UNIVERSIDAD DE SANTIAGO DE COMPOSTELA

FACULTAD DE FÍSICA

Departamento de Electrónica y Computación

FACTORIZACIÓN DE CHOLESKY MODIFICADA DE MATRICES DISPERSAS SOBRE MULTIPROCESADORES

María J. Martín Santamaría

Julio, 1999

Índice General

Índice de Tablas

Índice de Figuras

Prefacio

El objetivo de esta tesis es establecer estrategias de programación para la obtención de altos rendimientos en el procesamiento de códigos irregulares sobre arquitecturas multiprocesador. Este tipo de códigos aparecen en numerosas áreas científicas y de ingeniería. La complejidad que supone su paralelización hace que los reestructuradores de código automáticos actuales fracasen en el intento de encontrar soluciones eficientes, por lo que el esfuerzo que supone su paralelización manual está, sin duda, justificado. Dentro de los problemas irregulares hemos elegido como caso representativo la resolución de sistemas de ecuaciones dispersos y, más concretamente, nos hemos centrado en la parte computacionalmente más costosa del problema, la factorización de la matriz. Hemos elegido el algoritmo de Cholesky modificado para llevar a cabo dicha factorización, el cual no es más que una generalización del algoritmo de Cholesky estándar adecuado para resolver problemas de optimización. Se presentan técnicas de paralelización de dicho algoritmo, tanto sobre sistemas de memoria distribuida como de memoria compartida con tiempo de acceso a memoria no uniforme, centrándonos en los aspectos que son más decisivos para conseguir buenos rendimientos en las diferentes situaciones estudiadas. Esto es, estudio de las comunicaciones y del balanceo de la carga para el caso de memoria distribuida, y estudio de la localidad de los datos en el caso de sistemas de memoria compartida tipo NUMA.

Aunque todo el estudio se realiza sobre el algoritmo de Cholesky modificado, las técnicas de paralelización presentadas pueden ser generalizadas a otros problemas de factorización de matrices dispersas o a la resolución de sistemas triangulares. En general, serán aplicables a todos aquellos códigos cuyas dependencias puedan ser representadas por un árbol.

El trabajo se divide en 5 capítulos. El Capítulo 1 es una breve descripción del contexto de la tesis, motivación y objetivos, incluyendo las tendencias arquitectónicas actuales.

En el Capítulo 2 se introduce el algoritmo de Cholesky modificado para matrices dispersas. En este capítulo se discuten los aspectos más importante sobre matrices dispersas y se describen las diferentes orientaciones del algoritmo haciendo especial hincapié en sus grafos de dependencias.

Antes de presentar las estrategias de paralelización propuestas, en el Capítulo 3 se describen las máquinas paralelas con las que hemos trabajado, así como las herramientas software utilizadas. Se incluye una sección dedicada a la paralelización automática de código.

En el Capítulo 4 se proponen técnicas de paralelización para la implementación eficiente del algoritmo sobre sistemas de memoria distribuida. Se inicia el estudio partiendo de un modelo de paralelización de grano fino, aplicando una distribución bidimensional de los datos y centrándonos, fundamentalmente, en el estudio de las comunicaciones y el balanceo de la carga. Se llega a la conclusión de que se obtiene un mayor rendimiento si cada procesador almacena columnas enteras de la matriz, es decir, si aplicamos un modelo de programación paralelo de grano medio de acuerdo con una distribución unidimensional de la matriz. Adaptamos, por tanto, los algoritmos con distribuciones unidimensionales de la matriz utilizados para la factorización de Cholesky estándar a la factorización de Cholesky modificada, y proponemos la aplicación de estrategias para reducir los tiempos de espera en la ejecución de los códigos paralelos. Se muestran resultados experimentales sobre el Fujitsu AP1000 utilizando diferentes librerías de pase de mensajes y sobre el Cray T3E utilizando MPI.

En el Capítulo 5 se proponen y discuten técnicas de paralelización para la implementación eficiente del algoritmo sobre un sistema de memoria compartido tipo NUMA. Utilizamos la librería *parmacs* para realizar las construcciones paralelas. Nos centramos en el estudio de la localidad de los datos, proponiendo nuevas distribuciones basadas en un modelo de programación paralelo de grano grueso que aumenten dicha localidad y que consigan, por tanto, un mayor rendimiento. Se muestran resultados experimentales sobre el sistema 02000 de SGI.

Finalizamos con las conclusiones que se derivan de nuestro trabajo, sintetizando las principales aportaciones de esta tesis.

Los resultados de este trabajo de investigación han sido publicados en [71, 72, 73, 74,75,76,77, 81, 82].

Agradecimientos

En primer lugar me gustaría agradecer a mi director de tesis, Francisco Fernández Rivera, por su trabajo de dirección y ayuda. Su rigurosa supervisión y su constante apoyo han sido fundamentales para la realización de esta tesis.

Al Departamento de Electrónica y Computación de la Univ. de Santiago de Compostela y Departamento de Electrónica y Sistemas de la Univ. de A Coruña, por poner a mi disposición todo el material necesario para el óptimo desarrollo de esta tesis.

A los miembros del EPCC *(Edinburgh Parallel Computing Center,* University of Edinburgh, UK) y SUIF Group *(Department of Computer Science,* Stanford University, USA) por su asistencia durante mis visitas de investigación.

Agradezco también a los siguientes centros el haberme facilitado el acceso a sus plataformas: Fujitsu High Performance Computing Research Centre, FPCRF, Japan (Fujitsu AP1000); Imperial College/Fujitsu Parallel Computing Research Centre, IFPC, UK (Fujitsu AP1000); Centro de investigaciones energéticas, medioambientales y tecnológicas, CIEMAT, Madrid (Cray T3E); Universidad de Málaga (SGI 02000).

Al grupo de Sistemas Operativos y Microkernels del Departament d'Arquitectura de Computadors de la Univ. Politècnica de Catalunya por facilitarme las macros *parmacs* apropiadas para el 02000 de SGI.

A la Xunta de Galicia y el Ministerio de Educación y Ciencia por el soporte económico a través de la concesión de una beca predoctoral y una beca de FPU respectivamente y los proyectos TIC92-0942-C03-03, XUGA20606B93, TIC95-1754-E, TIC96-1125-C03-02, XUGA20605B96.

A Inma y David por sus sugerencias y ayuda en ciertas partes de este trabajo.

Finalmente, un millón de gracias a mi familia y amigos, sin todos ellos este trabajo no hubiese sido posible.

Capítulo 1

Introducción

1.1 Algoritmos irregulares

Del conjunto de aplicaciones que presentan una gran relevancia práctica y que demandan un elevado coste computacional, se han encontrado soluciones paralelas eficientes a un gran número de problemas. Sin embargo, todavía existe una amplia clase de aplicaciones, denominados problemas irregulares, que carecen de soluciones paralelas eficientes.

El esfuerzo que supone la ejecución eficiente de códigos irregulares está sin duda justificada: se estima que al menos el 50% de todos los programas científicos son irregulares y que más del 50% de todos los ciclos de reloj consumidos en supercomputadores son consumidos por este tipo de códigos.

Existen algunos grupos de reconocido prestigio trabajando en el campo del desarrollo de paralelizadores automáticos. Mediante sus compiladores (Polaris [15], PCA/PFA [98,99], SUIF [50], etc), son capaces de paralelizar la mayoría de los códigos regulares; sin embargo las prestaciones que ofrecen en la paralelización de problemas irregulares, como las que aparecen al procesar matrices dispersas, no se pueden comparar con las ofrecidas por un programador experto. Se hace por tanto necesario el estudio detallado de cada problema concreto y su paralelización manual.

1.2 Algoritmo de Cholesky modificado disperso: motivación y objetivos

El tema general de esta tesis es la paralelización de algoritmos para matrices dispersas, es decir, paralelización de códigos irregulares. Dentro de los problemas irregulares hemos elegido como caso representativo la resolución de sistemas de ecuaciones dispersos. La resolución de grandes sistemas lineales dispersos es un problema que aparece en una gran variedad de aplicaciones, incluyendo programación lineal, aplicaciones basadas en elementos finitos y diferencias finitas, simulación de procesos y problemas de optimización en general, entre otros. Dentro del problema de resolución, la factorización de la matriz es frecuentemente el cuello de botella computacional. Consecuentemente, existe un gran interés en llevar a cabo la computación de estos problemas sobre máquinas paralelas. Nosotros nos hemos centrado en la factorización de la matriz utilizando el algoritmo de Cholesky modificado. El algoritmo de Cholesky modificado es una generalización del algoritmo de Cholesky estándar adecuado para la resolución de problemas de optimización no lineal.

En este trabajo presentamos diferentes técnicas para aumentar el rendimiento de la factorización de Cholesky modificada para matrices dispersas sobre sistemas de memoria distribuida y memoria compartida físicamente distribuida que corresponden a los tipos de sistemas que actualmente ofrecen mayores prestaciones [110]. En el primer caso, el balanceo de la carga y el número y volumen de comunicaciones es un factor determinante en el rendimiento del programa, y será precisamente en estos dos aspectos en los que centraremos nuestro estudio. En el segundo caso, la localidad de los datos es potencialmente la característica más importante para conseguir buenos rendimientos. El árbol de eliminación de la matriz nos proporcionará la información precisa acerca del comportamiento del código para incrementar la localidad en el acceso a los datos y aumentar con ello los rendimientos.

La motivación de esta investigación es explotar el paralelismo existente en el algoritmo para resolver los problemas con requerimientos de elevado coste computacional en la práctica. Hace 20 años los sistemas lineales que los usuarios querían resolver tenían aproximadamente decenas o centenares de incógnitas (ver Tabla 1.6.1 en Duff et al. [21]). Hoy es común encontrar sistemas que implican más de 50000 incógnitas, por ejemplo, los problemas que surgen de las simulaciones en procesos en 3D. La resolución de estos sistemas es abordable gracias al aumento en la velocidad de los procesadores y el abaratamiento de la memoria principal. Sin embargo, son necesarios algoritmos eficientes que saquen partido a las características de las nuevas arquitecturas.

Todo nuestro estudio lo realizamos sobre el algoritmo de Cholesky modificado, aunque las ideas presentadas pueden ser generalizadas a otros problemas de factorización de matrices dispersas, o a la resolución de sistemas triangulares. En general

serán aplicables a todos aquellos códigos cuyas dependencias puedan ser representadas por un árbol.

1.3 Tendencias en arquitecturas de computadores

La creciente demanda de memoria y CPU por las aplicaciones computacionales ha originado el desarrollo de arquitecturas cada día más rápidas y con mejores prestaciones. En esta evolución de los computadores, las arquitecturas paralelas son un paso inevitable en la consecución de mayores velocidades de procesamiento. La computación paralela ha llegado a ser un componente crítico de la tecnología de computación de los años 90, y probablemente tendrá tanto impacto en los próximos 20 años como los microprocesadores tuvieron en los últimos 20.

Los avances tecnológicos en el diseño de microprocesadores han conducido a la explotación de paralelismo a nivel de instrucción, lo cual reduce significativamente el número de ciclos de reloj necesarios para ejecutar una instrucción, o incluso a la ejecución de instrucciones en paralelo usando unidades funcionales independientes. Ejemplos de este tipo de paralelismo son las arquitecturas superescalares y VLIW *(Very Long Instruction Word)* donde existen gran cantidad de unidades funcionales independientes. El rendimiento de esta clase de arquitecturas es fuertemente dependiente de la habilidad del compilador para reestructurar el código de forma que se maximice el número de unidades funcionales trabajando en paralelo.

Sin embargo, el crecimiento de las prestaciones de un procesador debido al paralelismo a nivel de instrucción es limitado; por ejemplo, actualmente no es rentable el coste asociado a una arquitectura que ejecuta 8 o más instrucciones en paralelo. Además el rendimiento real está muy lejos del teórico debido a las características de los códigos y al uso de compiladores que no son capaces de explotar todo su potencial *hardware.* Esto justifica la relevancia de los multiprocesadores, conjunto de procesadores que trabajan coordinadamente en la solución de problemas computacionales complejos. Información detallada y actualizada sobre multiprocesadores puede ser encontrada en una gran cantidad de referencias, de las que señalamos [19,54, 101].

La mayor parte de los computadores paralelos actuales se catalogan dentro de la clasificación tradicional de Flynn [26] como máquinas MIMD *(Multiple Intruction Multiple Data)* [110]. Los distintos procesadores que las componen pueden ejecutar simultáneamente segmentos de código distintos y operar con datos diferentes con un elevado grado de desacoplo. Dentro de esta clase se puede hacer una división en dos subgrupos atendiendo al modo de gestión de la memoria del sistema:

- Memoria compartida: Denominados por algunos autores multiprocesadores

propiamente dichos. En ellos el área de memoria global es direccionable directamente por todos los procesadores del sistema. Los procesadores se comunican mediante variables compartidas de memoria, con cargas y almacenamientos capaces de acceder a cualquier posición de memoria común. La programación de este tipo de máquinas es relativamente sencilla, pero en contrapartida son menos escalables ya que la memoria global sólo permite la conexión de un número reducido de procesadores. Ejemplos de este tipo de arquitectura son el SGI Power Challenge, Cray Y-MP, Cray C-90, etc.

- Memoria distribuida: Denominados multicomputadores por algunos autores, en ellos cada procesador tiene una memoria asociada que no es accesible por el resto de los procesadores del sistema y que se denomina memoria local. El modelo de comunicación entre procesadores es por pase de mensajes, el cual se ha de establecer de forma explícita. Este tipo de máquinas son más difíciles de programar, pero en contrapartida presentan una alta escalabilidad. Ejemplos de este tipo de arquitecturas son el IBM SP2, CM-5 *(Connection Machine)*, Fujitsu AP1000/3000, Intel Paragon, etc.

Actualmente se está produciendo un movimiento de convergencia de estas arquitecturas que intenta extraer las ventajas de ambos modelos (escalabilidad y facilidad de programación), dando lugar a arquitecturas de memoria compartida-distribuida. En estas arquitecturas la memoria se encuentra físicamente distribuida entre los procesadores pero incluyen un soporte hardware para permitir un espacio de direcciones globales, e incluso gestión de la coherencia entre caches locales a cada nodo. Es decir, son máquinas de memoria distribuida que pueden ser programadas virtualmente como máquinas de memoria compartida de una forma transparente al usuario y, consecuentemente, combinan las ventajas de ambas aproximaciones. Los supercomputadores de memoria compartida-distribuida pueden ser clasificados en dos grupos:

- NUMA (Non- *Uniform Memory Access):* Máquinas con un espacio de direcciones físico estático. Dependiendo del mecanismo de coherencia cache utilizado podemos dividir este grupo en varios subgrupos:
  - Sin coherencia cache: Por ejemplo el Cray T3D
  - Con coherencia cache parcial: Cray T3E
  - Con coherencia cache (CC-NUMA, *Cache Coherent* NUMA): SGI Origin 2000
- COMA: Máquinas con un espacio de direcciones dinámico en el cual las memorias locales son tratadas en su gestión como memorias cache. Este es un caso particular de la arquitectura tipo NUMA en la cual no hay una jerarquía de memoria

dentro de cada procesador. El ejemplo clásico es la KSR *(Kendall* ***Square Research).***

Una revisión general de los sistemas de memoria compartida-distribuida incluyendo algoritmos, decisiones de diseño y sistemas actuales puede ser encontrada en [89].

Los sistemas AP1000, Cray T3E y Origin 2000 fueron las máquinas utilizadas para validar nuestras propuestas paralelas abarcando una gran parte del espectro de la clasificación introducida. Sus arquitecturas serán descritas en detalle en el capítulo 3.

1.4 Modelos de programación paralela

Las aplicaciones paralelas deben ser escritas siguiendo un modelo de programación. El caso más simple de ejecución paralela consiste en el modelo de multiprogramación, en el cual varios programas secuenciales son ejecutados simultáneamente sobre diferentes procesadores sin ninguna interacción entre ellos. Pero el caso más interesante lo constituyen los programas paralelos propiamente dichos. Básicamente se puede decir que existen cuatro alternativas para construir un programa paralelo: la paralelización manual usando pase de mensajes, la paralelización manual de sistemas de memoria compartida, la paralelización semi-automática basada en el paradigma de paralelismo de datos, y la paralelización automática.

- Modelo de programación de sistemas de memoria compartida: Los programas paralelos ejecutados en sistemas de memoria compartida se descomponen en varios procesos que comparten los datos asociados a una porción de su espacio de direcciones. La coordinación y cooperación entre los procesos se realiza a través de la lectura y escritura de variables compartidas y a través de variables de sincronización. Cada proceso puede llevar a cabo la ejecución de un subconjunto de iteraciones de un lazo común, o bien, de forma más general, cada proceso puede obtener sus tareas de una cola compartida. La programación más eficiente, aunque dificultosa, se establece a través de construcciones de bajo nivel tales como barreras de sincronización, regiones críticas, ***locks,*** semáforos, etc.

- Modelo de programación de pase de mensajes: En este modelo se define un conjunto de procesos con su propio espacio de memoria, pero que pueden comunicarse con otros procesos mediante el envío y la recepción de mensajes a través de la red de interconexión. El modelo asume que cualquier proceso puede enviar un mensaje a cualquier otro. La implementación de esta metodología se suele realizar utilizando librerías añadidas a los lenguajes de programación estándar, fundamentalmente C y Fortran. Actualmente también existen librerías estándar

ampliamente difundidas como PVM *(Parallel Virtual Muchine)* [31] y MPI *(Message Passing Interface)* [78,79].

- Modelo de programación de paralelismo de datos: Es un modelo de programación claramente heredado de las máquinas SIMD y se utiliza principalmente para simplificar la programación de sistemas de memoria distribuida. En esta aproximación, un programa secuencial en un lenguaje estándar es complementado con directivas o anotaciones insertadas en el programa para guiar al compilador en su tarea de distribuir los datos y las computaciones. Actualmente los lenguajes de paralelismo de datos más populares son CM-Fortran *(Connection Muchine* Fortran) [108], Fortran D [27], Craft [84], Vienna Fortran [116] y especialmente el considerado como estándar *High-Performance Fortrun* (HPF) [55].

- Paralelización automática: El compilador asume todas las estrategias y decisiones y genera automáticamente la versión paralela equivalente al código secuencial escrito en un lenguaje de programación convencional. En general los paralelizadores automáticos actuales ofrecen buenos resultados cuando los códigos a paralelizar tienen patrones de acceso a los datos regulares. Sin embargo, la paralelización automática de códigos irregulares, y entre ellos los códigos dispersos, es una tarea mucho más complicada y que no ha encontrado todavía una solución eficiente.

Nosotros hemos programado el AP1000 y el Cray T3E utilizando el modelo de programación de pase de mensajes y el Origin 2000 mediante el modelo de programación de memoria compartida. Además, hemos comprobado el efecto de los compiladores para la paralelización automática (ver sección 3.6).

Capítulo 2

Resolución de sistemas lineales dispersos: La factorización de Cholesky modificada

En este capítulo se realiza una breve introducción a la resolución de grandes sistemas lineales dispersos, en donde se enmarca la aplicación del algoritmo de Cholesky modificado. Primero se analizan brevemente algunos conceptos relacionados con las matrices dispersas, tales como sus modos de almacenamiento y su relación con la teoría de grafos. Para una revisión más detallada de estos conceptos son excelente fuentes [40] y [21]. Posteriormente nos centramos en los distintos métodos de resolución de grandes sistemas lineales: métodos directos y métodos iterativos; y finalmente, se concretará el análisis en los algoritmos de Cholesky y de Cholesky modificado.

2.1 Matrices dispersas

Las matrices dispersas aparecen en problemas tan diversos como los de optimización a gran escala, programación lineal, problemas de planificación, análisis de circuitos, dinámica computacional de fluidos, elementos finitos, la solución numérica de ecuaciones diferenciales y muchos otros.

Se dice que una matriz de dimensión M x N es dispersa si “muchos” de sus términos son cero. Se denomina índice de dispersión (β) a la relación entre el número de elementos no nulos de la matriz (a) y el número total de elementos (M x N), es decir, $\beta = \alpha/(M \text{ x } N)$. El valor de β necesario para que la matriz pueda ser considerada dispersa depende del problema a resolver, el patrón de la matriz y la arquitectura de la máquina en la cual se implementa el código. En general, se dice que una matriz es

dispersa si contiene un número suficiente de elementos nulos como para que resulte ventajoso explotar este hecho.

2.1.1 Modos de almacenamiento

Una matriz densa M x N se suele almacenar en un array bidimensional de dimensiones M x N. Sin embargo, si la matriz es dispersa, este almacenamiento desperdicia mucho espacio en memoria porque la mayoría de los elementos de la matriz son nulos y no necesitan ser almacenados explícitamente. Para matrices dispersas, es común almacenar sólo las entradas diferentes de cero añadiendo información sobre la localización de estas entradas.

Existen muchos esquemas de almacenamiento de matrices dispersas, los más ampliamente utilizados son: el CRS (Compressed Row Storuge), CCS (Compressed *Column Storuge),* BCRS (Block *Compressed Row Storuge),* CDS *(Compressed Diagonal Storage),* JDS *(Jugged Diagonal Storuge) y* SKS *(Sky-line Storuge) [21, 14].* La elección dependerá de las características del problema a resolver y del patrón (situación de las entradas no nulas) de la matriz.

Nosotros vamos a considerar el formato CCS por ser un modo de almacenamiento general que no hace ninguna asunción sobre el patrón de la matriz y que resulta adecuado en problemas que requieren patrones de acceso a la matriz por columnas. Este esquema representa la matriz por medio de tres vectores (DA, RO y CO). DA almacena las entradas no nulas de la matriz en el orden que se encuentran al recorrer la matriz por columnas, RO almacena el índice fila de cada una de dichas entradas y CO almacena la posición del vector de datos DA en la que empieza cada nueva columna. La Figura 2.2 muestra estos tres vectores para la matriz ejemplo de la Figura 2.1. Este tipo de almacenamiento implica que para almacenar una matriz de orden M x N no se necesitan $M \; x \; N$ posiciones de memoria, si no únicamente 2α + N + 1, siendo α el número de elementos no nulos de la matriz.

2.1.2 Teoría de grafos

La teoría de grafos ha estado ligada a la computación dispersa desde que Seymour Parter utilizó grafos no dirigidos para modelar la eliminación gausiana de matrices simétricas hace casi 40 años [83]. A cada matriz A simétrica de orden N x N se le puede asociar un grafo no dirigido denotado por $G(A)$ de tal forma que cada fila/columna de la matriz se corresponde con un nodo (vértice) de dicho grafo y entre dos nodos i, j existirá un eje si la correspondiente entrada, a_{ij}, es una entrada no nula de la matriz. En la Figura 2.3 se muestra un ejemplo. La teoría de grafos ayuda a visualizar el cambio en el patrón

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 & 0 & 11 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 \\ 2 & 4 & 0 & 0 & 0 & 0 & 0 & 13 \\ 0 & 0 & 5 & 0 & 0 & 0 & 10 & 0 \end{pmatrix}$$

Figura 2.1: Matriz dispersa de dimensiones 8 x 8 y $\alpha = 13$

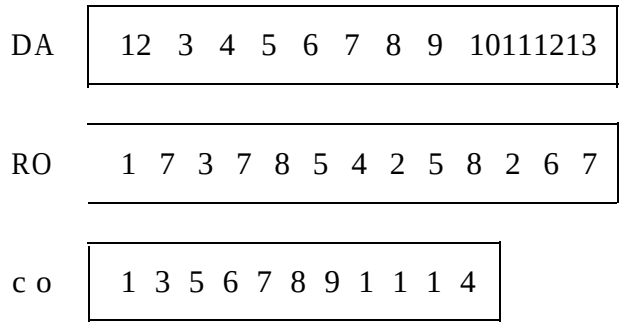

Figura 2.2: ***Modo de almacenamiento CCS para la matriz ejemplo de la Figura 2.1***

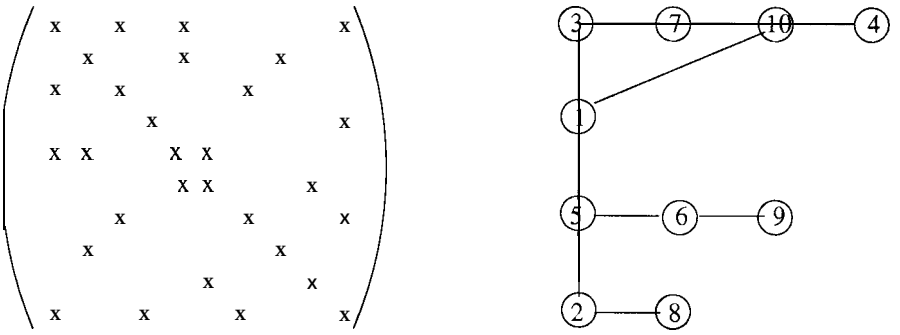

Figura 2.3: *Matriz dispersa simétrica y su* grafo

de entradas resultante al factorizar una matriz.

Si en un grafo existe un vértice r que está unido a cualquier otro nodo de G por una secuencia de nodos adyacente única, entonces G es un tipo especial de grafo denominado árbol, que denotaremos por T, siendo r la raíz de dicho árbol. Los arboles son una clase de estructura de datos fáciles de almacenar y manipular, sin embargo, en la práctica, es raro que el grafo asociado a una matriz tenga forma de árbol. Afortunadamente, ciertas propiedades entre las filas o columnas de las matrices dispersas se pueden representar como un grafo en forma de árbol; es el llamado árbol de eliminación [63, 96] del que hablaremos más adelante.

2.1.3 Conjuntos de matrices prueba

El método más equitativo de mostrar la eficiencia de los programas que trabajan con matrices dispersas es utilizar para validar los códigos un conjunto de matrices que sea lo más representativo posible.

Existe gran cantidad de conjuntos de matrices dispersas de dominio público de forma que se facilita el comparar resultados con otros grupos de investigación. El conjunto SPARSKIT, *Harwell-Boeing Sparse Matrix Collection,* la colección NEP *(Nonsymetric Eigenvalue Problem),* entre otros, son ejemplos de dichas colecciones.

Las matrices dispersas del conjunto conocido como *Harwell-Boeing Sparse Matrix Collection* han sido extensivamente usadas como matrices de prueba. El formato de almacenamiento de estas matrices es básicamente CCS junto con un cabecero con información adicional [22]. Nosotros hemos seleccionado para analizar el rendimiento de nuestros programas matrices de esta colección. Las características de las utilizadas, así como su procedencia, serán descritas en el momento de su utilización a lo largo de esta memoria.

2.2 Resolución de sistemas

Muchos de los problemas de matrices dispersas implican de un modo u otro la resolución de sistemas lineales dispersos de ecuaciones. Estos sistemas se pueden expresar en forma matricial como $Ax = b$, donde dada una matriz dispersa A de orden N y un vector b, *se* trata de determinar el vector solución x.

Existen dos estrategias de computación para abordar la resolución de estos sistemas lineales dispersos:

1. **Métodos iterativos:** Son técnicas que tratan de encontrar la solución de un sistema mediante sucesivas aproximaciones. Existen dos tipos de métodos iterativos: los métodos estacionarios, fáciles de entender e implementar pero con baja eficiencia, y los métodos no estacionarios, más complejos pero altamente eficientes [93, 14]. Los métodos no estacionarios difieren de los estacionarios en que las computaciones implican información que cambia en cada iteración. La velocidad a la que convergen estos métodos iterativos va a depender de la estructura de la matriz. Para que la convergencia sea más rápida se realizan una serie de transformaciones sobre dicha matriz, obteniéndose la denominada matriz precondicionada. Una propuesta de paralelización de diferentes métodos iterativos para la solución de sistemas lineales dispersos sobre máquinas de memoria distribuida fue desarrollada por nuestro grupo y puede ser encontrada en [53].

2. **Métodos directos:** Estos métodos se basan en la factorización de la matriz A para convertir el sistema lineal dado en un sistema con un formato de resolución simple, por ejemplo triangular [21]. Existen diferentes métodos de factorización de una matriz, entre los más habituales se encuentran la eliminación gausiana o factorización LU, la factorización QR, la factorización de Cholesky, etc. Todos ellos han dado lugar a varios estudios sobre su implementación en entornos de computación paralelos, ver por ejemplo [61,7, 111, 92].

La naturaleza del problema es la que determina cual es el método más apropiado. Los métodos directos son frecuentemente preferidos debido al esfuerzo que supone encontrar un buen precondicionador para los métodos iterativos, lo cual puede superar en coste a la factorización directa. Además, los métodos directos son la mejor alternativa para aplicaciones en las que es preciso resolver múltiples sistemas con la misma matriz de coeficientes y diferente vector b, ya que en este caso la factorización sólo tiene que ser computada una vez.

Existen una gran cantidad de librerías secuenciales disponibles para la resolución de sistemas lineales dispersos utilizando tanto métodos iterativos como métodos

directos. Entre ellas se pueden enumerar la librería HSL (Harwell *Subroutine Library) [107],* SPARSPAK [17] y YSMP (Yale *Sparse Matrix Package) [25].* El paquete SPARSPAK ha sido recientemente actualizado por dos nuevas versiones usando Fortran 90 y C++, estas dos nuevas aplicaciones han sido renombradas como SPARSPAK90 y SPARSPAK++ respectivamente [42]. Actualmente existen varios grupos de investigación trabajando en la implementación de librerías que permitan la resolución de sistemas lineales dispersos en paralelo. Nombrar por ejemplo la librería PSPASES (Parallel *SPArse Symmetric dirEct Solver) [48],* la cual ha sido desarrollada en MPI y cuyo objetivo es la resolución de sistemas lineales dispersos de matrices simétricas definidas positivas. Y el proyecto PARASOL [5], el cual tiene como objetivo el desarrollo de algoritmos dispersos paralelos de acuerdo con las necesidades industriales actuales. La librería incluye métodos directos de resolución, técnicas de descomposición de dominios y algoritmos *multigrid.*

2.3 Métodos directos

A pesar de su alto coste computacional, los métodos directos son útiles para la resolución de sistemas lineales dispersos porque son generales y estables numéricamente.

Consideremos el problema de resolver un sistema lineal de ecuaciones $Ax = b$. En general, la solución del sistema puede ser obtenida factorizando la matriz por medio de la eliminación gausiana, sin embargo, si la matriz es simétrica y definida positiva, se puede aplicar una variante más simple de la eliminación gausiana conocida como factorización de Cholesky. Aplicando la factorización de Cholesky a A se obtiene $A = LL^T$, donde el factor de Cholesky L es una matriz triangular inferior con elementos diagonales positivos. Con esta factorización, el vector solución del sistema, x, puede ser computado a través de sucesivas sustituciones hacia delante *(forward substitutions)* y hacia atrás *(back substitutions)* para resolver los sistemas triangulares $Ly = b$ y $L^T x = y$ respectivamente.

El patrón de la matriz L es similar al de A salvo por el hecho de que si la matriz es dispersa, durante el proceso de factorización algunas entradas que eran inicialmente nulas en la parte triangular inferior de A pueden llegar a ser entradas no nulas en L. Estas entradas son conocidas como fill o *fill-in.* En la Figura 2.4 cada "x" representa una entrada original en la matriz y cada "0" representa una nueva entrada creada en el proceso de factorización (fill). Para una ejecución eficiente conviene mantener la cantidad de fill lo más reducida posible. Sea P una matriz permutación, ya que PAP^T es también una matriz simétrica definida positiva, se puede elegir P de forma que el factor de Cholesky L' de PAP^T tenga menor fill que L. Con estas consideraciones se puede establecer que el problema de resolver sistemas dispersos definidos positivos de

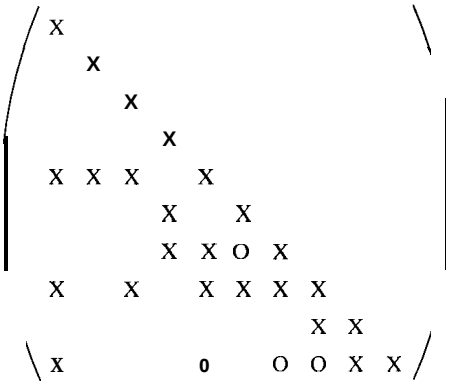

Figura 2.4: *Patrón de la matriz y su fill-in*

la forma $Ax = b$ puede ser dividido en cuatro fases:

- Ordenamiento: Encontrar un buen ordenamiento **P** tal que la factorización de Cholesky de $PAP^T = LL^T$ genere poco fill. Aunque es computacionalmente costoso encontrar una permutación **P** que minimice el fill, de hecho es un problema NP-completo, en la práctica se han encontrado y utilizado exitosamente muchas heurísticas tales como variantes del *minimum degree* [4, 9, 24, 41], ordenamientos de disección basados en particionamiento de grafos [13, 33,62, 87] y aproximaciones híbridas [ll, 16,521. Una revisión y comparación de las diferentes técnicas de ordenamiento ha sido realizada recientemente por uno de los miembros de nuestro grupo [86].

- Factorización simbólica: Determina la estructura del factor L *y* reserva la memoria necesaria para su almacenamiento [39].

- Factorización numérica: Se refiere a las computaciones sobre los coeficientes de la matriz A para calcular la matriz factor L.

- Solución triangular: Solución de los sistemas triangulares $Ly = b$ *y* $L^T x = y$. Dado que la solución triangular requiere muchas menos operaciones que la factorización, el tiempo que conlleva su resolución constituye una pequeña fracción del tiempo total, típicamente por debajo del 5% en un código secuencial.

Existe un elevado número de propuestas de paralelización de cada una de las fases. En [37] se discute una adaptación del algoritmo de ordenamiento de disección anidada automática [38] para su ejecución sobre un sistema de memoria distribuida. En [67] se proponen ordenamientos adecuados para el procesamiento paralelo de la matriz. En [35] se puede encontrar una propuesta de paralelización de la factorización simbólica.

```
1 . for j=1 to N do
2 .    l_jj = sqrt(l_jj)
3 .    for each l_sj ≠ 0 (s > j) do
4 .      l_sj = l_sj / l_jj
5 .    endfor
6 .    for each l_sj ≠ 0 (s > j) do
7 .      for each l_ij ≠ 0 (i > s) do
8 .        l_is = l_is − l_sj l_ij
9 .      endfor
10.    endfor
11. endfor
```

Figura 2.5: *Algoritmo de Cholesky orientado por columnas*

En cuanto a la resolución triangular, algunos trabajos recientes son [3], [49] y [47]. Nosotros nos hemos centrado en la paralelización de la factorización numérica, la cual es la fase computacionalmente más costosa de todo el proceso. Para una revisión de la factorización de Cholesky estándar dispersa paralela se puede consultar [5 1].

2.4 Factorización numérica

A continuación describiremos la factorización de Cholesky y la factorización de Cholesky modificada, sus diferentes orientaciones algorítmicas y todos aquellos factores que son relevantes en su realización paralela.

2.4.1 Factorización de Cholesky estándar

Como ya se ha introducido, la factorización de Cholesky consiste en la reducción de una matriz simétrica A definida positiva en el producto matricial LL^T, donde L *es* una matriz triangular inferior con los elementos diagonales positivos. En la Figura 2.5 mostramos un algoritmo secuencial in *place* orientado por columnas que lleva a cabo la factorización de Cholesky de una matriz dispersa N x N.

Siguiendo la notación introducida por Liu en [36], la factorización de Cholesky puede expresarse en términos de las columnas de la matriz dispersa, constando

```
1. for j=1 to N do
2.     cdiv(j)
3.     for each l_sj ≠ 0 (s > j) do
4.       cmod(s,j)
5.     endfor
6. endfor
```

Figura 2.6: *Algoritmo right-looking* para la *factorización de Cholesky*

básicamente de dos funciones:

1. cdiv('j): Normalización de la columna **j**. Computa la raíz cuadrada del elemento diagonal a_{jj} y escala la j-ésima columna de $\boldsymbol{A}$ por $1/\sqrt{l_{jj}}$ para producir la j-ésima columna factor L_{*j} (pasos *2-5).*

2. cmod(s,j): Modificación de la columna s por la columna $j, s > j$. Esto implica la sustracción de la columna j de la columna $s\, l_{sj}$ veces (lazo 6- 10).

Existen dependencias entre dichas funciones. De hecho, la modificación de una columna j por sus predecesoras debe realizarse con anterioridad a su normalización, y la modificación de una columna $k (k > j)$ por la columna j no puede empezar hasta que ésta haya sido normalizada. Dependiendo del orden en el que se realizan estas funciones se pueden encontrar en la bibliografía tres algoritmos básicos:

- El algoritmo *right-Zooking,* también llamado submatriz, en el cual primeramente se realiza la normalización de la columna j, y a continuación se utiliza dicha columna para modificar todas las posteriores (ver Figura 2.6).
- El algoritmo *Zeft-Zooking, o* por columnas, en el cual una columna es primeramente modificada por todas las columnas anteriores y posteriormente se normaliza (ver Figura 2.7).
- Y el algoritmo por filas, idéntico al anterior pero accediendo a la matriz por filas en vez de por columnas.

Estos tres algoritmos tienen diferentes patrones de referencia a memoria en base a que partes de la matriz son accedidas y modificadas en cada etapa de la factorización

```
1. for j=1 to N do
2.     for each l_js ≠ 0 (s < j) do
3.         cmod(j,s)
4.     endfor
5.     cdiv(j)
6. endfor
```

Figura 2.7: *Algoritmo left-looking para la factorización de Cholesky*

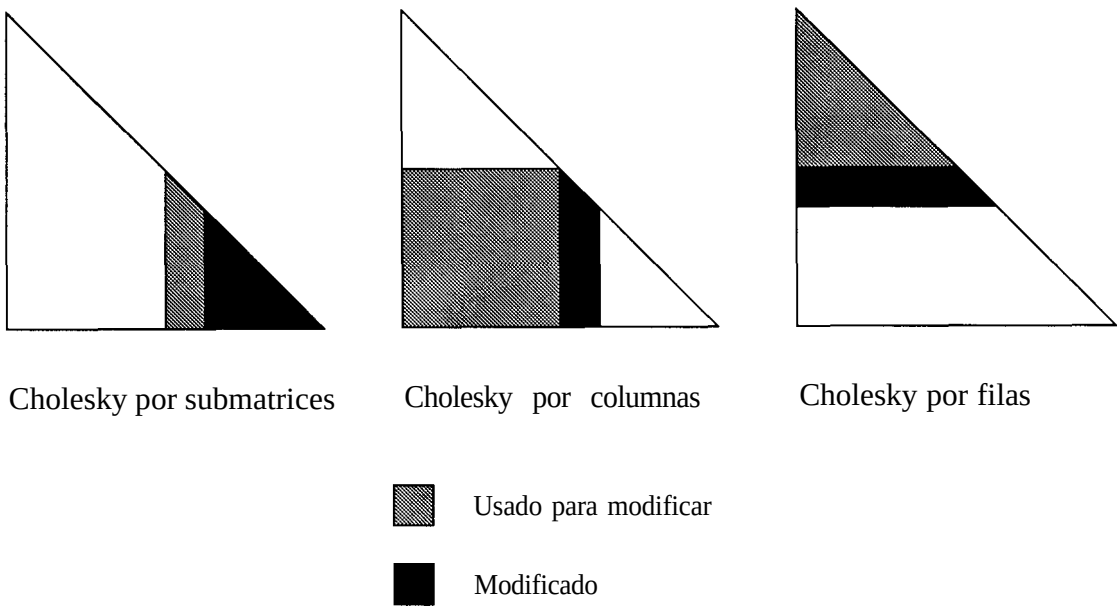

Figura 2.8: Los *tres tipos* básicos de factorización de Cholesky

(ver Figura 2.8), y cada uno de ellos tiene sus ventajas y desventajas dependiendo del entorno de programación y de la aplicación.

Otras propuestas se basan en aproximaciones multifrontales, que son esencialmente variaciones del método submatriz o right-looking. El algoritmo multifrontal para la factorización de matrices dispersas fue propuesto independientemente por Speelpening [104] y Duff y Reid [23]. Un buen estudio sobre este método puede ser encontrado en [69]. La idea consiste en reorganizar la factorización de una matriz dispersa en una secuencia de factorizaciones parciales de matrices densas más pequeñas. Una de las razones más importantes para usar el método multifrontal es que las matrices frontales se pueden tratar como densas, y si el hardware soporta, por ejemplo, unidades vectoriales, éstas se pueden explotar con una mayor eficiencia. Además, la localización de las referencias a memoria en este método es ventajosa en la explotación de la cache o en máquinas con memoria virtual. Una adaptación del algoritmo multifrontal para la factorización de Cholesky modificada en paralelo fue propuesta por nuestro grupo y es extensamente explicada en [80]

2.4.2 Factorización de Cholesky modificada

La factorización de Cholesky modificada fue introducida inicialmente por Gill y Murray [45], y posteriormente refinada por Gill, Murray y Wright [46]. Dada una matriz simétrica $A \in \mathbf{R}^{N \times N}$, no necesariamente definida positiva, calcula una factorización de Cholesky de otra matriz $\overline{A} = A + E$, donde E es cero si A *es* definida positiva, y es una matriz diagonal no negativa para la cual $A + E$ *es* definida positiva en otro caso. Esta técnica se utiliza para resolver sistemas lineales $Ax = b$ para matrices de coeficientes A que son simétricas pero no necesariamente definidas positivas. No se intenta resolver sistemas en el sentido usual, ya que el sistema modificado $\overline{A}\overline{x} = b$, con $\overline{A} \neq A$, puede producir una solución $\overline{x}$ que no se aproxima en absoluto a x. Esta factorización sin embargo es apropiada cuando existe justificación para modificar un sistema lineal, como en los métodos de Newton utilizados en problemas de optimización no lineal [46, 95] o para computar precondicionadores definidos positivos [20,94].

Nótese que para el caso de matrices definidas positivas, la factorización de Cholesky modificada coincide con la factorización de Cholesky estándar, es decir, la factorización de Cholesky modificada puede verse como una generalización del algoritmo estándar.

La factorización de Cholesky modificada descompone la matriz $A' = A + E$ en el producto LDL^T, donde D *es* una matriz diagonal y L *es* una matriz triangular inferior con unos en la diagonal. La matriz diagonal E no se calcula de forma explícita durante la factorización si no que es computada implícitamente al calcular D *y* L de tal forma que se cumpla que LDL^T *es* definida positiva y los factores están todos limitados. Esta propiedad se satisface si se asegura que los elementos de D *y* L verifican las

1 . for j=1 to N do
2 . Computar fila j de L:
3 . $l_{js} = a_{js}/d_s \quad s = 0, \ldots, j-1$
4 . Actualizar columna j de A:
5 . $a_{sj} = a_{sj} - \sum_{k=0}^{j-1} l_{jk} a_{sk} \qquad s = j+1, ..., N$
6 . calcular $\theta_j = max_{s \in \{j+1,\ldots,n\}} \{|a_{sj}|\}$
7 . computar $d_j = max\{\delta, |a_{jj}|, \theta_j^2/\beta^2\}$
8 . Actualizar la diagonal de A para $s > j$:
9 . $a_{ss} = a_{ss} - a_{sj}^2/d_j \qquad s = j+1, \ldots .$ N
10. endfor

Figura 2.9: Algoritmo de Cholesky modificado orientado por *filas*

condiciones:

$$d_k > \delta$$
$$||lik\sqrt{d_k}|| \leq \beta, i > k$$

donde δ es una pequeña cantidad positiva y β se calcula desde el mayor valor absoluto de los elementos de la diagonal y de los elementos por debajo de la diagonal de la matriz A, de tal forma que se minimice un límite superior para $||E||_\infty$, a la vez que se asegura que $E = 0$ si A *es* definida positiva. En el algoritmo de Cholesky modificado *in place* usual L *se* construye por filas siguiendo el proceso mostrado en la Figura 2.9

En el j-ésimo paso del algoritmo los primeros $j-1$ elementos de D y las primeras $j-1$ filas de L sobreescriben las primeras $j-1$ filas de A (ver Figura 2.10).

Ya que las matrices dispersas son almacenadas usando el formato CCS, es decir, se almacenan por columnas, será más conveniente utilizar un algoritmo orientado por columnas como el mostrado en la Figura 2.11.

En este algoritmo, en el paso j-ésimo los primeros $j-1$ elementos de D *y* las primeras $j-1$ columnas de L sobreescriben las primeras $j-1$ columnas de A, tal y como se ilustra en la Figura 2.12.

Nótese que en el algoritmo por columnas todas las actualizaciones realizadas con la columna j se llevan a cabo a la vez que se computa dicha columna (pasos 8 al 13), por eso, el cálculo de la columna s en la s-ésima iteración del lazo externo se reduce a una simple división por d_s. En contrapartida, en el algoritmo orientado por filas, la columna

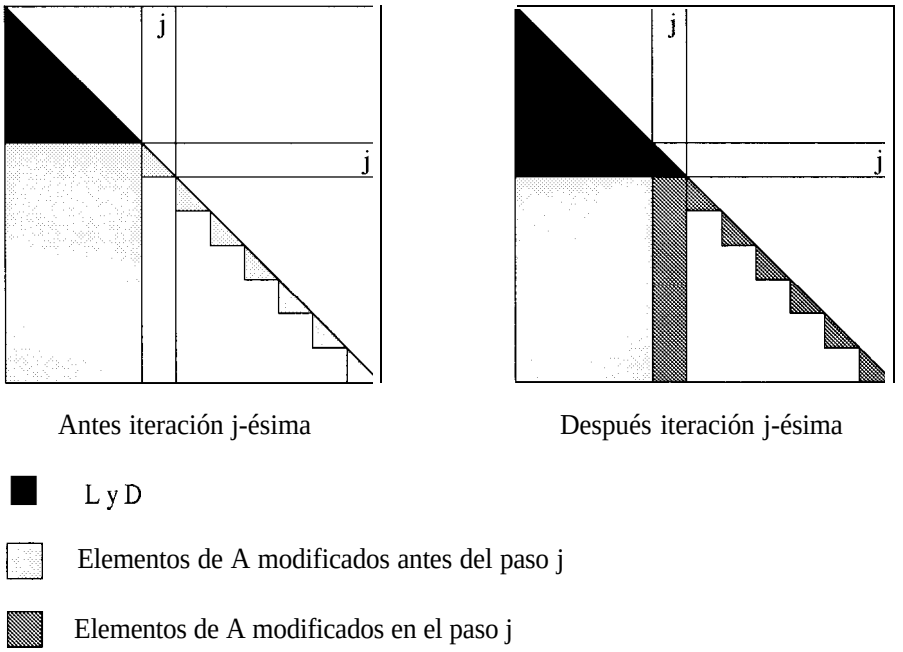

Figura 2.10: Paso ***j-ésimo en el algoritmo secuencial por filas***

```
1. for j=1 to N do
2.    calcular θ_j = max_{s∈{j+1,...,n}}{|a_sj|}
3.    computar d_j = max{δ, |a_jj|, θ_j^2/β^2}
4.    actualizar la diagonal de A para s > j:
5.      a_ss = a_ss − a_sj^2/d_j      s = j+1, .... N
6.    actualizar los elementos no diagonales de A
7.      mientras se computa la columna j de L:
8.      for s = j + 1 to N do
9.        l_sj = a_sj/d_j
10.         for k = s + 1 to N do
11.           a_ks = a_ks − l_sj a_kj
12.         endfor
13.       endfor
14. endfor
```

Figura 2.11: ***Algoritmo de Cholesky modificado orientado por columnas***

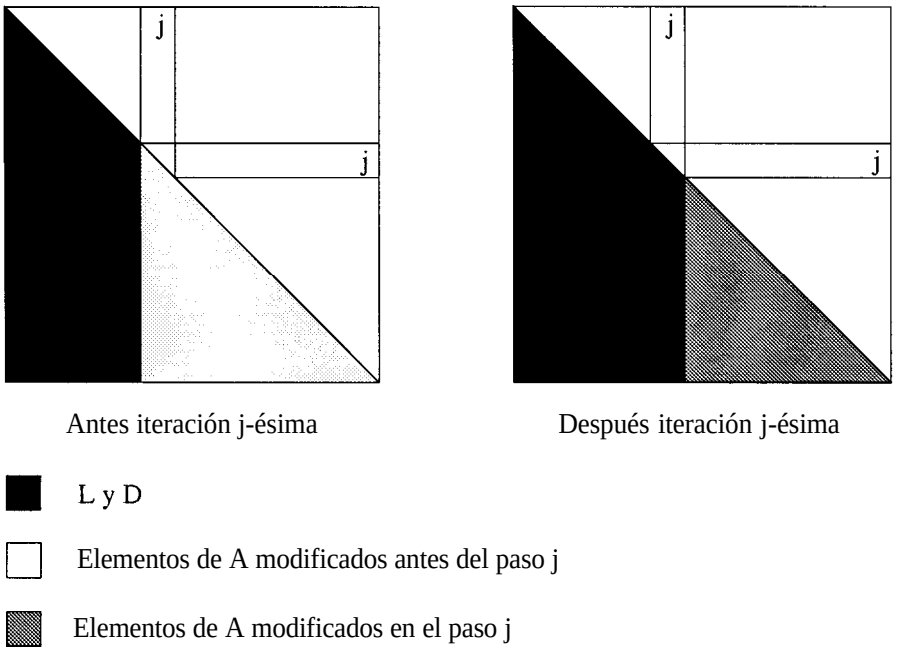

Figura 2.12: *Paso j-ésimo en el algoritmo secuencial orientado por columnas*

s no es actualizada (excepto su diagonal) hasta que se calcula la fila s en la iteración s-ésima del lazo externo.

Las implementaciones para matrices dispersas deben tener en cuenta el hecho de que una columna s sólo necesitará ser actualizada por otra columna j si $l_{sj} \neq 0$. Tales implementaciones vendrán precedidas por una etapa de ordenamiento de la matriz para reducir el fill-in en L y por una etapa de factorización simbólica que determine el patrón de L de forma que se puede realizar a priori la reserva de memoria. Ambas etapas son idénticas a las aplicadas para el caso del algoritmo de Cholesky estándar. En el caso de los métodos de Newton utilizados en problemas de optimización [46, 94] esta factorización se utiliza dentro de un proceso iterativo en el cual, el patrón de la matriz no varía y, por tanto, la factorización simbólica y el ordenamiento se computan solamente una vez mientras que la factorización numérica se ejecuta varias veces.

En la factorización de Cholesky estándar existen, como hemos visto, dos posibles orientaciones del algoritmo por columnas, la orientación *right-looking* y la *left-looking*. Para adaptar nuestro algoritmo a esos dos esquemas es necesario reestructurar el código de la forma mostrada en las Figuras 2.13 y 2.14, respectivamente. Siguiendo la notación introducida por Liu, ahora tendremos las funciones:

- $cdiv(j)$: cálculo de la j-ésima diagonal y división de todas las entradas de la

```
1 . for j=1 to N do
2 .    θ_j = max_{s∈{j+1,...,n}}{|a_sj|}
3 .    d_j = max{δ, |a_jj|, θ_j^2/β^2}
4 .    for each l_sj ≠ 0 (s > j) do
5 .      l_sj = a_sj/d_j
6 .    endfor
7 .    for each l_sj ≠ 0 (s > j) do
8 .      for each l_kj ≠ 0 (k ≥ s) do
9 .        a_ks = a_ks − l_sj l_kj d_j
10.      endfor
11.    endfor
12. endfor
```

```
for j=1 to N do
  cdiv(j)
  for each l_sj ≠ 0 (s > j) do
    cmod(s,j)
  endfor
endfor
```

Figura 2.13: *Algoritmo right-looking para* la *factorización de Cholesky modificada*

columna j por esta diagonal (pasos 2 al 6 del algoritmo *right-Zooking y* pasos 7 al 11 del *Zeft-Zooking).*

- $cmod(j, s)$*:* modificación de las entradas en la columna j por las columnas $s (j >$ s) incluyendo el elemento diagonal. Es decir, modificación de todas las columnas posteriores por la columna actual para el caso del algoritmo *right-Zooking* (pasos 7 al 1 1), y modificación de la columna actual por todas las precedentes en el caso del *Zeft-Zooking* (pasos 2 al 6).

Cabe destacar que cuando se utilizan matrices dispersas y se usa una columna j para modificar otra columna s, generalmente las columnas j y s tienen diferentes patrones de entradas; el patrón de la columna destino s es un superconjunto del de la columna fuente j. En la implementación directa del algoritmo *right-Zooking* secuencial el problema de sumar un múltiplo de la columna j dentro de la columna s se resuelve realizando una búsqueda a través de las entradas en la columna destino para encontrar las localizaciones apropiadas en las cuales se debe realizar la suma. Una posible optimización consiste en comprobar si el número de entradas en la columna j es igual al número de entradas en la columna k bajo la fila j. Si son iguales, entonces ambas columnas tienen la misma estructura. En este caso se pueden ignorar por completo los vectores de índices y realizar la suma directamente.

En el caso del algoritmo *Zeft-Zooking,* la misma columna j es utilizada varias veces consecutivas para ser modificada por otras columnas $s, s < j$. En este caso la forma

```
1. for j=1 to N do
2.    for each l_js ≠ 0(s < j) do
3.       for each l_ks ≠ 0(k ≥ j) do
4.          a_kj = a_kj − l_js l_ks d_s
5.       endfor
6.    endfor
7.    θ_j = max_{s∈{j+1,...,n}}{|a_sj|}
8.    d_j = max{δ, |a_jj|, θ_j^2/β^2}
9.    for each l_sj ≠ 0(s > j) do
10.      l_sj = a_sj/d_j
11.   endfor
12. endfor
```

```
for j=1 to N do
   for each l_sj ≠ 0(s < j) do
      cmod(j,s)
   endfor
   cdiv(j)
endfor
```

Figura 2.14: *Algoritmo left-looking para la factorización de Cholesky modificada*

más eficiente de llevar a cabo la suma de las columnas fuente dentro de la columna destino es a través de un vector denso expandido. En este vector se acumulan todas las modificaciones desde las diferentes columnas s a la columna j. Después de realizar todas las modificaciones, el vector se vuelca sobre la columna dispersa j tal y como se muestra en la Figura 2.15. En esta figura mostramos el vector de datos *(DA)* para la matriz ejemplo de la Figura 2.4, y el vector denso expandido para el cálculo de la quinta columna. Esta quinta columna será modificada por las columnas 1, 2 y 3, tras lo cual el vector denso será de nuevo volcado sobre el vector de datos *DA*.

Las columnas s que modifican a la columna j en el código *Zeft-Zooking* son exactamente aquellas para las cuales $l_{js} \neq 0$, es decir, el conjunto de columnas s que modifican una columna j viene dado por las entradas no nulas en la fila j de la matriz. Dado que se va a considerar la matriz dispersa almacenada por columnas en formato CCS, el acceso por filas a la matriz no es inmediato. Por eso en las implementaciones secuenciales del método *Zeft-Zooking es* eficiente utilizar una lista enlazada que mantenga los índices de las columnas s que modifican otra columna j [40]. La construcción de esta lista enlazada se efectúa en tiempo de ejecución.

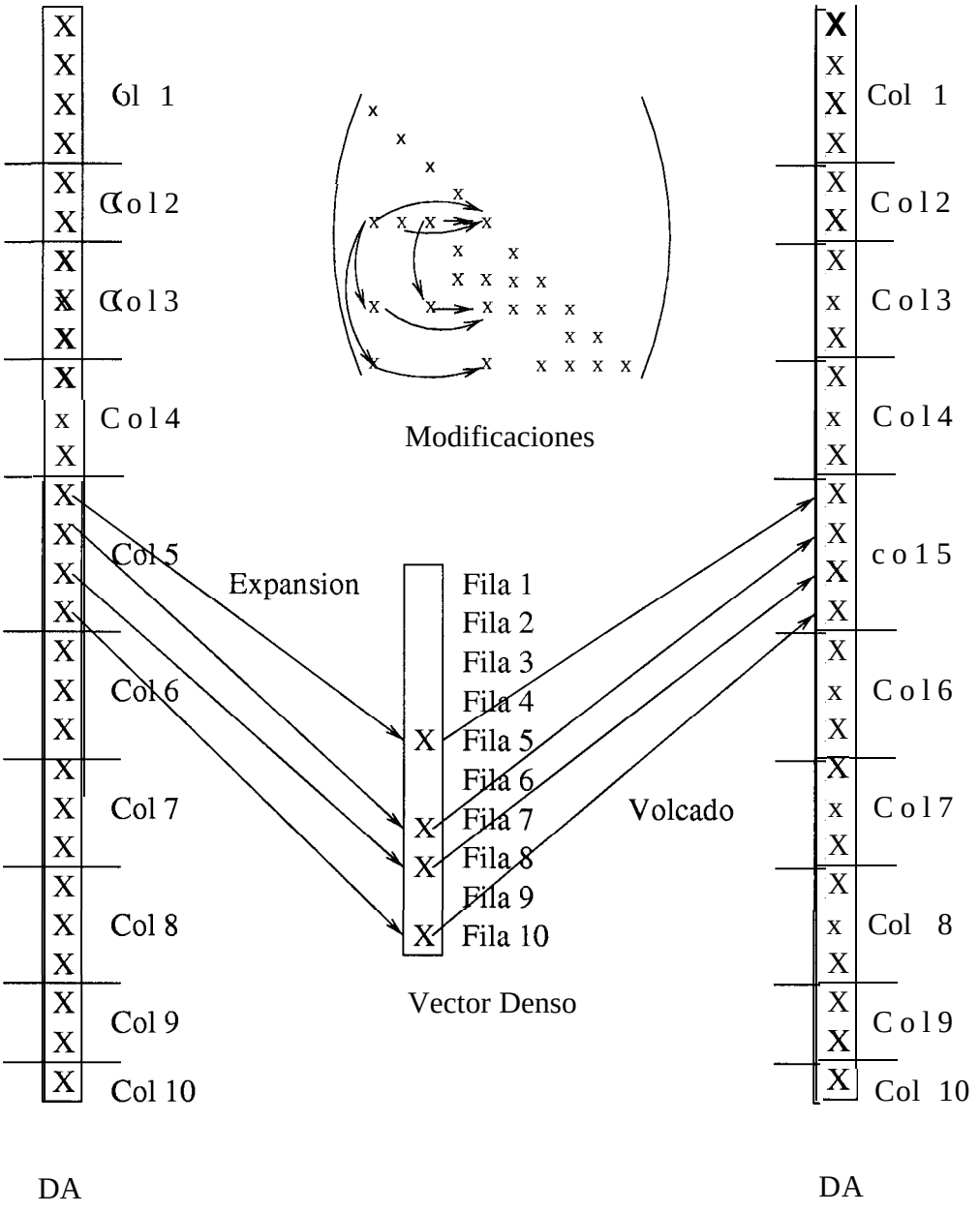

Figura 2.15: *Utilización de un vector denso expandido en el algoritmo left-looking*

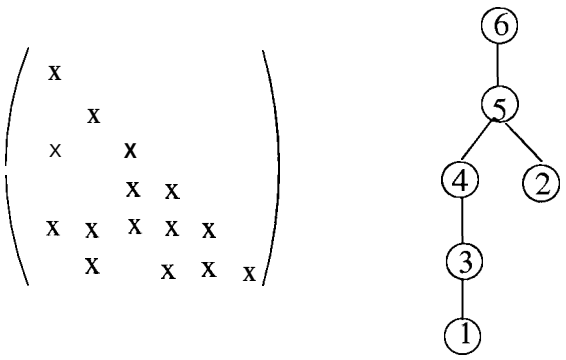

Figura 2.16: *Estructura del factor de Cholesky L y su árbol de eliminación asociado*

2.4.3 Árbol de eliminación

Sea A una matriz dispersa simétrica definida positiva e irreducible. El árbol de eliminación de A, T(A), se define como una estructura con N nodos, donde cada nodo representa una columna de la matriz. El nodo p *es* padre de j si y sólo si

$$p = min\{i \,/\, l_{ij} \neq 0,\ \mathrm{i} > j\}$$

donde L *es* el factor de Cholesky de la matriz A. Es decir, el padre de la columna j es determinado por el primer elemento no nulo debajo de la diagonal en la columna j. En otras palabras, el padre de la columna j es la primera columna modificada por j. En la Figura 2.16 se muestra la estructura del factor de Cholesky L *y su* correspondiente árbol de eliminación. El árbol de eliminación es el mínimo subgrafo de $G(A)$ que proporciona la descripción de las dependencias entre columnas en el cálculo del factor de Cholesky L. Una columna sólo puede modificar a sus antecesores en el árbol de eliminación y, equivalentemente, una columna sólo puede ser modificada por sus descendientes.

En [68] se discute el uso de los arboles de eliminación en diferentes aspectos de la resolución de grandes sistemas lineales dispersos, incluyendo el ordenamiento, la factorización simbólica, la factorización numérica y los esquemas de almacenamiento disperso. La resolución triangular está también condicionada por la estructura del árbol de eliminación. La sustitución hacia delante (forward *substitution, Ly = b)* avanza desde los nodos hoja hacia la raíz del árbol, y la sustitución hacia atrás *(backward substitution,* $L^T x = y$) avanza desde la raíz hacia las hojas.

La utilización del árbol de eliminación será de gran utilidad en la implementación de las soluciones paralelas, tanto en memoria distribuida como compartida. En concreto, en memoria distribuida el algoritmo de distribución eficiente de los datos para el algoritmo de Cholesky modificado multifrontal está basado en el árbol de eliminación de la matriz

[8 1] En memoria compartida el árbol de eliminación nos proporcionará información para aumentar la localidad de los datos en las implementaciones paralelas y, por tanto, ayudará a aumentar el rendimiento [74, 75]. En ambos casos las estrategias aplicadas son generalizables a otros problemas dispersos tales como otro tipo de factorización o la resolución de sistemas triangulares. En general serán aplicables a todos aquellos códigos en los cuales las dependencias puedan ser representadas mediante un árbol.

2.5 Algoritmos paralelos

El primer paso en el diseño de la factorización paralela es el establecimiento del modelo computacional que se va a utilizar. En ese sentido, Liu usa el árbol de eliminación de la matriz para analizar los niveles de paralelismo en la factorización de Cholesky estándar obteniendo las siguientes alternativas en el modelo de programación [64]:

- paralelismo de grano fino, en el cual cada tarea paralela es una operación en punto flotante, es el modelo introducido por Wing y Huang en [112].
- paralelismo de grano medio, en el cual cada tarea paralela es una operación con una columna, es decir, una operación cmod o $cdiv$, es el modelo introducido por Liu en [64].
- paralelismo de grano grueso, en el cual cada tarea paralela se corresponde con un subárbol del árbol de eliminación, es el modelo introducido por Jees y Kees en [57].

En esta memoria se analizan los rendimientos que se pueden obtener al explotar cada uno de estos tres tipos de paralelismo sobre la factorización de Cholesky modificada

Más concretamente, el paralelismo de grano grueso se refiere al trabajo independiente al computar columnas pertenecientes a subárboles disjuntos dentro del árbol de eliminación de la matriz. Este tipo de paralelismo es sólo disponible en la factorización dispersa y es en el que nos hemos centrado para proponer el algoritmo de distribución de los datos para la factorización de Cholesky modificada multifrontal sobre sistemas de memoria distribuida [8 1], y para proponer nuevos schedulings que mejoren la localidad de los datos sobre los sistemas de memoria compartida [74,75].

Desde luego no es el único paralelismo posible, dentro del mismo subárbol o entre subárboles no disjuntos se puede explotar el paralelismo que ofrecen las operaciones cmod de forma individual.

Sean j_1 y j_2 dos índices columna cuyos subárboles no son disjuntos, y sean k_1 y k_2 dos índices columna que modifican j_1 y j_2 respectivamente, claramente las operaciones

de modificación $cmod(j_1, k_1)$ y $cmod(j_2, k_2)$ pueden ser llevadas a cabo en paralelo. A esto se refiere el paralelismo de grano medio que es analizado en esta memoria tanto para los diferentes algoritmos fan-in y *fan-out* propuestos en el capítulo 4 para sistemas de memoria distribuida [77], como para los algoritmos *right-Zooking y left-looking* propuestos en el capítulo 5 para sistemas de memoria compartida [74,75].

En cuanto al paralelismo de grano fino, ha sido el utilizado en las primeras versiones paralelas sobre máquinas de memoria distribuida descritas también en el capítulo 4, y se ha demostrado poco rentable su implementación paralela, además de requerir un mayor grado de complejidad [76].

2.51 Límites al rendimiento paralelo

Un factor obvio que limita el rendimiento de los programas paralelos es la carga computacional asignada a cada procesador. La computación en paralelo no puede finalizar en menos tiempo que el máximo de los tiempos que precisa cada procesador individual para ejecutar las tareas que le han sido asignadas, ignorando todas las dependencias entre las tareas. Este límite es usualmente conocido como balanceo de la carga y va a ser la distribución de los datos realizada por el programador la que determine su valor.

Otro límite importante al rendimiento es lo que se conoce como el camino crítico. La computación en paralelo no puede finalizar en menos tiempo que el requerido para ejecutar secuencialmente la cadena de tareas dependientes más larga dentro del problema a resolver. En el caso del algoritmo de Cholesky, cada camino en el árbol de eliminación desde un nodo hoja hasta la raíz del árbol forma una cadena de tareas dependientes. Por tanto, el camino más largo desde un nodo hoja hasta la raíz define el camino crítico para cada una de las matrices a factor-izar.

Nosotros utilizamos, para las propuestas de memoria distribuida, una distribución cíclica de los datos que garantiza en la mayoría de los casos el buen balanceo de la carga y minimiza, en consecuencia, el efecto del primero de los límites.

En cuanto al camino crítico, existen métodos de ordenamiento de la matriz que intentan minimizar la altura del árbol y reducir, por tanto, este camino crítico, aumentando con ello el paralelismo potencial. Un ejemplo son los ordenamientos equivalentes presentados por Liu [67] basados en rotaciones del árbol de eliminación [66]. Se entiende por ordenamientos equivalentes aquellos que reestructuran el árbol de eliminación de la matriz sin generar fill-in adicional. Liu propone aplicar dos ordenamientos consecutivamente, el primero para minimizar *el fill-in, y el* segundo un ordenamiento equivalente que minimice la altura del árbol. Otro ejemplo es el algoritmo presentado por Lewis et al. [60], basado en el método introducido por Jees y Kees [57], en el cual el ordenamiento de la matriz para una factorización paralela eficiente también se lleva a cabo

en dos etapas. En la primera etapa se aplica un ordenamiento, P, para minimizar el *fi11* de la matriz *A* a factorizar. En la segunda etapa se computa un ordenamiento paralelo a partir del grafo de la matriz $F = L + L^T$, siendo *L* el factor de Cholesky de la matriz PAP^T. Entre todos los ordenamientos **cuyo** *fill* está incluido en el grafo de este ordenamiento es el que ofrece el número mínimo de pasos en la factorización paralela de *A*.

Nosotros nos hemos centrado a lo largo de esta memoria en la factorización numérica, y no consideramos ni estudiamos ningún tipo de ordenamiento que minimice el camino crítico. Por tanto, en nuestro caso, el camino crítico va a ser un límite importante al rendimiento de los códigos paralelos.

2.6 Resumen

Existe un grupo de códigos irregulares cuyas dependencias pueden ser representadas en términos de un grafo en árbol, conocido habitualmente como árbol de eliminación. La factorización de Cholesky se puede considerar como un algoritmo representativo de este tipo de códigos. Diferentes algoritmos de factorización de matrices, así como la resolución de sistemas triangulares, son ejemplos de esta clase de problemas.

En todos estos algoritmos, el camino crítico impuesto por las dependencias es un fuerte límite a la eficiencia de los códigos paralelos.

Además, la factorización de Cholesky modificada se puede considerar como una generalización de la factorización de Cholesky estándar. La principal diferencia entre ambos algoritmos radica en el hecho de que la factorización de Cholesky modificada requiere una mayor número de operaciones para el cálculo de los elementos diagonales y tiene, por tanto, un mayor coste computacional.

Cabe destacar que no existen, que nosotros conozcamos, versiones paralelas de la factorización de Cholesky modificada. Nuestra propuesta se basa en adaptar los algoritmos para la factorización de Cholesky estándar a la factorización de Cholesky modificada y optimizarlos en función, tanto de las características hardware, como de las condiciones específicas de la factorización.

Capítulo 3

Sistemas paralelos y entornos de programación

En este capítulo se describen brevemente las arquitecturas de las máquinas utilizadas, APlOOO de Fujitsu, Cray T3E y Origin 2000 de SGI, así como las librerías y herramientas de programación empleadas para desarrollar los códigos paralelos sobre ellas. Incluimos además una sección dedicada a la paralelización automática de la factorización de Cholesky modificada dispersa.

3.1 AP1000 de Fujitsu

El AP1000 es un computador multiprocesador de memoria distribuida comercializado por Fujitsu. Consta de 64 a 1024 procesadores o celdas los cuales se encuentran interconectados mediante tres redes de comunicación específicas e independientes, como se ilustra en la Figura 3.1:

- red de radiación o red B: para comunicaciones entre el *host y* las celdas, así como para la distribución y la recolección de datos,
- red toroide o red T: para comunicaciones punto a punto entre las celdas. La velocidad de comunicación entre celdas es de 25 MB/s.
- red de sincronización o red S: empleada en las sincronizaciones de barrera.

El sistema AP1000 requiere un computador *host* (por ejemplo, una estación de trabajo) para realizar tareas de control e interacción con el usuario. Las celdas del AP1000, cuya configuración se muestra en la Figura 3.2, están basadas en microprocesadores

SPARC y constan de una unidad de enteros (IU), una unidad de punto flotante (FPU), un controlador de mensajes (MSC), un controlador de encaminamiento (RTC), una in-*terface* con la red B (BIF), 16 MB de memoria RAM dinámica (DRAM) y 128 KB de memoria cache.

Durante la operación normal, los MSCs trabajan como controladores de memoria cache con un esquema de ubicación directa y una estrategia de postescritura *(write-back* para actualizar la memoria principal, siendo el tamaño de las líneas de 4 palabras.

Los controladores MSC, RTC, BIF y el de la DRAM de cada celda están conectados a través del bus local síncrono de 32 bits, denominado LBUS en la Figura 3.2. Además cada celda tiene un conector de LBUS externo que permite la instalación de varias opciones *hardware* tales como *interfaces* de E/S de alta velocidad, de disco y de memoria adicional. Más detalles sobre la arquitectura de este sistema pueden ser encontrados en [561.

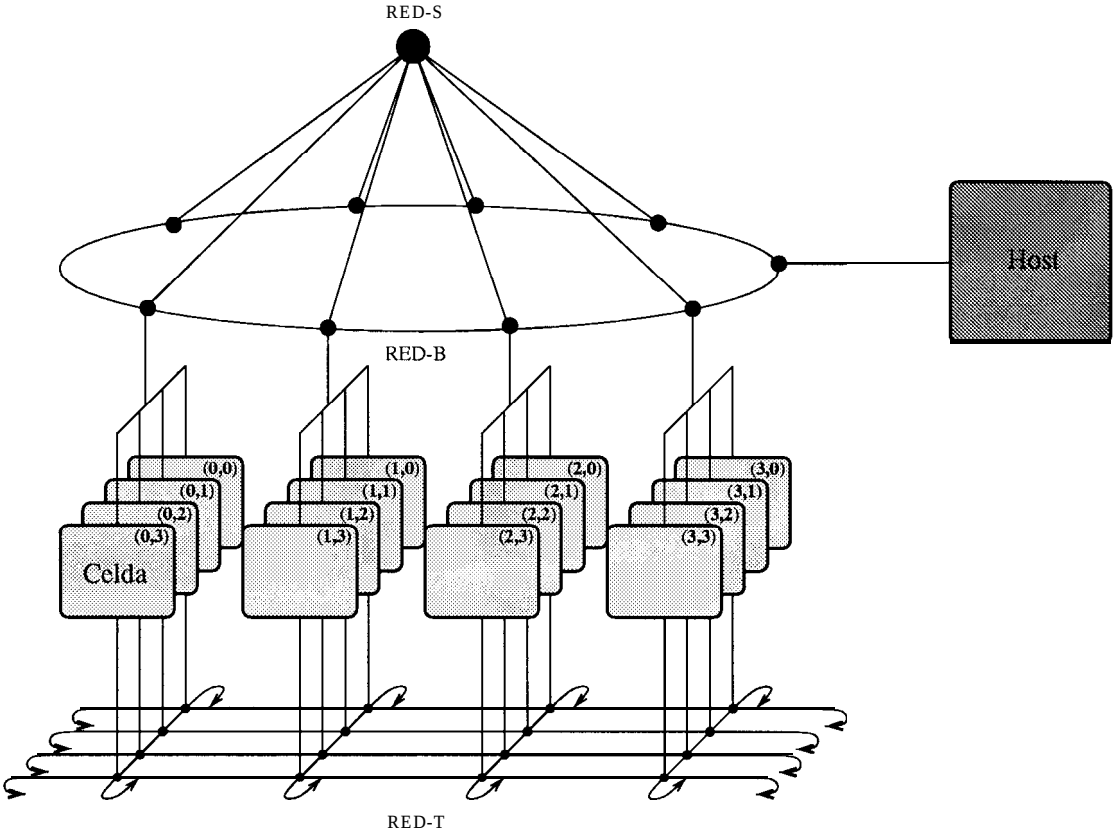

Figura 3.1: *Arquitectura del Fujitsu AP1 000*

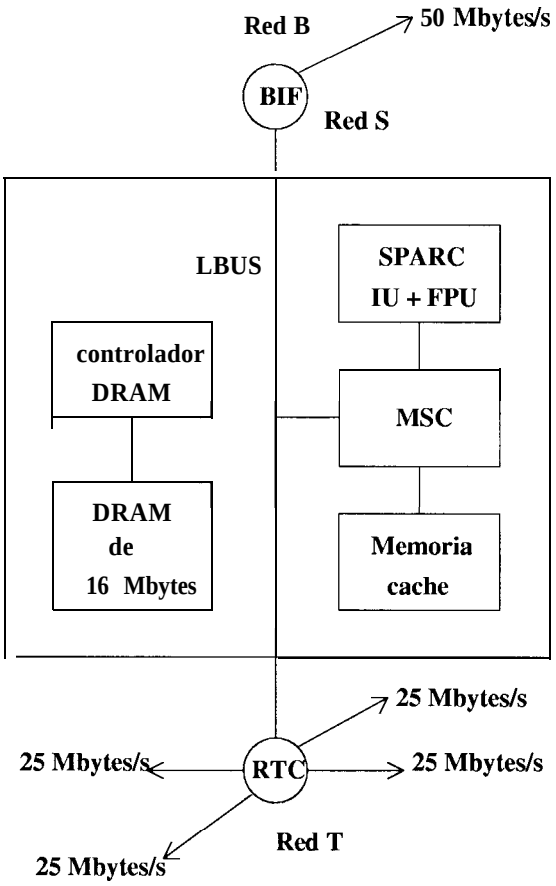

Figura 3.2: ***Configuración de*** las ***celdas del Fujitsu APIO00***

3.2 Cray T3E

El Cray T3E es un sistema de memoria compartida físicamente distribuida tipo NUMA. Consta de 16 a 2048 procesadores DEC Alpha 21164 conectados mediante una red bidireccional en toro tridimensional como se muestra en la Figura 3.3. La velocidad de comunicación entre procesadores en cualquier dirección a través del toro es de 480 MB/s.

Cada celda del T3E, como puede verse en la Figura 3.4, incluye un microprocesador DEC Alpha 21164, una memoria local de 64 MB a 2 GB, un ***router*** de comunicación y una lógica de control.

El DEC 21164 es un microprocesador RISC con una cache de primer nivel particionada para instrucciones y datos de 8 KB cada una, y una cache de segundo nivel de 96 KB, que reduce en gran medida la latencia media de los accesos a memoria. El

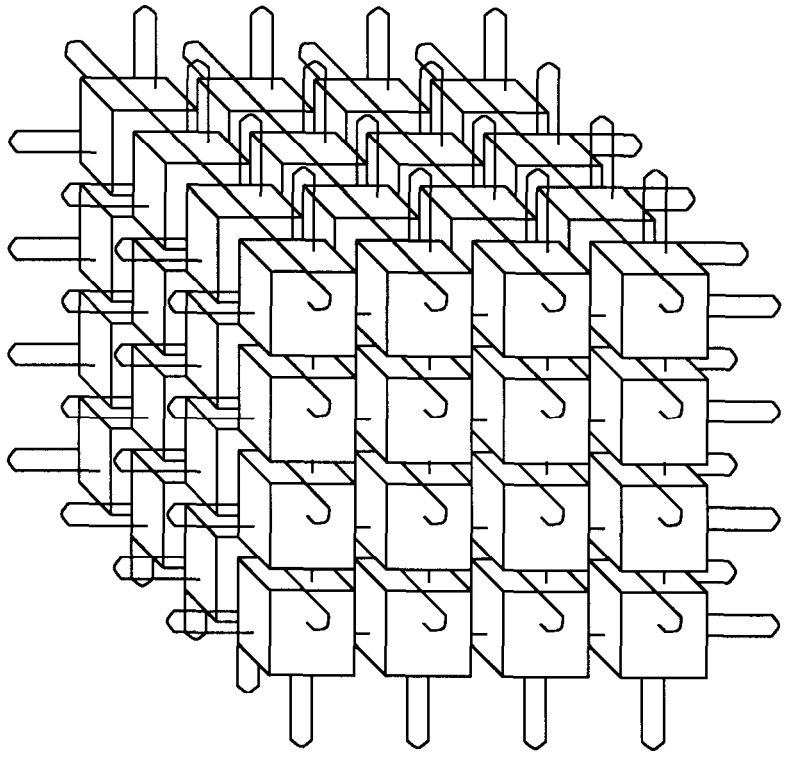

Figura 3.3: ***Red toro 3D en el Cray T3E***

T3E aumenta el *inteface* de memoria del microprocesador con un conjunto de registros externos que se utilizan como fuente o destino para las comunicaciones remotas. Todas las comunicaciones y sincronizaciones remotas se realizan entre estos registros y la memoria.

Este multiprocesador se puede programar utilizando un modelo de pase de mensajes o utilizando un modelo de programación de memoria compartida. Nosotros lo hemos programado con el modelo de programación de pase de mensajes. Más detalles sobre su arquitectura pueden ser encontrados en [97].

Las características más importantes del Fujitsu AP1000 y el Cray T3E son resumidas en la tabla 3.1.

3.3 SGI Origin 2000

El Origin 2000 de SGI se puede clasificar como un multiprocesador de memoria físicamente distribuida dotado de un sistema de coherencia cache basado en directorios, y con un acceso a memoria no uniforme. La gestión de esta coherencia es realizada

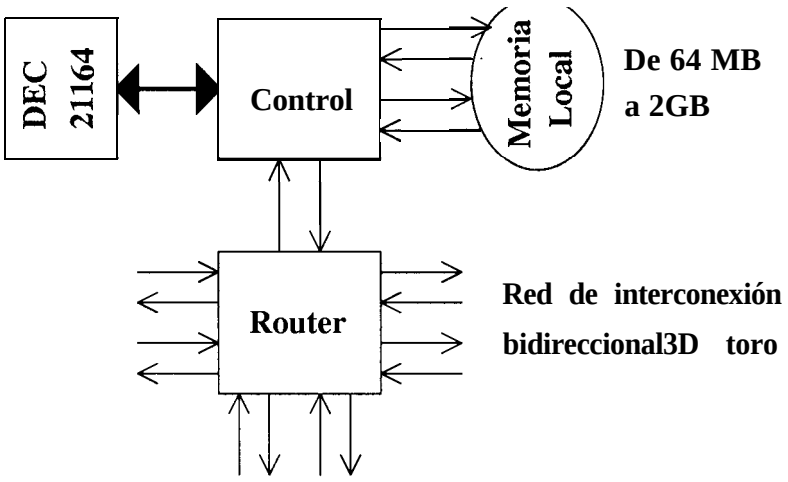

Figura 3.4: ***Diagrama de bloques de un procesador del Cray T3E***

	APlOOO	T3E
Número de PEs	4 a 1024	16 a 2048
Velocidad red de interconexión	5OMB/s (red B) 25MB/s (red T, toro 2D)	480 MB/s (toro 3D)
Procesador	SPARC	DEC Alpha 2 1164
Memoria cache	128 KB	primer nivel: 8 KB inst./datos segundo nivel: 96 k b
Memoria local	16MB	64MB a 2GB
Pico de rendimiento del procesador	8.3 MFLOPS (precisión simple) 5.6 MPLOPS (precisión doble)	600 MPLOPS

Tabla 3.1: ***Características del AP1 000 y T3E***

fundamentalmente en hardware.

La unidad estructural básica del sistema es el nodo. Un nodo está compuesto por uno o dos procesadores MIPS R10000 con sus correspondientes caches, contiene además una porción de la memoria compartida, un directorio para mantener la coherencia cache y dos *interfaces* que permiten la conexión tanto al resto de los nodos como a los dispositivos de *E/S.* Todos estos elementos se encuentran interconectados a través de un ***Hub,*** dispositivo responsable de facilitar a los procesadores y dispositivos de *E/S* el acceso transparente a todo espacio de memoria, tanto en accesos locales como remotos. El sistema soporta hasta 64 nodos con lo que se obtiene una configuración máxima de 128 procesadores y 256 GB de memoria principal.

La jerarquía de memoria cache consta de dos niveles. Localizada sobre el chip microprocesador se encuentra la cache primaria particionada en una cache para instruc-

ciones y otra para datos de 32 KB cada una, ambas asociativas de 2 vías. Fuera del chip se encuentra la memoria cache secundaria unificada de instrucciones y datos de 4 MB, también asociativa de 2 vías y que utiliza el algoritmo de reemplazo LRU *(Least Recently Used).*

Los nodos se conectan entre sí a través de *routers.* Existen distintas formas de realizar estas conexiones [100]. El sistema que nosotros hemos utilizado consta de 16 nodos conectados por medio de 4 *routers,* cada uno de los cuales se mantiene conectado con el resto y gestionan las comunicaciones de dos nodos como se muestra en la Figura *3.5.* En dicha figura las conexiones realizadas entre los *routers* con líneas punteadas son opcionales y se denominan *Xpress Links.* En un caso general, los *routers* formarán una topología hipercubo de dimensión n = log_2(Número de *routers).*

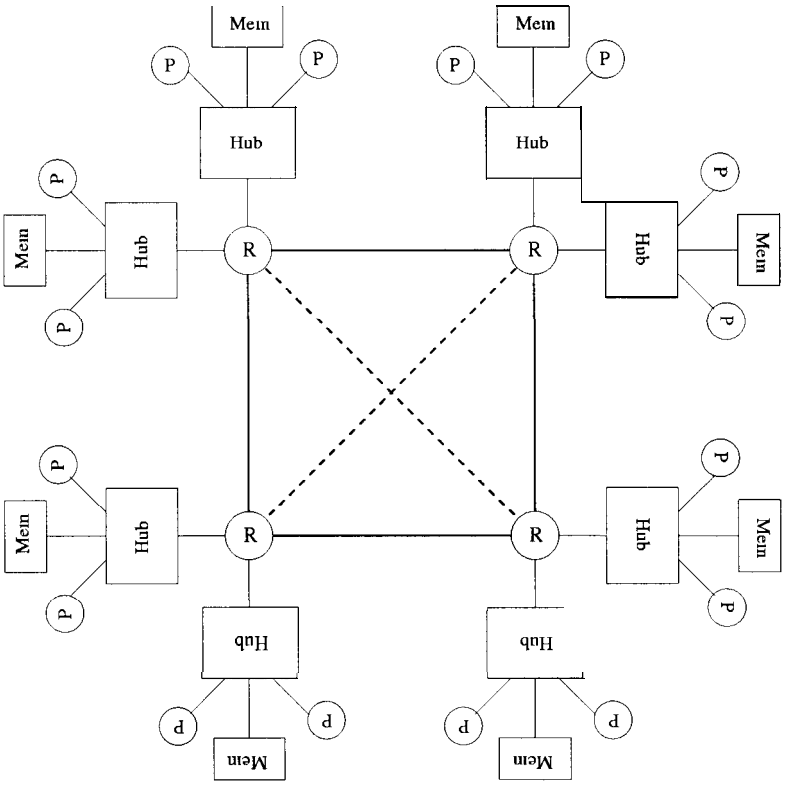

Figura 3.5: *Configuración del 02000 con 16 procesadores*

El acceso a las memorias remotas se realiza tanto a través de conexiones bidireccionales denominadas *CrayLinks* (marcadas en trazos gruesos en la figura) con un ancho de banda efectivo para datos de 624 MB/s en cada dirección, como de los *Xpress Links.* La latencia máxima en un acceso local es de 313 ns, para una conexión entre nodos que comparten un *router* (conexión directa *Hub-Hub)* aumenta hasta 497 ns y

para conexiones más lejanas aumenta aproximadamente en 100 ns por cada paso por un *router*. Una descripción más detallada de la arquitectura puede ser encontrada en [59].

3.4 Programación de los sistemas paralelos

El AP1OOO y el T3E fueron programados según el modelo de programación de pase de mensajes. El AP1OOO posee su propia librería de pase de mensajes llamada Cellos para realizar las construcciones paralelas tanto en C como en Fortran [28, 29]. Esta librería incluye instrucciones de distribución y recolección de datos *(gather/scatter),* instrucciones de reducción, de radiación, rutinas de comunicación, etc. También tiene disponible la librería estándar MPI (Message Passing *Interface),* implementada sobre esta máquina por David Sitsky [102]. En cuanto al T3E, las librerías de pase de mensajes disponibles son las librerías estándar MPI y PVM *(Parallel Virtual Machine)* [31] optimizadas por Silicon Graphics para su ejecución eficiente sobre sistemas Cray y Silicon Graphics. El AP1OOO fue programado usando Cellos y MPI, y el T3E utilizando MPI.

El 02000 fue programado según el modelo de programación de memoria compartida, utilizando para ello el paquete de macros para procesamiento paralelo denominado *purmucs.* Estas macros fueron originalmente desarrolladas en el *Argonne National Laboratory [70].* El paquete *purmucs es* de dominio público y la mayoría de las versiones C se pueden obtener desde netlib (`http://WWW.netlib.org`). implementado vía macros usando el macroproceso *m4* (estándar en la mayoría de los sistemas UNIX). El paquete de procesamiento paralelo del *Argonne National Laboratory* proporciona la abstracción de una máquina virtual la cual consta de una memoria global compartida y un número de procesadores con su propia memoria local. La ventaja de usar estas macros es la portabilidad y el control de lo procesos a un bajo nivel. Diferentes máquinas y diferentes sistemas operativos proporcionan distintas formas de llevar a cabo las construcciones generalmente necesitadas en los programas paralelos. Las macros inhiben al programador del conocimiento de la implementación específica sobre una máquina, permitiéndole concentrarse en realizar una solución correcta y eficiente a su problema. Nosotros hemos utilizado las macros desarrolladas en el departamento de Arquitectura de Computadores de la Univ. Politécnica de Cataluña [6] para su ejecución sobre el sistema operativo Irix del 02000. Los programas escritos usando *purmucs* asumen un número de tareas (procesos UNIX) que operan sobre un único espacio de direcciones compartidas. El proceso inicial o proceso padre crea un número de procesos hijo, uno por procesador. Las estructuras de sincronización son locks y *burriers.* Los *locks se* usan para realizar exclusión mutua y las *burriers* para mantener las dependencias. Dado que la creación y destrucción de procesos UNIX es costoso, los procesos son creados sólo una vez al principio del programa, hacen su trabajo y terminan al final de

la parte paralela del programa.

3.5 Herramientas para el desarrollo

Los códigos paralelos para el AP1000 eran previamente depurados utilizando CASIM [30], el cual es un simulador de la arquitectura del AP1000 que se ejecuta sobre una estación de trabajo. Por su parte, los códigos paralelos escritos para el T3E eran previamente depurados sobre un *cluster* de estaciones de trabajo sobre los que se ha instalado la librería MPI.

Para identificar los problemas de rendimiento sobre el Origin 2000, el procesador R10000 dispone de una serie de contadores implementados en el propio chip que permiten evaluar la ocurrencia de ciertos eventos en el sistema (fallos en memoria cache, operaciones en punto flotante, cargas y almacenamientos desde y a memoria . ..) [115] Una serie de herramientas *software* nos facilitan el acceso a dichos contadores:

- Perfex: Ejecuta un programa y devuelve el valor de los contadores de dos eventos seleccionados de los contadores de eventos disponibles en el R10000. Es una herramienta útil para identificar cual es el motivo que puede degradar el comportamiento del programa.
- Speedshop: Ejecuta un programa a la vez que muestrea el estado de los contadores y del *stack* del programa. Los datos obtenidos son escritos en un fichero. Es útil para localizar en que punto del programa existen problemas de rendimiento.
- Prof: Analiza el fichero generado por Speedshop mostrando los resultados de un modo amigable.
- Dprof: Muestrea un programa mientras se está ejecutando obteniendo información sobre el acceso a memoria. Es útil para identificar que estructuras de datos son las implicadas en los problemas de rendimiento.

Haciendo uso de estas herramientas encontramos las secciones del programa donde se consume la mayor parte del tiempo, de forma que se identifican la mayoría de los problemas de rendimiento en los programas paralelos.

3.6 Paralelización automática

Actualmente se están realizando grandes esfuerzos para mejorar los compiladores incluyendo técnicas complejas de forma que paralelicen el código automáticamente.

Muchas de estas técnicas para generar código paralelo optimizado son descritas en [113, 117]. Existen un determinado número de grupos de investigación desarrollando y mejorando sus herramientas de paralelización automática entre los que podemos destacar el proyecto SUIF [50] dirigido por Monica Lam, el proyecto Polaris [15] dirigido por David Padua y PFA/PCA [98, 99], herramienta comercial desarrollada por David Kuck.

SUIF (Stanford University *Intermediate Format) es* una compilador desarrollado por el *Stanford Compiler Group.* Es una infraestructura de acceso libre diseñada para soportar colaboraciones de investigación en los campos de compilación paralela y optimización de la compilación. SUIF transforma código secuencial en C o Fortran en código SPMD para máquinas de memoria compartida. El código paralelo generado incluye llamadas a una librería en tiempo de ejecución disponible actualmente para plataformas SGI, y el multiprocesador DASH.

Polaris es un compilador desarrollado en la Universidad de Illinois que transforma código secuencial en Fortran 77 en código paralelo para ser ejecutado tanto sobre máquinas de memoria compartida como sobre máquinas de memoria compartida-distribuida. Actualmente se está desarrollando una versión de Polaris para sistemas distribuidos de redes de computadores.

PFA/PCA es una herramienta comercial de paralelización automática que permite crear códigos paralelos para las plataformas de SGI (Power Challenge, Origin 200, Origin 2000, etc).

Nosotros utilizamos SUIF, Polaris y PFA para generar código paralelo automático para nuestros programas de factorización. En el Apéndice A mostramos un resumen de las salidas.

3.7 Resumen

A lo largo de esta memoria utilizaremos los sistemas paralelos AP1000 de Fujitsu, T3E de Cray y Origin 2000 de SGI para validar nuestras propuestas paralelas, con lo que abarcamos los dos tipos de arquitecturas que parecen ofrecer mayores perspectivas de futuro.

Programaremos el AP1000 y el T3E según el paradigma de pase de mensajes y el 02000 según el modelo de programación de memoria compartida utilizando construcciones paralelas de bajo nivel lo que implicará un control más detallado del paralelismo.

Elegimos el T3E y el AP1000 como representativas de los sistemas paralelos de memoria distribuida actuales por tener muy diferentes características en cuanto a la relación entre velocidad de procesamiento y ancho de banda de las comunicaciones.

Por otro lado, hemos considerado el 02000 de SGI como sistema representativo de las arquitecturas de memoria compartida físicamente distribuida.

Se ha comprobado que los reestructuradores de código automáticos actuales no son capaces de detectar suficiente paralelismo dentro de códigos tan irregulares como los que se presentan al factorizar matrices dispersas. Se hace, por tanto, necesaria su paralelización manual para obtener soluciones eficientes.

Capítulo 4

Paralelización sobre sistemas de memoria distribuida

En este capítulo estudiamos implementaciones paralelas eficientes del algoritmo de Cholesky modificado sobre sistemas de memoria distribuida. Nos centraremos en los aspectos claves de una ejecución paralela eficiente, las comunicaciones y el balanceo de la carga. El tiempo de ejecución de un algoritmo paralelo tiene normalmente dos componentes: tiempo de cálculo, el cual decrece al aumentar el número de procesadores, y tiempo de comunicación, el cual incrementa con el número de procesadores limitando la escalabilidad.

4.1 Introducción

Comenzamos el análisis del algoritmo paralelo considerando el caso general de una distribución bidimensional de la matriz, es decir, por filas y por columnas. Veremos que la situación particular en la cual cada procesador almacena columnas enteras de la matriz es la más eficiente. Es decir, obtenemos un mayor rendimiento si la matriz es solamente distribuida por columnas.

Estudiamos entonces el caso particular de una distribución unidimensional de la matriz, adaptando los algoritmos utilizados para la factorización de Cholesky estándar a la factorización de Cholesky modificada, y proponiendo la aplicación de estrategias para reducir los tiempos de espera en la ejecución de los códigos paralelos.

4.2 Distribución bidimensional de los datos

La selección de una distribución de los datos apropiada es crítico para alcanzar una ejecución eficiente sobre un sistema paralelo de memoria distribuida. Nosotros hemos optado por una distribución de la matriz utilizando un esquema de distribución cíclico por filas y por columnas conocido como distribución BCS. Hemos elegido este esquema por ser un esquema general que es aplicable a cualquier matriz independientemente de su patrón y porque garantiza un buen balanceo de la carga con un coste asociado reducido. La aplicación eficiente de esta distribución a algoritmos irregulares está avalada por numerosos trabajos, ver por ejemplo [111, 7].

Los esquemas BCS/BRS (Block Column/Row Scatter) son una generalización de la distribución cíclica para matrices dispersas. Suponiendo una red de Pf x PC procesadores, en la cual cada procesador se identifica por medio de sus coordenadas cartesianas (p_f, p_c), con $0 \leq p_f < P_f$ *y* $0 \leq p_c < P_c$, las entradas de la matriz A serán asignadas a los procesadores siguiendo la siguiente ecuación:

$$A_{ij} \longmapsto PE(imodP_f, jmodP_C), 0 \leq i, j < n$$

La matriz dispersa es distribuida sobre la red de procesadores cíclicamente por filas y por columnas como si la matriz fuese densa. El resultado es que cada procesador obtiene una submatriz dispersa que es almacenada localmente como tal utilizando el formato CCS (para la BCS) o CRS (para la BRS). Una descripción completa de estas distribuciones con sus propiedades estadísticas puede ser encontrada en [8, 90]. En la Figura 4.1 se muestra un ejemplo para una malla de 2 x 2 procesadores y la matriz dispersa de la Figura 2.1. La matriz dispersa es particionada en un conjunto de 4 submatrices que son almacenadas localmente en formato CCS dentro de cada procesador.

4.3 Algoritmo paralelo según una distribución bidimensional de los datos

Partimos del algoritmo secuencial por columnas mostrado en la Figura 2.11. En la versión densa del algoritmo, el lazo externo que recorre las columnas presenta una dependencia consistente en que para la obtención de la columna j de L *se* necesitan los valores de las columnas anteriores. Por consiguiente, en el caso denso, este lazo externo es necesariamente secuencial.

Analizando el algoritmo para el caso de que la matriz a factor-izar sea dispersa, se puede observar que la dependencia del lazo externo puede desaparecer parcialmente ya

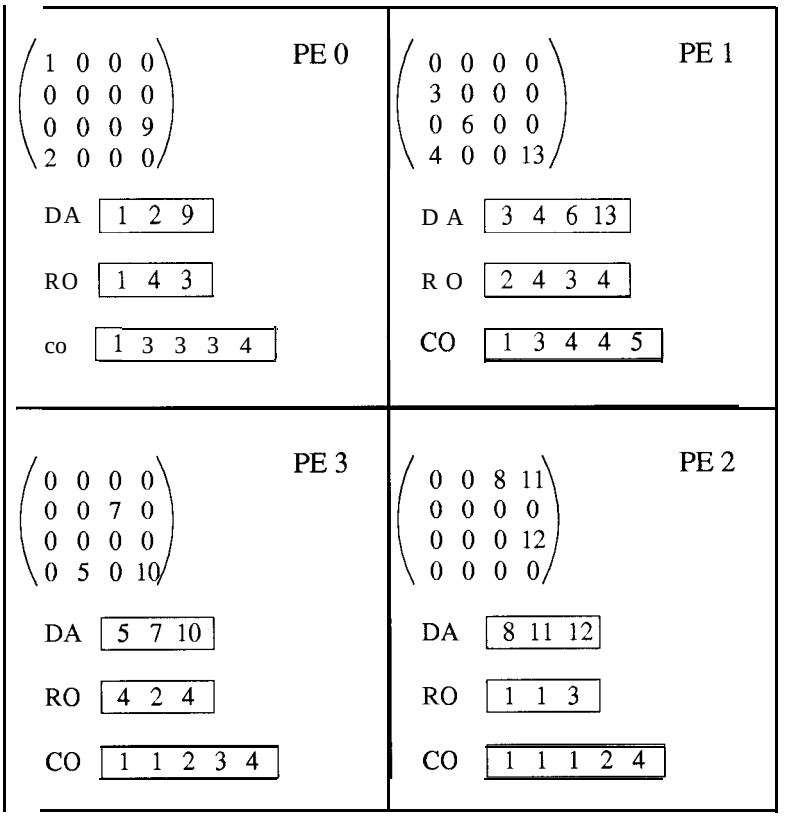

Figura 4.1: Distribución BCS para una malla de 2 x 2 procesadores

que, la columna j de L puede no necesitar ser modificada por todas las columnas $k < j$. Específicamente, la columna j es modificada solamente por aquellas columnas k para las cuales $l_{jk} \neq 0$.

Esta situación ventajosa para matrices dispersas se explota en la versión paralela que presentamos y constituye la idea clave en la que se ha centrado nuestra estrategia de paralelización, en la cual no sólo se paralelizan los lazos internos como en el caso denso, sino también el lazo que recorre las columnas en los casos indicados anteriormente.

El algoritmo paralelo consta de dos etapas. La primera de ellas consiste en un preprocesamiento, dependiente de la estructura del factor de Cholesky L, y en ella se realizan cálculos previos necesarios para la factorización propiamente dicha que será realizada en la segunda etapa. Estos cálculos tienen como objetivo que cada procesador conozca el patrón de las comunicaciones en las que va a estar implicado, es decir, a que procesador tiene que enviar cada mensaje y de cual tiene que recibirlos. Empezaremos describiendo la segunda etapa del algoritmo, pues a partir de ella resulta inmediato comprender los cálculos necesarios de la primera etapa.

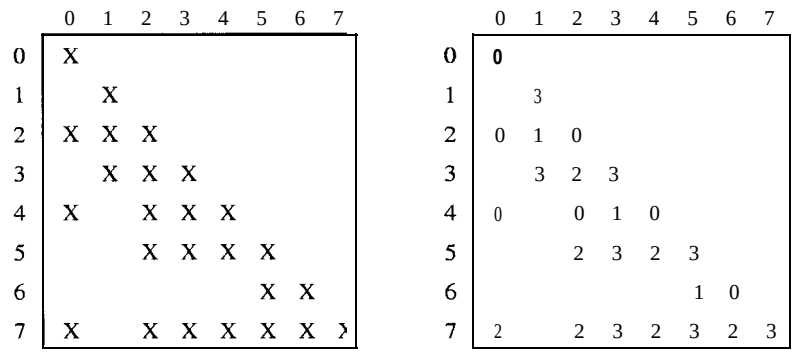

Figura 4.2: *Matriz ejemplo a factorizar*

4.3.1 Factorización

Para explicar la factorización de la matriz en paralelo nos basamos en la matriz ejemplo de la Figura 4.2 y en el algoritmo secuencial de la Figura 2.11. En la Figura 4.2 se muestra el patrón de una matriz 8 x 8 sobre la cual ya se ha incluido el fill-in y su distribución sobre una red de 4 procesadores indicando a que procesador (de 0 a 3) pertenece cada entrada. Se asume que los 4 procesadores forman una malla de 2 filas y 2 columnas, donde p = $p_f * 2 + p_c$ es el procesador que se encuentra en la fila p_f ($0 \leq p_f < 2$) y la columna p_c ($0 \leq p_c < 2$) de la malla. En la Figura 4.3 se muestra la evolución del algoritmo para dicha matriz, en concreto se muestran los pasos a seguir para el cálculo de la segunda columna de L, el símbolo "+" representa valores finales de L. El algoritmo paralelo tendrá la siguiente estructura, donde n_{local} es el número de columnas locales sobre cada procesador:

FOR j = 0 TO n_{local} - I D O

1. Modificar la columna j de A (ver Figura 4.3a): Para cada entrada se reciben los elementos de las columnas previas para calcular el sumatorio: $a_{kj} = a_{kj} - l_{js}a_{ks}$, con $0 \leq s < j$ y $j < k < N$. En el ejemplo de la Figura 4.3, para calcular la columna 2 el procesador 2 tendrá que recibir el producto resultante de multiplicar a_{21} por a_{31}. Este producto es enviado por el procesador 3 que lo calcula en su paso 7. Además el procesador 2 tendrá que recibir la entrada l_{20} desde el procesador 0 para computar localmente el producto l_{20} por a_{70}.

2. Computar la contribución local al cálculo del parámetro θ: Cada procesador calcula el parámetro $\theta_j = max_{s \in \{j+1 \cdots n\}}\{||a_{sj}||\}$. Por ejemplo, el

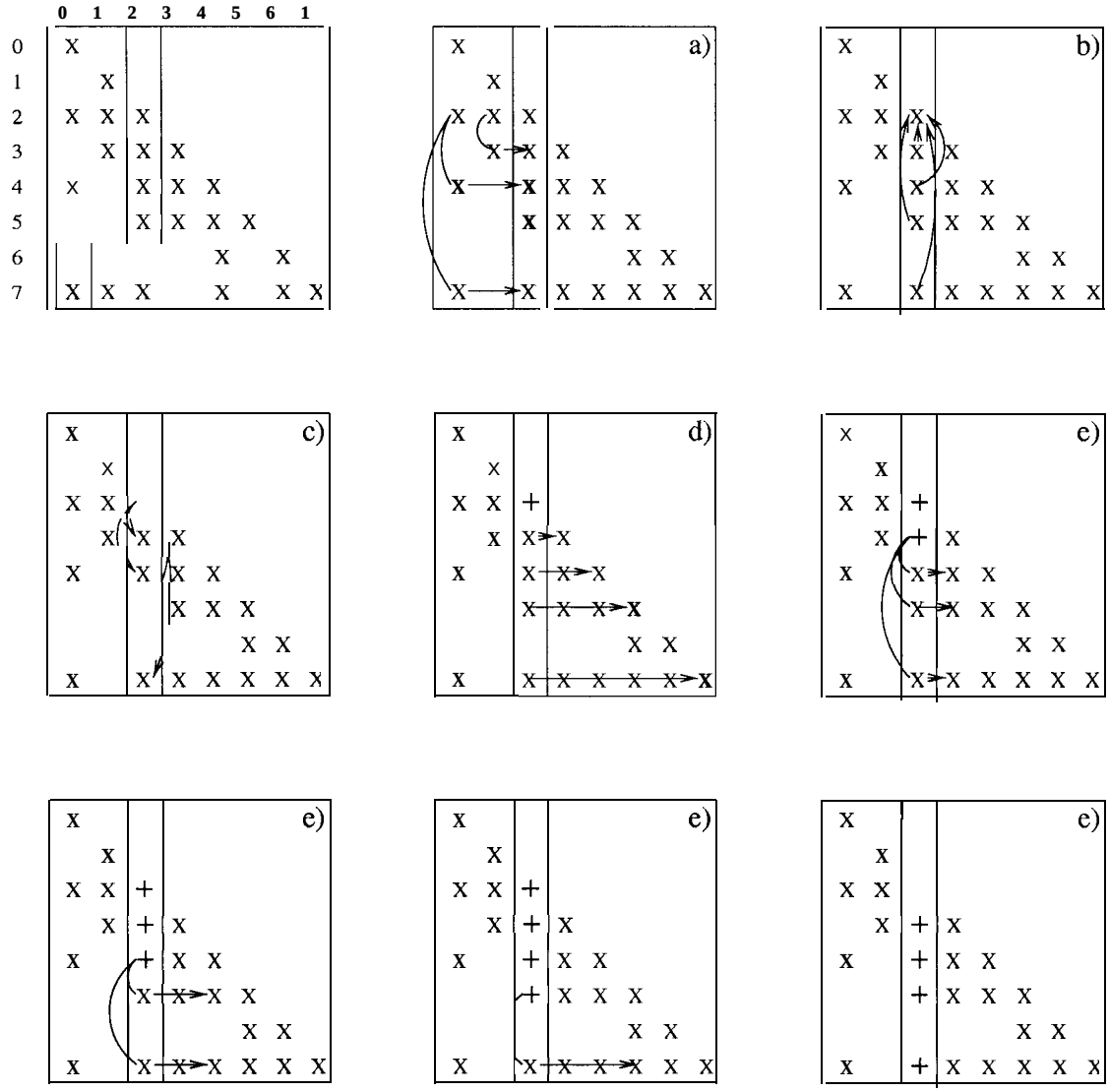

Figura 4.3: ***Evolución del algoritmo para la matriz ejemplo de la Figura 4.2***

procesador 2 calcula esta contribución a partir de los valores de a_{32}, a_{52} y a_{72} en el ejemplo de la Figura 4.3.

3. Si el procesador actual no contiene la diagonal se envía θ al procesador que la almacena para calcular el valor global (ver Figura 4.3b). Sobre el ejemplo, el procesador 2 envía θ_2 al procesador 0 para el cálculo de θ_2 final.

4. Si el procesador contiene el elemento de la diagonal, se comprueba si ya se han recibido todas las contribuciones locales al cálculo de θ. En este caso se calcula el elemento d_j ($d_j = max\{\gamma, \|a_{jj}\|, \theta_j^2/\beta^2\}$) y se envía a todos los procesadores que almacenan entradas en esa columna (ver Figura 4.3c). Sobre el ejemplo, el procesador 0 calcula d_2 y lo envía al procesador 2.

5. Esperar por el elemento de la diagonal: Se debe disponer del valor final de la diagonal para computar la columna j de L y modificar columnas posteriores. Continuando con el ejemplo, el procesador 2 tiene que esperar a recibir la diagonal desde el procesador 0.

6. Modificar los elementos diagonales de columnas posteriores de acuerdo con la ecuación: $a_{ss} = a_{ss} - a_{sj}^2/d_j$, con $j < s <$ N (ver Figura 4.3d). En el ejemplo, el procesador 2 envía a_{32}, a_{52} y a_{72} al procesador 3 para la actualización de a_{33}, a_{55} y a_{77} respectivamente.

7. Normalizar la columna j de L ($l_{sj} = a_{sj}/d_j, j < s < N$) a la vez que se van realizando los productos locales, enviándolos para la modificación de columnas posteriores, ver Figuras 4.3e. Una vez modificada a_{32} en el ejemplo, se realizan los productos con a_{52} y a_{72} sin modificar, productos que serán enviados al procesador 3 para modificar la tercera columna. Cada l_{sj} computado se envía a todos los procesadores que tengan entradas en filas posteriores dentro de la misma columna para el cálculo de los productos que modificarán columnas posteriores. Siguiendo con el ejemplo, a la vez que se realizan los productos, se envía l_{32} al procesador 0 para multiplicarlo con a_{42} y enviar el producto al procesador 1 para modificar la tercera columna.

4.3.1.1 Características del código paralelo

Para aprovechar el paralelismo sobre el lazo más externo, el algoritmo se implementa de forma asíncrona, minimizando además los tiempos en los que el procesador está

esperando un mensaje. Cada procesador realiza los envíos lo antes posible, por tanto el comportamiento del algoritmo vendrá regido por las recepciones. Las recepciones se producen solamente cuando cada procesador necesita el correspondiente dato, entonces, dicho procesador entra en un lazo de espera hasta que se produce la recepción. Mientras espera, lee todos los mensajes que'eventualmente pueda recibir, almacenándolos temporalmente en ***buffers, o,*** si ello es posible, adelanta computaciones que teóricamente se ejecutarían con posterioridad. En esencia, se sigue una estrategia *dataflow* evitando cualquier tipo de comunicación colectiva (radiaciones o reducciones) siempre que implique la presencia de algún punto de sincronización. Se pretende con ello reducir los tiempos de espera al mínimo posible marcado por el camino crítico.

Así por ejemplo, el paso 1 implica entrar en un lazo de espera hasta que se reciban todas las modificaciones que se realizan a esa columna desde columnas previas. Mientras espera, puede recibir todos los mensajes que le llegan o que ya están en cola. Puede ocurrir que sea, por ejemplo, un parámetro θ local de una columna k en la cual este procesador es diagonal. En ese caso comprueba si ya se han recibido todos los θ locales y si ya se han realizado todas las modificaciones a a_{kk}. En caso afirmativo calcula d_k y lo envía a todos los procesadores con entradas en esa columna, con lo cual está adelantando computaciones. También podría ocurrir que recibiese un valor l_{sk} para realizar productos con los a_{lk} que verifiquen $l > s$, y que la columna k aún no haya sido totalmente modificada. En ese caso los l_{sk} recibidos se almacenan en un ***buffer.*** Después de la modificación de cada columna, es decir, después del paso 1 del algoritmo, se comprueba si hay algún elemento en el ***buffer*** correspondiente a esa columna y en caso afirmativo se realizan los productos y se envían.

Para poder distinguir los diferentes tipos de mensaje, los mensajes son acompañados de una etiqueta que identifica de que mensaje se trata y que fila o columna va a modificar. Para el caso de la actualización de una columna (paso 1 del algoritmo paralelo) se debe de identificar, además, a que entrada dentro de la columna va a modificar. En este caso se envía el dato acompañado de un entero que especifica la fila a la que modifica. Lo mismo ocurre cuando se envía verticalmente el valor final de L (paso 3c del algoritmo), se tendrá que enviar el dato y un entero que especifique la fila a la que pertenece dicho dato para identificar con que datos se han de realizar los productos.

4.3.1.2 **Empaquetamiento de mensajes**

Dada una entrada en una columna, todos los productos resultantes de multiplicar esta entrada por los elementos que hay en filas posteriores modificarán la misma columna, y puesto que nuestra distibución es cíclica por filas y por columnas, sólo podrán modificar filas de esa columna pertenecientes, todas ellas, al mismo procesador. Por tanto, todos estos mensajes pueden ser empaquetados y enviados como un único mensaje consigu-

iendo una importante reducción en el número de comunicaciones.

Según el esquema planteado, cada vez que se modifica una columna se envían las entradas de dicha columna a los procesadores que contienen el elemento diagonal para modificarlo. Sin embargo, todas las modificaciones hechas por el mismo procesador se pueden acumular en dicho procesador enviando solamente la modificación total. De esta forma, se consigue reducir el número de envíos y de datos enviados. Para poder llevar a cabo este planteamiento se necesita incluir un conjunto de *buffers* que almacenen la suma de estas modificaciones. Se utiliza un *buffer* por cada elemento diagonal. También se necesita incluir un contador que indique cuando se han acabado de realizar las sumas locales.

4.3.2 Preprocesamiento

Para poder llevar a cabo el planteamiento anterior, cada procesador tiene que conocer a priori a donde enviar los mensajes y cuantos mensajes debe recibir. Esto es lo que se determina en la etapa de preprocesamiento. En concreto se computa:

- El patrón de comunicaciones para la modificación de la columna j-ésima: Se recorre localmente para cada entrada y se determina el número de elementos (productos) que va a recibir de columnas previas.
- El patrón de comunicaciones para el envío de la columna j-ésima de L: Para cada entrada k de la columna se necesita conocer cuales son los procesadores que tienen entradas por debajo en esa columna.
- El patrón de comunicaciones para la modificación de las diagonales: Se tiene que contar el número de modificaciones que se van a realizar.
- El patrón de comunicaciones para el cálculo de θ: Se necesita conocer cuantos procesadores contribuyen al parámetro θ_j. Serán todos aquellos con entradas en la columna j-ésima.
- El patrón de comunicaciones para el envío vertical de d_j: Se necesita conocer los procesadores a los que enviar la diagonal, serán todos aquellos procesadores con entradas en la columna j-ésima.

Todos estos cálculos se podrían llevar a cabo a la vez que se realiza la factorización simbólica. Nosotros hemos implementado esta etapa en paralelo, de forma que en cada procesador se calculan los parámetros propios que necesita.

Según la estrategia de paralelización descrita habrá tres factores determinantes en la eficiencia del código paralelo: el número total de comunicaciones necesarias para

llevar a cabo la factorización, el balanceo de la carga y los tiempos de espera de los procesadores. En las siguientes secciones se caracteriza cada uno de ellos.

4.4 Análisis de las comunicaciones para la distribución bidimensional

Analizamos en esta sección las comunicaciones presentes en el algoritmo paralelo y determinamos cual es la configuración de la red de procesadores que proporciona un menor número de comunicaciones. Suponemos para esta análisis que se realiza el empaquetamiento de mensajes del que hemos hablado en la sección anterior. Un estudio de las comunicaciones de una versión previa del algoritmo en la cual no se realizaba este empaquetamiento puede ser encontrado en [76].

El algoritmo que proponemos tiene como una de sus mayores ventajas el que las comunicaciones son muy regulares y que se pueden adecuar eficientemente a topologías tipo malla, ya que solamente se pueden dar dos tipos de mensajes:

1. Horizontales o a lo largo de las filas:
   - Para la modificación de la columna
   - Para la modificación de las diagonales
2. Verticales o a lo largo de las columnas:
   - Para el cálculo de θ
   - Para la normalización de L
   - Para el cálculo de los productos necesarios en columnas posteriores

El número de mensajes depende del patrón de la matriz y de la configuración de la red de procesadores. Así mismo, el comportamiento será diferente para procesadores que contienen diagonales y procesadores que no las contienen.

Se pueden evaluar el número de mensajes totales que envía cada procesador horizontal y verticalmente por cada fila o columna local. Se asume para esta discusión que los P procesadores forman una malla de P_f filas y P_c columnas, donde $p = p_f * P_c + p_c$ *es* el procesador que se encuentra en la fila p_f $(0 \leq p_f < P_f)$ *y* la columna p_c $(0 \leq p_c < P_c)$ de la malla. Utilizaremos mayúsculas para definir variables globales y minúsculas para variables locales. Definimos los siguientes parámetros:

- d_j^p, nos indica si el procesador p contiene la diagonal de la columna j. Si $d_j^p = 0$, ***p es*** el procesador que contiene la diagonal y que denominaremos procesador diagonal por simplicidad, si no, d_j^p=1.

- $d_i^{p'}$, nos indica si el procesador p contiene la diagonal de la fila i. Si $d_i^{p'} = 0$, ***p es*** el procesador diagonal, si no, $d_i^{p'}$=1.

- n_j^p es el número de entradas locales en la j-ésima columna del procesador p, siendo $J = j * P_c + p_c$ el índice global asociado a la columna j-ésima. Sobre el ejemplo de la Figura 4.2, $n_1^3 = 3$, en concreto, serían las entradas a_{33}, a_{53} y a_{73}.

- m_i^p es el número de entradas locales en la i-ésima fila del procesador p, siendo $I = \mathrm{i} * P_f + p_f$ el índice global asociado a la fila i-ésima. En el ejemplo, $m_3^3 = 3$, en concreto, serían las entradas a_{73}, a_{75} y a_{77}.

- $col^p = \lceil \frac{N}{P_c} \rceil$ es el número de columnas locales en el procesador p.

- Dada la k-ésima entrada de la columna j del procesador ***p se*** define:

  - P_{jk}^p como el número de procesadores distintos de p que tienen entradas con un índice fila S superior al índice fila I de dicha entrada k-ésima, con $I = \boldsymbol{i} * P_f + p_f$, donde i es la fila local de la k-ésima entrada. Sobre el ejemplo, $P_{10}^2 = 1$, en concreto, sólo el procesador 0 tiene una entrada con un índice fila global superior, la entrada a_{42}.
  - M_{jI}^p como el número de entradas en la columna $J = j * P_c + p_c$ con índice fila $S < I$ y $S \neq J$. En el ejemplo, $M_{17}^3 = 2$, en concreto las entradas a_{43} y a_{53}.
  - $M_{jI}^{'p}$ como el número de entradas en la columna $J = j * P_c + p_c$ con índice fila $S < I$, $S \neq J$ y que verifican $(S - p_c)\%P_c \neq 0$, ***es*** decir, la columna S no pertenece al procesador p. Continuando con el mismo ejemplo, $M_{17}^{'3} = 1$, ya que sólo la entrada a_{43} no pertenece al procesador 3.

Utilizando estos parámetros se puede hacer un recuento del número de mensajes horizontales y verticales.

Calculamos, además, los límites superiores e inferiores a este número de mensajes, así como una estimación en un caso promedio. El límite superior vendrá dado por el caso en el que la matriz sea densa. Para el cálculo del valor promedio consideramos que es el definido por la siguiente situación:

- El número de entradas por fila y columna está totalmente equilibrado, con lo que $\frac{\alpha}{N}$ es el número promedio de entradas por cada fila o columna de la matriz, siendo α el número de entradas totales en la matriz.

- Así mismo, consideramos que el balanceo de la carga entre procesadores está también totalmente equilibrado, con lo que $\frac{\alpha}{NP_f}$ es el número de entradas en cada columna dentro de cada procesador y $\frac{\alpha}{NP_c}$ es el número de entradas en cada fila dentro de cada procesador.

Proposición 1 *El* ***número total de mensajes verticales generados por la columna*** *j-ésima* ***de un procesador p será:***

$$MV_j^p = min\{d_j^p,\ 1,\ n_j^p\} + \sum_{k=0}^{n_j^p-1} P_{jk}^p$$

cumpliéndose que:

$$0 \leq MV_j^p \leq 1 + (P_f - 1)\frac{N}{P_f}$$

y siendo su valor promedio:

$$\overline{MV_j^p} = min\{d_j^p, 1\} + \frac{P_f - 1}{2P_f}\left(\frac{2\alpha}{N} - P_f\right) \quad \text{si} \quad \frac{\alpha}{N} \geq P_f$$

$$\overline{MV_j^p} = min\{d_j^p, \frac{\frac{\alpha}{N} - 1}{P_f - 1}\} + \frac{\left(\frac{\alpha}{N} - 1\right)\frac{\alpha}{N}}{2P_f} \quad \text{si} \quad \frac{\alpha}{N} < P_f$$

Demostración:

- Por un lado, un procesador p no diagonal debe enviar 1 mensaje para el cálculo de θ al procesador diagonal, siempre que $n_j^p > 0$.

- Por otro lado, el procesador diagonal debe enviar la diagonal a cada procesador que tenga entradas en la j-ésima columna local para su normalización, y cada entrada no diagonal k de la columna j debe ser radiada a todos los procesadores con entradas en filas superiores para el cálculo de los productos. El número de estos mensajes será, por tanto, $\sum_{k=0}^{n_j^p-1} P_{jk}^p$.

En el caso denso (límite superior):

- Un procesador ***p*** no diagonal debe enviar 1 mensaje para el cálculo de θ al procesador diagonal.

- Todas las entradas deben ser enviadas a todos los procesadores con entradas posteriores. Como estamos suponiendo una distribución cíclica, en el peor caso (caso en el cual el número de entradas es múltiplo de P_f) y considerando el procesador con $Pf = 0$, todos los procesadores con $Pf > 0$ tendrán entradas posteriores y, por tanto, todas tendrán que ser radiadas a todos los procesadores. El número de estos mensajes será por tanto: $\frac{N}{P_f}(P_f - 1)$.

En cuanto al valor promedio:

- Por un lado, los procesadores no diagonales envían un mensaje al procesador diagonal para el cálculo de θ. Este mensaje se envía siempre y cuando el procesador contenga entradas en esa columna.
  - Si $\frac{\alpha}{N} \geq Pf$, todos los procesadores contienen entradas en esa columna y, por tanto, todos los procesadores no diagonales envían un mensaje.
  - Si $\frac{\alpha}{N} < Pf$, no todos los procesadores contienen entradas en esa columna, y sólo enviarán un mensaje los $\frac{\alpha}{N} - 1$ procesadores no diagonales con entrada en esa columna. Nótese que existen $\frac{\alpha}{N} - 1$ procesadores con entradas ya que el reparto es equilibrado. En promedio, el procesador p no diagonal enviará $\frac{\frac{\alpha}{N}-1}{P_f-1}$ mensajes.
- Además, cada entrada en cada columna, incluida la diagonal, debe de ser radiada a todos los procesadores que tengan entradas por debajo.
  - Si $\frac{\alpha}{N} > Pf$, las últimas Pf entradas pertenecerán cada una a un procesador y por tanto las primeras $\frac{\alpha}{N} - Pf$ tendrán que ser enviadas a todos los demás procesadores, pues todos ellos tienen alguna entrada por debajo. A partir de ahí, el número de procesadores que contienen entradas por debajo de la considerada va disminuyendo en una unidad para cada entrada. Por tanto, el número de mensajes enviados en promedio y por cada entrada será:

$$\frac{(\frac{\alpha}{N} - Pf)(Pf - 1) + \sum_{i=1}^{P_f-1} i}{\frac{\alpha}{N}}$$

    Además, como el número de entradas que tiene cada procesador por cada columna es $\frac{\alpha}{NP_f}$, tendremos que:

$$\overline{MV_j^p} = min\{d_j^p, 1\} + \frac{(\frac{\alpha}{N} - P_f)(P_f - 1) + \sum_{i=1}^{P_f-1} i}{Pf}$$

$$= min\{d_j^p, 1\} + \frac{(\frac{\alpha}{N} - P_f)(P_f - 1) + \frac{P_f(P_f-1)}{2}}{Pf}$$
$$= min\{d_j^p, 1) + \frac{P_f - 1}{2P_f}(\frac{2\alpha}{N} - P_f).$$

- Si $\frac{\alpha}{N} < Pf$, y suponiendo de nuevo que las entradas están uniformemente repartidas, cada una de ellas pertenecerá a un procesador distinto, y por tanto la primera entrada tendrá que ser radiada a $\frac{\alpha}{N} - 1$ procesadores, a partir de ahí, el número de procesadores con entradas por debajo disminuye en una unidad. Por tanto, el número de mensajes enviados en promedio por cada entrada será:

$$\frac{\sum_{i=1}^{\frac{\alpha}{N}-1} i}{\frac{\alpha}{N}}$$

Además, como el número de entradas que tiene cada procesador por cada columna es $\frac{\alpha}{NP_f}$, tendremos que:

$$\overline{MV_j^p} = min\{d_j^P, \frac{\frac{\alpha}{N} - 1}{P_f - 1}\} + \frac{\sum_{i=1}^{\frac{\alpha}{N}-1} i}{P_f}$$
$$= min\{d_j^P, \frac{\frac{L}{N}L - 1}{f - ,>}\} + \frac{(\frac{\alpha}{N} - 1)\frac{\alpha}{N}}{2Pf}. \quad \square$$

Proposición 2 *El número total de mensajes horizontales generados por la columna j-ésima de un procesador p será:*

$$MH_j^p = M_{jI'}^{'p}$$

donde I' es la fila global de la última entrada de la columna j-ésima del procesador p.

y por la fila local i:

$$MH_i^p = min\{d_i^{p'}, 1, m_i^p\}$$

cumpliéndose que:

$$0 \leq MH_j^p \leq (N-2)\frac{P_c - 1}{P_c}$$

y siendo el valorpromedio:

$$\overline{MH_j^p} = (\frac{\alpha}{N} - \frac{P_f}{2} - \frac{3}{2})(\frac{P_c - 1}{P_c}) \quad \text{si} \quad \frac{\alpha}{N} \geq P_f$$
$$\overline{MH_j^p} = \frac{(\frac{\alpha}{N} - 1)(\frac{\alpha}{N} - 2)}{2P_f}(\frac{P_c - 1}{P_c}) \quad \text{si} \quad \frac{\alpha}{N} < P_f$$

y, además, por la *fila local i:*

$$\overline{MH_i^p} = min\{d_i^{p'}, 1, \frac{\frac{\alpha}{N} - 1}{P_c - 1}\}$$

Demostración:

- Para cada entrada global K en la columna J con índice global fila *1* se realiza un producto con todas las entradas correspondientes a la columna local j del procesador p con índices globales fila S > I. Estos productos se empaquetan y se envían al procesador que contiene la columna global I y que está en la misma fila que el procesador p para la modificación de dicha columna. Por tanto, si consideramos que I' es la fila global de la última entrada de la columna j-ésima del procesador p, el número de mensajes enviado será el número de entradas en la columna J con índice global fila $I < I'$, es decir, $M_{jI'}^p$. De todos ellos habrá que excluir los enviados a sí mismo, por tanto:

$$MH_j^p = M_{jI'}^{'p}.$$

- Además, por cada procesador p no diagonal y por cada fila local i con entradas no nulas ($m_i^p \neq 0$) se envía un mensaje al procesador que contiene la diagonal de la fila considerada para su modificación:

$$MH_i^p = min\{d_i^{p'}, 1, m_i^p\}.$$

En el caso denso:

- El peor caso se corresponde con la entrada de la última fila correspondiente a la primera columna, la cual tiene que ser multiplicada por todas las entradas en filas inferiores (excluyendo la diagonal) y cada uno de estos productos es enviado al procesador correspondiente. De todos estos mensajes habrá que excluir los que van destinados a sí mismo. Por tanto:

$$0 \leq MH_j^p \leq (N-2) - \frac{N-2}{P_c} = (N-2)\frac{P_c - 1}{P_c}.$$

En cuanto al valor promedio:

- Si $\frac{\alpha}{N} \geq P_f$, todos los procesadores tienen entradas en esa columna. La última entrada de cada procesador tendrá que ser multiplicada por todas las anteriores. Así,

el procesador que contiene la última entrada enviará $\frac{\alpha}{N} - 2$ mensajes, es decir, el resultado de multiplicar la última entrada por todas las entradas en filas inferiores excluyendo la diagonal. El procesador que contiene la penúltima entrada enviará $\frac{\alpha}{N} - 3$ mensajes, y así sucesivamente. De todos estos mensajes habrá que descontar los enviados a sí mismo, por tanto, el número total de mensajes generados por la columna j será:

$$\sum_{i=0}^{P_f-1}(\frac{\alpha}{N} - 2 - i)(\frac{P_c - 1}{P_c}) = \frac{(\frac{2\alpha}{N} - P_f - 3)P_f}{2}(\frac{P_c - 1}{P_c}).$$

Entonces, al procesador p le corresponderán:

$$\overline{MH_j^p} = \frac{(\frac{2\alpha}{N} - P_f - 3)}{2}(\frac{P_c - 1}{P_c}) = (\frac{\alpha}{N} - \frac{P_f}{2} - \frac{3}{2})(\frac{P_c - 1}{P_c}).$$

- Si $\frac{\alpha}{N} < P_f$, no todos los procesadores tienen entradas en esa columna. Sólo los $\frac{\alpha}{N} - 2$ procesadores con entradas en esa columna distinta de la diagonal y de la primera entrada por debajo de la diagonal enviarán mensajes. Así, el procesador que contiene la última entrada enviará $\frac{\alpha}{N} - 2$ mensajes, es decir, el resultado de multiplicar la última entrada por todas las entradas en filas inferiores excluyendo la diagonal. El procesador que contiene la penúltima entrada enviará $\frac{\alpha}{N} - 3$ mensajes, y así sucesivamente. De todos estos mensajes habrá que descontar los enviados a sí mismo, por tanto, el número total de mensajes generados por la columna j será:

$$\sum_{i=0}^{\frac{\alpha}{N}-3}(\frac{\alpha}{N} - 2 - i)(\frac{P_c - 1}{P_c}) = \frac{(\frac{\alpha}{N} - 1)(\frac{\alpha}{N} - 2)}{2}(\frac{P_c - 1}{P_c}).$$

Entonces, al procesador p le corresponderán en promedio:

$$\overline{MH_j^p} = \frac{(\frac{\alpha}{N} - 1)(\frac{\alpha}{N} - 2)}{2P_f}(\frac{P_c - 1}{P_c}).$$

- En cuanto al valor promedio por fila, un procesador p no diagonal enviará un mensaje al procesador que contiene el elemento diagonal de la fila i si el procesador p tiene entradas en dicha fila.
  - Si $\frac{\alpha}{N} \geq P_c$ todos los procesadores tienen entradas en esa fila, y por tanto, $\overline{MH_i^p} = min\{d_i^{p'}, 1\}$.
  - Por el contrario, si $\frac{\alpha}{N} < P_c$, sólo habrá $\frac{\alpha}{N}$ procesadores con entradas en esa fila, siendo uno de ellos el procesador diagonal, por tanto, en promedio, un procesador p no diagonal enviará $\frac{\frac{\alpha}{N}-1}{P_c-1}$ mensajes. Entonces: $\overline{MH_i^p} = min\{d_i^{p'}, \frac{\frac{\alpha}{N}-1}{P_c-1}\}$.

Por tanto:

$$\overline{MH_i^p} = min\{d_i^{p'}, 1, \frac{\frac{\alpha}{N} - 1}{P_c - 1}\}. \quad \square$$

Proposición 3 *El número total de mensajes verticales generados será:*

$$MV = \sum_{J=O}^{N-1} \sum_{p=0}^{P-1} min\{d_j^p, 1, n_j^p\} + \sum_{p=o}^{P-1} \sum_{j=0}^{col^p} \sum_{k=O}^{n_j^p - 1} P_{jk}^p$$

cumpliéndose que:

$$0 \leq MV \leq (P_f - 1)(N - \frac{P_f}{2}) + (P_f - 1)(N - P_f)(\frac{N+1}{2}) + P_f^2(P_f - 1)\frac{2 - P_f}{3}$$

y siendo el valor promedio:

$$\overline{MV} = (P_f - 1)(N + \alpha - \frac{NP_f}{2}) \quad \text{si} \quad \frac{\alpha}{N} \geq P_f$$
$$\overline{MV} = (\frac{\alpha}{N} - 1)(\frac{\alpha}{2} + N) \quad \text{si} \quad \frac{\alpha}{N} < P_f$$

Demostración:

- Por un lado, por cada columna, cada procesador con entradas en esa columna, y que no contenga el elemento diagonal, envía un mensaje al procesador que contiene la diagonal, esto es, por tanto:

$$\sum_{J=0}^{N-1} \sum_{p=o}^{P-1} min\{d_j^p, 1, n_j^p\}.$$

- Por otro lado, cada procesador envía $\sum_{k=0}^{n_j^p - 1} P_{jk}^p$ mensajes por cada columna j local.

En el caso denso:

- El número de mensajes enviados para el cálculo de θ se corresponderá con el número de procesadores no diagonales con entradas en esa columna. En las primeras $N - P_f$ columnas, todos los procesadores tendrán entradas, a partir de

ahí el número de procesadores con entradas en la columna considerada va disminuyendo en una unidad, por tanto:

$$(N - P_f)(P_f - 1) + \sum_{i=0}^{P_f-1} i = (P_f - 1)(N - \frac{P_f}{2}).$$

- Además, cada entrada en cada columna será radiada a todos los procesadores con entradas posteriores. Por tanto, en el caso denso, en la primera columna, las primeras N $-P_f$ entradas serán radiadas a todos los demás procesadores. Las últimas P_f corresponderán cada una a un procesador distinto y, por tanto, habrá que descontar un procesador a medida que avanzamos en el número de entradas. Entonces, para la primera columna el número de mensajes será:

$$(N - P_f)(P_f - 1) + \sum_{i=0}^{P_f-1} i.$$

Como el número de entradas por columna varía entre N y 1, para las primeras N $-P_f$ columnas el número de mensajes será:

$$\sum_{i=0}^{N-P_f-1} (N - i - P_f)(P_f - 1) + (\mathrm{N} - P_f) \sum_{\mathrm{i=O}}^{P_f-1} \mathrm{i}.$$

Mientras que, para las últimas P_f columnas:

$$\sum_{i=1}^{P_f} \sum_{j=0}^{\mathrm{i}-1} j = \sum_{i=1}^{P_f-1} (P_f - \mathrm{i})\mathrm{i}.$$

El número total será, por tanto:

$$(P_f - 1) \sum_{i=0}^{N-P_f-1} (N - i - P_f) + (N - P_f) \sum_{i=0}^{P_f-1} i + \sum_{i=1}^{P_f-1} (P_f - i)i =$$

$$(P_f - 1)\frac{(N - P_f)(N - P_f + 1)}{2} + (N - P_f)\frac{(P_f - 1)P_f}{2} + \sum_{i=0}^{P_f-1} (P_f - i)i =$$

$$(P_f - 1)(N - P_f)(\frac{N+1}{2}) + P_f \sum_{i=1}^{P_f-1} i - P_f \sum_{i=1}^{P_f-1} i^2 =$$

$$(P_f - 1)(N - P_f)(\frac{N+1}{2}) + P_f\frac{(P_f - 1)P_f}{2} - P_f\frac{P_f(P_f - 1)(2P_f - 1)}{6} =$$

$$(P_f - 1)(N - P_f)(\frac{N+1}{2}) + P_f^2(P_f - 1)\frac{2 - P_f}{3}.$$

- Sumando ambas contribuciones:

$$0 \leq MV \leq (P_f - 1)(N - \frac{P_f}{2}) + (P_f - 1)(N - P_f)(\frac{N+1}{2}) + P_f^2(P_f - 1)\frac{2 - P_f}{3}.$$

En cuanto al valor promedio:

- Para el cálculo de θ el número de mensajes enviados por columna será el número de procesadores no diagonales con entradas en esa columna, esto es: $P_f - 1$ si $\frac{\alpha}{N} \geq P_f$ y $\frac{\alpha}{N} - 1$ en otro caso.
- En cuanto al resto de los mensajes generados, basta multiplicar el número de mensajes generados por cada columna local dentro de cada procesador por el numero de procesadores por columna y el número total de columnas:

$$MV = (P_f - 1)(N + \alpha - \frac{NP_f}{2}) \quad \text{si} \quad \frac{\alpha}{N} \geq P_f$$
$$\overline{MV} = (\frac{\alpha}{N} - 1)(\frac{\alpha}{2} + N) \quad \text{si} \quad \frac{\alpha}{N} < P_f. \ \square$$

Proposición 4 *El número total de mensajes horizontales generados es:*

$$MH = \sum_{I=0}^{N-1} \sum_{p=o}^{P-1} min\{d_i^{p'}, 1, m_i^p\} + \sum_{p=0}^{P-1} \sum_{j=0}^{col^p} M'^p_{jI'}$$

cumpliéndose que:

$$0 \leq MH \leq (P_c - 1)(N - \frac{P_c}{2}) + (\frac{P_c - 1}{P_c})\frac{P_f(P_f + 1)}{2}(N - \frac{4}{3} - \frac{2P_f}{3}) + (\frac{P_c - 1}{P_c})\frac{Pf(Pf - 1)(P_f - 2)}{6}$$

y siendo el valor promedio:

$$\overline{MH} = NP_f(\frac{\alpha}{N} - \frac{P_f}{2} - \frac{3}{2})\frac{P_c - 1}{P_c} + N(P_c - 1) \quad \text{si} \quad \frac{\alpha}{N} \geq P_c$$
$$\overline{MH} = \frac{N}{2}(\frac{\alpha}{N} - 2)(\frac{\alpha}{N} - 1)\frac{P_c - 1}{P_c} + N(\frac{\alpha}{N} - 1) \quad \text{si} \quad \frac{\alpha}{N} < P_c$$

Demostración:

- Por un lado, por cada fila y por cada procesador no diagonal con entradas en dicha fila se envía un mensaje para la modificación del elemento diagonal:

$$\sum_{I=0}^{\text{N-1}} \sum_{p=0}^{P-1} min\{d_i^{p'}, 1, m_i^p\}.$$

- Además, cada procesador envía $M_{jI'}^{'p}$ por cada columna j local.

En el caso denso:

- Cada procesador no diagonal con entradas en una fila envía un mensaje al procesador que contiene la diagonal. Como el número de entradas por fila varía entre 1 y N, el número de procesadores con entradas en una fila también variará, de modo que para las N $-$ P_c últimas filas habrá al menos una entrada por procesador, y, por tanto, se generarán (N $-$ P_c) (P_c $-$ 1) mensajes. A partir de ahí, el número de procesadores no diagonales con entradas en una fila va disminuyendo en una unidad, hasta hacerse cero para la primera fila, tendremos por tanto $\sum_{i=0}^{P_c-1} i$ mensajes para las P_c primeras filas. En total:

$$(N - P_c)(P_c - 1) + \sum_{i=0}^{P_c-1} i = (P_c - 1)(N - \frac{P_c}{2}).$$

- Por otro lado, para las primeras N $-$ P_f columnas, todos los procesadores tendrán entradas en la columna considerada, y por tanto todos los procesadores enviarán mensajes horizontales. Por cada columna j, el procesador que tiene la última entrada enviará N $-$ j $-$ 2 mensajes, resultado de multiplicarla por las N $-$ j $-$ 2 entradas no diagonales que se encuentran en filas inferiores. El procesador que contiene la penúltima entrada de la columna j enviará N $-$ j $-$ 3 mensajes, y así sucesivamente. Por tanto, para las primeras N $-$ P_f columnas tendremos el siguiente número de mensajes:

$$\sum_{j=0}^{N-P_f-1} \sum_{i=0}^{P_f-1} (N - j - 2 - i).$$

De estos mensajes tendremos que descontar los enviados a si mismo:

$$\sum_{j=0}^{N-P_f-1} \sum_{i=0}^{P_f-1} (N - j - 2 - i)(\frac{P_c - 1}{P_c}) = (\frac{P_c - 1}{P_c}) \sum_{i=1}^{P_f} (N - 1 - i)i.$$

Para las últimas Pf columnas, el número de procesadores con entradas en esa columna disminuye y, por tanto, se debe ir descontando un procesador a medida que avanzamos en las columnas:

$$(\frac{P_c-1}{P_c})\sum_{j=0}^{P_f-1}\sum_{i=0}^{P_f-3-j}(P_f-2-i-j) = (\frac{P_c-1}{P_c})\sum_{i=1}^{P_f-2}(P_f-i-1)i.$$

- En total:

$$\begin{aligned} MH &\leq (P_c-1)(N-\frac{P_c}{2}) + (\frac{P_c-1}{P_c})\sum_{i=1}^{P_f} i-1)i \\ &+ (\frac{P_c-1}{P_c})\sum_{i=1}^{P_f-2}(P_f-i-1)i \end{aligned}$$

Como:

$$\begin{aligned} \sum_{i=1}^{P_f}(N-i-1)i &= N\sum_{i=1}^{P_f} i - \sum_{i=1}^{P_f} i^2 - \sum_{i=1}^{P_f} i \\ &= P_f(P_f+1)\frac{2}{1)} - \frac{P_f(P_f+1)}{2} - \frac{P_f\,(P_f+1)(2P_f+1)}{6} \\ &= \frac{}{2}(N-\frac{4}{3}-\frac{2P_f}{3}). \end{aligned}$$

y:

$$\begin{aligned} \sum_{i=1}^{P_f-2}(P_f\ -i-1)i = \ & Pf\ \sum_{i=1}^{P_f-2} i\ - \sum_{i=1}^{P_f-2} i^2 - \sum_{i=1}^{P_f-2} \mathrm{i} \\ &= \frac{P_f(P_f-1)(P_f-2)}{6}. \end{aligned}$$

Entonces:

$$\begin{aligned} 0 \leq MH &\leq (P_c-1)(N-\frac{P_c}{2}) + (\frac{P_c-1}{P_c})\frac{P_f(P_f+1)}{2}(N-\frac{4}{3}-\frac{2P_f}{3}) \\ &+ (\frac{P_c-1}{{}^{1}P_c})\frac{P_f(P_f-1)(P_f\ -2)}{6}. \end{aligned}$$

En cuanto al valor promedio:

- A los mensajes enviados para la modificación de columnas posteriores hay que sumarles el número de mensajes que se genera por cada fila para la modificación de la diagonal. □

Corolario 1 ***Para matrices grandes, en el caso denso, el número de mensajes totales generados para P_f = P es siempre mayor que el número de mensajes totales generados para P_c = P, siendo P $\geq$ 2***

Demostración:

- Si P_f **= P,** entonces P_c = 1 y por tanto ***MH = 0.*** En cuanto a ***MV,*** para matrices lo suficientemente grandes, y suponiendo que N >> ***P,*** tendremos:

$$M_{T1} = MV \simeq N^2 \frac{P - 1}{2}.$$

- Por otro lado, si P_c **= P,** entonces P_f = 1 y por tanto ***MV = 0.*** En cuanto a ***MH,*** si suponemos de nuevo matrices lo suficientemente grandes y N >> ***P,*** tendremos:

$$M_{T2} = MH \simeq N(P - 1 + \frac{P - 1}{P}).$$

Por tanto, $M_{T1} > M_{T2}$ cuando N es grande. □

Corolario 2 ***Para matrices grandes, en el caso denso, el máximo en el número de comunicaciones se produce para P_f = P y el mínimo para P_c = P***

Demostración:

- El número total de mensajes generados para matrices grandes y para una configuración de procesadores dada suponiendo N >> ***P*** será:

$$M_T = MV + MH \simeq N^2 \frac{P_f - 1}{2}.$$

Derivando con respecto a P_f:

$$\frac{dM_T}{dP_f} \simeq \frac{N^2}{2} > 0.$$

Por tanto, M_T ***es*** una función estrictamente creciente con P_f, ***es*** decir, en el intervalo [1,P] tendrá el máximo en P_f **= P y** el mínimo en P_f = 1 o, equivalentemente, en P_c **= P. Cl**

Corolario 3 ***Para la matriz promedio, el número de mensajes totales generados para*** P_f ***= P es siempre mayor que el número de mensajes totales generados para*** P_c ***= P, siendo*** $P \geq 2$

Demostración:

- Si P_f **= P** entonces P_c = 1 y por tanto ***MH = 0.*** Entonces:

$$\overline{M_{T1}} = \overline{MV} = N(P-1)(1+\frac{\alpha}{N}-\frac{P}{2}) \quad \text{si} \quad \frac{\alpha}{N} \geq P$$
$$\overline{M_{T1}} = \overline{\boldsymbol{MV}} = N(\frac{\alpha}{N}-1)(\frac{\frac{\alpha}{N}}{2} + \boldsymbol{1}) \quad \text{si} \quad \frac{\alpha}{N} < P.$$

- Por otro lado, si ***Pc = P,*** entonces P_f = 1 y , por tanto, ***MV = 0.*** Entonces:

$$\overline{M_{T2}} = \overline{MH} = N(P-1)(1+\frac{\frac{\alpha}{N}}{P}-\frac{2}{P}) \quad \text{si} \quad \frac{\alpha}{N} \geq P$$
$$\overline{M_{T2}} = \overline{\boldsymbol{MH}} = N(\frac{\alpha}{N}-1)(1+\frac{\frac{\alpha}{N}}{2}\frac{P-1}{P}-\frac{P-1}{\boldsymbol{P}}) \quad \text{si} \quad \frac{\alpha}{N} < P.$$

- Si $\frac{\alpha}{N} \geq P$:

$$\begin{aligned}\overline{M_{T1}} - \overline{M_{T2}} &= N(P-1)(\frac{\alpha}{N}-\frac{P}{2}-\frac{\frac{\alpha}{N}}{P}+\frac{2}{P}) \\ &= N(P-1)(\frac{\alpha}{N}(1-\frac{1}{P})+\frac{2}{P}-\frac{P}{2}) \\ &\geq \boldsymbol{N(P}-1)(P(1-\frac{1}{P})+\frac{2}{P}-\frac{P}{2}) \\ &= N(P-1)(\frac{P}{2}-1+\frac{2}{P}) \geq N(P-1)\frac{2}{P} > 0\end{aligned}$$

cumpliéndose, por tanto, que $\overline{M_{T1}} > \overline{M_{T2}}$.

- Si $\frac{\alpha}{N} < \boldsymbol{P}$:

$$\begin{aligned}M_{T2} &= N(\frac{\alpha}{N}-1)(1+\frac{\frac{\alpha}{N}}{2}\frac{P-1}{P}-\frac{P-1}{\boldsymbol{P}}) \\ &< N(\frac{\alpha}{N}-1)(1+\frac{\frac{\alpha}{N}}{2}-\frac{P-1}{P}) \\ &< N(\frac{\alpha}{N}-1)(1+\frac{\frac{\alpha}{N}}{2}) = M_{T1}. \quad \square\end{aligned}$$

MATRIZ	Origen	N	α en A	α en L
BCSSTM07	Análisis dinámico en ingeniería estructural	420	3836	14282
ERIS1176	Redes eléctricas	1176	9864	49639
ZENIOS	Control de tráfico aéreo	2873	15032	62105
RANDOM	Generada aleatoriamente	1250	1153	32784
RANDOM 1	Generada aleatoriamente	2000	1975	106545

Tabla 4.1: *Matrices prueba*

Estos resultados nos indican, en primera aproximación, que tanto en el caso denso como en un caso promedio artificial, la mejor configuración de una red de $P = P_f$ x P_c procesadores será aquella en la cual $P_c = P$ y $P_f = 1$, es decir, aquella para la cual se minimiza el número de comunicaciones. Sin embargo, esta conclusión depende del patrón de la matriz, y de hecho, se pueden encontrar estructuras artificiales, y posiblemente poco prácticas, en las que la mejor distribución no se corresponda con $P_f = 1$. Por ello, en el siguiente apartado, y para completar este estudio, presentamos resultados reales del número de comunicaciones generado para ciertas matrices de prueba obtenidas de los *benchmark* estándar. Cabe destacar que hablar de la mejor configuración de una red de procesadores es equivalente, en nuestro caso, a hablar de la mejor distribución cíclica de la matriz.

4.4.1 Comunicaciones para matrices prueba

En la práctica, el número de comunicaciones depende del patrón de la matriz. Dado el patrón de una matriz y el número de procesadores de la malla del sistema de memoria distribuida, se puede determinar cual es la configuración de la malla que menor número de mensajes genera. Para ilustrarlo hemos seleccionado un conjunto de cinco matrices dispersas: las matrices BCSSTM07, ERIS 1176 y ZENIOS de la colección de matrices *Harwell-Boeing* [22] y 2 matrices generadas aleatoriamente, RANDOM y RANDOMl. La Tabla 4.1 resume sus características, donde Origen indica la disciplina científica de la cual proceden, N es el número de filas y columnas y α es el número de entradas no nulas. Los patrones de estas matrices son mostrados en la Figura 4.4, dado que son simétricas sólo se muestra la parte triangular inferior que es con la que se trabaja.

En las Figuras 4.5, 4.6 y 4.7 se muestra el número de mensajes horizontales, verticales y totales generados para diferente configuración de la red de procesadores $(P = P_f \times P_c)$ y para $P = 16$, 32 y 64 respectivamente. Se concluye que la mejor configuración de la red es aquella para la cual $P_f = 1$ y $P_c = P$, es decir, aquella en la que cada procesador almacena una columna entera de la matriz. Esto se debe a

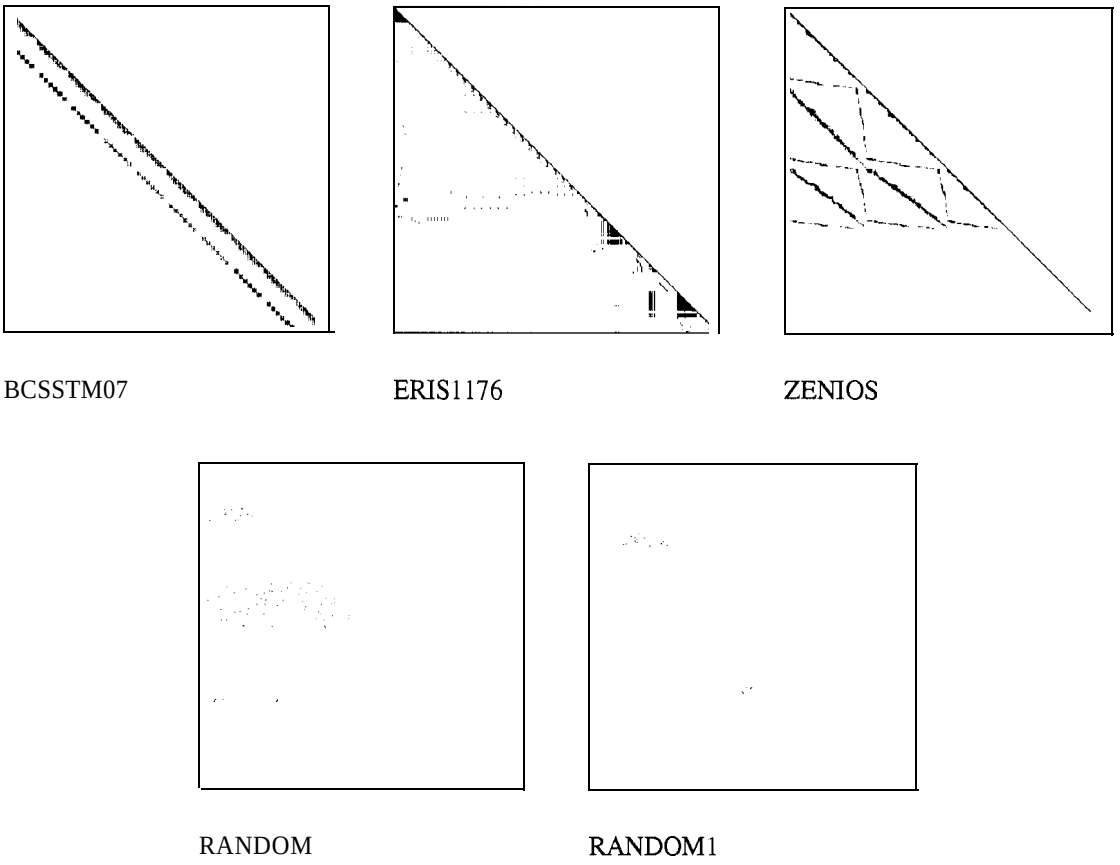

Figura 4.4: *Patrones de las matrices prueba*

dos razones, en primer lugar, para esta configuración se eliminan todos los mensajes verticales, y en segundo lugar, se maximiza el posible empaquetamiento de mensajes horizontales reduciendo con ello el número total de mensajes. Se observa, además, que la configuración que produce un mayor número de mensajes corresponde, en general, con la configuración $P_f = P$ y $P_c = 1$. Excepciones a este comportamiento son las matrices BCSSTM07, ERIS1176 y ZENIOS para $P = 64$.

En las Figuras 4.8,4.9 y 4.10 se ilustra el descenso en el número de comunicaciones que supone aplicar el empaquetamiento de mensajes propuesto en la Sección 4.3.1.2 para diferentes configuraciones de una red de 16,32 y 64 procesadores respectivamente. En dichas figuras se muestra el número total de mensajes enviados con y sin empaquetamiento. Obviamente el empaquetamiento sólo disminuye el número de mensajes horizontales. Por consiguiente, el número total de mensajes para las configuraciones 16 x 1,32 x 1 y 64 x 1 coincide para las versiones con y sin empaquetamiento. De igual forma, el mayor descenso en el número de comunicaciones debido al empaquetamiento se obtiene para las configuraciones 1 x P.

4.5 Balanceo de la carga

Utilizamos como medida del balanceo de la carga:

$$B = FLOP_{total}/(P * FLOP_{max})$$

donde $FLOP_{total}$ *es* el número de operaciones totales en punto flotante necesarias para llevar a cabo la factorización, P *es* el número de procesadores y $FLOP_{max}$ *es* el valor máximo del número de operaciones en punto flotante asignadas a los procesadores. B *es* tal que $0 < B \leq 1$. En el caso de carga completamente balanceado el resultado será la unidad, cuanto peor balanceada se encuentre la carga, más nos acercaremos al valor 0. Además esta variable constituye un límite superior en la eficiencia del programa paralelo: eficiencia $\leq B$.

Nótese que para el caso denso y la matriz promedio utilizada en la sección anterior el balanceo de la carga sería prácticamente perfecto, siendo $B \simeq 1$.

Las Figuras 4.11,4.12 y 4.13 muestran los resultados obtenidos para cada una de las matrices prueba y para P *=16,32 y 64* procesadores y diferentes configuraciones de la red de procesadores.

El balanceo de la carga para muchas configuraciones alcanza valores próximos a la unidad y, en todos los casos, se mantiene dentro de unos límites aceptables, no bajando nunca del valor 0.5. Por tanto van a ser las comunicaciones, y no el balanceo de la carga, el factor determinante en la mejor configuración de la red de procesadores con respecto a los tiempos de ejecución.

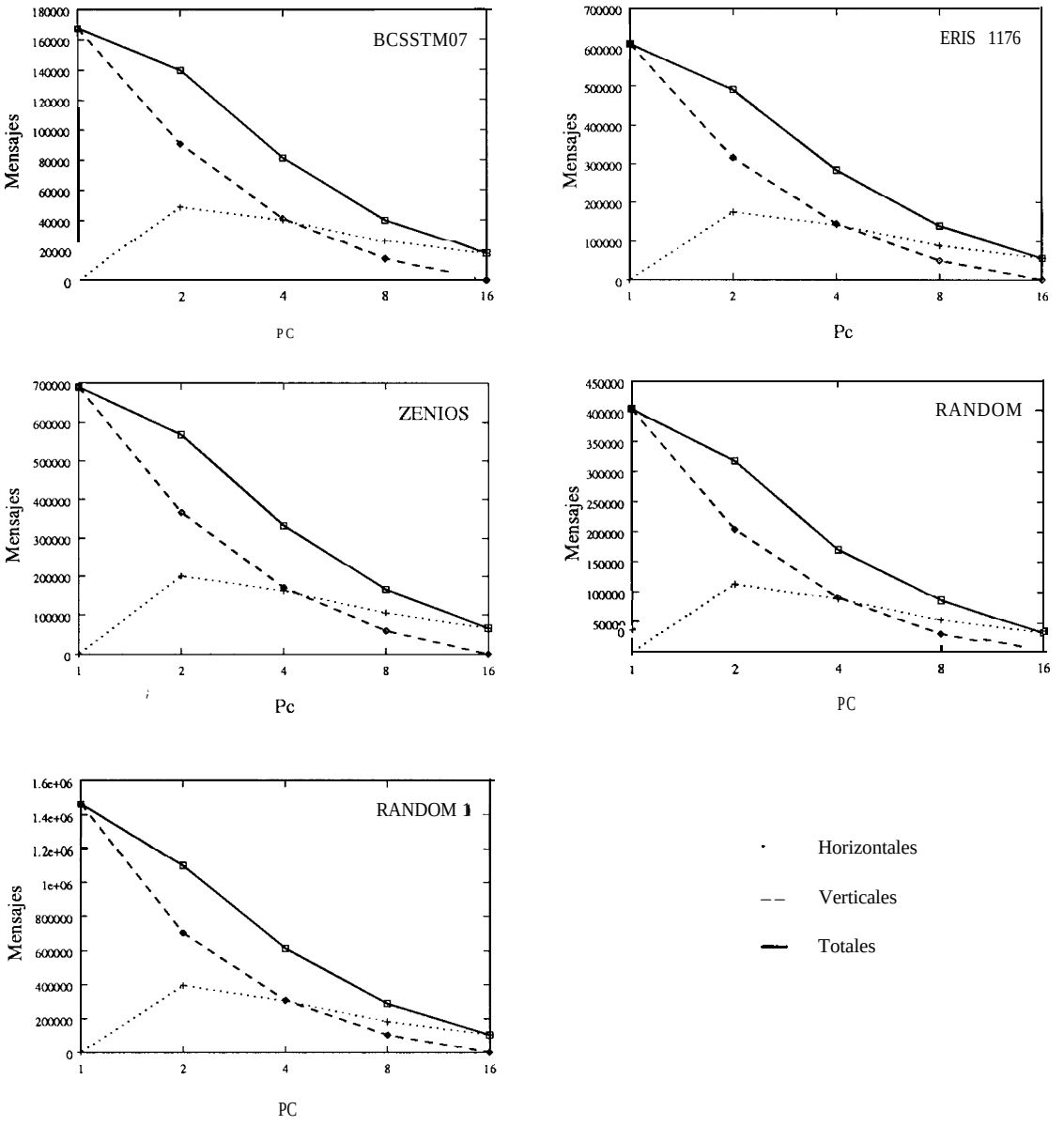

Figura 4.5: ***Número de comunicaciones para diferentes configuraciones de una red de 16 procesadores***

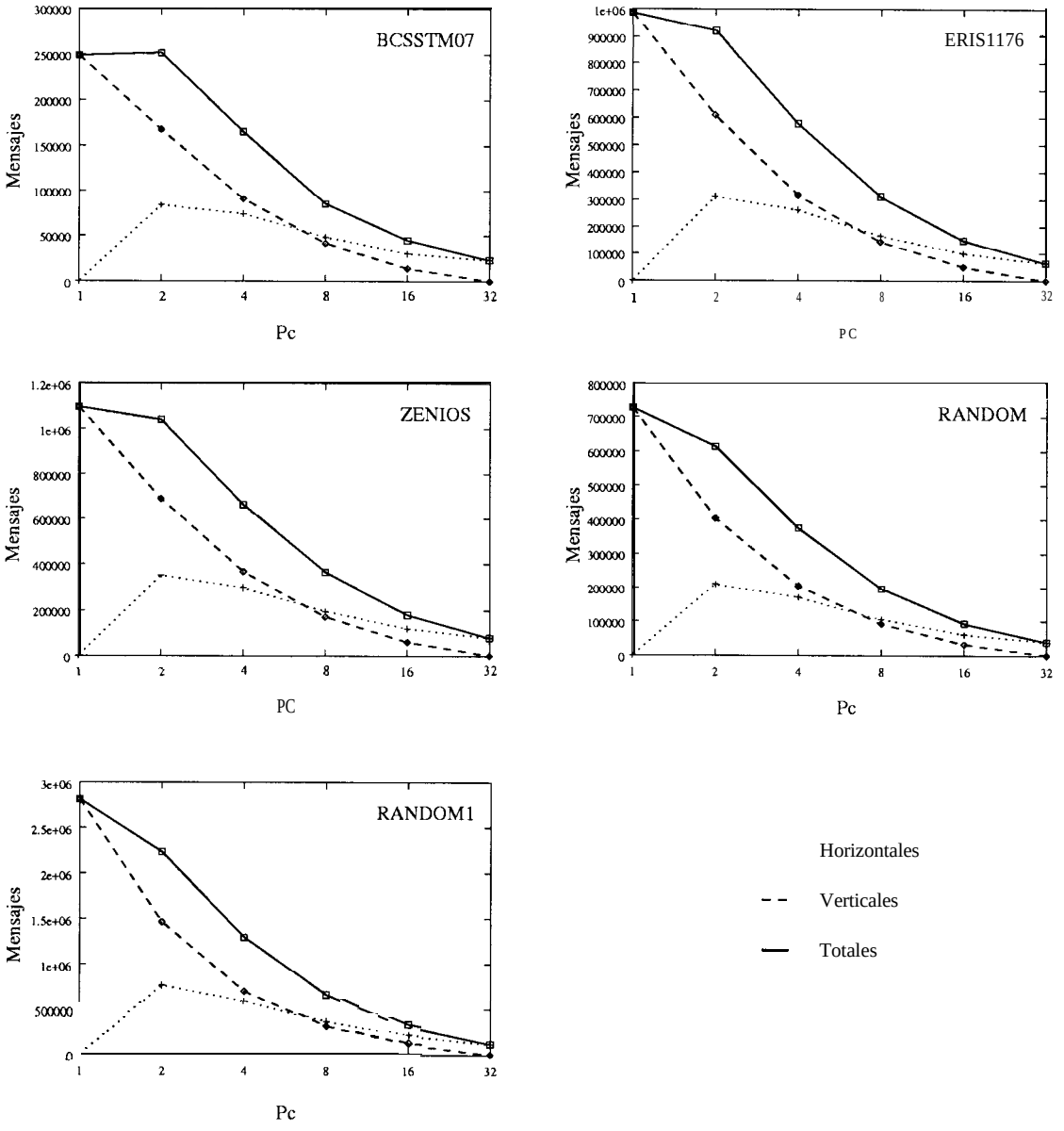

Figura 4.6: ***Número de comunicaciones para diferentes configuraciones de una red de 32 procesadores***

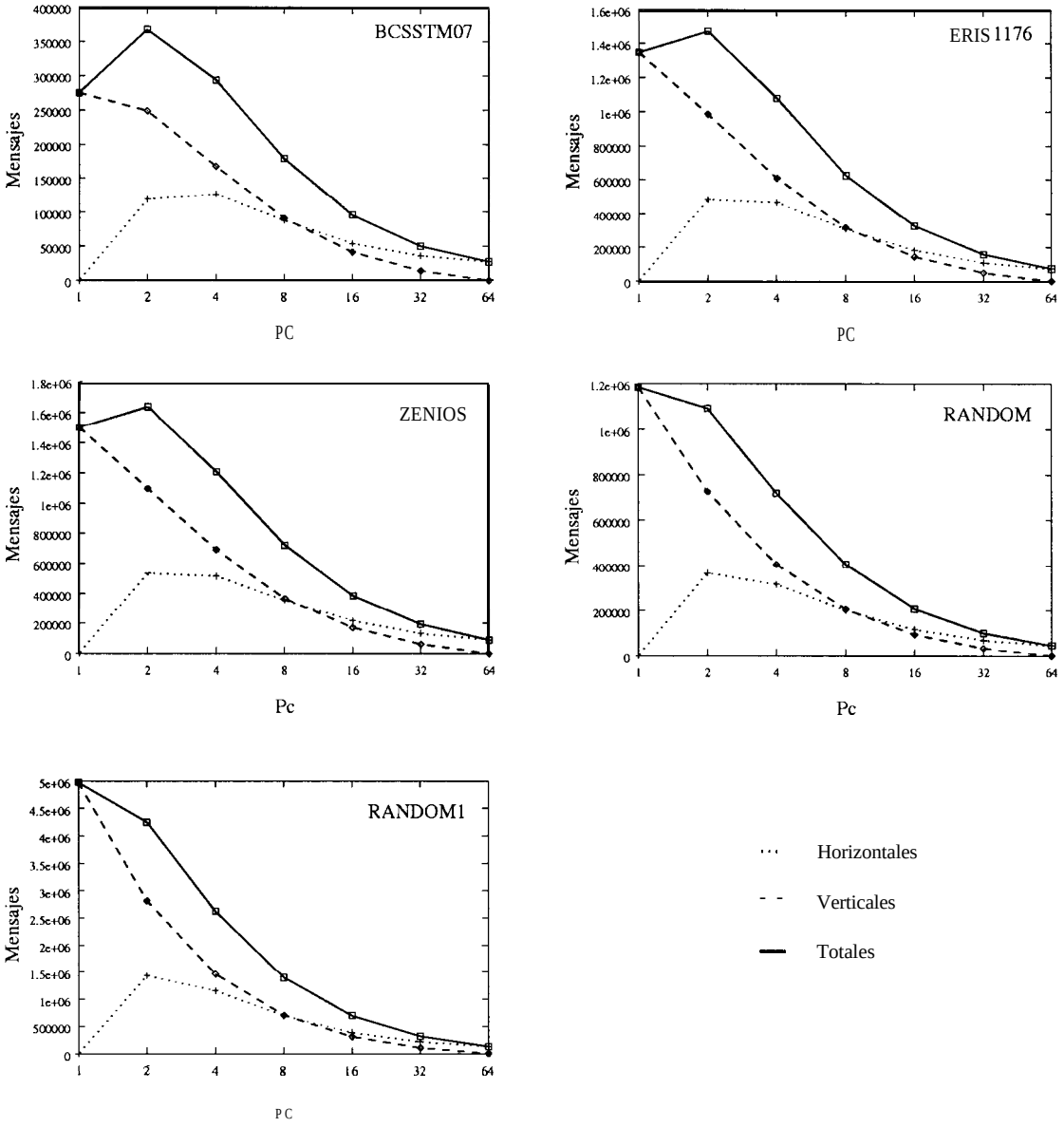

Figura 4.7: ***Número de comunicaciones para diferentes configuraciones de una red de 64 procesadores***

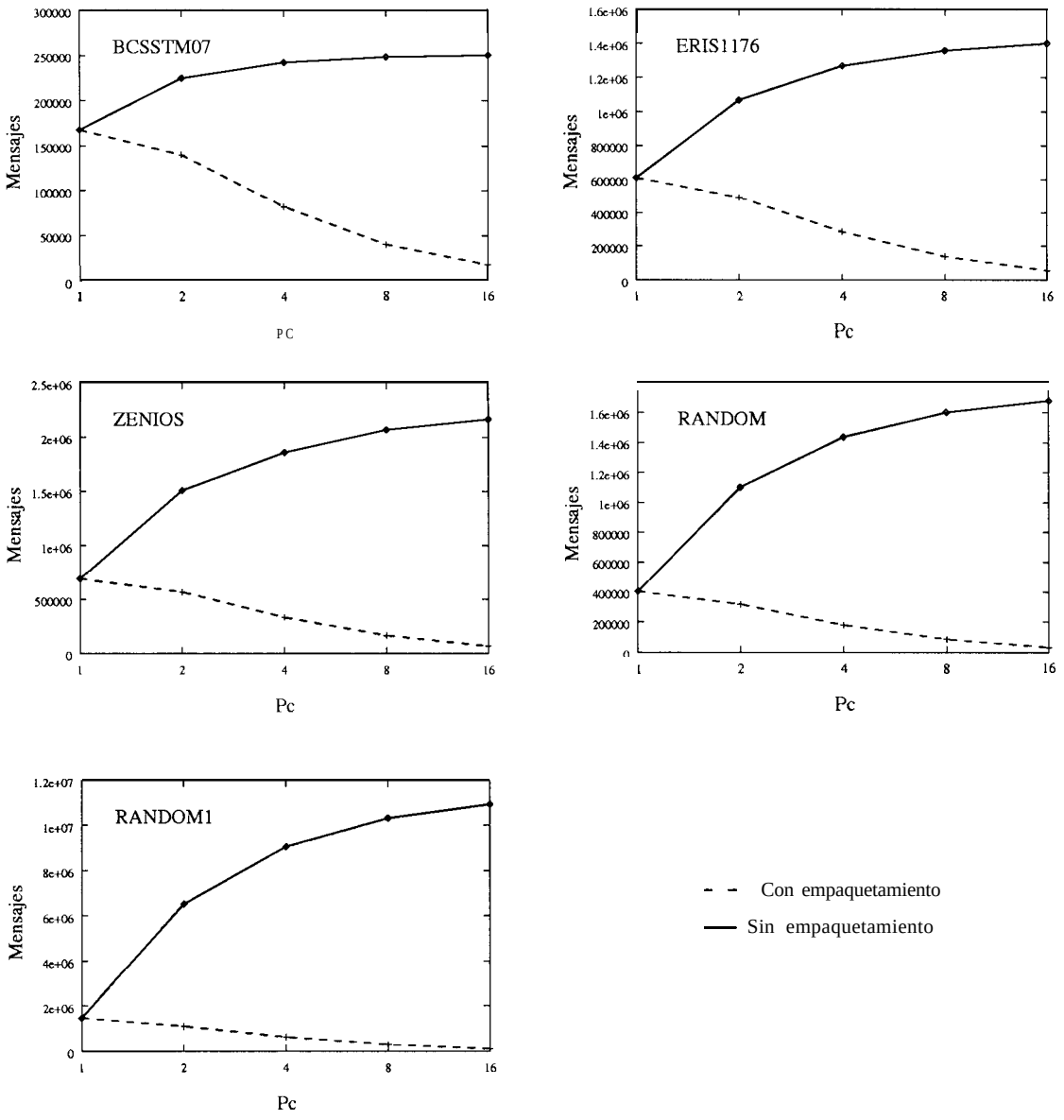

Figura 4.8: Número de ***comunicaciones con y sin empaquetamiento de mensajes para diferentes configuraciones de una red de 16 procesadores***

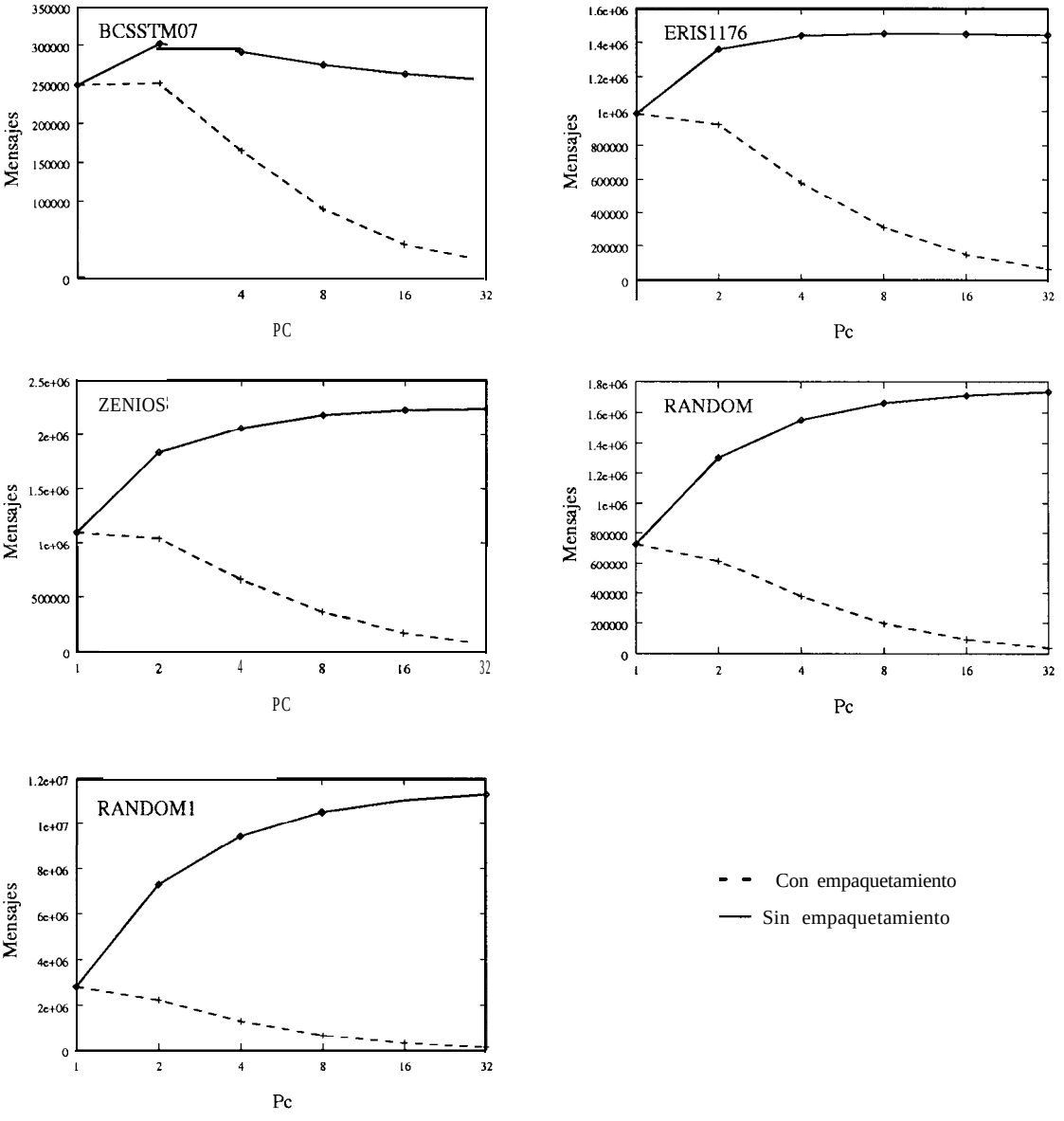

Figura 4.9: ***Número de comunicaciones con y sin empaquetamiento de mensajes para diferentes configuraciones de una red de 32 procesadores***

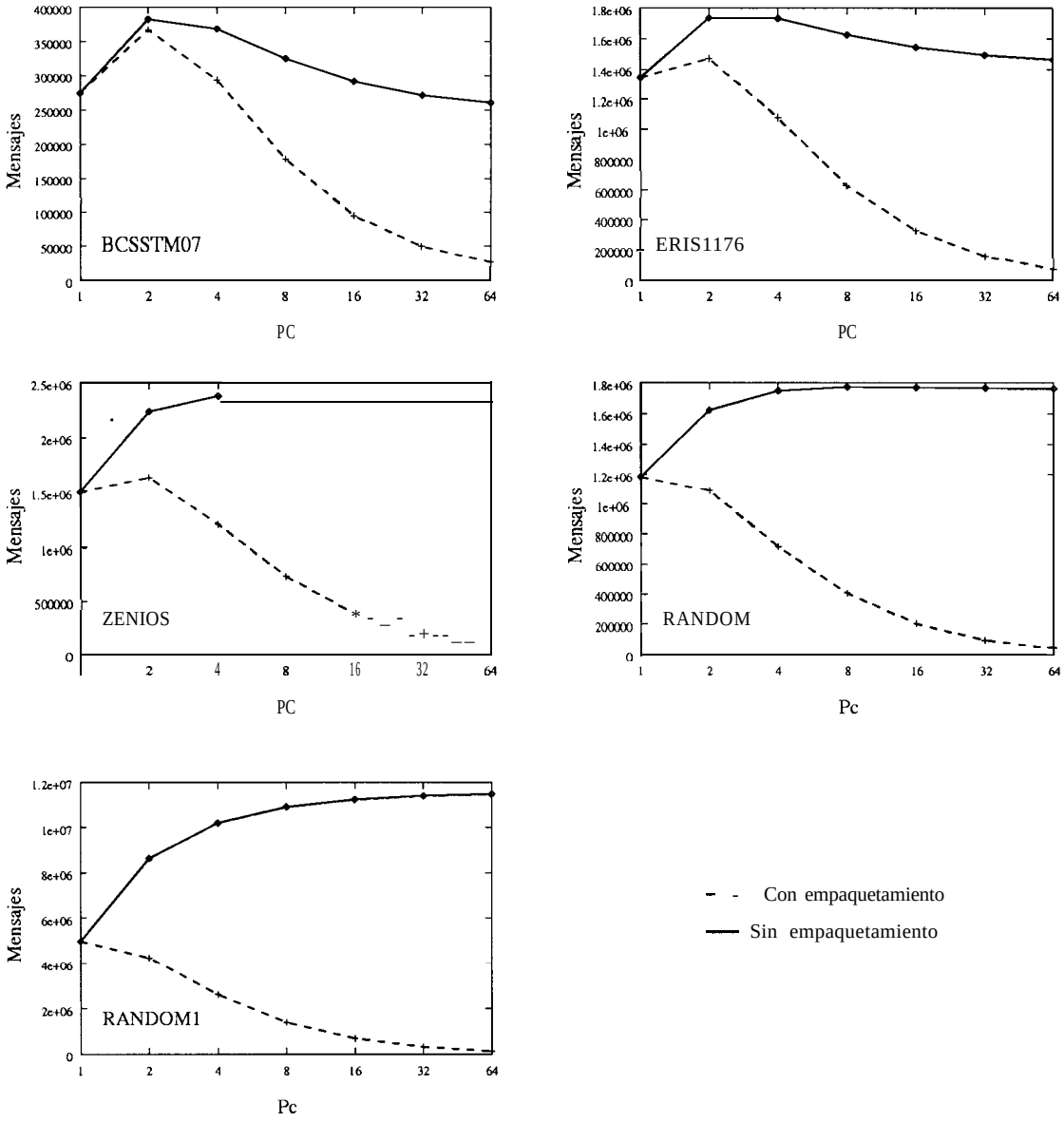

Figura 4.10: ***Número de comunicaciones con y sin empaquetamiento de mensajes para diferentes configuraciones de una red de 64 procesadores***

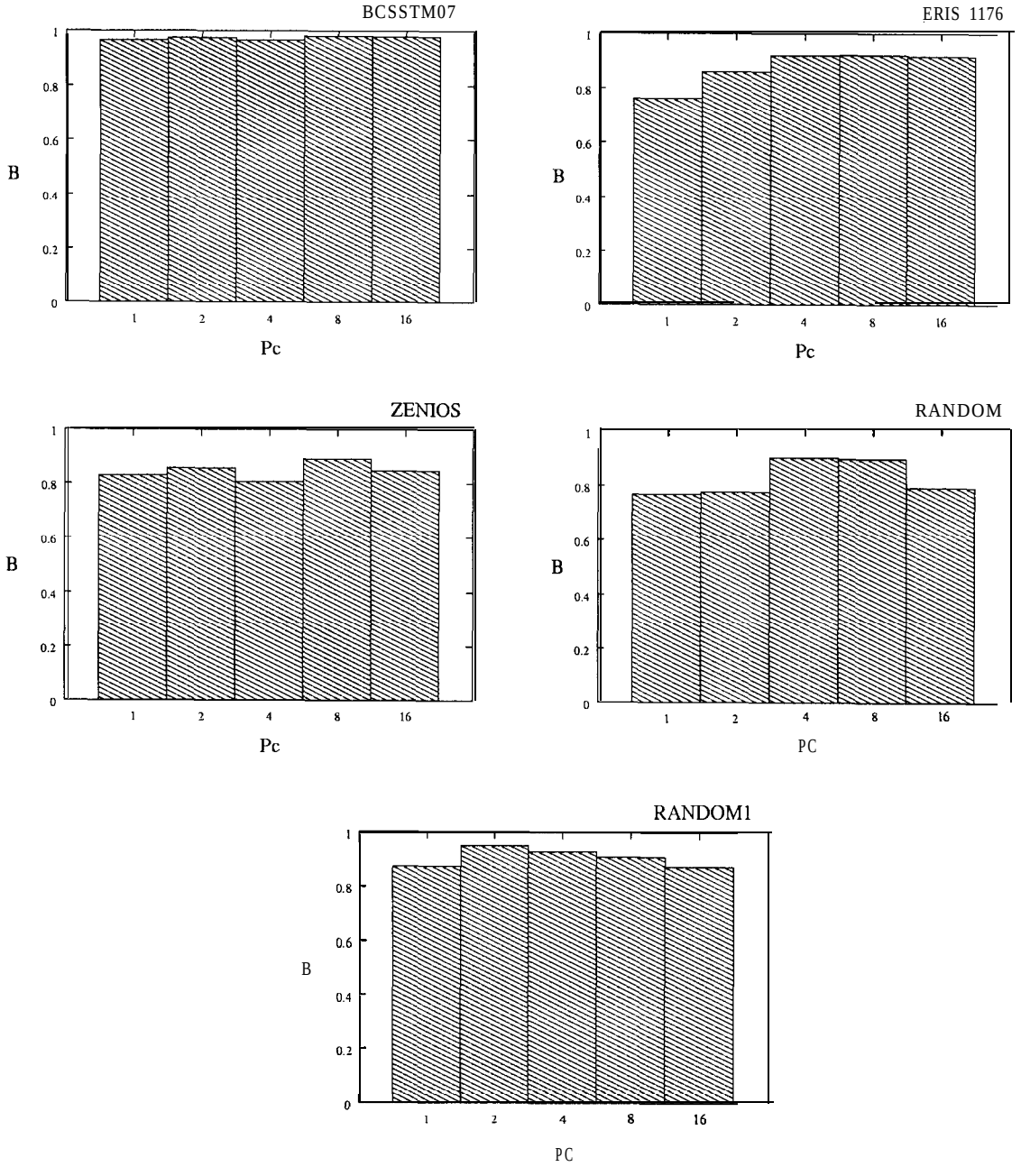

Figura 4.11: Balanceo de la carga para diferentes configuraciones de una red de 16 procesadores

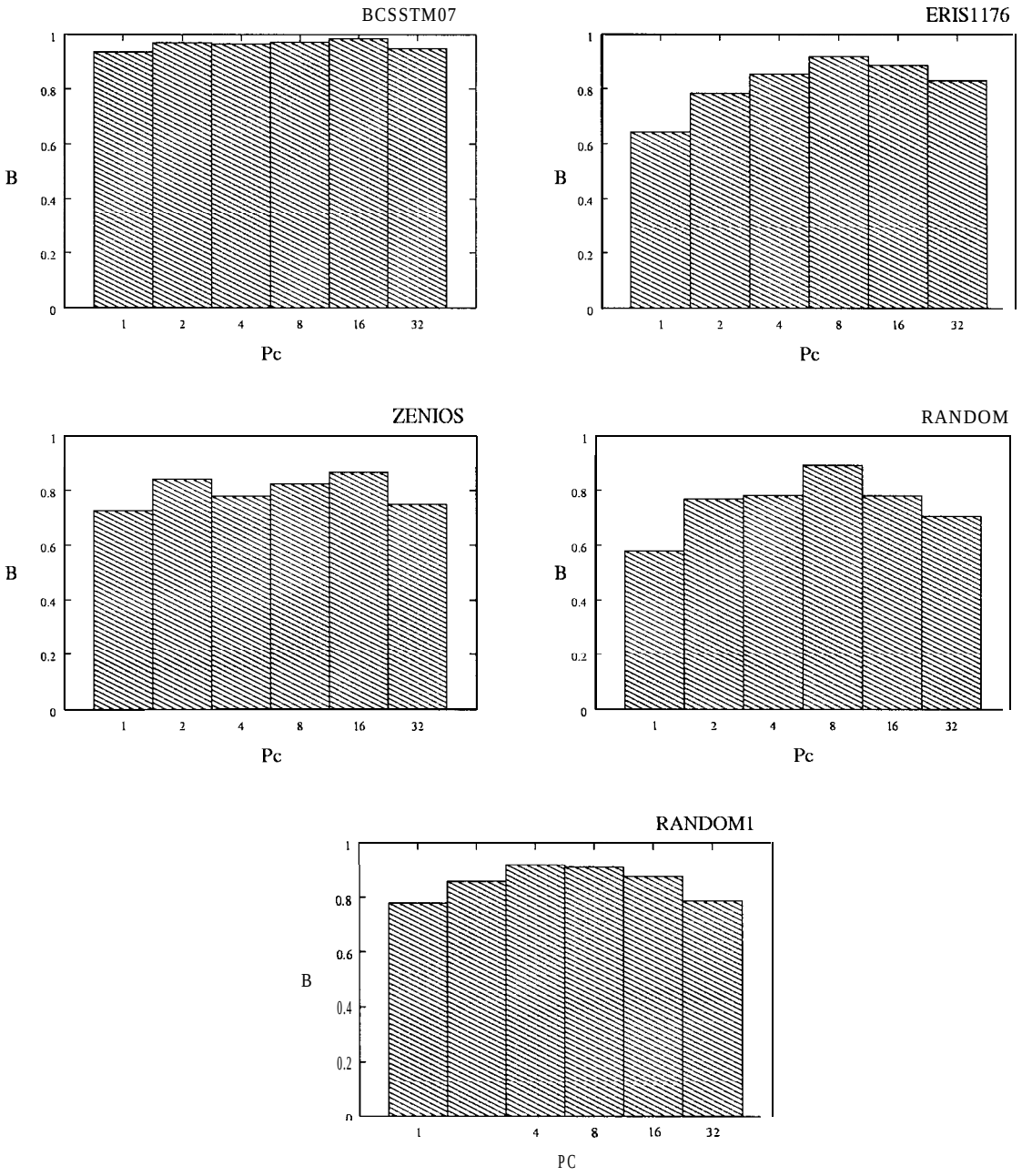

Figura 4.12: ***Balanceo de la carga para diferentes configuraciones de una red de 32 procesadores***

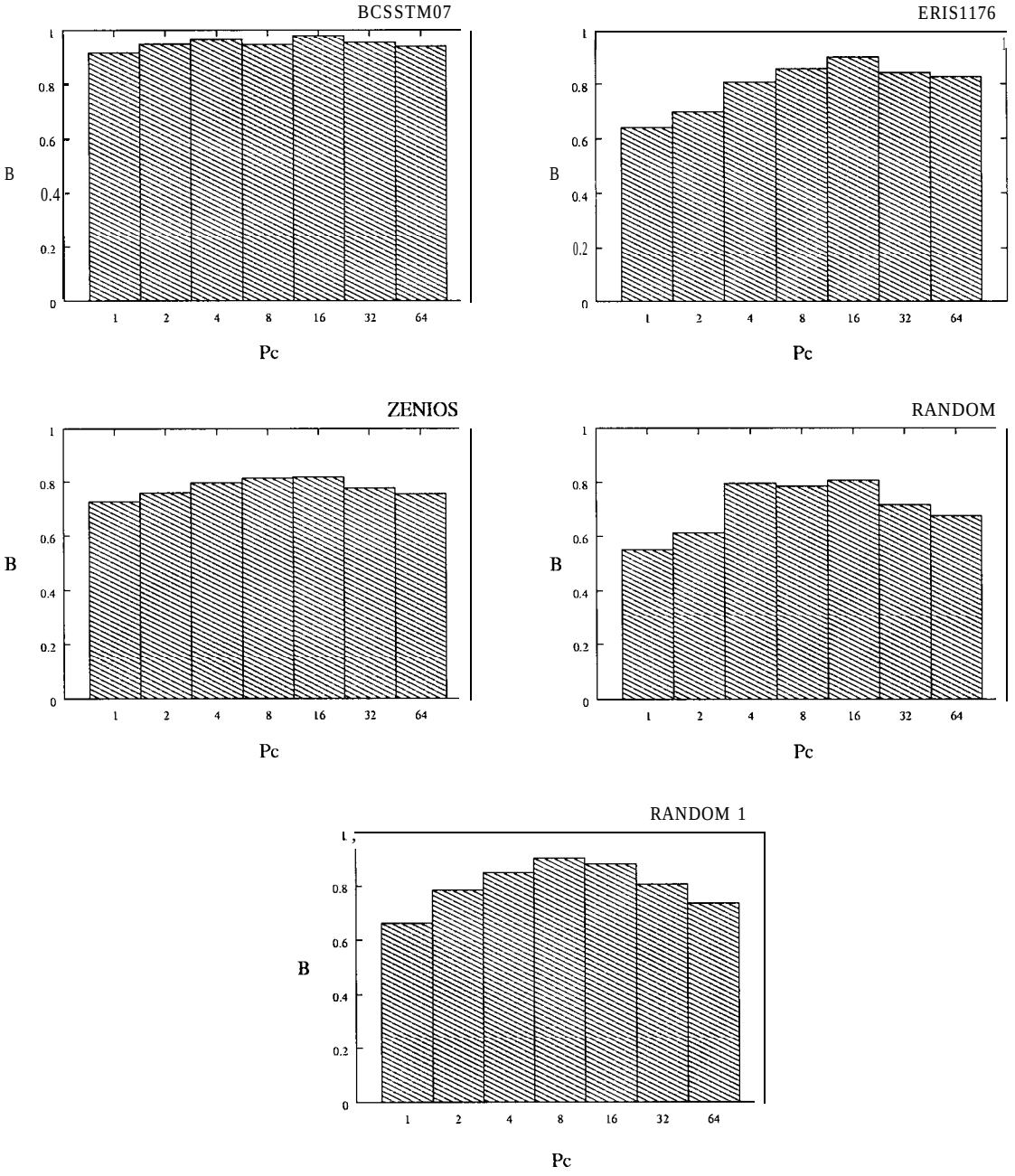

Figura 4.13: ***Balanceo de la carga para diferentes configuraciones*** de ***una red de*** 64 ***procesadores***

4.6 Tiempos de ejecución para la distribución bidimensional

Hemos implementado el programa sobre el sistema de memoria distribuida AP1000 de Fujitsu utilizando doble precisión y la librería nativa de pase de mensajes Cellos [72].

Se ha utilizado para el envío y la recepción de mensajes los denominados modo de envío por líneas (line-sending mode) y recepción por ***buffer (buffer-receiving)*** respectivamente.

- En el modo de envío por líneas los mensajes son enviados directamente desde la memoria cache sin utilizar ***buffers*** de envío. Si el mensaje no está en la cache se busca en la memoria principal y se envía directamente desde la memoria.
- La recepción por ***buffer se*** corresponde con un transferencia de datos asíncrona utilizando un ***buffer*** circular alocado en memoria principal y denominado ***ring buffer.*** Existe ***hardware*** específico para detectar posibles *overflows.* Una serie de punteros almacenados en registros mantienen las direcciones de memoria necesarias para su actualización.

Las operaciones básicas de envío sin esta utilidad trabajan conjuntamente con un sistema de ***buffers, es*** decir, los mensajes enviados son copiados previamente a un ***buffer*** dentro de la memoria. Y así mismo, los mensajes que están siendo recibidos por una celda son colocados en un ***buffer*** a medida que se están recibiendo. Utilizando *linc-sending* y ***buffer-receiving se*** evitan las operaciones extra de copia de datos al ***buffer y*** disminuye, por tanto, la latencia. En contrapartida, el ***ring buffer*** tiene un tamaño máximo de 512 KB, lo cual, en algunos casos, puede resultar insuficiente. A partir de ahora llamaremos a las comunicaciones que hacen uso de estas utilidades comunicaciones de baja latencia.

En la Figura 4.14 se muestran los tiempos de ejecución obtenidos en segundos para las diferentes matrices consideradas y para diferentes configuraciones de la red de procesadores, $\boldsymbol{P}$ = 16, 32 y 64. Tal y como cabía esperar, para un número de procesadores dado, la mejor configuración en malla bidimensional se corresponde con la configuración que genera el menor número de comunicaciones, es decir, la configuración en la cual cada procesador almacena columnas enteras de la matriz de forma cíclica.

En la Figura 4.15 se representa la aceleración (*Speed-up)* obtenida para cada una de las matrices y para la configuración $\boldsymbol{P}$ = 1 x P_c. Estos resultados corresponden al algoritmo paralelo sin tener en consideración el tiempo de la fase de preprocesamiento para su cálculo. En muchas aplicaciones que requieren la resolución de problemas de optimización no lineales este algoritmo se suele usar dentro de un proceso iterativo en

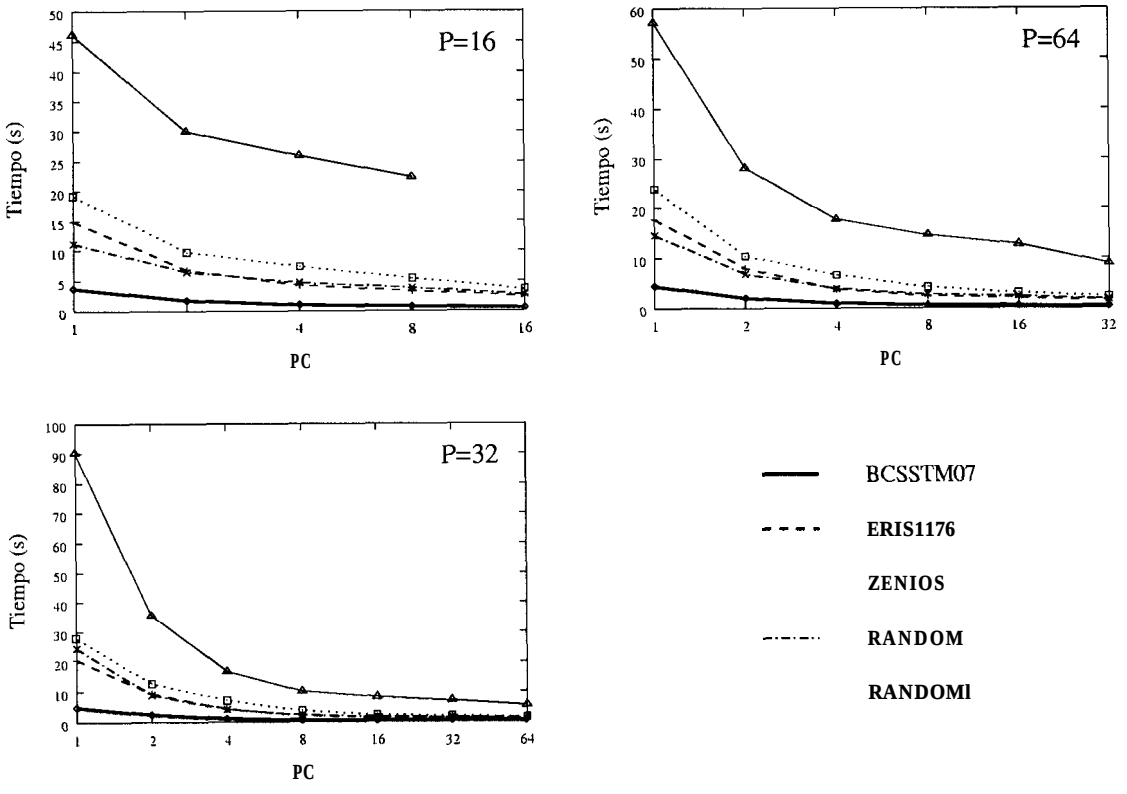

Figura 4.14: ***Tiempos de ejecución de los programas paralelos***

el cual el patrón de la matriz no varía. Por tanto, la fase de preprocesamiento, así como la factorización simbólica de la matriz, se convierten en unas fases previas las cuales se ejecutan solamente una vez.

La aceleración se ha calculado a partir del programa paralelo para un procesador. En la Tabla 4.2 se muestran las diferencias de tiempo en segundos entre el programa paralelo para un procesador y el programa secuencial. Se puede observar que las diferencias no son significativas debido, fundamentalmente, a que la principal diferencia entre ambos programas radica en la inclusión de funciones de comunicación con tiempos de espera asociados. Esto no puede venir reflejado en el programa paralelo para un procesador, se aprecia, sin embargo, en la relativamente baja aceleración conseguida para 2 procesadores. El paso de 1 a 2 procesadores introduce un *overhead* en el tiempo de ejecución que va a ser arrastrado en los resultados de aceleración obtenidos para más de 2 procesadores. Esto se puede ver más claramente representando la aceleración relativa. Definimos la aceleración relativa para $2p$ procesadores como el cociente entre el tiempo de ejecución del algoritmo paralelo para p procesadores y el tiempo de ejecución para $2p$ procesadores. Idealmente este cociente debería ser 2. En la Figura 4.16 se muestran los resultados obtenidos de esta aceleración relativa para las matrices consideradas.

Puede observarse que la aceleración relativa se mantiene prácticamente en todos los casos entre 1.5 y 2, siendo el resultado más bajo el obtenido para 2 procesadores salvo en el caso de que se aumente mucho el número de procesadores. Debe tenerse en cuenta, en este caso, que las matrices consideradas son bastante pequeñas y es imposible mantener la eficiencia del algoritmo al incrementar el número de procesadores debido al escaso número de computaciones a realizar. Esto se ve reflejado en los resultados obtenidos para la matriz BCSSTM07, esta matriz es la más pequeña de las consideradas, y para ella ya se degrada el comportamiento para 32 procesadores, en cambio para la matriz ZENIOS (la matriz más grande considerada) el comportamiento se empieza a degradar a partir de 64 procesadores.

Para la matriz RANDOM1 se observa además una superlinealidad, este comportamiento puede ser debido a dos motivos: a un mejor aprovechamiento de la jerarquía de memoria, o a una mejor distribución de las columnas a procesar, evitando dependencias y, por tanto, eliminando tiempos de espera.

4.6.1 Implementación en MPI

La librería estándar de pase de mensajes MPI está disponible en el AP1000. Su implementación se debe a David Sitsky del *Department of Computer Science, Computer Sciences Laboratory, The Australian National University [102].* En esta sección establecemos una comparación de los resultados obtenidos utilizando MPI y la librería CellOS.

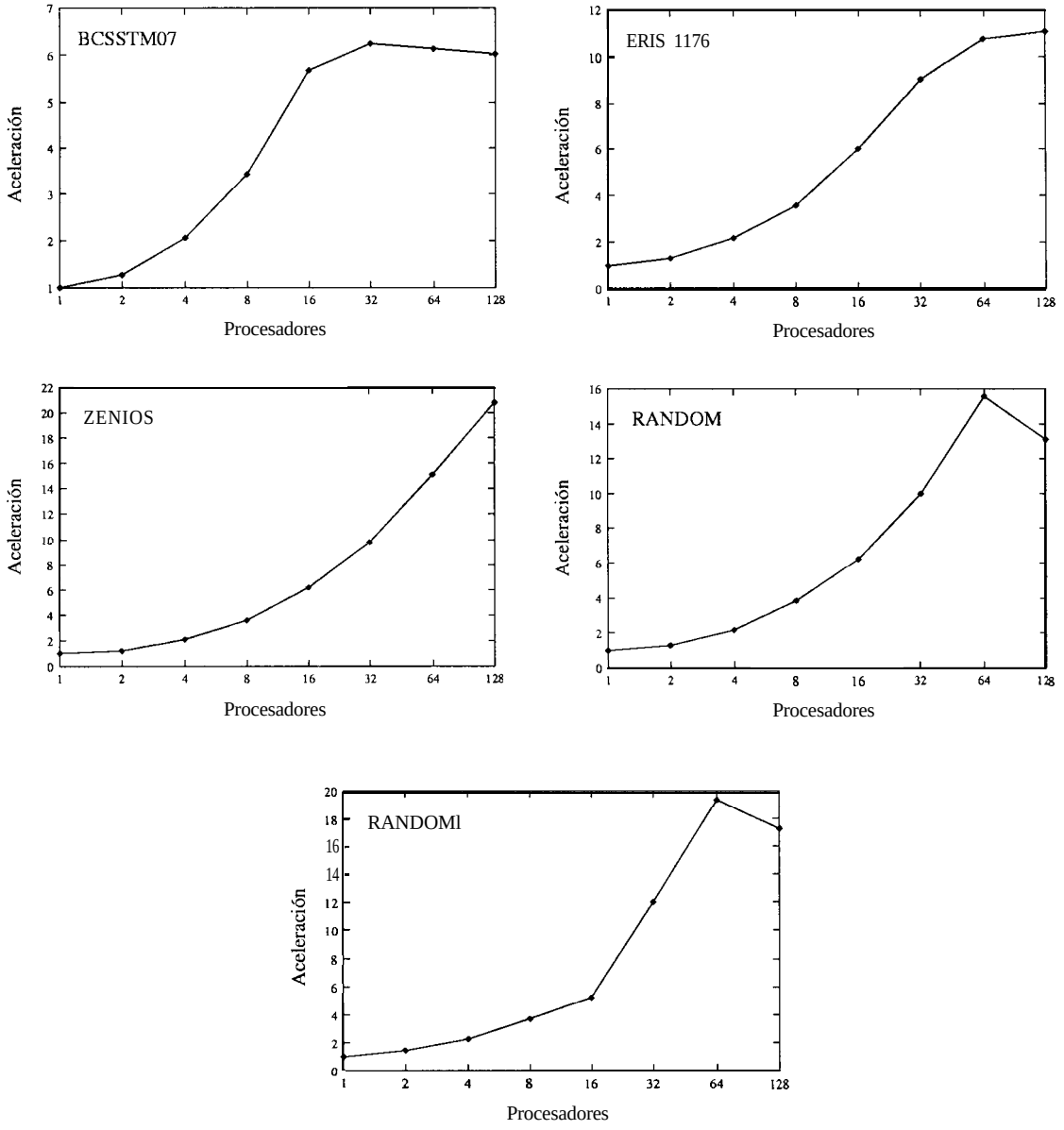

Figura 4.15: *Aceleración para las matrices prueba*

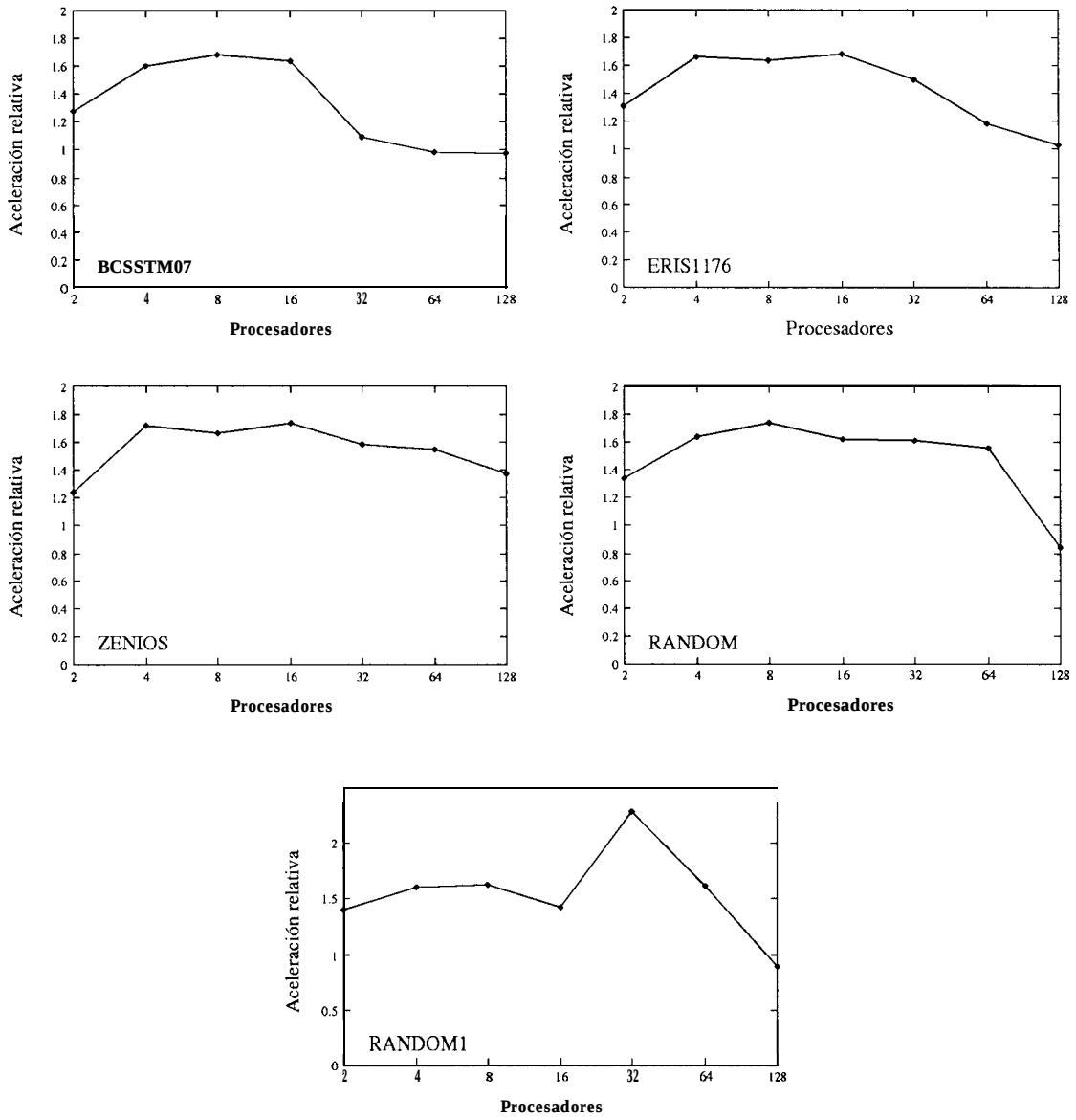

Figura 4.16: Aceleración relativa para las matrices prueba

MATRIZ	SECUENCIAL	PARALELO
BCSSTM07	2.22	2.67
ERIS 1176	13.54	15.20
ZENIOS	21.28	23.17
RANDOM	16.72	17.72
RANDOM1	109.71	113.63

Tabla 4.2: ***Tiempos de ejecución del programa secuencial y del programa paralelo sobre un procesador***

En la Figura 4.17 se representa la aceleración obtenida para cada una de las matrices utilizando las subrutinas nativas del AP1000 y las comunicaciones de baja latencia, las subrutinas nativas sin utilizar las comunicaciones de baja latencia y la librería MPI versión 1.4.3. En el caso de la librería MPI versión 1.4.3 las comunicaciones de baja latencia no están disponibles, sin embargo hay que destacar que la eficiencia es peor a la obtenida para las implementaciones con las funciones nativas sin esta utilidad. Esto es debido a la alta latencia del MPI para las comunicaciones punto a punto. En [103] se calcula una estimación de la latencia obteniéndose los valores de 246 microsegundos para Cellos mientras que en MPI tiene un valor de 318 microsegundos. Para este mismo tipo de comunicaciones también fueron medidas las velocidades de transferencia obteniéndose los valores de 2.76 MB/s para Cellos y 2.79 MB/s para MPI.

En nuestro caso lo que más va a afectar al rendimiento de nuestro programa es la latencia, debido a que la mayoría de las comunicaciones que se realizan son mensajes de corta longitud. La Tabla 4.3 muestra el número total de mensajes enviados en el programa paralelo (Nm) y el tamaño medio de estos mensajes ($\overline{\text{longitud}}$) en Bytes para diferente número de procesadores.

4.7 Algoritmos paralelos según una distribución unidimensional de los datos

En la sección anterior llegamos a la conclusión de que el algoritmo paralelo más eficiente es aquel en el cual cada procesador almacena columnas enteras de la matriz, ya que se corresponde con la configuración que genera un menor número de comunicaciones entre procesadores. Estudiamos ahora diferentes orientaciones del algoritmo suponiendo que cada columna de la matriz es almacenada en uno y sólo uno de los procesadores según algún esquema de distribución previamente establecido. Es decir, nos vamos a centrar en un modelo de programación de grano medio (ver sección 2.5).

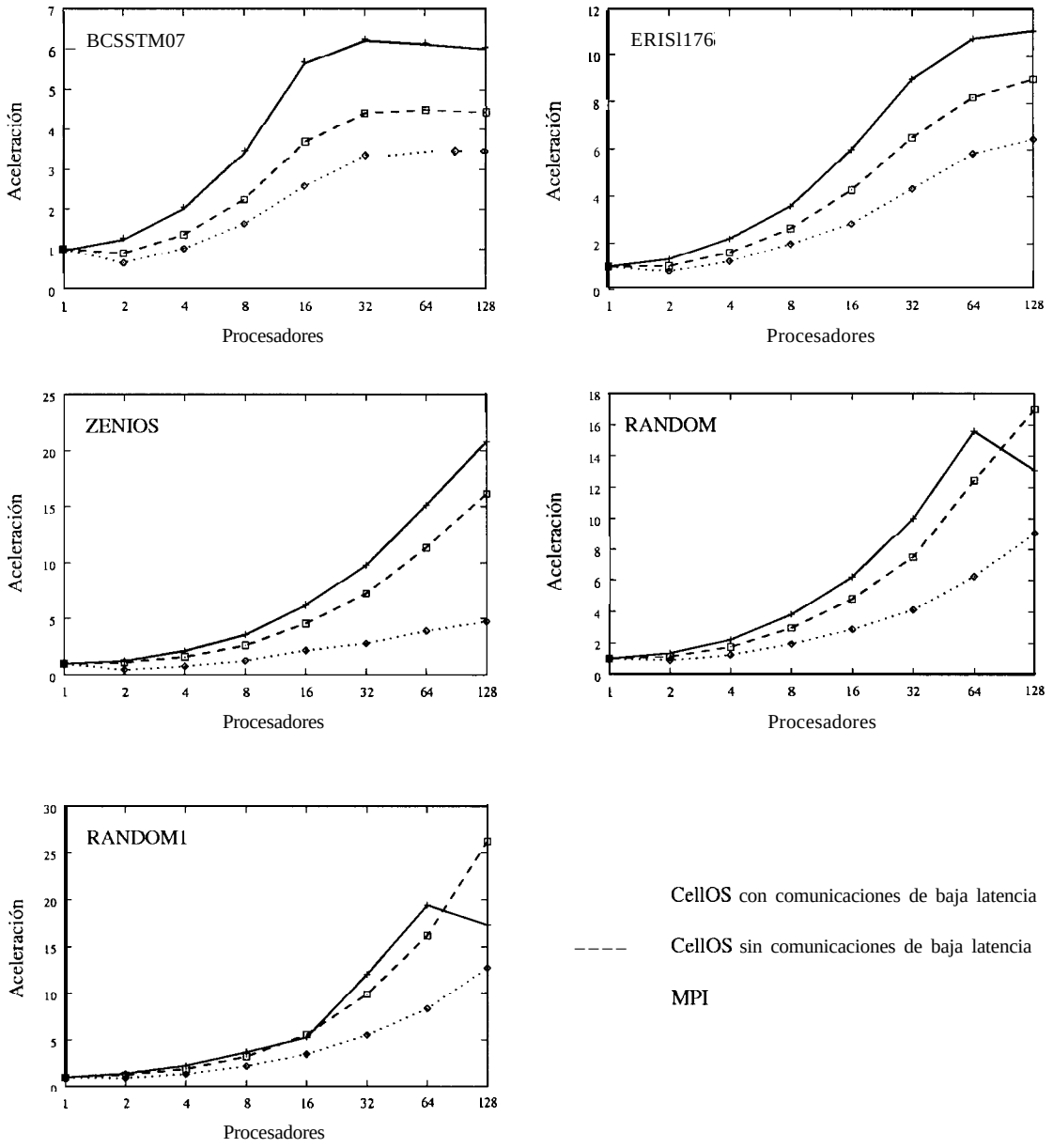

Figura 4.17: ***Comparación de las aceleraciones obtenidas con la librería nativa y con MPI***

P	BCSSTM07		ERIS 1176		ZENIOS	
	Nm	longitud	Nm	longitud	Nm	longitud
2	7243	281.19	25089	464.05	30087	59 1.57
4	ll443	267.20	38821	448.80	45622	577.77
8	14618	244.47	47921	423.77	56959	550.36
16	18167	211.16	55643	391.03	66880	505.92
32	23149	172.06	63671	352.98	77902	449.57
64	27305	148.88	73293	312.01	88690	401.80
128	27305	148.88	84851	272.01	99483	362.06

P	RANDOM		RANDOM1	
	Nm	longitud	Nm	longitud
2	15909	885.75	52456	1761.54
4	2435 1	867.87	79660	1741.32
8	29476	837.16	94902	1706.59
16	33528	787.84	105396	1646.03
32	38139	716.41	116051	1545.52
64	44414	626.20	130657	1395.65
128	52276	537.16	152343	1207.39

Tabla 4.3: Mensajes para las matrices prueba

La factorización de Cholesky estándar dispersa es típicamente implementada en paralelo asignando columnas enteras de la matriz a factor-izar sobre los diferentes procesadores. Las versiones paralelas de la factorización de Cholesky estándar de matrices dispersas según el modelo de programación de paralelismo de grano medio pueden ser clasificadas en dos tipos fundamentales: el método fan-out [36] y el método fan-in [l0], basados en el algoritmo *right-Zooking y* en el *Zeft-looking* respectivamente. Ya que el flujo de datos del algoritmo de Cholesky modificado es básicamente el mismo que en el algoritmo de Cholesky estándar, muchas de las ideas expuestas en estos trabajos pueden ser adaptadas para la implementación del algoritmo de Cholesky modificado paralelo.

En las siguientes secciones presentamos una adaptación de los algoritmosfun-out y *fan-in* a la factorización de Cholesky modificada, y variantes a dichos algoritmos que mejoran el rendimiento del programa paralelo. Se asume para la descripción de los diferentes algoritmos que cada columna de la matriz es inicialmente almacenada en uno y sólo uno de los procesadores según algún esquema de distribución previamente establecido. Finalizamos el capítulo con una breve referencia al método multifrontal.

4.7.1 Algoritmos fan-out

En el métodofun-out (FO) el proceso es dirigido por los datos disponibles. Cuando una columna j ya ha recibido todas las modificaciones de las columnas previas de las que depende, el procesador que contiene la columna j, calcula d_j y normaliza la columna. Tan pronto como finaliza esta tarea envía d_j *y* la columna j de L a todos los procesadores que la necesiten (incluyéndose el mismo) para modificar las columnas s (s > j) de las cuales son responsables. El procesador destino la recibe y realiza con ella todas las modificaciones oportunas. Cada columna es enviada una única vez a cada procesador aunque este la necesite para modificar más de una columna. Se mantiene en una cola las columnas que ya han recibido todas las modificaciones o que no necesitan ser modificadas por otras columnas. En estas columnas pueden ser calculadas las diagonales y se pueden actualizar sus valores con los valores finales de L. Por tanto las columnas no se modifican en orden si no en función de los datos disponibles.

Nótese que cuando se realiza un envío, se envía la columna ya actualizada con los valores de L, por tanto, se necesita enviar también la diagonal para hacer las modificaciones a columnas posteriores de la forma:

$$a_{sj} = a_{sj} - \sum_{k=O}^{j-1} l_{jk} l_{sk} d_k = a_{sj} - \sum_{k=O}^{j-1} l_{jk} a_{sk}; \quad s = j+1, \ldots n. \tag{4.1}$$

El algoritmo en forma simplificada se muestra en la Figura 4.18, donde $mycols(p)$ es el conjunto de índices de las columnas de las cuales el procesador p es responsable,

$ncol[p]$ es inicialmente la cardinalidad de $mycols(p)$, $nmod[s]$ es inicialmente el número de columnas j < s que modifican la columna s y $users(j)$ es el conjunto de índices de las columnas que son modificadas por j; $nmod[j]$ y $users(j)$ se pueden calcular previamente a la ejecución del algoritmo para la factorización numérica usando el árbol de eliminación de A.

```
for cada columna j ∈ mycols(p) con nmod[j] = 0 do
  computar d_j
  computar L_*j
  ncol[p] = ncol[p] − 1
  enviar d_j y L_*j a los procesadores responsable de las
      columnas con índices en users(j)
endfor
while ncol[p] > 0 do
  esperar por la recepción de una columna j
  for s ∈ mycols(p) ∩ users(j) do
    modificar la columna s
    nmod[s] = nmod[s] − 1
    if nmod[s] = 0 then
      computar d_s
      computar L_*s
      ncol[p] = ncol[p] − 1
      enviar d_s y L_*s a los procesadores responsable de
        las columnas con índices en users(s)
    endif
  endfor
endwhile
```

Figura 4.18: *Método fan-out para el procesador p (FO)*

En el métodofan-out se envían columnas enteras a los procesadores que las necesitan, y los productos del lado derecho de la Ecuación 4.1 se computan en el procesador destino. Si se envía d_j y L_{*j} a las columnas que los necesitan en vez de a los procesadores que las contienen, entonces los productos del lado derecho de la Ecuación 4.1 pueden ser calculados por el procesador responsable de la columna j, el cual envía a cada columna s de $users(j)$ el vector:

$$< j_s >= \{l_{sj} l_{kj} d_j\}_{k \in \{s+1,\ldots,n\}}$$

En este caso el número de comunicaciones entre procesadores es mucho mayor que en el

```
for cada columna j ∈ mycols(p) con nmod[j] = 0 do
   computar d_j
   computar L_*j
   ncol[p] = ncol[p] − 1
   computar y enviar productos no locales
   for s ∈ mycols(p) ∩ users(j) do
      computar < js > y modificar columna s
      nmod[s] = nmod[s] − 1
   endfor
endfor
Continúa . . . .
```

Figura 4.19: Método *fan-out con* pre-multiplicación *para el* procesador p (FOPM)

algoritmo FO. Si una columna modifica varias columnas dentro de un mismo procesador destino, será necesario enviar varios mensajes, en concreto, uno por cada columna. Si $s \in users(j)$, p(s) recibe $|mycols(p(s)) \cap users(j)|$ vectores, etiquetado cada uno con el índice de la columna destino. En contrapartida, el procesador destino no tiene que llevar a cabo la computación correspondiente a la Ecuación 4.1, incrementándose el solape entre computaciones y comunicaciones, lo cual reduce los tiempos de espera.

Este nuevo algoritmo, al que hemos denominado método *fan-out* con pre-multiplicación (FOPM), es descrito en las Figuras 4.19 y 4.20. En él, se distingue entre productos locales y no locales, entendiendo como productos no locales los vectores $< js >$ para los cuales $s \notin mycols(p)$. Solamente los productos no locales son enviados, evitando el envío de mensajes dentro del mismo procesador. En consecuencia, será necesaria la inclusión de un cierto mecanismo de control, ya que si la modificación de la columna s por la columna $j \in mycols(p(s))$ finaliza antes de que la columna j ha finalizado la modificación de todas las columnas $s^* \in mycols(p(j)) \cap users(j)$, entonces la columna s se añade a una cola de columnas listas para ser normalizadas y utilizadas para modificar columnas posteriores.

4.7.2 Algoritmos fan-in

El principal problema del algoritmo *fan-out es* el elevado volumen de comunicaciones que genera. El número y volumen de comunicaciones se puede reducir si las modificaciones a una columna s por todas las columnas j pertenecientes a un mismo procesador p(j) son acumuladas antes de ser enviadas desde p(j) a p(s). En este caso no se envía una columna para modificar varias, sino que se envían varias columnas acumuladas en

```
(Continuación . . . .
\While ncol[p] > 0 d o
   esperar un mensaje para modificar una columna j ∈ mycols(p)
   modificar j
   nmod[j] = nmod[j] − 1
   if nmod[j] = 0 then
     computar d_j
     computar L_*j
     ncol[p] = ncol[p] − 1
     computar y enviar productos no locales
     for s ∈ mycols(p) ∩ users(j)
       computar < js > y modificar columna s
       nmod[s] = nmod[s] − 1
       if nmod[s] = 0 then
         añadir la columna s a la cola
       endif
     endfor
   endif
   while cola no vacía do
     conseguir una columna de la cola (llamémosle j)
     computar d_j
     computar L_*j
     ncol[p] = ncol[p] − 1
     computar y enviar los productos no locales
     for s ∈ mycols(p) ∩ users(j)
       computar < js > y modificar la columna s
       nmod[s] = nmod[s] − 1
       if nmod[s] = 0 then
         añadir la columna s a la cola
       endif
     endfor
   endwhile
(endwhile
```

Figura 4.20: ***Método fan-out con pre-multiplicación para el procesador p*** (FOPM) ***(continuación)***

```
or s = 1 to n do
  if s ∈ mycols(p) o 3 j ∈ mycols(p) ∩ suppliers(s) then
    u[p, s] = 0
    for j ∈ mycols(p) ∩ suppliers(s) do
      u[p, s] = u[p, s] + l_sj(l_sj, ..., l_nj)^T
    endfor
    if s ∈ mycols(p) then
      (l_ss, ..., l_ns)^T = (a_ss, . . . . a_ns)^T − u[p, s]
      while pmods[s] ≠ 0 do
        esperar por la recepción de un vector u[p*, s] desde
        otro procesador p*
        (l_ss, ..., l_ns)^T = (l_ss, ..., l_ns)^T − u[p*, s]
        pmods[s] = pmods[s] − 1
      endwhile
      computar d_s
      L_*s = L_*s/d_s
    else
      enviar u[p, s] a p(s)
    endif
  endif
endfor
```

Figura 4.21: ***Método** fan-in **para el procesador p** (FI)*

una para modificar una única columna. Esto es lo que se conoce como algoritmo fan-in (FI) [10]

El algoritmo en forma simplificada se muestra en la Figura 4.21, donde ***suppliers(s) es*** el conjunto de índices de las columnas j tales que $s \in users(j)$, $u[p,s]$ ***es*** el vector que acumula las modificaciones a la columna s llevadas a cabo por las columnas $j \in mycols(p)$, y $pmods[s]$ ***es*** el número de procesadores ***p*** que proporcionan modificaciones a la columna s.

En FI, las columnas son computadas en orden creciente de sus índices, resultando que una columna j no puede ser modificada hasta que todas las columnas $s, s < j$ han sido totalmente modificadas. Para relajar esta limitación, el siguiente algoritmo propuesto (Figura 4.22), al que hemos denominada algoritmo fan-in dirigido por los datos disponibles (FIDD), combina la reducción en el número y volumen de comunicaciones del algoritmo ***fan-in*** con el carácter ***data-driven*** del algoritmo ***fan-out.*** Las columnas

son modificadas tan pronto como es posible. La variable $nlmod[s]$ es inicializada al valor $|suppliers(s) \cap mycols(p)|$.

FIDD tiene el mismo número y volumen de comunicaciones que FI, pero con la estrategia data-driven se consigue reducir los tiempos de espera.

La ejecución de ambos algoritmos es acelerada si se utiliza un vector expandido, $u[p, s]$, para llevar a cabo la acumulación de las modificaciones a una columna s. De esta forma se eliminan referencias a vectores de índices y se evitan búsquedas a través de la columna. Cuando se han acumulado todas las modificaciones, el resultado final puede ser enviado en un formato comprimido junto con un vector de índices fila o en un formato no comprimido. Nosotros nos referiremos a la implementación de FIDD con mensajes $u[p, s]$ comprimidos como FIDD1, y a la implementación de FIDD con mensajes $u[p, s]$ no comprimidos como FIDD2.

4.8 Distribución de las columnas entre los procesadores

Las matrices han sido distribuidas cíclicamente por columnas sobre la red de procesadores. Cada procesador almacenará su matriz local por columnas utilizando el modo de almacenamiento comprimido CCS. Se ha optado por esta distribución debido a su simplicidad de implementación y el buen balanceo de la carga que proporciona.

Una alternativa a esta distribución es la asignación de columnas según el esquema subárbol-a-subcubo descrito en [43]. El mapeo subárbol-a-subcubo sigue una estrategia divide-y-vencerás basándose en el árbol de eliminación de la matriz y distribuyendo recursivamente el trabajo de la factorización entre los P procesadores. El problema de este método es que causa un enorme desbalanceo de la carga para arboles desbalanceados. En general, este esquema se ha demostrado efectivo para matrices procedentes de los problemas de elementos finitos. Ya que nosotros estamos interesados en estudiar la factorización dispersa paralela de propósito general, este método de distribución de la matriz no es apropiado.

En [32] Geist y Ng desarrollan una generalización del mapeo subárbol-a-subcubo con la finalidad de conseguir un buen balanceo de la carga, incluso para arboles de eliminación arbitrariamente desbalanceados. Su estrategia está también basada en la estructura del árbol de eliminación asociado a la matriz dispersa a factorizar. El procedimiento se puede describir en dos etapas. En la primera etapa se divide el árbol en tantas ramas como número de procesadores. En la segunda etapa, si la carga no está bien balanceada, se divide la rama con una mayor carga computacional en dos nuevas ramas sumando una de ellas a la rama más pequeña. Este proceso se aplica recursivamente hasta llegar a un balanceo de la carga aceptable.

```
while ncol[p] ≠ 0 do
  while cola no vacía do
    conseguir una columna de la cola (llamémosle j)
    computar d_j
    computar L_*j
    ncol[p] = ncol[p] − 1
    for s ∈ users(j) do
      nlmod[s] = nlmod[s] − 1
      if nlmod[s] = 0 then
        u[p, s] = 0
        for l ∈ mycols(p) ∩ suppliers(s) do
          u[p, s] = u[p, s] + l_sl(l_sl, ..., l_nl)^T
        endfor
        if s ∈ mycols(p) then
          (l_ss, ..., l_ns)^T = (a_ss, ..., a_ns)^T − u[p, s]
          pmods[s] = pmods[s] − 1
          if pmods[s] = 0 then
            añadir la columna s a la cola
          endif
        else
          enviar u[p, s] a p(s)
        endif
      endif
    endfor
  endwhile
  esperar por la recepción de un vector u[p*, j] (j ∈ mycols(p))
    desde otro procesador p*
  (l_jj, ..., l_nj)^T = (l_jj, ..., l_nj)^T − u[p*, j]
  pmods[j] = pmods[j] − 1
  if pmods[j] = 0 then
    añadir la columna j a la cola
  endif
endwhile
```

Figura 4.22: ***Método fan-in guiado por los datos disponibles para el procesador p (FIDD)***

4.9 Análisis de las comunicaciones para la distribución unidimensional

Analizamos ahora el número y volumen de comunicaciones implicados en cada uno de los algoritmos previamente descritos. El número de mensajes enviados durante la ejecución de los diferentes métodos depende del patrón de la matriz a factorizar y del mapeo de las columnas de la matriz sobre los procesadores. Suponemos para nuestro análisis que el número de columnas de la matriz, N, es al menos tan grande como el número de procesadores, ***P, y*** que las columnas de la matriz son distribuidas cíclicamente. Consideramos que α es el número de entradas de la matriz y, por tanto, $\frac{\alpha}{N}$ es el número promedio de entradas por columna.

Proposición 5 ***El número de mensajes entre procesadores para el algoritmo*** **FO es:**

$$M_{FO} = \sum_{j} |\{s\%P \quad / \quad s > j \quad a_{sj} \neq 0\}|$$

cumpliéndose que:

$$N \leq M_{FO} \leq (N-P)P + \frac{P(P-1)}{2}$$

siendo el tamaño de cada mensaje:

$$S_{FO} = \alpha_j$$

donde α_j ***es el número de elementos no nulos en la columna*** j ***de L.***

es decir

$$S_{FO} \leq max_j\{\alpha_j\}$$

y siendo su tamaño medio:

$$S_{FO} = \frac{\alpha}{N}$$

Demostración:

- Se envía cada columna a los procesadores que la necesitan. Esto es, la columna j es enviada a todos aquellos procesadores (incluyéndose a si mismo) que contienen columnas s, (s > j) para las cuales $a_{sj} \neq 0$. Por tanto, el número de mensajes generados por esa columna será:

$$|\{s\%P \ a_{sj} \neq 0\}|.$$

- En el mejor caso, todas las columnas s pertenecen al mismo procesador y, por tanto, sólo se envía un mensaje por cada columna, con lo cual:

$$N \leq M_{FO}.$$

- El peor caso se obtiene para el caso denso. En este caso, todas las entradas a_{sj} con $s > j$ son distintas de cero, y por tanto las primeras N − P columnas tienen que ser enviadas a todos los procesadores. A partir de ahí, el número de entradas de cada columna por debajo de la diagonal es menor que el número de procesadores, y por tanto, el número de procesadores al que tendrán que ser enviadas va disminuyendo en una unidad:

$$M_{FO} \leq (N-P)P + \sum_{i=0}^{P-1} i = (N-P)P + \frac{P(P-1)}{2}.$$

- En cuanto al tamaño de los mensajes, lo que se envía es una columna, y por tanto el tamaño vendrá dado por el número de entradas en cada columna. □

Proposición 6 ***El número de mensajes enviados entre procesadores para el algoritmo*** **FOPM es:**

$$M_{FOPM} = \sum_{j} |\{s > j \ / \ a_{sj} \neq 0 \ \ y \ \ s\%P \neq j\%P\}|$$

cumpliéndose que:

$$0 \leq M_{FOPM} \leq \alpha - N$$

para el caso denso:

$$0 \leq M_{FOPM} \leq \frac{N(N-1)P-1}{2 \quad P}$$

y siendo el tamaño del mensaje enviado por la columna j ***a la columna s:***

$$S_{FOPM} = |\{k \ / \ a_{kj} \neq 0 \quad \textbf{\textit{und}} \quad k \geq s\}|$$

cumpliéndose que:

$$S_{FOPM} \leq max_j\{\alpha_j - 1\}$$

siendo el tamaño medio estimado:

$$\overline{S_{FOPM}} = \frac{\alpha}{2N}$$

Demostración:

- Para cada columna j, se realizan los productos de cada entrada a_{kj} por debajo de la diagonal (k > j) por todas las entradas a_{sj}, con $s \geq k$. Estos productos se empaquetan y se envían al procesador p = $k\%P$. Por tanto, por cada entrada por debajo de la diagonal se envía un mensaje, de todos ellos habrá que descontar los que van dirigidos a sí mismo:

$$|\{s > j \quad / \quad a_{sj} \neq 0 \quad y \quad s\%P \neq j\%P\}|.$$

- En el mejor caso, todas las entradas por debajo de la diagonal generan mensajes dirigidos al propio procesador cumpliéndose, por tanto, $0 \leq M_{FOPM}$.

- En el peor caso, todas las entradas por debajo de la diagonal generan un mensaje dirigido a otro procesador, por tanto, $M_{FOPM} \leq \alpha - N$.

- En el caso denso todas las entradas de la matriz por debajo de la diagonal generan un mensaje, de los cuales habrá que excluir los dirigidos a si mismo:

$$M_{FOPM} \leq \frac{N(N-1)}{2} - \frac{N(N-1)}{2P} = \frac{N(N-1)}{2}\frac{P-1}{P}.$$

- En cuanto al tamaño, la entrada a_{sj} modifica la diagonal en la columna s, por tanto, todas las entradas por debajo (a_{kj}, $k > s$) modificarán las entradas bajo la diagonal de la columna s:

$$S_{FOPM} = |\{k \; / \quad a_{kj} \neq 0 \quad y \quad k \geq s\}|.$$

- Suponiendo que el número promedio de entradas por columna es $\frac{\alpha}{N}$ y teniendo en cuenta que el tamaño de los mensajes enviados dentro de una columna varía entre $\frac{\alpha}{N} - 1$ y 1, (tamaños correspondientes a la primera entrada bajo la diagonal y última entrada de la columna respectivamente), entonces tendremos:

$$\overline{S_{FOPM}} = \frac{\sum_{i=1}^{\frac{\alpha}{N}-1} i}{\frac{\alpha}{N}} = \frac{\alpha}{2N} \; \square$$

Proposición 7 ***El número de mensajes enviados durante la ejecución de*** **FI es:**

$$M_{FI} = \sum_{p=1}^{P} |\{s \,/\, a_{sj} \neq 0, j\%P = p, s\%P \neq p\}|$$

cumpliéndose que:

$$\mathbf{0} \leq M_{FI} \leq \frac{P(2N - P - 1)}{\mathbf{2}} \; \frac{P - 1}{\mathbf{P}}$$

siendo el tamaño del mensaje enviado por el procesador p al procesador q para *modificar* ***la columna*** s ***(s*** $\in$ ***q):***

$$S_{FI} = |\{k \,/\, a_{kj} \neq 0, con\ k \geq s, j \in \{1, \ldots . s-1\} (j\%P = p\ y\ a_{sj} \neq 0)\}|$$

cumpliéndose que:

$$S_{FI} \leq max_j\{\alpha_j\}$$

Demostración:

- Cada procesador acumula las modificaciones a cada columna, de forma que el número de mensajes enviados se corresponde con el número de columnas que modifica cada procesador, siendo este número igual al número de entradas situadas en filas diferentes dentro de cada procesador, excluyendo aquellas entradas a_{sj} para las cuales $s\%P$ = p, es decir, excluyendo aquellas columnas que pertenecen al propio procesador. Por tanto, para cada procesador se envían el siguiente número de mensajes:

$$|\{s \,/\, a_{sj} \neq 0, j\%P = \text{P Y } s\%P \neq p\}|.$$

- En el mejor caso, todas las entradas a_{sj} pertenecientes a un procesador p, con $a_{sj} \neq 0$, cumplen que $s\%P$ **= *p,*** en cuyo caso M_{FI} **= *0.***

- El peor caso se corresponde con el caso denso, en el cual tenemos entradas en todas las filas. Si tenemos en cuenta que el número de entradas por columna va disminuyendo en una unidad a medida que avanzamos en el índice columna, el número de entradas con índice fila distinto en los ***P*** procesadores será:

$$\sum_{i=1}^{P} (N - i) = \frac{P(2N - P - 1)}{2}.$$

De todas ellas habrá que descontar las que cumplen $a_{sj} \neq 0$ y $s\%P$ **= *p, es*** decir, las que generan mensajes destinados al propio procesador. Por tanto:

$$M_{FI} \leq \frac{P(2N - \mathbf{P} - 1)}{2} \; \frac{\mathbf{P} - 1}{\mathbf{P}}.$$

- En cuanto al tamaño del mensaje, el mensaje enviado por un procesador p para modificar la columna s tendrá un tamaño igual al número de entradas $a_{kj} \neq 0$ con índice fila k ($k \geq$ s) distinto, y situadas en las columnas j que verifiquen $a_{sj} \neq 0$, ya que estas serán las entradas que modifiquen la columna s.
- Dado que el patrón de la columna destino s es un superconjunto del de la columna fuente j, el tamaño de un mensaje destinado a la columna s no podrá ser nunca mayor que α_s y por tanto:

$$S_{FI} \leq max_s\{\alpha_s\}. \quad \square$$

Corolario 4 ***Para matrices grandes, en el caso denso, el número de mensajes generados por el algoritmo FOPM es mayor que el número de mensajes generado por el algoritmo FO***

Demostración:

- Si suponemos que N >> ***P,*** entonces, por un lado:

$$M_{FO} \leq (N-P)P + \frac{P(P-1)}{2} \simeq NP$$

mientras que:

$$M_{FOPM} \leq \frac{\text{N (N - l) P - 1}}{\boldsymbol{2} \qquad \boldsymbol{P}} \simeq \frac{N^2}{2}\frac{P-1}{\boldsymbol{P}}$$

Por tanto, $M_{FO} < M_{FOPM}$.□

Corolario 5 ***Para matrices grandes, en el caso denso, el número de mensajes generado por el algoritmo FI es menor al número de mensajes generado por el algoritmo FO***

Demostración:

- Suponiendo de nuevo que N >> ***P:***

$$M_{FI} \leq \frac{P(2N-P-1)}{\boldsymbol{2}}\frac{P-1}{\boldsymbol{P}} \simeq NP\frac{P-1}{\boldsymbol{P}} \leq NP \simeq M_{FO}. \quad \square$$

Corolario 6 ***Para una columna dada s se cumple que*** $S_{FIDD2} \geq S_{FIDD1}\frac{N-s}{\alpha_s}$

Demostración:

- Por un lado, $S_{FIDD1} = S_{FI} \leq \alpha_s$.
- Por otro lado, para $FIDD2$ enviamos un vector denso del tamaño de la columna, es decir, S_{FIDD2} = N − s. Por tanto:

$$S_{FIDD2} \geq S_{FIDD1}\frac{\text{N - s}}{\alpha_s} \quad \square$$

4.9.1 Comunicaciones para matrices prueba

Hemos utilizado el mismo conjunto de 5 matrices que para el caso de distribución bidimensional de la matriz (ver Tabla 4.1) para comparar el número total y volumen de comunicaciones necesarias para llevar a cabo la factorización por cada uno de los métodos descritos.

La Figura 4.23 muestra el número total de mensajes implicados en el proceso de factorización como función del número de procesadores para cada una de las matrices y para cada uno de los algoritmos. Como es natural, los diferentes algoritmos fan-in son los que generan un menor número de mensajes. El algoritmo FOPM es el que requiere un mayor número de mensajes para todas las matrices. Sin embargo, este número incrementa más rápidamente para el algoritmo FO que para el algoritmo FOPM cuando se aumenta el número de procesadores. Se puede ver que al aumentar el número de procesadores M_{FOPM} *se* aproxima a su valor límite, es decir, a $\alpha -$ N. Además, el número de mensajes de ambos algoritmos tiende a ser igual a partir de un cierto número de procesadores.

Las diferencias entre los diferentes métodos fan-in se encuentran en el incremento en el volumen de comunicaciones cuando se envía el vector de acumulación de forma no comprimida (algoritmo FIDD2) tal y como se muestra en la Figura 4.24. Este incremento es especialmente claro para la matriz ZENIOS.

A pesar de que $\overline{S_{FOPM}} < S_{FO}$, el volumen de comunicaciones generado por *FOPM es* mayor que el generado por FO para P pequeño, debido a que el número de mensajes totales es mucho mayor para el algoritmo *FOPM* que para el algoritmo FO. Sin embargo, y dado que a medida que aumentamos el número de procesadores el número de mensajes generado por *FO y FOPM se* aproxima, para un número de procesadores grande el volumen generado por *FOPM es* menor que el generado por FO.

4.10 Resultados experimentales para la distribución unidimensional

Hemos implementado los algoritmos descritos sobre los computadores paralelos de memoria distribuida AP1000 de Fujitsu y Cray T3E. Ambas son máquinas con topología malla y memoria distribuida pero con relaciones entre velocidad del procesador y velocidad de las comunicaciones muy diferentes, 1.25 FLOPs/byte en el T3E frente sólo 0.22 FLOPs/byte para cálculos en doble precisión en el AP1000. Las características más importantes de ambos sistemas son resumidas en la Tabla 3.1. El AP1000 ha sido

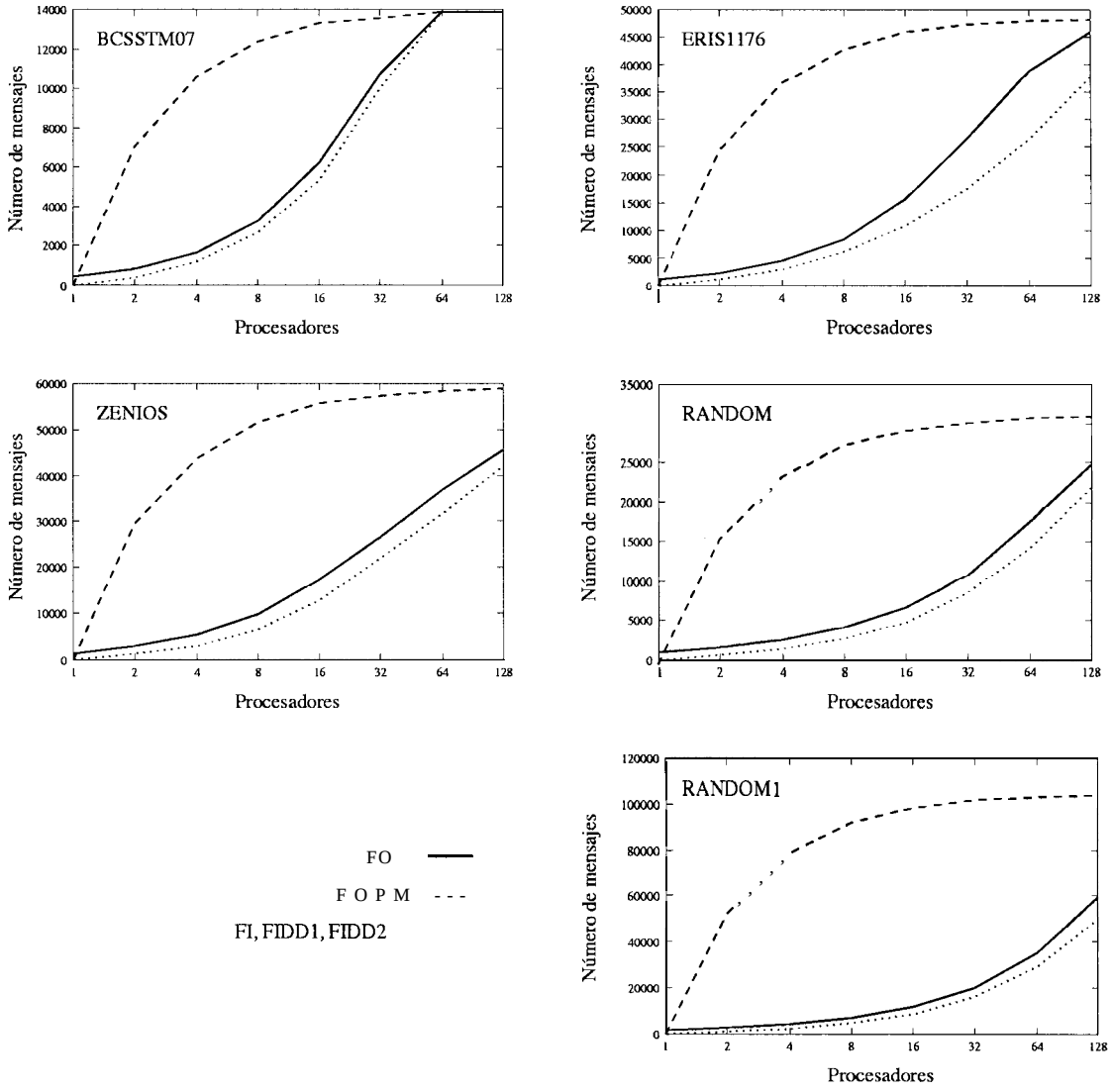

Figura 4.23: *Número de* mensajes *requeridos por los diferentes algoritmos*

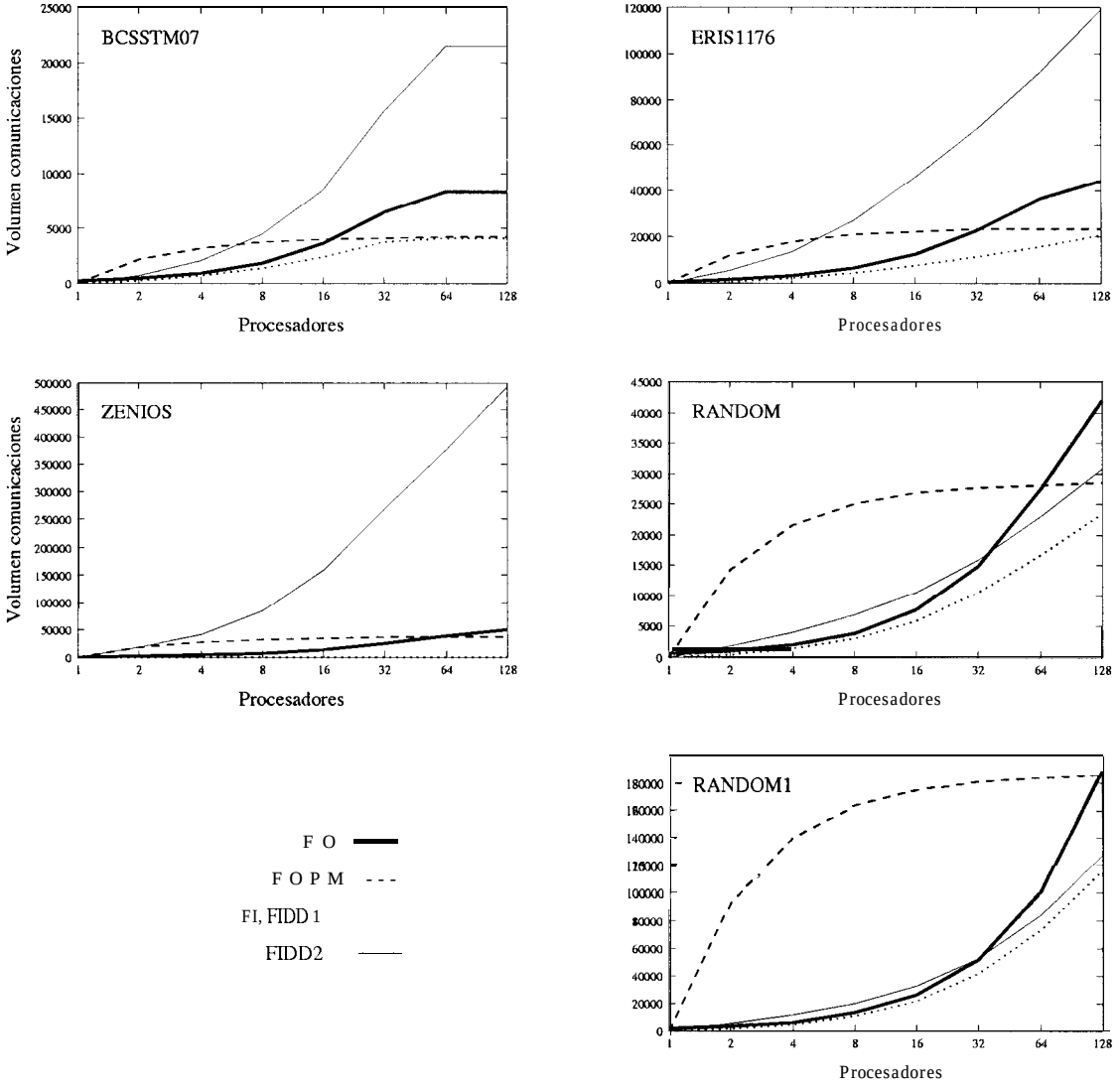

Figura 4.24: Volumen de comunicaciones en KB para cada algoritmo

programado usando las rutinas nativas del AP1000 como librería de pase de mensajes y el T3E ha sido programado usando la librería estándar MPI. En el caso del AP1000 disponíamos de hasta 128 procesadores, en el caso del T3E, sólo tuvimos acceso a 16. Todos los programas han sido implementados en doble precisión.

Se han medido los tiempos de espera y las aceleraciones obtenidas para cada uno de los métodos y para cada una de las matrices de la Tabla 4.1.

Las Figuras 4.25 y 4.26 muestran, para el AP1000 y el T3E respectivamente, el tiempo total de espera del procesador más lento durante la factorización de cada una de las matrices y para cada algoritmo. Para la mayoría de las matrices, el tiempo de espera para FOPM permanece constante o decrece para más de 4 procesadores; para un número de procesadores suficientemente alto se puede ver que los tiempos de espera del algoritmo FOPM son siempre menores que los correspondientes al algoritmo FO tal y como se pretendía (el pequeño número de procesadores disponibles en el T3E evita comprobar esta diferencia para las matrices aleatorias sobre este sistema). En cuanto a los métodos fan-in, el algoritmo FIDDl presenta unos tiempos de espera ligeramente inferiores a los del algoritmo FI para todas las matrices excepto para la matriz ZENIOS sobre el AP1000.

Las Figuras 4.27 y 4.28 muestran las aceleraciones alcanzadas para cada matriz y para cada algoritmo sobre el AP1000 y el T3E respectivamente. No hemos considerado el tiempo de cómputo de los diferentes contadores necesarios como parte del tiempo de factorización. Teniendo en cuenta que el número de operaciones en punto flotante implicadas en la factorización de las matrices prueba es relativamente pequeño, y que tenemos una limitación importante al rendimiento paralelo impuesta por el camino crítico, las aceleraciones alcanzadas por estos algoritmos son bastante considerables, incluso sobre el T3E.

Tal y como cabía esperar, a partir de un determinado número de procesadores el algoritmo FI es generalmente más rápido que el FO sobre el T3E (la única excepción es la matriz ZENIOS), mientras que sobre el AP1000 el algoritmo FO es más rápido para un número pequeño de procesadores. A medida que aumenta el número de procesadores compensa la utilización del algoritmo FI.

Se observa que en general para el caso del AP1000 y a partir de un determinado número de procesadores se obtiene una mayor aceleración para el algoritmo FOPM que para el algoritmo FO a pesar de que el número de mensajes enviados es mayor ya que, en este caso, hay un mayor entrelazamiento entre comunicaciones y computaciones lo que se traduce en menores tiempos de espera.

En el caso del T3E el algoritmo FOPM funciona peor. Hay que tener en cuenta que el T3E tiene procesadores muy rápidos que hace que la utilización de comunicaciones ínter-procesador sea un factor crítico en el rendimiento del programa, por eso al aumen-

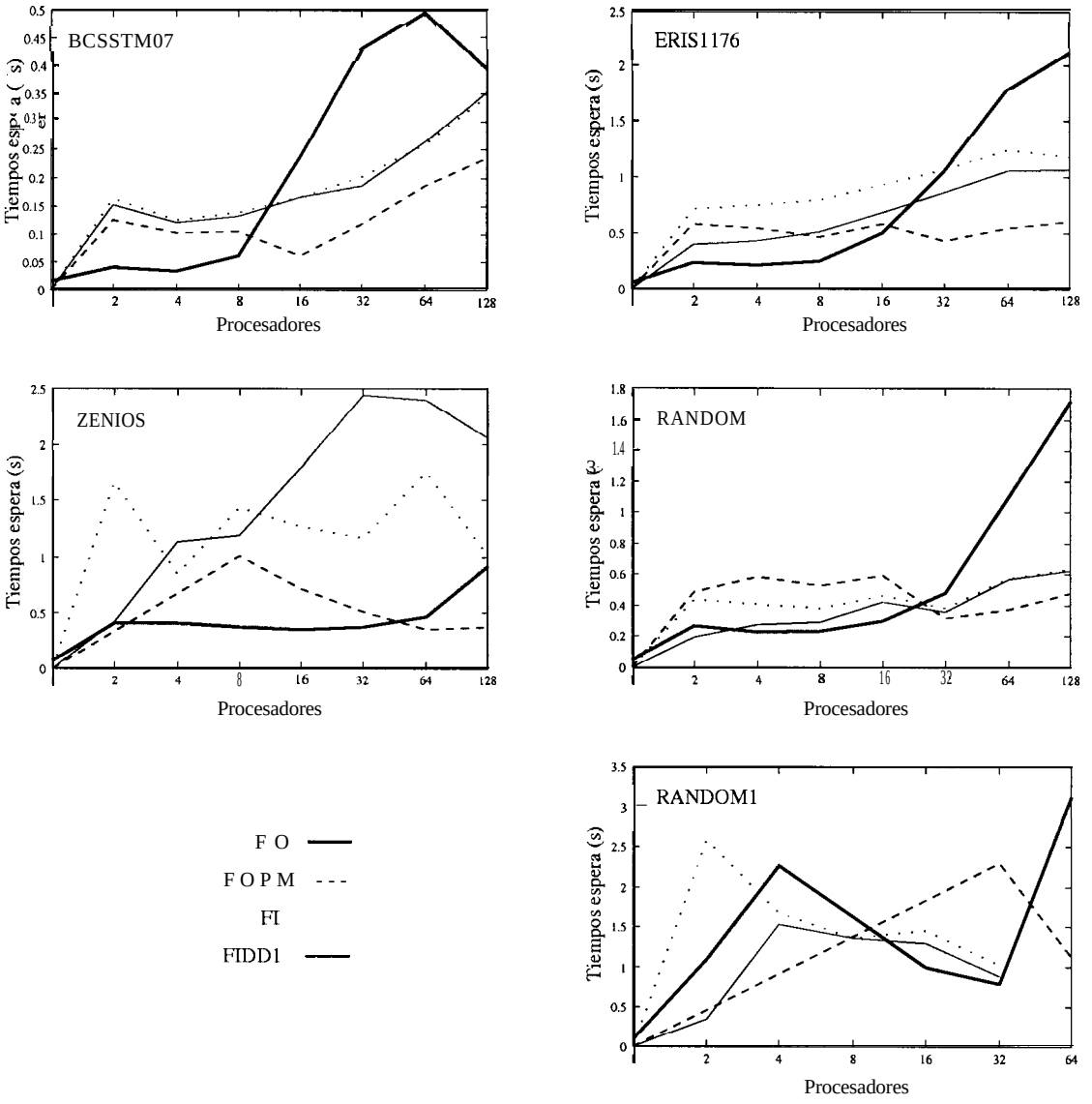

Figura 4.25: ***Tiempos de espera en el Fujitsu AP1O00***

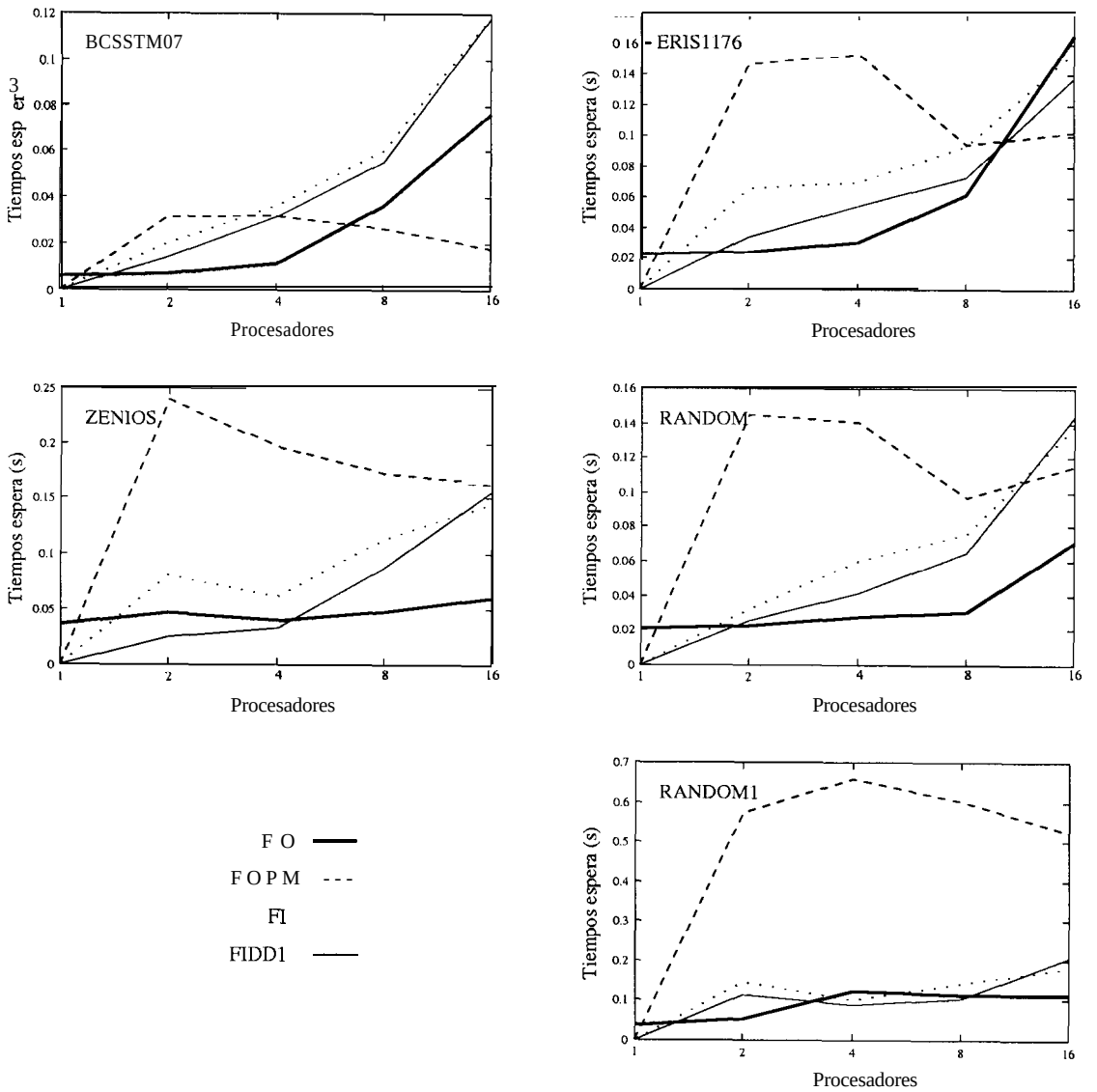

Figura 4.26: ***Tiempos de espera en el Cray*** T3E

tar el número de comunicaciones disminuye la aceleración. Incluso, en algunos casos, para 2 procesadores, se observa un aumento del tiempo de ejecución con respecto al programa secuencial debido al overhead introducido por las comunicaciones.

En cuanto a los métodos fan-in, no se aprecian muchas diferencias entre el algoritmo FI y el algoritmo FIDDl, aunque en la mayoría de los casos el algoritmo FIDDl es ligeramente superior. La escasa diferencia entre ambos es comprensible, ya que a las matrices prueba no se les ha aplicado ningún método de ordenamiento que elimine dependencias entre columnas además de disminuir el *fill-in* como puede ser el *nested dissection [52].* En este caso las dependencias entre columnas son muy fuertes, con lo cual, se saca poco partido al hecho de permitir la ejecución del programa de forma desordenada.

En cuanto al algoritmo FIDD2, en el caso del AP1OOO, la aceleración obtenida es superior a la del FI y del FIDD 1, y globalmente mejor para matrices grandes. En el caso del T3E, sólo compensa para matrices grandes. Recordar que en el T3E las comunicaciones son un hándicap muy importante, en cambio, en el AP1000, no es tan relevante en relación con la velocidad de los procesadores.

En cuanto a la comparación entre los métodos fan-in y *fun-out,* nótese que en general se obtienen mejores resultados para el algoritmofun-out. Esto nos indica que el número y volumen de las comunicaciones, aunque importante, no es crítico en este programa. El tiempo de ejecución está dominado por los tiempos de espera. Con el algoritmo fan-*out* estos tiempos de espera se compensan parcialmente adelantando computaciones, es decir, modificando todas aquellas columnas que se pueda aunque estas modificaciones no sean totales. La gran diferencia con el algoritmofun-in, es que este tipo de computaciones no se lleva a cabo hasta que se pueda calcular la modificación global, agravando así el problema de los tiempos de espera. A medida que aumenta el número de procesadores aumenta el número de comunicaciones necesario para llevar a cabo la factorización, es por esto, que a partir de un cierto número de procesadores puede ser más ventajoso el código correspondiente al algoritmofun-in ya que su principal característica es la reducción del número de comunicaciones. Esto se pone de manifiesto en los resultados correspondientes al AP1000. No es apreciable sobre el T3E ya que solamente disponemos de 16 procesadores. El mismo razonamiento puede ser hecho con respecto al tamaño de la matriz. A medida que aumenta el número de entradas, aumenta el posible número de comunicaciones y, por tanto, aumenta la importancia de su reducción Así, en la matriz RANDOMl, los resultados obtenidos para el algoritmofun-in mejoran a los obtenidos para *elfun-out* a partir de un número dado de procesadores.

Se puede concluir que los algoritmofun-in y fan-out modificados propuestos generalmente mejoran a los algoritmos originales excepto el algoritmo FOPM sobre el Cray T3E el cual se comporta peor que el algoritmo FO debido a la importancia de las comunicaciones sobre este sistema.

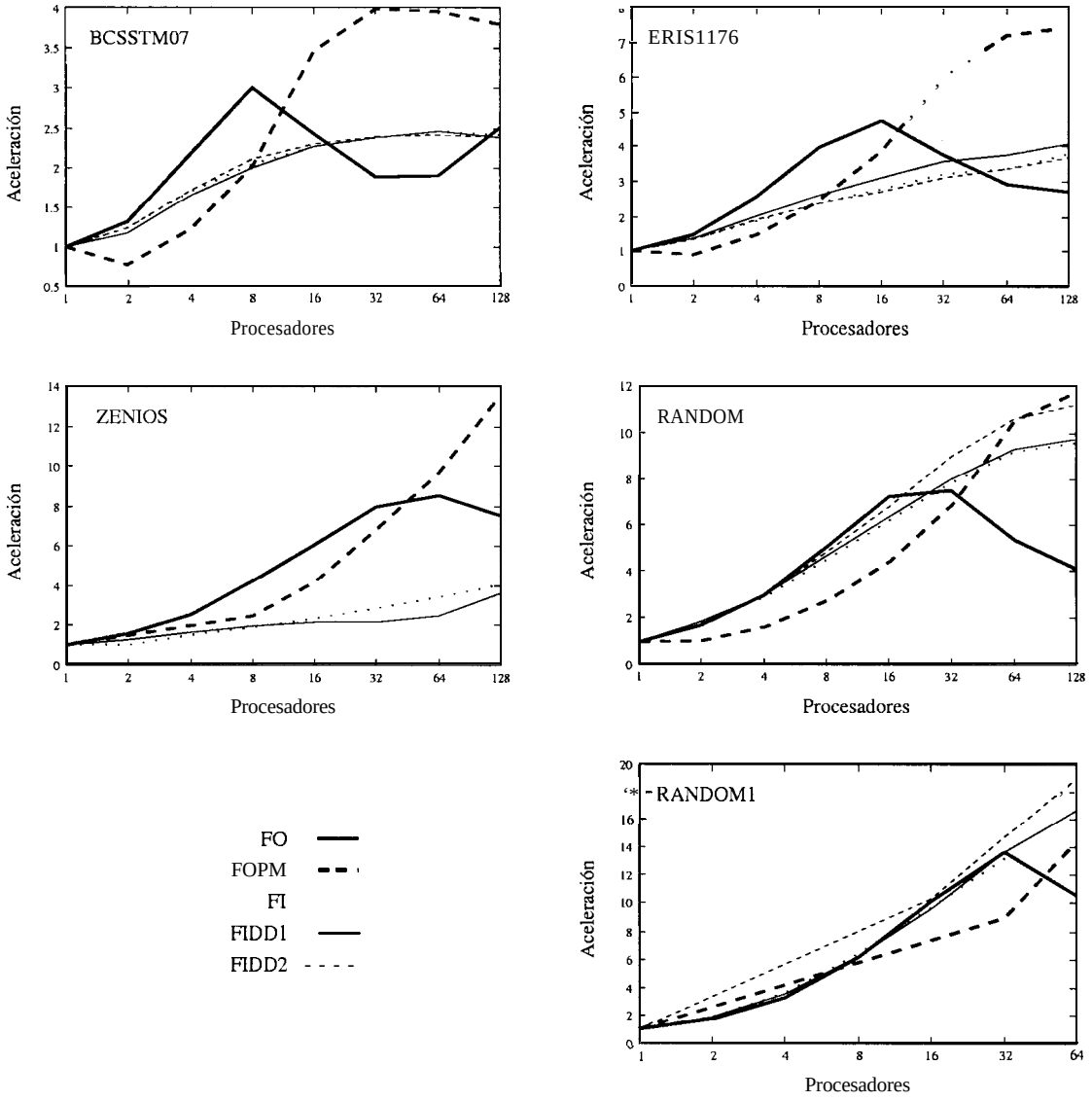

Figura 4.27: Aceleración en el *Fujitsu API 000*

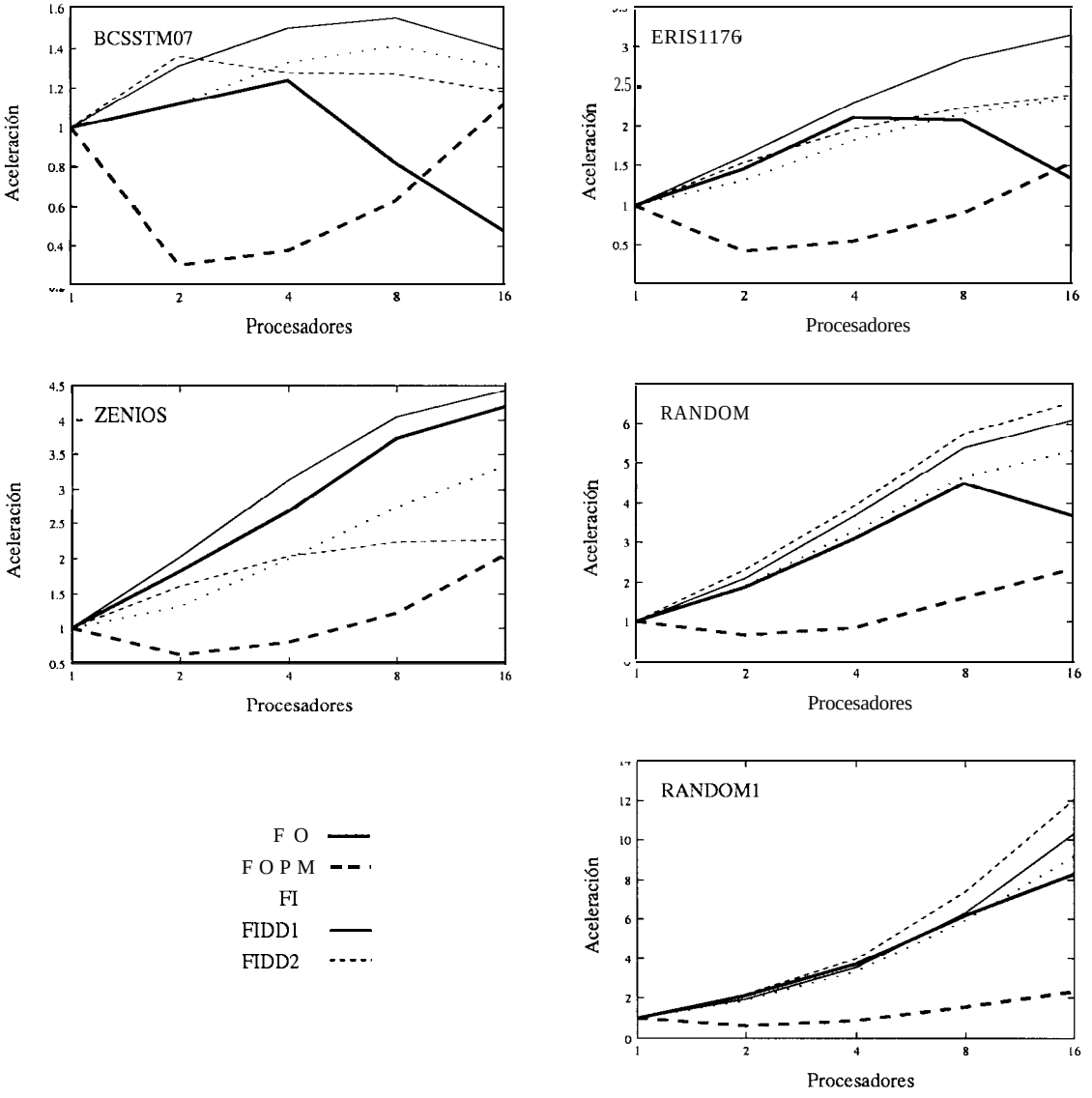

Figura 4.28: ***Aceleración en el Cray 7'3E***

4.11 Método multifrontal

El método multifrontal para la factorización de Cholesky desarrollado por Speelpening [104] y Duff y Reid [23] es esencialmente una variación del algoritmo right-looking y su aplicación es especialmente importante en aquellas máquinas en las que se dispone de unidades vectoriales. Su adaptación a la factorización de Cholesky modificada es extensamente explicada en [80].

Una implementación paralela del método multifrontal con resultados experimentales sobre el AP1000 y el T3E puede ser encontrada en [81]. En este trabajo nos centramos en la distribución de la carga sobre los procesadores teniendo en cuenta el árbol de eliminación de la matriz.

En general, dada una matriz dispersa A, *se* pueden representar sus dependencias por medio de un bosque compuesto de M arboles de eliminación. Cada nodo de dicho bosque tendrá asociada una carga computacional. Consideremos un sistema de memoria distribuida compuesto de P procesadores. Nosotros proponemos un algoritmo para distribuir los nodos sobre los procesadores dividido en dos etapas:

- En la primera etapa del algoritmo, se asigna un determinado número de procesadores a cada árbol de acuerdo con la carga computacional y el número de hojas del árbol.
- En la segunda etapa se realiza un particionamiento de los arboles a los que se les ha asignado más de un procesador. Utilizamos para ello la estrategia introducida por Tsay [106], generalizada para nodos con diferentes cargas computacionales, y utilizando lo que denominamos árbol de eliminación comprimido. En el árbol de eliminación comprimido cada nodo se corresponde con una secuencia de nodos sin ramificaciones del árbol de eliminación. Según la estrategia de distribución propuesta, los diferentes nodos que componen un nodo del árbol comprimido estarán asignados al mismo procesador.

4.12 Resumen

En este capítulo analizamos la paralelización de la factorización de Cholesky modificada sobre sistemas de memoria distribuida con un modelo de programación por pase de mensajes. Nos centramos, fundamentalmente, en el análisis de las comunicaciones, las cuales son el principal factor limitante del rendimiento en este tipo de arquitecturas.

Iniciamos el estudio planteando un código paralelo atendiendo a un modelo de programación de paralelismo de grano fino con el objetivo de explotar el máximo nivel

de paralelismo existente en el código. Distribuimos la matriz entre los procesadores según la distribución BCS. Elegimos esta distribución por ser una distribución general, aplicable a cualquier matriz independiente de su patrón, y porque su aplicación en la paralelización eficiente y simple de códigos irregulares está avalada por numerosos trabajos. Analizamos para este modelo las comunicaciones y comprobamos el buen balanceo de la carga a que dan lugar las matrices prueba utilizadas. Se llega a la conclusión de que el menor número de mensajes y, por tanto, el mayor rendimiento en el programa, se obtiene para el caso particular de que cada procesador almacene una columna entera de la matriz. Esto se corresponde con una granularidad de grano medio, es decir, se explota el paralelismo entre columnas de la matriz pero no dentro de cada columna.

Adaptamos los dos algoritmos básicos existentes para la factorización de Cholesky estándar y correspondientes a un nivel de paralelismo de grano medio a la factorización de Cholesky modificada: los algoritmos fan-in y fan-out. Proponemos además versiones de ambos algoritmos que reducen los tiempos de espera en los códigos paralelos. Analizamos las diferencias, en cuanto a comunicaciones se refiere, entre los diferentes algoritmos. Se muestran resultados experimentales sobre el AP1000 y el T3E.

Podemos concluir que el modelo de paralelismo de granularidad media es el más adecuado para los sistemas de memoria distribuida. Dentro de este modelo la elección entre la estrategia fan-in o fan-out, así como de las nuevas versiones propuestas, depende de las características concretas de la máquina en la que se van a ejecutar. Así, la modificación del algoritmo FO (algoritmo FOPM) es adecuada para aquellos sistemas en las cuales la diferencia entre velocidad de procesamiento y comunicaciones no es muy elevada. En aquellos sistemas en los cuales la utilización de comunicaciones ínter-procesador sea un factor crítico, será más conveniente utilizar los algoritmos que generen un menor número de comunicaciones, esto es, los algoritmos fan-in.

Capítulo 5

Paralelización sobre sistemas de memoria compartida tipo NUMA

Memoria compartida es considerado un paradigma atractivo ya que proporciona un modelo de programación simple y efectivo. Sin embargo, los sistemas con un único bus común no son escalables debido a que el ancho de banda del bus limita el número de procesadores que se pueden conectar. Esto ha motivado que los esfuerzos se centrasen en conseguir arquitecturas escalables basadas en diseños de multiprocesadores de memoria distribuida, dando lugar a los sistemas tipo NUMA (Non-Uniform Memory *Access).*

En este capítulo adaptamos los algoritmos right-looking y left-looking para su ejecución eficiente sobre una máquina de memoria compartida tipo NUMA. El principal factor limitante de la eficiencia de los códigos paralelos sobre sistemas NUMA es la localidad de la información, que juega un papel similar al de la distribución de los datos en los sistemas de memoria distribuida, y en ella centraremos la optimización del código paralelo. Utilizaremos para programar el sistema construcciones paralelas de bajo nivel tales como barreras de sincronización, regiones críticas, *locks,* semáforos, etc., las cuales nos permitirán un control detallado del programa.

5.1 Introducción

La arquitectura dominante para la próxima generación de multiprocesadores son los multiprocesadores de memoria compartida CC-NUMA (cache-coherent *non-uniform* memory architecture). Estas máquinas son atractivas debido a un acceso transparente a las memorias locales y remotas, sin embargo la latencia de acceso a memoria remota es de 3 a 5 veces la latencia de acceso a memoria local. Debido a ello la localidad

de los datos es potencialmente la característica más importante para conseguir buenos rendimientos.

En un sistema de memoria distribuida, el alto coste de las comunicaciones hace de ellas el factor predominante en el rendimiento del programa paralelo. En los sistemas multiprocesador con un espacio de direcciones compartido, y con una memoria físicamente distribuida, la migración de procesos y una pobre distribución de los datos puede hacer bajar la eficiencia a niveles subóptimos.

Sobre multiprocesadores tipo NUMA el comportamiento de las caches juega un papel clave en el rendimiento del programa paralelo ya que un fallo cache implicará, en muchos casos, un acceso a módulos de memoria remotos. Esto no ocurre en los sistemas de memoria distribuida, en los cuales los fallos cache se convierten en accesos locales y son las comunicaciones las que dan lugar a accesos remotos de memoria.

5.2 Algoritmos paralelos sobre sistemas tipo NUMA

Como se ha discutido en el capítulo 2, dependiendo del orden en el que se ejecuten las tareas habrá dos soluciones a la factorización de Cholesky modificada: el algoritmo right-looking y el algoritmo left-looking. En este capítulo adaptamos ambos algoritmos para su ejecución sobre sistemas de memoria compartida y proponemos nuevos *schedulings* basados en el árbol de eliminación de la matriz con el objetivo de aumentar la localidad de los datos y con ello el rendimiento. Suponemos en todos los casos que la matriz dispersa está almacenada según el modo de almacenamiento disperso CCS.

La obtención de una escalabilidad aceptable depende, en gran medida, de la distribución de los datos y el acceso que realicen a los mismos los distintos procesadores implicados en la ejecución del programa. Un efecto que se produce debido a una mala ubicación de los datos sobre la memoria distribuida de la máquina es el conflicto en el acceso a las líneas cache. Este fenómeno ocurre sólo para datos que son frecuentemente modificados y constituye una medida cuantitativa adecuada para caracterizar la localidad.

En el sistema 02000 de SGI cada procesador tiene asociada una memoria cache. Según el protocolo de coherencia, cuando un procesador modifica una línea cache cualquier otro procesador que posea una copia de la misma línea cache es notificado, invalidando su copia y solicitando una nueva copia cuando necesite el dato de nuevo. Esto produce problemas de rendimiento en dos situaciones:

- Un procesador actualiza repetidamente una línea cache que es utilizada en operaciones de lectura por otros nodos. En este caso, todos los nodos son forzados a tener una nueva copia de la línea y aumenta, por lo tanto, la latencia en el acceso

a los datos.

- Dos o más procesadores actualizan repetidamente la misma línea cache generándose una competencia por el acceso exclusivo a dicha línea. Cada procesador solicita una nueva copia de la línea cache antes de realizar su escritura, forzando a que la operación se realice en serie dado que los procesadores implicados en el conflicto deberán realizar las escrituras por turnos.

Para minimizar estos problemas, cada procesador debe operar sobre un conjunto de datos que tenga el menor número de elementos comunes con el resto de los procesadores. Idealmente, la estructura de datos utilizada en el programa debe ser distribuida en partes sobre las que opere de forma independiente cada procesador. Además, para obtener un buen balanceo de la carga, las distintas porciones de dicha distribución deben tener un tamaño similar. Así pues, este problema radica tanto en la implementación del algoritmo, como en la distribución que presenten los datos.

Si el conflicto se genera porque los procesadores‘acceden a un mismo dato se dice que se producen fallos de compartición verdaderos (true sharing). En ciertos casos esta situación no se puede evitar dado que los distintos procesadores que ejecutan una tarea paralela deben acceder a variables compartidas. Un ejemplo son las operaciones de sincronización para las que existe una disputa entre los distintos procesadores en el acceso a la misma variable de sincronización. Así pues, en una programa paralelo, este tipo de conflictos no puede ser eliminado completamente.

Si por otra parte, el conflicto se debe a que se accede a distintos datos que comparten la misma línea cache, entonces se dice que se producen fallos de compartición falsos (false sharing). Este último tipo de conflicto se puede eliminar a costa de un incremento en la cantidad de memoria utilizada, basta con separar los datos que lo producen y alocarlos en una línea cache diferente.

En nuestro caso, para evitar falsa compartición, todas las variables estrictamente locales de cada procesador son colocadas en estructuras que van asignadas a diferentes páginas de memoria y serán cargadas, por tanto, dentro de la memoria local del correspondiente procesador. En cuanto a los datos compartidos, creamos estructuras que son colocadas al inicio de las líneas cache. Las estructuras estarán formadas por datos que son modificados juntos. Para ello hemos incluido una macro que permite la reserva de memoria globalmente compartida y alineada con el tamaño de la línea cache secundaria. Esta macro ha sido incorporada al conjunto de macros de la librería parmacs que se ha utilizado para la implementación de los códigos paralelos.

En nuestros códigos, tanto en el caso del algoritmo *left-looking* como *right-looking*, la unidad de trabajo es una columna de la matriz. Todas las modificaciones se llevan a cabo sobre el array que mantiene los valores de cada columna de la matriz y

sobre el array que almacena el número de modificaciones que restan por realizar sobre cada columna, es decir, los arrays *DA y nmod* respectivamente. Ambos arrays son frecuentemente accedidos y modificados por diferentes procesadores provocándose muchos casos de falsa compartición. Para minimizar este efecto se ha modificado el formato de almacenamiento de la matriz. En concreto, los valores de la matriz no se almacenan en un array unidimensional, si no que se utiliza un array bidimensional de forma que cada columna se mapee al inicio de una línea cache. Se elimina con ello la falsa compartición en el acceso a los datos a costa de un ligero incremento en la memoria utilizada. En cuanto al array *nmod*, cuando una columna *j* se modifica, se actualiza también el contador *nmod[j].* Por tanto, es conveniente almacenar nmod[j] con los valores de la columna *j*. En la Figura 5.1 se muestra una matriz ejemplo y sus tres vectores en el modo de almacenamiento CCS. En la Figura 5.2 se muestra el nuevo modo de almacenamiento para esa matriz, donde cada vector es mapeado al inicio de una línea cache.

5.3 Análisis del rendimiento

El procesador MIPS R10000 incluye soporte *hardware* para evaluar el número de varios tipos de eventos que ocurren en el sistema como fallos cache, operaciones de coherencia en memoria, predicciones erróneas de saltos, etc. El diseño de estos contadores se ha realizado con la pretensión de captar el comportamiento dinámico del sistema. Nosotros hemos utilizado estos contadores a través de los programas de *interface* perfex, SpeedShop *y* prof, los cuales han sido introducidos en el capítulo 3, para evaluar el comportamiento del sistema de memoria. En concreto hemos medido las siguientes magnitudes:

- **Reuso de líneas cache en cache primaria: es** el número de veces, en promedio, que una línea cache de datos de la cache primaria es reusada una vez que ha sido cargada en la cache.

- **Reuso de líneas cache en cache secundaria**

- **Movimiento de datos entre la cache secundaria y la memoria compartida**

- **Mflops: Se** utiliza una aproximación que consiste en tomar el número de instrucciones en punto flotante en lugar del número de operaciones. De esta forma, en realidad, se está subestimando su valor.

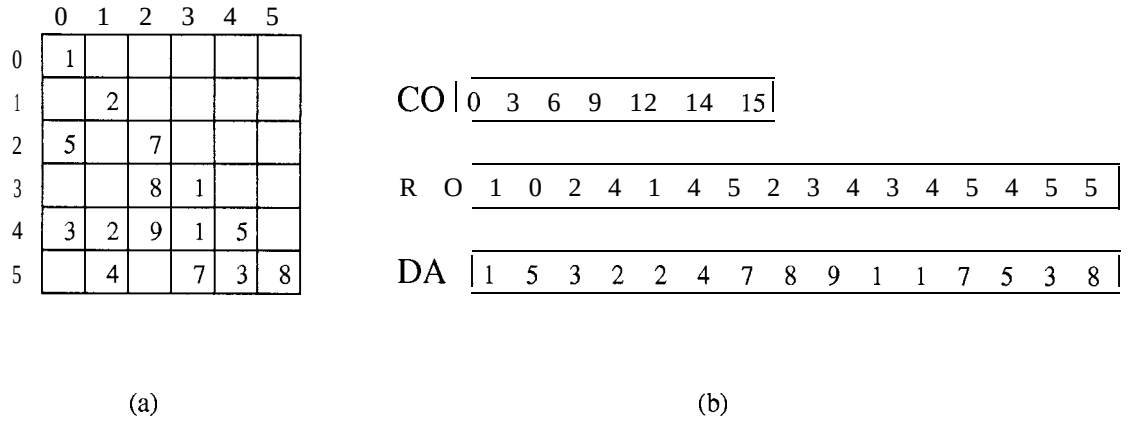

Figura 5.1: Matriz ejemplo: (a) estructura del factor ***L,*** (b) modo de almacenamiento CCS

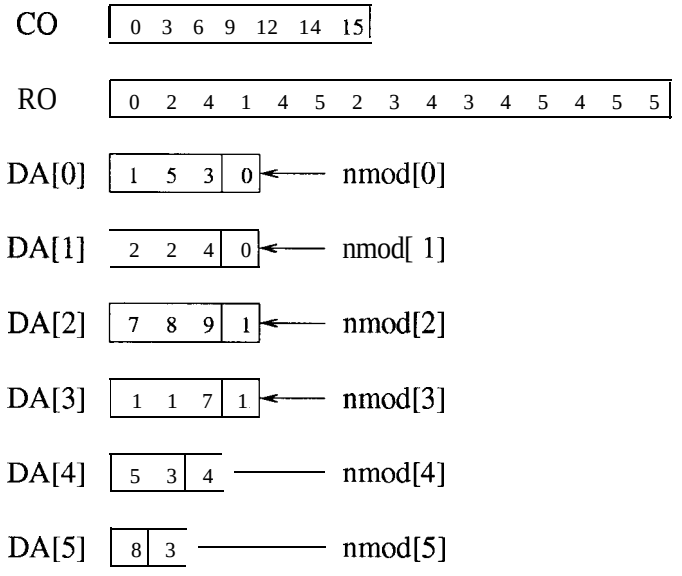

Figura 5.2: Modificación del modo de ***almacenamiento para la matriz ejemplo de la Figura 5.1***

```
1. while cola no vacía do
2.    conseguir una columna j de la cola
3.    calcular l_jj
4.    normalizar la columna j
5.    for each l_sj ≠ 0 (s > j) do
6.       modificar la columna s con la columna j
7.       nmod[s] = nmod[s] − 1
8.       if nmod[s] = 0 then
9.          añadir la columna s a la cola
10.      endif
11.   endfor
12. endwhile
```

Figura 5.3: *Algoritmo right-looking paralelo (RL)*

5.4 Algoritmo right-looking

En la versión *right-Zooking* del algoritmo de Cholesky estándar, una vez que una columna es normalizada, se emplea para modificar todas las columnas posteriores que dependen de ella. En este esquema está basada la propuesta paralela desarrollada por George et al. [37] para sistemas de memoria distribuida.

Proponemos la adaptación de este algoritmo para la factorización de Cholesky modificada y para su ejecución sobre sistemas de memoria compartida. En el programa paralelo cada tarea consiste en la computación de una columna, es decir, el cálculo de la diagonal, su normalización y la modificación de columnas posteriores. Con cada columna se mantiene un contador que almacena el número de modificaciones que quedan por realizar a esa columna. Además, una cola de tareas global y compartida mantiene todas las columnas cuyos contadores son cero. Estas columnas ya han recibido todas las modificaciones y por tanto están preparadas para ser normalizadas y modificar columnas posteriores. Cuando un procesador está libre se le asigna un índice columna de esta cola, calcula su diagonal, normaliza la columna y realiza nuevas modificaciones. Los contadores de las columnas destino son decrementados para reflejar estas modificaciones, si alguno de ellos llega a cero se coloca el índice columna correspondiente en la cola global de tareas disponibles. Un pseudo-código del algoritmo ejecutado sobre cada procesador se muestra en la Figura 5.3.

El patrón de acceso a la matriz está organizado atendiendo a las siguientes premisas:

- Una columna puede ser modificada por varios procesadores antes de ser colocada en la cola.
- Una vez colocada en la cola es accedida por un único procesador y usada por éste para modificar nuevas columnas.
- Después de que se han llevado a cabo todas las modificaciones esta columna no vuelve a ser referenciada por ningún procesador.

La única interación entre procesadores ocurre en el acceso a la cola global de tareas y en las modificaciones de una misma columna. Para evitar conflictos se utilizan operaciones ***LOCK*** implementadas en la librería parmacs.

El scheduling de tareas es totalmente dinámico, la cola de tareas se crea a la vez que se ejecuta el programa y los procesadores acceden a ella a medida que quedan libres, esto conduce a un buen balanceo de la carga.

El problema de este planteamiento es que más de un procesador puede modificar la misma columna y, por tanto, tendrá que acceder y escribir sobre la misma posición de memoria. Esto genera un elevado número de conflictos de acceso a memoria y de operaciones de mantenimiento de la coherencia, a la vez que un elevado número de fallos cache. Todo ello se traducen en un aumento en el número de accesos a memorias remotas, lo cual es una de las principales causas de rendimientos subóptimos en arquitecturas tipo NUMA.

5.4.1 Resultados experimentales

Para analizar el comportamiento de nuestros programas paralelos sobre el sistema de Silicon Graphics 02000 utilizamos un nuevo conjunto de matrices simétricas de la colección Harwell-Boeing [22]. El sistema 02000 dispone de una gran capacidad de memoria, es por esto que utilizamos matrices más grandes a las utilizadas en memoria distribuida, en la cual el sistema de memoria del AP1000, así como su ring ***buffer-,*** nos imponía una limitación en este sentido. La Tabla 5.1 muestra las características de las matrices usadas, donde N es el número de filas y columnas y α es el número de entradas no nulas. Todas ellas proceden de problemas de ingeniería estructural excepto la matriz LSHP3466 que procede de problemas de elementos finitos. Los patrones de estas matrices se muestran en la Figura 5.4. A todas ellas se les ha aplicado el algoritmo minimum ***degree*** para reducir el fill [9,4], los patrones tras aplicar este ordenamiento se muestran en la Figura 5.5.

El comportamiento del sistema de memoria nos da una aproximación a la localidad del programa. Por esta razón se analizará el código desde tres puntos de vista: tiempos de ejecución, balanceo de la carga y comportamiento del sistema de memoria.

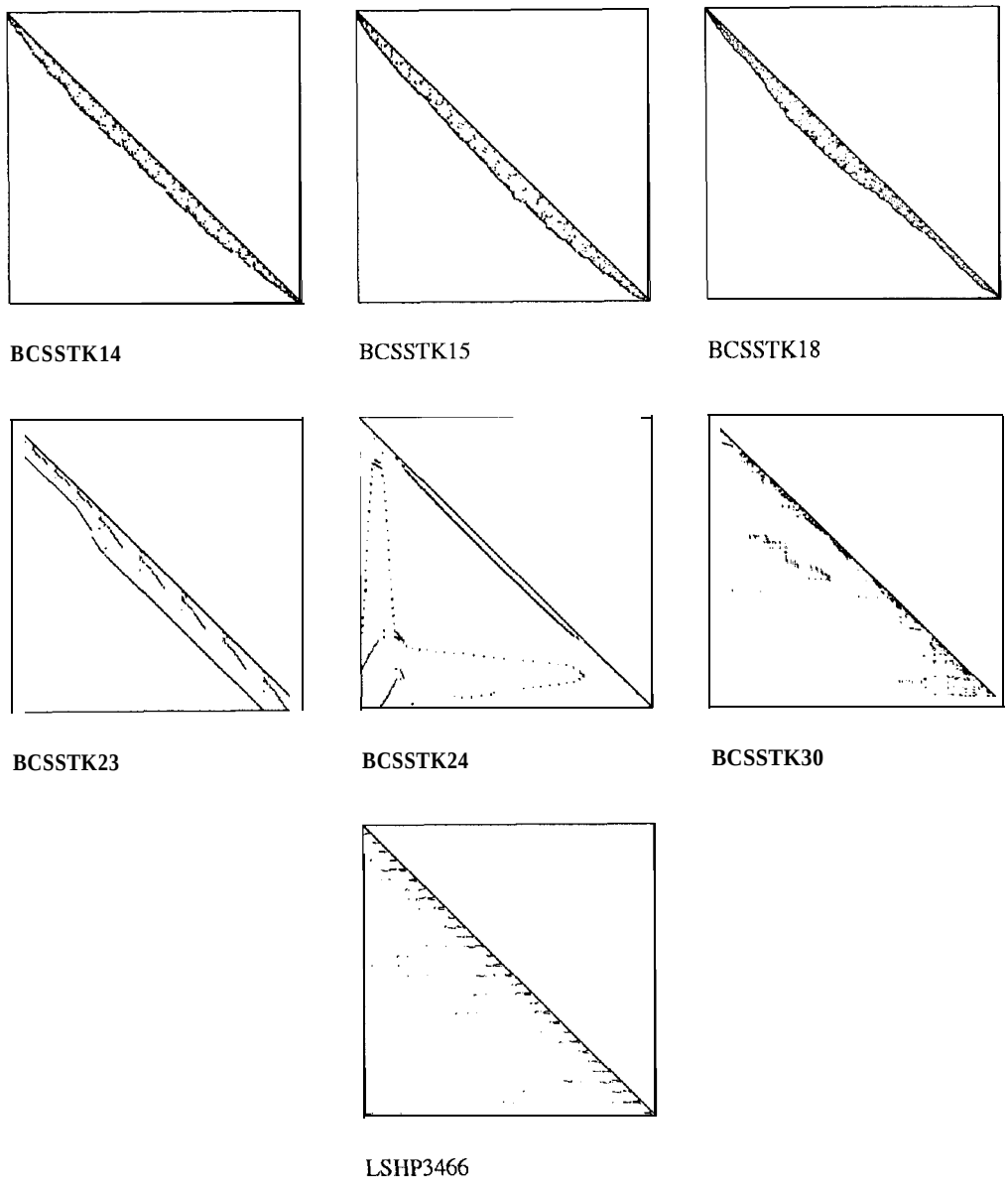

Figura 5.4: Patrones de las matrices prueba

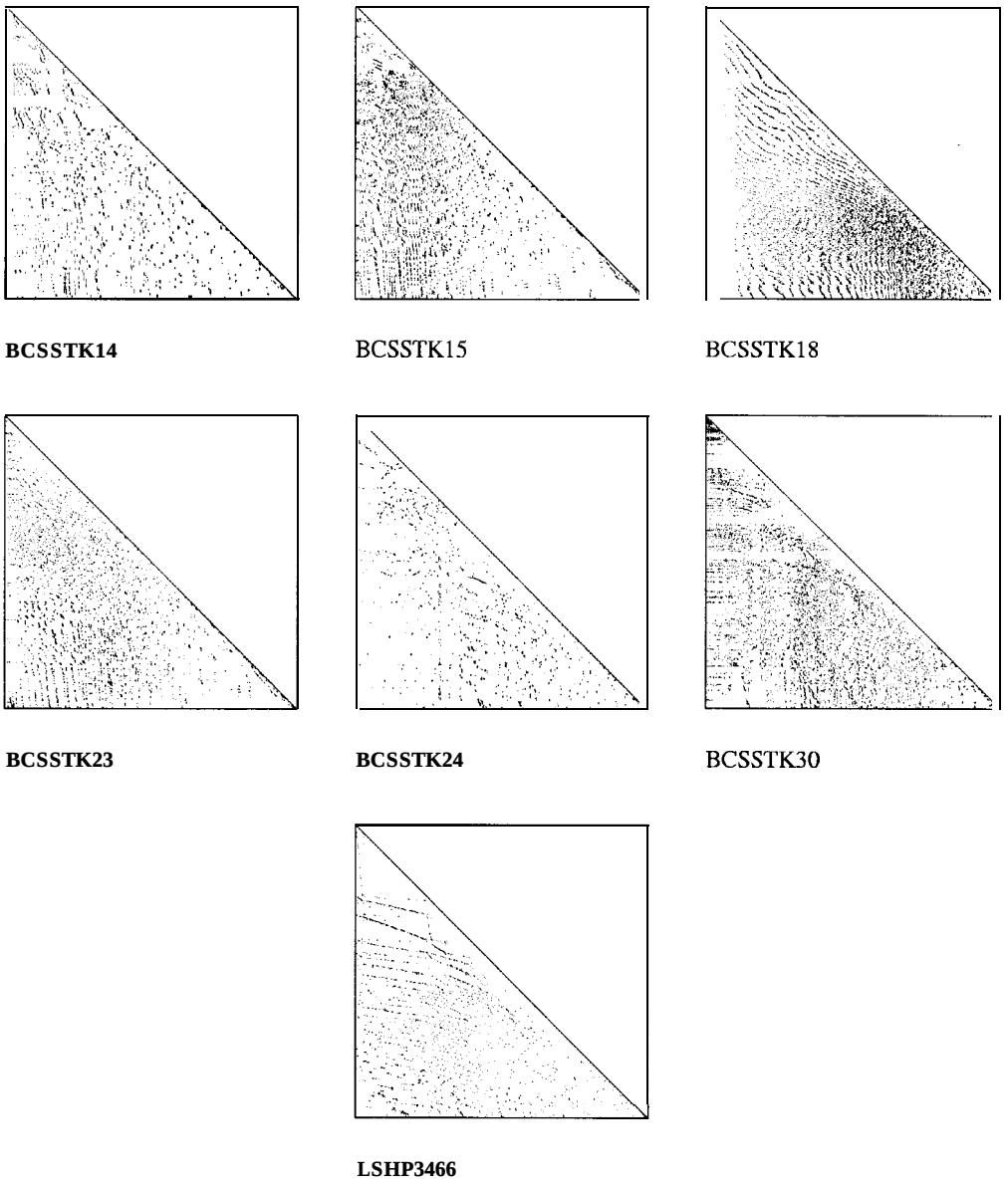

Figura 5.5: ***Patrones de las matrices prueba ordenadas***

MATRIZ	N	α en $\boldsymbol{A}$	α en $\boldsymbol{L}$
BCSSTK14	1806	36630	116071
BCSSTK15	3948	60882	707887
BCSSTK18	11948	80519	668217
BCSSTK23	3134	24156	450953
BCSSTK24	3562	81736	291151
BCSSTK30	28924	1036208	4677146
LSHP3466	3466	13681	93713

Tabla 5.1: ***Matrices prueba***

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	330	490	410	390
BCSSTK15	6000	9600	7300	6300
BCSSTK18	4500	7500	5600	4800
BCSSTK23	3800	6800	5200	4400
BCSSTK24	1100	1700	1300	1200
BCSSTK30	54500	65100	48000	40600
LSHP3466	200	300	250	230

Tabla 5.2: ***Tiempos de ejecución para el programa*** RL

5.4.1.1 **Tiempos de ejecución**

La Tabla 5.2 muestra los tiempos de ejecución obtenidos para el algoritmo ***right-Zooking*** paralelo *(RL)* en ms, siendo $\boldsymbol{P}$ el número de procesadores.

Cabe destacar que el tiempo de ejecución se incrementa considerablemente en el paso de ejecución secuencial a paralela debido a las sobrecargas ocasionadas por las operaciones de mantenimiento de la coherencia.

5.4.1.2 Balanceo de la carga

El lazo externo del programa está organizado como un lazo ***self-scheduling*** [***105]*** interactuando con una cola global de tareas listas para ser ejecutadas. Como ya se ha comentado, cuando un procesador queda libre accede a esta cola global y coge una tarea de la cola, la ejecuta y coloca nuevas tareas en la cola cuando están listas para ser ejecutadas. Esto conduce a un gran balanceo de la carga ya que la asignación de tareas a procesadores se realiza de una forma totalmente dinámica. En la Figura 5.6 se

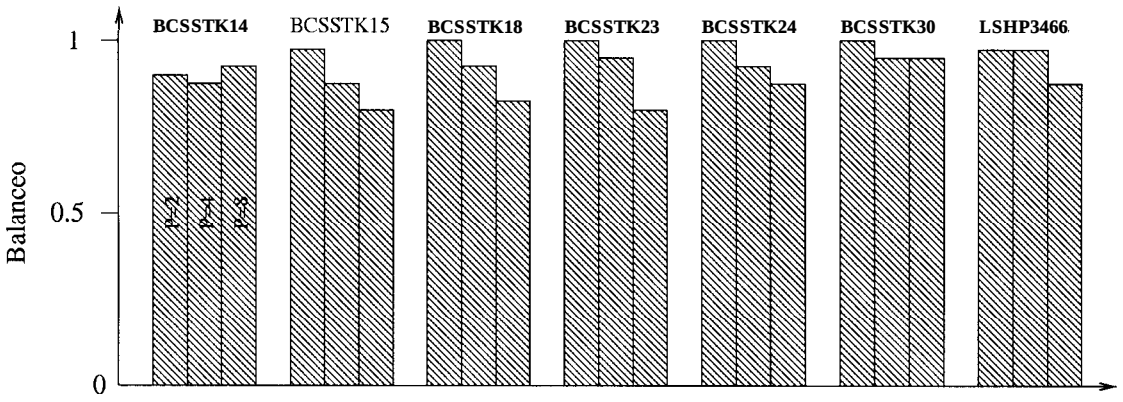

Figura 5.6: Balanceo de la carga en el programa *RL*

muestra el balanceo de la carga obtenido para cada una de las matrices sobre 2, 4 y 8 procesadores. Se utiliza como medida del balanceo:

$$B = I_{total}/(P * I_{max})$$

donde I_{total} es el número de instrucciones en punto flotante totales, P es el número de procesadores y I_{max} es el valor máximo del número de instrucciones en punto flotante asignadas a un procesador. Utilizamos los contadores de eventos del R10000 para contar el número de instrucciones en punto flotante que ejecuta cada procesador.

5.4.1.3 Comportamiento del sistema de memoria

Las operaciones de mantenimiento de la coherencia cache y el movimiento de datos que estas operaciones originan es el motivo de que exista un incremento en el tiempo de ejecución al pasar de uno a dos procesadores. Este overhead se arrastra para un número mayor de procesadores.

La Tabla 5.3 muestra el comportamiento del sistema de memoria para cada una de las matrices y para 1 y 2 procesadores. Se observa que para más de un procesador disminuye considerablemente el reuso de las líneas cache de la cache secundaria quedando reducido en un factor 3 en el mejor de los casos (matriz BCSSTK30) y en 1000 en el peor (matriz LSHP3466). Esto conduce a un elevado movimiento de datos entre la cache secundaria y memoria que influye notablemente en el tiempo de ejecución del programa, llegando incluso a lentificar el programa paralelo con respecto al secuencial.

El programa paralelo se compone de las siguientes funciones:

- CONSIGUETAREA() (A): Consigue un índice columna, llamémosle j, de la cola

P	BCSSTK14	BCSSTK15	BCSSTK18
REUS0 CACHE PRIMARIA			
1	14.58	12.29	11.69
2	14.24	12.22	8.32
REUSO CACHE SECUNDARIA			
1	1617.01	77.92	85.63
2	4.23	3.46	3.47
MOVIMIENTO DE DATOS (MB)			
1	0.04	86.38	59.48
2	78.04	1723.34	1326.58
MFLOPS			
1	17.97	18.28	16.70
2	11.68	11.42	9.92

P	BCSSTK23	BCSSTK24	BCSSTK30	LSHP3466
REUSO CACHE PRIMARIA				
1	11.23	13.92	12.66	16.22
2	11.19	13.79	12.66	26.24
REUSO CACHE SECUNDARIA				
1	270.95	386.89	8.16	7786.46
2	3.90	3.65	3.04	7.38
MOVIMIENTO DE DATOS (MB)				
1	13.71	2.39	5808.63	0.03
2	1134.80	295.58	13509.04	36.00
MFLOPS				
1	18.24	18.48	14.07	15.64
2	10.22	11.50	11.90	9.50

Tabla 5.3: ***Comportamiento del sistema de memoria en el programa RL***

de disponibles.

- NORMALIZA( ) (B): Calcula el elemento diagonal de la columna j y normaliza dicha columna.
- MODIFICA() (C): Modifica las columnas $s(s > j)$ que dependen de la columna j.
- INSERTACOLA( ) (D): Añade un índice columna a la cola de disponibles cuando se han realizado todas sus modificaciones.

Empleamos la utilidad SpeedShop para descubrir cual es la función que provoca este aumento en los tiempos de ejecución. Debe tenerse en cuenta, en este caso, que el uso de los contadores de eventos del R10000 introduce una sobrecarga en el programa debido al control software que aumenta el tiempo de ejecución en no más del 5 %. En la Figura 5.7 se muestra el tiempo que consume cada una de las funciones para cada una de las matrices en su ejecución secuencial, indicando además el porcentaje del tiempo total que supone. La suma de dichos porcentajes no alcanzan el 100 % debido a que un porcentaje del tiempo de ejecución lo consume funciones del sistema operativo.

Se observa que dentro de la factorización numérica la función más costosa es la de modificación de columnas posteriores, MODIFICA(). En la gráfica 5.8 se muestra el comportamiento paralelo de dicha función, tomando como tiempo paralelo el tiempo del procesador más lento con respecto a dicha función. Cabe destacar que para dos procesadores el tiempo de ejecución es muy superior a su ejecución secuencial. Hay que recordar que en esta versión del código una misma columna de la matriz es modificada por diferentes procesadores, circunstancia que ocurre muy habitualmente. Si un procesador modifica una línea cache, y luego otro procesador solicita la misma línea, se debe modificar la variable almacenada en memoria, por tanto aumenta el número de accesos a memoria. Además la escritura sobre una línea cache invalida todas las posibles copias que haya de esta línea sobre otros procesadores, con lo cual aumenta el número de operaciones de coherencia y el número de fallos cache.

En conclusión, esta versión del algoritmo paralelo es ineficiente debido a accesos a posiciones de memoria remotas en vez de locales con el consiguiente aumento de la latencia en el acceso a memoria. Tenemos, por tanto, un problema de localidad.

5.5 Mejoras al algoritmo right-looking paralelo

En la propuesta anterior se genera un elevado movimiento de datos entre procesadores que impone una gran penalización al rendimiento de la implementación paralela. Para

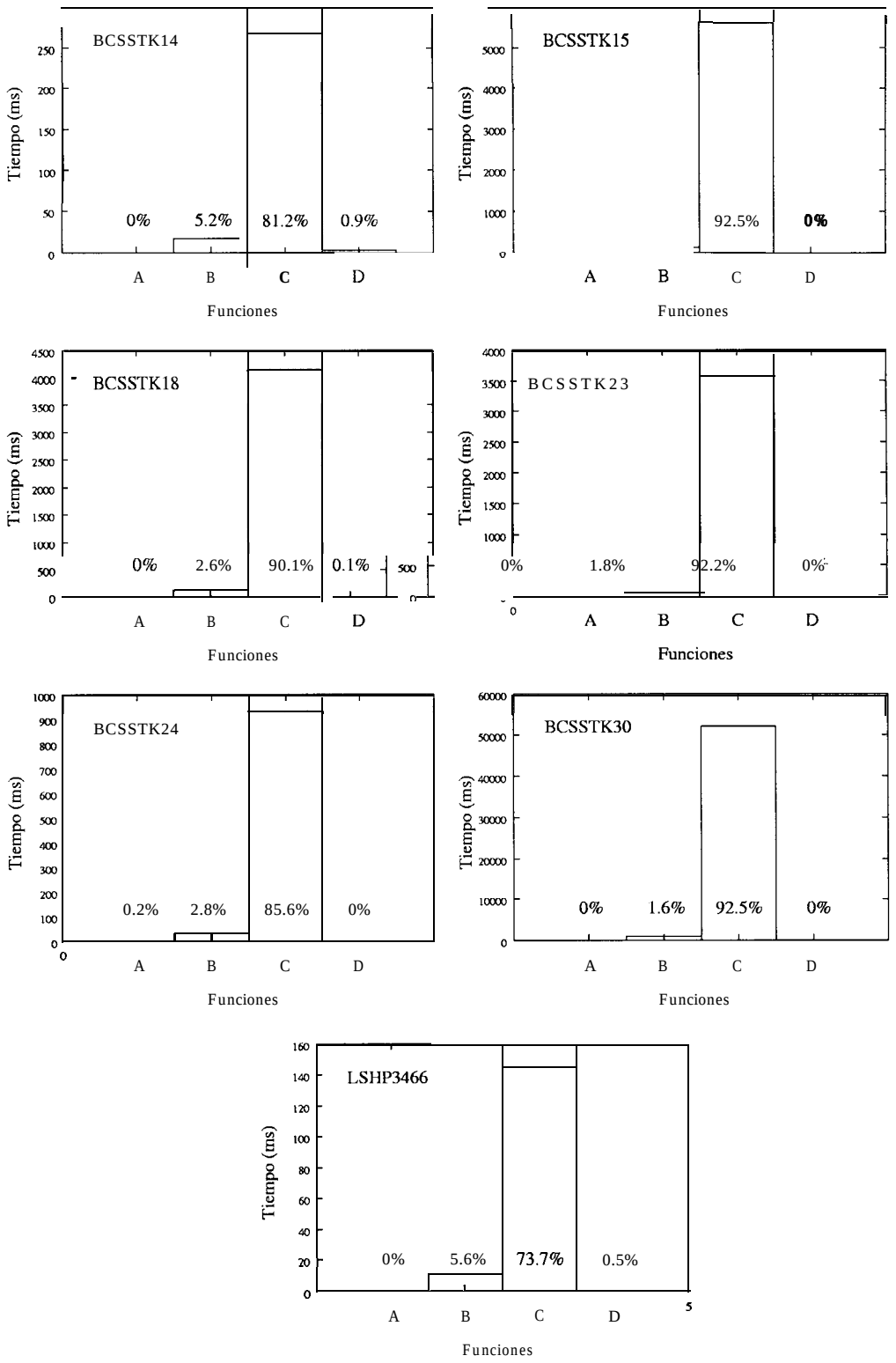

Figura 5.7: ***Tiempo de ejecución de las diferentes funciones del programa en el código RL secuencial***

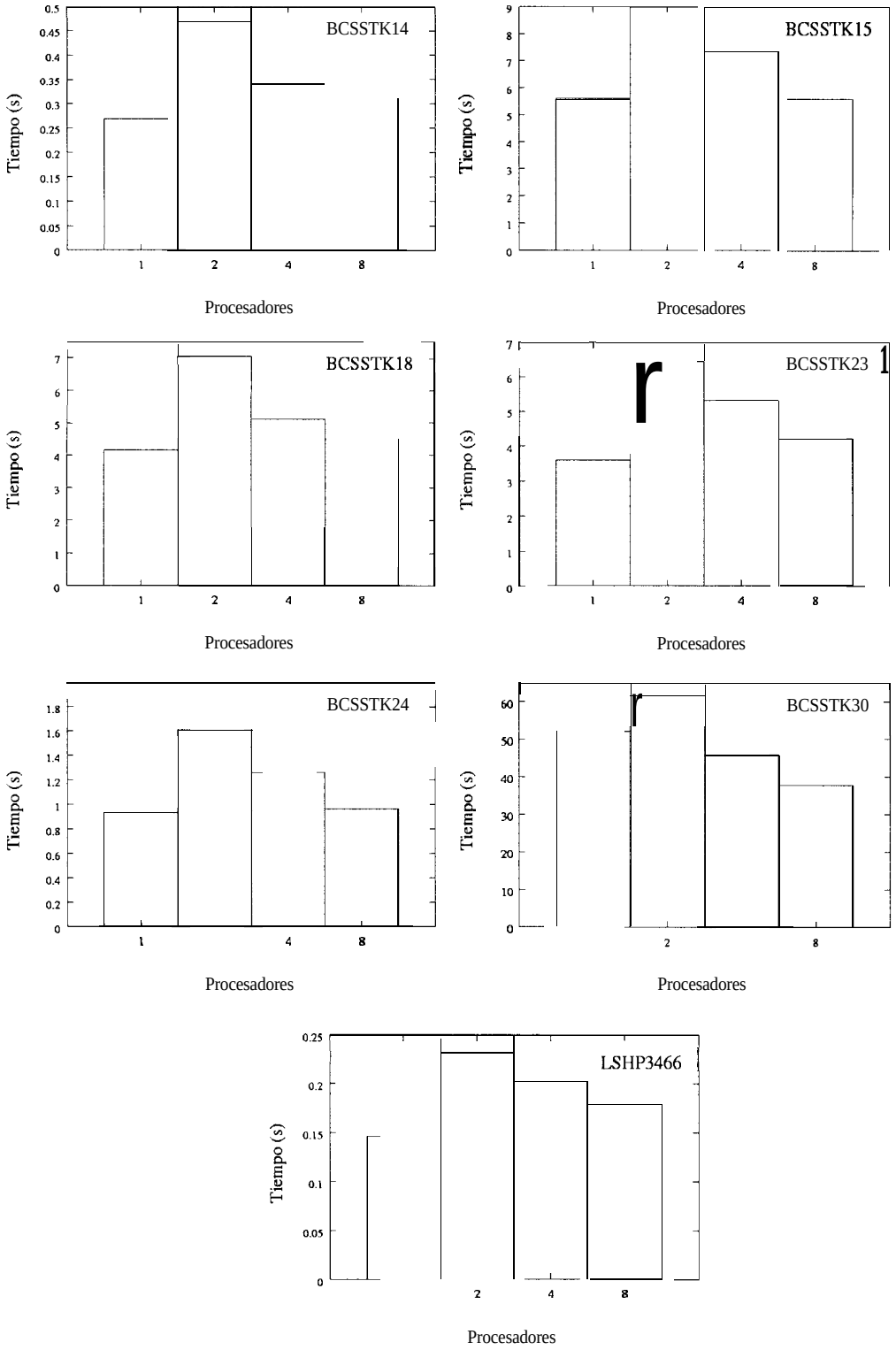

Figura 5.8: Comportamiento paralelo *de la función* **MODIFICA() para el** *programa* RL

reducir este movimiento trataremos de aumentar la localidad de los datos. Seguiremos para ello dos estrategias:

1. Proponemos un nuevo scheduling basado en el árbol de eliminación de la matriz.
2. Creamos un conjunto de arrays locales en los que vamos almacenando las modificaciones que se realizan sobre cada columna.

5.5.1 Nuevo scheduling para el algoritmo right-looking

Como hemos visto en 2.4.3, el árbol de eliminación [68] captura información sobre las dependencias entre columnas de la matriz. Una columna depende sólo de las columnas que se encuentran en su mismo subárbol del árbol de eliminación y por debajo de ella; equivalentemente, una columna sólo modifica las columnas que se encuentran en el camino entre ella y el nodo raíz. A su vez, las columnas que no tienen ninguna relación antecesor/descendiente, es decir, pertenecen a diferentes subárboles, no dependen una de la otra y, por tanto, no intervienen en el proceso de modificación de la otra columna. Consecuentemente, en cuanto a la localidad de los datos se refiere, sería ventajoso que cada procesador trabajase en la medida de lo posible dentro del mismo subárbol.

En la estrategia descrita en la sección anterior, la cola inicial de tareas está formada por los nodos hoja del árbol de eliminación almacenados en orden ascendiente de columnas, y los procesadores acceden a esta cola sin ningún criterio establecido. Nosotros proponemos una modificación al scheduling del algoritmo teniendo en cuenta el árbol de eliminación, y con el objetivo de que diferentes procesadores accedan a nodos pertenecientes a diferentes subárboles. Para ello se realiza una ordenación de la cola de disponibles usando una función distancia entre nodos del árbol de eliminación y seleccionando los nodos que se encuentren a mayor distancia entre si, aplicando para realizar dicha ordenación el método de Prim [88,44]. Sea P el número de procesadores, esta ordenación inicial seguirá los siguientes pasos:

1. Si el número de nodos hoja es menor o igual que P ir a 5.
2. Se elige como primer nodo el nodo que se encuentra a mayor profundidad en el árbol, es decir, el más alejado de la raíz.
3. Se ordenan los nodos hoja utilizando el algoritmo de Prim para maximizar distancias hasta obtener P nodos.
4. Se asigna cada uno de los P nodos anteriores a un procesador como tarea inicial.
5. Se empieza la factorización numérica.

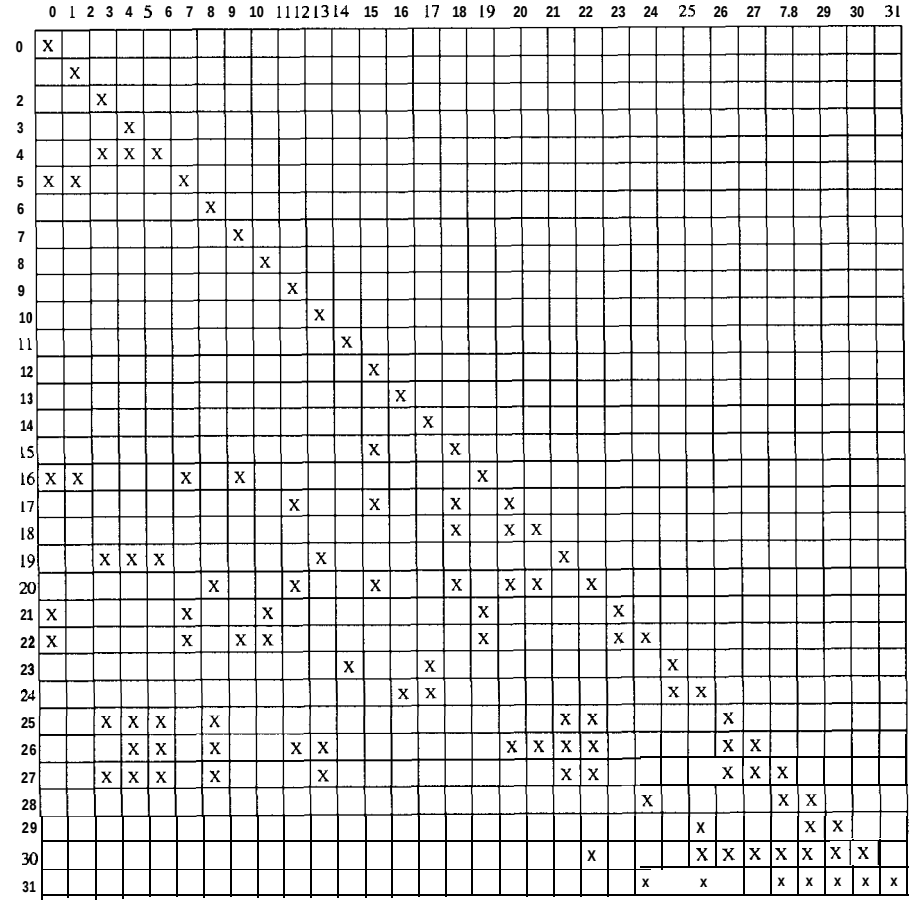

Figura 5.9: *Matriz ejemplo: estructura del factor L*

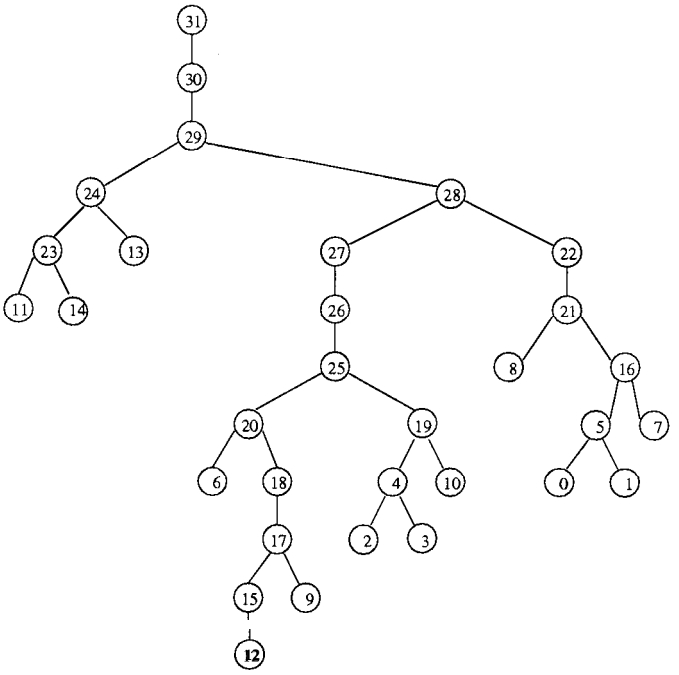

Figura 5.10: ***Árbol de eliminación para la matriz ejemplo de la Figura 5.9***

La función distancia debe representar una medida de la proximidad de dos nodos dentro del árbol de eliminación. Para ello se considera tanto el posible trabajo paralelo, como el número de niveles entre nodos y la altura del primer antecesor común. Sean i y j dos nodos con primer descendiente común k. Se define la distancia entre ellos como:

$$d_{ij} = d_{ji} = \alpha min\{P_{ik}, P_{jk}\} + \beta(P_{ik} + P_{jk}) + \gamma L_k \tag{5.1}$$

donde P_{ik} ***es*** el número de niveles entre el nodo i y el nodo ***k,*** L_k ***es*** la altura a la que se encuentra el nodo ***k, es*** decir, el número de niveles desde el nodo.más profundo hasta el nodo ***k.***

Por tanto, el primer término de esta suma refleja el trabajo en paralelo que se puede llevar a cabo al procesar los nodos i y j. Este término es lo que más va a influir en la cantidad de trabajo en paralelo posible. El segundo término tiene en cuenta el número de niveles que hay entre ambos nodos, cuanto mayor sea este número mayor será el trabajo en paralelo y menor el conflicto entre ambos. El tercer término refleja la altura del primer antecesor común. Cuanto mayor sea, menos nodos tendrá que modificar y, por tanto, menores serán los conflictos. Los parámetros α, β y γ ponderan la influencia relativa de los tres factores y han sido elegidos para nuestros experimentos con los valores α = N, β = 1 y γ = 1. Tomando como ejemplo el árbol de eliminación de la

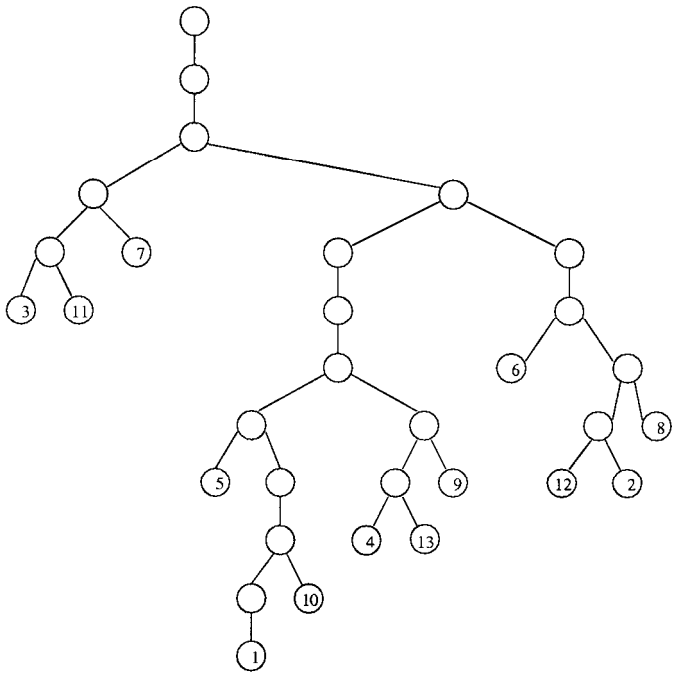

Figura 5.11: ***Ordenación de los nodos hoja del árbol de la Figura 5.10 utilizando Prim***

Figura 5.10, la distancia entre el nodo 0 y el 13 será:

$$\begin{aligned} d_{0,13} &= \alpha min\{P_{0,29}, P_{13,29}\} + \beta(P_{0,29} + P_{13,29}) + \gamma L_{29} \\ &= \alpha min\{6,2\} + \beta(6+2) + \gamma 9 = 2\alpha + 8\beta + 9\gamma \end{aligned}$$

En la Figura 5.11 se muestra como se ordenarían los nodos hoja del árbol de eliminación de la matriz correspondiente al ejemplo de la Figura 5.9 utilizando el algoritmo de Prim para maximizar distancias. Si consideramos dos procesadores, el método de Prim asigna a un procesador el nodo más profundo, es decir, el nodo 12, y al otro el nodo más alejado de 12 según la definición de la distancia introducida, es decir, el nodo 1. La cola inicial de tareas para dos procesadores quedaría, por tanto, ordenada de la siguiente forma:

$$\{\mathbf{12}, \mathbf{1}, 0, 2, 3, 6, 7, 8, 9, 10, \text{ll}, 13, 14\}$$

Una vez realizada la asignación inicial, cada procesador debe restringir su acceso, en la medida de lo posible, a columnas pertenecientes al mismo subárbol. Por ello, si un procesador es el último en modificar una columna, esa columna pasa a ser procesada por el mismo procesador y no se incluye en la cola. En términos del árbol de eliminación de la matriz, si un procesador es el último en modificar una columna, esta columna ha de ser el padre de la última columna procesada. Un pseudo-código del algotitmo resultante se muestra en la Figura 5.12, donde la variable ***nextcol*** mantiene la siguiente columna

```
1 . ordenar la cola de tareas utilizando Prim
2 . while cola no vacía do
3 .     conseguir una columna j de la cola
4 .     nextcol = -1
5 .     calcular l_jj
6 .     normalizar la columna j
7 .     modificar su nodo padre, llamémosle k
8 .     nmod[k] = nmod[k] - 1
9 .     if nmod[k] = 0 then
10.         nextcol = k
11.     endif
12.     for each l_sj ≠ 0 (s > k) do
13.       modificar la columna s con la columna j
14.       nmod[s] = nmod[s] - 1
15.     endfor
16.     if nextcol > 0
17.       j = nextcol
18.       go to 4
19.     endif
20. endwhile
```

Figura 5.12: *Algoritmo paralelo con el nuevo scheduling (RNS)*

a procesar. En nuestro algoritmo, si $nextcol$ = -1 se accede a la cola de tareas para obtener la nueva columna. Hemos denominado a este algoritmo como RNS.

Suponiendo la matriz ejemplo de la Figura 5.9, y en el caso ideal de que todas las columnas tardan el mismo tiempo en ejecutarse, para dos procesadores el árbol de eliminación de la matriz quedaría distribuido según se muestra en la Figura 5.14. En esta figura los números indican el orden en el que los nodos son procesados dentro de cada procesador. En términos de la localidad, no importa sólo que nodos procesa cada uno, si no también en que orden son procesados. Como vemos, con el esquema totalmente dinámico (Figura 5.13) no se obtiene un patrón de localidad en el árbol. Sin embargo, con nuestra propuesta de *scheduling* (Figura 5.14) las computaciones se mantienen dentro de una misma zona del árbol en mayor medida. En contrapartida, este nuevo *scheduling* introduce un desbalanceo de la carga que puede llegar a limitar el rendimiento del programa.

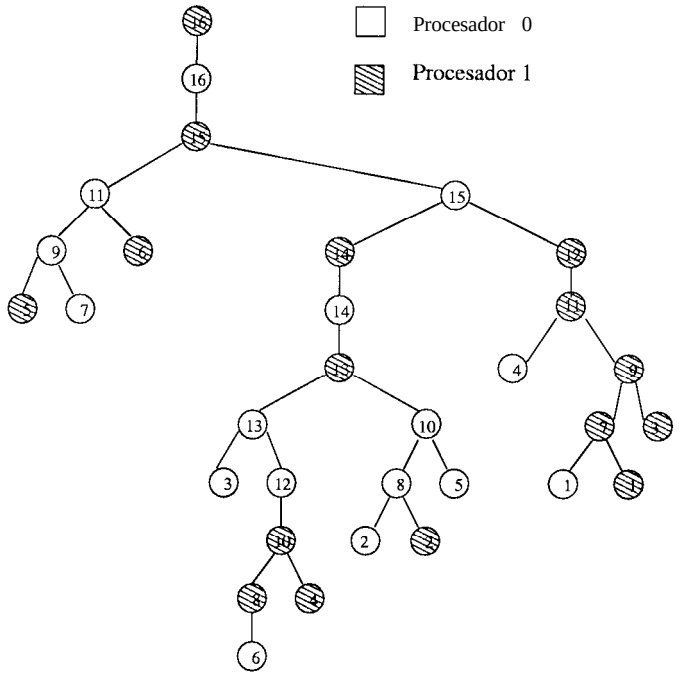

Figura 5.13: ***Distribución de las tareas entre los procesadores: scheduling dinámico***

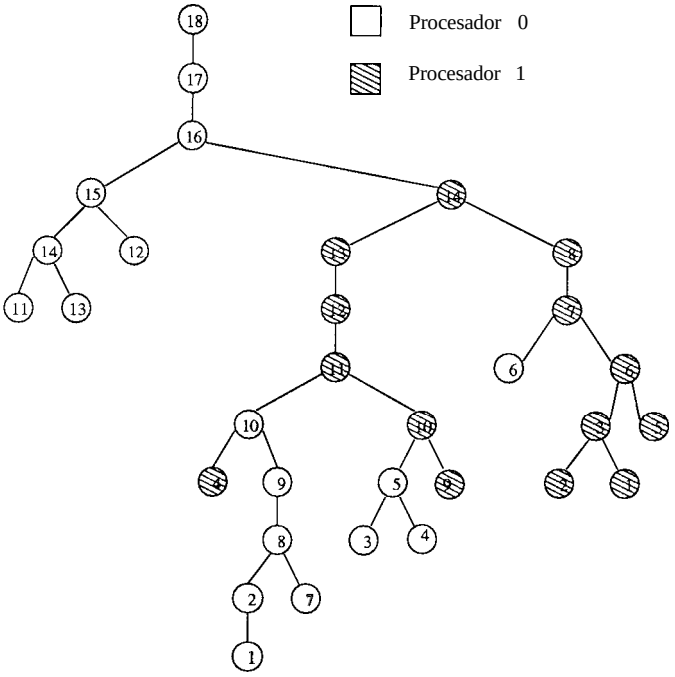

Figura 5.14: ***Distribución de las tareas entre los procesadores: nuevo scheduling***

Se obtienen varias ventajas con el nuevo *scheduling* propuesto. Por un lado se aumenta el reuso. Al trabajar con el nodo padre de la última columna procesada se está utilizando la misma columna más veces por el mismo procesador, pero principalmente se está utilizando una columna que modificará columnas pertenecientes al mismo subárbol que la última columna procesada, aumentando con ello las posibilidades de que el mismo procesador modifique las mismas columnas, con lo que disminuye el número de invalidaciones y, en consecuencia, el número de fallos cache. Por otro lado, con este planteamiento también se consigue un menor número de accesos a la cola global, con el consiguiente descenso en el *overhead* asociado a la sincronización.

Se podría hacer un refinamiento de esta versión del código consistente en un acceso selectivo a la cola de disponibles de manera que cuando un procesador tiene que acceder a la cola de disponibles, en primer lugar comprueba si alguno de los hermanos de la última columna procesada está listo para ser computado. Si es así, es asumido para su procesamiento; si no, se accede a cualquier columna disponible. Con este planteamiento se aumenta la localidad, pero en contrapartida, habrá un *overhead* asociado a la rutina CONSIGUETAREA(). Hemos denominado a esta versión del código algoritmo ***RBH.***

Otra alternativa consistiría en acceder al nodo que se encuentre a menor distancia del último nodo procesado utilizando la función distancia anteriormente descrita. De nuevo el objetivo es aumentar la localidad, se trata con ello de acceder a nodos pertenecientes al mismo subárbol. Para llevar a cabo este planteamiento se tendrá que convertir la cola de tareas disponibles en una lista enlazada, lo cual implica un incremento en el tiempo de ejecución debido a su gestión. Hemos denominado a esta versión del código algoritmo ***RBD.***

5.5.1.1 Resultados experimentales

Ejecutamos nuestro código sobre el sistema 02000 de Silicon Graphics utilizando las mismas matrices prueba de la sección anterior. Se realiza de nuevo un estudio en cuanto a tiempos de ejecución, balanceo de la carga y comportamiento del sistema de memoria.

Tiempos de ejecución En la Tabla 5.4 se muestran los tiempos de ejecución del programa con el nuevo *scheduling* y diferente número de procesadores.

La observación más inmediata es el drástico descenso en los tiempos de ejecución del programa para más de un procesador.

En la Tabla 5.5 se muestran los tiempos obtenidos para la versión del código correspondiente al acceso selectivo a la cola de disponibles en busca de un hermano del último nodo procesado. Como se puede apreciar se consigue disminuir el tiempo con respecto a la versión anterior en la mayoría de los casos.

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	300	210	200	210
BCSSTK15	5600	4400	4000	4000
BCSSTK18	4200	3000	2900	3000
BCSSTK23	3700	3300	3400	3600
BCSSTK24	1000	780	670	610
BCSSTK30	41100	26000	24800	22700
LSHP3466	170	120	110	110

Tabla 5.4: ***Tiempos de ejecución para el algoritmo RNS***

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	300	220	180	190
BCSSTK15	5600	4400	3900	3900
BCSSTK18	4200	3100	2900	3000
BCSSTK23	3700	3400	3400	3500
BCSSTK24	1000	730	570	580
BCSSTK30	41100	31600	26500	23600
LSHP3466	170	120	100	110

Tabla 5.5: ***Tiempos de ejecución para el algoritmo RBH***

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	310	200	180	190
BCSSTK15	5700	4300	3900	4000
BCSSTK18	5700	4400	4100	4100
BCSSTK23	3800	3300	3400	3400
BCSSTK24	1000	730	570	550
BCSSTK30	41600	30700	25200	22600
LSHP3466	220	150	130	140

Tabla 5.6: Tiempos de ejecución para el algoritmo RBD

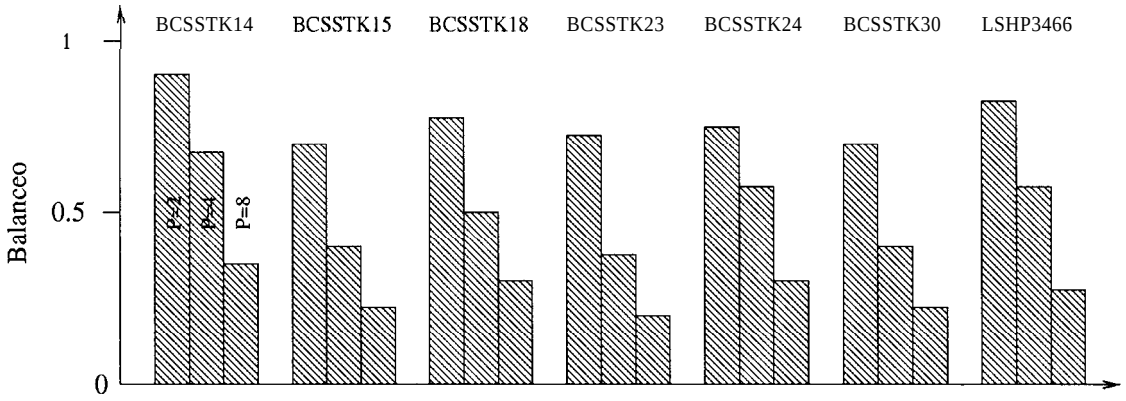

Figura 5.15: Balanceo de la carga en el programa RNS

En la Tabla 5.6 se muestran los resultados obtenidos al aplicar el acceso selectivo a la cola de disponibles utilizando la función distancia. Vemos que aunque se consiguen reducir los tiempos de ejecución en algunos casos, los resultados van a ser dependientes de la matriz, así, para la matriz BCSSTK18 no compensa la utilización de esta estrategia debido al *overhead* que introduce su implementación.

Balanceo de la carga El principal problema de este planteamiento va a ser el balanceo de la carga, lo cual limita la escalabilidad del programa paralelo. En contrapartida, el comportamiento de la memoria mejora notablemente como se verá en la siguiente sección.

En la Figura 5.15 se muestran los resultados obtenidos en cuanto a balanceo de la carga. Se observa que el problema del balanceo se agrava a medida que aumenta el número de procesadores.

BCSSTK14	BCSSTK15	BCSSTK18
REUS0 CACHE PRIMARIA		
23.46	20.48	16.62
REUS0 CACHE SECUNDARIA		
31.61	73.04	39.47
MOVIMIENTO DE DATOS (MB)		
9.46	104.54	138.29
MFLOPS		
26.18	25.14	24.40

BCSSTK23	BCSSTK24	BCSSTK30	LSHP3466
REUS0 CACHE PRIMARIA			
18.04	21.67	20.50	33.02
REUS0 CACHE SECUNDARIA			
25.64	32.54	129.34	15.06
MOVIMIENTO DE DATOS (MB)			
200.44	36.5	411.28	7.26
MFLOPS			
18.80	25.96	26.50	22.82

Tabla 5.7: *Comportamiento del sistema de memoria para el algoritmo RNS*

Comportamiento del sistema de memoria Hemos medido los mismos parámetros que para el código anterior, restringiéndonos ahora sólo a dos procesadores. En la Tabla 5.7 se muestran los resultados obtenidos. Se ha conseguido aumentar el reuso en la cache primaria y secundaria, especialmente el de la cache secundaria, el cual ha multiplicado por dos su valor en el peor de los casos (matriz LSHP3466) y por 40 en el mejor (matriz BCSSTK30). Como consecuencia se produce un gran descenso en el movimiento de datos entre esta cache y memoria, lo cual conduce a un aumento en el rendimiento del programa que se refleja en el aumento en el número de MFLOPS.

En la Figura 5.16 se muestra el nuevo comportamiento paralelo de la función MODIFICA(). En esta nueva versión del programa se consigue aceleración.

Resumiendo, nuestra propuesta aumenta la localidad y con ello la eficiencia del código paralelo a costa de crear un desbalanceo importante en la carga que limita su escalabilidad.

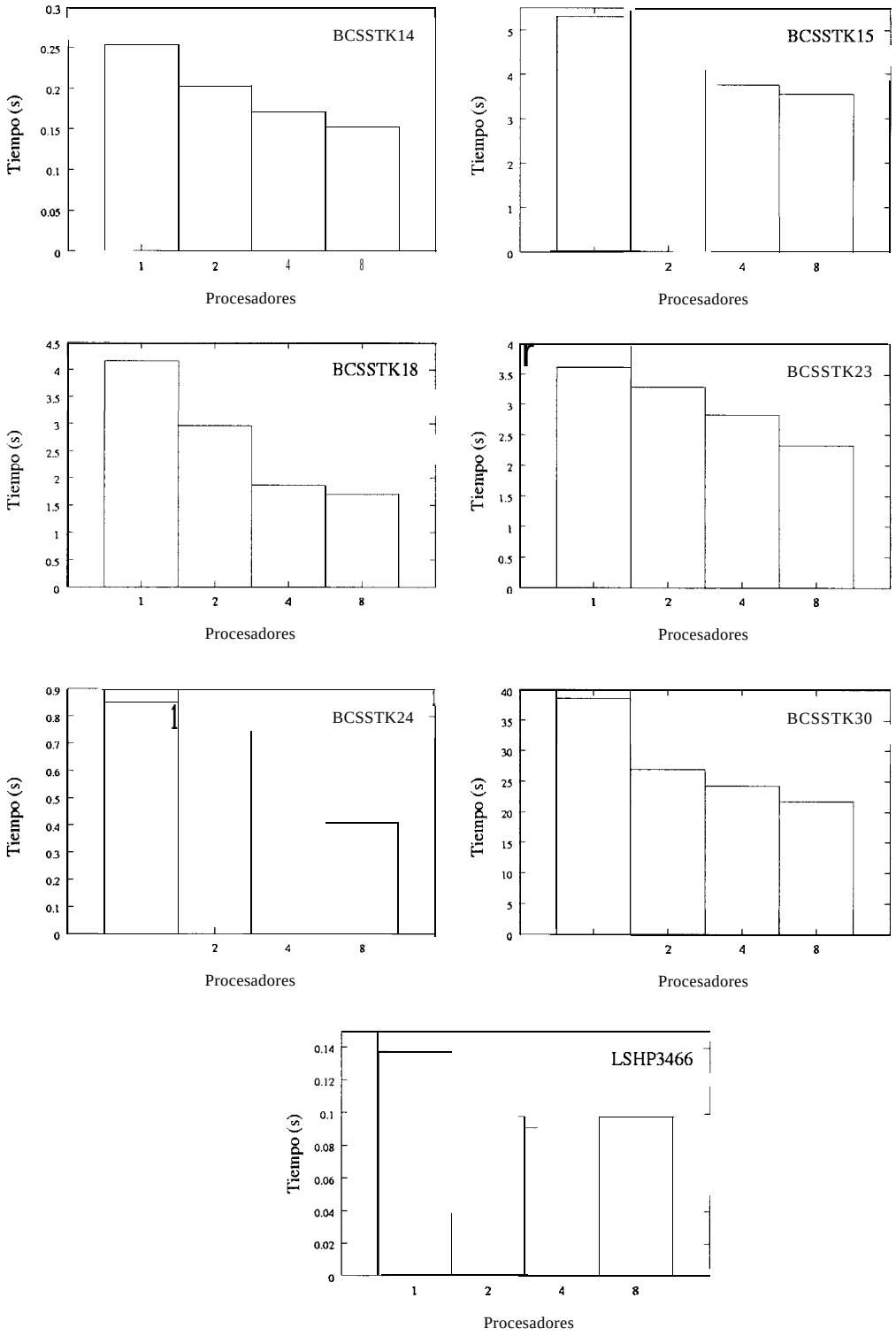

Figura 5.16: ***Comportamiento*** paralelo de la ***función*** MODIFICA() para el ***programa RNS***

5.5.2 Utilización de arrays locales en el algoritmo right-looking

El principal problema para obtener aceleración en el programa viene del elevado número de invalidaciones que se producen al modificar una columna. Para minimizar este efecto proponemos replicar el *array* que mantiene los valores de las entradas de la matriz (DA) para cada procesador, de forma que cada procesador pueda ir acumulando sus modificaciones locales a una columna en un *array* local. Es decir, se pasa de un *array* bidimensional $DA[i][j]$, a un *array* tridimensional de la forma $DA[k][i][j]$, con $0 \leq$ i, $j <$ N y $0 \leq k \leq P$. $DA[k]$ *[i]* $[j], 0 \leq$ i < N, almacena las modificaciones que realiza el procesador k sobre la columna j. $DA[P][i][j], 0 \leq$ i < N, almacena los valores iniciales de la columna j. Cada columna dentro de cada procesador se almacena, de nuevo, en líneas cache diferentes para evitar conflictos entre procesadores. El número de modificaciones de una columna j se contabiliza localmente en un *array* $nmod[j]$ local que se almacena junto con los valores de la columna j de la matriz. Cuando la columna j que se está modificando es el padre de la última columna procesada se comprueba si ya se han hecho todas las modificaciones. Para ello, se suman todas las contribuciones desde todos los procesadores (todos los $nmod[j]$ locales) dentro de una variable local, y se comprueba si este número coincide con el número total de modificaciones que debe recibir esa columna. Si es así, se suman todas las contribuciones desde todos los procesadores a dicha columna y pasa a ser la siguiente columna en ser procesada. Es decir, se suman todos los $DA[k][i][j], 0 \leq k <$ *P, 0* $\leq i <$ *N,* sobre $DA[P][i][j], 0 \leq$ i < N.

Hemos denominado a la versión del código que utiliza el nuevo *scheduling* propuesto y *arrays* locales algoritmo *RNS2.*

5.5.2.1 **Resultados experimentales**

Hemos aplicado el uso de *arrays* locales para mantener las modificaciones realizadas sobre cada columna. En la Tabla 5.8 se muestran los resultados obtenidos en cuanto a tiempos de ejecución. Vemos que en general los tiempos son mejores, teniendo como única excepción la matriz BCSSTK30. Con esta estrategia se eliminan intervenciones e invalidaciones de líneas cache, pero en contrapartida se utiliza una mayor cantidad de memoria que se traduce en un *overheud* asociado a su gestión. Este *overheud es* más importante cuanto mayor sea la matriz y de ahí que para matrices grandes no compense su utilización.

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	290	180	160	150
BCSSTK15	5700	4200	3800	3500
BCSSTK18	4200	2700	2000	2100
BCSSTK23	3800	2600	2600	2600
BCSSTK24	1000	660	460	460
BCSSTK30	42100	31100	26600	26700
LSHP3466	170	100	70	100

Tabla 5.8: Tiempos de ejecución para el algoritmo *RNS2*

5.6 Algoritmo left-looking

El problema del algoritmo *right-looking* es que una misma columna es modificada por distintos procesadores produciéndose un elevado número de conflictos que degradan el comportamiento del programa. Sin embargo, se puede modificar el flujo de ejecución del programa para que una columna solamente sea modificada por un procesador. Este comportamiento es la base del algoritmo *left-looking*. En este caso, primero se ejecutan las operaciones $cmod(j,k)$, con $k < j$, y a continuación se normaliza la columna j.

Para llevar a cabo este planteamiento, la cola de tareas se genera al inicio del programa especificándose en que orden se van a ejecutar las columnas. Cuando un procesador queda libre visita la cola para adquirir un índice columna j y realizar la tarea $Tcol(j)$, compuesta de las subtareas cmod y $cdiv$. La implementación de este algoritmo sobre una máquina de memoria compartida para la factorización de Cholesky estándar es descrita en [34].

Cuando un procesador adquiere un índice columna j inicialmente intenta llevar a cabo todas las modificaciones $cmod(j,k)$, donde *k es* el conjunto de columnas anteriores de las que depende. Antes de poder ejecutar $cmod(j,k)$ la columna *k* debe haber sido normalizada, si no es así el procesador entra en un bucle de espera hasta que la columna *k* esté disponible. Un importante detalle de implementación es que las subtareas $cmod(j,k)$ pueden ser ejecutadas en cualquier orden, es decir, no es necesario ejecutar $cmod(j,k)$ en orden creciente de *k*. Los procesadores sólo llegarán a estar desocupados si todas las subtareas $cmod(j,k)$ están esperando por la ejecución de las respectivas subtareas *cdiv (k).*

Para llevar a cabo $cmod(j,k)$ con $k < j$ *se* desempaqueta la forma compacta de la columna j de *L,* de forma que las modificaciones por otras columnas puedan ser hechas eficientemente. Esto implica que el procesador ejecutando $Tcol(j)$ requiere un *array* de trabajo local de tamaño N para facilitar la modificación de la columna j. Son nece-

```
1. Construir la lista priorizada de tareas
2. while lista no vacía do
3.     conseguir una columna j de la lista
4.     while nmod[j] > 0 do
5.       esperar hasta que link[j] > 0
6.       k = link[j]
7.       link[j] = link[k]
8.       modificar la columna j con la columna k
9.       nmod[j]=nmod[j]-1
10.      nextnz = next(j, k)
11.      if nextnz ≤ N then
12.        link[k] = link[nextnz]
13.        link[nextnz] = k
14.      endif
15.    endwhile
16. endwhile
```

Figura 5.17: *Algoritmo left-looking paralelo*

sarias además otras dos estructuras de N elementos: El vector nmod y la lista enlazada $link; link[j]$ *es* una lista enlazada que almacena las columnas que actualmente están listas para modificar la columna j [40]. Esta lista enlazada se construye dinámicamente durante la ejecución del programa y debe ser globalmente accesible por todos los procesadores. Debe, por tanto, estar protegida para evitar que dos procesadores intenten modificarla al mismo tiempo. Un pseudo-código del algoritmo ejecutado sobre cada procesador se muestra en la Figura 5.17, donde $next(j, k)$ *es* el índice fila de la entrada no nula en la columna k inmediatamente debajo de L_{jk}.

En el algoritmo presentado en [34] las columnas son procesadas en el orden dado por el ordenamiento previo de la matriz. En general, dado que la matriz es dispersa, el tiempo de ejecución del programa paralelo depende del orden en el que se ejecutan las columnas. Por lo tanto, la pieza clave en todo este planteamiento es la construcción de la cola de tareas. Cuando un procesador está libre le es asignada la primera tarea de la cola con independencia de si está lista para ser ejecutada o no. La elección de esta lista priorizada de una forma óptima que minimice los tiempos de espera es un problema NP completo. Para resolverlo existen varias heurísticas. Una de las más utilizadas es el método del camino crítico [18]. Otro posible *scheduling es* el propuesto por Geist y Ng [32], el cual es una generalización del esquema *subtree-to-subcube* diseñado para

reducir las comunicaciones sobre un hipercubo [43].

El camino crítico supone que hay una tarea terminal única, es decir, que la matriz es irreducible. Esta tarea terminal es aquella que no tiene sucesores (la raíz). Para cada tarea se calcula el trabajo necesario para moverse desde esa tarea hasta la tarea terminal. La lista de tareas se construye en orden decreciente de sus trabajos, de esta forma las tareas más lejanas del nodo terminal tendrán una prioridad más alta [58]. Esta forma de construir la cola de tareas no es más que una generalización del esquema del camino crítico propuesto por Jee and Kees [57], en el cual se asume que las tareas requieren un tiempo de ejecución unidad.

Implementamos el algoritmo *left-looking* descrito en [34] sobre el 02000 adaptándolo para el algoritmo de Cholesky modificado. Básicamente el algoritmo sigue el mismo flujo, salvo que antes de realizar la normalización de una columna se debe calcular su diagonal de acuerdo con la ecuación: $d_j = max\{\delta, |a_{jj}|, \theta_j^2/\beta^2\}$.

56.1 Resultados experimentales

Se utiliza para evaluar nuestros códigos el conjunto de matrices de la *Harwell Boeing* de la Tabla 5.1. Probamos el programa con tres *schedulings* diferentes:

- orden de las columnas después de aplicar *minimum degree* (*LNO*).
- orden de las columnas siguiendo un orden posterior del árbol [2]. Este ordenamiento se ha demostrado eficiente para la ejecución secuencial del algoritmo sobre entornos paginados [65] (*LPO*).
- orden de las columnas aplicando el ordenamiento según el camino crítico (*LCC*).

Las Figuras 5.18 y 5.19 muestran como se ordenarían los nodos del árbol de eliminación correspondiente a la matriz ejemplo de la Figura 5.9 aplicando un orden posterior y el ordenamiento según el camino crítico respectivamente.

En las Tablas 5.9, 5.10 y 5.11 se muestran los tiempos de ejecución obtenidos para cada caso. El método del orden posterior recorre el árbol por subárboles lo cual puede ser adecuado para su procesamiento secuencial al aumentar la localidad de los datos. Esto se puede ver por ejemplo en la reducción del tiempo secuencial sólo apreciable para matrices grandes como la BCSSTK30. Sin embargo no es apropiado para la construcción de la lista prior-izada para el procesamiento paralelo.

El programa paralelo se compone de las siguientes funciones:

- CONSIGUETAREA() (A): Asigna un índice columna de la lista prior-izada de tareas, llamémosle j:

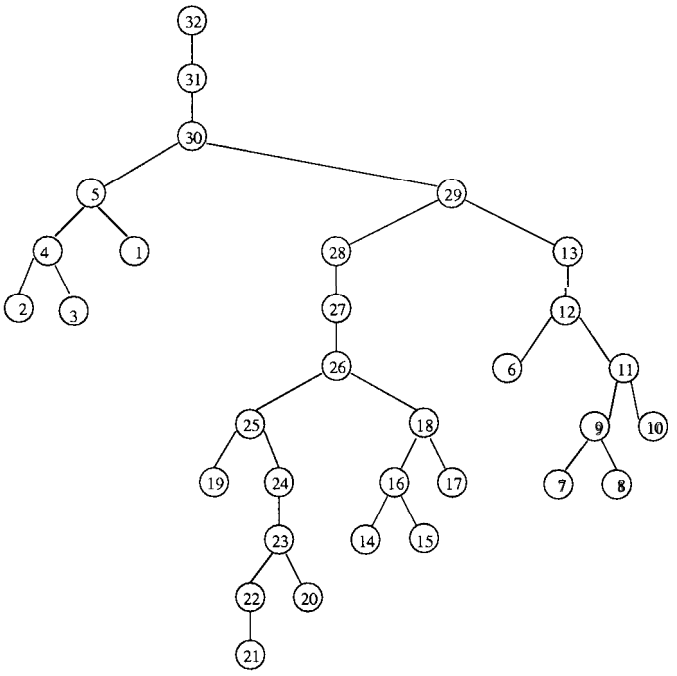

Figura 5.18: *Ordenación del árbol según un orden posterior*

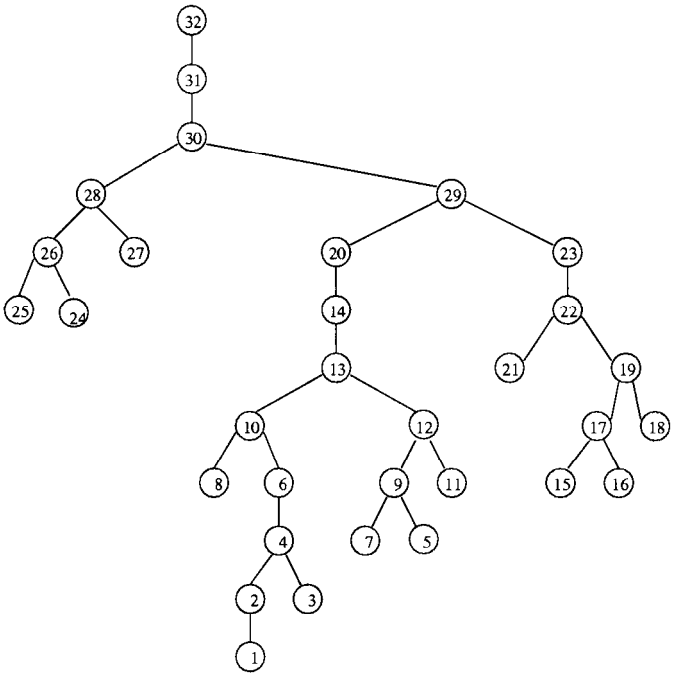

Figura 5.19: *Ordenación del árbol según el camino crítico*

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	360	810	1000	1800
BCSSTK15	6100	5000	6000	11700
BCSSTK18	4300	4100	5800	10400
BCSSTK23	3800	3200	3900	7400
BCSSTK24	1200	1700	2500	4700
BCSSTK30	42600	35600	43500	83900
LSHP3466	210	450	740	1300

Tabla 5.9: Tiempos de ejecución para el algoritmo LNO

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	360	630	1000	1700
BCSSTK15	5900	5000	6500	11200
BCSSTK18	4300	4100	6100	10200
BCSSTK23	3800	3200	4200	7300
BCSSTK24	1200	1700	2700	4400
BCSSTK30	42500	35600	40200	73400
LSHP3466	210	460	690	1100

Tabla 5.10: Tiempos de ejecución para el algoritmo *LPO*

- ESPERAPORCOLUMNA() (B): Espera a tener alguna columna disponible, llamémosle s, para modificar la columna j.
- MODIFICA() (C): Modifica la columna j por la columna s, almacena la modificación en una vector temporal denso.
- AGREGAMODIFICACION() (D): Cuando la columna j ya ha recibido todas las modificaciones se vuelca el vector denso que almacena las modificaciones sobre la columna de la matriz.
- NORMALIZA() (E): Calcula la diagonal y normaliza la columna.

La diferencia entre las diferentes estrategias a la hora de construir la lista priorizada de tareas está principalmente condicionada por el tiempo consumido en esperas. Hemos utilizado el programa de análisis de rendimiento SpeedShop para obtener los tiempos consumidos por la función ESPERAPORCOLUMNA() utilizando los tres métodos anteriormente descritos para construir la lista prior-izada de tareas. En las Figuras 5.20

MATRIZ	P=l	P=2	P=4	P=8
BCSSTK14	360	450	700	1100
BCSSTK15	5900	4400	5500	9200
BCSSTK18	4300	3300	4100	7000
BCSSTK23	3800	2900	3600	6200
BCSSTK24	1200	1100	1700	3000
BCSSTK30	42800	28600	30000	51400
LSHP3466	210	290	460	700

Tabla 5.11: Tiempos de ejecución para el algoritmo LCC

y 5.21 se muestran los resultados obtenidos utilizando 2 y 4 procesadores respectivamente. Se ha tomado como tiempo de la función el tiempo consumido por el procesador más lento. Nótese que el método del camino crítico es el que mejor resultados ofrece y es, por tanto, el que utilizaremos de aquí en adelante.

5.7 Mejoras al algoritmo left-looking paralelo

En este caso tenemos dos factores fundamentales que limitan el rendimiento de nuestro programa:

1. Los tiempos de espera: Proponemos la sustitución de la lista enlazada *link* por una estructura de n colas para disminuir las esperas en los procesadores.

2. La localidad de los datos: Proponemos la creación de la lista priorizada de tareas teniendo en cuenta la localidad.

5.7.1 Reducción de los tiempos de espera en el algoritmo left-looking

La lista enlazada *link* no es muy apropiada para el procesamiento paralelo ya que no recoge bien todas las posibles independencias y por tanto limita el posible trabajo paralelo. Consideremos por ejemplo la matriz dada en la Figura 5.22. Supongamos que se ha procesado la columna 0 de dicha matriz. Esta columna modifica las columnas 2, 4 y 7 y dichas modificaciones podrían llevarse a cabo en paralelo, sin embargo, tal y como se construye *link*, no se tiene constancia de que la columna 0 está lista para modificar la 4 hasta que no ha modificado la 2. Igualmente, la columna 0 no está disponible para

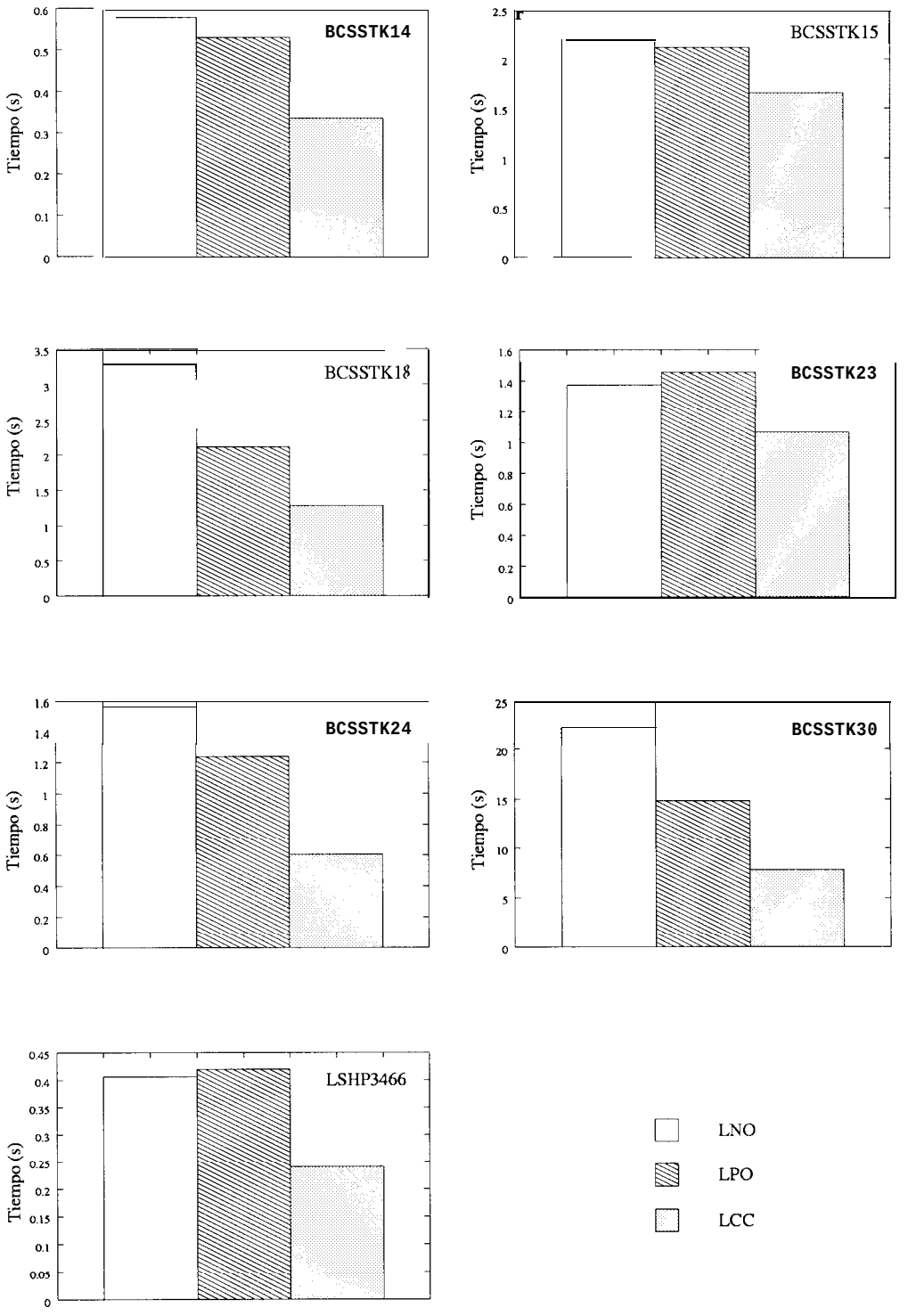

Figura 5.20: Tiempo ***consumido en esperas para los algoritmos*** LNO, *LPO* y LCC y 2 ***procesadores***

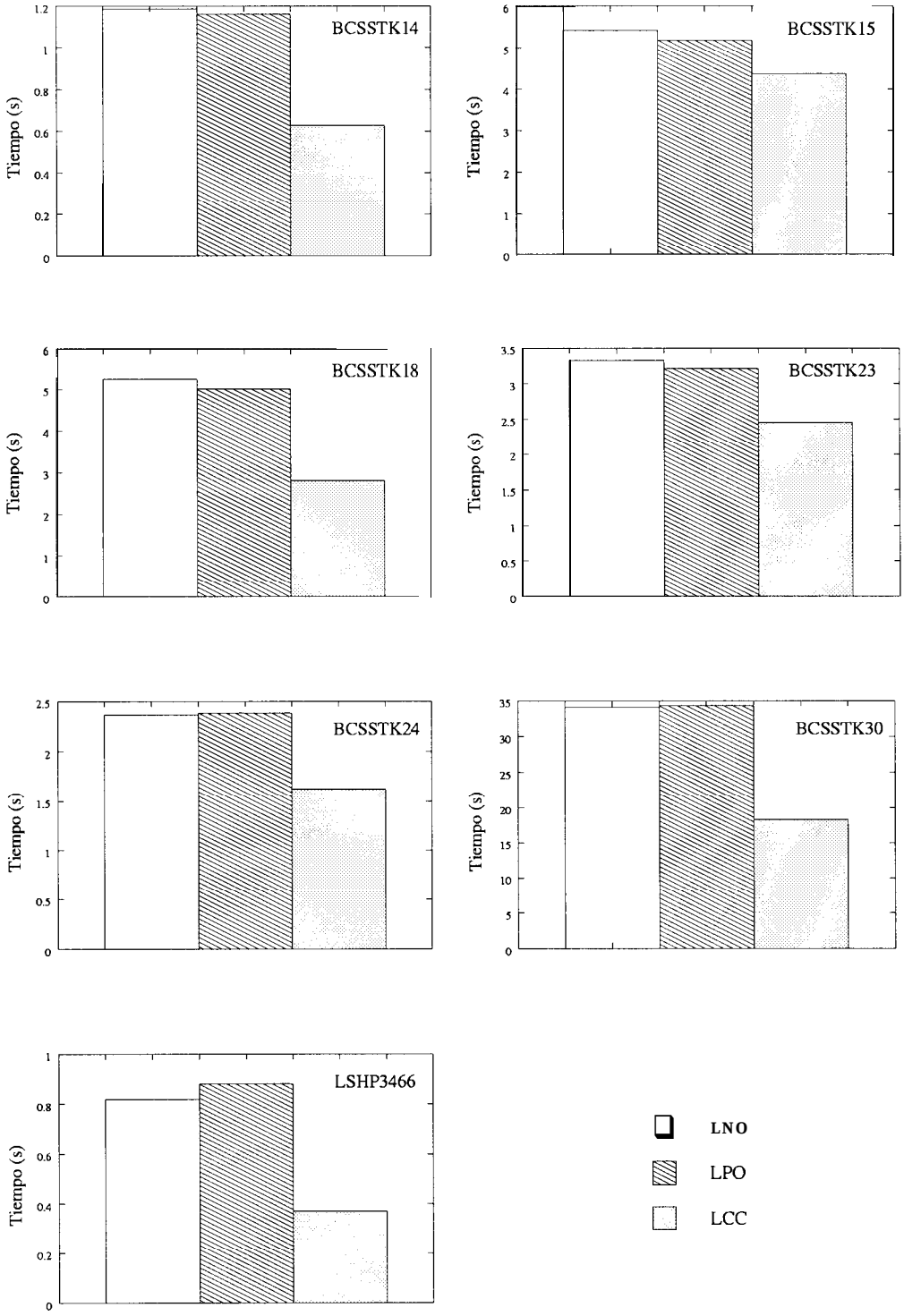

Figura 5.2 1: ***Tiempo consumido en esperas para los algoritmos LNO, LPO y LCC y 4 procesadores***

	0	1	2	3	4	5	6	7
0	X							
1		X						
2	X		X					
3		X		X				
4	X		X		X			
5			X	X	X	X		
6						X	X	
7	X		X		X	X	X	X

Figura 5.22: *Matriz ejemplo: estructura del factor L*

modificar la 7 hasta que no se han modificado la 0 y la 4. Para evitar esta fuente de ineficiencia, e incrementar con ello el paralelismo, se sustituye la lista enlazada *link* por un conjunto de N colas. En concreto, se considera una cola de columnas disponibles por cada columna. En estas colas se almacenan todos los índices de las columnas que están preparadas para modificar cada columna. Así, en la matriz ejemplo, una vez que la columna 0 es procesada se introduce en las colas 2, 4 y 7. Cuando se accede a las columnas 2, 4 o 7 para procesar se consultan sus respectivas colas y se accede a las columnas que están listas para modificarlas. Se podrá modificar 7 por 0 al mismo tiempo que 4 y 2. Se produce de esta forma una mayor independencia entre tareas, con lo que se disminuyen los tiempos de espera. En la Figura 5.23 se muestra el pseudo-código del nuevo algoritmo, al que hemos denominado LNC.

Para minimizar todavía más los tiempos de espera, se podría hacer que cada procesador intentase modificar a la vez más de una columna. Supongamos que cada procesador tiene asignadas dos columnas diferentes, llamémosle j_1 y j_2. Si j_1 no tiene ninguna columna disponible para modificarla, entonces procesa j_2 hasta que no haya columnas disponibles para modificarla, momento en el cual vuelve a ji. De esta forma va conmutando de j_1 a j_2 disminuyendo las posibilidades de que el procesador se quede parado. En contrapartida la implementación de este esquema añade una ligera complejidad al algoritmo lo cual se traduce en un *overhead* asociado a su implementación. La implementación de este idea ha sido realizada de la siguiente forma:

- Se construye la lista priorizada de tareas según el método del camino crítico y se utilizan N colas para mantener los índices de las columnas disponibles para modificar otras.
- Inicialmente se asignan dos tareas a cada procesador, accediendo para ello a la lista priorizada de forma cíclica.
- Los procesadores computan las tareas asociadas conmutando de una a otra tarea

```
1. Construir la lista priorizada de tareas
2. while lista no vacía do
3.     conseguir una columna j de la lista
4.     while nmod[j] > 0 do
5.       esperar hasta que cola[j] no vacía
6.       conseguir un índice columna de la cola, llamémosle k
7.       modificar la columna j con la columna k
8.       nmod[j]=nmod[j]-1
9.      if nmod[j] = 0 then
10.        for each l_sj ≠ 0 do
11.          añadir índice j a cola[s]
12.        endfor
13.      endif
14.    endwhile
15. endwhile
```

Figura 5.23: *Algoritmo left-looking paralelo utilizando N colas (LNC)*

cuando se quedan en espera.

- En el momento que finalizan las computaciones de alguna de las columnas se consigue un nuevo índice columna de la lista priorizada de tareas, de forma que siempre se tiene dos tareas para computar.

Esta idea se puede generalizar a más de dos tareas. Hemos denominado a este algoritmo LNC2.

5.7.1.1 Resultados experimentales

En la Tabla 5.12 se muestran los tiempos de ejecución utilizando N colas y el método del camino crítico para construir la lista priorizada de tareas.

Se observa que los tiempos para el programa secuencial aumentan. Esto es debido al *overhead* asociado con la gestión de las N colas de columnas disponibles. A pesar de ello, el programa paralelo presenta una mayor escalabilidad.

En la Tabla 5.13 se muestran los tiempos de ejecución que se obtienen si a cada procesador se le asignan dos columnas para procesar. Vemos que se consiguen disminuir

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	370	280	280	350
BCSSTK15	6100	3600	2700	2600
BCSSTK18	4400	2700	2300	2400
BCSSTK23	3900	2400	2000	1900
BCSSTK24	1200	800	850	1100
BCSSTK30	44300	25300	17100	16400
LSHP3466	210	180	210	300

Tabla 5.12: *Tiempos de ejecución para el algoritmo* **LNC**

todavía más los tiempos de ejecución. Esta disminución es más apreciable a medida que aumentamos el número de procesadores, ya que en este caso los tiempos asociados a esperas son todavía más críticos.

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	350	270	230	230
BCSSTK15	6000	3700	2400	1900
BCSSTK18	4300	2700	1900	1600
BCSSTK23	3800	2300	1500	1200
BCSSTK24	1100	830	630	630
BCSSTK30	43200	26100	16900	13500
LSHP3466	200	180	180	220

Tabla 5.13: *Tiempos de ejecución para* el *algoritmo LNC2*

Tanto la utilización de N colas, como la utilización de 2 columnas entre las que conmutar, tiene como objetivo disminuir los tiempos de espera en el programa paralelo. En las Figuras 5.24 y 5.25 se muestra los tiempos consumidos en esperas (tiempo consumido por la función ESPERAPORCOLUMNA()) para el algoritmo *left-looking* utilizando la lista enlazada *link*, utilizando N colas, y utilizando N colas y la conmutación entre 2 columnas para 2 y 4 procesadores respectivamente.

5.7.2 Aumento de la localidad de los datos en el algoritmo left-looking

Ahora se dispone de N colas de tareas que son accedidas y modificadas por todos los procesadores. Esto provoca un elevado movimiento de datos y operaciones de coherencia. Una forma de minimizar este efecto es hacer que las colas sean modificadas por

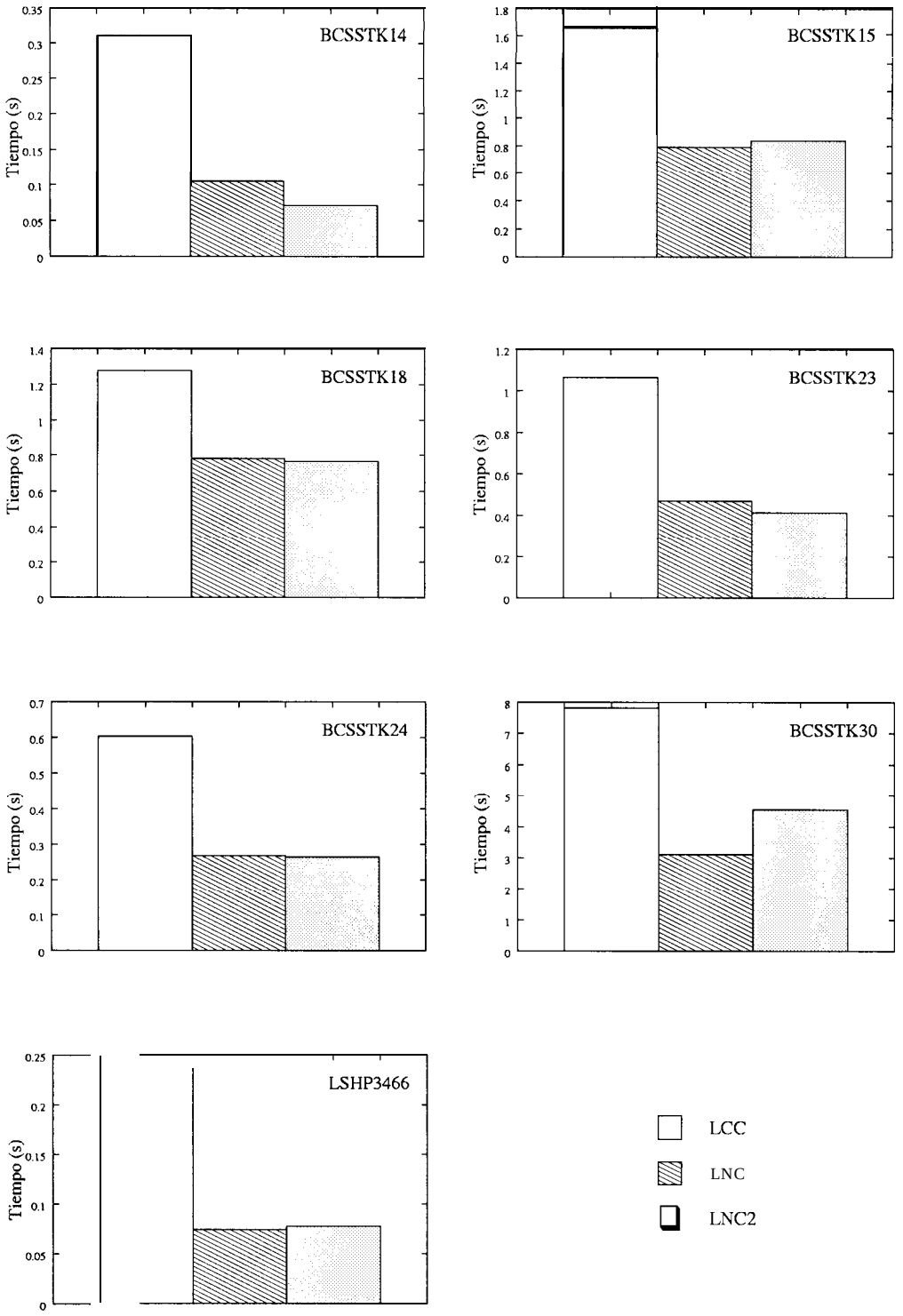

Figura 5.24: ***Tiempo consumido en esperas para los algoritmos*** *LCC, LNC y* LNC2 y 2 ***procesadores***

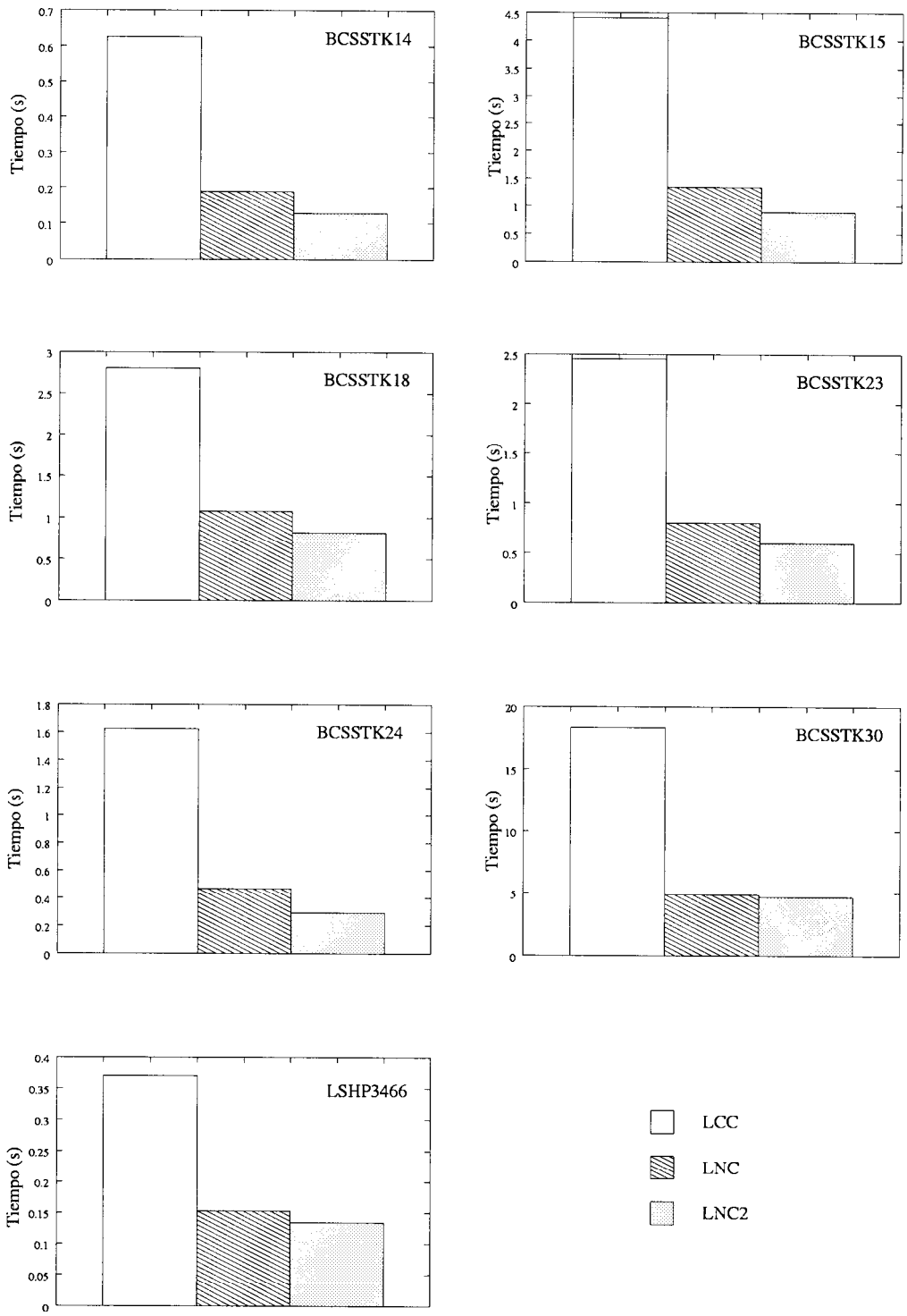

Figura 5.25: ***Tiempo consumido en esperas para los algoritmos*** LCC, LNC y LNC2 y 4 ***procesadores***

el mismo procesador en la mayor medida posible. Esto es equivalente a hacer que un procesador se mueva dentro de un subárbol del árbol de eliminación. Se diseña la lista priorizada de tareas teniendo en cuenta esta idea.

Para ello, se asigna inicialmente un nodo hoja a cada procesador. La idea es que se escojan nodos lo mas lejanos posible en el árbol. De igual modo a como se hizo para el algoritmo *right-looking*, se utiliza la función distancia de la Ecuación 5.1 y el algoritmo de Prim [88, 44] para escoger P nodos que se encuentren a distancia máxima, siendo P el número de procesadores. Se comienza asignando al procesador 0 el nodo que se encuentra a mayor profundidad en el árbol. Se asigna cada uno de estos nodos a cada uno de los procesadores como nodo inicial. A partir de ahí cada procesador crea su propia lista priorizada de columnas empezando por el nodo inicial y recorriendo el árbol por subárboles.

Ahora las listas serán locales. Cada procesador tendrá todas las columnas del árbol ordenadas de diferente forma dentro de su propia lista. Cada procesador accederá a su propia lista, pero se necesitarán etiquetas que informen de que columnas ya han sido procesadas. Estas etiquetas son almacenadas en un *array* global de N elementos protegido por un *LOCK*.

Suponiendo la matriz ejemplo de la Figura 5.9 y 2 procesadores, el procesador 0 tomaría como nodo inicial el nodo más profundo, es decir, el nodo 12, y a partir de él construiría la lista priorizada recorriendo el árbol tal y como se muestra en la Figura 5.26. El procesador 1 tomaría como nodo inicial el nodo más alejado de 12 según la distancia, es decir, el nodo 1, y a partir de él construiría la lista priorizada recorriendo el árbol tal y como se muestra en la Figura 5.27. En ambos casos la numeración de los nodos se corresponden con su posición en la lista priorizada de tareas.

Hemos denominado al algoritmo *left-looking* con este nuevo *scheduling* como algoritmo *LNS*.

5.7.2.1 Resultados experimentales

En la Tabla 5.14 se muestran los tiempos de ejecución para el programa con este nuevo *scheduling*. Vemos que se consigue disminuir los tiempos de ejecución con respecto al algoritmo *LNC, y* este descenso es más importante a medida que aumenta el número de procesadores.

A la vista de los resultados podemos concluir que la orientación *left-looking* del algoritmo es más adecuada para su implementación sobre sistemas de memoria compartida tipo NUMA ya que potencia la localidad de los datos y, consecuentemente, el código paralelo escala mejor.

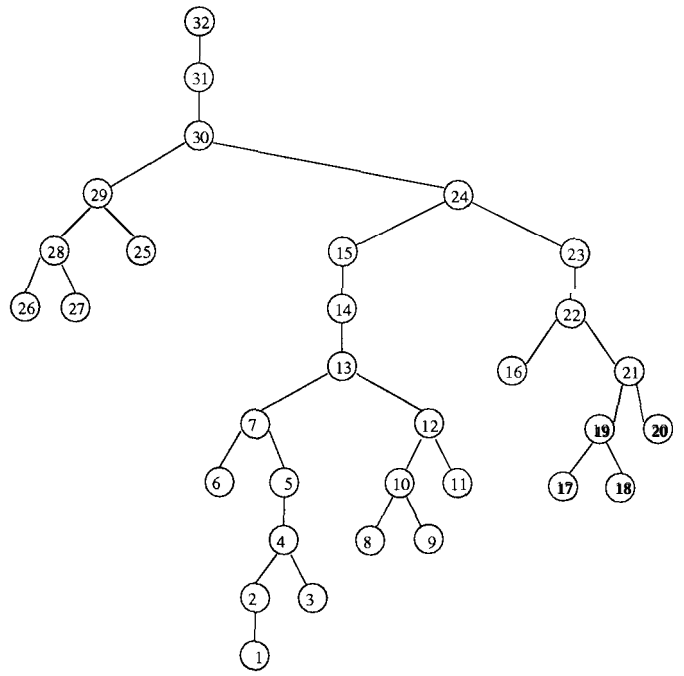

Figura 5.26: ***Lista priorizada para el procesador 0***

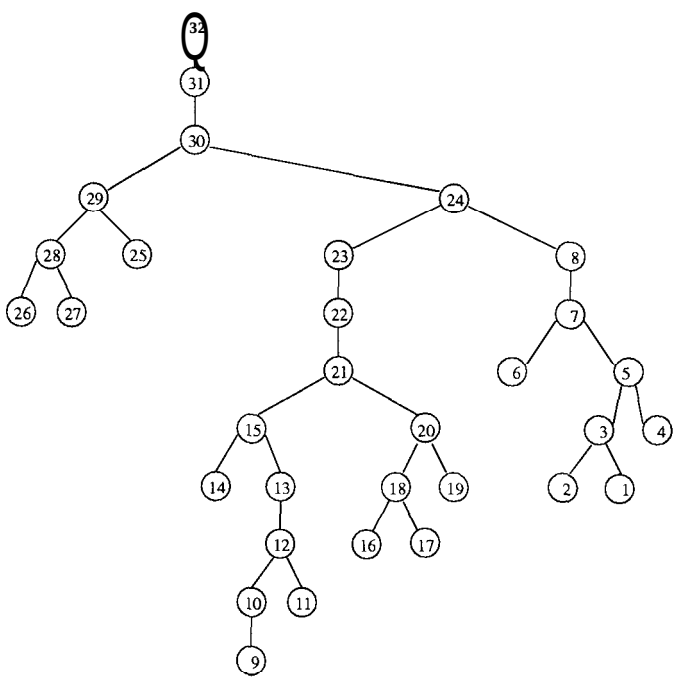

Figura 5.27: ***Lista priorizada para el procesador 1***

MATRIZ	P=1	P=2	P=4	P=8
BCSSTK14	340	290	190	190
BCSSTK15	5900	3500	2100	1400
BCSSTK18	4200	2500	1600	1300
BCSSTK23	3800	2500	1600	1000
BCSSTK24	1100	810	560	430
BCSSTK30	42300	26400	15500	9900
LSHP3466	210	160	130	130

Tabla 5.14: Tiempos de ejecución para el algoritmo LNS

5.8 Resumen

En este capítulo adaptamos las orientaciones *right-Zooking y Zeft-Zooking* de la factorización de Cholesky modificada para su ejecución eficiente sobre sistemas de memoria compartida tipo NUMA y proponemos nuevos *schedulings* basados en el árbol de eliminación de la matriz que aumentan la localidad de los datos y, por tanto, el rendimiento de los códigos paralelos.

Estos nuevos *schedulings se* fundamentan en la explotación de paralelismo de granulatidad gruesa, es decir, el paralelismo existente entre los diferentes subárboles del árbol de eliminación. El objetivo es que cada procesador trabaje, en la medida de lo posible, dentro del mismo subárbol incrementando con ello la localidad.

En este tipo de sistemas, el patrón de accesos a memoria condiciona de forma notable la eficiencia de los programas paralelos. En nuestro caso, se muestra que la orientación *Zeft-Zooking es* más adecuada a este tipo de entornos ya que provoca un menor número de conflictos en el acceso a memoria.

Los resultados obtenidos demuestran que a la hora de programar una máquina de memoria compartida tipo NUMA no sólo hay que intentar balancear la carga y reducir los tiempos de espera, sino que la localidad de los datos juega un papel fundamental en el rendimiento del programa. Por ello, desde el punto de vista del nivel de paralelismo, es el paralelismo de grano medio y grueso el que mejor se adecúa a estas arquitecturas.

Conclusiones y principales aportaciones

Las arquitecturas de los computadores están continuamente evolucionando en busca de mayores velocidades de procesamiento. Para obtener el máximo rendimiento posible del hardware de las máquinas actuales se deben adaptar los códigos a las arquitecturas presentes en dichas máquinas. La principal meta de esta tesis ha sido un estudio de las estrategias de paralelización que mejor se adecúan a la ejecución eficiente de códigos basados en arboles de eliminación sobre los sistemas multiprocesador dominantes en la actualidad, esto es, sistemas de memoria distribuida con topología de interconexión tipo malla y sistemas de memoria compartida-distribuida tipo NUMA. Estudiamos como caso representativo de los códigos basados en arboles de eliminación la factorización de Cholesky modificada dispersa.

Las principales aportaciones de esta memoria se pueden resumir en los siguientes puntos:

- Se ha comprobado el efecto de los reestructuradores automáticos (Polaris, SUIF y PFA) sobre este tipo de códigos, mostrando que el grado de paralelismo que explotan es muy bajo debido a las indirecciones, lo que justifica el esfuerzo en su paralelización manual.

- Comprender y predecir el rendimiento de códigos paralelos no es tarea fácil ya que intervienen muchos más factores que en los códigos secuenciales, como por ejemplo, tiempos de espera, comunicaciones, organización de la memoria de los procesadores, etc. Nosotros hemos determinado las causas fundamentales de sobrecarga en las implementaciones paralelas presentadas sobre las distintas máquinas consideradas, y proponemos alternativas que disminuyen su efecto.

- En el caso de los sistemas de memoria distribuida los factores que más influyen en la eficiencia del programa paralelo son las comunicaciones y el balanceo de la carga. Proponemos esquemas de empaquetamiento de mensajes para reducir tanto el número como el volumen de las comunicaciones y proponemos la utilización

de una distribución de la matriz de forma cíclica por filas y columnas (BCS) que garantiza, en la mayoría de las situaciones, el balanceo de los datos. Modelamos el número de comunicaciones presentes en el código paralelo con el objetivo de predecir la organización de los procesadores que lo minimiza y ofrece, por tanto, un mayor rendimiento.

- En el caso de los sistemas de memoria compartida tipo NUMA la localidad de los datos juega un papel fundamental en el rendimiento de los códigos. Proponemos, para aumentar dicha localidad, nuevos *schedulings* basados en el árbol de eliminación de la matriz y el algoritmo de Prim, los resultados experimentales demuestran su efectividad. Nos centramos también en el problema de la falsa compartición y proponemos nuevos modos de almacenamiento de las matrices dispersas y de los datos compartidos que disminuyen la falsa compartición en este tipo de máquinas.

- Tanto en memoria compartida como distribuida, tenemos una importante limitación al rendimiento paralelo, esto es, las propias dependencias presentes en el código, y que definen el camino crítico. Proponemos modificaciones a los algoritmos *fan-out* y *fan-in* para memoria distribuida y al algoritmo *left-looking* para memoria compartida que disminuyen los tiempos de espera generados por las dependencias.

- Se han analizado los diferentes niveles de paralelismo presentes en la factorización de Cholesky modificada dispersa. Analizamos la explotación de paralelismo de grano fino y medio para los sistemas de memoria distribuida, demostrando que el modelo de paralelismo de granularidad media es el más adecuado para este tipo de sistemas. Programamos los sistemas de memoria compartida tipo NUMA según el modelo de programación de grano medio, y proponemos nuevos *schedulings*, que aumentan el rendimiento, basados en el modelo de programación de granularidad gruesa.

- Aunque todo el estudio lo hemos centrado en la factorización de Cholesky modificada, las estrategias de paralelización planteadas son generalizables a otro tipo de problemas dispersos, tales como otro tipo de factorizaciones (QR, LU, etc..), o la resolución de sistemas lineales. En general, serán aplicables a todos aquellos problemas cuyas dependencias puedan ser representadas por un árbol.

Apéndice A

Paralelización automática de la factorización de Cholesky modificada dispersa

Utilizamos Polaris, SUIF y PFA para comprobar los resultados de la paralelización automática ofrecida por compiladores actuales al ser aplicados a la factorización de Cholesky modificada. Para ello utilizamos un código Fortran secuencial que sirva de entrada a los tres paralelizadores y probamos tanto el código *right-Zuoking* como el *left-looking* (ver capítulo 2 sección 2.4.2). A continuación mostramos los códigos secuenciales y las salidas generadas por cada uno de los compiladores, donde la matriz dispersa es almacenada en formato CCS usando tres vectores (DA, RO y CO), N es la dimensión de la matriz y *diag* mantiene el elemento diagonal de la columna procesada. Para el algoritmo *Zeft-Zooking* utilizamos un vector denso de dimensión N, el vector *sum,* donde se acumulan las modificaciones sobre cada columna y la lista enlazada *link* que nos informa de que columnas modifican la actual.

Cabe destacar que en ningún caso se detecta el paralelismo del lazo externo (lazo 40 tanto en el algoritmo *right-Zooking* como en el *Zeft-Zooking), el* que recorre las columnas, debido a dependencias. Sin embargo, es este' lazo el que tiene mayor paralelismo potencial como se comprueba a lo largo de esta memoria.

Por lo que respecta a los lazos internos, los programas los podemos desglosar en las siguientes operaciones:

- *cmod*(): modificación de una columna por las anteriores. Lazo 90 en el programa *right-looking y 80-90* en el *left-Zooking*
- *cdiv*(): normalización de la columna y cálculo de la diagonal. La podemos des-

glosar a su vez en:

- Cálculo del parámetro gamma: lazos 50 y 100 en los algoritmo *right-looking* y *left-looking* respectivamente.
- Cálculo de la diagonal.
- normalización: lazos 60 y 110 en los algoritmo *right-Zooking y Zeft-Zooking* respectivamente.

La operación de modificación de una columna se lleva a cabo varias veces por columna, mientras que la operación de normalización y cálculo de la diagonal se realiza una única vez por columna. Por tanto, va a ser la operación $cmod()$ la que domine el tiempo de ejecución.

El paralelizador Polaris consigue paralelizar $cmod()$ para el algoritmo *Zeft-Zooking y* la normalización y el cálculo de gamma en ambos algoritmos.

El paralelizador SUIF paraleliza la normalización y el cálculo de gamma para ambos algoritmos.

En cuanto a PFA, sólo es capaz de paralelizar la normalización de la columna tanto en el algoritmo *right-looking* como *lejii-looking*.

En cualquier caso, el mayor grado de paralelismo se alcanza procesando en paralelo columnas independientes y no haciendo en paralelo las operaciones dentro de cada columna, ya que el número de entradas por columna en una matriz dispersa es demasiado reducido como para que la explotación del paralelismo a este nivel sea rentable. Por tanto, ninguno de los paralelizadores evaluados consigue extraer un nivel de paralelismo aceptable.

Factorización de Cholesky modificada. Algoritmo *right-looking*:

```
        DO 40 j=1,N
           gamma=0 .0
           DO 50 i=CO(j)+1,CO(j+1)
              gamma=DMAX1(gamma,DABS(DA(i)))
 50        CONTINUE
C   Cálculo del elemento diagonal
           temp=DA(CO(j))
           temp1=DMAX1(DABS(temp),tao)
           diag=DMAX1(temp1,(gamma*gamma/beta))
           DA(CO(j))=diag
           E(j)=diag-temp
C   Normalización
           DO 60 i=CO(j)+1,CO(j+1)
              DA(i)=DA(i)/diag
 60        CONTINUE
C   Modificación de columnas posteriores
           DO 70 i=CO(j)+1,CO(j+1)
              ii=RO(i)
              DO 80 k=i,CO(j+1)
                 kk=RO(k)
                 jj=CO(ii)
                 DO 90 l=jj,CO(ii+1)
                    IF (RO(l).EQ.kk) THEN
                       DA(l)=DA(l)-DA(i)*DA(k)*DA(CO(j))
                       jj=l
                       GO TO 80
                    ENDIF
 90              CONTINUE
 80           CONTINUE
 70         CONTINUE
 40       CONTINUE
```

Factorización de Cholesky modificada. Algoritmo *left-looking*:

```
        DO 40 j=1,N
 50        IF (link(j).GT.0) THEN
              k=link(j)
              link(j)=link(k)
              DO 60 i=CO(k),CO(k+1)
                 IF (RO(i).EQ.j) THEN
                   entry=i
                   IF ((entry+1).LT.CO(k+1)) THEN
                      nextnz=RO(entry+1)
                   ELSE
                      nextnz=N
                   ENDIF
                   GOTO 70
                ENDIF
 60             CONTINUE
C   Acumulación de las modificaciones en sum
 70             DO 80 i=entry,CO(k+1)
                   sum(RO(i))=sum(RO(i))+DA(entry)*DA(i)*DA(CO(k))
 80             CONTINUE
                IF (nextnz.LT.N) THEN
                  link(k)=link(nextnz)
                  link(nextnz)=k
                ENDIF
                GOTO 50
             ENDIF
C   Adicción de las contribuciones
             DO 90 i=CO(j),CO(j+1)
                DA(i)=DA(i)-sum(RO(i))
                sum(RO(i))=0.0
 90          CONTINUE
C   Cálculo del elemento diagonal
             gamma=0.0
             DO 100 i=CO(j)+1,CO(j+1)
                gamma=DMAX1(gamma,DABS(DA(i)))
100          CONTINUE
             temp1=DA(CO(j))
             temp=DMAX1(DABS(temp1),tao)
             DA(CO(j))=DMAX1(temp,(gamma*gamma)/beta)
             diaq=DA(CO(j))
             E(j)=diag-temp
C   Normalización
             DO 110 i=CO(j)+1,CO(j+1)
                DA(i)=DA(i)/diag
110          CONTINUE
             IF ((CO(j+1)-CO(j)-1).GT.0) THEN
                nextnz=RO(CO(j)+1)
                link(j)=link(nextnz)
                link(nextnz)=j
             ENDIF
 40       CONTINUE
```

Presentamos a continuación un resumen de la salida de Polaris para estos códigos secuenciales. La salida dada por Polaris se compone de código Fortran al que se le han añadido un conjunto de directivas de compilación.

Código generado por Polaris para el algoritmo *right-looking*:

```
                    CSRD$ LOOPLABEL 'RIGHT_do40'
 72 A40-------------      DO j = 1, N, 1
    |                      i = 1+CO(j)
    |                C$DOACROSS IF((CO(1+j)+(-1)*CO(j).GT.5.OR.3.GT.300).AND.3+
    |                C$& 5*(CO(1+j)+(-1)*CO(j)).GT.150),LOCAL(I),SHARE(CO,J)
    |                CSRD$ LOOPLABEL 'RIGHT_do50'
 74 | B50-----------       DO i = 1, CO(1+j)+(-CO(j)), 1
 76 | B50-----------       ENDDO
    |                      i = 1+CO(j)
    |                CSDOACROSS IF((CO(1+j)+(-1)*CO(j).GT.5.OR.3.GT.300).AND.
    |                C$& 3+5*(CO(1+j)+(-1)*CO(j)).GT.150),LOCAL(I),SHARE(CO,J)
    |                CSRD$ LOOPLABEL 'RIGHT_do60'
 83 |B60-----------       DO i = 1, CO(1+j)+(-CO(j)), 1
 85 |B60-----------       ENDDO
    |                      i = 1+CO(j)
    |                CSRD$ LOOPLABEL 'RIGHT_do70'
 86 | B70-----------       DO i = 1, CO(1+j)+(-CO(j)), 1
    | |                     k = CO(j)+i
    | |                CSRD$ LOOPLABEL 'RIGHT_do80'
 88 | | C80---------        DO k = 1, 1+CO(1+j)+(-CO(j))+(-i), 1
    I I I                    l = CO(RO(CO(j)+i))
    | | |              CSRD$ LOOPLABEL 'RIGHT_do90'
 91   ||| D90-------         DO 10 = 1, 1+CO(1+RO(CO(j)+i))+(-CO(RO(CO(j)+i))), 1
    | |  II                   l = (-1)+CO(RO(CO(j)+i))+10
 92  ||  ||                   IF (RO(CO(RO(CO(j)+i))+10-1)+(-RO(CO(j)+i+k-1)).EQ.0) THEN
 96 | | | |                    GOTO 80
 97    || | |                 ENDIF
 98   ||| D90-------          ENDDO
 gg | | | |                80    CONTINUE
 g g  ||C80---------          ENDDO
100 | B70-----------         ENDDO
101 A40-------------        ENDDO
```

Código generado por Polaris para el algoritmo *left-looking*:

```
                      CSRDS LOOPLABEL 'LEFT_do40'
 84 A40--------             DO j = 1, N, 1
 86 |                     50  CONTINUE
 86 |                         IF (1+(-link(j)).LE.0) THEN
 87 |                          k = link(j)
 88 |                          link(j) = link(link(j))
                      CSRDS LOOPLABEL 'LEFT_do60'
 89 | B60------               DO i = CO(k), CO(1+k), 1
 90 | |                        IF (RO(i)+(-j).EQ.0) THEN
 91 | |                         entry = i
 92 | |                         IF (2+i+(-CO(1+k)).LE.0) THEN
 93 | |                          nextnz = RO(1+i)
 94 | |                         ELSE
 95 | |                          nextnz = N
 96 | |                         ENDIF
 97 | |                         GOTO 70
 98 | |                        ENDIF
 99 | B60------               ENDDO
100 |                      70   CONTINUE
                      CSDOACROSS IF((1+CO(1+k)+(-1)*entry.GT.5.OR.1.GT.300).AND.1+5*
                      C$& (1+CO(1+k)+(-1)*entry).GT.150),LOCAL(I),SHARE(ENTRY,CO,K)
                      CSRDS LOOPLABEL 'LEFT_do80'
100 | B80------               DO i = entry, CO(1+k), 1
103 | B80------               ENDDO
104 |                         IF (1+nextnz+(-N).LE.0) THEN
105 |                          link(k) = link(nextnz)
106 |                          link(nextnz) = k
107 |                         ENDIF
108 |                         GOTO 50
109 |                        ENDIF
                               i = CO(j)
                      CSDOACROSS IF((1+CO(1+j)+(-1)*CO(j).GT.5.OR.4.GT.300).AND.4+
                      C$& 5*(1+CO(1+j)+(-1)*CO(j)).GT.150),LOCAL(I),SHARE(CO,J)
                      CSRDS LOOPLABEL 'LEFT_do90'
111|  B90------               DO i = 1, 1+CO(1+j)+(-CO(j)), 1
114 | B90------               ENDDO
                               i = 1+CO(j)
                      CSDOACROSS IF((CO(1+j)+(-1)*CO(j).GT.5.OR.3.GT.300).AND.3+
                      C$& 5*(CO(1+j)+(-1)*CO(j)).GT.150),LOCAL(I),SHARE(CO,J)
                      CSRDS LOOPLABEL 'LEFT_do100'
116 | B100----------        DO i = 1, CO(1+j)+(-CO(j)), 1
118 | B100----------        ENDDO
    |                      i = 1+CO(j)
    |                 CSDOACROSS IF((CO(1+j)+(-1)*CO(j).GT.5.OR.3.GT.300).AND.3+
    |                 C$& 5*(CO(1+j)+(-1)*CO(j)).GT.150),LOCAL(I),SHARE(CO,J)
    |                 CSRD$ LOOPLABEL 'LEFT_do110'
124  |B110----------        DO i = 1, CO(1+j)+(-CO(j)), 1
126 | B110----------        ENDDO
127 |                       IF (2+CO(j)+(-CO(1+j)).LE.0) THEN
128 |                        nextnz = RO(1+CO(j))
129 |                        link(j) = link(RO(1+CO(j)))
130 |                        link(RO(1+CO(j))) = j
131 |                       ENDIF
132 A40-------------       ENDDO
```

Presentamos a continuación un resumen de la salida generada por SUIE SUIF transforma el código fuente en Fortran a código SPMD en C.

Código generado por SUIF para el algoritmo *right-looking*:

```
static void _MAIN___2_func(int _my_id)

    int *i;
    double *gamma;
    double (*matrizda)[1000000];
    int _lb12;
    int _ub13;
    int _my_nprocs;
    struct _BLDR_struct_002 *_my_struct_ptr;

    _my_struct_ptr = _suif_aligned_args;
    i = _my_struct_ptr->_field_0;
    gamma = _my_struct_ptr->_field_1;
    matrizda = _my_struct_ptr->_field_2;
    _lb12 = _my_struct_ptr->_field_3;
    _ub13 = _my_struct_ptr->_field_4;
    _my_nprocs = *_suif_aligned_my_nprocs;


        double __priv_gamma0;
        int __priv_i1;
        double __s2c_tmp2;

        init_max_double(&__priv_gamma0, 11);
        for (__priv_i1 = max((_ub13 + 1 - _lb12) * _my_id / _my_nprocs
+ _lb12,_lb12); __priv_i1 < min((_ub13 + 1 _lb12) * (_my_id + 1) /_my_nprocs
+ _lb12, _ub13 + 1); __priv_i1++)

            __s2c_tmp2 = ((double *)matrizda)[__priv_i1 - 11];
            __priv_gamma0 = max(__priv_gamma0, abs(__s2c_tmp2) );
          }
        suif_reduction_lock(0);
        reduce_max_double(gamma, &__priv_gamma0, ll);
        suif_reduction_unlock(0);
        if (_my_id == _my_nprocs - 1)

            *i = __priv_i1;



    return;


static void _MAIN___3_func(int _my_id)
  {
    int *i;
    double (*matrizda)[1000000];
    double diag;
    int _lb14;
    int _ub15;
    int _my_nprocs;
    struct _BLDR_struct_003 *_my_struct_ptr;

    _my_struct_ptr = _suif_aligned_args;
    i = _my_struct_ptr->_field_0;
```

```
    matrizda = _my_struct_ptr->_field_1;
    diaq = _my_struct_ptr->_field_2;
    _lb14 = _my_struct_ptr->_field_3;
    _ub15 = _my_struct_ptr->_field_4;
    _my_nprocs = *_suif_aligned_my_nprocs;

      {
        int __priv_i0;

        for (__priv_i0 = max((_ub15 + 1 - _lb14) * _my_id / _my_nprocs
+ _lb14, _lb14); __priv_i0 < min((_ub15 + 1 _lb14) * (_my_id + 1) /
_my_nprocs + _lb14, _ub15 + 1); __priv_i0++)

            ((double *)matrizda)[__priv_i0-11]=( (double *)matrizda)[__priv_i0 - 111/diag;
          }
        if (_my_id == _my_nprocs 1)

            *i = __priv_i0;



    return;
  }

extern int MAIN__()

    __s2c_tmp33 = columnasr;
    if (1 <= __s2c_tmp33)

        suif_tmp20 = sqrt(2.2e-16);
        suif_tmp21 = suif_tmp20;
        for (j = 1; j <= __s2c_tmp33; j++)

            gamma = 0.0;
            _lb12 = matrizco[j - 11] + 1;
            _ub13 = matrizco[(long) (j + 0) ];
            _doall_level_result = suif_doall_level();
            if (0 < _doall_level_result)

                double __s2c_tmp0;

                for (i = _lb12; i <= _ub13; i++)

                    __s2c_tmp0 = matrizda[i  111;
                    gamma = max(gamma, abs(__s2c_tmp0));


            else


                  {
                    struct _BLDR_struct_002 *_MAIN2____struct_ptr;

                    suif_start_packing(_MAIN__2__func,  "_MAIN 2 func");
                    *_suif_aligned_task_f = _MAIN2__func;
                    _MAIN2__struct_ptr  =  _suif_aligned_args;
                    _MAIN2__struct_ptr->_field_0  =  &i;
                    _MAIN2__struct_ptr->_field_1  =  &gamma;
                    _MAIN2__struct_ptr->_field_2  =  (double  (*)[1000000])matrizda;
                    _MAIN2__struct_ptr->_field_3  =  _lb12;
                    _MAIN2__struct_ptr->_field_4  =  _ub13;
                    suif_named_doall(_MAIN__2__func,"_MAIN  2  func",2,_ub13-_lb12+1,0);
```

```
        if (any_parallel_io)

            flush_putc();


suif_tmp26 = &matrizda[matrizco[j - 1l] - 1l];
temp = *suif_tmp26;
suif_tmp27 = abs(temp);
__s2c_tmp32 = sqrt((float)(columnasr * columnasr) - 1.0F);
suif_tmp28=gamma*gamma/max(max(landa,epsi/max(__s2c_tmp32,1.0)),2.2e-16);
diag = max(max(suif_tmp27, suif_tmp20), suif_tmp28);
*suif_tmp26 = max(max(suif_tmp27, suif_tmp21), suif_tmp28);
e[j - 1l]= diag - temp;
_lb14 = matrizco[j 1l] + 1;
_ub15 = matrizco[(long)(j + 0) ];
_doall_level_result = suif_doall_level();
if (0 < _doall_level_result)
  {
    for (i = _lb14; i <= _ub15; i++)

        matrizda[i - 1l] = matrizda[i - ll] / diag;


else



        struct _BLDR_struct_003 *_MAIN3____struct_ptr;

        suif_start_packing(_MAIN__3__func, "_MAIN 3 func");
        *_suif_aligned_task_f = _MAIN3func;
        _MAIN3__struct_ptr = _suif_aligned_args;
        _MAIN3__struct_ptr->_field_0 = &i;
        _MAIN3__struct_ptr->_field_1 = (double(*) [1000000])matrizda;
        _MAIN3__struct_ptr->_field_2 = diag;
        _MAIN3__struct_ptr->_field_3 = _lb14;
        _MAIN3__struct_ptr->_field_4 = _ub15;
        suif_named_doall(_MAIN__3__func,"_MAIN 3 func",3,_ub15-_lb14+1,0);
        if (any_parallel_io)

            flush_putc();


suif_tmp29 = matrizco[ (long) (j + 0)];
suif_tmp30 = suif_tmp29;
for (i = matrizco[j - 1l] + 1; i <= suif_tmp30; i++)

    if (i <= suif_tmp30)

        suif_tmp31 = matrizro[i - ll];
        suif_tmp23 = matrizco[suif_tmp31 1l];
        suif_tmp24 = matrizco[ (long) (suif_tmp31 + 0)];
        suif_tmp25 = suif_tmp23 <= suif_tmp24;
        for (k = i; k <= suif_tmp30; k++)

            kk = matrizro[k - 1l];
            jj = suif_tmp23;
            i-12 = suif_tmp24;
            if (!suif_tmp25)
                goto L2644;
```

```
                    l = suif_tmp23;
                  L2745:
                    if (!(matrizro[l - ll] == kk))
                        goto L25;
                    matrizda[l - 1l]= matrizda[l - 1l]
    -matrizda[i   ll]  * matrizda[k - 1l] * matrizda[matrizco[j - ll] - ll];
                    jj = 1;
                    goto L24;
                  L25:
                  L23:
                    l = l + 1;
                    if (!(i_12 < l))
                        goto L2745;
                  L24:
                    goto __done2846;
                  L2644:
                    l = jj;
                  __done2846:
                  L80:;



s_stop(__tmp_string_1, 0);
return 0;
```

Código generado por SUIF para el algoritmo *left-looking*:

```
static void _MAIN___3_func(int _my_id)

    int *i;
    double *gamma;
    double (*matrizda)[1000000];
    int _lb6;
    int _ub7;
    int _my_nprocs;
    struct _BLDR_struct_003 *_my_struct_ptr;

    _my_struct_ptr = _suif_aligned_args;
    i = _my_struct_ptr->_field_0;
    gamma = _my_struct_ptr->_field_1;
    matrizda = _my_struct_ptr->_field_2;
    _lb6 = _my_struct_ptr->_field_3;
    _ub7 = _my_struct_ptr->_field_4;
    _my_nprocs = *_suif_aligned_my_nprocs;


        double __priv_gamma0;
        int __priv_i1;
        double __s2c_tmp2;

        init_max_double(&__priv_gamma0, 11);
        for (__priv_i1 = max((_ub7 + 1 - _lb6) * _my_id / _my_nprocs
+ _lb6, _lb6); __priv_i1 < min((_ub7 + 1 _lb6) * (_my_id + 1) / _my_nprocs
+ _lb6, _ub7 + 1); __priv_i1++)

            __s2c_tmp2 = ((double *)matrizda)[__priv_i1 - 11];
            __priv_gamma0 = max(__priv_gamma0, abs(__s2c_tmp2));

        suif_reduction_lock(0);
        reduce_max_double(gamma, &__priv_gamma0, 11);
        suif_reduction_unlock(0);
        if (_my_id == _my_nprocs - 1)

            *i = __priv_i1;



    return;


static void _MAIN4___func(int  _my_id)
  {
    int *i;
    double (*matrizda)[1000000];
    double diaq;
    int _lb8;
    int _ub9;
    int _my_nprocs;
    struct _BLDR_struct_004 *_my_struct_ptr;

    _my_struct_ptr = _suif_aligned_args;
    i = _my_struct_ptr->_field_0;
    matrizda = _my_struct_ptr->_field_1;
    diaq = _my_struct_ptr->_field_2;
    _lb8 = _my_struct_ptr->_field_3;
```

```
    _ub9 = _my_struct_ptr->_field_4;
    _my_nprocs = *_suif_aligned_my_nprocs;

      {
        int __priv_i0;

        for (__priv_i0 = max((_ub9 + 1 - _lb8) * _my_id / _my_nprocs
+ _lb8, _lb8); __priv_i0 < min((_ub9 + 1 - _lb8) * (_my_id + 1) /_my_nprocs
+ _lb8, _ub9 + 1); __priv_i0++)

            ((double *)matrizda)[__priv_i0-11]=((double *)matrizda)[__priv_i0-11]/diag;
          }
        if (_my_id == _my_nprocs - 1)

            *i = __priv_i0;



    return;


extern int MAIN__()



    __s2c_tmp13 = columnasr;
    for (j = 1;  j <= __s2c_tmp13; j++)
      {
        int _lb6;
        int _ub7;
        int _lb8;
        int _ub9;
        double suif_tmp10;
        double suif_tmp11;
        int suif_tmp12;
        int *suif_tmp13;
        int suif_tmp14;
        int suif_tmp15;
        int suif_tmp16;
        int *suif_tmp17;
        double *suif_tmp18;
        double *suif_tmp19;
        int suif_tmp20;
        int suif_tmp21;
        int *suif_tmp22;
        int __s2c_tmp23;
        int __s2c_tmp24;
        double __s2c_tmp25;

      L50:
        suif_tmp13 = &link[j - ll];
        suif_tmp12 = *suif_tmp13;
        suif_tmp14 = suif_tmp12;
        if (!(0 < suif_tmp14))
            goto  L17;
        k = suif_tmp12;
        *suif_tmp13 = link[suif_tmp14 - ll];
        i_9 = matrizco[(long)(k + 0)];
        suif_tmp15 = matrizco[k - ll];
        if (!(suif_tmp15 <= i-9))
            goto  L3246;
        i = suif_tmp15;
```

```
L3347:
  if (!(matrizro[i - ll] == j))
      goto L20;
  entry__ = i;
  suif_tmp16 = i + 1;
  if (suif_tmp16 < matrizco[(long)(k + 0)])
    {
      nextnz= matrizro[suif_tmp16 - ll];
    }
  else
    {
      nextnz= columnasr;
    }
  goto L19;
L20:
L18:
  i = i + 1;
  if (!(i_9 < i))
      goto L3347;
L19:
  goto __done148;
L3246:
  i = matrizco[k - ll];
__done148:
L70:
  __s2c_tmp23 = matrizco[(long)(k + 0)];
  if (entry__ <= __s2c_tmp23)

      suif_tmp10 = matrizda[entry__ - ll];
      suif_tmp11 = matrizda[matrizco[k - ll] - ll];
      for (i = entry__; i <= __s2c_tmp23; i++)
        {
          sum[matrizro[i - ll] - ll] = sum[matrizro[i - ll] -111
                    + suif_tmp10 * matrizda[i - ll] * suif_tmp11;


  if (nextnz < columnasr)
    {
      suif_tmp17 = &link[nextnz - ll];
      link[k - ll] = *suif_tmp17;
      *suif_tmp17 = k;

  goto L50;
L17:
  __s2c_tmp24 = matrizco[(long)(j + 0)];
  for (i = matrizco[j - ll]; i <= __s2c_tmp24; i++)

      suif_tmp18 = &sum[matrizro[i - ll] - ll];
      matrizda[i - ll] = matrizda[i - ll] - *suif_tmp18;
      *suif_tmp18 = 0.0;

  gamma = 0.0;
  _lb6 = matrizco[j - ll] + 1;
  _ub7 = matrizco[(long)(j + 0)];
  _doall_level_result = suif_doall_level();
  if (0 < _doall_level_result)

      double __s2c_tmp0;

      for (i = _lb6; i <= _ub7; i++)

          __s2c_tmp0 = matrizda[i - ll];
```

```
        gamma = max(gamma, abs(__s2c_tmp0));

else

      {
        struct _BLDR_struct_003 *_MAIN3__structptr;

        suif_start_packing(_MAIN3__func, "_MAIN3__func");
        *_suif_aligned_task_f = _MAIN3__func;
        _MAIN3__struct_ptr = _suif_aligned_args;
        _MAIN3__struct_ptr->_field_0 = &i;
        _MAIN3__struct_ptr->_field_1 = &gamma;
        _MAIN3__struct_ptr->_field_2 = (double (*) [1000000])matrizda;
        _MAIN3__struct_ptr->_field_3 = _lb6;
        _MAIN3__struct_ptr->_field_4 = _ub7;
        suif_named_doall(_MAIN__3__func,"_MAIN__3__func",3, _ub7-_lb6+1,0);
        if (any_parallel_io)

            flush_putc();


suif_tmp19 = &matrizda[matrizco[j - 11] - 111;
__s2c_tmp25 = *suif_tmp19;
temp = max(abs(__s2c_tmp25), tao);
*suif_tmp19 = max(temp, gamma * gamma / beta);
suif_tmp20 = matrizco[j - 11];
diag = matrizda[suif_tmp20 - 11];
e[j - 11] = diag - temp;
_lb8 = suif_tmp20 + 1;
_ub9 = matrizco[(long)(j + 0)];
_doall_level_result = suif_doall_level();
if (0 < _doall_level_result)

    for (i = _lb8; i <= _ub9; i++)

        matrizda[i - 11] = matrizda[i - 11] / diag;


else

      {
        struct _BLDR_struct_004 *_MAIN4__structqtr;

        suif_start_packing(_MAIN4__func, "_MAIN4__func");
        *_suif_aligned_task_f = _MAIN__4_func;
        _MAIN4__structqtr = _suif_aligned_args;
        _MAIN__4_struct_ptr->_field_0 = &i;
        _MAIN__4_struct_ptr->_field_1 = (double (*)[1000000])matrizda;
        _MAIN4__struct_ptr->_field_2 = diaq;
        _MAIN4__struct_ptr->_field_3 = _lb8;
        _MAIN__4_struct_ptr->_field_4 = _ub9;
        suif_named_doall(_MAIN4__func,"_MAIN__4_func",4,_ub9- _lb8+1,0);
        if (any_parallel_io)

            flush_putc();
```

```
        suif_tmp21 = matrizco[j - 1l];
        if  (0  < matrizco[(long) (j + 0)1 -suif_tmp21-1)

            nextnz = matrizro[ (long) (suif_tmp21 + 0)];
            suif_tmp22 = &link[nextnz - 1l];
            link[j - 1l] = *suif_tmp22;
            *suif_tmp22 = j;


    s_stop(__tmp_string_1, 0);
    return 0;


extern int left_()

    MAIN__();
    return 0;
```

A continuación mostramos la salida proporcionada por PFA. PFA le añade al código en Fortran directivas de compilación.

Código generado por PFA para el algoritmo *right-looking*:

```
CSGI$ start 5
        DO j = 1, N, 1
          gamma = 0.0D00
          preg0 = CO(j + 1)
          _lb1 = (CO(j) + 1)
CSGI$ start 6
          DO i = _lb1, preg0, 1
            gamma = MAX(ABS(DA(i)), gamma)
          END DO
CSGI$ end 6
          temp = DA(CO(j))
          temp1 = MAX(ABS(temp), 1.4832396974191326D-08)
          diag = MAX((((gamma * gamma) / beta)), temp1)
          DA(CO(j)) = diag
          pregl = CO(j + 1)
          _lb2 = (CO(j) + 1)
C         PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__1
CSGI$ start 7
C$OMP PARALLEL DO if( ((DBLE(((__mp_sug_numthreads_func$() + -1)) *(DBLE(
C$& ((pregl - _lb2) + 1))  * 2.5D+01)) .GT.(DBLE(
C$& __mp_sug_numthreads_func$()) *((DBLE(__mp_sug_numthreads_func$ ()) *
C$&  1.23D+02) + 2.6D+03)))), private(i), shared(DA, _lb2, diag, pregl)
          DO i = _lb2, pregl, 1
            DA(i) = (DA(i) / diag)
          END DO
CSGI$ end 7
          preg2 = CO(j + 1)
CSGI$ start 8
          DO i = (CO(j) + 1), preg2, 1
            ii = RO(i)
            preq3 = CO(j + 1)
CSGIS start 3
            DO k = i, preg3, 1
              kk = RO(k)
              jj = CO(ii)
              preg4 = CO(ii + 1)
CSGIS start 10
              DO l = jj, preg4, 1
                IF(RO(l) .EQ. kk) THEN
                  DA(l) = (DA(l) -(DA(CO(j)) *(DA(i) * DA(k))))
                  GO TO 8
                ENDIF
              END DO
CSGIS end 10
8             CONTINUE
            END DO
CSGI$ end 9
          END DO
CSGI$ end 8
        END DO
CSGI$ end 5
```

Código generado por PFA para el algoritmo left-looking:

```
CSGI$ start 6
        DO j = 1, N, 1

C
C         whirl2f:: DO loop with termination test after first iteration
Ctmp0 = .TRUE.
          DO WHILE(tmp0)
6           CONTINUE
            goto_L6 = 0
            IF(link(j) .GT. 0) THEN
              k = link(j)
              link(j) = link(k)
              preg0 = CO(k + 1)
CSGI$ start 7
              DO i = CO(k), preg0, 1
                IF(RO(i) .EQ. j) THEN
                  entry = i
                  IF(CO(k + 1) .GT.(i + 1)) THEN
                    nextnz = RO(i + 1)
                  ELSE
                    nextnz = N
                  ENDIF
                  GO TO 8
                ENDIF
              END DO
CSGIS end 7
8             CONTINUE
              pregl = CO(k + 1)
CSGIS start 8
              DO i = entry, pregl, 1
                sum(RO(i)) = (sum(RO(i)) +(DA(CO(k)) *(DA(i) * DA(entry))))
              END DO
CSGIS end 8
              IF(N .GT. nextnz) THEN
                link(k) = link(nextnz)
                link(nextnz) = k
              ENDIF
              goto_L6 = 1
            ELSE
              preg2 = CO(j + 1)
              _lb1 = CO(j)
CSGIS start 9
              DO i = _lb1, preg2, 1
                DA(i) = (DA(i) - sum(RO(i)))
                sum(RO(i)) = 0.0D00
              END DO
CSGIS end 9
```

```
              gamma = 0.0D00
              preg3 = CO(j + 1)
              _lb2 = (CO(j) + 1)
CSGI$ start 10
              DO i = _lb2, preg3, 1
                gamma = MAX(ABS(DA(i)), gamma)
              END DO
CSGIS end 10
              temp1 = DA(CO(j))
              temp = MAX(ABS(temp1), 1.4832396974191326D-08)
              DA(CO(j)) = MAX(( ((gamma * gamma)) / beta), temp)
              diag = DA(CO(j))
              preg4 = CO(j + 1)
              _lb3 = (CO(j) + 1)
C             PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__2
CSGIS start 11
C$OMP PARALLEL DO if(((DBLE((__mp_sug_numthreads_func$() + -1)) * (DBLE(
C$& ((preg4 - _lb3) + 1)) * 2.5D+01)) .GT. (DBLE(
C$& __mp_sug_numthreads_func$()) *((DBLE(__mp_sug_numthreads_func$()) *
C$&  1.23D+02) + 2.6D+03)))), private(i), shared(DA, _lb3, diag,
C$&  preg4)
              DO i = _lb3, preg4, 1
                DA(i) = (DA(i) / diag)
              END DO
CSGIS end 11
              IF((CO(j + 1) - CO(j)) .GT. 1) THEN
                nextnz = RO(CO(j) + 1)
                link(j) = link(nextnz)
                link(nextnz) = j
              ENDIF
            ENDIF
            tmp0 = goto_L6
          END DO
        END DO
CSGI$ end 6
```

Bibliografía

[1] A.V. Aho, M.R. Garey and J.D. Ullman, *The transitive reduction of* a *directed graph,* SIAM J. Computing, 1(2), pp.131-137, 1972.

[2] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *Estructuras de datos y algoritmos,* Addison-Wesley, 1988.

[3] Fernando L. Alvarado and Robert Schreiber, *Optimal parallel solution of sparse triangular systems,* SIAM Journal on Scientific Computing, 14(2), pp.446-460, 1993.

[4] P.R. Amestoy, T.A. Davis and I.S. Duff, *An approximate minimum degree ordering algorithm, SIAM* J. Matrix Analysis and Applications, 17(4), pp.886-905, 1996.

[5] P. Amestoy, I.S. Duff, J.Y. L'Excellent and P. Plechác, *PARASOL. An integmted programming environment for parallel sparse matrix solvers,* Technical Report TR/PA/98/13, CERFACS, France, 1998. (`http://192.129.37.12/parasol/`)

[6] E. Artiaga, N. Navarro, X. Martorell and Y. Becerra, *Implementing PARMACS Macros for Shared Memory Multiprocessor Environments,* Technical Report UPC-DAC-1997-07, Departament d'Arquitectura de Computadors, Univ. Politècnica de Catalunya, Spain, 1997.

[7] R. Asenjo, *Factorización LU de matrices dispersas en multiprocesadores,* Tesis doctoral, Departamento de Arquitectura de Computadores, Universidad de Málaga, 1997.

[8] R. Asenjo, L.F. Romero, M. Ujaldón and E.L. Zapata, *Sparse block and cyclic data distributionsfor matrix computations,* En High Performance Computing: Technology, methods and applications, Elsevier Science B.V., pp.359-377, 1995.

[9] C. Ashcraft, *Compressed graphs and the minimum degree algorithm, SIAM* Journal on Scientific Computing, 16, pp.1404-1411, 1995.

[10] C. Ashcraft, S.C. Eisenstat and J.W.H. Liu, *A fan-in algorithm for distributed sparse numerical factorization,* SIAM J. Sci. Stat. Comput., 1 1(3), pp.593-599, 1990.

[11] C. Ashcraft and J. Liu, *Robust ordering of sparse matrices using multisection,* SIAM J. on Matrix Analysis and Applications, 19(3), pp.816-832, 1998.

[12] Cleve Ashcraft and Joseph Liu, *SMOOTH: Sparse Matrix Object-Oriented Ordering Methods,* November 1996.
(http://www.cs.yorku.ca/~joseph/SMOOTH.html)

[13] S.T. Barnard and H. Simon, *A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems*, En Proceedings of the Sixth SIAM conference on parallel processing for scientific computing, SIAM Press, pp.71 1-718, 1993.

[14] R. Barret, M. Berret, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods, SIAM* Publications, Philadelphia, 1994.

[15] W. Blume, R. Doallo, R. Eigenmanm, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, P. Tu, *Parallel programming with Polaris*, IEEE Computer, 29(12), pp.78-82, 1996

[16] T. Chan, J. Gilbert and S.H. Teng, *Geometrical spectral partitioning,* Technical Report CSL-94-15. Palo Alto Research Center, Xerox Corporation, California 1994.

1171 E. Chu, A. George, J.W.H. Liu and E.G.Y. Ng, *User's guide for SPARSPAK-A: Waterloo sparse linear equationspackage,* Technical Report CS-84-36, University of Waterloo, Waterloo, Ontario, 1984.

[18] E.G. Coffman Jr., *Computer and job shop scheduling theory,* Wiley-Interscience, New York, 1976.

[19] D.E. Culler, J.P. Singh and A. Gupta, *Parallel computer architecture: a hardware/software approach,* Morgan Kaufmann Publishers, Inc., 1999.

[20] Michel J. Daydé, Jean-Yves L'Excellent and Nicholas I.M. Gould, *Element-by-element preconditioners for large partially separable optimization problems, SIAM* Journal on Scientific Computing, 18(6), pp.1767-1787, 1997.

[21] I.S. Duff, I.M. Erisman and J.K. Reid, *Direct methods for sparse matrices,* Oxford University Press, London, 1986.

[22] I.S.Duff, R.G. Grimes, and J.G.Lewis, *User's guidefor the Harwell-Boeing sparse matrix collection*, Technical Report TR-PA-92-96, CERFACS, October 1992.

[23] I.S. Duff and J.K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Transactions on Mathematical Software, 9, pp.302-325, 1983.

[24] I.S. Duff and J.K. Reid, MA47 *a Fortran code for direct solution of indefinite sparse symmetric linear systems,* Technical Report RAL-95-001, Rutherford Appleton Laboratory, 1995.

[25] S. Eisenstat, M. Gursky, M. Schultz and A.H. Sherman, *The Yale sparse matrix package I: The symmetric codes*, Int. J. Numer. Meth. in Eng., 18, pp.1 145- 1151, 1982.

[26] M.J. Flynn, *Very high-speed computing systems,* Proceedings of the IEEE, pp.1901-1909, 1966.

[27] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. W. Tseng and M. Wu, *Fortran D language specification*, Technical Report CRPC-TR90079, Computer Science Dept., Rice University, 1990.

[28] Fujitsu Laboratories Ltd, *AP1000 Program Development Guide. C-Language Interface*, Third Edition, July 1993.

[29] Fujitsu Laboratories Ltd, *AP1000 Program Development Guide. Fortran-Language Interface*, Third Edition, July 1993.

[30] Fujitsu Laboratories Ltd, *Casim Users's Guide,* Fourth Edition, August 199 1.

[31] A. Geist, A. Beguelin, J. Dongan-a, W. Jiang, R. Manchek and V. Sunderam, *PVW 3 User's guide and reference manual,* Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1994.

[32] C.A. Geist and E. Ng, *Task scheduling forparallel sparse Cholesky factorization,* Int. Journal of Parallel Programming, 18(4), pp.291-314, 1989.

[33] J. George, *Nested dissection of a regular finite element mesh, SIAM* J. Numerical Analysis, 10, pp.345-363, 1973.

1341 A. George, M. Heath, J. Liu and E. Ng, *Solution of sparse positive definite systems on a shared memory multiprocessor*, Int. J. Parallel Programming, 15, pp.309-325, 1986.

1351 A. George, M.T. Heath, J. Liu and E. Ng, *Symbolic Cholesky factorization on a local-memory multiprocessor,* Parallel Computing, 5, pp.85-95, 1987.

[36] A. George, M.T. Heath, J. Liu and E. Ng, *Sparse Cholesky factorization on a local-memory multiprocessor, SIAM* J. Sci. Statist. Comput., 9, pp.327-340, 1988.

[37] A. George, M.T. Heath, J. Liu and E. Ng, *Solution of sparse positive definite systems on a hypercube,* J. Comput. Appl. Math., 27, pp.129-156, 1989.

[38] A. George and J.W.H. Liu, *An automatic nested dissection algorithm for irregular finite elementproblems, SIAM* J. Numer. Anal., 15, pp.1053-1069, 1978.

[39] A. George and J.W.H. Liu, *An optima1 algorithm for symbolic factorization of symmetric matrices,* SIAM J. Comput., 9, pp.583-593, 1980.

[40] Alan George and Joseph W.H. Liu, *Computer Solution of Large Sparse Positive Definite Systems,* Prentice Hall, Englewood Cliffs, NJ, 1981.

[41] Alan George and Joseph W.H. Liu, *The evolution of the minimum degree ordering algorithms,* SIAM Review, 31(1), pp.1-19, 1989.

[42] Alan George and Joseph Liu, *An object-oriented approach to the design of a user interface for a sparse matrixpackage,* SIMAX (pendiente de publicación).

[43] A. George, J. Liu and E. Ng, *Communication results forparallel sparse Cholesky factorization on a hypercube,* Parallel Computing, 10, pp.287-298, 1989.

[44] A. Gibbons, *Algorithmic graph theory,* Cambridge University Press, 1985.

[45] Philip E. Gill and Walter Murray, *Newton-type methods for unconstrained and linearly constrained optimization,* Math. Programming, 28, pp.331-350, 1974.

[46] Philip E. Gill, Walter Murray and Margaret H. Wright, *Practica1 optimization,* Academic Press, London, 198 1.

[47] P. González, J.C. Cabaleiro, T.F. Pena, *Solving sparse triangular systems on distributed memory multicomputers,* En Proceedings sixth EUROMICRO Workshop On Parallel and Distributed Processing (PDP'98), pp.470-478, 1998.

[48] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, V. Kumar, *Design* and *Implementation of a Scalable Parallel Direct Solver for Sparse Symmetric Positive Definite Systems: Preliminary Results*, En Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, USA, 1997. (http ://www--users.cs.umn.edu/~mjoshi/pspases/)

[49] A. Gupta and V. Kumar, *Parallel algorithmsforforward elimination and backward substitution in direct solution of sparse linear systems,* En Supercomputing'95 Proceedings, 1995.

[50] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.W. Liao, E. Bugnion and M.S. Lam, *Maximizing multiprocessorperformance with the SUIF Compiler,* IEEE Computer, 29(12), pp.84-89, 1996.

[51] M.T. Heath, E. Ng and B.W, Peyton, *Parallel algorithms for sparse linear systems*, SIAM Review, 33, pp.420-460, 1991.

[52] B. Hendrickson and E. Rothberg, *Improving the runtime and quality of nested dissection ordering,* SIAM Journal on Scientific Computing, 20(2), pp.468-489, 1999.

[53] D.B. Heras, M. Martín, V. Blanco, M. Amor, F. Argüello and F.F. Rivera, *Iterative methods for the solution of linear systems on the AP1000 multiprocessor,* PCW'95, pp.259-265, 1995.

[54] J.L. Hennesy and D.A. Patterson, *Computer architecture. A quantitative approach,* Morgan Kaufmann Publishers, Inc., second edition, 1996.

[55] High Performance Forum, *High performance language specification*, version 2.0, 1996.

[56] H. Ishihata, T. Horie and T. Shimizu, *Architecture for the AP1000 highly parallel computer,* FUJITSU Scientific & Technical Journal, 29(1), pp.6-14, 1993.

[57] J.A.G. Jees and H.G.M. Kees, *A data structure for parallel L/U decomposition,* IEEE Trans. Comput., 31, pp.231-239, 1982.

[58] W.H. Kohler, *A preliminar-y evaluation of the critica1 path method for scheduling tasks on multiprocessor systems,* IEEE Transactions on Computers, 24, pp.1235-1238, 1975.

[59] J. Laudon, D. Lenoski, *The SGI Origin: A ccNUMA Highly Scalable Server*, Computer Architecture News, 25(2), pp.241-252, 1997.

[60] John G. Lewis, Barry W. Peyton and Alex Pothen, *A fast algorithm for reordering sparse matrices for parallel factorization,* SIAM J. Sci. Stat. Comput., 10(6), pp.1 146-1173, 1989.

[61] Xiaoye S. Li, *Sparse Gaussian Elimination on High Performance Computers,* PhD thesis, Departament of Computer Sciencie, Universisty of California at Berkeley, 1996.

[62] R.J. Lipton, D.F. Rose and R.E. Tarjan, *Generalized nested dissection, SIAM* J. Numerical Analysis, *16,* pp.346-358, 1979.

[63] J.W.H. Liu, *A compact row storage scheme for Cholesky factors using elimination trees,* ACM Trans. Math. Software, *12, pp. 127-148, 1986.*

[64] J.W.H. Liu, *Computational models and task scheduling for parallel sparse Choleskyfactorization,* Parallel Computing, 3, pp.327-342, 1986.

[65] Joseph W.H. Liu, *A note on sparse factorization in a paging environment, SIAM* J. Sci. Stat. Comput., 8(6), 1987.

[66] J.W.H. Liu, *Equivalent sparse matrix reordering by elimination tree rotations, SIAM* J. Sci. Statist. Comput., 9, pp.424-444, 1988.

[67] J.W.H. Liu, *Reordering sparse matricesforparallel elimination,* Parallel Computing, ll, pp.73-91, 1989.

1681 Joseph W.H. Liu, The *role of elimination trees in sparse factorization, SIAM* J. Matrix Anal. Appl., ll(l), pp.134-172, 1990.

[69] Joseph W.H. Liu, *The multifrontal method for sparse matrix solution: Theory and practice, SIAM* Review, 34(1), pp.82-109, 1992.

[70] E.L. Lusk, *Portable Programs for Parallel Processors,* Holt Rinchart and Winston 1987.

[71] M.J. Martín, V. Blanco y F.F. Rivera, *Paralelización del algoritmo de Cholesky modificado para matrices dispersas,* VII Jornadas de Paralelismo, Santiago de Compostela, pp.479-494, 1996.

[72] M.J. Martín, V. Blanco and F.F. Rivera, *Sparse modified Cholesky factorization on the AP1000, PCW'96,* Kawasaki, Japan, pp.SC1-SC5, 1996.

[73] M.J. Martín, 1. Pardines and F.F. Rivera, *Optimum mapping of sparse matrices on processor meshes for the modified Cholesky algorithm, PCW'98,* Singapore, pp.175-181, 1998.

[74] M.J. Martín, 1. Pardines and F.F. Rivera, *Scheduling for algorithms based on elimination trees on NUMA systems,* Euro-Par'99 (aceptado).

[75] M.J. Martín 1. Pardines and F.F. Rivera, *Left-looking strategy for the modified Cholesky factorization on NUMA multiprocessors,* ParCo99 (aceptado).

[76] M.J. Martín and F.F. Rivera, *Modelling an asynchronous parallel modijed Cholesky factorization for sparse matrices,* International Journal of Parallel and Distributed Systems and Networks, 1(2), pp.93-101, 1998.

[77] M.J. Martín and F.F. Rivera, *Modified Cholesky factorization of sparse matrices on distributed memory systems: fan-in andfan-out algorithms with reduced idle times,* (enviado para su publicación).

[78] Message Passing Interface Forum. *MPI: A message-passing interface standard,* version 1.1, 1995.

II791 Message Passing Interface Forum. MPI-2: *Extensions to the message-passing interface*, 1996.

[80] 1. Pardines Lence, *Algoritmo paralelo para la factorización de Cholesky modificada,* Memoria de Licenciatura, Dpto. Electrónica y Computación, Univ. Santiago de Compostela, Junio 1997.

[81] 1. Pardines, M. Martín, M. Amor and F.F. Rivera, *Static mapping of the multifrontal method applied to modijed Cholesky factorization for sparse matrices,* Parallel Computing: Fundamentals, Applications and New Directions, E.H. D'Hollander, G.R. Joubert, EJ. Peters and U. Trottenberg editors, Elsevier Science B.V., pp.731-735, 1998.

[82] I. Pardines, M. Martín y EE Rivera, *Particionamiento en entornos distribuidos de computaciones basadas en arboles de eliminación,* Seid99, Santiago de Compostela, pp.221-228, 1999.

[83] S.V. Parter, *The use of linear graphs in Gauss elimination,* Siam Review, 3, pp.1 19-130, 1961.

[84] D.M. Pase, T. McDonald and A. Meltzer, The CRAFT *Fortranprogramming model,* Scientific Programming, 3, pp.227-253, 1994.

[85] Sergio Pissanetzky, *Sparse Matrix Technology,* Academic Press, Inc., 1984.

[86] J.J. Pombo García, *Estudio y comparación de las técnicas de reordenamiento para matrices dispersas,* Memoria de Licenciatura, Dpto. Electrónica y Computación, Univ. Santiago de Compostela, Julio 1997.

[87] A. Pothen, H.D. Simon, L. Wang and S. Barnard, *Towards a fast implementation of spectral nested dissection,* In Supercomputing, ACM press, pp.42-5 1, 1992.

[88] R.C. Prim, *Shortest connection networks and some generalisations,* Bel1 System Tech. J., 36, pp.1389-1401, 1957.

[89] Jelica Protic, Milo Tomasevic and Veljko Milutinovic, *Distributed Shared Memory. Concepts and Systems,* IEEE Computer Society, 1997.

[90] L.F. Romero and E.L. Zapata, *Data distributions for sparse matriz vector multiplication,* Parallel Computing, 21(4), pp.583-605, 1995.

[91] D.J. Rose, *Symmetric elimination on sparse positive definite systems and the potential flow networkproblem,* PhD thesis, Applied Math., Harvard Univ., 1970.

[92] E. E. Rothberg, *Exploiting the memory hierarchy in sequential andparallel sparse Cholesky factorizution,* PhD thesis, Department of Computer Science, Stanford University, 1992.

[93] Yousef Saad, *Iterative Methods for Sparse Linear Systems,* PWS Publishing Company, 1996.

[94] Tamar Schlick, *Modified Cholesky factorizations for sparse preconditioners,* SIAM J. Sci. Comput., 14, pp.424-445, 1993.

[95] Robert B. Schnabel and Elizabeth Eskow, *A new modified Cholesky factorization,* SIAM J. Sci. Stat. Comput., 11, pp.1 136-1158, 1990.

[96] R. Schreiber, *A new implementation of sparse Gaussian elimination,* ACM Trans. Mathematical Software, 8, pp.256-276, 1982.

[97] S.L. Scott, *Synchronization and communication in the Cray T3E multiprocessor,* ASPLOS-VII, Cambridge, MA, October 1996.

[98] Silicon Graphics, Inc., SGI, Inc., Mountain View, CA. *Iris Power C, User's Guide, 1996.*

[99] Silicon Graphics, Inc., SGI, Inc., Mountain View, CA. *Iris Power Fortran, User's Guide, 1996.*

[100] Silicon Graphics, Inc., SGI, Inc., Mountain View, CA, *Performance Tuning for the Origin2000,* 1997.

[101] D. Sima, T. Fountain and P. Kacsuk, *Advanced computer architectures. A design space approach,* Addison-Wesley Publishing Company, 1997.

[102] D. Sitsky, *Implementation of MPI on the Fujitsu AP1000: Technical Details,* Technical Report, Australian National University, September 1994.

[103] D. Sitsky, D. Walsh and C. Johnson, *Zmplementation and performance of the MPI Message Passing Interface on the Fujitsu AP1000 multicomputer,* Australian Computer Science Communications, [redacted] 1995.

[104] B. Speelpenning, *The generulized element method,* Technical Report UIUCDCS-R-78-946, Department of Computer Science, University of Illinois, Urbana, IL, November 1978.

[105] P. Tang and P.-C. Yew, *Processor self-scheduling for multiple nested parallel loops*, En Proceedings 1986 International Conference on Parallel Processing, pp.528-535, Aug. 1986.

[106] Jyh-Jong Tsay, *Mapping tree-structured computations onto mesh-connected arrays of processors,* En Proceedings of the fourth IEEE Symposium on Parallel and Distributed Processing 92, DEC 1-4, pp.77-84, Arlington, Texas, 1992.

[107] Theoretical Studies Department of AEA industrial Technology, *Harwell Subroutine Library: A Catalogue of Subroutines (Release 11)*, Technical Report, Theoretical Studies Department, AEA Industrial Technology, 1993.

[108] Thinking Machines Corporation, CM *Fortrun language reference manual, 1994.*

[109] W. F. Tinney and J. W. Walker, *Direct Solution of Sparse Network Equations by Optimally Ordered Triangular Factorization,* Proc. IEEE, 55, pp. 1801-l 809, 1967.

[110] TOP500 Supercomputing Sites, `http: //www. top500. org.`

[ll 1] J. Touriño, *Parallelization and compilation issues of sparse QR algorithms,* PhD thesis, Departamento de Electrónica y Sistemas, Universidad de A Coruña, 1998.

[112] 0. Wing and J.W. Huang, *A computational model of parallel solution of linear equations,* IEEE Trans. Comput., 29, pp.632-638, 1980.

[113] M. Wolfe, *Highperformance compilersforparallel computing,* Addison-Wesley Publishing Company, 1996

[114] M. Yannakakis, *Computing the minimum fill-in is NP-complete, SIAM* J. Alg. Disc. Meth., 1, pp.77-79, 1981.

[115] M. Zagha, B. Lar-son, S. Turner and M. Itzkowitz, *Performance analysis using the MIPS R10000 performance counters,* En Supercomputing'96 Conference Proceedings, 1996.

[116] H. Zima, P. Brezany, B. Chapman, P. Mehrotra and A. Schwald, *Vienna Fortran - A language specification, version 1.1,* Technical Report ACPC-TR92-4, Austrian Center for Parallel Computation, Univ. Vienna, Austria, 1992.

[117] H. Zima and B. Chapman, *Supercompilers for parallel and vector computers*, ACM Press, 1991.