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.

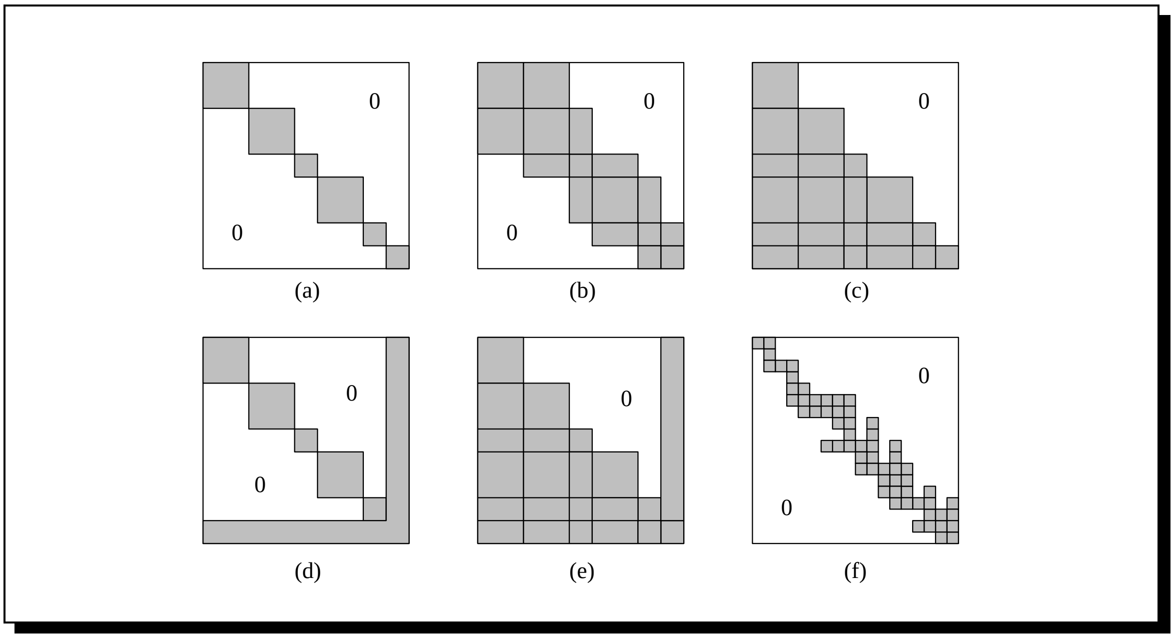

Figura 3.1: Patrones que podemos conseguir para una matriz dispersa A.

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.

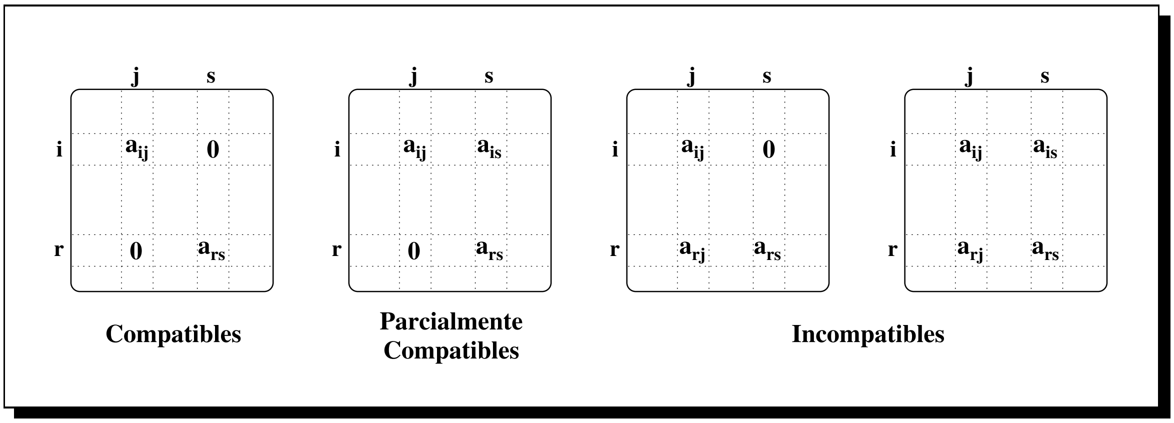

Figura 3.2: Compatibilidad entre coeficientes de la matriz.

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

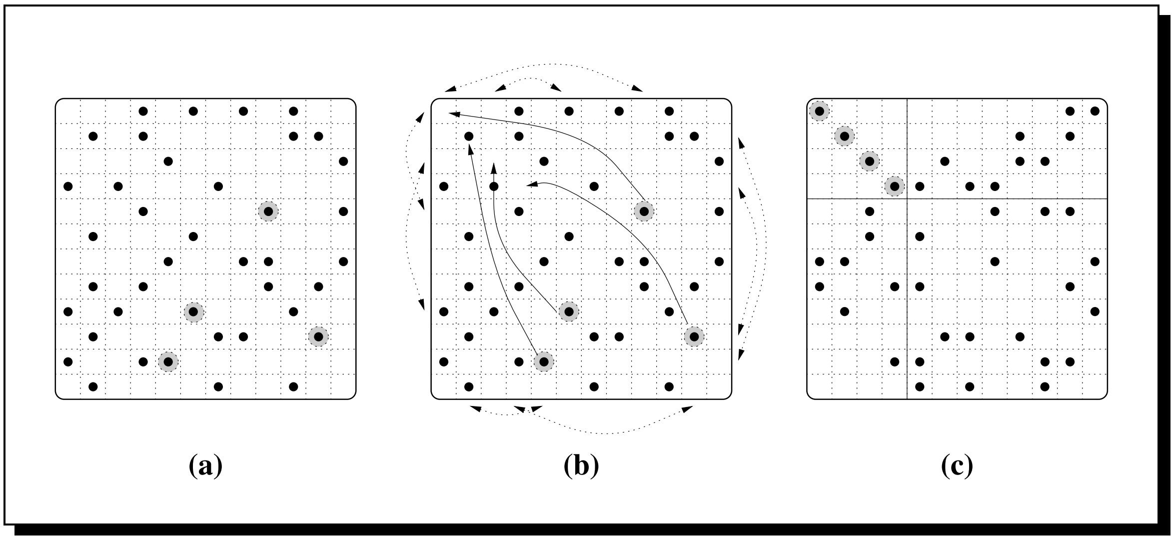

Figura 3.3: Fases del algoritmo de pivots paralelos: (a) Elección de los pivots compatibles; (b) Permutaciones de filas y columnas para llevar los pivots a la diagonal; (c) Actualización de rango `m`.

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 $k \leq j < k+\mathtt{m}$, 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 procesadores 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.

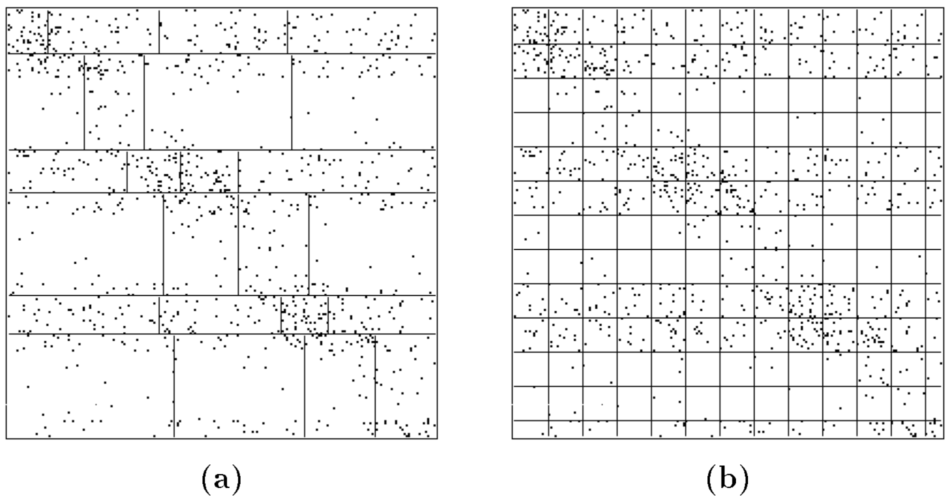

Figura 3.4: Distribuciones dispersas: (a) MRD y (b) Scatter.

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

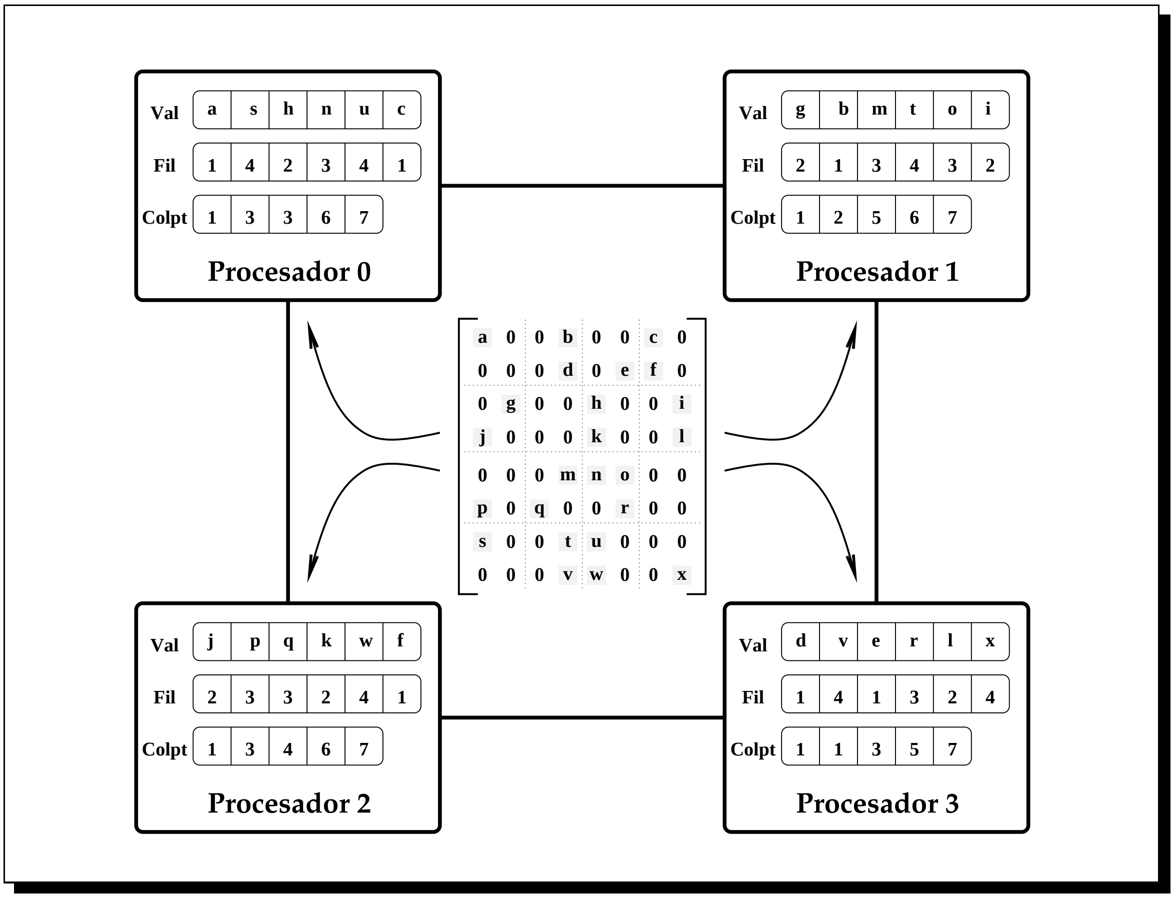

Figura 3.5: Esquema de distribución BCS para una malla de 2×2 procesadores.

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

$\pi = \gamma =$ identidad;
Inicializa R y C; Calcula `dens`
$k = 0$;
while ($(k < n)$ && (`dens<maxdens`))
  {
    Busca el conjunto de pivots: `PivotSet=` $(i_r, j_r)\ :\ 0 \leq r < m$;
    Permuta filas, π_i y R_i;
    Permuta columnas, γ_j y C_j;
    Actualiza $A^{(k)}$;
    Actualiza los vectores R y C;
    $k = k + m$;
  }
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:

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

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.

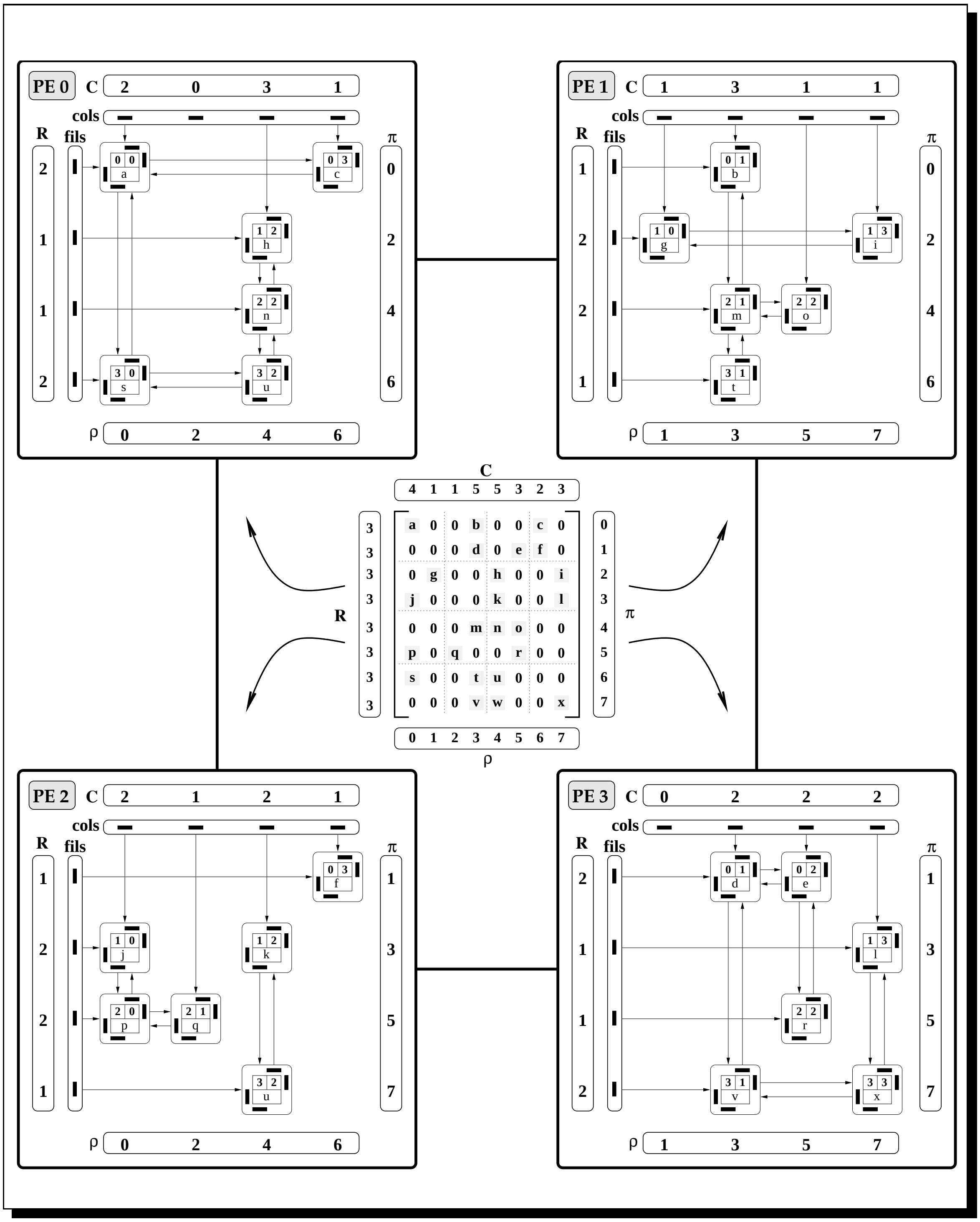

Figura 3.6: Esquema de distribución LLRCS-Scatter.

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.

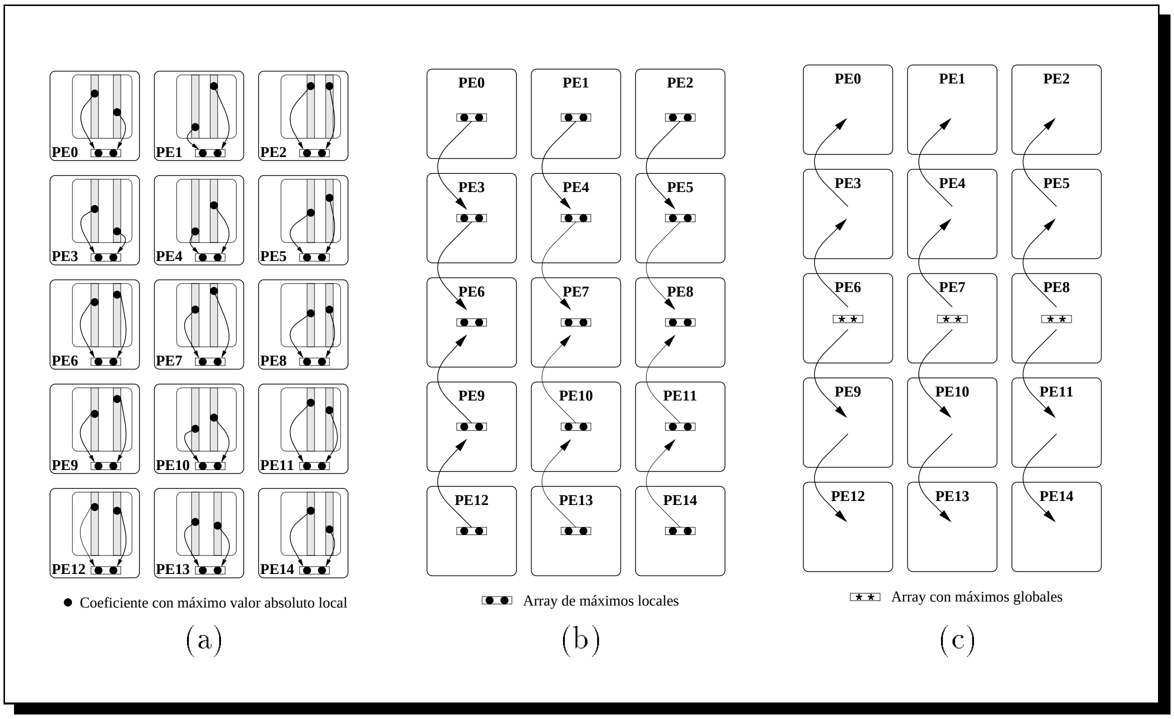

Figura 3.7: (a) Selección de máximo local. (b) Reducción global. (c) Radiación.

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).

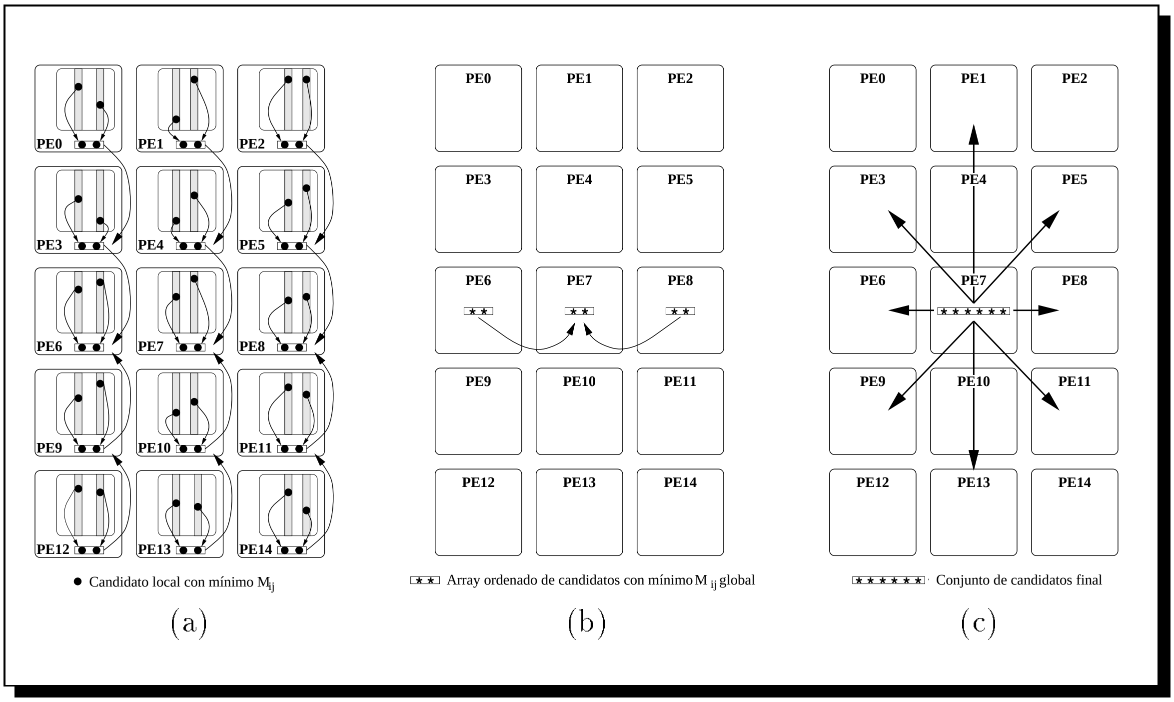

Figura 3.8: (a) Reducción de los candidatos a la fila centra. (b) Colección y ordenación de los candidatos hacia el procesador central. (c) Radiación del conjunto `Candidatos`.

Como podemos ver, si $u \rightarrow 0$, primaremos la estabilidad numérica, mientras que si $u \rightarrow 1$, 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`.

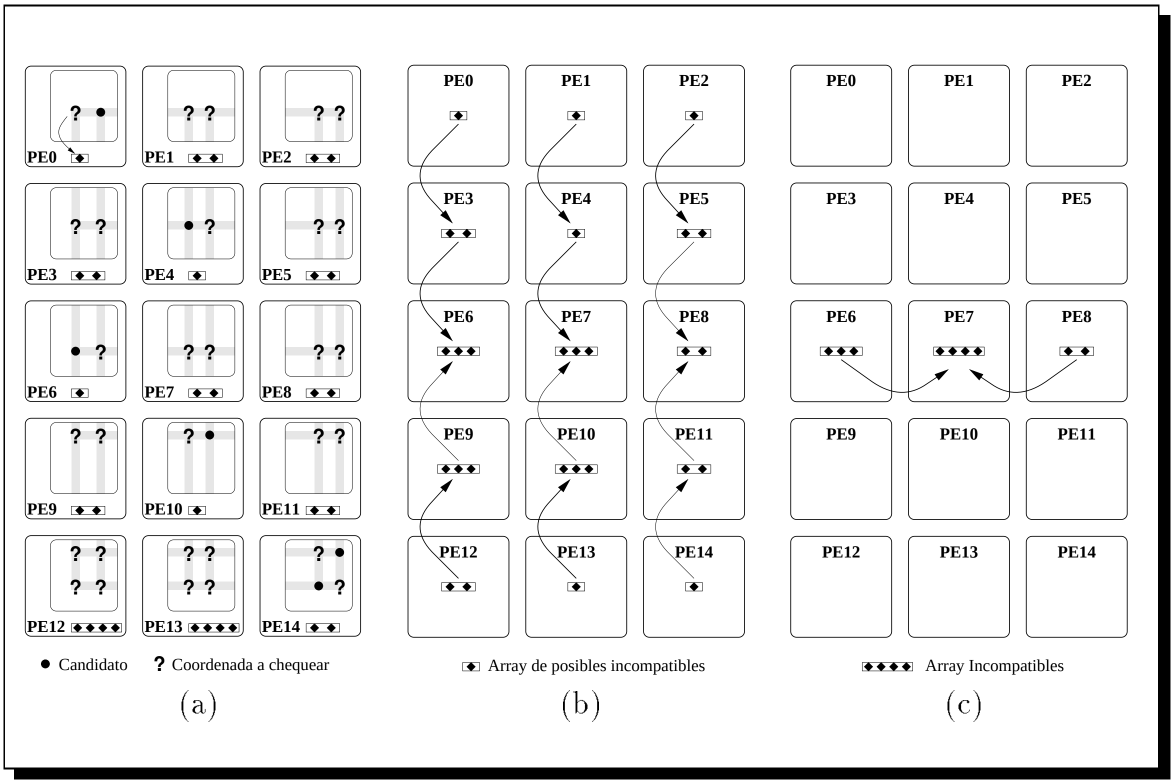

Figura 3.9: Búsqueda paralela de incompatibilidades (a). Colección de incompatibilidades por columnas (b) y en la fila central (c).

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:

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

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

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

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 $A_{jj}^{(k)} : (k \le j < k+m) \wedge (j \bmod P = p) \wedge (j \bmod Q = q)$ 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 $A_{lj}^{(k)} : (k \le l < k+m) \wedge (k+m \le j < n) \wedge (A_{lj}^{(k)} \neq 0)$, 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 $A_{il}^{(k)} : (k+m \le i < n) \wedge (k \le l < k+m) \wedge (A_{il}^{(k)} \neq 0)$, 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 $m \ge P$ $(m \ge Q)$, 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 $m \ge P$, 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 $(k \le l < k+m)$. 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

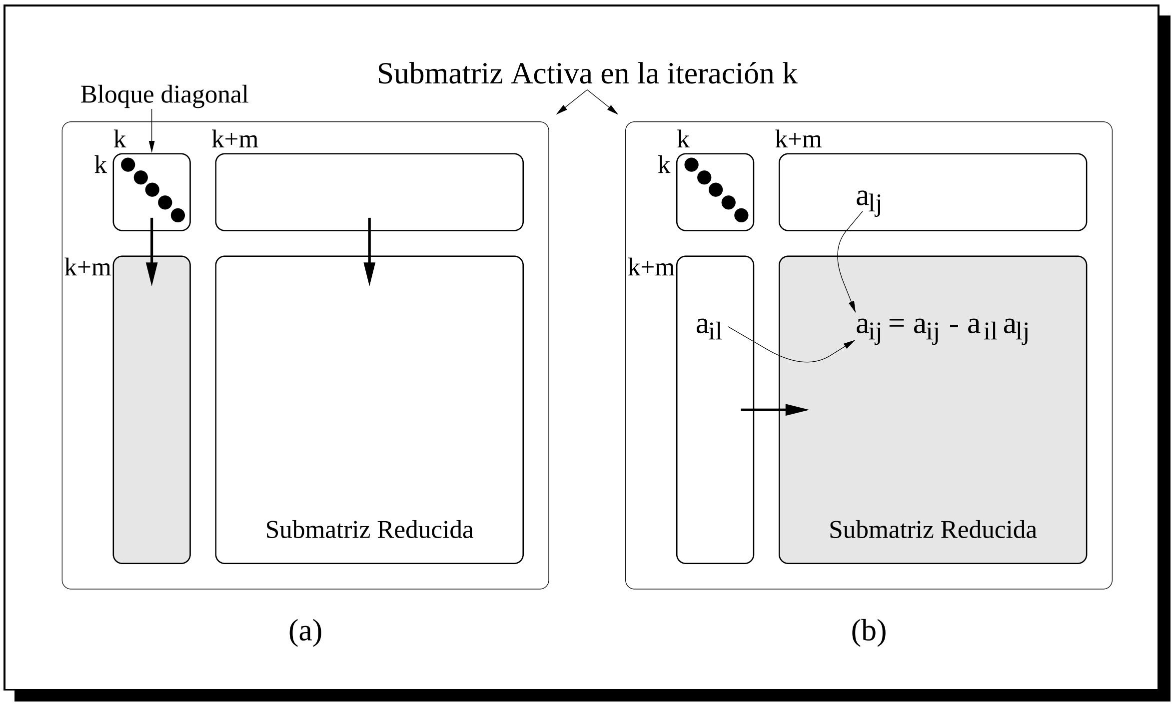

Figura 3.10: Pasos en la actualización. (a) Primero: Radiación por columnas y normalización del bloque de columnas (en gris). (b) Segundo: Radiación por filas y actualización de la submatriz reducida (en gris).

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 $PR_{\hat{\imath}}^{(k)}$ como $PC_{\hat{\jmath}}^{(k)}$. 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 $(i \bmod P, *)$. Simultáneamente, la fila de procesadores $(i \bmod P, *)$ 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

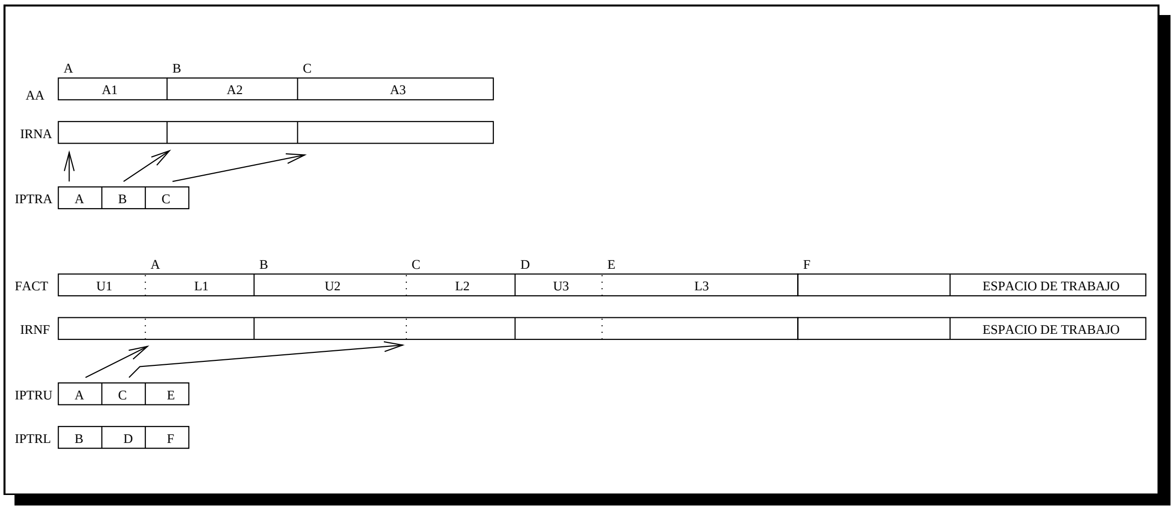

Figura 3.11: Estructura de datos utilizada en las rutinas MA50.

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 Gradiente 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

$\hat{w}(1:\hat{m}) = 0$
DO j=1,$\hat{n}$! Para cada columna
  IF (Soy propietario del pivot l_{jj}) **THEN**
    WHILE(lleguen_mensajes_horizontales $\wedge$ falta_pw(j))
      recv($pw(i)$, $i \geq j$) ! Recepción Horizontal
      $y_i = b_i - pw(i)$
    END WHILE
    broadcast(y_j) ! Radiación Vertical
  ELSE
    recv(y_j) ! Recepción Vertical
  END IF
  $\hat{w}(i) = \hat{w}(i) + \hat{l}_{ij} \cdot y_j, \quad \hat{l}_{ij} \neq 0 \ \wedge \ i > j.$
  IF($\nexists \ \hat{l}_{ij'} \neq 0, \ j' > j \wedge i > j$) **THEN**
    send($\hat{w}(i)$) ! Env*ío* Horizontal
  END IF
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.

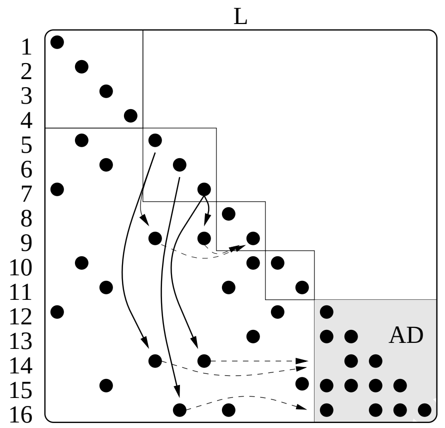

Figura 3.12: Flujo de datos en la substitución hacia adelante.

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 características 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

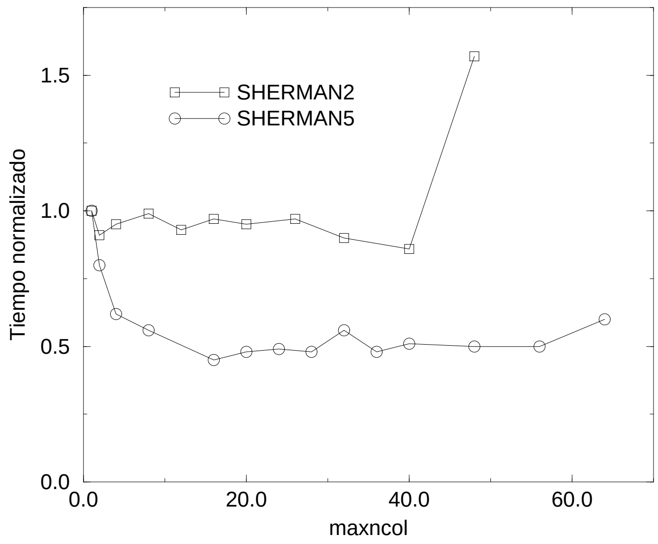

Figura 3.13: Influencia de la variable `ncol` en el tiempo de ejecución secuencial.

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

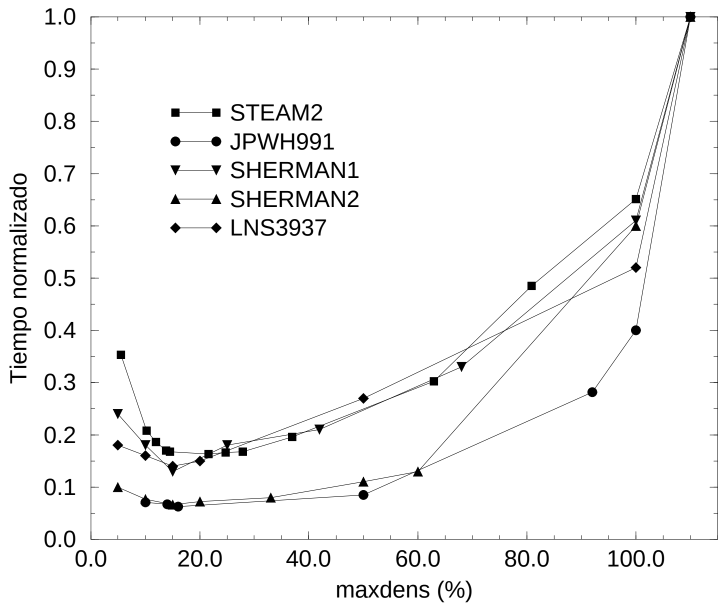

Figura 3.14: Influencia del parámetro `maxdens` en el tiempo de ejecución.

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.

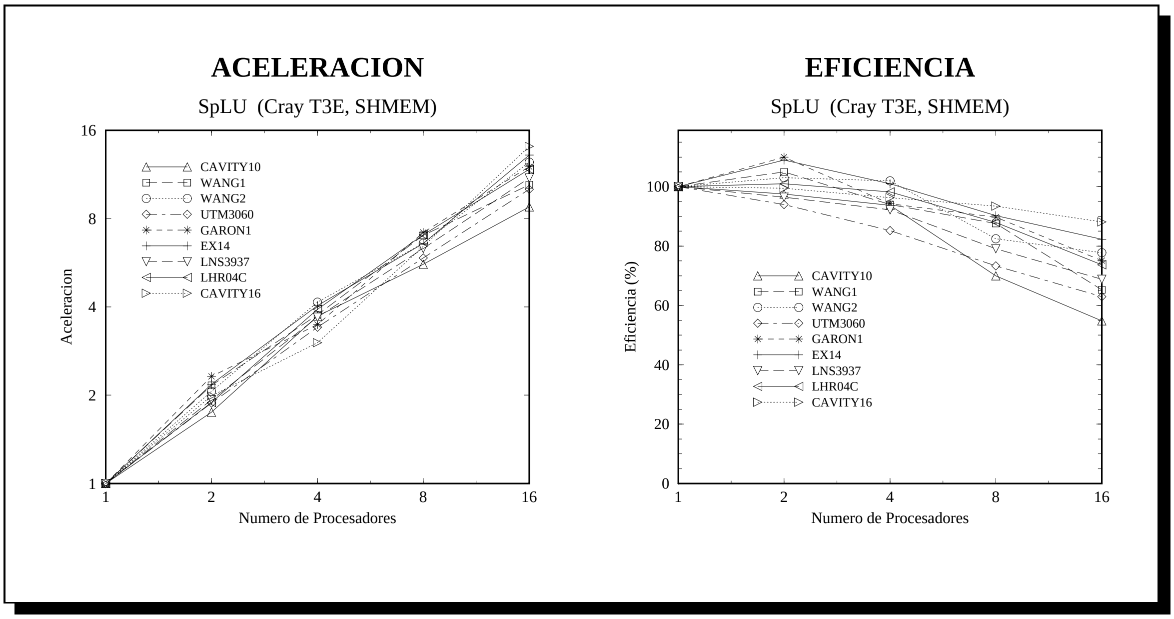

Figura 3.15: Aceleración y eficiencia del código SpLU en función del número de procesadores.

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.