Contenidos

[1, 2, 3, 4, 5] [6, 7, 8, 9, 10, 11] [12, 13, 14, 15, 16, 17, 18, 19], [20, 21, 22, 23, 24, 25, 26, 27, 28] [29, 30, 31, 32, 33, 34, 35, 36, 37, 38] [39, 40] [41, 42, 43, 44, 45, 46] [47, 48, 49, 50, 51, 52, 53] [54, 55, 56, 57, 58, 59, 60, 61] [62, 63, 64, 65, 66, 67, 68] [69, 70, 71, 72, 73, 74, 75, 76, 77] [78, 79, 80, 81, 82, 83, 84, 85, 86] [87, 88] [89, 90, 91, 92, 93, 94, 95, 96] [97, 98, 99, 100] [101, 102, 103, 104, 105, 106, 107, 108, 109, 110] [111, **?**]

INTRODUCCION

El **objetivo fundamental** de esta Tesis es el **estudio, desarrollo e implementación de algoritmos de Jacobi eficientes para la resolución del Problema Simétrico de Valores Propios**.

Para alcanzar este objetivo trabajaremos fundamentalmente con sistemas Multiprocesadores estudiando métodos que sean además utilizables en sistemas **Masivamente Paralelos**.

El Problema de Valores Propios juega un papel destacado en Algebra Lineal [102], y es especialmente importante el caso simétrico [82]. Este problema se puede resolver por diferentes métodos [34]: método de bisección, algoritmo QR, divide y vencerás (algoritmo de Cuppen), métodos de Jacobi. Más recientemente, se han desarrollado algoritmos basados en la descomposición en subespacios invariantes que usa multiplicaciones matriz-matriz [5, 65].

El método de Jacobi data de 1846 [67, 68], pero cayó en desuso entre los años 60 y 80 debido a la aparición del algoritmo QR que da mejores prestaciones en máquinas secuenciales [4]. Sin embargo, el método de Jacobi es una alternativa que ha adquirido importancia en los últimos años, debido principalmente a la aparición de los computadores paralelos y a algunos resultados sobre la estabibilad de los métodos que parecen indicar que el de Jacobi es más estable [35, 79]. Debido a este mayor paralelismo intrínseco, desde la aparición de los primeros ordenadores paralelos se desarrollaron métodos de Jacobi adecuados a estos sistemas. Así, ya a principios de los 70 apareció el primer estudio del método de Jacobi en Memoria Compartida [85], y a principios de los 80 aparece el primer estudio para Memoria Distribuida [19], y desde entonces hay una gran cantidad de trabajos sobre este método en Multiprocesadores, especialmente en Memoria Distribuida [38, 54, 83, 88, 91, 92, 98, 101].

El uso de máquinas paralelas para la resolución de grandes problemas en diferentes campos de la Ciencia o de la Técnica está propiciado por la necesidad de resolver problemas cada vez más grandes o más complejos, o de resolverlos en tiempo real. Esto, que es cierto para multitud de casos, es cierto también para el Problema de Valores Propios, de ahí que para alcanzar el objetivo que nos proponemos sea necesario el estudio de algoritmos de Jacobi para Multiprocesadores. De este modo, se estudiarán diferentes métodos de Jacobi para la resolución del problema en procesadores secuenciales, Multiprocesadores con Memoria Compartida y Memoria Distribuida. En este último caso las topologías que usaremos serán anillo, hipercubo y malla.

Para obtener algoritmos eficientes en estos tipos de computadores se debe trabajar en varios sentidos:

1. Estudiando **esquemas de almacenamiento** que permitan **explotar la simetría** de la matriz (obteniendo de este modo algoritmos teóricamente óptimos) sin aumentar considerablemente el coste de las comunicaciones con respecto a los algoritmos típicos que no explotan la simetría. El único algoritmo previo que explota la simetría en el método de Jacobi para el problema simétrico de valores propios en multicomputadores es el que se presenta en [95]. Este algoritmo tiene el inconveniente de que utiliza un esquema de almacenamiento adecuado para un anillo lógico de procesadores, lo que produce que tenga una baja escalabilidad. La mayor aportación de esta tesis está en el estudio de esquemas de almacenamiento que permiten explotar la simetría en multicomputadores, usando topología de anillo y de malla, con lo que el uso de una malla lógica resuelve el problema de la baja escalabilidad del algoritmo de [95].

2. Utilizando distintas topologías, en el caso de Multicomputadores (Multiprocesadores con Memoria Distribuida). En este sentido la topología más adecuada para obtener algoritmos escalables parece ser la de malla.

3. Rediseñando los algoritmos para obtener **algoritmos por bloques** que sean ricos en operaciones matriz-matriz, con lo que se podrán obtener algoritmos eficientes usando rutinas optimizadas del tipo BLAS 3 [36]. Además, el uso de estas rutinas hará los algoritmos más portables pues se podrán obtener programas eficientes siempre que se disponga de estas rutinas optimizadas para la máquina en que se ejecuten los programas.

Un método de Jacobi trabaja realizando sucesivas anulaciones de elementos no diagonales de la matriz original, obteniendo una sucesión de matrices con los mismos valores propios que la original, y tendiendo esta sucesión a una matriz diagonal cuyos elementos diagonales son los valores propios que se quiere obtener. Las diferentes formas en que se eligen los elementos a anular dan lugar a distintas versiones del método de Jacobi. Los métodos más usados (por ser más eficientes) son los métodos cíclicos, en los que se realizan sucesivos barridos anulándose en cada barrido una vez cada elemento no diagonal según un determinado orden. Esta ordenación de los elementos no diagonales dentro de cada barrido se llama **orden de Jacobi**.

Para la paralelización de métodos de Jacobi habrá que tener en cuenta que no todos los esquemas de distribución de los datos ni todos los algoritmos por bloques se pueden usar con cualquier ordenación. Por tanto, para obtener algoritmos eficientes habrá que tener en cuenta los siguientes factores: el algoritmo de Jacobi que se paraleliza, la ordenación que se utiliza, la topología del sistema en el caso de Memoria Distribuida, el esquema de distribución de los datos, y el uso de rutinas optimizadas.

La gran velocidad con que están evolucionando los sistemas informáticos hace que sea aconsejable el estudio de los algoritmos de una forma lo más general posible, pero al mismo tiempo, y para contrastar los resultados teóricos con los experimentales, será necesario hacer implementaciones sobre máquinas reales.

Los equipos que se han utilizado para obtener resultados experimentales que puedan ser contrastados con los resultados teóricos son diferentes monoprocesadores, el Multiprocesador de Memoria Compartida Alliant FX/80 [6, 104] y los Multicomputadores (Memoria Distribuida) Supernodo PARSYS SN-1040 basado en Transputers T800 [106, ?], varios iPSC/2 e iPSC/860 [108, 109] y el Touchstone DELTA [111].

Organización de la memoria:

El contenido de cada uno de los capítulos de esta tesis es el siguiente:

- Capítulo 1: Se estudian las características principales de los métodos de Jacobi secuenciales para la solución del Problema de Valores Propios Simétrico: distintos órdenes de Jacobi, el uso de umbrales y el uso de rutinas optimizadas de BLAS 1 y 3. Se aporta una alternativa al uso de umbrales que llamamos método **semiclásico**, y un algoritmo por bloques basado en multiplicaciones matriciales con el que se obtienen resultados satisfactorios en Monoprocesadores, y que se usará para obtener algoritmos en Multiprocesadores tanto de Memoria Compartida como Distribuida.

- Capítulo 2: Se analizan las características principales de los sistemas Multiprocesadores y de los equipos comerciales que se han usado para la obtención de resultados experimentales. También se estudian los procedimientos de comunicación usados en los algoritmos que se han realizado para Memoria Distribuida.

- Capítulo 3: Se estudia la paralelización de métodos de Jacobi en Multiprocesadores (Memoria Compartida). La principal aportación es el diseño de algoritmos por bloques usando el algoritmo secuencial por bloques del capítulo 1 y adecuándolo a las características de los Multiprocesadores.

- Capítulo 4: Se analizan las ideas principales sobre la paralelización de métodos de Jacobi en Multicomputadores (Memoria Distribuida). Los algoritmos que se estudian en este capítulo no explotan la simetría, por lo que la máxima eficiencia teórica que se puede alcanzar con ellos es del 50%. Se utilizan en capítulos sucesivos como punto de referencia para compararlos con otros que sí explotan la simetría y que permiten obtener un 100% de eficiencia teórica.

- Capítulo 5: Se aportan ideas sobre las características que debe tener una distribución de la matriz en un sistema Multicomputador para poder diseñar algoritmos que exploten la simetría y que tengan por tanto una eficiencia teórica del 100%. Se han diseñado dos algoritmos de estas características que son adecuados para topología de anillo y que usan distintos esquemas de almacenamiento.

- Capítulo 6: Se presenta un esquema de almacenamiento adecuado para la explotación de la simetría en una malla de procesadores, y se estudian dos algoritmos que usan este esquema obteniéndose una eficiencia teórica del 100%. Uno de los algoritmos usa un esquema por bloques como el que se estudió en el capítulo 1, obteniéndose altas prestaciones en cuanto a tiempos de ejecución y a escalabilidad.

- Capítulo 7: Se analizan algunas extensiones a otros problemas de los esquemas de almacenamiento que se han presentado en capítulos anteriores. En particular se analizan el Problema de Valores Propios Simétrico Generalizado, la iteración QR, y una extensión del problema de Jacobi en el caso en que la matriz es simétrica y compleja. Este último problema se presenta frecuentemente en el análisis de guiaondas lineales.

- Capítulo 8: Se resumen las conclusiones obtenidas a lo largo de los capítulos anteriores y se proponen algunas direcciones para el trabajo futuro.

Notación:

Para facilitar su localización en el texto, las Proposiciones, Algoritmos, Fórmulas, Figuras y Tablas se etiquetarán usando tres números. Los dos primeros indicarán el capítulo y la sección en que se encuentra, y el tercero el número que le corresponde dentro de la sección. Además, en la cabecera de las páginas pares figura el capítulo y en la de las impares la sección.

Capítulo 1

METODOS DE JACOBI

En este capítulo se estudian las características principales de los métodos de Jacobi secuenciales para la solución del Problema Simétrico de Valores Propios: distintos órdenes de Jacobi, el uso de umbrales y el uso de rutinas optimizadas de BLAS 1 y 3.

Las nuevas aportaciones del capítulo son: un método de Jacobi que llamamos **semiclásico** que puede ser en algunos casos una buena alternativa al uso de umbrales, y un **algoritmo por bloques** basado en multiplicaciones matriciales con el que se obtienen resultados satisfactorios en Monoprocesadores, y que se verá en capítulos sucesivos que se puede usar para obtener algoritmos en Multiprocesadores tanto de Memoria Compartida (capítulo 3) como Distribuida (capítulo 6).

El desarrollo del capítulo es el siguiente: se estudia el Problema Simétrico de Valores Propios, y su resolución por métodos de Jacobi clásico y cíclicos, y se enumeran algunas de las aplicaciones en las que aparecen problemas de valores propios. A continuación se analizan las ideas generales de los órdenes de Jacobi y se estudian algunos de los más usuales: cíclico por filas, par-impar, Round-Robin, de Eberlein, etc... Se estudian diferentes métodos que se usan para reducir el tiempo de ejecución: el uso de umbrales para acelerar la convergencia, el uso de BLAS 1 para reducir el tiempo de ejecución, y se presentan y analizan los que llamamos métodos de Jacobi **semiclásicos** y se comparan los resultados que se obtienen con estos métodos con los obtenidos usando umbral. Por último, se presenta un método de Jacobi **por bloques** que usa BLAS 3.

1.1 El Problema Simétrico de Valores Propios

1.1.1 El Problema de Valores Propios

Dada una matriz A compleja y cuadrada, se llaman valores propios de A a los números complejos λ para los que existe un vector complejo no nulo x que cumple la ecuación $Ax = \lambda x$. El vector x se llama vector propio asociado al valor propio λ.

Aunque este es el planteamiento general del Problema de Valores Propios estudiaremos el problema particular en que A sea una matriz cuadrada, real, simétrica y densa, siendo en este caso λ y x reales. En el capítulo 7 se analizará la extensión de los resultados aquí obtenidos al caso complejo.

1.1.2 Propiedades

Algunas propiedades sobre valores propios que se utilizarán a lo largo de este trabajo y de las que se pueden encontrar demostraciones en cualquier libro de computación matricial [26, 52, 53, 80, 82, 90, 99, 102] son las siguientes:

Proposición 1.1.1 *Los valores propios de una matriz diagonal son sus elementos diagonales.*

Proposición 1.1.2 *Si A y P son dos matrices complejas de tamaño $n \times n$, con P no singular, entonces λ es un valor propio de A con vector propio x si y sólo si λ es un valor propio de $P^{-1}AP$ con vector propio $P^{-1}x$.*

Esta proposición sugiere la posibilidad de computar los valores propios de una matriz A reduciéndola, por medio de transformaciones de semejanza ($P^{-1}AP$ es una transformación de semejanza), a otra matriz cuyos valores propios sean más fáciles de calcular. En particular, es posible transformar una matriz simétrica A en una matriz diagonal D_A, siendo los valores propios de A los elementos diagonales de D_A.

El método de Jacobi es uno de los que operan de este modo.

Proposición 1.1.3 *λ es un valor propio de A si y solo si $A - \lambda I$ es singular.*

Proposición 1.1.4 *Una matriz compleja A, de tamaño $n \times n$, tiene n valores propios que son los ceros de su polinomio característico.*

Proposición 1.1.5 *A y $P^{-1}AP$ tienen el mismo polinomio característico.*

Las proposiciones 1.1.3 y 1.1.4 sugieren otro posible método para calcular los valores propios, que consiste en calcular el polinomio característico de A, $det(A-\lambda I)$, y encontrar las raíces de la ecuación $det(A-\lambda I) = 0$. Este método se usa normalmente cuando la matriz A es estructurada y existe un procedimiento computacionalmente eficiente para calcular $det(A - \lambda I)$.

1.1.3 Métodos de resolución

Los métodos de cálculo de valores y vectores propios para matrices de tamaño mayor que cuatro son todos obligatoriamente iterativos pues de lo contrario se contradeciría el teorema de Abel que asegura que no se puede calcular por radicales las raíces de ecuaciones polinómicas de grado mayor que cuatro. Es claro que para algunos tipos de matrices (diagonales, triangulares, ...) es posible calcular en un número finito de pasos sus valores propios pero esto no contradice la afirmación anterior, pues lo que aseguramos es que no existen métodos directos para el cálculo de los valores propios de "cualquier matriz" para un tamaño dado mayor que cuatro.

Aquí nos interesan los métodos que se aplican a matrices reales simétricas, pero la mayoría de estos métodos se pueden aplicar también (a veces con algunas variaciones) a matrices complejas y no simétricas. Estos métodos se pueden encontrar en multitud de libros de computación matricial [26, 49, 53, 82, 89, 98, 99, 102] y de la mayoría de ellos existen versiones para Multiprocesadores con Memoria Compartida y Distribuida.

Como ya hemos dicho, una posiblilidad consiste en el cálculo del polinomio característico $det(A - \lambda I)$ y la resolución de la ecuación $det(A - \lambda I) = 0$ (proposición 1.1.4), habiendo distintos métodos para la realización de cada una de estas dos fases. Estas técnicas se usan normalmente para matrices estructuradas, pero no para matrices densas debido a que en este caso el coste es grande y se introducen muchos errores de redondeo.

Otros métodos más usados consisten en transformar la matriz A en otra matriz en forma condensada de la que podemos deducir directamente los valores propios (diagonal, triangular) o para la que es más fácil calcularlos (matrices tridiagonales o Hessenberg).

Normalmente se suele reducir la matriz mediante transformaciones de Householder o de Givens a forma tridiagonal si la matriz es simétrica, o Hessenberg si es no simétrica. Posteriormente se puede diagonalizar (o triangularizar) la matriz tridiagonal (o Hessenberg) por medio de algún método como la iteración QR, bisección e iteración inversa, o el algoritmo divide y vencerás de Cuppen [4, 34]. Este método ha sido tradicionalmente el más usado por ser el de mayor eficiencia, en monoprocesadores, de los conocidos y, aunque se han diseñado multitud de algoritmos para Multiprocesadores [7, 58, 86, 93, 95, 96, 97, 100], parece mucho menos adecuado para programación paralela que los métodos de Jacobi.

Los métodos de Jacobi reducen la matriz, si esta es simétrica, a forma diagonal, con lo que los valores propios son los elementos diagonales de esta matriz. Esta reducción se lleva a cabo sin realizar operaciones previas de tridiagonalización o paso a otras formas condensadas intermedias.

1.1.4 Métodos de Jacobi

Los valores y vectores propios de una matriz simétrica A se pueden obtener por medio de métodos de Jacobi. Un método de Jacobi consiste en construir una secuencia de matrices $\{A_l\}$ por medio de

$$A_{l+1} = Q_l A_l Q_l^t \ , \ l = 1, 2, \ldots \tag{1.1.1}$$

donde $A_1 = A$, y Q_l es una rotación de Givens en el plano (p,q), con $1 \leq p,q \leq n$. Bajo ciertas condiciones [47, 53], la secuencia $\{A_l\}$ converge a una matriz diagonal D,

$$D = Q_k Q_{k-1} \dots Q_2 Q_1 A Q_1^t Q_2^t \dots Q_{k-1}^t Q_k^t \tag{1.1.2}$$

cuyos elementos diagonales son los valores propios de A, y los vectores propios son las columnas de $Q_1^t Q_2^t \dots Q_{k-1}^t Q_k^t$.

Cada producto $Q_l A_l Q_l^t$ representa una transformación de semejanza que anula un par de elementos no diagonales, a_{ij} y a_{ji}, de la matriz A_l. La matriz Q_l coincide con la identidad excepto en los elementos $q_{ii} = c$, $q_{ij} = s$, $q_{ji} = -s$, y $q_{jj} = c$, donde $c = \cos\ \theta$, $s = \sin\ \theta$, y [53]

$$\tan\ 2\theta = \frac{2a_{ij}}{a_{ii} - a_{jj}} \quad . \tag{1.1.3}$$

Las distintas maneras de elegir los pares (i,j) han dado lugar a diferentes versiones del método de Jacobi.

1.1.5 Método clásico

El método de Jacobi clásico procede eligiendo en cada iteración como elemento a anular el de mayor valor absoluto de entre los no diagonales. Esto produce que el número de iteraciones sea pequeño pero que el tiempo de ejecución sea muy grande debido al trabajo adicional del cálculo del máximo.

Un esquema del método sería:

Algoritmo 1.1.1 *Esquema del método de Jacobi clásico.*

MIENTRAS $off(A) > cota$
  Calcular a_{ij} con máximo valor absoluto de entre los no diagonales
  Calcular Rotación $Q(i,j,\theta)$ tal que $(QAQ^t)_{ij} = 0$
  $A = QAQ^t$
FINMIENTRAS

donde $off(A) = \sqrt{\sum_{j=1, j\neq i}^{n} \sum_{i=1}^{n} a_{ij}^2}$.

El número de flops necesario para la actualización de la matriz en cada pasada por el núcleo del bucle es de orden $O(n)$, pero el cálculo del máximo produce que el orden de cada pasada por el núcleo del bucle teniendo en cuenta las comparaciones sea $O(n^2)$. De este modo, las $\frac{n(n-1)}{2}$ anulaciones suponen un coste de $O(n^3)$ flops, y de $O(n^4)$ comparaciones (es frecuente denominar barrido a la realización de $\frac{n(n-1)}{2}$ anulaciones, especialmente en los métodos cíclicos que se estudian a continuación).

1.1.6 Métodos cíclicos

Otros métodos de Jacobi [112, 19, 29, 41, 42, 47, 75, 77, 78, 80, 113, 85, 114, 91, 115] proceden realizando sucesivos barridos, anulando en cada barrido una vez cada elemento no diagonal (con lo que cada barrido constará de $\frac{n(n-1)}{2}$ anulaciones). Para ello se utiliza un cierto orden de anulación de los elementos, por ejemplo, el método cíclico por filas anula los elementos en el orden dado por los pares:

$$(1,2),(1,3),\ldots,(1,n),(2,3),(2,4),\ldots,(2,n),\ldots,(n-1,n) \quad . \tag{1.1.4}$$

La convergencia del método de Jacobi con una ordenación por filas se analiza en [47]. En [41, 42, 75, 87] se demuestra que el método de Jacobi converge con otros esquemas de ordenación bajo las mismas condiciones que con el orden cíclico por filas.

Un método de Jacobi cíclico trabaja realizando sucesivos barridos hasta que se cumple algún criterio de convergencia. Normalmente hasta que $off(A) < cota$.

Un esquema en este caso podría ser:

Algoritmo 1.1.2 *Esquema de un método de Jacobi cíclico.*

```
MIENTRAS off(A) > cota
   MIENTRAS queden pares en la ordenación
      Obtener el siguiente par (i, j)
      Calcular Rotación Q(i, j, θ) tal que (QAQ^t)_ij = 0
      A = QAQ^t
   FINMIENTRAS
FINMIENTRAS
```

De esta forma se evita el cálculo del máximo y se obtiene un orden $O(n^3)$ por barrido, pues cada anulación necesita de 11 flops en el cálculo de los parámetros de las rotaciones y $6n+12$ flops en la actualización de A, lo que hace $6n+23$ flops por anulación y $\frac{(6n+23)n(n-1)}{2}$ flops por barrido para el cálculo de los valores propios, lo que da un valor de $3n^3$ flops para los términos de mayor orden. El cálculo de los valores y vectores propios tendrá un coste por barrido de $6n^3$ flops, pues habrá que acumular las rotaciones. Por otro lado, ya sabemos que el coste del método clásico en $\frac{n(n-1)}{2}$ anulaciones, teniendo en cuenta las comparaciones, es $O(n^4)$. Sin embargo, con un método cíclico, al eliminarse los elementos en un orden predeterminado, para alcanzar la convergencia se necesita de más iteraciones que en el método clásico.

En la tabla 1.1.1 comparamos los tiempos de ejecución con el método clásico (JCLA) y con distintos métodos cíclicos (JFIL es el método por filas [47], JEBE con la ordenación de Eberlein [42], JRR con el Round-Robin [19]). Los resultados se muestran en segundos y han sido obtenidos en una HP Apollo 700, utilizando matrices reales,

tamaño	128	192	256	320	384	448	512
JCLA	479	2426	7868	20941			
JFIL	17	55	176	324	626	1004	1754
JEBE	17	55	177	339	634	1015	1772
JRR	17	55	179	340	642	1144	1787

Tabla 1.1.1: Comparación de métodos cíclicos y clásico. Tiempos de ejecución en segundos en una HP Apollo 700.

simétricas y densas con elementos de doble precisión generados aleatoriamente con valores entre -10 y 10.

En la tabla 1.1.2 comparamos la velocidad de convergencia con distintos métodos cíclicos y el método clásico. En los métodos cíclicos se representa el número de barridos y en el método clásico el número de anulaciones dividido por el número de anulaciones de un barrido ($\frac{n(n-1)}{2}$).

tamaño	128	192	256	320	384	448	512
JCLA	4	4	4	4			
JFIL	8	8	9	9	9	9	9
JEBE	8	8	9	9	9	9	9
JRR	8	8	9	9	9	10	9

Tabla 1.1.2: Velocidad de convergencia de métodos cíclicos y clásico (se representa el número de barridos en el caso de los métodos cíclicos y en el método clásico el número de anulaciones dividido por el número de anulaciones de un barrido). Resultados obtenidos en una HP Apollo 700.

Como se puede ver en las tablas 1.1.1 y 1.1.2, los resultados obtenidos con distintos métodos cíclicos son muy similares (el número de barridos se considera que es del orden $O(log_2\ n)$ [19]) por lo que a lo largo de este trabajo utilizaremos distintos métodos cíclicos dependiendo de lo adecuados que sean para los esquemas de almacenamiento y para la topología que utilicemos.

Se puede comprobar también que el método clásico es mucho peor que los métodos cíclicos en tiempo de ejecución, pero que necesita de menos anulaciones para alcanzar la convergencia. Este hecho será utilizado posteriormente para obtener un método mixto entre el clásico y los cíclicos que llamaremos **semiclásico**.

1.2 Aplicaciones

Hay multitud de aplicaciones donde hay que resolver problemas de valores propios [116, 43], siendo en muchos casos estos problemas de grandes dimensiones, de ahí el interés en resolver estos problemas con algoritmos eficientes que necesiten de un tiempo de ejecución reducido. En este sentido tiene especial interés la obtención de algoritmos para Multiprocesadores tanto de Memoria Compartida como de Memoria Distribuida, y el análisis de los algoritmos para Máquinas Masivamente Paralelas, con las que se puedan abordar problemas cada vez de mayores dimensiones conforme el estado de la ciencia y la ingeniería lo exijan.

Hay multitud de problemas de valores propios que tienen interés en diferentes aplicaciones. Aunque en esta tesis analizaremos principalmente la resolución del Problema Simétrico de Valores Propios con matrices densas, reales y simétricas por medio de métodos de Jacobi, y algunas posibles extensiones a problemas generalizados, con matrices complejas y no simétricas, enumeraremos algunas de las aplicaciones más usuales de los problemas de valores propios.

Algunas de las aplicaciones son:

- En **Química Cuántica** en la resolución de **ecuaciones de Scrödinger**. Esta es la aplicación más clara donde aparecen problemas simétricos de valores propios [43], y normalmente se plantea en el estudio de un sistema mecánico-cuántico de n partículas donde se obtiene un *operador hamiltoniano* $\hat{H}$ o energía del sistema y se plantea un Problema Simétrico de Valores Propios de la forma

  $$\hat{H}\psi = E\psi \tag{1.2.1}$$

  donde los números E son los valores propios del operador de energía, que corresponden a las posibles energías del sistema [24, 73, 74].

  Este tipo de aplicaciones presenta una gran variedad de problemas, en algunos casos interesa obtener sólo los valores propios y en otros los valores y vectores propios; en la mayoría de los casos interesa obtener todos los valores (o un número suficientemente grande para que sea preferible calcularlos todos) obteniéndose mayor precisión en la aproximación que se obtiene a los valores reales que se quiere estimar cuanto mayor es el tamaño del problema; en algunos casos los problemas son densos y en otros dispersos [43]; siendo en la mayoría de los problemas las matrices reales pero habiendo también aplicaciones con matrices complejas [10].

  Además, tradicionalmente este tipo de problemas se ha venido resolviendo por medio de métodos de Jacobi [74].

- En el **análisis de guiaondas**. También en este campo aparecen multitud de problemas de valores propios aunque normalmente no se necesita resolver problemas de dimensiones tan grandes como en Química Cuántica. Así, aparecen los siguientes tipos de problemas:

- Estándar de matrices simétricas densas de dimensiones unos pocos cientos, con valores complejos donde hay que calcular todos los valores propios [81].
- Generalizado ($Ax = \lambda Bx$) simétrico definido positivo (A y B simétricas y B definida positiva), con matrices densas de dimensiones unos pocos cientos, con valores complejos donde hay que calcular todos los valores propios generalizados [81].
- Cuadrático ($\lambda^2 Ax + \lambda Bx + Cx = 0$) donde las matrices A, B y C son simétricas, complejas y densas [81].
- Generalizado simétrico definido positivo con matrices reales y dispersas [45, 46].
- Generalizado con matrices reales y dispersas, siendo la matriz A no simétrica y la B simétrica definida positiva [46].
- Generalizado disperso definido positivo con matrices reales no simétricas o complejas no Hermitianas, y necesitando calcular un conjunto de valores y vectores propios [44].

• En **análisis de estructuras**. En [11] se estudian algunas posibles aplicaciones del problema de valores propios en ingeniería, en particular se analizan:

- El problema estándar
$$K\phi = \lambda\phi \tag{1.2.2}$$
donde K es la matriz de rigidez de un elemento o un montaje de elementos, siendo K definida positiva o semipositiva.
También aparece el problema estándar en el análisis de una matriz de masas, de conductividad o de calor.
- El problema generalizado
$$K\phi = \lambda M\phi \tag{1.2.3}$$
donde K es la matriz de rigidez y M la matriz de masa de un elemento o un montaje de elementos. Las matrices K y M son normalmente definidas positivas y banda. También aparece el problema generalizado en el análisis de transferencia de calor, siendo en este caso K la matriz de conductividad del calor y M la matriz de capacidad del calor.

• En **reconocimiento de patrones**. El cálculo de valores y vectores propios se puede usar en este caso para obtener una representación reducida de una serie de datos con una pérdida mínima de información [32, 33]. Se trata de encontrar los valores propios mayores de una matriz de correlaciones, correspondiendo estos valores propios a las características más discriminantes, con lo que será suficiente con almacenar los vectores propios asociados a dichos valores propios para obtener una representación aproximada de los objetos. En este caso las matrices son densas y simétricas, y hay que calcular algunos de los valores propios, aunque en muchos casos no se conoce de antemano la cantidad de valores a calcular.

1.3 Ordenes de Jacobi

Para la obtención de un método de Jacobi cíclico se ordenan los pares de índices correspondientes a elementos no diagonales para realizar los barridos en el orden dado (cuando se trabaja con el par (i, j) se anula el elemento a_{ij} de la matriz y su simétrico). Los diferentes órdenes que se pueden obtener con los $\frac{n(n-1)}{2}$ índices correspondientes a elementos no diagonales se llaman **órdenes de Jacobi**.

En la mayoría de los órdenes de Jacobi se agrupan los índices en pares formando conjuntos de pares de índices que se llaman **conjuntos de Jacobi**. De este modo, se agrupan inicialmente los n índices en $\frac{n}{2}$ pares formando un conjunto de $\frac{n}{2}$ pares de índices, y un orden de Jacobi se obtiene pasando de un conjunto de Jacobi al siguiente por medio de un intercambio de índices que produce un nuevo emparejamiento de índices.

Existe una amplia literatura [15, 41, 42, 75, 76, 77, 78, 80, 87] donde se estudian diferentes métodos para generar órdenes de Jacobi, la convergencia de los métodos cíclicos que usan esos órdenes, la paralelización de estos métodos, el posible trabajo por bloques, ... Aunque en esta Tesis no pretendemos el estudio de los distintos métodos cíclicos de Jacobi. En este apartado haremos un repaso de algunos de los órdenes de Jacobi más usados, para presentar las ideas generales sobre el tema y para analizar las ordenaciones que usaremos en el resto del trabajo. Utilizaremos algunos de ellos en combinación con diversos esquemas de almacenamiento de los datos para la obtención de algoritmos eficientes en Multiprocesadores.

1.3.1 Ideas generales sobre órdenes y conjuntos de Jacobi

En varios de los artículos antes citados se estudia la convergencia de diferentes métodos de Jacobi y se prueba experimentalmente que necesitan de un número de barridos del orden de $O(log_2\ n)$ [19] pudiendo este número variar ligeramente de una ordenación a otra. Aunque no se conoce con exactitud la velocidad de convergencia, Hansen [55] definió un "factor de preferencia" para comparar diferentes esquemas de ordenación.

Se pueden dar diferentes definiciones de lo que es un **orden paralelo óptimo para una determinada arquitectura**:

Luk y Park [76] lo definen como un orden tal que cada barrido se completa en a lo sumo n pasos, sólo se necesita comunicación entre procesadores vecinos en la comunicación de datos, y el movimiento de datos entre dos pasos consecutivos se realiza de una manera sistemática.

Bischof [15] lo define como un orden en el que la redistribución de los datos entre los procesadores, para pasar de un paso al siguiente, implica comunicación sólo entre procesadores vecinos y cada procesador necesita transferir un único bloque de datos. Esta definición puede ser adecuada para anillo e hipercubo (los sistemas que estudia Bischof en [15]) pero no parece adecuada para malla, pues en este caso se realiza normalmente una transferencia de columnas y de filas para obtener la nueva distribución de la matriz en el sistema tras cada iteración, con lo que es necesario transferir al menos dos bloques de datos.

Está claro que cuando n es par el número mínimo de pasos para llevar a cabo un barrido es $n-1$, y cuando n es impar el número mínimo es n, y que es preferible que cada procesador transfiera un único bloque de datos en cada transferencia (considerando una distribución unidimensional). Por todo ello, en nuestro caso consideraremos que un orden óptimo es aquel que cumple las propiedades exigidas por Luk y Park y Bischof necesitando del número mínimo de iteraciones según n sea par o impar (a lo largo del trabajo consideraremos n par, aunque hay órdenes que son óptimos cuando n es par pero no cuando es impar y al revés) e implicando la comunicación de un único bloque de datos cuando se utiliza una topología de anillo. Por tanto, un orden óptimo podrá implicar la comunicación de más de un bloque de datos en una topología de malla, mientras que en la de anillo implicaría la comunicación de un único bloque, por lo que lo consideraremos óptimo.

1.3.2 El orden cíclico por filas

La convergencia del método cíclico de Jacobi usando el orden por filas (descrito en (1.1.4)) está demostrada en [47], y la demostración de la convergencia con otros órdenes se basa normalmente en demostrar que son equivalentes al cíclico por filas (una ordenación se puede obtener de otra simplemente renumerando los índices [75]).

Debido a que se anulan de manera consecutiva elementos en la misma fila, no parece apropiado para obtener algoritmos paralelos, ya que para poder anular el elemento $a_{i,i+k+1}$, habrá que haber anulado previamente $a_{i,i+k}$ y haber actualizado la fila i-ésima, con lo que se pierde la posibilidad de anulación de varios elementos a la vez, mientras que esta posibilidad si se da con otros órdenes cíclicos.

1.3.3 La ordenación par-impar

Es uno de los órdenes de Jacobi más populares, habiendo sido introducido por Sameh en [85] quien lo utilizó en memoria compartida, y habiendo sido utilizado por Stewart

[91] para diseñar un método paralelo en un array lineal de procesadores. Van de Geijn lo usó en [94] para diseñar el primer método de Jacobi que explotaba la simetría de la matriz en Multicomputadores, en concreto en un anillo de procesadores.

En este trabajo lo usamos para diseñar métodos de Jacobi que explotan la simetría en anillo y en malla, y para la obtención de un método por bloques que explota la simetría.

En la figura 1.3.1 mostramos, para $n = 8$, cómo se obtienen los conjuntos de Jacobi para esta ordenación. En esta figura (y en las sucesivas de este apartado) se numeran los índices de 0 a $n-1$, los procesadores en los que se almacenan los bloques de datos correspondientes a esos índices se representan por cajas, y los movimientos de índices para pasar de un conjunto de Jacobi al siguiente se muestran por medio de flechas. En esta figura los pares de índices se representan agrupando los 2 índices por medio de paréntesis. Se puede observar que es una ordenación no óptima pues en los pasos impares el conjunto de pares consta de $\frac{n}{2}$ elementos y en los pares de $\frac{n}{2}-1$, con lo que se necesita de n pasos para llevar a cabo un barrido. Además, si se almacenan las columnas correspondientes a los índices en los procesadores tal como se muestra en la figura, la ordenación es apropiada para un array lineal de procesadores pues el movimiento de índices implicaría movimiento de columnas sólo entre procesadores vecinos. De forma similar se puede utilizar en una malla de procesadores implicando en este caso los movimientos de índices transferencias de filas y columnas de la matriz entre procesadores vecinos en la malla (tal como veremos en capítulos posteriores).

1.3.4 La ordenación de Eberlein

Es también una de las más populares [42] y es apropiada para un anillo bidireccional de procesadores y de modo similar para un toro necesitándose en este caso de transferencias de filas y columnas. El orden es óptimo y el sistema de procesadores debe ser bidireccional pues, como se ve en la figura 1.3.2, en los pasos pares los índices, y por tanto los datos, se mueven en diferente sentido que en los pasos impares. En esta figura los pares de índices están formados por los elementos en un mismo procesador. La transferencia es diferente en los pasos pares y en los impares, y en la figura se muestra cómo se realizan los movimientos de índices en los dos casos.

Ha sido también usada en este trabajo en un anillo de procesadores.

1.3.5 La ordenación de Eggers

Esta ordenación se estudia en [42] y se demuestra que es equivalente a la ordenación de Eberlein y a la Round-Robin. Tiene como ventaja sobre la ordenación de Eberlein el que se necesita comunicación en un único sentido. En la figura 1.3.3 se muestra el movimiento de índices en los 4 primeros pasos. En los pasos pares el movimiento es siempre del mismo modo, y en los impares se mueven hacia la derecha los índices

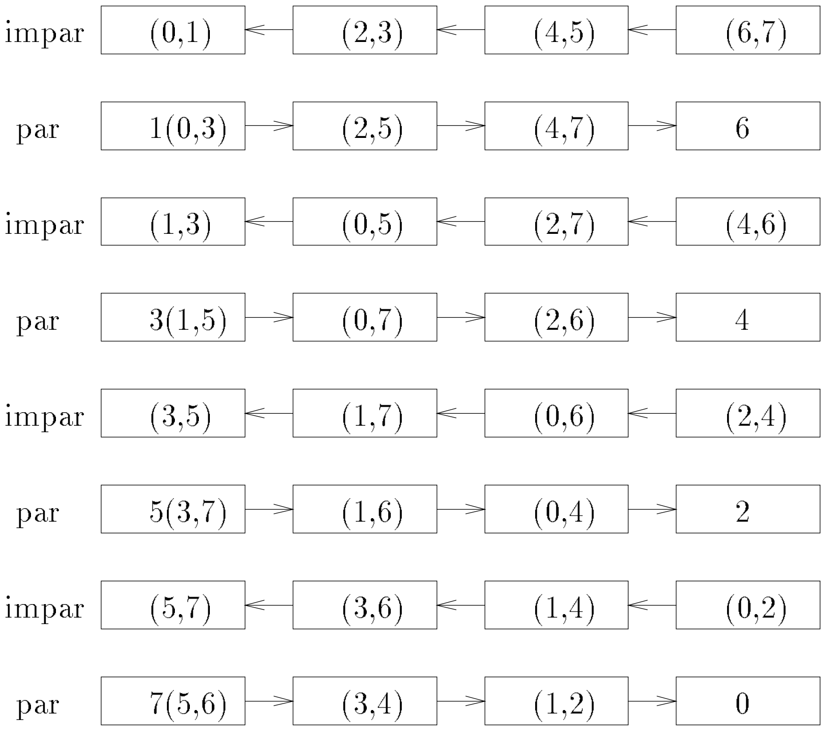


Figura 1.3.1: Ordenación par-impar.

Figura 1.3.2: Ordenación de Eberlein.

superiores de la figura, menos en el procesador en el que el elemento que participa en el movimiento es el de la parte inferior (en el paso $2i+1$ el procesador distinguido es el i-ésimo, si numeramos los procesadores empezando por 0).

Figura 1.3.3: Ordenación de Eggers.

1.3.6 La ordenación Round-Robin

Es uno de los órdenes más antiguos y fué introducido por Brent y Luk [19] que lo usaron para diseñar un algoritmo sistólico en malla para el Problema Simétrico de Valores Propios, aunque no explotaban la simetría de la matriz y sugerían que era posible explotarla usando un array triangular de procesadores.

En la figura 1.3.4 se muestra el movimiento de índices tras cada paso (en este caso se tiene siempre el mismo esquema de movimiento de los índices). Se puede observar que puede ser usado en un array lineal bidireccional de procesadores o en una malla abierta bidireccional (en este trabajo se usará en una malla abierta). En cada transferencia de datos los procesadores no extremos del sistema necesitarán transferir y recibir datos en los dos sentidos, por lo que no es óptimo al necesitarse transferir el doble de datos que con otros órdenes.

1.3.7 El caterpillar-track

Esta ordenación fué introducida en [101]. Se muestra en la figura 1.3.5, donde aparecen los 4 primeros pasos. Los pares de índices están formados por los índices en una misma columna. En los pasos impares tenemos $\frac{n}{2}$ pares de índices y en los pares $\frac{n}{2}-1$, por lo que no es óptimo. Se llama caterpillar-track porque el movimiento de índices es similar al de las ruedas en forma de oruga.

En [76] se estudia este orden y algunas modificaciones que se demuestra que son equivalentes entre sí y a las ordenaciones par-impar y Round-Robin.

Figura 1.3.4: Orden Round-Robin.

Figura 1.3.5: Caterpillar-track.

1.3.8 Ordenaciones por bloques

La realización de algoritmos de Jacobi por bloques es conveniente por dos motivos:

1. El uso de rutinas optimizadas de tipo BLAS 3 [36] hará que con estos métodos se obtengan mejores tiempos de ejecución.

2. El trabajar por bloques permite agrupar los datos a la hora de hacer las transferencias, con lo que se puede ahorrar algo de tiempo de ejecución y evitar algunos puntos de sincronización.

Esto hace que se hayan estudiado también órdenes de Jacobi por bloques [42, 78, 87]. En este caso, se agruparían los datos en bloques (por ejemplo bloques de columnas) y se trabajaría sobre estos bloques del mismo modo que en los métodos anteriores se hacía con una columna. Cada índice se asocia ahora a una columna de bloques en vez de a una columna, y el trabajo sobre los bloques indicados por un par de índices puede consistir en realizar un barrido (con algún orden que puede ser distinto del usado para determinar el movimiento de bloques) sobre elementos de ese bloque, mientras que en los métodos anteriores el trabajo asociado a un par de índices correspondía a una única anulación.

En la figura 1.3.6 mostramos, para $n = 8$ y 2 procesadores, un orden por bloques obtenido con la ordenación de Eggers para los movimientos de bloques. Dentro de cada bloque habría que utilizar alguna otra ordenación para obtener todos los emparejamientos de índices en el bloque (salvo los ya obtenidos previamente).

En este trabajo hemos implementado un algoritmo de Jacobi por bloques explotando la simetría, reorganizando las computaciones para obtener un algoritmo rico en operaciones matriz-matriz y utilizando el orden par-impar para el movimiento de bloques y el orden cíclico por filas para el trabajo dentro de cada bloque. Este método se puede combinar con un esquema de almacenamiento adecuado para obtener algoritmos paralelos, que serán descritos en un capítulo posterior.

1.4 Uso de Umbrales

Hasta ahora hemos estudiado los métodos de Jacobi anulando un par de elementos no diagonales en cada iteración. Para acelerar la convergencia es posible utilizar **umbrales**, de modo que cuando el valor absoluto del elemento a anular esté por debajo del umbral no se realice la anulación ni la actualización de la matriz. De este modo se evitarán anulaciones de elementos pequeños que contribuirían poco a la convergencia.

Utilizando una técnica de umbralización adecuada se asegura la convergencia del método [102] y se disminuye el tiempo de ejecución llegándose a obtener reducciones del 33% en los tiempos de ejecución dependiendo de la técnica de umbralización utilizada y de la máquina en que se ejecuten los programas. En [82, 102] se enumeran algunas posibles técnicas de umbralización:

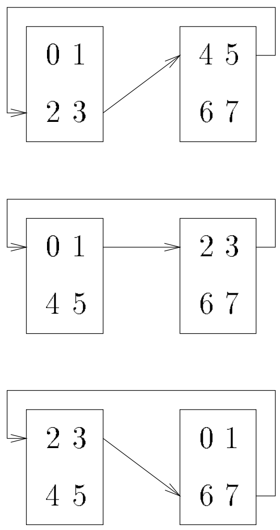


Figura 1.3.6: Ordenación por bloques.

1. Una primera posibilidad es usar un umbral fijo para todas las iteraciones. Si la condición de fin es que $off(A) < cota$, tomando el umbral como $\frac{cota}{n}$ se asegura que se cumple la condición de fin cuando se realiza un barrido completo sin anular ningún elemento.

2. La técnica de Rutishauser consiste en utilizar umbral variable durante los 4 primeros barridos, y en los restantes usar umbral fijo. El umbral variable viene dado por la fórmula:

$$\frac{1}{5}\left(\sum_{i=1}^{n-1}\sum_{j=1,i<j}^{n}\frac{|a_{ij}|}{n}\right)^2 \tag{1.4.1}$$

3. Utilizar una sucesión de umbrales, uno para cada uno de los barridos, disminuyendo el valor del umbral al aumentar el número de barrido, por ejemplo 2^{-3}, 2^{-6}, 2^{-10}, ...

4. La técnica de umbral variable de Kahan y Corneil, donde inicialmente se calcula ω como:

$$\omega = \sum_{i=1}^{n-1}\sum_{j=1,i<j}^{n} a_{ij}^2 \tag{1.4.2}$$

y tras cada actualización se recalcula ω restándole a_{ij}^2. Para decidir si se aplica o no una rotación al elemento a_{ij} estudiaremos si

$$\frac{n(n-1)}{2} a_{ij}^2 > \omega \tag{1.4.3}$$

lo que quiere decir que anularemos sólo los elementos cuyo cuadrado esté por encima de la media de los cuadrados.

Un esquema para los métodos de Jacobi cíclicos con umbral podría ser:

Algoritmo 1.4.1 *Esquema de un método de Jacobi cíclico con umbral.*

```
MIENTRAS off(A) > cota
    MIENTRAS queden pares en la ordenación
        Obtener el siguiente par (i, j)
        SI |a_ij| > umbral
            Calcular Rotation Q(i, j, θ) tal que (QAQ^t)_ij = 0
            A = QAQ^t
            Actualizar umbral
        FINSI
    FINMIENTRAS
FINMIENTRAS
```

Se han realizado algunos experimentos con las diferentes técnicas de umbralización que hemos enumerado utilizando la ordenación cíclica por filas, en los que se observa que se reduce el tiempo de ejecución con el uso de umbrales. Los mejores resultados los hemos obtenido con el umbral de Kahan y Corneil, por lo que en lo sucesivo utilizaremos esta técnica.

En la tabla 1.4.1 se comparan los tiempos de ejecución con el orden cíclico por filas no utilizando y utilizando umbral, y en la tabla 1.4.2 se compara la velocidad de convergencia. En el caso en que se usa umbral se representa el cociente del número de anulaciones llevadas a cabo entre el número de elementos por encima de la diagonal, y cuando no se usa umbral se representa el número de barridos (que es también en este caso el cociente entre el número anulaciones y el número de elementos por encima de la diagonal). Los resultados que se muestran han sido obtenidos en un procesador i860.

En la tabla 1.4.3 se muestran los cocientes de los tiempos de ejecución y del número de barridos obtenidos usando umbral y no usándolo. Se puede observar que la ganancia en el número de barridos es mucho mayor (alrededor de un 39%) que en el tiempo de ejecución (hasta un 27%), lo que es debido al coste adicional de la actualización del umbral, la comparación para decidir si se anula el elemento, etc...

En las tablas 1.4.4 y 1.4.5 se muestran los resultados obtenidos ejecutando los programas en una HP Apollo 700. La reducción en el número de barridos es igual que la obtenida en el procesador i860, por lo que no se muestra. Sin embargo, se observa que en

tamaño	256	320	384	448	512
sin umbral	134.80	279.87	493.09	825.54	1230.12
con umbral	114.39	224.88	383.82	618.49	907.56

Tabla 1.4.1: Uso de umbral (de Kahan y Corneil). Tiempos de ejecución en segundos en el procesador i860.

tamaño	256	320	384	448	512
sin umbral	10	10	10	10	10
con umbral	5.93	6.03	6.08	6.10	6.14

Tabla 1.4.2: Uso de umbral (de Kahan y Corneil). Número de anulaciones partido por $\frac{n(n-1)}{2}$. En un procesador i860.

tamaño	256	320	384	448	512
tiempo	0.84	0.80	0.77	0.74	0.73
barridos	0.59	0.60	0.60	0.61	0.61

Tabla 1.4.3: Cociente entre métodos usando umbral (de Kahan y Corneil) y no usando umbral. En un procesador i860.

diferentes máquinas una misma reducción en los barridos produce reducciones distintas en los tiempos de ejecución debido a la diferencia de los costes de las operaciones aritméticas, de comparación y de acceso a memoria.

tamaño	256	320	384	448	512
sin umbral	87	151	319	506	1087
con umbral	59	110	222	359	713

Tabla 1.4.4: Uso de umbral (de Kahan y Corneil). Tiempos de ejecución en segundos. En una HP Apollo 700.

tamaño	256	320	384	448	512
tiempo	0.68	0.72	0.69	0.71	0.65
barridos	0.59	0.60	0.60	0.61	0.61

Tabla 1.4.5: Cociente entre métodos usando umbral (de Kahan y Corneil) y no usando umbral. En una HP Apollo 700.

1.5 Uso de BLAS 1

En los esquemas que hemos visto de métodos de Jacobi aparece una actualización de la matriz A (QAQ^t) que puede hacerse usando BLAS 1 [37] obteniéndose de este modo programas más eficientes. Esta actualización de la matriz consiste en la aplicación de la rotación a las filas y columnas i y j de A, y esta aplicación se puede hacer usando la rutina **drot** de BLAS 1, con tres llamadas a **drot** para actualizar las partes 1, 2 y 3 de la matriz A en la figura 1.5.1. Puesto que la matriz A es simétrica sólo se trabaja con la parte triangular inferior de la misma.

La rutina **drot** realiza una rotación de ángulo θ. Una llamada a esta rutina tiene la forma $drot(\&dim, \&v1, \&ld1, \&v2, \&ld2, \&c, \&s)$, donde dim representa la dimensión de los vectores a actualizar, $\&v1$ y $\&v2$ son las posiciones de comienzo de los vectores a actualizar, $ldv1$ y $ldv2$ son las *leading dimension* de los vectores $v1$ y $v2$, y c y s son el coseno y el seno, respectivamente, del ángulo θ. De este modo, la llamada $drot(\&dim, \&v1, \&ld1, \&v2, \&ld2, \&c, \&s)$ corresponde a un bucle del tipo:

PARA $i = 0, 1, \ldots, *dim - 1$ HACER

$a = v1[i * (*ld1)]$

$b = v2[i * (*ld2)]$

Figura 1.5.1: Actualización de A usando BLAS 1.

$v1[i*(*ld1)] = c*a + s*b$
$v2[i*(*ld2)] = -s*a + c*b$
FINPARA

En la tabla 1.5.1 comparamos los tiempos de ejecución obtenidos en un procesador i860 con la ordenación cíclica por filas y usando el umbral de Kahan y Corneil, cuando se usa y cuando no se usa BLAS 1. En la tabla 1.5.2 se muestra el cociente entre los tiempos de ejecución cuando se usa BLAS 1 y cuando no se usa. La reducción del tiempo de ejecución es de alrededor del 33% cuando se usa BLAS 1, además, tal como pasa típicamente con el uso de estos núcleos computacionales, la mejora obtenida aumenta al aumentar el tamaño del problema hasta llegar a un punto donde se estabiliza.

tamaño	256	320	384	448	512
con BLAS 1	80.09	153.77	260.44	424.38	615.70
sin BLAS 1	114.39	224.88	383.82	618.49	907.56

Tabla 1.5.1: Uso de BLAS 1. Tiempos de ejecución en segundos, en un procesador i860, con la ordenación cíclica por filas y umbral de Kahan y Corneil.

1.6 Metodo Semiclásico

1.6.1 Idea general

Como ya vimos en la sección 1.1 (tablas 1.1.1 y 1.1.2), los métodos cíclicos producen mucho mejores tiempos de ejecución que el método clásico debido a que realizan un

tamaño	256	320	384	448	512
con BLAS 1/sin BLAS 1	0.70	0.68	0.67	0.69	0.67

Tabla 1.5.2: Uso de BLAS 1. Cociente entre los tiempos de ejecución usando y no usando BLAS 1 en un procesador i860, con la ordenación cíclica por filas y umbral de Kahan y Corneil.

barrido según una determinada ordenación evitando la obtención del máximo antes de cada anulación, obteniéndose de este modo un coste $O(n^3)$ por barrido mientras que con el método clásico el coste por barrido (por $\frac{n(n-1)}{2}$ anulaciones) es $O(n^4)$. Sin embargo, el método clásico necesita de un número de anulaciones pequeño para alcanzar la convergencia, y esto es debido a que en cada paso del algoritmo se anula el elemento de mayor valor absoluto que será el que más contribuya a la convergencia. En esta sección pretendemos estudiar un método que utilice características de los métodos clásico y cíclicos de manera que se acelere la convergencia (se reduzca el número de barridos que se necesita con los métodos cíclicos) aumentando algo el tiempo de ejecución por barrido. Se trata de encontrar un punto intermedio en el que este aumento en el tiempo de ejecución por barrido se vea compensado por la reducción en el número de barridos, obteniéndose de este modo mejores tiempos de ejecución que con los métodos cíclicos. Ideas similares se encuentran en [69] donde se diseña un método para Multiprocesadores con Memoria Compartida y en [84] para métodos de Jacobi unilaterales.

Una primera idea consiste en ordenar antes de cada barrido los elementos no diagonales de mayor a menor valor absoluto y realizar las anulaciones en este orden, con lo que necesitamos en cada barrido de una ordenación previa de $\frac{n(n-1)}{2}$ datos, lo que haciéndolo con un método óptimo como el Quicksort [71], representa un orden promedio $O(n^2 log_2\, n)$. Como el orden del método de Jacobi en cada barrido es $O(n^3)$ esta ordenación previa no modificará el orden por barrido. Por otra parte, la ordenación previa hará que en cada barrido anulemos primero los elementos de mayor valor absoluto, con lo que es posible que se rebaje el número de barridos necesarios para la convergencia.

Una vez hecha la ordenación las primeras iteraciones se hacen sobre los elementos con mayor valor absoluto, pero al anular estos elementos se modifican los valores iniciales de los elementos no diagonales, con lo que estos elementos ya no están ordenados según el orden obtenido al principio del barrido. Por este motivo, en nuestro método no haremos la ordenación de los datos sino que, siguiendo el esquema del Quicksort, tomamos el primer elemento y obtenemos los mayores y los menores que él en valor absoluto, haciendo una nueva llamada a este procedimiento de "semi-ordenación" con los elementos mayores, y así recursivamente hasta que hagamos una llamada con un único elemento. Esta "semi-ordenación" tiene un coste promedio de $O(n^2)$. Anulando en cada barrido los elementos en el orden obtenido con la "semi-ordenación" obtenemos unos tiempos de ejecución que se muestran en la tabla 1.6.1 y el número de barridos de la tabla 1.6.2 (datos obtenidos en un procesador i860).

tamaño	256	320	384	448	512
cíclico por filas	134.80	279.87	493.09	825.54	1230.12
semiclásico	129.05	267.69	465.66	777.96	1167.05

Tabla 1.6.1: Método semiclásico. Tiempos de ejecución en segundos en un procesador i860.

tamaño	256	320	384	448	512
cíclico por filas	10	10	10	10	10
semiclásico	9	9	9	9	9

Tabla 1.6.2: Método semiclásico. Número de barridos en un procesador i860.

En estas dos tablas se observa que utilizando el método semiclásico que hemos explicado se obtiene una pequeña reducción en el tiempo de ejecución, y que esta reducción es debida a que se necesita de un barrido menos para alcanzar la convergencia. Sin embargo, con este método no se mejoran los tiempos obtenidos con el uso de umbrales (tabla 1.4.1).

Intentaremos modificar este método para obtener mejores resultados. Al anularse en las últimas etapas de cada barrido términos que inicialmente son los de menor valor absoluto, estas anulaciones deben contribuir poco a la convergencia del método, por lo que consideramos preferible hacer en cada barrido $\frac{n(n-1)}{2DIVISION}$ anulaciones, indicando $\frac{1}{DIVISION}$ la porción de elementos no diagonales que anulamos en cada barrido (llamaremos a cada uno de estos barridos un **subbarrido**). Cuando $DIVISION = 1$ obtenemos el método anterior y cuando $DIVISION = \frac{n(n-1)}{2}$ el método es el de Jacobi clásico donde hemos obtenido el máximo haciendo una "semi-ordenación".

Compararemos experimentalmente en distintos casos (distintas máquinas, uso de umbral y de BLAS 1) los resultados obtenidos con este segundo método semiclásico y distintos valores de $DIVISION$ con los obtenidos con el método cíclico por filas. Lo que pretendemos es comprobar en qué casos se obtiene una mejora sobre los métodos hasta ahora presentados, y si se puede determinar de una manera empírica un valor óptimo de $DIVISION$.

1.6.2 Resultados experimentales sin uso de umbral

En la tabla 1.6.3 se compara la velocidad de convergencia del método cíclico por filas y el semiclásico para distintos valores de $DIVISION$. Se representa el número de anulaciones realizadas partido por el número de elementos que hay por encima de la diagonal, coincidiendo este valor con el número de barridos en el método cíclico por filas. Los resultados se han obtenido en un procesador i860, pero serían similares en otras máquinas. Se observa que el número de anulaciones disminuye al aumentar $DIVISION$ y que tiende a estabilizarse entre 5 y 6 veces $\frac{n(n-1)}{2}$ según el tamaño de la matriz.

tamaño	256	320	384	448	512
cíclico por filas	10	10	10	10	10
semiclásico 2	7.50	8.00	9.50	8.00	9.00
semiclásico 4	6.75	8.00	9.50	8.00	9.00
semiclásico 6	6.17	6.50	6.67	6.50	6.50
semiclásico 8	5.88	6.25	6.38	6.25	6.25
semiclásico 10	5.60	5.90	6.00	6.00	6.10
semiclásico 12	5.42	5.58	5.67	5.67	5.83
semiclásico 14	5.43	5.36	5.57	5.64	5.78

Tabla 1.6.3: Comparación de la velocidad de convergencia del método cíclico por filas y el método semiclásico con distintos valores de $DIVISION$. Número de anulaciones partido por número de elementos por encima de la diagonal. En un procesador i860.

Como se puede comprobar, se llega a obtener un número de barridos ligeramente menor que el que se obtenía usando el umbral de Kahan y Corneil (tabla 1.4.2). Pero con el método semiclásico se divide cada barrido en un número de subbarridos igual a $DIVISION$ realizándose en cada subbarrido una semiordenación, con lo que el coste por barrido será $O(n^3) + DIVISION\ O(n^2)$. De este modo, se obtiene que una reducción en el número de barridos no implicará siempre una reducción en el tiempo de ejecución, sino que dependerá del valor de $DIVISION$ y del coste de las diferentes operaciones (aritméticas, de comparación y de acceso a memoria) en la máquina en que se ejecuten los programas.

Mostraremos los resultados obtenidos experimentalmente en diferentes casos.

En la tabla 1.6.4 se muestran los tiempos obtenidos en una HP Apollo 700, y en la tabla 1.6.5 los obtenidos en un procesador i860. En ambas tablas se marcan los mejores tiempos obtenidos. Se observa que el valor óptimo de $DIVISION$ aumenta con el tamaño de la matriz y es distinto para distintos sistemas.

Se puede ver, comparando los valores óptimos obtenidos en la tabla 1.4.1 y los obtenidos en la tabla 1.6.5 que este método mejora en tiempo de ejecución muy ligera-

tamaño	256	320	384	448	512
cíclico por filas	87	151	319	506	1087
semiclásico 2	76	131	307	507	937
semiclásico 4	64	123	256	438	791
semiclásico 6	61	121	253	412	750
semiclásico 8	**59**	118	235	401	736
semiclásico 10	**59**	**116**	236	398	715
semiclásico 12	61	122	**234**	395	699
semiclásico 14	62	125	236	392	701
semiclásico 16	64	122	237	**391**	**695**

Tabla 1.6.4: Método semiclásico con distintos valores de $DIVISION$. Tiempo de ejecución en segundos en una HP Apollo 700.

tamaño	256	320	384	448	512
cíclico por filas	134.80	279.87	493.09	825.54	1230.12
semiclásico 2	114.30	245.80	491.58	711.74	1158.88
semiclásico 4	109.87	221.75	393.70	649.02	987.96
semiclásico 6	106.86	223.42	393.15	629.75	927.83
semiclásico 8	**106.29**	226.44	393.96	623.41	919.83
semiclásico 10	107.95	220.81	379.42	616.63	927.87
semiclásico 12	108.96	216.58	**375.63**	**597.92**	**902.46**
semiclásico 14	113.82	**216.31**	376.57	611.24	917.39

Tabla 1.6.5: Método semiclásico con distintos valores de $DIVISION$. Tiempo de ejecución en segundos en un procesador i860.

mente los resultados obtenidos con un método cíclico usando umbral. Intentaremos, por tanto, combinar el uso de umbral con el método semiclásico.

1.6.3 Resultados experimentales usando umbral

Como ya hemos visto, las mejoras relativas con el método semiclásico pueden depender de la máquina en que se ejecuten los programas. Además, hemos visto que usando el método semiclásico se puede reducir en algunos casos ligeramente el tiempo de ejecución que se obtenía usando umbral, por lo que puede ser conveniente combinar las dos técnicas. Aquí estudiaremos la combinación de estas dos técnicas presentando los resultados obtenidos en un procesador i860 y en una HP Apollo 700. En el procesador i860 los mejores resultados se obtienen con el método semiclásico y umbral fijo, y en la HP Apollo 700 los mejores resultados se obtienen con el método cíclico usando el umbral variable de Kahan y Corneil y con el método semiclásico con el mismo umbral variable y con $DIVISION = 1$.

En algunos casos es preferible no usar el mejor umbral junto con el semiclásico porque el uso de umbral hace que en cada barrido o subbarrido anulemos menos elementos que si no usáramos umbral, lo que produciría con el método semiclásico un mayor número de subbarridos y por tanto una mayor cantidad de semiordenaciones, empeorando así el tiempo de ejecución mientras se reduce el número de anulaciones.

Esto se muestra en las tablas 1.6.6, donde se muestran los resultados en la HP Apollo 700, 1.6.7, donde se muestran los de i860 sin usar BLAS 1, y 1.6.8, donde se muestran los de i860 usando BLAS 1.

tamaño	256	320	384	448	512
cíclico con umbral	59	107	216	348	696
semiclásico 1, umbral variable	58	109	218	368	670
semiclásico 2, umbral fijo	67	135	272	478	926
semiclásico 4, umbral fijo	63	119	249	411	837
semiclásico 6, umbral fijo	**61**	116	238	407	749
semiclásico 8, umbral fijo	**61**	**115**	238	406	712
semiclásico 10, umbral fijo	62	117	234	393	707
semiclásico 12, umbral fijo	62	121	**233**	398	705
semiclásico 14, umbral fijo	64	119	237	395	**689**
semiclásico 16, umbral fijo	65	123	240	**390**	691

Tabla 1.6.6: Método semiclásico con umbral. Tiempo de ejecución en segundos en una HP Apollo 700.

tamaño	256	320	384	448	512
cíclico con umbral	114.39	224.88	383.82	618.49	907.56
sc 2, umbral fijo	110.65	237.35	419.25	738.36	1099.50
sc 4, umbral fijo	103.01	212.83	381.59	616.19	997.19
sc 6, umbral fijo	98.53	202.06	359.24	613.45	900.06
sc 8, umbral fijo	**95.51**	196.68	354.67	592.14	857.55
sc 10, umbral fijo	96.63	197.48	345.17	**571.00**	851.49
sc 12, umbral fijo	95.71	200.86	340.74	583.95	850.12
sc 14, umbral fijo	97.76	**193.89**	**339.12**	572.76	**826.20**
sc 1, umbral variable	131	246	406	659	956
sc 2, umbral variable	128	242	417	643	928
sc 3, umbral variable	**123**	**236**	**395**	**629**	**921**
sc 4, umbral variable	127	244	414	640	932

Tabla 1.6.7: Método semiclásico con umbral. Tiempo de ejecución en segundos en un procesador i860 sin BLAS 1.

tamaño	256	320	384	448	512
cíclico con umbral	80.09	153.77	260.44	424.38	615.70
sc 4, umbral fijo	68.18	141.77	245.07	424.23	620.59
sc 6, umbral fijo	**63.64**	134.09	232.38	394.31	620.91
sc 8, umbral fijo	64.30	137.46	233.60	402.80	570.84
sc 10, umbral fijo	63.79	131.94	**228.49**	395.24	567.76
sc 12, umbral fijo	65.31	**130.22**	231.15	397.25	578.66
sc 14, umbral fijo	67.66	132.13	237.56	393.43	**556.30**
sc 16, umbral fijo	68.21	137.15	235.98	**382.06**	566.23
sc 1, umbral variable	96.29	180.76	295.52	476.59	685.41
sc 2, umbral variable	93.76	176.51	301.01	463.41	660.60
sc 3, umbral variable	**88.98**	**171.21**	**291.66**	**456.87**	**655.93**

Tabla 1.6.8: Método semiclásico con umbral. Tiempo de ejecución en segundos en un procesador i860 con BLAS 1.

En la tabla 1.6.9 se muestran los cocientes entre los valores óptimos mostrados en las tablas 1.6.6 a 1.6.8 y el tiempo de ejecución con el método cíclico por filas correspondiente. Se puede observar que los resultados que se obtienen son muy diferentes en las dos máquinas en que se ha experimentado. Esto es debido a la diferencia en el coste de las operaciones aritméticas, de comparación, de acceso a memoria, etc. Lo que se puede deducir es que en algunos casos el uso de un método semiclásico en combinación con una estrategia de umbralización adecuada puede dar lugar a algoritmos que reducen el tiempo de ejecución con respecto a los que sólo usan umbral. Así, los resultados obtenidos en la HP son muy parecidos usando umbral o usando el método semiclásico en combinación con umbral fijo o variable, y lo más adecuado parece ser no usar un método semiclásico. Sin embargo, en un procesador i860 lo más conveniente parece ser usar el método semiclásico en combinación con un umbral fijo, tanto si se usa como si no se usa BLAS 1.

tamaño	256	320	384	448	512
HP: umbral/cíclico	0.67	0.70	0.67	0.68	0.64
HP: sc umb fijo/cíclico	0.70	0.76	0.73	0.77	0.63
HP: sc umb var/cíclico	0.66	0.72	0.68	0.72	0.61
i860 sin BLAS: umbral/cíclico	0.95	0.85	0.82	0.71	0.78
i860 sin BLAS: sc umb fijo/cíclico	0.71	0.69	0.69	0.69	0.67
i860 sin BLAS: sc umb var/cíclico	0.91	0.84	0.80	0.76	0.75
i860 con BLAS: umbral/cíclico	0.95		0.82		0.78
i860 con BLAS: sc umb fijo/cíclico	0.76	0.72	0.72	0.64	0.70
i860 con BLAS: sc umb var/cíclico	1.06	0.95	0.92	0.76	0.83

Tabla 1.6.9: Ganancia con los distintos métodos respecto al cíclico por filas.

1.6.4 Otras posibles mejoras y aplicaciones

Se han realizado algunos otros experimentos intentando acelerar la convergencia del método, pero sin éxito:

1. Ordenando los elementos que se van a anular (en vez de semiordenando) se consigue una mayor reducción en el número de anulaciones necesarias para alcanzar la convergencia, pero esta reducción no compensa (al menos en los experimentos realizados) el tiempo de ejecución adicional necesario para realizar la ordenación, con lo que los tiempos de ejecución que se obtienen son peores ordenando.

2. Utilizando el método propuesto en [69] se reduce aún más el número de anulaciones necesarias para alcanzar la convergencia, pero a costa de un mayor

tiempo de ejecución en cada subbarrido, con lo que se obtienen peores tiempos de ejecución. Este método consiste en tomar el máximo de cada columna en valor absoluto, ordenar estos valores máximos y aplicar las rotaciones en este orden, no anulando elementos en una fila o columna donde ya se ha anulado previamente otro elemento (el método está pensado para Multiprocesadores con Memoria Compartida, por lo que se toma una secuencia de pares de índices que permita aplicar las rotaciones en paralelo). Esta manera de proceder hace que cada barrido se divida en al menos n subbarridos, y el trabajo adicional en cada subbarrido tiene un coste de $\frac{n^2}{2}+n\ log_2\ n$, con lo que el coste por barrido que hay que sumar a los $3n^3$ flops es del orden $O(n^3)$. Combinando este método con el método semiclásico se obtiene una pequeña reducción en el tiempo de ejecución en algunos casos [22, 27], lo que hace que pensemos que utilizando una buena técnica de ordenación (o semiordenación) en paralelo se pueda obtener un método apropiado para Memoria Compartida.

También se podría tratar de aplicar ideas similares a un método de Jacobi para el Problema Simétrico de Valores Propios Generalizado, pero utilizando el método tipo Jacobi que se propone en [80] no se obtiene reducción en el tiempo de ejecución [22].

1.7 Problemas con distintas condiciones

Hasta ahora hemos realizado experimentos generando los elementos de la matriz de manera uniforme, pero puede ser interesante estudiar también el comportamiento de problemas con otro tipo de condiciones. En particular, nos interesará conocer el comportamiento (en cuanto a número de anulaciones y barridos necesarios para la convergencia, y la precisión de los resultados obtenidos) de los algoritmos cuando los valores propios de la matriz A se concentran alrededor de uno o varios valores.

1.7.1 Generación de los problemas

Para generar los problemas con las condiciones mencionadas hemos seguido los siguientes pasos:

1. Se genera una matriz diagonal D con valores propios entre -1 y 1 distribuidos uniformemente.

2. Se elevan esos valores propios a un *exponente*, con lo que se obtiene, a mayor *exponente*, valores propios más cercanos a 0. Los valores usados de *exponente* han sido 1, 3, 5, 7 y 9.

3. Opcionalmente, se suman a distintas partes de la diagonal de D distintos valores v_i, con lo que en las partes donde se haya sumado v_i se tendrán valores cercanos

a v_i y más cercanos a mayor valor de *exponente*. Se han realizado experimentos con matrices con valores propios cercanos a 1 y -1, a 1, 2, -1 y -2, y a 1, 2, 4, 8, -1, -2, -4 y -8.

4. Por último, y para obtener una matriz no diagonal con los mismos valores propios que D, se genera un vector unitario u y se realiza la actualización de rango dos de D utilizando u:

$$D' = \left(I - 2uu^t\right) D \left(I - 2uu^t\right) \tag{1.7.1}$$

En las restantes subsecciones de esta sección mostraremos algunos de los resultados obtenidos con las matrices así generadas.

En las tablas se mostrarán resultados obtenidos con distintos métodos: FIL es el método de Jacobi con ordenación cíclica por filas y sin usar umbral, SCd será el método semiclásico con valor de $DIVISION$ indicado por el número d, KC será el método cíclico por filas cuando se usa el umbral variable de Kahan y Corneil, SCUFd será el método semiclásico con umbral fijo indicando d el valor de $DIVISION$, y SCKCd será el método semiclásico con umbral de Kahan-Corneil indicando d el valor de $DIVISION$.

Debido a que, como ya hemos visto, los tiempos de ejecución de las distintas versiones del método semiclásico dependen en gran medida de las características de la máquina en que se esté trabajando, no estudiaremos los tiempos de ejecución, sino el número de anulaciones y barridos y la precisión.

Todos los experimentos se han realizado en una HP Apollo 700 pero, al no obtener tiempos de ejecución, los resultados serían similares en otras máquinas.

1.7.2 Número de anulaciones

En la tabla 1.7.1 se muestra el número de anulaciones, dividido por el número de elementos no diagonales, necesarias para alcanzar la convergencia con distintos métodos y con distintos tamaños de matriz (con FIL coincidirá con el número de barridos) cuando la matriz A tiene todos sus valores propios cercanos a 0. Se utilizan valores de *exponente* 1, 3, 5 y 7 (al aumentar el valor de *exponente* se obtienen valores propios más próximos).

El valor de $DIVISION$ que se utiliza en los métodos semiclásicos es 12, que es un valor intermedio que nos permite considerar que el número de anulaciones con el valor de $DIVISION$ óptimo estará alrededor del que aquí se muestra.

Se puede observar que al estar los valores propios más cerca aumenta el número de anulaciones con todos los métodos, siendo este aumento considerable en el caso de FIL, y bastante más pequeño en los demás casos, necesitando los métodos semiclásicos con umbral de menos anulaciones que el método cíclico que utiliza el umbral de Kahan-Corneil. Además, el método que menos anulaciones necesita es el semiclásico con umbral de Kahan-Corneil. Aunque de esto no se pueda concluir que con los métodos semiclásicos se mejore el uso de umbral, sí se deduce que cuando los valores propios son cercanos tenemos resultados similares (en cuanto a número de anulaciones) a los que obteníamos generando la matriz A de manera uniforme, por lo que, también en este caso, cuál sea el método más rápido dependerá de la máquina a utilizar.

exponente=1	128	192	256	320
FIL	8	8	9	9
KC	4.21	4.25	4.25	4.22
SCUF12	3.81	3.80	3.81	3.64
SCUV12	3.52	3.43	3.47	3.46
exponente=3	128	192	256	320
FIL	11	13	11	14
KC	4.35	4.45	4.62	4.44
SCUF12	3.90	3.76	3.82	4.05
SCUV12	3.65	3.79	3.67	3.50
exponente=5	128	192	256	320
FIL	13	17	18	16
KC	4.48	4.71	4.70	4.85
SCUF12	4.08	4.11	4.10	4.03
SCUV12	3.70	3.75	3.88	3.83
exponente=7	128	192	256	320
FIL	17	16	19	19
KC	4.93	5.02	5.00	5.45
SCUF12	4.05	4.45	4.32	5.28
SCUV12	3.82	3.76	3.99	3.89

Tabla 1.7.1: Número de anulaciones partido por número de elementos no diagonales, con distintos métodos de Jacobi y distintos tamaños de matriz, cuando los valores propios se acumulan alrededor de 0. En una HP Apollo 700.

Resultados similares se obtienen cuando hay varios puntos donde se acumulan los valores propios. En la tabla 1.7.2 se muestra el número de anulaciones partido por el número de elementos no diagonales, cuando los valores propios están cercanos a 1, 2, -1 y -2. A mayor valor de *exponente* los valores propios estarán más cercanos a esos cuatro puntos.

En este caso también aumenta el número de anulaciones al aumentar *exponente*, aunque en menor medida que lo hace en la tabla 1.7.1 al estar los valores propios más dispersos. Hay un gran aumento en el número de anulaciones en FIL y un aumento pequeño en los demás métodos. Este aumento es especialmente pequeño en KC, lo que hace que al aumentar la acumulación de los valores propios alrededor de 1, 2, -1 y -2 KC sea preferible a los métodos semiclásicos, mientras que con $exponente = 1$ el menor número de anulaciones se obtiene con el método semiclásico usando el umbral de Kahan-Corneil.

exponente=1	128	192	256	320
FIL	8	9	9	9
KC	3.79	3.81	3.77	3.77
SCUF12	3.77	3.67	3.80	3.74
SCUV12	3.64	3.55	3.60	3.46
exponente=3	128	192	256	320
FIL	10	10	10	12
KC	3.75	3.88	3.84	3.92
SCUF12	3.87	4.08	4.05	4.24
SCUV12	3.63	3.61	3.77	3.68
exponente=5	128	192	256	320
FIL	11	12	13	12
KC	3.69	3.80	3.96	3.74
SCUF12	4.20	4.13	4.44	4.49
SCUV12	3.80	3.93	3.88	4.00
exponente=7	128	192	256	320
FIL	11	12	13	16
KC	3.86	3.87	4.04	4.09
SCUF12	4.50	4.58	4.81	4.83
SCUV12	4.02	3.95	4.14	4.16

Tabla 1.7.2: Número de anulaciones partido por número de elementos no diagonales, con distintos métodos de Jacobi y distintos tamaños de matriz, cuando los valores propios se acumulan alrededor de 1, 2, -1 y -2. En una HP Apollo 700.

1.7.3 Número de barridos

Aunque en los métodos que hasta ahora hemos visto una disminución en el número de anulaciones representa una disminución proporcional en el número de flops, no siempre es lo más interesante reducir el número de anulaciones. Esto pasa por ejemplo con los métodos semiclásicos y como veremos en la sección siguiente con un método por bloques. Algunas veces puede ser interesante reducir el número de barridos (el número de veces que se trata cada uno de los elementos no diagonales ya sea anulándolo o no).

En la tabla 1.7.3 representamos el número de barridos necesarios para alcanzar la convergencia con distintos métodos, distintos tamaños de matriz y distintos valores de *exponente* cuando los valores propios se acumulan alrededor de 0. En los métodos semiclásicos se utiliza $DIVISION = 1$ porque este valor es el que nos da menor número de barridos que es lo que pretendemos reducir.

Se observa que el uso del umbral de Kahan-Corneil, aunque reduce el número de anulaciones, aumenta el número de barridos, mientras que la semiordenación de los elementos para anularlos en ese orden reduce el número de barridos, especialmente si se usa umbral fijo para evitar anular elementos de valor absoluto muy pequeño. Este hecho lo aplicaremos en la sección siguiente para combinar el método semiclásico con un método por bloques de manera que se reduzca en algunos casos el número de barridos y por tanto el tiempo de ejecución.

En la tabla 1.7.4 representamos el número de barridos necesarios para alcanzar la convergencia cuando los valores propios se acumulan alrededor de 1, 2, -1 y -2. También en este caso es preferible un método semiclásico, y no está claro si es preferible usar umbral fijo o no usar umbral. En cambio, sí está claro que el uso de umbral variable aumenta el número de barridos.

1.7.4 Precisión de los resultados

Para comparar la precisión con que se obtienen los resultados con los distintos métodos se han sumado los valores absolutos de la diferencia entre los valores propios generados y los obtenidos. En las tablas 1.7.5 y 1.7.6 se muestran estas sumas para distintos métodos, distintos tamaños de matriz, distintos valores de *exponente* y cuando los valores propios son cercanos a 0 (tabla 1.7.5) ó a 1, 2, -1 y -2 (tabla 1.7.6).

Para el estudio de estas tablas hay que hacer las siguientes consideraciones:

1. En todos los algoritmos se ha utilizado como cota para considerar que el método converge el valor 10^{-15}. Esta cota podría ser menor obteniéndose menores errores.

2. En todos los casos se evalúa $off(A)$ y se compara con dicho valor tras cada barrido.

3. En las tablas se muestran diferencias en valor absoluto entre los valores propios generados y los que obtienen los algoritmos, y estas diferencias se suman, con lo que se obtendrán valores mayores a mayor tamaño de la matriz.

4. Hay que tener en cuenta que en el caso de valores propios cercanos a 0 (tabla 1.7.5) los valores que se muestran estarán más cercanos a 0 al aumentar el valor de *exponente*, pues en este caso los valores propios tienden a estar más cercanos a 0. Por el mismo motivo los valores que se muestran deben ser menores en la tabla 1.7.5 que en la 1.7.6.

exponente=1	128	192	256	320
FIL	8	8	9	9
KC	9	9	9	10
SC1	7	7	8	7
SCUF1	7	7	7	7
SCUV1	11	12	12	12
exponente=3	128	192	256	320
FIL	11	13	11	14
KC	14	15	16	16
SC1	10	12	12	14
SCUF1	8	10	11	11
SCUV1	12	14	12	12
exponente=5	128	192	256	320
FIL	13	17	18	16
KC	13	15	18	16
SC1	11	11	13	13
SCUF1	10	12	12	14
SCUV1	13	16	16	15
exponente=7	128	192	256	320
FIL	17	16	19	19
KC	15	17	17	20
SC1	12	13	16	16
SCUF1	15	14	14	15
SCUV1	15	15	16	18

Tabla 1.7.3: Número de barridos, con distintos métodos de Jacobi y distintos tamaños de matriz, cuando los valores propios se acumulan alrededor de 0. En una HP Apollo 700.

exponente=1	128	192	256	320
FIL	8	9	9	9
KC	10	9	10	11
SC1	7	7	7	7
SCUF1	7	7	7	7
SCUV1	12	13	13	13
exponente=3	128	192	256	320
FIL	10	10	10	12
KC	10	11	11	12
SC1	8	9	9	9
SCUF1	7	9	10	10
SCUV1	14	13	13	14
exponente=5	128	192	256	320
FIL	11	12	13	12
KC	11	12	14	13
SC1	10	11	10	11
SCUF1	9	10	11	12
SCUV1	13	13	16	15
exponente=7	128	192	256	320
FIL	11	12	13	16
KC	11	14	14	16
SC1	10	10	11	13
SCUF1	10	11	12	12
SCUV1	14	15	15	16

Tabla 1.7.4: Número de barridos, con distintos métodos de Jacobi y distintos tamaños de matriz, cuando los valores propios se acumulan alrededor de 1, 2, -1 y -2. En una HP Apollo 700.

A la vista de estas tablas se pueden sacar algunas conclusiones:

- La precisión obtenida se puede considerar satisfactoria en todos los casos.
- El método que presenta mayor imprecisión es el que usa la ordenación cíclica por filas. Esto puede ser debido a que en los demás métodos se realizan menos anulaciones y por lo tanto menos acumulación de error.
- La anulación de elementos de entre los de mayor valor absoluto contribuye a una reducción en la imprecisión. De este modo, el uso de umbral es preferible cuando se quieren obtener valores propios de una manera muy precisa.
- La acumulación de los valores propios produce un aumento en la imprecisión, siendo este aumento muy pequeño en el caso de usar umbrales.

1.8 Uso de BLAS 3

Una metodología de trabajo con la que es posible obtener buenos resultados en problemas matriciales numéricos consiste en el diseño de algoritmos por bloques, reorganizando las operaciones para obtener algoritmos ricos en operaciones matriciales de alto nivel, y realizar estas operaciones con rutinas optimizadas, por ejemplo, rutinas de BLAS 3 [36]. De este modo, se pueden resolver problemas matriciales numéricos de una manera muy eficiente y realizar programas independientes de la máquina, si suponemos que el BLAS 3 está disponible para la máquina en que queremos ejecutar los programas. Esta técnica se viene usando recientemente para la obtención de algoritmos eficientes y portables [3, 4, 15, 28, 30, 31].

En este apartado desarrollaremos un algoritmo de Jacobi rico en multiplicaciones matriciales, con lo que se obtendrá una gran reducción en el tiempo de ejecución cuando estas multiplicaciones se hagan usando BLAS 3. Se usará la ordenación par-impar para determinar los movimientos de bloques, y dentro de cada bloque se anularán los elementos usando una ordenación cíclica por filas.

1.8.1 Idea general

Si la matriz A se divide en filas y columnas de bloques cuadrados de tamaño t, es posible hacer un barrido sobre estos bloques usando algún orden de Jacobi (aquí se usará el orden par-impar que se muestra en la figura 1.3.1). De este modo, con $n = 8t$, los bloques se tratan en el orden indicado por los números en la figura 1.8.1, donde el trabajo se hace en la parte triangular inferior de la matriz. Dentro de cada bloque el algoritmo trabaja haciendo un barrido sobre los elementos en el bloque, lo que quiere decir que se anulará una vez cada elemento del bloque usando algún orden

exponente=1	128	192	256	320
FIL	2e-12	7e-12	1e-11	2e-11
KC	3e-13	8e-13	1e-12	2e-11
SC1	8e-13	5e-11	1e-11	2e-11
SC12	3e-13	9e-13	2e-12	2e-12
SCUF1	8e-13	5e-11	1e-11	2e-11
SCUF12	3e-13	1e-12	2e-12	6e-12
SCUV1	2e-13	7e-13	1e-12	2e-12
SCUV12	3e-13	6e-13	1e-12	1e-12
exponente=3	128	192	256	320
FIL	1e-11	1e-11	1e-11	6e-11
KC	3e-13	5e-13	1e-11	1e-11
SC1	1e-12	4e-12	1e-11	8e-12
SC12	2e-13	1e-12	8e-13	1e-12
SCUF1	2e-12	1e-11	2e-11	1e-11
SCUF12	5e-13	7e-13	8e-13	2e-12
SCUV1	2e-13	4e-13	8e-13	1e-12
SCUV12	2e-13	4e-12	6e-13	1e-12
exponente=5	128	192	256	320
FIL	3e-12	4e-11	1e-10	1e-10
KC	1e-13	4e-13	8e-12	1e-11
SC1	5e-13	7e-12	4e-12	1e-11
SC12	2e-10	2e-12	1e-12	1e-12
SCUF1	1e-12	2e-12	5e-12	2e-11
SCUF12	2e-13	4e-13	1e-12	2e-12
SCUV1	1e-13	3e-13	6e-13	1e-12
SCUV12	1e-13	3e-13	5e-13	9e-13
exponente=7	128	192	256	320
FIL	1e-11	4e-11	8e-10	2e-10
KC	3e-13	4e-13	1e-11	1e-11
SC1	1e-12	3e-12	6e-12	1e-11
SC12	1e-12	8e-13	1e-12	8e-13
SCUF1	1e-12	4e-12	1e-11	1e-11
SCUF12	3e-13	3e-12	6e-13	1e-12
SCUV1	1e-13	2e-13	5e-13	1e-12
SCUV12	9e-14	3e-13	4e-13	7e-13

Tabla 1.7.5: Suma de las diferencias en valor absoluto entre los valores propios generados y los obtenidos, con distintos métodos de Jacobi y distintos tamaños de matriz, cuando los valores propios se acumulan alrededor de 0. En una HP Apollo 700.

exponente=1	128	192	256	320
FIL	2e-10	4e-11	6e-11	3e-10
KC	1e-12	3e-12	6e-12	1e-11
SC1	7e-12	2e-11	3e-11	7e-11
SC12	1e-12	3e-12	6e-12	1e-11
SCUF1	7e-12	2e-11	3e-11	7e-11
SCUF12	1e-12	5e-12	8e-12	1e-11
SCUV1	1e-12	3e-12	6e-12	1e-11
SCUV12	1e-12	3e-12	6e-12	1e-11

exponente=3	128	192	256	320
FIL	4e-11	7e-11	2e-10	7e-10
KC	1e-12	3e-12	7e-12	1e-11
SC1	1e-11	1e-10	1e-10	2e-10
SC12	7e-12	6e-12	9e-12	1e-11
SCUF1	8e-10	5e-11	7e-11	4e-10
SCUF12	3e-12	6e-12	9e-12	1e-11
SCUV1	1e-12	3e-12	6e-12	1e-11
SCUV12	1e-12	3e-12	7e-12	1e-11

exponente=5	128	192	256	320
FIL	3e-11	8e-10	2e-8	1e-8
KC	1e-12	4e-12	7e-12	1e-11
SC1	1e-11	5e-9	7e-11	3e-10
SC12	4e-11	1e-9	9e-12	1e-11
SCUF1	3e-11	4e-11	4e-10	4e-10
SCUF12	2e-12	4e-11	7e-11	5e-11
SCUV1	1e-12	4e-12	7e-12	1e-11
SCUV12	1e-12	3e-12	7e-12	1e-11

exponente=7	128	192	256	320
FIL	5e-11	8e-10	5e-9	1e-9
KC	2e-12	4e-12	8e-12	1e-11
SC1	1e-8	1e-10	3e-10	3e-10
SC12	3e-12	7e-12	1e-11	7e-10
SCUF1	1e-12	4e-12	1e-11	1e-11
SCUF12	2e-12	1e-9	3e-11	1e-10
SCUV1	1e-12	4e-12	8e-12	1e-11
SCUV12	1e-12	4e-12	8e-12	1e-11

Tabla 1.7.6: Suma de las diferencias en valor absoluto entre los valores propios generados y los obtenidos, con distintos métodos de Jacobi y distintos tamaños de matriz, cuando los valores propios se acumulan alrededor de 1, 2, -1 y -2. En una HP Apollo 700.

predeterminado (para programar el método se ha utilizado un orden por filas). Pero hay algunos elementos no diagonales que no pertenecen a ninguno de los bloques cuadrados numerados en la figura 1.8.1 (los elementos subdiagonales en bloques en la diagonal principal). Para anular estos elementos, los bloques 1, 2, 3 y 4 de la figura 1.8.1 se consideran de tamaño $2t \times 2t$ añadiendo a cada uno de estos bloques los dos bloques diagonales adyacentes y el bloque simétrico (figura 1.8.2).

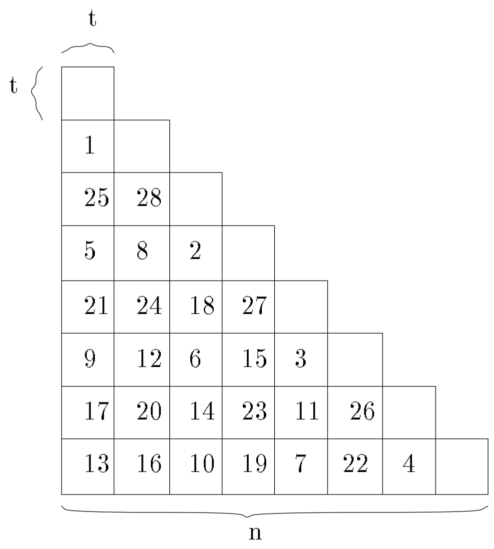


Figura 1.8.1: Orden par-impar por bloques.

El algoritmo procesa los bloques en el orden 1, 2, 3, 4, ..., hasta que se completa un barrido, y realiza sucesivos barridos hasta que alcanza la convergencia.

Es posible diseñar un algoritmo rico en operaciones del nivel 3 (en este caso multiplicaciones matriciales) realizando un subbarrido sobre el bloque 1, explotando la simetría de la matriz y usando BLAS 1. Después de esto, si se acumulan las rotaciones sobre una matriz Q de tamaño $2t \times 2t$, se puede actualizar la primera columna de bloques $2t \times 2t$ postmultiplicando Q por esta columna, y esta multiplicación de matrices se puede realizar usando BLAS 3. El algoritmo trabaja sobre los bloques 2, 3 y 4 de manera similar (con los bloques 2 y 3 habrá que hacer pre y postmultiplicaciones por la matriz de rotaciones, y con el bloque 4 sólo premultiplicaciones).

Después de esto se lleva a cabo un movimiento de filas y columnas, de acuerdo con el orden par-impar. Esto es un trabajo adicional pero produce como resultado que los siguientes bloques de tamaño $2t \times 2t$ que se van a anular queden en la diagonal, con lo

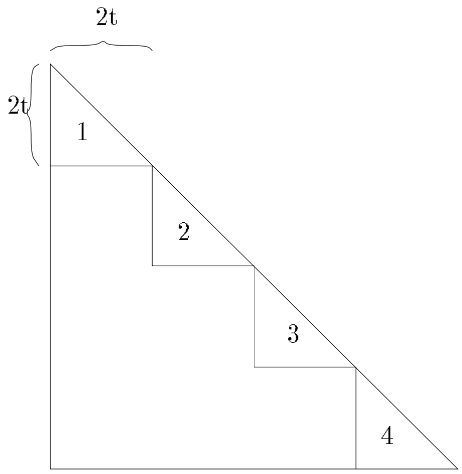


Figura 1.8.2: Bloques triangulares.

que será posible trabajar con estos bloques del mismo modo como se había hecho con los bloques 1, 2, 3 y 4, aunque en este caso los elementos a anular están en bloques de tamaño $t \times t$ dentro de bloques diagonales de tamaño $2t \times 2t$.

De este modo, si inicialmente los pares de índices son $\{(1,2),(3,4),(5,6),(7,8)\}$, tras el intercambio de índices se obtiene un nuevo conjunto de pares $\{(1,4),(3,6),(5,8)\}$, e intercambiando filas y columnas de bloques del mismo modo que se hizo con los índices se obtiene una nueva distribución de los bloques (figura 1.8.3). Ahora el trabajo se hace sobre los bloques diagonales de tamaño $2t \times 2t$ que aparecen en la figura 1.8.3, pero anulando únicamente elementos en los bloques 5, 6 y 7, de tamaño $t \times t$. Después de esto, el algoritmo trabaja de una manera similar con el resto de los bloques hasta que se completa un barrido.

El intercambio de filas y columnas se puede hacer eficientemente usando BLAS 1 para intercambiar y copiar partes de las filas (si la matriz A se almacena por filas) como se muestra en las figuras 1.8.4 y 1.8.5, donde se muestran los intercambios y las copias que se hacen tras iteraciones impares y pares. En estas figuras una flecha $\longleftrightarrow$ representa un intercambio y una flecha $\rightarrow$ una copia. Parte de este movimiento de datos puede evitarse, ya que consiste en la pre y postmultiplicación de la matriz actualizada $A = QAQ^t$ por una matriz de permutación P, y estas multiplicaciones se pueden reorganizar agrupando en la forma $(PQ)A(Q^tP^t)$, de modo que se puede evitar el movimiento de datos en A y hacerlo en Q que tiene casi todos sus elementos nulos, con lo que sólo habría que mover datos en los bloques no nulos de Q (bloques en la diagonal principal de bloques).

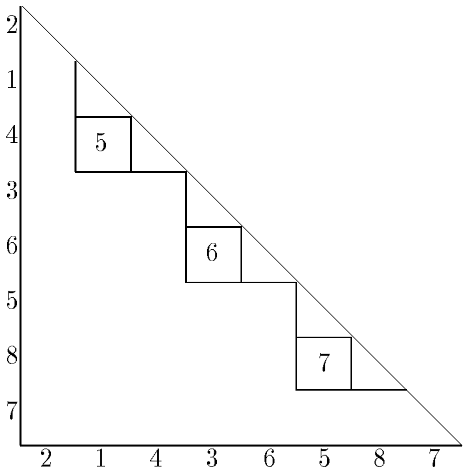


Figura 1.8.3: Trabajo sobre bloques cuadrados.

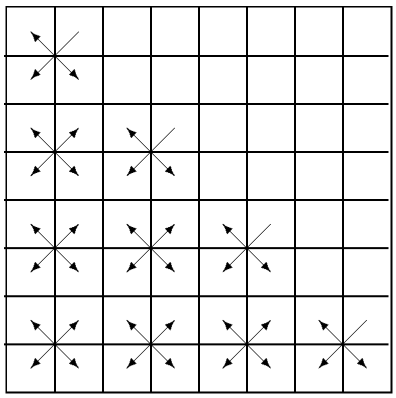


Figura 1.8.4: Movimiento de bloques. Paso impar.

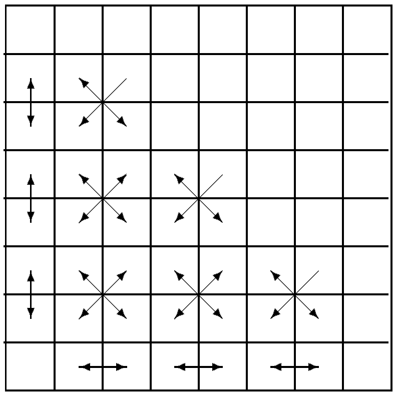


Figura 1.8.5: Movimiento de bloques. Paso par.

1.8.2 Demostración de la convergencia

La convergencia del método se puede demostrar fácilmente utilizando algunos de los resultados que aparecen en [75, 87].

Para hacer la demostración necesitaremos de algunas definiciones y resultados de dichos artículos.

Definición 1.8.1 Ordenes equivalentes. *Sean O_1 y O_2 dos órdenes cíclicos, con T_1 y T_2 productos de transformaciones de la forma T_{ij} que representan un barrido del método de Jacobi con dichos órdenes. Se dice que O_1 y O_2 son equivalentes si T_1 se puede transformar en T_2 por un conjunto de transposiciones de pares de transformaciones T_{ij} que conmuten.*

Definición 1.8.2 Método de Jacobi por bloques. *Se obtiene un método de Jacobi por bloques si se considera la matriz A como una matriz $b \times b$ con b^2 bloques cada uno de tamaño $d \times d$, y un par de índices (i, j) identifica un bloque no diagonal A_{ij} al que se asocia la submatriz*

$$P = \begin{pmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{pmatrix} \tag{1.8.1}$$

y se resuelve el subproblema en P por medio de un barrido de rotaciones de Jacobi en alguno o todos los elementos de la matriz P y se acumulan las rotaciones obteniendo una "matriz de rotaciones" que se aplica a las filas y columna de bloques i y j.

Definición 1.8.3 Orden Antidiagonal. *Un orden antidiagonal es aquel en que una rotación (i, j) con $1 \leq i < j \leq n$ va seguida por*

$$\begin{array}{ll} (i+1, j-1) & si\ j-i>2 \\ (1, i+j) & si\ j-i \leq 2,\ i+j \leq n \\ (i+j+1-n, n) & si\ j-i \leq 2,\ n<i+j<2n-1 \\ (1,2) & si\ j=n\ y\ i=n-1 \end{array} \quad . \qquad (1.8.2)$$

Definición 1.8.4 Orden cíclico frontal. *Si en un orden cíclico O de los pares $\{(i,j),\ 1 \leq i < j \leq n\}$ llamamos $I(i,j)$ el índice en que aparece (i,j), diremos que O es un orden cíclico frontal si se cumple*

$$I(i, j-1) < I(i,j) < I(i+1, j) \qquad (1.8.3)$$

para todo $1 \leq i < j \leq n$.

Proposición 1.8.1 [75].
Los órdenes par-impar y por antidiagonales son equivalentes.

Proposición 1.8.2 [87].
Un orden de Jacobi cíclico es equivalente al orden cíclico por filas si y sólo si es un orden cíclico frontal.

Proposición 1.8.3 [87].
Sean M_1 y M_2 dos métodos de Jacobi por bloques que usan los órdenes O_1 y O_2 para elegir los subproblemas. El orden usado en los subproblemas y la forma en que la matriz es dividida en subbloques son los mismos en los dos métodos. Si M_2 converge y O_1 es equivalente a O_2, entonces M_1 converge.

Proposición 1.8.4 *El método de Jacobi por bloques aquí propuesto converge.*

Demostración

Para demostrar la convergencia de nuestro método por bloques es suficiente con determinar unos métodos M_1 y M_2 con órdenes O_1 y O_2 a los que se les pueda aplicar la proposición 1.8.3 siendo M_1 nuestro método por bloques.

El método por bloques que explicamos en la subsección anterior, y que consideraremos que es el método M_1 de la proposición, es un método por bloques según la definición 1.8.2, en el que se divide la matriz A de tamaño $n \times n$ en subbloques cuadrados de tamaño $t \times t$, y se trabaja dentro de cada subbloque anulando los elementos por un orden cíclico por filas (en algunos bloques se anulan todos los elementos no diagonales y en otros se anula una parte de esos elementos no diagonales). El orden que se usa para generar los subproblemas es el par-impar, por tanto, en la proposición 1.8.3 O_1 será el orden par-impar.

Consideraremos O_2 la ordenación por antidiagonales y M_2 el método por bloques obtenido generando los subproblemas con el orden O_2 y anulando los elementos en los subproblemas asociados a cada bloque de la misma forma que se hace en M_1.

El método cíclico M_2 converge pues es frontal, y O_1 es equivalente a O_2. Por tanto, y como consecuencia de la proposición 1.8.3, M_1 converge.

1.8.3 Costes teóricos

Para estudiar el coste del algoritmo sólo consideraremos el número de flops, aunque habrá un tiempo adicional debido al movimiento de datos.

Como se ve en la figura 1.8.1, el número de bloques de tamaño $t \times t$ ($q = \frac{n}{t}$) es

$$\frac{\frac{n}{t}\left(\frac{n}{t}-1\right)}{2} = \frac{q\,(q-1)}{2} \qquad . \tag{1.8.4}$$

Los primeros $\frac{q}{2}$ bloques son bloques triangulares de tamaño $2t \times 2t$, con lo que se necesitan $3(2t)^3$ flops para realizar un barrido sobre un bloque y $3(2t)^3$ flops para acumular las rotaciones, con lo que se obtienen $6(2t)^3$ flops.

En el resto de los bloques sólo se anulan t^2 elementos, con lo que se necesitan $24t^3$ flops por bloque.

De este modo, el número total de flops necesarios para llevar a cabo un barrido sobre todos los bloques es:

$$\frac{q}{2}6(2t)^3 + \left(\frac{q^2}{2} - q\right)24t^3 = 12n^2t \qquad flops \qquad . \tag{1.8.5}$$

Los dos valores extremos de t son 1 y $\frac{n}{2}$, con lo que los valores extremos en (1.8.5) son $12n^2$ y $6n^3$, pero se obtienen tiempos de ejecución óptimos con valores moderados de t (tal como veremos en los resultados experimentales).

En el algoritmo se realizan también actualizaciones de filas y columnas de bloques. Después de un barrido sobre un bloque de tamaño $2t \times 2t$, se actualizan matrices de tamaños $2t \times (n-2t-i)$ y $i \times 2t$ (donde i toma los valores $0,\ t,\ 2t, \ldots,\ n-2t$), y estas actualizaciones se realizan pre y postmultiplicando por matrices de tamaño $2t \times 2t$, con lo que se necesitan $2\,(2t)^2\,(n-2t) = \ 8t^2\,(n-2t)$ flops. Como la cantidad de bloques $t \times t$ es $\frac{q(q-1)}{2}$, el coste de actualización de la matriz es:

$$\frac{q(q-1)}{2}\,8t^2\,(n-2t) = 4n^3 - 12n^2t + 8nt^2 \quad flops. \tag{1.8.6}$$

De esta manera, sumando las fórmulas (1.8.5) y (1.8.6) se obtiene el coste del algoritmo (si t es pequeño), pero es importante observar que la fórmula (1.8.5) está afectada por un coeficiente correspondiente al nivel 1, y la (1.8.6) por un coeficiente correspondiente al nivel 3. Así, se obtiene un coste de $4n^3$ flops, pero este coste es obtenido con operaciones de nivel 3, y es posible que se obtenga una reducción en el tiempo de ejecución con respecto al obtenido con una versión del método de Jacobi que use operaciones de nivel 1. La mejora obtenida depende de las implementaciones de BLAS 1 y BLAS 3 disponibles en la máquina donde se ejecute el programa pero, si t es pequeña en comparación con n (como se deduce de los resultados experimentales), el cociente entre los tiempos de ejecución usando BLAS 1 y BLAS 3 será

$$\frac{3k_1}{4k_3} \tag{1.8.7}$$

donde k_1 y k_3 son constantes que afectan al coste en flops cuando se usa BLAS 1 y BLAS 3, respectivamente, y que dependen también de la máquina.

1.8.4 Cálculo de vectores propios

Para calcular los vectores propios es necesario acumular las transformaciones ortogonales, y esto se puede hacer usando multiplicaciones matriciales. Después de un subbarrido sobre un bloque de tamaño $2t \times 2t$, es necesario multiplicar la matriz $2t \times 2t$ donde se han acumulado las transformaciones y las filas correspondientes de la matriz Q donde se obtendrán los vectores propios. Para hacer esto eficientemente es necesario un movimiento de filas en Q después de cada paso de la ordenación, y este movimiento se hace para agrupar bloques que van a ser tratados juntos en el siguiente paso. Igual que en el cálculo de los valores propios, este movimiento se hace permutando filas en la matriz $2t \times 2t$ donde se acumularon las rotaciones antes de multiplicar la matriz por Q.

El coste de actualizar Q en cada iteración es $2(2t)^2n = 8t^2n$, y son necesarios $\frac{q(q-1)}{2}$ pasos por barrido. Con lo que el coste por barrido de actualizar Q es:

$$4n^3 - 4n^2t \qquad flops \qquad . \tag{1.8.8}$$

Por tanto, el coste por barrido cuando se computan los valores y los vectores propios es:

$$8k_3n^3 + (12k_1 - 16k_3)n^2t + 8k_3nt^2 \qquad flops \tag{1.8.9}$$

y en este caso se obtiene el mismo cociente (1.8.7) que se obtenía en el cálculo de los valores propios.

1.8.5 Resultados experimentales

Los resultados que presentamos aquí se han obtenido en un procesador i860. Las matrices han sido generadas aleatoriamente con elementos entre -10 y 10, y los programas se han hecho en C usando doble precisión. El número de barridos necesarios para alcanzar la convergencia es similar al obtenido con otros métodos cíclicos, y es aproximadamente $log_2\ n$, por tanto, sólo estudiaremos tiempos por barrido.

En la tabla 1.8.1 se muestran los tiempos de ejecución por barrido en segundos, cuando sólo se calculan los valores propios. Los tiempos que se muestran son los obtenidos con el algoritmo que usa BLAS 1 y con el que usa BLAS 3 con distintos tamaños de bloque. En la tabla se marcan los valores óptimos obtenidos. El valor óptimo del tamaño de los bloques aumenta al aumentar el tamaño de la matriz, y con las matrices usadas se sitúa entre 16 y 32.

tamaño	B1	B3. 4	B3. 8	B3. 16	B3. 32	B3. 64
128	1.14	1.27	0.92	**0.89**	1.10	
256	9.25	8.86	5.73	**4.92**	5.48	7.64
384	35.36	28.77	17.44	**14.18**	15.10	21.12
512	88.54	68.09	40.11	**30.83**	31.81	43.62
640	179.20	128.87	74.65	**57.21**	57.72	76.37
768	306.29	220.16	126.73	94.15	**92.30**	117.47
896	505.91	344.50	196.94	146.41	**141.28**	175.10
1024	738.13	510.51	291.65	211.12	**207.18**	244.39
1152	1128.88	718.58	407.78	298.87	**279.90**	332.86

Tabla 1.8.1: Tiempo de ejecución por barrido en segundos de los algoritmos que usan BLAS 1 y BLAS 3 con diferentes tamaños de bloque, en un procesador i860, cuando se calculan únicamente los valores propios.

La tabla 1.8.2 muestra los tiempos por barrido cuando se computan los valores y los vectores propios. Sólo se muestran tiempos obtenidos con el algoritmo que usa BLAS 3, y se marcan los valores óptimos obtenidos.

tamaño	B3. 4	B3. 8	B3. 16	B3. 32	B3. 64
128	2.08	1.33	**1.15**	1.27	
256	17.47	9.87	7.56	**7.38**	9.06
384	59.40	32.05	23.59	**21.96**	26.53
512	143.53	75.88	53.57	**48.23**	57.15
640		144.92	102.84	**91.48**	104.01

Tabla 1.8.2: Tiempo de ejecución por barrido en segundos del algoritmo que usa BLAS 3 con diferentes tamaños de bloque, en un procesador i860, cuando se calculan los valores y los vectores propios.

En este caso, el valor óptimo de t aumenta más rápidamente que cuando sólo se calculan los valores propios, y esto es debido a que la cantidad de computación es mayor cuando se calculan los valores y los vectores propios, con lo que es posible obtener mayores prestaciones usando BLAS 3.

La tabla 1.8.3 muestra el cociente entre los tiempos de ejecución que se obtienen usando BLAS 3 con tamaño óptimo de bloque y usando BLAS 1, cuando sólo se calculan los valores propios. Este cociente tiende a $\frac{1}{4}$ cuando aumenta el tamaño de la matriz.

tamaño	128	256	384	512	640	768	896	1024	1152
cociente	0.78	0.53	0.40	0.34	0.31	0.30	0.27	0.28	0.24

Tabla 1.8.3: Cociente entre los tiempos de ejecución usando BLAS 3 y BLAS 1, en un procesador i860, cuando se calculan los valores propios.

La tabla 1.8.4 muestra los Mflops que se obtienen con el algoritmo que usa BLAS 3 con tamaño de bloque óptimo, y cuando se calculan valores propios y valores y vectores propios. Se puede observar en esta tabla que se obtienen mejores prestaciones cuando se calculan los valores y los vectores propios, debido a un mayor coste aritmético en este último caso.

tamaño	128	256	384	512	640	768	896	1024	1152
valores	9.42	13.64	15.97	17.41	18.32	19.63	20.36	20.73	21.84
val-vec	14.58	18.18	20.62	22.26	22.92				

Tabla 1.8.4: Mflops obtenidos usando BLAS 3 con tamaño de bloque óptimo, en un procesador i860.

Finalmente, decir que se han realizado otros experimentos intentando mejorar los resultados obtenidos con este algoritmo, pero estos experimentos no han tenido éxito:

1. Cuando se trabaja siempre con bloques de tamaño $2t \times 2t$ (haciendo cada subbarrido sobre estos bloques como un barrido completo y no como un barrido sobre los subbloques de tamaño $t \times t$). En algunos casos, con tamaño pequeño de las matrices, se ha obtenido una reducción de 1 ó 2 barridos, pero el tiempo de ejecución total ha sido peor que con el algoritmo presentado.

2. Cuando se aumenta el número de barridos que se hace sobre cada bloque en los barridos iniciales, también se obtiene una pequeña reducción en el número de barridos en algunos casos cuando la matriz es pequeña, pero se sigue teniendo un peor tiempo de ejecución.

1.8.6 Problemas con distintas condiciones

Al igual que hicimos con los métodos que no trabajan por bloques, podría ser interesante analizar el comportamiento de este método por bloques con distintas condiciones del problema, por lo que se han realizado el mismo tipo de experimentos que se realizó con los algoritmos que no trabajan por bloques y se han obtenido resultados similares:

- El número de barridos necesarios para la convergencia aumenta al aumentar la proximidad de los valores propios, y este aumento en el número de barridos es similar al que se obtenía con el método cíclico por filas.
- La precisión que se obtiene es similar a la que se obtenía con los métodos que no trabajan por bloques.

En las tablas 1.8.5 y 1.8.6 se muestran, respectivamente, en número de barridos para alcanzar la convergencia y la suma de los valores absolutos de la diferencia entre los valores propios generados y los obtenidos cuando se generan las matrices como se explicó en la sección anterior. Los resultados han sido obtenidos en un procesador i860.

	128	256	384
exponente=1	7	8	9
exponente=3	11	13	14
exponente=5	13	17	18
exponente=7	14	19	20

Tabla 1.8.5: Número de barridos necesarios para la convergencia con el método por bloques con tamaño de bloque óptimo para matrices con valores propios cercanos, en un procesador i860.

	128	256	384
exponente=1	6e-12	2e-11	1e-10
exponente=3	7e-12	3e-11	9e-11
exponente=5	5e-11	1e-10	2e-9
exponente=7	5e-11	1e-10	7e-10

Tabla 1.8.6: Suma de los valores absolutos de la diferencia entre los valores propios generados y los calculados con el método por bloques con tamaño de bloque óptimo para matrices con valores propios cercanos, en un procesador i860.

1.8.7 Combinación con el método semiclásico

Las ideas del método semiclásico y del trabajo por bloques usando BLAS 3 son totalmente contrapuestas pues en un caso se acelera la convergencia trabajando elemento a elemento y en otro se reduce el tiempo de ejecución trabajando por bloques de elementos. De cualquier modo, hemos intentado combinar estas dos ideas de manera que el trabajo se siga haciendo por bloques con el mismo esquema que hemos estudiado, pero el orden en que se anulen los elementos dentro de cada bloque se determine realizando una semiordenación de los datos en el bloque. El trabajo sobre cada bloque se puede realizar con distintos valores de $DIVISION$ y realizando una cantidad determinada de subbarridos (SB) sobre cada bloque antes de realizar los intercambios de bloque. De este modo, si por ejemplo $DIVISION = 2$ y $SB = 4$ sobre cada bloque se hará el doble de anulaciones que se hacía en el algoritmo por bloques previamente estudiado. Valores de $DIVISION$ altos deberían contribuir a una mayor aceleración de la convergencia aunque a costa de unos tiempos adicionales por la semiordenación de los elementos (tal como pasaba en el método semiclásico), y valores de SB altos contribuirán a un mayor aprovechamiento del uso de los bloques pues se realizarán más anulaciones antes de realizar los intercambios de bloques, pero harán que se postponga el trabajo sobre los bloques que no están siendo tratados y por tanto la anulación de los mayores elementos en valor absoluto en esos bloques, haciendo que se ralentice la convergencia necesitando de más barridos para alcanzarla.

Hemos realizado experimentos con distintos valores de $DIVISION$ y SB y estos experimentos confirman las ideas que hemos expuesto. Los mejores resultados se han obtenido con $DIVISION = 1$ y $SB = 1$. En la tabla 1.8.7 se muestran los resultados obtenidos en un i860. Aparecen los tiempos en segundos y los barridos necesarios para la convergencia usando el método por bloques (BLAS 3) y el método híbrido (BLAS 3.sem) cuando se calculan los valores propios y con tamaño de bloque 16 y 32. Se observa que en algunos casos se consigue una pequeña reducción en el tiempo de ejecución usando el método híbrido y que esta reducción es debida al ahorro de un barrido para alcanzar la convergencia.

1.9 Conclusiones

Hemos estudiado los métodos de Jacobi para resolver el Problema de Valores Propios Simétrico en Monoprocesadores. Se ha hecho un resumen de los diferentes métodos y se han analizado algunas ideas para la obtención de métodos eficientes. Algunas de las ideas son ampliamente conocidas y otras tienen algo de nuevo o son aplicaciones a este problema de ideas que se han aplicado con éxito a otros problemas.

En concreto, se han utilizado como métodos para reducir el tiempo de ejecución, las siguientes técnicas:

1. **Uso de umbrales.** Esta técnica es bien conocida y puede llegar a producir una

método bloque	BLAS 3				BLAS 3.sem			
	16		32		16		32	
	barridos	tiempo	barridos	tiempo	barridos	tiempo	barridos	tiempo
128	9	**7.70**	10	10.31	9	8.18	9	9.10
256	10	49.06	10	53.59	9	**46.62**	9	50.65
384	10	**144.43**	11	162.20	10	148.35	10	156.51
512	11	346.78	11	344.42	10	**325.77**	10	329.23
640			11	628.29	10	604.41	10	**601.55**
768			11	1012.94	10	999.24	10	**956.28**

Tabla 1.8.7: Combinación del método por bloques y el semiclásico. Tiempos en segundos y número de barridos obtenidos en un i860.

reducción de un 33% en el tiempo de ejecución, aunque el porcentaje de reducción depende de la máquina en que se ejecute el programa. Una buena técnica es la de usar el umbral variable propuesto por Kahan y Corneil [82].

2. **Uso de BLAS 1**. Consiste en utilizar rutinas optimizadas de nivel 1, lo que se puede hacer fácilmente con la rutina drot de BLAS 1 [37]. Utilizando este tipo de rutinas se obtiene una reducción de alrededor del 33%, y se deja parte del trabajo de obtener algoritmos eficientes a la implementación de estas rutinas.

3. **Método semiclásico**. Es un nuevo método [23, 50] aunque se han aplicado ideas semejantes en otros casos [69, 84]. La reducción obtenida en el tiempo de ejecución depende ampliamente de la máquina y de las opciones de compilación. En los experimentos realizados se llega a obtener unas reducciones de entre el 30% y el 40%.

4. **Uso de BLAS 3**. Consiste en utilizar rutinas optimizadas de nivel 3. En este caso no es tan sencilla la utilización de estas rutinas como en el caso de BLAS 1, sino que hay que rediseñar el algoritmo para obtener un esquema por bloques que dé lugar a un método rico en operaciones del tipo matriz-matriz [51]. Hemos visto cómo hacer esto realizando las operaciones de nivel 3 (multiplicaciones de matrices) con BLAS 3, con lo que se obtiene una reducción en el tiempo de ejecución de hasta el 75% y los algoritmos que se obtienen de esta manera son muy portables al dejarse el trabajo de obtener una buena eficiencia en las rutinas de BLAS 3.

Algunas de estas técnicas se pueden combinar para obtener mejores prestaciones. De este modo se puede combinar el uso de BLAS 1 con el de umbrales o con el método semiclásico.

Sin embargo, la combinación del uso de umbrales con el método semiclásico no está tan clara, y puede dar buenos resultados o no dependiendo de la máquina. En el caso de una HP Apollo 700 los resultados experimentales muestran que los mejores resultados en la combinación de estas dos técnicas se obtienen usando umbral variable, pero que los resultados que se obtienen son similares a los obtenidos usando únicamente umbral. En un i860 los mejores resultados se obtienen utilizando umbral fijo con el método semiclásico.

El mejor método es el de diseñar un algoritmo rico en operaciones de nivel 3 y utilizar BLAS 3. Este método da buenos resultados en Monoprocesadores y puede darlos también en Multiprocesadores con Memoria Compartida siempre que se disponga del BLAS 3 optimizado para esa máquina (en el capítulo 3 se verá como hacerlo). Ideas similares se pueden utilizar para diseñar algoritmos en Multiprocesadores con Memoria Distribuida (en el capítulo 6 se estudiará un algoritmo por bloques en una malla de procesadores).

La combinación del método por bloques con las técnicas de umbralización y con el método semiclásico no parece que llegue a dar buenos resultados, aunque hemos mostrado que un método híbrido usando BLAS 3 y el método semiclásico puede reducir en algunos casos muy ligeramente el tiempo de ejecución que se obtiene usando sólo BLAS 3.

Capítulo 2

ARQUITECTURAS PARALELAS

En este capítulo presentamos una panorámica general de los sistemas Multiprocesadores con los que vamos a trabajar en el resto de la tesis. Nuestro objetivo principal es obtener algoritmos paralelos eficientes para la resolución del Problema Simétrico de Valores Propios de matrices densas de tamaño elevado. Para conseguir este objetivo, utilizaremos Multiprocesadores con Memoria Compartida (Multiprocesadores) y con Memoria Distribuida (Multicomputadores). Intentaremos también diseñar algoritmos que sean válidos para Máquinas Masivamente Paralelas, ya que ésta es una de las tendencias actuales en el campo del Procesamiento Paralelo. Se analizarán estos sistemas de una manera general y se describirán los equipos con los que se ha trabajado. En el caso de Multicomputadores analizaremos las distintas topologías que se han usado en el diseño de los algoritmos que se estudian (anillo, hipercubo y malla), detallando los procedimientos de comunicación de datos que se han usado en los algoritmos implementados.

En la primera sección se analizan las características generales de los sistemas Multiprocesadores tanto con Memoria Compartida como Distribuida así como de las máquinas comerciales utilizadas para obtener resultados experimentales de los algoritmos diseñados.

En la segunda sección se estudian las características generales de las comunicaciones en los sistemas Multicomputadores que se han usado, y las primitivas de comunicación utilizadas en los algoritmos diseñados para estos sistemas.

2.1 Multiprocesadores

2.1.1 Memoria Compartida

Características generales

Las características principales de los sistemas de Memoria Compartida son [63, 66, 70]:

1. Todos los procesadores comparten una memoria global y cada uno de ellos dispone de una memoria local de menor dimensión. Por este motivo, los sistemas se llaman **fuertemente acoplados**.

2. Los procesadores comparten los datos y la comunicación entre ellos se realiza por medio de esta memoria global.

3. El acceso a esta memoria se realiza por medio de un bus común, una red crossbar o una red multietapa.

4. Son sistemas del tipo MIMD y adecuados para paralelismo de grano grueso.

5. Suelen tener procesadores vectoriales, con lo que se puede obtener paralelismo a dos niveles: de forma global dividiendo el trabajo entre los procesadores, y en cada procesador utilizando procesamiento vectorial.

Por tanto, las ideas generales a tener en cuenta a la hora de desarrollar algoritmos paralelos para estos sistemas serán:

1. División del trabajo de forma global, evitando los conflictos de acceso a memoria y propiciando la reusabilidad de los datos por los procesadores.

2. Organización del trabajo en cada procesador para aprovechar al máximo el procesamiento vectorial.

3. El uso de núcleos computacionales del mayor nivel posible, lo que hará que los programas sean más eficientes y portables. Esto producirá la necesidad de rediseñar los algoritmos para obtener el mayor provecho de dichos núcleos.

Equipos comerciales

El equipo de Memoria Compartida que se ha usado es un Alliant FX/80 [104]. Este es un Multiprocesador típico de Memoria Compartida por lo que tiene todas las características enumeradas usando una arquitectura de bus. Consta de 8 procesadores con capacidad vectorial y el compilador usado tiene opciones de compilación para paralelizar y/o vectorizar los programas. Se pueden utilizar también núcleos computacionales como BLAS [105].

2.1.2 Memoria Distribuida

Características generales

Las características principales de los sistemas de Memoria Distribuida son [63, 66, 70]:

1. El sistema está formado por varios procesadores cada uno con una memoria local. Por este motivo, los sistemas se llaman **débilmente acoplados**.

2. Los procesadores comparten los datos por medio de paso de mensajes. Para estudiar el tiempo de las comunicaciones en estos sistemas se consideran dos parámetros: el tiempo de inicio de la comunicación (start-up time) que designaremos por β, y el tiempo de transmisión de un dato (sending word time) que designaremos por τ. El coste de enviar N datos desde un procesador a otro vecino será $\beta + \tau N$.

3. La comunicación entre los distintos procesadores se hace utilizando una red de interconexión por la que circulan los mensajes. Esta red de interconexión determina la topología del sistema.

4. Son sistemas del tipo MIMD y adecuados para paralelismo de grano grueso.

5. Para obtener paralelismo será necesario determinar el esquema de división del trabajo entre los procesadores y el tiempo de comunicación que conlleva esa división.

Equipos comerciales

Los sistemas de Memoria Distribuida que se han usado para obtener los resultados experimentales que se muestran en este trabajo son:

1. Un Supernodo PARSYS SN-1040 basado en Transputers T800 en el que se pueden usar hasta 16 procesadores.

2. Un Intel iPSC/2 con 4 procesadores.

3. Dos Intel iPSC/860, uno con 8 procesadores y otro con 32.

4. Un Intel Touchstone DELTA con 512 procesadores.

De estos sistemas describimos a continuación las características principales:

Supernodo

Un Supernodo es un Multicomputador basado en Transputers [106]. El que hemos usado para obtener resultados experimentales de las implementaciones de los algoritmos que aquí se presentan es un PARSYS SN-1040 con 23 Transputers T800. En los experimentos realizados se han utilizado hasta 18 Transputers, uno como HOST, 16 como nodos, con 4 Mb de memoria RAM cada uno, y un Transputer auxiliar que ha sido usado de puente entre dos nodos del sistema para trabajar en un hipercubo de dimensión 4. Cada Transputer tiene cuatro enlaces por los que se puede realizar comunicaciones, conectados a una red crossbar (Backplane Switch) con la que se implementa la topología de la red de interconnexión. Además hay otra red crossbar (C004 Switch) que permite la comunicación con el mundo exterior a través de un PC (figura 2.1.1).

Un estudio de las prestaciones del Supernodo se puede encontrar en [2].

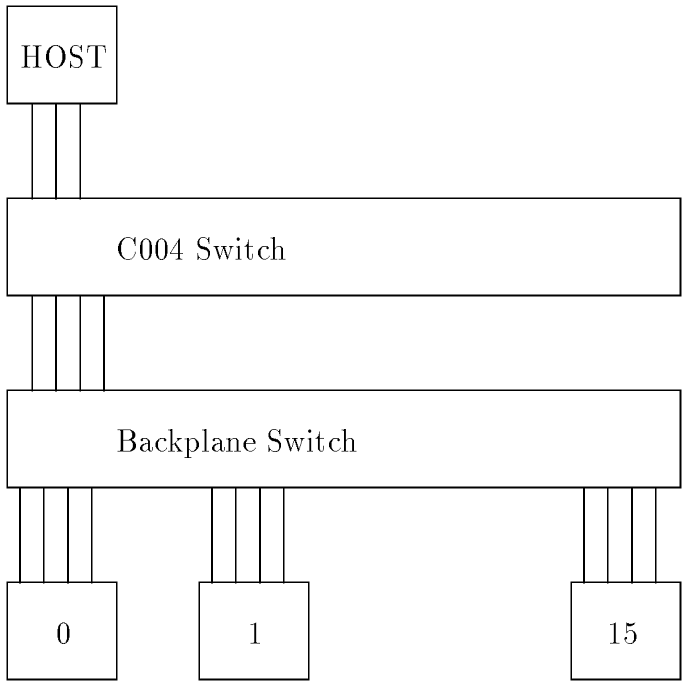


Figura 2.1.1: Arquitectura de un Supernodo.

Por medio del Backplane Switch se pueden configurar distintas topologías. En la figura 2.1.2 se tiene un hipercubo de dimensión 3.

En cada procesador se pueden declarar varios procesos que trabajarían compartiendo el tiempo de ejecución y enviando mensajes utilizando enlaces lógicos. Los enlaces lógicos y las conexiones entre enlaces físicos que quedan libres se hacen utilizando un lenguaje especial para ese fin [110].

Trabajando en OCCAM [?] la declaración de distintos procesos se hace de forma natural y es posible hacer transferencias y recepciones de datos al mismo tiempo, pero hemos trabajado con 3L C [110] y considerado que las comunicaciones son simples (es decir, consideraremos que cada procesador, en un momento dado, puede enviar o recibir datos, pero no las dos cosas a la vez, y que esta recepción o envío se realiza por un único canal). Se podría haber asociado a cada procesador, aparte del proceso que realiza el trabajo aritmético, otros procesos para recepción y envío de mensajes, con lo que las comunicaciones no serían simples pero habría más procesos compartiendo el tiempo

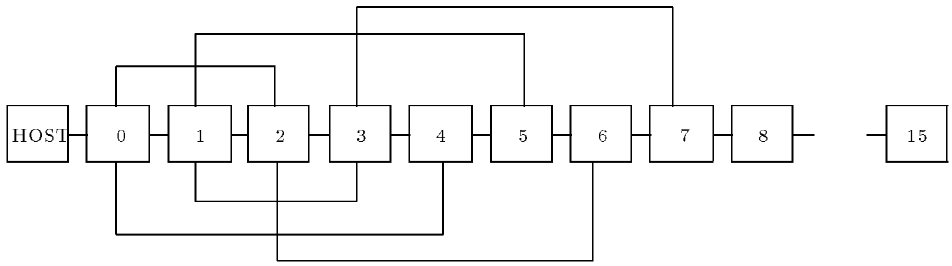


Figura 2.1.2: Hipercubo de dimensión 3 en un Supernodo.

de ejecución y necesitaríamos sincronización en cada procesador entre el proceso que realiza el trabajo aritmético y los procesos auxiliares.

Intel iPSC/2

Un iPSC/2 [108] tiene topología de hipercubo y puede constar de hasta 128 procesadores 80386, cada uno con una memoria local de 1, 4, 8 ó 16 Mb, un coprocesador numérico 80387 que puede ser sustituido por un procesador SX para acelerar la aritmética en coma flotante, y se puede añadir un procesador vectorial VX (en este caso la máxima memoria permitida es de 8 Mb). Además, en cada procesador hay un DCM (Direct-Connect Module) encargado de llevar a cabo las comunicaciones, y que posibilita el ver el sistema como completamente conectado, pues al encargarse este DCM de las comunicaciones no se necesita "store and forward" para transmitir los mensajes. De este modo, se pueden ejecutar los algoritmos para cualquier topología en un iPSC/2 sin tener en cuenta que éste tiene topología de hipercubo, aunque el olvidarnos de esta característica reducirá un poco las prestaciones de los programas.

Un System Resource Manager (SRM) sirve como interface entre el usuario y los nodos. En él se ejecutan los comandos que manejan el sistema: reserva de cubos, carga de programas en los procesadores reservados, liberación del cubo, etc...

Va acompañado por compiladores de C y FORTRAN con algunas funciones para paso de mensajes. El lenguaje que se ha usado para implementar los algoritmos ha sido C [107].

El iPSC/2 que se ha utilizado consta de cuatro nodos, cada uno con 8 Mb de memoria y un procesador escalar, y un HOST que es un PC.

Un estudio de las prestaciones del iPSC/2 se puede encontrar en [18].

Intel iPSC/860

Un iPSC/860 [109], al igual que el iPSC/2, tiene topología de hipercubo de dimensión hasta 7. En este caso el procesador de cada uno de los nodos es un i860. Cada nodo tiene una memoria local que puede variar en tamaño. También tiene un

DCM que se encarga de las comunicaciones pudiendo tratarse por tanto el sistema como completamente conectado.

También en este caso un SRM realiza las mismas funciones que en el iPSC/2, y se dispone de compiladores C y FORTRAN con las mismas características que en el iPSC/2 [109].

Se han utilizado dos iPSC/860, uno con 8 nodos y otro con 32, el primero con 16 Mb de memoria local en cada nodo y el segundo con 8 Mb, y en ambos casos el HOST es una estación SUN.

Estudios de las prestaciones de esta máquina se pueden encontrar en [40, 56, 62].

Intel Touchstone DELTA

En el Touchstone Delta [111] cada nodo tiene un procesador i860 (como el iPSC/860), pero el sistema está organizado como una malla de 16 por 32 procesadores. Cada procesador tiene asociado un MRC (Mesh Routing Chip) con lo que el sistema se puede ver también en este caso como completamente conectado. Los MRC se encargan de las comunicaciones enviando los datos primero en la fila de procesadores y una vez que se ha llegado a la columna destino moviendo los datos en esta columna.

También en este caso un SRM realiza las mismas funciones que en los iPSC, y se dispone de compiladores C y FORTRAN con las mismas características que en estos sistemas.

La potencia aritmética de un iPSC/860 y el Touchstone Delta es la misma, pero con el Touchstone Delta se pueden obtener mejores prestaciones debido a la mejora en las comunicaciones [62].

2.2 Comunicaciones en Multicomputadores

En esta sección estudiaremos los procedimientos básicos de comunicación que utilizamos en los algoritmos diseñados para Multicomputadores.

Las características principales de los procedimientos de comunicación que aquí se muestran y de su utilización en los programas realizados son las siguientes:

- Consideraremos que en cada procesador se ejecuta el mismo programa y que cada procesador tiene un identificador con lo que se podrá decidir, testeando el identificador del procesador, que diferentes procesadores ejecuten partes distintas del programa.

- En los programas, cuando se va a realizar una comunicación se realiza una llamada a una función de comunicación del tipo de las que estudiaremos en esta sección. Al trabajar los procesadores de forma asíncrona las llamadas a estas funciones de comunicación no las hacen todos los procesadores al mismo tiempo, e incluso puede que no todos los procesadores se vean envueltos en esa comunicación.

- Los procesadores comparten datos por medio de envío y recepción de mensajes. En los algoritmos que estudiaremos una petición de recepción de mensaje se mostrará con "recibir", y el envío de un mensaje se mostrará con "enviar". Estas primitivas ("recibir" y "enviar") tienen sus equivalentes en los sistemas utilizados, que son chan_in_message y chan_out_message en el PARSYS SN-1040 y crecv y csend en los iPSC y el Touchstone Delta.

- Cuando se realiza un envío se especifica el procesador al que se envía. En el PARSYS SN-1040 esto se realiza con un parámetro de chan_out_message que es el canal por el que se realiza la comunicación, y un canal une dos procesadores. En los demás Multicomputadores utilizados se especifica en csend el identificador del procesador destino. Después de que un procesador realice un envío el procesador puede seguir trabajando o esperar a que el procesador destino reciba los datos. En el PARSYS SN-1040 se espera a que los datos se reciban pero en los demás casos el procesador que envía puede seguir trabajando debido a que el DCM o el MRC se encargan de gestionar las comunicaciones. En los procedimientos que aquí estudiaremos suponemos que la comunicación es síncrona y que se realiza cuando el procesador origen y destino están preparados.

- Cuando se realiza una recepción se puede especificar o no cuál es el procesador origen. En en PARSYS SN-1040 se especifica el procesador origen al ser un parámetro de chan_in_message el canal por el que se realiza la comunicación. En los demás sistemas no se especifica el origen como un parámetro de crecv, pero se puede utilizar un parámetro de crecv que indica el tipo de comunicación para decir entre que dos procesadores se está realizando. En algunos casos no es necesario conocer el origen de la comunicación, pero en la mayoría de los casos sí, por lo que consideraremos que en una recepción se especifica el procesador origen.

- Cuando un procesador realiza una petición de recepción no continúa hasta que no ha recibido los datos que solicita. Esto se puede evitar en los iPSC y el Touchstone Delta utilizando procedimientos de comunicación asíncrona (isend e irecv), pero incluso si se realiza una recepción asíncrona, en los algoritmos que hemos implementado, cuando llega el momento de utilizar los datos que se han solicitado en dicha comunicación hay que esperar su recepción (por medio de una función como msgwait), por lo que supondremos que las comunicaciones son síncronas.

- En todos los procedimientos de comunicación que estudiaremos cada procesador conocerá el procesador o procesadores origen de la comunicación y los destinos, pues estos datos dependerán de por donde vaya ejecutándose el programa. De este modo, cada procesador determina por medio de su identificador el trabajo que tiene que realizar en la función de comunicación.

- Una comunicación supondremos que se realiza entre dos procesadores adyacentes según la topología que tengamos. En el PARSYS SN-1040 habrá que utilizar la topología determinada por los canales (físicos o lógicos) del sistema, y en los demás casos se trata de una topología lógica pues, aunque la topología física del sistema sea de hipercubo o malla, en estos casos al ser posible la comunicación directa entre cualesquiera dos procesadores se puede considerar una red de interconexión completa.

- Aunque en algunos sistemas un procesador puede enviar y/o recibir por varios enlaces a la vez y estos envíos y/o recepciones se pueden realizar en los dos sentidos del canal (MLA: Multiple Link Assumption), en nuestro caso supondremos que en un momento dado un procesador sólo puede enviar o recibir y que la comunicación se hace por un solo enlace, y que por un enlace dado en un momento determinado los datos circulan en una única dirección (SLA: Single Link Assumption). Esto hará que los tiempos teóricos de comunicación que se obtengan sean cotas superiores de los que se obtendrían teniendo en cuenta las características del sistema. De cualquier modo, esto no es muy importante en los algoritmos implementados pues para poder obtener buenas prestaciones es necesario que el tiempo aritmético sea de mayor orden que el de comunicaciones y el considerar SLA o MLA no modifica el orden de las comunicaciones en los procedimientos de comunicación diseñados.

Estudiaremos los procedimientos básicos de comunicación que utilizamos en los algoritmos y que son:

1. En anillo e hipercubo:

   - difusión simple (envío de un mismo dato desde un procesador a todos los demás),
   - acumulación simple (dual de la difusión simple y que consiste en recibir en un único nodo información de todos los demás),
   - difusión múltiple (consiste en una difusión simple desde cada uno de los nodos del sistema),
   - y comunicación entre procesadores adyacentes en un anillo (en el caso del hipercubo, de un anillo inmerso en el hipercubo).

2. En malla:

   - difusión simple desde el procesador P_{00} (nos referiremos al procesador en la fila i, columna j, como P_{ij}, y numeraremos las filas y columnas empezando por 0) a todos los demás procesadores en el sistema,
   - acumulación simple de todos los procesadores sobre el procesador P_{00},
   - difusión desde los procesadores en la diagonal o en la antidiagonal principal del sistema (P_{ii} o $P_{i,r-1-i}$, con $i = 0, 1, ..., r-1$ y $r^2 = p$ siendo p el número de procesadores en el sistema) a los demás procesadores en la misma fila y columna de procesadores (la llamaremos difusión múltiple restringida),
   - y comunicación entre procesadores adyacentes en el array lineal formado por cada fila o columna de procesadores.

Con los algoritmos básicos de comunicación que aquí estudiamos se obtienen resultados satisfactorios en cuanto a tiempo de ejecución. Hay que tener en cuenta que los tiempos de ejecución teóricos que se obtendrán serán cotas superiores de los tiempos reales debido a las consideraciones que hemos hecho, y además estos procedimientos de comunicación básicos se usarán siempre, en nuestras implementaciones, dentro de programas con partes de computación aritmética. Al trabajar el sistema de modo asíncrono, habrá solapamiento entre las partes de comunicación y las de cómputo aritmético de los programas, por lo que para determinar las prestaciones reales de los algoritmos que se estudien será necesario realizar, además del estudio teórico, un estudio experimental.

Más referencias sobre funciones de comunicación pueden encontrarse para todas las topologías en [13], para la topología de hipercubo en [12, 60, 61], y para la de malla en [8, 9].

2.2.1 Anillo

Supongamos que tenemos p procesadores numerados $0, 1, 2, \ldots, p-1$ y p enlaces entre los procesadores $(i, (i+1)\ mod\ p)$, con $i = 0, 1, \ldots, p-1$. Cuando la comunicación por los enlaces es posible en un único sentido (de i a $(i+1)\ mod\ p$) decimos que tenemos un anillo unidireccional (figura 2.2.1), y cuando la comunicación es posible en los dos sentidos (de i a $(i+1)\ mod\ p$ y de $(i+1)\ mod\ p$ a i) tenemos un anillo bidireccional (figura 2.2.2).

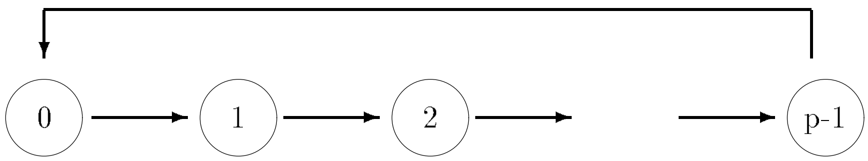


Figura 2.2.1: Anillo unidireccional.

El diámetro de un anillo unidireccional es $p-1$, pues para enviar un dato del procesador i al $(i-1)\ mod\ p$ hay que recorrer un camino con $p-1$ enlaces, sin embargo, el diámetro de un anillo bidireccional es $\lfloor \frac{p}{2} \rfloor$. Esto da idea de que las comunicaciones serán menos costosas en un anillo bidireccional.

Difusión simple

Supogamos que se quiere enviar un paquete de N datos desde el procesador 0 a todos los demás procesadores. En el caso de un anillo unidireccional, como el diámetro es

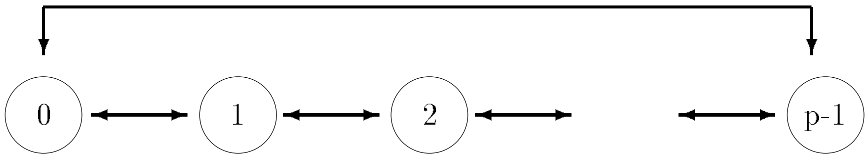


Figura 2.2.2: Anillo bidireccional.

$p-1$, los datos tendrán que pasar por $p-1$ enlaces hasta llegar al procesador $p-1$, y tendrán que ser enviados sucesivamente por los procesadores $0, 1, 2, \ldots, p-2$; por tanto, tendremos $p-1$ inicializaciones de transmisión y $p-1$ envíos de los datos. El coste de la comunicación será

$$(\beta + \tau N)(p-1) \quad . \tag{2.2.1}$$

Hay que tener en cuenta que, cuando este algoritmo se utilice dentro de otro programa, mientras unos procesadores están realizando comunicaciones otros pueden estar realizando otros trabajos.

El código del algoritmo es:

Algoritmo 2.2.1 *Difusión simple en anillo unidireccional.*

EN PARALELO para $r = 0, 1, ..., p-1$ en P_r:
  SI $r = origen$
    enviar datos a $(r+1)\ mod\ p$
  EN OTRO CASO SI $r \neq (origen - 1)\ mod\ p$
    recibir datos de $(r-1)\ mod\ p$
    enviar datos a $(r+1)\ mod\ p$
  EN OTRO CASO
    recibir datos de $(r-1)\ mod\ p$
  FINSI

Para poder entender este algoritmo y los sucesivos en este capítulo hay que tener en cuenta las consideraciones hechas anteriormente:

- Todos los procesadores tienen el mismo código en el procedimiento de comunicación, y la parte que cada procesador ejecuta se decide testeando su identificador (r en el algoritmo 2.2.1).
- Todos los procesadores saben, porque depende del paso de la ejecución del programa en que se encuentren, cuál es el *origen* de la comunicación. De este modo se pueden determinar distintos conjuntos de procesadores que hacen distintas

operaciones: el procesador *origen*, el último al que llegan los datos y que sólo los recibe y no los manda a ningún otro procesador (procesador $(origen-1)\ mod\ p$), y los procesadores intermedios que reciben y envían los datos ($r \neq origen$ y $r \neq (origen-1)\ mod\ p$).

- Los envíos y recepciones de datos se especifican con las palabras "enviar" y "recibir", especificando en ambos casos los procesadores destino y origen, respectivamente.

En el caso de anillo bidireccional el procesador *origen* enviará a $(origen+1)\ mod\ p$ y $(origen-1)\ mod\ p$, los procesadores $(origen+\lfloor\frac{p}{2}\rfloor)\ mod\ p$ y $(origen+\lfloor\frac{p}{2}\rfloor+1)\ mod\ p$ recibirán de $(origen+\lfloor\frac{p}{2}\rfloor-1)\ mod\ p$ y $(origen+\lfloor\frac{p}{2}\rfloor+2)\ mod\ p$ respectivamente, y los demás procesadores reciben y envían según la figura 2.2.3 y el algoritmo 2.2.2.

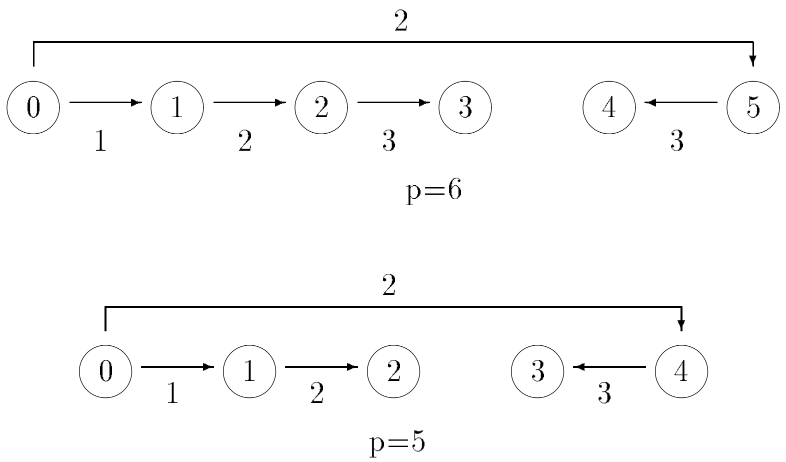


Figura 2.2.3: Difusión simple en anillo bidireccional.

Algoritmo 2.2.2 *Difusión simple en anillo bidirecional.*
EN PARALELO para $r=0,1,...,p-1$ en P_r:
SI $r = origen$
enviar datos a $(r+1)\ mod\ p$
enviar datos a $(r-1)\ mod\ p$
EN OTRO CASO SI r entre $(origen+1)\ mod\ p$ y $(origen+\lfloor\frac{p}{2}\rfloor-1)\ mod\ p$
recibir datos de $(r-1)\ mod\ p$
enviar datos a $(r+1)\ mod\ p$
EN OTRO CASO SI r entre $(origen+\lfloor\frac{p}{2}\rfloor+2)\ mod\ p$ y $(origen-1)\ mod\ p$

recibir datos de $(r+1)\ mod\ p$
enviar datos a $(r-1)\ mod\ p$
EN OTRO CASO SI $r = origen + \lfloor \frac{p}{2} \rfloor$
recibir datos de $(r-1)\ mod\ p$
EN OTRO CASO
recibir datos de $(r+1)\ mod\ p$
FINSI

Este algoritmo tiene un tiempo de ejecución

$$(\beta + \tau N)\left\lceil \frac{p}{2} \right\rceil \quad . \tag{2.2.2}$$

Acumulación simple

Tendremos un procesador destino al que todos los demás envían un paquete de N datos, y hay que realizar algún tipo de operación entre los pN datos (los datos en el procesador destino también se combinan), pudiendo realizarse esta combinación durante el camino de llegada al destino o una vez han llegado todos los datos. En el caso del anillo unidireccional el tiempo de comunicación es igual que en la difusión simple, pues los datos que se envían desde $(destino+1)\ mod\ p$ a $destino$ deben pasar por $p-1$ enlaces y ser enviados por $p-1$ procesadores, es decir:

$$(\beta + \tau N)(p-1) \quad . \tag{2.2.3}$$

En el caso bidireccional el coste de las comunicaciones también será

$$(\beta + \tau N)\left\lceil \frac{p}{2} \right\rceil \quad . \tag{2.2.4}$$

Difusión múltiple

En este caso se trata de transferir un paquete de N datos desde cada uno de los procesadores del sistema a todos los demás, de modo que al final todos dispongan de los pN datos.

En un anillo unidireccional cada paquete de N datos tendrá que cruzar $p-1$ enlaces y ser enviado por $p-1$ procesadores, por lo que el coste de las comunicaciones resulta ser:

$$(\beta + \tau N)(p-1) \quad . \tag{2.2.5}$$

El algoritmo correspondiente es el siguiente.

Algoritmo 2.2.3 *Difusión múltiple en anillo unidireccional.*
EN PARALELO para $r = 0, 1, ..., p-1$ en P_r:
PARA $i = 1, 2, ..., p-1$
SI r es par
recibir paquete $(r-i)$ *mod* p de $(r-1)$ *mod* p
enviar paquete $(r-i+1)$ *mod* p a $(r+1)$ *mod* p
EN OTRO CASO
enviar paquete $(r-i+1)$ *mod* p a $(r+1)$ *mod* p
recibir paquete $(r-i)$ *mod* p de $(r-1)$ *mod* p
FINSI
FINPARA

Como las transmisiones son simples (SLA), al no poderse hacer la transmisión y recepción a la vez, el coste es

$$2(\beta + \tau N)(p-1) \quad . \tag{2.2.6}$$

Cuando el anillo es bidireccional el coste es el mismo si las comunicaciones son simples.

Comunicación entre procesadores adyacentes en un anillo

Esta comunicación es muy simple pero la tratamos por ser una de las que aparecen en los algoritmos de los capítulos siguientes. Se trata de enviar un paquete de N datos desde cada nodo i a su nodo vecino $(i+1)$ *mod* p.

Al ser las comunicaciones simples el coste será

$$2(\beta + \tau N) \quad . \tag{2.2.7}$$

Algoritmo 2.2.4 *Comunicación entre procesadores adyacentes en anillo.*
EN PARALELO para $r = 0, 1, ..., p-1$ en P_r:
SI r es par
enviar datos a $(r+1)$ *mod* p
recibir datos de $(r-1)$ *mod* p
EN OTRO CASO
recibir datos de $(r-1)$ *mod* p
enviar datos a $(r+1)$ *mod* p
FINSI

2.2.2 Hipercubo

Un hipercubo cero dimensional consta de un único nodo numerado con 0. Un hipercubo de dimensión k consta de $p = 2^k$ nodos y se forma a partir de dos hipercubos de dimensión $k-1$, conectando mediante un enlace bidireccional los nodos con la misma numeración en los dos hipercubos. Los nodos de uno de los dos hipercubos se renumeran con el número que tenían, en el hipercubo de dimensión $k-1$, mas 2^{k-1} (figura 2.2.4).

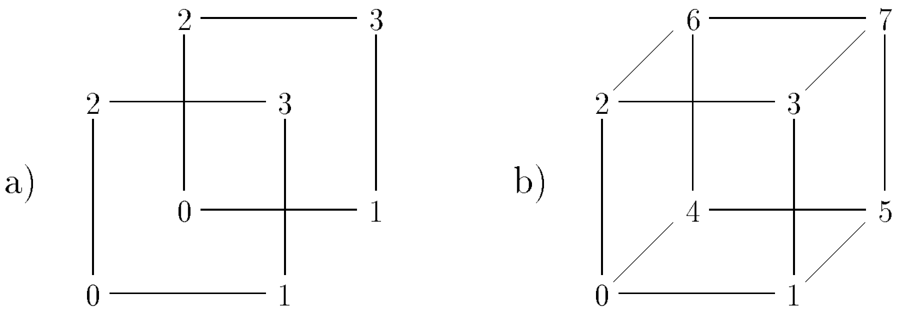


Figura 2.2.4: Construcción de un hipercubo tridimensional.

El diámetro de un hipercubo de p nodos es $log_2\ p$, por lo que las comunicaciones pueden ser menos costosas que en otras topologías (en concreto serán menos costosas que en anillo). Además, muchas topologías se pueden mapear sobre el hipercubo, en particular se puede mapear un anillo conectando los nodos utilizando el código de Gray:

$$\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \\ 0 & 1 & 3 & 2 & 6 & 7 & 5 & 4 \end{array} \tag{2.2.8}$$

obteniéndose un anillo inmerso en el hipercubo tal como se muestra en la figura 2.2.5.

Difusión simple

Supongamos que se quiere enviar un paquete de N datos desde el procesador cero a todos los demás procesadores. Como el diámetro es $log_2\ p$ necesitaremos recorrer un camino de longitud $log_2\ p$. En la figura 2.2.6 se muestra, en un hipercubo de dimensión 3, un método para realizar la difusión en ese orden de tiempo. El paquete lo envía 000 a 001, 010 y 100, en este orden; 001 lo envía a 011 y 101, en este orden.

De este modo, su coste es:

$$(\beta + \tau N) log_2\ p \quad . \tag{2.2.9}$$

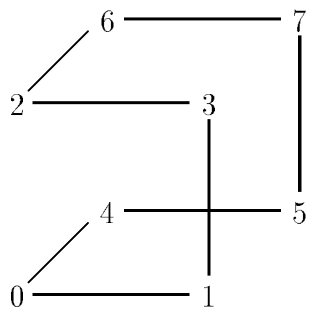


Figura 2.2.5: Anillo inmerso en un hipercubo.

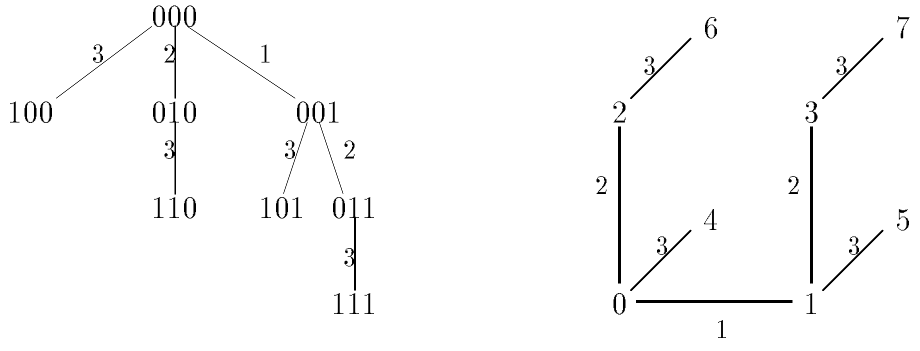


Figura 2.2.6: Difusión simple desde el nodo 0 en un hipercubo de dimensión 3.

Un algoritmo podría ser el siguiente:

Algoritmo 2.2.5 *Difusión simple en hipercubo.*

EN PARALELO para $r = 0, 1, ..., p - 1$ en P_r:
  SI $r = 0$
    PARA $i = 0, 1, ..., log_2\, p - 1$
      enviar datos al nodo 2^i
    FINPARA
  EN OTRO CASO
    recibir datos
    $s = min\{i/2^i > r\}$
    PARA $i = s, s + 1, ...log_2\, p - 1$
      enviar datos al nodo $r + 2^i$
    FINPARA
  FINSI

Aunque en los programas sólo haremos difusiones simples con origen el nodo cero, si el origen fuera otro nodo se modificaría el árbol de la figura 2.2.6 haciéndose el **or exclusivo** del nuevo origen con cada nodo del árbol. La figura 2.2.7 corresponde, por ejemplo, al caso de $origen = 101$.

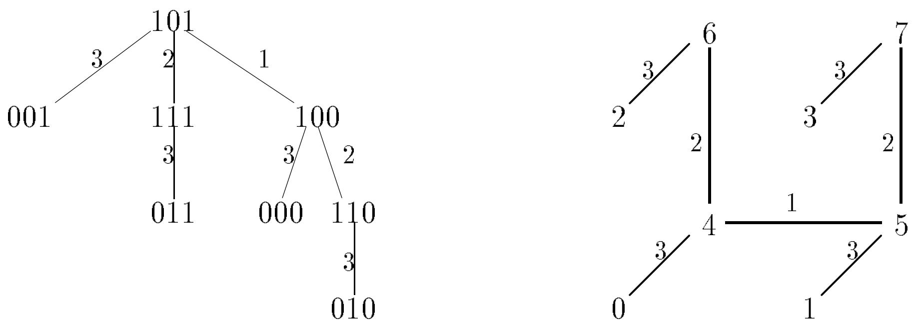


Figura 2.2.7: Difusión simple desde el nodo 5 en un hipercubo de dimensión 3.

Acumulación simple

En este caso el proceso es el dual de la difusión simple y se representa en la figura 2.2.8. Se puede observar que es igual a la figura 2.2.6 pero cambiando los instantes en que se realizan las comunicaciones y siendo en este caso las comunicaciones en sentido ascendente.

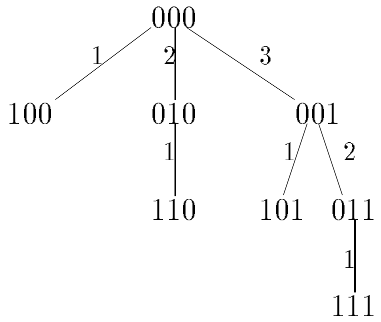


Figura 2.2.8: Acumulación simple en un hipercubo de dimensión 3.

Por tanto el coste de las comunicaciones es también

$$(\beta + \tau N) log_2\ p \quad . \tag{2.2.10}$$

El algoritmo resultante es el siguiente:

Algoritmo 2.2.6 *Acumulación simple en hipercubo.*

EN PARALELO para $r = 0, 1, ..., p-1$ en P_r:
  SI $r = 0$
    PARA $i = log_2\ p - 1, ..., 0$
      recibir datos de 2^i
    FINPARA
  EN OTRO CASO
    $s = min\{i/2^i > r\}$
    PARA $i = log_2\ p - 1, ..., s$
      recibir datos del nodo $r + 2^i$
    FINPARA
    enviar datos al nodo $r - 2^{s-1}$
  FINSI

Difusión múltiple

Se procederá intercambiando un paquete de N datos entre procesadores cuyo número difiera en 2^0, a continuación intercambiando dos paquetes entre procesadores que difieran en 2^1, y así sucesivamente $log_2\ p$ veces (figura 2.2.9).

En cada paso cada nodo recibe y envía. Hay $2 log_2\ p$ inicializaciones de comunicaciones. Se envían $1+2+4+\ldots+\frac{p}{2}$ paquetes de N datos y se reciben la misma cantidad, por lo que en cada nodo se envían $p-1$ paquetes y se reciben la misma cantidad, con

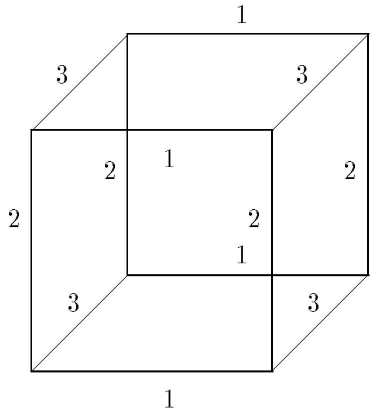


Figura 2.2.9: Difusión múltiple en un hipercubo de dimensión 3.

lo que el coste de envío y recepción de datos sería $2(p-1)N$. Por tanto, el coste de las comunicaciones es:

$$2\beta log_2\, p + 2(p-1)\tau N \qquad . \tag{2.2.11}$$

Se comprueba que se obtiene mejora respecto al coste en anillo en el término que afecta a β.

El algoritmo correspondiente viene dado por:

Algoritmo 2.2.7 *Difusión múltiple en hipercubo.*

EN PARALELO para $r = 0, 1, ..., p-1$ en P_r:
  $pos = rN$
  $nodo = r$
  PARA $i = 0, 1, ..., log_2\, p - 1$
    $bit = nodo\ mod\ 2$
    SI $bit = 0$
      enviar datos pos a $pos + 2^i N - 1$ al nodo $r + 2^i$
      recibir datos $pos + 2^i N$ a $pos + 2^{i+1}N - 1$ del nodo $r + 2^i$
    EN OTRO CASO
      recibir datos $pos - 2^i N$ a $pos - 1$ del nodo $r - 2^i$
      enviar datos pos a $pos + 2^i N - 1$ al nodo $r - 2^i$
      $pos = pos - 2^i N$
    FINSI
    $nodo = nodo\ div\ 2$
  FINPARA

Donde suponemos que los datos se almacenan en cada procesador en un buffer de p bloques de datos de tamaño N, teniendo por tanto pN datos en el buffer, y que los datos a enviar por el procesador P_i están inicialmente entre las posiciones iN y $(i+1)N-1$ de su buffer local, y que al final de la difusión múltiple cada procesador debe contener los p bloques de datos en su buffer local, estando los datos difundidos desde el procesador P_i entre las posiciones iN y $(i+1)N-1$ de cada buffer local. La variable *pos* indica la posición de inicio de los datos a enviar en cada paso del algoritmo.

Comunicación entre procesadores adyacentes en un anillo

El algoritmo será igual al algoritmo 2.2.4, pero consideramos el anillo inmerso en el hipercubo, por lo que en el procesador r trabajaremos con su numeración en el anillo inmerso que es $ginv(r)$, y se enviarán datos a $gray((ginv(r)+1)\ mod\ p)$ y se recibirán de $gray((ginv(r)-1)\ mod\ p)$. En la figura 2.2.10 tenemos un hipercubo de dimensión tres con la numeración de los nodos en el hipercubo y la que tienen en el anillo inmerso y que se obtiene con la función *ginv*. Se observa, con el ejemplo de los nodos 1 y 3, cómo las fórmulas dadas determinan al nodo (con numeración en el hipercubo) al que se envían los datos y del que se reciben.

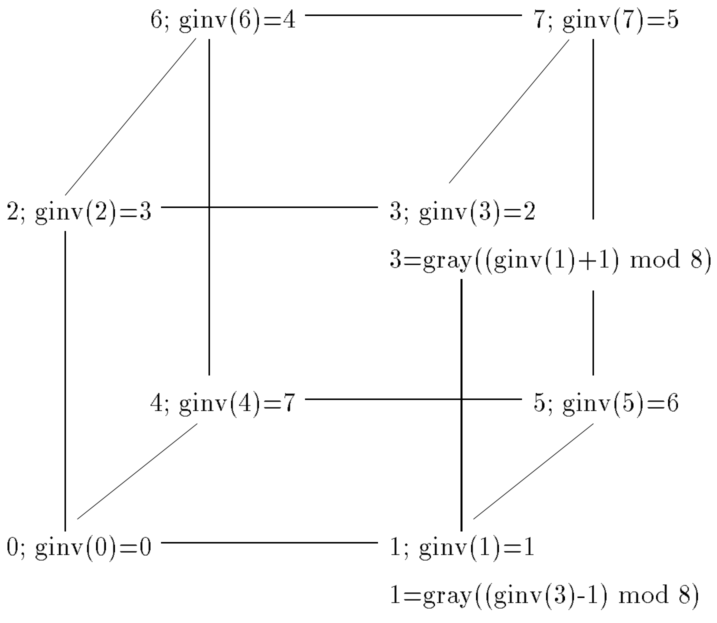


Figura 2.2.10: Uso de *gray* y *ginv* para la obtención del anillo inmerso en un hipercubo.

Los iPSC tienen las funciones *ginv* y *gray* [108], siendo estas funciones la que proporciona el código de Gray (*gray*) y su inversa (*ginv*).

El algoritmo 2.2.4 se modifica para dar lugar al siguiente algoritmo:

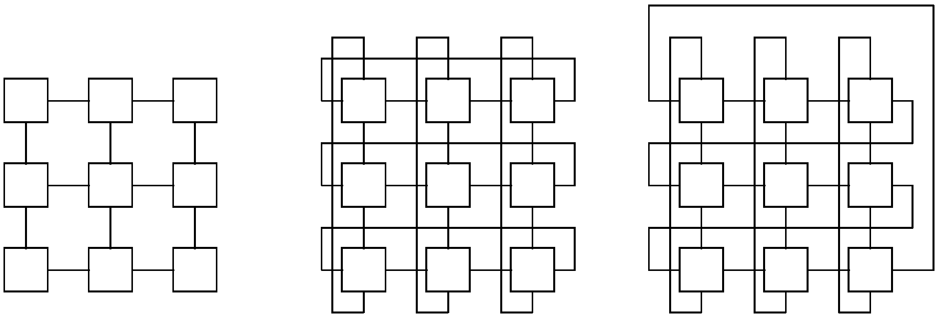
Algoritmo 2.2.8 *Comunicación entre procesadores vecinos en un anillo inmerso en el hipercubo.*

EN PARALELO para $r = 0, 1, ..., p-1$ en P_r:
  SI $ginv(r)$ es par
    enviar datos a $gray((ginv(r)+1)\ mod\ p)$
    recibir datos de $gray((ginv(r)-1)\ mod\ p)$
  EN OTRO CASO
    recibir datos de $gray((ginv(r)-1)\ mod\ p)$
    enviar datos a $gray((ginv(r)+1)\ mod\ p)$
  FINSI

El coste de comunicación será el mismo de la expresión 2.2.7.

2.2.3 Malla

En la figura 2.2.11 se muestran distintos tipos de mallas bidimensionales. La malla que se ha usado en este trabajo es una malla abierta cuadrada, aunque el esquema de almacenamiento de los datos que se utiliza puede servir para otros tipos de malla dependiendo de la ordenación que se utilice para diseñar el método de Jacobi.



a) b) c)

Figura 2.2.11: Mallas bidimensionales: a) malla abierta, b) toro, c) malla cerrada.

En este trabajo se presentarán esquemas de almacenamiento para anillo y para malla, y se estudiarán en anillo, hipercubo y malla. Los algoritmos para anillo e hipercubo difieren sólo en las comunicaciones al hacerse éstas teniendo en cuenta la topología en la que se está trabajando, pero usarán los mismos esquemas de almacenamiento al trabajarse siempre con un anillo de procesadores (en el caso del hipercubo con un anillo inmerso). En la topología de malla se usa otro esquema de almacenamiento distinto, lo que hace que las funciones de comunicación que se utilizan no sean en algunos casos las típicas, por lo que para esta topología analizaremos los procedimientos de comunicación que se utilizarán en los algoritmos sobre malla.

Difusión simple

Estudiaremos la difusión desde el procesador P_{00} a todos los demás procesadores de la malla. El esquema es bien simple y consiste en enviar los datos por la primera fila de procesadores y después hacerlos circular por cada columna de procesadores hasta llegar a la última fila.

Un algoritmo podría ser:

Algoritmo 2.2.9 *Difusión simple en malla abierta.*

```
EN PARALELO para i = 0,1,...,r-1,   j = 0,1,...,r-1,
con r^2 = p, en P_ij:
   SI i = 0 y j = 0
      enviar datos a P_0,1
      enviar datos a P_1,0
   EN OTRO CASO SI i = 0 y j ≠ r-1
      recibir datos de P_0,j-1
      enviar datos a P_0,j+1
      enviar datos a P_1,j
   EN OTRO CASO SI i = 0
      recibir datos de P_0,r-2
      enviar datos a P_1,r-1
   EN OTRO CASO SI i ≠ r-1
      recibir datos de P_i-1,j
      enviar datos a P_i+1,j
   EN OTRO CASO
      recibir datos de P_r-2,j
   FINSI
```

El coste de este procedimiento es

$$2\,(\sqrt{p}-1)\,(\beta+\tau N) \qquad , \tag{2.2.12}$$

y corresponde al tiempo que tardan los datos en llegar desde el procesador P_{00} al $P_{r-1,r-1}$, para lo que atraviesan $2\sqrt{p}-2$ enlaces.

Acumulación simple

La acumulación simple sobre el procesador P_{00} sigue el esquema inverso a la difusión simple (algoritmo 2.2.9).

Algoritmo 2.2.10 *Acumulación simple en malla abierta.*

EN PARALELO para $i = 0, 1, ..., r-1, \quad j = 0, 1, ..., r-1$,
con $r^2 = p$, en P_{ij}:
  SI $i = 0$ y $j = 0$
    recibir datos de P_{10}
    recibir datos de P_{01}
  EN OTRO CASO SI $i = 0$ y $j \neq r-1$
    recibir datos de P_{1j}
    recibir datos de $P_{0,j+1}$
    enviar datos a $P_{0,j-1}$
  EN OTRO CASO SI $i = 0$
    recibir datos de $P_{1,r-1}$
    enviar datos a $P_{0,r-2}$
  EN OTRO CASO SI $i \neq r-1$
    recibir datos de $P_{i+1,j}$
    enviar datos a $P_{i-1,j}$
  EN OTRO CASO
    enviar datos a $P_{r-2,j}$
  FINSI

El coste en este caso es el mismo de (2.2.12), que corresponde al tiempo que tarda en llegar a P_{00} los datos enviados desde $P_{r-1,r-1}$.

Difusión múltiple restringida

Como ya hemos dicho, estudiaremos la difusión de datos desde los procesadores en la diagonal principal (P_{ii}) y en la antidiagonal principal ($P_{i,r-1-i}$) a los procesadores en la misma fila y columna de procesadores.

Un esquema en el caso de difundir desde la diagonal principal podría ser:

Algoritmo 2.2.11 *Difusión desde procesadores en la diagonal principal a los demás procesadores en la misma fila y columna, en una malla abierta.*

EN PARALELO para $i = 0, 1, ..., r-1, \quad j = 0, 1, ..., r-1$,
con $r^2 = p$, en P_{ij}:
  SI $i = 0$ y $j = 0$
    enviar datos a P_{01}
    enviar datos a P_{10}

EN OTRO CASO SI $i = r-1$ y $j = r-1$
    enviar datos a $P_{r-1,r-2}$
    enviar datos a $P_{r-2,r-1}$
EN OTRO CASO SI $i = j$
    SI $i < \frac{r}{2}$
        enviar datos a $P_{i,j+1}$
        enviar datos a $P_{i-1,j}$
        enviar datos a $P_{i,j-1}$
        enviar datos a $P_{i+1,j}$
    EN OTRO CASO
        enviar datos a $P_{i,j-1}$
        enviar datos a $P_{i+1,j}$
        enviar datos a $P_{i,j+1}$
        enviar datos a $P_{i-1,j}$
    FINSI
EN OTRO CASO SI $i < \frac{r}{2}$
    SI $i < j$
        recibir datos de $P_{i,j-1}$
        SI $j \neq r-1$
            enviar datos recibidos a $P_{i,j+1}$
        FINSI
        recibir datos de $P_{i+1,j}$
        SI $i \neq 0$
            enviar datos recibidos a $P_{i-1,j}$
        FINSI
    EN OTRO CASO
        recibir datos de $P_{i-1,j}$
        enviar datos recibidos a $P_{i+1,j}$
        recibir datos de $P_{i,j+1}$
        SI $j \neq 0$
            enviar datos recibidos a $P_{i,j-1}$
        FINSI
    FINSI
EN OTRO CASO
    SI $i < j$
        recibir datos de $P_{i+1,j}$
        enviar datos recibidos a $P_{i-1,j}$
        recibir datos de $P_{i,j-1}$
        SI $j \neq r-1$
            enviar datos recibidos a $P_{i,j+1}$
        FINSI

```
    EN OTRO CASO
        recibir datos de P_{i,j+1}
        SI j ≠ 0
            enviar datos recibidos a P_{i,j-1}
        FINSI
        recibir datos de P_{i-1,j}
        SI i ≠ r - 1
            enviar datos recibidos a P_{i+1,j}
        FINSI
    FINSI
FINSI
```

En la figura 2.2.12 se muestra cómo trabaja esta difusión en una malla abierta cuadrada con 16 procesadores. En esta figura un número al lado de una flecha indica el orden en que un procesador inicia la comunicación correspondiente.

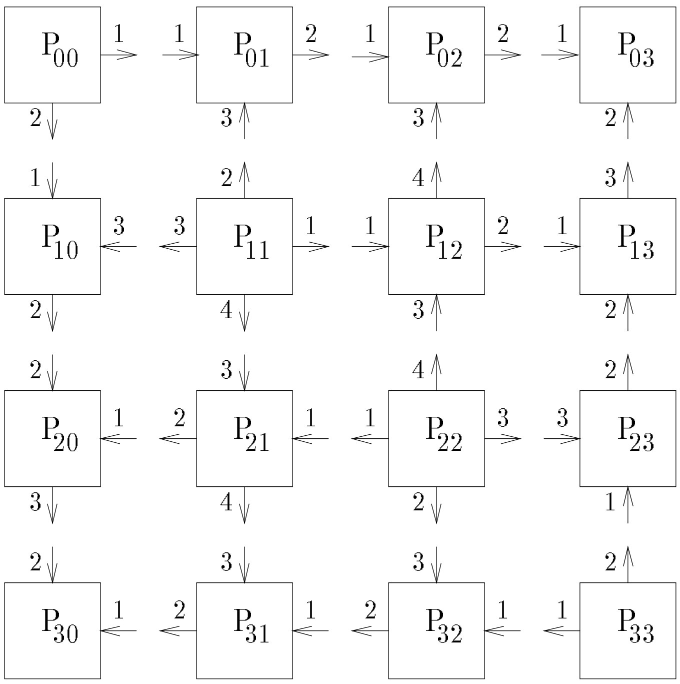


Figura 2.2.12: Difusión desde procesadores en la diagonal principal.

El coste de las comunicaciones con esta difusión es

$$\sqrt{p}\,(\beta + \tau N) \qquad . \tag{2.2.13}$$

Y en el caso de difundir desde la antidiagonal principal:

Algoritmo 2.2.12 *Difusión desde procesadores en la antidiagonal principal a los demás procesadores en la misma fila y columna, en una malla abierta.*

EN PARALELO para $i = 0, 1, ..., r-1$, $\quad j = 0, 1, ..., r-1$,
con $r^2 = p$, en P_{ij}:
  SI $j = 0$ y $i = r-1$
    enviar datos a $P_{r-1,1}$
    enviar datos a $P_{r-2,0}$
  EN OTRO CASO SI $i = 0$ y $j = r-1$
    enviar datos a $P_{0,r-2}$
    enviar datos a $P_{1,r-1}$
  EN OTRO CASO SI $i = r-1-j$
    SI $i < \frac{r}{2}$
      enviar datos a $P_{i,j-1}$
      enviar datos a $P_{i+1,j}$
      enviar datos a $P_{i,j+1}$
      enviar datos a $P_{i-1,j}$
    EN OTRO CASO
      enviar datos a $P_{i,j+1}$
      enviar datos a $P_{i-1,j}$
      enviar datos a $P_{i,j-1}$
      enviar datos a $P_{i+1,j}$
    FINSI
  EN OTRO CASO SI $i < \frac{r}{2}$
    SI $i < r-1-j$
      recibir datos de $P_{i,j+1}$
      SI $j \neq 0$
        enviar datos recibidos a $P_{i,j-1}$
      FINSI
      recibir datos de $P_{i+1,j}$
      SI $i \neq 0$
        enviar datos recibidos a $P_{i-1,j}$
      FINSI
    EN OTRO CASO
      recibir datos de $P_{i-1,j}$
      enviar datos recibidos a $P_{i+1,j}$
      recibir datos de $P_{i,j-1}$

```
            SI $j \neq r-1$
                enviar datos recibidos a $P_{i,j+1}$
            FINSI
        FINSI
    EN OTRO CASO
        SI $i < r-1-j$
            recibir datos de $P_{i+1,j}$
            enviar datos recibidos a $P_{i-1,j}$
            recibir datos de $P_{i,j+1}$
            SI $j \neq 0$
                enviar datos recibidos a $P_{i,j-1}$
            FINSI
        EN OTRO CASO
            recibir datos de $P_{i,j-1}$
            SI $j \neq r-1$
                enviar datos recibidos a $P_{i,j+1}$
            FINSI
            recibir datos de $P_{i-1,j}$
            SI $i \neq r-1$
                enviar datos recibidos a $P_{i+1,j}$
            FINSI
        FINSI
    FINSI
```

En la figura 2.2.13 se muestra cómo trabaja esta difusión en una malla abierta cuadrada con 16 procesadores. En esta figura un número al lado de una flecha indica el orden en que un procesador inicia la comunicación correspondiente.

En este caso el coste de las comunicaciones es también el dado por la fórmula (2.2.13).

Comunicación entre procesadores adyacentes

En este caso realizamos una comunicación de paquetes de N datos entre procesadores adyacentes en cada array lineal de procesadores formado por los procesadores en una fila o columna, enviando y recibiendo cada procesador datos de sus procesadores vecinos en el array lineal, por lo que los procesadores extremos se comunican con un único procesador y los internos con dos.

El seudocódigo de la comunicación en una fila de procesadores puede ser:

Algoritmo 2.2.13 *Comunicación entre procesadores adyacentes por filas en una malla abierta.*

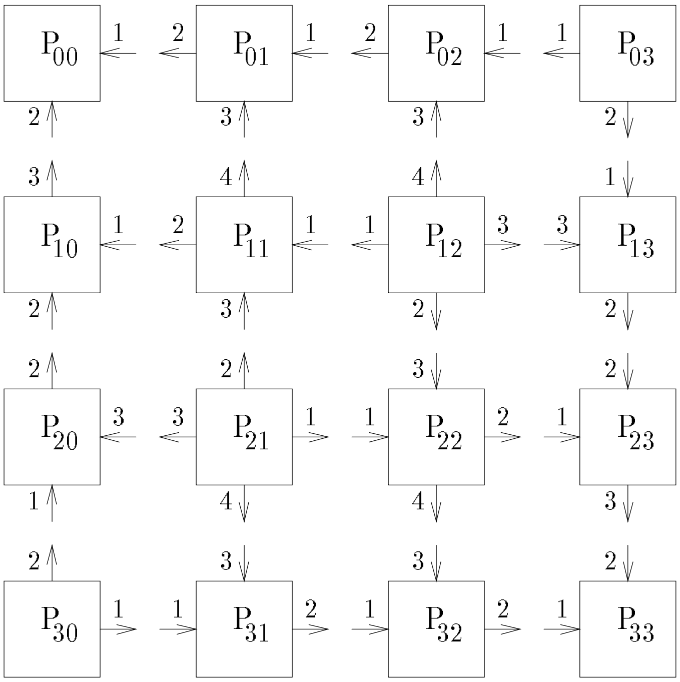


Figura 2.2.13: Difusión desde procesadores en la antidiagonal principal.

EN PARALELO para $i = 0, 1, ..., r-1, \quad j = 0, 1, ..., r-1$,
con $r^2 = p$, en P_{ij}:
SI $j = 0$
enviar datos a P_{i1}
recibir datos de P_{i1}
EN OTRO CASO SI $j = r-1$
recibir datos de $P_{i,r-2}$
enviar datos a $P_{i,r-2}$
EN OTRO CASO
SI $j \ mod \ 2 = 0$
enviar datos a $P_{i,j+1}$
recibir datos de $P_{i,j-1}$
recibir datos de $P_{i,j+1}$
enviar datos a $P_{i,j-1}$
EN OTRO CASO
recibir datos de $P_{i,j-1}$
enviar datos a $P_{i,j+1}$
enviar datos de $P_{i,j-1}$
recibir datos de $P_{i,j+1}$
FINSI
FINSI

El coste de las comunicaciones es:

$$4\beta + 4N\tau \quad . \tag{2.2.14}$$

2.3 Conclusiones

En este capítulo hemos hecho un análisis general de los sistemas multiprocesadores que se han utilizado para la obtención de resultados experimentales. Se han analizado las características principales de los sistemas multiprocesadores en general, tanto de memoria compartida como memoria distribuida, y de los equipos comerciales sobre los que se han hecho las implementaciones. En el caso de los multicomputadores (memoria distribuida) se han enumerado las características que les suponemos a la hora del diseño y el análisis de los algoritmos que estudiamos.

Capítulo 3

METODOS DE JACOBI EN MULTIPROCESADORES

En este capítulo analizaremos distintas técnicas de paralelización de métodos de Jacobi en Multiprocesadores, cuando se usan algoritmos basados en BLAS 1 y algoritmos por bloques basados en BLAS 3, y compararemos los resultados obtenidos con los que se obtienen usando LAPACK. Previamente estudiaremos algunas opciones de compilación del FX/C [103], entre ellas las que nos permiten paralelizar y vectorizar de manera automática (sección 1).

La técnica más usada en Multiprocesadores con Memoria Compartida para explotar el paralelismo del sistema en la resolución de problemas de Algebra Lineal suele ser la de diseñar algoritmos que hagan llamadas a rutinas básicas optimizadas para la máquina en que se esté trabajando (rutinas tipo BLAS [36, 37]) de manera que si estas rutinas están optimizadas para esta máquina explotarán todas sus características de procesamiento vectorial y paralelo obteniéndose programas eficientes y portables. Un problema con este tipo de aproximación es que en algunos casos las rutinas de BLAS no están lo suficientemente optimizadas para explotar el paralelismo del sistema (para usar eficientemente todos los procesadores de que se dispone), lo que produce una reducción drástica en las prestaciones al aumentar el número de procesadores; mientras que se dispone de una versión secuencial eficiente de BLAS, lo que ocasiona que pueda ser preferible diseñar algoritmos por bloques basándose en el uso de la versión secuencial de BLAS, tal como se hace en [31] con la multiplicación de matrices. De este modo, otra posibilidad de explotación del paralelismo en este tipo de sistemas consiste en la división del trabajo entre los distintos procesadores del sistema y la utilización en cada procesador de las rutinas de BLAS. Esta aproximación puede dar lugar a programas más eficientes, aunque esto será a costa de una menor portabilidad de los programas, al hacerse la división del trabajo entre los procesadores de manera distinta en distintas máquinas. De cualquier modo, esta reducción de la portabilidad no es muy importante pues ligeras modificaciones en los programas diseñados para una determinada máquina producirán de una manera muy sencilla programas para un nuevo Multiprocesador de similares características.

En este capítulo compararemos estas dos formas de paralelización en un Alliant FX/80, que es un Multiprocesador con 8 procesadores con Memoria Compartida a la que se accede por medio de un bus común.

3.1 El compilador FX/C

El compilador C del Alliant FX/80 (FX/C) permite diferentes opciones de optimización automática, entre ellas las de vectorización y paralelización automática. Una referencia completa se encuentra en [103].

La opción de optimización global (sin paralelizar ni vectorizar) se especifica con -Og en la línea de compilación. Esta opción realiza las siguientes optimizaciones: cálculo de constantes en tiempo de compilación, eliminación de expresiones redundantes, almacenamiento temporal en registros de variables referenciadas frecuentemente, eliminación de variables no referenciadas, eliminación de expresiones que dan lugar a una modificación constante en un bucle, reducción de algunas operaciones de multiplicación a sumas, extracción de los bucles de código que produce siempre el mismo resultado, secuenciamiento de instrucciones para aprovechar el pipeline, eliminación de ramas innecesarias, minimización de desplazamiento, y optimización de código eliminando tests y copias innecesarios y usando instrucciones más rápidas.

La opción de vectorización automática es -Ov. De esta manera se puede llegar a obtener una división por 4 en el tiempo de ejecución, cuando se usa un único procesador.

La opción de paralelización automática es -Oc. De esta manera el compilador intenta distribuir la ejecución de los bucles entre los procesadores. Se puede llegar a obtener que con 8 procesadores la ejecución sea 8 veces más rápida que con un procesador.

Estas opciones se pueden combinar, con lo que el uso de -Ogvc realizará una compilación global, con vectorización y paralelización automática.

Con paralelización y vectorización automática se pueden llegar a conseguir unos tiempos de ejecución de entre 30 y 32 veces menores que cuando sólo se usa optimización global. Pero para conseguir que el compilador optimice de manera adecuada llegando a reducciones considerables en el tiempo de ejecución es necesario preparar los programas de manera que sea posible la vectorización y paralelización automática de los bucles que aparecen en el programa, lo que no siempre es posible debido a dependencias de datos. Además, el paralelismo que obtiene el compilador lo extrae siempre de los bucles, mientras que nosotros estamos interesados en una paralelización lo más global posible del problema, puesto que con ella se consiguen mejores tiempos de ejecución. En este sentido, se puede utilizar una llamada al sistema **concurrent_call** que permite la ejecución concurrente de tantos procesos como procesadores estemos utilizando.

Esta aproximación será la que utilicemos para paralelizar nuestros programas. De este modo, nuestros programas estarán divididos en una serie de pasos haciéndose en cada uno de ellos una ejecución concurrente con tantos procesos como procesadores tengamos. Utilizaremos la llamada concurrent_call, estando pasos consecutivos del programa sincronizados pues no continuará la ejecución del programa global que hace la llamada a concurrent_call hasta que todos los procesadores no acaben de hacer el trabajo que se les ha encomendado. Con la llamada a concurrent_call se ponen en marcha los procesadores y se les pasa una función que tienen que ejecutar, trabajando cada procesador sobre las variables que se les pasan como parámetros, variables locales y variables globales. Como la matriz de datos estará en la memoria global, para obtener programas eficientes tendremos que determinar en cada paso de los algoritmos (en cada llamada a concurrent_call) zonas disjuntas de la matriz y asignar estas zonas disjuntas a procesadores distintos, entendiendo que lo que se hace no es asignar datos a procesadores (pues los datos están todos en la memoria global y todos los procesadores tienen acceso a ellos), sino que lo que se asigna a los procesadores es trabajo al determinar con qué zonas de la matriz tiene que trabajar cada uno de ellos.

3.2 Uso de BLAS 1

Como ya vimos en el capítulo 1, en los métodos de Jacobi, la actualización de la matriz A (QAQ^t) puede hacerse usando BLAS 1, obteniéndose de este modo programas más eficientes. Esta actualización de la matriz consiste en la aplicación de la rotación a las filas y columnas i y j de A, y esta aplicación se puede hacer usando la rutina **drot** de BLAS 1, con tres llamadas a **drot**.

Trabajando de esta manera en el Alliant FX/80 se obtendría un programa paralelo y que explota las características de vectorización y paralelización del sistema, dejándose el trabajo de vectorizar y paralelizar a BLAS 1. Para ver si puede ser conveniente la paralelización de otra manera alternativa a la del uso de BLAS 1 estudiaremos el funcionamiento de la rutina **drot** en el Alliant FX/80. En la tabla 3.2.1 mostramos la velocidad en Mflops obtenida ejecutando la rutina **drot** con distintos tamaños de vectores y utilizando distinto número de procesadores. Se observa un comportamiento irregular en cuanto a las prestaciones al aumentar el tamaño de los vectores, lo cual es típico de esta máquina [48] y ocurre debido a que se obtienen mejores prestaciones con vectores cuyo tamaño se acomoda más al tamaño de la caché.

En las tablas 3.2.2 y 3.2.3 se muestran, respectivamente, el speed-up y la eficiencia obtenidas con **drot**, cuando se deja la extracción del paralelismo y la vectorización exclusivamente al compilador y a BLAS. A lo largo de este capítulo, y mientras no se diga otra cosa, obtendremos la eficiencia y el speed-up considerando como programa secuencial el que estamos paralelizando pero ejecutándolo en un único procesador.

En general, al aumentar el número de procesadores disminuyen las prestaciones, y esto se nota principalmente con 8 procesadores (la eficiencia está por debajo de 0.5, utilizando exclusivamente concurrencia), y se observa que con 2 y 4 procesadores se

procesadores	1	2	4	8
8	0.66	0.70	0.75	0.70
16	1.14	1.34	1.38	1.29
32	1.70	2.25	2.42	2.43
64	1.97	3.55	4.04	4.64
128	2.17	3.75	6.50	9.26
256	2.74	4.79	9.06	7.62
512	2.81	6.62	14.07	10.32
1024	2.43	6.15	10.96	7.82
2048	2.84	7.07	14.88	9.28

Tabla 3.2.1: Mflops obtenidos con la rutina **drot** para distintos tamaños de vector.

obtienen buenas prestaciones, obteniéndose valores por encima del óptimo teórico (por encima del número de procesadores en el caso del speed-up y por encima de 1 en el caso de la eficiencia), lo que es una característica también típica de estos sistemas y es debido al trabajo en cada procesador con menos memoria al trabajar en paralelo. Esta reducción en las prestaciones al aumentar el número de procesadores apoya la idea de dividir el trabajo para hacer las llamadas a **drot** en cada nodo del sistema como alternativa a dejar todo el trabajo de paralelización y vectorización a BLAS 1.

procesadores	2	4	8
8	1.06	1.13	1.06
16	1.17	1.21	1.13
32	1.32	1.42	1.42
64	1.80	2.05	2.35
128	1.72	2.99	4.26
256	1.74	3.30	2.78
512	2.35	5.00	3.67
1024	2.53	4.51	3.21
2048	2.48	5.23	3.26

Tabla 3.2.2: Speed-up obtenida con la rutina **drot** para distintos tamaños de vector.

Estudiaremos a continuación el comportamiento de BLAS 1 en el método de Jacobi para el Problema Simétrico de Valores Propios en el Alliant FX/80 comparando distintos algoritmos:

- Un método de Jacobi usando la ordenación par-impar (PI), utilizando la idea de [91] para evitar movimientos de datos, haciéndose estos en la actualización de la

procesadores	2	4	8
8	0.53	0.28	0.13
16	0.58	0.30	0.14
32	0.66	0.35	0.17
64	0.90	0.51	0.29
128	0.86	0.74	0.53
256	0.87	0.82	0.34
512	1.17	1.25	0.45
1024	1.26	1.12	0.40
2048	1.24	1.30	0.40

Tabla 3.2.3: Eficiencia obtenida con la rutina **drot** para distintos tamaños de vector.

matriz. Se ha utilizado para analizar las prestaciones de la máquina en cuanto a paralelización cuando no se vectoriza.

- Un método de Jacobi usando la ordenación par-impar y BLAS 1 (PIB1), realizando las actualizaciones con **drot** y posteriormente los movimientos de datos por medio de la rutina **dswap**, lo que producirá un empeoramiento de las prestaciones, haciendo que no se obtengan las que se muestran en la tabla 3.2.1.

- Un método de Jacobi usando la ordenación cíclica por filas y BLAS 1 (FIB1) para explotar las posibilidades de paralelización y vectorización, cuando se deja esta tarea al compilador. Sólo usará la rutina **drot** de BLAS 1, no habiendo intercambio de datos.

- Un método de Jacobi usando la ordenación par-impar utilizando BLAS 1 en cada procesador de manera individual y balanceando la carga en la actualización de la matriz (PIBB1).

- Un método de Jacobi usando la ordenación par-impar utilizando BLAS 1 en cada procesador de manera individual y distribuyendo el trabajo a los procesadores en la actualización de la matriz asignando los pares de filas o columnas cíclicamente (PICB1).

Explicaremos de una manera más detallada la distribución del trabajo entre los procesadores cuando se balancea la carga y cuando se distribuyen los datos cíclicamente.

3.2.1 Balanceo de la carga

Este es el método de distribución del trabajo utilizado en PI y PIBB1.

Cada barrido tiene el esquema:

Algoritmo 3.2.1 *Esquema de un barrido en el método de Jacobi con ordenación par-impar.*

PARA $i = 1, 2, \ldots, \frac{n}{2}$
  calcular rotaciones impares
  premultipicar por las rotaciones impares
  postmultiplicar por rotaciones impares
  calcular rotaciones pares
  premultipicar por las rotaciones pares
  postmultiplicar por rotaciones pares
FINPARA

En el cálculo de rotaciones impares hay que calcular $\frac{n}{2}$ rotaciones, calculando cada procesador $\frac{n}{2p}$ rotaciones, si suponemos que n es múltiplo de $2p$ y siendo p el número de procesadores.

Para la premultiplicación por las rotaciones impares se supone la matriz dividida en pares de filas consecutivas y se asigna a cada procesador el trabajo correspondiente a $\frac{n}{2p}$ pares de filas (los elementos de la parte triangular inferior), asignándose al procesador P_i (numeraremos los procesadores de 0 a $p-1$ y los pares de filas consecutivas de 0 a $\frac{n}{2}-1$) los pares de filas del $i\frac{n}{4p}$ al $(i+1)\frac{n}{4p}-1$ y del $\frac{n}{2}-(i+1)\frac{n}{4p}$ al $\frac{n}{2}-i\frac{n}{4p}-1$, tal como se muestra el la figura 3.2.1, donde se muestra una matriz de tamaño 32×32 (la parte triangular inferior) agrupando las filas por pares, y suponemos que se dispone de 4 procesadores, con lo que a cada procesador le corresponden 4 pares de filas.

Para la postmultiplicación por las rotaciones impares se supone la matriz dividida en pares de columnas consecutivas y se asigna a cada procesador el trabajo correspondiente a $\frac{n}{2p}$ pares de columnas (los elementos de la parte triangular inferior), asignándose al procesador P_i los pares de columnas del $i\frac{n}{4p}$ al $(i+1)\frac{n}{4p}-1$ y del $\frac{n}{2}-(i+1)\frac{n}{4p}$ al $\frac{n}{2}-i\frac{n}{4p}-1$.

De esta manera, la actualización de la matriz en los pasos impares está balanceada, necesitándose sincronización después de cada uno de los tres procedimientos del paso, pero trabajando los procesadores en cada paso de manera independiente al trabajar con datos independientes.

En el cálculo de las rotaciones pares un procesador calcula una rotación menos que los demás, al tener que anularse $\frac{n}{2}-1$ elementos.

Del mismo modo, la premultiplicación y la postmultiplicación por las matrices de rotación tampoco estará totalmente balanceada.

Para la premultiplicación por las rotaciones pares se supone la matriz dividida en pares de filas consecutivas, tal como en los pasos impares pero habiendo en este caso $\frac{n}{2}-1$ pares de filas, y se asigna a cada procesador el trabajo correspondiente a $\frac{n}{2p}$ pares de filas (los elementos de la parte triangular inferior) salvo al procesador P_0 al que se le asigna una menos.

Para la postmultiplicación por las rotaciones pares se supone la matriz dividida en

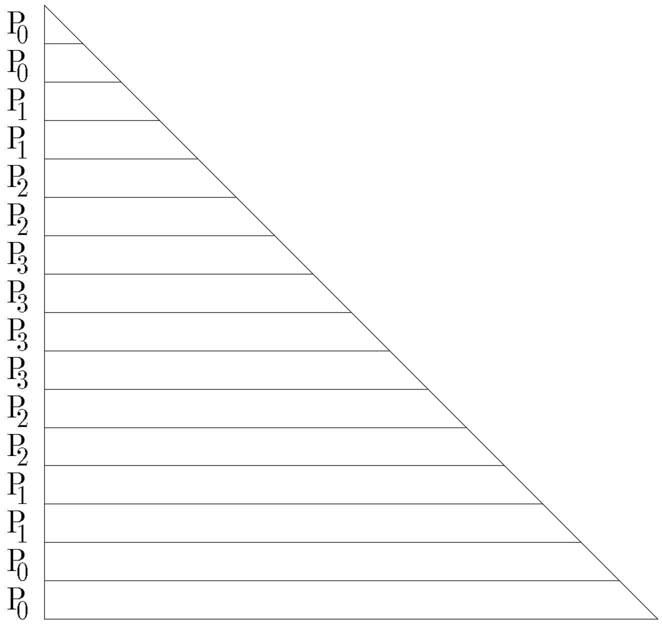


Figura 3.2.1: Distribución por balanceo de la carga

pares de columnas consecutivas, tal como en los pasos impares pero habiendo en este caso $\frac{n}{2}-1$ pares de columnas, y se asigna a cada procesador el trabajo correspondiente a $\frac{n}{2p}$ pares de columnas (los elementos de la parte triangular inferior) salvo al procesador P_0 al que se le asigna una menos.

Después de un paso hay que hacer un intercambio de filas y columnas entre filas y columnas correspondientes a los índices de ese paso, pero estos intercambios consisten en la premultiplicación y postmultiplicación de la matriz resultado de la actualización (QAQ^t) por una matriz de permutación P y su traspuesta.

En PI se puede agrupar la actualización de A por la matriz de rotaciones Q y la permutación de filas y columnas multiplicando por P de la forma:

$$P(QAQ^t)P^t = (PQ)A(Q^tP^t) \tag{3.2.1}$$

por lo que si se realiza la multiplicación PQ antes de la actualización de A nos evitaremos la parte de intercambios de filas y columnas en A [91]. Esta multiplicación PQ consiste simplemente en considerar los bloques diagonales 2×2 de la matriz Q (que es una matriz con todos los elementos 0 salvo los que están en bloques diagonales 2×2 correspondientes a las rotaciones de Givens) de la forma:

$$\begin{pmatrix} -s & c \\ c & s \end{pmatrix} \tag{3.2.2}$$

en vez de de la forma:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \tag{3.2.3}$$

En PIBB1, al utilizarse la rutina **drot** en las actualizaciones, no se puede incluir el intercambio de filas y de columnas junto con las actualizaciones, por lo que después de cada llamada a **drot** habrá una llamada a la rutina **dswap** para intercambiar las filas o columnas que se acaban de actualizar, habiendo en este caso un trabajo adicional al de actualización de la matriz, lo que no ocurre en PI.

3.2.2 Distribución cíclica

Es el método de distribución del trabajo entre los procesadores usado en PICB1.

En el paso impar la matriz se divide en pares de filas consecutivas y se asigna a cada procesador el trabajo correspondiente a $\frac{n}{2p}$ pares de filas (los elementos de la parte triangular inferior), asignándose al procesador P_i los pares de filas $i, i+p, i+2p, ...$ (figura 3.2.2).

Para la postmultiplicación por las rotaciones impares se supone la matriz dividida en pares de columnas consecutivas y se asigna a cada procesador el trabajo correspondiente a $\frac{n}{2p}$ pares de columnas, asignándose al procesador P_i los pares de columnas $i, i+p, i+2p, ...$

De modo similar es la actualización en los pasos impares.

3.2.3 Resultados experimentales

En esta subsección estudiaremos experimentalmente el comportamiento de los algoritmos descritos (PI, PIB1, FIB1, PIBB1 y PICB1). Usaremos en todos los casos matrices con valores en doble preción entre -10 y 10 generados aleatoriamente. En todas las tablas se mostrarán tiempos de ejecución en segundos por barrido, ya que el número de barridos necesarios para la convergencia es siempre el mismo y es del orden $O(log_2\ n)$.

En la tabla 3.2.4 se muestran los tiempos obtenidos con el algoritmo PI para distintos tamaños de matriz y del sistema. Se muestra también la eficiencia. Se observan unas buenas prestaciones en cuanto a eficiencia obtenida, siendo en algunos casos mayor que 1 y teniendo valores alrededor de 0.98 con 2 procesadores, de 0.94 con 4 procesadores y 0.85 con 8 procesadores, siendo normal que al aumentar el número de procesadores disminuya la eficiencia. Los tiempos de ejecución no son muy buenos debido a que no se usa BLAS 1 ni se ha vectorizado el programa, por lo que los resultados de la tabla

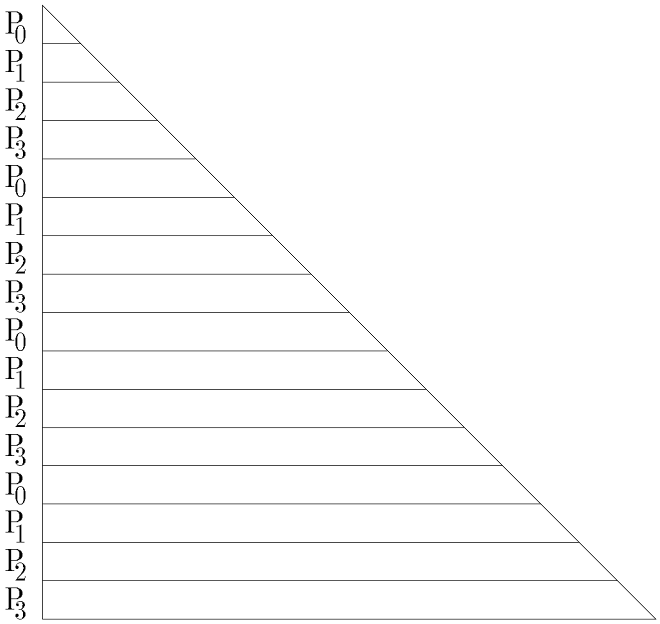


Figura 3.2.2: Distribución cíclica

3.2.4 sólo nos sirven para ver el comportamiento del sistema en cuanto a paralelización y compararlo con el comportamiento con los demás programas.

En las tablas 3.2.5, 3.2.6 y 3.2.7 se muestran los resultados de tiempos de ejecución y eficiencias obtenidos con PIBB1, PICB1 y FIB1, respectivamente. En general se observa un comportamiento más irregular que el obtenido con PI, debido al uso de BLAS 1 (las rutinas **drot** y **dswap** en PIBB1 y PICB1, y la rutina **drot** en FIB1). Comparando los tiempos obtenidos en un procesador con PI y FIB1 (tablas 3.2.4 y 3.2.7) observamos que el uso de BLAS 1 produce una gran mejora en el tiempo de ejecución debido al uso eficiente de los recursos de la máquina, entre ellos la vectorización, siendo FIB1 alrededor de 7 veces más rápido que PI en un procesador, pero se observa que mientras que con PI se obtienen buenas eficiencias con FIB1 estas son muy bajas, principalmente al aumentar el número de procesadores. Comparando PI con PIBB1 y PICB1 se observa también una reducción en el tiempo de ejecución, pero en este caso los programas PIBB1 y PICB1 llegan a ser sólo aproximadamente 3 veces más rápidos que PI en un procesador, pero hay que tener en cuenta que en este caso se están creando procesos con la llamada **concurrent_call** para llevar a cabo cada uno de los 6 procedimientos en cada paso del barrido, y que tras las actualizaciones se hace un intercambio de filas y de columnas, lo cual es bastante costoso en este tipo de sistemas, produciendo un tiempo de ejecución adicional que puede ser importante.

	tiempos				*eficiencia*		
procesadores	1	2	4	8	2	4	8
128	17.51	8.65	4.22	2.50	1.01	1.05	0.87
192	59.26	30.25	15.18	7.65	0.97	0.97	0.96
256	140.88	71.43	36.89	19.65	0.98	0.95	0.89
320	276.97	139.32	72.87	39.68	0.99	0.95	0.87
384	488.01	252.02	127.03	70.03	0.96	0.96	0.87
448	773.54	398.51	206.40	113.31	0.97	0.93	0.85
512	1190.47	606.77	321.38	183.42	0.98	0.92	0.81
576	1658.82	846.82	443.80	248.83	0.97	0.93	0.83
640	2282.27	1168.21	625.31	347.11	0.97	0.91	0.82

Tabla 3.2.4: Tiempos de ejecución por barrido en segundos y eficiencia, con PI.

En cuanto a la eficiencia, se observa que con FIB1 se obtienen unas eficiencias muy pobres al paralelizarse sólo a nivel de las llamadas a **drot**, en las que se paraleliza y vectoriza, y además se obtiene la mayor eficiencia en un punto concreto (alrededor de tamaño de la matriz 384) y a partir de ahí la eficiencia vuelve a caer, lo que puede ser debido a que el trabajo con unos ciertos tamaños de matriz permita a BLAS hacer un mejor uso de la caché. También se observa una gran reducción en la eficiencia al utilizar 8 procesadores, lo que se corresponde con el mal funcionamiento de **drot** al aumentar el tamaño del sistema, lo que se había mostrado en las tablas 3.2.2 y 3.2.3. En cualquier caso, no se pueden extrapolar los resultados obtenidos con **drot** a los métodos de Jacobi que estamos estudiando, ya que en la tabla 3.2.1 mostrábamos los Mflops obtenidos con unos tamaños de vector fijos y estando los datos almacenados en posiciones contiguas de memoria, pero en los métodos de Jacobi, en cada paso de un barrido tenemos 3 procedimientos, uno de ellos es de cálculo de rotaciones, donde no se usa BLAS, otro es de actualización de la matriz premultiplicando por las rotaciones obtenidas, lo que se hace sobre vectores almacenados en posiciones contiguas de memoria, pero de tamaños variables, teniendo vectores desde tamaño 2 hasta $n-2$, y en el procedimiento de actualización de la matriz postmultiplicando se trabaja sobre vectores de distinto tamaño y además almacenados en posiciones no contiguas de memoria (siendo la distancia entre dos elementos consecutivos de un vector de n posiciones de memoria). En la tabla 3.2.8 se muestran los Mflops obtenidos con PICB1, que es el mejor algoritmo de los que usa BLAS. Si comparamos con la tabla 3.2.1 se observan unas prestaciones mucho peores que las que se obtienen con **drot**, lo que es debido a lo que hemos mencionado además de al movimiento de datos en memoria usando **dswap** en PICB1. Además, en PICB1 se produce una gran reducción en las prestaciones al aumentar el tamaño de la matriz y del sistema debido a que la carga no está totalmente balanceada entre los procesadores, y este desbalanceo se nota más al aumentar los tamaños.

	tiempos				*eficiencia*		
procesadores	1	2	4	8	2	4	8
128	5.83	2.32	1.53	0.66	1.25	0.95	1.10
192	17.52	8.93	5.41	3.11	0.98	0.80	0.70
256	39.94	20.08	12.39	8.18	0.99	0.80	0.61
320	76.18	39.57	23.26	15.77	0.96	0.81	0.60
384	132.04	69.04	45.04	33.03	0.95	0.73	0.49
448	216.33	130.89	76.37	50.56	0.82	0.70	0.53
512	410.39	233.34	158.47	127.38	0.87	0.64	0.40
576	488.82	256.82	193.81	121.83	0.87	0.57	0.46
640	751.26	382.64	272.80	220.10	0.98	0.68	0.42

Tabla 3.2.5: Tiempos de ejecución por barrido en segundos y eficiencia, con PIBB1.

	tiempos				*eficiencia*		
procesadores	1	2	4	8	2	4	8
128	5.83	2.32	1.56	0.67	1.25	0.93	1.08
192	17.54	8.93	4.56	3.29	0.98	0.96	0.66
256	39.95	20.12	11.81	7.38	0.99	0.84	0.67
320	76.11	39.64	23.13	15.79	0.96	0.82	0.60
384	132.22	77.63	41.15	27.56	0.85	0.80	0.59
448	221.15	118.66	81.30	50.02	0.93	0.68	0.55
512	414.56	226.35	145.99	110.04	0.91	0.70	0.47
576	545.80	252.48	148.17	122.74	1.08	0.92	0.55
640	714.04	354.53	208.35	160.45	1.00	0.85	0.55

Tabla 3.2.6: Tiempos de ejecución por barrido en segundos y eficiencia, con PICB1.

Comparando los tiempos de ejecución obtenidos con PIBB1 y PICB1 (tablas 3.2.5 y 3.2.6) se observa que, aunque en PIBB1 el trabajo está más balanceado que en PICB1, en este último los tamaños de los vectores con los que trabajan los procesadores en las actualizaciones de la matriz son más uniformes, lo que produce un uso más equilibrado de BLAS 1 obteniéndose mejores tiempos de ejecución generalmente, y principalmente cuando se aumenta el tamaño del sistema.

En la tabla 3.2.9 se muestran los cocientes entre los tiempos de ejecución obtenidos con PIBB1 y PICB1 y los obtenidos con FIB1 (PIBB1/FIB1 y PICB1/FIB1), indicando un valor mayor que 1 que FIB1 tiene mejor tiempo de ejecución y un valor menor que 1 que tiene peor tiempo de ejecución. Se muestran también los cocientes medios

	tiempos				*eficiencia*		
procesadores	1	2	4	8	2	4	8
128	3.15	2.27	2.32	2.32	0.69	0.23	0.16
192	11.82	8.06	5.61	5.65	0.73	0.52	0.26
256	26.34	16.50	11.80	11.18	0.79	0.55	0.29
320	49.32	30.93	22.41	19.85	0.79	0.55	0.31
384	107.01	51.04	37.04	32.04	1.04	0.72	0.41
448	152.01	94.27	59.39	50.23	0.80	0.63	0.37
512	280.11	192.97	160.89	155.58	0.72	0.43	0.22
576	222.79	248.77	170.80	209.75	0.44	0.32	0.13
640	317.87	311.07	249.97	197.27	0.51	0.31	0.20

Tabla 3.2.7: Tiempos de ejecución por barrido en segundos y eficiencia, con FIB1.

procesadores	1	2	4	8
128	1.07	2.71	4.03	9.39
192	1.21	2.37	4.65	6.45
256	1.25	2.50	4.26	6.82
320	1.29	2.47	4.25	6.22
384	1.28	2.18	4.12	6.16
448	1.21	2.27	3.31	5.39
512	0.97	1.77	2.75	3.65
576	1.05	2.27	3.86	4.67
640	1.10	2.21	3.77	4.90

Tabla 3.2.8: Mflops con PICB1.

obtenidos para cada tamaño del sistema y de la matriz, y se marca en cada caso la mejor de las medias obtenidas, correspondiendo ésta en la gran mayoría de los casos a PICB1/FIB1. Se observa que los mejores tiempos de ejecución se obtienen con PIBB1 y PICB1, con lo que podemos concluir que en algunos casos es preferible, para obtener mejores tiempos de ejecución en un Multiprocesador, paralelizar dividiendo el trabajo entre los procesadores y utilizar BLAS 1 en cada procesador en vez de dejar todo el trabajo de paralelización y vectorización a BLAS 1. Además, y tal como ya hemos dicho, algunas veces es preferible no balancear perfectamente la carga en la distribución de los datos entre los procesadores sino que permitiendo un ligero desbalanceo de la carga que permita un uso más equilibrado de BLAS se pueden obtener mejores tiempos de ejecución (el cociente correspondiente a PICB1 es menor que el correspondiente a PIBB1 en la tabla 3.2.9 en la mayoría de los casos).

	PIBB1/FIB1				PICB1/FIB1			
procesadores	2	4	8	media	2	4	8	media
128	1.02	0.65	0.28	**0.65**	1.02	0.67	0.28	0.66
192	1.10	0.96	0.55	0.87	1.10	0.81	0.58	**0.83**
256	1.21	1.05	0.73	1.00	1.21	1.00	0.66	**0.96**
320	1.27	1.03	0.79	**1.03**	1.28	1.03	0.79	**1.03**
384	1.35	1.21	1.03	1.19	1.52	1.11	0.86	**1.16**
448	1.38	1.28	1.00	1.22	1.25	1.36	0.99	**1.20**
512	1.20	0.98	1.03	1.07	1.17	0.90	0.70	**0.93**
576	1.03	1.13	0.58	0.92	1.01	0.86	0.58	**0.82**
640	1.23	1.09	1.11	1.14	1.13	0.83	0.81	**0.93**
media	1.19	1.04	0.79	1.00	**1.18**	**0.96**	**0.70**	**0.95**

Tabla 3.2.9: Cocientes entre los tiempos de ejecución obtenidos con PIBB1 y PICB1 y los obtenidos con FIB1.

Finalmente, en la tabla 3.2.10 mostramos los tiempos de ejecución por barrido y las eficiencias obtenidas con PIB1. Comparando los resultados de las tablas 3.2.7 y 3.2.10 observamos que los tiempos de ejecución son mucho mejores con FIB1 que con PIB1, a pesar de dejar en los dos casos la explotación del paralelismo del sistema en el BLAS 1. Esto se explica porque con PIB1, después de cada llamada a la rutina **drot**, se utiliza la rutina **dswap** para intercambiar las filas o columnas que se acaban de actualizar. Esto explica también la ventaja tan reducida que se obtiene con PICB1 al compararlo con FIB1 (tabla 3.2.9), ya que en FIB1 no se realiza el intercambio de datos que se lleva a cabo en PIB1 y PICB1.

	tiempos				*eficiencia*		
procesadores	1	2	4	8	2	4	8
128	4.96	3.16	2.25	2.26	0.78	0.55	0.27
192	16.15	10.14	7.99	7.15	0.80	0.51	0.28
256	37.63	22.86	16.79	14.22	0.82	0.56	0.33
320	77.88	43.00	31.54	27.52	0.91	0.62	0.35
384	143.26	87.29	54.25	58.06	0.82	0.66	0.31
448	205.58	154.93	91.30	88.03	0.66	0.56	0.29
512	404.83	264.56	224.41	245.96	0.77	0.45	0.21
576	505.24	328.32	224.15	334.66	0.77	0.56	0.19
640	694.08	451.60	354.03	290.11	0.77	0.49	0.30

Tabla 3.2.10: Tiempos de ejecución por barrido en segundos y eficiencia, con PIB1.

3.3 Uso de BLAS 3

En esta sección estudiaremos la utilización del método de Jacobi por bloques para la solución del Problema Simétrico de Valores Propios, que se estudió en el capítulo 1, en el Multiprocesador Alliant FX/80. Las implementaciones que se estudiarán son:

- Un método que deja la explotación de las características del sistema en BLAS 3 (B3). Que es el método secuencial estudiado cuando se deja la tarea de explotar el paralelismo y la vectorización a BLAS 3.
- Un método que divide el trabajo entre los procesadores de manera cíclica y usa BLAS 3 en cada procesador (B3C).
- Un método que divide el trabajo entre los procesadores balanceando la carga y con bloques adyacentes y usa BLAS 3 en cada procesador (B3B).
- Un método que divide el trabajo entre los procesadores balanceando la carga y de manera cíclica y usa BLAS 3 en cada procesador (B3CB).
- Un método que divide el trabajo ente los procesadores balanceando la carga de manera cíclica y que trabaja con la parte triangular inferior de la matriz utilizando la parte triangular superior como auxiliar para evitar copias de datos (B3M).

El esquema básico de todos los métodos es el algoritmo que se muestra en el capítulo 1. Este algoritmo trabaja por bloques generando los bloques a tratar con la ordenación par-impar (por lo que las ideas de la sección anterior se pueden aplicar en este caso si consideramos cada bloque como un elemento de la matriz de bloques) y anulando los elementos de cada bloque con una ordenación cíclica por filas.

Compararemos el algoritmo por bloques dejando la explotación del paralelismo a BLAS 3 (B3) con otros métodos en los que se explota el paralelismo por división del trabajo entre los procesadores y haciendo llamadas a la rutina **dgemm** de BLAS 3 en cada procesador (B3C, B3B, B3CB y B3M).

Los métodos en los que se divide el trabajo entre los procesadores se diferencian en la forma en que se realiza esta división. El trabajo se divide en B3C asignando a los procesadores pares de filas (o columnas) de bloques de forma cíclica, tal como se muestra en la figura 3.2.2 para BLAS 1. En B3B se divide balanceando la carga asignando a cada procesador una serie de filas (o columnas) consecutivas de la mitad superior de la matriz y otra de la mitad inferior, tal como se muestra en 3.2.1 para BLAS 1. De esta manera, en B3C se consigue una buena distribución en cuanto al trabajo con BLAS 3, al hacerse las actualizaciones sobre matrices de tamaños similares, pero esto es a costa de un desequilibrio de la carga; mientras en B3B tenemos un mejor balanceo de la carga pero hay una mayor diferencia en los tamaños de matrices que actualiza cada procesador, lo que puede producir un uso más desequilibrado de BLAS 3. Para buscar un punto intermedio entre B3C y B3B se puede utilizar una distribución cíclica balanceada (B3CB) dividiendo la matriz en dos partes, la superior formada por la primera mitad de filas (o columnas) de bloques y la inferior formada por la segunda mitad; de esta forma se asignan a los procesadores pares de filas de bloques de manera cíclica siguiendo el orden P_0, P_1, ..., P_{p-1} en la parte superior y el orden P_{p-1}, P_{p-2}, ..., P_0 en la parte inferior. En B3M se utiliza la misma división del trabajo que en B3CB y se utiliza la parte triangular superior de la matriz como auxiliar para evitar las copias que hay que hacer con **dcopy** (BLAS 1) tras cada llamada a **dgemm** (BLAS 3) al no permitir esta última rutina sobreescribir la matriz que se está actualizando.

3.3.1 Prestaciones de BLAS 3

En esta subsección analizaremos las prestaciones que se pueden obtener con la rutina **dgemm** de BLAS 3 ya que con esta rutina se actualiza la matriz, siendo esta la parte del algoritmo que ocasiona que tengamos un coste de $4n^3$ por barrido. La rutina **dgemm** multiplica dos matrices dando como resultado una tercera, no pudiendo utilizarse como zona de almacenamiento en la memoria para la matriz resultado zonas ocupadas por las matrices a multiplicar, lo que producirá que en la mayoría de los algoritmos implementados que una llamada a **dgemm** sea seguida de una copia (con llamadas a **drot**) de la matriz resultado sobre una de las matrices a multiplicar.

En la tabla 3.3.1 se muestran los Mflops obtenidos con **dgemm** con distinto número de procesadores (se deja la explotación del paralelismo a BLAS 3) y con distintos tamaños de matrices (en esta tabla y en las 3.3.2 y 3.3.3 cada fila corresponde a Mflops obtenidos al multiplicar matrices de tamaños los que se indica en la primera columna de la tabla). Se observa que se obtienen buenas eficiencias, principalmente al aumentar el tamaño de las matrices que se multiplican. Pero el algoritmo por bloques presenta un par de inconvenientes:

1. La rutina **dgemm** no nos permite sobreescribir la matriz que estamos actualizando, por lo que en la mayoría de los métodos que estudiamos se necesitará de una copia de datos usando **dcopy** tras cada llamada a **dgemm**, lo que producirá una reducción en las prestaciones que podemos esperar. En la tabla 3.3.2 se muestran los Mflops obtenidos cuando tras la llamada a **dgemm** se realiza una copia de la matriz resultado realizando sucesivas llamadas a **dcopy**. Comparando los resultados de esta tabla con los de la tabla 3.3.1 se puede observar la reducción en las prestaciones que hemos comentado.

2. Al aumentar el tamaño de los subbloques aumenta el trabajo a realizar con BLAS 1 al hacerse los subbarridos para obtener las matrices de rotaciones sobre subbloques mayores, lo que hará disminuir las prestaciones que se obtienen usando BLAS 3. Por tanto, los resultados óptimos se obtendrán normalmente con tamaños de bloque pequeños (entre 8 y 32). Además, los resultados que se muestran en la tabla 3.3.1 han sido obtenidos con matrices muy grandes, pero en la actualización de la matriz hay que actualizar bloques de tamaños muy diferentes al actualizarse sólo la parte triangular inferior. De este modo, resultados más realistas de acuerdo a los tamaños de matrices con los que se va a trabajar son los que se muestran en la tabla 3.3.3, donde podemos ver que los buenos resultados de eficiencia que se obtenían en las tablas 3.3.1 y 3.3.2 se han reducido considerablemente, por lo que puede ser aconsejable la división del trabajo entre los procesadores y el uso de BLAS 3 en cada procesador.

procesadores :	1	2	4	6	8
$8 \times 8, 8 \times 512$	5.60	12.39	22.26	31.76	30.58
$16 \times 16, 16 \times 256$	7.39	15.13	36.65	35.17	71.14
$32 \times 32, 32 \times 128$	10.32	18.47	33.47	97.66	58.47
$64 \times 64, 64 \times 64$	10.05	19.97	47.51	62.42	100.86

Tabla 3.3.1: Mflops con **dgemm**. Para distintos tamaños de las dos matrices a multiplicar.

3.3.2 Resultados experimentales

En esta subsección analizaremos los resultados obtenidos con los distintos métodos por bloques implementados (B3, B3C, B3B, B3CB y B3M), comparándolos entre sí y con los métodos que usan BLAS 1. Todos los resultados se han obtenido con matrices de reales de doble precisión generados aleatoriamente, y los programas se han hecho en el

$procesadores:$	1	2	4	6	8
$8 \times 8, 8 \times 512$	4.43	9.06	16.94	19.89	21.15
$16 \times 16, 16 \times 256$	6.37	12.94	25.53	29.09	37.62
$32 \times 32, 32 \times 128$	8.23	15.96	25.60	44.44	46.90
$64 \times 64, 64 \times 64$	9.02	17.21	44.98	46.59	46.60

Tabla 3.3.2: Mflops con **dgemm** seguido de copia de la matriz resultado usando **dcopy**. Para distintos tamaños de las dos matrices a multiplicar.

$procesadores:$	1	2	4	6	8
$8 \times 8, 8 \times 8$	1.45	1.77	1.95	1.95	2.05
$8 \times 8, 8 \times 32$	2.87	4.02	5.11	6.18	6.23
$8 \times 8, 8 \times 128$	3.58	6.75	10.60	12.71	16.30
$8 \times 8, 8 \times 512$	4.43	9.06	16.98	19.89	21.15
$16 \times 16, 16 \times 8$	2.95	3.90	4.51	4.65	4.73
$16 \times 16, 16 \times 32$	5.08	8.12	11.38	12.37	13.78
$16 \times 16, 16 \times 128$	5.99	11.84	22.37	24.56	35.95
$16 \times 16, 16 \times 512$	6.17	13.31	26.25	30.79	33.04
$32 \times 32, 32 \times 8$	5.05	7.23	9.61	9.21	9.52
$32 \times 32, 32 \times 32$	7.53	12.13	17.93	24.20	29.79
$32 \times 32, 32 \times 128$	8.23	15.96	25.60	44.44	46.90
$32 \times 32, 32 \times 512$	8.25	19.49	30.98	71.39	46.08

Tabla 3.3.3: Mflops con **dgemm** seguido de copia de la matriz resultado usando **dcopy**. Para distintos tamaños de las dos matrices a multiplicar.

lenguaje C. Los tiempos que se mostrarán serán en todos los casos tiempos en segundos y por barrido, ya que el número de barridos es en todos los casos del orden de $log_2\ n$.

Los resultados se han obtenido con tamaños de matriz desde 128 a 640 y variando el tamaño de 64 en 64, y para tamaños de bloque 8, 16, 24, 32, ..., siempre que el tamaño de la matriz fuera múltiplo del tamaño del bloque. En las tablas no se mostrarán todos los resultados debido a la gran cantidad de datos que se han generado, sino que se presentarán los más significativos.

En la tabla 3.3.4 se muestran los tiempos de ejecución obtenidos cuando se deja la explotación del paralelismo exclusivamente a BLAS 3 (B3). Se representan tiempos de ejecución para 1, 2, 4, 6 y 8 procesadores , tamaños de matriz 256, 384, 512 y 640 y diversos tamaños de bloque. Se marcan los tiempos óptimos obtenidos para cada número de procesadores y tamaño de matriz. Observando esta tabla se pueden sacar algunas conclusiones:

1. En cuanto al tamaño de bloque óptimo, este está en todos los casos entre 16 y 32, aumentando al aumentar el tamaño de la matriz. Además, los tiempos de ejecución cuando se toman tamaños de bloque cercanos al óptimo son muy parecidos.

2. Se observa que al aumentar el número de procesadores disminuye en general el tiempo de ejecución, aunque las eficiencias que se obtienen son muy bajas (tal como pasaba cuando se dejaba la explotación del paralelismo a BLAS 1, tabla 3.2.7), debido a la pobre utilización de los recursos del sistema que se hace cuando se deja todo el trabajo a BLAS 3.

3. Comparando los resultados con los de la tabla 3.2.7 observamos que se obtiene una reducción en el tiempo de ejecución usando BLAS 3 en vez de BLAS 1 al aumentar el tamaño de la matriz (tamaños mayores que 256) y que esta reducción aumenta al aumentar el tamaño de la matriz.

En las tablas 3.3.5, 3.3.6 y 3.3.7 se muestran los tiempos de ejecución obtenidos con los programas B3C, B3B y B3CB, respectivamente, para los mismos tamaños de sistema y matriz mostrados en la tabla 3.3.4.

Algunas conclusiones que se pueden sacar observando las tablas y comparando con la tabla 3.3.4 son:

1. En cuanto al tamaño de bloque óptimo, es en casi todos los casos 16, aumentando al aumentar el tamaño de la matriz. En este caso el tamaño de bloque óptimo es menor que con B3. Además, los tiempos de ejecución cuando se toman tamaños de bloque cercanos al óptimo son muy parecidos, tal como ocurría con B3.

2. Las eficiencias que se obtienen están lejos de ser óptimas pero son mucho mayores que las que se obtenían con B3, lo que produce que, aunque los tiempos de

	1 procesador			
bloque:	8	16	32	64
256	23.56	**20.61**	22.78	30.90
384	70.01	**57.81**	59.97	76.66
512	154.20	**121.41**	123.22	152.16
640	281.29	215.63	**213.90**	253.13
	2 procesadores			
bloque:	8	16	32	64
256	18.56	**15.94**	16.96	20.71
384	53.02	43.28	**42.21**	50.95
512	111.21	86.06	**82.42**	95.63
640	198.27	149.70	**140.37**	158.76
	4 procesadores			
bloque:	8	16	32	64
256	16.56	14.67	**14.11**	16.85
384	44.00	35.38	**33.53**	38.59
512	90.20	69.25	**64.18**	70.46
640	158.28	115.50	**105.93**	114.72
	6 procesadores			
bloque:	8	16	32	64
256	15.56	**12.38**	12.67	15.09
384	41.01	31.81	**30.47**	33.69
512	84.21	62.53	**59.43**	65.08
640	148.25	105.72	**97.18**	104.22
	8 procesadores			
bloque:	8	16	32	64
256	15.57	**13.42**	13.59	15.08
384	41.04	32.17	**30.60**	33.59
512	82.21	62.12	**56.82**	61.32
640	142.25	102.23	**93.02**	98.53

Tabla 3.3.4: Tiempos de ejecución por barrido en segundos de B3.

ejecución en un procesador son muy parecidos con los cuatro programas debido a que se basan todos en el mismo algoritmo, se obtenga una gran reducción en el tiempo de ejecución usando la técnica de utilización de BLAS 3 en cada procesador de manera individual, aumentando esta reducción al aumentar el número de procesadores y llegándose a obtener programas entre dos y tres veces más rápidos con ocho procesadores.

3. El mejor de los tres métodos parece B3CB (dividir de manera cíclica y balanceando), pero las diferencias no son muy grandes por lo que no se pueden obtener conclusiones definitivas.

Para analizar con más detalle cómo evoluciona el tamaño de bloque óptimo, representamos este tamaño en la tabla 3.3.8 para los distintos tamaños de matriz con que se ha experimentado, 2, 4, 6 y 8 procesadores y los programas B3, B3C, B3B y B3CB.

Se pueden hacer los siguientes comentarios:

1. El uso que BLAS 3 hace de la jerarquía de memorias hace que el tamaño de bloque óptimo no evolucione uniformemente con el tamaño de la matriz. Esto se observa principalmente con B3.

2. En B3 los tamaños de bloque óptimo son casi siempre 32 ó 64, a pesar de haber experimentado con tamaños que no son potencias de 2. Este comportamiento debe ser ocasionado por el uso que de la memoria hace BLAS.

3. En los programas B3C, B3B y B3CB se observa que el aumento del tamaño de la matriz produce un ligero aumento en el tamaño de bloque óptimo, y que un aumento en el número de procesadores produce una disminución en el tamaño de bloque óptimo, aunque se puede considerar que un tamaño de bloque de 16 puede ser el adecuado.

Para comparar los métodos por bloques presentados (B3, B3C, B3B y B3CB) mostramos en la tabla 3.3.9 los cocientes de los tiempos de ejecución obtenidos con B3C, B3B y B3CB con relación a los obtenidos con B3, utilizando siempre el tamaño de bloque óptimo. Un valor mayor de 1 indicaría que B3 es mejor que el método con el que se está comparando. Se observa que los métodos que se basan en llamadas a **dgemm** en cada procesador son claramente mejores que B3 al ser todos los valores de la tabla menores que 1. Los métodos con balanceo de la carga (B3B y B3CB) son algo mejores que el que utiliza una distribución cíclica, pero no se encuentra una diferenciación clara entre ellos. En la tabla 3.3.10 se muestra la media de los cocientes de la tabla 3.3.9 para cada uno de los métodos y los distintos tamaños de matriz, y en la tabla 3.3.11 se hace lo mismo para los distintos números de procesadores. Los resultados de estas tablas confirman las apreciaciones que hemos hecho sobre la comparación de B3C, B3B y B3CB. Se observa que al aumentar el tamaño de las matrices la ganancia

	1 procesador					
bloque:	8	16	24	32	40	48
256	25.25	**22.01**		23.97		
384	71.32	**58.88**	59.27	61.60		68.86
512	153.97	**121.72**		124.29		
640	285.32	218.64		**217.00**	229.23	
	2 procesadores					
bloque:	8	16	24	32	40	48
256	13.41	**11.75**		14.27		
384	37.67	**31.91**	33.15	34.82		41.60
512	80.58	**65.43**		69.43		
640	149.30	**116.58**		119.70	129.33	
	4 procesadores					
bloque:	8	16	24	32	40	48
256	7.48	**6.70**		8.35		
384	20.82	**18.99**	19.74	24.26		25.97
512	44.87	**36.63**		41.22		
640	83.72	**66.18**		75.31	78.30	
	6 procesadores					
bloque:	8	16	24	32	40	48
256	**5.50**	6.20		8.33		
384	15.88	**14.09**	17.88	17.71		28.17
512	34.97	**30.08**		38.94		
640	62.87	**53.21**		61.54	71.80	
	8 procesadores					
bloque:	8	16	24	32	40	48
256	4.51	**4.35**		9.15		
384	13.90	**13.21**	13.51	17.63		27.22
512	30.00	**25.77**		29.65		
640	52.89	**44.60**		55.17	55.32	

Tabla 3.3.5: Tiempos de ejecución por barrido en segundos de B3C.

	1 procesador					
bloque:	8	16	24	32	40	48
256	24.25	**21.16**		23.23		
384	70.31	**58.13**	59.47	60.89		69.27
512	154.95	121.74		**120.95**		
640	281.30	215.25		**214.23**	226.55	
	2 procesadores					
bloque:	8	16	24	32	40	48
256	13.40	**12.01**		13.87		
384	36.66	**31.02**	32.05	34.69		39.74
512	78.55	**63.08**		66.17		
640	148.23	**114.22**		118.82	124.35	
	4 procesadores					
bloque:	8	16	24	32	40	48
256	7.47	**7.20**		9.40		
384	19.82	18.90	**17.54**	24.81		26.28
512	43.85	**35.32**		38.93		
640	80.71	**66.64**		76.28	69.26	
	6 procesadores					
bloque:	8	16	24	32	40	48
256	**5.50**	6.22		8.30		
384	13.87	**12.31**	17.80	17.33		25.90
512	34.95	**30.85**		40.08		
640	64.86	**53.13**		55.40	73.13	
	8 procesadores					
bloque:	8	16	24	32	40	48
256	**3.51**	3.95		8.77		
384	13.89	**13.48**	14.30	17.76		26.69
512	26.99	**21.70**		28.76		
640	53.92	**47.23**		59.54	54.34	

Tabla 3.3.6: Tiempos de ejecución por barrido en segundos de B3B.

	1 procesador					
bloque:	8	16	24	32	40	48
256	23.26	**20.47**		22.71		
384	69.32	**57.57**	58.60	60.69		68.72
512	152.97	**121.08**		122.69		
640	287.29	215.44		**213.33**	226.79	
	2 procesadores					
bloque:	8	16	24	32	40	48
256	13.41	**12.10**		14.00		
384	35.67	**30.18**	32.15	33.84		39.89
512	80.58	**64.25**		67.02		
640	147.29	**114.16**		118.29	124.10	
	4 procesadores					
bloque:	8	16	24	32	40	48
256	6.48	**6.21**		8.47		
384	18.83	18.00	**16.89**	24.21		25.69
512	42.87	**34.49**		36.56		
640	80.73	**65.62**		75.78	68.17	
	6 procesadores					
bloque:	8	16	24	32	40	48
256	**5.50**	6.25		8.32		
384	13.88	**12.38**	17.92	16.55		25.41
512	33.96	**30.53**		39.40		
640	63.88	**52.66**		56.78	73.03	
	8 procesadores					
bloque:	8	16	24	32	40	48
256	**3.51**	3.98		8.80		
384	**12.90**	12.94	13.81	16.42		26.43
512	27.01	**22.09**		29.44		
640	53.93	**47.33**		60.18	54.01	

Tabla 3.3.7: Tiempos de ejecución por barrido en segundos de B3CB.

	B3				B3C				B3B				B3CB			
	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
128	16	16	32	32	8	8	8	8	8	16	8	8	8	16	8	8
192	64	64	64	64	16	24	8	8	16	24	8	8	16	24	16	8
256	16	32	16	16	16	16	8	16	16	16	8	8	16	16	8	8
320	64	64	64	64	16	16	16	8	16	8	16	8	16	8	16	8
384	32	32	32	32	16	16	16	16	16	24	16	16	16	24	16	8
448	32	64	64	64	16	16	16	16	16	16	16	16	16	16	16	16
512	16	32	32	32	16	16	16	16	16	16	16	16	16	16	16	16
576	32	64	64	64	16	16	16	16	32	32	24	24	16	24	24	24
640	32	32	32	32	16	16	16	16	16	16	16	16	16	16	16	16

Tabla 3.3.8: Tamaño de bloque óptimo con los diferentes algoritmos que trabajan por bloques.

que se obtiene distribuyendo el trabajo disminuye, lo que es normal al hacer B3 un mejor uso de las características paralelas del sistema con mayor tamaño de matriz. Sin embargo, al aumentar el tamaño del sistema la ganancia que se obtiene distribuyendo el trabajo entre los procesadores es mayor, y también es mayor la mejora obtenida por balanceo de la carga.

Para obtener una versión más eficiente modificamos B3CB para obtener un método en el que se eviten las copias que se hacen en los métodos estudiados después de cada llamada a **dgemm**. De este modo se puede obtener una reducción en el tiempo de ejecución; y usando la parte triangular superior de la matriz como auxiliar utilizándola como destino en las premultiplicaciones por matrices de rotación y como origen (y la parte triangular inferior como destino) en las postmultiplicaciones se consigue además un pequeño ahorro de memoria al no necesitar la memoria auxiliar donde se almacenaban los resultados de las multiplicaciones. Este método es el que hemos llamado B3M, y su comportmiento es similar al de B3CB en cuanto a tamaño de bloque óptimo. Los tiempos de ejecución por barrido de este método se muestran en la tabla 3.3.12, siendo estos tiempos los obtenidos con el tamaño de bloque óptimo. Comparando con la tabla 3.3.7 se observa una reducción en el tiempo de ejecución con B3M, reducción que aumenta al aumentar el tamaño de las matrices y que se llega a situar alrededor de un diez por ciento.

Comparamos las prestaciones de B3M con las de PICB1, mostrando en la tabla 3.3.13 el cociente de los tiempos de B3M en relación a los de PICB1. En esta tabla se observa una mejora relativa de B3M respecto a PICB1 tanto al aumentar el tamaño de la matriz como el del sistema, llegándose a obtener que B3M es cerca de 4 veces más rápido que PICB1 con tamaños de matriz y de sistema grandes.

	2			4			6			8		
	B3C	B3B	B3CB	B3C	B3B	B3CB	B3C	B3B	B3CB	B3C	B3B	B3CB
128	0.59	0.59	0.59	0.35	0.21	0.21	0.35	0.35	0.35	0.36	0.36	0.36
192	0.83	0.86	0.86	0.58	0.57	0.57	0.56	0.35	0.43	0.55	0.35	0.55
256	0.73	0.75	0.75	0.47	0.51	0.44	0.44	0.44	0.44	0.33	0.26	0.26
320	0.74	0.73	0.74	0.58	0.60	0.60	0.49	0.49	0.49	0.51	0.50	0.45
384	0.75	0.73	0.71	0.56	0.52	0.53	0.46	0.40	0.40	0.43	0.44	0.42
448	0.78	0.76	0.76	0.64	0.61	0.60	0.52	0.53	0.55	0.45	0.40	0.39
512	0.75	0.77	0.74	0.57	0.55	0.53	0.50	0.51	0.51	0.45	0.38	0.38
576	0.81	0.78	0.80	0.64	0.61	0.64	0.53	0.48	0.47	0.51	0.52	0.52
640	0.82	0.81	0.81	0.62	0.62	0.61	0.54	0.54	0.54	0.47	0.50	0.50

Tabla 3.3.9: Cocientes de los tiempos de ejecución con los diferentes métodos con distribución de trabajo entre los procesadores y B3.

	B3C	B3B	B3CB
128	0.41	0.37	0.37
192	0.63	0.53	0.60
256	0.49	0.49	0.47
320	0.58	0.58	0.57
384	0.55	0.52	0.51
448	0.59	0.57	0.57
512	0.56	0.55	0.54
576	0.62	0.59	0.60
640	0.61	0.61	0.61
total	0.56	0.53	0.54

Tabla 3.3.10: Media de los cocientes de los tiempos de ejecución con los diferentes métodos con distribución de trabajo entre los procesadores y B3, para los distintos tamaños de matriz variando el número de procesadores.

	B3C	B3B	B3CB
2	0.75	0.75	0.75
4	0.55	0.53	0.52
6	0.48	0.45	0.46
8	0.45	0.41	0.42
total	0.56	0.53	0.54

Tabla 3.3.11: Media de los cocientes de los tiempos de ejecución con los diferentes métodos con distribución de trabajo entre los procesadores y B3, para los distintos tamaños del sistema variando el tamaño de la matriz.

	1	2	4	6	8
128	4.64	1.58	0.63	0.60	0.43
192	9.68	5.66	2.70	1.72	1.72
256	20.00	11.41	5.00	5.50	3.51
320	34.29	17.88	11.19	7.96	7.24
384	54.06	28.43	16.41	11.53	10.91
448	78.63	41.36	24.38	19.49	14.81
512	111.26	57.61	31.06	25.84	18.42
576	148.37	79.43	47.62	32.75	31.76
640	195.80	102.08	57.97	46.25	39.84

Tabla 3.3.12: Tiempos de ejecución en segundos y por barrido con B3M.

	1	2	4	8
128	0.79	0.68	0.40	0.64
192	0.55	0.63	0.59	0.52
256	0.50	0.56	0.42	0.47
320	0.45	0.45	0.48	0.45
384	0.40	0.36	0.39	0.39
448	0.35	0.34	0.29	0.29
512	0.26	0.25	0.21	0.16
576	0.27	0.31	0.32	0.25
640	0.27	0.28	0.27	0.24

Tabla 3.3.13: Cociente de los tiempos de ejecución de B3M entre PICBB1.

En la tabla 3.3.14 se muestran los Mflops obtenidos con B3M para distintos tamaños de matrices y de sistema y usando tamaño de bloque óptimo. Las prestaciones son similares a las que se muestran en [48] para otros problemas de algebra lineal. La ganancia respecto a PICB1 es mayor que la que se muestra en la tabla 3.3.13 (comparar con la tabla 3.2.8) ya que los métodos que no trabajan por bloques tienen un coste por barrido de $3n^3$ flops, y los que trabajan por bloques lo tienen de $4n^3$ flops.

	1	2	4	6	8
128	1.80	5.27	13.30	13.90	19.22
192	2.92	4.99	10.45	16.45	16.39
256	3.35	5.88	13.39	12.19	19.08
320	3.82	7.32	11.71	16.45	18.08
384	4.18	7.96	13.79	19.64	20.74
448	4.57	8.69	14.74	18.44	24.27
512	4.82	9.31	17.28	20.77	29.13
576	5.15	9.62	16.05	23.33	24.06
640	5.35	10.27	18.08	22.67	26.31

Tabla 3.3.14: Mflops obtenidos con B3M con tamaño de bloque óptimo.

3.4 Comparación con LAPACK

Puede ser interesante comparar los tiempos de ejecución obtenidos con nuestros algoritmos con los que se obtienen utilizando las rutinas de LAPACK. Hay que tener en cuenta que el coste de la rutina de LAPACK para calcular los valores propios de una matriz simétrica es $\frac{4}{3}n^3 + O(n^2)$, por lo que los tiempos obtenidos con LAPACK serán claramente mejores que los que se obtienen con el mejor de nuestros algoritmos (B3M). Por tanto, el análisis de los tiempos obtenidos con LAPACK y su comparación con los obtenidos con B3M será interesante para comprobar si la paralelización que se ha hecho en B3M es eficiente.

En la tabla 3.4.15 se muestran los tiempos de ejecución (en segundos) cuando se obtienen los valores propios de una matriz densa, simétrica con valores reales, con números generados aleatoriamente entre -10 y 10, y con distinto número de procesadores. Y en la tabla 3.4.16 se muestran los tiempos de ejecución cuando lo que se calculan son los valores y los vectores propios.

En la tabla 3.4.17 mostramos los cocientes de los tiempos de ejecución obtenidos con B3M con un barrido y los obtenidos con LAPACK (tablas 3.3.12 y 3.4.15). Se observa que el cálculo de los valores propios con LAPACK es más rápido que con B3M. Como para calcular los valores propios con B3M se necesitan entre 8 y 10 barridos (con los tamaños de matriz que hemos experimentado) y los cocientes de la tabla 3.4.17

	1	2	4	6	8
128	1.75	1.03	0.69	0.65	0.55
192	4.54	2.64	1.70	1.58	1.31
256	9.30	5.31	3.40	3.14	2.59
320	16.81	9.48	5.96	5.47	4.41
384	26.89	15.65	9.72	8.81	7.02
448	42.35	24.09	14.89	13.44	10.69
512	61.01	34.79	21.55	19.35	15.52
576	85.54	48.40	29.93	26.79	21.62
640	115.32	65.49	40.02	35.93	29.10

Tabla 3.4.15: Tiempo de ejecución, en segundos, en el cálculo de valores propios usando LAPACK.

	1	2	4	6	8
128	5.65	3.38	2.24	2.01	1.77
192	17.86	10.33	6.65	5.68	4.85
256	41.73	23.96	14.74	12.67	10.46
320	80.42	46.28	27.58	23.11	19.82
384	135.83	78.47	45.70	37.69	32.66
448	218.72	124.21	73.77	60.81	51.84
512	321.43	183.95	107.32	89.05	75.55
576	456.62	262.34	152.42	123.22	107.99
640	615.61	354.69	203.22	167.80	143.45

Tabla 3.4.16: Tiempo de ejecución, en segundos, en el cálculo de valores y vectores propios usando LAPACK.

están casi siempre entre 1.5 y 2, obtenemos que el cálculo de los valores propios con B3M es entre 12 y 20 veces más lento que con LAPACK. El resultado no es extraño teniendo en cuenta que el coste teórico del algoritmo de LAPACK es $\frac{4}{3}n^3$ y el de B3M es $\log\ n\ 3n^3$. Además, en general, la ganancia que se obtiene con LAPACK respecto a B3M disminuye al aumentar el número de procesadores, con lo se confirma una de las principales ventajas del método de Jacobi, que es lo adecuado que es para obtener buenos rendimientos en paralelo. Además, cuando se calculan los valores y los vectores propios con LAPACK los tiempos de ejecución son entre 3 y 5 veces los que se obtiene cuando se calculan sólo los valores propios (tablas 3.4.15 y 3.4.16), mientras que el cálculo de los valores y vectores propios con algoritmos del tipo de Jacobi es algo menos que el doble de lento que el cálculo de los valores propios (tablas 1.8.1 y 1.8.2), con lo que LAPACK es alrededor de 7 veces más rápido que un algoritmo de Jacobi del tipo B3M en el cálculo de los valores y vectores propios. Esto hace pensar que los métodos tipo Jacobi pueden ser competitivos con otro tipo de algoritmos basados en la reducción a forma tridiagonal en el caso en que la matriz esté muy cercana a la diagonal (en cuyo caso el número de barridos necesarios en los métodos de Jacobi es muy pequeño [1]) y en el caso en que se utilicen muchos procesadores.

	1	2	4	6	8
128	2.65	1.22	0.91	0.92	0.78
192	2.13	2.14	1.59	1.09	1.31
256	2.15	2.15	1.47	1.75	1.36
320	2.04	1.89	1.88	1.46	1.64
384	2.01	1.82	1.69	1.31	1.55
448	1.86	1.72	1.64	1.45	1.39
512	1.82	1.66	1.47	1.34	1.19
576	1.73	1.64	1.59	1.22	1.47
640	1.70	1.56	1.45	1.29	1.37

Tabla 3.4.17: Cociente entre los tiempos de ejecución de un barrido de B3M, y el del cálculo de los valores propios con LAPACK.

3.5 Conclusiones

En este capítulo hemos estudiado métodos de Jacobi para el Problema Simétrico de Valores Propios en el multiprocesador Alliant FX/80.

Se han comparado métodos que trabajan por bloques con otros que no trabajan por bloques, obteniéndose unos resultados mucho mejores con los métodos que trabajan por bloques cuando aumentan los tamaños de las matrices y el número de procesadores.

El método por bloques propuesto en el capítulo 1 para Monoprocesadores es válido también en Multiprocesadores con Memoria Compartida si se deja la explotación del paralelismo a BLAS 3, pero en algunos casos se pueden obtener mejores prestaciones dividiendo el trabajo entre los procesadores y usando una versión secuencial de BLAS 3 en cada procesador [31], por lo que se ha adecuado dicho método a las características del Multiprocesador en estudio [6, 103, 104], obteniendo distintas versiones que se diferencian en la manera de distribuir el trabajo entre los procesadores (de forma cíclica en B3C, balanceando el trabajo en B3B, y con una distribución cíclica balanceada en B3CB) y siendo la versión más eficiente de las implementadas aquella en la que se utiliza la parte triangular superior de la matriz como auxiliar para evitar copias de datos tras el uso de la rutina **dgemm** de BLAS 3.

Las prestaciones que se obtienen son similares a las obtenidas en [1, 48] con otros problemas de algebra lineal. En la comparación con LAPACK se comprueba que el método de Jacobi es muy adecuado para programación paralela, acercándose su tiempo de ejecución al de LAPACK al aumentar el número de procesadores, y también cuando se calculan valores y vectores propios.

El principal inconveniente con los métodos por bloques es que no sabemos a priori el tamaño de bloque óptimo, aunque un tamaño de bloque pequeño (entre 8 y 32) da buenos resultados, pareciendo lo más adecuado utilizar un tamaño de bloque alrededor de 16 cuando el tamaño de la matriz es grande.

Capítulo 4

ALGORITMOS DE JACOBI EN MULTICOMPUTADORES

En este capítulo analizaremos dos métodos de Jacobi en Multicomputadores, uno para topología de anillo y otro para topología de malla. Estos métodos no explotan la simetría, con lo que la máxima eficiencia teórica que se puede alcanzar con ellos es del 50%. El objetivo principal de esta tesis es el estudio de la explotación de la simetría en multicomputadores en métodos de Jacobi paralelos para el Problema Simétrico de Valores Propios, por lo que este capítulo está dedicado a mostrar las técnicas tradicionales en el diseño de métodos de Jacobi en multicomputadores y a resaltar el inconveniente principal de estas técnicas tradicionales (la no explotación de la simetría) para en capítulos posteriores estudiar cómo es posible solventar ese problema para obtener una eficiencia teórica del 100%.

4.1 Introducción

El método de Jacobi ha mostrado tener un alto grado de paralelismo intrínseco, lo que motivó que fuese utilizado en los primeros trabajos, de comienzos de los 70, sobre algoritmos paralelos de cálculo de valores propios o valores singulares [85]. Sin embargo, es a partir de principios de los ochenta [19] cuando empiezan a aparecer de manera ininterrumpida trabajos sobre el método de Jacobi en paralelo en multiprocesadores con memoria distribuida o con memoria compartida y procesadores sistólicos.

Los diferentes métodos de Jacobi se diferencian en el orden en que se eligen los elementos no diagonales para anularlos.

Como ya vimos en el capítulo 1, en el método clásico (algoritmo 1.1.1) se elige el elemento mayor en valor absoluto entre los no diagonales. Esto produce que el número de anulaciones para que converja sea menor que en otros métodos, pero que cada anulación sea más costosa al tener que obtener previamente el mayor (en valor absoluto) de $\frac{n(n-1)}{2}$ elementos. El método clásico parece poco apropiado para programación paralela, pues este máximo que se calcula antes de cada anulación habría que calcularlo también de forma distribuida con el consiguiente costo de las comunicaciones. Hay, sin embargo, algunos estudios sobre modificaciones del método clásico que sugieren la posibilidad de hacerlo apropiado para multiprocesadores [69]. Además, se sabe que el método clásico no es el mejor método secuencial.

Por todo esto, parece más apropiado realizar sucesivos barridos de modo que en cada barrido se anule una vez cada elemento no diagonal, con lo que en cada barrido se realizarían $\frac{n(n-1)}{2}$ anulaciones. Cuando en un barrido se ha anulado un elemento en el siguiente barrido ese elemento puede volver a ser distinto de cero, con lo que es necesario realizar múltiples barridos para que el método converja.

Han sido propuestas multitud de ordenaciones para la elección de los elementos a anular. Para algunas de ellas se ha demostrado la convergencia del método bajo ciertas condiciones [41, 42, 75, 77, 87], y para otras no. Algunas son más apropiadas que otras para programación en Memoria Distribuida o para una determinada topología.

Notación:

En el resto del trabajo nos referiremos a los algoritmos que se estudian por medio de la notación que aquí se indica. El nombre de cada algoritmo estará formado por un máximo de 16 letras AaBbCcDdEeFfGgHh; siendo Aa=Ja, indicando que es un método de **Ja**cobi; Bb=Es ó Ge, indicando que se trata de un algoritmo para el Problema de Valores Propios Simétrico **Es**tándar o **Ge**neralizado, respectivamente; Cc=Se ó Pa, indicando que se trata de un algoritmo **Se**cuencial o **Pa**ralelo, respectivamente; Dd=Si ó Ns, indicando que se trata de un algoritmo que explota la **Si**metría o que **N**o explota la **s**imetría, respectivamente; Ee=Fi, Pi, Rr ó Eb, si el algoritmo es secuencial, indicando que usa la ordenación por **Fi**las, **P**ar-**i**mpar, **R**ound-**r**obin, o de **Eb**erlein, respectivamente; si el algoritmo es paralelo, Ee=An, Hi ó Ma, indica que es un algoritmo para **An**illo, **Hi**percubo o **Ma**lla, respectivamente; Ff=Co, Cu, An, Ma ó Pl, que el almacenamiento de los datos en los procesadores se hace por **Co**lumnas, submatrices **Cu**adradas, **An**tidiagonales, **Ma**rcos o **Pl**egaminento, respectivamente; y Gg=Pi, Rr ó Eb, que se usa la ordenación **P**ar-**i**mpar, **R**ound-**r**obin, o de **Eb**erlein, respectivamente. De este modo, los nombres de los algoritmos secuenciales constarán de un máximo de 10 letras: JaBbSeDdEe.

Por último, estos nombres pueden contener en algunos casos otros dos caracteres al final (Ff en los algoritmos secuenciales y Hh en los paralelos) dando información adicional sobre las características del algoritmo, como puede ser B1 ó B3, que indicarán que se está usando **B**las **1** o **B**las **3**.

Si en algún momento se varía esta notación se indicará en el lugar apropiado.

4.1.1 Nociones sobre la paralelización de métodos de Jacobi

Para analizar las posibilidades de paralelización de los métodos de Jacobi habrá que desglosar el algoritmo en los distintos procedimientos para determinar qué partes se pueden paralelizar y las necesidades de comunicación.

Un esquema del método utilizando una ordenación cíclica podría ser el del algoritmo 1.1.2.

Para la paralelización del procedimiento de cálculo de los parámetros hay que tener en cuenta que dadas dos rotaciones de Givens Q_1 y Q_2 que anulan elementos a_{ij}, a_{ks}, de índices disjuntos, $\{i,j\}\bigcap\{k,s\} = \emptyset$, el cálculo de sus parámetros se puede hacer en paralelo puesto que dependen de elementos distintos de A.

También es posible explotar el paralelismo existente en los cálculos asociados a las actualizaciones de la matriz A, $A_{l+1} = Q_l A_l Q_l^t$. Por ejemplo, si $n = 4$ y

$$Q_1 = \begin{bmatrix} \hat{Q}_1 & 0 \\ 0 & I \end{bmatrix} \quad , \quad Q_2 = \begin{bmatrix} I & 0 \\ 0 & \hat{Q}_2 \end{bmatrix} \tag{4.1.1}$$

donde $\hat{Q}_1$, y $\hat{Q}_2$ representan rotaciones de Givens a nivel 2×2, entonces

$$Q_1 Q_2 = Q_2 Q_1 = \begin{bmatrix} \hat{Q}_1 & 0 \\ 0 & \hat{Q}_2 \end{bmatrix} \quad . \tag{4.1.2}$$

La actualización de A mediante las correspondientes transformaciones de semejanza vendría dada por

$$\begin{bmatrix} \hat{Q}_1 & 0 \\ 0 & \hat{Q}_2 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \hat{Q}_1^t & 0 \\ 0 & \hat{Q}_2^t \end{bmatrix} = \begin{bmatrix} \hat{Q}_1 A_{11} \hat{Q}_1^t & \hat{Q}_1 A_{12} \hat{Q}_2^t \\ \hat{Q}_2 A_{21} \hat{Q}_1^t & \hat{Q}_2 A_{22} \hat{Q}_2^t \end{bmatrix} \quad . \tag{4.1.3}$$

Se observa que la premultiplicación por $\hat{Q}_i$ afecta sólo a un par de filas y la postmultiplicación por $\hat{Q}_i^t$ sólo a un par de columnas. Por tanto, estos cálculos pueden realizarse en paralelo si la matriz A se distribuye de forma adecuada. Una posibilidad es distribuir la matriz de manera que cada procesador contenga un par de columnas.

Para la actualización de A hay que difundir los parámetros de las rotaciones pues todos los procesadores necesitarán de ellos para actualizar los elementos que contienen de la matriz.

También se necesitará de un movimiento de datos entre los procesadores para obtener las sucesivas agrupaciones de datos que vendrán determinadas por la ordenación que se use y por el esquema de almacenamiento de los datos.

4.2 En anillo

La ordenación que utilizaremos para obtener un método de Jacobi paralelo en un anillo de procesadores sin explotar la simetría será la **par-impar**, aunque de manera similar se podrían diseñar algoritmos usando alguno de los otros órdenes de Jacobi analizados en el capítulo 1.

Mostraremos la versión de la ordenación par-impar que utilizamos, por medio de un ejemplo con $n = 8$. Si partimos del conjunto inicial de pares $\{(0,1),(2,3),(4,5),(6,7)\}$, se formarán los conjuntos de pares:

$$
\begin{array}{ll}
\text{k=1} & \{(0,1),(2,3),(4,5),(6,7)\} \\
\text{k=2} & \{(0,3),(2,5),(4,7),[6,1]\} \\
\text{k=3} & \{(0,5),(2,7),(4,6),(3,1)\} \\
\text{k=4} & \{(0,7),(2,6),[4,3],(5,1)\} \\
\text{k=5} & \{(0,6),(2,4),(5,3),(7,1)\} \\
\text{k=6} & \{(0,4),[2,5],(7,3),(6,1)\} \\
\text{k=7} & \{(0,2),(7,5),(6,3),(4,1)\} \\
\text{k=8} & \{[0,7],(6,5),(4,3),(2,1)\}
\end{array}
\tag{4.2.1}
$$

Si suponemos que se dispone de un anillo con 4 procesadores y que el procesador P_i ($i = 0, \ldots, 3$) contiene en cada paso las columnas correspondientes al i-ésimo par del correspondiente conjunto de Jacobi, se necesitará sólo comunicación unidireccional. En cada barrido hay n pasos pues en los pasos impares se hacen $\frac{n}{2}$ anulaciones y en los pares $\frac{n}{2} - 1$ (no se anulan los elementos correspondientes a pares entre corchetes).

Un algoritmo general que permite calcular estos conjuntos en el caso en que n es par y p representa el número de procesadores sería el siguiente. Denotamos por $q_{i,j,k}$ al valor del j-ésimo elemento del i-ésimo par en la k-ésima iteración, con $i = 0, \ldots, \frac{n}{2} - 1$, $j = 1, 2$ y $k = 1, \ldots, n$; y suponemos que se conocen los valores correspondientes a $k = 1$, $q_{i,j,1}$.

Algoritmo 4.2.1 *Generación de índices en la ordenación par-impar.*

SI $k = 1$
  PARA $i = 0, \ldots, \frac{n}{2} - 1$
    $q_{i,2,k+1} = q_{(i+1)\ mod\ p,2,k}$
  FINPARA
EN OTRO CASO SI $k = n$
  PARA $i = 0, \ldots, \frac{n}{2} - 1$
    $q_{i,1,k+1} = q_{(i+1)\ mod\ p,1,k}$
  FINPARA
EN OTRO CASO
  PARA $i = 0, \ldots, (n - k - 1)\ div\ 2 - 1$
    $q_{i,2,k+1} = q_{i+1,2,k}$
  FINPARA
  $q_{(n-k-1)\ div\ 2,2,k+1} = q_{(n-k-1)\ div\ 2+1,1,k}$
  PARA $i = (n - k - 1)\ div\ 2 + 1, \ldots, \frac{n}{2} - 2$
    $q_{i,1,k+1} = q_{i+1,1,k}$

FINPARA
$q_{\frac{n}{2}-1,1,k+1} = q_{0,2,k}$
FINSI

Hay que hacer notar que el i-ésimo par de la k-ésima iteración vendría dado por $(q_{i,1,k}, q_{i,2,k})$. Con este algoritmo sólo se modifica un elemento de cada par, quedando los elementos que no se modifican con el mismo valor que tenían en el paso anterior. Al acabar un barrido, k volvería a tomar el valor uno.

En el método de Jacobi secuencial para matrices simétricas se puede sacar provecho de la simetría de la matriz actualizando sólo la parte triangular superior obteniéndose un coste de $3n^3$ flops por barrido. Esto no es tan sencillo en métodos de Jacobi paralelos, en particular, con el esquema de distribución de los datos por columnas (o por bloques consecutivos de columnas) se necesita de la actualización de toda la matriz y no sólo de la parte triangular superior o inferior, ya que elementos simétricos se encuentran en distintos procesadores. Por esto, la máxima eficiencia que se puede alcanzar en el cálculo de los valores propios utilizando este esquema es del 50%, y en el cálculo de los valores y vectores propios del 66%.

4.2.1 Esquema del algoritmo

Distribución de los datos

Suponemos que se dispone de un sistema multiprocesador con una topología de anillo unidireccional constituido por p procesadores, numerados de 0 a $p-1$, con p par. La matriz inicial, cuyos valores propios se quiere calcular, es de tamaño $n \times n$ con n par, y ha sido designada como A y sus filas y columnas numeradas de 0 a $n-1$. Supondremos además que $n = 2bp$ siendo b un número natural.

Utilizaremos una distribución de la matriz por bloques de dos columnas

$$A = [A_0, A_1, \ldots, A_{\frac{n}{2}-1}] \tag{4.2.2}$$

donde inicialmente cada bloque A_i contiene las columnas $2i$ y $2i+1$ de A, y está contenido en el procesador $i\ div\ b$.

De este modo, el procesador P_i contiene b bloques numerados localmente desde 0 a $b-1$, que serán los bloques $bi,\ bi+1, \ldots,\ b(i+1)-1$ de la matriz A.

Diseño del algoritmo

Para la implementación del algoritmo hay que tener en cuenta que se realizan barridos, anulando en cada barrido los elementos no diagonales, mientras no se alcance una cota determinada de $off(A)$.

Cada barrido constará de un número fijo de pasos (con el método par-impar n pasos) y, en cada paso, cada procesador tendrá que calcular los parámetros de las rotaciones asociadas a los pares correspondientes a las columnas que contiene, difundir dichos parámetros a los demás procesadores, actualizar la parte de la matriz almacenada en el procesador y transferir columnas de manera que la distribución de los índices corresponda al siguiente conjunto de pares de índices de la ordenación elegida (en este caso la par-impar). Además, al final de cada barrido, todos los procesadores intervendrán en el cálculo de $off(A)$ para determinar si el método converge. Para calcular el valor de $off(A)$ cada procesador debe calcular la parte correspondiente a las columnas que contiene, y compararla con $\frac{COTA}{p}$. Cuando en un procesador el valor local de $off(A)$ es menor que $\frac{COTA}{p}$, pone un testigo a 1 y lo envía a través del anillo. En otro caso el testigo se pone a 0. El algoritmo acaba cuando el testigo recorre dos veces el anillo conservando el valor 1.

Código del algoritmo paralelo

Supongamos que se quiere calcular los valores propios y que tenemos los datos distribuidos en los procesadores tal como hemos indicado. Es decir, el bloque A_i contiene las columnas $2i$ y $2i+1$, y está situado en el procesador $i\ div\ b$. En todos los procesadores se ejecutará el mismo código.

Algoritmo 4.2.2 *Esquema del método de Jacobi en un anillo de procesadores, sin explotar la simetría.*

```
EN PARALELO para r = 0,1,...,p-1 en P_r:
  REPETIR
    PARA i = 1,...,n
      calcular rotaciones (i,r)
      difundir rotaciones (r)
      actualizar matriz (i,r)
      transferir columnas (r)
    FINPARA
  HASTA off(A) < COTA
```

Los procedimientos utilizados en este algoritmo tienen los siguientes significados.

Algoritmo 4.2.3 *Cálcular rotaciones* (i, r).

```
SI i mod 2 = 1
  PARA j = 0,...,b-1
    calcular rotación correspondiente a los índices del par (br+j)-ésimo
  FINPARA
EN OTRO CASO
  PARA j = 0,...,b-1
    calcular rotación correspondiente a los índices del par (br+j)-ésimo,
      tomando c = 1 y s = 0 para el par ((n-i) div 2)-ésimo
```

FINPARA
FINSI

Algoritmo 4.2.4 *Difundir rotaciones* (r).
PARA $j = 1, \ldots, p-1$
SI $r \ mod \ 2 = 0$
enviar parámetros a $P_{(r-1) \ mod \ p}$
recibir parámetros de $P_{(r+1) \ mod \ p}$
EN OTRO CASO
recibir parámetros de $P_{(r+1) \ mod \ p}$
enviar parámetros a $P_{(r-1) \ mod \ p}$
FINSI
FINPARA

Se puede observar que en el paso $j = 1$ cada procesador envía los parámetros que ha calculado, y en los siguientes envía los últimos que ha recibido. Así, después de $p-1$ pasos todos los procesadores disponen de los parámetros de todas las rotaciones.

Algoritmo 4.2.5 *Actualizar matriz* (i, r).
PARA $m = 0, \ldots, b-1$
PARA $q = 0, \ldots, \frac{n}{2} - 1$

$$\begin{bmatrix} a_{u0} & a_{u1} \\ a_{v0} & a_{v1} \end{bmatrix} = \begin{bmatrix} c_q & s_q \\ -s_q & c_q \end{bmatrix} \begin{bmatrix} a_{u0} & a_{u1} \\ a_{v0} & a_{v1} \end{bmatrix} \begin{bmatrix} c_{br+m} & -s_{br+m} \\ s_{br+m} & c_{br+m} \end{bmatrix}$$

FINPARA
FINPARA

En este algoritmo llamamos (u, v) al par q-ésimo, que vendrá determinado por el orden que se esté usando (en este caso la ordenación par-impar cuyos índices determina el algoritmo 4.2.1), y denotamos a las dos columnas de cada bloque 2×2 como $a_{.0}$ y $a_{.1}$. Después de cada actualización de la matriz (en paralelo) se hará un movimiento de columnas de acuerdo a la ordenación par-impar que se está utilizando, con lo que, en cada paso, pares de columnas correspondientes a pares de índices en la ordenación estarán en un mismo bloque de columnas, pero no realizamos un movimiento de filas, con lo que la posición que ocupan las filas correspondientes al par de índices q-ésimo vendrá dada por (u, v). De este modo, si consideramos las columnas agrupadas en b bloques, cada uno de ellos de 2 columnas, y numerados en cada procesador de 0 hasta $b-1$, en el algoritmo 4.2.5 se utiliza el índice m para recorrer los bloques de pares de columnas, y q se utiliza para recorrer los $\frac{n}{2}$ pares de índices de la ordenación en el paso i-ésimo, y obtener el par (u, v) de filas (y columnas) correspondientes a ese par de índices, con lo que la matriz 2×2 de columnas $a_{.0}$ y $a_{.1}$ en el bloque m se actualiza premultiplicando por los parámetros correspondientes al q-ésimo par de índices y postmultiplicando por los correspondientes al par de índices $(br+m)$-ésimo.

Algoritmo 4.2.6 *Transferir columnas* (r).

SI $r\ mod\ 2 = 0$
  enviar columna a $P_{(r-1)\ mod\ p}$
  recibir columna de $P_{(r+1)\ mod\ p}$
EN OTRO CASO
  recibir columna de $P_{(r+1)\ mod\ p}$
  enviar columna a $P_{(r-1)\ mod\ p}$
FINSI

Aquí habría también transferencias de columnas entre bloques consecutivos, determinadas por el algoritmo de actualización de índices (algoritmo 4.2.1), pero sólo se realiza transferencia entre procesadores distintos cuando se transfieren columnas del bloque cero de cada procesador.

4.2.2 Costes teóricos

Usaremos el flop (tiempo necesario para ejecutar una operación en coma flotante) como unidad de medida para el tiempo aritmético.

En cada paso del algoritmo, cada procesador calcula b rotaciones de Givens, siendo $b = \frac{n}{2p}$, con un coste de $11b$ flops y a continuación actualiza $\frac{bn}{2}$ submatrices 2×2, lo que requiere $12nb$ flops. Además, al finalizar cada barrido, el cálculo de $off(A)$ tiene un coste de $\frac{n^2}{p} - \frac{n}{p}$ flops. Por tanto, el coste aritmético por barrido, despreciando los términos de menor orden, viene dado por

$$T_a = \frac{6n^3}{p} + \frac{13n^2}{2p} \qquad flops. \tag{4.2.3}$$

Para evaluar el tiempo de comunicación suponemos que el envío de n datos desde un procesador a un procesador vecino cuesta un tiempo $\beta + n\tau$, donde β representa el tiempo de establecimiento de la señal y τ el requerido para enviar un dato (trabajaremos con números reales en doble precisión). Además, supondremos a lo largo de todo el trabajo que los enlaces son simples (un procesador sólo puede enviar o recibir en un momento dado por un único enlace).

El coste de las comunicaciones en cada paso, utilizando los procedimientos **difundir rotaciones** y **transferir columnas** descritos en la subsección anterior será: $(p-1)(2\beta + 4b\tau)$ para la difusión de los parámetros de las rotaciones, $2\beta + 2n\tau$ para la transferencia de columnas, y al final de cada barrido $4p(\beta + \tau)$ para el cálculo de $off(A)$.

Por tanto, el coste de las comunicaciones por barrido viene dado por

$$T_c = 2n(p+1)\beta + 4n^2\tau \tag{4.2.4}$$

donde hemos despreciado los términos de orden inferior.

Como el coste aritmético por barrido del algoritmo secuencial puede ser aproximado por $T_1 = 3n^3$ flops, podemos obtener la siguiente cota superior de la eficiencia

$$E_p = \frac{T_1}{p(T_a + T_c)} \leq \frac{T_1}{pT_a} \leq \frac{1}{2} \quad . \tag{4.2.5}$$

El algoritmo estudiado se puede modificar ligeramente para implementarlo sobre un hipercubo. La distribución de los datos se hará según el esquema utilizado para un anillo pero considerando el anillo inmerso en el hipercubo. La única diferencia será el procedimiento **difundir rotaciones**, que en este caso será una difusión en la que se utilizan las características del hipercubo, con lo que tendrá un coste de $2\log_2 p\beta + 4(p-1)b\tau$. Los costes del algoritmo sobre hipercubo y sobre anillo sólo difieren en el término que aparece multiplicado por β, que viene dado en el caso de hipercubo por $2n(1+\log p)$. Por tanto el algoritmo sobre hipercubo debe tener, teóricamente, mayor eficiencia que el de anillo para valores de $p > 2$.

Si se quiere calcular también los vectores propios habrá que acumular las rotaciones haciendo la multiplicación $Q_k Q_{k-1} \ldots Q_2 Q_1$, y esto se puede hacer almacenando todos los procesadores la matriz donde se acumulan las rotaciones por filas en el sistema de procesadores, es decir almacenando cada procesador $\frac{n^2}{p}$ datos, lo que representa $\frac{6n}{p}$ flops por actualización, y un coste adicional por barrido del orden de $\frac{3n^3}{p}$ flops. De cualquier modo, como lo que nos interesa en este trabajo es la explotación de la simetría, y la acumulación de las rotaciones se hace de manera idéntica tanto si se explota la simetía como si no se explota, no analizaremos (en general y salvo que se diga lo contrario) el cálculo de los vectores propios.

4.2.3 Resultados experimentales

Hemos implementado el algoritmo para anillo unidireccional (JaEsPaNsAnCoPi) en un PARSYS SN-1040 basado en Transputers T800, un iPSC/2 y un iPSC/860.

Las implementaciones se han hecho asignando a cada procesador b bloques consecutivos, siendo b par y $pb = \frac{n}{2}$ y por tanto las implementaciones servirán para valores de n múltiplos de cuatro. Esto no es un inconveniente pues, en otro caso, se pueden añadir las filas y columnas de ceros, con unos en la diagonal, que hagan falta.

Los resultados que se muestran han sido obtenidos con programas en C y primitivas de comunicación. Corresponden en todos los casos, y a lo largo de todo el trabajo mientras no se diga lo contrario, a resultados obtenidos calculando sólo los valores propios. Se han utilizado matrices densas de distintos tamaños generadas aleatoriamente, con datos en doble precisión con valores entre -10 y 10.

tamaños:	64	128	192	256	320
JaEsPaNsAnCoPi 2	3.75	29.12	97.12	226.12	439.77
JaEsPaNsAnCoPi 4	1.95	14.69	49.37	116.12	219.66
JaEsPaNsAnCoPi 8	1.08	7.73	24.90	58.14	112.18
JaEsPaNsAnCoPi 16	0.66	4.23	13.60	31.10	59.71
JaEsSeSiFi	3.46	27.11	91.49	217.91	424.38

Tabla 4.2.1: Tiempos de ejecución por barrido (en segundos) del algoritmo JaEsPaNsAnCoPi. En el PARSYS SN-1040.

tamaños:	64	128	192	256	320	384	448	512
JaEsPaNsAnCoPi 2	6	46	155	367	716	1233	1957	2922
JaEsPaNsAnCoPi 4	3	26	78	185	359	620	982	1464
JaEsSeSiFi	5	42	143	342	668	1156	1838	2745

Tabla 4.2.2: Tiempos de ejecución por barrido (en segundos) del algoritmo JaEsPaNsAnCoPi. En el iPSC/2.

En la tabla 4.2.1 aparecen los resultados obtenidos en el PARSYS SN-1040 cuando se utilizan 2, 4, 8 y 16 Transputers y matrices cuadradas de tamaños 64, 128, 192, 256 y 320. Los resultados que aparecen en la tabla son tiempos de ejecución por barrido y en segundos. También figuran los tiempos obtenidos con un algoritmo secuencial que explota la simetría (JaEsSeSiFi).

En la tabla 4.2.2 aparecen los resultados obtenidos en el iPSC/2 utilizando 2 y 4 procesadores. Los resultados que aparecen en la tabla son tiempos en segundos por barrido. También figuran los tiempos obtenidos con un algoritmo secuencial que explota la simetría.

En la tabla 4.2.3 aparecen los resultados obtenidos en el iPSC/860 utilizando 2, 4, 8 y 16 procesadores. Los resultados que aparecen en la tabla son tiempos en segundos por barrido, y se ha utilizado BLAS 1 en la actualización de la matriz. También figuran los tiempos obtenidos con un algoritmo secuencial que explota la simetría.

En la tabla 4.2.4 se muestran los valores obtenidos de la eficiencia en el PARSYS SN-1040. Se observa que no se llega a una eficiencia de 0.5, tal como se deduce del estudio teórico del algoritmo. También se aprecia que al aumentar el tamaño de las matrices aumenta la eficiencia pero al aumentar el número de procesadores disminuye, debido fundamentalmente a las difusiones de los parámetros de las rotaciones.

En la tabla 4.2.5 se muestran los valores obtenidos de la eficiencia en el iPSC/2, y en la 4.2.6 las eficiencias obtenidas en el iPSC/860. Se observan los mismos fenómenos

	64	128	192	256	320	384	448	512
JaEsPaNsAnCoPiB1 2	0.99	2.14	5.48	12.00	22.20	37.77	59.03	93.16
JaEsPaNsAnCoPiB1 4	1.19	1.73	3.13	6.93	12.41	20.34	31.55	48.13
JaEsPaNsAnCoPiB1 8	0.78	1.42	2.63	4.81	7.76	12.38	18.34	27.48
JaEsPaNsAnCoPiB1 16	0.78	1.20	2.01	4.17	6.21	9.19	12.63	17.77
JaEsSeSiFiB1	0.39	1.14	3.80	9.36	20.04	35.35	60.23	88.19

Tabla 4.2.3: Tiempos de ejecución por barrido (en segundos) del algoritmo JaEsPaNsAnCoPiB1. En el iPSC/860.

	64	128	192	256	320
JaEsPaNsAnCoPi 2	0.462	0.465	0.471	0.482	0.482
JaEsPaNsAnCoPi 4	0.444	0.462	0.463	0.469	0.483
JaEsPaNsAnCoPi 8	0.402	0.438	0.459	0.468	0.473
JaEsPaNsAnCoPi 16	0.327	0.400	0.420	0.438	0.444

Tabla 4.2.4: Eficiencia del algoritmo JaEsPaNsAnCoPi. En el PARSYS SN-1040.

que en el PARSYS SN-1040.

Las eficiencias obtenidas en los iPSC son menores que las obtenidas en el PARSYS SN-1040 debido a un mayor coste de las comunicaciones en relación con el coste aritmético en los iPSC. En el iPSC/860 el uso de BLAS 1 reduce el tiempo aritmético, lo que produce un aumento del coste de las comunicaciones en relación con el coste aritmético y por tanto una reducción en la eficiencia. Además, al usar más procesadores disminuye el tamaño de los vectores sobre los que actúa **drot** en la premultiplicación de los vectores fila, habiendo por tanto un peor uso de BLAS 1 (y también una reducción en la eficiencia) al aumentar el número de procesadores.

	64	128	192	256	320	384	448	512
JaEsPaNsAnCoPi 2	0.446	0.463	0.462	0.466	0.467	0.469	0.469	0.469
JaEsPaNsAnCoPi 4	0.403	0.447	0.457	0.463	0.464	0.466	0.468	0.469

Tabla 4.2.5: Eficiencia del algoritmo JaEsPaNsAnCoPi. En el iPSC/2.

	64	128	192	256	320	384	448	512
JaEsPaNsAnCoPiB1 2	0.20	0.27	0.35	0.39	0.45	0.47	0.51	0.47
JaEsPaNsAnCoPiB1 4	0.10	0.16	0.30	0.34	0.40	0.43	0.48	0.46
JaEsPaNsAnCoPiB1 8	0.06	0.10	0.18	0.24	0.32	0.36	0.41	0.40
JaEsPaNsAnCoPiB1 16	0.03	0.05	0.12	0.14	0.20	0.24	0.30	0.31

Tabla 4.2.6: Eficiencia del algoritmo JaEsPaNsAnCoPiB1. En el iPSC/860.

4.3 En malla

La ordenación que utilizaremos para obtener un método de Jacobi paralelo en una malla abierta y cuadrada de procesadores, sin explotar la simetría, será la Round-Robin (el algoritmo será el JaEsPaNsMaCuRr). Esta ordenación nos permite utilizar una malla abierta aunque a costa de algunos movimientos adicionales de datos (comunicación en los dos sentidos en las filas y columnas de procesadores). De manera similar se podrían diseñar algoritmos usando alguno de los otros órdenes de Jacobi analizados, necesitándose con algunos de ellos que la malla se encuentre cerrada formando la topología de toro.

Al igual que con el algoritmo para anillo estudiado, la máxima eficiencia que se puede alcanzar en el cálculo de los valores propios utilizando este esquema será del 50%, y en el cálculo de los valores y vectores propios del 66%.

4.3.1 Esquema del algoritmo

Distribución de los datos

Suponemos que se dispone de un multiprocesador con memoria distribuida formado por $p = r^2$ procesadores, situados en r filas y r columnas numeradas de 0 a $r-1$. Denotaremos a cada procesador por su índice de fila y de columna en el sistema. Cada procesador P_{ij} está conectado a los procesadores $P_{i-1,j}$, $P_{i+1,j}$, $P_{i,j-1}$ y $P_{i,j+1}$, con $i-1 \geq 0$, $i+1 \leq r-1$, $j-1 \geq 0$ y $j+1 \leq r-1$.

La matriz A de tamaño $n \times n$ se divide en bloques de la forma

$$A = \begin{bmatrix} A_{00} & A_{01} & \dots & A_{0k} \\ A_{10} & A_{11} & \dots & A_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k0} & A_{k1} & \dots & A_{kk} \end{bmatrix} \tag{4.3.1}$$

con $k = r-1$. Cada bloque A_{ij} tiene dimensión $\frac{n}{r} \times \frac{n}{r}$, y se asigna al procesador P_{ij}.

Con esta distribución tendremos elementos simétricos en procesadores distintos y no será posible explotar la simetría.

Diseño del algoritmo

Igual que en el algoritmo para anillo, el método consiste en realizar sucesivos barridos hasta que se alcanza la convergencia, de acuerdo a un criterio determinado, y en cada barrido cada elemento no diagonal es anulado una vez. Cada barrido está dividido en un número fijo de pasos, que con la ordenación Round-Robin es $n-1$ si el tamaño de la matriz es $n \times n$. En el algoritmo para malla sólo los procesadores en la diagonal principal de procesadores calculan rotaciones de Givens, por lo que en esta parte se produce un mal balanceo de la carga, y difunden los parámetros que han calculado a los demás procesadores en la misma fila y columna de procesadores. Después de la difusión, la actualización de la matriz se realiza trabajando en paralelo todos los procesadores. Finalmente, se realiza una transferencia de columnas y de filas entre procesadores vecinos, para obtener la nueva agrupación de datos de acuerdo a la nueva agrupación de índices de la ordenación Round-Robin.

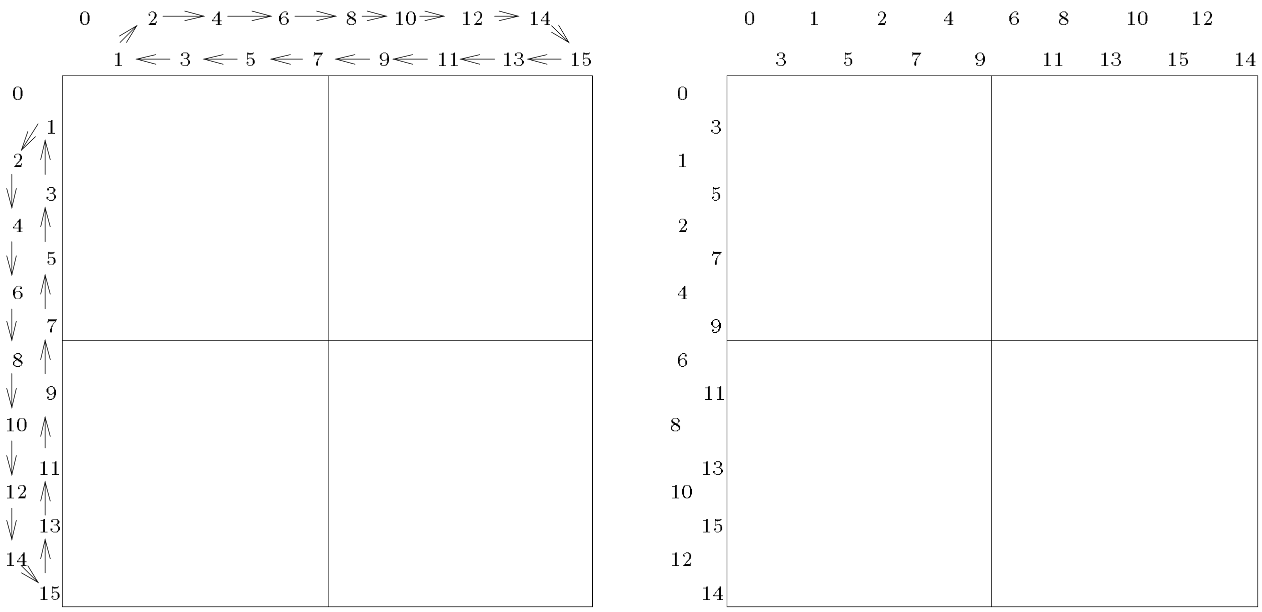
De este modo, un esquema del algoritmo sería el siguiente.

Algoritmo 4.3.1 *Esquema del método de Jacobi en una malla abierta de procesadores, sin explotar la simetría.*

```
EN PARALELO: PARA i = 0,...,r−1; j = 0,...,r−1 EN P_ij:
  REPETIR
    PARA k = 1,...,n−1
      SI i = j
        Calcular rotaciones
      FINSI
      Difundir parámetros (i,j)
      Actualizar matriz
      Transferir columnas (i,j)
      Transferir filas (i,j)
    FINPARA
  HASTA off(A) < COTA
```

En este caso el cálculo de $off(A)$ se llevará a cabo calculando cada procesador el valor de $off(A)$ de la parte de matriz que contiene, haciéndose una acumulación para obtener $off(A)$ en el procesador P_{00}, y difundiendo desde P_{00} al resto de los procesadores la información sobre si se ha alcanzado o no la convergencia.

A diferencia del algoritmo para anillo (4.2.2), en el algoritmo para malla, después de la actualización de la matriz se realiza un movimiento de datos por filas y columnas. En este movimiento de filas y columnas habrá transferencias externas entre los procesadores, y movimientos de datos en la memoria de cada procesador, de manera que después de la transferencia de datos las filas y columnas deben estar almacenadas



a) b)

Figura 4.3.1: Movimiento de filas y columnas tras un paso del algoritmo para malla, con $n = 16$ y $p = 4$: a) distribución de las filas y columnas antes del movimiento de datos, b) distribución después del movimiento de datos.

en el sistema de procesadores en el orden dado por el conjunto de pares de índices de ese paso. En la figura 4.3.1 se muestra el movimiento de datos cuando $n = 16$ y tenemos una malla de 4 procesadores, suponiendo que inicialmente los pares de índices son $\{(0,1),(2,3),(4,5),(6,7),(8,9),(10,11),(12,13),(14,15)\}$ (figura 4.3.1.a) y tras el movimiento de datos el nuevo conjunto de pares de índices es $\{(0,3),(1,5),(2,7),(4,9),(6,11),(8,13),(10,15),(12,14)\}$ (figura 4.3.1.b). De este modo, los elementos a anular en cada paso (y los elementos necesarios para calcular los parámetros de las rotaciones) se encuentran almacenados siempre en bloques de tamaño 2×2 dentro de la diagonal principal de bloques 2×2 de la memoria, con lo que no es necesario utilizar parámetros en los procedimientos de cálculo de rotaciones y actualización de la matriz que aparecen en el algoritmo 4.3.1.

Código del algoritmo paralelo

Los procedimientos usados en este algoritmo tienen los siguientes significados.

Algoritmo 4.3.2 *Calcular rotaciones (Sólo trabajan los procesadores diagonales).*

PARA $m = 0, 2, 4, \ldots, \frac{n}{r} - 2$
  Computar parámetros de rotaciones que anulan $a_{m,m+1}$ y $a_{m+1,m}$
FINPARA

En el algoritmo 4.3.2 suponemos que cada matriz A_{ij} se almacena localmente como $(a_{ij})_{i=0,1,\ldots,\frac{n}{r}-1;\ j=0,1,\ldots,\frac{n}{r}-1}$.

La difusión de los parámetros es una difusión desde procesadores en la diagonal principal a los demás procesadores en la misma fila y columna (algoritmo 2.2.11 y figura 2.2.12).

Algoritmo 4.3.3 *Actualizar matriz.*
PARA $m = 0, 1, 2, \ldots, \frac{n}{2r} - 1$
  PARA $q = 0, 1, 2, \ldots, \frac{n}{2r} - 1$

$$\begin{bmatrix} a_{2m,2q} & a_{2m,2q+1} \\ a_{2m+1,2q} & a_{2m+1,2q+1} \end{bmatrix} = \begin{bmatrix} c_m & s_m \\ -s_m & c_m \end{bmatrix} \begin{bmatrix} a_{2m,2q} & a_{2m,2q+1} \\ a_{2m+1,2q} & a_{2m+1,2q+1} \end{bmatrix} \begin{bmatrix} c_q & -s_q \\ s_q & c_q \end{bmatrix}$$

  FINPARA
FINPARA

La transferencia de columnas se realiza como en el algoritmo de comunicación entre procesadores adyacentes por filas (algoritmo 2.2.13), siendo los datos a enviar columnas de los bloques cuadrados que contienen los procesadores. La transferencia de filas se realiza de manera similar, consistiendo en una comunicación entre procesadores adyacentes por columnas.

4.3.2 Costes teóricos

En cada paso, cada procesador diagonal calcula $\frac{n}{2r}$ rotaciones de Givens con un coste de $\frac{11n}{2r}$ flops. Después, cada procesador actualiza $\frac{n^2}{4p}$ submatrices 2×2 con un coste de $\frac{6n^2}{p}$ flops. Además, después de cada barrido tendremos un coste de $\frac{2n^2}{p} + 2r - \frac{n}{p} - 2$ flops, debido al cálculo de $off(A)$.

Por tanto, el coste por barrido puede aproximarse por

$$T_a = \frac{6n^3}{p} + \left(\frac{11}{2\sqrt{p}} - \frac{4}{p} \right) n^2 \qquad flops. \tag{4.3.2}$$

En cada paso, la difusión de los parámetros de las rotaciones tiene un coste $r\beta + n\tau$, la transferencia de columnas tiene un coste $4\beta + \frac{4n}{r}\tau$, y la transferencia de filas un coste $4\beta + \frac{4n}{r}\tau$. Al final de cada barrido tenemos un coste de $2(r-1)(\beta+\tau)$ debido al cálculo de $off(A)$.

De este modo, podemos aproximar el coste por barrido de las comunicaciones como

$$T_c = (\sqrt{p}+8)n\beta + \left(1+\frac{8}{\sqrt{p}}\right)n^2\tau \quad . \tag{4.3.3}$$

Se obtiene la misma cota superior para la eficiencia obtenida para el algoritmo JaEsPaNsAnCoPi (expresión 4.2.5).

4.3.3 Resultados experimentales

Hemos implementado el algoritmo JaEsPaNsMaCuRr en el PARSYS SN-1040, el iPSC/860 y el Touchstone DELTA.

Se ha realizado el mismo tipo de experimentos que con el algoritmo JaEsPaNsAnCoPi: los programas se han hecho en C con primitivas de comunicación, y se han hecho tests con matrices densas generadas aleatoriamente con datos entre -10 y 10 y con distintos tamaños de matriz.

En la tabla 4.3.1 aparecen los resultados obtenidos en el PARSYS SN-1040 cuando se utilizan mallas de 4, 9 y 16 Transputers. Los resultados que aparecen en la tabla corresponden a tiempos de ejecución en segundos por barrido. También figuran los tiempos obtenidos con un algoritmo secuencial que explota la simetría.

	96	192	288	384
JaEsPaNsMaCuRr 4	6.92	52.12	172.65	405.04
JaEsPaNsMaCuRr 9	3.49	24.81	80.40	186.79
JaEsPaNsMaCuRr 16	2.15	14.62	46.69	107.59
JaEsSeSiFi	10.82	84.89	285.48	

Tabla 4.3.1: Tiempos de ejecución por barrido (en segundos) del algoritmo JaEsPaNsMaCuRr. En el PARSYS SN-1040.

En la tabla 4.3.2 aparecen los resultados obtenidos en el iPSC/860 utilizando 4 y 16 procesadores. Los resultados que aparecen en la tabla son tiempos en segundos por barrido, y se ha utilizado BLAS 1 en la actualización de la matriz. También figuran los tiempos obtenidos con un algoritmo secuencial que explota la simetría.

En la tabla 4.3.3 aparecen los resultados obtenidos en el Touchstone DELTA utilizando 4, 9 y 16 procesadores. Los resultados que aparecen en la tabla son tiempos en segundos por barrido, y se ha utilizado BLAS 1 en la actualización de la matriz. También figuran los tiempos obtenidos con un algoritmo secuencial que explota la simetría.

En la tabla 4.3.4 se muestran los valores obtenidos de la eficiencia en el PARSYS SN-1040. Se observa que no se llega a una eficiencia de 0.5, tal como se deduce del estudio teórico del algoritmo. Se aprecian los mismos fenómenos que con el algoritmo

	64	128	192	256	320	384	448	512
JaEsPaNsMaCuRrB1 4	0.79	1.53	3.55	7.45	13.82	23.16	36.97	54.96
JaEsPaNsMaCuRrB1 16	0.79	1.01	1.82	3.29	5.02	8.03	11.69	16.41
JaEsSeSiFiB1	0.39	1.14	3.80	9.36	20.04	35.35	60.23	88.19

Tabla 4.3.2: Tiempos de ejecución por barrido (en segundos) del algoritmo JaEsPaNsMaCuRrB1. En el iPSC/860.

	192	384	576	768	960
JaEsPaNsMaCuRrB1 4	3.01	22.76	81.29	193.60	409.18
JaEsPaNsMaCuRrB1 9	1.59	10.23	33.38	81.31	159.36
JaEsPaNsMaCuRrB1 16	1.01	6.23	18.95	44.70	92.39
JaEsSeSiFiB1	3.83	35.36	135.09	306.29	646.29

Tabla 4.3.3: Tiempos de ejecución por barrido (en segundos) del algoritmo JaEsPaNsMaCuRrB1. En el Touchstone DELTA.

para anillo: que al aumentar el tamaño de las matrices aumenta la eficiencia pero al aumentar el número de procesadores disminuye.

	96	192	288
JaEsPaNsMaCuRr 4	0.391	0.407	0.414
JaEsPaNsMaCuRr 9	0.344	0.380	0.395
JaEsPaNsMaCuRr 16	0.316	0.363	0.382

Tabla 4.3.4: Eficiencia del algoritmo JaEsPaNsMaCuRr. En el PARSYS SN-1040.

En la tabla 4.3.5 se muestran los valores obtenidos de la eficiencia en el iPSC/860, y en la 4.3.6 las eficiencias obtenidas en el Touchstone DELTA. Los resultados son similares a los obtenidos con el PARSYS SN-1040, pero se obtienen mejores prestaciones (al aumentar el tamaño de las matrices) en el iPSC/860 y en el Touchstone DELTA, debido a que estas máquinas permiten realizar comunicaciones más rápidas. Además, como ya vimos con JaEsPaNsAnCoPi, para un mismo tamaño de matriz y tamaños pequeños, se obtienen mejores eficiencias en el PARSYS SN-1040 por los motivos ya señalados. Aunque el iPSC/860 y el Touchstone DELTA están basados en el procesador i860, se obtienen mejores eficiencias en el Touchstone DELTA, principalmente al aumentar el tamaño del sistema, debido a que este permite un menor tiempo de comunicación.

	64	128	192	256	320	384	448	512
JaEsPaNsMaCuRrB1 4	0.12	0.19	0.27	0.31	0.36	0.38	0.41	0.40
JaEsPaNsMaCuRrB1 16	0.03	0.07	0.13	0.18	0.24	0.28	0.32	0.34

Tabla 4.3.5: Eficiencia del algoritmo JaEsPaNsMaCuRrB1. En el iPSC/860.

	192	384	576	768	960
JaEsPaNsMaCuRrB1 4	0.32	0.39	0.42	0.40	0.39
JaEsPaNsMaCuRrB1 9	0.27	0.38	0.45	0.42	0.45
JaEsPaNsMaCuRrB1 16	0.24	0.35	0.45	0.43	0.44

Tabla 4.3.6: Eficiencia del algoritmo JaEsPaNsMaCuRrB1. En el Touchstone DELTA.

Para analizar las prestaciones de este algoritmo cuando aumenta el número de procesadores, mostramos en la tabla 4.3.7 los Mflops por nodo obtenidos al ejecutar el programa en el Touchstone DELTA con mallas de distintos tamaños y con dimensions de la matriz alrededor de 1100. Se observa que las prestaciones por nodo disminuyen al aumentar el tamaño de la malla. Esto es normal en computación paralela, pero en este caso esta reducción se puede deber en buena parte a que en sistemas de mayor número de procesadores cada procesador trabaja con submatrices más pequeñas obteniéndose menos beneficio del uso de BLAS 1. De cualquier modo, el algoritmo para malla presenta buena escalabilidad.

4.4 Conclusiones

De la comparación de los resultados obtenidos con los dos algoritmos en el PARSYS SN-1040 (tablas 4.2.1 y 4.3.1) e iPSC/860 (tablas 4.2.3 y 4.3.2) no se puede dedudir que uno de los dos algoritmos sea mejor que el otro, aunque al aumentar el número de procesadores el algoritmo para malla se comporta mejor que el de anillo, debido a la buena escalabilidad del algoritmo de malla (tabla 4.3.7). En la tabla 4.4.8 mostramos el cociente entre los tiempos obtenidos con el algoritmo para anillo y para malla en el iPSC/860. Se observa que al aumentar el número de procesadores este cociente aumenta, lo que da idea de la mejor escalabilidad del algoritmo para malla. Que no se note esta mejora con pocos nodos es comprensible si tenemos en cuenta que en el algoritmo de malla (aun siendo escalarmente mejor) hay zonas en que la carga no está balanceada. Además, en la transferencia de datos se hace transferencia de filas y columnas y, al usar la ordenación Round-Robin, cada procesador (salvo los extremos)

PROCESADORES	1	4	9	16
Mflops/nodo	4.06	3.20	3.40	3.51
Tamaño matriz	1152	1086	1104	1088
PROCESADORES	25	36	49	64
Mflops/nodo	3.57	3.68	3.80	3.37
Tamaño matriz	1200	1248	1232	1296
PROCESADORES	81	100	121	144
Mflops/nodo	3.51	3.53	3.50	3.83
Tamaño matriz	1152	1120	1232	1152
PROCESADORES	169	196	225	256
Mflops/nodo	3.08	3.09	3.02	2.56
Tamaño matriz	1040	1120	1200	1024

Tabla 4.3.7: Mflops por nodo obtenidos con el algoritmo JaEsPaNsMaCuRrB1. En el Touchstone DELTA.

envía y recibe datos en los dos sentidos, tanto en la comunicación de filas como en la de columnas.

	64	128	192	256	320	384	448	512
4 nodos	1.50	1.13	0.88	0.85	0.89	0.87	0.85	0.87
16 nodos	0.79	1.18	1.10	1.00	1.23	1.14	1.08	1.08

Tabla 4.4.8: Cociente entre los tiempos de ejecución de los algoritmos JaEsPaNsAn-CoPiB1 y JaEsPaNsMaCuRrB1. En el iPSC/860.

Por otra parte, si hacemos un estudio teórico del speed-up escalado (speed-up cuando el número de procesadores y el tamaño del problema aumentan proporcionalmente [117, 72]), obtenemos que, si $p = kn$ siendo p el número de procesadores, n el tamaño de la matriz y k una constante, el speed-up escalado del algoritmo de anillo es

$$\frac{3k}{6 + 2k^2\beta + 4k\tau}\, n \qquad , \tag{4.4.4}$$

y el del algoritmo de malla

$$\frac{3k}{6 + k\tau}\, n \qquad , \tag{4.4.5}$$

con lo que comprobamos que el algoritmo de malla presenta mejor escalabilidad.

Capítulo 5

EXPLOTACION DE LA SIMETRIA EN ANILLO

En los algoritmos que hemos analizado en el capítulo anterior no se explota la simetría de la matriz, por lo que la eficiencia está acotada superiormente por 0.5 cuando se calculan los valores propios, y por 0.66 cuando se calculan los valores y vectores propios. Para poder explotar la simetría de las matrices y obtener de este modo algoritmos teóricamente óptimos necesitaremos almacenar elementos simétricos en un mismo procesador, con lo que no será necesaria la actualización de toda la matriz, sino sólo de la parte triangular superior o inferior. Mostraremos algunos esquemas de almacenamiento que nos permitirán explotar la simetría, dividiendo de este modo el coste aritmético por dos, aunque esto se hará a costa de empeorar las comunicaciones. Por ello, con estos esquemas de almacenamiento se debe conseguir no sólo la división por dos del tiempo aritmético, sino también que el coste de las comunicaciones no aumente considerablemente con respecto a los algoritmos que no explotan la simetría.

5.1 Esquemas de almacenamiento

En esta sección estudiaremos algunos esquemas de almacenamiento que permitirán explotar la simetría de la matriz en anillo (o topologías relacionadas como array lineal, o anillo con un host conectado con todos los procesadores, anillo inmerso en hipercubo, ...). Analizaremos tres esquemas de almacenamiento, por **codos**, por **marcos** y por **antidiagonales**.

Para diseñar algoritmos utilizando estos esquemas de almacenamiento hay que tener en cuenta la ordenación que se utiliza en el método de Jacobi cíclico, por lo que es necesario combinar el **esquema de almacenamiento**, la **ordenación** y la **topología**. Es necesario combinar estos tres factores porque cada uno de ellos influye en los demás. Por ejemplo, la ordenación que se use determinará el movimiento de datos entre dos pasos consecutivos del algoritmo, movimiento que vendrá determinado por el movimiento de índices de la ordenación, y la cantidad de comunicaciones que implica este movimiento de datos vendrá determinada por el esquema de almacenamiento, pues éste determina donde están los datos inicialmente y donde deben quedar después de la transferencia y, además, la forma en que se llevan a cabo las comunicaciones de datos viene determinada por la topología.

En las secciones sucesivas de este capítulo mostraremos cómo es posible combinar estos tres factores para diseñar algoritmos de Jacobi que exploten la simetría de la matriz en un anillo de procesadores. La diferencia en las prestaciones que se obtienen con los diferentes esquemas en anillo son mínimas, ya que con todas ellas se alcanza una eficiencia teórica del 100%, por lo que se diseñarán algoritmos que usan los esquemas de **antidiagonales** y **marcos**, y de manera similar se pueden diseñar algoritmos con el esquema por codos o con otros esquemas que permitan explotar la simetría.

Para obtener la topología óptima asociada a cada esquema de almacenamiento hay que tener en cuenta la ordenación que se use y el esquema del algoritmo, que seguirá siendo el que vimos en el capítulo anterior en un anillo de procesadores (y usaremos la ordenación par-impar), aunque habrá que determinar el funcionamiento de cada uno de los procedimientos, lo que se hará cuando estudiemos el diseño de los algoritmos que utilizan los esquemas de almacenamiento que proponemos.

5.1.1 Almacenamiento por codos

Para obtener este esquema se divide la matriz de tamaño $n \times n$ en $\frac{n}{2}$ **codos** (figura 5.1.1) que llamaremos C_i con $i = 0, \ldots, \frac{n}{2}-1$, conteniendo cada **codo** elementos de dos filas y dos columnas consecutivas. El **codo** C_i contendrá los elementos de las filas y columnas $2i$ y $2i+1$ que no pertenezcan a ninguna fila o columna j con $j = 0, \ldots, 2i-1$. Si se distribuye la matriz en los procesadores por **codos**, elementos simétricos estarán en un mismo procesador (por tanto sólo hace falta almacenar la parte triangular superior o inferior de la matriz). Para que la distribución de los datos sea balanceada agruparemos los **codos** por parejas que formarán un **bloque**, obteniendose de este modo $\frac{n}{4}$ **bloques** que llamaremos B_i, siendo $B_i = C_i \bigcup C_{\frac{n}{2}-1-i}$, con $i = 0, \ldots, \frac{n}{4}-1$ (figura 5.1.1). De este modo se agrupan el mayor y el menor **codo** en el mismo **bloque**, etc ..., y si consideramos $n = 4pb$ con p el número de procesadores y b un número natural, se pueden asignar a cada procesador b bloques consecutivos, conteniendo P_i los **bloques** B_j con $j = ib, \ldots, (i+1)b-1$, y estando el **bloque** B_i almacenado en $P_{i\ div\ p}$ (figura 5.1.1).

Topología asociada

Con este esquema la transferencia de columnas entre los **codos** no es suficiente. Por ejemplo, si suponemos una matriz 8×8, dos procesadores y distribución inicial de los datos $\{(0,1),(2,3),(4,5),(6,7)\}$, y usamos la ordenación par-impar, inicialmente la distribución de los datos en los procesadores es la que se muestra en la figura 5.1.2.a),

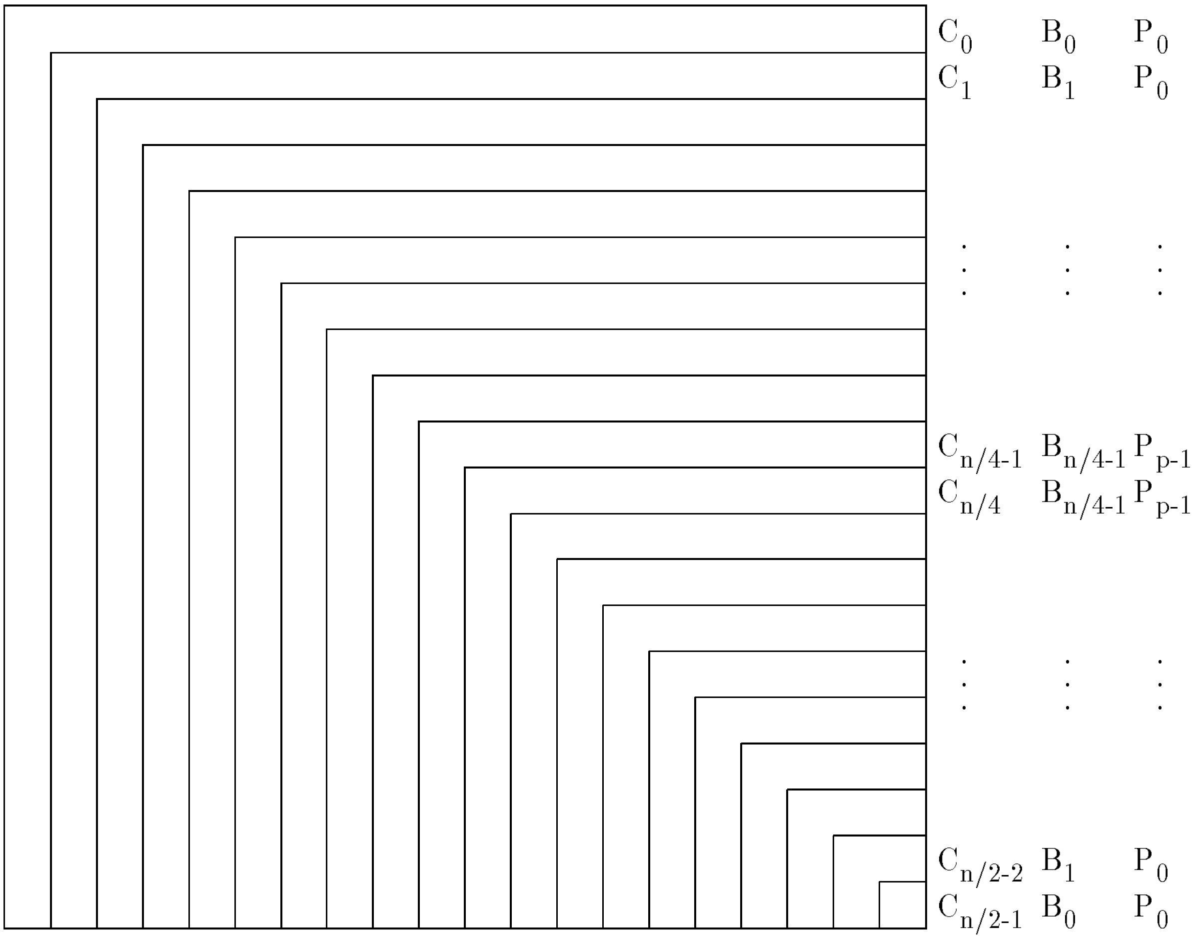


Figura 5.1.1: Esquema de almacenamiento por **codos**.

$$\begin{array}{cccccccc|c}
a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} & a_{06} & a_{07} & \\
a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & P_0 \\
\hline
a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & \\
a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & \\
a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & P_1 \\
a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & \\
\hline
a_{60} & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & \\
a_{70} & a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & P_0
\end{array}$$

a)

$$\begin{array}{cccccccc|c}
a_{00} & a_{03} & a_{02} & a_{05} & a_{04} & a_{07} & a_{06} & a_{01} & \\
a_{10} & a_{13} & a_{12} & a_{15} & a_{14} & a_{17} & a_{16} & a_{11} & P_0 \\
\hline
a_{20} & a_{23} & a_{22} & a_{25} & a_{24} & a_{27} & a_{26} & a_{21} & \\
a_{30} & a_{33} & a_{32} & a_{35} & a_{34} & a_{37} & a_{36} & a_{31} & \\
a_{40} & a_{43} & a_{42} & a_{45} & a_{44} & a_{47} & a_{46} & a_{41} & P_1 \\
a_{50} & a_{53} & a_{52} & a_{55} & a_{54} & a_{57} & a_{56} & a_{51} & \\
\hline
a_{60} & a_{63} & a_{62} & a_{65} & a_{64} & a_{67} & a_{66} & a_{61} & \\
a_{70} & a_{73} & a_{72} & a_{75} & a_{74} & a_{77} & a_{76} & a_{71} & P_0
\end{array}$$

b)

$$\begin{array}{cccccccc|c}
a_{00} & a_{03} & a_{02} & a_{05} & a_{04} & a_{07} & a_{06} & a_{01} & \\
a_{30} & a_{33} & a_{32} & a_{35} & a_{34} & a_{37} & a_{36} & a_{31} & P_0 \\
\hline
a_{20} & a_{23} & a_{22} & a_{25} & a_{24} & a_{27} & a_{26} & a_{21} & \\
a_{50} & a_{53} & a_{52} & a_{55} & a_{54} & a_{57} & a_{56} & a_{51} & \\
a_{40} & a_{43} & a_{42} & a_{45} & a_{44} & a_{47} & a_{46} & a_{41} & P_1 \\
a_{70} & a_{73} & a_{72} & a_{75} & a_{74} & a_{77} & a_{76} & a_{71} & \\
\hline
a_{60} & a_{63} & a_{62} & a_{65} & a_{64} & a_{67} & a_{66} & a_{61} & \\
a_{10} & a_{13} & a_{12} & a_{15} & a_{14} & a_{17} & a_{16} & a_{11} & P_0
\end{array}$$

c)

Figura 5.1.2: Movimiento de filas y columnas para recomponer la simetría: a) distribución inicial de los datos, b) distribución de los datos tras el movimiento de columnas, c) distribución de los datos tras la transferencia de filas.

donde elementos simétricos están en el mismo procesador, con lo que se puede explotar la simetría. Tras el movimiento de datos tendremos la nueva agrupación de índices $\{(0,3),(2,5),(4,7),(6,1)\}$, con lo que el primer **codo** debe contener elementos de las filas y columnas 0 y 3, el segundo de las filas y columnas 2 y 5, etc..., pero intercambiando sólo columnas se obtendría la distribución que se muestra en la figura 5.1.2.b), donde se ve que se ha perdido la simetría en el almacenamiento de los datos (por ejemplo, el elemento a_{31} está en P_1 pero su simétrico, a_{13}, está en P_0), por lo que es necesario un movimiento de filas de manera que se obtenga la distribución de la figura 5.1.2.c), con lo que se tienen de nuevo elementos simétricos en el mismo procesador.

Por tanto, se necesitará de un movimiento de columnas y el mismo movimiento de filas de acuerdo con la ordenación que se esté usando. De este modo se aumentaría el coste de las comunicaciones si se ejecuta el algoritmo de Jacobi, pero probablemente este coste es de menor orden que el coste aritmético, con lo que la eficiencia teórica de un algoritmo que use este esquema de almacenamiento será del 100%.

Tras el movimiento de datos (de filas y columnas) los elementos con los que se calculan las nuevas rotaciones de Givens están siempre en la misma posición, en bloques de la diagonal principal de la matriz si esta se divide en bloques 2×2.

Para determinar la topología a emplear es necesario estudiar las transferencias de filas y columnas, pues la comunicación de las rotaciones de Givens y de $off(A)$ se realizan por medio de difusiones.

La topología dependerá de la ordenación que se use además del esquema de almacenamiento. Ya que consideramos la ordenación par-impar, el **bloque** B_0 transfiere una columna de C_0 de la que deben transferirse datos a todos los demás **codos** y por lo tanto a todos los procesadores. Se necesita por tanto que un procesador esté comunicado con todos los demás. Para hacer esto de forma eficiente el **bloque** B_1 transfiere datos a B_0 y B_2, el B_2 a B_1 y B_3, etc ..., y el $B_{\frac{n}{4}-1}$ a $B_{\frac{n}{4}-2}$ y a sí mismo.

De este modo, la topología adecuada para este esquema de almacenamiento usando la ordenación par-impar sería la de la figura 5.1.3 (con otras ordenaciones pueden ser necesarias otras topologías, por ejemplo con la Round-Robin sólo se necesitaría un array lineal).

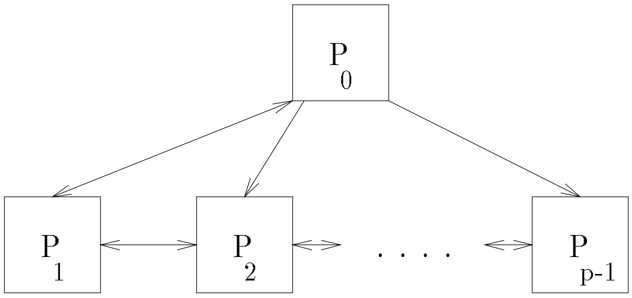


Figura 5.1.3: Topología asociada al esquema de almacenamiento por **codos** usando la ordenación **par-impar**.

Costes teóricos

Para el estudio de los costes teóricos hay que tener en cuenta que el método de Jacobi que consideramos tiene el mismo esquema que el método sin explotación de la simetría del capítulo anterior, por lo que usa la ordenación par-impar, aunque el esquema de almacenamiento es por **codos**.

Estudiamos el coste aritmético y de las comunicaciones en el cálculo de los valores propios, pues el del cálculo de los vectores propios no varía con respecto a los algoritmos que no explotan la simetría.

Para estudiar el coste aritmético, si tenemos en cuenta que en cada procesador hay b **bloques** y que por cada **bloque** se calculan dos rotaciones de Givens, en cada procesador se calculan $2b$ rotaciones de Givens, lo que se traduce en $22b$ flops. Como el número de pasos por barrido es n (con la ordenación par-impar), el coste por barrido del cálculo de las rotaciones de Givens será $22bn = \frac{11n^2}{2p}$.

En la actualización de la matriz, como cada **bloque** tiene n submatrices 2×2 de las que $n-2$ son simétricas, tendremos que actualizar $\frac{n}{2}+1$ matrices 2×2, necesitando de 24 flops para la actualización de cada una de ellas. Resulta entonces que por **bloque** se necesitan $12n + 24$ flops, por procesador $(12n + 24)b = \frac{3n^2}{p} + \frac{6n}{p}$ flops, y por barrido $\frac{3n^3}{p} + \frac{6n^2}{p}$ flops.

El cálculo de $off(A)$ se hace calculando $2n-2$ cuadrados y sumas por **bloque**, lo que hace $\frac{n^2-n}{p}$ flops por barrido.

Por tanto, el coste aritmético por barrido será

$$T_a = \frac{3n^3}{p} + \frac{25n^2}{2p} \qquad flops. \tag{5.1.1}$$

Para estudiar el coste de las comunicaciones llamaremos β_1 y β_2 a los costes de inicio de la comunicación en enlaces con el host (procesador P_0) y en el array lineal respectivamente, y τ_1 y τ_2 a los costes de envío y recepción de un dato, también con el host y en el array lineal.

Cada procesador calcula $2b$ rotaciones de Givens y las difunde a los demás procesadores, con lo que el coste de mandar los parámetros de las rotaciones de Givens de un procesador a otro adyacente en el anillo será $\beta_2 + 4b\tau_2$ y con el host $\beta_1 + 4b\tau_1$.

Supondremos que las comunicaciones entre procesadores del array lineal se hacen a través de este y las comunicaciones con el host por los enlaces directos.

De este modo, si un procesador sólo puede enviar o recibir un mensaje a la vez por un enlace (comunicaciones simples), el coste de la difusión de los parámetros de las rotaciones en el array sería $2(\beta_2+4b\tau_2)(p-2)$, y si las comunicaciones son múltiples (un procesador puede recibir y enviar a la vez por un enlace) el coste se divide por dos. En lo que se refiere al host el coste será: $(\beta_1 + 4b\tau_1)2$ si no puede enviar y recibir a la vez por un enlace (enlace con P_1) y puede trabajar con todos los enlaces simultáneamente, $(\beta_1 + 4b\tau_1)$ si puede enviar y recibir a la vez por un enlace y puede trabajar con todos los enlaces simultáneamente, $(\beta_1 + 4b\tau_1)p$ si no puede enviar y recibir a la vez por un enlace y sólo puede utilizar un enlace cada vez, y $(\beta_1 + 4b\tau_1)(p-1)$ si puede enviar y recibir a la vez por un enlace y sólo puede utilizar un enlace cada vez.

No consideraremos en adelante que se pueda o no enviar y recibir a la vez por un enlace pues esto sólo afecta al coste en un factor constante.

En la transferencia de las columnas el host transfiere datos a los $p-1$ procesadores restantes y transfiere $\frac{n}{p}$ datos a cada uno de ellos, lo que lleva un tiempo $\beta_1 + \frac{n}{p}\tau_1$ si puede transferir a todos los procesadores a la vez, ó $(\beta_1 + \frac{n}{p}\tau_1)(p-1)$ si no puede transferir por todos los enlaces a la vez.

Los procesadores del array transfieren n datos a un procesador adyacente, una parte al procesador siguiente y otra al anterior. En el caso más desfavorable de que no se pueda transferir simultáneamente por los dos enlaces el coste será $2\beta_2 + n\tau_2$.

Una vez transferidas las columnas se transfieren las filas, con lo que el coste de la transferencia de datos es dos veces el coste de la transferencia de columnas.

El coste de la transferencia de filas y columnas es $max\{2(\beta_1 + \frac{n}{p}\tau_1), 2(2\beta_2 + n\tau_2)\}$ o $max\{2(\beta_1 + \frac{n}{p}\tau_1)(p-1), 2(2\beta_2 + n\tau_2)\}$, dependiendo de que el host pueda comunicar con todos los procesadores al mismo tiempo o no.

De este modo, el coste por iteración será: $max\{2\beta_1 + \frac{2n}{p}\tau_1, 2\left(\beta_2 + \frac{n}{p}\tau_2\right)(p-2)\} + max\{2\beta_1 + \frac{2n}{p}\tau_1, 4\beta_2 + 2n\tau_2\}$ si el host puede comunicar con todos los procesadores a la vez; y $max\{\left(\beta_1+\frac{n}{p}\tau_1\right)p, 2\left(\beta_2+\frac{n}{p}\tau_2\right)(p-2)\}+max\{\left(2\beta_1+\frac{n}{p}\tau_1\right)(p-1), 4\beta_2+2n\tau_2\}$ si el host no puede comunicar con todos los procesadores a la vez. A este coste hay que sumarle el coste de las comunicaciones en el cálculo de $off(A)$. Pero este coste no aumenta el orden, por lo que el coste por barrido es de orden $O(n^2)$, menor que el orden del coste aritmético (expresión 5.1.1).

El inconveniente de las comunicaciones con este esquema es que no en todos los nodos se hacen del mismo modo y no están equilibradas las transferencias que se hacen por los distintos enlaces.

Si comparamos los resultados con los obtenidos con el esquema que no explota la simetría en un anillo (expresiones 4.2.3 y 4.2.4), vemos que se divide por dos el coste aritmético sin empeorar el orden del coste de las comunicaciones. De cualquier modo, con el esquema por **codos** se pueden obtener algoritmos teóricamente óptimos tal como hemos visto con la ordenación par-impar.

5.1.2 Por marcos

Para obtener este esquema se divide la matriz de tamaño $n \times n$ en $\frac{n}{4}$ **marcos** (figura 5.1.4) que llamaremos M_i con $i = 0, \ldots, \frac{n}{4}-1$, conteniendo cada **marco** elementos de cuatro filas y cuatro columnas. El **marco** M_i contendrá los elementos de las filas y columnas $2i$, $2i+1$, $n-1-2i$ y $n-2-2i$ que no pertenezcan a ninguna fila o columna j con $j = 0, \ldots, 2i-1$ ó $j = n-2i, \ldots, n-1$. Si se distribuye la matriz en los procesadores por **marcos**, elementos simétricos estarán en un mismo procesador, y por tanto no es necesario almacenarlos. Para que la distribución de los datos sea balanceada agruparemos los **marcos** por parejas que formarán un **bloque**, obteniendose de este modo $\frac{n}{8}$ **bloques** que llamaremos B_i, siendo $B_i = M_i \bigcup M_{\frac{n}{4}-1-i}$, con $i = 0, \ldots, \frac{n}{8}-1$ (figura 5.1.4). De este modo se agrupan el mayor y el menor **marco** en el mismo **bloque**, etc ..., y si consideramos $n = 8pb$ con p el número de procesadores y b un número natural, se pueden asignar a cada procesador b bloques consecutivos, conteniendo P_i los **bloques** B_j con $j = ib, \ldots, (i+1)b-1$, y estando el **bloque** B_i almacenado en $P_{i\ div\ p}$ (figura 5.1.4).

Topología asociada

Por idénticas razones que en la sección anterior, es necesario hacer transferencias de filas y columnas, y para determinar la topología a utilizar es suficiente con estudiar estas transferencias.

Seguimos considerando la ordenación par-impar. De este modo, de B_0 se transfieren datos a B_1, de B_1 a B_0 y B_2, de B_2 a B_1 y B_3, y así con los demás **bloques**, hasta el $B_{\frac{n}{8}-2}$ que transfiere al $B_{\frac{n}{8}-3}$ y al $B_{\frac{n}{8}-1}$, y el $B_{\frac{n}{8}-1}$ que transfiere al $B_{\frac{n}{8}-2}$, por lo que será suficiente con un array lineal con comunicaciones en los dos sentidos (figura 5.1.5).

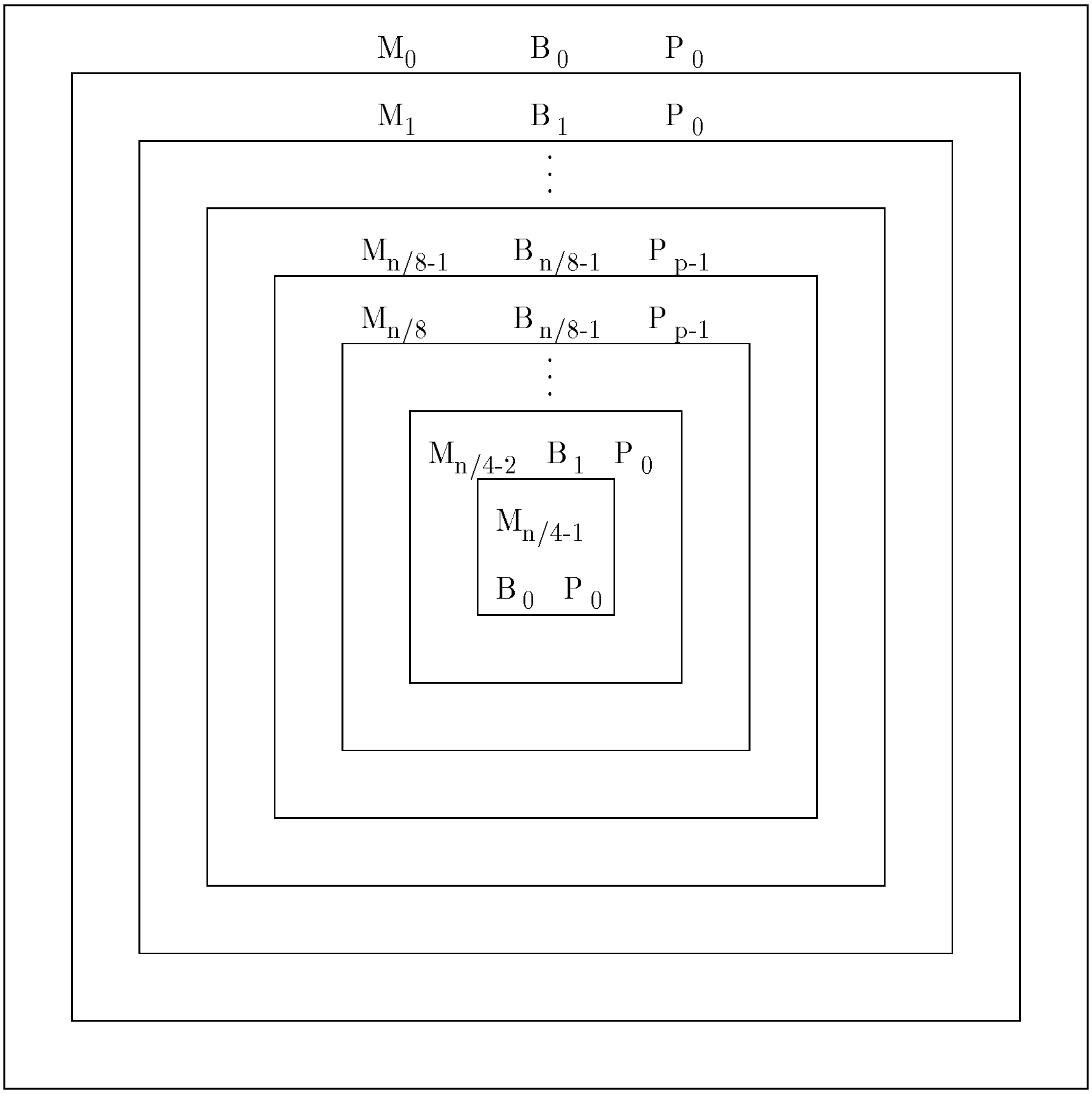


Figura 5.1.4: Esquema de almacenamiento por **marcos**.

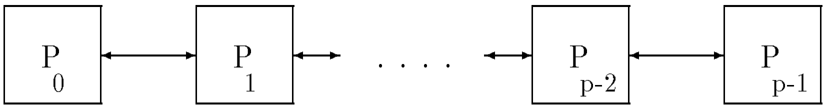


Figura 5.1.5: Topología asociada al esquema de almacenamiento por **marcos** usando la ordenación **par-impar**.

Costes teóricos

También en este caso el método de Jacobi que consideramos tiene el mismo esquema que el método sin explotación de la simetría del capítulo anterior, por lo que usa la ordenación par-impar, aunque el esquema de almacenamiento es por **marcos**.

Estudiamos el coste aritmético y de las comunicaciones en el cálculo de los valores propios.

Para estudiar el coste aritmético, si tenemos en cuenta que en cada procesador hay b **bloques** y que por cada **bloque** se calculan cuatro rotaciones de Givens, entonces cada procesador calcula $4b$ rotaciones de Givens, con un coste de $44b$ flops. Como el número de pasos por barrido es n (con la ordenación par-impar), el coste por barrido del cálculo de las rotaciones de Givens será $44bn = \frac{11n^2}{2p}$.

En la actualización de la matriz, como cada **bloque** tiene $2n$ submatrices 2×2 de las que $2n-4$ son simétricas dos a dos, tendremos que actualizar $n+2$ matrices 2×2, necesitando de 24 flops para la actualización de cada una de ellas. Resulta entonces que por **bloque** se necesitan $24n + 48$ flops, por procesador $(24n + 48)b = \frac{3n^2}{p} + \frac{6n}{p}$ flops, y por barrido $\frac{3n^3}{p} + \frac{6n^2}{p}$ flops.

El cálculo de $off(A)$ se hace calculando $4n-4$ cuadrados y sumas por **bloque**, lo que hace $\frac{n^2-n}{p}$ flops por barrido.

Por tanto, el coste aritmético por barrido será el mismo de la expresión 5.1.1.

El coste de las comunicaciones es en este caso más sencillo de estudiar que con el esquema por **codos**, pues ahora todos los nodos presentan las mismas características.

Cada procesador calcula $4b$ rotaciones de Givens y las difunde a los demás procesadores, y el coste de enviar los parámetros de las rotaciones a un procesador vecino será $\beta + 8b\tau$. Dependiendo de las características de la topología, si se pueden enviar y recibir datos simultáneamente por los enlaces el coste de la difusión es $(\beta + 8b\tau)(p-1)$, y si sólo se puede enviar o recibir por un enlace en un instante el coste es $2(\beta + 8b\tau)(p-1)$, pero esta característica no influirá en el orden de las comunicaciones.

En la transferencia de columnas el procesador P_i transfiere $n - 4bi$ datos de la columna mayor al procesador P_{i-1} y $4b + 4bi$ datos de la columna mayor de los **marcos** menores al procesador P_{i+1}, siendo $1 \leq i \leq p-2$.

El procesador P_0 transfiere al P_1 $4b = \frac{n}{2p}$ datos y el P_{p-1} al P_{p-2} $n - 4b(p-1) = \frac{n(p+1)}{2p}$ datos.

Si la transferencia se hace primero por un enlace y después por el otro el coste es $2\beta + \frac{n(p+1)\tau}{2p}$ si los enlaces permiten comunicaciones en los dos sentidos al mismo tiempo, y si sólo se permiten comunicaciones en un sentido cada vez el coste es $4\beta + \frac{n(p+1)\tau}{p}$.

El coste por barrido será en el peor caso

$$T_c = n\left(\left(2\beta + \frac{n\tau}{p}\right)(p-1) + 4\beta + \frac{n\,(p+1)\,\tau}{p}\right) \tag{5.1.2}$$

que es de orden $O(n^2)$, menor que el orden del coste aritmético.

Seguimos teniendo el problema de que en los nodos no está balanceado el volumen de las transferencias que se hace por cada uno de los enlaces pero hemos evitado el tener nodos de distinto tipo, y además la topología es ahora más simple.

Se trataría por tanto de un algoritmo teóricamente óptimo.

5.1.3 Por antidiagonales

Se divide la matriz en submatrices 2×2

$$A = \begin{bmatrix} A_{00} & A_{01} & \dots & A_{0k} \\ A_{10} & A_{11} & \dots & A_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k0} & A_{k1} & \dots & A_{kk} \end{bmatrix} \tag{5.1.3}$$

con $k = \frac{n}{2} - 1$, y se agrupan estas submatrices por **antidiagonales** (figura 5.1.6) que llamaremos A_i con $i = 0, \dots, \frac{n}{2} - 1$. Cada **antidiagonal** A_i está formada por las submatrices A_{jk} con $i = (j + k)\ mod\ \frac{n}{2}$. Si se distribuye la matriz en los procesadores por **antidiagonales** elementos simétricos estarán en un mismo procesador, y por tanto no es necesario almacenarlos. El cálculo de las rotaciones de Givens se realiza utilizando datos en las submatrices A_{ii}, por lo que, para balancear este cálculo, agruparemos las **antidiagonales** por parejas adyacentes que formarán un **bloque**, obteniéndose de este modo $\frac{n}{4}$ **bloques** que llamaremos B_i, siendo $B_i = A_{2i} \bigcup A_{2i+1}$, con $i = 0, \dots, \frac{n}{4}$ (figura 5.1.6). Si consideramos $n = 4pb$ con p el número de procesadores y b un número natural, se pueden asignar a cada procesador b bloques consecutivos, conteniendo P_i los **bloques** B_j, con $j = ib, \dots, (i + 1)b - 1$, y estando el **bloque** B_i almacenado en $P_{i\ div\ p}$ (figura 5.1.6).

Topología asociada

Tanto si n es múltiplo de cuatro como si no lo es las transferencias de filas y columnas se hacen de un **bloque** al anterior (con el módulo correspondiente y si seguimos utilizando la ordenación par-impar) con lo que es suficiente una topología de anillo unidireccional.

Costes teóricos

También en este caso el método de Jacobi que consideramos tiene el mismo esquema que el método sin explotación de la simetría del capítulo anterior, por lo que usa la ordenación par-impar, aunque el esquema de almacenamiento es por **antidiagonales**.

Estudiaremos el coste en el caso de calcular los valores propios.

Consideraremos n múltiplo de 4, con lo que tendremos $\frac{n}{4}$ **bloques** y en cada procesador b **bloques** siendo $n = 4pb$.

A_0 B_0 P_0	A_1 B_0 P_0	A_2 B_1 P_0	A_3 B_1 P_0			$A_{n/2\text{-}2}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	$A_{n/2\text{-}1}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$
A_1 B_0 P_0	A_2 B_1 P_0	A_3 B_1 P_0			$A_{n/2\text{-}2}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	$A_{n/2\text{-}1}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	A_0 B_0 P_0
A_2 B_1 P_0	A_3 B_1 P_0			$A_{n/2\text{-}2}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	$A_{n/2\text{-}1}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	A_0 B_0 P_0	A_1 B_0 P_0
A_3 B_1 P_0			$A_{n/2\text{-}2}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	$A_{n/2\text{-}1}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	A_0 B_0 P_0	A_1 B_0 P_0	A_2 B_1 P_0
		$A_{n/2\text{-}2}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	$A_{n/2\text{-}1}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	A_0 B_0 P_0	A_1 B_0 P_0	A_2 B_1 P_0	A_3 B_1 P_0
	$A_{n/2\text{-}2}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	$A_{n/2\text{-}1}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	A_0 B_0 P_0	A_1 B_0 P_0	A_2 B_1 P_0	A_3 B_1 P_0	
$A_{n/2\text{-}2}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	$A_{n/2\text{-}1}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	A_0 B_0 P_0	A_1 B_0 P_0	A_2 B_1 P_0	A_3 B_1 P_0		
$A_{n/2\text{-}1}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$	A_0 B_0 P_0	A_1 B_0 P_0	A_2 B_1 P_0	A_3 B_1 P_0			$A_{n/2\text{-}2}$ $B_{n/4\text{-}1}$ $P_{p\text{-}1}$

Figura 5.1.6: Esquema de almacenamiento por **antidiagonales**.

En cada procesador se calculan $2b$ rotaciones de Givens, lo que se hace en $22b = \frac{11n}{2p}$ flops, requiriéndose $\frac{11n^2}{2p}$ flops por barrido en el cálculo de las rotaciones.

En la actualización de la matriz, como cada **bloque** tiene n submatrices 2×2 de las que $n-2$ son simétricas, tendremos que actualizar $\frac{n}{2}+1$ submatrices 2×2, lo que se hace en $12n + 24$ flops.

Por tanto, el coste aritmético por barrido será el mismo de la expresión 5.1.1.

Cada procesador calcula $2b$ rotaciones de Givens y las difunde a los demás procesadores, y el coste de enviar los parámetros de las rotaciones a un procesador vecino será $\beta + 4b\tau$. Al ser el anillo bidireccional el coste de la difusión será $2(\beta + 4b\tau)(p-1)$.

En la transferencia de columnas un procesador transfiere al anterior columnas de las submatrices más próximas al procesador destino, en el caso más favorable en todas las submatrices la misma columna (la primera o la segunda) con lo que el tiempo será $\beta + n\tau$, y en otros casos de las primeras submatrices la columna segunda y de las últimas la primera. Si se copian las columnas a enviar en una columna auxiliar antes de la transmisión y se transmite esta columna el tiempo será $\beta + n\tau$, pero si se transfieren en dos partes el tiempo será $2\beta + n\tau$ (lo mismo ocurre con la transferencia de filas).

Por tanto, el coste de las comunicaciones por barrido será

$$T_c = 2n\,(p+2)\,\beta + 2\left(2n^2 - \frac{n^2}{p}\right)\tau \quad . \tag{5.1.4}$$

También con este esquema se obtiene un algoritmo teóricamente óptimo.

El coste de las comunicaciones se puede mejorar si tenemos en cuenta que sólo es necesario trabajar con la parte triangular superior (o triangular inferior) de la matriz, con lo que se puede evitar el envío de datos en la parte que no se esté usando. Estudiaremos estas comunicaciones con más detalle en las siguientes secciones.

5.1.4 Comparación

Con los tres esquemas llegamos a alcanzar un speed-up de p y una eficiencia del 100%, y en los tres se hace a costa de empeorar las comunicaciones con respecto al algoritmo que no explota la simetría en un factor constante, al hacerse transferencia de filas y columnas.

Los esquemas por **marcos** y por **antidiagonales** necesitan de una topología más simple (si usamos la ordenación par-impar), pero con los esquemas de **codos** y **marcos** se puede obtener una ligera mejora en las comunicaciones si se pueden enviar y recibir datos a la vez por un mismo enlace y por los distintos enlaces de un procesador. Además, estos esquemas producen un almacenamiento más compacto de los datos, por lo que es más fácil usar BLAS eficientemente. Más adelante veremos cómo se puede usar BLAS eficientemente con el esquema por marcos.

En los tres esquemas la memoria necesaria en cada procesador se reduce aproximadamente a la mitad al utilizar el hecho de que la matriz es simétrica.

El esquema menos restrictivo en cuanto a la dimensión de la matriz es el de **antidiagonales** en el que es suficiente con que n sea un número par, después el de **codos** en el que n debe ser múltiplo de 4 y por último el de **marcos** en el que debe ser múltiplo de 8. En cualquier caso, esta no es una restricción importante pues basta con construir una matriz añadiendo filas y columnas de ceros con unos en la diagonal hasta obtener la dimensión necesaria.

5.2 Algoritmo con almacenamiento por antidiagonales

5.2.1 Descripción

El algoritmo simétrico que presentamos está basado en las siguientes consideraciones. La matriz A de tamaño $n \times n$, siendo n un número par, se divide en bloques 2×2

$$A = \begin{bmatrix} A_{00} & A_{01} & \dots & A_{0k} \\ A_{10} & A_{11} & \dots & A_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k0} & A_{k1} & \dots & A_{kk} \end{bmatrix} \tag{5.2.1}$$

donde $k = \frac{n}{2} - 1$. Supongamos que disponemos de $p = \frac{n}{2}$ procesadores de modo que asignamos, de manera cíclica, a cada procesador todos los bloques situados en una misma **antidiagonal**. Parece evidente que, por la simetría de la matriz, en cada procesador sólo es necesario almacenar aquellos bloques pertenecientes a una misma **antidiagonal** que están situados en o por encima de la diagonal principal de bloques, es decir, del tipo A_{ij} con $i \leq j$, pero, como veremos más adelante, dependiendo de la ordenación que se use será necesario almacenar algunos bloques adicionales. La ordenación que usaremos es la par-impar, y será necesario almacenar los bloques $A_{i,i-1}$ con $i = 1, 2, \dots, k$. Si en cada paso únicamente anulamos elementos no diagonales situados en bloques A_{ii}, en los procesadores impares hay que calcular dos rotaciones de Givens y en los pares ninguna, como se observa en el ejemplo de la figura 5.2.1, donde los ceros indican los elementos a anular y el procesador en que se encuentra cada bloque se representa por P_i.

Para tratar de mejorar el balanceo de la carga, supongamos que se dispone de $p = \frac{n}{4}$ procesadores y que asignamos cíclicamente a cada procesador todos los bloques situados en dos **antidiagonales** consecutivas. En el ejemplo de la figura 5.2.2 puede comprobarse que con esta distribución cada procesador calcula dos rotaciones de Givens.

En el caso general, si $n = 2bp$, tendremos que asignar a cada procesador los bloques situados en b **antidiagonales** consecutivas. Si b es par se conseguirá balancear el

$$
\left[\begin{array}{ccc|ccc|ccc|ccc}
\star & & 0 & \star & & \star & \star & & \star & \star & & \star \\
& P_0 & & & P_1 & & & P_2 & & & P_3 & \\
0 & & \star & \star & & \star & \star & & \star & \star & & \star \\
\hline
\star & & \star & \star & & 0 & \star & & \star & \star & & \star \\
& P_1 & & & P_2 & & & P_3 & & & P_0 & \\
\star & & \star & 0 & & \star & \star & & \star & \star & & \star \\
\hline
\star & & \star & \star & & \star & \star & & 0 & \star & & \star \\
& P_2 & & & P_3 & & & P_0 & & & P_1 & \\
\star & & \star & \star & & \star & 0 & & \star & \star & & \star \\
\hline
\star & & \star & \star & & \star & \star & & \star & \star & & 0 \\
& P_3 & & & P_0 & & & P_1 & & & P_2 & \\
\star & & \star & \star & & \star & \star & & \star & 0 & & \star
\end{array}\right]
$$

Figura 5.2.1: n=8, k=3, p=4, b=1.

$$
\left[\begin{array}{ccc|ccc|ccc|ccc}
\star & & 0 & \star & & \star & \star & & \star & \star & & \star \\
& P_0 & & & P_0 & & & P_1 & & & P_1 & \\
0 & & \star & \star & & \star & \star & & \star & \star & & \star \\
\hline
\star & & \star & \star & & 0 & \star & & \star & \star & & \star \\
& P_0 & & & P_1 & & & P_1 & & & P_0 & \\
\star & & \star & 0 & & \star & \star & & \star & \star & & \star \\
\hline
\star & & \star & \star & & \star & \star & & 0 & \star & & \star \\
& P_1 & & & P_1 & & & P_0 & & & P_0 & \\
\star & & \star & \star & & \star & 0 & & \star & \star & & \star \\
\hline
\star & & \star & \star & & \star & \star & & \star & \star & & 0 \\
& P_1 & & & P_0 & & & P_0 & & & P_1 & \\
\star & & \star & \star & & \star & \star & & \star & 0 & & \star
\end{array}\right]
$$

Figura 5.2.2: n=8, k=3, p=2, b=2.

cálculo de las rotaciones de Givens, calculándose en cada procesador b rotaciones.

De este modo, el procesador P_r contendrá los bloques 2×2 A_{ij} y A_{kl}, donde los índices i, j, k, l, se calculan por medio del siguiente algoritmo.

Algoritmo 5.2.1 *Cálculo de las submatrices 2×2 que se almacenan en el procesador P_r utilizando el esquema de almacenamiento por* ***antidiagonales****.*

```
PARA m = 0, 1, ..., b − 1
   t = br + m
   PARA i = 0, ..., t div 2
      j = t − i
   FINPARA
   t = br + m + 1
   PARA i = t, ..., (t − 1) div 2 + n/4
      j = t − i − 1 + n/2
   FINPARA
FINPARA
```

Sólo se almacenan los bloques A_{ij}, $i \leq j$. Y los $A_{i,i-1}$ se conocen por sus simétricos.

Llamaremos q-ésima **antidiagonal** del procesador P_r a los bloques de la matriz A generados en el algoritmo anterior para $m = q$.

Por otro lado, un bloque A_{ij} con $i \leq j + 1$ estará contenido en el procesador P_r siendo

$$\begin{array}{ll} r = (i+j)\ div\ b, & si\ i+j < \frac{n}{2} \\ r = (i+j-\frac{n}{2})\ div\ b, & si\ i+j \geq \frac{n}{2}. \end{array} \tag{5.2.2}$$

En el apartado siguiente suponemos que la distribución de la matriz en el sistema multiprocesador sigue este esquema.

5.2.2 Algoritmo

Transferencia de filas y columnas

Después del cálculo de las rotaciones de Givens que anulan elementos no diagonales situados en bloques de la forma A_{ii} es necesaria su difusión para la actualización de la matriz. En este caso sólo necesitamos actualizar la mitad de sus elementos pues los elementos simétricos se encuentran en un mismo procesador.

Con esta distribución de los datos, y para seguir teniendo elementos simétricos en un mismo procesador, necesitamos transferir columnas y filas, como se muestra en el siguiente ejemplo para $n = 8$, $b = p = 2$.

Consideramos el conjunto inicial de pares $\{(0,1),(2,3),(4,5),(6,7)\}$, por lo que el estado inicial de la matriz será el de la figura 5.2.3.

$$
\left[\begin{array}{ccc|ccc|ccc|ccc}
a_{00} & & \boxed{a_{01}} & a_{02} & & a_{03} & a_{04} & & a_{05} & a_{06} & & a_{07} \\
& P_0 & & & P_0 & & & P_1 & & & P_1 & \\
\boxed{a_{10}} & & a_{11} & a_{12} & & a_{13} & a_{14} & & a_{15} & a_{16} & & a_{17} \\
\hline
a_{20} & & a_{21} & a_{22} & & \boxed{a_{23}} & a_{24} & & a_{25} & a_{26} & & a_{27} \\
& P_0 & & & P_1 & & & P_1 & & & P_0 & \\
a_{30} & & a_{31} & \boxed{a_{32}} & & a_{33} & a_{34} & & a_{35} & a_{36} & & a_{37} \\
\hline
a_{40} & & a_{41} & a_{42} & & a_{43} & a_{44} & & \boxed{a_{45}} & a_{46} & & a_{47} \\
& P_1 & & & P_1 & & & P_0 & & & P_0 & \\
a_{50} & & a_{51} & a_{52} & & a_{53} & \boxed{a_{54}} & & a_{55} & a_{56} & & a_{57} \\
\hline
a_{60} & & a_{61} & a_{62} & & a_{63} & a_{64} & & a_{65} & a_{66} & & \boxed{a_{67}} \\
& P_1 & & & P_0 & & & P_0 & & & P_1 & \\
a_{70} & & a_{71} & a_{72} & & a_{73} & a_{74} & & a_{75} & \boxed{a_{76}} & & a_{77}
\end{array}\right]
$$

Figura 5.2.3: n=8, b=p=2.

Tras anular, mediante rotaciones planas, los elementos señalados, pasaríamos a realizar la transferencia de columnas, utilizando el algoritmo de modificación de pares de la ordenación par-impar. Esto es equivalente a postmultiplicar la matriz anterior por la matriz de permutación $P = [e_1, e_4, e_3, e_6, e_5, e_8, e_7, e_2]$, siendo e_i el i-ésimo vector columna unitario de R^8. Con esto se obtiene la distribución que aparece en la figura 5.2.4.

Se puede observar que algunos elementos simétricos están situados en procesadores distintos (por ejemplo, a_{05} está en P_0 y a_{50} en P_1), por lo que es necesario realizar una transferencia de filas. Para ello basta con premultiplicar por la misma matriz de permutación P, obteniéndose como resultado la distribución que aparece en la figura 5.2.5.

A continuación se podría llevar a cabo el siguiente paso, con el conjunto de pares de índices $\{(0,3),(2,5),(4,7),[6,1]\}$.

Se sobreentiende que los elementos de cada bloque A_{ij} van variando conforme avanza el algoritmo. Por ejemplo, en el caso particular que estamos considerando, con $n = 8$ y $b = p = 2$, el bloque A_{00} vendría dado inicialmente por

$$
\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \quad .
$$

Tras la anulación de a_{01}, a_{10} y la transferencia de columnas se convierte en

$$
\begin{bmatrix} a_{00} & a_{03} \\ 0 & a_{33} \end{bmatrix}
$$

y por último tras la transferencia de filas en

$$
\left[\begin{array}{ccc|ccc|ccc|ccc}
a_{00} & & a_{03} & a_{02} & & a_{05} & a_{04} & & a_{07} & a_{06} & & 0 \\
& P_0 & & & P_0 & & & P_1 & & & P_1 & \\
0 & & a_{13} & a_{12} & & a_{15} & a_{14} & & a_{17} & a_{16} & & a_{11} \\
\hline
a_{20} & & 0 & a_{22} & & a_{25} & a_{24} & & a_{27} & a_{26} & & a_{21} \\
& P_0 & & & P_1 & & & P_1 & & & P_0 & \\
a_{30} & & a_{33} & 0 & & a_{35} & a_{34} & & a_{37} & a_{36} & & a_{31} \\
\hline
a_{40} & & a_{43} & a_{42} & & 0 & a_{44} & & a_{47} & a_{46} & & a_{41} \\
& P_1 & & & P_1 & & & P_0 & & & P_0 & \\
a_{50} & & a_{53} & a_{52} & & a_{55} & 0 & & a_{57} & a_{56} & & a_{51} \\
\hline
a_{60} & & a_{63} & a_{62} & & a_{65} & a_{64} & & 0 & a_{66} & & a_{61} \\
& P_1 & & & P_0 & & & P_0 & & & P_1 & \\
a_{70} & & a_{73} & a_{72} & & a_{75} & a_{74} & & a_{77} & 0 & & a_{71}
\end{array}\right]
$$

Figura 5.2.4: n=8, b=p=2.

$$
\left[\begin{array}{ccc|ccc|ccc|ccc}
a_{00} & & a_{03} & a_{02} & & a_{05} & a_{04} & & a_{07} & a_{06} & & 0 \\
& P_0 & & & P_0 & & & P_1 & & & P_1 & \\
a_{30} & & a_{33} & 0 & & a_{35} & a_{34} & & a_{37} & a_{36} & & a_{31} \\
\hline
a_{20} & & 0 & a_{22} & & a_{25} & a_{24} & & a_{27} & a_{26} & & a_{21} \\
& P_0 & & & P_1 & & & P_1 & & & P_0 & \\
a_{50} & & a_{53} & a_{52} & & a_{55} & 0 & & a_{57} & a_{56} & & a_{51} \\
\hline
a_{40} & & a_{43} & a_{42} & & 0 & a_{44} & & a_{47} & a_{46} & & a_{41} \\
& P_1 & & & P_1 & & & P_0 & & & P_0 & \\
a_{70} & & a_{73} & a_{72} & & a_{75} & a_{74} & & a_{77} & 0 & & a_{71} \\
\hline
a_{60} & & a_{63} & a_{62} & & a_{65} & a_{64} & & 0 & a_{66} & & a_{61} \\
& P_1 & & & P_0 & & & P_0 & & & P_1 & \\
0 & & a_{13} & a_{12} & & a_{15} & a_{14} & & a_{17} & a_{16} & & a_{11}
\end{array}\right]
$$

Figura 5.2.5: n=8, b=p=2.

$$\begin{bmatrix} a_{00} & a_{03} \\ a_{30} & a_{33} \end{bmatrix} \quad .$$

Las transferencias de datos realizadas se efectúan mediante envíos de una **antidiagonal** de bloques a la **antidiagonal** inmediata superior. En la "transferencia de columnas" se envían datos de cada bloque A_{ij} al bloque $A_{i,(j-1)\ mod\ \frac{n}{2}}$, y en la "transferencia de filas" del bloque A_{ij} al bloque $A_{(i-1)\ mod\ \frac{n}{2},j}$. De este modo, tanto en la transferencia de filas como en la de columnas, sólo los datos de bloques situados en la primera **antidiagonal** de cada procesador se envían a un procesador distinto (del procesador P_i al $P_{(i-1)\ mod\ p}$), como se puede observar en el dibujo de la figura 5.2.6.

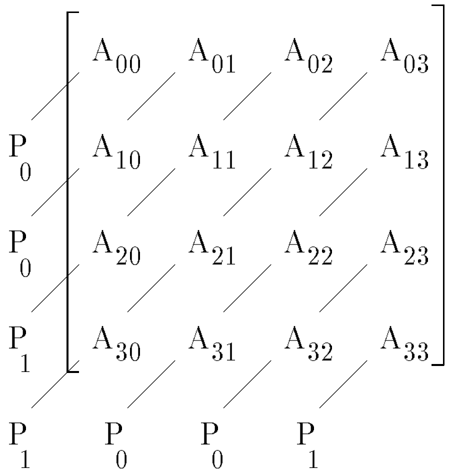


Figura 5.2.6: n=8, b=p=2.

Si se hace primero la transferencia de columnas y después la transferencia de filas, tendremos un coste de comunicación de $4\beta + 4n\tau$. Este coste se puede mejorar si tenemos en cuenta que, por la simetría, sólo necesitamos conocer el triángulo superior de la matriz. Para ilustrar estas ideas acudamos de nuevo al ejemplo de las figuras 5.2.3, 5.2.4, 5.2.5 y 5.2.6, analizando cómo podemos reorganizar las comunicaciones para obtener la distribución indicada en la figura 5.2.5, pero limitándonos a los bloques A_{ij} con $i \leq j$. Los datos correspondientes de estos bloques son sustituidos, en la transferencia de filas, por datos provenientes de bloques de la forma $A_{(i+1)\ mod\ \frac{n}{2},j}$. En efecto, representemos, en la figura 5.2.7, cada bloque A_{ij} de la matriz de la figura 5.2.4 mediante X_i ó F_i, donde el subíndice indica el procesador donde se encuentra almacenado dicho bloque.

Los datos que provienen de la segunda fila de los bloques señalados con F son los únicos que sustituyen a datos de los bloques A_{ij} con $i \leq j$ (cada bloque F_i modifica

$$\begin{bmatrix} X_0 & X_0 & X_1 & F_1 \\ F_0 & F_1 & F_1 & F_0 \\ X_1 & F_1 & F_0 & F_0 \\ X_1 & X_0 & F_0 & F_1 \end{bmatrix}$$

Figura 5.2.7: n=8, b=p=2.

el bloque inmediato superior situado en su misma columna). Como en la transferencia entre procesadores sólo intervienen datos de la primera **antidiagonal** de cada procesador, se puede comprobar que cada procesador sólo necesita transferir $\frac{n}{2}$ datos.

Del análisis realizado sobre la transferencia de filas, se deduce que en la transferencia de columnas bastará con enviar datos de bloques que modifiquen bloques del tipo A_{ij} con $i-1 \leq j$. Por tanto, se enviarán datos de los bloques A_{ij} con $i \leq j$ y de A_{i0} con $i = 0, 1, \ldots, \frac{n}{2} - 1$. Para fijar ideas, en la figura 5.2.8 representamos cada bloque A_{ij} de la matriz de la figura 5.2.3 (suponiendo que ya hemos anulado los elementos señalados) mediante X_i ó C_i, donde el subíndice indica el procesador donde se encuentra almacenado dicho bloque. Los datos que provienen de la segunda columna de los bloques señalados con C son los únicos que modifican a los bloques situados en el triángulo superior y en la subdiagonal principal de subbloques 2×2. Hay que hacer notar que los bloques A_{i0} con $i = 1, 2, \ldots, \frac{n}{2} - 1$ no es necesario almacenarlos pues se conocen por sus simétricos, pero tras la transferencia de columnas se pierde la simetría (figura 5.2.4), por lo que al hacer la transferencia de filas el contenido de los bloques $A_{i,i-1}$ no se puede deducir del de sus simétricos, lo que hace necesario el trabajo directo con los bloques $A_{i,i-1}$ y por tanto tenerlos almacenados en la memoria.

$$\begin{bmatrix} C_0 & C_0 & C_1 & C_1 \\ C_0 & C_1 & C_1 & C_0 \\ C_1 & X_1 & C_0 & C_0 \\ C_1 & X_0 & X_0 & C_1 \end{bmatrix}$$

Figura 5.2.8: n=8, b=p=2.

Puede comprobarse que, con esta transferencia de columnas, cada procesador sólo tiene que enviar $\frac{n}{2} + 4$ datos, en el peor de los casos.

Con esta estrategia de transferencia de columnas y filas el coste de las comunicaciones en la transferencia de datos será

$$4\beta + (2n + 8)\tau \quad . \tag{5.2.3}$$

Código del algoritmo

Para el cálculo de los valores propios, con los datos distribuidos por **antidiagonales**, el esquema es similar al del algoritmo que no explota la simetría pero con transferencia de columnas y de filas (de parte de las columnas y de las filas).

En todos los procesadores se ejecutará el mismo código

Algoritmo 5.2.2 *Esquema del algoritmo JaEsPaSiAnAnPi.*

EN PARALELO para $r = 0, 1, \ldots, p-1$ en P_r:
  REPETIR
    PARA $i = 1, \ldots, n$
      calcular rotaciones (i, r)
      difundir rotaciones (r)
      actualizar matriz (r)
      transferir columnas (i, r)
      transferir filas (i, r)
    FINPARA
    calcular $off(A)$
  HASTA $off(A) < COTA$

Los procedimientos utilizados en este algoritmo tienen los siguientes significados.

Algoritmo 5.2.3 *Cálculo de rotaciones (i,r).*

SI $i \ mod \ 2 = 1$
  PARA $j = 0, \ldots, b-1$
    SI $j \ mod \ 2 = 0$
      calcular rotación correspondiente a los índices del par
        $((br + j) \ div \ 2)$-ésimo
        (*utilizando los datos en el bloque $A_{(br+j) \ div \ 2, (br+j) \ div \ 2}$*)
      calcular rotación correspondiente a los índices del par
        $((br + j) \ div \ 2 + \frac{n}{4})$-ésimo
        (*utilizando los datos en el bloque $A_{(br+j) \ div \ 2+\frac{n}{4}, (br+j) \ div \ 2+\frac{n}{4}}$*)
    FINSI
  FINPARA
EN OTRO CASO
  PARA $j = 0, \ldots, b-1$
    SI $j \ mod \ 2 = 0$
      calcular rotación correspondiente a los índices del par
        $((br + j) \ div \ 2)$-ésimo
        (*utilizando los datos en el bloque $A_{(br+j) \ div \ 2, (br+j) \ div \ 2}$*)
      calcular rotación correspondiente a los índices del par
        $((br + j) \ div \ 2 + \frac{n}{4})$-ésimo

(*utilizando los datos en el bloque $A_{(br+j)\ div\ 2+\frac{n}{4},(br+j)\ div\ 2+\frac{n}{4}}$*)
(*Hacer los cálculos tomando $c = 1$ y $s = 0$ para el par $((n-i)\ div\ 2)$-ésimo.*)
FINSI
FINPARA
FINSI

La difusión de los parámetros es igual a la del algoritmo que no explota la simetría, ya que se trata de un broadcast usando la topología de anillo (algoritmo 4.2.4).

Algoritmo 5.2.4 *Actualizar matriz (r).*
PARA $j = 0, \ldots, b-1$
$k = br + j$
PARA $q = 0, \ldots, k\ div\ 2$
$m = k - q$

$$\begin{bmatrix} a_{2q,2m} & a_{2q,2m+1} \\ a_{2q+1,2m} & a_{2q+1,2m+1} \end{bmatrix} = \begin{bmatrix} c_q & s_q \\ -s_q & c_q \end{bmatrix} \begin{bmatrix} a_{2q,2m} & a_{2q,2m+1} \\ a_{2q+1,2m} & a_{2q+1,2m+1} \end{bmatrix} \begin{bmatrix} c_m & -s_m \\ s_m & c_m \end{bmatrix}$$

(*$A_{qm} = Q_q A_{qm} Q_m^t$*)
FINPARA
$k = br + j + 1$
PARA $q = k, \ldots, (k-1)\ div\ 2 + \frac{n}{4}$
$m = k - q - 1 + \frac{n}{2}$

$$\begin{bmatrix} a_{2q,2m} & a_{2q,2m+1} \\ a_{2q+1,2m} & a_{2q+1,2m+1} \end{bmatrix} = \begin{bmatrix} c_q & s_q \\ -s_q & c_q \end{bmatrix} \begin{bmatrix} a_{2q,2m} & a_{2q,2m+1} \\ a_{2q+1,2m} & a_{2q+1,2m+1} \end{bmatrix} \begin{bmatrix} c_m & -s_m \\ s_m & c_m \end{bmatrix}$$

(*$A_{qm} = Q_q A_{qm} Q_m^t$*)
FINPARA
FINPARA

Obviamente, en el algoritmo 5.2.4 la actualización se hace sobre valores obtenidos en cada A_{ij}, pero estos valores cambian en cada iteración al aplicar los procedimientos de transferencia de columnas y de filas.

Algoritmo 5.2.5 *Transferencia de columnas (i,r).*
PARA todos los bloques A_{jk} en P_r:
SI $A_{j,(k-1)\ mod\ \frac{n}{2}}$ no está en P_r
enviar columna m del bloque A_{jk} a $P_{(r-1)\ mod\ p}$
SI $b = 1$
recibir columna del bloque $A_{j,(k+1)\ mod\ \frac{n}{2}}$ y almacenarla
en la columna m del bloque A_{jk}
EN OTRO CASO

$A_{jk}(:,m) = A_{j,(k+1)\ mod\ \frac{n}{2}}(:,m)$
FINSI
EN OTRO CASO
$A_{j,(k-1)\ mod\ \frac{n}{2}}(:,m) = A_{jk}(:,m)$
SI $A_{j,(k+1)\ mod\ \frac{n}{2}}$ no está en P_r
recibir columna del bloque $A_{j,(k+1)\ mod\ \frac{n}{2}}$ y almacenarla
en la columna m de A_{jk}
EN OTRO CASO
$A_{jk}(:,m) = A_{j,(k+1)\ mod\ \frac{n}{2}}(:,m)$
FINSI
FINSI
FINPARA

En este algoritmo el valor de m para A_{jk} es

$$\begin{array}{ll} m = 1, & si\ k = 0, 1, \ldots, (n-i-1)\ div\ 2 \\ m = 0, & si\ k = (n-i-1)\ div\ 2+1, \ldots, \frac{n}{2}-1. \end{array} \tag{5.2.4}$$

Hay que hacer notar que los bloques A_{jk} en P_r se deben referenciar en el bucle de modo que no se pierda información al hacer la asignación de columnas, es decir, por filas y dentro de cada fila en orden creciente de columnas. Los bloques de la forma A_{ij} con $i > j$ de los que se necesita transferir columnas son conocidos gracias a sus simétricos.

Algoritmo 5.2.6 *Transferencia de filas (i,r).*
PARA todos los bloques A_{jk} en P_r:
SI $A_{(j-1)\ mod\ \frac{n}{2},k}$ no está en P_r
enviar fila m del bloque A_{jk} a $P_{(r-1)\ mod\ p}$
SI $b = 1$
recibir fila del bloque $A_{(j+1)\ mod\ \frac{n}{2},k}$ y almacenarla
en la fila m del bloque A_{jk}
EN OTRO CASO
$A_{jk}(m,:) = A_{(j+1)\ mod\ \frac{n}{2},k}(m,:)$
FINSI
EN OTRO CASO
$A_{(j-1)\ mod\ \frac{n}{2},k}(m,:) = A_{jk}(m,:)$
SI $A_{(j+1)\ mod\ \frac{n}{2},k}$ no está en P_r
recibir fila del bloque $A_{(j+1)\ mod\ \frac{n}{2},k}$ y almacenarla
en la fila m del bloque A_{jk}
EN OTRO CASO
$(A_{jk})(m,:) = A_{(j+1)\ mod\ \frac{n}{2},k}(m,:)$
FINSI

FINSI
FINPARA

En este algoritmo el valor de m para A_{jk} es

$$\begin{array}{ll} m = 1, & si\ j = 0, 1, \ldots, (n-i-1)\ div\ 2 \\ m = 0, & si\ j = (n-i-1)\ div\ 2+1, \ldots, \frac{n}{2}-1. \end{array} \qquad (5.2.5)$$

Hay que hacer notar que los bloques A_{jk} en P_r se deben referenciar en el bucle de modo que no se pierda información al hacer la asignación de filas, es decir, por columnas, y dentro de cada columna en orden creciente de filas.

5.2.3 Costes teóricos

Suponemos $n = 2bp$, con lo que en cada procesador tendremos b **antidiagonales** consecutivas.

El coste aritmético del cálculo de las rotaciones de Givens y de $off(A)$ no varía respecto al del algoritmo que no explota la simetría, pero el de la actualización de la matriz sí varía. Al tener en este caso $\frac{n^2+2n}{8p}$ bloques 2×2 en cada procesador, el coste aritmético por barrido, despreciando los términos de menor orden, viene dado por

$$T_a = \frac{3n^3}{p} + \frac{25n^2}{2p} \quad flops. \qquad (5.2.6)$$

El coste de las comunicaciones en la difusión de los parámetros de las rotaciones y en el cálculo de $off(A)$ es igual al del algoritmo no simétrico, pero el de la transferencia de columnas es $2\beta + (n+8)\tau$, y el de la transferencia de filas $2\beta + n\tau$, con lo que el coste de las comunicaciones por barrido viene dado por

$$T_c = 2n(p+1)\beta + 4n^2\tau \quad , \qquad (5.2.7)$$

donde hemos despreciado los términos de orden inferior.

También se puede diseñar este algoritmo para hipercubo utilizando el anillo inmerso en el hipercubo, pero utilizando en las difusiones la topología de hipercubo, obteniéndose de este modo que el término que afecta a β sería de $2n(2 + log_2\ p)$.

De cualquier modo, en el algoritmo que explota la simetría la cota superior que se obtiene para la eficiencia es

$$E_p = \frac{T_1}{p(T_a + T_c)} \leq \frac{T_1}{pT_a} \leq 1. \qquad (5.2.8)$$

5.2.4 Resultados experimentales

En este apartado presentaremos los resultados del algoritmo que explota la simetría en anillo (JaEsPaSiAnAnPi). Analizaremos estos resultados y los compararemos con los que se obtienen con el algoritmo que no explota la simetría, estudiado en el capítulo anterior (JaEsPaNsAnCoPi).

Se mostrarán resultados obtenidos en el PARSYS SN-1040 basado en Transputers T800, el iPSC/2 y el iPSC/860.

Hemos utilizado una topología de anillo unidireccional, y hemos hecho las implementaciones asignando a cada procesador b bloques consecutivos, con lo que $pb = \frac{n}{4}$ y por tanto las implementaciones servirán para valores de n múltiplos de cuatro. Esto no es un inconveniente pues siempre se pueden añadir las filas y las columnas de ceros con unos en la diagonal que hagan falta.

En las implementaciones usamos el orden par-impar en un anillo unidireccional pues, aunque se necesitan n pasos en cada barrido y se podría hacer en $n-1$ pasos usando otras ordenaciones, hemos considerado que cuando n crece la diferencia de un paso no es significativa, y que el método par-impar tiene la ventaja de que, utilizado con el esquema de almacenamiento por **antidiagonales**, necesita de un anillo unidireccional y en la transferencia de datos cada procesador envía datos a uno de sus vecinos y recibe del otro, mientras que con otras ordenaciones se necesitaría de más comunicaciones (ordenación Round-Robin) o de un anillo bidireccional (ordenación de Eberlein).

En la tabla 5.2.1 aparecen los resultados obtenidos en un PARSYS SN-1040 utilizando topología de anillo unidireccional cuando se utilizan 2, 4, 8 y 16 Transputers y matrices densas generadas aleatoriamente con datos entre -10 y 10 y con tamaños 64, 128, 192, 256 y 320. En todos los casos ejecutamos los mismos programas paralelos que hemos llamado JaEsPaNsAnCoPi y JaEsPaSiAnAnPi, coincidiendo el número de procesos con el de nodos de la red (un proceso en cada nodo). La implementación se ha hecho en 3L C y los resultados que aparecen en la tabla son tiempos por barrido, en segundos. También figuran los tiempos obtenidos con un algoritmo secuencial que explota la simetría y el cociente entre los tiempos de JaEsPaSiAnAnPi y JaEsPaNsAnCoPi. Idealmente este cociente debería ser 0.5 puesto que con JaEsPaSiAnAnPi trabajamos aproximadamente con la mitad de la matriz, mientras que con JaEsPaNsAnCoPi trabajamos con toda la matriz (al no explotar la simetría), pero lo que se observa en la tabla es que cuando el tamaño de la matriz crece el cociente tiende a estar alrededor de 0.6. Esto es debido a dos razones:

1. El programa correspondiente a JaEsPaSiAnAnPi presenta mayor "overhead" que el JaEsPaNsAnCoPi debido a que la distribución de los datos en los procesadores es más irregular.

2. En JaEsPaSiAnAnPi se realizan transferencias de filas y columnas produciéndose por esto mayor contención en las sincronizaciones.

	64	128	192	256	320
JaEsPaNsAnCoPi 2	3.75	29.12	97.12	226.12	439.77
JaEsPaSiAnAnPi 2	2.61	18.02	57.71	133.55	256.71
Cociente 2	0.6969	0.6188	0.5941	0.5906	0.5837
JaEsPaNsAnCoPi 4	1.95	14.69	49.27	116.12	219.66
JaEsPaSiAnAnPi 4	1.46	9.64	30.06	69.05	130.52
Cociente 4	0.7465	0.6562	0.6089	0.5946	0.5941
JaEsPaNsAnCoPi 8	1.08	7.73	24.90	58.14	112.18
JaEsPaSiAnAnPi 8	0.90	5.41	14.41	37.19	69.17
Cociente 8	0.8344	0.6993	0.6558	0.6396	0.6166
JaEsPaNsAnCoPi16	0.66	4.23	13.60	31.10	59.71
JaEsPaSiAnAnPi 16	0.65	3.36	9.49	20.61	37.70
Cociente 16	0.9762	0.7928	0.6979	0.6628	0.6313
JaEsSeSiFi	3.46	27.11	91.49	217.91	424.38

Tabla 5.2.1: Tiempos de ejecución por barrido (en segundos) de los algoritmos JaEsPaNsAnCoPi y JaEsPaSiAnCoPi. En el PARSYS SN-1040.

Vemos (tabla 5.2.1) que cuando el número de nodos aumenta la mejora de JaEsPaSiAnAnPi respecto a JaEsPaNsAnCoPi disminuye y cuando el tamaño de la matriz disminuye se reduce la mejora obtenida con JaEsPaSiAnAnPi. Esto se debe a que aunque el coste aritmético es de mayor orden que el de las comunicaciones y en JaEsPaSiAnAnPi se mejora el coste aritmético a costa de empeorar las comunicaciones, cuando las matrices son pequeñas el coste de las comunicaciones es muy importante en comparación con el aritmético. Esto produce que en pruebas que hemos hecho con tamaños menores de 64 se llega a obtener mejores resultados con JaEsPaNsAnCoPi que con JaEsPaSiAnAnPi.

En las tablas 5.2.2 y 5.2.3 se muestran los valores obtenidos del speed-up y de la eficiencia. Se observa que mientras con JaEsPaNsAnCoPi no se llega a una eficiencia de 0.5, con JaEsPaSiAnANPi se sobrepasa ampliamente aunque está bastante alejada de 1 por los motivos señalados. Se aprecia que al aumentar el tamaño de las matrices aumenta la eficiencia pero al aumentar el número de procesadores disminuye, debido fundamentalmente a las difusiones de los parámetros de las rotaciones.

En la tabla 5.2.4 se muestra la eficiencia obtenida con JaEsPaSiAnCoPi y JaEsPaSiAnAnPi en iPSC/2, y en la tabla 5.2.5 la obtenida en iPSC/860. Estos resultados confirman los obtenidos en el PARSYS SN-1040.

	64	128	192	256	320
JaEsPaNsAnCoPi 2	0.923	0.931	0.942	0.964	0.965
JaEsPaSiAnAnPi 2	1.324	1.504	1.585	1.632	1.653
JaEsPaNsAnCoPi 4	1.775	1.846	1.853	1.877	1.932
JaEsPaSiAnAnPi 4	2.377	2.813	3.043	3.156	3.251
JaEsPaNsAnCoPi 8	3.218	3.506	3.674	3.748	3.783
JaEsPaSiAnAnPi 8	3.857	5.014	5.601	5.859	6.135
JaEsPaNsAnCoPi 16	5.225	6.406	6.725	7.008	7.107
JaEsPaSiAnAnPi 16	5.352	8.080	9.635	10.573	11.257

Tabla 5.2.2: Speed-up de los algoritmos JaEsPaNsAnCoPi y JaEsPaSiAnAnPi. En el PARSYS SN-1040.

	64	128	192	256	320
JaEsPaNsAnCoPi 2	0.462	0.465	0.471	0.482	0.482
JaEsPaSiAnAnPi 2	0.662	0.752	0.793	0.816	0.827
JaEsPaNsAnCoPi 4	0.444	0.462	0.463	0.469	0.483
JaEsPaSiAnAnPi 4	0.594	0.703	0.761	0.789	0.813
JaEsPaNsAnCoPi 8	0.402	0.438	0.459	0.468	0.473
JaEsPaSiAnAnPi 8	0.482	0.627	0.700	0.732	0.767
JaEsPaNsAnCoPi 16	0.327	0.400	0.420	0.438	0.444
JaEsPaSiAnAnPi 16	0.334	0.505	0.602	0.661	0.704

Tabla 5.2.3: Eficiencia de los algoritmos JaEsPaSiAnAnPi y JaEsPaNsAnCoPi. En el PARSYS SN-1040.

	64	128	192	256	320	384	448	512
JaEsPaNsAnCoPi 2	0.446	0.463	0.462	0.466	0.467	0.469	0.469	0.469
JaEsPaSiAnAnPi 2	0.702	0.806	0.837	0.856	0.866	0.874	0.879	0.882
JaEsPaNsAnCoPi 4	0.403	0.447	0.457	0.463	0.464	0.466	0.468	0.469
JaEsPaSiAnAnPi 4	0.574	0.750	0.806	0.837	0.850	0.861	0.868	0.874

Tabla 5.2.4: Eficiencia de los algoritmos JaEsPaSiAnAnPi y JaEsPaNsAnCoPi. En iPSC/2.

	64	128	192	256	320	384	448	512
JaEsPaNsAnCoPi 2	0.23	0.38	0.44	0.51	0.57	0.59	0.63	0.59
JaEsPaSiAnAnPi 2	0.21	0.45	0.61	0.74	0.84	0.92	1.00	0.95
JaEsPaNsAnCoPi 4	0.08	0.26	0.38	0.44	0.52	0.55	0.59	0.56
JaEsPaSiAnAnPi 4	0.09	0.28	0.47	0.61	0.72	0.79	0.90	0.87
JaEsPaNsAnCoPi 8	0.04	0.13	0.24	0.32	0.40	0.43	0.51	0.51
JaEsPaSiAnAnPi 8	0.03	0.13	0.26	0.37	0.49	0.61	0.71	0.71

Tabla 5.2.5: Eficiencia de los algoritmos JaEsPaSiAnAnPi y JaEsPaNsAnAnPi. En iPSC/860.

5.3 Algoritmo con almacenamiento por marcos

Si dividimos la matriz A de tamaño $n \times n$ en $\frac{n}{2}$ marcos (M_0, M_1, ... , $M_{\frac{n}{2}-1}$) tal como se vio en 5.1.2 y agrupamos los marcos M_i y $M_{\frac{n}{2}-1-i}$ en un mismo bloque B_i y disponemos de $p = \frac{n}{4b}$ procesadores, podemos asignar al procesador P_i los bloques B_j con $j = ib, ib+1, \ldots, (i+1)b-1$. Por la simetría de la matriz sólo es necesario almacenar en cada procesador P_i aquellos elementos de los bloques B_j que pertenezcan a la parte triangular superior de la matriz (o la parte triangular inferior) pero, tal como sucedía con el esquema por **antidiagonales**, será necesario almacenar algunos elementos de la parte triangular inferior dependiendo de la ordenación que se use y de las necesidades de comunicación que dicha ordenación genere. Vamos a utilizar en nuestra implementación la ordenación de Eberlein.

Si consideramos la matriz A dividida en bloques 2×2 las rotaciones de Givens se calcularán en cada caso en bloques diagonales (tal como en el esquema por **antidiagonales**), necesitándose por tanto un movimiento de filas y columnas entre dos pasos consecutivos del algoritmo.

5.3.1 Algoritmo

El esquema del método es el mismo del algoritmo 4.2.2 pero consistiendo en este caso la transferencia de datos entre procesadores vecinos en una transferencia de columnas y posteriormente una transferencia de filas. Además se usará la ordenación de Eberlein por lo que cada barrido constará de $n-1$ pasos en vez de los n que se necesitan con la ordenación par-impar.

Almacenamiento de los datos

Como ya indicamos al analizar el esquema de distribución por **marcos**, la matriz se divide en **marcos** que se agrupan en bloques de manera que todos los bloques contengan el mismo número de elementos. En la figura 5.3.1 mostramos cómo se almacenan estos marcos en los procesadores. Consideramos los bloques externos de filas y columnas (parte de esas filas y columnas) asociados a un procesador como un bloque de $\frac{n}{4p}$ filas y columnas, y lo mismo con los bloques internos. En la figura se representan los bloques almacenados en el procesador P_1. Hay que tener en cuenta que la parte triangular inferior de la matriz no se almacena pero se conoce al estar elementos simétricos en un mismo procesador. Cada uno de los bloques interno y externo se divide en dos subbloques que llamamos norte y este, con lo que en un procesador tendremos 4 bloques que llamaremos A_{en}, A_{ee}, A_{in} y A_{ie}. Estos bloques, en el procesador P_r tendrán dimensiones $\frac{n}{4p}\times\left(n-\frac{n}{2p}i\right)$, $\left(n-\frac{n}{2p}i-\frac{n}{4p}\right)\times\frac{n}{4p}$, $\frac{n}{4p}\times\left(\frac{n}{2p}i+\frac{n}{2p}\right)$ y $\left(\frac{n}{2p}i+\frac{n}{4p}\right)\times\frac{n}{4p}$, respectivamente.

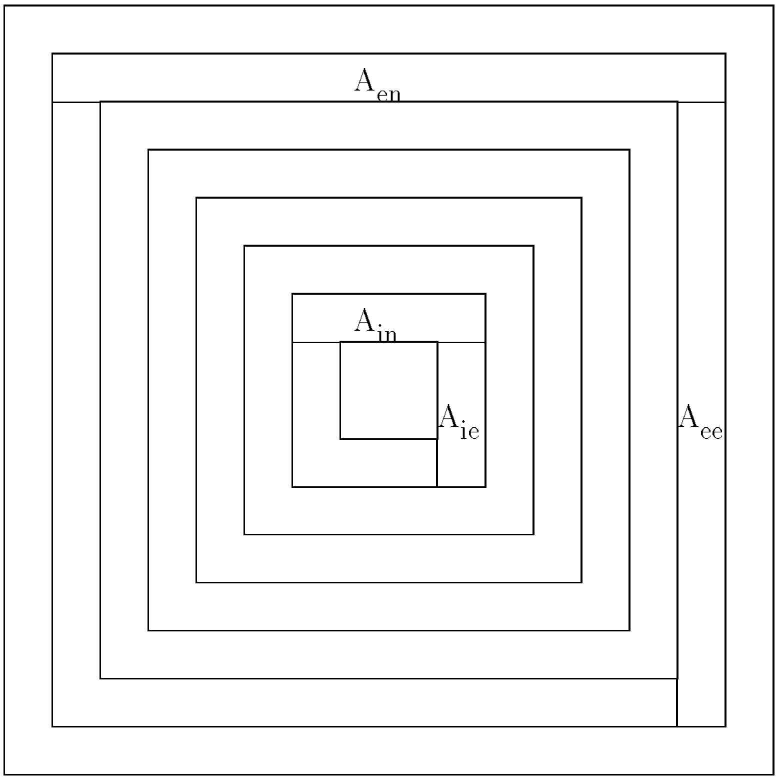


Figura 5.3.1: Almacenamiento de datos con el esquema por **marcos**.

Cálculo de rotaciones

Todos los procesadores calculan el mismo número de rotaciones, $\frac{n}{2p}$, calculando $\frac{n}{8p}$ en cada una de las submatrices (A_{kl} con $k=e,i;\ l=n,e$) que contiene. El código sería:

Algoritmo 5.3.1 *Cálculo de rotaciones* (r). *En el esquema por* ***marcos.***

PARA $j = 0, \ldots, b-1$
  calcular rotación $(r\frac{n}{8p} + j)$-ésima que anule el elemento $A_{en}[2j, 2j+1]$
FINPARA
PARA $j = 0, \ldots, b-1$
  calcular rotación $(\frac{n}{4} - (r+1)\frac{n}{8p} + j)$-ésima que anule el elemento $A_{in}[2j, 2j+1]$
FINPARA
PARA $j = 0, \ldots, b-1$
  calcular rotación $(\frac{n}{4} + r\frac{n}{8p} + j)$-ésima que anule el elemento
    $A_{ie}[filas(A_{ie}) - columnas(A_{ie}) + 2j, 2j+1]$
FINPARA
PARA $j = 0, \ldots, b-1$
  calcular rotación $(\frac{n}{2} - (r+1)\frac{n}{8p} + j)$-ésima que anule el elemento
    $A_{ee}[filas(A_{ee}) - columnas(A_{ee}) + 2j, 2j+1]$
FINPARA
(*siendo $b = \frac{n}{8p}$*)

En este algoritmo usamos las funciones $filas$ y $columnas$ que nos devuelven, respectivamente, el número de filas y columnas de una matriz.

Difusión de las rotaciones

La difusión de las rotaciones puede ser una difusión normal pero teniendo en cuenta que los parámetros que cada procesador calcula no corresponden a elementos adyacentes en la diagonal principal, por lo que tras la recepción de estos parámetros cada procesador debe reorganizarlos.

Por otro lado, el esquema de almacenamiento por **marcos** permite un pequeño ahorro en la difusión de los parámetros ya que no todos los procesadores necesitan de todos los parámetros, así, todos los procesadores necesitan los parámetros que se calculan para anular elementos de las submatrices A_{in} y A_{ie}, pero los parámetros calculados para anular elementos en A_{en} y A_{ee} en el procesador P_r no los necesitan los procesadores P_i con $r+1 \leq i \leq p-1$.

El coste de las difusiones usando una difusión normal sería de $2(p-1)\beta + (2n - \frac{2n}{p})\tau$, y haciendo una difusión adecuada al esquema por **marcos** tal como hemos indicado sería $2(p-1)\beta + (2n - \frac{5n}{2p})\tau$. Más importante que el pequeño ahorro que se pueda obtener en los tiempos teóricos de ejecución puede ser el solapamiento que se puede producir. Ilustramos la realización de una difusión de este tipo en la figura 5.3.2, donde se muestran, para 4 procesadores, los tres pasos de la difusión. Se indica en una flecha con un número el número de procesador que ha calculado los parámetros que se están mandando, y los subíndices indican de que bloque son las rotaciones que se envían (**i**nterior o **e**xterior).

Un algoritmo que utiliza esta idea es:

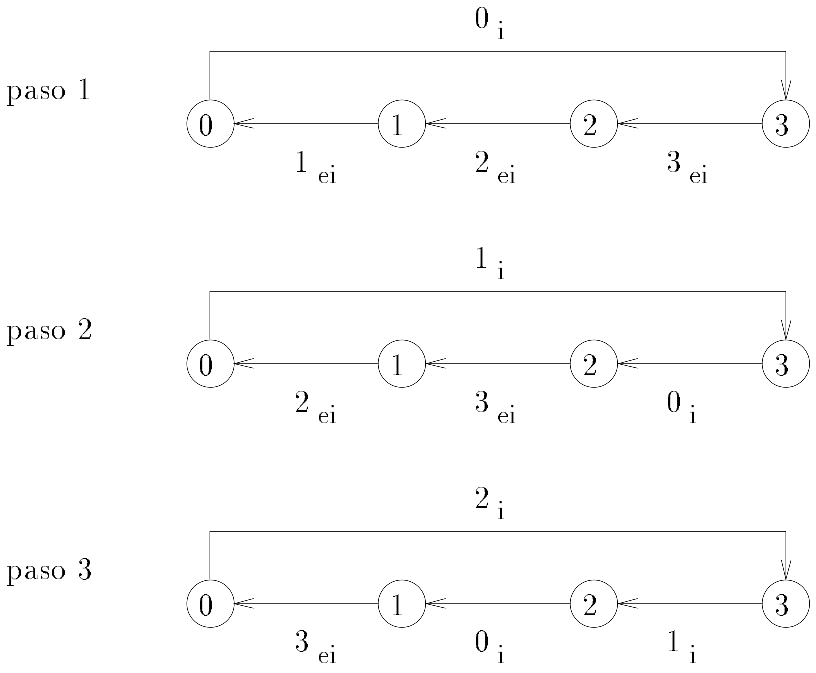


Figura 5.3.2: Difusión de los parámetros en el esquema por **marcos**.

Algoritmo 5.3.2 *Difusión de las rotaciones* (r). *En el esquema por* ***marcos.***

$frontera = p - 1$
PARA $j = 1, \ldots, p-1$
  SI $r = 0$
    enviar parámetros calculados en A_{in} y A_{ie} a P_{p-1}
    recibir parámetros calculados en A_{en}, A_{ee}, A_{in} y A_{ie} de P_1
  EN OTRO CASO SI $r = frontera$
    SI $r \ mod \ 2 = 0$
      enviar parámetros calculados en A_{en}, A_{ee}, A_{in} y A_{ie} a $P_{(r-1) \ mod \ p}$
      recibir parámetros calculados en A_{in} y A_{ie} de $P_{(r+1) \ mod \ p}$
    EN OTRO CASO
      recibir parámetros calculados en A_{in} y A_{ie} de $P_{(r+1) \ mod \ p}$
      enviar parámetros calculados en A_{en}, A_{ee}, A_{in} y A_{ie} a $P_{(r-1) \ mod \ p}$
    FINSI
  EN OTRO CASO SI $r > frontera$
    SI $r \ mod \ 2 = 0$
      enviar parámetros calculados en A_{in} y A_{ie} a $P_{(r-1) \ mod \ p}$
      recibir parámetros calculados en A_{in} y A_{ie} de $P_{(r+1) \ mod \ p}$
    EN OTRO CASO
      recibir parámetros calculados en A_{in} y A_{ie} de $P_{(r+1) \ mod \ p}$
      enviar parámetros calculados en A_{in} y A_{ie} a $P_{(r-1) \ mod \ p}$
    FINSI
  EN OTRO CASO
    SI $r \ mod \ 2 = 0$
      enviar parámetros calculados en A_{en}, A_{ee}, A_{in} y A_{ie} a $P_{(r-1) \ mod \ p}$
      recibir parámetros calculados en A_{en}, A_{ee}, A_{in} y A_{ie} de $P_{(r+1) \ mod \ p}$
    EN OTRO CASO
      recibir parámetros calculados en A_{en}, A_{ee}, A_{in} y A_{ie} de $P_{(r+1) \ mod \ p}$
      enviar parámetros calculados en A_{en}, A_{ee}, A_{in} y A_{ie} a $P_{(r-1) \ mod \ p}$
    FINSI
  FINSI
  $frontera = frontera - 1$
FINPARA

La variable $frontera$ la usaremos para determinar, en cada paso del algoritmo, qué procesadores envían o reciben todos los parámetros de las rotaciones y cuales envían o reciben sólo la mitad de los parámetros.

Actualización de la matriz

En la actualización todos los procesadores realizan la misma cantidad de trabajo actualizando cada uno las cuatro submatrices A_{kl}, con $k = e, i$; $l = n, e$, que contiene, utilizando para ello las rotaciones que ha recibido.

No veremos el código de este procedimiento pues es similar al del esquema por **antidiagonales** (algoritmo 5.2.4), pero en este caso el esquema de almacenamiento es más compacto, lo que hace que se puedan usar las rutinas de BLAS 1 para reducir fácilmente el tiempo de ejecución.

Transferencia de filas y columnas

Para seguir teniendo elementos simétricos en un mismo procesador, necesitamos transferir columnas y filas, pues si sólo se transfieren columnas se perdería la simetría y además los nuevos elementos a anular no estarían en la diagonal principal de bloques 2×2.

Analizaremos la transferencia de filas y columnas considerando $p = 4$ procesadores y $p = \frac{n}{4b}$, con lo que cada procesador contendrá b bloques de **marcos** (cada bloque constituido por un **marco** interno y otro externo, y estando formado cada **marco** por elementos de dos filas y columnas).

Empezamos analizando la transferencia en un paso impar. En el algoritmo se realizará primero la transferencia de columnas y después la de filas, por lo que analizaremos primero la de filas y después la de columnas, tal como hicimos con el esquema de almacenamiento por **antidiagonales**.

En la figura 5.3.3 se muestra cómo se realizaría la transferencia de filas en este caso (suponemos que $b \geq 2$). Se representan los ocho **marcos** constituido cada uno de ellos por b **marcos** de pares de filas y columnas, cada procesador contiene dos **marcos** en la figura, uno interno y otro externo, y los números escritos dentro de los **marcos** representan el número de procesador donde está almacenado el **marco**. Aunque en la figura se representa toda la matriz consideraremos que sólo se trabaja con la parte triangular superior y posiblemente con algunos elementos de la parte triangular inferior que se necesiten para la transferencia de datos. Habrá un movimiento de datos de acuerdo con la ordenación de Eberlein, con lo que se moverán los datos de la fila 1 a las posiciones de la fila 2 (consideramos las filas y columnas numeradas de 0 a $n-1$), los de la fila 2 a la 4, etc , algunos de estos movimientos de datos serán entre procesadores y por tanto conllevarán comunicaciones, pero otros serán internos; además, sólo se necesitará mover datos que vayan a parar a la zona triangular superior de la matriz, que es la que necesitamos tener actualizada tras el movimiento de datos (la transferencia de columnas y filas). Así, en cada marco aparecen rectángulos que corresponden a transferencias de datos entre los procesadores. Cada rectángulo representa una fila que debe ser transferida de acuerdo con el movimiento de índices de la ordenación de Eberlein en los pasos impares. Cada rectángulo tiene asociada una expresión que representa el número de elementos que contiene. Además del movimiento de datos entre los procesadores, representado en la figura, habría otro movimiento de datos interno en cada uno de los procesadores. Por ejemplo, las filas que se representan como rectángulos continúan hacia la derecha en la figura y los elementos en esta continuación deben moverse a la misma fila a la que se mueven los elementos en los rectángulos, pero ese movimiento es interno. Se observa que las comunicaciones no están balanceadas en la transferencia de filas debido a que los procesadores extremos realizan menos comunicaciones que los internos, así, el procesador P_0 envía $n-2\frac{n}{4p}$ datos y recibe $2\frac{n}{4p}$, el procesador P_1 envía $n-2\frac{n}{4p}$ datos y recibe $n+2\frac{n}{4p}$, el procesador P_2 envía $n-2\frac{n}{4p}$ datos y recibe $n+2\frac{n}{4p}$, y el procesador P_3 envía $6\frac{n}{4p}$ datos y recibe $n-6\frac{n}{4p}$; por tanto, el coste de la transferencia de filas en un paso impar es de $2\beta+2n\tau$.

En la figura 5.3.4 se muestra cómo se realizaría la transferencia de columnas en un paso impar. En cada marco aparecen rectángulos que corresponden a transferencias de datos entre los procesadores. Cada rectángulo representa una columna que debe ser transferida de acuerdo con el movimiento de índices de la ordenación de Eberlein en los pasos impares. Cada rectángulo tiene asociada una expresión que representa el número de elementos que contiene. Además del movimiento de datos entre los procesadores, representado en la figura, habría otro movimiento de datos interno en cada uno de los procesadores, tal como explicamos en la transferencia de filas. Los rectángulos de longitud dos corresponden a dos elementos de la parte triangular inferior que deben transferirse entre procesadores para actualizar datos de la parte triangular superior, y estos elementos de la parte triangular inferior los conocen los procesadores correspondientes porque contienen sus simétricos al usarse un almacenamiento por **marcos**. La transferencia de columnas tampoco está balanceada, así, el procesador P_0 envía $2\frac{n}{4p}+2$ datos y recibe $n-2\frac{n}{4p}+2$, el procesador P_1 envía $n+2\frac{n}{4p}+4$ datos y recibe $n-2\frac{n}{4p}+4$, el procesador P_2 envía $n+2\frac{n}{4p}+4$ datos y recibe $n-2\frac{n}{4p}+4$, y el procesador P_3 envía $n-6\frac{n}{4p}+2$ datos y recibe $6\frac{n}{4p}+2$; por tanto, el coste de la transferencia de filas en un paso impar es de $2\beta+(2n+8)\tau$.

En la figura 5.3.5 se muestra cómo se realizaría la transferencia de filas en los pasos pares. Habrá un movimiento de datos de acuerdo con la ordenación de Eberlein, con lo que se moverán los datos de la fila 1 a las posiciones de la fila $n-1$, los de la fila 3 a la 1, etc , algunos de estos movimientos de datos serán entre procesadores y por tanto conllevarán comunicaciones, pero otros serán internos; además, sólo se necesitará mover datos que vayan a parar a la zona triangular superior de la matriz, que es la que necesitamos tener actualizada tras el movimiento de datos (la transferencia de columnas y filas). Además del movimiento de datos entre los procesadores, representado en la figura, habría otro movimiento de datos interno en cada uno de los procesadores. En esta transferencia de filas se envían datos de la parte triangular inferior (los que están en triángulos de base 2), por lo que en la transferencia de columnas habrá que tener en cuenta que deben actualizarse, además de los elementos de la parte triangular superior, los elementos de la parte triangular inferior que se usan en la transferencia de filas para actualizar elementos de la parte triangular superior. Tampoco en este caso las comunicaciones están balanceadas. El procesador P_0 envía $2\frac{n}{4p}+2$ datos y recibe $n-2\frac{n}{4p}+2$, el procesador P_1 envía $n+2\frac{n}{4p}+4$ datos y recibe $n-2\frac{n}{4p}+4$, el procesador

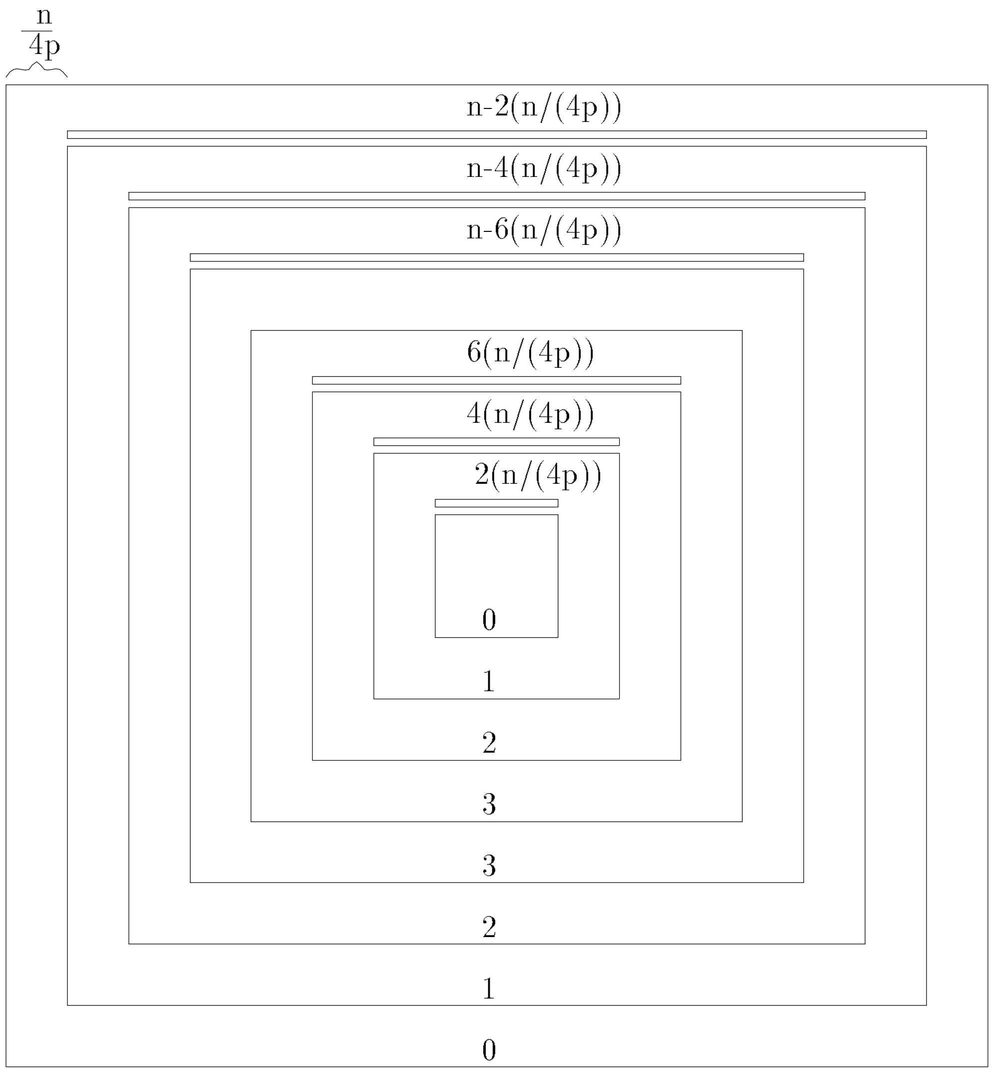


Figura 5.3.3: Transferencia de filas en un paso impar con el almacenamiento por **marcos** y la ordenación de Eberlein

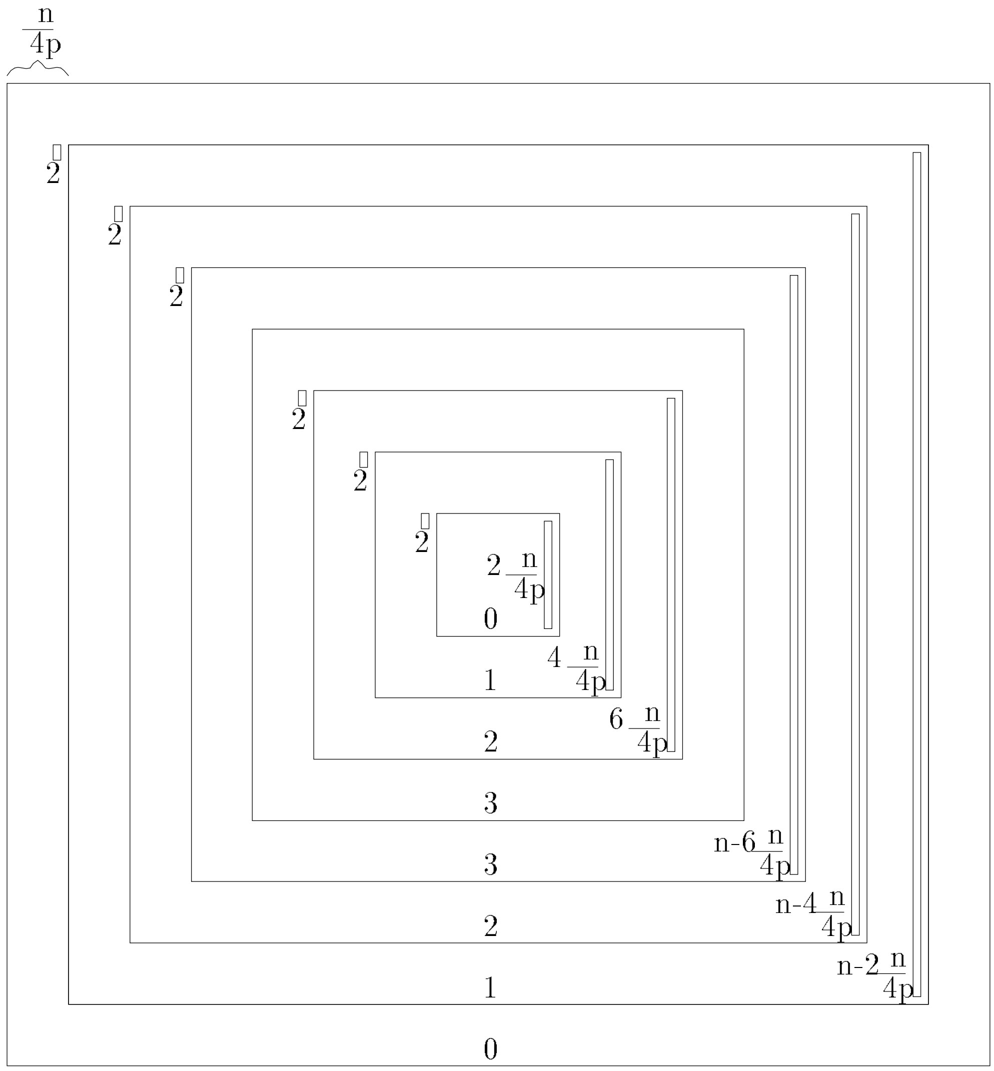


Figura 5.3.4: Transferencia de columnas en un paso impar con el almacenamiento por **marcos** y la ordenación de Eberlein

P_2 envía $n + 2\frac{n}{4p} + 4$ datos y recibe $n - 2\frac{n}{4p} + 4$, y el procesador P_3 envía $n - 6\frac{n}{4p} + 2$ datos y recibe $6\frac{n}{4p} + 2$; por tanto, el coste de la transferencia de filas en un paso par es de $2\beta + (2n+8)\tau$.

Por último, en la figura 5.3.6 se muestra cómo se realizaría la transferencia de columas en los pasos pares. Además del movimiento de datos entre los procesadores, representado en la figura, habría otro movimiento de datos interno en cada uno de los procesadores. En esta transferencia de columnas se envían datos de la parte triangular inferior (los que actualizan datos de los bloques de tamaño 2 en la figura 5.3.5). Tampoco en este caso las comunicaciones están balanceadas. El procesador P_0 envía $n - 2\frac{n}{4p} + 2$ datos y recibe $2\frac{n}{4p} + 2$, el procesador P_1 envía $n - 2\frac{n}{4p} + 4$ datos y recibe $n + 2\frac{n}{4p} + 4$, el procesador P_2 envía $n - 2\frac{n}{4p} + 4$ datos y recibe $n + 2\frac{n}{4p} + 4$, y el procesador P_3 envía $6\frac{n}{4p} + 2$ datos y recibe $n - 6\frac{n}{4p} + 2$; por tanto, el coste de la transferencia de columnas en un paso impar es de $2\beta + (2n+8)\tau$.

Por tanto, al haber $n-1$ pasos en cada barrido, $\frac{n}{2}$ pasos impares y $\frac{n}{2} - 1$ pares, el coste total por barrido de las comunicaciones en la transferencia de filas y columnas será:

$$(4n-4)\beta + (4n^2 + 8n - 16)\tau \tag{5.3.1}$$

5.3.2 Costes teóricos

Para el cálculo del coste aritmético por barrido hay que tener en cuenta que se calculan $\frac{n}{2p}$ rotaciones por procesador en cada paso, lo que representa un coste de $\frac{11n}{2p}$ flops, y que cada procesador actualiza $\frac{n(n+1)}{2p}$ datos en cada paso, lo que representa $\frac{3n(n+1)}{p}$ flops. De este modo, tenemos un coste de $\frac{3n^2}{p} + \frac{17n}{2p}$ flops por cada paso, y hay un total de $n-1$ pasos por barrido. Además, al final de cada barrido hay que añadir $\frac{n^2}{p} - \frac{n}{p}$ del cálculo de $off(A)$. En total tendremos un coste aritmético por barrido de:

$$T_a = \frac{3n^3}{p} + \frac{12n^2}{p} - \frac{15n}{2p} \qquad flops \tag{5.3.2}$$

El coste de las comunicaciones, teniendo en cuenta los costes de las difusiones y de la transferencia de los datos (estudiados en subsecciones anteriores) y que en el cálculo y comunicación de $off(A)$ se tiene un coste de $4p(\beta + \tau)$ y que sólo representamos los términos de mayor orden, sería:

$$T_c = 2np\beta + \left(6n^2 - \frac{2n^2}{p}\right)\tau \tag{5.3.3}$$

si se usa la difusión normal, y:

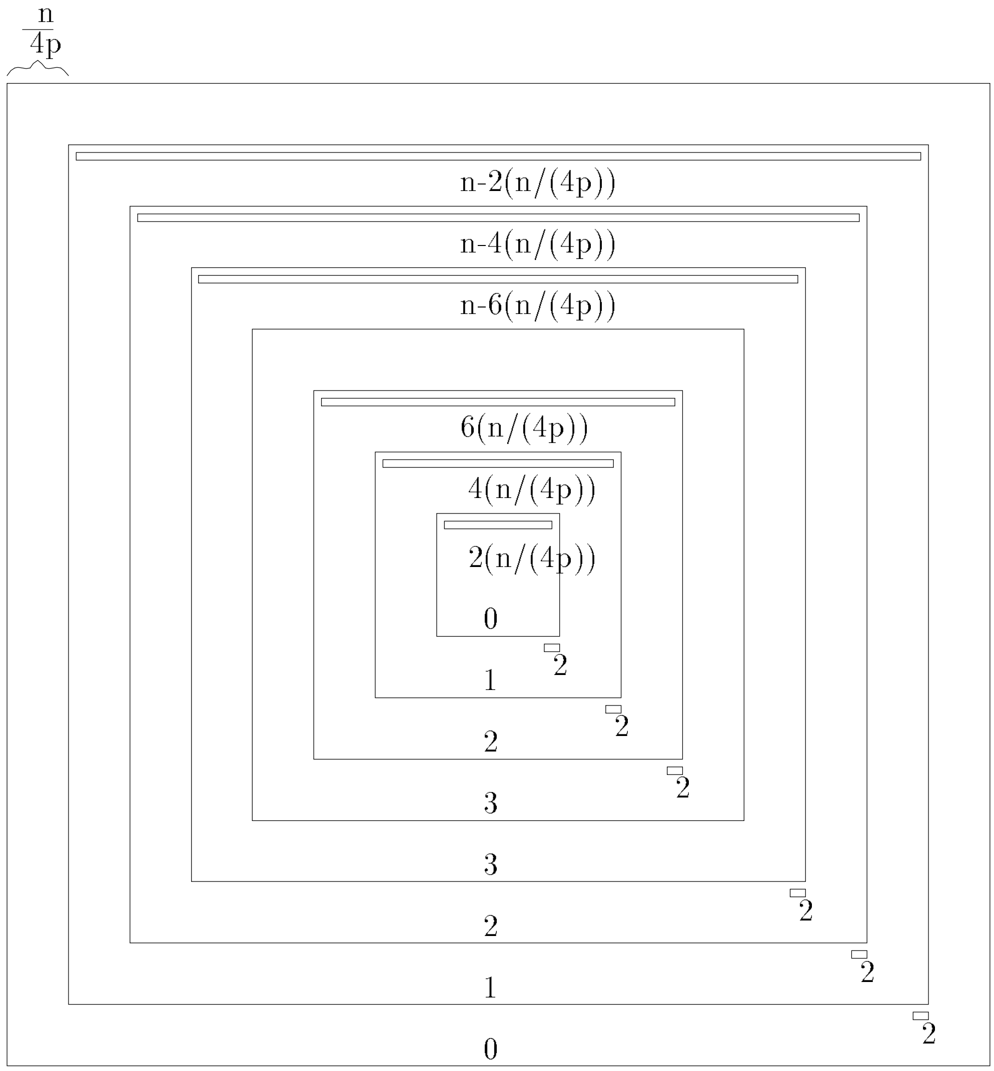


Figura 5.3.5: Transferencia de filas en un paso par con el almacenamiento por **marcos** y la ordenación de Eberlein

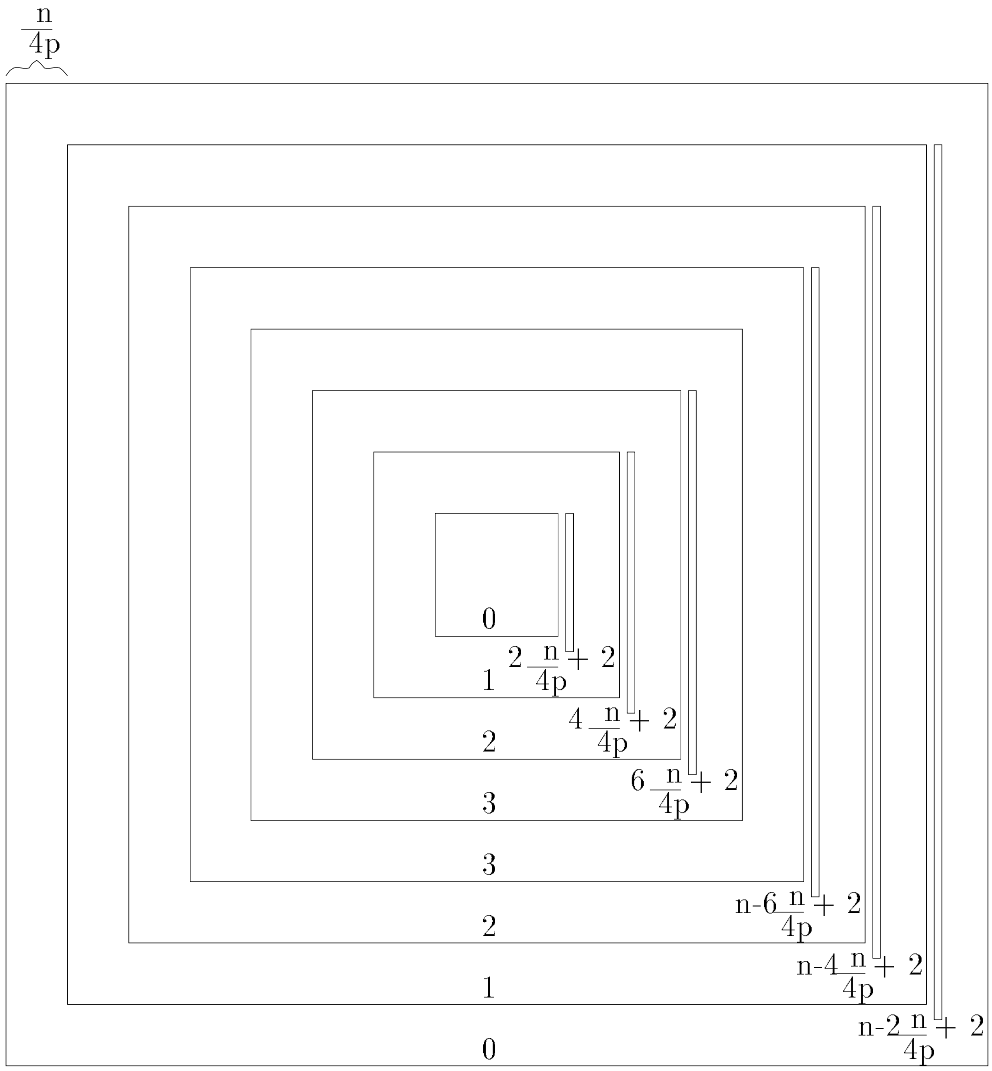


Figura 5.3.6: Transferencia de columnas en un paso par con el almacenamiento por **marcos** y la ordenación de Eberlein

$$T_c = 2np\beta + \left(6n^2 - \frac{5n^2}{2p}\right)\tau \tag{5.3.4}$$

si se usa la difusión adecuada al esquema de almacenamiento que estamos usando. Los costes son prácticamente los mismos en los dos casos habiendo diferencias sólo en términos de menor orden.

5.3.3 Resultados experimentales

Presentamos resultados experimentales obtenidos en un iPSC/860 con programas en C con primitivas de comunicación, y con matrices densas generadas aleatoriamente con valores entre -10 y 10. Se mostrarán tiempos de ejecución en segundos por barrido, ya que el número de barridos es el mismo con los distintos algoritmos y es del orden de $log_2\, n$.

En la tabla 5.3.1 se representan los tiempos obtenidos con un algoritmo secuencial que explota la simetría (JaEsSeSiFi), con el algoritmo que explota la simetría usando el esquema por **antidiagonales** con la ordenación par-impar (JaEsPaSiAnAnPi) y con el algoritmo que explota la simetría usando el esquema de almacenamiento por marcos y la ordenación de Eberlein (JaEsPaSiAnMaEb). Se puede observar que JaEsPaSiAnMaEb da mejores prestaciones que JaEsPaSiAnAnPi, y esto es debido a dos factores: que la ordenación de Eberlein es óptima y la par-impar no, y que el esquema por **marcos** produce un almacenamiento más compacto y por tanto un mejor uso de la memoria. De este modo, con JaEsPaSiAnMaEb se llegan a obtener buenas eficiencias (tabla 5.3.2). En algunos casos se llegan a tener eficiencias por encima de 1, a causa de que al usar más procesadores se utiliza menos memoria en cada uno de ellos y los accesos a memoria son más rápidos.

	128	192	256	320	384	448	512
JaEsPaSiAnAnPi 2	1.86	4.80	9.43	17.34	28.04	43.05	66.70
JaEsPaSiAnMaEb 2	1.59	3.59	7.43	13.63	22.23	34.60	50.64
JaEsPaSiAnAnPi 4	1.50	3.09	5.71	10.15	16.26	23.97	36.55
JaEsPaSiAnMaEb 4	1.39	2.99	5.59	9.19	14.39	21.21	30.97
JaEsPaSiAnAnPi 8	1.67	2.80	4.68	7.45	10.64	15.24	22.22
JaEsPaSiAnMaEb 8	1.64	2.80	4.90	7.21	10.40	14.59	19.99
JaEsSeSiFi	1.67	5.85	13.96	29.13	51.54	86.26	126.67

Tabla 5.3.1: Tiempos de ejecución por barrido (en segundos) de los algoritmo JaEsPaSiAnAnPi y JaEsPaSiAnMaEb. En iPSC/860.

	128	192	256	320	384	448	512
JaEsPaSiAnMaEb 2	0.53	0.81	0.94	1.07	1.16	1.25	1.25
JaEsPaSiAnMaEb 4	0.30	0.49	0.62	0.79	0.90	1.02	1.02
JaEsPaSiAnMaEb 8	0.13	0.26	0.36	0.51	0.62	0.74	0.79

Tabla 5.3.2: Eficiencia del algoritmo JaEsPaSiAnMaEb. En iPSC/860.

En la tabla 5.3.3 comparamos los resultados obtenidos con el algoritmo que usa el broadcast normal (JaEsPaSiAnMaEbBn) y el que usa el broadcast mejorado (JaEsPaSiAnMaEb). Se observan unas pequeñas diferencias a favor de JaEsPaSiAnMaEb cuando el número de procesadores y el tamaño de la matriz crecen, pero estas diferencias no se pueden considerar significativas.

	128	192	256	320	384	448	512
JaEsPaSiAnMaEbBn 2	1.78	3.79	7.81	13.81	22.80	34.76	52.00
JaEsPaSiAnMaEb 2	1.59	3.59	7.43	13.63	22.23	34.60	50.64
JaEsPaSiAnMaEbBn 4	1.54	3.18	5.59	9.38	14.94	21.98	31.19
JaEsPaSiAnMaEb 4	1.39	2.99	5.59	9.19	14.39	21.21	30.97
JaEsPaSiAnMaEbBn 8	1.59	2.86	4.66	7.14	10.39	14.61	20.21
JaEsPaSiAnMaEb 8	1.64	2.80	4.90	7.21	10.40	14.59	19.99
JaEsPaSiAnMaEbBn 16			5.07		9.45		16.15
JaEsPaSiAnMaEb 16			4.74		9.36		15.97

Tabla 5.3.3: Tiempos de ejecución por barrido (en segundos) de los algoritmos JaEsPaSiAnMaEbBn y JaEsPaSiAnMaEb. En iPSC/860.

Como ya hemos dicho, el esquema de almacenamiento por **marcos** nos proporciona un esquema de almacenamiento compacto con el que se sacará provecho del uso de BLAS 1 en la actualización de la matriz. Esto se muestra en la tabla 5.3.4, donde se muestran los tiempos obtenidos no usando BLAS (JaEsPaSiAnMaEb) y usando BLAS 1 en el movimiento de datos y en la aplicación de las rotaciones de Givens (JaEsPaSiAnMaEbB1). Se observa que el uso de BLAS permite una reducción importante en el tiempo de ejecución cuando el tamaño de la matriz aumenta, al ser en ese caso el BLAS más eficiente, y que esta mejora con el uso de BLAS disminuye al aumentar el número de procesadores, ya que en este caso las submatrices con las que trabaja cada procesador son más pequeñas.

Por último, en la tabla 5.3.5 se comparan los resultados obtenidos en anillo (JaEsPaSiAnMaEbB1) y en hipercubo (JaEsPaSiHiMaEbB1) usando en ambos casos BLAS

	128	192	256	320	384	448	512
JaEsPaSiAnMaEbB1 2	1.39	2.83	5.60	9.60	15.85	24.08	35.10
JaEsPaSiAnMaEb 2	1.59	3.59	7.43	13.63	22.23	34.60	50.64
JaEsPaSiAnMaEbB1 4	1.57	2.89	4.59	7.19	11.30	16.22	22.85
JaEsPaSiAnMaEb 4	1.39	2.99	5.59	9.19	14.39	21.21	30.97
JaEsPaSiAnMaEbB1 8	1.60	2.77	4.46	6.19	9.40	12.16	16.78
JaEsPaSiAnMaEb 8	1.64	2.80	4.90	7.21	10.40	14.59	19.99
JaEsPaSiAnMaEbB1 16			4.67		8.59		14.99
JaEsPaSiAnMaEb 16			4.74		9.36		15.97

Tabla 5.3.4: Tiempos de ejecución por barrido (en segundos) de los algoritmos JaEsPaSiAnMaEb y JaEsPaSiAnMaEbB1. En iPSC/860.

1, y utilizando en hipercubo el anillo inmerso para las transferencias de filas y columnas, y diferenciándose los dos algoritmos sólo en el broadcast. Como es lógico, se obtienen menores tiempos de ejecución en hipercubo al aumentar el número de procesadores.

	128	192	256	320	384	448	512
JaEsPaSiAnMaEbB1 2	1.39	2.83	5.60	9.60	15.85	24.08	35.10
JaEsPaSiHiMaEbB1 2	1.39	2.79	5.45	9.67	15.67	23.59	35.10
JaEsPaSiAnMaEbB1 4	1.57	2.89	4.59	7.19	11.30	16.22	22.85
JaEsPaSiHiMaEbB1 4	1.59	2.79	4.59	7.18	10.98	15.82	22.47
JaEsPaSiAnMaEbB1 8	1.60	2.77	4.46	6.19	9.40	12.16	16.78
JaEsPaSiHiMaEbB1 8	1.40	2.37	3.99	5.99	8.79	11.78	15.78
JaEsPaSiAnMaEbB1 16			4.67		8.59		14.99
JaEsPaSiHiMaEbB1 16			3.74		7.39		12.80

Tabla 5.3.5: Tiempos de ejecución por barrido (en segundos) de los algoritmos JaEsPaSiAnMaEbB1 y JaEsPaSiHiMaEbB1. En iPSC/860.

5.4 Conclusiones

En este capítulo hemos analizado las ideas generales para explotar la simetría de la matriz en topologías lógicas de tipo de anillo (anillo unidireccional o bidireccional, arrays lineales, arrays con un host central, ...). La idea básica consiste en considerar elementos simétricos en un mismo procesador, con lo que será suficiente trabajar con la parte triangular superior o inferior de la matriz (y almacenar sólo esa parte) y con algunos elementos adicionales dependiendo de la ordenación y la topología que se usen. La combinación de un esquema de almacenamiento con una determinada ordenación generará unas necesidades de comunicación que determinarán la topología lógica para la que es adecuado el algoritmo.

Hemos analizado con detalle los casos de combinar el esquema de almacenamiento por antidiagonales con la ordenación par-impar y el esquema de almacenamiento por marcos con la ordenación de Eberlein. Los resultados que se obtienen en ambos casos son similares y siempre mucho mejores que los obtenidos con métodos que no exploten la simetría.

Capítulo 6

EXPLOTACION DE LA SIMETRIA EN MALLA

Los métodos de Jacobi que estudiamos en el capítulo anterior utilizan un esquema de almacenamiento adecuado para un anillo lógico de procesadores, y como consecuencia de esto tienen el inconveniente de una baja escalabilidad. En este capítulo estudiaremos cómo es posible diseñar algoritmos de Jacobi que exploten la simetría en una malla lógica de procesadores, obteniéndose así una eficiencia teórica del 100% y una mejor escalabilidad que en los algoritmos diseñados para un anillo lógico.

Estudiaremos dos algoritmos que explotan la simetría en una malla. El primero de ellos utiliza la ordenación Round-Robin y el segundo la par-impar. El segundo de los algoritmos utiliza el esquema por bloques que estudiamos en el capítulo 1 y que aplicamos en el capítulo 3 para obtener algoritmos para multiprocesadores de memoria compartida. La combinación del esquema por bloques con una ordenación adecuada (la par-impar) y con un esquema de almacenamiento que permite explotar la simetría en una malla lógica de procesadores, da lugar a un algoritmo que además de teóricamente óptimo (en cuanto a eficiencia) y con una buena escalabilidad, permite obtener buenas prestaciones cuando se resuelven problemas de grandes dimensiones.

6.1 Introducción

Estamos interesados en la explotación de la simetría en una malla de procesadores en métodos de Jacobi para el Problema Simétrico de Valores Propios.

Es posible utilizar un esquema de distribución de los datos que se ha usado con éxito en el problema de resolución de sistemas triangulares [17] y que nos permitiría en nuestro caso trabajar con la parte triangular superior (o inferior) de la matriz.

Si se dispone de $p = r^2$ procesadores se puede dividir la matriz A, de tamaño $n \times n$, en bloques A_{ij} con $i = 0, 1, \ldots, kr-1$, $j = 0, 1, \ldots, kr-1$, cada uno de ellos de tamaño $\frac{n}{kr} \times \frac{n}{kr}$, y siendo k un número natural tal que n sea múltiplo de kr.

Si se asignan a cada procesador P_{ij} los bloques $A_{i+mr,j+qr}$ con $m = 0, 1, \ldots, k-1$, $q = 0, 1, \ldots, k-1$, se obtiene una distribución que se muestra en la figura 6.1.1 para $k = 2$ y $r = 3$. En esta figura los subíndices representan los bloques de la matriz A y los superíndices los procesadores donde se almacenan los bloques. Sólo se muestran bloques de la parte triangular superior de la matriz.

$$
\begin{array}{cccccc}
A^{00}_{00} & A^{01}_{01} & A^{02}_{02} & A^{00}_{03} & A^{01}_{04} & A^{02}_{05} \\
 & A^{11}_{11} & A^{12}_{12} & A^{10}_{13} & A^{11}_{14} & A^{12}_{15} \\
 & & A^{22}_{22} & A^{20}_{23} & A^{21}_{24} & A^{22}_{25} \\
 & & & A^{00}_{33} & A^{01}_{34} & A^{02}_{35} \\
 & & & & A^{11}_{44} & A^{12}_{45} \\
 & & & & & A^{22}_{55}
\end{array}
$$

Figura 6.1.1: Distribución de datos con $k = 2$ y $r = 3$.

De este modo, cada procesador contiene datos de la matriz A y trabaja en la actualización de la matriz, pero la carga no está balanceada, y así, en la figura 6.1.1 algunos procesadores contienen 3 bloques y otros sólo uno. Supuesto que se utilice un esquema de algoritmo similar al estudiado en el capítulo 4 en una malla de procesadores pero sin explotar la simetría de la matriz, se necesitaría, tras la actualización de la matriz, de una transferencia de filas y columnas entre procesadores vecinos. Si se divide la matriz en bloques más pequeños, al hacer la transferencia de filas y columnas habrá más transferencias externas (se necesita transferir $\frac{6n}{\sqrt{p}}$ datos desde P_{01} cuando se transfieren columnas, y con el algoritmo no simétrico se transfieren $\frac{4n}{\sqrt{p}}$). De este modo, este esquema de almacenamiento que es adecuado para otros problemas [17], no parece adecuado para el problema de valores propios. Si se aumenta k para obtener una mejor distribución del trabajo en la actualización de la matriz, el coste de las comunicaciones tiende a ser del mismo orden que el aritmético. Para solventar este problema, en la siguiente sección mostraremos que es posible explotar la simetría manteniendo el mismo coste de las comunicaciones que en el algoritmo que no explota la simetría.

6.2 Algoritmo con almacenamiento por plegamiento

6.2.1 Esquema de almacenamiento

Si se dispone de $p = r^2$ procesadores organizados en una malla cuadrada, para obtener la distribución de los datos en la malla se divide la matriz A en bloques A_{ij} con $i = 0, 1, \ldots, 2r-1$, $j = 0, 1, \ldots, 2r-1$, y se hace la siguiente distribución entre los procesadores:
bloques A_{ii}, $A_{2r-1-i,2r-1-i}$ y $A_{i,2r-1-i}$, con $i = 0, 1, \ldots, r-1$, al procesador $P_{i,r-1-i}$,

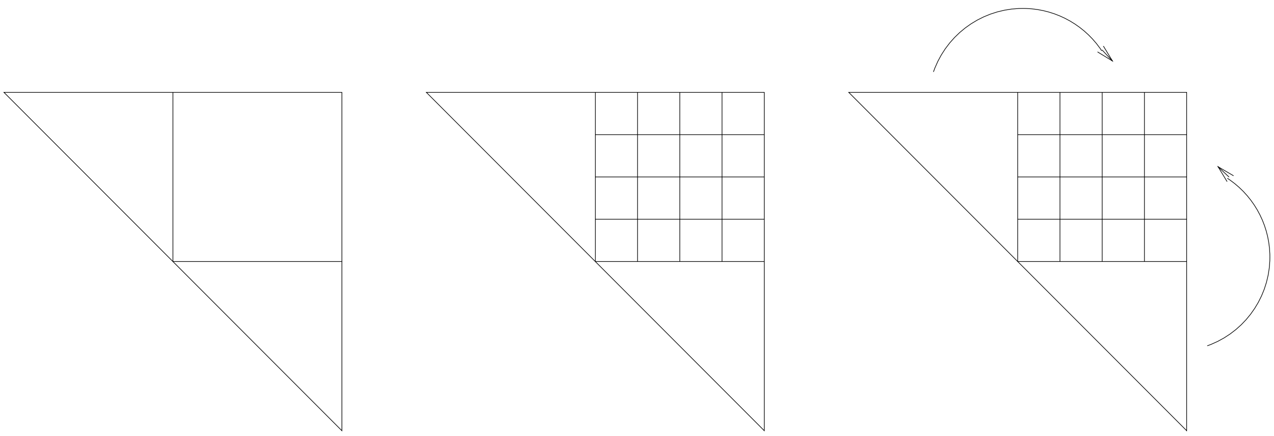
bloques $A_{i,j+r}$ y $A_{i,r-1-j}$, con $i+j<r-1$, a P_{ij},
y bloques $A_{i,j+r}$ y $A_{2r-1-i,j+r}$, con $i+j>r-1$, a P_{ij}.

Mostramos esta distribución en la figura 6.2.1, con $r=3$.

$$
\begin{array}{cccccc}
A_{00}^{02} & A_{01}^{01} & A_{02}^{00} & A_{03}^{00} & A_{04}^{01} & A_{05}^{02} \\
 & A_{11}^{11} & A_{12}^{10} & A_{13}^{10} & A_{14}^{11} & A_{15}^{12} \\
 & & A_{22}^{20} & A_{23}^{20} & A_{24}^{21} & A_{25}^{22} \\
 & & & A_{33}^{20} & A_{34}^{21} & A_{35}^{22} \\
 & & & & A_{44}^{11} & A_{45}^{12} \\
 & & & & & A_{55}^{02}
\end{array}
$$

Figura 6.2.1: Distribución de datos con $r=3$.

A este esquema de almacenamiento le llamamos por **plegamiento**. La distribución de la matriz en la malla cuadrada de procesadores se puede obtener dividiendo la matriz A en cuatro submatrices cuadradas de tamaño $\frac{n}{2}\times\frac{n}{2}$ (figura 6.2.2.a), asignando la matriz de la parte superior derecha a los procesadores según la figura 6.2.2.b), y **plegando** las submatrices triangulares superiores sobre la malla tal como se muestra en la figura 6.2.2.c).



a) b) c)

Figura 6.2.2: Almacenamiento de los datos con el esquema de almacenamiento por **plegamiento**.

6.2.2 Algoritmo

Utilizaremos el esquema de almacenamiento por **plegamiento** antes descrito y la ordenación Round-Robin que tiene un patrón constante de movimiento de índices (y por tanto de movimiento de filas y columnas) entre dos pasos consecutivos del algoritmo, por lo que nos simplifica el diseño del algoritmo.

El esquema usado para calcular los valores propios es similar al del algoritmo paralelo que no explota la simetría.

Algoritmo 6.2.1 *Esquema de un barrido del método JaEsPaSiMaPlRr.*

EN PARALELO: PARA $i = 0, \dots, r-1$; $j = 0, \dots, r-1$ EN P_{ij}:
  REPETIR
    PARA $k = 1, \dots, n-1$
      SI $i = \sqrt{p} - j - 1$
        Calcular rotaciones (i)
      FINSI
      Difundir parámetros (i, j)
      Actualizar matriz (i, j)
      Transferir columnas (i, j)
      Transferir filas (i, j)
    FINPARA
  HASTA $off(A) < COTA$

Y los procedimientos usados en este esquema tienen los siguientes códigos:

Algoritmo 6.2.2 *Calcular rotaciones (i). Sólo trabajan los procesadores en la antidiagonal principal.*

PARA $m = 0, 2, 4, \dots, \frac{n}{2r} - 2$
  Computar parámetros de rotaciones que anulan $a_{m,m+1}$ y $a_{m+1,m}$ de A_{ii}
FINPARA
PARA $m = 0, 2, 4, \dots, \frac{n}{2r} - 2$
  Computar parámetros de rotaciones que anulan $a_{m,m+1}$
    y $a_{m+1,m}$ de $A_{2r-1-i,2r-1-i}$
FINPARA

Algoritmo 6.2.3 *Difundir parámetros (i,j).*

*(*En cada procesador los parámetros 1 son los parámetros obtenidos en el primer bucle de* ***calcular rotaciones*** *y los parámetros 2 son los obtenidos en el segundo bucle.*)*

SI $j = 0$ y $i = r-1$
  enviar parámetros 1 y 2 a $P_{r-1,1}$

```
    enviar parámetros 1 y 2 a P_{r-2,0}
EN OTRO CASO SI i = 0 y j = r - 1
    enviar parámetros 1 a P_{0,r-2}
    enviar parámetros 2 a P_{1,r-1}
EN OTRO CASO SI i = r - 1 - j
    SI i < r/2
        enviar parámetros 1 a P_{i,j-1}
        enviar parámetros 2 a P_{i+1,j}
        enviar parámetros 1 y 2 a P_{i,j+1}
        enviar parámetros 1 y 2 a P_{i-1,j}
    EN OTRO CASO
        enviar parámetros 1 y 2 a P_{i,j+1}
        enviar parámetros 1 y 2 a P_{i-1,j}
        enviar parámetros 1 a P_{i,j-1}
        enviar parámetros 2 a P_{i+1,j}
    FINSI
EN OTRO CASO SI i < r/2
    SI i < r - 1 - j
        recibir parámetros de P_{i,j+1}
        SI j ≠ 0
            enviar parámetros recibidos a P_{i,j-1}
        FINSI
        recibir parámetros de P_{i+1,j}
        SI i ≠ 0
            enviar parámetros recibidos a P_{i-1,j}
        FINSI
    EN OTRO CASO
        recibir parámetros de P_{i-1,j}
        enviar parámetros recibidos a P_{i+1,j}
        recibir parámetros de P_{i,j-1}
        SI j ≠ r - 1
            enviar parámetros recibidos a P_{i,j+1}
        FINSI
    FINSI
EN OTRO CASO
    SI i < r - 1 - j
        recibir parámetros de P_{i+1,j}
        enviar parámetros recibidos a P_{i-1,j}
        recibir parámetros de P_{i,j+1}
        SI j ≠ 0
            enviar parámetros recibidos a P_{i,j-1}
```

```
        FINSI
      EN OTRO CASO
        recibir parámetros de P_{i,j-1}
        SI j ≠ r - 1
          enviar parámetros recibidos a P_{i,j+1}
        FINSI
        recibir parámetros de P_{i-1,j}
        SI i ≠ r - 1
          enviar parámetros recibidos a P_{i+1,j}
        FINSI
      FINSI
    FINSI
```

El orden de las comunicaciones en cada procesador se muestra en la figura 6.2.3 en una malla con 16 procesadores. Las flechas tienen un número indicando el orden en que se realiza esa comunicación, y este número está en la parte superior en las flechas horizontales y a la izquierda en las flechas verticales. Las flechas de procesadores en la antidiagonal principal tienen también asociados uno o dos números que indican qué parámetros intervienen en la correspondiente comunicación, y estos números están debajo de la flecha en flechas horizontales y a la derecha en flechas verticales.

En la actualización de la matriz, cada procesador determina los parámetros que debe usar para actualizar pre y postmultiplicando cada uno de los bloques 2×2 que contiene.

Algoritmo 6.2.4 *Actualizar matriz (i,j).*

SI $i = r - 1 - j$

(*Actualizar A_{ii}*)

PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$

PARA $q = m, m + 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} c_{\frac{m}{2}+i\frac{n}{4r}} & s_{\frac{m}{2}+i\frac{n}{4r}} \\ -s_{\frac{m}{2}+i\frac{n}{4r}} & c_{\frac{m}{2}+i\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} \begin{bmatrix} c_{\frac{q}{2}+i\frac{n}{4r}} & -s_{\frac{q}{2}+i\frac{n}{4r}} \\ s_{\frac{q}{2}+i\frac{n}{4r}} & c_{\frac{q}{2}+i\frac{n}{4r}} \end{bmatrix}$$

FINPARA

FINPARA

(*Actualizar $A_{i,2r-1-i}$*)

PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$

PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} c_{\frac{m}{2}+i\frac{n}{4r}} & s_{\frac{m}{2}+i\frac{n}{4r}} \\ -s_{\frac{m}{2}+i\frac{n}{4r}} & c_{\frac{m}{2}+i\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

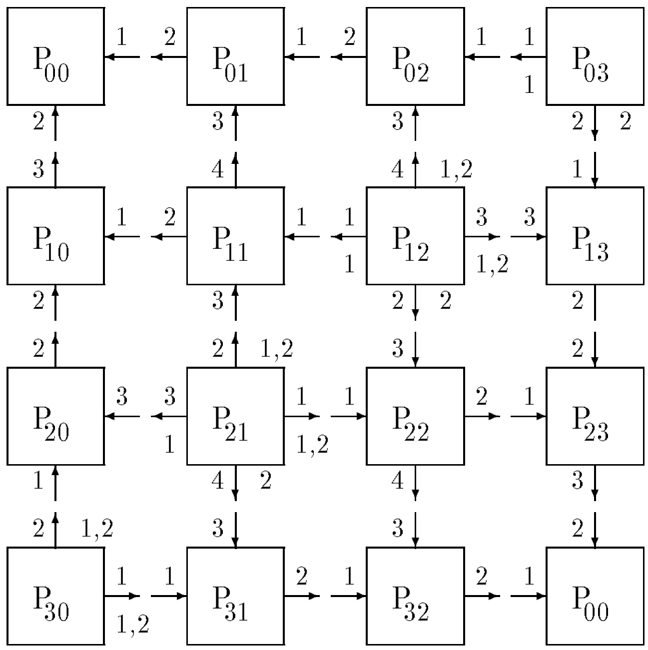


Figura 6.2.3: Comunicación de rotaciones.

$$\begin{bmatrix} c_{\frac{q}{2}+(2r-1-i)\frac{n}{4r}} & -s_{\frac{q}{2}+(2r-1-i)\frac{n}{4r}} \\ s_{\frac{q}{2}+(2r-1-i)\frac{n}{4r}} & c_{\frac{q}{2}+(2r-1-i)\frac{n}{4r}} \end{bmatrix}$$

FINPARA
FINPARA
(*Actualizar $A_{i+r,i+r}$*)
PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$
PARA $q = m, m+2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} c_{\frac{m}{2}+(i+r)\frac{n}{4r}} & s_{\frac{m}{2}+(i+r)\frac{n}{4r}} \\ -s_{\frac{m}{2}+(i+r)\frac{n}{4r}} & c_{\frac{m}{2}+(i+r)\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} c_{\frac{q}{2}+(i+r)\frac{n}{4r}} & -s_{\frac{q}{2}+(i+r)\frac{n}{4r}} \\ s_{\frac{q}{2}+(i+r)\frac{n}{4r}} & c_{\frac{q}{2}+(i+r)\frac{n}{4r}} \end{bmatrix}$$

FINPARA
FINPARA
EN OTRO CASO SI $i + j < r - 1$
(*Actualizar $A_{i,r-1-j}$*)
PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$
PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} c_{\frac{m}{2}+i\frac{n}{4r}} & s_{\frac{m}{2}+i\frac{n}{4r}} \\ -s_{\frac{m}{2}+i\frac{n}{4r}} & c_{\frac{m}{2}+i\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} c_{\frac{q}{2}+(r-1-j)\frac{n}{4r}} & -s_{\frac{q}{2}+(r-1-j)\frac{n}{4r}} \\ s_{\frac{q}{2}+(r-1-j)\frac{n}{4r}} & c_{\frac{q}{2}+(r-1-j)\frac{n}{4r}} \end{bmatrix}$$

FINPARA
FINPARA
(*Actualizar $A_{i,j+r}$*)
PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$
PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} c_{\frac{m}{2}+i\frac{n}{4r}} & s_{\frac{m}{2}+i\frac{n}{4r}} \\ -s_{\frac{m}{2}+i\frac{n}{4r}} & c_{\frac{m}{2}+i\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} c_{\frac{q}{2}+(r+j)\frac{n}{4r}} & -s_{\frac{q}{2}+(r+j)\frac{n}{4r}} \\ s_{\frac{q}{2}+(r+j)\frac{n}{4r}} & c_{\frac{q}{2}+(r+j)\frac{n}{4r}} \end{bmatrix}$$

FINPARA
FINPARA
EN OTRO CASO
(*Actualizar $A_{i,j+r}$*)
PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$
PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} c_{\frac{m}{2}+i\frac{n}{4r}} & s_{\frac{m}{2}+i\frac{n}{4r}} \\ -s_{\frac{m}{2}+i\frac{n}{4r}} & c_{\frac{m}{2}+i\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} c_{\frac{q}{2}+(r+j)\frac{n}{4r}} & -s_{\frac{q}{2}+(r+j)\frac{n}{4r}} \\ s_{\frac{q}{2}+(r+j)\frac{n}{4r}} & c_{\frac{q}{2}+(r+j)\frac{n}{4r}} \end{bmatrix}$$

FINPARA
FINPARA
(*Actualizar $A_{2r-1-i,j+r}$*)
PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$
PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} c_{\frac{m}{2}+(2r-1-i)\frac{n}{4r}} & s_{\frac{m}{2}+(2r-1-i)\frac{n}{4r}} \\ -s_{\frac{m}{2}+(2r-1-i)\frac{n}{4r}} & c_{\frac{m}{2}+(2r-1-i)\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} c_{\frac{q}{2}+(r+j)\frac{n}{4r}} & -s_{\frac{q}{2}+(r+j)\frac{n}{4r}} \\ s_{\frac{q}{2}+(r+j)\frac{n}{4r}} & c_{\frac{q}{2}+(r+j)\frac{n}{4r}} \end{bmatrix}$$

FINPARA
FINPARA

Explicaremos los procedimientos de transferencia de filas y columnas con un ejemplo con $n = 24$ y $r = 3$. No damos el código porque es similar al del algoritmo que no explota la simetría.

Después de la transferencia de columnas y filas la parte triangular superior de la matriz debe quedar adecuadamente modificada. Veremos cómo se modifican los datos de la parte triangular superior en la transferencia de filas, y de esto deduciremos qué datos deben moverse en la transferencia de columnas para actualizar datos en la parte triangular superior de la matriz y aquellos que intervienen en el movimiento de datos en la transferencia de filas

La figura 6.2.4 muestra la transferencia de filas. Con I, S, N, E y O marcamos los datos que se transfieren. Una I indica una transferencia interna, y S, N, E y O indican datos que deben ser enviados al sur, norte, este y oeste, respectivamente. Una marca de X indica un elemento que no se transfiere pero que debe quedar actualizado tras la transferencia de datos. Las flechas pequeñas corresponden a movimientos internos y las grandes a movimiento de datos entre procesadores. Los cuadros corresponden a bloques de la matriz, y los cuadros dibujados en grueso corresponden a bloques que están almacenados en la malla. Para obtener en qué procesador se almacenan los otros bloques hay que plegar esas partes de la matriz (figura 6.2.2).

Es necesario también almacenar algunos bloques 2×2 de la parte triangular inferior de A. Estos bloques aparecen en la figura con una letra, y el procesador donde se almacenan es el mismo donde está almacenado su bloque simétrico.

La figura 6.2.5 muestra de una manera similar la transferencia de columnas.

6.2.3 Costes teóricos

El coste aritmético del cálculo de las rotaciones es igual al del algoritmo que no explota la simetría (JaEsPaNsMaCuRr). El coste de la actualización de la matriz es diferente en este caso porque cada procesador que no está en la antidiagonal principal actualiza $\frac{n^2}{8p}$ bloques 2×2, y cada procesador en la antidiagonal principal actualiza $\frac{n^2}{8p} + \frac{n}{4\sqrt{p}}$ bloques 2×2, obteniéndose de este modo un coste de $\frac{3n^2}{p} + \frac{6n}{\sqrt{p}}$ flops. Además, hay que sumar un coste de $\frac{n^2}{p} + 2\sqrt{p} - 3$ flops al final de cada barrido debido al cálculo de $off(A)$. De este modo, el coste aritmético por barrido se puede expresar:

$$T_a = \frac{3n^3}{p} + \left(\frac{23}{2\sqrt{p}} - \frac{2}{p}\right) n^2 \quad flops \tag{6.2.1}$$

En cada paso, la difusión de los parámetros tiene un coste de $\sqrt{p}\beta + n\tau$, la transferencia de columnas tiene un coste $6\beta + \left(\frac{4n}{\sqrt{p}} + 4\right)\tau$, y la transferencia de filas un coste $6\beta + \left(\frac{4n}{\sqrt{p}} + 2\right)\tau$. Al final de cada barrido hay un coste de $2\left(\sqrt{p} - 1\right)(\beta + \tau)$ debido al cálculo de $off(A)$.

De este modo, el coste de las comunicaciones por barrido se puede aproximar por:

$$T_c = (\sqrt{p} + 12)\, n\beta + \left(1 + \frac{8}{\sqrt{p}}\right) n^2\tau \tag{6.2.2}$$

En este caso se obtiene como cota superior de la eficiencia:

$$E_p = \frac{T_1}{p(T_a + T_c)} \leq \frac{T_1}{pT_a} \leq 1 \quad . \tag{6.2.3}$$

6.2.4 Resultados experimentales

Hemos implementado el algoritmo en el Multiprocesador de Memoria Distribuida basado en transputers PARSYS SN-1040.

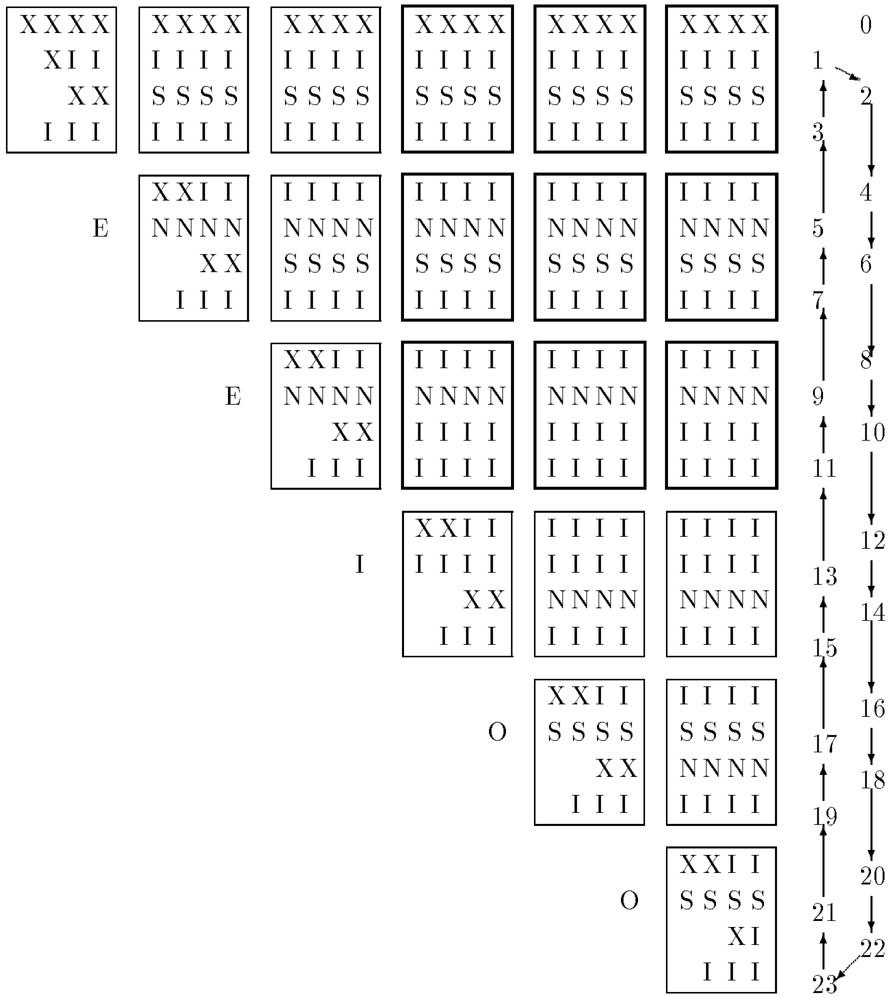


Figura 6.2.4: Movimiento de datos en la transferencia de filas.

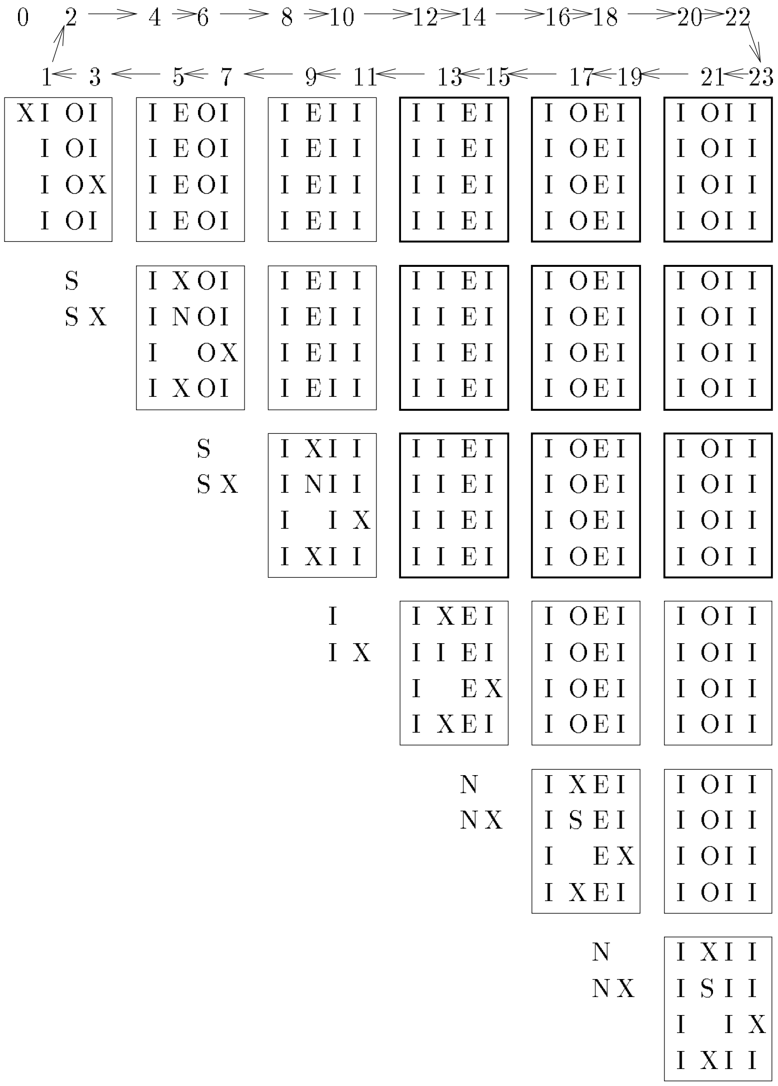


Figura 6.2.5: Movimiento de datos en la transferencia de columnas.

Comparamos el programa que explota la simetría (JaEsPaSiMaPlRr) con el que no la explota (JaEsPaNsMaCuRr). Los resultados que se muestran corresponden a tiempos por barrido, y han sido obtenidos con matrices con valores de doble precisión generados aleatoriamente entre -10 y 10, y con programas hechos en C. En la trabla 6.2.1 se muestran los tiempos obtenidos con JaEsPaSiMaPlRr y JaEsPaNsMaCuRr y con el algoritmo secuencial que explota la simetría, y los cocientes entre los tiempos de JaEsPaSiMaPlRr y JaEsPaNsMaCuRr. Este cociente debería ser 0.5 ya que con JaEsPaSiMaPlRr se trabaja aproximadamente con la mitad de la matriz, pero vemos que el valor que se obtiene cuando aumenta el tamaño de la matriz tiende a estar alrededor de 0.52. Al igual que con los algoritmos para anillo esto se debe a dos razones:

1. JaEsPaSiMaPlRr tiene un mayor overhead debido a una distribución más irregular de los datos en los procesadores.

2. La transferencia de filas y columnas es más costosa en JaEsPaSiMaPlRr.

	96	192	288	384
JaEsPaNsMaCuRr 4	6.97	52.12	172.54	405.04
JaEsPaSiMaPlRr 4	4.28	29.04	92.63	309.30
Cociente 4	0.619	0.557	0.533	0.517
JaEsPaNsMaCuRr 9	3.49	24.81	80.40	186.79
JaEsPaSiMaPlRr 9	2.28	14.01	42.88	96.66
Cociente 9	0.653	0.565	0.537	0.518
JaEsPaNsMaCuRr16	2.15	14.62	46.69	107.59
JaEsPaSiMaPlRr 16	1.53	8.69	25.89	57.43
Cociente 16	0.712	0.594	0.555	0.534
JaEsSeSiFi	10.82	84.89	285.48	

Tabla 6.2.1: Tiempos de ejecución por barrido (en segundos) de los algoritmos JaEsPaSiMaPlRr y JaEsPaNsMaCuRr. En PARSYS SN-1040.

En la tabla 6.2.2 se muestra la eficiencia obtenida, y en la 6.2.3 se comparan los tiempos obtenidos con el algoritmo para anillo que no explota la simetría ejecutándolo en anillo (JaEsPaNsAnCoPi) y en hipercubo (JeEsPaNsHiCoPi), el que no explota la simetría en malla (JaEsPaNsMaCuRr), y los algoritmos que explotan la simetría en anillo con el esquema de almacenamiento por **antidiagonales** cuando se ejecuta en anillo (JaEsPaSiAnAnPi) y en hipercubo (JaEsPaSiHiAnPi) y el que usa el esquema de almacenamiento por **plegamiento** en una malla de procesadores (JaEsPaSiMaPlRr).

Se observa que en anillo e hipercubo se obtienen resultados similares (algo mejores en hipercubo cuando aumenta el tamaño del sistema, pero la diferencia tiende a reducirse cuando aumenta el tamaño de la matriz) ya que el esquema de almacenamiento es el mismo y la única diferencia está en el broadcast de los parámetros. Sin embargo, el algoritmo para malla es claramente mejor al aumentar el número de procesadores porque el coste de las comunicaciones es menor en este algoritmo y además permite un mayor solapamiento de las operaciones, de este modo, en malla se obtiene un algoritmo más escalable, como ya vimos en el caso de los algoritmos que no explotan la simetría.

	96	192	288
JaEsPaNsMaCuRr 4	0.391	0.407	0.414
JaEsPaSiMaPlRr 4	0.632	0.731	0.770
JaEsPaNsMaCuRr 9	0.344	0.380	0.395
JaEsPaSiMaPlRr 9	0.526	0.673	0.740
JaEsPaNsMaCuRr 16	0.314	0.363	0.382
JaEsPaSiMaPlRr 16	0.442	0.610	0.689

Tabla 6.2.2: Eficiencia de los algoritmos JaEsPaSiMaPlRr y JaEsPaNsMaCuRr. En PARSYS SN-1040.

Estos resultados se confirman con los obtenidos en iPSC/860 y el Touchstone DELTA. En la tabla 6.2.4 se muestra el tiempo de ejecución por barrido obtenido en iPSC/860 con JaEsPaNsMaCuRr y JaEsPaSiMaPlRr, el algoritmo secuencial que explota la simetría y el cociente JaEsPaSiMaPlRr/JaEsPaNsMaCuRr. En la tabla 6.2.6 se muestran los mismos resultados en el Touschtone DELTA. En las tablas 6.2.5 y 6.2.7 se muestran las eficiencias obtenidas en ambas máquinas. Todos los resultados se han obtenido utilizando BLAS 1 en la actualización de la matriz y los movimientos de datos. Se observa que estos resultados confirman los obtenidos en el PARSYS SN-1040 en cuanto a la reducción en el tiempo de ejecución usando el esquema de almacenamiento por **plegamiento** y a la escalabilidad que presenta este esquema. Además, se puede ver que se obtienen mejores resultados en el Touchstone DELTA debido a que, aunque en ambas máquinas los nodos son procesadores i860, las comunicaciones son mejores en el Touchstone DELTA.

Por último, en la tabla 6.2.8 mostramos los cocientes obtenidos en el Touchstone DELTA entre JaEsPaSiMaPlRr y JaEsPaNsMaCuRr con diferentes tamaños de la malla y con el máximo tamaño de matriz con que se ha experimentado. Aparece también el tamaño de la matriz usada, este tamaño es similar en todos los casos (entre 1024 y 1232) pero el cociente aumenta considerablemente al aumentar el número de procesadores, lo que es normal teniendo en cuenta el mayor overhead, el mayor coste de las comunicaciones y el peor uso de BLAS 1 del algoritmo que explota la simetría,

	64	128	192	256	320
JaEsSeSiFi	3.48	27.13	90.61	214.59	417.81
JaEsPaNsAnCoPi 4	2.08	15.44	51.13	119.16	231.04
JaEsPaNsHiCoPi 4	2.08	15.52	51.13	119.85	232.39
JaEsPaNsMaCuRr 4	2.18	15.91	52.12	121.75	235.75
JaEsPaSiAnAnPi 4	1.54	8.30	30.64	69.72	132.82
JaEsPaSiHiAnPi 4	1.49	9.70	30.41	69.39	132.35
JaEsPaSiMaPlRr 4	1.48	9.36	29.04	65.98	125.63
JaEsPaNsAnCoPi 16	0.72	4.55	14.18	32.29	61.59
JaEsPaNsHiCoPi 16	0.70	4.52	14.14	32.25	61.52
JaEsPaNsMaCuRr 16	0.75	4.70	14.62	33.26	63.33
JaEsPaSiAnAnPi 16	0.63	3.59	10.00	21.40	39.24
JaEsPaSiHiAnPi 16	0.61	3.26	9.40	20.52	38.04
JaEsPaSiMaPlRr 16	0.60	3.07	8.69	18.79	34.67

Tabla 6.2.3: Tiempos de ejecución por barrido (en segundos) de los algoritmos JaEsPaNsAnCoPi, JaEsPaNsHiCoPi, JaEsPaNsMaCuRr, JaEsPaSiAnAnPi, JaEsPaSiHiAnPi y JaEsPaSiMaPlRr. En PARSYS SN-1040.

	64	128	192	256	320	384	448	512
JaEsPaNsMaCuRr 4	0.795	1.534	3.553	7.451	13.825	23.167	36.978	54.963
JaEsPaSiMaPlRr 4	0.650	1.211	2.395	4.779	7.797	13.012	18.980	28.684
Cociente 4	0.82	0.79	0.67	0.64	0.56	0.56	0.51	0.52
JaEsPaNsMaCuRr 16	0.796	1.012	1.822	3.293	5.027	8.031	11.693	16.413
JaEsPaSiMaPlRr 16	0.648	1.136	1.769	2.778	3.955	6.001	8.028	11.235
Cociente 16	0.81	1.12	0.97	0.84	0.79	0.75	0.69	0.68
JaEsSeSiFi	0.390	1.141	3.807	9.361	20.044	35.354	60.237	88.195

Tabla 6.2.4: Tiempos de ejecución por barrido (en segundos) de los algoritmos JaEsPaNsMaCuRr y JaEsPaSiMaPlRr. En iPSC/860.

	64	128	192	256	320	384	448	512
JaEsPaNsMaCuRr 4	0.12	0.19	0.27	0.31	0.36	0.38	0.41	0.40
JaEsPaSiMaPlRr 4	0.15	0.24	0.40	0.49	0.64	0.68	0.79	0.77
JaEsPaNsMaCuRr 16	0.03	0.07	0.13	0.18	0.25	0.28	0.32	0.34
JaEsPaSiMaPlRr 16	0.03	0.06	0.13	0.21	0.32	0.37	0.47	0.49

Tabla 6.2.5: Eficiencia de los algoritmos JaEsPaNsMaCuRr y JaEsPaSiMaPlRr. En iPSC/860.

	64	128	192	256	320	384	448	512
JaEsPaNsMaCuRr 4	0.151	1.008	3.012	6.831	13.074	22.769	37.221	56.716
JaEsPaSiMaPlRr 4	0.172	0.703	1.807	4.042	6.949	12.014	17.767	27.384
Cociente 4	1.14	0.70	0.60	0.59	0.53	0.53	0.48	0.48
JaEsPaNsMaCuRr 16	0.120	0.387	1.018	2.204	3.738	6.230	9.282	13.853
JaEsPaSiMaPlRr 16	0.175	0.463	0.938	1.735	2.751	4.219	6.031	8.979
Cociente 16	1.46	1.20	0.92	0.79	0.74	0.68	0.65	0.65
JaEsSeSiFi	0.166	1.146	3.835	9.259	20.065	35.367	60.290	88.549

Tabla 6.2.6: Tiempos de ejecución por barrido (en segundos) de los algoritmos JaEsPaNsMaCuRr y JaEsPaSiMaPlRr. En Touchstone DELTA.

	64	128	192	256	320	384	448	512
JaEsPaNsMaCuRr 4	0.27	0.28	0.32	0.34	0.38	0.39	0.40	0.39
JaEsPaSiMaPlRr 4	0.24	0.41	0.53	0.57	0.72	0.74	0.85	0.81
JaEsPaNsMaCuRr 16	0.08	0.19	0.24	0.26	0.34	0.35	0.41	0.40
JaEsPaSiMaPlRr 16	0.05	0.15	0.26	0.33	0.46	0.52	0.62	0.62

Tabla 6.2.7: Eficiencia de los algoritmos JaEsPaNsMaCuRr y JaEsPaSiMaPlRr. En Touchstone DELTA.

acentuándose todos estos factores al aumentar el tamaño del sistema. Vemos que con pocos procesadores el cociente es menor del 0.5 teórico, y esto puede ser debido a un ahorro de memoria en el algoritmo simétrico que puede producir una reducción en el tiempo de ejecución.

procesadores	4	9	16	25	36
JaEsPaSiMaPlRr/JaEsPaNsMaCuRr	0.45	0.44	0.45	0.58	0.58
tamaño matriz	1086	1104	1088	1068	1152
procesadores	49	64	81	100	121
JaEsPaSiMaPlRr/JaEsPaNsMaCuRr	0.57	0.62	0.63	0.63	0.65
tamaño matriz	1232	1152	1152	1120	1232
procesadores	144	169	196	225	256
JaEsPaSiMaPlRr/JaEsPaNsMaCuRr	0.65	0.74	0.74	0.72	0.77
tamaño matriz	1152	1040	1120	1200	1024

Tabla 6.2.8: Cociente JaEsPaSiMaPlRr/JaEsPaNsMaCuRr con diferentes tamaños de la malla. En Touchstone DELTA.

6.3 Un algoritmo por bloques

En esta sección describiremos cómo diseñar un algoritmo de Jacobi eficiente en una malla de procesadores, combinando la idea de la sección anterior sobre cómo explotar la simetría de la matriz en una malla de procesadores utilizando un esquema de almacenamiento de los datos por **plegamiento**, y la idea del capítulo uno sobre cómo diseñar un algoritmo de Jacobi por bloques rico en multiplicaciones matriciales para poder usar BLAS 3.

6.3.1 Descripción

Al igual que hicimos en el algoritmo de Jacobi para una malla de procesadores que estudiamos en la sección anterior, la matriz A se divide en cuatro submatrices cuadradas de tamaño $\frac{n}{2} \times \frac{n}{2}$ (figura 6.3.1), y llamamos a esas cuatro submatrices A_{NO}, A_{NE}, A_{SO} y A_{SE} (indicando los subíndices la posición de las submatrices en la matriz A, representando N, S, E y O, norte, sur, este y oeste, respectivamente). La matriz es simétrica y, por tanto, si decidimos trabajar con la parte triangular inferior, es necesario trabajar sólo con la submatriz A_{SO} y la parte triangular inferior de las submatrices A_{NO} y A_{SE} (y con algunos bloques adicionales de la parte triangular superior de la matriz).

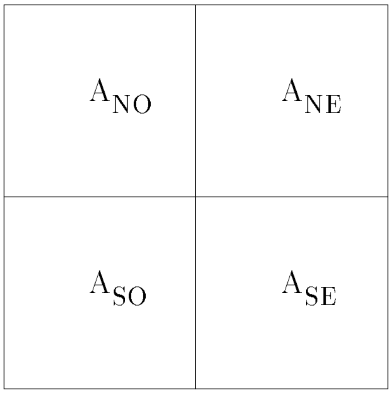


Figura 6.3.1: Partición de la matriz en submatrices.

Para obtener la distribución de los datos en los procesadores, si disponemos de una malla cuadrada con $p = r^2$ procesadores, dividimos la submatriz A_{SO} en p submatrices cuadradas de tamaño $\frac{n}{2r} \times \frac{n}{2r}$:

$$\begin{array}{cccc} A_{SO_{0,0}} & A_{SO_{0,1}} & \dots & A_{SO_{0,r-1}} \\ \dots & \dots & \dots & \dots \\ A_{SO_{r-1,0}} & A_{SO_{r-1,1}} & \dots & A_{SO_{r-1,r-1}} \end{array} \tag{6.3.1}$$

y asignamos cada matriz $A_{SO_{i,j}}$ al procesador $P_{i,j}$. Para determinar cómo se almacena la parte triangular inferior de las submatrices A_{NO} y A_{SE}, se **pliegan** estas partes triangulares inferiores sobre A_{SO} (figura 6.3.2). De este modo, si consideramos las matrices A_{NO} y A_{SE} divididas en submatrices de tamaño $\frac{n}{2r} \times \frac{n}{2r}$, $A_{NO_{i,j}}$ y $A_{SE_{i,j}}$, con $i = 0, \dots, r-1$ y $j = 0, \dots, i$, estas submatrices se asignarán a los procesadores:

a $P_{i,j}$ con $i = 0, \dots, r-1$; $j = 0, \dots, i$; la submatriz $A_{NO_{r-1-i,j}}$

a $P_{i,j}$ con $i = 0, \dots, r-1$; $j = r-1-i, \dots, r-1$; la submatriz $A_{SE_{i,r-1-j}}$

Para hacer la partición de la matriz y la asignación de los bloques a los procesadores de esta manera es necesario que $n = 2rk$, donde k es el tamaño de las submatrices, y para aplicar un esquema usando BLAS 3 tal como vimos en el capítulo 1 necesitamos que k sea múltiplo de $2t$ (donde t era el tamaño de los subbloques con los que se hacen las multiplicaciones matriciales), con lo que n debe ser múltiplo de $4rt$. Por tanto, para poder combinar este esquema de almacenamiento de los datos con el diseño de un algoritmo por bloques rico en multiplicaciones matriciales, debemos dividir cada submatriz $A_{SO_{i,j}}$, $A_{NO_{i,j}}$ y $A_{SE_{i,j}}$ en submatrices de tamaño $t \times t$ que llamaremos $A_{SO_{i,j,k,l}}$, $A_{NO_{i,j,k,l}}$ y $A_{SE_{i,j,k,l}}$, con $k = 0, \dots, m-1$, $l = 0, \dots, m-1$, donde $m = \frac{n}{2rt}$.

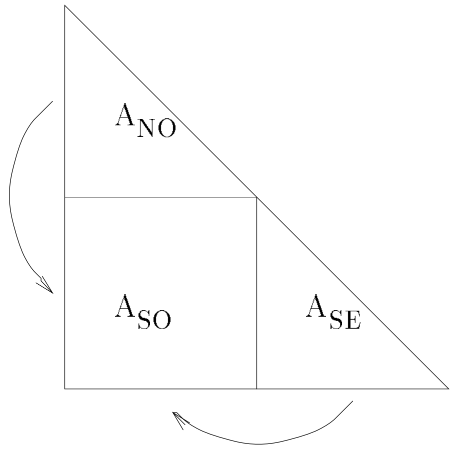


Figura 6.3.2: Almacenamiento de la matriz en el sistema de procesadores por **plegamiento**.

6.3.2 Algoritmo

En este caso el algoritmo tendría el mismo esquema del algoritmo 6.2.1.

Explicaremos brevemente cómo trabajan cada uno de los procedimientos del esquema en este caso.

Calcular rotaciones

Para el diseño del algoritmo usaremos la ordenación par-impar (figura 1.3.1) que se utilizó en el capítulo 1 para obtener el esquema de trabajo por bloques que combinamos con el esquema de distribución de los datos por plegamiento. Con este orden se tienen dos esquemas diferentes de anulación, uno en los pasos pares y otro en los impares:

impar: $(0,1),(2,3),(4,5),(6,7),\ldots$

par: $0,(1,2),(3,4),(5,6),(7,8),\ldots$

De este modo, en los pasos impares cada procesador en la antidiagonal principal calcula el mismo número de rotaciones, pero en los pasos pares uno de estos procesadores calculará una rotación menos que los demás (en el algoritmo que hemos programado el procesador distinguido será el $P_{r-1,0}$).

Mostramos con un ejemplo, con $p = 4$ y $\frac{n}{4t} = 4$, cómo se almacenan los bloques de la matriz en memoria en un paso impar (figura 6.3.3) y en un paso par (figura 6.3.4). En estas figuras aparecen bloques de memoria que contienen bloques de la matriz (marcados con una X), y otros bloques de memoria que no están marcados y que corresponden a memoria adicional que se usará sólo durante algunas partes de la ejecución del programa. También aparecen marcados los bloques de tamaño $2t \times 2t$ sobre los que se realizan los subbarridos, por lo que es necesario acumular las rotaciones correspondientes a cada uno de estos bloques para su posterior difusión.

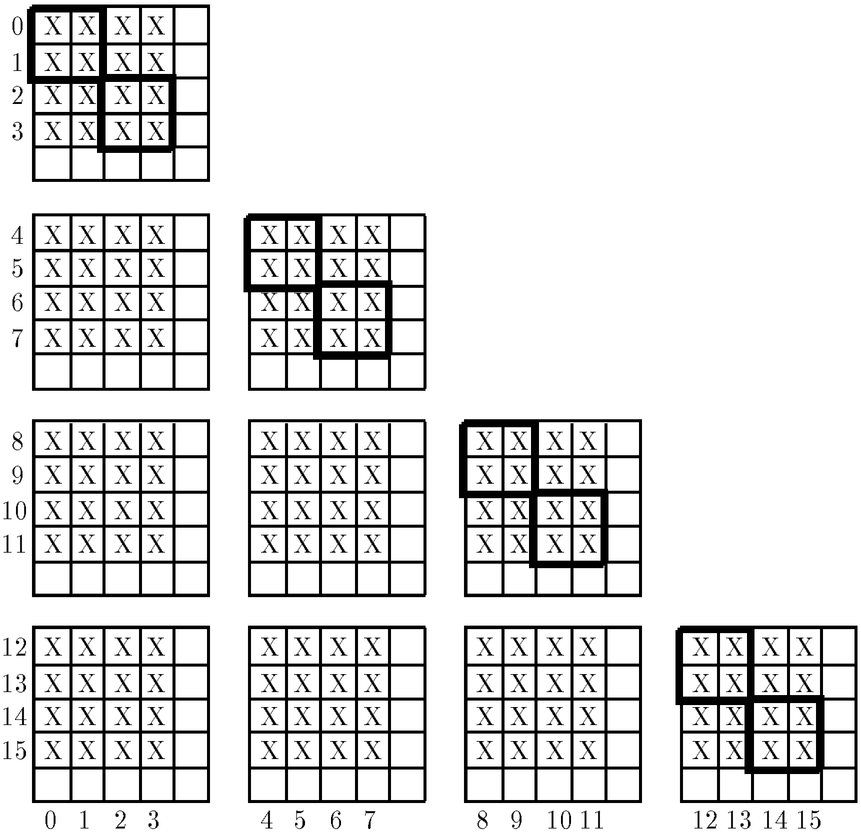


Figura 6.3.3: Cálculo de rotaciones. Paso impar

El paso de la figura 6.3.3 a la figura 6.3.4, y viceversa, se explicará en la transferencia de datos.

Difundir rotaciones

La difusión de las rotaciones se hace del mismo modo en cada paso del algoritmo.

Para actualizar un bloque $2t \times 2t$ es necesario conocer las matrices donde se han acumulado las rotaciones que modifican bloques en la diagonal principal en la misma fila o columna de bloques donde está el bloque a actualizar. De este modo, es necesario

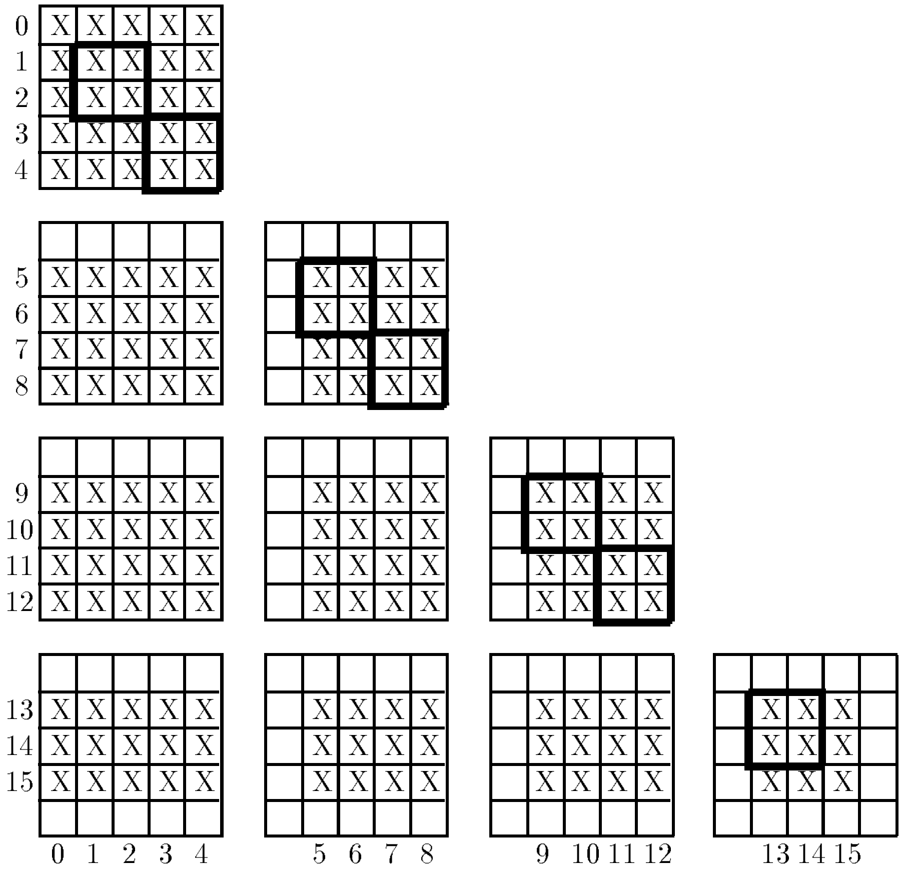


Figura 6.3.4: Cálculo de rotaciones. Paso par

transferir cada una de las matrices donde se han acumulado las rotaciones a los procesadores que contienen bloques de la misma fila o columna de bloques que el bloque actualizado por las rotaciones acumuladas en la matriz a transferir, con lo que la difusión se hará entre filas y columnas de procesadores. Pero no es necesaria la difusión a todos los procesadores de todas las matrices de rotaciones, ya que trabajamos sólo con la parte triangular inferior de la matriz, lo que hace que las matrices de rotaciones se tengan que enviar al sur y al oeste de la matriz y, en la malla, las matrices de rotaciones computadas con A_{NO} se envían al norte, sur y oeste, y las matrices computadas con A_{SE} se envían al oeste, este y sur (figura 6.3.5).

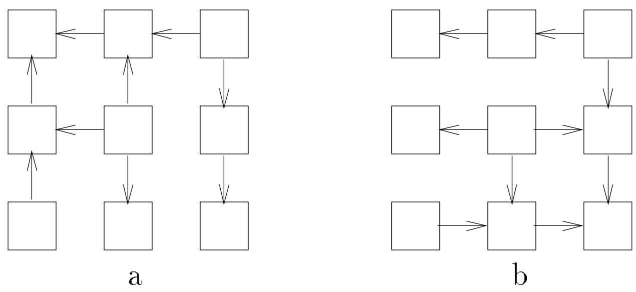


Figura 6.3.5: Difusión de rotaciones. a) rotaciones calculadas en A_{NO}. b) rotaciones calculadas en A_{SE}.

Las matrices correspondientes a A_{NO} y A_{SE} se envían juntas a los procesadores adyacentes, y son enviadas en el orden:

procesador $P_{0,r-1}$ a $P_{0,r-2}$ y $P_{1,r-1}$

procesador $P_{r-1,0}$ a $P_{r-2,0}$ y $P_{r-1,1}$

procesador $P_{i,r-1-i}$ con $i = 1, \ldots, \left\lfloor \frac{r}{2} \right\rfloor$, a $P_{i,r-2-i}$, $P_{i+1,r-1-i}$, $P_{i,r-i}$ y $P_{i-1,r-1-i}$

y procesador $P_{i,r-1-i}$ con $i = \left\lfloor \frac{r}{2} \right\rfloor + 1, \ldots, r-2$, a $P_{i,r-i}$, $P_{i-1,r-1-i}$, $P_{i,r-2-i}$ y $P_{i+1,r-1-i}$.

El orden en que se realiza la difusión de las matrices de rotaciones desde cada procesador antidiagonal se muestra en la figura 6.3.6, donde los números que aparecen al lado de las flechas indican el orden en que se realiza la transferencia correspondiente. Los procesadores que no están en la antidiagonal principal únicamente envían los matrices de rotaciones que reciben. Esta difusión se hace de manera similar a la realizada en el algoritmo de la sección anterior.

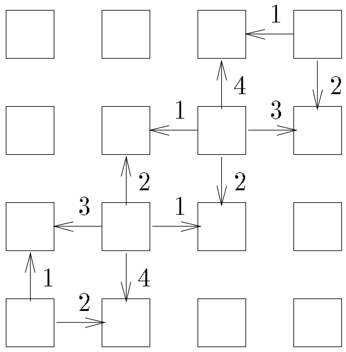


Figura 6.3.6: Orden en que se envían los parámetros de las rotaciones

Actualizar matriz

La actualización de la matriz es similar a la que se hace en el algoritmo secuencial, pero ahora las filas y columnas de bloques se encuentran distribuidas en filas y columnas de procesadores, y cada procesador actualizará (usando BLAS 3) la parte de cada fila y columna de bloques que contiene. De este modo, se obtendrá una utilización de BLAS 3 un poco peor que en el caso secuencial, al realizarse la multiplicación de matrices con matrices más pequeñas.

La actualización estará balanceada en los pasos impares, pero no estará totalmente balanceada en los pares.

Transferir datos

La transferencia de datos en los pasos impares se muestra en la figura 6.3.7. De forma similar sería la transferencia de datos en los pasos pares.

Después de un paso impar la transferencia de datos se divide en:
transferencia de columnas al este,
transferencia de columnas al oeste,
transferencia de filas al norte,
transferencia de filas al sur.
Y después de un paso par se divide en:
transferencia de filas al norte,
transferencia de filas al sur,
transferencia de columnas al este,
transferencia de columnas al oeste.

Tras la transferencia de datos entre procesadores se necesita de un movimiento interno de datos en cada procesador, usando BLAS 1 para copiar e intercambiar filas,

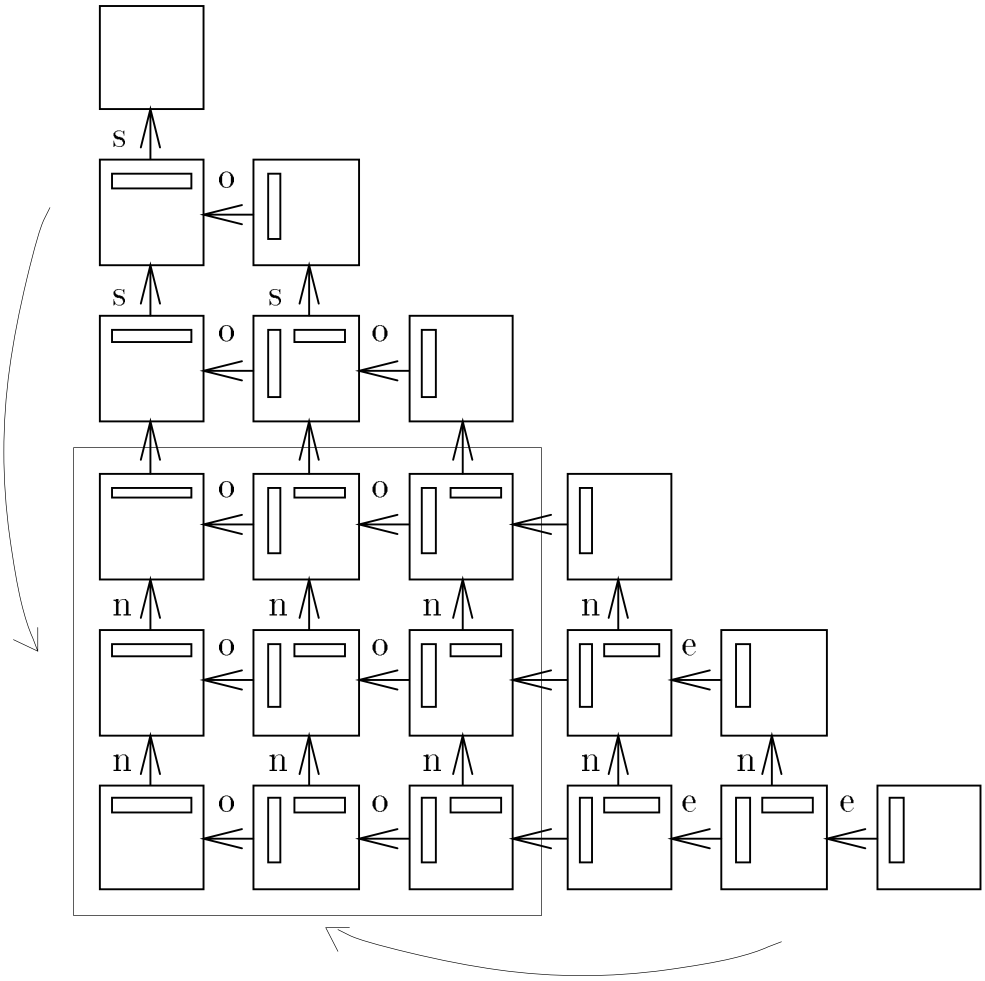


Figura 6.3.7: Transferencia de datos. Paso impar

tal como se hacía en el algoritmo secuencial.

6.3.3 Costes teóricos

En cada paso de un barrido los procesadores en la antidiagonal principal de la malla de procesadores computan un total de $\frac{n}{2t}$ ó $\frac{n}{2t}-1$ rotaciones, computando cada uno de estos procesadores $\frac{n}{2rt}=q$ ó $\frac{n}{2rt}-1=q-1$ rotaciones. De esta manera, el coste de computar rotaciones en cada paso es:

$$q48t^3=\frac{24nt^2}{r} \qquad flops \tag{6.3.2}$$

en el primer paso y:

$$q24t^3=\frac{12nt^2}{r} \qquad flops \tag{6.3.3}$$

en los demás pasos.

Con lo que, el coste por barrido de computar rotaciones es:

$$\frac{24nt^2}{r}+\left(\frac{n}{t}-1\right)\frac{12nt^2}{r}=\frac{12n^2t}{\sqrt{p}}+\frac{12nt^2}{\sqrt{p}} \qquad flops \quad . \tag{6.3.4}$$

En la difusión de las matrices de rotaciones la comunicación más costosa es la que se hace desde el procesador $P_{0,r-1}$ porque en este caso las matrices de rotaciones necesitan pasar a través de $r-1$ enlaces hasta llegar a los procesadores destino (las matrices que se envían desde $P_{r-1,0}$ también pasan por $r-1$ enlaces, pero en este caso sólo se envían la mitad de las matrices, figura 6.3.5).

Cada procesador en la antidiagonal principal computa $\frac{q}{2}$ matrices de rotaciones de tamaño $2t\times 2t$ con la matriz A_{NO} y otras tantas con la matriz A_{SE}, con lo que en cada difusión el procesador $P_{0,r-1}$ envía $4qt^2=\frac{2nt}{r}$ datos, enviando primero por el enlace 1 y después por el 2 (figura 6.3.6). Por tanto el coste de difundir rotaciones en cada paso del algoritmo en el procesador $P_{0,r-1}$ (y en todo el sistema) es:

$$\beta+\frac{2nt}{r}\tau+\left(\beta+\frac{2nt}{r}\tau\right)(r-1)=r\beta+2nt\tau \quad . \tag{6.3.5}$$

Y el coste por barrido es:

$$\frac{n}{t}\left(r\beta+2nt\tau\right)=\frac{n\sqrt{p}}{t}\beta+2n^2\tau \quad . \tag{6.3.6}$$

Para la actualización de la matriz, en los pasos impares, cada procesador no en la antidiagonal principal tiene la misma cantidad de trabajo que realizar, la actualización, pre y postmultiplicando, de $2\left(\frac{n}{2r}\frac{n}{2r}\right)=\frac{n^2}{2r^2}$ datos, lo que se hace en $\frac{4n^2t}{r^2}$ flops.

En los pasos pares, los procesadores en la columna 0 tienen además que actualizar premultiplicando $2t\frac{n}{2r}$ datos, lo que se hace en $\frac{4t^2n}{r}$ flops, necesitándose $\frac{4n^2t}{r^2}+\frac{4t^2n}{r}$ flops para actualizar la matriz en los pasos pares.

El coste por barrido de actualizar matriz será:

$$\frac{n}{2t}\frac{4n^2t}{r^2}+\frac{n}{2t}\left(\frac{4n^2t}{r^2}+\frac{4t^2n}{r}\right)=\frac{4n^3}{p}+\frac{2tn^2}{\sqrt{p}} \qquad flops \quad . \tag{6.3.7}$$

En la transferencia de bloques entre los procesadores, en los pasos impares los procesadores que realizan más trabajo envían $\frac{n}{r}t$ datos al oeste y $\frac{n}{2r}t$ datos al este, $\frac{n}{r}t$ datos al norte y $\frac{n}{2r}t$ al sur, con lo que tenemos un coste $4\beta+\frac{3nt}{r}\tau$.

Y en los pasos pares los procesadores que realizan más trabajo envían $\frac{n}{2r}t$ datos al norte, $\frac{n}{r}t$ datos al sur, $\frac{n}{r}t$ datos al este y $\frac{n}{2r}t$ datos al oeste, con lo que tenemos un coste $4\beta+\frac{3nt}{r}\tau$.

El coste de transferir datos por barrido es por tanto:

$$\frac{n}{2t}\left(8\beta+\frac{6nt}{r}\tau\right)=\frac{4n}{t}\beta+\frac{3n^2}{\sqrt{p}}\tau \quad . \tag{6.3.8}$$

El coste aritmético total por barrido sumando (6.3.4) y (6.3.7) es:

$$T_a=4k_3\frac{n^3}{p}+(2k_3+12k_1)\frac{n^2t}{\sqrt{p}}+12k_1\frac{nt^2}{\sqrt{p}} \qquad flops \tag{6.3.9}$$

donde k_3 y k_1 son las constantes que afectan a los tiempos de ejecución obtenidos utilizando BLAS 3 y BLAS 1, respectivamente.

Y el coste de las comunicaciones por barrido, sumando (6.3.6) y (6.3.8), es:

$$T_c=(4+\sqrt{p})\frac{n}{t}\beta+\left(\frac{3}{\sqrt{p}}+2\right)n^2\tau \tag{6.3.10}$$

Con lo que la eficiencia teórica del algoritmo paralelo con respecto al algoritmo secuencial que usa BLAS 3 es del 100%.

6.3.4 Resultados experimentales

Los resultados experimentales que se muestran se han obtenido en un iPSC/860 y en el Touchstone DELTA, con matrices con elementos de doble precisión generados aleatoriamente entre -10 y 10, y con programas en C. Los tiempos de ejecución que se muestran en las tablas son tiempos por barrido en segundos.

En la tabla 6.3.1 se comparan los tiempos de ejecución obtenidos en un iPSC/860 y en el Touchstone DELTA, y se observa que los tiempos son mejores en el segundo pues, aunque los procesadores son los mismos, las comunicaciones son mejores en el DELTA.

tamaño de bloque	8		16	
4 procesadores	iPSC/860	DELTA	iPSC/860	DELTA
256	2.80	2.16	2.80	2.03
512	14.37	12.82	12.39	13.36
16 procesadores	iPSC/860	DELTA	iPSC/860	DELTA
256	1.60	0.93	1.80	0.99
512	6.49	4.62	6.32	4.52
768	16.17	13.03	14.79	11.74

Tabla 6.3.1: Comparación entre iPSC/860 y Touchstone DELTA. Tiempos por barrido (en segundos).

En la tabla 6.3.2 se muestran los tiempos de ejecución obtenidos en el Touchstone DELTA para distintos tamaños de la malla y de la matriz y con tamaño de subbloque 8, 16 y 32. Se puede observar que el tamaño de bloque óptimo es prácticamente siempre 16 aunque cuando el tamaño de la matriz crece la diferencia entre los tiempos obtenidos con los tamaño de bloque 16 y 32 disminuye, tal como pasaba en el algoritmo secuencial. En el algoritmo paralelo las filas y columnas de bloques están divididas entre los procesadores, lo que hace que la pre y postmultiplicación por las matrices de rotaciones se haga con submatrices de menor tamaño que en el caso secuencial, con lo que se obtiene menos beneficio del uso del BLAS 3, y esto se nota más al aumentar el número de procesadores pues hace que las submatrices con las que se realizan las multiplicaciones sean más pequeñas.

	procesadores								
	4			16			64		
	tamaño de bloque								
	8	16	32	8	16	32	8	16	32
512	12.82	**10.83**	13.36	4.62	**4.52**	5.99	**1.98**	2.01	
1024	86.92	**66.30**	71.18	27.14	**23.53**	27.90	9.62	**9.16**	12.22
1536		**202.74**	203.00		**64.80**	72.56	26.51	**23.64**	
2048					**138.08**	147.46	55.64	**47.70**	56.89

Tabla 6.3.2: Tiempos por barrido (en segundos) obtenidos en el Touchstone DELTA.

En la tabla 6.3.3 se muestra la velocidad en Mflops por nodo obtenida con mallas de distintos tamaños, y en cada malla con las matrices de mayor tamaño con las que se ha experimentado. Se obtienen resultados similares a los obtenidos en [**?**, **?**]. Aunque este valor disminuye al aumentar el número de procesadores (debido a las comunicaciones, las sincronizaciones y al uso de BLAS 3 sobre matrices de menor tamaño) los resultados obtenidos son satisfactorios, debido a la buena escalabilidad que presenta el esquema de almacenamiento por **plegamiento** en una malla.

procesadores	1	4	9	16	25	36	64	256
Mflops/nodo	20.59	17.87	17.44	17.34	16.38	16.37	15.19	13.50

Tabla 6.3.3: Mflops por nodo obtenidos con distintos tamaños de la malla. En el Touchstone DELTA.

6.4 Conclusiones

En este capítulo se presenta un esquema de almacenamiento que permite explotar la simetría en el Problema Simétrico de Valores Propios en una malla de procesadores. Se muestra cómo combinar este esquema con la ordenación Round-Robin y con el esquema por bloques estudiado en el capítulo uno y utilizado en el capítulo tres para obtener algoritmos en multiprocesadores de memoria compartida.

Con los algoritmos que explotan la simetría en una malla de procesadores se obtienen eficiencias teóricas del 100%, tal como ocurre con los que explotan la simetría en un anillo, pero el uso de una malla nos permite obtener mejores resultados en cuanto a escalabilidad.

Los mejores tiempos de ejecución, para tamaños de matriz suficientemente grandes, se obtienen con el algoritmo por bloques que usa BLAS 3 para llevar a cabo las multiplicaciones matriciales, a pesar de que este algoritmo tiene un coste por barrido de $4n^3$ flops.

Capítulo 7

EXTENSIONES

En los capítulos anteriores hemos estudiado el Problema Simétrico de Valores Propios para matrices densas y con valores reales, y en este capítulo analizaremos algunas de las posibles extensiones de las ideas estudiadas hasta ahora sobre esquemas de almacenamiento para explotación de la simetía. En concreto, estudiaremos extensiones al Problema de Valores Propios Generalizado cuando el sistema es simétrico definido positivo, al problema estándar simétrico de valores propios con matrices complejas, y a la iteración QR en el problema no simétrico.

7.1 El Problema Generalizado

7.1.1 Introducción

El Problema de Valores Propios Generalizado

Dadas dos matrices complejas, $n \times n$, A y B, se llaman valores propios generalizados del haz (A, B) a los números complejos λ para los que existe un vector complejo no nulo x tal que $Ax = \lambda Bx$. Este vector x se llama vector propio generalizado del haz (A, B) asociado al valor propio generalizado λ.

En este caso, al igual que en el problema estándar, nos interesa el problema en que las matrices A y B son reales y simétricas. Además, consideraremos B definida positiva ($x^t Bx > 0, \forall x \neq 0$). Los haces que cumplen estas condiciones se llaman **haces simétricos definidos positivos** y para ellos existen algoritmos de Jacobi con los que se pueden calcular los valores y vectores propios generalizados [80].

Propiedades

Se pueden demostrar propiedades del Problema de Valores Propios Generalizado similares a las del problema estándar [53, 80, 90]:

Proposición 7.1.1 *Los valores propios de un haz diagonal* (A, B), $A = diag(a_i)$, $B = diag(b_i)$, *son los cocientes* $\lambda_i = \frac{a_i}{b_i}$, $i = 1, 2, ..., n$.

Proposición 7.1.2 *Si* A, B, P *y* Q *son matrices complejas,* $n \times n$, *con* P *y* Q *no singulares, entonces* λ *es un valor propio del haz* (A, B) *con vector propio* x *si y sólo si* λ *es valor propio del haz* (QAP, QBP) *con vector propio* $P^{-1}x$.

A partir de este resultado es posible calcular los valores propios de un haz (A, B) mediante su reducción, a base de transformaciones de equivalencia ($Q(A, B)P$ se denomina transformación de equivalencia), a otro haz cuyos valores propios se puedan calcular más fácilmente. En el caso de un haz simétrico definido positivo es posible transformarlo en un haz de matrices diagonales (D_A, D_B), mediante una transformación de equivalencia con $Q = P^{-1}$.

Proposición 7.1.3 λ *es un valor propio del haz* (A, B) *si y sólo si* $A - \lambda B$ *es singular.*

Proposición 7.1.4 *Un haz* (A, B) *simétrico, definido positivo, de tamaño* $n \times n$ *tiene* n *valores propios que son las raíces de su ecuación característica* $det(A - \lambda B) = 0$.

Métodos de resolución

Una forma posible de resolver el problema planteado consiste en reducir el problema generalizado a uno estándar por algún método determinado. En [80] se proponen los siguientes métodos:

1. Si la matriz B es no singular, calcular los valores propios de $B^{-1}A$. Es fácil comprobar en este caso que las ecuaciones características $det(A - \lambda B) = 0$ y $det(B^{-1}A - \lambda I) = 0$ son equivalentes. El inconveniente de este método es que la matriz B puede ser mal condicionada. También cabe destacar que en el caso simétrico, con esta aproximación se pierde la simetría.

2. Si la matriz B es simétrica y definida positiva resolver un problema estándar descomponiendo B como $B = JD^2J^t$, donde J es una matriz ortogonal y D una matriz diagonal, con lo que $A - \lambda B = JD(D^{-1}J^tAJD^{-1} - \lambda I)DJ^t$. Tomando $A' = D^{-1}J^tAJD^{-1}$ el problema generalizado se reduce al estándar $A'x' = \lambda x'$, con $x' = DJ^tx$.

3. Si la matriz B es simétrica y definida positiva obtener la descomposición de Cholesky de B, $B = LL^t$, donde L es triangular inferior no singular, con lo que $A - \lambda B = L(L^{-1}AL^{-t} - \lambda I)L^t$. Tomando $A' = L^{-1}AL^{-t}$ el problema generalizado se reduce al estándar $A'x' = \lambda x'$, con $x' = L^tx$. Este método es el que se usa en [49, 89].

Otra posibilidad (proposición 7.1.4) consiste en calcular los valores propios obteniendo los ceros del polinomio $det(A - \lambda B)$, pero este método presenta los mismos inconvenientes que en el caso estándar.

Por último, y por la proposición 7.1.2, se puede reducir el haz (A, B) a otro equivalente (D_A, D_B) de matrices diagonales $D_A = diag(a_{ii})$, $D_B = diag(b_{ii})$, siendo los cocientes $\frac{a_{ii}}{b_{ii}}$ los valores propios del haz.

Recientemente se han propuesto algunos métodos de este tipo. El método de Charlier-Van Dooren [25] procede triangularizando la matriz B y a continuación reduciendo A a forma triangular manteniendo el carácter triangular de B (en el caso de haces simétricos los reduciría a matrices diagonales). Se demuestra que el método converge bajo ciertas condiciones, pero experimentalmente se comprueba [21] que converge mucho más lentamente que otros métodos de Jacobi.

Por otra parte en [21] se propone un método que converge mucho más rápidamente que el de [25] pero no hay una demostración de su convergencia. El algoritmo de [21] sirve para haces de matrices complejas, $n \times n$, (A, B) y procede anulando en sucesivos barridos pares de elementos a_{ij}, b_{ij}, de A y B.

Métodos de Jacobi

En el método de Jacobi generalizado para haces simétricos definidos positivos, al igual que en el estándar, la diagonalización se consigue por sucesivas anulaciones de elementos no diagonales. En este caso se anulan simultáneamente dos elementos, uno de A y el correspondiente de B [80]. Cada par de productos $Q_l A_l Q_l^t$, $Q_l B_l Q_l^t$, anula pares de elementos no diagonales simétricos a_{ij} y a_{ji} de A y b_{ij} y b_{ji} de B. La matriz Q_l coincide con la identidad excepto en los elementos $q_{ij} = -\alpha$ y $q_{ji} = \beta$, con $\alpha = \frac{\delta_i}{v}$, $\beta = \frac{\delta_j}{v}$ donde v satisface la ecuación cuadrática

$$v^2 - \delta_{ij} v - \delta_i \delta_j = 0 \tag{7.1.1}$$

y

$$\delta_i = \begin{bmatrix} a_{ii} & b_{ii} \\ a_{ij} & b_{ij} \end{bmatrix} \quad , \quad \delta_j = \begin{bmatrix} a_{jj} & b_{jj} \\ a_{ij} & b_{ij} \end{bmatrix} \quad , \quad \delta_{ij} = \begin{bmatrix} a_{ii} & b_{ii} \\ a_{jj} & b_{jj} \end{bmatrix} \quad . \tag{7.1.2}$$

Cuando B es definida positiva y δ_{ij} y $\delta_i \delta_j$ no son ambos cero las soluciones de (7.1.1) son reales y hay al menos una solución no nula. Se aconseja tomar como v la solución de mayor valor absoluto para que α y β sean menores y se produzcan menores errores de redondeo.

Si $D_A = diag(a_{ii})$, $D_B = diag(b_{ii})$, los valores propios generalizados de (A, B) son los cocientes $\frac{a_{ii}}{b_{ii}}$, $i = 1, \ldots, n$, y los vectores propios generalizados son las filas del producto $Q_k Q_{k-1} \ldots Q_2 Q_1$.

Cada anulación requiere de 20 flops para el cálculo de δ_{ij}, δ_i, δ_j, v, α y β y $8n + 16$ flops para la actualización de A y B, lo que hace $8n + 36$ flops por anulación y $\frac{(8n+36)n(n-1)}{2}$ flops por barrido para el cálculo de los valores propios generalizados, lo que da un orden de $4n^3$ flops. Si se calculan los vectores propios el orden es de $6n^3$ flops, pues hay que acumular las transformaciones Q_i para obtener el producto $Q_kQ_{k-1}...Q_2Q_1$.

7.1.2 Algoritmo que no explota la simetría

Como ya hemos visto, el Problema de Valores Propios Generalizado se puede resolver en algunos casos por medio de métodos tipo Jacobi [21, 25]. En este problema los métodos de Jacobi trabajan construyendo una secuencia de pares de matrices (A_l, B_l) con $A_{l+1} = Q_lA_lP_l$, $B_{l+1} = Q_lB_lP_l$, $l = 1, 2, \ldots$, donde $A_1 = A$, $B_1 = B$, y Q_l, P_l se calculan de manera que anulen elementos no diagonales a_{ij} y b_{ij} de A y B. El método no siempre converge, pero cuando el haz (A, B) es simétrico definido positivo (A y B simétricas y B definida positiva) se puede obtener $P_l = Q_l^t$ con las fórmulas ya estudiadas (expresiones 7.1.1, 7.1.2) y la sucesión (A_l, B_l) converge a un par de matrices diagonales (D_A, D_B) siendo los valores propios generalizados los cocientes de los elementos diagonales de D_A y D_B. En este caso la matriz Q_l coincide con la identidad excepto en los elementos $q_{ij} = -\alpha$ y $q_{ji} = \beta$, con α y β dadas por las fórmulas ya comentadas.

Las mismas ideas sobre la paralelización del problema estándar sirven para el problema generalizado, por lo que en este apartado mostraremos cómo es posible aplicar al problema generalizado los mismos esquemas de distribución de los datos en un sistema multiprocesador que se aplican al problema estándar. Mostraremos esto diseñando un algoritmo para anillo usando el mismo esquema de distribución de los datos por bloques de columnas que hemos utilizado para el problema estándar, almacenando en este caso en cada procesador bloques de columnas de la matriz A y los correspondientes bloques de la matriz B.

Esquema del algoritmo

Distribución de los datos Suponemos que se dispone de un multiprocesador con memoria distribuida con p procesadores, numerados por $0, 1, \ldots, p-1$. Las filas y columnas de A y B se numeran por $0, 1, \ldots, n-1$. Además, suponemos que $n = 2bp$, donde b es un número natural. Consideramos las matrices A y B particionadas en $\frac{n}{2}$ bloques de dos columnas

$$[A_0, A_1, \ldots, A_{\frac{n}{2}-1}], \quad [B_0, B_1, \ldots, B_{\frac{n}{2}-1}] \tag{7.1.3}$$

donde cada bloque A_i (B_i) contiene inicialmente las columnas $2i$ y $2i+1$ de la matriz A (B). Los bloques A_i y B_i se almacenan en el procesador $i \; div \; b$. De este modo, cada procesador P_i contendrá b bloques consecutivos de dos columnas, numerados localmente de 0 a $b-1$, correspondientes a los bloques $bi, bi+1, \ldots, b(i+1)-1$ de la matriz A, y los correspondientes b bloques de dos columnas de B.

Un barrido consistirá en la anulación de cada uno de los $\frac{n(n-1)}{2}$ elementos no diagonales de A y B. El orden en que se anulan los elementos no diagonales dentro de cada barrido viene dado por la ordenación de Jacobi que se utilice. Para el diseño del algoritmo que aquí presentamos hemos utilizado la ordenación par-impar y la de Eberlein. Con la ordenación par-impar, tal como hemos mostrado con el problema estándar, se necesita un anillo unidireccional y n pasos por barrido, puesto que se anulan $\frac{n}{2}$ elementos en los pasos impares y $\frac{n}{2}-1$ en los pares. Sin embargo, con la ordenación de Eberlein se necesita un anillo bidireccional y $n-1$ pasos por barrido, al ser la ordenación de Eberlein óptima. Esta pequeña diferencia en el número de pasos por barrido no será importante cuando aumente el tamaño del problema, tal como veremos en los resultados experimentales.

Diseño del algoritmo Al igual que en el problema estándar, para la implementación del algoritmo hay que tener en cuenta que se realizan barridos, anulando en cada barrido los elementos no diagonales, mientras no se alcance una cota determinada de $off(A,B)$, que en este caso vendrá definido por la expresión

$$off(A,B) = \sum_{i\neq j} a_{ij}^2 + \sum_{i\neq j} b_{ij}^2 \quad . \tag{7.1.4}$$

Cada barrido constará de un número fijo de pasos (con el método par-impar n pasos y con la ordenación de Eberlein $n-1$) y, en cada paso, cada procesador tendrá que calcular los parámetros que anulan elementos asociados a los pares correspondientes a las columnas que contiene, difundir dichos parámetros a los demás procesadores, actualizar la parte de las matrices almacenadas en el procesador y transferir columnas de manera que la distribución de los índices corresponda al siguiente conjunto de pares de índices de la ordenación elegida.

El esquema del algoritmo será igual al del problema estándar (algoritmo 4.2.2), obteniéndose en este caso los parámetros con distintas fórmulas, realizándose una difusión de rotaciones igual a la del problema estándar, siendo la actualización una actualización de las matrices A y B y la transferencia de columnas una transferencia de columnas de A y de las correspondientes columnas de B.

Para calcular el $off(A,B)$ cada procesador debe calcular la parte de la expresión 7.1.4 correspondiente a las columnas que contiene.

Código del algoritmo paralelo Se supone que se quiere calcular los valores propios generalizados y que los datos están distribuidos en los procesadores tal como hemos indicado. En todos los procesadores se ejecutará el mismo código, siendo éste similar al del algoritmo 4.2.2 en el caso de usar la ordenación par-impar. Si se usa la ordenación de Eberlein el bucle $PARA$ será desde 1 hasta $n-1$.

Los procedimientos utilizados en este algoritmo tienen también códigos similares a los del problema estándar.

Algoritmo 7.1.1 *Cálculo de parámetros* (i, r).

SI $i\ mod\ 2 = 1$
  PARA $j = 0, \ldots, b-1$
    calcular parámetros correspondientes a los índices del par $(br + j)$-ésimo
  FINPARA
EN OTRO CASO
  PARA $j = 0, \ldots, b-1$
    calcular parámetros correspondientes a los índices del par $(br + j)$-ésimo,
      tomando $\alpha = 0$ y $\beta = 0$ para el par $((n - i)\ div\ 2)$-ésimo
  FINPARA
FINSI

La difusión de los parámetros tienen el mismo código del algoritmo 4.2.4 correspondiente a la difusión de los parámetros de las rotaciones en el caso estándar.

Algoritmo 7.1.2 *Actualizar matrices* (r).

PARA $m = 1, \ldots, b-1$
  PARA $q = 0, \ldots, \frac{n}{2} - 1$

$$\begin{bmatrix} a_{u0} & a_{u1} \\ a_{v0} & a_{v1} \end{bmatrix} = \begin{bmatrix} 1 & -\alpha_q \\ \beta_q & 1 \end{bmatrix} \begin{bmatrix} a_{u0} & a_{u1} \\ a_{v0} & a_{v1} \end{bmatrix} \begin{bmatrix} 1 & \beta_{br+m} \\ -\alpha_{br+m} & 1 \end{bmatrix}$$

$$\begin{bmatrix} b_{u0} & b_{u1} \\ b_{v0} & b_{v1} \end{bmatrix} = \begin{bmatrix} 1 & -\alpha_q \\ \beta_q & 1 \end{bmatrix} \begin{bmatrix} b_{u0} & b_{u1} \\ b_{v0} & b_{v1} \end{bmatrix} \begin{bmatrix} 1 & \beta_{br+m} \\ -\alpha_{br+m} & 1 \end{bmatrix}$$

  FINPARA
FINPARA

En el algoritmo 7.1.2 llamamos (u, v) al par q-ésimo y denotamos a las dos columnas de cada bloque como $a_{.0}$ y $a_{.1}$, y $b_{.0}$ y $b_{.1}$.

La transferencia de columnas tendrá el mismo código del algoritmo 4.2.6 pero enviándose en este caso datos de las dos matrices A y B.

Utilizando la ordenación de Eberlein la única diferencia estaría en los procedimientos de cálculo de parámetros y de transferencia de columnas, que serían en este caso los siguientes.

Algoritmo 7.1.3 *Cálculo de parámetros* (i, r). *(JaGePaNsAnCoEb).*

PARA $j = 0, \ldots, b-1$
  calcular parámetros correspondientes a los índices del par $(br + j)$-ésimo
FINPARA

Algoritmo 7.1.4 *Transferir columnas* (i, r). *(JaGePaNsAnCoEb).*

SI $i \ mod \ 2 = 0$
  SI $r \ mod \ 2 = 0$
    enviar columnas a $P_{(r+1) \ mod \ p}$
    recibir columnas de $P_{(r-1) \ mod \ p}$
  EN OTRO CASO
    recibir columnas de $P_{(r-1) \ mod \ p}$
    enviar columnas a $P_{(r+1) \ mod \ p}$
  FINSI
EN OTRO CASO
  SI $r \ mod \ 2 = 0$
    enviar columnas a $P_{(r-1) \ mod \ p}$
    recibir columnas de $P_{(r+1) \ mod \ p}$
  EN OTRO CASO
    recibir columnas de $P_{(r+1) \ mod \ p}$
    enviar columnas a $P_{(r-1) \ mod \ p}$
  FINSI
FINSI

El algoritmo quedaría así

Algoritmo 7.1.5 *Esquema del método de Jacobi para el problema generalizado, en un anillo de procesadores, sin explotar la simetría y usando la ordenación de Eberlein (JaGePaNsAnCoEb).*

EN PARALELO para $r = 0, 1, \ldots, p-1$ en P_r:
  REPETIR
    PARA $i = 1, \ldots, n-1$
      calcular parámetros (i, r)
      difundir parámetros (r)
      actualizar matriz (i, r)
      transferir columnas (i, r)
    FINPARA
  HASTA $off(A, B) < COTA$

Costes teóricos

En un barrido, cada procesador calcula b parámetros en cada paso con un coste de $20b$ flops. Después actualiza bn submatrices 2×2 con un coste de $16nb$ flops. Además, al final de cada barrido se tiene un coste de $\frac{2n^2}{p} - \frac{2n}{p}$ flops debido al cálculo de $off(A, B)$. Como $b = \frac{n}{2p}$, el coste total por barrido se puede aproximar por

$$T_a = \frac{8n^3}{p} + \frac{12n^2}{p} \quad flops \tag{7.1.5}$$

con el método par-impar, y

$$T_a = \frac{8n^3}{p} + \frac{4n^2}{p} \quad flops \tag{7.1.6}$$

con la ordenación de Eberlein.

Para evaluar el coste de las comunicaciones en un anillo, el broadcast de los parámetros tendrá un coste $(p-1)(2\beta + 4b\tau)$, y la transferencia de columnas un coste $2\beta + 4n\tau$. Además, se debe añadir un término de menor orden debido al cálculo de $off(A, B)$ tras cada barrido. De este modo, el coste de las comunicaciones por barrido se puede aproximar por

$$T_c = 2np\beta + 6n^2\tau \tag{7.1.7}$$

independientemente de que se use la ordenación par-impar o la de Eberlein (aunque con esta última los términos de menor orden que no aparecen en la fórmula 7.1.7 son menores que en la par-impar).

En un hipercubo (si consideramos el algoritmo sobre un anillo inmerso en el hipercubo) el coste de la difusión de los parámetros sería $2\ log\ p\beta + 4b(p-1)\tau$, y

$$T_c = 2n(log_2\ p + 1)\beta + 6n^2\tau \quad . \tag{7.1.8}$$

Como el coste aritmético del algoritmo secuencial se puede aproximar por $T_1 = 4n^3$ flops, se obtiene también en este caso la cota superior de 0.5 para la eficiencia.

Resultados experimentales

En la tabla 7.1.1 mostramos los tiempos de ejecución por barrido (en segundos) obtenidos en el PARSYS SN-1040 con los algoritmos JaGePaNsAnCoPi, JaGePaNsHiCoPi, JaGePaNsAnCoEb y JaGePaNsHiCoEb. En la tabla 7.1.2 se muestran las eficiencias obtenidas con dichos algoritmos paralelos cuando se comparan con JaGeSeSiFi. Los resultados se han obtenido con la matriz A generada aleatoriamente en doble precisión con valores entre -10 y 10, y con distintas matrices B definidas positivas tomadas de [59]. Se han obtenido resultados similares para los distintos tipos de matrices B consideradas, aunque los que se muestran en las tablas corresponden al caso en que la matriz B es tridiagonal. Se observa que los mejores resultados se obtienen con los algoritmos para hipercubo y con la ordenación de Eberlein, al ser el broadcast en hipercubo menos costoso y la ordenación de Eberlein óptima. También se observa que al aumentar el tamaño del sistema aumenta la mejora relativa al usar la ordenación de Eberlein en vez de la par-impar. Esto es así porque aumenta el coste de las comunicaciones en la transferencia de columnas, a pesar de que los costes de las comunicaciones con las dos ordenaciones son iguales si tenemos en cuenta sólo los términos de mayor orden.

	64	128	192	256	320
JaGePaNsAnCoPi 4	3.07	22.73	74.69	174.73	337.96
JaGePaNsHiCoPi 4	3.04	22.51	73.94	172.99	334.57
JaGePaNsAnCoEb 4	2.97	22.45	74.32	174.51	338.28
JaGePaNsHiCoEb 4	2.94	22.25	73.67	172.96	335.31
JaGePaNsAnCoPi 8	1.74	12.13	39.00	90.25	173.45
JaGePsNsHiCoPi 8	1.70	11.97	38.57	89.28	171.61
JaGePaNsAnCoEb 8	1.65	11.82	38.47	89.52	172.65
JaGePaNsHiCoEb 8	1.62	11.68	38.06	88.69	171.09
JaGePaNsAnCoPi 16	1.09	6.87	21.24	48.15	91.39
JaGePaNsHiCoPi 16	1.07	6.87	21.36	48.50	92.10
JaGePaNsAnCoEb 16	1.00	6.55	20.62	47.17	90.03
JaGePaNsHiCoEb 16	0.97	6.42	20.25	46.34	88.44
JaGeSeSiFi	5.27	41.75	140.16	333.21	649.75

Tabla 7.1.1: Tiempos de ejecución por barrido (en segundos) de los algoritmos JaGePaNsAnCoPi, JaGePaNsHiCoPi, JaGePaNsAnCoEb y JaGePaNsHiCoEb. En el PARSYS SN-1040.

	64	128	192	256	320
JaGePaNsAnCoPi 4	0.43	0.46	0.47	0.48	0.48
JaGePaNsHiCoPi 4	0.43	0.46	0.47	0.48	0.49
JaGePaNsAnCoEb 4	0.44	0.46	0.47	0.48	0.48
JaGePaNsHiCoEb 4	0.45	0.47	0.48	0.48	0.48
JaGePaNsAnCoPi 8	0.38	0.43	0.45	0.46	0.47
JaGePsNsHiCoPi 8	0.39	0.44	0.45	0.47	0.47
JaGePaNsAnCoEb 8	0.40	0.44	0.46	0.47	0.47
JaGePaNsHiCoEb 8	0.41	0.45	0.46	0.47	0.47
JaGePaNsAnCoPi 16	0.30	0.38	0.41	0.43	0.44
JaGePaNsHiCoPi 16	0.31	0.38	0.41	0.43	0.44
JaGePaNsAnCoEb 16	0.33	0.40	0.42	0.44	0.45
JaGePaNsHiCoEb 16	0.34	0.41	0.43	0.45	0.46

Tabla 7.1.2: Eficiencias de los algoritmos JaGePaNsAnCoPi, JaGePaNsHiCoPi, JaGePaNsAnCoEb y JaGePaNsHiCoEb. En el PARSYS SN-1040.

7.1.3 Algoritmo que explota la simetría

Bajo las mismas suposiciones (matrices de tamaño $n \times n$ con n par, y p procesadores en anillo) que hicimos al explicar el esquema de almacenamiento por **antidiagonales** para el problema estándar, las matrices A y B se dividen en submatrices de tamaño 2×2 y se distribuyen en el anillo de procesadores del mismo modo que en el caso estándar:

- Se asignan a cada procesador b antidiagonales consecutivas de A y las correspondientes de B, siendo $p = \frac{n}{2b}$, donde la antidiagonal q-ésima de A (de B) está formada por los bloques A_{ij} (B_{ij}) con $(i+j)\ mod\ \frac{n}{2} = q$.
- Si se trabaja con la parte triangular superior de las matrices será necesario almacenar los bloques A_{ij} y B_{ij} con $i = 0, \ldots, \frac{n}{2} - 1;\ j \geq i - 1$.
- A lo largo de la ejecución los parámetros para anular elementos de A y B se calcularán siempre usando datos de submatrices A_{ii} y B_{ii}, con lo que se necesitará de transferencia de filas y columnas (en este caso de A y B) entre cada dos pasos del algoritmo.

Código del algoritmo

El esquema de este algoritmo que explota la simetría es similar al que estudiamos en la subsección anterior, con la única diferencia de que en este caso la transferencia de datos se hace con una transferencia de columnas (de parte de las columnas de A y B) seguida de una transferencia de filas (de parte de las filas de A y B); además, su esquema será el del algoritmo con esquema de almacenamiento por **antidiagonales** y ordenación par-impar para el problema estándar (algoritmo 5.2.2).

El significado de los procedimientos será:

Algoritmo 7.1.6 *Cálculo de parámetros (i,r).*

SI $i\ mod\ 2 = 1$
    PARA $j = 0, \ldots, b-1$
        SI $j\ mod\ 2 = 0$
            calcular rotación correspondiente a los índices del par
                $((br+j)\ div\ 2)$-ésimo (utilizando los datos en los bloques
                $A_{(br+j)\ div\ 2,(br+j)\ div\ 2}$ y $B_{(br+j)\ div\ 2,(br+j)\ div\ 2}$).
            calcular rotación correspondiente a los índices del par
                $((br+j)\ div\ 2 + \frac{n}{4})$-ésimo (utilizando los datos en los bloques
                $A_{(br+j)\ div\ 2+\frac{n}{4},(br+j)\ div\ 2+\frac{n}{4}}$ y $B_{(br+j)\ div\ 2+\frac{n}{4},(br+j)\ div\ 2+\frac{n}{4}}$).
        FINSI
    FINPARA
EN OTRO CASO

PARA $j = 0, \ldots, b-1$
SI $j \ mod \ 2 = 0$
calcular rotación correspondiente a los índices del par
$((br+j) \ div \ 2)$-ésimo (utilizando los datos en los bloques
$A_{(br+j) \ div \ 2,(br+j) \ div \ 2}$ y $B_{(br+j) \ div \ 2,(br+j) \ div \ 2}$).
calcular rotación correspondiente a los índices del par
$((br+j) \ div \ 2 + \frac{n}{4})$-ésimo (utilizando los datos en los bloques
$A_{(br+j) \ div \ 2+\frac{n}{4},(br+j) \ div \ 2+\frac{n}{4}}$ y $B_{(br+j) \ div \ 2+\frac{n}{4},(br+j) \ div \ 2+\frac{n}{4}}$).
(*Hacer los cálculos tomando $\alpha = \beta = 0$ para el par $((n-i) \ div \ 2)$-ésimo.*)
FINSI
FINPARA
FINSI

La difusión de los parámetros es igual a la del algoritmo que no explota la simetría, ya que es un broadcast usando la topología de anillo.

Algoritmo 7.1.7 *Actualizar matriz (r).*
PARA $j = 0, \ldots, b-1$
$k = br + j$
PARA $q = 0, \ldots, k \ div \ 2$
$m = k - q$

$$\begin{bmatrix} a_{2q,2m} & a_{2q,2m+1} \\ a_{2q+1,2m} & a_{2q+1,2m+1} \end{bmatrix} = \begin{bmatrix} 1 & -\alpha_q \\ \beta_q & 1 \end{bmatrix} \begin{bmatrix} a_{2q,2m} & a_{2q,2m+1} \\ a_{2q+1,2m} & a_{2q+1,2m+1} \end{bmatrix} \begin{bmatrix} 1 & \beta_m \\ -\alpha_m & 1 \end{bmatrix}$$

$$\begin{bmatrix} b_{2q,2m} & b_{2q,2m+1} \\ b_{2q+1,2m} & b_{2q+1,2m+1} \end{bmatrix} = \begin{bmatrix} 1 & -\alpha_q \\ \beta_q & 1 \end{bmatrix} \begin{bmatrix} b_{2q,2m} & b_{2q,2m+1} \\ b_{2q+1,2m} & b_{2q+1,2m+1} \end{bmatrix} \begin{bmatrix} 1 & \beta_m \\ -\alpha_m & 1 \end{bmatrix}$$

(*$A_{qm} = Q_q A_{qm} Q_m^t$, $B_{qm} = Q_q B_{qm} Q_m^t$ *)
FINPARA
$k = br + j + 1$
PARA $q = k, \ldots, (k-1) \ div \ 2 + \frac{n}{4}$
$m = k - q - 1 + \frac{n}{2}$

$$\begin{bmatrix} a_{2q,2m} & a_{2q,2m+1} \\ a_{2q+1,2m} & a_{2q+1,2m+1} \end{bmatrix} = \begin{bmatrix} 1 & -\alpha_q \\ \beta_q & 1 \end{bmatrix} \begin{bmatrix} a_{2q,2m} & a_{2q,2m+1} \\ a_{2q+1,2m} & a_{2q+1,2m+1} \end{bmatrix} \begin{bmatrix} 1 & \beta_m \\ -\alpha_m & 1 \end{bmatrix}$$

$$\begin{bmatrix} b_{2q,2m} & b_{2q,2m+1} \\ b_{2q+1,2m} & b_{2q+1,2m+1} \end{bmatrix} = \begin{bmatrix} 1 & -\alpha_q \\ \beta_q & 1 \end{bmatrix} \begin{bmatrix} b_{2q,2m} & b_{2q,2m+1} \\ b_{2q+1,2m} & b_{2q+1,2m+1} \end{bmatrix} \begin{bmatrix} 1 & \beta_m \\ -\alpha_m & 1 \end{bmatrix}$$

(*$A_{qm} = Q_q A_{qm} Q_m^t$, $B_{qm} = Q_q B_{qm} Q_m^t$ *)
FINPARA
FINPARA

La transferencia de filas y columnas tiene el mismo código de los algoritmos 5.2.5 y 5.2.6, pero transfiriendo en este caso las filas o columnas de A tal como aparece en esos algoritmos y las correspondientes filas y columnas de B.

Costes teóricos

El coste aritmético del cálculo de los parámetros y de $off(A,B)$ es igual al del algoritmo que no explota la simetría, pero el coste de actualizar las matrices es diferente en este caso porque sólo se almacenan $\frac{n^2+2n}{4p}$ bloques de tamaño 2×2 en cada procesador. De este modo, el coste aritmético por barrido es:

$$T_a = \frac{4n^3}{p} + \frac{20n^2}{p} \qquad flops \tag{7.1.9}$$

El coste de comunicaciones en el broadcast de los parámetros y el cálculo de $off(A,B)$ es igual al del algoritmo que no explota la simetría. Pero ahora la transferencia de columnas tiene un coste de $2\beta + (2n+16)\tau$ y la transferencia de filas un coste $2\beta + 2n\tau$. Así, el coste de las comunicaciones por barrido es:

$$T_c = 2n(p+1)\beta + 6n^2\tau \tag{7.1.10}$$

Resultados experimentales

Hemos implementado el algoritmo en el Multiprocesador PARSYS SN-1040 basado en Transputers, y hemos usado topología de anillo.

Mostraremos y compararemos los resultados obtenidos con un algoritmo secuencial que explota la simetría (JaGeSeSiFi), con un algoritmo que no explota la simetía en anillo (JaGePaNsAnCoPi) y con un algoritmo que explota la simetría en anillo (JaGePaSiAnAnPi).

Se han realizado experimentos con matrices de distintos tamaños y con diferente número de procesadores. En todos los casos la matriz A es una matriz generada aleatoriamente con valores entre -10 y 10. Para obtener los resultados que se muestran aquí se ha tomado la matriz B como una matriz tridiagonal, pero hemos obtenido resultados similares con otras matrices definidas positivas tomadas de [59].

Los programas han sido escritos en 3L C y los tiempos de las tablas son tiempos por barrido en segundos.

Los resultados que se obtienen en cuanto a eficiencias y a la comparación entre los métodos que no explotan y que explotan la simetría son similares a los obtenidos en el problema estándar, obteniendose con el problema generalizado mejores resultados debido al mayor volumen de cómputo, lo que hace que se obtengan mayores eficiencias y que la reducción en el tiempo de ejecución al explotar la simetría sea mayor.

En la tabla 7.1.3 se muestran los tiempos obtenidos con el algoritmo secuencial que explota la simetría y con JaGePaNsAnCoPi y JaGePaSiAnAnPi, así como el cociente entre los tiempos de JaGePaSiAnAnPi y JaGePaNsAnCoPi, y en la tabla 7.1.4 las eficiencias obtenidas con los mismos algoritmos.

	64	128	192	256	320
JaGePaNsAnCoPi 2	6.72	51.82	172.65	406.70	791.14
JaGePaSiAnAnPi 2	4.77	32.96	105.57	243.72	467.99
Cociente 2	0.70	0.63	0.61	0.59	0.59
JaGePaNsAnCoPi 4	3.56	26.63	87.93	206.19	399.99
JaGePaSiAnAnPi 4	2.63	17.43	54.78	125.41	239.56
Cociente 4	0.73	0.65	0.62	0.60	0.59
JaGePaNsAnCoPi 8	1.98	14.07	45.59	105.96	204.43
JaGePaSiAnAnPi 8	1.58	9.61	29.35	66.10	125.03
Cociente 8	0.79	0.68	0.64	0.62	0.61
JaGePaNsAnCoPi 16	1.21	7.82	24.48	55.91	106.74
JaGePaSiAnAnPi 16	1.10	5.80	16.77	36.66	68.07
Cociente 16	0.90	0.74	0.68	0.65	0.63
JaGeSeSiFi	6.16	48.99	164.61	398.04	

Tabla 7.1.3: Tiempos de ejecución por barrido y en segundos de los algoritmos JaGePaNsAnCoPi y JaGePaSiAnAnPi. En PARSYS SN-1040.

	64	128	192	256
JaGePaNsAnCoPi 2	0.458	0.472	0.476	0.489
JaGePaSiAnAnPi 2	0.647	0.743	0.780	0.816
JaGePaNsAnCoPi 4	0.433	0.460	0.468	0.482
JaGePaSiAnAnPi 4	0.586	0.703	0.751	0.793
JaGePaNsAnCoPi 8	0.390	0.435	0.451	0.469
JaGePaSiAnAnPi 8	0.488	0.637	0.701	0.753
JaGePaNsAnCoPi 16	0.319	0.392	0.420	0.445
JaGePaSiAnAnPi 16	0.351	0.528	0.614	0.679

Tabla 7.1.4: Eficiencias de los algoritmos JaGePaNsAnCoPi y JaGePaSiAnAnPi. En PARSYS SN-1040.

7.2 Problema con valores complejos

7.2.1 Introducción

El Problema Simétrico de Valores Propios con matrices con números complejos se puede abordar con métodos tipo Jacobi [118, 119, 120], que tienen en este caso el inconveniente de que no siempre convergen.

En [119] se muestra un método de Jacobi que converge cuando la matriz A es diagonal dominante, siendo la definición de matriz diagonal dominante la siguiente.

Definición 7.2.1 *Una matriz A compleja de tamaño $n \times n$ es* ***diagonal dominante con respecto a una separación*** η *si* $\|A - diag(A)\| \leq c\eta$ *para algún* $c < \frac{1}{2}$.

El método que se utiliza en [119] utiliza la ordenación Round-Robin para elegir los elementos a anular, y actualiza la matriz A con una transformación de la forma $A_{l+1} = T^{-1}A_lT$. Si la matriz T se utiliza para anular los elementos a_{ij} y a_{ji}, los elementos de T serán los de la matriz identidad salvo t_{ii}, t_{ij}, t_{ji} y t_{jj}, que vendrán dados por la expresión

$$\begin{pmatrix} t_{ii} & t_{ij} \\ t_{ji} & t_{jj} \end{pmatrix} = \begin{pmatrix} \sqrt{\frac{1}{2} + \frac{1}{2F}} & \frac{-a_{ij}}{\nu\sqrt{\frac{1}{2}F(1+F)}} \\ \frac{a_{ij}}{\nu\sqrt{\frac{1}{2}F(1+F)}} & \sqrt{\frac{1}{2} + \frac{1}{2F}} \end{pmatrix} , \qquad (7.2.1)$$

donde $\nu = a_{ii} - a_{jj}$, y $F = \sqrt{1 + 4\frac{a_{ij}a_{ji}}{\nu^2}}$ con la parte real de F mayor que cero.

Se demuestra en [119] que este método converge cuadráticamente. En esta sección analizaremos la paralelización de este método utilizando el esquema de almacenamiento por plegamiento en una malla de procesadores propuesto en el capítulo 6.

7.2.2 Algoritmo

El esquema del algoritmo es idéntico al que hemos estudiado para el problema estándar con matrices reales (algotimo 6.2.1). Indicaremos las diferencias en cada uno de los procedimientos.

Calcular parámetros (i)

El cálculo de los parámetros se hace como en el algoritmo de cálculo de rotaciones 6.2.2, pero calculando los parámetros según la expresión 7.2.1. De esta manera el número de operaciones de números complejos a realizar en el cálculo de ν, F y de los parámetros de T y T^{-1} son 26, de las cuales tendremos: 4 sumas o restas de un número real con un complejo, 2 restas de números complejos, 2 multiplicaciones de un número real por un complejo, 5 multiplicaciones de números complejos, 1 cuadrado de un número complejo, 1 división de un número real por un complejo, 7 divisiones de números complejos, y 4 raíces cuadradas de números complejos. El coste en flops de estas operaciones depende de la implementación realizada.

Difundir parámetros (i, j)

El código del algoritmo es como el del algoritmo 6.2.3.

Actualizar matriz (i, j)

Es como el algoritmo 6.2.4 pero en este caso actualizando con la fórmula $T^{-1}AT$. Si denotamos la matriz de la expresión 7.2.1 por

$$\begin{pmatrix} p & q \\ r & s \end{pmatrix} \quad , \tag{7.2.2}$$

y su inversa como

$$\begin{pmatrix} p' & q' \\ r' & s' \end{pmatrix} \quad , \tag{7.2.3}$$

el algoritmo quedaría:

Algoritmo 7.2.1 *Actualizar matriz (i,j).*
SI $i = r - 1 - j$
(*Actualizar A_{ii}*)
PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$
PARA $q = m, m + 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} p'_{\frac{m}{2}+i\frac{n}{4r}} & q'_{\frac{m}{2}+i\frac{n}{4r}} \\ r'_{\frac{m}{2}+i\frac{n}{4r}} & s'_{\frac{m}{2}+i\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} p_{\frac{q}{2}+i\frac{n}{4r}} & q_{\frac{q}{2}+i\frac{n}{4r}} \\ r_{\frac{q}{2}+i\frac{n}{4r}} & s_{\frac{q}{2}+i\frac{n}{4r}} \end{bmatrix}$$

FINPARA
FINPARA
(*Actualizar $A_{i,2r-1-i}$*)
PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$
PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} p'_{\frac{m}{2}+i\frac{n}{4r}} & q'_{\frac{m}{2}+i\frac{n}{4r}} \\ r'_{\frac{m}{2}+i\frac{n}{4r}} & s'_{\frac{m}{2}+i\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} p_{\frac{q}{2}+(2r-1-i)\frac{n}{4r}} & q_{\frac{q}{2}+(2r-1-i)\frac{n}{4r}} \\ r_{\frac{q}{2}+(2r-1-i)\frac{n}{4r}} & s_{\frac{q}{2}+(2r-1-i)\frac{n}{4r}} \end{bmatrix}$$

FINPARA

FINPARA

(*Actualizar $A_{i+r,i+r}$*)

PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$

PARA $q = m, m+2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} p'_{\frac{m}{2}+(i+r)\frac{n}{4r}} & q'_{\frac{m}{2}+(i+r)\frac{n}{4r}} \\ r'_{\frac{m}{2}+(i+r)\frac{n}{4r}} & s'_{\frac{m}{2}+(i+r)\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} p_{\frac{q}{2}+(i+r)\frac{n}{4r}} & q_{\frac{q}{2}+(i+r)\frac{n}{4r}} \\ r_{\frac{q}{2}+(i+r)\frac{n}{4r}} & s_{\frac{q}{2}+(i+r)\frac{n}{4r}} \end{bmatrix}$$

FINPARA

FINPARA

EN OTRO CASO SI $i + j < r - 1$

(*Actualizar $A_{i,r-1-j}$*)

PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$

PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} p'_{\frac{m}{2}+i\frac{n}{4r}} & q'_{\frac{m}{2}+i\frac{n}{4r}} \\ r'_{\frac{m}{2}+i\frac{n}{4r}} & s'_{\frac{m}{2}+i\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} p_{\frac{q}{2}+(r-1-j)\frac{n}{4r}} & q_{\frac{q}{2}+(r-1-j)\frac{n}{4r}} \\ r_{\frac{q}{2}+(r-1-j)\frac{n}{4r}} & s_{\frac{q}{2}+(r-1-j)\frac{n}{4r}} \end{bmatrix}$$

FINPARA

FINPARA

(*Actualizar $A_{i,j+r}$*)

PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$

PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix} = \begin{bmatrix} p'_{\frac{m}{2}+i\frac{n}{4r}} & q'_{\frac{m}{2}+i\frac{n}{4r}} \\ r'_{\frac{m}{2}+i\frac{n}{4r}} & s'_{\frac{m}{2}+i\frac{n}{4r}} \end{bmatrix} \begin{bmatrix} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{bmatrix}$$

$$\begin{bmatrix} p_{\frac{q}{2}+(j+r)\frac{n}{4r}} & q_{\frac{q}{2}+(j+r)\frac{n}{4r}} \\ r_{\frac{q}{2}+(j+r)\frac{n}{4r}} & s_{\frac{q}{2}+(j+r)\frac{n}{4r}} \end{bmatrix}$$

FINPARA
FINPARA
EN OTRO CASO
(*Actualizar $A_{i,j+r}$*)
PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$
PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\left[\begin{array}{cc} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{array}\right] = \left[\begin{array}{cc} p'_{\frac{m}{2}+i\frac{n}{4r}} & q'_{\frac{m}{2}+i\frac{n}{4r}} \\ r'_{\frac{m}{2}+i\frac{n}{4r}} & s'_{\frac{m}{2}+i\frac{n}{4r}} \end{array}\right] \left[\begin{array}{cc} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{array}\right]$$

$$\left[\begin{array}{cc} p_{\frac{q}{2}+(j+r)\frac{n}{4r}} & q_{\frac{q}{2}+(j+r)\frac{n}{4r}} \\ r_{\frac{q}{2}+(j+r)\frac{n}{4r}} & s_{\frac{q}{2}+(j+r)\frac{n}{4r}} \end{array}\right]$$

FINPARA
FINPARA
(*Actualizar $A_{2r-1-i,j+r}$*)
PARA $m = 0, 2, 4, \ldots, \frac{n}{2r} - 2$
PARA $q = 0, 2, \ldots, \frac{n}{2r} - 2$

$$\left[\begin{array}{cc} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{array}\right] = \left[\begin{array}{cc} p'_{\frac{m}{2}+(2r-1-i)\frac{n}{4r}} & q'_{\frac{m}{2}+(2r-1-i)\frac{n}{4r}} \\ r'_{\frac{m}{2}+(2r-1-i)\frac{n}{4r}} & s'_{\frac{m}{2}+(2r-1-i)\frac{n}{4r}} \end{array}\right] \left[\begin{array}{cc} a_{mq} & a_{m,q+1} \\ a_{m+1,q} & a_{m+1,q+1} \end{array}\right]$$

$$\left[\begin{array}{cc} p_{\frac{q}{2}+(j+r)\frac{n}{4r}} & q_{\frac{q}{2}+(j+r)\frac{n}{4r}} \\ r_{\frac{q}{2}+(j+r)\frac{n}{4r}} & s_{\frac{q}{2}+(j+r)\frac{n}{4r}} \end{array}\right]$$

FINPARA
FINPARA

Transferencias de filas y de columnas

Es como en el algoritmo JaEsPaSiMaPlRr pero transfiriendo en este caso el doble de datos al estar la matriz A formada por dos matrices de reales ($A = A_1 + iA_2$).

7.2.3 Costes teóricos

El coste aritmético (en flops) depende de la implementación que se haga de las operaciones con números complejos. Para hacer los programas hemos trabajado en C y el coste de calcular los parámetros de T y T^{-1} es de 181 flops, y el de actualizar (premultiplicando y postmultiplicando) una submatriz 2×2 de A tiene un coste de 112 flops.

En cada paso, cada procesador en la antidiagonal principal calcula $\frac{n}{2r}$ grupos de parámetros (8 parámetros en cada caso) con un coste de $\frac{181n}{2r}$ flops. En la actualización de la matriz, cada procesador que no está en la antidiagonal principal actualiza $\frac{n^2}{8p}$ bloques 2×2, y cada procesador en la antidiagonal principal actualiza $\frac{n^2}{8p}+\frac{n}{4\sqrt{p}}$ bloques 2×2, obteniéndose de este modo un coste de $\frac{14n^2}{p}+\frac{28n}{\sqrt{p}}$ flops. De este modo, el coste aritmético por barrido se puede expresar:

$$T_a = \frac{14n^3}{p} + \left(\frac{209}{\sqrt{p}} - \frac{14}{p}\right) n^2 \quad flops. \tag{7.2.4}$$

En cada paso, la difusión de los parámetros tiene un coste de $\sqrt{p}\beta + n\tau$, la transferencia de columnas tiene un coste $6\beta + \left(\frac{8n}{\sqrt{p}} + 8\right)\tau$, y la transferencia de filas un coste $6\beta + \left(\frac{8n}{\sqrt{p}} + 4\right)\tau$. De este modo, el coste de las comunicaciones por barrido se puede aproximar por:

$$T_c = (\sqrt{p} + 12)\, n\beta + \left(1 + \frac{16}{\sqrt{p}}\right) n^2\tau \tag{7.2.5}$$

Un algoritmo secuencial que usara la ordenación por filas (o la misma Round-Robin) tendría un coste por barrido de

$$\frac{n\,(n-1)}{2}\,(28n + 181) \qquad flops. \tag{7.2.6}$$

Con lo que la eficiencia teórica del algoritmo paralelo propuesto sería del 100%.

7.2.4 Resultados experimentales

Los programas se han hecho en C, por lo que se han implementado las operaciones necesarias sobre complejos. Para obtener resultados experimentales se han generado matrices complejas (generando dos matrices de números reales con valores en doble precisión). Se han obtenido resultados experimentales con matrices de la forma $A = D + \alpha M$, donde α es un número real que toma los valores 0.1, 0.5, 1, 5 y 10, M es una matriz simétrica con valores complejos obtenidos generando su parte real e imaginaria aleatoriamente entre -1 y 1, y con los elementos de la diagonal nulos, y D es una matriz compleja diagonal. Los experimentos se han obtenido con matrices D de dos tipos:

- caso 1: matrices diagonales con elementos complejos obtenidos generando aleatoriamente su parte real e imaginaria entre -100 y 100.
- caso 2: matrices diagonales con elementos complejos de la forma $D = diag(1 + 1i, 2 + 2i, \ldots, n + ni)$.

De esta forma la matriz A es diagonal dominante para valores pequeños de α, y el método convergerá según [119]. En la tabla 7.2.1 se muestran los barridos necesarios para alcanzar la convergencia cuando se utiliza un algoritmo secuencial que usa la ordenación cíclica por filas (y que llamaremos JaCoSeSiFi). Cuando el valor de α y el tamaño de la matriz aumentan el método no converge, al alejarse la matriz de ser diagonal dominante, con lo que no se asegura la convergencia. Experimentos realizados con un método de Jacobi secuencial cíclico con la ordenación Round-Robin dan resultados semejantes. En [118, 119, 120] se muestran experimentos con matrices de diferentes características y diferentes ordenaciones.

	caso 1					caso 2				
α	0.1	0.5	1	5	10	0.1	0.5	1	5	10
64	4	4	5	7	no con.	5	6	6	16	no con.
128	4	5	6	10	no con.	5	6	6	no con.	no con.
192	4	5	8	no con.	no con.	5	6	7	no con.	no con.
256	4	6	14	no con.	no con.	5	6	7	no con.	no con.
320	5	6	no con.	no con.	no con.	5	6	7	no con.	no con.
384	5	8	no con.	no con.	no con.	5	6	7	no con.	no con.
448	5	9	no con.	no con.	no con.	5	6	7	no con.	no con.
512	5	no con.	no con.	no con.	no con.	5	6	7	no con.	no con.
576	5	no con.	no con.	no con.	no con.	5	6	7	no con.	no con.
640	5	no con.	no con.	no con.	no con.	5	7	7	no con.	no con.

Tabla 7.2.1: Barridos necesarios para alcanzar la convergencia con un método de Jacobi cíclico por filas, con matrices complejas simétricas.

No estamos interesados aquí en estudiar la convergencia de los métodos de Jacobi para matrices simétricas complejas, sino en estudiar el comportamiento del algoritmo propuesto que explota la simetría, por lo que mostramos en la tabla 7.2.2 los tiempos de ejecución obtenidos por barrido con el programa paralelo propuesto (que llamaremos JaCoPaSiMaPlRr) y con el algoritmo secuencial que utiliza la ordenación cíclica por filas (JaCoSeSiFi), y las eficiencias obtenidas con JaCoPaSiMaPlRr cuando se compara con JaCoSeSiFi. Los resultados que se muestran en la tabla han sido obtenidos en un iPSC/860, utilizando un procesador para JaCoSeSiFi, y mallas de 4 y 16 procesadores para la ejecución de JaCoPaSiMaPlRr. Los resultados se han obtenido generando las matrices como en el caso 1 y con valor de $\alpha = 0.1$ (en todos los demás casos experimentados los tiempos por barrido son similares a los que aquí se muestran). Mientras que los métodos que no explotan la simetría tienen una cota superior teórica de la eficiencia de un 50%, con este método de explotación de la simetría se obtiene una cota superior teórica de la eficiencia de un 100%, y en la tabla se observa que a partir de tamaños de matriz suficientemente grandes se supera la cota del 50% e incluso se llega a superar la cota del 100% para matrices muy grandes, cuando aumenta mucho el coste aritmético con respecto al de las comunicaciones y el tamaño de la matriz hace que el uso de la memoria se haga de manera más eficiente dividiendo el espacio de trabajo entre distintos procesadores. Comparando las eficiencias con las obtenidas en el caso de matrices reales (tabla 6.2.5) se observa que las eficiencias son mayores en el caso complejo, lo que se debe al mayor coste computacional del algoritmo en este caso.

	tiempos			eficiencias	
	secuencial	malla 4	malla 16	malla 4	malla 16
64	1.43	1.79	1.35	0.20	0.06
128	11.25	7.63	2.39	0.37	0.29
192	26.70	9.58	4.69	0.70	0.36
256	65.28	18.67	7.79	0.87	0.52
320	124.32	33.07	12.39	0.94	0.63
384	225.32	55.84	19.38	1.01	0.73
448	347.22	84.02	27.89	1.03	0.78
512	560.32	125.16	40.17	1.12	0.87
576	764.26	171.47	52.86	1.11	0.90
640	1080.46	237.49	69.96	1.14	0.97

Tabla 7.2.2: Tiempos de ejecución por barrido (en segundos) de JaCoSeSiFi y JaCoPaSiMaPlRr, y eficiencias comparando JaCoPaSiMaPlRr con JaCoSeSiFi. En iPSC/860.

7.3 Iteración QR

7.3.1 Introducción

Como ya dijimos en el capítulo 1, el método tradicionalmente más usado (por más rápido) en monoprocesadores para resolver problemas de valores propios ha sido el de transformar la matriz A en otra matriz en forma condensada (matrices tridiagonales o Hessenberg) para la que es más fácil resolver el problema. Se suele reducir la matriz A mediante transformaciones de Householder o de Givens a forma tridiagonal si la matriz es simétrica, o Hessenberg si es no simétrica. Posteriormente se puede diagonalizar (triangularizar, o reducir a forma de Schur, dependiendo de las características del problema) la matriz tridiagonal (o Hessenberg) por medio de algún método como la iteración QR, bisección e iteración inversa, o el algoritmo divide y vencerás de Cuppen [4, 34, 53].

Se han diseñado multitud de algoritmos para Multiprocesadores [7, 58, 86, 93, 95, 96, 97, 100], pero parece mucho menos adecuado para programación paralela que los métodos de Jacobi o que los métodos más recientes basados en la descomposición en subespacios invariantes [5, 121, 65]. En este sentido, en [58] se analiza el algoritmo QR para el problema de valores propios no simétrico y se demuestra teóricamente que no puede ser escalable. Esto hace que métodos como el de Jacobi o el basado en la descomposición en subespacios invariantes sean más prometedores para máquinas masivamente paralelas.

En esta sección mostraremos cómo uno de los esquemas que hemos usado con métodos de Jacobi para el Problema Simétrico de Valores Propios se puede utilizar también con éxito (con un éxito relativo, debido a la observación de [58]) en la iteración QR no simétrica. En particular, estudiaremos la utilización del esquema de almacenamiento por antidiagonales en la iteración QR para reducir una matriz Hessenberg superior a forma de Schur. La utilización del esquema de almacenamiento **block-Hankel wrapped** (del que el esquema por antidiagonales es un caso particular) en la iteración QR se propuso inicialmente en [95, 97].

7.3.2 La iteración QR no simétrica secuencial

Una matriz en forma de Schur tiene una forma casi triangular superior (o inferior) con bloques no nulos de tamaño 1×1 ó 2×2 en la diagonal principal, siendo los elementos en bloques de tamaño 1×1 valores propios reales, y teniendo asociados cada bloque de tamaño 2×2 dos valores propios complejos conjugados. Por esto, una forma de calcular los valores propios de una matriz no simétrica consiste en reducirla mediante transformaciones de Householder o rotaciones de Givens a forma de Hessenberg superior, y posteriormente reducirla por la iteración QR a forma de Schur.

Una matriz Hessenberg H, de tamaño $n\times n$, se puede reducir a forma de Schur mediante la iteración QR no simétrica. En [53, 99] se estudia con detenimiento esta iteración, y aquí analizaremos un paso de la iteración siguiendo [53].

Un paso de la iteración QR consiste en obtener la descomposición QR de H ($H = QR$) y sobreescribir H con RQ. La iteración QR es un proceso iterativo que sigue el esquema:

Algoritmo 7.3.1 *Iteración QR.*

$H_0 = U_0^t A U_0$
PARA $k = 1, 2, \dots$
  $H_{k-1} = Q_k R_k$
  $H_k = R_k Q_k$
FINPARA

Cada paso por el bucle PARA se llama un paso de la iteración QR, y siendo H una matriz Hessenberg superior, su descomposión QR y la multiplicación RQ se pueden hacer de una manera eficiente: actualizando $n-1$ pares de filas de H para hacer la descomposición QR, y actualizando $n-1$ pares de columnas de H para hacer la actualización $H = RQ$. De esta forma, el esquema de un paso de la iteración QR sería:

Algoritmo 7.3.2 *Paso de la iteración QR.*

```
PARA k = 1,...,n-1
   Obtener c_k y s_k parámetros de la rotación de Givens que anula
      h_{k+1,k} utilizando h_{kk}
   Actualizar las filas k-ésima y (k+1)-ésima de H con la rotación
      de parámetros c_k y s_k
FINPARA
PARA k = 1,...,n-1
   Actualizar las columnas k-ésima y (k+1)-ésima de H con la rotación
      de parámetros c_k y s_k
FINPARA
```

Este algoritmo tiene un coste de $6n^2$ flops.

El principal inconveniente para su paralelización es que mientras en el primer bucle PARA se actualizan pares de filas, en el segundo se actualizan pares de columnas, lo que hace que no sean adecuadas distribuciones de los datos en los procesadores ni por filas ni por columnas. Veremos a continuación cómo modificando ligeramente el algoritmo 7.3.2 y utilizando el esquema de almacenamiento por antidiagonales se puede obtener un algoritmo paralelo teóricamente óptimo.

7.3.3 Paralelización de un paso de la iteración QR

Esquema de almacenamiento

El esquema de almacenamiento será por antidiagonales, de manera similar al usado en la sección 5.2, pero en este caso dividiendo la matriz H en bloques de mayor tamaño.

Supongamos que disponemos de p procesadores conectados formando un anillo y que los numeraremos de 0 a $p-1$. Para obtener la distribución de los datos en los procesadores se divide la matriz H de tamaño $n \times n$, siendo n múltiplo de p, en bloques de tamaño $\frac{n}{p} \times \frac{n}{p}$

$$H = \begin{bmatrix} H_{00} & H_{01} & \dots & H_{0,p-1} \\ H_{10} & H_{11} & \dots & H_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ H_{p-1,0} & H_{p-1,1} & \dots & H_{p-1,p-1} \end{bmatrix} . \tag{7.3.1}$$

A cada procesador P_i, con $i = 0, 1, \dots, p-1$ se le asignan los bloques en la i-ésima antidiagonal (bloques H_{kl} con $(k+l)$ *mod* $p = i$). Debido a que H es Hessenberg superior los bloques H_{kl} con $k - l > 1$ son nulos, por lo que no se almacenan, y los

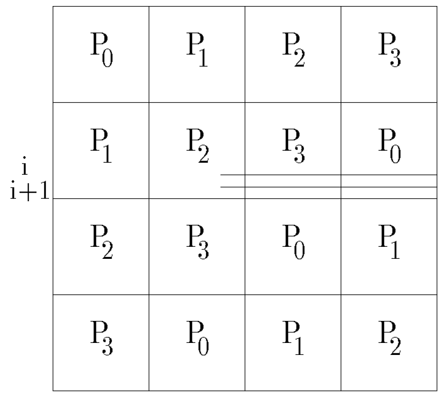


Figura 7.3.1: Actualización de un par de filas en un paso de la iteración QR.

bloques H_{kl} con $k - l = 1$ sólo tienen un elemento no nulo, por lo que no es necesario almacenar todo el bloque y reciben un tratamiento especial.

Con este esquema de almacenamiento y trabajando con el algoritmo 7.3.2, en la mayoría de los casos no estaría balanceada la actualización de los pares de filas ni de los pares de columnas, tal como se muestra en la figura 7.3.1. En esta figura se marcan los datos a actualizar (porciones de las filas i e $i+1$) y la distribución de la matriz en los procesadores suponiendo que se dispone de 4 procesadores. Se observa que el trabajo no está balanceado.

Para poder obtener una distribución del trabajo balanceada es necesario modificar ligeramente el algoritmo 7.3.2

Un paso de la iteración QR modificado

Para poder realizar las actualizaciones de filas (que nos proporcionan la descomposición QR) y las de columnas (que nos proporcionan la actualización $H = RQ$) de manera conjunta tal como se muestra en la figura 7.3.2, y que en cada paso todos los procesadores tengan el mismo trabajo, el algoritmo 7.3.2 se modifica para obtener:

Algoritmo 7.3.3 *Paso de la iteración QR.*

PARA $k = 1, 2$
  Obtener c_k y s_k parámetros de la rotación de Givens que anula
    $h_{k+1,k}$ utilizando h_{kk}
  Actualizar las filas k-ésima y $(k+1)$-ésima de H con la rotación
    de parámetros c_k y s_k
FINPARA
PARA $k = 3, \ldots, n-1$

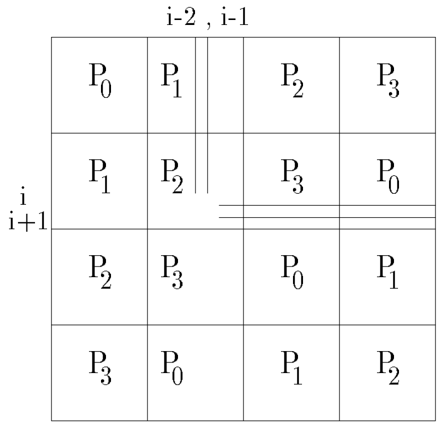


Figura 7.3.2: Actualización de un par de filas y de columnas en un paso modificado de la iteración QR.

Obtener c_k y s_k parámetros de la rotación de Givens que anula
$h_{k+1,k}$ utilizando h_{kk}
Actualizar las filas k-ésima y $(k+1)$-ésima de H con la rotación
de parámetros c_k y s_k
Actualizar las columnas $(k-2)$-ésima y $(k-1)$-ésima de H con la rotación
de parámetros c_{k-2} y s_{k-2}
FINPARA
PARA $k = n-2, n-1$
Actualizar las columnas k-ésima y $(k+1)$-ésima de H con la rotación
de parámetros c_k y s_k
FINPARA

Un paso de la iteración QR en paralelo

Utilizando el algoritmo secuencial 7.3.3 se obtiene el algoritmo paralelo:

Algoritmo 7.3.4 *Esquema de un paso de la iteración QR en paralelo.*
EN PARALELO para $r = 0, 1, \ldots, p-1$ en P_r:
PARA $k = 1, 2$
calcular rotación (k, r)
difundir rotación (k, r)
enviar filas (k, r)
actualizar filas (k, r)
devolver filas (k, r)

FINPARA
PARA $k = 3, 4, \ldots, n-1$
calcular rotación (k, r)
difundir rotación (k, r)
enviar filas (k, r)
enviar columnas $(k-2, r)$
actualizar filas (k, r)
actualizar columnas $(k-2, r)$
devolver filas (k, r)
devolver columnas $(k-2, r)$
FINPARA
PARA $k = n-2, n-1$
enviar columnas (k, r)
actualizar columnas (k, r)
devolver columnas (k, r)
FINPARA

El significado de cada uno de los procedimientos es el siguiente:

Algoritmo 7.3.5 *Calcular rotación (k,r).*
SI P_r contiene h_{kk} y $h_{k+1,k}$
calcular c_k y s_k
EN OTRO CASO SI P_r contiene h_{kk}
recibir $h_{k+1,k}$ de $P_{(r+1)\ mod\ p}$
calcular c_k y s_k
EN OTRO CASO SI P_r contiene $h_{k+1,k}$
enviar $h_{k+1,k}$ a $P_{(r-1)\ mod\ p}$
FINSI

La difusión de rotaciones es una difusión en un anillo de procesadores.

Algoritmo 7.3.6 *Enviar filas (k,r).*
SI $(k+1)\ mod\ \frac{n}{p} = 0$
SI P_r contiene elementos de la fila $k+1$
enviar elementos de la fila $k+1$ que contiene a $P_{(r-1)\ mod\ p}$
FINSI
SI P_r contiene elementos de la fila k
recibir elementos de la fila $k+1$ de $P_{(r+1)\ mod\ p}$
FINSI
FINSI

Algoritmo 7.3.7 *Actualizar filas (k,r).*

SI $(k+1)\ mod\ \frac{n}{p}=0$
actualizar la parte de la fila k que contiene y la fila
recibida en $enviarfilas$ utilizando c_k y s_k
EN OTRO CASO
actualizar la parte de las filas k y $k+1$ que contiene
utilizando c_k y s_k
FINSI

Algoritmo 7.3.8 *Devolver filas (k,r).*
SI $(k+1)\ mod\ \frac{n}{p}=0$
SI en P_r se han actualizado elementos de la fila $k+1$
enviar elementos actualizados de la fila $k+1$ a $P_{(r+1)\ mod\ p}$
FINSI
SI P_r contiene elementos de la fila $k+1$ de la parte no nula de H
recibir elementos de la fila $k+1$ actualizados por $P_{(k-1)\ mod\ p}$
FINSI
FINSI

Algoritmo 7.3.9 *Enviar columnas (k,r).*
SI $(k+1)\ mod\ \frac{n}{p}=0$
SI P_r contiene elementos de la columna $k+1$
enviar elementos de la columna $k+1$ que contiene a $P_{(r-1)\ mod\ p}$
FINSI
SI P_r contiene elementos de la columna k
recibir elementos de la columna $k+1$ de $P_{(r+1)\ mod\ p}$
FINSI
FINSI

Algoritmo 7.3.10 *Actualizar columnas (k,r).*
SI $(k+1)\ mod\ \frac{n}{p}=0$
actualizar la parte de la columna k que contiene y la columna
recibida en $enviarcolumnas$ utilizando c_k y s_k
EN OTRO CASO
actualizar la parte de las columnas k y $k+1$ que contiene
utilizando c_k y s_k
FINSI

Algoritmo 7.3.11 *Devolver columnas (k,r).*
SI $(k+1)\ mod\ \frac{n}{p}=0$
SI en P_r se han actualizado elementos de la columna $k+1$
enviar elementos actualizados de la columna $k+1$ a $P_{(r+1)\ mod\ p}$
FINSI

SI P_r contiene elementos de la columna $k+1$ de la parte no nula de H
recibir elementos de la columna $k+1$ actualizados por $P_{(k-1)\ mod\ p}$
FINSI
FINSI

Costes teóricos El coste aritmético del algoritmo 7.3.4 es aproximadamente de

$$T_a = \frac{6n^2}{p} + 6n \qquad flops, \tag{7.3.2}$$

ya que en cada uno de los pasos por el bucle cada procesador actualiza aproximadamente $\frac{2n}{p}$ elementos de H.

En cuanto al coste de las comunicaciones, tenemos $n-1$ difusiones de dos parámetros, lo que representa un coste (en un anillo) de

$$(n-1)(p-1)(\beta + 2\tau) \quad . \tag{7.3.3}$$

A lo largo de la ejecución del algoritmo se realizan $p-1$ envíos de filas, otras tantas devoluciones de filas, la misma cantidad de envíos de columnas y la misma de devoluciones de columnas, teniendo cada una de estas comunicaciones un coste de $\beta + \frac{n}{p}\tau$ a causa de que al menos un procesador envía $\frac{n}{p}$ datos. Por tanto, el coste de las comunicaciones debido a los envíos y devoluciones de filas y columnas a lo largo de todo el algoritmo es

$$4(p-1)\left(\beta + \frac{n}{p}\tau\right) \quad . \tag{7.3.4}$$

Por lo que el coste total de las comunicaciones si tenemos en cuenta sólo los términos de mayor orden será

$$T_c = p(n+3)\beta + 2n(p+1)\tau \quad . \tag{7.3.5}$$

Como el coste aritmético es de mayor orden que el de las comunicaciones la eficiencia teórica es del 100% pero, tal como veremos a continuación, los valores experimentales que se obtienen están muy lejos de ese 100% teórico, principalmente al aumentar el número de procesadores y disminuir el tamaño de la matriz. Esto es lógico teniendo en cuenta los resultados de [58], que el coste total del algoritmo es bastante pequeño ($O(\frac{n^2}{p})$) y tiene una parte ($6n$) que no disminuye al aumentar el número de procesadores, y que las comunicaciones tienen un coste alto ($O(np)$) en relación con el aritmético.

7.3.4 Resultados experimentales

Mostramos resultados experimentales obtenidos en un iPSC/860 con 32 procesadores con topología de hipercubo. Se ha usado un anillo inmerso en el hipercubo ya que el algoritmo ha sido diseñado para un anillo lógico. En las tablas 7.3.1 y 7.3.2 se muestran los tiempos de ejecución (en segundos) obtenidos en la ejecución de un paso de la iteración QR. Las matrices utilizadas son matrices Hessenberg superior con datos en doble precisión entre -10 y 10 y generados aleatoriamente. Los programas han sido hechos en C. En ambas tablas se muestran tiempos para diferentes tamaños de matriz (256, 512, 768, 1024, 1280, 1536, 1792 y 2048) y diferente número de procesadores (1, 2, 4, 8, 16 y 32). Los tiempos de la tabla 7.3.1 corresponden a un programa en el que las difusiones se realizan utilizando la topología de hipercubo, y en la tabla 7.3.2 las difusiones se realizan en anillo. Los tiempos son similares, pero se puede observar que el uso de la topología de anillo para realizar las difusiones proporciona mejores tiempos de ejecución cuando aumenta el número de procesadores. Esto puede deberse a que el uso de la topología de anillo para hacer las difusiones permite una mejor "pipelinización" en la ejecución.

	procesadores					
	1	2	4	8	16	32
256	0.073	0.075	0.076	0.089	0.101	0.104
512	0.331	0.236	0.182	0.185	0.189	0.224
768	0.716	0.507	0.337	0.303	0.288	0.318
1024		0.900	0.536	0.431	0.393	0.421
1280		1.363	0.821	0.595	0.510	0.524
1536			1.130	0.771	0.640	0.635
1792			1.560	0.999	0.779	0.748
2048				1.187	0.908	0.870

Tabla 7.3.1: Tiempos de ejecución en segundos de un paso de la iteración QR cuando la difusión se hace utilizando la topología de hipercubo. En el iPSC/860.

En la tabla 7.3.3 se muestra la eficiencia obtenida cuando se utiliza la topología de anillo para llevar a cabo las difusiones. Se comprueba que el algoritmo presenta muy mala escalabilidad (tal como se desprende del análisis de los costes teóricos y de [58]). Así, en la tabla 7.3.2 podemos comprobar que el tiempo de ejecución se reduce al pasar de 1 a 2 procesadores a partir de tamaño de matriz 512, al pasar de 2 a 4 procesadores también a partir de tamaño de matriz 512, al pasar de 4 a 8 ó de 8 a 16 procesadores a partir de tamaño de matriz 768, y al pasar de 16 a 32 procesadores a partir de tamaño 1536. Por tanto, sólo se obtienen resultados aceptables al paralelizar con un número reducido de procesadores o cuando el tamaño de la matriz es muy grande.

	procesadores					
	1	2	4	8	16	32
256	0.073	0.073	0.080	0.090	0.100	0.100
512	0.331	0.234	0.190	0.187	0.182	0.203
768	0.716	0.501	0.347	0.304	0.275	0.286
1024		0.896	0.545	0.433	0.375	0.380
1280		1.354	0.829	0.597	0.488	0.471
1536			1.139	0.773	0.615	0.572
1792			1.572	1.017	0.751	0.672
2048				1.205	0.874	0.786

Tabla 7.3.2: Tiempos de ejecución en segundos de un paso de la iteración QR cuando la difusión se hace utilizando la topología de anillo. En el iPSC/860.

	procesadores				
	2	4	8	16	32
256	0.50	0.23	0.10	0.04	0.02
512	0.71	0.44	0.22	0.11	0.05
768	0.71	0.52	0.29	0.16	0.07

Tabla 7.3.3: Eficiencias obtenidas en un paso de la iteración QR, cuando se utiliza la topología de anillo para realizar las difusiones. En el iPSC/860.

7.4 Conclusiones

En este capítulo hemos analizado la posible utilización de los esquemas de almacenamiento previamente estudiados en diversos problemas de valores propios:

- Los esquemas utilizados para explotar la simetría en métodos de Jacobi para el problema simétrico de valores propios real se pueden utilizar también para explotar la simetría cuando se utilizan métodos tipo Jacobi en el problema simétrico de valores propios generalizado y en el problema simétrico estándar cuando la matriz es compleja. Hemos analizado la utilización del esquema por antidiagonales en el problema generalizado, y del esquema por plegamiento en el problema estándar de datos complejos. Los resultados son similares a los que se obtienen en el problema estándar de datos reales, aunque se obtienen mayores eficiencias con los problemas estudiados en este capítulo debido al mayor coste computacional.

- Hemos analizado cómo un esquema de almacenamiento por antidiagonales se puede usar en la iteración QR para la reducción de una matriz Hessenberg a forma de Schur.

Capítulo 8

CONCLUSIONES Y TRABAJOS FUTUROS

En este último capítulo resumiremos las principales conclusiones que se pueden extraer del estudio realizado, así como los trabajos (technical reports, comunicaciones, publicaciones, ...) generados en relación con esta tesis, y presentaremos algunas de las posibles vías futuras de trabajo.

Las conclusiones que se presentan en la primera sección de este capítulo se han ido presentando a lo largo del trabajo y se presentan aquí reunidas y resumidas para permitir una visión general de los resultados obtenidos. Debido a la gran cantidad de tablas que se presentan a lo largo de los diferentes capítulos (que representan una pequeña parte de los experimentos realizados) se mostrarán un par de tablas para comparar los métodos de Jacobi para el Problema Simétrico de Valores Propios en Multicomputadores (tema central de esta tesis) analizados en el texto. En una de las tablas se comparan los costes teóricos de estos métodos y en otra los resultados experimentales para algunos tamaños de matriz que muestran resultados significativos. Los resultados experimentales confirman las conclusiones que se pueden obtener de la comparación de los costes teóricos.

A continuación, en la segunda sección se enumeran los trabajos generados durante la realización de la tesis. La mayoría de estos trabajos han sido citados con anterioridad a lo largo del texto pero aquí se enumeran en orden cronológico y se resumen las ideas generales que se presentan en cada uno de ellos, de manera que se puede ver la evolución del trabajo a lo largo de su realización.

Finalmente, se resumen algunas posibles vías futuras de trabajo. Debido a la gran riqueza de los Problemas de Valores Propios en general (en cuanto a diferentes problemas matemáticos, métodos de resolución y aplicaciones) y a los métodos tipo Jacobi para resolver estos problemas, se podría pensar en multitud de posibles trabajos futuros, pero aquí se resumirán sólo algunos de los que pueden tener relación más inmediata con los trabajos realizados hasta ahora. En algunos casos las propuestas de continuidad que se realizan corresponden a trabajos en curso que no se han completado todavía y en otros casos únicamente a ideas que no se han empezado a abordar pero que se tiene la intención de abordar en un futuro no muy lejano.

8.1 Conclusiones

Las conclusiones que se pueden extraer del trabajo presentado son:

- Respecto a los métodos secuenciales:

  1. El diseño de **algoritmos por bloques** ricos en operaciones matriz-matriz y la realización de estas operaciones por medio de rutinas optimizadas tipo BLAS 3 da lugar a métodos de Jacobi eficientes. Estos esquemas son portables en el sentido de que se puede usar con éxito el mismo algoritmo en distintos tipos de sistemas (monoprocesadores y multiprocesadores) siempre que se tenga una versión eficiente de BLAS para el sistema.
  2. Otra posibilidad de reducir el tiempo de ejecución en métodos de Jacobi secuenciales es por técnicas que intenten reducir el número de barridos o de anulaciones para alcanzar la convergencia, como son las técnicas de **umbralización** y los que hemos llamado métodos **semiclásicos**. En los experimentos realizados se ha comprobado que con estas técnicas, aplicándolas por separado o combinándolas de manera adecuada, se puede reducir el tiempo de ejecución de métodos de Jacobi que no trabajen por bloques. También se pueden combinar técnicas del tipo semiclásico con el trabajo por bloques aunque, debido a que las ideas son totalmente distintas al basarse un método en el trabajo por bloques y el otro en el trabajo elemento a elemento, en este caso la reducción que se obtiene en el tiempo de ejecución con respecto a un método por bloques es mínima.

- Respecto a los métodos paralelos en multiprocesadores:

  Se ha mostrado cómo se puede usar un esquema por bloques para obtener buenas prestaciones en multiprocesadores, obteniéndose los mejores resultados dividiendo el trabajo entre los procesadores y trabajando por bloques (usando BLAS 3 para realizar las multiplicaciones matriciales) en cada uno de los procesadores.

- Respecto a los métodos paralelos en multicomputadores:

  Como en multicomputadores se han estudiado diferentes métodos en diferentes capítulos (4, 5 y 6), hacemos aquí una comparación conjunta de los diferentes métodos (comparaciones más detalladas se encuentran a lo largo de todo el trabajo).

En la tabla 8.1.1 comparamos los costes teóricos de los diferentes algoritmos estudiados para multicomputadores: los dos que no explotan la simetría (JaEsPaNsAnCoPi en anillo, y JaEsPaNsMaCuRr en malla) y de los que explotan la simetría en anillo (JaEsPaSiAnAnPi con esquema de almacenamiento por antidiagonales, y JaEsPaSiAnMaEb con esquema de almacenamiento por marcos) y en malla (JaEsPaSiMaPlRr con plegamiento y ordenación Round-Robin, y JaEsPaSiMaPlPiB3 por bloques con esquema de almacenamiento por plegamiento y generando los bloques con la ordenación par-impar).

Se pueden hacer las siguientes observaciones:

- Los métodos que explotan la simetría tienen menor coste aritmético que los que no la explotan, obteniéndose una eficiencia teórica del 100% cuando se explota la simetría y del 50% cuando no se explota. Esta reducción en el coste teórico va acompañada normalmente de un aumento en términos de menor orden (n^2) en el coste aritmético, y también de un aumento en el coste de las comunicaciones aunque manteniéndose estas en el mismo orden tanto si se explota la simetría como si no. Este aumento en el coste en términos de menor orden producirá que los algoritmos que explotan la simetría no llegan en la práctica a ser dos veces más rápidos que los que no la explotan, aunque sí proporcionen una reducción notable en el tiempo de ejecución.
- Los algoritmos para malla presentan unos costes de comunicaciones menores que los algoritmos para anillo, aunque manteniendo el mismo orden. Esto produce que al aumentar el número de procesadores se note una mayor reducción en el tiempo de ejecución al usar el esquema de almacenamiento por plegamiento en una malla de procesadores.
- El algoritmo por bloques estudiado tiene un mayor coste teórico ($4\frac{n^3}{p}$) que los demás algoritmos que explotan la simetría ($3\frac{n^3}{p}$), pero este coste se obtiene con operaciones de nivel 3, con lo que en la práctica, para tamaños de matrices suficientemente grandes, los mejores resultados se obtienen con el algoritmo por bloques.

En la tabla 8.1.2 se comparan experimentalmente los diferentes métodos diseñados para multicomputadores usando BLAS (no se compara el método JaEsPaSiAnAnPi pues no se ha realizado ninguna implementación de este método usando BLAS 1, aunque los resultados que se obtienen con él son similares a los obtenidos con JaEsPaSiAnMaEb). Los resultados que se muestran han sido obtenidos en el iPSC/860. A lo largo de los diferentes capítulos se han mostrado tiempos en diferentes sistemas, pero los resultados en cuanto a eficiencia son similares en todos ellos. Se muestran tiempos de ejecución con tamaños de matriz 256 y 512 y con 4, 8 y 16 procesadores. Se pueden hacer las mismas observaciones que comparando los costes teóricos:

	coste aritmetico	coste de las comunicaciones
JaEsPaNsAnCoPi	$\frac{6n^3}{p}+\frac{13n^2}{2p}$	$2np\beta+4n^2\tau$
JaEsPaNsMaCuRr	$\frac{6n^3}{p}+\left(\frac{11}{2\sqrt{p}}-\frac{4}{p}\right)n^2$	$\left(\sqrt{p}+8\right)n\beta+\left(1+\frac{8}{\sqrt{p}}\right)n^2\tau$
JaEsPaSiAnAnPi	$\frac{3n^3}{p}+\frac{25n^2}{2p}$	$2np\beta+4n^2\tau$
JaEsPaSiAnMaEb	$\frac{3n^3}{p}+\frac{12n^2}{p}$	$2np\beta+\left(6-\frac{5}{2p}\right)n^2\tau$
JaEsPaSiMaPlRr	$\frac{3n^3}{p}+\left(\frac{23}{2\sqrt{p}}-\frac{2}{p}\right)n^2$	$\left(\sqrt{p}+12\right)n\beta+\left(1+\frac{8}{\sqrt{p}}\right)n^2\tau$
JaEsPaSiMaPlPiB3	$\frac{4k_3n^3}{p}+(2k_3+12k_1)\frac{n^2t}{\sqrt{p}}+12k_1\frac{nt^2}{\sqrt{p}}$	$\left(4+\sqrt{p}\right)\frac{n}{t}\beta+\left(\frac{3}{\sqrt{p}}+2\right)n^2\tau$

Tabla 8.1.1: Costes teóricos de los diferentes algoritmos estudiados para multicomputadores.

- Con los algoritmos que explotan la simetría se obtienen mejores tiempos de ejecución que con los que no la explotan al aumentar el tamaño de la matriz.
- Los algoritmos para malla son más escalables que los algoritmos para anillo, por lo que al aumentar el número de procesadores los algoritmos para malla dan mejores tiempos de ejecución.
- De todos los algoritmos estudiados el que proporciona mejores tiempos de ejecución es el de bloques.

	4 procesadores		8 procesadores		16 procesadores	
	256	512	256	512	256	512
JaEsPaNsAnCoPiB1	6.93	48.13	4.81	27.48	4.17	17.77
JaEsPaNsMaCuRrB1	7.45	54.96			3.29	16.41
JaEsPaSiAnMaEbB1	4.59	22.85	4.46	16.78	4.67	14.99
JaEsPaSiMaPlRrB1	4.77	28.68			2.77	11.23
JaEsPaSiMaPlPiB3	2.80	12.39			1.60	6.32

Tabla 8.1.2: Tiempos de ejecución por barrido en segundos de diferentes algoritmos estudiados para multicomputadores, cuando se usa BLAS 1 (y BLAS 3 en el algoritmo por bloques). En iPSC/860.

De las observaciones anteriores se obtienen las siguientes conclusiones:

1. Se han comparado métodos paralelos que explotan la simetría con otros que no la explotan, usando distintas topologías (anillo, hipercubo y malla). Los métodos que explotan la simetría son teóricamente el doble de rápidos que

los que no la explotan pero, debido a un mayor coste de las comunicaciones y una distribución de los datos más compleja, no se llega a obtener esa división por dos en el tiempo de ejecución en los resultados experimentales, aunque se obtiene una reducción sustancial y que podemos considerar satisfactoria al aumentar el tamaño del problema, y esto se consigue, en mayor o menor medida, en todos los equipos con los que se ha experimentado (PARSYS SN-1040, iPSC/2, iPSC/860 y Touchstone DELTA).

2. En cuanto a métodos que exploten la simetría en un anillo de procesadores:
   - Se han estudiado las ideas generales para obtener esquemas de almacenamiento que permitan explotar la simetría.
   - Se han diseñado algoritmos con los esquemas de almacenamiento por **antidiagonales** y por **marcos**, con distintas ordenaciones de Jacobi (par-impar y de Eberlein), obteniéndose resultados similares con las distintas ordenaciones y los distintos esquemas de almacenamiento, al tener los algoritmos a los que dan lugar un comportamiento idéntico asintóticamente.
3. En cuanto a la explotación de la simetría en una malla de procesadores, se ha estudiado un esquema de almacenamiento de los datos por **plegamiento** con el que se pueden obtener algoritmos teóricamente óptimos, con los que se obtienen buenos resultados tanto a nivel de tiempos de ejecución como de escalabilidad. Este esquema de almacenamiento se puede combinar con distintas ordenaciones de Jacobi, habiéndose estudiado en este trabajo con las ordenaciones Round-Robin y par-impar.
4. Los algoritmos estudiados en malla presentan mejor escalabilidad, tanto a nivel teórico como de resultados experimentales, que los de anillo, aunque estos últimos se ejecuten en un hipercubo utilizando un anillo inmerso en el hipercubo y haciendo la difusión de los parámetros de las rotaciones utilizando la topología de hipercubo.
5. Al ser el esquema de almacenamiento por **plegamiento** en un malla de procesadores el que presenta mejores prestaciones, se ha estudiado la combinación de este esquema con un esquema de trabajo por bloques, obteniéndose un algoritmo eficiente, portable y escalable.

- Respecto a las posibles extensiones:

1. En cuanto al **Problema de Valores Propios Generalizado** para haces simétricos definidos positivos, se ha diseñado un algoritmo para un anillo de procesadores usando el esquema de almacenamiento por **antidiagonales** y se ha comparado con otro algoritmo que no explota la simetría. Se obtienen resultados similares a los obtenidos con el problema estándar, por lo que para

este tipo de problemas generalizados pueden servir los mismos esquemas de almacenamiento que se usan en el problema estándar.

2. En cuanto al **problema simétrico estándar con matrices complejas** se ha diseñado un algoritmo para una malla de procesadores utilizando el esquema de almacenamineto por **plegamiento** para explotar la simetría. El método no siempre converge, pero las eficiencias que se obtienen son satisfactorias y mayores que las que se obtienen en el problema con matrices reales, debido a un mayor coste computacional. Debido a que la única diferencia con el problema real está en el trabajo con números complejos, los mismos esquemas usados con el problema real se pueden usar con el problema complejo.

3. En cuanto a la **iteración QR** para transformar una matriz Hessenberg a forma de Schur se ha estudiado la aplicación de un esquema de almacenamiento por **antidiagonales** para paralelizar un paso de la iteración QR en un anillo lógico de procesadores. El algoritmo obtenido tiene una eficiencia teórica asintótica del 100%, pero las eficiencias obtenidas experimentalmente están muy alejadas del óptimo teórico debido al alto coste de las comunicaciones al aumentar el número de procesadores y al bajo coste aritmético del algoritmo secuencial (n^2).

8.2 Trabajos realizados

Los trabajos realizados que se enumeran se pueden clasificar en distintas categorías:

- Métodos semiclásicos: [50, 23, 122].
- Métodos por bloques: [123, 51, 124, 122, 125, 126, **?**].
- Algoritmos para multiprocesadores de memoria compartida: [124, 125, 126].
- Algoritmos para multicomputadores de memoria distribuida: [127, 128, 129, 130, 123, 131, 134, **?**].
- Extensiones: [129, 132, 133].
- Aplicaciones: [126].

El apartado más importante es el que proporciona el tema central de la tesis: los métodos para multicomputadores de memoria distribuida.

Los enumeramos a continuación cronológicamente:

- **Explotación de la simetría en el algoritmo de Jacobi para el cálculo de valores propios en memoria distribuida** [128]. Es un trabajo realizado para la III Reunión Española de Paralelismo, celebrada en Valencia en septiembre de 1992. Se estudia un método de Jacobi que explota la simetría en un anillo de procesadores utilizando el esquema de almacenamiento por **antidiagonales**. Se analizan los resultados obtenidos con el algoritmo y con una modificación para el problema simétrico de valores propios generalizado en un iPSC/2 (se utilizó el iPSC/2 con 4 procesadores del Centro de Estudios Avanzados de Blanes del Consejo Superior de Investigaciones Científicas) y en un PARSYS SN-1000 (del Grupo de Computación Paralela del Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia). Una descripción más detallada del método se encuentra en el technical report **Explotación de la simetría en el algoritmo de Jacobi paralelo: una implementación en el multiprocesador PARSYS SN-1000** [127].

- **On Jacobi methods for the standard and generalized symmetric eigenvalue problem on Multicomputers** [129]. Es una comunicación presentada en la tercera conferencia de SIAM sobre Algebra Lineal y Aplicaciones en Señales, Sistemas y Control, celebrada en Seattle en agosto de 1993. Se presenta por primera vez un método de Jacobi que explota la simetría en una malla cuadrada de procesadores utilizando el esquema de almacenamiento por **plegamiento** y la ordenación Round-Robin. Se compara este algoritmo con el presentado en [128] obteniéndose que el algoritmo para malla presenta mejor escalabilidad. La comparación experimental se realiza en el PARSYS SN-1000 y en el iPSC/860 con 8 procesadores del Argonne National Laboratory. En el technical report **Computing eigenvalues of symmetric matrices on a mesh of processors** [130] se detalla el algoritmo de Jacobi que explota la simetría en una malla de procesadores. En esta conferencia se entra en contacto con el profesor Robert van de Geijn que ya había utilizado en [94] el esquema de almacenamiento **block-Hankel wrapped** (del que el esquema por antidiagonales es un caso particular) para obtener un método de Jacobi que explotara la simetría en un anillo de procesadores, tal como proponíamos en [128]. Se acuerda trabajar en colaboración para tratar de diseñar un método de Jacobi eficiente y escalable para multicomputadores.

- **Computing the generalized eigenvalues of symmetric positive definite pencils on networks of transputers** [133]. Es una publicación en el número de Microprocessing and Microprogramming dedicado al congreso EUROMICRO93, realizado en septiembre de 1993 en Barcelona. Presenta un algoritmo para la solución del problema simétrico de valores propios generalizado utilizando el esquema de almacenamiento por antidiagonales y la ordenación par-impar. Hace un estudio experimental en el PARSYS SN-1000. Un technical report sobre este trabajo es **Computing generalized eigenvalues on multicomputers** [132].

- **Una modificación del método de Jacobi clásico** [50]. Es una comunicación al III Congreso Español de Matemática Aplicada, realizado en Madrid en septiembre de 1993. Se exponen por primera vez las ideas del método de Jacobi **semiclásico**.

- **On the Semiclassical Jacobi Algorithm** [23]. En esta comunicación a la quinta conferencia de SIAM en Algebra Lineal Aplicada, celebrada en Utah en junio de 1994 se presentan las ideas básicas del método semiclásico y algunos de los resultados presentados en [22].

- **A Jacobi method by blocks to solve the symmetric eigenvalue problem on a mesh of processors** [123]. En esta comunicación en el Congreso de ILAS (International Linear Algebra Society) celebrada en Rotterdam en agosto de 1994 se presenta el método de Jacobi por bloques que explota la simetría en una malla de procesadores utilizando el esquema de almacenamiento por plegamiento y la ordenación par-impar (método estudiado en el capítulo 6). Se presentan resultados experimentales obtenidos en el Touchstone DELTA de 512 procesadores del Californian Institute of Technology. Los resultados obtenidos permiten considerar el algoritmo como eficiente y escalable. En el technical report **A Jacobi method by blocks to solve the symmetric eigenvalue problem** [51] se detalla el método por bloques secuencial utilizado.

- **Esquemas de almacenamiento para el problema simétrico de valores propios en multiprocesadores** [131]. En esta comunicación en los Encuentros de Análisis Matricial celebrados en Victoria en septiembre de 1994 se analizan las características que debe tener un esquema de almacenamiento para poder explotar la simetría en el problema simétrico de valores propios en multicomputadores, y se comparan los esquemas de almacenamiento por antidiagonales, marcos y plegamiento.

- **Efficient Jacobi algorithms on multicomputers** [134]. Este trabajo ha sido admitido en PARA95 (Workshop on Applied Parallel Computing in Physics, Chemistry and Engineering Science) a celebrar en Copenhage en agosto de 1995. Los artículos de este congreso se publicarán como un número del Computer Science Lecture Notes. El contenido del artículo prácticamente coincide con el contenido de [131].

- **Aceleración de la convergencia en métodos de Jacobi para el problema simétrico de valores propios** [122]. Este trabajo ha sido admitido en el IV Congreso Español de Matemática Aplicada a celebrar en Vic en septiembre de 1995. En él se continúa el estudio de los métodos semiclásicos, analizando su comportamiento en diferentes máquinas (HP Apollo 700, i860 y Sillicon Graphic Power Challenge [135]; esta última máquina es un multiprocesador de memoria compartida, y la que se ha usado consta de 4 procesadores y se encuentra en el Servicio de Cálculo Científico de la Universidad de Murcia), con matrices de distintas condiciones y cuando se combina con técnicas de umbralización y con el método por bloques estudiado en [51].

- **Uso de núcleos computacionales en el método de Jacobi para el cálculo de valores propios en multiprocesadores** [125]. Este trabajo también ha sido admitido en el IV Congreso Español de Matemática Aplicada a celebrar en Vic en septiembre de 1995. Básicamente se presentarán los resultados del technical report **Paralelización en multiprocesadores por división del trabajo como alternativa a paralelización usando BLAS: un estudio con el método de Jacobi para el Problema Simétrico de Valores Propios en el Alliant FX/80** [124] donde se estudian diferentes métodos de Jacobi en el multiprocesador Alliant FX/80. Se estudia la paralelización de métodos que usan BLAS 1 y de los que usan BLAS 3 con el esquema de bloques de [51]. Se comprueba que una buena técnica de distribución del trabajo entre los procesadores y el uso de BLAS 3 en cada uno de ellos proporciona mejores resultados que la paralelización exclusivamente por el uso de BLAS 3 y de opciones de compilación. Además, se comprueba de esta manera que el esquema por bloques diseñado en [51] es bastante portable al ser válido para procesadores secuenciales, multiprocesadores y multicomputadores [123] con algunas modificaciones propias del sistema en que se trabaje.

- **Métodos de Jacobi para el problema Simétrico de Valores Propios en Multiprocesadores. Aplicación a solución de ecuaciones de Schrödinger** [126]. Este trabajo está siendo finalizado y se trata de un intento de aplicar la experiencia obtenida sobre la paralelización de métodos de Jacobi a la resolución de ecuaciones de Schrödinger que aparecen en Química Cuántica. Con este fin se está trabajando con el profesor José Zúñiga del Departamento de Química Física de la Universidad de Murcia.

- **Exploiting the symmetry on the Jacobi method on a mesh of processors** [?]. Este artículo ha sido enviado para su evaluación al cuarto EUROMICRO Workshop on Parallel and Distributed Processing, a celebrar en Braga en enero de 1996.

8.3 Trabajos futuros

Se pueden considerar al menos tres grandes líneas de continuación de los trabajos aquí desarrollados: optimización de los métodos de Jacobi estudiados para el Problema Simétrico de Valores Propios, estudio de posibles extensiones además de las analizadas, y aplicaciones de los métodos estudiados a problemas de física, química e ingeniería.

Resumimos las posibles líneas de trabajo de cada uno de estos apartados (en algunas ya se ha empezado a trabajar):

- **Optimización de métodos de Jacobi** para el problema simétrico de valores propios:
  - **Métodos secuenciales**:
    - Combinación de esquemas de aceleración de la convergencia (semiclásico) con otros métodos tipo Jacobi con menor coste computacional [136, 137].
    - Estudio de variantes del método semiclásico para adecuarlo al trabajo por bloques [122].
  - **Métodos paralelos en multicomputadores**:
    - Estudio de otros esquemas de distribución de los datos para minimizar las comunicaciones o asignar tamaños de bloques que permitan un mejor balanceo del uso de BLAS [138].
    - Combinación de los esquemas de almacenamiento que permiten explotar la simetría con métodos de ocultamiento de las comunicaciones [83, 139].
    - Optimización del método por bloques estudiado (posiblemente utilizando las técnicas enunciadas antes) para compararlo con otros métodos adecuados para Máquinas Masivamente Paralelas. En particular se puede comparar con la técnica de descomposición en subespacios invariantes que parece prometedora para este tipo de sistemas y con la que actualmente se obtienen resultados ligeramente mejores que los que obtenemos con el método de Jacobi por bloques en el Touchstone Delta [121].

- **Extensiones**.

  Hemos analizado cómo los esquemas de almacenamiento usados para explotar la simetría en el Problema Simétrico de Valores Propios se pueden aplicar con éxito a otros problemas de valores propios: para explotar la simetría en el Problema Simétrico Generalizado de Valores Propios y en el Problema Simétrico Estándar Complejo, y para obtener un algoritmo teóricamente óptimo para la iteración QR aplicada a la reducción a forma de Schur de una matriz Hessenberg. Hay multitud de otras posibles extensiones, algunas de las cuales son

  - Aplicación de los esquemas a problemas banda.
  - Aplicación a problemas dispersos, donde los métodos tipo Jacobi pueden parecer más apropiados que la transformación a forma tridiagonal.
  - Métodos de Jacobi unilaterales, que son muy adecuados para programación paralela.

- **Aplicaciones**.

  En el capítulo 1 ya se señalaban algunas posibles aplicaciones a diferentes campos de la Física, la Química y la Ingeniería de los problemas de valores propios. Se puede considerar la aplicación de los métodos de Jacobi aquí estudiados a la resolución de grandes problemas de valores propios que aparecen en estos campos. En particular, se han establecido contactos para trabajar en los siguientes problemas:

  - En **Química Cuántica** en la resolución de **ecuaciones de Scrödinger** que se plantean en el estudio de sistemas de partículas [126].
  - En el **análisis de guiaondas**, en problemas con matrices simétricas densas de dimensiones unos pocos cientos, con valores complejos donde hay que calcular todos los valores propios [81].
  - En **reconocimiento de patrones**, en el cálculo de los valores propios mayores de una matriz de correlaciones. En este caso las matrices son densas y simétricas, y hay que calcular algunos de los valores propios, aunque en muchos casos no se conoce de antemano la cantidad de valores a calcular.

Bibliografia

[1] P. Arbenz y M. Oettli. Block implementations of the symmetric QR and Jacobi algorithms. Technical report, 1992.

[2] J. I. Aliaga. Estudio de las prestaciones del Supernodo PARSYS SN-1000. Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia. 1992.

[3] P. R. Amestoy, M. J. Dayde y I. S. Duff. Use of Level 3 BLAS kernels in the solution of full and sparse linear equations. Technical Report TR 89/9, March 1989.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov y D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.

[5] L. Auslander y A. Tsao. On parallelizable eigensolvers. *Ad. App. Math.*, 13:253–261, 1992.

[6] Robert G. Babb II. *Programming Parallel Processors*. Addison-Wesley, 1988.

[7] Zhaojun Bai y James Demmel. On a block implementation of Hessenberg multishift QR iteration. *International Journal of High Speed Computing*, 1(1):97–112, 1989.

[8] M. Barnett, R. Littlefield, D. G. Payne y R. van de Geijn. On the efficiency of global combine algorithms for 2-D meshes with wormhole routing. preprint, 1993.

[9] Michael Barnett, David Payne, Robert van de Geijn y Jerrel Watts. Broadcasting on meshes with worm-hole routing. preprint, 1993.

[10] Ilan Bar-On y Victor Ryaboy. A New Fast Algorithm for the Diagonalization of Complex Symmetric Matrices, with Applications to Quantum Dynamics. In David H. Bailey, Petter E. Bjørstad, John R. Gilbert, Michael V. Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia J. Torczon y Layne T. Watson,

editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 571–572. SIAM, 1995.

[11] Klaus-Jürgen Bathe. *Finite Element Procedures in Engineering Analisys.* Prentice-Hall, 1982.

[12] D. P. Bertsekas, C. Özveren, G. D. Stamoulis, P. Tseng y J. N. Tsitsiklis. Optimal communication algorithms for hypercubes. *Journal of Parallel and Distributed Computing*, 11:263–275, 1991.

[13] Dimitri P. Bertsekas y John N. Tsitsiklis. *Parallel and Distributed Computation. Numerical Methods.* Prentice Hall, 1989.

[14] Christian H. Bischof. Adaptive blocking in the QR factorization. *The Journal of Supercomputing*, 3:193–208, 1989.

[15] Christian H. Bischof. Computing the singular value decomposition on a distributed system of vector processors. *Parallel Computing*, 11:171–186, 1989.

[16] Christian H. Bischof. A parallel QR factorization algorithm with controlled local pivoting. *SIAM J. Sci. Stat. Comput.*, 12(1):36–57, 1991.

[17] R. H. Bisseling y J. G. G. Van de Vorst. Parallel triangular system solving on a mesh network of transputers. *SIAM J. Sci. Stat. Comp.*, 12(4):787–799, 1991.

[18] Luc Bomans y Dick Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience*, 1(1):3–18, 1989.

[19] Richard P. Brent y Franklin T. Luk. A systolic architecture for almost linear-time solution of the symmetric eigenvalue problem. Technical Report TR-CS-82-10, Department of Computer Science, Australian National University, Camberra, August 1982.

[20] Richard P. Brent y Franklin T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Stat. Comput.*, 6(1):69–84, 1985.

[21] A. Bunse-Gerstner y H. Fassbender. On the generalized Schur decomposition of a matrix pencil for parallel computation. *SIAM J. Sci. Stat. Comput.*, 12(4), 1991.

[22] Mari Trini Cámara. Métodos de Jacobi semiclásicos en problemas de valores propios. Facultad de Informática. Universidad de Murcia. Proyecto Fin de Carrera. Dirigido por D. Giménez. 1994.

[23] Maria T. Cámara y Domingo Giménez. On the Semiclassical Jacobi Algorithm. In John G. Lewis, editor, *Proceedings of the Fifth SIAM Conference on Applied Linear Algebra*, pages 285–289. SIAM, 1994.

[24] Francisco José Carvajal Rojo. Optimización de coordenadas vibracionales hiperesféricas para la molécula de H_3^+. Facultad de Química. Universidad de Murcia. Tesina. Dirigido por J. Zúñiga. 1995.

[25] J-P. Charlier y P. Van Dooren. A Jacobi-like algorithm for computing the generalized Schur form of a regular pencil. *Journal of Computational and Applied Mathematics*, 27:17–36, 1989.

[26] C. Conde Lázaro y G. Winter Althaus. *Métodos y algoritmos básicos del álgebra numérica*. Reverté, 1990.

[27] Javier Cuenca Muñoz. Métodos de Jacobi semiclásicos en Multiprocesadores. Facultad de Informática. Universidad de Murcia. Proyecto Fin de Carrera. Dirigido por D. Giménez. 1994.

[28] K. Dackland, E. Elmroth, B. Kågström y Charles Van Loan. Parallel block matrix factorizations on the shared memory multiprocessos IBM 3090 VF/600J. *The International Journal of Supercomputer Applications*, 6(1):69–97, 1986.

[29] Ray O. Davies y J. J. Modi. A Direct Method for Completing Eigenproblem Solutions on a Parallel Computer. *Linear Algebra and its Applications*, 77, 1986.

[30] M. J. Daydé y I. S. Duff. Use of Level 3 BLAS in LU factorization in a multiprocessing environment on three vector multiprocessors: the Alliant FX/80, the Cray-2, and the IBM 3090 VF. *The International Journal of Supercomputer Applications*, 5(3):92–110, 1991.

[31] Michel J. Daydé, Iain S. Duff y Antoine Petitet. A Parallel Block Implementation of Level-3 BLAS for MIMD Vector Processors. *ACM Transactions on Mathematical Software*, 20(2):178–193, 1994.

[32] Edmund X. Dejesus. Face Value. *BYTE*, pages 85–88, 1995.

[33] David DeMers y Garrison Cotrell. Non-Linear Dimensionality Reduction. 1995.

[34] James W. Demmel, Michael T. Heath y Henk A. van der Vorst. Parallel numerical linear algebra. In *Acta Matematica*, pages 111–197. Cambridge University Press, 1993.

[35] J. Demmel y K. Veselić. Jacobi's method is more accurate than QR. *SIAM J. Matrix Anal. Appl.*, 13:1204–1245, 1992.

[36] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling y Iain Duff. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. on Math. Soft.*, 16(1):1–17, 1990.

[37] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling y Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. on Math. Soft.*, 14(1):1–17, 1988.

[38] Jack J. Dongarra y D. C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. Stat. Comput.*, 8(2):139–154, 1987.

[39] Jack J. Dongarra y Robert A. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory architectures. *Parallel Computing*, 18:973–982, 1992.

[40] T. H. Dunigan. Performance of the Intel iPSC/860 and the Ncube6400 hypercubes. *Parallel Computing*, 17:1285–1302, 1987.

[41] P. J. Eberlein y Mythili Mantharam. Jacobi sets for the eigenproblem and their effect on convergence studied by graphic representation. In *Proceedings Fourth SIAM Conf. on Parallel Processing for Scientific Computing*, pages 15–22. SIAM, 1989.

[42] P. J. Eberlein y Haesun Park. Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 8:358–366, 1990.

[43] A. Edelman. Large dense linear algebra in 1993: The parallel computing influence. *The International Journal of Supercomputer Applications*, 7(2):113–128, 1993.

[44] F. A. Fernández, J. B. Davies, S. Zhu y Y. Lu. Sparse matrix Eigenvalue Solver for Finite Element solution of Dielectric Waveguides. *ELECTRONICS LETTERS*, 27(20):1824–1826, 1991.

[45] F. A. Fernández y Y. Lu. Variational Finite Element analysis of Dielectric Waveguides with no spurious solutions. *ELECTRONICS LETTERS*, 26(25):2125–2126, 1990.

[46] F. A. Fernández y Y. Lu. A variational Finite Element formulation for Dielectric Waveguides in terms of transverse magnetic fields. *IEEE Trans. on Magnetics*, 27(5):3864–3867, 1991.

[47] G. E. Forsythe y P. Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Trans. Amer. Math. Soc.*, 94:1–23, 1960.

[48] K. A. Gallivan, R. J. Plemmons y A. H. Sameh. Parallel algorithms for dense linear algebra computations. In K. A. Gallivan, Michael T. Heath, Esmong Ng, James M. Ortega, Barry W. Peyton, R. J. Plemmons, Charles H. Romine, A. H. Sameh y Robert G. Voigt, editor, *Parallel Algorithms for Matrix Computations*, pages 1–82. SIAM, 1990.

[49] B. S. Garbow, J. M. Boyle, J. J Dongarra y C. B. Moler. *Matrix Eigensystem Routines-EISPACK Guide Extension*. Springer-Verlag, 1972.

[50] Domingo Giménez. Una modificación del método de Jacobi clásico. III Congreso Español de Matemática Aplicada. Madrid., 1993.

[51] Domingo Giménez, Vicente Hernández, Robert A. van de Geijn y Antonio M. Vidal. A Jacobi method by blocks to solve the symmetric eigenvalue problem. Technical Report TR 1-95, Dept de Informática y Sistemas. Universidad de Murcia, 1995.

[52] Gene H. Golub y James M. Ortega. *Scientific Computing, an introduction with Parallel Computing*. Academic Press, 1993.

[53] G. H. Golub y C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989. Segunda Edición.

[54] J. Götze, S. Paul y M. Sauer. An efficient Jacobi-like algorithm for parallel eigenvalue computation. *IEEE Trans. Computers*, 42(9), 1993.

[55] Eldon R. Hansen. On cyclic Jacobi methods. *J. Soc. Indust. Appl. Math.*, 11(2):448–459, 1963.

[56] Michael T. Health, George A. Geist y John B. Drake. Early experience with the Intel iPSC/860 at Oak Ridge National Laboratory. *The International Journal of Supercomputer Applications*, 5(2):10–26, 1991.

[57] Bruce Hendrickson. Parallel QR factorization using the torus-wrap mapping. *Parallel Computing*, 19:1259–1271, 1993.

[58] Greg Henry y Robert van de Geijn. Parallelizing the QR Algorithm for the Unsymmetric Algebraic Eigenvalue Problem: myths and reality. Technical Report LAPACK WN 79.

[59] Nicholas J. Higham. Algorithm 694. A Collection of Test Matrices in MATLAB. *ACM Trans. on Math. Software*, 17(3):289–305, 1991.

[60] Ching-Tien Ho. Spanning trees and communication on hypercubes. In Füsun Özgüner y Fikret Ercal, editor, *Parallel Computing on Distributed Memory Multiprocessors*, pages 47–75. NATO ASI Series, 1992.

[61] Ching-Tien Ho y S. L. Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercube. *IEEE*, pages 640–648, 1986.

[62] Roger W. Hockney y Edward A. Carmona. Comparison of communications on the Intel iPSCiPSC/860 and Touchstone Delta. *Parallel Computing*, 18:1067–1072, 1992.

[63] R. W. Hockney y C. R. Jesshope. *Parallel Computers*. Adam Hilger, 1984.

[64] Alston S. Householder. *The theory of matrices in numerical analysis*. Dover Publications, 1975.

[65] S. Huss-Lederman, A. Tsao y G. Zhang. A parallel implementation of the invariant subspace decomposition algorithm for dense symmetric matrices. In *Proceedings Sixth SIAM Conf. on Parallel Processing for Scientific Computing*. SIAM, 1993.

[66] Kai Hwang. *Advanced Computer Architecture. Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[67] C. G. J. Jacobi. Uber ein leichtes Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen. *Journal für die Reine and Anbewante Mathematic*, 30:51–94, 1846.

[68] C. G. J. Jacobi. On a New Way of Solving the Linear Equations that Arise in the Method of Least Squares. Traducción de G. W. Stewart. Technical Report TR-92-42,TR-2877, Institute for Advanced Computer Studies. Department of Computer Science. University of Maryland, April 1992.

[69] Alan H. Karp y John Greenstadt. A improved parallel Jacobi method for diagonalizing a symmetric matrix. *Parallel Computing*, 5:281–294, 1987.

[70] Steven Kartashev y Svetlana Kartashev, editor. *Supercomputing Systems. Architectures, Design, and Performance*. Van Nostrand Reinhold, 1989.

[71] D. E. Knuth. *El arte de programar ordenadores. Vol 3: clasificación y búsqueda*. Reverté, 1987.

[72] Vipin Kumar, Ananth Grama, Anshul Gupta y George Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. The Benjamin Cummings Publishing Company, 1994.

[73] Ira N. Levine. *Química Cuántica*. Editorial AC, 1977.

[74] Ira N. Levine. *Espectroscopia Molecular*. Editorial AC, 1980.

[75] Franklin T. Luk y Haesun Park. A proof of convergence for two parallel Jacobi SVD algorithms. *IEEE Trans. Computers*, 28(6):806–811, 1989.

[76] Franklin T. Luk y Haesun Park. On parallel Jacobi orderings. *SIAM J. Sci. Stat. Comput.*, 10(1):18–26, 1989.

[77] Mythili Mantharam y P. J. Eberlein. New Jacobi-sets for parallel computations. *Parallel Computing*, 19:437–454, 1993.

[78] Mythili Mantharam y P. J. Eberlein. Block recursive algorithm to generate Jacobi-sets. *Parallel Computing*, 19:481–496, 1993.

[79] Walter F. Mascarenhas. A note on Jacobi being more accurate than QR. *SIAM J. Matrix Anal. Appl.*, 15(1):215–218, 1994.

[80] J. J. Modi. *Parallel Algorithms and Matrix Computation.* Clarendon Press, 1988.

[81] Luis Nuño, Juan V. Balbastre, Margarita Ramón, Jorge Igual y Miguel Ferrando. Analysis of Inhomogeneus and Anisotropic Waveguides by the Finite Element Method. preprint, 1995.

[82] B. N. Parlett. *The Symmetric Eigenvalue Problem.* Prentice-Hall, 1980.

[83] Makan Pourzandi y Bernard Tourancheau. A Parallel Performance Study of Jacobi-like Eigenvalue Solution. Technical report, March 1994.

[84] P. P. M. Rijk. A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer. *SIAM J. Sci. Stat. Comput.*, 10(2):359–371, 1989.

[85] A. H. Sameh. On Jacobi and Jacobi-like algorithms for a parallel computer. *Math. Comput.*, 25:579–590, 1971.

[86] Thomas Schreiber, Peter Otto y Fridolin Hofmann. A new efficient parallelization strategy for the QR algorithm. *Parallel Computing*, 20:63–75, 1994.

[87] Gautam Schroff y Robert Schreiber. On the convergence of the cyclic Jacobi method for parallel block orderings. *SIAM J. Matrix Anal. Appl.*, 10(3):326–346, 1989.

[88] Uwe Schwiegelshon y Lothar Thiele. A systolic array for ciclic-by-rows Jacobi algorithms. *Journal of Parallel and Distributed Computing*, 4:334–340, 1987.

[89] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema y C. B. Moler. *Matrix Eigensystem Routines-EISPACK Guide.* Springer-Verlag, 1976.

[90] G. W. Stewart. *Introduction to matrix computations.* Academic Press, 1973.

[91] G. H. Stewart. A Jacobi-like algorithm for computing the Schur decomposition of a nonhermitian matrix. *SIAM J. Sci. Stat. Comput.*, 4:853–864, 1985.

[92] P. Tervola y W. Yeung. Parallel Jacobi algorithm for matrix diagonalisation on transputer networks. *Parallel Computing*, 17:155–163, 1991.

[93] Robert A. van de Geijn. Implementing the QR-algoritm on an array of processors. Technical Report TR-1897, University of Maryland, August 1987.

[94] Robert A. van de Geijn. A novel storage scheme for parallel Jacobi methods. Technical Report TR-88-26, Department of Computer Science, University of Texas at Austin, Austin, July 1988.

[95] Robert A. van de Geijn. Storage schemes for Parallel Eigenvalue Algorithms. In G. H. Golub y P. Van Dooren, editor, *Numerical Linear Algebra. Digital Signal Processing and Parallel Algorithms*, volume 70 of *NATO ASI Series*. Springer-Verlag, 1991.

[96] Robert A. van de Geijn. Deferred shifting schemes for parallel QR methods. *SIAM J. Matrix Anal. Appl.*, 14(1):180–194, 1993.

[97] Robert A. van de Geijn y D. G. Hudson III. Efficient parallel implementation of the nonsymmetric QR algorithm. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1991.

[98] Antonio M. Vidal. Algoritmos Matriciales Paralelos y Arquitecturas Asociadas para el cálculo de la SVD. Ph. Tesis. D.S.I.C. Univ Politécnica de Valencia, 1990.

[99] David S. Watkins. *Matrix Computations.* John Wiley & Sons, 1991.

[100] David S. Watkins. Shifting strategies for the parallel QR algorithm. *SIAM J. Sci. Comput.*, 15(4):953–958.

[101] Robert A. Whiteside, Neil S. Ostlund y Peter G. Hibbard. A parallel Jacobi diagonalization algorithm for a loop multiple processor system. *IEEE Trans. Computer*, 33(5):409–413, 1984.

[102] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, 1965.

[103] *FX/C Programmer's Handbook.* Alliant Computer Systems Corporation, 1989.

[104] *FX Series Architecture Handbook.* Alliant Computer Systems Corporation, 1989.

[105] *Alliant Scientific Library.* Alliant Computer Systems Corporation, 1989.

[106] *The Transputer Databook.* INMOS, 1989.

[107] *iPSC/2 .C Programmer's reference Manual.* Intel Corporation, 1988.

[108] *iPSC/2 Users´ s Guide.* Intel Corporation, October 1989.

[109] *iPSC/860 C Compiler Users´ s Guide.* Intel Corporation, April 1991.

[110] *Parallel C User Guide.* Intel Corporation, 1989.

[111] *Touchstone DELTA System Description.* Intel Corporation, February 1991.

[112] Michael Berry y Ahmed Sameh. An overview of parallel algorithms for the singular value and symmetric eigenvalue problems. *Journal of Computational and Applied Mathematics*, 27, 1989.

[113] J. J. Modi y J. D. Pryce. Efficient Implementation of Jacobi's Diagonalization Method on the DAP. *Numerische Mathematik*, pages 443–454, 1985.

[114] David S. Scott, Michael T. Heath y Robert C. Ward. Parallel Block Jacobi Eigenvalue Algorithms Using Systolic Arrays. *Linear Algebra and its Applications*, 77:345–355, 1986.

[115] J. S. Weston y M. Clint. Two algorithms for the parallel computation of eigenvalues and eigenvectors of large symmetric matrices using the ICL DAP. *Parallel Computing*, 13:281–288, 1990.

[116] Zhaojun Bai. Progress in the numerical solution of the nonsymmetric eigenvalue problem. Technical report, Department of Mathematics, University of Kentucky, 1992.

[117] Vipin Kumar y Anshul Gupta. Analyzing Scalability of Parallel Algorithms and Architectures. *Journal of Parallel and Distributed Computing*, 22:379–391, 1994.

[118] P. J. Eberlein. On the Schur decomposition of a matrix for parallel computation. *IEEE Trans. Computers*, 36(2), 1987.

[119] M. H. C. Paardekooper. A quadratically convergent parallel Jacobi process for diagonally dominant matrices with distinct eigenvalues. *Journal of Computational and Applied Mathematics*, 127:3–16, 1989.

[120] M. H. C. Paardekooper. A quadratically convergent parallel Jacobi-like eigenvalue algorithm. In D. J. Evans, G. R. Joubert y F. J. Peters, editor, *Parallel Computing 89*, pages 173–179. North-Holland, 1990.

[121] Christian Bischof, Steven Huss-Lederman, Anna Tsao, Thomas Turnbull y Xiaobai Sun. The prism project: Portable application-driven eigensolvers. Report, Concurrent Supercomputing Consortium, 1995.

[122] Mari Trini Cámara, Javier Cuenca y Domingo Giménez. Aceleración de la convergencia en métodos de Jacobi para el problema simétrico de valores propios. IV Congreso Espñol de Matemática Aplicada, 1995.

[123] Domingo Giménez, Vicente Hernández, Robert van de Geijn y Antonio M. Vidal. A Jacobi method by blocks to solve the symmetric eigenvalue problem on a mesh of processors. ILAS 94 Conference, Rotterdam 15-18 august 1994, 1994.

[124] Javier Cuenca, Domingo Giménez, Vicente Hernández, Antonio M. Vidal y Javier Zapata. Paralelización en multiprocesadores por división del trabajo como alternativa a paralelización usando BLAS: un estudio con el método de Jacobi para el Problema Simétrico de Valores Propios en el Alliant FX/80. Technical Report TR 3-95, Universidad de Murcia, 1995.

[125] Javier Cuenca Muñoz, Domingo Giménez Cánovas y Javier Zapata Martínez. Uso de núcleos computacionales en el método de Jacobi para el cálculo de valores propios en multiprocesadores. IV Congreso Español de Matemática Aplicada. 1995.

[126] Javier Cuenca Muñoz, Domingo Giménez Cánovas, Javier Zapata Martínez, Vicente Hernández García, Antonio M. Vidal Maciá, Robert van de Geijn y José Zúñiga. Métodos de Jacobi para el problema Simétrico de Valores Propios en Multiprocesadores. Aplicación a solución de ecuaciones de Schrödinger. En preparación. 1995.

[127] Domingo Giménez, Vicente Hernández, y Antonio M. Vidal. Explotación de la simetría en el algoritmo de Jacobi paralelo: una implementación en el multiprocesador PARSYS SN-1000. Technical Report DSIC-II/16/92, Dept de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia, 1992.

[128] Domingo Giménez, Vicente Hernández y Antonio M. Vidal. Explotación de la simetría en el algoritmo de Jacobi para el cálculo de valores propios en memoria distribuida. *III Reunión Española de Paralelismo*, 1992.

[129] Domingo Giménez, Vicente Hernández y Antonio M. Vidal. On Jacobi methods for the standard and generalized symmetric eigenvalue problem on multicomputers. *Third SIAM Conference on Linear Algebra in Signals, Systems and Control*, 1993.

[130] Domingo Giménez, Vicente Hernández y Antonio M. Vidal. Computing eigenvalues of symmetric matrices on a mesh of processors. Technical Report TR 1-94, Universidad de Murcia, 1994.

[131] Domingo Giménez, Vicente Hernández y Antonio M. Vidal. Esquemas de almacenamiento para el problema simétrico de valores propios en multiprocesadores. *Encuentros de Análisis Matricial*, 1994.

[132] Domingo Giménez, Vicente Hernández, y Antonio M. Vidal. Computing generalized eigenvalues on multicomputers. Technical Report TR 1-93, Dept de Informática y Automática. Universidad de Murcia, 1993.

[133] Domingo Giménez, Vicente Hernández y Antonio M. Vidal. Computing the generalized eigenvalues of symmetric positive definite pencils on networks of transputers. *Microprocessing and Microprogramming*, 38:335–342, 1993.

[134] Domingo Giménez, Vicente Hernández y Antonio M. Vidal. Efficient Jacobi Algorithms on Multicomputers. In Kaj Madsen y Jerzy Waśniewski Jack J. Dongarra, editor, *Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, Second International Workshop, PARA'95*, pages 247–256. Springer-Verlag. Lecture Notes in Computer Science 1041, 1995.

[135] Symmetric Multiprocessing Systems. Technical report, Sillicon Graphics. Computer Systems.

[136] Niloufer Mackey. Hamilton and Jacobi meet again: quaternions and the eigenvalue problem. *SIAM Journal on Matrix Analysis and applications*, 16(2):421–435, 1995.

[137] Mythili Mantharam y P. J. Eberlein. The Real Two-Zero Algorithm: A Parallel Algorithm to Reduce a Real Matrix to a Real Schur Form. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):48–62, 1995.

[138] Juan Manuel Beltrán Barrionuevo. Un algoritmo genético para el problema de la asignación de tareas a trabajadores. Aplicación a algoritmos paralelos. Proyecto Fin de Carrera. Dirigido por F. Jiménez y D. Giménez. Facultad de Informática. Universidad de Murcia. 1994.

[139] Volker Strumpen y Peter Arbenz. Improving Scalability by Communication Latency Hiding. In David H. Bailey, Petter E. Bjørstad, John R. Gilbert, Michael V. Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia J. Torczon y Layne T. Watson, editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 778–779. SIAM, 1995.

TESIS PRESENTADA EN EL

DEPARTAMENTO DE SISTEMAS INFORMATICOS Y COMPUTACION

DE LA UNIVERSIDAD POLITECNICA DE VALENCIA

METODOS DE JACOBI PARA LA RESOLUCION DEL PROBLEMA SIMETRICO DE VALORES PROPIOS EN MULTICOMPUTADORES

Presentada por:

D. Domingo Giménez Cánovas

Bajo la dirección de:

Dr. D. Vicente Hernández García

Dr. D. Antonio M. Vidal Maciá

Mayo, 1995