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

Javier Cuenca Muñoz Domingo Giménez Cánovas Javier Zapata Martínez *
Vicente Hernández García Antonio M. Vidal Maciá †

1 Introducción

La técnica más usada en Multiprocesadores (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 [1, 2]) 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 [3] con la multiplicación de matrices.

En este trabajo analizaremos cómo la técnica de división del trabajo entre los distintos procesadores del sistema y la utilización en cada procesador de las rutinas de BLAS 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 trabajo compararemos estas dos formas de paralelización en un Alliant FX/80, que es un Multiprocesador de Memoria Compartida con 8 procesadores que acceden a la memoria global por medio de un bus común. El lenguaje de programación usado en la implementación de los algoritmos ha sido C [4].

El problema que se tratará será el de la solución del Problema Simétrico de Valores Propios (PSVP) por medio de métodos de Jacobi. Hay multitud de métodos de Jacobi para el PSVP en los que se puede usar fácilmente BLAS 1 [5, 6, 7, 8, 9]. En [10] proponemos otro método que usa BLAS 3 utilizando ideas similares a las que se utilizaban en [11] para el problema de valores singulares. En el

*Departamento de Informática y Sistemas. Univ de Murcia. Aptdo 4021. 30001 Murcia. (domingo@dif.um.es)
†Departamento de Sistemas Informáticos y Computación. Univ Politécnica de Valencia. Aptdo 22012. 46071 Valencia. (cpvhg@dsic.upv.es, cpavm@dsic.upv.es)

presente trabajo analizaremos cómo este esquema de trabajo por bloques puede ser utilizado también en Multiprocesadores para obtener algoritmos eficientes. Por tanto, estudiaremos la utilización tanto de BLAS 1 (sección 3) como de BLAS 3 (sección 4) para explotar el paralelismo de forma global y compararemos esta técnica con la de explotar el paralelismo dividiendo el trabajo y usando las rutinas de BLAS (1 ó 3) en cada procesador de forma local.

2 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 trabaja construyendo una secuencia de matrices $\{A_l\}$ por medio de

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

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 [12, 13], la secuencia $\{A_l\}$ converge a una matriz diagonal D,

$$D = Q_k Q_{k-1} \ldots Q_2 Q_1 A Q_1^t Q_2^t \ldots Q_{k-1}^t Q_k^t \tag{2}$$

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

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 [13]

$$\tan\ 2\theta = \frac{2a_{ij}}{a_{ii} - a_{jj}} \tag{3}$$

Los métodos de Jacobi más usados son los métodos cíclicos, que trabajan realizando sucesivos barridos hasta que se cumple algún criterio de convergencia, normalmente hasta que $off(A) < cota$, consistiendo cada barrido en la anulación sucesiva de todos los elementos no diagonales de A utilizando para esto un cierto orden de Jacobi, por ejemplo, el método cíclico por filas anula los elementos en el orden dado por los pares:

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

La convergencia del método de Jacobi con una ordenación por filas se analiza en [12]. Y en [14, 6, 7, 15, 16] se demuestra que el método de Jacobi converge con otros esquemas de ordenación bajo las mismas condiciones que con el orden cíclico por filas.

En este trabajo utilizaremos el orden cíclico por filas y el par-impar, en el que los índices se agrupan como se indica a continuación para $n = 8$:

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

De este modo, un esquema del método podría ser:

Algoritmo .1 *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 $Q(i, j, \theta)$ tal que $(QAQ^t)_{ij} = 0$

$A = QAQ^t$

FINMIENTRAS

FINMIENTRAS

3 Uso de BLAS 1

En el esquema que hemos visto aparece una actualización de la matriz A (QAQ^t) que puede hacerse usando BLAS 1 [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** para actualizar las partes 1, 2 y 3 de la matriz A en la figura 1.

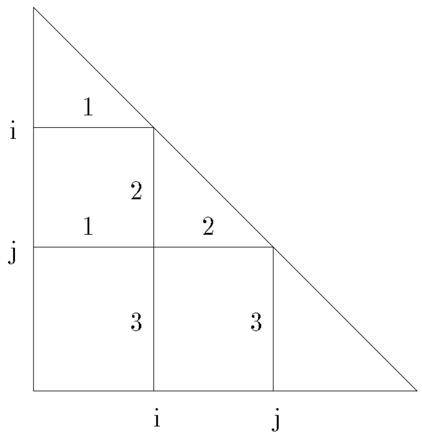

Figura 1: Actualización de A usando BLAS 1.

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 y al compilador. 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 1 mostramos los Mflops obtenidos 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 [17] y ocurre debido a que se obtienen mejores prestaciones con vectores cuyo tamaño se acomoda más al tamaño de la caché.

En las tablas 2 y 3 se muestran, respectivamente, el speed-up y la eficiencia obtenidas con **drot**. Estos valores se obtienen tomando como tiempo de ejecución secuencial el obtenido ejecutando los programas de prueba en un único procesador, y nos dan idea de las prestaciones que podemos llegar a esperar de los métodos de Jacobi que usen BLAS1.

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 1: Mflops obtenidos con la rutina **drot**

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), y se observa que con 2 y 4 procesadores se 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). Esta reducción en las prestaciones al aumentar el número de procesadores apoya nuestra 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 2: Speed-up obtenida con la rutina **drot**

Estudiaremos el comportamiento de BLAS 1 en el método de Jacobi para el PSVP en el Alliant FX/80 comparando distintos algoritmos:

- Un método de Jacobi usando la ordenación par-impar (PI), utilizando la idea de [8] para evitar movimientos de datos, haciéndose éstos en la actualización de la 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 contribuirá a que no se obtengan las prestaciones (en cuanto a Mflops) que se muestran en la tabla 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.

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: Eficiencia obtenida con la rutina **drot**

- 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).

En todos los algoritmos que usan la ordenación par-impar se realiza en cada paso del algoritmo un movimiento de datos entre filas y columnas correspondientes a pares de índices (además de aplicar la correspondiente rotación), y este movimiento es necesario para tener siempre los elementos a anular en bloques de tamaño 2×2 en la diagonal principal, tal como se explica en [10] para el algoritmo por bloques que utilizaremos en la siguiente sección. Este movimiento de datos nos permitirá realizar una asignación del trabajo entre los procesadores en cada paso del algoritmo de manera que sea sencilla la explotación de la simetría.

Explicaremos de una manera más detallada la distribución del trabajo entre los procesadores cuando se balancea la carga (PI y PIBB1) y cuando se distribuyen los datos cíclicamente (PICB1).

3.1 Balanceo de la carga

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

Cada barrido tiene el esquema:

Algoritmo .2 *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

Debido a que se dispone de una memoria común a todos los procesadores cada uno de ellos tendrá acceso a todos los elementos de la matriz en cada paso del algoritmo, pero, para obtener una implementación eficiente, necesitaremos determinar en cada paso del algoritmo zonas disjuntas de la matriz en las que trabajen procesadores diferentes. De este modo, en cada paso del algoritmo se determinan los datos sobre los que trabajará cada procesador, pero entendiendo que esta asignación de datos a procesadores es en realidad una asignación de trabajos a realizar.

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 $\frac{n}{2p}$ 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 $\frac{n}{2}-i\frac{n}{4p}-1$.

En la figura 2, se muestra esta asignación cuando se trabaja con una matriz de tamaño 32×32 (la parte triangular inferior) agrupando las filas por pares, y suponiendo que se dispone de 4 procesadores, con lo que a cada procesador le corresponden 4 pares de filas.

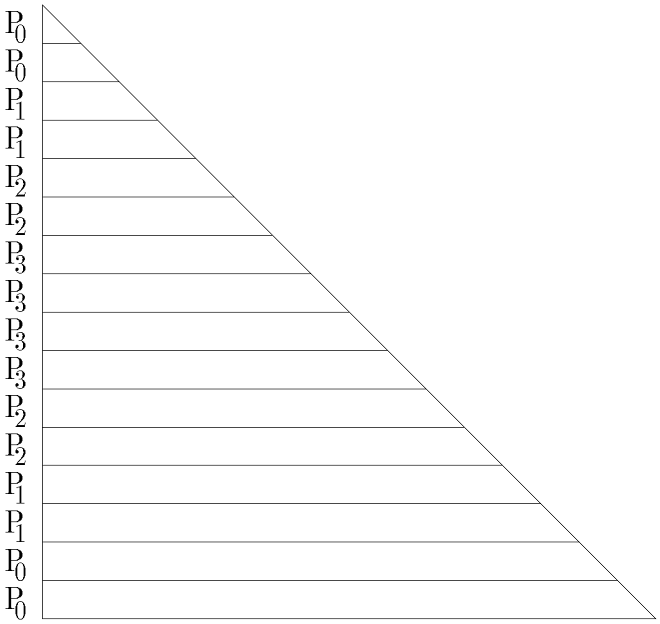

Figura 2: Distribución por balanceo de la carga

Para la postmultiplicación por las rotaciones impares se supone la matriz dividida en pares de columnas consecutivas y se asigna a cada procesador $\frac{n}{2p}$ 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 (calcular rotaciones, premultiplicar por los parámetros y postmultiplicar por los parámetros), 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, tal como se ve en (5).

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

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 $\frac{n}{2p}$ 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 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 $\frac{n}{2p}$ 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:

$$P(QAQ^t)P^t = (PQ)A(Q^tP^t) \tag{6}$$

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 [8]. 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:

$$\begin{pmatrix} -s & c \\ c & s \end{pmatrix} \tag{7}$$

en vez de considerarla de la forma:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \tag{8}$$

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 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 $\frac{n}{2p}$ 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).

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

De modo similar es la actualiación en los pasos impares.

3.3 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 generados aleatoriamente. En todas las tablas se mostrarán tiempos de ejecución en segundos por

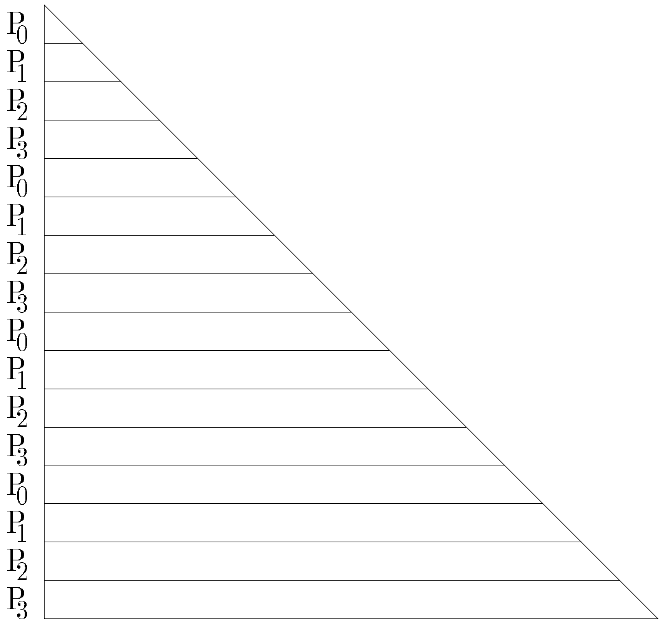

Figura 3: Distribución cíclica

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 4 se muestran los tiempos obtenidos con el algoritmo PI para distintos tamaños de matriz y del sistema. Se muestra también la eficiencia. En esta tabla y en las siguientes la eficiencia se obtendrá considerando como algoritmo secuencial el mismo algoritmo que se está paralelizando, pues estamos interesados en estudiar las posibilidades de paralelización de algoritmos de distintos tipos en Multiprocesadores. 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 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 5, 6 y 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 4 y 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 entre 5 y 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 tantos procesos como

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

procesadores se están usando 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 que produce un tiempo de ejecución adicional que puede ser importante. 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 2 y 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 2 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 8 se muestran los Mflops obtenidos con PICB1, que es el mejor algoritmo de los que usa BLAS. Si comparamos con la tabla 2 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.

Comparando los tiempos de ejecución obtenidos con PIBB1 y PICB1 (tablas 5 y 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 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 obtenidos para cada tamaño del sistema y de la matriz, y se

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

	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 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 8: Mflops con PICB1

marca en cada caso la mejor de las medias obtenidas, correspondiendo esta en la gran mayoría de los casos a PICB1/FIB1.

De las observaciones anteriores podemos concluir:

- 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 y al compilador (tal como conjeturábamos).
- 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 de lugar a 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 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 9: Cocientes entre los tiempos de ejecución obtenidos con PIBB1 y PICB1 y los obtenidos con FIB1.

Finalmente, en la tabla 10 mostramos los tiempos de ejecución por barrido y las eficiencias obtenidos con PIB1. Comparando los resultados de las tablas 7 y 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 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 10: Tiempos de ejecución por barrido en segundos y eficiencia, con PIB1.

4 Uso de BLAS 3

En esta sección estudiaremos un método de Jacobi por bloques para el PSVP y algunas implementaciones de este método en el Alliant FX/80:

- Un método que deja la explotación de las características del sistema en BLAS 3 y al compilador (B3).
- 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 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 [10]. 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 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 2 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 desbalanceo 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 desbalanceado 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.

Usaremos como base para el diseño de estos algoritmos el esquema por bloques de [10]. Con este esquema la matriz A se divide en filas y columnas de bloques cuadrados. Se trata cada bloque como si fuera un elemento consistiendo el trabajo sobre cada bloque en la anulación de una serie de elementos del bloque, y se hace un barrido sobre estos bloques usando algún orden de Jacobi (aquí se usa el orden par-impar).

Se obtiene un coste por barrido de $4n^3$ flops, mientras que con los algoritmos que usan BLAS 1 el coste es de $3n^3$ flops, pero este coste se obtiene con los algoritmos que usan BLAS 3 con operaciones de nivel 3, lo que hace que se obtenga una reducción en el tiempo de ejecución con respecto al obtenido con una versión del método de Jacobi que usa 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, el cociente entre los tiempos de ejecución usando BLAS 1 y BLAS 3 será

$$\frac{3k_1}{4k_3} \tag{9}$$

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

4.1 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.

En la tabla 11 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 (se muestran los tamaños de las dos matrices a multiplicar). 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 12 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 11 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 11 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 13, donde podemos ver que los buenos resultados de eficiencia que se obtenían en las tablas 11 y 12 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 tal como proponemos.

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 11: Mflops con **dgemm**

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 12: Mflops con **dgemm** seguido de copia de la matriz resultado usando **dcopy**

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 13: Mflops con **dgemm** seguido de copia de la matriz resultado usando **dcopy**

4.2 Resultados experimentales

En este apartado analizaremos los resultados obtenidos con los distintos métodos por bloques implementados (B3, B3C, B3B, B3CB y B3M), comparándolos entre si y con los métodos que usan BLAS 1. Todos los resultados se han obtenido con matrices con números de doble precisión generados aleatoriamente, y los programas se han hecho en 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 14 se muestran los tiempos de ejecución por barrido 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 7), debido a la pobre utilización de los recursos del sistema que se hace cuando se deja todo el trabajo a BLAS 3. Esto refuerza nuestra idea alternativa de explotar el paralelismo dividiendo el trabajo entre los procesadores y usar BLAS 3 en cada uno de ellos.

3. Comparando los resultados con los de la tabla 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 15, 16 y 17 se muestran los tiempos de ejecución por barrido obtenidos con los programas B3C, B3B y B3CB, respectivamente, para los mismos tamaños de sistema y matriz mostrados en la tabla 14. Se utilizan tamaños de bloque entre 8 y 48, variando de 8 en 8, y para cada tamaño de matriz n se utilizan tamaños de bloque s de manera que $2s$ sea divisor de n, para poder usar el esquema por bloques de [10].

Algunas conclusiones que se pueden sacar observando las tablas y comparando con la tabla 14 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 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.

	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 14: Tiempos de ejecución por barrido en segundos de B3

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.

	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 15: Tiempos de ejecución por barrido en segundos de B3C

Para analizar con más detalle como evoluciona el tamaño de bloque óptimo, representamos este tamaño en la tabla 18 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.

	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 16: 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 17: Tiempos de ejecución por barrido en segundos de B3CB

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.

	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 18: Tamaño de bloque óptimo con los diferentes algoritmos que trabajan por bloques

Para comparar los métodos por bloques presentados (B3, B3C, B3B y B3CB) mostramos en la tabla 19 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 el B3 al ser todos los valores de la tabla menores que 1, lo que confirma nuestra hipótesis de trabajo. 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 20 se muestra la media de los cocientes de la tabla 19 para cada uno de los métodos y los distintos tamaños de matriz, y en la tabla 21 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 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 22, siendo estos tiempos los obtenidos con el tamaño de bloque óptimo. Comparando con la tabla 17 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 23 el cociente de los tiempos de B3M en relación a los de PICB1. En esta tabla se observa una mejora relativa de B3M

	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 19: 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 20: 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 21: 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 22: Tiempos de ejecución en segundos y por barrido con 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.

	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 23: Cociente de los tiempos de ejecución de B3M entre PICBB1

En la tabla 24 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 [17] para otros problemas de algebra lineal. La ganancia respecto a PICB1 es mayor que la que se muestra en la tabla 23 (comparar con la tabla 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.

5 Conclusiones

En este trabajo 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 [10] con otros que no trabajan por bloques, obteniéndose unos resultados mucho mejores con los métodos que trabajan por bloques cuando aumentan el tamaño de las matrices y el número de procesadores.

El método propuesto en [10] para Monoprocesadores trabaja también en Multiprocesadores 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 [3], por lo que se ha adecuado dicho método a las características del Multiprocesador en estudio, obteniendo distintas versiones que se diferencian en la manera de distribuir el trabajo entre

	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 24: Mflops obtenidos con B3M con tamaño de bloque óptimo

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 [17] con otros problemas de algebra lineal.

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.

Referencias

[1] 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.

[2] 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.

[3] 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.

[4] *FX/C Programmer's Handbook.* Alliant Computer Systems Corporation, 1989.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Christian H. Bischof. Computing the singular value decomposition on a distributed system of vector processors. *Parallel Computing*, 11:171–186, 1989.

[12] 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.

[13] G. H. Golub y C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989. Segunda Edición.

[14] Franklin T. Luk y Haesun Park. A proof of convergence for two parallel Jacobi SVD algorithms. *IEEE Trans. Computers*, 28(6):806–811, 1989.

[15] Mythili Mantharam y P. J. Eberlein. New Jacobi-sets for parallel computations. *Parallel Computing*, 19:437–454, 1993.

[16] 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.

[17] 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.

PARALELIZACION EN MULTIPROCESADORES POR DIVISION DEL TRABAJO COMO ALTERNATIVA A PARALELIZACION USANDO BLAS: UN ESTUDIO CON EL METODO DE JACOBI PARA EL PROBLEMA SIMETRICO DE VALORES PROPIOS EN EL ALLIANT FX/80

Javier Cuenca, Domingo Giménez, Vicente Hernández,
Antonio M. Vidal, Javier Zapata

TR3-95