ANÁLISIS DE RENDIMIENTO DE UN ALGORITMO DE DIAGONALIZACIÓN DE MATRICES POR EL MÉTODO DE JACOBI SOBRE UNA ARQUITECTURAMULTICORE

Trabajo Final presentado para obtener el grado de Especialista en Cómputo de Altas Prestaciones y Tecnología GRID

Autor: Lic. Victoria María Sanz.

Director: Ing. Armando De Giusti.

Co-Director: Dr. Marcelo Naiouf

FACULTAD DE INFORMÁTICA - UNIVERSIDAD NACIONAL DE LA PLATA

Julio de 2012

Este trabajo fue aceptado para su publicación en los Proceedings de la Conferencia PDPTA 2012 (The 2012 International ConferenceonParallel and DistributedProcessingTechniques and Applications).

ÍNDICE

OBJETIVO

El objetivo general de este trabajo es mostrar la aceleración en el tiempo de cómputo que se obtiene al paralelizar el algoritmo de diagonalización de matrices simétricas por el método de Jacobi, de forma de aprovechar el paralelismo a nivel de thread que provee la arquitectura multicore actual.

Los temas a abordar abarcan el análisis del problema, el estudio de distintas implementaciones del algoritmo secuencial y optimizaciones posibles, la adaptación de dicho algoritmo para hacer uso de una implementación de la API BLAS (Basic Linear Algebra Subprograms) optimizada para la arquitectura subyacente, y la implementación de un algoritmo paralelo utilizando la herramienta de programación sobre memoria compartida OpenMP.

El método de Jacobi para diagonalizar matrices simétricas tiene aplicaciones en áreas como biometría, visión artificial, procesamiento digital de señales, entre otros. A medida que el volumen de datos de entrada se incrementa, la cantidad de tiempo requerido para el cómputo aumenta en forma significativa. La combinación de librerías de álgebra lineal optimizadas para la arquitectura subyacente, junto con la potencia que brinda un multicore y una herramienta adecuada de programación paralela para dicha arquitectura permitirá reducir el tiempo de ejecución.

Este trabajo pretende aportar un análisis del rendimiento (speedup, eficiencia) obtenido por el algoritmo paralelo propuesto sobre una arquitectura multicore, a medida que se incrementa el volumen de datos de entrada (tamaño de la matriz) y al aumentar la cantidad de threads /cores.

INTRODUCCIÓN

La demanda de potencia de cómputo por gran parte de las aplicaciones científicas se ha incrementado de modo tal que es esencial su resolución sobre una plataforma paralela. Si bien las arquitecturas cluster de *singlecore* se han vuelto habituales debido a la relación costo/rendimiento respecto a los grandes sistemas multiprocesador de memoria compartida, hoy en día es común poseer una máquina con más de un procesador.

La arquitectura multicore surge como respuesta a la dificultad de seguir aumentando la velocidad de los procesadores *singlecore* por problemas térmicos y energéticos. Un procesador multicore integra en un mismo chip 2 o más cores, por lo que las aplicaciones deben adaptarse para poder explotar el paralelismo a nivel de thread que provee dicha arquitectura [1]. Asimismo, se puede considerar tener varias máquinas multicore conectadas en red, lo que abre la posibilidad de contar con clusters con una gran cantidad de procesadores.

El método de Jacobi para diagonalizar matrices simétricas tiene aplicaciones en áreas como biometría, visión artificial, procesamiento digital de señales, entre otros[2][3][4]. A medida que el volumen de datos de entrada se incrementa, la cantidad de tiempo requerido para el cómputo aumenta en forma significativa. La combinación de librerías de álgebra lineal optimizadas para la arquitectura subyacente, junto con la potencia que brinda un multicore y una herramienta adecuada de programación paralela para dicha arquitectura permitirá reducir el tiempo de ejecución.

Dos de los puntos principales en el análisis de performance de un sistema paralelo son el *factor de Speedup*(*Sp*) [5]y la *Eficiencia* (*E*) que relaciona el Speedup con el número de procesadores (*P*) utilizados [6]. La *Escalabilidad* es un tercer factor muy significativo en las aplicaciones paralelas: normalmente los problemas “escalan”, es decir aumenta el volumen de trabajo a realizar, y también las arquitecturas que se utilizan pueden “escalar” incrementando los procesadores utilizados. Es de interés investigar el efecto de escalar trabajo y/o procesadores sobre el rendimiento de los algoritmos paralelos, considerando *Sp y E.* Se dice que un sistema paralelo es escalable si puede mantenerse una eficiencia constante al aumentar el trabajo y los procesadores [5][7].

El propósito de este trabajo es implementar un algoritmo paralelo para diagonalizar matrices por el método de Jacobi utilizando la herramienta de programación sobre memoria compartida OpenMP [8] y la librería BLAS de Álgebra lineal [9], de modo de aprovechar la potencia de cómputo de una máquina multicore.

Por último, se presentan pruebas experimentales y se realiza un análisis del rendimiento obtenido (speedup, eficiencia) a medida que escala el tamaño de la matriz y la cantidad de threads/cores utilizados.

Capítulo 1: Método de Jacobi para la diagonalización de matrices

En este capítulo se describe el método de Jacobi para diagonalizar matrices simétricas de números reales,comenzando por el caso particular de diagonalización de una matriz de dimensión 2x2 y finalizando en el caso general de diagonalización de una matriz de dimensión nxn.

1.1 Definición

El problema de diagonalización de una matriz simétrica *S* consiste en encontrar una matriz ortogonal *X* tal que haga que S se reduzca a la forma diagonal. El método de Jacobi permite encontrar la matriz *X* tal que:

$$X^T S\, X = D$$

Los elementos de la diagonal de la matriz *D* son denominados los *autovalores* de *S*, y las columnas de la matriz *X* los *autovectores* de *S*.

1.2 Caso particular: diagonalización de una matriz simétrica de 2x2

Si *S* es una matriz simétrica de dimensiones 2x2 (Figura 1.1.a), se puede proceder a diagonalizar la misma mediante la matriz ortogonal *X* que se muestra en la Figura 1.1.b. Dicha matriz es conocida como *matriz de rotación*.

$$S = \begin{pmatrix} s_{0,0} & s_{0,1} \\ s_{1,0} & s_{1,1} \end{pmatrix} \quad X = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$$

a **b**

Figura 1.1. a) Matriz de dimensiones 2x2, para ser simétrica sus elementos $s_{1,0}$ y $s_{0,1}$deben ser iguales. **b)** Matriz ortogonal X (conocida como matriz de rotación).

Dado que el objetivo del método es obtener *D*, los valores de coseno α y seno α deben ser tal que se anulen los elementos fuera de la diagonal de *D*, es decir:

$$D_{0,1} = D_{1,0} = s_{0,0}\sin\alpha\cos\alpha - s_{0,1}\sin^2\alpha + s_{0,1}\cos^2\alpha - s_{1,1}\sin\alpha\cos\alpha = 0$$

En la Figura 1.2 se muestran las fórmulas para obtener los valores de coseno α y seno α, que surgen a partir de despejar la ecuación anterior.

$$\cos\alpha = \sqrt{\frac{1}{1+\tan^2\alpha}} \quad \sin\alpha = \cos\alpha\tan\alpha \quad \tan\alpha = \frac{sign(\theta)}{|\theta| + \sqrt{1+\theta^2}} \quad \theta = \frac{s_{1,1} - s_{0,0}}{2s_{0,1}}$$

Figura 1.2. Cálculo de los valores seno α y coseno α

Los valores en la diagonal de la matriz D, se calculan de la siguiente manera:

$$D_{0,0} = s_{0,0}\cos^2\alpha - 2s_{0,1}\sin\alpha\cos\alpha + s_{1,1}\sin^2\alpha$$

$$D_{1,1} = s_{0,0}\sin^2\alpha + 2s_{0,1}\sin\alpha\cos\alpha + s_{1,1}\cos^2\alpha$$

1.3 Caso general: diagonalización de una matriz simétrica de nxn

El método general de diagonalización[10] mantiene una matriz *X* inicializada como la matriz identidad, donde quedarán los *autovectores* de *S*, y realiza una serie de etapasque actualizan *S* y *X* hasta que el elemento máximo del triángulo superior de *S* sea menor a un umbral seleccionado.

En cada etapa, para cada elemento $s_{p,q}$distinto de cero del triángulo superior se realiza una rotación en el plano diseñada para anular dicho elemento. La matriz de rotación J_i(Figura 1.3), donde i indica que se va a realizar la i-ésima transformación, tendrá los valores coseno α, seno α, -seno α y coseno α en las posiciones $j_{p,p}$, $j_{p,q}$, $j_{q,p}$, y $j_{q,q}$respectivamente, calculados a partir de los valores $s_{p,p}$, $s_{q,q}$y $s_{p,q}$; tendrá además valores 1 en la diagonal, y 0 en los elementos restantes.

$$J_i = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cos\alpha & \cdots & \sin\alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & -\sin\alpha & \cdots & \cos\alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix}$$

Figura 1.3. Matriz de rotación J para el caso general de diagonalización.

Los pasos a seguir para realizar cada rotación son:

1. Calcular los elementos $s_{p,p}$ y $s_{q,q}$ de la siguiente manera:
   $s_{p,p} = s'_{p,p}\cos^2\alpha - 2\, s'_{p,q}\text{sen}\alpha\, \cos\alpha + s'_{q,q}\, \text{sen}^2\alpha$
   $s_{q,q} = s'_{p,p}\, \text{sen}^2\alpha + 2\, s'_{p,q}\text{sen}\alpha\, \cos\alpha + s'_{q,q}\cos^2\alpha$
   Donde $s'_{p,p}$, $s'_{q,q}$ y $s'_{p,q}$ son valores de la matriz *S* antes de realizar esta rotación.
2. Anular $s_{p,q}$ y $s_{q,p}$.
3. Reemplazar el valor de la matriz *S* por el resultado de la operación $J^T\, S\, J$ (sin tener en cuenta las posiciones actualizadas los pasos previos).
4. Reemplazar el valor de la matriz *X* por el resultado de la operación *X J*.

Así las matrices J_i^T y J_i se irán multiplicando por *S,* actualizando esta última hasta que alcance su forma diagonal. La matriz *X* es el resultado del producto de las matrices J_i, y X^T es el resultado de multiplicar las matrices J_i^T (Figura 1.4).

$$\ldots J_3^T [\, J_2^T [\, J_1^T S_{original}\, J_1]\, J_2]\, J_3 \ldots = S_{diagonal}$$

$$\underbrace{(\cdots J_3^T\, J_2^T\, J_1^T\ )}_{X^T}\, S_{original}\, \underbrace{(\ J_1\ J_2\ J_3\ \cdots\ )}_{X} = S_{diagonal}$$

Figura 1.4. Multiplicación entre S y las matrices de rotación para obtener la matriz diagonalizada.

Capítulo 2: Arquitectura multicore y herramienta para programación paralela sobre memoria compartida (OpenMP)

En este capítulo se estudian las razones que llevaron al surgimiento de la arquitectura multicore y las características principales de la herramienta de programación sobre memoria compartida OpenMP.

2.1 Evolución hacia la arquitectura multicore

Durante muchos años el incremento en la frecuencia del reloj, la incorporación de cachés y pequeñas modificaciones en el código de la aplicación para utilizarlas en forma sensata, junto con cambios introducidos en la arquitectura para explotar el paralelismo a nivel de instrucción ó *IPL* (*pipelining*, ejecución fuera de orden, predicción de saltos, ejecución especulativa, etc.) provocaron una mejora en el rendimiento de las aplicaciones. [11]

Sin embargo se ha alcanzado un límite: problemas térmicos y energéticos imposibilitaron continuar aumentando la frecuencia del reloj; la velocidad de la memoria no se incrementóal ritmo que lo hizo la velocidad de los microprocesadores; y las técnicas para explotar el *IPL* pueden no ajustarse bien para aplicaciones cuyo código es difícil de predecir. Lo anterior ha motivado la búsqueda de mejoras en rendimiento a partir de explotar el paralelismo inherente de algunas aplicaciones mediante arquitecturas con multithreading y multiprocesadores [12].

La arquitectura multicore, también conocida como *CMP*(*Chip Multiprocessor*), es un multiprocesador que incluye en un único chip 2 o más procesadores llamados *core* o *núcleo*, teniendo un único socket para la conexión del chip en el motherboard. Dependiendo del modelo,

los cores podrán agruparse de forma de compartir niveles de memoria caché (L2 ó L3). Hoy en día la mayoría de los sistemas multicore son homogéneos, es decir todos sus cores son idénticos. La Figura 2.1 muestra un Quad-core similar al usado para las pruebas realizadas en este trabajo, conformado por 4 cores que comparten una caché L2 de 6MB entre pares [13].

Con la introducción de esta tecnología, las aplicaciones paralelizadas podrán manifestar una mejora en su rendimiento que dependerá del algoritmo en sí, de su componente secuencial, del overhead de comunicación y sincronización entre los procesos/hilos, entre otros factores.

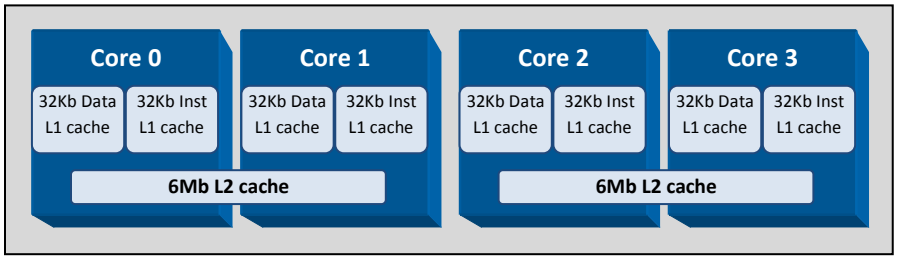


Figura 2.1 Quad-core: 4 cores físicos que comparten la caché L2 entre pares.

2.2 OpenMP: herramienta para programación sobre memoria compartida

OpenMP [8][14] es una API que reúne una colección de directivas, librerías y variables de entornopara la programación de aplicaciones paralelas sobre arquitecturas con memoria compartida, disponible para los lenguajes C, C++ y Fortran. Es una herramienta multiplataforma ya que posee versiones para diversas arquitecturas y sistemas operativos, haciendo que las aplicaciones paralelas que la utilizan sean fácilmente portables.

En general, el programa paralelo surge a partir de añadir simplemente directivas al programa secuencial que permiten: crear un conjunto de hilos, repartir tareas paralelas independientes entre ellos y sincronizarlos para el acceso de datos compartidos.

OpenMP utiliza el modelo *fork-join*, en el cual el programa comienza ejecutándose como un único hilo (llamado *maestro*), cuando se encuentra la primera construcción paralela se crea un conjunto de hilos (cada uno ejecutará el mismo código) los cuales podrán repartirse el trabajo y/o coordinarse para el uso de datos compartidos a través del uso de directivas para tal fin, y al terminar la región paralela continúa la ejecución el hilo maestro. La Figura 2.2 muestra un programa con 3 regiones paralelas, en cada región paralela se puede visualizar el modelo *fork-join*.

Los hilosse ejecutarán dentro del mismo espacio de direcciones y podrán compartir el acceso a variables declaradas en el mismo; también se permite que una variable sea designada como

privada a un hilo, en este caso cada hilo tendrá una copia que usará mientras dure la región paralela.

Figura 2.2. Modelo Fork – Join. Tres construcciones paralelas dentro del mismo programa.

Entre las directivas más importantes se encuentran:

- Constructor para la creación de regiones paralelas (directiva *parallel*): esta directiva permite crear un conjunto de hilos, tantos como indique la variable de entorno OMP_NUM_THREADS, que ejecutarán el mismo código (siguiendo el modelo SPMD – *Single ProgramMultiple Data*).
- Constructores de reparto de trabajo: una vez creados los hilos (dentro de una directiva *parallel*), los mismos podrán repartirse el trabajo a realizar. OpenMP soporta tanto Paralelismo a nivel de datos (directiva *For*) y Paralelismo funcional (directiva *Section)*. Estas directivas no crean nuevos hilos, y tienen una barrera implícita por defecto al final.
    - ✓ Directiva *For*: se utiliza para dividir las iteraciones de un bucle entre los hilos que están actualmente activos.
    - ✓ Directiva*Section***:** permite declarar un conjunto de funciones ó secciones, y que se asigne un hilo para la ejecución de cada sección.
    - ✓ Directiva*Single***:** permite asignar a un único hilo del conjunto una tarea que debe ser realizada sólo una vez, mientras que los demás esperarán en la barrera implícita. La especificación de OpenMP no establece cuál será el hilo que ejecute el bloque *Single.*
- Constructores de sincronización
    - ✓ Directiva*Master:* define una tarea que ejecutará sóloel hilo *maestro*, mientras los demás continúan la ejecución.
    - ✓ Directiva*Barrier:* permite sincronizar a todos los hilos de un grupo, haciendo que se esperen en un punto del programa hasta que todos lo hayan alcanzado.

Los constructores antes mencionados pueden ir acompañados de una serie de *cláusulas*, que varían según la directiva en sí. Para las directivas que poseen una barrera implícita al final, la cláusula *nowait*permite que el hilo continúe la ejecución, sin esperar a los demás; otras cláusulas

permiten delimitar el ámbito de las variables, es decir especificar si serán privadas o compartidas; la directiva *For* permite cláusulas para especificar cómo se repartirán las iteraciones del *For* (por ejemplo: en forma estática o dinámica) y operaciones de reducción,entre otras.

Capítulo 3: Implementaciones de algoritmos dediagonalizaciónde matrices por el método de Jacobi

En este capítulo se estudian las distintas implementaciones secuenciales realizadas del método de diagonalización (una de ellas utilizando BLAS), y la implementación del algoritmo paralelo conBLAS y la herramienta de programación sobre memoria compartida OpenMP.

3.1 Implementaciones y software utilizado

Se implementaron diferentes algoritmos de diagonalización de matrices basados en el método de Jacobi:

- Cuatro versiones del algoritmo secuencial clásico: la primera es simplemente la implementación del método detallado en el Capítulo 1 Sección 3, mientras que las restantes varían la forma en que se almacenan las matrices sobre las que opera el algoritmo.
- Se adaptó el algoritmo secuencial clásico para trabajar con bloques de datos o submatrices, generando así dos versiones secuenciales adicionales (una de ellas utiliza la librería ATLAS BLAS [15] optimizada para acelerar el tiempo de cómputo de las operaciones de álgebra lineal).
- Se implementó un algoritmo paralelo, que se basa en el algoritmo secuencial por bloques con ATLAS BLAS, y utiliza la herramienta de programación paralela OpenMP para arquitecturas con memoria compartida.

En todos los casos el lenguaje de programación utilizado fue C, y se realizó la compilación con GCC sobre plataforma Linux.

3.2 Algoritmo Secuencial Clásico

El algoritmo para resolver el problema de la diagonalización (detallado en el Capítulo 1 Sección 3) no requiere tener almacenada la matriz J_i cuando se realiza una rotación. Las actualizaciones realizadas en *S* y *X* por los pasos 3 y 4 se remplazan por las siguientes operaciones:

Paso 3:

$$\mathbf{s_{i,p}} = \mathbf{s_{p,i}} = s'_{i,p}\cos\alpha - s'_{i,q}\text{sen}\alpha \text{ parai} \neq p,\ i \neq q,\ i=0..n-1$$
$$\mathbf{s_{i,q}} = \mathbf{s_{q,i}} = s'_{i,q}\cos\alpha + s'_{i,p}\text{sen}\alpha \text{ parai} \neq p,\ i \neq q,\ i=0..n-1$$

Paso 4:

$$\mathbf{x_{i,p}} = x'_{i,p}\cos\alpha - x'_{i,q}\text{sen}\alpha \text{ parai} = 0..n-1$$
$$\mathbf{x_{i,q}} = x'_{i,q}\cos\alpha + x'_{i,p}\text{sen}\alpha \text{ parai} = 0..n-1$$

Donde $s'_{i,p}$, $s'_{i,q}$, $x'_{i,p}$ y $x'_{i,q}$ son valores de la matriz *S* y *X* antes de realizar esta rotación.

Es decir, el paso 3 solamente realiza actualizaciones sobre las filas y columnas *p* y *q* de *S*, mientras que el paso 4 actualiza únicamente las columnas *p* y *q* de *X*.

La Figura 3.1 muestra el algoritmo clásico de diagonalización de matrices por el método de Jacobi.

Entrada: *matriz simétrica de números reales S, precisión*
Salida: *S, X*
X = 𝕀 *//inicializa X como la matriz identidad*
Mientras(abs(max(triangulo_superior(S))) > precisión)
Para p=0..n-1 *// para cada $s_{p,q}$ del triángulo superior*
Para q= p+1 ..n-1
Si ($s_{p,q} \neq 0$) *// si $s_{p,q} \neq 0$ se procede a anular dicho elemento*
1. $\mathbf{s_{p,p}} = s'_{p,p}{}^{2}\cos\alpha - 2*s'_{p,q}*\text{sen}\alpha*\cos\alpha + s'_{q,q}{}^{2}*\text{sen}\alpha$
$\mathbf{s_{q,q}} = s'_{p,p}{}^{2}*\text{sen}\alpha + 2*s'_{p,q}*\text{sen}\alpha*\cos\alpha + s'_{q,q}{}^{2}*\cos\alpha$
2. Anular $s_{p,q}$; Anular $s_{q,p}$
3. Para i = 0 .. n-1 *//actualiza filas y columnas p y q de S*
Si ($i \neq p$ y $i \neq q$)
$\mathbf{s_{i,p}} = \mathbf{s_{p,i}} = s'_{i,p}*\cos\alpha - s'_{i,q}*\text{sen}\alpha$
$\mathbf{s_{i,q}} = \mathbf{s_{q,i}} = s'_{i,q}*\cos\alpha + s'_{i,p}*\text{sen}\alpha$
4. Para i = 0 .. n-1 *//actualiza columnas de X*
$\mathbf{x_{i,p}} = x'_{i,p}*\cos\alpha - x'_{i,q}*\text{sen}\alpha$
$\mathbf{x_{i,q}} = x'_{i,q}*\cos\alpha + x'_{i,p}*\text{sen}\alpha$

Figura 3.1. Algoritmo clásico de diagonalización de matrices por el método de Jacobi.

3.2.1 Optimizaciones

Se pueden realizar dos optimizaciones respecto al almacenamiento de las matrices *S* y *X* en memoria:

- Dado que *S* es una matriz simétrica, se pueden almacenar sólo los elementos correspondientes al triángulo superior, siendo en total *n(n+1)/2* elementos, donde n es la dimensión de filas y columnas de *S*. De esta manera se evitarán operaciones realizadas por simetría.

  El algoritmo planteado en la Figura 3.1 deberá ser modificado en la forma de acceso a los elementos, ya que si bien la matriz *S* se sigue almacenando en memoria por filas[1] (es decir los elementos de la fila 0 se encuentran consecutivos, luego aquellos de la fila 1, así siguiendo), la cantidad de elementos de las filas varía (la fila i, donde i=0..n-1, tendrá n-i elementos).

  Otra modificación a aplicar al algoritmo de la Figura 3.1 es la forma de actualizar la matriz *S* en el paso 3: si se guarda sólo el triángulo superior de *S*, las filas y columnas *p* y *q* no se almacenan enteras, por lo que la actualización de S deberá comenzar por las columnas *p* y *q* desde i = 0.. p-1 modificando $s_{i,p}$ y $s_{i,q}$(paso a), luego desde i=p+1..q-1 se actualizarán los elementos $s_{p,i}$ y $s_{i,q}$ (paso b) y por último desde i=q+1..n-1 se actualizarán $s_{p,i}$ y $s_{q,i}$(paso c). La Figura 3.2 muestra el procedimiento de actualización de la matriz *S* almacenando sólo sus elementos del triángulo superior, dondelos elementos actualizados por los 3 pasos anteriores se encuentran resaltados en distinto color.

$$S = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} & s_{0,5} & s_{0,6} & s_{0,7} \\ & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} & s_{1,5} & s_{1,6} & s_{1,7} \\ & & s_{2,2} & s_{2,3} & s_{2,4} & s_{2,5} & s_{2,6} & s_{2,7} \\ & & & s_{3,3} & s_{3,4} & s_{3,5} & s_{3,6} & s_{3,7} \\ & & & & s_{4,4} & s_{4,5} & s_{4,6} & s_{4,7} \\ & & & & & s_{5,5} & s_{5,6} & s_{5,7} \\ & & & & & & s_{6,6} & s_{6,7} \\ & & & & & & & s_{7,7} \end{pmatrix}$$

Figura 3.2. Actualización de la matriz S, cuando se almacena solo el triángulo superior.

[1]Rowmajororder es el método utilizado por C para almacenar arreglos multidimensionales en memoria.

- Al ser que *X* se accede siempre por columnas, se obtendrá mejor rendimiento si se almacenan en memoria sus elementos ordenados por columna[2] (es decir, primero estarán todos los elementos de la columna 0consecutivos, luego aquellos elementos de la columna 1, así siguiendo). La mejora en rendimiento se debe a que cada vez que se accede a memoria para buscar un elemento de *X*, se traslada un bloque de elementos consecutivos a la memoria caché los cuales se utilizarán en corto plazo. Esto se conoce como optimización por *localidad espacial*.

3.3 Algoritmo Secuencial por Bloques

El desarrollo de nuevas arquitecturas lleva a implementar nuevas técnicas para que los algoritmos se adapten a la plataforma subyacente, y así obtener mejor rendimiento. El algoritmo clásico de diagonalización por el método Jacobi puede adaptarse para hacer uso de librerías optimizadas de álgebra lineal existentes.

Con el fin de utilizar las operaciones matriciales definidas por el nivel 3 de BLAS, se implementó un algoritmo secuencial para diagonalizar matrices por el método de Jacobi que opera sobre bloques de elementos, el cual se utilizará como base para el algoritmo paralelo, y por esa razón la matriz *S* se almacena completa.

Este algoritmo considera que la matriz *S* es de dimensión n = Nxr (Figura 3.3), donde N es la cantidad de bloques por dimensión y r es la dimensión de cada bloque. El algoritmo es similar al clásico. Cada elemento $S_{P,Q}$, donde P,Q=0..N-1, denotará un bloque. Dentro del bloque los elementos se referenciarán con la notacións$_{p,q}$, donde p,q = 0..r-1 [16].

$$S = \begin{pmatrix} S_{0,0} & \cdots & S_{0,N-1} \\ \vdots & \ddots & \vdots \\ S_{N-1,0} & \cdots & S_{N-1,N-1} \end{pmatrix}$$

Figura 3.3. Matriz S, de dimensión n=N*r.

El método mantiene una matriz *X* inicializada como la matriz identidad, organizada por bloques al igual que *S*, donde quedarán los *autovectores* de *S*. El algoritmo realiza una serie de etapas que actualizan *S* y *X*hasta que el elemento máximo del triángulo superior de *S* sea menor a un umbral seleccionado.

Las operaciones de una etapa consisten en anular cada bloque $S_{P,Q}$del triángulo superior de bloques de *S*. Los pasos a seguir para realizar cada rotación son:

[2]Columnmajororder es uno de los métodos utilizados por lenguajes como Fortran para almacenar arreglos multidimensionales en memoria.

1. Calcular Jacobi con el método clásico sobre la matriz de 2x2 bloques conformada por S_{PP}, S_{PQ}, S_{QP} y S_{QQ}. Los autovectores quedarán en una matriz *auxX*de 2x2 bloques.
2. Calcular la traspuesta de auxX, llamémosla *traspX*.
3. Hacer rotaciones sobre la matriz S
    a. $S_{I,P} = S'_{I,P} * auxX_{0,0} + S'_{I,Q} * auxX_{1,0}$ para I = 0..N-1, I ≠ P ≠ Q
    b. $S_{I,Q} = S'_{I,P} * auxX_{0,1} + S'_{I,Q} * auxX_{1,1}$ para I = 0..N-1, I ≠ P ≠ Q
    c. $S_{P,I} = trasp_{0,0} * S'_{P,I} + trasp_{0,1} * S'_{Q,I}$ para I = 0..N-1, I ≠ P ≠ Q
    d. $S_{Q,I} = trasp_{1,0} * S'_{P,I} + trasp_{1,1} * S'_{Q,I}$ para I = 0..N-1, I ≠ P ≠ Q
4. Hacer rotaciones sobre la matriz X
    a. $X_{I,P} = X'_{I,P} * auxX_{0,0} + X'_{I,Q} * auxX_{1,0}$ para I = 0..N-1
    b. $X_{I,Q} = X'_{I,P} * auxX_{0,1} + X'_{I,Q} * auxX_{1,1}$ para I = 0..N-1

Donde $S'_{I,P}$, $S'_{I,Q}$, $S'_{P,I}$, $S'_{Q,I}$, $X'_{I,P}$ y $X'_{I,Q}$ son valores de la matriz *S* y *X* antes de realizar esta rotación.

Las operaciones 3.c y 3.d se pueden remplazar por la asignación a $S_{P,I}$y$S_{Q,I}$ de la transpuesta de $S_{I,P}$ y$S_{I,Q}$ respectivamente.

3.3.1 BLAS

BLAS (*Basic Linear Algebra Subprograms)* [9] es una API estándar que define un conjunto de rutinas comunes de álgebra lineal, que operan sobre vectores y matrices.

Existen diversas implementaciones de este estándar para lenguajes C y Fortran:

- Versiones no optimizadas (open source): Netlib BLAS (Fortran) y Netlib CBLAS.
- Versiones optimizadas (open source): ATLAS BLAS (C y Fortran), donde la optimización se realiza en tiempo de compilación de acuerdo a la máquina subyacente; y Goto BLAS optimizada en particular para procesadores como Intel Nehalem, Intel Atom y AMDOpteron.
- Versiones propietarias: son versiones que proveen especialmente los fabricantes de hardware (Intel, AMD, IBM, etc) optimizadas para una arquitectura particular. Por ejemplo: Intel MathKernel Library y AMD CoreMath Library.

Las operaciones de la especificación BLAS se dividen en niveles: el nivel 1 define operaciones sobre vectores (por ejemplo: producto escalar); las rutinas del nivel 2 realizan operaciones de tipo vector-matriz (por ejemplo: producto vector-matriz); las rutinas de nivel 3 realizan operaciones de tipo matriz-matriz (por ejemplo: producto de matrices).

3.3.2 Implementación del algoritmo por bloques utilizando BLAS

En el algoritmo por bloques, las actualizaciones en las matrices *S* y *X* (pasos 3.a, 3.b, 3.c, 3.d, 4.a y 4.b) involucran una secuencia de operaciones algebraicas sobre bloques o matrices de tamaño rxr y copias de bloques para mantener datos auxiliares que no se realizan en el algoritmo clásico. Para lograr una mejora en el rendimiento en las operaciones matriciales se utilizó la librería de álgebra

lineal ATLAS BLAS [9]. En particular, se emplearon las funciones *cblas_dgemm* para reducir el tiempo de procesamiento de la multiplicación y suma entre bloques, y *cblas_dcopy* para optimizar las copias de bloques.

3.4 Algoritmo Paralelo por Bloques usando BLAS

El algoritmo paralelo se basa en el hecho que se puede anular más de un elemento en forma conjunta en un paso del algoritmo, siempre que n >=4. Por ejemplo: si se tiene una matriz con n=4 y se quiere anular los elementos en las posiciones (0,2) y (1,3) a la vez, se debe elegir una matriz de rotación $J_{(0,2)(1,3)}$ tal que en las posiciones (0,0), (0,2), (2,0) y (2,2) tenga los valores cos α_1, sin α_1, -sin α_1 y cos α_1 calculados a partir de los elementos en $s_{0,0}$, $s_{0,2}$ y $s_{2,2}$, mientras que en las posiciones (1,1), (1,3), (3,1) y (3,3) debe tener los valores cos α_2, sin α_2, -sin α_2 y cos α_2 calculados a partir de los elementos $s_{1,1}$, $s_{1,3}$ y $s_{3,3}$ (Figura 3.4.c).

$$J_{0,2} = \begin{pmatrix} \cos\alpha_1 & 0 & \sin\alpha_1 & 0 \\ 0 & 0 & 0 & 0 \\ -\sin\alpha_1 & 0 & \cos\alpha_1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

a)

$$J_{1,3} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & \cos\alpha_2 & 0 & \sin\alpha_2 \\ 0 & 0 & 0 & 0 \\ 0 & -\sin\alpha_2 & 0 & \cos\alpha_2 \end{pmatrix}$$

b)

$$J_{(0,2)(1,3)} = \begin{pmatrix} \cos\alpha_1 & 0 & \sin\alpha_1 & 0 \\ 0 & \cos\alpha_2 & 0 & \sin\alpha_2 \\ -\sin\alpha_1 & 0 & \cos\alpha_1 & 0 \\ 0 & -\sin\alpha_2 & 0 & \cos\alpha_2 \end{pmatrix}$$

c)

Figura 3.4. a) Matriz de rotación para anular el elemento (0,2)
b) Matriz de rotación para anular el elemento (1,3)
c) Matriz de rotación J compuesta, para anular elementos (0,2) y (1,3).

La matriz de rotación compuesta de la Figura 3.4.c surge a partir de hacer un producto de las matrices de rotación que se usarían para anular un único elemento (Figura 3.4.a y Figura 3.4.b); además se destaca que si se tienen dos pares de coordenadas de los elementos que se quieren

anular (P,Q) y (P',Q'), con P≠Q≠P'≠Q', se puede comprobar que el producto de dichas matrices de rotación es conmutativo $J_{P,Q} * J_{P',Q'} = J_{P',Q'} * J_{P,Q}$. [17][18]

Dado que la matriz de entrada es de dimensión n, y los pares a anular en forma conjunta deben tener coordenadas distintas, solo se podrán anular $\lfloor n/2 \rfloor$elementos a la vez.

Cuando se aplica la etapa de actualización sobre *S* se observan dos grandes pasos:

- *Pre-multiplicación*: consiste en realizar el producto $J^T S$, lo que modificará sólo las columnas de *S*.
- *Post-multiplicación*: consiste en realizar el producto $S J$, lo que modificará sólo las filas de *S*.

La Figura 3.5.a muestra las actualizaciones realizadas por el paso de pre-multiplicación sobre una matriz simétrica *S* de dimensión 4x4 y utilizando la matriz de rotación de la Figura 3.4.c, mientras que la Figura 3.5.b muestra las actualizaciones realizadas por la post-multiplicación. Se resaltaron en distinto color aquellas modificaciones resultantes de anular el elemento (0,2) y (1,3).

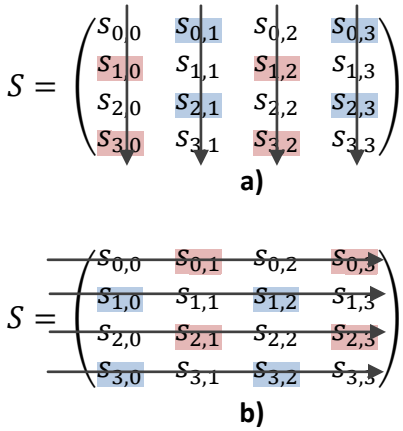


Figura 3.5. a) Actualizaciones en S por pre-multiplicación
b) Actualizaciones en S por post-multiplicación

De lo anterior se observa que:

- Dado que las posiciones a anular tienen índice distinto (P_i,Q_i), se puede hacer en paralelo el paso de algoritmo que actualiza el valor de las posiciones (P_i,P_i) (P_i,Q_i)(Q_i,P_i)(Q_i,Q_i).
- Todas las actualizaciones del paso pre-multiplicación se realizan sobre columnas distintas, por lo que las mismas se pueden hacer en paralelo.
- Todas las actualizaciones del paso post-multiplicación se realizan sobre filas distintas, por lo que las mismas se pueden hacer en paralelo.

El algoritmo paralelo de diagonalización de matrices propuesto se implementará en base al algoritmo secuencial por bloques planteado en el Capítulo 3 Sección 3.2, ya que es el que optimiza los tiempos.

La estrategia de paralelización consiste en realizar en simultáneo las rotaciones de cada etapa del algoritmo. La cantidad de tareas paralelas a ejecutar por etapa es *N/2*, dado que las mismas deben tener coordenadas diferentes, y se calcularán sus índices siguiendo la estrategia *ChessTournament*[19]. En caso de existir mayor cantidad de tareas que threads, se repartirán las mismas en forma equitativa.

Una tarea cuya coordenada es (P,Q) consiste en calcular los *autovectores* y *autovalores* de la submatriz de 2x2 bloques conformada por $S_{P,P}$, $S_{P,Q}$, $S_{Q,P}$, $S_{Q,Q}$ con el algoritmo secuencial clásico, en forma independiente. Luego se deben actualizar las matrices *S* y *X* de la siguiente manera:

- Rotaciones en *S*: como varios hilos pueden estar en esta etapa de actualización, es necesario que estos se esperen antes de comenzar (sincronización por barrera), luego actualizarán cada uno sus columnas (dado que cada hilo tiene coordenadas cuyos índices son disjuntos). Después, tras una segunda etapa de sincronización por barrera, actualizan sus filas de *S*. La primera sincronización evita que hilos rezagados actualizando filas de *S* entren en conflicto con otros hilos que estén próximos a actualizar las columnas de tareas siguientes. La segunda sincronización evita que surjan conflictos en caso que un hilo que este modificando sus columnas y otro estéactualizando sus filas.
- Rotaciones en *X*: estas rotaciones modificansólo las columnas, y dado que todas las tareas tienen coordenadas distintas es posible realizarlas en paralelo.

Luego de realizar las rotaciones, los hilos pueden proceder a completar su siguiente tarea, así hasta completar la etapa actual. A continuación, se calcula en forma paralela el máximo del triángulo superior de la matriz *S* y luego uno de los hilos calcula el conjunto de índices de las tareas paralelas para la siguiente etapa (según *ChessTournament*). Este último paso debe hacerse en forma secuencial debido a dependencias de datos.

En el próximo paso, cada hilo independientemente verifica la condición de terminación, a partir del valor máximo de *S* obtenido, y de acuerdo al resultado de la misma se termina el algoritmo o se procede a realizar una nueva etapa.

3.4.1 Estrategia ChessTournament

La estrategia “Torneo de Ajedrez” (*ChessTournament*) se utilizará para generar las coordenadas de las tareas que pueden calcularse en paralelo en cada etapa del algoritmo.

Para explicar cómo funciona el algoritmo se planteará un ejemplo: sea *S* la matriz de entrada de dimensión N=8, en cada etapa habrá 4 tareas paralelas y al principio sus coordenadas son (0,1) (2,3) (4,5) (6,7). La generación de las coordenadas de las tareas paralelas para la próxima etapa surge de imaginar un torneo de ajedrez entre 8 jugadores, donde cada jugador juega contra cada uno de los restantes sólo una vez. Cuando terminan de jugar en una etapa, cada jugador se mueve a la mesa adyacente. Así si hay 8 jugadores, habrá 7 etapas o rondas de juego. [19]

La Figura 3.6 muestra las coordenadas de las tareas paralelas para cada una de las 7 etapas del algoritmo cuando la matriz de entrada tiene N=8.

Ronda 1:	*Ronda 2:*	*Ronda 3:*
0 2 4 6 1 3 5 7	0 1 2 4 3 5 7 6	0 3 1 2 5 7 6 4
Ronda 4:	***Ronda 5:***	***Ronda 6:***
0 5 3 1 7 6 4 2	0 7 5 3 6 4 2 1	0 6 7 5 4 2 1 3
	Ronda 7:	
	0 4 6 7 2 1 3 5	

Figura 3.6. Coordenadas de tareas paralelas en cada etapa/ronda del algoritmo paralelo, cuando la matriz de entrada tiene N=8.

3.4.2 Implementación utilizando OpenMP

La implementación del algoritmo paralelo para resolver el método de Jacobi con OpenMP [8]carga la matriz de entrada *S*, e inicializa la matriz *X* en forma secuencial.

A continuación se crean un conjunto de hilos, tantos como cores provea la arquitectura multicore, quienes inicializan en forma paralela la estructura con los índices de las tareas a ejecutar en cada etapa.

En cada etapa del algoritmo, cada hilo tomará un conjunto consecutivo de índices de tareas paralelas (*directiva* de reparto de trabajo *For)*, y hará lo que se describe en la Sección 3.4 de este Capítulo. Una vez que se termina la ejecución de las tareas de una etapa, los hilos proceden a calcular el máximo del triángulo superior de S, aplicando una operación de reducción. Luego uno de ellos, actualiza el conjunto de índices de tareas paralelas para la próxima etapa, según *ChessTournament,* utilizando la *directiva single*.

Capítulo 4: Análisis de rendimiento

En este capítulo se estudian las métricas *Speedup* y *Eficiencia,* usadas comúnmente para evaluar el rendimiento de un sistema paralelo, junto con el concepto de escalabilidad. Además se enumeran las causas de *overhead* más comunes que provocan la degradación del rendimiento de un sistema paralelo.

A continuación se presentan las características del hardware a utilizar,se exponen las pruebas secuenciales realizadas con las distintas versiones del algoritmo clásico y las dos versiones del algoritmo por bloques, y posteriormente se muestran los resultados obtenidos por la ejecución del algoritmo paralelo.

Por último se realiza un análisis del speedup y la eficiencia alcanzada por el algoritmo paralelo a medida que se incrementa el tamaño de los datos de entrada y la cantidad de cores de la arquitectura.

4.1 Métricas: Speedup y Eficiencia

El factor de *Speedup absoluto*S_p mide la ganancia en rendimiento obtenida al resolver un problema en paralelo con respecto a su implementación secuencial, y se define como la relación entre el tiempo de ejecución del mejor algoritmo secuencial (T_s) y el tiempo de ejecución del algoritmo paralelo sobre una máquina con P procesadores (T_p)[5]. La Figura 4.1 muestra la fórmula para calcular el *Speedup*.En este caso se asume que la arquitectura de la máquina sobre la que se ejecutó el algoritmo secuencial es idéntica a la arquitectura de la máquina multiprocesador usada para ejecutar el algoritmo paralelo.

$$S_p = \frac{T_s}{T_p}$$

Figura 4.1: Speedup

Teóricamente, si la arquitectura paralela está conformada por *P* procesadores se esperaría obtener un *Speedup* entre 1 y *P.* En algunos casosel *Speedup* puede ser mayor a *P,* dando lugar al concepto de *Superlinealidad,* que ocurre usualmente en algunos algoritmos no determinísticos donde el trabajo realizado por el algoritmo paralelo es menor al realizado por el algoritmo secuencial, ó cuando debido a características de hardware el algoritmo secuencial se encuentra en desventaja respecto al algoritmo paralelo ejecutado sobre una máquina multiprocesador.

Normalmente el *Speedup* estará limitado por diversos factores, que deberán tenerse en cuenta con el objetivo de alcanzar el máximo rendimiento posible:

- El máximo grado de concurrencia que puede obtenerse de la aplicación paralela
- El componente secuencial del algoritmo que no puede ser resuelto en paralelo.
- Exceso de cómputo en el algoritmo paralelo: el algoritmo secuencial más rápido puede ser difícil o imposible de paralelizar, teniendo que implementar un algoritmo paralelo en base a una técnica deficiente. Asimismo, un algoritmo paralelo basado en técnicas óptimas puede presentar exceso de cálculo debido a que algunos resultados, que en el algoritmo secuencial podían ser reusados, no puedan serlodurante el cómputo paralelo debido a que están siendo generados por distintos procesadores, por lo que se deben calcular múltiples veces.
- *Overhead* de sincronización y comunicación entre procesos
- Desbalance de carga

La *Eficiencia* mide el uso efectivo de los recursos de cómputo y resulta una métrica de calidad y de costo del algoritmo paralelo. La misma se define como la relación entre el *Speedup* y el *Speedup óptimo*. En el caso de una arquitectura homogénea, el Speedup óptimo es *P (*el número de procesadores utilizados). La Figura 4.2 muestra la fórmula para calcular la Eficiencia.

$$E = \frac{S_p}{P}$$

Figura 4.2: Eficiencia

La Eficiencia varía entre 0 y 1, alcanzar valores cercanos a 1 significa que se logra un S_p cercano al óptimo *P*, y no siempre puede mantenerse al incrementar el tamaño los problemas, al incrementar el número de procesadores o al portar el algoritmo sobre otra arquitectura multiprocesador.

4.2 Escalabilidad

La *Escalabilidad* de un sistema paralelo es una medida de su capacidad para utilizar eficientemente un número creciente de procesadores.

Diversos autores han propuesto métodos para evaluar la escalabilidad de los sistemas paralelos. Un método representativo para arquitecturas homogéneas es el análisis de *isoeficiencia*, el cual plantea que un sistema es escalable si es posible mantener la eficiencia del sistema en un valor fijo al aumentar el número de procesadores (escalar la arquitectura) y el tamaño del problema (escalar el problema). La *función de isoeficiencia* indica cuánto tiene que aumentar el tamaño del problema para poder incluir más procesadores sin que la eficiencia del sistema se vea afectada. [20]

El modelo de *isoeficiencia* se basa en las siguientes consideraciones:

- Si un sistema paralelo es usado para resolver un problema de tamaño fijo, el *Speedup* no continúa creciendo a medida que se incrementa el número de procesadores (debido a los factores mencionados en la sección anterior), provocando la caída de la *Eficiencia*. Esto sucede porque el tiempo de overhead (T_o) se incrementa a medida que P crece.
- En algunos sistemas paralelos, la eficiencia aumenta al escalar el tamaño del problema manteniendo el número de procesadores constante. En estos casos el aumento en la *Eficiencia* se debe a que $T_o$crece en menor medida que el tiempo de resolución del problema.

Para el caso de estudio propuesto, escalar el problema significa aumentar el tamaño de la matriz, lo cual causará un crecimiento importante de la cantidad de operaciones a realizar incrementando significativamente el tiempo de respuesta, mientras que la arquitectura se escala aumentando la cantidad de cores usados para resolver el problema en paralelo.

4.3 Resultados experimentales

En esta Sección se presentan los tiempos de ejecución en segundos de los distintos algoritmos. Las pruebas a comparar en las Secciones 3.2 y 3.3 de este Capítulo se realizaron sobre matrices con datos iguales e igual precisión (0.0001), ya que los tiempos de convergencia dependen de los datos de entrada y de la precisión seleccionada, lo mismo ocurre en las pruebas de la Sección 3.4 que evalúa la escalabilidad del algoritmo paralelo.

La implementación de BLAS utilizada es *ATLAS BLAS*.

4.3.1 Hardware utilizado

Las pruebas se realizaron en una máquina con dos procesadores Intel QuadCoreXeon 5400 modelo E5405 2.0Ghz [13]. Cada core cuenta con una caché L1 de 32KB de datos y 32KB de instrucciones. Entre cada par de cores comparten una caché L2 de 6MB. La memoria disponible es de 10GB RAM.

4.3.2 Algoritmo Secuencial Clásico

Se implementaron 4 versiones que varían en cómo se almacena la matriz *S* y la matriz *X*:

- Versión 1: almacena *S* completa por filas y *X* por filas
- Versión 2: almacena *S* completa por filas y *X* por columnas. Optimización por localidad espacial en *X*.
- Versión 3: almacena sólo el triángulo superior de *S* y almacena *X* por filas. Optimización de almacenamiento de *S* y menor cantidad de operaciones por simetría.
- Versión 4: almacena sólo el triángulo superior de *S* y *X* por columnas.

La Tabla 4.1 muestra los tiempos de ejecución para cada una de las versiones del algoritmo clásico, y matrices con n=100,200,300,400,500,600,700,800,900,1000. Las versiones 2 y 3 mejoran los tiempos respecto a la versión 1, siendo la optimización realizada por la versión 3 la que produce mayor impacto. Por esta razón, al incorporar ambas optimizaciones a la versión 4, se logra reducir aún más los tiempos de ejecución.

Tabla 4.1. Tiempos algoritmo secuencial Jacobi clásico con distintas optimizaciones.

Dimensión / Versión	Versión 1	Versión 2	Versión 3	Versión 4
100	0,21	0,2	0,18	0,18
200	2,1	1,92	1,78	1,62
300	7,56	6,77	6,17	5,54
400	20,51	18,58	17,37	14,97
500	40,8	36,72	34,7	29,57
600	72,4	64,44	60,7	52,4
700	120,7	104,34	98,56	84,2
800	187,3	159,41	151,9	127,4
900	307,03	258,07	250,8	205
1000	387,52	323,5	318,5	256,42

4.3.3 Algoritmo Secuencial por Bloques

Para las pruebas de los algoritmos secuenciales por bloques se utilizaron las mismas matrices que en la Sección anterior, pero se organizan sus datos en bloques consecutivos, cuyo tamaño posible dependerá del tamaño de la matriz.

4.3.3.1 Algoritmo secuencial por bloques (sin BLAS)

El algoritmo por bloques que no utiliza BLAS no mejoró los tiempos obtenidos por el algoritmo clásico versión 4. De las pruebas que se visualizan en la Tabla 4.2, se observa que el mejor tiempo de respuesta se obtiene con el máximo tamaño de bloque posible. En estos casos el algoritmo por bloques se comporta en forma similar al algoritmo clásico, ya que realiza una única etapa en la cual obtendrá los *autovalores* y *autovectores* de la matriz de 2x2 bloques conformada por $S_{0,0}$, $S_{0,1}$,$S_{1,0}$ y $S_{1,1}$ aplicando el algoritmo clásico, obteniendo así los*autovalores* y *autovectores* de la matriz completa.

La razón por la cual el algoritmo por bloques sin BLAS no mejoró los tiempos se debe al tiempo extra que se utiliza para multiplicar y copiar bloques, y a medida que el tamaño del bloque es más chico, se deben hacer más de estas operaciones.

Tabla 4.2. Tiempos obtenidos por algoritmo secuencial por bloques sin BLAS.

Dimensión / Tamaño bloque	100	200	300	400	500	600	700	800	900	1000
5	0,5	4,29	15,92	37,54	72,87	129,72	198,97	337,1	468,61	578,75
10	0,52	4,05	14,46	33,27	65,12	108,82	172,14	282,6	401,9	549,71
25	0,57	4,78	14,66	36,09	67,01	121,98	189,6	278,54	428,09	530,05
50	0,21	4,94	18,7	41,81	82,4	135,84	224,11	324,8	446,6	650,55
75			18,8							
100		2,03		47,9		172,2		405,77		795,07
125					97,84					818,09
150			7,19			172,18			632,6	
175							319,66			
200				19,94				485,77		
225									702,07	
250					39,9					1004
300						69,65				
350							111,99			
400								194,47		
450									270,99	
500										387,25

4.3.3.2 Algoritmo secuencial por bloques usando BLAS

El algoritmo secuencial por bloques que hace uso de la librería ALTAS BLAS, mejoró los tiempos de los algoritmos secuenciales evaluados en las secciones anteriores. En la mayoría de los casos, el mejor tiempo se obtuvo con un tamaño de bloque 10, como se muestra en la Tabla 4.3.

Para analizar la causa de esta mejora se calculó la cantidad de etapas que realiza el algoritmo por bloques en sus dos versiones y el tiempo promedio que lleva realizar una rotación. Estas pruebas se ejecutaron para la matriz de tamaño 1000 con bloques 10 y 500.

Ambos algoritmos realizaron una única etapa para el tamaño de bloque 500, y 10 etapas cuando se utiliza bloque de tamaño 10. Esto demuestra que al disminuir el tamaño de bloque el algoritmo realizará más iteraciones para converger. Sin embargo, el tiempo promedio que tarda cada rotación difiere. Teniendo en cuenta que en cada etapa se realizan *N(N+1)/2* rotaciones, se observa que:

Tabla 4.3. Tiempos obtenidos por algoritmo secuencial por bloques con BLAS.

Dimensión / Tamaño bloque	100	200	300	400	500	600	700	800	900	1000
5	0,22	1,83	5,68	14,58	30,86	48,12	75,2	111,45	175,6	241,5
10	0,26	1,47	4,25	8,83	15,75	25,38	41,9	60,6	83,83	112,9
25	0,44	2,68	6,86	14,17	24,5	36,35	51,77	70,77	94,12	119,7
50	0,2	3,82	12,35	25,19	43,46	65,83	97,2	131,54	169,9	226,18
75			14,76							
100		1,95		38,37		118,44		247,5		427,22
125					78,36					508,5
150			6,87			153,86			446,86	
175							258,65			
200				19,34				390,27		
225									581,52	
250					38,48					811,84
300						67,05				
350							108,62			
400								189,45		
450									259,3	
500										371,03

- En el *algoritmo por bloques sin BLAS* el tiempo promedio de rotación para el bloque de tamaño 10 es 0,011. La cantidad de rotaciones realizadas por etapa es 5050. Por ello cada etapa demorará 56,4 segundos aproximadamente, lo que justifica el tiempo obtenido en la prueba.

En cambio el tiempo promedio de rotación para el bloque de tamaño 500 es 386,74. Este tiempo es similar al tiempo final obtenido, y al tiempo obtenido en el algoritmo clásico (versión 1). Esto se debe a que se realiza una etapa y una única rotación.

- En el *algoritmo por bloques con BLAS*, para el bloque de tamaño 10 el tiempo promedio de rotación es 0,00229; como se ejecutan 5050 rotaciones por etapa, cada etapa demorará aproximadamente 11,56 segundos, lo que justifica el tiempo obtenido en la prueba. Para el bloque de tamaño 500 el tiempo de iteración fue 370,815543.

La mejora en el algoritmo por bloques con BLAS se debe al uso de las funciones *cblas_dgemm* y *cblas_dcopy* al calcular las rotaciones.

El tamaño de bloque que optimiza los tiempos debe ser tal que calcular Jacobi Clásico para una matriz de 2x2 bloques no sea tan costoso, ya que se obtendrán tiempos similares al algoritmo clásico versión 1. Se debe buscar la relación justa, ya que si el bloque es muy pequeño por más que disminuya el tiempo promedio de rotación, hará aumentar la cantidad de rotaciones. Por ejemplo: para el algoritmo por bloques con BLAS, el tiempo promedio de rotación para la prueba con matriz de tamaño 1000 y bloque 5 es 0,001209. El algoritmo realizó 10 etapas y 20100 rotaciones por etapa. Cada etapa en promedio demoró 24,3 segundos, provocando un incremento en los tiempos.

4.3.4 Algoritmo paralelo por bloques usando BLAS

Como el análisis de escalabilidad fue realizado sobre 2, 4 y 8 hilos/cores, para las pruebas se utilizaron matrices cuyo tamaño es múltiplo de 2, así los hilos se repartirán las tareas proporcionalmente.

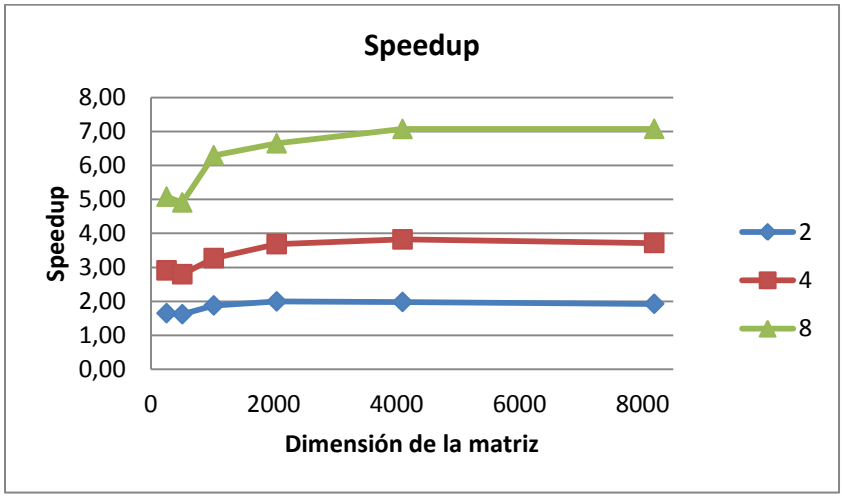
La Tabla 4.4 muestra los tiempos de ejecución del algoritmo secuencial por bloques que utiliza BLAS para dichas matrices, variando el tamaño del bloque entre los valores posibles. Se resalta la prueba que optimizó el tiempo para cada tamaño de matriz. La Tabla 4.5. muestra el speedup y la eficiencia obtenida en las pruebas paralelas.

Tabla 4.4. Tiempos obtenidos por el algoritmo secuencial por bloques con BLAS para matrices con n múltiplo de 2.

	Dimensión Matriz					
Tamaño Bloque	256	512	1024	2048	4096	8192
2	25,76	220,76	2070,03	17403,73		
4	4,86	38,54	364,5	3011,32		
8	2,59	17,28	136,36	1100,64	8685,8	
16	3	17,78	106,14	686,25	5097,1	39495,26
32	4,98	27,9	156,73	862,54	5493,24	37061,84
64	7,01	47,02	259,42	1254,09		30345,76
128	4,05	69,09	462,23	2448,01		53193,4
256		40,16	687,7	4590,44		
512			1483,56	26982,67		
1024				12963,09		

Tabla 4.5. Valores de Speedup y eficiencia a medida que escala el tamaño de la matriz e hilos/cores.

	Speedup			**Eficiencia**		
Dimensión matriz (bloque) / cantidad de tareas paralelas	2	4	8	2	4	8
256(8) – 16	1,65	2,91	5,08	0,82	0,73	0,63
512(8) – 32	1,62	2,80	4,91	0,81	0,70	0,61
1024(16) – 32	1,88	3,27	6,30	0,94	0,82	0,79
2048(16) – 64	2,00	3,68	6,65	1,00	0,92	0,83
4096(16) – 128	1,98	3,82	7,07	0,99	0,96	0,88
8192(64) – 64	1,92	3,72	7,07	0,96	0,93	0,88



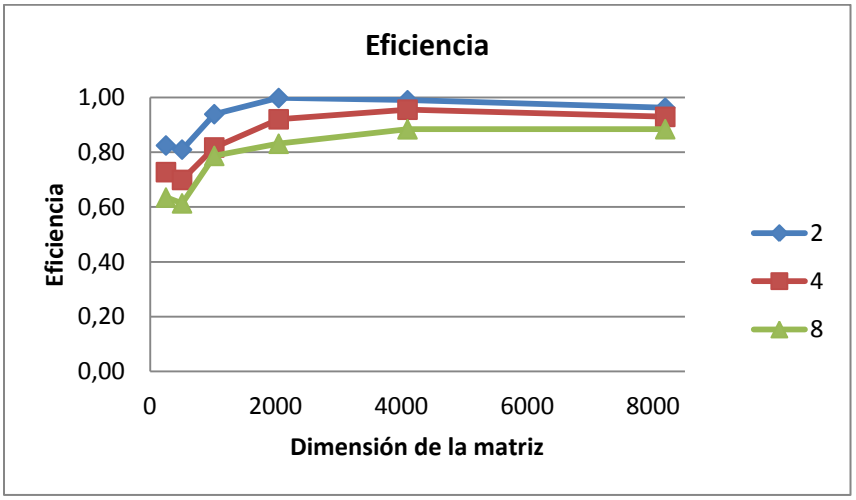


Figura 4.6.a) Speedup a medida que escala el tamaño de la matriz e hilos/cores **b)** eficiencia a medida que escala el tamaño de la matriz e hilos / cores.

De los resultados se observa que al mantener fijo el tamaño de la matriz y aumentar la cantidad de procesadores, el speedup obtenido es mejor, es decir se resuelve el problema en menor tiempo, pero esa mejora no mantiene la eficiencia constante. La disminución de la eficiencia se debe al overhead de la creación de hilos, sincronizaciones (barreras), y la porción secuencial del problema (en cada etapa se actualiza la estructura de tareas paralelas para la próxima vuelta según *ChessTournament*en forma secuencial).

Al escalar el problema manteniendo fijo el número de procesadores, tal como se ve en la Tabla 4.5 y la Figura 4.6, tanto el speedup como la eficiencia mejoran en general, dado que el overhead mencionado anteriormente tiene menor peso en el tiempo total de procesamiento.

Este comportamiento expuesto es propio de un sistema paralelo escalable, donde la eficiencia puede mantenerse en un valor constante incrementando simultáneamente el número de cores/procesadores y el tamaño del problema.

CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURO

Se presentaron distintas implementaciones secuenciales y una implementación paralela para resolver el problema de la diagonalización de matrices de modo de aprovechar la potencia brindada por una arquitectura multicore.

Se realizó un análisis de los tiempos de ejecución resultantes de las distintas implementaciones, observando un mejor rendimiento para los algoritmos que hacen uso de librerías optimizadas para álgebra lineal (ATLAS). Además, de los experimentos se observa la mejora en el rendimiento al paralelizar el algoritmo sobre una arquitectura multicore.

En los últimos años, las GPUs (GraphicsProcessingUnits) [21] han ganado importancia debido al alto rendimiento alcanzado en aplicaciones de propósito general. Una línea de trabajo futuro se basa en la migración del algoritmo de diagonalización de Jacobi para su ejecución sobre GPU, para luego hacer un estudio sistemático del rendimiento alcanzado a medida que se aumenta el tamaño del problema y el número de hilos. Asimismo se plantea realizar un análisis del consumo energético de este algoritmo paralelo sobre distintas arquitecturas multicore [22].

REFERENCIAS

[1] Olukotun K, Hammond L, Laudon J. *Chip Multiprocesor Architecture*. Synthesis Lecture on Computer Science, Morgan&Claypool; 2007.

[2] Turk M., Pentland A.*Eigenfaces for recognition*. Journal of Cognitive Neuroscience Vol 3, No. 1, pp- 71–86; 1991.

[3] Bravo Muñoz I. *Arquitectura basada en FPGAs para la detección de objetos en movimiento, utilizando visión computacional y técnicas PCA*. Tesis Doctoral, Universidad de Alcalá; 2007.

[4] Brent R., *Parallel Algorithms for Digital Signal Processing*. Numerical Linear Algebra. Digital Signal Processing and Parallel Algorithms, pp. 93-110. Springer, Heidelberg; 1991.

[5] Grama A., Gupta A., Karypis G., Kumar V. *Introduction to Parallel Computing*. Second Edition. Addison Wesley; 2003.

[6] Quiin M. J. *Parallel Computing: Theory and Practice*. McGraw-Hill Companies; 1993.

[7] Hwang K. *Advanced Computer Architecture*. Parallelism, Scalability, Programmability. McGraw Hill; 1993.

[8] *The OpenMP API specification.* http://openmp.org/wp/.

[9] *BLAS Basic Linear Algebra Subprograms*. http://www.netlib.org/blas/

[10] Rutishauser H. *The Jacobi Method for Real Symmetric Matrices*. Numer. Math. 9, 1-10; 1966.

[11] Hennessy J., Patterson D. *Computer Architecture: A Quantitative Approach* (5ta edición) Morgan Kaufmann Publishers; 2012.

[12] Akhter S., Roberts J. *Multicore Programming. Increasing Performance through SoftwareMulti-threading.* Intel Press; 2006.

[13] *Intel Product Specifications.* http://ark.intel.com/products/33079/Intel-Xeon-Processor-E5405-(12M-Cache-2_00-GHz-1333-MHz-FSB).

[14] Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R. *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers; 2001.

[15] *Automatically Tuned Linear Algebra Software (ATLAS)*http://math-atlas.sourceforge.net/

[16] Hansen E. On *Jacobi methods and block-jacobi methods for computing matrix eigenvalues*. Tesis Doctoral, Stanford; 1960

[17] Sameh A. *On Jacobi and Jacobi like algorithms for a parallel computer*. Mathematics of computation, Vol 25, No. 115, 1971.

[18] Shroff G. *A parallel algorithm for the eingenvalues and eigenvectors of a general complex matrix.* Research Institute for Advanced Computer Science. NASA Ames Research Center. Technical Report.; 1989

[19] Bischof C., Van Loan C. *Computing the singular value decomposition on a ring of array processors.* Proceedings of the IBM Europe Institute Workshop on Large Scale Eigenvalue Problems. Vol. 127, pp 51–66; 1986.

[20] Grama A., Gupta A., Kumar V. Isoefficiency: *Measuring the Scalability of Parallel Algorithms and Architectures.* IEEE Parallel and Distributed Computer, IEEE Computer Society Press; 1993.

[21] *General-Purpose Computation on Graphics Hardware* http://gpgpu.org/.

[22] Wu-chunFeng, XizhouFeng&RongGe. *Green Supercomputing Comes of Age*. Journal IT Professional. Vol. 10, No. 1, pp 17-23; 2008.