BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Universidad de Alicante

Algoritmos divide y vencerás para la resolución de sistemas lineales tridiagonales en un computador BSP

Leandro Tortosa Grau

Tesis de Doctorado

Facultad: Escuela Politécnica Superior

Director: Dr. Joan Josep Climent Coloma

2000

Universidad de Alicante

Departamento de Ciencia de la Computación e Inteligencia Artificial

Algoritmos divide y vencerás para la resolución de sistemas lineales tridiagonales en un computador BSP

TESIS DOCTORAL

Presentada por: Leandro Tortosa Grau

Dirigida por: Joan Josep Climent Coloma

Universidad de Alicante

Departamento de Ciencia de la Computaci'on e Inteligencia Artificial

Algoritmos divide y vencerás para la resolución de sistemas lineales tridiagonales en un computador BSP

Memoria presentada para optar al grado de doctor Ingeniero en Informática por Leandro Tortosa Grau.

Alicante, 14 de Diciembre de 1999

Don Joan Josep Climent Coloma, Profesor Titular de Universidad del Departamento de Ciencia de la Computaci'on e Inteligencia Artificial de la Universidad de Alicante,

CERTIFICA:

Que la presente memoria *Algoritmos divide y vencerás para la resolución de sistemas lineales tridiagonales en un computador BSP*, ha sido realizada bajo su dirección, en el Departamento de Ciencia de la Computaci'on e Inteligencia Artificial de la Universidad de Alicante, por el licenciado Don Leandro Tortosa Grau, y constituye su tesis para optar al grado de Doctor en Ingeniero en Informática. Para que conste, en cumplimiento de la legislación vigente, autoriza la presentación de la referida tesis doctoral ante la comisión de Doctorado de la Universidad de Alicante, firmando el presente certificado.

Alicante, 14 de Diciembre de 1999.

Fdo.: Joan Josep Climent Coloma

A Carmen, su apoyo constante y confianza en mí
me han proporcionado las fuerzas necesarias
para seguir hacia adelante.

En primer lugar, quiero expresar mi agradecimiento al profesor Joan Josep Climent por todos los conocimientos, consejos y amistad que me ha transmitido a lo largo de todos estos años. Su paciencia y habilidad han sido determinantes para que este trabajo llegara a buen puerto, así como su enorme capacidad de trabajo y disciplina.

En segundo lugar, agradecer a Antonio Zamora su amistad y comprensión diaria. Ha sido el compañero perfecto en tan larga travesía. Juntos hemos celebrado las alegrías y hemos sufrido las decepciones propias del trabajo diario.

Una persona a la que admiro mucho como profesional y como persona es Natividad Llorca, a la que conocí en Novelda cuando comenzaba mi carrera docente y, por tanto, mi experiencia era nula. Ella me puso en contacto con el profesor Joan Josep Climent.

Quiero también agradecer a algunas personas la confianza y apoyo que me han transmitido, lo que ha resultado fundamental para mí, pues me han dado ánimos para superar los momentos difíciles. En primer lugar está mi familia, especialmente mi padre, con el que siempre puedo contar y mis suegros, a los que aprovecho para agradecer que siempre me hayan tratado como a un hijo más. Su apoyo y confianza en mí han sido muy importantes.

Mi labor profesional durante estos últimos años se ha desarrollado en el I.E.S. Haygón de Sant Vicent del Raspeig y la Universidad de Alicante. En la Universidad formo parte del grupo de Computación Paralela del Departamento de Ciencia de la Computación e Inteligencia Artificial. Agradezco a todos los miembros del mismo el clima de amistad y buen ambiente del cual he disfrutado estos últimos años.

En el Instituto Haygón he encontrado unos compañeros y amigos que han contribuido a que mi labor docente e investigadora se desarrollara en un ambiente óptimo. El director del mismo, Jorge Mateo, siempre me ha apoyado y facilitado la asistencia a las Jornadas

y Conferencias en las que he participado estos últimos años. Un buen amigo es Angel Luis García, profesor de Inglés y compañero desde hace bastantes años. Su ingenio nunca deja de sorprenderme. A él he acudido en diversas ocasiones y siempre me ha demostrado su amistad y generosidad. Desde el verano pasado desempeño la labor de secretario del Centro. Todos los compañeros me han ayudado mucho en esta labor completamente nueva para mí. Quiero destacar a una persona que sufre diariamente mis preocupaciones, mis despistes y mi desorganización. Es María Dolores Belda, a la que quiero agradecer su capacidad de trabajo y, por encima de todo, la amistad que me demuestra día a día.

Dos personas trabajan conmigo en temas de educación matemática y tecnología en la enseñanza de las matemáticas. Son Javier Santacruz y Rosario Martín. Juntos hemos compartido algún largo viaje y lo hemos pasado bien juntos. Su amistad es muy importante para mí.

Esta memoria ha sido subvencionada por el proyecto de investigación número PB98-0977 de la Dirección General de Enseñanza Superior e Investigación Científica.

Índice

Lista de tablas **viii**

Lista de figuras **xii**

Prólogo **xiii**

1 Planteamiento del problema **1**

1.1 Preliminares 1

1.2 Algoritmos del tipo divide y vencerás 7

1.3 Métodos de resolución de sistemas lineales tridiagonales 12

1.3.1 Métodos *divide y vencerás* para sistemas tridiagonales 13

1.3.2 Método del *recursive doubling* 15

1.3.3 Método de reducción cíclica 19

1.4 Métodos *divide y vencerás* para calcular la inversa de una matriz 27

1.5 Breve sumario de aplicaciones de los sitemas tridiagonales 28

2 El modelo BSP **31**

2.1 Breve introducción a la computación paralela 31

2.2 El modelo de computación BSP 35

2.3 El modelo de programación 39

2.4 El modelo de coste 40

2.5 Valores de s, g y l en distintas máquinas 42

3 Algoritmos *divide y vencerás* basados en la fórmula de Sherman-Morrison **47**

3.1 Introducción 47

3.2 Un algoritmo recursivo para sistemas tridiagonales 49

3.3 Un algoritmo BSP de tipo *fan-in* 65

3.3.1 Descripción del algoritmo 65

3.3.2 Coste computacional 73

3.4 Un algoritmo BSP basado en el método *recursive doubling* 78

3.4.1 Descripción del algoritmo 78

3.4.2 Coste computacional 82

3.5 Resultados teóricos y comparaciones 83

3.5.1 Tiempos en un IBM SP2 83

3.5.2 Tiempos en un CRAY T3D 88

3.5.3 Tiempos en un cluster de Pentiums 91

3.5.4 Estudio del *speedup* 93

4 Métodos divide y vencerás basados en la fórmula de Sherman-Morrison-Woodbury 101

4.1 Introducción 101

4.2 Un algoritmo general del tipo *divide y vencerás* 104

4.2.1 Descripción del método 104

4.2.2 Coste computacional 117

4.3 Un algoritmo BSP del tipo *divide y vencerás* general . . . 118

4.3.1 Descripción del método . . . 118

4.3.2 Coste computacional . . . 121

4.4 Algoritmos BSP *divide y vencerás* utilizando el método *recursive doubling* . . . 123

4.4.1 Descripción del método . . . 124

4.4.2 Coste computacional . . . 132

4.5 Algoritmo BSP *divide y vencerás* basado en el método de reducción cíclica . . . 138

4.5.1 Descripción del método . . . 138

4.5.2 Coste computacional . . . 143

4.6 Resultados teóricos y comparaciones . . . 146

4.6.1 Tiempos en un IBM SP2 . . . 147

4.6.2 Tiempos en un CRAY T3D . . . 148

4.6.3 Tiempos en un Cluster de Pentiums . . . 154

4.6.4 Estudio del speedup . . . 154

5 Método divide y vencerás de Bondeli 163

5.1 Introducción 163

5.2 Descripción del método 166

5.3 Algoritmos BSP divide y vencerás 176

5.3.1 Algoritmo BSP basado en el método de Bondeli 176

5.3.2 Coste computacional 180

5.3.3 Algoritmo BSP modificado basado en el método de Bondeli 182

5.3.4 Coste computacional 185

5.3.5 Algoritmo BSP utilizando el método *recursive doubling* 187

5.3.6 Coste computacional 191

5.4 Resultados teóricos y comparaciones 193

5.4.1 Tiempos en un IBM SP2 193

5.4.2 Tiempos en un CRAY T3D 196

5.4.3 Tiempos en un cluster de Pentiums 199

5.4.4 Estudio del speedup 202

6 Estudio comparativo de los algoritmos 209

6.1 Algoritmos *divide y vencerás* en el IBM SP2 209

6.2 Algoritmos *divide y vencerás* en el CRAY T3D 219

6.3 Algoritmos *divide y vencerás* en un cluster de Pentiums 226

A Conclusiones y líneas futuras de trabajo **231**

Bibliografía **235**

Índice de Tablas

2.1 Valores de los parámetros s, g, l y $n_{1/2}$. 44

3.1 Tiempos teóricos para los algoritmos 3.3, 3.4 y 3.5 medidos en un IBM SP2 dotado con un switch de alto rendimiento y conexión ethernet. 84

3.2 Diferencias de tiempos entre los algoritmos 3.4 y 3.5, medidos en porcentajes. 86

3.3 Tiempos teóricos para los algoritmos 3.3, 3.4 y 3.5 medidos en un CRAY T3D para 2, 4, 8, 16, 32, 64, 128 y 256 procesadores. 88

3.4 Número óptimo de procesadores y diferencias de tiempos para la ejecución de los algoritmos 3.4 y 3.5 en un CRAY T3D. 90

3.5 Tiempos teóricos y diferencias de tiempos para los algoritmos 3.3, 3.4 y 3.5 en un cluster de Pentiums. 92

3.6 Speedup y eficiencia de los algoritmos 3.3, 3.4 y 3.5 para $n = 2097152$. 94

4.1 Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un IBM SP2 con switch. 147

4.2 Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un IBM SP2 con ethernet. 149

4.3 Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un CRAY T3D desde $n = 2048$ hasta $n = 65536$. 150

4.4 Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un CRAY T3D desde $n = 131072$ hasta $n = 4194304$. 151

4.5 Número óptimo de procesadores en un CRY T3D para valores de n comprendidos entre $n = 2048$ y $n = 4194304$. 153

4.6 Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un cluster de Pentiums. 155

4.7 Speedup y eficiencia del algoritmo Alg. 4.2 para una matriz de tamaño $n = 2097152$. 157

5.1 Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un IBM SP2 dotado con un switch de alto rendimiento. . . 194

5.2 Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un IBM SP2 utilizando conexión ethernet. 195

5.3 Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un CRAY T3D utilizando 2, 4, 8 y 16 procesadores. 197

5.4 Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un CRAY T3D para 32, 64, 128 y 256 procesadores. 198

5.5 Número óptimo de procesadores en un CRAY T3D para tamaños de matrices comprendidos entre $n = 2048$ y $n = 4194304$.200

5.6 Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un Cluster de Pentiums para 2, 4 y 8 procesadores. 201

5.7 Speedup y eficiencia de los algoritmos 5.1, 5.2 y 5.3 para $n = 2097152$. 203

Índice de Figuras

2.1 Algoritmo *fan-in* para la suma de ocho números. 34

3.1 Inicialización y actualización del algoritmo 3.2 para el ejemplo 3.2. 62

3.2 Esquema del algoritmo 3.4 para $n = 32$ y $p = 4$. 70

3.3 Inicialización y actualizaciones que realiza el procesador P_0 cuando ejecuta el algoritmo 3.4 para el ejemplo 3.2. 72

3.4 Esquema del algoritmo 3.4 para el caso $n = 32$ y $p = 4$. 80

3.5 Valores del *speedup* en un IBM SP2 96

3.6 Valores del *speedup* en un CRAY y un cluster de Pentiums 97

3.7 Valores de la eficiencia en un IBM SP2 98

3.8 Valores del *speedup* en un CRAY y un cluster de Pentiums 99

4.1 Esquema de ejecución del algoritmo 4.2, para 4 procesadores. 120

4.2 Esquema de comunicaciones del método divide y vencerás basado en el método recursive doubling, para $p = 16$. 127

4.3 Esquema de ejecución del algoritmo 4.4, para 8 procesadores. 133

4.4 Esquema de ejecución del algoritmo (4.5), para 8 procesadores. 141

4.5 Valores del *speedup* en un IBM SP2 158

4.6 Valores del *speedup* en un CRAY T3D y un cluster de Pentiums 159

4.7 Valores de la eficiencia en un IBM SP2 160

4.8 Valores de la eficiencia en un CRAY T3D y un cluster de Pentiums 161

5.1 Esquema de ejecución del algoritmo 5.1, para 4 procesadores. 179

5.2 Esquema de ejecución del algoritmo 5.2, para 4 procesadores. 184

5.3 Valores del *speedup* en un IBM SP2 204

5.4 Valores del *speedup* en un CRAY T3D y un cluster de Pentiums 205

5.5 Valores de la eficiencia en un IBM SP2 206

5.6 Valores de la eficiencia en un CRAY T3D y un cluster de Pentiums 207

6.1 Tiempos en un IBM SP2 con switch para $n = 512$, $n = 1024$, $n = 2048$ y $n = 4096$ 211

6.2 Tiempos en un IBM SP2 con switch para $n = 8192$, $n = 16384$, $n = 32768$ y $n = 65536$ 212

6.3 Tiempos en un IBM SP2 con switch para $n = 8192$, $n = 16384$, $n = 32768$ y $n = 65536$ 213

6.4 Tiempos en un IBM SP2 con switch para $n = 2097152$ y $n = 4194304$ 214

6.5 Tiempos en un IBM SP2 con ethernet para $n = 512$, $n = 1024$, $n = 2048$ y $n = 4096$ 215

6.6 Tiempos en un IBM SP2 con ethernet para $n = 8192$, $n = 16384$, $n = 32768$ y $n = 65536$ 216

6.7 Tiempos en un IBM SP2 con ethernet para $n = 131072$, $n = 262144$, $n = 524288$ y $n = 1048576$ 217

6.8 Tiempos en un IBM SP2 con ethernet para $n = 2097152$ y $n = 4194304$ 218

6.9 Tiempos en un CRAY T3D para $n = 2048$ y $n = 4096$. 220

6.10 Tiempos en un CRAY T3D para $n = 8192$ y $n = 16384$. 221

6.11 Tiempos en un CRAY T3D para $n = 32768$ y $n = 65536$. 222

6.12 Tiempos en un CRAY T3D para $n = 131072$ y $n = 262144$. 223

6.13 Tiempos en un CRAY T3D para $n = 524288$ y $n = 1048576$. 224

6.14 Tiempos en un CRAY T3D para $n = 2097152$ y 4194304. 225

6.15 Tiempos en un cluster de Pentiums para $n = 512$, $n = 1024$, $n = 2048$ y $n = 4096$ 227

6.16 Tiempos en un cluster de Pentiums para $n = 8192$, $n = 16384$, $n = 32768$ y $n = 65536$ 228

6.17 Tiempos en un cluster de Pentiums para $n = 8192$, $n = 16384$, $n = 32768$ y $n = 65536$ 229

6.18 Tiempos en un cluster de Pentiums para $n = 2097152$ y $n = 4194304$ 230

Prólogo

En esta memoria se estudian diversos algoritmos BSP del tipo *divide y vencerás* para la resolución de sistemas de ecuaciones lineales de la forma $A\boldsymbol{x} = \boldsymbol{d}$, donde la matriz de coeficientes es tridiagonal e irreducible. Además, aprovechando las posibilidades de predicción del coste computacional que el modelo BSP incorpora, analizamos la complejidad computacional de los distintos algoritmos estudiados, realizando comparaciones entre los mismos para obtener el óptimo. Los tiempos teóricos se analizan en tres máquinas distintas: un IBM SP2 dotado con un switch de alto rendimiento y una conexión ethernet que nos permite trabajar como si de dos máquinas distintas se tratase, un CRAY T3D, en el que se dispone de hasta 256 procesadores y un cluster de Pentiums que consta de 8 procesadores.

Entendemos por **algoritmo** cualquier procedimiento computacional bien definido con un conjunto de datos de entrada permitidos que produce un valor o conjunto de valores como salida. Uno de los aspectos fundamentales que estudiamos en los algoritmos es su **eficiencia**, es decir, el análisis de la cantidad de recursos informáticos consumidos por el algoritmo, teniendo en cuenta que los recursos que habitualmente se contabilizan incluyen la cantidad de memoria y la cantidad de tiempo de cómputo requeridos por el algoritmo. Existen diversas técnicas algorítmicas con propiedades especiales para resolver tipos particulares de problemas. Tenemos algoritmos diseñados utilizando *programación dinámica*, *algoritmos voraces*, *algoritmos probabilísticos* y otros muchos tipos. En esta memoria nos centramos en los algoritmos del tipo *divide y vencerás* cuya idea básica consiste en que resuelven un problema descomponiéndolo en varios subproblemas que son similares al problema original, pero con un tamaño menor. Cada subproblema es resuelto de manera independiente para, a continuación, combinar los resultados parciales con el fin de obtener la solución del problema original.

Es importante resaltar la diferencia existente entre un *problema* y un *algoritmo* que resuelve un problema. Un problema tiene un sólo enunciado que lo describe en términos generales; sin embargo, en muchas ocasiones existen diversas formas de resolver un problema por lo que algunas soluciones pueden ser más eficaces que otras. Así, a menudo nos encontramos con diferentes algoritmos que resuelvan el mismo problema computacional, proporcionando cada uno de ellos un tiempo de ejecución distinto. En consecuencia, un problema fundamental que nos surge cuando debemos resolver un problema y disponemos de diversos algoritmos para su resolución es el de encontrar el algoritmo más rápido que proporciona la solución.

El modelo de computación paralela Bulk-Synchronous Parallel Computing, en lo sucesivo BSP, propuesto por Valiant [109] constituye un nuevo modelo de paralelismo en el que es sencillo crear programas que se ejecutan eficientemente en diferentes máquinas paralelas, con independencia de su arquitectura.

A lo largo de esta memoria cuando hacemos referencia a la sección 3.1, por ejemplo, nos estamos refiriendo a la sección primera del capítulo tercero. Así mismo, el algoritmo 3.2 hace referencia al segundo algoritmo del capítulo tercero, y así sucesivamente con el resto de referencias. Esta memoria se ha estructurado en seis capítulos que cubren los diferentes aspectos del problema que se aborda. En el capítulo 1 se presenta el problema en su contexto. Para ello, se introducen los conceptos básicos relativos a los sistemas tridiagonales y se introducen las características esenciales de los algoritmos del tipo *divide y vencerás* comentando brevemente algunos problemas de la computación donde aparecen dichos algoritmos. Se da una breve reseña histórica de la evolución de los métodos de resolución de sistemas tridiagonales, para analizar en detalle el método de reducción cíclica y el *recursive doubling*. Finalmente, se citan algunas aplicaciones donde aparecen sistemas de este tipo.

En el capítulo 2 se introducen los conceptos básicos relacionados con la computación paralela y se describen las características más importantes del modelo de computación paralela BSP. Se analizan diferentes aspectos de dicho modelo, como es el de la programación, basado en el concepto de *superpaso*. También se estudia detenidamente la capacidad de predecir el coste de los algoritmos. Sin duda alguna, la capacidad del modelo de predecir el rendimiento de los programas a partir de un reducido grupo de parámetros, constituye una de las características más importantes e innovadoras de este modelo de computación.

En el capítulo 3 se describen dos algoritmos BSP del tipo *divide y vencerás* basados en el método del desacoplamiento recursivo (véase Evans [44], Spaletta y Evans [101] y Mehrmann [90]), siguiendo una estrategia de actualización de rango uno y la división en bloques de la matriz de coeficientes de un sistema tridiagonal. La diferencia básica entre ambos algoritmos radica en el modelo de comunicación que se utiliza; en el primero se sigue un esquema de tipo *fan-in*, mientras que en el segundo todos los procesadores comunican datos al procesador principal, que finaliza el proceso.

En el capítulo 4 se obtiene un algoritmo general del tipo *divide y vencerás* basado en la fórmula de Sherman-Morrison-Woodbury (véase la sección 1.1). De este algoritmo se deducen cuatro algoritmos BSP cuyas diferencias básicas se encuentran en los modelos de comunicación que se utilizan y la forma en que se resuelve el sistema tridiagonal auxiliar a partir del cuál se obtiene la solución final. La resolución de dicho sistema auxiliar puede realizarse en forma secuencial o paralela, utilizando en este último caso dos algoritmos clásicos de resolución de sistemas tridiagonales, como son el método del desacoplamiento recursivo y el de la reducción cíclica.

En el capítulo 5 se describen tres algoritmos BSP del tipo *divide y vencerás* basados en el método propuesto por Bondeli [13] para la resolución de sistemas tridiagonales. Las diferencias entre estos algoritmos se basan en la forma en que se construye y resuelve el sistema tridiagonal auxiliar que nos permite calcular la solución final. En el último de los algoritmos estudiados se utiliza el método *recursive doubling* para resolver dicho sistema en paralelo.

Finalmente, en el capítulo 6 se realiza un estudio comparativo de los diferentes algoritmos estudiados a lo largo de los capítulos anteriores, obteniéndose conclusiones de carácter general sobre los algoritmos más rápidos y más lentos en cada una de las máquinas donde se realiza el estudio comparativo.

La memoria termina con un apéndice en el que se presentan las conclusiones y las líneas futuras de trabajo, así como la bibliografía utilizada para la elaboración de la misma.

Contrariamente a lo que suele ser habitual en la mayoría de las memorias, la orientación de esta memoria es apaisada. El motivo es la dificultad de escribir en modo vertical muchas de las expresiones y matrices que aparecen a lo largo de la misma.

Capítulo 1 Planteamiento del problema

1.1 Preliminares

La solución del sistema lineal

$$A\boldsymbol{x} = \boldsymbol{d}, \tag{1.1}$$

donde A es una matriz tridiagonal y $\boldsymbol{d}$ es el vector de términos independiente, dados por

$$A = \begin{bmatrix} a_1 & b_1 & & & \\ c_2 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & c_n & a_n \end{bmatrix} \quad \text{y} \quad \boldsymbol{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix}, \tag{1.2}$$

puede obtenerse utilizando el clásico método de eliminación de Gauss. Este método, que es puramente secuencial, consiste en anular los elementos que se encuentran por debajo de la diagonal principal utilizando las operaciones elementales adecuadas sobre la matriz y

el vector de términos independientes. Después obtenemos la última componente de la solución y finalmente, siguiendo un proceso de sustitución regresiva, calculamos el resto de las componentes de la solución.

El siguiente algoritmo recoge los pasos necesarios para resolver el sistema (1.1) mediante el método de eliminación de Gauss.

Algoritmo 1.1 Método de eliminación de Gauss para sistemas tridiagonales.

1 Para $i = 2, 3, \ldots, n$, calcular

1.1 $h = \dfrac{c_i}{a_i}$,

1.2 $a_i = a_i - hb_i$,

1.3 $d_i = d_i - hd_i$.

2 $x_n = \dfrac{d_n}{a_n}$.

3 Para $i = n-1, n-2, \ldots, 1$, calcular $x_i = \dfrac{d_i - x_{i+1}b_i}{a_i}$

Es fácil deducir del algoritmo 1.1 que el número de operaciones que se realizan es de $3(n-1)$ restas, $3(n-1)$ productos y $2n-1$ divisiones. Antes de hablar del coste computacional de este algoritmo debemos establecer la unidad que tomamos como base para medir el tiempo de ejecución del mismo. De aquí en adelante y a lo largo de esta memoria tomaremos como unidad de tiempo un *flop*, que representa el tiempo que supone la realización de una operación en coma flotante. Así, resolver un sistema tridiagonal utilizando el método de eliminación de Gauss por medio del algoritmo 1.1 supone un coste computacional de $8n-7$ flops.

Definición 1.1 Sea A la matriz tridiagonal de la expresión (1.2). Llamamos **diagonal dominanza** de A a la expresión

$$\delta = \min_{1 \le i \le n} \frac{a_i}{c_i + b_i},$$

donde, por comodidad, hemos supuesto que $c_1 = b_n = 0$. Diremos que la matriz A es **diagonal dominante** si $\delta \ge 1$ y **estrictamente diagonal dominante** si $\delta > 1$.

La eliminación de Gauss es un método que resulta estable para sistemas cuya matriz de coeficientes es diagonal dominante o estrictamente diagonal dominante. De aquí en adelante y a lo largo de esta memoria consideraremos sistemas tridiagonales cuya matriz de coeficientes es diagonal dominante con el fin de asegurarnos su estabilidad desde el punto de vista numérico.

En adelante, denotaremos por I_n a la matriz identidad de tamaño $n \times n$ y por $\boldsymbol{e}_k$, para $k = 1, 2, \ldots, n$, la columna k-ésima de I_n.

En los distintos métodos del tipo *divide y vencerás* que se analizan a lo largo de capítulos posteriores es necesario resolver varios sistemas tridiagonales con la misma matriz de coeficientes y distintos vectores de términos independientes. En estos casos es recomendable utilizar un método de resolución basado en la descomposición (también llamada factorización) LU de la matriz de coeficientes. Dicha descomposición para matrices tridiagonales consiste en escribir la matriz A como producto de dos matrices: L, bidiagonal inferior, y U, bidiagonal superior, de manera que

$$A = LU = \begin{bmatrix} 1 & & & & & \\ l_2 & 1 & & & & \\ & l_3 & 1 & & & \\ & & l_4 & 1 & & \\ & & & \ddots & \ddots & \\ & & & & l_n & 1 \end{bmatrix} \begin{bmatrix} u_1 & b_1 & & & & \\ & u_2 & b_2 & & & \\ & & u_3 & b_3 & & \\ & & & u_4 & \ddots & \\ & & & & \ddots & b_{n-1} \\ & & & & & u_n \end{bmatrix}.$$

El siguiente algoritmo calcula la descomposición LU para la matriz tridiagonal A de la expresión (1.2).

Algoritmo 1.2 Descomposición LU para la matriz tridiagonal A de la expresión (1.2).

1 $u_1 = a_1$.

2 Para $i = 2, 3, \ldots, n$, calcular

2.1 $l_i = \dfrac{c_i}{u_{i-1}}$,

2.2 $u_i = a_i - b_{i-1} l_i$.

De acuerdo con el algoritmo 1.2, el cálculo de la descomposición LU de una matriz tridiagonal de tamaño $n \times n$ requiere un total de $n-1$ restas, $n-1$ productos y $n-1$ divisiones, lo que representa un coste computacionel de $3n-3$ flops.

A partir de la descomposición LU de una matriz tridiagonal A podemos resolver el sistema (1.1) de forma sencilla. Para ello, sustituimos A por LU en (1.1), con lo que nos queda el sistema $LU\boldsymbol{x} = \boldsymbol{d}$. Ahora, llamando $U\boldsymbol{x} = \boldsymbol{y}$, resolvemos inicialmente el sistema $L\boldsymbol{y} = \boldsymbol{d}$ y utilizamos su solución $\boldsymbol{y}$ para obtener la solución general $\boldsymbol{x}$ resolviendo $U\boldsymbol{x} = \boldsymbol{y}$. El siguiente algoritmo resume las operaciones necesarias para resolver el sistema (1.1) una vez calculada la descomposición LU de la matriz A.

Algoritmo 1.3 Resolución del sistema (1.1) utilizando la descomposición LU de la matriz A de la expresión (1.2).

1 Calcular la factorización LU de A mediante el algoritmo 1.2.

2 $y_1 = d_1$.

3 Para $i = 2, 3, \ldots, n$, calcular $y_i = d_i - y_{i-1} l_i$.

4 $x_n = \dfrac{y_n}{u_n}$.

5 Para $i = n-1, n-2, \ldots, 1$, calcular $x_i = \dfrac{y_i - x_{i+1} b_i}{u_i}$.

El número de operaciones necesarias para resolver el sistema (1.1), de acuerdo con el algoritmo 1.3, es de $5n-4$, ya que se realizan $2n-2$ restas, $2n-2$ productos y n divisiones, por lo que el coste computacional será de $5n-4$ flops.

En el desarrollo de algunos métodos del tipo *divide y vencerás* para sistemas tridiagonales debemos resolver sistemas particulares donde el vector de términos independientes es la primera o última columna de la matriz I_n. Es decir, tendremos que resolver sistemas de la forma $A\boldsymbol{x} = \boldsymbol{e}_1$ y $A\boldsymbol{x} = \boldsymbol{e}_n$. Supongamos que la descomposición LU de la matriz A es conocida. Veamos las modificaciones que deben introducirse respecto de los algoritmos anteriores para resolver estos sistemas particulares.

Algoritmo 1.4 Resolución del sistema $A\boldsymbol{x} = \boldsymbol{e}_1$ utilizando la descomposición LU de A.

1 Calcular la factorización LU de A mediante el algoritmo 1.2.

2 $y_1 = 1$.

3 Para $i = 2, 3, \ldots, n$, calcular $y_i = -y_{i-1} l_i$.

4 $x_n = \dfrac{y_n}{u_n}$.

5 Para $i = n-1, n-2, \ldots, 1$, calcular $x_i = \dfrac{y_i - x_{i+1} b_i}{u_i}$.

La diferencia respecto al algoritmo 1.3 está en el proceso de sustitución progresiva en el que se calculan las componentes de la solución parcial $\boldsymbol{y}$ realizando únicamente $n-1$ operaciones. En consecuencia, el coste computacional de este algoritmo es de $4n-3$ flops.

Algoritmo 1.5 Resolución del sistema $A\boldsymbol{x} = \boldsymbol{e}_n$ utilizando la descomposición LU.

1 Calcular la factorización LU de A mediante el algoritmo 1.2.

2 $x_n = 1$.

3 Para $i = n-1, n-2, \ldots, 1$, calcular $x_i = -\dfrac{x_{i+1}b_i}{u_i}$

Nótese que al resolver un sistema de la forma $A\boldsymbol{x} = \boldsymbol{e}_n$, no es necesario efectuar ninguna operación para la resolución del sistema auxiliar $L\boldsymbol{y} = \boldsymbol{e}_n$, por lo que las únicas operaciones que se realizan son para el cálculo de la solución final mediante un proceso de sustitución regresiva. Así, el coste computacional del algoritmo 1.5 es de $2n-1$ flops.

Finalizamos esta sección con la introducción de las fórmulas de Sherman-Morrison y Sherman-Morrison-Woodbury que nos permiten resolver un sistema de ecuaciones lineales cualquiera. Supongamos que podemos escribir la matriz A de la forma $A = G + \boldsymbol{u}\boldsymbol{v}^T$ donde $\boldsymbol{u}, \boldsymbol{v}$ son vectores de $\mathbb{R}^n$. Supongamos que es fácil calcular G^{-1}. La fórmula de Sherman-Morrison (véase por ejemplo Golub y Van Loan [52, página 51]), nos proporciona un método para calcular la inversa de A como

$$A^{-1} = G^{-1} - \frac{1}{1+\boldsymbol{v}^T G^{-1}\boldsymbol{u}}\, G^{-1}\boldsymbol{u}\boldsymbol{v}^T G^{-1}. \tag{1.3}$$

Podemos resolver el sistema (1.1) utilizando la expresión (1.3) como

$$\boldsymbol{x} = A^{-1}\boldsymbol{d} = \left(G^{-1} - \frac{1}{1+\boldsymbol{v}^T G^{-1}\boldsymbol{u}} G^{-1}\boldsymbol{u}\boldsymbol{v}^T G^{-1}\right)\boldsymbol{d} = \left(I - \frac{1}{1+\boldsymbol{v}^T G^{-1}\boldsymbol{u}} G^{-1}\boldsymbol{u}\boldsymbol{v}^T\right) G^{-1}\boldsymbol{d}. \tag{1.4}$$

Si la matriz A puede escribirse de la forma $A = G + UV^T$ donde U y V son matrices de $\mathbb{R}^{n\times k}$ y conocemos de nuevo la inversa de G, entonces la fórmula de Sherman-Morrison-Woodbury nos prooporciona una expresión para el cálculo de la inversa de A, de la forma

$$A^{-1} = \left(A + UV^T\right)^{-1} = A^{-1} - A^{-1}U(I + V^TA^{-1}U)^{-1}V^TA^{-1}. \tag{1.5}$$

1.2 Algoritmos del tipo divide y vencerás

El conocido proverbio *divide y vencerás*, que en tantos ámbitos de la vida cotidiana se escucha, constituye en la actualidad una estrategia fundamental en la resolución de problemas. Debido al gran número de campos en los que se aplica podemos afirmar que es una estrategia básica en el diseño de ciertos algoritmos en computación. La idea básica en que se fundamenta esta estrategia es la de dividir un problema de gran tamaño que no podemos resolver de forma directa en diversos subproblemas más pequeñas que sí pueden resolverse. Posteriormente, se combinan las soluciones de estos subproblemas más pequeños para llegar a la solución del problema original. Podemos ver los textos de Aho, Hopcroft y Ullman [2, 3] para una descripción más detallada de los algoritmos de este tipo.

Así pues, el arquetipo *divide y vencerás* es un modelo de resolución de ciertos problemas que se basa en una estrategia muy simple. Esta estrategia la podemos resumir diciendo que cuando nos enfrentamos a un problema de gran tamaño, efectuamos los siguientes pasos:

(i) Tomamos el problema original y lo dividimos en problemas más pequeños del mismo tipo. Seguimos dividiendo estos problemas en problemas más pequeños hasta que llegamos a problemas de un tamaño tal que pueden resolverse de forma sencilla. Estos problemas los llamamos **problemas base**.

(ii) Resolvemos los problemas base.

(iii) Tomamos la solución de los problemas base y las combinamos para obtener la solución de problemas más grandes hasta que llegamos a la solución del problema original.

Como vemos, los algoritmos del tipo *divide y vencerás* son de tipo recursivo y presentan dos partes claramente diferenciadas. En la primera es donde se rompe el problema original en subproblemas más pequeños y se conoce con el nombre de ***split***; en la segunda, se unen los resultados parciales para obtener la solución general y se conoce con el nombre de ***merge***.

Al desarrollar un algoritmo del tipo *divide y vencerás*, debemos definir claramente los siguientes elementos:

Tamaño del problema. Debemos definir una función que nos proporcione el tamaño del problema original.

Problema base. Debemos identificar el tamaño del problema base puesto que es el problema más grande para el que tenemos una solución.

Solución del problema base. Debemos definir un procedimiento que tenga como entrada el problema base y como salida su solución, imponiendo la condición de que el tamaño de la entrada coincida con el tamaño del problema base.

Split. Es necesario establecer un procedimiento que divida el problema general en subproblemas. Así, si el problema original es $\mathcal{P}$, se debe crear un procedimiento cuya entrada sea $\mathcal{P}$ y cuya salida sean $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, un conjunto de problemas cuyo tamaño debe ser menor que el tamaño de $\mathcal{P}$.

Merge. Es necesario establecer un procedimiento que permita unir las soluciones de los subproblemas para obtener la solución del problema original. Así, es necesario definir una función cuya entrada sea $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$, las soluciones de los problemas $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, respectivamente y cuya salida sea $\mathcal{S}$, la solución del problema $\mathcal{P}$.

Esta idea básica, que en principio es puramente secuencial, puede fácilmente paralelizarse si los subproblemas, obtenidos por división del problema original, son independientes. De este modo, el algoritmo es fácilmente paralelizable únicamente para el caso en el que los subproblemas se resuelven de forma independiente unos de otros. Los pasos para desarrollar algoritmos paralelos del tipo *divide y vencerás* son esencialmente los mismos que para algoritmos secuenciales. Las esacasas diferencias se resumen en los siguientes puntos

- Para la resolución de los problemas base es muy probable que resulte mucho más eficiente una implementación secuencial que una implementación paralela. Esto es debido a que el tamaño de los problemas base suele ser muy pequeño y por tanto el tiempo del envío y recepción de datos entre los procesadores puede sobrepasar el tiempo de cálculo secuencial.

- Para poder resolver el problema en paralelo, los subproblemas que nos proporciona el procedimiento *split*, deben ser independientes entre sí.

También cabe reseñar que en los últimos años se han desarrollado nuevos lenguajes de programación diseñados específicamente para la paralelización automática de algoritmos del tipo *divide y vencerás*. Por ejemplo, podemos citar el proyecto de creación del lenguaje APERITIF (Automatic Parallelization of Divide and Conquer Algorithms), cuyo autor es Erlebach [43] de la Universidad Técnica de Munich, Alemania. El compilador de dicho lenguaje, llamado APRIL, traduce los programas de este lenguaje a programas en C, que pueden ejecutarse en computadores paralelos. Además, contiene ciertas llamadas a funciones específicas de paso de mensajes. El objetivo que persigue este proyecto es la creación de un lenguaje fácil de utilizar que nos permita una explotación eficiente de la potencia dispponible hoy en día a través de los computadores paralelos. El compilador y documentación relativa al mismo se puede obtener en [43]. Este programa ha sido desarrollado y probado en redes de workstations del tipo HP 9000/720 utilizando PVM.

De entre la enorme variedad de campos y problemas de la computación donde se aplica la estrategia *divide y vencerás* en paralelo comentamos brevemente algunos de ellos a continuación.

Problemas de ordenación (o clasificación). El problema de la ordenación o clasificación consiste en la reorganización de un conjunto de objetos en una secuencia específica. Este problema aparece como parte fundamental en muchos algoritmos. Los dos ejemplos más conocidos de ordenación son el método de ordenación por fusión ó mezcla, conocido como ***MergeSort*** y el método de ordenación rápido, conocido como ***QuickSort***.

El problema de la ordenación es fundamental en programación y consiste básicamente en la reorganización de una colección de elementos en orden ascendente o descendente. Más detalladamente, supongamos que nos dan una secuencia de n elementos

$a_1, a_2, \ldots, a_n$ y que cada elemento a_i tiene una clave asociada denotada por $[a_i]$. Supongamos también que existe una relación de orden total sobre las claves, es decir, que para cualesquiera tres valores de la clave $[a_m], [a_n], [a_p]$

(i) se cumple que $[a_m] < [a_n]$, $[a_m] = [a_n]$ o $[a_m] > [a_n]$ y

(ii) si $[a_m] < [a_n]$ y $[a_n] < [a_p]$ entonces $[a_m] < [a_p]$.

Bajo estas condiciones, el problema de ordenación consiste en reorganizar la colección de n elementos de tal forma que sus claves formen una secuencia no decreciente. Es decir, debe encontrarse una permutación σ del conjunto $\{1, 2, \ldots, n\}$ tal que $[a_{\sigma(1)}] \leq [a_{\sigma(2)}] \leq \cdots \leq [a_{\sigma(n)}]$. Mencionamos brevemente a continuación dos tipos de algoritmos de ordenación que utilizan algoritmos del tipo *divide y vencerás*.

El algoritmo de ordenación *MergeSort* fue uno de los primeros algoritmos de ordenación que se desarrollaron, siendo introducido por John von Newmann en 1945 (véase [4]). Es un ejemplo clásico de algoritmo del tipo *divide y vencerás*. Para ordenar un vector de n elementos divide éste en dos vectores de tamaño $n/2$ (fase divide del algoritmo). Estos dos vectores son ordenados de forma recursiva por el algoritmo de ordenación por mezcla (fase vencerás del algoritmo). Finalmente, los dos vectores ordenados son combinados para producir el vector final ordenado.

El algoritmo de ordenación *QuickSort*, propuesto por Hoare [62], sigue un esquema similar al de ordenación por mezcla: particiona una lista en dos listas y las reordena mezclando los resultados. Las diferencias radican en el proceso de división: ahora se elige un objeto x de la lista y se divide la lista en dos partes, la primera formada por los elementos anteriores a x y la segunda formada por los elementos posteriores. El proceso de ordenación posterior y mezcla no presenta diferencias con el caso anterior.

De entre la numerosa bibliografía que existe relacionada con estos algoritmos de ordenación, podemos citar entre otros Akl [4], Chandra, Jain, Basu y Kumar [20], Evans y Yousif [45] y Wheat y Evans [116].

Problemas de multiplicación de objetos. El problema de la multiplicación de objetos estructurados nos proporciona otra aplicación de estrategias del tipo *divide y vencerás* como podemos ver en Gonnet y Baeza-Yates [53, capítulo 6]. Supongamos que queremos multiplicar los enteros positivos n y m y supongamos que el entero positivo n puede escribirse, respecto de una base β como

$(n_l, \ldots, n_0)$, verificándose que

$$n = n_l\beta^l + n_{l-1}\beta^{l-1} + \cdots + n_1\beta + n_0, \qquad \text{con} \qquad 0 \leq n_0, \ldots, n_l < \beta.$$

Esta expresión nos sugiere que todo número entero puede verse como una tabla de dígitos. Al multiplicar n por m, las tablas que los representan pueden romperse en dos partes de longitud prácticamente igual. Así, podemos escribir que $n = a\beta^k + b$ y $m = c \cdot \beta^k + d$. Según Mignotte [93, página 34], el producto puede obtenerse mediante la expresión

$$nm = y\beta^{2k} + (x - y - z)\beta^k + z, \quad \text{con} \qquad x = (a+b)(c+d), \quad y = ac, \quad z = bd. \tag{1.6}$$

Los productos x, y, z se obtiene de forma recursiva utilizando la misma estrategia. Notemos que esta estrategia del tipo *divide y vencerás* nos proporciona un ahorro computacional, ya que sólo son necesarias tres multiplicaciones para calcular (1.6).

En cuanto al producto de dos matrices A y B, es conocido el algoritmo *divide y vencerás* de Strassen, (véase [74] para una descripción más detallada del algoritmo), que calcula el producto $C = AB$ utilizando una técnica del tipo *divide y vencerás* basada en la división de A y B en cuatro bloques, lo que permite que mediante siete productos sea posible calcular C, frente a los ocho productos necesarios mediante un algoritmo tradicional.

Geometría computacional. Diversos problemas de geometría computacional también utilizan estrategias de tipo *divide y vencerás*. Entre éstos cabe citar el problema del cálculo de la envoltura convexa de un conjunto de puntos (véase por ejemplo Edelsbrunner [41, página 145] y Preparata y Shamos [99, página 112]). En este caso, se construye una lista ordenada de puntos de acuerdo con el valor de su abcisa. Después, dicha lista se divide en dos nuevas listas de la misma longitud y se calcula recursivamente la envoltura convexa de ambas listas. Finalmente, las dos envolturas se mezclan para calcular la envoltura del conjunto inicial de puntos.

Preparata y Shamos [99] también utilizan la técnica *divide y vencerás* para una aplicación relacionada con el problema de la construcción de los diagramas de Voronoi.

Problemas de búsqueda binaria. Se trata de un problema de búsqueda en un vector ordenado S de claves. Para buscar una clave q en S, comparamos q con el elemento situado en la mitad del vector. Así, si el tamaño de S es n, comparamos q con el elemento

$S[n/2]$. Si q aparece antes de $S[n/2]$, entonces ya sabemos que q se encuentra en la mitad superior de nuestro conjunto; si no, debe estar en la mitad inferior del conjunto. Procedemos recursivamente de la misma forma hasta que finalmente obtenemos la clave, siendo el número de comparaciones que llevamos a cabo de orden $\log n$.

Problemas de procesamiento de imágenes. Aunque técnicas del tipo *divide y vencerás* no son utilizadas ampliamente en el procesamiento de imágenes, sí hay ciertos problemas donde este tipo de algoritmos han probado su eficiencia. Stout [105] examina diversas características de los algoritmos *divide y vencerás*, así como algunas de sus implicaciones para el diseño de máquinas y lenguajes que puedan soportar la programación y ejecución eficiente de algoritmos de este tipo. Otros problemas donde se aplican algoritmos de este tipo están relacionados con las técnicas de visualización de flujos para la representación de campos vectoriales. Una de éstas, conocida con el nombre de *Spot Noise*, utiliza texturas para la visualización de campos de flujo (véase van Wiijk [112]). Leeuw y Liere [35] proponen un algoritmo *Spot Noise* del tipo divide y vencerás y lo aplican a dos problemas concretos: un modelo de polución atmosférica y a una simulación numérica directa del flujo de una turbulencia.

1.3 Métodos de resolución de sistemas lineales tridiagonales

Existe una amplia literatura que aborda el problema general de la resolución de sistemas de ecuaciones lineales tridiagonales. Numerosos algoritmos se han propuesto a lo largo de las últimas décadas para resolver tales sistemas en paralelo, aprovechando el extraordinario auge y desarrollo de la computación paralela. Algunos de estos algoritmos han sido especialmente diseñados para ser ejecutados en máquinas con arquitecturas paralelas específicas, mientras que un buen número de ellos son de carácter general.

El primer método que resolvía sistemas tridiagonales fue el algoritmo de la reducción cíclica propuesto por Hockney [63] en 1965, precursor de una amplia familia de algoritmos basados en este método. Aunque el método original no era paralelo, posteriormente fue paralelizado para sistemas tridiagonales por bloques y para sistemas tridiagonales. La idea básica consiste en eliminar la mitad de las incógnitas, reagrupar las ecuaciones y eliminar, de nuevo, la mitad de las incógnitas. El proceso continúa de forma recursiva hasta que

podemos despejar una de las incógnitas que a su vez nos permite despejar el resto de incógnitas mediante un proceso de sustitución. Este algoritmo ha demostrado su potencia y eficiencia para la resolución de problemas matriciales en general y, en particular, para problemas donde aparecen matrices estructuradas. Algunos problemas concretos donde se ha aplicado este método con gran éxito es en la resolución de la ecuación de Poisson mediante una aproximación en diferencias finitas y para la resolución de ciertas recurrencias. El algoritmo ha sido ampliamente paralelizado y utilizado en una gran variedad de arquitecturas (véase Golub y Ortega [51]). Para una exposición y estudio exahustivo de este método y de algunas de sus aplicaciones, véase Gander y Golub [48] y Bondeli y Gander [14].

Muchos de los métodos propuestos para sistemas tridiagonales se puden agrupar en tres grandes familias: reducción cíclica, de la que ya se ha hablado anteriormente, *recursive doubling* y *divide y vencerás*. El método *recursive doubling* fue propuesto por Stone [103] y puede considerarse como el primer método diseñado explícitamente en paralelo para la resolución de sistemas tridiagonales. Es un método que resuelve recurrencias de primer orden y resulta estable únicamente para matrices estrictamente diagonal dominantes. Para un estudio más detallado de su estabilidad y del tipo de matrices que lo hacen estable, véase Dubois y Rodríguez [40] donde además se realizan comparaciones numéricas con el método de eliminación de Gauss. Tanto el método *recursive doubling* y de reducción cíclica se analizan detenidamente en la sección 1.3.3, ya que ambos se utilizan a lo largo de esta memoria.

Cabe mencionar el método propuesto por Larriba, Jorba y Navarro [79] para sistemas por bandas estrictamente diagonal dominantes y que se conoce con el nombre de *método de las particiones superpuestas*. Un estudio de la precisión de este nuevo método puede verse en Larriba, Jorba y Navarro [78].

1.3.1 Métodos *divide y vencerás* para sistemas tridiagonales

El primer método del tipo *divide y vencerás* para resolver sistemas tridiagonales data de 1978 y fue propuesto por Sameh y Kuck [100], quienes propusieron dos algoritmos paralelos para la solución de sistemas de este tipo. Estos algoritmos utilizaban rotaciones de Givens para realizar la eliminación de los elementos y requerían el uso de un número de procesadores con la única restricción de ser menor que el tamaño del sistema.

En 1984, Lawrie y Sameh [81] introdujeron una estrategia del tipo *divide y vencerás* para la resolución de sistemas tridiagonales por bloques, en el que utilizaban la eliminación de Gauss para la eliminación en los bloques. Además analizaron el comportamiento del algoritmo y su complejidad para máquinas paralelas con diferentes topologías en sus redes de conexión. Un año después, en 1985, Meier [91] obtuvo un algoritmo *divide y vencerás* que resolvía sistemas de ecuaciones por bandas. Esto supuso la primera generalización de esta estrategia para matrices no tridiagonales. Conroy [32] también obtuvo una generalización similar a la de Meier.

En 1987, Bar-On [6] introdujo una modificación importante en un algoritmo *divide y vencerás* para sistemas tridiagonales, consistente en la paralelización de la resolución del sistema tridiagonal reducido que se obtiene y cuya solución nos permite obtener las soluciones del sistema original. Además, demuestra que el algoritmo es comparable desde el punto de vista de su coste computacional con el método de reducción cíclica. Esta modificación es interesante desde el punto de vista de introducir un grado más de paralelización en los algoritmos basados en estrategias *divide y vencerás*. A lo largo de esta memoria veremos que algunas de las modificaciones que se realizan sobre algunos algoritmos básicos tienen como objetivo la paralelización del método en la resolución del sistema tridiagonal auxiliar que se construye.

Diversas comparaciones entre algoritmos del tipo *divide y vencerás* con otros métodos clásicos de resolución de sistemas tridiagonales, como el de reducción cíclica, fueron realizadas por Johnsson [68] y Johnsson y Ho [69], quienes llevaron a cabo las comparaciones en distintos tipos de computadores paralelos con diferentes topologías en sus redes de comunicación. Dongarra y Johnsson [39] propusieron un nuevo algoritmo *divide y vencerás* para sistemas por bandas, analizando su comportamiento y complejidad en diferentes arquitecturas.

En años posteriores diversos autores propusieron y analizaron algoritmos de este tipo para máquinas concretas, analizando el comportamiento de estos algoritmos cuando se utilizaban diversos modelos de comunicación. Entre estos trabajos cabe citar los desarrollados por Krechel, Plum y Stüben [75, 76], Cox y Knisely [33], Hoffmann y Potma [66] y Kumar [77].

En 1988 Van der Vorst [111] analizó dos versiones del algoritmo *divide y vencerás* para sistemas bidiagonales en máquinas paralelas y dos versiones en procesadores vectoriales. Un análisis del error en un algoritmo paralelo para resolver sistemas tridiagonales fue presentado por Van der Vorst [110] un año antes.

En 1991, Bondeli [13] propone un algoritmo *divide y vencerás* para sistemas tridiagonales válido tanto para máquinas paralelas como vectoriales. En dicho trabajo se realiza un estudio comparativo del coste computacional de este método frente a otros métodos clásicos para sistemas tridiagonales y analiza el caso de sistemas estrictamente diagonal dominantes. Este método, no válido para sistemas tridiagonales por bloques, es generalizado posteriormente para este tipo de sistemas por Mehrmann [90].

Sun, Zhang y Ni [106] proponen una variante del algoritmo *divide y vencerás* en el que el sistema tridiagonal auxiliar se resuelve utilizando el método *recursive doubling*. Además realizan comparaciones numéricas con otros métodos tradicionales.

En 1993, Larriba, Jorba y Navarro [79] establecen un nuevo criterio de parada para algoritmos *divide y vencerás* y muestran resultados en computadores vectoriales. Más recientemente, López [85] propone en su tesis doctoral una arquitectura unificada para algoritmos *divide y vencerás* en sistemas tridiagonales.

1.3.2 Método del *recursive doubling*

Si escribimos con detalle el sistema (1.1) tendremos

$$\left.\begin{array}{ccccccccccc}
a_1x_1 & + & b_1x_2 & & & & & & & = & d_1 \\
c_2x_1 & + & a_2x_2 & + & b_2x_3 & & & & & = & d_2 \\
 & & c_3x_2 & + & a_3x_3 & + & b_3x_4 & & & = & d_3 \\
 & & & \ddots & & \ddots & & & & \vdots & \\
 & & & & c_{n-1}x_{n-2} & + & a_{n-1}x_{n-1} & + & b_{n-1}x_n & = & d_{n-1} \\
 & & & & & & c_nx_{n-1} & + & a_nx_n & = & d_n
\end{array}\right\}, \tag{1.7}$$

que puede representarse mediante una relación de recurrencia de tres términos de la forma

$$c_i x_{i-1} + a_i x_i + b_i x_{i+1} = d_i, \qquad i = 1, 2, \dots, n, \tag{1.8}$$

con $c_1 = b_n = 0$. Por comodidad en la notación, supondremos también que $x_0 = x_{n+1} = 0$.

Despejando x_{i+1} de la ecuación (1.8) obtenemos, para $i = 1, 2, \dots, n-1$, que

$$x_{i+1} = \alpha_i x_i + \beta_i x_{i-1} + \gamma_i,$$

siendo

$$\alpha_i = -\frac{a_i}{b_i}, \qquad \beta_i = -\frac{c_i}{b_i}, \qquad \gamma_i = \frac{d_i}{b_i}. \tag{1.9}$$

Esta fórmula de recurrencia puede escribirse matricialmente como

$$\begin{bmatrix} x_{i+1} \\ x_i \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix}.$$

Si definimos, para $i = 1, 2, \dots, n-1$,

$$\boldsymbol{x}_{i-1} = \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix} \qquad \text{y} \qquad B_i = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{1.10}$$

entonces tenemos que

$$\boldsymbol{x}_i = B_i \boldsymbol{x}_{i-1} \tag{1.11}$$

y como

$$\boldsymbol{x}_0 = \begin{bmatrix} x_1 \\ x_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ 0 \\ 1 \end{bmatrix}, \tag{1.12}$$

el único elemento que necesitamos calcular para comenzar la recursión es x_1.

Ahora bien, de la ecuación (1.11) tenemos que

$$\boldsymbol{x}_1 = B_1\boldsymbol{x}_0, \quad \boldsymbol{x}_2 = B_2B_1\boldsymbol{x}_0, \quad \ldots, \quad \boldsymbol{x}_n = B_nB_n\cdots B_2B_1\boldsymbol{x}_0.$$

Si, para $i = 1, 2, \ldots, n$, escribimos $C_i = B_iB_{i-1}\cdots B_2B_1$, entonces

$$\boldsymbol{x}_i = C_i\boldsymbol{x}_0 \tag{1.13}$$

y en particular

$$\boldsymbol{x}_n = C_n\boldsymbol{x}_0. \tag{1.14}$$

Por la forma que tienen las matrices B_i es evidente que

$$C_n = \begin{bmatrix} l_{00} & l_{01} & l_{02} \\ l_{10} & l_{11} & l_{12} \\ 0 & 0 & 1 \end{bmatrix},$$

por tanto, como $x_{n+1} = x_0 = 0$, de la ecuación (1.14) tendremos que $0 = l_{00}x_1 + l_{02}$ por lo que

$$x_1 = -\frac{l_{02}}{l_{00}}. \tag{1.15}$$

En consecuencia, para empezar la recursión necesitamos calcular l_{00} y l_{02}. El siguiente ejemplo detalla las operaciones que se realizan para resolver un sistema tridiagonal utilizando este método.

Ejemplo 1.1 *Supongamos que queremos resolver el sistema (1.1) donde $n = 12$,*

$$A = \begin{bmatrix} 2 & 1 & & & & & & & & & & \\ 1 & 4 & 1 & & & & & & & & & \\ & -1 & 4 & -1 & & & & & & & & \\ & & -2 & 4 & 2 & & & & & & & \\ & & & -2 & 4 & 2 & & & & & & \\ & & & & 2 & -4 & 1 & & & & & \\ & & & & & 1 & 2 & 1 & & & & \\ & & & & & & -2 & 4 & 1 & & & \\ & & & & & & & -1 & 4 & -1 & & \\ & & & & & & & & 1 & -4 & 1 & \\ & & & & & & & & & -2 & 6 & 1 \\ & & & & & & & & & & 1 & 2 \end{bmatrix} \quad y \quad \boldsymbol{d} = \begin{bmatrix} 3 \\ 6 \\ 2 \\ 4 \\ 4 \\ -1 \\ 4 \\ 3 \\ 2 \\ -2 \\ 5 \\ 3 \end{bmatrix}.$$

Entonces, a partir de la expresión (1.9) obtenemos que

$$B_1 = \begin{bmatrix} -2 & -1 & 3 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_2 = \begin{bmatrix} -4 & -1 & 6 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_3 = \begin{bmatrix} 4 & -1 & -2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_4 = \begin{bmatrix} -2 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_5 = \begin{bmatrix} -2 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_6 = \begin{bmatrix} 4 & -2 & -1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$B_7 = \begin{bmatrix} -2 & -1 & 4 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_8 = \begin{bmatrix} -4 & 2 & 3 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_9 = \begin{bmatrix} 4 & -1 & -2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_{10} = \begin{bmatrix} 4 & -1 & -2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_{11} = \begin{bmatrix} -6 & 2 & 5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; B_{12} = \begin{bmatrix} -2 & -1 & 3 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

luego

$$C_1 = \begin{bmatrix} -2 & -1 & 3 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad C_2 = \begin{bmatrix} 7 & 4 & -6 \\ -2 & -1 & 3 \\ 0 & 0 & 1 \end{bmatrix}, \quad \ldots, \qquad C_{12} = \begin{bmatrix} 1106996 & 626732 & -1106996 \\ -609200 & -344902 & 609201 \\ 0 & 0 & 1 \end{bmatrix}$$

y por la expresión (1.15) se tiene que $x_1 = -\dfrac{l_{02}}{l_{00}} = 1$, *luego* $\boldsymbol{x}_0 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$. *Ahora, de la ecuación (1.13) calculamos* $\boldsymbol{x}_i$, *para* $i = 1, 2, \ldots, 12$, *obteniendo el vector solución, dado por*

$$\boldsymbol{x} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}^T.$$

1.3.3 Método de reducción cíclica

Ya vimos en la sección 1.3.2 que el sistema (1.1) puede ser descrito mediante la relación de recurrencia de tres términos dada por la expresión (1.8). La idea en que se basa el método de la reducción cíclica es tomar conjuntos de tres ecuaciones consecutivas superponiendo la última del conjunto anterior y aplicar operaciones elementales sobre la ecuación central con el fin de anular ciertos coeficientes. Tomando las ecuaciones centrales que han sido modificadas, podemos construir sistemas tridiagonales auxiliares hasta que al final del proceso queda una sola ecuación. Veamos este proceso con detalle.

Suponemos que el número de ecuaciones del sistema es $2^m - 1$, siendo m un entero positivo. Podemos escribir tres ecuaciones adyacentes de la expresión (1.8) como

$$\left.\begin{array}{rcrcrcrcrcl} c_{i-1}x_{i-2} & + & a_{i-1}x_{i-1} & + & b_{i-i}x_i & & & & & = & d_{i-1} \\ & & c_ix_{i-1} & + & a_ix_i & + & b_ix_{i+1} & & & = & d_i \\ & & & & c_{i+1}x_i & + & a_{i+1}x_{i+1} & + & b_{i+1}x_{i+2} & = & d_{i+1} \end{array}\right\} \tag{1.16}$$

para $i = 1, 3, 5, \ldots, 2^m - 3$; ahora modificamos la ecuación central realizando las siguientes operaciones elementales por filas

(i) restamos a la ecuación central la primera ecuación multiplicada por $\alpha_i = \dfrac{c_i}{a_{i-1}}$,

(ii) restamos a la nueva ecuación central la tercera ecuación multiplicada por $\gamma_i = \dfrac{b_i}{a_{i+1}}$,

con lo que la nueva ecuación central del sistema (1.16) se transforma en

$$c_i^{(1)}x_{i-2} + a_i^{(1)}x_i + b_i^{(1)}x_{i+2} = d_i^{(1)}, \tag{1.17}$$

donde

$$c_i^{(1)} = -\alpha_i c_i, \quad b_i^{(1)} = -\gamma_i b_{i+1}, \quad a_i^{(1)} = a_i - \alpha_i b_{i-1} - \gamma_i c_{i+1}, \quad d_i^{(1)} = d_i + \alpha_i d_{i-1} + \gamma_i d_{i+1}. \tag{1.18}$$

Los superíndices que aparecen en las expresiones (1.17) y (1.18) indican que nos encontramos en la primera fase de la reducción cíclica. Las ecuaciones (1.17) establecen nuevas relaciones entre las variables impares, ya que eliminamos los coeficientes de las variables

pares en la ecuación central. Además, constituyen un sistema tridiagonal, dado por

$$\left.\begin{array}{ccccccccccc}
a_1^{(1)}x_1 & + & b_1^{(1)}x_3 & & & & & & & = & d_1^{(1)} \\
c_3^{(1)}x_1 & + & a_3^{(1)}x_3 & + & b_3^{(1)}x_5 & & & & & = & d_3^{(1)} \\
 & & c_5^{(1)}x_3 & + & a_5^{(1)}x_5 & + & b_5^{(1)}x_7 & & & = & d_5^{(1)} \\
 & & & \ddots & & \ddots & & \ddots & & \vdots & \\
 & & & & c_{n-3}^{(1)}x_{n-5} & + & a_{n-3}^{(1)}x_{n-3} & + & b_{n-3}^{(1)}x_{n-3} & = & d_{n-3}^{(1)} \\
 & & & & & & c_{n-1}^{(1)}x_{n-3} & + & a_{n-1}^{(1)}x_{n-1} & = & d_{n-1}^{(1)}
\end{array}\right\}.$$

En este nuevo sistema tridiagonal, el número de ecuaciones se ha reducido a la mitad. Claramente, este proceso descrito hasta aquí puede repetirse sucesivamente hasta que, después de $s = \log(2^m) - 1$ niveles de reducción, sólamente tenemos una ecuación central, para $i = 2^{m-1}$. Esta ecuación es $a_{2^{m-1}}^{(s)} x_{2^{m-1}} = d_{2^{m-1}}^{(s)}$, de donde podemos despejar la variable central como

$$x_{2^{m-1}} = \frac{d_{2^{m-1}}^{(s)}}{a_{2^{m-1}}^{(s)}}. \tag{1.19}$$

Las variables que quedan por determinar pueden ser calculadas siguiendo un proceso conocido como *fan-in*. Como conocemos la variable $x_{2^{u-1}}$, podemos determinar las variables intermedias x_i, para $i = \dfrac{2^m}{4}, \dfrac{3 \cdot 2^m}{4}$, mediante la expresión

$$x_i = \frac{d_i^{(s-1)} - c_i^{(s-1)} x_{i-\frac{2^m}{4}} - b_i^{(s-1)} x_{i+\frac{2^m}{4}}}{a_i^{(s-1)}}. \tag{1.20}$$

Este proceso puede aplicarse de forma repetida hasta que se obtienen todas las componentes de la solución. Veamos un ejemplo que resuma las características de este método.

Ejemplo 1.2 *Supongamos que queremos resolver el sistema (1.1) donde* $n = 15$,

$$A = \begin{bmatrix} 2 & 3 & & & & & & & & & & & & & \\ 4 & -1 & 2 & & & & & & & & & & & & \\ & 1 & -1 & 3 & & & & & & & & & & & \\ & & 4 & 5 & 1 & & & & & & & & & & \\ & & & 2 & -1 & 3 & & & & & & & & & \\ & & & & 1 & 4 & -1 & & & & & & & & \\ & & & & & 2 & 1 & 2 & & & & & & & \\ & & & & & & -3 & 5 & 2 & & & & & & \\ & & & & & & & 4 & -1 & 6 & & & & & \\ & & & & & & & & -1 & 5 & 2 & & & & \\ & & & & & & & & & 7 & 2 & 3 & & & \\ & & & & & & & & & & 1 & -5 & 1 & & \\ & & & & & & & & & & & 4 & 1 & 1 & \\ & & & & & & & & & & & & 3 & 1 & 4 \\ & & & & & & & & & & & & & 3 & 1 \end{bmatrix} \quad y \quad \boldsymbol{d} = \begin{bmatrix} 5 \\ 5 \\ 3 \\ 10 \\ 4 \\ 4 \\ 5 \\ 4 \\ 9 \\ 6 \\ 12 \\ 3 \\ 6 \\ 8 \\ 4 \end{bmatrix}.$$

En la primera fase del proceso de reducción cíclica tomamos las siguientes ternas de ecuaciones

$$\left.\begin{array}{rcrcrcrcr} 2x_1 & + & 3x_2 & & & & & = & 5 \\ 4x_1 & - & x_2 & + & 2x_3 & & & = & 5 \\ & & x_2 & - & x_3 & + & 3x_4 & = & 3 \end{array}\right\}, \qquad \left.\begin{array}{rcrcrcrcrcrcr} x_2 & - & x_3 & + & 3x_4 & & & & & = & 3 \\ & & 4x_3 & + & 5x_4 & + & x_5 & & & = & 10 \\ & & & & 2x_4 & - & x_5 & + & 3x_6 & = & 4 \end{array}\right\},$$

$$\left.\begin{array}{rcrcrcrcrcr} 2x_4 & - & x_5 & + & 3x_6 & & & & & = & 4 \\ & & x_5 & + & 4x_6 & - & x_7 & & & = & 4 \\ & & & & 2x_6 & + & x_7 & + & 2x_8 & = & 5 \end{array}\right\}, \qquad \left.\begin{array}{rcrcrcrcrcr} 2x_6 & + & x_7 & + & 2x_8 & & & & & = & 5 \\ & - & 3x_7 & + & 5x_8 & + & 2x_9 & & & = & 4 \\ & & & & 4x_8 & - & x_9 & + & 6x_{10} & = & 9 \end{array}\right\},$$

$$\left.\begin{array}{rcrcrcrcrcr} 4x_8 & - & x_9 & + & 6x_{10} & & & & & = & 9 \\ & - & x_9 & + & 5x_{10} & + & 2x_{11} & & & = & 6 \\ & & & & 7x_{10} & + & 2x_{11} & + & 3x_{12} & = & 12 \end{array}\right\}, \qquad \left.\begin{array}{rcrcrcrcrcr} 7x_{10} & + & 2x_{11} & + & 3x_{12} & & & & & = & 12 \\ & & x_{11} & - & 5x_{12} & + & x_{13} & & & = & -3 \\ & & & & 4x_{12} & + & x_{13} & + & x_{14} & = & 6 \end{array}\right\},$$

$$\left.\begin{array}{rcrcrcrcr} 4x_{12} & + & x_{13} & + & x_{14} & & & = & 6 \\ & & 3x_{13} & + & x_{14} & + & 4x_{15} & = & 8 \\ & & & & 3x_{14} & + & x_{15} & = & 6 \end{array}\right\}.$$

Efectuando sobre las ecuaciones centrales de estas ternas las operaciones elementales descritas por las expresiones (1.17) y (1.18) obtenemos el sistema

$$\left.\begin{array}{rcrcrcrcrcrcrcr} -5x_2 & + & 6x_4 & & & & & & & & & & & = & 1 \\ 4x_2 & + & 19x_4 & + & 3x_6 & & & & & & & & & = & 26 \\ & & 2x_4 & + & 9x_6 & + & 2x_8 & & & & & & & = & 13 \\ & & & & 6x_6 & + & 19x_8 & + & 12x_{10} & & & & & = & 37 \\ & & & & & & -4x_8 & - & 8x_{10} & - & 3x_{12} & & & = & -15 \\ & & & & & & & & -\frac{7}{2}x_{10} & - & \frac{21}{2}x_{12} & - & x_{14} & = & -15 \\ & & & & & & & & & & -12x_{12} & - & 14x_{14} & = & -26 \end{array}\right\}.$$

Agrupamos de nuevo en ternas de ecuaciones consecutivas, con lo que obtenemos

$$\left.\begin{array}{rcrcrcrcr} -5x_2 & + & 6x_4 & & & & & = & 1 \\ 4x_2 & + & 19x_4 & + & 3x_6 & & & = & 26 \\ & & 2x_4 & + & 9x_6 & + & 2x_8 & = & 13 \end{array}\right\}, \quad \left.\begin{array}{rcrcrcrcrcrcr} 2x_4 & + & 9x_6 & + & 2x_8 & & & & & = & 13 \\ & & 6x_6 & + & 19x_8 & + & 12x_{10} & & & = & 37 \\ & & & - & 4x_8 & - & 8x_{10} & - & 3x_{12} & = & -15 \end{array}\right\}, \tag{1.21}$$

$$\left.\begin{array}{rcrcrcr} -4x_8 & - & 8x_{10} & - & 3x_{12} & & = & -15 \\ & - & \frac{7}{2}x_{10} & - & \frac{21}{2}x_{12} & - & x_{14} = & -15 \\ & & & - & 12x_{12} & - & 14x_{14} = & -26 \end{array}\right\}. \tag{1.22}$$

Volvemos a efectuar las operaciones elementales habituales sobre las ecuaciones centrales, para llegar al sistema tridiagonal de tres ecuaciones

$$\left.\begin{array}{rcrcrcr} \frac{347}{15}x_4 & - & \frac{2}{3}x_8 & & & = & \frac{337}{15} \\ \frac{-4}{3}x_4 & + & \frac{35}{3}x_8 & - & \frac{9}{2}x_{12} & = & \frac{35}{6} \\ & & \frac{7}{4}x_8 & - & \frac{933}{112}x_{12} & = & -\frac{737}{112} \end{array}\right\}. \tag{1.23}$$

Después de este paso únicamente podemos formar una terna con estas tres ecuaciones, que es exactamente el conjunto de ecuaciones (1.23). Repitiendo de nuevo las operaciones elementales que se han realizado en etapas anteriores, reducimos la ecuación central a una única ecuación lineal cuya incógnita es x_8*, de la forma* $\frac{1152867}{107917}x_8 = \frac{1152867}{107917}$*, lo que significa que* $x_8 = 1$*. Una vez calculada la componente central de la solución, efectuamos un proceso de sustitución regresiva conducente a la obtención del resto de componentes de la solución. Así, a partir de la componente* x_8*, podemos obtener las componentes* x_4 *y* x_{12} *por simple sustitución en la terna de ecuaciones (1.23), obteniéndose que* $x_4 = x_{12} = 1$*. Ahora, a partir de las variables* x_4*,* x_8 *y* x_{12} *y sustituyendo de forma adecuada en (1.21) y (1.22) se obtiene que* $x_2 = x_6 = x_{10} = x_{14} = 1$*.*

Siguiendo un proceso de sustitución regresiva similar, se obtiene el resto de las componentes del vector solución, resultando que $x_1 = x_2 = \cdots = x_{14} = x_{15} = 1$, *por lo que la solución es el vector cuyas componentes son todas iguales a uno.*

En algunos casos tendremos que resolver un sistema tridiagonal de la forma $B\boldsymbol{z} = \boldsymbol{h}$ mediante el método de reducción cíclica donde

$$B = \begin{bmatrix} a_0 & 1 & & & & & & \\ 1 & a_1 & b_1 & & & & & \\ & c_2 & a_2 & 1 & & & & \\ & & 1 & a_3 & b_3 & & & \\ & & & c_4 & a_4 & 1 & & \\ & & & & \ddots & \ddots & \ddots & \\ & & & & & c_{2p-4} & a_{2p-4} & 1 \\ & & & & & & 1 & a_{2p-3} \end{bmatrix}, \qquad \boldsymbol{z} = \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ \vdots \\ z_{2p-4} \\ z_{2p-3} \end{bmatrix} \quad \text{y} \quad \boldsymbol{h} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ \vdots \\ h_{2p-4} \\ h_{2p-3} \end{bmatrix}. \tag{1.24}$$

Siguiendo un proceso similar al descrito en esta sección para la resolución del sistema (1.1), inicialmente tomamos tres ecuaciones adyacentes

$$\left.\begin{array}{rcrcrcrcrcl} c_{i-1}z_{i-2} & + & a_{i-1}z_{i-1} & + & z_i & & & & & = & h_{i-1} \\ & & z_{i-1} & + & a_iz_i & + & b_iz_{i+1} & & & = & h_i \\ & & & & c_{i+1}z_i & + & a_{i+1}z_{i+1} & + & z_{i+2} & = & h_{i+1} \end{array}\right\}. \tag{1.25}$$

El primer paso de la reducción cíclica consiste en multiplicar la primera ecuación por $\alpha_i = \dfrac{1}{a_{i-1}}$ y restarla de la segunda ecuación, con lo que la ecuación central se convierte en

$$-\frac{c_{i-1}}{a_{i-1}}z_{i-2} + \left(a_i - \frac{1}{a_{i-1}}\right)z_i + b_iz_{i+1} = h_i - \frac{h_{i-1}}{a_{i-1}}. \tag{1.26}$$

El segundo paso consiste en multiplicar la tercera ecuación por $\gamma_i = \dfrac{b_i}{a_{i+1}}$ y restarla de la segunda ecuación, con lo que la ecuación central queda modificada como

$$-\frac{c_{i-1}}{a_{i-1}} z_{i-2} + \left(a_i - \frac{1}{a_{i-1}} - \frac{b_i c_{i+1}}{a_{i+1}} \right) z_i - \left(\frac{b_i}{a_{i+1}} \right) z_{i+2} = h_i - \frac{h_{i-1}}{a_{i-1}} - \frac{b_i h_{i+1}}{a_{i+1}},$$

que puede escribirse como

$$c_i^{(1)} z_{i-2} + a_i^{(1)} z_i + b_i^{(1)} z_{i+2} = h_i^{(1)}, \tag{1.27}$$

donde

$$\left.\begin{aligned} c_i^{(1)} &= -\frac{c_{i-1}}{a_{i-1}} \\ a_i^{(1)} &= a_i - \frac{1}{a_{i-1}} - \frac{b_i c_{i+1}}{a_{i+1}} \\ b_i^{(1)} &= -\frac{b_i}{a_{i+1}} \\ h_i^{(1)} &= h_i - \frac{h_{i-1}}{a_{i-1}} - \frac{b_i h_{i+1}}{a_{i+1}} \end{aligned}\right\}. \tag{1.28}$$

Se utiliza la notación $a_i^{(1)}$, $b_i^{(1)}$ y $c_i^{(1)}$ para denotar los coeficientes que acompañan a las variables de la ecuación i-ésima en el primer nivel de reducción de variables. Análogamente, los coeficientes del segundo nivel de reducción para la ecuación i-ésima se denotarán por $a_i^{(2)}$, $b_i^{(2)}$ y $c_i^{(2)}$ y así sucesivamente. El resto del proceso es similar al descrito anteriormente.

1.4 Métodos *divide y vencerás* para calcular la inversa de una matriz

El problema del cálculo de la inversa de una matriz A es mucho más costoso, desde el punto de vista computacional, que el de resolver el sistema (1.1), por lo que la mayoría de problemas que incluyen el cálculo de inversas pueden ser reformulados en términos de la resolución de sistema lineales. Por ello, podemos afirmar en términos generales que el cálculo explícito de la inversa de una matriz debe evitarse siempre que sea posible. Sin embargo, como veremos más detalladamente a continuación, en algunas ocasiones su cálculo es necesario.

En ingeniería estructural, el cálculo de la inversa de una matriz es un problema que aparece con cierta frecuencia. La razón de esto es que en el análisis de una estructura en equilibrio aparecen sistemas con un número muy alto de ecuaciones e incógnitas. A menudo, dichas ecuaciones son lineales aun incluso después de considerar deformaciones sobre el material. Por ejemplo, consideramos el problema de una viga elástica horizontal sujeta en cada extremo y sometida a fuerzas en n puntos de la misma $1, 2, \ldots, n$. Si $\boldsymbol{x} \in \mathbb{R}^n$ es la relación de fuerzas en estos puntos y $\boldsymbol{d} \in \mathbb{R}^n$ representa la desviación de la viga en estos n puntos, entonces por la ley de Hook, tenemos un sistema lineal donde la matriz de coeficientes recibe el nombre de matriz de flexibilidad. Su inversa recibe el nombre de matriz **stiffness**. A menudo, es necesario calcular la inversa de esta matriz debido al significado físico de sus columnas. Así, la primera columna de la matriz de *stiffness* representa las fuerzas que deben aplicarse en los n puntos con el fin de producir una desviación particular en el punto 1 y una desviación nula en el resto de puntos. El mismo razonamiento es válido para el resto de las columnas.

Otro problema donde es necesario el cálculo explícito de una matriz inversa es el llamado **spring-mass problem** en física, en el que algunas masas se encuentran suspendidas verticalmente por una serie de muelles y debemos considerar las constantes de los muelles y los desplazamientos de cada muelle desde su posición de equilibrio. Cuando aplicamos la segunda ley de Newton a este problema obtenemos un sistema cuya matriz de coeficientes es tridiagonal y diagonal dominante. La inversa de esta matriz es la matriz de *stiffness*, que está directamente relacionada con el desplazamiento de las masas cuando algunas fuerzas externas se imponen sobre ellas. Así, el elemento (i, j) de esta matriz inversa nos proporciona el desplazamiento de la masa i cuando sobre la masa j se impone una fuerza externa de una unidad de fuerza.

También podemos encontrar ciertas aplicaciones fuera del campo de la ingeniería o la física donde se deben calcular matrices inversas. Entre éstas encontramos aplicaciones en el campo de la teoría de sistemas dinámicos, relacionadas con la determinación de los puntos fijos de un sistema dinámico mediante el método de Newton (véase, por ejemplo, Stoer y Burlirsch [102]). Para una descripción más detallada de ciertas aplicaciones de las inversas, véase Keener y Bogar [73] y Meurant [92], quien describe ciertas aplicaciones relacionadas con precondicionadores y proporciona un gran número de referencias relacionadas con matrices inversas.

Diversos autores han estudiado el problema del cálculo de la inversa de una matriz A tridiagonal, irreducible y diagonal dominante, proponiendo algoritmos eficientes para su cálculo. Podemos citar a Lewis [84], Meurant [92], Usnami [108] y Huang y McColl [67]. Spaletta y Evans [101] y Mehrmann [90] proponen el método *recursive decoupling* para el cálculo de la inversa de una matriz tridiagonal.

Climent, Tortosa y Zamora [31] proponen un algoritmo BSP *divide y vencerás* para calcular la inversa de una matriz tridiagonal, irreducible y diagonal dominante, que constituye una aplicación de la fórmula de Sherman-Morrison para matrices con estas características. En dicho artículo se estudian dos algoritmos BSP siguiendo diferentes modelos de comunicación entre los procesadores, realizándose un estudio numérico teórico y experimental de los tiempos que se obtienen para los dos algoritmos en una máquina IBM SP2. Finalmente, se realiza un estudio comparativo computacional entre estos algoritmos y un algoritmo tradicional del tipo *divide y vencerás* basado en el método de Bondeli para la resolución de sistemas tridiagonales, adaptado al cálculo de la inversa de una matriz tridiagonal. Los resultados teóricos y experimentales demuestran que los algoritmos del tipo *divide y vencerás* que calculan directamente la matriz inversa son más rápidos que el algoritmo clásico de resolución de sistemas tridiagonales modificado para calcular la inversa.

1.5 Breve sumario de aplicaciones de los sitemas tridiagonales

De entre las numerosas aplicaciones y problemas en el ámbito de la computación donde aparecen sistemas tridiagonales citamos muy brevemente algunos de los más notables, en los que la resolución de dichos sistemas constituye una parte esencial del proceso conducente a la obtención de la solución de dichos problemas.

- **Ecuaciones diferenciales parciales.** Diversos métodos iterativos para la solución de ecuaciones diferenciales parciales necesitan resolver un conjunto de sistemas tridiagonales múltiples como resultado de un proceso de discretización por mallas. Entre estos métodos destaca el *Alternating Direction Implicit* (ADI), estudiado por Johnsson, Saad y Schultz [70] para computadores paralelos y por Ortega y Voigt [97] entre otros.

- **Ajuste de curvas por esplines cúbicos.** El ajuste de curvas se utiliza en muchas aplicaciones. Por ejemplo, en Diseño Asistido por Ordenador (CAD) se utiliza el ajuste de curvas para definir la forma de las superficies que forman los objetos diseñados. También se utiliza en la modelización de datos obtenidos de diferentes procesos. El proceso de ajuste de una curva se desarrolla a partir de un conjunto de puntos dados. Entre las diferentes técnicas que se utilizan para obtener un modelo de curva que se ajuste a ese conjunto de puntos destacamos los esplines cúbicos naturales y la interpolación por B-esplines. En el proceso de cálculo que requiere la aplicación de estas técnicas, es necesario resolver sistemas tridiagonales cuya matriz de coeficientes es Toeplitz y diagonal dominante, siendo su diagonal dominanza 2. Véase Chung y Shen [22] para una descripción más detallada de esta aplicación.

- **Discretización de ecuaciones diferenciales.** El comportamiento eléctrico de las neuronas puede modelizarse por medio de ecuaciones diferenciales parciales que evolucionan con el tiempo, como puede verse en Hines [61]. La discretización por diferencias finitas de estas ecuaciones dan lugar a sistemas tridiagonales que deben resolverse para comprender la evolución de las neuronas con el tiempo. La solución de estos sistemas puede desarrollarse tanto por métodos directos como iterativos. Los sistemas tridiagonales que deben resolverse mediante esta técnica son diagonal dominantes ya que las ecuaciones diferenciales parciales son de tipo parabólico, variando su tamaño de forma considerable con el dominio de la discretización. Si el número de dominios es muy grande, el paralelismo se obtiene de una forma natural asignando un conjunto de sistemas trididagonales a cada procesador, intentando que la carga computacional se encuentre equilibrada.

Capítulo 2 El modelo BSP

2.1 Breve introducción a la computación paralela

En los años 70 comenzaron a aparecer algunos computadores con instrucciones de *hardware* para operar sobre vectores y algunos computadores con diversos procesadores que operaban en paralelo. Los primeros recibieron el nombre de **computadores vectoriales** y a los segundos se los llamó **computadores paralelos**.

La idea en la que se basa un computador paralelo es que un determinado número de procesadores trabajan en equipo para desarrollar una única tarea. La motivación de esta idea es que si un único procesador necesita un tiempo de t segundos para llevar a cabo una cierta tarea, entonces p procesadores trabajando en conjunto para realizar la misma tarea deberían de tardar un tiempo de t/p segundos. Esta constituye una situación óptima que en muy pocas ocasiones puede alcanzarse. Sin embargo, nuestro objetivo es proponer algoritmos que resulten cada vez más rápidos, aprovechando las características de estos ordenadores que disponen de diversos procesadores para ejecutar las tareas computacionales que cada proceso conlleva.

En los años ochenta se construyeron un buen número de ordenadores paralelos en los que se obtenía un rendimiento aceptable de aquellas aplicaciones que tenían en cuenta la arquitectura específica de la máquina en la que se ejecutaba la aplicación. Esto resultaba lento

y costoso debido a la gran diversidad de arquitecturas que surgían. Desde principios de los noventa hasta ahora se advierte una evolución más estándar en los modelos arquitectónicos que se desarrollan, consistentes básicamente en un conjunto de pares (procesador,memoria) inteconectados mediante una red de comunicación que permite el empleo de direcciones globales. Esta convergencia se basa en razones de tipo tecnológico y económico; los grandes fabricantes aprovechan los microprocesadores existentes para construir máquinas paralelas en lugar de crear nuevos microprocesadores y, por otro lado, la memoria distribuida presenta la ventaja de la rapidez de acceso a la misma.

En las máquinas paralelas nos encontramos con algunas dicotomías importantes que analizamos muy brevemente. La primera de ellas está relacionada con la forma en la que se controla el trabajo de los procesadores. En un sistema de *simple instrucción, múltiples datos* (SIMD), todos los procesadores se encuentran bajo el control del procesador principal o maestro, ejecutando todos los procesadores la misma instrucción al mismo tiempo. El primer gran sistema paralelo desasrrollado a principios de los 70, el Illiac IV era una máquina de este tipo. La alternativa a estos sistemas la constituye los sistemas de *múltiple instrucción, múltiples datos* (MIMD), en el que los procesadores individuales trabajan bajo el control de sus propios programas. Aquí se presenta el problema de la sincronización, ya que no hay un procesador principal que pueda llevar a cabo esa tarea.

Otra importante dicotomía en los computadores paralelos es la de la *memoria compartida* frente a la *memoria local o distribuida.* En los computadores con memoria compartida todos los procesadores tienen acceso a una memoria común, lo cual presenta la ventaja de que todas las comunicaciones entre los procesadores se realiza a través de la memoria común, por lo que las comunicaciones en este sistema resultan muy rápidas. La desventaja es que muchos procesadores al mismo tiempo pueden acceder a esa memoria compartida por todos, lo que puede provocar un retraso hasta que la memoria está libre. La alternativa a estos sistemas son los sistemas con memoria local, en los que cada procesador accede a su propia memoria. Las comunicaciones entre los procesadores tienen lugar a través de un proceso de paso de mensajes, en el que los datos se transfieren de un procesador a otro.

Otro aspecto fundamental de los computadores paralelos es el modo en el que los diferentes procesadores se comunican unos con otros. Existen distintos esquemas de interconexión de los procesadores. Las conexiones más importantes son las de anillo, de malla, de hypercubo y clusters. Para una descripción más detallada de estos esquemas de conexión podemos ver Ortega [96].

Definimos a continuación algunos conceptos fundamentales relacionados con el rendimiento de los algoritmos paralelos. Supongamos que tenemos un computador paralelo que consta de p procesadores.

Definición 2.1 Llamamos **grado de paralelismo** de un algoritmo numérico al número de operaciones en el algoritmo que pueden realizarse en paralelo.

Con el fin de ilustrar el significado de este concepto, consideremos el problema de sumar dos vectores $\boldsymbol{a}$ y $\boldsymbol{b}$ de n componentes. Las sumas de las distintas componentes $a_i + b_i$, para $i = 1, 2, \ldots, n$, son independientes y pueden realizarse en paralelo. Así, el grado de paralelismo de este algoritmo es n y es independiente del número de procesadores de nuestro sistema; constituye una medida intrínseca del paralelismo del algoritmo. El número de procesadores sólo afectará al tiempo necesario para completar la computación. Consideramos ahora el problema de sumar n números $a_1, a_2, \ldots, a_n$ mediante el algoritmo en serie

$$s = a_1, \qquad s = s + a_i, \qquad i = 2, \ldots, n.$$

Este algoritmo resulta poco adecuado para el cálculo en paralelo aunque el problema en sí mismo puede paralelizarse mediante un algoritmo distinto que a continuación describimos. Supongamos que $n = 8$ y que queremos sumar los números $a_1 + a_2 + \cdots + a_8$. La figura 2.1 nos muestra la adición de estos ocho números en tres etapas, cada una de ellas con distinto color. En la primera etapa se realizan cuatro sumas en paralelo, en la segunda se realizan dos sumas, mientras que en la tercera etapa se lleva a cabo una única suma. El algoritmo que muestra la figura 2.1 se conoce con el nombre de ***fan-in*** e ilustra perfectamente el principio *divide y vencerás* que ya se estudió en la sección 1.2. Dividimos el problema de la suma en problemas más pequeños que pueden ser resueltos independientemente unos de otros de forma sencilla.

Los algoritmos de tipo *fan-in* son muy utilizados en computación paralela para la ejecución en paralelo de sumas, productos, cálculos del máximo o mínimo de n números, etc. Claramente vemos que si $n = 2^q$, entonces un algoritmo *fan-in* como el que muestra la figura 2.1 consta de $q = \log_2 n$ etapas donde se realizan $n/2$ sumas en la primera etapa, $n/4$ sumas en la segunda etapa y así sucesivamente. Esto significa que el grado de paralelismo en la primera etapa es de $n/2$, en la segunda es de $n/4$ y así sucesivamente.

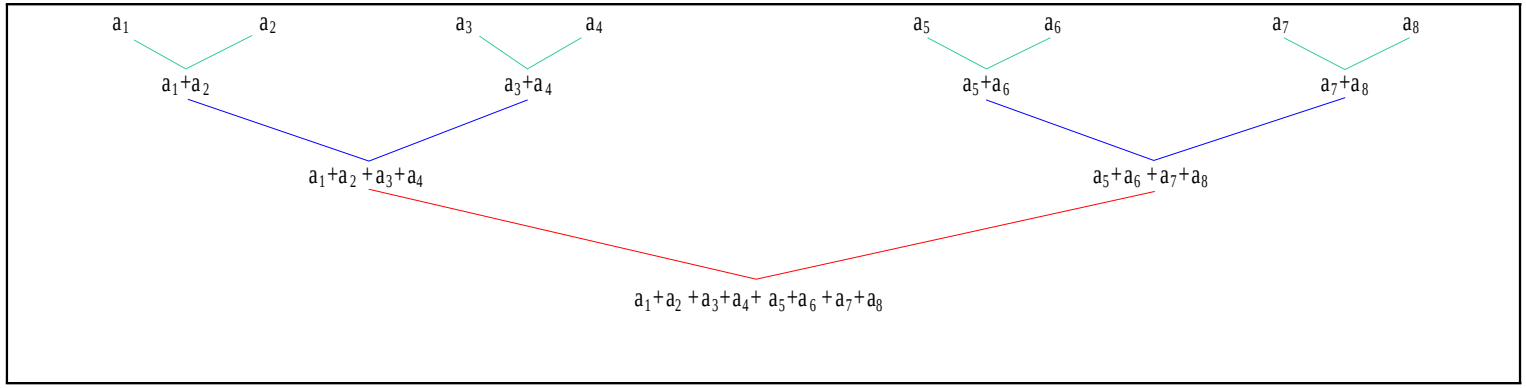


Figura 2.1: *Algoritmo fan-in para la suma de ocho números.*

Relacionado con el grado de paralelismo de un algoritmo está la idea de **granularidad**. Una granularidad a gran escala significa que pueden realizarse grandes tareas de forma independiente en paralelo; una granularidad a pequeña escala significa que pequeñas tareas pueden realizarse en paralelo. Un ejemplo simple de granularidad a pequeña escala puede ser la suma de dos vectores donde la tarea que puede realizarse en paralelo es la adición de dos escalares.

Los conceptos de *speedup* y *eficiencia* son los más utilizados para medir el paralelismo de un algoritmo.

Definición 2.2 El *speedup* de un algoritmo paralelo, que denotamos por S_p, es $S_p = \dfrac{t_1}{t_p}$, donde t_1 es el tiempo de ejecución para un único procesador y t_p es el tiempo de ejecución utilizando p procesadores.

El speedup es una medida de comparación de un algoritmo cuando se utilizan 1 y p procesadores. Relacionado con este concepto está el de eficiencia, E_p, que definimos a continuación.

Definición 2.3 Definimos la eficiencia de un algoritmo paralelo como $E_p = \dfrac{S_p}{p}$.

Un objetivo fundamental en el desarrollo de algoritmos paralelos es conseguir el mayor speedup posible; idealmente, $S_p = p$. Los factores que más influyen en una reducción del speedup son la pérdida de paralelismo en el algoritmo unido al exceso de comunicaciones y el tiempo de sincronización.

Otro de los factores que puede degradar sensiblemente el grado de paralelismo de un algoritmo es el problema del equilibrado de la carga computacional. Por equilibrado de carga queremos significar el reparto de tareas entre los diferentes procesadores del sistema. Este reparto debe ser lo más equilibrado posible de manera que todos los procesadores estén activos el mayor tiempo posible.

2.2 El modelo de computación BSP

El modelo Bulk-Synchronous Parallel Computing BSP, inicialmente propuesto por Valiant [109] y desarrollado posteriormente por McColl [88, 89], abarca una amplia gama de arquitecturas ya que realiza una sencilla abstracción de una computación paralela. La sencillez de esta abstracción está en que un reducido número de parámetros recoge las variaciones más importantes entre las arquitecturas paralelas. El modelo no distingue ciertas diferencias cualitativas como por ejemplo la topología de comunicaciones de las redes de trabajo o las diferencias entre memoria compartida y distribuida que aparecen sólo en forma de diferentes costes para acceder a datos no locales.

Uno de los problemas clásicos en la computación paralela consiste en la escritura de algoritmos que sean válidos para las diferentes arquitecturas existentes. Una solución es la abstracción de los programas de la arquitectura de la máquina. El modelo BSP constituye un estilo de programación paralela desarrollado para paralelismo de propósito general, es decir, para un paralelismo que abarca todas las áreas de aplicación y un amplio rango de arquitecturas (véase McColl [89]). Por tanto, pretende cubrir unos objetivos no alcanzados por los tradicionales modelos de programación paralela que resultan adecuados para ciertos tipos concretos de aplicaciones o que trabajan

de forma eficiente en ciertos tipos particulares de máquinas.

Las propiedades más importantes de este modelo de computación se resumen en los siguientes puntos:

- Los programas BSP son fáciles de escribir. En cierto modo, esa simplicidad hace que sean muy similares a los programas secuenciales.
- El modelo es independiente del tipo de arquitectura. Esto supone que los programas sean independientes de la máquina en la que se estén ejecutando, lo que hace que el grado de portabilidad sea total.
- El rendimiento de un programa en una cierta arquitectura es predecible. Esto significa que el tiempo de ejecución de un algoritmo puede calcularse a partir del texto del programa, utilizando un reducido número de parámetros distintos en cada arquitectura.

Una primera versión de los componentes que integran una computadora BSP sería: un conjunto de procesadores secuenciales con memoria local, un sistema de comunicación que posibilita que los procesadores puedan acceder a datos no locales y un mecanismo para la sincronización global de todos los procesadores.

El término *bulk-synchronous* refleja la posición intermedia que ocupa el modelo BSP entre los extremos del paralelismo síncrono y asíncrono, conceptos que a continuación comentamos brevemente. En un sistema síncrono puro, los procesos se ejecutan en pasos fijos, sincronizados por un reloj global en cada instrucción. En un sistema asíncrono, los procesos pueden trabajar independientemente a diferentes ratios y se sincronizan implícitamente por parejas sólo cuando tiene lugar la comunicación. Los procesos de una computadora BSP se ejecutan concurrentemente a través de un programa, aunque se pueden realizar tareas de comunicación sin que lleven aparejadas una sincronización inmediata.

Definición 2.4 Se define un **superpaso** como un segmento de la computación durante el cuál cada procesador puede realizar una secuencia de operaciones utilizando sólo sus propios datos locales y puede iniciar requerimientos de lectura y escritura de datos contenidos en otros procesadores. Al final de cada superpaso y antes de continuar con el siguiente, todos los procesadores esperan en un determinado punto, llamado **barrera de sincronización**, hasta que se completa el intercambio de datos no locales.

A partir de esta definición podemos subdividir cada superpaso en tres fases claramente diferenciadas: una primera fase de computación local en cada procesador, una segunda fase de comunicación y la última fase que coincide con la barrera de sincronización, que posibilita el movimiento de datos en las memorias locales de cada procesador.

Es importante resaltar que en el transcurso de un superpaso no tiene porqué producirse cálculos de tipo aritmético y comunicaciones. Puede haber superpasos donde únicamente se realizan comunicaciones y superpasos con cálculos aritméticos sólamente. La barrera de sincronización siempre se produce al final de un superpaso, por lo que el tamaño mínimo para un superpaso viene dado por la frecuencia con la que pueda ejecutarse cada barrera de sincronización. Este límite se cuantifica en el modelo BSP mediante un parámetro l que es la periodicidad de sincronización. Al hablar de periodicidad de sincronización nos referimos a la mínima cantidad de tiempo que debe transcurrir entre sucesivas sincronizaciones globales. El tiempo transcurrido se mide en pasos de computación básicos, que es el tiempo que tarda un procesador en realizar una operación sobre datos locales, es decir, un *flop*.

El coste de las comunicaciones globales se expresa mediante el parámetro g y se define a partir del concepto de h-relación, que a continuación se expone.

Definición 2.5 Definimos una **h-relación** como un patrón de comunicación global en el que cada procesador puede enviar y recibir hasta un total de h unidades de datos. A partir del concepto de h-relación, podemos definir el parámetro g diciendo que una h-relación arbitraria puede comunicarse en hg unidades de tiempo, imponiendo que h es mayor que algún valor h_0.

Un modelo de comunicación en el que cada procesador envía un dato a otro procesador distinto representa una 1-relación. Sin embargo, un modelo de comunicación en el que un procesador envía un único dato al resto representa una p-relación. Este último modelo de comunicación se conoce con el nombre de **broadcast** y tiene gran importancia por tratarse de un modelo de comunicación muy utilizado en algoritmos de álgebra lineal. El modelo BSP no distingue entre un mensaje de longitud m y m mensajes de longitud 1. El coste es similar en ambos casos y viene determinado por mg. En cuanto a los procesos de comunicación cabe señalar que si en un superpaso la cantidad total de datos comunicados es muy pequeña, entonces los efectos del *start-up* de la máquina pueden jugar un papel importante en el rendimiento del algoritmo.

El parámetro g representa la razón de la computación global por la capacidad de comunicación global y su cota depende de si se tienen suficientes datos para intercambiar, por lo que el sistema de comunicaciones puede alcanzar la capacidad indicada. Además, g constituye una medida bastante precisa del coste de comunicar grandes cantidades de datos en una amplia gama de computadoras paralelas.

En resumen, podemos afirmar que una computadora BSP queda definida mediante los siguientes parámetros:

- p, que es el número de procesadores.
- s, que es la velocidad de computación de cada procesador, medida en megaflops por segundo, Mflop/seg (millones de operaciones aritméticas en coma flotante por segundo).
- l, que es el número de unidades de tiempo necesarias para realizar una sincronización entre los procesadores. El parámetro corresponde a la latencia de la red de comunicación.
- g, que representa una medida de la razón entre la capacidad total de la CPU por segundo y la capacidad total de comunicación por segundo, esto es, el cociente entre el número total de operaciones básicas que se pueden realizar en un segundo por todos los procesadores y el número total de unidades de datos que se pueden repartir por la red de comunicación en un segundo.

Notemos que con el fin de poder comparar el rendimiento paralelo en diversos sistemas, los parámetros de tiempo transcurrido se miden en pasos básicos de comunicación. Si la velocidad del procesador viene dada en flop/seg (flops por segundo), entonces el parámetro l también se expresará en flops y g vendría dada por flop/palabra (flops por palabra). La unidad flop/palabra representa el número de operaciones locales que cada procesador podría ejecutar durante el tiempo que tarda cada uno de ellos en comunicar una palabra en coma flotante.

2.3 El modelo de programación

El modelo propuesto por Valiant [109] no presupone ningún tipo de máquina específica que identifique dicho modelo. En la práctica la implementación se realiza con posterioridad y se organiza de acuerdo con el formato SPMD. En este modelo cada procesador paralelo ejecuta el mismo texto del programa, lo que significa que cada procesador ejecuta una y sólo una copia del programa original y dicha copia en ejecución se denomina proceso, por lo que hablar de proceso y procesador es equivalente. Los procesos se ejecutan a través de una secuencia de *superpasos*. Dentro de un superpaso cada procesador puede ejecutar cálculos distintos pero todos deben llegar al final de dicho superpaso antes de continuar con el siguiente. Cada procesador tiene su propio espacio de datos, pudiendo acceder de forma explícita al espacio de memoria del resto. Dicho acceso a datos remotos es asíncrono, lo que significa que no se puede garantizar que una operación de búsqueda de datos o almacenaje de un elemento se produzca hasta el final del superpaso actual, momento en el que se sincronizan todos los procesadores.

Como ya se mencionó en la sección 2.2 el concepto fundamental del modelo de programación es el de superpaso. Un algoritmo BSP es una serie de superpasos en los que se pueden desarrollar operaciones aritméticas y llamadas a datos no locales. Desde que este modelo fue propuesto por Valiant en 1990, se han venido desarrollando diversas librerías de funciones que han permitido la implementación de un buen número de algoritmos en computadores BSP. Algunas de estas librerías son extensiones de lenguajes de programación ya existentes como el C/C++ y FORTRAN. A continuación citamos algunas librerías que han sido implementadas en diferentes arquitecturas.

- **La librería BSP de Oxford**. Esta librería se desarrolló en Oxford en 1993 en el grupo de computación paralela dirigido por McColl y fue la primera librería que se implementó para este modelo utilizando los lenguajes C y FORTRAN. La versión original de esta librería era muy reducida, constando únicamente de seis funciones. Posteriores versiones han desarrollado notablemente las posibilidades de la misma a través de nuevas funciones que completan las existentes. Véase Donaldson, Hill y Skillicorn [37] para una descripción más detallada de la última versión de esta librería.

- **BSP++**. Constituye una aproximación a la programación orientada a objetos para computadores BSP. Es una extensión de la

librería de Oxford y aparece en el año 1994. Véase Lecomber [82] para un análisis más detallado de esta librería.

- **La librería Green BSP**. Esta librería data de 1995 y su característica más importante es que implementa como único método de comunicación el paso de mensajes. Su implementación es en lenguaje C. Véase Goudreau y otros [56] para una descripción más detallada de esta librería.
- **La librería BSP Worlwide Standard**. Es el resultado de la colaboración entre los grupos de Oxford y de la librería Green. Data de 1996 y se implementa tanto en lenguaje C como FORTRAN. Véase Goudreau y otros [55] para una descripción más detallada de esta librería.
- **Librería PUB**. Se trata de la librería desarrollada más recientemente, concretamete en 1999 en la Universidad de Paderborn (Alemania). Está implementada en C. Para profundizar en las características particulares de esta librería, véase Bonorden y otros [15].

Estas librerías tienen en común el haber sido desarrolladas y probadas en una amplia variedad de plataformas entre las que podemos citar IBM SP1 e IBM SP2, SGI *Power Challenge*, *Silicon Grafics Origin* 2000 CRAY T3D y CRAY T3E, *Convex* SPP, *clusters* de estaciones de trabajo Solaris (TCP/IP), *clusters* de Pentiums bajo Linux (TCP/IP), *clusters* de estaciones de trabajo IBM RS6000, *Hitachi* SR2001, *SunOS* 4.1.x y *Parsytec Explorer*.

2.4 El modelo de coste

Como ya se ha visto anteriormente, los parámetros l y g cuantifican el rendimiento del sistema. Podemos utilizar estos parámetros para predecir el rendimiento de un determinado programa o algoritmo que se esté ejecutando en el sistema. Un programa BSP está compuesto por una secuencia de superpasos que se ejecutan síncronamente, con lo que el tiempo total transcurrido será la suma de los tiempos invertidos en cada superpaso.

El modelo de coste, por tanto, se basa en el coste individual de cada superpaso del programa, que viene dado por la expresión

$$C_s = \max_{1 \leq i \leq p} w_i + \max_{1 \leq i \leq p} h_i g + l, \tag{2.1}$$

donde C_s representa el coste total del superpaso, w_i es el tiempo de computación aritmética del procesador P_i en el transcurso del superpaso, h_i es el número de unidades de datos que el procesador P_i envía o recibe durante el superpaso y g y l son los valores de los parámetros definidos en la sección 2.2.

En consecuencia, el coste de un programa BSP es la suma del coste de todos sus superpasos. Las características de este modelo de coste nos sugieren diversas estrategias muy apropiadas para obtener programas BSP eficientes. Entre éstas podemos citar

- Equilibrar la computación en cada superpaso entre los diferentes procesadores, es decir, que cada procesador tenga asignado un trabajo similar, ya que la barrera de sincronización debe esperar a que el procesador más lento acabe las tareas encomendadas.
- Equilibrar las comunicaciones entre los procesadores, ya que el coste de comunicación es un máximo entre elementos enviados y recibidos.
- Minimizar el número de superpasos, con lo que estamos minimizando las barreras de sincronización.

Analizamos ahora, siguiendo el modelo de coste anteriormente expuesto, el coste de algunos modelos de comunicación habituales en los algoritmos del tipo *divide y vencerás*. Así, si tenemos en cuenta que una h-relación es un patrón de comunicación global en el que cada procesador puede enviar o recibir a lo sumo h unidades de datos y su coste es de hg flops, establecemos el coste BSP de los siguientes modelos de comunicación.

(i) Cada procesador envía un elemento al procesador siguiente. El coste de esta comunicación es $1g$ flop, ya que tenemos una 1-relación.

(ii) Cada procesador tiene un mensaje de longitud h entrando y saliendo de él. El coste BSP de este patrón de comunicación es de hg flops, de acuerdo con la expresión (2.1).

(iii) Desde p procesadores se envía a otro procesador un mensaje de longitud $\frac{n}{p}$ unidades, suponiendo que n es múltiplo de p. En este caso, el coste viene dado por el número de elementos recibidos por el procesador receptor de los mismos, ya que es el máximo. En este procesador entran n unidades de datos lo que representa una n-relación, con un coste de ng flops.

(iv) Los procesadores pares envían a los procesadores pares un mensaje de longitud m, suponiendo que p es múltiplo de 2. Según este modelo de comunicación, en cada procesador par entran y salen $\frac{p}{2} - 1$ mensajes de longitud m, por lo que se trata de una $\left(\frac{p}{2} - 1\right)$-relación, y su coste viene dado por $\left(\frac{p}{2} - 1\right) g$ flops.

(v) Se considera el patrón de comunicación en el que desde un procesador se realiza un *broadcast* de n elementos. El coste de este patrón de comunicación es de $n(p-1)g$ flops, ya que el procesador del que salen los datos envía un total de $n(p-1)$ datos al resto de procesadores, esto es, una $n(p-1)$-relación.

2.5 Valores de s, g y l en distintas máquinas

El modelo de coste nos permite la predicción en la ejecución de los programas BSP en diversas arquitecturas. Basta con calcular el coste computacional y de comunicación en cada superpaso y sustituir los valores de p, s, g y l en la expresión (2.1). Los valores de estos parámetros pueden medirse en cada máquina por medio de software. A lo largo de esta memoria se presentan resultados teóricos para todos los algoritmos estudiados con el fin de establecer comparaciones en los tiempos de ejecución de dichos algoritmos. Las máquinas elegidas para realizar un estudio comparativo de tiempos entre los diferentes algoritmos presentados son

- un IBM SP2, dotado con dos tipos de conexiones para efectuar las comunicaciones entre los procesadores: un switch de alto rendimiento y una conexión ethernet. Consta de ocho procesadores P2SC a 66.7 Mhz y 128 Mbytes de memoria principal por cada procesador,

- un CRAY T3D, que consta de 256 procesadores del tipo DECchip 21164 a 150 Mhz con 64 Mbytes de memoria local por cada procesador,
- un cluster de Pentiums formado por ocho Pentiums II a 400 Mhz con 128 Mbytes de memoria por procesador, conectados entre sí utilizando un switch ethernet Cisco 2916XL a 100 Mbytes por segundo.

Los valores de los parámetros p, s, g y l para las máquinas citadas anteriormente se muestran en la tabla 2.1 y son los que publica el grupo de BSP de la Universidad de Oxford. Las unidades utilizadas para medir estos parámetros son las siguientes: s se mide en megaflops por segundo (Mflops), p es el número de procesadores, l se mide en flops y g se mide en flops por palabra (flop/word), que en este caso son palabras de 32 bits. En la tabla 2.1 aparece $n_{1/2}$, que constituye una medida del tamaño del mensaje que produce una comunicación óptima en la red de comunicaciones de la computadora paralela.

Describimos brevemente a continuación la forma en que el grupo BSP de Oxford calcula los parámetros que aparecen en la tabla 2.1. El cálculo del parámetro s depende fuertemente del tipo de cálculo que llevamos a cabo. Se realizan dos cálculos del parámetro y se toma el valor medio. El primero consiste en medir el coste de un producto de vectores de tamaño n donde el número de operaciones que se realizan es de orden n. Se trabaja con estructuras de datos de tamaño n, siendo este tamaño superior al tamaño del caché de cada procesador. Este valor de s constituye una cota inferior para la velocidad del procesador. Por otra parte, se calcula el parámetro s para el coste de un producto de matrices densas donde el número de operaciones que se realizan es de orden n^3. Las estructuras de datos ahora son de tamaño n^2 y una buena parte de los cálculos se guardan en el caché de cada procesador. Este valor de s constituye una cota superior de la velocidad del procesador.

Como ya se ha comentado en el transcurso de esta sección, la eficiencia de un algoritmo BSP depende en gran medida del diseño de un modelo de comunicación lo más equilibrado posible. Así, del mismo modo que para la medida de s tomábamos el valor medio cuando se desarrollaban dos tipos distintos de cálculos, ahora medimos la capacidad de comunicación g del sistema utilizando dos modelos distintos de comunicación equilibrada. El primero de ellos es el de una 1-relación, donde cada procesador comunica a su procesador vecino una unidad de datos. Como segundo modelo de comunicación se considera una k-relación en la que se produce un intercambio de datos entre

s	p	l	g	$n_{1/2}$ words
26	2	1903	6.3	6
	4	3583	6.4	7
	8	5412	6.9	6

(a) *IBM SP2 switch*

s	p	l	g	$n_{1/2}$ words
26	2	18759	182.1	3
	4	39025	388.2	5
	8	88795	1246.6	2

(b) *IBM SP2 ethernet*

s	p	l	g	$n_{1/2}$ words
12	2	164	0.7	71
	4	168	0.7	66
	8	175	0.8	59
	16	181	0.9	61
	32	201	1.1	28
	64	148	1.0	27
	128	301	1.1	20
	256	387	1.2	15

(c) *CRAY T3D*

s	p	l	g	$n_{1/2}$ words
88	2	5654	31.9	5
	4	11759	31.4	5
	8	18347	30.5	32

(d) *Cluster de Pentiums*

Tabla 2.1: *Valores de los parámetros s, g, l y $n_{1/2}$.*

todos los procesadores (cada procesador realiza un broadcast de k elementos). Esto nos proporciona dos medidas de g distintas, una que llamamos **local**, correspondiente a la 1-relación y otra que llamamos **total**, correspondiente a la pg-relación. Los valores de g que se muestran en la tabla 2.1 se refieren al valor de g local y para comunicaciones de palabras de 32-bits.

La tabla 2.1 nos muestra un valor único del parámetro g para cada valor de p, independientemente del tamaño de la comunicación que se realiza. En la predicción teórica de tiempos que se realiza para cada algoritmo a lo largo de los siguientes capítulos hemos utilizado el modelo propuesto por Hockney [64] y modificado por Miller [95]. En este modelo se debe tener en cuenta el efecto de la granularidad, por lo que se realizan comunicaciones aumentando el tamaño de los bloques comunicados, obteniéndose una gráfica que relaciona el cociente entre el tiempo t (en flops) y el tamaño n del bloque comunicado. Si suponemos que existe una relación lineal entre t y n, tendremos que

$$t(n) = kn + l,$$

siendo k el tiempo que tarda en comunicarse una palabra y l el tiempo de una sincronización. La relación entre n y t se puede expresar como $g(n) = \frac{t}{n}$. Suponemos ahora que g_∞ es el valor asintótico al que se aproxima $g(n)$ cuando n tiende a ∞ y que $n_{1/2}$ es el tamaño del mensaje que produce la mitad del ancho de banda óptimo de la máquina, es decir, $g\left(n_{1/2}\right) = 2g_\infty$. Bajo estos supuestos, podemos expresar

$$t(n) = \left(n + n_{1/2}\right) g_\infty, \tag{2.2}$$

de donde

$$g(n) = \frac{t(n)}{n} = \left(1 + \frac{n_{1/2}}{n}\right) g_\infty. \tag{2.3}$$

La expresión (2.2) nos muestra que cuando $n = n_{1/2}$ entonces la mitad del tiempo se utiliza en la comunicación propiamente dicha y la otra mitad en el start-up. Se considera como valor BSP de la máquina $g = g(n_{1/2}) = 2g_\infty$.

Para calcular el coste g de la comunicación de una palabra se utiliza la expresión (2.3). Sin embargo, distinguimos los casos siguientes:

- si $n \gg n_{1/2}$ entonces $g = g_\infty$.
- si $n \ll n_{1/2}$ entonces $\frac{n}{n_{1/2}} \cong 0$, por lo que podemos tomar $t(n) = n_{1/2} g_\infty$. Así, tendremos que

$$g = \frac{t(n)}{n} = \frac{n_{1/2} g_\infty}{n}.$$

El valor de l representa el coste de una barrera de sincronización y puede ser calculado empíricamente en cada máquina. Una forma de calcular l es mediante un programa que incluya una sucesión de superpasos vacíos para determinar el coste medio de una barrera de sincronización.

Analizando los parámetros que aparecen en la tabla 2.1, se observa que el CRAY T3D es una máquina donde la velocidad de los procesadores no es excesivamente alta, sin embargo, las comunicaciones son más rápidas que las operaciones aritméticas hasta $p = 16$ y prácticamente igual de rápidas para el resto de procesadores. Para $p = 256$ el valor de g es 1.2, lo que nos da idea de la rapidez con que se comunican datos en una máquina de este tipo. Teniendo en cuenta estas características cabe esperar que aquellos algoritmos con una gran carga de comunicaciones y menor carga computacional sean los que mejor rendimiento ofrezcan. Hay que resaltar los valores de los parámetros que se consiguen para 64 procesadores. Se observa que el coste de realizar una barrera de sincronización desciende hasta 148 flop/seg., mientras que realizar la comunicación de una palabra de 32 bits cuesta exactamente lo mismo que realizar una operación en coma flotante. Esto significa que deberíamos obtener mejores tiempos en los algoritmos que estudiemos para $p = 64$ que para $p = 32$.

La máquina que presenta peores valores para los parámetros BSP es el IBM SP2 con conexión ethernet. Como muestra digamos que realizar una comunicación trabajando con $p = 8$ cuesta lo mismo que realizar 1246.6 operaciones aritméticas y una barrera de sincronización tarda un tiempo equivalente a realizar 88795 operaciones aritméticas.

Capítulo 3 Algoritmos *divide y vencerás* basados en la fórmula de Sherman-Morrison

3.1 Introducción

En esta sección presentamos un método general para resolver un sistema lineal

$$A\boldsymbol{x} = \boldsymbol{d} \tag{3.1}$$

siguiendo una estrategia de actualización de rango uno de la matriz A. Empezamos describiendo el método general para, posteriormente en la sección 3.2, estudiar el caso tridiagonal.

Supongamos que podemos obtener una matriz no singular G de manera que

$$A = G + \sum_{i=1}^{m-1} \boldsymbol{u}_i \boldsymbol{v}_i^T \tag{3.2}$$

donde $\boldsymbol{u}_i, \boldsymbol{v}_i \in \mathbb{R}^n$, para $i = 1, 2, \ldots, m-1$.

Sea

$$B_i = B_{i-1} + \boldsymbol{u}_i \boldsymbol{v}_i^T, \qquad i = 1, 2, \ldots, m-1 \tag{3.3}$$

con $B_0 = G$. Claramente $B_{m-1} = A$, por tanto, la única solución $\boldsymbol{x} = A^{-1}\boldsymbol{d}$ del sistema (3.1) puede obtenerse utilizando la fórmula de Sherman-Morrison (véase la sección 1.1 para una descripción más detallada de dicha fórmula), como

$$\boldsymbol{x} = \left(I - \frac{1}{1 + \boldsymbol{v}_{m-1}^T \boxed{B_{m-2}^{-1}\boldsymbol{u}_{m-1}}} \boxed{B_{m-2}^{-1}\boldsymbol{u}_{m-1}} \boldsymbol{v}_{m-1}^T \right) \boxed{B_{m-2}^{-1}\boldsymbol{d}},$$

lo cual implica calcular previamente los vectores $B_{m-2}^{-1}\boldsymbol{d}$ y $B_{m-2}^{-1}\boldsymbol{u}_{m-1}$. Aplicando de nuevo la fórmula de Sherman-Morrison a la ecuación (3.3) para $i = m-2$ obtenemos que

$$B_{m-2}^{-1}\boldsymbol{d} = \left(I - \frac{1}{1 + \boldsymbol{v}_{m-2}^T \boxed{B_{m-3}^{-1}\boldsymbol{u}_{m-2}}} \boxed{B_{m-3}^{-1}\boldsymbol{u}_{m-2}} \boldsymbol{v}_{m-2}^T \right) \boxed{B_{m-3}^{-1}\boldsymbol{d}},$$

$$B_{m-2}^{-1}\boldsymbol{u}_{m-1} = \left(I - \frac{1}{1 + \boldsymbol{v}_{m-2}^T \boxed{B_{m-3}^{-1}\boldsymbol{u}_{m-2}}} \boxed{B_{m-3}^{-1}\boldsymbol{u}_{m-2}} \boldsymbol{v}_{m-2}^T \right) \boxed{B_{m-3}^{-1}\boldsymbol{u}_{m-1}},$$

lo que significa que debemos conocer los vectores $B_{m-3}^{-1}\boldsymbol{d}$, $B_{m-3}^{-1}\boldsymbol{u}_{m-2}$ y $B_{m-3}^{-1}\boldsymbol{u}_{m-1}$.

Siguiendo con este esquema de cálculo, obtenemos que si $i = m-2, m-3, \ldots, 2, 1$, para calcular $B_i^{-1}\boldsymbol{d}$ y $B_i^{-1}\boldsymbol{u}_j$, con $j = i+1, i+2, \ldots, m-1$, necesitamos calcular previamente $B_{i-1}^{-1}\boldsymbol{d}$ y $B_{i-1}^{-1}\boldsymbol{u}_l$, para $l = i, i+1, \ldots, m-1$.

En consecuencia, si por $\boldsymbol{x}_i$ denotamos la solución del sistema $B_i\boldsymbol{x} = \boldsymbol{d}$, es decir $\boldsymbol{x}_i = B_i^{-1}\boldsymbol{d}$ y por $\boldsymbol{w}_{i,k}$ la solución del sistema $B_i\boldsymbol{x} = \boldsymbol{u}_k$, es decir $\boldsymbol{w}_{i,k} = B_i^{-1}\boldsymbol{u}_k$ podemos calcular la solución del sistema (3.1) dada por $\boldsymbol{x} = A^{-1}\boldsymbol{d} = B_{m-1}^{-1}\boldsymbol{d} = \boldsymbol{x}_{m-1}$ utilizando el siguiente algoritmo basado en la fórmula de Sherman-Morrison y los comentarios anteriores.

Algoritmo 3.1 Un algoritmo recursivo general.

1 Inicialización.
Resolver los sistemas $B_0\boldsymbol{x} = \boldsymbol{d}$ y $B_0\boldsymbol{x} = \boldsymbol{u}_i$, para $i = 1, 2, \ldots, m-1$, con el fin de obtener $\boldsymbol{x}_0 = B_0^{-1}\boldsymbol{d}$ y $\boldsymbol{w}_{0,i} = B_0^{-1}\boldsymbol{u}_i$, para $i = 1, 2, \ldots, m-1$.

2 Actualización.
Para $i = 1, 2, \ldots, m-2$ calcular

$$\boldsymbol{x}_i = \boldsymbol{x}_{i-1} - \frac{\boldsymbol{v}_i^T\boldsymbol{x}_{i-1}}{1+\boldsymbol{v}_i^T\boldsymbol{w}_{i-1,i}}\boldsymbol{w}_{i-1,i} \quad \text{y} \quad \boldsymbol{w}_{i,j} = \boldsymbol{w}_{i-1,j} - \frac{\boldsymbol{v}_i^T\boldsymbol{w}_{i-1,j}}{1+\boldsymbol{v}_i^T\boldsymbol{w}_{i-1,i}}\boldsymbol{w}_{i-1,i} \quad \text{para} \quad j = i+1, i+2, \ldots, m-1.$$

3 Obtención de la solución.
Calcular

$$\boldsymbol{x}_{m-1} = \boldsymbol{x}_{m-2} - \frac{\boldsymbol{v}_{m-1}^T\boldsymbol{x}_{m-2}}{1+\boldsymbol{v}_{m-1}^T\boldsymbol{w}_{m-2,m-1}}\boldsymbol{w}_{m-2,m-1}.$$

3.2 Un algoritmo recursivo para sistemas tridiagonales

A partir de esta sección y en lo que resta del capítulo supondremos que la matriz

$$A = \begin{bmatrix} a_1 & b_1 & & & \\ c_2 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & c_n & a_n \end{bmatrix} \tag{3.4}$$

del sistema (3.1) es tridiagonal, irreducible y diagonal dominante. Con objeto de simplificar la notación, supondremos a partir de ahora que $n = 2^q$ para un cierto entero $q \geq 1$, que

$$G = \begin{bmatrix} G_1 & & & \\ & G_2 & & \\ & & \ddots & \\ & & & G_{2^{q-1}} \end{bmatrix} \qquad \text{con} \qquad G_i = \begin{bmatrix} \hat{a}_{2i-1} & b_{2i-1} \\ c_{2i} & \hat{a}_{2i} \end{bmatrix}, \quad i = 1, 2, \ldots, 2^{q-1},$$

donde

$$\begin{aligned} \hat{a}_1 &= a_1 \\ \hat{a}_{2i} &= a_{2i} - c_{2i+1}, \quad i = 1, 2, \ldots, 2^{q-1} - 1, && (3.5) \\ \hat{a}_{2i-1} &= a_{2i-1} - b_{2i-2}, \quad i = 2, 3, \ldots, 2^{q-1}, && (3.6) \\ \hat{a}_n &= a_n \end{aligned}$$

y que $\boldsymbol{u}_i = \boldsymbol{e}_{2i} + \boldsymbol{e}_{2i+1}$ y $\boldsymbol{v}_i = c_{2i+1}\boldsymbol{e}_{2i} + b_{2i}\boldsymbol{e}_{2i+1}$, donde $\boldsymbol{e}_{2i}$ y $\boldsymbol{e}_{2i+1}$ son las columnas $2i$-ésima y $(2i+1)$-ésima de la matriz I_n.

Debido a la estructura de las matrices A y G y de los vectores $\boldsymbol{u}_i$ y $\boldsymbol{v}_i$, en esta sección proponemos una modificación del método introducido en la sección 3.1, que puede implementarse utilizando un algoritmo recursivo en el que únicamente se actualizan los bloques de vectores no nulos. Se utiliza la notación $\boldsymbol{x}(k : l)$ con $k \leq l$ para denotar el vector formado por las componentes $k, k+1, \ldots, l$ del vector $\boldsymbol{x}$.

Algoritmo 3.2 Un algoritmo basado en el método de desacoplamiento recursivo para sistemas tridiagonales.

1 Inicialización.

1.1 Para $i = 1, 2, \ldots, 2^{q-1}$, resolver los sistemas

$$G_i \boldsymbol{x}(2i-1:2i) = \boldsymbol{d}(2i-1:2i). \tag{3.7}$$

1.2 Para $i = 1, 2, \ldots, 2^{q-1}-1$, resolver los sistemas

$$G_i \boldsymbol{w}_i(2i-1:2i) = \boldsymbol{u}_i(2i-1:2i) \qquad \text{y} \qquad G_{i+1} \boldsymbol{w}_i(2i+1:2i+2) = \boldsymbol{u}_i(2i+1:2i+2). \tag{3.8}$$

2 Actualización. Para $k = q-2, q-3, \ldots, 2, 1$, y para $i = 1, 2, \ldots, 2^k$, calcular

$$\begin{aligned}\boldsymbol{x}_{(2^{q-k}(i-1)+1:2^{q-k}i)} &= \begin{bmatrix} \boldsymbol{x}_{(2^{q-k}(i-1)+1:2^{q-1-k}(2i-1))} \\ \boldsymbol{x}_{(2^{q-1-k}(2i-1)+1:2^{q-k}i)} \end{bmatrix} \\ &- \frac{\boldsymbol{v}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)^T} \begin{bmatrix} \boldsymbol{x}_{(2^{q-k}(i-1)+1:2^{q-1-k}(2i-1))} \\ \boldsymbol{x}_{(2^{q-1-k}(2i-1)+1:2^{q-k}i)} \end{bmatrix}}{1 + \boldsymbol{v}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)^T} \boldsymbol{w}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)}} \boldsymbol{w}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)},\end{aligned} \tag{3.9}$$

$$\begin{aligned}\boldsymbol{w}_{2^{q-1-k}i(2^{q-k}(i-1)+1:2^{q-k}i)} &= \begin{bmatrix} \boldsymbol{0}_{(1:2^{q-1-k})} \\ \boldsymbol{w}_{2^{q-1-k}i(2^{q-1-k}(2i-1)+1:2^{q-k}i)} \end{bmatrix} \\ &- \frac{\boldsymbol{v}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)^T} \begin{bmatrix} \boldsymbol{0}_{(1:2^{q-1-k})} \\ \boldsymbol{w}_{2^{q-1-k}i(2^{q-1-k}(2i-1)+1:2^{q-k}i)} \end{bmatrix}}{1 + \boldsymbol{v}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)^T} \boldsymbol{w}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)}} \boldsymbol{w}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)},\end{aligned} \tag{3.10}$$

$$\boldsymbol{w}_{2^{q-1-k}(i-1)(2^{q-k}(i-1)+1:2^{q-k}i)} = \begin{bmatrix} \boldsymbol{w}_{2^{q-1-k}(i-1)(2^{q-k}(i-1)+1:2^{q-1-k}(2i-1))} \\ \mathbf{0}_{(1:2^{q-1-k})} \end{bmatrix}$$
$$- \frac{\boldsymbol{v}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)^T} \begin{bmatrix} \boldsymbol{w}_{2^{q-1-k}(i-1)(2^{q-k}(i-1)+1:2^{q-1-k}(2i-1))} \\ \mathbf{0}_{(1:2^{q-1-k})} \end{bmatrix}}{1+\boldsymbol{v}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)^T}\boldsymbol{w}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)}} \boldsymbol{w}_{2^{q-2-k}(2i-1)(2^{q-k}(i-1)+1:2^{q-k}i)}. \quad (3.11)$$

En este paso, suponemos que $\boldsymbol{w}_0$ y $\boldsymbol{w}_{2^{q-1}}$ son vectores nulos.

3 Obtención de la solución. Calcular

$$\boldsymbol{x}_{(1:2^q)} = \begin{bmatrix} \boldsymbol{x}_{(1:2^{q-1})} \\ \boldsymbol{x}_{(2^{q-1}+1:2^q)} \end{bmatrix} - \frac{\boldsymbol{v}_{2^{q-2}(1:2^q)^T} \begin{bmatrix} \boldsymbol{x}_{(1:2^{q-1})} \\ \boldsymbol{x}_{(2^{q-1}+1:2^q)} \end{bmatrix}}{1+\boldsymbol{v}_{2^{q-2}(1:2^q)^T}\boldsymbol{w}_{2^{q-2}(1:2^q)}} \boldsymbol{w}_{2^{q-2}(1:2^q)}. \quad (3.12)$$

El algoritmo 3.2 constituye una modificación del método propuesto por Evans [44], Spaletta y Evans [101] y Mehrmann [90]. Spaletta y Evans [101] aplican la fórmula de Sherman-Morrison para calcular A^{-1} mediante la expresión

$$\left(G + \sum_{i=1}^{m-1} \boldsymbol{u}_i \boldsymbol{v}_i^T\right)^{-1} = G^{-1} - \sum_{i=1}^{m-1} \alpha_i G^{-1} \boldsymbol{u}_i \boldsymbol{v}_i^T G^{-1} \quad (3.13)$$

siendo

$$\alpha_i = \frac{1}{1+\boldsymbol{v}_i^T G^{-1} \boldsymbol{u}_i}, \qquad i = 1, 2, \ldots, m-1.$$

Sin embargo, esta fórmula no es correcta para $m > 2$ como se puede ver en el ejemplo siguiente.

Ejemplo 3.1 *Consideremos el problema de resolver el sistema (3.1) con*

$$A = \begin{bmatrix} 2 & -1 & & & & & & \\ -1 & 2 & -1 & & & & & \\ & -1 & 2 & -1 & & & & \\ & & -1 & 2 & -1 & & & \\ & & & -1 & 2 & -1 & & \\ & & & & -1 & 2 & -1 & \\ & & & & & -1 & 2 & -1 \\ & & & & & & -1 & 2 \end{bmatrix} \qquad y \qquad \boldsymbol{d} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

De acuerdo con la expresión (3.3) y teniendo en cuenta las definiciones de los vectores $\boldsymbol{u}_i$ y $\boldsymbol{v}_i$, podemos escribir la matriz A como

$$A = \begin{bmatrix} G_1 & & & \\ & G_2 & & \\ & & G_3 & \\ & & & G_4 \end{bmatrix} + \boldsymbol{u}_1\boldsymbol{v}_1^T + \boldsymbol{u}_2\boldsymbol{v}_2^T + \boldsymbol{u}_3\boldsymbol{v}_3^T,$$

siendo

$$\boldsymbol{u}_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \qquad \boldsymbol{u}_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \qquad \boldsymbol{u}_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad y \qquad \boldsymbol{v}_1 = \begin{bmatrix} 0 \\ -1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \qquad \boldsymbol{v}_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \\ -1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \qquad \boldsymbol{v}_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ -1 \\ 0 \end{bmatrix}.$$

Entonces

$$G = \begin{bmatrix} 2 & -1 & & & & & & \\ -1 & 3 & & & & & & \\ & & 3 & -1 & & & & \\ & & -1 & 3 & & & & \\ & & & & 3 & -1 & & \\ & & & & -1 & 3 & & \\ & & & & & & 3 & -1 \\ & & & & & & -1 & 2 \end{bmatrix} \qquad y \qquad G^{-1} - \sum_{i=1}^{3} \alpha_i G^{-1} \boldsymbol{u}_i \boldsymbol{v}_i G^{-1} = \begin{bmatrix} \frac{7}{9} & \frac{5}{9} & \frac{1}{3} & \frac{1}{9} & 0 & 0 & 0 & 0 \\ \frac{5}{9} & \frac{10}{9} & \frac{2}{3} & \frac{2}{9} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{2}{3} & \frac{17}{16} & \frac{25}{48} & \frac{3}{16} & \frac{1}{16} & 0 & 0 \\ \frac{1}{9} & \frac{2}{9} & \frac{25}{48} & \frac{145}{144} & \frac{9}{16} & \frac{3}{16} & 0 & 0 \\ 0 & 0 & \frac{3}{16} & \frac{9}{16} & \frac{145}{144} & \frac{25}{48} & \frac{2}{9} & \frac{1}{9} \\ 0 & 0 & \frac{1}{16} & \frac{3}{16} & \frac{25}{48} & \frac{17}{16} & \frac{2}{3} & \frac{1}{3} \\ 0 & 0 & 0 & 0 & \frac{2}{9} & \frac{2}{3} & \frac{10}{9} & \frac{5}{9} \\ 0 & 0 & 0 & 0 & \frac{1}{9} & \frac{1}{3} & \frac{5}{9} & \frac{7}{9} \end{bmatrix}.$$

Sin embargo,

$$A^{-1} = \begin{bmatrix} \frac{8}{9} & \frac{7}{9} & \frac{2}{3} & \frac{5}{9} & \frac{4}{9} & \frac{1}{3} & \frac{2}{9} & \frac{1}{9} \\ \frac{7}{9} & \frac{14}{9} & \frac{4}{3} & \frac{10}{9} & \frac{8}{9} & \frac{2}{3} & \frac{4}{9} & \frac{2}{9} \\ \frac{2}{3} & \frac{4}{3} & 2 & \frac{5}{3} & \frac{4}{3} & 1 & \frac{2}{3} & \frac{1}{3} \\ \frac{5}{9} & \frac{10}{9} & \frac{5}{3} & \frac{20}{9} & \frac{16}{9} & \frac{4}{3} & \frac{8}{9} & \frac{4}{9} \\ \frac{4}{9} & \frac{8}{9} & \frac{4}{3} & \frac{16}{9} & \frac{20}{9} & \frac{5}{3} & \frac{10}{9} & \frac{5}{9} \\ \frac{1}{3} & \frac{2}{3} & 1 & \frac{4}{3} & \frac{5}{3} & 2 & \frac{4}{3} & \frac{2}{3} \\ \frac{2}{9} & \frac{4}{9} & \frac{2}{3} & \frac{8}{9} & \frac{10}{9} & \frac{4}{3} & \frac{14}{9} & \frac{7}{9} \\ \frac{1}{9} & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \frac{5}{9} & \frac{2}{3} & \frac{7}{9} & \frac{8}{9} \end{bmatrix}.$$

La modificación que introduce el algoritmo 3.2 respecto del método que proponen Evans [44], Spaletta y Evans [101] y Mehrmann [90], se basa en una reducción del número de actualizaciones que se realizan, ya que únicamente se consideran las componentes no nulas de los vectores que son actualizados. A continuación resolvemos detalladamente un ejemplo donde ponemos de manifiesto el funcionamiento de dicho algoritmo.

Ejemplo 3.2 *Consideremos el problema de resolver el sistema (3.1), donde A y $\boldsymbol{d}$ tienen la misma estructura que en el ejemplo 3.1 pero ahora $n = 16$.*

Utilizando las expresiones (3.5) y (3.6) podemos determinar los bloques diagonales G_i, para $i = 1, 2, \ldots, 8$, de la matriz G. Así, tendremos que

$$G_1 = \begin{bmatrix} 2 & -1 \\ -1 & 3 \end{bmatrix}, \qquad G_i = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix}, \quad \textit{para} \quad i = 2, 3 \ldots, 7 \qquad y \qquad G_8 = \begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix}.$$

Además, para $i = 1, 2, \ldots, 7$,

$$\boldsymbol{u}_i = \boldsymbol{e}_{2i} + \boldsymbol{e}_{2i+1} \qquad y \qquad \boldsymbol{v}_i = -\boldsymbol{e}_{2i} - \boldsymbol{e}_{2i+1},$$

donde $\boldsymbol{e}_{2i}$ *y* $\boldsymbol{e}_{2i+1}$ *son las columnas* $2i$*-ésima y* $(2i+1)$*-ésima de la matriz* I_{16}.

Con estos datos estamos ya en condiciones de aplicar el algoritmo 3.2.

1 ***Inicialización****.*

1.1 Resolvemos los sistemas

$$\begin{array}{llll} G_1\boldsymbol{x}_{(1:2)} = \boldsymbol{d}_{(1:2)}, & G_2\boldsymbol{x}_{(3:4)} = \boldsymbol{d}_{(3:4)}, & G_3\boldsymbol{x}_{(5:6)} = \boldsymbol{d}_{(5:6)}, & G_4\boldsymbol{x}_{(7:8)} = \boldsymbol{d}_{(7:8)}, \\ G_5\boldsymbol{x}_{(9:10)} = \boldsymbol{d}_{(9:10)}, & G_6\boldsymbol{x}_{(11:12)} = \boldsymbol{d}_{(11:12)}, & G_7\boldsymbol{x}_{(13:14)} = \boldsymbol{d}_{(13:14)}, & G_8\boldsymbol{x}_{(15:16)} = \boldsymbol{d}_{(15:16)}, \end{array}$$

cuyas soluciones son

$$\boldsymbol{x}_{(1:2)} = \begin{bmatrix} 4/5 \\ 3/5 \end{bmatrix}, \qquad \boldsymbol{x}_{(3:4)} = \boldsymbol{x}_{(5:6)} = \boldsymbol{x}_{(7:8)} = \boldsymbol{x}_{(9:10)} = \boldsymbol{x}_{(11:12)} = \boldsymbol{x}_{(13:14)} = \begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix} \qquad y \qquad \boldsymbol{x}_{(15:16)} = \begin{bmatrix} 3/5 \\ 4/5 \end{bmatrix}.$$

Nótese cómo la estructura particular de la matriz de coeficientes y del vector de términos independiente hace que muchos cálculos a lo largo del algoritmo sean repetitivos y coincidentes.

1.2 Debemos resolver ahora los sistemas

$$\begin{array}{llll} & G_2\boldsymbol{w}_{1(3:4)} = \boldsymbol{u}_{1(3:4)}, & G_3\boldsymbol{w}_{2(5:6)} = \boldsymbol{u}_{2(5:6)}, & G_4\boldsymbol{w}_{3(7:8)} = \boldsymbol{u}_{3(7:8)}, \\ G_1\boldsymbol{w}_{1(1:2)} = \boldsymbol{u}_{1(1:2)}, & G_2\boldsymbol{w}_{2(3:4)} = \boldsymbol{u}_{2(3:4)}, & G_3\boldsymbol{w}_{3(5:6)} = \boldsymbol{u}_{3(5:6)}, & G_4\boldsymbol{w}_{4(7:8)} = \boldsymbol{u}_{4(7:8)}, \end{array}$$

$$\begin{array}{llll} G_5\boldsymbol{w}_{4(9:10)} = \boldsymbol{u}_{4(9:10)}, & G_6\boldsymbol{w}_{5(11:12)} = \boldsymbol{u}_{5(11:12)}, & G_7\boldsymbol{w}_{6(13:14)} = \boldsymbol{u}_{6(13:14)}, & G_8\boldsymbol{w}_{7(15:16)} = \boldsymbol{u}_{7(15:16)}, \\ G_5\boldsymbol{w}_{5(9:10)} = \boldsymbol{u}_{5(9:10)}, & G_6\boldsymbol{w}_{6(11:12)} = \boldsymbol{u}_{6(11:12)}, & G_7\boldsymbol{w}_{7(13:14)} = \boldsymbol{u}_{7(13:14)}. & \end{array}$$

Como ya se comentó anteriormente, la estructura del sistema que estamos resolviendo provoca que muchos vectores solución de los

sistemas anteriores sean iguales. Así, es fácil comprobar que

$$\boldsymbol{w}_1(1:2) = \begin{bmatrix} 1/5 \\ 2/5 \end{bmatrix},$$

$$\boldsymbol{w}_1(3:4) = \boldsymbol{w}_2(5:6) = \boldsymbol{w}_3(7:8) = \boldsymbol{w}_4(9:10) = \boldsymbol{w}_5(11:12) = \boldsymbol{w}_6(13:14) = \begin{bmatrix} 3/8 \\ 1/8 \end{bmatrix},$$

$$\boldsymbol{w}_2(3:4) = \boldsymbol{w}_3(5:6) = \boldsymbol{w}_4(7:8) = \boldsymbol{w}_5(9:10) = \boldsymbol{w}_6(11:12) = \boldsymbol{w}_7(13:14) = \begin{bmatrix} 1/8 \\ 3/8 \end{bmatrix},$$

$$\boldsymbol{w}_7(15:16) = \begin{bmatrix} 2/5 \\ 1/5 \end{bmatrix}.$$

2 ***Actualización***.

En nuestro ejemplo, $n = 2^4$, por lo que $q = 4$. Así, existen dos pasos de actualización para $k = q - 2 = 2$ y $k = q - 3 = 1$.

Para $k = 2$, tendremos que $i = 1, 2, 3, 4$ por lo que para $i = 1$, se calculan los vectores $\boldsymbol{x}(1:4)$ y $\boldsymbol{w}_2(1:4)$ únicamente, ya que el vector $\boldsymbol{w}_0$ es nulo. Se utilizan las expresiones (3.9), (3.10) y (3.11) que aparecen en el algoritmo 3.2. Así,

$$\boldsymbol{x}(1:4) = \begin{bmatrix} \boldsymbol{x}(1:2) \\ \boldsymbol{x}(3:4) \end{bmatrix} - \frac{\boldsymbol{v}_1(1:4)^T \begin{bmatrix} \boldsymbol{x}(1:2) \\ \boldsymbol{x}(3:4) \end{bmatrix}}{1 + \boldsymbol{v}_1(1:4)^T \boldsymbol{w}_1(1:4)} \boldsymbol{w}_1(1:4) = \begin{bmatrix} 4/5 \\ 3/5 \\ 1/2 \\ 1/2 \end{bmatrix} + \frac{44}{9} \begin{bmatrix} 4/5 \\ 3/5 \\ 1/2 \\ 1/2 \end{bmatrix} = \begin{bmatrix} 16/9 \\ 23/9 \\ 7/3 \\ 10/9 \end{bmatrix},$$

$$\boldsymbol{w}_2(1:4) = \begin{bmatrix} \mathbf{0}(1:2) \\ \boldsymbol{w}_2(3:4) \end{bmatrix} - \frac{\boldsymbol{v}_1(1:4)^T \begin{bmatrix} \mathbf{0}(1:2) \\ \boldsymbol{w}_2(3:4) \end{bmatrix}}{1 + \boldsymbol{v}_1(1:4)^T \boldsymbol{w}_1(1:4)} \boldsymbol{w}_1(1:4) = \begin{bmatrix} 0 \\ 0 \\ 1/8 \\ 3/8 \end{bmatrix} + \frac{40}{72} \begin{bmatrix} 1/5 \\ 2/5 \\ 3/8 \\ 1/8 \end{bmatrix} = \begin{bmatrix} 1/9 \\ 2/9 \\ 1/3 \\ 4/9 \end{bmatrix},$$

Para $i = 2$, se calculan los vectores $\boldsymbol{x}(5:8)$, $\boldsymbol{w}_4(5:8)$ y $\boldsymbol{w}_2(5:8)$ aplicando las mismas expresiones que en el caso $i = 1$.

$$\boldsymbol{x}(5:8) = \begin{bmatrix} \boldsymbol{x}(5:6) \\ \boldsymbol{x}(7:8) \end{bmatrix} - \frac{\boldsymbol{v}_3(5:8)^T \begin{bmatrix} \boldsymbol{x}(5:6) \\ \boldsymbol{x}(7:8) \end{bmatrix}}{1 + \boldsymbol{v}_3(5:8)^T \boldsymbol{w}_3(5:8)} \boldsymbol{w}_3(5:8) = \begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{bmatrix} + 4 \begin{bmatrix} 1/8 \\ 3/8 \\ 3/8 \\ 1/8 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix},$$

$$\boldsymbol{w}_4(5:8) = \begin{bmatrix} \mathbf{0}(1:2) \\ \boldsymbol{w}_4(7:8) \end{bmatrix} - \frac{\boldsymbol{v}_3(5:8)^T \begin{bmatrix} \mathbf{0}(1:2) \\ \boldsymbol{w}_4(7:8) \end{bmatrix}}{1 + \boldsymbol{v}_3(5:8)^T \boldsymbol{w}_3(5:8)} \boldsymbol{w}_3(5:8) = \begin{bmatrix} 0 \\ 0 \\ 1/8 \\ 3/8 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1/8 \\ 3/8 \\ 3/8 \\ 1/8 \end{bmatrix} = \begin{bmatrix} 1/16 \\ 3/16 \\ 5/16 \\ 7/16 \end{bmatrix},$$

$$\boldsymbol{w}_2(5:8) = \begin{bmatrix} \boldsymbol{w}_2(5:6) \\ \mathbf{0}(1:2) \end{bmatrix} - \frac{\boldsymbol{v}_3(5:8)^T \begin{bmatrix} \boldsymbol{w}_2(5:6) \\ \mathbf{0}(1:2) \end{bmatrix}}{1 + \boldsymbol{v}_3(5:8)^T \boldsymbol{w}_3(5:8)} \boldsymbol{w}_3(5:8) = \begin{bmatrix} 3/8 \\ 1/8 \\ 0 \\ 0 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1/8 \\ 3/8 \\ 3/8 \\ 1/8 \end{bmatrix} = \begin{bmatrix} 7/16 \\ 5/16 \\ 3/16 \\ 1/16 \end{bmatrix}.$$

Para $i = 3$, se calculan los vectores $\boldsymbol{x}(9:12)$, $\boldsymbol{w}_6(9:12)$ y $\boldsymbol{w}_4(9:12)$ realizando cálculos similares a los casos anteriores.

$$\boldsymbol{x}(9:12) = \begin{bmatrix} \boldsymbol{x}(9:10) \\ \boldsymbol{x}(11:12) \end{bmatrix} - \frac{\boldsymbol{v}_5(9:12)^T \begin{bmatrix} \boldsymbol{x}(9:10) \\ \boldsymbol{x}(11:12) \end{bmatrix}}{1 + \boldsymbol{v}_5(9:12)^T \boldsymbol{w}_5(9:12)} \boldsymbol{w}_5(9:12) = \begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{bmatrix} + 4 \begin{bmatrix} 1/8 \\ 3/8 \\ 3/8 \\ 1/8 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix},$$

$$\boldsymbol{w}_6(9:12) = \begin{bmatrix} \boldsymbol{0}(1:2) \\ \boldsymbol{w}_6(11:12) \end{bmatrix} - \frac{\boldsymbol{v}_5(9:12)^T \begin{bmatrix} \boldsymbol{0}(1:2) \\ \boldsymbol{w}_6(11:12) \end{bmatrix}}{1 + \boldsymbol{v}_5(9:12)^T \boldsymbol{w}_5(9:12)} \boldsymbol{w}_5(9:12) = \begin{bmatrix} 0 \\ 0 \\ 1/8 \\ 3/8 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1/8 \\ 3/8 \\ 3/8 \\ 1/8 \end{bmatrix} = \begin{bmatrix} 1/16 \\ 3/16 \\ 5/16 \\ 7/16 \end{bmatrix},$$

$$\boldsymbol{w}_4(9:12) = \begin{bmatrix} \boldsymbol{w}_4(9:10) \\ \boldsymbol{0}(1:2) \end{bmatrix} - \frac{\boldsymbol{v}_5(9:12)^T \begin{bmatrix} \boldsymbol{w}_4(9:10) \\ \boldsymbol{0}(1:2) \end{bmatrix}}{1 + \boldsymbol{v}_5(9:12)^T \boldsymbol{w}_5(9:12)} \boldsymbol{w}_5(9:12) = \begin{bmatrix} 3/8 \\ 1/8 \\ 0 \\ 0 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1/8 \\ 3/8 \\ 3/8 \\ 1/8 \end{bmatrix} = \begin{bmatrix} 7/16 \\ 5/16 \\ 3/16 \\ 1/16 \end{bmatrix}.$$

Finalmente, para $i = 4$, se calculan los vectores $\boldsymbol{x}(13:16)$ y $\boldsymbol{w}_6(13:16)$ de forma análoga a las anteriores.

$$\boldsymbol{x}(13:16) = \begin{bmatrix} \boldsymbol{x}(13:14) \\ \boldsymbol{x}(15:16) \end{bmatrix} - \frac{\boldsymbol{v}_7(13:16)^T \begin{bmatrix} \boldsymbol{x}(13:14) \\ \boldsymbol{x}(15:16) \end{bmatrix}}{1 + \boldsymbol{v}_7(13:16)^T \boldsymbol{w}_7(13:16)} \boldsymbol{w}_7(13:16) = \begin{bmatrix} 1/2 \\ 1/2 \\ 3/5 \\ 4/5 \end{bmatrix} + \frac{44}{9} \begin{bmatrix} 1/8 \\ 3/8 \\ 2/5 \\ 1/5 \end{bmatrix} = \begin{bmatrix} 10/9 \\ 7/3 \\ 23/9 \\ 16/9 \end{bmatrix},$$

$$\boldsymbol{w}_6(13:16) = \begin{bmatrix} \boldsymbol{w}_6(13:14) \\ \mathbf{0}(1:2) \end{bmatrix} - \frac{\boldsymbol{v}_7(13:16)^T \begin{bmatrix} \boldsymbol{w}_6(13:14) \\ \mathbf{0}(1:2) \end{bmatrix}}{1 + \boldsymbol{v}_7(13:16)^T \boldsymbol{w}_7(13:16)} \boldsymbol{w}_7(13:16) = \begin{bmatrix} 3/8 \\ 1/8 \\ 0 \\ 0 \end{bmatrix} + \frac{40}{72} \begin{bmatrix} 1/8 \\ 3/8 \\ 2/5 \\ 1/5 \end{bmatrix} = \begin{bmatrix} 4/9 \\ 1/3 \\ 2/9 \\ 1/9 \end{bmatrix}.$$

Una vez cumplida la actualización para $k = 2$, procedemos a la actualización para $k = 1$, por lo que $i = 1, 2$. Para $i = 1$ se calculan los vectores $\boldsymbol{x}(1:8)$ y $\boldsymbol{w}_4(1:8)$ utilizando de nuevo las expresiones (3.9), (3.10) y (3.11).

$$\boldsymbol{x}(1:8) = \begin{bmatrix} \boldsymbol{x}(1:4) \\ \boldsymbol{x}(5:8) \end{bmatrix} - \frac{\boldsymbol{v}_2(1:8)^T \begin{bmatrix} \boldsymbol{x}(1:4) \\ \boldsymbol{x}(5:8) \end{bmatrix}}{1 + \boldsymbol{v}_2(1:8)^T \boldsymbol{w}_2(1:8)} \boldsymbol{w}_2(1:8) = \begin{bmatrix} 64/17 & 111/17 & 141/17 & 154/17 & 150/17 & 129/17 & 91/17 & 36/17 \end{bmatrix}^T,$$

$$\boldsymbol{w}_4(1:8) = \begin{bmatrix} \mathbf{0}(1:4) \\ \boldsymbol{w}_4(5:8) \end{bmatrix} - \frac{\boldsymbol{v}_2(1:8)^T \begin{bmatrix} \mathbf{0}(1:4) \\ \boldsymbol{w}_4(5:8) \end{bmatrix}}{1 + \boldsymbol{v}_2(1:8)^T \boldsymbol{w}_2(1:8)} \boldsymbol{w}_2(1:8) = \begin{bmatrix} 1/17 & 2/17 & 3/17 & 4/17 & 5/17 & 6/17 & 7/17 & 8/17 \end{bmatrix}^T.$$

Para $i = 2$ se calculan, como en el caso anterior, los vectores $\boldsymbol{x}(9:16)$ y $\boldsymbol{w}_4(9:16)$.

$$\boldsymbol{x}(9:16) = \begin{bmatrix} \boldsymbol{x}(9:12) \\ \boldsymbol{x}(13:16) \end{bmatrix} - \frac{\boldsymbol{v}_6(9:16)^T \begin{bmatrix} \boldsymbol{x}(9:12) \\ \boldsymbol{x}(13:16) \end{bmatrix}}{1 + \boldsymbol{v}_6(9:16)^T \boldsymbol{w}_6(9:16)} \boldsymbol{w}_6(9:16) = \begin{bmatrix} 36/17 & 91/17 & 129/17 & 150/17 & 154/17 & 141/17 & 111/17 & 64/17 \end{bmatrix}^T,$$

$$\boldsymbol{w}_4(9:16) = \begin{bmatrix} \boldsymbol{w}_4(9:12) \\ \mathbf{0}(1:4) \end{bmatrix} - \frac{\boldsymbol{v}_6(9:16)^T \begin{bmatrix} \boldsymbol{w}_4(9:12) \\ \mathbf{0}(1:4) \end{bmatrix}}{1 + \boldsymbol{v}_6(9:16)^T \boldsymbol{w}_6(9:16)} \boldsymbol{w}_6(9:16) = \begin{bmatrix} 8/17 & 7/17 & 6/17 & 5/17 & 4/17 & 3/17 & 2/17 & 1/17 \end{bmatrix}^T.$$

3 ***Obtención de la solución.***

Una vez terminada la fase de actualización, ya podemos calcular el vector $\boldsymbol{x}_{(1:16)}$ *mediante la expresión (3.12), que ya es la solución del sistema inicial.*

$$\boldsymbol{x}_{(1:16)} = \begin{bmatrix} \boldsymbol{x}_{(1:8)} \\ \boldsymbol{x}_{(9:16)} \end{bmatrix} - \frac{\boldsymbol{v}_{4(1:16)}{}^T \begin{bmatrix} \boldsymbol{x}_{(1:8)} \\ \boldsymbol{x}_{(9:16)} \end{bmatrix}}{1 + \boldsymbol{v}_{4(1:16)}{}^T \boldsymbol{w}_{4(1:16)}} \boldsymbol{w}_{4(1:16)} = \begin{bmatrix} 7/2 & 6 & 15/2 & 8 & 15/2 & 6 & 7/2 & 0 & 0 & 7/2 & 6 & 15/2 & 8 & 15/2 & 6 & 7/2 \end{bmatrix}^T.$$

Como se desprende del ejemplo 3.2, en la fase de inicialización, se resuelven una serie de sistemas de tamaño 2×2. Estos sistemas se utilizan en la fase de actualización para actualizar los bloques de los vetores $\boldsymbol{x}_i$ y $\boldsymbol{w}_i$ que van duplicando el número de sus componentes y que nos permiten en la última fase, calcular la solución general.

En la figura 3.1 se visualizan las componentes de los vectores $\boldsymbol{w}_i$, para $i = 1, 2, \ldots, 7$, involucradas en la inicialización y actualización del algoritmo 3.2 para el ejemplo 3.2. En dicha figura no aparecen las componentes que se actualizan del vector solución $\boldsymbol{x}$ debido a que su actualización no ofrece ninguna dificultad de tipo conceptual. En la fase de inicialización, se calculan todas las componentes del vector $\boldsymbol{x}$ por bloques de tamaño dos, resolviendo sistemas de tamaño 2×2. En los pasos siguientes de actualización se duplica cada vez el tamaño de los bloques que se actualizan hasta llegar a la obtención de la solución final de acuerdo con la expresión (3.12).

En la fase de inicialización observamos que todos los sistemas que se resuelven son de tamaño 2×2. Como se observa en la figura 3.1, el resultado de esta fase es que hemos calculado cuatro componentes de cada uno de los vectores $\boldsymbol{w}_i$, para $i = 1, 2, \ldots, 7$. Esto significa que estos vectores tienen nulas todas sus componentes salvo a lo sumo estas cuatro, que son las que se han actualizado.

En el primer paso de la fase de actualización, es decir, para $k = 2$, combinamos los resultados de la etapa de inicialización para actualizar bloques de cuatro componentes de los vectores $\boldsymbol{w}_2$, $\boldsymbol{w}_4$ y $\boldsymbol{w}_6$, de manera que después de dicha fase de actualización cada uno de estos vectores tiene nulas todas sus componentes salvo a lo sumo estas ocho componentes que se han actualizado. La figura 3.1 nos muestra las componentes de la fase 1 que son necesarias en cada una de las actualizaciones que se llevan a cabo en cada fase. Se utilizan

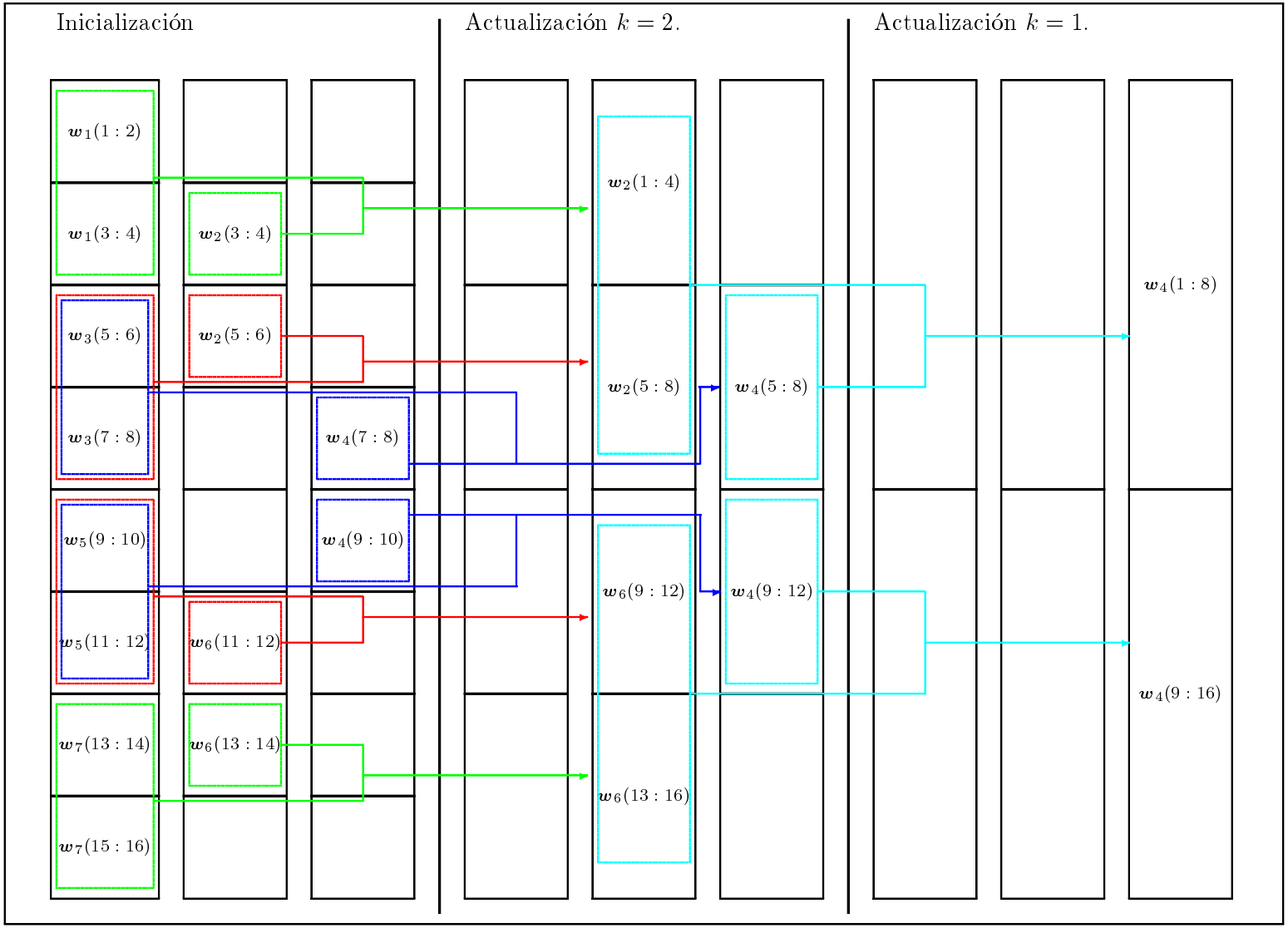


Figura 3.1: *Inicialización y actualización del algoritmo 3.2 para el ejemplo 3.2.*

colores distintos para visualizar de una forma más clara qué componentes son necesarias en cada caso. Así, por ejemplo se observa que para calcular la actualización del vector $\boldsymbol{w}_2(1:4)$ utilizamos las componentes actualizadas en la fase anterior de los vectores $\boldsymbol{w}_2$ y $\boldsymbol{w}_1$, calculadas previamente, es decir, $\boldsymbol{w}_2(3:4)$ y $\boldsymbol{w}_1(1:4)$.

Para $k = 1$ únicamente se actualizan las componentes del vector $\boldsymbol{w}_4$ en dos bloques de ocho componentes. Notemos que ahora para actualizar el bloque $\boldsymbol{w}_4(1:8)$ se utilizan los bloques calculados previamente $\boldsymbol{w}_2(1:8)$ y $\boldsymbol{w}_4(5:8)$, como se aprecia en la figura 3.1. El resultado de esta actualización nos permite obtener la solución final. A lo largo de este proceso queda de manifiesto una característica esencial de esta implementación: únicamente actualizamos bloques de vectores a partir de bloques de vectores no nulos calculados previamente a través de un proceso recursivo.

En la figura 3.1 se pone de manifiesto también que, debido a las características del método, los cálculos que se realizan en las distintas fases conducentes a la obtención del vector $\boldsymbol{w}_4$ pueden desarrollarse de forma independiente unos de otros. Si observamos detenidamente las relaciones gráficas que se establecen en la figura 3.1 entre las distintas componentes, podemos llegar a la conclusión de que el cálculo de las componentes $\boldsymbol{w}_4(1:8)$ que se realiza en el paso de actualización para $k = 1$ es independiente del cálculo de las componentes $\boldsymbol{w}_2(9:16)$. Esto nos lleva a que si consideramos que disponemos de dos procesadores, P_0 y P_1, cada uno de ellos puede desarrollar perfectamente en paralelo los cálculos conducentes a la obtención de $\boldsymbol{w}_4(1:8)$ y $\boldsymbol{w}_4(1:16)$, respectivamente. Para la obtención de la solución final ya es necesaria una comunicación entre procesadores ya que estos dos vectores deben encontrarse en el mismo procesador para calcular $\boldsymbol{x}(1:16)$ mediante la expresión (3.12). Notemos que si en lugar de disponer de dos procesadores dispusiéramos de cuatro procesadores, la comunicación de elementos sería necesaria antes de calcular $\boldsymbol{w}_4(1:8)$, ya que el bloque $\boldsymbol{w}_2(1:4)$ estaría en el procesador P_0 y los bloques $\boldsymbol{w}_2(5:8)$ y $\boldsymbol{w}_4(5:8)$ se encontrarían en el procesador P_1, por lo que su cálculo requiere una comunicación previa.

Dedicamos el resto de la sección al estudio del coste computacional del algoritmo 3.2. Para ello calculamos los costes de cada una de las tres fases que componen dicho algoritmo.

Comenzamos analizando el coste de la fase 1. Sabemos que, para $i = 1, 2, \ldots, 2^{q-1}$, las matrices G_i son de tamaño 2×2, por lo que para calcular los elementos de su diagonal mediante las expresiones (3.5) y (3.6) son necesarios $2(2^{q-1} - 1)$ flops. Ahora, la resolución de

los sistemas involucrados supone un total de $7 \cdot 2^{q-1} + 6$ flops. Por lo tanto, el coste total de esta fase es de

$$2(2^{q-1} - 1) + 7 \cdot 2^{q-1} + 6 = 9 \cdot 2^{q-1} + 4 \quad \text{flops.} \tag{3.14}$$

En cuanto al coste computacional de la fase de actualización, éste puede dividirse en varias partes. Calculamos primeramente el coste de cada una de las operaciones que se realizan en cada paso de la actualización.

- El cálculo del denominador de las expresiones (3.9), (3.10) y (3.11) requiere 4 flops. El cálculo del numerador de la expresión (3.9) únicamente requiere 3 flops, mientras que el cálculo de los numeradores de las expresiones (3.10) y (3.11) requieren 1 flop respectivamente.
- El cálculo del vector $\boldsymbol{x}_{(2^{q-k}(i-1)+1\,:\,2^{q-k}i)}$ supone $1 + 2^{q-k+1}$ flops.
- Para calcular los vectores $\boldsymbol{w}_{2^{q-1-k}i(2^{q-k}(i-1)+1\,:\,2^{q-k}i)}$ y $\boldsymbol{w}_{2^{q-1-k}(i-1)(2^{q-k}(i-1)+1\,:\,2^{q-k}i)}$ necesitamos $2(1 + 3 \cdot 2^{q-k-1})$ flops.

Ahora, para calcular el coste total, sumamos los costes de cada una de las etapas que se realizan en esta fase de actualización, obteniendo

$$\sum_{k=1}^{q-2} \sum_{i=1}^{2^k} \left[10 + 2^{q-k+1} + 2(1 + 3 \cdot 2^{q-k-1})\right] = 12(2^{q-1} - 2) + 5 \cdot 2^q (q-2) \quad \text{flops.} \tag{3.15}$$

Finalmente, el coste computacional de la obtención del vector solución utilizando la expresión (3.12) es de

$$8 + 2^{q+1} \quad \text{flops.} \tag{3.16}$$

Consecuentemente, si sumamos las expresiones (3.14), (3.15) y (3.16) obtenemos el coste computacional del algoritmo 3.2, que es de

$$(5 + 10q) \cdot 2^{q-1} - 20 \quad \text{flops.} \tag{3.17}$$

3.3 Un algoritmo BSP de tipo *fan-in*

3.3.1 Descripción del algoritmo

Supondremos en adelante que el número de procesadores disponibles es $p = 2^m$, para un cierto entero $m \geq 1$, ya que si $m = 0$ tendremos el algoritmo secuencial. Las características del método expuestas en la sección 3.2 sugieren la implementación del mismo utilizando un algoritmo paralelo del tipo *fan-in*. Se describen a continuación las características esenciales del proceso de cálculo y de comunicación que requiere un algoritmo de este tipo.

Inicialmente, la responsabilidad de la realización de las operaciones de la fase 1 del algoritmo 3.2 se divide entre todos los procesadores. Después de estos cálculos, cada procesador P_j, para $j = 0, 1, \ldots, 2^m - 1$, es responsable de la actualización de los bloques de vectores que se especifican en la fase 2 del algoritmo 3.2. Al final de esta fase, se obtienen los vectores

$$\boldsymbol{x}_{(2^{q-m}j+1\,:\,2^{q-m}j+2^{q-m}i)}, \quad \boldsymbol{w}_{2^{q-1-m}(j+1)(2^{q-m}j+1\,:\,2^{q-m}j+2^{q-m}i)} \quad \text{y} \quad \boldsymbol{w}_{2^{q-1-m}(j)(2^{q-m}j+1\,:\,2^{q-m}j+2^{q-m}i)}. \tag{3.18}$$

Nótese que en todo el proceso de cálculo conducente a la obtención de los vectores (3.18) no es necesario realizar ninguna comunicación de elementos entre los procesadores, debido a que cada procesador tiene los elementos que necesita para efectuar los cálculos. En total, se llevan a cabo $q - m$ pasos de actualización, en los que cada procesador calcula sucesivamente bloques de vectores utilizando las expresiones (3.9), (3.10) y (3.11) de forma recursiva.

Consideramos inicialmente el caso en que $m = 1$, es decir, cuando $p = 2$. En este caso, cada procesador desarrolla los primeros $q - 1$ pasos de la fase 2 de forma independiente. Posteriormente, el procesador P_1 comunica al procesador P_0 los bloques de vectores que permiten al procesador P_0 obtener la solución final. En consecuencia, únicamente se produce una comunicación de datos entre los dos procesadores, por lo que el modelo de comunicación que utilicemos es indiferente ya que disponemos únicamente de dos procesadores. Así, el siguiente algoritmo resume las características expuestas anteriormente para el caso en que $m = 1$.

Algoritmo 3.3 Un algoritmo BSP de tipo *fan-in* para sistemas tridiagonales cuando $p = 2$.

Superpaso 1

El procesador P_0 envía al procesador P_1 los elementos $a_{2^{q-1}j+i}$, $b_{2^{q-1}j+i-1}$ y $c_{2^{q-1}j+i+1}$, para $i = 1, 2, \ldots, 2^{q-1}$, suponiendo que $c_{n+1} = 0$.

Superpaso 2

1 En el procesador P_j, para $j = 0, 1$,

1.1 calcular $\hat{a}_{2^{q-1}j+i}$, para $i = 1, 2, \ldots, 2^{q-1}$, utilizando las expresiones (3.5) y (3.6),

1.2 calcular $\boldsymbol{x}_{(2^{q-1}j+2i-1\,:\,2^{q-1}j+2i)}$, para $i = 1, 2, \ldots, 2^{q-2}$ utilizando la expresión (3.7),

1.3 calcular $\boldsymbol{w}_{2^{q-2}j+i(2^{q-1}j+2i-1\,:\,2^{q-1}j+2i)}$ y $\boldsymbol{w}_{2^{q-2}j+i-1(2^{q-1}j+2i-1\,:\,2^{q-1}j+2i)}$, para $i = 1, 2, \ldots, 2^{q-2}$, utilizando las expresiones (3.8) y suponiendo que $\boldsymbol{w}_0$ y $\boldsymbol{w}_{2^q}$ son vectores nulos,

1.4 para $k = q-2, q-3, \ldots, 1$, y para $i = 1, 2, \ldots, 2^{k-1}$ calcular

1.4.1 $\boldsymbol{x}_{(2^{q-1}j+2^{q-k}(i-1)+1\,:\,2^{q-1}j+2^{q-k}i)}$ utilizando la expresión (3.9),

1.4.2 $\boldsymbol{w}_{2^{q-2}j+2^{q-k-1}i(2^{q-1}j+2^{q-1}(i-1)+1\,:\,2^{q-1}j+2^{q-k}i)}$ utilizando la expresión (3.10),

1.4.3 $\boldsymbol{w}_{2^{q-2}j+2^{q-k-1}(i-1)(2^{q-1}j+2^{q-1}(i-1)+1\,:\,2^{q-1}j+2^{q-k}i)}$ utilizando la expresión (3.11).

2 El procesador P_1 envía al procesador P_0 los bloques de vectores $\boldsymbol{x}_{(2^{q-1}j+1\,:\,2^{q-1}j+2^{q-1})}$, $\boldsymbol{w}_{2^{q-2}j+2^{q-2}(2^{q-1}j+1\,:\,2^{q-1}j+2^{q-1})}$ y $\boldsymbol{w}_{2^{q-2}j(2^{q-1}j+1\,:\,2^{q-1}j+2^{q-1})}$.

Superpaso 3

El procesador P_0 calcula $\boldsymbol{x}(1 : 2^q)$ utilizando la expresión (3.12).

Consideramos ahora el caso general en que $m \geq 2$. Cada procesador inicialmente desarrolla los primeros $q - m$ pasos de la fase 2 de forma independiente. En los siguientes m pasos, los procesadores deben comunicar sus bloques de vectores actualizados a otros

procesadores, que utilizan estos resultados para actualizar bloques de vectores de tamaño cada vez mayor, siguiendo un esquema de comunicación de tipo *fan-in*. En dicho esquema de comunicación, los procesadores pueden dividirse en dos categorías: activo, cuando el procesador realiza cálculos e inactivo, cuando no realiza cálculos.

Al principio del proceso, la matriz A del sistema se distribuye entre todos los procesadores. En los pasos $k = q-1, q-2, \ldots, m$, todos los procesadores trabajan al mismo tiempo de forma independiente. Al final del paso m de actualización, cada procesador ha calculado bloques de vectores de 2^{q-m} componentes. Para poder seguir el proceso de actualización, se hace necesario mezclar los resultados obtenidos por procesadores distintos. Así, cada procesador impar, por ejemplo P_d, envía sus bloques de vectores a su procesador vecino P_{d-1}. El procesador P_{d-1} utiliza estos bloques, mezclándolos y calculando nuevos bloques de vectores con doble número de componentes en el paso $m+1$. En este paso, los procesadores pares se han convertido en procesadores activos, ya que han recibido bloques de vectores de otros procesadores mientras que los procesadores impares se han convertido en procesadores inactivos. Al final del paso $m+1$, se hace necesario mezclar de nuevo los resultados obtenidos por dos procesadores activos. En el paso $m-2$ cada procesador cuyo número es múltiplo de 4 se convierte en procesador activo, mientras que el resto de procesadores permanecen inactivos. Este proceso continúa de forma recursiva hasta que en el último paso el procesador P_0 es el único que permanece activo, estando inactivos los restantes.

El siguiente algoritmo BSP resume las características de este método siguiendo un esquema de comunicaciones *fan-in* cuando el número de procesadores es mayor que dos.

Algoritmo 3.4 Un algoritmo BSP de tipo *fan-in* para sistemas tridiagonales cuando $p > 2$.

Superpaso 1

El procesador P_0 envía al procesador P_j, para $j = 1, 2, \ldots, 2^m - 1$, los elementos $a_{2^{q-m}j+i}$, $b_{2^{q-m}j+i-1}$ y $c_{2^{q-m}j+i+1}$, para $i = 1, 2, \ldots, 2^{q-m}$, suponiendo que $c_{n+1} = 0$.

Superpaso 2

1 En el procesador P_j, para $j = 0, 1, \ldots, 2^m - 1$,

1.1 calcular $\hat{a}_{2^{q-m}j+i}$, para $i = 1, 2, \ldots, 2^{q-m}$, utilizando las expresiones (3.5) y (3.6).

1.2 calcular $\boldsymbol{x}(2^{q-m}j + 2i - 1 : 2^{q-m}j + 2i)$, para $i = 1, 2, \ldots, 2^{q-m-1}$ utilizando la expresión (3.7).

1.3 calcular $\boldsymbol{w}_{2^{q-m-1}j+i}(2^{q-m}j + 2i - 1 : 2^{q-m}j + 2i)$ y $\boldsymbol{w}_{2^{q-m-1}j+i-1}(2^{q-m}j + 2i - 1 : 2^{q-m}j + 2i)$, para $i = 1, 2, \ldots, 2^{q-m-1}$, utilizando las expresiones (3.8) y suponiendo que $\boldsymbol{w}_0$ y $\boldsymbol{w}_{2^q}$ son vectores nulos.

1.4 para $k = q-2, q-3, \ldots, m$, y para $i = 1, 2, \ldots, 2^{k-m}$ calcular

1.4.1 $\boldsymbol{x}(2^{q-m}j + 2^{q-k}(i-1) + 1 : 2^{q-m}j + 2^{q-k}i)$ utilizando la expresión (3.9),

1.4.2 $\boldsymbol{w}_{2^{q-m-1}j+2^{q-k-1}i}(2^{q-m}j + 2^{q-m}(i-1) + 1 : 2^{q-m}j + 2^{q-k}i)$ utilizando la expresión (3.10),

1.4.3 $\boldsymbol{w}_{2^{q-m-1}j+2^{q-k-1}(i-1)}(2^{q-m}j + 2^{q-m}(i-1) + 1 : 2^{q-m}j + 2^{q-k}i)$ utilizando la expresión (3.11).

2 El procesador P_{2j+1}, con $j = 0, 1, \ldots, 2^{m-1} - 1$, envía al procesador P_{2j} los bloques de vectores

2.1 $\boldsymbol{x}(2^{q-m}(2j+1) + 1 : 2^{q-m}(2j+1) + 2^{q-m})$,

2.2 $\boldsymbol{w}_{2^{q-m-1}(2j+1)+2^{q-m-1}}(2^{q-m}(2j+1) + 1 : 2^{q-m}(2j+1) + 2^{q-m})$,

2.3 $\boldsymbol{w}_{2^{q-m-1}(2j+1)}(2^{q-m}(2j+1) + 1 : 2^{q-m}(2j+1) + 2^{q-m})$.

Superpaso $m - k + 2$, para $k = m-1, m-2, \ldots, 2, 1$

1 El procesador $P_{2^{m-k}i}$, para $i = 0, 1, \ldots, 2^k - 1$ calcula

1.1 $\boldsymbol{x}(2^{m-k+2}i + 1 : 2^{m-k+2}(i+1))$, utilizando la expresión (3.9),

1.2 $\boldsymbol{w}_{2^{m-k+1}(i+1)}(2^{m-k+2}i + 1 : 2^{m-k+2}(i+1))$, utilizando la expresión (3.10),

1.3 $\boldsymbol{w}_{2^{m-k+1}i}(2^{m-k+2}i + 1 : 2^{m-k+2}(i+1))$, utilizando la expresión (3.11).

2 El procesador $P_{2^{m-k}+2^{m-k+1}i}$, para $i = 0, 1, \ldots, \lfloor \frac{2^k-1}{2} \rfloor$, comunica al procesador $P_{2^{m-k+1}i}$ los bloques de vectores $\boldsymbol{x}(2^{m-k+2}(2i+1) + 1 : 2^{m-k+3}(i+1))$, $\boldsymbol{w}_{2^{m-k+2}(i+1)}(2^{m-k+2}(2i+1) + 1 : 2^{m-k+3}(i+1))$ y $\boldsymbol{w}_{2^{m-k+1}(2i+1)}(2^{m-k+2}(2i+1) + 1 : 2^{m-k+3}(i+1))$.

Superpaso $m + 2$

El procesador P_0 calcula $\boldsymbol{x}(1 : 2^q)$ utilizando la expresión (3.12).

La diferencia entre este algoritmo y el algoritmo 3.3 está en que ahora es necesario introducir el superpaso $m - k + 2$, para $k = m - 1, m - 2, \ldots, 2, 1$, en el que se produce la comunicación de bloques de vectores entre los procesadores activos y la actualización y cálculo de las nuevas componentes de estos vectores.

Con el fin de analizar detalladamente la estructura que presenta el algoritmo 3.4, analizamos los cálculos aritméticos y los esquemas de comunicación que se requieren para resolver un ejemplo concreto, donde A es una matriz tridiagonal de tamaño 32×32. En este caso, $n = 2^5$, por lo que $q = 5$. Además, supondremos que el número de procesadores es $p = 2^2$, en consecuencia, $m = 2$. En estas condiciones,

$$A = \begin{bmatrix} G_1 & & & & \\ & G_2 & & & \\ & & \ddots & & \\ & & & G_{15} & \\ & & & & G_{16} \end{bmatrix} + \boldsymbol{u}_1\boldsymbol{v}_1^T + \boldsymbol{u}_2\boldsymbol{v}_2^T + \cdots + \boldsymbol{u}_{15}\boldsymbol{v}_{15}^T.$$

A lo largo de todo el proceso tendremos que calcular y actualizar los bloques de vectores que denotamos por $\boldsymbol{x}$ y $\boldsymbol{w}_i$, para $i = 1, 2, \ldots, 15$, utilizando las expresiones (3.9), (3.10) y (3.11) tal como se indica en el algoritmo 3.2.

La figura 3.2 recoge las actualizaciones y las comunicaciones que se realizan en cada procesador hasta obtener la solución final cuando se ejecuta el algoritmo 3.4 para este caso concreto.

Como se desprende de la figura 3.2, el algoritmo 3.4 presenta dos partes claramente diferenciadas: en la primera de ellas se producen todos los cálculos aritméticos de forma independiente en cada procesador sin realizar ningún tipo de comunicación entre los mismos. En la segunda parte se desarrolla un esquema de comunicaciones de tipo *fan-in*, conducente a la obtención de la solución final en el procesador principal. En nuestro ejemplo, la primera parte de cálculo aritmético del algoritmo constaría de la fase 1 y los primeros dos pasos de la fase 2 de actualización de los vectores. Después, es necesaria una comunicación de elementos desde los procesadores impares a los procesadores pares, que son los únicos activos en el siguiente paso. Los procesadores pares calculan cada uno un bloque de 16

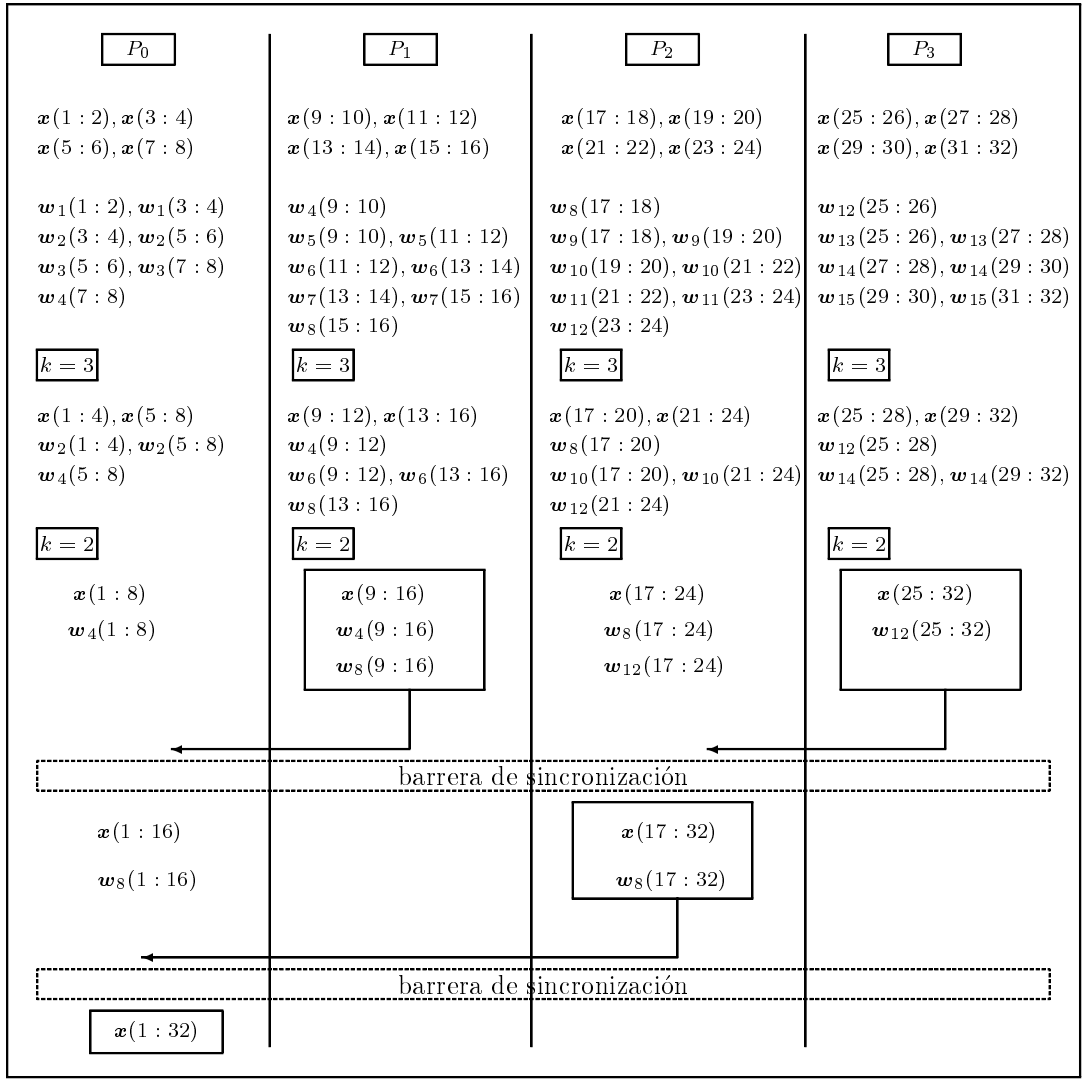


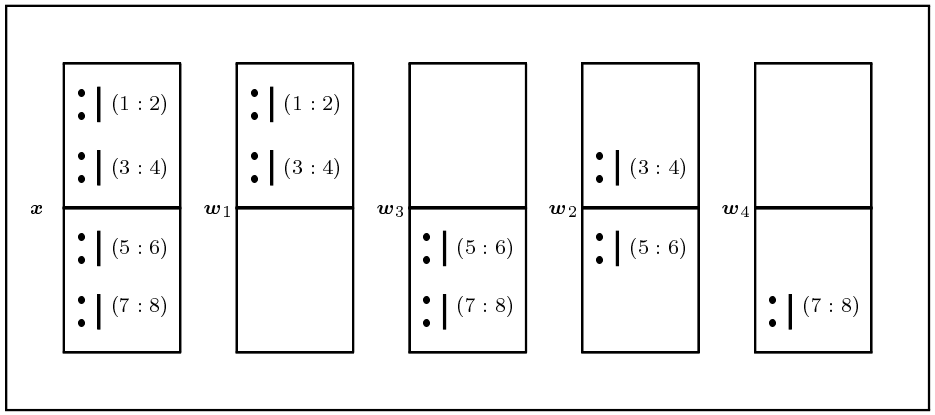
Figura 3.2: *Esquema del algoritmo 3.4 para* $n = 32$ *y* $p = 4$.

componentes del vector $\boldsymbol{x}$ y $\boldsymbol{w_8}$ y el procesador P_2 comunica al procesador P_0 sus bloques actualizados para que el procesador principal calcule la solución utilizando la expresión (3.12). Para $p = 4$, basta con realizar dos barreras de sincronización, ya que después de la segunda comunicación ya se puede obtener la solución final en el procesador principal.

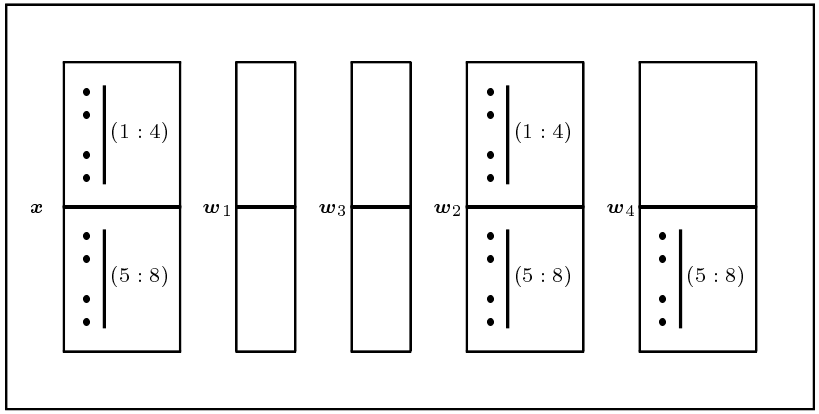
Se observa también que el proceso no es totalmente equilibrado en el sentido que los procesadores centrales tienen una carga computacional mayor. Es interesante destacar que, cuando el número de procesadores es bajo, la mayor parte de los cálculos aritméticos se realizan antes de que sea necesaria una comunicación entre procesadores; el tamaño de los bloques de vectores que se envían en la primera comunicación es $\frac{n}{p}$. En la siguiente comunicación el número de componentes del bloque se va duplicando hasta llegar a la última comunicación, en la que se envían dos bloques de $\frac{n}{2}$ componentes desde el procesador $P_{\frac{n}{2}}$ al procesador P_0.

La figura 3.3 nos muestra los cálculos y actualizaciones que lleva a cabo el procesador P_0 para el caso $n = 32$ que estamos considerando. En dicha figura se ponen de manifiesto todas las actualizaciones que realiza P_0 hasta que es necesaria una comunicación de elementos con otros procesadores. Por tanto, se detalla únicamente el trabajo computacional que realiza dicho procesador en el superpaso 2 del algoritmo 3.4 para este ejemplo. Notemos que el tamaño máximo del bloque que puede actualizar es de $2^{q-m} = 8$. Como se aprecia en la figura 3.3(a), los primeros cálculos se realizan con bloques de dos componentes; posteriormente, la figura 3.3(b) nos muestra cómo las actualizaciones son de bloques de cuatro componentes, para acabar en la figura 3.3(c) actualizando las ocho componentes de los vectores $\boldsymbol{x}$ y $\boldsymbol{w}_8$.

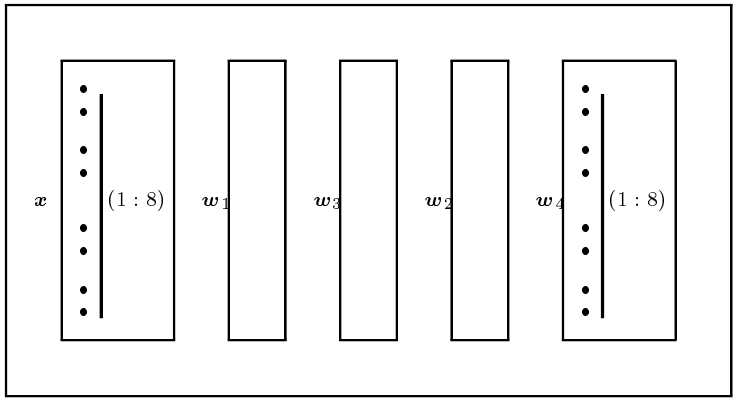
La figura 3.3 pone de manifiesto, de nuevo, la característica fundamental de esta implementación en la que no se tienen en cuenta los bloques nulos de los vectores que se actualizan en cada paso. Así, por ejemplo, para el cálculo del bloque $\boldsymbol{w}_4(1:8)$ que aparece en la figura 3.3(c) utilizamos los bloques $\boldsymbol{w}_4(5:8)$, $\boldsymbol{w}_2(1:4)$ y $\boldsymbol{w}_2(5:8)$ calculados previamente, no siendo necesarias el resto de sus componentes para los cálculos que realiza este procesador.



(a) *Inicialización*



(b) *Actualización,* $k = 2$



(c) *Actualización,* $k = 1$

Figura 3.3: *Inicialización y actualizaciones que realiza el procesador* P_0 *cuando ejecuta el algoritmo 3.4 para el ejemplo 3.2.*

3.3.2 Coste computacional

En esta sección calculamos el coste computacional de los algoritmos 3.3 y 3.4 estudiando el coste de cada uno de los superpasos que los componen. Comenzamos calculando el coste del algoritmo 3.3.

Coste del superpaso 1. En este superpaso no se realiza ninguna operación aritmética, por lo que el coste computacional viene dado por el coste de comunicación. El procesador principal envía $4n - 4\frac{n}{p}$ elementos de la matriz A y del vector $\boldsymbol{d}$ al procesador P_1, quien también recibe un elemento situado fuera de los bloques tridiagonales de A. El último procesador únicamente recibe un elemento situado fuera de los bloques tridiagonales. Por lo tanto, el coste de comunicación es de $(2n+1)g$ flops, por lo que el coste total de este superpaso es de

$$(2n+1)g + l \quad \text{flops.} \tag{3.19}$$

Coste del superpaso 2. Comenzamos calculando el coste aritmético. Para calcular los elementos diagonales de las matrices G_i, utilizando las expresiones (3.5) y (3.6) se necesitan 2^{q-1} flops; es decir, $\dfrac{n}{2}$ flops. Para resolver los sistemas de los apartados 1.2 y 1.3 del superpaso 2 se necesitan $7 \cdot 2^{q-2}$ flops en cada procesador. En consecuencia, el coste aritmético de realizar los apartados 1.1, 1.2 y 1.3 del superpaso 2 es de

$$9 \cdot 2^{q-2} = \frac{9}{4}n \quad \text{flops.} \tag{3.20}$$

Debido a la estructura particular de los vectores que aparecen en el apartado 1.4.3 del superpaso 2, son necesarios

$$10 + 2^{q-k+1} + 2(1 + 3 \cdot 2^{q-k+1}) = 12 + 5 \cdot 2^{q-k} \quad \text{flops}$$

para calcular los bloques de vectores $\boldsymbol{x}$, $\boldsymbol{w}_{2^{q-1-k}i}$ y $\boldsymbol{w}_{2^{q-1-k}(i-1)}$.

Como el número de bloques de vectores que se actualizan para cada procesador en esta etapa es 2^{k-1}, tenemos que el coste aritmético del apartado 1.4.3 del algoritmo es

$$2^{k-1}\left(12 + 5 \cdot 2^{q-k}\right) \quad \text{flops.} \tag{3.21}$$

Sumando las expresiones (3.20) y (3.21), se obtiene el coste aritmético del superpaso 2, que es de

$$\frac{9}{4}n + \sum_{k=1}^{q-2} 2^{k-1}\left[12 + 5 \cdot 2^{q-k}\right] = \left[5(q-1) + \frac{11}{2}\right]\frac{n}{2} - 12 \quad \text{flops.} \tag{3.22}$$

Para obtener el coste de comunicación de este superpaso, hay que tener en cuenta que el último procesador envía sólamente dos vectores de tamaño 2^{q-1} a su procesador par contiguo. Así, el coste de comunicación corresponde al de una $2 \cdot 2^{q-1}$-relación, es decir, ng flops.

Sumando el coste aritmético y de comunicación en este superpaso y efectuando las oportunas simplificaciones, se obtiene un coste computacional para el superpaso 2 de

$$\left[5(q-1) + \frac{11}{2}\right]\frac{n}{2} - 12 + ng + l \quad \text{flops.} \tag{3.23}$$

Coste del superpaso 3. En este superpaso, únicamente el procesador principal realiza operaciones aritméticas, ya que es el encargado de calcular el vector solución $\boldsymbol{x}(1:2^q)$ y no realiza ninguna comunicación, por lo que el coste computacional es de

$$8 + 2n + l \quad \text{flops.} \tag{3.24}$$

Finalmente, sumando las expresiones (3.19), (3.23) y (3.24) y realizando las simplificaciones oportunas, se obtiene que el coste computacional total del algoritmo 3.3 es de

$$\left[\frac{5}{2}q + \frac{9}{4}\right]n - 4 + (3n+1)g + 3l \quad \text{flops.} \tag{3.25}$$

Una vez calculado el coste del algoritmo 3.3 procedemos a calcular el coste del algoritmo 3.4 de forma análoga.

Coste del superpaso 1. En este superpaso no se realiza ninguna operación aritmética, por lo que el coste computacional viene dado por el coste de comunicación. El procesador principal envía $4n - 4\frac{n}{p}$ elementos de la matriz A y del vector $\boldsymbol{d}$ al resto de procesadores. Además, todos los procesadores excepto el último, reciben dos elementos situados fuera de los bloques tridiagonales de A. El último procesador únicamente recibe un elemento situado fuera de los bloques tridiagonales. En consecuencia, el coste de comunicación es de $(4n - 4\frac{n}{p} + 2p - 3)g$ flops, por lo que el coste total de este superpaso es de

$$\left(4n - 4\frac{n}{p} + 2p - 3\right) g + l \quad \text{flops.} \tag{3.26}$$

Coste del superpaso 2. Comenzamos calculando el coste aritmético. Para calcular los elementos diagonales de las matrices G_i, utilizando las expresiones (3.5) y (3.6) se necesitan 2^{q-m} flops; es decir, $\dfrac{n}{p}$ flops. Para resolver los sistemas de los apartados 1.2 y 1.3 del superpaso 2 se necesitan $7 \cdot 2^{q-m-1}$ flops en cada uno de los procesadores no extremos. En consecuencia, el coste aritmético de realizar los apartados 1.1, 1.2 y 1.3 del superpaso 2 es de

$$9 \cdot 2^{q-m-1} = \frac{9}{2}\frac{n}{p} \quad \text{flops.} \tag{3.27}$$

Debido a la estructura particular de los vectores que aparecen en el apartado 1.4.3 del superpaso 2, son necesarios

$$10 + 2^{q-k+1} + 2(1 + 3 \cdot 2^{q-k+1}) = 12 + 5 \cdot 2^{q-k} \quad \text{flops}$$

para calcular los bloques de vectores $\boldsymbol{x}$, $\boldsymbol{w}_{2^{q-1-k}i}$ y $\boldsymbol{w}_{2^{q-1-k}(i-1)}$.

Como el número de bloques de vectores que se actualizan para cada procesador en esta etapa es 2^{k-m}, tenemos que el coste aritmético del apartado 1.4.3 del algoritmo es

$$2^{k-m}\left(12 + 5 \cdot 2^{q-k}\right) \quad \text{flops.} \tag{3.28}$$

Sumando las expresiones (3.27) y (3.28), se obtiene el coste aritmético del superpaso 2, que es de

$$\frac{9}{2}\frac{n}{p} + \sum_{k=m}^{q-2} 2^{k-m}\left[12 + 5\cdot 2^{q-k}\right] = \left(5q - 5m + \frac{11}{2}\right)\frac{n}{p} - 12 \quad \text{flops.} \tag{3.29}$$

Para obtener el coste de comunicación de este superpaso, hay que tener en cuenta que únicamente la mitad de los procesadores envían datos a otros procesadores. Cada procesador impar, salvo el último, envía tres vectores de tamaño 2^{q-m} al procesador contiguo, mientras que el último procesador envía sólamente dos vectores de tamaño 2^{q-m} a su procesador par contiguo. Así, como cada procesador, salvo el último, envía o recibe $3\cdot 2^{q-m}$ unidades de datos, el coste de comunicación es el de una $3\cdot 2^{q-m}$-relación, es decir, $3\frac{n}{p}g$ flops.

Sumando el coste aritmético y de comunicación en este superpaso y efectuando las oportunas simplificaciones, se obtiene un coste computacional para el superpaso 2 de

$$\left(5q - 5m + \frac{11}{2}\right)\frac{n}{p} - 12 + 3\frac{n}{p}g + l \quad \text{flops.} \tag{3.30}$$

Coste del superpaso $\boldsymbol{m-k+2}$, para $k = m-1, m-2, \ldots, 2, 1$. Para calcular el coste de estos $m-1$ superpasos, fijamos un valor de k y calculamos el coste computacional de este superpaso concreto. Después, sumamos los costes de todos los superpasos en función de k.

Por lo tanto, dado un cierto k, calculamos en primer lugar el coste del superpaso $m-k+2$. En este superpaso cada procesador activo sólo efectúa una actualización de los bloques de los vectores $\boldsymbol{x}$, $\boldsymbol{w}_{2^{q-1-k}i}$ y $\boldsymbol{w}_{2^{q-1-k}(i-1)}$, utilizando las expresiones (3.9), (3.10) y (3.11). Por la tanto, el coste aritmético de este superpaso es de

$$12 + 5\cdot 2^{q-k} \quad \text{flops.} \tag{3.31}$$

En cuanto al coste de comunicación, notemos que en todos los superpasos se repite el mismo patrón de comunicación: unos cuantos procesadores envían datos a sus procesadores vecinos. La diferencia en cada superpaso radica en el tamaño de la h-relación. Fijado k, la

comunicación se corresponde con una $3 \cdot 2^{q-k}$-relación, por lo que el coste de comunicación es de $3 \cdot 2^{q-k} g$ flops. Por lo tanto, sumando los costes aritmético y de comunicación obtenemos que el coste computacional del superpaso $m-k+2$ es de

$$12 + 5 \cdot 2^{q-k} + 3 \cdot 2^{q-k} g + l \quad \text{flops.}$$

En consecuencia, calculamos ahora el coste computacional de los superpasos $m-k+2$, para $k = m-1, m-2, \ldots, 2, 1$, que viene dado por

$$\sum_{k=1}^{m-1} \left(12 + 5 \cdot 2^{q-k} + 3 \cdot 2^{q-k} g + l\right)$$

o equivalentemente después de hacer las simplificaciones oportunas

$$12(m-1) + 5\left(1 + \frac{2}{p}\right) n + 3\left(1 - \frac{2}{p}\right) ng + (m-1)l \quad \text{flops.} \tag{3.32}$$

Coste del superpaso m + 2. Este superpaso es análogo al superpaso 3 del algoritmo 3.3, por lo que su coste computacional viene dado por la expresión (3.24). Así pues, para obtener el coste computacional del algoritmo 3.4 sumamos las expresiones (3.26), (3.30), (3.24) y (3.32) y realizamos las simplificaciones oportunas, lo que produce un coste total de

$$\left[\frac{1}{p}\left(5q - 5m - \frac{9}{2}\right) + 7\right] n + 12m - 16 + \left(7n - 3\frac{n}{p} + 2p - 3\right) g + (m+2)l \quad \text{flops.} \tag{3.33}$$

3.4 Un algoritmo BSP basado en el método *recursive doubling*

3.4.1 Descripción del algoritmo

Podemos considerar una modificación en el método descrito por medio del algoritmo 3.4. Esta modificación se basa en el modo en que se llevan a cabo las comunicaciones que nos conducen al cálculo y actualización de los bloques de vectores a partir de los que se obtiene $\boldsymbol{x}(1:2^q)$. Consideramos un nuevo algoritmo dividido también en dos fases claramente diferenciadas; una primera fase totalmente análoga a la primera parte del algoritmo 3.4 en la que se realizan únicamente cálculos de tipo aritmético, sin efectuar comunicaciones. En la segunda parte del algoritmo se realizan las comunicaciones de forma distinta al modelo *fan-in* que seguimos en el algoritmo 3.4. Ahora se sigue un modelo en el que todos los procesadores comunican sus bloques de vectores al procesador principal, que es el encargado de calcular el resto de actualizaciones hasta llegar a la obtención de la solución final. Nótese que, para el caso particular en que $m = 1$, el algoritmo que produce este nuevo modelo de comunicación coincide exactamente con el algoritmo 3.3, ya que las comunicaciones se producen desde el procesador P_1 al P_0, al igual que ocurría en el caso del modelo de comunicación *fan-in*.

El algoritmo 3.5 resume las características esenciales de este método siguiendo el esquema de comunicación anteriormente expuesto y teniendo en cuenta que $m \geq 2$.

Algoritmo 3.5 Un algoritmo BSP para sistemas tridiagonales para $m \geq 2$.

Superpaso 1

El procesador P_0 envía al procesador P_j, para $j = 1, 2, \ldots, 2^m - 1$, los elementos $a_{2^{q-m}j+i}$, $b_{2^{q-m}j+i-1}$ y $c_{2^{q-m}j+i+1}$, para $i = 1, 2, \ldots, 2^{q-m}$, suponiendo que $c_{n+1} = 0$.

Superpaso 2

1 En el procesador P_j, para $j = 1, 2, \ldots, 2^m - 1$,

1.1 calcular $\hat{a}_{2^{q-m}j+i}$, para $i = 1, 2, \ldots, 2^{q-m}$, utilizando las expresiones (3.5) y (3.6),

1.2 calcular $\boldsymbol{x}_{(2^{q-m}j+2i-1\,:\,2^{q-m}j+2i)}$, para $i = 1, 2, \ldots, 2^{q-m-1}$ utilizando la expresión (3.7),

1.3 calcular $\boldsymbol{w}_{2^{q-m-1}j+i(2^{q-m}j+2i-1\,:\,2^{q-m}j+2i)}$ y $\boldsymbol{w}_{2^{q-m-1}j+i-1(2^{q-m}j+2i-1\,:\,2^{q-m}j+2i)}$, para $i = 1, 2, \ldots, 2^{q-m-1}$, utilizando (3.8) y suponiendo que $\boldsymbol{w}_0$ y $\boldsymbol{w}_{2^q}$ son vectores nulos,

1.4 para $k = q-2, q-3, \ldots, m$, y para $i = 1, 2, \ldots, 2^{k-m}$ calcular

1.4.1 $\boldsymbol{x}_{(2^{q-m}j+2^{q-k}(i-1)+1\,:\,2^{q-m}j+2^{q-k}i)}$ utilizando la expresión (3.9),

1.4.2 $\boldsymbol{w}_{2^{q-m-1}j+2^{q-k-1}i(2^{q-m}j+2^{q-m}(i-1)+1\,:\,2^{q-m}j+2^{q-k}i)}$ utilizando la expresión (3.10),

1.4.3 $\boldsymbol{w}_{2^{q-m-1}j+2^{q-k-1}(i-1)(2^{q-m}j+2^{q-m}(i-1)+1\,:\,2^{q-m}j+2^{q-k}i)}$ utilizando la expresión (3.11).

2 El procesador P_j, para $j = 1, 2, \ldots, p-1$, envía al procesador P_0 los bloques de vectores $\boldsymbol{x}_{(2^{q-m}j+1\,:\,2^{q-m}j+2^{q-m})}$, $\boldsymbol{w}_{2^{q-m-1}j+2^{q-m-1}(2^{q-m}j+1\,:\,2^{q-m}j+2^{q-m})}$, y $\boldsymbol{w}_{2^{q-m-1}j(2^{q-m}j+1\,:\,2^{q-m}j+2^{q-m})}$.

Superpaso 3

1 Para $k = m-1, m-2, \ldots, 2, 1$ y para $i = 0, 1, \ldots, 2^k-1$, el procesador P_0 calcula

1.1 $\boldsymbol{x}_{(2^{m-k+3}i+1\,:\,2^{m-k+3}(i+1))}$, utilizando la expresión (3.9),

1.2 $\boldsymbol{w}_{2^{m-k+2}(i+1)(2^{m-k+3}i+1\,:\,2^{m-k+3}(i+1))}$, utilizando la expresión (3.10),

1.3 $\boldsymbol{w}_{2^{m-k+2}i(2^{m-k+3}i+1\,:\,2^{m-k+3}(i+1))}$, utilizando la expresión (3.11).

2 El procesador P_0 calcula $\boldsymbol{x}(1 : 2^q)$ utilizando la expresión (3.12).

La figura 3.4 muestra de forma detallada un ejemplo para una matriz de tamaño 32×32, siguiendo el nuevo esquema de comunicaciones expuesto al inicio de esta sección.

Notemos que el algoritmo 3.5 únicamente requiere tres superpasos, ya que la comunicación se produce de forma masiva al final del superpaso 2. En este superpaso cada procesador envía al procesador principal sus bloques de vectores actualizados de manera que es el

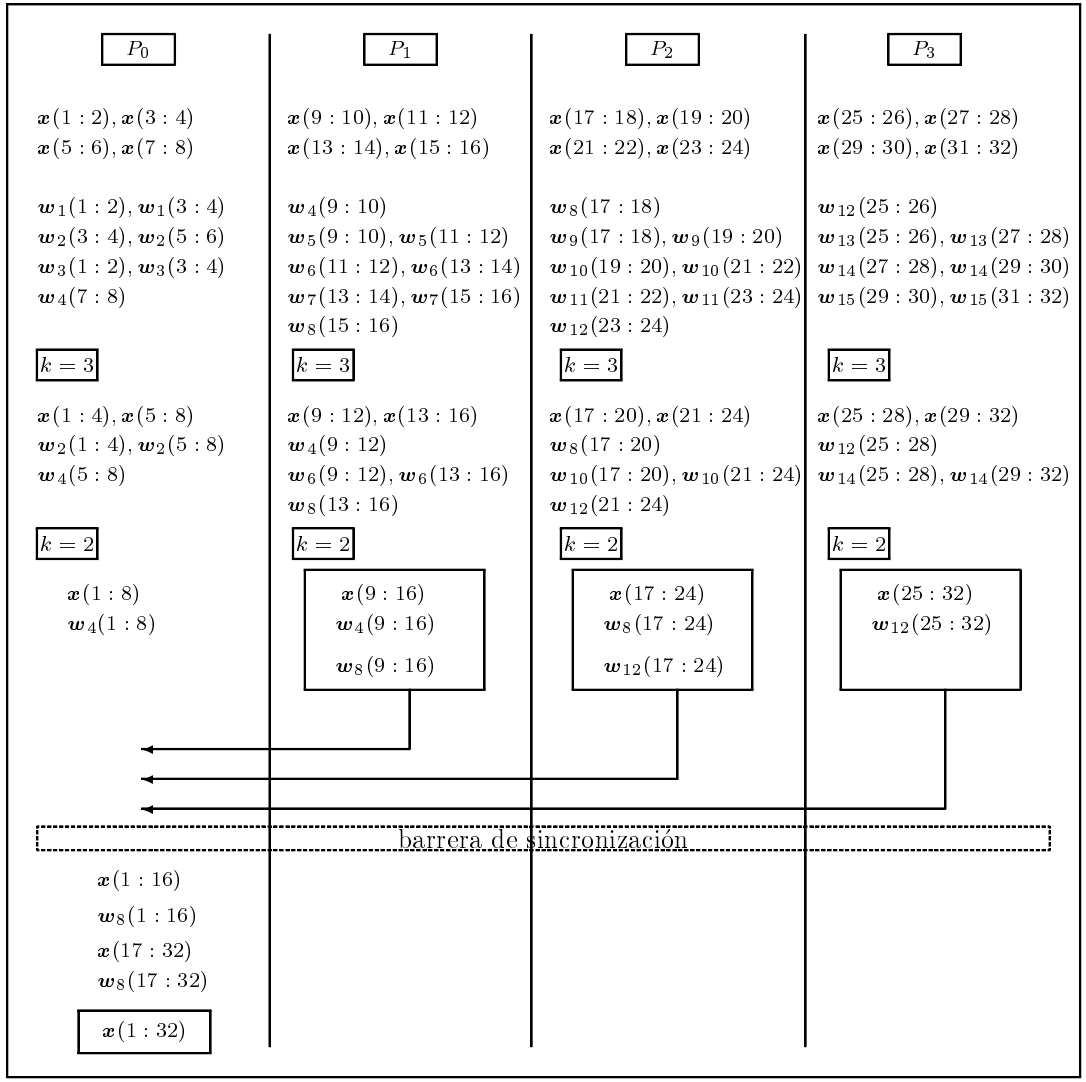


Figura 3.4: *Esquema del algoritmo 3.4 para el caso $n = 32$ y $p = 4$.*

procesador principal el encargado de finalizar el proceso de una forma secuencial. Esta es una de las diferencias más importantes con el algoritmo 3.4, en el que el número de superpasos necesarios estaba en función del parámetro m, por lo que depende del número de procesadores, ya que $p = 2^m$. Cuando el número de procesadores que se utilizan es bajo no hay diferencias significativas entre ambas implementaciones en lo que respecta al número total de superpasos requeridos. Así, si $p = 4$ tendremos que $m = 2$, por lo que en el caso del algoritmo 3.4 se necesitan cuatro superpasos, mientras que en el caso del algoritmo 3.5 únicamente se requieren tres superpasos. Si aumentamos el número de procesadores hasta por ejemplo $p = 128$, la situación cambia ya que se necesitarán 9 y 3 superpasos para ejecutar los algoritmos 3.4 y 3.5, respectivamente. Esta diferencia sí puede resultar significativa, especialmente si la máquina en la que se trabaja presenta un coste alto de sincronización.

Otro aspecto digno de análisis en esta nueva implementación es el modelo de comunicación que se establece en el superpaso 2 del algoritmo 3.5. De acuerdo con este modelo cada procesador, excepto el último, envía al procesador P_0 tres bloques de vectores de tamaño 2^{q-m}, mientras que el último procesador únicamente envía dos bloques de vectores de tamaño 2^{q-m}. En consecuencia, el coste de comunicación es de

$$\left(3n - 4\frac{n}{p}\right) g = \left(3 - \frac{4}{p}\right) ng \quad \text{flops}, \tag{3.34}$$

lo que supone un coste bastante mayor que el que se obtiene siguiendo el proceso de comunicación descrito en el superpaso 2 del algoritmo 3.4, que viene dado por la expresión (3.30). Si comparamos las expresiones (3.34) y (3.30), llegamos a la conclusión que

$$\frac{3}{p} ng \leq \left(3 - \frac{4}{p}\right) ng$$

siempre que $p \geq \frac{7}{3}$, lo que se cumple para $p \geq 3$. Hay que tener en cuenta que en el algoritmo 3.5 no se realizan más comunicaciones a lo largo de todo el proceso, mientras que en el algoritmo 3.4 necesitamos llevar a cabo otros tres pasos de comunicación.

3.4.2 Coste computacional

En esta sección se calcula el coste computacional del método descrito en la sección 3.3, de acuerdo con la implementación en paralelo que proporciona el algoritmo 3.5. Para ello, se calcula el coste de cada uno de los superpasos que integran el algoritmo.

Coste del superpaso 1. Este superpaso es análogo al superpaso 1 del algoritmo 3.4, por lo que su coste viene dado por la expresión (3.26).

Coste del superpaso 2. La única diferencia existente entre este superpaso y el superpaso 2 del algoritmo 3.4 se encuentra en el proceso de comunicación, por lo que ambos superpasos tienen el mismo coste aritmético dado por la expresión (3.29).

En cuanto al coste de comunicación, éste viene dado por la expresión (3.34), como ya se analizó anteriormente. En consecuencia, sumando las expresiones (3.29) y (3.34) tendremos el coste computacional del superpaso 2 del algoritmo 3.5, que es de

$$\left(5q - 5m + \frac{11}{2}\right)\frac{n}{p} - 12 + \left(3n - 4\frac{n}{p}\right)g + l \quad \text{flops.} \tag{3.35}$$

Coste del superpaso 3. En este superpaso no se producen comunicaciones de elementos entre los procesadores, por lo que únicamente contamos las operaciones aritméticos que se realizan.

Recordemos que son necesarios $12 + 5 \cdot 2^{q-k}$ flops (véase la expresión (3.31)), para calcular los bloques de vectores $\boldsymbol{x}$, $\boldsymbol{w}_{2^{q-1-k}i}$ y $\boldsymbol{w}_{2^{q-1-k}(i-1)}$. Además, como el número de bloques de vectores que se actualizan en el procesador principal es 2^k, tenemos que el coste aritmético de cualquiera de los k pasos que se llevan a cabo en este superpaso es de $2^{k-m}\left(12 + 5 \cdot 2^{q-k}\right)$ flops.

De esta forma, el coste aritmético de este superpaso es de

$$\sum_{k=1}^{m-1} 2^k \left[12 + 5 \cdot 2^{q-k}\right] = 5(m-1)n + 12(p-2) \quad \text{flops.}$$

Finalmente, el procesador principal calcula la solución al final del superpaso 3, lo que representa un coste de $8 + 2n$ flops.

Por lo tanto, el coste total del superpaso 3 es de

$$5(m-3)n + 12(p-2) + 8 + l \quad \text{flops.} \tag{3.36}$$

Sumando las expresiones (3.26), (3.35) y (3.36) y realizando las simplificaciones oportunas, se obtiene que el coste computacional del algoritmo 3.5 es

$$\frac{1}{p}\left[m(5p-3) + 5q - 3p - \frac{11}{2}\right] n + 12(p-2) - 4 + \left(7n - \frac{8}{p}n + 2p - 3\right) g + 3l \quad \text{flops.} \tag{3.37}$$

3.5 Resultados teóricos y comparaciones

En esta sección se presentan los tiempos teóricos de ejecución de los algoritmos 3.4 y 3.5 sobre las tres máquinas paralelas que se describieron brevemente en la sección 2.5. Los valores que aparecen en las columnas de 2 procesadores se han calculado utilizando la expresión (3.25) que proporciona el coste computacional del algoritmo 3.3. Cuando el número de procesadores aumenta ya se comparan los tiempos obtenidos para los algoritmos 3.4 y 3.5 utilizando las expresiones (3.33) y (3.37), respectivamente.

3.5.1 Tiempos en un IBM SP2

Analizamos de forma separada los resultados que se obtienen en esta máquina cuando se utilizan los dos tipos de conexión: switch y ethernet. La tabla 3.4 recoge los tiempos comparados de los algoritmos estudiados en este capítulo para un IBM SP2 dotado con un switch de alto rendimiento y con una conexión de tipo ethernet.

IBM SP2 switch					
	p				
	2	4		8	
n	Alg. 3.3	Alg. 3.4	Alg. 3.5	Alg. 3.4	Alg. 3.5
512	0.0009	0.0013	0.0010	0.0017	0.0014
1024	0.0017	0.0020	0.0017	0.0024	0.0021
2048	0.0033	0.0035	0.0031	0.0038	0.0036
4096	0.0068	0.0066	0.0060	0.0066	0.0067
8192	0.0142	0.0131	0.0119	0.0123	0.0130
16384	0.0297	0.0263	0.0241	0.0239	0.0256
32768	0.0622	0.0537	0.0494	0.0475	0.0514
65536	0.1305	0.1099	0.1016	0.0954	0.1037
131072	0.2735	0.2255	0.2090	0.1929	0.2099
262144	0.5719	0.4631	0.4302	0.3910	0.4255
524288	1.1940	0.9508	0.8852	0.7934	0.8630
1048576	2.4886	1.9514	1.8203	1.6110	1.7505
2097152	5.1786	4.0033	3.7410	3.2713	3.5507
4194304	10.7602	8.2077	7.6833	6.6424	7.2016

(a) *Switch*

IBM SP2 ethernet					
	p				
	2	4		8	
n	Alg. 3.3	Alg. 3.4	Alg. 3.5	Alg. 3.4	Alg. 3.5
512	0.0081	0.0312	0.0247	0.1015	0.0868
1024	0.0141	0.0554	0.0442	0.1830	0.1608
2048	0.0261	0.1039	0.0831	0.3461	0.3088
4096	0.0503	0.2010	0.1612	0.6724	0.6048
8192	0.0992	0.3954	0.3175	1.3250	1.197
16384	0.1978	0.7846	0.6305	2.6305	2.3814
32768	0.3966	1.5638	1.2572	5.2418	4.7508
65536	0.7972	3.1237	2.5124	10.4652	9.4903
131072	1.6048	6.2468	5.0257	20.9136	18.971
262144	3.2327	12.4991	10.0588	41.8135	37.9354
524288	6.5135	25.0165	20.1374	83.6197	75.8705
1048576	13.1256	50.0765	40.3200	167.2446	151.7534
2097152	26.4505	100.2468	80.7356	334.5197	303.5444
4194304	53.3022	200.6883	161.6675	669.1203	607.1767

(b) *Ethernet*

Tabla 3.1: *Tiempos teóricos para los algoritmos 3.3, 3.4 y 3.5 medidos en un IBM SP2 dotado con un switch de alto rendimiento y conexión ethernet.*

La tabla 3.1(a) nos muestra los tiempos teóricos esperados con switch. Analizando de forma general esta tabla se observa que para 4 procesadores el algoritmo 3.4, que recordemos se basaba en una comunicación de tipo *fan-in*, siempre es más lento que el algoritmo 3.5. Sin embargo, cuando el número de procesadores aumenta a 8 ya no podemos afirmar esta conclusión para cualquier valor de n. A partir de $n = 4096$ ya se observa que el algoritmo 3.4 es más rápido que el algoritmo 3.5. Estas diferencias se van acrecentando a medida que aumenta n. También se observa que al ir aumentando p, los tiempos de ejecución van disminuyendo, especialmente para valores grandes de n. Este descenso en los tiempos es más notable para el algoritmo 3.4 que para el algoritmo 3.5, en el que para valores pequeños de n se observa un ligero incremento en los tiempos al aumentar p.

La tabla 3.1(b) nos muestra los tiempos teóricos esperados utilizando una conexión ethernet. En dicha tabla se observa que para 4 y 8 procesadores el algoritmo 3.4 siempre es más lento que el algoritmo 3.5. Las diferencias se van reduciendo a medida que aumenta el número de procesadores. De la misma forma, se observa que al aumentar el número de procesadores, los tiempos de ejecución de los dos algoritmos aumentan, lo que nos da una idea de que no resultan eficientes en una máquina con estas características.

Con el objetivo de realizar un estudio comparativo de los algoritmos presentados en este capítulo, no sólamente nos interesa establecer qué algoritmo es más rápido sino también las diferencias que existen entre ambos en los tiempos previstos de ejecución. Para determinar estos valores formamos una nueva tabla en la que calculamos el pocentaje de tiempo que nos ahorramos al ejecutar el algoritmo más rápido para esos valores concretos. Así, si para un determinado valor de n y p, $T_{alg.\ 3.4}$ representa el tiempo de ejecución del algoritmo 3.4, $T_{alg.\ 3.5}$ representa el tiempo de ejecución del algoritmo 3.5 y suponemos que $T_{alg.\ 3.4} > T_{alg.\ 3.5}$, entonces el valor que aparece en la tabla 3.2 viene dado por la expresión

$$\frac{T_{alg.\ 3.4} - T_{alg.\ 3.5}}{T_{alg.\ 3.4}} 100. \tag{3.38}$$

Como en la mayoría de los casos el más rápido es el algoritmo 3.5, únicamente señalaremos en la tabla cuando el porcentaje de ahorro de tiempos sea a favor del algoritmo 3.4. Señalizaremos en la tabla este caso con una F.

Se observa que, por ejemplo utilizando ethernet, al resolver el sistema mediante el algoritmo 3.5 nos podemos ahorrar aproximadamente

IBM SP2				
	p			
	4		8	
n	sw	et	sw	et
512	23.08	20.83	17.65	14.48
1024	15.00	20.22	12.50	12.13
2048	11.43	20.02	5.26	10.78
4096	9.09	19.80	1.49(F)	10.05
8192	9.16	19.70	5.38(F)	9.66
16384	8.37	19.64	6.64(F)	9.47
32768	8.01	19.61	7.59(F)	9.37
65536	7.55	19.57	8.00(F)	9.32
131072	7.32	19.55	8.10(F)	9.29
262144	7.10	19.52	8.11(F)	9.27
524288	6.90	19.50	8.06(F)	9.27
1048576	6.72	19.48	7.97(F)	9.26
2097152	6.55	19.46	7.87(F)	9.26
4194304	6.39	19.44	7.76(F)	9.26

Tabla 3.2: *Diferencias de tiempos entre los algoritmos 3.4 y 3.5, medidos en porcentajes.*

el 20% del tiempo de ejecución del algoritmo 3.4 si se utilizan 4 procesadores y de alrededor del 10% cuando empleamos los 8 procesadores. Estas cifras resultan muy significativas.

De la lectura de la tabla 3.2 se pueden destacar varios aspectos notables.

- Al aumentar el número de procesadores de 4 a 8, las diferencias de tiempo a favor del algoritmo 3.5 se van reduciendo para cualquier valor de n y utilizando cualquier hardware para las comunicaciones. Para el caso de 8 procesadores y comunicación switch, cuando $n > 2048$, ya resulta más conveniente utilizar el algoritmo 3.4.

- Para $p = 4$, las diferencias de tiempos entre los dos algoritmos son mayores para la máquina con conexión ethernet que para la misma máquina con switch. Por ejemplo, cuando $n = 1048576$, se observa en la tabla 3.2 que utilizando el switch el algoritmo 3.4 es un 6.7% aproximadamente más lento que el algoritmo 3.5. Sin embargo, cuando se utiliza la conexión ethernet, el algoritmo 3.5 es un 20% aproximadamente más rápido que el algoritmo 3.4. Esto representa una variación significativa a tener en cuenta. Además, las diferencias se mantienen constantes alrededor del 20% para $p = 4$ y alrededor del 10% para $p = 8$, independientemente del tamaño de la matriz.

- Para 4 procesadores el máximo ahorro de tiempo se produce para ethernet y $n = 512$, siendo el valor del porcentaje del 20.83%. Para 8 procesadores, el máximo ahorro de tiempo se produce para switch y $n = 512$, siendo del 17.65%. Nótese el descenso en los valores máximos al ir aumentando el número de procesadores.

En consecuencia, como conclusión general para una máquina de este tipo se puede decir que el algoritmo 3.4, basado en un modelo de comunicación *fan-in*, siempre es más lento que el algoritmo 3.5, salvo para 8 procesadores y conexión con switch. Cuando se utiliza el switch de alto rendimiento, las diferencias en tiempos disminuyen sensiblemente al aumentar el número de procesadores. Este comportamiento nos lleva a pensar que el algoritmo 3.4 presenta un mejor comportamiento en máquinas donde las comunicaciones no son muy costosas.

CRAY T3D															
	p														
	2	4		8		16		32		64		128		256	
n	Alg. 3.3	Alg. 3.4	Alg. 3.5	Alg. 3.4	Alg. 3.5	Alg. 3.4	Alg. 3.5	Alg. 3.4	Alg. 3.5	Alg. 3.4	Alg. 3.5	Alg. 3.4	Alg. 3.5	Alg. 3.4	Alg. 3.5
2048	0.0053	0.0034	0.0034	0.0026	0.0035	0.0024	0.0041	0.0024	0.0049	0.0025	0.0059	0.0029	0.0072	0.0037	0.0088
4096	0.0114	0.0071	0.0072	0.0052	0.0070	0.0045	0.0080	0.0044	0.0096	0.0043	0.0112	0.0048	0.0133	0.0056	0.0158
8192	0.0245	0.0150	0.0152	0.0106	0.0143	0.0089	0.0159	0.0085	0.0189	0.0081	0.0218	0.0086	0.0257	0.0094	0.0299
16384	0.0523	0.0316	0.0320	0.0220	0.0293	0.0180	0.0320	0.0169	0.0376	0.0157	0.0431	0.0162	0.0504	0.0171	0.0582
32768	0.1115	0.0665	0.0673	0.0455	0.0601	0.0364	0.0645	0.0338	0.0752	0.0309	0.0859	0.0314	0.0999	0.0326	0.1147
65536	0.2365	0.1397	0.1414	0.0942	0.1235	0.0743	0.1304	0.0680	0.1509	0.0617	0.1716	0.0620	0.1989	0.0636	0.2277
131072	0.5003	0.2930	0.2964	0.1950	0.2537	0.1516	0.2640	0.1373	0.3032	0.1237	0.3436	0.1233	0.3972	0.1257	0.4539
262144	1.0552	0.6131	0.6199	0.4035	0.5209	0.3097	0.5346	0.2776	0.6095	0.2484	0.6882	0.2464	0.7943	0.2501	0.9065
524288	2.2195	1.2808	1.2944	0.8341	1.0689	0.6326	1.0825	0.5616	1.2255	0.4996	1.3792	0.4935	1.5892	0.4993	1.8121
1048576	4.6575	2.6707	2.6980	1.7227	2.1923	1.2922	2.1919	1.1365	2.4642	1.0055	2.7647	0.9893	3.1808	0.9987	3.6243
2097152	9.7518	5.5597	5.6143	3.5544	4.4937	2.6387	4.4382	2.2999	4.9554	2.0240	5.5425	1.9844	6.3674	1.9991	7.2502
4194304	20.3774	11.5563	11.6655	7.3271	9.2058	5.3863	8.9853	4.6540	9.9651	4.0748	11.1117	3.9814	12.7475	4.0034	14.5055

Tabla 3.3: *Tiempos teóricos para los algoritmos 3.3, 3.4 y 3.5 medidos en un CRAY T3D para* 2, 4, 8, 16, 32, 64, 128 *y* 256 *procesadores.*

3.5.2 Tiempos en un CRAY T3D

La tabla 3.3 muestra los resultados teóricos esperados en un CRAY T3D, una máquina altamente paralela que dispone de hasta 256 procesadores y en la que las comunicaciones son muy rápidas. En las tablas que resumen los resultados teóricos los tamaños de la matriz de coeficientes comienzan a partir de 2048 debido al elevado número de procesadores disponibles en dicha máquina.

La primera conclusión que extraemos observando la tabla 3.3 es que, salvo para el caso en que $p = 2$, el algoritmo 3.4 es más rápido

que el algoritmo 3.5. Para $p = 4$ los resultados que se obtienen en ambos algoritmos son muy parecidos, especialmente en matrices de tamaño pequeño y medio. Por tanto, el comportamiento general de ambos algoritmos difiere radicalmente de lo que ocurría en el IBM SP2 en el que el algoritmo 3.4 era siempre más lento que el algoritmo 3.5, salvo cuando la conexión era a través del switch y $p = 8$.

También se observa de forma generalizada que al ir aumentando el número de procesadores, los tiempos van disminuyendo, como es de esperar en una máquina de estas características. Esta tendencia a la reducción de tiempos se cumple en el algoritmo 3.4 hasta $p = 128$. Para $p = 128$ ya únicamente se obtienen mejores tiempos para matrices con $n > 65536$. Para valores de $n < 65536$ los tiempos son muy parecidos, aunque ya se advierte que aumentan muy ligeramente al pasar de 64 a 128 procesadores. Al aumentar el número de procesadores de 128 a 256 se produce un aumento generalizado en los tiempos de ejecución para todos los tamaños, aunque dicho aumento de tiempo no es significativo. Para el algoritmo 3.5, se produce un aumento de tiempos para cualquier tamaño a partir de $p = 32$, aunque para matrices pequeñas ya se constatan aumentos a partir de $p = 8$. En general, el comportamiento paralelo de este algoritmo es peor que el del algoritmo 3.4.

En consecuencia, en una máquina de estas características, podemos hablar de un número de procesadores óptimo para la ejecución de cada uno de los algoritmos. Así, por ejemplo, observamos que resolver un sistema con $n = 1048576$ mediante el algoritmo 3.4 utilizando 128 procesadores cuesta un total de 0.9893 segundos, mientras que si utilizamos los 256 procesadores, costará 0.9987 segundos. Así, en general, podemos decir que, dependiendo del tamaño del sistema que queramos resolver, nos interesa utilizar un número alto de procesadores para ejecutar el algoritmo 3.4, por ejemplo, 64 o 128 procesadores; sin embargo, si ejecutamos el algoritmo 3.5 para resolver un sistema tridiagonal nos interesa utilizar un número bajo de procesadores para obtener los mejores resultados, por ejemplo, 8 o 16 procesadores. La tabla 3.4(a) resume el número óptimo de procesadores para los que se obtienen los mejores tiempos, dependiendo del tamaño de la matriz de coeficientes.

Volvemos a construir una tabla similar a la que construimos para la máquina IBM SP2 donde se reflejan las diferencias en procentajes de los tiempos de ejecución de ambos algoritmos. La tabla 3.4(b) muestra estos resultados, teniendo en cuenta que la expresión (3.38) es la utilizada para la obtención de estos porcentajes.

CRAY T3D		
n	Alg. 3.4	Alg. 3.5
2048	32	4
4096	64	8
8192	64	8
16384	64	8
32768	64	8
65536	64	8
131072	128	8
262144	128	8
524288	128	8
1048576	128	16
2097152	128	16
4194304	128	16

(a) *Número óptimo de procesadores*

CRAY T3D							
	p						
n	4	8	16	32	64	128	256
2048	0	25.71	41.46	51.02	57.63	59.72	57.95
4096	1.39	25.71	43.75	54.17	61.61	63.91	64.56
8192	1.32	25.87	44.02	55.03	62.84	66.54	68.56
16384	1.25	24.91	43.75	55.05	63.57	67.86	70.62
32768	1.19	24.29	43.57	55.05	64.03	68.57	71.58
65536	1.20	23.72	43.02	54.94	64.04	68.83	72.07
131072	1.15	23.13	42.58	54.72	64.00	68.96	72.31
262144	1.10	22.54	42.07	54.45	63.91	68.98	72.41
524288	1.05	21.97	41.56	54.17	63.78	68.95	72.45
1048576	1.01	21.42	41.05	53.88	63.63	68.90	72.44
2097152	0.97	20.90	40.56	53.59	63.48	68.83	72.43
4194304	0.94	20.41	40.05	53.30	63.33	68.77	72.40

(b) *Diferencias de tiempos entre los algoritmos 3.4 y 3.5*

Tabla 3.4: *Número óptimo de procesadores y diferencias de tiempos para la ejecución de los algoritmos 3.4 y 3.5 en un CRAY T3D.*

A partir de 2 procesadores, el algoritmo 3.4 siempre es más rápido. Se observa que al ir aumentando el número de procesadores, el porcentaje de ahorro de tiempos al ejecutar dicho algoritmo va aumentando progresivamente, especialmente al pasar de 8 a 16 y de 16 a 32 procesadores. Para $p = 32$ se observa que el tiempo de ejecución del algoritmo 3.5 es más del doble del tiempo de ejecución del algoritmo 3.4. Los valores máximos se obtienen para 256 procesadores, donde el tiempo de ejecución del algoritmo 3.4 es casi tres cuartas partes el tiempo de ejecución del algoritmo 3.5.

Así pues, la conclusión general que podemos extraer es que en esta máquina, salvo que trabajemos con dos procesadores, siempre es más rápido el algoritmo 3.4, aumentando las diferencias de tiempos de ejecución a medida que aumenta el número de procesadores. Los valores óptimos del algoritmo 3.4 se obtienen para 64 y 128 procesadores, dependiendo del tamaño de la matriz, mientras que los valores óptimos del algoritmo 3.5 se obtienen para 8 y 16 procesadores, dependiendo también del tamaño de la matriz.

3.5.3 Tiempos en un cluster de Pentiums

Ahora se analizan los resultados esperados sobre un cluster de Pentiums, siguiendo un esquema similar al descrito para las máquinas anteriores. Los resultados numéricos esperados se muestran en la tabla 3.5(a).

De esta tabla se desprende que, al igual que sucedía en el caso del IBM SP2, los tiempos de ejecución del algoritmo 3.4 son siempre mayores que el del algoritmo 3.5, para cualquier valor de n y de p. En este caso, a diferencia de lo que ocurría en las máquinas anteriores, no encontramos ningún caso en el que el algoritmo 3.4 sea más rápido que el algoritmo 3.5. A la vista de esta tabla sí podemos afirmar que, a medida que aumenta el número de procesadores, las diferencias entre los tiempos de ejecución se van acercando cada vez más, especialmente para matrices de tamaño medio y grande. Esto ya nos ocurría en las máquinas anteriores. En cuanto a los tiempos óptimos que se obtienen para los diferentes tamaños de la matriz, podemos decir que hasta $n = 524288$ los mejores tiempos se obtienen para $p = 2$ mediante el algoritmo 3.3. Sin embargo, cuando el valor de $n \geq 524288$ los mejores tiempos nos los proporciona el algoritmo 3.5 para $p = 4$, ya que al aumentar a $p = 8$, los tiempos también experimentan un incremento para este algoritmo. El comportamiento del algoritmo 3.4 es distinto; aunque no produce valores óptimos del tiempo, dichos tiempos se reducen al pasar de 4 a 8 procesadores para

Cluster de Pentiums					
	p				
	2	4		8	
n	Alg. 3.3	Alg. 3.4	Alg. 3.5	Alg. 3.4	Alg. 3.5
512	0.0006	0.0012	0.0010	0.0020	0.0015
1024	0.0011	0.0019	0.0015	0.0026	0.0022
2048	0.0020	0.0032	0.0027	0.0040	0.0034
4096	0.0039	0.0060	0.0050	0.0066	0.0060
8192	0.0079	0.0115	0.0096	0.0119	0.0111
16384	0.0160	0.0227	0.0191	0.0226	0.0215
32768	0.0328	0.0453	0.0382	0.0441	0.0423
65536	0.0673	0.0910	0.0768	0.0874	0.0841
131072	0.1381	0.1834	0.1551	0.1744	0.1682
262144	0.2835	0.3699	0.3136	0.3494	0.3373
524288	0.5817	0.7468	0.6342	0.7012	0.6775
1048576	1.1930	1.5079	1.2828	1.4085	1.3615
2097152	2.4453	3.0450	2.5950	2.8306	2.7370
4194304	5.0095	6.1490	5.2493	5.6896	5.5029

(a) *Tiempos*

Cluster de Pentiums		
	p	
n	4	8
512	16.67	25.00
1024	21.05	15.38
2048	15.63	15.00
4096	16.67	9.09
8192	16.52	6.72
16384	15.86	4.87
32768	15.67	4.08
65536	15.60	3.78
131072	15.43	3.56
262144	15.22	3.46
524288	15.08	3.38
1048576	14.93	3.34
2097152	14.78	3.31
4194304	14.63	3.28

(b) *Diferencias de tiempos*

Tabla 3.5: *Tiempos teóricos y diferencias de tiempos para los algoritmos 3.3, 3.4 y 3.5 en un cluster de Pentiums.*

$n \geq 16384$.

De la misma forma que hemos realizado en las máquinas anteriores un estudio de las diferencias entre los tiempos de ejecución de los algoritmos, ahora formamos la tabla 3.5(b), que muestra las diferencias de tiempo, en porcentaje, al ejecutar los algoritmos 3.4 y 3.5, siempre a favor del segundo. Como se observa en dicha tabla, las diferencias en los tiempos varían aproximadamente alrededor del 15% para 4 procesadores y disminuyen hasta el 3% para 8 procesadores. Las diferencias más importantes se producen para 4 procesadores y tamaños pequeños de la matriz de coeficientes. En ningún caso las diferencias de tiempo llegan al 50% como ocurría en las máquinas anteriores.

Como conclusión podemos decir que en este tipo de máquinas nuevamente el algoritmo 3.4 es más lento que el algoritmo 3.5, para cualquier valor de n y de p, aunque presenta un mejor comportamiento paralelo. También hay que resaltar que los valores óptimos de los tiempos hasta $n = 524288$ se obtienen para $p = 2$ utilizando el algoritmo 3.3 y, para tamaños de $n > 524288$, los valores óptimos se obtienen para $p = 4$ procesadores utilizando el algoritmo 3.4.

3.5.4 Estudio del *speedup*

En esta sección realizamos un estudio del speedup para los algoritmos 3.4 y 3.5 que hemos estudiado en este capítulo. Recordemos brevemente que el *speedup* es un parámetro que representa una medida de comparación de un algoritmo cuando se ejecuta utilizando 1 y p procesadores. Para una descripción más detallada de este parámetro, véase la sección 2.1.

La tabla 3.6 nos muestra tres tablas en las que aparecen los valores del *speedup* y de la eficiencia para una matriz de tamaño $n = 2097152$ en las tres máquinas en las que se han calculado tiempos teóricos.

La tabla 3.6(a) nos muestra el speedup y la eficiencia de los algoritmos 3.4 y 3.5 en un IBM SP2 para una matriz de gran tamaño, $n = 2097152$. La eficiencia la medimos en porcentaje.

IBM SP2 switch						
	speedup			eficiencia		
p	Alg. 3.3	Alg. 3.4	Alg. 3.5	Alg. 3.3	Alg. 3.4	Alg. 3.5
2	1.59			79.30		
4		2.17	2.32		54.15	57.95
8		2.65	2.44		33.13	30.53

(a) *IBM SP2 switch*

IBM SP2 ethernet						
	speedup			eficiencia		
p	Alg. 3.3	Alg. 3.4	Alg. 3.5	Alg. 3.3	Alg. 3.4	Alg. 3.5
2	0.29			14.32		
4		0.09	0.11		2.16	2.68
8		0.03	0.03		0.32	0.36

(b) *IBM SP2 ethernet*

CRAY T3D						
	speedup			eficiencia		
p	Alg. 3.3	Alg. 3.4	Alg. 3.5	Alg. 3.3	Alg. 3.4	Alg. 3.5
2	1.85			92.71		
4		3.38	3.35		84.48	83.66
8		5.29	4.18		66.07	52.26
16		7.12	4.23		44.50	26.46
32		8.17	3.79		25.53	11.85
64		9.28	3.39		17.19	6.28
128		9.47	2.95		7.40	2.31
256		9.40	2.59		3.67	1.01

(c) *CRAY T3D*

Cluster de Pentiums						
	speedup			eficiencia		
p	Alg. 3.3	Alg. 3.4	Alg. 3.5	Alg. 3.3	Alg. 3.4	Alg. 3.5
2	0.95			47.74		
4		0.84	0.99		21.03	24.68
8		0.91	0.94		11.31	11.70

(d) *Cluster de Pentiums*

Tabla 3.6: *Speedup y eficiencia de los algoritmos 3.3, 3.4 y 3.5 para* $n = 2097152$.

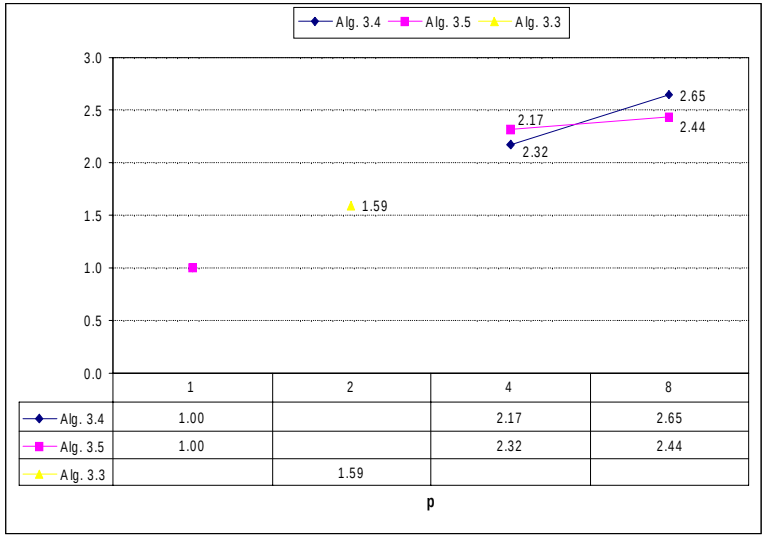
Notemos la gran diferencia en los valores del speedup y de la eficiencia en los dos algoritmos cuando se utiliza el switch de alto rendimiento frente a una conexión de tipo ethernet. Con el switch, la eficiencia alcanza un valor máximo del 79.30% para el algoritmo 3.3 cuando se utilizan 2 procesadores; sin embargo, con la conexión ethernet para efectuar comunicaciones, únicamente se alcanza un 14.32% de eficiencia como valor máximo para el mismo algoritmo con 2 procesadores. En general, cuando se utiliza un hardware lento para las comunicaciones como la conexión ethernet, la pérdida de eficiencia en ambos algoritmos es notable, obteniéndose unos resultados muy discretos para el speedup. La situación cambia cuando las comunicaciones son rápidas, obteniéndose valores muy aceptables del speedup.

Observamos que, aunque los porcentajes de eficiencia son mayores para el algoritmo 3.5 que para el algoritmo 3.4, este último algoritmo presenta un mejor comportamiento paralelo, como se desprende del aumento del speedup para conexión ethernet. Dicho aumento es mayor en proporción que el que obtenemos para el algoritmo 3.5.

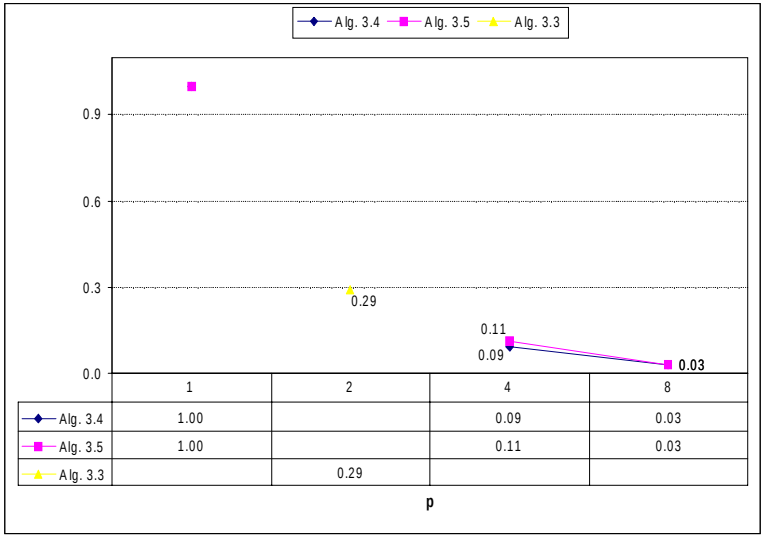
La tabla 3.6(c) es similar a las tablas 3.6(a) y 3.6(b) pero utilizando ahora una máquina del tipo CRAY T3D. En la misma se observa que se obtienen unos valores muy buenos del speedup y de la eficiencia, especialmente en el algoritmo 3.4, llegando a unos valores máximos de 9.47 para 128 procesadores, que es cuando se obtienen los mejores tiempos. También vemos claramente que los valores de la eficiencia y speedup del algoritmo 3.4 mejoran los del algoritmo 3.5, especialmente cuando el número de procesadores es alto, en el que las diferencias aumentan significativamente. En el caso $p = 2$, la eficiencia del algoritmo 3.5 es de 92.71%, lo que representa un máximo en este conjunto de valores. Sin embargo, al aumentar el número de procesadores, la eficiencia para este algoritmo decrece más rápidamente de lo que lo hace cuando se ejecuta el algoritmo 3.4.

A la vista de estos datos concluimos que, contrariamente a lo que ocurría con el IBM SP2, para una máquina como el CRAY T3D, el algoritmo 3.4 resulta no sólamente más rápido que el algoritmo 3.5, sino que presenta unos valores de *speedup* y eficiencia muy buenos, que lo convierten en un algoritmo altamente paralelo.

La tabla 3.6(d) nos muestra los valores del speedup y la eficiencia para un cluster de Pentiums, utilizando hasta un máximo de 8 procesadores. En este caso, siempre los valores de speedup y eficiencia son mejores para el algoritmo 3.5 frente a los obtenidos para el



(a) *IBM SP2 switch*



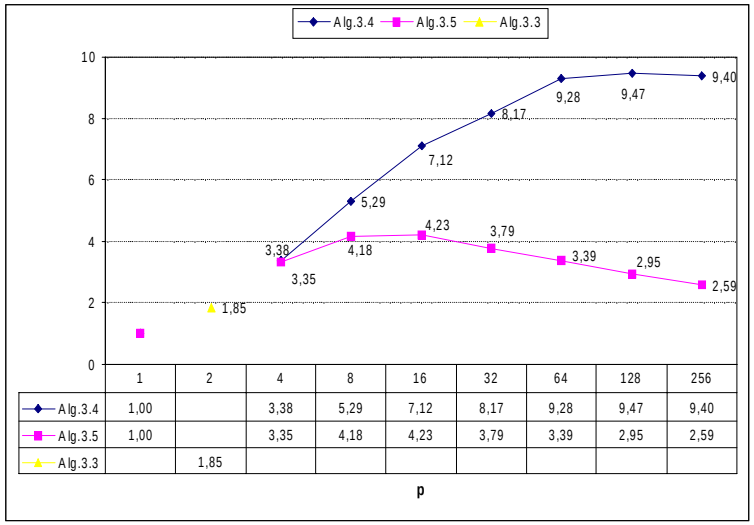
(b) *IBM SP2 ethernet*

Figura 3.5: *Valores del speedup en un IBM SP2*

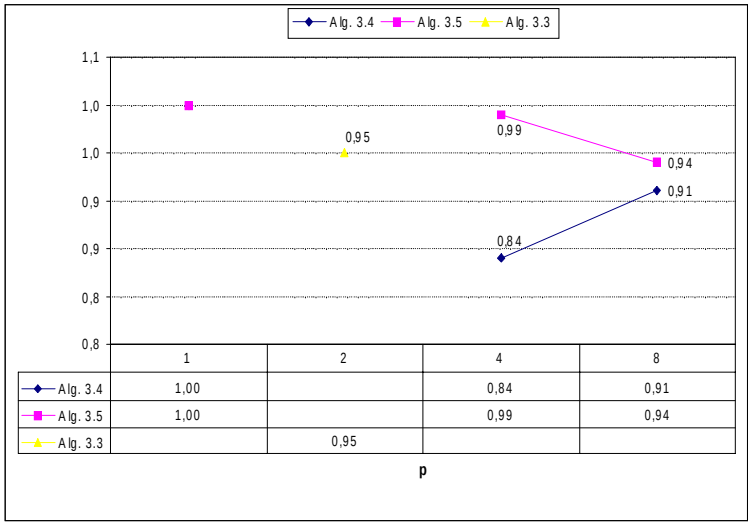
algoritmo 3.4. Los valores de la eficiencia no son tan buenos como los que se obtienen en el CRAY T3D.

Las figuras 3.5 y 3.6 nos muestran los valores del *speedup* obtenidos en la tabla 3.6.

Las figuras 3.7 y 3.8 nos muestra los valores obtenidos de la eficiencia para los algoritmos 3.4 y 3.5 en las tres máquinas donde se han estudiado los resultados numéricos teóricos de dichos algoritmos.

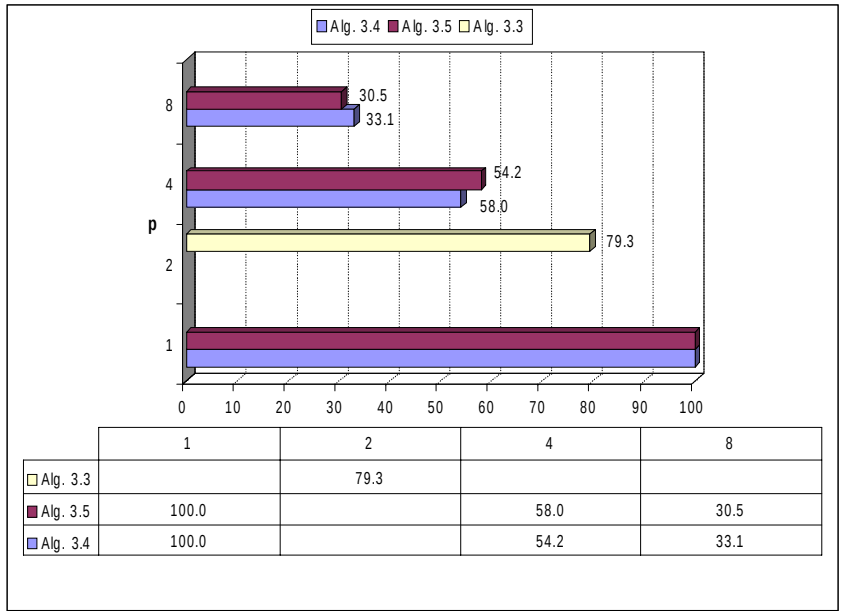


(a) *CRAY T3D*



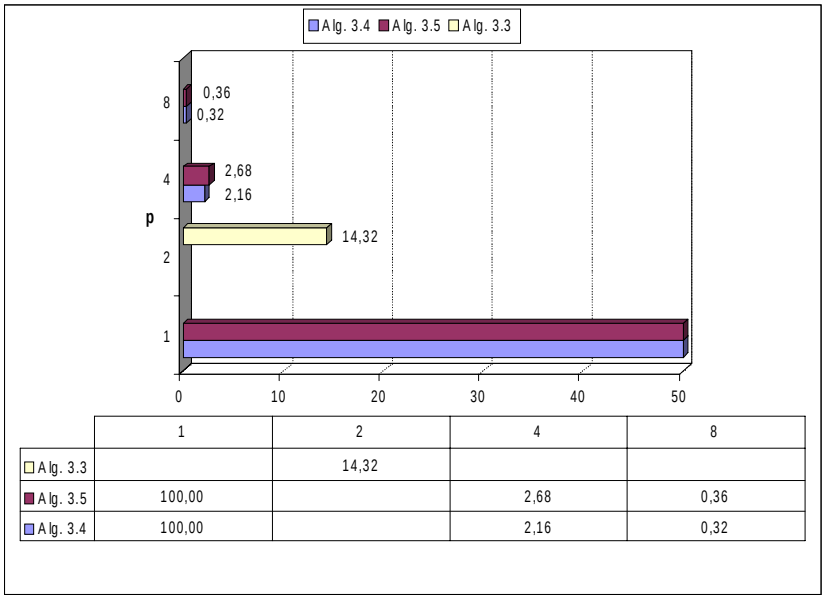
(b) *Cluster de Pentiums*

Figura 3.6: *Valores del speedup en un CRAY y un cluster de Pentiums*



	1	2	4	8
Alg. 3.3		79.3		
Alg. 3.5	100.0		58.0	30.5
Alg. 3.4	100.0		54.2	33.1

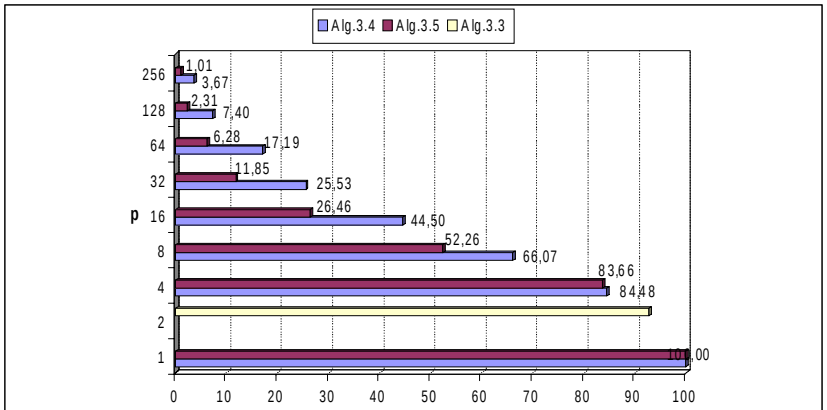
(a) *IBM SP2 switch*



	1	2	4	8
Alg. 3.3		14,32		
Alg. 3.5	100,00		2,68	0,36
Alg. 3.4	100,00		2,16	0,32

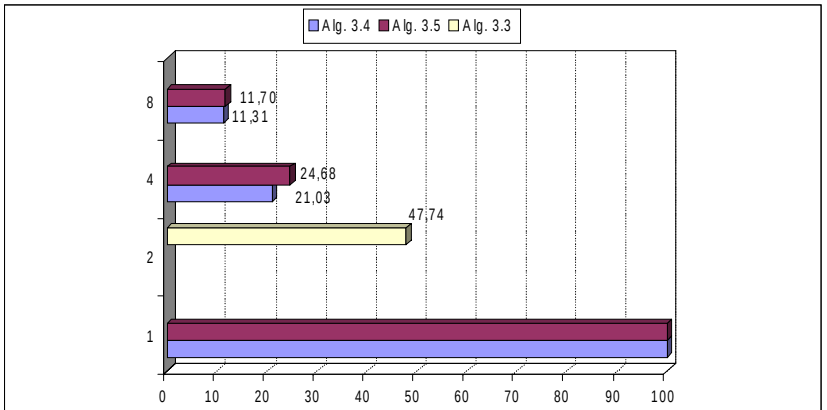
(b) *IBM SP2 ethernet*

Figura 3.7: *Valores de la eficiencia en un IBM SP2*



	1	2	4	8	16	32	64	128	256
Alg.3.3		92,71							
Alg.3.5	100,00		83,66	52,26	26,46	11,85	6,28	2,31	1,01
Alg.3.4	100,00		84,48	66,07	44,50	25,53	17,19	7,40	3,67

(a) *CRAY T3D*



	1	2	4	8
Alg. 3.3		47,74		
Alg. 3.5	100,00		24,68	11,70
Alg. 3.4	100,00		21,03	11,31

(b) *Cluster de Pentiums*

Figura 3.8: *Valores del speedup en un CRAY y un cluster de Pentiums*

Capítulo 4 Métodos divide y vencerás basados en la fórmula de Sherman-Morrison-Woodbury

4.1 Introducción

Consideramos el problema general de obtener la solución del sistema lineal

$$A\boldsymbol{x} = \boldsymbol{d}, \tag{4.1}$$

donde

$$A = \begin{bmatrix} a_1 & b_1 & & & \\ c_2 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & c_n & a_n \end{bmatrix} \qquad \text{y} \qquad \boldsymbol{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix}. \tag{4.2}$$

Suponemos que A es una matriz tridiagonal e irreducible y que existen dos naturales k y p tales que $k = \frac{n}{p}$. Consideramos en A la siguiente partición por bloques

$$A = \begin{bmatrix} A_0 & B_0 & & & \\ C_1 & A_1 & B_1 & & \\ & \ddots & \ddots & \ddots & \\ & & C_{p-2} & A_{p-2} & B_{p-2} \\ & & & C_{p-1} & A_{p-1} \end{bmatrix},$$

donde cada uno de los bloques diagonales

$$A_i = \begin{bmatrix} a_{ik+1} & b_{ik+1} & & & \\ c_{ik+2} & a_{ik+2} & b_{ik+2} & & \\ & \ddots & \ddots & \ddots & \\ & & c_{(i+1)k-1} & a_{(i+1)k-1} & b_{(i+1)k-1} \\ & & & c_{(i+1)k} & a_{(i+1)k} \end{bmatrix}, \qquad i = 0, 1, \dots, p-1, \tag{4.3}$$

es una matriz tridiagonal de tamaño $k \times k$ y, para $i = 0, 1, \dots, p-2$, cada bloque subdiagonal y superdiagonal, respectivamente, tienen la forma

$$C_{i+1} = \left[\begin{array}{cccc|c} 0 & 0 & \cdots & 0 & c_{(i+1)k+1} \\ \hline & & & & 0 \\ & & O & & \vdots \\ & & & & 0 \\ & & & & 0 \end{array}\right] = c_{(i+1)k+1}\boldsymbol{e}_1\boldsymbol{e}_k^T, \qquad B_i = \left[\begin{array}{c|cccc} 0 & & & & \\ 0 & & O & & \\ \vdots & & & & \\ 0 & & & & \\ \hline b_{(i+1)k} & 0 & \cdots & 0 & 0 \end{array}\right] = b_{(i+1)k}\boldsymbol{e}_k\boldsymbol{e}_1^T \tag{4.4}$$

y son de tamaño $k \times k$ con un único elemento no nulo. Los vectores $\boldsymbol{e}_1$ y $\boldsymbol{e}_k$ representan la primera y la última columna, respectivamente, de la matriz identidad I_k.

En los vectores $\boldsymbol{x}$ y $\boldsymbol{d}$ consideramos una partición por bloques conforme con la matriz de coeficientes A, es decir,

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_{p-2} \\ \boldsymbol{x}_{p-1} \end{bmatrix} \qquad \text{y} \qquad \boldsymbol{d} = \begin{bmatrix} \boldsymbol{d}_0 \\ \boldsymbol{d}_1 \\ \vdots \\ \boldsymbol{d}_{p-2} \\ \boldsymbol{d}_{p-1} \end{bmatrix}$$

con

$$\boldsymbol{x}_i = \begin{bmatrix} x_{ik+1} \\ x_{ik+2} \\ \vdots \\ x_{(i+1)k-1} \\ x_{(i+1)k} \end{bmatrix} \qquad \text{y} \qquad \boldsymbol{d}_i = \begin{bmatrix} d_{ik+1} \\ d_{ik+2} \\ \vdots \\ d_{(i+1)k-1} \\ d_{(i+1)k} \end{bmatrix}, \quad \text{para} \quad i = 0, 1, \ldots, p-1.$$

4.2 Un algoritmo general del tipo *divide y vencerás*

4.2.1 Descripción del método

Comenzamos describiendo un algoritmo general de tipo *divide y vencerás* para sistemas tridiagonales cuya idea básica es dividir en bloques el sistema inicial y el vector de términos independientes para resolver en cada procesador ciertos subsistemas tridiagonales. A partir de esas soluciones construimos un sistema tridiagonal auxiliar que nos permite obtener la solución del sistema inicial.

De las expresiones (4.3) y (4.4) tenemos que

$$A = \begin{bmatrix} A_0 & & & & & \\ & A_1 & & & & \\ & & A_2 & & & \\ & & & \ddots & & \\ & & & & A_{p-2} & \\ & & & & & A_{p-1} \end{bmatrix} + \begin{bmatrix} 0 & b_k \boldsymbol{e}_k \boldsymbol{e}_1^T & & & & \\ c_{k+1} \boldsymbol{e}_1 \boldsymbol{e}_k^T & 0 & b_{2k} \boldsymbol{e}_k \boldsymbol{e}_1^T & & & \\ & c_{2k+1} \boldsymbol{e}_1 \boldsymbol{e}_k^T & 0 & b_{3k} \boldsymbol{e}_k \boldsymbol{e}_1^T & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{(p-2)k+1} \boldsymbol{e}_1 \boldsymbol{e}_k^T & 0 & b_{(p-1)k} \boldsymbol{e}_k \boldsymbol{e}_1^T \\ & & & & c_{(p-1)k+1} \boldsymbol{e}_1 \boldsymbol{e}_k^T & 0 \end{bmatrix} = G + UV^T$$

con $G = \mathrm{diag}(A_0, A_1, \ldots, A_{p-1})$ y U, V matrices de tamaño $n \times 2(p-1)$ dadas por

$$U = \begin{bmatrix} \mathbf{0} & b_k\boldsymbol{e}_k & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ c_{k+1}\boldsymbol{e}_1 & \mathbf{0} & \mathbf{0} & b_{2k}\boldsymbol{e}_k & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & c_{2k+1}\boldsymbol{e}_1 & \mathbf{0} & \mathbf{0} & b_{3k}\boldsymbol{e}_k & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & c_{2k+1}\boldsymbol{e}_1 & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & b_{(p-2)k}\boldsymbol{e}_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & c_{(p-2)k+1}\boldsymbol{e}_1 & \mathbf{0} & \mathbf{0} & b_{(p-1)k}\boldsymbol{e}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & c_{(p-1)k+1}\boldsymbol{e}_1 & \mathbf{0} \end{bmatrix}$$

y

$$V = \begin{bmatrix} \boldsymbol{e}_k & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{e}_1 & \boldsymbol{e}_k & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{e}_1 & \boldsymbol{e}_k & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \boldsymbol{e}_1 & \boldsymbol{e}_k & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \boldsymbol{e}_1 \end{bmatrix},$$

donde $\mathbf{0}$ denota una columna nula con k componentes. Utilizando la fórmula de Sherman-Morrison-Woodbury (véase la sección 1.1), podemos calcular la solución $\boldsymbol{x}$ del sistema (4.1) como

$$A^{-1}\boldsymbol{d} = G^{-1}\boldsymbol{d} - G^{-1}U\left(I + V^TG^{-1}U\right)^{-1}V^TG^{-1}\boldsymbol{d}. \tag{4.5}$$

Es fácil comprobar que $I + V^T G^{-1} U =$

$$\begin{bmatrix}
1 & b_k \boldsymbol{e}_k^T A_0^{-1} \boldsymbol{e}_k & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
c_{k+1} \boldsymbol{e}_1^T A_1^{-1} \boldsymbol{e}_1 & 1 & 0 & b_{2k} \boldsymbol{e}_1^T A_1^{-1} \boldsymbol{e}_k & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
c_{k+1} \boldsymbol{e}_k^T A_1^{-1} \boldsymbol{e}_1 & 1 & 0 & b_{2k} \boldsymbol{e}_k^T A_1^{-1} \boldsymbol{e}_k & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
0 & 0 & c_{2k+1} \boldsymbol{e}_1^T A_2^{-1} \boldsymbol{e}_1 & 1 & 0 & b_{3k} \boldsymbol{e}_1^T A_2^{-1} \boldsymbol{e}_k & \cdots & 0 & 0 & 0 & 0 \\
0 & 0 & c_{2k+1} \boldsymbol{e}_k^T A_2^{-1} \boldsymbol{e}_1 & 1 & 0 & b_{3k} \boldsymbol{e}_k^T A_2^{-1} \boldsymbol{e}_k & \cdots & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c_{(p-2)k+1} \boldsymbol{e}_1^T A_{p-2}^{-1} \boldsymbol{e}_1 & 0 & 1 & b_{(p-1)k} \boldsymbol{e}_k^T A_{p-2}^{-1} \boldsymbol{e}_k \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c_{(p-1)k+1} \boldsymbol{e}_k^T A_{p-2}^{-1} \boldsymbol{e}_1 & 0 & 1 & b_{(p-1)k} \boldsymbol{e}_k^T A_{p-2}^{-1} \boldsymbol{e}_k \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & c_{(p-1)k+1} \boldsymbol{e}_1^T A_{p-1}^{-1} \boldsymbol{e}_1 & 1
\end{bmatrix}$$

es una matriz pentadiagonal de tamaño $2(p-1) \times 2(p-1)$ en la que muchos de los elementos de las diagonales superiores e inferiores son nulos; por tanto, si consideramos ahora la matriz de permutación P de tamaño $2(p-1) \times 2(p-1)$ dada por

$$P = \begin{bmatrix}
0 & 1 & 0 & 0 & \cdots & 0 & 0 \\
1 & 0 & 0 & 0 & \cdots & 0 & 0 \\
0 & 0 & 0 & 1 & \cdots & 0 & 0 \\
0 & 0 & 1 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & 0 & 1 \\
0 & 0 & 0 & 0 & \cdots & 1 & 0
\end{bmatrix}$$

tenemos que

$$P + V^T G^{-1} U P = \begin{bmatrix} b_k \boldsymbol{e}_k^T A_0^{-1} \boldsymbol{e}_k & 1 & & & & & & \\ 1 & c_{k+1} \boldsymbol{e}_1^T A_1^{-1} \boldsymbol{e}_1 & b_{2k} \boldsymbol{e}_1^T A_1^{-1} \boldsymbol{e}_k & & & & & \\ & c_{k+1} \boldsymbol{e}_k^T A_1^{-1} \boldsymbol{e}_1 & b_{2k} \boldsymbol{e}_k^T A_1^{-1} \boldsymbol{e}_k & 1 & & & & \\ & & 1 & c_{2k+1} \boldsymbol{e}_1^T A_2^{-1} \boldsymbol{e}_1 & b_{3k} \boldsymbol{e}_1^T A_2^{-1} \boldsymbol{e}_k & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & c_{(p-2)k+1} \boldsymbol{e}_k^T A_{p-2}^{-1} \boldsymbol{e}_1 & b_{(p-1)k} \boldsymbol{e}_k^T A_{p-2}^{-1} \boldsymbol{e}_k & 1 & \\ & & & & & 1 & c_{(p-1)k+1} \boldsymbol{e}_1^T A_{p-1}^{-1} \boldsymbol{e}_1 & \end{bmatrix}$$

es una matriz tridiagonal de tamaño $2(p-1) \times 2(p-1)$. Ahora, teniendo en cuenta que $P = P^{-1}$, podemos escribir la expresión (4.5) como

$$A^{-1}\boldsymbol{d} = G^{-1}\boldsymbol{d} - G^{-1}UP\left(P + V^T G^{-1} U P\right)^{-1} V^T G^{-1}\boldsymbol{d},$$

y por tanto, podemos obtener la solución $\boldsymbol{x}$ del sistema (4.1) mediante el siguiente algoritmo.

Algoritmo 4.1 Un algoritmo general para resolver sistemas tridiagonales utilizando la fórmula de Sherman-Morrison-Woodbury.

1 Resolver el sistema

$$G\boldsymbol{y} = \boldsymbol{d} \tag{4.6}$$

para obtener $\boldsymbol{y} = G^{-1}\boldsymbol{d}$.

2 Resolver el sistema

$$GW = UP \tag{4.7}$$

para obtener $W = G^{-1}UP$.

3 Calcular el vector

$$\boldsymbol{h} = V^T \boldsymbol{y}. \tag{4.8}$$

4 Calcular la matriz

$$B = P + V^T W \tag{4.9}$$

y resolver el sistema

$$B\boldsymbol{z} = \boldsymbol{h} \tag{4.10}$$

para obtener $\boldsymbol{z} = B^{-1}\boldsymbol{h} = (P + V^T W)^{-1}\boldsymbol{h}$.

5 Calcular el vector

$$\boldsymbol{t} = W\boldsymbol{z}. \tag{4.11}$$

6 Calcular el vector

$$\boldsymbol{x} = \boldsymbol{y} - \boldsymbol{t}. \tag{4.12}$$

El algoritmo 4.1 constituye un método del tipo *divide y vencerás* para obtener la solución del sistema (4.1) ya que se divide inicialmente el problema en varios subproblemas que deben ser resueltos independientemente en cada procesador (sistemas (4.6) y (4.7)). Estas soluciones intermedias se utilizan como puente para llegar a la solución final (se construye y resuelve el sistema (4.10) y se calcula la solución por medio de la expresión (4.12)).

Obsérvese que los sistemas (4.6) y (4.7) tienen la misma matriz de coeficientes y distinto vector de términos independientes lo cual suguiere la utilización de la factorización LU como técnica óptima para su resolución. Describimos con detalle un ejemplo que nos permita comprender mejor los pasos que se realizan en el algoritmo 4.1 hasta llegar a la solución del sistema (4.1).

Ejemplo 4.1 *En el sistema (4.1) se considera* $n = 16$,

$$A = \begin{bmatrix} 2 & 1 & & & & & & & & & & & & & & \\ 1 & 3 & 1 & & & & & & & & & & & & & \\ & 2 & 2 & 1 & & & & & & & & & & & & \\ & & 1 & 2 & 1 & & & & & & & & & & & \\ & & & 2 & 4 & 1 & & & & & & & & & & \\ & & & & 1 & 2 & 1 & & & & & & & & & \\ & & & & & -1 & 3 & 2 & & & & & & & & \\ & & & & & & -1 & 3 & 1 & & & & & & & \\ & & & & & & & 1 & 4 & -1 & & & & & & \\ & & & & & & & & 1 & 4 & 1 & & & & & \\ & & & & & & & & & 2 & 4 & 2 & & & & \\ & & & & & & & & & & 1 & 3 & 1 & & & \\ & & & & & & & & & & & -1 & 2 & 1 & & \\ & & & & & & & & & & & & 1 & 2 & -1 & \\ & & & & & & & & & & & & & 1 & 4 & -1 \\ & & & & & & & & & & & & & & 1 & 4 \end{bmatrix} \quad y \quad \boldsymbol{d} = \begin{bmatrix} 3 \\ 5 \\ 5 \\ 4 \\ 7 \\ 4 \\ 4 \\ 3 \\ 4 \\ 6 \\ 8 \\ 5 \\ 2 \\ 2 \\ 4 \\ 5 \end{bmatrix}.$$

Por la forma en que se han tomado la matriz A *y el vector* $\boldsymbol{d}$ *resulta inmediato que la solución* $\boldsymbol{x}$ *del sistema* $A\boldsymbol{x} = \boldsymbol{d}$ *tiene todas sus componentes iguales a* 1.

Supongamos que tomamos $p = 4$, por lo que $k = 4$. Podemos escribir la matriz A como

$$
\begin{aligned}
A &= \begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & A_2 & \\ & & & A_3 \end{bmatrix} + \begin{bmatrix} \mathbf{0} & b_4\boldsymbol{e}_4\boldsymbol{e}_1^T & \mathbf{0} & \mathbf{0} \\ c_5\boldsymbol{e}_1\boldsymbol{e}_4^T & \mathbf{0} & b_8\boldsymbol{e}_4\boldsymbol{e}_1^T & \mathbf{0} \\ \mathbf{0} & c_9\boldsymbol{e}_1\boldsymbol{e}_4^T & \mathbf{0} & b_{12}\boldsymbol{e}_4\boldsymbol{e}_1^T \\ \mathbf{0} & \mathbf{0} & c_{13}\boldsymbol{e}_1\boldsymbol{e}_4^T & \mathbf{0} \end{bmatrix} \\
&= \begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & A_2 & \\ & & & A_3 \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \boldsymbol{e}_4\boldsymbol{e}_1^T & \mathbf{0} & \mathbf{0} \\ 2\boldsymbol{e}_1\boldsymbol{e}_4^T & \mathbf{0} & \boldsymbol{e}_4\boldsymbol{e}_1^T & \mathbf{0} \\ \mathbf{0} & \boldsymbol{e}_1\boldsymbol{e}_4^T & \mathbf{0} & \boldsymbol{e}_4\boldsymbol{e}_1^T \\ \mathbf{0} & \mathbf{0} & -\boldsymbol{e}_1\boldsymbol{e}_4^T & \mathbf{0} \end{bmatrix} \\
&= G + UV^T,
\end{aligned}
$$

donde $G = \operatorname{diag}(A_0, A_1, A_2, A_3)$ es diagonal por bloques con

$$
A_0 = \begin{bmatrix} 2 & 1 & & \\ 1 & 3 & 1 & \\ & 2 & 2 & 1 \\ & & 1 & 2 \end{bmatrix}, \qquad A_1 = \begin{bmatrix} 4 & 1 & & \\ 1 & 2 & 1 & \\ & -1 & 3 & 2 \\ & & -1 & 3 \end{bmatrix}, \qquad A_2 = \begin{bmatrix} 4 & -1 & & \\ 1 & 4 & 1 & \\ & 2 & 4 & 2 \\ & & 1 & 3 \end{bmatrix} \quad y \quad A_3 = \begin{bmatrix} 2 & 1 & & \\ 1 & 2 & -1 & \\ & 1 & 4 & -1 \\ & & 1 & 4 \end{bmatrix}.
$$

Las matrices U y V son de tamaño 16×16 y tienen la forma

$$U = \begin{bmatrix} \mathbf{0} & \boldsymbol{e}_4 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 2\boldsymbol{e}_1 & \mathbf{0} & \mathbf{0} & \boldsymbol{e}_4 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{e}_1 & \mathbf{0} & \mathbf{0} & \boldsymbol{e}_4 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & -\boldsymbol{e}_1 & \mathbf{0} \end{bmatrix}, \qquad V = \begin{bmatrix} \boldsymbol{e}_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{e}_1 & \boldsymbol{e}_4 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{e}_1 & \boldsymbol{e}_4 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{e}_1 & \boldsymbol{e}_4 \end{bmatrix}, \tag{4.13}$$

donde $\mathbf{0} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^T$. *Para este ejemplo, tomamos la matriz de permutación*

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \tag{4.14}$$

que es de tamaño 6×6.

Comenzamos la ejecución del algoritmo 4.1 con la resolución del sistema (4.6). Nótese que realizamos sobre el vector $\boldsymbol{d}$ *una partición conforme con la realizada sobre la matriz de coeficientes* A, *con lo que tendremos los bloques* $\boldsymbol{d}_i$, *para* $i = 0, 1, 2, 3$, *dados por*

$$\boldsymbol{d}_0 = \begin{bmatrix} 3 \\ 5 \\ 5 \\ 4 \end{bmatrix}, \qquad \boldsymbol{d}_1 = \begin{bmatrix} 7 \\ 4 \\ 4 \\ 3 \end{bmatrix}, \qquad \boldsymbol{d}_2 = \begin{bmatrix} 4 \\ 6 \\ 8 \\ 5 \end{bmatrix} \qquad y \qquad \boldsymbol{d}_3 = \begin{bmatrix} 2 \\ 2 \\ 4 \\ 5 \end{bmatrix}.$$

Resolver el sistema (4.6) supone resolver los subsistemas $A_i\boldsymbol{y}_i = \boldsymbol{d}_i$, *para* $i = 0, 1, 2, 3$, *cuyas soluciones son*

$$\boldsymbol{y}_0 = \begin{bmatrix} 0.8571 \\ 1.2857 \\ 0.2857 \\ 1.8571 \end{bmatrix}, \qquad \boldsymbol{y}_1 = \begin{bmatrix} 1.5393 \\ 0.8427 \\ 0.7753 \\ 1.2584 \end{bmatrix}, \qquad \boldsymbol{y}_2 = \begin{bmatrix} 1.2466 \\ 0.9863 \\ 0.8082 \\ 1.3973 \end{bmatrix} \qquad e \qquad \boldsymbol{y}_3 = \begin{bmatrix} 0.3559 \\ 1.2881 \\ 0.9322 \\ 1.0169 \end{bmatrix}.$$

El segundo paso del algoritmo 4.1 consiste en la resolución del sistema (4.7). Antes de resolver este sistema calculamos el producto UP *y determinamos la forma de la matriz* W. *A partir de las expresiones (4.13) y (4.14) obtenemos el producto*

$$UP = \begin{bmatrix} \boldsymbol{e}_4 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & 2\boldsymbol{e}_1 & \boldsymbol{e}_4 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{e}_1 & \boldsymbol{e}_4 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & -\boldsymbol{e}_1 \end{bmatrix}.$$

Teniendo en cuenta la forma de las matrices UP *y* G, *podemos escribir el sistema (4.7) como*

$$\begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & A_2 & \\ & & & A_3 \end{bmatrix} \begin{bmatrix} \boldsymbol{f}_0 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{g}_1 & \boldsymbol{f}_1 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{g}_2 & \boldsymbol{f}_2 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{g}_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_4 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & 2\boldsymbol{e}_1 & \boldsymbol{e}_4 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{e}_1 & \boldsymbol{e}_4 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & -\boldsymbol{e}_1 \end{bmatrix},$$

donde por $\boldsymbol{f}_i$ *y* $\boldsymbol{g}_{i+1}$, *para* $i = 0, 1, 2$ *denotamos los vectores no nulos de 4 componentes de la matriz* $W = G^{-1}UP$. *De esta forma, resolver el sistema (4.7) consiste en resolver los subsistemas*

$$A_1\boldsymbol{g}_1 = c_5\boldsymbol{e}_1, \quad A_2\boldsymbol{g}_2 = c_9\boldsymbol{e}_1, \quad A_3\boldsymbol{g}_3 = c_{13}\boldsymbol{e}_1,$$
$$A_0\boldsymbol{f}_0 = b_4\boldsymbol{e}_4, \quad A_1\boldsymbol{f}_1 = b_8\boldsymbol{e}_4, \quad A_2\boldsymbol{f}_2 = b_{12}\boldsymbol{e}_4,$$

cuyas soluciones, para este ejemplo, vienen dadas por

$$\boldsymbol{f}_0 = \begin{bmatrix} -0.1429 \\ 0.2857 \\ -0.7143 \\ 0.8571 \end{bmatrix}, \quad \boldsymbol{f}_1 = \begin{bmatrix} -0.0225 \\ 0.0899 \\ -0.1573 \\ 0.2809 \end{bmatrix}, \quad \boldsymbol{f}_2 = \begin{bmatrix} 0.0137 \\ 0.0548 \\ -0.2329 \\ 0.4110 \end{bmatrix} \quad y \quad \boldsymbol{g}_1 = \begin{bmatrix} 0.5618 \\ -0.2472 \\ -0.0674 \\ -0.0225 \end{bmatrix}, \quad \boldsymbol{g}_2 = \begin{bmatrix} 0.2329 \\ -0.0685 \\ 0.0411 \\ -0.0137 \end{bmatrix}, \quad \boldsymbol{g}_3 = \begin{bmatrix} -0.6441 \\ 0.2881 \\ -0.0678 \\ 0.0169 \end{bmatrix}.$$

El siguiente paso es la formación del vector $\boldsymbol{h}$ *que, de acuerdo con la expresión (4.8) es*

$$\boldsymbol{h} = \begin{bmatrix} 1.8571 \\ 1.5393 \\ 1.2584 \\ 1.2466 \\ 1.3973 \\ 0.3559 \end{bmatrix}.$$

Ahora, a partir de las soluciones de los sistemas (4.7) y las matrices V *y* P*, construimos* B *por medio de la expresión (4.9), obteniendo*

$$B = \begin{bmatrix} 0.8571 & 1.0000 & 0 & 0 & 0 & 0 \\ 1.0000 & 0.5648 & -0.0225 & 0 & 0 & 0 \\ 0 & -0.0225 & 0.2809 & 1.0000 & 0 & 0 \\ 0 & 0 & 1.0000 & 0.2329 & 0.0137 & 0 \\ 0 & 0 & 0 & -0.0137 & 0.4110 & 1.0000 \\ 0 & 0 & 0 & 0 & 1.0000 & -0.6441 \end{bmatrix}.$$

Ahora resolvemos, una vez calculados B y $\boldsymbol{h}$ el sistema (4.10), cuya solución es

$$\boldsymbol{z} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix},$$

que utilizamos en la expresión (4.11) para obtener el vector $\boldsymbol{t} =$

$$\begin{bmatrix} -0.1429 & 0.2857 & -0.7143 & 0.8571 & 0.5393 & -0.1573 & -0.2247 & 0.2584 & 0.2466 & -0.0137 & -0.1918 & 0.3973 & -0.6441 & 0.2881 & -0.0678 & 0.0169 \end{bmatrix}^T.$$

Finalmente, utilizando la expresión (4.12) obtenemos la solución del sistema (4.1),

$$\boldsymbol{x} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}^T.$$

Una vez estudiado con detalle este caso particular, consideramos ahora el caso general en el que disponemos de una máquina con p procesadores y A es una matriz de tamaño $n \times n$.

En general, podemos resumir los pasos 1 y 2 del algoritmo 4.1 diciendo que se deben resolver los subsistemas

$$A_i \begin{bmatrix} \boldsymbol{y}_i & \boldsymbol{f}_i & \boldsymbol{g}_{i+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{d}_i & b_{(i+1)k}\boldsymbol{e}_k & c_{(i+1)k+1}\boldsymbol{e}_1 \end{bmatrix}, \qquad \text{para} \quad i = 0, 1, \ldots, p-2. \tag{4.15}$$

En cuanto a la paralelización del método hay que señalar que la carga de cálculo en cada procesador no está totalmente equilibrada, puesto que los procesadores centrales resuelven tres subsistemas, mientras que el primer y último procesador únicamente resuelven dos subsistemas.

Para calcular el vector $\boldsymbol{h}$ de la expresión (4.8) y teniendo en cuenta la forma particular en que se ha construido la matriz V y el vector $\boldsymbol{y}$, no es necesario efectuar ninguna operación ya que basta con elegir las componentes adecuadas de los vectores $\boldsymbol{y}_i$. En general,

$$\boldsymbol{h} = \begin{bmatrix} \boldsymbol{y}_0(k) \\ \boldsymbol{y}_1(1) \\ \boldsymbol{y}_1(k) \\ \boldsymbol{y}_2(1) \\ \boldsymbol{y}_2(k) \\ \vdots \\ \boldsymbol{y}_{p-2}(1) \\ \boldsymbol{y}_{p-2}(k) \\ \boldsymbol{y}_{p-1}(1) \end{bmatrix}, \tag{4.16}$$

donde $\boldsymbol{y}_j(l)$ denota la l-ésima componente del vector $\boldsymbol{y}_j$.

Como se desprende del ejemplo 4.1 la matriz auxiliar B que formamos utilizando la expresión (4.9) es tridiagonal y presenta la característica particular que algunos elementos de las diagonales superior e inferior son unos. En realidad esta matriz se forma tomando ciertas componentes de los vectores solución de los subsistemas (4.15). La forma general de esta matriz viene dada por

$$B = \begin{bmatrix} \boldsymbol{f}_0(k) & 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 1 & \boldsymbol{f}_1(1) & \boldsymbol{g}_1(1) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & \boldsymbol{f}_1(k) & \boldsymbol{g}_1(k) & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & \boldsymbol{f}_2(1) & \boldsymbol{g}_2(1) & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & \boldsymbol{f}_2(k) & \boldsymbol{g}_2(k) & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \boldsymbol{f}_{p-2}(1) & \boldsymbol{g}_{p-2}(1) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \boldsymbol{f}_{p-2}(k) & \boldsymbol{g}_{p-2}(k) & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & \boldsymbol{g}_{p-1}(1) \end{bmatrix}, \tag{4.17}$$

donde $\boldsymbol{f}_j(l)$ y $\boldsymbol{g}_j(l)$ denotan la l-ésima componente de los vectores $\boldsymbol{f}_j$ y $\boldsymbol{g}_j$ respectivamente. Siguiendo con el proceso que nos permita obtener la solución del sistema inicial, una vez construida la matriz B, resolvemos el sistema (4.10) para obtener el vector $\boldsymbol{z}$ que nos permitirá calcular la solución final. En general, desarrollando la expresión (4.11) tendremos

$$\begin{bmatrix} \boldsymbol{t}_0 \\ \boldsymbol{t}_1 \\ \boldsymbol{t}_2 \\ \vdots \\ \boldsymbol{t}_{p-2} \\ \boldsymbol{t}_{p-1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f}_0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{f}_1 & \boldsymbol{g}_1 & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{f}_2 & \boldsymbol{g}_2 & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \boldsymbol{f}_{p-2} & \boldsymbol{g}_{p-2} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \boldsymbol{g}_{p-1} \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ \vdots \\ z_{2p-4} \\ z_{2p-3} \end{bmatrix},$$

que escribimos de forma abreviada como

$$\boldsymbol{t}_0 = \boldsymbol{f}_0 z_0, \qquad \boldsymbol{t}_i = [\boldsymbol{f}_i \quad \boldsymbol{g}_i] \begin{bmatrix} z_{2i-1} \\ z_{2i} \end{bmatrix}, \quad i = 1, 2, \ldots, p-2, \qquad \text{y} \qquad \boldsymbol{t}_{p-1} = \boldsymbol{g}_{p-1} z_{2p-3}.$$

De esta forma, la solución del sistema (4.1) viene dada por $\boldsymbol{x}_i = \boldsymbol{y}_i - \boldsymbol{t}_i$, con $i = 0, 1, \ldots, p-1$.

4.2.2 Coste computacional

Calculamos el coste computacional del algoritmo 4.1. Los sistemas (4.6) y (4.7) tienen la misma matriz de coeficientes y diferentes vectores de términos independientes. Esta característica nos sugiere la descomposición LU como método óptimo para su resolución.

Como ya vimos en la sección 1.1, por medio de los algoritmos 1.2 y 1.3, la resolución del sistema (4.6) mediante el cálculo de la descomposición LU requiere un total de $8n - 7$ flops. Por otra parte, resolver el sistema (4.7) es equivalente a resolver los subsistemas (4.15). Teniendo en cuenta la forma particular de los vectores de términos independientes de estos subsistemas, su resolución requiere un total de $6n - 4$ flops (véase la sección 1.1 para una descripción más detallada del número de operaciones requeridas para este cálculo). En consecuencia, la resolución de los sistemas (4.6) y (4.7) representa un coste de $14n - 11$ flops.

Por la propia construcción del método es fácil comprobar que la formación del vector $\boldsymbol{h}$ y la matriz B, definida mediante la expresión (4.9), no requiere ninguna operación algebraica, ya que basta con tomar las componentes adecuadas de las soluciones de los sistemas (4.6) y (4.7), como se aprecia en las expresiones (4.16) y (4.17).

La resolución del sistema (4.10) por el método de Gauss requiere $8(2p-2) - 7$ flops, ya que B es una matriz tridiagonal de tamaño $(2p-2) \times (2p-2)$. Debido a la estructura de la matriz W, para calcular el vector $\boldsymbol{t}$ son necesarios $3n - 4k$ flops. Finalmente, para el cálculo de las componentes de la solución mediante la expresión (4.12) se requieren n flops.

En consecuencia, el coste total del algoritmo 4.1 es de

$$18n - 4k + 16p - 34 \quad \text{flops.} \tag{4.18}$$

4.3 Un algoritmo BSP del tipo *divide y vencerás* general

Consideremos ahora que disponemos de una máquina paralela con p procesadores. Nótese que, debido a la forma en que se divide inicialmente la matriz A en p bloques de tamaño $k \times k$, los elementos de la matriz B se encuentran distribuidos en los distintos procesadores de manera que los elementos de la primera fila se encuentran en el procesador P_0, los elementos de las filas $2i-1$ y $2i$ se encuentran en el procesador P_i, para $i = 1, 2 \ldots, p-2$, mientras que el último procesador tiene los elementos de la última fila. Esta característica será esencial posteriormente en la elección del método paralelo para la resolución de este sistema auxiliar.

Basándonos en las características del método expuestas por medio del algoritmo 4.1 podemos realizar diversas implementaciones según el modelo BSP. Las diferencias esenciales de las distintas implementaciones que se realizan en las siguientes secciones se basan en la forma en la que se resuelve el sistema tridiagonal auxiliar (4.9) y el modelo de comunicación utilizado para la transmisión de datos entre procesadores.

4.3.1 Descripción del método

La primera implementación que consideramos se basa en la resolución del sistema (4.9) del algoritmo 4.1 en cada uno de los procesadores de forma simultánea y secuencial. Se considera la división por bloques dada por las expresiones (4.3) y (4.4). Las características esenciales de este método se recogen en el siguiente algoritmo BSP.

Algoritmo 4.2 Algoritmo BSP general para sistemas tridiagonales.

Superpaso 1

El procesador P_0 envía al procesador P_i, para $i = 1, 2, \ldots, p-1$, los bloques A_i, $\boldsymbol{d}_i$ y los elementos b_{ik} y c_{ik+1}.

Superpaso 2

1 Resolver los sistemas (4.15) utilizando la factorización LU de la matriz de coeficientes A_i.

2 Enviar desde el procesador P_i, para $i = 0, 1, \ldots, p-1$, los elementos $\boldsymbol{y}_i(1)$, $\boldsymbol{y}_i(k)$, $\boldsymbol{f}_i(1)$, $\boldsymbol{f}_i(k)$, $\boldsymbol{g}_i(1)$, $\boldsymbol{g}_i(k)$ a todos los procesadores.

Superpaso 3

1 Formar el vector $\boldsymbol{h}$ y la matriz B mediante las expresiones (4.16) y (4.17) en cada procesador. Resolver el sistema $B\boldsymbol{z} = \boldsymbol{h}$ utilizando el método de Gauss.

2 Calcular $\boldsymbol{f}_i$ y $\boldsymbol{x}_i$ en el procesador P_i, para $i = 0, 1, \ldots, p-1$, mediante las expresiones (4.11) y (4.12).

3 Enviar desde el procesador P_i, para $i = 1, 2, \ldots, p-1$, las soluciones parciales $\boldsymbol{x}_i$ al procesador P_0.

La figura 4.1 sintetiza las tareas que deben realizarse en cada procesador al ejecutar el algoritmo 4.2 para obtener la solución del sistema inicial, suponiendo que $p = 4$.

Debemos notar ciertas características particulares de esta implementación en paralelo. En primer lugar, en el superpaso 3, cada procesador realiza al mismo tiempo la formación del vector $\boldsymbol{h}$, la matriz B y la resolución del sistema tridiagonal auxiliar (4.10). Posteriormente, todos los procesadores trabajan en paralelo para el cálculo de las soluciones parciales, que son enviadas al procesador P_0 al final del superpaso.

En cuanto al tipo de comunicaciones que se realizan a lo largo del algoritmo, es interesante resaltar que las comunicaciones del superpaso 2 representan un *broadcast* donde todos los procesadores comunican elementos al resto de procesadores. Sin embargo, este *broadcast* es de muy pocos elementos, por lo que no representa un alto coste de comunicación.

Parece lógico plantear una modificación de este algoritmo donde cada procesador P_i comunique al procesador principal sus elementos $\boldsymbol{y}_i(1)$, $\boldsymbol{y}_i(k)$, $\boldsymbol{f}_i(1)$, $\boldsymbol{f}_i(k)$, $\boldsymbol{g}_i(1)$, $\boldsymbol{g}_i(k)$ y el procesador P_0 obtenga las soluciones finales a partir de la solución del sistema (4.10). Sin embargo, esto no es posible ya que para el cálculo del vector solución $\boldsymbol{x}$ mediante las expresiones (4.11) y (4.12), son necesarias todas las componentes de los vectores $\boldsymbol{f}_i$ y $\boldsymbol{g}_i$. Esto significa que todos los procesadores centrales deben enviar al procesador principal $2k$

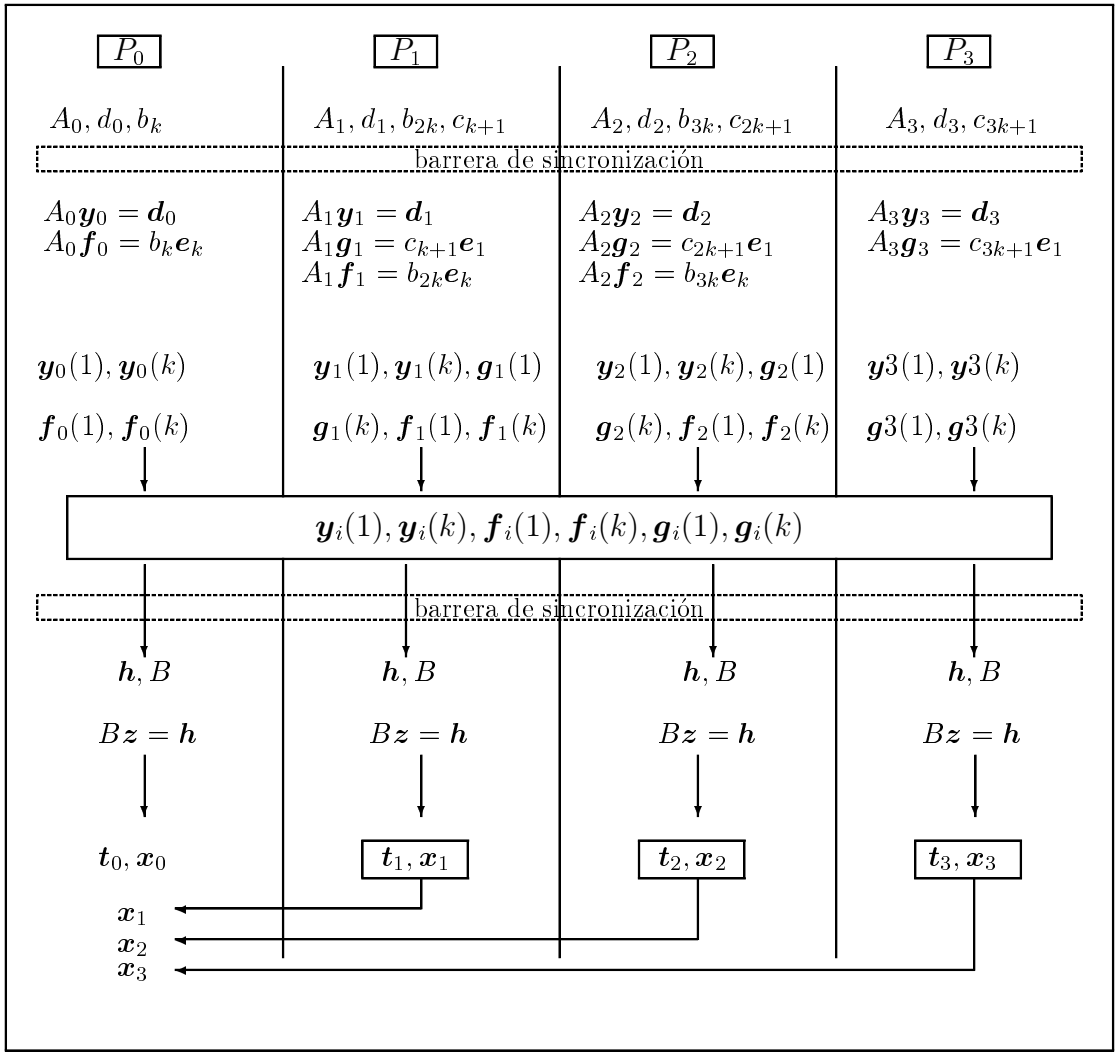


Figura 4.1: *Esquema de ejecución del algoritmo 4.2, para 4 procesadores.*

componentes correspondientes a estos vectores, salvo para el último procesador que envía las k componentes del vector $\boldsymbol{f}_{p-1}$. Este esquema de comunicación representa un coste de

$$[2k(p-2)+k]g = (2n-3k)g \quad \text{flops},$$

mientras que el coste de realizar un *broadcast* de 6 elementos entre p procesadores, es de $6pg$ flops. Como normalmente $p \ll n$, se cumple que $6p \ll 2n-3k$, por lo que el coste de realizar el *broadcast* es mucho menor. De esta forma podemos afirmar que el proceso de comunicación que se lleva a cabo en el superpaso 2 resulta óptimo desde el punto de vista del coste computacional.

4.3.2 Coste computacional

En esta sección calculamos el coste computacional del algoritmo 4.2 calculando los costes individuales de cada uno de los superpasos que componen el algoritmo.

Coste del superpaso 1. Cada procesador recibe un bloque cuadrado de tamaño $k \times k$ de la matriz de coeficientes, con $k = \frac{n}{p}$ y el bloque correspondiente del vector $\boldsymbol{d}$. Además, cada procesador recibe dos elementos fuera de los bloques diagonales, salvo el último que recibe un único elemento. Esto representa un coste de $[4k(p-1)+2p-3]g$ flops, lo que significa que el coste del superpaso 1 es de

$$(4n-4k+2p-3)\,g + l \quad \text{flops}. \tag{4.19}$$

Coste del superpaso 2. Debemos calcular el coste aritmético que supone la resolución de los sistemas (4.15). Los procesadores P_i, para $i = 1, 2, \ldots, p-2$, son los que más operaciones realizan, ya que deben resolver tres sistemas. Subdividimos las tareas que se realizan en los procesadores P_i, para $i = 1, 2, \ldots, p-2$, en varias etapas que describimos brevemente a continuación.

Como vimos en la sección 1.1, la resolución de un sistema tridiagonal mediante la descomposición LU con una matriz de tamaño $n \times n$ supone $8n-7$ flops, por lo que resolver el sistema $G_i\boldsymbol{y}_i = \boldsymbol{d}_i$ mediante la descomposición LU requiere un total de $8k-7$ flops. Resolver

los sitemas del tipo $G_i\boldsymbol{g}_i = c_{ik+1}\boldsymbol{e}_1$ y $G_i\boldsymbol{f}_i = b_{(i+1)k}\boldsymbol{e}_k$ suponen un total de $4k-3$ y $2k-1$ flops, respectivamente, como ya se comentó en dicha sección. En consecuencia, se requieren $14k-11$ flops para resolver los subsistemas (4.15) en cualquiera de los procesadores P_i, para $i = 1, 2, \ldots, p-2$.

En cuanto al coste de comunicación, hay que tener en cuenta que cada procesador P_i, para $i = 1, 2, \ldots, p-2$, envía los elementos

$$\boldsymbol{y}_i(1), \quad \boldsymbol{y}_i(k), \quad \boldsymbol{f}_i(1), \quad \boldsymbol{f}_i(k), \quad \boldsymbol{g}_i(1), \quad \boldsymbol{g}_i(k)$$

al resto de procesadores, mientras que el procesador P_0 envía los elementos

$$\boldsymbol{y}_0(1), \quad \boldsymbol{y}_0(k), \quad \boldsymbol{g}_0(1), \quad \boldsymbol{g}_0(k)$$

al resto de procesadores y el procesador P_{p-1} envía al resto de procesadores los elementos

$$\boldsymbol{y}_{p-1}(1), \quad \boldsymbol{y}_{p-1}(k), \quad \boldsymbol{f}_{p-1}(1), \quad \boldsymbol{f}_{p-1}(k).$$

En consecuencia, cada procesador no extremo realiza un *broadcast* al resto de procesadores de 6 unidades de datos. Esto significa que el número máximo de unidades de datos enviados o recibidos por cualquiera de los procesadores P_i, para $i = 1, 2, \ldots, p-2$, es de $6(p-1)$, lo que representa el modelo de comunicación de una $6(p-1)$-relación. Así, el coste de comunicación de este superpaso es de $6(p-1)g$ flops. Sumando los costes aritmético y de comunicación de este superpaso obtenemos que el coste total es de

$$14k - 11 + 6(p-1)g + l \quad \text{flops.} \tag{4.20}$$

Coste del superpaso 3. En cuanto al coste aritmético debemos considerar el coste de las distintas tareas que se llevan a cabo en cada uno de los procesadores a lo largo del superpaso. Notar que todos los procesadores realizan las mismas tareas simultáneamente.

La formación de la matriz B de la expresión (4.9) y del vector $\boldsymbol{h}$ de la expresión (4.10) no requiere cálculo aritmético, ya que ambos se forman eligiendo de forma adecuada ciertas componentes de los vectores solución de los sistemas (4.15).

Para resolver el sistema $B\boldsymbol{z} = \boldsymbol{h}$ cuya matriz de coeficientes es de tamaño$(2p-2) \times (2p-2)$, podemos utilizar el método de reducción de Gauss lo cual representa un coste total de $8(2p-2) - 7$ flops. Para calcular el vector $\boldsymbol{t}_i$ en el procesador P_i utilizando la expresión (4.11), cada procesador realiza $3m$ operaciones. Para calcular el vector $\boldsymbol{x}_i = \boldsymbol{y}_i - \boldsymbol{t}_i$ en el procesador P_i utilizando la expresión (4.12) cada procesador realiza k operaciones. En consecuencia, sumando los costes de cada una de estas fases, se obtiene que el coste aritmético de este superpaso es de $4k + 16p - 23$ flops.

El coste de comunicación viene dado por la comunicación de las k componentes del vector $\boldsymbol{x}_i$ desde el procesador P_i, para $i = 1, 2, \ldots, p-1$, al procesador P_0. Esto significa que el procesador principal recibe un número máximo de $k(p-1)$ datos, lo que representa una $k(p-1)$-relación, por lo que el coste de comunicación de este superpaso es de $[k(p-1)]g$ flops.

Sumando los costes aritmético y de comunicación, obtenemos un coste para este superpaso de

$$4k + 16p - 23 + (n-k)g + l \quad \text{flops.} \tag{4.21}$$

Sumando las expresiones (4.19), (4.20) y (4.21) se obtiene el coste computacional del algoritmo 4.2, que es de

$$18k + 16p - 34 + (5n - 5k + 8p - 9)\, g + 3l \quad \text{flops.} \tag{4.22}$$

4.4 Algoritmos BSP *divide y vencerás* utilizando el método *recursive doubling*

El algoritmo 4.2 funciona bien cuando $p \ll n$. Ello se debe fundamentalmente a que la resolución del sistema $B\boldsymbol{z} = \boldsymbol{h}$ del punto 4 del algoritmo 4.1 se realiza simultáneamente en cada procesador. Si el número de procesadores es alto, por ejemplo, $p = 256$, la matriz de coeficientes B es de tamaño 510×510, con lo que estamos resolviendo en cada procesador simultáneamente un sistema tridiagonal de tamaño 510×510. Esto supone una pérdida de eficiencia en el proceso de cálculo. Por ello, podemos utilizar algún algoritmo adecuado para resolver en paralelo el sistema (4.10), de manera que todos los procesadores participen en su resolución. En lo que resta de capítulo

supondremos que el número de procesadores es $p = 2^m$, con $m \geq 1$. Este valor, como veremos en el transcurso de esta sección, juega un papel importante en la implementación que se realiza.

4.4.1 Descripción del método

De entre los numerosos métodos que existen para resolver sistemas tridiagonales (véase la sección 1.3), uno de los que mejor aprovecha las características particulares de la matriz B de la expresión (4.17) es el método *recursive doubling*, descrito en la sección 1.3.2. Por tanto, parece razonable pensar en implementar un método del tipo divide y vencerás que resuelva el sistema auxiliar (4.10) en paralelo utilizando dicho método. El nuevo algoritmo se diferencia básicamente del algoritmo 4.2 en los siguientes puntos

(i) En el superpaso 2 del algoritmo 4.2, después de resolver los sistemas (4.15), en cada procesador tenemos los elementos de las filas $(2i-1)$-ésimas y $2i$-ésimas de B y $\boldsymbol{h}$, para $i = 0, 1, \ldots, p-1$, teniendo en cuenta que las filas -1 y $2p-2$ de la matriz B y las componentes -1 y $2p-2$ del vector $\boldsymbol{h}$ son nulas.

(ii) El paso de comunicación del superpaso 2 se elimina.

(iii) A partir del superpaso 2, se aplica el método *recursive doubling* para resolver el sistema (4.10).

Introducimos la notación $B[i \mid j]$, con $j < i$, para indicar el producto de matrices

$$B[i \mid j] = B_i B_{i-1} \cdots B_{j-1} B_j, \tag{4.23}$$

donde las matrices B_i son las que se definen en la sección 1.3.2 mediante las expresiones (1.9) y (1.10). Nótese que en la sección 1.3.2 se describe el método del *recursive doubling* para un sistema con coeficientes a_i, b_i y c_i en su matriz de coeficientes y componentes d_i en el vector de términos independiente. Ahora, estos coeficientes han cambiado ya que resolvemos el sistema $B\boldsymbol{z} = \boldsymbol{h}$, cuyo tamaño es

$(2p-2)\times(2p-2)$ y donde la matriz B y el vector $\boldsymbol{h}$ vienen dados por las expresiones (4.17) y (4.16), respectivamente. En consecuencia, las matrices

$$B_i = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad \text{para} \quad i = 0, 1, 2, \ldots, 2p-3$$

quedan definidas por los elementos

$$\alpha_i = -\frac{a_i}{b_i}, \qquad \beta_i = -\frac{c_i}{b_i} \qquad \text{y} \quad \gamma_i = \frac{d_i}{b_i},$$

donde los coeficientes a_i, b_i y c_i vienen definidos de la siguiente forma

$$\begin{array}{llll} a_0 = \boldsymbol{f}_0(k), & a_{2i} = \boldsymbol{g}_i(k), & a_{2i-1} = \boldsymbol{f}_i(1), & a_{2p-3} = \boldsymbol{g}_{p-1}(1), \\ b_0 = 1, & b_{2i} = 1, & b_{2i-1} = \boldsymbol{g}_i(1), & \\ & c_{2i} = \boldsymbol{f}_i(k), & c_{2i-1} = 1, & c_{2p-3} = 1, \\ d_0 = \boldsymbol{y}_0(k), & d_{2i} = \boldsymbol{y}_i(k), & d_{2i-1} = \boldsymbol{y}_i(1), & d_{2p-3} = \boldsymbol{y}_{p-1}(1), \end{array}$$

para $i = 0, 1, \ldots, \lfloor \frac{2p-5}{2} \rfloor$.

El algoritmo que implementamos ahora coincide exactamente con el algoritmo 4.2 en los superpasos 1 y 2, salvo en la comunicación final del superpaso 2. Dicha comunicación se suprime y comienza la ejecución en paralelo del método *recursive doubling* para calcular la solución del sistema (4.10).

Las características del algoritmo 4.2 determinan la forma en que debe ejecutarse en paralelo dicho método para este caso concreto. Así, después de resolver los sistemas (4.15), se dispone en el procesador P_i, para $i = 1, 2, \ldots, p-2$, de los elementos necesarios para el cálculo de las matrices B_{2i-1} y B_{2i}. En los procesadores P_0 y P_{p-1} se encuentran los elementos necesarios para el cálculo de las

matrices B_0 y B_{2p-3} respectivamente. Por comodidad en la notación, podemos generalizar el resultado diciendo que el procesador P_i, para $i = 0, 1, \ldots, p-1$, calcula las matrices B_{2i-1} y B_{2i}, teniendo en cuenta que tomamos $B_{-1} = B_{2p-2} = I_3$. La razón de esta elección es que cuando en el siguiente paso del algoritmo multipliquemos las matrices B_{2i-1} y B_{2i} para obtener la matriz $B[2i \mid 2i-1]$, entonces tengamos los productos adecuados, es decir, $B[0 \mid -1] = B_0$ y $B[2p-2 \mid 2p-3] = B_{2p-3}$, que son las matrices que necesitamos para obtener las componentes de la solución.

El modelo de comunicación que utilizamos para este algoritmo es el que sugieren Sun, Zhang y Ni [106] en la implementación del algoritmo *Parallel Prefix*, en el que existen $\log_2 p + 1$ pasos de comunicación en paralelo utilizando p procesadores. Un ejemplo del modelo de comunicaciones que se realizan cuando $p = 16$ se muestra en la figura 4.2. En dicha figura se utiliza la notación $B[i \mid j]$ dada por (4.23) y se muestran detalladamente los pasos de cálculo y comunicación necesarios para la obtención en el procesador P_{p-1} de la matriz $B_[30 \mid -1]$, que nos permitirá calcular z_0 y el resto de componentes del vector $\boldsymbol{z}$. Se observa que son necesarios cuatro pasos de comunicación entre los procesadores para llegar a la solución en el último procesador.

Notemos que, de acuerdo con la posición en que se encuentran los elementos del sistema (4.10) en los distintos procesadores, en el procesador P_i, para $i = 1, 2, \ldots, p-2$, podemos calcular dos componentes del vector $\boldsymbol{z}$ de la solución, mientras que en el procesador P_0 podemos calcular la primera componente de $\boldsymbol{z}$, es decir, z_0 y en el procesador P_{p-1} la última componente de $\boldsymbol{z}$, es decir, z_{2p-3}. Nótese que en este caso, por coherencia con la notación empleada, la expresión (1.11) se convierte en

$$\boldsymbol{z}_{i+1} = B_i \boldsymbol{z}_i, \qquad \text{para } \ i = 0, 1, 2, \ldots, 2p-3, \tag{4.24}$$

siendo $\boldsymbol{z}_i = \begin{bmatrix} z_i \\ z_{i-1} \\ 1 \end{bmatrix}$.

La expresión (4.24) nos permite el cálculo de ciertas componentes del vector $\boldsymbol{z}$ en función de las anteriores. En consecuencia, de igual modo que en la sección 1.3 se dedujo la expresión (1.13) para obtener unas componentes en función del vector inicial $\boldsymbol{x}_0$, ahora podemos

P_0	B_{-1}				
P_1	B[2 | 1]	B[2 | -1]			
P_2	B[4 | 3]	B[4 | 1]	B[4 | -1]		
P_3	B[6 | 5]	B[6 | 3]	B[6 | -1]		
P_4	B[8 | 7]	B[8 | 5]	B[8 | 1]	B[8 | -1]	
P_5	B[10 | 9]	B[10 | 7]	B[10 | 3]	B[10 | -1]	
P_6	B[12 | 11]	B[12 | 9]	B[12 | 5]	B[12 | -1]	
P_7	B[14 | 13]	B[14 | 11]	B[14 | 7]	B[14 | -1]	
P_8	B[16 | 15]	B[16 | 13]	B[16 | 9]	B[16 | 1]	B[16 | -1]
P_9	B[18 | 17]	B[18 | 15]	B[18 | 11]	B[18 | 3]	B[18 | -1]
P_{10}	B[20 | 19]	B[20 | 17]	B[20 | 13]	B[20 | 5]	B[20 | -1]
P_{11}	B[22 | 21]	B[22 | 19]	B[22 | 15]	B[22 | 7]	B[22 | -1]
P_{12}	B[24 | 23]	B[24 | 21]	B[24 | 17]	B[24 | 9]	B[24 | -1]
P_{13}	B[26 | 25]	B[26 | 23]	B[26 | 19]	B[26 | 11]	B[26 | -1]
P_{14}	B[28 | 27]	B[28 | 25]	B[28 | 21]	B[28 | 13]	B[28 | -1]
P_{15}	B[30 | 29]	B[30 | 27]	B[30 | 23]	B[30 | 15]	B[30 | -1]

Figura 4.2: *Esquema de comunicaciones del método divide y vencerás basado en el método recursive doubling, para* $p = 16$.

escribir la expresión

$$\boldsymbol{z}_{i+1} = B[i \mid 0]\boldsymbol{z}_0, \qquad \text{para } \ i = 0, 1, 2, \ldots, 2p-3, \tag{4.25}$$

siendo $\boldsymbol{z}_0 = \begin{bmatrix} z_0 \\ 0 \\ 1 \end{bmatrix}$, que nos permite calcular las componentes del vector $\boldsymbol{z}$ en función del vector $\boldsymbol{z}_0$. Ahora, la componente inicial que calculamos es z_0 y es la que nos permite calcular las restantes componentes mediante un proceso recursivo.

Siguiendo el esquema de comunicaciones que se muestra en la figura 4.2, el cálculo de la componente z_0 se realiza en el último procesador mediante una expresión análoga a la expresión (1.14) pero que, de acuerdo con la notación introducida, se escribe como

$$\boldsymbol{z}_{2p-2} = B[2p-3 \mid -1]\boldsymbol{z}_0, \tag{4.26}$$

Como el resto de componentes se obtienen a partir de z_0, debe haber un paso de comunicación en el que el procesador P_{p-1} comunica al resto de procesadores esta variable. De esta manera, una vez que cada procesador tiene almacenado en su memoria el valor de z_0, puede realizar el cálculo de las componentes de la solución que le corresponden. Así, podemos considerar el siguiente algoritmo que resume las características anteriormente expuestas.

Algoritmo 4.3 Algoritmo BSP divide y vencerás utilizando el método *recursive doubling*.

Superpaso 1

El procesador P_0 envía al procesador P_i, para $i = 1, 2, \ldots, p-1$, los bloques A_i, $\boldsymbol{d}_i$ y los elementos b_{ik} y c_{ik+1}. Inicialmente tomamos $\mu = 1$.

Superpaso 2

1 El procesador P_i, para $i = 0, 1, \ldots, p-1$,

1.1 resuelve los sistemas dados por (4.15) que le corresponden, utilizando la factorización LU de la matriz de coeficientes,

1.2 calcula las matrices B_{2i-1} y B_{2i},

1.3 calcula el producto $B[2i \mid 2i-1]$.

2 Comunicación.

2.1 El procesador P_i, para $i = 0, 1, \ldots, p-2$, comunica al procesador P_{i+1} la matriz $B[2i \mid 2i-1]$.

Superpaso $m - q + 2$, para $q = m-1, m-2, \ldots, 2, 1$

1 El procesador P_i, para $i = 2^{m-q-1}, 2^{m-q-1}+1, 2^{m-q-1}+2, \ldots, p-2, p-1$, calcula la matriz $B[2i \mid 2i-(2\mu+1)]$.

2 El procesador P_i, para $i = 0, 1, \ldots, (p-1) - 2^{m-q}$, comunica al procesador $P_{i+2^{m-q}}$ la matriz $B[2i \mid 2i-(2\mu+1)]$.

3 Actualizamos μ, como $\mu = 2\mu + 1$.

Superpaso $m + 2$

1 El procesador P_i, para $i = 2^{m-1}, 2^{m-1}+1, 2^{m-1}+2, \ldots, p-2, p-1$, calcula la matriz $B[2i \mid 2i-(2\mu+1)]$.

2 El procesador P_{p-1} calcula z_0 utilizando la expresión (4.26).

3 El procesador P_{p-1} comunica z_0 al resto de procesadores.

Superpaso $m + 3$

1 El procesador P_i, para $i = 1, 2, \ldots, p-1$ calcula z_{2i}, z_{2i-1}, mediante la expresión (4.25).

2 El procesador P_i, para $i = 0, 1, \ldots, p-1$,

2.1 calcula $\boldsymbol{t}_i$ y $\boldsymbol{x}_i$ mediante las expresiones (4.11) y (4.12), respectivamente,

2.2 envía las soluciones $\boldsymbol{x}_i$ al procesador principal.

Debemos notar que a lo largo de todo el algoritmo, cuando nos referimos a una matriz $B[i \mid j]$, con $j < -1$, tomamos $B[i \mid -1]$. Nótese la diferencia existente entre $B[i \mid -1]$ y B_{-1}. Hemos tomado como criterio de notación $B_{-1} = I_3$, mientras que la notación $B[i \mid -1]$ nos indica el producto de matrices $B_i B_{i-1} \cdots B_0 B_{-1}$.

Analizando este algoritmo con detalle, observamos que efectuamos comunicaciones y realizamos productos de matrices que no son necesarias para llegar a la obtención de la componente z_0 de la solución del sistema (4.10), a partir de la cual se obtiene el resto de componentes de la solución. El modelo de comunicación que seguimos tiene como objetivo el cálculo de z_0 en el último procesador y su comunicación al resto de procesadores para que calculen las componentes de las soluciones que le corresponden con el fin de obtener la solución final por bloques en cada procesador. El cálculo de z_{2i-1} y z_{2i} en el procesador P_i se realiza mediante la expresión (4.25).

Podemos plantearnos un nuevo algoritmo basado en un modelo de comunicación distinto al que se utiliza en el algoritmo 4.3. El modelo de comunicación que se utiliza ahora es de tipo *fan-in*, que intenta minimizar el número de elementos que circulan por la red y el número de procesadores que comunican elementos en cada superpaso, con lo que se reduce el coste computacional. La diferencia básica respecto al esquema de comunicación usual que presenta el algoritmo 4.3 es que en este nuevo algoritmo el cálculo de las componentes z_i, para $i = 0, 1, \ldots, 2p-3$, no se realiza en paralelo por cada procesador; ahora, el procesador P_0 es el encargado de calcular todas las componentes z_i a partir de la componente inicial z_0. Esto va a disminuir el número de comunicaciones en los distintos superpasos y el número de procesadores que comunican elementos, aunque requiere una primera comunicación de las matrices B_i, para $i = 0, 1, \ldots, 2p-3$ desde todos los procesadores al procesador principal. Esta comunicación es necesaria para el cálculo de las componentes del vector $\boldsymbol{z}$ a partir de la componente z_0. Teniendo en cuenta estas modificaciones, implementamos el siguiente algoritmo.

Algoritmo 4.4 Algoritmo BSP divide y vencerás utilizando el método *recursive doubling* modificado.

Superpaso 1

El procesador P_0 envía al procesador P_i, para $i = 1, 2, \ldots, p-1$, los bloques A_i, $\boldsymbol{d}_i$ y los elementos b_{ik} y c_{ik+1}. Inicialmente tomamos $\mu = -1$.

Superpaso 2

1 El procesador P_i, para $i = 0, 1, \ldots, p-1$,

1.1 resuelve los sistemas dados por (4.15) que le corresponden, utilizando la factorización LU de la matriz de coeficientes,

1.2 calcula las matrices B_{2i-1} y B_{2i},

1.3 calcula el producto $B[2i \mid 2i-1]$.

2 Comunicación,

2.1 el procesador P_i, para $i = 1, 2, \ldots, p-1$, comunica al procesador P_0 las matrices B_{2i-1} y B_{2i},

2.2 el procesador P_{2i+1}, para $i = 0, 1, \ldots, 2^{r-1}-1$, comunica al procesador P_{2i} la matriz $B[2i \mid 2i-1]$.

Superpaso $m - q + 2$, para $q = m-1, m-2, \ldots, 2, 1$

1 El procesador $P_{2^{m-q}i}$, para $i = 0, 1, \ldots, 2^q - 1$ calcula la matriz

$$B[2^{m-q} + 2^{m-q+1}i + 2(\mu+1) \mid 2^{m-q+1}i - 1].$$

2 El procesador $P_{2^{m-q}+2^{m-q+1}i}$, para $i = 0, 1, \ldots, \lfloor \frac{2^q-1}{2} \rfloor$ comunica al procesador $P_{2^{m-q+1}i}$ la matriz

$$B[2^{m-q} + 2^{m-q+1}(2i+1) + 2(\mu+1) \mid 2^{m-q+1}(2i+1) - 1].$$

3 Actualizamos μ, como $\mu = 2(\mu+1)$.

Superpaso $m + 2$

El procesador P_0,

1 calcula la matriz $B[2p-2 \mid -1]$,

2 calcula z_0 utilizando la expresión (4.26) y, mediante la expresión (4.25), obtiene el resto de componentes z_i, para $i = 1, 2, \ldots, 2p-3$,

3 comunica las componentes z_{2i-1}, z_{2i} al procesador P_i, para $i = 1, 2, \ldots, p-1$.

Superpaso $m + 3$

El procesador P_i, para $i = 0, 1, \ldots, p-1$,

1 calcula $\boldsymbol{t}_i$ y $\boldsymbol{x}_i$ mediante las expresiones (4.11) y (4.12), respectivamente,

2 envía las soluciones $\boldsymbol{x}_i$ al procesador principal.

La figura 4.3 muestra un ejemplo del algoritmo 4.3 para $p = 8$. En este ejemplo se recogen todas las comunicaciones que se realizan hasta obtener la solución. Debemos notar que en el procesador P_0 se deben formar las matrices B_{-1} y B_0.

Hay que observar también que, de acuerdo con el algoritmo 4.4, el cálculo de las componentes de la solución del sistema (4.10) se realiza en el procesador principal, una vez calculada previamente la componente z_0 de la misma, que sirve como punto de partida de la ecuación recurrente (4.24). Como el cálculo de las componentes z_i, para $i = 1, 2, \ldots, 2p-3$, se realiza secuencialmente en el procesador inicial, únicamente se necesitan las matrices B_i que se forman en el superpaso 2, lo que significa que el procesador principal debe contener en su memoria los elementos de las matrices B_i, para $i = 1, 2, \ldots, 2p-3$. Esto supone que en el superpaso 2 existe una fase de comunicación donde cada procesador comunica 6 elementos al procesador principal, lo que representa un coste de $6(p-1)$ flops. Esta es la razón de que aparezca dicha comunicación al final del superpaso 2.

4.4.2 Coste computacional

En esta sección calculamos los costes computacionales de los algoritmos 4.3 y 4.4 que se basan en la resolución del sistema tridiagonal (4.10) por medio del método *recursive doubling*.

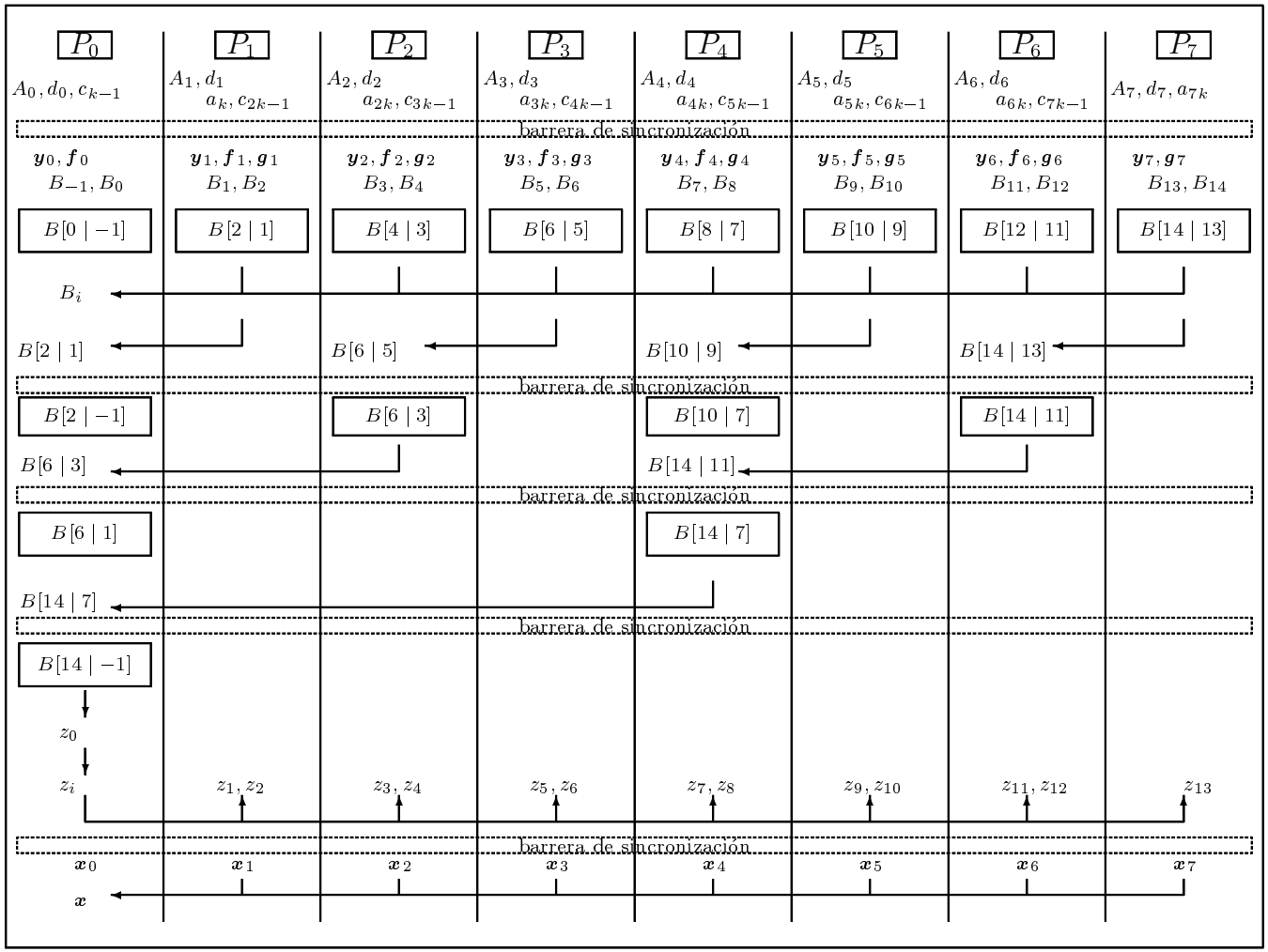


Figura 4.3: *Esquema de ejecución del algoritmo 4.4, para 8 procesadores.*

Comenzamos calculando de forma detallada el coste del algoritmo 4.3.

Coste del superpaso 1. El superpaso 1 del algoritmo 4.3 es análogo al superpaso 1 del algoritmo 4.2, por lo que su coste viene dado por la expresión (4.19).

Coste del superpaso 2. La primera tarea que se realiza es la descomposición LU y la resolución de los sistemas (4.15). Esto requiere un total de $14k-11$ flops. Posteriormente, se produce la formación de las matrices B_{2i-1}, B_{2i} en el procesador P_i, para $i = 0, 1, \ldots, p-1$. Debido a la forma particular que tienen estas matrices, sólo los procesadores centrales necesitan efectuar 3 divisiones. Por consiguiente, esta fase requiere 3 flops. Finalmente, se calcula el producto matricial $B[2i \mid 2i-1]$ en el procesador P_i. Esta operación requiere un total de 5 flops, debido a la estructura particular de dichas matrices, por lo que el coste de esta fase es de 5 flops.

En cuanto al coste de comunicación, cada procesador P_i, para $i = 0, 1, \ldots, p-2$, comunica al procesador P_{i+1} una matriz de tamaño 3×3, de la forma

$$\begin{bmatrix} * & * & * \\ * & * & * \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.27}$$

Nos encontramos ante un patrón de comunicación en el que cada procesador, salvo el principal, tiene un mensaje de longitud 6 unidades entrando y saliendo de él, es decir, una 6-relación. Este modelo de comunicación supone un coste de $6g$ flops, como ya se comentó en el capítulo 2.

Por lo tanto, el coste total del superpaso 2 es de

$$14k - 3 + 6g + l \quad \text{flops}. \tag{4.28}$$

Coste del superpaso $\boldsymbol{m-q+2}$, para $q = m-1, m-2, \ldots, 2, 1$. En estos $m-1$ superpasos se repite el mismo esquema de

computación y de comunicación. Por esta razón calculamos globalmente los costes aritmético y de comunicación.

En cuanto a los cálculos aritméticos, cada procesador que se encuentra activo en el superpaso realiza un producto de dos matrices de la forma (4.27). El cálculo de un producto de dos matrices de esta forma requiere un total de 20 flops, por lo que el coste aritmético de estos $m-1$ superpasos es de $20(m-1)$ flops.

En cuanto a la comunicación, cada procesador activo efectúa una comunicación de 6 elementos al final de cada superpaso, por lo que en cada uno de estos superpasos los procesadores activos reciben o comunican un máximo de 6 elementos. En consecuencia, el modelo de comunicación es el correspondiente a una 6-relación. Como se realizan $m-1$ superpasos el coste global de comunicación es de $6(m-1)g$ flops.

En consecuencia, sumando los costes aritmético y de comunicación tendremos que el coste de estos $m-1$ superpasos es de

$$20(m-1) + 6(m-1)g + (m-1)l \quad \text{flops.} \tag{4.29}$$

Coste del superpaso $m+2$. Dividimos el coste aritmético en dos fases. La primera fase corresponde con la obtención de la matriz $B[2i \mid -1]$ en el procesador P_i, para $i = 2^{m-1}, 2^{m-1}+1, \ldots, p-1$, lo que supone 20 flops. La segunda fase consiste en el cálculo de z_0 en P_{p-1}, a partir de la matriz $B[2p-2 \mid -1]$, lo que únicamente representa 1 división. El coste aritmético de este superpaso es, por lo tanto, de 21 flops.

En cuanto al coste de comunicación, el procesador P_{p-1} envía al resto de procesadores la componente z_0, por lo que el coste es de $(p-1)g$ flops.

Sumando el coste aritmético y de comunicación obtenemos el coste del superpaso, que es de

$$21 + (p-1)g + l \quad \text{flops.} \tag{4.30}$$

Coste del superpaso $m+3$. Inicialmente cada procesador P_i, para $i = 1, 2, \ldots, p-2$ calcula las componentes z_{2i-1} y z_{2i}, lo que

supone un coste aritmético de 10 flops.

El resto del superpaso coincide exactamente con la última fase del superpaso 3 del algoritmo 4.2, por lo que su coste será de $4k+(n-k)g$ flops. En consecuencia, el coste del superpaso será de

$$4k + 10 + (n - k)g + l \quad \text{flops.} \tag{4.31}$$

Sumando las expresiones (4.19), (4.28), (4.29), (4.30) y (4.31) obtenemos el coste total del algoritmo 4.3, que es de

$$18k + 20m + 8 + (5n - 5k + 6m + 3p - 10)g + (m + 3)l \quad \text{flops.} \tag{4.32}$$

Seguidamente procedemos de forma análoga calculando el coste computacionel del algoritmo 4.4.

Coste del superpaso 1. Al igual que sucedía con el algoritmo 4.3, este superpaso es análogo al superpaso 1 del algoritmo 4.2. Por tanto, el coste del superpaso viene dado por la expresión (4.19).

Coste del superpaso 2. El coste aritmético de este superpaso es análogo al coste aritmético del superpaso 2 del algoritmo 4.3, es decir, $14k - 11$ flops. En cuanto al coste de comunicación, en primer lugar, el procesador P_i, para $i = 1, 2, \ldots, p - 1$, comunica al procesador P_0 las matrices B_{2i-1}, B_{2i}. Los procesadores no extremos envían dos matrices, mientras que el procesador P_{p-1} envía la matriz B_{2p-3}. Por cada matriz que se envía deben comunicarse 3 elementos. Esto significa que el procesador P_0 recibe un total de $6(p-2)+3 = 6p-9$ elementos, por lo que el coste es de $(6p-9)g$ flops. En segundo lugar, cada procesador impar comunica al procesador par contiguo una matriz de tamaño 3×3, de la forma (4.27). Esto representa nuevamente una 6-relación, con un coste de $6g$ flops. En consecuencia, el coste total de comunicación de este superpaso es de

$$(6p - 3)g \quad \text{flops.} \tag{4.33}$$

Sumando los costes aritmético y de comunicación obtenemos el coste total del superpaso 2, que es de

$$14k - 3 + (6p - 3)g + l \quad \text{flops.} \tag{4.34}$$

Coste del superpaso $m - q + 2$, para $q = m-1, m-2, \ldots, 2, 1$. Como vimos al calcular el coste del algoritmo 4.3, cada procesador activo en el superpaso realiza un producto de dos matrices de la forma (4.27), lo que supone 20 flops. En cuanto a la comunicación, cada procesador que comunica datos al final de cada superpaso, únicamente comunica 6 elementos. El modelo de comunicación que se sigue en estos $m - 1$ superpasos es análogo al del algoritmo 4.3, por lo que el coste de comunicación es de $6(m - 1)g$ flops.

En consecuencia, el coste de estos $m - 1$ superpasos es de

$$20(m - 1) + 6(m - 1)g + (m - 1)l \quad \text{flops.} \tag{4.35}$$

Coste del superpaso $m + 2$. Dividimos el coste aritmético en varias fases, de acuerdo con las tareas que se realizan. Obtener la matriz $B[2p - 2 \mid -1]$ requiere 20 flops, mientras que el cálculo de z_0, a partir de la matriz $B[2p - 2 \mid -1]$ sólamente requiere 1 división. Para el cálculo del resto de componentes de la solución se utiliza la expresión (4.25). Debido a la forma de estas matrices, para obtener la componente z_{i+1} sólamente multiplicamos la primera fila de B_i por z_i, lo que representa un total de 5 flops por cada componente. En total, se necesitan $5(2p - 3) = 10p - 15$ flops. El coste aritmético de este superpaso es, por lo tanto, de $10p + 6$ flops.

En cuanto al coste de comunicación, el procesador P_0 envía un total de $2p - 3$ elementos, por lo que el coste es de $(2p - 3)g$ flops.

Sumando el coste aritmético y de comunicación obtenemos el coste del superpaso, que es

$$10p + 6 + (2p - 3)g + l \quad \text{flops.} \tag{4.36}$$

Coste del superpaso $m + 3$. Las operaciones y comunicaciones que se realizan en este superpaso coinciden exactamente con las operaciones y comunicaciones que se realizan en los últimos dos puntos del superpaso 3 del algoritmo 4.2. El coste aritmético era de $4k$

flops, mientras que el coste de comunicación era de $(n-k)g$ flops, por lo que el coste de este superpaso será de

$$4k + (n-k)g + l \quad \text{flops.} \tag{4.37}$$

Sumando las expresiones (4.19), (4.34), (4.35), (4.36) y (4.37) y realizando las simplificaciones adecuadas obtenemos el coste total del algoritmo 4.4, que es de

$$18k + 20m + 10p - 27 + (5n - 5k + 6m + 10p - 15)\, g + (m+3)l \quad \text{flops.} \tag{4.38}$$

4.5 Algoritmo BSP *divide y vencerás* basado en el método de reducción cíclica

Otro de los métodos de resolución de sistemas tridiagonales que se ajusta perfectamente a las características del sistema (4.10) es el de reducción cíclica. Véase la sección 1.3.3 para una descripción más detallada del mismo. Podemos implementar un nuevo algoritmo BSP del tipo *divide y vencerás* utilizando las características del método de reducción cíclica para resolver en paralelo el sistema (4.10).

4.5.1 Descripción del método

Recordemos que la forma general de la matriz de coeficientes del sistema (4.10) viene dada por la expresión (4.17), siendo ésta de tamaño $(2p-2) \times (2p-2)$. Además, los elementos de esta matriz y las componentes del vector de términos independientes $\boldsymbol{h}$ están distribuidos de manera que el procesador principal tiene los coeficientes de la primera ecuación, el segundo procesador tiene los coeficientes de las dos siguientes ecuaciones, el tercer procesador los coeficientes de las dos siguientes y así sucesivamente hasta que el último procesador tiene las componentes de la última ecuación. Esta particular situación de los elementos en los procesadores determina la implementación del método, basada en la resolución del sistema (4.10) en paralelo mediante el método de la reducción cíclica, siguiendo

los pasos que se describen con detalle en la sección 1.3.3. La forma del sistema (4.10) así como las operaciones elementales que se desarrollan sobre los coeficientes de esas ecuaciones y sus resultados se muestran mediante las expresiones (1.24), (1.27) y (1.28). Señalar que, de acuerdo con las expresiones (1.28), se requieren un total de 12 flops para realizar una reducción cíclica con cada conjunto de tres ecuaciones, descritas por la expresión (1.25).

Analizando con detalle el método expuesto en dicha sección, notamos que no es necesario resolver ningún sistema de ecuaciones, sino únicamente realizar operaciones elementales sobre los coeficientes de los sistemas que se forman. En consecuencia establecemos la siguiente notación. Denotamos por $E[z, 2i-2 : 2i : 1] = h_{2i-1}$ a una ecuación con unos ciertos coeficientes en las variables z_{2i-2}, z_{2i-1} y z_{2i} y término independiente h_{2i-1}. Así, escribimos la variable, el subíndice de la primera variable de la ecuación, el subíndice de la última variable de la ecuación y el salto de los subíndices.

Utilizando esta notación, podemos generalizar la situación particular de los coeficientes de las ecuaciones del sistema (4.10) diciendo que el procesador P_i, para $i = 1, 2, \ldots, p-2$, tiene los coeficientes de las ecuaciones

$$E[z, 2i-2 : 2i : 1] = h_{2i-1} \tag{4.39}$$

$$E[z, 2i-1 : 2i+1 : 1] = h_{2i}, \tag{4.40}$$

mientras que el procesador P_0 tiene los coeficientes de la ecuación

$$E[z, 0 : 1 : 1] = h_0 \tag{4.41}$$

y el procesador P_{p-1} los coeficientes de la ecuación

$$E[z, 2p-4 : 2p-3 : 1] = h_{2p-3}. \tag{4.42}$$

Esta distribución de los coeficientes de las ecuaciones en los distintos procesadores obliga a realizar un paso inicial de comunicación, ya que para poder aplicar el método de reducción cíclica son necesarias ternas consecutivas de ecuaciones a las que se aplica la reducción.

En nuestro caso, inicialmente sólo disponemos de los coeficientes de dos ecuaciones por cada procesador como máximo, lo que obliga a realizar un paso de comunicación en el que cada procesador P_i, para $i = 1, 2, \ldots, p-2$, comunica al procesador P_{i+1} la ecuación (4.40), mientras que el procesador P_0 comunica al procesador P_1 la ecuación (4.41). Con esta comunicación conseguimos que todos los procesadores, salvo el principal y el último, dispongan de los elementos necesarios para poder comenzar a aplicar la fase inicial de la reducción cíclica dada por las expresiones (1.27) y (1.28). El procesador principal no va a realizar operaciones para obtener la variable central. El último procesador únicamente tiene dos ecuaciones adyacentes, lo que significa que sólamente realiza la primera operación elemental de la reducción.

Podemos dividir la implementación del método en dos fases claramente diferenciadas: en la primera se obtiene, mediante repetidas reducciones cíclicas, la componente central de la solución, es decir $z_{2^{m-1}}$, mediante un proceso de tipo *fan-in*. En la segunda fase se obtienen las restantes componentes de la solución a partir de $z_{2^{m-1}}$, siguiendo un proceso del tipo *fan-in*.

La figura 4.4 nos muestra un ejemplo del método descrito anteriormente para el caso $p = 8$. Para este ejemplo son necesarios ocho superpasos para ejecutar el algoritmo. En general, si trabajamos con $p = 2^m$ procesadores, necesitaremos efectuar un total de $2(m+1)$ superpasos.

Notemos que cada procesador necesita calcular ciertas componentes de la solución del sistema dado por (1.24) para posteriormente obtener la solución del sistema inicial. Más concretamente, el procesador P_i, para $i = 1, 2, \ldots, p-2$, necesita calcular las componentes z_{2i-1} y z_{2i}, mientras que el procesador P_0 calcula la componente z_0 y el procesador P_{p-1} calcula la componente z_{2p-3}. Después del cálculo de la variable intermedia $z_{2^{m-1}}$, mediante la expresión (1.19), esta variable debe ser enviada desde el procesador $P_{2^{m-1}}$ a ciertos procesadores para calcular nuevas componentes de la solución. En la figura 4.4 aparecen únicamente las comunicaciones que son imprescindibles para el cálculo de las componentes en superpasos posteriores.

Resumiendo las cacacterísticas expuestas a lo largo de esta sección se puede implementar el siguiente algoritmo.

Algoritmo 4.5 Algoritmo BSP divide y vencerás utilizando el método de la reducción cíclica.

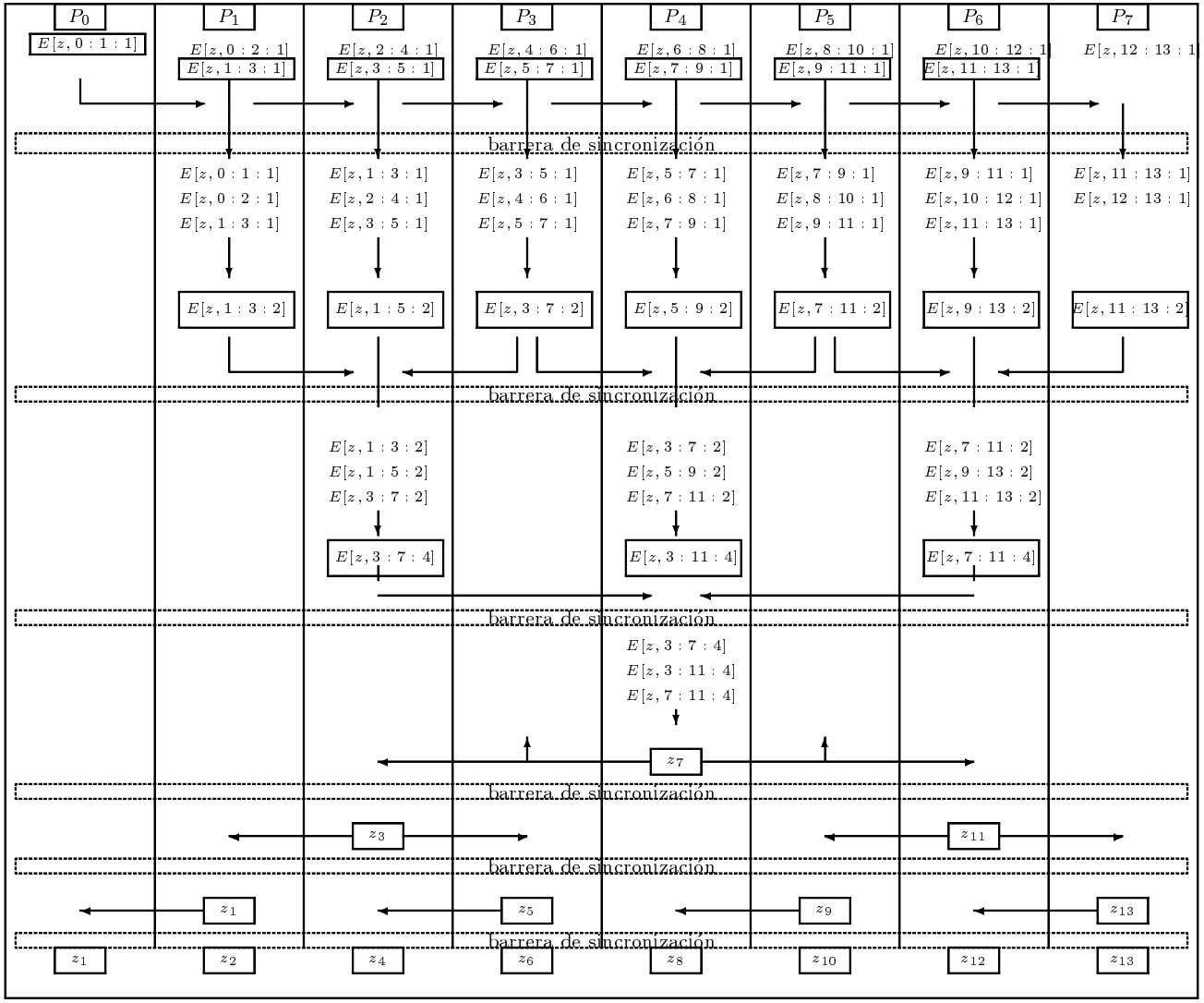


Figura 4.4: *Esquema de ejecución del algoritmo (4.5), para 8 procesadores.*

Superpaso 1

El procesador P_0 envía al procesador P_i, para $i = 1, 2, \ldots, p-1$, los bloques A_i, $\boldsymbol{d}_i$ y los elementos b_{ik} y c_{ik+1}.

Superpaso 2

1 Cálculo. El procesador P_i, para $i = 0, 1, \ldots, p-1$, resuelve los sistemas (4.15) que le corresponden, utilizando la factorización LU de la matriz de coeficientes.

2 Comunicación. El procesador P_i, para $i = 1, 2, \ldots, p-2$, comunica al procesador P_{i+1} los coeficientes de la ecuación (4.40), mientras que el procesador P_0 comunica al procesador P_1 los coeficientes de la ecuación (4.41).

Superpaso $\boldsymbol{m-q+2}$, para $q = m-1, m-2, \ldots, 2, 1$

1 Cálculo. El procesador P_δ, siendo $\delta = 2^{m-q+1}i$, para $i = 1, 2, \ldots, 2^{q+1}-1$, calcula los coeficientes de la ecuación

$$E[2\delta - 1 - 2^{m-q} : 2\delta - 1 + 2^{m-q} : 2^{m-q}] = h_{2\delta-1}^{(m-q)}. \tag{4.43}$$

2 Comunicación. El procesador $P_{2^{m-q}i}$, para $i = 1, 2, \ldots, 2^q - 1$, recibe de los procesadores $P_{2^{m-q}i \pm 2^{m-q-1}}$ los coeficientes de la ecuación reducida (4.43).

Superpaso $\boldsymbol{m+2}$

1 Cálculo. El procesador P_δ, siendo $\delta = 2^{m-1}$ calcula la componente $z_{2\delta-1}$ de la solución.

2 Comunicación. El procesador $P_{2^{m-1}}$ comunica al procesador $P_{\delta \pm j}$, para $j = 1, 2, \ldots, 2^{m-2}$ su componente $z_{2\delta-1}$.

Superpaso $\boldsymbol{m+l}$, para $l = 3, 4, \ldots, m+1$

1 Cálculo. El procesador P_δ, siendo $\delta = 2^{m-1} \pm 2^{m-2} \pm \ldots \pm 2^{m-(l-1)}$, calcula la componente $z_{2\delta-1}$ de la solución.

2 Comunicación. El procesador P_δ comunica al procesador $P_{\delta\pm j}$, para $j = 1, 2, \ldots, 2^{m-l}$ su componente $z_{2\delta-1}$.

Superpaso $2m + 2$

1 El procesador P_i, para $i = 0, 1, \ldots, p-1$, calcula su componente z_{2i}.

2 El procesador P_i, para $i = 0, 1, \ldots, p-1$,

2.1 calcula $\boldsymbol{t}_i$ y $\boldsymbol{x}_i$ mediante las expresiones (4.11) y (4.12), respectivamente,

2.2 envía las soluciones $\boldsymbol{x}_i$ al procesador principal.

4.5.2 Coste computacional

Calculamos el coste del algoritmo 4.5 de forma análoga a como se ha realizado en las secciones anteriores, sumando los costes de los superpasos individuales.

Coste del superpaso 1. El superpaso 1 es análogo al superpaso 1 del del algoritmo 4.2, por lo que el coste viene dado por la expresión (4.19).

Coste del superpaso 2. En cuanto al coste aritmético, hay que tener en cuenta que la factorización LU y la resolución de los sistemas (4.15) requieren un total de $14k - 11$ flops, como ya vimos en el algoritmo 4.2. Después de resolver estos sistemas, cada procesador tiene los coeficientes de las ecuaciones (4.39), (4.40), (4.41) y (4.42), por lo que no realiza operaciones aritméticas.

En cuanto a la comunicación, cada procesador P_i, para $i = 0, 1, \ldots, p-2$, debe enviar tres coeficientes al procesador siguiente, lo que representa una 3-relación con un coste total de $3g$ flops.

El coste total del superpaso 2 es, por tanto, de

$$14k - 11 + 3g + l \quad \text{flops.} \tag{4.44}$$

Coste del superpaso $m - q + 2$, para $q = m-1, m-2, \ldots, 2, 1$. Como podemos apreciar en la figura 4.4, estos $m-1$ superpasos presentan la misma estructura, ya que en todos hay una fase de computación, en la que se realiza una reducción cíclica y una segunda fase de comunicación. Las operaciones aritméticas que realizan los procesadores que están activos son únicamente las que suponen una reducción cíclica a un conjunto de tres ecuaciones, es decir, 12 flops (véanse las expresiones (1.27) y (1.28) de la sección 1.3). Por consiguiente, el coste aritmético de este conjunto de superpasos es de $12(m-1)g$ flops.

En cuanto a la comunicación, observamos que en cada superpaso sólamente la mitad de los procesadores activos comunican los coeficientes de la ecuación reducida obtenida en la reducción cíclica. Además, cada procesador que comunica lo hace a los dos procesadores contiguos a él que han estado activos en la fase de cálculos aritméticos. Debemos notar que en el superpaso anterior únicamente se comunicaban tres coeficientes porque uno de ellos era la unidad, debido a las características del método. Sin embargo, cuando efectuamos una reducción cíclica sobre el primer conjunto de ecuaciones ya desaparece esa estructura particular, por lo que cuando se comunican las ecuaciones reducidas, deben comunicarse un total de 4 elementos.

En la figura 4.4, observamos que los procesadores pares activos reciben coeficientes de dos ecuaciones de sus procesadores vecinos activos, por lo que recibe cada uno de ellos un total de 8 unidades de datos como máximo. Esta comunicación es una 8-relación. Notemos que para el cómputo del coste de este patrón de comunicación no resulta determinante el número de procesadores que se encuentran activos en cada superpaso. En consecuencia, podemos afirmar que en cada superpaso el modelo de comunicación que se sigue es el de una 8-relación, por lo que el coste de comunicación de estos $m-1$ superpasos es de $8(m-1)g$ flops. Sumando los costes aritmético y de comunicación de estos $m-1$ superpasos se obtiene un coste computacional total de

$$12(m-1) + 8(m-1)g + (m-1)l \quad \text{flops.} \tag{4.45}$$

Coste del superpaso $m + 2$. En el superpaso $m+2$ se realiza una reducción cíclica antes de proceder al cálculo de la componente

central de la solución $z_{2^{m-1}}$ por medio de la expresión (1.19). Esto supone 13 flops, 12 flops correspondientes a la reducción cíclica y 1 flop correspondiente al cálculo de la componente de la solución.

Estudiamos ahora el coste de comunicación. Si observamos de nuevo el ejemplo de la figura 4.4, vemos que el coste de comunicación viene dado por el envío de la componente x_7 desde el procesador P_4 a los procesadores P_i, para $i = 2, 3, 5, 6$. Esto supone un coste de $4g$ flops. En general, el coste de comunicación será de $2^{m-1}g$ flops o, lo que es lo mismo, $\frac{p}{2}g$.

El coste total del superpaso es de

$$13 + \frac{p}{2}g + l \quad \text{flops},$$

Coste del superpaso $\boldsymbol{m + l}$, para $l = 3, 4, \ldots, m + 1$. Las operaciones aritméticas que realizan los procesadores activos consisten en el cálculo de una componente de la solución, utilizando la expresión (1.20).

En cuanto a la comunicación, en todos los superpasos se sigue el mismo esquema: ciertos procesadores activos envían una componente de la solución a otros procesadores. Notemos que en ningún caso se produce un *broasdcast* de elementos de un procesador a todos. Aunque de un superpaso al siguiente aumenta al doble el número de procesadores activos que comunican la componente de la solución calculada, este hecho no modifica el cálculo del coste de comunicación.

En el superpaso 6 de la figura 4.4 los dos procesadores activos, P_2 y P_6, calculan una componente de la solución y la comunican a sus dos procesadores contiguos. Por tanto, el número máximo de elementos enviados o recibidos por un procesador es 2, por lo que el coste de este superpaso será de

$$5 + 2g + l \quad \text{flops}.$$

Siguiendo un razonamiento análogo, podríamos calcular el coste individual de los m superpasos que debemos realizar antes de calcular

la solución final. La suma de los costes de comunicación de estos m superpasos es de

$$\left(2^{m-2} + \cdots + 2^1 + 2^0\right) g \quad \text{flops},$$

que, efectuando las oportunas simplificaciones, es de

$$\left(\frac{p}{2} - 1\right) g \quad \text{flops}.$$

En consecuencia, el coste total de estos superpasos es de

$$5(m-1) + \left(\frac{p}{2} - 1\right) g + (m-1)l \quad \text{flops}. \tag{4.46}$$

Coste del superpaso $2m + 2$. En este superpaso, inicialmente cada procesador P_i, para $i = 0, 1, \ldots, p-1$, debe calcular la componente z_{2i} de la solución parcial. Esto supone 5 flops. Después, cada procesador puede calcular las componentes de la solución final que le corresponden y enviarlas al procesador principal. Esta última parte del superpaso coincide exactamente con la última fase del superpaso 3 del algoritmo 4.2, por lo que el coste total será de

$$4k + 5 + (n-k)g + l \quad \text{flops}. \tag{4.47}$$

Sumando las expresiones (4.19), (4.44), (4.45), (4.46) y (4.47) y realizando las simplificaciones adecuadas obtenemos el coste total del algoritmo 4.5, que es de

$$18k + 17m - 10 + (5n - 5k + 3p + 8m - 12)g + (2m+2)l \quad \text{flops}. \tag{4.48}$$

4.6 Resultados teóricos y comparaciones

En esta sección se presentan los tiempos teóricos de ejecución de los algoritmos 4.2, 4.3, 4.4 y 4.5 sobre las tres máquinas cuya descripción se realizó en el capítulo 1. Los parámetros son los que aparecen en la tabla 2.1.

IBM SP2 switch												
	p											
	2				4				8			
n	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5
512	0.0006	0.0006	0.0006	0.0006	0.0008	0.0010	0.0010	0.0012	0.0010	0.0016	0.0016	0.0020
1024	0.0009	0.0010	0.0010	0.0010	0.0011	0.0014	0.0014	0.0015	0.0014	0.0020	0.0020	0.0024
2048	0.0016	0.0016	0.0016	0.0016	0.0017	0.0020	0.0020	0.0021	0.0021	0.0027	0.0027	0.0031
4096	0.0029	0.0030	0.0030	0.0030	0.0030	0.0033	0.0033	0.0034	0.0034	0.0040	0.0040	0.0044
8192	0.0055	0.0056	0.0056	0.0056	0.0056	0.0059	0.0059	0.0060	0.0062	0.0068	0.0068	0.0072
16384	0.0109	0.0109	0.0109	0.0109	0.0108	0.0111	0.0111	0.0112	0.0116	0.0122	0.0122	0.0126
32768	0.0215	0.0216	0.0216	0.0216	0.0212	0.0215	0.0215	0.0216	0.0226	0.0232	0.0232	0.0236
65536	0.0428	0.0428	0.0428	0.0428	0.0420	0.0423	0.0423	0.0424	0.0444	0.0450	0.0450	0.0454
131072	0.0853	0.0854	0.0854	0.0854	0.0836	0.0839	0.0839	0.0840	0.0881	0.0887	0.0887	0.0891
262144	0.1704	0.1704	0.1704	0.1704	0.1668	0.1671	0.1671	0.1672	0.1756	0.1762	0.1762	0.1766
524288	0.3405	0.3406	0.3406	0.3406	0.3332	0.3334	0.3334	0.3336	0.3504	0.3510	0.3510	0.3514
1048576	0.6808	0.6809	0.6809	0.6809	0.6659	0.6661	0.6662	0.6663	0.7002	0.7008	0.7008	0.7012
2097152	1.3614	1.3614	1.3614	1.3614	1.3313	1.3316	1.3316	1.3317	1.3996	1.4002	1.4002	1.4006
4194304	2.7225	2.7225	2.7225	2.7225	2.6622	2.6625	2.6626	2.6626	2.7986	2.7992	2.7992	2.7996

Tabla 4.1: *Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un IBM SP2 con switch.*

4.6.1 Tiempos en un IBM SP2

La tabla 4.1 resume los tiempos de los algoritmos estudiados en este capítulo para un IBM SP2 con switch.

Observando detenidamente la tabla 4.1 destacamos varios aspectos notables de los cuatro algoritmos estudiados. En primer lugar y, quizás como hecho más destacable, no se aprecian diferencias significativas en el tiempo de ejecución de los cuatro algoritmos. Resulta notable que, independientemente del número de procesadores que se utilicen, las diferencias en cuanto a tiempos son factores del orden

de 10^{-4}, aunque para dos procesadores las diferencias son nulas para muchos valores de n. Estas pequeñas diferencias de tiempo son en todos los casos a favor del algoritmo 4.2, por lo que si hay que buscar un óptimo en esta máquina habría que decir que el algoritmo más rápido es el algoritmo 4.2.

También se observa al pasar de $p = 2$ a $p = 4$ que los tiempos de ejecución de los cuatro algoritmos disminuye ligeramente a partir de $n = 32768$, aunque para el algoritmo 4.2 esta disminución de tiempo se produce ya para $n = 16384$. Sin embargo, al pasar de $p = 4$ a $p = 8$, los tiempos aumentan de forma generalizada en todos los casos.

La tabla 4.2 recoge los tiempos comparados de los algoritmos estudiados en este capítulo para un IBM SP2 con ethernet.

Analizando detenidamente la tabla 4.2 destacamos un comportamiento de los algoritmos estudiados muy similar al comportamiento que presentaron en el caso anterior con switch. Las características generales son similares: las diferencias entre los tiempos son muy ligeras, del orden nuevamente de 10^{-4}, aunque de nuevo vuelven a ser en favor del algoritmo 4.2, que resulta el más rápido de los cuatro.

El hecho quizás más destacable en esta máquina es el considerable incremento en los tiempos al aumentar el número de procesadores. En una máquina de este tipo con unas comunicaciones bastante costosas los algoritmos no presentan un buen comportamiento desde el punto de vista de su paralelización. Prácticamente los tiempos quedan multiplicados en un factor de 4 cada vez que duplicamos el número de procesadores.

4.6.2 Tiempos en un CRAY T3D

Las tablas 4.3 y 4.4 muestra los resultados teóricos esperados en un CRAY T3D, una máquina altamente paralela que dispone de hasta 256 procesadores y cuyos valores de los parámetros aparecen en la tabla 2.1.

Observando la tabla 4.3 destacamos varios aspectos generales. En primer lugar, cabe señalar como característica general que no se

IBM SP2 ethernet												
	p											
	2				4				8			
n	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5
512	0.0069	0.0076	0.0076	0.0076	0.0200	0.0227	0.0229	0.0242	0.0736	0.0797	0.0787	0.0844
1024	0.0116	0.0123	0.0123	0.0123	0.0344	0.0371	0.0373	0.0386	0.1272	0.1334	0.1324	0.1381
2048	0.0209	0.0216	0.0216	0.0216	0.0632	0.0660	0.0661	0.0674	0.2347	0.2408	0.2399	0.2455
4096	0.0395	0.0402	0.0403	0.0403	0.1209	0.1237	0.1238	0.1251	0.4496	0.4558	0.4548	0.4605
8192	0.0768	0.0775	0.0775	0.0775	0.2363	0.2390	0.2392	0.2405	0.8795	0.8857	0.8848	0.8905
16384	0.1514	0.1521	0.1521	0.1521	0.4671	0.4698	0.4699	0.4713	1.7394	1.7456	1.7447	1.7504
32768	0.3005	0.3012	0.3012	0.3012	0.9286	0.9313	0.9314	0.9328	3.4592	3.4654	3.4645	3.4702
65536	0.5987	0.5994	0.5994	0.5994	1.8516	1.8543	1.8545	1.8558	6.8989	6.9050	6.9041	6.9098
131072	1.1951	1.1958	1.1958	1.1958	3.6976	3.7004	3.7005	3.7018	13.7781	13.7843	13.7833	13.7890
262144	2.3880	2.3887	2.3887	2.3887	7.3897	7.3924	7.3926	7.3939	27.5366	27.5427	27.5418	27.5475
524288	4.7738	4.7745	4.7745	4.7745	14.7738	14.7766	14.7767	14.7780	55.0535	55.0597	55.0587	55.0644
1048576	9.5453	9.5460	9.5460	9.5460	29.5421	29.5449	29.5450	29.5463	110.0873	110.0935	110.0926	110.0983
2097152	19.0883	19.0891	19.0891	19.0891	59.0787	59.0814	59.0816	59.0829	220.1550	220.1612	220.1603	220.1660
4194304	38.1744	38.1752	38.1752	38.1752	118.1518	118.1546	118.1547	118.1560	440.2905	440.2967	440.2957	440.3014

Tabla 4.2: *Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un IBM SP2 con ethernet.*

CRAY T3D														
n=2048					n=4096					n=8192				
p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5
2	0.0017	0.0018	0.0018	0.0018	2	0.0034	0.0034	0.0034	0.0034	2	0.0068	0.0068	0.0068	0.0068
4	0.0011	0.0011	0.0011	0.0011	4	0.0021	0.0021	0.0021	0.0021	4	0.0040	0.0041	0.0041	0.0041
8	0.0008	0.0009	0.0009	0.0009	8	0.0015	0.0015	0.0015	0.0016	8	0.0029	0.0029	0.0029	0.0029
16	0.0009	0.0009	0.0009	0.0009	16	0.0014	0.0014	0.0014	0.0014	16	0.0025	0.0025	0.0025	0.0025
32	0.0012	0.0011	0.0009	0.0010	32	0.0017	0.0016	0.0015	0.0015	32	0.0028	0.0027	0.0026	0.0026
64	0.0028	0.0016	0.0011	0.0011	64	0.0028	0.0019	0.0015	0.0015	64	0.0036	0.0028	0.0024	0.0024
128	0.0113	0.0038	0.0016	0.0016	128	0.0090	0.0036	0.0021	0.0021	128	0.0086	0.0042	0.0030	0.0030
256	0.0581	0.0114	0.0026	0.0024	256	0.0403	0.0088	0.0030	0.0028	256	0.0322	0.0083	0.0039	0.0038
n=16384					n=32768					n=65536				
p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5
2	0.0135	0.0136	0.0135	0.0135	2	0.0270	0.0270	0.0270	0.0270	2	0.0540	0.0540	0.0540	0.0540
4	0.0080	0.0080	0.0080	0.0081	4	0.0159	0.0160	0.0160	0.0160	4	0.0318	0.0318	0.0318	0.0319
8	0.0056	0.0056	0.0056	0.0057	8	0.0111	0.0111	0.0111	0.0111	8	0.0220	0.0220	0.0220	0.0220
16	0.0047	0.0047	0.0047	0.0047	16	0.0091	0.0091	0.0091	0.0092	16	0.0180	0.0180	0.0180	0.0180
32	0.0050	0.0049	0.0048	0.0048	32	0.0094	0.0093	0.0092	0.0092	32	0.0182	0.0181	0.0180	0.0180
64	0.0053	0.0046	0.0043	0.0043	64	0.0090	0.0083	0.0081	0.0081	64	0.0165	0.0158	0.0155	0.0155
128	0.0099	0.0060	0.0050	0.0050	128	0.0135	0.0099	0.0089	0.0089	128	0.0211	0.0177	0.0167	0.0167
256	0.0297	0.0096	0.0060	0.0059	256	0.0316	0.0134	0.0101	0.0100	256	0.0388	0.0215	0.0185	0.0184

Tabla 4.3: *Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un CRAY T3D desde* $n = 2048$ *hasta* $n = 65536$.

CRAY T3D														
n=131072					n=262144					n=524288				
p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5
2	0.1079	0.1079	0.1079	0.1079	2	0.2158	0.2158	0.2158	0.2158	2	0.4315	0.4315	0.4315	0.4315
4	0.0636	0.0636	0.0636	0.0636	4	0.1271	0.1271	0.1271	0.1271	4	0.2524	0.2541	0.2541	0.2541
8	0.0438	0.0439	0.0439	0.0439	8	0.0029	0.0029	0.0029	0.0029	8	0.1749	0.1749	0.1749	0.1750
16	0.0356	0.0356	0.0356	0.0357	16	0.0709	0.0710	0.0710	0.0710	16	0.1416	0.1416	0.1416	0.1416
32	0.0358	0.0357	0.0356	0.0357	32	0.0710	0.0709	0.0709	0.0709	32	0.1415	0.1414	0.1414	0.1414
64	0.0314	0.0308	0.0305	0.0305	64	0.0614	0.0607	0.0605	0.0605	64	0.1213	0.1206	0.1204	0.1204
128	0.0367	0.0333	0.0324	0.0324	128	0.0680	0.0646	0.0637	0.0637	128	0.1307	0.1273	0.1264	0.1264
256	0.0549	0.0381	0.0352	0.0351	256	0.0881	0.0715	0.0686	0.0685	256	0.1547	0.1383	0.1354	0.1353
n=1048576					n=2097152					n=4194304				
p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	p	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5
2	0.8629	0.8630	0.8630	0.8630	2	1.7258	1.7258	1.7258	1.7258	2	3.4516	3.4516	3.4516	3.4516
4	0.5080	0.5080	0.5080	0.5080	4	1.0159	1.0159	1.0159	1.0159	4	2.0317	2.0317	2.0317	2.0317
8	0.3497	0.3497	0.3497	0.3497	8	0.6992	0.6992	0.6992	0.6992	8	1.3982	1.3983	1.3983	1.3983
16	0.2829	0.2829	0.2829	0.2830	16	0.5655	0.5656	0.5656	0.5656	16	1.1308	1.1308	1.1308	1.1308
32	0.2825	0.2824	0.2823	0.2824	32	0.5644	0.5643	0.5643	0.5643	32	1.1283	1.1282	1.1282	1.1282
64	0.2411	0.2404	0.2402	0.2402	64	0.4807	0.4800	0.4798	0.4798	64	0.9599	0.9593	0.9590	0.9590
128	0.2560	0.2527	0.2518	0.2518	128	0.5067	0.5034	0.5025	0.5025	128	1.0081	1.0044	1.0039	1.0039
256	0.2883	0.2719	0.2690	0.2689	256	0.5555	0.5392	0.5363	0.5362	256	1.0900	1.0737	1.0708	1.0707

Tabla 4.4: *Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un CRAY T3D desde* $n = 131072$ *hasta* $n = 4194304$.

aprecian grandes diferencias en los tiempos de ejecución de los cuatro algoritmos, salvo en algunos casos concretos que ahora estudiamos. Las tablas nos muestran que para un número de procesadores comprendido entre 2 y 16 no existen diferencias apreciables en los tiempos de ejecución, independientemente del valor de n. Cuando el número de procesadores aumenta hasta 32 comienzan a detectarse pequeñas diferencias de tiempo para valores pequeños de n, más concretamente hasta $n = 4096$. En este caso, el algoritmo más costoso es el algoritmo 4.2, mientras que los mejores tiempos se obtienen para los algoritmos 4.4 y 4.5, contrariamente a lo que ocurría en el IBM SP2, en el que los mejores tiempos se obtenían para el algoritmo 4.2. Para $n \geq 4096$, los tiempos vuelven a ser prácticamente idénticos.

Cuando el número de procesadores aumenta hasta 64, 128 y 256, se observa un comportamiento totalmente análogo al descrito para 32 procesadores; las diferencias de tiempo para valores pequeños de n va aumentando progresivamente y el valor de n para el que se detectan diferencias de tiempo crece cada vez más. Así, por ejemplo, si para $p = 32$ y $n = 16384$ la diferencia entre los tiempos del algoritmo 4.2 y del algoritmo 4.5 es de $2 \cdot 10^{-4}$, esta diferencia aumenta hasta $238 \cdot 10^{-4}$ para $p = 256$ y el mismo valor de n. Las diferencias de tiempos son mayores cuanto más pequeño es n. También se observa que al aumentar el valor de n, las diferencias entre los cuatro algoritmos se va reduciendo. Por ejemplo, si para $p = 256$ y $n = 16384$ ya hemos visto que la diferencia entre los algoritmos 4.2 y 4.5 era de $238 \cdot 10^{-4}$, la diferencia cuando $n = 4194304$ es de $193 \cdot 10^{-4}$.

Observamos que, a diferencia de lo que ocurría en el IBM SP2, siempre que existen diferencias en los tiempos, éstas son en favor del algoritmo 4.5, por lo que podemos decir que en una máquina paralela del tipo CRAY T3D el algoritmo óptimo es el 4.5. Recordemos que el algoritmo 4.5 utilizaba el método de reducción cíclica para resolver un sistema tridiagonal auxiliar, a partir del cual se obtenía la solución general.

Otro aspecto que diferencia el comportamiento de estos algoritmos en esta máquina respecto del comportamiento en el IBM SP2 es la reducción en los tiempos al aumentar el número de procesadores. En una máquina altamente paralela como es el CRAY T3D en el que las comunicaciones no son muy costosas, se advierte que al aumentar el número de procesadores de 2 a 4 y de 4 a 8 se produce una reducción en los tiempos generalizada para cualquier valor de n. A partir de 16 procesadores también se produce un descenso en los tiempos pero ya no de forma generalizada para todos los tamaños. Se observa que a medida que el valor de n es mayor, el número de procesadores para el que se produce una reducción de tiempos al duplicar los procesadores es más alto. Así, por ejemplo, para $n = 1048576$ cuando

CRAY T3D				
n	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5
2048	8	16	16	16
4096	16	16	16	16
8192	16	16	64	64
16384	16	64	64	64
32768	64	64	64	64
65536	64	64	64	64
131072	64	64	64	64
262144	64	64	64	64
524288	64	64	64	64
1048576	64	64	64	64
2097152	64	64	64	64
4194304	64	64	64	64

Tabla 4.5: *Número óptimo de procesadores en un CRY T3D para valores de n comprendidos entre $n = 2048$ y $n = 4194304$.*

el número de procesadores es de 64 los tiempos para el algoritmo 4.5, que es el más rápido, es de 0.2402 mientras que si aumentamos el número de procesadores a 128, el tiempo de ejecución del mismo algoritmo es de 0.2518, lo que representa un ligero incremento. Por lo tanto, para este tamaño de matriz, los mejores resultados de tiempo se obtienen trabajando con 64 procesadores.

En consecuencia, no podemos afirmar de forma rotunda que existe un número óptimo de procesadores para la ejecución de estos algoritmos; como se desprende de las tablas 4.3 y 4.4 el número de procesadores óptimo depende del valor de n. La tabla 4.5 recoge el número óptimo de procesadores para los distintos valores de n cuando se ejecutan los diferentes algoritmos.

Como se desprende de la tabla 4.5, salvo para algunos casos concretos, se puede decir que el número óptimo de procesadores que se deben utilizar para resolver un sistema tridiagonal utilizando estos algoritmos es de 64, lo que no debe sorprendernos si observamos

los valores de los parámetros para esta máquina que se muestran en la tabla 2.1. De esta tabla se desprende que para $p = 64$ las comunicaciones y las barreras de sincronización tienen un coste menor (en flops) que cuando $p = 32$. Como conclusión podemos afirmar que con $p = 64$ se consiguen los mejores tiempos en todos los algoritmos siempre que $n > 16384$.

4.6.3 Tiempos en un Cluster de Pentiums

La tabla 4.6 nos muestra los resultados teóricos esperados en un cluster de Pentiums, para 2, 4 y 8 procesadores.

A la vista de los resultados que presentan la tabla 4.6, podemos extraer unas conclusiones muy similares a las que se dedujeron para el IBM SP2. Las diferencias en los tiempos de ejecución de los cuatro algoritmos son muy similares (diferencias del orden de 10^{-4}), especialmente para 2 procesadores, con resultados prácticamente idénticos. Las escasas diferencias que se producen para 4 y 8 procesadores siempre son en favor del algoritmo 4.2, que resulta el más rápido de los cuatro. Por otra parte, coincidiendo también con el comportamiento observado en el IBM SP2, los tiempos aumentan al duplicar el número de procesadores.

4.6.4 Estudio del speedup

En esta sección se realiza un estudio del speedup para los algoritmos 4.2, 4.3, 4.4 y 4.5 implementados a lo largo de este capítulo. Dado que los tiempos teóricos obtenidos para estos algoritmos son muy similares en las tres máquinas, los valores del speedup y de la eficiencia van a ser prácticamente idénticos. Por esta razón en las tablas sólamente se reflejan los valores para el algoritmo 4.2, puesto que para el resto de algoritmos se repiten los valores en la mayoría de los casos.

La tabla 4.7 nos muestra el speedup y la eficiencia del algoritmo 4.2 en las máquinas IBM SP2, CRAY T3D y un cluster de Pentiums, para una matriz de gran tamaño, $n = 2097152$. La eficiencia la medimos en porcentaje. Dicha tabla corrobora los aspectos más destacables que ya se han comentado para estos algoritmos; su comportamiento paralelo no es bueno, salvo para el caso del CRAY T3D,

Cluster de Pentiums												
	p											
	2				4				8			
n	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5	Alg. 4.2	Alg. 4.3	Alg. 4.4	Alg. 4.5
512	0.0005	0.0005	0.0005	0.0005	0.0008	0.0011	0.0011	0.0012	0.0013	0.0019	0.0019	0.0023
1024	0.0008	0.0008	0.0008	0.0008	0.0012	0.0014	0.0014	0.0016	0.0017	0.0023	0.0023	0.0027
2048	0.0013	0.0014	0.0014	0.0014	0.0019	0.0022	0.0022	0.0023	0.0025	0.0031	0.0031	0.0035
4096	0.0025	0.0025	0.0025	0.0025	0.0034	0.0036	0.0036	0.0038	0.0041	0.0047	0.0047	0.0051
8192	0.0047	0.0048	0.0048	0.0048	0.0063	0.0066	0.0066	0.0067	0.0073	0.0079	0.0079	0.0083
16384	0.0093	0.0094	0.0094	0.0094	0.0121	0.0125	0.0125	0.0126	0.0137	0.0143	0.0143	0.0147
32768	0.0184	0.0185	0.0185	0.0185	0.0240	0.0243	0.0243	0.0244	0.0266	0.0272	0.0271	0.0276
65536	0.0366	0.0367	0.0367	0.0367	0.0476	0.0479	0.0479	0.0480	0.0522	0.0528	0.0528	0.0532
131072	0.0730	0.0731	0.0731	0.0731	0.0948	0.0951	0.0951	0.0952	0.1036	0.1042	0.1042	0.1046
262144	0.1458	0.1459	0.1459	0.1459	0.1892	0.1895	0.1895	0.1896	0.2063	0.2069	0.2069	0.2073
524288	0.2914	0.2915	0.2915	0.2915	0.3780	0.3783	0.3783	0.3784	0.4118	0.4124	0.4124	0.4128
1048576	0.5826	0.5826	0.5826	0.5826	0.7556	0.7558	0.7558	0.7560	0.8227	0.8233	0.8233	0.8237
2097152	1.1650	1.1650	1.1650	1.1650	1.5107	1.5110	1.5110	1.5111	1.6445	1.6451	1.6451	1.6455
4194304	2.3297	2.3298	2.3298	2.3298	3.0210	3.0213	3.0213	3.0214	3.2881	3.2887	3.2887	3.2891

Tabla 4.6: *Tiempos teóricos para los algoritmos 4.2, 4.3, 4.4 y 4.5 en un cluster de Pentiums.*

una máquina paralela con procesos de comunicación muy rápidos. En esta máquina se alcanzan valores máximos del speedup de 5.10, aunque con una eficiencia del 10%. El valor máximo de la eficiencia en esta máquina se alcanza para dos procesadores, siendo del 71%. En las otras máquinas los valores del speedup y de la eficiencia son muy discretos, salvo para el IBM SP2 con switch, donde los valores son mucho mejores que en las otras máquinas pero quedan muy por debajo de los que se alcanzan en el CRAY T3D (alrededor del 40% como valor máximo de la eficiencia en el IBM con switch).

Las figuras 4.5 y 4.6 nos muestran los valores del *speedup* obtenidos en la tabla 4.7. Las figuras 4.7 y 4.8 nos muestran los valores de la eficiencia para los algoritmos 3.4 y 3.5 en las tres máquinas donde se presentan resultados.

IBM SP2 switch		
	speedup	eficiencia
p	Alg. 4.2	Alg. 4.2
2	0.8229	41.47
4	0.8480	21.20
8	0.8070	10.09

(a) *IBM SP2 switch*

IBM SP2 ethernet		
	speedup	eficiencia
p	Alg. 4.2	Alg. 4.2
2	0.06	2.96
4	0.02	0.48
8	0.01	0.06

(b) *IBM SP2 ethernet*

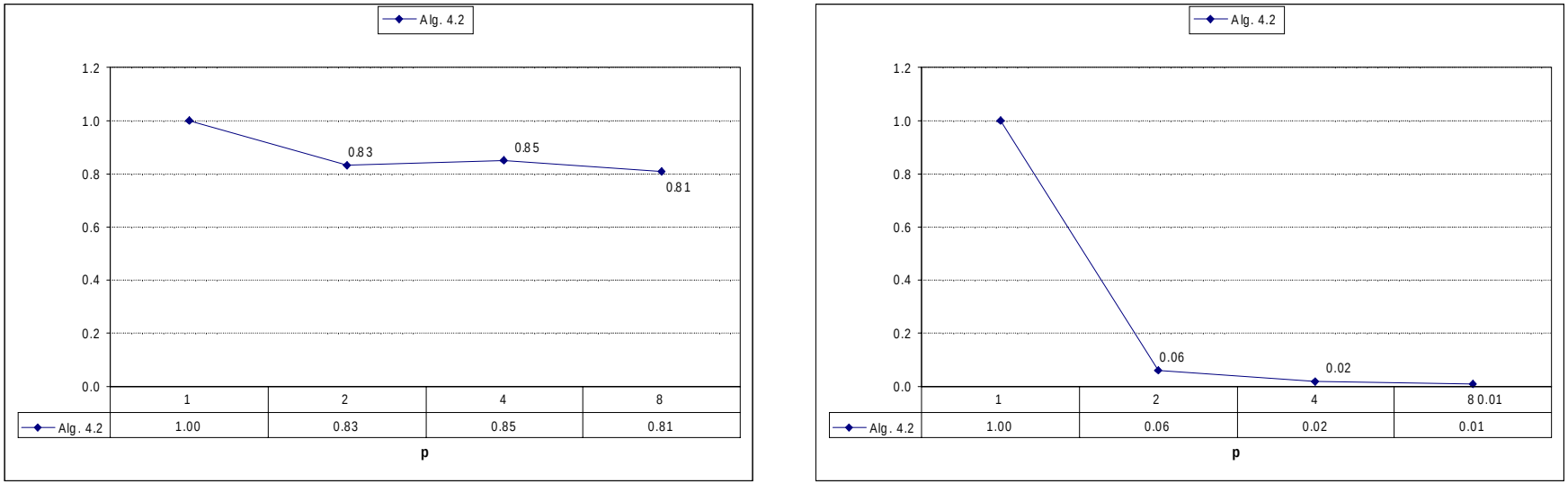
CRAY T3D		
	speedup	eficiencia
p	Alg. 4.2	Alg. 4.2
2	1.42	71.13
4	2.40	59.97
8	3.50	43.69
16	4.29	26.83
32	4.34	13.56
64	5.10	9.44
128	4.89	3.82
256	4.45	1.74

(c) *CRAY T3D*

Cluster de Pentiums		
	speedup	eficiencia
p	Alg. 4.2	Alg. 4.2
2	0.20	10.15
4	0.22	5.56
8	0.20	2.55

(d) *Cluster de Pentiums*

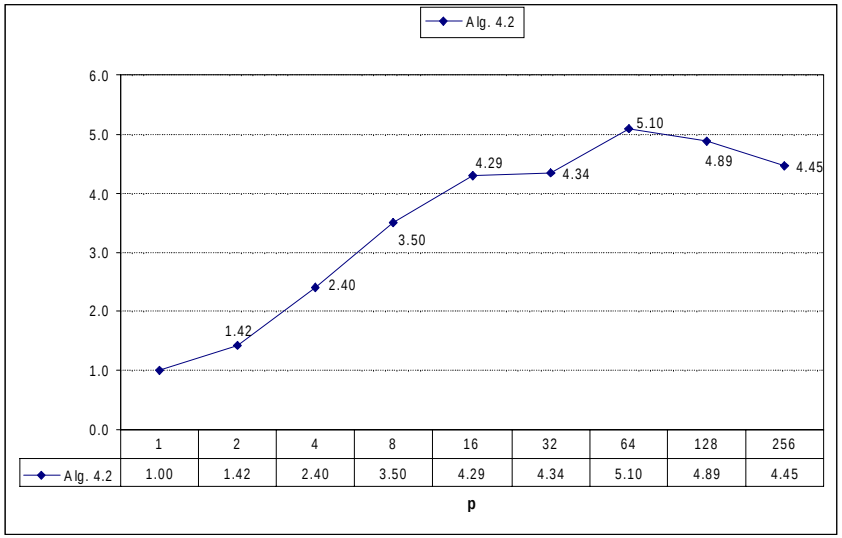
Tabla 4.7: *Speedup y eficiencia del algoritmo Alg. 4.2 para una matriz de tamaño* $n = 2097152$.



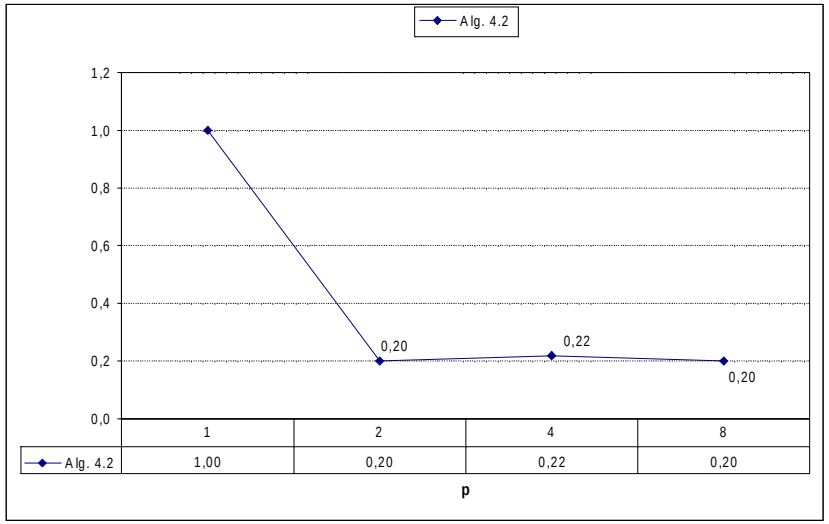
(a) *IBM SP2 switch*

(b) *IBM SP2 ethernet*

Figura 4.5: *Valores del speedup en un IBM SP2*

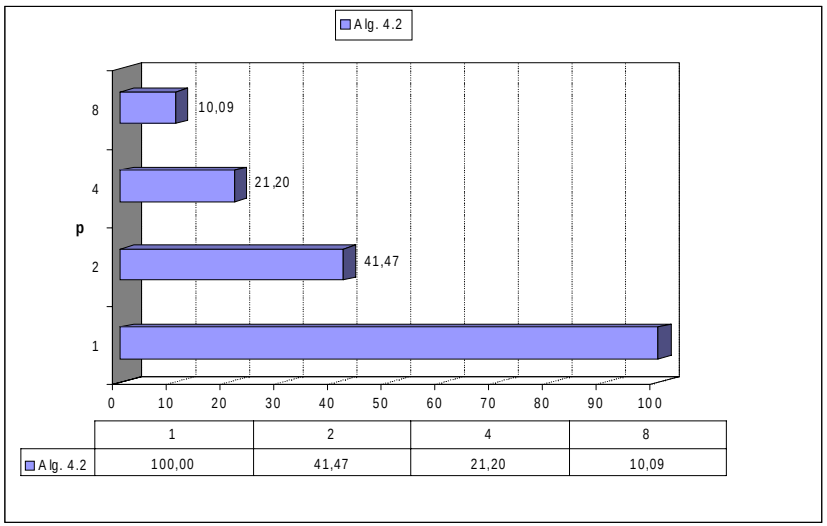


(a) *CRAY T3D*

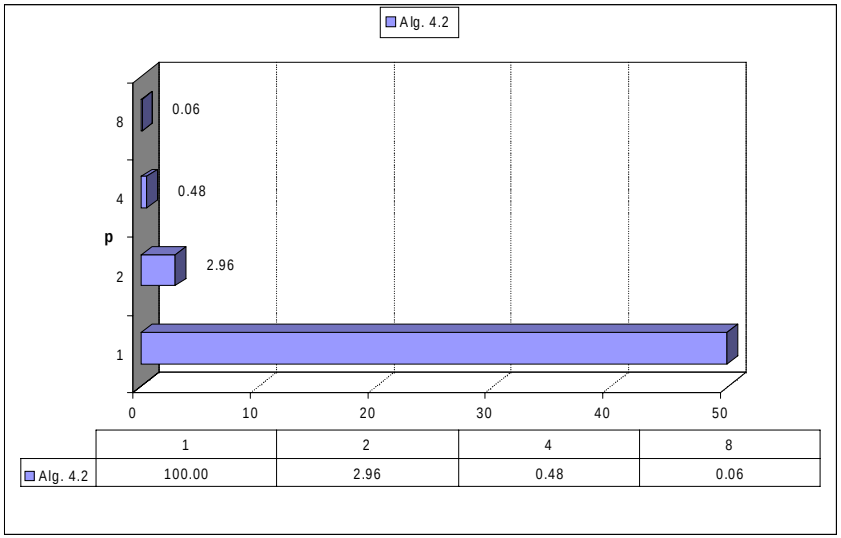


(b) *Cluster de Pentiums*

Figura 4.6: *Valores del speedup en un CRAY T3D y un cluster de Pentiums*

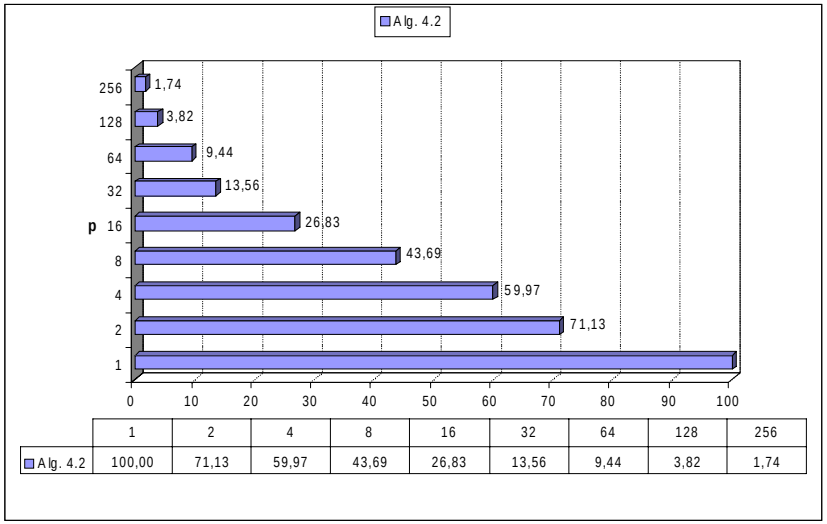


(a) *IBM SP2 switch*



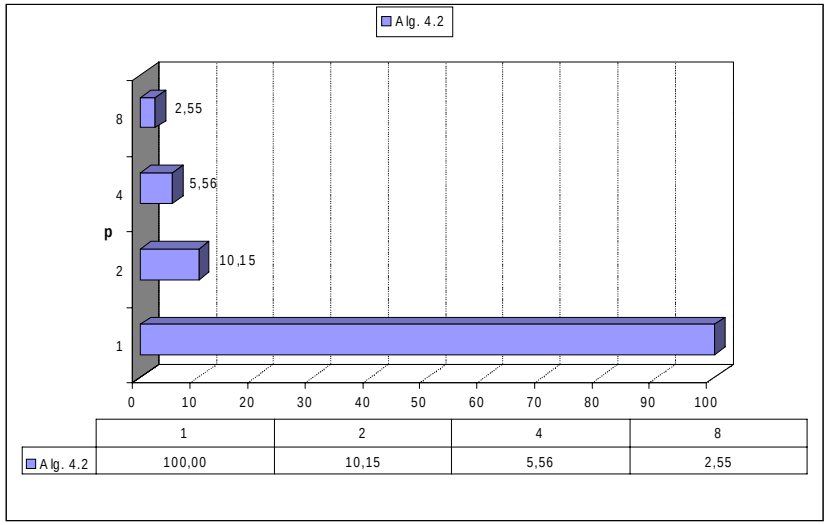
(b) *IBM SP2 ethernet*

Figura 4.7: *Valores de la eficiencia en un IBM SP2*



	1	2	4	8	16	32	64	128	256
Alg. 4.2	100,00	71,13	59,97	43,69	26,83	13,56	9,44	3,82	1,74

(a) *CRAY T3D*



	1	2	4	8
Alg. 4.2	100,00	10,15	5,56	2,55

(b) *Cluster de Pentiums*

Figura 4.8: *Valores de la eficiencia en un CRAY T3D y un cluster de Pentiums*

Capítulo 5 Método divide y vencerás de Bondeli

5.1 Introducción

La idea en que se basa el método de Bondeli constituye una técnica del tipo *divide y vencerás* puesto que se particiona en bloques el sistema inicial y el vector de términos independiente para resolver en cada procesador un conjunto de subsistemas tridiagonales. Posteriormente, a partir de la definición de unas nuevas variables, se construye un sistema tridiagonal auxiliar que nos permite obtener la solución general.

Las diferencias básicas entre los distintos algoritmos se encuentran en la forma en que se construye y resuelve el sistema tridiagonal auxiliar y el procesador donde se obtienen las soluciones finales. En el último de los algoritmos estudiados se utiliza el método paralelo *recursive doubling* para la resolución del sistema tridiagonal auxiliar.

Se considera el problema general de obtener la solución del sistema

$$A\boldsymbol{x} = \boldsymbol{d}, \tag{5.1}$$

donde

$$A = \begin{bmatrix} a_1 & b_1 & & & \\ c_2 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & c_n & a_n \end{bmatrix} \quad \text{y} \quad \boldsymbol{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \tag{5.2}$$

son, respectivamente, una matriz tridiagonal e irreducible y el vector de términos independiente. Supongamos que podemos encontrar dos números naturales k y p tales que $k = \frac{n}{p}$ y consideremos la siguiente partición por bloques, como en el capítulo 4,

$$A = \begin{bmatrix} A_0 & B_0 & & & \\ C_1 & A_1 & B_1 & & \\ & \ddots & \ddots & \ddots & \\ & & C_{p-2} & A_{p-2} & B_{p-2} \\ & & & C_{p-1} & A_{p-1} \end{bmatrix}, \tag{5.3}$$

de la matriz A donde cada uno de los bloques diagonales

$$A_i = \begin{bmatrix} a_{ik+1} & b_{ik+1} & & & \\ c_{ik+2} & a_{ik+2} & b_{ik+2} & & \\ & \ddots & \ddots & \ddots & \\ & & c_{(i+1)k-1} & a_{(i+1)k-1} & b_{(i+1)k-1} \\ & & & c_{(i+1)k} & a_{(i+1)k} \end{bmatrix}, \qquad i = 0, 1, \ldots, p-1,$$

es una matriz tridiagonal de tamaño $k \times k$ y, para $i = 0, 1, \ldots, p-2$, cada bloque subdiagonal

$$C_{i+1} = \left[\begin{array}{cccc|c} 0 & 0 & \cdots & 0 & c_{(i+1)k+1} \\ \hline & & & & 0 \\ & & O & & \vdots \\ & & & & 0 \\ & & & & 0 \end{array}\right] = c_{(i+1)k+1}\boldsymbol{e}_1\boldsymbol{e}_k^T \tag{5.4}$$

y superdiagonal

$$B_i = \left[\begin{array}{c|cccc} 0 & & & & \\ 0 & & O & & \\ \vdots & & & & \\ 0 & & & & \\ \hline b_{(i+1)k} & 0 & \cdots & 0 & 0 \end{array}\right] = b_{(i+1)k}\boldsymbol{e}_k\boldsymbol{e}_1^T \tag{5.5}$$

es de tamaño $k \times k$ con un único elemento no nulo. Los vectores $\boldsymbol{e}_1$ y $\boldsymbol{e}_k$, siguiendo la notación habitual, representan la primera y la última columna, respectivamente, de la matriz identidad I_k.

En los vectores $\boldsymbol{x}$ y $\boldsymbol{d}$ se considera una partición por bloques conforme con la partición en bloques de la matriz A dada por la expresión (5.3), es decir,

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_{p-2} \\ \boldsymbol{x}_{p-1} \end{bmatrix} \quad \text{y} \quad \boldsymbol{d} = \begin{bmatrix} \boldsymbol{d}_0 \\ \boldsymbol{d}_1 \\ \vdots \\ \boldsymbol{d}_{p-2} \\ \boldsymbol{d}_{p-1} \end{bmatrix},$$

con

$$\boldsymbol{x}_i = \begin{bmatrix} x_{ik+1} \\ x_{ik+2} \\ \vdots \\ x_{(i+1)k-1} \\ x_{(i+1)k} \end{bmatrix} \quad \text{y} \quad \boldsymbol{d}_i = \begin{bmatrix} d_{ik+1} \\ d_{ik+2} \\ \vdots \\ d_{(i+1)k-1} \\ d_{(i+1)k} \end{bmatrix}, \quad i = 0, 1, \ldots, p-1.$$

5.2 Descripción del método

La notación introducida en la sección 5.1 nos permite escribir el sistema (5.1) como

$$\begin{bmatrix} A_0 & B_0 & & & \\ C_1 & A_1 & B_1 & & \\ & \ddots & \ddots & \ddots & \\ & & C_{p-2} & A_{p-2} & B_{p-2} \\ & & & C_{p-1} & A_{p-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_{p-2} \\ \boldsymbol{x}_{p-1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{d}_0 \\ \boldsymbol{d}_1 \\ \vdots \\ \boldsymbol{d}_{p-2} \\ \boldsymbol{d}_{p-1} \end{bmatrix}, \tag{5.6}$$

o en forma abreviada,

$$C_i \boldsymbol{x}_{i-1} + A_i \boldsymbol{x}_i + B_i \boldsymbol{x}_{i+1} = \boldsymbol{d}_i, \qquad i = 0, 1, \ldots, p-1, \tag{5.7}$$

teniendo en cuenta que $C_0 = B_{p-1} = O$.

Como los bloques diagonales A_i, para $i = 0, 1, \ldots, p-1$, son no singulares, sustituyendo las expresiones (5.4) y (5.5) en la expresión (5.7) y despejando el vector $\boldsymbol{x}_i$ obtenemos el sistema

$$\boldsymbol{x}_i = A_i^{-1}\boldsymbol{d}_i - c_{ik+1}A_i^{-1}\boldsymbol{e}_1\boldsymbol{e}_k^T\boldsymbol{x}_{i-1} - b_{(i+1)k}A_i^{-1}\boldsymbol{e}_k\boldsymbol{e}_1^T\boldsymbol{x}_{i+1}, \qquad i = 0, 1, \ldots, p-1. \tag{5.8}$$

Si definimos, para $i = 0, 1, \ldots, p-1$,

$$\boldsymbol{y}_i = A_i^{-1}\boldsymbol{d}_i, \tag{5.9}$$

$$\boldsymbol{z}_{2i-1} = A_i^{-1}\boldsymbol{e}_1, \quad \boldsymbol{z}_{2i} = A_i^{-1}\boldsymbol{e}_k, \tag{5.10}$$

$$\alpha_{2i-1} = -c_{ik+1}\boldsymbol{e}_k^T\boldsymbol{x}_{i-1}, \qquad \alpha_{2i} = -b_{(i+1)k}\boldsymbol{e}_1^T\boldsymbol{x}_{i+1}, \tag{5.11}$$

entonces podemos escribir la ecuación (5.8) como

$$\boldsymbol{x}_i = \boldsymbol{y}_i + \alpha_{2i-1}\boldsymbol{z}_{2i-1} + \alpha_{2i}\boldsymbol{z}_{2i}, \qquad i = 0, 1, \ldots, p-1. \tag{5.12}$$

Hemos formado, a partir del conjunto de ecuaciones (5.8), mediante las expresiones (5.9), (5.10) y (5.11), un nuevo sistema dado por la expresión (5.12) en el que ahora las incógnitas son los escalares α_0, α_1, $\alpha_2, \ldots,$ α_{2p-4}, α_{2p-3}. Estudiamos detalladamente la relación existente entre estos escalares y los vectores $\boldsymbol{z}_i$, para $i = 0, 1, \ldots, 2p-4, 2p-3$, por medio de las expresiones (5.8) y (5.12).

De

$$\alpha_0 = -b_k(\boldsymbol{e}_1^T\boldsymbol{x}_1) = -b_k\left[\boldsymbol{e}_1^T(\boldsymbol{y}_1 + \alpha_1\boldsymbol{z}_1 + \alpha_2\boldsymbol{z}_2)\right] = -b_k\left[\boldsymbol{e}_1^T\boldsymbol{y}_1 + (\boldsymbol{e}_1^T\boldsymbol{z}_1)\alpha_1 + (\boldsymbol{e}_1^T\boldsymbol{z}_2)\alpha_2\right],$$

tenemos que

$$\frac{1}{b_k}\alpha_0 + (\boldsymbol{e}_1^T\boldsymbol{z}_1)\alpha_1 + (\boldsymbol{e}_1^T\boldsymbol{z}_2)\alpha_2 = -\boldsymbol{e}_1^T\boldsymbol{y}_1,$$

mientras que

$$\alpha_1 = -c_{k+1}(\boldsymbol{e}_k^T \boldsymbol{x}_0) = -c_{k+1}\left[\boldsymbol{e}_k^T(\boldsymbol{y}_0 + \alpha_0 \boldsymbol{z}_0)\right] = -c_{k+1}\left[\boldsymbol{e}_k^T \boldsymbol{y}_0 + (\boldsymbol{e}_k^T \boldsymbol{z}_0)\alpha_0\right],$$

por lo que

$$(\boldsymbol{e}_k^T \boldsymbol{z}_0)\alpha_0 + \frac{1}{c_{k+1}}\alpha_1 = -\boldsymbol{e}_k^T \boldsymbol{y}_0.$$

Siguiendo un razonamiento análogo para $i = 2, 3, \ldots, 2p-4, 2p-3$, obtendremos el conjunto de ecuaciones

$$(\boldsymbol{e}_k^T \boldsymbol{z}_{2i-1})\alpha_{2i-1} + (\boldsymbol{e}_k^T \boldsymbol{z}_{2i})\alpha_{2i} + \frac{1}{c_{(i+1)k+1}}\alpha_{2i+1} = -(\boldsymbol{e}_k^T \boldsymbol{y}_i), \qquad i = 0, 1, \ldots, p-2, \tag{5.13}$$

$$\frac{1}{b_{ik}}\alpha_{2i-2} + (\boldsymbol{e}_1^T \boldsymbol{z}_{2i-1})\alpha_{2i-1} + (\boldsymbol{e}_1^T \boldsymbol{z}_{2i})\alpha_{2i} = -(\boldsymbol{e}_1^T \boldsymbol{y}_i), \qquad i = 1, 2, \ldots, p-1. \tag{5.14}$$

Las ecuaciones (5.13) y (5.14) constituyen un sistema tridiagonal de tamaño $(2p-2) \times (2p-2)$, que escribimos como

$$H\boldsymbol{\alpha} = \boldsymbol{\beta} \tag{5.15}$$

con incógnitas α_i, para $i = 0, 1, \ldots, 2p-3$, matriz de coeficientes

$$H = \begin{bmatrix} \delta_0 & \mu_0 & & & \\ \nu_1 & \delta_1 & \mu_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \nu_{2p-4} & \delta_{2p-4} & \mu_{2p-4} \\ & & & \nu_{2p-3} & \delta_{2p-3} \end{bmatrix} \tag{5.16}$$

con elementos diagonales, superdiagonales y subdiagonales, respectivamente

$$\delta_i = \begin{cases} \boldsymbol{e}_k^T \boldsymbol{z}_i & i = 0, 2, \ldots, 2p-4, \\ \boldsymbol{e}_1^T \boldsymbol{z}_i & i = 1, 3, \ldots, 2p-3, \end{cases}, \qquad \mu_i = \begin{cases} \frac{1}{c_{\left(\frac{i}{2}+1\right)k+1}} & i = 0, 2, \ldots, 2p-4, \\ \boldsymbol{e}_1^T \boldsymbol{z}_{i+1} & i = 1, 3, \ldots, 2p-5, \end{cases}, \qquad \nu_i = \begin{cases} \boldsymbol{e}_k^T \boldsymbol{z}_{i-1} & i = 2, 4, \ldots, 2p-4, \\ \frac{1}{b_{\frac{i+1}{2}k}} & i = 1, 3, \ldots, 2p-3, \end{cases} \tag{5.17}$$

y vector de términos independientes $\boldsymbol{\beta} = [\ \beta_0 \quad \beta_1 \quad \ldots \quad \beta_{2p-3} \quad \beta_{2p-3}\ ]^T$ con

$$\beta_i = \begin{cases} -\boldsymbol{e}_k^T \boldsymbol{y}_{\frac{i}{2}} & i = 0, 2, \ldots, 2p-4, \\ -\boldsymbol{e}_1^T \boldsymbol{y}_{\frac{i+1}{2}} & i = 1, 3, \ldots, 2p-3. \end{cases} \tag{5.18}$$

Una vez obtenida la solución $\boldsymbol{\alpha}$ del sistema (5.15), podemos obtener la solución $\boldsymbol{x}$ del sistema (4.1) sustituyendo las componentes de $\boldsymbol{\alpha}$ en la expresión (5.12).

Un algoritmo paralelo que describa las características de este método debería constar de las siguientes fases claramente diferenciadas:

- En una primera fase debemos distribuir la matriz A y los vectores $\boldsymbol{x}$ y $\boldsymbol{d}$ de acuerdo con la expresión (5.6).
- En la segunda fase cada procesador P_i, para $i = 0, 1, \ldots, p-1$, debe calcular en paralelo los vectores $\boldsymbol{y}_i$, $\boldsymbol{z}_{2i-1}$ y $\boldsymbol{z}_{2i}$, solución de los sistemas (5.9) y (5.10).
- En la tercera fase del algoritmo se construye y resuelve el sistema (5.15) mediante las expresiones (5.16), (5.17) y (5.18), cuya solución es $\boldsymbol{\alpha}$.
- En la última fase del algoritmo se sustituye $\boldsymbol{\alpha}$ en la expresión (5.12) para obtener la solución del sistema inicial.

Si se analizan detalladamente las fases anteriores del método, se observa que la fase donde se realizan la mayor parte de los cálculos aritméticos es la segunda, en la que se deben calcular las variables $\boldsymbol{y}_i$, para $i = 0, 1, \ldots, p-1$, y $\boldsymbol{z}_j$, para $j = 0, 1, \ldots, 2p-4, 2p-3$. Estas variables pueden ser calculadas en paralelo en los diferentes procesadores mediante las expresiones (5.9) y (5.10). Nótese que en estos sistemas la matriz de coeficientes no cambia, únicamente cambia el vector de términos independientes. Podemos resumir las tareas aritméticas que debe llevar a cabo cada procesador de la siguiente forma:

- En el procesador P_0 se resuelven los sistemas

$$A_0[\ \boldsymbol{y}_0 \quad \boldsymbol{z}_0\ ] = [\ \boldsymbol{d}_0 \quad \boldsymbol{e}_k\ ]. \tag{5.19}$$

- En el procesador P_i, para $i = 1, 2, \ldots, p-2$, se resuelven los sistemas

$$A_i[\ \boldsymbol{y}_i \quad \boldsymbol{z}_{2i-1} \quad \boldsymbol{z}_{2i}\ ] = [\ \boldsymbol{d}_i \quad \boldsymbol{e}_1 \quad \boldsymbol{e}_k\ ]. \tag{5.20}$$

- En el procesador P_{p-1} se resuelven los sistemas

$$A_{p-1}[\ \boldsymbol{y}_{p-1} \quad \boldsymbol{z}_{2p-3}\ ] = [\ \boldsymbol{d}_{p-1} \quad \boldsymbol{e}_1\ ]. \tag{5.21}$$

Observamos que el método en esta fase no está totalmente balanceado en cuanto a carga de trabajo computacional, ya que mientras que el procesador primero y último resuelven dos sistemas, los procesadores centrales resuelven tres sistemas, de la misma forma que ocurría en el capítulo 4 con el método basado en la fórmula de Sherman-Morrison-Woodbury.

Consideramos el siguiente ejemplo que resume los cálculos expuestos a lo largo de esta sección para un sistema con $n = 16$.

Ejemplo 5.1 *Consideramos la resolución del sistema (5.1), siendo*

$$A = \begin{bmatrix} 12 & 1 & & & & & & & & & & & & & & \\ 3 & 12 & 1 & & & & & & & & & & & & & \\ & 3 & 12 & 1 & & & & & & & & & & & & \\ & & 3 & 12 & 1 & & & & & & & & & & & \\ & & & 3 & 12 & 1 & & & & & & & & & & \\ & & & & 3 & 12 & 1 & & & & & & & & & \\ & & & & & 3 & 12 & 1 & & & & & & & & \\ & & & & & & 3 & 12 & 1 & & & & & & & \\ & & & & & & & 3 & 12 & 1 & & & & & & \\ & & & & & & & & 3 & 12 & 1 & & & & & \\ & & & & & & & & & 3 & 12 & 1 & & & & \\ & & & & & & & & & & 3 & 12 & 1 & & & \\ & & & & & & & & & & & 3 & 12 & 1 & & \\ & & & & & & & & & & & & 3 & 12 & 1 & \\ & & & & & & & & & & & & & 3 & 12 & 1 \\ & & & & & & & & & & & & & & 3 & 12 \end{bmatrix} \quad y \quad \boldsymbol{d} = \begin{bmatrix} 13 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 16 \\ 15 \end{bmatrix}.$$

Por la forma en que se han tomado la matriz A y el vector $\boldsymbol{d}$ resulta inmediato que la solución $\boldsymbol{x}$ del sistema $A\boldsymbol{x} = \boldsymbol{d}$ tiene todas sus componentes iguales a 1.

Supongamos que tomamos $p = 4$, por lo que $k = 4$. Entonces, el procesador P_0 tendrá los bloques

$$A_0 = \begin{bmatrix} 12 & 1 & & \\ 3 & 12 & 1 & \\ & 3 & 12 & 1 \\ & & 3 & 12 \end{bmatrix} \qquad y \qquad \boldsymbol{d} = \begin{bmatrix} 13 \\ 16 \\ 16 \\ 16 \end{bmatrix}.$$

Los procesadores no extremos P_i, para $i = 1, 2$ tendrán los bloques

$$A_i = \begin{bmatrix} 12 & 1 & & \\ 3 & 12 & 1 & \\ & 3 & 12 & 1 \\ & & 3 & 12 \end{bmatrix} \qquad y \qquad \boldsymbol{d} = \begin{bmatrix} 16 \\ 16 \\ 16 \\ 16 \end{bmatrix},$$

mientras que el procesador P_3 tendrá los bloques

$$A_3 = \begin{bmatrix} 12 & 1 & & \\ 3 & 12 & 1 & \\ & 3 & 12 & 1 \\ & & 3 & 12 \end{bmatrix} \qquad y \qquad \boldsymbol{d}_3 = \begin{bmatrix} 16 \\ 16 \\ 16 \\ 15 \end{bmatrix}.$$

Con estos bloques, el procesador P_0 resuelve los sistemas

$$A_0 \boldsymbol{y}_0 = \boldsymbol{d}_0, \qquad A_0 \boldsymbol{z}_0 = \boldsymbol{e}_k,$$

cuyas soluciones son

$$\boldsymbol{y}_0 = \begin{bmatrix} 1.000 \\ 1.001 \\ 0.993 \\ 1.085 \end{bmatrix}, \qquad \boldsymbol{z}_0 = \begin{bmatrix} -0.00005 \\ 0.00060 \\ -0.00720 \\ 0.08510 \end{bmatrix}.$$

El procesador P_1 resuelve los sistemas

$$A_1\boldsymbol{y}_1 = \boldsymbol{d}_1, \quad A_1\boldsymbol{z}_1 = \boldsymbol{e}_1, \quad A_1\boldsymbol{z}_2 = \boldsymbol{e}_k,$$

cuyas soluciones son

$$\boldsymbol{y}_1 = \begin{bmatrix} 1.2550 \\ 0.9354 \\ 1.0090 \\ 1.0810 \end{bmatrix}, \qquad \boldsymbol{z}_1 = \begin{bmatrix} 0.0850 \\ -0.2170 \\ 0.0056 \\ -0.0014 \end{bmatrix}, \qquad \boldsymbol{z}_2 = \begin{bmatrix} -0.00005 \\ 0.00060 \\ -0.00700 \\ 0.08510 \end{bmatrix}.$$

El procesador P_2 resuelve los sistemas

$$A_2\boldsymbol{y}_2 = \boldsymbol{d}_2, \quad A_2\boldsymbol{z}_3 = \boldsymbol{e}_1, \quad A_2\boldsymbol{z}_4 = \boldsymbol{e}_k.$$

Nótese que, debido a la forma de la matriz de coeficientes de este ejemplo, los bloques de los procesadores P_1 y P_2 son iguales, por lo

que las soluciones de los sistemas intermedios que resuelven son las mismas; así,

$$\boldsymbol{y}_2 = \begin{bmatrix} 1.2550 \\ 0.9354 \\ 1.0090 \\ 1.0810 \end{bmatrix}, \qquad \boldsymbol{z}_3 = \begin{bmatrix} 0.0850 \\ -0.2170 \\ 0.0056 \\ -0.0014 \end{bmatrix}, \qquad \boldsymbol{z}_4 = \begin{bmatrix} -0.00005 \\ 0.00060 \\ -0.00700 \\ 0.08510 \end{bmatrix}.$$

El procesador P_3 resuelve los sistemas

$$A_3\boldsymbol{y}_3 = \boldsymbol{d}_3, \qquad A_3\boldsymbol{z}_5 = \boldsymbol{e}_1,$$

cuyas soluciones son

$$\boldsymbol{y}_3 = \begin{bmatrix} 1.2550 \\ 0.9348 \\ 1.0170 \\ 0.9958 \end{bmatrix}, \qquad \boldsymbol{z}_5 = \begin{bmatrix} 0.0850 \\ -0.2170 \\ 0.0056 \\ -0.0014 \end{bmatrix}.$$

Ahora construimos el sistema tridiagonal (5.15) para este caso concreto, que viene dado por

$$\begin{bmatrix} 0.085 & 0.3333 & & & & \\ 1.000 & 0.0850 & -0.00005 & & & \\ & -0.0014 & 0.08510 & 0.3333 & & \\ & & 1.00000 & 0.0850 & -0.00005 & \\ & & & -0.0014 & 0.08510 & 0.333 \\ & & & & 1.00000 & 0.085 \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \alpha_5 \end{bmatrix} = \begin{bmatrix} -1.085 \\ 1.255 \\ 1.081 \\ 1.255 \\ 1.081 \\ 1.255 \end{bmatrix},$$

cuya solución es

$$\begin{bmatrix} -1.000 \\ -3.000 \\ -1.000 \\ -3.001 \\ -1.000 \\ -3.001 \end{bmatrix}.$$

Una vez calculada la solución $\boldsymbol{\alpha}$*, cada procesador utiliza ciertas componentes de esta solución para calcular las soluciones parciales del sistema inicial. Así, el procesador* P_0 *calcula las cuatro primeras componentes de la solución final de la forma*

$$\boldsymbol{x}_0 = \boldsymbol{y}_0 + \alpha_0 \boldsymbol{z}_0 = \begin{bmatrix} 1.000 \\ 1.001 \\ 0.993 \\ 1.085 \end{bmatrix} - \begin{bmatrix} -0.00005 \\ 0.00060 \\ -0.00720 \\ 0.08510 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

De una forma análoga, el procesador P_1 *calcula las cuatro siguientes componentes de la solución final mediante la expresión*

$$\boldsymbol{x}_1 = \boldsymbol{y}_1 + \alpha_1 \boldsymbol{z}_1 + \alpha_2 \boldsymbol{z}_2 = \begin{bmatrix} 1.2550 \\ 0.9354 \\ 1.0090 \\ 1.0810 \end{bmatrix} - 3 \begin{bmatrix} 0.0850 \\ -0.2170 \\ 0.0056 \\ -0.0014 \end{bmatrix} - \begin{bmatrix} -0.00005 \\ 0.00060 \\ -0.00700 \\ 0.08510 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

El procesador P_2 calcula

$$\boldsymbol{x}_2 = \boldsymbol{y}_2 + \alpha_3 \boldsymbol{z}_3 + \alpha_4 \boldsymbol{z}_4 = \begin{bmatrix} 1.2550 \\ 0.9354 \\ 1.0090 \\ 1.0810 \end{bmatrix} - 3.001 \begin{bmatrix} 0.0850 \\ -0.2170 \\ 0.0056 \\ -0.0014 \end{bmatrix} - 1 \begin{bmatrix} -0.00005 \\ 0.00060 \\ -0.00700 \\ 0.08510 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Finalmente, el procesador P_3 calcula las últimas cuatro componentes de la solución de una forma similar a los casos anteriores

$$\boldsymbol{x}_3 = \boldsymbol{y}_3 + \alpha_5 \boldsymbol{z}_5 = \begin{bmatrix} 1.2550 \\ 0.9348 \\ 1.0170 \\ 0.9958 \end{bmatrix} - 3.001 \begin{bmatrix} 0.0850 \\ -0.2170 \\ 0.0056 \\ -0.0014 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

De esta forma queda calculada la solución del sistema propuesto inicialmente en este ejemplo que, como cabía esperar, tiene todas sus componentes iguales a uno.

5.3 Algoritmos BSP divide y vencerás

5.3.1 Algoritmo BSP basado en el método de Bondeli

De acuerdo con las características de este método expuestas en la sección 5.2 y observando detalladamente el ejemplo 5.1, llegamos a la conclusión de que únicamente son necesarios tres superpasos para construir un algoritmo BSP que resuelva un sistema tridiagonal por el método de Bondeli. Un superpaso será necesario para las comunicaciones iniciales; en un segundo superpaso se resolverán los sistemas

iniciales de forma independiente en cada procesador, mientras que en el tercer superpaso se construirá y resolverá el sistema tridiagonal auxiliar que nos permitirá obtener la solución general. El siguiente algoritmo BSP recoge las características fundamentales del método divide y vencerás desarrolladas en la sección 5.2.

Algoritmo 5.1 Método BSP divide y vencerás de Bondeli para sistemas tridiagonales.

Superpaso 1

Comunicación de datos a los procesadores. Para $i = 1, 2, \ldots, p-1$, el procesador P_i recibe la submatriz A_i y el vector $\boldsymbol{d}_i$ desde el procesador principal.

Superpaso 2

1 Resolución de los subsistemas intermedios y envío de los vectores auxiliares al procesador principal.

- 1.1 En el procesador P_0 se resuelven los sistemas (5.19).
- 1.2 En el procesador P_i, para $i = 1, 2, \ldots, p-2$, se resuelven los sistemas (5.20).
- 1.3 En el procesador P_{p-1} se resuelven los sistemas (5.21).

2 Comunicación. El procesador P_i, para $i = 1, 2, \ldots, p-1$, envía los vectores $\boldsymbol{y}_i, \boldsymbol{z}_{2i-1}, \boldsymbol{z}_{2i}$ al procesador P_0.

Superpaso 3

Obtención de la solución final. En el procesador P_0,

1 se calculan H y el vector $\boldsymbol{\beta}$ mediante las expresiones (5.16), (5.17) y (5.18),

2 se resuelve el sistema (5.15),

3 se calcula la solución $\boldsymbol{x}$ utilizando la expresión (5.12).

Ya se ha comentado en la sección 5.2 que en el segundo superpaso se realizan la mayoría de los cálculos aritméticos para calcular los vectores $\boldsymbol{y}$ y $\boldsymbol{z}$ mediante la resolución de los sistemas (5.19), (5.20) y (5.21).

Si observamos las expresiones (5.19), (5.20) y (5.21), notamos que debemos resolver sistemas con la misma matriz de coeficientes y distintos vectores de términos independientes. Como ya vimos en la sección 1.1, estas características nos llevan a la utilización de la factorización LU de la matriz de coeficientes como método óptimo para el cálculo de los vectores $\boldsymbol{y}_i$, $\boldsymbol{z}_{2i-1}$ y $\boldsymbol{z}_{2i}$, para $i = 0, 1, \ldots, p-1$, teniendo en cuenta que $\boldsymbol{z}_{2p-2} = 0$.

En consecuencia, podemos describir los cálculos que realiza cada uno de los procesadores en el superpaso 2 del algoritmo 5.1 mediante el siguiente esquema:

Procesador P_0:

- Cálculo de $A_0 = L_0 U_0$.

- Resolución de los sistemas $L_0 \boldsymbol{u}_0 = \boldsymbol{d}_0$ y $U_0[\ \boldsymbol{y}_0 \quad \boldsymbol{z}_0\ ] = [\ \boldsymbol{u}_0 \quad \boldsymbol{e}_k\ ]$.

Procesador P_i, para $i = 1, 2, \ldots, p-2$:

- Cálculo de $A_i = L_i U_i$.
- Resolución de los sistemas $L_i[\ \boldsymbol{u}_i \quad \boldsymbol{v}_i\ ] = [\ \boldsymbol{d}_i \quad \boldsymbol{e}_1\ ]$ y $U_i[\ \boldsymbol{y}_i \quad \boldsymbol{z}_{2i-1} \quad \boldsymbol{z}_{2i}\ ] = [\ \boldsymbol{u}_i \quad \boldsymbol{v}_i \quad \boldsymbol{e}_k\ ]$.

Procesador P_{p-1}:

- Cálculo de $A_{p-1} = L_{p-1} U_{p-1}$.
- Resolución de los sistemas $L_{p-1}[\ \boldsymbol{u}_{p-1} \quad \boldsymbol{v}_{p-1}\ ] = [\ \boldsymbol{d}_{p-1} \quad \boldsymbol{e}_1\ ]$ y $U_{p-1}[\ \boldsymbol{y}_{p-1} \quad \boldsymbol{z}_{2p-3}\ ] = [\ \boldsymbol{u}_{p-1} \quad \boldsymbol{v}_{p-1}\ ]$.

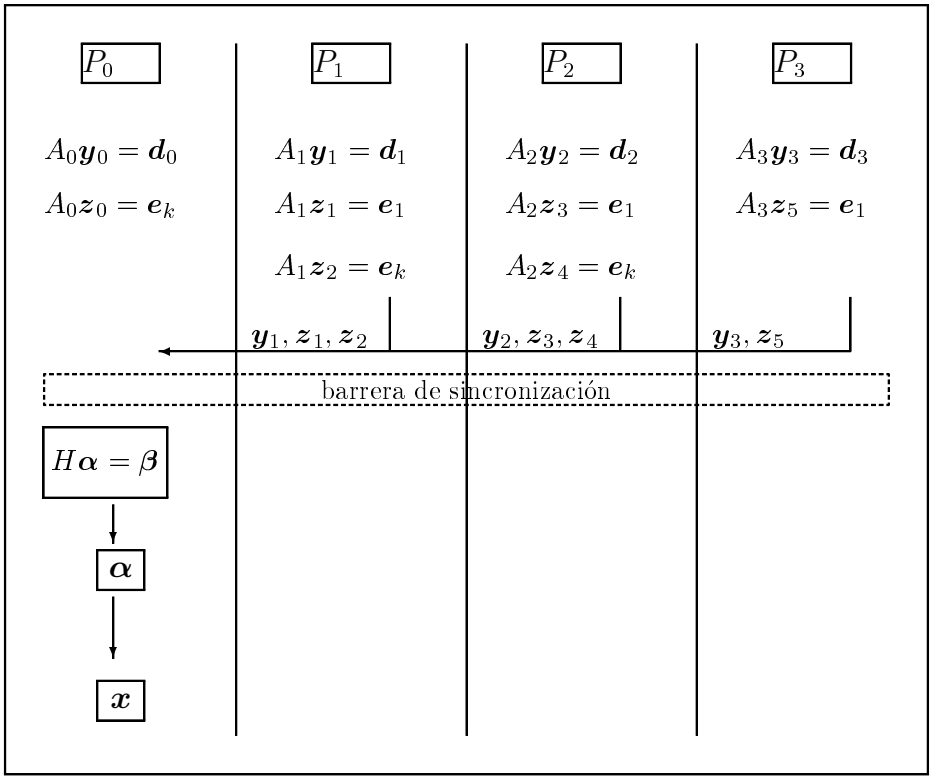


Figura 5.1: *Esquema de ejecución del algoritmo 5.1, para 4 procesadores.*

La figura 5.1 muestra un ejemplo de ejecución del algoritmo 5.1 para el caso particular en que $p = 4$.

Una característica fundamental que presenta el algoritmo 5.1 es que los cálculos que se realizan en el superpaso 3 los realiza únicamente el procesador principal; por eso, al final del superpaso 2 se produce una comunicación de elementos desde todos los procesadores al procesador principal, que es el encargado de formar el sistema (5.15) y resolverlo, de forma secuencial, para obtener la solución final.

5.3.2 Coste computacional

Seguidamente, calculamos los costes de cada uno de los superpasos que integran el algoritmo 5.1 con el fin de obtener su coste computacional.

Coste del superpaso 1. El coste del superpaso viene dado únicamente por el proceso de comunicación habitual desde el procesador principal al resto de procesadores. En total son $4k$ los elementos que recibe cada procesador, por lo que el coste de este superpaso es de

$$[4k(p-1)]g + l = 4(n-k)g + l \quad \text{flops.} \tag{5.22}$$

Coste del superpaso 2. En el superpaso 2 tenemos coste aritmético y coste de comunicación, por lo que los calculamos de forma separada.

Para obtener el coste aritmético, se deben tener en cuenta el número de operaciones aritméticas que realiza cualquiera de los procesadores P_i, para $i = 1, 2, \ldots, p-2$, ya que resuelven tres sistemas. Estas tareas, con su coste computacional, se resumen en los siguientes puntos

- La factorización LU de la matriz A_i requiere $3k - 3$ flops.
- La resolución del sistema $A_i \boldsymbol{y}_i = \boldsymbol{d}_i$, mediante la factorización LU requiere $5k - 4$ flops.

- La resolución del sistema $A_i \boldsymbol{z}_{2i-1} = \boldsymbol{e}_1$ supone un total de $4k - 3$ flops ya que, al ser e_1 el vector de términos independientes, no se realiza ninguna suma.
- La resolución del sistema $A_i \boldsymbol{z}_{2i} = \boldsymbol{e}_k$ requiere sólamente $2k - 1$ flops.

En consecuencia, necesitamos efectuar un total de

$$14k - 11 \quad \text{flops} \tag{5.23}$$

para completar las tareas aritméticas que se llevan a cabo en este superpaso.

Por otro lado, el coste de comunicación está relacionado con el envío de los vectores $\boldsymbol{y}_i$, para $i = 1, 2, \ldots, p-1$, y los vectores $\boldsymbol{z}_j$, para $j = 1, 2, \ldots, 2p-3$, al procesador principal. El procesador principal recibe dos vectores de tamaño k del procesador P_{p-1} y tres vectores de tamaño k de los procesadores no extremos. Esto significa que el procesador principal recibe un total de $3k(p-2) + 2k = 3n - 4k$ unidades de datos, por lo que el coste de comunicación de este superpaso es de

$$[3k(p-2) + 2k]g \quad \text{flops.} \tag{5.24}$$

En consecuencia, sumando los costes aritmético y computacional dados por las expresiones (5.23) y (5.24), tenemos que el coste total del superpaso 2 es de

$$(14k - 11) + (3n - 4k)g + l \quad \text{flops.} \tag{5.25}$$

Coste del superpaso 3. En este superpaso sólo se realizan operaciones aritméticas en el procesador principal y no se producen comunicaciones entre los procesadores. Las tareas que se realizan en el procesador principal, con su coste computacional son

- La obtención de la matriz H requiere $2p - 2$ flops.

- Para resolver el sistema (5.15), utilizando el método de eliminación de Gauss, son necesarios $8(2p-2)-7$ flops.
- El cálculo de la solución $\boldsymbol{x}$ a partir de la expresión (5.12) supone la realización de $4(n-k)$ flops.

En consecuencia, el coste del superpaso 3 es de

$$4n + 18p - 4k - 25 \quad \text{flops.} \tag{5.26}$$

Sumando las expresiones (5.22), (5.25) y (5.26) obtenemos que el coste computacional del algoritmo 5.1 es de

$$4n + 18p + 10k - 36 + (7n - 8m)g + 2l \quad \text{flops.}$$

5.3.3 Algoritmo BSP modificado basado en el método de Bondeli

En el algoritmo 5.1 advertimos la pérdida de paralelismo en el último superpaso, donde los cálculos se realizan en el procesador principal y el resto están inactivos. La ventaja que presenta es que minimiza el número de superpasos y, por tanto, el número de barreras de sincronización necesarias para ejecutar el algoritmo.

Con el fin de evitar esta pérdida de paralelismo en el superpaso 3 podemos considerar una modificación consistente en que cada procesador calcule y resuelva el sistema (5.15) de forma simultánea. Posteriormente, cada procesador calcula en paralelo las componentes de la solución final que le corresponden y las envía al procesador principal. El siguiente algoritmo recoge esta modificación respecto del algoritmo 5.1.

Algoritmo 5.2 Método divide y vencerás modificado según el modelo BSP para sistemas tridiagonales basado en el método divide y vencerás de Bondeli.

Superpaso 1

Comunicación de datos a los procesadores. Para $i = 1, 2, \ldots, p-1$, el procesador P_i recibe la submatriz A_i y el vector $\boldsymbol{d}_i$ desde el procesador principal.

Superpaso 2

1 Resolución de los subsistemas intermedios y envío de los vectores auxiliares al procesador principal.

  1.1 En el procesador P_0 se resuelve el sistema (5.19).

  1.2 En el procesador P_i, para $i = 1, 2, \ldots, p-2$, se resuelve el sistema (5.20).

  1.3 En el procesador P_{p-1} se resuelven los sistemas (5.21).

2 Comunicación. El procesador P_i, para $i = 0, 1, \ldots, p-1$, envía los vectores $\boldsymbol{y}_i$, $\boldsymbol{z}_{2i-1}$, $\boldsymbol{z}_{2i}$ al resto de procesadores, siendo $\boldsymbol{z}_{-1} = \boldsymbol{z}_{2p-2} = \boldsymbol{0}$.

Superpaso 3

Obtención de la solución final.

1 En el procesador P_i, para $i = 0, 1, \ldots, p-1$,

  1.1 Se calculan H y el vector β mediante las expresiones (5.16), (5.17) y (5.18).

  1.2 Se resuelve el sistema (5.15).

  1.3 Se calcula el vector solución parcial $\boldsymbol{x}_i$ utilizando la expresión (5.12).

2 Comunicación. El procesador P_i, para $i = 1, 2, \ldots, p-1$, envía su vector solución parcial $\boldsymbol{x}_i$ al procesador P_0.

La figura 5.2 nos muestra la ejecución del algoritmo 5.2 para el caso particular en que $p = 4$. Notemos las diferencias más importantes entre los algoritmos 5.1 y 5.2. En el algoritmo 5.2 no perdemos el paralelismo a lo largo de todo el proceso, como ya se ha mencionado;

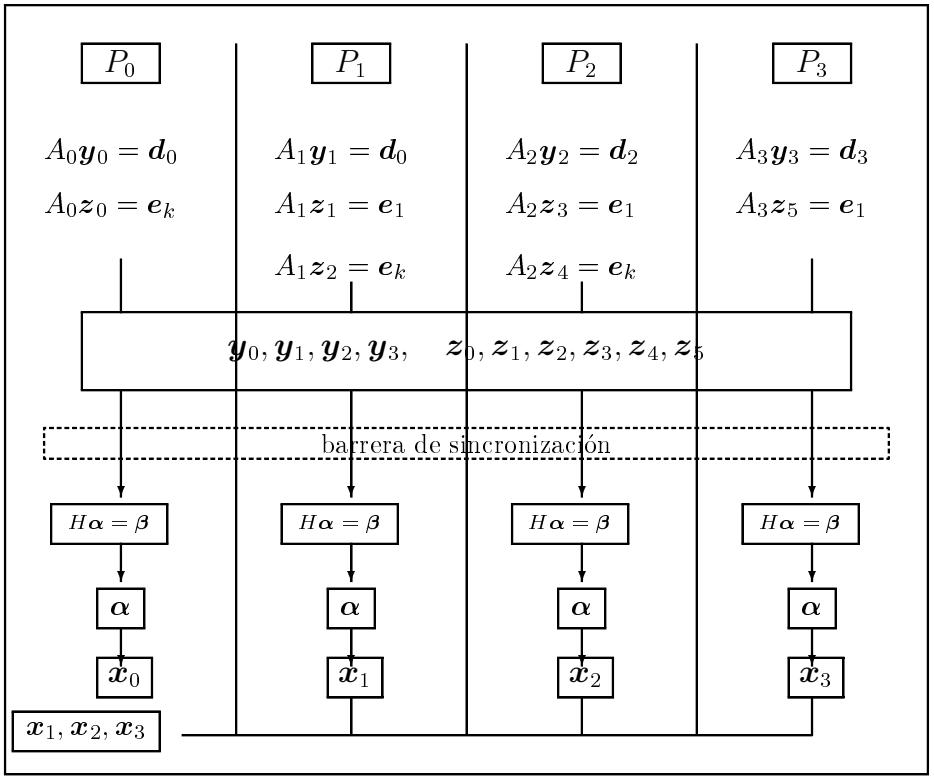


Figura 5.2: *Esquema de ejecución del algoritmo 5.2, para 4 procesadores.*

sin embargo, se requieren tres barreras de sincronización, ya que en el superpaso 3 hay una comunicación final de las soluciones parciales al procesador principal.

Otro aspecto que los diferencia es el proceso de comunicación que se lleva a cabo en el superpaso 2. En el algoritmo 5.2 se produce una comunicación donde cada procesador no extremo envía al resto de procesadores tres vectores de tamaño k. Los procesadores principal y final únicamente envía al resto de procesadores dos vectores de tamaño k. Este modelo de comunicación es un *broadcast* de cada procesador al resto. En el algoritmo 5.1 la comunicación es desde todos los procesadores al principal.

5.3.4 Coste computacional

Calculamos ahora los costes de cada uno de los superpasos que componen el algoritmo 5.2.

Coste del superpaso 1. Este superpaso es análogo al superpaso 1 del algoritmo 5.1, por lo que su coste computacional viene determinado por la expresión (5.22).

Coste del superpaso 2. La diferencia entre el superpaso 2 del algoritmo 5.2 y el superpaso 2 del algoritmo 5.1 se encuentra en la comunicación. Las tareas aritméticas que cada procesador desarrolla son las mismas, por lo que el coste aritmético viene dado por la expresión (5.23).

Para determinar el coste de comunicación de este superpaso hemos de tener en cuenta el máximo de las unidades de datos enviadas o recibidas por el procesador que más datos comunica o recibe. El número máximo de unidades de datos enviadas por los procesadores P_i, para $i = 1, 2, \ldots, p-2$, es de $3k(p-1)$ mientras que el número máximo de datos recibidos por los procesadores que más elementos reciben es $3k(p-2) + 2k$. Así, el número máximo de datos enviados o recibidos por cualquiera de estos procesadores es $3k(p-1)$, por lo que el modelo de comunicación es el de una $3k(p-1)$-relación. En consecuencia, el coste de comunicación es de

$$3k(p-1)g = 3(n-k)g \quad \text{flops.} \tag{5.27}$$

Por tanto, sumando las expresiones (5.23) y (5.27)obtenemos que el coste del superpaso 2 es de

$$14k - 11 + 3(n-k)g + l \quad \text{flops.} \tag{5.28}$$

Coste del superpaso 3. Para calcular el coste del superpaso 3 hemos de tener en cuenta el coste aritmético y de comunicación. El coste aritmético de este superpaso viene determinado por la ejecución de las siguientes tareas en cada uno de los procesadores

- La obtención de la matriz H requiere $2p-2$ flops.
- La resolución del sistema (5.15), utilizando el método de eliminación de Gauss, supone $8(2p-2)-7$ flops.
- El cálculo del vector solución parcial $\boldsymbol{x}_i$, a partir de la expresión (5.12), en cada uno de los procesadores centrales representa la realización de $4k$ operaciones.

En consecuencia, el coste aritmético es de

$$4k + 18(p-1) - 7 \quad \text{flops.} \tag{5.29}$$

Por otro lado, el coste de comunicación del superpaso viene determinado por la comunicación desde cada procesador al procesador principal de su vector correspondiente de soluciones parciales, lo cual representa un coste de

$$n - k \quad \text{flops.} \tag{5.30}$$

Así, si sumamos las expresiones (5.29) y (5.30) obtenemos que el coste computacional de este superpaso es de

$$4k + 18(p-1) - 7 + (n-k)g + l \quad \text{flops.} \tag{5.31}$$

Como consecuencia de todo lo dicho, sumando las expresiones (5.22), (5.28) y (5.31) obtenemos que el coste computacional del algoritmo 5.2 es de

$$18k + 18p - 36 + 8(n-k)g + 3l \quad \text{flops.}$$

5.3.5 Algoritmo BSP utilizando el método *recursive doubling*

Al igual que sucedía en el capítulo 4 con el algoritmo 4.2, los algoritmos 5.1 y 5.2 se basan en la resolución de un sistema tridiagonal auxiliar cuya solución nos permite obtener la solución del sistema inicial. Este sistema tridiagonal auxiliar se resuelve de forma secuencial en estos algoritmos. Podemos plantearnos, de la misma forma que en el capítulo 4 con los algoritmos 4.3, 4.4 y 4.5, la resolución de este sistema tridiagonal en paralelo utilizando todos los procesadores disponibles. En el capítulo 4 comprobamos que las diferencias entre los algoritmos eran mínimas y que los tiempos de los algoritmos que utilizaban diversos métodos de resolución en paralelo del sistema tridiagonal auxiliar eran prácticamente idénticos. Por esta razón, únicamente consideramos un nuevo algoritmo basado en el método de Bondeli que resuelva el sistema tridiagonal auxiliar en paralelo utilizando el método del *recursive doubling*. El motivo por el que utilizamos este método en lugar del de reducción cíclica se basa en que se obtienen tiempos ligeramente mejores en el IBM SP2 y en el cluster de Pentiums, como se aprecia en la sección 4.6.

A partir de las características del método *recursive doubling* expuestas en la sección 1.3.2, en esta sección estudiamos un método del tipo divide y vencerás basado en el método divide y vencerás de Bondeli, utilizando el método *recursive doubling* para resolver el sistema 5.15 en paralelo. Así, la diferencia básica entre este nuevo algoritmo y el algoritmo 5.1 es la forma en que se resuelve el sistema intermedio (5.15).

La matriz de coeficientes H del sistema (5.15), para el caso $p = 4$, viene dada por

$$H = \begin{bmatrix} \delta_0 & \mu_0 & & & & \\ \nu_1 & \delta_1 & \mu_1 & & & \\ & \nu_2 & \delta_2 & \mu_2 & & \\ & & \nu_3 & \delta_3 & \mu_3 & \\ & & & \nu_4 & \delta_4 & \mu_4 \\ & & & & \nu_5 & \delta_5 \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_k^T \boldsymbol{z}_0 & \frac{1}{c_{k+1}} & & & & \\ \frac{1}{b_k} & \boldsymbol{e}_1^T \boldsymbol{z}_1 & \boldsymbol{e}_1^T \boldsymbol{z}_2 & & & \\ & \boldsymbol{e}_k^T \boldsymbol{z}_1 & \boldsymbol{e}_k^T \boldsymbol{z}_2 & \frac{1}{c_{2k+1}} & & \\ & & \frac{1}{b_{2k}} & \boldsymbol{e}_1^T \boldsymbol{z}_3 & \boldsymbol{e}_1^T \boldsymbol{z}_4 & \\ & & & \boldsymbol{e}_k^T \boldsymbol{z}_3 & \boldsymbol{e}_k^T \boldsymbol{z}_4 & \frac{1}{c_{3k+1}} \\ & & & & \frac{1}{b_{3k}} & \boldsymbol{e}_1^T \boldsymbol{z}_5 \end{bmatrix} \tag{5.32}$$

Si observamos detenidamente la matriz (5.32) notamos, en primer lugar, que la situación de los elementos en los distintos procesadores sigue un esquema similar al del algoritmo 4.3. Los elementos de la primera fila se encuentran en el procesador P_0, los elementos de la segunda y tercera filas se encuentran en el procesador P_1, los de la tercera y cuarta filas están en P_2, mientras que el procesador P_3 tiene los elementos de la última fila. Una situación análoga sucede con los elementos del vector $\boldsymbol{\beta}$. En segundo lugar, observamos que, a diferencia de lo que ocurría en el algoritmo 4.3, la formación de la matriz H supone que cada uno de los procesadores P_i, para $i = 1, 2, \ldots, p-2$, realiza dos divisiones para calcular cada uno de los elementos ν_i, μ_i y δ_i, para $i = 0, 2, \ldots, 2p-4$.

En consecuencia, el nuevo algoritmo que resume estas características se diferencia básicamente del algoritmo 5.1 en los siguientes puntos:

(i) En el superpaso 2 del algoritmo 5.1, después de resolver los sistemas (5.19), (5.20) y (5.21) en los procesadores correspondientes, tenemos en el procesador P_i, para $i = 0, 1, \ldots, p-1$, los elementos de las filas $(2i-1)$-ésima y $2i$-ésimas de H y $\boldsymbol{\beta}$, respectivamente.

(ii) El paso de comunicación del superpaso 2 se elimina.

(iii) A partir del superpaso 2 se aplica el método *recursive doubling* para resolver el sistema (5.15).

El modelo de comunicación que se emplea para la ejecución del método *recursive doubling* es el mismo que se emplea en el algoritmo 4.4, ya que es el modelo que minimiza el número de elementos que circulan por la red utilizando un esquema de comunicación de tipo *fan-in*. Así mismo, el cálculo de las matrices B_i, para $i = 0, 1, \ldots, 2p-3$, se realiza utilizando las expresiones que aparecen en la sección 1.3.2.

De acuerdo con los comentarios anteriores y teniendo en cuenta la descripción del método *recursive doubling* de la sección 1.3.2, podemos dar el siguiente algoritmo BSP que resume todas estas características.

Algoritmo 5.3 Algoritmo BSP divide y vencerás basado en el método de Bondeli utilizando el método *recursive doubling*.

Superpaso 1

Comunicación de datos a los procesadores. Para $i = 1, 2, \ldots, p-1$, el procesador P_i recibe la submatriz A_i y el vector $\boldsymbol{d}_i$ desde el procesador principal. Inicialmente tomamos $\mu = -1$.

Superpaso 2

1 Resolución de los subsistemas intermedios.

- 1.1 En el procesador P_0 se resuelve el sistema (5.19).
- 1.2 En el procesador P_i, para $i = 1, 2, \ldots, p-2$, se resuelve el sistema (5.20).
- 1.3 En el procesador P_{p-1} se resuelve el sistema (5.21).
- 1.4 El procesador P_i, para $i = 0, 1, \ldots, p-1$,
  - 1.4.1 Calcula las matrices B_{2i-1} y B_{2i}.
  - 1.4.2 Calcula el producto $B[2i \mid 2i-1]$.

2 Comunicación.

- 2.1 El procesador P_i, para $i = 1, 2, \ldots, p-1$, comunica al procesador P_0 las matrices B_{2i-1} y B_{2i}.

2.2 El procesador P_{2i+1}, para $i = 0, 1, \ldots, 2^{m-1} - 1$, comunica al procesador P_{2i} la matriz $B[2i \mid 2i - 1]$.

Superpaso $m - q + 2$, para $q = m - 1, m - 2, \ldots, 2, 1$

1 El procesador $P_{2^{m-q}i}$, para $i = 0, 1, \ldots, 2^q - 1$ calcula la matriz

$$B[2^{m-q} + 2^{m-q+1}i + 2(\mu + 1) \mid 2^{m-q+1}i - 1].$$

2 El procesador $P_{2^{m-q}+2^{m-q+1}i}$, para $i = 0, 1, \ldots, \lfloor \frac{2^q-1}{2} \rfloor$ comunica al procesador $P_{2^{m-q+1}i}$ la matriz

$$B[2^{m-q} + 2^{m-q+1}(2i + 1) + 2(\mu + 1) \mid 2^{m-q+1}(2i + 1) - 1].$$

3 Actualizamos μ, como $\mu = 2(\mu + 1)$

Superpaso $m + 2$

El procesador P_0,

1 Calcula $B[2p - 2 \mid -1]$.

2 Calcula α_0 utilizando la expresión (4.26) y, mediante la expresión (4.25), obtiene el resto de componentes α_i, para $i = 1, 2, \ldots, 2p-3$.

3 Comunica las componentes $\alpha_{2i-1}, \alpha_{2i}$ al procesador P_i, para $i = 1, 2, \ldots, p - 1$.

Superpaso $m + 3$

El procesador P_i, para $i = 0, 1, \ldots, p - 1$,

1 Calcula $\boldsymbol{x}_i$ mediante la ecuación (5.12).

2 Envía el vector de soluciones parciales $\boldsymbol{x}_i$ al procesador principal.

Destacamos las escasas diferencias existentes entre este algoritmo y el algoritmo 4.4 basado en la fórmula de Sherman-Morrison-Woodbury. En ambos, el proceso de cálculo y comunicaciones que seguimos es muy parecido. Las diferencias se encuentran en la forma en la que se obtiene la matriz tridiagonal auxiliar que nos permite obtener la solución del sistema inicial y en algunos elementos de dicha matriz. Estas analogías van a provocar un coste y un comportamiento similar de ambos algoritmos.

5.3.6 Coste computacional

Coste del superpaso 1. Este superpaso es análogo al superpaso 1 del algoritmo 5.1, por lo que su coste viene dado por la expresión (5.22).

Coste del superpaso 2. Los apartados 1.1, 1.2 y 1.3 del superpaso 2 del algoritmo 5.3 coinciden con los apartados 1.1, 1.2 y 1.3 del superpaso 2 del algoritmo 5.1, por lo que el coste de estos apartados viene dado por la expresión (5.23), es decir, $14k - 11$ flops. El cálculo de las matrices B_{2i-1} y B_{2i}, que aparecen en la sección 1.3.2, supone 6 operaciones, mientras que por la forma especial que tienen estas matrices el producto de matrices $B[2i \mid 2i-1]$ requiere 5 operaciones. Por lo tanto, el coste aritmético de este superpaso es de $14k$ flops.

Por otro lado notemos que el modelo de comunicación que seguimos en este superpaso coincide con el modelo de comunicación del superpaso 2 del algoritmo 4.4, que viene dado por la expresión (4.33), es decir, $(6p-3)g$ flops.

Así pues, sumando los costes aritmético y de comunicación se obtiene que el coste computacional del superpaso 2 es de

$$14k + (6p-3)g + l \quad \text{flops.} \tag{5.33}$$

Coste de los superpasos $\boldsymbol{m-q+2}$, para $q = m-1, m-2, \ldots, 2, 1$. Tanto las operaciones aritméticas como las comunicaciones que se realizan en este conjunto de $m-1$ superpasos coincidene exactamente con las que se realizan en el algoritmo 4.4, por lo que su

coste computacional vendrá dado por la expresión (4.35), es decir,

$$20(m-1)+6(m-1)g+(m-1)l \quad \text{flops.} \tag{5.34}$$

Coste del superpaso $m+2$. De nuevo, este superpaso coincide exactamente con el superpaso $m+2$ del algoritmo 4.4, por lo que su coste viene dado por la expresión (4.36), es decir

$$10p+6+(2p-3)g+l \quad \text{flops.} \tag{5.35}$$

Coste del superpaso $m+3$. Finalmente, el cálculo de las soluciones parciales $\boldsymbol{x}_i$ mediante la expresión (5.12) requiere $4k$ flops, mientras que el coste de comunicación es de $(n-k)g$ flops, ya que el procesador principal recibe $n-k$ datos desde todos los procesadores.

Por tanto, el coste de este superpaso es de

$$4k+(n-k)g+l \quad \text{flops.} \tag{5.36}$$

Sumando las expresiones (5.22), (5.33), (5.34), (5.35) y (5.36) y realizando las simplificaciones adecuadas obtenemos el coste total del algoritmo 5.3, que es de

$$18k+20m+10p-14+(5n-5k+6m+8p-12)\,g+(m+3)l \quad \text{flops.} \tag{5.37}$$

Como cabía esperar dadas las escasas diferencias existentes entre este algoritmo y el algoritmo 4.4, los costes teóricos obtenidos son casi idénticos, como se desprende de las expresiones (5.37) y (4.32).

5.4 Resultados teóricos y comparaciones

En esta sección presentamos los tiempos teóricos de ejecución de los algoritmos 5.2, 5.3 y 5.4 para las tres máquinas paralelas con las que se ha venido realizando las comparaciones en los capítulos anteriores. Recordemos que los parámetros de estas máquinas vienen dados por la tabla 2.1.

5.4.1 Tiempos en un IBM SP2

Las tablas 5.1 y 5.2 resumen, respectivamente, los tiempos comparados de los algoritmos estudiados en este capítulo para un IBM SP2 dotado con un switch de alto rendimiento y una conexión de tipo ethernet.

Analizamos de forma separada los resultados en esta máquina para los dos tipos de conexión que se utilizan. De la tabla 5.1 se observan varios hechos destacables. El primero de ellos es que el algoritmo 5.3 siempre es el más rápido de los tres para matrices con $n \geq 4096$. Para tamaños inferiores el más rápido es el algoritmo 5.1, aunque no podemos decir que existan diferencias significativas entre ellos dado el tamaño tan pequeño de las matrices. Para cualquier valor de n, el algoritmo 5.2 resulta el más lento.

Otro aspecto destacable es que mientras que en los algoritmos 5.1 y 5.2 los tiempos crecen al aumentar el número de procesadores de 2 a 4 y de 4 a 8, no sucede lo mismo con el algoritmo 5.3. El algoritmo 5.3 reduce los tiempos de ejecución al pasar de 2 a 4 procesadores para matrices con valores de n superiores a 32768, como se aprecia en la tabla 5.1. Cuando se aumenta de 4 a 8 el número de procesadores, vuelven a aumentar los tiempos aunque no excesivamente.

También se observa que las diferencias de tiempo entre el algoritmo más rápido y el más lento crecen notablemente al aumentar el número de procesadores. Por ejemplo, si consideramos el caso $n = 4194304$ vemos que la diferencia en los tiempos entre el algoritmo 5.3 (el más rápido) y el algoritmo 5.2 (el más lento), para $p = 2$, es de 0.7621 segundos, mientras que para $p = 4$, la diferencia en tiempos

IBM SP2 switch									
	p								
	2			4			8		
n	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3
512	0.0005	0.0007	0.0006	0.0007	0.0009	0.0010	0.0010	0.0012	0.0016
1024	0.0009	0.0011	0.0010	0.0012	0.0014	0.0014	0.0015	0.0017	0.0020
2048	0.0016	0.0019	0.0016	0.0021	0.0023	0.0020	0.0025	0.0028	0.0027
4096	0.0031	0.0036	0.0030	0.0038	0.0042	0.0033	0.0045	0.0048	0.0040
8192	0.0060	0.0070	0.0056	0.0074	0.0079	0.0059	0.0086	0.0090	0.0068
16384	0.0118	0.0138	0.0109	0.0145	0.0154	0.0111	0.0168	0.0173	0.0122
32768	0.0234	0.0274	0.0216	0.0287	0.0303	0.0215	0.0332	0.0339	0.0231
65536	0.0467	0.0547	0.0428	0.0570	0.0602	0.0423	0.0659	0.0672	0.0450
131072	0.0932	0.1091	0.0854	0.1137	0.1199	0.0839	0.1313	0.1338	0.0887
262144	0.1862	0.2180	0.1704	0.2271	0.2394	0.1671	0.2621	0.2669	0.1762
524288	0.3722	0.4358	0.3406	0.4540	0.4783	0.3334	0.5237	0.5330	0.3510
1048576	0.7442	0.8714	0.6809	0.9077	0.9563	0.6662	1.0470	1.0654	0.7008
2097152	1.4883	1.7425	1.3614	1.8151	1.9121	1.3316	2.0936	2.1301	1.4002
4194304	2.9765	3.4847	2.7226	3.6300	3.8237	2.6625	4.1867	4.2595	2.7992

Tabla 5.1: *Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un IBM SP2 dotado con un switch de alto rendimiento.*

IBM SP2 ethernet									
	p								
	2			4			8		
n	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3
512	0.0071	0.0096	0.0076	0.0230	0.0284	0.0227	0.0829	0.0989	0.0776
1024	0.0126	0.0169	0.0123	0.0422	0.0514	0.0372	0.1556	0.1849	0.1314
2048	0.0237	0.0316	0.0216	0.0807	0.0975	0.0660	0.3041	0.3568	0.2388
4096	0.0460	0.0610	0.0403	0.1577	0.1896	0.1237	0.5991	0.7006	0.4538
8192	0.0904	0.1198	0.0775	0.3116	0.3738	0.2391	1.1891	1.3884	0.8838
16384	0.1793	0.2374	0.1521	0.6194	0.7421	0.4698	2.3691	2.7638	1.7437
32768	0.3571	0.4726	0.3012	1.2351	1.4788	0.9313	4.7291	5.5146	3.4635
65536	0.7127	0.9429	0.5994	2.4664	2.9523	1.8543	9.4490	11.0163	6.9031
131072	1.4239	1.8836	1.1959	4.9290	5.8991	3.7004	18.8888	22.0197	13.7823
262144	2.8463	3.7650	2.3887	9.8543	11.7928	7.3924	37.7685	44.0264	27.5408
524288	5.6910	7.5278	4.7745	19.7049	23.5802	14.7766	75.5278	88.0399	55.0577
1048576	11.3806	15.0533	9.5460	39.4060	47.1550	29.5449	151.0464	176.0668	110.0916
2097152	22.7596	30.1044	19.0891	78.8083	94.3046	59.0814	302.0837	352.1207	220.1593
4194304	45.5178	60.2066	38.1752	157.6128	188.6039	118.1546	604.1583	704.2285	440.2947

Tabla 5.2: *Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un IBM SP2 utilizando conexión ethernet.*

entre ambos algoritmos es de 1.46 segundos. Esta diferencia de tiempos significa que ejecutar el algoritmo 5.3 supone un ahorro de tiempo del 34% aproximadamente frente a la ejecución del algoritmo 5.2, lo que representa un valor significativo.

La tabla 5.2 recoge los valores teóricos cuando se utiliza una conexión ethernet. Las características más notables descritas en el caso de conexión con switch se pueden extrapolar para esta conexión. El algoritmo 5.3 sigue siendo el más rápido y el algoritmo 5.2 es el más lento. Ahora esta característica es independiente del valor de n.

Una característica que no se observa en este caso es la reducción de tiempos en el algoritmo 5.3 al pasar de 2 a 4 procesadores; cuando se utiliza una conexión ethernet los tiempos aumentan de forma considerable para los tres algoritmos. En general podemos decir que al duplicar el número de procesadores los tiempos aumentan en un factor superior a tres. Esto nos da una idea del comportamiento paralelo altamente negativo de estos algoritmos en estas máquinas. Sin duda, el peso de las comunicaciones y el alto coste de las mismas cuando se utiliza este tipo de conexión determina este comportamiento negativo al duplicar el número de procesadores.

En cuanto a las diferencias de tiempo que se miden para los algoritmos 5.2 y 5.3, que son el más lento y más rápido respectivamente, debemos decir que aumentan ligeramente respecto al caso con switch, aunque no de forma significativa; los porcentajes de diferencia que antes se aproximaban al 35% ahora están entre e 38 y el 40%.

5.4.2 Tiempos en un CRAY T3D

Las tablas 5.3 y 5.4 muestran los resultados teóricos esperados en un CRAY T3D para los tres algoritmos estudiados en este capítulo.

La primera conclusión que podemos extraer al observar las tablas 5.3 y 5.4 es que, a diferencia de lo que ocurría en el IBM SP2, en esta máquina el algoritmo más lento es el algoritmo 5.1 a partir de 4 procesadores. El algoritmo más rápido sigue siendo el algoritmo 5.3, para cualquier número de procesadores y para cualquier valor de n. Para dos procesadores el algoritmo 5.2 ofrece peores resultados que el algoritmo 5.1; sin embargo, como ya se ha comentado, al aumentar a 4 el número de procesadores, los resultados que se obtienen al

CRAY T3D												
	p											
	2			4			8			16		
n	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3
2048	0.0018	0.0018	0.0018	0.0015	0.0012	0.0011	0.0014	0.0010	0.0009	0.0016	0.0011	0.0009
4096	0.0035	0.0036	0.0034	0.0029	0.0023	0.0021	0.0027	0.0019	0.0015	0.0029	0.0019	0.0014
8192	0.0069	0.0072	0.0068	0.0057	0.0046	0.0041	0.0054	0.0036	0.0029	0.0054	0.0034	0.0025
16384	0.0138	0.0143	0.0136	0.0113	0.0091	0.0080	0.0106	0.0071	0.0056	0.0106	0.0065	0.0047
32768	0.0275	0.0285	0.0270	0.0226	0.0181	0.0160	0.0210	0.0140	0.0111	0.0209	0.0126	0.0091
65536	0.0549	0.0569	0.0540	0.0451	0.0361	0.0318	0.0419	0.0277	0.0220	0.0415	0.0249	0.0180
131072	0.1098	0.1137	0.1079	0.0902	0.0722	0.0636	0.0837	0.0553	0.0439	0.0828	0.0495	0.0356
262144	0.2196	0.2272	0.2158	0.1803	0.1443	0.1271	0.1672	0.1105	0.0876	0.1652	0.0986	0.0710
524288	0.4391	0.4544	0.4315	0.3605	0.2884	0.2541	0.3344	0.2208	0.1749	0.3302	0.1969	0.1416
1048576	0.8782	0.9088	0.8630	0.7210	0.5768	0.5080	0.6686	0.4414	0.3497	0.6600	0.3936	0.2829
2097152	1.7564	1.8176	1.7258	1.4419	1.1535	1.0159	1.3371	0.8827	0.6992	1.3197	0.7868	0.5655
4194304	3.5128	3.6351	3.4515	2.8837	2.3070	2.0317	2.6740	1.7653	1.3983	2.6392	1.5732	1.1308

Tabla 5.3: *Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un CRAY T3D utilizando* 2, 4, 8 *y* 16 *procesadores.*

CRAY T3D												
	p											
	32			64			128			256		
n	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3
2048	0.0017	0.0012	0.0009	0.0019	0.0014	0.0010	0.0024	0.0020	0.0015	0.0032	0.0028	0.0024
4096	0.0031	0.0021	0.0015	0.0032	0.0021	0.0015	0.0037	0.0027	0.0020	0.0046	0.0037	0.0028
8192	0.0058	0.0037	0.0026	0.0058	0.0036	0.0024	0.0064	0.0043	0.0030	0.0074	0.0053	0.0038
16384	0.0113	0.0070	0.0048	0.0110	0.0065	0.0043	0.0118	0.0073	0.0049	0.0130	0.0086	0.0058
32768	0.0223	0.0136	0.0092	0.0213	0.0122	0.0080	0.0226	0.0135	0.0088	0.0242	0.0153	0.0100
65536	0.0442	0.0268	0.0180	0.0421	0.0237	0.0155	0.0442	0.0258	0.0167	0.0467	0.0285	0.0184
131072	0.0880	0.0531	0.0356	0.0836	0.0468	0.0305	0.0873	0.0504	0.0323	0.0916	0.0550	0.0351
262144	0.1757	0.1058	0.0709	0.1665	0.0929	0.0605	0.1735	0.0996	0.0637	0.1814	0.1080	0.0685
524288	0.3510	0.2112	0.1413	0.3324	0.1850	0.1204	0.3459	0.1981	0.1264	0.3609	0.2140	0.1353
1048576	0.7016	0.4220	0.2823	0.6642	0.3693	0.2402	0.6908	0.3950	0.2517	0.7201	0.4259	0.2689
2097152	1.4028	0.8436	0.5643	1.3277	0.7380	0.4798	1.3806	0.7887	0.5024	1.4384	0.8499	0.5362
4194304	2.8053	1.6869	1.1281	2.6548	1.4753	0.9590	2.7601	1.5762	1.0038	2.8750	1.6978	1.0707

Tabla 5.4: *Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un CRAY T3D para* 32, 64, 128 *y* 256 *procesadores.*

ejecutar el algoritmo 5.2 son sensiblemente mejores que los obtenidos con el algoritmo 5.1. Este hecho no resulta extraño si tenemos en cuenta que la diferencia básica entre ambos algoritmos radica en que el algoritmo 5.3 calcula en paralelo las soluciones finales, mientras que el algoritmo 5.1 las calcula de forma secuencial. Una máquina de este tipo con un coste de las comunicaciones muy bajo permite que la resolución del sistema tridiagonal auxiliar se realice en paralelo y resulte más ventajoso que su resolución secuencial.

También se observa que al aumentar inicialmente el número de procesadores, los tiempos disminuyen, como también cabe esperar en una máquina altamente paralela de estas características. Los tiempos se van reduciendo hasta llegar a un cierto número de procesadores, a partir del cuál vuelven a ir aumentando. Esta característica no es novedosa, ya se había producido anteriormente en los capítulos 3 y 4, cuando se expusieron los resultados numéricos para los métodos basados en las fórmulas de Sherman-Morrison y Sherman-Morrison-Woodbury. Esto nos permite hablar de un número de procesadores óptimo para la ejecución de los distintos algoritmos y este número de procesadores óptimo depende del valor de n. Así, la tabla 5.5 recoge el número óptimo de procesadores para los distintos tamaños de matrices cuando se ejecutan los tres algoritmos estudiados en este capítulo.

Nótese la gran similitud de esta tabla con la tabla 4.5 que nos permite afirmar, salvo para algunos casos concretos con valores pequeños de n, que el número óptimo de procesadores al ejecutar estos algoritmos en esta máquina es de 64.

5.4.3 Tiempos en un cluster de Pentiums

En esta sección se analizan los resultados teóricos sobre un cluster de Pentiums, siguiendo un esquema similar al descrito para las máquinas anteriores. La tabla 5.6 recoge los tiempos teóricos de ejecución de los tres algoritmos estudiados en este capítulo para 2, 4 y 8 procesadores.

La tabla 5.6 nos muestra claramente que el algoritmo 5.3 es el más rápido para cualquier número de procesadores siempre que $n > 2048$. Para valores de n muy pequeños el algoritmo más rápido es el algoritmo 5.1, lo que no resulta extraño ya que la matriz tridiagonal auxiliar no es lo suficientemente grande como para que compense, desde el punto de vista computacional, su resolución en

CRAY T3D			
n	Alg. 5.1	Alg. 5.2	Alg. 5.3
2048	8	8	16
4096	8	16	16
8192	16	16	16
16384	16	64	64
32768	64	64	64
65536	64	64	64
131072	64	64	64
262144	64	64	64
524288	64	64	64
1048576	64	64	64
2097152	64	64	64
4194304	64	64	64

Tabla 5.5: *Número óptimo de procesadores en un CRAY T3D para tamaños de matrices comprendidos entre* $n = 2048$ *y* $n = 4194304$.

Cluster de Pentiums									
	p								
	2			4			8		
n	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3	Alg. 5.1	Alg. 5.2	Alg. 5.3
512	0.0005	0.0006	0.0005	0.0008	0.0010	0.0011	0.0012	0.0016	0.0019
1024	0.0008	0.0010	0.0008	0.0013	0.0016	0.0014	0.0018	0.0022	0.0023
2048	0.0015	0.0019	0.0014	0.0023	0.0027	0.0022	0.0029	0.0035	0.0031
4096	0.0028	0.0036	0.0025	0.0042	0.0050	0.0036	0.0052	0.0060	0.0047
8192	0.0054	0.0070	0.0048	0.0082	0.0096	0.0055	0.0097	0.0111	0.0079
16384	0.0107	0.0138	0.0094	0.0161	0.0188	0.0125	0.0187	0.0212	0.0143
32768	0.0213	0.0273	0.0185	0.0319	0.0372	0.0243	0.0367	0.0415	0.0271
65536	0.0425	0.0544	0.0367	0.0636	0.0739	0.0479	0.0727	0.0821	0.0528
131072	0.0848	0.1086	0.0731	0.1269	0.1474	0.0951	0.1448	0.1633	0.1042
262144	0.1695	0.2171	0.1459	0.2535	0.2944	0.1895	0.2889	0.3256	0.2069
524288	0.3388	0.4339	0.2915	0.5067	0.5885	0.3783	0.5771	0.6503	0.4124
1048576	0.6775	0.8677	0.5826	1.0131	1.1765	0.7558	1.1535	1.2997	0.8233
2097152	1.3549	1.7351	1.1650	2.0259	2.3526	1.5110	2.3064	2.5985	1.6451
4194304	2.7097	3.4700	2.3298	4.0516	4.7047	3.0213	4.6120	5.1962	3.2887

Tabla 5.6: *Tiempos teóricos para los algoritmos 5.1, 5.2 y 5.3 medidos en un Cluster de Pentiums para* 2, 4 *y* 8 *procesadores.*

paralelo. Sin embargo, cuando trabajamos con valores grandes de n, los tiempos de ejecución del algoritmo 5.3 demuestran que resulta muy conveniente utilizar algún método paralelo para resolver el sistema auxiliar (5.15). Así, por ejemplo, para $n = 4096$ y $p = 4$, la diferencia de tiempos entre el algoritmo 5.1 y el algoritmo 5.3 es de 0.0006 segundos a favor del algoritmo 5.3, lo que representa un 14% del tiempo total. Sin embargo, si tomamos $n = 4194304$ y seguimos con $p = 4$, ahora la diferencia de tiempos entre ambos algoritmos es de 1.03 segundos, lo que representa un 25% del tiempo total. Esto significa que resolviendo un sistema tridiagonal de este tamaño con $p = 4$, podemos ahorrar un 25% del tiempo total si utilizamos el algoritmo 5.3 frente al algoritmo 5.1, que es bastante considerable. Si duplicamos el número de procesadores, el comportamiento es el mismo, con pequeñas variaciones en los porcentajes. Para $n = 4096$ ahora el ahorro en tiempo es del 9% únicamente, mientras que si aumentamos hasta $n = 4194304$ el ahorro en tiempo es del 28%.

También destacamos que el comportamiento paralelo de estos algoritmos no es demasiado bueno, ya que aumentan los tiempos al aumentar el número de procesadores, independientemente del tamaño de la matriz y del número de procesadores. El incremento de tiempos es mayor cuando pasamos de 2 a 4 procesadores que cuando duplicamos a 8 procesadores. Por ejemplo, si consideramos el algoritmo 5.1 para $n = 4194304$ tenemos un aumento del 33% al pasar de 2 a 4 procesadores, mientras que el porcentaje disminuye hasta el 12% al pasar de 4 a 8 procesadores. Esta característica se repite para los otros algoritmos, si bien con pequeñas variaciones en los porcentajes.

5.4.4 Estudio del speedup

En esta sección realizamos un estudio del *speedup* y de la eficiencia para los algoritmos 5.1, 5.2 y 5.3 que hemos estudiado a lo largo de este capítulo.

En la tabla 5.7 aparecen los valores del *speedup* y la eficiencia en las tres máquinas que venimos utilizando para obtener los resultados teóricos. Los mejores valores se obtienen en el CRAY T3D, donde se alcanza un valor del 70% en el algoritmo 4.5 para $p = 2$. Al aumentar el número de procesadores va disminuyendo la eficiencia de forma notable. Este descenso se hace más notorio en el IBM SP2 y en el cluster. Los resultados más negativos se obtienen en el IBM SP2 con conexión ethernet, al igual que ha venido ocurriendo en los

IBM SP2 switch						
	speedup			eficiencia		
p	Alg. 5.1	5.2	5.3	Alg. 5.1	5.2	Alg. 5.3
2	0.72	0.62	0.79	36.18	30.90	39.55
4	0.59	0.56	0.81	14.83	14.08	20.22
8	0.51	0.51	0.77	6.43	6.32	9.62

(a) *IBM SP2 switch*

IBM SP2 ethernet						
	speedup			eficiencia		
p	Alg. 5.1	5.2	5.3	Alg. 5.1	5.2	Alg. 5.3
2	0.05	0.04	0.06	2.37	1.79	2.82
4	0.01	0.01	0.02	0.34	0.29	0.46
8	0.004	0.003	0.005	0.05	0.04	0.06

(b) *IBM SP2 ethernet*

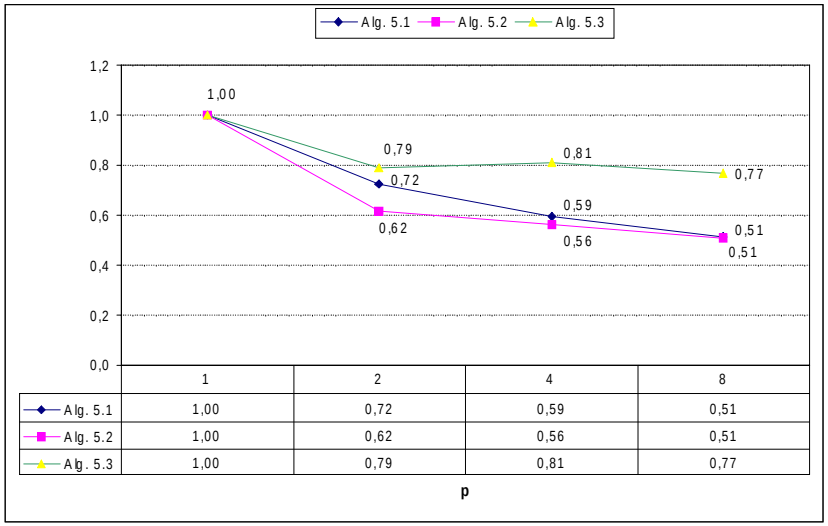
CRAY T3D						
	speedup			eficiencia		
p	Alg. 5.1	5.2	Alg. 5.3	Alg. 5.1	5.2	Alg. 5.3
2	1.38	1.33	1.40	68.90	66.58	70.13
4	1.68	2.10	2.38	48.97	52.46	59.57
8	1.81	2.74	3.46	22.63	34.28	43.27
16	1.83	3.08	4.28	11.46	19.23	26.75
32	1.72	2.87	4.29	5.39	8.97	13.40
64	1.82	3.28	5.05	3.38	6.07	9.34
128	1.75	3.07	4.82	1.37	2.40	3.76
256	1.68	2.85	4.51	0.66	1.11	1.76

(c) *CRAY T3D*

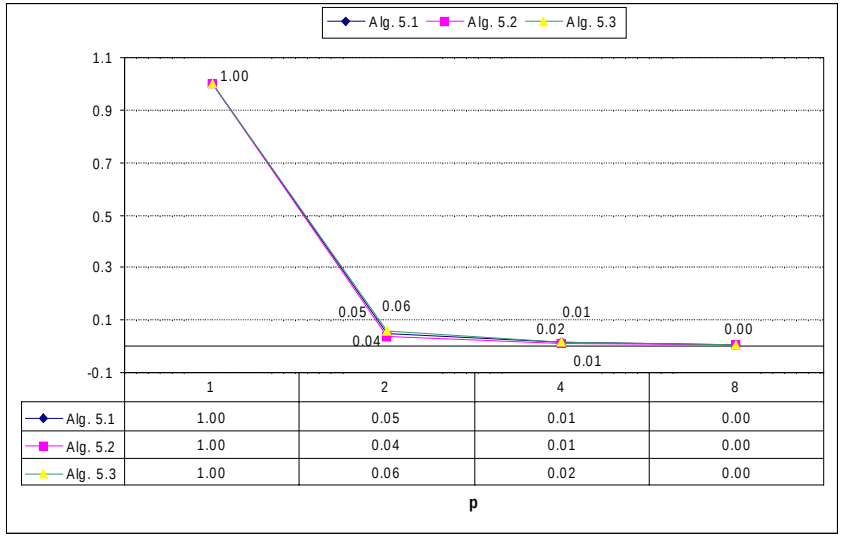
Cluster de Pentiums						
	speedup			eficiencia		
p	Alg. 5.1	5.2	Alg. 5.3	Alg. 5.1	5.2	Alg. 5.3
2	0.24	0.18	0.27	11.74	9.17	13.66
4	0.16	0.14	0.21	3.93	3.38	5.26
8	0.14	0.12	0.19	1.72	1.53	2.42

(d) *Cluster de Pentiums*

Tabla 5.7: *Speedup y eficiencia de los algoritmos 5.1, 5.2 y 5.3 para* $n = 2097152$.



(a) *IBM SP2 switch*



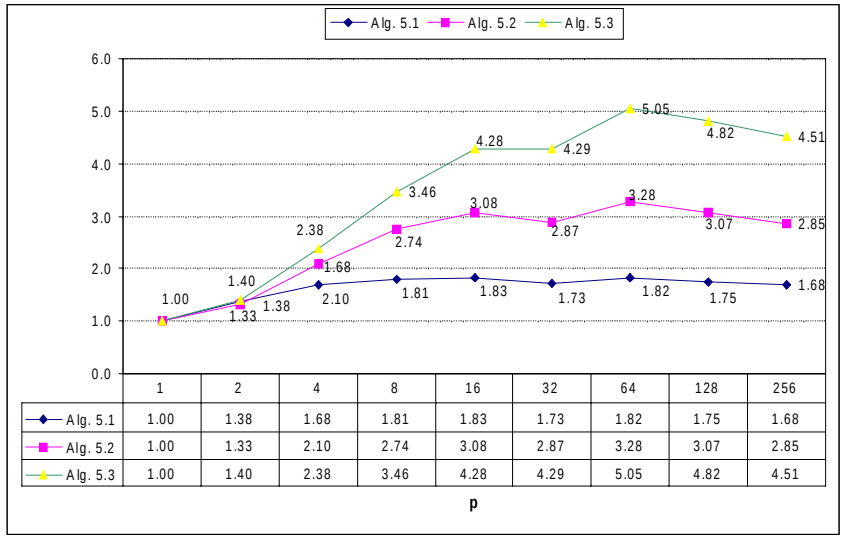
(b) *IBM SP2 ethernet*

Figura 5.3: *Valores del speedup en un IBM SP2*

capítulos anteriores con todos los algoritmos estudiados.

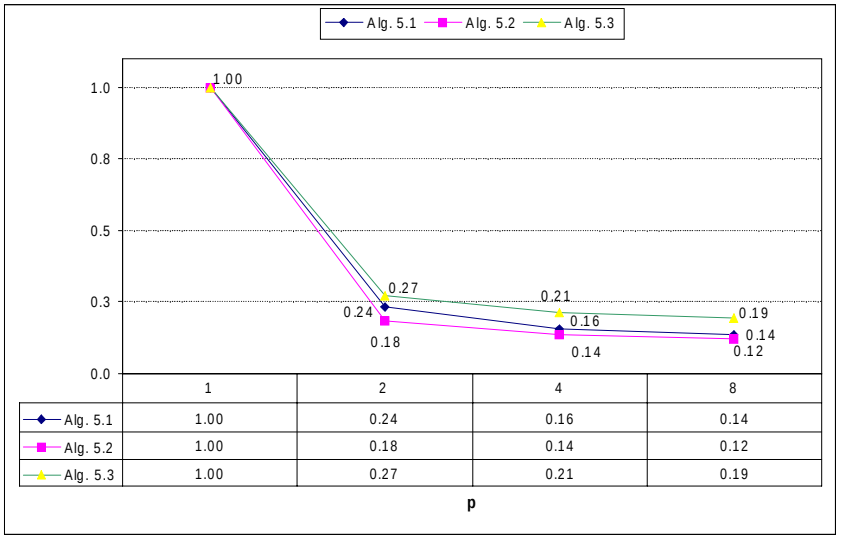
Hay que notar que los valores del *speedup* y la eficiencia que se obtienen para estos algoritmos son sensiblemente peores que los que se obtuvieron para los algoritmos estudiados en los capítulos anteriores. Por ejemplo, en los algoritmos basados en la fórmula de Sherman-Morrison (véase el capítulo 3), se obtuvieron valores del speedup de 9.47. Ahora, sin embargo, el valor máximo del *speedup* es de 5.05, lo que representa casi la mitad del primero. Por otra parte, los valores en el IBM SP2 y en el cluster son muy bajos.

Las figuras 5.3 y 5.4 nos muestran los valores del *speedup* obtenidos en la tabla 5.7.



	1	2	4	8	16	32	64	128	256
Alg. 5.1	1.00	1.38	1.68	1.81	1.83	1.73	1.82	1.75	1.68
Alg. 5.2	1.00	1.33	2.10	2.74	3.08	2.87	3.28	3.07	2.85
Alg. 5.3	1.00	1.40	2.38	3.46	4.28	4.29	5.05	4.82	4.51

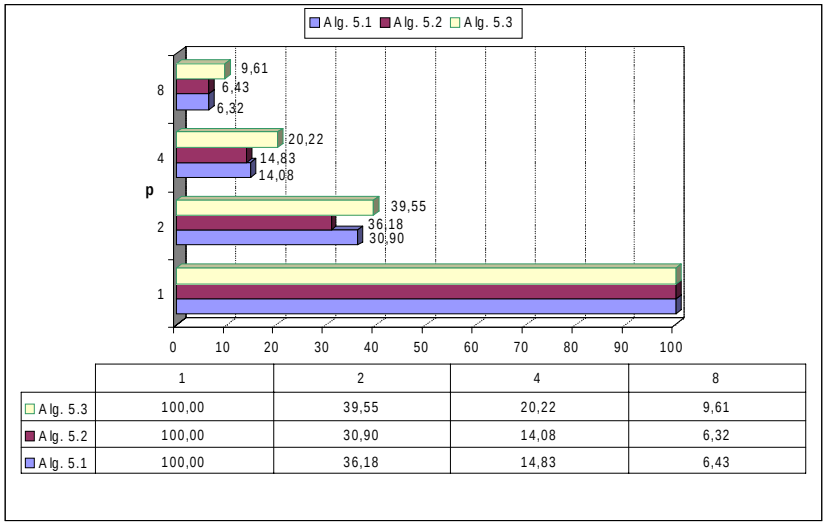
(a) *CRAY T3D*



	1	2	4	8
Alg. 5.1	1.00	0.24	0.16	0.14
Alg. 5.2	1.00	0.18	0.14	0.12
Alg. 5.3	1.00	0.27	0.21	0.19

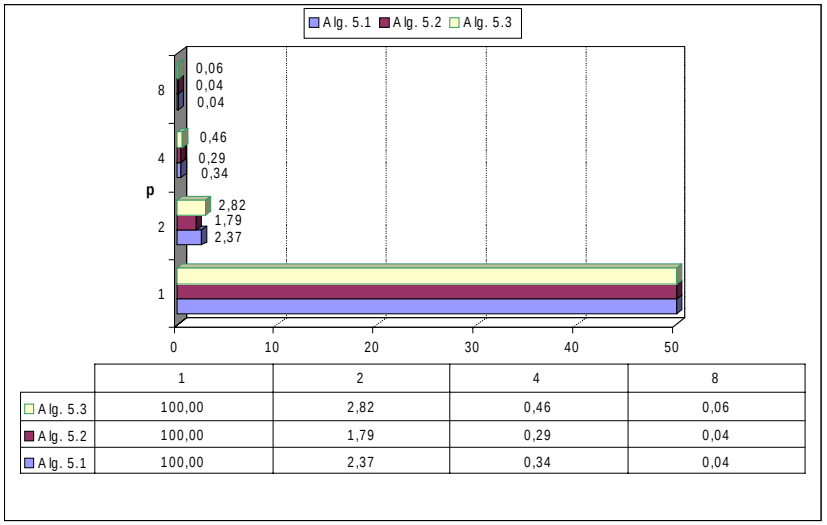
(b) *Cluster de Pentiums*

Figura 5.4: *Valores del speedup en un CRAY T3D y un cluster de Pentiums*



	1	2	4	8
Alg. 5.3	100,00	39,55	20,22	9,61
Alg. 5.2	100,00	30,90	14,08	6,32
Alg. 5.1	100,00	36,18	14,83	6,43

(a) *IBM SP2 switch*

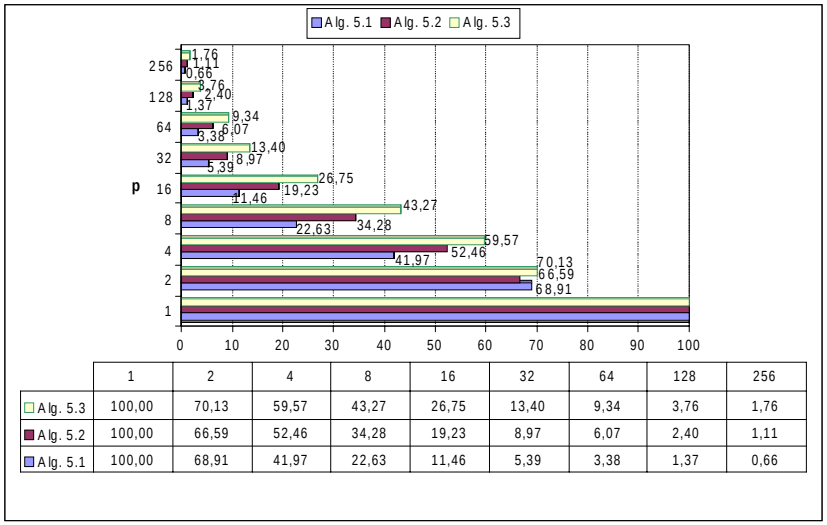


	1	2	4	8
Alg. 5.3	100,00	2,82	0,46	0,06
Alg. 5.2	100,00	1,79	0,29	0,04
Alg. 5.1	100,00	2,37	0,34	0,04

(b) *IBM SP2 ethernet*

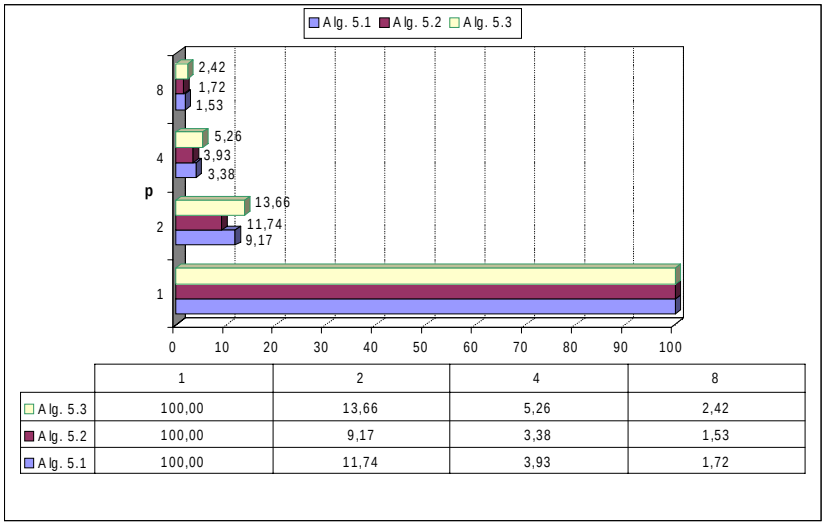
Figura 5.5: *Valores de la eficiencia en un IBM SP2*

Las figuras 5.5 y 5.6 nos muestra los valores obtenidos de la eficiencia para los algoritmos 5.1, 5.2 y 5.3 en las tres máquinas donde se han estudiado los resultados numéricos teóricos de dichos algoritmos.



	1	2	4	8	16	32	64	128	256
Alg. 5.3	100,00	70,13	59,57	43,27	26,75	13,40	9,34	3,76	1,76
Alg. 5.2	100,00	66,59	52,46	34,28	19,23	8,97	6,07	2,40	1,11
Alg. 5.1	100,00	68,91	41,97	22,63	11,46	5,39	3,38	1,37	0,66

(a) *CRAY T3D*



	1	2	4	8
Alg. 5.3	100,00	13,66	5,26	2,42
Alg. 5.2	100,00	9,17	3,38	1,53
Alg. 5.1	100,00	11,74	3,93	1,72

(b) *Cluster de Pentiums*

Figura 5.6: *Valores de la eficiencia en un CRAY T3D y un cluster de Pentiums*

Capítulo 6 Estudio comparativo de los algoritmos

En los capítulos anteriores hemos analizado y estudiado en profundidad diversos algoritmos del tipo *divide y vencerás* que podemos resumir en tres tipos: algoritmos basados en la aplicación de la fórmula de Sherman-Morrison, algoritmos basados en la aplicación de la fórmula de Sherman-Morrison-Woodbury y algoritmos basados en el método *divide y vencerás* de Bondeli para sistemas tridiagonales. A lo largo de cada capítulo hemos obtenido tiempos y realizado comparaciones entre algoritmos del mismo tipo; además, también hemos estudidado el comportamiento paralelo de cada uno de los algoritmos a través de su *speedup*. Sin embargo, no hemos realizado comparaciones entre todos los algoritmos a nivel general ni hemos analizado qué algoritmo resulta óptimo desde el punto de vista de tiempo de ejecución en cada una de las máquinas. Este es nuestro objetivo en este capítulo y para ello analizamos los tiempos obtenidos para todos los algoritmos en las distintas máquinas.

6.1 Algoritmos *divide y vencerás* en el IBM SP2

En esta sección realizamos un estudio comparativo de los tiempos de ejecución de los diferentes algoritmos estudiados en el transcurso de esta memoria en una máquina IBM SP2 como la que hemos venido utilizando para obtener resultados teóricos en los capítulos anteriores. Ahora nuestro objetivo será determinar de entre todos los algoritmos estudiados el óptimo en esta máquina. Analizamos de

forma separada los resultados para conexión switch y ethernet.

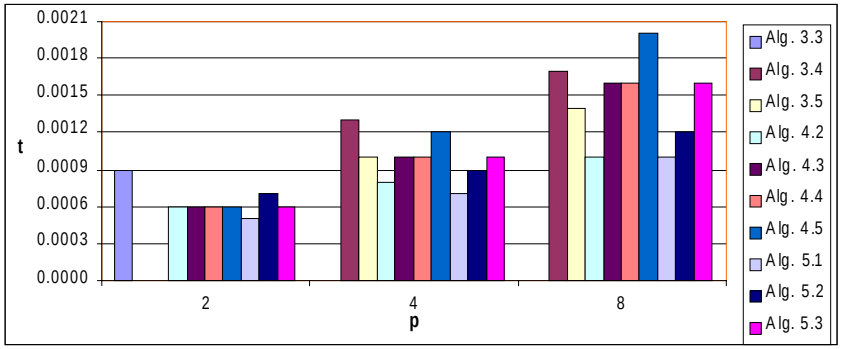
Las figuras 6.1, 6.2, 6.3 y 6.4 nos muestran los resultados numéricos teóricos de todos los algoritmos estudiados en una máquina IBM SP2 utilizando el switch de alto rendimiento. Se observa que para cualquier tamaño de matriz y para cualquier número de procesadores, el algoritmo más rápido de tosos es el algoritmo 4.2. También observamos claramente en las gráficas que las diferencias de tiempos entre los cuatro algoritmos basados en la fórmula de Sherman-Morrison-Woodbury es mínima, obteniéndose los mismos tiempos para algunos tamaños como por ejemplo para $n = 2048$, $n = 16384$ y otros. Otro aspecto que resulta destacable es que el algoritmo 5.3 tiene unos tiempos de ejecución prácticamente idénticos a los tiempos del algoritmo más rápido, siendo las diferencias del orden de 10^{-5}. Recordemos que el algoritmo 5.3 estaba basado en el método de Bondeli y resolvía el sistema tridiagonal auxiliar en paralelo utilizando el método del *recursive doubling*.

Los algoritmos que ofrecen peores tiempos son los que se basan en la fórmula de Sherman-Morrison, utilizando la técnica del desacoplamiento recursivo. Las diferencias de tiempo entre estos y el resto es muy significativa. Así, por ejemplo, para $n = 524288$, tenemos que para $p = 4$ el algoritmo 4.2 tarda 0.1668 segundos, mientras que el algoritmo 3.4 tarda 0.4631, lo que representa un coste de casi el triple. Si observamos la gráficas 6.4(a) y 6.4(b) que corresponden a tamaños de matriz grandes, notamos que para dos procesadores se producen las diferencias más importantes, ya que el algoritmo 3.3 es más de cuatro veces más lento que los algoritmos más rápidos.

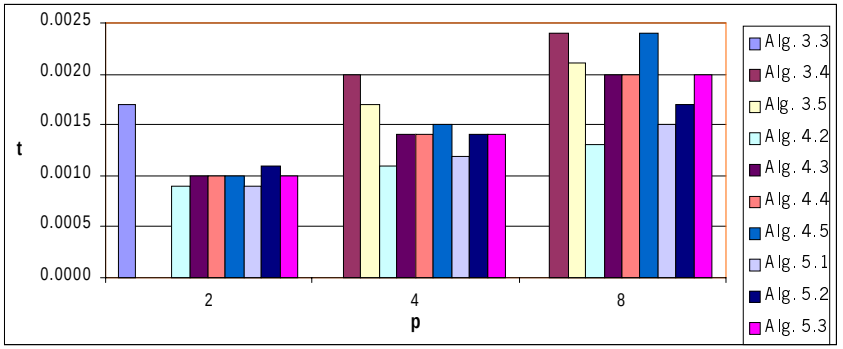
En las figuras 6.2, 6.3 y 6.4 también se observa la disminución en los tiempos que se produce al ejecutar el algoritmo 3.4 cuando el número de procesadores aumenta de 4 a 8. El algoritmo 3.5 únicamente disminuye sus tiempos para tamaños de matriz superiores a $n = 262144$.

En resumen, en una máquina de este tipo con switch, el algoritmo más rápido es el 4.2, aunque no existen diferencias significativas entre este y el resto de algoritmos basados en la fórmula de Sherman-Morrison-Woodbury y el algoritmo 5.3.

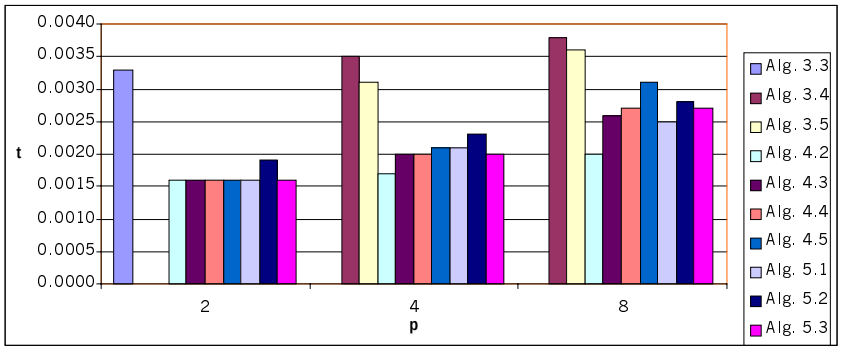
Las figuras 6.5, 6.6, 6.7 y 6.8 nos muestran los resultados numéricos teóricos de todos los algoritmos estudiados en una máquina IBM SP2 utilizando una conexión ethernet. Las mismas conclusiones que obtuvimos anteriormente respecto al algoritmo más rápido



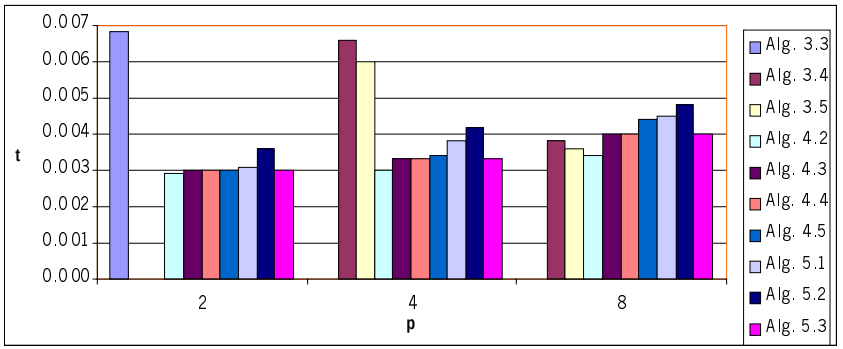
(a) $n = 512$



(b) $n = 1024$

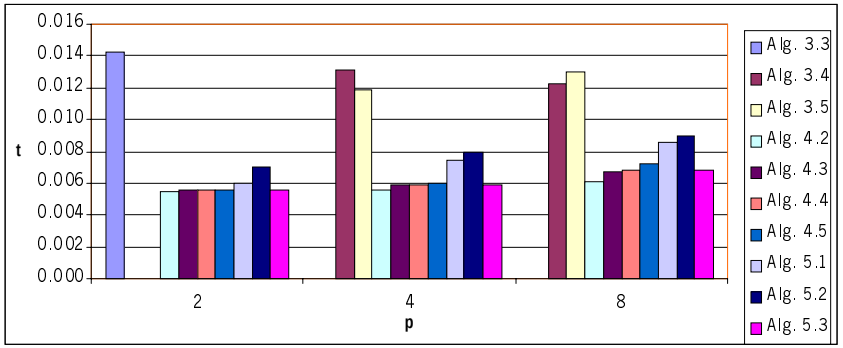


(c) $n = 2048$

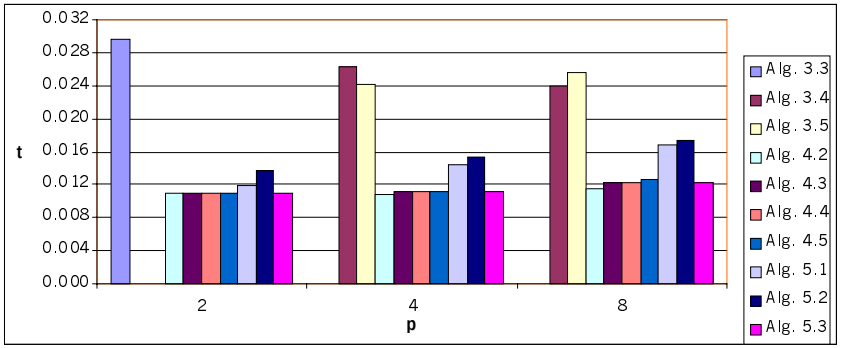


(d) $n = 4096$

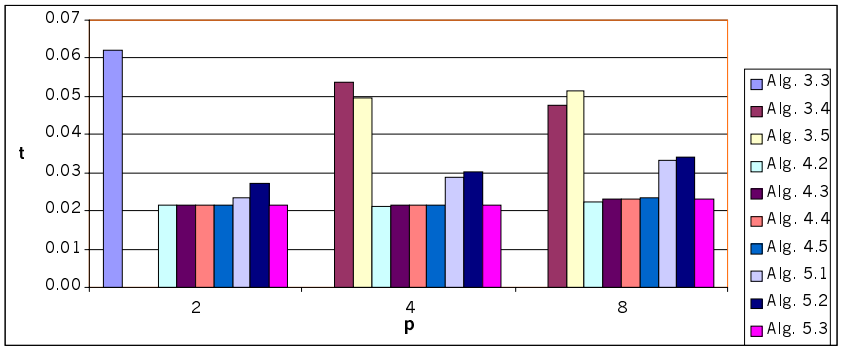
Figura 6.1: *Tiempos en un IBM SP2 con switch para* $n = 512$, $n = 1024$, $n = 2048$ *y* $n = 4096$



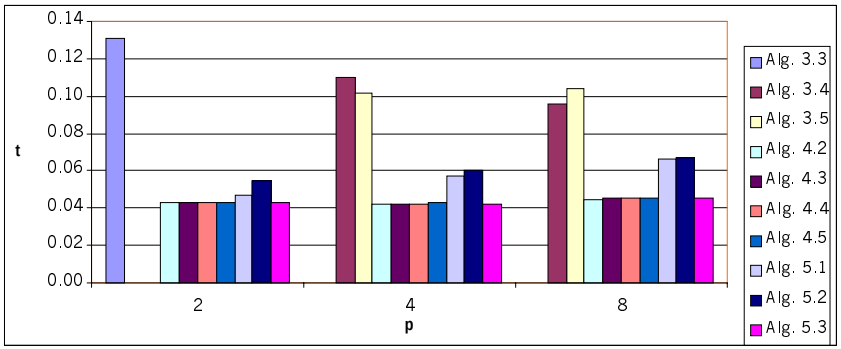
(a) $n = 8192$



(b) $n = 16384$

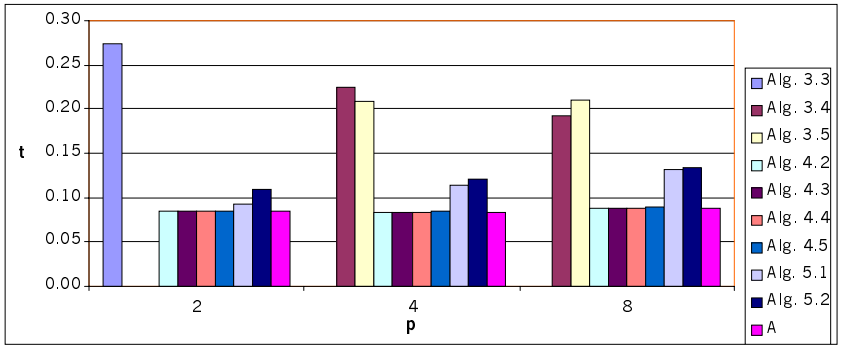


(c) $n = 32768$

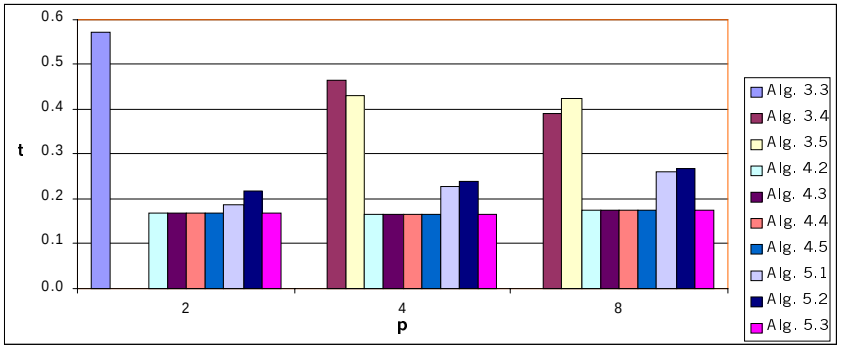


(d) $n = 65536$

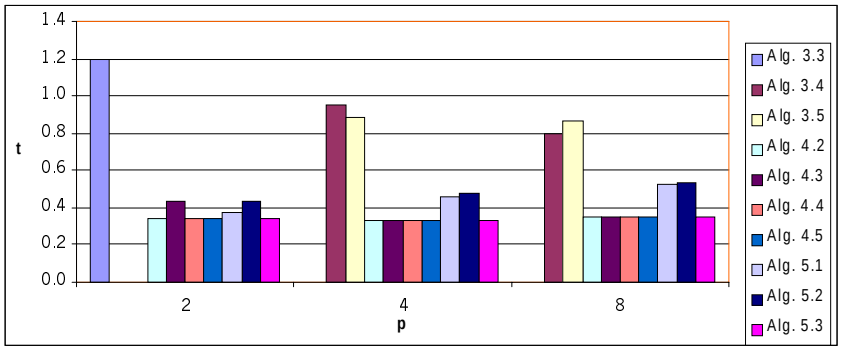
Figura 6.2: *Tiempos en un IBM SP2 con switch para* $n = 8192$, $n = 16384$, $n = 32768$ *y* $n = 65536$



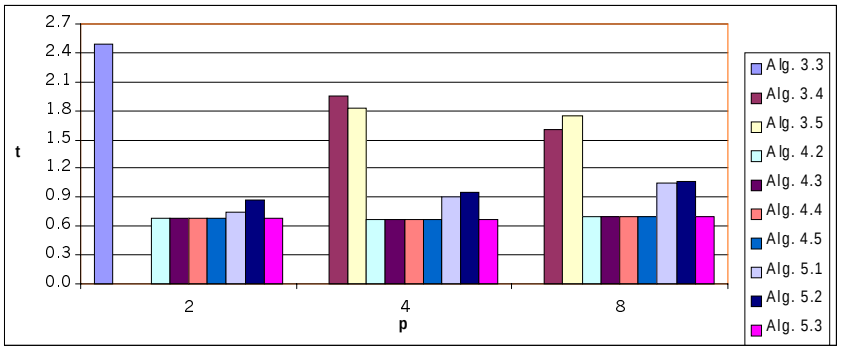
(a) $n = 131072$



(b) $n = 262144$

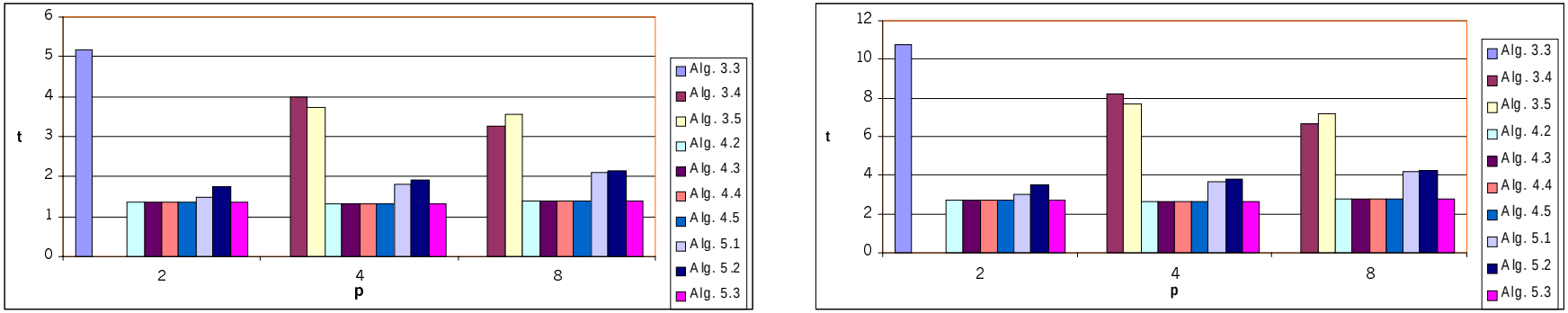


(c) $n = 524288$



(d) $n = 1048576$

Figura 6.3: *Tiempos en un IBM SP2 con switch para* $n = 8192$, $n = 16384$, $n = 32768$ *y* $n = 65536$



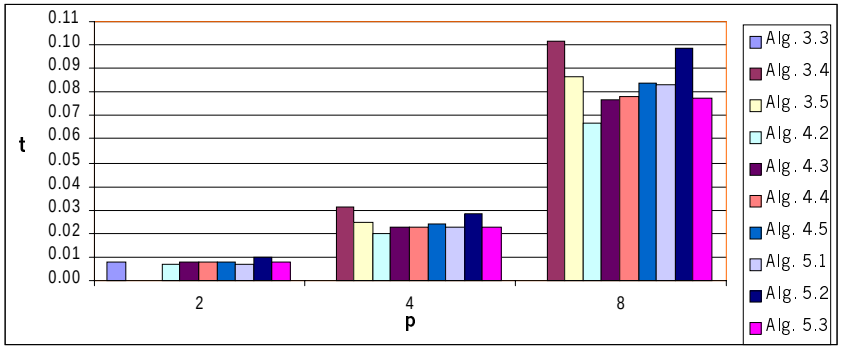
(a) $n = 2097152$ **(b)** $n = 4194304$

Figura 6.4: *Tiempos en un IBM SP2 con switch para* $n = 2097152$ *y* $n = 4194304$

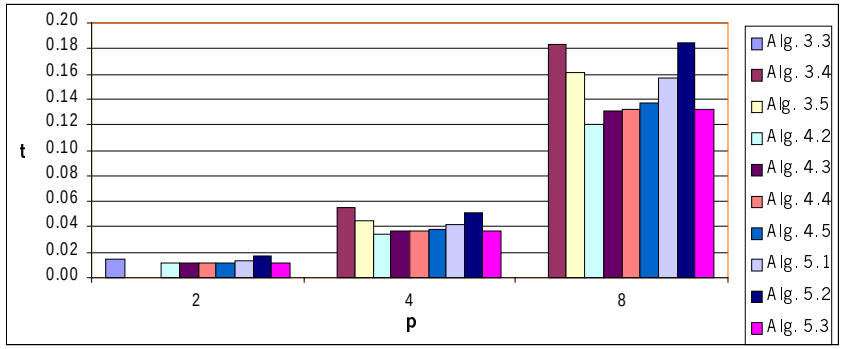
son válidas ahora para una conexión de tipo ethernet. El algoritmo 4.2 vuelve a ser el más rápido para cualquier tamaño de matriz y para cualquier número de procesadores. Asímismo, como se aprecia en las distintas gráficas, no existen diferencias significativas entre el mejor algoritmo, el resto de algoritmos basados en la fórmula de Sherman-Morrison-Woodbury y el algoritmo 5.3, que tiene unos tiempos prácticamente idénticos al mejor algoritmo.

Se observa en todas las gráficas las diferencias tan significativas de tiempo que se producen al ir aumentando el número de procesadores. Sin duda, los elevados valores de los parámetros de comunicación y sincronización de esta máquina con esta conexión producen un aumento considerable en los tiempos para todos los algoritmos, independientemente del tamaño de la matriz. Así, aumentar de $p = 2$ a $p = 4$ o de $p = 4$ a $p = 8$ supone que los tiempos se multiplican por un factor de tres, lo que nos da una idea del mal comportamiento paralelo de todos los algoritmos en esta máquina con este tipo de conexión. Esto queda reflejado en todas las gráficas.

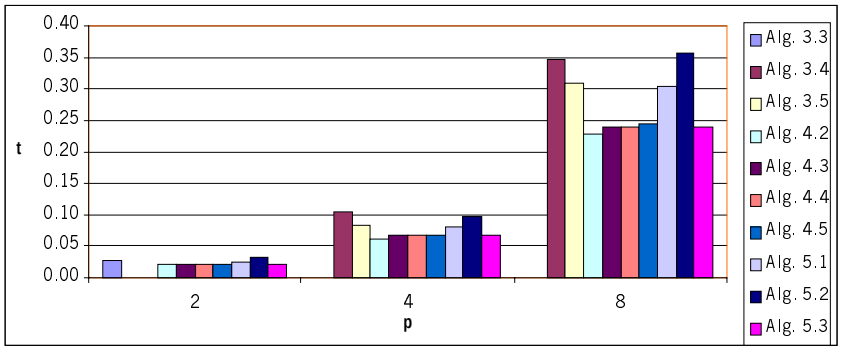
La diferencia más significativa respecto al caso con switch se encuentra en el algoritmo más lento. Si anteriormente el algoritmo más



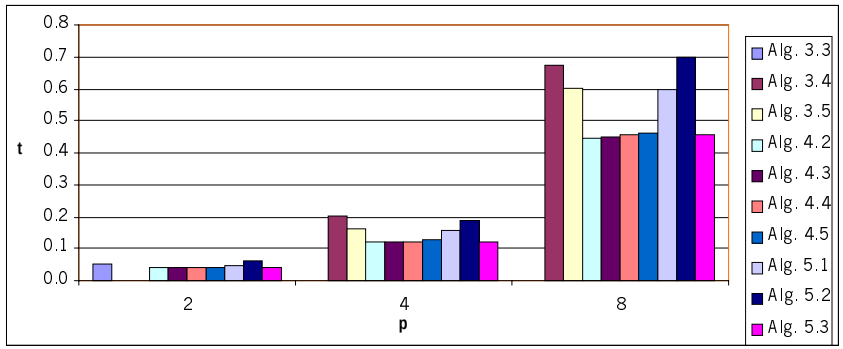
(a) $n = 512$



(b) $n = 1024$

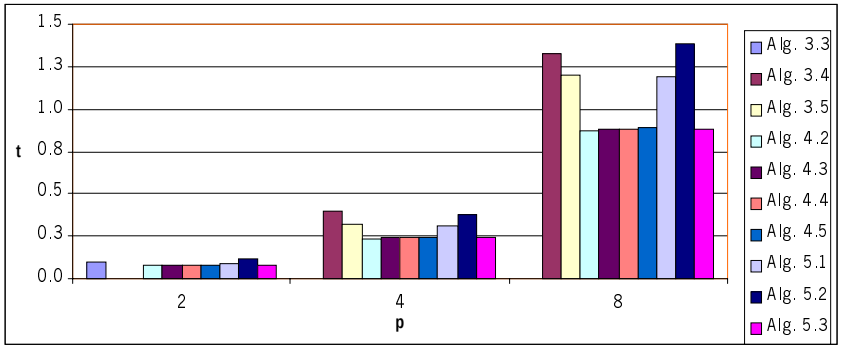


(c) $n = 2048$

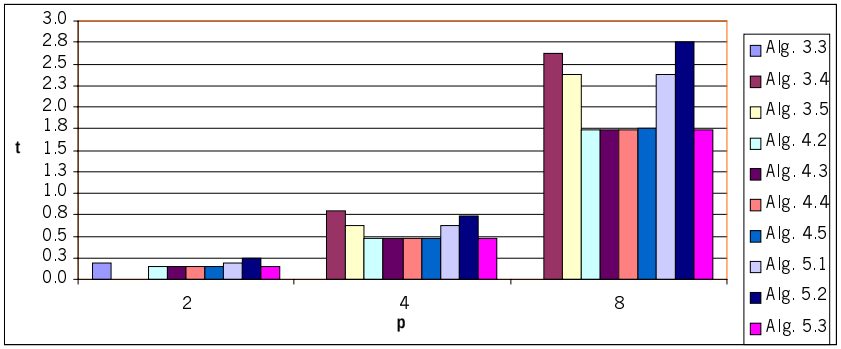


(d) $n = 4096$

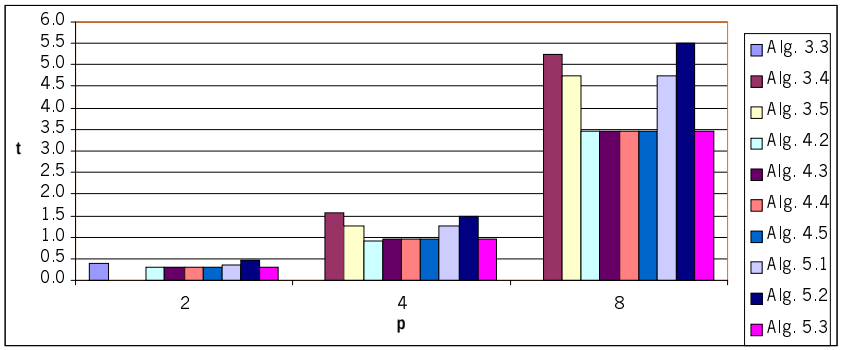
Figura 6.5: *Tiempos en un IBM SP2 con ethernet para* $n = 512$, $n = 1024$, $n = 2048$ *y* $n = 4096$



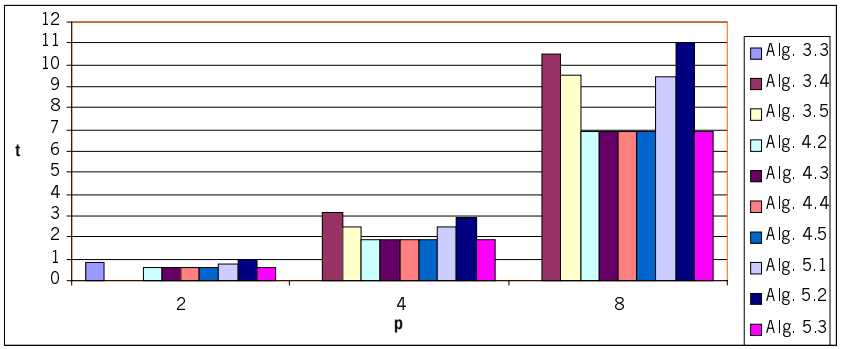
(a) $n = 8192$



(b) $n = 16384$

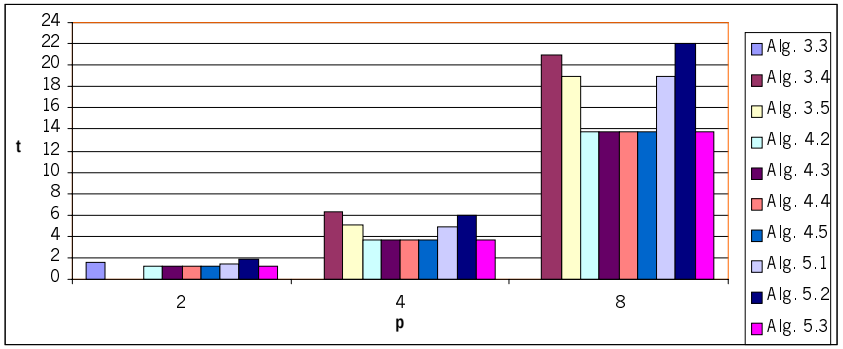


(c) $n = 32768$

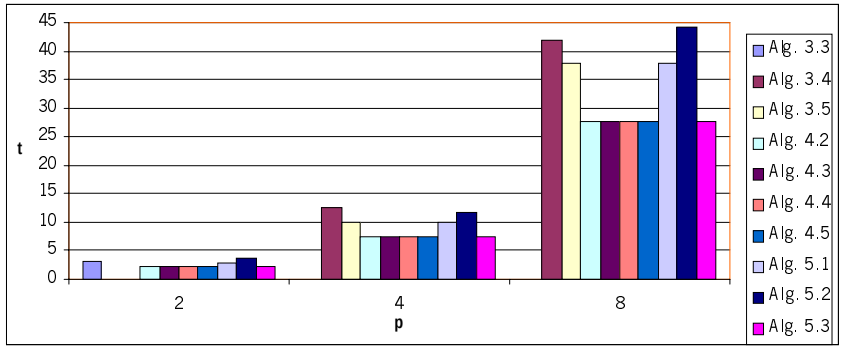


(d) $n = 65536$

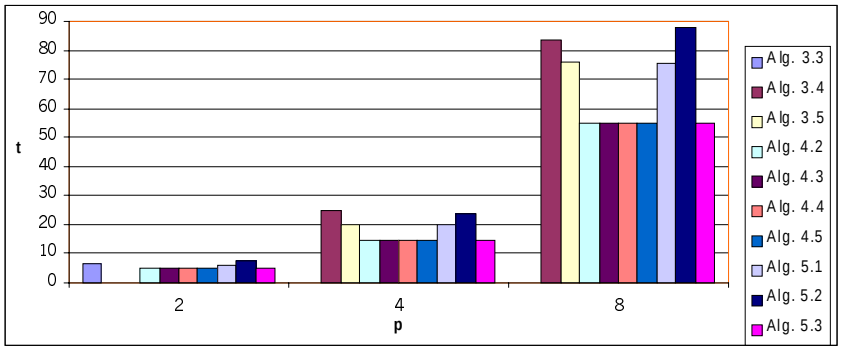
Figura 6.6: *Tiempos en un IBM SP2 con ethernet para* $n = 8192$, $n = 16384$, $n = 32768$ *y* $n = 65536$



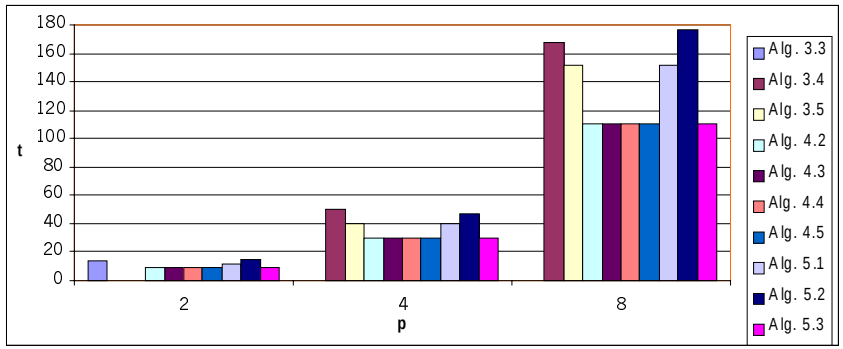
(a) $n = 131072$



(b) $n = 262144$

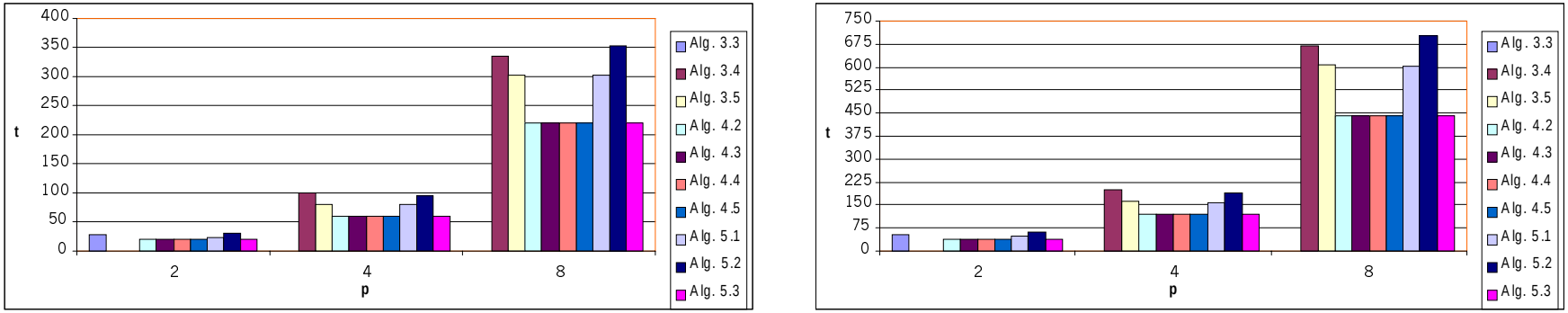


(c) $n = 524288$



(d) $n = 1048576$

Figura 6.7: *Tiempos en un IBM SP2 con ethernet para* $n = 131072$, $n = 262144$, $n = 524288$ *y* $n = 1048576$



(a) $n = 2097152$ **(b)** $n = 4194304$

Figura 6.8: *Tiempos en un IBM SP2 con ethernet para* $n = 2097152$ *y* $n = 4194304$

lento era el 3.4 en todos los casos y las diferencias con el resto de algoritmos eran muy grandes, ahora no podemos decir lo mismo. Para $p = 2$ el algoritmo más lento es el 5.2, salvo para tamaños de $n = 2097152$ y $n = 4194304$, en los que el algoritmo 3.4 es ligeramente más lento. Para $p = 4$, el algoritmo 3.4 siempre es algo más lento que el algoritmo 5.2. Sin embargo, cuando $p = 4$ esta tendencia se invierte y el algoritmo 5.2 ya es más lento que el otro. Lo más destacable es que no se producen esas diferencias tan enormes que se producían en el caso anterior entre los algoritmos más rápidos y más lentos.

En resumen, los algoritmos basados en la fórmula de Sherman-Morrison-Woodbury son los que ofrecen mejores tiempos junto con el algoritmo 5.3.

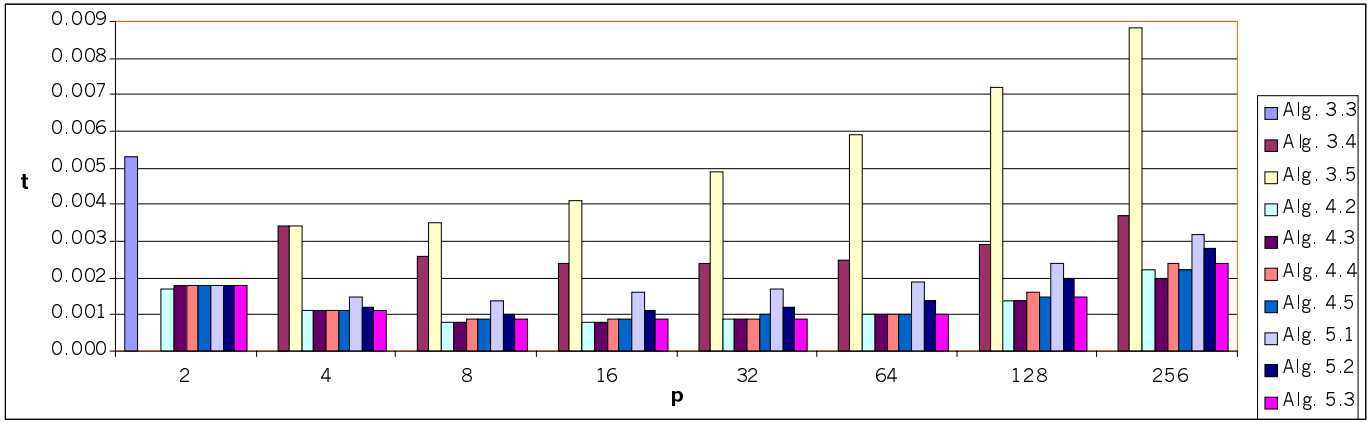
6.2 Algoritmos *divide y vencerás* en el CRAY T3D

En esta sección realizamos un estudio comparativo de los tiempos de ejecución de los diferentes algoritmos estudiados en el transcurso de esta memoria en una máquina CRAY T3D como la que hemos venido utilizando para obtener resultados teóricos en los capítulos anteriores. Nuestro objetivo vuelve a ser encontrar el algoritmo óptimo en esta máquina.

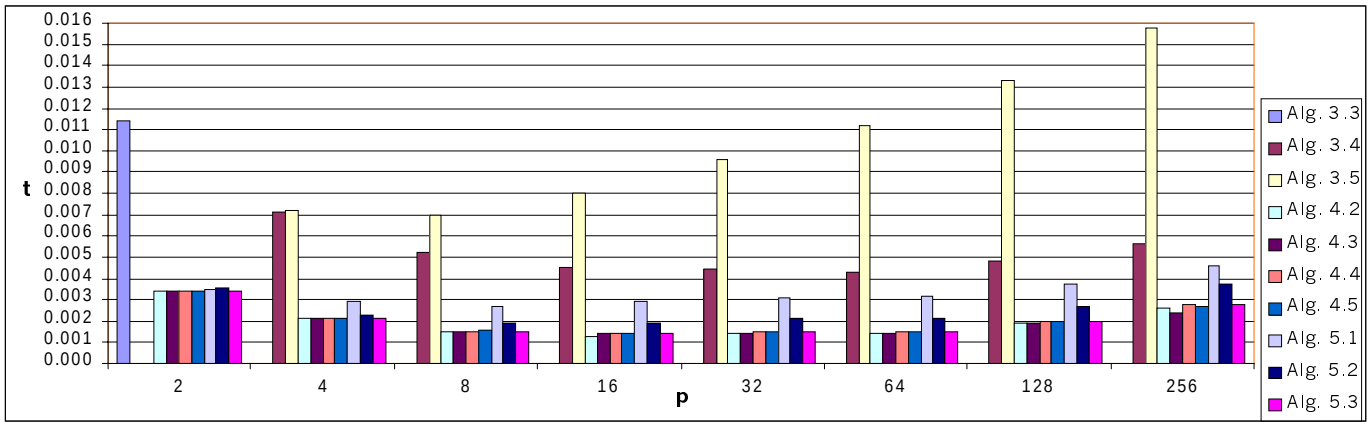
Las figuras 6.9, 6.10, 6.11, 6.12, 6.13 y 6.14 nos muestran los resultados numéricos teóricos de todos los algoritmos estudiados en una máquina CRAY T3D. Las conclusiones de carácter general sobre el algoritmo más rápido en esta máquina no difieren de las obtenidas en los casos anteriores. Los algoritmos basados en la fórmula de Sherman-Morrison-Woodbury y el algoritmo 5.3 son los más rápidos, no existiendo diferencias significativas entre ellos.

Las figuras anteriormente citadas nos muestran diversos aspectos en la ejecución de estos algoritmos en esta máquina que resultan interesantes. Por ejemplo, uno de los más destacados es la enorme diferencia de tiempos que existe entre el algoritmo 3.5 y el resto de algoritmos. Dicho algoritmo es con diferencia el más lento para todos los tamaños de la matriz de coeficientes y cualquier número de procesadores. Además, las diferencias aumentan a medida que aumenta el tamaño de la matriz. Así, para $n = 8192$ y $p = 64$ tenemos que el algoritmo 3.5 tarda 0.0218 segundos mientras que el algoritmo 4.2 tarda únicamente $0,0024$, lo que representa un tiempo nueve veces menor. Si tomamos $n = 4194304$ y $p = 64$, el tiempo que supone la ejecución del algoritmo 4.2 es casi 12 veces menor que el del algoritmo 3.5. Notemos también el incremento que experimenta el algoritmo 3.3, válido para $p = 2$, cuando aumenta el tamaño de la matriz. Para tamaños de matriz superiores o iguales a 32768 dicho algoritmo proporciona el tiempo más lento considerando todos los algoritmos y cualquier número de procesadores, como se aprecia en la gráfica 6.11(a) y posteriores.

Resulta destacable la enorme diferencia en el comportamiento de los algoritmos 3.4 y 3.5 en esta máquina. Si en el IBM SP2 se había observado que las diferencias de tiempos entre estos algoritmos no era demasiado grande y en la mayoría de los casos siempre era favorable al algoritmo 3.5, ahora sucede todo lo contrario. En primer lugar, las diferencias entre ambos algoritmos son únicamente muy pequeñas para $p = 4$. A medida que aumenta el número de procesadores, estas diferencias van creciendo hasta llegar en algunos casos

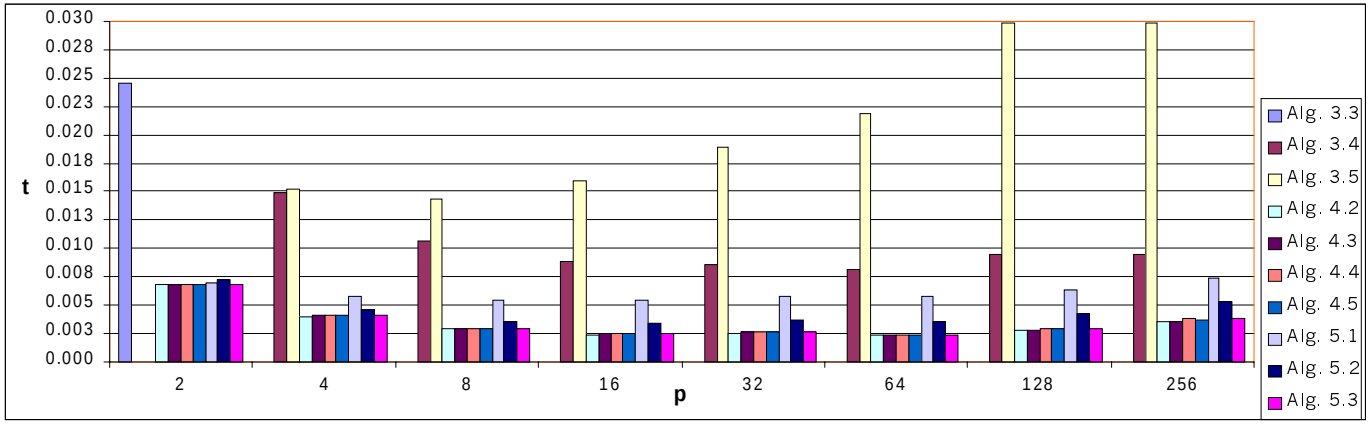


(a) $n = 2048$

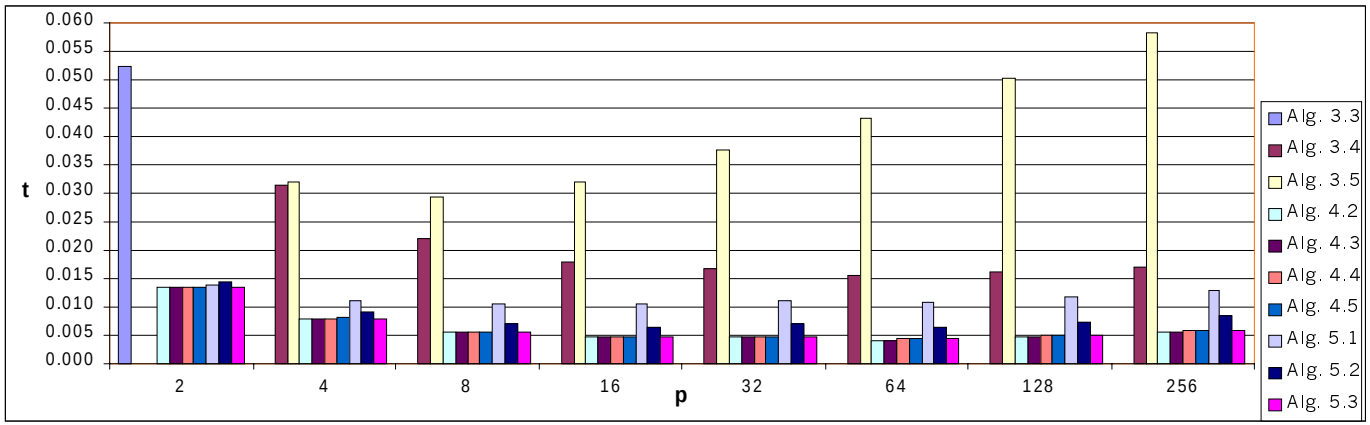


(b) $n = 4096$

Figura 6.9: *Tiempos en un CRAY T3D para* $n = 2048$ *y* $n = 4096$.

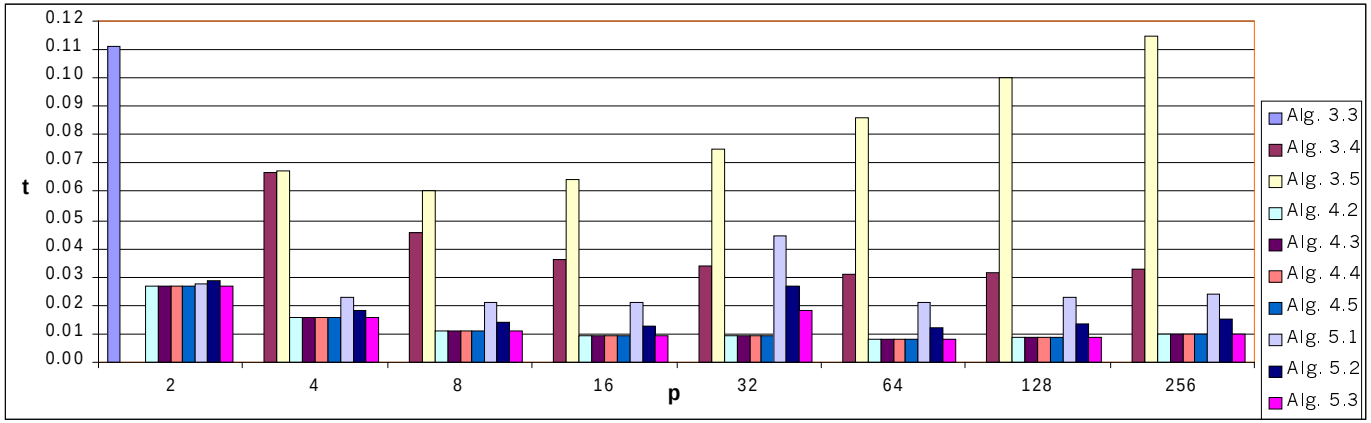


(a) $n = 8192$

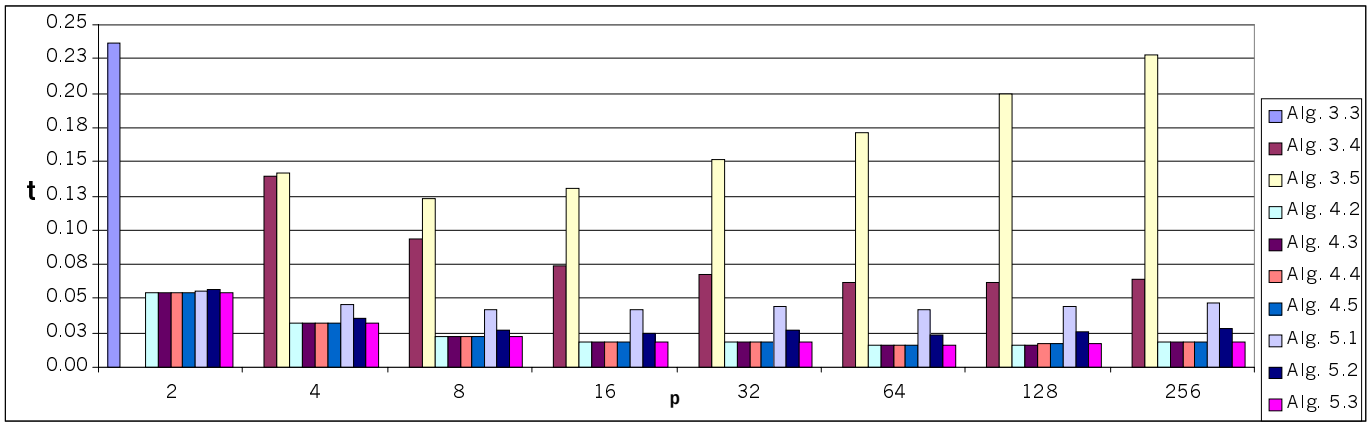


(b) $n = 16384$

Figura 6.10: *Tiempos en un CRAY T3D para* $n = 8192$ *y* $n = 16384$.

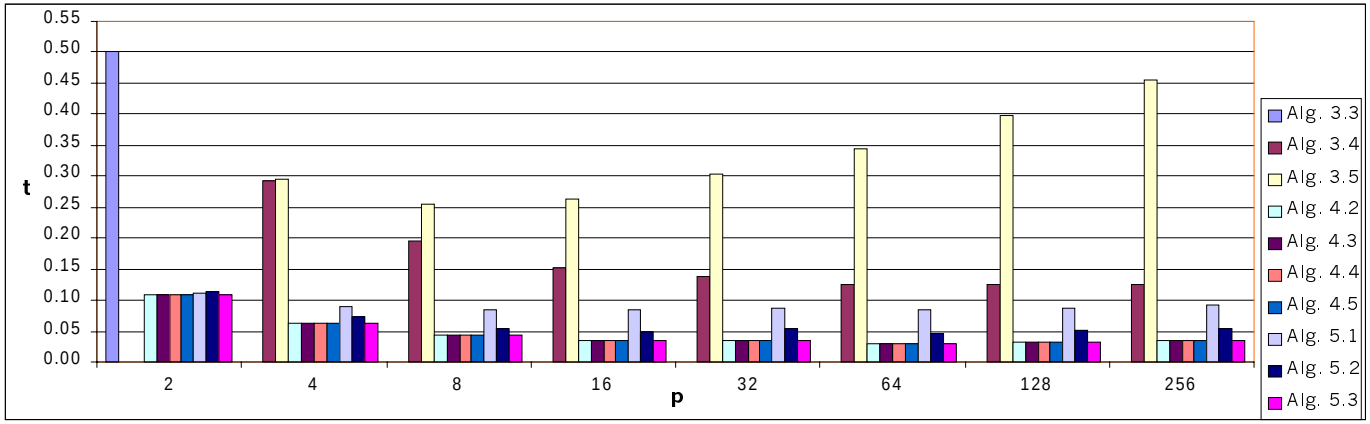


(a) $n = 32768$

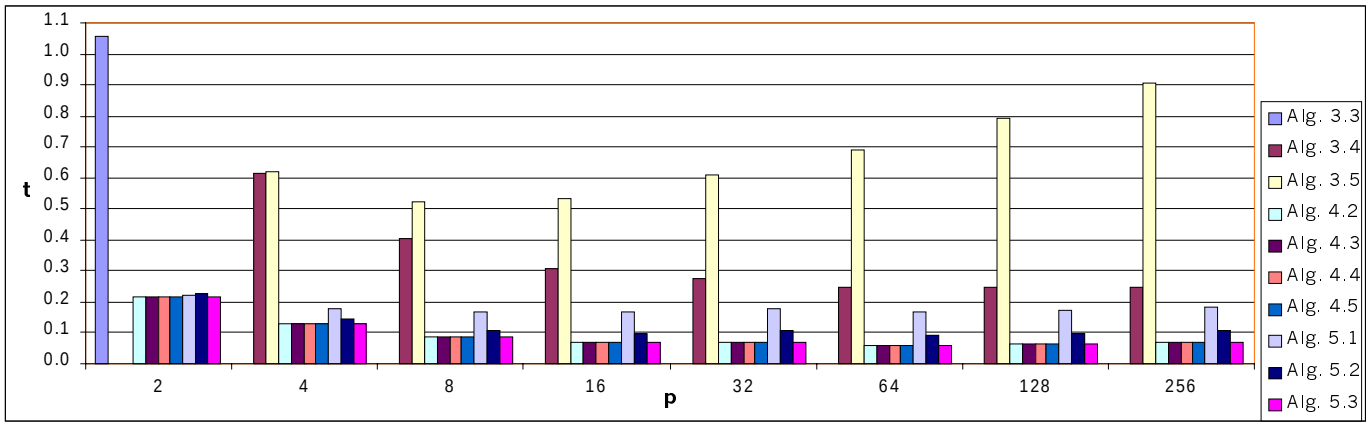


(b) $n = 65536$

Figura 6.11: *Tiempos en un CRAY T3D para* $n = 32768$ *y* $n = 65536$.

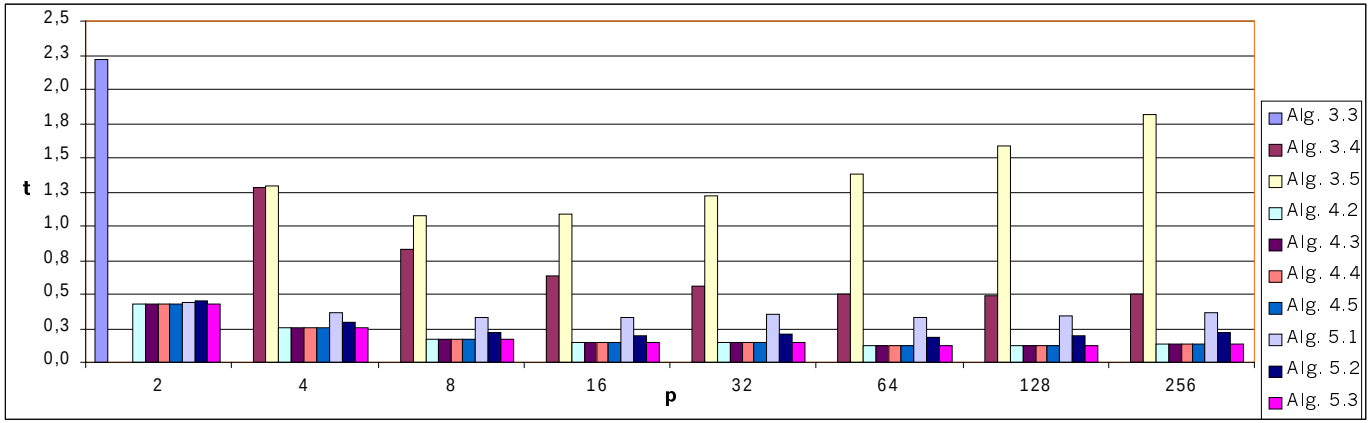


(a) $n = 131072$

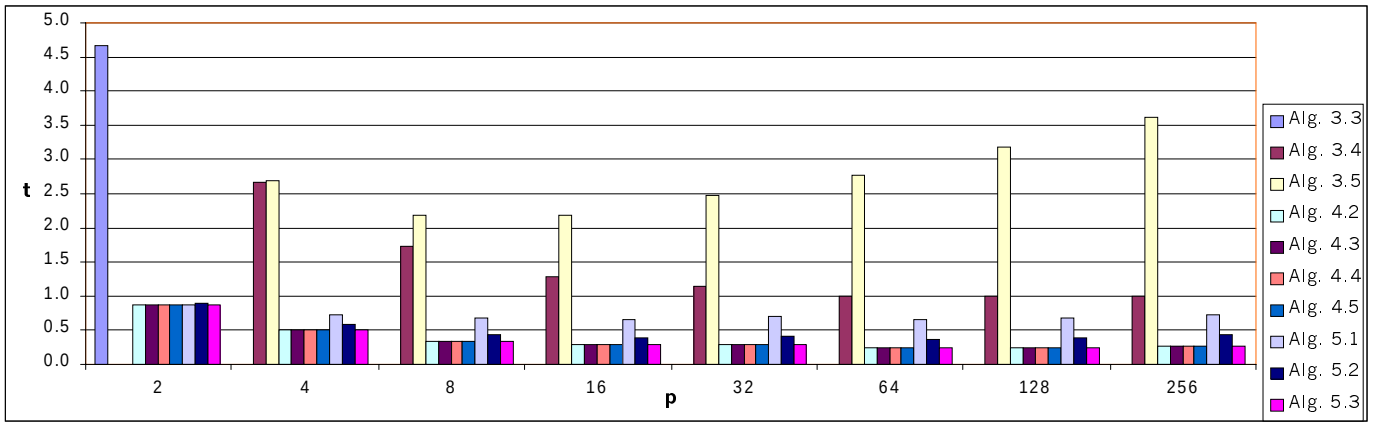


(b) $n = 262144$

Figura 6.12: *Tiempos en un CRAY T3D para* $n = 131072$ *y* $n = 262144$.

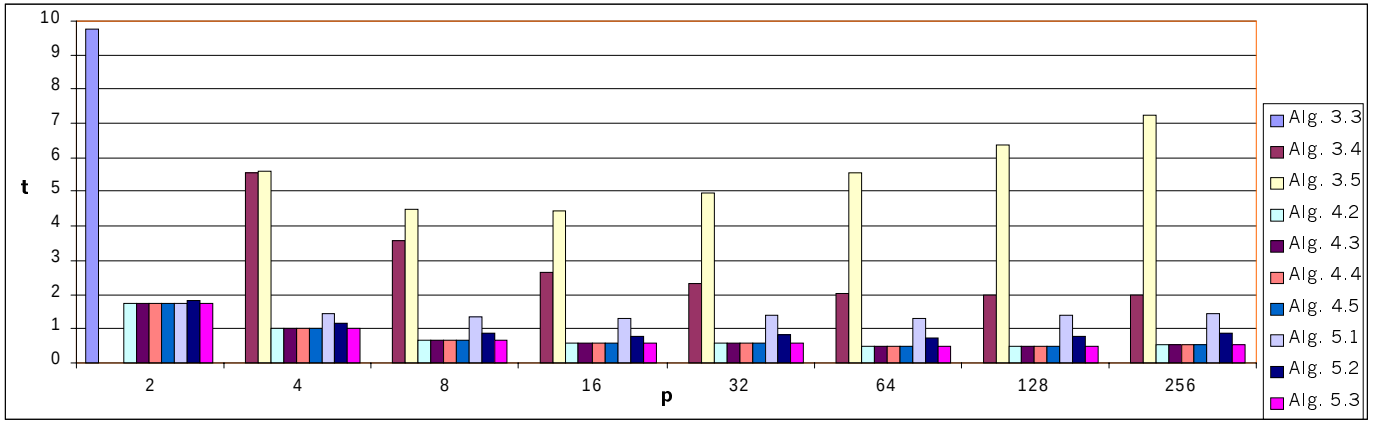


(a) $n = 524288$

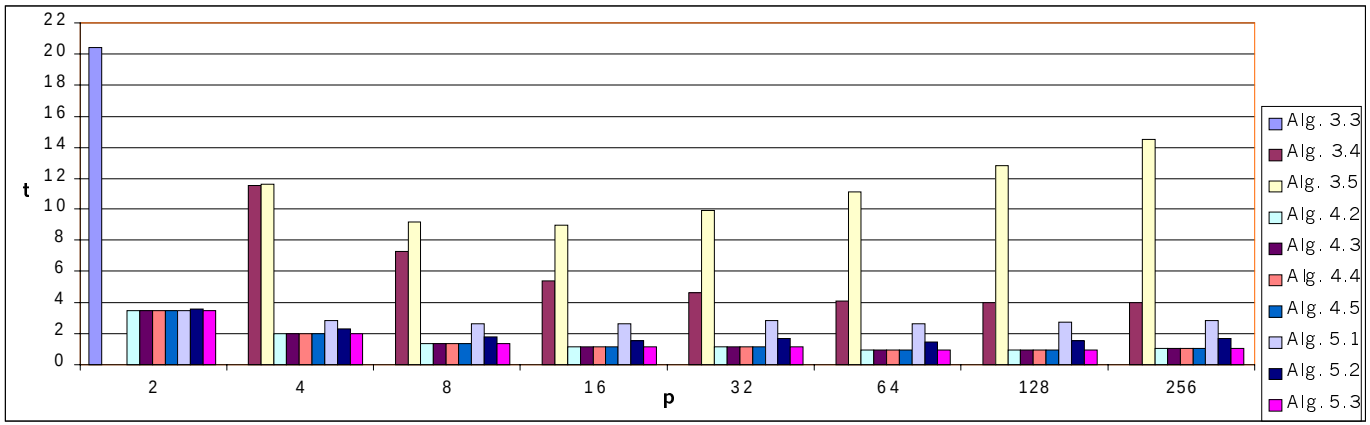


(b) $n = 1048576$

Figura 6.13: *Tiempos en un CRAY T3D para* $n = 524288$ *y* $n = 1048576$.



(a) $n = 2097152$



(b) $n = 4194304$

Figura 6.14: *Tiempos en un CRAY T3D para* $n = 2097152$ *y* 4194304.

a ser del triple. Por otra parte, se observa que el algoritmo 3.4 posee un comportamiento paralelo similar al del resto de algoritmos, disminuyendo los tiempos al aumentar el número de procesadores hasta 64. Este hecho no se repite para el algoritmo 3.5, que aumenta continuamente sus tiempos al aumentar el número de procesadores.

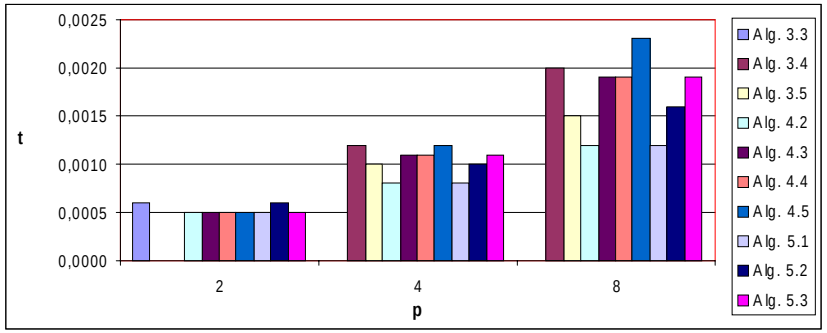
Nos detenemos un poco más en el aspecto del comportamiento paralelo de estos algoritmos. Uno de los aspectos más notables del comportamiento de estos algoritmos en una máquina de este tipo es que, a diferencia de lo que ocurría en los casos anteriores con el IBM SP2, se observa una disminución de los tiempos al ir aumentando el número de procesadores. Esta disminución no se aprecia claramente en las gráficas debido a las diferencias de tiempos entre los algoritmos más rápidos y el más lento. Sin embargo, sí se advierte una clara disminución de tiempos al aumentar de $p = 2$ a $p = 4$, de $p = 4$ a $p = 8$ y de $p = 8$ a $p = 16$. El número de procesadores para el que se obtienen los mejores tiempos es 64, aunque las diferencias de tiempos para 32, 64 y 128 procesadores es mínima.

6.3 Algoritmos *divide y vencerás* en un cluster de Pentiums

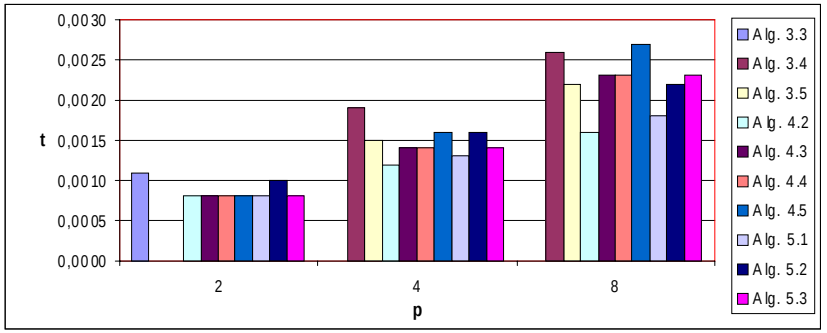
Las figuras 6.15, 6.16, 6.17 y 6.18 nos muestran los resultados numéricos teóricos de todos los algoritmos estudiados en un cluster de Pentiums.

Las conclusiones de carácter general que se obtienen en esta máquina no difieren sustancialmente de las obtenidas en las máquinas anteriores respecto al algoritmo más rápido. Los algoritmos basados en la fórmula de Sherman-Morrison-Woodbury y el algoritmo 5.3 nos proporcionan los mejores resultados, especialmente el algoritmo 4.2 que es ligeramente más rápido que el resto para ciertos valores de n y p. En las gráficas 6.15, 6.16(a), 6.16(b) y 6.16(c) se observa esta pequeña diferencia de tiempos para valores de $n \leq 32768$. A medida que n toma valores cada vez mayores, los tiempos de estos algoritmos se van igualando paulatinamente hasta que en muchos casos son idénticos.

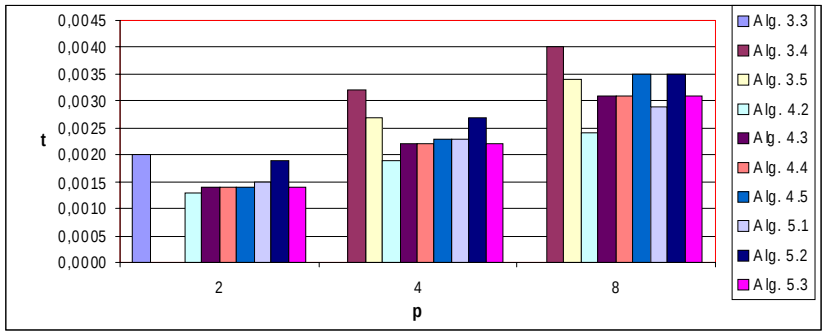
Comparando las gráficas que se obtienen en esta máquina con las obtenidas en el resto de máquinas notamos claramente que el



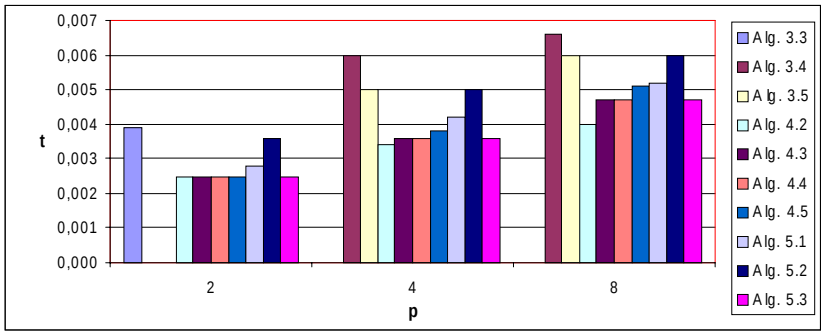
(a) $n = 512$



(b) $n = 1024$

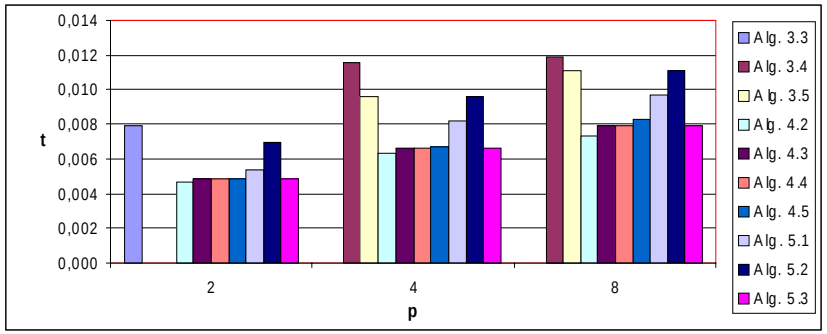


(c) $n = 2048$

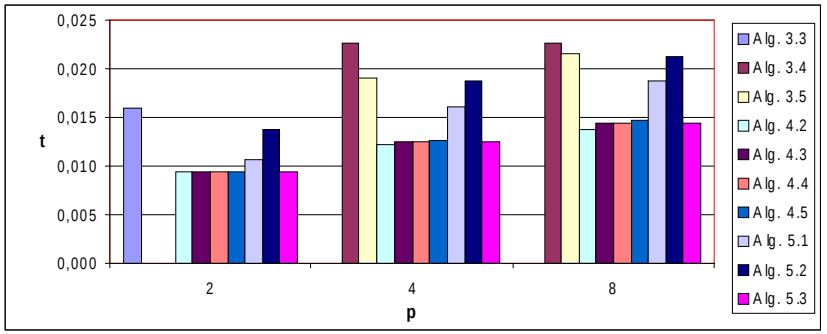


(d) $n = 4096$

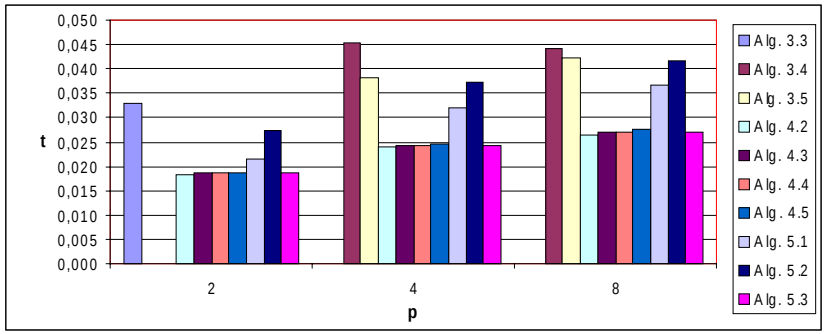
Figura 6.15: *Tiempos en un cluster de Pentiums para* $n = 512$, $n = 1024$, $n = 2048$ *y* $n = 4096$



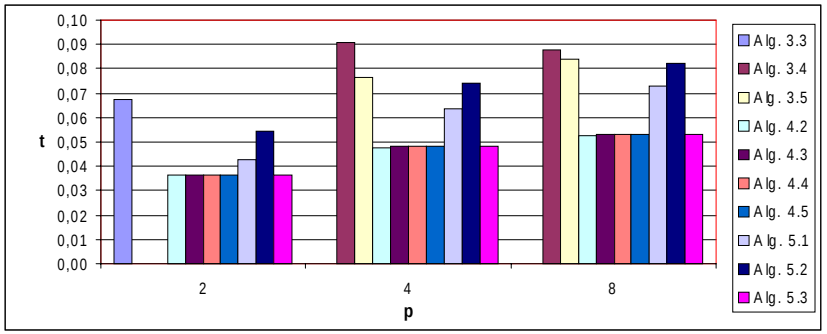
(a) $n = 8192$



(b) $n = 16384$

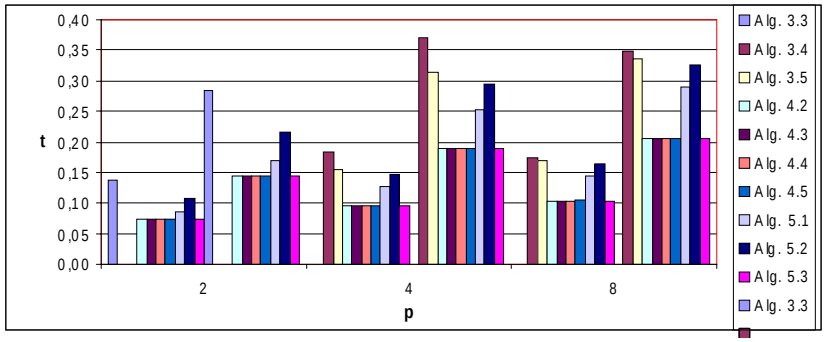


(c) $n = 32768$

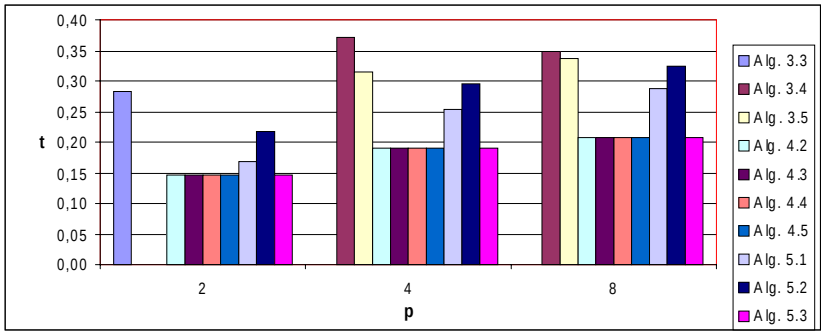


(d) $n = 65536$

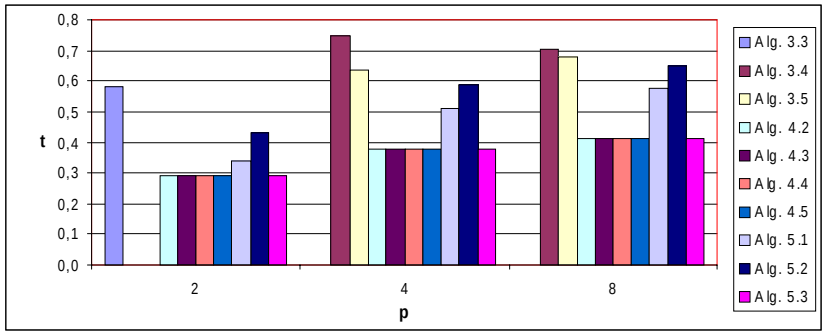
Figura 6.16: *Tiempos en un cluster de Pentiums para* $n = 8192$, $n = 16384$, $n = 32768$ *y* $n = 65536$



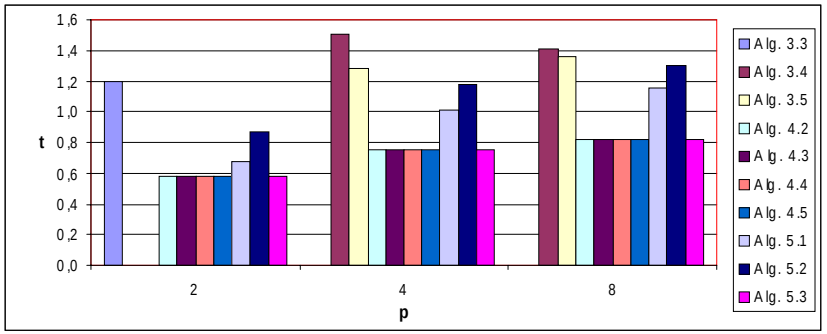
(a) $n = 131072$



(b) $n = 262144$

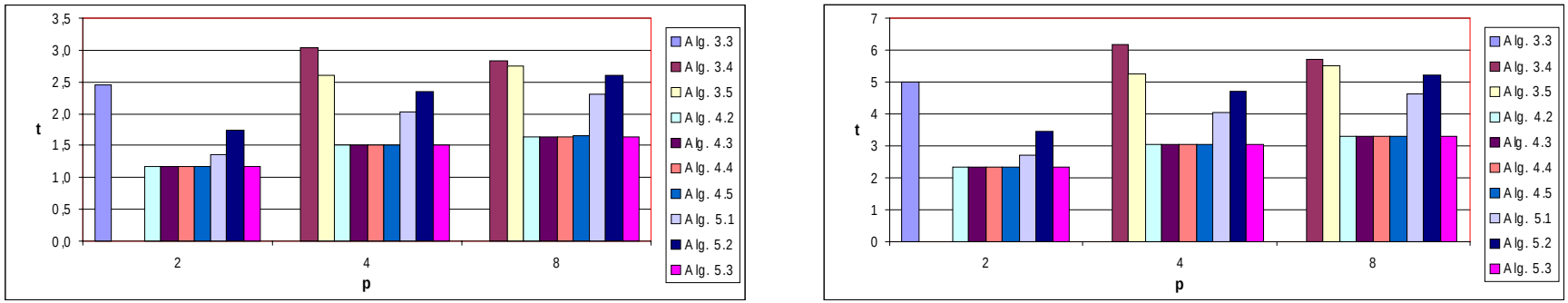


(c) $n = 524288$



(d) $n = 1048576$

Figura 6.17: *Tiempos en un cluster de Pentiums para* $n = 8192$, $n = 16384$, $n = 32768$ *y* $n = 65536$



(a) $n = 2097152$ **(b)** $n = 4194304$

Figura 6.18: *Tiempos en un cluster de Pentiums para* $n = 2097152$ *y* $n = 4194304$

comportamiento de estos algoritmos es mucho más parecido al comportamiento de los mismos en el IBM SP2 que en el CRAY T3D. Notemos que el comportamiento numérico de los algoritmos 3.4 y 3.5 no sigue las características observadas en el CRAY T3D. Ahora los tiempos calculados para dichos algoritmos están mucho más próximos al resto de algoritmos de lo que lo estaban en el CRAY T3D. Así, por ejemplo, para un valor pequeño de n como 1024 o 2048, el algoritmo 3.4 ofrece mejores tiempos que los algoritmos 4.3, 4.4, 4.5, 5.2 y 5.3. Esta característica no se produce cuando $n \geq 8192$, donde los algoritmos basados en la fórmula de Sherman-Morrison vuelven a ser más lentos, aunque las diferencias entre ellos no son significativas.

Apéndice A
Conclusiones y líneas futuras de trabajo

En esta memoria se estudian diversos algoritmos BSP del tipo *divide y vencerás* para la resolución de sistemas lineales tridiagonales, utilizando tres métodos distintos. El primero se basa en la aplicación de la fórmula de Sherman-Morrison y constituye una modificación del método del desacoplamiento recursivo; el segundo está basado en la aplicación de la fórmula de Sherman-Morrison-Woodbury para el cálculo de la inversa de una matriz, mientras que el tercero se basa en el método de Bondeli para sistemas tridiagonales.

Se calcula el coste computacional de todos los algoritmos estudiados, siguiendo el modelo de coste que nos proporciona el modelo de computación paralela BSP. También se realiza un estudio del comportamiento paralelo de todos estos algoritmos, basándonos en el cálculo del *speedup* y de la eficiencia de los mismos. Las máquinas donde se predicen estos resultados son un IBM SP2, un CRAY T3D y un cluster de Pentiums. El IBM SP2 admite un switch de alto rendimiento y una conexión de tipo ethernet, lo que produce distintos valores de los parámetros BSP de la máquina.

Basándonos en el método del desacoplamiento recursivo, se obtiene un algoritmo para 2 procesadores y dos algoritmos para 2^m procesadores, con $m > 1$. La diferencia básica entre ambos radica en el modelo de comunicación que utilizan: en el primero de ellos se utiliza un esquema de tipo *fan-in*, mientras que en el segundo se envían los datos al procesador principal, que es el encargado de acabar la

computación. Una comparativa de ambos algoritmos nos permite afirmar que cuando utilizamos una máquina donde las comunicaciones son muy rápidas, como es el CRAY T3D, el algoritmo basado en un esquema de comunicaciones *fan-in* siempre es más rápido que el otro. Esto no sucede para el resto de máquinas, salvo para el caso del IBM SP2 con switch, trabajando con 8 procesadores y para valores de $n \geq 4096$, siendo $n \times n$ el tamaño de la matriz de coeficientes.

Basados en la fórmula de Sherman-Morrison-Woodbury, se estudian cuatro algoritmos BSP cuyas diferencias se encuentran en el modelo de comunicaciones que utilizan y en el método empleado para la resolución del sistema tridiagonal auxiliar necesario para calcular la solución del sistema inicial. Comparando estos algoritmos, concluimos que tanto en el IBM SP2 como en el cluster de Pentiums no se aprecian diferencias notables en los tiempos de ejecución de los mismos, aunque el más rápido de todos es el algoritmo 4.2, en el que cada procesador resuelve el sistema tridiagonal auxiliar secuencialmente y las comunicaciones se producen hacia el procesador principal. En el CRAY T3D, los más rápidos son los algoritmos 4.4 y 4.5, cuya característica principal es la resolución en paralelo del sistema tridiagonal auxiliar utilizando los métodos *recursive doubling* y de reducción cíclica, respectivamente.

Tomando como referencia el método divide y vencerás de Bondeli, se estudian tres algoritmos distintos. En el primero la solución se obtiene en el procesador principal; en el segundo todos los procesadores trabajan hasta el final en la obtención de la solución y en el tercero se resuelve el sistema tridiagonal auxiliar en paralelo utilizando el método *recursive doubling*. Para todas las máquinas el algoritmo más rápido es el algoritmo 5.3, que es el que resuelve en paralelo el sistema tridiagonal auxiliar. Las diferencias son notables respecto a los otros algoritmos.

El estudio comparativo de todos los algoritmos nos lleva a afirmar que aquellos que presentan los mejores tiempos de ejecución son los basados en la fórmula de Sherman-Morrison-Woodbury, especialmente el algoritmo 4.2, junto con el algoritmo 5.3. Los tiempos de estos dos algoritmos para valores grandes de n son prácticamente idénticos. En general, los peores tiempos se obtienen para los algoritmos basados en la fórmula de Sherman-Morrison. Si comparamos a nivel global los valores del *speedup* y de la eficiencia notamos que los mejores valores los proporciona el algoritmo 3.4, que sigue un esquema de comunicación de tipo *fan-in*. Son especialmente destacables los valores que se obtienen en el CRAY T3D. Su comportamiento paralelo es notablemente mejor que el resto de algoritmos. El comportamiento paralelo de todos los algoritmos estudiados en el IBM SP2 con conexión ethernet es bastante deficiente como consecuencia de la lentitud

en las comunicaciones.

En cuanto a las líneas de investigación futuras, por una parte tenemos el estudio y generalización de estos métodos a matrices por bandas, efectuando comparaciones entre los distintos algoritmos con el fin de obtener un algoritmo óptimo. También es interesante considerar el caso de matrices tridiagonales simétricas, analizando las simplificaciones que deben ser introducidas en los algoritmos para aprovechar convenientemente esta estructura.

Una segunda línea de trabajo consistiría en el estudio y profundización de estos métodos del tipo *divide y vencerás* aplicados al problema del cálculo de valores y vectores propios.

Bibliografía

[1] J.C. Agüí y J. Jiménez. A binary tree implementation of a parallel distributed tridiagonal solver. *Parallel Computing*, **21**: 233–241 (1995).

[2] A.V. Aho, J.E. Hopcroft y J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[3] A.V. Aho, J.E. Hopcroft y J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.

[4] S.G. Akl. *Parallel Sorting Algorithms.* Academic Press, Orlando, FL, 1985.

[5] I. Babuska. Numerical stability in problems of linear algebra. *SIAM Journal on Numerical Analysis*, **9**: 53–77 (1972).

[6] I. Bar-On. A practical parallel algorithm for solving band symmetric positive definite systems of linear equations. *ACM Transactions on Mathematical Software*, **13**: 323–332 (1987).

[7] A. Beguelin, J. Dongarra, A. Geist, R. Manchek y V. Sunderam. A users' guide to PVM parallel virtual machine. Technical Report CS-91-136, University of Tennessee, julio 1991.

[8] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci y P. Spirakis. BSP vs LogP. En *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, páginas 25–32, junio 1996.

[9] G. Bilardi y F.P. Preparata. Horizons of parallel computation. *Journal on Parallel and Distributed Computing*, (1995).

[10] R.H. Bisseling. Basic techniques for numerical linear algebra on bulk synchronous parallel computers. En *First Wokshop on Numerical Analysis and Applications*, volumen 1196, páginas 46–57. Springer-Verlag, Berlin, 1997.

[11] R.H. Bisseling, M.W. Goudreau, J.M.D. Hill, K. Lang, B. McColl, S.B. Rao, D.C. Stefanescu, T. Suel y T. Tsantilas. The BSP Programming Library. Report, Oxford University Computing Laboratory, mayo 1997. Puede consultarse en la dirección `http://www.bsp-worldwide.org`

[12] R.H. Bisseling y W.F. McColl. Scientific computing on bulk synchronuos parallel architectures. Report, University of Utrecht, 1994. Una versión reducida aparece en Proceedings of the 13th IFIP World Computer Congress. Volumen I (1994). *B. Pehrson y I. Simon*, editores, Elsevier, páginas 509–514.

[13] S. Bondeli. Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations. *Parallel Computing*, **17**: 419–434 (1991).

[14] S. Bondeli y W. Gander. Cyclic reduction for special tridiagonal matrices. *SIAM Journal on Matrix Analysis and Applications*, **15**: 419–434 (1994).

[15] O. Bonorden, B. Juurlink, I. von Otte y I. Rieping. The Paderborn University BSP (PUB) Library - design, implementation and performance. En *Proc. of 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, San Juan, Puerto Rico, abril 1999.

[16] R. Bru y J. Marín. Coste BSP del método del gradiente conjugado precondicionado. En *Actas de las VII Jornadas de Paralelismo*, Santiago de Compostela, España, septiembre 1996.

[17] R.L. Burden y J. D. Faires. *Análisis Numérico.* Grupo Editorial Iberoamérica, Méjico, 1985.

[18] D. A. Burgess, P. I. Crumpton y J. M. D. Hill. Theory, practice, and a tool for BSP performance prediction. En *EuroPar'96*, páginas 697–705, agosto 1996. Number 1124 in Lecture Notes in Computer Science, Springer-Verlag.

[19] A.W. Burks, H.H. Goldstine y J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Volume 1, part 1. U.S. Army Ordnance Department, The Institute of Advanced Study,, Princenton 1946, Primera edición, 28 junio de 1946, Segunda edición, 2 septiembre 1947,. También en *Papers of John von Newmann on Computing and Computer Theory*, *W. Aspray y A. Burks* editores. Volumen **12** de Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987.

[20] S. Chandra, M. Jain, A. Basu y P.S. Kumar. Sorting algorithms on transputer arrays. *Parallel Computing*, **19**: 595–607 (1993).

[21] L. Chen, J. He y Q. Miller. Algebraic laws for BSP programming. En *Proceedings of Europar'96*, 1996.

[22] K. Chung y L. Shen. Vectorized algorithm for B-spline curve fitting on Cray X-MP EZ 16se. En *Proceedings of the Supercomputing'92 Conference*, 1992.

[23] J.J. Climent, L. Tortosa y A. Zamora. Comparing the BSP cost of different algorithms for tridiagonal systems. Sometido para publicación.

[24] J.J. Climent, L. Tortosa y A. Zamora. Coste BSP en distintos métodos de resolución de sistemas tridiagonales. En *Actas de las VIII Jornadas de Paralelismo*, páginas 31–40, Cáceres, España, septiembre 1997.

[25] J.J. Climent, L. Tortosa y A. Zamora. A note on the recursive decoupling method for solving tridiagonal systems. Sometido para publicación.

[26] J.J. Climent, L. Tortosa y A. Zamora. An hybrid algorithm for solving tridiagonal linear systems in a BSP computer. Sometido para publicación.

[27] J.J. Climent, L. Tortosa y A. Zamora. Un nuevo algoritmo BSP para sistemas tridiagonales. En *Actas de las IX Jornadas de Paralelismo*, páginas 183–190, San Sebastián, España, septiembre 1998.

[28] J.J. Climent, L. Tortosa y A. Zamora. Un algoritmo paralelo para el cálculo de la inversa de una matriz tridiagonal. En *Actas de las IX Jornadas de Paralelismo*, páginas 191–198, San Sebastián, España, septiembre 1998.

[29] J.J. Climent, L. Tortosa y A. Zamora. An hybrid parallel algorithm for solving tridiagonal linear systems versus de Wang's method in a CRAY T3D BSP computer. En *Actas de las X Jornadas de Paralelismo*, páginas 79–84, La Manga del Mar Menor (Murcia), España, septiembre 1999.

[30] J.J. Climent, L. Tortosa y A. Zamora. A recursive decoupling method for solving tridiagonal linear systems in a BSP computer. En *Actas de las X Jornadas de Paralelismo*, páginas 73–78, La Manga del Mar Menor (Murcia), España, septiembre 1999.

[31] J.J. Climent, L. Tortosa y A. Zamora. A BSP recursive divide and conquer algorithm to compute the inverse of a tridiagonal matrix. Journal of Parallel and Distributed Computing, en prensa.

[32] J.M. Conroy. Parallel algorithms for the solution of narrow banded systems. *Applied Numerical Mathematics*, **5**: 409–421 (1989).

[33] C. Cox y J. Knisely. A tridiagonal system solver for distributed memory parallel processors with vector nodes. *Journal of Parallel and Distributed Computing*, **13**: 325–331 (1991).

[34] B.N. Datta. *Numerical Linear Algebra and Applications.* Brooks/Cole Publishing Company, CA, 1995.

[35] W.C. de Leeuw y R. van Liere. Comparing LIC and spot noise. En *Proceedings IEEE Visualization'98*, páginas 359–365, 1998.

[36] D.S. Dodson y S.A. Levin. A tricyclic tridiagonal equation solver. *SIAM Journal on Matrix Analysis and Applications*, **13**: 1246–1254 (Octubre 1992).

[37] S.R. Donaldson, J.M.D. Hill y A. McEwan. Installation and user guide for the Oxford BSP Toolset (v1.4) Implementation of BSPlib. Report, Oxford University Computing Laboratory, septiembre 1998.

[38] S.R. Donaldson, J.M.D. Hill y D.B. Skillicorn. BSP clusters: high performance, reliable and very low cost. Report PRG-TR-5-98, Oxford University Computing Laboratory, 1998.

[39] J.J. Dongarra y L. Johnsson. Solving banded systems on a parallel processor. *Parallel Computing*, **5**: 219–246 (1987).

[40] P. Dubois y G. Rodrigue. An analysis of the recursive doubling algorithm. En *High speed computer and algorithm organization*. Academic Press, New York, 1977.

[41] H. Edelsbrumer. Algorithms in combinatorial geometry. En *EATCS monographs on theoretical computer science*, volumen 10. Springer-Verlag, 1987.

[42] O. Egecioglu, C.K. Koc y J. Laub. A recursive doubling algoritm for solution of tridiagonal systems on hypercube multiprocessors. *Journal of Computational and Applied Mathematics*, **27**: 95–108 (1989).

[43] T. Erlebach. APERITIF, automatic parallelization of divide and conquer algorithms, 1995. Puede obtenerse una copia en `ftp://flop.informatik.tu-muenchen.de/pub/april/`

[44] D.J. Evans. A recursive decoupling method for solving tridiagonal linear systems. *International Journal of Computer Mathematics*, **33**: 95–102 (1990).

[45] D.J. Evans y N.Y. Yousif. Analysis of the performance of the parallel quicksort method. *BIT*, **25**: 106–112 (1985).

[46] M.J. Flynn. Very high speed computing systems. En *Proceedings IEEE*, volumen 14, páginas 1901–1909, 1966.

[47] K.A. Gallivan, R.J. Plemmons y A.H. Sameh. *Parallel algorithms for dense linear algebra computations*. SIAM Review, marzo 1990.

[48] W. Gander y G.H. Golub. Cyclic reduction-history and applications. En *Conference proceedings of the workshop on scientific computing 97*. Springer-Verlag, 1997.

[49] G.A. Geist. Reduction of a general matrix to tridiagonal form. *SIAM Journal on Matrix Analysis and Applications*, **12**: 362–373 (1991).

[50] A. V. Gerbessiotis y L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, **22**: 251–267 (1994).

[51] G.H. Golub y J.M. Ortega. *Scientific Computing. An Introduction with Parallel Computing.* Academic Press, Inc., San Diego, CA, 1993.

[52] G.H. Golub y C. van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 1989.

[53] G.H. Gonnet y R. Baeza-Yates. *Handbook of Algorithms and Data Structures: in Pascal and C.* Addison-Wesley, 1991.

[54] M. Goudreau, K. Lang, S. Rao, T. Suel y T. Tsantilas. Towards efficiency and portability: programming the BSP model. En *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, páginas 1–12, junio 1996.

[55] M. W. Goudreau, J.M.D. Hill, K. Lang, B. McColl, S. B. Rao, D. C. Stefanescu, T. Suel y T. Tsantilas. A Proposal for the BSP Worldwide Standard Library. Report, julio 1996. http://www.bsp-w orldwide.org.

[56] M. W. Goudreau, K. Lang, S. B. Rao y T.Tsantilas. The Green BSP library. Report CS-TR-95-11, University of Central Florida, Department of Computer Science, junio 1995.

[57] J.M. Hill, P.I. Crumpton y D.A. Burgess. Theory, practice, and a tool for BSP performance prediction. En *Lecture Notes in Computer Science*, páginas 697–705. EuroPar'96, No. 1124, Springer-Verlag, agosto 1996.

[58] J.M.D. Hill, W.F. McColl y D.B.Skillicorn. Questions and answers about BSP. Report PRG-TR-15-96, Oxford University Computing Laboratory, Oxford 0X1 3QD, noviembre 1996.

[59] J.M.D. Hill y D.B. Skillicorn. Lessons learned from implementing BSP. Report TR-96-21, Oxford University Computing Laboratory, noviembre 1996.

[60] J.M.D. Hill y D.B. Skillicorn. Practical barrier synchronisation. Report TR-96-16, Oxford University Computing Laboratory, agosto 1996.

[61] M. Hines. Efficient computation of branched nerve equations. *International Journal on Bio-Medical Computing*, **15**: 69–76 (1984).

[62] C.A.R. Hoare. Quicksort. *The Computer Journal*, **5**: 10–15 (1962).

[63] R. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM*, **12**: 95–113 (1965).

[64] R. Hockney. Peformance parameters and benchmarking of supercomputers. *Parallel Computing*, **17**: 1111–1130 (1991).

[65] R. Hockney y C.R. Jesshope. *Parallel Computers* 2. Institute of Physics Publishing, Bristol, 1988.

[66] W. Hoffmann y K. Potma. Experiments with basic linear algebra algorithms on a Meiko computing surface. Report CS-90-02, Department of Computer Systems, University of Amsterdam, 1990.

[67] Y. Huang y W. F. McColl. A two-way BSP algorithm for tridiagonal systems. En *HPCN'97*, Viena, abril 1997. Lecture Notes on Computer Science, Springar-Verlag.

[68] S.L. Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM Journal on Scientific and Statistical Computing*, **8**: 354–392 (1987).

[69] S.L. Johnsson y C. Ho. Multiple tridiagonal systems, the alternating direction methods and boolean cube configured multiprocessors. Report DCS/TR-532, Department of Computer Science, Yale University, 1987.

[70] S.L. Johnsson, Y. Saad y M. Shultz. Alternating directions methods on multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, **8**: 686–700 (1987).

[71] B. H. H. Juurlink y H. A. G. Wijshoff. Communication primitives for BSP computers. *Information Processing Letters*, **58**: 303–310 (1996).

[72] R. M. Karp, M. Luby y F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. En *Proceedings of the Twenty-Fourth Annual ACM Symposium of the Theory of Computing*, páginas 318–326, mayo 1992.

[73] J.P. Keener y K. Bogar. A numerical method for the solution of the bidomain equations in cardiac tissue. *Chaos*, **8**: 234–241 (1998).

[74] D. Kozen. The design and analysis of algorithms. En *Texts and monographs in computer science*. Springer-Verlag, 1992.

[75] A. Krechel, H. Plum y K. Stuben. Parallel solutions of tridiagonal linear systems. En *1st European workshop on hypercube and distributed computing*, Amsterdam, 1989. North-Holland.

[76] A. Krechel, H. Plum y K. Stuben. Parallelization and vectorization aspects of the solution of tridiagonal linear systems. *Parallel Computing*, **14**: 31–40 (1990).

[77] S. Kumar. Solving tridiagonal linear systems on the butterfly parallel computer. *The International Journal on Supercomputers Applications*, **3**: 75–81 (1989).

[78] J.L. Larriba, A. Jorba y J.J. Navarro. A proof of the accuracy of OPM. Report CEPBA 92/10, Universitat Politècnica de Catalunya, Catalunya, Spain, septiembre 1992.

[79] J.L. Larriba, A. Jorba y J.J. Navarro. Solution of strictly diagonal dominant tridiagonal systems on vector computers. Report CEPBA 93/09, Universitat Politècnica de Catalunya, Catalunya, Spain, septiembre 1993.

[80] J.L. Larriba, J.J. Navarro, A. Jorba y O. Roig. Review of general and Toeplitz vector bidiagonal solvers. *Parallel Computing*, **22**: 1091–1126 (1996).

[81] D. Lawrie y A. Sameh. The computation and communication complexity of a parallel banded system solver. *ACM Transactions on Mathematical Software*, **10**: 185–195 (1984).

[82] D. Lecomber. Abstract data types in the bulk synchronous parallel model. Report, Oxford University Computing Laboratory, 1996.

[83] S. Lele. Compact finite schemes with spectral-like resolution. *Journal of Computational Physics*, **103**: 16–42 (1992).

[84] J.W. Lewis. Inversion of tridiagonal matrices. *Numerische Mathematik*, **38**: 333–345 (1982).

[85] J. López. *Arquitectura unificada para sistemas tridiagonales.* Tesis doctoral, Universidad de Málaga, Málaga, 1994.

[86] L.M. Losa, V. Migallón y J. Penadés. Métodos iterativos paralelos en dos etapas. Análisis del coste mediante el modelo BSP. En *Actas de las VII Jornadas de Paralelismo*, Santiago de Compostela, España, septiembre 1996.

[87] A. Martínez. Sobre la eficiencia y estabilidad de los algoritmos básicos del algebra lineal en paralelo. Tesis doctoral, Universidad Politécnica de Valencia, septiembre 1998.

[88] W.F. McColl. BSP Programming. En G.E. Blelloch, K.M. Chandy, y S. Jagannathan, editors, *Proceedings of the DIMACS workshop Parallel Processing Specialist Group Workshop*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, páginas 21–35. American Mathematical Society, 1994.

[89] W.F. McColl. Scalable computing. En J. Van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 100, *Lecture Notes in Computer Science*, páginas 46–61. Springer–Verlag, 1995.

[90] V. Mehrmann. Divide and conquer for block tridiagonal systems. *Parallel Computing*, **19**: 257–279 (1993).

[91] V. Meier. A parallel partition method for solving banded systems of linear equations. *Parallel Computing*, **2**: 33–43 (1985).

[92] G. Meurant. A review on the inverse of symmetric tridiagonal and block tridiagonal matrices. *SIAM Journal on Matrix Analysis and Applications*, **13**: 707–728 (1992).

[93] M. Mignotte. *Mathematiques pour le calcul formel.* Presses Universitaires de France, 1989.

[94] R. Miller. A library for bulk-synchronous parallel programming. En *Proceedings of the British Computer Society Parallel Processing Specialist Group Workshoop on General Purpose Parallel Computing*, páginas 100–108, diciembre 1993.

[95] R. Miller. Two approaches to architecture-independent parallel computation. Tesis Doctoral, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 1994.

[96] J.M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, New York, NY, 1988.

[97] J.M. Ortega y R.G. Voigt. *Solution of Partial Differential Equations on Vector and Parallel Computers.* SIAM, Philadelphia, PA, 1985.

[98] J.M. Ortega, R.G. Voigt y C.H. Romine. A bibliography on parallel and vector numerical algorithms. En *Parallel algorithms for matrix computations.* SIAM, Philadelphia, 1990.

[99] F. Preparata y M. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, New York, NY, 1985.

[100] A. Sameh y D. Kuck. On stable parallel linear system solvers. *Journal of ACM*, **25**: 81–91 (1978).

[101] G. Spaletta y D.J. Evans. The parallel recursive decoupling algorithm for solving tridiagonal linear systems. *Parallel Computing*, **19**: 563–576 (1993).

[102] J. Stoer y R. Burlisch. *Introduction to Numerical Analysis.* Springer-Verlag, New York, NY, 1992.

[103] H. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of ACM*, **20**: 27–38 (1973).

[104] H. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, **1**: 12 (1975).

[105] Q.F. Stout. Supporting divide and conquer algorithms for image processing. *Journal of Parallel and Distributed Computing*, **4**: 95–115 (1987).

[106] X. Sun, H. Zhang, y L.M. Ni. Efficient tridiagonal solvers on multicomputers. *IEEE Transactions on Computers*, **41**: 286–296 (1992).

[107] A.M. Turing. On computable numbers, with an application to the entscheidungsproblem. En *Proceedings of the London Mathematical Society, Series 2*, volumen 42, páginas 230–265, 1936.

[108] R.A. Usnami. Inversion of Jacobi's tridiagonal matrix. *Computers and Mathematics with Applications*, **27**: 59–66 (1994).

[109] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, **33**: 103–111 (1990).

[110] H. A. van der Vorst. Analysis of a parallel solution method for tridiagonal linear systems. *Parallel Computing*, **5**: 303–311 (1987).

[111] H. A. van der Vorst. Parallel solution of bidiagonal systems coming from discretized PDE's. *Trans. Magnetics*, **24**: 286–290 (1988).

[112] J.J. van Wijk. Spot noise-texture synthesis for data visualization. En *Computer Graphics (SIGGRAPH'91 Proceedings)*, volumen 25, páginas 263–272, julio 1988.

[113] C. Walshaw y S.J. Farr. A two-way parallel partition method for solving tridiagonal systems. Report 93.25, University of Leeds, junio 1993.

[114] C.H. Walshaw. Diagonal dominance in the parallel partition method for tridiagonal systems. *SIAM Journal on Matrix Analysis and Applications*, **16**: 1086–1099 (1995).

[115] H.H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, **7**: 170–183 (1981).

[116] M. Wheat y D.J. Evans. An efficient parallel sorting algorithm for shared memory multiprocessors. *Parallel Computing*, **18**: 91–102 (1992).

[117] H. Zhang. On the accuracy of the parallel diagonal dominant algorithm. *Parallel Computing*, **17**: 265–272 (1991).