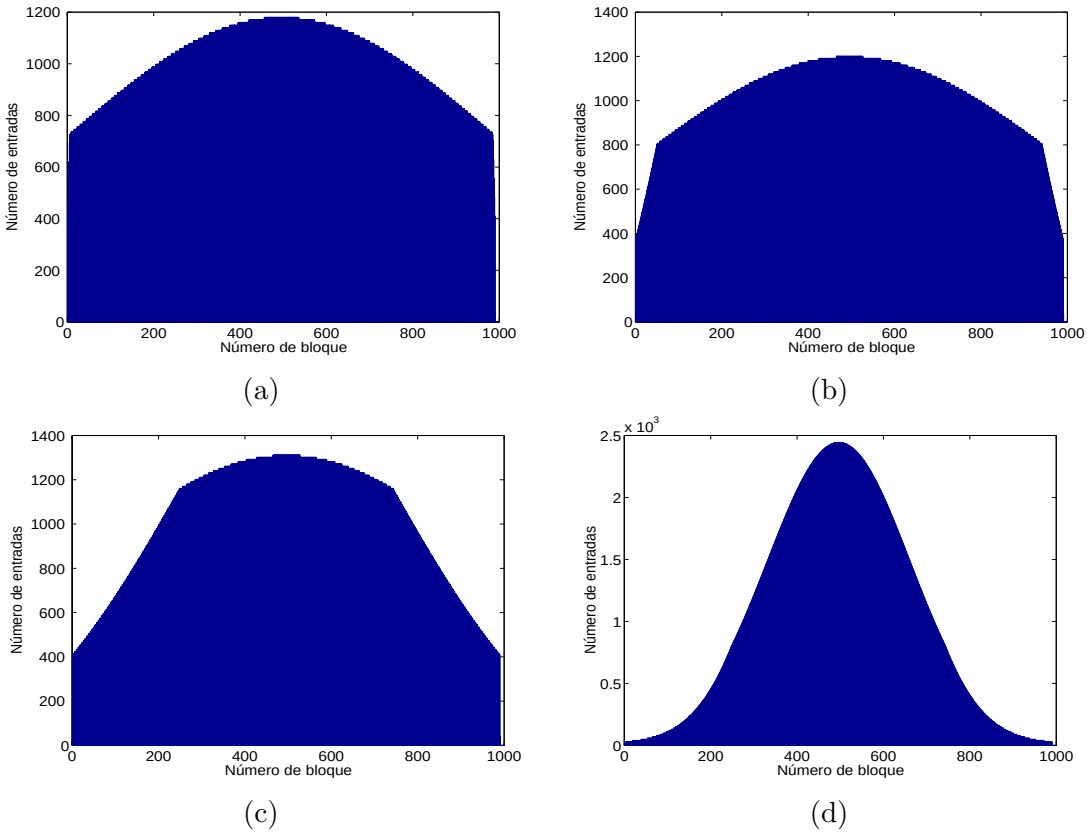


Figura 4.45: Histograma de accesos para matrices sintéticas desbalanceadas con N_x = 1M: (a) $\sigma = 2, \lambda = 1k$, (b) $\sigma = 2, \lambda = 10k$, (c) $\sigma = 2, \lambda = 50k$, (d) $\sigma = 6, \lambda = 50k$.

desbalanceos, hemos generado una familia de matrices sintéticas con una distribución no homogénea de sus entradas. Del mismo modo que en ejemplos anteriores, el patrón de estas matrices es una banda de anchura λ constante. El número de columnas es constante, y de valor 100k para toda la familia de matrices. Sin embargo, la distribución de las entradas en el interior de la banda no es homogénea, sino que guarda una distribución gausiana cuyo centro está situado en el *slice* s = 50k. La Figura 4.45 muestra el histograma de accesos de estas nuevas matrices. En este histograma se divide el vector a en 1000 bloques contiguos de 100 entradas, y se representa el número de accesos realizados sobre cada uno de estos bloques. En la Figura 4.45, se muestran patrones con $\sigma = 2$ y $\sigma = 6$, donde σ es la desviación estándar de la distribución.

Para este conjunto de matrices hemos aplicado el algoritmo `BLNC` para la técnica `SPRT`.

Específicamente, hemos utilizado $\alpha = 1$, $\beta = 0$ y $\gamma = 1$. Los resultados obtenidos se ilustran en las tablas 4.20 y 4.21. La estructura de estas tablas es la siguiente: para cada tipo de matriz, la variable $r^{inicial}$ indica el cociente de la mayor y menor carga (evaluadas mediante la Expresión 4.43) cuando se realiza un particionamiento de a en bloques de igual tamaño. Para esta distribución aplicamos el algoritmo `BLNC` con un $r_{limit} = 0.99$. Adicionalmente hemos limitado el máximo número de iteraciones que puede realizar a un

N_x	Parámetro	Ancho de banda				
		λ =100	λ =1000	λ =10000	λ =25000	λ =50000
5M	$r^{inicial}$	0.65	0.64	0.54	0.45	0.45
	r^{final}	0.99	0.99	0.99	0.99	0.98
	Tiempo (μs)	3642	3426	9534	26530	223869
	Iterariones	17	18	49	132	1000
1M	$r^{inicial}$	0.67	0.68	0.57	0.48	0.48
	r^{final}	0.99	0.99	0.99	0.99	0.99
	Tiempo (μs)	3153	4120	10740	28150	209955
	Iterariones	15	19	52	133	913
0.5M	$r^{inicial}$	0.66	0.65	0.58	0.50	0.51
	r^{final}	0.99	0.99	0.99	0.99	0.98
	Tiempo (μs)	4055	3049	10843	30077	196985
	Iteraciones	19	14	52	136	1000

Tabla 4.20: Eficiencia del inspector para distintos particionamientos ($\sigma = 2$).

N_x	Parámetro	Ancho de banda				
		λ =100	λ =1000	λ =10000	λ =25000	λ =50000
5M	$r^{inicial}$	0.04	0.03	0.03	0.04	0.12
	r^{final}	0.99	0.99	0.99	0.91	0.74
	Tiempo (μs)	6843	7988	24481	191817	190723
	Iteraciones	37	39	127	1000	1000
1M	$r^{inicial}$	0.09	0.09	0.09	0.09	0.17
	r^{final}	0.99	0.99	0.99	0.95	0.75
	Tiempo (μs)	6811	7575	24312	183328	188806
	Iteraciones	34	35	117	1000	1000
0.5M	$r^{inicial}$	0.20	0.20	0.20	0.18	0.25
	r^{final}	0.99	0.99	0.99	0.98	0.99
	Tiempo (μs)	7622	7502	19533	175501	237342
	Iteraciones	36	35	108	1000	926

Tabla 4.21: Eficiencia del inspector para distintos particionamientos ($\sigma = 6$).

valor de 1000, de modo que una vez alcanzado este valor el algoritmo finaliza aún sin haber alcanzado r_{limit}. La variable r^{final} muestra el balanceo obtenido con nuestra propuesta, nótese los excelentes resultados obtenidos. Adicionalmente estas tablas muestran el tiempo de ejecución del algoritmo `BLNC` expresado en μs y el número de iteraciones realizadas.

Se puede apreciar que partiendo de situaciones muy mal balanceadas, nuestra propuesta es capaz de conseguir unos balanceos óptimos en un reducido número de iteraciones. Una importante característica del algoritmo `BLNC` es que su tiempo de ejecución no depende del número de entradas del patrón de indirección, ya que emplea únicamente la información geométrica asociada al patrón de acceso.

En el próximo capítulo abordamos la paralelización de lazos irregulares que contienen más de un vector de indirección. Como veremos, debido a la existencia de más de un acceso irregular, buena parte de las propuestas anteriores no pueden ser aplicadas siendo necesario plantear nuevas soluciones.

Capítulo 5

Optimización de códigos irregulares con varias indirecciones

En este capítulo extendemos nuestro estudio a códigos irregulares con un número arbitrario de indirecciones y un tipo de acceso genérico. En el contexto en el que está enmarcado nuestro trabajo, esta clase de códigos supone el caso más general que podemos contemplar.

Como frecuentemente ocurre en el ámbito de la optimización de programas, el rendimiento de una técnica de optimización que intenta abordar un gran número de escenarios suele ser inferior al obteniendo mediante técnicas específicas a cada problema concreto. Este fenómeno ocurre para las distintas aportaciones presentadas en este capítulo: en total presentamos cuatro técnicas noveles de paralelización de lazos con varias indirecciones. Dos de ellas, denominadas `LCYT` y `LO-LCYT`, abordan la paralelización de códigos con una estructura general, obtenido una mejora significativa respecto a otras propuestas existentes. Las otras dos técnicas presentadas en este capítulo, y que denominamos `OWNCR` y `SLCSRT`, están orientadas a la paralelización de una clase particular de códigos irregulares. Estas propuestas son menos generales que las anteriores, sin embargo, obtienen una mayor eficiencia en la ejecución del programa paralelo. Una quinta propuesta, centrada en la paralelización de códigos cuyos accesos son exclusivamente reducciones irregulares, es presentada en el siguiente capítulo. En el marco del esquema general mostrado en la Figura 1.3, todas estas contribuciones pertenecen al módulo Paralelización de lazos con varias indirecciones. Las dos primeras se corresponden a la familia de códigos etiquetados como de clase A, mientras que las técnicas `OWNCR` y `SLCSRT` abordan códigos pertenecientes a la clase B.

El trabajo presentado en este capítulo ha contribuido al desarrollo de las siguientes publicaciones: "Improving Locality in the Parallelization of Doacross Loops" presentada en el *8th International EUROPAR 2002 Conference* en agosto del 2002 [97]; "Exploiting Locality in the Run-Time Parallelization of Irregular Loops" publicada en el *31th International Conference on Parallel Processing* en agosto del 2002 [96] y "Increasing the parallelism of irregular loops with dependences" publicada en el *9th International EUROPAR 2003 Conference* en agosto del 2003 [133] .

La organización de este capítulo es la siguiente: comenzaremos con una introducción en la que se describe la problemática que presenta esta clase de códigos en el contexto de la paralelización automática. En esta misma sección se introducirá alguna terminología que nos permitirá describir de una manera más rigurosa la estructura de este tipo de códigos. A continuación, en la Sección 5.2 se realiza una revisión bibliográfica de los trabajos relacionados con la paralelización automática de códigos irregulares de más de una indirección. Una de las propuestas existentes, denominada algoritmo `CYT`, se describe de forma más detalla en la Sección 5.3. Basándonos en esta propuesta, introducimos dos nuevas técnicas denominadas algoritmos `LCYT` y `LO-LCYT`, las cuales son descritas y evaluadas en la Sección 5.4. Posteriormente, en la Sección 5.5 evaluamos el uso de la representación IARD como una alternativa a la paralelización de esta clase de códigos. En la última sección, presentamos dos nuevas propuestas, denominadas `OWNCR` y `SLCSRT`, que permiten extraer el paralelismo de un modo más eficiente para ciertos códigos irregulares.

5.1 Introducción

Vamos a comenzar nuestro estudio con el análisis del lazo mostrado en la Figura 5.1(a). En este lazo irregular, los elementos del vector a son leídos y escritos por medio de los vectores de indirección x_1 y x_2. En función del valor particular de los vectores de indirección, pueden existir dependencias de datos. Supongamos, por ejemplo, que $N_x = 4$, $N_a = 8$, $x_1 = \{1, 3, 5, 7\}$ y $x_2 = \{2, 4, 6, 8\}$. Para estos valores, las entradas pares de a son leídas, mientras que las impares son escritas. Además, y dado que todos los accesos se realizan en distintas posiciones de memoria, no existe ninguna dependencia, por lo que el lazo completo puede ser ejecutado en paralelo. Supongamos ahora otra situación en la que los vectores de indirección tienen los valores $x_1 = \{2, 4, 6, 8\}$ y $x_2 = \{1, 3, 2, 4\}$. Para este nuevo caso, en la tercera iteración se lee la segunda entrada del vector a, la cual ha sido previamente escrita en la primera, existiendo una dependencia verdadera entre ambas iteraciones. Siguiendo el mismo argumento, tenemos que la cuarta iteración guarda una dependencia verdadera respecto a la segunda. La existencia de estas dependencias hace

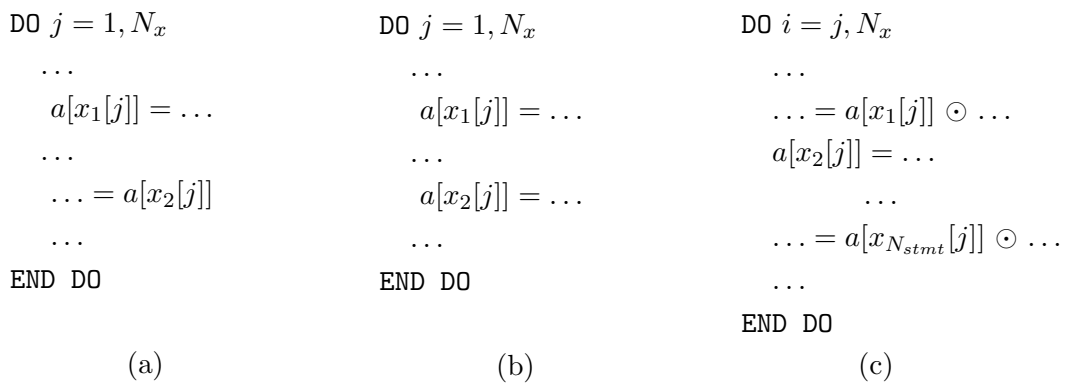


Figura 5.1: Ejemplos de lazo irregulares con varias indirecciones: (a) asignación de escritura y lectura (b) doble asignación de escritura doble, (c) esquema general.

que el lazo no sea totalmente paralelo. Sin embargo, tampoco es puramente secuencial, ya que la segunda iteración puede ejecutarse a la vez que la primera sin originar dependencias de datos. Del mismo modo, la cuarta iteración puede ejecutarse a la vez que la tercera.

En base a este ejemplo observamos que el grado de paralelismo de esta clase de lazos irregulares depende totalmente de las características e interrelación entre los patrones de acceso asociados a cada indirección. Desde un punto de vista general, el código de la Figura 5.1(a) puede tener los tres tipos de dependencias comentadas en Capítulo 1. Expresadas en función de las indirecciones, tenemos el siguiente conjunto de expresiones:

Dependencias verdaderas:	$\exists\, i',\quad i' \leq i,\quad x_1[i'] = x_2[i]$
Dependencias de salida:	$\exists\, i',\quad i' < i,\quad x_1[i'] = x_1[i]$
Falsas dependencias:	$\exists\, i',\quad i' < i,\quad x_2[i'] = x_1[i]$
Dependencias de lectura:	$\exists\, i',\quad i' < i,\quad x_2[i'] = x_2[i]$

En esta clasificación hemos incluido las dependencias de lectura. Estas dependencias son ficticias, dado que un cambio en el orden de accesos no origina un cambio en el resultado. Sin embargo, hemos citado este tipo de dependencias porque, como se verá posteriormente, van a ejercer una influencia sobre el rendimiento de nuestras propuestas.

Aparte de los valores concretos almacenados en los vectores de indirección, el tipo y número de dependencias está también determinado por los tipos de accesos dentro del lazo irregular. Consideremos, por ejemplo, el lazo de la Figura 5.1(b). En este caso, las únicas

dependencias que pueden existir son de salida, que pueden aparecer asociadas a accesos consecutivos de x_1 o de x_2; o bien, a accesos cruzados de ambos.

De este modo, las herramientas de paralelización automática deben tener en cuenta tanto el patrón de acceso a memoria como la estructura del lazo irregular. Con el fin de formalizar las características del código irregular considerado, vamos a introducir una serie de definiciones.

Definición 5.1.1 Dado un lazo genérico, denominamos **estamento j-k** del lazo, y lo denotamos como $stmt_j^k$, a la computación asociada a la k-ésima línea del cuerpo de dicho lazo, cuando es ejecutada para la j-ésima iteración. □

Definición 5.1.2 Dado un lazo arbitrario, denominamos la **k-ésima familia de estamentos**, y la denotamos por $STMT^k$ al conjunto de estamentos asociados a la k-ésima línea del cuerpo del lazo. Más formalmente,

$$STMT^k = \{stmt_1^k, stmt_2^k, \ldots stmt_{N_x}^k\} \tag{5.1}$$

□

En este capítulo vamos a considerar una estructura de lazo como la mostrada en la Figura 5.1(c), la cual está compuesta por N_{stmt} familias de estamentos. Si asumimos que cada una de las familias realiza un acceso irregular sobre a, y que es diferente en cada una, tendremos que existen N_{stmt} vectores de indirección. Adicionalmente, cada familia de estamentos puede tener un tipo arbitrario de acceso sobre a. Es decir, aunque en el caso de la Figura 5.1(c), la primera familia tenga una operación de lectura, y la segunda una operación de escritura, estos accesos bien podrían ser cualquier otro tipo de operación, como lecturas y escrituras irregulares, acumulaciones, reducciones, etc. Vamos a introducir como restricción que todos los accesos sobre la matriz a se realizan utilizando únicamente vectores de indirección.[1]

De forma general, y con el fin de simplificar nuestro análisis, asumimos que la única fuente de dependencias de datos es la debida a los accesos sobre la matriz a. Sin embargo, existen algunas excepciones que también vamos a poder considerar. Es frecuente encontrar ejemplos en códigos irregulares reales en los que existen familias de estamentos que no realizan accesos sobre la matriz a. Este es el caso del código de la Figura 5.2, donde para cada iteración, el primer estamento almacena en la variable $tmp1$ el resultado de una función genérica. Posteriormente, otro estamento lee esa variable, estableciéndose una dependencia verdadera entre ambos. En este caso, podemos considerar la primera

[1] No vamos a abordar la existencia de accesos mixtos (regulares e irregulares) sobre el mismo vector. Sin embargo, tal y como se comentó con anterioridad, un acceso regular siempre puede considerarse como dado a través de una indirección, quedando así contemplado dentro de nuestra propuesta.

```
DO j = 1, N_x
    tmp1 = f(j)
    ...
    a[x_1[j]] = tmp1
    ... = a[x_2[j]]
END DO
```

Figura 5.2: Ejemplo de lazo irregular.

familia de estamentos asociada a la segunda, de modo que, en una ejecución paralela del lazo, se fuerce la ejecución del estamento asociado a la lectura de $tmp1$ después de haber ejecutado el estamento asociado a su escritura. De este modo, la dependencia de datos entre ambas familias se preserva. En el caso de existir algún otro estamento que también acceda o modifique dicha variable, se puede realizar el mismo proceso, asociándolo también a los estamentos anteriores. En [143, 136] se presenta un método, denominado *program slicing*, que aplica este concepto agrupando las familias de estamentos en función de los accesos a los vectores de indirección. En este contexto, nuestra propuesta contempla estas situaciones si las dependencias de datos existentes en el conjunto de estamentos que han sido agrupados se producen dentro de la misma iteración, y si este conjunto de estamentos contiene un único acceso irregular.

5.2 Trabajo previo

En la literatura existe un gran número de trabajos en los que se plantean propuestas para la paralelización de lazos irregulares parcialmente paralelos. En una primera aproximación, estas propuestas pueden clasificarse en dos grandes familias: las técnicas basadas en la ejecución especulativa del lazo, y las técnicas que adoptan el paradigma inspector-ejecutor.

5.2.1 Ejecución especulativa

Una ejecución especulativa consiste en procesar en paralelo el lazo irregular sin tener conocimiento del tipo de dependencias que pueden existir. De este modo, el análisis de dependencias se realiza de forma simultánea a la ejecución del lazo irregular. En caso de detectarse dependencias, la solución más común consiste en cancelar la ejecución paralela y ejecutar el lazo de forma secuencial. Existen varios trabajos [107, 120] que emplean esta

estrategia de paralelización. Por lo general, estas propuestas obtienen un buen rendimiento si el lazo es totalmente paralelo, pero no permiten abordar de una manera eficiente lazos en los que existen dependencias. Parte de estas carencias pueden ser reducidas mediante mecanismos que eliminan dependencias no verdaderas y que aumentan la precisión y eficiencia del proceso de análisis de dependencias [50]. No obstante, estas estrategias cuentan con la desventaja adicional de no tener capacidad de estimar el rendimiento asociado a la ejecución paralela del lazo. Por este motivo, no es del todo clara su capacidad para abordar tópicos como el balanceo de la carga y la mejora de la localidad.

5.2.2 Estrategia Inspector-Ejecutor

Una de los primeros trabajos basado en el paradigma del inspector-ejecutor y orientado a la paralelización de lazos parcialmente paralelos fue el realizado por Zhu y Yew [154]. Un concepto clave introducido en este trabajo es el *wavefront*. Un *wavefront* se define como un conjunto de iteraciones del lazo que pueden ser ejecutadas concurrentemente y en un orden arbitrario preservando las dependencias de datos. De este modo, todas las iteraciones de un mismo *wavefront* pueden ejecutarse en paralelo mediante un DOALL. Mediante el empleo de esta técnica, las dependencias intrínsecas del lazo son preservadas asignando las iteraciones entre las que existen dependencias verdaderas a *wavefronts* diferentes. Por este motivo, el procesamiento de los distintos *wavefronts* se realiza secuencialmente. Cuando se finaliza la ejecución paralela de todas las iteraciones asignadas a un *wavefront*, es necesario realizar una operación de sincronización tipo BARRERA.

La estrategia propuesta por Zhu y Yew tiene tres importantes limitaciones. La primera de ellas está originada por el hecho de que la fase de inspección y ejecución están fuertemente acopladas, formando, de hecho, una única etapa. Esto causa que ambas deban ser ejecutadas cada vez que se ejecuta en paralelo el lazo irregular. Dicho de otro modo, la información del inspector no puede ser reutilizada por el ejecutor. El segundo gran inconveniente de la propuesta de Zhu y Yew es que realiza una paralelización del lazo a nivel de iteraciones. Esto quiere decir que el mínimo elemento distribuido (nivel de grano utilizado) es el conjunto de estamentos asociados a una iteración. Como veremos posteriormente, el empleo de un tamaño de grano más reducido aumenta significativamente la eficiencia del ejecutor paralelo. Finalmente, la tercera limitación de esta propuesta consiste en que la ejecución de lecturas consecutivas sobre la misma posición de memoria se considera dependencia de datos y, consiguientemente, se fuerza la serialización de las lecturas durante la ejecución paralela del lazo.

En [86, 109] se planten otras propuestas para la paralelización de este tipo de lazos.

Sin embargo, su aplicación resulta limitada, dado que se restringen a situaciones en las que todos los elementos del vector de indirección son diferentes. La propuesta realizada por Midkiff y Padua [98]mejora el trabajo de Zhu y Yew, permitiendo la ejecución de lecturas consecutivas en paralelo. No obstante, esta propuesta no resuelve la primera y segunda de las limitaciones anteriores, las cuales tienen una importante repercusión en el rendimiento del programa. Posteriormente, Saltz et al. [124] proponen un planteamiento alternativo en el que el inspector y ejecutor están desacoplados. Esta propuesta está limitada a lazos en los que no existen dependencias de salida, lo cual introduce una importante restricción al dominio de aplicación de la misma. La propuesta de Saltz et al. fue mejorada por Leung y Zahorjan [90], permitiendo la extensión de su uso a lazos con dependencias de salida, y proponiendo distintas estrategias para la paralelización del inspector.

Un planteamiento diferente es el realizado por Rauchwerger y Padua en [119], en el que se presenta una técnica basada en el inspector-ejecutor, para la paralelización de lazos irregulares con varias indirecciones. El empleo de esta técnica resulta limitado, ya que en caso de existir algún tipo de dependencias de datos, el lazo se ejecuta secuencialmente. Otra propuesta se puede ver en [118], en donde se propone una técnica basada en la creación de un grafo de dependencias entre iteraciones. Este grafo agrupa las iteraciones en *wavefronts* [154] que pueden ser ejecutados en paralelo. Todos estos trabajos se basan en distribuir sobre los procesadores las iteraciones del lazo, introduciendo sincronizaciones para garantizar el correcto orden de ejecución. Es decir, el nivel de grano utilizado se corresponde a una iteración del lazo irregular.

Una alternativa a estas estrategias consiste en distribuir las computaciones que conforman el cuerpo del lazo, considerando un grano del tamaño de un estamento individual. En este contexto, el algoritmo propuesto por Chen et al. (algoritmo `CYT`) [26] propone la paralelización de lazos irregulares parcialmente paralelos permitiendo un solape parcial entre los estamentos que conforman el cuerpo del lazo. Adicionalmente, el algoritmo `CYT` intenta solventar otra de las principales carencias de la propuesta realizada por Zhu y Yew, haciendo que la fase de inspección y la de ejecución estén completamente desacopladas, y permitiendo el reuso de la información de la primera. Una descripción más detallada de esta propuesta se realiza en la Sección 5.3. Recientemente, en [149] y [150] se realiza una mejora de este algoritmo, permitiendo la ejecución de lecturas consecutivas.

Un estudio comparativo del rendimiento de las propuestas basadas en un paralelismo a nivel de iteración y en un paralelismo a nivel de estamento se puede consultar en [148]. Los estudios realizados abarcan lazos con estructuras diferentes, distintos patrones de acceso a memoria y un amplio rango de cargas computacionales asociadas al lazo. Los resultados obtenidos demuestran que las estrategias basadas en un paralelismo a nivel de estamento

obtienen mejores resultados en la práctica totalidad de los casos.

5.3 Algoritmo CYT

Debido a que la estrategia CYT representa una de las propuestas punteras en este campo, hemos decidido utilizarla como punto de partida de nuestro trabajo. En las siguientes secciones realizamos una descripción pormenorizada de su estructura y funcionamiento, analizando su eficiencia tanto en términos de nivel de paralelismo obtenido como capacidad de explotación de la jerarquía de memoria. En base a los resultados obtenidos, vamos a poder plantear nuevas propuestas que permiten aumentar la eficiencia del código paralelo.

El conjunto de inspector-ejecutor que conforma la estrategia CYT propuesta por los autores en [26], incluye un inspector paralelo. La complejidad de la estructura de este inspector aumenta de forma considerable respecto a su contrapartida secuencial. Con el fin de simplificar la descripción de esta estrategia, y dado que conceptualmente el funcionamiento es el mismo, en esta sección vamos a considerar únicamente la estructura del inspector CYT secuencial. Del mismo modo, el ejecutor CYT que vamos a describir está adaptado al funcionamiento del inspector secuencial. Una información más detallada de la estrategia CYT se puede encontrar en [26].

5.3.1 Fase de inspección del algoritmo CYT

En la fase de inspección del algoritmo CYT, el patrón de acceso es analizado, almacenándose la información relativa a las posibles dependencias de datos en una estructura denominada *ticket table*. Esta estructura tiene un papel clave en esta propuesta, ya que contiene información que es empleada por el ejecutor para establecer el orden de procesamiento del lazo paralelo. Veamos, a continuación, una definición de esta tabla.

Definición 5.3.1 Una *ticket table*, que denotaremos como *ticket*, es una matriz de enteros con N_{stmt} filas y N_x columnas, donde N_{stmt} es el número de familias de estamentos que acceden a la matriz a. □

Por simplicidad, asumiremos que N_{stmt} coincide con el número total de familias de estamentos existentes en el cuerpo del lazo. Por ejemplo, para el lazo de la Figura 5.1(c), $N_{stmt} = 2$.

En la notación que vamos a emplear, utilizaremos los índices k y j para hacer referencia, respectivamente, a cada fila y columna de esta tabla. De este modo, el índice k se corresponderá a la familia de estamento y el j a la iteración del lazo.

i	1	2	3	4	5	6	7	8	9	10
X1	4	14	5	5	7	9	5	10	3	8
X2	1	7	9	10	16	3	8	14	5	12

Figura 5.3: Ejemplo de vectores de indirección.

Ticket Table

i	1	2	3	4	5	6	7	8	9	10
	1	1	1	2	2	2	3	2	2	2
	1	1	1	1	1	1	1	2	4	1

Figura 5.4: Ejemplo de *ticket table*.

Definición 5.3.2 Se define una **cadena de dependencias** asociada a $a[i]$ como el conjunto ordenado de estamentos que acceden a la i-ésima entrada de la matriz a. Estos estamentos están dispuestos de forma ordenada, respetando el mismo orden en que se procesan cuando se ejecuta el lazo secuencial. □

Consideremos el ejemplo del código mostrado en la Figura 5.1(c) y asumamos unos vectores de indirección con los valores mostrados en la Figura 5.3. En este caso, únicamente hay un acceso sobre la entrada $a[1]$, por lo que la cadena de dependencias asociada a esta entrada tendrá un único estamento que será el $stmt_1^2$. Consideremos ahora la entrada $a[5]$. Existen en total 5 accesos sobre la misma, de modo que la cadena de dependencias será el siguiente conjunto de estamentos $\{stmt_3^1, stmt_4^1, stmt_7^1, stmt_9^2\}$.

Definición 5.3.3 Dado un estamento $stmt_j^k$ de un lazo irregular, definimos el **número de secuencia** del estamento, como el puesto que dicho estamento ocupa en la cadena de dependencias a la que pertenece. □

Cada entrada $ticket[k, j]$ de la *ticket table*, contiene el *número de secuencia* de estamento $stmt_j^k$ del lazo irregular. Retomando el ejemplo anterior, y considerando los estamentos que acceden a $a[5]$, tendremos que el número de secuencia de $stmt_3^1$ es 1, y está almacenado en $ticket[1, 3]$, el número de secuencia de $stmt_4^1$ es 2, y se almacena en $ticket[1, 4]$, etc. El contenido completo de la *ticket table* se muestra en la Figura 5.4.

Una vez introducidos los principales conceptos de esta propuesta, y una vez definida la estructura de datos generada por el inspector, únicamente nos resta introducir la implementación concreta del inspector CYT. El pseudocódigo del algoritmo secuencial se muestra en la Figura 5.5. Este consiste en dos lazos anidados. El más externo recorre las iteraciones del lazo irregular y el más interno cada una de las familias de estamentos. La

```
Algoritmo CYT
entrada
        {x_1, x_2, ... x_Nstmt}: vectores de indirección
salida
        ticket: ticket table asociada al código irregular.
inicio del algoritmo
         DO  j = 1, N_x
            DO  k = 1, N_stmt
                ticket[k, j] = preparado[x_k[j]]
                preparado[x_k[j]] + +
            END DO
         END DO
fin del algoritmo
```

Figura 5.5: Inspector secuencial de la estrategia `CYT`.

idea clave de esta propuesta es que los distintos estamentos (que no familias) son procesados siguiendo el mismo orden que el algoritmo secuencial. El vector *preparado* tiene N_a entradas y se utiliza para almacenar el número de estamentos que han accedido sobre cada entrada de a. De este modo, cada vez que se considera un estamento $stmt_j^k$, su número de secuencia asociado será el valor de la entrada $x_k[j]$ del vector *preparado*.

5.3.2 Fase de ejecución del algoritmo `CYT`

El ejecutor `CYT` emplea un vector adicional, denominado *key*, para especificar el orden de ejecución de los estamentos. Este nuevo vector tiene N_a entradas y su contenido se actualiza a lo largo de la ejecución paralela del lazo. Para una iteración dada, el valor almacenado en la entrada $key[i]$ se corresponde al número de estamentos que acceden sobre la entrada $a[i]$ y que ya han sido ejecutados. De este modo, combinando la información del vector *key* con la almacenada en la *ticket table*, se puede determinar para cada estamento del lazo irregular el instante en que puede ser ejecutado.

La Figura 5.6 muestra un ejemplo de la estructura del ejecutor. El lazo externo (etiquetado como `L1`) es totalmente paralelo y tiene tantas iteraciones como número de procesadores. En el siguiente lazo (etiquetado como `L2`), las N_x iteraciones del lazo original son distribuidas de forma cíclica sobre el conjunto de procesadores.

Dentro del cuerpo del lazo, la ejecución de cada estamento está precedida por una

```
        SHARED key,a,x1,x2,ticket

L1       DOALL p = 1, Np
L2         DO  j = p, Nx, Np
                WHILE (key[a[x1[j]]] ≠ ticket[1, j])
                    rutina de espera
                END WHILE
                ... = a[x1[j]]⊙
                key[a[x1[j]] + +

                WHILE (key[a[x2[j]]] ≠ ticket[2, j])
                   rutina de espera
                END WHILE
                a[x2[j]] = ...
                key[a[x2[j]] + +
           END DO
       END DOALL
```

Figura 5.6: Ejecutor paralelo del algoritmo CYT.

rutina de sincronización. Esta rutina comprueba si el número de secuencia asociado a cada estamento coincide con el almacenado en *key*. Cuando esta igualdad se cumple, todos los estamentos pertenecientes a la misma cadena de dependencias han sido ejecutados, por lo que el estamento considerado puede ser procesado sin riesgo de dependencias. En caso contrario, se ejecuta una rutina de espera, y a continuación, se vuelve a comprobar la condición anterior. Mediante esta rutina de espera se establece el tiempo que transcurre entre dos comprobaciones consecutivas, tiempo en el que no se realiza ninguna operación. Cuando el estamento considerado es ejecutado, el campo del vector *key* se incrementa, permitiendo ejecutar el siguiente estamento de la cadena de dependencias. Mediante el empleo de esta política de ejecución, cada procesador va considerando secuencialmente los estamentos que le han sido asignados. Esto tiene como inconveniente que, a lo largo de la ejecución paralela del lazo, un procesador no puede evaluar ni ejecutar aquellos estamentos posteriores al que está siendo considerado.

5.3.3 Análisis de eficiencia

Tal y como se ha comentado en la sección previa, el ejecutor del algoritmo `CYT` realiza una distribución cíclica de las iteraciones. Existen dos motivos para la elección de este esquema de distribución. El primero de ellos se basa en que la distribución cíclica asegura, para la mayor parte de los casos, un buen balanceo de la carga computacional. El segundo motivo es que optimiza los tiempos de espera. Aquellos estamentos asociados a las últimas iteraciones del lazo tienen, en promedio, un número de secuencia superior a los asociados a las primeras iteraciones. Esto es debido a que para los últimos estamentos del lazo existe una mayor probabilidad de que otros estamentos hayan accedido previamente sobre la misma posición de memoria. De este modo, si en vez de utilizar una distribución cíclica, se empleara una por bloques, el tiempo de espera del procesador asignado al último bloque (últimas iteraciones del lazo) sería mayor que el asignado al primero.

Considerando la estructura modular de esta estrategia, el desacoplo entre la fase de inspección y la fase de ejecución permite ocultar el coste del inspector en aquellos lazos para los que se pueda reusar del inspector. Otra gran ventaja de la estrategia `CYT` es que considera dependencias entre accesos en vez de entre iteraciones. De este modo, para una iteración j dada, el estamento $stmt_j^1$ puede ser ejecutado a pesar de que el estamento $stmt_j^2$ no pueda serlo debido a la existencia de dependencias no resueltas.

Otro aspecto que es necesario considerar es el grado de explotación de la jerarquía de memoria de la estrategia `CYT`. En la Figura 5.6 se puede apreciar que la distribución de las iteraciones no explota la localidad en los accesos. Del mismo modo, la fase de inspección no realiza ningún análisis del patrón de acceso, y la información almacenada en la *ticket table* es insuficiente para obtener información acerca de su grado de localidad. De este modo, para un patrón de acceso genérico, el grado de localidad explotado por la técnica `CYT` resulta reducido. Dado que cada procesador accede a posiciones arbitrarias de a, es de esperar que el número de invalidaciones y de falsas comparticiones de líneas cache sea elevado. Del mismo modo, es de esperar que el número de fallos cache sea, para la mayor parte de los casos, significativo.

Como conclusión observamos que una importante carencia de esta estrategia es la falta de mecanismos que permitan mejorar la localidad en los accesos y explotar convenientemente la jerarquía de memoria del computador. Este hecho nos ha motivado al desarrollo de nuevas propuestas que, basadas en la estrategia `CYT`, permitan solventar eficientemente esta carencia. El resultado de nuestro trabajo ha derivado en el desarrollo de dos nuevas estrategias de paralelización que describimos en la siguiente sección.

5.4 Algoritmos LCYT y LO-LCYT

Nuestras propuestas para la paralelización del lazos parcialmente paralelos adoptan, como punto de partida, el algoritmo CYT. Hemos centrado nuestro esfuerzo en el desarrollo de una etapa mejorada de inspección que permita tener en cuenta el grado de localidad de los accesos. En las nuevas propuestas que realizamos, el ejecutor ya no realiza una distribución cíclica de las iteraciones, sino que tiene en cuenta la localidad a la hora de realizar la distribución. En las siguientes secciones describimos en detalle cada una de nuestras propuestas.

5.4.1 Fase de inspección del algoritmo LCYT

El inspector que planteamos en esta sección, y que denominamos LCYT (*Local* CYT), realiza una doble función. Por una parte, y con el fin de realizar la distribución eficiente de la carga computacional sobre los procesadores, analiza el patrón de acceso a memoria asociado a la ejecución del lazo irregular. Esta distribución debe dar lugar a un alto grado de localidad en los accesos y a un buen balanceo. Para cumplir este objetivo empleamos un grafo que representa el patrón de accesos. La segunda meta que debe alcanzarse consiste en la determinación de las dependencias de datos existentes en el lazo original, y en la elaboración de un esquema de ejecución paralelo que permita respetarlas aprovechando el máximo paralelismo. Para ello empleamos una *ticket table* para clasificar cada una de sus entradas y preservar las dependencias que puedan existir. La organización de nuestra propuesta de inspector consta de tres etapas en las que se elaboran y procesan ambas estructuras de datos. Estas etapas son la construcción del grafo de accesos, el particionamiento del grafo de acceso y la creación de la Ticket Table. A continuación describimos el funcionamiento de cada una de ellas.

1. ***Construcción del grafo de accesos.*** Con el fin de mejorar la localidad en los accesos, realizamos una división del vector a en bloques de $N_{mult}N_L$ entradas. Siendo N_L el número de entradas de a existentes en una línea cache y N_{mult} un número entero mayor o igual que la unidad. El valor concreto de N_L dependerá de las características de la cache del sistema sobre el que el código va a ser ejecutado.

   Representamos los accesos realizados sobre cada uno de estos bloques mediante un grafo, de forma que cada nodo representa un bloque de a. Dos nodos están unidos por un vértice cuando ambos contienen entradas que son accedidas en la misma iteración. Denominamos a este grafo como **grafo de acceso**. El grafo de acceso es un grafo pesado tanto en nodos como en vértices. El peso de cada nodo es el

número de iteraciones que acceden sobre el bloque asociado mediante operaciones de escritura. Con el propósito de clasificar las iteraciones del lazo, utilizamos una estructura auxiliar en forma de tabla que nos permite almacenar, para cada nodo, los índices de las iteraciones que acceden sobre el mismo. Por otra parte, el peso de cada vértice representa el número de veces que ambos son accedidos dentro de una misma iteración.

Así pues, los nodos del grafo de acceso dan idea del espacio de indireccionamiento existente, mientras que los vértices informan acerca de la estructura del patrón de acceso del lazo irregular. Mediante el grafo de acceso tenemos una representación simplificada del patrón de acceso, cuya precisión puede ser establecida mediante el parámetro N_L. Conforme disminuimos el valor de este parámetro, se obtiene una representación más detallada del patrón, a costa de aumentar el espacio de almacenamiento del grafo.

2. ***Particionamiento del grafo de acceso.*** Dado que cada nodo tiene asociado el conjunto de iteraciones que acceden sobre el mismo mediante escrituras, el particionamiento de este grafo implica la distribución de los nodos sobre los procesadores y, consiguientemente, la distribución de las iteraciones del lazo. El algoritmo de particionamiento debe obtener una distribución balanceada de los nodos y minimizar el número de vértices cortados por la partición, es decir, se debe evitar que nodos unidos por un vértice estén asignados a particiones diferentes. El primero de estos requisitos implica un correcto balanceo de la carga computacional, mientras que el segundo conlleva una reducción del número de invalidaciones en memoria cache y un aumento en el reuso.

   Adicionalmente, y dado que cada nodo representa un número de entradas igual o múltiplo del tamaño de línea cache, conseguimos obtener una mejora en la localidad en los accesos a los datos, eliminando la falsa compartición de líneas cache.

   Para llevar a cabo este particionamiento hemos utilizado el programa *pmetis* [74], el cual forma parte de la librería METIS. Esta distribución ofrece un conjunto de herramientas para llevar a cabo el análisis y particionamiento de grafos. Hemos elegido el programa *pmetis* debido a que resulta el más adecuado para los ejemplos considerados, y nos ofrece una implementación paralela del particionador. De forma más concreta, este programa emplea la estrategia de particionamiento de *multilevel recursive bisection* [12], la cual consta de tres etapas:

   2.1. La primera etapa, denominada fase de *coarsening*, tiene como objetivo reducir el tamaño del grafo y hacer más sencillo y rápido su particionamiento. De

este modo, se realiza la fusión, en un único elemento, de distintos conjuntos de nodos. Esta operación se realiza preservando la conectividad con el resto de los nodos del grafo, por lo que el grafo resultante sigue representando el patrón de acceso, pero ahora de un modo más simplificado. Esta operación es equivalente a agrupar, de forma local, los distintos bloques de a, los cuales pasan a formar un nuevo bloque con una longitud de bloque mayor o múltiplo de la original.

2.2. La segunda etapa implica un particionamiento del grafo reducido. El proceso de particionamiento debe tener en cuenta el peso de los vértices y de los nodos del grafo. El programa *pmetis* realiza este proceso mediante el algoritmo de particionamiento propuesto por Simon et al. [111].

2.3. En la última etapa, el grafo particionado es transformado en el original. Esta operación se realiza mediante la operación de *uncoarsening*, la cual realiza el procedimiento inverso a la primera etapa: aquellos nodos que fueron fusionados son ahora divididos en sus nodos primitivos. Dado que esta operación se realiza sobre el grafo particionado, el resultado final consiste en la distribución de todos los nodos sobre el espacio de particionamiento.

3. ***Creación de la Ticket Table***. El proceso de creación de esta tabla es idéntico al empleado por el algoritmo CYT. El resultado es una tabla de dependencias con $N_x * N_{stmt}$ entradas, siendo N_x el número de iteraciones del lazo y N_{stmt} el número de familias de estamentos que conforman su cuerpo. El proceso de creación de esta tabla es independiente al de las fases previas, por lo que puede realizarse en paralelo con las mismas.

Vamos a considerar el código mostrado en la Figura 5.1(c) con $N_{stmt} = 2$, y emplearemos los vectores de indirección de la Figura 5.3. El contenido de la *Ticket Table* asociada a este ejemplo es la de la Figura 5.4, mientras que la Figura 5.7 representa la distribución de las entradas de la matriz a sobre los nodos del grafo. Nótese que aparece indicado el peso de cada nodo y vértice, junto con su particionamiento en dos conjuntos asociados a cada procesador

5.4.2 Fase de inspección del algoritmo LO-LCYT

De todas las etapas del algoritmo LCYT, es el particionamiento del grafo la fase más costosa. La complejidad de la misma depende del número de nodos del grafo, que es $\lceil N_a/(N_{mult}N_L)\rceil$. Esta complejidad puede ser reducida aumentando el valor de N_{mult}, pero esto puede conllevar problemas de desbalanceo de carga.

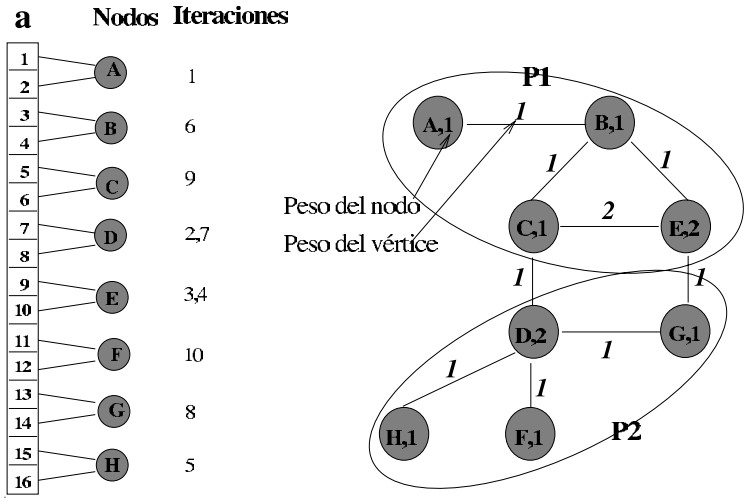


Figura 5.7: Representación del patrón de acceso mediante un grafo.

Con el propósito de reducir el coste computacional del algoritmo `LCYT`, hemos desarrollado una variante del mismo, denominada algoritmo `LO-LCYT` (*Low Overhead Local* CYT), que realiza un particionamiento del grafo utilizando un procedimiento menos costoso. De esta forma, el coste del inspector puede ser reducido significativamente a costa de obtener una solución más imprecisa, y por lo tanto, un ejecutor paralelo menos eficiente.

A diferencia del algoritmo `LCYT`, para realizar la distribución de las iteraciones únicamente consideraremos los accesos de escritura. La primera y segunda etapa del algoritmo `LCYT` son reemplazadas por las siguientes fases:

1. ***Distribución del espacio de indireccionamiento***. El vector a es nuevamente divido en $N_L * N_{mult}$ bloques. Cada uno de estos bloques equivale a un nodo de la representación anterior.

2. ***Distribución de las iteraciones***. Los bloques anteriores se distribuyen de forma cíclica sobre los procesadores. De esta forma, cada procesador tiene asignado un conjunto de bloques de a y realiza todas las iteraciones que acceden a los mismos mediante operaciones de escritura.

La tercera fase del algoritmo `LO-LCYT` es la creación de la *Ticket Table*, que se realiza de idéntica manera a la propuesta previa. Esta fase se puede realizar en paralelo con las dos anteriores.

Es necesario destacar que esta propuesta no realiza ninguna consideración respecto al número de iteraciones asociado a cada uno de los bloques, es decir, no considera la

optimización en el balanceo de la carga computacional. Sin embargo, y dado que se realiza una distribución cíclica de los nodos, para un gran número de situaciones el balanceo de la carga resulta aceptable.

Aumentando el valor del parámetro N_{mult} se puede cambiar el tamaño de bloque, incrementando así la localidad en los accesos dado que estos se realizan sobre posiciones de memoria mucho más próximas entre sí. Sin embargo, un aumento en N_{mult} tiene como inconveniente un peor balanceo de la carga. Así pues, es necesario obtener un correcto equilibrio entre ambos factores (localidad en los accesos y balanceo de la carga) mediante la elección del valor más adecuado de N_{mult}.

Dado que no utiliza un algoritmo de particionamiento, la complejidad del inspector LO-LCYT depende de N_x, el número de iteraciones del lazo. En general esta complejidad será menor que la del algoritmo LCYT, excepto para aquellos casos en los que $N_a \ll N_x$.

5.4.3 Fase de ejecución

Ambas propuestas tienen el mismo ejecutor, que utiliza la información de la *Ticket Table* para preservar las dependencias de datos en las ejecución paralela del lazo. Consiguientemente, todas las iteraciones son ejecutadas en paralelo, con la excepción de aquellas que originen dependencias.

La Figura 5.8 muestra el pseudocódigo del ejecutor. El lazo más externo tiene tantas iteraciones como número de procesadores utilizados. El uso de este lazo permite identificar cada procesador mediante el valor p del índice del lazo. La principal diferencia entre esta propuesta y la utilizada en la estrategia CYT (Figura 5.8), es que ahora las distintas iteraciones del lazo han sido previamente asignadas por el inspector. Denominamos x_1^p y x_2^p a los vectores de indirección asignados al p-ésimo procesador. Nótese que el número de entradas de estos vectores puede ser distinto para cada procesador. Con el fin de que todos los procesadores recorran el espacio de indireccionamiento que tiene asignado, el parámetro N_x tiene ahora N_p entradas, siendo cada una de ellas el número de estamentos de cada familia asignados a cada procesador. De este modo, se verifica que,

$$\| x_1^p \| = \| x_2^p \| = N_x[p] \quad \forall p \in [1, N_p] \tag{5.2}$$

5.4.4 Evaluación del rendimiento

En esta sección se comparan experimentalmente las estrategias CYT, LCYT y LO-LCYT. Comenzaremos con una descripción de las condiciones en las que hemos realizado dichas

```
SHARED key,a,x_1^{N_p},x_2^{N_p},ticket

DOALL p = 1, N_p
    DO j = 1, N_x[p]
       WHILE (key[a[x_1^p[j]]] ≠ ticket(1, j))
          rutina de espera
       END WHILE
       ... = a[x_1^p[j]]⊙
       key[a[x_1^p[j]] + +

       WHILE (key[a[x_2^p[j]]] ≠ ticket(2, j))
          rutina de espera
       END WHILE
       a[x_2^p[j]] = ...
       key[a[x_2^p[j]] + +
    END DO
END DOALL
```

Figura 5.8: Ejecutor paralelo de los algoritmos `LCYT` y `LO-LCYT`.

comparaciones. Posteriormente, evaluaremos el rendimiento de cada una de ellas. Como criterios de comparación utilizaremos el rendimiento computacional, el nivel de reuso cache, el nivel de falsa compartición de líneas cache y el balanceo de la carga. En esta sección también se abordará el impacto del coste del inspector en el rendimiento neto de cada una de las propuestas.

Condiciones experimentales

Existen tres factores que determinan en gran medida el rendimiento de un código irregular paralelo. Estos son: el tamaño del lazo, su coste computacional y el patrón de acceso a memoria. Con el fin de evaluar la influencia de cada uno de estos factores, y siguiendo los mismos pasos que otros autores [26, 150], hemos utilizado el lazo sintético mostrado en la Figura 5.9. El parámetro N_x representa el tamaño del lazo, W está asociado al coste computacional asociado a cada iteración, y el vector x determina el patrón de acceso a memoria.

La estructura de este lazo sintético es muy similar al de las rutinas de resolución de

```
DO j = 1, Nx
    tmp1 = a[x[2j − 1]]
    a[x[2j]] = tmp2
     DO w = 1, W
       carga de trabajo
     END DO
END DO
```

Figura 5.9: Código irregular de prueba.

sistemas de ecuaciones lineales dispersos. Ejemplo de estas rutinas son *lsol*, *ldsol* y *ldsoll* de la librería *Sparskit* [122]. Adicionalmente, existen en el mercado un gran número de aplicaciones que emplean esta clase de rutinas. Como ejemplos podemos citar las aplicaciones de programación lineal, códigos de elementos o diferencias finitas, problemas de optimización, etc.

Con el propósito de aproximar nuestro código de evaluación a una situación lo más cercana posible a un entorno real, hemos utilizado como vectores de indirección matrices extraídas de la librería *Harwell-Boeing* [37]. Tal y como se ha comentado con anterioridad, estas matrices representan patrones de acceso extraídos de aplicaciones reales. Adicionalmente, y con objeto de aislar e identificar el efecto en el rendimiento de ciertos parámetros críticos, hemos desarrollado un conjunto adicional de matrices sintéticas. En total, y de forma más específica, hemos utilizado cinco patrones de acceso reales y cuatro sintéticos. Las principales características de los mismos se muestran en la Tabla 5.1. En esta tabla

Matriz	$2 \times N_x$	N_a	CC	$CC * 100/N_x$
gemat1	47368	4929	4938	20.85
gemat12	33110	4929	49	0.30
mbeacxc	49920	496	487	1.95
beaflw	53402	507	500	1.87
psmigr_2	540022	3140	2626	0.97
25600_U	25600	25600	9	0.07
25600_90_10	25600	25600	45	0.35
51200_U	51200	51200	11	0.04
51200_90_10	51200	51200	46	0.18

Tabla 5.1: Características de las matrices de prueba.

aparece un campo etiquetado como CC que definimos a continuación.

Definición 5.4.1 Definimos **camino crítico** de un patrón de acceso, y lo denotamos como **CC** a la longitud de la mayor cadena de dependencias del lazo. □

El camino crítico da una indicación del grado de paralelismo de un lazo irregular. En el caso de que $CC = 1$ el lazo es totalmente paralelo, mientras que si $CC = N_x$ el lazo es completamente secuencial. Frecuentemente, este valor suele aparecer normalizado por el número de iteraciones del lazo. La última columna de la tabla muestra esta cantidad.

Retomando las propiedades de los patrones de acceso sintéticos, estos pueden clasificarse en dos grupos denominados **uniformes** y **no uniformes**. Los patrones de acceso uniformes, denotados con la terminación "*U*", han sido generados aleatoriamente asumiendo que todas las entradas de la matriz a tienen la misma probabilidad de ser accedidas. En el caso de los patrones de acceso no uniformes, se aumentó la probabilidad de acceso a ciertas entradas de a. Concretamente, para estos patrones, el 90% de los accesos se realizan únicamente sobre el 10% de las entradas de a y se denotan con la terminación "90_10". La asignación y distribución de todos estos accesos se ha realizado de forma aleatoria. El motivo del uso de patrones de acceso no homogéneo es el favorecer la presencia de *hot spots*, es decir, de regiones de memoria frecuentemente accedidas que originan un gran número de dependencias de datos. Este hecho queda patente en la Tabla 5.1, en donde se puede apreciar que el camino crítico es mucho mayor para los patrones de acceso sintéticos no uniformes.

Como plataforma de evaluación se ha utilizado una Silicon Graphics Origin 2000 con procesadores MIPS R10000 a 250MHz. Los códigos de prueba se escribieron en Fortran 77 con el compilador MIPSpro `f77` v7.4 y se paralelizaron utilizando las directivas de OpenMP. Todos los datos fueron alineados a nivel de cache mediante la opción de compilación "`-aling128`".

En nuestro caso, el tiempo de ejecución de cada iteración T se puede modificar a través del parámetro W. Más concretamente, el tiempo de ejecución de cada iteración del lazo es modelado por la siguiente ecuación:

$$T(W) \simeq 8 * 10^{-5}(1 + W) \quad ms \tag{5.3}$$

Experimentalmente, hemos evaluado el coste del los lazos irregulares existentes en las rutinas de la librería *SparsKit*. Estos costes, expresados en tiempo de ejecución por iteración, están comprendidos en nuestro código de prueba para valores de W en el rango de 5 a 30 unidades.

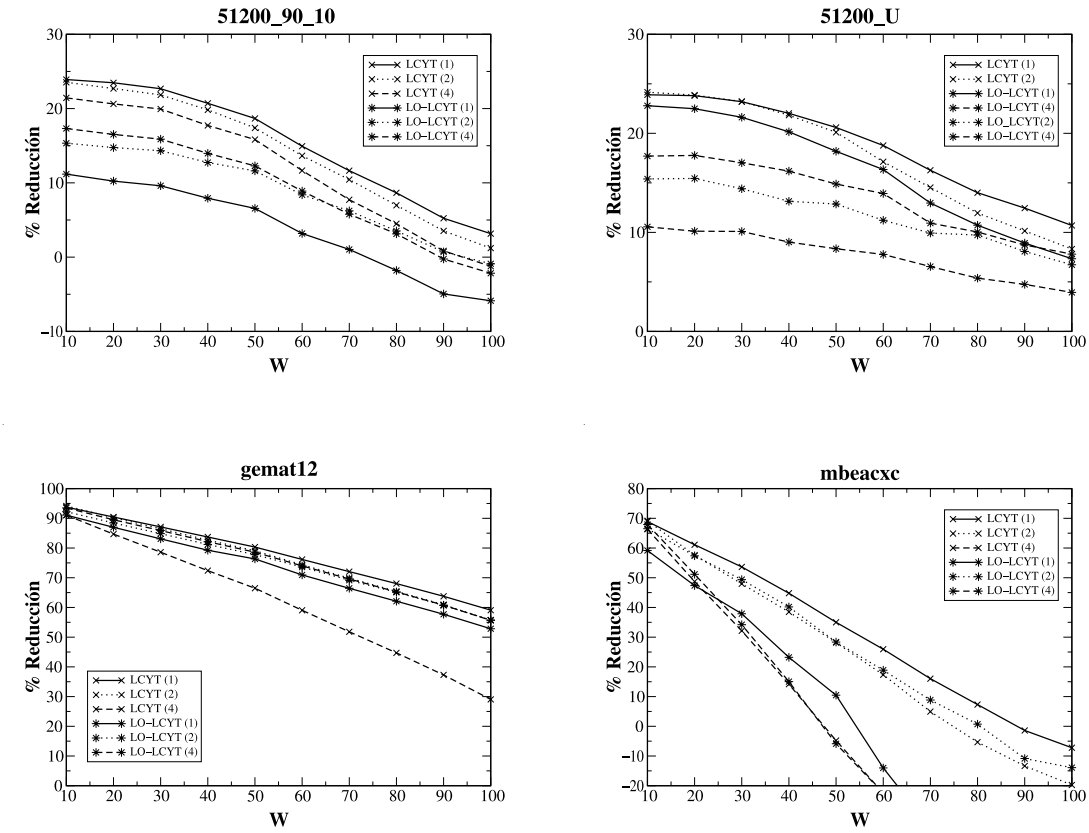


Figura 5.10: Porcentaje de reducción en el tiempo de ejecución para 8 procesadores.

Resultados experimentales

La Figura 5.10 muestra el porcentaje de reducción en el tiempo de ejecución de las propuestas LCYT y LO-LCYT respecto a la técnica CYT. En esta figura denotamos entre paréntesis el número líneas cache asignadas a cada nodo. Por ejemplo, la serie etiquetada como LCYT(4) indica que $N_{mult} = 4$. Analizando los resultados obtenidos, se puede apreciar que el rendimiento de la técnica LCYT aumenta conforme más exacta es la representación del patrón de acceso, es decir, cuanto menor sea el parámetro N_{mult}, obteniendo el mejor rendimiento para $N_{mult} = 1$. Para el caso del algoritmo LO-LCYT, los mejores resultados se obtienen para un tamaño de bloque equivalente a dos líneas cache, es decir, para $N_{mult} = 2$. Para valores más pequeños, la pérdida en la localidad hace que el código paralelo sea más ineficiente, mientras que para valores superiores, el balanceo de la carga es el principal

factor que limita el rendimiento.

Desde el punto de vista de la precisión de ambas propuestas, el algoritmo `LCYT` analiza todos los accesos del patrón de indirección, mientras que la técnica `LO-LCYT` únicamente considera los accesos de escritura. Desde esta perspectiva, el análisis del algoritmo `LCYT` es mucho más preciso, lo que justifica los mejores tiempos de ejecución alcanzados mediante esta propuesta.

Consideraremos el grado de mejora obtenido respecto al algoritmo `CYT`. Las mayores reducciones en los tiempo de ejecución se obtienen para lazos con menor coste por iteración (valores pequeños de W). Debido al reducido coste del cuerpo del lazo, el modo de indireccionamento tiene un mayor impacto sobre la eficiencia del código. Conforme W aumenta, este coste tiene una menor repercusión, pasando a ser el balanceo de la carga y el tiempo de espera asociado a las operaciones de sincronización los factores determinantes en el rendimiento del lazo. Para cuantificar el grado de balanceo de carga de nuestro lazo irregular, introducimos la magnitud r.

$$r = \frac{N_x}{N_p * It_{max}} \tag{5.4}$$

Donde It_{max} representa el máximo número de iteraciones asignadas a un procesador. De este modo tenemos que $0 < r \leq 1$, alcanzando para un valor igual a la unidad el balanceo de carga óptimo. La Tabla 5.2 muestra el valor de r para los algoritmos `LCYT` y `LO-LCYT` con $N_p = 8$. Dado que el algoritmo `CYT` realiza la distribución de las iteraciones de forma cíclica, el balanceo de la carga es muy próximo a la unidad, motivo por el cual no fue

Matriz	`LCYT`(N_L)			`LO-LCYT`(N_{mult})		
	N_L=1	N_L=2	N_L=4	N_{mult}=1	N_{mult}=2	N_{mult}=4
gemat1	.971	.912	.854	.948	.941	.920
gemat12	.985	.918	.822	.956	.953	.922
mbeacxc	.639	.550	.331	.555	.643	.336
beaflw	.582	.553	.332	.568	.644	.338
psmigr_2	.919	.853	.768	.797	.711	.699
25600_U	.997	.988	.956	.972	.973	.958
25600_90_10	.982	.981	.947	.913	.925	.926
51200_U	.997	.992	.977	.968	.983	.975
51200_90_10	.998	.983	.985	.909	.963	.953

Tabla 5.2: Balanceo de carga de las estrategias `LCYT` y `LO-LCYT`.

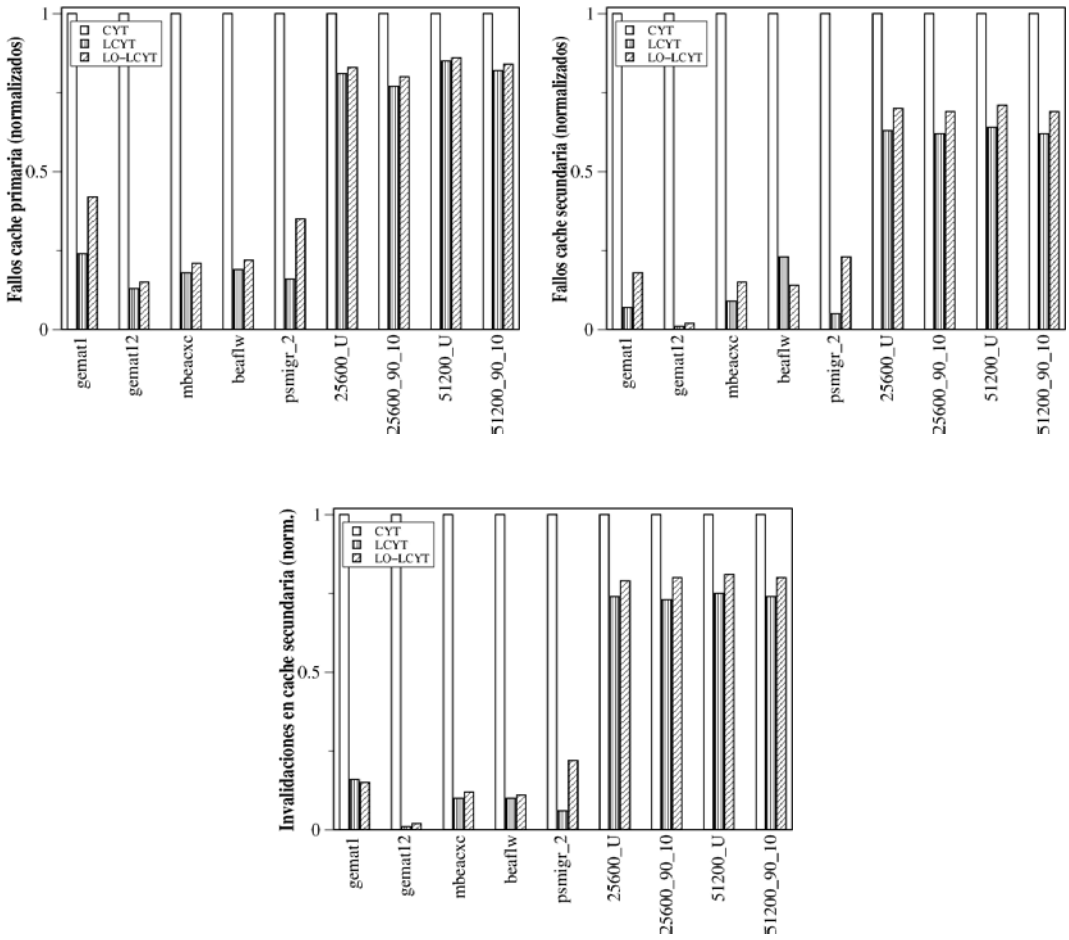


Figura 5.11: Rendimiento a nivel de memoria cache de las tres estrategias.

incluido en la tabla. Analizando el balanceo obtenido para el resto de las propuestas, se puede apreciar que, en general, el balanceo de carga de los algoritmos LCYT y LO-LCYT alcanza unos valores próximos al óptimo. La única excepción son aquellas matrices con un valor de N_a reducido, como, por ejemplo, la *mbeacxc* y la $beaflw$. Para estos casos, el número de nodos del grafo es reducido, lo que dificulta la obtención de una distribución óptima de a.

La reducción en los tiempos de ejecución mostrada en la Figura 5.10 es una consecuencia directa de la mejora en la localidad en los datos de nuestras propuestas. Con el fin de evaluar el grado de explotación de la jerarquía de memoria de cada una de ellas, hemos utilizado los contadores de eventos del MIPS R10000 [153] para medir, en el ejecutor paralelo, los fallos de memoria cache primaria y secundaria, así como el número de inva-

lidaciones de líneas de cache secundaria. La Figura 5.11 muestra los resultados de estas medidas para el algoritmo `LCYT` con $N_{mult} = 1$ y para el algoritmo `LO-LCYT` con $N_{mult} = 2$. Estas medidas están normalizadas respecto al algoritmo `CYT` y se corresponden con una ejecución utilizando 8 procesadores. Se puede apreciar como la reducción en el número de fallos cache y de invalidaciones es muy significativa en todos los casos. Esta mejora es mayor para las matrices de la Harwell-Boeing, debido a que $N_a \ll N_x$ y, consiguientemente, la probabilidad de reuso es mayor. Para la matriz *gemat*12 se obtienen los mejores resultados para todos los casos, y se alcanzan las mayores reducciones en el tiempo de ejecución, demostrando la importancia de la jerarquía de memoria en el rendimiento del ejecutor paralelo.

La Figura 5.12 muestra las aceleraciones obtenidas por nuestro ejecutor para los algoritmos `CYT`, `LCYT` con $N_{mult} = 1$ y `LO-LCYT` con $N_{mult} = 2$. Cada una de las gráficas se corresponden con un valor distinto de carga de trabajo y todas ellas fueron realizadas utilizando 8 procesadores. En general, el algoritmo `LCYT` obtiene los mejores resultados, mientras que el algoritmo `LO-LCYT` obtiene resultados aceptables en la práctica totalidad de los casos. Nuevamente podemos apreciar que ambas propuestas alcanzan mejores aceleraciones cuando la carga de trabajo del lazo es pequeña.

Debido a la existencia de dependencias de datos, la máxima aceleración que podemos extraer está limitada por el grado de paralelismo existente en el lazo. De este modo, es de esperar que la longitud de camino crítico influya de manera significativa en el rendimiento de todas las estrategias. Sin embargo, esta influencia no tiene por qué ser la única. Otro factor importante es el modo en que se distribuyen las iteraciones entre los procesadores (o equivalentemente, el modo en que las iteraciones están asociadas a uno u otro bloque de a). La elección del esquema de distribución determina el número de sincronizaciones que debe realizar cada procesador y, por lo tanto, repercute fuertemente sobre el rendimiento del programa. Considerando los resultados obtenidos, la estrategia `CYT` presenta un mejor rendimiento para las matrices sintéticas, que son las que tienen asociado un menor camino crítico. Del mismo modo, esta estrategia muestra un peor comportamiento para la matriz *gemat*1, con la que el camino crítico es mayor. Finalmente, hay que destacar que las aceleraciones aumentan conforme lo hace la carga de trabajo del lazo irregular, ya que conforme aumenta el parámetro W, el coste asociado a las rutinas de paralelización tiene un menor peso en el tiempo total de ejecución.

Un nuevo factor que es necesario considerar es el coste temporal del inspector para cada estrategia. Típicamente, en aquellos códigos que contienen lazos irregulares parcialmente paralelos, estos lazos son ejecutados en procesos iterativos en los que el patrón de acceso no sufre modificaciones. En este tipo de casos, la información del inspector puede ser

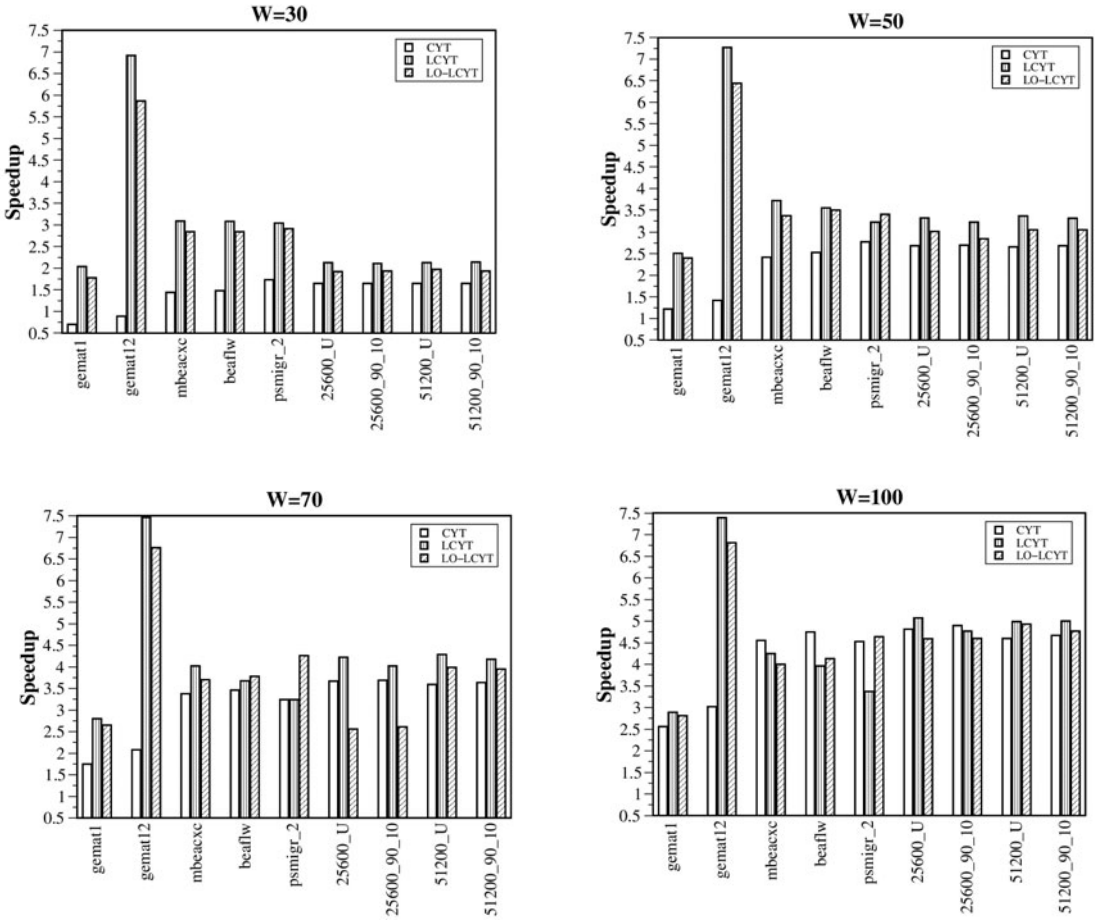


Figura 5.12: Aceleraciones obtenidas con 8 procesadores para diferentes cargas de trabajo.

reutilizada, disminuyendo su coste en la ejecución del programa paralelo. Un ejemplo de esta situación pueden ser los de resolución de sistemas lineales dispersos. Si N_{it} es el número de veces que el código es ejecutado, y asumiendo un reuso del inspector, el tiempo total de ejecución (t_{tot}), es:

$$t_{tot}(N_{it}) = T_{insp} + N_{it} * T_{ejec} \tag{5.5}$$

Donde T_{insp} y T_{ejec} son, respectivamente, el tiempo de ejecución del inspector y del ejecutor. En nuestro caso, hemos obtenido el valor de t_{tot} para un rango comprendido entre 1 y 100 iteraciones. La Figura 5.13 muestra el tiempo de ejecución total para las estrategias `CYT`, `LCYT` con $N_{mult} = 1$ y `LO-LCYT` con $N_{mult} = 2$. Analizando los resultados de la

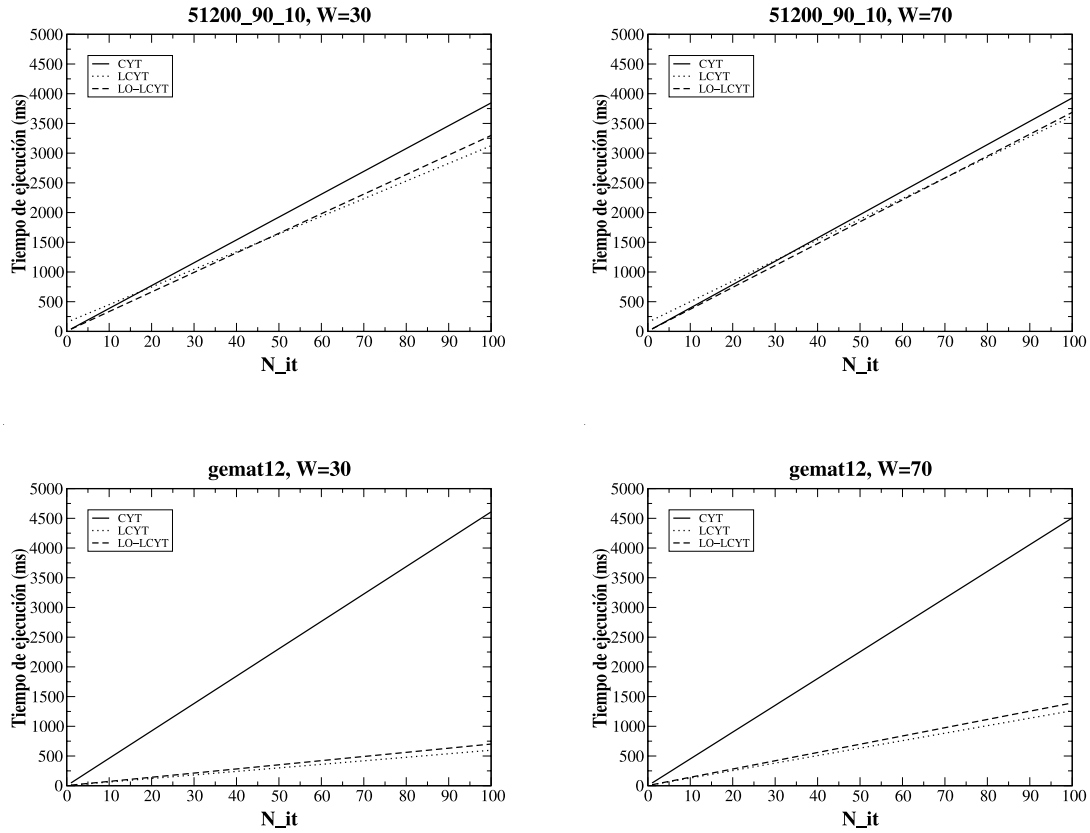


Figura 5.13: Tiempo total de ejecución para 8 procesadores.

Figura 5.13, el tiempo total de ejecución de los algoritmos `LCYT` y `LO-LCYT` son siempre menores que los del algoritmo `CYT` para las matrices de la librería Harwell-Boeing. Como es de esperar, esta diferencia aumenta conforme el coste del lazo disminuye. Hemos utilizado las matrices *gemat*12 y 51200_U como ejemplos ilustrativos de patrones de acceso reales y sintéticos. Se puede apreciar que el algoritmo `LCYT` obtiene los mejores resultados para la matriz *gemat*12 para todas las situaciones debido a que el número de entradas de a (y consiguientemente el número de nodos) es reducido. Esto hace que el grafo asociado al patrón de acceso sea pequeño y que el coste del proceso de inspección (especialmente el coste del particionamiento) no resulte elevado. Respecto a las matrices sintéticas, y dado que N_a es mucho mayor, el coste del inspector del algoritmo `LCYT` es más importante, por lo que únicamente supera al `CYT` cuando existe reuso del ejecutor. Específicamente, para un $W = 30$ son necesarias 20 iteraciones, mientras que para $W = 70$ este valor aumenta

a 26. Bajo este umbral, la estrategia más eficiente es LO-LCYT.

La Tabla 5.3 muestra el impacto del coste del inspector, para diferentes valores de W, con todas las matrices que hemos utilizado. Esta tabla contiene el número de iteraciones (N_{it}) necesarias para que el tiempo de ejecución total de los algoritmos LCYT y LO-LCYT sea menor que el existente para el CYT. En el caso de que $N_{it} = 1$, el tiempo de ejecución es siempre menor y no es necesario reutilizar el inspector. El valor de N_{it} igual a la unidad se alcanza para todas las matrices de la Harwell–Boeing, y con la estrategia LO-LCYT, para todas las matrices sintéticas excepto si $W = 70$. Una entrada representada por "-" significa que el tiempo total de ejecución de la propuesta nunca es inferior al del CYT. Esta situación únicamente se da para la matriz *psmigr_2* con la estrategia LO-LCYT para $W \geq 70$.

Los números representados entre paréntesis en la tabla representan el número de iteraciones para el que la estrategia LCYT supera a la LO-LCYT. Observando los resultados obtenidos, apreciamos que la estrategia LCYT es superior a LO-LCYT para la práctica totalidad de patrones de acceso reales, mientras que para las matrices sintéticas, es necesario reusar el inspector. Esto es debido a que para esta familia de matrices se tiene que $N_a = 2N_x$ por lo que el coste de procesamiento del inspector LCYT es importante.

Del mismo modo que en la estrategia CYT, nuestras propuestas tienen como inconveniente la serialización de los accesos de lectura después de una lectura, los cuales no originan dependencias. En [150] Xu y Chaudhary proponen una mejora de la estrategia CYT en la que estas dependencias no son serializadas. A pesar de que en este trabajo no

Matriz	$W = 30$		$W = 50$		$W = 70$	
	LCYT	LO-LCYT	LCYT	LO-LCYT	LCYT	LO-LCYT
gemat1	1 (2)	1	1 (1)	1	1 (2)	1
gemat12	1 (1)	1	1 (1)	1	1 (1)	1
mbeacxc	1 (1)	1	1 (1)	1	1 (1)	1
beaflw	1 (1)	1	1 (1)	1	1 (1)	2
psmigr_2	1 (1)	1	1 (-)	1	- (-)	1
25600_U	18 (49)	1	21 (49)	1	31 (56)	2
25600_90	17 (48)	1	23 (56)	1	46 (75)	3
51200_U	20 (49)	1	22 (56)	1	26 (64)	1
51200_90	18 (47)	1	22 (56)	1	33 (71)	2

Tabla 5.3: Umbral de iteraciones para superar la estrategia CYT.

hemos considerado la adaptación de nuestras propuestas a esta nueva estrategia, dicha operación puede ser realizada de un modo inmediato, permitiendo superar esta limitación y aumentar la eficiencia de las estrategias `LCYT` y `LO-LCYT`.

5.5 Uso del IARD para la mejora del rendimiento

En esta sección evaluamos la capacidad de la representación IARD para caracterizar el tipo de lazos discutido en este capítulo. Para los códigos irregulares que tienen una única indirección, la traza de acceso a memoria coincide con los valores almacenados en la misma. En estos casos, el procedimiento consiste en analizar el contenido de la indirección, caracterizar el patrón de acceso y ejecutar, con la información generada en el proceso de análisis, el lazo paralelo.

Cuando existe más de un vector de indirección el procedimiento anterior deja de ser válido, dado que, como veremos a continuación, no es posible combinar la información de dos representaciones IARD para obtener el patrón de acceso del programa. Comenzamos nuestro estudio con el análisis del código de la Figura 5.1(a) que muestra un lazo irregular con dos indirecciones. Como se vio previamente, en función de los valores de x_1 y x_2 se pueden originar cualquiera de los tres tipos de dependencias de datos. Mediante una caracterización IARD independiente de los vectores x_1 y x_2, y dado que no se almacenan los valores concretos de las entradas de estos vectores, no es posible determinar si la entrada $x_1[j]$ coincide con la $x_2[j']$ para unas iteraciones j y j' dadas.

La única posibilidad de análisis de este tipo de lazo es la generación de una caracterización IARD conjunta de ambas indirecciones. En la siguiente sección profundizamos en este campo mediante el desarrollo de una propuesta de caracterización IARD para un lazo con dos o más indirecciones. Vamos a suponer que las lecturas generan las mismas dependencias que las escrituras, y por lo tanto, el tipo concreto de acceso de cada familia de estamentos es irrelevante dentro de nuestro análisis.

5.5.1 Representación IARD de dos indirecciones

Para este tipo de lazos hemos desarrollado una nueva técnica de paralelización basada en el concepto de *slice*. La esencia de este concepto radica en que un *slice* agrupa un conjunto de entradas del vector de indirección para los que no se originan dependencias.

En nuestro caso pretendemos extender la definición de *slice* a lazos con dos o más indirecciones. Para este caso, un *slice* contendrá conjuntos de entradas contiguas de cada

indirección que no originan dependencias. A modo de ejemplo, vamos a considerar el código de la Figura 5.1(a) en el cual tenemos dos indirecciones. Dado que inicialmente estamos considerando lecturas serializadas, imponemos como condición que todos los accesos a memoria, dentro de cada *slice*, se realicen sobre distintas entradas. Para este caso, cada *slice* contiene el siguiente conjunto de entradas:

$$S_i = \{[j_i^{x_1}, j_{i+1}^{x_1}), [j_i^{x_2}, j_{i+1}^{x_2})\}, \qquad 1 \leq i \leq N_S \tag{5.6}$$

Tal que,

$$\begin{gathered} \forall\ j^{x_1}, k^{x_1}, j^{x_2}, k^{x_2} \in S_i,\ \ j^{x_1} \neq k^{x_1}\ \ \&\ \ j^{x_2} \neq k^{x_2}, \\ x_1[j^{x_1}] \neq x_1[k^{x_1}]\ \ \&\ \ x_1[j^{x_1}] \neq x_2[k^{x_2}]\ \ \&\ \ x_2[j^{x_2}] \neq x_2[k^{x_2}] \end{gathered} \tag{5.7}$$

De acuerdo con esta definición, para cada *slice* se puede ejecutar cualquier estamento, tanto asociado al vector x_1 como al vector x_2, sin que existan dependencias de datos. Adicionalmente, imponemos la siguiente condición:

$$\begin{gathered} \exists\ \{j^{x_1}, j^{x_2}\} \in S_i,\ \{k^{x_1}, k^{x_2}\} \in S_{i-1}\ \ /\ \ x_1[j^{x_1}] = x_1[k^{x_1}]\ \ |\ \ x_1[j^{x_1}] = x_2[k^{x_2}]\ \ | \\ x_2[j^{x_2}] = x_2[k^{x_2}]\ \ |\ \ x_2[j^{x_2}] = x_1[k^{x_1}] \quad \forall i \in (1, N_S] \end{gathered} \tag{5.8}$$

Donde "|" es el operador `OR`; por lo que de las cuatro igualdades de la Ecuación 5.8 al menos una de ellas se verifica en cada *slice*.

Para cada vector de indirección x_k, definimos el vector densidad ρ_k, como el que contiene en su entrada k-ésima el número de entradas de x_k asociadas al k-ésimo slice. A partir de este vector, se puede producir el vector de densidad acumulada del k-ésimo estamento, ρ_k^{ini}, definido como:

$$\rho_k^{ini}[s] = 1 + \sum_{i=1}^{s-1} \rho_k[i] \tag{5.9}$$

Una vez generalizado el concepto de *slice*, el siguiente paso es el desarrollo de un algoritmo que permita clasificar las entradas de los vectores de indirección. Este algoritmo, al que hemos denominado `CS3`, puede derivarse adaptando el algoritmo clásico del IARD (algoritmo `CS` descrito en la Sección 2.3). El único cambio realizado consiste en que ahora el algoritmo contempla los accesos a memoria de todos los vectores de indirección. La Figura 5.14 muestra el pseudocódigo de nuestra propuesta.

Dado que no establecemos ninguna distinción entre las operaciones de lectura y escritura, no es necesario clasificar los accesos de modo diferente. Cuando una de estas operaciones se repite (existe "solape" entre los accesos) se pasa a aumentar el número de

```
Algoritmo CS3
entrada
        {x_1, x_2, ... x_{N_stmt}}        vectores de indirección
salida
        {u, l, , ρ_1, ρ_2 ... ρ_{N_stmt}}        representación por slices
inicio del algoritmo
        s = 1
        flag[1 : N_a] = 0
        buffer = ∅
        DO j = 1, N_x
            DO k = 1, N_stmt
                IF (a[x_k[j]] = s)
                    {u[s], l[s], ρ_1[s], ρ_2[s] ... ρ_{N_stmt}[s]} ⟵ caracteriza_slice(buffer)
                    buffer = ∅
                    s = s + 1
                END IF
                buffer ⟵ x_k[j]
                flag[x_k[j]] = s
            END DO
        END DO
fin del algoritmo
```

Figura 5.14: Pseudocódigo del algoritmo de clasificación por *slices* de varias indirecciones (CS3).

slice. Previamente, la función *caracteriza_slice*, introducida en la Sección 2.3.1, devuelve la entrada máxima y mínima accedida en el *slice* procesado y el número de entradas de cada uno de los vectores de indirección. A continuación, del mismo modo que el algoritmo CS, se realiza un aumento de *slice* y se repite el proceso anterior.

El resultado devuelto por este algoritmo consiste en el conjunto de elementos $\{u, l, \rho_1, \rho_2 \ldots \rho_{N_{stmt}}\}$. Aplicando el algoritmo EH (introducido en la Sección 2.4.1) a los vector u y l obtenemos la representación IARD del lazo irregular con N_{stmt} indirecciones. Esta representación estará compuesta por el conjunto $\{\mathcal{E}^u, \mathcal{E}^l, \rho_1, \rho_2 \ldots \rho_{N_{stmt}}\}$ de $2N_{\mathcal{E}}+N_{stmt}N_S$ elementos.

```
SHARED: ρ^ini, ρ^ini − 1, a, x1, x2

DO s = 1, N_S
    DOALL k = 1, N_stmt
        DOALL j = ρ_k^ini[s], ρ_k^ini[s + 1] − 1
            ejecuta: stmt_j^k
        END DOALL
    END DOALL
  BARRIER
END DO
```

Figura 5.15: Ejecutor paralelo para una representación por *slices* de dos indirecciones.

5.5.2 Ejecutor paralelo

Asumamos que los vectores de indirección se agrupan en *slices* siguiendo el criterio anterior. Entonces, y de acuerdo con su definición, todas las entradas de un mismo *slice* se pueden ejecutar en paralelo, y como no guardan ningún tipo de dependencia, el orden en que son ejecutadas puede ser totalmente arbitrario. De este modo, el ejecutor paralelo puede escribirse tal y como se muestra en la Figura 5.15. En el ejecutor los lazos internos son totalmente paralelos, mientras que el lazo más externo es el que introduce las sincronizaciones. Nótese que después de procesar cada uno de los *slices* siempre es necesario ejecutar una operación de sincronización. En una versión paralela más optimizada, y dado que todas las entradas son independientes entre si, los dos lazos internos paralelos se pueden fusionar en uno, reduciendo el coste asociado al control del lazo.

5.5.3 Análisis de eficiencia

Hemos evaluado nuestra propuesta con una muestra de patrones de acceso reales y sintéticos. Como ejemplo de patrones de acceso reales, hemos utilizado los asociados a las matrices *bcsstk14*, *beaflw* y *psmigr_2*. Por otra parte, como patrones de acceso sintéticos utilizamos las matrices *25600_90* y *25600_U*. Las matrices están originalmente almacenadas en formato CCS, de modo que el vector $fila$, con N_x entradas, fue el empleado para generar los vectores de indirección. El algoritmo que hemos utilizado para producir dichos vectores de indirección se muestra en la Figura 5.16. El lazo más externo recorre todos los elementos del vector N_x, y mediante el lazo interno sus entradas se distribuyen cíclicamente entre los N_{stmt} vectores de indirección. En nuestro caso, y dado que hemos utilizado el código

de prueba mostrado en la Figura 5.1(a), tenemos que $N_{stmt} = 2$.

El número de *slices* obtenido para cada una de las matrices se muestra en la Tabla 5.4. En general, este número es elevado para todas las matrices. Cabe destacar especialmente el caso de las matrices sintéticas, para las que la matriz *25600_90* triplica el número de *slices* de la matriz *25600_U*. Nuevamente podemos ver la influencia de la presencia de *hot spots* en la cantidad de paralelismo extraído.

La Figura 5.17 muestra el patrón de acceso y el número de entradas ocupadas en cada *slice* para las matrices *bcsstk14*, *beaflw* y *25600_90*. Podemos apreciar cómo el número de entradas dentro de cada *slice* es muy reducido. Este hecho hace que el código paralelo sea muy poco eficiente, ya que la carga computacional asociada al procesamiento de cada *slice* es pequeña y comparable al coste de la operación de sincronización. Adicionalmente, la localidad en los accesos al vector a es muy pobre, dado que, hasta el momento no se ha establecido ningún criterio de distribución de los estamentos.

Debido a los pobres resultados obtenidos en las pruebas realizadas, hemos decidido no incluir los tiempos de ejecución y aceleraciones conseguidas mediante el ejecutor paralelo. Sin embargo, consideramos interesante incluir los resultados mostrados en términos de número de *slices*, ya que servirán como comparativa con los obtenidos mediante otras propuestas. El criterio empleado por el algoritmo `CS3` es muy restrictivo, ya que clasifica las entradas pertenecientes a cada *slice* de acuerdo con el orden en que son procesadas por el algoritmo secuencial. Es decir, aunque realiza una paralelización a nivel de estamento, no permite la reordenación de los mismos. En la siguiente sección se introducen dos propuestas que intentan aumentar la eficiencia de esta técnica. Como veremos posteriormente, la propuesta más eficiente reordena dinámicamente los vectores de indirección, permitiendo cambiar el orden de ejecución de los estamentos que conforman el lazo irregular.

Matriz	*bcsstk14*	*beaflw*	*psmigr_2*	*25600_90*	*25600_U*
Número de *slices*	1666	499	3035	383	127

Tabla 5.4: Eficiencia de la clasificación por *slices* para $N_{stmt} = 2$ con distintos patrones de acceso.

```
Algoritmo GENERADOR
entrada
        row: vector de almacenamiento de la matriz
salida
        {x_1, x_2, ... x_{N_stmt}}: vectores de indirección
inicio del algoritmo
      DO k = 1, N_stmt
         tmp = 1
         DO j = 1, N_x
            IF (mod(j, N_stmt) = k - 1)
               x_k[tmp] = row[j]
            END IF
         END DO
         tmp++
      END DO
fin del algoritmo
```

Figura 5.16: Algoritmo de generación de indirecciones.

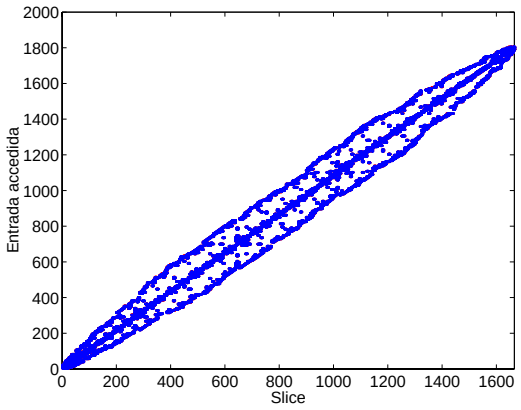
5.6 Algoritmos OWNCR y SLCSRT

Vamos a abordar la paralelización del lazo de la Figura 5.1(c) asumiendo que todas las familias de estamentos contienen accesos irregulares sobre la matriz a. Nuevamente, el tipo de acceso realizado en cada una de las familias es arbitrario, pudiendo ser operaciones de lectura, de escritura, reducciones irregulares, operaciones de acumulación, etc.

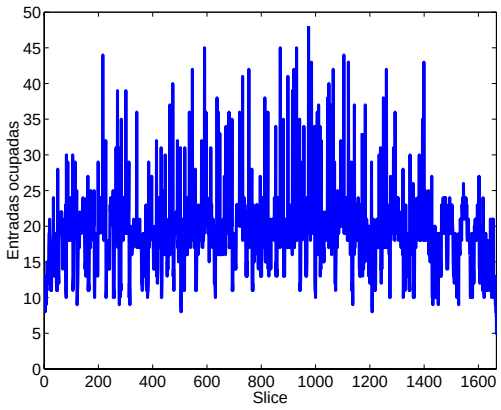
El objetivo de nuestras propuestas va a consistir en aumentar el paralelismo modificando el orden de ejecución de los estamentos. Para poder realizar esta operación, es necesario imponer dos restricciones a la estructura del lazo irregular:

1. Cada estamento accede sobre una única entrada de la matriz a, aunque es arbitrario el número y tipo de accesos que cada estamento realiza sobre la misma. Un ejemplo de estamento con más de un acceso sobre la misma posición de memoria es la reducción irregular, para las que existen operaciones tanto de lectura como de escritura.

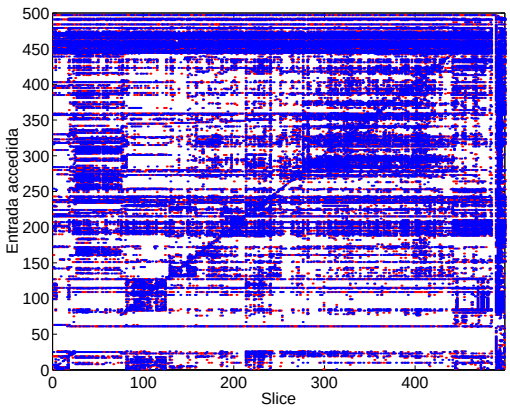
2. Entre estamentos distintos, las únicas dependencias de datos que pueden existir son las asociadas a los accesos sobre la matriz a.



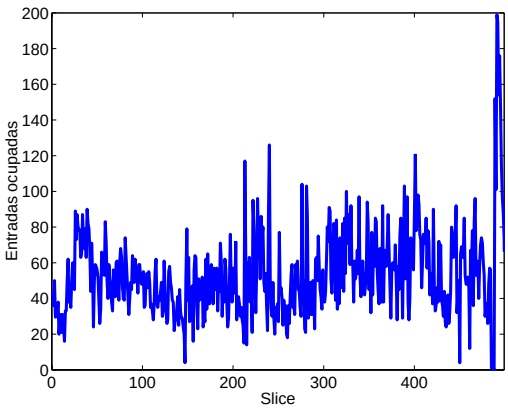
Patrón de la matriz *bcsstk14*



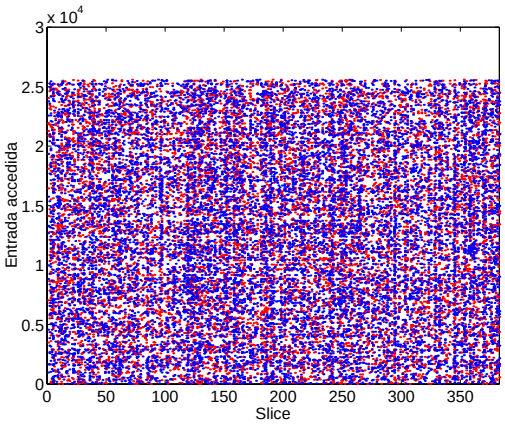
Nivel de ocupación de la matriz *bcsstk14*



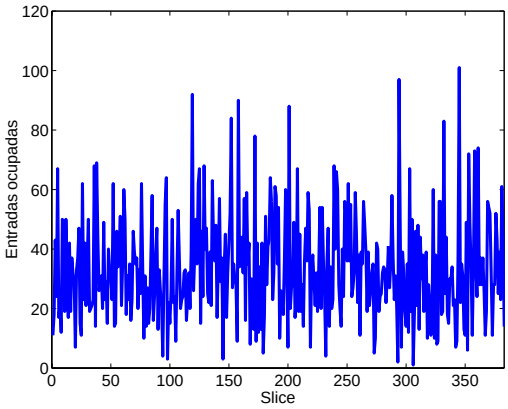
Patrón de la matriz *beaflw*



Nivel de ocupación de la matriz *beaflw*



Patrón de la matriz *25600_90*



Nivel de ocupación de la matriz *25600_90*

Figura 5.17: Resultado del algoritmo de clasificación por *slices*.

```
SHARED: x_1, x_2, ... x_{N_stmt}

DOALL p = 1, N_pt
   DO j = 1, N_x
      IF(x_1[j] ∈ A_p)          ejecuta: stmt_j^1
      IF(x_2[j] ∈ A_p)          ejecuta: stmt_j^2
        ...
      IF(x_{N_stmt}[j] ∈ A_p)   ejecuta: stmt_j^{N_stmt}
 END DO
END DOALL
```

Figura 5.18: Código paralelo para la regla del propietario.

Bajo estas restricciones, el dominio de aplicación de nuestras nuevas propuestas es algo más limitado que el existente para las técnicas introducidas en la Sección 5.4. Para aquellos lazos irregulares que cumplen las dos restricciones planteadas, podemos formular la siguiente propiedad.

Propiedad 5.6.1 Dados dos estamentos $stmt_j^k$ y $stmt_{j'}^{k'}$ que realizan un acceso sobre la misma entrada de a. Si no existe ningún estamento intermedio que accede a esta entrada, y $j' > j$, entonces el estamento $stmt_{j'}^{k'}$ puede ser ejecutado en cualquier instante posterior al estamento $stmt_j^k$. □

Esta propiedad nos permite establecer un criterio para modificar el orden de ejecución de los estamentos del lazo sin violar ningún tipo de dependencia. En las siguientes secciones se describen dos propuestas que explotan dicha propiedad.

5.6.1 Estrategia OWNCR

Como punto de partida, proponemos un ejecutor paralelo basado únicamente en la aplicación de la regla del propietario mediante rutinas condicionales. El código paralelo correspondiente al ejemplo de la Figura 5.1(c), se ilustra en la Figura 5.18. Denominamos a esta propuesta estrategia *owner compute rule* (OWNCR).

Mediante el empleo de la estrategia OWNCR, la matriz a ha sido dividida en N_{pt} bloques contiguos que representamos como $\mathcal{A}_i$, $1 \leq i \leq N_{pt}$. En la ejecución paralela del código, cada procesador recorre todos los estamentos del lazo original, comprobando, mediante

una rutina condicional, si el acceso sobre el vector a se realiza dentro del bloque que tiene asignado. Dado que todos los estamentos son analizados (y ejecutados) en el mismo orden que el lazo original, las posibles dependencias de datos son preservadas. La principal ventaja de esta propuesta es la explotación de la localidad en los accesos al vector a, ya que cada procesador sólo accede sobre la partición que tiene asignada.

Como contrapartida, esta propuesta tiene el inconveniente de que cada procesador tiene que recorrer todo el espacio de iteraciones y analizar, para cada una de ellas, el contenido completo del cuerpo del lazo. Este factor, sumado al coste introducido por la rutinas de comprobación, limita de forma importante su escalabilidad.

El hecho de intentar superar estas causas de ineficiencia nos ha motivado al desarrollo de otra estrategia para la paralelización de esta clase de lazos irregulares. Como punto de partida, hemos establecido los siguientes requisitos:

- Debe explotarse del modo más eficiente la jerarquía de memoria del sistema, dado que está probada su gran repercusión en el rendimiento del código paralelo.
- Es necesario reducir el coste computacional del ejecutor. Particularmente, es necesario eliminar la existencia de rutinas condicionales y evitar hacer que cada procesador analice todos los estamentos del lazo.
- Es necesario especificar un criterio que permita realizar un balanceo de la carga computacional en función de las características del patrón de acceso.

En la siguiente sección presentamos una nueva técnica, denominada *Slice Sort*, que se adapta a estos requisitos.

5.6.2 Estrategia `SLCSRT`

Esta nueva propuesta utiliza la estrategia basada en el inspector-ejecutor. De este modo, tenemos un inspector que realiza el análisis del patrón de acceso del lazo y determina la mejor distribución de los datos y de las computaciones. Posteriormente, un ejecutor utiliza esta información para ejecutar en paralelo el lazo irregular. Denominamos a esta propuesta *Slice Sort* (`SLCSRT`).

El criterio de distribución de los datos que empleamos se basa en la regla del propietario. Del mismo modo que en el algoritmo `OWNCR`, mediante este criterio conseguimos que los accesos sobre a presenten una alta localidad. Con el fin de extender esta localidad

sobre los accesos a los vectores de indirección, la rutina de inspección realiza una reordenación de los mismos de modo que cada procesador accede, durante la ejecución paralela del lazo, a elementos consecutivos de los mismos. La reordenación de las indirecciones tiene dos ventajas adicionales: por una parte, evita utilizar rutinas condicionales, y por otra, permite que cada procesador procese únicamente aquellos estamentos que acceden sobre el bloque de a que tiene asignado.

Inspector

El objetivo del inspector propuesto es la clasificación de los estamentos ejecutados por cada procesador de modo que no se violen las posibles dependencias de datos. El mecanismo de clasificación utilizado es por medio de *slices*. En secciones previas este esquema de clasificación fue aplicado sobre el patrón de acceso a memoria producido por una indirección (algoritmo `CS`) y por varias indirecciones (algoritmo `CS3`), considerando en este último caso que los diferentes elementos se tenían que ejecutar en orden. En esta sección generalizaremos el modelo de modo que se permite una ejecución fuera de orden de los estamentos.

El algoritmo básico de nuestra propuesta se muestra en la Figura 5.19. Este algoritmo es totalmente paralelo y consta de tres fases denominadas *fase de análisis*, *fase de desplazamiento* y *fase de reordenación*. A lo largo de la descripción del funcionamiento de cada una de estas fases, utilizaremos como ejemplo el código de la Figura 5.1(a) con $N_x = 5$, $N_a = 6$, $x_1 = \{5, 3, 1, 4, 5\}$ y $x_2 = \{3, 4, 1, 6, 2\}$.

1. **Fase de análisis.** En esta fase cada procesador analiza todas las entradas de cada uno de los vectores de indirección, y determina cuales acceden a la partición $\mathcal{A}_p$ que tiene asignado. Mediante la matriz ρ_k^{fila} se almacena el histograma de accesos que la k-ésima familia de estamentos realizan sobre la partición considerada. Este histograma nos permite conocer el número de accesos existentes en cada entrada de a, y, consiguientemente, el *slice* al que cada estamento es asignado.

   Para un estamento dado, el menor *slice* al que puede pertenecer está limitado por la Propiedad 5.6.1. Dentro de nuestra propuesta, este valor viene dado por la función *slice*. Denotando por ρ^{fila} al conjunto de todos los ρ_k^{fila} $\forall k$, y dado que es necesario analizar los accesos manteniendo el orden de ejecución del programa, tenemos que:

$$slice(\rho^{fila}, x_k[j]) = \sum_{q=1}^{N_{stmt}} \rho_q^{fila}[x_k[j]] \tag{5.10}$$

```
Algoritmo SLCSRT
entrada
        {x_1, x_2 ... x_{N_stmt}}        vectores de indirección
salida
        {x_1^fin, x_2^fin ... x_{N_stmt}^fin}        vectores de indirección reordenados
        {ρ_1^slc, ρ_2^slc ... ρ_{N_stmt}^slc}        vectores de densidad
inicio del algoritmo
      SHARED:x, ρ^fila, ρ^slc, x^fin
      DOALL p = 1, N_pt
L1          DO j = 1, N_x                                  % Fase de análisis
              DO k = 1, N_stmt
                 IF (x_k[j] ∈ A_p)
                    ρ_k^fila[x_k[j]] + +
                    ρ_k^slc[slice(ρ^fila, x_k[j]), p] + +
                 END IF
              END DO
            END DO

            DO s = N_S, 1                                  % Fase de desplazamiento
              DO k = 1, N_stmt
                 ρ_k^slc[s, p] = C1(x_k, p) + C2(ρ_k^slc, p, s)
              END DO
            END DO

            DO j = N_x, 1                                  % Fase de reordenación
              DO k = N_stmt, 1
                 IF (x_k[j] ∈ A_p)
                    ρ_k^slc[slice(ρ^fila, x_k[j]), p]] − −
                    x_k^fin[ρ_k^slc[slice(ρ^fila, x_k[j]), p]]] = x_k[j]
                    ρ_k^fila[x_k[j]] − −
                 END IF
              END DO
            END DO
      END DOALL
fin del algoritmo
```

Figura 5.19: Pseudocódigo del inspector paralelo del algoritmo SLCSRT.

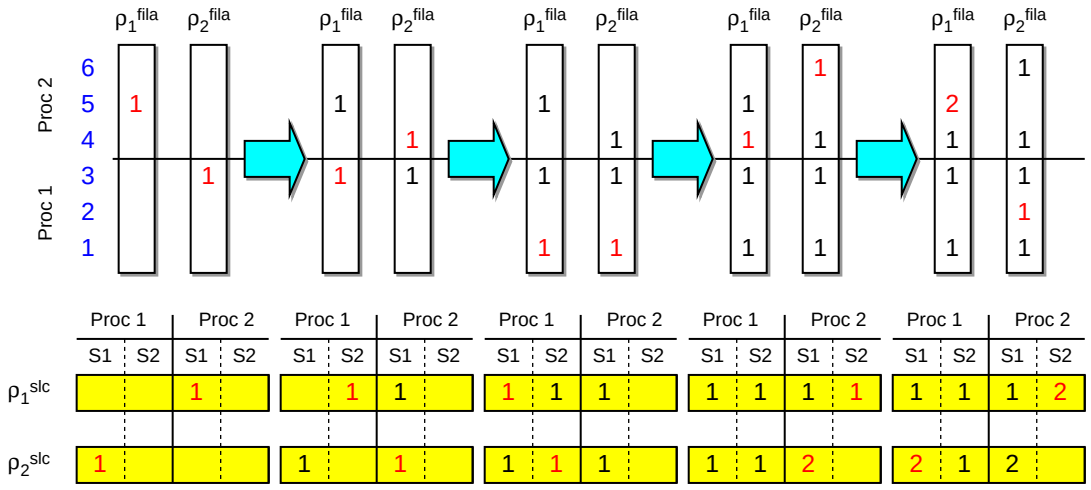


Figura 5.20: Secuencia de valores de los vectores ρ^{slc} y ρ^{fila}.

Una vez conocidos estos valores, los vectores ρ_k^{slc} son convenientemente actualizados. Estos vectores almacenan, para cada procesador, el número de estamentos de la k-ésima familia asignados a cada uno de los *slices*. Nótese que mediante esta estrategia estamos considerando todos los tipos de dependencias, incluidas las lecturas después de lecturas. La Figura 5.20 muestra la secuencia de valores obtenida en los vectores ρ^{slc} y ρ^{fila} conforme se procesa cada iteración del lazo. Se puede apreciar que las entradas de los vectores ρ^{slc} aparecen clasificadas en función del *slice* ($s = 1, 2$) y del procesador ($p = 1, 2$). Estamos asumiendo que $\mathcal{A}_1 = [1, 3]$ y $\mathcal{A}_2 = [4, 6]$.

2. **Fase de desplazamiento.** El propósito de esta fase consiste en modificar el contenido de los vectores ρ_k^{slc}. Inicialmente los vectores ρ_k^{slc} contienen el número de accesos asociados a cada familia de estamentos. Por ejemplo, la entrada $\rho_k^{slc}[s, p]$ almacena el número de accesos realizados por la k-ésima familia de estamentos sobre la partición $\mathcal{A}_p$ asignados al *slice* s.

   Después de realizar la fase de desplazamiento, los vectores ρ_k^{slc} van a contener las posiciones de memoria de los vectores de indirección reordenados. Por ejemplo, la entrada $\rho_k^{slc}[s, p]$ pasa a almacenar el número de entrada del vector x_k asociado al procesador p y al *slice* s.

   El proceso de desplazamiento consiste en reemplazar cada entrada de estos vectores por una nueva cantidad igual a la suma de dos contribuciones. La primera, dada por la función $C1$, es el número de entradas del vector de indirección considerado que están asignadas a los procesadores previos. Más formalmente, y definiendo como

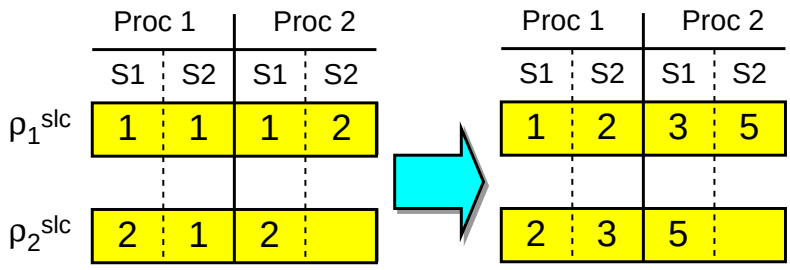


Figura 5.21: Operación de desplazamiento sobre los vectores ρ_k^{slc}.

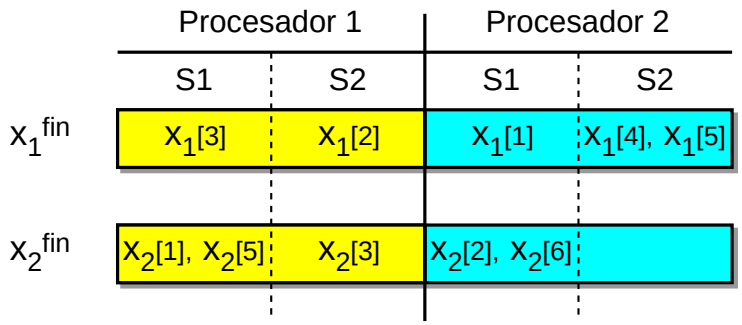


Figura 5.22: Distribución de los vectores de indirección reordenados.

$\| \cdot \|$ el operador cardinalidad, para un vector de indirección x_k y un procesador p tenemos que:

$$C1(x_k, p) = \| x_k[j] \| \quad / \; x_k[j] \quad \in \quad \mathcal{A}_{p'}, \quad \forall p' < p \quad \forall j \in [1, N_x] \tag{5.11}$$

La segunda contribución, denotada como $C2$, es la debida a todas las entradas del vector de indirección en el mismo procesador y pertenecientes a un *slice* anterior.

$$C2(\rho_k^{slc}, p, s) = \sum_{i=1}^{s-1} \rho_k^{slc}[i, p] \tag{5.12}$$

La combinación de ambas cantidades es la nueva posición de memoria. La Figura 5.21 muestra la distribución final de los vectores de densidad, una vez que han sido transformados. En este caso: $C1(x_1, 1) = C1(x_2, 1) = 0$, $C1(x_1, 2) = 2$ y $C1(x_2, 2) = 3$.

3. **Fase de reordenación.** En esta fase se generan los vectores de indirección reordenados. Una vez más, se recorren todas las iteraciones del lazo. Pero esta vez en orden inverso, es decir, empezando por la iteración N_x y finalizando en la primera.

```
SHARED: ρ^slc

DOALL p = 1, N_pt
   DO s = 1, N_S − 1
      DO k = 1, N_stmt
         DO j = ρ_k^slc[s, p], ρ_k^slc[s + 1, p] − 1
            ejecuta: stmt_k^j
         END DO
      END DO
   END DO
END DOALL
```

Figura 5.23: Ejecutor paralelo del algoritmo SLCSRT.

Para cada entrada del vector x_k, la nueva posición en el vector reordenado x_k^{fin} está almacena en el vector ρ_k^{slc}. Cada vez que una entrada en asignada, el vector de densidad es decrementado en una unidad. Con esto se consigue generar los vectores reordenados empezando por el último elemento de cada *slice*. La Figura 5.22 muestra la distribución final de los vectores de indirección, una vez que han sido reordenados.

Ejecutor

La Figura 5.23 muestra el ejecutor paralelo de nuestra propuesta. Gracias a la reordenación de los vectores de indirección, cada procesador ejecuta únicamente aquellos estamentos que acceden sobre la partición de a que tiene asignada accediendo a entradas contiguas de los vectores de indirección. De este modo, se evita el empleo de rutinas condicionales. Mediante el algoritmo SLCSRT, las dependencias de datos son preservadas, asegurando un resultado correcto.

5.6.3 Balanceo de carga

El balanceo de la carga computacional puede ser dinámicamente ajustado cambiando el tamaño de cada partición de a. La carga de trabajo asociada a la entrada $a[i]$ viene

dada por la siguiente expresión:

$$carga[i] = \sum_{k=1}^{N_{stmt}} (C_k \parallel x_k[j] \parallel \ / \ x_k[j] = a[i], \ \forall \ j \in [1, N_x]) \tag{5.13}$$

Donde C_k representa el coste computacional asociado a la k-ésima familia de estamentos. Una vez fijado el tamaño de partición $\mathcal{A}_p$ y mediante la Ecuación 5.13, podemos obtener una estimación de la carga computacional asociada a cada uno de los procesadores mediante la suma de las contribuciones de todas las entradas asociadas a la partición que tiene asignado.

El método de balanceo de carga que hemos utilizado se muestra en la Figura 5.24. Inicialmente (etiqueta `L1`), se realiza una estimación de la carga computacional total del lazo, la cual viene dada por la suma de las contribuciones de todas las entradas de a mediante la Expresión 5.13. Una vez conocido este valor, se obtiene de forma ordenada el tamaño de cada una de las particiones $\mathcal{A}_p$ tales que la carga asociada tenga el valor más próximo por exceso a C_{tot}/N_{pt}. La forma de realizar esto consiste en aumentar el límite superior (variable lim_p^{sup}) de cada partición hasta que su carga asociada supere a C_{tot}/N_{pt}. Una vez alcanzado este valor, se pasa a procesar la siguiente partición. La complejidad máxima de esta propuesta es:

$$\mathcal{O}_{\mathcal{BALAN}} = N_{pt} N_a \tag{5.14}$$

En situaciones reales esta complejidad suele reducirse en un factor N_{pt}, siendo, de este modo, aproximadamente igual a N_a.

5.6.4 Análisis de eficiencia

Hemos evaluado la eficiencia de nuestra propuesta para un gran número de situaciones. Inicialmente, hemos hecho un análisis teórico con el fin de obtener los requisitos de memoria y la complejidad del inspector `SLCSRT`. Posteriormente, hemos comparado de forma empírica el rendimiento del ejecutor con el de otras propuestas altamente competitivas. Las pruebas realizadas incluyen códigos de prueba reales y sintéticos. Finalmente, hemos evaluado el rendimiento global de nuestra propuesta, es decir, la eficiencia del ejecutor teniendo en cuenta el coste computacional del inspector. En las siguientes secciones describimos detalladamente cada uno de estas pruebas.

```
Algoritmo BALANC
entrada
        carga: vectores de carga
salida
        LIM: conjunto de particionamiento
inicio del algoritmo
L1      C_tot = sum_{i=1}^{N_a} carga(i)
        lim_1^inf = 1
        DO p = 1, N_pt
            lim_p^sup = lim_p^inf
            carga_tmp = carga(lim_p^sup)
            WHILE(carga_tmp < C_tot/N_p)
                  lim_p^sup ++
                  carga_tmp += carga(lim_p^sup)
            END WHILE
            IF(p < N_pt)  lim_{p+1}^inf = lim_p^sup
        END DO
fin del algoritmo
```

Figura 5.24: Algoritmo de balanceo de la carga BALANC.

Consideraciones teóricas

La complejidad del inspector SLCSRT puede desglosarse en la suma de las complejidades de cada una de sus tres etapas. Teniendo en cuenta que el lazo más externo es totalmente paralelo, tenemos que,

$$\mathcal{O}_{\texttt{SLCSRT}} = N_x N_{stmt} + N_S N_{stmt} + N_x N_{stmt} \tag{5.15}$$

Típicamente $N_x \gg N_S$ por lo que $\mathcal{O}_{\texttt{SLCSRT}} \simeq 2N_x N_{stmt}$. Es decir, la complejidad máxima es proporcional al producto del número de entradas del vector de indirección por el número de familias de estamentos. La distribución del vector a se realiza, por defecto, en bloques de igual tamaño. Si bajo estas circunstancias el código irregular muestra un bajo balanceo de carga, es necesario utilizar un algoritmo de balanceo cuya complejidad también hay que considerar. Hemos utilizado el algoritmo BALAN que tiene una complejidad máxima dada por la Expresión 5.14.

Un segundo aspecto que es necesario evaluar desde el punto de vista teórico, es el coste de memoria del ejecutor paralelo. Este coste depende de las características del código

```
DO j = 1, N_x
   a[x_1[j]] = b[j]
   a[x_2[j]] = a[x_2[j]] ⊕ b[j]
END DO
```

(a)

```
SHARED: ρ_1^slc, ρ_2^slc, x_1, x_2, b_1, b_2, a

DOALL p = 1, N_pt
   DO s = 1, N_S − 1
      DO j = ρ_1^slc[s,p], ρ_1^slc[s+1,p] − 1
         a[x_1^fin[j]] = b_1^fin[j]
      END DO
      DO j = ρ_1^slc[s,p], ρ_1^slc[s+1,p] − 1
         a[x_2^fin[j]] = a[x_2^fin[j]] ⊕ b^fin[j]
      END DO
   END DO
END DOALL
```

(b)

Figura 5.25: Ejemplo de código irregular: (a) código secuencial, (b) código paralelo.

irregular, las cuales se pueden clasificar en dos clases:

1. *Códigos irregulares en el que únicamente los vectores de indirección son accedidos mediante el índice el lazo.* Un ejemplo es el código de la Figura 5.9. En este caso, el coste temporal del ejecutor es debido a los vectores ρ^{slc} y viene dado por la siguiente expresión.

$$\Delta M_{\texttt{SLCSRT}} = N_S N_{pt} N_{stmt} \tag{5.16}$$

2. *Código irregulares para los que existen otros vectores, que no son de indirección, accedidos mediante el índice del lazo.* Vamos a denominar b a dicho vector, y por simplicidad vamos a asumir que únicamente puede ser accedido mediante el índice del lazo j. Igualmente asumiremos que el vector b aparece en N_{uso} familias de estamentos, con $1 < N_{uso} \leq N_{stmt}$. La Figura 5.25(a) muestra un ejemplo de esta clase de códigos con $N_{uso} = N_{stmt} = 2$. El código paralelo asociado se muestra en la Figura 5.25(b). Como cada uno de los vectores de indirección es reordenado, también es necesario reordenar el vector b. De este modo aseguramos que, cuando la entrada $a[x_k^{fin}[j]]$ ha sido calculada, se utiliza el valor correcto de $b_k^{fin}[j]$.

   Si $N_{uso} = 1$, únicamente basta con reordenar el vector existente, por lo que el coste de memoria asociado al ejecutor será el mismo que en el caso anterior (Expresión 5.16).

   Si $N_{uso} > 1$ y dado que cada vector de indirección ha sido reordenado de una forma diferente, será necesario crear $N_{uso} - 1$ copias del mismo, de modo que cada una

mantenga la misma reordenación que la del vector de indirección asociado al mismo estamento. De este modo, y denotando como N_b al número de entradas del vector b, tenemos,

$$\Delta M_{\texttt{SLCSRT}} = N_S N_{pt} N_{stmt} + (N_{uso} - 1) N_b \tag{5.17}$$

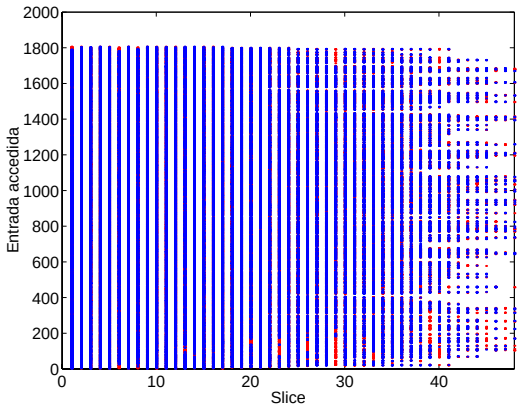
Nótese que la Expresión 5.16 es un caso particular de la 5.17, cuando $N_{uso} \leq 1$.

El proceso de reordenación de b se realiza en la fase de inspección sin ningún coste computacional añadido, ya que la operación de permutación realizada sobre el mismo es la misma que la aplicada sobre el vector de indirección.

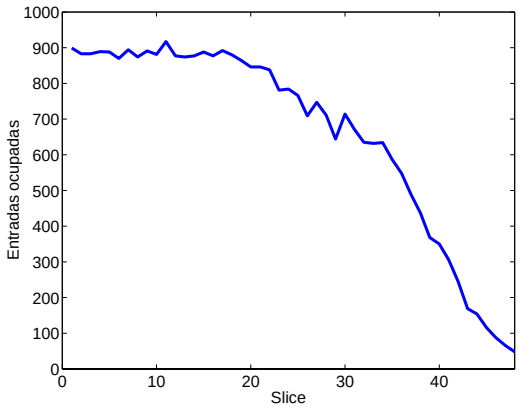
Eficiencia en la extracción de paralelismo

Al reordenar los accesos se consigue un menor número de *slices*, es decir, un mayor grado de paralelismo. En esta sección evaluamos la eficiencia del algoritmo SLCSRT en términos de su capacidad de clasificar las entradas del patrón de acceso en *slices*. En esta sección hemos utilizado el mismo código de prueba y los mismos patrones de acceso que los empleados en la Sección 5.5.3, en donde evaluábamos la eficiencia del algoritmo CS3. La Tabla 5.5 muestra el número de *slices* obtenidos con la estrategia SLCSRT. Comparando estos valores con los obtenidos con la estrategia CS3 (mostrados en la Tabla 5.4) se puede apreciar que existe una importante reducción en el número de *slices* para la práctica totalidad los casos. La única excepción es la matriz *beaflw*, para la cual la existencia de un camino crítico elevado limita la capacidad de reducción en el número de *slices*.

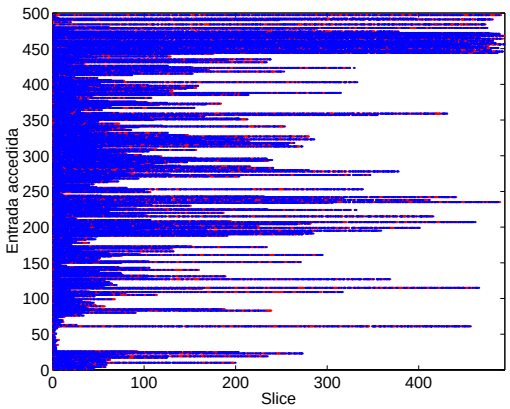
La Figura 5.26 muestra el patrón de acceso y nivel de ocupación para tres de estas matrices. Nuevamente, estos resultados se pueden comparar con los obtenidos para la estrategia CS3 en la Figura 5.17. Observando ambas figuras se puede apreciar un cambio en la distribución de las entradas en los *slices*. Si para el algoritmo CS3 esta distribución presentaba bruscas oscilaciones, con la estrategia SLCSRT los primeros *slices* pasan a tener una alta densidad de entradas, y los últimos un bajo nivel de ocupación. Para la matriz *beaflw* el número total de *slices* es próximo al obtenido con el algoritmo CS, pero la distribución de las entradas cambia radicalmente. La causa de este comportamiento radica en que existe un gran número de entradas que no originan dependencias y que están ahora concentradas en los primeros *slices*, mientras que es menos frecuente encontrar aquellas que tengan asociado un número de secuencia elevado y, consiguientemente, estén ubicadas en los últimos *slices*.



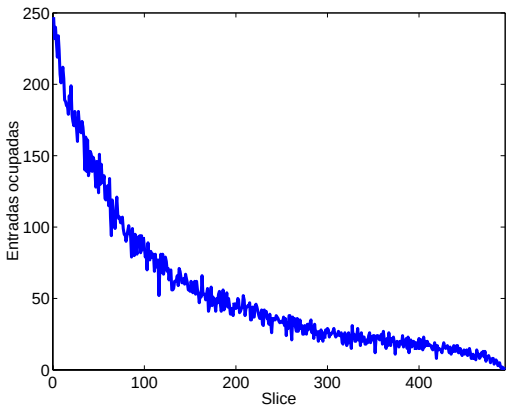
Patrón de la matriz *bcsstk14*



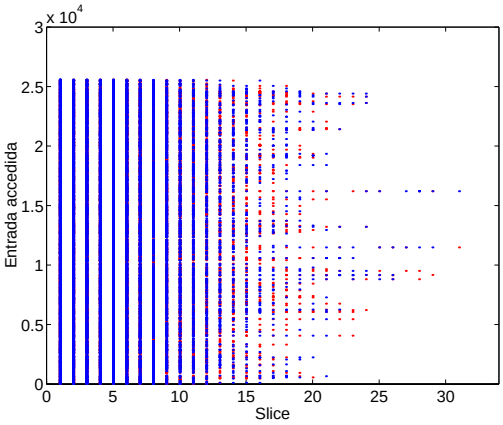
Nivel de ocupación de la matriz *bcsstk14*



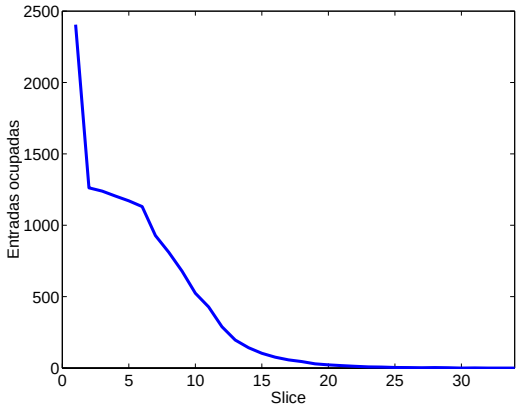
Patrón de la matriz *beaflw*



Nivel de ocupación de la matriz *beaflw*



Patrón de la matriz *25600_90*



Nivel de ocupación de la matriz *25600_90*

Figura 5.26: Resultado del algoritmo de clasificación por *slices* `SLCSRT`.

```
DO j = 1, N_x
   DO w = 1, W
      carga para el estamento stmt_j^1
   END DO
   tmp1 = tmp1 + a[x_1[j]]
   DO w = 1, W
      carga para el estamento stmt_j^2
   END DO
   a[x_2[j]] = tmp2
END DO
```

Figura 5.27: Código de prueba sintético.

Rendimiento obtenido

Como punto de partida hemos evaluado la eficiencia de nuestras propuestas sobre un código sintético que permite modificar sus principales características. Hemos comparado el rendimiento de nuestra propuesta con el de las estrategias CYT, LCYT y LO-LCYT. El sistema utilizado es una Silicon Graphics Origin2000 con 8 procesadores MIPS R10000. Todos los códigos (tanto del inspector como del ejecutor) fueron escritos en Fortran 77 y las versiones paralelas se implementaron utilizando las directivas OpenMP. El compilador empleado es el MIPSpro f77 v7.4. Todas las matrices están alineadas en memoria mediante la opción "-aling128".

La Figura 5.27 muestra la estructura del código sintético utilizado. Este consiste en un lazo que contiene dos familias de estamentos, cada uno de los cuales realiza un acceso irregular sobre la matriz a. Con el propósito de modificar la carga computacional, cada uno de los estamentos tiene asociado un lazo de W iteraciones que regula el valor de la carga computacional. Por simplicidad asumimos que W es igual para ambos estamentos.

Los patrones de acceso que hemos utilizado provienen de dos fuentes distintas: por una parte los hemos extraído de rutinas dispersas provenientes de programas reales [37], y por

Matriz	*bcsstk14*	*beaflw*	*psmigr_2*	*25600_90*	*25600_U*
Número de *slices*	48	494	2293	34	6

Tabla 5.5: Eficiencia de técnica SLCSRT con distintos patrones de acceso.

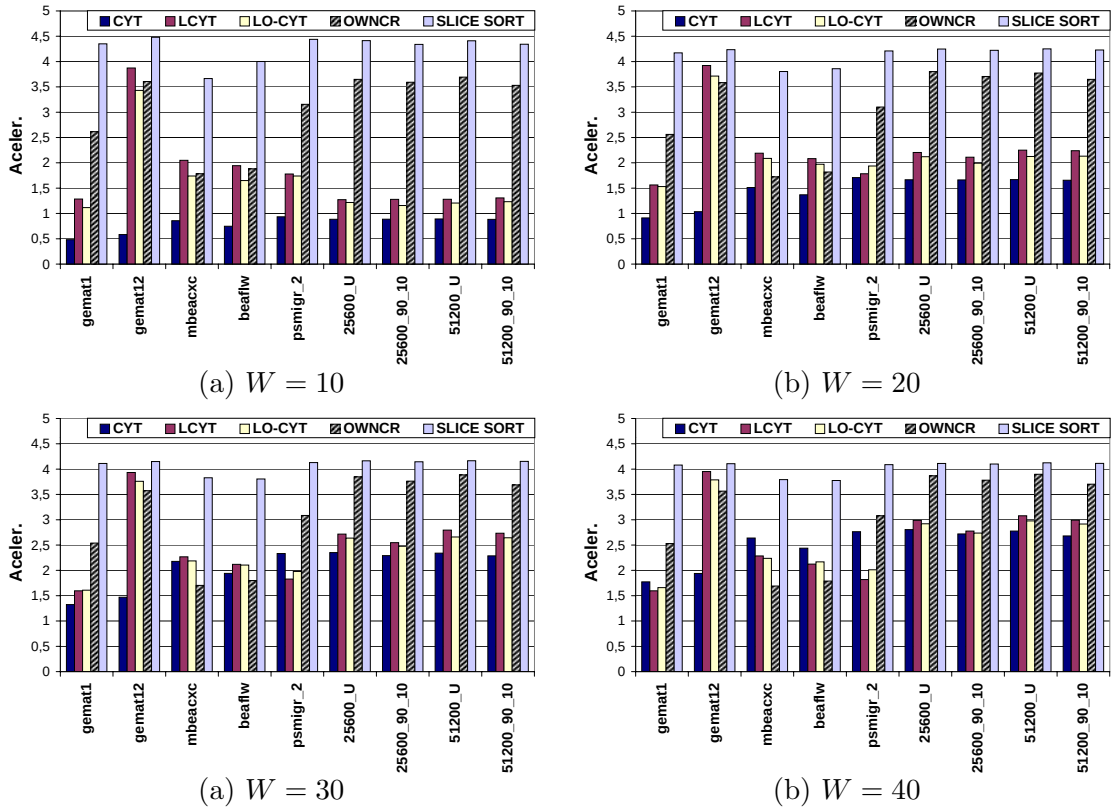
Matriz	N_x	N_a	CC	N_S
$gemat1$	23684	4929	4938	4928
$gemat12$	16555	4929	49	44
$mbeacxc$	24960	496	487	485
$beaflw$	26701	507	500	495
$psmigr\_2$	270011	3140	2626	2294
25600_U	12800	25600	9	7
25600_90_10	12800	25600	45	35
51200_U	25600	51200	11	9
51200_90_10	25600	51200	46	30

Tabla 5.6: Patrones de acceso utilizados para el código de prueba sintético.

otra, los hemos generado sintéticamente. De forma análoga a la Sección 5.4.4, los patrones de acceso sintético se pueden clasificar en uniformes y no uniformes. La Tabla 5.6 muestra las principales características de los patrones de acceso utilizados. CC hace referencia al camino crítico, introducido en la Definición 5.4.1. Esta variable nos permite estimar el grado de paralelismo del lazo cuando se utilizan las estrategias `CYT`, `LCYT` y `LO-LCYT`. Para el caso de la técnica `SLCSRT`, el grado de paralelismo viene determinado por el número de *slices*(N_S).

En la Tabla 5.6 se puede apreciar que para todos los casos el camino crítico es mayor que el número de *slices*. Esto es debido a que las estrategias anteriores tienen en cuenta las dependencias entre iteraciones, en vez de estamentos individuales. Aunque es cierto que la estrategia `CYT` permite un solape en la ejecución de estamentos asociados a iteraciones consecutivas, no permite un reordenamiento total de los mismos. Por ejemplo, consideremos el lazo de la Figura 5.1(c) con $N_x = 3$, $N_{stmt} = 2$, $x_1 = \{1, 2, 3\}$ y $x_2 = \{2, 3, 4\}$. Para el caso de las estrategias basadas en la técnica `CYT`, el camino crítico es 3, dado que cada iteración depende de la previa. Sin embargo, realizando la distribución individual de cada uno de los estamentos, únicamente son necesarios dos *slices* para preservar las dependencias de datos existentes en el lazo, ya que se puede ejecutar en paralelo la iteración primera con la tercera. De este modo, podemos concluir que los resultados mostrados en la Tabla 5.6 demuestran que mediante el empleo de la estrategia `SLCSRT` se obtiene una mejor explotación del paralelismo existente en el lazo.

La Figura 5.28 muestra, para distintas cargas de trabajo, las aceleraciones obtenidas con los ejecutores de cada una de las propuestas. En el caso de la regla del propietario



(a) $W = 10$ (b) $W = 20$

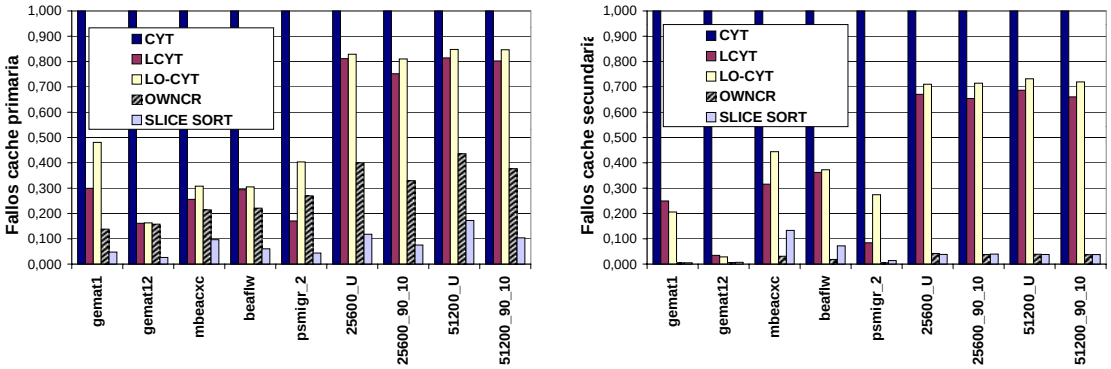
(a) $W = 30$ (b) $W = 40$

Figura 5.28: Aceleraciones obtenidas para diferentes cargas de trabajo con 8 procesadores.

(denotada como OWNCR) se realizó una distribución de las entradas de a en bloques de igual tamaño. El rendimiento de esta propuesta tiene un buen comportamiento para las matrices sintéticas, dado que estas están bien balanceadas[2]. Sin embargo, este rendimiento decae fuertemente cuando se emplean patrones de acceso reales, los cuales están mucho más desbalanceados.

Para todos los casos, la estrategia SLCSRT obtiene los mejores resultados en términos de aceleración del ejecutor. Esto es debido a que conjuga un correcto balanceo de carga y una eficiente explotación de la jerarquía de memoria del sistema. Del mismo modo que sucedía con las estrategias LCYT y LO-LCYT, el rendimiento de la estrategia SLCSRT decrece conforme W aumenta. Nuevamente, esto es debido a que para valores cada vez

[2]Notar que las matrices no uniformes también están bien balanceadas desde el punto de vista de la estrategia OWNCR. Esto se debe a que aunque el 10% de las entradas tiene asociada la máxima carga computacional, la distribución de las mismas sobre a es homogénea.



(a) (b)

Figura 5.29: Fallos en memoria cache para 8 procesadores con $W = 10$: (a) fallos de cache primaria, (b) fallos de cache secundaria.

mayores de W, el impacto en la mejora de la localidad tiene una menor repercusión sobre el rendimiento del programa.

La Figura 5.29 muestra el número de fallos en cache primaria y secundaria. Estas medidas se corresponden al valor máximo obtenido en una ejecución paralela en 8 procesadores. Los resultados están normalizados respecto a la estrategia `CYT`. Se puede apreciar que, para todos los casos, mediante nuestra propuesta existe una importante reducción del número de fallos.

Matriz	$W = 10$				$W = 30$			
	OWNCR	CYT	LCYT	LO-LCYT	OWNCR	CYT	LCYT	LO-LCYT
*gemat*1	**2**	0	0	0	0	0	0	0
*gemat*12	**5**	0	**5**	**2**	**2**	0	**4**	**2**
mbeacxc	**1**	0	**1**	0	0	0	0	0
beaflw	**1**	0	0	0	0	0	0	0
*psmigr*_2	**5**	0	**1**	**1**	**2**	0	0	0
25600_*U*	**7**	0	0	0	**6**	0	0	0
25600_90_10	**7**	0	0	0	**5**	0	0	0
51200_*U*	**9**	0	0	0	**8**	0	0	0
51200_90_10	**7**	0	0	0	**4**	0	0	0

Tabla 5.7: Umbral de iteraciones para que la estrategia `SLCSRT` supere al resto de las propuestas.

La Tabla 5.7 muestra el rendimiento total de la estrategia SLCSRT tendiendo en cuenta el coste computacional del inspector. De forma más específica, esta tabla contiene el número de veces que el inspector debe ser reusado para que el tiempo de ejecución total (inspector + ejecutor) sea inferior al del resto de las propuestas. Por ejemplo, para la matriz *gemat*1, basta que el lazo irregular se ejecute dos o más veces para que la propuesta SLCSRT sea más rápida que la OWNCR. Nótese que estamos considerando el coste asociado al inspector SLCSRT, mientras que la estrategia OWNCR, al no emplear una rutina de inspección, no tiene ningún coste adicional añadido. Una entrada con valor nulo significa que nuestra propuesta es siempre la más rápida, incluso ejecutando una única vez el lazo paralelo. Hay que destacar que el umbral de iteraciones mostrado en la Tabla 5.7 no es, por lo general, muy alto. Este umbral decrece conforme W aumenta y es más importante cuando se compara con la estrategia OWNCR.

Finalmente, hemos evaluado la eficiencia de nuestra propuesta con dos aplicaciones reales. La primera de ellas se corresponde a la subrutina *mltxmlt* del paquete PLTMG, edición 7.1 [11]. Este paquete es utilizado para resolver ecuaciones diferenciales parciales elípticas en regiones generales del plano. El código original de esta rutina se muestra en la Figura 5.30(a).

Aunque las dependencias de datos existentes en esta rutina puede ser eliminadas mediante el empleo de la técnica *array expansion*, hemos decidido incluirla en nuestro análisis por dos motivos. El primero de ellos es que va a permitir comparar el rendimiento de las estrategias OWNCR y SLCSRT con el obtenido mediante la estrategia *array expansion*. El segundo motivo radica en que para cierto tipo de aplicaciones, es necesario ejecutar el lazo en paralelo manteniendo el orden de accesos del lazo original. Esto no se consigue con la técnica *array expansion*, ya que modifica el orden de los accesos a memoria pudiendo producir cambios en el valor numérico del resultado. Nuestra propuesta reordena los estamentos asegurando el mismo orden de acceso que en el programa original, de modo que

Características				Aceleración con $N_p = 4$			Aceleración con $N_p = 8$		
Matriz	N_x	N_a	N_{slc}	OWNCR	*arr.exp.*	SLCSRT	OWNCR	*arr.exp.*	SLCSRT
*matriz*1	7500	1300	9	0.65	0.65	1.88	0.75	0.72	2.32
*matriz*2	21090	3595	9	0.55	0.80	2.25	0.76	0.91	3.21
*matriz*3	90480	15240	9	0.79	1.10	2.91	1.93	1.46	2.52

Tabla 5.8: Patrones de acceso utilizados y aceleraciones obtenidas para el código de prueba PLTMG.

```
DO i = 1, N_ja - 1
   DO k = ja[i], ja[i+1] - 1
      j = ja[k]
      b[i] = b[i] + a[k+u] * x[j]
      b[j] = b[j] + a[k+l] * x[i]

   END DO
END DO
```

(a)

```
DO j = 1, nop - 1
   t[ix[1]+j] += -2α⌊t[ix[1]+j]/α⌋
   t[ix[2]+j] += -2α⌊t[ix[2]+j]/α⌋
   t[ix[3]+j] += -2⌊t[ix[3]+j]⌋
END DO
```

(b)

Figura 5.30: Ejemplos de lazos irregulares reales: (a) subrutina *mltxmlt*, (b) lazo 711 de la rutina *correc*.

el resultado obtenido con el código paralelo coincide exactamente con el secuencial.

El paquete PLTMG contiene patrones de acceso propios que pueden ser modificados mediante parámetros. En nuestro caso, hemos hecho las distintas pruebas comparativas para tres tamaños diferentes de problema. Las principales características de cada uno de los problemas y los resultados obtenidos se muestran en la Tabla 5.8. Se puede apreciar que la estrategia `SLCSRT` es la única en conseguir aceleraciones significativas.

El segundo código de pruebas ha sido extraído del paquete *Perfect Club Benchmarks* [14]. Más concretamente, hemos utilizado el lazo 711 de la rutina *correc* de la aplicación BDNA. La Figura 5.30(b) muestra este lazo. De igual forma que en código anterior, el patrón de acceso es generado automáticamente por la aplicación. Empíricamente hemos podido comprobar que los valores del vector ix aseguran la no existencia de dependencias de datos, por lo que este lazo puede ser ejecutado en paralelo. Esto concuerda con el resultado obtenido mediante el inspector `SLCSRT`, que determinó la existencia de un único *slice*.

Características				Aceleración con 4 procesadores		Aceleración con 8 procesadores	
Matriz	N_x	N_a	N_{slc}	OWNCR	SLCSRT	OWNCR	SLCSRT
*matriz*1	1520	5830	1	1.1	2.1	1.5	1.7

Tabla 5.9: Patrones de acceso utilizados y aceleraciones obtenidas para el código de prueba BDNA.

En los trabajos [26] y [150] este lazo fue paralelizado mediante la estrategia CYT, obteniéndose aceleraciones muy pobres. Los resultados obtenidos mediante nuestra propuesta se ilustran en la Tabla 5.9. Destacamos que las estrategias LCYT y LO-LCYT no han podido ser aplicadas ya que únicamente consideran lazos con un único acceso de escritura. Los resultados obtenidos con nuestras propuestas no consiguen aceleraciones muy altas. Sin embargo, estos resultados son razonables dadas las pequeñas dimensiones del problema.

5.6.5 Mejoras de la estrategia SLCSRT

En esta sección presentamos dos propuestas de mejora de la estrategia SLCSRT. La primera está orientada a refinar su mecanismo de análisis de dependencias, mientras que la segunda aplica la representación IARD para aumentar la eficiencia del inspector paralelo.

Mejora en la precisión del análisis de dependencias

Del mismo modo que los algoritmos CYT, LCYT y LO-LCYT, la estrategia SLCSRT considera que lecturas consecutivas sobre la misma posición de memoria originan dependencias de datos. En el caso de las propuestas previas, este hecho implica un aumento del número de elementos en la cadena de dependencias, de modo que, cuando el lazo es ejecutado en paralelo, la ejecución de varias lecturas sobre un mismo elemento es serializada mediante una rutina de sincronización. En el caso de la estrategia SLCSRT, ambas operaciones de lectura están asignadas al mismo procesador (dado que se aplica la regla del propietario) y la única repercusión de la serialización de este tipo de accesos es que ambas quedan asignadas a *slices* diferentes.

Desde el punto de vista de la eficiencia del ejecutor, un aumento en el número de *slices* implica un mayor número de iteraciones en el lazo que los recorre (lazo de índice s en Figura 5.23). El número de estamentos de la familia j accedidas por cada procesador p (denotado como $N^p_{x_k}$) viene dado por:

$$N^p_{x_k} = \sum_{s=1}^{N_S-1} (\rho^{slc}_k[s+1,p] - \rho^{slc}_k[s,p]) = (\rho^{slc}_k[N_S,p] - \rho^{slc}_k[1,p]) \tag{5.18}$$

Este valor es constante con independencia del número y tipo de dependencias que puedan existir en el lazo irregular. De este modo, un aumento en el número de *slices* únicamente implica una disminución de la densidad de entradas por *slice*. Las derivaciones en coste computacional no son evidentes. Inicialmente, se puede esperar que a mayor número de *slices* más veces es necesario inicializar el lazo interno. Este fenómeno será más patente en

aquellos códigos en los que la mayor parte de las familias de estamentos realicen operaciones de lectura.

Para evaluar estos efectos hemos desarrollado una variante del algoritmo `SLCSRT` denominada algoritmo `SLCSRT` con Lecturas Paralelas (`SLCSRT-LP`). Esta versión no considera a las lecturas del mismo elemento como dependencias de datos. El pseudocódigo de nuestra propuesta se muestra en la Figura 5.31. Dado que es necesario distinguir entre operaciones de lectura y de escritura, empleamos una notación con cambio de signo en los vectores ρ_k^{fila} para indicar el tipo de la última operación realizada. Previamente (líneas `L1`, `L2`, `L3` y `L4`), se detecta el tipo de acceso mediante una operación condicional. El propósito es almacenar en los vectores ρ_k^{fila} un resultado negativo si se trata de una operación de escritura (denotada como ``E''). Este es el caso de las líneas `L1` y `L2`, donde el valor se hace negativo y se incrementa (en valor absoluto) en una unidad. Este incremento es el que nos permite clasificar en otro *slice* al siguiente acceso sobre la misma entrada. En el caso de tratarse de un acceso de lectura (denotada como ``L'') pueden darse dos situaciones: la primera sucede cuando la entrada asociada en el vector ρ^{fila} es negativa (línea `L3`) lo cual indica que ha habido un acceso previo de escritura. En este caso se incrementa este vector y se cambia su signo a un valor positivo. En el segundo caso (línea `L4`), la entrada del vector ρ^{fila} es positiva, lo cual indica que el acceso previo ha sido una lectura por lo que se modifica el vector ρ^{fila}. Esto se traduce en que la entrada considerada va a ser asignada al mismo *slice* que la previa, por lo que no se incrementa el valor.

Dado que el vector ρ_y^{fila} puede tener valores negativos, es necesario modificar la función *slice*. En nuestro caso utilizamos la función *slice*2 (línea `L5`) que evalúa la siguiente expresión:

$$slice2(\rho^{fila}, x_k[j]) = \sum_{q=1}^{N_{stmt}} ABS(\rho_q^{fila}[x_k[j]]) \tag{5.19}$$

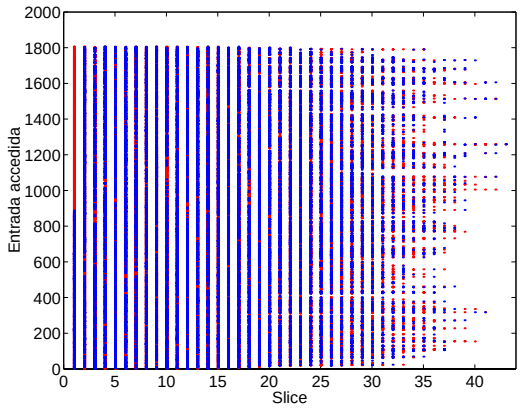
Mediante la función ABS se opera con valores absolutos eliminando la contribución asociada al signo del vector densidad. Finalmente, la introducción de un signo en los vectores de densidad de filas imposibilita que la fase de reordenamiento se pueda realizar recorriendo las entradas en sentido opuesto. En este caso es necesario modificar la fase de

Matriz	*bcsstk14*	*beaflw*	*psmigr_2*	*25600_90*	*25600_U*
Número de *slices*	44	494	1707	24	6

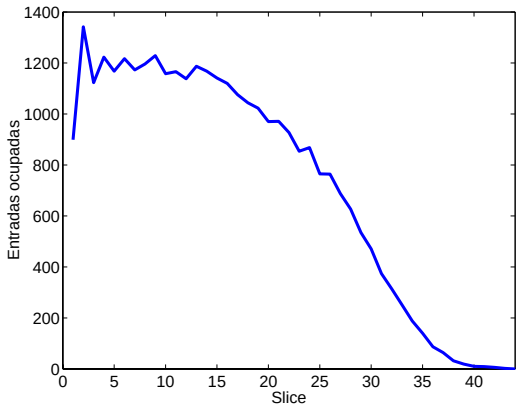
Tabla 5.10: Eficiencia de la técnica `SLCSRT-LP` con distintos patrones de acceso.

```
Algoritmo SLCSRT-LP
entrada
        {x_1, x_2 ... x_{N_stmt}}: vectores de indirección
salida
        {x_1^fin, x_2^fin ... x_{N_stmt}^fin}: vectores de indirección reordenados
        {ρ_1^slc, ρ_2^slc ... ρ_{N_stmt}^slc}: vectores de densidad
inicio del algoritmo

   SHARED: x, ρ^fila, ρ^slc, X^fin
   DOALL p = 1, N_pt
      DO j = 1, N_x                                   % Fase de análisis
        DO k = 1, N_stmt
          IF (x_k[j] ∈ A_p)
   L1       IF (acceso^k = ''E'' & ρ_k^fila[x_k[j]] > 0)        ρ_k^fila[x_k[j]] = −ρ_k^fila[x_k[j]] − 1
   L2       ELSE IF (acceso^k = ''E'' & ρ_k^fila[x_k[j]] < 0)   ρ_k^fila[x_k[j]] − −
   L3       ELSE IF (acceso^k = ''L'' & ρ_k^fila[x_k[j]] < 0)   ρ_k^fila[x_k[j]] = 1 − ρ_k^fila[x_k[j]]
   L4       ELSE IF (acceso^k = ''L'' & ρ_k^fila[x_k[j]] > 0)   nop
   L5       ρ_k^slc[slice2(ρ^fila, x_k[j]), p] + +
          END IF
        END DO
      END DO

      DO s = N_S, 1                                   % Fase de desplazamiento
        DO k = 1, N_stmt
   L6     ρ_k^slc[s, p] = C1(x_k, p) + C3(ρ_k^slc, p, s)
        END DO
      END DO
   L7 ρ^fila = 0

      DO j = N_x, 1                                   % Fase de reordenamiento
        DO k = 1, N_stmt
          IF (x_k[j] ∈ A_p)
            IF (acceso^k = ''E'' & ρ_k^fila[x_k[j]] > 0)        ρ_k^fila[x_k[j]] = −ρ_k^fila[x_k[j]] − 1
            ELSE IF (acceso^k = ''E'' & ρ_k^fila[x_k[j]] < 0)   ρ_k^fila[x_k[j]] − −
            ELSE IF (acceso^k = ''L'' & ρ_k^fila[x_k[j]] < 0)   ρ_k^fila[x_k[j]] = 1 − ρ_k^fila[x_k[j]]
            ELSE IF (acceso^k = ''L'' & ρ_k^fila[x_k[j]] > 0)   nop
            ρ_k^slc[slice2(ρ^fila, x_k[j]), p] + +
            x_k^fin[ρ_k^slc[slice2(ρ^fila, x_k[j]), p]]] = x_k[j]
          END IF
        END DO
      END DO
   END DOALL
fin del algoritmo
```

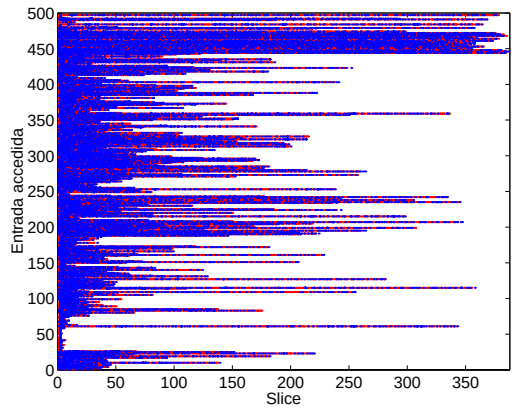
Figura 5.31: Pseudocódigo del inspector paralelo con lecturas paralelas (SLCSRT-LP).



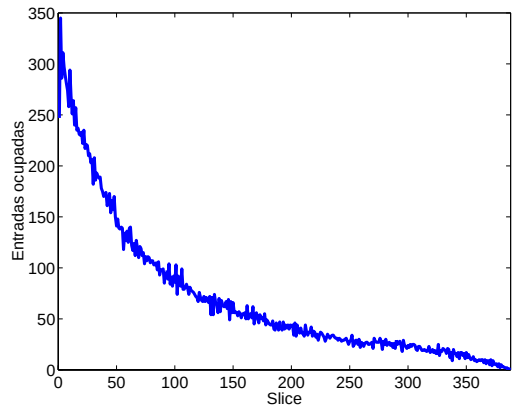
Patrón de la matriz *bcsstk14*



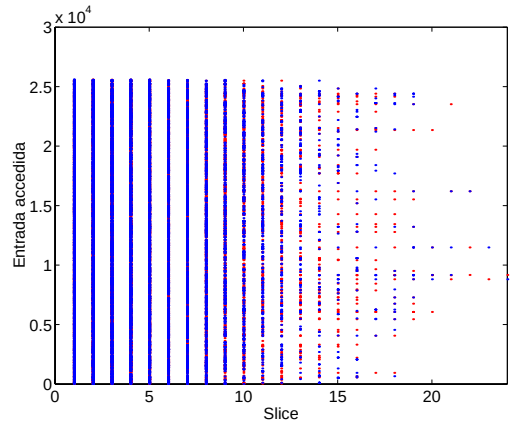
Nivel de ocupación de la matriz *bcsstk14*



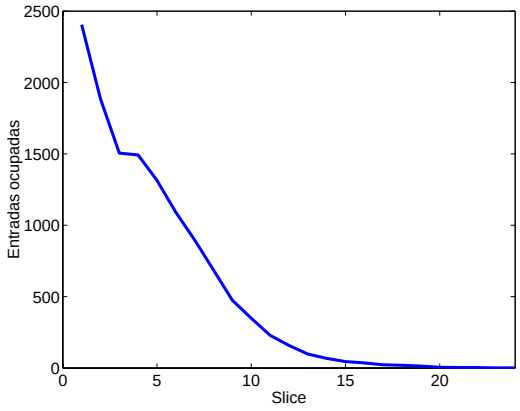
Patrón de la matriz *beaflw*



Nivel de ocupación de la matriz *beaflw*



Patrón de la matriz *25600_90*



Nivel de ocupación de la matriz *25600_90*

Figura 5.32: Resultado del algoritmo `SLCSRT-LP`.

desplazamiento (línea L6) reemplazando la función de coste $C2$ por la siguiente función:

$$C3(\rho_k^{slc}, p, s) = \sum_{i=1}^{s-1} \rho_k^{slc}[i, p] \tag{5.20}$$

De este modo conseguimos que cada entrada de los vectores ρ^{slc} apunte a la primera entrada del *slice* reordenado, en vez de a la última. Adicionalmente, los vectores ρ^{fila} son puestos a cero (línea L7) con el fin de que la fase de reordenamiento procese las entradas en el mismo orden que en la de análisis.

Hemos evaluado este algoritmo siguiendo los mismos criterios que los empleados en el algoritmo CS3 (Sección 5.5.3) y la estrategia SLCSRT (Sección 5.6.4). Los resultados obtenidos se muestran en la Tabla 5.10 y la Figura 5.32. Se puede apreciar una sensible reducción en el número de *slices* cuando se compara esta propuesta con la SLCSRT. En algunas matrices, como *psmigr_2* y *25600_90* esta reducción es más importante. Considerando el nivel de ocupación de cada *slice*, y comparando las figuras 5.32 y 5.26, se puede notar como mediante esta última estrategia el nivel de ocupación aumenta para los primeros *slices*. Cabe destacar que esta reducción sería mucho mayor si el código de prueba tuviera un mayor número de familias de estamentos con operaciones de lectura. Es este caso la reducción del número de *slice* sería mucho más significativa, lo que implicaría una mejora en el rendimiento del ejecutor paralelo.

Aplicación de la representación IARD

En el pseudocódigo de la Figura 5.19 se puede apreciar que en el algoritmo SLCSRT cada procesador debe ejecutar el lazo etiquetado como L1, en el que se recorren todas las entradas del vector de indirección. El número de iteraciones de este lazo se mantiene constante con el número de procesadores, lo que hace que el rendimiento del inspector SLCSRT paralelo sea pobre y se obtengan unas aceleraciones reducidas.

Una mejora de este algoritmo consiste en emplear la representación IARD con el fin de que cada procesador únicamente evalúe aquellas entradas de los vectores de indirección que acceden a la partición que tiene asignada. El proceso de aplicación es simple, y parte de la representación IARD de varias indirecciones introducida en la Sección 5.5. Asumiendo conocido el conjunto de particionamiento $\mathcal{LIM}$, sobre esta representación podemos aplicar el algoritmo PT (Sección 4.2.1) para obtener el conjunto de *slices* asociados a cada partición. Siendo más específicos, cada partición p tiene asociado el conjunto de *slices* $\{s_p^1, s_p^2, s_p^3, s_p^4\}$. Dado que cada partición está asignada a un procesador, este únicamente debe considerar el conjunto de entradas comprendidas entre los *slices* s_p^1 y s_p^4, ambos inclusive. Una

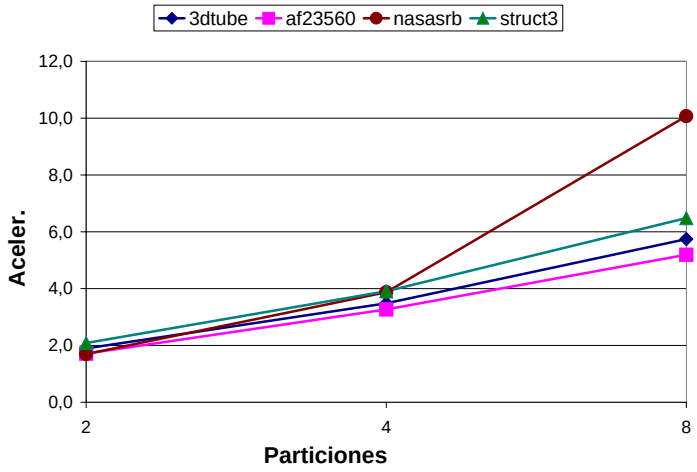


Figura 5.33: Aceleraciones obtenidas con el inspector paralelo empleando la representación IARD.

solución conservadora para determinar el intervalo de iteraciones que cada procesador debe considerar, es la dada por las siguientes expresiones:

$$j^p_{min} = min(\rho^{ini}_k[s^1_p], \quad k = 1, N_{stmt}) \tag{5.21}$$

$$j^p_{max} = max(\rho^{ini}_k[s^4_p + 1], \quad k = 1, N_{stmt}) \tag{5.22}$$

De este modo, el procesador p únicamente debe considerar el intervalo de iteraciones $[j^p_{min}, j^p_{max}]$. Este intervalo contiene todos los accesos de los vectores de indirección que se realizan sobre la partición $\mathcal{A}_p$ que tiene asignada.

En el caso de que los patrones de acceso asociados a cada indirección tengan una estructura bandeada, el número de iteraciones consideradas disminuirá conforme aumente el número de procesadores. Bajo estas circunstancias, el inspector paralelo presenta una buena escalabilidad. La Figura 5.33 muestra las aceleraciones obtenidas para el algoritmo con tres matrices de la librería Harwell-Boeing. En todos los casos se utilizó el mismo número de procesadores que de particiones.

Una vez finalizada la descripción de las distintas propuestas que hemos realizado para esta clase concreta de códigos irregulares, sólo nos resta el análisis de aquellos que están compuestos por reducciones irregulares. Este estudio se realiza en el siguiente capítulo.

Capítulo 6

Optimización de códigos irregulares con operaciones de reducción. El problema de los n-cuerpos

Una familia de códigos irregulares de especial importancia, y cuya paralelización hemos abordado de manera específica, son las reducciones irregulares. Y particularizando aún más, nos hemos centrado en las aplicaciones denominadas comúnmente *códigos de simulación de n-cuerpos* para las que las operaciones de reducción juegan un papel fundamental. En estas aplicaciones se simula la interacción de un conjunto de cuerpos confinados en una región espacial y sometidos a un campo de fuerzas. Este tipo de simulación se aparece en un gran número de aplicaciones científicas y comerciales, comprendiendo desde la simulación de movimientos de cuerpos celestes (sometidos a la interacción gravitatoria) hasta la interacción de un conjunto de partículas (sometidos a la interacción electromagnética).

La estructura de código irregular que vamos a considerar en este capítulo se corresponde, dentro del esquema general de la Figura 1.3, a lazos de la Clase C del módulo Paralelización de lazos con varias indirecciones. La principal característica de esta familia de códigos es que la única fuente de dependencias se debe a los accesos irregulares a una matriz mediante operaciones de reducción. Debido a la naturaleza de esta clase de operaciones, es posible realizar una paralelización del código cambiando el orden de ejecución de los distintos estamentos que conforman el programa. Este hecho nos permite aplicar estrategias de paralelización mucho más agresivas que las descritas en el capítulo previo,

pero también aparecen nuevas causas de ineficiencia que deben ser tratadas.

En este capítulo realizamos dos propuestas que abordan la paralelización de este tipo de estructuras mediante una estrategia de clasificación por *slices*. Inicialmente presentamos una primera aproximación que, a pesar de no producir un resultado competitivo, resulta de gran utilidad para identificar los aspectos que limitan la eficiencia en la ejecución paralela del programa. Nuestra segunda propuesta, denominada algoritmo `SLCCLS`, resuelve las desventajas de la propuesta previa, y ofrece una solución eficiente a la paralelización de esta clase de códigos. Las distintas propuestas realizadas en este capítulo han contribuido a la publicación del siguiente trabajo: "Automatic generation of optimized parallel code for n-body simulations" publicado en el *5th International Conference on Parallel Processing and Applied Mathematics* en septiembre del 2003 [132].

En la Sección 6.1 se introduce el tipo de estructura de código cuya paralelización vamos a abordar. Adicionalmente, en esta misma sección hacemos una revisión de las distintas técnicas existentes que permiten paralelizar esta clase de códigos. En la Sección 6.2 introducimos y evaluamos una primera propuesta para su paralelización, la cual nos va a permitir identificar los aspectos que determinan de forma más importante el rendimiento del programa paralelo. En base a esta información, en la Sección 6.3 presentamos una nueva propuesta para la paralelización de esta clase de códigos. En esta misma sección estudiamos su rendimiento y lo comparamos con otras técnicas existentes. Finalmente, en la Sección 6.4 refinamos el diseño de la propuesta, introduciendo diversas mejoras.

6.1 Trabajo previo

Inicialmente, vamos a describir la estructura del código irregular cuya paralelización hemos abordado. Existen distintas implementaciones de aplicación de simulación de n-cuerpos [22, 134, 108]. Una de las estructuras más populares consiste en el uso de vectores de indirección como el mostrado en la Figura 6.1. El vector *posición* almacena las coordenadas de cada uno de los cuerpos. Sin pérdida de generalidad y con el fin de simplificar nuestra representación, asumimos un entorno de simulación unidimensional para el que existe una única coordenada espacial.

Cada uno de los cuerpos únicamente interacciona con aquellos elementos con los que guarda la mayor proximidad especial. El lazo más interno (lazo j) recorre el número total de interacciones existente en el sistema simulado. Por medio de las indirecciones x_1 y x_2, en cada una de las iteraciones del lazo se accede a una pareja de cuerpos sobre la que existe una interacción. Una vez obtenidas las coordenadas de cada uno de ellos, la función

```
DO  t = 1, N_t
    DO  j = 1, N_x
        fuerza = calcula_fuerza(posición[x_1[j]], posición[x_2[j]])
        a[x_1[j]] = a[x_1[j]] + fuerza
        a[x_2[j]] = a[x_2[j]] - fuerza
    END DO
    posición[1 : N_a] = actualiza_posición(a)
END DO
```

Figura 6.1: Lazo irregular de n-cuerpos.

calcula_fuerza obtiene la fuerza de atracción (o repulsión) debida a su interacción. El valor de esta fuerza se acumula, mediante una operación reducción, sobre el vector a. Dicho vector tiene una entrada por cada cuerpo simulado y almacena, en su posición i-ésima, la suma de todas las fuerzas que el conjunto de vecinos produce sobre el cuerpo i-ésimo. Este conjunto de valores es empleado por la función *actualiza_posición* para obtener el conjunto de nuevas posiciones espaciales de cada uno de los cuerpos. Una vez actualizadas, se ejecuta una nueva iteración del lazo de tiempos (lazo t), volviéndose a repetir todo el proceso anterior para el siguiente intervalo de tiempo.

Este tipo de estructura se puede paralelizar aplicando diversas técnicas. Si retomamos la descripción que realizamos en el Capítulo 4 sobre las distintas estrategias existentes, buena parte de ellas pueden ser generalizas a esta nueva situación. Así pues, tanto las técnicas basadas en el empleo de primitivas de sincronización, como las basadas en la privatización del vector de reducción, pueden ser utilizadas sin introducir grandes modificaciones en su estructura.

La aplicación de la técnica *Data Write Affinity with Loop Index Prefetching* (DWA-LIP) [52, 54] sobre un lazo con más de una indirección sufre modificaciones respecto a la descrita en el Capítulo 4. La Figura 6.2 muestra la estructura del inspector DWA-LIP secuencial para un lazo con dos indirecciones. Este inspector analiza el espacio de iteraciones y las clasifica en función del grado de localidad en los accesos sobre a realizados por cada iteración. Previamente, las entradas de a han sido distribuidas sobre los procesadores empleando una distribución por bloques. La función *owner* recibe como argumento el índice de una entrada de a, y devuelve el procesador propietario de la misma. Para cada iteración j del lazo original, el inspector obtiene tres valores denominados b_{min}, b_{max} y *db*. Los dos primeros valores contienen, respectivamente, el mayor y menor índice de procesador asociado a cada uno de los accesos, mientras que *db* representa la diferencia de

```
Algoritmo DWA-LIP
entrada
      {x1, x2}: vectores de indirección
salida
      count: número de elementos de cada clase
      init, next: punteros a los elementos de cada clase
inicio del algoritmo
        DO j = 1, Nx
            bmin = min(owner(x1[j]), owner(x2[j]))
            bmax = max(owner(x1[j]), owner(x2[j]))
            db = bmax − bmin
            IF(count(bmin, db) = 0)
              init[bmin, db] = j
            ELSE
              next[prev[bmin, db]] = j
            END IF
            prev[bmin, db] = i
            count(bmin, db) + +
        END DO

        DO b = 1, Np
            DO db = 0, Np − b
              next[prev[b, db]] = prev[b, db]
            END DO
        END DO
fin del algoritmo
```

Figura 6.2: Pseudocódigo del inspector secuencial de la técnica DWA-LIP.

estos valores.

En función de estos parámetros el inspector realiza una clasificación del espacio de iteraciones. Para cada valor de db, se definen $N_p - db$ conjuntos de iteraciones diferentes. Por ejemplo, las iteraciones con $db = 0$ corresponden a iteraciones exclusivas, dado que realizan ambos accesos en el mismo bloque de a. Estas iteraciones se clasifican en N_p conjuntos distintos, estando cada uno de ellos asociado a un bloque diferente de a. Las iteraciones con $db > 0$ son clasificadas en nuevos conjuntos en función de los valores de

```
SHARED a[1 : N_a], init[N_p, N_p], count[N_p, N_p], next[N_x]

DO db = 1, N_p
   DO is = 1, db + 1
      DOALL b = is, N_p − db, db + 1
         j = init[b, db]
         DO k = 1, count[b, db]
            ...
            f = fuerza(pos[x_1[j]], pos[x_2[j]]])
            a[x_1[j]] = a[x_1[j]] + f
            a[x_2[j]] = a[x_2[j]] + f
            j = next[k]
         END DO
      END DOALL
   END DO
END  DO
```

Figura 6.3: Ejecutor para la paralelización mediante la técnica DWA-LIP.

b_{min} y db. Los autores establecen reglas de ejecución de cada uno de estos conjuntos, determinando aquellos que se pueden ejecutar en paralelo sin originar conflictos en los accesos sobre a. En un caso general, para un valor de db dado, el número máximo de conjuntos que es posible ejecutar en paralelo es:

$$N_p^{max} \simeq \lceil \frac{N_p - db}{db + 1} \rceil \tag{6.1}$$

La Figura 6.3 muestra el ejecutor paralelo de esta técnica. Para poder realizar la clasificación anteriormente comentada, se emplean las matrices *count* e *init*. La matriz *count* indica el número de iteraciones pertenecientes a cada conjunto, mientras que la matriz *init* almacena el índice de la primera iteración asignada a cada uno de estos conjuntos. Cada columna de estas matrices se corresponde con los distintos conjuntos asociados a un valor db dado. Mediante el vector *next*, de N_x entradas, se recorren el resto de las iteraciones asociadas a cada uno de los conjuntos. Con el fin de evitar conflictos de acceso entre clases consecutivas, es necesario introducir operaciones de sincronización. En el caso del código de la Figura 6.3, existe una barrera implícita al final del lazo `DOALL`, la cual separa aquellos conjuntos de iteraciones entre los que pueden existir dependencias.

La Figura 6.4(a) muestra un ejemplo del contenido de estos vectores para $N_x = 9$, $N_a = 8$, $x_1 = \{3, 3, 1, 1, 1, 7, 7, 5, 4\}$, $x_2 = \{5, 4, 8, 2, 4, 5, 8, 6, 7\}$ y $N_p = 4$. En esta figura,

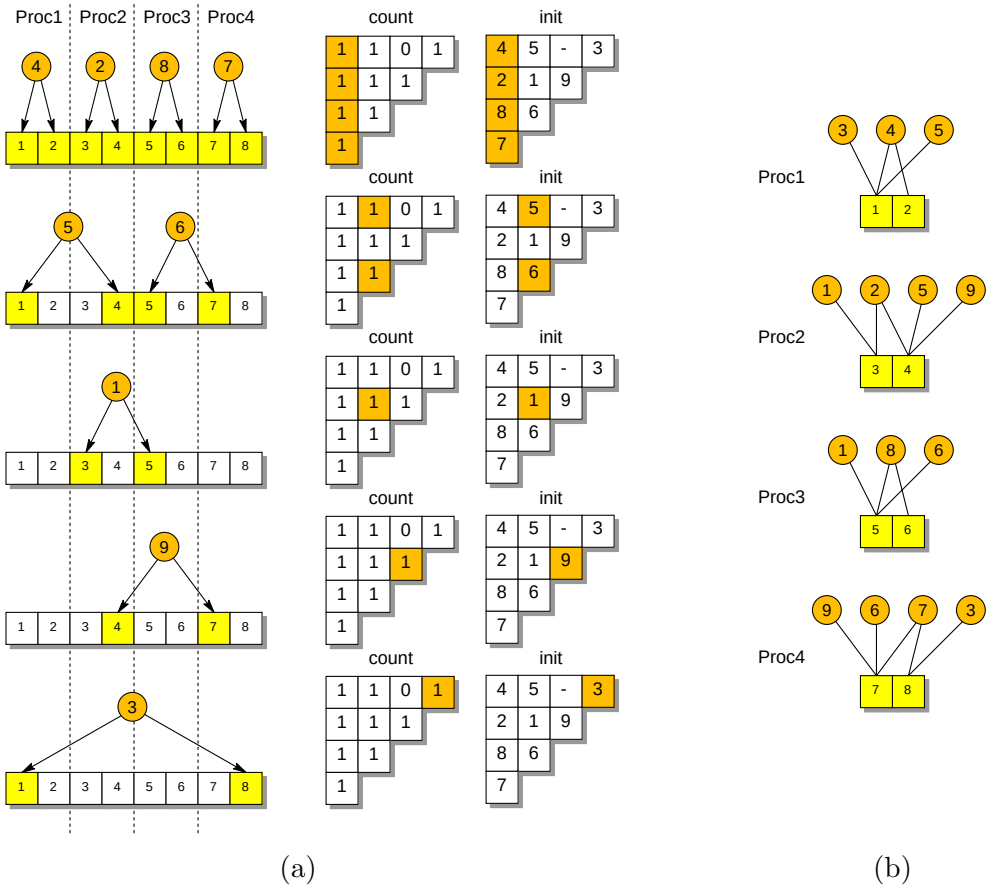


Figura 6.4: Esquema de ejecución para: (a) técnica DWA-LIP, (b) técnica LOCALWRITE.

las iteraciones del lazo aparecen denotadas por círculos, mientras que las entradas de a se representan con cuadrados. Cada fila de la representación muestra las iteraciones que pueden ser ejecutas en paralelo, denotando de forma sombreada las entradas accedidas en las tablas *count* e *init*. Así pues, las iteraciones con $db = 0$ son las 4, 2, 8 y 7 asignadas a cada uno de los conjuntos. Aplicando la Expresión 6.1 tenemos, para este ejemplo, que las cuatro iteraciones se pueden ejecutar en paralelo. Para $db = 1$ únicamente podremos ejecutar concurrentemente las iteraciones 5 y 6, mientras que el resto deberán ser ejecutadas de forma secuencial.

La técnica DWA-LIP no replica datos ni iteraciones, ofreciendo una solución escalable. El principal inconveniente que presenta es una considerable pérdida de paralelismo conforme se procesan conjuntos con valores db mayores. En el ejemplo de la figura, podemos

apreciar cómo de las nueve iteraciones, sólo cuatro pueden ejecutarse utilizando todos los procesadores disponibles. Dos iteraciones se pueden ejecutar en paralelo, mientras que el resto deben ser ejecutadas secuencialmente. El número de iteraciones asignadas a cada una de las clases depende de la localidad del patrón de acceso. De este modo, si existe una alta localidad, el valor *db* asociado a cada iteración será reducido, lo cual permite a esta propuesta extraer la mayor parte del paralelismo existente en el lazo irregular. El segundo gran inconveniente de esta propuesta es el número de operaciones de sincronización, las cuales aumentan de forma cuadrática con el número de procesadores. En [53] se propone una técnica que combina el empleo de la DWA-LIP con la *array expansion*. Mediante esta propuesta los autores aumentan el grado de paralelismo del la técnica DWA-LIP a costa de replicar (de forma limitada) la matriz a. Esta técnica, denominada *partial array expansion*, consigue una mejora en el el rendimiento respecto a la DWA-LIP obteniendo unos resultados comparables a la técnica de *array expansion* pero con unos costes de memoria menores.

Otra técnica que aborda la paralelización de esta clase de lazos es LOCAL-WRITE [59, 60, 61]. Esta técnica está orientada a arquitecturas de memoria compartida, y se ha implementado mediante el empleo de una versión mejorada del protocolo de coherencia CVM [77, 75]. La Figura 6.5 muestra el código del conjunto inspector-ejecutor correspondiente a esta técnica. La estructura del inspector es conceptualmente idéntica a la descrita en el Capítulo 4. El inspector LOCAL-WRITE es paralelo, en donde cada procesador recorre el espacio completo de iteraciones, seleccionando las que acceden a la partición de a que tiene asignada. Sin embargo, y debido a la existencia de dos indirecciones, es necesario clasificar las iteraciones del lazo en tres conjuntos.

Para un procesador p, el primero de estos conjuntos se almacena en la lista $list1_p$, y contiene todas las iteraciones exclusivas a la partición $\mathcal{A}_p$. Es decir, este conjunto contiene las que realizan ambos accesos sobre la partición a que dicho procesador tiene asignada. El segundo y tercer conjunto contienen, respectivamente, las iteraciones que acceden sobre la partición considerada mediante x_1 o x_2. Estas iteraciones son almacenadas respectivamente en las listas $list2_p$ y $list3_p$, locales a cada procesador.

Durante la ejecución paralela del lazo, cada procesador ejecuta las iteraciones de cada una de sus listas. En el caso de aquellas pertenecientes a la lista $list1_p$, y dado que son exclusivas, se almacena el acceso dado por ambas indirecciones. En el caso de las listas $list2_p$ y $list3_p$, únicamente se realiza el acceso local, que se corresponde, respectivamente, al dado por x_1 y x_2. La Figura 6.4(b) muestra un ejemplo de las iteraciones ejecutadas por cada procesador.

```
Algoritmo LOCALWRITE
entrada
     {x1, x2}: vectores de indirección
salida
     {list1, list2, list3}p: listas de iteraciones
inicio del algoritmo
  DOALL p = 1, Np
      DO j = 1, Nx
         IF(owner(x1[j]) = p & owner(x2[j]) = p)
               insert(j, list1p)
         ELSE IF(owner(x1[j]) = p)
               insert(j, list2p)
         ELSE IF(owner(x2[j]) = p)
               insert(j, list2p)
      END  DO
  END  DOALL
```

(a)

```
SHARED a[1 : Na]

DOALL p = 1, Np
    DO (i ∈ list1p)
       f = fuerza(pos[x1[j]], pos[x2[j]])
       a[x1[j]] = a[x1[j]] + f
       a[x2[j]] = a[x2[j]] + f
    END  DO

    DO (i ∈ list2p)
       f = fuerza(pos[x1[j]], pos[x2[j]])
       a[x1[j]] = a[x1[j]] + f
    END  DO

    DO (i ∈ list3p)
       f = fuerza(pos[x1[j]], pos[x2[j]])
       a[x2[j]] = a[x2[j]] + f
    END  DO
END  DOALL
```

(b)

Figura 6.5: Paralelización mediante la técnica de LOCAL-WRITE: (a) inspector, (b) ejecutor.

Mediante el empleo de la técnica LOCAL-WRITE se consigue una explotación eficiente de la localidad en los accesos sobre a. El principal inconveniente de esta propuesta es la replicación en las computaciones. Esta replicación se debe a que aquellas iteraciones que no son exclusivas deben ser ejecutadas de forma independiente para cada procesador propietario de las particiones sobre las que realizan acceso. Por otra parte, el coste de almacenamiento de esta propuesta puede ser importante, dado que la replicación de las computaciones implica la existencia de copias de una misma iteración (iteración compartida) asignadas a listas diferentes. Siendo más específicos, asumiendo que para un patrón de acceso y un esquema de particionamiento dados hay N_x^{excl} iteraciones exclusivas y N_x^{comp} compartidas (con $N_x = N_x^{excl} + N_x^{comp}$), entonces, el coste de memoria del ejecutor viene dado por la siguiente expresión:

$$\Delta M_{LOCAL-WRITE}^{ejecutor} = N_x^{excl} + 2N_x^{comp} \tag{6.2}$$

Esta técnica es susceptible de varias mejoras. Por ejemplo en [62, 57] se propone el empleo de particionadores con el fin de mejorar la localidad en los accesos.

Ambas técnicas de paralelización (la DWA-LIP y la LOCAL-WRITE) serán empleadas para establecer comparativas con el rendimiento de nuestras propuestas.

6.2 Algoritmo básico de clasificación por *slices*

En esta sección iniciamos la descripción del trabajo que hemos realizado, comenzado por el estudio de la viabilidad en el empleo de la representación por *slices* para la paralelización de esta clase de códigos.

6.2.1 Estructura del inspector-ejecutor

La idea básica que presentamos en nuestra primera propuesta consiste en distribuir las iteraciones sobre los procesadores forzando que el orden de los accesos sobre cada entrada de a sea el mismo que el programa secuencial. Hemos utilizado un tamaño de grano correspondiente a una iteración. De este modo, y estableciendo una analogía con la paralelización de códigos parcialmente paralelos, estamos asumiendo que en el lazo existe una única familia de estamentos que engloba todo el cuerpo del lazo. Consiguientemente, únicamente se pueden producir dependencias de datos entre iteraciones distintas.

La Figura 6.6 muestra el algoritmo básico de clasificación por *slices* que denominamos `SLCCL`. Para una iteración i dada, existen dos escrituras sobre el vector a en las posiciones de memoria $x_1[j]$ y $x_2[j]$. Vamos a adoptar la definición realizada en la Sección 5.5 para denotar los *slices* en esta nueva clase de códigos irregulares. De acuerdo con la definición empleada, cada *slice* contiene un conjunto de iteraciones que pueden ser ejecutadas en paralelo sin originar dependencias de datos. Una posible clasificación que satisface esta definición consiste en hacer que los *slices* contengan aquellas iteraciones en las que no se repiten accesos sobre la misma posición de memoria. Nuestra propuesta básica elabora esta estructura mediante el empleo de dos vectores auxiliares denominados ρ^a y ρ^s. El primero de ellos se utiliza para almacenar el número de accesos realizados sobre cada entrada de a. Empleando el contenido de este vector, para una iteración dada, el número de *slice* asociado viene dado por el valor máximo del número de accesos sobre cada una de estas dos posiciones. Es decir, para la j-ésima iteración, el *slice* s viene dado por:

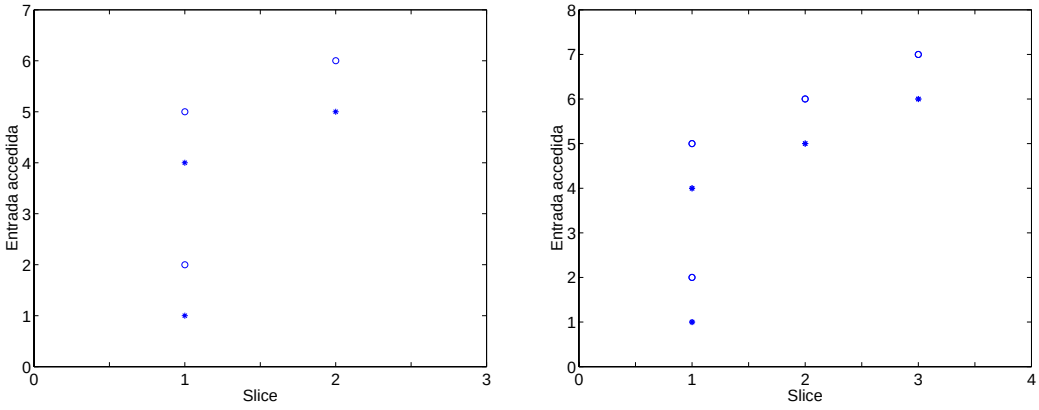
$$s = max(\rho^a[x_1[j]], \rho^a[x_2[j]]) + 1 \tag{6.3}$$

```
Algoritmo SLCCL
entrada
      {x_1, x_2}: vectores de indirección
salida
      {x_1^fin, x_2^fin}: vectores de indirección reordenados
      ρ^s: vector de densidad
inicio del algoritmo
  DO j = 1, N_x                          %Fase de análisis
      s = max(ρ^a[x_1[j]], ρ^a[x_2[j]]) + 1
      ρ^a[x_1[j]] = s
      ρ^a[x_2[j]] = s
      ρ^s[s] + +
  END DO

  DO s = 2, N_S                          %Fase de desplazamiento
      ρ^s[s] = ρ^s[s] + ρ^s[s − 1]
  END DO
  ρ^a ⟵ 0

  DO j = 1, N_x                          %Fase de reordenación
      s = max(ρ^a[x[j]], ρ^a[x_2[j]]) + 1
      ρ^a[x_1[j]] = s
      ρ^a[x_2[j]] = s
      x_1^fin[ρ^s[s] + cnt^s[s]] = x_1[j]
      x_2^fin[ρ^s[s] + cnt^s[s]] = x_2[j]
      cnt^s[s] + +
  END DO
fin del algoritmo
```

Figura 6.6: Algoritmo básico de clasificación por *slices* (SLCCL).

Donde la función *max* devuelve el máximo de los dos valores que recibe como argumento. Finalmente, este valor es utilizado para incrementar el vector ρ^s, el cual se emplea para contar el número de iteraciones asociadas a cada uno de los *slices*.

Posteriormente, en la fase de desplazamiento, este último vector es acumulado, convirtiéndose en un puntero al primer elemento de cada *slice* del vector reordenado. Finalmente, en la fase de reordenación todas las iteraciones son nuevamente analizadas y los vectores



(a) (b)

Figura 6.7: Ejemplo de generación del patrón de acceso a memoria.

de indirección reordenados son generados utilizando la información de este puntero.

Un ejemplo del funcionamiento de esta propuesta es el siguiente: asumamos los valores $x_1 = \{1, 4, 5\}$, $x_2 = \{2, 5, 6\}$. En este caso, tendremos que la primera iteración realiza escrituras sobre las entradas $a[1]$ y $a[2]$, y dado que no existen accesos previos sobre estas posiciones de memoria, dicha iteración queda asociada al primer *slice*. La segunda iteración escribe sobre las entradas $a[4]$ y $a[5]$. Como tampoco existe ningún acceso previo sobre esas posiciones de memoria, la segunda iteración también quedará asociada al primer *slice*. No obstante, la tercera iteración (que accede sobre los elementos $a[5]$ y $a[6]$) no puede ser asociada al primer *slice*, dado que la entrada $a[5]$ ya ha sido previamente accedida en la segunda iteración. El algoritmo básico propuesto establece que el *slice* asociado a esta iteración es el segundo. La Figura 6.7(a) muestra el patrón de acceso a memoria para este ejemplo. En esta figura se han representado los accesos de x_1 con el símbolo "•" y los de x_2 mediante el símbolo "○".

El ejecutor paralelo de este algoritmo se muestra en la Figura 6.8. El lazo interno es totalmente paralelo, sin embargo, es necesario realizar una operación de sincronización entre *slices* diferentes, ya que existe riesgo de que las iteraciones asociadas a distintos *slices* realicen accesos sobre la misma posición de memoria. Nótese que el número de sincronizaciones del ejecutor paralelo (realizadas de forma implícita al finalizar el lazo paralelo) aumenta conforme lo hace el número de *slices*, introduciendo una importante limitación en el paralelismo extraído del lazo.

```
DO  paso_tiempo = 1, Nt
   DO s = 1, NS
        DOALL j = ρ^s[s], ρ^s[s+1] − 1
              fuerza = calcula_fuerza(posición[x1[j]], posición[x2[j]])
              a[x1[j]]+ = fuerza
              a[x2[j]]− = fuerza
        END DOALL
    END DO
   posición[1 : Na] = actualiza_posición(a)
END DO
```

Figura 6.8: Ejecutor paralelo asociado al algoritmo SLCCL.

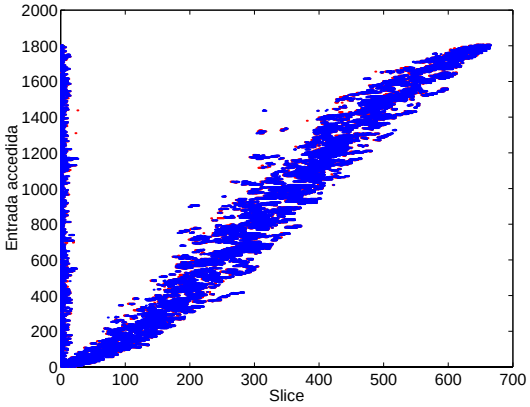
6.2.2 Análisis de eficiencia

La Figura 6.9 muestra el patrón de acceso a memoria obtenido bajo nuestra propuesta para las matrices *bcsstk14*, *beaflw* y *25600_90*. Dado que nuestro análisis es meramente teórico, no hemos considerado necesario emplear patrones de acceso pertenecientes a aplicaciones de simulación de n-cuerpos. Observando los valores de la tabla, se puede apreciar que para los primeros *slices* se consigue un alto nivel de ocupación, mientras que para los últimos el número de huecos es muy elevado. Este efecto se aprecia de un modo más significativo para patrones de acceso en banda, como es el caso de la matriz *bcsstk14*.

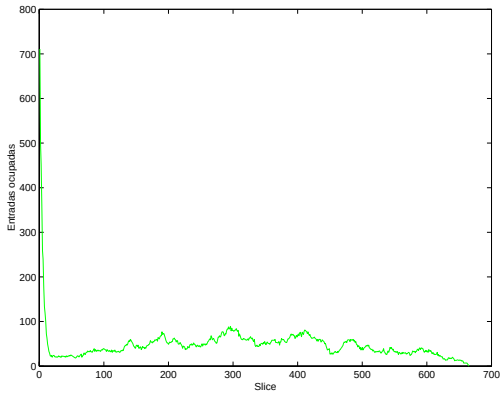
La Tabla 6.1 muestra el número de entradas del vector de indirección y el número de *slices* obtenidos para estos patrones de acceso. Observando los resultados mostrados en la tabla, podemos apreciar que el número de *slices* (y por consiguiente, de sincronizaciones) es muy elevado, a pesar de conseguir una reducción significativa respecto al número de entradas del vector de indirección. Otro aspecto que hay que considerar es el balanceo de la carga computacional. Debido a la introducción de una operación de sincronización

Matriz	*bcsstk14*	*beaflw*	*psmigr_2*	*25600_90*	*25600_U*
N_x	63453	26701	270011	12800	12800
Número de *slices*	665	501	2627	46	10

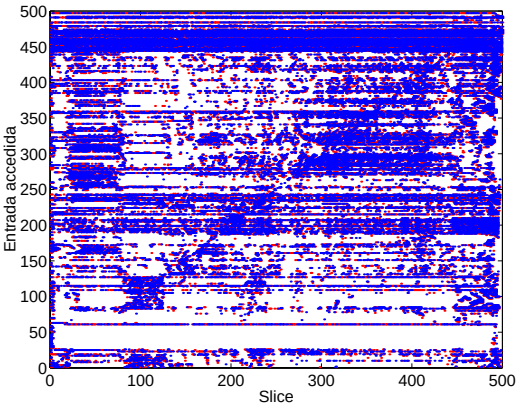
Tabla 6.1: Eficiencia de la clasificación por *slices* para SLCCL con distintos patrones de acceso.



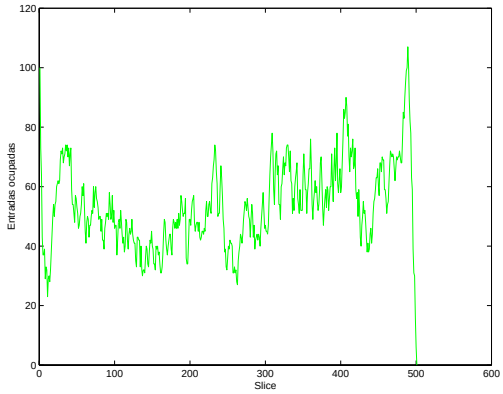
Patrón de la matriz *bcsstk14*



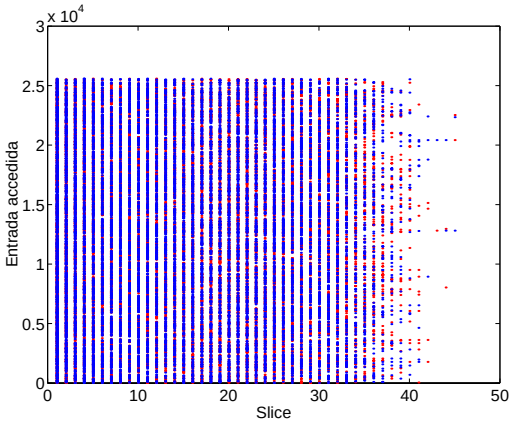
Nivel de ocupación de la matriz *bcsstk14*



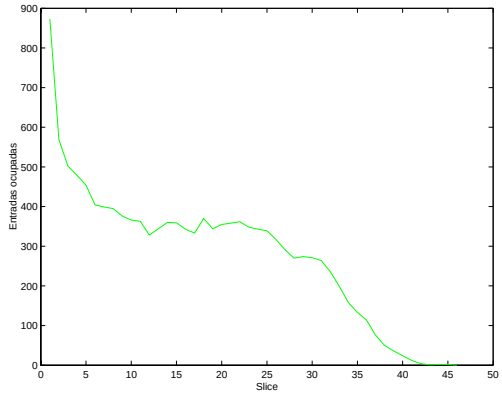
Patrón de la matriz *beaflw*



Nivel de ocupación de la matriz *beaflw*



Patrón de la matriz *25600_90*



Nivel de ocupación de la matriz *25600_90*

Figura 6.9: Patrón de acceso asociado al algoritmo `SLCCL` para las matrices *bcsstk14*, *beaflw* y *25600_90*.

tipo barrera entre la ejecución de *slices* consecutivos, es necesario balancear la carga de cada uno de los *slices* de forma independiente. Cuando el número de entradas de cada *slice* es reducido (lo cual sucede de forma especialmente acentuada con patrones de acceso bandeados), resulta muy difícil obtener un buen balanceo.

Cuando ejecutamos el inspector con patrones de acceso reales, la eficiencia alcanzada es bastante baja. Por este motivo no ofrecemos resultados de tiempo de ejecución ni aceleraciones. Hemos centrado nuestro estudio en la identificación de las causas de ineficiencia de esta propuesta, lo cual nos va a permitir desarrollar una alternativa con unas prestaciones competitivas.

Así pues, hemos identificado las siguientes necesidades asociadas a las principales causas de ineficiencia:

- El algoritmo `SLCCL` no extrae todo el paralelismo disponible. Vamos a evaluar el algoritmo `SLCCL` con el siguiente ejemplo: consideremos unos vectores $x_1 = \{1, 4, 5, 6\}$, $x_2 = \{2, 5, 6, 7\}$. El procesamiento de las iteraciones 1, 2 y 3 es análogo al del ejemplo anterior (Figura 6.7(a)). Si embargo, cuando se procesa la cuarta iteración se detecta un acceso previo sobre la entrada $a[6]$, por lo que se asigna al tercer *slice*. La Figura 6.7(b) muestra el patrón de accesos a memoria asociado a este nuevo ejemplo. Si la cuarta iteración hubiera sido procesada antes que la tercera, el *slice* que tendría asociada sería el primero, consiguiendo de este modo un mayor grado de ocupación, y un menor número de *slices*. Por lo tanto, es necesario aplicar, haciendo uso de las propiedades asociativas y conmutativas de las operaciones de reducción, un reordenamiento de las iteraciones. Mediante este reordenamiento se debe procurar reducir el número total de *slices*.

- El algoritmo `SLCCL` no explota la localidad en las operaciones de escritura. Cada procesador debe acceder, cuando ejecuta una iteración del lazo, a dos posiciones de memoria que están determinadas por el contenido de los dos vectores de indirección. En este caso, no resulta posible aplicar la regla del propietario con un tamaño de grano de iteración, dado que resulta imposible asegurar que ambos accesos se realizan sobre la partición considerada.[1] Mediante una reordenación de los vectores de indirección no se consigue aumentar la localidad de las operaciones de escritura,

[1] Una posibilidad consiste en replicar las computaciones, tal y como se propone en [59]. Sin embargo, hemos descartado esta opción dado que presenta como inconveniente una sobrecarga del coste computacional del ejecutor. Otra posibilidad consiste en el reetiquetado [94], mediante un algoritmo de reordenación, de cada uno de los cuerpos simulados. Esta opción resulta de gran interés pero tiene como inconveniente el no mitigar por completo este efecto.

dado que únicamente se consigue modificar el orden en el que las iteraciones son ejecutadas. Por este motivo, consideramos necesario reformar la estructura del ejecutor paralelo, de modo que permita explotar la localidad tanto en las lecturas de los vectores de indirección como en las escrituras sobre a.

- El algoritmo no balancea la carga. Es necesario desarrollar una correcta distribución de las distintas iteraciones sobre los procesadores con el fin de obtener un buen balanceo de la carga computacional.

En base a estas observaciones, podemos concluir que el empleo de una estrategia básica de clasificación por *slices* no resuelve de un modo eficiente la paralelización de este tipo de códigos. En la siguiente sección presentamos una nueva propuesta que implementa todas estas observaciones y permite extraer de un modo eficiente el paralelismo existente en este tipo de lazos.

6.3 Algoritmo avanzado de clasificación por *slices*

En esta propuesta presentamos un conjunto de técnicas orientadas a alcanzar cada uno de los objetivos expuestos en la sección previa. Nuestra propuesta está organizada de forma modular, constando de una serie de fases que gozan de cierto grado de independencia. La primera de nuestras técnicas consiste en una nueva estructura de almacenamiento que permite explotar la jerarquía de memoria del sistema. Adicionalmente, proponemos un nuevo esquema de ejecución paralelo del lazo irregular, con el que se alcanza un menor grado de conflictos de acceso a memoria. Finalmente, proponemos una nueva técnica de distribución de las iteraciones sobre los procesadores con la que se obtiene unos niveles casi óptimos de balanceo de la carga computacional.

Aunque estas tres propuestas están interrelacionadas, cada una de ellas puede ser modificada de forma independiente, obteniendo una gran flexibilidad en el desarrollo y optimización de nuestra propuesta. A continuación, en las siguientes secciones vamos a describir detalladamente cada uno de los aspectos que hemos enunciado.

6.3.1 Esquema de almacenamiento

En este trabajo vamos a abordar la paralelización de un código irregular que presenta la misma estructura que el mostrado en la Figura 6.1. Siguiendo la misma estrategia que en otras propuestas previas, y con el fin de aumentar la localidad en los accesos, vamos

a orientar todo nuestro esquema de paralelización automática a la aplicación de la regla del propietario sobre el vector en el que se realizan operaciones de escritura. En nuestro caso, este vector es el que almacena las contribuciones de las fuerzas y que denominamos vector a.

Hemos aplicado la regla del propietario sobre una distribución por bloques de a. Cada procesador p, accede únicamente sobre una partición dada a, que denominamos $\mathcal{A}_p$. Para una iteración j del lazo considerado, la función *extrae_propietario* devuelve el par de valores $\{proc_1, proc_2\}$ que identifican, respectivamente, el procesador propietario de las entradas $a[x_1[j]]$ y $a[x_2[j]]$. Asumiendo un particionamiento de a en bloques de igual tamaño, esta relación se puede obtener fácilmente por la siguiente expresión:

$$\{proc_1, proc_2\} = \{\lfloor \frac{x_1[j]N_p}{N_a} \rfloor, \lfloor \frac{x_2[j]N_p}{N_a} \rfloor\} \tag{6.4}$$

Hemos utilizado un grano de paralelismo a nivel de iteración. Es decir, las distintas iteraciones del lazo se distribuyen sobre los procesadores y no existe replicación de las computaciones. Dado que aplicamos la regla del propietario, y en cada iteración se realizan dos operaciones de escritura sobre a, vamos a establecer el siguiente criterio de distribución de las iteraciones.

Regla 6.3.1 Sean $\{proc_1, proc_2\}$ los procesadores propietarios de las dos regiones de a a las que accede la iteración i del lazo irregular. Entonces dicha iteración únicamente puede ser ejecutada por el procesador $proc_1$ o por el procesador $proc_2$. □

Definición 6.3.1 Dada una iteración i del lazo irregular ejecutada por un procesador p, denominamos **acceso local** a aquel que se realiza sobre la partición de a cuyo propietario coincide con el procesador que ejecuta la iteración (procesador p). Adicionalmente, denominamos **acceso no local**, o **acceso remoto**, a aquel que se realiza sobre una partición de a cuyo propietario es diferente del procesador que ejecuta la iteración.

Más formalmente, asumiendo que el procesador p ejecuta la iteración j, tenemos las siguientes relaciones: $a[x_1[j]]$ es acceso local $\iff proc_1 = p$ y $a[x_2[j]]$ es acceso local $\iff proc_2 = p$□

Cada iteración j dada, puede ser clasificada de acuerdo con los valores de los procesadores propietarios de las dos entradas a las que accede. Concretamente, una iteración puede pertenecer a una de estas tres clases: *exclusiva*, *compartida-1* y *compartida-2*. Las características de cada una de ellas son las siguientes:

- **Iteración exclusiva.** Todas las iteraciones de esta clase realizan ambos accesos sobre la misma partición de a. En términos de la notación empleada, se verifica que ambos accesos son locales, es decir, $proc_1 = proc_2 = p$.

- **Iteración compartida-1.** Toda iteración perteneciente a esta clase realiza un acceso local mediante el vector x_1 y un acceso no local con x_2. Es decir, para un procesador p dado, se tiene que $proc_1 = p$ y $proc_2 \neq p$.

- **Iteración compartida-2.** Toda aquella iteración perteneciente a esta clase realiza un acceso local con x_2 y un acceso no local con x_1, de modo que $proc_1 \neq p$ y $proc_2 = p$.

Con el fin de obtener una alta localidad en los accesos proponemos una reordenación de los vectores de indirección del mismo modo que en el algoritmo básico de clasificación por *slices* (algoritmo `SLCCL`). Dado que cada iteración es ejecutada por un único procesador, ambos vectores de indirección deben de ser reordenados de manera solidaria. Supongamos por ejemplo que la iteración $i = 5$ se ejecuta (una vez reordenado el espacio de iteraciones) en primer lugar. Entonces tendremos que ambos vectores de indirección reordenados (x_1^{fin} y x_2^{fin}) verifican que $x_1^{fin}[1] = x_1[5]$ y $x_2^{fin}[1] = x_2[5]$. De esta forma, existe una dualidad entre una clasificación y reordenamiento de las iteraciones, respecto a una clasificación y reordenamiento de los vectores de indirección. A lo largo de esta sección nos referiremos indistintamente a una o a la otra.

El esquema de almacenamiento que proponemos está basado en una clasificación de las iteraciones en varias categorías, cada una de las cuales se subclasifica en diferentes clases. De este modo, establecemos un sistema de organización jerárquica de la información que consta de cuatro niveles y que describimos a continuación.

- **NIVEL 1:** *Procesador propietario.* El conjunto de N_x iteraciones del lazo son agrupados, en un primer nivel, en N_p conjuntos asociados a cada uno de los procesadores. De manera que todas las iteraciones pertenecientes a un mismo procesador están dispuestas en posiciones consecutivas de memoria.

- **NIVEL 2:** *Fase de ejecución.* Las entradas asignadas a un mismo procesador son clasificadas y agrupadas en tres subconjuntos en función de las particiones de a a las que accede. De este modo, tendremos un subconjunto de entradas exclusivas, otro subconjunto de entradas compartida-1 y otro subconjunto de entradas compartida-2.

- **NIVEL 3:** *Número de* slice. Aquellas entradas clasificadas como compartidas son nuevamente clasificadas en unos nuevos subconjuntos que denominamos *slices*.

- **NIVEL 4:** *Procesador propietario remoto.* Aquellas entradas pertenecientes a un *slice* son clasificadas en función del identificador del procesador propietario de la entrada accedida mediante el acceso remoto.

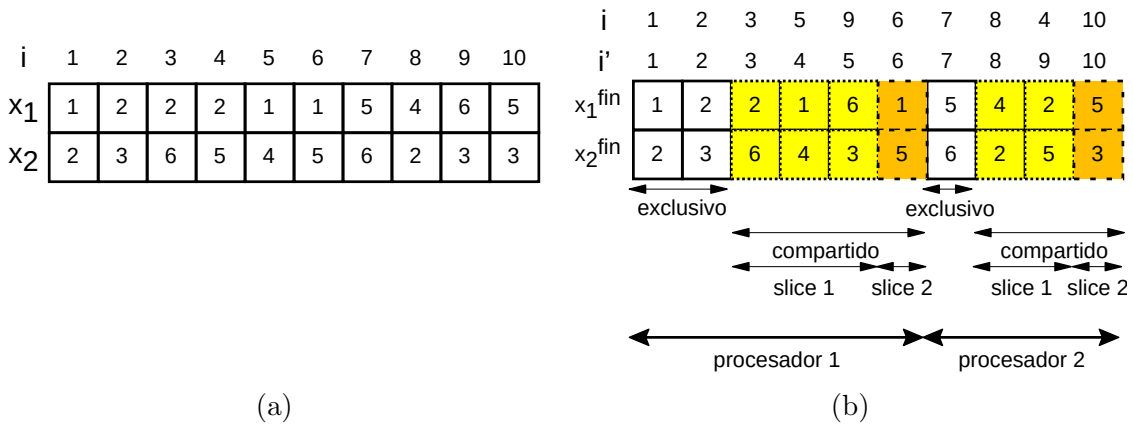


Figura 6.10: Ejemplo de vectores de indirección: (a) vectores originales, (b) vectores reordenados.

Es importante destacar que aquellas entradas pertenecientes a un mismo nivel están dispuestas en posiciones contiguas de memoria. La Figura 6.10(b) muestra la reordenación de los vectores de indirección correspondiente a la Figura 6.10(a) con $N_x = 10$, $N_a = 6$ y $N_p = 2$. Hemos considerado que $\mathcal{A}_1 = [1, 3]$ y $\mathcal{A}_2 = [4, 6]$.

Por ejemplo, el primer procesador tiene asignado el intervalo $[1, 6]$ de iteraciones reordenadas. Este intervalo se corresponde con el primer nivel de nuestra clasificación. Dentro de este conjunto, las entradas están agrupadas en dos subconjuntos. Por una parte, tenemos el intervalo $[1, 2]$ asociado a las iteraciones exclusivas (notar que en cada uno de los accesos de estas iteraciones se realiza dentro de la partición $\mathcal{A}_1$), y por otra, el segundo subconjunto se corresponde a aquellas entradas que, asignadas al primer procesador, son ejecutadas en una fase compartida (intervalo de entradas $[3, 6]$). Finalmente, las iteraciones compartidas son subclasificadas (en un tercer nivel) en conjuntos denominados *slices*. Como veremos posteriormente, empleamos este nivel de clasificación para eliminar los conflictos de acceso a los datos de las iteraciones compartidas. En nuestro caso, tenemos en un primer *slice* el intervalo de iteraciones $[3, 5]$, mientras que la sexta iteración pertenece al segundo *slice*. Una clasificación análoga puede realizarse para las iteraciones asignadas al segundo procesador. La Figura 6.10(b) muestra el resultado de clasificación del conjunto completo de iteraciones.

En un caso general, debemos de contemplar el cuarto nivel de nuestro esquema de clasificación. Es decir, las entradas compartidas pertenecientes al mismo *slice* (dentro del mismo procesador) aparecen subdivididas en función del procesador propietario del acceso remoto. A modo de ejemplo, la Figura 6.11 muestra el esquema general de distribución

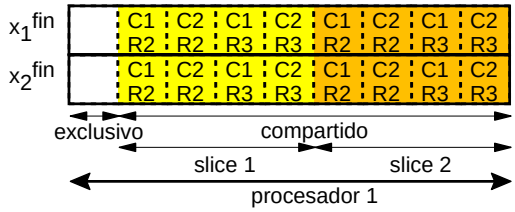


Figura 6.11: Distribución de las entradas del vector de indirección para el procesador 1 con $N_p = 3$.

de las entradas asignadas al procesador 1, con $N_p = 3$. En esta figura se denota como Ci al intervalo en entradas de la región compartida-i y como Ri al identificador del procesador propietario remoto. Por ejemplo, el intervalo C2R3 representa el conjunto de accesos asociados a una región compartida-2 cuyos accesos remotos (dados por x_1) acceden sobre el intervalo de a asociado al procesador 3. En la siguiente sección describimos la implementación del ejecutor paralelo y cómo este hace uso de este esquema de clasificación de las iteraciones.

6.3.2 Estructura del ejecutor

La Figura 6.12 muestra nuestra propuesta de ejecutor. De acuerdo con nuestro esquema de clasificación de las iteraciones, la ejecución paralela del lazo irregular es divida en 4 etapas denominadas *exclusiva*, *compartida-1*, *compartida-2* y etapa de *comunicación*.

En la **etapa exclusiva** cada procesador ejecuta aquellas iteraciones clasificadas como exclusivas. Dado que todos los accesos se realizan sobre la partición local de a, no es necesario el uso de operaciones de sincronización.

En las **etapas compartidas**, de acuerdo con su definición, uno de los accesos no se realiza sobre la partición local. En el caso de la etapa compartida-1, el acceso remoto es el dado por x_1, mientras que en el caso de la compartida-2 tendremos como remoto el dado por x_2. Debido a la existencia de este tipo de accesos, la ejecución de las iteraciones compartidas implica la posible existencia de conflictos de acceso a memoria. En este trabajo, proponemos como solución a este problema el empleo de un vector auxiliar de N_a entradas que denominaremos vector de **guarda** g. Mientras que el vector a se utiliza para almacenar el resultado final con las contribuciones de todos los procesadores, el vector g es utilizado para almacenar la contribución parcial de cada procesador. Por consiguiente, sobre a se realiza una operación de acumulación mediante una suma (del mismo modo que en el código original), mientras que sobre g el valor obtenido sobrescribe cualquier valor previamente almacenado. Durante la ejecución de las iteraciones compartidas, cada uno

```
SHARED a, g, ρ1^S, ρ2^S, x1, x2, ρ^excl

DOALL p = 1, Np                                          %Etapa exclusiva
   DO j = ρ1^s[p,1,1] − ρ^excl[p], ρ2^s[p,1,1] − 1
      fuerza = calcula_fuerza(posición(x1[j]), posición(x2[j]))
      a[x1[j]] += fuerza
      a[x2[j]] −= fuerza
   END DO
END DOALL

DO s = 1, NS
   DOALL p = 1, Np
      DO j = ρ1^s[p,s,1], ρ2^s[p,s,1] − 1                %Etapa compartida-1
         fuerza = calcula_fuerza(posición(x1[j]), posición(x2[j]))
         a[x1[j]] += fuerza
         g[x1[j]] = fuerza
      END DO
      DO j = ρ2^s[p,s,1], ρ1^s[p,s+1,1] − 1              %Etapa compartida-2
         fuerza = calcula_fuerza(posición(x1[j]), posición(x2[j]))
         g[x2[j] = fuerza
         a[x2[j]] −= fuerza
      END DO
   END DOALL
   BARRIER                                               %Sincronización

   DOALL p = 1, Np                                       %Etapa de recopilación
      DO k = 1, Np
         DO j = ρ1^s[k,s,p], ρ1^s[k,s,p+1] − 1
            a[x2[j]] −= g[x1[j]]
         END DO
         DO j = ρ2^s[k,s,p], ρ2^s[k,s,p+1] − 1
            a[x1[j]] += g[x2[j]]
         END DO
      END DO
   END DOALL
   BARRIER                                               %Sincronización
END DO
```

Figura 6.12: Algoritmo ejecutor SLCCLS.

de los procesadores accede a los vectores a y g de acuerdo con la siguiente regla:

Regla 6.3.2 Dada una iteración compartida j, y una vez calculada la fuerza existente entre los elementos asociados a las entradas $x_1[j]$ y $x_2[j]$, el valor obtenido se almacena tanto sobre a como sobre g en aquella posición de memoria accedida localmente. Es decir, en el caso de tratarse de una iteración compartida-1 el valor se almacena en las entradas $a[x_1[j]]$ y $g[x_1[j]]$, mientras que en el caso de ser una iteración compartida-2 el valor se almacena en las entradas dadas por $x_2[j]$. □

A modo de ejemplo, para el conjunto de datos de la Figura 6.10, el primer procesador ejecuta la tercera y cuarta iteración en la etapa compartida-1. Para la tercera iteración, y mediante nuestra propuesta, la fuerza resultante es almacenada en $a[2]$ y $g[2]$. Del mismo modo, en la cuarta iteración la fuerza calculada se almacena en $a[1]$ y $g[1]$. En el caso de la etapa compartida-2 aplicamos el mismo procedimiento. En el ejemplo de la figura, el primer procesador ejecuta la quinta iteración actualizando $a[3]$ y $g[3]$.

Mediante el empleo de esta estrategia de acceso se verifica la regla del propietario en las operaciones de escritura realizadas en las iteraciones compartidas. Cada procesador p únicamente accede a las particiones $\mathcal{A}_p$ y $\mathcal{G}_p$ que tiene asignadas[2], explotando eficientemente la localidad en los accesos.

Hay que resaltar que, a diferencia de a, el vector g sólo puede almacenar un único valor, por lo que el procedimiento anterior no puede continuar cuando una iteración procesada debe escribir sobre una entrada de g que ha sido previamente accedida. Con el fin de agrupar aquellas iteraciones que no causan conflictos, introducimos el concepto de *slice*, aplicado a nuestra estructura de ejecutor paralelo.

Definición 6.3.2 Un *slice* se define como un conjunto de iteraciones compartidas que pueden ser ejecutadas sin originar conflictos de accesos sobre el vector de guarda g. □

Entendemos por conflictos de acceso a la existencia de más de un acceso sobre la misma entrada de g. Esta definición mantiene la esencia del concepto de *slice*. En los casos anteriores considerábamos la existencia de conflictos de acceso sobre el vector a. Para la nueva estructura de ejecutor que proponemos, los conflictos de acceso están asociados al vector de guarda. De acuerdo con esta definición, podemos enunciar la siguiente propiedad.

Propiedad 6.3.1 Dadas dos iteraciones j y j' de un lazo irregular con la estructura mostrada en la Figura 6.1. Estas iteraciones pertenecen a un *slice* diferente si los accesos locales de cada una de ellas son sobre la misma entrada.

[2]Notar que el particionamiento y distribución del vector g es idéntico al realizado sobre el vector a.

PRUEBA: La demostración de esta propiedad es directa si se tiene en cuenta que las iteraciones compartidas deben ser procesadas aplicando la Regla 6.3.2 y la Definición 6.3.2. □

A modo de ejemplo, para el conjunto de datos de la Figura 6.10, las iteraciones $\{3, 4, 5, 8, 9\}$ forman un único *slice*, mientras que la iteración 6 debe ser clasificada en un *slice* diferente por realizar el mismo acceso local (sobre $a[1]$) que la cuarta iteración.

De acuerdo con la estructura del ejecutor y el esquema de clasificación empleado, es necesario asignar una serie de campos a cada una de las iteraciones compartidas. Específicamente, cada una de ellas debe tener asociada los siguientes elementos:

- **Campo1:** procesador propietario, definido en el intervalo $[1, N_p]$.
- **Campo2:** tipo de región compartida (compartida-1 o compartida-2).
- **Campo3:** *slice* al que pertenece la iteración, definido en el intervalo $[1, N_S]$.
- **Campo4:** procesador propietario de la partición sobre la que se realiza el acceso remoto. Tiene un rango de valores de 1 a N_p.

Con el fin de organizar toda esta información, y mantener la estructura de clasificación de las iteraciones, empleamos los vectores ρ_1^{slice} y ρ_2^{slice} para delimitar aquellas entradas que pertenecen a cada una de las clases. Ambos vectores constan de tres dimensiones que se corresponden a los campos 1, 3 y 4 del esquema de clasificación anterior. Concretamente obtenemos la siguiente disposición de campos:

$$\rho_1^{slice}[campo1, campo3, campo4] \tag{6.5}$$

El campo que denota el tipo de región compartida (segundo campo) está implícito en la propia definición del vector, dado que ρ_1^{slice} y ρ_2^{slice} están asociados a aquellas iteraciones ejecutadas, respectivamente, en las fases compartida-1 y compartida-2.

A modo de ejemplo, y considerando un caso arbitrario, la entrada $\rho_1^{slice}[4, 5, 3]$ apunta a la primera iteración asignada al procesador 4, ejecutada como compartida-1 en el *slice* número 5, y cuyo acceso remoto se realiza sobre una partición $\mathcal{A}_3$ que pertenece al procesador 3. La Figura 6.13 muestra los valores de ρ_1^{slice} y ρ_2^{slice} para el ejemplo de la Figura 6.10(a). Dado que $N_p = 2$, hemos ignorado la primera y segunda dimensión de estos vectores, y mostramos únicamente los valores almacenados para cada *slice* en cada uno de los procesadores.

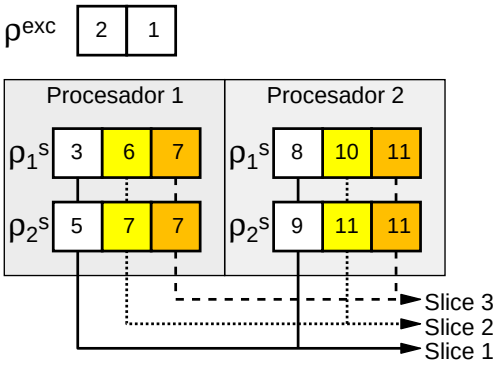


Figura 6.13: Ejemplo de valores de los vectores de densidad.

A continuación introducimos un vector de N_p entradas que denotamos como ρ^{excl}, el cual se emplea para almacenar el número de iteraciones que cada procesador ejecuta en la fase exclusiva. De acuerdo con el esquema de distribución de los datos empleado, para un procesador genérico p, el índice de la primera iteración compartida está almacenado en la entrada $\rho_1^{slice}[p,1,1]$. De acuerdo con el esquema de almacenamiento de nuestra propuesta, las iteraciones exclusivas son inmediatamente anteriores a las compartidas, por lo que el intervalo de entradas de la región exclusiva asociada a un procesador p está especificado por el intervalo:

$$[\rho_1^{slice}[p,1,1] - \rho^{excl}[p], \rho_1^{slice}[p,1,1]) \tag{6.6}$$

Para el ejemplo de la Figura 6.10(a), tenemos el vector de densidades mostrado en la Figura 6.13, el cual origina los siguientes intervalos de entradas exclusivas: intervalo [1,3) asociado al procesador 1 e intervalo [7,8) asociado al procesador 2.

La distribución de los datos en memoria asociada a las entradas compartidas se realiza de la siguiente forma: para un *slice* dado, se almacena el intervalo de entradas de la región compartida-1 y a continuación el intervalo de entradas de la región compartida-2. A continuación, es almacenado el intervalo de entradas de la región compartida-1 correspondiente al siguiente *slice*, etc.

Las iteraciones pertenecientes a las regiones compartidas son ejecutadas en función del *slice* al que pertenecen. De acuerdo con la Regla 6.3.2, todas las iteraciones compartidas que pertenecen a un mismo *slice* pueden ser ejecutadas en paralelo de forma asíncrona. Para un *slice* genérico, este conjunto de iteraciones consiste en la unión de los intervalos asociados a cada una de las regiones compartidas del mismo *slice*. Más formalmente,

$$[\rho_1^{slice}[p,s,1], \rho_2^{slice}[p,s,1]) \cup [\rho_2^{slice}[p,s,1], \rho_1^{slice}[p,s+1,1]) \tag{6.7}$$

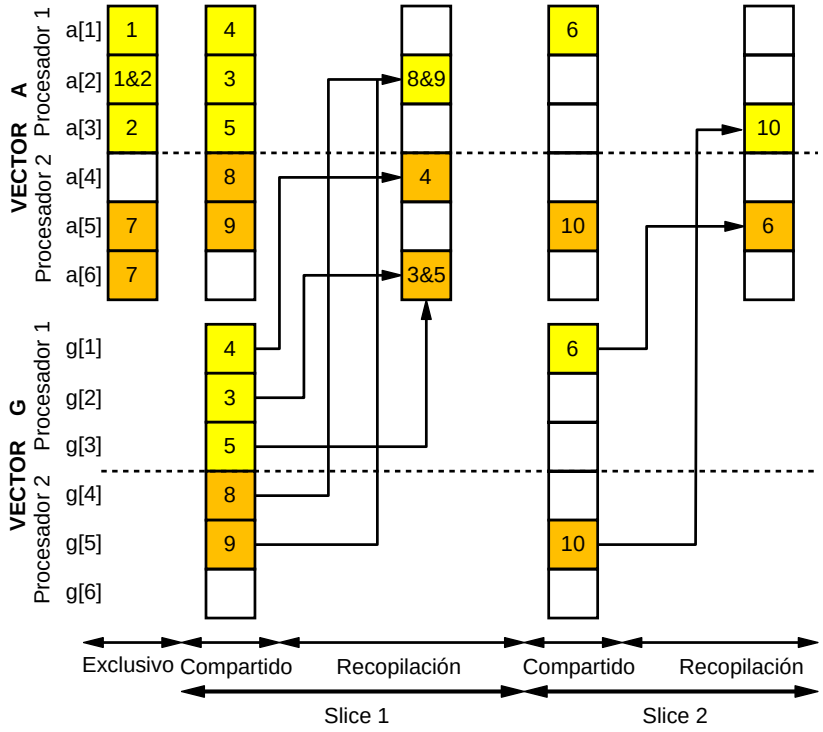


Figura 6.14: Diagrama de acceso a los vectores a y g.

Nuevamente, considerando el ejemplo de la Figura 6.13, los intervalos asociados al primer *slice* del primer procesador son $[3,5) \cup [5,6) = [3,5]$. Notar que, el número de *slices* asociado a los vectores ρ^{slice} es igual al número "real" de *slices* N_S, más la unidad: $N_S + 1$. El motivo de este aumento es mantener la coherencia a la hora de especificar el intervalo de entradas asociado al último *slice* (*slice* N_S), dado que el último elemento de este intervalo hace siempre referencia al *slice* siguiente.

Una vez ejecutadas las etapas compartidas, el resultado final se obtiene en la **fase de comunicación**. En esta fase cada procesador recorre las entradas de g escritas por otro procesador y asociadas a un acceso local. Por medio de los vectores ρ^{slice} es posible determinar exactamente la posición de estas entradas. De forma más concreta, cada procesador p debe considerar la contribución del procesador k en el siguiente intervalo de entradas de g.

$$[\rho_1^{slice}[k,s,p], \rho_1^{slice}[k,s,p+1]) \cup [\rho_2^{slice}[k,s,p], \rho_2^{slice}[k,s,p+1]) \tag{6.8}$$

Dicho de otro modo, el intervalo dado por 6.8 se corresponde con el conjunto de iteraciones ejecutadas por el procesador k y que tienen un acceso remoto sobre el procesador p.

Una vez que el vector a es actualizado, el contenido de g puede ser borrado, permitiendo su reuso durante el procesamiento del siguiente *slice*.

La Figura 6.14 muestra el diagrama de acceso a los vectores a y g para el conjunto de datos mostrado en la Figura 6.10. El conjunto de la ejecución de este problema tiene tres etapas: la ejecución de las entradas exclusivas, y la ejecución del primer y segundo *slice*. Dentro de estas últimas, hemos agrupado las sub-etapas compartida-1 y compartida-2 en una única etapa, que denominamos compartida. Los accesos realizados en cada una de las etapas aparecen oscurecidos. Nótese que, a lo largo de toda la ejecución paralela del lazo, los vectores a y g son escritos por cada procesador únicamente en la partición que tiene asignada, es decir, se aplica la regla del propietario en las operaciones de escritura sobre a y g.

6.3.3 Estructura del inspector

En esta sección describimos el proceso de obtención de la estructura de datos propuesta y presentada la sección previa. La Figura 6.15 muestra el pseudocódigo de nuestro inspector, denominado *Slice Classification 2* (`SLCCLS`). Dicho inspector consta de tres etapas denominadas *análisis*, *desplazamiento* y *reordenación*.

En la **fase de análisis** todas las entradas del vector de indirección son clasificadas en cada uno de los cuatro niveles anteriormente descritos. Mediante vectores auxiliares se almacena el número de iteraciones asociadas a cada una de las distintas clases. Adicionalmente, durante el proceso de clasificación empleamos las funciones *extrae_propietario* y *scheduler* para determinar el procesador que ejecutará cada una de las iteraciones compartidas.

La primera de estas funciones, introducida en la Sección 6.3.1, se utiliza para distinguir aquellas iteraciones exclusivas de las compartidas. La comprobación que hay que realizar es simple, dado que únicamente es necesario comprobar la coincidencia entre los procesadores propietarios asociados a ambos accesos. En el caso de verificarse esta igualdad tendremos una iteración exclusiva, por lo que incrementaremos en una unidad la entrada correspondiente de ρ^{excl} . En el caso de que la iteración tenga asociada distintos propietarios, empleamos la función *scheduler* para asignarla a uno de ellos. La función *scheduler* devuelve dos valores, denominados $\{proc, s\}$. El primero de ellos identifica el procesador al que la iteración es asignada. En caso de un valor igual a $proc_1$, la iteración es clasificada como compartida-1, por lo que el vector al que queda asociada es ρ_1^{slice}. En caso contrario, la iteración es clasificada como compartida-2 y será asignada al vector ρ_2^{slice}. El segundo valor devuelto por la función *scheduler* contiene el número de *slice* al que la iteración es

```
Algoritmo SLCCLS
entrada
      {x_1, x_2}: vectores de indirección
salida
      {x_1^fin, x_2^fin}: vectores de indirección reordenados
      {ρ^excl, ρ_1^s, ρ_2^s}: vectores de densidad
inicio del algoritmo
L1    DO j = 1, N_x                                          %Fase de análisis
         {proc_1, proc_2} = extrae_propietario(x_1[j], x_2[j], N_a, N_p)
         IF (proc_1 = proc_2)
            ρ^excl[proc_1] + +
         ELSE
               {proc, s} = scheduler(x_1[j], x_2[j], proc_1, proc_2)
               IF (proc = proc_1)
                  ρ_1^s[proc_1, s, proc_2] + +
               ELSE
                  ρ_2^s[proc_2, s, proc_1] + +
               END IF
         END IF
      END DO
      {ρ_1^s, ρ_2^s} = desplazamiento(ρ_1^s, ρ_2^s)          %Fase de desplazamiento
L2    DO j = 1, N_x                                          %Fase de reordenación
         {proc_1, proc_2} = extrae_propietario(x_1[j], x_2[j], N_a, N_p)
         IF (proc_1 = proc_2)
            x_1^fin[ρ_1^s[1,1,1] − ρ^excl[proc_1] + cnt^excl[proc_1]] = x_1[j]
            x_2^fin[ρ_1^s[1,1,1] − ρ^excl[proc_1] + cnt^excl[proc_1]] = x_2[j]
            cnt^excl[proc_1] + +
         ELSE
               {proc, s} = scheduler(x_1[j], x_2[j], proc_1, proc_2)
               IF (proc = proc_1)
                  x_1^fin[ρ_1^s[proc_1, s, proc_2] + cnt_1^s[proc_1, s, proc_2]] = x_1[j]
                  x_2^fin[ρ_1^s[proc_1, s, proc_2] + cnt_1^s[proc_1, s, proc_2]] = x_2[j]
                  cnt_1^s[proc_1, proc_2, s] + +
               ELSE
                  x_1^fin[ρ_2^s[proc_2, s, proc_1] + cnt_2^s[proc_2, s, proc_1]] = x_1[j]
                  x_2^fin[ρ_2^s[proc_2, s, proc_1] + cnt_2^s[proc_2, s, proc_1]] = x_2[j]
                  cnt_2^s[proc_2, s, proc_1] + +
               END IF
         END IF
      END DO
fin del algoritmo
```

Figura 6.15: Algoritmo inspector SLCCLS.

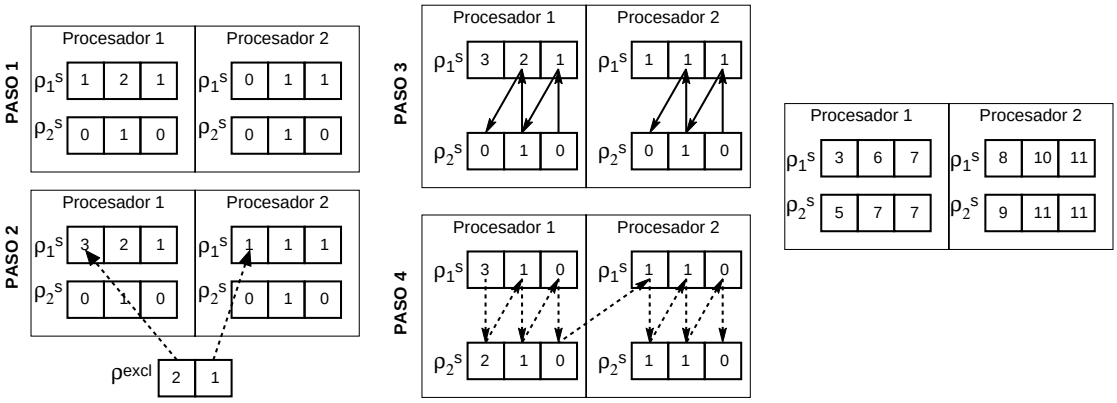


Figura 6.16: Ejemplo de fase de desplazamiento.

asignada. De este modo, obtenemos toda la información (valor del procesador propietario, valor del procesador asociado al acceso remoto e índice de *slice*) para incrementar la entrada correspondiente del vector ρ_1^{slice} o ρ_2^{slice}. En la siguiente sección realizaremos una descripción detallada de la estructura y funcionamiento de la función *scheduler*, por lo que postergaremos su descripción hasta entonces.

La **fase de desplazamiento** se utiliza para realinear los vectores de densidad de modo que almacenen la posición de la entrada reordenada asociada al comienzo de cada uno de los *slices*. Esta fase se muestra en el pseudocódigo en la función `desplazamiento`, la cual consta de una serie de pasos que se resumen a continuación. La Figura 6.16 muestra un ejemplo de cada uno de ellos para el conjunto de datos considerado.

- **Paso 1:** todas las entradas de ρ_1^{slice} y ρ_2^{slice} son desplazadas una posición. La primera entrada de ρ_1^{slice} es incrementada en una unidad.

- **Paso 2:** para todo procesador p, con $1 \leq p \leq N_p$, la entrada $\rho_1^{slice}[p, 1, 1]$ es incrementada por el valor almacenado en ρ^{excl}.

- **Paso 3:** Todas las entradas de ρ_1^{slice} y ρ_2^{slice} son recorridas de forma ordenada y desplazadas de acuerdo con:

  - Para cada *slice* $s \neq 1$ y cada procesador $1 \leq p \leq N_p$, $\rho_1^{slice}[p, s, 1]$ es reemplazado por $\rho_2^{slice}[p, s, N_p]$.
  - Para cada *slice* $s \neq N_s$ y cada procesador $1 \leq p \leq N_p$, $\rho_2^{slice}[p, s, 1]$ es reemplazado por $\rho_1^{slice}[p, s + 1, N_p]$.

- **Paso 4:** Nuevamente se recorren todas las entradas de ρ_1^{slice} y ρ_2^{slice} con el siguiente orden de acceso: procesador propietario, número de *slice* y procesador remoto. En esta operación las entradas de estos vectores son sumadas y acumuladas.

En la última fase, denominada **fase de reordenación**, se generan los nuevos vectores de indirección. La estructura de esta fase es idéntica a la de análisis, de modo que partiendo de unas condiciones iniciales iguales, cada iteración del lazo recibe la misma clasificación que la realizada en la fase previa. La diferencia es que ahora los vectores ρ_1^{slice} y ρ_2^{slice} han sido realineados, y se emplean para apuntar al primer elemento de cada clasificación. En esta fase empleamos los vectores cnt^{excl}, cnt_1^{slice} y cnt_2^{slice} para almacenar el número de entradas asignadas a cada una de las fases. De este modo, cada una de las entradas de los vectores de indirección es correctamente copiada sobre los vectores reordenados. Hay que destacar que los valores finales de cnt^{excl}, cnt_1^{slice} y cnt_2^{slice} coinciden, respectivamente, con los obtenidos en la fase de análisis para ρ^{excl}, ρ_1^{slice} y ρ_2^{slice}.

6.3.4 Estructura del *scheduler*

La función *scheduler* se utiliza para determinar el procesador propietario de cada iteración compartida. Adicionalmente, también determina el número de *slice* al que la iteración queda asignada.

Los criterios empleados para realizar la distribución de las iteraciones tiene en cuenta la maximización de la localidad en los accesos y el mantenimiento de un correcto balanceo de carga. En el proceso de distribución realiza una clasificación por *slices* de las entradas compartidas. Para aplicar este proceso introducimos una nueva estructura de almacenamiento que denominados **tabla de carga** y denotamos como *carga*. Esta tabla consta de dos dimensiones correspondientes al índice de procesador y número de *slice* asociado (p y s respectivamente). De forma más concreta, la entrada $carga[p][s]$ almacena la carga computacional asociada al procesador p en el *slice* s. En este trabajo, vamos a utilizar como unidad de carga computacional el coste de una iteración.[3] De este modo, en la tabla de carga se almacena el número de iteraciones asociado a cada procesador en cada uno de los *slices*.

La estructura del algoritmo de *scheduler* se muestra en la Figura 6.17. Inicialmente, cada iteración j realiza un acceso a las posiciones $x_1[j]$ y $x_2[j]$ de a. De acuerdo con la

[3]Cabe realizar varias observaciones: esta aproximación puede realizarse por haber sido utilizado un paralelismo de grano iteración. Adicionalmente, la aproximación será más exacta cuanto más próxima sea la carga computacional asociada a las distintas iteraciones. Finalmente, estamos asumiendo, por simplicidad, que el poder computacional de todos los procesadores es el mismo.

```
Algoritmo SCHEDULER
entrada
    {x1[j], x2[j]}: vectores de indirección procesados
    {proc1, proc2}: procesadores propietarios de acceso de x1[j] y x2[j]
salida
    {owner, s}: Identificado del propietarios y slice asociado
inicio del algoritmo
    s1 = ρ^a[x1[j]] + 1
    s2 = ρ^a[x2[j]] + 1
    IF (s1 = s2)
        {proc, k} =extrae_carga_mínima(carga, proc1, proc2, s1, s2)
    ELSE IF (s1 < s2)
        IF (carga[proc1, s1] = carga_máxima(carga, s1))
            {proc, k} =extrae_carga_mínuma(carga, proc1, proc2, s1, s2)
        ELSE
            {proc, k} = {proc1, x1[j]}
        END IF
    ELSE
        IF (carga[proc2, s2] = carga_máxima(carga, s2))
            {proc, l} =extrae_carga_mínuma(carga, proc1, proc2, s1, s2)
        ELSE
            {proc, k} = {proc2, x2[j]}
        END IF
    END IF
    ρ^a[k] + +
    carga[proc, ρ^a[k]] + +
    return({proc, ρ^a[k]})
fin del algoritmo
```

Figura 6.17: Algoritmo *scheduler*.

Regla 6.3.1, vamos a establecer que para esta iteración, los procesadores $proc_1$ y $proc_2$ son los únicos candidatos para ejecutarla. El identificador concreto de estos procesadores se obtiene por medio de la función *extrae_propietario* descrita en la Sección 6.3.1. Si $proc_1$ es el procesador elegido, entonces $x_1[j]$ será el acceso local y $x_2[j]$ el remoto. En caso contrario, tendremos a $proc_2$ como el procesador propietario, $x_1[j]$ como acceso remoto y $x_2[j]$ como acceso local.

Introducimos el vector $\rho^a[k]$ como el que almacena el número de accesos locales sobre la k-ésima entrada de a. Este vector se emplea para obtener el número de *slice* asignado a cada iteración compartida. Se aplica la siguiente regla de obtención del número de *slice*.

Regla 6.3.3 El número de *slice* asociado a una iteración compartida i, con un acceso local $x[j]$, viene dado por $\rho^a[x[j]] + 1$. □

De este modo, si el procesador $proc_1$ es asignado como propietario de la iteración i, entonces el *slice* asociado a dicha iteración vendrá dado por $\rho^a[x_1[j]]+1$. En caso contrario, el valor del *slice* será $\rho^a[x_2[j]] + 1$. Nótese que, debido al empleo del vector de guarda, el acceso remoto no tiene repercusión en las fases compartidas del ejecutor, ya que es en la fase de comunicación donde se resuelven este tipo de accesos.

En nuestra propuesta, empleamos el siguiente conjunto de reglas para la determinación del propietario de cada iteración compartida:

Regla 6.3.4 Si ambos candidatos ejecutan la iteración en el mismo *slice*, entonces el procesador propietario es aquel que tiene la menor carga de trabajo. □

Regla 6.3.5 En caso de no verificarse la Regla 6.3.4, el procesador propietario es aquel que asigna el menor *slice* a la iteración considerada. □

Regla 6.3.6 Existe una excepción en la Regla 6.3.5 para el caso de que el procesador propietario tenga la mayor carga computacional del *slice*. En ese caso, el propietario es aquel que tenga una menor carga. □

Para evaluar la Regla 6.3.4 empleamos la función *extrae_mínima_carga*, la cual selecciona el menor valor de la tabla de carga. Es decir,

$$\begin{aligned} extrae\_minima\_carga = \{proc_1, x_1[j]\} \quad & si \quad (carga[proc_1, s_1] \leq carga[proc_2, s_2]) \\ \{proc_2, x_2[j]\} \quad & si \quad (carga[proc_1, s_1] > carga[proc_2, s_2]) \end{aligned} \tag{6.9}$$

La función *máxima_carga* se emplea para seleccionar el valor de la carga máxima de un *slice*. La estructura de esta función es:

$$máxima\_carga(carga, s) = max(carga[1, s], carga[2, s] \ldots carga[N_p, s]) \tag{6.10}$$

La Figura 6.18 muestra, para los vectores de indirección del ejemplo de la Figura 6.10, el valor del vector ρ^a y de los propietarios de cada iteración compartida. Adicionalmente, se muestra el contenido final de la tabla de carga asociada a este ejemplo.

Con este tópico finalizamos la descripción de los distintos elementos que componen nuestra propuesta. A continuación vamos a realizar, en la siguiente sección, un análisis de la eficiencia obtenida con ella.

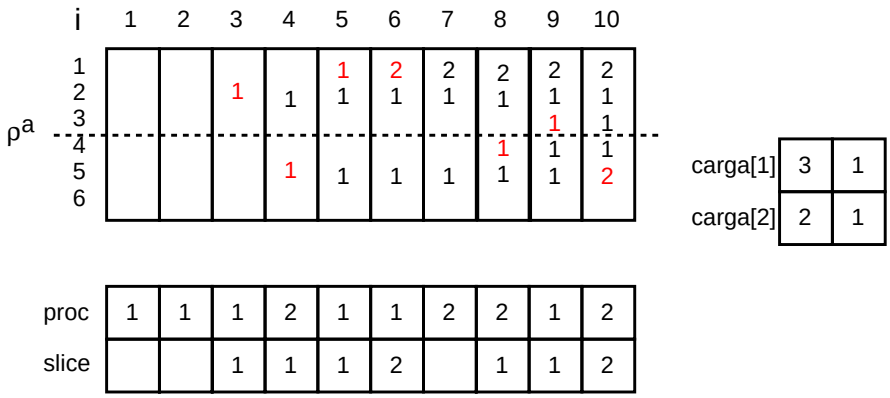


Figura 6.18: Ejemplo de funcionamiento del *scheduler*.

6.3.5 Resultados

En esta sección presentamos los resultados correspondientes a las distintas pruebas de evaluación del rendimiento que hemos realizado. Inicialmente, evaluaremos la complejidad máxima del inspector y el coste de memoria del ejecutor paralelo. Posteriormente verificamos la capacidad de extracción de paralelismo de la estrategia `SLCCLS`, comparando su eficiencia con la obtenida mediante la estrategia `SLCCL`. Finalmente, emplearemos nuestra propuesta para la paralelización de una aplicación de simulación de n-cuerpos, y los resultados obtenidos se compararán con el de otras estrategias de paralelización.

Consideraciones teóricas

Para evaluar la complejidad del inspector `SLCCLS` es necesario considerar la complejidad del *scheduler*, dado que esta función es invocada cada vez que se procesa una entrada compartida. La complejidad del *scheduler* únicamente tiene asociado el coste del acceso a la tabla de carga. Concretamente,

$$\mathcal{O}_{scheduler} = N_p \tag{6.11}$$

Considerando ahora la complejidad del inspector. En el caso más desfavorable, el lazo irregular únicamente consta de iteraciones compartidas. Para esta situación,

$$\mathcal{O}_{\texttt{SLCCLS}} = N_x\mathcal{O}_{scheduler} + 6N_s + N_x\mathcal{O}_{scheduler} = N_xN_p + 6N_s + N_xN_p \tag{6.12}$$

Dicha expresión tiene en cuenta las contribuciones de cada una de las tres fases que comprenden el inspector.

Hemos evaluado el coste de memoria del ejecutor, el cual se define como la diferencia entre la memoria requerida por el código original y el ejecutor paralelo. En este último caso únicamente debemos considerar el coste de almacenamiento del vector de guarda y de los vectores ρ_1^s y ρ_2^s. De este modo, el coste de memoria resulta:

$$\Delta M_{\texttt{SLCCLS}}^{ejecutor} = N_a + 2N_p^2 N_S \tag{6.13}$$

Evaluación del grado de paralelismo extraído

Las tablas 6.2 y 6.3 muestran, respectivamente, el número de *slices* obtenidos con la estrategia `SLCCLS` para $N_p = 4$ y $N_p = 64$. Estos resultados pueden compararse con los obtenidos con la estrategia `SLCCL` de la Tabla 6.1. Podemos apreciar una importante reducción en número de *slices* respecto al caso anterior. Además, podemos apreciar que conforme aumenta el número de procesadores el número de *slices* se mantiene constante e incluso experimenta una pequeña disminución. En base a estos resultados podemos afirmar que nuestra nueva propuesta resuelve uno de los principales inconvenientes del algoritmo `SLCCL`, que es la poca capacidad de extracción de paralelismo.

Paralelización de una aplicación de n-cuerpos

Hemos aplicado la estrategia `SLCCLS` a la paralelización de una aplicación de simulación de dinámica molecular. Específicamente utilizamos el *Steve Plimpton's Lennard-Jones benchmark* [108], en el que se simula las interacciones de un conjunto de partículas

Matriz	*bcsstk14*	*beaflw*	*psmigr_2*	*25600_90*	*25600_U*
N_S	20	190	133	12	5

Tabla 6.2: Eficiencia de la clasificación por *slices* para $N_{stmt} = 2$ y $N_p = 4$ con distintos patrones de acceso.

Matriz	*bcsstk14*	*beaflw*	*psmigr_2*	*25600_90*	*25600_U*
N_S	18	100	154	9	3

Tabla 6.3: Eficiencia de la clasificación por *slices* para $N_{stmt} = 2$ y $N_p = 64$ con distintos patrones de acceso.

```
DO  paso_tiempo = 1, N_t
    DO  k = 1, N_cuerpos
        DO j = ρ_interacc[k], ρ_interacc[k+1] − 1
            fuerza = calcula_fuerza(posición[k], posición[x_1[j]])
            a[k]+ = fuerza
            a[x_1[j]]− = fuerza
        END DO
    END DO
    posición = actualiza_posición(a)
END DO
```

Figura 6.19: Lazo irregular de n-cuerpos.

sometidas a un potencial de Lennard-Jones. Esta aplicación consiste en diversas implementaciones en las que se realizan distintos tipos de descomposiciones del problema. En nuestro caso nos hemos centrado en la paralelización del código denominado LJA, en el que se realiza una estrategia de descomposición por átomos. El núcleo de este código de pruebas tiene la estructura del código mostrado en la Figura 6.19. La principal diferencia es que el lazo interno aparece ahora dividido en dos lazos anidados. El primero de ellos (índice k) recorre cada uno de los cuerpos simulados. Para cada uno de estos cuerpos, y mediante el vector $\rho_{interacc}$, el lazo más interno (índice j) recorre las iteraciones que dicho cuerpo tiene asociadas. Adicionalmente se verifica que $\rho_{interacc}[j+1] \geq \rho_{interacc}[j]$. Es decir, el índice j es siempre monótono creciente. El vector de indirección x_2 puede obtenerse de una forma sencilla a partir de los valores de $\rho_{interacc}$. De este modo, el código original puede ser fácilmente convertido en un esquema de indireccionamiento como el mostrado en la Figura 6.1.

Con el fin de comparar la eficiencia de nuestra propuesta con el de otras estrategias, hemos desarrollado distintas versiones paralelas del código LJA. Concretamente, hemos implementado la técnica *array expansion*, LOCAL-WRITE, DWA-LIP[4] y nuestra propuesta denominada `SLCCLS`. La plataforma empleada es un multiprocesador Silicon Graphics Origin 2000, el lenguaje de programación es el Fortran 77 en conjunción con directivas OpenMp. El compilador utilizado es el MIPSpro `f77` v7.4. En la compilación de todos los programas se emplearon las opciones "`-aling128`". La estrategia *array expansion* no utiliza inspector, y puede ser directamente aplicada al código fuente mostrado en la Figura 6.19. Las propuestas LOCAL-WRITE, DWA-LIP y `SLCCLS` emplean un inspector que

[4]Respecto a la técnica DWA-LIP, hemos implementado la técnica básica ofrecida por el autor [54].

analiza tanto x_1 como x_2, por lo que únicamente pueden ser utilizados con el código de la Figura 6.1. Existe una variante de la técnica DWA-LIP que permite ser directamente aplicado sobre el código original. Sin embargo, el rendimiento de esta variante resulta sensiblemente inferior al de la propuesta que hace uso de x_1 y x_2. Por este motivo, decidimos utilizar la primera de ellas.

Todas las estrategias, con la excepción de *array expansion*, están basadas en la aplicación de la regla del propietario sobre el vector a. En estos casos, realizamos una distribución de a por bloques de igual tamaño. En el caso de *array expansion*, y con el fin de obtener un correcto balanceo de la carga, realizamos una distribución por bloques del espacio de iteraciones del problema.

El código de prueba LJA permite especificar el número de cuerpos del problema que se desea simular y el grado medio de conectividad (número de vértices dividido por el número de nodos) existente en el sistema simulado. El primer parámetro se corresponde con N_a, la dimensión del vector en el que se almacenan las fuerzas a las que cada uno de los cuerpos se ve sometido. El segundo parámetro es una estimación del número medio de interacciones que sufre cada uno de los cuerpos, el cual es proporcional a N_x/N_a. En este trabajo hemos intentado cubrir el mayor espacio posible de situaciones. Por este motivo, hemos considerado tres escenarios diferentes cuyas principales características se muestran en la Tabla 6.4. El primero de ellos, denominado *prueba*1, se corresponde a un problema con una conectividad media de entorno a 10, y un número de cuerpos de 97k. Por otra parte, los otros dos escenarios se corresponden a situaciones más extremas. Así pues, el problema denominado *prueba*2, representa una situación con un alto grado de conectividad, un bajo grado de dispersión, y un reducido número de cuerpos. Por otra parte, el escenario *prueba*3 presenta un escenario con un bajo grado de conectividad, un alto grado de dispersión, y un gran número de cuerpos.

Un tercer parámetro asociado a cada escenario es la localidad del patrón de acceso. Este parámetro indica la separación "espacial" entre los accesos realizados dentro de cada iteración. Al comienzo de la ejecución del código LJA existe un alto grado de localidad. Sin embargo, esta localidad disminuye conforme aumenta el número de iteraciones, dado

Escenario	N_a	N_x	Conectividad
*prueba*1	$97K$	$1008K$	10.4
*prueba*2	$32K$	$1202K$	37.6
*prueba*3	$97K$	$297K$	3.1

Tabla 6.4: Características de los patrones de acceso.

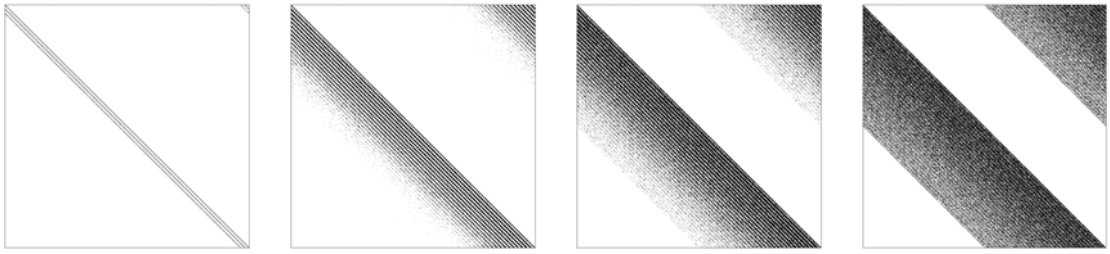
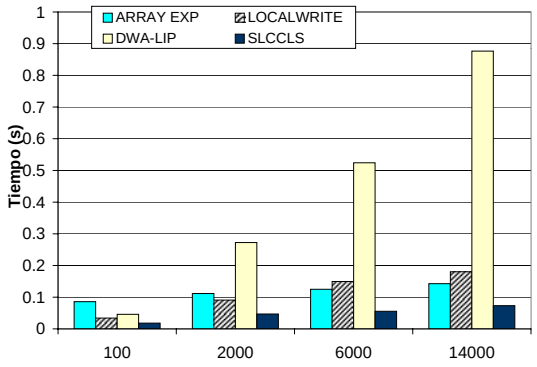


Figura 6.20: Patrón de acceso a memoria para el escenario *prueba3*.

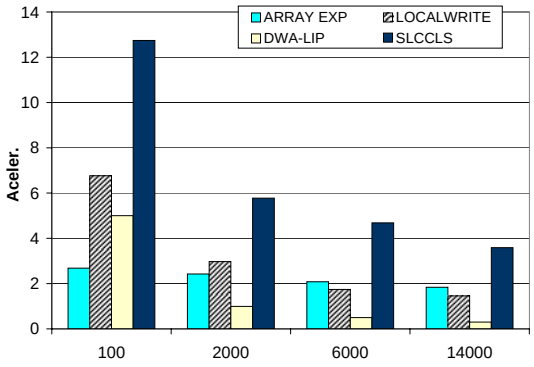
que la posición de los cuerpos se va modificando y surgen interacciones con vecinos cuyas posiciones de memoria son más distantes (es decir, menos locales). En este trabajo hemos considerado diferentes grados de localidad del patrón de acceso, almacenando su estructura a lo largo de la ejecución del programa. Concretamente, hemos elegido las iteraciones 100, 2000, 6000 y 14000 cuyo rango entendemos que abarca un amplio espectro de grados de localidad. La Figura 6.20 muestra los diferentes patrones de acceso para el escenario *prueba*3. Hemos constatado experimentalmente que el número entradas no nulas del patrón de acceso (es decir, el número de interacciones entre los cuerpos) es aproximadamente el mismo a lo largo de toda la ejecución del programa. Gracias a esta propiedad, podemos asumir que el parámetro N_x no varía para todo el rango de iteraciones.

La Figura 6.21 muestra el tiempo de ejecución y las aceleraciones obtenidas con el ejecutor cuando se emplean 32 procesadores. En el eje de categorías se representa el número de iteraciones asociado a cada uno de los cuatro grados de localidad que hemos considerado.

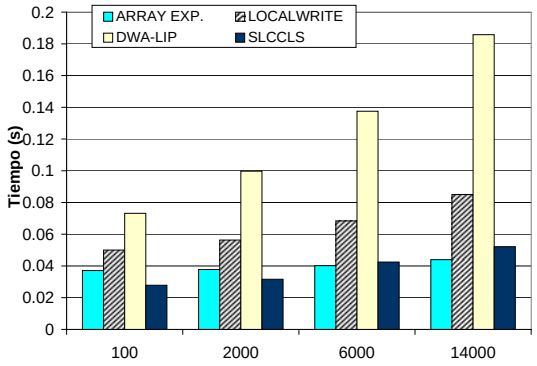
La estrategia DWA-LIP presenta un buen rendimiento para problemas dispersos con una alta localidad en los accesos. La principal desventaja de esta técnica aparece en el tratamiento de las iteraciones compartidas, es decir, en aquellas en las que ambos accesos se realizan sobre particiones distintas de a. Para este tipo de iteraciones, el ejecutor DWA-LIP presenta un bajo rendimiento debido a que el número máximo de procesadores que es posible utilizar disminuye conforme avanza la ejecución paralela del programa. Esta disminución se hace más notable cuanto mayor es el número de iteraciones compartidas. Podemos apreciar que el rendimiento de esta propuesta decrece conforme disminuye la localidad del patrón de acceso, dado que en estos casos aumenta la proporción de en-



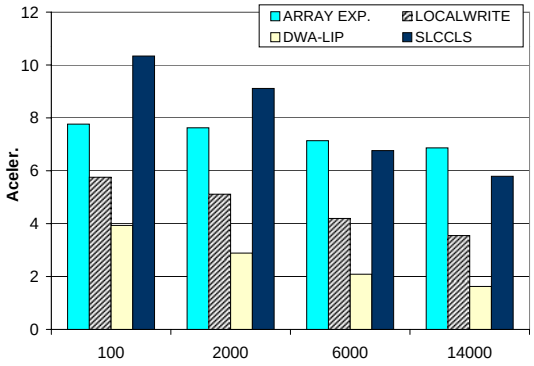
Tiempo ejecución *prueba*1



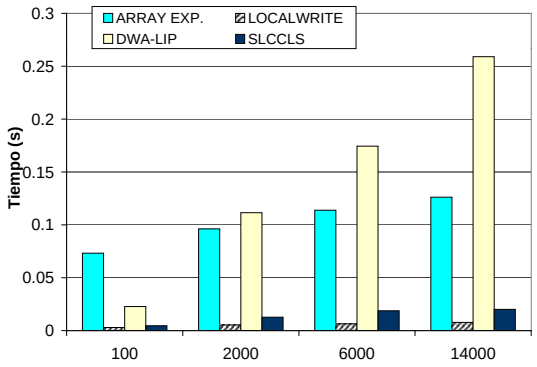
Aceleración *prueba*1



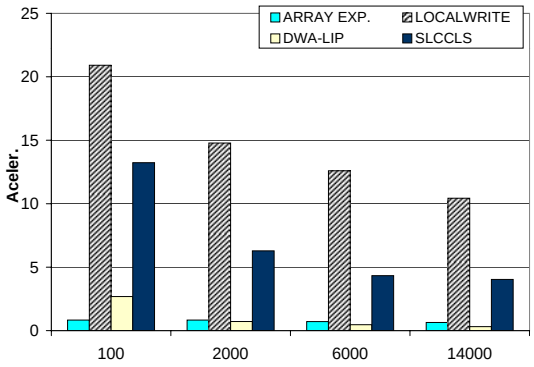
Tiempo ejecución *prueba*2



Aceleración *prueba*2



Tiempo ejecución *prueba*3



Aceleración *prueba*3

Figura 6.21: Tiempos de ejecución y aceleraciones con $N_p = 32$.

tradas compartidas. Adicionalmente, la estrategia DWA-LIP emplea un vector adicional de N_x entradas cuyo coste de almacenamiento es importante, y puede llegar a afectar al rendimiento en los accesos a memoria cache.

La técnica *array expansion* presenta como principal inconveniente el coste de las rutinas de comunicación asociado a la operación final de reducción. Adicionalmente, existen dos factores que influyen de forma importante sobre el rendimiento de código paralelo. Por una parte, conforme aumenta el tamaño del vector a, el coste de almacenamiento del ejecutor y el coste de la operación de reducción aumentan, lo cual produce una disminución del rendimiento. Por otra parte, cuando decrece el número de iteraciones del lazo más interno (lazo j), el coste de comunicación tiene un mayor peso en la ejecución paralela, influyendo negativamente sobre la eficiencia de esta propuesta. La técnica *array expansion* obtiene los mejores resultados para el escenario *prueba*2, en el que la localidad de los accesos es baja. Nótese que el rendimiento de esta técnica no se ve influido de forma importante por las características del patrón de acceso. Una importante ventaja de esta propuesta es que, debido a la política de distribución por bloques de las iteraciones, se consigue un alto grado de balaceo de la carga. Sin embargo, este hecho implica otro de los grandes inconvenientes de esta propuesta, que es la baja explotación de la localidad en los accesos sobre la matriz a.

Con la estrategia LOCAL-WRITE cada procesador únicamente accede a la partición de a que tiene asignado, por lo que la localidad en los accesos a este vector es explotada adecuadamente. Sin embargo, esta propuesta tiene como desventaja una replicación en las computaciones. Esta replicación es mayor cuanto menor sea la localidad en el patrón de acceso, teniendo un gran impacto sobre el rendimiento de la propuesta. Para el escenario *prueba*3 la estrategia LOCAL-WRITE obtiene los mejores resultados, debido a que la conectividad del problema es reducida y, consiguientemente, existe una baja replicación de las computaciones. El rendimiento de la estrategia LOCAL-WRITE depende fuertemente del coste computacional de cada iteración. Cuando este aumenta, el coste de las replicaciones también lo hace, disminuyendo su eficiencia.

A continuación vamos a realizar un análisis del rendimiento del ejecutor paralelo presentado en nuestra propuesta. Este análisis puede dividirse en dos etapas: la ejecución de las entradas exclusivas y la ejecución de las entradas compartidas. La primera de ellas es altamente paralela, los procesadores trabajan totalmente desacoplados. En esta etapa cada procesador escribe sobre la partición de a que tiene asignada, y accede a entradas consecutivas de x_1 y x_2. De este modo, no existe compartición de líneas cache (salvo las existentes en el límite de cada bloque de a y g) y el grado de localidad en los accesos es elevado.

En la segunda etapa se realiza el procesamiento de las iteraciones compartidas y la fase de comunicación final. En la primera parte, nuevamente se accede a entradas consecutivas de los vectores de indirección, y únicamente se escribe sobre las particiones locales de a y

g, obteniendo otra vez un alto grado de localidad en los accesos. Esta situación no se da en la última etapa, la etapa de comunicación. En ella, se accede a elementos no consecutivos de los vectores de indirección y se leen particiones no locales del vector g. Esta última fase es la principal causa de ineficiencia de nuestra propuesta. Cabe destacar que los accesos no locales son siempre operaciones de lectura, mientras que todas las operaciones de escritura se aplican sobre la partición local de a y g asignada a cada procesador.

En términos de rendimiento, nuestra propuesta obtiene los mejores resultados para todas las variantes del problema *prueba*1, y para el escenario *prueba*2 cuando existe cierta localidad en los accesos. Otra de las principales causas de ineficiencia de nuestra propuesta lo representa el número de *slices* empleados. Observando la estructura del ejecutor mostrada en la Figura 6.12, cada vez que se ejecutan las iteraciones pertenecientes a un *slice*, es necesario ejecutar una operación de sincronización tipo barrera.

Sin embargo, nuestra propuesta tiene una importante característica, y es que el número de *slices* necesario para ejecutar un determinado problema no se ve influido por el número de procesadores. Este número estará principalmente determinado por el grado de localidad y las características del patrón de acceso. Por ejemplo, para el escenario *prueba*1, el número de *slices* producido para 8, 32 y 128 procesadores es 8, 10 y 12, respectivamente. Esta característica contribuye a que nuestra propuesta tenga una buena escalabilidad.

La Tabla 6.5 muestra el número total de iteraciones exclusivas y compartidas obtenidas por nuestra propuesta. Destacamos que, conforme se incrementa el número de procesadores, el número de iteraciones compartidas también lo hace. Sin embargo, en este caso, el número de iteraciones compartidas asignadas a cada procesador (ASPP en la Tabla 6.5) disminuye conforme aumenta el número de procesadores. Esto es debido a que a cada procesador tiene asignado una menor porción del espacio de iteraciones, reduciéndose el coste de la fase de comunicación y permitiendo obtener una buena escalabilidad con esta técnica.

Este no es el caso de la técnica *array expansion*, para la cual cada procesador siempre

Escenario	Iter.	8 procesadores				32 procesadores			
		excl.	comp.	ASPP	balanc.	excl.	comp.	ASPP	balac.
prueba1	100	835,142	120,658	15,082	1.007	497,618	458182	14,318	1.023
	2000	515,539	490,127	61,265	1.019	154,990	850,676	26,584	1.029
	6000	326,243	681,412	85,177	1.014	87,184	920,471	28,765	1.022
	14000	218,122	789,813	98,727	1.014	56,265	951,670	29,740	1.014

Tabla 6.5: Distribución de las iteraciones para la estrategia `SLCCLS`.

debe comunicar la misma cantidad de información, que se corresponde a la copia local de a que tiene asignada. Para el caso de las estrategias DWA-LIP y LOCAL-WRITE el rendimiento también decrece conforme el número de procesadores aumenta. Para la primera de ellas, el ejecutor muestra una pérdida de paralelismo. Por ejemplo, para el escenario *prueba*2, en la iteración 6000, el porcentaje de iteraciones que se ejecutan con menos de la mitad del número de procesadores disponibles aumenta del 62% con 8 procesadores, hasta el 86% con 32 procesadores. Para la estrategia LOCAL-WRITE, la replicación de las computaciones también es fuertemente dependiente del número de procesadores. Por ejemplo, para el mismo escenario, con 8 procesadores se calculan un 170% más de iteraciones que las calculadas usando *array expansion*, y este porcentaje aumenta al 190% con 32 procesadores. Por estos motivos entendemos que nuestra propuesta presenta una escalabilidad potencialmente superior al de las otras alternativas.

El balanceo de carga es un factor importante en el rendimiento del ejecutor paralelo. En nuestra propuesta, es el *scheduler* el encargado de distribuir las iteraciones sobre los procesadores. Hemos evaluado esta magnitud mediante la siguiente expresión.

$$balanceo\_carga = \frac{max(carga_{exclusiva}) + \sum_{s=1}^{N_S} max(carga_{compartida}[s])}{min(carga_{exclusiva}) + \sum_{s=1}^{N_S} min(carga_{compartida}[s])} \tag{6.14}$$

Donde $carga_{exclusiva}$ y $carga_{compartida}$ son vectores de N_p entradas que almacenan, respectivamente, el número de iteraciones exclusivas y compartidas que son ejecutadas por cada procesador en el *slice* s. Dado que después de procesar cada *slice* se realiza una operación de sincronización tipo barrera, es necesario evaluar el balanceo de carga de forma independiente para cada *slice*. De este modo, la Expresión 6.14 tiene en cuenta para cada uno de los *slices*, las contribuciones de aquellos procesadores con máxima y mínima carga computacional.

La Tabla 6.5 muestra el balanceo de carga obtenido con nuestra propuesta. Nótese unos valores muy próximos a la unidad, valor que supone el nivel de balanceo óptimo. Estos resultados prueban la eficiencia del *scheduler* para todos los escenarios considerados.

Evaluación del coste computacional asociado al cuerpo del lazo

Este parámetro (el coste computacional de cada iteración) depende de las implementación particular de cada aplicación y ejerce una gran influencia sobre el rendimiento. Con el fin de realizar una evaluación precisa del mismo, proponemos una modificación del código LJA incluyendo una carga artificial de trabajo dada a través de la inserción de un lazo con W iteraciones. De este modo, podemos modelar el coste computacional de cada iteración. La Figura 6.22 muestra el pseudocódigo de este nuevo código de prueba, mientras

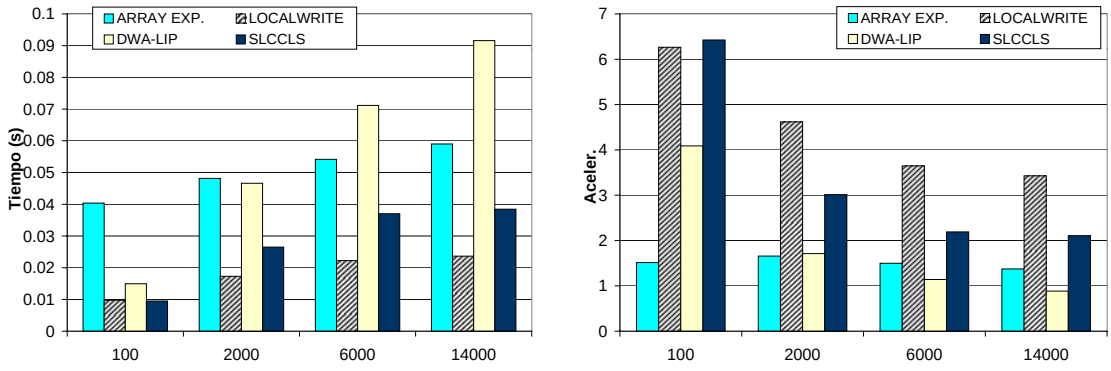
```
DO  t = 1, N_t
   DO  j = 1, N_x
       fuerza = calcula_fuerza(posición[x_1[j]], posición[x_2[j]])
       a[x_1[j]] += fuerza
       a[x_2[j]] -= fuerza
       DO w = 1, W
          carga de trabajo
       END DO
   END DO
   posición[1 : N_a] = actualiza_posición(a)
END DO
```

Figura 6.22: Lazo irregular de n-cuerpos con carga de trabajo variable.

que la Figura 6.23 muestra el tiempo de ejecución paralelo y las aceleraciones obtenidas con valores diferentes de W para cada una de estas propuestas. Debido a la política de distribución de las iteraciones de nuestra propuesta, el rendimiento de la técnica `SLCCLS` obtiene resultados competitivos para todo el rango de W considerado. Para valores de W elevados, el rendimiento de las técnicas `SLCCLS` y *array expansion* aumenta, y tiende a alcanzar niveles similares en ambas estrategias. En estas situaciones, el incremento en el tiempo de ejecución de cada iteración enmascara el coste de comunicaciones, beneficiando a ambas estrategias. Respecto a las técnicas DWA-LIP y LOCAL-WRITE, la serialización (en la primera) y replicación (en la segunda) de las computaciones tiene una influencia negativa sobre el rendimiento. A pesar de mejorar sus aceleraciones conforme W aumenta, en la figura podemos apreciar que este incremento es mucho menor que el alcanzado con las técnicas `SLCCLS` y *array expansion*. Siendo más exactos, para $W = 0$ la técnica LOCAL-WRITE supera a la `SLCCLS`, mientras que la DWA-LIP es equiparable, o incluso supera, a la *array expansion*. Para $W = 10$ esta relación se invierte, siendo ahora la técnica `SLCCLS` claramente superior al resto. Esta diferencia se ve más acentuada para $W = 20$ en donde para todos los escenarios, con excepción de aquellos correspondientes a 100 iteraciones, la técnica *array expansion* es la segunda más eficiente.

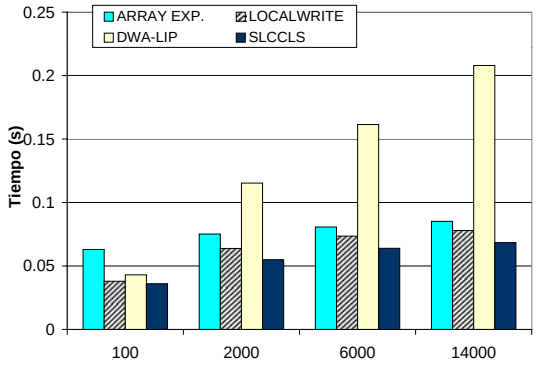
Evaluación del coste del inspector

Con el fin de evaluar el rendimiento global de nuestra propuesta debemos considerar el coste computacional del inspector. Típicamente, las aplicaciones de simulación de n-cuerpos establecen un intervalo de iteraciones en las que el patrón de acceso no sufre

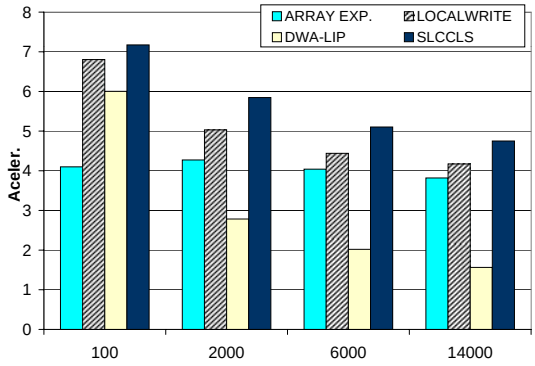


Tiempo ejecución ($W = 0$)

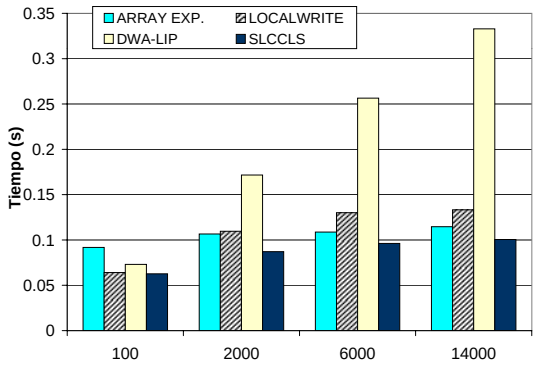
Aceleración ($W = 0$)



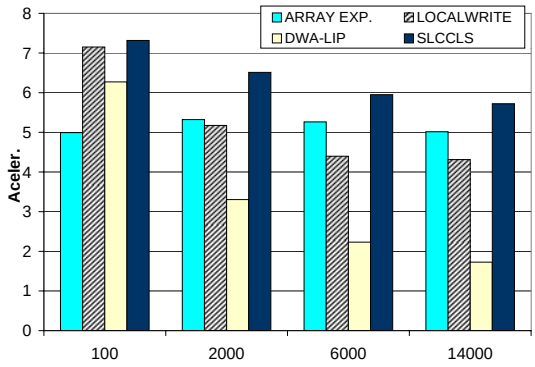
Tiempo ejecución ($W = 10$)



Aceleración ($W = 10$)



Tiempo ejecución ($W = 20$)



Aceleración ($W = 20$)

Figura 6.23: Tiempos de ejecución y aceleraciones para el escenario *prueba*3 con $N_p = 8$ y diferentes valores de W.

modificaciones. Dicho de otro modo, la rutina que evalúa los vecinos que interactúan con cada uno de los cuerpos se ejecuta cada cierto intervalo de iteraciones. De este modo, un umbral inferior a este valor hace rentable el empleo del inspector, respecto a la ejecución secuencial del programa.

La Tabla 6.6 muestra el umbral de iteraciones para distinto número de procesadores. Definimos dicho umbral como el número de iteraciones en las que la información del inspector debe ser reusada para compensar su coste. En particular, un valor de reuso mayor o igual que el umbral de iteraciones hace que nuestra propuesta (inspector más ejecutor) sea más eficiente que la ejecución secuencial del programa. La última fila de la Tabla 6.6 representa el umbral de iteraciones para 16 procesadores y $W = 10$. Se puede apreciar la fuerte disminución en este umbral debido al aumento en el coste del lazo. Hay que destacar el hecho que, debido a la alta escalabilidad de nuestra propuesta, el umbral de iteraciones se mantiene constante e incluso decrece conforme aumenta el número de procesadores.

La Tabla 6.7 muestra para $N_p = 32$, el umbral de iteraciones existente entre la estrategia `SLCCLS` y las técnicas *array expansion* y DWA-LIP. La técnica *array expansion* no utiliza un inspector, por lo que el umbral de iteraciones representa en número de veces que es necesario ejecutar el código paralelo para compensar el coste del inspector `SLCCLS`. En el caso de la técnica DWA-LIP se empleó un inspector secuencial (del mismo modo que con nuestra propuesta) por lo que el umbral de iteraciones debe tener en cuenta la diferencia de tiempo entre nuestro inspector y el inspector DWA-LIP. Un valor ∞ significa que nuestra propuesta nunca supera a la otra, debido a que el ejecutor paralelo es menos eficiente. Nuevamente, es necesario destacar que estos valores son puramente indicativos, dado que en una aplicación real se emplearían inspectores paralelos.

En el caso de la técnica LOCALWRITE hemos considerado el tiempo de ejecución del inspector paralelo. La Tabla 6.8 muestra para $N_p = 32$ el umbral de iteraciones de nuestra propuesta respecto a esta técnica. La columna encabezada con "*aceleración=1*"

Proc.	*prueba1*				*prueba2*				*prueba3*			
	100	$2K$	$6K$	$14K$	100	$2K$	$6K$	$14K$	100	$2K$	$6K$	$14K$
8	5	8	9	10	4	8	7	7	2	4	6	6
16	8	13	9	10	4	8	8	13	3	4	5	6
32	6	13	10	11	5	6	8	9	3	4	5	7
16($W = 10$)	2	3	2	2	0	2	2	3	0	0	0	0

Tabla 6.6: Umbral de iteraciones para 32 procesadores.

representa el umbral de iteraciones considerando el inspector `SLCCLS` secuencial. Debido a la gran diferencia de tiempos de ejecución entre ambos inspectores, este umbral presenta unos valores muy elevados. El resto de las columnas de la tabla asume unas aceleraciones de nuestro inspector con valores 5 y 25. Tal y como se podrá ver en la Sección 6.4, las aceleraciones alcanzadas por el inspector `SLCCLS` van a estar típicamente comprendidas entre estos valores. En la Tabla 6.8 se puede apreciar un fuerte descenso del umbral de iteraciones cuando se emplea un inspector `SLCCLS` paralelo. Un valor igual a cero indica

Matriz	Iteración	*array expansion*	DWA-LIP
*Prueba*1	100	17	29
	2000	28	8
	6000	29	4
	14000	26	2
*Prueba*2	100	132	22
	2000	239	20
	6000	∞	19
	14000	∞	14
*Prueba*3	100	2	6
	2000	3	2
	6000	4	2
	14000	3	2

Tabla 6.7: Umbral de iteración de la estrategia `SLCCLS` con $N_p = 32$.

Matriz	Iteración	*aceleración=1*	*aceleración=5*	*aceleración=25*
*Prueba*1	100	70	12	1
	2000	41	8	1
	6000	21	4	1
	14000	17	3	0
*Prueba*2	100	50	9	2
	2000	54	9	0
	6000	73	13	1
	14000	62	12	2
*Prueba*3	100	∞	∞	∞
	2000	∞	∞	∞
	6000	∞	∞	∞
	14000	∞	∞	∞

Tabla 6.8: Umbral de iteración de la estrategia `SLCCLS` con $N_p = 32$.

que nuestra propuesta es siempre la más eficiente.

Comparación con otros trabajos relacionados

En este apartado se compara la estructura de nuestra propuesta con las técnicas de *selective privatization* [152] y *sparse reductions with privatization in hash tables* [152]. Nuestra propuesta coincide con estas en emplear un *buffer* intermedio para almacenar los accesos compartidos, que en nuestro caso es el vector de guarda. Sin embargo, mediante el algoritmo `SLCCLS` se mantiene el tamaño del vector de guarda g dentro de un valor constante que no depende del número de procesadores. Adicionalmente, y gracias a la buena política de *scheduling*, el coste asociado a las operaciones de sincronización es mínimo, y tampoco aumenta con el número de procesadores. Dado que en nuestra propuesta el análisis de dependencias se realiza en la fase de inspección, los resultados del mismo pueden aplicarse, a diferencia de técnica *sparse reductions with privatization in hash tables*, desde la primera invocación del ejecutor.

Nuestra propuesta obtiene unos correcto balanceo de la carga. Por ejemplo, mediante pequeñas modificaciones en el algoritmo *scheduler*, la técnica `SLCCLS` puede contemplar un escenario de ejecución sobre procesadores con diferentes características o sobre redes de interconexión heterogéneas.

Una importante característica de nuestra propuesta es que reordena los vectores de indirección. Esto permite obtener una alta localidad, tanto en las lecturas de dichos vectores como en las escrituras sobre la matriz que almacena las fuerzas. El esquema de almacenamiento y acceso a las entradas del vector de indirección que hemos empleado permite reducir los costes de acceso a memoria y aumentar la escalabilidad de nuestra propuesta. Finalmente, la técnica `SLCCLS` está organizada en varios módulos, con lo que resulta sencillo ajustar el funcionamiento de cada uno de ellos de forma independiente.

6.4 Paralelización del inspector `SLCCLS`

La etapa de inspección del algoritmo `SLCCLS` puede ser fácilmente paralelizada distribuyendo las iteraciones sobre los procesadores y aplicando sobre cada uno de estos conjuntos el inspector `SLCCLS` secuencial. Con el fin de conseguir un reducido número de *slices*, es necesario asignar a cada procesador las iteraciones que originen el menor número de conflictos de acceso a g. En el caso de emplear una distribución de las iteraciones por bloques, y debido a la alta localidad en los accesos, existe una alta probabilidad de coincidencia en

```
DO p = 1, Nth
   DO i1 = p, Ni1, Np
      DO i2 = 1, Ni2
         i = i2 * Ni1 + i1
```

Figura 6.24: Distribución de las iteraciones para el inspector SLCCLS paralelo.

Figura 6.25: Secuencia de entradas accedidas por el procesador 1, para $N_x = 12$, $N_{th} = 2$ y $N_{i1} = 4$.

el valor del acceso local de las iteraciones asignadas a un mismo procesador. Esto tiene dos implicaciones: por una parte se produce un aumento del número de *slices*, y por otra se produce una disminución del rendimiento del *scheduler* en términos de balanceo de carga.

El mismo efecto sucede en una distribución cíclica de las entradas para la que cada procesador tiene asignadas las iteraciones que están separadas por $N_p - 1$ entradas. Hemos podido evaluar que esta separación no es suficiente para asegurar un correcto funcionamiento de nuestra propuesta.

Experimentalmente, hemos comprobado que la distribución que ofrece los mejores resultados es una distribución doblemente cíclica de las iteraciones del lazo irregular. Esta distribución se consigue reemplazando los lazos etiquetados como L1 y L2 de la Figura 6.15, por los que se ilustran en la Figura 6.24. Donde N_{th} es el número de procesadores que intervienen en la ejecución paralela del inspector. N_{i1} es un parámetro externo definido por el usuario que debe verificar que, $N_{i1} = k * N_{th}$ con $k \in \mathbb{N}$. El parámetro N_{i2} se obtiene a partir del anterior mediante la siguiente expresión:

$$N_{i2} = \lceil \frac{N_x}{N_{i1}} \rceil \tag{6.15}$$

Hemos implementado nuestra propuesta distinguiendo entre el número de procesadores que intervienen en la ejecución paralela del inspector (N_{th}) y los que intervienen en la ejecución del lazo irregular paralelo (N_p). La Figura 6.25 muestra un ejemplo del esquema de acceso asociado al procesador 1 para un escenario con $N_x = 12$, $N_{th} = 2$ y $N_{i1} = 4$ empleando la distribución doblemente cíclica. En dicha figura se muestran en un tono claro aquellas entradas recorridas por el procesador 1. El valor numérico contenido dentro de cada una denota el orden con el que son accedidas.

De este modo, en la ejecución paralela del inspector, cada uno de los procesadores opera independientemente sobre un subconjunto de iteraciones. Una segunda modificación que es necesario realizar consiste en expandir los vectores ρ_1^{slice} y ρ_2^{slice} en una dimensión adicional que identifica a cada uno de los procesadores que intervienen en la ejecución del inspector.

Con el fin de asegurar el correcto reordenamiento de los vectores de indirección, es necesario hacer una modificación en la fase de desplazamiento (ver Sección 6.3.3). Concretamente, al final de esta fase, es necesario introducir el siguiente paso.

- **Paso 5:** Cada procesador que interviene en la ejecución paralela del inspector, evalúa el número de iteraciones asignadas a todos los procesadores que tienen un identificador menor que el suyo. Este valor es sumado a todas las entradas de ρ_1^{slice} y ρ_2^{slice} que dicho procesador tiene asociadas.

La implementación de este nuevo paso es muy sencilla, dado que se conoce con antelación la distribución empleada y, consiguientemente, el número de iteraciones asignadas a cada uno de los procesadores. De este modo, asumiendo que cada procesador p, recibe $\Delta N_x[p]$ entradas, el valor $K[q]$ con el que el procesador q debe actualizar sus entradas asociadas a ρ_1^{slice} y ρ_2^{slice} viene dado por:

$$K[q] = \sum_{p=1}^{q-1} \Delta N_x[p] \tag{6.16}$$

Respecto al ejecutor, es necesario realizar pequeñas modificaciones en el mismo para poder emplear la información generada por el inspector paralelo. Ahora las entradas de los vectores de indirección se encuentran fragmentadas en N_{th} conjuntos independientes (obtenidos por cada procesador) con su correspondientes regiones exclusivas y compartidas. Durante la ejecución paralela, es necesario recorrer cada uno de estos conjuntos de forma consecutiva.

La Figura 6.26 muestra el nuevo pseudocódigo del ejecutor paralelo. Hemos añadido los lazos etiquetados como `L1` y `L2` para recopilar la distribución obtenida por cada procesador.

Las iteraciones exclusivas obtenidas en paralelo y asociadas a la misma partición pueden ser ejecutadas sin sincronizaciones. Respecto a las iteraciones compartidas, cada procesador únicamente debe procesar aquellos *slices* que le han sido asignados por cada uno de los procesos empleados en el inspector paralelo.

```
SHARED a, g, ρ1^S, ρ2^S, x1, x2, ρ^excl

    DOALL p = 1, Np                                          %Etapa exclusiva
L1      DO q = 1, Nth
            DO j = ρ1^s[q,p,1,1] − ρ^excl[q,p], ρ2^s[q,p,1,1] − 1
                fuerza = calcula_fuerza(posición(x1[j]), posición(x2[j]))
                a[x1[j]] += fuerza
                a[x2[j]] −= fuerza
            END DO
        END DO
    END DOALL

L2  DO q = 1, Nth
        DO s = 1, NS
            DOALL p = 1, Np
                DO j = ρ1^s[q,p,s,1], ρ2^s[q,p,s,1] − 1          %Etapa compartida-1
                    fuerza = calcula_fuerza(posición(x1[j]), posición(x2[j]))
                    a[x1[j]] += fuerza
                    g[x1[j]] = fuerza
                END DO
                DO j = ρ2^s[q,p,s,1], ρ1^s[q,p,s+1,1] − 1        %Etapa compartida-2
                    fuerza = calcula_fuerza(posición(x1[j]), posición(x2[j]))
                    g[x2[j] = fuerza
                    a[x2[j]] −= fuerza
                  END DO
            END DOALL
            BARRIER                                              %Sincronización
            DOALL p = 1, Np                                      %Etapa de recopilación
                DO k = 1, Np
                    DO j = ρ1^s[q,k,s,p], ρ1^s[q,k,s,p+1] − 1
                        a[x2[j]] −= g[x1[j]]
                    END DO
                    DO j = ρ2^s[q,k,s,p], ρ2^s[q,k,s,p+1] − 1
                        a[x1[j]] += g[x2[j]]
                    END DO
                END DO
            END DOALL
            BARRIER                                              %Sincronización
        END DO
    END DO
```

Figura 6.26: Algoritmo ejecutor modificado para un inspector paralelo.

6.4.1 Análisis de eficiencia

Para evaluar la calidad del proceso de distribución de las iteraciones, hemos aplicado al inspector paralelo los tres tipos de distribuciones contempladas (por bloques, cíclica y doblemente cíclica). La Tabla 6.9 muestra el máximo número de *slices* y el balanceo de la carga para distintos valores de N_p y N_{th} en el problema $prueba1$ con $iter = 2000$. El número máximo de *slices* representa al valor máximo de esta magnitud de entre los N_{th} procesadores que intervienen en la ejecución del inspector. Cabe destacar que el número de *slices* obtenido por cada uno de los procesadores tiene un valor similar al valor máximo, por lo que esta magnitud nos da una estimación de la eficiencia del inspector. Por otra parte, el balanceo de la carga se ha obtenido empleando la Ecuación 6.14 y teniendo en cuenta las contribuciones de cada procesador. Analizando los valores de la Tabla 6.9, podemos concluir que la distribución doblemente cíclica resulta la más eficiente tanto en términos de reducción del número de *slices* como en obtención de un correcto balanceo de la carga.

La Tabla 6.10 muestra el número total de *slices* para una ejecución en 8 procesadores para valores de $N_{th} = 1$ y $N_{th} = 8$. Podemos apreciar que el número de *slices* crece con una menor progresión que el número de procesadores. Este aumento del número de *slices* causa un mayor número de operaciones de sincronización, que afecta a las aceleraciones del ejecutor, tal y como se refleja en la tabla. A pesar de este efecto se puede apreciar que el rendimiento del ejecutor paralelo sigue alcanzando niveles aceptables.

La Tabla 6.11 muestra los tiempos de ejecución y aceleraciones obtenidas con nuestro ejecutor paralelo para $N_p = 8$ con $N_{th} = 1$ y $N_{th} = 8$. Gracias al alto nivel de desacoplo existente en la ejecución paralela del inspector, podemos obtener aceleraciones competitivas. La principal causa de ineficiencia de esta etapa son los accesos que cada procesador debe realizar sobre entradas no consecutivas de los vectores de indirección. Es-

Distribución	$N_p = N_{th} = 4$		$N_p = N_{th} = 16$		$N_p = N_{th} = 32$	
	N_S^{max}	*balanc.*	N_S^{max}	*balanc.*	N_S^{max}	*balanc.*
Bloques	15	1.025	11	1.274	13	1.249
Cíclica	5	1.074	4	1.091	4	1.068
Doblemente cíclica	5	1.008	3	1.030	3	1.035

Tabla 6.9: Eficiencia del inspector paralelo para distintas estrategias de distribución de las iteraciones para $prueba1$ con $iter = 2000$.

te hecho causa una baja explotación de la jerarquía de memoria, la cual impide alcanzar las aceleraciones ideales.

		*prueba*1		*prueba*2		*prueba*3	
		N_S	aceleración	N_S	aceleración	N_S	aceleración
100	$N_{th}=1$	9	5.7	18	5.0	5	6.4
	$N_{th}=8$	24	5.3	40	4.8	16	5.0
2000	$N_{th}=1$	14	3.5	24	4.5	7	3.0
	$N_{th}=8$	33	3.2	47	3.9	26	2.3
6000	$N_{th}=1$	18	2.7	28	3.8	9	2.2
	$N_{th}=8$	40	2.2	48	3.8	32	1.8
14000	$N_{th}=1$	21	2.2	31	3.7	10	2.1
	$N_{th}=8$	42	1.9	56	3.0	32	1.5

Tabla 6.10: Eficiencia del ejecutor para $N_p = 8$ con distintos valores de N_{th}.

		*prueba*1		*prueba*2		*prueba*3	
		T. (seg)	aceleración.	T. (seg)	acel.	T. (seg)	acel.
100	$N_{th}=1$	0.86		0.80		0.08	
	$N_{th}=8$	0.22	3.9	0.30	2.7	0.02	4.0
2000	$N_{th}=1$	1.42		1.58		0.20	
	$N_{th}=8$	0.27	5.3	0.31	5.1	0.04	5.3
6000	$N_{th}=1$	2.45		1.45		0.22	
	$N_{th}=8$	0.31	7.9	7.8	5.4	0.04	5.5
14000	$N_{th}=1$	1.26		1.54		0.24	
	$N_{th}=8$	0.34	3.7	0.30	5.1	0.04	6

Tabla 6.11: Tiempos de ejecución y aceleraciones para el inspector paralelo para $N_p = 8$ con distintos valores de N_{th}.

Capítulo 7

Árbol de decisión

El trabajo enmarcado en esta tesis se inició con el desarrollo de una caracterización del patrón de acceso asociado a un vector de indirección. Basándonos en esta estructura, e inspirándonos en su proceso de construcción, hemos desarrollado diversas técnicas de análisis de dependencias y de paralelización. El objetivo de este capítulo es establecer un procedimiento que sea capaz de determinar, para cada problema concreto, la técnica de optimización más eficiente. Existen distintas alternativas para implementar este procedimiento [151], en nuestro caso hemos optado por una estructura basada en un árbol de decisión. Debido a la naturaleza de los códigos irregulares, como criterios de clasificación debemos considerar no sólo la estructura del programa, sino también las características de su patrón de acceso. Por este motivo, es necesario aplicar este procedimiento durante la ejecución del programa. Como veremos con posterioridad, la representación IARD va a jugar nuevamente un papel importante para el correcto funcionamiento del proceso de clasificación.

Así pues, comenzaremos con una descripción de otras técnicas existentes que han abordado el desarrollo de árboles de decisión para la paralelización de códigos irregulares. Basándonos en estos trabajos desarrollaremos nuestra propuesta de árbol. Debido a que hemos considerado distintas clases de códigos irregulares, el proceso de decisión está dividido en varios subprocesos de forma que cada uno de ellos atiende a características particulares de cada clase de código concreta.

7.1 Trabajo previo

Debido a que las técnicas de paralelización existentes en este campo son relativamente recientes, existen pocos trabajos en los que se aborde su clasificación. Nuestra revisión bibliográfica comienza con la propuesta realizada por Yu y Rauchweger [152], la cual está destinada a la determinación, en tiempo de ejecución, de la técnica de paralelización más eficiente en función de las características del problema. Esta propuesta particulariza su estudio a códigos irregulares compuestos exclusivamente por operaciones de reducción, en las que el orden de las iteraciones (y los accesos a memoria) puede ser alterado. Esta propuesta se basa en la extracción de una serie de parámetros que permiten realizar una estimación del método más adecuado para cada situación. Los autores proponen un árbol de decisión basado en la comparación de los valores de estos parámetros con ciertos umbrales preestablecidos. Posteriormente, en [151] se amplia esta caracterización y se evalúa el impacto de nuevos factores en la eficiencia de distintas propuestas. En este último trabajo los autores elaboran dos modelos diferentes de decisión: un modelo estadístico con un bajo grado de regresión, y un árbol de decisión generado de forma automática [115] con una estructura mucho más compleja que el del trabajo previo.

Es importante destacar que en [152] y [151] la información acerca de la topología del patrón de acceso es muy reducida. En este contexto entendemos que puede resultar de gran utilidad el empleo de la caracterización IARD, dado que permitiría obtener una información mucho más precisa sobre el patrón de acceso.

7.2 Estructura del árbol de decisión

Siguiendo el planteamiento desarrollado por Rauchwerger et al., en esta sección proponemos un árbol de decisión que selecciona la mejor estrategia de optimización de código para cada caso concreto. Nuestra estrategia no se restringe sólo a reducciones irregulares, sino que considera todas las situaciones evaluadas a lo largo de esta tesis (lazos con una indirección, lazos con varias indirecciones y con operaciones de reducción). Para la generación del árbol de decisión hacemos uso de los siguientes parámetros:

- **Reusabilidad (R)**, indica, para un valor superior a la unidad, el número de veces que se ejecuta el lazo sin que el patrón de acceso sufra modificaciones. Un valor de este parámetro inferior a la unidad implica cambios en el patrón cada vez que el lazo es ejecutado. Mientras sea más próximo a cero, los cambios en la estructura del patrón de acceso serán más importantes.

- **Dimensionalidad (DIM)**, es el cociente entre las dimensiones de la matriz de reducción y el número de entradas que se pueden almacenar la memoria cache. Este parámetro da idea acerca de la relación entre el tamaño del problema ejecutado y los recursos disponibles por el sistema. Vamos a asumir que la memoria cache del sistema tiene N_{LC} líneas, y en cada línea tenemos N_L entradas de a. De este modo, la dimensionalidad se puede expresar por

$$DIM = \frac{N_a}{N_L N_{LC}} \tag{7.1}$$

- **Anchura media del patrón de acceso (λ)**. Representa una estimación del grado de localidad del patrón de acceso a memoria asociado al código irregular. Este parámetro se puede obtener mediante un análisis de la representación IARD.

- **Coste del lazo (W)**, representa la carga computacional del lazo que no está asociada a los estamentos con accesos irregulares.

- **Conectividad (CON)**, expresada como el cociente del número de iteraciones del lazo dividido por el número de accesos sobre posiciones de memoria distintas. Este parámetro da una estimación acerca de la relación entre computaciones y comunicaciones existentes en una ejecución paralela del lazo. A modo de ejemplo, una alta conectividad implica un gran número de iteraciones (con su carga computacional asociada) que acceden sobre un reducido espacio de memoria (el cual tiene asociado, en un entorno de ejecución paralelo, un coste de comunicación). La conectividad de un patrón de acceso se obtiene por la siguiente relación:

$$CON = \frac{N_x}{N_a} \tag{7.2}$$

A continuación vamos a describir la organización del esquema de paralelización automática que hemos desarrollado. Nuestra intención es ofrecer un criterio que permita elegir, de entre el conjunto de técnicas existentes, la que optimice la ejecución de un determinado código irregular. Los mecanismos de análisis que proponemos se inician en tiempo de compilación, con el estudio de la estructura del código y la identificación de aquellos lazos candidatos a ser optimizados. Los procesos concretos para realizar estas dos tareas no son considerados en el marco de nuestro trabajo, y nos remitimos a otros estudios [126, 79, 80].

Así pues, vamos a asumir que conocemos la estructura del código irregular y que podemos extraer la información relativa al patrón de acceso que tiene asociado. La Figura 7.1 muestra la estructura del árbol de decisión que realiza esta tarea. En total son evaluadas las tres condiciones que se describen a continuación:

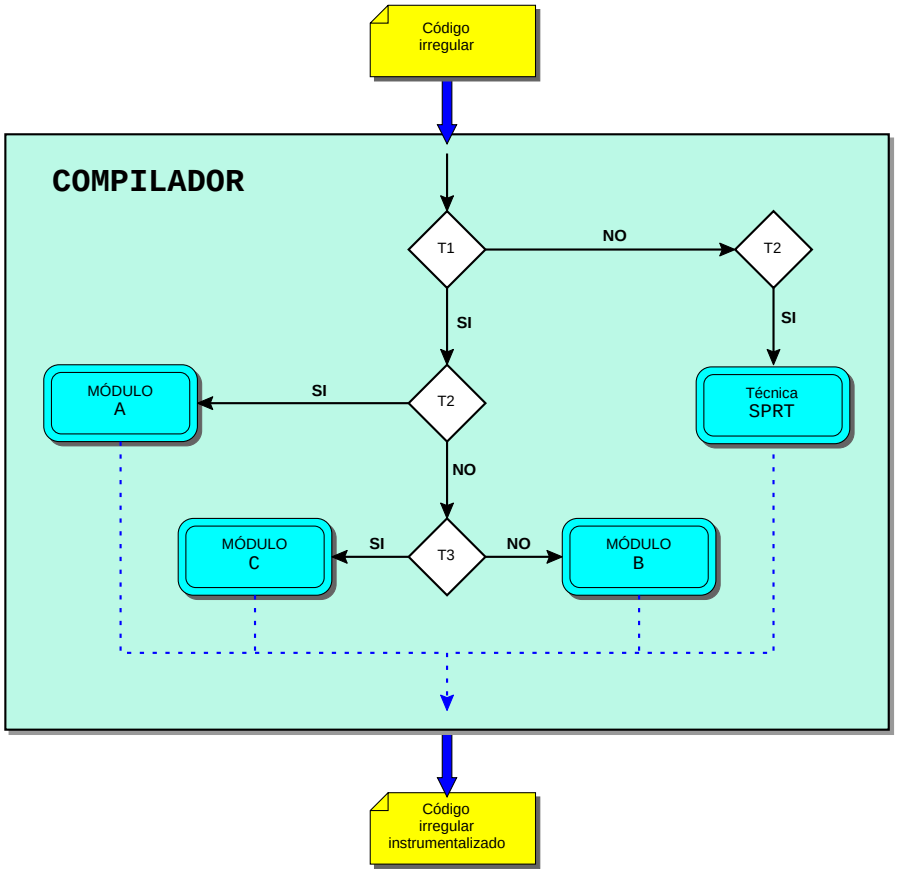


Figura 7.1: Árbol de decisión del esquema general.

- **T1:** ¿Disponemos de más de un procesador para ejecutar el programa en paralelo?
- **T2:** ¿Existen bloques de código irregular con una única indirección?
- **T3:** ¿Es posible cambiar el orden de los accesos a memoria?

La primera condición únicamente considera la posibilidad de paralelizar el lazo irregular. En caso de que no existan recursos para realizar esta tarea, la única optimización que podemos aplicar es la mejora en la localidad de los accesos mediante la técnica `SPRT`. Esta técnica sólo puede ser empleada sobre lazos con una indirección (condición T2).

En el caso de poder paralelizar el programa, hemos agrupado las distintas técnicas que pueden ser aplicadas en tres grandes conjuntos. El Módulo A contiene aquellas propuestas destinadas a la paralelización de lazos con una única indirección (condición T2). En el caso de existir más de una indirección, es necesario distinguir aquellas estructuras que pueden

ejecutarse en paralelo cambiando el orden de los accesos a memoria. En la Sección 1.4 se realizó una descripción de los requisitos que debe cumplir la estructura del lazo para tener esta característica. En caso de no cumplir este tipo de requisitos (condición T3), podemos aplicar las técnicas pertenecientes al Módulo B. En caso contrario, sólo es posible utilizar el Módulo C.

Todo este análisis se puede realizar en tiempo de compilación, una vez conocida la estructura del código irregular. Tras determinar el conjunto de técnicas que podemos aplicar, es necesario realizar la instrumentalización del programa. Ese proceso consta de los siguientes pasos:

1. Determinación de los puntos del programa en los que el vector de indirección se inicializa o sufre modificaciones. En estas regiones se insertan llamadas a rutinas de caracterización que permiten obtener la representación IARD.

2. Extracción de la información del contexto del código irregular. Esta información incluye el esquema de acceso al vector de indirección, el número de iteraciones, los parámetros asociados a la carga del lazo, el tipo de acceso realizado, etc. En el Capítulo 2 introducimos técnicas de transformación para obtener, en base a la información del contexto y de la caracterización IARD, el patrón de acceso asociado al código irregular. De este modo, cuando el código instrumentalizado sea ejecutado es posible obtener la información relativa a su patrón de acceso a memoria.

3. Inserción de las rutinas del inspector y ejecutor asociadas a cada una de las técnicas de paralelización consideradas. En el marco de nuestro trabajo no consideramos de forma rigurosa el proceso de ubicación de las rutinas de inspección. Nuevamente, nos remitimos a diversos trabajos [140, 141, 69] en los que se aborda esta problemática.

La elección de la técnica concreta de paralelización que va a ser empleada se realiza en tiempo de ejecución, mediante un árbol de decisión específico para cada módulo. En las siguientes secciones describimos la estructura de cada uno de ellos. Para cada uno de estos casos, únicamente hemos contemplado aquellas técnicas de paralelización que evaluamos de forma experimental. Hemos optado por la no inclusión de otras técnicas de paralelización que no fueron evaluadas, debido a que únicamente pueden hacerse estudios comparativos en base al comportamiento teórico de estas alternativas.

El esquema de decisión que hemos utilizado resulta más cualitativo que el empleado por Yu y Rauchweger. Nuestro objetivo no es la determinación "exacta" de la técnica más eficiente, sino la evaluación de la influencia de ciertos parámetros sobre la eficiencia de cada propuesta. A lo largo de esta sección emplearemos los símbolos "K:↓" y "K:↑"

para indicar, de manera aproximada, que cualquier parámetro K tiene un valor elevado o reducido. Mediante los símbolos "K:↑↑" y "K:↓↓" representamos valores especialmente grandes o pequeños. No hemos establecido un umbral concreto de cada parámetro ya que este sería dependiente de las características del sistema de computación particular.

7.2.1 Códigos irregulares con una única indirección

Este conjunto de propuestas se corresponden al Módulo A del esquema general de nuestra propuesta. La Figura 7.2 ilustra el árbol de decisión correspondiente a este módulo.

En total se evalúan seis condiciones que se muestran a continuación:

1. **TA1:** ¿Se desea ejecutar en paralelo distintas partes del código?
2. **TA2:** R:↓ | (DIM:↓ & λ:↑)
3. **TA3:** ¿Está mal balanceado el lazo irregular?
4. **TA4:** ¿Son los accesos dentro de cada *slice* monótonos? & λ:↓
5. **TA5:** N_{pt}:↓ & λ:↓
6. **TA6:** W :↑↑ | λ:↑↑

La primera de ellas determina el grano de paralelismo que deseamos aplicar. Este parámetro depende de varios factores, como la estructura del código, el número de iteraciones de los lazos y la arquitectura del sistema. En el caso de emplear un tamaño de grano de un lazo irregular (condición TA1 cierta), nuestra propuesta da soporte al análisis de dependencias entre distintas estructuras de código. Para realizar este análisis empleamos la caracterización del patrón de acceso en base a la representación IARD y el algoritmo DS. Mediante el empleo de este algoritmo, y el procedimiento descrito en la Sección 3.3, es posible evaluar las posiciones de memoria comunes a ambas regiones, y consiguientemente, la existencia de posibles dependencias de datos.

Si se desea extraer paralelismo a nivel de iteración, nuestra propuesta considera un total de cinco técnicas que están agrupadas en dos familias: las basadas en la privatización del vector de reducción (*array expansion*) y las basadas en la aplicación de la regla del propietario (DWA-LIP, PRT, SPRT-IP, SPRT-PAR). La elección entre una u otra familia depende fundamentalmente del nivel de reuso asociado al ejecutor paralelo. Dado que la técnica de *array expansion* no emplea ninguna rutina de inspección, resulta adecuada cuando el grado de reuso del lazo es nulo o muy reducido (condición TA2). La otra

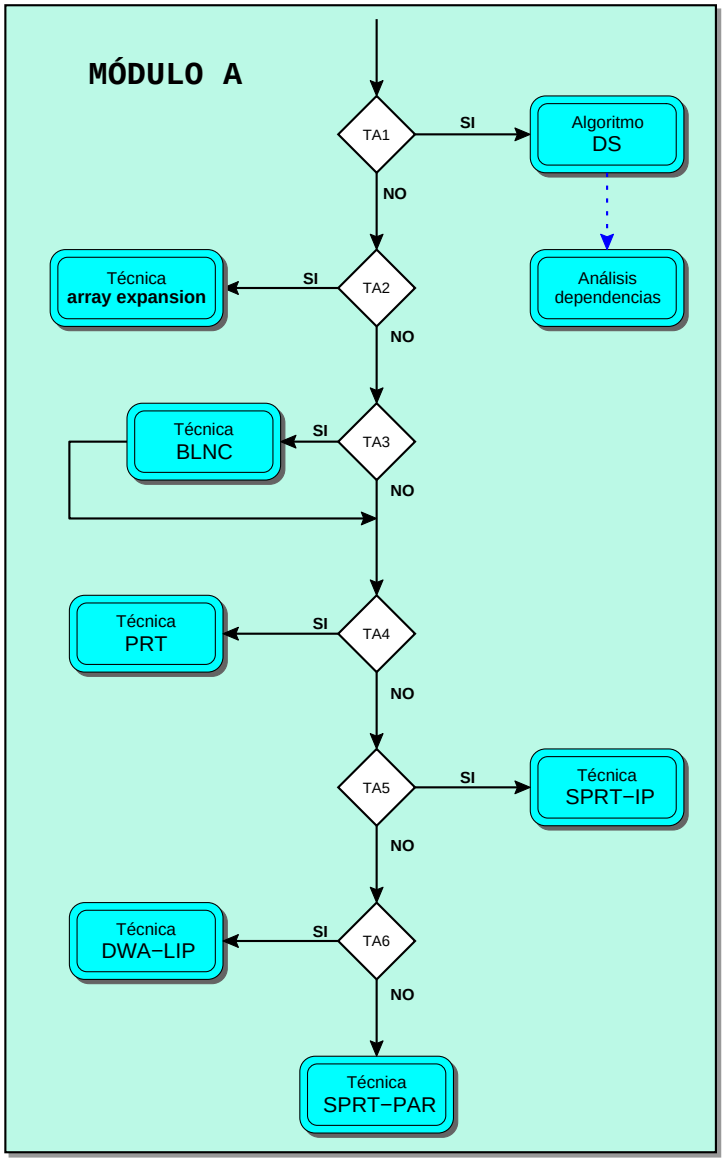


Figura 7.2: Árbol de decisión del Módulo A.

condición bajo la que el rendimiento de esta propuesta es competitivo se produce cuando la dimensionalidad del problema es reducida y el grado de dispersión (que nosotros evaluamos por la anchura media del patrón de acceso λ) es alto. Al contrario de las otras propuestas, el rendimiento de la técnica *array expansion* no se ve influenciado de forma negativa por el parámetro λ.

En el caso de la segunda familia de técnicas, es necesario considerar el balanceo de la carga (condición TA3). Mediante el algoritmo `BLNC` podemos obtener un buen balanceo de la carga para todas las técnicas de esta familia. La condición TA4 evalúa la posibilidad de aplicación de la técnica `PRT`. En el caso que los accesos dentro de cada *slice* sean monónotos[1] es posible aplicar la técnica `PRT` sin realizar un reordenamiento del vector de indirección. Esto nos permite reducir los costes de almacenamiento de esta propuesta y disminuir el coste de la rutina de inspección. Adicionalmente, la segunda condición tiene en cuenta que esta rutina sólo funciona eficientemente cuando la anchura de la banda es reducida.

Si el número de particiones y la anchura de la banda son reducidos, la técnica `PRT-IP` obtiene las mejores prestaciones (condición TA5). En esta elección es necesario considerar que esta técnica realiza el ordenamiento *in-place* del vector de indirección, por lo que, en caso de que resulte necesario preservar el vector original, debe hacerse una copia del mismo.

Finalmente, en el caso de descartar todas estas opciones, las técnicas más eficientes son la DWA-LIP y la `SPRT-PAR`. El coste asociado a la técnica DWA-LIP se ve enmascarado cuando la carga de trabajo asociada al cuerpo del lazo es elevada. Por otra parte, el coste de almacenamiento de esta técnica no guarda dependencia con la anchura media del patrón de acceso. En cualquiera de estas dos situaciones la técnica DWA-LIP es la más competitiva. Respecto a la técnica `SPRT-PAR`, el aumento de la carga de trabajo también beneficia a nuestra propuesta, alcanzado un rendimiento similar a la DWA-LIP. Sin embargo, el coste asociado a esta técnica depende de las características topológicas del patrón de acceso. Estos factores son evaluados por la condición TA6.

7.2.2 Códigos irregulares con varias indirecciones

En este módulo (Módulo B) consideramos los códigos irregulares con un número arbitrario de indirecciones y en los que los acceso a las entradas de a deben realizarse en el mismo orden que en el programa original. La estructura del árbol de decisión para este escenario se ilustra en la Figura 7.3. Se evalúan las siguientes condiciones:

1. **TB1:** ¿Se desea ejecutar en paralelo distintas partes de código?
2. **TB2:** ¿El lazo considerado permite la ejecución fuera de orden de los estamentos?
3. **TB3:** R:↑ | ¿Existe desbalanceo de la carga? | W :↓

[1] Es decir, para un *slice* S_i genérico, si $j \in S_i$ y $k \in S_i$ con $j < k$, entonces $x[j] > x[k]$ o bien $x[j] < x[k]$.

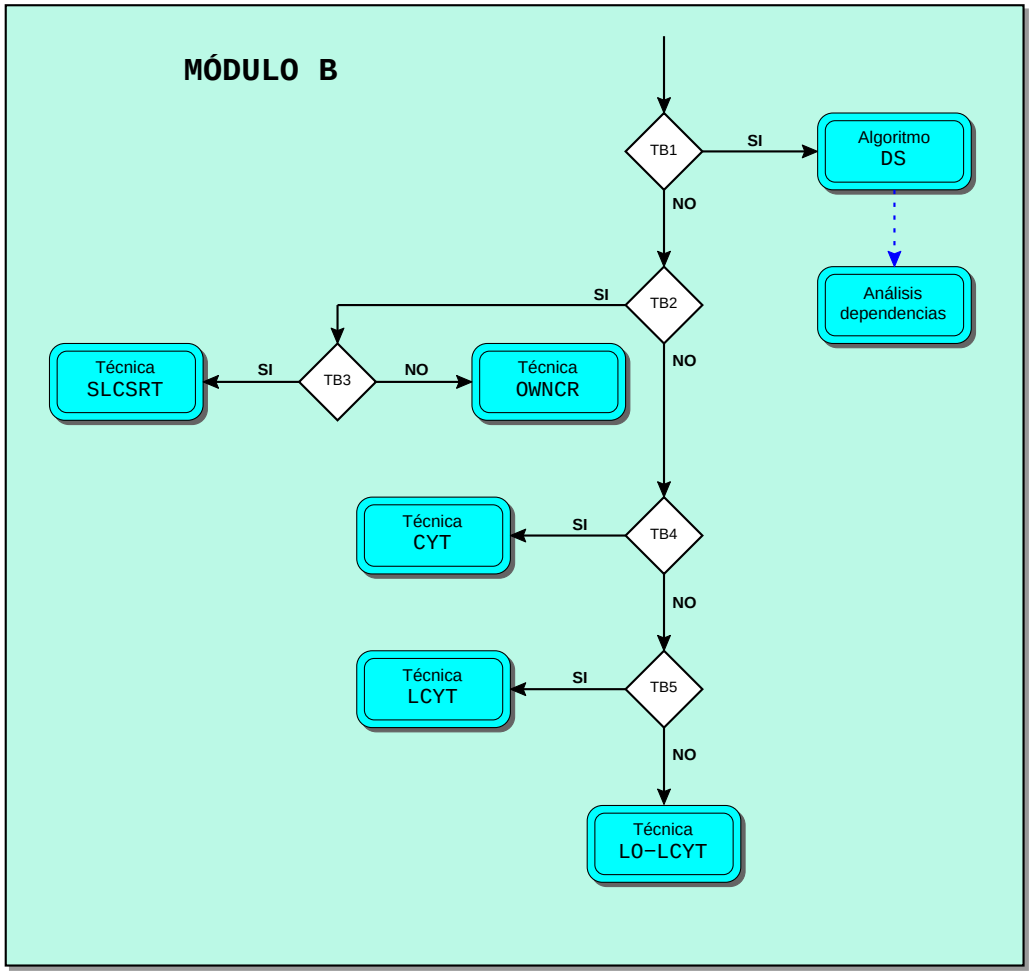


Figura 7.3: Árbol de decisión del Módulo B.

4. **TB4:** W :↑↑

5. **TB5:** R:↑ & $N_x \gg N_a$

La primera de ellas es idéntica a la descrita en el módulo anterior. Únicamente es necesario destacar que, debido a la estructura del lazo irregular que estamos considerando, hay que aplicar el algoritmo de clasificación por *slices* `CS3` (introducido en la Sección 5.5) con el fin de caracterizar el patrón de acceso a memoria. Posteriormente, se pueden generar las envolventes superiores e inferiores del mismo y obtener su caracterización IARD asociada. Nótese que en este caso, la caracterización IARD no hace referencia a un vector de indirección, sino al patrón de acceso asociado a la sección del código.

Si se desea realizar una paralelización a nivel de lazo las cinco técnicas posibles, se pueden clasificar en dos familias: las que realizan una ejecución de los estamentos del lazo fuera de orden (SLCSRT y OWNCR) y las que no lo hacen (CYT, LCYT y LO-LCYT).

En el Capítulo 5 se demostró que el rendimiento de las técnicas pertenecientes a la primera familia es superior al de la segunda. Por este motivo, la condición TB2 evalúa si el lazo considerado permite una ejecución fuera de orden de los estamentos. En caso de ser cierta, la condición TB3 determina cual de las dos técnicas resulta la más eficiente. Basándonos en los resultados obtenidos, la técnica SLCSRT supera a la OWNCR si existe reuso del ejecutor. En el caso de que exista un alto balanceo de la carga o que el coste por iteración sea elevado, el rendimiento de la técnica OWNCR puede ser reducido. En este caso la técnica SLCSRT es la más eficiente, aún existiendo bajo nivel de reuso.

En el caso de que ninguna de estas técnicas puedan ser aplicadas a la paralelización del lazo considerado, es necesario evaluar la condición TB4. La técnica CYT es la más eficiente si la carga de trabajo por iteración resulta extremadamente grande. Cabe destacar que en esta situación el rendimiento de las demás propuestas no resulta muy inferior al de la CYT.

En caso que la carga de trabajo no sea elevada, tenemos dos últimas opciones: las técnicas LCYT y LO-LCYT. La elección de una u otra la realiza TB5, y los criterios que emplea son el grado de reuso del ejecutor, y la relación entre el número de iteraciones y el número de entradas de a. Cuando ambos valores no son reducidos, la técnica LCYT supera en rendimiento a la técnica LO-LCYT. Esto se debe a que la técnica LCYT emplea un inspector más costoso que el empleado por la LO-LCYT, especialmente, cuando $N_x \simeq N_a$.

7.2.3 Códigos irregulares con operaciones de reducción

Esta clase de código se corresponde al último de los módulos descritos, el Módulo C. La Figura 7.4 muestra el árbol de decisión recogido en nuestra propuesta. En este árbol se evalúan las siguientes condiciones:

1. **TC1:** ¿Se desea ejecutar en paralelo distintas partes de código?
2. **TC2:** (W :↑ & R:↓) | λ:↑↑
3. **TC3:** W :↑ | CON:↑

Una vez más, la primera condición determina el tamaño de grano utilizado. Del mismo modo que en el módulo anterior, es necesario utilizar el algoritmo CS3 para clasificar los

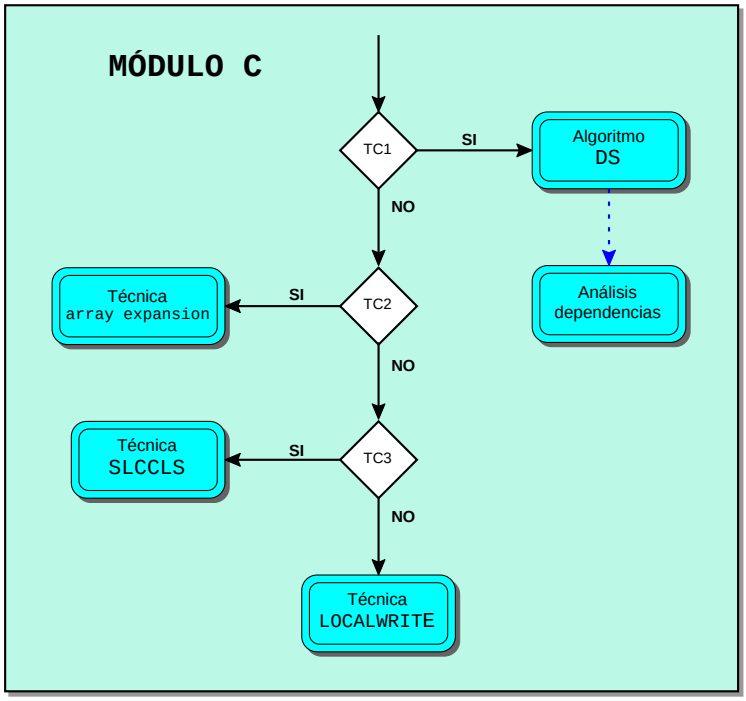


Figura 7.4: Árbol de decisión del Módulo C.

accesos a memoria del lazo irregular. En base a esta clasificación se genera la representación IARD del patrón de acceso de cada sección particular del programa. Mediante el algoritmo DS y las técnicas de análisis de dependencias se determina la existencia de riesgo de solape entre las zonas de memoria accedidas por cada región.

En caso de elegir la paralelización individual de cada lazo, proponemos el empleo de un conjunto de tres técnicas de paralelización: *array expansion*, LOCALWRITE y SLCCLS. El motivo de esta elección es que cada una representa el caso más competitivo de las tres estrategias distintas de paralelización evaluadas: la privatización del vector de reducción, la aplicación de la regla del propietario mediante una replicación parcial de las iteraciones, y la aplicación de la regla del propietario mediante un esquema de sincronización entre procesadores.

El criterio de elección de la técnica *array expansion* se realiza mediante TC2. Esta técnica alcanza un alto rendimiento si la carga de trabajo asociada al lazo es elevada. Bajo estas condiciones, *array expansion* supera en rendimiento a SLCCLS si el nivel de reuso es muy reducido. Empleamos el término "global" para indicar que es necesario considerar (igual que en los casos anteriores) el coste de la rutina de inspección. Finalmente, si la

anchura media de banda del patrón de acceso es muy elevada, el rendimiento de esta propuesta también puede superar al de las demás.

En caso de que la condición TC2 sea falsa, TC3 determina cuál de las dos técnicas restantes es potencialmente más eficiente. El rendimiento de `LOCALWRITE` es inferior a `SLCCLS` en dos situaciones:

1. El lazo tiene asociado una alta carga computacional. Aunque el número de las iteraciones compartidas de la técnica `LOCALWRITE` sea reducido, el coste asociado a la replicación de las computaciones puede influir de forma importante sobre la eficiencia de esta propuesta.

2. Existe una alta conectividad. Esta condición implica un aumento importante del número de iteraciones replicadas.

Conclusiones y principales aportaciones

En esta tesis hemos establecido las bases para la caracterización y optimización de códigos irregulares, prestando especial atención a su ejecución paralela sobre multiprocesadores de memoria compartida. La motivación fundamental de este trabajo radica en que la eficiencia del código paralelo generado por las estrategias existentes resulta limitada, siendo únicamente competitivas en situaciones concretas. Por ejemplo, en operaciones de reducción irregular es posible explotar las propiedades asociativas y conmutativas de los vectores de indirección, permitiendo utilizar técnicas de paralelización que no requieren del empleo de una rutina de inspección.

Debido a la naturaleza de los códigos irregulares, únicamente pueden ser completamente analizados durante la ejecución del programa, momento en el que también se deben aplican las distintas técnicas de mejora del rendimiento. Dado que este análisis se realiza en tiempo de ejecución, su coste repercute de forma negativa sobre el rendimiento del programa optimizado. Así pues, es necesario introducir nuevos mecanismos de análisis que tengan un bajo impacto en la eficiencia del programa. En este contexto, encontramos que no se han desarrollado representaciones eficientes del patrón de acceso de un programa que permitan a las rutinas de análisis disminuir su complejidad.

Otro tópico de gran importancia es la explotación eficiente de la jerarquía de memoria del sistema, la cual ejerce una gran influencia en la ejecución secuencial y/o paralela del programa. Así pues, entendemos que es un requisito indispensable, para la optimización de códigos irregulares, el introducir las transformaciones necesarias que permitan maximizar la localidad en los accesos a memoria. Adicionalmente, si deseamos paralelizar el programa de forma eficiente, debemos obtener un buen balanceo de la carga computacional y minimizar el coste de las sincronizaciones y comunicaciones entre los procesadores.

Este trabajo ha abordado tanto el desarrollo de técnicas de caracterización de códigos irregulares, como la introducción de distintos procesos de optimización sobre esta clase de códigos. A continuación, resumimos las principales aportaciones de esta tesis.

- Hemos introducido un nuevo modo de caracterización del conjunto de posiciones de memoria accedidas por un código irregular. Esta caracterización se basa en obtener una representación geométrica de los valores contenidos en el vector de indirección mediante la representación IARD. Dicha representación presenta las siguientes características:

  1. El coste de generación de esta estructura es pequeño, y puede ser minimizado solapándolo con la ejecución del propio código.
  2. Su coste de almacenamiento es muy reducido, típicamente de varios órdenes de magnitud inferior al de la indirección.
  3. El coste asociado al acceso a esta representación también es muy reducido, lo cual permite su empleo de forma eficiente en un gran número de aplicaciones.
  4. Hemos introducido mecanismos que permiten transformar esta representación adaptándola a las características del código irregular. Dicha transformación es dependiente del esquema de acceso al vector de indirección y su coste computacional es muy reducido.

- Hemos desarrollado una técnica de análisis de dependencias entre dos secciones de código irregular. Nuestra propuesta hace uso de la caracterización IARD para obtener una representación del patrón de acceso asociado a cada una de las regiones. Las principales características de esa propuesta son:

  1. Es capaz de determinar si ambas secciones del código acceden a distintas regiones de memoria, y, por lo tanto, determinar la no existencia de dependencias asociadas a sus accesos.
  2. Es capaz de determinar la existencia de riesgo de dependencias sobre una determinada región de memoria, delimitando de forma precisa dicha región e identificando el conjunto de entradas de los vectores de indirección que acceden sobre la misma.
  3. El análisis de dependencias se realiza de forma analítica y su coste es muy reducido.

- Hemos realizado un esquema de clasificación de los códigos irregulares atendiendo al número de indirecciones que contienen y a las propiedades que presentan sus accesos.

En particular, hemos diferenciado los accesos basados en las reducciones irregulares, dado que estas permiten el empleo de técnicas de optimización específicas.

- Hemos abordado la paralelización de códigos irregulares con una indirección. Se ha propuesto un conjunto de técnicas de paralelización que tiene las siguientes características:

  1. Hemos desarrollado un conjunto de rutinas de inspección con el objetivo de abordar eficientemente un amplio espectro de situaciones.
  2. Las rutinas de inspección tienen un bajo coste asociado, debido al empleo de la caracterización IARD.
  3. El código paralelo explota de forma eficiente la jerarquía de memoria del sistema, obteniendo una alta localidad en los accesos.
  4. Nuestras propuestas mantienen el orden de los accesos a memoria del código original, pudiendo ser aplicadas no sólo a reducciones irregulares, sino a códigos en los que el orden de dichos accesos debe ser preservado.
  5. Hemos evaluado bajo un gran número de situaciones la eficiencia de las técnicas más relevantes existentes en la actualidad. Nuestras propuestas obtienen unos resultados competitivos para la práctica totalidad de los escenarios considerados.

- Hemos desarrollado un algoritmo que permite realizar el balanceo de la carga en códigos irregulares con una indirección. Sus principales características son:

  1. Su funcionamiento está basado en una parametrización de la estructura del código paralelo. Variando los valores de esos parámetros puede evaluar el balanceo de la carga de distintas técnicas de paralelización, incluidas todas nuestras propuestas.
  2. El coste de procesamiento y almacenamiento que tiene asociado es muy reducido debido al empleo de la caracterización IARD.

- Hemos estudiado técnicas de mejora de localidad en códigos secuenciales, probando que este factor ejerce un gran impacto sobre el tiempo de ejecución del programa. Adicionalmente, hemos desarrollado una técnica que permite explotar este factor y optimizar la ejecución de códigos irregulares secuenciales.

- Hemos abordado la paralelización de códigos irregulares con un número arbitrario de indirecciones y diferentes grados de paralelismo. Hemos introducido una nueva

caracterización de esta clase de códigos mediante la especificación de un nuevo criterio que distingue dos formas de ejecución de los distintos estamentos que conforman el lazo: ejecución en orden y ejecución fuera de orden.

- Para aquellos lazos irregulares con un número arbitrario de indirecciones y cuyos estamentos se deben ejecutar en orden, hemos desarrollado dos técnicas de paralelización que explotan de manera eficiente la localidad en los accesos a memoria. Hemos evaluado su eficiencia de forma exhaustiva, probando el beneficio que suponen en el rendimiento del código paralelo.

- Para aquellos lazos irregulares con un número arbitrario de indirecciones y cuyos estamentos se pueden ejecutar fuera de orden, presentamos dos propuestas que abordan su paralelización. Hemos probado que dichas propuestas obtienen un rendimiento superior a las existentes y permiten aumentar, aún más, el grado de localidad en los accesos.

- Hemos estudiado la eficiencia de las distintas técnicas de paralelización existentes en códigos irregulares con varias indirecciones y cuyos accesos se realizan exclusivamente mediante operaciones de reducción. Hemos identificado los parámetros que más influyen en el rendimiento del programa paralelo. En base a este estudio hemos desarrollado una nueva propuesta cuyas principales características son:

  1. Explota eficientemente la localidad en los accesos, tanto de lectura de los vectores de indirección, como de escritura sobre la matriz de reducción.
  2. Extrae el paralelismo existente en el programa, reduciendo tanto en número, como en coste, las operaciones de comunicación y sincronización entre los procesadores. Estas operaciones guardan una dependencia mínima con el número de procesadores, lo que hace que nuestra propuesta resulte altamente escalable.
  3. Incluye una propuesta de distribución de las iteraciones, con la cual se obtiene un alto balanceo de carga.
  4. Hemos contenido el gasto de memoria del inspector, haciendo que este no dependa ni del número de iteraciones ni del de procesadores.
  5. La rutina de inspección puede ser paralelizada de forma eficiente. El impacto de la paralelización del inspector sobre el rendimiento del ejecutor es muy reducido.

- Finalmente, hemos establecido criterios para la determinación de las técnicas de paralelización más eficientes para cada problema particular. Nuestra propuesta no sólo emplea información acerca de la estructura del código (obtenida en tiempo de

compilación) sino también información acerca de la estructura del patrón de acceso (obtenida mediante la caracterización IARD).

Existen diversas líneas de investigación que se pueden derivar del trabajo de esta tesis. A continuación enunciamos algunas de las más relevantes.

- Un aspecto de gran interés es la evaluación de la eficiencia de algoritmos de reordenamiento sobre el patrón de acceso. Mediante estos algoritmos es posible aumentar el grado de localidad mediante un reetiquetado de los distintos elementos. Este estudio tiene especial importancia en el caso de los problemas de n-cuerpos, dado que la estructura del patrón ejerce una gran influencia sobre el rendimiento del código paralelo. En este sentido hemos realizado diversas pruebas preliminares que muestran un aumento importante en el grado de localidad del patrón reordenado. Cabe esperar que mediante una técnica de reordenación, el rendimiento de nuestras propuestas experimente una mejora significativa.

- Otro aspecto interesante es la elaboración de un árbol de decisión en el que los umbrales en los parámetros evaluados estén establecidos de manera precisa. El valor concreto de estos umbrales es dependiente tanto de la arquitectura del sistema paralelo como del modelo de paralelización utilizado. Para obtener estos valores es necesario evaluar la eficiencia de las distintas propuestas de forma exhaustiva sobre un gran número de arquitecturas.

- Finalmente, proponemos el desarrollo de nuevos algoritmos que aborden problemas de mejora del rendimiento y que permitan explotar más eficientemente la información almacenada en la caracterización IARD. Esta caracterización ofrece información precisa acerca de la distribución de los accesos en memoria. Basándonos en la estructura de las curvas $\mathcal{E}_f^u$ y $\mathcal{E}_f^l$ se puede, por ejemplo, diseñar algoritmos que realicen el balanceo de la carga en base a un modelo analítico del patrón de acceso. De este modo conseguimos técnicas mucho más precisas y rápidas, dado que el resultado final se obtendría de forma analítica, sin tener que estimar mediante un proceso iterativo el valor óptimo de balanceo de la carga.

Apéndice A

Arquitecturas empleadas

En este apéndice se realiza una descripción de las principales características de los sistemas utilizados en esta tesis.

Silicon Graphics Origin 2000

El sistema Origin 2000 es una arquitectura *Scalable Shared-memory Multoprocessing* S2MP de tipo CC-NUMA. El sistema concreto que hemos utilizado está constituido por un conjunto de 16 nodos unidos por una red de interconexión de alta velocidad. Cada nodo dispone de 2 procesadores RISC MIPS R10000 a 250 MHz y 512 MB de memoria, conformando un sistema con 32 procesadores y 8192 MB de memoria. La topología de la red es un *bristled fat hypercube* con un ancho de banda de hasta 800 MB/s en comunicaciones entre nodos y 3200 MB/s en accesos a memoria local. El sistema operativo empleado es el IRIX64 6.5 IP27 de 64 bits.

La coherencia de los datos se realiza a nivel de línea cache mediante un protocolo basado en directorios. Este protocolo es gestionado a nivel hardware y su almacenamiento se realiza en una porción de la memoria principal, que suele ser inferior al 6% del espacio global de memoria. La memoria principal del sistema se organiza en páginas de tamaño comprendido entre 4 KB y 16 MB. Este sistema incorpora un mecanismo de migración

de páginas que permite reubicar su posición en los nodos que realizan un mayor uso de la página considerada. Adicionalmente, el sistema incorpora un mecanismo de replicación de páginas para aquellas que únicamente son leídas.

Tanto la memoria cache primaria como la secundaria son no bloqueantes. La memoria cache secundaria está unificada para instrucciones y datos. Tiene un tamaño de 4 MB con una arquitectura SSRAM y un tamaño de línea de 128 bytes (32 palabras). Es una memoria asociativa de dos vías con un protocolo de reemplazo LRU y está implementada mediante un bus dedicado de 128 bits a 250 MHz. Por otra parte, la memoria cache primaria está dividida en dos memorias de 32 KB cada una y con un uso exclusivo para instrucciones o para datos. Nuevamente, es una memoria asociativa de dos vías, y en el caso de la memoria destinada al almacenamiento de datos, se emplea algoritmo de reemplazo LRU. La Tabla A.1 muestra las latencias expresadas en *ns* asociadas a los diferentes niveles de jerarquía de este sistema.

Cache primaria	Cache secundaria	Memoria local	Memoria no local*
8-12 ns	32-48 ns	318 ns	836 ns

*Tiempo máximo de acceso en un sistema con 32 procesadores

Tabla A.1: Latencia de acceso asociada al MIPS R10000.

Silicon Graphics Origin 200

El sistema usado cuenta con cuatro procesadores MIPS R10000 a 225 MHz. La arquitectura de este sistema es similar a al del Origin 2000. Como principales diferencias respecto a esta, destacamos una memoria cache secundaria de tamaño 2 MB y una memoria principal de 1024 MB de capacidad. La plataforma utilizada emplea un sistema operativo IRIX 6.4 IP32.

SUN HPC 4500

El sistema HPC 4500 que hemos utilizado consta de 12 procesadores UltraSPARC II a 400 MHz interconectados por una red Gigaplane con una velocidad de transferencia de 2.5 GB/s. Este sistema se basa en una arquitectura UMA en la que tanto los procesadores como los bancos de memoria están distribuidos en distintos nodos. La memoria principal del sistema es de 4 GB, con una cache secundaria de 4 MB por procesador y una cache primaria de 16 KB para instrucciones y 16 KB para datos. El sistema operativo empleado es el Solaris SunOS 5.6.

SUN Enterprise 250

Este sistema adopta una arquitectura UMA dado que consta de un único nodo con dos procesadores UltraSPARC II a 296 MHz. Como sistema de interconexión emplea un bus UPA de 128 bits de datos a 100 MHz. La memoria cache secundaria es de 2 MB por procesador, y la primaria es de 16 KB exclusivos para instrucciones y para datos. La memoria principal del sistema es de 768 MB. El sistema operativo es el Solaris SunOS 5.7.

Bibliografía

[1] A. Aggarwal, J. S. Chang y C. K. Yap. Minimum area circumscribing polygons. *The Visual Computer* **1**(2), 112–117 (octubre 1985).

[2] A. Aggarwal y J. Park. Notes on searching in multidimensional monotone arrays. En "Proceedings of the 29th Annual Symposium on Foundations of Computer Science", páginas 497–512 (octubre 1988).

[3] S. P. Amarasinghe. "Parallelizing Compiler Techniques Based on Linear Inequalities". Tesis Doctoral, Stanford University, Computer Systems Laboratory (1997).

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu y W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer* **29**(2), 18–28 (febrero 1996).

[5] C. Ancourt y F. Irigoin. Scanning polyhedra with DO loops. En "Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming", páginas 39–50 (abril 1991).

[6] T. Asano, N. Katoh, E. Lodi y T. Roos. Optimal approximation of monotone curves on a grid. En "Proceedings of the 7th Canadian Conference on Computational Geometry", páginas 37–42 (agosto 1995).

[7] F. Attneave. Some informational aspects of visual perception. *Psychological Review* **61**, 183–193 (1954).

[8] V. Balasundaram y K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. En "Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation", páginas 41–53 (1989).

[9] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy y E. Su. The Paradigm compiler for distributed-memory multicomputers. *IEEE Computer* **28**(10), 37–47 (octubre 1995).

[10] U. Banerjee. "Dependence analysis for supercomputing". Kluwer Academic (1988).

[11] R. E. Bank. "PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, Users' Guide 7.0". SIAM (1994).

[12] S. T. Barnard y H. D. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience* **6**(2), 101–117 (1994).

[13] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, C. Romine y H. van der Vorst. "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods". SIAM (1994).

[14] M. D. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum y J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications* **3**(3), 5–40 (1989).

[15] A. J. C. Bik. "Compiler Support for Sparse Matrix Computations". Tesis Doctoral, Leiden University (1996).

[16] A. J. C. Bik y H. A. G. Wijshoff. Nonzero structure analysis. En "Proceedings of the International Conference on Supercomputing", páginas 226–235 (1994).

[17] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington y R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software* **28**(2), 135–151 (junio 2002).

[18] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. A. Padua, Y. Paek, B. Pottenger, L. Rauchwerger y Peng Tu. Parallel programming with polaris. *Computer* **29**(12), 78–82 (1996).

[19] W. Blume y R. Eigenmann. Nonlinear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems* **9**(12), 1180–1194 (diciembre 1998).

[20] W. J. Blume. "Symbolic analysis techniques for effective automatic parallelization". Tesis Doctoral, Department of Computer Science, University of Illinois at Urbana-Champaign (1995).

[21] J. Boyce, D. Dobkin, R. Drysdale y L. Guibas. Finding extremal polygons. En "Proceedings of the 14th Annual ACM Symposium, Theory of Computing", páginas 282–289 (1982).

[22] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan y M. Karplus. CHARMM: a program for macromolecular energy minimization and dynamics calculations. *Journal Comput. Chemistry* **4**(2), 187–217 (1983).

[23] J. B. Carter, J. K. Bennett y W. Zwaenepoel. Implementation and performance of Munin. En "Proceedings of the 13th ACM Symposium on Operating Systems Principles", páginas 152–164 (octubre 1991).

[24] R. Chandra. "Parallel programming in OpenMP". Morgan Kaufmann Publishers (2001).

[25] B. M. Chapman, P. Mehrotra y H. P. Zima. Programming in Vienna Fortran. *Scientific Programming* **1**(1), 31–50 (1992).

[26] D.-K. Chen, J. Torrellas y P.-C. Yew. An Efficient Algorithm for the Run-Time Parallelization of DOACROSS Loops. En "Proceedings of the International Conference on Supercomputing", páginas 518–527 (1994).

[27] J. Choi, J. J. Dongarra y D. W. Walker. PB-BLAS: A set of parallel block Basic Linear Algebra Subprograms. En "Proceedings of the Scalable High-Performance Computing Conference", páginas 534–541 (1994).

[28] L. P. Cordella y G. Dettori. An $O(N)$ algorithm for polygonal approximation. *Pattern Recognition Letters* **3**, 93–97 (1985).

[29] B. Creusillet y F. Irigoin. Exact versus approximate array region analyses. *Lecture Notes in Computer Science* **1239**, 86–100 (1997).

[30] D. E Culler y J. P. Singh. "Parallel Computer Architecture". Pitman/MIT Press (1989).

[31] E. Cuthill y J. McKee. Reducing the bandwidth of sparse symmetric matrices. En "Proceedings of the 24th National Conference ACM", páginas 157–172. ACM (1969).

[32] R. Das, J. Saltz, D. Mavriplis, J. Wu y H. Berryman. Unstructured mesh problems, PARTI primitives, and the ARF compiler. En "Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing", páginas 570–572. SIAM (1991).

[33] R. Das, M. Uysal, J. Saltz y Y.-S. S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing* **22**(3), 462–478 (1994).

[34] T. Davis. University of Florida sparse matrix collection. *NA Digest* **97**(23) (Julio 1997).

[35] I. Debled y J. Reveilles. A linear algorithm for segmentation of digital curves. *International Journal of Pattern Recognition and Artificial Intelligence* **9**(6), 635–662 (1995).

[36] DePano y Aggarwal. Finding restricted k–envelopes for convex polygons. En "Proceedings of the 22th Annual Allerton Conference on Communication, Control, and Computing", páginas 81–90 (1984).

[37] I. S. Duff, R. G. Grimes y J. G. Lewis. "Users' Guide for the Harwell-Boeing Sparse Matrix Collection". Boeing Computer Services (1992).

[38] R. Eigenmann, J. Hoeflinger y D. A. Padua. On the automatic parallelization of the Perfect Benchmarks. *IEEE Transactions on Parallel and Distributed Systems* **9**(1), 5–23 (enero 1998).

[39] D. Eppstein, M. Overmars, G. Rote y G. Woeginger. Finding minimum area k-gons. *GEOMETRY: Discrete & Computational Geometry* **7**, 45–58 (1992).

[40] B. L. Evans, C. Schwarz, J. Teich, A. Vainshtein y E. Welzl. Minimal enclosing parallelogram with application. En "Proceedings of the 11th Annual Symposium on Computational Geometry", páginas 34–35. ACM Press (junio 1995).

[41] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua y P. M. Petersen. The Polaris internal representation. *International Journal of Parallel Programming* **22**(5), 553–586 (octubre 1994).

[42] P. Feautrier. Array expansion. En "Proceedings of the International Conference on Supercomputing", páginas 429–441. ACM Press (julio 1988).

[43] The Parallel Computing Forum. PCF parallel fortran extensions. *ACM SIGPLAN Fortran Forum* **10**(3), 1–57 (1991).

[44] C. Fu y T. Yang. Run-time compilation for parallel sparse matrix computations. En "Proceedings of the International Conference on Supercomputing", páginas 237–244. ACM Press (1996).

[45] A. J. García-Loureiro, T. F. Pena, J. M. López-González y Ll. Prat. Parallel finite element method to solve the 3D Poisson equation and its application to abrupt heterojunction bipolar transistors. *International Journal for Numerical Methods in Engineering* **49**(5), 639–652 (2000).

[46] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek y V. Sunderam. "PVM3 User's Guide and Reference Manual". Engineering Physics and Mathematics Division, ORNL (mayo 1993).

[47] Silicon Graphics. MIPSpro compiling and performance tuning guide (1997). Silicon Graphics, Inc., Mountain View, CA.

[48] W. Gropp, E. Lusk y A. Skjellum. "USING MPI Portable Parallel Programming with the Message-Passing Interface". The MIT Press (1994).

[49] J. Gu, Z. Li y G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. En "Proceedings of the Conference on Supercomputing". ACM/IEEE (diciembre 1995).

[50] M. Gupta y R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. En "Proceedings of the Conference on Supercomputing", páginas 1–12 (1998).

[51] E. Gutiérrez, O. Plata y E. L. Zapata. On automatic parallelization of irregular reductions on scalable shared memory systems. En "Proceedings of the Euro-Par International Conference on Parallel and Distributed Computing", páginas 422–429 (1999).

[52] E. Gutiérrez, O. Plata y E. L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. En "Proceedings of the International Conference on Supercomputing", páginas 78–87. ACM SIGARCH (mayo 2000).

[53] E. Gutiérrez, O. Plata y E. L. Zapata. Improving parallel irregular reductions using partial array expansion. En "Proceedings of the High Performance Networking and Computing", páginas 38–46. ACM Press and IEEE Computer Society Press (2001).

[54] E. Gutiérrez. "Paralelización Automática de Reducciones". Tesis Doctoral, Departamento de Arquitectura de Computadores, Universidad de Málaga (2001).

[55] M. W. Hall, J. M. A., S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion y M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer* **29**(12), 84–89 (diciembre 1996).

[56] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao y M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. En "Proceedings of the Conference on Supercomputing". ACM Press and IEEE Computer Society Press (1995).

[57] H. Han. "Locality Transformations for Adaptive Irregular Applications". Tesis Doctoral, Faculty of the Graduate School of the University of Maryland (2001).

[58] H. Han y C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. En "Proceedings of the International Conference on Parallel Architectures and Compilation Techniques", páginas 393–400. IEEE Computer Society Press (octubre 1998).

[59] H. Han y C.-W. Tseng. Improving compiler and run-time support for irregular reductions using local writes. En "Proceedings of the International Workshop on Languages and Compilers for Parallel Computing", páginas 181–196 (1998).

[60] H. Han y C.-W. Tseng. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing* **26**(13–14), 1861–1887 (diciembre 2000).

[61] H. Han y C.-W. Tseng. A comparison of parallelization techniques for irregular reductions. En "Proceedings of the 15th International Parallel & Distributed Processing Symposium", páginas 27–29. IEEE Computer Society (abril 2001).

[62] H. Han y C.-W. Tseng. Improving locality for adaptive irregular scientific codes. *Lecture Notes in Computer Science* **2017**, 173–188 (2001).

[63] P. Havlak y K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems* **2**(3), 350–360 (julio 1991).

[64] D. Heras. "Modelado y mejora de localidad en códigos irregulares". Tesis Doctoral, Universidad de Santiago de Compostela (enero 2000).

[65] D. Heras, J. Cabaleiro y F. F. Rivera. Modeling data locality for the sparse matrix-vector product using distance measures. *Journal of Parallel Computing* **27**, 897–912 (2001).

[66] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer y C.-W. Tseng. An overview of the Fortran D programming system. *Lecture Notes in Computer Science* **589**, 18–34 (1991).

[67] J. Hoeflinger. "Interprocedural parallelization using memory classification analysis". Tesis Doctoral, Department of Computer Science, University of Illinois at Urbana-Champaign (1998).

[68] J. Hoeflinger y Y. Paek. The access region test. En "Proceedings of the 12th International Workshop of Languages and Compilers for Parallel Computing", páginas 271–285. Springer-Verlag (2000).

[69] Y.-S. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das y J. H. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed-memory machines. *Software Practice and Experience* **25**(6), 597–621 (junio 1995).

[70] E.-J. Im y K. Yelick. Model-based memory hierarchy optimizations for sparse matrices. En "Proceedings of the Workshop on Profile and Feedback-Directed Compilation" (1998).

[71] F. Irigoin, P. Jouvelot y R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. En "Proceedings of the International Conference on Supercomputing", páginas 244–251. ACM SIGARCH (junio 1991).

[72] M. Kandemir, A. Choudhary, J. Ramanujam y P. Banerjee. A matrix-based approach to the global locality optimization problem. En "Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques", páginas 306–313. IEEE Computer Society Press (octubre 12–18, 1998).

[73] M. S. Kankanhalli. An adaptive dominant point detection algorithm for digital curves. *Pattern Recognition Letters* **14**, 385–390 (1993).

[74] G. Karypis y V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* **20**(1), 359–392 (1999).

[75] P. Keleher. Update protocols and iterative scientific applications. En "Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing", páginas 675–681. IEEE Computer Society (marzo 1998).

[76] P. Keleher, A. L. Cox y W. Zwaenepoel. Lazy consistency for software distributed shared memory. En "Proceedings the 19th Annual International Symposium on Computer Architecture,ACM SIGARCH", páginas 13–21 (mayo 1992).

[77] P. Keleher y C.-W. Tseng. Enhancing software DSM for compiler-parallelized applications. En "Proceedings of the 11th International Parallel Processing Symposium", páginas 490–499. The Institute of Electrical and Electronics Engineers (abril 1997).

[78] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman y D. Wonnacott. The Omega Library Interface Guide. Informe técnico, Dept. of Computer Science, Univ. of Maryland, College Park (abril 1996).

[79] C. W. Keßler. Applicability of program comprehension to sparse matrix computations. *Lecture Notes in Computer Science* **1300**, 347–351 (1997).

[80] C. W. Keßler y C. H. Smith. The SPARAMAT approach to automatic comprehension of sparse matrix computations. En "Proceedings of the 7th International Workshop on Program Comprehension", páginas 200–207. IEEE Computer Society Press (1999).

[81] D. E. Knuth. "Sorting and Searching", tomo 3 de "The Art of Computer Programming". Addison-Wesley (1975).

[82] C. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr. y M. E. Zosel. "The High Performance Fortran handbook". Scientific and engineering computation. MIT Press (enero 1994).

[83] C. Koelbel y P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. En "Proceedings of the International Conference on Supercomputing", páginas 414–423. ACM SIGARCH (junio 1991).

[84] C. Koelbel, P. Mehrotra y J. Van Rosendale. Supporting shared data structures on distributed memory machines. En "Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming", páginas 177–186 (marzo 1990).

[85] V. Kotlyar, K. Pingali y P. Vinson Stodghill. Compiling parallel code for sparse matrix applications. En "Proceedings of Conference on Supercomputing", páginas 1–18. ACM SIGARCH and IEEE (noviembre 1997).

[86] V. P. Krothapalli y P. Sadayappan. An approach to synchronization for parallel computing. *Proceedings of the International Conference on Supercomputing* páginas 573–581 (julio 1988).

[87] A. Lain, D. R. Chakrabarti y P. Banerjee. Compiler and run-time support for exploiting regularity within irregular applications. *IEEE Transactions on Parallel and Distributed Systems* **11**(2), 119–135 (febrero 2000).

[88] M. Lam. Overview of the SUIF2 system. Tutorial presentation at the ACM SIGPLAN Conference on Programming Language Design and Implementation (2002).

[89] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta y J. Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems* **4**(1), 41–61 (enero 1993).

[90] S.-T. Leung y J. Zahorjan. Extending the Applicability and Improving the Performance of Runtime Parallelization. Informe técnico 95-01-08, Department of Computer Science and Engineering, University of Washington (1995).

[91] S.-W. Liao, A. Diwan, R. P. Bosch, A. Ghuloum y M. S. Lam. SUIF Explorer: an interactive and interprocedural parallelizer. *ACM SIGPLAN Notices* **34**(8), 37–48 (agosto 1999).

[92] Techpubs Library. "Origin 2000 and Onyx2 Performance Tunning and optimization guide. Document number: 007-3430-003". Silicon Graphics inc. (2002).

[93] Y. Lin y D. A. Padua. On the automatic parallelization of sparse and irregular Fortran programs. *Lecture Notes in Computer Science* **1511**, 41–56 (1998).

[94] J. Liu y A. Sherman. Comparative analysis of the Cuthill-McKee and reverse Cuthill-McKee ordering algorithms for sparse matrics. *SIAM Journal Numerical Analysis* **13**, 198–213 (1976).

[95] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony y W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. En "Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming", páginas 48–56 (junio 1997).

[96] M. J. Martín, D. E. Singh, J. Touriño y F. F. Rivera. Exploiting locality in the runtime parallelization of irregular loops. En "Proceedings of the 31th International Conference on Parallel Processing", páginas 27–34 (2002).

[97] M. J. Martín, D. E. Singh, J. Touriño y F. F. Rivera. Improving locality in the parallelization of doacross loops. En "Proceedings of the 8th Euro-Par International Conference on Parallel and Distributed Computing", Lecture Notes in Computer Science, páginas 275–279. Springer-Verlag (2002).

[98] S. P. Midkiff y D. A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers* **36**(12), 1485–1495 (diciembre 1987).

[99] B. Moon, M. Uysal y J. Saltz. Index translation schemes for adaptive computations on distributed memory multicomputers. En "Proceedings of the 9th International Symposium on Parallel Processing", páginas 812–819. IEEE Computer Society Press (abril 1995).

[100] D. M. Mount y R. Silverman. Minimum enclosures with specified angles. Technical Report CS-TR-3219, University of Maryland, College Park (febrero 1994).

[101] A. G. Navarro, R. Asenjo, E. L. Zapata y D. A. Padua. Access descriptor based locality analysis for distributed-shared memory multiprocessors. En "Proceedings of the 28th International Conference on Parallel Processing", páginas 86–94 (septiembre 1999).

[102] J. Nieplocha, R. J. Harrison y R. J. Littlefield. Global arrays: a portable programming model for distributed memory computers. En "Proceedings of the International Conference on Supercomputing", páginas 340–349 (1994).

[103] J. O'Rourke, A. Aggarwal, S. Maddila y M. Baldwin. An optimal algorithm for finding minimal enclosing triangles. *Journal of Algorithms* **7**, 258–269 (1986).

[104] Y. Paek. "Automatic Parallelization for Distributed Memory Machines based on Access Region Analysis". Tesis Doctoral, University of Illinois at Urbana-Champaign (1997).

[105] Y. Paek, J. Hoeflinger y D. A. Padua. Simplification of array access patterns for compiler optimizations. En "Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation", páginas 60–71 (1998).

[106] Y. Paek, J. Hoeflinger y D. A. Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems* **24**(1), 65–109 (enero 2002).

[107] D. Patel y L. Rauchwerger. Implementation Issues of Loop-Level Speculative Run-Time Parallelization. En "Proceedings of the 8th International Conference on Compiler Construction", páginas 183–197 (Amsterdam, The Netherlands, 1999).

[108] S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics* **117**, 1–19 (March 1995).

[109] C. D. Polychronopoulos. Advanced loop optimizations for parallel computers. En "Proceedings of the International Conference on Supercomputing", tomo 297, páginas 255–277. Springer-Verlag (junio 1987).

[110] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang y G. Fox. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems* **6**(8), 815–831 (agosto 1995).

[111] A. Pothen, H. D. Simon y K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications* **11**(3), 430–452 (julio 1990).

[112] SUN product documentation. "Fortran 77 4.0 User's Guide". SUN Microsystems.

[113] W. Pugh. A practical algorithm for exact array dependency analysis. *Communications of the ACM* **35**(8), 102 (agosto 1992).

[114] W. Pugh y D. Wonnacott. Experiences with constraint-based array dependence analysis. En "Principles and Practice of Constraint Programming", páginas 312–325. Springer-Verlag (1994).

[115] J. R. Quinlan. Improved use of continuous attributes in C4.5. *Journal of Artificial Intelligence Research* **4**, 77–90 (1996).

[116] J. Ramanujam, S. Dutta y A. Venkatachar. Code generation for complex subscripts in data-parallel programs. *Lecture Notes in Computer Science* **1366**, 49–63 (1998).

[117] L. Rauchwerger. "Run-time parallelization: A framework for parallel computation". Tesis Doctoral, Department of Computer Science, University of Illinois at Urbana-Champaign (1995).

[118] L. Rauchwerger, N. M. Amato y D. A. Padua. Run-time methods for parallelizing partially parallel loops. En "Proceedings of the International Conference on Supercomputing", páginas 137–146. ACM Press (1995).

[119] L. Rauchwerger y D. A. Padua. The privatizing DOALL test: a run-time technique for DOALL loop identification and array privatization. En "Proceedings of the International Conference on Supercomputing", páginas 33–43 (1994).

[120] L. Rauchwerger y D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. En "Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation", páginas 218–232 (junio 1995).

[121] R. Rosin. Techniques for assessing polygonal approximations of curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **19**(6), 659–666 (1997).

[122] Y. Saad. Sparskit: A basic tool kit for sparse computations (1994).

[123] J. Saltz, H. Berryman y J. Wu. Runtime compilation for multiprocessors. *Concurrency, Practice and Experience* **3**(6), 573–592 (1991).

[124] J. Saltz, R. Mirchandaney y K. Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers* **40**(5), 603–612 (1991).

[125] J. Saltz, R. Ponnusamy, S. D. Sharma, B. Moon, Y.-S. Hwang, M. Uysal y R. Das. A manual for the CHAOS runtime library. Informe técnico CS-TR-3437, University of Maryland Institute for Advanced Computer Studies Department of Computer Science (marzo 1995).

[126] M. A. Silva. "Compiler Framework for the automatic detection of loop-level parallelism". Tesis Doctoral, Universidad de A Coruña (enero 2003).

[127] D. E. Singh, M. J. Martín y F. F. Rivera. The envelope of a digital curve based on dominant points. En "Proceedings of the 9th International Conference of Discrete Geometry for Computer Imagery", Lecture Notes in Computer Science, páginas 452–463. Springer-Verlag (2000).

[128] D. E. Singh, M. J. Martín y F. F. Rivera. Run-time characterization of irregular accesses applied to parallelization of irregular reductions. En "Proceedings of the Workshop on High Performance Scientific and Engineering computing with Applications in conjunction with the International Conference on Parallel Processing", páginas 17–22 (2001).

[129] D. E. Singh, M. J. Martín y F. F. Rivera. Improving load balance of the owner-computes rule in irregular reductions. En "Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics", páginas 356–361 (2002).

[130] D. E. Singh, M. J. Martín y F. F. Rivera. Paralelización de lazos con accesos irregulares. En "XIII Jornadas de Paralelismo", páginas 147–152 (2002).

[131] D. E. Singh, M. J. Martín y F. F. Rivera. Run–time support for parallel irregular assignments. En "Proceedings of the 6th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers", páginas 19–24 (2002).

[132] D. E. Singh, M. J. Martín y F. F. Rivera. Automatic generation of optimized parallel code for n-body simulations. En "Proceedings of the 5th International Conference on Parallel Processing and Applied Mathematics", Lecture Notes in Computer Science. Springer-Verlag (2003).

[133] D. E. Singh, M. J. Martín y F. F. Rivera. Increasing the parallelism of irregular loops with dependences. En "Proceedings of the 9th Euro-Par International Conference on Parallel and Distributed Computing", Lecture Notes in Computer Science, páginas 287–296. Springer-Verlag (2003).

[134] E. Swanson y T. P. Lybrand. PVM-AMBER: a parallel implementation of the AMBER molecular mechanics package for workstation clusters. *Journal of Computational Chemistry* **16**(9), 1131–1140 (septiembre 1995).

[135] C. H. Teh y R. T. Chin. On the detection of dominant points on digital curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **11**, 859–872 (1989).

[136] F. Tip. A survey of program slicing techniques. *Journal of programming languages* **3**, 121–189 (1995).

[137] R. Triolet, F. Irigoin y P. Feautrier. Direct parallelization of CALL statements. *ACM SIGPLAN Notices* **21**(7), 176–185 (julio 1986).

[138] P. Tu. "Automatic array privatization and demand-driven symbolic analysis". Tesis Doctoral, University of Illinois at Urbana-Champaign (1995).

[139] P. Tu y D. A. Padua. Automatic array privatization. En "Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing", Lecture Notes in Computer Science, páginas 500–521. Springer-Verlag (agosto 1993).

[140] R. v. Hanxleden. Handling irregular problems with Fortran D - A preliminary report. En "Proceedings of the 4th Workshop on Compilers for Parallel Computers", páginas 353–364 (diciembre 1993).

[141] R. v. Hanxleden y K. Kennedy. Give-n-take – a balanced code placement framework. En "Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation", páginas 107–120. ACM Press (junio 1994).

[142] F. Voisin y G. R. Perrin. Sparse computation with PEI. *International Journal of Foundations of Computer Science* **10**(4), 425–442 (1999).

[143] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering* **SE-10**(4), 352–357 (julio 1984).

[144] R. G. Wilmoth. Direct simulation Monte Carlo analysis of rarefied flows on parallel processors. *AIAA Journal of Thermophysics and Heat Transfer* **5**(1), 292–300 (1991).

[145] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam y J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices* **29**(12), 31–37 (diciembre 1994).

[146] M. E. Wolf y M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* **2**(4), 452–471 (octubre 91).

[147] J. Wu, J. Saltz, H. Berryman y S. Hiranandani. Distributed memory compiler design for sparse problems. ICASE Report 91-13, Institute for Computer Application in Science and Engineering (enero 1991).

[148] C. Xu. Effects of Parallelism Degree on Run-Time Parallelization of Loops. En "Proceedings of the Hawaii International Conference on System Sciences", páginas 86–95 (1998).

[149] C. Xu y V. Chaudhary. Time-stamping algorithms for parallelization of loops at run-time. En "Proceedings of the 11th International Parallel Processing Symposium", páginas 443–450 (abril 1997).

[150] C. Xu y V. Chaudhary. Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences. *IEEE Transactions on Parallel and Distributed Systems* **12**(5), 433–450 (2001).

[151] H. Yu, F. Dang y L. Rauchwerger. Parallel reductions: An application of adaptive algorithm selection. En "Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing" (2002).

[152] H. Yu y L. Rauchwerger. Adaptive reduction parallelization techniques. En "Proceedings of the International Conference on Supercomputing", páginas 66–77. ACM SIGARCH (mayo 2000).

[153] M. Zagha, B. Larson, S. Turner y M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. En "Proceedings of the ACM/IEEE Conference on Supercomputing", páginas 173–188 (1996).

[154] C.-Q. Zhu y P.-C. Yew. A scheme to enforce data dependence on multiprocessor systems. *IEEE Transactions on Software Engineering* **13**(6), 726–739 (junio 1987).