Factorizacion LU de Matrices Dispersas en Multiprocesadores

Rafael Asenjo Plaza

December 1997
Technical Report No: UMA-DAC-97/32

Published in:
Ph.D. Thesis
December 15, 1997

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

Departamento de Arquitectura de Computadores
Universidad de Málaga

TESIS DOCTORAL

Factorización LU de matrices dispersas en multiprocesadores

Rafael Asenjo Plaza

Málaga, 15 de Diciembre de 1997

Dr. Emilio López Zapata
Catedrático del Departamento de
Arquitectura de Computadores de
la Universidad de Málaga

CERTIFICA:

Que la memoria titulada *"Factorización LU de matrices dispersas en multiprocesadores"*, ha sido realizada por D. Rafael Asenjo Plaza bajo mi dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y concluye la Tesis que presenta para optar al grado de Doctor en Ingeniería de Telecomunicaciones.

Málaga, 31 de Octubre de 1997

Fdo: Dr. Emilio López Zapata
Director de la Tesis Doctoral
Director del Departamento de
Arquitectura de Computadores

A Mª Ángeles.

Muchos años después, [...] Aureliano Buendía había de recordar aquella tarde remota en que su padre lo llevó a conocer el hielo.

Cien años de soledad.
Gabriel García Márquez.

"Cualquier tecnología suficientemente avanzada es indistinguible de la magia."

Tercera ley de Arthur C. Clarke.

Agradecimientos

Muchas veces me he planteado la opción de comenzar y acabar esta sección únicamente con la palabra "Gracias."... y nada más. Sencillamente por evitar el resumir en una sola página la mención de tantas personas a las que debo mucho:

A mi director de Tesis, Emilio L. Zapata, fuente de motivación y al que considero mi maestro y ejemplo de dedicación. Muchas gracias por, como tú dices, "enseñarnos a pescar".

A David Padua por la atención, dedicación en largas horas de reuniones, y su espléndida hospitalidad. También a Iain Duff por sus consejos y supervisión. A Juan Touriño, Oscar Plata, Eladio Gutiérrez, Guillermo Pérez, Gerardo Bandera y Manuel Ujaldón, les debo su productiva colaboración.

A mis padres, siempre preocupados por mi felicidad, alentadores y reconfortantes en los momentos bajos y a los que es imposible pagar su cariño como se merecen, también debo este trabajo. A mi hermana, cómplice y amiga, tanto en la infancia como en la madurez.

A la memoria de Gabriel y Adelaida, y a María Teresa, que gracias a su esfuerzo y lucha son, seguramente en un mayor porcentaje de lo que incluso yo creo, responsables de la gran cantidad de oportunidades que he tenido.

A los amigos y compañeros de departamento, Nico, Mario, Juan, Oswaldo, Manuel, Julio, Felipe, Pablo, Julián, Andrés, Francisco, Mª Antonia, Patricia, Ezequiel y Mª Carmen, maestros del trabajo bien hecho y a todos los que mantienen día a día el departamento como un lugar de camaradería, alegre, distendido y bullicioso de trabajo.

A Nacho, Javi, Pedro, Juan, Miguel, Jesús, José, Pepe y Ester, y demás amigos pese a la distancia y los años.

A los miembros del departamento de Computer Science de la Univ. de Illinois, especialmente a Yuan Lin, Bill Pottenger y Sheila Clark. También a Jacko Koster y los demás compañeros del Parallel Algorithm Team de CERFACS (Francia).

A K.I.M. McKinnon y su grupo del Depto. de Matemáticas de la Univ. de Edimburgo, así como a los miembros del EPCC de la misma universidad, por su soporte e introducción en la programación paralela del Cray T3D.

También agradecemos al CIEMAT (Madrid), CINECA (Italia), EPFL (Suiza) y Cray Research Inc. (Minnesota, USA), por habernos brindado acceso a sus plataformas Cray T3D y T3E.

Agradecemos el soporte y la financiación de este trabajo y de las estancias realizadas en el extranjero a los proyectos TIC96-1125-C03 de la CICYT, BRITE-EURAM III BE95-1564 de la Unión Europea, ERB4050P1921660 del programa de Movilidad y Capital Humano y al programa TRACS (Training and Research on Advanced Computing Systems) del Centro de Computación Paralela de Edimburgo (EPCC).

Aunque en último lugar, con mayor énfasis quiero agradecer a Mª Ángeles todo el cariño, apoyo incondicional, alegrías, colaboración y comprensión que ella me da, y con la cual vivo cada día con una sonrisa en la cara.

Índice General

Prefacio

A menudo nos encontramos frente a compromisos del tipo prestaciones/coste. Esta dualidad que se manifiesta en todos los ámbitos, aparece, como no, en muchos de los párrafos de esta memoria. Ya en la propia tarea de realizar una tesis, aflora un primer compromiso: ante un problema, bien definido por el director de tesis, al estudiante se le presentan diversos caminos sin explorar. Cuando compensa seguir por un camino sin saber lo que hay al final, o volver a atrás para tomar otra dirección es una cuestión de intuición y presenta un ineludible compromiso. Con ingenio y perseverancia se pueden encontrar distintas soluciones al final de sus correspondientes caminos. Si se ha explorado una buena parte del mapa y se han topografiado las regiones hasta entonces vírgenes, el relato del viaje y la descripción de las soluciones, constituyen una metodología apropiada para realizar la tesis. Es claramente una tarea de investigación, pero no podemos negar que en el proceso también hay una gran componente de ingeniería, donde aparece el compromiso entre el tiempo invertido y el número de caminos explorados.

En la presente tesis, el problema estaba bien definido: encontrar soluciones paralelas eficientes para algoritmos de factorización dispersa LU y aplicar las técnicas encontradas en el camino a otros problemas irregulares. Nosotros hemos explorado tres rutas: paralelización manual, semi-automática y por último, automática. En este punto, aparece de nuevo el compromiso entre las prestaciones del código generado y el coste en términos de tiempo de desarrollo. Es decir, la paralelización manual consigue las mejores prestaciones a cambio de un gran esfuerzo en el desarrollo y depuración de los códigos. La situación al paralelizar usando herramientas automáticas es recíproca, situándose la paralelización semi-automática en un punto intermedio entre los otros dos paradigmas.

La motivación que soporta el esfuerzo que realizamos en resolver problemas irregulares es definitiva: se estima que al menos el cincuenta por ciento de todos los programas científicos son irregulares y que más del cincuenta por ciento de todos los ciclos de reloj consumidos en supercomputadores son consumidos por este tipo de códigos.

Dentro de los problemas irregulares hemos elegido como caso de estudio la resolución de sistemas de ecuaciones dispersos. Esta elección se justifica en que este problema abarca varios de los paradigmas presentes en otros códigos irregulares. De esta forma, las técnicas encontradas son útiles a otros problemas con similares características al primero: principalmente algoritmos que presentan llenado o movimiento de datos dentro de una estructura irregular.

De cualquier modo, la resolución paralela de sistemas de ecuaciones dispersos es un problema relevante por sí sólo. Así como es una realidad que los grandes problemas necesitan grandes computadores para ser resueltos, es igualmente cierto que en el corazón de la mayoría de los problemas computacionales a gran escala está la solución de grandes sistemas de ecuaciones dispersos.

Sin embargo, durante la realización de esta tesis, hemos querido ir más lejos, dirigiendo nuestra búsqueda con el objetivo de encontrar soluciones generales para la paralelización de códigos irregulares, y no el de resolver únicamente el problema particular de la factorización de matrices dispersas. En este punto es donde entran en juego las técnicas de compilación ya sean automáticas o semi-automáticas.

De este modo, la memoria se organiza en dos partes bien diferenciadas: la primera aborda directamente el problema de la paralelización manual de códigos dispersos de resolución de ecuaciones; la segunda parte trata de aplicar los conocimientos adquiridos en la primera, para extraer técnicas genéricas que puedan ser aplicadas automática o semi-automáticamente mediante un compilador.

La primera parte tiene tres capítulos. En el primero, formulamos el problema a resolver, enunciamos las distintas alternativas para abordarlo y profundizamos en una de ellas. Además describimos las características relevantes y las estructuras de datos apropiadas al trabajar con matrices dispersas. En el capítulo 2 introducimos los conceptos básicos de paralelismo (arquitecturas y programación paralela) utilizados a lo largo de esta memoria de forma que podamos seguir sin problemas su lectura. Finalmente en el capítulo 3 desarrollamos un eficiente programa paralelo para la resolución de sistemas de ecuaciones dispersos. Este algoritmo es evaluado experimentalmente y comparado con un algoritmo estándar de resolución, comprobándose las buenas prestaciones que proporciona tanto en secuencial como en paralelo. Además de este resultado, el manejo con este algoritmo nos ha permitido familiriazarnos con el problema, adquiriendo una visión global que nos facilita ahora abordar la segunda parte de la tesis. En esta primera parte nos ha sido de gran ayuda la colaboración con Iain Duff durante nuestra estancia en CERFACS, así como el entrenamiento en la programación del Cray T3D que nos brindaron en el EPCC de la Universidad de Edimburgo. Los trabajos publicados de esta primera parte de la tesis se encuentran en [17, 21].

La segunda parte la organizamos en dos capítulos. Abordamos primero la paralelización semi-automática de códigos dispersos en el capítulo 4. Aquí se determinan las limitaciones de los compiladores actuales de paralelismo de datos al intentar expresar códigos dispersos de factorización. Para salvar esas limitaciones proponemos una extensión sencilla y eficiente al lenguaje HPF-2 que resulta válida para otros problemas dispersos o irregulares. En el capítulo 5 se da un paso más en complejidad, tratando de delegar completamente el problema de la paralelización en el compilador. La tecnología actual de compiladores paralelos no es capaz de manejar eficientemente problemas irregulares, por lo que presentamos en este capítulo algunas técnicas que permiten solventar este inconveniente. Para ello hemos partido de un algoritmo secuencial de factorización dispersa, identificando las técnicas necesarias para su paralelización automática. Para justificar la implementación de estas técnicas en un compilador paralelo hemos comprobado que también son aplicables a una gran variedad de códigos numéricos dispersos. En esta segunda parte hemos contado con el apoyo de David Pa-

dua durante nuestras estancias en la Universidad de Illinois. En [19, 18, 14, 145, 15, 16] encontramos más detalladamente la evolución del trabajo desarrollado en esta línea.

Finalmente, en el capítulo de conclusiones, sintetizamos las principales aportaciones de esta tesis, así como enumeramos las principales líneas en las que seguiremos trabajando en el futuro.

Recientemente celebraron en la Universidad de Illinois el nacimiento simbólico de HAL 9000, que en la novela de Arthur C. Clarke "2001: A Space Odyssey" despierta en Enero de 1997 en el Digital Computer Laboratory de la misma Universidad. Es cierto que aún no existe ninguna máquina tan inteligente ni versátil como HAL, pero siempre he sido optimista al imaginarme el futuro y la evolución de la ciencia... Como dijo David J. Kuck con motivo de esta celebración: "La computación paralela práctica es algo escurridizo por el momento. HAL tendría que ser una máquina paralela, y hasta que el problema del paralelismo práctico sea resuelto, no seremos capaces de construir HAL".

Parte I

Algoritmos paralelos de factorización LU dispersa

Capítulo 1

Factorización LU dispersa

No podemos negar, que a medida que las grandes aplicaciones actuales requieren mayores tiempos de computación, resulta inevitable la explotación de máquinas con mayores capacidades de proceso. A la hora de diseñar este tipo de plataformas, el arquitecto de computadores encuentra un elevado número de opciones. Una de las estrategias más ampliamente aceptadas en esta década consiste en utilizar varios procesadores. Dentro de estos sistemas multiprocesadores se están imponiendo aquellos con memoria distribuida o compartida-distribuida, topología de interconexión toro, malla o hipercubo y con filosofía *Single Program Multiple Data* (SPMD) (CRAY T3D/T3E, CM5, Fujitsu AP-1000, Meiko CS2, Origin 2000, etc).

Por otra parte, el álgebra lineal, en particular la solución de sistemas de ecuaciones lineales, constituye el núcleo computacional de muchas de las aplicaciones científicas asistidas por computador. También en el caso en que la matriz del sistema es dispersa (sólo un pequeño porcentaje de los coeficientes de la matriz son no nulos) y de grandes dimensiones, la relevancia del problema genera un considerable interés entre la comunidad científica.

Es cierto que se conoce una significativa cantidad de algoritmos secuenciales para la resolución de dicho problema. Pero dada la incuestionable complejidad asociada a los algoritmos dispersos que involucran un número muy elevado de computaciones, el tiempo de ejecución secuencial puede ser prohibitivo para ciertas aplicaciones. En estas situaciones se impone el uso de técnicas de programación paralela y de explotación de sistemas multiprocesador para minimizar en lo posible el tiempo de ejecución de nuestro algoritmo.

Por tanto uno de los objetivos de esta Tesis Doctoral es aportar soluciones algorítmicas paralelas para la resolución de sistemas de ecuaciones lineales donde la matriz del sistema es dispersa y de grandes dimensiones, para su posterior inclusión en el núcleo de algunas aplicaciones de más alto nivel (problemas de optimización, inversión de matrices, precondicionadores, simulación, etc.). Dichos algoritmos paralelos serán ejecutados sobre sistemas multiprocesador con características similares a las descritas en el primer párrafo.

En la búsqueda de la consecución de ese objetivo irán apareciendo otros proble-

mas que han de ser tratados, como la elección y el manejo de estructuras de datos (dinámicas o no) eficientes, gestión del llenado (en inglés *fill-in*) o crecimiento de la carga computacional durante la resolución del problema, movimiento de datos en el procesado, como ocurre durante la permutación de filas y/o columnas en el pivoteo, etc. Estos problemas no son patrimonio exclusivo del campo de la factorización de matrices, sino que aparecen también en otros ámbitos, en general de problemas irregulares o no estructurados, y por tanto aumenta el interés en su resolución.

Pero, por otro lado, también queremos que las técnicas que nosotros hemos aplicado para paralelizar "a mano" estos códigos de factorización de matrices dispersas, salvando los dos principales problemas: el pivoteo y el llenado, puedan ser también utilizadas por una herramienta de paralelización automática o al menos semi-automática. Un paralelizador automático devuelve un código paralelo a partir de uno secuencial, mientras que a una herramienta semi-automática hay que darle algunas indicaciones adicionales anotadas en el código secuencial.

La motivación para este segundo objetivo es la siguiente. Los multiprocesadores están cada vez más extendidos, de modo que las estaciones de trabajo paralelas son cada vez más accesibles. Incluso podemos decir que la mayoría de los PC's de un futuro cercano serán también paralelos. Evidentemente será necesario desarrollar las aplicaciones y el *software* (SW) paralelo específico para estas nuevas máquinas. Pero con el objetivo de reducir considerablemente los costes de desarrollo de este tipo de SW, es urgente el desarrollo de compiladores que trasladen programas secuenciales convencionales en eficientes programas paralelos.

Existen algunos grupos reconocidos trabajando en el campo del desarrollo de paralelizadores automáticos. Mediante sus compiladores (Polaris [31], PFA [137], Parafrase [113], SUIF [84], etc), son capaces de paralelizar eficientemente muchos de los códigos regulares, sin embargo las prestaciones que ofrecen paralelizando programas irregulares, como los que aparecen al trabajar con matrices dispersas, dejan aún bastante que desear. Nosotros añadiremos en esta memoria algunas posibles estrategias sencillas basadas en la automatización de las técnicas que hemos utilizado para paralelizar *manualmente* códigos simples para la LU en Fortran 77.

Sin embargo, cuando el código a paralelizar es algo más complejo[1], las técnicas automáticas aún son insuficientes. Se hace, por tanto, necesario simplificar el problema y bajar al nivel semi-automático, intermedio entre la paralelización manual y la automática. Bajo esta filosofía, podemos aportar soluciones para códigos de factorización más completos, a costa, claro está, de que el usuario dedique algo de tiempo a añadir las anotaciones necesarias al código secuencial. En particular extenderemos el lenguaje HPF-2 con las directivas necesarias para que nuestros códigos puedan ser expresados y ejecutados en paralelo eficientemente.

El capítulo consta de tres partes bien diferenciadas. Primero, en la sección 1.1, introducimos el problema a resolver, junto con distintas alternativas para abordarlo. Dado que esta memoria se enmarca en el contexto de la computación irregular y más particularmente de las matrices dispersas, dedicamos la sección 1.2 a la caracterización de este tipo de matrices. Adicionalmente, presentamos las distintas opciones que se

[1]Con pivoteo explícito, estructuras de datos dinámicas, Fortran 90, etc.

manejan para almacenar y representar estas matrices dispersas de forma compacta en memoria. La elección de una u otra estructura de datos será definitiva en la eficiencia y complejidad posterior del algoritmo. En la sección 1.3 profundizamos en una de las alternativas presentadas en 1.1 para resolver sistemas de ecuaciones, destacando los aspectos relacionados con la dispersidad de este sistema.

1.1 Formulación del problema

Decimos que una matriz es dispersa si contiene un número suficiente de elementos nulos como para que resulte ventajoso explotar este hecho. Las matrices dispersas son un caso particular de las matrices densas, por lo que en principio podríamos utilizar las mismas técnicas algorítmicas para ambos tipos de matrices. Sin embargo, es evidente que el uso de técnicas específicas para el tratamiento de matrices dispersas redundará en un considerable ahorro tanto de tiempo de computación como de memoria. Es decir, podemos reducir el número de operaciones en punto flotante si evitamos realizar operaciones aritméticas con los ceros y además procuramos, en la medida de lo posible, preservar el coeficiente de dispersión de la matriz durante la computación. También es obvia la disminución en el consumo de memoria ya que los elementos nulos no van a ser almacenados. La matriz puede ser almacenada en un formato comprimido, con alguna información extra (*overhead*) necesaria para una eficiente manipulación de la matriz.

En general, el uso de técnicas dispersas estará justificado sólo en el caso de que la densidad de la matriz sea lo suficientemente baja. Esto es así, debido a que los algoritmos dispersos sufren de un coste computacional considerablemente mayor que el de sus homólogos densos. Por otro lado, también es importante la variación de la densidad de la matriz durante la computación. Es decir, si una matriz en la que dominan los elementos nulos se mantiene dispersa durante la computación tendrá sentido aplicar las técnicas dispersas. Sin embargo, si la matriz se llena significativamente durante su procesado, es conveniente combinar ambas técnicas: debemos elegir el punto a partir del cual deja de ser ventajoso seguir utilizando los métodos dispersos y empieza a tener sentido conmutar a un código denso.

Las matrices dispersas no son sólo un fenómeno informático, aunque esté ampliamente reconocida su importancia en el campo de las ciencias de la computación, ya que actualmente se procesan mediante métodos numéricos. Este tipo de matrices aparecen en problemas tan diversos como los de optimización a gran escala, programación lineal, problemas de planificación, análisis de circuitos, dinámica computacional de fluidos, elementos finitos, y en general, la solución numérica de ecuaciones diferenciales. Muchos de los problemas que son resueltos en términos de redes (grafos) usan matrices dispersas para representar la conectividad, pesada o no, entre los nodos. Si cada nodo esta conectado a sólo unos pocos nodos de la red, la matriz correspondiente contendrá una gran mayoría de coeficientes nulos.

Cheung y Reeves [42] clasifican las aplicaciones de matrices dispersas en tres grupos fundamentales: (i) aplicaciones con matrices de patrón disperso regular, en las que las matrices procesadas tienen una estructura regular, como puedan ser las matrices tridia-

gonales, en banda, triangulares, diagonal, diagonal por bloques, etc; (ii) aplicaciones con matrices de patrón disperso aleatorio; (iii) aplicaciones densas con computación dispersa, en las cuales, aunque las matrices sean densas, se opera con una fracción pequeña y limitada de los datos. Nosotros nos centraremos en la segunda categoría, considerada como el caso más general.

Muchos de los problemas de matrices dispersas envuelven de un modo u otro la solución de un sistema lineal disperso de ecuaciones. Típicamente, la forma de estos sistemas de n ecuaciones es la siguiente:

$$Ax = b \tag{1.1}$$

donde A es una matriz dispersa no singular de $n \times n$ ecuaciones con $\alpha = c \cdot n$ coeficientes no nulos, tal que $\alpha \ll n^2$. El valor c representa el valor medio de elementos no nulos por filas (o por columnas). También encontramos en la ecuación 1.1 un vector conocido de términos independientes, b, de dimensión n, y por otro lado, el vector x de incógnitas de igual dimensión que b. Podemos definir la densidad, ρ, de la matriz por la ecuación $\rho = \alpha/n^2$. Definimos el coeficiente de dispersión como $\sigma = 1 - \rho$.

1.1.1 Resolución de sistemas de ecuaciones dispersos

Como es de suponer, existen multitud de aplicaciones y códigos secuenciales para resolver el problema formulado. Sin embargo, las prestaciones de estas alternativas secuenciales son en muchas ocasiones insuficientes, pese al continuo desarrollo de procesadores de altas prestaciones. En estos casos, agrupar varios de estos procesadores bajo un misma carcasa permite conseguir arquitecturas más potentes en cálculo y con mayores capacidades de memoria. Por tanto si requerimos de nuestra aplicación menores tiempos de ejecución y mayores capacidades de almacenamiento, deberemos modificar nuestro algoritmo para que pueda ser ejecutado por algún potente multiprocesador. Es decir, deberemos desarrollar la versión paralela de nuestra aplicación, para conseguir así las prestaciones deseadas.

El estudio de algoritmos paralelos para resolver el esquema planteado en la sección anterior es importante no sólo por la solución del problema en sí mismo, sino también porque, en general, el tipo de computación requerida lo convierte en un paradigma ideal de la computación científica paralela a gran escala. En otras palabras, un estudio de los métodos directos para sistemas dispersos, engloba muchos de los aspectos que aparecen con frecuencia en problemas científicos, la gran mayoría incluidos entre los que se suelen llamar "Grand Challenge" (grandes desafíos).

Los aspectos principales pueden ser resumidos a continuación:

1. Las operaciones en punto flotante constituyen una pequeña proporción del código total. Por lo general, estas operaciones tienen lugar en bucles que recorren vectores cortos, en los cuales se almacenan de forma comprimida los elementos no nulos. Desde el punto de vista paralelo, este aspecto ya es, de por sí, una fuente de desbalanceo, ya que los vectores tienen distinto número de coeficientes.

2. El manejo de las estructuras de datos representa un problema significativo. En

cierto modo, este aspecto es el principal responsable de la irregularidad del problema.

3. La cantidad de memoria, a menudo, es un factor limitador, con lo que se puede hacer necesario el uso de almacenamiento secundario.

4. La penalización por comunicaciones crece rápidamente si no se mantiene una buena localidad de los datos, especialmente en este tipo de problemas con acceso irregular a datos.

5. Aunque normalmente el bucle más interno es "*bien definido*", con frecuencia una fracción importante del tiempo de ejecución se pierde en otras zonas del código.

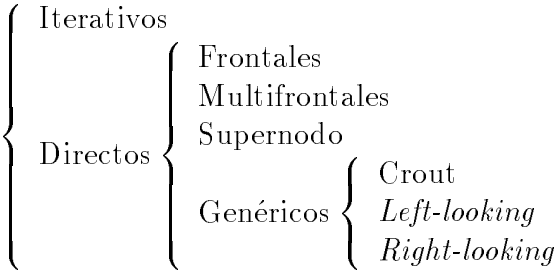
Sobre todo, los puntos 2 y 4 resaltan el contraste entre los problemas dispersos y densos del álgebra lineal, mientras que los puntos 1–3 están relacionados con el manejo de las estructuras de datos dispersas. Particularizando y profundizando un poco más en este problema de la resolución de la ecuación 1.1, dos problemas adicionales son:

- Dado que pretendemos por un lado conseguir una buena estabilidad numérica, y por otro mantener baja la densidad de la matriz durante el procesado, debemos realizar pivoteo. Es decir debemos realizar ciertas permutaciones de filas o columnas (pivoteo parcial) o de ambos tipos (pivoteo total) para obtener el vector solución de forma precisa y minimizando el tiempo de ejecución. La búsqueda de los vectores de permutación apropiados deja de ser despreciable para algoritmos dispersos y menos aún si éstos son paralelos, debido al incremento en las comunicaciones que suponen. También la implementación de las permutaciones de filas y/o columnas implica un movimiento de datos cuyo precio dependerá de la estructura de datos asociada a la matriz y añade en paralelo un consumo de tiempo en comunicaciones.

- El ineludible llenado de la matriz en el transcurso de la resolución implica una gestión de la estructura de datos para la inserción de los nuevos elementos. Este aspecto puede llegar a complicar significativamente el algoritmo tanto en secuencial como en paralelo.

1.1.2 Clasificación de estrategias

Una posible clasificación de las alternativas a la hora de resolver la ecuación 1.1, es la siguiente:



Podemos decir que existen principalmente dos estrategias para resolver sistemas de ecuaciones lineales: mediante métodos directos y por métodos iterativos. En los siguientes subapartados comentaremos brevemente en que se fundamentan cada una de las alternativas de esta clasificación.

1.1.3 Métodos iterativos

Los métodos iterativos que resuelven el sistema $Ax = b$ generan una sucesión de soluciones aproximadas x_k pertenecientes al subespacio de Krylov definido por los vectores $\{b, Ab, ..., A^{(k-1)}b\}$ tal que $||r_k|| = ||b - Ax_k||$ sea minimizado para alguna norma[76].

Los núcleos computacionales[2] de los métodos iterativos para resolver el sistema $Ax = b$ son (usando nomenclatura de las rutinas BLAS):

- SAXPY's: Suma de vectores, $y = y + \alpha x$, donde x e y son vectores y α es un número real.
- SDOT's: Producto escalar de vectores, $dot = x^T y$.
- SGEMV's: Producto matriz por vector general, $y = \alpha Ax + \beta y$, donde A es la matriz del sistema (y frecuentemente también una matriz relacionada con A, K^{-1}, el precondicionador), y β es otro real.

Normalmente, atacamos el problema, optimizando cada uno de los núcleos específicos de forma separada. Evidentemente, SAXPY's y SDOT's no necesitan demasiada atención, pero sí tiene más interés el casi siempre imprescindible producto matriz-vector involucrado en estos métodos. En [17] presentamos un algoritmo paralelo eficiente para realizar el producto de matriz dispersa por vector. Aplicando ese algoritmo han sido resueltos una gran cantidad de métodos iterativos de forma paralela [14, 126].

Los métodos iterativos son muy atractivos en numerosas situaciones para su uso en computadores vectoriales y paralelos dado la inexistencia de dependencias para el producto matriz-vector. El hecho de que no se produzca llenado en la matriz, permite manejar sistemas de mayor volumen de datos. Además la precisión de la solución puede ser especificada por el usuario estableciendo el criterio y umbral de convergencia.

Desafortunadamente, no existe un único método iterativo lo suficientemente robusto como para resolver cualquier sistema lineal disperso, con suficiente precisión y de forma dispersa. Generalmente un método iterativo es aconsejable sólo para una clase específica de problemas, ya que la velocidad de convergencia depende fuertemente de las propiedades espectrales de la matriz.

A su vez, estos métodos iterativos pueden ser clasificados [54]:

- Métodos estacionarios (Jacobi, Gauss-Seidel, Sobre-Relajación Sucesiva (SOR))

[2]Zonas de código donde se consume una fracción importante del tiempo de ejecución.

- Métodos no estacionarios (Gradiente Conjugado, Residuos Mínimos Generalizado, Gradiente Bi-Conjugado, Gradiente Conjugado Cuadrado, Gradiente Bi-Conjugado Estabilizado, Iteración de Chebyshev)

La razón de esta clasificación inicial estriba en considerar constantes ciertos parámetros del algoritmo en los métodos estacionarios, o permitir que estos parámetros varíen en cada iteración para los métodos no estacionarios.

1.1.4 Métodos directos

Los métodos directos abordan el problema de resolver la ecuación $Ax = b$ factorizando la matriz A (matriz del sistema) en un producto de otras matrices [57]. Estos factores, por su estructura, por ejemplo triangular, permiten realizar sencillas operaciones de sustitución hacia adelante (*forward substitution*) y sustitución hacia atrás (*backward substitution*).

Entre estos métodos encontramos los siguientes:

- Eliminación gausiana o factorización LU en el que la matriz A se descompone en las matrices L y U, triangular inferior y superior respectivamente.
- Factorización QR mediante la cual descomponemos la matriz del sistema en una matriz ortogonal Q (cuya inversa es sencillamente la transpuesta) y una matriz triangular superior R. Este método es de 2 a 5 veces más lento que la factorización LU y suele dar lugar a matrices bastante más llenas. Por contra, presenta una mayor estabilidad numérica, maneja apropiadamente sistemas rectangulares y entre sus aplicaciones se encuentran problemas de mínimos cuadrados[3], autovalores, etc... [52, 110, 10].
- Factorización WZ que descompone la matriz A en las matrices W y Z con un patrón especial que permite una fácil sustitución hacia adelante y atrás [20]. Este algoritmo divide por dos el número de iteraciones respecto a la LU para el caso denso, aunque cada una de estas iteraciones sea ahora ligeramente más costosa computacionalmente. Este método tiene difícil aplicación al caso disperso por la complejidad que implican las operaciones de pivoteo. De hecho, hasta donde alcanza nuestro conocimiento, sólo se ha probado la existencia de la factorización WW^T para matrices dispersas simétricas definidas positivas [26].

Nosotros nos centraremos en esta memoria en aquellos métodos basados en la eliminación Gausiana. Es decir, nuestros algoritmos realizarán la factorización LU de la matriz de coeficientes, de forma que $\Pi A \Gamma = LU$, donde Π y Γ son las matrices de permutación, y L y U son matrices triangular inferior y superior, respectivamente. Estos factores serán usados para resolver el sistema 1.1 mediante una substitución hacia adelante, $Ly = \Pi^T b$, seguida de una substitución hacia atrás, $U(\Gamma^T x) = y$. Más detalladamente, si llamamos π y γ a los vectores de permutación asociados a Π y Γ los pasos necesarios para resolver el sistema son:

[3] Aunque actualmente algunos de estos problemas pueden ser abordados con menor coste mediante la factorización LU [90].

1. Factorizar A tal que $A_{\pi_i,\gamma_j} = (LU)_{ij} \ \forall\, i,j, \ \ 1 \leq i,j \leq n$. Obtenemos las matrices L y U y los correspondientes vectores de permutaciones de filas y columnas, π y γ.

2. Permutar b según $d_i = b_{\pi_i}, \ 1 \leq i \leq n$, para obtener el vector d.

3. Resolver el sistema $Ly = d$ para conseguir el vector y. Este sistema triangular inferior es resuelto por medio de un algoritmo de substitución hacia adelante o *forward substitution.*

4. Resolver el sistema $Uz = y$, para obtener el vector z. Utilizamos aquí un algoritmo de substitución hacia atrás o *backward substitution.*

5. Permutar z según $x_{\gamma_j} = z_j, \ 1 \leq j \leq n$, resultando el vector solución x.

Dejando por el momento el problema de la elección de las matrices Π y Γ, el algoritmo general para realizar el núcleo de la factorización $A = LU$ (paso 1) es el siguiente:

Ejemplo 1.1 CÓDIGO GENERAL DE FACTORIZACIÓN LU

```
1.  for k=1 to n do
2.    factoriza A_kk = L_kk U_kk
3.    for i=k+1 to n do
4.      L_ik ← A_ik U_kk^-1
5.    for j=k+1 to n do
6.      U_kj ← L_kk^-1 A_kj
7.      for i=k+1 to n do
8.        A_ij ← A_ij − L_ik U_kj
```

Para el caso particular en que realicemos el algoritmo "*in place*" (es decir, la matriz de salida LU se calcula sobre la propia matriz de entrada A) y haciendo $L_{kk} = 1$ quedan las dos siguientes ecuaciones envueltas en el proceso de factorización dispersa LU:

$$A_{ik} = A_{ik}/A_{kk} \quad \forall \ \ i : (k < i \leq n) \tag{1.2}$$

$$A_{ij} = A_{ij} - A_{ik}A_{kj} \qquad \forall \ \ i,j : (k < i,j \leq n) \tag{1.3}$$

Estas ecuaciones se ejecutarán n veces, indexadas por la variable k. Llamaremos $A^{(k)}$ a la matriz A antes de comenzar la iteración k del algoritmo. La ecuación 1.2 implementa la división entre el elemento $A^{(k)}_{kk}$, llamado **pivot**, de todos los elementos por debajo del pivot. Por su parte, la ecuación 1.3 o ecuación de **actualización de rango uno**, recorre la submatriz inferior derecha de dimensiones $(n-k-1) \times (n-k-1)$. A esta submatriz, para la que utilizaremos también la notación $A^{(k)}(k+1:n, k+1:n)$, se le conoce por el nombre de **submatriz reducida** definida por el pivot $A^{(k)}_{kk}$. A la submatriz inferior derecha de dimensiones $(n-k) \times (n-k)$, $A^{(k)}(k:n, k:n)$, se le llama **submatriz activa** en la iteración k. A la columna k en la que se encuentra el

pivot, la llamaremos **columna del pivot** y a la parte de esa columna que pertenece a la submatriz activa excepto el pivot, es decir $A^{(k)}(k+1:n,k)$, la llamamos **columna del pivot activa**. De forma similar se define la **fila del pivot**, $A^{(k)}(k,1:n)$, y la **fila del pivot activa**, $A^{(k)}(k,k+1:n)$. En general, el uso de los términos fila (columna) activa o reducida hace referencia a la subfila (subcolumna) perteneciente a la submatriz activa o reducida respectivamente.

Si A es simétrica y definida positiva ($x^T Ax \geq 0, \quad \forall x \neq 0$) podemos aplicar la factorización de Cholesky mediante la que $A = LL^T$, rebajando significativamente la complejidad del algoritmo [83, 130]. En este caso, el algoritmo del ejemplo 1.1 se particulariza para que sólo trabaje con la submatriz triangular inferior y para que $L_{ij} = U_{ji}$, se toma $L_{kk} = U_{kk} = \sqrt{A_{kk}}$ en la línea 2 de dicho código.

Por otra parte, como vemos en la clasificación de la subsección 1.1.2, desde el punto de vista algorítmico, existen cuatro aproximaciones para abordar el problema de la eliminación gausiana dispersa. Dicho de otra forma, existes cuatro tipos de algoritmos para aplicar las ecuaciones 1.2 y 1.3, a saber: técnicas generales, métodos frontales, aproximación multifrontal y técnicas supernodo. Las técnicas frontales son una extensión de los esquemas para matrices en banda o banda variable, y dan buenos resultados con sistemas de poco ancho de banda. Los métodos multifrontales son una extensión de los anteriores, que consigue tiempos de ejecución reducidos, en matrices con un patrón de elementos no nulos simétrico o cercano al simétrico. Tanto en estas dos últimas alternativas como en la técnica supernodo, las operaciones aritméticas se aplican sobre submatrices densas. De esa forma podemos apoyarnos en las eficientes rutinas BLAS de los niveles 2 o 3 (operaciones matriz-vector y matriz-matriz)[55]. Aunque inicialmente estos métodos tenían su principal aplicación para matrices simétricas, de patrón simétrico o cercano al simétrico, también se han realizado suficientes avances para conseguir códigos de estas características que factoricen matrices no simétricas (UMFPACK [49] y SuperLU [51]). Las tres siguientes subsecciones dedican algunos párrafos para describir en mayor detalle cada una de las aproximaciones mencionadas.

1.1.4.1 Métodos Multifrontales

Mediante una estrategia frontal podemos factorizar una matriz en banda, de semiancho de banda b, usando una *matriz frontal* de $b \times b$ (o $b \times (2b-1)$ si permitimos pivoteo de filas en la matriz frontal), que va recorriendo la banda actuando a modo de "ventana". En cada iteración sólo hay que realizar una factorización densa de la *matriz frontal*. Este método tiene su razón histórica en los códigos *out-of-core*[4], ya que sólo se necesita memoria para la matriz frontal. El método también puede ser extendido a matrices de banda variable, pero para una matriz genérica la matriz frontal puede resultar demasiado grande. De hecho, si el ancho de banda no es suficientemente pequeño, los métodos frontales requieren más memoria y más operaciones en punto flotante que una técnica general de factorización [56].

La aproximación multifrontal es un intento de retener algunas de las ventajas de los métodos frontales, economizando al mismo tiempo en operaciones en punto flotante. Estos métodos, descritos con mayor detalle en [54, 57], tratan la factorización de una

[4]Códigos que necesitan más memoria física de la que realmente tienen disponible.

matriz dispersa como una jerarquía de subproblemas de factorización densa. El árbol de eliminación asociado a una matriz es el árbol usado para determinar el orden de precedencia durante la factorización.

Por ejemplo, en la figura 1.1 (a), presentamos el patrón de una matriz simétrica de 5×5 donde el símbolo '$\times$' representa a los elementos no nulos iniciales y el símbolo '$\bigcirc$' representa los nuevos elementos generados durante la factorización (*fill-in*).

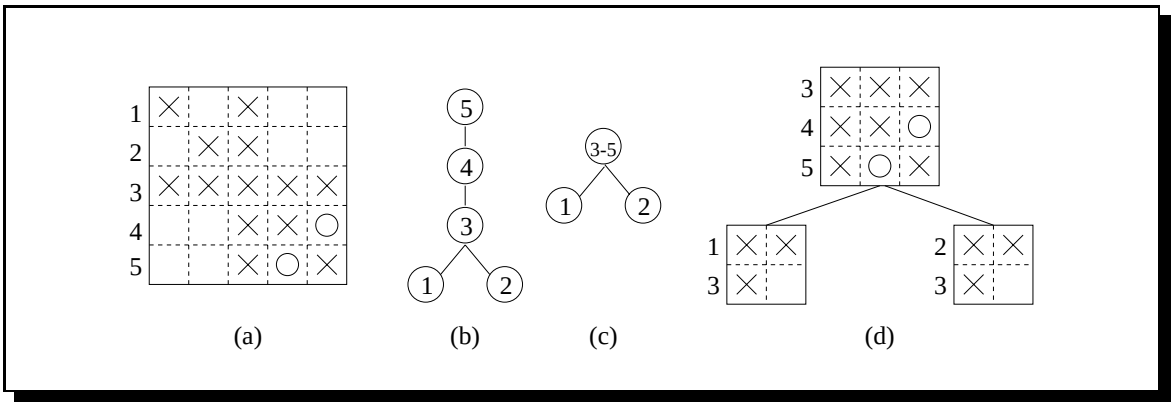


Figura 1.1: Factorización multifrontal.

Durante la factorización, como podemos deducir de las ecuaciones 1.2 y 1.3 para este patrón dado, la primera y segunda columna son independientes. Sin embargo estas dos primeras columnas deben estar factorizadas para poder proseguir con la factorización de la columna 3, luego la 4 y por último la 5. Eso queda representado mediante el árbol de eliminación de la figura 1.1 (b). Los nodos de este árbol que tienen un único hijo pueden ser colapsados (o amalgamados), como se muestra en la figura 1.1 (c), ahorrando operaciones en el algoritmo multifrontal. Cada nodo de este nuevo árbol de eliminación está asociado a una matriz frontal, como vemos en la figura 1.1 (d). Por ejemplo, el nodo uno está asociado a la matriz frontal resultado de "ensamblar" los elementos no nulos de la columna uno. De igual forma para el nodo dos. Estos dos nodos pueden ser factorizados independientemente, pero los dos dan lugar a una contribución al elemento $A_{3,3}$, que debe ser sumada en el nodo padre antes de que éste se pueda factorizar. A la operación de combinar los resultados parciales de los hijos en el padre se le llama *suma extendida*.

Si la matriz no es simétrica puede ser considerada como tal, almacenando explícitamente los ceros necesarios, es decir, considerando el patrón de $A + A^T$, a costa del evidente consumo de memoria necesario si la matriz no tiene la suficiente simetría. Como podemos apreciar del ejemplo anterior, la elección del árbol de eliminación en función de la reordenación que se haga de la matriz puede ser decisiva en su rápida factorización. Por ejemplo en la figura 1.2 (a) hemos permutado la columna 3 con la 5, y la fila 3 con la 5, de forma que ahora el árbol de eliminación presenta más concurrencia, como vemos en 1.2 (b). Además ahora no hay llenado. Por tanto, previa a la etapa de factorización es necesaria una etapa de reordenación y análisis simbólico.

Sin embargo los inconvenientes de estos métodos son:

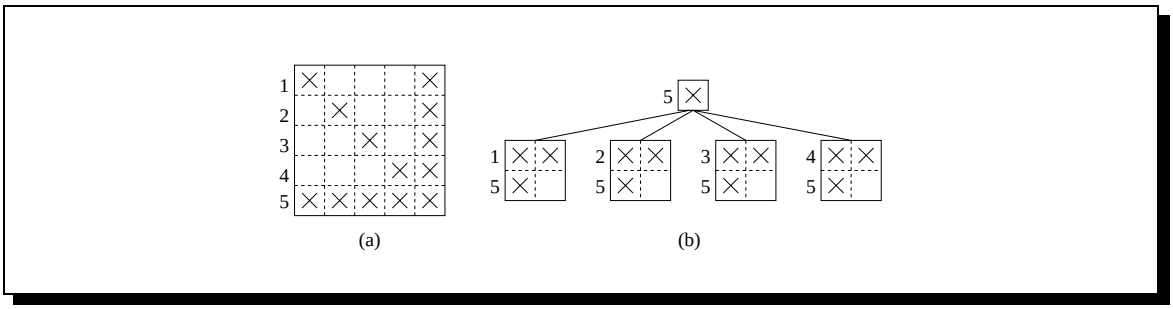


Figura 1.2: Reordenación previa y sus ventajas.

- Suele existir un coste sustancial asociado a movimientos de datos.
- Se asume una simetría estructural
- Se supone que cualquier elemento de la diagonal puede ser pivot.

Aunque Davis y Duff extendieron los métodos multifrontales para matrices genéricas no simétricas en su algoritmo UMFPACK [49], usando matrices frontales rectangulares y grafos acíclicos dirigidos [73], esta extensión es más eficiente que la estándar sólo si la matriz es altamente asimétrica.

1.1.4.2 Métodos Supernodo

Otra posibilidad para convertir el problema de factorización dispersa en un conjunto de subproblemas densos, nace de considerar conjuntamente aquellas columnas con patrón común. Factorizar usando bloques de columnas con patrón común, o *supernodos*, permite disminuir el número de iteraciones del algoritmo y sacar un mayor provecho de la cache. Dentro de esta clase de algoritmos encontramos los llamados columna-supernodo, supernodo-columna y supernodo-supernodo. En principio aplicados a matrices simétricas, fueron extendidos para matrices asimétricas por Demmel y col. [51] con su algoritmo SuperLU, que utiliza un árbol de eliminación de columnas (versión asimétrica del árbol de eliminación). Aunque no alcanzan prestaciones semejantes a las que proporcionan los métodos supernodo para matrices simétricas, para un gran número de matrices (sobre todo si son poco dispersas o ganan densidad durante la factorización) el método es más rápido que el UMFPACK de Davis y Duff.

Como vemos, intentar explotar la densidad de una matriz dispersa tiene sus ventajas, pero cuando la matriz de entrada, A es suficientemente dispersa y sus factores, L y U, siguen siéndolo no presenta reducciones en el tiempo de computación sino más bien consumo en rutinas de reordenación, generación del árbol de eliminación, etc.

1.1.4.3 Técnicas Generales

Las principales características de estas técnicas son:

- La búsqueda del pivot se realiza tanto para mantener el coeficiente de dispersión de la matriz como para garantizar la estabilidad numérica.
- No imponen condiciones a la matriz del sistema.
- La estructura de datos dispersa se usa en todo el código, incluso en el bucle más interno.

Sobre todo, las dos últimas características, nos inclinan a centrarnos en estos métodos. En primer lugar porque son válidas para cualquier matriz dispersa independientemente del patrón, simetría, densidad y demás consideraciones. Pero también porque, en cierto sentido, son una generalización de los algoritmos densos aunque recorran una estructura de datos dispersa. El hecho de que el núcleo computacional esté delimitado únicamente por tres bucles, permite aplicar paralelismo a este nivel, de forma que el problema también puede ser abordado desde el punto de vista de los compiladores semi-automáticos (o de paralelismo de datos) y automáticos.

Por otro lado, los métodos multifrontales y supernodo basan su eficacia en convertir el problema disperso en un conjunto de subproblemas densos. Este aspecto es en ocasiones difícil de explotar cuando la matriz es suficientemente dispersa. Y por otro lado, nos parece un desafío más provocador el de aceptar el problema disperso como tal, evaluar el comportamiento de las estructuras de datos necesarias y sacar conclusiones de todo ello, seguramente extrapolables a otros problemas dispersos o irregulares.

Algoritmos secuenciales estándar basados en técnicas generales son por ejemplo la subrutina MA48 [61] o la Y12M [163]. En la sección 1.3 presentaremos detalladamente la organización y parámetros de este tipo de algoritmos numéricos. Previamente, es necesario introducir los conceptos básicos relacionados con las matrices dispersas, aspecto que abordamos en la siguiente sección.

1.2 Matrices dispersas

En esta sección enumeramos las características relevantes de las matrices dispersas y sus posibles representaciones en la memoria del computador. Decimos que una matriz es dispersa si resulta ventajoso explotar sus ceros. Es decir, cuando tenemos en cuenta únicamente los elementos no nulos de la matriz tanto desde el punto de vista del almacenamiento como del procesamiento.

Así como la representación de una matriz densa en memoria no es más que el encadenamiento de sus coeficientes ya sea por filas o por columnas, para el caso disperso, el no considerar los elementos nulos, proporciona un grado de libertad mucho mayor a la hora de elegir la estructura de datos. De igual forma, si ya en códigos densos es importante tener en cuenta si la matriz se ordena por filas o por columnas para computadores con jerarquía de memoria, mucho más determinante es, para un código disperso, la elección de la estructura de datos en su posterior comportamiento. Introducimos primero las características de las matrices dispersas para luego tratar las estructuras de datos apropiadas para su almacenamiento.

1.2.1 Características

Una matriz dispersa, A, de dimensiones $n \times n$ con α elementos distintos de cero, con $\alpha \ll n^2$, decimos que tiene **densidad** $\rho = \alpha/n^2$. Normalmente la densidad de una matriz dispersa es inferior al 1%.

Se llama **entrada** (o *entry* en inglés) a cada coeficiente no nulo de la matriz. El **patrón** de una matriz dispersa, o matriz simbólica, es el conjunto de índices (i,j) tal que $A_{ij} \neq 0$. En función de este patrón tendremos matrices dispersas:

- Diagonal: $A_{ij} \neq 0; i = j$.
- Banda: $A_{ij} = 0$, $(i+p \leq j) \wedge (j+q \leq i)$. Ancho de banda$= p+q-1$; $1 < p, q < n$.
- Tridiagonal: Matriz banda con $p = q = 2$
- Banda variable o *Skyline*: Matriz banda, de ancho de banda variable.
- Tridiagonal por bloques.
- Triangular o triangular por bloques.
- Genérica, cuando el patrón no es ninguno de los anteriores.

Normalmente se llama dispersa a la matriz que tiene un patrón aleatorio, y, sin perdida de generalidad, nos centraremos en este tipo.

Si consideramos los coeficientes de la matriz también hay que contemplar el **tipo** de la matriz: real o compleja, ya sea en simple o doble precisión. Por otro lado, las matrices dispersas también se caracterizan por sus parámetros de **simetría**:

- Simétrica: $A_{ij} = A_{ji}$.
- Simétrica definida positiva: $x^T Ax \geq 0$, $\forall\, x \neq 0$.
- Hermítica.
- Patrón simétrico: $A_{ij} \neq 0$ si $A_{ji} \neq 0$.
- Asimétrica.

Dado que las matrices no simétricas representan el caso general y también el de más difícil solución en sistemas lineales, serán las consideradas en esta tesis.

1.2.2 Estructuras de almacenamiento

Como hemos dicho anteriormente, la estructura de datos elegida para representar la matriz es muy relevante en los algoritmos dispersos.

Distinguiremos principalmente entre estructuras de datos estáticas y dinámicas. En las primeras, la gestión de memoria es estática y por tanto el espacio de almacenamiento

es constante y definido al comienzo del algoritmo. Por contra, las estructuras de datos dinámicas permiten reservar y liberar memoria en tiempo de ejecución en función de las necesidades de almacenamiento. La siguiente matriz dispersa de 5×5 con 8 entradas nos servirá de ejemplo para visualizar los distintos formatos de almacenamiento.

$$\begin{pmatrix} \mathbf{a} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{b} & 0 & \mathbf{c} & 0 \\ \mathbf{d} & 0 & \mathbf{e} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \mathbf{f} & 0 & \mathbf{g} & \mathbf{h} & 0 \end{pmatrix} \tag{1.4}$$

Las características que nos interesa considerar de la estructura de datos elegida serán:

- **Consumo de memoria**. Es un parámetro que queremos minimizar, pero que entra en compromiso con las demás características.
- **Flexibilidad de acceso**. Es claro que el cálculo de la dirección de memoria donde se almacena un coeficiente de una matriz densa es inmediato. Sin embargo, los almacenamientos comprimidos dan lugar a mayores tiempos de acceso a un elemento cualquiera de la matriz.
- **Fragmentación de memoria**. Cuando se crean nuevos elementos (llenado), o es necesario cambiarlos de posición (pivoteo), puede incrementarse la fragmentación de memoria. Ésto reduce significativamente la velocidad del código en máquinas con jerarquía de memoria.
- **Versatilidad** en la generación de nuevas entradas y movimiento de datos. Poder incluir cómodamente una nueva entrada en la estructura de datos aliviará el coste asociado al llenado. Si es necesario permutar filas (o columnas), mover entradas implica la necesidad de borrarlas de una fila para insertarlas en otra. Por tanto también debemos facilitar la operación de borrado.

Aparte de las estructuras que presentamos a continuación, existen otras, discutidas en [57, 24, 112, 121, 133, 161, 103, 158, 42], y que no contemplamos aquí por consistir en modificaciones particulares para otros problemas o patrones específicos (formatos como *Jagged Diagonal*, *Skyline*, *Banded Linpack*, *Quadtree*, etc), que no son relevantes para el propósito de esta tesis.

1.2.2.1 Estructuras estáticas

Quizás la estrategia más natural para representar de forma compacta una matriz dispersa sea indicando para cada entrada, tanto su índice de fila, `i`, como de columna, `j`. A este almacenamiento se le llama de **coordenadas** por razones obvias. Por tanto, necesitaremos tres vectores de longitud α: uno de igual tipo que la matriz para almacenar los coeficientes, `Val`, y dos enteros que almacenan sendas coordenadas, `Fil` y `Col`. Para el caso de la matriz 1.4, ordenando los coeficientes por filas (a), o por columnas (b), quedaría:

Val	a	b	c	d	e	f	g	h
Fil	1	2	2	3	3	5	5	5
Col	1	2	4	1	3	1	3	4

(a)

Val	a	d	f	b	e	g	c	h
Fil	1	3	5	2	3	5	2	5
Col	1	1	1	2	3	3	4	4

(b)

Sin embargo, no es necesario tener los coeficientes ordenados, aunque en ese caso el acceso se complica. Los dos principales inconvenientes de esta estrategia son el alto consumo de memoria (por cada entrada se almacenan tres valores), y la necesidad de acceder a las entradas mediante indirecciones.

Otra estructura de datos es la conocida con el nombre de *Compressed Row Storage* (**CRS**) o almacenamiento comprimido por filas. Ahora, en vez de un vector con índices de columna, almacenaremos un vector con punteros al principio de cada fila, `Filpt`. Este vector será de $n+1$ elementos con el consiguiente ahorro de memoria, donde por definición, `Filpt[n+1]`$=\alpha+1$. En este caso la representación sería:

Val	a	b	c	d	e	f	g	h
Col	1	2	4	1	3	1	3	4
Filpt	1	2	4	6	6	9		

Claramente las filas deben estar ordenadas aunque dentro de cada fila no es obligatorio ningún orden especial. Evidentemente, también podemos emplear un almacenamiento por columnas, de forma que obtendríamos la estructura *Compressed Column Storage*, abreviadamente, **CCS**. En este caso, tendríamos un vector de índices de fila, `Fil`, y otro de punteros a columnas, `Colpt`. Estos dos últimos almacenamientos comprimidos ahorran espacio pero también necesitan indirecciones a través del vector de punteros para acceder a una fila (columna) dada.

Como se ha podido apreciar estas estructuras de datos basadas en vectores son totalmente compatibles con el Fortran 77 y su gestión estática de memoria. El consumo de memoria es casi óptimo y es posible diseñar códigos que no generen fragmentación de memoria. Sin embargo el acceso está restringido a una de las dos direcciones: filas o columnas. Por ejemplo, en una CCS acceder a una columna `j` sólo implica recorrer `Val[Colpt(j):Colpt(j+1)-1]`. Por contra, recorrer la fila `i` supone barrer todas las columnas de la matriz, chequeando si hay o no elemento en la fila `i`. Por otro lado, la versatilidad que ofrecen estas estructuras para aceptar nuevas entradas o moverlas deja mucho que desear. Por tanto, estas estructuras comprimidas son especialmente útiles en problemas que no presentan llenado (como los métodos iterativos para resolver sistemas de ecuaciones) o cuando se ha cuantificado de antemano el llenado y donde tendrá lugar (como en la factorización de Cholesky cuando se antecede de una etapa de análisis). En caso contrario, las estructuras comprimidas, como veremos en el capítulo siguiente, son un inconveniente a la hora de implementar operaciones de pivoteo y llenado, necesarias en la factorización dispersa LU.

1.2.2.2 Estructuras dinámicas

Los inconvenientes discutidos en la sección anterior nos motivan a estudiar otras alternativas, ahora dinámicas, para la representación de la matriz dispersa en memoria.

Por lo pronto, utilizar estructuras dinámicas con su gestión de memoria asociada, nos obliga a cambiar el lenguaje de programación de Fortran 77 a por ejemplo C o Fortran 90. Éstos, nos permiten declarar tipos de datos definidos por el usuario que se ajusten a la definición de entrada: la terna (valor, índice de fila, índice de columna); utilizando un campo de la estructura para cada elemento. Para almacenar todas las entradas en memoria podríamos ponerlas una detrás de otra, pero eso no sería más que una variación de la estructura por coordenadas. Más bien, podemos añadir cuatro campos adicionales a la estructura `entrada` con punteros a las entradas anterior y posterior en la misma fila y columna. Es decir, enlazaremos las entradas por filas y columnas, y utilizaremos dos vectores de punteros a entradas, `fils` y `cols`, para indicar el comienzo de cada columna y cada fila.

A esta estructura de datos se la conoce con el nombre de **lista bidimensional doblemente enlazada**, y la llamaremos abreviadamente con las siglas **LLRCS** de *Linked List Row-Column Storage*. La definición de las estructuras necesarias se muestran en el ejemplo 1.3, apoyándonos en la sintaxis del Fortran 90 y del C, y suponiendo la matriz dispersa de coeficientes reales. La representación de la matriz 1.4 bajo esta perspectiva se muestra en la figura 1.3.

Ejemplo 1.3 DECLARACIÓN DE LA ESTRUCTURA LLRCS

FORTRAN90

```
TYPE entrada
   INTEGER:: Fil,Col
   REAL:: Val
   TYPE (entrada), POINTER:: previ, &
                   prevj, nexti, nextj
END TYPE entrada
TYPE ptr
   TYPE (entrada), POINTER:: p
END TYPE ptr
TYPE (ptr), DIMENSION(n):: fils, cols
```

C

```
struct entrada {
  int Fil,Col;
  double Val;
  struct entrada *previ,*prevj,
                 *nexti,*nextj;
};
struct matriz
{
  struct entrada *fils[n], *cols[n];
};
```

En este caso hemos enlazado las entradas por filas y columnas siguiendo un orden creciente de los índices, dando lugar a una estructura ordenada, pero también podríamos optar por una estrategia desordenada.

Es fácil percatarse de las **ventajas** de esta completa estructura de almacenamiento. Insertar una nueva entrada consistirá en reservar el espacio que necesita[5], inicializarla con los valores correspondientes, y finalmente, insertarla actualizando adecuadamente los punteros de las entradas vecinas. Borrar una entrada también es trivial gracias a que la lista está doblemente enlazada. Por último, y quizás más importante, el acceso

[5]Mediante la función ALLOCATE() del F90 o malloc() del C, por ejemplo.

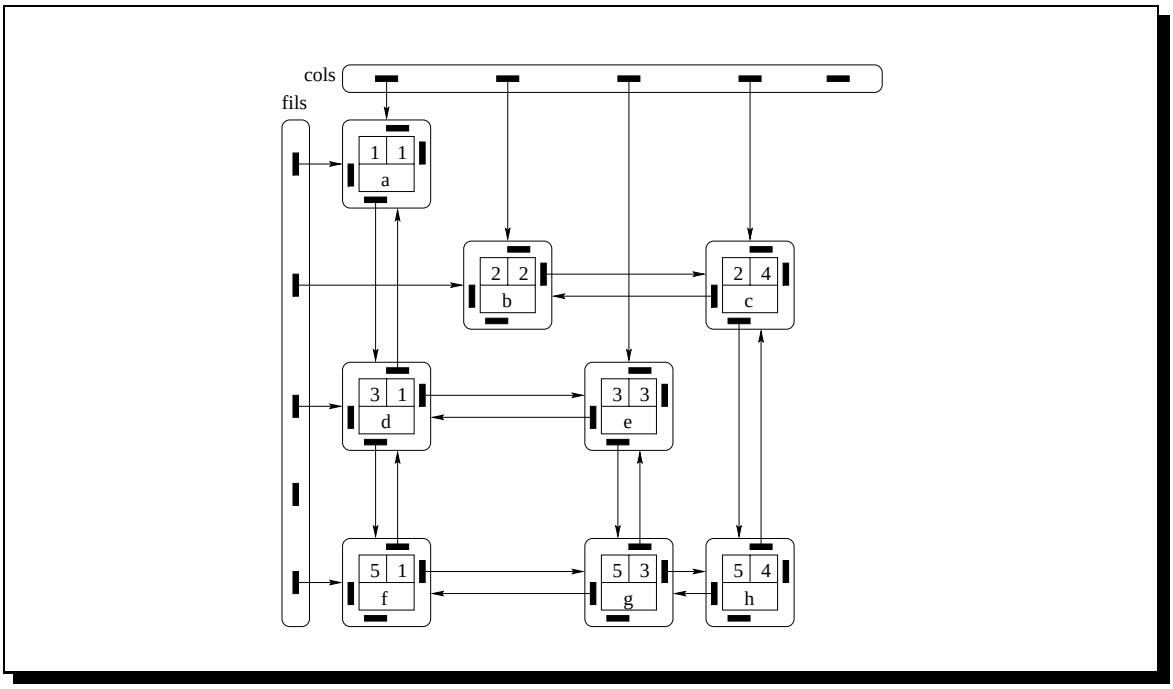


Figura 1.3: Estructura de datos de lista bidimensional doblemente enlazada.

es posible tanto por filas como por columnas simultáneamente.

Pero a cambio, los **inconvenientes** tampoco son fáciles de obviar. A saber, el consumo de memoria es significativo, debido a que por cada entrada, no sólo guardamos el valor de los coeficientes como ocurre en una matriz densa, sino también los índices de fila y columna, y más aún, los cuatro punteros de enlace. El reservar y liberar entradas independientes provocará fragmentación de memoria y que entradas vecinas estén físicamente en posiciones lejanas de memoria. Ésto redunda en un bajo aprovechamiento de la cache. Además, las operaciones de reserva y liberación de memoria junto con la necesidad de recorrer las listas a través de los punteros suponen un coste adicional.

Existe un compromiso entre el consumo de memoria y la flexibilidad en el acceso, al igual que entre la versatilidad y la fragmentación. Por ejemplo si queremos reducir el consumo de memoria, podemos enlazar las entradas sólo en un sentido y no en los dos. Nos ahorramos así dos punteros por entrada, pero ahora sólo podemos recorrer las listas en un sentido y borrar una entrada será más caro. Tendríamos una lista bidimensional simplemente enlazada.

También podemos utilizar listas unidimensionales simple o doblemente enlazadas. Utilizaremos las siglas **LLRS** (de *Linked List Row Storage*) si enlazamos por filas y **LLCS** (de *Linked List Column Storage*) si enlazamos por columnas. De esta forma, reducimos el acceso a filas o columnas y eliminamos punteros junto con alguno de los índices. La estructura de datos para el caso de enlace doble por columnas se muestra en el ejemplo 1.4 y gráficamente en la figura 1.4. Aunque hemos perdido un grado de accesibilidad, aún seguimos teniendo versatilidad para añadir y borrar entradas.

Ejemplo 1.4 DECLARACIÓN DE LA ESTRUCTURA LLCS

FORTRAN90

```
TYPE entrada
  INTEGER:: Fil
  REAL:: Val
  TYPE (entrada), POINTER:: previ, nexti
END TYPE entrada
TYPE (ptr), DIMENSION(n):: cols
```

C

```
struct entrada {
  int Fil;
  double Val;
  struct entrada *previ, *nexti; };
struct matriz {
  struct entrada *cols[n]; };
```

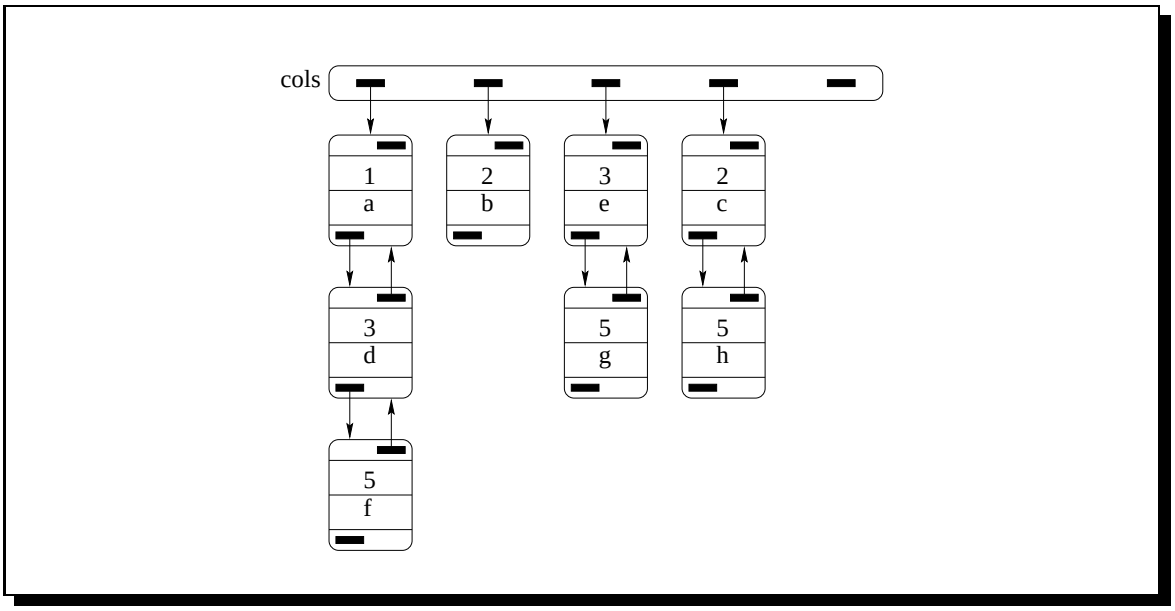


Figura 1.4: Lista unidimensional doblemente enlazada por columnas.

Sin embargo, aún podemos reducir más el consumo de memoria asociado a los punteros, sencillamente eliminándolos. La idea está en considerar cada columna como un **vector empaquetado**. Un vector disperso puede ser comprimido mediante tres campos: el tamaño, `size`; un vector con los coeficientes, `Val`; y un vector con los índices, `Ind`. A esta estructura, cuya declaración mostramos en el ejemplo 1.5, la llamaremos **CVS** (de *Compressed Vector Storage*).

Ejemplo 1.5 DECLARACIÓN DE LA ESTRUCTURA CVS

FORTRAN90

```
TYPE cvs
  INTEGER:: size
  INTEGER, DIMENSION(:), POINTER:: Ind
  REAL, DIMENSION(:), POINTER:: Val
END TYPE pv
```

C

```
struct cvs {
  int size;
  int *Ind;
  real *Val;
}
```

Una vez declarado el vector comprimido, podemos agrupar `n` de éstos, para representar la matriz dispersa. Si elegimos un almacenamiento por filas, el índice, `Ind`, de los vectores será de columnas, y en caso contrario, de filas. Para acceder a cada vector tenemos varias opciones. Una aproximación basada en punteros (más acorde con el lenguaje C), utiliza un array de punteros para direccionar cada vector. Si queremos darle una visión Fortran a la misma representación, podemos declarar la matriz como un array de `n` estructuras tipo CVS. Las dos alternativas quedan representadas gráficamente en la figura 1.5 (a) y (b) respectivamente, donde también se especifica la declaración para la estructura de datos que representa a la matriz dispersa.

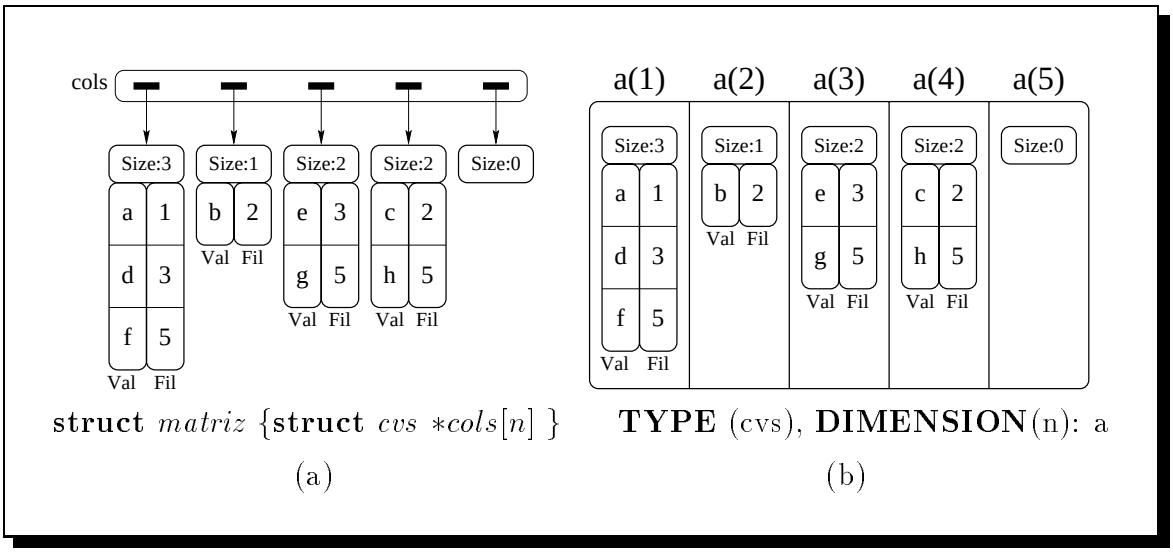


Figura 1.5: Matriz dispersa como conjunto de columnas CVS.

El problema de insertar una nueva entrada se encarece debido a que es necesario redimensionar el vector empaquetado y mover algunas entradas. Sin embargo en el proceso de factorización ésto no tiene porque suponer demasiado inconveniente ya que la operación de actualización puede incluir, sin un excesivo coste adicional, el mencionado movimiento de datos.

Esta estructura representa un compromiso entre las alternativas de almacenamiento estáticas y dinámicas. La inserción de una nueva entrada es más costosa que si tenemos listas enlazadas, pero mucho más económica que con un almacenamiento CCS, el cual suele implicar el movimiento de datos en gran parte de la matriz. La fragmentación de memoria en este caso también sera intermedia entre las listas y los almacenamientos comprimidos.La tabla 1.1 resume una comparativa entre las características de las distintas alternativas presentadas.

Queremos hacer notar que si la matriz no es suficientemente dispersa, representar la matriz dispersa vía estas estructuras puede requerir incluso más memoria que si utilizásemos una almacenamiento denso (almacenando los ceros). Una comparación entre las estructuras dinámicas de listas enlazadas bidimensionalmente (simple o doblemente enlazadas, ordenadas y desordenadas), en cuanto a consumo de memoria, y también complejidad computacional para un algoritmo particular de factorización LU dispersa, se presenta en [154].

Estructura	Consumo	Flexibilidad	Fragmentación	Versatilidad
Lista enlazada 2D	Alto	Alta	Alta	Alta
Lista enlazada 1D	Medio	Media	Alta	Alta
Vectores empaquetados	Bajo	Media	Media	Media
CCS/CRS	Bajo	Baja	Baja	Baja

Tabla 1.1: Comparación entre las alternativas de almacenamiento.

1.2.3 Conjuntos de matrices de prueba

En esta subsección describimos las matrices dispersas que hemos utilizado para validar las distintas implementaciones de la factorización LU. Existe gran cantidad de conjuntos de matrices dispersas de dominio publico, de forma que se facilita el comparar resultados con otros grupos de investigación. Las matrices dispersas del conjunto conocido como *Harwell-Boeing Sparse Matrix Collection* han sido extensivamente usadas como matrices de prueba. El formato de almacenamiento de este conjunto es CCS junto con cuatro líneas iniciales con información adicional [58].

Últimamente los distintos conjuntos de matrices se han ido agrupando en dos direcciones de Internet:

- MatrixMarket [34, 120] con las colecciones completas de Harwell-Boeing, SPARSKIT de Yousef Saad [131] y NEP (*Nonsymmetric Eigenvalue Problem*). Además contiene motores de búsqueda para seleccionar matrices que se adecúen a ciertos requerimientos, y cada matriz tiene su propia "*home page*" con detalles del patrón y propiedades de la matriz. Actualmente mantiene 482 matrices y 25 generadores aleatorios de matrices dispersas. Las matrices se almacena en formato CCS y/o mayoritariamente por coordenadas.

- Colección de la Univ. de Florida [50] más completa en número de matrices (665 en Julio del 97) y en información gráfica sobre cada una de ellas, aunque menos potente en utilidades de búsqueda y generación. Esta colección también incluye las Harwell-Boeing, pero además contiene conjuntos de matrices más grandes y algunas muy mal condicionadas.

Nosotros hemos seleccionado un conjunto de 17 matrices resultantes del modelado en diversos campos de la ciencia, con distintas dimensiones y densidad. Todas ellas coinciden en ser asimétricas ya que para matrices simétricas la factorización LDL^T o Cholesky proporcionan evidentemente menores tiempos de factorización. En la tabla 1.2 se resumen las propiedades de cada una de ellas, mientras en la figura 1.6 se muestra la distribución espacial de sus entradas o el patrón para algunas de ellas.

1.3 Técnicas generales de factorización secuencial

Gracias a la relevancia e interés que despierta la resolución de sistemas de ecuaciones dispersas, no es extraño que esta línea sea particularmente activa y generosa en ideas,

Matriz	Origen	n	Nº de entradas	Densidad (ρ)
STEAM2	Modelado de reservas de petróleo	600	13760	3.82%
JPWH991	Modelado de circuitos físicos	991	6027	0.61%
SHERMAN1	Modelado de reservas de petróleo	1000	3750	0.37%
SHERMAN2	Modelado de reservas de petróleo	1080	23094	1.98%
EX10	Flujo 2D en un canal multietapa	2410	54840	0.94%
ORANI678	Modelado de economía	2529	90158	1.41%
EX10HS	Flujo 2D en un canal multietapa	2548	57308	0.88%
CAVITY10	Elementos finitos	2597	76367	1.13%
WANG1	Simulación de semiconductores	2903	19093	0.22%
WANG2	Simulación de semiconductores	2903	19093	0.22%
UTM3060	Colección SPARSKIT	3060	42211	0.45%
GARON1	Navier-Stokes CFD	3175	88927	0.88%
EX14	Flujo de filtración isotérmico 2D	3251	66775	0.63%
SHERMAN5	Modelado de reservas de petróleo	3312	20793	0.19%
LNS3937	Flujo de fluidos comprimibles	3937	25407	0.16%
LHR04C	Ingeniería Química	4101	82682	0.49%
CAVITY16	Elementos finitos	4562	138187	0.66%

Tabla 1.2: Matrices de prueba.

trabajos y aportaciones frecuentes. En este continuo intento de mejorar el estado del arte, han surgido distintas aproximaciones para la resolución del mismo problema, como los comentados métodos frontales, multifrontales, supernodo y generales.

Nuestra decisión de abordar los métodos generales, descartando los demás, se basa en tres motivos ya mencionados:

- Vamos a afrontar el problema disperso sin intentar evitarlo mediante su representación mediante un conjunto de problemas densos, esperando extraer técnicas aplicables a otros tipos de problemas irregulares.

- La organización de estos algoritmos en base a bucles anidados más o menos bien definidos, permitirá enfocar el problema desde el punto de vista de la compilación automática o semi-automática.

- Las técnicas generales no imponen condiciones a las características de las matrices dispersas de entrada.

Sin embargo, incluso cuando nos centramos en los métodos generales, encontramos una gran diversidad de alternativas. Éstas serán discutidas y, en cierto modo, clasificadas, en el primer apartado de esta sección. A continuación se presentan las consideraciones particulares al trabajar con matrices dispersas, haciendo hincapié en el flujo de datos de los algoritmos. Creciendo en complejidad, el objetivo de la subsección 1.3.3 será introducir los problemas de estabilidad numérica y preservación del coeficiente de dispersión. Estos dos aspectos en mutuo compromiso tendrán su especial tratamiento. Con todo ello podemos esbozar, en la última subsección, las distintas fases en que podemos dividir el problema de la resolución de sistemas de ecuaciones dispersos.

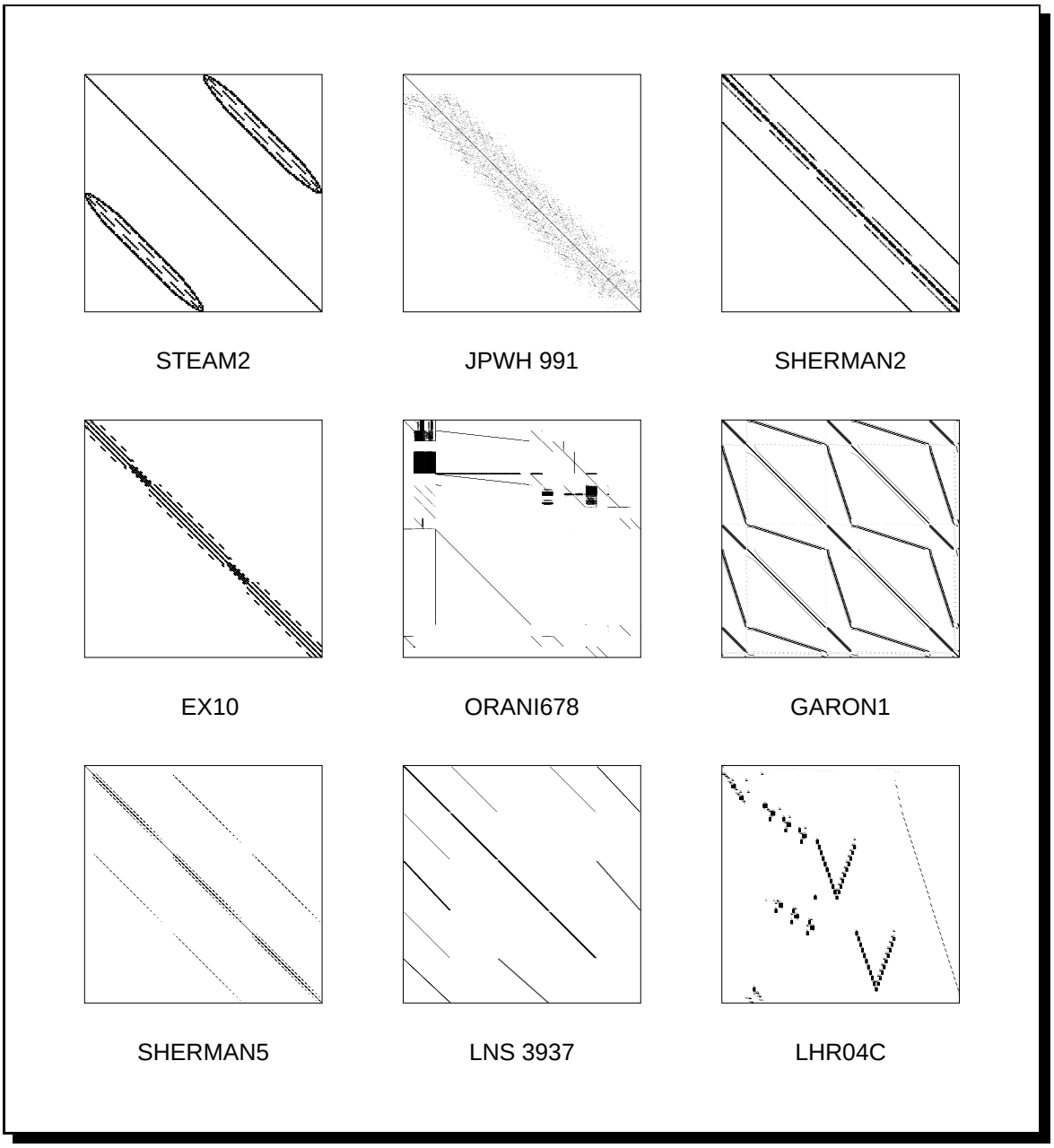


Figura 1.6: Patrones de algunas matrices de prueba.

1.3.1 Alternativas en los métodos generales

Dejando para la subsección 1.3.3 el problema de preservar el coeficiente de dispersión de la matriz y asegurar la estabilidad numérica, podemos decir que el núcleo computacional de estos métodos genéricos está centrado en tres bucles anidados: el de índice k que recorre las n iteraciones del algoritmo para cada uno de los n coeficientes de la diagonal (a partir de ahora, **pivots**); el de índice i que recorre las filas de la submatriz reducida, definida para cada iteración k por la submatriz $A(k+1:n, k+1:n)$; y el de índice j que recorre las columnas de esa misma submatriz reducida.

En función del orden en que anidemos los bucles obtendremos tres variantes: *right-*

looking (*submatrix-oriented* o *fan-out*), *left-looking* (*column-oriented* o *fan-in*) y Crout. Los términos *right* y *left* hacen referencia a las regiones de datos accedidas y el método Crout es un híbrido de estas dos primeras versiones.

En la figura 1.7 podemos ver los patrones de acceso a la matriz para cada una de las variantes. En el caso general estos algoritmos pueden ser aplicados por bloques. De esta forma, la zona sombreada oscura representa el bloque de columnas o filas que está siendo computado para pasar a formar parte de las matrices L y U, para una iteración k dada. Las zonas con sombreado claro delimitan los elementos de la matriz que serán accedidos en dicha iteración. Las flechas indican hacia donde evoluciona el patrón de acceso a cada iteración. Como se puede apreciar en la figura, el método de Crout queda descartado por la mayor irregularidad en el acceso a los datos, generando por tanto un menor aprovechamiento de la cache.

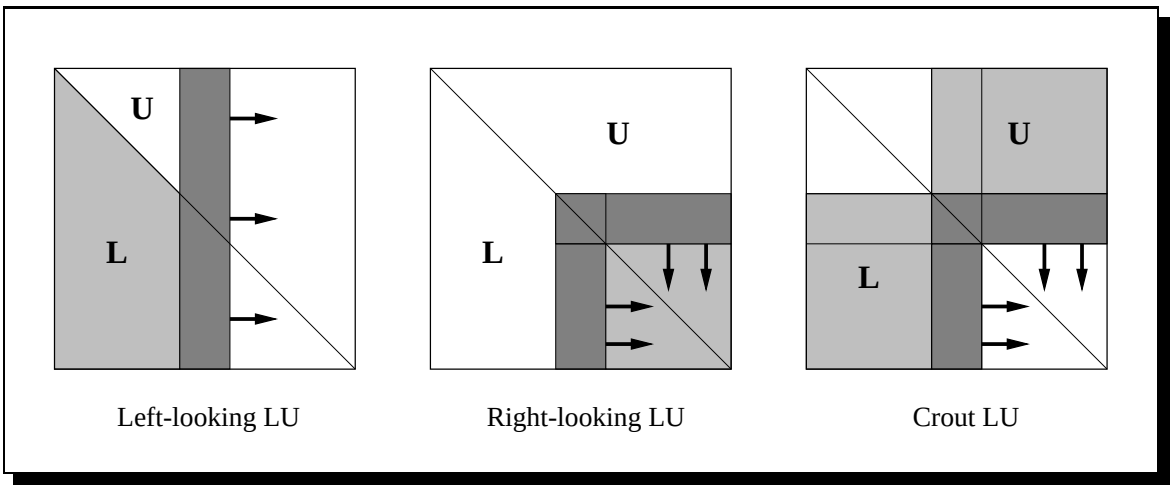


Figura 1.7: Patrones de acceso a datos para las variantes de la factorización LU.

A estas tres posibles alternativas se les conoce con el nombre de variantes ijk debido que dependen del orden en que anidemos los bucles correspondientes [109]. Por ejemplo, el pseudocódigo asociado a la factorización *right-looking* puede ser expresado como se indica en el ejemplo 1.6.

Ejemplo 1.6 PSEUDOCÓDIGO RIGHT-LOOKING LU

```
1.  for k=1 to n do
2.      for i=k+1 to n do
3.          A_ik ← A_ik / A_kk
4.      for j=k+1 to n do
5.          for i=k+1 to n do
6.              A_ij ← A_ij − A_ik A_kj
```

Al concluir el algoritmo, la parte triangular estrictamente inferior de la matriz A contendrá los coeficientes de la matriz L (de diagonal unidad), y la parte triangular superior de A contendrá los coeficientes de la matriz U (con la misma diagonal que A).

Siguiendo una organización "por columnas" (por ejemplo, la que proporciona Fortran) esta computación puede ser expresada en términos de columnas. Las líneas 2 y 3, encargadas de realizar la ecuación 1.2, pueden ser expresadas mediante una operación de división de columna, `divcol(k)`. Análogamente las líneas 5 y 6 representan la actualización de una subcolumna, `actcol(j,k)`, indicando que la columna j está siendo actualizada por la columna k. En ese caso la computación quedará como se muestra en el apartado *right-looking* del ejemplo 1.7.

Ejemplo 1.7 PSEUDOCÓDIGOS ABREVIADOS:

Right-looking

```
1. for k=1 to n do
2.     divcol(k)
3.     for j=k+1 to n do
4.         actcol(j,k)
```

Left-looking

```
1. for j=1 to n do
2.     for k=1 to j-1 do
3.         actcol(j,k)
4.     divcol(j)
```

Vemos como en la iteración k la columna k actualiza las $n-k-1$ siguientes columnas. A esta actualización completa de la submatriz reducida descrita por las líneas 3 y 4 del ejemplo anterior, se le conoce con el nombre de actualización de rango uno, y como vemos, coincide con la ejecución de la ecuación 1.3.

Si reordenamos los bucles j y k, obtenemos la versión *left-looking* del algoritmo, en la que en la iteración j, la columna j es actualizada por las $j-1$ columnas previas, como se ve en el ejemplo 1.7, apartado *left-looking*.

En los dos casos hemos considerado que el índice k itera sobre las columnas *fuente* y el índice j sobre las columnas *destino*. Es importante hacer notar que la operación `actcol()` se ejecuta varias veces para cada iteración del bucle más externo, mientras que la operación `divcol()` sólo se ejecuta una vez por iteración. Por tanto la operación `actcol()` será la que domine en el tiempo de ejecución. Desde las perspectiva BLAS, la operación `actcol()` no es más que una llamada a la rutina `SAXPY`: $y = y + \alpha x$, donde ahora y es la columna destino, $A(k+1:n, j)$, x es la columna fuente, $A(k+1:n, k)$, y α es el coeficiente $A(k, j)$.

Evidentemente, también podríamos haber orientado el código siguiendo una organización "por filas" en lugar de "por columnas". Para ello sólo tenemos que dividir las filas de la matriz por el pivot reordenando el código del ejemplo 1.6, según se muestra en el ejemplo 1.8. Como vemos, el índice i sigue recorriendo las filas y el j las columnas. Según esta estrategia, será la matriz U la que tenga diagonal unidad (*unit upper triangular*). Este código *right-looking* es igualmente transformable en su homólogo *left-looking* por filas. Otras variaciones también están permitidas, como por ejemplo en la subrutina MA48 [61] donde las filas se dividen por el pivot, pero las actualizaciones se realizan por columnas, siguiendo la alternativa *left-looking*.

En caso de aplicar estos algoritmos a matrices densas el coste computacional es [57]:

$$\frac{2}{3}n^3 + O(n^2).$$

Ejemplo 1.8 PSEUDOCÓDIGO RIGHT-LOOKING LU POR FILAS

```
1.  for k=1 to n do
2.     for j=k+1 to n do
3.        A_kj ← A_kj / A_kk
4.     for i=k+1 to n do
5.        for j=k+1 to n do
6.           A_ij ← A_ij − A_ik A_kj
```

1.3.2 Aproximación a las versiones dispersas

Si consideramos matrices dispersas, sólo tendremos almacenados los elementos no nulos. La organización de los bucles se mantiene, pero la estructura de datos debe ser distinta, de forma que sólo se almacenen y consideren los elementos no nulos. Por ejemplo nos puede interesar, colocar en posiciones consecutivas de memoria aquellos elementos de cada columna. Las computaciones se realizaran por tanto, únicamente sobre los elementos no nulos de la matriz. Una muestra: la línea 3 del ejemplo 1.6 sólo se ejecutará para las iteraciones i en las que A_{ik} sea distinto de 0, que serán los únicos coeficientes almacenados. Esto también quiere decir que sólo un subconjunto de la columna destino j será afectada por la operación `actcol()`. Las ecuaciones 1.2 y 1.3 quedan ahora:

$$A_{ik} = A_{ik}/A_{kk} \quad \forall \; i : (k < i \leq n) \wedge (A_{ik} \neq 0) \tag{1.5}$$

$$A_{ij} = A_{ij} - A_{ik}A_{kj} \qquad \forall \; i,j : \begin{cases} (k < i, j \leq n) \\ (A_{ik} \neq 0) \wedge (A_{kj} \neq 0) \end{cases} \tag{1.6}$$

Subrayamos aquí que es la operación `actcol()` la responsable del llenado de la matriz, ya que la operación SAXPY con vectores dispersos puede modificar un coeficiente de la columna destino que inicialmente era cero. Es decir, el patrón final de la columna destino j, una vez actualizada, será la unión del patrón inicial con los patrones de todas las columnas k fuente.

En la figura 1.8 vemos gráficamente la evolución de una iteración k dada para un algoritmo *right-looking* sobre una matriz dispersa. Para esta iteración, los elementos no nulos de la columna k por debajo del pivot son divididos por éste (operación `divcol` representada gráficamente por tres flechas en el paso ①). Posteriormente, se realiza la actualización de rango uno en la submatriz reducida generándose 5 nuevos elementos no nulos y actualizándose un coeficiente ya existente de la iteración anterior (paso ② en la figura). En este caso sólo hay dos columnas destino ya que hay dos elementos en la fila `k`, por tanto la operación `actcol()` se realiza dos veces.

En la misma figura 1.8 podemos comparar el patrón de acceso para la variante *left-looking*. Al procesar la columna `j` de la matriz dispersa es necesario realizar inicialmente un estudio simbólico del patrón de dicha columna, que llamaremos `S`. Las componentes

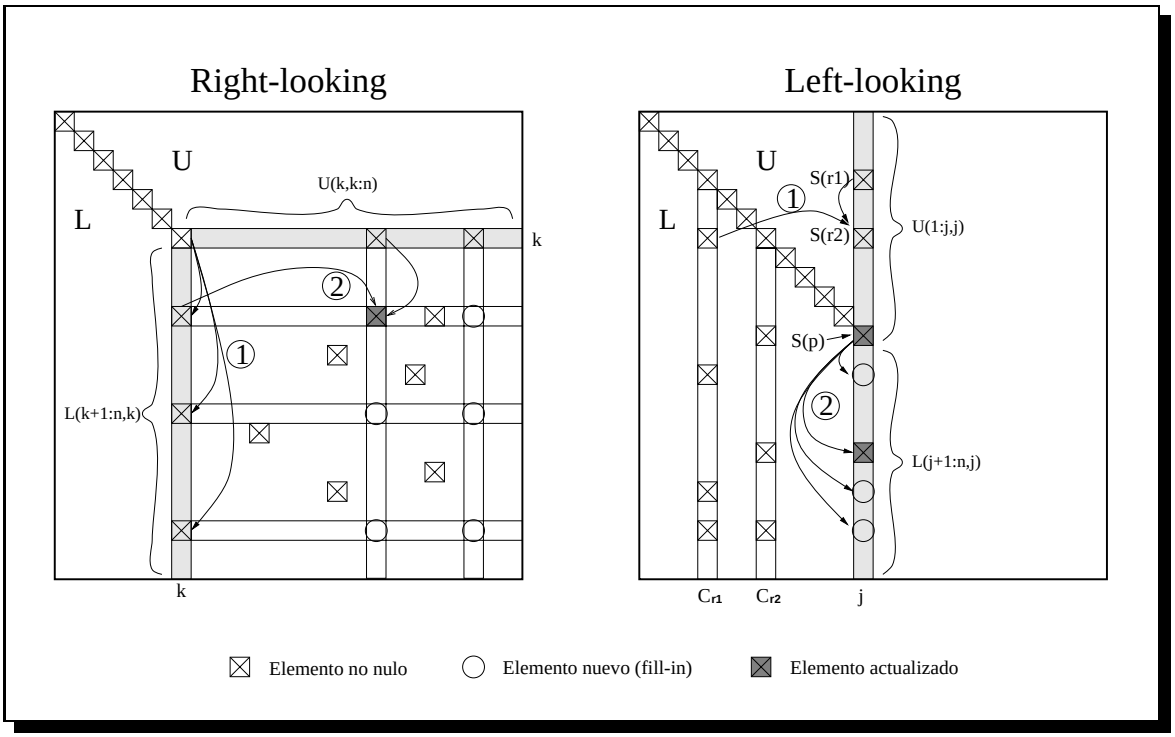


Figura 1.8: Flujo de acceso a datos para las alternativas *right* y *left-looking* en el caso disperso.

de filas menor o igual que `j` pasarán a formar parte de U, mientras que las restantes completarán la columna `j` de L. Dado que en la subcolumna superior tenemos dos coeficientes distintos de cero, $S(r_1)$ y $S(r_2)$, la columna destino, `S`, se actualizará con las dos columnas fuente, c_{r_1} y c_{r_2}, asociadas a los dos coeficientes mencionados (paso ① en la figura). Esta actualización debe realizarse en un determinado orden: el de índices crecientes es el más típico. En efecto, la columna c_{r_2} multiplicada por $S(r_2)$ actualiza el resto de la columna `S`, pero esa operación debe realizarse una vez $S(r_2)$ haya sido actualizado en el `actcol()` anterior asociado a $S(r_1)$. En este caso, se dice que $S(r_2)$ depende de $S(r_1)$ debido a la existencia del segundo elemento no nulo de la columna c_{r_1}. De no existir esta dependencia las dos actualizaciones se podrían realizar en cualquier orden.

Una vez actualizada la columna `S` con las columnas de L oportunas, se dividen las entradas con índice de fila mayor que `j` por el coeficiente de coordenadas (j, j) o pivot, como se indica con un ② en el apartado *left-looking* de la figura 1.8. En los dos apartados de esta figura, la fila/columna en gris representa los coeficientes que pasan a formar parte de los factores L y U para una iteración del bucle más externo.

La complejidad computacional de la factorización de matrices dispersas es del orden de $O(c^2 n)$, donde, como se apuntó en la sección 1.1, c representa el número medio de elementos no nulos por columna (o fila). Una de las sutilezas importantes a tener en cuenta cuando se escribe código para matrices dispersas, consiste en evitar una complejidad de $O(n^2)$ o más. Es relativamente fácil incurrir inocentemente en códigos con bucles de recorrido n, que si están anidados resultan en la complejidad mencionada.

Para terminar de aclarar estos aspectos, presentamos en el ejemplo 1.9 la factoriza-

ción de una matriz dispersa de dimensión $n = 4$ con $\alpha = 9$ elementos no nulos. Hemos utilizado para ello un algoritmo *right-looking*. Después de tres iteraciones la matriz A contiene los factores L (en la submatriz triangular estrictamente inferior) y U (en la submatriz triangular superior).

Ejemplo 1.9 FACTORIZACIÓN DISPERSA RIGHT-LOOKING

$$\boxed{\textit{Matriz } A\text{: } A^{(1)}} \qquad \boxed{A^{(2)}} \qquad \boxed{A^{(3)}} \qquad \boxed{A^{(4)}}$$

$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 2 & 3 & 2 \\ 9 & -4 & 0 & 2 \\ 0 & 4 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 2 & 3 & 2 \\ 3 & -4 & -3 & 2 \\ 0 & 4 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 2 & 3 & 2 \\ 3 & -2 & 3 & 6 \\ 0 & 2 & -6 & -4 \end{pmatrix} \quad \begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 2 & 3 & 2 \\ 3 & -2 & 3 & 6 \\ 0 & 2 & -2 & 8 \end{pmatrix}$$

$$LU = \begin{pmatrix} 1 & & & \\ 0 & 1 & & \\ 3 & -2 & 1 & \\ 0 & 2 & -2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 & 0 & 1 & 0 \\ & 2 & 3 & 2 \\ & & 3 & 6 \\ & & & 8 \end{pmatrix} = \begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 2 & 3 & 2 \\ 9 & -4 & 0 & 2 \\ 0 & 4 & 0 & 0 \end{pmatrix} = A$$

1.3.3 El problema de la estabilidad y la dispersión

Está claro que ni la representación de los datos ni la computación es exacta en el ordenador. Por tanto las soluciones numéricas de los problemas computacionales tampoco serán precisas, y es importante saber y poder controlar el error numérico cometido. En la solución del problema $Ax = b$, tendremos en cuenta dos aspectos:

- Un algoritmo es **estable** si permite obtener una solución cercana a la exacta, $\tilde{x} \approx x$ (utilizamos la tilde, $\tilde{}$, para indicar "valor aproximado"). Es decir, el error cometido no será mayor que el que resultaría de perturbar ligeramente el sistema, $\tilde{A}$, y luego resolverlo de forma exacta.

- Una matriz es **mal-condicionada** si un pequeño cambio en los coeficientes de la matriz da lugar a un cambio significativo en la solución. En caso contrario la matriz es **bien-condicionada**.

Es importante subrayar que la condición es patrimonio de la matriz, mientras que la estabilidad lo es del algoritmo. Sin embargo, es evidente que es necesario un algoritmo especialmente estable para resolver un sistema mal-condicionado.

Ha sido comprobado que el algoritmo de factorización LU es estable siempre que se evite un desmesurado crecimiento de los coeficientes de la matriz [124, 157]. Para ello es suficiente que en la operación `divcol()` el pivot siempre sea el de mayor valor absoluto de toda la submatriz activa en la iteración `k`, es decir:

$$|A_{kk}^{(k)}| \geq |A_{ij}^{(k)}|, \quad i \geq k,\ j \geq k.$$

Por tanto, la submatriz activa debe ser recorrida para encontrar el coeficiente de mayor valor absoluto. Mediante una permutación de filas y columnas, esa entrada pasará a ser el pivot. Esta estrategia se llama de **pivoteo total** o completo. El alto coste computacional de esta estrategia, $\frac{1}{3}n^3 + O(n^2)$, nos inclina a utilizar la técnica de **pivoteo parcial** en la que sólo se recorren y permutan filas o columnas.

En la figura 1.9 queda plasmado gráficamente el acceso a datos requerido por las distintas alternativas. Se aprecia claramente, cómo el tipo de pivoteo requerido tiene implicaciones directas sobre la estructura de datos que almacena la matriz. O a la inversa, si por ejemplo la estructura de datos permite únicamente acceso por filas o por columnas resultará muy costoso aplicar la técnica de pivoteo total. Más bien respaldados por la experiencia que por el riguroso análisis, se puede afirmar que el algoritmo de factorización LU con pivoteo parcial es estable [55]. Para matrices mal condicionadas siempre se pueden utilizar técnicas de escalado o de refinamiento iterativo [57].

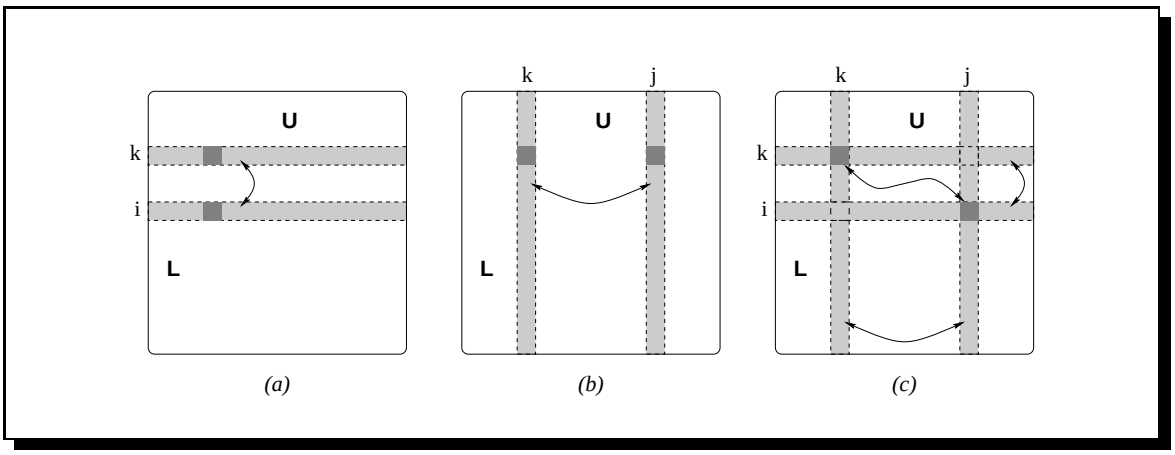


Figura 1.9: Flujo de acceso en operaciones de pivoteo: (a) parcial por filas; (b) parcial por columnas; y (c) total.

Para monitorizar la estabilidad es suficiente con que $||H||$ sea mucho menor que $||A||$ en alguna norma, donde $H = \tilde{U}\tilde{L} - A$, o si ya tenemos el vector solución $\tilde{x}$, comparando la norma del residuo $r = b - \tilde{x}A$ con b. Más precisamente, podemos indicar el error cometido resolviendo el sistema $Ac = b$ y calculando $||c - \tilde{c}||$. Para ello, partimos de un vector conocido, c, de coeficientes aleatorios entre, y calculamos $b = Ac$, para luego resolver el sistema con el vector b obtenido. Nosotros utilizaremos norma infinito, es decir:

$$\text{error} = \max_i \{|c_i - \tilde{c_i}|\}.$$

Para matrices dispersas la estabilidad numérica no es el único factor a tener en cuenta a la hora de elegir el pivot. También es muy importante que la matriz se mantenga dispersa durante la factorización. De esta forma, el tiempo de factorización será menor, ya que minimizando el llenado de la matriz ahorramos operaciones aritméticas. El problema del llenado mínimo se ha demostrado NP-Completo incluso para versiones simplificadas del mismo [128]. Por tanto ha sido inevitable el aplicar heurísticos en la persecución del llenado mínimo. Los dos más extensivamente usados

son el **criterio de Markowitz** de 1957 [102] y un caso particular de éste para matrices simétricas llamado **mínimo grado** de 1967 [143]. A pesar de la antigüedad de estos métodos prácticamente no se han aportado modificaciones significativas. Para matrices simétricas las alternativas modernas son la versión rápida del algoritmo de mínimo grado, AMD (*Aproximate Minimun Degree*) [8] y las técnicas por bisección, recientes competidoras de las de mínimo grado [22].

Cuando la matriz es no simétrica, el criterio de Markowitz aún se mantiene como la mejor opción entre los heurísticos de llenado mínimo, así pues, veamos su justificación. Para una iteración k del algoritmo, supongamos la selección de la entrada $A_{ij}^{(k)}$ como pivot. De las ecuación 1.6 de actualización dispersa, se deduce fácilmente que esta elección dará lugar a la modificación de $M_{ij}^{(k)} = (R_i^{(k)} - 1)(C_j^{(k)} - 1)$ coeficientes en la submatriz reducida, donde $R_i^{(k)}$ representa el número de entradas en la fila i de la matriz activa en la iteración k, mientras que $C_j^{(k)}$ el número de entradas en la columna j de la misma matriz activa.

A la variable $M^{(k)}$ se la denomina **cuenta de Markowitz** y representa el número máximo de nuevas entradas que pueden crearse para una iteración `k` del algoritmo. El pivot debe ser elegido de forma que se minimice este valor $M^{(k)}$. Las búsqueda de tal pivot resulta costosa al tener que recorrer toda la submatriz activa. Por ello, es preferible buscar un mínimo $R_i^{(k)}$ en sólo algunas columnas con mínimo $C_j^{(k)}$ (llamada técnica de *mínima fila en mínima columna*) o viceversa. Esta estrategia puede dar lugar a un mayor llenado pero el efecto global es positivo al reducir considerablemente el tiempo de selección del pivot.

Así pues, por un lado tenemos que asegurar la estabilidad numérica del algoritmo, mientras que por otro, evitar un excesivo llenado de la matriz. Por tanto, el pivot elegido debe tener el mayor valor absoluto junto con la mínima cuenta de Markowitz posible. Para buscar ese candidato se han de relajar en cierta medida ambos criterios. Más prioridad debe tener la estabilidad numérica, que está relacionada con la precisión en la solución del problema, mientras que el llenado está asociado principalmente al consumo de tiempo y memoria.

Por tanto, el heurístico utilizado va a consistir en una combinación de los criterios de pivoteo total y de Markowitz. Por lo pronto, dentro de una columna j de la matriz activa, serán seleccionadas como entradas, $A_{ij}^{(k)}$, candidatas a ser pivots, todas aquellas que tengan mayor valor absoluto que el máximo valor absoluto multiplicado por un parámetro u:

$$|A_{ij}^{(k)}| \geq u \cdot \max_l |A_{lj}^{(k)}| \tag{1.7}$$

La columna j se selecciona minimizando $C_j^{(k)}$, y entre los candidatos que superan el umbral, se selecciona como pivot aquel que pertenezca a la fila con mínimo $R_i^{(k)}$. Es evidente, que el parámetro de umbral u, de rango: $0 < u \leq 1$, controla si se prima la estabilidad numérica ($u \to 1$) o la preservación del coeficiente de dispersión ($u \to 0$). Un valor típico avalado por la experiencia es $u \approx 0.1$ [57, Cap. 7], como también corroboraremos en la sección 3.6 de resultados experimentales.

1.3.4 Etapas en la resolución del sistema

Dada la complejidad que presenta la resolución del sistema disperso $Ax = b$ mediante métodos directos, es habitual dividir el problema en fases abordables independientemente. Típicamente estos métodos suelen seguir los siguientes pasos para completar la factorización:

1. **Reordenación**: En esta etapa reordenamos la matriz con la intención de reducir la complejidad computacional de las etapas siguientes. Por ejemplo, podemos reordenar la matriz A en una matriz triangular por bloques o diagonal por bloques. De esta forma, cada bloque puede ser factorizado independientemente, ya que el llenado quedará confinado dentro de el. Profundizaremos en los aspectos de esta etapa en el capítulo siguiente.

2. **Análisis**: Determina las matrices de permutaciones de filas, Π, y columnas, Γ, apropiadas para la factorización. Estas matrices de permutación serán las que determinen qué elementos serán pivots. La elección de los pivots debe hacerse cuidadosamente, de forma que, por un lado preserven el coeficiente de dispersión de la matriz (aplicando, por ejemplo, el criterio de Markowitz), y por otro lado, se garantice la estabilidad numérica (por ejemplo, seleccionando pivots con un valor absoluto superior a cierto umbral).

3. **Factorización**: Como entrada a esta etapa debemos proporcionar tanto la matriz A como las matrices de permutación, Π y Γ obtenidas de la etapa anterior. Con esa información se realizará la factorización de la matriz A para dar lugar a las matrices L y U, tal que $\Pi A \Gamma = LU$. Dado que en esta etapa es donde se realizarán las operaciones aritméticas en punto flotante, podemos afirmar que es la más costosa computacionalmente.

4. **Resolución triangular**: A partir de las matrices L y U se resuelve el sistema de ecuaciones $Ax = b$ o $A^T x = b$. Incluye por tanto los algoritmos de sustitución hacia adelante (*forward substitution*) y sustitución hacia atrás (*backward substitution*).

El origen de la separación entre las etapas de análisis y factorización tiene su razón de ser en el procesado de matrices simétricas, principalmente definidas positivas, o casi-simétricas. En este caso la elección de los pivots se puede realizar inicialmente operando únicamente con el patrón de la matriz sin necesidad de referenciar los valores numéricos. Ésto reduce sustancialmente el tiempo de análisis, al final del cual se conoce el número final de entradas en la matriz, y de esta forma se puede evitar la gestión dinámica de memoria en la etapa de factorización.

En el caso de resolver matrices no simétricas donde tiene especial relevancia considerar los valores numéricos de los coeficientes al elegir los pivots, es menos ventajoso separar las dos etapas mencionadas, sobre todo en códigos *right-looking*. Aquí, el análisis y la factorización se ejecutan de forma simultánea en una única fase conocida como etapa **análisis-factorización**. Esta unificación es inevitable en algoritmos que explotan el paralelismo asociado a la dispersión de la matriz buscando un cierto

número de pivots compatibles, como se discute en el capítulo 3. De esta forma, para cada iteración k del algoritmo, primero se determina cuál ha de ser el siguiente pivot, a continuación se permutan las filas y las columnas adecuadas para colocar el pivot en la posición (k,k) de la matriz, y finalmente se actualiza la submatriz reducida para dicha iteración.

Sin embargo en códigos *left-looking* o cuando se va a realizar únicamente un pivoteo parcial en la etapa de factorización sigue siendo recomendable preceder la factorización de una rápida etapa de análisis en la que se reordenan filas y columnas en aras de la estabilidad y la dispersión. El coste de esta etapa es mucho menor que el de la factorización y permite conocer las necesidades de memoria con anterioridad.

De cualquier modo, incluso cuando la etapa de factorización está precedida por una etapa análisis independiente, es necesario incluir durante la factorización algún tipo de pivoteo numérico para evitar una posible pérdida de estabilidad numérica. Por ejemplo, esta pérdida de estabilidad podría tener lugar eventualmente si un pivot que ha sido seleccionado como tal durante la etapa de análisis se modifica, tomando un valor absoluto pequeño durante la etapa de factorización. Por tanto, dentro de esta última aparecen secciones de código encargadas de la **factorización numérica** donde realmente se actualiza la matriz, y secciones de **factorización simbólica** donde se determina para cada iteración si es necesario un pivoteo parcial adicional.

Las etapas de factorización y resolución triangular pueden ser subdivididas a su vez en sub-fases. Dado que el procesado de la matriz va recorriendo la matriz siguiendo la diagonal, el llenado tiene mayor impacto sobre la submatriz inferior derecha, que es la que más veces sufre la operación de actualización. Es normal que llegados a cierto punto la submatriz reducida a procesar adquiera tal densidad que tenga sentido **conmutar del código disperso a uno denso**. Por tanto, la etapa de factorización puede dividirse en una factorización dispersa más otra densa. A su vez, la resolución triangular consistirá en en una substitución dispersa y otra densa hacia adelante y posteriormente una substitución densa más otra dispersa hacia atrás. Se comprueba experimentalmente que el coste asociado a la conmutación y cambio en la estructura de datos es despreciable frente a la reducción en el tiempo de ejecución conseguido.

En la figura 1.10 (a) presentamos una matriz ejemplo de dimensiones $n = 760$ y con $\alpha = 5976$. A su lado, (b), encontramos el patrón de la matriz una vez factorizada, donde la submatriz inferior derecha marcada tiene una densidad del 20% con 10860 entradas, mientras que el número de entradas en el resto de la matriz es de 9131. Esa densidad se alcanza en la iteración $k = 527$, cuando aún quedan 233 iteraciones por completar. Es evidente la reducción del tiempo de computación que podemos obtener si procesamos la submatriz reducida de 233×233 mediante un algoritmo denso evitando los costes y las estructuras de datos asociadas a la versión dispersa. La densidad a la que resulta ventajoso permutar a un algoritmo denso ha sido estudiada y se discute en la sección 3.6. La iteración de conmutación puede ser determinada por la etapa de análisis si ésta existe como tal, o durante el proceso de análisis-factorización chequeando la densidad de la submatriz reducida en cada iteración.

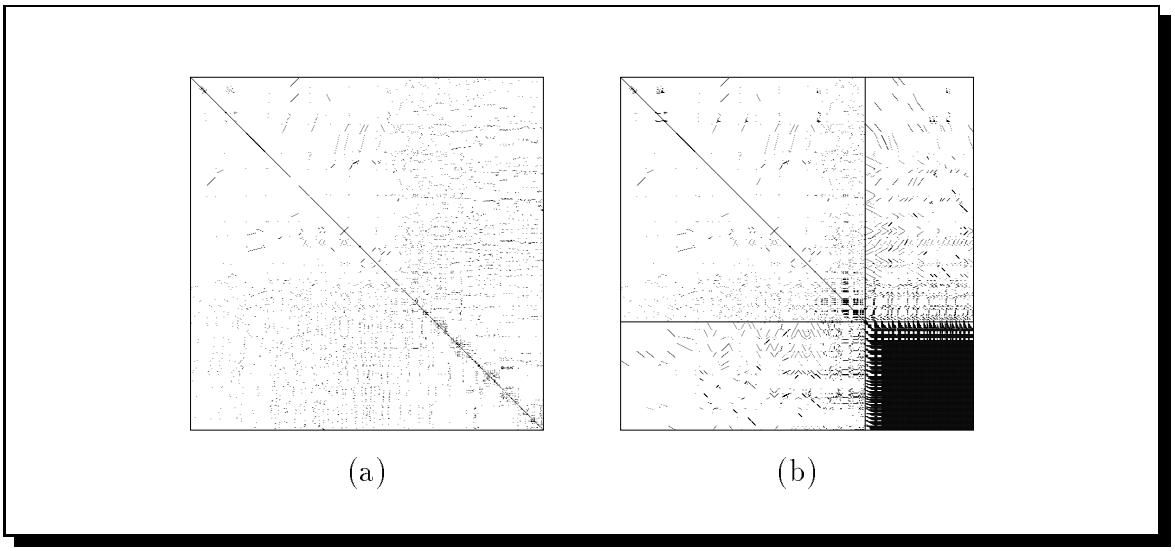


Figura 1.10: Una matriz ejemplo antes de factorizar (a), y la misma matriz después de su factorización (b). La submatriz inferior derecha tiene una densidad del 20%.

Capítulo 2

Arquitecturas y programación paralela

Gracias al claro incremento del coste computacional de las aplicaciones actuales, existe una gran demanda de arquitecturas cada día más rápidas y con mejores prestaciones. En esta evolución de los computadores, las arquitecturas paralelas son un paso inevitable para la consecución de mayores velocidades de procesamiento.

Como consecuencia, los modelos y técnicas de programación también han evolucionado para cubrir el desarrollo de algoritmos para estas nuevas plataformas paralelas. Sin embargo, es un hecho conocido, que la evolución de las herramientas de desarrollo de algoritmos paralelos sigue un menor ritmo que el de estas arquitecturas. Esta situación impide en gran medida el uso generalizado de las máquinas paralelas existentes. Por ello, actualmente se están incrementando los esfuerzos para idear y mejorar distintos paradigmas de programación paralela.

Así pues, en este capítulo dedicamos dos secciones a introducir las nociones básicas implicadas en los dos parrafos anteriores: en la sección 2.1 daremos un repaso rápido al soporte *hardware* paralelo disponible actualmente, centrándonos en la plataforma particular que hemos utilizado para validar nuestras soluciones: el Cray T3D/T3E; seguidamente, el punto de vista *software*, es decir el de las utilidades de ayuda a la programación disponibles para las plataformas paralelas, será tratado en la sección 2.2.

2.1 Las arquitecturas paralelas

Por motivos tecnológicos, el número de transistores utilizados en un procesador está creciendo más rápidamente (40% por año) que la frecuencia de reloj (10% por año). Ese mayor número de transistores ha redundado en avances de tipo arquitectural, que han dominado, frente al aumento de la frecuencia, en el incremento global de prestaciones del procesador. Estos avances han conducido a procesadores con **paralelismo a nivel de instrucción** (ILP) que reducen (en media) el número de ciclos de reloj necesarios para ejecutar una instrucción o que incluso ejecutan varias instrucciones en paralelo.

Esto ocurre por ejemplo en las arquitecturas **superescalares** o en las más complejas **VLIW** (Very Long Instruction Word) donde existen gran cantidad de unidades funcionales independientes. Es tarea del compilador el reestructurar los programas de forma que se maximice el número de unidades funcionales que operan en paralelo.

Sin embargo, el crecimiento de las prestaciones de un procesador debido al paralelismo a nivel de instrucción también es limitado (por ejemplo, actualmente no es rentable el coste asociado a una arquitectura que lanza a ejecutar ocho instrucciones o más). Así que, si la tendencia es seguir aumentando el número de transistores, tendremos que cambiar la estrategia ILP por otra que permita un paralelismo a un nivel mayor (o más grueso). Esto justifica la relevancia de los **multiprocesadores**, o supercomputadores con un conjunto de procesadores que cooperan para la rápida resolución de grandes problemas computacionales [89]. Cuando el número de procesadores es elevado utilizaremos la denominación de sistemas MPP *Massively Parallel Processor* o procesadores masivamente paralelos.

En el siguiente apartado introducimos algunas nociones, parámetros característicos y clasificación de los multiprocesadores actuales. Nos centraremos más tarde en la plataforma paralela sobre la que hemos montado y chequeado nuestros algoritmos.

2.1.1 Generalidades y clasificación

La evolución de las arquitecturas paralelas ha derivado en el desarrollo de distintos modelos, como los arrays sistólicos, máquinas de flujo de datos, arquitecturas de memoria compartida, computadores de pase de mensajes, etc. Actualmente se está produciendo un movimiento de convergencia de estas arquitecturas que, en base a la experiencia pasada, extrae y auna las características beneficiosas de cada estrategia descartando las demás.

Por lo pronto, las tendencias actuales han extendido la noción clásica de arquitectura de computadores de forma que se contemplen los conceptos de **comunicación** y **cooperación**. Es decir, en el diseño de un sistema multiprocesador, además de la arquitectura del procesador o nodo del sistema, está implicada una arquitectura de comunicaciones. Esta última se organiza en una capa de más alto nivel, de primitivas de comunicación (HW/SW) visibles por el programador, y en la capa de bajo nivel que contempla la implementación eficiente de esas primitivas ya sea mediante memoria compartida o distribuida.

Los parámetros de diseño importantes de esta arquitectura de comunicaciones son:

- **Latencia**: Tiempo necesario para iniciar una comunicación.
- **Ancho de Banda**: Número de datos que se pueden comunicar por segundo.
- **Sincronización**: Modo en que se sincronizan los productores y consumidores de datos.
- **Granularidad de los Nodos**: Criterios para asignar silicio (o número de transistores) a unidades de procesamiento o a memoria.

- **Aplicabilidad**: Propósito general o especial.

- **Denominación**: Define como referenciamos a los datos compartidos entre los procesos, y la semántica de las primitivas que soportan los lenguajes de programación. Existen dos alternativas básicas:

  - Modelo de **Memoria centralizada o global** donde todos los procesadores pueden acceder de forma directa a todas las posiciones de memoria del sistema. Un dato escrito en memoria es accesible por cualquier procesador de forma que los nodos pueden intercambiar información (cooperación). Es decir la comunicación se integra a nivel de memoria.
  - Modelo de **Memoria privada** o **Pase de mensajes** donde los procesadores tienen acceso a su espacio de direcciones local y las operaciones de cooperación (comunicación y sincronización) se realizan mediante pase de mensajes. Es decir, éstas operaciones se integran a nivel de E/S y son necesarias llamadas explícitas al S.O. para realizarlas (rutinas *send/receive*).

Estos modelos de memoria global y pase de mensajes pueden ser considerados como la capa de alto nivel visible por el programador, pero no tiene por qué establecer restricciones HW fundamentales. De hecho, ambos representan modelos de programación que pueden ser simulados tanto en maquinas físicamente con **memoria compartida** (los procesadores acceden a cualquier módulo de memoria mediante una red de interconexión), o **memoria distribuida** (los procesadores acceden a su memoria local e intercambian mensajes con otros nodos mediante una red de interconexión).

Independientemente de si la red de interconexión está entre procesadores y módulos de memoria o entre nodos compuestos de procesador y memoria local, las necesidades de eficiencia de la red de interconexión son similares: es necesario reducir la latencia y aumentar el ancho de banda. Si no es posible reducir la latencia mediante caches, controladores de comunicaciones, etc, podemos reducir el coste asociado agrupando comunicaciones o solapando computaciones con comunicaciones. La topología de la red de interconexión afecta directamente a la latencia, pero consideraciones de coste resultan en diseños finales con topologías más baratas aunque tengan menor conectividad. Las topologías más populares actualmente en orden de mayor a menor conectividad son: hipercubo, toro y malla; bidi- o tridimensional estas dos últimas [89].

Ejemplos de máquinas comerciales recientes con memoria compartida son el SGI Power Challenge y los Y-MP y C90 de Cray. En el otro bando, máquinas con memoria distribuida son: SP2 de IBM con RS6000, CM5 Connection Machine de Thinking Machines con procesadores SPARC, CS2 Computing Surface 2 de Meiko, Paragon de Intel con i860, NCube, Maspar MP-2, y otras.

Sin embargo, la convergencia arquitectural, de la que hablábamos al comienzo, está dando lugar a arquitecturas de **memoria compartida-distribuida**, donde físicamente la memoria está distribuida entre los procesadores, pero existe un soporte HW para permitir un espacio de direcciones globales e incluso gestión de la coherencia entre caches locales a cada nodo. Esta filosofía auna la sencillez de la programación en un modelo de memoria compartida junto con la escalabilidad que proporcionan los sistemas de memoria distribuida.

Las máquinas con memoria compartida-distribuida se pueden organizar en dos grandes bloques[1]:

- Máquinas **NUMA** (*NonUniform Memory Access*) o con espacio de direcciones físico estático entre las que encontramos arquitecturas **sin coherencia cache**, como el Cray T3D con procesadores DEC-Alpha 21064, **con coherencia parcial de cache**, como el Cray T3E con DEC-Alpha 21164, o **con coherencia cache** (CC-NUMA o *Cache Coherent NonUniform Memory Access*) como el Origin 2000 de SGI con procesadores R10000.

- Máquinas **COMA** (*Cache Only Memory Architecture*) o con espacio de direcciones físico dinámico, donde las memorias distribuidas son convertidas en caches. En este caso particular de máquinas NUMA no hay jerarquía de memoria en cada procesador. Esto ocurre en la KSR-1 de Kendall Square Research y en la arquitectura I-ACOMA propuesta en la universidad de Illinois [144].

Por último, otro modelo que queremos tratar es el de las **arquitecturas de paralelismo de datos**. Históricamente, estas máquinas se construyen mediante una matriz de gran cantidad de procesadores simples con una memoria local limitada. Esta matriz se conecta a un procesador de control de propósito general que distribuye las instrucciones que serán ejecutadas síncronamente. La arquitectura se denomina **SIMD** de *Single Instruction Multiple Data*. Los procesadores trabajan con sus datos locales y cooperan mediante una red de comunicaciones eficiente y con capacidad de sincronización global rápida. Ejemplos de esta arquitectura encontramos en el CM-2 de Thinking Machines y el Maspar MP-2. Su aplicabilidad reducida a cálculos científicos regulares provoca que el modelo converja (de nuevo la idea de convergencia arquitectural) o evolucione en el modelo **SPMD** (Single Program Multiple Data). Bajo el modelo SPMD, un mismo programa es cargado en los distintos nodos del multiprocesador, mientras que los datos se distribuyen entre ellos, de forma que los procesadores operarán mayoritariamente con la fracción local de datos que les ha correspondido. Sin embargo el modelo de programación asociado a las arquitecturas de paralelismo de datos, sigue vigente debido a su sencillez de programación, y será tratado en la sección 2.2.

Cuando en la arquitectura del sistema multiprocesador, no existe una sincronización a nivel de instrucciones, los procesadores tienen una capacidad de proceso elevada, capaces de ejecutar programas de forma independiente, pero están conectados mediante una red de comunicaciones, a la arquitectura se la conoce con el nombre **MIMD** (*Multiple Instruction Multiple Data*).

2.1.2 Los multiprocesadores Cray T3D y T3E

Dedicamos una subsección a los computadores masivamente paralelos Cray T3D y T3E ya que han sido utilizados para comprobar la validez de los algoritmos descritos en esta memoria.

[1] Aunque esta taxonomía aparece inicialmente para clasificar máquinas de memoria compartida.

Cray T3D

El **Cray T3D** es un sistema MPP MIMD con arquitectura NUMA que puede llegar a proporcionar 300 GFLOPS[2] de pico en su configuración máxima de 2048 procesadores [35]. El sistema completo consta de un array de procesadores conectado a un "host" donde se ejecuta el sistema operativo. El array de procesadores está compuesto de nodos de procesamiento, donde cada uno de ellos contiene dos procesadores DEC Alpha 21064 a una frecuencia de 150 MHz. Este procesador soporta operaciones enteras de 64 bits y flotantes en formato IEEE-754 de 64 bits, permitiendo unas prestaciones de pico de 150 MFlops de 64 bits. En cuanto a la jerarquía de memoria, el Alpha 21064 incluye una cache de datos de 8Kbytes y otra de instrucciones del mismo tamaño. Las caches contienen 256 líneas de 32 bytes cada una. Cada procesador tiene disponible una memoria local de 64Mbytes.

Los nodos están conectados entre sí por una topología de toro tridimensional, donde cada uno de los seis enlaces soporta velocidades de transferencia de hasta 300 Mbyte/s. Aunque, como vemos, la memoria está físicamente distribuida, existe un soporte *hardware* para proporcionar un espacio único de direcciones aunque no está garantizada la coherencia cache en el sistema multiprocesador [45]. La circuitería de soporte necesaria para extender el control y las funciones de direccionamiento del procesador permite:

- Interpretación de direcciones. En el Cray T3D los pines de direcciones del procesador no se conectan directamente al bus de direcciones de la memoria física, sino que la circuitería de soporte interpreta la dirección y encamina los datos entre el procesador y la memoria local o remota.

- Escrituras no bloqueantes[3], lecturas cacheables[4] o no, y prebúsqueda de datos de memorias no locales.

- Soporte para el paso de mensajes entre procesadores. Cuando se recibe un mensaje, la circuitería de soporte lo coloca en una cola de mensajes e interrumpe al procesador para que pueda leerlo de la cola. La cola de mensajes se sitúa en memoria local y puede almacenar hasta 256 Kbytes de información.

- Barreras de sincronización. Se soportan dos tipos de operaciones de sincronización: barreras y eurekas. Existen dos registros de barrera de 8 bits cada uno, donde cada bit está asociado a circuitos de sincronización de barrera independientes y con idéntico funcionamiento. El circuito de sincronización de barrera está formado por un árbol de puertas AND que devuelve un 1 a todos los procesadores si todos tienen el bit correspondiente a 1. Las sincronizaciones de eureka permite avisar a los demás procesadores cuando uno de ellos ha completado cierta información. El circuito de sincronización de eurekas está asociado a una operación OR global.

[2] Un GFLOPS *Giga FLoating-point OPearations per Second* es equivalente a mil millones de operaciones en punto flotante por segundo.

[3] Este tipo de escritura permite al procesador seguir procesando instrucciones mientras que la circuitería de soporte completa la operación de escritura, ya sea local o remota.

[4] Una operación de lectura cacheable copia el dato leído en cache.

En cuanto a las operaciones de entrada/salida, el array de procesadores incluye cuatro puertos de E/S: dos de ellos proporcionan conexión directa con dispositivos de E/S, mientras que los otros dos conectan el array con el "host" que puede ser un Cray Y_MP o un C90, y que proporciona posibilidades adicionales de E/S (conexión a red, almacenamiento masivo, etc...). Estos puertos de E/S permiten un ancho de banda teórico bidireccional de 400 Mbyte/s.

Cray T3E

El Cray T3E es el siguiente miembro de la familia de arquitecturas MPP de Cray Research Inc. Mantiene las características MIMD y NUMA de su predecesor, pero ahora es escalable hasta 2176 procesadores de forma que puede llegar a proporcionar un TeraFLOP de pico [46].

El T3E no necesita un "host" al que conectarse sino que se dice que es *self hosted*. La matriz de procesadores está compuesta por nodos de procesamiento, pero esta vez, cada nodo sólo tiene un procesador: el DEC Alpha 21164 a 300MHz capaz de proporcionar 600 MFLOPS de pico ya que su característica superescalar le permite procesar hasta dos operaciones en punto flotante y dos enteras por cada ciclo de reloj.

En cuanto a la jerarquía de memoria, el procesador incluye:

- Caches internas de datos e instrucciones de 8Kbytes (256 líneas) cada una y con asignación directa. La cache de datos tiene una política de actualización de memoria principal de escritura directa (*write-through*). La cache de instrucciones puede almacenar hasta 2048 instrucciones.

- Segundo nivel de Cache interna para datos e instrucciones, asociativa por conjuntos de 3 líneas por conjunto y con capacidad de 96Kbytes. Utiliza la política de escritura retardada (copy-back) y las lecturas y escrituras son cacheables.

- También dentro del chip existe un buffer de escrituras y un fichero de direcciones fallidas (Missed Address File –MAF–) que combina fallos de cache de distintas palabras de una misma línea para que sean cargadas en una única operación de lectura.

En el nodo de procesamiento se encuentra la memoria local configurable entre 64Mbytes y 2Gbytes, junto con la circuitería de soporte [134]. Esta circuitería de soporte consiste en:

- 32 circuitos de barrera y eureka para 32 contextos distintos.

- Registros E, usados para ocultar la latencia al acceder a posiciones de memoria no locales. Existen 512 registros E de usuario y 128 de sistema.

- *Hardware* auxiliar para translación de direcciones, control de errores, coherencia cache, etc.

En relación al último punto hay que matizar que la coherencia cache se mantiene sólo parcialmente. Es decir, si un procesador tiene un dato modificado en su cache, un acceso por parte de otro procesador a ese dato de la memoria principal provoca una actualización previa de dicha memoria. Sin embargo, si la misma dirección global está en caches de procesadores distintos, las modificaciones de una no se reflejan en la otra.

La interconexión de los nodos también utiliza una topología de toro tridimensional, pero el ancho de banda de la red aumenta a 480Mbytes/s. Esta red está controlada por un *hardware* específico de enrutado. En cuanto a la entrada/salida, existen controladores de E/S que permiten una conexión en anillo, llamada GigaRing, para permitir la comunicación entre cualquier procesador del T3E y un dispositivo externo.

2.2 Herramientas para el desarrollo

En la sección anterior hemos repasado varias de las arquitecturas adoptadas en máquinas paralelas. Sin embargo, ese aspecto *hardware* debe ser combinado con las utilidades *software* necesarias en función de la técnica de programación que vaya a ser utilizada. Básicamente, podemos decir que existen cuatro alternativas, de aceptación general, para construir un programa paralelo: paralelización manual usando **pase de mensajes**, paralelización semi-automática basada en el paradigma de **paralelismo de datos**, **paralelización automática** y paralelización manual en **memoria compartida**. Las cuatro han sido abordadas en la memoria, por lo que serán introducidas bajo este epígrafe.

Ya sea considerando una u otra aproximación, debemos contemplar el mismo aspecto básico: en máquinas paralelas el problema a resolver se divide entre los procesadores disponibles, con la intención de conseguir el mismo resultado que obtendríamos con uno sólo, pero en un tiempo inversamente proporcional al número de procesadores. Existen muchas formas de implementar esta descomposición del problema. Por ejemplo, en función de la granularidad, tendremos paralelismo a:

- **nivel de instrucción** (grano fino) cubierto por los procesadores superescalares, VLIW, etc, como se comentó al comienzo de la sección 2.1.
- **nivel de bucle** (grano medio) donde manteniendo la estructura del código secuencial, los bucles sin dependencias tienen repartido su espacio de iteraciones entre los procesadores disponibles. El paralelismo a este nivel domina en problemas de computación científica. Una razón importante para ello es que, si aumenta el tamaño del problema, suele ser suficiente con incrementar el número de procesadores para conseguir la solución en el mismo tiempo.
- **nivel de tarea** (grano grueso) aplicable cuando tenemos un gran número de tareas semi-independientes que pueden ser ejecutadas simultáneamente en varios procesadores. La simplicidad de este modelo, permite a sistemas operativos como Solaris, Windows NT, u OS/2 [5] explotar este tipo de paralelismo operando sobre

[5] Generalmente sistemas operativos *multithread* que manejan procesos simples que se comunican mediante memoria compartida: 'threads'.

una máquina con pocos procesadores y memoria compartida.

Pero quizás el aspecto más importante resida en elegir una descomposición del problema que provea una distribución equitativa de la carga (el trabajo) entre los procesadores de forma que éstos estén ocupados durante el mismo tiempo (**balanceo de la carga**), y por otro lado, minimice el número de comunicaciones interprocesador necesarios para la resolución global del problema (**overhead de comunicaciones**). Esta última cuestión estará condicionada en gran medida por el nivel de **localidad** que consigamos alcanzar. Explotar la localidad consiste en maximizar el número de computaciones que un procesador realiza con un conjunto de datos que tiene asignado, manteniendo en un mínimo el número de interacciones inter-procesador. Este aspecto cobra especial interés en maquinas con memoria y/o caches locales al procesador[6].

Las distintas técnicas de programación proporcionan distintos medios para controlar estos aspectos (balanceo, *overhead*, localidad, ...) con distintos costes de desarrollo. El compromiso aparece entre estos dos parámetros, ya que el paradigma de pase de mensajes proporciona un mayor control pese a requerir un gran esfuerzo por parte del programador de la aplicación paralela. Por contra, el paradigma de paralelización automática delega la tarea de generar el código eficiente en un compilador. Actualmente, esta aproximación consigue menores prestaciones, pero a cambio, presenta un coste de desarrollo mínimo. El paradigma de paralelismo de datos representa una solución de compromiso intermedia, donde tanto el programador como un compilador ponen algo de su parte para resolver el problema de la generación del código paralelo, típicamente a nivel de bucle, eficiente y en poco tiempo. La programación en memoria compartida es en cierto modo una simplificación del paradigma de paralelismo de datos. Presenta un esquema sencillo con espacio único de direcciones, pero con problemas para conseguir buenos resultados ya que por lo general el programador se suele despreocupar de explotar la localidad. Aparte aparecen otros problemas, como exclusión mutua, coherencia cache y *false sharing*, contención de memoria, etc, que discutimos en la subsección 2.2.4.

Para cuantificar las prestaciones del algoritmo paralelo se utilizan diferentes medidas. La **aceleración** representa el tiempo de ejecución del algoritmo secuencial dividido entre el tiempo de ejecución del algoritmo paralelo. Generalmente, el valor máximo de esta medida coincide con el número de procesadores implicados en la resolución paralela, aunque excepcionalmente puede tomar un valor mayor[7]. La **eficiencia** se define como el cociente entre la aceleración y el número de procesadores que han colaborado en resolver el problema en paralelo. Salvo en los casos excepcionales mencionados, su valor óptimo es uno. Se dice que un algoritmo es **escalable** si al aumentar el tamaño de problema linealmente con el número de procesadores la eficiencia se mantiene constante. Una medida de la escalabilidad es la **isoeficiencia** [89] que relaciona el tamaño del problema con el número de procesadores para mantener una eficiencia constante.

Una vez aclarados los parámetros que nos permiten cuantificar la bondad de nues-

[6]Lo cual es típico en los procesadores actuales que al menos llevan un nivel de cache en el propio chip (on-chip).

[7]Este efecto, conocido con el nombre de super-aceleración o super-linearidad, se justifica en muchas situaciones debido a que al aumentar el número de procesadores aumenta el tamaño total de la cache usada, ya que cada procesador suele disponer de una cache local.

tros algoritmos, dedicamos un apartado a cada una de las técnicas contempladas en esta memoria. Adicionalmente, dirigimos una última subsección a introducir algunas librerías matemáticas que permiten optimizar códigos numéricos en computadores con jerarquía de memoria.

2.2.1 Interfaces de pases de mensajes: PVM, MPI y SHMEM

El paradigma del pase de mensajes, donde un número de procesadores se comunican mediante el envío de mensajes entre ellos, ha sido extensivamente utilizado para programar sistemas multiprocesador. Una de las razones para ello, es que prácticamente, la totalidad de las plataformas paralelas pueden soportar el pase de mensajes. Los programas escritos bajo este modelo pueden ejecutarse en multiprocesadores, típicamente MIMD, con memoria compartida o distribuida (principalmente esta última), redes de estaciones de trabajo, o combinación de ambas.

La gran flexibilidad y el alto grado de libertad en el control del programa pueden ser considerados como un punto a favor del paradigma de pase de mensajes. Sin embargo, esa misma característica presenta otro filo: el programador tiene la responsabilidad absoluta de mantener todos los procesadores ocupados, reducir el coste asociado a las comunicaciones, y evidentemente, debe asumir el elevado tiempo de desarrollo y depuración (por ejemplo, evitando interbloqueos, solapando comunicaciones y computaciones, etc), que conlleva este paradigma.

Aunque existen algunas variaciones, los conceptos básicos sobre procesos comunicantes a través de mensajes están bien asentados. En los últimos 10 años, se han conseguido sustanciales progresos en el desarrollo de aplicaciones bajo este paradigma. Más recientemente, varios sistemas han demostrado que la programación paralela mediante pase de mensajes puede ser implementada de forma eficiente, e incluso puede que más importante, de forma portable, ya que es difícil encontrar una plataforma paralela que no soporte alguno o varios de los interfaces de pase de mensajes. Los más populares parecen ser, **PVM** (Parallel Virtual Machine) [70] y **MPI** (Message Passing Interface) [105] el cual se está convirtiendo en el estándar en este tipo de interfaces. De hecho, los avances y mejoras de MPI se han materializado en el conocido como MPI-2[106], el cual supera con creces las prestaciones y posibilidades del PVM.

Las características comunes de los dos interfaces son:

- *Software* de dominio público, diseñado, especificado e implementado por grupos independientes de las empresas de plataformas paralelas.
- Definen funciones portables de alto nivel en C o Fortran 77 para el intercambio de información entre grupos de procesadores (`send` y `recv`), realizándose las conversiones necesarias en caso de operar sobre máquinas heterogéneas.

Sin embargo existen diferencias importantes que constatan la superioridad de MPI:

- Las implementaciones de MPI suelen ser más eficiente que las de PVM (que fue inicialmente pensado para trabajar sobre *clusters* de estaciones de trabajo).

- MPI soporta el concepto de topología entre los procesos comunicantes, disminuyendo la penalización por comunicaciones cuando una topología virtual se adapta a la topología de la red de interconexión física del multiprocesador.

- El soporte de grupos, comunicadores, tipos de datos definidos por el usuario, mayor variedad de comunicaciones síncronas y asíncronas, comunicaciones globales (*gather*, *scatter*, comunicaciones todos a todos, reducciones), y manejo eficiente de las situaciones de error, etc, proporciona a MPI una versatilidad, potencia y facilidad de uso superior a la de PVM.

Estos dos interfaces para el paso de mensajes están disponibles tanto en el Cray T3D como en el T3E. En este caso, tanto las rutinas de PVM como las de MPI, están construidas mediante llamadas a procedimientos de una librería de más bajo nivel. Esta librería, con el nombre **SHMEM** [25], está diseñada sobre el principio básico de sacrificar la facilidad de uso en aras de una mayor velocidad y flexibilidad. Por tanto, se pueden conseguir mayor ancho de banda y menor latencia utilizando directamente las rutinas de esta librería, pagando esas ventajas con un tiempo adicional de desarrollo de los códigos paralelos.

En esta librería, disponible para C y Fortran 77, sólo se han implementado un número pequeño de operaciones, pero útiles para la gran mayoría de programadores de aplicaciones. Las rutinas básicas de la librería SHMEM se clasifican en:

- **Individuales**, es decir, son llamadas por un único procesador, independientemente de cuantos procesadores estén involucrados en la operación. Estas rutinas utilizan las capacidades de memoria compartida del Cray para soportar transferencia de datos por medio de un protocolo unilateral (*single-party protocol*). Las dos rutinas principales son `shmem_put` y `shmem_get` que escriben y leen, respectivamente, de memorias remotas, sin ningún tipo de intervención por parte del procesador remoto. Esto tiene las siguientes consecuencias:

  1. Es necesaria alguna sincronización vía barrera (`shmem_barrier()`) o *handshake* (`shmem_wait()`) para evitar inconsistencias (por ejemplo, en el caso del `put`, que el receptor lea el dato antes de que esté escrito, o que no "se entere" de cuando está el dato disponible).
  2. En el Cray T3D no existe el *hardware* de soporte adecuado para mantener la coherencia cache, por lo que es necesario que se ocupe de ello el programador. Para ello puede utilizar rutinas como `shmem_set_cache_inv()` y `shmem_clear_cache_inv()` (habilitan y deshabilitan la invalidación automática de las líneas de cache necesarias), o `shmem_udcflush()` (invalida todas las líneas de la cache de datos).
  3. Las direcciones de lectura y escritura pasadas como parámetro a las rutinas `put` y `get` referencian posiciones locales, y por tanto el procesador que hace la llamada debe conocer la dirección donde va a escribir el dato (put) o de donde lo va a leer (get).

- **Colectivas** o globales, cuando todos los procesadores involucrados en la operación deben llamar síncronamente a la rutina. Estas rutinas permiten realizar

operaciones de reducción, barreras, radiaciones y colecciones, y se ejecutan en un conjunto específico de procesadores que puede ser seleccionado mediante los parámetros de la rutina.

Una comparación del rendimiento de estos tres interfaces para el pase de mensajes se incluye en la tabla 2.1, caracterizando cada uno de ellos usando los parámetros de Hockney y Carmona [88]:

- r_∞, la velocidad de transferencia máxima en MBytes/s (equivale al ancho de banda).
- $n_{\frac{1}{2}}$, la longitud del mensaje necesaria para alcanzar la mitad de la velocidad especificada por r_∞.

Interfaz	Protocolo	r_∞	$n_{\frac{1}{2}}$	Latencia (μs)
PVM	shmem	25.7	3357	138
MPI (EPCC)	shmem	50.4	2662	36.4
`shmem_get`	shmem	58	362	6
`shmem_put`	shmem	120.2	780	6

Tabla 2.1: Comparación entre los sistemas de pase de mensajes.

Es claro que las rutinas SHMEM consiguen un mayor ancho de banda y una menor latencia. Y en particular la rutina `shmem_put` tiene el doble del ancho de banda que `shmem_get` debido a que en esta última rutina, por cada palabra remota que se lee, es necesario mandar su dirección y luego recibir la palabra correspondiente.

Es importante matizar que algunos investigadores distingue el paradigma de programación, que a veces llaman `put-get`, del de pase de mensajes. Otros, sin embargo lo consideran un caso particular, denominándolo paradigma de `mensajes unilaterales`. Aunque no queremos entrar en discusiones acerca de la terminología correcta, si es admitido por todos que esta aproximación evita el *rendez-vous* de dos procesadores, cuando en principio es suficiente con implicar a uno de ellos. Esto, en cierto modo, simplifica el desarrollo de códigos paralelos, ya que el programador no tiene por qué aparear cada `send` con su correspondiente `recv`, fuente de interbloqueos y problemas en la planificación de comunicaciones.

En el capítulo 3 los códigos paralelos presentados se apoyan en el modelo de pase de mensajes, donde los tres interfaces, PVM, MPI y SHMEM han sido utilizados.

2.2.2 Lenguajes de paralelismo de datos: CRAFT y HPF

La idea que subyace bajo el paradigma de paralelismo de datos, claramente heredada de las máquinas SIMD, es soportar completamente las operaciones sobre arrays en paralelo. En general, la distribución de los datos y las computaciones es una tarea

que implementará el compilador, guiado por las indicaciones que le proporciona el programador mediante ciertas directivas o anotaciones.

Aunque se pierde control sobre el código y la eficiencia del programa paralelo suele ser inferior a la que se consigue mediante pase de mensajes, realmente el inconveniente más importante es que los compiladores de paralelismo de datos actuales aún no permiten codificar cualquier tipo de problema. En particular los problemas irregulares son considerados como la *bête-noire* de este paradigma, y una de las cuestiones más urgentes a resolver.

Sin embargo, para problemas regulares, donde los códigos están bien estructurados, estas herramientas pueden generar, en tiempo de compilación, eficientes programas paralelos utilizando simples distribuciones de datos y computaciones. En este caso las ventajas de este paradigma son claras: facilidad y simplicidad de las implementaciones, portabilidad de código existente o futuro y mantenimiento de la compatibilidad con otro estándar, el Fortran90 [1, 64].

Actualmente los lenguajes de paralelismo de datos más populares son el CM-Fortran [142], Fortran D [67], Vienna-Fortran [160], y sobre todo el considerado como estándar *High-Performance Fortran* (HPF) [40, 85, 86, 104]. Todos estos compiladores tienen las siguientes características en común:

- Son compiladores fuente-fuente que a partir del programa secuencial con anotaciones (directivas) generan un único código que se puede ejecutar en varios procesadores. Es decir, se apoya en el modelo SPMD (*Single Program Multiple Data*) donde un único programa determina las operaciones a realizar. Sin embargo no existe la sincronización a nivel de instrucción que encontrábamos en las arquitecturas SIMD.

- Espacio de direcciones global, ya que el programador ve una única memoria, dejándose al compilador la tarea de distribuir los datos, acceder a memoria, gestionar las comunicaciones, etc. Las directivas ayudan al compilador a decidir esta distribución y a gestionar adecuadamente las comunicaciones necesarias (ya sea mediante pase de mensajes o memoria compartida).

- Las directivas se indican mediante una palabra clave después de un comentario (por ejemplo, !HPF$), de forma que el código se puede seguir compilando en secuencial. Las directivas del compilador sólo afectarán a las prestaciones del código ejecutable, pero no a su significado.

- Las operaciones sobre estructuras de datos uni- o multi-dimensionales se realizarán en paralelo entre varios procesadores. Son necesarias algunas extensiones del lenguaje (aparte de las directivas) como las que proporciona el Fortran90: la sentencia FORALL para bucles sin dependencias, funciones intrínsecas de reducción incluídas en librerías, etc.

La implementación de HPF soportada por las plataformas MPP de Cray se llama HPF-Craft [46]. Esta implementación es un híbrido entre HPF_LOCAL (entorno

extrínseco definido por HPF1.1 y 1.2), HPF2 [86] y CRAFT (antiguo entorno de paralelismo de datos proporcionado por Cray para el T3D), diseñado por Cray e implementado por PGI (The Portland Group). La implementación actual del compilador consiste en un preprocesador fuente-fuente que genera código Fortran90 (o Fortran77) con llamadas a rutinas de la librería SHMEM.

En el capítulo 4 se describe una aproximación para la representación en HPF-Craft de nuestros algoritmos de factorización dispersos que no dio buenos resultados, así que posteriormente introducimos unas mínimas extensiones del lenguaje HPF que permiten una eficiente implementación del problema.

2.2.3 Paralelización automática: Polaris y PFA

Un compilador no sólo simplifica la tarea de escribir y leer programas, sino que además proporciona portabilidad entre distintas plataformas. Sin embargo estas ventajas no deben suponer una significativa disminución en la eficiencia del código ejecutable. La coexistencia de estos tres aspectos dio lugar a que desapareciese progresivamente la programación en ensamblador, en una evolución hacia el uso intensivo de los lenguajes de alto nivel. Actualmente, la programación a mano de códigos paralelos mediante un modelo de pase de mensajes es el equivalente de lo que en aquellos tiempos era la programación en ensamblador. Sin embargo, los usuarios, esperan que algún día ellos puedan seguir escribiendo su programas en algún lenguaje de alto nivel y que un compilador se encarge de generar el ejecutable para cualquier plataforma independientemente de su arquitectura.

Sin lugar a equívocos, un compilador para una arquitectura paralela presenta una tarea mucho más compleja que para una máquina secuencial. Los primeros, deben afrontar aspectos como la cuantificación y explotación eficiente del paralelismo, sincronización, manejo de la localidad de datos espacial y temporal, así como otros más dependientes de la arquitectura en particular. La gran cantidad de información que será necesario procesar para abordar eficientemente estos aspectos, induce a la interacción entre el compilador, un subsistema en tiempo de ejecución, el sistema operativo e incluso puede que el usuario.

Distintamente a como sucedió con la vectorización automática de programas, totalmente entendida y superada [98], la paralelización automática no llega a madurar aunque se encuentra en una etapa de rápido desarrollo. Los dos ejemplos más significativos de herramientas de paralelización los encontramos avalados por sendos proyectos en la Universidad de Illinois: Parafrase-2 [75, 113] y Polaris [31, 32]. Ambos, reestructuran código secuencial en Fortran (y Parafrase-2 también en C), para convertirlos en sus versiones paralelas, proporcionando un interfaz adecuado para que el usuario pueda interaccionar en las distintas fases del proceso de transformación. Mientras Parafrase-2 no ha resultado totalmente operativo con códigos reales, Polaris proporciona códigos paralelos competitivamente eficientes no sólo para plataformas de memoria compartida, sino también compartida-distribuida como el Cray T3D [108]. Otro compilador representativo es SUIF [84] (*Stanford University Intermediate Format*). Este compilador tranforma códigos secuenciales en F77 y C en sus versiones paralelas SPMD para memoria compartida. El código generado contiene llamadas a una librería en tiempo

de ejecución disponible actualmente para las plataformas de SGI y el multiprocesador DASH.

Un compilador paralelo comercial con el cual se compara Polaris en las referencias anteriores es PFA [137, 136] de KAI. En sus versiones para Fortran y C permite generar código paralelos para sus plataformas multiprocesador (Power Challenge, Origin 2000, etc), aunque suele necesitar un posterior ajuste por parte del usuario, para el cual se dispone de un potente interfaz gráfico.

En el capítulo 5 discutimos en mayor profundidad estas utilidades y abordamos el problema de paralelización automática de códigos dispersos aportándose nuevas técnicas. Nos centraremos principalmente en el compilador Polaris al que hemos tenido acceso. Esta elección se justifica en los excelentes resultados que Polaris ha proporcionado con un grán número de códigos reales, no sólo para plataformas de memoria compartida, sino también a través de rutinas de comunicación unilateral (`put` y `get`). El reestructurador de Polaris incluye varias técnicas tanto estáticas como dinámicas (en tiempo de ejecución), a saber: análisis de dependencias (en tiempo de compilación y/o de ejecución), análisis simbólico, detección de variables de inducción y de reducción, análisis interprocedural, privatización de arrays, etc.

2.2.4 Modelo de programación de memoria compartida

Un importante sector de los programadores de aplicaciones prefieren escribir sus algoritmos bajo el paradigma de programación de memoria compartida. La razón de esta opcción estriba principalmente en la facilidad de programación. Esta mayor sencillez en la programación es también una de las razones para que muchos de los paralelizadores automáticos generen código según este paradigma de programación (PFA, Polaris y SUIF, entre otros).

A cambio, por lo general, las aplicaciones desarrolladas según este esquema suelen reportar peores prestaciones que las basadas por ejemplo en pase de mensajes. Algunos arguyen que la razón pricicipal de estos resultados está en el menor control del programador sobre la localidad de los datos. Esto conduce a pérdidas de rendimiento debidas a contención de memoria, acceso a memorias no locales, coste temporal debido al soporte de coherencia cache como el *false sharing*[8], etc.

Básicamente, este modelo de programación se basa en la inclusión de ciertas directivas en el programa paralelo, en lugar de extender la sintaxis del lenguaje (normalmente Fortran 77 o 90). La directiva más importante es el `DOACROSS` o `DOALL` seguida de determinadas cláusulas. Su uso fuerza al compilador a generar un código paralelo que ejecuta las iteraciones del bucle al que acompaña sobre distintos procesadores. Por ejemplo, en las plataformas de SGI la directiva es `C$ DOACROSS` y las cláusulas pueden ser `IF`, para controlar el efecto de la directiva en tiempo de ejecución, `LOCAL`, para indicar las variables locales a cada procesador, `SHARE`, para indicar las variables compartidas, y `REDUCTION` para indicar las variables que se reducen, entre otras [135].

Otras directivas suelen ajustarse a las propuestas por el Foro de Computación

[8]Problema que aparece cuando dos procesadores acceden en paralelo a celdas de memoria correspondientes a una misma línea cache y alguno de ellos la modifica.

Paralela (PCF), cuyo modelo es la base para el estándard propuesto ANSI–X3H5. Estas directivas permiten identificar secciones de código paralelas (aparte de los bucles), secciones críticas, barreras de sincronización, etc. Sin embargo el paralelismo de bucle es mucho más utilizado y generalmente suficiente.

La condición esencial requerida para paralelizar un bucle correctamente es que cada iteración sea independiente de las demás. Formalmente se dice que no deben existir dependencias entre iteraciones o dependencias generadas por bucle (*loop-carried dependences*). Si el bucle cumple esta condición las iteraciónes se pueden ejecutar en cualquier orden o al mismo tiempo y el resultado sigue siendo el mismo. Existen tres tipos de dependencias: verdaderas o de flujo (también llamada de lectura despues de escritura); anti-dependencias (también de escritura después de lectura); y de salida (o de escritura después de escritura). Las del primer tipo, cuando aparecen, son inevitables ya que representan el flujo de los datos a través de los distintos pasos en la ejecución del programa. Sin embargo las anti-dependencias y las de salida son dependencias "falsas" causadas únicamente por el reuso de posiciones de memoria. Estos dos tipos de dependencias relacionadas con la memoria pueden ser eliminadas utilizando diferentes posiciones de memoria para distintos accesos. Eliminar estas dependencias permite explotar un mayor grado de paralelismo y aumentar así las prestaciones del código paralelo. La privatización es una técnica que permite romper estas falsas dependencias en máquinas de memoria compartida. Cada procesador realiza una copia local de la variable en conflicto y accede únicamente a su copia local.

Este modelo de programación será discutido más extensivamente a lo largo del capítulo 5 ya que el paralelizador automático Polaris genera código paralelo siguiendo este paradigma.

2.2.5 Librerías: BLAS, LAPACK y HSL

Hasta ahora hemos revisado utilidades y modelos de programación que permiten generar códigos paralelos más o menos eficientes con mayor o menor sencillez. Pero en cualquier caso, al final, ya sea uno o varios procesadores ejecutarán algún código donde es muy importante evitar referencias innecesarias a memoria. Los procesadores modernos basan gran parte de su eficiencia en una adecuada gestion de la jerarquía de memoria, que debe ser tenida en cuenta también por el programador para evitar que las prestaciones del algoritmo se vean limitadas por el tráfico de información con memoria. Esto es así debido a que el movimiento de datos en memoria puede ser, en los procesadores actuales, más costoso que las operaciones aritméticas. Reducir este coste es una motivación suficiente como para reorganizar los algoritmos existentes de forma que se minimicen los movimientos de datos.

Con esa intención aparece a principios de los 70's una librería orientada a la solución eficiente de problemas del álgebra lineal, llamada BLAS (de *Basic Linear Algebra Subprograms*) [99]. BLAS es un interfaz que especifica el nombre y la secuencia de argumentos de subrutinas FORTRAN que realizan operaciones básicas y frecuentes del algebra lineal. Existen tres niveles o conjuntos de rutinas BLAS. En el nivel uno las rutinas tienen al menos un vector como parámetro, por ejemplo la rutina SAXPY realiza la operación $y = y + \alpha x$ donde x e y son vectores y α es un escalar. Esta rutina

tiene una relación operaciones de memoria – operaciones punto flotante de 3/2. Sin embargo el nivel dos de BLAS incluye operaciones matriz-vector como SGEMV que realiza la operación $y = \alpha Ax + \beta y$, donde A es una matriz. En este caso la relación anterior es de 1/2, es decir se realizan el doble de operaciones en punto flotante que de movimiento de datos. Por último el nivel 3 trata las operaciones matriz-matriz, como SGEMM que realiza la operación $C = \beta Cx + \alpha AB$, donde A, B y C son matrices. Estas operaciones reportan la mejor relación entre operaciones de memoria y FLOPS igual a $2/n$ donde n es la dimensión de las matrices [54]. Por ejemplo, las prestaciones en MFLOPS alcanzadas por estas rutinas en el Cray T3D, que alcanza 150 MFLOPS de pico, son las siguientes[5]:

Nivel BLAS:	1 (DAXPY)	2 (DGEMV)	3 (DGEMM)
MFLOPS:	29.3	54.7	107.9

Dado que estas rutinas proporcionan eficiencia, portabilidad y legibilidad de los códigos han sido muy utilizadas incluso para crear librerías matemáticas más completas. Por ejemplo la librería LAPACK (de *Linear Algebra Package*) [11] proporciona rutinas para resolver sistemas de ecuaciones lineales, mínimos cuadrados, autovalores, etc. Esta librería se creó a partir de otras dos: EISPACK [139] y LINPACK [53] pero prestando un mayor interes en explotar la localidad en cache. Aparte de estas rutinas secuenciales se están desarrollando las correspondientes versiones paralelas para plataformas de memoria distribuida y paradigma de pase de mensajes mediante PVM o MPI. Estas rutinas paralelas se agrupan dentro del paquete *ScaLAPACK* [30] apoyado en PBLAS (*Parallel BLAS*) y las rutinas BLACS (*Basic Linear Algebra Communications Subprogram*) para simplificar la gestión de las comunicaciones.

Sin embargo LAPACK está orientada a matrices densas o en banda, pero no proporciona rutinas para matrices dispersas. De esto se encargan otro paquetes como HSL (*Harwell Subroutine Library*) [141] o SPARSPACK [72] ambas proporcionan algoritmos dispersos secuenciales para resolver sistemas lineales, mínimos cuadrados, autovalores, etc. Para operaciones más sencillas se está completando la versión dispersa de las rutinas BLAS: SBLAS [59]. Dadas su eficiencia y portabilidad, las operaciones disponibles en las librerías BLAS y SBLAS han sido extensivamente usadas en nuestros algoritmos.

En el caso particular de algoritmos dispersos para la resolución de sistemas, otros paquetes con mantenimiento *software* son Y12M[163], métodos iterativos en YSMP, desarrollado en la Univ. de Yale [62], en las rutinas IMSL o en las ESSL de IBM. Evidentmente, existen más códigos secuenciales en desarrollo o como herramientas de investigación (como el paquete SMMS desarrollado en Wisconsin [6]). También existe un conjunto de librería para la resolución paralela de sistemas dispersos por métodos iterativos llamado *BlockSolve95* [92] o las rutinas CAPSS [122] incluidas en ScaLAPACK para la factorización Cholesky paralela de matrices simétricas definidas positivas. Pero hasta donde conocemos, no existe ningún algoritmo paralelo comercial para la factorización LU de matrices dispersas genéricas. En el capítulo 3 resumimos los trabajos y el estado del arte en la paralelización del problema de la factorización de matrices dispersas.

Capítulo 3

Paralelización de métodos directos

Es indiscutible, que implementar a mano un algoritmo paralelo suele ser laborioso y consume una cantidad significativa de tiempo, tanto en desarrollo como en depuración. Sin embargo, y especialmente con problemas irregulares, este camino lleva, por el momento, a implementaciones más eficientes que las que resultan de aplicar herramientas de paralelización automática o semi-automática. Dedicamos este capítulo al problema de la paralelización manual de algoritmos para la resolución de sistemas de ecuaciones lineales dispersos. Como ya avanzamos en el capítulo 1 nos centraremos en esta memoria en los métodos directos con especial interés en las técnicas genéricas de factorización LU.

Nos centraremos principalmente en las etapas de análisis y factorización y en su implementación paralela, discutiendo la etapa de resolución triangular en la penúltima sección. Como se justifica en la primera sección, la etapa de reordenación no resulta relevante actualmente y, dado el ínfimo tiempo que consume, incluso menos interesante su paralelización.

La organización del capítulo se esboza a continuación. En la primera sección se discuten la posibles fuentes de paralelismo que pueden ser explotadas en los algoritmos de factorización dispersos. Pero además, simultáneamente se van presentando los trabajos, relacionados con el nuestro, que han aparecido recientemente, comprobándose la gran actividad de este área de investigación. En la misma línea, la sección 3.2 aborda el estudio de los distintos tipos de distribuciones de datos aplicables en nuestro algoritmo, comparándolas entre ellas. Seguidamente, la sección 3.3 describe el proceso de paralelización de un código *right-looking* de actualizaciones de rango m que utiliza una lista bidimensional doblemente enlazada para representar la matriz dispersa. En la sección 3.4, se discute otra alternativa: la paralelización de un código secuencial *left-looking* muy optimizado como es el de la rutina MA48 de las Harwell Subroutine Library. Estas dos secciones previas se centran por tanto en la paralelización de las etapas de análisis y factorización, dejando para la sección 3.5 la paralelización de la etapa de resolución triangular. Finalmente, evaluamos experimentalmente las prestaciones de todos estos algoritmos en la sección 3.6.

3.1 Fuentes de paralelismo y trabajos relacionados

Un análisis detallado del código encargado de la factorización de matrices dispersas nos revela las tres posibles fuentes de paralelismo que podemos explotar:

1. Paralelismo a nivel de tarea: Reordenado en bloques independientes.
2. Paralelismo a nivel de bucle: Bucles sin dependencias.
3. Paralelismo a consecuencia de la estructura dispersa de la matriz: Actualizaciones de rango m.

Dedicaremos una breve subsección a cada una de ellas, incluyendo al final de éstas, una mención de trabajos recientes, relacionados con esa línea y de reconocido interés.

3.1.1 Reordenado en bloques independientes

Las matrices dispersas pueden ser reordenadas en muchas ocasiones de forma que adopten determinados patrones. Por ejemplo en la figura 3.1 encontramos distintos patrones útiles.

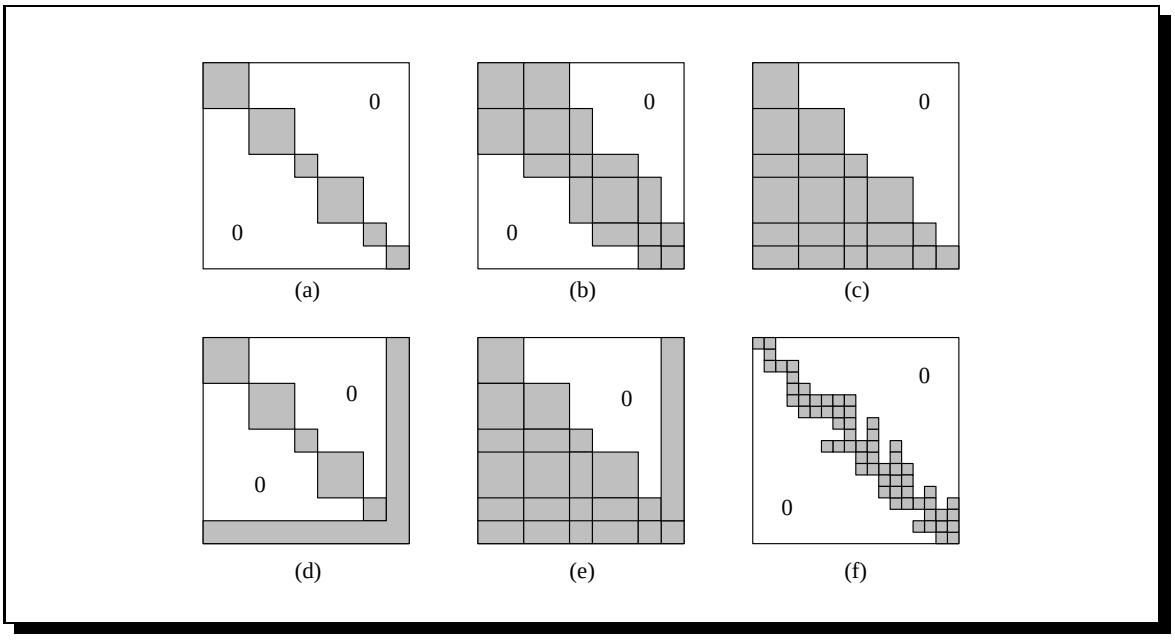


Figura 3.1: Patrones que podemos conseguir para una matriz dispersa A.

La figura 3.1 (a) muestra una matriz diagonal por bloques donde los elementos no nulos están concentrados en bloques de densidad mucho mayor que la que presenta la matriz completa. Igualmente podemos reordenar una matriz dispersa para que tome una forma tridiagonal por bloques como la mostrada en la figura 3.1 (b). Si resulta imposible mover todos los elementos no nulos a alguna posición cercana a la diagonal nos tendremos que conformar con reordenar la matriz en una triangular por bloques

como la de la figura 3.1 (c). En las figuras (d) y (e) tenemos las versiones con borde (*bordered*) de la diagonal y la triangular por bloques respectivamente.

Por último en la figura 3.1 (f) se presenta un patrón en banda variable[1]. En este caso, para cada fila almacenamos todos los coeficientes entre la primera entrada no nula de la fila y la diagonal, y para cada columna, de forma análoga, almacenamos todos los elementos entre la primera entrada no nula de la columna y la diagonal. Al número total de coeficientes almacenados se le llama perfil (o *profile*). Es fácil ver que durante la factorización de esta matriz no aumentará el perfil y que se preservará la forma de la banda variable. De esta forma se simplifica sustancialmente el algoritmo de factorización.

Por supuesto, el resto de los patrones también presentan sus ventajas. Por ejemplo si reordenamos nuestra matriz A de forma que presente un patrón tipo diagonal por bloques, habremos conseguido por un lado, que el llenado quede confinado dentro de cada bloque (disminuyendo probablemente el llenado de la matriz LU), y por otro lado, también muy importante, habremos desacoplado el procesado de cada bloque del resto de ellos. Es decir la factorización de cada bloque será ahora una tarea independiente que no interacciona en ningún momento con la factorización del resto de los bloques. De esta forma, cada bloque podría ser factorizado en paralelo por un procesador independiente. Aunque el procesado que se realice fuese secuencial, factorizando un bloque tras otro, habríamos conseguido un sustancial ahorro de tiempo respecto a la factorización de la matriz no reordenada. A esta conclusión llegamos fácilmente si tenemos en cuenta la dependencia de la complejidad del algoritmo de factorización con la dimensión, n, de la matriz a factorizar. Además, la etapa de resolución triangular también es totalmente independiente para cada bloque, de forma que se puede realizar también en paralelo. En el apéndice A se discute la solución matemática para desacoplar los bloques en el resto de los posibles patrones de la matriz. La conclusión final resulta en que es suficiente con factorizar cada uno de los bloques de la diagonal en paralelo aunque la etapa de resolución triangular debe ser secuencial en la mayor parte de los casos.

Como vemos, las ventajas de conseguir reordenar una matriz dispersa en una de estas formas están claras. Durante la década pasada, la etapa de reordenación sólo era usada extensivamente con este propósito de organizar la matriz en su forma triangular por bloques y factorizar los bloques de la diagonal con algoritmos densos. Sin embargo, aunque para algunas matrices (por ejemplo de ingeniería química) tiene sentido, para la gran mayoría restante, la dimensión del bloque irreducible más grande conseguido es cercana al orden de la matriz. En el caso peor en el que la reordenación sólo genera un único bloque se dice que la matriz es irreducible. La elevada probabilidad de encontrar matrices irreducibles o cercanas a esa condición ha dado lugar a que actualmente la etapa de reordenación caiga en desuso. En nuestro trabajo supondremos, sin pérdida de generalidad, que los sistemas a resolver son ya irreducibles.

De cualquier forma, el paralelismo que podemos conseguir mediante estas reordenaciones será a nivel de tarea. Es decir, podemos asignar a cada procesador la tarea de factorizar cada uno de los bloques diagonales de forma totalmente paralela sin interaccionar para nada en el proceso. Los problemas que este hecho plantea, son los propios de cualquier algoritmo paralelo a este nivel:

[1] También conocido como *skyline*, perfil (*profile*) o envolvente (*envelope*).

- Desbalanceo: es decir, los tamaños de cada bloque serán dispares y dependerán de cada matriz en particular, de forma que habrá procesadores que tengan más trabajo que realizar que otros. Incluso en el caso de tener muchos bloques que factorizar y utilizando una distribución dinámica de los bloques el rendimiento se verá degradado cuando tengamos bloques de tamaños muy diferentes y algunos de gran tamaño.

- Escalabilidad: probablemente sea un problema de mayor envergadura que el anterior, ya que en muchos casos es difícil encontrar un número suficiente de bloques en la matriz reordenada. Eso quiere decir que la técnica sería inviable para ser usada con un gran número de procesadores.

Todo eso no quita, que podamos combinar esta técnica con las demás que se comentan a continuación, en la que el paralelismo es a nivel de bucle. De esta forma podemos asignar bloques a un número pequeño de grupos de procesadores que trabajarán independientemente.

A pesar de todo, salvo en el caso de las matrices diagonales por bloques, el paralelismo que conseguimos aquí sólo puede ser aplicado para las etapas de análisis y factorización que se ejecutarán para cada bloque diagonal. La etapa de resolución triangular deberá ejecutarse de forma secuencial a no ser que se implemente una costosa redistribución global de la matriz que nos permita explotar un paralelismo de bucle para esta etapa. Este inconveniente no tendría demasiada importancia si la etapa de resolución triangular se va a ejecutar sólo una vez. Pero en muchas situaciones (como problemas de optimización, por ejemplo), la etapa de resolución triangular se realiza varias veces para una misma matriz previamente factorizada, de forma que su ejecución secuencial implicaría una gran pérdida de eficiencia para el algoritmo global.

Trabajos recientes en esta dirección se resumen a continuación. Arioli y Duff (1990) [13] intentaron extender las ideas de triangularización usando técnicas de partición que resultasen en matrices triangulares por bloques con banda. Encontraron que normalmente el número de columnas en la banda era demasiado alto, de forma que el número de operaciones aritméticas resultaba mayor que usando un algoritmo disperso estándar para el sistema completo. Pothen y Fan (1990) [116] desarrollaron una técnica de partición que generaliza la forma triangular por bloques para sistemas rectangulares, lo cual fue usado por Amestoy y col. (1996) [10] para la factorización QR. Gallivan, Marsolf y Wijshoff (1996) [69] utilizan una descomposición en grano grueso del problema, obteniendo una matriz triangular por bloques con borde. Similarmente, Zlatev y col. (1995) [162] han desarrollado un paquete, PARASPAR, para la resolución paralela del sistema, en multiprocesadores de memoria compartida, mediante una reordenación de la matriz original. Permiten que los bloques de la diagonal sean rectangulares para explotar algunas ventajas, pero la partición no garantiza que estos bloques estén bien condicionados o incluso que no sean singulares. Ha sido necesario por tanto, un posterior desarrollo de métodos para mantener la estabilidad. Estos métodos se implementan en el código LORA-P[5] encargado de la fase de reordenación del programa Y12M3. Este código consigue una aceleración entre 3.0 y 4.7 para algunas de las matrices mayores del conjunto Harwell-Boeing, factorizadas con 8 procesadores en el Alliant FX/80 de memoria compartida [156].

Otra forma de explotar paralelismo se insinuó con anterioridad en la sección 1.1.2 al tratar los métodos multifrontales. En efecto, el árbol de eliminación asociado a estos métodos es claramente una herramienta tanto para visualizar como para implementar paralelismo en la factorización, ya que las hojas del árbol son independientes entre sí. Dado que tradicionalmente estos métodos se han aplicado casi exclusivamente a matrices simétricas, las versiones paralelas que existen son del algoritmo Cholesky multifrontal. En máquinas de memoria compartida los ejemplos más recientes son el de Johnson y Davis (1992) [91] y el de Amestoy y Duff (1993) [9]. También en memoria compartida pero ahora bajo un modelo de paralelismo de datos encontramos los trabajos de Conroy, Kratzer y Lucas (1994) [44] para la plataforma MasPar MP-2, y también para la misma máquina Manne y Hafsteinsson (1995) [101] presentan otro algoritmo Cholesky multifrontal. El mismo problema, implementado ahora sobre arquitecturas de memoria compartida, fue abordado sin demasiado éxito en [118, 129], hasta que Gupta, Karypis y Kumar (1995) [83] han resuelto en el trabajo con, posiblemente, mejores prestaciones para la implementación en memoria distribuida de la factorización Cholesky dispersa. Ha sido decisivo para ello la combinación de un especial esmero en el estudio del balanceo del árbol y una distribución regular de las matrices densas para explotar paralelismo a nivel de bucle cuando se agota el de nivel de tarea al acercarse a la raíz del árbol.

Sin embargo la revolucionaria evolución de códigos multifrontales paralelos para matrices simétricas no ha tenido, ni mucho menos, lugar para sistemas no simétricos. La aproximación a este problema se apoya en el árbol de supernodos que puede ser utilizado para explotar paralelismo a nivel de tarea, también ahora sobre matrices no simétricas. Una versión paralela para memoria compartida de este algoritmo se presenta en [100], obteniendo, para 21 matrices dispersas, una aceleración media de 3.59 en una SGI Power Challenge, 4.01 en la DEC AlphaServer 8400, 3.85 en el Cray C90 y 4.29 en el Cray J90, ejecutando siempre en 8 procesadores. Sin embargo no ha sido hasta muy recientemente cuando Fu y Yang (1996) [68] han desarrollado un versión paralela para memoria distribuida del algoritmo secuencial SuperLU discutido en la sección 1.1.2, obteniendo buenos resultados para matrices no demasiado dispersas.

3.1.2 Bucles sin dependencias

Explotar el paralelismo de grano medio o a nivel de bucle, implica para nuestro problema, repartir apropiadamente los datos entre los procesadores y ejecutar posteriormente un código en cada uno de ellos, que tiene la misma estructura que el código secuencial salvo por dos particularidades:

- El espacio de iteraciones de los bucles se verá reducido de forma que cada procesador trabaje sólo con el subconjunto de los datos que le ha correspondido.

- Para resolver las pequeñas dependencias de datos existentes, se incluyen en el código, ciertas etapas de comunicación entre los procesadores. De esta forma, el procesador contendrá en su memoria local aquella información que necesite antes del procesado.

De esta forma, el código paralelo no es más que una generalización del secuencial. Es decir, si ejecutamos el código paralelo en un sólo nodo, éste procesará todos los datos y no se ejecutarán las etapas de comunicación, con lo que estaremos particularizando la ejecución al caso secuencial.

Si queremos tener éxito programando a este nivel debemos minimizar todo el *overhead* paralelo generado. Las fuentes de *overhead* más importantes son debidas a las etapas de comunicaciones, el desbalanceo de la carga y la ejecución de código para manejo de índices y cotas de bucles, que no aparecen en el algoritmo secuencial.

Frente al paralelismo a nivel de tarea, el paralelismo de grano medio ofrece una mayor escalabilidad y, mediante una buena distribución de datos, permite conseguir balanceos de carga cercanos al óptimo. A cambio se incrementa el consumo de tiempo debido al manejo de índices, aunque éste suele ser despreciable.

Pues bien, en nuestro algoritmo secuencial de factorización LU encontramos tres bucles como veíamos en la sección 1.3. Ya sea en la variante *right-looking* o *left-looking*, el bucle más externo es inherentemente secuencial como se deduce rápidamente del algoritmo secuencial[2]. Sin embargo, continuando con un entorno orientado a columnas, el bucle de índice `i` implícito en las operaciones `divcol(k)` y `actcol(j,k)` es totalmente paralelo (recordemos que en el primero dividimos una columna por el pivot, y el segundo es equivalente a una operación SAXPY).

En la versión *right-looking*, como vemos en el ejemplo 1.7, la operación `actcol(j,k)` está anidada en un bucle de índice `j` de forma que se lleva a cabo la actualización de toda la submatriz reducida. En este caso tenemos realmente dos bucles anidados que van recorriendo las columnas de la submatriz reducida actualizando sus valores. Estos dos bucles son paralelos ya que escriben en posiciones de la matriz reducida $(i, j > k)$, mientras se lee de posiciones distintas: la primera fila y columna de la submatriz activa, $(i = k \wedge j > k)$ y $(j = k \wedge i > k)$; de forma que no hay dependencia posible. Esto permite paralelizar los dos bucles sobre una topología bidimensional de procesadores, por ejemplo una malla, o cualquier otra con mayor conectividad (toro o hipercubo por ejemplo).

Por contra, en la versión *left-looking*, la operación `actcol(j,k)` está anidada en un bucle de índice `k` mientras el bucle más externo es ahora `j`. En cada iteración de este último, actualizamos la columna `j` con las $j-1$ columnas previamente actualizadas, recorridas por el índice `k`. Los dos bucles anidados ahora son el de índice `k` y el de índice `i` implícito en la operación `actcol(j,k)`. Sin embargo el primero presenta dependencias generadas por bucle, ya que, como se explicó en la sección 1.3.2, coeficientes de la columna `j` escritos en una iteración `k` pueden ser leídos en otra posterior. Esto permite explotar únicamente un paralelismo por columnas para realizar la operación `actcol` concurrentemente. Para ello, partimos las columnas en subcolumnas que asignamos a los procesadores de una malla unidimensional (un array de procesadores). Este nivel de paralelismo es explotado por los códigos densos de factorización LU, como ocurre por ejemplo en la implementación paralela de ScaLAPACK [30]. Zlatev [161] explota únicamente esta fuente de paralelismo en su código Y12M1, primera versión paralela

[2]Existen dependencias generadas por bucle (en inglés *loop-carried dependences*) al existir coeficientes de la matriz A que se escriben en una iteración y se leen en la siguiente.

del anterior Y12M. Para matrices dispersas este nivel de paralelismo se puede combinar con el de tarea, como en el mencionado trabajo de Gupta (1995) [83], o con otra fuente de paralelismo adicional que comentamos bajo el siguiente epígrafe.

3.1.3 Actualizaciones de rango m

La gran cantidad de coeficientes nulos de la matriz y el relativo grado de libertad que tenemos para elegir los pivots nos puede permitir alcanzar un mayor grado de paralelismo. La idea parte de la definición de **pivots compatibles** o independientes observada por Calahan (1973) [37]. Como se muestra en la figura 3.2, dos entradas de la matriz a_{ij} y a_{rs} son compatibles si a_{is} y a_{rj} son cero. Si únicamente imponemos la condición $a_{rj} = 0$ se dice que los pivots son parcialmente compatibles, mientras que son incompatibles en cualquier otro caso.

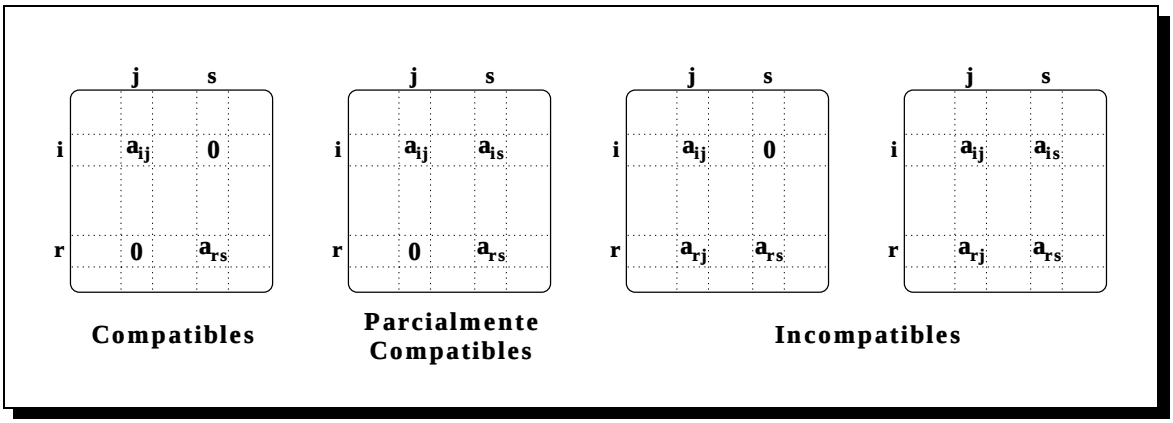


Figura 3.2: Compatibilidad entre coeficientes de la matriz.

El proceso consta entonces de los siguientes pasos: encontrar un conjunto de `m` pivots compatibles; realizar un número máximo de `m` permutaciones de filas y columnas para llevar los mencionados pivots a la diagonal; y por último realizar la actualización en paralelo correspondiente a cada uno de ellos, aplicando lo que se da en llamar **actualización de rango** `m`. La razón es clara: el seleccionar `m` pivots compatibles va a eliminar las dependencias generadas por bucle en esas `m` iteraciones del bucle externo. Ese desacoplo se justifica en que ninguna entrada de las `m` primeras filas y columnas de la submatriz activa va a ser modificada durante la actualización. En la figura 3.3 vemos las tres etapas del algoritmo comentado, usualmente llamado de **pivots paralelos**.

La selección de los posibles coeficientes para ser pivots debe atender de nuevo a criterios de estabilidad numérica y de preservación del coeficiente de dispersión. Por ejemplo, tiene sentido buscar en las columnas con menor $C_j^{(k)}$, entradas que superen un umbral y al mismo tiempo minimicen $R_i^{(k)}$. De entre todos los coeficientes que cumplan esas condiciones, serán elegidos como pivots aquellos que sean compatibles, descartando los demás.

El objetivo de la etapa de permutaciones o pivoteo es llevar los pivots elegidos a la diagonal, construyendo lo que se llama un **bloque diagonal** o submatriz diagonal

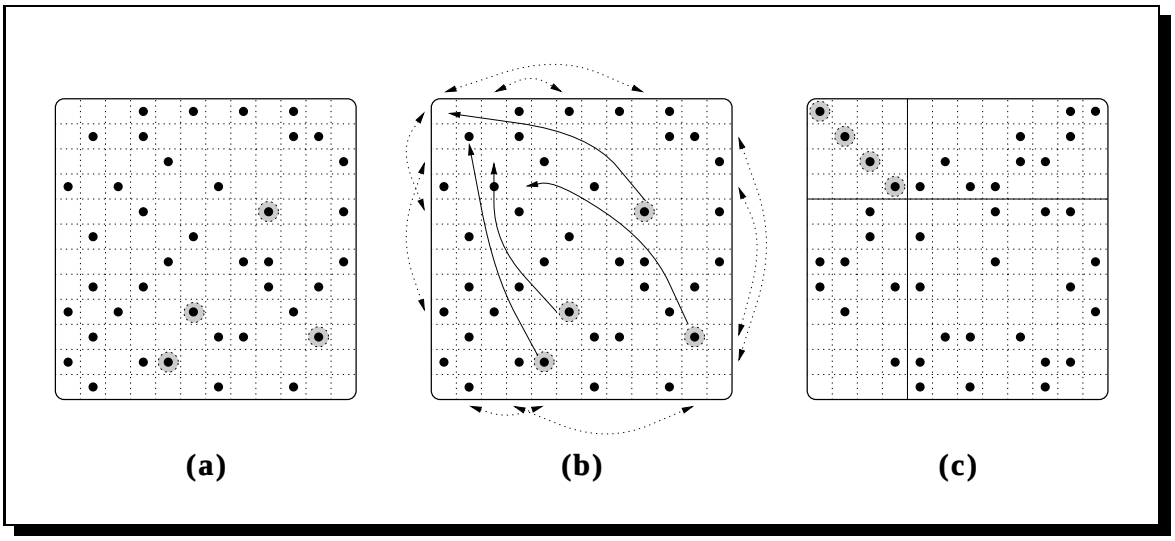


Figura 3.3: Fases del algoritmo de pivots paralelos: (a) Elección de los pivots compatibles; (b) Permutaciones de filas y columnas para llevar los pivots a la diagonal; (c) Actualización de rango `m`.

de dimensión `m`×`m`. Las permutaciones pueden ser **explícitas**, lo que implica mover las entradas de las filas y columnas [154, 43, 71], o **implícitas** [138, 155] si usamos los vectores de permutación, π y γ, para acceder indirectamente a la matriz. En implementaciones paralelas, se ha comprobado en [43, 71], que el pivoteo explícito da lugar a un mayor balanceo de la carga y, que el incremento de la eficiencia debido a este factor, compensa la penalización temporal por comunicaciones de filas y columnas.

Por último en la actualización, las `m` operaciones `divcol(j)`, con $k \leq j < k+$`m`, se pueden realizar totalmente en paralelo. De igual forma, la actualización de la submatriz reducida, ahora definida por $A(k+m:n, k+m:n)$, da lugar a una estructura de tres bucles anidados (en `m`, `j` e `i`) sin ninguna dependencia entre ellos. Dicho de otra forma, la operación `actcol(j,k)` está anidada en dos bucles paralelos, de índices `m` y `j`. Es claro que el bucle externo `k` se incrementa con un paso `m` en cada iteración, consiguiéndose un número de iteraciones muy inferior a la dimensión de la matriz, `n`.

Este tipo de paralelismo, así como el comentado en la sección anterior es explotado en los siguientes trabajos que resumimos a continuación. Para máquinas con memoria compartida, encontramos las aportaciones de Alaghband (1995) [3, 4], Davis y Yew (1990) [47, 48] y Zlatev y col. (1995) [162]. Alaghband usa una tabla de $n \times n$ para representar la compatibilidad entre los elementos de la diagonal. Si algún elemento de la diagonal es inestable numéricamente siempre es posible realizar una permutación de filas o columnas para reemplazarlo. Inicialmente todos los elementos de la diagonal se consideran compatibles, para posteriormente, mediante una búsqueda en árbol descartar los incompatibles. Sorprendentemente, sólo presenta resultados para dos matrices de $n = 144$ y $n = 505$, ambas con la misma densidad, $\rho \approx 2\%$, consiguiendo unas eficiencias del 33% y del 53% respectivamente, en los 8 procesadores del *Sequent Symmetry* con memoria compartida.

El algoritmo D2 de Davis utiliza un conjunto S de pivots compatibles. Todos los pro-

cesadores buscan candidatos en paralelo para intentar añadirlos al conjunto S evitando conflictos mediante secciones críticas. Experimentos en el Alliant FX/8 de memoria compartida reportan una eficiencia media inferior al 50% en ocho procesadores.

El código Y12M2 de Zlatev es una versión paralela, con actualizaciones de rango `m`, de un algoritmo secuencial previo llamado Y12M. En este código se aceptan como pivots aquellos que sean parcialmente compatibles, consiguiendo bloques diagonales más grandes a costa de una etapa de actualización más costosa. Comparando con la versión Y12M1, que sólo realiza actualizaciones de rango uno, concluyen que Y12M2 es más rápido para matrices dispersas que no se llenan demasiado, mientras que Y12M1 es aconsejable para matrices más densas o de rápido llenado. En cualquier caso, para 27 matrices del conjunto Harwell-Boeing, Y12M1 presenta aceleraciones entre 2.4 y 5.4 mientras que Y12M2 entre 2.3 y 5.0 en los 8 procesadores de la FX/80. Podemos concluir que las eficiencias de las implementaciones actuales en memoria compartida, a duras penas superan el 50% de eficiencia en 8 procesadores.

Mejores resultados se consiguen en arquitecturas de memoria distribuida, como los presentados por Stappen, Bisseling y van de Vorst (1993) [154] para una malla cuadrada de Transputer, Koster y Bisseling (1994) [94, 96] para el Meiko CS2 y el IBM SP2, y nuestra implementación que aventaja a las anteriores [21]. Esta última se presenta en detalle a continuación, observándose las ventajas frente a las implementaciones anteriores.

3.2 Distribuciones y esquemas de distribución

Típicamente, tanto en algoritmos de paralelismo de datos como de pase de mensajes, la distribución del trabajo entre los procesadores normalmente se desprende de la conjunción de dos aspectos:

- La distribución de los datos a procesar. Más formalmente, distribuir un array es aplicar una función, llamada de distribución, que asigna cada índice del array a un procesador. Existe una función de distribución por cada dimensión del array.

- La regla "computa el propietario" (o en inglés *Compute the owner*). Es decir, cada procesador se ocupa del subconjunto de la carga que le corresponde. Aumentando la localidad evitaremos que los procesadores tengan que interaccionar para intercambiar información, ahorrando así el coste asociado.

Si ésto es así, la elección de la distribución óptima conducirá a códigos paralelos balanceados, con elevada localidad y por tanto de escasa penalización por comunicaciones. En efecto, la determinación de una buena distribución es crítica en las prestaciones finales del algoritmo paralelo.

Para algoritmos regulares y códigos bien estructurados, y por tanto también, para algoritmos que operan sobre matrices densas, sólo dos tipos de distribuciones han bastado para conseguir los propósitos comentados en el párrafo anterior:

- Distribución por **bloques**, `BLOCK`. El espacio computacional es descompuesto en subespacios regulares de igual tamaño. La asignación de subespacios a procesadores se lleva a cabo de forma que subespacios vecinos correspondan a procesadores vecinos.

- Distribución **k-cíclica**, `CYCLIC(k)`. Distribuye bloques consecutivos de tamaño `k` en procesadores consecutivos. Cuando `k` es uno, tenemos un caso particular de la distribución que llamamos entonces **cíclica**. Cuando el valor de `k` coincide con el del tamaño del problema dividido entre el número de procesadores, la distribución es equivalente a la de bloques.

Cuando consideramos algoritmos irregulares, o en particular, dispersos, estas distribuciones no redundan en los resultados que alcanzaban en el caso denso. Ésto es así, principalmente por la siguiente razón: la información espacial, es decir, las coordenadas, de una entrada forman parte de la estructura de datos, pero no son tenidas en cuenta por las distribuciones arriba mencionadas a la hora de partir la matriz. Con otras palabras, la distribución por bloques y k-cíclica parten estructuras n-dimensionales sin reparar en lo que representan. Por tanto, cuando de matrices dispersas se trata, no atender a la información espacial conlleva a una importante pérdida de localidad y a un eventual desbalanceo desproporcionado.

Una vez encontrada la razón de nuestros problemas, la solución aparece casi trivialmente: el algoritmo de distribución no debe obviar ni prescindir de la información espacial. Muy bien, si con dos distribuciones fue suficiente para repartir el trabajo con códigos regulares, una generalización de éstas, ampliadas para soportar matrices dispersas nos puede bastar. Las propuestas, que llamaremos distribuciones **pseudo-regulares**, se comentan a continuación [127]:

- **Descomposición recursiva múltiple** (MRD o *Multiple Recursive Decompositon*). Es una generalización de la distribución propuesta por Berger y Bokhary [27] para un número arbitrario de procesadores, $P \times Q$. Asigna bloques de tamaño variable a procesadores, respetando el balanceo en el número de entradas por bloque. En la figura 3.4 (a) ilustramos el resultado de esta distribución para una matriz dispersa en una malla de 6×4 procesadores. Las regiones en la figura marcan cada una de las submatrices, tantas como procesadores, y se puede apreciar la desigualdad de sus dimensiones en aras del balanceo de las entradas. Es claro que esta distribución es un caso general de la distribución por bloques.

- Distribución **k-cíclica dispersa**. La clave de esta técnica reside sencillamente en aplicar la distribución k-cíclica [86, 93] considerando las coordenadas de las entradas. Lo dicho es equivalente a considerar, en tiempo de distribución, la matriz dispersa como densa, y distribuir cíclicamente ésta última. El resultado es un conjunto de submatrices dispersas que serán almacenadas como tales. Al caso cíclico puro ($k = 1$) disperso se le suele llamar distribución **scatter** o **grid**. En la figura 3.4 (b) encontramos una matriz distribuida cíclicamente para una malla de 16×16 procesadores. Las regiones de la figura representa una cuadrícula de 16×16 posiciones, y cada una de estas posiciones está asociada a un procesador: Una entrada situada en las coordenadas (i, j) de una cuadrícula será asignada al

procesador de coordenadas (i, j). Como anteriormente, esta distribución es un caso general de la distribución k-cíclica.

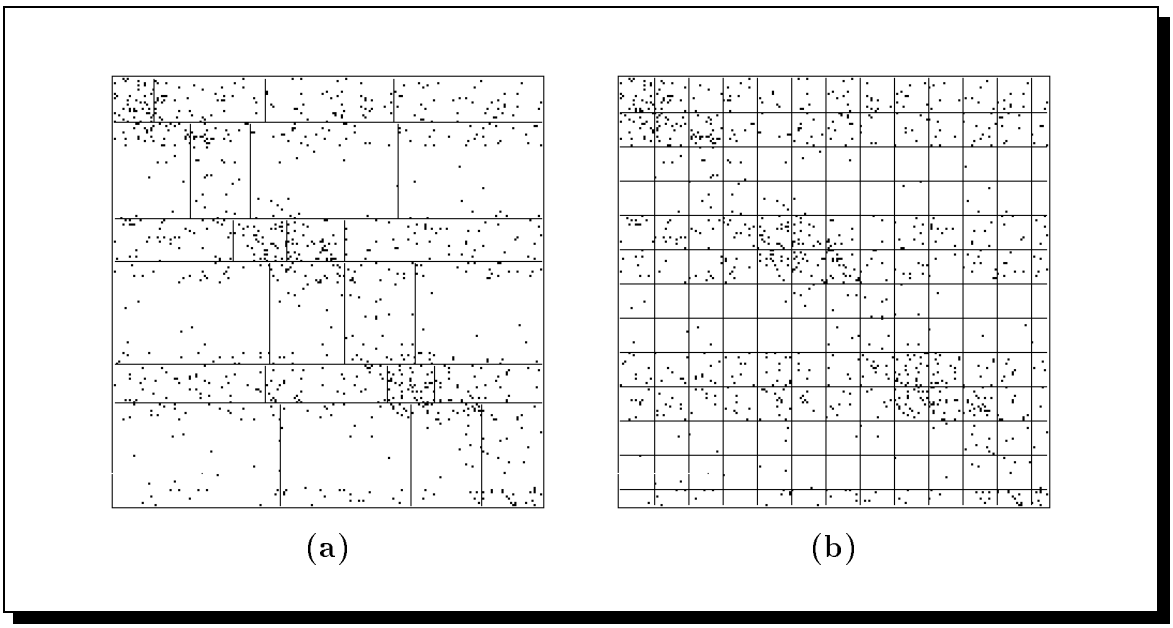


Figura 3.4: Distribuciones dispersas: (a) MRD y (b) Scatter.

Así como en la distribución MRD el balanceo de la carga delega en el algoritmo de partición, para el caso scatter, la distribución será equitativa cuando la probabilidad de que un coeficiente sea distinto de cero es independiente de sus coordenadas. Esta condición es cierta para todas las matrices de patrón aleatorio, pero también para todas aquellas que no presenten periodicidad[3] en la aparición de entradas. Además la distribución scatter va a dispersar núcleos de entradas agrupadas (*cluster* de entradas), en procesadores distintos.

Para el problema de la factorización LU paralela, donde el bucle `k` va recorriendo la diagonal, la distribución de datos debe ser la scatter. Si se aplica la distribución MRD, los procesadores encargados de las primeras filas y columnas de la matriz quedarán ociosos a las pocas iteraciones de comenzar el algoritmo. Por otro lado, es sabido que el llenado tiene mayor impacto en el cuadrante inferior derecho de la matriz, de forma que en esa zona aparecerá una agrupación *cluster* de entradas. Sin embargo mediante la distribución cíclica este llenado aparecerá equitativamente distribuido, hecho que no tendría lugar bajo una distribución MRD. De hecho, la distribución scatter es útil en problemas del álgebra matricial donde además aparecen vectores densos, que distribuidos cíclicamente quedan automáticamente alineados con las filas o las columnas de la matriz. Ejemplos son otros métodos directos como QR [146] o Cholesky [130]; iterativos como Gradiente Conjugado, Biconjugado Estabilizado, Jacobi, etc; métodos de autovalores [148, 149]; etc...

Por otro lado, la localidad que alcanza la distribución MRD, asignando a un mismo procesador elementos vecinos, la hace apropiada para su aplicación en problemas de

[3]De período no primo con el número de procesadores.

elementos y diferencias finitas, dinámica molecular y de fluidos, procesado de imágenes y otros problemas relacionados con el modelado.

Otras distribuciones dispersas orientadas a problemas basados en grafos, como la bipartición de grafos o espectral [117, 38], conducen a pérdidas de localidad en problemas del álgebra matricial y en particular en la factorización LU. Una comparativa entre estas distribuciones no regulares y las pseudo-regulares propuestas se encuentra en [39], donde se concluye que las primeras no resultan en una mejora sustancial del balanceo y además son lentas en el cálculo de la distribución.

Aparte de la distribución, es importante definir otras operaciones relacionadas: el **alineamiento** y la **replicación**. Alinear un array (llamado *alineado*) con otro ya distribuido (llamado *alineador*) consiste en aplicar la misma función de distribución a los índices del array que se alinea. Cuando los arrays son de distintas dimensiones es necesario especificar que dimensiones de los arrays *alineado* y *alineador* han de ser tenidas en cuenta. Por otro lado, un array puede ser replicado cuando alguna de sus funciones de distribución asigna cada índice a más de un procesador.

Por último, queremos dejar claro un concepto importante. Dado que tanto la estrategia de distribución como la estructura de datos determinan en gran medida el código final, se suelen representar conjuntamente bajo el concepto de **esquema de distribución de los datos**.

En efecto, la conjunción del tipo de distribución con la estructura de datos aplicada comprende toda la información necesaria para localizar cualquier entrada en el sistema multiprocesador. El esquema de distribución agrupa estos dos parámetros. Por ejemplo, la combinación de CRS o CCS y MRD genera los esquemas **CRS-MRD** y **CCS-MRD**, mientras que al esquema CRS-Scatter lo llamamos **BRS** (*Block Row Scatter*) y a CCS-Scatter, **BCS** (*Block Column Scatter*). Estos esquemas fueron validados estadística y empíricamente en [17]. Otros esquemas válidos son el resultado de combinar las estructuras LLRS/LLCS/LLRCS con las distribuciones MRD y Scatter. Si nos centramos ya en la distribución scatter, la figura 3.5 muestra como descomponemos la matriz y el tipo de almacenamiento utilizado, para un esquema de distribución BCS sobre una malla de 2×2 procesadores.

Sin embargo, quizás la idea más importante que subyace bajo el concepto de esquema de distribución es la de, independientemente de la distribución, mantener la misma estructura de representación de los datos del código secuencial para la versión paralela. Respetar este aspecto permitirá que el código paralelo no sea más que una generalización del secuencial, al que únicamente se le han ajustado las cotas de los bucles y se le han añadido puntos de comunicación.

3.3 Algoritmo Right-looking paralelo

En esta sección presentamos un algoritmo paralelo, que llamaremos **SpLU**, para la factorización LU de matrices dispersas genéricas con las siguientes características básicas:

- El algoritmo es de tipo *right-looking*. El código, escrito en lenguaje C, es SPMD y está diseñado para un multiprocesador con memoria distribuida. Se dispone de

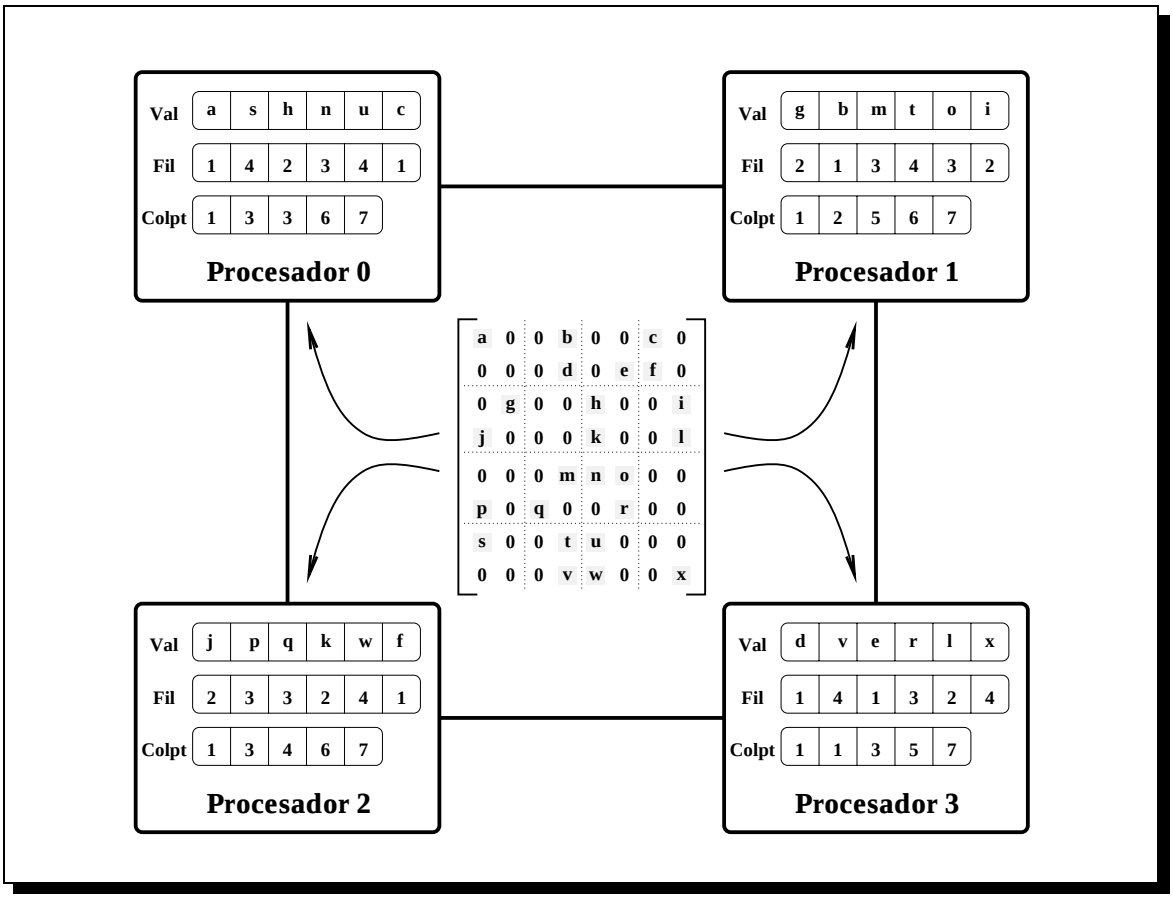


Figura 3.5: Esquema de distribución BCS para una malla de 2×2 procesadores.

una versión portable que hace uso del interfaz de pase de mensajes MPI y otra más optimizada para el Cray T3D o T3E mediante la librería SHMEM.

- La distribución de la matriz dispersa sigue el esquema *scatter* en dos dimensiones, de forma que el algoritmo se ajusta a una malla de $P \times Q$ procesadores (PE's).

- La estructura de datos utilizada para almacenar las matrices locales es una lista bidimensional doblemente enlazada. Las entradas se enlazan por columnas en orden creciente del índice de filas, i, pero el orden de enlazado por filas es independiente del índice de columna. Decimos que la lista es semi-ordenada por columnas.

- Se explota tanto el paralelismo inherente a los bucles de actualización como el que extraemos de seleccionar `m` pivots paralelos. Las etapas de análisis y factorización se encuentran combinadas en una etapa de análisis-factorización.

- Se utiliza un heurístico basado en umbral para asegurar la estabilidad numérica, y el criterio de Markowitz de mínima fila en mínima columna para mantener el coeficiente de dispersión. El número de columnas en las que se buscan pivots compatibles se adapta dinámicamente a la densidad de la matriz. El pivoteo completo explícito reduce los problemas de desbalanceo.

- Cuando la densidad de la matriz alcanza cierto umbral se conmuta a un código paralelo de factorización LU de matrices densas.

Este algoritmo paralelo ejecuta una serie de iteraciones, en cada una de las cuales se distinguen tres fases: generación del conjunto de `m` pivots compatibles, llamado `PivotSet`; permutaciones de filas y columnas; y actualización de la submatriz reducida. Las tres etapas son paralelas.

Aparte de la estructura asociada a la matriz A, la implementación del criterio de Markowitz implica la utilización de dos vectores, R y C, de dimensión n, que llamaremos vectores de cuenta de entradas. El valor $R_i^{(k)}$ representa el número de entradas en la fila i de la matriz activa en la iteración `k`, mientras que $C_j^{(k)}$ el número de entradas en la columna j.

Los vectores $C^{(k)}$ o $R^{(k)}$ también son utilizados para decidir la iteración en que conmutamos a un código de factorización densa. La densidad, `dens`, de la submatriz reducida se calcula en cada iteración. Para ello es suficiente con sumar los valores de $C^{(k)}$ o $R^{(k)}$. De esta forma tenemos el número de entradas en la submatriz reducida y también, por tanto, su densidad. Cuando esta densidad supera el valor `maxdens`, inicializado por el usuario, y aún queda una submatriz reducida lo suficientemente grande como para compensar el cambio de estructura de datos, automáticamente conmutamos al código denso.

Por otra parte, los vectores de permutación necesarios para la posterior resolución del sistema, serán π y γ, ambos de dimensión n. Si $\pi_i^{(k)} = r$ quiere decir que en la iteración `k` la fila `r` de la matriz original, $A^{(0)}$, está ahora en la fila `i` de la matriz $A^{(k)}$. De igual forma para el vector de permutaciones de columnas, γ, cambiando fila por columna, en la definición anterior. Dado que el programa está escrito en C, el recorrido de todos los índices comentados en este párrafo, i, j y k, va de cero a $n-1$.

Una descripción más detallada del algoritmo es entonces la siguiente:

Ejemplo 3.1 PASOS EN EL ALGORITMO PARALELO

1. $\pi = \gamma =$ identidad;
2. Inicializa R y C; Calcula `dens`
3. $k = 0$;
4. **while** ($(k < n)$ && (`dens`<`maxdens`))
   {
5. Busca el conjunto de pivots: `PivotSet`$= (i_r, j_r) \ : \ 0 \leq r < m$;
6. Permuta filas, π_i y R_i;
7. Permuta columnas, γ_j y C_j;
8. Actualiza $A^{(k)}$;
9. Actualiza los vectores R y C;
10. $k = k + m$;
   }
11. Factoriza la submatriz densa;

En la descripción del algoritmo del ejemplo anterior, 3.1, asumimos implícitamente que cada procesador, (p, q), realiza las computaciones sobre su datos locales. En las

líneas 1–3, se inicializan los vectores de permutación con vectores identidad, los vectores de cuenta de entradas, y el índice `k` del bucle principal al valor 0. Las líneas 5–10 componen el cuerpo del bucle principal, de las cuales, la línea 5 se encarga de la selección del conjunto de `m` pivots paralelos –figura 3.3 (a)–; las líneas 6 y 7, de las permutaciones de filas y columnas y consecuentemente también de π, γ, R y C –figura 3.3 (b)–; y las líneas 8 y 9 representan la actualización de la submatriz reducida de A, actualizando apropiadamente los vectores de cuenta –figura 3.3 (c)–. Finalmente en la línea 10 se incrementa el índice del bucle en `m`, pasando la submatriz reducida de la iteración anterior a ser la submatriz activa de la iteración actual. En la línea 11 conmutamos a un código de factorización densa.

Al final del algoritmo, la matriz $A^{(n-1)}$ almacena los coeficientes de la matriz L en su parte triangular estrictamente inferior, y los coeficientes de U en su submatriz triangular superior. Los vectores $\pi^{(n-1)}$ y $\gamma^{(n-1)}$ contendrán sus valores finales. Este algoritmo paralelo no es más que una generalización de su versión secuencial, la cual presenta la misma estructura descrita pero opera sobre la totalidad de la matriz A.

3.3.1 Esquema de distribución

La matriz A se distribuye sobre una malla de $P \times Q$ procesadores. Identificaremos cada procesador por medio de sus coordenadas cartesianas (p, q), con $0 \leq p < P$ y $0 \leq q < Q$, o por su número de procesador, nproc $= pQ + q$. Las entradas de la matriz A son asignadas a los procesadores según la distribución *scatter* o *cíclica dispersa*, por las ventajas comentadas en la sección 3.2. Más precisamente esta distribución se ajusta a la siguiente ecuación:

$$A_{ij} \longmapsto PE(i \bmod P,\ j \bmod Q) \quad \forall i, j,\ \ 0 \leq i, j < n. \tag{3.1}$$

Los vectores de permutación, π y γ son replicados parcialmente. Es decir, almacenaremos π_i en todos los procesadores con coordenadas $(i \mod P, *)$, donde $*$ indica cualquier valor entre cero y $Q-1$. Similarmente, γ_j se asignará a todos los procesadores con coordenadas $(*, j \mod Q)$. Los vectores R y C serán distribuidos de la misma forma que π y γ, respectivamente. Diremos, por tanto, que π y R están alineados con las columnas de A y replicados en las columnas de la malla. De la misma forma, γ y C están alineados con las filas de A y replicados en las filas de la malla.

Llamaremos $\hat{A}$ a la matriz local de $\hat{m} \times \hat{n}$, donde $\hat{m} = \lceil n/P \rceil$, y $\hat{n} = \lceil n/Q \rceil$. La notación con *sombrero* nos permite distinguir las variables e índices locales de los globales. Por tanto, en el procesador (p, q), la relación entre A y $\hat{A}$ vendrá dada por la siguiente ecuación:

$$\hat{A}_{\hat{i}\hat{j}} = A_{\hat{i}P+p, \hat{j}Q+q} \quad \forall \hat{i}, \hat{j},\ \ 0 \leq \hat{i}P + p, \hat{j}Q + q < n. \tag{3.2}$$

En cuanto a la elección de la estructura de datos, hay que tener en cuenta que el algoritmo que vamos a implementar utiliza pivoteo total y por tanto necesita accesos tanto por filas como por columnas. Por tanto, almacenaremos las submatrices locales dispersas mediante una lista bidimensional doblemente enlazada, LLRCS (de *Linked List Row-Column Storage*). Esta estructura dinámica, enlaza todos los elementos de una misma fila de forma ordenada (según los índices crecientes de columna), y las

entradas de una misma columna en cualquier orden. Adicionalmente, utilizamos dos vectores de punteros al comienzo de cada fila, `fils`, y columna, `cols`, de longitud $\hat{m}$ y $\hat{n}$ respectivamente. Cada entrada en la lista, almacena el valor del coeficiente, los índices locales de fila y columna y punteros a entradas anteriores y posteriores en la misma fila y columna. La declaración de dicha estructura de datos en C se muestra en el ejemplo 3.2.

Ejemplo 3.2 DECLARACIÓN EN C DE LA LISTA ENLAZADA

```
struct matriz {
                struct entrada *fils[m̂], *cols[n̂];
              };
struct entrada {
                int Fil,Col;
                double Val;
                struct entrada *previ, *prevj, *nexti, *nextj;
              };
```

Dado que la lista está enlazada en las dos direcciones, el borrado de elementos necesarios en la permutación se simplifica. Sin embargo es claro el consumo de memoria debido a los punteros. Por tanto, combinando la distribución scatter con la estructura LLRCS, tendremos un esquema de distribución LLRCS-Scatter. En la figura 3.6, mostramos un ejemplo de este esquema para una matriz de dimensión $n = 8$, $\alpha = 24$ y para una malla de 2×2 procesadores.

Una vez decidido el esquema de distribución podemos pasar a discutir en detalle las tres etapas del algoritmo: búsqueda de los pivots compatibles, permutaciones y actualización; dedicando una subsección a cada una de ellas.

3.3.2 Búsqueda de los pivots en paralelo

Una estrategia simple y efectiva para realizar la búsqueda de los pivots compatibles consiste en restringir la búsqueda entre los coeficientes de las columnas más vacías. En general, el algoritmo usado para encontrar el mencionado conjunto de pivots debe ser simple, o en caso contrario las comunicaciones implicadas encarecerán esta etapa e incluso pueden hacerla prohibitiva. En nuestro caso, la búsqueda se realizará en tres pasos: búsqueda de candidatos, evaluación de la compatibilidad y construcción del conjunto de pivots paralelos, `PivotSet`.

Ya se ha comentado que a consecuencia del llenado que tiene lugar durante la factorización, a cada iteración, la submatriz reducida es potencialmente más densa. Por tanto llegará un momento en que intentar explotar el paralelismo asociado a la dispersión de la matriz sea improductivo. Es decir, buscar un elevado número de pivots compatibles no tiene sentido si la matriz es poco dispersa, ya que la mayoría de los candidatos serán incompatibles entre sí.

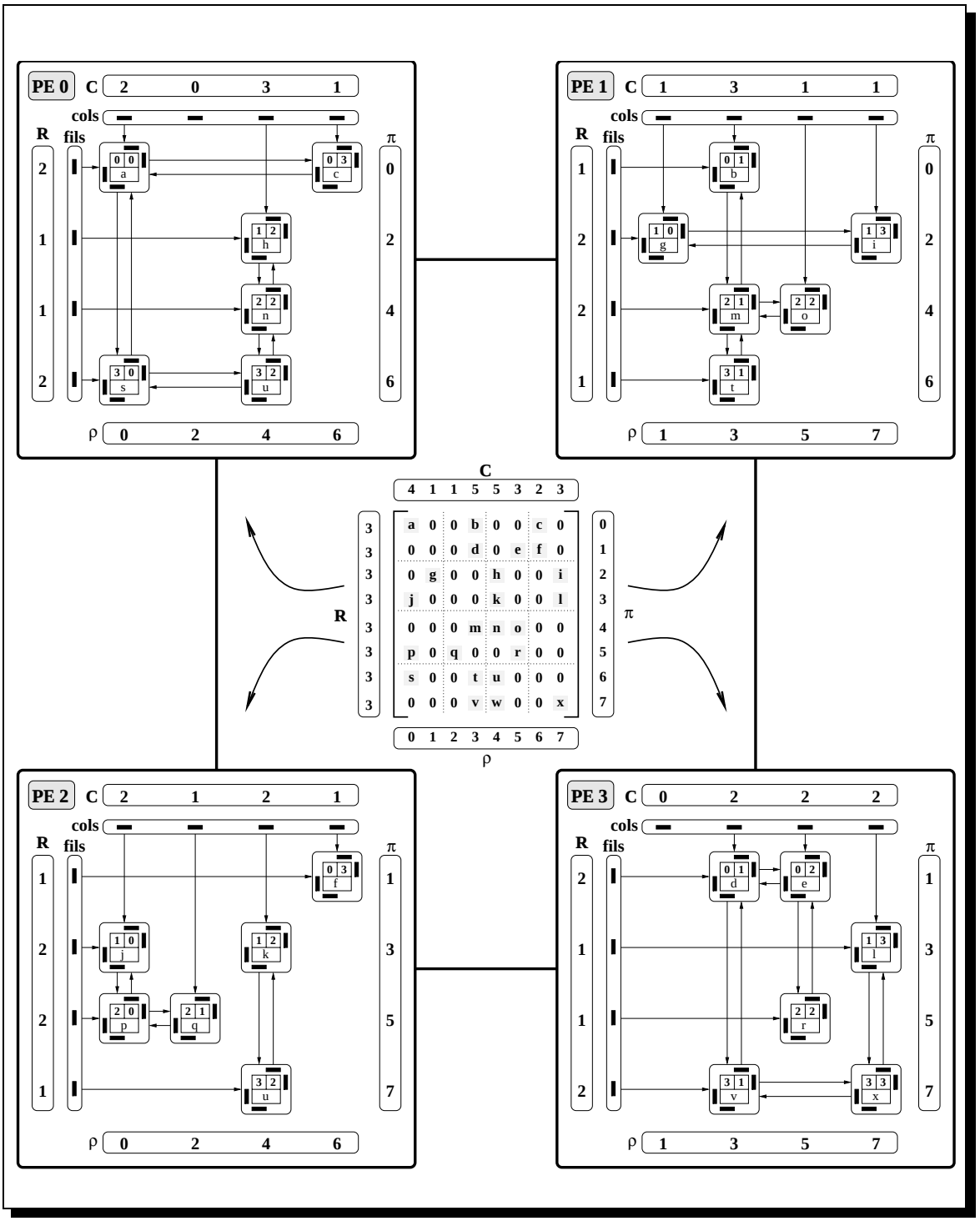


Figura 3.6: Esquema de distribución LLRCS-Scatter.

Nuestro algoritmo implementa una estrategia adaptativa al explotar la dispersión de la matriz. Inicialmente, la variable `ncol` controlada por el usuario, contiene el número de columnas en las que cada columna de procesadores buscará pivots candidatos. Por tanto, inicialmente el valor máximo de `m` será igual a `ncol`$\times Q$. El tamaño del conjunto de pivots compatibles, `m`, puede ser chequeado en cada iteración, y mantener en un

array, `histm` de `ncol`$\cdot Q$ posiciones un histograma de la evolución de `m`. Es decir, en `histm[i]` tendremos el número de iteraciones en que hemos encontrado `i` pivots compatibles. Después de un determinado número de iteraciones, `NumIterTest`, se determina el máximo de este histograma, que dará una idea de como de fructífera o infructuosa está resultando nuestra búsqueda. Si este máximo es menor o igual que (`ncol`$-1) \cdot Q$ decrementamos `ncol` en una unidad, mientras que si es igual a `ncol`$\cdot Q$ incrementamos `ncol`. En el resto de los casos `ncol` se mantiene constante. En la búsqueda del máximo se inicializa a cero dicho histograma, de forma que considere la historia reciente del tamaño de los conjuntos `PivotSet` de las últimas `NumIterTest` iteraciones. El valor máximo de `ncol` está limitado por el parámetro `maxncol`.

Es también al comienzo de cada iteración `k` cuando calculamos la densidad de la submatriz reducida. Para ello basta con realizar una suma local en paralelo de los coeficientes asociados en el vector $C^{(k)}$ seguida de una reducción global por filas de la malla[4]. Para evitar en la medida de lo posible el coste asociado a esa reducción, el cálculo de la densidad sólo se realiza a partir del momento en que `ncol` baje por debajo de cierto umbral, `minncol`.

3.3.2.1 Búsqueda de candidatos

El parámetro de entrada `ncol` nos indica el número de columnas en las que cada columna de la malla realizará la búsqueda de pivots compatibles. Cada procesador (p, q) tendrá parte de cada columna j, de la submatriz activa ($k \leq j < n$), tal que $j \mod Q = q$. De este conjunto de columnas serán elegidas aquellas con $j : C_j^{(k)} = \min_l C_l^{(k)} : k \leq l < n$. De estas columnas obtendremos los `ncol` pivots candidatos. Si $n - k < ncol$, la búsqueda tendrá lugar en las $n - k$ columnas restantes.

Al elegir los pivots debemos asegurar la estabilidad numérica y preservar el coeficiente de dispersión de la matriz. Para perseguir el primer aspecto, sólo consideraremos pivots candidatos a aquellos que superen un valor umbral. En nuestro caso, este umbral viene determinado por el producto de un parámetro de entrada, `u`, por el valor absoluto del máximo elemento en la columna activa, tal y como se indica en la ecuación 1.7. Por tanto el cálculo del umbral implica determinar el máximo global en las `ncol` columnas activas. Esta reducción se organiza en dos pasos: primero cada procesador busca los `ncol` máximos locales en paralelo y los almacena en un array; en segundo lugar se realiza la reducción global de este array. En esta reducción global, los procesadores frontera envían su array a sus vecinos, éstos calculan el *array máximo* y vuelven a enviar. Este proceso continúa hasta que se alcanza el procesador central de la columna de procesadores. Este último, determinará el array con los `ncol` máximos globales, que será radiado a toda la columna de la malla.

El proceso de reducción global y la subsiguiente radiación de los arrays se muestra en la figura 3.7: En (a) mostramos las submatrices locales asignadas a cada uno de los 15 procesadores de una malla de 5×3. En este caso `ncol`=2 y cada columna de procesadores selecciona la dos columnas más dispersas de su submatriz activa. Los máximos locales, marcados por un círculo en negro en la figura, se seleccionan en

[4]Si $P > Q$ la reducción por columnas es ventajosa y por tanto se suman los coeficientes de $R^{(k)}$.

paralelo y se copia su valor absoluto en un array. En la figura 3.7 (b), se muestra la reducción global del array de máximos hacia la fila del centro de la malla. Por último, en (c) se muestra la radiación de los máximos globales en cada columna de procesadores.

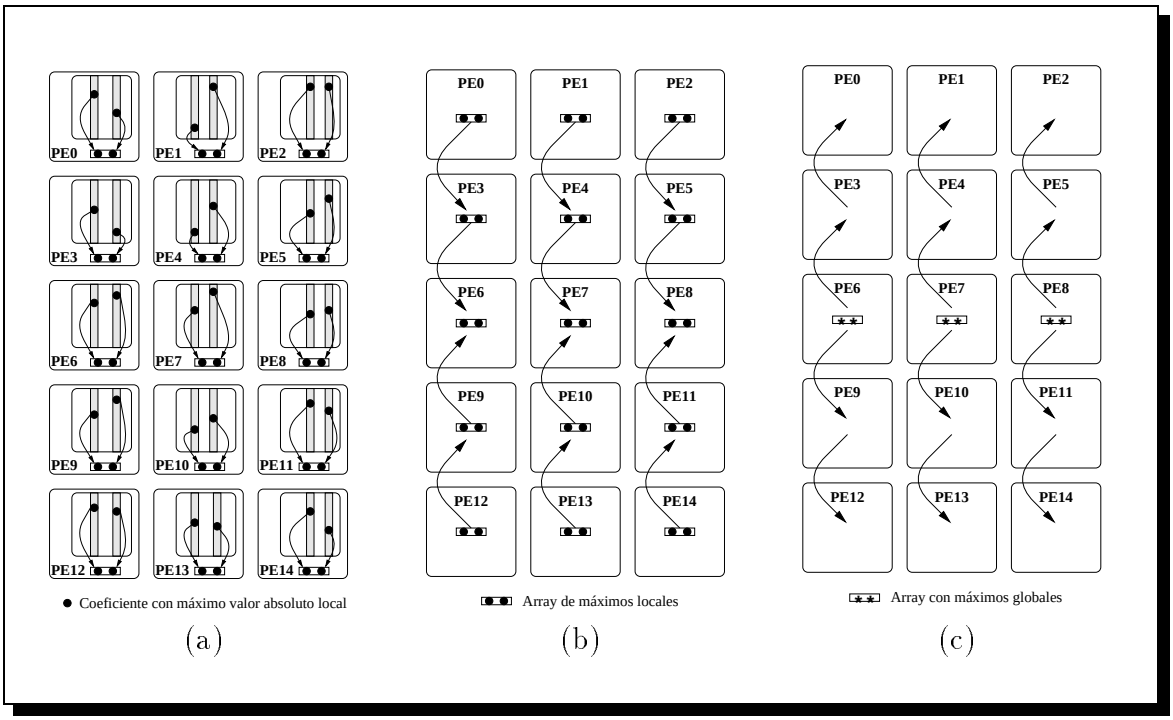


Figura 3.7: (a) Selección de máximo local. (b) Reducción global. (c) Radiación.

Para atender ahora al problema de preservar el coeficiente de dispersión, utilizaremos el criterio de Markowitz comentado. Es decir, los pivots candidatos que vamos a seleccionar de cada columna deben superar el valor de umbral y minimizar la cuenta de Markowitz, M_{ij}. Para ello, cada procesador comienza buscando un elemento local que cumpla las dos condiciones por cada una de sus `ncol` columnas. A continuación, en un proceso similar a la búsqueda del máximo global, los candidatos locales se reducen hacia la fila central de la malla. En la selección, un candidato descarta a otro de su misma columna si tiene menor M_{ij}, o en caso de empate, si tiene mayor valor absoluto. Al final del proceso, los procesadores de la fila central de la malla tendrán un array con los `ncol` pivots candidatos que superen el umbral y minimicen M_{ij}. Los candidatos se ordenan localmente en este array según un orden creciente de M_{ij}. Este proceso se muestra gráficamente en la figura 3.8 (a).

La búsqueda de los pivots candidatos concluye con un último paso en que se crea un conjunto, `Candidatos`, disponible en todos los procesadores. Inicialmente, los procesadores frontera de la fila central de la malla envían su conjunto local de candidatos a sus vecinos. Estos, amplían su conjunto de candidatos, insertando apropiadamente los candidatos que llegan según un orden creciente de M_{ij} y vuelven a enviar, como se muestra en la figura 3.8 (b). Si eventualmente aparecen candidatos pertenecientes a la misma fila, se eliminan los de mayor M_{ij}, ya que son claramente incompatibles. Este proceso se repite hasta alcanzar el procesador central de la malla. En este punto

el procesador central contiene el conjunto $Candidatos = (i_r, j_r) : 0 \leq r < ncand$ donde $M_{i_r,j_r} < M_{i_{r'},j_{r'}}$ si $r < r'$, y $ncand \leq min(Q \cdot ncol, n - k)$ representa el número total de candidatos.

Para tener un mayor control sobre el llenado de la matriz que tendrá lugar durante la factorización, utilizaremos otro parámetro de entrada, `a`: todos los candidatos (i, j) con cuenta de Markowitz inaceptable, $M_{ij} > a \cdot M_{i_0,j_0}$, serán rechazados. Davis hace uso de este heurístico en [48], realizando sus experimentos con $a = 4$. El conjunto `Candidatos` resultante es radiado a toda la malla, según se ve en la figura 3.8 (c).

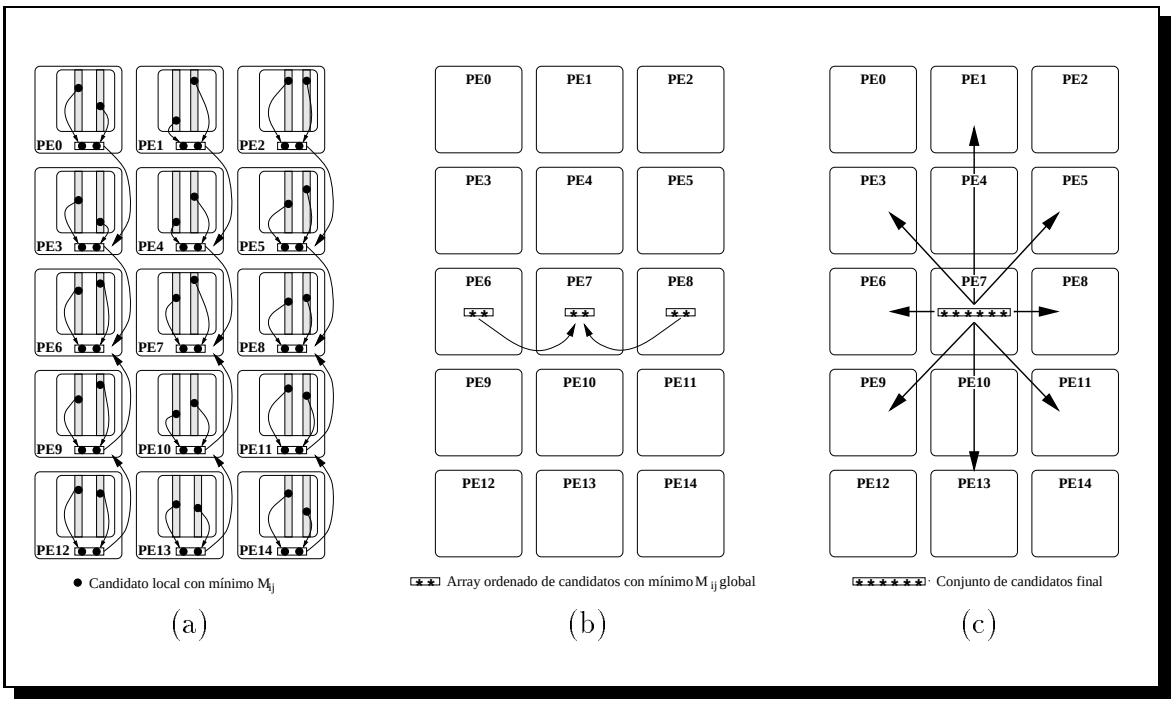


Figura 3.8: (a) Reducción de los candidatos a la fila centra. (b) Colección y ordenación de los candidatos hacia el procesador central. (c) Radiación del conjunto `Candidatos`.

Como podemos ver, si $u \to 0$, primaremos la estabilidad numérica, mientras que si $u \to 1$, un mayor número de candidatos superarán el umbral, existiendo una mayor probabilidad de conseguir un candidato de mejor M_{ij}, lo cual resulta en un menor llenado de la matriz factorizada.

3.3.2.2 Compatibilidad de los candidatos

En esta segunda fase en la búsqueda de pivots compatibles, determinamos la compatibilidad de cada pivot candidato con los restantes del conjunto. Como ya fue comentado, dos entradas de la matriz A_{ij} y A_{rs} son compatibles si $A_{is} = A_{rj} = 0$. Ya que todos los procesadores disponen del conjunto `Candidatos`, cada procesador puede realizar la búsqueda en paralelo de los coeficientes que impliquen incompatibilidad entre ciertos candidatos. Esto es, para cada par de candidatos A_{ij} y A_{rs}, el procesador (i mod P, s mod Q) determinará si el elemento A_{is} es cero o no. En paralelo, el procesador

(r mod P, j mod Q) hará lo propio con el elemento A_{rj}. Si alguno de estos dos coeficientes, (A_{is} o A_{rj}) es distinto de cero, los candidatos A_{ij} y A_{rs} no son compatibles. Por tanto, en el procesador correspondiente, o puede que en ambos, el par $((i,j),(r,s))$ será añadido a un conjunto de candidatos incompatibles. Esto se muestra gráficamente en la figura 3.9 (a), donde un array de rombos representa el máximo número de pares $((i,j),(r,s))$ que pueden aparecer en cada procesador.

En la figura 3.9 (b) y (c), se presenta la colección de todas las incompatibilidades, primero por columnas y luego en la fila central de la malla. Al final del proceso, en el procesador central de la malla tendremos el conjunto `Incompatibles`.

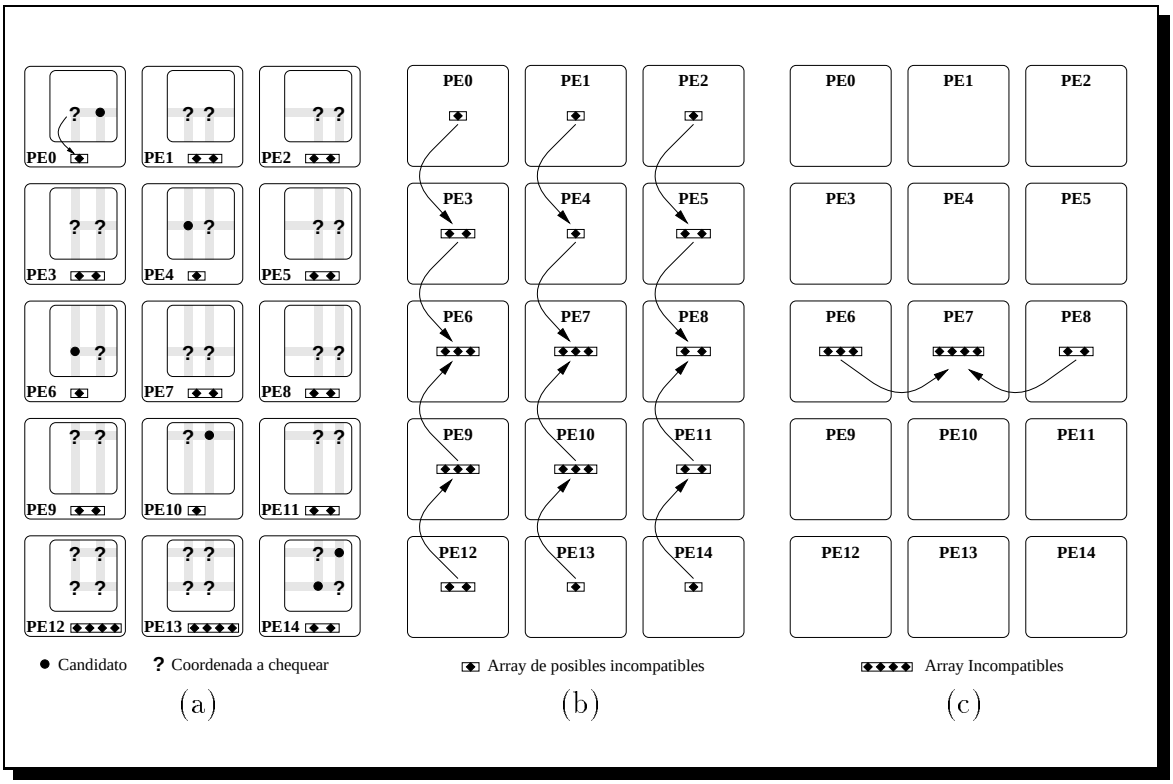


Figura 3.9: Búsqueda paralela de incompatibilidades (a). Colección de incompatibilidades por columnas (b) y en la fila central (c).

3.3.2.3 Construcción del conjunto `PivotSet`

En esta fase, el procesador central crea el conjunto `PivotSet` y lo distribuye a toda la malla. Para ello, este procesador va recorriendo el conjunto `Candidatos` en orden creciente de M_{ij} y marcando los candidatos no compatibles de mayor cuenta de Markowitz. Esto es, si el par $((i,j),(r,s))$ se encuentra en el conjunto de `Incompatibles`, el candidato (r,s) será marcado para su eliminación. Una vez marcados todos los candidatos incompatibles, copiaremos en el conjunto `PivotSet` los pares (i,j), no marcados. Este conjunto estará por tanto definido por el conjunto de pares $\{(i_l,j_l)\}$ $0 \leq l < m$.

Este conjunto de candidatos es reordenado a continuación para minimizar las comunicaciones implicadas en las permutaciones de filas y columnas, como se comentará en

la siguiente sección. Posteriormente el conjunto `PivotSet`, con `m` pivots compatibles, es radiado a toda la malla.

3.3.3 Permutaciones en paralelo

Una vez encontrados `m` pivots compatibles, debemos permutar las filas y columnas para colocarlos en la diagonal. Siguiendo las recomendaciones aportadas en [43, 71] aplicaremos un pivoteo explícito para mantener un mejor balanceo de la carga. El ahorro de comunicaciones resultante de aplicar pivoteo implícito (donde no permutamos vectores sino que accedemos a ellos a través de los vectores π y γ) no compensa la pérdida de balanceo.

Mediante el proceso de permutación pretendemos crear un bloque diagonal de dimensiones $m \times m$ con las m entradas del conjunto $PivotSet$ en las posiciones A_{ii}, $k \leq i < k+m$. Inicialmente realizamos una permutación de filas, lo que implica permutar las filas fuente $i_0, i_1, ..., i_{m-1}$ (indicadas en el conjunto $PivotSet$) con las filas destino $k, k+1, ..., k+m-1$ respectivamente. Para minimizar el número de permutaciones, en la fase anterior, el conjunto $PivotSet$ fue ordenado de forma que las filas destino coincidan con las filas fuente en la medida de lo posible. Esa ordenación también contempla las ventajas de que el procesador que posee la fila fuente también posea la fila destino, reduciéndose el número de comunicaciones. También con ese propósito, los mensajes se empaquetan tanto como es posible. Es decir, si un procesador envía dos filas a otro, estas dos filas se envían en un único mensaje.

También explotamos el paralelismo y el asincronismo en el proceso de permutación, realizando el envío de todas las filas, fuente o destino, en una primera fase, para posteriormente recibirse en cada procesador las filas que corresponda. La permutación de columnas es un proceso totalmente similar. Sin embargo, así como las entradas de una columna están enlazadas independientemente de la coordenada i, las filas sí están ordenadas en orden creciente de la coordenada j. Por tanto el coste de permutar columnas es algo superior, y por tanto la función que reordena el conjunto `PivotSet` prima evitar permutaciones de columnas frente a permutaciones de filas.

Los vectores π y γ se actualizan adecuadamente cuando cada procesador conoce el contenido de $PivotSet$, ya que esa información, junto con el valor actual del índice k, identifica perfectamente donde acabarán situadas las filas y columnas implicadas. Por otro lado, los valores de $R^{(k)}$ y $C^{(k)}$ asociados a filas/columnas que se permutan, también son permutados simultáneamente, empaquetándose los valores dentro de los mensajes que contienen las filas o columnas.

3.3.4 Actualización de rango `m` en paralelo

El proceso de actualización de la matriz tiene, a su vez, dos etapas diferentes. En la primera, se realiza una operación `divcol(j)` con $(k \leq j < k+m)$ en paralelo. Es decir, dividimos las subcolumnas por debajo del bloque diagonal por el pivot correspondiente,

como indica más precisamente la siguiente ecuación:

$$A_{ij}^{(k)} = A_{ij}^{(k)} / A_{jj}^{(k)} \quad \forall i,j : (k+m \leq i < n) \wedge (k \leq j < k+m) \wedge (A_{ij}^{(k)} \neq 0) \tag{3.3}$$

En la segunda etapa, actualizamos la submatriz reducida, ajustándonos a la siguiente ecuación:

$$A_{ij}^{(k)} = A_{ij}^{(k)} - A_{il}^{(k)} A_{lj}^{(k)} \qquad \forall \ i,j,l : \begin{cases} (k+m \leq i,j < n) \\ (k \leq l < k+m) \\ (A_{il}^{(k)} \neq 0) \wedge (A_{lj}^{(k)} \neq 0) \end{cases} \tag{3.4}$$

Para llevar a cabo las operaciones implicadas en la ecuación 3.3 en paralelo, es necesaria una etapa de comunicación previa para hacer visible a los procesadores que tienen una de las columnas a normalizar, el valor de los pivots por los que van a dividir. Es decir, implementaremos una radiación de los pivots por columnas, tal que los elementos $A_{jj}^{(k)} : (k \leq j < k+m) \wedge (j \bmod P = p) \wedge (j \bmod Q = q)$ se radien desde el procesador (p,q) a todos los procesadores de su misma columna: $(*,q)$.

Por otro lado, para completar la ecuación 3.4, necesitamos de una radiación por columnas de las filas activas a la derecha del bloque diagonal: (elementos $A_{lj}^{(k)} : (k \leq l < k+m) \wedge (k+m \leq j < n) \wedge (A_{lj}^{(k)} \neq 0)$, de los procesadores $(l \bmod P, j \bmod Q)$ a los procesadores $(*, j \bmod Q)$. Así mismo, es también necesaria una radiación por filas de las columnas activas por debajo de la diagonal (elementos $A_{il}^{(k)} : (k+m \leq i < n) \wedge (k \leq l < k+m) \wedge (A_{il}^{(k)} \neq 0)$, desde los procesadores $(i \bmod P, l \bmod Q)$ a los procesadores $(i \bmod P, *)$).

Así pues, los pasos a dar para actualizar la submatriz en paralelo serán los siguientes: el primer paso se muestra en la figura 3.10 (a), donde radiamos la diagonal de pivots y las filas correspondientes al mismo tiempo, empaquetando todos los coeficientes en el mismo mensaje. Con los coeficientes de la diagonal completamos la ecuación 3.3, que normaliza las columnas. El segundo paso, representado en la figura 3.4 (b), realiza una radiación de las columnas ya actualizadas al resto de la malla. De esta forma es posible llevar a cabo las operaciones implicadas en la ecuación 3.4.

Las radiaciones por filas (columnas) cuando $m \geq P$ $(m \geq Q)$, pueden considerarse como una comunicación todos a todos, debido a la distribución cíclica. Por tanto, para radiar m filas a toda la columna de procesadores, todos los procesadores tendrán que ejecutar operaciones de envío y recepción, si $m \geq P$, y de forma similar para la radiación de columnas.

El proceso de actualización se realiza directamente sobre la estructura de datos que almacena la matriz, es decir, la lista enlazada. En nuestra implementación seguimos una actualización por filas en lugar de columnas. Atendiendo a la ecuación 3.4, está claro que serán actualizadas todas las filas i de la submatriz reducida para las que exista algún $(A_{il}^{(k)} \neq 0)$ con $(k \leq l < k+m)$. Por tanto recorreremos esas filas siguiendo la lista enlazada, actualizando las entradas oportunas y creando aquellas que contendrán nuevos coeficientes distintos de cero. Por tanto, si queremos que la implementación sea eficiente, las filas deben estar ordenadas. En otro caso, deberíamos recorrer toda la fila por cada elemento a actualizar, ya que hay que averiguar si ya existe (con lo cual sólo

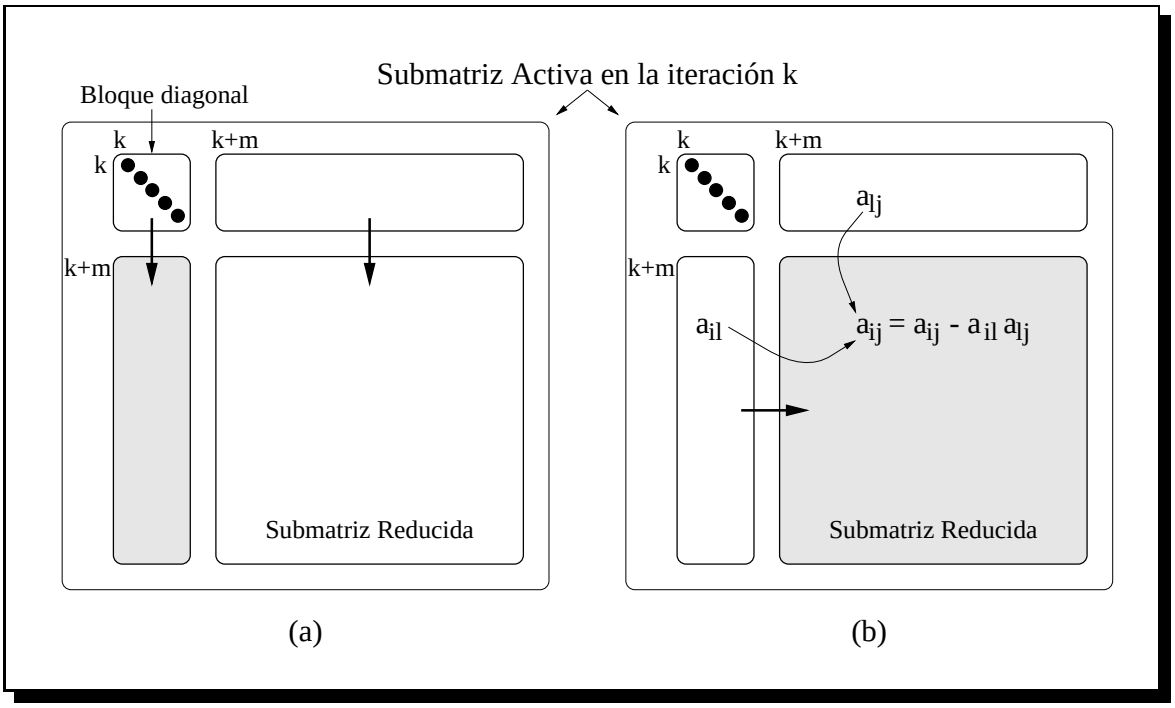


Figura 3.10: Pasos en la actualización. (a) Primero: Radiación por columnas y normalización del bloque de columnas (en gris). (b) Segundo: Radiación por filas y actualización de la submatriz reducida (en gris).

hay que modificar su valor) o no (en cuyo caso hay que crear una entrada nueva en la lista).

Otra aproximación, descrita en [57, Cap. 2.4], contempla las siguientes etapas: (i) copia la fila a actualizar sobre un vector denso de $\hat{n}$ posiciones inicializadas a cero (operación conocida con el nombre de *scatter*); (ii) actualiza el mencionado vector denso; (iii) recorre el vector denso enlazando de nuevo los coeficientes no nulos del vector en la lista enlazada, dejando a cero las posiciones del vector (operación conocida con el nombre de *gather*). La complejidad de esta técnica es independiente del ordenamiento de la lista, pero introduce un coste temporal y de memoria significativo asociado a las operaciones de *scatter* y *gather* y al vector denso de $\hat{n}$ elementos respectivamente.

También se ha comentado que mantener la lista enlazada ordenada por filas tiene asociado un coste adicional al permutar columnas. En efecto, cuando insertamos elementos en una fila, debemos encontrar primero su posición. Sin embargo, de media, para una iteración k, el número de columnas a permutar (m en el caso peor), es mucho menor que el número de filas a actualizar ($(n-k)$ en el caso peor).

En cuanto a la creación de nuevas entradas es importante tener en cuenta que la reserva de memoria implica una llamada al sistema operativo y por tanto tiene un coste temporal asociado. Por tanto, las reservas de memoria se realizan en bloques de tamaño `TAMBLOC`[5], entradas. Cuando se libera una entrada se enlaza en una lista de entradas libres. Si se necesita una entrada nueva se pide primero a la lista de entradas

[5]Por defecto, `TAMBLOC`=1000.

libres. Si no hay disponibles se extrae del bloque de `TAMBLOC` entradas, y si el bloque está agotado se reserva otro de igual tamaño.

Durante la actualización de la submatriz reducida, también vamos actualizando el número de entradas por filas y columnas en esta submatriz, reflejándolo en los vectores $R^{(k)}$ y $C^{(k)}$. Estos vectores se modifican no sólo porque las dimensiones de la submatriz se reducen, sino también debido al llenado de la matriz. En el algoritmo paralelo se utilizan dos vectores auxiliares, $PR^{(k)}$ y $PC^{(k)}$, inicializados a cero. Cuando se crea una nueva entrada en las coordenadas (i, j), el procesador $(i \bmod P, j \bmod Q)$ incrementa en uno tanto su copia local de $PR^{(k)}_{\hat{\imath}}$ como $PC^{(k)}_{\hat{\jmath}}$. Después de la actualización de la submatriz es necesaria una reducción por filas que acumule las aportaciones locales de $PR^{(k)}$ para que todos los procesadores actualicen adecuadamente el vector $R^{(k)}$. De igual forma $PC^{(k)}$ se reduce por columnas. De esta forma, los vectores $R^{(k)}$ y $C^{(k)}$ se mantienen coherentes para el cálculo de la cuenta de Markowitz en la subsiguiente búsqueda de los pivots.

3.3.5 Conmutación a código denso

Cuando se alcanza una densidad determinada en la submatriz reducida y ésta presenta aún suficiente dimensión, se procede a la conmutación a un algoritmo de factorización densa. Claramente, el paso inicial consiste en cambiar de estructura de datos de una lista enlazada, que representa la submatriz reducida $A^{(k)}$, a un array bidimensional, AD, de dimensiones $(nd \times nd)$ con $nd = n-k$. Esta operación es totalmente paralela, y una vez completada, la matriz densa AD queda automáticamente distribuida según un esquema cíclico. En primer lugar, cada procesador reserva un espacio inicializado a cero para cada submatriz local densa: $\widehat{AD}$ de $\widehat{md} \times \widehat{nd}$ con $\widehat{md} = \lceil nd/P \rceil$ y $\widehat{nd} = \lceil nd/Q \rceil$, donde $\widehat{AD}_{\hat{\imath}\hat{\jmath}} = 0 \ : \ 0 \leq \hat{\imath} < md \ \wedge \ 0 \leq \hat{\jmath} < nd$. Una vez terminada esa inicialización, cada procesador puede recorrer su submatriz reducida local, $\hat{A}^{(k)}$, copiar el valor de cada entrada en la posición $\widehat{AD}_{\hat{\imath}\hat{\jmath}}$ que corresponda y liberar el espacio asociado a esa entrada.

En ese punto podemos llamar a una rutina de factorización densa paralela. La que hemos diseñado implementa un pivoteo parcial por filas para mantener la estabilidad numérica y utiliza en lo posible rutinas BLAS-2. A cada iteración del bucle secuencial más externo, de índice `kd` se realizan los siguientes pasos: búsqueda de un pivot numéricamente estable en la columna `kd`; permutación de la fila del pivot, `i`, con la fila `kd`, para llevar el pivot a la diagonal; y actualización de rango uno en la submatriz reducida.

La búsqueda del pivot más estable numéricamente se reduce a la búsqueda global del coeficiente con mayor valor absoluto dentro de la columna activa `kd`. Como siempre, la realización de esta operación consta de una búsqueda local seguida de una reducción global en la columna (`kd` mod Q) de la malla.

En cuanto a la permutación de filas, la fila de procesadores (`kd` mod $P, *$) envía la fila `kd` a la fila de procesadores (i mod $P, *$). Simultáneamente, la fila de procesadores (i mod $P, *$) radia la fila `i` por columnas de la malla, ya que ésta es la fila del pivot y sus coeficientes son necesarios para la actualización. A continuación, todos los procesadores

de la malla reciben la porción de fila que les corresponde: los de la fila (`kd` mod $P, *$) reciben la fila del pivot y actualizan sus matrices locales; los de la fila (i mod $P, *$) reciben la fila `kd` que guardan en la fila `i` de sus submatrices locales; y el resto de los procesadores reciben una copia de la fila del pivot para poder participar en la actualización.

Dado que en la etapa anterior la fila del pivot, y por tanto también el pivot, está disponible en todos los procesadores, en la actualización de rango uno sólo son necesarias las tres siguientes operaciones: (i) llamar a la función DSCAL de BLAS, que dividirá la columna apropiada por el pivot en la columna de procesadores (*, `kd` mod Q); (ii) radiar esta columna actualizada por filas y; (iii) llamar en todos los procesadores a la función DGER de BLAS para actualizar la submatriz reducida correspondiente a cada uno de ellos.

En la sección 3.6 presentamos la evaluación de este algoritmo completo de factorización dispersa, para un conjunto de matrices dispersas. Pero antes de pasar a ese estudio, abordamos a continuación la paralelización de un código de factorización *left-looking*.

3.4 Algoritmo Left-looking paralelo

Una de las rutinas secuenciales estándar más extensamente usadas para la factorización de matrices dispersas es la MA48. Desarrollada por Duff y Reid (1993) [60], esta rutina se distribuye en la librería comercial HSL, por *Harwell Laboratory*. Ajustándose al estándar, la rutina presenta las etapas de reordenación, análisis, factorización y resolución triangular, las tres últimas de ellas asociadas a las rutinas en doble precisión MA48AD, MA48BD y MA48CD, respectivamente, y sin la "D" final para simple precisión. Estas subrutinas MA48 llaman a las rutinas MA50AD, MA50BD y MA50CD donde reside realmente en núcleo computacional del análisis, factorización y resolución respectivamente.

Centrándonos en la etapa de factorización, la rutina MA50BD implementa un algoritmo *left-looking* disperso que incluye pivoteo parcial de filas para asegurar la estabilidad numérica[6]. Un esquema de esta rutina se presenta en el ejemplo 3.3, utilizando la misma terminología que en la figura 1.8 de la sección 1.3.2.

Se puede apreciar como el algoritmo realiza dos funciones en cada iteración del bucle más externo de índice `k`:

- Factorización simbólica (líneas 3, 7 y 10): Etapa inicial, en la que se predice el patrón final de la columna `k`, que llamaremos columna `S`. Esta predicción (línea 3) no consume más tiempo que la posterior factorización numérica de dicha columna. Esto se consigue aplicando una búsqueda primero en profundidad según demuestran Gilbert y Peierls en [74]. De esta búsqueda obtenemos el orden topológico que determina la secuencia de actualizaciones a realizar en la línea 5. La factorización simbólica se optimiza si además utilizamos una técnica de podado

[6]La preservación del coeficiente de dispersión se contempla principalmente en la rutina MA50AD.

(línea 10) desarrollada por Eisenstat y Liu [63]. Esta técnica reduce la cantidad de información estructural requerida en la factorización simbólica. En esta etapa también se contempla un pivoteo parcial por filas para mantener la estabilidad numérica (línea 7).

- Factorización numérica (líneas 5, 8 y 9): Aquí se realizan las operaciones aritméticas estrictamente necesarias para actualizar la columna, usando para ello la información generada con anterioridad en la etapa de factorización simbólica. Desde el punto de vista computacional, el bucle de la línea 4 es el que consume el porcentaje más significativo del tiempo de ejecución.

Ejemplo 3.3 MA48 LEFT-LOOKING LU

```
1   DO k=1,n                                          ! Para cada columna
2       S = A(:,k)                                    ! Columna k de A
3       Determina las columnas C = {c_r1, c_r2, ..., c_ri}   (ri < k) de L que modifican S
4       DO para cada c_r ∈ C en orden topológico
5           S = S − S(r) · c_r
6       END DO
7       Pivoteo: intercambia S(k) y S(p), donde |S(p)| = max|S(k : n)|
8       U(1 : k, k) = S(1 : k)
9       L(k + 1 : n, k) = S(k + 1 : n, k)/U(k, k)
10      Poda la estructura simbólica según el patrón final de S.
11  END DO
```

El código está escrito en Fortran 77, de forma que se descartan las estructuras dinámicas basadas en punteros. Por contra, una estructura comprimida por columnas, CCS, es apropiada ya que el llenado tiene lugar en una columna (S) a cada iteración `k`. Sin embargo, el algoritmo no puede ser *in place* para ser eficiente. Como se muestra en la figura 3.11, necesitamos una estructura CCS para la matriz de entrada, `A`, y otra para la matriz de salida, `LU`. La primera viene determinada por los vectores `AA` de coeficientes, `IRNA` de índices y `IPTRA` de punteros al comienzo de cada columna. De forma similar la estructura de salida se compone de los vectores `FACT`, `IRNF` y dos vectores de punteros, `IPTRU` e `IPTRL`. Dado que cada columna de la matriz factorizada contiene al comienzo las entradas de la matriz `U` y a continuación las de `L`, `IPTRU(i)` e `IPTRL(i)` apuntan al final de sendas subcolumnas de cada columna `i`. En la figura 3.11 se ha supuesto que la matriz de entrada tiene tres columnas (`A1`, `A2` y `A3`). Por tanto la matriz factorizada también tiene tres columnas donde cada una de ellas contiene coeficientes de `U` y de `L`. Las últimas posiciones de los arrays `FACT` e `IRNF` se utilizan a cada iteración `k` para la factorización de la columna S.

Si nos fijamos en el código del ejemplo 3.3 podemos apreciar que los bucles `DO` de las líneas 1 y 4 son secuenciales (este último debido a que las iteraciones se deben realizar en un determinado orden). Por tanto sólo podemos explotar paralelismo al procesar coeficientes de una misma columna, como ocurre en las líneas 5, 8 y 9. Es decir, esta

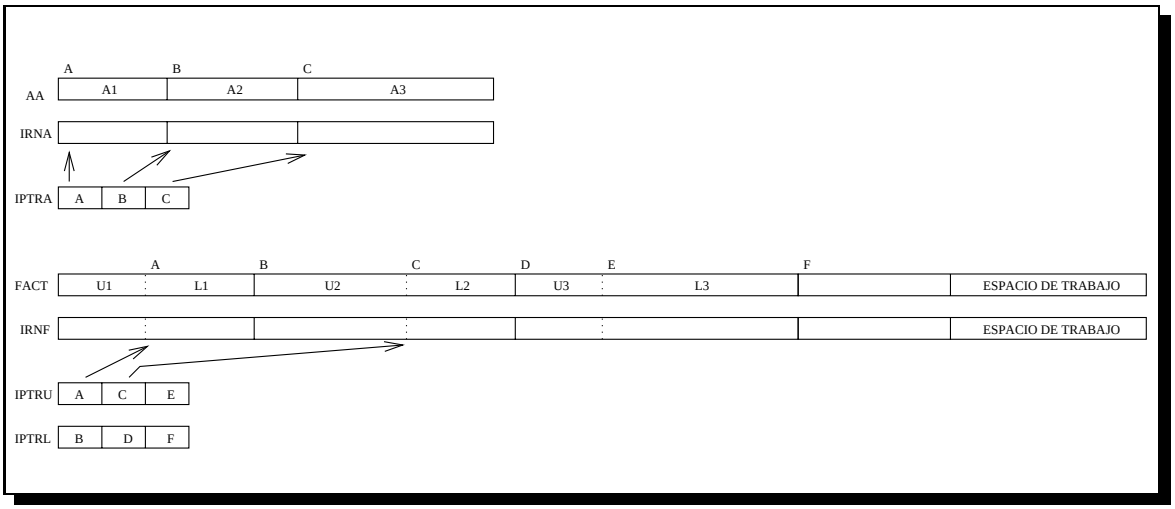


Figura 3.11: Estructura de datos utilizada en las rutinas MA50.

organización *left-looking* del algoritmo no nos permite extraer tanto paralelismo como la versión *right-looking*.

Sin embargo, hemos intentado explotar todo el paralelismo exhibido por el código. Para ello hemos utilizado una distribución BCS unidimensional por filas. Es decir, los coeficientes de la fila `i` se asignan al procesador (`i` mod P) para una topología de `P` procesadores en array. Con esta distribución, las líneas 3, 5, 7, 8 y 9 se ejecutan en paralelo. De esta forma, las operaciones `divcol()` y `actcol()` son ejecutadas por todos los procesadores para subcolumnas disjuntas de datos.

Pese a todas las variantes paralelas estudiadas y las distintas estrategias de optimización aplicadas, los resultados experimentales reportan una escalabilidad bajísima. En efecto, el procesado de una columna no presenta la suficiente carga computacional como para compensar el coste de comunicaciones implicado. Adicionalmente, la búsqueda primero en profundidad, que debe realizarse en paralelo debido a la distribución de las filas, genera una gran cantidad de comunicaciones.

Estos dos aspectos resultan en una baja eficiencia para la versión paralela de la rutina MA48. Concluimos por tanto, que este código da lugar a tiempos de ejecución muy bajos en secuencial, debido principalmente al ahorro en el movimiento de datos y aprovechamiento de la cache, pero dada su organización no ha podido ser paralelizado eficientemente.

3.5 Resolución triangular paralela

Esta sección se ocupará de presentar un algoritmo paralelo para la etapa final de resolución del sistema $LUx = b$. Aunque el tiempo consumido en la etapa de resolución triangular es dos o tres órdenes de magnitud inferior al tiempo de la etapa de factorización, en ciertas situaciones es interesante su paralelización. Por ejemplo, en algoritmos de optimización (como el Simplex [121]) o en métodos iterativos como el Gradien-

te Conjugado con precondicionador ILU (*Incomplete LU*) es necesario resolver varias etapas de substitución hacia adelante y/o hacia atrás.

En la literatura encontramos varias propuestas para resolver este problema en paralelo, aunque ninguno presenta resultados definitivos para matrices dispersas genéricas. En [12] encontramos una aproximación clásica por bloques para arquitecturas de memoria compartida. Otra idea, presentada en [7], consiste en representar la matriz `L` como un producto de factores dispersos, reemplazando la etapa de substitución por un conjunto de productos matriz-vector. Sin embargo, la estabilidad de esta técnica está limitada a matrices bien condicionadas, como se demuestra en [87]. Cuando la matriz dispersa es simétrica, el problema se simplifica sustancialmente. Gupta y Kumar [83] completan su algoritmo de factorización Cholesky multifrontal con una etapa de resolución que aprovecha el árbol de eliminación para explotar paralelismo. Utilizan una distribución unidimensional dado que deben resolver varios supernodos densos. Por tanto, para enlazar esta etapa con la de factorización (con distribución bidimensional) necesitan una etapa de redistribución. Un versión más actual, presentada en [82], evita la etapa de redistribución. Nosotros nos hemos inspirado en una de las aproximaciones discutidas en [77].

Nos centraremos en la etapa de resolución apropiada para los factores `L` y `U` que obtenemos del algoritmo de factorización discutido en la sección 3.3. Es decir, un algoritmo genérico para matrices no simétricas. Dado que la etapa de factorización conmuta a un código denso en cierta iteración, las etapas de substitución también conmutan en la misma iteración. De este modo y por orden, primero se realiza una etapa de substitución hacia adelante dispersa y luego densa, para posteriormente ejecutar la substitución hacia atrás densa y finalmente dispersa.

El algoritmo de substitución, por ejemplo hacia adelante, $Ly = b$, se resume en la siguiente ecuación:

$$y_i = \frac{1}{l_{ii}}[b_i - \sum_{j=1}^{i-1} l_{ij} y_j] \tag{3.5}$$

Como vemos, existe una dependencia de flujo en el índice `i`, ya que y_i puede ser calculada sólo cuando conocemos las componentes $y_1, \cdots, y_{i-1}$, siempre que $l_{ij} \neq 0, \quad 1 \leq j < i$. Sin embargo, cuando la matriz `L` es dispersa, muchas de las componentes l_{ij} son cero, de forma que muchas de las incógnitas quedan desacopladas y pueden ser evaluadas en paralelo. De hecho, las incógnitas $y_i, \cdots, y_{i+c}$ son independientes si los pivots $A_{i,i}, \cdots, A_{i+c,i+c}$ son compatibles y pertenecen al mismo bloque diagonal. De esta forma, podemos explotar el paralelismo asociado al producto interno expresado en el sumatorio de la ecuación 3.5, junto con el que presenta la dispersión de la matriz. En términos de programación, podemos realizar en paralelo iteraciones de los bucles de índices `i` y `j` manteniendo la distribución bidimensional de la matriz `L` a la salida de la etapa de factorización paralela.

Sin embargo, al resolver la submatriz densa, sólo disponemos del paralelismo asociado al producto interno. Por tanto, no tiene sentido mantener la distribución bidimensional de la submatriz densa ya que aumentan las comunicaciones sin obtener a cambio un incremento del número de operaciones en paralelo. Para evitarlo, antes de entrar en la etapa de resolución densa, redistribuimos la submatriz únicamente por

filas o por columnas.

Nos centraremos en la etapa de resolución dispersa por su mayor complejidad, y en particular en la fase de substitución hacia adelante, obviando la substitución hacia atrás por ser totalmente análoga. La única diferencia a tener en cuenta es que la matriz `L` es de diagonal unidad y la matriz `U` no. De esta forma, la división entre l_{ii} de la ecuación 3.5 no tiene lugar en la etapa de substitución hacia adelante.

3.5.1 Algoritmo paralelo de substitución hacia adelante disperso

Como hemos dicho, la distribución de la matriz `L` se mantiene de forma cíclica, tal y como queda al final de la etapa de factorización. Para completar el esquema de distribución debemos precisar también el tipo de datos para esta matriz `L`. En principio podemos mantener la lista doblemente enlazada (LLRCS), pero dado que en este algoritmo no hay llenado, también es posible conmutar a una estructura CRS o CCS para ahorrarnos el coste de recorrer las listas. El vector `b` estará alineado con las columnas de la matriz `L` y durante la factorización ha sufrido las mismas permutaciones que la matriz `A`.

Si elegimos una organización por columnas del algoritmo, éste tendrá el pseudocódigo mostrado en el ejemplo 3.4, descrito gráficamente en la figura 3.12.

Ejemplo 3.4 SUBSTITUCIÓN HACIA ADELANTE DISPERSA

```
1   ŵ(1 : m̂) = 0
2   DO j=1,n̂                                          ! Para cada columna
3       IF (Soy propietario del pivot l_jj) THEN
4           WHILE(lleguen_mensajes_horizontales ∧ falta_pw(j))
5               recv(pw(i), i ≥ j)                     ! Recepción Horizontal
6               y_i = b_i − pw(i)
7           END WHILE
8           broadcast(y_j)                             ! Radiación Vertical
9       ELSE
10          recv(y_j)                                  ! Recepción Vertical
11      END IF
12      ŵ(i) = ŵ(i) + l̂_ij · y_j,   l̂_ij ≠ 0  ∧  i > j.
13      IF(∄ l̂_ij' ≠ 0, j' > j ∧ i > j) THEN
14          send(ŵ(i))                                 ! Envío Horizontal
15      END IF
16  END DO
```

Al comienzo, un vector local $\hat{w}$ alineado con las filas se inicializa a cero. Este vector acumulará las aportaciones locales al sumatorio de la ecuación 3.5, siendo $\hat{w}(i) = \sum_j \hat{l}_{ij} y_j$. Cuando un procesador no contiene más aportaciones a alguna componente

$\hat{w}(i)$, envía esta última al procesador propietario del pivot l_{ii}, el cual recibe el mensaje en la componente $pw(i)$.

Con más detalle, cada procesador ejecuta un bucle externo de índice `j` que recorre las columnas locales. El procesador propietario de un coeficiente de la diagonal, l_{jj} debe esperar a recibir las sumas parciales, `pw(j)`, del producto interno ($\sum_{\hat{k}} \hat{l}_{j\hat{k}} y_{\hat{k}}$) calculadas en otros procesadores (líneas 4 a 6). Las variables con sombrero indican que son locales al procesador. En esta espera también se reciben sumas parciales correspondientes a otras filas, de forma que se anticipa este trabajo. Cuando se ha completado la reducción en la fila `j`, y_j toma su valor final y debe ser radiado a los procesadores de la misma columna de la malla (línea 8). Cuando todos los procesadores de esta columna tienen y_j, cada uno de ellos recorre su columna local de `L` actualizando las posiciones que les corresponden del vector `w` (línea 12). Si en la iteración `j`, no existe ninguna entrada local $\hat{l}_{ij'}$ con $j' > j$ se envía el valor $w(i)$ al procesador propietario del pivot l_{ii} (líneas 13 y 14). Es importante hacer notar que las comunicaciones son asíncronas de forma que se solapan comunicaciones con computaciones.

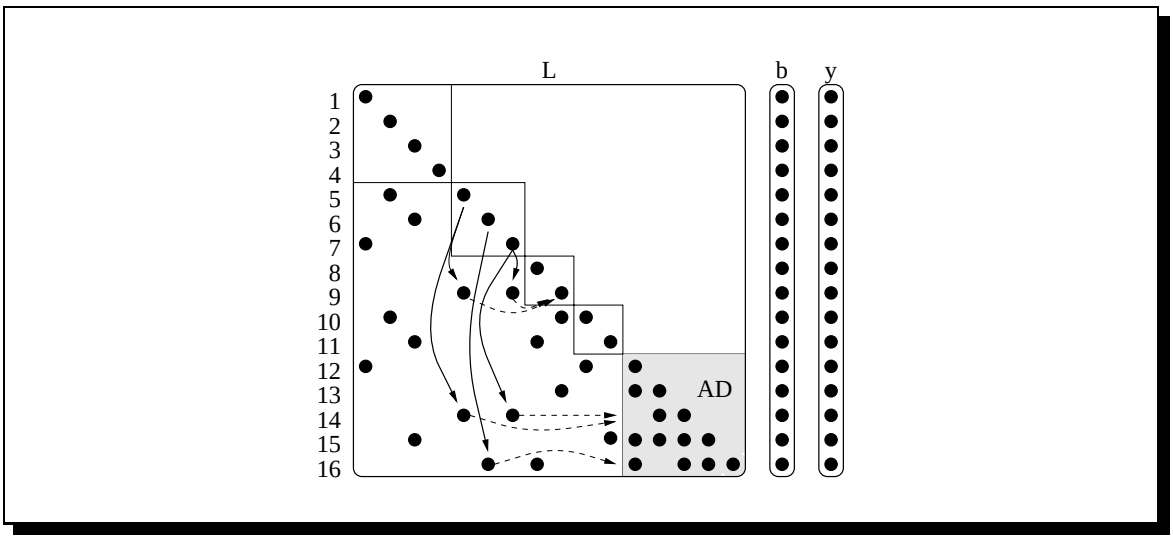


Figura 3.12: Flujo de datos en la substitución hacia adelante.

El único punto que queda por aclarar de este pseudocódigo es cómo determina el procesador propietario de la diagonal cuantas sumas parciales del producto interno va a recibir. Es decir, cuantos mensajes horizontales con $pw(j)$ debe recibir, para concluir que y_j tiene su valor definitivo. Para ello, durante la etapa de factorización los procesadores anotan cuando tienen alguna entrada en la fila `j`. Una única reducción previa a la etapa de resolución permite calcular cuantos procesadores van a enviar su contribución al procesador propietario de l_{jj}.

3.6 Validación experimental

Los algoritmos presentados no dan por sí sólos una idea completa de sus prestaciones a no ser que se validen experimentalmente. Es decir, han de ser evaluadas las carac-

terísticas que cuantifican la bondad del algoritmo, a saber: precisión numérica en los resultados, tiempo de ejecución secuencial y paralelo en función del número de procesadores implicados y cuantificación del llenado que provocan en la matriz factorizada, principalmente. Para ello se selecciona un conjunto de matrices asimétricas de patrón genérico de distintas dimensiones, densidad, y procedencia. Utilizando estas matrices como entrada, el programa se lanza a ejecución varias veces, cambiando los parámetros de entrada definidos por el usuario. De esta forma podemos inferir el comportamiento del algoritmo en función de los valores de los mencionados parámetros. Los resultados de estos experimentos son los que se presentan en esta sección.

El código está escrito en lenguaje C, utilizando las rutinas SHMEM de Cray para el pase de mensajes. La versión secuencial del programa se obtiene de simplificar el programa paralelo, eliminando todas las sentencias redundantes o no ejecutadas cuando $P = 1$ y $Q = 1$.

3.6.1 Llenado y estabilidad

Los dos parámetros de entrada que podemos ajustar para controlar el llenado y la estabilidad de la matriz son u y a. El primero limita el número de candidatos a ser pivot mediante la ecuación 1.7. Ya hemos comentado que si $u \to 1$ reducimos el error de factorización, mientras que cuando $u \to 0$ conseguimos un menor llenado. El parámetro a se utiliza para descartar los candidatos con cuenta de Markowitz M_{ij}, tal que $M_{ij} > a \cdot M_{i_0,j_0}$, donde M_{i_0,j_0} es la cuenta de Markowitz mínima y $a \geq 1$. Si $a = 1$ sólo aceptamos pivots con cuenta de Markowitz mínima, de forma que el llenado será menor. Por el contrario, cuando mayor es a, más densas serán las matrices `L` y `U` aunque los bloques diagonales serán más grandes y podremos explotar más paralelismo.

Un estudio de la incidencia del parámetro u se muestra en la tabla 3.1. Para distintos valores de u, vemos como varía el tamaño medio del bloque diagonal, $\overline{m}$, el número de iteraciones dispersas que se ejecutan, el llenado de la matriz y el error de factorización. De entre todas las matrices que han sido factorizadas en función del parámetro u presentamos por brevedad los resultados para la matriz LNS3937. El resto de las matrices presentan el mismo comportamiento, pero LNS3937 es una de las matrices peor condicionada de forma que se aprecia mejor las variaciones del error de factorización. Los experimentos se han realizado con $a = 4$ y $ncol = 16$.

Valor de u	0.9	0.5	0.1	0.05	0.01	0.001
$\overline{m}$	5.84	6.17	6.85	6.90	7.28	8.48
Iteraciones	493	475	429	420	410	349
Llenado	283772	250748	241163	222655	216017	216196
Error	2.32E-2	1.05E-2	9.56E-3	2.27E-2	2.57E-2	4.02E-1

Tabla 3.1: Dependencia con el parámetro u para la matriz LNS3937.

En la tabla comprobamos como, cuanto menor es u mayor es el tamaño medio del conjunto de pivots compatibles lo que permite explotar más paralelismo. El mismo efecto se aprecia en la fila que indica el número de iteraciones, ya que al subir el valor

$\overline{m}$ el número de iteraciones disminuye, decrementándose el tiempo de ejecución tanto secuencial como paralelo. También vemos como el llenado disminuye al bajar u ya que existe más libertad para elegir pivots compatibles con cuenta de Markowitz pequeña. Por otro lado, el error de factorización aumenta al reducir u, por lo que es necesario elegir un valor de compromiso. Adicionalmente, se ha comprobado que si los factores `L` y `U` son más densos el error de factorización es mayor, debido a incremento en el número de operaciones aritméticas. Por ese motivo para $u = 0.1$ obtenemos el mínimo error. Los experimentos realizados corroboran que el compromiso $u \approx 0.1$ [57, 54] consigue buenos resultados en muchas de las situaciones. De cualquier modo, la mejor elección de u es dependiente del problema, por lo que para cada matriz particular pueden ser necesarias varias pruebas para encontrar el mejor valor de u.

El comportamiento del algoritmo en función del parámetro a se presenta en la tabla 3.2 para la misma matriz LNS3937, con $u = 0.1$ y $ncol = 16$.

Valor de a	10	8	6	4	2	1
$\overline{m}$	8.38	7.91	7.69	6.85	4.27	1.84
Iteraciones	361	377	384	429	687	1611
Llenado	254176	251622	245374	241163	238923	220624

Tabla 3.2: Dependencia con el parámetro a para la matriz LNS3937.

Podemos ver que para valores grandes de a, mayor es el tamaño medio del conjunto de pivots compatibles y menor el número de iteraciones necesarias en la factorización dispersa. Pero también es claro que no limitar la cuenta de Markowitz da lugar a la elección de pivots que provocan un mayor llenado. El compromiso en este caso está en mantener un valor $\overline{m}$ elevado sin provocar un llenado excesivo. En nuestros experimentos el valor $a = 4$ se muestra como una buena elección en muchas situaciones. Este valor de compromiso también se utiliza en [47, 95].

Dado que el valor de `ncol` se ajusta dinámicamente durante la ejecución del programa, su valor inicial no es especialmente significativo. De cualquier modo, hemos encontrado que un valor inicial apropiado es `ncol`=16 a partir del cual subir o bajar en función de la densidad de la matriz.

Encontrar un conjunto de pivots paralelos no es interesante sólo para la factorización en paralelo, sino que también en la ejecución en secuencial. En la figura 3.13 hacemos un estudio del tiempo de ejecución secuencial en función del valor `maxncol`. En este experimento hemos fijado `ncol=maxncol`, anulando la sección de código que actualiza `ncol` adaptativamente. Hemos escogido la segunda matriz más dispersa del conjunto (SHERMAN5) y la segunda más densa (SHERMAN2). Para incluir los tiempos de ejecución en la misma gráfica, ambos están normalizados por el tiempo obtenido cuando `maxncol=1`.

Vemos como en el caso de trabajar con matrices dispersas, incluso en el algoritmo secuencial es interesante buscar un conjunto grande de pivots compatibles. Para SHERMAN5, el tiempo secuencial fijando `ncol`=16 es un 45% menor que el que corresponde a `ncol`=1. La variable $\overline{m}$ llega a alcanzar el valor 26.7 con `ncol`=36 aunque el tiempo de ejecución ya es peor por consumir demasiado tiempo en la búsqueda de

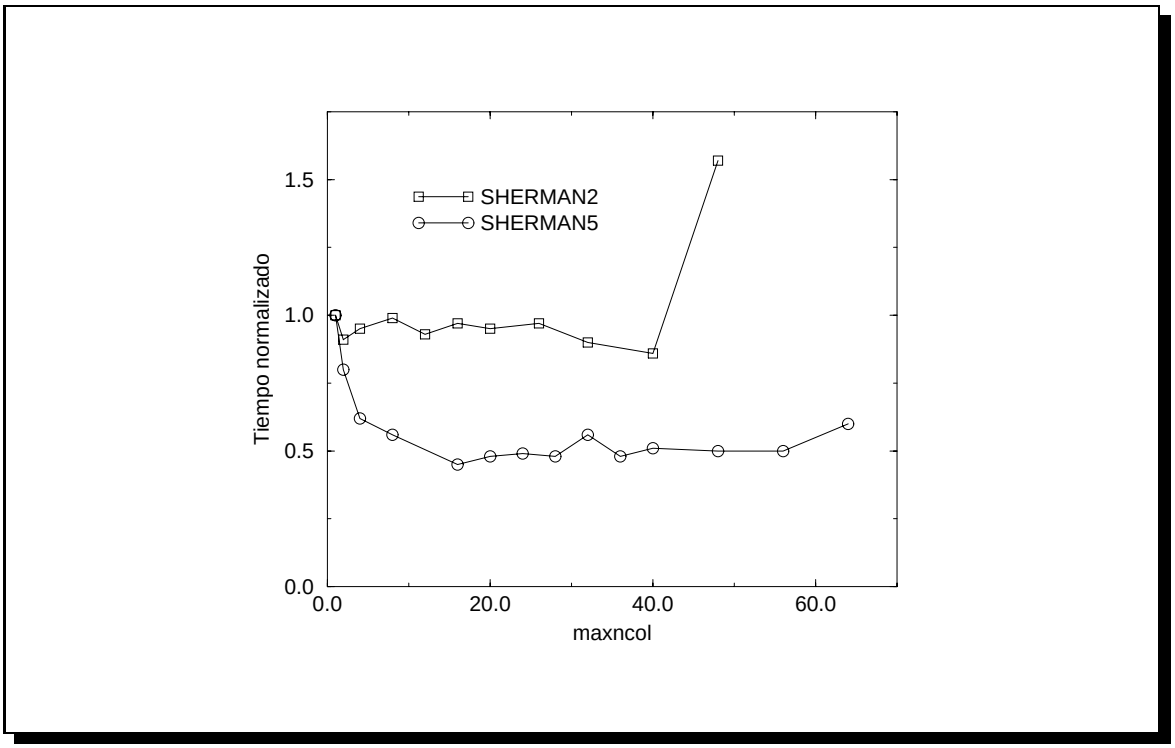


Figura 3.13: Influencia de la variable `ncol` en el tiempo de ejecución secuencial.

pivots compatibles. Sin embargo, al factorizar matrices densas, como SHERMAN2, fijar `ncol` a un valor alto no es tan ventajoso. Por ejemplo, `ncol`=48, resulta en una importante pérdida de tiempo en la búsqueda de candidatos que luego resultan ser incompatibles debido a la densidad de la matriz.

Por último haremos también un estudio de la variación del tiempo de ejecución en función de la densidad de la submatriz que queremos factorizar de forma densa. Recordemos que el parámetro de entrada `maxdens` representa la densidad de la submatriz reducida que, una vez alcanzada, implica la conmutación a un código de factorización denso. En la figura 3.14 mostramos la relación entre el valor de `maxdens` y el tiempo de ejecución para varias matrices. Para poder comparar apropiadamente los tiempos de ejecución, éstos se han normalizado. Es decir, para cada matriz, el tiempo de ejecución está entre cero y uno, ya que los hemos dividido por el tiempo en el caso peor. Este caso peor se da cuando no conmutamos a código denso (en la gráfica `maxdens`=110).

Como vemos, conmutar a código denso introduce una importante mejora en la ejecución total del algoritmo. En la gráfica se aprecia que los tiempos de ejecución mínimos se obtienen cuando `maxdens`$\approx$15%.

3.6.2 Comparación con la rutina MA48

Antes de estudiar las prestaciones del algoritmo paralelo, es importante comprobar que la versión secuencial es suficientemente eficiente. La rutina secuencial de factorización LU dispersa más extensamente usada y con la que se comparan otros algoritmos de

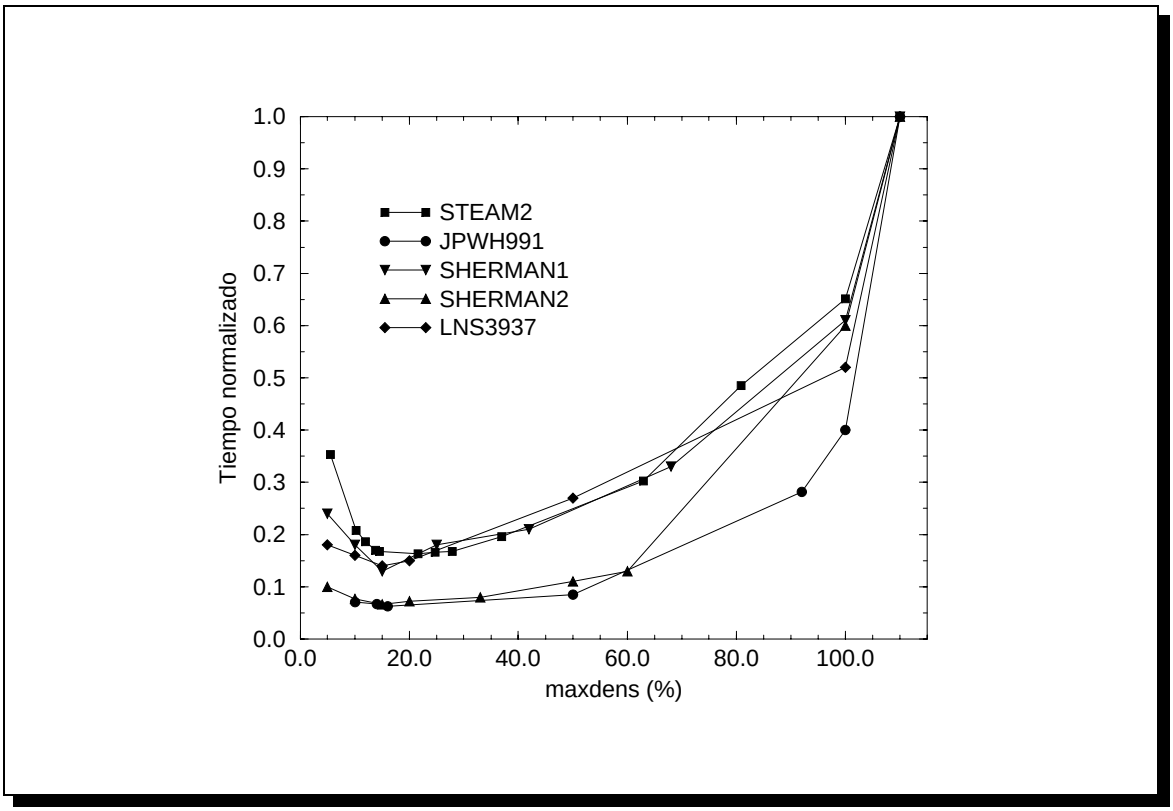


Figura 3.14: Influencia del parámetro `maxdens` en el tiempo de ejecución.

factorización dispersa es la MA48. De hecho, hasta donde alcanza nuestro conocimiento, es la única rutina comercial basada en métodos generales y la más rápida al factorizar matrices de patrón aleatorio. Estas propiedades las debe, en gran medida, a su organización *left-looking* que permite minimizar el tráfico de datos con memoria y consecuentemente un buen aprovechamiento de la cache. Sin embargo, precisamente esa organización, convierte a la rutina MA48 en un código básicamente secuencial en el que casi no podemos explotar paralelismo.

Pues bien, nuestro algoritmo de factorización secuencial permite extraer más paralelismo dada su organización *right-looking*, pero debe ser comparable a la MA48 en tiempo de ejecución y prestaciones. En otras palabras, si el algoritmo paralelo es 16 veces más rápido en 16 PE's que su versión secuencial, pero ésta última es 16 veces más lenta que la MA48 no habremos resuelto nada.

En la tabla 3.3 presentamos una comparación entre los dos algoritmos para las características más significativas: tiempo de ejecución, llenado y error de factorización. Por supuesto, los parámetros comunes están igualmente fijados en ambas versiones: u=0.1 y `maxdens`=15%.

Como vemos, la relación entre los tiempos de ejecución (MA48 entre SpLU) es mayor que uno (marcado en negrita) en cinco ocasiones. En estos casos el tiempo de factorización del algoritmo SpLU es inferior al de la rutina MA48, llegando a existir hasta un factor de 4.26 para la matriz EX10HS. Sin embargo, para las 12 matrices restantes la rutina MA48 es más rápida, aunque la relación no baja de 0.5, salvo para

	Tiempos		Error	Llenado
Matriz	**SpLU–MA48**	**relación**	**SpLU–MA48**	**SpLU–MA48**
STEAM2	1.10–.61	(0.55)	**.29E-13**–.13E-11	**79552**–110466
JPWH991	1.05–.88	(0.84)	**.18E-14**–.82E-13	**89810**–101892
SHERMAN1	.37–.19	(0.51)	.22E-12–.16E-12	43320–43171
SHERMAN2	6.87–16.7	**(2.43)**	**.69E-6**–.15E-5	**325706**–656307
EX10	7.49–24.51	**(3.27)**	**.23E-6**–.31E-6	**283378**–296270
ORANI678	9.23–6.48	(0.70)	.15E-12–.74E-13	**406568**–439280
EX10HS	10.24–43.71	**(4.26)**	**.72E-7**–.20E-6	**321031**–336832
CAVITY10	51.78–25.94	(0.50)	**.16E-9**–.36E-9	1139121–1087769
WANG1	46.99–21.18	(0.45)	.26E-12–.97E-13	1124807–808989
WANG2	49.82–21.02	(0.42)	**.47E-13**–.52E-13	1178085–808989
UTM3060	42.30–26.13	(0.62)	**.16E-8**–.58E-8	**1066896**–1073933
GARON1	69.73–35.07	(0.50)	**.17E-8**–.21E-8	1431657–1257874
EX14	131.75–206.63	**(1.56)**	**.19E+1**–.93E+1	**2293851**–2658661
SHERMAN5	8.69–11.02	**(1.26)**	**.17E-12**–.59E-12	**363186**–519855
LNS3937	34.1–25.8	(0.75)	.95E-2–.13E-2	1078221–1002494
LHR04C	101.43–14.05	(0.13)	**.89E-5**–.16E-3	1988258–870784
CAVITY16	193.46–109.86	(0.56)	.85E-9–.49E-9	2683852–2581086

Tabla 3.3: Comparación entre los algoritmos SpLU y MA48.

las matrices WANG1, WANG2 y LHR04C. Es decir, SpLU no suele duplicar en tiempo a MA48 en la mayoría de los casos. Para esta última matriz, LHR04C, el tiempo de factorización de la rutina MA48 sí es claramente inferior al de la SpLU, pero a cambio el error de factorización es casi 20 veces menor.

También es muy importante notar que el error de factorización es menor en el algoritmo SpLU para 12 de las 17 matrices factorizadas, y en los cinco casos restantes nunca hay más de un orden de magnitud de diferencia. En cuanto al llenado, éste es mayor en los factores L y U obtenidos mediante la MA48 en nueve ocasiones.

Pese a la elevada optimización del algoritmo MA48 creemos que la razón por la que puede ser mejorado en ciertas ocasiones reside en la etapa de análisis. Aunque en la rutina MA48 esta etapa de análisis también se apoya en el criterio de Markowitz y de umbral, el hecho de que sea una etapa previa a la factorización tiene sus inconvenientes. Los vectores de permutación orientados a preservar el índice de dispersión se deciden con antelación, pero en la etapa de factorización se permite un pivoteo parcial que atiende principalmente a consideraciones numéricas y que en cierto modo puede deshacer el trabajo realizado en la etapa de análisis.

En nuestro algoritmo las etapas de análisis y factorización se encuentran combinadas en una etapa de análisis-factorización donde la elección del conjunto de pivots se hace a cada iteración y sobre los candidatos de la submatriz activa. Esto permite en muchos casos hacer una mejor elección de los pivots durante la factorización, resultando en una mayor precisión en los factores. A cambio, la sección de código dedicada al análisis en nuestro código consume más tiempo que la etapa correspondiente de la MA48, debido a que no sólo busca un pivot, sino varios compatibles entre sí.

3.6.3 Prestaciones del algoritmo paralelo

En esta sección compararemos el tiempo de ejecución del algoritmo paralelo ejecutado en una malla de $P \times Q$ procesadores con la versión secuencial, ejecutada en un único procesador Alpha.

Para que los tiempos sean comparables los parámetros de entrada deben estar igualmente fijados en la versión paralela y secuencial. Como hemos visto en la subsección 3.6.1 resulta adecuado fijar `u`=0.1 y `a`=4. En cuanto al valor de `ncol`, comentamos dos aspectos. Por un lado, si queremos que las características del algoritmo sean independientes del número de procesadores, debemos inicializar el valor `ncol` inicial según la ecuación `ncol`=`ncol`/Q. De esta forma, el número global de columnas en las que se seleccionan pivots inicialmente es independiente de las dimensiones de la malla. Por otro lado, el algoritmo paralelo se adaptará dinámicamente a la matriz particular que esté factorizando, aumentando o disminuyendo `ncol` en cada procesador.

El algoritmo paralelo presenta el mismo llenado y error numérico que el algoritmo secuencial, ya que los parámetros u, a y `maxdens`, no afectan al algoritmo paralelo de forma distinta a como afectan al secuencial.

La tabla 3.4 presenta la aceleración que obtenemos al factorizar las 14 matrices más grandes de nuestro conjunto. Las tres primeras, STEAM2, JPWH991 y SHERMAN1, no se consideran por presentar un tiempo de factorización apenas superior al segundo, como vemos en la tabla 3.3. Las últimas tres columnas de la tabla presentan la dimensión, n, densidad inicial, ρ_0, y final, ρ_n. La figura 3.15 presenta la aceleración y la eficiencia conseguida al factorizar en paralelo las 9 matrices computacionalmente más costosas.

	Aceleración				Densidad		
Matriz	2	4	8	16	n	ρ_0	ρ_n
SHERMAN2	1.85	3.62	5.98	9.82	1080	1.98%	27.92%
EX10	1.74	2.99	4.25	4.96	2410	0.94%	4.87%
ORANI678	1.77	3.02	4.96	6.67	2529	1.41%	6.35%
EX10HS	1.74	3.65	5.39	5.63	2548	0.88%	4.94%
CAVITY10	1.94	3.72	5.59	8.77	2597	1.13%	16.88%
WANG1	2.16	3.76	7.01	10.44	2903	0.22%	13.94%
WANG2	2.06	4.15	6.60	12.45	2903	0.22%	13.97%
UTM3060	1.88	3.41	5.87	10.07	3060	0.45%	11.39%
GARON1	2.32	3.76	7.18	12.02	3175	0.88%	14.20%
EX14	2.18	4.04	7.22	13.17	3251	0.63%	21.70%
SHERMAN5	1.63	3.00	4.28	5.66	3312	0.19%	3.31%
LNS3937	1.93	3.69	6.33	11.01	3937	0.16%	6.95%
LHR04C	2.02	3.93	7.04	11.79	4101	0.49%	11.82%
CAVITY16	1.99	3.85	7.48	14.11	4562	0.66%	12.89%

Tabla 3.4: Aceleración para distintos tamaños de la malla.

Como podemos ver, la aceleración es monótona creciente al aumentar el número de

procesadores. Los incrementos de aceleración menos notables tienen lugar al pasar de 8 a 16 procesadores para las matrices EX10 y EX10HS. Estas matrices no presentan suficiente carga computacional como para compensar el incremento de comunicaciones que conlleva duplicar el número de procesadores. Esto da lugar a una baja relación entre el número de operaciones locales y comunicaciones. Es más, los mensajes enviados tienen pocos datos de forma que la latencia predomina sobre el ancho de banda. También es importante tener en cuenta la gran relación entre la potencia del procesador Alpha 21164 y el ancho de banda de la red de interconexión del Cray T3E.

El coste computacional de las matrices vemos que no depende únicamente de la dimensión de la matriz (como ocurre al factorizar matrices densas) sino también de su densidad inicial y sobre todo final. En efecto, las matrices EX10 y EX10HS, junto con SHERMAN5, son las únicas que no superan el 5% de densidad una vez factorizadas.

Por tanto, las matrices que alcanzan mayor aceleración en 16 procesadores son aquellas que no sólo tienen una dimensión mayor, sino también presentan un mayor llenado (diferencia entre la densidad final y la inicial). Las mejor aceleración se alcanza para la matriz CAVITY16. Esta es la matriz con mayor dimensión del conjunto y presenta una alta densidad final. Para algunas de las matrices con mayor llenado, como WANG1, WAN2 y EX14, la factorización en paralelo presenta una aceleración super-lineal incluso para cuatro procesadores.

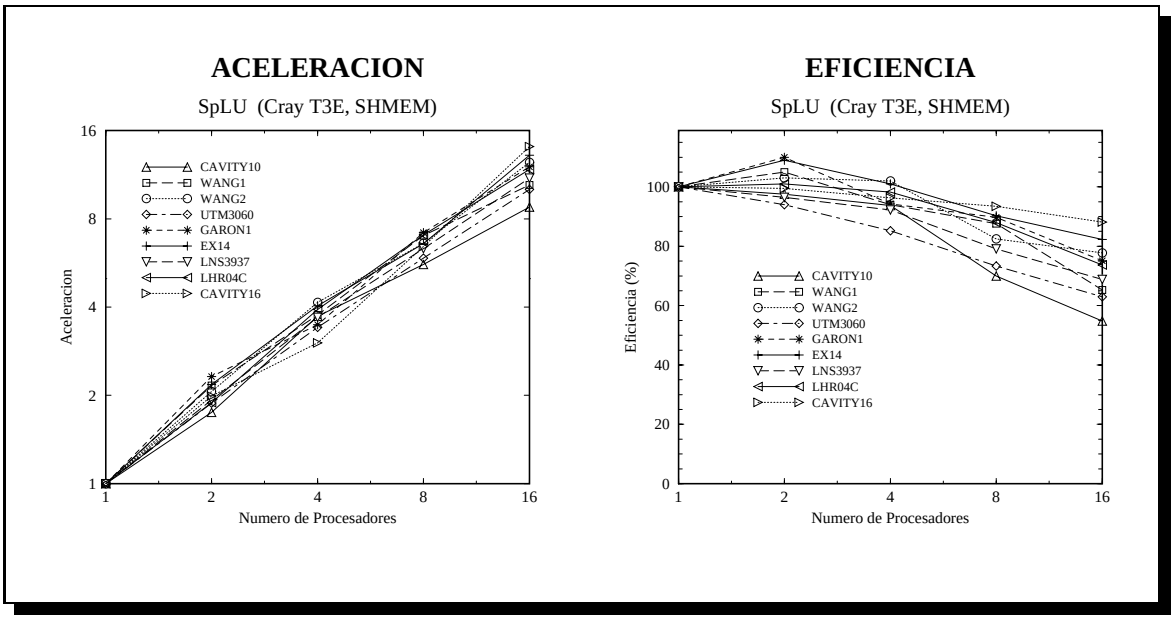


Figura 3.15: Aceleración y eficiencia del código SpLU en función del número de procesadores.

En cuanto a la etapa de resolución triangular, no hemos conseguido ninguna aceleración al ejecutar en más de cuatro procesadores. La razón estriba sencillamente en la poca carga computacional implicada en las etapas de substitución hacia adelante y atrás. Como vemos en la tabla 3.5, el tiempo consumido en la etapa de resolución triangular no es superior a las 3 décimas de segundo para ninguna de las 17 matrices factorizadas. La matriz CAVITY16 vuelve a ser la más costosa también en esta etapa, consumiendo 0.25 segundos.

Matriz	JPWH991	SHERMAN1	SHERMAN2	EX10
Tiempo	1.72E-2	2.77E-2	2.90E-2	6.22E-2
Matriz	ORANI678	EX10HS	CAVITY10	WANG1
Tiempo	4.43E-2	6.22E-2	1.09E-2	9.91E-2
Matriz	WANG2	UTM3060	GARON1	EX14
Tiempo	1.03E-1	1.04E-1	1.40E-1	1.79E-1
Matriz	SHERMAN5	LNS3937	LHR04C	CAVITY16
Tiempo	5.37E-2	1.12E-1	1.51E-1	2.54E-1

Tabla 3.5: Tiempo de ejecución secuencial de la etapa de resolución triangular.

El trabajo con mejores resultados y más actual que conocemos, para la etapa de resolución triangular por métodos generales, se describe en [77]. Como ya comentábamos en la sección 3.5, nosotros nos hemos inspirado en uno de los algoritmos presentados en dicho trabajo, que los autores llaman *data driven flow*. Las diferencias principales con su algoritmo son: (i) nuestro algoritmo está orientado "por columnas" en lugar de "por filas"; (ii) la etapa de preprocesado necesaria para determinar el patrón de comunicaciones está incluida en nuestra etapa de análisis-factorización[7]; (iii) nuestros algoritmos de resolución se dividen en una subetapa dispersa y otra densa; y (iv) nosotros hemos resuelto los factores L y U resultantes de la factorización de una matriz A mientras que en [77] se presentan resultados tomando como matriz L la matriz triangular inferior de A.

Así como el punto (ii) nos favorece por ocultar la etapa de preprocesamiento, los puntos (iii) y (iv) juegan en nuestra contra. El primero debido a que la etapa de resolución densa necesita una redistribución de la submatriz densa (pasar de distribución bidimensional a unidimensional). Y el segundo, debido a que los factores L y U son significativamente más densos que la propia matriz original. Esto implica una mayor dependencia entre iteraciones lo que reduce el grado de paralelismo. En sus experimentos, para la matriz simétrica BCSPWR10 de n=5300 y ρ=0.077%, el tiempo de resolución en un array lineal de procesadores en el Cray T3E, supera al tiempo secuencial cuando el número de procesadores es superior a 16.

De cualquier modo, tener la etapa de resolución triangular paralela nos resulta beneficioso, ya que aunque el tiempo de resolución paralelo sea similar al secuencial, en el primer caso nos ahorramos la colección de los factores a un solo procesador. Es más, podremos resolver sistemas que no puedan ser almacenados en la memoria de un solo procesador. También creemos que para sistemas significativamente más grandes, la carga computacional puede llegar a compensar el coste de comunicaciones con un mayor número de procesadores. Todo esto no quita que sigamos trabajando en intentar reducir el coste asociado a las comunicaciones en esta etapa de resolución, por ejemplo, utilizando una distribución cíclica por bloques en lugar de cíclica pura.

[7]Gracias a que nuestra rutina de resolución está incluida como una subrutina más dentro del programa de resolución de sistemas de ecuaciones dispersos.

Parte II

Compilación

Capítulo 4

Soporte HPF-2 para códigos dispersos

Los esfuerzos de investigación y desarrollo en paradigmas de paralelismo de datos que simplifiquen la tarea de construir eficientes algoritmos paralelos, se han incrementado en las últimas décadas. De entre los distintos lenguajes de paralelismo de datos disponibles se erige como estándar el HPF-2 (*High-Performance Fortran 2*) [40, 85, 86, 104].

En el modelo de paralelismo de datos, el usuario escribe el código usando un espacio de iteraciones global y ciertas directivas para indicar la distribución de los datos. Los compiladores de paralelismo de datos utilizan esta información, junto con la regla "computa el propietario"[1], para traducir las direcciones globales a locales e insertar las etapas de comunicación necesarias que permitan la ejecución del código en un multiprocesador acorde a un modelo SPMD.

Por ahora no es posible describir un código de factorización dispersa LU bajo un paradigma de paralelismo de datos con la tecnología actual de sus compiladores. Principalmente, son las computaciones implicadas en el llenado de la matriz y las permutaciones de filas y columnas, las que dificultan la representación de este código bajo el paradigma en cuestión.

En este capítulo abordamos el problema de la paralelización de algoritmos que presentan llenado y pivoteo, desde el punto de vista de los lenguajes de paralelismo de datos y de la compilación semi-automática. Como se ha comentado, es necesaria una estructura de datos que implica la necesidad de manejar indirecciones, referencias a datos a través de punteros y reserva dinámica de memoria. Estas características escapan a la actual tecnología de compilación de paralelismo de datos. Nuestra solución consiste en proponer un pequeño conjunto de nuevas extensiones a HPF-2 que permitan el manejo de este tipo de códigos. El compilador se apoyará parcialmente en una librería (DDLY) de rutinas necesarias para contemplar las nuevas directivas. Esta aproximación se evalúa posteriormente en el Cray T3E para el problema particular de la factorización

[1] Es decir, las asignaciones a la parte izquierda de una sentencia las realiza el procesador propietario de la variable correspondiente.

dispersa LU.

La organización del presente capítulo queda como sigue. En la introducción presentamos los lenguajes de paralelismo de datos y la motivación del trabajo presentado en este capítulo. En la segunda sección comprobamos la ineficiencia de los compiladores actuales de paralelismo de datos al manejar problemas irregulares. En esta línea de razonamiento, los dos siguientes capítulos presentan soluciones basadas en la extensión de las directivas HPF-2, para dos familias de códigos dispersos. En la sección 4.5 se aplican estas soluciones al algoritmo de factorización dispersa LU. Seguidamente se describe el compilador capaz de procesar las nuevas directivas introducidas. Las prestaciones de los algoritmos que hemos implementado siguiendo esta aproximación se presentan en la sección 4.7. Finalmente se enumeran otros problemas que pueden ser abordados con este paradigma de paralelismo de datos si contemplamos las extensiones propuestas.

4.1 Introducción

Aparte del mencionado HPF-2, existen otros muchos lenguajes de paralelismo de datos populares como son el CM-Fortran [142], Fortran D [67], Vienna-Fortran [160], y Craft [111].

Todos estos lenguajes se han orientado inicialmente hacia problemas regulares, esto es, códigos bien estructurados que pueden ser paralelizados eficientemente en tiempo de compilación, usando sencillas distribuciones de datos y computaciones. Sin embargo, la situación es radicalmente diferente en códigos irregulares, donde los patrones de acceso a datos y/o la cantidad de trabajo se conoce únicamente en tiempo de ejecución. En la literatura podemos reconocer, principalmente, dos aproximaciones que permiten manejar este tipo de códigos irregulares. La primera desarrolla técnicas en tiempo de compilación, extendiendo el lenguaje paralelo con nuevas construcciones capaces de expresar paralelismo no estructurado. Con esta información el compilador puede realizar en tiempo de compilación algunas optimizaciones, generalmente delegando el resto de ellas en librerías en tiempo de ejecución. La segunda aproximación contempla las técnicas en tiempo de ejecución, donde el paralelismo no estructurado es explotado y manejado completamente en tiempo de ejecución.

Respecto a la primera estrategia, los lenguajes Fortran D y Vienna-Fortran, por ejemplo, incluyen cierto soporte para distribuciones de datos irregulares. En Fortran D, el programador puede especificar una asignación de elementos de un array a procesadores, usando para ello otro array. Vienna-Fortran, por otro lado, permite al programador definir funciones para especificar la distribución irregular. Sin embargo, HPF no soporta directamente ninguna de estas construcciones. Este hecho ha sido reconocido por algunos investigadores, que han propuesto ciertas extensiones en HPF para intentar corregir este defecto [41, 107], así como por el propio Foro HPF que tomó la decisión de desarrollar la versión 2.0 de HPF [86]. Esta segunda versión de HPF ha sido mejorada y proporciona ahora una distribución por bloques generalizada (GEN-BLOCK), donde las particiones contiguas de arrays pueden ser de tamaños diferentes, y una distribución indirecta (INDIRECT), donde se define un array (llamado en inglés

mapping array) para especificar cualquier asignación arbitraria de coeficientes del array a procesadores.

Sin embargo, otros investigadores prefieren la segunda aproximación y proponen técnicas en tiempo de ejecución que automáticamente manejan la distribución de datos definida por el usuario, reparten las iteraciones de los bucles y generan las comunicaciones oportunas. La mayoría de estas soluciones se basan en el paradigma *Inspector-Ejecutor* [115, 114, 36].

En cualquier caso, las construcciones incluidas actualmente en estos lenguajes y las librerías en tiempo de ejecución que las soportan están pobremente desarrolladas, resultando en bajas eficiencias cuando se aplican a un espectro amplio de códigos irregulares, como los que aparecen en la mayoría de las aplicaciones científicas y de ingeniería. Para contribuir a la solución de este problema fueron desarrollados y validados extensivamente un cierto número de esquemas de distribución de datos pseudo-regulares (BRS y MRD), diseñados como extensiones naturales de las distribuciones regulares [17, 23, 127, 147, 152]. La clave de estos esquemas de distribución es su simplicidad para ser incorporados en un lenguaje de paralelismo de datos y ser usados por el programador. Adicionalmente resultan de gran efectividad y obtienen altas eficiencias en la paralelización de códigos irregulares.

Sin embargo, las estructuras de datos contempladas bajo los esquemas de distribución mencionados resultan poco efectivas a la hora de manejar operaciones de llenado y pivoteo. Por lo tanto, presentamos en este capítulo las aportaciones necesarias para permitir la gestión del llenado y pivoteo en matrices dispersa en un entorno de paralelismo de datos. Como ejemplo, la factorización LU dispersa, que presenta este tipo de operaciones, se podrá describir bajo este paradigma. Claramente, tendremos que utilizar el lenguaje Fortran, en lugar de C, para describir los códigos correspondientes.

4.2 Una implementación inicial con Craft

Craft [111] es el nombre del compilador de paralelismo de datos disponible en las plataformas masivamente paralelas de Cray Research. Básicamente, consiste en el lenguaje Fortran 77 más algunas extensiones del Fortran 90, como la sintaxis de arrays y funciones intrínsecas para realizar operaciones globales con datos compartidos. También incluye un conjunto de directivas (que deben comenzar con la palabra clave `CDIR$`) para distribuir los datos. Aparte del modelo de paralelismo de datos, Craft también soporta otros paradigmas (pase de mensajes, datos compartidos y trabajo compartido) que se pueden combinar en un mismo programa. Más recientemente, Cray ha desarrollado un híbrido entre Craft y HPF que llaman HPF_Craft [46].

Como primera aproximación al estilo de programación de paralelismo de datos, intentamos implementar un algoritmo LU disperso simplificado (sin pivoteo de columnas ni reserva dinámica de memoria por estar escrito en Fortran77) mediante Craft para el Cray T3D. Básicamente, el algoritmo secuencial sigue una política *right-looking* y utiliza una estructrua de datos estática como la CCS (ver sección 1.2.2). Para gestionar el llenado es inevitable un movimiento de columnas en cada iteración que permita la inclusión de las nuevas entradas generadas durante la actualización. La estrategia des-

crita en [57, Cap. 2.4] se aplica para actualizar cada columna (`actcol()`). Aunque se ha mencionado en la sección 3.3.4, recordemos que consta de tres etapas: descompresión del vector empaquetado en uno denso (operación *scatter*); actualización del vector denso; y empaquetado de los elementos no nulos de éste último (operación *gather*). Las operaciones de *scatter* y *gahter* se muestran en el ejemplo 4.1 (a) y (b) respectivamente. Este algoritmo se detalla en el capítulo 5 y en [145].

Craft sólo proporciona la distribución clásica `BLOCK(x)` para matrices densas, donde `x` indica el tamaño del bloque contiguo del array que se asigna a cada procesador, normalmente conocida como distribución cíclica por bloques. Si utilizamos este esquema para distribuir los tres vectores, `Val`, `Fil` y `Colpt` que representan una matriz dispersa, `A`, con formato CCS, el primer resultados es una alta perdida de localidad. En efecto, la distribución cíclica por bloques no atiende a las coordenadas de las entradas y asigna trozos de columnas de tamaño variable a procesadores. En cuanto a la distribución del trabajo, encontramos un problema similar. Por ejemplo, en el código del ejemplo 4.1 (a), la directiva `DO SHARED` distribuye el trabajo del bucle, asignando la iteración `i` al procesador que tenga asignado la componente `Val(i)`. Esta sección de código transforma una columna comprimida de la matriz `A` en un vector denso, `Dens`. La sentencia de asignación con una indirección en la parte izquierda dará lugar a una pérdida de localidad inevitable, ya que aunque garanticemos que `Val(i)` esté alineado con `Fil(i)`, no podremos hacer lo mismo con `Dens(Fil(i))`.

Ejemplo 4.1 Extractos del código LU en Craft

```
CDIR$ DO SHARED(i) on Val(i)
DO i = Colpt(k), Colpt(k+1)-1
   Dens(Fil(i)) = Val(i)
ENDDO
```

(a)

```
induc = pos
DO i = 1, n
   IF (Dens(i) .NE. 0.0) THEN
      Val(induc) = Dens(i)
      Fil(induc) = i
      induc = induc + 1
   ENDIF
ENDDO
```

(b)

Otro problema de similar importancia consiste en la imposibilidad de paralelizar en Craft ciertos bucles que son paralelizables aplicando la distribución apropiada, por ejemplo BCS. El código del ejemplo 4.1 (b) realiza la operación opuesta a la comentada anteriormente: empaqueta los elementos no nulos de un vector denso en un vector comprimido. El código es paralelizable si los vectores `Val` y `Fil` se distribuyen cíclicamente en función de las coordenadas almacenadas en `Fil`, como ocurre en la distribución BCS. De esta forma un vector denso, `Dens` distribuido cíclicamente quedaría automáticamente alineado. Sin embargo Craft reporta resultados erróneos al intentar ejecutar ese bucle en paralelo con su distribución regular, principalmente debido a la variable de inducción `induc` que se incrementa dentro de una cláusula condicional. Por otro lado, como se indicó en el capítulo 3, no sólo la operación `actcol()` es paralela, sino que es posible actualizar todas las columnas de la submatriz reducida en paralelo. Con la distribución BCS eso es totalmente factible, sin embargo con Craft no podemos manejar adecuadamente la variable de inducción. Estos resultado son a grandes rasgos extrapolables al resto de las herramientas de compilación semi-automática disponibles,

como Fortran-D, Vienna-Fortran, etc.

Después de este intento, infructuoso pero definitivo, queda patente la imperiosa necesidad de enriquecer la potencia de los compiladores de paralelismo de datos de forma que permitan siquiera paralelizar los bucles que son inherentemente paralelos. De cara a conseguir una buena eficiencia bajo este paradigma, la localidad y la minimización de las comunicaciones son los factores que perseguiremos con más énfasis. Con todos esos propósitos encaramos las siguentes secciones.

4.3 Cuando el problema es estático

Los problemas mencionados en la sección anterior han sido presentados bajo el prisma de la factorización LU dispersa y en general de los problemas con llenado y pivoteo. En este tipo de problemas las entradas se mueven (pivoteo) y puede aparecer llenado por lo que puede ser considerado un problema dinámico. Una simplificación que permite una primera solución resulta de obviar las características dinámicas y abordar problemas donde no cambien las coordenadas de los coeficientes ni aparezcan o desaparezcan éstos durante el procesado. En estos casos, por ejemplo, en el producto matriz dispersa por vector, el problema es estático, aunque su naturaleza siga siendo irregular. A pesar de su mayor simplicidad, este conjunto de problemas tampoco era gestionado eficientemente por los compiladores de paralelismo de datos hasta muy recientemente.

Los esquemas de distribución CRS-MRD/CCS-MRD y BRS/BRD (ver sección 3.2) se adaptan perfectamente a muchos de estos problemas estáticos, pero no estaban cubiertos por las herramientas semi-automáticas. Ujaldón y col. (1997) [152] resuelven este punto al incorporar la directiva `SPARSE` en los compiladores Vienna Fortran y HPF. Aunque esos esquemas de distribución podrían ser especificados en HPF-2 mediante la directiva `INDIRECT`, el uso de los arrays de distribución (*maping arrays*) necesarios, da lugar a varios inconvenientes, entre ellos: el alto consumo de memoria; la necesidad de realizar todo el análisis de optimizaciones en tiempo de ejecución; y principalmente, el incremento de las comunicaciones que tiene lugar cuando los arrays de distribución están repartidos entre los procesadores. Sin embargo, el manejo de los esquemas de distribución pseudo-regulares son mucho más eficientes, ya que una simple operación de módulo o de comparación es suficiente para localizar cualquier elemento.

Con esta nueva directiva `SPARSE`, el esquema de distribución BCS puede especificarse como se indica en el ejemplo 4.2.

Ejemplo 4.2 Especificación HPF del esquema BCS

```
      REAL:: Val(alfa)
      INTEGER:: Fil(alfa), Colpt(n+1)
!HPF$ PROCESSORS, DIMENSION(P,Q):: malla
!HPF$ REAL, DYNAMIC, SPARSE(CCS(Val,Fil,Colpt)):: A(m,n)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO malla:: A
```

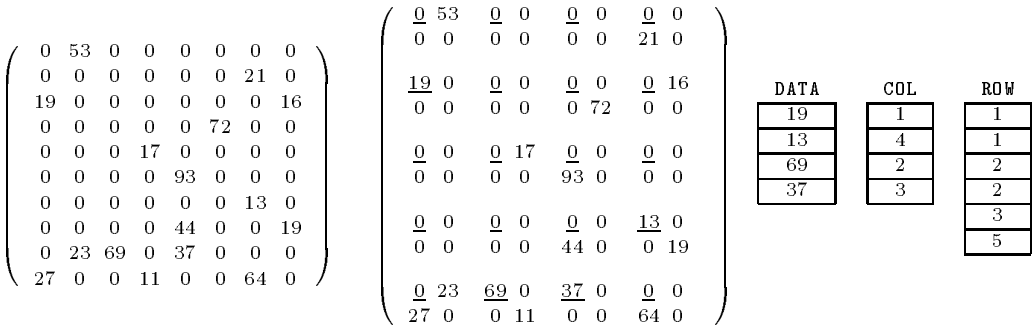
En este ejemplo, la directiva `SPARSE` informa al compilador que la matriz dispersa

`A`, de dimensión $m \times n$ y `alfa` entradas, esta representada en formato CCS mediante tre vectores, `Val`, `Fil` y `Colpt`. La palabra clave DYNAMIC, si acompaña a la directiva `SPARSE`, indica que la distribución se determina en tiempo de ejecución. De lo contrario, el compilador entiende que tanto la distribución como la representación local a cada procesador se puede construir en tiempo de compiliación. En la práctica, esto significa que el patrón de entradas está disponible antes de compilar el código, lo cual aunque raramente ocurre, debe contemplarse.

La directiva `DISTRIBUTE` informa de la distribución elegida para la matriz dispersa. Aplicando (CYCLYC,CYCLIC), consegimos un esquema de distribución BCS, ya que tenemos una distribución cíclica dispersa junto con una reprepresantación CCS de la matriz. De esta forma, obtenemos los beneficios de la distribución cíclica (balanceo de la carga y direccionamiento simple de los datos, lo que redunda en patrones de comunicación sencillos) aplicados a una matriz dispersa, independientemente de la estructura de datos comprimida que la representa. Otras alternativas, como (CYCLYC,:) o (:,CYCLYC) permiten distribuir las filas replicando las columnas y viceversa respectivamente. Aunque no es aplicable a nuestro problema de factorización, también está disponible la distribución (MRD,MRD) descrita en 3.2.

En el ejemplo 4.3 se muestra el resultado de aplicar esta directiva para una matriz dispersa con $n = 10$, $n = 8$, $P = Q = 2$ y $alfa = 14$. En este ejemplo la matriz dispersa (descomprimida), presentada en (a), se divide en un un conjunto de submatrices de tamaño 2×2, que son posteriormente proyectadas sobre la malla de procesadores. Por ejemplo, las entradas subrayadas de la matriz en (b), se asignan al procesador de coordenadas (0,0) de la malla. Finalmente, las submatrices locales se almacenan siguiendo el formato CRS, como se muestra en (c) para el procesador (0,0).

Ejemplo 4.3 ESQUEMA BRS SOBRE UNA MALLA DE 2×2 PROCESADORES



4.4 Y cuando el problema es dinámico

Hemos visto como la directiva `SPARSE` junto con las palabras clave CRS o CCS permiten contemplar los esquemas de distribución BRS y BCS respectivamente. Aplicado a problemas irregulares como el producto matriz dispersa por vector, esta aproximación reporta resultados hasta ahora no conseguidos por otras estrategias. Sin embargo, ya hemos comentado en la sección 1.2.2 los inconvenientes de las estructuras comprimidas por filas o columnas a la hora de manejar problemas con llenado o pivoteo. Por tanto, cuando en el problema encontramos estas operaciones dinámicas tiene sentido utilizar estructuras de datos también dinámicas, no contempladas tampoco por los compiladores de paralelismo de datos. Definir las estructuras de datos que necesitamos para una eficiente implementación de nuestro problema, y las modificiaciones necesarias para que sean contempladas por el compilador, conforman los objetivos de esta sección.

De entre los distintos compiladores de paralelismo de datos existentes, nos hemos centrado en HPF-2 para proponer sobre él las extensiones que necesitamos. Esta decisión se fundamenta en las siguientes razones:

- HPF-2 es un estándar diseñado por el *High Performance Fortran Forum* (HPFF). HPFF esta formado por grupos académicos y de la industria que trabajan en sugerir extensiones estándar en Fortran90. La intención es proporcionar un soporte para la programación paralela de altas prestaciones en un amplio espectro de plataformas, incluyendo sistemas masivamente paralelos SIMD y MIMD, asi como procesadores vectoriales. La última versión es el HPF 2.0 [86] que proporciona extensiones al actual estándar de Fortran, el Fortran95.

- La tendencia en paralelismo de datos es usar Fortran debido a su extensiva utilización en códigos científicos y por ser menos flexible que el C, lo cual simplifica el compilador. Esta excesiva simplicidad del Fortran77 nos impedía diseñar un código que realice eficientemente las operaciones de pivoteo y llenado. Sin embargo, el hecho de que HPF-2 se base en Fortran90, nos da flexibilidad para implementar estas operaciones, ya que este lenguaje permite el uso de estructuras dinámicas de memoria, punteros, reserva de memoria, etc, que son necesarios para soportar eficientemente el pivoteo y el llenado.

Pues bién, con todo lo dicho, nuestra intención es entonces extender HPF-2 de forma que contemple estructuras de datos apropiadas para manejar operaciones de llenado y pivoteo. En particular, contemplaremos las ya descritas en la sección 1.2.2: LLCS (*Linked List Column Storage*), LLRS (*Linked List Row Storage*), LLRCS (*Linked List Row-Column Storage*) y CVS (*Compressed Vector Storage*). Las tres primeras estructuras se utilizarán para representar matrices dispersas mientras que la última para almacenar vectores dispersos (arrays dispersos unidimensionales). El almacenamiento LLCS corresponde al presentado en la figura 1.4 de la sección 1.2.2. Es decir, la matriz se representa a través de un conjunto de listas enlazadas, una por cada columna. Aunque en la figura la lista es doblemente enlazada, también puede ser definida como simplemente enlazada. El esquema LLRS es análogo al LLCS pero enlazando por filas en lugar de columnas. Si enlazamos tanto por filas como por columnas tenemos la

estructura LLRCS, cuya variante doblemente enlazada se representa en la figura 1.3. Finalmente la estructura CVS permite representar un vector disperso mediante dos arrays y un escalar: el array `Val` que contiene el valor de las entradas; el array `Ind` que almacena los índices asociados; y el escalar `Tam` que indica el número de entradas. La declaración de los tipos de datos en Fortran90 se encuentra en el ejemplo 1.5 de la sección 1.2.2.

Una vez han sido descritos los esquemas de almacenamiento, podemos combinarlos con la directiva `SPARSE`, descrita en la sección anterior. Pretendemos, de esta forma, informar al compilador que una matriz dispersa (o un vector disperso), por ejemplo `A`, se representa mediante una de estas estructuras de datos particulares. Esto es, `A` representa una plantilla (en inglés *place holder*) de la matriz dispersa, y la directiva `SPARSE` establece una conexión entre la entidad lógica `A` y su representación interna (la lista enlazada, por ejemplo). Las ventajas de esta aproximación son que podemos usar las directivas estándar de HPF-2, como `DISTRIBUTE` y `ALIGN` aplicadas a la matriz `A`, y al mismo tiempo almacenar ésta usando una estructura comprimida.

La directiva `SPARSE` descrita en la sección anterior puede ser extendida fácilmente para que incorpore las estructuras de datos mencionadas. En el ejemplo 4.4 se muestra la sintaxis simplificada BNF (Backus-Naur Form) [2] de nuestras propuestas para esta directiva, donde sólo vamos a contemplar las nuevas estructuras de datos.

Ejemplo 4.4 SINTAXIS BNF DE LA DIRECTIVA SPARSE

<*sparse-directive*>::= <*datatype*>, **SPARSE** (<*sparse-content*>) :: <*array-objects*>
<*datatype*>::= **REAL** | **INTEGER**
<*sparse-content*>::= **LLRS** (<*ll-spec*>)
| **LLCS** (<*ll-spec*>)
| **LLRCS** (<*ll2-spec*>)
| **CVS** (<*cvs-spec*>)
<*ll-spec*>::= <*pointer-array-name*>, <*pointer-array-name*>,
<*size-array-name*>,
<*link-spec*>
<*ll2-spec*>::= <*pointer-array-name*>, <*pointer-array-name*>,
<*pointer-array-name*>, <*pointer-array-name*>,
<*size-array-name*>, <*size-array-name*>,
<*link-spec*>
<*cvs-spec*>::= <*index-array-name*>, <*value-array-name*>, <*size-scalar-name*>
<*link-spec*>::= **SINGLY** | **DOUBLY**
<*array-objects*>::= <*sized-array*>{,<*sized-array*>}
<*sized-array*>::= <*array-name*>(<*subscript*>[,<*subscript*>])

Las dos primeras estructuras de datos, LLRS y LLCS se definen mediante dos arrays de punteros (<*pointer-array-name*>), que apuntan al comienzo y al final, respectivamente, de cada fila (o columna), y un tercer array (<*size-array-name*>), que contiene el número de elementos por fila (para LLRS) o por columna (para LLCS). La opción <*link-spec*> especifica el tipo de enlace de la estructura enlazada: simple o doble. En cuanto a la estructura LLRCS, tenemos cuatro arrays de punteros que apuntan al co-

mienzo y al final de cada fila y columna, y dos vectores adicionales para almacenar el número de entradas por fila y columna, respectivamente.

La siguiente sentencia es un ejemplo de uso de esta directiva:

```
!HPF$ REAL, DYNAMIC, SPARSE (CVS(Ind, Val, Tam)):: V(10)
```

Donde se declara una plantilla lógica, `V`, para un vector disperso, la cual no ocupa memoria. Lo que físicamente sí ocupa memoria son los coeficientes no nulos del array (`Val`), los índices correspondientes (`Ind`), y un entero para el número de entradas (`Tam`). La función de la plantilla es proporcionar un objeto abstracto que puede ser distribuido y con el que otros objetos pueden ser alineados. El atributo DYNAMIC lo usaremos siempre con la directiva `SPARSE` ya que como se indicó anteriormente la distribución de estas estructuras debe realizarse en tiempo de ejecución cuando se conocen los índices y el tamaño de los vectores. Además, en este contexto, el atributo DYNAMIC es indicativo de que el tamaño del array puede cambiar durante la computación.

Las directivas `DISTRIBUTE` y `ALIGN` de HPF-2, pueden ser aplicadas a las plantillas dispersas con la misma sintaxis que en el estándar. Como se comentó en la sección 4.3, al distribuir la plantilla dispersa ésta es considerada como densa. En el caso de la directiva `ALIGN`, sin embargo, la semántica puede ser ligeramente distinta. Consideremos el segmento de código del ejemplo 4.5.

Ejemplo 4.5 ESPECIFICACIÓN DE UN VECTOR COMPRIMIDO REPLICADO

```
REAL, DIMENSION(10,10):: A
INTEGER, DIMENSION(10):: Ind
REAL, DIMENSION(10):: Val
INTEGER:: Tam
!HPF$ PROCESSORS, DIMENSION(2,2):: malla
!HPF$ REAL, DYNAMIC, SPARSE (CVS(Ind, Val, Tam)):: V(10)
!HPF$ ALIGN V(:) WITH A(*,:)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO malla:: A
```

El efecto de la directiva `ALIGN` en este fragmento de código es el siguiente: las entradas del vector `V` (es decir, `Val`), se alinean con las columnas de `A` dependiendo de los índices almacenados en el array `Ind`, y no de las posiciones del propio vector `Val` (como ocurriría bajo la semántica estándar para arrays densos). Ahora, la directiva `DISTRIBUTE`, replica el vector `V` en la primera dimensión del array de procesadores, `malla`, y lo distribuye en la segunda dimensión de la misma forma que fué distribuida la segunda dimensión de la matriz `A`. Así, cada fila completa de procesadores de la malla almacena una copia completa del vector `V`. Es importante subrayar que en esta operación de distribución, el vector `Ind` se utiliza como un array de índices de las entradas almacenadas en `Val`. Como podemos ver, para replicar un objeto (`V`) debe ser alineado con otro objeto que tenga una mayor dimensionalidad (matriz `A`) o una plantilla que sirva para este propósito. Aclaramos el proceso atendiendo a la figura 4.1, la cual muestra el efecto combinado de alineamiento/distribución para un caso particular. En la figura, el prefijo `loc` identifica a los arrays locales en cada uno de los cuatros procesadores de la malla.

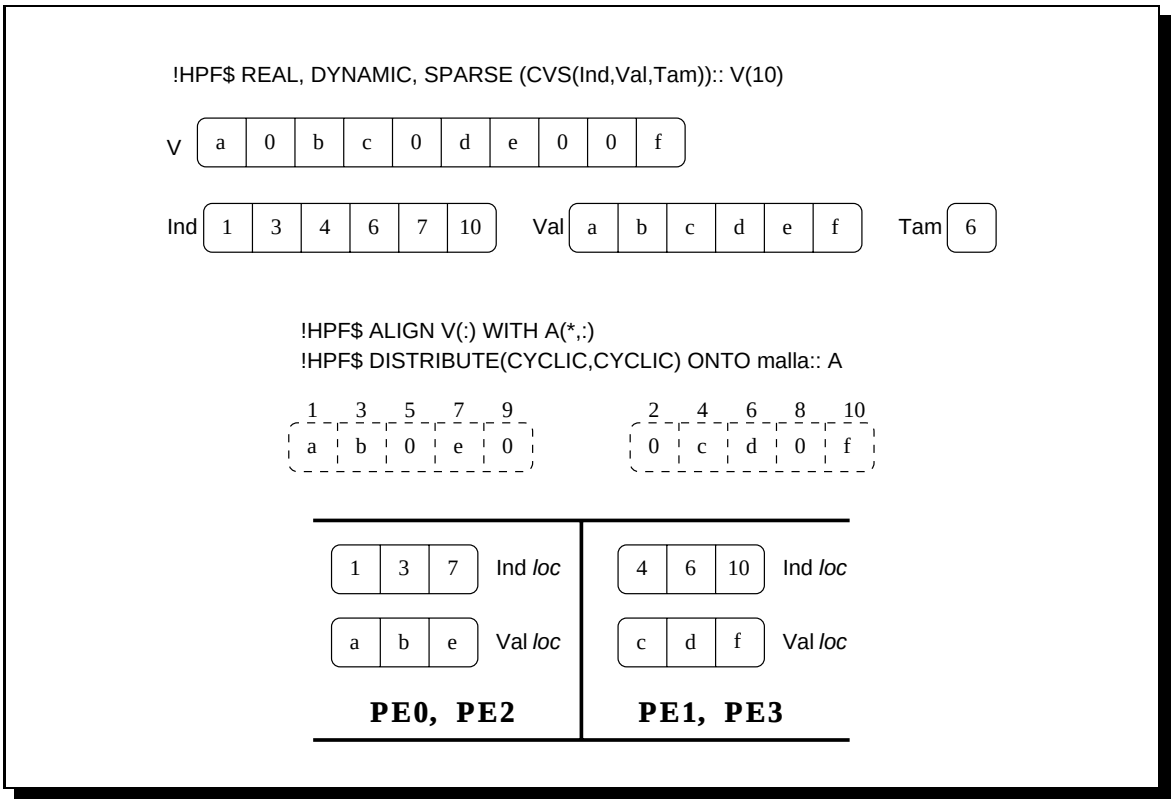


Figura 4.1: Alineamiento y distribución de una array disperso en una malla de 2×2.

4.5 Códigos en HPF-2 extendido

Mediante el uso de la directiva `SPARSE` hemos conseguido establecer un enlace entre un objeto disperso (matriz o vector) y su estructura de almacenamiento. En este punto, podríamos optar por ocultar la estructura de datos al programador, permitiéndole escribir código paralelo disperso usando notación de matrices densas. En esta propuesta inicial, discutida en [145], el compilador se hace responsable de traducir la notación densa a un código disperso y paralelo, teniendo en cuenta el esquema de almacenamiento utilizado. Sin embargo, esta aproximación supone un gran esfuerzo en la implementación del compilador, y para ser sinceros, no tenemos la medida objetiva de cómo de factible resulta tal implementación. Por otro lado, sería necesario combinar en el mismo código plantillas o *place holders* (es decir, entidades *lógicas*) con arrays *reales*, con una primera consecuencia: obtendremos resultados erróneos cuando compilemos el mismo programa para su ejecución secuencial. Esto contradice radicalmente la filosofía básica del paradigma del paralelismo de datos: la inclusión de directivas en el código sólo puede afectar al tiempo de ejecución de éste y nunca a sus resultados. Bik [29] y Pingali [97] proponen una aproximación similar, basada en la transformación de programas secuenciales densos, anotados con directivas dispersas, en los códigos secuenciales dispersos equivalentes. Sin embargo, incluso para códigos secuenciales, la implementación de tal compilador resulta tan compleja que no existe una versión disponible que maneje adecuadamente problemas reales.

Por tanto, siendo realistas, si los códigos dispersos son complejos, los programadores deben asumir parte de la complejidad al programar, dando pistas al compilador para que pueda realizar un buen trabajo. Mirando a través de ese prisma, la aproximación consiste en forzar al programador a usar explícitamente la estructura de datos que representa la información, y permitirle usar las plantillas únicamente con propósitos de alineamiento y distribución. Esta es la solución empleada en la versión de paralelismo de datos de la LU dispersa que presentamos a continuación. La misma aproximación es aplicada para la factorización QR dispersa en [15].

4.5.1 Código LU disperso

Presentamos los detalles de implementanción de las ideas expuestas apoyándonos en el algoritmo de factorización LU disperso. Más precisamente nos centraremos en la etapa de factorización con pivoteo parcial numérico, por lo que ejecutaremos la etapa de análisis en un paso anterior, la cual consume un tiempo despreciable.

Esta organización con pivoteo parcial elimina la necesidad de acceder tanto por filas como por columnas. De este modo, si optamos por un almacenamiento LLCS, donde enlazamos los elementos de una misma columna en una lista unidimensional, la permutación de columnas se puede llevar a cabo eficientemente. Con esta estructura de datos, como se discute en 1.2.2, ahorramos memoria, maneteniendo al nivel deseado la flexibilidad en el acceso y la versatilidad en la generación y movimiento de entradas.

Una vez elegida la estructura de datos, y para completar el esquema de distribución, paralelizaremos el código LU mediante una simple distribución cíclica de columnas sobre un array lineal de procesadores. En el ejemplo 4.6 se muestra la sección declarativa del código LU paralelo aplicando las extensiones propuestas en HPF-2.

Ejemplo 4.6 SECCIÓN DE DECLARACIÓN

```
INTEGER, PARAMETER :: n=1000
TYPE (ptr), DIMENSION(n):: first(:), last(:), vpiv(:)
INTEGER, DIMENSION(n):: vsize(:)
REAL, DIMENSION(n):: vcolv, vmaxval
INTEGER, DIMENSION(n):: vcoli, iq
INTEGER :: sizec,
REAL, PARAMETER:: u=0.1  !*** Parámetro de umbral

!HPF$ PROCESSORS, DIMENSION(P):: linear
!HPF$ REAL, DYNAMIC, SPARSE(LLCS(first,last,vsize,DOUBLY)):: A(n,n)
!HPF$ REAL, DYNAMIC, SPARSE(CVS(vcoli,vcolv,size)):: VCOL(n)
!HPF$ ALIGN iq(:) WITH A(*,:)
!HPF$ ALIGN vpiv(:) WITH A(*,:)
!HPF$ ALIGN vmaxval(:) WITH A(*,:)
!HPF$ ALIGN VCOL(:) WITH A(:,*)
!HPF$ DISTRIBUTE (*,CYCLIC) ONTO linear:: A
```

La estructura representada por la plantilla `A` contiene inicialmente la matriz de entrada, que será leída de un fichero, y al final almacena los factores `L` y `U` (el algoritmo es *in place*). La plantilla es físicamente accesible a través de los arrays de punteros `first` y `last`, que apunta a la primera y última entrada, respectivamente, de cada columna de `A`. También hemos definido un vector disperso `VCOL`, representado mediante una

estructura de vector enpaquetado en formato CVS. Este array almacenará la columna del pivot después de normalizarla en la operación `divcol()` a cada iteración del bucle más externo.

La última sentencia de la sección de declaración distribuye cíclicamente las columnas de la matriz A sobre un array abstracto de `P` procesadores, declarado previamente con el nombre `linear` con la directiva `PROCESSORS`. Unas líneas antes, tres vectores densos, `iq`, `vpiv` y `vmaxval`, son alineados con las columnas de `A`. De esta forma, después de la distribución de `A`, estos tres vectores también quedan distribuidos cíclicamente sobre el array de procesadores. El vector `iq` representa el vector de permutaciones de columnas recomendado, por la etapa de análisis, para mantener el coeficiente de dispersión; `vpiv` es un vector de punteros a la fila del pivot y `vmaxval` se utiliza para calcular el máximo valor absoluto en la búsqueda de un pivot numéricamente estable.

Por último, el array disperso `VCOL` es alineado con las filas de `A`. Así pues, después de la distribución de `A`, `VCOL` queda replicado en todos los procesadores. La razón que justifica esta replicación es hacer visible la columna del pivot normalizada a todos los procesadores para completar en paralelo la actualización de la submatriz reducida. En efecto, como veremos más adelante, en cada iteración del bucle más externo, `k`, el procesador (`k`-1) mod P, propietario de la columna `k`, realiza la operación `divcol(k)` localmente. Por tanto, los coeficientes de la columna actualizados deben radiarse al resto de los procesadores, lo cual se realiza implícitamente si copiamos esa columna en `VCOL`, que está replicada en todos los procesadores. Queremos también subrayar que el uso, en esta sección de declaración, de un array de procesadores no es una limitación esencial. De hecho se puede extender de forma similar a una malla de procesadores, pero nuestra elección unidimensional se debe principalmente a los mejores resultados experimentales obtenidos en esta topología.

En la figura 4.2 aclaramos gráficamente la declaración del esquema de distribución propuesto para nuestro algoritmo. Podemos ver la partición final de la mayoría de los arrays implicados junto con la matriz `A`, para el caso de un array de $P = 2$ procesadores. También se supone en la figura que el valor actual de la variable índice del bucle externo `k` es cuatro. La leyenda en la esquina inferior izquierda contiene la correspondencia entre la notación empleada en la figura para nombrar los objetos y la notación usada en la sección de declaración del ejemplo 4.6.

El resto del código encargado de la factorización dispersa se muestra a continuación. Primero nos centraremos en la etapa de pivoteo, cuyo código aparece en el ejemplo 4.7. Las primeras líneas muestran la inicialización del vector `vpiv` para que apunte a la fila del pivot. El bucle que realiza esa inicialización es totalmente paralelo, lo cual se le indica al compilador mediante la directiva `INDEPENDENT`, que mantiene la funcionalidad estándar de HPF: las iteraciones se pueden ejecutar en cualquier orden (o concurrentemente) sin cambiar la semántica del bucle. Esta directiva junto con la regla "computa el propietario" da lugar a que cada procesador inicialice la porción de vector que tiene asignado, y dado que los dos vectores, `vpiv` y `first`, están alineados, dicha inicialización no requiere comunicaciones. En seguida encontramos la sentencia DO asociada al bucle más externo `k`. Considerando que la etapa de análisis ha determinado la iteración, `IterConm`, a partir de la cual compensa conmutar a un código denso, el espacio de iteraciones para el índice `k` es de uno a `IterConm`.

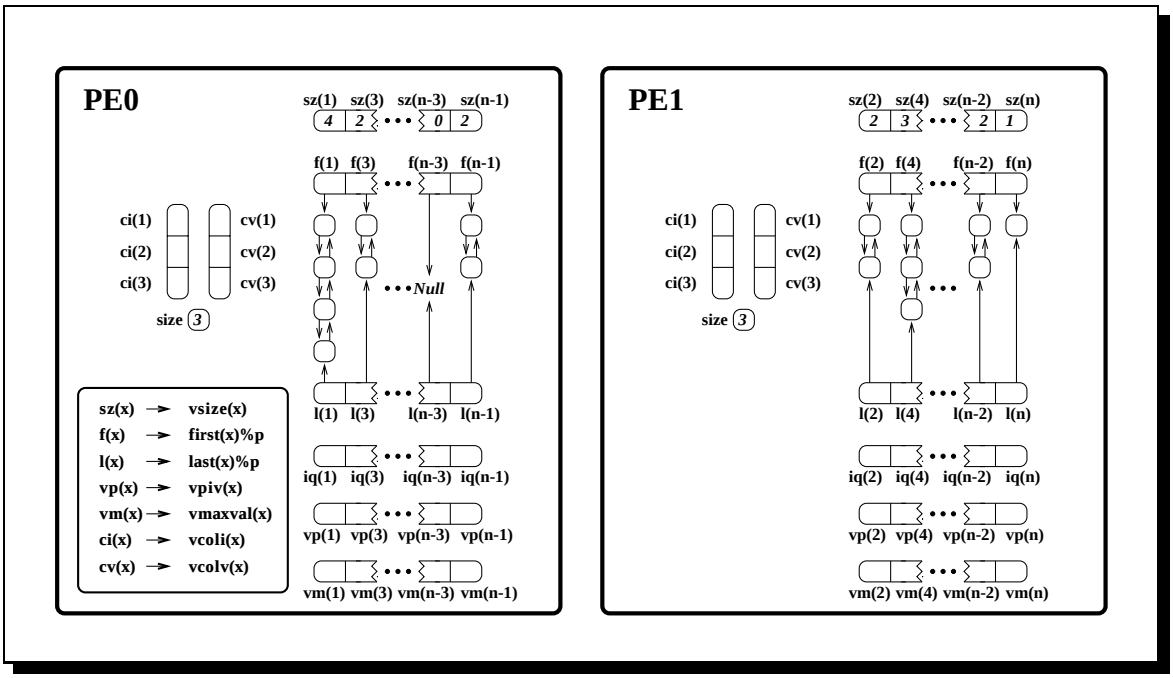


Figura 4.2: Esquema de distribución para la LU en HPF-2.

Ejemplo 4.7 Esquema del código HPF-2. 1ª Parte: Pivoteo

```
! --> Inicialización
!HPF$ INDEPENDENT
      DO j = 1, n
        vpiv(j)%p => first(j)%p
      END DO

! --> Bucle principal de la LU
main: DO k = 1, IterConm

! --> Pivoteo
! --> Búsqueda del máximo valor absoluto en la fila activa k
!HPF$ INDEPENDENT
        DO j = k, n
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
          IF (.NOT.ASSOCIATED(vpiv(j)%p)) CYCLE
          IF (vpiv(j)%p%index /= k) CYCLE
          vmaxval(j) = ABS(vpiv(j)%p%value)
!HPF$ END ON
        END DO
        maxpiv = MAXVAL(vmaxval(k:n))
        maxpiv = maxpiv*u             ! Valor umbral

! --> Elección de un pivot numéricamente estable (operación de reducción)
        actpiv = vmaxval(k)
        pivcol = k
!HPF$ INDEPENDENT, REDUCTION(actpiv,pivcol)
        DO j = k+1, n
          IF (vmaxval(j) > maxpiv .AND. iq(pivcol) > iq(j)) THEN
            actpiv = vmaxval(j)
            pivcol = j
          END IF
        END DO
        IF(pivcol /= k) THEN
! ----> Intercambio de columnas
          CALL swap(k,pivcol,first,last,vpiv,vsize,iq)
        END IF
```

La primera acción que tiene lugar dentro del bucle es la búsqueda de un pivot númericamente estable y, si puede ser, acorde con la recomendación de permutaciones en `iq` (proporcionada por la etapa de análisis) ya que ésta minimiza el llenado. Para asegurar la estabilidad numérica, el pivot debe superar un umbral, resultante de multiplicar el parámetro de entrada `u` por el máximo valor absoluto de la fila del pivot. El cálculo del máximo se realiza gracias a la función intrínseca `MAXVAL` del Fortran90, que tiene como entrada un vector, `vmaxval`, con el valor absoluto de todas las entradas en la fila del pivot activa. La actualización de `vmaxval` tiene lugar en el segundo bucle paralelo marcado con la directiva `INDEPENDENT`. Este bucle recorre la fila del pivot activa almacenando el valor absoluto de las entradas en `vmaxval`. La directiva `ON HOME(vpiv(j))` le indica al compilador que el procesador propietario de `vpiv(j)` será el encargado de la asignación de la iteración `j`, reafirmando la regla "computa el propietario". Además, esta directiva viene acompañada de la claúsula `RESIDENT(a(*,j))` que informa al compilador que toda la columna `j` es local al procesador propietario de `vpiv(j)`, y por tanto, no es necesaria ninguna etapa de comunicación.

Una vez tenemos el valor umbral, seleccionamos el pivot que lo supere y que, por otro lado, respete el coeficiente de dispersión de la matriz. Para ello, tendremos en cuenta que en la etapa de análisis se recomienda que la columna `r` preceda a la columna `s` si $iq(r) < iq(s)$. Seleccionar el pivot que cumpla esas condiciones es equivalente a una reducción, lo cual es indicado al compilador añadiendo a la directiva `INDEPENDENT`, la claúsula `REDUCTION()` con las dos variables que se reducen: `actpiv`, el valor absoluto del pivot; y `pivcol`, el índice de la columna a la que pertenece dicho pivot. Este tipo de reducción definida por el usuario no está considerada actualmente por el estándar HPF-2, pero su inclusión no añade ninguna complejidad significativa a la implementación del compilador. De hecho, existen investigaciones orientadas a detectar reducciones paralelas complejas [140], donde pueden aparecer sentencias `IF` dentro del bucle de reducción. Finalmente, si la columna `pivcol` no es la columna `k`, ambas se permutan, actualizando los vectores correspondientes, mediante la rutina `swap`.

Ejemplo 4.8 ESQUEMA DEL CÓDIGO HPF-2. 2ª PARTE: DIVCOL()

```
                                    •
                                    •
! --> La columna del pivot es actualizada y empaquetada en VCOL
!HPF ON HOME (vpiv(k)), RESIDENT (A(*,k)) BEGIN
       aux => vpiv(k)%p
       pivot = 1/(aux%value)
       aux%value = pivot
       aux => aux%next
       size = vsize(k)-1

       DO i = 1, size
         aux%value = aux%value*pivot
         vcolv(i) = aux%value
         vcoli(i) = aux%index
         aux => aux%next
       END DO
!HPF END ON
                                    •
                                    •
```

En la siguiente sección de código, presentada en el ejemplo 4.8, contempla la nor-

malización de la columna del pivot por el valor de éste (operación `divcol`).

Esta operación, `divcol()`, sólo la realiza el procesador propietario de la columna `k`, lo cual indicamos mediante la directiva `ON HOME`. De nuevo, para simplificar el análisis que debe realizar el compilador y pueda generar un código más optimizado, le adelantamos que la columna `k` es local al procesador en cuestión anotando la directiva `RESIDENT`. Dado que estamos escribiendo en un vector replicado, como es `VCOL`, cualquier modificación realizada en un procesador debe verse reflejada posteriormente en todos los demás. Depende de la complejidad que derrochemos en la implementación del compilador, las comunicaciones implicadas pueden realizarse: (i) instantáneamente a cada modificación de una componente de `VCOL`; (ii) agrupadas al acabar el bucle que actualiza el vector; (iii) o incluso cuando otro procesador lee una componente de `VCOL` que no ha sido adecuadamente refrescada; obteniéndose una mayor o menor eficiencia en la ejecución del bucle en función de la opción seleccionada.

Finalmente, la sección de código donde se actualiza la submatriz activa es mostrado en el ejemplo 4.9.

Ejemplo 4.9 Esquema del código HPF-2. 3ª Parte: Actualización

```
                                    •
                                    •
! --> Actualización de la submatriz reducida de A
!HPF$ INDEPENDENT, NEW (aux,i,amul,product)
loopj:  DO j = k+1, n
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
          aux => vpiv(j)%p
          IF (.NOT.ASSOCIATED(aux)) CYCLE
          IF (aux%index /= k) CYCLE
          amul = aux%value
          vsize(j) = vsize(j)-1
          vpiv(j)%p => aux%next
          aux => aux%next
loopi:    DO i = 1, size
            product = -amul*vcolv(i)
            DO
              IF (.NOT.ASSOCIATED(aux)) EXIT
              IF (aux%index >= vcoli(i)) EXIT
              aux => aux%next
            END DO
outer_if:   IF (ASSOCIATED(aux)) THEN
              IF (aux%index == vcoli(i)) THEN
                aux%value = aux%value + product
              ELSE
! ----> Inserción en la primera posición o una intermedia
                CALL insert(aux,vcoli(i),product,first(j)%p,vsize(j))
                IF (vpiv(j)%p%index >= aux%prev%index) vpiv(j)%p => aux%prev
              END IF
            ELSE outer_if
! ----> Inserción un una posición final
              CALL append(vcoli(i),product,first(j)%p,last(j)%p,vsize(j))
              IF (.NOT.ASSOCIATED(vpiv(j)%p)) vpiv(j)%p => last(j)%p
            END IF outer_if
          END DO loopi
!HPF$ END ON
        END DO loopj
      END DO main

                                    •
                                    •
```

El cuerpo de esta actualización de la submatriz definida está delimitado por un bucle paralelo de índice `j` que recorre las columnas de la matriz. El bucle está por tanto marcado con la directiva `INDEPENDENT`, pero además se le añade la cláusula `NEW`

para indicar que las variables especificadas entre paréntesis deben ser consideradas privadas a cada iteración. Esto permite eliminar las dependencias generadas por bucle que detectaría el compilador si considera que el valor de estas variables, a la salida de una iteración, se utilizan en la iteración siguiente. Sin embargo, con la cláusula `NEW`, el compilador entiende que el valor de las variables especificadas está indefinido al comienzo del bucle y que en cada iteración se les asignará un valor nuevo e independiente, volviendo a quedar indefinidas al final del bucle. En este bucle, la directiva `ON HOME ..., RESIDENT ...` también se incluye para facilitar al compilador las tareas de análisis y generación optimizada de código.

Este fragmento de código contiene llamadas a las rutinas `append()` e `insert()` incluidas dentro de un módulo de Fortran90. Estas rutinas son necesarias para el manejo de listas y el insertado de nuevas entradas cuando hay llenado. En el ejemplo 4.10 mostramos el código de la rutina `append()`, que añade una entrada al final de una lista.

Por otro lado, la rutina `insert()`, incluye una nueva entrada al principio o en algún punto intermedio de la lista. Este módulo requiere de otro, `data_structure`, no mostrado en el ejemplo, donde se incluyen las declaraciones de los tipos de datos derivados (ya descritos) que soportan la estructura de datos enlazadas.

Ejemplo 4.10 MÓDULO FORTRAN90 PARA EL MANEJO DE LISTAS

```
MODULE list_routines
  USE data_structure
  IMPLICIT NONE

CONTAINS

  SUBROUTINE append(ind, product, fptr, lptr, nel)
    INTEGER:: ind, nel
    REAL:: product
    TYPE (entry), POINTER:: fptr, lptr

    TYPE (entry), POINTER:: new
    ALLOCATE(new)
    new%index = ind
    new%value = product
    nel = nel + 1


    IF (.NOT.ASSOCIATED(lptr)) THEN
      NULLIFY(new%prev, new%next)  ! Lista vacía
      fptr => new
      lptr => new
    ELSE
      new%prev => lptr
      NULLIFY(new%next)
      lptr%next => new
      lptr => new
    END IF
  END SUBROUTINE append

        ...

END MODULE list_routines
```

Es importante resaltar que en caso de compilar este código mediante un compilador de Fortran90 (es decir, si las directivas HPF-2 son consideradas simples comentarios), el código se ejecuta correctamente en secuencial.

4.5.2 Conmutación a código denso

Cuando el índice `k` alcanza la iteración `IterConm`, el bucle principal de factorización dispersa concluye y la submatriz reducida restante se procesa mediante un algoritmo denso. Es clara la posibilidad de describir este tipo de código denso y regular en un entorno de paralelismo de datos. Sin embargo, queremos presentar en esta sección la estrategia para conmutar de estructura de datos y la sencillez con que se puede abordar la transición entre un algoritmo disperso y otro denso, coexisitiendo en un mismo código, bajo este paradigma extendido con las aportaciones presentadas.

La sección declarativa debe ser extendida con las siguientes líneas:

```
      INTEGER, PARAMETER:: nd = n-IterConm
      REAL, DIMENSION(nd):: ipd
      REAL, DIMENSION(nd,nd):: densa
!HPF$ ALIGN densa(:,:) WITH A(:,IterConm+1:n)
```

La variable `nd` especifica la dimensión de la submatriz reducida que se va a factorizar según el algoritmo denso. El vector denso, `ipd`, de `nd` posiciones es utilizado durante la factorización densa para anotar las permutaciones de filas que aseguren la estabilidad numérica. Por último la matriz, `densa`, de `nd`$\times$`nd` posiciones encargada de almacenar la submatriz reducida cuando termina la factorización dispersa. No merece una explicación exhaustiva la sentencia en la que se alinean las columnas de la matriz `densa` con las `nd` últimas columnas de la matriz `A`. En el ejemplo 4.11 vemos la sección de código encargada del cambio de estructura de datos y la llamada a la rutina de factorización densa, `FactorDensa()`.

Ejemplo 4.11 ESQUEMA DEL CÓDIGO HPF-2. 4ª PARTE: DENSA

```
                                    •
                                    •
! Inicializa la matriz densa densa
      densa=0
!HPF$ INDEPENDENT, NEW (aux,i)
      DO j=IterConm+1,n
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)), BEGIN
        aux => vpiv(j)%p
        DO
           IF (.NOT. ASSOCIATED(aux)) exit
           i=aux%index
           densa(i-IterConm,j-IterConm) = aux%value
           aux=>aux%next
        END DO
!HPF$ END ON
      END DO
! Realiza la factorización densa.
      CALL FactorDensa(nd,densa,ipd)
                                    •
                                    •
```

En primer lugar, la matriz `densa` se inicializa a cero, para luego ocupar sólo las posiciones que especifica el patrón de la submatriz reducida. Para ello, un bucle `INDEPENDET` recorre las columnas de la lista enlazada y escribe en paralelo las columnas correspondientes de la matriz `densa`. Una vez hemos conmutado a la nueva estructura de datos, podemos llamar a una rutina HPF de factorización LU densa que nosotros hemos llamado `FactorDensa`. Una descripción más detallada de esta función se encuentra, como ejemplo, en [104].

4.5.3 Resolución triangular

Una vez tenemos los factores `L` y `U`, resolvemos el sistema $LUx = b$ mediante los procedimientos de substitución hacia adelante, $Ly = b$, y de substitución hacia atrás, $Ux = y$. A su vez estos procedimientos tienen su sección de código dispersa y la correspondiente densa. En esta sección, presentamos el código de la etapa substitución hacia adelante dispersa, obviando la substitución hacia atrás por su similitud y las etapas densas por su trivialidad.

De nuevo necesitamos extender la sección declarativa con unas cuantas líneas que especifiquen las nuevas estructuras de datos necesarias.

```
      REAL, DIMENSION(n):: x,y,b
      TYPE (ptr), DIMENSION(n):: vaux
!HPF$ ALIGN x(:) WITH A(*,:)
!HPF$ ALIGN y(:) WITH A(*,:)
!HPF$ ALIGN b(:) WITH A(:,*)
!HPF$ ALIGN vaux(:) WITH A(*,:)
```

Como vemos, tanto el vector `x` como el `y` están distribuidos cíclicamente en el array de procesadores. Sin embargo, el vector de términos independientes, `b` está replicado en todos los procesadores. El vector `vaux` se utiliza para permitir el acceso por filas durante el proceso, por lo que sigue una distribución cíclica. En el ejemplo 4.12 se muestra el código paralelo para la etapa de substitución hacia adelante dispersa.

Ejemplo 4.12 ESQUEMA DEL CÓDIGO HPF-2. 5ª PARTE: RESOLUCIÓN

```
                    •
                    •
! El bucle i recorre las filas
      DO i=1,IterConm
        acum=0.0
        vaux(i)%p => vpiv(i)%p%next
!HPF$ INDEPENDENT, REDUCTION(acum)
        DO j=1,i-1
          IF (ASSOCIATED(vaux(j)%p)) THEN
            IF (vaux(j)%p%index == i) THEN
              acum = acum + vaux(j)%p%value * y(j)
              vaux(j)%p => vaux(j)%next
            END IF
          END IF
        END DO
        y(i) = b(i) - acum
      END DO
```

El bucle `i` que recorre las filas es inherentemente secuencial. Para cada iteración externa se resetea el valor del acumulador `acum` y se inicializa el puntero `vaux(i)` para que apunte al primer elemento de `L` en la columna `i`. Es importante recordar, que al terminar la factorización dispersa el vector `vpiv` apunta a los elementos de la diagonal, los cuales pertenecen a la matriz `U`.

Por otro lado, el bucle `j` presenta una reducción paralela. En efecto, la variable `acum` debe acumular las contribuciones de las entradas en la fila `i` que posteriormente restaremos a `b(i)` para obtener la correspondiente componente del vector `y`. Así como ocurría en la selección del pivot, la reducción, en este caso una suma, consta de una etapa local y paralela en la que se considera la porción de fila asignada a cada

procesador, seguida de una suma global implementada por el compilador. De nuevo, esta operación no está aún contemplada por HPF-2, pero es evidente que cualquier sentencia del tipo $a = a$ `op` b, donde la operación `op` es asociativa, es susceptible de su paralelización directa y trivial.

Otra dirección desde la que abordar la resolución triangular se basa en el cambio de estructura de datos. Dado que durante esta fase no aparecen operaciones de pivoteo ni de llenado, no es necesaria una estructura dinámica de datos. Podríamos por tanto, al tiempo que inicializamos la matriz `densa`, conmutar a una estructura CCS o CRS para la parte dispersa de los factores `L` y `U`. Esta aproximación, redunda en un ahorro de memoria, pero principalmente, consigue un mejor aprovechamiento de la cache y evita los accesos a entradas mediante punteros, salvando el coste temporal asociado.

4.6 Compilación y soporte en tiempo de ejecución

En la sección 4.4 hemos propuesto ciertas extensiones a HPF-2 que deben ser soportadas ahora por un módulo de compilación correctamente integrado en un compilador HPF-2. A grades rasgos este compilador realizará las fases de análisis del código fuente, paralelización y generación del código SPMD.

Para ayudarnos en la tarea de construcción de este compilador hemos utilizado la herramienta *Cocktail* [78]. Consiste en un conjunto de utilidades apropiadas para la consecución de los distintos pasos en la construcción del compilador: un analizador léxico (*scanner*) y sintáctico (*parser*) leen el código fuente, chequean la sintaxis y generan un árbol sintáctico; el análisis semántico se realiza sobre este árbol que posteriormente se traduce a una representación intermedia; finalmente el generador de código produce el código de salida [2].

4.6.1 Soporte para compilación

El análisis léxico del código HPF-2 extendido es la primera etapa a completar por el compilador. Los caracteres (letras, dígitos y símbolos) que definen la directiva `SPARSE` especificada en el ejemplo 4.4 deben ser identificados como palabras clave del lenguaje. Para ello la secuencia de caracteres que identifica una palabra clave se asocia a un componente léxico (*token*), por ejemplo "LLRCS". El generador de analizadores léxicos de *Cocktail* se llama *Rex* (*Regular Expression Tool*) [80]. El analizador léxico generado consiste en un autómata finito determinista dirigido por tablas que reconoce y clasifica los componentes léxicos del código.

A continuación el analizador sintáctico determina si la cadena de componentes léxicos proporcionados por el analizador léxico se ajusta a una cierta gramática. Para ello se construye un árbol sintáctico según la secuencia de *tokens* en el código fuente. De entre los tres generadores de analizadores sintácticos que ofrece *Cocktail* (*Lalr*, *Ell* y *Lark* [79, 81]) hemos usado el primero, el cual procesa gramáticas del tipo *LALR(1)* (*lookahead-LR*) [2].

En los ejemplos 4.13 y 4.14 se muestra el contenido del fichero que especifica la

sintaxis asociada a la directiva SPARSE. En el primero encontramos una sección etiquetada con la clave TOKEN donde se definen los componentes léxicos y su codificación (coherente con la que devuelve el analizador léxico). La sección siguiente, OPER, especifica la precedencia y asociatividad de los operadores para resolver ambigüedades. Estos operadores son necesarios para contemplar las expresiones enteras que pueden acompañar al atributo DIMENSION.

Ejemplo 4.13 FICHERO DE SINTÁXIS DE LA DIRECTIVA SPARSE (1)

```
SCANNER shpf PARSER hpf

TOKEN
  Identifier     = 1
  '('            = 2
  ')'            = 3
  ','            = 4
  '::'           = 5
  Int_constant   = 6
  '!HPF$'        = 15
  'DYNAMIC'      = 16
  'DIMENSION'    = 17
  'SPARSE'       = 10
  'LLRS'         = 20
  'LLCS'         = 21
  'LLRCS'        = 22
  'CVS'          = 23
  'SINGLY'       = 30
  'DOUBLY'       = 31
  'REAL'         = 40
  'INTEGER'      = 41
  '+'            = 50
  '-'            = 51
  '*'            = 52
  '/'            = 53
  '**'           = 54

OPER
  LEFT '+' '-'
  LEFT '*' '/'
  RIGHT '**'
  LEFT unary_defined_operator
                                        .
                                        .
                                        .
```

En el ejemplo 4.14 se muestra la última sección del fichero, RULE, que contiene las reglas de la gramática. Estas reglas están expresadas en notación *EBNF* (*Extended BNF*) pero son convertidas internamente a la notación BNF que acepta el generador *Lalr* [81].

Una vez se ha completado la etapa de análisis, comienza la etapa de paralelización. Ésta consiste en extraer la información almacenada en el árbol sintáctico para aplicar las estrategias de paralelización adecuadas. La información para la paralelización se obtiene principalmente de las directivas que proporciona el programador. De esta forma se identifica la estructura de datos dispersa, su distribución y alineamiento. Por otro lado, se determinan los bucles a paralelizar, traduciendo el espacio de iteraciones y anotando los requerimientos de información no local para la posterior gestión de las comunicaciones.

En el último paso se genera el código SPMD en Fortran90. Este código contiene llamadas a nuestra librería en tiempo de ejecución DDLY que discutimos en el siguiente

epígrafe. Por tanto se llevarán a cabo las transformaciones necesarias para insertar el soporte en tiempo de ejecución y las modificaciones anotadas en la etapa de paralelización previa. El código generado puede ser convertido en el correspondiente ejecutable mediante un compilador estándar de Fortran90 para las plataformas paralelas que soportan los interfaces de pase de mensajes discutidos en la sección 2.2.1 (PVM, MPI y Cray SHMEM).

Ejemplo 4.14 Fichero de sintáxis de la directiva SPARSE (y 2)

•
•
•

```
RULE
hpf_directive: hpf_header sparse_directive '::' array_objects
             .
hpf_header: '!HPF$' datatype
          .
datatype: 'REAL'
        | 'INTEGER'
        .
sparse_directive: sparse_stmt
                | sparse_stmt opt_dyn
                | sparse_stmt opt_dim
                | sparse_stmt opt_dyn opt_dim
                | sparse_stmt opt_dim opt_dyn
                | opt_dyn sparse_stmt
                | opt_dim sparse_stmt
                | opt_dyn opt_dim sparse_stmt
                | opt_dim opt_dyn sparse_stmt
                | opt_dyn sparse_stmt opt_dim
                | opt_dim sparse_stmt opt_dyn
                .
array_objects: (Identifier [array_content]) || ','
             .
array_content: '(' int_expr [',' int_expr] ')'
             .
opt_dyn: ',' 'DYNAMIC'
       .
opt_dim: ',' 'DIMENSION' array_content
       .
int_expr: '(' int_expr ')'
        | int_expr '+' int_expr
        | int_expr '-' int_expr
        | int_expr '*' int_expr
        | int_expr '/' int_expr
        | int_expr '**' int_expr
        | '-' int_expr  PREC unary_defined_operator
        | Int_constant
        | Identifier
        .
sparse_stmt: ',' 'SPARSE' '(' sparse_content ')'
           .
sparse_content: 'LLRS'  '(' ll_spec ')'
              | 'LLCS' '(' ll_spec ')'
              | 'LLRCS' '(' ll2_spec ')'
              | 'CVS' '(' cvs_spec ')'
              .
cvs_spec: Identifier ',' Identifier ',' Identifier
        .
ll_spec: cvs_spec ',' link_spec
       .
ll2_spec: cvs_spec ',' cvs_spec ',' link_spec
        .
link_spec: 'SINGLY'
         | 'DOUBLY'
         .
```

4.6.2 Soporte en tiempo de ejecución

Dado que la distribución y el patrón de acceso a los datos depende de los datos que se vayan a procesar, parte del análisis de los códigos propuestos en HPF-2 extendido, se debe realizar en tiempo de ejecución. Para soportar el incremento de funcionalidad del compilador, hemos extendido nuestra librería en tiempo de ejecución DDLY (del inglés, `Data Distribution Layer`) [147]. Más precisamente, hemos añadido un conjunto de rutinas para manejar listas enlazadas, que serán llamadas por el código generado por el compilador de HPF-2 extendido.

Las rutinas DDLY realizan un conjunto de operaciones de alto nivel orientadas a la distribución y alineamiento de arrays en multiprocesadores de memoria distribuida. Concretamente, la librería proporciona un interfaz que permite el manejo de arrays como objetos abstractos, ocultando al usuario la estructura interna usada para almacenar y distribuir datos. También proporciona el manejo necesario en el pase de mensajes[2], optimización de las comunicaciones y gestión de la entrada/salida. El interfaz de usuario en DDLY está basado en descriptores de objetos. Una operación básica es la declaración de un descriptor de un array o matriz y su asignación a un objeto particular. Estos descriptores se gestionan de forma similar a como UNIX maneja los descriptores de ficheros.

Para declarar un descriptor disponemos de la rutina `ddly_new`, mientras que para asignar un descriptor a un objeto llamamos a la rutina `ddly_init`. Una vez enlazados el objeto y su descriptor, cualquier acceso y manipulación del primero se lleva a cabo a través de su correspondiente descriptor. Por ejemplo, la librería incluye ciertas rutinas para distribuir y alinear objetos. En la llamada a estas rutinas, sólo es necesario especificar el descriptor del objeto y algún que otro argumento más que indique el tipo de distribución/alineamiento, o el descriptor del objeto con el que se va a alinear.

Con este interfaz, cada directiva de distribución/alineamiento de HPF-2 es traducida en su correspondiente llamada a una de las funciones de la librería DDLY. El paradigma que subyace en la librería es SPMD, es decir, las funciones son llamadas por todos los procesadores con los mismos argumentos. De esta forma, las funciones se ejecutan de forma coordinada en el multiprocesador.

En los siguientes ejemplos presentamos el código generado por el compilador de HPF-2 extendido. En la salida hipotética en Fortran 90 de este compilador, todas las directivas introducidas en el código secuencial han sido traducidas en llamadas a las rutinas de la librería DDLY.

4.6.2.1 Sección de declaración

En el ejemplo 4.15 encontramos la salida compilada de la sección de declaración descrita en el ejemplo 4.6. La primera rutina DDLY que encontramos en el ejemplo es el procedimiento `ddly_new_topology` el cual inicializa la topología deseada sobre la que se ejecutará el código SPMD. En este caso se especifica un vector lineal de `Q` procesadores. La información relativa a la topología se almacena en el descriptor `td`. El procedimiento

[2]Para ello se apoya en los interfaces estándar MPI, PVM y SHMEM.

ddly_HB_read simplifica la lectura de la matriz de un fichero con formato Harwell-Boeing. La matriz dispersa se almacena en memoria en formato CCS (ver sección 1.2.2), mediante los tres vectores (Val,Col,Filpt) y el vector b para los términos independientes.

Ejemplo 4.15 SALIDA DEL COMPILADOR: SECCIÓN DE DECLARACIÓN

```
!HPF$ PROCESSORS, DIMENSION(Q):: linear
! --> td el el descriptor de la topología para una malla (1 x Q)
      CALL ddly_new_topology(td,1,Q)

!HPF$ REAL, DYNAMIC, SPARSE(LLCS(first, last, vsize, DOUBLY)):: A(n,n)
!HPF$ REAL, DYNAMIC, SPARSE(CVS(vcoli, vcolv, size)):: VCOL(n)
! --> Lee la matriz Harwell-Boeing  (Val,Col,Filpt)
      CALL ddly_HB_read(n,alpha,Val,Col,Filpt,b)
! --> Crea el descriptor de la matriz A, md_a, en formato CCS
      CALL ddly_new(md_a,CCS,DDLY_MF_REAL)
! --> Inicializa md_a
      CALL ddly_init(md_a,n,alpha,Val,Col,Filpt)
! --> Crea el descriptor del array VCOL, vd_vcol, en formato CVS
      CALL ddly_new(vd_vcol,CVS,DDLY_VF_REAL)
! --> Inicializa vd_vcol
      CALL ddly_init(vd_vcol,vcoli,vcolv,size)

! --> Crea e inicializa los descriptores para otros arrays distribuidos ...
      ...
!HPF$ DISTRIBUTE (*,CYCLIC) ONTO linear:: A
! --> Distribución BCS de A (especificada por md_a)
      CALL ddly_bcs(md_a, td)
! --> md_a es ahora el descriptor de la matriz distribuida
! --> Cambia la estructura de datos de CCS a LLCS
      CALL ddly_ccs_to_llcs(md_a,first,last,vsize,DOUBLY)

!HPF$ ALIGN iq(:) WITH A(*,:)
! --> iq es alienado con la segunda dimensión de A
      CALL ddly_alignv(vd_iq,md_a,SecondDim)

!HPF$ ALIGN vpiv(:) WITH A(*,:)
      CALL ddly_alignv(vd_vpiv,md_a,SecondDim)

!HPF$ ALIGN vmaxval(:) WITH A(*,:)
      CALL ddly_alignv(vd_vmaxval,md_a,SecondDim)

!HPF$ ALIGN vcol(:) WITH A(:,*)
! --> VCOL es alineado con la primera dimensión de A
      CALL ddly_aligncvs(vd_vcol,md_a,FirstDim)
```

En la llamada ddly_new la matriz A se asocia al descriptor de matriz md_a, el cual indica que la matriz dispersa A está almacenada en formato comprimido por columnas (CCS) y que el tipo de datos de la matriz es real (DDLY_MF_REAL). Posteriormente ddly_init inicializa la matriz A según el contenido de los tres vectores (Val,Col,Filpt) leídos anteriormente.

En cuanto al vector comprimido VCOL el procedimiento es parecido al que hemos seguido con la matriz A: ddly_new asocia el vector VCOL (en formato CVS y con entradas reales) con el descriptor de vectores vd_vcol; posteriormente ddly_init inicializa el descriptor asociándole la variable size y los vectores vcoli y vcolv (inicialmente vacíos). El resto de los vectores que se distribuyen o alinean (iq, vpiv y vmaxval) también se asocian a descriptores de vectores para simplificar la gestión de la distribución/alineamiento.

Una vez tenemos inicializados los descriptores de vectores y matrices pasamos a la distribución y alineamiento de estas estructuras. El procedimiento ddly_bcs distri-

buye la matriz A (identificada por su descriptor `md_a`) sobre el array de procesadores (identificado por el descriptor de topología `td`), aplicando el esquema de distribución BCS (ver sección 3.2). Mediante la llamada `ddly_ccs_to_llcs` todos los procesadores realizan localmente y en paralelo un cambio de la estructura de datos CCS a LLCS.

Por último se procede a alinear algunos vectores con la matriz distribuida. Como vemos en la sección de declaración los vectores pueden ser alineados con la primera o la segunda dimensión de la matriz `A`. De esta forma las directivas `ALIGN` de HPF se convierten en llamadas `ddly_alignv` (para alinear vectores densos) y `ddly_aligncvs` (para alinear vectores comprimidos). Dado que el vector comprimido `VCOL` se alinea con la primera dimensión de `A` el vector queda replicado.

Como vemos, ha sido necesario ampliar la librería DDLY con un conjunto de nuevas rutinas de inicialización/distribución/alineamiento para contemplar la amplia variedad de situaciones que podemos encontrar usando la directiva `SPARSE`.

4.6.2.2 Sección de ejecución

La salida asociada a la etapa de factorización dispersa discutida en los ejemplos 4.7, 4.8 y 4.9 se muestra en el código del ejemplo 4.16.

Como vemos, todos los bucles anotados con la directiva `INDEPENDENT` son ejecutados en paralelo debido a la distribución y la localización del espacio de iteraciones para cada procesador. Para ello, las funciones `ddly_LowBound` y `ddly_UpBound` se aplican a las cotas inferior y superior, respectivamente, de los bucles paralelos. Evidentemente, estas funciones, presentadas en el ejemplo 4.17, realizan la transformación dependiendo de la distribución que se esté utilizando. Estas funciones también se aplican a las cotas de las operaciones que utilizan notación de subarrays[3], ya que son totalmente equivalentes a un bucle DO paralelo.

Los comentarios "`Cuerpo del bucle`" y "`Cuerpo del ON HOME`" indican que la sección de código HPF-2 correspondiente de los ejemplos 4.7, 4.8 y 4.9 se copia directamente (sin modificaciones) en el código de salida.

Las operaciones de reducción realizadas en la etapa de búsqueda del pivot merecen una atención especial. De forma similar a como opera el paralelizador Polaris (que comentaremos en la siguiente sección), el compilador de HPF-2 divide los bucles de reducción (anotados con la cláusula `REDUCTION`) en dos pasos. En el caso de la reducción para obtener el máximo valor absoluto en la fila activa `k`, tenemos:

- Primero se realiza una reducción local donde la variable local `maxpiv` toma el valor máximo de la sección de array `vmaxval` que le corresponde.
- Seguidamente tiene lugar la reducción global de los valores `maxpiv` correspondientes a cada procesador. Para ello, la rutina `ddly_ReduceScalarMax` realiza las comunicaciones necesarias para que todos los procesadores almacenen en `maxpiv` el máximo global.

[3]Expresiones del tipo A(1:n), soportadas en F90 [1].

Esta reducción simple está contemplada en las versiones actuales de HPF, sin embargo, la que encontramos a continuación necesita un soporte adicional. Nos referimos a la reducción que selecciona un pivot numéricamente estable. Durante la reducción local se identifica el pivot que cumple la condición especificada en la sentencia `IF`. En una segunda etapa, la reducción global se realiza gracias al procedimiento `ddly_ReduceLocMaxAbs`.

Ejemplo 4.16 SALIDA DEL COMPILADOR: SECCIÓN DE EJECUCIÓN

```
!HPF$ INDEPENDENT
! --> Bucle paralelo
      DO j = ddly_LowBound(1), ddly_UpBound(n)
        vpiv(j)%p => first(j)%p
      END DO

! --> Bucle principal de la LU
main: DO k = 1, n

! --> Pivoteo
! --> Búsqueda del máximo valor absoluto en la fila activa k
!HPF$ INDEPENDENT
! --> Bucle paralelo
        DO j = ddly_LowBound(k), ddly_UpBound(n)
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
              ****** Cuerpo del bucle ****** (local a cada procesador)
!HPF$ END ON
        END DO
! --> Reducción paralela en dos etapas
! --> ... Primero, reducción local, ...
        maxpiv = MAXVAL(vmaxval(ddly_LowBound(k):ddly_UpBound(n)))
! --> ... Segundo, reducción global
        ddly_ReduceScalarMax(maxpiv)
! --> Elección de un pivot numéricamente estable (operación de reducción)
        actpiv=vmaxval(k)
        pivcol=k
        maxpiv = maxpiv*u
!HPF$ INDEPENDENT, REDUCTION(actpiv,pivcol)
! --> Bucle paralelo
        DO j = ddly_LowBound(k), ddly_UpBound(n)
          IF ( vmaxval(j) > maxpiv .AND. iq(pivcol) > iq(j) ) THEN
            actpiv = vmaxval(j)
            pivcol = j
          END IF
        END DO
! --> Reducción global
        pivcol = ddly_ReduceLocMaxAbs(actpiv,maxpiv,pivcol,iq)
        IF (pivcol /= k) THEN
! ----> Intercambio de columnas
           CALL swap(k,pivcol,first,last,vpiv,vsize,iq)
        END IF

! --> La columna del pivot es actualizada y empaquetada en VCOL
!HPF ON HOME (vpiv(k)), RESIDENT (A(*,k)) BEGIN
! --> Este bucle aparece debido a la directiva ON HOME
        DO dum = ddly_LowBound(k),ddly_UpBound(k)
          ****** Cuerpo del ON HOME ******
        END DO
!HPF END ON

! --> Radiación de la columna del pivot (dado que VCOL está replicado)
        CALL ddly_aligncvs(vd_vcol,md_a,FirstDim)

! --> Actualización de la submatriz reducida de A
!HPF$ INDEPENDENT, NEW (aux,i,amul,product)
! --> Bucle paralelo
        DO j = ddly_LowBound(k+1), ddly_UpBound(n)
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
          ****** Cuerpo del ON HOME ******
!HPF$ END ON
        END DO
      END DO main
```

Se han implementado varias rutinas DDLY para soportar una gran variedad de

reducciones. Sin embargo, tampoco es difícil construir estas rutinas en tiempo de compilación a partir de las sentencias del núcleo de la reducción. Esto se puede llevar a cabo siguiendo la estrategia de generación de código de reducción que incorpora Polaris, con la ventaja de que no es necesario detectar el bucle de reducción (el cual está ya marcado con la cláusula `REDUCTION`).

Ejemplo 4.17 FUNCIONES PARA DETERMINAR EL ESPACIO DE ITERACIONES

```
! Los bucles paralelos (DO j = globa, globb) se convierten en
! (DO j = ddly_LowBound(globa), ddly_UpBound(globb))
! my_pe es una variable global que contiene el identificador de proceso
! N$PES = número de PEs (variable global en el Cray T3E)

      INTEGER FUNCTION ddly_LowBound (i)
        INTEGER i
        ddly_LowBound = (i-1)/N$PES+1
        IF (my_pe < MOD(i-1,N$PES)) ddly_LowBound = ddly_LowBound+1
      END FUNCTION ddly_LowBound

      INTEGER FUNCTION ddly_UpBound (i)
        INTEGER i
        ddly_UpBound = i/N$PES
        IF (my_pe < MOD(i,N$PES)) ddly_UpBound = ddly_UpBound+1
      END FUNCTION ddly_UpBound
```

En la sección de código en la que la columna activa `k` se actualiza y copia en VCOL, el compilador incluye un bucle de índice `dum`. De esta forma, siguiendo la directiva ON HOME, el procesador propietario de `vpiv(k)` es el único que ejecuta esta sección de código. El compilador detecta que la estructura de datos (CVS) asociada a `VCOL` es modificada. Dado que este vector está replicado, cada componente modificada en un procesador se debe actualizar en el resto de los procesadores. No es difícil incluir una etapa de optimización que agrupa todas las escrituras al final del bucle de actualización, realizando la radiación de todo el vector mediante la llamada a `ddly_aligncvs`.

4.7 Evaluación de resultados

En esta sección nos centramos en la evaluación de las prestaciones del algoritmo paralelo LU discutido en la sección anterior. Los experimentos se han realizado en el multiprocesador Cray T3E, utilizando su compilador de Fortran90 y las librerías de comunicación SHMEM [25] incluidas en las llamadas DDLY.

Como se ha mencionado, las columnas de la matriz `A` se distribuyen cíclicamente sobre una array unidimensional de `Q` procesadores, y se almacenan mediante listas doblemente enlazadas. Queremos remarcar que el algoritmo paralelo, presentado en el ejemplo 4.15, es básicamente la versión secuencial con índices locales en lugar de globales y rutinas SHMEM para realizar las operaciones comunicación/sincronización. Estas dos diferencias entre el código paralelo y el secuencial se ocultan al programador gracias a llamadas a las rutinas DDLY comentadas en la sección anterior. Este código paralelo debe ser considerado como la salida del compilador HPF-2 extendido y no una optimización del código realizada manualmente.

En la figura 4.3 presentamos los tiempos de ejecución y la aceleración variando el número de procesadores. Las matrices de prueba han sido escogidas del conjunto Harwell-Boeing y se describen en la tabla 1.2. Esta elección es fruto de la accesibilidad y uso extensivo de estas matrices, lo cual permite comparar los resultados de otras implementaciones.

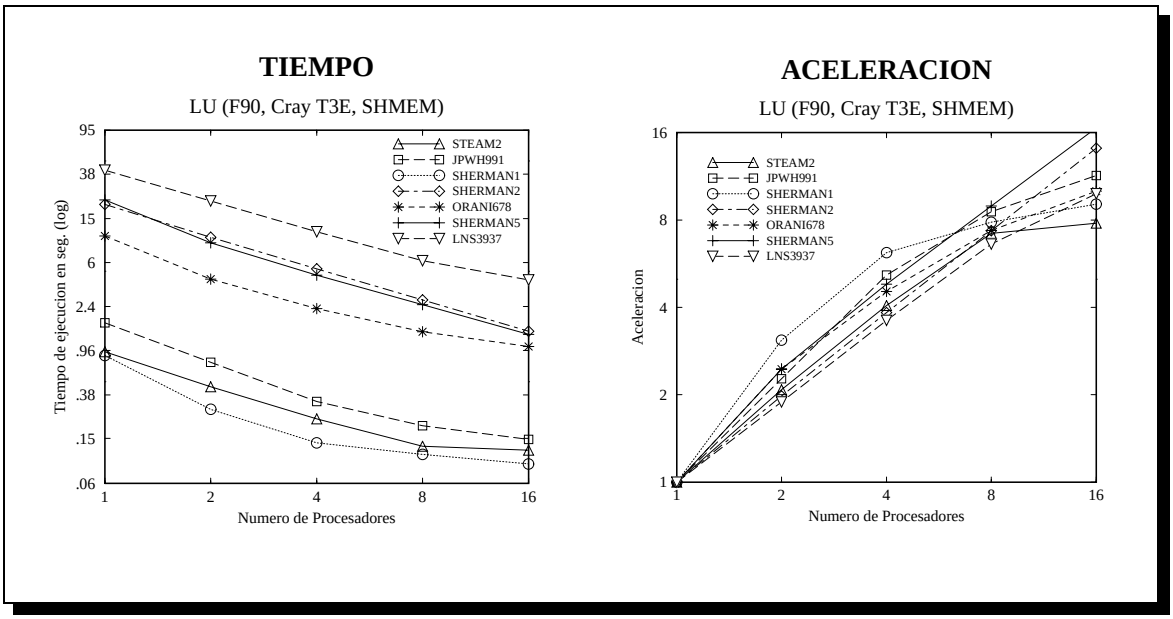


Figura 4.3: Tiempos de ejecución de la LU dispersa en paralelo y aceleración para diferentes tamaños del array de procesadores con comunicaciones SHMEM en un Cray T3E.

Obsérvese las altas eficiencias obtenidas cuando el tamaño de la matriz es suficientemente grande. De cualquier modo, matrices más grandes reportarían mejores resultados y nos permitirían explotar más paralelismo usando un mayor número de procesadores. También han sido evaluadas otro tipo de configuraciones de mallas en lugar de arrays de procesadores, pero los mejores tiempos paralelos se consiguen para topologías lineales, donde las matrices se distribuyen por columnas.

En nuestros experimentos, una etapa de análisis se ejecuta inicialmente para proporcionar los vectores de permutación adecuados. Esta etapa también proporciona la iteración `IterConm`, después de la cual se conmuta a código denso, lo cual tiene lugar en nuestros experimentos cuando la densidad de la submatriz reducida supera el 15%. El valor del parámetro `u` que afecta al umbral que han de superar los pivots tiene el valor 0.1 como se recomienda en el capítulo 3. Los tiempos de análisis y resolución triangular se muestran en la tabla 4.1, donde se aprecia que no suponen un tiempo significativo comparado con el tiempo de factorización.

También debemos justificar por qué el tiempo secuencial de nuestro algoritmo en F90 es muchas veces superior al de otras implementaciones codificadas en C, por ejemplo, la descrita en el capítulo 3. Después de realizar una serie de pruebas a los dos compiladores, en las que se evalúa el rendimiento de las operaciones con punteros, podemos afirmar que el compilador de F90 tiene un comportamiento pobre en relación al compilador de C al manejar este tipo de estructuras de datos (al menos en el Cray

	Tiempos		Llenado
Matriz	**Análisis**	**Resolución**	**nº de entradas**
STEAM2	.059	.0078	11869+(314x314)
JPWH991	.054	.0074	8866+(305x305)
SHERMAN1	.028	.0037	6306+(192x192)
SHERMAN2	.43	.039	24281+(795x795)
ORANI678	.35	.029	32235+(638x638)
SHERMAN5	.35	.034	32650+(698x698)
LNS3937	.95	.067	78972+(961x961)

Tabla 4.1: Tiempos de la etapa de análisis y resolución triangular (en seg.) y llenado final en los factores LU.

T3E). Más precisamente, nuestros códigos de prueba realizan inserciones en listas (con llamadas a `ALLOCATE` en F90 y `alloc` en C), las recorre y borran elementos (con llamadas a `DEALLOCATE` en F90 y `free` en C). Después de un gran número de medidas el C ha resultado, en media, cuatro veces más rápido en la inserción, dos veces más rápido recorriendo listas, y cinco veces más rápido al borrar entradas.

Por otro lado, es importante validar las prestaciones de nuestro código secuencial comparándolo con otro considerado como estándar, por ejemplo, la rutina en Fortran77 MA48 de las Harwell Subroutine Library [60]. En la tabla 4.2 se presenta una comparativa de las características significativas de ambas implementaciones, como son el tiempo de ejecución secuencial y el error de factorización. Podemos apreciar que la rutina MA48 es más rápida que nuestro algoritmo, pero también es justo considerar la ineficiencia del compilador de F90 al manejar listas y estructuras dinámicas. Sin embargo, los errores de factorización son prácticamente idénticos en los dos algoritmos. De igual modo, los tiempos de análisis, resolución triangular y, por otro lado, el llenado de la matriz LU, son completamente semejantes. La principal ventaja de nuestra aproximación es que resulta fácilmente paralelizable, aspecto que hemos demostrado altamente cuestionable, en el capítulo 3, para el caso de la MA48, dadas sus características inherentemente secuenciales a causa de su organización *left-looking*.

	Tiempos		Error
Matriz	**F90 – MA48**	**ratio**	**F90 – MA48**
STEAM2	.9361 – .61	(1.53)	.15E-11 – .13E-11
JPWH991	1.707 – .88	(1.94)	.44E-13 – .82E-13
SHERMAN1	.8646 – .19	(4.55)	.14E-12 – .16E-12
SHERMAN2	20.21 – 16.7	(1.21)	.14E-5 – .15E-5
ORANI678	10.46 – 6.48	(1.61)	.70E-13 – .74E-13
SHERMAN5	22.24 – 11.0	(2.02)	.75E-12 – .59E-12
LNS3937	41.26 – 25.8	(1.60)	.15E-2 – .13E-2

Tabla 4.2: Comparación entre el algoritmo en F90 y la MA48 (tiempos en seg.)

También es interesante comparar nuestra solución paralela con otras aproximaciones a la paralelización semi-automáticas basadas en técnicas en tiempo de ejecución. Por ejemplo, la librería en tiempo de ejecución CHAOS [115, 132] se basa en el paradigma Inspector-Ejecutor para manejar accesos irregulares a datos. Mediante estas

rutinas se pueden generar tablas distribuidas de traducción de índices globales a direcciones locales a cada procesador. Las rutinas CHAOS también generan la planificación de comunicaciones necesarias en los accesos a datos no locales. En [15] mostramos resultados en los que resulta manifiesta la superioridad de nuestra aproximación basada en distribuciones pseudo-regulares frente a la que puede ser implementada con CHAOS. En general, y para aplicaciones con llenado y pivoteo, como los algoritmos dispersos de factorización LU y QR, la implementación basada en el esquema de distribución LLCS-Scatter es casi un orden de magnitud más rápida que bajo el paradigma proporcionado por CHAOS. Resultados similares se presentan en [153] para problemas estáticos en los que no aparecen operaciones de llenado o pivoteo. Esto se justifica principalmente por la gran cantidad de comunicaciones y de posiciones de memoria, necesarias en CHAOS para acceder a grandes tablas de directorio distribuidas entre los procesadores. Por contra, el esquema LLCS-Scatter es adecuado para nuestros problemas dispersos ya que explota la localidad y minimiza las comunicaciones. Además no requiere de memoria adicional ni de comunicaciones para encontrar la dirección de datos no locales, ya que todos los procesadores conocen la posición de cualquier entrada global mediante una simple operación aritmética. Podemos concluir, por tanto, que nuestro esquema de distribución pseudo-regular consigue mejores prestaciones que el modelo general Inspector-Ejecutor proporcionado por CHAOS.

4.8 Otras aplicaciones

Aunque en este capítulo nos hemos centrado en el caso de la factorización dispersa LU en su versión *right-looking*, las directivas y aportaciones introducidas son válidas para un amplio espectro de aplicaciones dispersas en las que aparecen operaciones de llenado o pivoteo.

Por ejemplo, el código descrito en el capítulo 3 donde se explota el paralelismo asociado a la actualización de rango `m`, también podría ser expresado con las extensiones propuestas. En este caso, un esquema LLRCS-Scatter que permita acceso tanto por filas como por columnas y algunas etapas de reducción algo más complejas para determinar la compatibilidad de los pivots serían prácticamente suficientes. Sin embargo, también está claro que los tiempos de ejecución serán mayores, debido a que la herramienta de paralelización semi-automática no genera un código tan optimizado como el que obtenemos de paralelizar a mano.

En cuanto a las soluciones multifrontales, las implementaciones paralelas a nivel de tarea no se ajustan al paradigma implícito en HPF, donde los bucles constituyen la fuente esencial de paralelismo. Sin embargo, el paralelismo a nivel de bucle es también aplicable y puede ser representado en muchos casos mediante nuestras directivas. Por ejemplo, la factorización de Cholesky dispersa presentada en [65], en la que el problema del llenado y pivoteo está contemplado en una etapa de análisis anterior, un esquema BCS permite describir los dos bucles que consumen más de un 90% del tiempo de ejecución total. Lo dicho es aplicable hasta cierto punto en un algoritmo con filosofía supernodo [51], ya que puede ser necesaria una simplificación de la estructura de datos para hacer realizable la implementación en HPF-2. El paralelismo a nivel de tarea inherente en el árbol de supernodos o de eliminación puede explotarse gracias a la

directiva `TASK_REGION` aprobada en HPF-2.

Otros algoritmos con llenado y pivoteo como las factorizaciones QR, ya sea mediante un algoritmo de Gram-Schmidt modificado (MGS) o por reflexiones Householder (HR) han sido totalmente expresados en HPF-2 extendido y validados experimentalmente en [15].

Aparte de los algoritmos de factorización, encontramos otro tipo de problemas dispersos que aunque no presentan operaciones de pivoteo, sí contienen operaciones de llenado. Nos referimos a códigos como la suma, el producto o la transposición de matrices dispersas. Estos algoritmos son totalmente abordables de forma sencilla en el entorno de paralelismo de datos con las nuevas directivas presentadas, y los consideramos un caso particular de los anteriores.

Capítulo 5

Paralelización automática de códigos dispersos

Está claro que las herramientas de paralelización automática, son las que conducen a menores tiempos de desarrollo en la construcción de un algoritmo paralelo. Estas herramientas se materializan en compiladores fuente-fuente que a partir de un código secuencial, detectan las secciones paralelas para luego reestructurar y añadir las directivas, variables, sentencias, y comunicaciones necesarias que resultan, a la salida, en un código paralelo. Aunque la idea del compilador paralelo pueda parecer novedosa, en realidad no tiene otra misión que la que pueda tener cualquier compilador: ocultar al programador los detalles de la arquitectura para la que desarrolla sus códigos. Llevando esta idea al extremo, el lenguaje utilizado para expresar los programas debe ser también independiente de la arquitectura que presenta la plataforma, ya sea superescalar, vectorial, o, ¿por qué no?, paralela.

Bajo este paradigma, el programador sólo debe escribir y depurar la versión secuencial de su algoritmo, ya que delega en el compilador toda la responsabilidad de la paralelización. Sin embargo, tampoco es difícil imaginar que, actualmente, las prestaciones que presenta el algoritmo paralelo obtenido automáticamente son claramente superadas por una versión paralelizada a mano. Esta yuxtaposición entre facilidad de uso y prestaciones finales es comparable a la que existía entre la programación en ensamblador y con un lenguaje de alto nivel. Pero en este caso, es patente como la primera alternativa dejó de usarse[1], cuando los compiladores de lenguajes de alto nivel consiguieron, con el tiempo, rendimientos similares amén de una mucho mayor facilidad de programación. La fracción de la comunidad científica que trabaja en el ámbito del paralelismo sabe que, tarde o temprano, terminarán teniendo el mismo éxito los compiladores paralelos. La clave está en aumentar la potencia y el radio de acción de los compiladores paralelos para, también esta vez, romper el compromiso entre la comodidad y la eficiencia de la aproximación utilizada.

Es el objetivo de este capítulo el presentar las propuestas que aportamos en esta dirección. Embarcados, como estamos, en la resolución de problemas de índole irregu-

[1] Salvo para núcleos computacionales de poca longitud y de los que se requiere una alta optimización.

lar, más particularmente en algoritmos dispersos, cabe preguntarse cuanto paralelismo puede explotar un compilador paralelo automático de uno de nuestros códigos. En cierto modo, ésto ya ha supuesto ampliar el radio de acción de estos compiladores, que hasta ahora no habían encarado específicamente este tipo de problemas. Efectivamente, los compiladores automáticos suelen conseguir resultados competitivos en códigos regulares, debido a la mayor simplicidad en los bucles. Sin embargo, inicialmente, estos compiladores no son capaces de extraer paralelismo de los códigos dispersos con los que los alimentábamos. Ha sido necesario el desarrollo de una serie de técnicas adicionales y de algunas simplificaciones sencillas en los códigos para que las salidas paralelas fuesen competitivas.

Existe un determinado número de grupos desarrollando y mejorando sus herramientas de paralelización automática, a saber: el proyecto Polaris dirigido por David Padua [32]; SUIF por Mónica Lam [84]; Parafrase-2 por Polychronopoulos [113]; y PFA [137], herramienta comercial desarrollada por David Kuck.

Actualmente, estos compiladores no son capaces de extraer paralelismo cuando el código secuencial presenta estructuras de datos dinámicas o basadas en punteros. Por tanto, sólo para poder empezar a trabajar, ha sido necesario desarrollar un algoritmo disperso de factorización LU en Fortran77. Este algoritmo, presentado en la sección 5.1, es paralelizado a mano obteniéndose buenas eficiencias. Sin embargo, en un primer intento, Polaris, compilador en el que nos centramos en el resto del capítulo, no consigue explotar suficiente paralelismo por las razones que se comentan en la sección 5.2. La elección de Polaris como compilador paralelo viene avalada por los excelentes resultados que esta herramienta ha proporcionado con un gran número de códigos tanto del conjunto de *benchmarks*[2] SPEC [125] como Perfect Club [28], así como por su gran disponibilidad, lo que ha permitido a otros grupos investigar con el mencionado compilador. En la sección 5.3 se discuten un conjunto de técnicas orientadas a corregir las carencias y puntos débiles de Polaris al compilar códigos dispersos. Por último, en la sección 5.4, se presentan otros códigos que son paralelizados exitosamente mediante la aplicación de las mismas técnicas.

5.1 Factorización dispersa: Un caso de estudio

Actualmente, Polaris procesa código secuencial escrito en Fortran77 y, por otro lado, no está preparado para detectar paralelismo cuando los bucles recorren listas enlazadas. Por tanto, para chequear el compilador con un código disperso que presenta el problema del llenado, hemos escrito en Fortran77 el algoritmo de factorización dispersa LU.

5.1.1 Características generales

Este código, de tipo *right-looking*, emplea una estructura de datos CCS tanto para almacenar la matriz `A`, como los factores `L` y `U`. De esta forma evitamos la utilización de punteros, aunque seguimos utilizando una estructura comprimida. Al contrario que

[2]Término que identifica a los programas usados para evaluar las prestaciones de una arquitectura.

en la aproximación *left-looking*, donde para cada iteración `k` el llenado sólo tiene lugar en la columna `k`, en nuestro código *right-looking* el llenado aparece en toda la submatriz reducida. A pesar de ello, la organización *left-looking* no se presenta como una opción, ya que, como se comentó en el capítulo 3, es inherentemente secuencial.

El llenado en toda la submatriz reducida representada mediante la estructura de datos CCS puede ser manejado a costa de un inevitable movimiento de columnas. Para evitar un mayor tráfico de datos durante la fase de factorización, una etapa de análisis se encarga, en una primera fase, de determinar los vectores de permutación de filas y columnas. De esta forma se seleccionan los pivots que aseguran la estabilidad numérica y mantienen el coeficiente de dispersión. Durante esta fase de análisis también se decide la iteración a partir de la cual la submatriz restante puede ser tratada como densa y, por tanto, podemos conmutar a un código denso de factorización LU. Así pues, inicialmente se ejecuta un código de factorización dispersa, pero una vez alcanzada la iteración de conmutación, se permuta a un código denso basado en rutinas de nivel 2 (o 3 si la distribución es cíclica por bloques) BLAS. En este código denso se implementa pivoteo parcial de filas para asegurar la estabilidad numérica.

La distribución de datos en el código paralelo es BCS para la fase de factorización dispersa. Cuando esta fase termina, se realiza el cambio de estructura de datos en la submatriz reducida. Por tanto la conmutación a código denso consume un tiempo despreciable y la submatriz densa queda distribuida automáticamente según la distribución cíclica regular.

5.1.2 Estructura de datos

Para colaborar en la disminución del número de columnas a mover durante la factorización, en lugar de utilizar un algoritmo "in-place", ocuparemos una zona de memoria con la matriz de entrada y otra distinta para la matriz que contiene los factores `L` y `U`. La estructura de datos que organiza ambas zonas de memoria es CCS, es decir, un vector para almacenar coeficientes, otro para índices de fila y otro que contiene punteros al comienzo de cada columna. Además se utiliza un vector adicional con punteros, por lo que podemos llamar a la estructura completa CCS modificada. En la figura 5.1 presentamos gráficamente la estructura de datos mencionada.

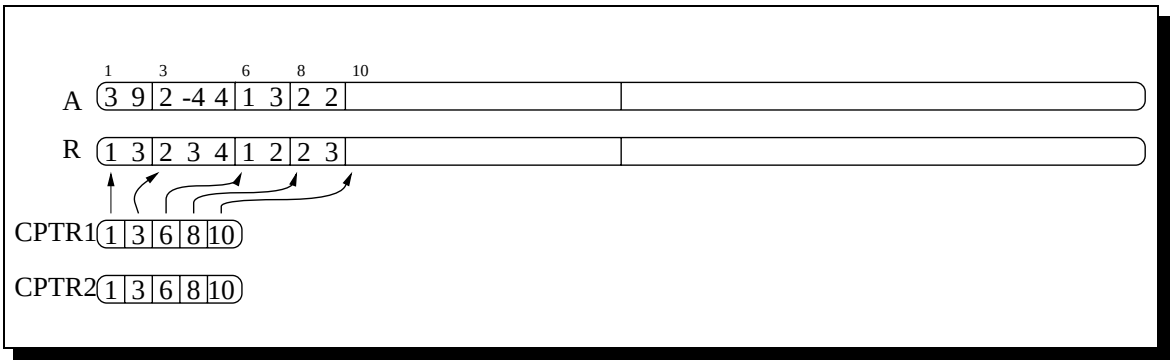


Figura 5.1: Estructura de datos CCS modificada para la matriz A.

Los arrays: `A` para los coeficientes; `F` para los índices; `CPTR1` para los punteros a comienzo de fila; y `CPTR2` usado para apuntar a la fila activa del pivot; contienen al comienzo del algoritmo la matriz inicial `A`. En la figura, estos tres vectores contienen los valores de la matriz dispersa $A^{(1)}$ del ejemplo 1.9, donde vemos que el vector `CPTR2` es una copia de `CPTR1` al comienzo del algoritmo. El espacio reservado en los arrays `A` y `F` se organiza en dos mitades, cuyo propósito se discute en la siguiente subsección. Inicialmente, $A^{(1)}$ debe caber en la mitad inferior.

La estructura de salida se muestra en 5.2, donde la matriz factorizada, $A^{(4)}$, del ejemplo 1.9, se organiza en el formato CCS modificado. El vector `FACT` contiene los coeficientes, `FR` los índices de fila y `FPTR1` los punteros al comienzo de cada columna. De nuevo, es necesario un vector adicional que apunte al pivot de cada columna. De esta forma marcamos donde acaba `U` y, por tanto empieza `L`, en cada columna. En la figura los pivots están marcados con fondo en gris. Esta estructura de datos es suficiente para diseñar el algoritmo disperso que se describe a continuación.

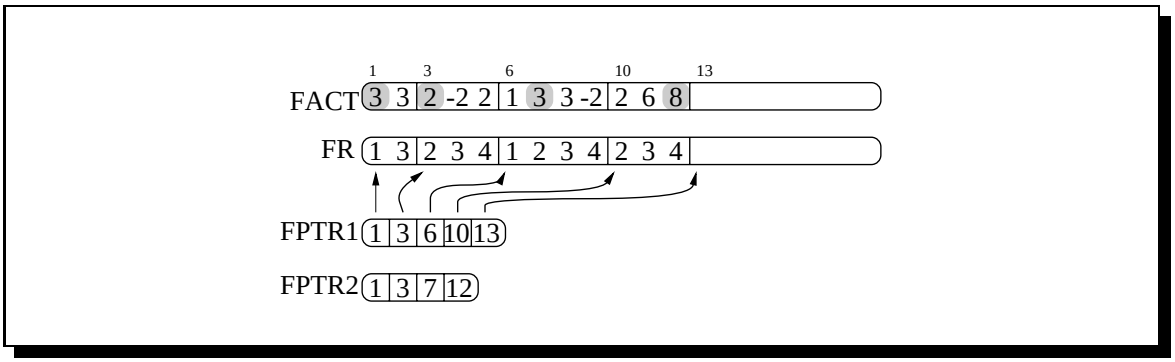


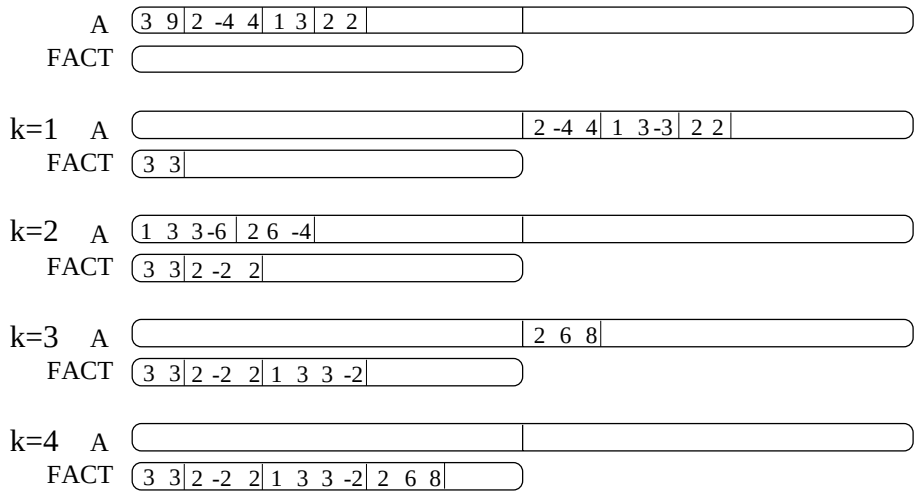
Figura 5.2: Estructura de datos CCS modificada para la matriz factorizada.

5.1.3 Organización del algoritmo

Como se ha repetido en otras ocasiones, la estructura de datos elegida es de una influencia determinante en la organización del posterior algoritmo disperso. En nuestro caso, el principal problema es el de encajar el llenado en la estructura comprimida. Es fácil entender que cuando aparece una nueva entrada en una columna debemos moverla a otra zona en la que quepa la columna completa. Así pues, durante la actualización de la submatriz reducida vamos actualizando y moviendo columnas para mantener éstas ordenadas tal y como impone la estructura CCS. A grandes rasgos, a cada iteración `k` del algoritmo, copiamos la columna del pivot (`k`) a la estructura de salida (`FACT, FR, FPTR`) una vez normalizada. El resto de las columnas de la matriz reducida se actualizan y mueven de una mitad de los arrays `A` y `R` a la otra, actualizándose convenientemente los punteros `CPTR1` y `CPTR2`. Cuando se alcanza la iteración de conmutación a código denso, la submatriz reducida restante se arregla en un array bidimensional, apropiado para la subsiguiente llamada a la rutina de factorización densa.

Antes de describir el código del algoritmo, nos detenemos un instante en el ejemplo 5.1 donde se muestra la evolución de los vectores de coeficientes `A` y `FACT` en las cuatro iteraciones necesarias para factorizar la matriz de la figura 5.1.

Ejemplo 5.1 FACTORIZACIÓN DE UNA MATRIZ 4×4



Al comienzo, la matriz de entrada se encuentra completamente en el array `A`. Después de la primera iteración, la primera columna normalizada por el pivot pasa a formar parte de la estructura de salida (`FACT`), mientras, las restantes columnas se actualizan en la segunda mitad del array `A`. Es ahora esta segunda mitad la que se lee durante la iteración `k=2` para obtener la segunda columna de `FACT` y actualizar las restantes en la primera mitad de `A`. Al final del algoritmo, `FACT` contiene los coeficientes de la matriz factorizada.

Ejemplo 5.2 FACTORIZACIÓN DISPERSA EN F77 (1ª PARTE)

```
      DO 20 I=1,N               !Inicialización
         CPTR2(I)=CPTR1(I)
 20   CONTINUE
      EYE=1
      DO 100 K=1,IterConm       ! Bucle externo
         FPTR1(K)=EYE           ! Comienza U
         DO 30 I=CPTR1(K),CPTR2(K)-1
            FACT(EYE)=A(I)
            FR(EYE)=R(I)
            EYE=EYE+1
 30      CONTINUE
         FPTR2(K)=EYE           ! Acaba U
C Copia el pivot
         AMUL=1/A(CPTR2(K))
         FACT(EYE)=AMUL
         FR(EYE)=R(CPTR2(K))
         EYE=EYE+1
         F1=EYE                 ! Comienzo del vector x en SAXPY (y=y-ax)
C Normaliza la columna del pivot
         DO 40 I=CPTR2(K)+1,CPTR1(K+1)-1
            FACT(EYE)=A(I)*AMUL
            FR(EYE)=R(I)
            EYE=EYE+1
 40      CONTINUE
         F2=EYE-1               ! Final del vector x
                                    •
                                    •
```

En el ejemplo 5.2, presentamos la primera parte del código. Al comienzo encon-

tramos el bucle DO 20 encargado de inicializar el vector `CPTR2` para que apunte a la fila del pivot. A continuación tenemos el bucle DO 100 que recorre las `IterConm` primeras iteraciones de factorización dispersa. Las líneas de código mostradas en este ejemplo son las encargadas de copiar la columna `K` de `A` en `FACT`. El bucle DO 30 copia la fracción de esa columna que pertenece a `U`, a continuación se copia el pivot (apuntado por `CPTR2`) y finalmente se normaliza la columna al tiempo que se mueve a la estructura de salida en el bucle DO 40. Las variables F1 y F2 apuntan al comienzo y al final, respectivamente, de la columna del pivot activa. Esta subcolumna jugará el papel de vector x en las operaciones SAXPY dispersas ($y = y - \alpha x$) que tienen lugar en la subsiguiente actualización. En el ejemplo 5.3, presentamos la segunda parte del bucle: la actualización de la submatriz reducida.

Ejemplo 5.3 FACTORIZACIÓN DISPERSA EN F77 ($2^{\underline{A}}$ PARTE)

•
•

```
C Actualiza la submatriz reducida
         SHIFT=MOD(K,2)*LFACT+1
         DO 50 J=K+1,N
            C1=SHIFT
C Copia la parte que no se modifica
            DO 60 I=CPTR1(J),CPTR2(J) ! Al menos se ejecuta una vez el bucle.
               A(SHIFT)=A(I)
               R(SHIFT)=R(I)
               SHIFT=SHIFT+1
 60         CONTINUE

            C2=SHIFT-1
C Copia el elemento de la fila K
            IF (R(C2).EQ.K) THEN
               AMUL=A(C2)
               C2=C2+1
C Scatter
               DO 70 I=CPTR2(J)+1,CPTR1(J+1)-1
                  W(R(I))=A(I)
 70            CONTINUE
C Actualiza
               DO 80 I=F1,F2
                  W(FR(I))=W(FR(I))-(AMUL*FACT(I))
 80            CONTINUE
C Gather
               DO 90 I=1,N
                  IF (W(I).EQ.0) GOTO 90
                  A(SHIFT)=W(I)
                  R(SHIFT)=I
                  SHIFT=SHIFT+1
                  W(I)=0
 90            CONTINUE
            ELSE
               DO 95 I=CPTR2(J)+1,CPTR1(J+1)-1
                  A(SHIFT)=A(I)
                  R(SHIFT)=R(I)
                  SHIFT=SHIFT+1
 95            CONTINUE
            END IF

            CPTR1(J)=C1
            CPTR2(J)=C2
 50      CONTINUE
         CPTR1(N+1)=SHIFT
         IF (SHIFT.GT.((MOD(K,2)+1)*LFACT)) STOP !Error por falta de espacio
 100  CONTINUE
```

La variable `SHIFT` es la encargada de apuntar al comienzo del vector `A` en las iteraciones impares o al comienzo de su segunda mitad en las iteraciones pares[3]. El bucle

[3]El array `A` es declarado de longitud `2`×`LFACT`.

DO 50, cuya variable índice es `J`, es el encargado de recorrer las columnas a la derecha de la columna del pivot para su actualización. En el cuerpo de este bucle encontramos primero un bucle, DO 60, que copia la parte de la columna por encima de la matriz activa, ya que no va a ser modificada. La variable `SHIFT` evoluciona dentro de este bucle apuntando a las posiciones del vector `A` donde se copian las columnas. Cuando una variable se utiliza de esta forma (ver bucle DO 60), con incrementos de la variable a cada iteración, se la llama **variable de inducción** [159].

A continuación se chequea si existe una entrada en la fila `K` para esa columna. En caso de no existir se ejecuta el bucle DO 95 que sencillamente mueve el resto de la columna. Por el contrario, si existe tal valor, se almacena en la variable `AMUL` que representa el escalar α de la operación SAXPY dispersa. Esta operación se organiza en tres etapas, discutidas en detalle en [57, Cap. 2.4]. Primero desempaquetamos la columna comprimida `J` sobre un array denso, `W`, inicialmente a cero, en el bucle DO 70 (operación *scatter*). A continuación se realiza la operación SAXPY dispersa, donde el vector `W` toma el papel de la variable y, en el bucle DO 80. Por último, el bucle DO 90 empaqueta los elementos no nulos de `W` al tiempo que vuelve a dejar a cero los coeficientes usados de `W` (operación *gather*). Si `SHIFT` supera la mitad del array en el que escribe, el programa se aborta por falta de espacio para encajar el llenado. Con ésto se cierran los bucles DO 50 y DO 100.

Las secciones de código de conmutación a código denso, factorización densa y resolución triangular son totalmente análogas a las descritas en capítulos anteriores. Por tanto pasamos a evaluar este nuevo algoritmo en sus versiones secuencial y paralela para comparar los resultados con los que inicialmente proporciona Polaris.

5.2 Un primer resultado con Polaris

El código descrito es fácil de paralelizar manualmente. Excepto el bucle exterior de índice `K`, el resto son paralelizables. El bucle DO 50 accede a las columnas que son independientes y los demás procesan estas columnas ya sea en la operación `divcol()` o en `actcol()`, inherentemente paralelas como se discute en el capítulo 1.

En resumen, el algoritmo paralelo necesita:

- Una distribución cíclica dispersa, que unida a la estructura de datos CCS conforman el esquema de distribución BCS. Tanto los índices de fila como los punteros a columnas son locales a la submatriz asignada a cada procesador.
- Dos etapas de comunicación: la primera radia por columnas de la malla la fila del pivot activa junto al propio pivot; y la segunda radia en las filas de la malla la columna del pivot activa.
- Modificación del espacio de iteraciones de los bucles para recorrer únicamente los coeficientes locales.

Con estas tres modificaciones realizadas sobre el algoritmo secuencial obtenemos el código paralelo que hemos testeado en el Cray T3D. El sólo hecho de distribuir

desacopla las iteraciones que presentan dependencias generadas por bucle. Por ejemplo, la variable de inducción `SHIFT` es privada a cada procesador y recorre estructuras locales, por lo que se cancela la dependencia generada por bucle que presenta en el algoritmo secuencial. De igual forma, el bucle DO 50 es secuencial mientras no se asegure que las escrituras en las variables dentro del bucle se realizan en posiciones distintas de memoria a cada iteración. Por tanto la distribución, y la consiguiente privatización de las variables escalares así como el vector de trabajo `W`, consigue que cada procesador modifique su sección de datos local sin interaccionar con los demás. En cuanto a las etapas de comunicación, pueden ser interpretadas como una fase en la que los procesadores *leen* la información no local que necesitan de otros procesadores.

En la tabla 5.1 se cuantifican las prestaciones del algoritmo paralelo comentado. El interfaz de pase de mensajes utilizado es el que proporcionan las librerías SHMEM de Cray y como rutinas de test algunas del conjunto Harwell-Boeing.

Matriz	Tiempo (seg.)				Aceleración			Eficiencia(%)		
	sec.	2×2	4×4	8×8	2×2	4×4	8×8	2×2	4×4	8×8
STEAM2	2.33	0.77	0.34	0.21	3.02	6.85	11.09	75.64	42.83	17.33
JPWH 991	3.73	1.30	0.55	0.34	2.86	6.78	10.97	71.73	42.38	17.14
SHERMAN1	2.18	0.88	0.42	0.26	2.47	5.19	8.38	61.93	32.44	13.10
SHERMAN2	36.22	8.84	2.58	1.02	3.99	14.38	35.50	99.75	87.74	55.48
LNS 3937	157.48	44.20	12.17	4.24	3.56	12.94	37.14	89.07	80.87	58.03

Tabla 5.1: Tiempos, aceleración y eficiencia del algoritmo paralelo en F77 para distintos tamaños de la malla en el CRAY T3D.

Sin embargo, intentar obtener resultados similares mediante la paralización automática fueron inicialmente infructuosos. Una prueba es el resultado de ejecutar la salida paralela de Polaris en el SGI Power Challenge con 16 procesadores R10000 a 196MHz. En la tabla 5.2 se aprecia por un lado el coste añadido que introduce el compilador en el algoritmo paralelo[4], y por otro que los tiempos de ejecución en más procesadores son incluso peores.

Matriz	Secuencial	1	2	4	8	16
LNS 3937	77.14	179.41	350.66	425.49	492.75	710.01

Tabla 5.2: Tiempo (seg.) de ejecución del algoritmo paralelo generado por Polaris para distinto número de procesadores del SGI Power Challenge.

La razón de esto es clara. Para este código irregular, Polaris sólo detecta el paralelismo trivial de los bucles 20, 30 y 40, eliminando automáticamente la variable de inducción `EYE` cuando aparece. En el ejemplo 5.4 se muestra el código que genera Polaris para este bucle DO 40. En este caso, el bucle puede ser expresado en lo que conocemos como su **forma cerrada**, es decir substituyendo la variable de inducción por una función del índice, `I`, del bucle. Como vemos en el ejemplo, en la última iteración la variable de inducción `EYE` toma el valor adecuado, ya que es usada fuera

[4]Comparando el tiempo de ejecución secuencial con el paralelo ejecutado en un nodo.

del bucle. No es difícil imaginar que el coste introducido en la versión paralela no compensa el trabajo de procesar la columna en paralelo, ya que para matrices de, por ejemplo, $n = 1000$ una columna puede tener de media menos de 100 elementos a procesar. Además este bucle encargado de la operación `divcol()` sólo se ejecuta una vez por cada iteración del bucle externo. Por tanto es realmente el bucle DO 50 el que merece mayor atención, por ser el que completa las $n - k$ operaciones `actcol()` que actualizan la submatriz reducida, a cada iteración `k`.

Ejemplo 5.4 CÓDIGO GENERADO POR POLARIS PARA DO 40

```
                      eye1 = eye
                C$DOACROSS IF(((-1)+cptr1(1+k)+(-1)*cptr2(k).GT.5.OR.29.GT.300).AND.29+
                C$& 5*((-1)+cptr1(1+k)+(-1)*cptr2(k)).GT.150),LOCAL(EYE3,I),SHARE(EYE,
                C$& CPTR1,K,CPTR2,A,AMUL,EYE1,FACT,R,FR)
                CSRD$ LOOPLABEL 'DPFAC_do40'
B40-------------      DO i = 1, (-1)+cptr1(1+k)+(-cptr2(k)), 1
|               CSRD$ INDUCTION eye3, 'DPFAC_do40', 0, eye1, (-1)+cptr1(1+k)+eye1+(-1)*cptr2(k)
|                      IF ((-1)+cptr1(1+k)+(-cptr2(k)).NE.i) THEN
|                         fact((-1)+eye1+i) = a(cptr2(k)+i)*amul
|                         fr((-1)+eye1+i) = r(cptr2(k)+i)
|                         eye3 = eye1+i
|                  40     CONTINUE
|                      ELSE
|                         fact((-1)+eye1+i) = a(cptr2(k)+i)*amul
|                         fr((-1)+eye1+i) = r(cptr2(k)+i)
|                         eye = eye1+i
|                1002    CONTINUE
|                      ENDIF
B40-------------      ENDDO
```

Pues bien, el bucle DO 50 es considerado como serie por Polaris porque las variables `A`, `R`, `CPTR1`, `W` y `SHIFT`, pueden presentar dependencias generadas por bucle. Discutimos este punto más adelante. Sin embargo dentro de este bucle Polaris detecta una operación de **reducción de histograma** [119]: el bucle DO 80 que escribe en el vector W(I). Para este bucle, la salida de polaris se muestra en ejemplo 5.5.

Este bucle, que realiza la operación `W(FR(I))=W(FR(I))-AMUL*FACT(I)` para el rango de `I` en (`F1:F2`) no representa en realidad una operación de reducción. Ya ha sido mencionado que el rango `F1:F2` marca la sección de la columna del pivot activa que actua de vector x en la operación SAXPY dispersa. Por tanto es el rango de una sección de un vector comprimido donde los índices no se repiten en el vector `FR(I)`. Se dice que, en ese rango, `FR(I)` es un vector de permutación. Por tanto las escrituras en `W` se realizan en posiciones distintas para cuaquier `I` en el rango `F1:F2`. Dado que las escrituras son en posiciones distintas, el bucle es paralelo y no presenta el coste asociado a la implementación de la reducción paralela que vemos en el ejemplo anterior, y que comentamos a continuación.

La reducción presentada en el ejemplo 5.5 tiene tres etapas, cada una de ellas marcada por la directiva del compilador de fortran de SGI `DOACROSS`, que antecede a un bucle paralelo. La primera de ellas se encarga de la puesta a cero en paralelo de tantos arrays privados, `W0`, como procesadores estén colaborando en la reducción[5]. El segundo bucle realiza en paralelo la operación de reducción local. Finalmente el último `DOACROSS` recorre en paralelo el vector `W` donde para cada coordenada se van

[5]Las variables `cpuvar` y `noproc` indican el número de procesadores trabajando, mientras que `mp_my_threadnum()` devuelve el identificador de cada proceso.

sumando secuencialmente las aportaciones correspondientes de la copia privada, `W0`, a cada procesador.

Ejemplo 5.5 CÓDIGO GENERADO POR POLARIS PARA DO 80

```
                  ptr = malloc(8*n*noproc)
              C$DOACROSS IF((cpuvar.GT.5.OR.n.GT.300).AND.cpuvar*(6+n).GT.1000),
              C$& LOCAL(TPINIT,I),SHARE(CPUVAR,N,W0)
              CSRD$ LOOPLABEL 'DPFAC_do80_RINIT'
C-----------          DO i = 1, cpuvar, 1
|             CSRD$ LOOPLABEL 'DPFAC_do80_RINIT_0'
| D---------           DO tpinit = 1, n, 1
| |                     w0(tpinit, i) = 0.0
| D---------           ENDDO
C-----------          ENDDO
              C$DOACROSS IF(8+(eye+(-1)*f1).GT.75),LOCAL(I),SHARE(F1,EYE,W0,FR,FACT,
              C$& AMUL)
              CSRD$ LOOPLABEL 'DPFAC_do80'
C80---------          DO i = f1, (-1)+eye, 1
|                      w0(fr(i), mp_my_threadnum()+1) = w0(fr(i), mp_my_threadnum()+1
|                 *)+(-fact(i))*amul
|                80     CONTINUE
C80---------          ENDDO
              C$DOACROSS IF((n.GT.5.OR.5*cpuvar.GT.300).AND.5*cpuvar+5*n.GT.150),
              C$& LOCAL(I,TPINIT),SHARE(N,CPUVAR,W,W0)
              CSRD$ LOOPLABEL 'DPFAC_do80_RSUM_0'
C-----------          DO tpinit = 1, n, 1
|             CSRD$ LOOPLABEL 'DPFAC_do80_RSUMI_0'
| D---------           DO i = 1, cpuvar, 1
| |                     w(tpinit) = w(tpinit)+w0(tpinit, i)
| D---------           ENDDO
C-----------          ENDDO
                      CALL free(ptr)
```

Aunque Polaris hubiese detectado de alguna forma que `FR(I)` es un vector de permutación en el rango `F1:F2` la paralelización del bucle DO 80 como tal, en lugar de su reducción, tampoco hubiera resuelto demasiado la falta de eficiencia. De hecho, en una modificacion manual de la salida de Polaris se paralelizan también los bucles 60, 70, 90 y 95 consiguiendo una sustancial mejora en el sentido de que el código paralelo al menos no es más lento que el secuencial, pero el resultado global sigue siendo muy bajo, ya que en general la eficiencia no superó el 2%. En efecto, los bucles 60, 70, 90 y 95 que Polaris, conservativamente, marca como secuenciales por existir riesgo de dependencia generada por bucle[6], son, al contrario, paralelos. Por ejemplo, es fácil demostrar que los bucles 60 y 95 no presentan dependencias generadas por bucle en las variables `A` y `R` si estos arrays han sido declarados con suficiente espacio. En particular, si una mitad del array es capaz de almacenar las columnas de la submatriz activa actualizada en cada iteración `k`, no existe solapamiento entre la zona del array `A (R)` que se lee y la que se escribe. Por otro lado, en caso contrario, incluso el programa secuencial se aborta por falta de espacio para encajar el llenado.

Al mismo tiempo, la dependencia generada por bucle en DO 70 debido a la sentencia `W(R(I))=A(I)`, se anula al comprobar que el rango de asignaciones corresponde al de una columna reducida, por lo que `R(I)` en ese rango es un vector de permutación.

El bucle 90 también se puede paralelizar mediante una privatización. La idea es que cada procesador ejecute ese bucle en paralelo leyendo cada uno de un bloque distinto de `W`. Las escrituras se deben realizar en arrays privados a cada procesador,

[6]Las variables que pueden provocar estas dependencias son `A` y `R` en los bucles 60, 90 y 95, `W` en el bucle 70, y `SHIFT` también en el bucle 90.

por ejemplo, `pA` y `pR`. Después, esos arrays privados se copian en una etapa paralela en la que cada procesador vuelca sus coeficientes en las variables de memoria compartida `A` y `R`, conociendo cada uno donde debe empezar a escribir. A esta última operación se la llama etapa de *copy-out*.

Sin embargo, como ya hemos dicho, aún con la ejecución en paralelo de todos estos bucles (30, 40, 60, 70, 80, 90 y 95) el resultado no es alentador. La razón es que estos bucles recorren columnas o secciones de columnas. El coste asociado a la ejecución paralela del procesado de una columna, no se compensa con la pequeña carga computacional presentada por las columnas dispersas.

Queda otra alternativa: es cierto que el trabajo dentro de una columna es independiente, pero también lo son las columnas entre sí. Retomemos la discusión sobre el paralelismo que presenta el bucle DO 50. Como decíamos, Polaris advierte que las variables `A`, `R`, `CPTR1`, `W` y `SHIFT`, pueden presentar dependencias, por tanto, conservativamente marca el bucle como secuencial. La situación es complicada ya que los arrays `A` y `R` se acceden mediante la varible de inducción `SHIFT`, que para empeorar las cosas, se incrementa condicionalmente (dependiendo de la cantidad de llenado en cada columna) en el bucle 90.

En efecto, las dependencias existen y el bucle no se puede ejecutar en paralelo en memoria compartida sin más. Pero tampoco deja de ser verdad que el procesado de una columna (`actcol()`) es independiente de las demás. Para desacoplar el trabajo entre columnas es necesario privatizar las columnas que van a ser procesadas en paralelo por cada procesador. Adicionalmente, el vector de trabajo `W` también se privatiza ya que se harán operaciones distintas sobre él en paralelo. Evidentemente, cada procesador tendrá una copia local de `SHIFT`. El problema es detectar automáticamente que esta paralelización es factible y realizar las privatizaciones correspondientes. Abordamos este aspecto en la siguiente sección.

5.3 Técnicas orientadas a códigos dispersos

Determinar de forma automática que las iteraciones del bucle DO 50 son independientes y paralelizarlo conlleva precisamente esas dos etapas. Por un lado hay que completar los test de dependencias para que sean capaces de inferir que el bucle es paralelo. En segundo lugar las dependencias generadas por bucle, que también van a ser detectadas, se pueden romper gracias a la privatización adecuada de las variables (escalares o arrays) que se definen y usan dentro de una única iteración [150]. Dedicamos una subsección a cada aspecto.

5.3.1 Detección de paralelismo: test de dependencias

Es evidente que dentro del bucle DO 50, las variables `A` y `R` se leen indexadas por el índice `I` que recorre una columna desde `CPTR1(I)` a `CPTR1(I+1)-1`, como bien puede inferir el analizador de acceso a regiones de Polaris [33]. Por otro lado las escrituras en `A` y `R` dentro del bucle siempre se indexan mediante la variable `SHIFT`. Con esto, y como discutimos en [16], dos hechos son suficientes para demostrar que el bucle que

nos concierne se puede ejecutar en paralelo:

1. El rango de SHIFT no puede solapar el mismo rango de SHIFT para iteraciones del bucle J ejecutadas en otros procesadores. Estamos diciendo que no se puede escribir donde otro procesador va a escribir en paralelo. Más formalmente: decimos que no van a existir dependencias de salida.

2. El rango de SHIFT no puede solapar el rango CPTR1(j), CPTR1(j+1)-1 recorrido en otros procesadores. Igual que antes, es intuitivo que no puedes leer de la zona en la que otro procesador va a escribir en paralelo. Formalmente, imponemos que no existan dependencias de flujo ni anti-dependencias.

Para probar el primer punto es claramente suficiente con realizar el siguiente análisis simbólico: siempre que se escribe en A y R la variable SHIFT se incrementa, por lo que dentro del bucle nunca se va a escribir dos veces en la misma posición de estos vectores. Es decir, no existen dependencias de salida.

Es más, el compilador podría hacer, aunque no es necesario para este primer punto, un análisis más completo demostrando que la variable SHIFT es estrictamente creciente. Esto puede ser inferido fácilmente por el compilador, haciendo un estudio simbólico de las sentencias que escriben en SHIFT dentro del bucle. Es fácil ver que la variable nunca se decrementa. Adicionalmente, la condición inicial impuesta en el bucle DO 20 garantiza que el bucle 60 ejecuta al menos una iteración, con lo que SHIFT se incrementa al menos en una unidad. Por tanto, uno de los invariantes de este bucle es que la variable de inducción SHIFT es estrictamente creciente. Por tanto, no existe dependencias de salida.

El segundo test debe demostrar que no existen dependencias de flujo ni anti-dependencias, es decir que el rango de I (CPTR1(j):CPTR1(j+1)-1) no va a solapar el rango de SHIFT en otros procesadores. En este caso necesitamos de un test en tiempo de ejecución ya que el contenido de SHIFT y CPTR1 no es conocido en tiempo de compilación. Este test debe probar que:

- el índice más pequeño que accede al vector A para lectura, min_i, es mayor que el índice más grande que accede al vector A para escritura, max_shift, y,

- el índice más grande que accede al vector A para lectura, max_i, es menor que el índice más pequeño que accede al vector A para escritura, min_shift.

Si no tenemos ninguna información adicional sobre el vector CPTR1, la búsqueda del valor mínimo y máximo de I debe hacerse recorriendo las n-k componentes del vector. Sin embargo, el compilador puede eliminar complejidad al comprobar que CPTR1 es no decreciente[7] antes de entrar en el bucle 50 y para cada iteración del bucle K. Efectivamente, un análsis simbólico determina que a este array se le asigna, en cada iteración del bucle, la variable C1. Ésta, a su vez toma su valor de SHIFT, que, como ya hemos demostrado, es estrictamente creciente. Por tanto en este caso, CPTR1 es

[7]Es decir CPTR1(J)$\leq$CPTR1(J+1) para todo el rango de J (entre K+1 y N).

estrictamente creciente después de la primera iteración del bucle K[8]. Si esto es así, para una iteración completa del bucle DO 50 con J barriendo entre K+1 y N, tenemos min_i=CPTR1(K+1) y max_i=CPTR1(N+1)-1.

En cuanto al valor mínimo y máximo de SHIFT, la situación es un poco más compleja, ya que SHIFT se incrementa condicionalmente en el bucle 90. Por un lado, tenemos trivialmente que min_shift=SHIFT, pero debemos estimar el valor max_shift suponiendo el caso peor. Considerando el máximo número de escrituras en A y R obtenemos max_shift=SHIFT+(N-K)*(F2-F1+1)+max_i-min_i.

Con todo, en el ejemplo 5.6 se muestra el test de dependencias en tiempo de ejecución. Este test comprueba que no hay solapamiento en el bucle 50, entre posiciones leídas de A y R y posiciones escritas. Vemos como la variable booleana parallel se inicializa a .false. si hay solapamiento. En ese caso el bucle DO 50, conservativamente, se ejecuta en secuencial gracias a la clausual IF(parallel) de la directiva DOACROSS que precede al bucle.

Ejemplo 5.6 TEST EN TIEMPO DE EJECUCIÓN EN DO 50

```
        if(maxi.ge.minsh .and. maxsh.ge.mini) then
           parallel = .true.
        else
           parallel = .false.
        end if

C$DOACROSS if(parallel), local(...), share(...)
        DO 50 J=K+1,N
          ....
   50   CONTINUE
```

5.3.2 Paralelización: privatización automática

Una vez sabemos que no hay dependencias, es decir, el bucle es paralelo, hay que abordar la generación del código paralelo para ese bucle. Es decir, generar el código que permita que los procesadores procesen conjuntos disjuntos de columnas en paralelo. Sin embargo, en el código secuencial existen variables escalares (C1, C2, AMUL, SHIFT) y vectoriales (el vector de trabajo W) que se utilizan para procesar una columna. Cada procesador debe privatizar estas variables para que el procesado de su columna no interfiera con el procesado de las demás que se está realizando en paralelo.

La privatización es una técnica que elimina las anti-dependencias entre iteraciones (generadas por bucle) [151]. Una variable se privatiza cuando cada procesador realiza una copia de la variable y sólo accede a su copia local. En cada iteración de un bucle, si toda lectura de una variable está dominada por alguna definición de esta variable en la misma iteración, podemos afirmar que la variable es privatizable. También son privatizables las variables que tienen al final de una iteración el mismo valor que tenían al comienzo.

[8]Si queremos probar la monotonía del vector CPTR1 para K=1 podemos probar que CPTR(I) $\leq$ CPTR(I+1) para I=1,n.

Evidentemente son privatizables las variables que no conducen a dependencias generadas por bucle, es decir aquellas que se definen y usan en una iteración sin provocar efectos laterales en ninguna otra iteración, por ejemplo `C1, C2, AMUL`. Queremos recordar que esta privatización, que bajo este paradigma es automática, se lleva a cabo en los compiladores semi-automáticos de paralelismo de datos utilizando la cláusula NEW junto a la directiva `INDEPENDENT`.

La variable `SHIFT` sí tiene efectos laterales en otras iteraciones, pero es una variable de inducción que tiene un tratamiento especial a comentar en un parrafo posterior. Por último, se debe demostrar que el array `W` no induce a dependencias generadas por bucle para poder privatizar todas las variables de trabajo usadas en el bucle. Para determinar si un array es privatizable, analizamos las secciones del array que son definidas y usadas en cada iteración. La sección usada de un array está dominada por la sección definida de un array si la sección definida cubre la sección usada [151]. En nuestro caso, el patrón de acceso a la variable W(I) se muestra de forma simplificada en el ejemplo 5.7.

Ejemplo 5.7 PATRÓN DE ACCESO A W EN DO 50

```
do i=1,n
     W(i)=0
end do

do j=...
    ...
    do i=...
        W(R(i))=...
    end do
    do i=...
        W(FR(i))=...
    end do

    do i=1,n
        if(W(i).ne.0) then
            ...=W(i)
            ...
            W(i)=0
        end if
    end do
end do
```

Esta situación es más complicada que las que se presentan en [151], debido a la escritura en `W` mediante indirecciones a través de los arrays de índices `R` y `FR`. Sin embargo la solución es fácilmente implementable en Polaris. Un análisis simbólico reporta que el array `W`, inicializado a cero al comienzo del programa, vuelve a estar en ese estado al acabar cada iteración `j`, si `R(i)` y `FR(i)` toman valores entre uno y `n` en el rango en el que están definidos estos vectores. Esa comprobación es factible mediante un rápido test en tiempo de ejecución que determine que esa condición es cierta, una vez, al comienzo del programa, para `R(i)`, unido al siguiente análisis simbólico de propagación de rangos [33]: `FR(EYE)` se copia de `R(i)` fuera del bucle 50, y `R(SHIFT)` se copia de `R(i)` dentro de este bucle, excepto en el bucle 90 donde tenemos `R(SHIFT)=I` con `I` en el rango `1:N`. Por tanto Polaris puede concluir que el rango de valores almacenados en `R` y `FR` es (`1:N`) y por tanto el array es privatizable. Existen otros patrones, también en el ámbito de la computación dispersa, que se pueden tratar de forma similar y que abordamos en la última sección de este capítulo.

En cuanto a los arrays `A` y `R`, su acceso en escritura está determinado por la variable

de inducción SHIFT. En estas situaciones la paralelización presenta dos alternativas:

- Cuando el bucle que maneja la variable de inducción se puede expresar en su forma cerrada (del inglés *closed form*), es decir expresar la variable de inducción como una función del índice del bucle, no hay problema. Bajo este supuesto es fácil determinar en qué posición del array debe escribir cada procesador.

- Sin embargo, existen situaciones en las que esto no es posible. Por ejemplo, en nuestro caso, la variable SHIFT se incrementa condicionalmente en bucle 90 (–etapa *gahter*– por lo que el incremento de SHIFT depende del llenado que sufra cada columna en particular). Por tanto un procesador no sabe donde debe escribir hasta que no se haya completado la escritura en el procesador que gestiona la columna previa.

Si el bucle es paralelizable, la solución en esta segunda situación está de nuevo en el uso de técnicas de privatización. Es decir, se debe reservar un espacio local y temporal para copiar las columnas que corresponda actualizar a cada procesador. Más precisamente, en nuestro caso, la escritura se realizará sobre vectores privados, pA y pR, indexados mediante la también privatizada pSHIFT[9]. Tras el proceso de actualización local, debe existir una etapa en la que se calcule a partir de qué posición del array global, A y R, debe escribir cada procesador. En este punto, cada procesador vuelca en paralelo, sus vectores locales sobre los globales, en una etapa llamada de **copy out**.

Otro prisma desde el que mirar el mismo problema se presenta en [119] en el que la privatización de A y R es consecuencia de la detección automática de la operación *gather* del bucle 90. El autor de este trabajo, presenta la técnica de privatización como método de paralelización de operaciones de fusión de bucle (del inglés *coalescing loop operation*) siempre que éstas sean asociativas, y la operación *gather* lo es.

5.3.3 Evaluación del código generado

Algunas de las técnicas comentadas no están totalmente disponibles aún en Polaris[10]. Antes de incluirlas en el compilador es necesario validar que permitirán conseguir mejoras significativas en la eficiencia de los códigos paralelos generados. Por tanto, hemos simulado a mano la hipotética salida del futuro compilador con este y con otros códigos, como presentamos en la siguiente sección. Como muestra, presentamos la sección de código generada para el bucle DO 50 en el ejemplo 5.8.

Las variables escalares se privatizan mediante la cláusula local() en el DOACROSS. La excepción es pSHIFT que se privatiza dimensionandola en función del número de procesos –mp_numthreads()–. Esto es así ya que pSHIFT debe mantenter su valor a la salida del bucle paralelo. Para evitar la compartición falsa (del inglés *false sharing*) en la cache, el array pSHIFT se accede en posiciones que corresponden a líneas distintas de cache. Los vectores pA, pW, pR, pCPTR1 y pCPTR2 son privatizados siguiendo la

[9]Utilizaremos el prefijo "p" minúscula para referirnos a la copia local asociada a los distintos vectores y escalares.

[10]Aunque ya ha comenzado su implementación.

misma técnica: se les añade una dimensión adicional que es indexada en función del número de proceso –mp_my_threadnum()–. Como se puede apreciar en el ejemplo, el código no cambia, salvo en que se escribe en las variables privadas en lugar de en las variables globales.

En [16] presentamos una aceleración de 3.84, conseguida con esta versión del algoritmo, para la matriz LNS3937 (ver tabla 1.2), en los 8 procesadores R4400 a 150MHz del multiprocesador de memoria compartida SGI Challenge. Esa aceleración se traduce en una eficiencia del 48%, la cual mejora muchos de los resultados obtenidos en memoria compartida, como los discutidos en 3.1, con la diferencia de que estos últimos están desarrollados y optimizados manualmente.

Ejemplo 5.8 SIMULACIÓN DE LA SALIDA DE POLARIS PARA DO 50

```
        do i = 1,mp_numthreads()
           psi = lshift(i-1,6)+33
           pSHIFT(psi) = 1
        enddo
C$DOACROSS if(parallel),local(pc1,pc2,psi,amul,j,tid,i,ii),share(pA,pW,pR,pCPTR1,pCPTR2)
        DO 50 J=K+1,N
           tid = mp_my_threadnum() + 1
           psi = pSHIFT(lshift(tid-1,6)+33)
           pC1=psi
           DO 60 I=CPTR1(J),CPTR2(J) ! Al menos se ejecuta una vez el bucle.
              pA(psi,tid)=A(I)
              pR(psi,tid)=R(I)
              psi=psi+1
 60        CONTINUE
           pC2=psi-1
C Copia el elemento de la fila K
           IF (pR(pC2,tid).EQ.K) THEN
              AMUL=pA(pC2,tid)
              pC2=pC2+1
C Scatter
              DO 70 I=CPTR2(J)+1,CPTR1(J+1)-1
                 pW(R(I),tid)=A(I)
 70           CONTINUE
C Actualiza
              DO 80 I=F1,F2
                 pW(FR(I),tid)=pW(FR(I),tid)-(AMUL*FACT(I))
 80           CONTINUE
C Gather
              DO 90 I=1,N
                 IF (pW(I,tid).EQ.0) GOTO 90
                 pA(psi,tid)=pW(I,tid)
                 pR(psi,tid)=I
                 psi=psi+1
                 pW(I,tid)=0
 90           CONTINUE
           ELSE
              DO 95 I=CPTR2(J)+1,CPTR1(J+1)-1
                 pA(psi,tid)=A(I)
                 pR(psi,tid)=R(I)
                 psi=psi+1
 95           CONTINUE
           END IF
           pCPTR1(J,tid)=pC1
           pCPTR2(J,tid)=pC2
           pSHIFT(lshift(tid-1,6)+33) = psi
 50     CONTINUE
```

Un estudio más detallado que el presentado en [16] se ha realizado sobre una plataforma más moderna: el SGI Power Challenge con 16 procesadores R10000. La tabla 5.3 resume los resultados para la misma matriz LNS3937.

En esta tabla, la primera columna contiene el tiempo en segundos del algoritmo secuencial inicial sin ninguna modificación. El resto de las columnas dan una medida de las prestaciones del algoritmo paralelo para distinto número de procesadores. Como

	Secuencial	1	2	4	8	16
Tiempo (seg.)	77	138	67	31	18	15
Aceleración	–	0.56	1.15	2.48	4.28	5.13
Eficiencia	–	56%	57%	62%	53%	32%

Tabla 5.3: Prestaciones del algoritmo en el SGI Power Challenge.

vemos, la eficiencia se mantiene por encima del 50% para 2, 4, y 8 procesadores, aunque baja significativamente en 16 PE's. La principal razón de esta pérdida de escalabilidad no es tanto el coste introducido por el algoritmo paralelo sino el desbalanceo en la planificación del bucle. Esto es así ya que las iteraciones del bucle 50 se planifican según un esquema por bloques para simplificar la operación de copy-out. Sin embargo esto redunda en una mayor carga para el procesador que se queda con el último bloque, debido al llenado de la matriz.

También es apreciable cómo el algoritmo secuencial es un 56% más rápido que el paralelo ejecutado en un único procesador. Esto nos da una idea del coste que está introduciendo la paralelización del bucle 50, principalmente la etapa copy-out que implica un gran volumen de movimiento de datos.

Por tanto, aunque suficientes, a pesar de todo estos resultados son mejorables, pero por el momento no automáticamente. En efecto, la etapa de *copy out* es costosa porque implica un gran movimiento de datos. Esta etapa ha sido eliminada manualmente para estudiar la mejora. Las modificiaciones implicaban la utilización de otro vector de punteros para indicar el final de cada columna, permitiendo así que las columnas no sean consecutivas. De esta forma se puede simular un algoritmo en el que el bucle DO 50 se ejecuta siempre sobre arrays privados, lo cual, en cierto modo, no es más que una emulación de la versión del algoritmo para memoria distribuida (ver sección 5.2).

5.4 Aplicación a otros problemas dispersos

A pesar de los resultados obtenidos, la dura tarea de automatizar estas técnicas en Polaris no se justifica si no van a tener mayor aplicación que la de paralelizar automáticamente nuestro código de factorización dispersa. Sin embargo, el caso de estudio elegido, como ya se ha dicho en otras ocasiones, presenta un conjunto de características que lo convierten en un buen caso para extraer conclusiones aplicables a problemas similares. Las características a las que nos referimos son básicamente dos: el uso de una estructura de datos comprimida (CRS o CCS) y la existencia de llenado durante la computación. En general son dos aspectos muy frecuentes en computación de problemas dispersos.

En [16] se discuten los resultados de la aplicacion de estas técnicas y otras a un conjunto de códigos irregulares. En las siguientes subsecciones presentamos un resumen de ese trabajo, significativamente extendido con otros códigos dispersos o irregulares. En general, hemos encotrando que las técnicas de detección de paralelismo y privatización discutidas anteriormente son aplicables con éxito a una gran variedad de códigos reales.

5.4.1 Operaciones vectoriales y matriciales

Uno de los códigos de test (*benchmarks*) discutidos en [16] contempla un conjunto de rutinas dispersas, entre las que se incluyen el producto matriz dispersa por vector y el producto y suma de dos matrices. En ese programa las rutinas incluidas siempre escriben el resultado de la operación sobre un vector o una matriz densa. En estos casos, la computación no presenta dependencias ya que las estructuras dispersas únicamente se leen, y la escritura en matrices densas es regular y fácilmente paralelizable. Por ejemplo, el producto $y = Ax$, donde A es una matriz dispersa en formato CRS o CCS, y x e y son vectores densos, presenta el código mostrado en el ejemplo 5.9.

Ejemplo 5.9 CÓDIGOS PARA EL PRODUCTO MATRIZ DISPERSA VECTOR

CRS

```
      DO 100 I=1,n
         Y(I)=0.0
         DO 100 J=Filpt(I),Filpt(I+1)-1
            Y(I)=Y(I)+Val(J)*X(Col(J))
100   CONTINUE
```

CCS

```
      Y(1:n)=0.0
      DO 200 I=1,n
         DO 200 J=Colpt(I),Colpt(I+1)-1
            Y(Fil(J))=Y(Fil(J))+Val(J)*X(I)
200   CONTINUE
```

En este ejemplo, el bucle DO 100 marca el núcleo computacional del producto, cuando la matriz `A` está representada mediante una estructura CRS. Como vemos, el bucle más externo es paralelo ya que las escrituras se realizan en componentes distintas de `Y` en cada iteración. Si representamos `A` mediante una etructura CCS, el algoritmo cambia, como se muestra en la sección de código delimitada por el bucle DO 200. En este caso, Polaris determina que el bucle más externo representa una reducción, y por tanto aplica la técnica de paralelización de histograma descrita en [119]. Idénticas conclusiones se extraen del estudio de códigos de suma y producto de matrices dispersas, cuando el resultado se almacena sobre una matriz densa.

El problema es más interesante y presenta un mayor desafío, si el resultado de la suma o producto de dos matrices dispersas se almacena también de forma dispersa. En cuanto al algoritmo de suma, como ocurre en otros problemas dispersos, éste se suele dividir en dos etapas: suma simbólica, en la que se determina el patrón de la matriz resultado; y suma numérica que realiza las operaciones de suma apoyándose en la información obtenida de la etapa anterior [112].

En el ejemplo 5.10 tenemos la sección de código asociada a la suma numérica de dos matrices dispersas, `A` y `B` en formato CRS, resultando la matriz suma `C` también en formato CRS. Las tres matrices se representan mediante los vectores `VAL`, para los coeficientes, `IND`, para los índices de columna y `PTR`, que apunta a las filas, precedidos de las letras `A`, `B` y `C`, respectivamente. Como vemos, si la etapa de anáisis ha determinado inicialmente (en un tiempo despreciable) el patrón de `C`, la suma numérica tiene tres pasos: (i) desempaquetar una fila de `A` en un vector de trabajo `X`; (ii) sumar en `X` la fila de `B`; y (iii) empaquetar los elementos no nulos de `C` en la estructura CRS.

Es fácil ver que una vez conocido el patrón de la matriz resultado, el bucle 50 del ejemplo 5.10 es paralelo. Polaris analiza los vectores en los que se escribe, `CVAL` y `X`. En cuanto al primero, vemos que está indexado por la variable `IP` recorriendo

`CPTR(I):CPTR(I+1)-1`. Por tanto es aplicable la técnica de monotonía en el vector de punteros, `CPTR`, que va a consistir en una única prueba en tiempo de ejecución antes del bucle. En cuanto a `X`, de nuevo, se puede demostrar que es un vector de trabajo privatizable ya que el análisis de rangos junto con un test en tiempo de ejecución pueden comprobar que los coeficientes escritos en `X` en los bucles 20 y 30 son leídos y reinicializados a cero en el bucle 40.

Ejemplo 5.10 SUMA NUMÉRICA DE DOS MATRICES DISPERSAS

```
      DO 10 I=1,N
         X(I)=0
10    CONTINUE
      DO 50 I=1,N
         DO 20 IP=APTR(I),APTR(I+1)-1
            X(AIND(IP))=AVAL(IP)
20       CONTINUE
         DO 30 IP=BPTR(I),BPTR(I+1)-1
            X(BIND(IP))=X(BIND(IP)+BVAL(IP)
30       CONTINUE
         DO 40 IP=CPTR(I),CPTR(I+1)-1
            CVAL(IP)=X(CIND(IP))
            X(CIND(IP))=0
40       CONTINUE
50    CONTINUE
```

Comentarios análogos merece el estudio del producto disperso, cuya sección numérica presentamos en el ejemplo 5.11. En este caso, multiplicamos `A` $\times$ `B`, ambas con almacenamiento CRS y escribimos el resultado en la matriz dispersa `C` también CRS. La etapa de análisis previa proporciona el patrón de la matriz producto y consume un tiempo de ejecución insignificante. Al igual que en el caso anterior, las escrituras en las distintas filas de `C` se pueden ejecutar en paralelo al demostrar la monotonía del vector `CPTR`, así como la privatización del vector de trabajo se puede inferir del análisis de rangos discutidos en el párrafo anterior. De esta forma concluimos que también para este código, Polaris podría paralelizar automáticamente el bucle 50.

Ejemplo 5.11 PRODUCTO NUMÉRICO DE DOS MATRICES DISPERSAS

```
      DO 10 JP=1,N
         X(JP)=0
10    CONTINUE

      DO 50 I=1,N
         DO 20 JP=APTR(I),APTR(I+1)-1
            DO 30 KP=BPTR(AIND(JP)),BPTR(AIND(JP)+1)-1
               X(BIND(KP))=X(BIND(KP)+AVAL(JP)*BVAL(KP)
30          CONTINUE
20       CONTINUE

         DO 40 J=CPTR(I),CPTR(I+1)-1
            CVAL(J)=X(CIND(J))
            X(CIND(J))=0
40       CONTINUE
50    CONTINUE
```

5.4.2 Transposición de matrices

Otra situación distinta se presenta en el código de transposición de matrices descrito en el ejemplo 5.12. La matriz de entrada `A` de `N`×`M`, se almacena en formato CRS mediante los vectores (`AVAL, AIND, APTR`). A la salida, obtenemos la matriz transpuesta `AT` en una estructura CRS identificada por los vectores (`ATVAL, ATIND, ATPTR`).

Como vemos, aparte del cálculo del histograma de `A` en el bucle 20, también presenta un paralelismo trivial el bucle 50. En éste, el índice `AIND(J)` selecciona a través de una indirección la posición `K` en la que se va a escribir. Para ejecutar este bucle en paralelo sólo hace falta demostrar que: (i) `AIND` es un vector de permutación en el rango `(APTR(I),APTR(I+1)-1)`; y que (ii) por otro lado, no se va a escribir dos veces en la misma posición de `ATIND` y `ATVAL`, ya que cada vez que escribimos en una posición `K=ATPTR(J)`, `ATPTR(J)` se incrementa en una unidad (monotonía).

Ejemplo 5.12 TRANSPOSICIÓN DE UNA MATRIZ DISPERSA

```
      DO 10 I=3,M+1
         ATPTR(I)=0
10    CONTINUE
      DO 20 I=1,APTR(N+1)-1
         J=AIND(I)+2
         ATPTR(J)=ATPTR(J)+1
20    CONTINUE
      ATPTR(1)=1
      ATPTR(2)=1
      DO 30 I=3,M+1
         ATPTR(I)=ATPTR(I)+ATPTR(I-1)
30    CONTINUE
      DO 60 I=1,N
         DO 50 JP=APTR(I),APTR(I+1)-1
            J=AIND(JP)+1
            K=ATPTR(J)
            ATIND(K)=I
            ATVAL(K)=AVAL(J)
            ATPTR(J)=K+1
50       CONTINUE
60    CONTINUE
```

5.4.3 Factorización QR y Cholesky multifrontal

Aparte de la factorización dispersa LU, existen otros métodos directos de aplicación en problemas más particulares. Por ejemplo, para factorizar una matriz simétrica, la factorización de Cholesky consigue menores tiempos de ejecución que el algoritmo general LU. Por otro lado, la factorización QR, aunque provoca un mayor llenado y consume un mayor tiempo en la factorización, es de utilidad en problemas de mínimos cuadrados o en la factorización de matrices muy mal condicionadas.

En cuanto a la factorización QR, las variantes Gram-Schmidt Modificado y Householder, presentan en su algoritmo una etapa de actualización análoga a la que encontramos en la factorización LU. En estas dos variantes, el procesado de las columnas es independiente y el bucle que actualiza las `n-k` columnas en la iteración `k` es también paralelo. Por tanto, la discusión acerca de las técnicas necesarias para paralelizar automáticamente estos códigos es idéntica a la expuesta en 5.3.

Un comentario análogo merece la factorización de Cholesky, si utilizamos una aproximación general (ver esquema de la página 8). Para encarar otro tipo de organizaciones, hemos estudiado la aplicabilidad de nuestras técnicas a un algoritmo Cholesky multifrontal, ya que ésta es la solución más eficiente para factorizar matrices simétricas. El código secuencial, discutido en [65], utiliza rutinas SPARSPACK [72] para las etapas de reordenación, usando el algoritmo de mínimo grado, y de factorización simbólica. A continuación construyen el árbol de eliminación para pasar a la etapa de factorización numérica. En este punto se conoce el patrón de la matriz factorizada. La etapa de factorización numérica está delimitada por un bucle de índice `k` que recorre las columnas. En su interior, se distinguen dos situaciones: la columna `k` es la última de un supernodo (ver figura 1.1), o no. En el primer caso un bucle, DO 1020, aplica una rutina de actualización densa a las restantes columnas del supernodo. En la seguna situación, el bucle DO 1021 actualiza las columnas afectadas mediante llamadas a una rutina de actualización dispersa. Estos dos bucles consumen el 82% del tiempo de ejecución del algorimto completo. Dado que la estructura de datos asociada es CCS, tiene sentido aplicar las técnicas de detección de paralelismo y privatización presentadas. En este caso sólo es necesario probar el no solapamiento de las secciones (columnas) escritas en las actualizaciones para detectar que los dos bucles son paralelos. De nuevo, la detección de arrays monótonos crecientes usados para acotar bucles, se muestra útil.

El código paralelo, entendido como la salida simulada de una futura versión mejorada de Polaris, reporta una eficiencia en el SGI Power Challenge, del 55% en 16 procesadores y 60% en 8, si atendemos sólo a las prestaciones de los bucles paralelos. Dado que un 18% del código queda sin paralelizar, las prestaciones del código completo son inferiores (20% y 35% respectivamente). El algoritmo puede ser optimizado a mano si paralelizamos la rutina de reordenación de mínimo grado que consume un 11% del tiempo total (la factorización simbólica y la construcción del árbol de eliminación son menos significativas). El nuevo código paralelo cubre el 93% del tiempo de ejecución y la eficiencia es de 30% en 16 procesadores y 47% en 8. Sin embargo, no está claro aún la posibilidad de desarrollar nuevas técnicas para la paralelización automática del algoritmo de mínimo grado, aspecto que mantenemos en evaluación.

5.4.4 Otros códigos irregulares

Adicionalmente, otros códigos irregulares son abordados en [16], y recogidos aquí en la tabla 5.4. Tres de ellos están incluidos en el compendio de aplicaciones que motivan la extensión de HPF-2 [66]. El código de Lanczos, desarroyado en nuestro grupo se discute detalladamente en [149], mientras GCCG proviene del Instituto de Tecnología Software y Sistemas Paralelos de la Univ. de Viena.

Aparte del tamaño en líneas de código de cada programa, presentamos en las últimas dos columnas de esta tabla, la aceleración alcanzada en los 8 procesadores del SGI Challenge, para la versión paralela generada por PFA [137] y por Polaris.

Si empezamos con los algoritmos que presentan mejores resultados, encontramos GCCG y LANCZOS. Ambos contienen indirecciones únicamente en la parte derecha de las asignaciones, por lo que no aparecen problemas de dependencias. Adicionalmente

Programa	Descripción	Origen	líneas	PFA	Polaris
GCCG	Dinámica de fluidos	Vienna	407	6.32	8.49
LANCZOS	Cálculo de autovalores	Malaga	269	7.36	7.17
NBFC	Dinámica molecular	HPF-2	206	0.99	5.00
EULER	Euler en una malla 3D	HPF-2	1990	1.02	1.97 (5.38)
DSMC3D	Monte Carlo	HPF-2	1794	1.02	1.22 (4.95)

Tabla 5.4: Aceleración para otros códigos.

las reducciones que aparecen son del tipo escalar o reducciónes de "dirección única"[11]. Por tanto, ni Polaris ni PFA encuentran demasiado problema al generar un buen código paralelo.

En cuanto al algoritmo NBFC, el problema es distinto. El núcleo de la computación está dominado por un triple bucle anidado que realiza una operación de reducción. En este caso la reducción es de histograma[12] ya que aparece una indirección en el lado izquierdo de la asignación asociada a la reducción. Esta situación está contemplada actualmente en Polaris [119], aunque no en PFA, por lo que apreciamos una diferencia en las prestaciones de una y otra versión.

En los dos últimos códigos, EULER y DSMC3D, Polaris consigue mejores resultados que PFA debido a que contempla la técnica de reducción de histograma. Sin embargo, como vemos la mejora no es suficiente debido por un lado al gasto que introduce la reducción de histograma (ver ejemplo 5.5), y por otro a que estas reducciones no tienen el suficiente peso en términos de porcentaje sobre el tiempo total de ejecución. En DSMC3D, hemos aplicado manualmente la técnica que prueba el no solapamiento de las variables de inducción junto con la detección de generadores de números aleatorios, consiguiendo un incremento de la aceleración de 1.22 a 4.95. En el código de EULER se ha mostrado útil la técnica que prueba la existencia de vectores de permutación. Explotando esta alternativa hemos conseguido subir la aceleración de 1.97 a 5.38 en 8 procesadores.

5.4.5 Resumen de técnicas

En general, el paralelismo entre columnas o filas de una matriz dispersa es factible en muchas ocasiones, sobre todo dentro del ámbito del cálculo numérico.

Evidentemente, el compilador no puede inferir de forma inmediata, y únicamente de la información que encuentra en el código, que éste procesa columnas o filas de una matriz dispersa. Es importante "enseñarle" cual es el patrón de acceso típico en esta clase de códigos. De otra forma: no siempre es útil la aplicación de las técnicas de detección y privatización de paralelismo discutidas en 5.3. Por tanto, debemos diseñar el compilador de forma que sepa cuándo debe aplicarlas. En una primera aproximación, el compilador aplicará estas técnicas cuando encuentre el patrón mostrado en el ejemplo

[11] Es decir, la variable que se reduce es un único elemento de una array.
[12] Varias componentes del array se reducen en cualquier orden. Ver [119].

5.13, y presente en todos los códigos discutidos en esta sección.

Ejemplo 5.13 PATRÓN DEL PROBLEMA ESTUDIADO

```
            DO 10 I= ...
               ....
               DO 20 J=PTR(I),PTR(I+1)-1
                  ....
                  .... [ARRAY(IND(J))] .... [VAL(J)] ....
                  ....
20             CONTINUE
10          CONTINUE
```

En este patrón, los arrays entre corchetes indican que pueden aparecer opcionalmente, y los puntos suspensivos, representan cualquier sentencia o función válida en F77. Este patrón es claramente indicativo de que, con un porcentaje elevado de probabilidades, el código esta haciendo uso de una estructura CRS o CCS. El compilador intentara demostrarlo probando las propiedades que caracterizan este tipo de estructuras de datos, a saber: el vector `PTR` es monótono no-decreciente; el vector `IND` es de permutación en el rango `PTR(I):PTR(I+1)-1`; y el array de índices contiene valores en un determinado rango.

Si se demuestran estas propiedades, es posible sacar provecho del paralelismo normalmente asociado a la independencia en el procesado de las filas o las columnas de la matriz. Como vemos, este patrón presentado puede resultar muy genérico.

Otra aproximación intermedia entre el paralelismo de datos y la paralelización automática consiste en darle alguna información sobre la estructura de datos al compilador. Por ejemplo, asegurandole que los vectores `VAL`, `IND` y `PTR` representan una matriz dispersa en formato CCS, libera al compilador de tener que demostrar que PTR es no decreciente, que los valores contenidos en `IND` estan en el rango `1:n` donde `n` es la dimensión de la matriz, y que `IND` puede ser visto en la sección que representa cada columna como un vector de permutación. Es decir, le damos las propiedades implícitas en el formato de representación de los datos.

Por tanto, una primera solución puede consistir en dejar al compilador intentar encontrar el máximo paralelismo posible. Pero si en alguna de las fases de análisis simbólico el compilador no es capaz de demostrar lo que busca, puede formular la pregunta pertinente al usuario, para volver a compilar sobre un supuesto adicional.

En resumen, las cuatro técnicas útiles para paralelizar muchos de los códigos dispersos (y otros por extensión que presentan similares propiedades) se discuten a continuación. Adicionalmente, ya que el patrón del ejemplo 5.13 es indicativo pero muy general, adjuntamos a cada técnica el patrón particular que provoca la aplicación de dicha técnica.

Monotonía en los arrays que acotan bucles

El ejemplo 5.14 muestra el patrón típico que identifica cuando usar esta técnica. Es claro que las escrituras en VAL son independientes para cada iteración `I` si el vector

PTR es mónotono no decreciente. Más formalmente, se debe cumplir la expresión:

$$\bigcap_{i=1}^{n} [PTR(I) : PTR(I+1) - 1] = \emptyset \tag{5.1}$$

Ejemplo 5.14 PATRÓN PARA APLICAR "MONOTONÍA"

```
                DO 10 I= ...
                   DO 20 J=PTR(I),PTR(I+1)-1
                      VAL(J)=....
20                 CONTINUE
10              CONTINUE
```

En general para detectar si un vector, `PTR` es no decreciente en un rango `(1:n)` se estudian tres situaciones:

- Si el vector no se modifica durante la ejecución del programa, es suficiente un sencillo test en tiempo de ejecución, consistente en un bucle al comienzo del programa que chequea que `PTR(I)` $\leq$ `PTR(I+1)` para `I=1,n`.
- Si las componentes del vector `PTR` son asignadas durante la ejecución, se hace un estudio simbólico de las variables relacionadas con la escritura en `PTR`. Suponiendo que la condición es cierta al comienzo del programa (lo cual es probado posteriormente mediante el test del punto anterior), en muchas situaciones se puede inferir que se mantiene invariante en ciertos puntos del programa.
- En el caso en el que el punto anterior no pueda decidir sobre la monotonía del vector, el compilador puede formular la pregunta correspondiente al usuario para volver a compilar con esa información.

No solapamiento de variables de inducción

El caso en el que la escritura en un array se realiza dentro de un bucle, indexada por una variable de inducción en lugar de la variable del índice, se presenta en el ejemplo 5.15.

Ejemplo 5.15 PATRÓN PARA APLICAR "NO SOLAPAMIENTO"

```
      DO 10 I= ...                          DO 20 J=M, N
         ...                                   DO 30 I=PTR(J),PTR(J+1)-1
         VAL(J)=VAL(I) ...                        VAL(K)=VAL(I) ...
         ....                                     K=K + (valor_positivo)
         J=J + (valor_positivo)          30    CONTINUE
10    CONTINUE                           20    CONTINUE
```

Para determinar si el bucle `I` del ejemplo anterior es paralelo se estudian dos situa-

ciones:

- Si podemos expresar la variable `J` en función del índice `I` decimos que el bucle se puede expresar en su forma cerrada. En este caso se sustituye `J` por la función de `I` y se aplican los tests de dependencias estándar. Adicionalmente, si el último valor de `J` se utiliza fuera del bucle, aplicamos técnicas de "valor último" [119].
- En caso contrario, por ejemplo cuando no se conoce la cantidad `valor_positivo` en tiempo de compilación, una primera aproximación que demuestra la inexistencia de dependencias de flujo o anti-dependecias consiste en probar que el rango de `J` no solapa el rango de `I`.

Este segundo caso se puede presentar en una gran variedad de situaciones complicando significativamente el estudio general. Por tanto nuestra solución se centra en un caso particular del anterior, caracterizado por el bucle `J` del ejemplo 5.15 y siempre que se pueda estimar una cota máxima del `valor_positivo` en todas la iteraciones. En este supuesto, para poder ejecutar el bucle `J` en paralelo debemos:

1. Encontrar el valor máximo y mínimo de `I` para todo el rango de `J`. Esto se puede inferir de recorrer el vector `PTR` en tiempo de ejecución, o más económicamente, si se ha probado la monotonía de `PTR` usando la técnica anterior.
2. Determinar el valor mínimo de la variable de inducción `K` y estimar en tiempo de compilación la formula que especifica el valor máximo de `K` en el caso peor.
3. Aplicar el test en tiempo de ejecución que prueba que

$$(min_I > max_K) \wedge (max_I < min_K) \tag{5.2}$$

Como antes, en caso de que en esta situación no pueda ser alcanzada, se puede preguntar al usuario por la información necesaria para completar alguno de los puntos de la lista anterior.

Detección de vectores de permutación

En los casos en los que se ha encontrado un patrón similar al presentado en 5.13, y/o adicionalmente encontramos una sección de código con la estructura del ejemplo 5.16, podemos aplicar esta técnica si las anteriores no han dado resultados positivos.

Ejemplo 5.16 PATRÓN PARA APLICAR "VECTORES DE PERMUTACIÓN"

```
      DO 10 I= ...
         DO 20 J=PTR(I),PTR(I+1)-1
            VAL(IND(J))=....
20       CONTINUE
10    CONTINUE
```

Si el bucle `I` de este ejemplo, no puede ser paralelizado por las técnicas anteriores,

es posible paralelizar el bucle `J` si demostramos que `IND` es un vector de permutación en el rango `PTR(I):PTR(I+1)-1`. Contemplamos dos situaciones:

- Si el vector `IND` sólo es leído dentro del bucle `I`, y sólo es escrito una vez durante la fase de inicialización del programa, es suficiente con un test en tiempo de ejecución que compruebe que la condición es cierta después de la mencionada inicialización.
- Si el vector `IND` se modifica en otros puntos del programa se puede realizar un estudio simbólico que determine si los valores escritos en `IND` son distintos en cada rango `PTR(I):PTR(I+1)-1`.

De nuevo, si estos tests fallan, se puede pedir información al usuario.

Extensión de las técnicas de privatización

Como deciamos en la subsección 5.3.2, la privatización de variables es definitiva a la hora de romper dependencias de salida o anti-dependencias generadas por bucle. Distinguimos dos situaciones diferentes: la privatización de variables o arrays de trabajo por un lado, y por otro la privatización de arrays accedidos por variables de inducción, ambos discutidos en 5.3.2. Cada uno de ellos presenta un problema particular no contemplado hasta ahora por los compiladores paralelos.

En cuanto a la privatización de un array de trabajo `W`, el problema se presenta cuando pretendemos probar que los coeficientes escritos de `W` coindicen con los coeficientes leídos en cada iteración. El uso del array `W` puede estar caracterizado por un código como el presentado en el ejemplo 5.7 en cuyo caso un análisis simbólico acompañado de un sencillo test en tiempo de ejecución soluciona el problema (ver sección 5.3.2). Sin embargo, existen situaciones más complicadas como las que presentan los algoritmos de suma y multiplicación de matrices dispersas en los ejemplos 5.10 y 5.11, respectivamente. En este caso, el test que garantiza que el array de trabajo es privatizable puede consistir en la ejecución simbólica del bucle que se quiere paralelizar, eliminando las operaciones aritméticas y chequeando si se cumplen las condiciones para privatizar `W` en la subsiguiente ejecución del bucle original.

Esta aproximación basada totalmente en test en tiempo de ejecución puede llegar a ser demasiado costosa. En esta línea, Rauchwerger [123] propone una ejecución especulativa en paralelo del bucle. Simultáneamente, se chequea la condición de privatización y en caso de que ésta resulte falsa, se desecha el trabajo realizado y el bucle se vuelve a ejecutar en secuencial. Obviamente, es necesario mantener una copia del estado del proceso antes de comenzar el bucle. En la aproximación que proponemos, la alternativa elegida, introduce un menor coste añadido en el tiempo final de ejecución, pero complica el compilador. El entorno básico a partir del cual construir la técnica completa, está especificado en la memoria de tesis de Peng Tu [151] dedicada a la privatización y al análisis del flujo de datos. Las técnicas propuestas en este trabajo, ya implemetadas en Polaris, cubren la privatización de arrays accedidos en un rango continuo de posiciones. Como hemos visto en los ejemplos, en nuestro caso, cuando el array de trabajo se accede mediante indirecciones, las técnicas de Tu no son suficientes.

Por tanto, el esquema final va a consistir en una complementación de las ideas de Tu con los trabajos de Blume, orientados a la propagación de rangos [33], y algunas de las técnicas en tiempo de ejecución discutidas en [123].

Por otro lado, la privatización de arrays accedidos por variables de inducción que no pueden ser expresadas en su forma cerrada es más directa, siempre que se haya probado que no existe otro tipo de dependencias que las que provoca la variable de inducción. Sin embargo, ahora la complicación aparece al ser necesaria la generación automática de la etapa de copy-out. Este aspecto no esta aún implementado por Polaris pero no presenta ninguna complejidad adicional. De hecho, la etapa de copy-out puede ser interpretada como un caso particular de operación de reducción, ya evaluada en [119].

Conclusiones

Sabedores de la importancia de la computación dispersa e irregular, hemos profundizado en esta tesis en la paralelización de este tipo de problemas. Hemos elegido la resolución de ecuaciones dispersas como caso de estudio por constituir un problema con muchos de los paradigmas representativos de la computación irregular. En resumen, en nuestra búsqueda hemos programado en C, F90 y F77, para plataformas como el CRAY T3D, T3E y el SGI Power Challenge, hemos utilizado librerías de comunicación PVM, MPI y SHMEM, los entornos de paralelismo de datos CRAFT y HPF-2, hemos extendido la librería DDLY y usado la herramienta Cocktail en el desarrollo del compilador, hemos comparado con CHAOS, y finalmente hemos identificado algunas limitaciones de los paralelizadores PFA y Polaris, proponiendo soluciones.

Las principales aportaciones y las líneas de investigación que pretendemos seguir en el futuro se discuten a continuación.

Principales aportaciones

Volviendo a la analogía, comentada en el prefacio, entre la investigación y la búsqueda del camino apropiado en una región desconocida, en esta tesis hemos topografiado tres rutas para alcanzar soluciones al problema de la resolución paralela de sistemas de ecuaciones dispersos: la paralelización manual, semi-automática y automática.

Paralelización manual

- Hemos desarrollado un algoritmo secuencial, SpLU, para resolver sistemas de ecuaciones dispersos, que incluye las etapas de análisis-factorización y resolución triangular. El algoritmo secuencial conduce en muchos casos a un error de factorización y a un llenado menor que el que consigue la rutina estándar MA48. En general, en los experimentos realizados nuestro algoritmo SpLU ha resultado ser más estable que la rutina MA48. Adicionalmente el tiempo de factorización puede llegar a ser bastante menor que el de la MA48.

- Hemos comprobado que la rutina MA48 no puede ser paralelizada eficientemente debido a su organización *left-looking*. Sin embargo nuestro algoritmo exhibe un alto grado de paralelismo, no sólo por basarse en una aproximación *right-looking*,

sino también por explotar la dispersión de la matriz agrupando un cierto número de pivots compatibles. El algoritmo paralelo presenta una eficiencia muy alta, superior a las que proporcionan otros trabajos previos en la misma línea.

Paralelización semi-automática

- Hemos identificado las limitaciones de los compiladores de paralelismo de datos CRAFT y HPF a la hora de expresar códigos dispersos. Básicamente estas limitaciones se derivan de la imposibilidad de estas herramientas para manejar estructuras de datos dinámicas o comprimidas. Esto impide la paralelización eficiente de algoritmos dispersos dado que se apoyan en las mencionadas estructuras de datos. Nuestra aportación ha consistido en extender la capacidad de HPF-2, mediante la inclusión de una directiva (`SPARSE`) y la extensión semantica de otras ya existentes. De esta forma, permitimos al programador escribir códigos en F90 basados en estructuras de datos con listas enlazadas o vectores comprimidos, delegando en el compilador la tarea de generar la versión paralela. La idea más importante de nuestra extensión es permitir al programador especificar mediante directivas, no sólo la distribución de los datos, sino también la estructura de estos datos en memoria. Las extensiones son mínimas y simples pero al mismo tiempo suficientes para proporcionar una gran flexibilidad al programador.

- Adicionalmente, hemos desarrollado parte del módulo de compilación que completa el compilador de HPF-2 para soportar las nuevas extensiones propuestas. Más precisamente, disponemos del analizador léxico y sintáctico para reconocer la directiva añadida, junto con las rutinas en tiempo de ejecución encargadas de la distribución de datos y comunicaciones. Para construir los analizadores hemos hecho uso de la herramienta Cocktail para diseño de compiladores, mientras que las rutinas en tiempo de ejecución se han construido siguiendo la filosofía de las DDLY.

- Hemos simulado la salida paralela generada por nuestro compilador, partiendo de un algoritmo secuencial en F90 anotado con las nuevas directivas, comparable en prestaciones a la rutina MA48. La salida paralela ha resultado ser bastante eficiente, presentando el mismo error de factorización y llenado que la versión secuencial. También hemos comparado nuestra aproximación con la técnica Inspector-Executor implementada en la librería CHAOS para la paralelización de códigos irregulares, resultando esta última claramente insuficiente.

Paralelización automática

- Hemos analizado la salida de dos paralelizadores automáticos, PFA y Polaris, cuando la entrada es un código de factorización dispersa en F77. Ninguno de los dos obtiene buenos resultados debido a ciertas limitaciones en las técnicas de detección y paralelización de ambos compiladores. Para resolverlo hemos introducido cuatro técnicas para la detección de dependencias y paralelización automática de códigos dispersos: monotonía en los arrays que acotan bucles, no

solapamiento de variables de inducción, detección de vectores de permutación, y extensión de las técnicas de privatización.

- Hemos aplicado estas técnicas a nuestro algoritmo de factorización en F77 obteniendo eficiencias muy altas en la SGI Power Challenge de memoria compartida. Adicionalmente estas técnicas se han mostrado válidas en otros códigos irregulares dispersos, justificándose así su necesaria implementación en los paralelizadores automáticos.

Líneas de investigación futura

Nuestro algoritmo SpLU para la resolución paralela de sistemas de ecuaciones dispersos puede ser mejorado en dos puntos principales. El primero está orientado a reducir el coste de las comunicaciones aplicando una distribución cíclica por bloques en lugar de una cíclica pura. El segundo se centra en reducir el movimiento de datos y facilitar el insertado de entradas utilizando una estructura de listas enlazadas desordenadas tanto por filas como por columnas. Estos dos aspectos unidos a un mayor cuidado en el aprovechamiento de la cache pueden resultar en mejores prestaciones.

Sin embargo, más que optimizar un problema particular, nos parece más interesante simplificar las técnicas de programación paralela ya sea de forma semi-automática o automática. Nosotros hemos abordado la construcción de un módulo de compilación que procesa sólo la directiva `SPARSE` introducida, pero también es necesario contemplar extesiones para problemas no estructurados. En cuanto a la aproximación automática, las técnicas propuestas para poder paralelizar códigos basados en estructuras de datos comprimidas, ya están siendo implementadas en Polaris en colaboración con el grupo de investigación dirigido por el profesor David Padua en Illinois.

En la misma línea de la paralelización automática, aparte de las estructuras de datos comprimidas, es necesario trabajar en otras situaciones con estructuras de datos dinámicas basadas en punteros, como las listas, árboles, etc. La detección automática de paralelismo de este tipo de problemas presenta un gran desafío al que nos queremos enfrentar. Por ejemplo, los códigos de factorización LU basados en listas enlazadas en C o en F90 no pueden ser paralelizados automáticamente con la tecnología actual.

Dando un paso más en complejidad y en aras de facilitar la tarea al programador, no descartamos la idea de dedicar algunos esfuerzos al campo de los reestructuradores de códigos densos secuenciales en dispersos y paralelos. Bik y Pingali presentan algunas soluciones para generar un código disperso secuencial a partir del código denso asociado, pero por el momento, sus resultados aún dejan un amplio margen para la investigación en este campo. La ventaja que presenta abordar la paralelización automática de un código disperso partiendo de la versión densa, estriba principalmente en la mayor cantidad de información que puede obtener el compilador. Más precisamente, el análisis de dependencias de algoritmos numéricos dispersos se simplifica si partimos de la versión densa.

Apéndice A

Factorización por bloques

Supongamos que mediante un algoritmo de reordenación conseguimos permutar una matriz dispersa para que tome una forma triangular inferior por bloques (fig. 3.1 (c)). En este caso, podemos factorizar cada bloque diagonal independientemente de los demás, al igual que explicábamos en la subsección 3.1.1. Si representamos la matriz triangular por bloques como se muestra en la ecuación A.1, la solución de la ecuación $Ax = b$, puede ser resuelta por un simple proceso de substitución hacia adelante, como se muestra en la ecuación A.2.

$$A = \begin{pmatrix} B_{11} & & & & & \\ B_{21} & B_{22} & & & & \\ B_{31} & B_{32} & B_{33} & & & \\ \cdot & \cdot & \cdot & \cdot & & \\ \cdot & \cdot & \cdot & \cdot & \cdot & \\ B_{N1} & B_{N2} & B_{N3} & \cdot & \cdot & B_{NN} \end{pmatrix} \tag{A.1}$$

$$B_{ii}\ x_i = b_i - \sum_{j=1}^{i-1} B_{ij}\ x_j, \qquad i = 1, 2, ..., N \tag{A.2}$$

De esta forma, sólo es necesario factorizar los bloques B_{ii}, operación que puede ser realizada en paralelo, con la propiedad de que el llenado quedará confinado en cada bloque. Sin embargo, como se desprende de la ecuación A.1 y al contrario que en el caso de matrices diagonales por bloques, la etapa de resolución triangular debe ser resuelta secuencialmente. Es decir, no podemos resolver $B_{ii}x_i$ hasta que tengamos todas las $x_j,\ j = 1, 2, ..., i-1$.

Si partimos de una matriz tridiagonal por bloques como la de la figura 3.1 (b), en principio, puede parecer que los bloques superior y a la izquierda de cada bloque diagonal con elementos distintos de cero, implican una dependencia entre bloques diagonales que impide la factorización en paralelo. Sin embargo podemos trasladar esa dependencia a la etapa de de resolución triangular, de forma que sólo necesitemos factorizar cada

uno de los bloques diagonales, sin dependencias entre ellos, realizando, por contra, la etapa de resolución triangular en secuencial.

Para ello, factorizaremos la matriz A de la siguiente forma:

$$\begin{pmatrix} A_{11} & A_{12} & & & \\ A_{21} & A_{22} & A_{23} & & \\ & A_{32} & A_{33} & \cdot & \\ & & \cdot & \cdot & \cdot \\ & & & \cdot & A_{NN} \end{pmatrix} = \begin{pmatrix} I & & & & \\ A_{21}D_1^{-1} & I & & & \\ & \cdot & \cdot & & \\ & & \cdot & \cdot & \\ & & & \cdot & I \end{pmatrix} \cdot \begin{pmatrix} D_1 & A_{12} & & & \\ & D_2 & A_{23} & & \\ & & D_3 & \cdot & \\ & & & \cdot & \cdot \\ & & & & D_N \end{pmatrix}$$

donde

$$D_1 = A_{11} \tag{A.3}$$
$$D_i = A_{ii} - A_{i,i-1}D_{i-1}^{-1}A_{i-1,i}, \qquad i = 2, ..., N \tag{A.4}$$

Mediante esta factorización podemos resolver el conjunto de ecuaciones $Ax = b$ mediante una sustitución hacia adelante:

$$y_1 = b_1 \tag{A.5}$$
$$y_i = b_i - A_{i,i-1}D_{i-1}^{-1}y_{i-1}, \qquad i = 2, ..., N \tag{A.6}$$

seguida de una substitución hacia atrás:

$$x_N = D_N^{-1}y_n \tag{A.7}$$
$$x_i = D_i^{-1}(y_i - A_{i,i+1}y_{i+1}), \qquad i = N-1, ..., 1 \tag{A.8}$$

Por tanto, es suficiente con mantener los bloques no diagonales de A sin modificar, y factorizar los bloques diagonales, D_i de forma:

$$D_i = L_iU_i, \qquad i = 1, ..., N,$$

operación que podemos realizar totalmente en paralelo. Sin embargo, como se aprecia en las ecuaciones A.6 y A.8, las substituciones hacia adelante y hacia atrás deberán ejecutarse secuencialmente. Por último, hacer notar que estas ecuaciones (de la A.4 a la A.8) se pueden generalizar también al caso en que la matriz sea en banda o banda variable por bloques, es decir, tengamos otros bloques con elementos no nulos fuera de la diagonal, aparte de los mencionados.

Finalmente, las matrices por bloques con borde, como las de las figuras 3.1 (d) y (e), pueden seguir el mismo tratamiento que las anteriores, salvo por el último bloque (el borde). Por tanto, es posible factorizar en paralelo todos los bloques diagonales excepto el último, que será procesado una vez todos los anteriores estén factorizados.

Bibliografía

[1] J.C. Adams, W.S Brainerd, J.T. Martin, B.T. Smith, and J.L. Wagener. *Fortran 90 Handbook. Complete ANSI/ISO Reference*. McGraw-Hill Book Company, 1992.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1985.

[3] G. Alaghband. Parallel pivoting combined with parallel reduction and fill-in control. *Parallel Comput.*, 11:201–221, 1989.

[4] G. Alaghband. Parallel sparse matrix solution and performance. *Parallel Computing*, 21(9):1407–1430, 1995.

[5] R.J. Allan and M.F. Guest (Eds.). Parallel application software on high performance computers. The IBM SP2 and Cray T3D. The CCLRC HPCI Centre at Daresbury Laboratory, January 1996.

[6] F.L. Alvarado. Manipulation and visualization of sparse matrices. *ORSA J. Computing*, 2:186–207, 1989.

[7] F.L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM J. Sci. Comput.*, 14(2):446–460, 1993.

[8] P.R. Amestoy, T.A. Davis, and I.S. Duff. An approximate minimun degree ordering algorithm. Technical Report TR/PA/95/09, CERFACS, Toulouse, France, 1995. To appear in SIAM J. Matrix Analysis and Applications.

[9] P.R. Amestoy and I.S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. Supercomputer Applics.*, 7:64–82, 1993.

[10] P.R. Amestoy, I.S. Duff, and C. Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Numerical Linear Algebra with Applications*, 3(4):275–300, 1996.

[11] E. Anderson, Z. Bai, C. Bishof, J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. Technical Report CS-90-101, Computer Science, University of Tennessee, April 1990.

[12] E. Anderson and Y. Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1(1):73–95, 1989.

[13] M. Arioli and I.S. Duff. *Reliable Numerical Computation*, pages 207–226. M.G. Cox and S. Hammarling, eds., Oxford University Press, Oxford, 1990.

[14] R. Asenjo, G. Bandera, G.P. Trabado, O. Plata, and E.L. Zapata. Iterative and direct sparse solvers on parallel computers. In *Euroconference: Supercomputations in Nonlinear and Disordered Systems: Algorithms, Applications and Architectures*, pages 23–28, San Lorenzo de El Escorial, Madrid, Spain, Sep. 1996.

[15] R. Asenjo, R. Doallo, J.P. Tourino, O.Plata, and E.L. Zapata. Parallel sparse computations involving pivoting and fill-in. Submitted to *IEEE Transaction on Parallel and Distributed Systems*.

[16] R. Asenjo, E. Gutierrez, Y. Lin, D. Padua, W. Pottenger, and Emilio Zapata. On the automatic parallelization of sparse and irregular Fortran codes. Technical Report 1512, Univ. of Illinois at Urbana-Champaign, CSRD, Dec 1996.

[17] R. Asenjo, L.F. Romero, M. Ujaldón, and E.L. Zapata. *Sparse Block and Cyclic Data Distributions for Matrix Computations*, pages 359–377. Elsevier Science, Amsterdam, The Netherlands, J.J. Dongarra, L. Grandinetti, G.R. Joubert and J.Kowalik, Eds., Cetraro, Italy, June 1994.

[18] R. Asenjo, G.P. Trabado, M. Ujaldón, and E.L. Zapata. Compilation issues for irregular problems. In *Workshop on Parallel Programming Environments for High-Performance Computing*, pages 187–199, L'Alpe d'Huez, France, April 1996.

[19] R. Asenjo, M. Ujaldón, and E. L. Zapata. *SpLU – Sparse LU Factorization. HPF-2 Scope of Activities and Motivating Applications. Version 0.8*. High Performance Fortran Forum, November 1994.

[20] R. Asenjo, M. Ujaldón, and E.L. Zapata. Parallel WZ factorization on mesh multiprocessors. *J. Microprocessing and Microprogramming*, 38(5):319–326, September 1993.

[21] R. Asenjo and E. L. Zapata. Sparse LU factorization on the CRAY T3D. In *Int'l Conf. on High-Performance Computing and Networking*, number 919 in LNCS, pages 690–696, Milan, Italy, May 3-5 1995. Springer-Verlag, Berlin, Germany, B. Hertzberger and G. Serazzi, Eds.

[22] C. Ashcraft and J.W.H. Liu. Rosbust ordering of sparse matrices using multisection. Technical Report CS-96-01, Dept. of Computer Science, York Univ. Ontario, Canada, 1996.

[23] G. Bandera, G.P. Trabado, and E.L. Zapata. Extending data-parallel languages for irregularly structured applications. In *NATO Advanced Research Workshop on High Performance Computing: Technology and Applications*, pages 235–251, Cetraro, Italy, 1996.

[24] R. Barret, M. Berry, T. Chan, J. Demmel, J. Doato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution fo Linear Systems*. SIAM, Philadelphia, 1994.

[25] R. Barriuso and A. Knies. *SHEM User's Guide for Fortran*. Cray Research, Inc., August 1994. Revison 2.2.

[26] Abdelhamid Benaini. The WW^T factorization of dense and sparse matrices. *Intern. J. of Computa. Math.*, 1994. Submitted.

[27] M.J. Berger and S.H. Bokhari. A partitioning startegy for nonuniform problems on multiprocessors. *IEEE Trans. Comput.*, 36(5):570–580, 1987.

[28] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, 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, G.Swanson, R. Goodrum, and J. Martin. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[29] A. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, University of Leiden, The Netherlands, 1996.

[30] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide*, 1997. Copia impresa disponible de SIAM o en internet: http://www.netlib.org/scalapack/index.html.

[31] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, Dec 1996.

[32] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, 1994.

[33] William Joseph Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization*. PhD thesis, Univ. of Illinois at Urbana-Champaign. CSRD., June 1995.

[34] R. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra. Matrix market version 3.0. At URL http://math.nist.gov/MatrixMarket.

[35] S. Booth, J. Fisher, N. MacDonald, P. Maccallum, E. Minty, and A. Simpson. *Parallel Programming on the Cray T3D*. Edinburgh Parallel Computing Centre, University of Edinburgh, U.K., September 1994.

[36] P. Brezany, K. Sanjari, O. Cheron, and E. Van Konijnenburg. Processing irregular codes containing arrays with multi-dimensional distributions by the PREPARE HPF compiler. In *Int'l. Conf. on High-Performance Computing and Networking (HPCN'95)*, pages 526–531, Milan, Italy, 1995. Springer-Verlag, LNCS 919.

[37] D.A. Calahan. Parallel solution of sparse simultaneous linear squations. In *11th Annual Allerton Conference on Circuits and System Theory*, pages 729–735. University of Illinois, 1973.

[38] W.J. Camp, S.J. Plimpton, B.A. Hendrickson, and R.W. Leland. Massively parallel methods for engineering and science problems. *Communications of the ACM*, 36(5):570–580, 1994.

[39] Augusto Castro. *Análisis de Distribuciones de Matrices Dispersas en Multiprocesadores con Topología Malla.* Facultad de Informática, Univ. de Málaga, June 1995. Proyecto Fin de Carrera, Directores: R. Asenjo y E.L. Zapata.

[40] Koelbel C.H., D.B. Loveman, R.S Schreiber, G.L. Steele, and M.E. Zosel. *The High Performance Fortran Handbook.* Scientific and Engineering Computation. The MIT Press, Cambridge, Massachusetts, 1994.

[41] B. Chapman, P. Mehrotra, and H. Zima. HPF+: A new language and implementation mechanisms for the support of advanced irregular applications. In *6th Works. on Compilers for Parallel Computers*, pages 195–206, Aachen, Germany, 1996.

[42] A.L. Cheung and A.P. Reeves. Sparse data representation. In *Proceedings Scalable High Performance Computing Conference*, 1992.

[43] E. Chu and A. George. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Comput.*, 5:65–74, 1987.

[44] J.M. Conroy, S.G. Kratzer, and R.F. Lucas. Data-parallel sparse matrix factorization. In J.G. Lewis, editor, *Proceedings 5th SIAM Conference on Linear Algebra*, pages 377–381, Philadelphia, 1994. SIAM Press.

[45] Cray Research. *CRAY T3D Technical Summary*, 1993.

[46] Cray Research, Inc. *CRAY T3E Applications Programming. TR-T3EAPPL(B)*, 2.0 edition, August 1995. Software Education Services.

[47] T. A. Davis. *A parallel algorithm for sparse unsymmetric LU factorization.* PhD thesis, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, IL, September 1989.

[48] T. A. Davis and P. C. Yew. A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization. *SIAM J. Matrix Anal. Appl.*, 11:383–402, 1990.

[49] T.A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report RAL-TR-93-036, Rutherford Appleton Laboratory, Chilton Didcot Oxon OX11 0QX, 1993. To appear in SIAM J. Matrix Analysis and Applications.

[50] Tim Davis. Sparse matrix collection. At URL http://www.cise.ufl.edu/ davis/.

[51] J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li, and J.W.H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB//CSD-95-883, Computer Science Division, U.C. Berkeley, Berkeley, California, July 1995.

[52] R. Doallo, B.B. Fraguela, J. Touriño, and E.L. Zapata. Parallel sparse modified Gram-Schmidt QR decomposition. In *Int'l. Conf. on High-Performance Computing and Networking (HPCN'96)*, pages 646–653, Brussels, Belgium, 1996. Springer-Verlag, LNCS 1067.

[53] J.J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK User's Guide*. SIAM Pub. Philadelphia, 1979.

[54] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.

[55] Iain S. Duff. Sparse numerical linear albegra: Direct methods and preconditioning. Technical Report RAL-TR-96-047, Rutherford Appleton Laboratory, Chilton Didcot Oxon OX11 0QX, April 1996. State of the Art in Numerical Analysys Meeting, York. Report availabe by anonymous ftp (seamus.cc.rl.ac.uk).

[56] I.S. Duff. MA32 a package for solving sparse unsymmetric systems using the frontal method. Technical Report R8730, AERE Harwell, HMSO, 1981.

[57] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, U.K., 1986.

[58] I.S. Duff, R.G. Grimes, and J.G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection (Release I). Technical report, CERFACS, Toulouse, France, 1992.

[59] I.S. Duff, M. Marrone, G. Radicati, and C. Vittoli. A set of level 3 basic linear algebra subprograms for sparse matrices. Technical Report RAL-TR-95-049, Rutherford Appleton Laboratory, 1995.

[60] I.S. Duff and J.K. Reid. MA48, a fortran code for direct solution of sparse unsymmetric linear systems of equations. Technical Report RAL-93-072, Rutherford Appleton Laboratory, October 1993.

[61] I.S. Duff and J.K. Reid. The design of MA48: A code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Math. Softw.*, 22(2):187–226, June 1996.

[62] S.C. Eisenstat, M.C. Gursky, M.H. Schultz, and A.H. Sherman. Yale sparse matrix package. *Int. J. Num. Methods in Eng.*, 18:1145–1151, 1982.

[63] S.C. Eisenstat and J.W.H. Liu. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM J. Scientific and Statistical Computing*, 14(1):253–257, January 1993.

[64] T.M.R. Ellis, I.R. Philips, and T.M. Lahey. *Fortran 90 Programming*. International Computer Science. Addison-Wesley Publishing Company, 1994.

[65] K. Eswar, C.-H. Huang, and P. Sadayappan. *Cholesky. HPF-2 Scope of Activities and Motivating Applications. Version 0.8.* High Performance Fortran Forum, November 1994. Technical Report CRPC-TR94492. Rice University.

[66] I. Foster, R. Schreiber, and P. Havlak. *HPF-2 Scope of Activities and Motivating Applications. Version 0.8.* High Performance Fortran Forum, November 1994. Technical Report CRPC-TR94492. Rice University.

[67] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C-W. Tseng, and M. Wu. Fortran D language specification. Technical Report COMP TR90-141, Computer Science Dept., Rice University, 1990.

[68] C. Fu and T. Yang. Sparse LU factorization with partial pivoting on distributed memory machines. In *ACM/IEEE Supercomputing'96*, Pittsburgh, USA, November 1996.

[69] K. Gallivan, B. Marsolf, and H.A.G. Wijshoff. Solving large nonsymmetric sparse linear systems using MCSPARSE. *Parallel Computing*, 22(10):1291–1333, 1996.

[70] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Refernce Manual.* Engineering Physic and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1993.

[71] G. A. Geist and C. H. Romine. LU factorization algorithm on distributed-memory multiprocessor architecture. *SIAM J. Sci. Statist. Comput.*, 9:639–649, 1989.

[72] A. George and E. Ng. SPARSPACK: Waterloo sparse matrix package user's guide for SPARSPACK-B. Technical Report CS-84-37, Dept. of Computer Science. Univ. of Waterloo, 1984.

[73] J.R. Gilbert and J.W.H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM J. Matrix Analysis and Applications*, 14:334–354, 1993.

[74] J.R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sientific and Estatistical Computing*, 9(5):862–874, September 1988.

[75] M.B. Girkar, M. Haghighat, C.L. Lee, B.P. Leung, and C.A. Shouten. *Parafrase-2 Programmers's Manual.* CSRD, Univ. of Illinois, 1992.

[76] G.H. Golub and C.F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, MD, 2 edition, 1989.

[77] P. González, J.C. Cabaleiro, and T.F. Pena. Solving sparse triangular systems on distributed memory multicomputers. In *6th Euromicro Workshop on Parallel and Distributed Processing*, January 1998. To appear.

[78] J. Grosh. Cocktail: Toolbox for Compiler Construction. Toolbox Introduction. Technical Report 25, Univ. of Karlsruhe, Germany, November 1994.

[79] J. Grosh. Cocktail: Toolbox for Compiler Construction. Lark - An LR(1) Parser Generator with Backtracking. Technical Report 32, Univ. of Karlsruhe, Germany, August 1996.

[80] J. Grosh. Cocktail: Toolbox for Compiler Construction. Rex - A Scanner Generator. Technical Report 5, Univ. of Karlsruhe, Germany, April 1996.

[81] J. Grosh and B.Vielsack. Cocktail: Toolbox for Compiler Construction. The Parser Generators Lalr and Ell. Technical Report 8, Univ. of Karlsruhe, Germany, July 1992.

[82] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. Design and implementation of a scalable parallel direct solver for sparse symmetric positive definite systems. In *Eighth SIAM Conference on Parallel Processing*, 1997.

[83] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5), 1995. Available at URL: http://www.cs.umn.edu/~kumar.

[84] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.W. Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996. 1996.

[85] High Performance Fortran Forum. *High Performance Language Specification, Ver. 1.0*, 1993.

[86] High Performance Fortran Forum. *High Performance Language Specification, Ver. 2.0*, 1996.

[87] N.J. Higham. Stability of parallel triangular system solvers. *SIAM J. Sci. Comput.*, 16(2):400–413, 1995.

[88] R.W. Hockney and E.A. Carmona. Comparison of communications on the Intel iPSC/860 and Touchstone Delta. *Parallel Computing*, 18:1067–1072, 1992.

[89] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. Computing Science. McGraw-Hill, Mc.Graw-Hill, Inc., New York, USA, 1993.

[90] J.M. Jimenez, J. Cardenal, and I.S. Duff. A projection method for the solution of linear least-squares problems. Technical Report RAL-TR-96-013, Rutherford Appleton Laboratory, 1996.

[91] T. Johnson and T.A. Davis. Parallel buddy memory management. *Parallel Processing Letters*, 2(4):391–398, 1992.

[92] M.T. Jones and P.E. Plassmann. *BlockSolve95 Users Manual: Scalable Library Software for the Parallel Solution of Sparse Linear Systems.* Argonne National Laboratory, Argonne, Illinois, USA., June 1997.

[93] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *9th ACM International Conference on Supercomputing*, Barcelona, 1995.

[94] J. Koster. *Parallel solution of sparse systems of linear equations on a mesh network of transputers*. Institute for Continuing Education, Eindhoven University of Technology, Eindoven, The Netherlands, July 1993. Final Report.

[95] J. Koster and R.H. Bisseling. An improved algorithm for parallel sparse LU decomposition on a distributed memory multiprocessor. In J.G. Lewis, editor, *Fifth SIAM Conference on Applied Linear Algebra*, pages 397–401, 1994.

[96] J. Koster and R.H. Bisseling. Paralle sparse LU decomposition on a distributed-memory multiprocessor. *Submitted to SIAM J. Scientific Computing*, 1994.

[97] V. Kotlyar and K. Pingali. Sparse code generation for imperfectly nested loops with dependences. In *International Conference on Supercomputing*, pages 188–195, Vienna, Austria, July 1997. ACM.

[98] D.J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, New York, 1978.

[99] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–329, 1979.

[100] X. Li. *Sparse Gaussian Elimination on High Performance Computers*. PhD thesis, CS, UC Berkeley, 1996.

[101] F. Manne and H. Hafsteinsson. Efficient sparse Cholesky factorization on a massively parallel SIMD computer. *SIAM J. Scientific Computing*, 16(4):934–950, 1995.

[102] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3:255–269, 1957.

[103] U. Meier, G. Skinner, and J. Gunnels. A collection of codes for sparse matrix computations. Technical Report 1134, CSRD, Univ. of Illionis at Urbana-Champaign, 1991.

[104] John Merlin and Anthony Hey. An introduction to High Performance Fortran. *Scientific Programming*, 4:87–113, 1995.

[105] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, April 1994.

[106] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, November 1996.

[107] A. Müller and R. Rühl. Extending High-Performance Fortran for the support of unstructured computations. In *9th ACM Int'l. Conf. on Supercomputing*, pages 127–136, Barcelona, Spain, 1995.

[108] A. Navarro, Y. Paek, E.L. Zapata, and D. Padua. Compiler techniques for effective communication on distributed-memory multiprocessors. In Hank Dietz, editor, *International Conference on Parallel Processing*, pages 74–77. IEEE Computer society, August 1997.

[109] J. Ortega and C. Romine. The ijk forms of factorization II. *Parallel Computing*, 7(2):149–162, 1988.

[110] T. Ostromsky, P.C. Hansen, and Z. Zlatev. A parallel sparse QR factorization algorithm. In LNCS 1041 Springer-Verlag, editor, *2nd Int'l. Works. on Applied Parallel Computing (PARA'95)*, pages 462–472, Lyngby, Denmark, 1995.

[111] D.M. Pase, T. McDonald, and A. Meltzer. The CRAFT Fortran programming model. *Scientific Programming*, 3:227–253, 1994.

[112] S. Pissanetzky. *Sparse Matrix Technology*. Academic Press Inc., 1984.

[113] C.D. Polychronopoulos, M.B Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung, and D.A. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In *Workshop on Languages and Compilers for Parallel Computing*, pages 423–453, August 1989.

[114] R. Ponnusamy, Y.S. Hwang, R. Das, J. Saltz, A. Choudhary, and G. Fox. Supporting irregular distributions using data-parallel language. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(1):12–24, 1995.

[115] R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang, and G. Fox. Runtime support and compilation methods for user-specified data distributions. *IEEE Trans. on Parallel and Distributed Systems*, 6(8):815–831, 1995.

[116] A. Pothen and C. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math Softw.*, 16(4):303–324, 1990.

[117] A. Pothen, H.D. Simon, and K.P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11(3):430–452, 1990.

[118] A. Pothen and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM J. Scientific Computing*, 14(5):1253–1257, 1993.

[119] Bill Pottenger. *Theory, Techniques, and Experiments in Solving Recurrences in Computer Programs*. PhD thesis, Univ. of Illinois at Urbana-Champaign. CSRD., May 1997.

[120] R. Pozo. The matrix market exchange format. In *SIAM Conference on Sparse Matrices*, Coeur d'Alene, Idaho, Oct 1996.

[121] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN*. Cambridge University Press, 1992.

[122] Padma Raghavan. *CAPSS: A Cartesian Parallel Sparse Solver*. Dept. of Computer Science, Univ. of Tennessee. http://www.netlib.org/scalapack/.

[123] Lawrence Rauchwerger. *Run-Time Parallelization: A Framework for Parallel Computation.* PhD thesis, Univ. of Illinois at Urbana-Champaign. CSRD., August 1995.

[124] J.K. Reid. A note on the stability of gaussian elimination. *J. Inst. Maths. Applics.*, 8:374–375, 1971.

[125] J. Reilly. SPEC95 products and benchmarks. *SPEC Newsletter*, September 1995.

[126] L.F. Romero. *Simulación Numérica de Laseres de Semiconductores en Multiprocesadores.* PhD thesis, Dept. Arquitectura de Computadores, Univ. de Málaga, 1996.

[127] L.F. Romero and E.L. Zapata. Data distributions for sparse matrix vector multiplication. *J. Parallel Computing*, 21(4):583–605, April 1995.

[128] D.J. Rose and R.E. Tarjan. Algorithmic aspect of vertex elimination on directed graphs. *SIAM J. Appl. Math.*, 34:176–197, 1978.

[129] E. Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Scientific Computing*, 17(3):699–713, 1996.

[130] E. E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization.* PhD thesis, Computer Sicence Dept, Standford University, December 1992.

[131] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computation. Technical Report 1029, CSRD, Univ. of Illinois at Urbana Champaign, 1990.

[132] J. Saltz, R. Ponnusamy, S. Sharma, B. Moon, Y. Hwang, M. Uysal, and R. Das. *A Manual for the CHAOS Runtime Library.* Computer Science Dept., University of Maryland, 1995. Tech. Report CS-TR-3437 and UMIACS-TR-95-34.

[133] U. Schendel. *Sparse Matrices: Numerical Aspects with Applications for Scientists and Engineers.* Mathematics and its Applications. Ellis Horwood, New York, 1989.

[134] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. *ACM*, pages 26–36, 1996.

[135] Silicon Graphics. *MIPSpro Fortran77 Programmer's Guide.*

[136] Silicon Graphics, Inc., SGI, Inc., Mountain View, CA. *IRIS Power C, User's Guide*, 1996.

[137] Silicon Graphics, Inc., SGI, Inc., Mountain View, CA. *IRIS Power Fortran, User's Guide*, 1996.

[138] A. Skjellum. *Concurrent dinamic simulation: Multicomputer algorithm research applied to ordinary differential-algebraic process systems in chemical engineering.* PhD thesis, California Institute of Technology, Pasadena, CA, USA, May. 1990.

[139] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V. Klema, and C. Moler. *Matrix Eigensystem Routines—EISPACK Guide.* Springer-Verlag, New York, second ed. edition, 1976.

[140] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *10th Int'l. Conf. on Supercomputing*, pages 18–25, Philadelphia, PA, 1996.

[141] Theoretical Studies Dept. AEA Technology, 424.4 Harwell, Didcot, Oxfordshire, OX11 0RA, UK. *Harwell Subroutine Library*, release 11 edition, June 1993.

[142] Thinking Machines Corporation, TMC, Cambridge, MA. *CM Fortran Language Reference Manual*, 1994.

[143] W.F. Tinney and J.W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE*, 55:1801–1809, 1967.

[144] J. Torrellas and D. Padua. The Illinois aggressive COMA multiprocessor project (I-ACOMA). In *6th Symposium on the Frontiers of Massively Parallel Computing*, Oct. 1996.

[145] J.P. Touriño, R. Doallo, R. Asenjo, O.Plata, and E.L. Zapata. Analyzing data structures for parallel sparse direct solvers: pivoting and fill-in. In *Sixth Workshop on Compilers for Parallel Computers*, pages 151–168, Aachen, Germany, December 1996.

[146] Juan Touriño. *Parallelization and Compilation Issues for Sparse QR Algorithms.* PhD thesis, Dept. Electronics and Systems, Univ. de La Coruña, Spain., 1998.

[147] G.P. Trabado and E.L. Zapata. Exploiting locality on parallel sparse matrix computations. In *3rd EUROMICRO Works. on Parallel and Distributed Processing*, pages 2–9, San Remo, Italy 1995.

[148] M.A. Trenas. *Algoritmos Paralelos para el Cálculo de Autovalores en Matrices Dispersas.* E.T.S.I. Telecomunicación, Univ. de Málaga, October 1995. Proyecto Fin de Carrera, Directores: R. Asenjo y E.L. Zapata.

[149] M.A. Trenas, R. Asenjo, and E.L. Zapata. Eigenvalues of symmetric sparse matrices applying parallelism. In *Euroconference: Supercomputations in Nonlinear and Disordered Systems: Algorithms, Applications and Architectures*, pages 23–28, San Lorenzo de El Escorial, Madrid, Spain, Sep 1996.

[150] P. Tu and David Padua. Automatic array privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Sixth Workshop on Languages and Compilers for Parallel Computing*, volume 768, pages 500–521. Lecture Notes in Computer Science, August 12–14 1993.

[151] Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, Univ. of Illinois at Urbana-Champaign. CSRD., May 1995.

[152] M. Ujaldón, E.L. Zapata, B. Chapman, and H.P. Zima. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Trans. on Parallel and Distributed Systems*, 1997. to appear.

[153] M. Ujaldón, E.L. Zapata, S.D. Sharma, and J. Saltz. Parallelization techniques for sparse matrix applications. *Journal of Parallel and Distributed Computing*, 38(2):256–266, 1996.

[154] A. F. van der Stappen, R. H. Bisseling, and J. G. G. van de Vorst. Parallel sparse LU decomposition on a mesh network of transputers. *SIAM J. Matrix Anal. Appl.*, 14(3):853–879, July 1993.

[155] E.F. van der Velde. Experiments with multicomputer LU-decomposition. *Concurrency: Practice and Experience*, 2:1–26, 1990.

[156] A.C.N. van Duin, P.C. Hansen, T. Ostromsky, H.A.G. Wijshoff, and Z. Zlatev. Improving the numerical stability and the performance of a parallel sparse solver. *Computers Math. Applic.*, 30:81–96, 1995.

[157] J.H. Wilkinson. Error analysis of direct methods of matrix inversion. *J. ACM*, 8:281–330, 1961.

[158] D.S. Wise and J. Franco. Costs of quadtree representation of nondense matrices. *Journal of Parallel and Distrbuted Computing*, 9:282–296, 1990.

[159] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[160] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – A Language Specification. Technical Report ACPC–TR92–4, Austrian Center for Parallel Computation, University of Vienna, Austria, 1992.

[161] Z. Zlatev. *Computational Methods for General Sparse Matrices, Mathematics and Its Applications.* 65. Kluwer Academin Publisher, Dordrecht, the Netherlands, 1991.

[162] Z. Zlatev, J. Waśniewski, P.C. Hansen, and T. Ostromsky. PARASPAR: a package for the solution of large linear algebraic equations on parallel computers with shared memory. Technical Report 95-10, Tech. Univ. Denmark, Lyngby, 1995.

[163] Z. Zlatev, J. Waśniewski, and K. Schaumburg. Y12M—Solution of Large and Sparse Systems of Linear Algebraic Equations. In *Lecture Notes in Computer Science 121*, pages 61–77, Berlin, 1981. Springer-Verlang.