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 problemas 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 tridiagonales, 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:

$$\left\{ \begin{array}{l} \text{Iterativos} \\ \text{Directos} \left\{ \begin{array}{l} \text{Frontales} \\ \text{Multifrontales} \\ \text{Supernodo} \\ \text{Genéricos} \left\{ \begin{array}{l} \text{Crout} \\ \textit{Left-looking} \\ \textit{Right-looking} \end{array} \right. \end{array} \right. \end{array} \right.$$

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

```
for k=1 to n do
   factoriza A_kk = L_kk U_kk
   for i=k+1 to n do
      L_ik ← A_ik U_kk^-1
   for j=k+1 to n do
      U_kj ← L_kk^-1 A_kj
      for i=k+1 to n do
         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 ($x^TAx \geq 0, \quad \forall x \neq 0$) 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*).

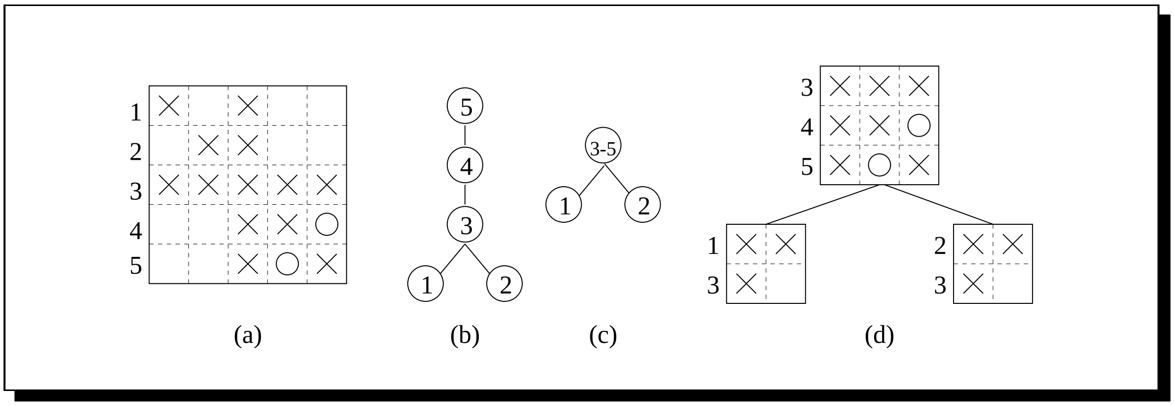


Figura 1.1: Factorización multifrontal.

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:

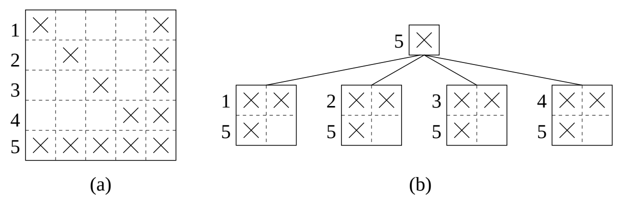


Figura 1.2: Reordenación previa y sus ventajas.

- 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: $x^T A x \geq 0$, $\forall\, x \neq 0$.
- 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.

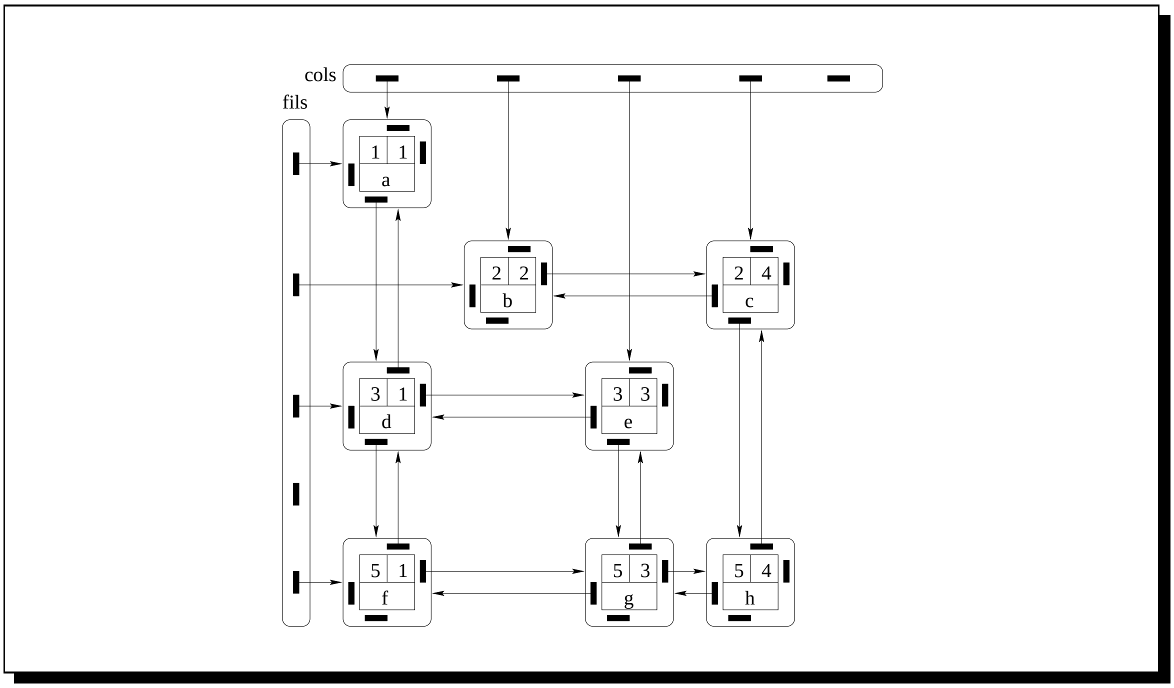


Figura 1.3: Estructura de datos de lista bidimensional doblemente enlazada.

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	C
TYPE entrada	**struct** *entrada* {
INTEGER:: Fil	**int** *Fil*;
REAL:: Val	**double** *Val*;
TYPE (entrada), **POINTER**:: previ, nexti	**struct** *entrada* $*previ, *nexti$; };
END TYPE entrada	**struct** *matriz* {
TYPE (ptr), **DIMENSION(n)**:: cols	**struct** *entrada* $*cols[n]$; };

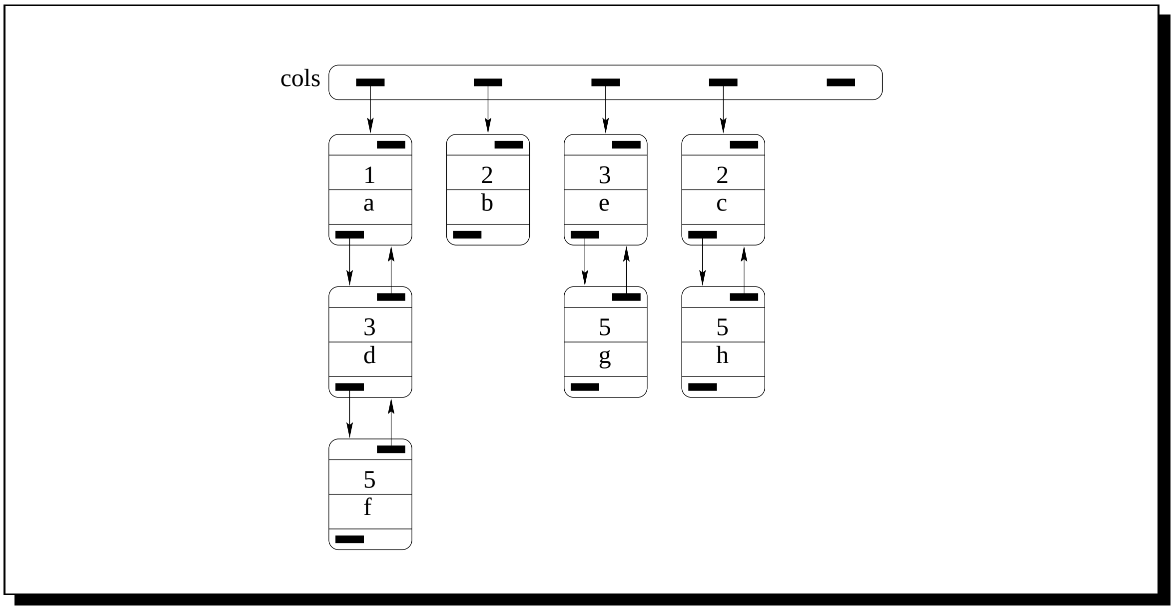


Figura 1.4: Lista unidimensional doblemente enlazada por columnas.

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	C
TYPE cvs	**struct** *cvs* {
INTEGER:: size	**int** *size*;
INTEGER, DIMENSION(:), POINTER:: Ind	**int** $*Ind$;
REAL, DIMENSION(:), POINTER:: Val	**real** $*Val$;
END TYPE pv	}

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.

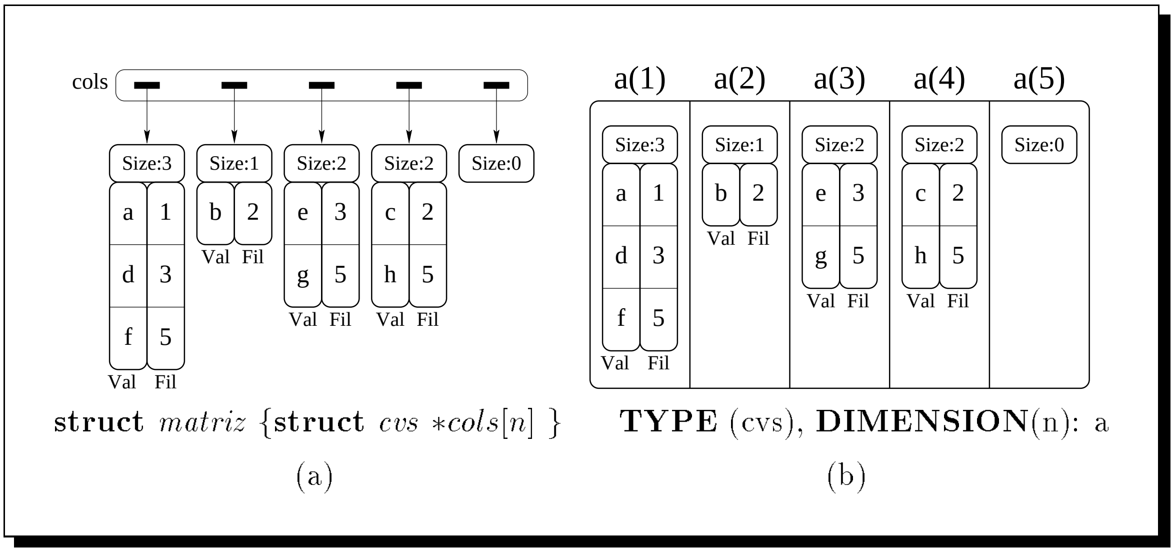


Figura 1.5: Matriz dispersa como conjunto de columnas CVS.

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.

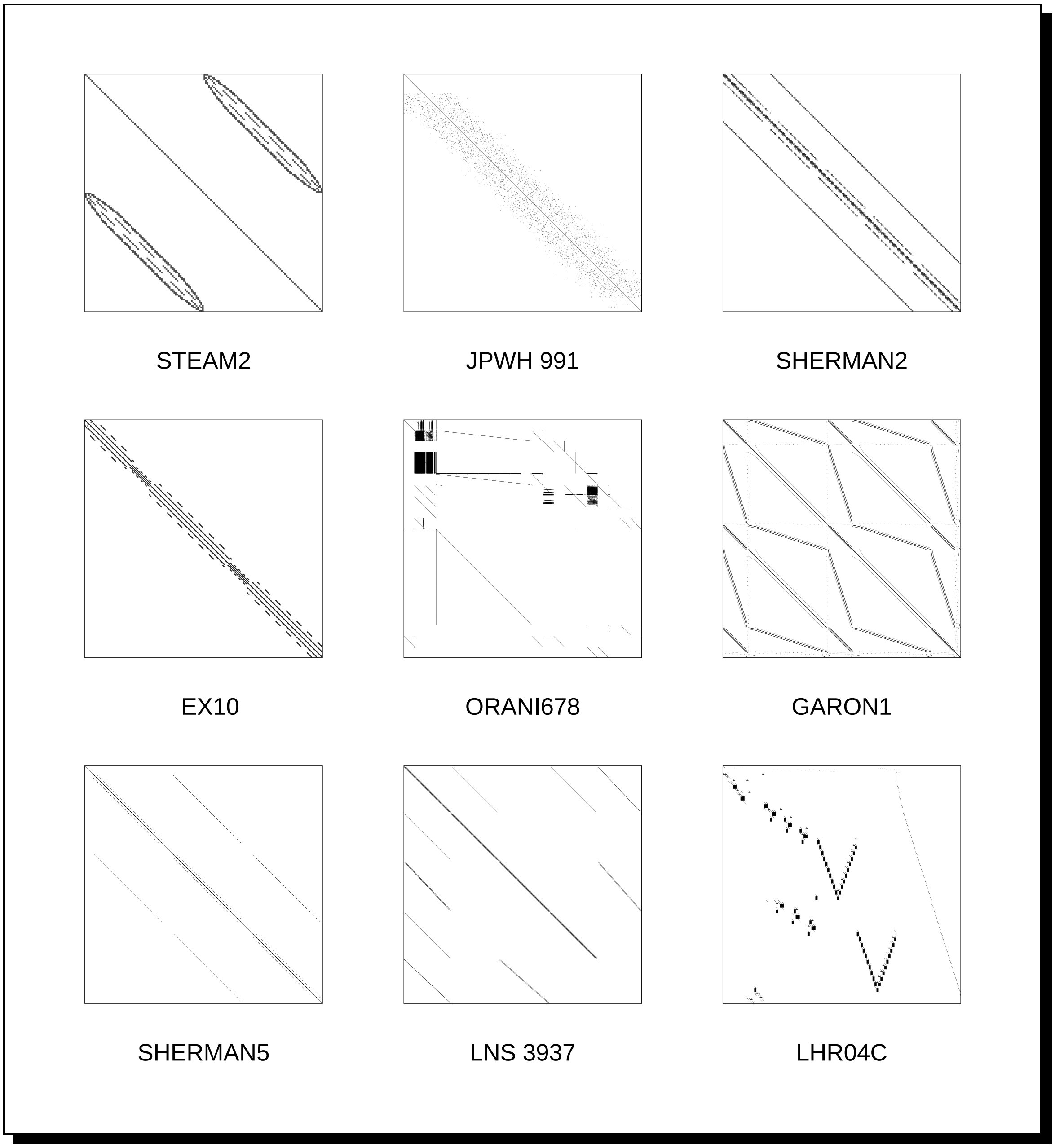


Figura 1.6: Patrones de algunas matrices de prueba.

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.

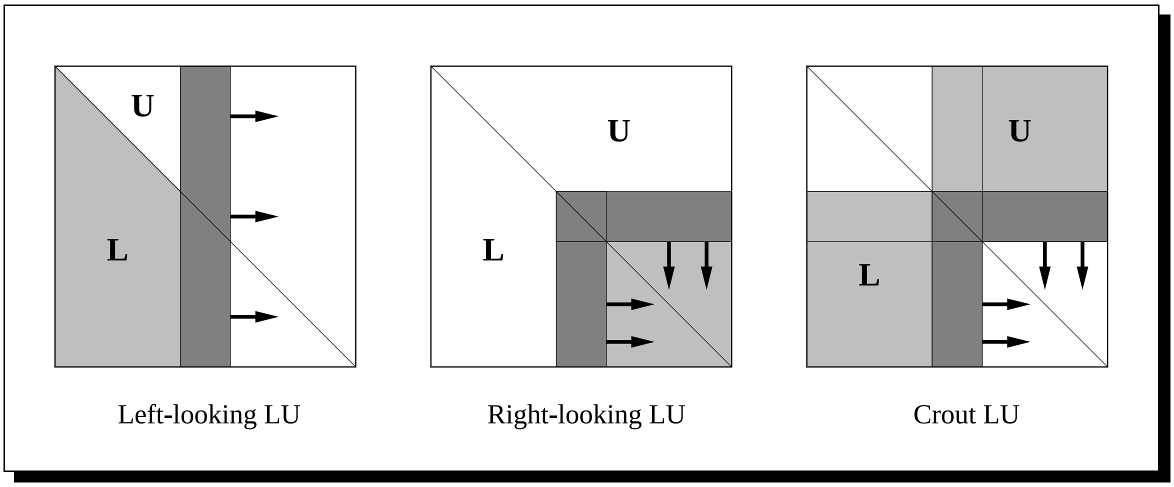


Figura 1.7: Patrones de acceso a datos para las variantes de la factorización LU.

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

```
for k=1 to n do
  for i=k+1 to n do
    A_ik ← A_ik / A_kk
  for j=k+1 to n do
    for i=k+1 to n do
      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

```
for k=1 to n do
   for j=k+1 to n do
      A_kj ← A_kj / A_kk
   for i=k+1 to n do
      for j=k+1 to n do
         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

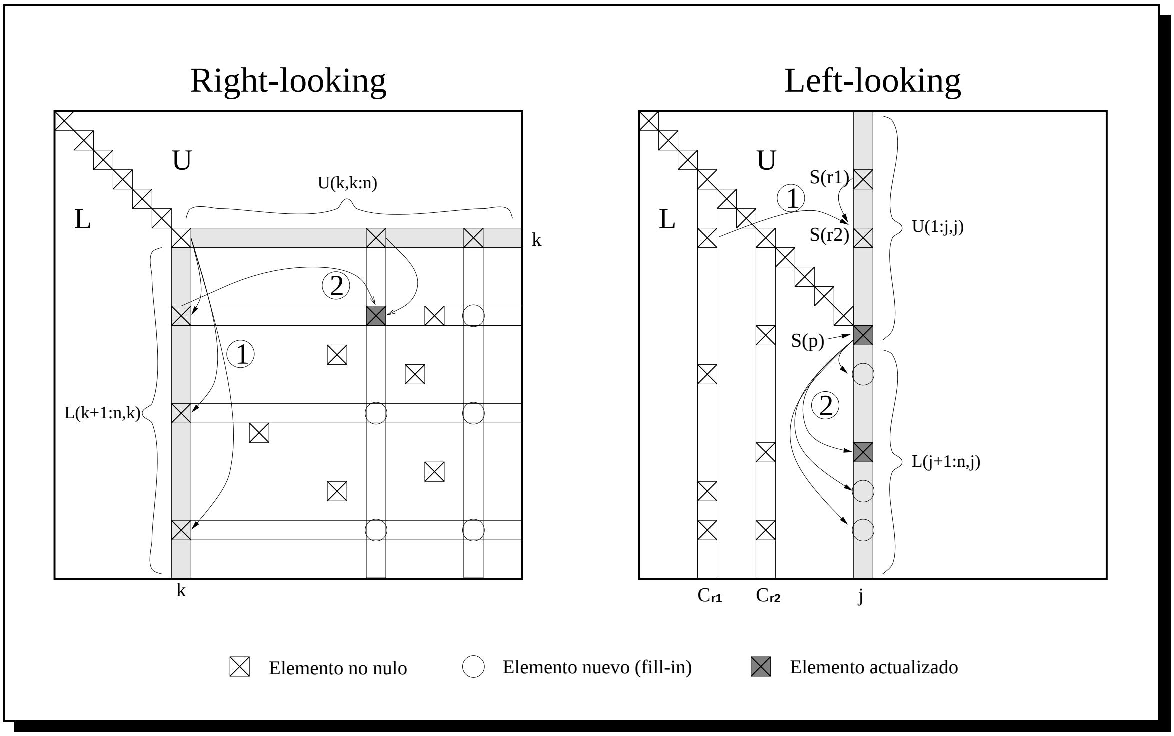


Figura 1.8: Flujo de acceso a datos para las alternativas *right* y *left-looking* en el caso disperso.

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 $O(c^2n)$, 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 factorización 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

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

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

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

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, ˜, 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].

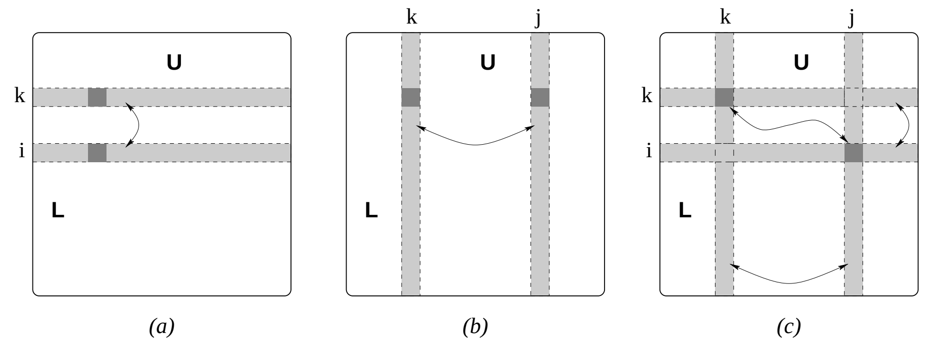


Figura 1.9: Flujo de acceso en operaciones de pivoteo: (a) parcial por filas; (b) parcial por columnas; y (c) total.

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.

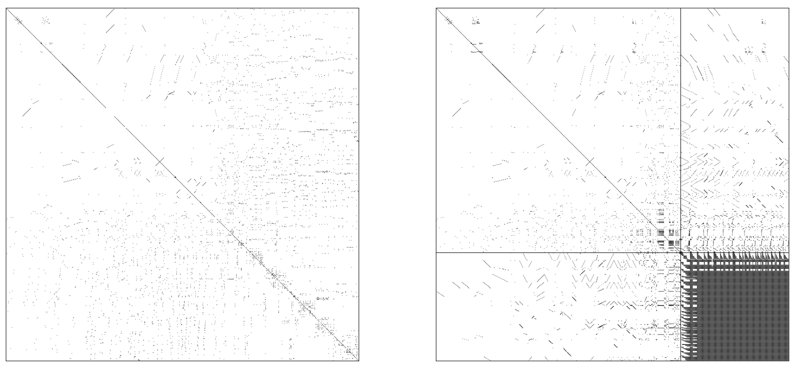
En el caso de resolver matrices no simétricas donde tiene especial relevancia considerar los valores numéricos de los coeficientes al elegir los pivots, es menos ventajoso separar las dos etapas mencionadas, sobre todo en códigos *right-looking*. Aquí, el análisis y la factorización se ejecutan de forma 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.



(a) (b)

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%.