Optimización de un programa de Química Cuántica para el cálculo DVR tridimensional: estudio en Origin 2000 *

Domingo Giménez Cánovas†

Resumen

Dentro del proyecto CICYT sobre "Algoritmos paralelos para el cálculo de valores propios de matrices densas" (TIC96-1062-C03-02) pretendíamos abordar una aplicación de la resolución de problemas de valores propios de matrices densas en Química Cuántica. Estudiaremos la posible optimización (en cuanto a reducción del tiempo de ejecución) del cálculo DVR tridimensional en diferentes sistemas de coordenadas: Jacobi ponderadas, hiperesféricas de Jacobi de curvatura variable, Jacobi rotadas, Radau ponderadas, hiperesféricas de Radau de curvatura variable, Radau rotadas, y coordenadas cinemáticas. En todos los casos los problemas pueden ser de grandes dimensiones, por lo que puede ser necesaria su resolución por técnicas out-of-core, paralelas, o incluso out-of-core-paralelas. En este trabajo abordamos la resolución del problema en una Origin 2000, que es en una máquina con espacio de memoria compartido. En trabajos posteriores abordaremos la resolución del mismo problema con técnicas out-of-core y de paso de mensajes.

1 Introducción

En Química Cuántica aparece la resolución de problemas de valores propios en la resolución de ecuaciones de Scrödinger. Esta es la aplicación más clara donde aparecen problemas simétricos de valores propios [1], 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

$$\hat{H}\psi = E\psi \tag{1}$$

donde los números E son los valores propios del operador de energía, que corresponden a las posibles energías del sistema [2, 3, 4].

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 [1]; siendo en la mayoría de los problemas las matrices reales pero habiendo también aplicaciones con matrices complejas [5].

*Parcialmente subvencionado por la Comisión Interministerial de Ciencia y Tecnología, projecto TIC96-1062-C03-02. Los experimentos han sido realizados en parte en el Origin2000 del Centro Europeo de Paralelismo de Barcelona (CEPBA).

†Departamento de Informática: Lenguajes y Sistemas Informáticos. Univ de Murcia. Aptdo 4021. 30001 Murcia, Spain. domingo@dif.um.es. http://morgana.dif.um.es/~domingo

El problema en que nos centramos en este trabajo corresponde al cálculo vibracional para determinar niveles de energía vibracionales fuertemente excitados de moléculas triatómicas. Diversas técnicas de han desarrollado durante los últimos veinte años para resolver este problema [6], siendo uno de los métodos más eficientes el método adiabático DVR, que es válido para distintos tipos de coordenadas. Este método ha sido utilizado por el Grupo de Investigación de Química Cuántica del Departamento de Química-Física de la Universidad de Murcia para obtener el espectro vibracional completo de moléculas triatómicas [7], y se pretende seguir utilizándolo en problemas de alto coste computacional, por lo que es interesante la optimización de la resolución del problema, principalmente en sistemas del tipo de a los que se tiene acceso: el sistema de espacio de memoria compartido Origin2000 del Servicio de Cálculo Científico de la Universdiad de Murcia, y redes de estaciones de trabajo.

En estos problemas se calcula el potencial para distintos sistemas de coordenadas, apareciendo problemas de valores propios de diferentes tamaños pero que son en todos los casos densos, simétricos y reales, y se trata de calcular los valores propios (y los vectores propios a ellos asociados) por debajo de unos ciertos valores, y dependiendo de cuáles sean esos valores puede ser interesante el resolver el problema completo o el reducido. Además del cálculo de los valores propios hay otras partes de la resolución que tienen alto coste computacional, especialmente la formación de un Hamiltoniano a partir de la información obtenida de la resolución de problemas de valores propios de tamaño mediano.

La resolución del problema la consideraremos dividida en tres partes:

- La primera consiste en el `cálculo y diagonalización de las matrices de los operadores coordenada`. En esta parte aparece la resolución de tres problemas de valores propios de dimensión *pequeña* y que corresponden a las dimensiones DVR que se utilizarán. Dado que los problemas son pequeños, la importancia (en tiempo de ejecución) de esta parte en la solución total del problema es mínima, por lo que no la analizaremos.

- La segunda consiste en el `cálculo de los cortes 2D`. En esta segunda parte se resuelven varios problemas de valores propios de dimensión *mediana* y se guarda cierta información de cada uno de los problemas resueltos. La cantidad de problemas a resolver, su dimensión y la cantidad de información a guardar depende de varios parámetros de entrada que detallaremos más adelante. Las dos subpartes dentro de esta parte pueden tener cierto peso en el tiempo total de resolución del problema, variando este peso con los parámetros de entrada.

- En la tercera parte tenemos la resolución de problemas de valores propios de dimensión *grande*, y típicamente es un número reducido de problemas y posiblemente un único problema. Previamente a la resolución del problema de valores propios hay que formar el problema construyendo un Hamiltoniano a partir de la información guardada de la resolución de los problemas de la parte segunda. Las dimensiones de los problemas dependen en este caso también de los parámetros de entrada, y por tanto varía su peso en el tiempo total de resolución.

Empezaremos analizando la importancia de cada una de las partes mencionadas en la resolución del problema utilizando un programa base a partir del cual comenzamos el análisis del problema. Posteriormente estudiaremos la posible reducción en los tiempos de ejecución de cada una de las partes de la resolución, analizando la paralelización en un sistema de espacio de memoria compartido.

2 Análisis experimental del programa base

El trabajo de optimización y paralelización se ha realizado a partir de un programa no optimizado, que analizaremos para pasar posteriormente a su optimización secuencial y paralela.

Analizaremos las cuatro partes más costosas en la resolución del problema:

- La resolución de los problemas de valores propios que aparecen en el cálculo de los cortes 2D. Que llamaremos D2D.
- El almacenamiento de los resultados obtenidos al llevar a cabo las diagonalizaciones. Que llamaremos A2D.
- La formación del Hamiltoniano. Que llamaremos G3D.
- La resolución del problema de valores propios asociado al Hamiltoniano. Que llamaremos D3D.

La importancia de estas cuatro partes en la ejecución total del programa varía al variar algunos parámetros. Los parámetros que hemos variado para comprobar experimentalmente cómo influyen en el tiempo de ejecución y cómo varía la importancia de cada una de las cuatro partes del programa en este tiempo son:

- Las dimensiones DVR de las tres matrices se han tomado las mismas, con los valores 30 y 40. A este parámetro le llamamos DVR.
- El potencial máximo 2D se ha tomado con valores 20000 y 25000. A este parámetro le llamamos PM2D.
- La energía de corte 3D se ha tomado 15000 y 20000. A este parámetro le llamamos EC3D.
- El tamaño de memoria reservada para la resolución de los problemas de valores propios *medianos* y *grandes* viene determinado en función del parámetro NFUNCI, que será la dimensión máxima del problema que se puede resolver. Los valores que se han utilizado son 1200 y 1600.

Por simplificar el análisis no se han variado otros parámetros que pueden influir en la ejecución: la base máxima 2D se ha dejado con valor 800, el número mínimo de funciones 2D por ángulo es 0 y no se utiliza filtro de funciones 2D.

La variación del parámetro del programa NFUNDVR, que se utiliza para reservar memoria en la que trabajar en la resolución de los problemas de valores propios *pequeños*, no influye de una manera apreciable en el tiempo de ejecución.

Además, los experimentos se han realizado con coordenadas hiperesféricas de Jacobi sin desplazamiento, utilizando siempre las mismas funciones de base y los mismos parámetros iniciales para las coordenadas.

En la resolución de los problemas de valores propios se ha utilizado la rutina EIGRS de IMSL, resolviéndose el problema completo.

Los experimentos se han realizado en dos máquinas de características diferentes: una SUN Ultra 1 140 (SUNUltra) y un procesador de una Silicon Graphic Origin (Origin).

Los resultados obtenidos en tiempo de ejecución de cada una de las cuatro partes y su tanto por ciento con respecto al tiempo total se muestran en las tablas 1, para SUNUltra, y 2 para Origin. En SUNUltra se ha usado la opción de compilación `-fast`, y en Origin la opción `-O3`.

Tabla 1: Tiempo de ejecución (en segundos) de cada una de las cuatro partes del programa, y tanto por ciento respecto al tiempo total. En SUNUltra.

Parámetros				Tiempo de ejecución				Tanto por ciento			
				D2D	A2D	G3D	D3D	D2D	A2D	G3D	D3D
DVR=30	PM2D=20000	EC3D=15000	NFUNCI=1200	82	743	384	58	6	59	30	5
DVR=30	PM2D=20000	EC3D=15000	NFUNCI=1600	83	1373	396	61	4	71	20	3
DVR=30	PM2D=20000	EC3D=20000	NFUNCI=1200	114	1087	1571	458	4	34	49	14
DVR=30	PM2D=20000	EC3D=20000	NFUNCI=1600	112	1926	1729	534	2	44	40	12
DVR=30	PM2D=25000	EC3D=15000	NFUNCI=1200	191	845	511	61	12	53	32	4
DVR=30	PM2D=25000	EC3D=15000	NFUNCI=1600	183	1491	499	64	8	66	22	2
DVR=30	PM2D=25000	EC3D=20000	NFUNCI=1200	181	844	1451	333	6	30	52	12
DVR=30	PM2D=25000	EC3D=20000	NFUNCI=1600	183	1491	1675	418	4	39	44	11
DVR=40	PM2D=20000	EC3D=15000	NFUNCI=1200	672	997	1748	276	18	27	47	7
DVR=40	PM2D=20000	EC3D=15000	NFUNCI=1600	721	1904	1855	300	15	39	38	6
DVR=40	PM2D=20000	EC3D=20000	NFUNCI=1200	943	1353	3130	603	15	22	51	10
DVR=40	PM2D=20000	EC3D=20000	NFUNCI=1600	964	2588	5078	1201	9	26	51	12
DVR=40	PM2D=25000	EC3D=15000	NFUNCI=1200	1254	1036	2240	293	26	21	46	6
DVR=40	PM2D=25000	EC3D=15000	NFUNCI=1600	1329	2039	2226	297	22	34	37	5
DVR=40	PM2D=25000	EC3D=20000	NFUNCI=1200	1210	977	2451	346	24	20	49	7
DVR=40	PM2D=25000	EC3D=20000	NFUNCI=1600	1319	2004	4531	884	15	22	51	10

Intentaremos obtener algunas conclusiones a partir de los resultados mostrados en las tablas, pero podemos ver que el número de parámetros es grande y la variación de un parámetro puede afectar en un sentido mientras que la de otro parámetro afecta en otro sentido diferente. Además, en algunos casos pueden aparecer resultados extraños que pueden ser debidos a diferentes condiciones en la ejecución (otros usuarios, gestión del sistema de ficheros, ...).

Analizaremos detalladamente el comportamiento del programa en diferentes casos.

2.1 Importancia de las diferentes partes del programa

El comportamiento en SUNUltra y Origin es distinto, tanto en el tiempo de ejecución que es bastante mayor en SUNUltra debido a las menores prestaciones de esta máquina, como en el porcentaje de tiempo dedicado a cada parte del programa. Aun así, podemos observar algunas características comunes:

- La parte más costosa es G3D en Origin y ésta y A2D en SUNUltra. Las dos partes son en las que no se resuelven problemas de valores propios, por lo que parece que el cálculo de valores propios no es lo más costoso en el programa, y merecería la pena analizar y optimizar también esas partes del programa.

  Empezamos analizando A2D. En ella, una vez calculados los ángulos de corte resolviendo un problema *mediano* de valores propios, se guardan los nuevos valores propios calculados y sus vectores propios asociados, manteniendo el conjunto de los valores propios nuevos y los correspondientes a ángulos anteriores ordenado, perdiéndose los últimos de ellos, pues sólo se guarda hasta un cierto número de valores. Cuando el número mínimo de valores a guardar de cada ángulo no es cero (parámetro N2DCORTEMIN$\neq$0) se realiza una copia de valores y vectores antes de realizar la mezcla antes mencionada. En el caso en que sea cero, como en los experimentos que hemos hecho, esa parte no tiene por qué realizarse. Aun en el caso en que se tenga que realizar se puede hacer al mismo tiempo que la mezcla para

Tabla 2: Tiempo de ejecución (en segundos) de cada una de las cuatro partes del programa, y tanto por ciento respecto al tiempo total. En Origin.

Parámetros				Tiempo de ejecución				Tanto por ciento			
				D2D	A2D	G3D	D3D	D2D	A2D	G3D	D3D
DVR=30	PM2D=20000	EC3D=15000	NFUNCI=1200	21	36	46	14	18	31	39	12
DVR=30	PM2D=20000	EC3D=15000	NFUNCI=1600	37	109	78	26	14	43	31	10
DVR=30	PM2D=20000	EC3D=20000	NFUNCI=1200	23	34	137	113	7	11	45	37
DVR=30	PM2D=20000	EC3D=20000	NFUNCI=1600	36	115	238	260	5	16	36	40
DVR=30	PM2D=25000	EC3D=15000	NFUNCI=1200	70	63	94	27	28	25	37	11
DVR=30	PM2D=25000	EC3D=15000	NFUNCI=1600	80	116	103	29	24	35	31	8
DVR=30	PM2D=25000	EC3D=20000	NFUNCI=1200	73	65	278	225	11	10	43	35
DVR=30	PM2D=25000	EC3D=20000	NFUNCI=1600	80	125	335	242	10	15	42	30
DVR=40	PM2D=20000	EC3D=15000	NFUNCI=1200	270	73	342	184	31	8	39	21
DVR=40	PM2D=20000	EC3D=15000	NFUNCI=1600	264	122	327	149	30	14	37	17
DVR=40	PM2D=20000	EC3D=20000	NFUNCI=1200	271	81	391	228	27	8	40	23
DVR=40	PM2D=20000	EC3D=20000	NFUNCI=1600	261	118	663	518	16	7	42	33
DVR=40	PM2D=25000	EC3D=15000	NFUNCI=1200	555	78	428	184	44	6	34	14
DVR=40	PM2D=25000	EC3D=15000	NFUNCI=1600	514	128	411	156	42	10	33	12
DVR=40	PM2D=25000	EC3D=20000	NFUNCI=1200	549	78	488	226	41	6	36	17
DVR=40	PM2D=25000	EC3D=20000	NFUNCI=1600	501	121	822	507	25	6	42	25

evitar movimientos duplicados de valores y vectores propios. La mezcla se hace con una especie de ordenación por inserción, lo que no es efectivo, por lo que habría que sustituirla por una especie de mezcla.

Analizando G3D podemos observar que se forma una matriz de *gran* dimensión a partir de la información guardada en A2D. Se almacena sólo la parte triangular superior pues la matriz es simétrica. Cada elemento i, j se forma a partir de los ángulos i, j almacenados. Para ángulos que vienen del mismo corte se hace una especie de producto escalar de los vectores asociados pero afectado cada término por un valor dependiente del número cuántico. Esta operación es de nivel 1 y podría vectorizarse. Cuando los valores propios no vienen del mismo corte las longitudes de los vectores propios son distintas y también se hace una especie de producto escalar pero en este caso multiplicando cada elemento del primer vector por un elemento del segundo vector cuya posición no conocemos a priori sino que hay que calcular en función de los números cuánticos y los ángulos, lo que dificulta (si no imposibilita) la vectorización. Además, en todos los casos se calculan senos y cosenos repetidamente, lo que se podría evitar guardándolos en tablas ya que se calculan siempre de un número finito y reducido de posibles valores.

Por lo tanto, podemos pensar que A2D es bastante mejorable pero no así G3D que es la parte más costosa.

- También en la importancia del cálculo de valores propios hay diferencia en las dos máquinas, llegando a ser como mucho de un 32% en SUNUltra y de un 58% en Origin.

  Los programas se han probado con la rutina EIGRS de IMSL sin optimizar por lo que habrá que comprobar si utilizando la rutina optimizada en Origin, o la rutina DSYEVX de LAPACK [8] en SUNUltra y Origin (en esta máquina también la versión paralela), se reduce el tiempo de ejecución. El uso de LAPACK nos puede generar unas necesidades adicionales de memoria.

Tabla 3: Cociente de los tiempos de ejecución entre SUNUltra y Origin en las distintas partes del programa.

Parámetros				D2D	A2D	G3D	D3D
DVR=30	PM2D=20000	EC3D=15000	NFUNCI=1200	3.73	20.64	8.35	4.14
DVR=30	PM2D=20000	EC3D=20000	NFUNCI=1200	4.96	31.97	11.47	4.05
DVR=30	PM2D=25000	EC3D=15000	NFUNCI=1200	2.73	13.41	5.44	2.26
DVR=30	PM2D=25000	EC3D=20000	NFUNCI=1200	2.48	12.98	5.22	1.48
DVR=40	PM2D=20000	EC3D=15000	NFUNCI=1200	2.49	13.66	5.11	1.50
DVR=40	PM2D=25000	EC3D=20000	NFUNCI=1200	2.20	12.53	5.02	1.53

En los experimentos realizados se ha comprobado que, dependiendo de los valores de corte, el número de valores propios a utilizar de cada ángulo de corte es pequeño en relación con el tamaño de las matrices (se calculan entre un 15 y un 35% de los valores propios), por lo que puede ser conveniente resolver problemas parciales de valores propios, con lo que se reducirá el tiempo de ejecución y las necesidades de memoria. Esto se puede hacer con la rutina DSYEVX de LAPACK o con un método de cálculo de un número reducido de valores o vectores propios.

2.2 Comparación de los resultados en SUNUltra y Origin

Como ya hemos mencionado, la proporción de los tiempos de ejecución de las distintas partes del programa varía considerablemente en los dos sistemas usados, lo que puede ser debido a que en las distintas partes del programa el tipo de operaciones que se usan intensivamente es distinta: en la rutina de cálculo de valores propios y en G3D operaciones aritméticas, y en A2D comparaciones y accesos a memoria, y en G3D también accesos a memoria; y puede que los compiladores usados en los dos sistemas generen código con eficiencia distinta para distintos tipos de operaciones. En la tabla 3 mostramos la proporción del tiempo de ejecución de SUNUltra dividido por el de Origin en las distintas partes del programa.

Comentamos algunos aspectos de la tabla:

- La proporción es mucho mayor (alrededor de 13) en A2D que en las otras partes del programa, menor (entre 1.5 y 2.5) en las rutinas de diagonalización, y toma un valor intermedio en G3D (un poco mayor que 5). Esto puede ser debido a que en la Origin los accesos a memoria se hagan de manera más eficiente que en la SUNUltra.

- Que el cálculo de valores propios sólo sea dos veces más rápido en la Origin que en la SUNUltra hace pensar que la rutina de cálculo de valores propios no se está optimizando muy bien en la Origin, por lo que es especialmente interesante la optimización de esta parte del programa en dicha máquina.

- Por otra parte, el que el cociente para G3D dé un valor intermedio, y el constar G3D de operaciones de acceso a memoria y de operaciones aritméticas, hace pensar que el problema sea debido a la diferencia de optimización de los distintos tipos de operaciones por los dos compiladores.

- Cuando aumenta DVR dismimuye el cociente en D2D y D3D, lo que es debido a que aumenta el tamaño de los problemas de valores propios que se resuelven y la rapidez de

Origin respecto a SUNUltra disminuye al aumentar el tamaño del problema de valores propios a resolver.

- En las dos primeras filas de la tabla los cocientes son mayores que en el resto de filas, y esto es debido a que los primeros experimentos se han realizado en la Origin en modo interactivo, mientras que los últimos se han realizado con el comando `nice +10` debido a su tiempo de ejecución y a las normas de utilización del sistema. Por tanto, en cuanto a comparación de la velocidad real de los sistemas las dos primeras filas dan resultados más realistas.

2.3 Variación de las dimensiones DVR

Al aumentar DVR aumenta el número de ángulos de corte y la dimensión de los problemas de valores propios asociados a esos ángulos y al Hamiltoniano. Esto se puede ver en la tabla 4, donde se muestra el tamaño de los distintos ángulos y el tiempo invertido en su diagonalización y en A2D, en SUNUltra y Origin, cuando DVR=30 y DVR=40 y manteniendo NFUNCI=1200, PM2D=20000 y EC3D=15000. El tiempo invertido en la generación y diagonalización del Hamiltoniano aparece en las tablas 1 y 2, y el aumento de tiempos al pasar de DVR=30 a DVR=40 es debido a que la dimensión del Hamiltoniano es 703 en el primer caso y 1122 en el segundo.

Los tamaños de los ángulos y el Hamiltoniano aumentan al aumentar DVR pero no hay una relación directa entre ellos. También aumenta el número de ángulos de corte, que es siempre DVR/2. Por esto aumenta la importancia de la resolución de los problemas de valores propios.

Vemos que el tamaño del Hamiltoniano es mayor que el de los problemas asociados a los ángulos, pero la diferencia no es tan grande como para que sea siempre D3D más importante que D2D.

El coste de A2D disminuye al aumentar el ángulo, lo que puede ser lógico pues al añadir nuevos ángulos la cantidad de valores y vectores propios a insertar (con una ordenación por inserción) disminuye, pero por otro lado es extraño que para el primer ángulo el coste sea tan grande, ya que no habría trabajo de inserción. Esto sugiere, de nuevo, que esta parte del programa puede ser fácilmente mejorable.

2.4 Variación del potencial máximo 2D

En las tablas 1 y 2 podemos observar que aumentando el valor de PM2D manteniendo los demás parámetros fijos, normalmente aumentan los tiempos de ejecución de todas las partes del programa (no siempre es así debido a las diferentes condiciones de ejecución), pero el incremento en el tiempo de ejecución es considerablemente mayor en D2D. Esto es debido a que al aumentar PM2D aumentan apreciablemente los tamaños de los problemas de valores propios a resolver en el corte de los ángulos, y hay un aumento minúsculo en el tamaño del Hamiltoniano. Esto se puede comprobar en la tabla 5, donde se muestran los tamaños de los problemas asociados a los ángulos de corte y al Hamiltoniano cuando DVR=30 y 40 y PM2D=20000 y 25000, y manteniendo los demás parámetros constantes.

Vemos que con DVR=40 y PM2D=25000 hay varios ángulos para los que el problema es de tamaño 800, lo que es debido al valor de un parámetro que impone ese máximo. Si ese parámetro tuviera un valor mayor los tamaños de los ángulos serían todavía más grandes.

Tabla 4: Tiempos de ejecución (en segundos) y tamaños de los problemas para los distintos ángulos de corte, con NFUNCI=1200, PM2D=20000, EC3D=15000, y tomando DVR los valores 30 y 40. En SUNUltra y Origin.

DVR=30					
		SUNUltra		Origin	
ángulo	tamaño	D2D	A2D	D2D	A2D
0	254	2	107	0	6
1	261	3	96	0	5
2	276	3	87	0	4
3	298	3	77	2	3
4	355	7	68	2	3
5	424	12	60	3	2
6	417	10	48	3	2
7	401	10	39	2	2
8	372	8	33	2	2
9	345	6	28	1	2
10	319	4	25	1	1
11	299	4	21	1	1
12	288	4	19	1	1
13	276	3	18	1	1
14	267	3	17	1	1

DVR=40					
		SUNUltra		Origin	
ángulo	tamaño	D2D	A2D	D2D	A2D
0	450	15	183	5	16
1	455	15	141	6	10
2	467	16	101	6	7
3	482	17	78	8	5
4	517	22	65	9	4
5	568	31	56	11	4
6	683	56	51	22	4
7	744	73	44	33	3
8	733	70	38	31	2
9	716	64	34	25	3
10	684	56	30	22	2
11	647	47	27	19	2
12	611	38	25	15	2
13	577	32	22	12	2
14	546	27	20	10	1
15	523	23	19	8	2
16	496	20	17	7	1
17	481	18	16	8	1
18	473	16	16	7	1
19	469	16	14	6	1

Tabla 5: Tamaños de los problemas para los distintos ángulos de corte, con NFUNCI=1200 y EC3D=15000, y tomando DVR los valores 30 y 40 y PM2D los valores 20000 y 25000.

DVR=	30		40	
PM2D=	20000	25000	20000	25000
0	254	299	450	529
1	261	308	455	538
2	276	325	467	555
3	298	359	482	579
4	355	467	517	620
5	424	556	568	728
6	417	543	683	800
7	401	512	744	800
8	372	480	733	800
9	345	449	716	800
10	319	416	684	800
11	299	386	647	800
12	288	358	611	799
13	276	351	577	753
14	267	347	546	708
15			523	682
16			496	650
17			481	625
18			473	611
19			469	606
Ham	703	705	1122	1128

2.5 Variación de la energía de corte 3D

Al aumentar el valor de la energía de corte 3D aumenta considerablemente el tamaño del Hamiltoniano, mientras que permanece el número de ángulos de corte y el tamaño de los problemas de valores propios asociados. Esto produce un aumento en el tiempo de ejecución en G3D, y un aumento aun mayor en D3D. En las tablas 1 y 2 esto se aprecia más para DVR=30 que para DVR=40, debido a que para DVR=40 el tamaño que tendría el Hamiltoniano es mayor que el límite impuesto con NFUNCI (1200 o 1600) y por tanto el problema de valores propios que se resuelve está limitado por ese valor. Para tener una idea de cómo varía el tamaño del Hamiltoniano vemos algunos casos:

- Con DVR=30, PM2D=20000 y EC3D=15000, el tamaño es 703, y variando EC3D a 20000 el tamaño pasa a ser 1257.
- Con DVR=30 y PM2D=25000, los tamaños para EC2D=15000 y 20000 son, respectivamente, 705 y 1279.
- Con DVR=40 y PM2D=20000, los tamaños son 1122 y 1600 (este último limitado por NFUNCI=1600).
- Con DVR=40 y PM2D=20000, los tamaños son 1128 y 1600 (limitado por NFUNCI=1600).

2.6 Variación del tamaño de memoria reservada para los problemas de valores propios

Variando el valor del parámetro NFUNCI varía normalmente el tiempo de ejecución de todas las partes del programa, aumentando al aumentar NFUNCI. Los tiempos que más aumentan son los de G3D y D3D pues, como hemos visto, NFUNCI impone un límite en el tamaño del Hamiltoniano a tratar. También en D2D y A2D hay un aumento en el tiempo de ejecución que se debe a la gestión de más memoria. Esto se podría evitar haciendo asignación dinámica de memoria. En A2D el aumento es mayor pues en la ordenación por inserción que se hace para mantener los valores y vectores propios ordenados se van copiando los vectores hasta la dimensión dada por NFUNCI, lo cual es innecesario pues conocemos el tamaño de los vectores y este suele ser mucho menor que NFUNCI.

3 Modificaciones al programa base

Dado que, como hemos visto, partimos de un programa base secuencial susceptible de ser mejorado en algunos puntos, antes de pasar a analizar la paralelización de las distintas partes del programa realizaremos algunas modificaciones al programa base para intentar mejorarlo.

3.1 Modificaciones en A2D

Después de calcular los valores y vectores propios de cada ángulo de corte 2D, los valores menores y sus vectores propios asociados se almacenan en una estructura donde se mantienen los valores propios ordenados y se ordenan del mismo modo los vectores propios y los números cuánticos asociados. Los valores se mantienen ordenados por inserción, moviendo toda la información una posición hasta encontrar la posición donde insertar los nuevos datos. Esto representa un coste importante, del orden de n^2m con n el número de valores a insertar y m la dimensión de los vectores propios, y el tiempo de ejecución puede disminuir fácilmente si no se mueve toda la información no relevante (zonas de valores y vectores donde no hay datos obtenidos de los ángulos). De esta manera se obtiene una reducción en el tiempo de ejecución en esta parte del programa, como se puede ver en la tabla 6 donde se muestra el cociente de los tiempos de ejecución en A2D entre el tiempo mejorado y el inicial. Se observa una reducción de entre 31 y 42 por ciento en SUNUltra y entre 28 y 72 por ciento en Origin. La reducción se obtiene en los ángulos iniciales principalmente, pues es en estos casos cuando nos encontramos en las estructuras más información no correspondiente a ángulos tratados. Esto se observa en la tabla 7, donde se comparan los tiempos de A2D en los cinco primeros ángulos para el caso NFUNCI=1200, DVR=40, PM2D=25000 y EC3D=20000.

Obviamente, se podría reducir más el tiempo de ejecución de A2D si no se mueven los vectores sino que asociados a los valores tenemos una lista de índices apuntando a donde están los vectores (reducción a n^2), pero esto conllevaría más modificaciones en el programa en la parte de formación del Hamiltoniano.

3.2 Modificaciones en el cálculo de los valores propios

3.2.1 Utilización de distintas rutinas

Sustituir la rutina de cálculo de valores propios de IMSL por la rutina correspondiente de LAPACK (`dsyevx` cuando se trabaja con la rutina completa y `dspevx` cuando se trabaja con almacenamiento compacto) no produce reducción considerable en el tiempo de ejecución. Ya

Tabla 6: Cociente entre los tiempos de ejecución de A2D mejorado e inicial.

Parámetros				SUNUltra	Origin
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=15000	0.65	0.57
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=20000	0.60	0.63
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=15000	0.60	0.59
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=20000	0.62	0.49
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=15000	0.66	0.62
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=20000	0.67	0.68
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=15000	0.67	0.61
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=20000	0.70	0.72
NFUNCI=1600	DVR=30	PM2D=20000	EC3D=15000	0.61	0.31
NFUNCI=1600	DVR=30	PM2D=20000	EC3D=20000	0.58	0.27
NFUNCI=1600	DVR=30	PM2D=25000	EC3D=15000	0.62	0.29
NFUNCI=1600	DVR=30	PM2D=25000	EC3D=20000	0.60	0.28
NFUNCI=1600	DVR=40	PM2D=20000	EC3D=15000	0.69	0.33
NFUNCI=1600	DVR=40	PM2D=20000	EC3D=20000	0.68	0.33
NFUNCI=1600	DVR=40	PM2D=25000	EC3D=15000	0.69	0.34
NFUNCI=1600	DVR=40	PM2D=25000	EC3D=20000	0.68	0.35

Tabla 7: Tiempos de ejecución de A2D en los cinco primeros ángulos, con el programa base y el A2D mejorado (NFUNCI=1200, DVR=40, PM2D=25000 y EC3D=20000).

SUNUltra		Origin	
A2D base	A2D mejorado	A2D base	A2D mejorado
194.58	0.35	11.91	0.12
141.07	55.43	6.82	1.65
96.94	92.32	4.99	5.01
74.48	73.41	3.82	4.10
61.68	60.72	3.34	3.33

que las dos rutinas trabajan reduciendo la matriz a forma tridiagonal y resolviendo el problema reducido, podemos pensar que la diferencia es debida a pequeños detalles de programación.

En la tabla 8 comparamos el tiempo de ejecución en D2D para distintas condiciones y utilizando la rutina EIGRS de IMSL y las rutinas `dsyevx` y `dspevx` de LAPACK. Los tiempos que se muestran no son tiempos de reloj sino tiempos "elapsed", por lo que se pueden apreciar diferencias con respecto a tablas anteriores. Se comprueba que `dspevx` da los peores tiempos de ejecución, mientras que no hay diferencia apreciable entre EIGRS y `dsyevx`, aunque parece que para problemas con un mayor coste la tendencia es que sea preferible usar `dsyevx`.

El inconveniente con `dsyevx` es que no trabaja con la matriz almacenada de manera compacta, por lo que podemos tener problemas de memoria antes si utilizamos esta rutina.

Tabla 8: Tiempo de ejecución de D2D resuelto con distintas rutinas.

				SUNUltra			Origin		
Parámetros				EIGRS	dsyevx	dspevx	EIGRS	dsyevx	dspevx
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=15000	**78.49**	96.62	109.79	**22.37**	24.10	25.75
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=20000	**81.26**	95.89	109.11	**22.37**	24.21	25.81
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=15000	**182.98**	202.18	230.13	48.56	**48.29**	50.64
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=20000	**180.71**	202.52	233.78	**48.23**	48.59	50.82
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=15000	**666.57**	692.08	807.67	178.49	**176.33**	187.16
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=20000	**677.45**	691.86	804.14	182.79	**169.15**	192.16
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=15000	1212.36	**1197.71**	1417.16	365.94	**339.44**	362.91
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=20000	1203.07	**1196.25**	1446.64	360.85	**345.16**	368.18

Tabla 9: Tiempo de ejecución de las distintas partes del programa resolviendo problemas completos de valores propios o problemas parciales.

				completo				parcial			
Parámetros				D2D	A2D	G3D	D3D	D2D	A2D	G3D	D3D
SUNUltra											
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=15000	96.62	452.87	385.27	56.26	21.44	46.78	389.27	17.57
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=15000	202.18	491.18	496.71	56.90	43.66	47.28	498.37	20.18
Origin											
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=15000	24.10	24.05	47.78	18.59	5.19	1.75	46.55	3.58
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=20000	24.21	24.00	142.36	168.02	6.81	6.30	137.95	20.70
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=15000	48.29	26.60	64.09	18.75	9.73	1.85	62.11	4.42
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=20000	48.59	25.88	189.27	160.87	12.02	6.50	182.60	26.41
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=15000	176.33	34.01	233.56	130.88	30.68	5.33	224.33	13.24
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=20000	169.15	32.92	265.03	159.95	36.76	13.60	258.95	18.30
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=15000	339.44	36.64	293.73	138.22	50.32	5.49	284.40	16.16
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=20000	345.16	36.52	333.08	166.91	58.02	13.93	324.07	20.42

3.2.2 Resolución de problema parcial

Dado que no es necesario resolver los problemas completos de valores propios en D2D y D3D, sino que tenemos unos valores por debajo de los cuales nos interesan los valores propios (PM2D para los problemas en D2D, y EC3D en D3D), se puede usar la rutina `dsyevx` indicando que los valores propios que se quiere están entre 0 y la cota superior (PM2D o EC3D). En caso de no saber que los valores propios son todos positivos podríamos utilizar el círculo de Gershgorin [9] para obtener una cota inferior.

En la tabla 9 se comparan los tiempos de ejecución en las distintas partes del programa cuando se utiliza la rutina `dsyevx` calculando todos los valores propios y almacenándolos todos con A2D mejorado, y cuando se calculan y se almacenan los valores y vectores propios por debajo de las cotas superiores antes mencionadas. En el caso de la SUNUltra se muestran menos valores pues para los que faltan las mediciones de tiempos dan valores extraños, lo que puede ser debido a problemas de memoria. En Origin disponemos de más memoria y los resultados son más representativos.

Se observa que hay una gran reducción en el tiempo de ejecución en las partes de cálculo de valores propios (D2D y D3D), siendo entre 3 y 9 veces más rápido resolver el problema parcial. Resolver problemas parciales y por tanto almacenar sólo la información obtenida para valores por debajo de PM2D hace que se obtenga una reducción importante en el tiempo de A2D, siendo en este caso entre 3 y 10 veces más rápido, y quedando esta parte con un tiempo de ejecución muy pequeño en comparación con las demás partes del programa. También se produce una reducción mínima en G3D, que queda claramente como la parte más costosa del programa.

Tabla 10: Comparación de los tiempos de ejecución con las distintas maneras de calcular el Hamiltoniano.

Parámetros				prog base	versión 1	versión 2	versión 3
SUNUltra							
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=15000	389.27	91.47	91.74	91.64
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=15000	498.37	121.64	118.16	120.17
Origin							
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=15000	46.55	24.57	35.00	24.74
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=20000	137.95	73.22	104.39	73.41
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=15000	62.11	30.03	45.45	33.91
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=20000	182.60	101.77	135.34	100.43
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=15000	224.33	127.73		124.99
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=20000	258.95	144.07		145.91
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=15000	284.40	164.60		160.59
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=20000	324.07	185.86		187.18

3.3 Modificaciones en la generación del Hamiltoniano

En la generación del Hamiltoniano se usan los valores y vectores propios obtenidos, calculando para cada par de valores propios una entrada de la matriz. Como la matriz es simétrica se calculan $n\frac{n+1}{2}$ entradas con n el número de valores propios almacenados (máximo NFUNCI). Para cada par, dependiendo de que correspondan o no al mismo corte, el trabajo que se realiza es distinto, pero proporcional al tamaño de los vectores (m), por lo que el coste es del orden $\frac{n^2}{2}m$. El número de operaciones en coma flotante en casa paso es de 12 y se llama a las funciones DCOS y DSIN, y cuando los ángulos son distintos también a las IFIX y SGNL.

Si no se llama a las funciones DCOS y DSIN sino que se almacenan los valores de los senos y cosenos que hay que obtener repetidamente se ahorra tiempo de ejecución, tal como se ve en la tabla 10, donde se compara el tiempo de ejecución en la formación del Hamiltoniano calculando los senos y cosenos (columna primera) y teniendo estos almacenados (columna segunda). La memoria adicional para almacenarlos es de 480 números en doble precisión, lo que es poco en comparación con la memoria total del programa.

Se puede pensar en evitar del mismo modo las llamadas repetidas a IFIX y SGNL almacenando estos valores en un array tridimensional, pero en este caso se necesita una cantidad considerable de memoria (512000 enteros) y el tiempo de ejecución no mejora con respecto a la mejora anterior, e incluso empeora en Origin (tabla 10, tercera columna, con senos y cosenos precalculados y con IFIX(SGNL(POTENX())) precalculado).

El cálculo para cada par de valores finaliza con unas fórmulas que se pueden modificar para realizar menos accesos a memoria y menos operaciones, pero como esta parte es de menor orden que el trabajo realizado con los vectores, su reestructuración no influye sustancialmente en el tiempo de ejecución (tabla 10, columna cuatro, con senos y cosenos precalculados y las fórmulas finales reestructuradas).

3.4 Reducción en el tiempo de ejecución

Para dar una idea de la reducción conseguida hasta ahora en el tiempo de ejecución comparamos en la tabla 11 los tiempos obtenidos en las distintas partes del programa con valores de

Tabla 11: Comparación de los tiempos de ejecución del programa inicial y el mejorado en Origin, con NFUNCI=1200, DVR=40, PM2D=25000 y EC3D=20000.

	D2D	A2D	G3D	D3D	Total
tiempo "elapsed"					
prog base	360.85	50.62	321.30	148.78	881.55
prog mejorado	58.02	13.93	185.86	20.47	278.28
mejorado/base	0.16	0.28	0.58	0.14	0.32
tiempo de reloj					
prog base	549	78	488	226	1341
prog mejorado	90	18	251	37	396
mejorado/base	0.16	0.23	0.51	0.16	0.30

los parámetros NFUNCI=1200, DVR=40, PM2D=25000 y EC3D=20000, en Origin, con el programa base inicial y con la versión con A2D modificada y resolviendo problemas parciales de valores propios. Se muestran tiempos "elapsed" y de reloj, y el cociente entre la versión mejorada y el programa base. Se observa una reducción de un 70% en el tiempo de ejecución, lo que representa unos quince minutos.

4 Paralelización en Memoria Compartida

4.1 Sistemas

El programa se intentará modificar para obtener un programa paralelo para máquinas de Espacio de Memoria Compartido. Los sistemas sobre los que se trabaja son:

- Un Silicon Graphic Power Challenge con 2 procesadores, del Servicio de Cálculo Científico de la Universidad de Murcia.
- Un Silicon Graphic Origin 2000 con 6 procesadores, del Servicio de Cálculo Científico de la Universidad de Murcia.
- Un Silicon Graphic Origin 2000 con 32 procesadores, del Centro Europeo de Paralelismo de Barcelona.

La forma de programación es la misma para los tres equipos, pero las prestaciones que se pueden obtener de ellos diferentes.

Los sistemas proporcionan la posibilidad de paralelización automática, utilizando un compilador paralelizante, o explícita, por medio de directivas de multiprocesamiento [10, 11].

Cada procesador del Challenge tiene unas prestaciones teóricas máximas de 300 Mflops, y cada uno de los Origin de 500 Mflops [12]. Sin embargo, no son estas prestaciones máximas las que determinan principalmente las prestaciones que se pueden obtener con un programa paralelo en estos sistemas, sino que las prestaciones obtenibles vienen determinadas en gran manera por la forma en que el sistema gestiona los `threads`. En los equipos del Servicio de Cálculo Científico de la Universidad de Murcia no se asegura en la actualidad que se utilicen los recursos en modo exclusivo, por lo que cuando se pone en marcha un trabajo que utiliza varios threads estos entran en competencia con los de otros usuarios, resultado que una ejecución paralela puede ser mucho más lenta que una secuencial porque unos threads deben esperar a que se asignen recursos a otros. Esto se nota principalmente en la Origin, pues esta máquina es la que está siendo masivamente utilizada, mientras que la Challenge es poco utilizada en la actualidad (por más antigua y más lenta). En la máquina del Centro de Paralelismo de Barcelona hay varias colas de distintas características en cuanto a cantidad de memoria, número de procesadores, ejecución secuencial y paralela, y asignación de los procesadores en modo exclusivo o compartido. Por esto, para estudiar el paralelismo utilizaremos principalmente el equipo del CEPBA, aunque las conclusiones serían extrapolables al Origin del Servicio de Cálculo Científico si se pudiera disponer de procesadores en él de modo exclusivo.

4.1.1 Paralelización automática

Los sistemas antes mencionados proporcionan un compilador paralelizante al que se puede llamar incluyendo la opción `-pfa` cuando se compila con fortran77 o `-pca` cuando se utiliza C. Antes de ejecutar el programa paralelo obtenido se asigna a la variable de entorno `MP_SET_NUMTHREADS` el número de threads que se asume que se ponen en marcha en una ejecución paralela.

En [10] se analiza el comportamiento de la multiplicación de matrices en un Power Challenge, obteniéndose que para tamaño de matriz 1000 la ejecución con cuatro procesadores es 3.61 veces más rápida que la secuencial en el caso de no haber optimizado los códigos. Se podría pensar que la paralelización automática nos va a proporcionar buenas prestaciones en cuanto a eficiencia y speed-up, pero no es así pues si se realiza la compilación optimizando el código con la opción `-O3` resulta que se divide por 24.24 el tiempo de ejecución del programa secuencial, y sólo por 3.45 el del paralelo, siendo la ejecución secuencial casi el doble de rápida que la paralela en 4 procesadores.

Esto nos da idea de que la paralelización automática difícilmente nos va a proporcionar buenas prestaciones en un programa cualquiera ya que no se obtienen ni siquiera en el caso de la multiplicación matricial que es una operación bastante sencilla.

4.1.2 Rutinas paralelas

En el programa que nos ocupa se resuelven problemas de valores propios utilizando rutinas de IMSL o LAPACK. En los equipos en que trabajamos se dispone de rutinas optimizadas para dichos equipos, tanto secuenciales, en la librería `complib.sgimath`, como paralelas, en la librería `complib.sgimath_mp`. Para resolver los problemas de valores propios utilizamos la rutina `dsyevx`, pudiendo tomar la de la librería secuencial o la de la paralela cuando se genera código paralelo.

La utilización de la librería paralela permite que la rutina correspondiente se ejecute en paralelo y se pueda producir una reducción importante en el tiempo de ejecución. En [10] se comparan los tiempos obtenidos en una multiplicación de matrices de dimensión 1000 cuando se optimiza el código (`-O3`) y se ejecuta en un procesador o cuatro de un Power Challenge, utilizando la rutina `gemm` de BLAS y no utilizando la rutina. Los tiempos en segundos se muestran en la tabla 12. En la tabla 13 se muestran los tiempos obtenidos en una Origin 2000 variando el número de procesadores. En la Origin se han ejecutado los trabajos en una cola compartida, pero no se observan grandes diferencias con los obtenidos en una cola exclusiva.

Se observa que el uso de las rutinas produce una reducción en el tiempo de ejecución en el Power Challenge, pero no así en el Origin 2000, donde se produce una reducción en el tiempo de ejecución con el uso de la librería en un procesador, pero esta reducción no aparece en el caso paralelo. Estos resultados nos sugieren que para reducir el tiempo de ejecución en nuestro programa puede ser preferible el desarrollo de algoritmos paralelos de cálculo de valores propios

Tabla 12: Tiempo de ejecución, en segundos, en la multiplicación de matrices de tamaño 1000, en Power Challenge.

	Procesadores	
	1	4
sin librería	10.62	20.70
con librería	9.69	3.22

Tabla 13: Tiempo de ejecución, en segundos, en la multiplicación de matrices de tamaño 1000, en Origin 2000.

	Procesadores					
	1	2	3	4	5	6
sin librería	6.67	4.01	3.45	2.56	4.85	3.40
con librería	5.26	4.52	3.70	3.48	3.19	3.73

adecuados a la arquitectura que nos ocupa, más que usar directamente las rutinas que se nos proporcionan. Algunos intentos se están haciendo en este sentido [13].

En cuanto al paralelismo obtenido en los dos sistemas, vemos que en el Power Challenge se llega a tener un speed-up de 3.01 con cuatro procesadores cuando se utiliza la función de librería, mientras que en el Origin 2000 el speed-up que se obtiene con cuatro procesadores es de 2.60 si no se usa la librería, y de 1.51 si sí se usa. El speed-up alcanzado es muy reducido en el caso del Origin 2000, y con la rutina `dsyevx`, que es la que usamos en nuestro programa, los resultados serán todavía peores.

4.1.3 Resultados experimentales

Para empezar analizando las posibilidades de paralelización en nuestro programa hemos realizado algunos experimentos con paralelización automática y usando las librerías paralelas. En la tabla 14 se muestran los resultados de algunos experimentos. La tabla muestra tiempos de ejecución con 1, 2, 3 y 4 procesadores en las distintas partes del programa (D2D, A2D, G3D y D3D) con el programa optimizado y con la versión que usa la rutina de IMSL y calcula todos los valores y vectores propios. Los tiempos se han obtenido en la Origin 2000 del CEPBA mandando los trabajos a una cola no exclusiva. Se han realizado algunos experimentos mandando los trabajos a una cola exclusiva, pero los resultados son similares a los mostrados, no habiendo una mejora clara en los tiempos de ejecución, y obteniéndose las misma conclusiones que se pueden obtener de la tabla 14.

Vemos que en todos los casos, en la resolución de los problemas de valores propios no se produce una reducción clara en el tiempo de ejecución, tanto si se usa la librería paralela como si se aplica paralelización automática a la rutina de IMSL, ni con problemas con un tiempo de ejecución reducido ni cuando los problemas son mayores y el tiempo de ejecución aumenta. Esto nos lleva a considerar que para reducir el tiempo de ejecución en D2D y D3D deberemos rediseñar los algoritmos usados (que trabajan reduciendo la matriz a forma condensada y resolviendo el problema de valores propios reducido) para adecuarlos a la arquitectura del sistema con que estamos trabajando, utilizar otros métodos que son normalmente peores pero que pueden ser adecuados para nuestro problema por ser más fácilmente paralelizables o por

Tabla 14: Tiempo de ejecución, en segundos, en las diferentes partes del programa, con paralelización automática y uso de la rutina `dsyevx` de la librería secuencial y paralela resolviendo problemas parciales de valores propios, y con paralelización automática y rutina de IMSL para resolver los problemas completos de valores propios. En Origin 2000.

Problemas parciales de valores propios DVR=30,PM2D=20000,EC3D=15000,NFUNCI=1200				
Procesadores:	1	2	3	4
D2D	**5.15**	5.30	5.48	6.92
A2D	1.72	0.80	**0.69**	0.93
G3D	24.64	17.69	**16.09**	17.60
D3D	3.62	3.43	**3.39**	3.60
Total	36.26	29.01	**26.73**	30.41
Problemas parciales de valores propios DVR=40,PM2D=25000,EC3D=20000,NFUNCI=1200				
Procesadores:	1	2	3	4
D2D	60.19	**57.16**	64.62	60.63
A2D	13.15	5.22	4.15	**2.80**
G3D	183.50	122.12	99.36	**86.16**
D3D	**19.79**	19.92	22.06	19.88
Total	283.30	208.04	195.05	**175.09**
Problemas completos de valores propios DVR=30,PM2D=20000,EC3D=15000,NFUNCI=1200				
Procesadores:	1	2	3	4
D2D	22.35	**19.70**	29.18	35.70
A2D	25.00	6.52	4.66	**4.50**
G3D	49.60	27.60	**20.93**	20.95
D3D	14.41	14.12	**13.81**	18.64
Total	113.31	70.18	**70.09**	82.31

trabajar particularmente bien para el tipo de matrices que se nos presentan, o resolver en D2D varios problemas de valores propios a la vez, cada uno en secuencial en un procesador distinto.

En A2D se aprecia un comportamiento curioso, al obtenerse speed-up superlineal. Esto puede ser debido a que las operaciones en esta parte del programa son de acceso a memoria y puede que la organización de la memoria nos permita el acceso concurrente a zonas distintas.

En G3D se obtiene una reducción en el tiempo no demasiado grande, con un speed-up entre 1.5 y 2.3. Dado que esta parte es la más costosa del programa y que queda margen para obtener un speed-up mayor, merece la pena dedicar esfuerzos a su optimización.

Una vez analizado el comportamiento del programa con paralelización automática, pasaremos a ver en las subsecciones siguientes algunas posibilidades de paralelización explícita en cada una de las partes del programa.

4.2 Paralelización de la generación del Hamiltoniano

La generación del Hamiltoniano se hace formando la parte triangular inferior de la matriz utilizando los vectores propios almacenados en la parte anterior de la ejecución. Se genera el

Hamiltoniano utilizando dos bucles, el primero para recorrer las filas de la matriz y el segundo para las columnas (sólo la parte triangular inferior), y para cada elemento se recorren uno por uno los elementos de dos de los vectores propios almacenados, que pueden corresponder a la misma o a distinta matriz. El esquema del código es:

```
for i = 1 to max
    for j = 1 to i
        for k = 1 to dim(vector)
            computación
        endfor
    endfor
endfor
```

Según este esquema el coste es de orden $O\left(n^2 d\right)$, siendo n la dimensión del Hamiltoniano y d la de los vectores, que no es siempre la misma. La parte de computación consta de una serie de accesos a memoria y unas doce operaciones en coma flotante.

La paralelización automática sobre el programa inicial sólo ha paralelizado el más interno de los bucles, lo que hace que las prestaciones no sean demasiado buenas al crearse threads $\frac{1+max}{2}max$ veces para trabajar con datos de tamaño $dim(vector)$. No se han paralelizado los bucles externos porque había una dependencia de datos al utilizarse una variable que se actualizaba en cada pasada por el bucle y que indicaba la posición de memoria del elemento del Hamiltoniano que se estaba generando. Esta dependencia se puede evitar calculando la posición en función de i y j.

Otra posibilidad es utilizar paralelización explícita paralelizando el bucle más externo, pero en este caso se produce desbalanceo ya que no todos los pasos por el bucle tienen la misma cantidad de computación.

Otra posible mejora puede consistir en cambiar el orden de los índices para que se acceda por columnas, ya que es así como se almacenan las matrices en Fortran. De este modo los dos primeros bucles quedarían:

```
for j = 1 to max
    for i = j to max
```

Con paralelización explícita se han realizado experimentos en los siguientes casos:

- Paralelizando el bucle en i con una asignación de los índices a los threads por bloques. De este modo habrá desbalanceo pues se está generando una matriz triangular inferior. A esta versión le llamaremos BLOQ.

- Paralelizando el bucle i con una asignación cíclica a los threads. Habrá menos desbalanceo que en el caso anterior, pero el que los vectores tengan distintas dimensiones nos introduce un desbalanceo imprevisible. A esta versión le llamamos CICL.

- Realizando acceso por columnas (cambiando el orden de los índices) y con una asignación cíclica. Le llamaremos COL-CIC.

- Con acceso por columnas y con asignación dinámica de los índices a los threads. En este caso, se crea un determinado número de threads y se asigna a cada uno el trabajo correspondiente a un j, conforme los threads van acabando el trabajo que tienen asignado toman un nuevo índice. De esta manera se evita el desbalanceo, pero se genera un coste adicional de asignación de índices a los threads. Se le ha llamado COL-DIN.

En la tabla 15 se comparan los tiempos de ejecución en G3D en Origin 2000 para distinto número de procesadores y los distintos tipos de paralelización.

Tabla 15: Tiempo de ejecución, en segundos, en G3D con distintos tipos de paralelización. En Origin 2000.

DVR=30,PM2D=20000,EC3D=15000,NFUNCI=1200					
Proc.	-pfa	BLOQ	CICL	COL-CIC	COL-DIN
1	24.64	26.44	26.00	26.22	24.75
2	17.69	18.73	19.98	18.80	12.87
3	16.09	13.98	13.62	13.88	8.80
4	17.60	11.51	10.99	12.13	6.35
5		9.17	9.55	8.85	5.40
6		8.13	8.32	8.07	4.29
7		7.03	6.76	6.55	3.63
8		6.25	6.27	6.20	3.32
DVR=40,PM2D=25000,EC3D=20000,NFUNCI=1200					
Proc.	-pfa	BLOQ	CICL	COL-CIC	COL-DIN
1	183.50	193.34	202.35	185.64	196.25
2	122.12	148.10	172.35	147.56	95.28
3	99.36	140.35	114.08	104.43	69.89
4	86.16	92.39	88.43	81.01	52.08
5		77.10	79.63	71.78	43.21
6		73.49	65.87	69.66	33.57
7		54.19	57.28	57.02	30.35
8		50.53	51.08	48.10	26.84

Debido a que los tamaños de los vectores no son fijos, distintas asignaciones pueden dar lugar a balanceos distintos. Esto, junto con el hecho de que el sistema no siempre tiene la misma carga, hace que sea difícil obtener una conclusiones significativas. Las conclusiones más claras pueden ser:

- Los mejores resultados se obtienen con una asignación dinámica del trabajo.
- Comparando CICL con COL-CIC vemos que hay una ligera mejora cuando se accede por columnas y se trabaja con un volumen grande de datos. Cuando el volumen de datos es pequeño o cuando el número de procesadores aumenta, y por tanto disminuye la cantidad de datos por procesador, esa mejora no es tan significativa, lo que puede ser debido a que los datos quepan en las zonas más bajas de la jerarquía de memoria y el coste de acceso por filas y columnas no difiera sustancialmente.
- Hay diferencia en las ejecuciones secuenciales que pueden ser debidas a variaciones en las condiciones del sistema al hacer la ejecución, a los diferentes patrones de acceso a los datos (filas o columnas) o a la gestión de la asignación de los trabajos (COL-CIC).
- La paralelización automática da unos pobres resultados con un volumen pequeño de datos, pero con un volumen mayor llega a mejorar la paralelización explícita si esta no se optimiza. Esto es debido a que la paralalización implícita paraleliza el bucle más interno de una forma balanceada, mientras que la paralelización en el bucle más externo no nos asegura el balanceo salvo en el caso de COL-CIC.

Tabla 16: Eficiencia en G3D con distintos tipos de paralelización. En Origin 2000.

DVR=30,PM2D=20000,EC3D=15000,NFUNCI=1200					
Proc.	-pfa	BLOQ	CICL	COL-CIC	COL-DIN
2	0.70	0.66	0.62	0.66	0.96
3	0.51	0.59	0.60	0.59	0.93
4	0.35	0.54	0.56	0.51	0.97
5		0.54	0.52	0.56	0.91
6		0.51	0.49	0.51	0.96
7		0.50	0.52	0.54	0.97
8		0.49	0.49	0.50	0.93
DVR=40,PM2D=25000,EC3D=20000,NFUNCI=1200					
Proc.	-pfa	BLOQ	CICL	COL-CIC	COL-DIN
2	0.75	0.62	0.53	0.62	0.96
3	0.62	0.44	0.54	0.59	0.88
4	0.53	0.50	0.52	0.57	0.88
5		0.48	0.46	0.51	0.85
6		0.42	0.46	0.44	0.91
7		0.48	0.46	0.46	0.86
8		0.45	0.45	0.48	0.85

En la tabla 16 se muestra la eficiencia obtenida con los diferentes métodos de paralelización cuando se toma como tiempo secuencial el obtenido con paralelización implícita, que es el mejor en esos casos.

Con paralelización implícita se obtiene mayor eficiencia con mayor volumen de datos y disminuye al aumentar el número de procesadores. Ese es un comportamiento normal en computación paralela.

Con paralelización explícita el comportamiento es algo extraño, pues con mayor volumen de cómputo se reduce un poco la eficiencia. Esto puede ser debido a que en el segundo experimento los tamaños de los vectores propios calculados en D2D varían en un rango más amplio que en el primer experimento, lo que puede producir un desbalanceo mayor en la asignación de los trabajos.

Los resultados de COL-DIN se pueden considerar satisfactorios. Se podría intentar mejorarlos al menos de dos maneras: agrupando varios índices por bloques y asignando estos bloques dinámicamente, con lo que se reduce el tiempo de gestión pero se disminuiría el balanceo; o paralelizando los dos blucles externos, con lo que se aumenta el tiempo de gestión pero se puede aumentar el balanceo. Algunos experimentos en este sentido no han producido resultados reseñables.

4.2.1 Otra mejora secuencial en la formación del Hamiltoniano

Como ya hemos dicho, la formación del Hamiltoniano tiene un coste de orden $O(n^2 d)$ siendo d la longitud de los vectores calculados en la fase anterior de la ejecución. En el programa se realiza un bucle hasta $dim(vector)$, siendo $vector$ el que corresponde al primer índice (i), y para cada elemento de este vector se obtiene otro elemento en un vector correspondiente al índice j. Si se toma como vector el de menor longitud de los correspondientes a los dos índices se puede

Tabla 17: Comparación del tiempo de formación del Hamiltoniano variando el número de procesadores con COL-DIN cuando no se elige el menor de los dos vectores y cuando sí se elige. En Origin 2000.

Proc:	1	2	3	4	5	6	7	8
DVR=30,PM2D=20000,EC3D=15000,NFUNCI=1200								
original	24.75	12.87	8.80	6.35	5.40	4.29	3.63	3.32
mejorado	23.83	11.79	7.83	5.72	4.84	4.18	3.58	3.11
% reducción	3.72	8.39	11.02	9.92	10.37	2.56	1.38	6.33
DVR=40,PM2D=25000,EC3D=20000,NFUNCI=1200								
original	196.25	95.28	69.89	52.08	43.21	33.57	30.35	26.84
mejorado	189.18	92.35	61.97	47.86	39.43	32.02	27.87	25.31
% reducción	3.60	3.08	11.33	8.07	8.75	4.62	8.17	5.70

reducir el tiempo de ejecución. No podemos predecir la reducción que se producirá, pues ésta depende de las longitudes de los vectores, de cuántos vectores tengamos de cada longitud, de la posición que acupen una vez ordenados según los valores propios, ...

Si hacemos una estimación suponiendo que tenemos $m+1$ vectores v_i con $i=0,1,\ldots,m$, de longitudes $\frac{m+i}{2m}l$ (lo que quiere decir que el vector más grande es de longitud el doble que el más pequeño y que las longitudes de los vectores están uniformemente distribuidas entre $\frac{l}{2}$ y l), las longitudes medias de los vectores usados en el bucle más interno son $\frac{3}{4}l$ si no se elige el vector más corto para hacer el bucle, y $\frac{2}{3}l$ si sí se elige este vector. De esta forma, la reducción esperada en el tiempo de ejecución con las longitudes de los vectores que hemos estimado es de un 11%.

En la tabla 17 comparamos los tiempos de ejecución obtenidos al variar el número de procesadores con el método que realiza el acceso por columnas y hace una asignación dinámica del trabajo (COL-DIN en la tabla 16) sin elegir el menor de los vectores y eligiendo el menor. Vemos que las reducciones obtenidas en el tiempo de ejecución son menores del 11% estimado, pero si vemos los tamaños de los vectores para estos dos problemas (tabla 4) observamos que la diferencia de tamaño entre el menor y el mayor vector es menor que la que hemos asumido en la estimación, lo que permite predecir una reducción algo menor que la estimada.

4.2.2 Prestaciones

Para analizar las prestaciones obtenidas en la paralelización del Hamiltoniano compararemos resultados obtenidos con diferentes experimentos. Ya que en los experimentos se han utilizado colas con distintas características: diferente capacidad de memoria, y uso en forma exclusiva o compartida; pueden aparecer fluctuaciones en los resultados que mostramos debidos a esas diferencias.

En la tabla 18 se muestran las prestaciones obtenidas según la fórmula $\frac{n^3}{pt10^6}$, donde p es el número de procesadores, t el tiempo de ejecución, y n la dimensión del Hamiltoniano que se forma. De esta forma tenemos unas prestaciones por procesador, y multiplicando por el número de operaciones en coma flotante que conlleva el cálculo de un elemento de la matriz obtendríamos los Mflops/nodo. La mayoría de los datos de la tabla se han obtenido trabajando en colas donde no está asegurada la exclusión, pero en el caso en que se dispone de datos de colas exclusivas se utilizan estos tiempos; estos casos se muestran en itálica en la tabla. Se comprueba

Tabla 18: Prestaciones por nodo en la formación del Hamiltoniano. Con asignación dinámica del trabajo, y valores de PM2D=25000, EC3D=20000. En Origin 2000.

p=2	p=4	p=6	p=8	p=12	p=16
DVR=40,NFUNCI=1600,n=1600					
9.59	10.04	*11.69*	10.22		
DVR=40,NFUNCI=2000,n=2000					
	12.06	12.02	12.88		
DVR=40,NFUNCI=3000,n=2078					
		10.60	10.46	*10.99*	*11.07*
DVR=50,NFUNCI=2500,n=2500					
6.37	6.76	6.55	7.07		
DVR=50,NFUNCI=3000,n=2930					
	8.76	*8.80*	7.90		
DVR=50,NFUNCI=4000,n=2930					
		7.73	7.27	*7.64*	
DVR=50,NFUNCI=5000,n=2930					
				6.32	

Tabla 19: Comparación de tiempos de ejecución, en segundos, en la formación del Hamiltoniano con asignación dinámica cuando se utilizan colas exclusivas y no exclusivas. Valores de PM2D=25000, EC3D=20000. En Origin 2000.

	exclusiva	no exclusiva
DVR=40,NFUNCI=1600,n=1600,p=6	58.39	69.20
DVR=50,NFUNCI=3000,n=2930,p=6	476.53	511.49

que las prestaciones se mantienen e incluso suben al aumentar el número de procesadores, lo que confirma la buena escalabilidad del sistema. También se observa que manteniendo el tamaño del Hamiltoniano constante pero aumentando la cantidad de memoria utilizada (n=2930 y NFUNCI=3000, 4000 o 5000) las prestaciones disminuyen al gestionar el sistema una mayor cantidad de memoria y estar los datos más separados en ésta.

En cuanto a las diferencias entre ejecutar los programas en colas que proporcionan exclusividad o no, vemos que las prestaciones cuando se trabaja en forma exclusiva son algo mayores, pero la diferencia no es muy grande. Comparando tiempos de ejecuciones en los dos casos se aprecia una diferencia en el tiempo de entre un 10 y un 15% (tabla 19).

4.2.3 Posible paralelización con asignación estática balanceada

En las tablas 15 y 16 podemos observar que la asignación dinámica del trabajo a los threads da buenos resultados de paralelización, por lo que será difícil reducir aún más el tiempo de ejecución.

De cualquier modo, la gestión dinámica produce un trabajo que puede ser evitado si se realiza una asignación estática balanceada, aunque esto será a costa de un trabajo inicial para realizar

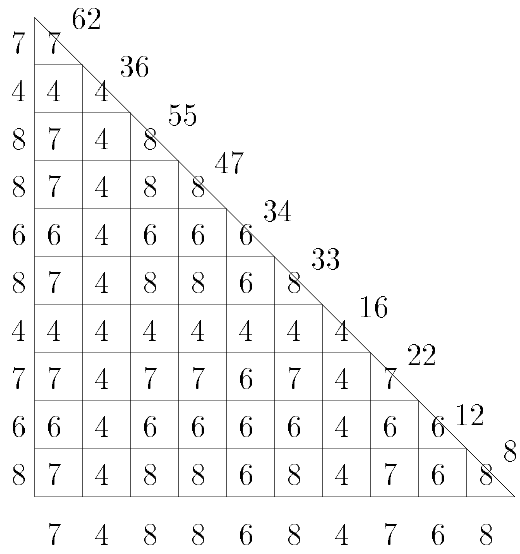

Figura 1: Coste de la formación de cada elemento del Hamiltoniano con el ejemplo 7, 4, 8, 8, 6, 8, 4, 7, 6 y 8.

la asignación, y puede que esta parte sea más costosa que la de gestión dinámica.

Veremos con un ejemplo cómo se podría hacer la asignación estática balanceada. Suponemos problemas de tamaños 7, 4, 8 y 6, y que tenemos 10 vectores propios para formar el Hamiltoniano, y que los tamaños de los vectores son 7, 4, 8, 8, 6, 8, 4, 7, 6 y 8. De este modo, cada elemento del Hamiltoniano se calcula con un coste que se muestra en la figura 1, donde aparece también el coste de cada columna y los tamaños de los vectores que intervienen en la formación.

Si la asignación dinámica se hace asignando las primeras columnas a los threads y asignando las siguientes según fueran quedando libres, la asignación de trabajos con 2 y 5 procesadores sería:

- Con 2 procesadores. Columna 1 al procesador 1 (coste 62), columna 2 al 2 (coste 36), columna 3 al 2 (coste total 91), columna 4 al 1 (coste total 109), columna 5 al 2 (coste total 125), columna 6 al 1 (coste 142), columna 7 al 2 (coste 141), columna 8 al 2 (coste 163), columna 9 al 1 (coste 154), y columna 10 al 1 (coste 162).

- Con 5 procesadores. Columna 1 al 1 (62), columna 2 al 2 (36), columna 3 al 3 (55), columna 4 al 4 (47), columna 5 al 5 (34), columna 6 al 5 (67), columna 7 al 2 (52), columna 8 al 4 (69), columna 9 al 3 (67), y columna 10 al 2 (60).

La asignación estática podría realizarse asignando todas las columnas, salvo la primera, que normalmente es la más costosa, y finalmente asignar dicha columna intentando balancear la asignación. Con 2 y 5 procesadores sería:

- Con 2 procesadores. Columna 2 al 1 (coste 36), columna 3 al 2 (55), columna 4 al 1 (83), columna 5 al 2 (89), columna 6 al 1 (116), columna 7 al 2 (105), columna 8 al 2 (127), columna 9 al 1 (128), columna 10 al 2 (135); de la columna 1, elemento 1 al 1 (135), elemento 2 al 1 (139), elemento 3 al 2 (142), elemento 4 al 1 (146), elemento 5 al 2 (148), elemento 6 al 1 (153), elemento 7 al 2 (152), elemento 8 al 2 (159), elemento 9 al 1 (159), elemento 10 al 2 (166).

- Con 5 procesadores. Columna 2 al 1 (36), columna 3 al 2 (55), columna 4 al 3 (47), columna 5 al 4 (34), columna 6 al 5 (33),columna 7 al 5 (49), columna 8 al 4 (56), columna 9 al 1 (48), columna 10 al 3 (55); de la columna 1, elemento 1 al 1 (55), elemento 2 al 5 (53), elemento 3 al 5 (60), elemento 4 al 1 (62), elemento 5 al 2 (61), elemento 6 al 3 (62), elemento 7 al 4 (60), elemento 8 al 4 (67), elemento 9 al 5 (66), y elemento 10 al 2 (68).

De esta manera, se tiene un tiempo de ejecución estimado de 163 con asignación dinámica y 166 con asignación estática (que podría ser de 163 también si se analiza estáticamente qué asignación es mejor) para 2 procesadores, y de 69 con asignación dinámica y 68 con estática para 5 procesadores.

Habría que comprobar experimentalmente si la asignación estática puede mejorar a la dinámica, pero quizás no merezca la pena el esfuerzo que hay que hacer para la pequeña mejora que se puede obtener. En el ejemplo, con 2 procesadores el coste mínimo que se puede alcanzar es de 161, y con 5 procesadores de 65, lo que supone una reducción que no llega al 5%.

4.2.4 Resolución incremental

En algunos casos puede ser preferible no formar el Hamiltoniano completo y resolver el problema de valores propios asociado, sino que se puede formar una sucesión de partes del Hamiltoniano, H_1, H_2, ..., ir resolviendo los problemas de valores propios asociados y cuando los valores propios que se quiere calcular de H_i coinciden, con una cierta tolerancia, con los de H_{i+1} se consideran estos valores propios como los que se quiere calcular. De esta manera G3D y D3D se solaparían y el paralelismo se obtendría sobre un volumen mayor de cómputo. Además, como veremos más adelante, la paralelización de D3D no da muy buenos resultados, y la paralelización de G3D y D3D de forma conjunta puede consistir básicamente en la resolución secuencial de los problemas de valores propios asociados a los H_i y la formación paralela por el resto de los procesadores de las filas (o columnas) de H_{i+1} que no forman parte de H_i.

Un esquema del algoritmo puede ser el que se muestra en la figura 2. El esquema consiste en formar primero un Hamiltoniano con una dimensión inicial $filaini$ e interviniendo todos los procesadores. Una vez formado el Hamiltoniano inicial uno de los procesadores (P_0) se dedica a resolver problemas de valores propios mientras que el resto de procesadores van añadiendo filas al Hamiltoniano: construir $H(filaloc, 1 : filaloc)$; por cada fila añadida se aumenta en uno la variable global $fila$ a la que accede P_0 tras resolver un problema de valores propios para obtener la dimensión del siguiente problema a resolver. P_0 tras resolver un problema de valores propios comprueba si se cumple el criterio de parada: obtener $finloc$; y pone su valor en la variable global fin.

Dado que el coste de G3D es mayor que el de D3D el trabajo puede balancearse, pero habría que imponer unos límites en el tamaño máximo del Hamiltoniano o en los incrementos que se utilizan entre Hamiltonianos sucesivos. Cuando el tamaño del problema de valores propios a resolver es suficientemente grande puede ser conveniente utilizar más de un procesador en su resolución y menos en la formación de los Hamiltonianos. Otra posibilidad es que en la resolución de los problemas de valores propios se calculen sólo los valores, y los vectores se calculen con el Hamiltoniano final una vez que se consideran obtenidos los valores propios, y en el cálculo de los vectores podrían intervenir todos los procesadores. Todas estas posibilidades necesitan de un estudio complementario que debería incluir simulaciones del tipo de las realizadas en 2D y que se verán más adelantes, y estudios experimentales.

```
fin=FALSE
filaloc=filaini
fila=filaini
formar en paralelo con procesadores P_0, ...  P_{p-1}, H(1 : filaloc, 1 : filaloc)
En cada thread P_i, i = 0, 1, ..., p-1; compartido fila, fin; local finloc, filaloc:
SI i = 0
        resolver H(1 : filaloc, 1 : filaloc)
        finloc=FALSE
        MIENTRAS no finloc
                C$OMP CRITICAL(EMfila)
                        filaloc = fila
                C$OMP END CRITICAL (EMfila)
                resolver H(1 : filaloc, 1 : filaloc)
                obtener finloc
                C$OMP CRITICAL (EMfin)
                        fin = finloc
                C$OMP END CRITICAL (EMfin)
        FINMIENTRAS
EN OTRO CASO
        filaloc = filaloc + 1
        C$OMP CRITICAL (EMfin)
                finloc = fin
        C$OMP END CRITICAL (EMfin)
        MIENTRAS no finloc
                construir en paralelo con procesadores P_1, ...  P_{p-1}, H(filaloc, 1 : filaloc)
                barrera 1, 2, ..., p-1
                SI i = 1
                        C$OMP CRITICAL (EMfila)
                                fila = filaloc
                        C$OMP END CRITICAL (EMfila)
                FINSI
                filaloc = filaloc + 1
                C$OMP CRITICAL (EMfin)
                        finloc = fin
                C$OMP END CRITICAL (EMfin)
        FINMIENTRAS
FINSI
```

Figura 2: Esquema de la paralelización conjunta de G3D y D3D cuando se resuelven de forma incremental.

4.3 Paralelización de la resolución de los problemas 2D

Como vemos en la tabla 14, la resolución de problemas de valores propios en el Origin 2000, utilizando las rutinas de LAPACK supuestamente paralelizadas y optimizadas para esta máquina, no da buenos resultados, siendo preferible resolver los problemas usando un único procesador.

Para evitar esta mala paralelización se podría intentar diseñar métodos de resolución de problemas de valores propios en paralelo que funcionen de manera eficiente, o en la parte de resolución de problemas "medianos" de valores propios, asignar distintos problemas a threads distintos de manera que los problemas de valores propios se resuelvan en paralelo y cada uno de ellos en secuencial. Hay que tener en cuenta que tenemos unas restricciones de memoria que harán que no podamos resolver al mismo tiempo un número ilimitado de problemas, sino que este número estará limitado por el número de procesadores y por la memoria ocupada. Cuando los problemas son muy grandes puede que sólo quepan en memoria unos pocos o incluso sólo uno. En D3D se resuelven problemas de valores propios de "gran" dimensión, y normalmente un número reducido, incluso sólo uno, por lo que en esa parte del programa tendría más sentido la paralelización de la resolución de un problema de valores propios. Por el contrario, en 2D los problemas son normalmente más pequeños, por lo que pueden ejecutarse más al mismo tiempo al caber en memoria, y además su tamaño puede no ser lo suficientemente grande como para que la paralelización de la resolución de manera individual dé resultados satisfactorios.

Por tanto, empezaremos analizando la paralelización de la resolución "global" de los problemas "medianos" de valores propios. Se trata de determinar una asignación temporizada de los trabajos y la memoria a los threads, de manera que en cada momento los trabajos asignados tengan memoria suficiente para poder ejecutarse. Esta asignación puede ser estática, determinada secuencialmente antes de poner en marcha los threads, o dinámica poniendo en marcha los threads y accediendo estos en exclusión mutua a las variables que describen la asignación de memoria y trabajos en cada momento. En cualquier caso se trata de un problema de administración de memoria [14].

Cada thread, además de resolver el problema de valores propios a él asignado, accedería después de la resolución a mezclar los valores y vectores propios calculados con los calculados por otros threads.

Para explicar y analizar el funcionamiento de este método realizaremos una simulación suponiendo que la ocupación de memoria en un problema de valores propios es de orden n^2, y que el coste es de orden n^3, con lo que no tendremos en cuenta términos de menor orden ni el coste correspondiente a la mezcla.

Posteriormente al análisis de la simulación realizaremos un programa que cree los threads y estos determinen dinámicamente los trabajos a realizar por cada uno de ellos y las zonas de memoria donde trabajar.

4.3.1 Ejemplo de simulación

Vamos a analizar cómo se realizaría la asignación de trabajos y memoria con un pequeño ejemplo. La memoria tendrá capacidad 121, y tendremos 6 problemas de tamaños 8, 7, 7, 6, 5 y 4, con ocupaciones de memoria 64, 49, 49, 36, 25 y 16, y coste aproximado de resolución 512, 343, 343, 216, 125 y 64. Supondremos 3 procesadores.

Utilizaremos una lista que describirá la ocupación de la memoria y la asignación de trabajos. Inicialmente la lista contendrá:

1	121	0	0

donde la primera componente indica el inicio de la zona de memoria, la segunda la longitud, la tercera el procesador que está trabajando en esta zona (un cero indica que está libre), y la cuarta el instante de tiempo en que se ha llegado a esta configuración.

Se empieza a asignar trabajos a los procesadores, empezando por los trabajos más costosos para intentar disminuir el desbalanceo y para que las zonas de memoria que se liberen puedan ser utilizadas en la resolución de trabajos posteriores al ser estos de menor tamaño que los previamente resueltos. Al asignar el primer trabajo al primer procesador la lista queda:

1	64	1	512
65	57	0	0

donde vemos que en el instante cero queda una zona de memoria de capacidad 57 libre (no tenemos en cuenta el tiempo de gestión de la memoria y de asignación de los trabajos), y una primera zona donde el primer procesador empieza a resolver el primer problema que acabará de resolverse en el instante 512.

Después se asigna el segundo trabajo al segundo procesador, y la lista queda:

1	64	1	512
65	49	2	343
114	8	0	0

Sólo queda una zona de memoria libre, donde no cabe ninguno de los trabajos a realizar. Cuando acaba el primero de los trabajos que están ejecutándose la lista vuelve a quedar:

1	64	1	512
65	57	0	343

Los sucesivos estados de la lista son:

1	64	1	512
65	49	2	686
114	8	0	343

1	64	0	512
65	49	2	686
114	8	0	512

1	36	1	728
37	28	0	512
65	49	2	686
114	8	0	512

1	36	1	728
37	25	3	637
62	3	0	512
65	49	2	686
114	8	0	512

1	36	1	728
37	28	0	637
65	49	2	686
114	8	0	637

1	36	1	728
37	16	3	701
53	12	0	637
65	49	2	686
114	8	0	637

Con lo que el procesador uno acaba en el instante 728 habiendo ejecutado los trabajos 1 y 4, que empieza en los instantes 0 y 512; el procesador dos ejecuta los trabajos 2 y 3, que empieza en los instantes 0 y 343, y acaba su ejecución en el 686; y el procesador tres ejecuta los trabajos 5 y 6, que empieza en los instantes 512 y 637, y acaba en el 701. El tiempo de ejecución del programa será 728, mientras que el tiempo de ejecución en secuencial sería 1603. La eficiencia es de 0.73, que no es muy buena debido a que la capacidad de memoria nos limita las posibilidades de paralelismo. Por otro lado, lo más rápido que se podría resolver los problemas en paralelo contando con suficiente memoria sería 512, por lo que el coste del mayor de los problemas también nos impone un límite en la posibilidad de paralelización, y el speed-up máximo alcanzable es 3.13.

Esta forma de asignar memoria y procesadores para la resolución de los problemas parece buena al dejar para el final los trabajos más cortos, pero el no reorganizar la memoria hace que en algunos casos dispongamos de memoria suficiente para alojar un trabajo y éste no pueda ser ejecutado al no estar la memoria contigua. Esto se puede ver en el ejemplo en que las dimensiones de los trabajos son 4, 3, 3, 3, 3, 3 y la de la memoria 27. En este caso los sucesivos estados del mapa de memoria son:

1	27	0	0

1	16	1	64
17	11	0	0

1	16	1	64
17	9	2	27
26	2	0	0

1	16	1	64
17	11	0	27

1	16	1	64
17	9	2	54
26	2	0	27

1	16	1	64
17	11	0	54

1	16	1	64
17	9	2	81
26	2	0	54

1	16	0	64
17	9	2	81
26	2	0	64

1	9	1	91
10	7	0	64
17	9	2	81
26	2	0	64

y en este momento, aunque la cantidad de memoria libre es de 9 posiciones, no podemos asignar esta memoria para resolver un nuevo problema porque la memoria libre está dividida en dos trozos. Una posible solución sería hacer una compactación de la memoria, pero ello conllevaría paralizar las ejecuciones en curso y reorganizar los datos en la memoria, lo que no daría buenos resultados. Otra posible solución sería hacer una asignación estática de los trabajos obteniendo en qué posiciones deben alojarse para evitarnos problemas de este tipo (por ejemplo asignando los trabajos de dimensión 3 a partir de la posición 19 en vez de la 17 queda una zona libre de 9 posiciones entre dos zonas ocupadas, y el trabajo se puede asignar), pero parece que este problema de asignación no es de fácil solución.

4.3.2 Resultados de simulaciones

Para analizar el posible comportamiento del método con los problemas del tipo que estamos resolviendo, simulamos los dos casos de la tabla 14. En los dos casos el tamaño de la memoria es 1.440.000. En el primer caso tenemos 15 problemas de tamaños 424, 417, 401, 372, 355, 345, 319, 299, 298, 288, 276, 276, 267, 261 y 254. En el segundo caso tenemos 20 problemas de tamaños 744, 733, 715, 684, 683, 647, 611, 577, 568, 546, 523, 517, 496, 482, 481, 473, 469, 467, 455 y 450. La diferencia en la ocupación de memoria y principalmente de tiempo de ejecución de los trabajos es muy grande, por lo que hay que poner cuidado en hacer una asignación balanceada.

En la tabla 20 se muestran los tiempos de ejecución simulados de los dos problemas variando en número de procesadores y con tamaño de memoria 1.440.000, y del segundo problema también con tamaños 2.560.000 y 3.240.000. En la tabla 21 se muestran las eficiencias obtenidas. Se observa lo que ya habíamos comentado: que la asignación de trabajos está bien balanceada, pero que la capacidad de la memoria impone un límite en las prestaciones que se pueden obtener, llegando a ser improductivo el aumentar el número de procesadores que se usan. En el caso 1 sí se llega a obtener el tiempo mínimo que se puede alcanzar con una paralelización de este tipo y que corresponde al tiempo de ejecución del trabajo más costoso; pero en el caso 2 no se llega a alcanzar este tiempo mínimo, que es 412, debido a que la memoria de que disponemos en las simulaciones y el número de procesadores no es suficiente.

4.3.3 Otra mejora secuencial en el almacenamiento de las soluciones de los problemas ”medianos” de valores propios

En la paralelización de la solución de los problemas ”medianos” de valores propios se pueden obtener buenos resultados, según las simulaciones, si se paraleliza la solución de forma global, pero en ese caso cada procesador tendrá que almacenar los datos que ha obtenido mezclándolos con los obtenidos por los demás procesadores. Para dejar las mezclas para después de resolver los problemas de valores propios se necesitaría almacenar todos los valores y vectores calculados, lo que nos puede dar necesidades de memoria imprevistas. Como las mezclas se realizan estando los procesadores ocupados el único procesador que podemos asegurar que esté disponible para realizar la mezcla es el que acaba de resolver el problema de valores propios, que será el que realice la mezcla accediendo en exclusión a la zona donde se almacenan las soluciones. Esto hace que la parte 2D del programa se paralelice de forma conjunta pero realizándose A2D en forma

Tabla 20: Tiempo de ejecución simulado, en millones de unidades.

Proc.	caso 1 mem. 1.440.000	caso 2 mem. 1.440.000	caso 2 mem. 2.560.000	caso 2 mem. 3.240.000
1	555	3969	3969	3969
2	285	1984	1984	1984
3	185	1353	1353	1353
4	143	1226	994	994
5	116	1223	797	797
6	94	1223	788	712
7	85	1223	695	596
8	76	1223	691	579
9	76	1223	691	573
10	76	1223	691	568

Tabla 21: Eficiencia con los tiempos simulados.

Proc.	caso 1 mem. 1.440.000	caso 2 mem. 1.440.000	caso 2 mem. 2.560.000	caso 2 mem. 3.240.000
2	0.97	1.00	1.00	1.00
3	1.00	0.98	0.98	0.98
4	0.97	0.81	1.00	1.00
5	0.96	0.65	1.00	1.00
6	0.98	0.54	0.84	0.93
7	0.93	0.46	0.82	0.95
8	0.91	0.41	0.72	0.86
9	0.81	0.36	0.64	0.77
10	0.73	0.32	0.57	0.70

secuencial, por lo que se puede convertir en un cuello de botella según los tiempos de la tabla 9, además de perder la reducción de tiempo que se obtiene paralelizando A2D (tabla 14).

Hasta ahora nos habíamos conformado con la reducción de tiempos obtenida en A2D con respecto al programa inicial (tabla 9), pero ahora vemos necesaria su reducción aún más para que esta parte no se nos convierta en un cuello de botella en el programa paralelo. En el programa original los valores y vectores propios calculados en cada paso se mezclan con los de los pasos anteriores con una ordenación por inserción, lo que provoca una gran cantidad de movimientos de datos, pero se pueden mezclar del array donde están los datos ordenados ($E2D$) con los de los valores propios calculados (D) en $E2D$, ordenando del mismo modo los vectores propios y los números cuánticos de cada valor. De este modo se pasa de un coste cúbico a uno cuadrático, produciéndose una gran reducción en el tiempo de ejecución (tabla 22) que hace que, ahora sí, podamos considerar la paralelización de 2D sin que la realización paralela de A2D sea un problema importante.

Tabla 22: Tiempo de ejecución de A2D, en segundos, con almacenamiento por inserción y por mezcla.

SUNUltra					
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=15000	46.78	0.98
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=15000	47.28	1.25
Origin					
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=15000	1.75	0.20
NFUNCI=1200	DVR=30	PM2D=20000	EC3D=20000	6.30	0.38
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=15000	1.85	0.32
NFUNCI=1200	DVR=30	PM2D=25000	EC3D=20000	6.50	0.56
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=15000	5.33	0.94
NFUNCI=1200	DVR=40	PM2D=20000	EC3D=20000	13.60	1.41
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=15000	5.49	1.13
NFUNCI=1200	DVR=40	PM2D=25000	EC3D=20000	13.93	1.69

4.3.4 Paralelización de 2D

El esquema de la paralelización se da en la figura 3. Cada thread, mientras quedan trabajos por hacer, consulta en exclusión mutua el mapa de memoria libre y busca un trabajo que quepa en la memoria, si no lo encuentra libera la exclusión en el acceso al mapa de memoria y vuelve a consultar, y si encuentra un trabajo que quepa en la memoria, reserva esa zona de memoria modificando el mapa y marca el trabajo como asignado; una vez realizado el trabajo se accede en exclusión mutua a mezclar los valores y vectores calculados con los ya almacenados, y después se accede en exclusión mutua al mapa de memoria para indicar la memoria que ya no se está usando. Los trabajos se van intentando asignar del mayor al menor, de manera que queden los trabajos más pequeños para el final, con lo que se pretende balancear la asignación de trabajos.

En la tabla 23 se muestran los tiempos de ejecución y las eficiencias obtenidas en la resolución de 2D cuando varía el número de procesadores y el tamaño de la memoria utilizada en la resolución de problemas de valores propios. En la tabla 24 se muestra la eficiencia. Si comparamos con las tablas de la simulación (tablas 20 y 21) vemos que los resultados de la paralelización son algo peores que los simulados, lo que es normal por los comentarios que habíamos hecho: la mezcla se hace en forma secuencial, no se simulaba el coste de gestión del mapa de memoria, y sólo hemos tenido en cuenta el término de mayor orden para la estimación de los tiempos. De cualquier modo el comportamiento es parecido al que se observa en la simulación: al aumentar el número de procesadores se reduce el tiempo de ejecución pero esta reducción está limitada por el tamaño de la memoria y el de los problemas, lo que hace que se llega a un punto a partir del cual no se reduce el tiempo aumentando el número de procesadores, pero sí se reduce aumentando la memoria utilizada.

La memoria que aparece indica la que se utiliza para almacenar los problemas, y otra de igual tamaño se utiliza para los vectores propios. Además de esa memoria se utilizan otras zonas de memoria para variables auxiliares, para zonas de trabajo temporales, para los valores propios, etc. Los tamaños de memoria utilizados para 1.440.000, 2.560.000, 3.240.000 y 4.000.000 son aproximadamente 34 Mb, 60 Mb, 77 Mb y 96 Mb, respectivamente. Estos tamaños pueden parecer pequeños, pero el método es general y sirve para tamaños más grandes y, además, en algunos casos puede ser preferible no utilizar toda la memoria de la que podemos disponer por

```
En cada thread:
MIENTRAS queden trabajos sin realizar
       C$OMP CRITICAL (memoria)
              buscar trabajo que quepa en la memoria
              SI hay trabajo marcar la zona de memoria como reservada
       C$OMP END CRITICAL (memoria)
       SI hay trabajo
              resolver el PVP correspondiente a trabajo
       C$OMP CRITICAL (mezcla)
              mezclar valores y vectores propios obtenidos con los ya almacenados
       C$OMP END CRITICAL (mezcla)
       C$OMP CRITICAL (memoria)
              liberar zona de memoria en el mapa
       C$OMP END CRITICAL (memoria)
       FINSI
FINMIENTRAS
```

Figura 3: Esquema de la paralelización de 2D.

varios motivos: más usuarios en el sistema que también necesitan utilizar memoria, mejor tiempo de respuesta de la cola a la que se han mandado los trabajos si se demandan menos recursos, mejor utilización de la jerarquía de memorias, ...

4.3.5 Comparación de simulaciones y resultados experimentales

Las simulaciones deberían servinos para predecir el comportamiento real del programa, con lo que se podrían realizar simulaciones previas a la ejecución para determinar cuál es el tamaño de memoria y/o el número de procesadores a utilizar según lo que nos interese: resolver el problema en el menor tiempo dentro de unos límites de memoria y procesadores, resolverlo con una cierta eficiencia, ... Hemos visto que en general la forma del comportamiento se predice de manera adecuada, pero hay muchas cosas que no hemos tenido en cuenta en las simulaciones anteriores: se considera coste n^3 sin términos de menor orden, no se consideran las mezclas, no se considera

Tabla 23: Tiempo de ejecución, en segundos, en la resolución de 2D. En Origin 2000.

Proc.	caso 1 mem. 1.440.000	caso 2 mem. 1.440.000	caso 2 mem. 2.560.000	caso 2 mem. 3.240.000	caso 2 mem. 4.000.000
1	5.35	61.88	61.88	61.88	61.88
2	3.09	31.11	31.60	40.43	32.45
3	2.07	32.83	23.17	25.18	22.69
4	1.63	27.74	19.54	19.12	18.17
5	1.36	27.68	17.44	15.61	15.55
6	1.15	28.08	17.88	14.79	15.03
7	1.05	30.14	16.05	14.74	13.41
8	0.92	27.75	17.76	16.38	14.19

Tabla 24: Eficiencia en la resolución de 2D. En Origin 2000.

Proc.	caso 1 mem. 1.440.000	caso 2 mem. 1.440.000	caso 2 mem. 2.560.000	caso 2 mem. 3.240.000	caso 2 mem. 4.000.000
2	0.87	0.99	0.98	0.77	0.95
3	0.86	0.63	0.89	0.82	0.91
4	0.82	0.58	0.79	0.81	0.85
5	0.66	0.45	0.71	0.79	0.80
6	0.78	0.38	0.58	0.70	0.69
7	0.73	0.29	0.55	0.60	0.66
8	0.73	0.28	0.44	0.47	0.55

Tabla 25: Resultados de simulaciones para DVR=30, PM2D=20000 y EC3D=15000. En millones de operaciones.

	34 Mb
1	2692
2	1388
3	901
4	698
5	561
6	466
7	427
8	372
9	348

el coste de gestión de los procesos, ... Las mezclas seguiremos sin considerarlas pues su coste es de un orden muy pequeño en comparación con el de resolver los problemas de valores propios; la gestión de los procesos hará que los tiempos experimentales sean mayores que los simulados sobre todo al aumentar el número de procesadores, pero no la tendremos en cuenta; y para mejorar la estimación realizada suponiendo un coste n^3 se puede utilizar una mejor aproximación al coste según los costes que se estudiarán en el apartado de paralelización de los problemas de valores propios, pero no sabemos el número de valores y vectores a calcular, con lo que seguirá habiendo una cierta diferencia entre los resultados reales y de las simulaciones.

En las tablas 25, 26, 27, 28 y 29 mostramos los resultados de algunas de las simulaciones realizadas. En todas las tablas se muestran millones de operaciones al variar el número de procesadores y la memoria utilizada en Mb. La tabla 25 corresponde a valores DVR=30, PM2D=20000 y EC3D=15000; la tabla 26 a DVR=40, PM2D=25000 y EC3D=20000; la 27 a DVR=50, PM2D=25000 y EC3D=20000; la 28 a DVR=60, PM2D=25000 y EC3D=20000; y la 29 a DVR=70, PM2D=25000 y EC3D=20000.

En todas las tablas se aumenta la memoria hasta obtener el máximo speed-up alcanzable, que viene determinado por el mayor problema de valores propios a resolver; para reducir más el tiempo de ejecución aumentando el número de procesadores tendríamos que obtener una manera de asignar problemas a varios procesadores para resolverlos en paralelo.

Vemos también que para resolver problemas de un alto coste puede ser conveniente la utili-

Tabla 26: Resultados de simulaciones para DVR=40, PM2D=25000 y EC3D=20000. En millones de operaciones.

	62 Mb	96 Mb	216 Mb
1	28185		
2	14117		
3	9592		
4	7087	7087	
5	6807	5705	
6	6729	4972	4972
7	6729	4690	4240
8		4240	3794
9		4240	3500
10			3027
11			2965
12			2762
13			2650
14			2200
15			2039

zación de un gran número de procesadores y una gran cantidad de memoria, aunque dados unos ciertos valores máximos la simulación nos permite obtener el número conveniente de procesadores y la cantidad de memoria necesaria para resolver el problema: por ejemplo, en el caso de la tabla 26, si tenemos 120 Mb y 10 procesadores, la simulación nos aconseja utilizar 8 procesadores con 96 Mb de memoria, obteniendo un speed-up del 6.65. Dado que la simulación no es muy exacta por todos los motivos mencionados anteriormente los resultados reales no van a coincidir con los simulados. En la tabla 30 se compara el speed-up simulado y el obtenido experimentalmente variando el número de procesadores y la memoria utilizada con los problemas de las tablas 25, 26 y 27. En el tercer caso, al no haberse realizado ejecución secuencial, el speed-up experimental se ha obtenido considerando que el tiempo secuencial es el doble del obtenido con 2 procesadores. Vemos que el speed-up experimental es menor que el simulado, aumentando la diferencia al aumentar el número de procesadores, lo que es debido a que la gestión de los procesos no se ha incluido en la simulación. Quizás con una asignación estática del trabajo esto se podría evitar en parte. En el caso anterior de disponer de 120 Mb y 10 procesadores vemos que experimentalmente el máximo speed-up alcanzado es de 4.92, y no de 6.65, y que se alcanza con 6 procesadores y no con los 8 dados por la simulación, por lo que los valores obtenidos de la simulación deben ser tomados como aproximaciones a los valores óptimos.

Para ver cómo aumentan las necesidades de procesadores y memoria para obtener el máximo speed-up alcanzable mostramos en la tabla 31 dichas necesidades según se obtienen de la simulación. En la tabla se muestran datos para los cinco problemas de las tablas 25, 26, 27, 28 y 29, en las filas 2 a 6 de la tabla. En las columnas se muestran el número de problemas de valores propios a resolver, el número de procesadores y la memoria en Mb necesarios para obtener el speed-up máximo, dicho speed-up, y el speed-up máximo experimental en los dos casos en que éste se tiene. Se observa que las necesidades de recursos para reducir el tiempo de ejecución aumentan considerablemente al aumentar la dimensión de los problemas.

Tabla 27: Resultados de simulaciones para DVR=50, PM2D=25000 y EC3D=20000. En millones de operaciones.

	150 Mb	216 Mb	384 Mb	600 Mb
1	149505			
2	75789	75789		
3	50551	51264		
4	44816	38752	38752	
5	43583	33767	30146	
6	42270	32307	26367	
7	42270	29995	22206	
8		29995	20129	20129
9			18792	17406
10			17895	15646
11			17759	14098
12			17759	13458

4.3.6 Generalización en la asignación de trabajos

En las simulaciones realizadas cada problema de valores propios se ha asignado a un único procesador que lo resuelve secuencialmente. Otra posibilidad es asignar cada problema a varios procesadores a la vez para que lo resuelvan en paralelo. De esta manera el planteamiento del problema sería:

Problema: Dado un sistema de memoria compartida con capacidad de memoria M y p procesadores, y t trabajos independientes a realizar, con necesidades de memoria $m_1, m_2 \dots, m_t$, y tiempos de ejecución paralelo del trabajo t_i utilizando q procesadores $c_i(q)$. Obtener una asignación temporal de los trabajos a la memoria y a los procesadores de manera que se minimice el tiempo de ejecución.

En nuestro caso $m_i = n_i^2$, siendo n_i la dimensión del trabajo t_i; y q lo hemos considerado siempre 1 y hemos tomado $c_i = n_i^3$ o el valor obtenido del artículo de Demmel-Stanley ([15]). En este caso hemos realizado una asignación que parece buena para obtener buenas prestaciones pues se asignan primero los trabajos mayores de manera que cuando se acaban y se libera su memoria estas posiciones de memoria pueden ser ocupadas por otros trabajos sin necesidad de compactación; además el dejar los trabajos menores para el final puede balancear la asignación de trabajos. Esta manera de asignar los trabajos no tiene porqué dar una solución óptima, tal y como hemos visto en una de las simulaciones, y aunque parece que da buenos resultados, habría que demostrar para distintas funciones de memoria y tiempo cuál es la asignación óptima.

Es obvio que la asignación de los trabajos a varios procesadores puede reducir más el tiempo de ejecución, pues en algunos casos tenemos procesadores ociosos al no caber tantos trabajos en memoria como procesadores disponibles. Dependiendo del coste paralelo de resolver los trabajos la asignación a más de uno puede ser beneficiosa o no. En nuestro caso no obtenemos una gran reducción en el tiempo de ejecución al paralelizar el algoritmo de resolución del problema de valores propios, pero habría que seguir mejorando su paralelización, y diseñar rutinas de resolución a las que se pase la dirección de memoria donde se trabaja y también qué procesadores de los disponibles van a trabajar en la resolución paralela del problema; no nos servirán por tanto las rutinas paralelas de `complib.sgimath_mp`.

Tabla 28: Resultados de simulaciones para DVR=60, PM2D=25000 y EC3D=20000. En millones de operaciones.

	312 Mb	384 Mb	600 Mb	864 Mb	1176 Mb	1536 Mb
1	482530					
2	241431					
3	161266	161266				
4	145791	123407	123407			
5	136454	113723	96821			
6	131082	106343	81102	81102		
7	131082	104582	75661	72807		
8		104475	75038	62201		
9		104475	70510	57065	57065	
10			69997	51489	49210	
11			69758	50290	46304	
12			69576	49132	43293	
13			68662	48057	38233	38233
14			68662	47685	37578	37657
15				47685	37578	35030
16						32650
17						30822
18						28747

4.4 Paralelización de la resolución del problema 3D

Después de formar el Hamiltoniano se resuelve uno o varios problemas "grandes" de valores propios. En algunos casos, los problemas de valores propios 2D, que hemos llamado "medianos", son también de gran dimensión y no es posible resolver varios a la vez debido a la memoria necesaria. Por todo esto, merece la pena intentar paralelizar la resolución de un problema de valores propios.

Una posibilidad sería utilizar la rutina `dsyevx` de LAPACK utilizando la librería paralela `complib.sgimath_mp`, pero ya hemos visto (tabla 14) que el uso de esa rutina supuestamente paralela y optimizada para el sistema en que estamos trabajando no reduce el tiempo de ejecución con respecto a la versión secuencial, por lo que la paralelización obteniendo buenas prestaciones se presenta como una tarea difícil.

Por otro lado, cuando comparamos los tiempos de ejecución en la multiplicación de matrices variando el número de procesadores y usando la librería BLAS paralela (tabla 13), vemos que las prestaciones obtenidas en cuanto a paralelismo no son muy buenas: usando un procesador se obtienen 380 Mflops y aumentando el número de procesadores lo más que se llega a obtener son 626 Mflops, y el máximo speed-up alcanzado es de 1.64. Dado que la multiplicación de matrices es una operación de orden n^3, como la operación que nos ocupa, y que es bastante simple y sencilla de optimizar, podemos considerar estos números como una referencia en cuanto a las máximas prestaciones que podemos alcanzar paralelizando la resolución del problema de valores propios.

Trabajeremos en la paralelización de la rutina `dsyevx`, por lo que empezaremos analizando esta rutina.

Tabla 29: Resultados de simulaciones para DVR=70, PM2D=25000 y EC3D=20000. En millones de operaciones.

	577 Mb	600 Mb	864 Mb	1176 Mb	1536 Mb	1944 Mb	2400 Mb	2904 Mb	3456 Mb
1	1186213								
2	600543								
3	400005								
4	300251	300251	300251						
5	274624	272601	238130						
6	273227	267711	200106	200106					
7	262151	259859	192732	170593					
8	262151	259859	179284	157068	157068				
9		259859	179284	139977	136057				
10			179284	139909	122934				
11				138446	115157	115157			
12				138446	112902	106109			
13					106109	96143			
14					99253	91370	91370		
15					98150	88516	87071		
16					98150	87071	82638		
17						86512	79345		
18						84076	72433	72433	
19						84076	72433	71744	
20							72433	69023	
21								66169	
22								64723	
23								63827	
24								57034	57034
25								57034	50177
26									49075

4.4.1 Análisis de la rutina `dsyevx`

La rutina `dsyevx` trabaja reduciendo la matriz inicial a forma tridiagonal, utilizando la rutina `dsytrd`; después de esto se trabaja con la matriz reducida calculando sus valores propios (que serán los de la matriz inicial) en el intervalo deseado, utilizando la rutina `dstebz`; se calculan los vectores propios asociados a los valores calculados en la matriz reducida, usando `dstein`; y finalmente se calculan los vectores propios de la matriz inicial con la rutina `dormtr`.

En [15] se estudia detalladamente la rutina `pdsyevx`, que es una versión paralela para memoria distribuida de `dsyevx`. En ese artículo podemos ver los costes de cada una de las rutinas que se utilizan en el caso en que se calcule todo el espectro: `dsytrd` tiene un coste $\frac{4}{3}n^3$, de los que la mitad corresponden a operaciones de nivel 3 y la otra mitad a operaciones de nivel 2; `dstebz` tiene un coste $120n^2$ y `dstein` un coste $400n^2$, todas ellas de nivel 2; y `dormtr` un coste $2n^3$, con operaciones de nivel 3. Las partes más costosas de la ejecución son las de orden n^3 (reducción a forma tridiagonal y cálculo de los vectores de la matriz inicial), aunque para matrices de di-

Tabla 30: Comparación de speed-up simulado y experimental. Experimentos en Origin2000.

	p=2	p=4	p=6	p=8	p=12	p=16
DVR=30,PM2D=20000,EC3D=15000,Memoria=34 Mb						
simulado	1.94	3.86	5.78	7.24		
experimental	1.71	3.06	4.69	6.15		
DVR=40,PM2D=25000,EC3D=20000,Memoria=62 Mb						
simulado	2.00	3.98	4.19	4.19		
experimental	1.60	3.56	3.54	3.61		
DVR=40,PM2D=25000,EC3D=20000,Memoria=96 Mb						
simulado		3.98	5.67	6.65		
experimental		3.12	4.92	4.81		
DVR=40,PM2D=25000,EC3D=20000,Memoria=216 Mb						
simulado			5.67	7.43	10.20	13.82
experimental			4.46	5.40	7.24	8.49
DVR=50,PM2D=25000,EC3D=20000,Memoria=150 Mb						
simulado	1.97	3.34	3.54	3.54		
experimental	2.00	3.30	3.12	3.53		
DVR=50,PM2D=25000,EC3D=20000,Memoria=216 Mb						
simulado		3.86	4.63	4.98		
experimental		4.07	4.55	4.26		
DVR=50,PM2D=25000,EC3D=20000,Memoria=384 Mb						
simulado			5.67	7.43	8.42	
experimental			5.53	6.01	7.02	
DVR=50,PM2D=25000,EC3D=20000,Memoria=600 Mb						
simulado					11.12	
experimental					9.17	

mensión no muy grande las partes correspondientes al trabajo con la matriz reducida pueden tener gran importancia debido al valor de la constante que las afecta. Además, como en nuestro caso no se calcula todo el espectro, el tiempo de ejecución de las tres últimas rutinas se reducirá, mientras que el de la rutina `dsytrd` no varía independientemente del número de valores que se calculen. Por todo esto, la parte más costosa debe ser la de reducción a forma tridiagonal.

En la tabla 32 se muestra el tiempo de ejecución de las distintas rutinas y su porcentaje en el tiempo total para dos de los problemas que estamos considerando. Comprobamos que la rutina `dsytrd` es la más costosa y que su porcentaje aumenta al aumentar el tamaño del problema. Dado que estamos interesados en la resolución de "grandes" problemas, debemos intentar inicialmente la optimización de dicha rutina.

4.4.2 Paralelización de la rutina `dsytrd`

Para paralelizar la rutina `dsytrd` consideramos la matriz dividida en bloques de filas con b filas en cada bloque. Desde el último bloque hasta el primero se reduce cada uno de los bloques a forma tridiagonal utilizando la rutina `dlatrd`, y con la información obtenida en esta reducción se actualizan los bloques superiores de la matriz usando la rutina `dsyr2k`. En la figura 4 se

Tabla 31: Recursos necesarios para obtener el speed-up máximo en los problemas de las tablas anteriores.

PVP	proc.	mem.	sp. sim.	sp. exp.
15	8	34	7.24	6.15
20	15	216	13.82	8.49
25	12	600	11.12	
30	19	1536	16.79	
35	26	3456	24.17	

Tabla 32: Tiempo de ejecución, en segundos, y porcentaje respecto al tiempo total de las distintas rutinas de `dsyevx`.

	`dsytrd`	`dstebz`	`dstein`	`dormtr`	Total
SUNUltra,FUNCI=1200,DVR=30,PM2D=20000,EC3D=15000					
Tiempo	11.38	1.00	2.88	2.02	17.28
Porcentaje	65.9	5.8	16.7	11.7	
Origin2000,FUNCI=1200,DVR=30,PM2D=20000,EC3D=15000					
Tiempo	1.78	0.53	0.78	0.44	3.53
Porcentaje	50.6	14.9	22.0	12.5	
Origin2000,FUNCI=1200,DVR=40,PM2D=25000,EC3D=20000					
Tiempo	14.14	1.34	2.99	2.00	20.47
Porcentaje	69.1	6.5	14.6	9.8	

muestra el orden y las zonas de trabajo en cada paso.

Una versión paralela puede consistir en asignar el trabajo sobre los bloques de manera cíclica a los distintos procesadores, de manera que en cada paso en `dlatrd` trabajará el procesador que contenga el bloque y en la actualización de la matriz trabajarán todos los procesadores (figura 5). De esta manera, el aumentar el tamaño de los bloques producirá un mayor desbalanceo pero también un mejor uso de la jerarquía de memorias al asignarse datos contiguos al mismo procesador.

Sería también posible paralelizar el cálculo de `dlatrd`, pero no se ha hecho así en esta primera aproximación.

En la tabla 33 se muestran los tiempos de ejecución en D3D cuando varía el número de procesadores, con distintos tamaños de problema y de bloque, y con dos versiones en la asignación del trabajo en `dsytrd`. En las dos versiones la rutina `dsyr2k` se aplica como se indica en la figura 5; pero en la primera versión el cálculo de `dlatrd` se hace en el procesador al que corresponde según la figura 5, necesitándose la gestión de más procesos pero habiendo un esquema fijo de acceso a los datos que puede hacer que se realice de forma más eficiente; en la segunda versión `dlatrd` se aplica siempre en el procesador principal, con lo que se gestionan menos procesos pero puede ser necesario un mayor movimiento de datos en la memoria. A pesar de las grandes fluctuaciones en los tiempos de ejecución se pueden obtener algunas conclusiones:

- Se puede reducir algo el tiempo de ejecución al aumentar el número de procesadores, pero sólo a partir de tamaños suficientemente grandes.

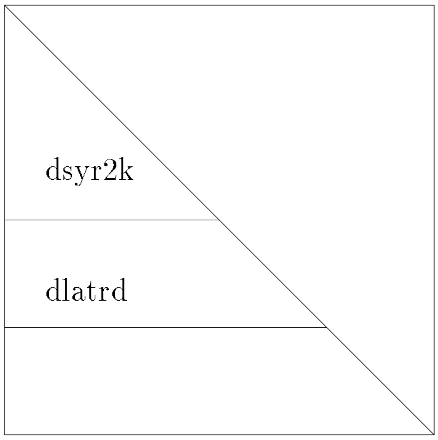

Figura 4: Esquema de un paso de `dsytrd`.

- La reducción obtenida en el tiempo de ejecución al paralelizar no es muy grande, pero si comparamos con los tiempo de ejecución obtenidos con `complib.sgimath_mp` (tabla 14) y con las prestaciones de BLAS (tabla 13) podemos considerarla satisfactoria para una primera aproximación. Obviamente se puede paralelizar algo más la rutina `dsyevx` paralelizando `dlatrd`, `dstebz`, `dstein` y `dormtr`.
- El tamaño de bloque influye en el tiempo de ejecución pareciendo el más adecuado un tamaño entre 10 y 18 para tamaños grandes del problema. Para tamaños de problema pequeños el tamaño óptimo de bloque es menor, siendo prácticamente iguales los resultados obtenidos con valores 6 y 10.
- El número de procesadores para el que se obtienen los menores tiempos de ejecución está entre 4 y 8, siendo los tiempos muy parecidos, como ocurre en la multiplicación matricial.
- No se ve una diferencia clara entre las dos versiones programadas.

5 Conclusiones y trabajos futuros

Se ha mostrado cómo es posible la paralelización en Origin 2000 de un programa de cálculo de niveles de energía vibracionales de moléculas triatómicas usando el método adiabático DVR. En la paralelización del programa se han usado distintas técnicas que dan lugar a reducciones en el tiempo de ejecución distintas para distintas partes del programa: asignación dinámica de memoria, utilización de directivas de compilación, y paralelización de problemas de valores propios. En la tabla 34 se muestran los tiempos de ejecución para las distintas partes del programa y para el total, con los problemas utilizados en las tablas anteriores y variando el número de procesadores. La reducción del tiempo de ejecución alcanzada en el conjunto del programa podemos considerar que es satisfactoria.

Quedan algunos puntos que se han ido señalando y en los que se puede seguir trabajando para reducir aún más el tiempo de ejecución:

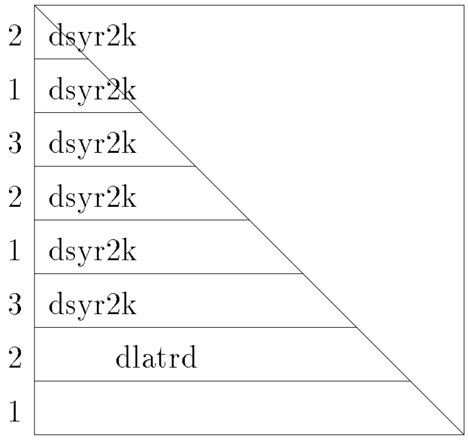

Figura 5: Esquema de paralelización de un paso de `dsytrd`.

- La paralelización del algoritmo de resolución de problemas de valores propios se ha hecho sólo en una de las cuatro rutinas que lo componen, la de reducción a forma tridiagonal, y dentro de esta sólo en la actualización de la matriz al aplicar grupos de transformadas de Householder. A pesar de la dificultad en la reducción del tiempo de ejecución en este apartado merece la pena seguir trabajando al menos por dos motivos: la reducción del tiempo aunque sea pequeña; y en algunos casos en la parte 2D del programa tenemos procesadores ociosos al resolver los problemas secuencialmente.
- Habría que obtener una asignación óptima de los trabajos a los procesadores y la memoria en el caso que nos ocupa, lo que se haría resolviendo un problema de asignación antes de la puesta en marcha de los procesos.
- La parte 2D del programa se puede resolver haciendo asignaciones de problemas de valores propios a varios procesadores que los resuelvan en paralelo utilizando preferiblemente algoritmos optimizados mencionados en el punto uno. Las rutinas correspondientes deberían diseñarse de manera que se les pase el subconjunto de procesadores, del total disponible, que trabajan en la resolución del problema. Se debería hacer un programa de simulación para este caso similar al realizado para el caso más simple que hemos analizado.
- Habría que diseñar un algoritmo para la resolución conjunta de G3D y D3D de manera incremental y comprobar, con simulaciones y experimentalmente, si este método de resolución puede ser mejor en algunos casos que el método utilizado.

Por otro lado, sólo hemos analizado la resolución del problema en SMP, pero puede ser interesante estudiar la resolución en sistemas distribuidos y/o usando técnicas out-of-core que permitan resolver problemas con necesidades de memoria mayores que la memoria principal disponible.

Agradecimientos

El trabajo que aquí se muestra se ha hecho a partir de programas secuenciales que se han desarrollado en el grupo de investigación de Química Cuántica del Departamento de Química-

Tabla 33: Tiempo de ejecución, en segundos, en `dsytrd` paralelo. En Origin 2000.

	Procesadores							
bloques	1	2	3	4	5	6	7	8
Versión 1, tamaño 703								
6	1.78	2.31	2.01	1.94	1.79	1.81	2.02	2.22
10	1.78	2.22	2.28	1.72	1.99	1.77	2.17	2.03
Versión 1, tamaño 1200								
6	14.14	14.73	11.27	10.58	10.79	9.58	11.06	10.99
10	14.14	11.53	11.09	11.52	12.43	9.74	9.59	10.41
Versión 1, tamaño 1600								
6		38.98	35.26	33.54	27.53	27.49	25.61	32.99
10		31.81	30.85	32.65	24.34	27.85	24.00	28.43
14		32.12	26.38	28.51	24.58	27.33	23.72	27.23
18		31.33	27.55	26.53	26.57	22.80	26.28	25.23
Versión 2, tamaño 1600								
6		32.76	28.65	31.27	26.71	28.88	34.89	26.02
10		33.03	28.37	22.97	24.93	26.33	26.98	27.73
14		34.61	29.46	28.01	24.58	23.91	26.38	23.85
18		32.29	31.02	25.74	23.01	35.20	31.39	27.41

Física de la Universidad de Murcia. Dichos programas han sido facilitados por los profesores Adolfo Bastida y José Zúñiga, que además han tenido la paciencia de explicarme su funcionamiento y de intentar hacer que comprendiera algo del significado Químico-Físico del problema. Como resultado de esta colaboración hemos presentado una comunicación en el Fifth European SGI/Cray MPP Workshop, Bolonia, Septiembre 1999 [16], donde se muestra un resumen de los resultados presentados en este trabajo.

Referencias

[1] A. Edelman. Large dense linear algebra in 1993: The parallel computing influence. *The International Journal of Supercomputer Applications*, 7(2):113–128, 1993.

[2] Francisco José Carvajal Rojo. Optimización de coordenadas vibracionales hiperesféricas para la molécula de H_3^+. Facultad de Química. Universidad de Murcia. Tesina. . 1995.

[3] Ira N. Levine. *Química Cuántica*. Editorial AC, 1977.

[4] Ira N. Levine. *Espectroscopia Molecular*. Editorial AC, 1980.

[5] 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.

Tabla 34: Tiempo de ejecución, en segundos, en las distintas partes del programa. En Origin 2000.

Procesadores:	1	2	3	4	5	6	7	8
	DVR=30, PM2D=20000, EC3D=15000							
2D	5.35	3.09	2.07	1.63	1.36	1.15	1.05	0.92
G3D	23.83	11.79	7.83	5.72	4.84	4.18	3.58	3.11
D3D	3.63	3.97	4.03	3.47	3.74	3.52	3.92	3.78
Total	32.81	18.85	13.93	10.82	9.94	8.85	8.55	7.81
	DVR=40, PM2D=25000, EC3D=20000							
2D	61.88	32.45	22.69	18.17	15.55	15.03	13.41	14.19
G3D	189.18	92.35	61.97	47.97	39.43	32.02	27.87	25.31
D3D	19.79	17.86	17.42	17.85	18.76	16.07	15.92	16.74
Total	270.85	142.66	102.08	83.99	73.12	63.12	57.20	56.24

[6] Z. Bačic y J. C. Light. Theoretical Methods for Rovibrational States of Floppy Molecules. *Annu. Re. Phys. Chem.*, 40:469, 1989.

[7] J. Vives. Determinación discreta y análisis estadístico del espectro vibracional del ion FHF^-. Facultad de Química. Universidad de Murcia. Tesina. 1998.

[8] 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, 1995.

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

[10] Oriol Farré, Luis Gregoris y Jordi López. Ejemplos de Paralelización con *Power Challenge*. Centro Europeo de Paralelismo de Barcelona.

[11] MIPSpro Fortran 77 Programmer's Guide. Silicon Graphics.

[12] Jack J. Dongarra, Hans W. Meuer y Erich Strohmaier. TOP500 Supercomputer Sites. Technical Report RUM 55/98, Mannheim University, 1998.

[13] Wilfried N. Gansterer, Dieter F. Kvasnicka y Christoph W. Ueberhuber. Multi-sweep Algorithms for the Symmetric Eigenproblem. In José M. L. M. Palma, Jack Dongarra y Vicente Hernández, editor, *Vector and Parallel Processing-VECPAR'98.*, number 1573 in Lecture Notes in Computer Science, pages 20–28. Springer, 1999.

[14] Aho, Hoptcroft y Ullman. *Estructuras de datos y algoritmos*. Adison-Wesley, 1988.

[15] J. Demmel y K. Stanley. The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers. 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 528–533. SIAM, 1995.

[16] Adolfo Bastida, Domingo Giménez, Alberto Requena y José Zúñiga. Computation of vibrational DVR eigenvalues on Origin 2000. In *Proceedings of Fifth Cray/SGI MPP European Workshop, Bologna, Italy, 8-9 september 1999*, 1999.