Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Implementación paralela híbrida heterogénea de la descomposición en valores singulares en matrices densas

Tesis que presenta

André Fabián Castellanos Aldama

para obtener el Grado de

Maestría en Ciencias en Computación

Directores de Tesis

Dr. Amilcar Meneses Viveros
Dr. Sergio Víctor Chapa Vergara

Ciudad de México **Agosto 2023**

Abstract

Singular value decomposition (SVD) in dense matrices is used in machine learning techniques, recommendation systems, among other applications. The widespread use of this decomposition is in the implementation in the LAPACK and SCALAPACK libraries, which are widely used.

However, the computation of this decomposition has a high computational cost that slows down its execution in large and dense matrices. Furthermore, current implementations are not designed for new clusters that have nodes with multiple CPUs and GPUs, which not only accelerate mathematical calculations but also have better efficiency than a traditional cluster, meaning they have a higher number of floating-point operations per Watt used than their traditional counterpart.

For this reason, a parallel hybrid heterogeneous implementation of the Jacobi method was carried out using tools (frameworks, APIs, specifications) such as CUDA, OpenMP, and MPI to take advantage of the resources of these new types of clusters. Additionally, other implementations of the same numerical method were performed for different programming models: parallel, heterogeneous parallel, hybrid parallel.

It was found that our parallel implementation for shared memory using OpenMP had a shorter execution time than the LAPACK implementation. The most important result of this thesis was that our heterogeneous parallel implementation had the shortest execution time compared to LAPACK and our other parallel implementations and had the lowest absolute error compared to our other implementations. Overall, an acceleration of almost up to 5 times was observed compared to our parallel shared memory implementation.

Resumen

La descomposición en valores singulares en matrices densas es usado en técnicas de aprendizaje de máquina, en sistemas de recomendación, entre otras aplicaciones. El amplio número de usuarios de esta descomposición se observa en la implementación en la biblioteca LAPACK y SCALAPACK que son ampliamente usadas.

Sin embargo, el cálculo de esta descomposición tiene un alto costo computacional que ralentiza su ejecución en matrices de gran tamaño y densas. Además, las implementaciones actuales no están diseñadas para los nuevos clusters que cuentan con nodos con multiples CPUs y GPUs que no solo aceleran cálculos matemáticos sino que tienen una mejor eficiencia que un cluster tradicional, es decir, tienen un mayor número de operaciones flotantes por Watt usado que su contra parte tradicional.

Por esta razón se realizó una implementación paralela híbrida heterogénea del método de Jacobi por un lado usando las herramientas (marcos de trabajo, API, especificaciones) CUDA, OpenMP y MPI; para aprovechar los recursos de estos nuevos tipos de clusters. Además se realizaron otras implementaciones del mismo método numérico pero para diferentes modelos de programación: paralela, paralela heterogénea, paralela híbrida.

Se encontró que nuestra implementación paralela para memoria compartida con OpenMP tuvo un menor tiempo de ejecución que la implementación de LAPACK. El resultado más importante de este texto fue que nuestra implementación paralela heterogénea es el menor en tiempo de ejecución comparado con LAPACK y nuestras otras implementaciones y fue el menor en error absoluto comparado con nuestras otras implementaciones. En general, se observó una aceleración de casi hasta 5 veces comparado con nuestra implementación paralela para memoria compartida.

Agradecimientos

Agradezco a mis padres, a mi amada Vania por la paciencia y el apoyo, a mis amigos Josue y Rogelio y a la Dra. Brisbane Ovilla por el apoyo y enseñanzas.

Agradezco enormemente a mis codirectores, el Dr. Amilcar Meneses Viveros y el Dr. Sergio Victor Chapa Vergara por todo su apoyo, tiempo y esfuerzo que dedicaron durante el desarrollo de mi trabajo de tesis.

Agradezco al Dr. Cuauhtemoc Mancillas Lopez y al Dr. Oliver Schütze por su valiosa contribución como sinodales en mi examen de grado. Su compromiso y experiencia fueron de gran importancia en este proceso y estoy sinceramente agradecido por su participación.

Agradezco al CINVESTAV (Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional), en particular al Departamento de Computación por aceptarme en su programa de Maestría en Ciencias en Computación.

Agradezco al CADS (Centro de Análisis de Datos y Supercómputo) de la Universidad de Guadalajara, en particular a la Dra. Liliana Ibeth Barbosa Santillán y a la Dra. Verónica Lizette Robles Dueñas por la oportunidad de realizar mis pruebas en la supercomputadora Leo Atrox, también agradezco al equipo de soporte y monitoreo por su ayuda.

De igual manera, agradezco al CONAHCYT (Consejo Nacional de Humanidades, Ciencia y Tecnología) por el apoyo económico brindado durante mis estudios de maestría.

Índice general

Índice de figuras

Códigos

Índice de tablas

Capítulo 1

Introducción

El álgebra lineal numérica tiene muchas aplicaciones en las ciencias como Física, Química, Ingeniería y Computación, entre otras. Existen múltiples métodos numéricos en la álgebra lineal numérica para algunos problemas matemáticos con varias aplicaciones, por ejemplo, los sistemas de ecuaciones lineales, mínimos cuadrados y obtención de vectores y valores propios por mencionar algunos. Estos problemas se usan para resolver incógnitas, ajustar datos y encontrar soluciones de problemas en mecánica cuántica, respectivamente [4].

Las supercomputadoras tienen bibliotecas de álgebra lineal numérica, las más usadas son BLAS, LAPACK y SCALAPACK. Estos programas contienen rutinas secuenciales, paralelas, distribuidas o alguna combinación de las anteriores que solucionan problemas, en particular, valores propios, resolución de ecuaciones lineales, factorización matricial, etc [5]. Por ejemplo, las tres primeras supercomputadoras en el ranking TOP 500 de supercomputadoras usan la biblioteca SCALAPACK para álgebra lineal numérica [6][7][8][9].

En el álgebra lineal hay una factorización de cualquier matriz A a una matriz diagonal multiplicado por ambos lados por matrices ortogonales. Esta factorización se le conoce como descomposición en valores singulares (o por sus siglas en inglés SVD), se usa para varios problemas matemáticos, computacionales, etc. Por ejemplo, análisis de componentes principales, compresión de imágenes, reducción de dimensionalidad, procesamiento de señales, aprendizaje de máquina (o machine learning), por nombrar algunos [10][11][12].

La factorización SVD se usa como base en técnicas de aprendizaje de máquina como spectral clustering [13][14], clustering [15], clasificación [10] o redes neuronales convolucionales [16]; En minería de datos es una técnica para la reducción de dimensionalidades; En sistemas de recomendación se usó en el algoritmo PageRank de Google [10] y se usó una instancia de SVD llamado descomposición UV que mejoró en 7 % el sistema de recomendación de Netflix de 2006 [11][17]. Además, en las últimas décadas se ha observado un crecimiento en investigaciones y aplicaciones en la I.A [18].

Otra aplicación de SVD es la compresión de imagenes donde se descompone la imagen y en la matriz diagonal se ponen ceros en sus valores singulares de manera

ascendente. De tal manera que se preservan las características distinguibles de una foto y el resultado es una matriz con menos columnas linealmente independientes que número de columnas [10][11].

Se puede aseverar que SVD tiene un amplio uso en varias ramas científicas y aplicaciones en la industria. De aquí que tenga una extensa gama de usuarios y haya un amplio interés que se puede observar en el incremento exponencial en la investigación de SVD [18].

Planteamiento del problema

La descomposición en valores singulares de una matriz tiene un alto costo computacional, el cálculo del número de operaciones se realiza en el capítulo 4. Actualmente para matrices densas existen tres esquemas de obtención de SVD: Algoritmos de Jacobi por uno o dos lados, algoritmos que usan como entrada una matriz bidiagonal y algoritmo que usan una matriz ordenada por bloques [19][20].

Los ambientes híbridos heterogéneos son una tendencia en diseño e implementación de clusters. Clusters como Frontier o Summit tienen nodos con múltiples CPUs y GPUs que maximizan la cantidad de operaciones de punto flotante por Watt. Sin embargo, la mayoría de las bibliotecas de álgebra lineal mejoran el tiempo de ejecución distribuyendo el cómputo en nodos homogéneos multi-core [5][8][9][21].

Los ambientes híbridos heterogéneos presentan un gran reto en el uso óptimo de recursos. A nivel programación, el uso óptimo de recursos se lograría con programas híbridos paralelos heterogéneos [21]. La programación híbrida paralela es la que combina los modelos de memoria distribuida y compartida, lo que se realiza cuando se programa con MPI y OpenMP. La programación heterogénea se lleva a cabo cuando el programa se ejecuta en múltiples procesadores asimétricos, es decir, que los procesadores tienen diferentes capacidades o que usan diferentes conjuntos de instrucciones de bajo nivel. A nivel paralelo la computación heterogénea sería usar aceleradores gráficos (GPUs). A nivel programación, CUDA es un marco de trabajo para incorporar programación heterogénea a otros paradigmas de programación. Así, la programación híbrida paralela heterogénea es la que hace uso de las bibliotecas MPI y OpenMP en el marco de trabajo CUDA.

La programación en estos ambientes es complicado debido a que se presentan problemas que introducen cuellos de botella que ralentizan la ejecución, algunos de estos problemas conocidos son: condiciones de carrera (data race), sincronización, secciones críticas, administración de memoria compartida no uniforme (NUMA), administración de memoria entre dispositivos asimétricos, entre otros. Existen múltiples modelos de programación con base en tareas, grafos y por la arquitectura de memoria del ambiente de ejecución. Programas para álgebra lineal en matrices densas como MAGMA pueden ajustar la distribución de carga entre CPU y GPU. En particular para la factorización SVD, la biblioteca MAGMA solo usa CPU [21].

Las bibliotecas de álgebra lineal en matrices densas actuales, y las más usadas, son simples en el modelo de programación usado, existen las combinaciones MPI +

Nucleo BLAS, OpenMP, MPI + OpenMP y OpenMP + CUDA. En 2020, se creó un marco de trabajo que incorpora múltiples modelos de programación, pero que se usa para nodos homogéneos y donde en cada nodo usa la unidad de procesamiento más potente que tenga al alcance ya sea CPU o GPU. Este marco está destinado a los proyectos de cómputo de exa-escala [22].

Para el cómputo de la factorización SVD existen múltiples implementaciones que usan algunos modelos de programación anteriormente mencionados. Las implementaciones más usadas son las de las bibliotecas usuales: SCALAPACK (MPI + núcleo BLAS) usa el modelo paralelo distribuido y que es una iteración de la biblioteca LAPACK (núcleo BLAS) que usa el modelo de programación paralelo, MAGMA (CPU + GPU) su modelo de programación se enfoca en una sola computadora con tecnología multi-core y aceleradores de cálculo aunque en el caso de SVD solo aprovecha los entornos multi-core; y hay algoritmos paralelos heterogéneos para SVD, pero que no distribuyen su carga para un cluster moderno como el que hemos descrito [23]. Por lo anterior, se propone realizar un programa híbrido heterogéneo paralelo para factorización SVD en matrices densas (MPI+OpenMP+CUDA) que aproveche los cluster modernos descritos anteriormente y las ventajas que estos ofrecen.

Objetivos

General

Desarrollar un programa híbrido paralelo heterogéneo para factorización SVD en matrices densas.

Particulares

1. Identificar indicadores para evaluar el desempeño del programa a desarrollar.

2. Identificar los algoritmos que realizan la factorización SVD de matrices densas.

3. Describir el comportamiento de las cargas de trabajo de los diferentes pasos que tienen los algoritmos.

4. Desarrollar un programa híbrido paralelo heterogéneo para la factorización SVD de matrices densas.

5. Analizar desempeño y correctitud del programa para diversas unidades de ejecución heterogéneas.

6. Comparar el programa contra implementaciones en otros modelos y el estado del arte.

Justificación

SVD es usado para grandes datos como en el caso de sistemas de recomendación donde no es posible usar un solo equipo para procesar toda la información, igualmente el costo de cómputo de SVD en matrices densas es alto, en nuestro caso para el método de Jacobi por un lado en una matriz $A \in \mathbb{R}^{n \times n}$, una iteración del algoritmo tiene como cota inferior $O(n^3)$ multiplicaciones y sumas de números flotantes.

En el área de ciencia de datos y aprendizaje de máquina aplicado en otras áreas se desean analizar datos que no vienen de un modelo sino una medición como son imagenes, videos en visión computacional o matrices de muestras contra genes donde se obtienen matrices de 168 muestras con 29285 genes muestreados en bioinformática [24][25].

En las bibliotecas LAPACK y SCALAPACK que consideramos estado del arte no hay implementaciones heterogéneas, ni híbrido heterogéneas para el cálculo de SVD, es decir, que no aprovechan los nuevos clusters con nodos heterogéneos que maximizan el número de operaciones flotantes por Watt usado.

En la Figura 1.1 se comparan las 4 bibliotecas del estado del arte por hardware, modelo de programación y tecnología. Se observa que ninguna de las bibliotecas usa el GPU.

	Hardware		Modelo			Tecnología			
Biblioteca	CPU	GPU	Híbrido	Paralelo	Heterogéneo	MPI	OpenMP	CUDA	Kernel BLAS
LAPACK (MKL)	X			X					X
SCALAPACK	X		X	X		X			X
MAGMA	X			X			X		
SLATE	X		X	X		X	X		

Figura 1.1: Comparación entre 4 bibliotecas del estado del arte entre el hardware que usan, modelo de programación y tecnología (biblioteca o marco de trabajo)

Este documento de tesis se encuentra dividido en 6 capítulos. En el siguiente capítulo se discute la descomposición en valores singulares y algunos métodos numéricos para realizarla. En el tercer capítulo se presentan los diferentes modelos de programación paralela haciendo énfasis en los modelos que se implementaron. El capítulo 4 se explica y detalla la implementación que se utilizó acompañado de código fuente. El capítulo 5 presenta los resultados experimentales de las implementaciones. Y finalmente, el último capítulo presenta las conclusiones de este trabajo de tesis.

Capítulo 2

Descomposición en valores singulares y métodos numéricos para el cómputo de SVD

El objetivo principal de este trabajo de tesis es la implementación híbrida paralela heterogénea de un método numérico para SVD que opere sobre matrices densas. Por ello, en este capítulo se presenta una discusión sobre la factorización SVD y los diferentes métodos numéricos de diagonalización. La primera sección de este capítulo comienza con una síntesis de la descomposición en valores singulares en las matemáticas y algunos ejemplos de aplicación. En la segunda sección definimos y demostramos múltiples propiedades y lemas de las normas matriciales. Como se verá, antes de realizar la computación de SVD hay que usar otra factorización, la tercera sección enumera algunas factorizaciones que se pueden realizar antes del cómputo de SVD. La última sección detalla múltiples algoritmos numéricos para diagonalización y cálculo de SVD.

2.1. Descomposición en valores singulares

La descomposición en valores singulares es una descomposición que se relaciona como veremos mas adelante con la descomposición espectral de una matriz. La importancia de la descomposición SVD radica en su capacidad para extraer características clave y estructuras subyacentes en los datos, además de que proporciona una representación compacta y significativa de la información contenida en una matriz, lo que facilita la manipulación y el análisis de los datos. Tiene aplicaciones en el análisis de componentes principales, aprendizaje de máquina, indexación semántica latente (o por sus siglas en inglés LSI) [4][10][11][15][26].

2.1.1. Definición, propiedades, entre otros

Definición 2.1.1 (Descomposición en valores singulares)**.** Sea $A \in \mathbb{R}^{m\times n}$ con rango r y diferente de la matriz $\mathbf{0}$. Su descomposición en valores singulares es

$$A = U\Sigma V^T,$$

con $U \in \mathbb{R}^{m\times m}$ ortogonal, $V \in \mathbb{R}^{n\times n}$ ortogonal y $\Sigma \in \mathbb{R}^{m\times n}$ es una matriz diagonal

$$\Sigma = diag(\sigma_1, \ldots, \sigma_r) = \begin{pmatrix} \sigma_1 & 0 & 0 & 0 & 0 & \ldots & 0 \\ 0 & \sigma_2 & 0 & 0 & 0 & \ldots & 0 \\ 0 & 0 & \ddots & 0 & 0 & \ldots & 0 \\ 0 & 0 & 0 & \sigma_r & 0 & \ldots & 0 \\ 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

con $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$, para mas detalles consultar [4][27][19].

En la Figura 2.1 se presenta una visualización de las dimensiones de las matrices A, U, Σ, V.

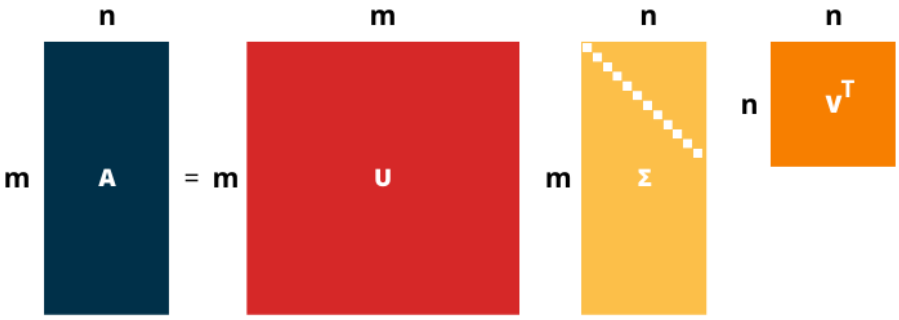

Figura 2.1: Visualización de la descomposición en valores singulares.

2.1.2. Interpretación geométrica

Supongamos $A \in \mathbb{R}^{2\times 2}$ una matriz cuadrada con rango $r = 2$ que representa una transformación lineal $T : \mathbb{R}^2 \to \mathbb{R}^2$. Su descomposición en valores singulares es $U\Sigma V^T$.

$$A = \begin{pmatrix} u_0 & | & u_1 \end{pmatrix} \begin{pmatrix} \sigma_0 & 0 \\ 0 & \sigma_1 \end{pmatrix} \begin{pmatrix} v_0 & | & v_1 \end{pmatrix}^T,$$

con $u_0, u_1 \in \mathbb{R}^2$ sus vectores singulares por la izquierda, $\sigma_0, \sigma_1 \in \mathbb{R}$ sus valores singulares y $v_0, v_1 \in \mathbb{R}^2$ los vectores singulares por la derecha. Veamos que ocurre en el caso que Av_0.

$$\begin{aligned} Av_0 &= \begin{pmatrix} u_0 & | & u_1 \end{pmatrix} \begin{pmatrix} \sigma_0 & 0 \\ 0 & \sigma_1 \end{pmatrix} \begin{pmatrix} v_0 & | & v_1 \end{pmatrix}^T v_0 \\ &= \begin{pmatrix} \sigma_0 u_0 & | & \sigma_1 u_1 \end{pmatrix} \begin{pmatrix} v_0 & | & v_1 \end{pmatrix}^T v_0 \\ &= \begin{pmatrix} \sigma_0 u_0 & | & \sigma_1 u_1 \end{pmatrix} \begin{pmatrix} 1 \\ - \\ 0 \end{pmatrix} \\ &= \sigma_0 u_0. \end{aligned}$$

Análogamente obtenemos $Av_1 = \sigma_1 u_1$. Esto significa que la base ortonormal $\{v_0, v_1\}$ de $\mathbb{R}^2$ como se muestra en la Figura 2.2 con el círculo unitario, se transforma en vectores de otra base con diferentes escalas respectivamente $\{\sigma_0 u_0, \sigma_1 u_1\}$ de $\mathbb{R}^2$ como se muestra en la Figura 2.3 y la elipse que forma [28].

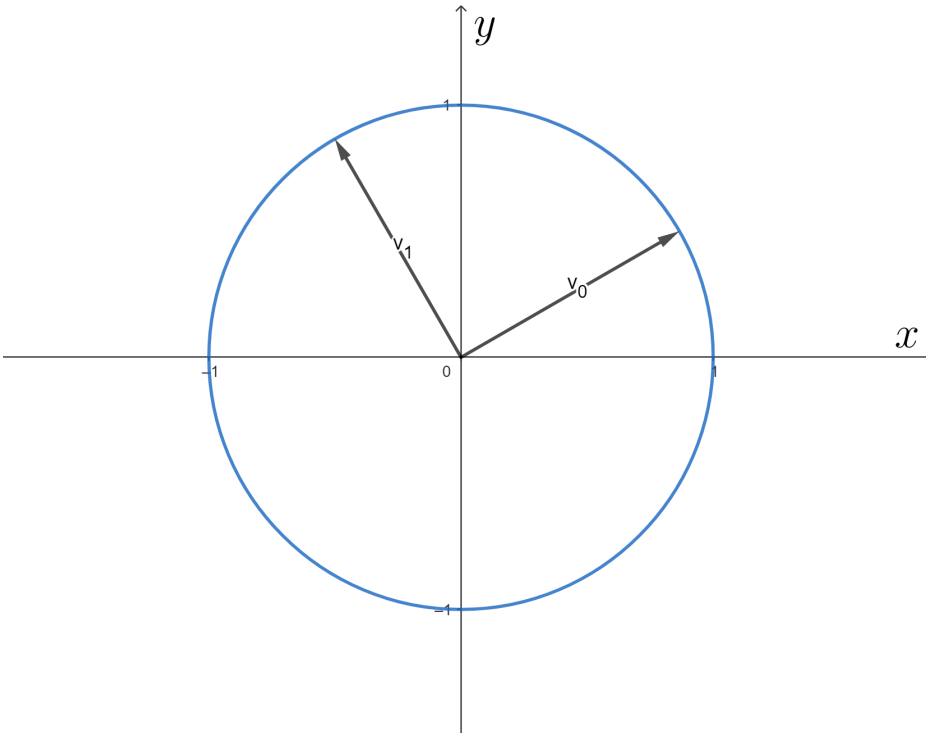

Figura 2.2: Base ortonormal $\{v_0, v_1\}$ de $\mathbb{R}^2$.

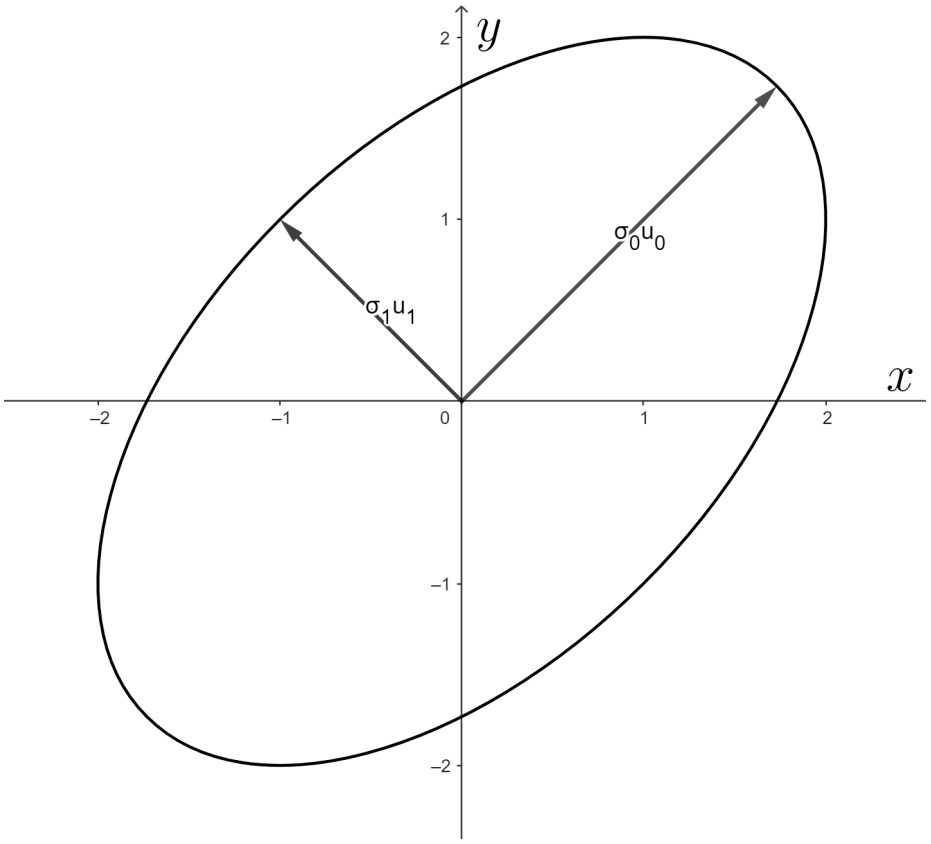

Figura 2.3: $\{v_0, v_1\}$ después de transformación $U\Sigma$ forma un escalamiento de la base como $\{\sigma_0 u_0, \sigma_1 u_1\}$ de $\mathbb{R}^2$.

Si $A \in \mathbb{R}^{m\times n}$ es una matriz con rango $r = n$ y $A = U'\Sigma'V'^T$ y que representa una transformación lineal $T : \mathbb{R}^n \to \mathbb{R}^m$, es posible demostrar que la transformación lineal T mapea la esfera unitaria en $\mathbb{R}^n$ a un elipsoide en $\mathbb{R}^m$ [4].

2.1.3. Relación con la descomposición espectral de una matriz simétrica

Existe una relación entre la descomposición en valores singulares de A y la descomposición espectral de AA^T o A^TA.

$$AA^T = (U\Sigma V^T)(U\Sigma V^T)^T = U\Sigma V^T V\Sigma U^T = U\Sigma^2 U^T,$$
$$A^TA = (U\Sigma V^T)^T(U\Sigma V^T) = V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T.$$

2.1.4. Aplicaciones

Aproximación de bajo rango

Sea $A \in \mathbb{R}^{m\times n} \setminus \{\mathbf{0}\}$ con rango n, $m \geq n \geq 1$ y $\Sigma = diag(\sigma_1, \ldots, \sigma_n)$. Tomamos $\epsilon > 0$ y sea $1 \leq r \leq n$ el entero más pequeñó, tal que $\sigma_{r+1}^2 + \cdots + \sigma_n^2 < \epsilon^2$. Formamos A' usando la descomposición en valores singulares de A y transformando la matriz Σ en una con menos valores singulares $\Sigma' = diag(\sigma_1, \ldots, \sigma_r)$ [4].

Si se toma un ϵ pequeño, entonces A está cerca de la matriz A'. También la aproximación puede ser visto como una forma de reducir o truncar la "información" de una matriz sin perder "calidad" [26].

$$A \approx \sum^{r} \sigma_r u_r v_r^T.$$

Esta aproximación se puede usar para reducir la cantidad de almacenamiento de una imagen. Por ejemplo, si se considera una imagen en escala de grises como una matriz A con valores en el intervalo $[0, 2^n - 1]$ con n bits. Se obtiene la descomposición en valores singulares de la imagen $A = U\Sigma V^T$ y elegimos los primeros k valores singulares que mantienen una imagen con "calidad". Así, reducimos la cantidad de almacenamiento.

En la Figura 2.4 se presenta una imagen de 640×597, tiene rango de 597 en cada canal de color. Se retienen los 70 valores singulares más grandes, donde se observa que obtenemos una buena aproximación.

Figura 2.4: Compresión de imagen obtenida a través de [1].

Agrupamiento espectral

El agrupamiento espectral es una técnica de particionamiento de grafos y se cataloga como aprendizaje de máquina no supervisado. El problema de particionamiento está relacionado a la descomposición en valores singulares de tensores simétricos [14].

Un agrupamiento espectral es cuando se tienen n objetos $\mathcal{V} = v_1, ..., v_n$ y se agrupan estos objetos entre k conjuntos disjuntos $C_1, ..., C_k$. Se mide la similaridad de estos objetos con una función simétrica y no negativa lo que genera una matriz de similaridad entre objetos [13].

La descomposición de valores singulares se usa en esta matriz de similaridad para obtener los vectores singulares por la izquierda [13][14].

Algoritmo 2.1 Algoritmo de agrupamiento espectral no normalizado, extraido de [13]

Require: Una matriz de similaridad $S \in \mathbb{R}^{n \times n}$

1: Construir un grafo de similaridad con W su matriz de adyacencia con pesos.
2: Computar una matriz laplaciana L.
3: Computar los primeros k vectores singulares por la izquierda de L en la matriz U.
4: Para cada vector columna de U, agrupar con el algoritmo *k-means* en los k agrupamientos.

2.2. Álgebra lineal numérica

En esta sección se presenta una revisión de definiciones, propiedades y resultados de normas matriciales, dado que se usarán de manera regular estos conceptos en este texto. También se presenta una discusión sobre el número de condición que se encuentra relacionado a la perturbación del error en sistemas de ecuaciones lineales. Para más detalles sobre estos conceptos en [4][28].

2.2.1. Normas matriciales

Definición 2.2.1 (**Norma matricial**)**.** Una norma matricial $||\cdot|| : \mathbb{R}^{m\times n} \to \mathbb{R}$ es una norma si satisface que $\forall A, B \in \mathbb{R}^{m\times n}$ y $\forall \alpha \in \mathbb{R}$: [4]

1. $||A|| \geq 0$ (Positivo).
2. $||A|| = 0$ si y solo si $A = 0$ (La matriz 0).
3. $||\alpha A|| = |\alpha|||A||$ (Homogéneo).
4. $||A + B|| \leq ||A|| + ||B||$ (Desigualdad del triángulo).

Las normas matriciales son equivalentes entre si,

Teorema 2.2.1 (**Equivalencia de normas matriciales**)**.** Todas las normas sobre $\mathbb{R}^{m\times n}$ son equivalentes. En consecuencia, si $||\cdot||$ y $||\cdot||'$ son dos normas matriciales sobre $\mathbb{R}^{m\times n}$, entonces $\exists \mu, M \in \mathbb{R}$ tal que:

$$\mu||A|| \leq ||A||' \leq M||A||,$$

con $A \in \mathbb{R}^{m\times n}$.

Las dos normas que se emplearán de manera regular es la norma de Frobenius y las normas matriciales inducidas que se definen a continuación.

Definición 2.2.2 (**Norma de Frobenius**)**.** La norma de Frobenius $||\cdot||_F$ de una matriz $A \in \mathbb{R}^{m\times n}$ está definido como

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}.$$

Definición 2.2.3 (**Norma matricial inducida por una norma vectorial**)**.** Una norma matricial $||\cdot||_\mu : \mathbb{R}^{m\times n} \to \mathbb{R}$ es inducida por la norma vectorial $||\cdot||_\mu : \mathbb{R}^n \to \mathbb{R}$ como:

$$||A||_\mu = \max_{||x||_\mu = 1} ||Ax||_\mu = \max_{||x||_\mu \neq 0} \frac{||Ax||_\mu}{||x||_\mu},$$

con $x \in \mathbb{R}^n$.

A continuación se definen múltiples propiedades que algunas normas pueden tener.

Definición 2.2.4 (**Norma matricial invariante bajo multiplicación de matrices ortogonales**)**.** Una norma matricial $||\cdot||$ sobre $\mathbb{R}^{m\times n}$ es invariante bajo multiplicación de matrices ortogonales si $||UAV|| = ||A||$ con $\forall A \in \mathbb{R}^{m\times n}$ y matrices ortogonales $U \in \mathbb{R}^{m\times m}$ y $V \in \mathbb{R}^{n\times n}$.

Definición 2.2.5 (**Norma matricial subordinada por una norma vectorial**)**.** Una norma matricial $||\cdot|| : \mathbb{R}^{m\times n} \to \mathbb{R}$ es subordinada por la norma vectorial $||\cdot||_\mu : \mathbb{R}^n \to \mathbb{R}$ si $\forall x \in \mathbb{R}^n$:

$$||Ax||_\mu \leq ||A||||x||_\mu.$$

Definición 2.2.6 (**Norma consistente**)**.** Una norma matricial $||\cdot|| : \mathbb{R}^{m\times n} \to \mathbb{R}$ es consistente si $\forall m, n \in \mathbb{N}$ está definido y es la misma formula en todos los casos.

Definición 2.2.7 (**Norma matricial submultiplicativa**)**.** Una norma matricial consistente $||\cdot|| : \mathbb{R}^{m\times n} \to \mathbb{R}$ es submultiplica si satisface:

$$||AB|| \leq ||A||\ ||B||.$$

Ahora se demostrarán algunos lemas que se usarán en las siguientes secciones.

Lema 2.2.2. Las normas inducidas por una norma vectorial son subordinadas.

Demostración. Sea una norma matricial $||\cdot||_\mu : \mathbb{R}^{m\times n} \to \mathbb{R}$ inducida por la norma vectorial $||\cdot||_\mu : \mathbb{R}^n \to \mathbb{R}$, tenemos entonces

$$||A||_\mu = \max_{||x||_\mu=1} ||Ax||_\mu = \max_{||x||_\mu\neq 0} \frac{||Ax||_\mu}{||x||_\mu},$$

con $x \in \mathbb{R}^n$.

Si $x = 0$, entonces $||Ax||_\mu \leq ||A||_\mu||x||_\mu$ se cumple trivialmente. En otro caso $x \neq 0$

$$\begin{aligned} ||A||_\mu = \max_{||x||_\mu\neq 0} \frac{||Ax||_\mu}{||x||_\mu} &\geq \frac{||Ax||_\mu}{||x||_\mu} \\ \iff ||A||_\mu||x||_\mu &\geq ||Ax||_\mu. \end{aligned}$$

□

Lema 2.2.3. La norma de Frobenius es submultiplicativa.

Demostración. Sea $A \in \mathbb{R}^{m\times n}$ y $B \in \mathbb{R}^{n\times k}$

$$\begin{aligned} ||AB||_F^2 &= \sum_{i=1}^{m}\sum_{j=1}^{k} |a_i^T b_j|^2 \\ &\leq \sum_{i=1}^{m}\sum_{j=1}^{k} ||a_i^T||_2^2||b_j||_2^2 \text{ por la desigualdad de Cauchy-Schwarz [29]} \\ &= ||A||_F^2||B||_F^2, \end{aligned}$$

con b_j el j-ésimo vector columna de B, a_i^T el i-ésimo vector columna de A^T, i.e. el i-ésimo vector fila de A. □

Lema 2.2.4. La norma de Frobenius es subordinada por la 2-norma vectorial.

Demostración. Del lema 2.2.3, con $B = x \in \mathbb{R}^n$, es trivial que $||B||_F = ||B||_2$. Así

$$||Ax||_F \leq ||A||_F ||x||_2.$$

□

Lema 2.2.5. Sea $U \in \mathbb{R}^{n\times n}$ una matriz ortogonal y $x \in \mathbb{R}^n$, $||Ux||_2 = ||x||_2$

Demostración.

$$\begin{aligned} ||Ux||_2^2 &= (Ux)^T(Ux) \\ &= x^T U^T U x \\ &= x^T x \\ &= ||x||_2^2. \\ \implies ||Ux||_2 &= ||x||_2 \end{aligned}$$

□

Lema 2.2.6. La norma de Frobenius es invariante bajo multiplicación de matrices ortogonales

Demostración. Sea $U \in \mathbb{R}^{m\times m}$ y $V \in \mathbb{R}^{n\times n}$ matrices ortogonales y $A \in \mathbb{R}^{m\times n}$.

1. $$\begin{aligned} ||UA||_F^2 &= \sum_{i=1}^{n} ||Ua_i||_2^2 \text{ , } a_i \text{ es el i-ésimo vector columna de } A \\ &= \sum_{i=1}^{n} ||a_i||_2^2 \text{ por 2.2.5} \\ &= ||A||_F^2. \end{aligned}$$
2. $$\begin{aligned} ||AV||_F^2 &= ||(AV)^T||_F^2 = ||V^T A^T||_F^2 \\ &= \sum_{i=1}^{m} ||V^T a_i^T||_2^2 \text{ , } a_i^T \text{ es el i-ésimo vector columna de } A^T \\ &= \sum_{i=1}^{m} ||a_i^T||_2^2 \text{ por 2.2.5} \\ &= ||A^T||_F^2 = \sum_{i=1}^{n}\sum_{j=1}^{m} |a_{ij}|^2 \text{ , definición de } ||\cdot||_F \\ &= \sum_{j=1}^{m}\sum_{i=1}^{n} |a_{ij}|^2 \\ &= ||A||_F^2. \end{aligned}$$

□

2.2.2. Número de condición y perturbación en sistemas lineales

En el álgebra lineal numérica, se desea saber el efecto que tiene cambiar ligeramente $A \in \mathbb{R}^{n\times n}$ invertible o $b \in \mathbb{R}^n$ sobre A^{-1} y sobre la solución $x \in \mathbb{R}^n$ del sistema lineal $Ax = b$.

Perturbación del resultado b sobre la solución x

Si se perturba un sistema de ecuaciones lineales $Ax = b$, la ecuación se transforma en $A(x + \delta x) = (b + \delta b)$. El objetivo de esta perturbación es determinar una función $\kappa(A, b, \delta b)$ que nos de una cota sobre cuanto el error relativo en b amplifica el error relativo en la solución x. Esto es:

$$\frac{||\delta x||}{||x||} \leq \kappa(A, b, \delta b)\frac{||\delta b||}{||b||}.$$

A no es singular (invertible), esto para que el sistema lineal tenga solución. Si se realizan algunas operaciones sobre el sistema de ecuación lineal y el perturbado se obtiene:

$$\begin{aligned} & A(x + \delta x) - Ax = (b + \delta b) - b \\ \iff & A\delta x = \delta b \\ \implies & \delta x = A^{-1}\delta b. \end{aligned}$$

Si se usa una norma vectorial $||\cdot||$ y su norma matricial inducida $||\cdot||$, entonces se obtiene lo siguiente [28]:

$$\begin{aligned} & 1.\ ||b|| = ||Ax|| \leq ||A||\ ||x|| \\ & \implies \frac{1}{||x||} \leq ||A||\frac{1}{||b||}. \\ & 2.\ ||\delta x|| = ||A^{-1}\delta b|| \leq ||A^{-1}||\ ||\delta b|| \\ & \implies ||\delta x|| \leq ||A^{-1}||\ ||\delta b||. \\ \text{De 1. y 2.} \quad & \implies \frac{||\delta x||}{||x||} \leq ||A||\ ||A^{-1}||\frac{||\delta b||}{||b||}. \end{aligned}$$

Así, $\kappa(A, b, \delta b) = ||A||\ ||A^{-1}||$. A este valor se le llama el **número de condición** $\kappa(A)$ de una matriz no singular. Esto nos dice que el error relativo en b es amplificado $\kappa(A)$ veces el error relativo en la solución x.

2.3. Preprocesamiento de datos

El preprocesamiento de datos para SVD es importante porque se puede usar como transformación para la entrada de algún método numérico de diagonalización, para

reducir el tiempo de cómputo del método numérico de diagonalización seleccionado o para descomponer una matriz densa, es decir, una matriz donde casi todos sus elementos son distintos de 0. Las factorizaciones ortogonales comúnmente usadas son la descomposición QR y la bidiagonalización de una matriz [2].

La diagonalización de una matriz es el método numérico que se usa para obtener la descomposición SVD de una matriz, existen muchos métodos de diagonalización y todos son iterativos. A continuación, se muestra una figura sobre el cómputo de la descomposición en valores singulares [2].

La Figura 2.5 muestra primero el paso del preprocesamiento de una matriz densa y seguido de esto el uso de un método iterativo de diagonalización, en particular vemos que los métodos basados en rotaciones de Jacobi pueden usar de entrada una matriz triangular o bidiagonal y los algoritmos de Kahan en particular el Golub-Kahan y Demmel-Kahan solo aceptan como entrada una matriz bidiagonal.

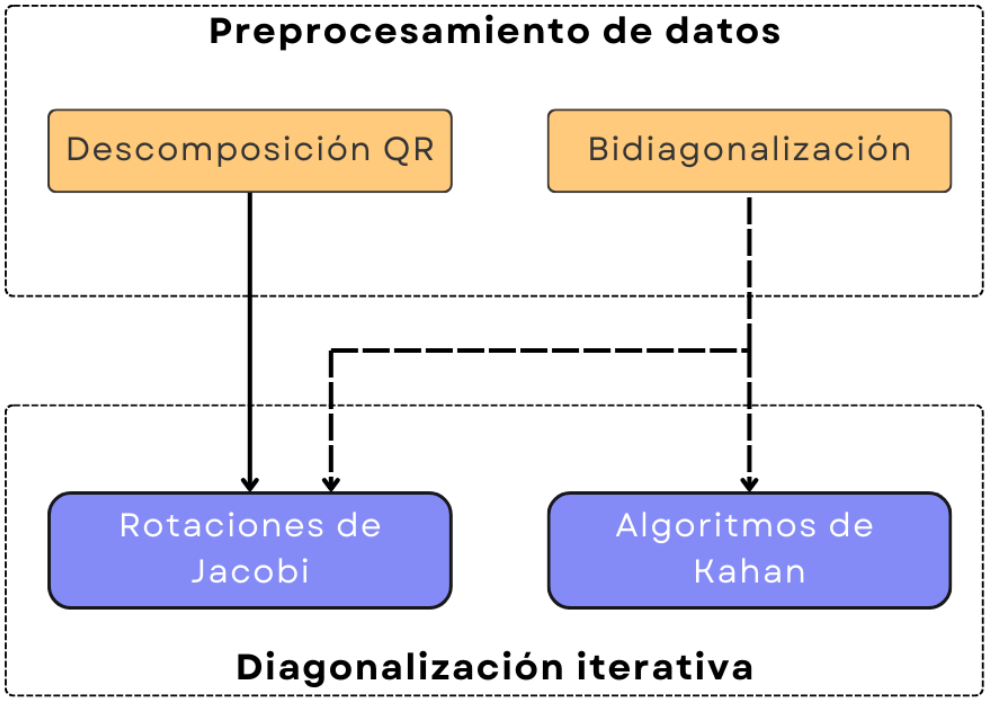

Figura 2.5: Esquema de cómputo de SVD [2].

2.3.1. Descomposición QR

Definición 2.3.1 (**Descomposición QR**)**.** Sea $A \in \mathbb{R}^{m\times n}$ con m filas y n columnas. $A = QR$ es una **descomposición QR** de A si $Q \in \mathbb{R}^{m\times m}$ es cuadrado y ortogonal, i.e. $QQ^T = \mathbb{1}$ y $R \in \mathbb{R}^{m\times n}$ es una matriz triangular superior. Si $m \geq n$, entonces R toma la forma de

$$\begin{bmatrix} R_1 \\ 0_{m-n,n} \end{bmatrix},$$

donde $0_{m-n,n} \in \mathbb{R}^{m-n\times n}$ es la matriz cero con $m-n$ filas y n columnas y $R_1 \in \mathbb{R}^{n\times n}$ es una matriz cuadrada triangular superior con la siguiente forma.

$$\begin{pmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,n} \\ 0 & r_{2,2} & \dots & r_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & r_{n,n} \end{pmatrix}.$$

La descomposición QR siempre existe, pero su unicidad está condicionado si A tiene columnas linealmente independientes, i.e. si su rango es n y R tiene elementos diagonales positivos.

Transformación de Householder

Definición 2.3.2 (**Transformación de Householder**)**.** Una transformación de Householder $H \in \mathbb{R}^{n\times n}$ tiene la forma de

$$H = \mathbb{1} - 2\frac{uu^T}{u^Tu},$$

con $u \neq 0 \in \mathbb{R}^n$ y $\forall u \in \mathbb{R}^n$ [4].

Lema 2.3.1. Las transformaciones de Householder son simétricas.

Demostración. Sea $H \in \mathbb{R}^{n\times n}$ una transformación de Householder, por definición se tiene $H = \mathbb{1} - 2\frac{uu^T}{u^Tu}$ con $u \neq 0 \in \mathbb{R}^n$ arbitrario. Así

$$H^T = (\mathbb{1} - 2\frac{uu^T}{u^Tu})^T = (\mathbb{1} - 2\frac{(u^T)^Tu^T}{u^Tu}) = (\mathbb{1} - 2\frac{uu^T}{u^Tu}) = H.$$

□

Lema 2.3.2. Las transformaciones de Householder son ortogonales.

Demostración. Recordar que $(AB)^T = B^TA^T$ con $A \in \mathbb{R}^{l\times m}, B \in \mathbb{R}^{m\times n}$.

$$\begin{aligned} H^2 &= (\mathbb{1} - 2\frac{uu^T}{u^Tu})(\mathbb{1} - 2\frac{uu^T}{u^Tu}) = \mathbb{1} - 4\frac{uu^T}{u^Tu} + 4\frac{uu^Tuu^T}{(u^Tu)^2} \\ &= \mathbb{1} - 4\frac{uu^T}{u^Tu} + 4\frac{u(u^Tu)u^T}{(u^Tu)^2} = \mathbb{1} - 4\frac{uu^T}{u^Tu} + 4\frac{u^Tu(uu^T)}{(u^Tu)^2} \\ &= \mathbb{1} - 4\frac{uu^T}{u^Tu} + 4\frac{uu^T}{u^Tu} = \mathbb{1}. \end{aligned}$$

□

Introduciendo ceros a un vector columna de una matriz

Definición 2.3.3 (**Ceros en vectores**). $\forall x \neq 0 \in \mathbb{R}^n$ hay una transformación de Householder $H \in \mathbb{R}^{n\times n}$ tal que:

$$Hx = \alpha \mathbf{e_1},$$

donde $u = x + (sign(x_1)|x|_2)\mathbf{e_1}$. [4]

A continuación se dará una pequeña prueba de lo anterior.

Por simplicidad $u = x + |x|\mathbf{e_1}$ con $x_1 > 0$. Y observando que $Hx = (\mathbb{1} - 2\frac{uu^T}{u^Tu})x = x - 2\frac{u^Tx}{u^Tu}u$. Así, calculando u^Tx:

$$\begin{aligned} u^Tx &= (x + |x|_2\mathbf{e_1})^Tx = x^Tx + |x|_2\mathbf{e_1}{}^Tx \\ &= |x|_2^2 + x_1 \cdot |x|_2. \end{aligned}$$

Después, computando u^Tu

$$\begin{aligned} u^Tu = u^T(x + |x|_2\mathbf{e_1}) &= u^Tx + |x|_2u^T\mathbf{e_1} = |x|_2^2 + |x|_2 \cdot x_1 + |x|_2(x + |x|_2\mathbf{e_1})^T\mathbf{e_1} \\ &= |x|_2^2 + |x|_2 \cdot x_1 + |x|_2(x^T\mathbf{e_1} + |x|_2\mathbf{e_1}{}^T\mathbf{e_1}) = |x|_2^2 + |x|_2 \cdot x_1 + |x|_2(x_1 + |x|_2) \\ &= 2(|x|_2^2 + |x|x_1). \end{aligned}$$

Finalmente, computando Hx:

$$\begin{aligned} Hx = (\mathbb{1} - 2\frac{uu^T}{u^Tu})x &= x - 2\frac{u^Tx}{u^Tu}u \\ &= x - 2\frac{(|x|_2^2 + |x|_2 \cdot x_1)}{2(|x|_2^2 + |x|_2 \cdot x_1)}u = x - u = x - (x + |x|_2\mathbf{e_1}) \\ &= -|x|_2\mathbf{e_1}. \end{aligned}$$

El resultado nos indica que el vector columna tiene ceros excepto por el primer elemento [30].

Triangularización de Householder

Sea $A \in \mathbb{R}^{m\times n}$, se desea encontrar una secuencia $\{H_1, ..., H_n\}$, tal que

$$A_{n+1} = H_n...H_3H_2H_1A = \begin{bmatrix} \mathbf{R_1} \\ \mathbf{0} \end{bmatrix} = \mathbf{R}$$

donde $\mathbf{R_1}$ es cuadrado y triangular superior; y $(H_n...H_3H_2H_1)^T = Q$ [4].

Supongamos en la iteración k, A_k tiene la siguiente forma:

$$A_k = \begin{bmatrix} \mathbf{B_k} & \mathbf{C_k} \\ \mathbf{0} & \mathbf{D_k} \end{bmatrix} = \mathbf{R},$$

donde $\mathbf{B_k}$ es triangular superior. Se desea introducir ceros en la siguiente $k+1$ columna de la matriz A_k, i.e. la primera columna de D_k. Sea $\mathbf{\hat{H}_k}$ una transformación de Householder computada por d_k^1 la primera columna de D_k, i.e. $u = d_k^1 + sign(d_k^1 \cdot \mathbf{e_1})|d_k^1|\mathbf{e_1}$. Entonces, $H_k = \begin{bmatrix} \mathbb{1}_\mathbf{k} & \mathbf{0} \\ \mathbf{0} & \mathbf{\hat{H}_k} \end{bmatrix}$ es una transformación de Householder que se computa como:

$$A_{k+1} = H_k A_k = \begin{bmatrix} \mathbf{B_k} & \mathbf{C_k} \\ \mathbf{0} & \mathbf{\hat{H}_k D_k} \end{bmatrix} = \begin{bmatrix} \mathbf{B_{k+1}} & \mathbf{C_{k+1}} \\ \mathbf{0} & \mathbf{D_{k+1}} \end{bmatrix}.$$

2.3.2. Bidiagonalización

Definición 2.3.4 (Matriz bidiagonal)**.** Sea $B \in \mathbb{R}^{m \times n}$ con m filas y n columnas; y $m \geq n$. B es una matriz bidiagonal superior si toma la forma de

$$\begin{bmatrix} B_1 \\ 0_{m-n,n} \end{bmatrix},$$

donde $B_1 \in \mathbb{R}^{n \times n}$ es una matriz con la siguiente forma [4]:

$$\begin{bmatrix} d_1 & b_1 & 0 & 0 & \dots & 0 \\ 0 & d_2 & b_2 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & d_{n-1} & b_{n-1} \\ 0 & 0 & 0 & 0 & 0 & d_n \end{bmatrix}.$$

Es posible realizar un método numérico que bidiagonalice una matriz $A \in \mathbb{R}^{m \times n}$ usando transformaciones de Householder por ambos lados.

Bidiagonalización de Householder

Supongamos $A \in \mathbb{R}^{m \times n}$ y $m \geq n$. Podemos encontrar secuencias de transformaciones de Householder $U = U_1 U_2 ... U_n$ y $V = V_1 V_2 ... V_{n-2}$, tal que $U^T A V = B$ donde B es una matriz bidiagonal superior [31].

Introduzcamos la siguiente notación, sea $A \in \mathbb{R}^{m \times n}$, cuando escribimos $A[i:j][k:l]$ con $1 \leq i \leq j \leq m$ y $1 \leq k \leq l \leq m$ estamos denotando la submatriz:

$$A[i:j][k:l] = \begin{bmatrix} a_{i,k} & a_{i,k+1} & \cdots & a_{i,l} \\ a_{i+1,k} & a_{i+1,k+1} & \cdots & a_{i+1,l} \\ \vdots & \vdots & \ddots & \vdots \\ a_{j,k} & a_{j,k+1} & \cdots & a_{j,l} \end{bmatrix}.$$

El algoritmo es relativamente sencillo y es como sigue [31]:

1. Empezamos con $j = 1$

2. Obtenemos la matriz de transformación de Householder del vector columna con $x = A[j : m][j]$.

3. Multiplicamos por la izquierda la matriz de transformación a la submatriz $A[j : m][j : n]$.

4. Si $j > n - 2$, saltamos al paso 7. De lo contrario continuamos

5. Obtenemos la matriz de transformación de Householder del vector fila con $x = A[j][j + 1 : n]$.

6. Multiplicamos por la derecha la matriz de transformación a la submatriz $A[j : m][j + 1 : n]$.

7. $j \longleftarrow j + 1$. Si $j < n + 1$, regresamos al paso 2, de lo contrario terminamos.

2.4. Algoritmos numéricos de SVD para matrices densas

En esta sección se explica la definición de métodos numéricos y su velocidad de convergencia, seguido de algunas operaciones matemáticas que se usarán cuando se enumeren algunos métodos numéricos para obtener la descomposición en valores singulares.

2.4.1. Métodos numéricos y velocidad de convergencia

Los métodos numéricos son métodos de computación matemáticos para encontrar una solución aproximada a un problema matemático, a diferencia de un algoritmo que son pasos para resolver un problema. En general, los métodos numéricos se dividen en dos, directos e iterativos. Los métodos directos son aquellos que en número de pasos finito y conocido llegamos a la solución; los métodos iterativos son aquellos que dada una aproximación x_0, computamos una secuencia iterativa de aproximaciones donde cada iteración disminuye el error absoluto entre la solución exacta x y la solución obtenida, i.e. $|x_k - x|$, donde $|\cdot|$ denota una norma [4][32].

Los métodos iterativos están clasificados según la velocidad de convergencia, las velocidades más comunes son superlineal, lineal y cuadrático [32]. Sea $\{x_k\}$ una secuencia que converge, las velocidades de convergencia son:

- Lineal. Cuando la secuencia converge como $|x_{k+1} - x| \leq c|x_k - x|$, con $k > n_0$ para algún $n_0 \in \mathbb{N}$.

- Superlineal. Si para una secuencia $\{c_k\}$ que converge a 0, la secuencia converge como $|x_{k+1} - x| \leq c_k|x_k - x|$, con $k > n_0$ para algún $n_0 \in \mathbb{N}$.

- Cuadrático. Cuando la secuencia converge como $|x_{k+1} - x| \leq c|x_k - x|^2$, con $k > n_0$ para algún $n_0 \in \mathbb{N}$.

2.4.2. Descomposición de Schur de una matriz de 2×2

Dado una matriz simétrica de $A \in \mathbb{R}^{2\times 2}$, se calcula la matriz $J \in \mathbb{R}^{2\times 2}$ que diagonaliza a A.

La matriz A tiene la siguiente forma:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}.$$

Y la matriz J tiene la forma:

$$J = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}.$$

De tal manera que:

$$J^T A J = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix} = \begin{pmatrix} \lambda_0 & 0 \\ 0 & \lambda_1 \end{pmatrix} = \Lambda.$$

Diagonalizar A significa encontrar $c = cos(\theta)$ y $s = sin(\theta)$ que transforme de A a Λ:

$$0 = a_{12}(c^2 - s^2) + (a_{11} - a_{22})cs. \tag{2.1}$$

Por un lado si $a_{12} = 0$, entonces se obtiene trivialmente a Λ. Por otro lado si $a_{12} \neq 0$, con $c = cos(\theta)$ y $s = sin(\theta)$, entonces $t = tan(\theta) = \frac{s}{c}$. Se define τ como

$$\tau = \frac{a_{22} - a_{11}}{2a_{12}}. \tag{2.2}$$

Usando τ de 2.2 en 2.1, obtenemos

$$\begin{aligned} 0 &= a_{12}(c^2 - s^2) + (a_{11} - a_{22})cs \\ &= a_{12}(1 - t^2) + (a_{11} - a_{22})t \\ &= 1 - t^2 + \frac{(a_{11} - a_{22})}{a_{12}}t \\ &= 1 - t^2 - 2\tau t \\ &= t^2 + 2\tau t - 1. \end{aligned}$$

La solución de la ecuación 2.3 satisface $|\theta| < \frac{\pi}{4}$, i.e. que $|t| < 1$ [31].

$$t_{min} = \begin{cases} \frac{1}{(\tau + \sqrt{1+\tau^2})}, & \text{si } \tau \geq 0 \\ \frac{1}{(\tau - \sqrt{1+\tau^2})}, & \text{si } \tau < 0 \end{cases}. \tag{2.3}$$

Así,

$$c = 1/(1 + t_{min}^2), s = t_{min}c. \tag{2.4}$$

Ejemplo de descomposición de Schur

A continuación se realiza la diagonalización de la siguiente matriz:

$$A = \begin{pmatrix} 1 & 3 \\ 3 & 5 \end{pmatrix}.$$

Se calcula τ

$$\tau = \frac{5-1}{2*3} = \frac{2}{3} \geq 0,$$

se determina t_{min} como

$$\begin{aligned} t_{min} &= \frac{1}{2/3 + \sqrt{1 + 4/9}} \\ &= \frac{1}{2/3 + \sqrt{13/9}} \\ &= 0.5351. \end{aligned}$$

Entonces, c y s son:

$$\begin{aligned} c &= \frac{1}{1 + (0.5351)^2} = 0.777 \\ s &= 0.5351 \cdot c = 0.4158. \end{aligned}$$

Se realiza la operación $J^T AJ$:

$$\begin{aligned} J^T AJ &= \begin{pmatrix} 0.777 & -0.4158 \\ 0.4158 & 0.777 \end{pmatrix} \begin{pmatrix} 1 & 3 \\ 3 & 5 \end{pmatrix} \begin{pmatrix} 0.777 & 0.4158 \\ -0.4158 & 0.777 \end{pmatrix} \\ &= \begin{pmatrix} 0.777 & -0.4158 \\ 0.4158 & 0.777 \end{pmatrix} \begin{pmatrix} -0.4704 & 2.7468 \\ 0.252 & 5.1324 \end{pmatrix} \\ &= \begin{pmatrix} -0.47 & 0 \\ 0 & 5.13 \end{pmatrix}. \end{aligned}$$

2.4.3. Rotación de Jacobi

Una matriz de rotación de Jacobi $J(p, q, \theta) \in \mathbb{R}^{n \times n}$ es lo mismo que una matriz de rotación de Givens $G(p, q, \theta)$, solo que se usa el término rotación de Jacobi, en honor al método de Jacobi y tiene la siguiente forma:

$$J(p,q,\theta) = \begin{pmatrix} \mathbf{1} & \dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ \mathbf{0} & \dots & \mathbf{c} & \dots & \mathbf{s} & \dots & \mathbf{0} \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ \mathbf{0} & \dots & \mathbf{-s} & \dots & \mathbf{c} & \dots & \mathbf{0} \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{1} \end{pmatrix} \begin{matrix} \\ \\ p \\ \\ q \\ \\ \\ \end{matrix}$$
$$\begin{matrix} & & p & & q & & \end{matrix} .$$

La matriz de rotación de Jacobi $J(p,q,\theta)$ es ortogonal $J^TJ = \mathbb{1}$ porque $c^2+s^2 = 1$. Supongamos una matriz simétrica $B \in \mathbb{R}^{n\times n}$ y $p,q \in \mathbb{N}$ con $p,q \in [1,n]$ y $p \neq q$. Usando el resultado de la subsección 2.4.2, se obtienen los valores c y s para la rotación $J(p,q,\theta)$, tal que, si $B' = J^TBJ$, entonces $b'_{pq} = 0$.

El resultado de $B' = BJ$ es solo la modificación de los vectores columna en p y q como:

$$\begin{aligned} b'_{ip} &= cb_{ip} - sb_{iq}, \\ b'_{iq} &= sb_{ip} + cb_{iq}. \end{aligned} \tag{2.5}$$

y el resultado de $B' = JB$ es solo la modificación de los vectores fila en p y q como:

$$\begin{aligned} b'_{pi} &= cb_{pi} + sb_{qi}, \\ b'_{qi} &= cb_{qi} - sb_{pi}. \end{aligned} \tag{2.6}$$

Así, la multiplicación de matrices de rotación de Jacobi se reduce a una operación de suma sobre vectores columna o filas y no de multiplicación entre matrices, disminuyendo así la cantidad de operaciones necesarias para realizar este cálculo.

2.4.4. Método de Jacobi para descomposición espectral

Supongamos una matriz simétrica $A \in \mathbb{R}^{n\times n}$. El algoritmo original de Jacobi, busca minimizar el siguiente valor:

$$off(A) = \sqrt{\sum_{i=1}^{n}\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}^2} = \sqrt{||A||_F^2 - \sum_{i=1}^{n} a_{ii}^2}.$$

Aplicando la norma de Frobenius a la matriz de 2×2 de la descomposición de Schur para el par de índices (p,q) con $1 \leq p < q \leq n$ para calcular la rotación de Jacobi $J(p,q,\theta)$:

$$\begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}.$$

Entonces, se obtiene la siguiente ecuación:

$$a_{pp}^2 + a_{qq}^2 + 2a_{pq}^2 = b_{pp}^2 + b_{qq}^2 + 2b_{pq}^2 = b_{pp}^2 + b_{qq}^2 a.$$

Usando el lema 2.2.6 y las ecuaciones 2.5 y 2.6 para $B = J^T AJ$ se obtiene lo siguiente [31]:

$$\begin{aligned} off(B)^2 &= ||B||_F^2 - \sum_{i=1}^{n} b_{ii}^2 \\ &= ||A||_F^2 - (b_{11}^2 + b_{22}^2 + ... + b_{pp}^2 + ... + b_{qq}^2 + ... + b_{nn}^2) \\ &= off(A)^2 + \sum_{i=1}^{n} a_{ii}^2 - (b_{11}^2 + b_{22}^2 + ... + b_{pp}^2 + ... + b_{qq}^2 + ... + b_{nn}^2) \\ &= off(A)^2 + (a_{pp}^2 + a_{qq}^2 - b_{pp}^2 - b_{qq}^2) \\ &= off(A)^2 - 2a_{pq}^2. \end{aligned} \tag{2.7}$$

De la ecuación 2.7 se observa que si se quieren reducir los elementos fuera de la diagonal, basta con eliminar el mayor de los elementos fuera de la diagonal. Este es el método clásico de Jacobi para descomposición espectral [33]. Sin embargo, es lento ya que se buscaría el mayor elemento en $\frac{n(n-1)}{2}$ elementos fuera de la diagonal por cada actualización.

Algoritmo cíclico del método de Jacobi

Dada una matriz simétrica $A \in \mathbb{R}^{n \times n}$ y una tolerancia positiva ϵ. El método de diagonalización de Jacobi cíclico por filas es [31]:

Algoritmo 2.2 Algoritmo cíclico por filas del método de Jacobi

Require: A, $U = \mathbb{1} \in \mathbb{R}^{n \times n}$, tol
  $\delta = tol \cdot ||A||_F^2$
  while $off(A) > \delta$ **do**
    for $p = 1 \to n - 1$ **do** ▷ Terminar el ciclo se conoce como una barrida
      for $q = p + 1 \to n$ **do**
        Calcular c, s para a_{pq}
        $A \leftarrow J^T AJ$ ▷ Usando las ecuaciones 2.5 y 2.6
        $U \leftarrow UJ$ ▷ Usando la ecuación 2.5

El método sobreescribe $A = U^T AU$, donde $U = J_1 ... J_i ...$ y U es ortogonal, este algoritmo converge cuadráticamente [34]. Se sugiere que la tolerancia este en el orden de 10^{-16} para precisión doble [33].

2.4.5. Método de Jacobi de dos lados

Supongamos $A \in \mathbb{R}^{m \times n}$, donde $m \geq n$ y $A = U\Sigma V^T$. De 2.1.3 tenemos que:

$$A^T A = V\Sigma^2 V^T. \tag{2.8}$$

Esto quiere decir que si $A^T A$ es diagonalizado con el algoritmo 2.2 para descomposición espectral, su diagonal contiene el cuadrado de los valores singulares de A y se obtiene V.

Es posible obtener U como $U = AV\Sigma^{-1}$, el inverso de Σ tiene como elementos el inverso multiplicativo de cada elemento en Σ. Sin embargo su número de condición es

$$\kappa(A^T A) = ||A^T A||||(A^T A)^{-1}|| \leq ||A||^2||(A^T A)^{-1}||.$$

2.4.6. Método de Jacobi de un lado

Supongamos $A \in \mathbb{R}^{m \times n}$, donde $m \geq n$, $A = U\Sigma V^T$ es su descomposición SVD y $B \in \mathbb{R}^{n \times n}$ la matriz simétrica $B = A^T A$. Si se aplica el método de Jacobi a B para obtener la matriz diagonal D:

$$\begin{aligned} D &= (...J_2^T J_1^T) B (J_1 J_2 ...) \\ &= Q^T B Q \\ \iff A^T A = B &= Q D Q^T. \end{aligned}$$

De la ecuación 2.8 se obtiene que $V = Q = J_1 J_2 ...$, entonces

$$\begin{aligned} A = U\Sigma V^T &= U\Sigma Q^T \\ \iff AQ = U\Sigma &= A(J_1 J_2 ...) = \hat{A}. \end{aligned}$$

Cuando B es aproximadamente diagonal, la matriz A tiene columnas mutuamente ortogonales, por lo que $\Sigma = diag(||\hat{a}_1||_2, ||\hat{a}_2||_2, ..., ||\hat{a}_n||_2)$ con $||\hat{a}_i||_2$ la norma euclidiana del i-ésimo vector columna de A. Entonces U se calcula como:

$$U = \hat{A}\Sigma^{-1}.$$

Por lo que no resulta necesario calcular la matriz B, sino calcular cada elemento conforme sea necesario para generar una matriz de rotación de Jacobi $J(p, q, \theta)$.

Algunas de las ventajas que se encuentran con este algoritmo, es que este método es más preciso, no toma mas de 6 iteraciones, el tiempo de ejecución depende menos de las dimensiones que el algoritmo de Jacobi por dos lados y su número de condición es $\kappa(A) = ||A||||(A)^{-1}||$ y $||A|| \leq ||A||^2$ [35].

2.4.7. Ordenamiento paralelo para rotaciones en el método de Jacobi por un lado

Definición 2.4.1 (Ordenamiento paralelo)**.** Sea $B \in \mathbb{R}^{n\times n}$ una matriz simétrica cuadrada. Un "sweep" o barrido es una iteración completa del método de Jacobi, i.e. rotaciones en todos los elementos por encima de la diagonal sin repetir. En la Figura 2.6 se ofrece una visualización de los elementos rotados en un barrido.

$$\begin{bmatrix} d_1 & \text{x} & \text{x} & \text{x} & \dots & \text{x} \\ \text{x} & d_2 & \text{x} & \text{x} & \text{x} & \text{x} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \text{x} & \text{x} & \text{x} & \text{x} & d_{n-1} & \text{x} \\ \text{x} & \text{x} & \text{x} & \text{x} & \text{x} & d_n \end{bmatrix}$$

Figura 2.6: Visualización de un barrido, donde x es una entrada donde se realizó una rotación.

Sea $C = \{x|x = (i, j)\ \forall i \in \{1, ..., n-1\}$, entonces $j \in \{i+1, ..., n\}\}$ el conjunto de todos los pares ordenados que denotan una rotación de Jacobi por encima de la diagonal, i.e. un barrido.

Una secuencia paralela es un conjunto $S \subseteq C$, tal que $\forall s_1, s_2 \in S \wedge s_1 \neq s_2$, entonces el par ordenado $s_1 = (s_1.x, s_1.y)$ y $s_2 = (s_2.x, s_2.y)$ cumple que

$$s_1.x \neq s_2.x \wedge s_1.y \neq s_2.y \wedge s_1.x \neq s_2.y \wedge s_1.y \neq s_2.x,$$

esto quiere decir que no hay índices repetidos.

Si tenemos l secuencias paralelas $S_1, ..., S_l$, tal que $S_i \neq \emptyset$, $(S_i \cap S_j) = \emptyset$ con $i \neq j$ y $S_1 \cup S_2 \cup ... \cup S_l = C$, entonces $S_1, ..., S_l$ definen un **ordenamiento paralelo**.

Lema 2.4.1. Una secuencia paralela S de rotaciones de Jacobi sobre $B = A^T A$ y V con $A \in \mathbb{R}^{m\times n}$ y $V \in \mathbb{R}^{n\times n}$ cumple con las condiciones de Bernstein.

Demostración. Dos declaraciones u y v son independientes y su orden no influye en la salida del programa si la siguientes condiciones se cumplen (Condiciones de Bernstein) [36]:

- $W(u) \cap W(v) = \emptyset$
- $W(u) \cap R(v) = \emptyset$
- $R(u) \cap W(v) = \emptyset$

donde $R(u)$ es el conjunto de celdas de direcciones de memoria de la cual se lee la información durante la ejecución de u y $W(u)$ es el conjunto de celdas de direcciones de memoria donde se escribe la información durante la ejecución de u.

Sea $s_1, s_2 \in S$ con $s_1 \neq s_2$, dos puntos arbitrarios tomados de la misma secuencia paralela S y sea $B = A^T A$ con $A \in \mathbb{R}^{m\times n}$. Probaremos que las declaraciones u y v como se muestran en el algoritmo 2.3 cumplen con las condiciones de Bernstein.

Algoritmo 2.3 Declaraciones u, v

Require: $A \in \mathbb{R}^{m\times n}, V \in \mathbb{R}^{n\times n}$

1: $RotacionJacobi(s_1)$ ▷ Declaración u
2: $RotacionJacobi(s_2)$ ▷ Declaración v
3: **procedure** RotacionJacobi(s)
4: $(i, j) \leftarrow (s.x, s.y)$
5: $a \leftarrow \sum_{k=1}^{n} a_{ki}^2$ ▷ Calcular los elementos $\begin{bmatrix} a & c \\ c & b \end{bmatrix} \equiv (i, j) \in A^T A$
6: $b \leftarrow \sum_{k=1}^{n} a_{kj}^2$
7: $c \leftarrow \sum_{k=1}^{n} a_{ki} a_{kj}$
8:
9: $\zeta \leftarrow (b - 1)/(2c)$ ▷ Calcular la rotación de Jacobi que diagonaliza $\begin{bmatrix} a & c \\ c & b \end{bmatrix}$
10: $t \leftarrow sign(\zeta)/(|\zeta| + \sqrt{1 + \zeta^2})$
11: $cs \leftarrow 1/\sqrt{1 + t^2}$
12: $sn \leftarrow cs * t$
13:
14: **for** $k = 1, ..., m$ **do** ▷ Actualizar las columnas i y j de A
15: $tmp \leftarrow a_{ki}$
16: $a_{ki} \leftarrow cs * tmp - sn * a_{kj}$
17: $a_{kj} \leftarrow sn * tmp + cs * a_{kj}$
18: **for** $k = 1, ..., n$ **do** ▷ Actualizar las columnas i y j de V
19: $tmp \leftarrow v_{ki}$
20: $v_{ki} \leftarrow cs * tmp - sn * v_{kj}$
21: $v_{kj} \leftarrow sn * tmp + cs * v_{kj}$

Se empieza determinando el conjunto $R(u)$, las lecturas de A y V ocurren en las líneas $5, 6, 7, 15, 16, 17, 19, 20, 21$ del algoritmo 2.3. Las líneas $5, 6, 7, 15, 16, 17$ del algoritmo 2.3 leen los vectores columna $a_{s_1.x}, a_{s_1.y}$ y las líneas $19, 20, 21$ del algoritmo 2.3 leen los vectores columna $v_{s_1.x}, v_{s_1.y}$. Así, $R(u) = \{a_{s_1.x}, a_{s_1.y}, v_{s_1.x}, v_{s_1.y}\}$.

Determinando el conjunto $W(u)$ con las escrituras sobre A y V ocurren en las líneas $16, 17, 20, 21$ del algoritmo 2.3. Las líneas $16, 17$ del algoritmo 2.3 escriben sobre los vectores columna $a_{s_1.x}, a_{s_1.y}$ y las líneas $20, 21$ del algoritmo 2.3 escriben los vectores columna $v_{s_1.x}, v_{s_1.y}$. Así, $W(u) = \{a_{s_1.x}, a_{s_1.y}, v_{s_1.x}, v_{s_1.y}\}$.

De manera análoga, se obtiene

$$R(v) = \{a_{s_2.x}, a_{s_2.y}, v_{s_2.x}, v_{s_2.y}\}, W(v) = \{a_{s_2.x}, a_{s_2.y}, v_{s_2.x}, v_{s_2.y}\}.$$

Por definición se tiene que $s_1.x \neq s_2.x \wedge s_1.y \neq s_2.y \wedge s_1.x \neq s_2.y \wedge s_1.y \neq s_2.x$.
$\forall w_u \in W(u) \wedge \forall w_v \in W(v) \implies w_u \neq w_v$. Así, $W(u) \cap W(v) = \emptyset$.
$\forall w_u \in W(u) \wedge \forall r_v \in R(v) \implies w_u \neq r_v$. Así, $W(u) \cap R(v) = \emptyset$.
$\forall r_u \in R(u) \wedge \forall w_v \in W(v) \implies r_u \neq w_v$. Así, $R(u) \cap W(v) = \emptyset$. □

Ordenamiento paralelo de Ahmed

El ordenamiento paralelo de Ahmed Sameh tiene la ventaja que cada secuencia paralela tiene la misma cardinalidad. En el caso de que n es par tiene $n/2$ elementos, si n es impar $\lfloor n/2 \rfloor$. Esto permite que la carga por ordenamiento sea la misma.

A continuación se muestra el ordenamiento paralelo de Ahmed para una matriz simétrica de 9×9, donde cada número $k = 1, ..., 9$ representa el ordenamiento paralelo S_k.

$$\begin{pmatrix} \text{x} & 4 & 8 & 3 & 7 & 2 & 6 & 1 & 5 \\ & \text{x} & 3 & 7 & 2 & 6 & 1 & 5 & 9 \\ & & \text{x} & 2 & 6 & 1 & 5 & 9 & 4 \\ & & & \text{x} & 1 & 5 & 9 & 4 & 8 \\ & & & & \text{x} & 9 & 4 & 8 & 3 \\ & & & & & \text{x} & 8 & 3 & 7 \\ & & & & & & \text{x} & 7 & 2 \\ & & & & & & & \text{x} & 6 \\ & & & & & & & & \text{x} \end{pmatrix}.$$

El ordenamiento paralelo se construye con el algoritmo 2.4 [37].

Algoritmo 2.4 Ordenamiento paralelo de [37]

Require: n
  $m \leftarrow \lfloor (n+1)/2 \rfloor$
  for $k = 1, ..., m-1$ **do**
    $S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
    for $q = m-k+1, ..., n-k$ **do**
      if $m-k+1 \leq q \leq 2m-2k$ **then**
        $p \leftarrow (2m-2k+1)-q$
      else if $2m-2k < q \leq 2m-k-1$ **then**
        $p \leftarrow (4m-2k)-q$
      else if $2m-k-1 < q$ **then**
        $p \leftarrow n$
      $S_k \cup \{(p,q)\}$ ▷ Agregar el par (p,q) a la secuencia paralela S_k
  for $k = m, ..., 2m-1$ **do**
    $S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
    for $q = 4m-n-k, ..., 3m-k-1$ **do**
      if $q < 2m-k+1$ **then**
        $p \leftarrow n$
      else if $2m-k+1 \leq q \leq 4m-2k-1$ **then**
        $p \leftarrow (4m-2k)-q$
      else if $4m-2k-1 < q$ **then**
        $p \leftarrow (6m-2m-1)-q$
      $S_k \cup \{(p,q)\}$ ▷ Agregar el par (p,q) a la secuencia paralela S_k

Por las ventajas mencionadas en la subsección 2.4.6 y por la facilidad del ordenamiento paralelo, se eligió al método de Jacobi por un lado como el método a implementar junto con el ordenamiento paralelo de [37].

Capítulo 3

Programación paralela

En este capítulo se presentan las arquitecturas de computación paralela y sus modelos de programación. Por cada modelo de programación paralela se mencionan definiciones, ventajas y herramientas. Después, se describe en una sección algunas combinaciones de los modelos de programación anteriormente explicados. En la última sección se explican con ejemplos el uso de las herramientas OpenMP, MPI y CUDA, las cuales fueron las seleccionadas para realizar la implementación.

3.1. Arquitecturas de cómputo paralelas en HPC

El cómputo de alto desempeño (o por sus siglas en inglés HPC que es *High Performance Computing*) o supercómputo, es una clase especial de computación donde el tiempo de ejecución es el objetivo prioritario a minimizar usando múltiples paradigmas, prácticas y tecnologías [38].

Las arquitecturas de cómputo paralelas están divididos en dos [39]:

- Paralelismo en sistemas de memoria. Es una arquitectura de replicación de computadoras con alguna forma de comunicación de datos entre ellos. Los dos tipos de memoria que se presentarán son los de memoria compartida y distribuida.

- Paralelismo en CPU. Los CPU ya cuentan con diferentes tipos de paralelismo el más común es la segmentación (o en inglés pipeline), otro tipo de paralelismo con el que cuentan es con las instrucciones vectoriales, por ejemplo la extensión vectorial avanzada (por sus siglas en inglés AVX) de la arquitectura de x86.

Sin embargo, en la actualidad ha estado cobrando relevancia otra arquitectura de cómputo paralela que usa a las unidades de procesamiento de gráficos (o por sus siglas en inglés GPU) como aceleradores de cálculo. Su modelo de programación es el paralelo heterogéneo y es del tipo en la taxonomia de flynn como un flujo o una sola corriente de instrucciones sobre múltiples datos (o por sus siglas en inglés SIMD o Single Instruction Multiple Data) [40].

Dada una arquitectura de computadora y sus características, se abstrae esta arquitectura y se realiza un modelo de programación que es un conjunto de técnicas de programación, herramientas, bibliotecas que usan las principales características de esta arquitectura. Por ejemplo, en las arquitecturas paralelas de memoria compartida, se puede usar el modelo de programación fork/join para ejecución asíncrona de hilos y sincronización, este modelo abstrae los mecanismos que usa el sistema operativo y el hardware para realizar las operaciones fork y join [38].

3.2. Programación paralela en sistemas de memoria compartida

Las arquitecturas de memoria compartida es la replicación de CPUs sobre un solo espacio de direcciones de memoria (respetando la jerarquía de memoria habitual de una arquitectura de computadora con sus registros, diferentes niveles de cache y memoria virtual), la implementación de esta arquitectura se le llama multiprocesador de memoria compartida. El acceso a la memoria es simple mediante las instrucciones usuales del conjunto de instrucciones usado [39].

Un multiprocesador de memoria compartida tienen múltiples procesadores donde cada procesador tiene sus registros y cache. Todos los procesadores pueden acceder al mismo espacio de direcciones de la memoria lo que dificulta la coherencia de datos. Este tipo de computadora se cataloga como múltiples flujos de instrucciones sobre múltiples datos (o por sus siglas en inglés MIMD) [39][41][42].

A nivel programación, el sistema operativo opera sobre estos procesadores y provee a usuarios la funcionalidad de creación de hilos como forma de operar paralelamente estos procesadores. Esto se le conoce como paralelismo a nivel de hilos donde se ejecutan tareas computacionales simultaneas [41][42].

3.2.1. Hilos

Un hilo es el estado de un proceso que es calendarizado al CPU. Todos los hilos creados por un proceso viven en el espacio de memoria de este proceso, cada hilo tiene como información de su estado su contador de programa, registros, pila y otros datos. El paralelismo ocurre cuando cada hilo es asignado por el calendarizador del sistema operativo a cada procesador como se visualiza en la Figura 3.1 [41][43][44].

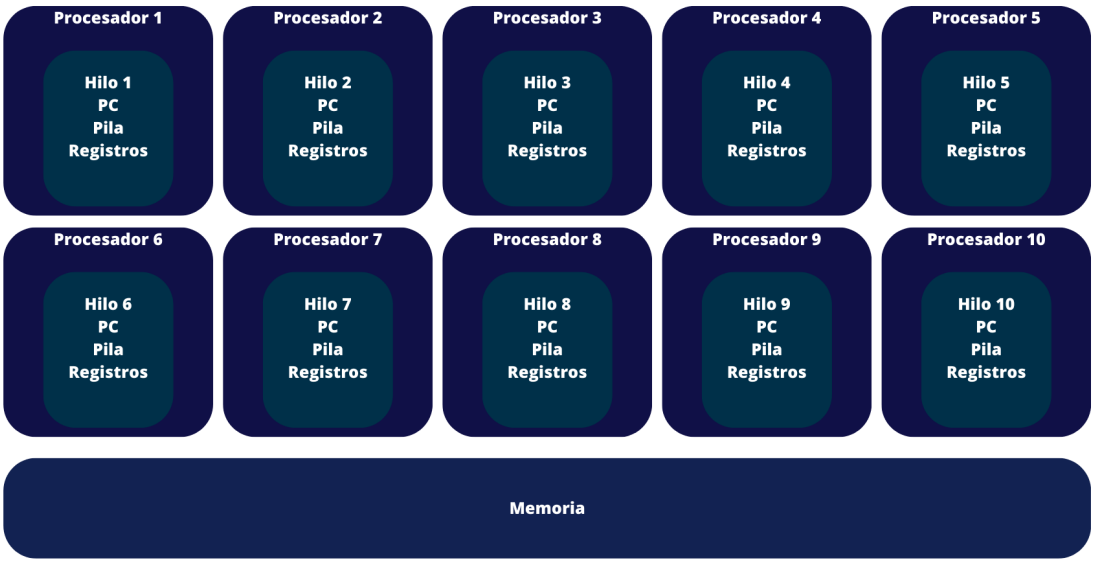

Figura 3.1: 10 hilos mapeados a los 10 procesadores de un CPU para generar paralelismo.

3.2.2. Cuellos de botella

Algunos de los problemas usuales encontrados al usar el modelo de programación paralela son:

- Granularidad. La tarea que realiza un hilo puede ser muy grande que la administración del pedazo de trabajo es mucho; o puede ser tan pequeño que dejamos a los procesadores esperando más tareas [41].
- Uso compartido falso. Modificar un bloque de cache produce una eliminación de otros elementos a modificar.
- Sincronización. Accesar o modificar un dato genera comportamientos o resultados no esperados [36].

3.2.3. Herramientas de programación

La especificación OpenMP (de Open Multi-Processing) contiene directivas de compilación, bibliotecas y variables de ambiente en C/C++ y Fortran. El modelo de ejecución es por equipo de hilos donde existe un hilo maestro que ejecuta de forma secuencial hasta que una directiva de compilación indica que una región de código es paralelizado por un equipo de hilos esclavos del hilo maestro como se muestra en la Figura 3.2[41][38][36].

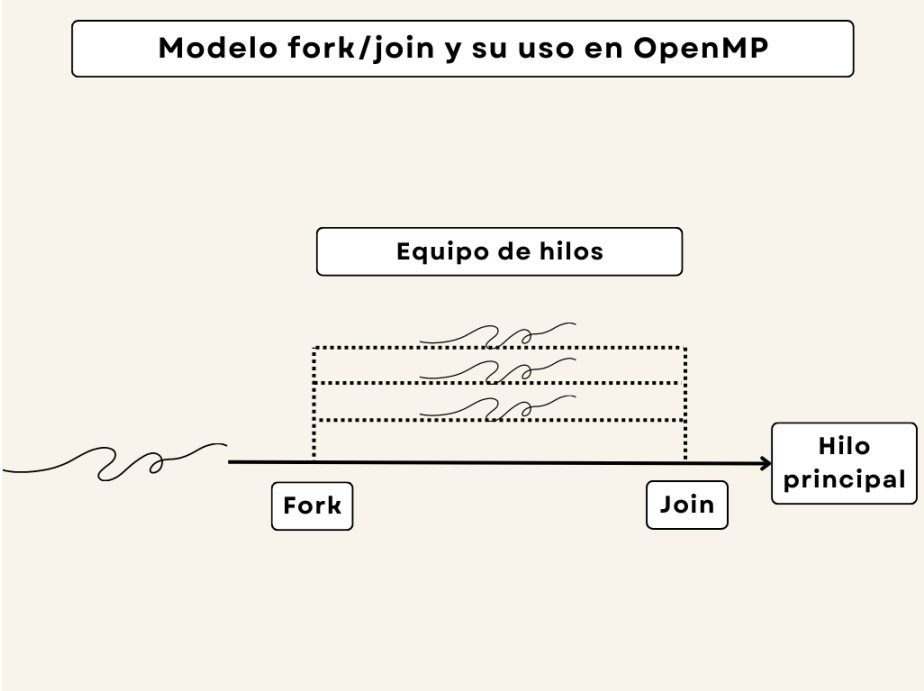

Figura 3.2: Visualización de fork y join de hilos en programación paralela con OpenMP.

La biblioteca pthreads es una interfaz que permite la creación, manejo y destrucción de hilos en un proceso en linux. pthreads puede realizar una ejecución paralela usando ciertas funciones de la interfaz que configuran esta capacidad. Existen otras herramientas para realizar programación paralela en sistemas de memoria compartida Chapel, MPI (Message Passing Interface), entre otros [45].

3.3. Programación paralela en sistemas con memoria distribuida

Un sistema de memoria distribuida es un conjunto de computadoras (llamado cluster) individuales (llamados nodos) interconectadas por una red donde cada computadora es capaz de ejecutar programas. El modelo de programación consiste de procesos separados en cada nodo con paso de datos entre ellos a través de un comunicador [39][38]. Esta programación paralela se usa como método de distribución de datos para que cada nodo ejecute una tarea [45]. Se muestra en la Figura 3.3 una ilustración de un sistema de memoria distribuida con su red de comunicación y conformado por 5 nodos.

Figura 3.3: Cluster con 5 nodos interconectados.

3.3.1. Cuellos de botella

Existen varios cuellos de botella en la programación distribuida como lo son [41]:

- Sobrecoste. El uso de bibliotecas de programación distribuida genera un sobrecoste.
- Latencia. El sistema está restringido a la velocidad de la red. En HPC comúnmente se usa una red infiniband.
- Balanceo de carga. Si el sistema es heterogéneo, es decir, cuenta con nodos que son diferentes en capacidad de computación y/o memoria entonces ocurrirá que un nodo ocupe (o no ocupe) toda su capacidad de cómputo.

3.3.2. Herramientas de programación

La interfaz de paso de mensajes (o por sus siglas en inglés MPI) es un estandar para arquitecturas de memoria distribuida. Se escribe un programa de alto nivel que se corre en todos los nodos. Se usa un mecanismo de paso de mensajes usando funciones y también se permite acceder de manera paralela a recursos de entrada y salida como son archivos [36].

3.4. Programación paralela heterogénea

El cómputo sobre estos sistemas son en unidades de hardware con alta capacidad paralela que ejecutan paralelamente de manera masiva un solo flujo de instrucción sobre múltiples datos [38][40]. La programación paralela heterogénea se realiza sobre unidades de procesamiento de gráficos (por sus siglas en inglés GPU).

Unidad de procesamiento de gráficos

Los GPUs son unidades de procesamiento que tienen miles de procesadores con muy poca lógica de control, menor cache y con un calendarizador de flujo de instrucciones.

NVIDIA llama a su modelo de programación como única instrucción múltiples hilos porque se programa una función (o un flujo de instrucciones) que se calendarizan en los procesadores (o multiprocesador de flujo en el caso de NVIDIA) en un mismo espacio de direcciones de memoria. NVIDIA separa en un mismo código pero con diferentes claves la ejecución de un programa para CPU y GPU, o como NVIDIA le llama un anfitrión y dispositivo, respectivamente [46][40]. En la Figura 3.4 se muestra un esquema de arquitectura de una GPU NVIDIA inspirado en una imagen de la arquitectura Pascal [3].

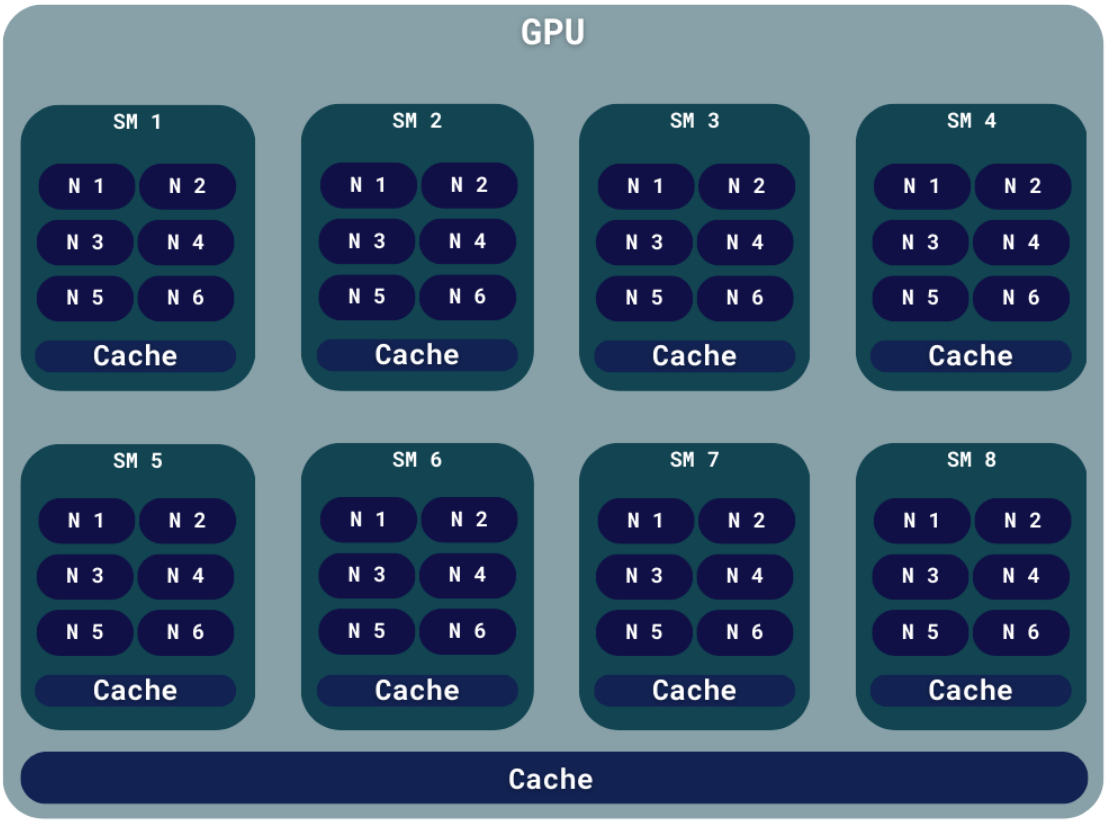

Figura 3.4: Visualización de una unidad de procesamiento de gráficos con 8 multiprocesadores inspirado en la página [3].

3.4.1. Cuellos de botella

Existen varios cuellos de botella en la programación paralela heterogénea en GPUs como lo son:

- Transferencia de datos. Existe una latencia de transferencia entre datos del anfitrión al dispositivo y viceversa.
- Sincronización. En la programación heterogénea en GPUs no es trivial implementar secciones críticas, cierres de exclusión mutuas, entre otros.
- Granularidad. Los GPUs son dispositivos que son excelentes para problemas altamente paralelizables sobre arreglos, por lo que se usa para tareas de grano fino.

3.4.2. Herramientas de programación

La plataforma de programación que se usó en este trabajo es la arquitectura unificada de dispositivo de cómputo (o por sus siglas en inglés CUDA Compute Unified Device Architecture) desarrollado por NVIDIA y que tiene diversas herramientas como compiladores, especificaciones, bibliotecas para programar sobre GPUs [47].

Otras herramientas que permiten la programación heterogénea en GPUs son OpenACC y OpenCL.

3.5. Combinaciones de modelos de programación

Hasta ahora se han presentado los modelos de programación usadas y sus arquitecturas de cómputo de alto desempeño como modelos separados, en realidad se pueden mezclar estás técnicas de aceleración de un algoritmo. La programación híbrida es la que se refiere cuando juntamos dos modelos de programación [39].

3.5.1. Programación híbrida paralela en memoria distribuida y compartida

La programación híbrida paralela en memoria distribuida y compartida o también llamada híbrida paralela es la mezcla de la arquitectura de memoria distribuida donde cada nodo conectado cuenta con un multiprocesador. A nivel programación esto sería combinar las herramientas MPI con OpenMP.

3.5.2. Programación paralela de memoria distribuida heterogénea: MPI + CUDA

La programación paralela de memoria distribuida heterogénea es la mezcla de la arquitectura de memoria distribuida, donde cada nodo usa su(s) GPU(s) como se

detalla en la sección 3.4. A nivel programación esto sería combinar las herramientas MPI y CUDA.

3.5.3. Programación paralela de memoria compartida heterogénea: OpenMP + CUDA

La programación paralela de memoria compartida heterogénea es la mezcla de la arquitectura de memoria compartida y la paralela heterogénea. En este caso, una computadora con un multiprocesador tiene una o varias GPUs. A nivel programación esto sería combinar las herramientas OpenMP y CUDA.

3.5.4. Programación paralela híbrida heterogénea: MPI + OpenMP + CUDA

La programación paralela híbrida heterogénea es la mezcla de la arquitectura de memoria distribuida, donde cada nodo usa una arquitectura de memoria compartida paralela junto con el modelo de programación paralela heterogénea.

El modelo híbrido que se usará es MPI + OMP + CUDA para distribuir datos, paralelizar la ejecución de tareas y usar los GPU(s) como dispositivo(s) de cómputo paralelo sobre algunas operaciones del método numérico. La Figura 3.5 ejemplifica el tipo de cluster para el cual se desarrollará la implementación.

3.6. Herramientas de programación paralela usadas

En esta sección se especifican y ejemplifican las herramientas usadas de modelos de programación paralela en arquitecturas de memoria distribuida y compartida, además de la programación paralela heterogénea.

3.6.1. Programación paralela en memoria compartida con OpenMP

Se usó para programación en sistemas de memoria compartida la especificación OpenMP (OMP). OpenMP hace uso de equipo de hilos para paralelizar regiones de código. Además es basada por directivas de compilador, donde se le indica al compilador específicamente donde y como se paralelizará una región de código [45][48].

Todas las directivas de compilación de OpenMP se escriben como `#pragma omp` *directive-especification*. A continuación se describen las únicas dos directivas que se usaron en el desarrollo

Directiva de paralelización de un ciclo for

Una directiva que se usó en este trabajo fue la de paralelización de un ciclo for `#pragma omp parallel for` *directive-especification*. En el código 3.1 se muestra una sección de código donde se usó.

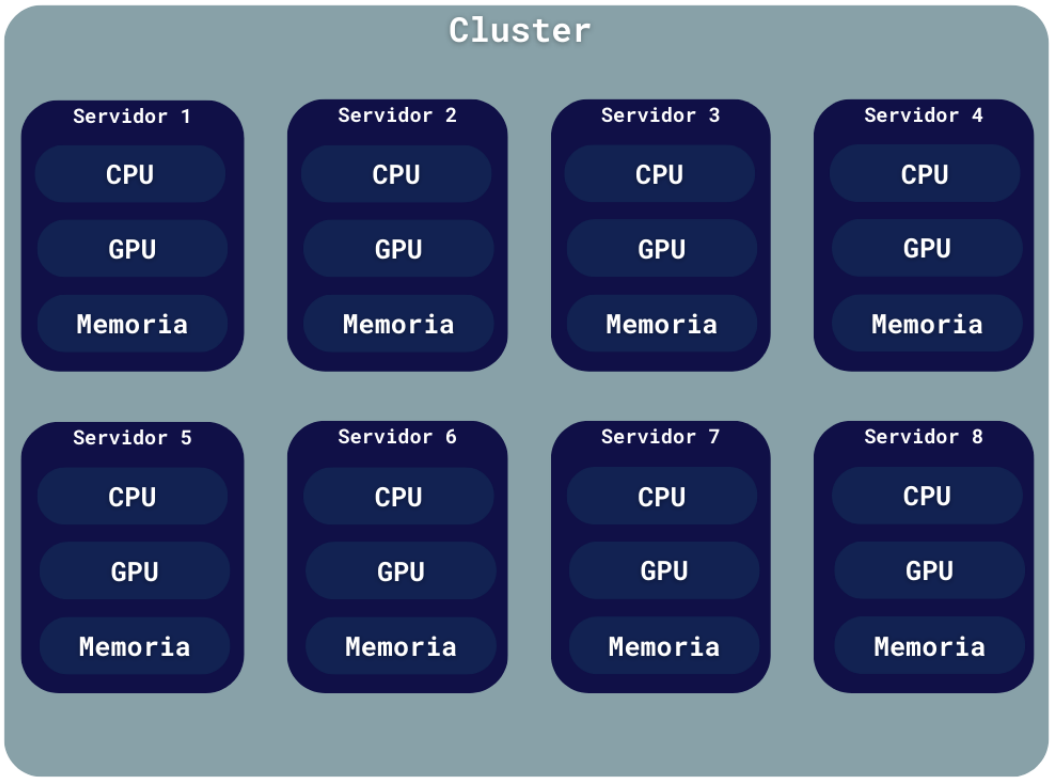

Figura 3.5: Visualización de un cluster, donde cada nodo cuenta con una GPU.

```
1  // Parallel ordering S_k
2  for (size_t k = m_ordering; k < 2 * m_ordering; ++k) {
3    size_t p = 0;
4    size_t p_trans = 0;
5    size_t q_trans = 0;
6
7    // Create l threads teams with directive
8    #pragma omp parallel for private(p, p_trans, q_trans)
9    for (size_t q = (4 * m_ordering) - n - k; q < (3 * m_ordering) - k; ++q) {
10 ...
```

Código 3.1: Directiva de paralelización de un ciclo for.

Directiva de alcance de datos

Por defecto en OpenMP, todo los datos fuera de una región de código paralelizado es compartido, esto quiere decir que modificar las variables declaradas fuera de la región de código paralelizado dentro de la región modifica su valor dentro y fuera de la región.

Si se desean usar variables dentro de la región paralelizada que se encuentran fuera de la región sin modificar el valor que contiene, entonces se usa la directi-

va de alcance de datos privados `#pragma omp private(`*`variables-comma-separated`*`)` *directive-especification*, esta directiva inicializa por defecto la variable para su uso dentro de la región

En el código 3.1 se ilustra como en la directiva de paralelización del ciclo for se tiene una copia privada para cada hilo de las variables `p`, `p_trans` y `q_trans`.

3.6.2. Programación paralela heterogénea con CUDA

El marco de trabajo CUDA permite al anfitrión (CPU) hacer uso del dispositivo (GPU) a través de funciones que se llaman kernels (flujo de instrucciones) que son ejecutados por miles de hilos CUDA [47].

Los kernels son calendarizados por jerarquías de hilos. El usuario puede definir arreglos de hilos hasta en 3 dimensiones como la Figura 3.6.

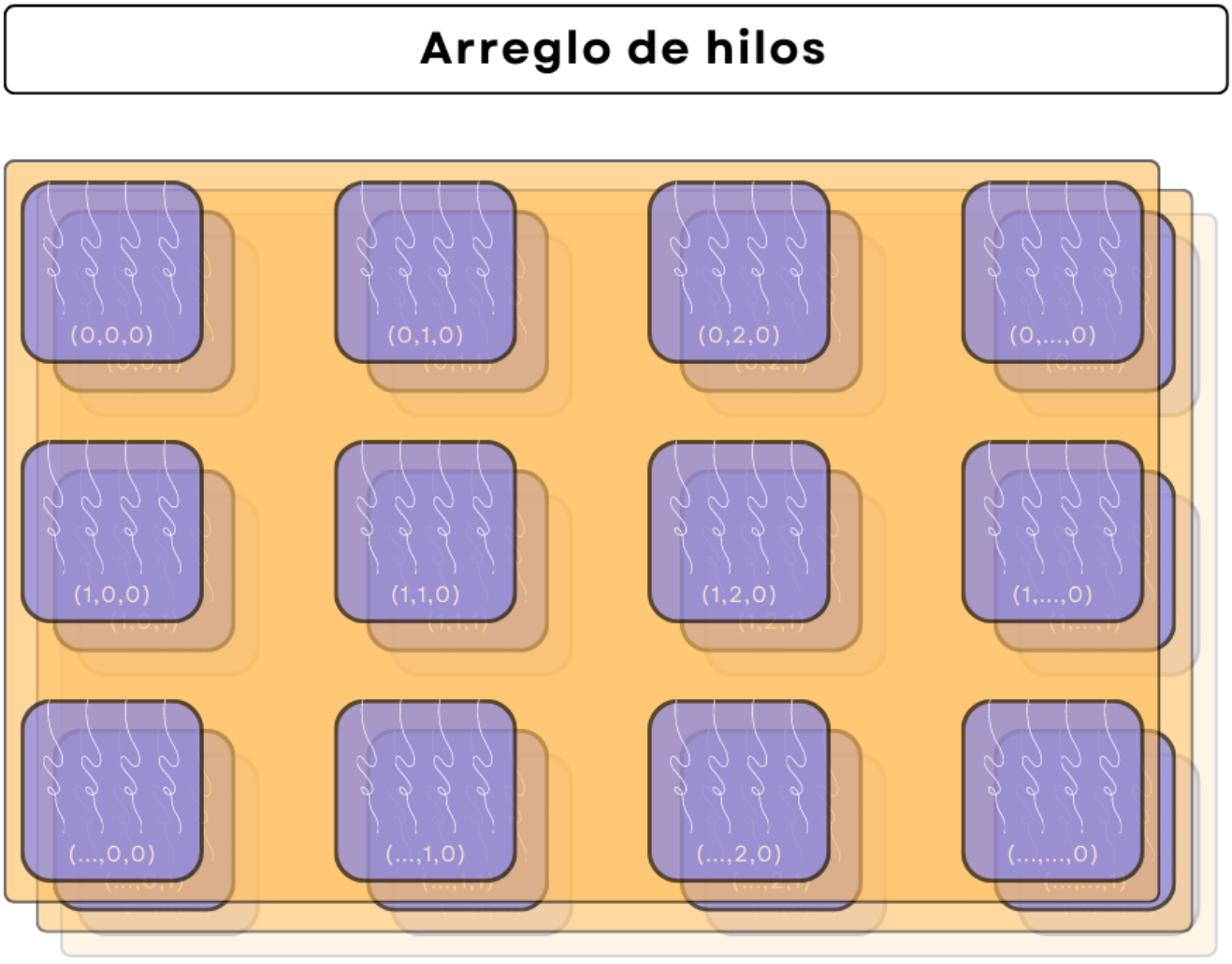

Figura 3.6: Visualización de un arreglo tridimensional de hilos CUDA.

Los bloques de hilos son arreglos de arreglos de hilos que pueden ser de hasta tres dimensiones como se muestra en la Figura 3.7 y que son calendarizados a la GPU.

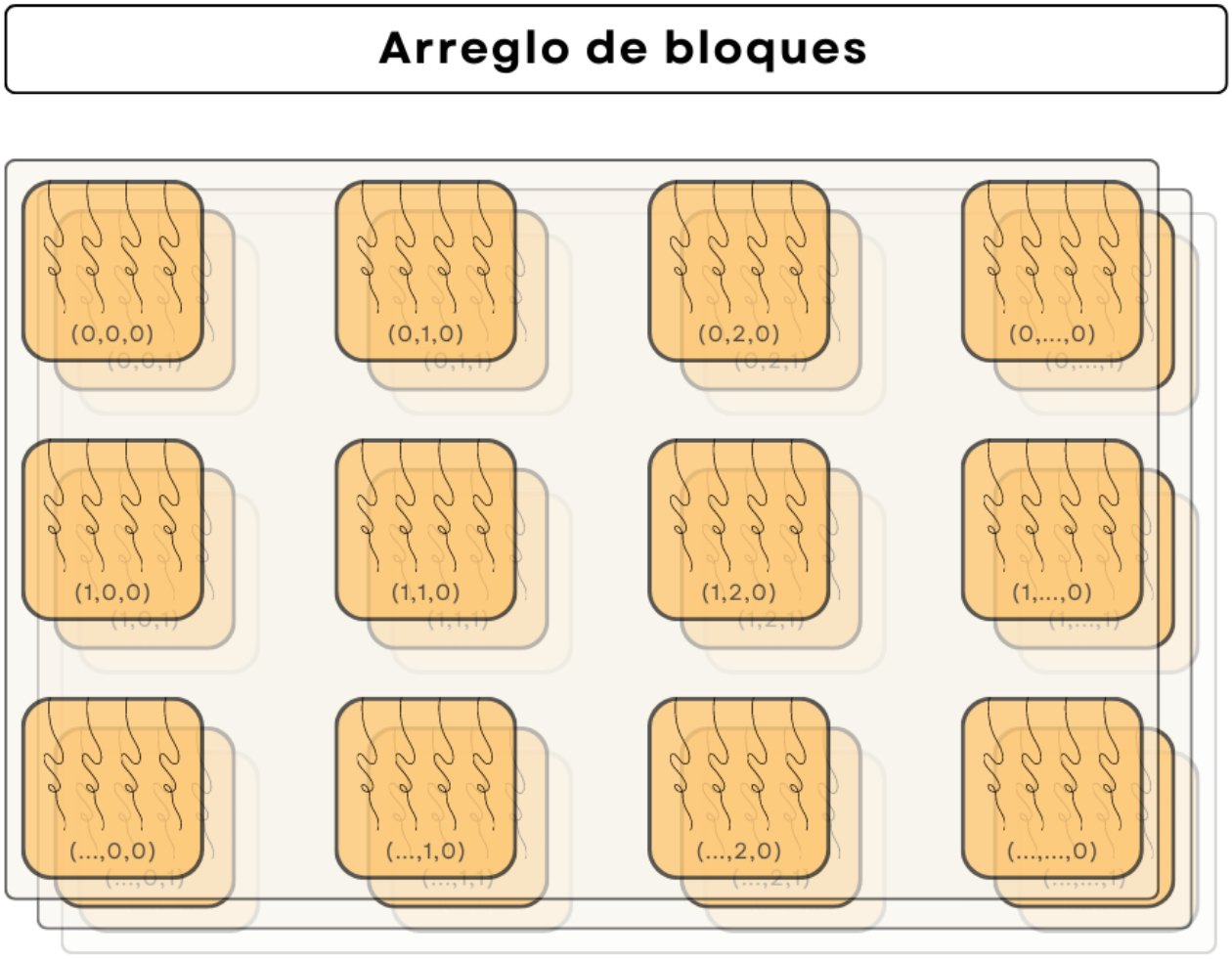

Figura 3.7: Visualización de un arreglo tridimensional de bloques de hilos CUDA.

Los kernels solo pueden operar sobre datos en el dispositivo, para esto CUDA provee de funciones que permiten crear memoria en el dispositivo y mover memoria entre anfitrión y dispositivo.

En el código 3.2 se presenta la estructura de datos `CUDAMatrix` donde tiene varios constructores y métodos que usan la función de creación de memoria en el dispositivo `cudaError_t cudaMalloc ( void** devPtr, size_t size )` y la de movimiento de memoria `cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind )`.

```
1  struct CUDAMatrix{
2    unsigned long width;
3    unsigned long height;
4    double *elements;
5
6    CUDAMatrix(unsigned long height, unsigned long width){
```

```
7       this->width = width;
8       this->height = height;
9
10      cudaMalloc(&this->elements, height * width * sizeof(double));
11      cudaMemset(&this->elements, 0, height * width * sizeof(double));
12    }
13
14    CUDAMatrix(double *arr, size_t length){
15      this->height = 1;
16      this->width = length;
17      cudaMalloc(&this->elements, length * sizeof(double));
18      cudaMemcpy(this->elements, arr, length * sizeof(double),
19                 cudaMemcpyHostToDevice);
20    }
21
22    explicit CUDAMatrix(Matrix &matrix){
23      this->width = matrix.width;
24      this->height = matrix.height;
25
26      cudaMalloc(&this->elements, height * width * sizeof(double));
27      cudaMemcpy(this->elements, matrix.elements, this->height * this->width *
            sizeof(double),
28                 cudaMemcpyHostToDevice);
29    }
30
31    void copy_to_host(Matrix &matrix) const{
32      cudaMemcpy(matrix.elements,
33                 this->elements,
34                 matrix.width * matrix.height * sizeof(double),
35                 cudaMemcpyDeviceToHost);
36    }
37
38    void copy_to_host(double *arr, size_t length) const{
39      cudaMemcpy(arr, this->elements, length * sizeof(double),
            cudaMemcpyDeviceToHost);
40    }
41
42    void copy_from_host(Matrix &matrix) const{
43      cudaMemcpy(this->elements,
44                 matrix.elements,
45                 width * height * sizeof(double),
46                 cudaMemcpyHostToDevice);
47    }
48
49    void copy_from_device(CUDAMatrix &matrix) const{
50      cudaMemcpy(this->elements,
51                 matrix.elements,
52                 width * height * sizeof(double),
53                 cudaMemcpyDeviceToDevice);
54    }
55
56    void free() const{
57      cudaFree(this->elements);
58    }
59  };
```

Código 3.2: Estructura de datos `CUDAMatrix`.

3.6.3. Programación paralela en memoria distribuida con MPI

El estandar MPI en principio fue diseñado para uso en arquitecturas de memoria distribuida, pero también puede usarse en un solo nodo con varios multiprocesadores

con memoria no compartida o NUMA; también se puede usar en arquitecturas de memoria compartida [49].

En MPI, el tamaño del mundo es el número de rangos que hay en el sistema donde se ejecutan los procesos MPI. Un rango es donde se ejecuta un proceso MPI, estos pueden ser un nodo, un multiprocesador, un procesador, entre otros.

MPI proporciona funciones globales para información del contexto de ejecución, comunicaciones colectivas y de punto a punto. En nuestro caso solo se usaron las funciones para obtener el contexto de ejecución y comunicación punto a punto. En el código 3.3 se muestra como se envían y reciben datos por MPI.

```
1  if(rank == ROOT_RANK){
2    // Create matrix by rank and send
3    for(auto index_rank = 1; index_rank < size; ++index_rank){
4      std::vector<double> tmp_matrix;
5      ...
6      MatrixMPI A_rank(tmp_matrix, m, root_set_columns_by_rank[index_rank].size()
           );
7      tmp_matrix.clear();
8      auto return_status = MPI_Send(A_rank.elements, m * root_set_columns_by_rank
           [index_rank].size(), MPI_DOUBLE, index_rank, 0, MPI_COMM_WORLD);
9      if(return_status != MPI_SUCCESS){
10       std::cout << "problem on MPI_Send on rank: " << rank << ", return status"
             << return_status << "\n";
11     }
12
13     A_rank.free();
14   }
15 } else {
16   MPI_Status status;
17   auto return_status = MPI_Recv(A_local.elements, m * local_set_columns.size(),
         MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
18   if(return_status != MPI_SUCCESS){
19     std::cout << "problem on MPI_Recv on rank: " << rank << ", return status"
           << return_status << "\n";
20   }
21 }
```

Código 3.3: Envío y recibo de datos en MPI.

Capítulo 4

Desarrollo de las implementaciones paralelas

El principal objetivo de la tesis es desarrollar una implementación híbrida paralela heterogénea de la descomposición SVD y comparar con otras implementaciones en otros modelos de programación. En este trabajo se realizaron implementaciones paralela en memoria compartida, paralela en memoria compartida heterogénea, híbrida paralela e híbrida paralela heterogénea. A continuación en cada sección se describe profundamente cada implementación usando pseudo-algoritmos y algunas partes del código fuente. En una sección especial se describe profundamente el esquema de distribución de datos para las implementaciones en memoria distribuida. Todas las secciones se enfocan en una iteración del método.

4.1. Método de Jacobi por un lado secuencial

Todas las implementaciones desarrolladas están basadas en el pseudo-algoritmo propuesto por LAPACK con variaciones para hacerlo más eficiente [35]. A diferencia del propuesto por LAPACK, nuestra implementación solo realiza una iteración del método ya que nuestro objetivo es codificar una implementación rápida y eficiente más que proponer una condición de paro o analizar la convergencia de dicho método.

El algoritmo 4.5 es el pseudo-algoritmo del método de Jacobi por un lado propuesto por LAPACK [35]:

Algoritmo 4.5 Una iteración del algoritmo de Jacobi por un lado [35]

Require: $A \in \mathbb{R}^{m \times n}, V \in \mathbb{R}^{n \times n}$

1: **for all** pares indexados (i, j) en la porción triangular superior con $i \neq j$ **do**
2: $a \leftarrow \sum_{k=1}^{n} a_{ki}^2$ ▷ Calcular los elementos $\begin{bmatrix} a & c \\ c & b \end{bmatrix} \equiv (i, j) \in A^T A$
3: $b \leftarrow \sum_{k=1}^{n} a_{kj}^2$
4: $c \leftarrow \sum_{k=1}^{n} a_{ki} a_{kj}$
5:
6: $\zeta \leftarrow (b - 1)/(2c)$ ▷ Calcular la rotación de Jacobi que diagonaliza $\begin{bmatrix} a & c \\ c & b \end{bmatrix}$
7: $t \leftarrow sign(\zeta)/(|\zeta| + \sqrt{1 + \zeta^2})$
8: $cs \leftarrow 1/\sqrt{1 + t^2}$
9: $sn \leftarrow cs * t$
10:
11: **for** $k = 1, ..., m$ **do** ▷ Actualizar las columnas i y j de A
12: $tmp \leftarrow a_{ki}$
13: $a_{ki} \leftarrow cs * tmp - sn * a_{kj}$
14: $a_{kj} \leftarrow sn * tmp + cs * a_{kj}$
15: **for** $k = 1, ..., n$ **do** ▷ Actualizar las columnas i y j de V
16: $tmp \leftarrow v_{ki}$
17: $v_{ki} \leftarrow cs * tmp - sn * v_{kj}$
18: $v_{kj} \leftarrow sn * tmp + cs * v_{kj}$
19: ▷ Los valores singulares son las normas de los vectores columnas en A
20: ▷ Los vectores singulares por la izquierda son las columnas normalizadas de A con los valores singulares

4.1.1. Cambios para eficiencia de implementación

Los cambios presentados a continuación se encuentran en todas las implementaciones.

Cálculo de los elementos para realizar la rotación sobre b_{pq} en $B = A^T A$

Se observa que para el cálculo de los elementos

$$\begin{bmatrix} b_{pp} & b_{pq} \\ b_{pq} & b_{qq} \end{bmatrix}.$$

Es realizado con las sumatorias en las líneas $2, 3, 4$ en el pseudo-algoritmo 4.5, pero esto puede realizarse con un solo ciclo como se muestra en el código 4.1.

```
1  // \alpha = a_p^T\cdot a_q, \beta = a_p^T\cdot a_p, \gamma = a_q^T\cdot a_q
2  double alpha = 0.0, beta = 0.0, gamma = 0.0;
3  double tmp_p, tmp_q;
4
5  for (size_t i = 0; i < m; ++i) {
6    tmp_p = A.elements[iteratorC(i, p_trans, lda)];
7    tmp_q = A.elements[iteratorC(i, q_trans, lda)];
8    alpha += tmp_p * tmp_q;
9    beta += tmp_p * tmp_p;
10   gamma += tmp_q * tmp_q;
11 }
```

Código 4.1: Cálculo de los elementos b_{pp}, b_{pq}, b_{qq}.

donde `alpha` $= b_{pq}$, `beta` $= b_{pp}$, `gamma` $= b_{qq}$.

Calculo de la rotación solo si b_{pq} en $B = A^T A$ es 0

Se sabe que la rotación de Jacobi introduce $b_{pq} = 0$. En la proposición de LAPACK en el algoritmo 4.5, no se verifica si $b_{pq} = 0$. Se propone que $b_{pq} = 0$, si $|b_{pq}| < 10^{-16}$ que se muestra en el código 4.2.

```
1  // Schur
2  double c_schur = 1.0, s_schur = 0.0, aqq = gamma, app = beta, apq = alpha;
3
4  // Si $b_{pq}=0$ no realizar la rotación
5  if (std::abs(apq) > tolerance) {
6    // Calcular c,s de la rotación de Jacobi
7    double tau = (aqq - app) / (2.0 * apq);
8    double t = 0.0;
9
10   if (tau >= 0) {
11     t = 1.0 / (tau + sqrt(1 + (tau * tau)));
12   } else {
13     t = 1.0 / (tau - sqrt(1 + (tau * tau)));
14   }
15
16   c_schur = 1.0 / sqrt(1 + (t * t));
17   s_schur = t * c_schur;
18 ...
```

Código 4.2: Verificar si $b_{pq} = 0$.

4.1.2. Número de operaciones en una iteración

Se desea calcular una cota inferior del número de operaciones y complejidad del algoritmo.

1. Las líneas $2, 3, 4$ del algoritmo 4.5 ejecuta $3n$ flam operaciones (adición y multiplicación flotante).

2. El ciclo for de la línea 11 se realiza m veces.

3. Las líneas $12, 13, 14$ del algoritmo 4.5 realiza 2 fladd operaciones (adición flotante) y 4 flmlt (multiplicación flotante). Para facilitar el cálculo de la cota inferior, decimos que hay 2 flam operaciones.

4. Por lo anterior, las líneas $11, 12, 13, 14$ efectúa $2m$ flam operaciones.

5. Análogamente las líneas $15, 16, 17, 18$ realiza $2n$ flam operaciones.

Se realizan $n(n-1)/2$ rotaciones, por lo que para una matriz cuadrada, el algoritmo realiza aproximadamente $(n(n-1)/2) \cdot (7n)$ flam operaciones, entonces el algoritmo es $O(n^3)$. Para mas información sobre flam, fladd y flmlt consultar [30].

4.2. Implementación paralela en memoria compartida

Se incorpora el ordenamiento paralelo del algoritmo 2.4 en el algoritmo 4.5 y usando hilos se paraleliza el algoritmo del método de Jacobi por un lado. Supongamos se paraleliza en l hilos y recordando que cada secuencia S_k puede realizarse paralelamente, el pseudo-algoritmo 4.6 ejemplifica como paralelizar el algoritmo del método de Jacobi por un lado con el uso de fork/join.

Algorithm 4.6 Una iteración del algoritmo de Jacobi por un lado paralelo

Require: $A \in \mathbb{R}^{m \times n}, V \in \mathbb{R}^{n \times n}$

1: $m\_ordering \leftarrow \lfloor (n+1)/2 \rfloor$
2: **for** $k = 1, ..., m\_ordering - 1$ **do**
3: $S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
4: **for** $q = m\_ordering - k + 1, ..., n - k$ **do**
5: **if** $m\_ordering - k + 1 \leq q \leq m\_ordering - 2k$ **then**
6: $p \leftarrow (2m\_ordering - 2k + 1) - q$
7: **else if** $2m\_ordering - 2k < q \leq 2m\_ordering - k - 1$ **then**
8: $p \leftarrow (4m\_ordering - 2k) - q$
9: **else if** $2m\_ordering - k - 1 < q$ **then**
10: $p \leftarrow n$
11: $S_k \cup \{(p, q)\}$ ▷ Agregar el par (p, q) a la secuencia paralela S_k
12: Particionar S_k equitativamente entre l subconjuntos, i.e. $S_{k_1}, ..., S_{k_l}$.
13: Fork l hilos.
14: **for** $t = 1, ..., l$ hilos **do**
15: **for** $(i, j) \in S_{k_t}$ **do**
16: $a \leftarrow \sum_{k=1}^{n} a_{ki}^2$ ▷ Calcular los elementos $\begin{bmatrix} a & c \\ c & b \end{bmatrix} \equiv (i, j) \in A^T A$
17: $b \leftarrow \sum_{k=1}^{n} a_{kj}^2$
18: $c \leftarrow \sum_{k=1}^{n} a_{ki} a_{kj}$
19:
20: $\zeta \leftarrow (b - 1)/(2c)$ ▷ Calcular la rotación de Jacobi para $\begin{bmatrix} a & c \\ c & b \end{bmatrix}$
21: $t \leftarrow sign(\zeta)/(|\zeta| + \sqrt{1 + \zeta^2})$
22: $cs \leftarrow 1/\sqrt{1 + t^2}$

23: $sn \leftarrow cs * t$
24:
25: **for** $k = 1, ..., m$ **do** ▷ Actualizar las columnas i y j de A
26: $tmp \leftarrow a_{ki}$
27: $a_{ki} \leftarrow cs * tmp - sn * a_{kj}$
28: $a_{kj} \leftarrow sn * tmp + cs * a_{kj}$
29: **for** $k = 1, ..., n$ **do** ▷ Actualizar las columnas i y j de V
30: $tmp \leftarrow v_{ki}$
31: $v_{ki} \leftarrow cs * tmp - sn * v_{kj}$
32: $v_{kj} \leftarrow sn * tmp + cs * v_{kj}$
33: Join los l hilos.
34: **for** $k = m\_ordering, ..., 2m\_ordering - 1$ **do**
35: $S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
36: **for** $q = 4m\_ordering - n - k, ..., 3m\_ordering - k - 1$ **do**
37: **if** $q < 2m\_ordering - k + 1$ **then**
38: $p \leftarrow n$
39: **else if** $2m\_ordering - k + 1 \leq q \leq 4m\_ordering - 2k - 1$ **then**
40: $p \leftarrow (4m\_ordering - 2k) - q$
41: **else if** $4m\_ordering - 2k - 1 < q$ **then**
42: $p \leftarrow (6m\_ordering - 2m\_ordering - 1) - q$
43: $S_k \cup \{(p, q)\}$ ▷ Agregar el par (p, q) a la secuencia paralela S_k
44: Particionar S_k equitativamente entre l subconjuntos, i.e. $S_{k_1}, ..., S_{k_l}$.
45: Fork l hilos.
46: **for** $t = 1, ..., l$ hilos **do**
47: **for** $(i, j) \in S_{k_t}$ **do**
48: ▷ Se repiten las líneas 16-32
49: Join los l hilos.
50: ▷ Los valores singulares son las normas de los vectores columnas en A
51: ▷ Los vectores singulares por la izquierda son las columnas normalizadas de A con los valores singulares

---

La Figura 4.1 ofrece una visualización de la partición de tareas en l hilos.

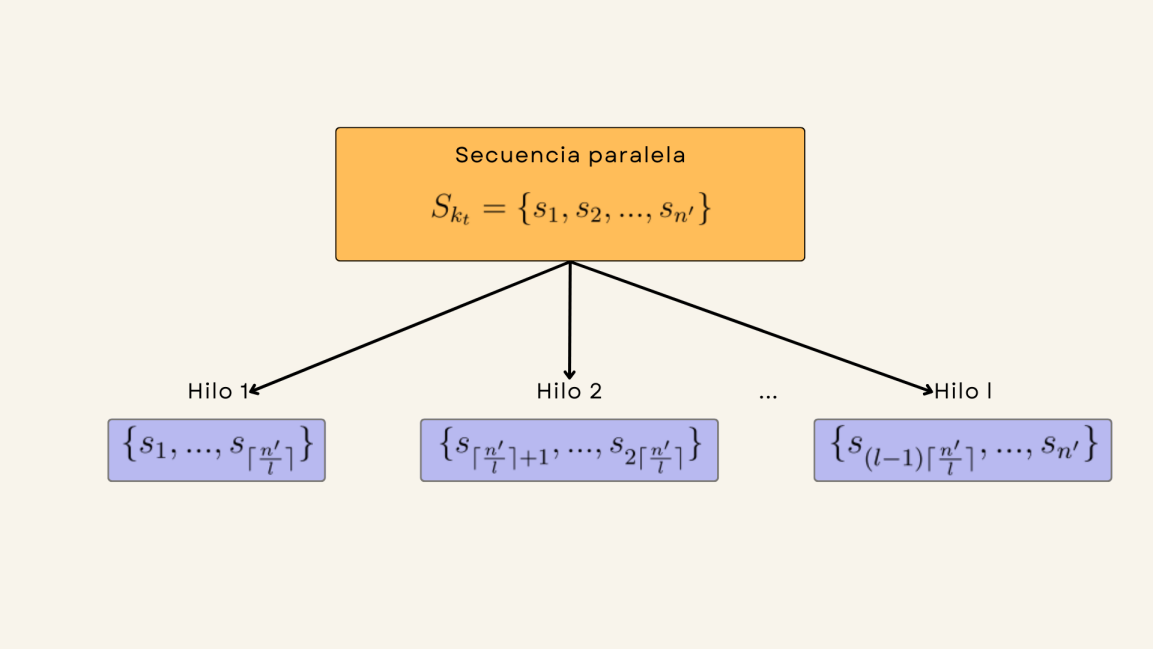

Figura 4.1: Visualización de la implementación paralela de la secuencia paralela S_{k_t} en l hilos.

4.2.1. Código fuente

En el código 4.3 se muestra el código que crea un equipo de hilos para un ordenamiento paralelo S_k con OpenMP, en la línea 8. Contrario al pseudo-algoritmo 4.6, usamos la directiva de paralelización de un ciclo for que se explico en el capítulo 3 en la sección 3.6.

```
1  // Parallel ordering S_k
2  for (size_t k = m_ordering; k < 2 * m_ordering; ++k) {
3    size_t p = 0;
4    size_t p_trans = 0;
5    size_t q_trans = 0;
6
7    // Create l threads teams with directive
8    #pragma omp parallel for private(p, p_trans, q_trans)
9    for (size_t q = (4 * m_ordering) - n - k; q < (3 * m_ordering) - k; ++q) {
10     if (q < (2 * m_ordering) - k + 1) {
11       p = n;
12     } else if ((2 * m_ordering) - k + 1 <= q && q <= (4 * m_ordering) - (2 * k)
           - 1) {
13       p = ((4 * m_ordering) - (2 * k)) - q;
14     } else if ((4 * m_ordering) - (2 * k) - 1 < q) {
15       p = ((6 * m_ordering) - (2 * k) - 1) - q;
16     }
17
18     // Translate to (0,0)
19     p_trans = p - 1;
20     q_trans = q - 1;
21 ...
```

Código 4.3: Ordenamiento paralelo S_k con paralelización en OpenMP.

4.3. Implementación paralela en memoria compartida heterogénea

Se decidió solo realizar las rotaciones de Jacobi con el dispositivo heterogéneo. Supongamos se paraleliza en l hilos del anfitrion, el pseudo-algoritmo 4.7 muestra como se incorpora la rotación en GPU donde $RotacionJacobi(a'_i, a'_j, n, cs, sn)$ es una función que realiza la rotación en GPU.

Algorithm 4.7 Una iteración del algoritmo de Jacobi por un lado paralelo-heterogéneo

Require: $A \in \mathbb{R}^{m\times n}, V \in \mathbb{R}^{n\times n}$

$m\_ordering \leftarrow \lfloor (n+1)/2 \rfloor$
for $k = 1, ..., m\_ordering - 1$ **do**
  $S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
  for $q = m\_ordering - k + 1, ..., n - k$ **do**
    if $m\_ordering - k + 1 \leq q \leq m\_ordering - 2k$ **then**
      $p \leftarrow (2m\_ordering - 2k + 1) - q$
    else if $2m\_ordering - 2k < q \leq 2m\_ordering - k - 1$ **then**
      $p \leftarrow (4m\_ordering - 2k) - q$
    else if $2m\_ordering - k - 1 < q$ **then**
      $p \leftarrow n$
    $S_k \cup \{(p, q)\}$ ▷ Agregar el par (p, q) a la secuencia paralela S_k
  Particionar S_k equitativamente entre l subconjuntos, i.e. $S_{k_1}, ..., S_{k_l}$.
  Fork l hilos.
  for $t = 1, ..., l$ hilos **do**
    for $(i, j) \in S_{k_t}$ **do**
      $a \leftarrow \sum_{k=1}^{n} a_{ki}^2$ ▷ Calcular los elementos $\begin{bmatrix} a & c \\ c & b \end{bmatrix} \equiv (i, j) \in A^T A$
      $b \leftarrow \sum_{k=1}^{n} a_{kj}^2$
      $c \leftarrow \sum_{k=1}^{n} a_{ki} a_{kj}$

      $\zeta \leftarrow (b - 1)/(2c)$ ▷ Calcular la rotación de Jacobi para $\begin{bmatrix} a & c \\ c & b \end{bmatrix}$
      $t \leftarrow sign(\zeta)/(|\zeta| + \sqrt{1 + \zeta^2})$
      $cs \leftarrow 1/\sqrt{1 + t^2}$
      $sn \leftarrow cs * t$

      Cargar los vectores columna i, j de A a la GPU en a'_i, a'_j.
      Cargar los vectores columna i, j de V a la GPU en v'_i, v'_j.

      $RotacionJacobi(a'_i, a'_j, n, cs, sn)$
      $RotacionJacobi(v'_i, v'_j, m, cs, sn)$

31: Descargar a'_i, a'_j de la GPU en los vectores columna i, j de A.
32: Descargar v'_i, v'_j de la GPU en los vectores columna i, j de V.
33: Join los l hilos.
34: **for** $k = m\_ordering, ..., 2m\_ordering - 1$ **do**
35: $S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
36: **for** $q = 4m\_ordering - n - k, ..., 3m\_ordering - k - 1$ **do**
37: **if** $q < 2m\_ordering - k + 1$ **then**
38: $p \leftarrow n$
39: **else if** $2m\_ordering - k + 1 \leq q \leq 4m\_ordering - 2k - 1$ **then**
40: $p \leftarrow (4m\_ordering - 2k) - q$
41: **else if** $4m\_ordering - 2k - 1 < q$ **then**
42: $p \leftarrow (6m\_ordering - 2m\_ordering - 1) - q$
43: $S_k \cup \{(p, q)\}$ ▷ Agregar el par (p, q) a la secuencia paralela S_k
44: Particionar S_k equitativamente entre l subconjuntos, i.e. $S_{k_1}, ..., S_{k_l}$.
45: Fork l hilos.
46: **for** $t = 1, ..., l$ hilos **do**
47: **for** $(i, j) \in S_{k_t}$ **do**
48: ▷ Se repiten las líneas 16-32
49: Join los l hilos.
50: ▷ Los valores singulares son las normas de los vectores columnas en A
51: ▷ Los vectores singulares por la izquierda son las columnas normalizadas de A con los valores singulares

La Figura 4.2 ofrece un esquema de la partición de tareas en l hilos y como usan la misma GPU para realizar la rotación.

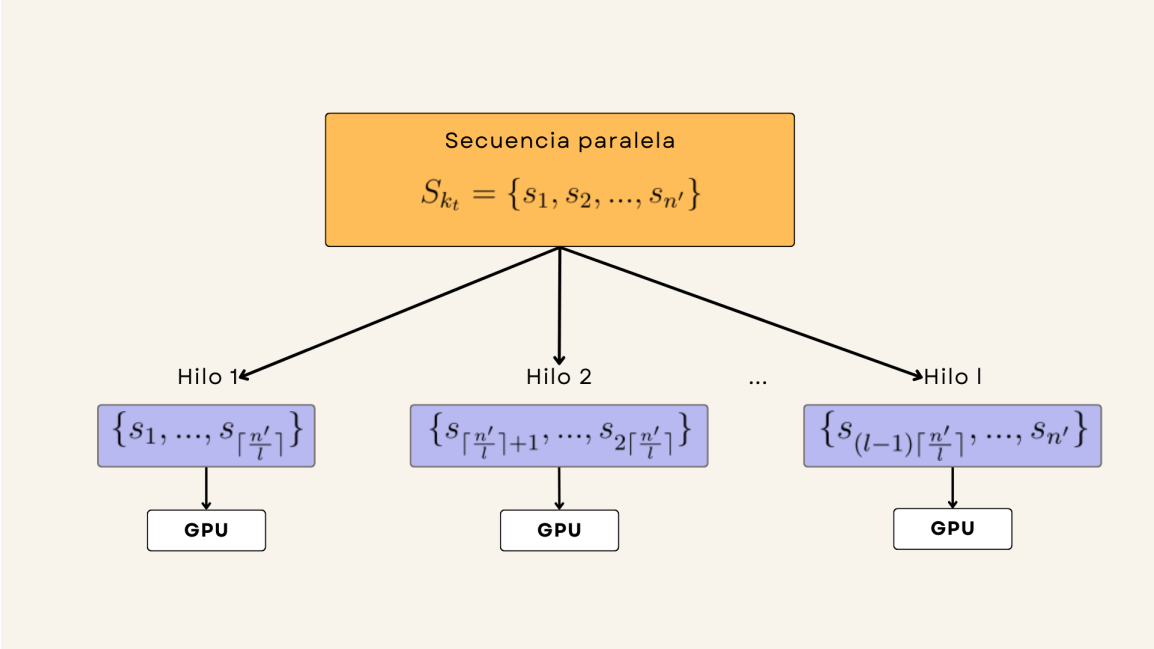

Figura 4.2: Visualización de implementación paralela-heterogénea de la secuencia paralela S_{k_t} en l hilos.

4.3.1. Código fuente

A continuación se muestran secciones del código fuente donde se describe como se crea, mueve y destruye la memoria en GPU junto con como se realizaron las ejecuciones de los kernel para realizar las rotaciones sobre A y V.

En el código 4.4 se muestra el código fuente que crea la memoria en el dispositivo (GPU) que se usará para mover memoria entre CPU (anfitrión) y dispositivo de las columnas de la matriz A y V para l hilos.

```
1  std::vector<double*> d_p_vectors(num_of_threads);
2  std::vector<double*> d_q_vectors(num_of_threads);
3  std::vector<double*> d_v_p_vectors(num_of_threads);
4  std::vector<double*> d_v_q_vectors(num_of_threads);
5
6  for(size_t i = 0; i < num_of_threads; i++){
7    cudaMalloc(&d_p_vectors[i], m * sizeof(double));
8    cudaMalloc(&d_q_vectors[i], m * sizeof(double));
9    cudaMalloc(&d_v_p_vectors[i], n * sizeof(double));
10   cudaMalloc(&d_v_q_vectors[i], n * sizeof(double));
11 }
```

Código 4.4: Creación de memoria en GPU para A y V.

Sin perdida de generalidad, se explica a profundidad el código 4.5 para la matriz A, `p_trans` y `q_trans` son los índices de las columnas sobre el cual se realizan las rotaciones, como la matriz A está ordenado por columna mayor (los elementos por cada vector columna están almacenados contiguamente) podemos cargar a memoria de GPU con un puntero al inicio de la columna e indicar que copie los m elementos siguientes como se indica en `cudaMemcpy(d_p_vectors[thread_id], (A.elements +`

`p_trans*lda), m * sizeof(double), cudaMemcpyHostToDevice);`, donde `thread_id` es el hilo OpenMP donde se realiza la rotación, `m` es la longitud de los vectores columna y `cudaMemcpyHostToDevice` indica que la copia es de anfitrión a dispositivo.

Despues, se realiza la rotación sobre las ambas columnas `p_trans` y `q_trans` de A `jacobi_rotation«<A_blocksPerGrid, threadsPerBlock»>(m, d_p_vectors[thread_id], d_q_vectors[thread_id], c_schur, s_schur);`, donde `m` es la longitud de los vectores columna, `thread_id` es el hilo OpenMP donde se realiza la rotación (sobre el mismo dispositivo), `c_schur` y `s_schur` son el seno y coseno obtenido para realizar la rotación; `A_blocksPerGrid` resulta de dividir la longitud m del vector columna entre el número de hilos `threadsPerBlock` para una sola dimensión como se explicó en el capítulo 3 en la sección 3.6.

Para finalizar y tomando en cuenta lo anterior solo transferimos de vuelta al anfitrión los vectores columna en el dispositivo como se muestra en `cudaMemcpy((A.elements + p_trans*lda), d_p_vectors[thread_id],m * sizeof(double), cudaMemcpyDeviceToHost);`, donde `cudaMemcpyDeviceToHost` indica que la copia es de dispositivo a anfitrión.

```
1  cudaMemcpy(d_p_vectors[thread_id], (A.elements + p_trans*lda), m * sizeof(
       double),
2                       cudaMemcpyHostToDevice);
3  cudaMemcpy(d_q_vectors[thread_id], (A.elements + q_trans*lda), m * sizeof(
       double),
4            cudaMemcpyHostToDevice);
5
6  jacobi_rotation<<<A_blocksPerGrid, threadsPerBlock>>>(m, d_p_vectors[thread_id
       ], d_q_vectors[thread_id], c_schur, s_schur);
7
8  cudaMemcpy((A.elements + p_trans*lda), d_p_vectors[thread_id],m * sizeof(double
       ),
9            cudaMemcpyDeviceToHost);
10 cudaMemcpy((A.elements + q_trans*lda), d_q_vectors[thread_id],m * sizeof(double
       ),
11           cudaMemcpyDeviceToHost);
12
13 cudaMemcpy(d_v_p_vectors[thread_id], (V.elements + p_trans*ldv), n * sizeof(
       double),
14           cudaMemcpyHostToDevice);
15 cudaMemcpy(d_v_q_vectors[thread_id], (V.elements + q_trans*ldv), n * sizeof(
       double),
16           cudaMemcpyHostToDevice);
17 jacobi_rotation<<<V_blocksPerGrid, threadsPerBlock>>>(n, d_v_p_vectors[
       thread_id], d_v_q_vectors[thread_id], c_schur, s_schur);
18
19 cudaMemcpy((V.elements + p_trans*ldv), d_v_p_vectors[thread_id],n * sizeof(
       double),
20           cudaMemcpyDeviceToHost);
21 cudaMemcpy((V.elements + q_trans*ldv), d_v_q_vectors[thread_id],n * sizeof(
       double),
22           cudaMemcpyDeviceToHost);
```

Código 4.5: Carga, descarga de columnas y rotación de Jacobi en GPU para el i-esimo hilo.

4.4. Esquema de distribución de matrices en MPI

Antes de profundizar en las implementaciones de memoria distribuida, en esta sección se explica y se ejemplifica como se particionó el ordenamiento paralelo S_k entre l número de nodos. En el código 4.3 se mostró como se realiza el ordenamiento paralelo S_k, este se puede repartir entre los diferentes nodos.

En general lo que se desea lograr es lo siguiente:

1. Particionar el ordenamiento paralelo S_k en l subconjuntos lo más equitativamente posible aunque esto es un problema computacional NP-complejo.

2. Usar estructuras de datos que permitan tener seguimiento de la distribución y la recolección de las particiones usando las interfaces de comunicación de MPI.

3. Realizar el menor número de envíos y recolecciones de las particiones.

El particionamiento del ordenamiento paralelo S_k en l subconjuntos $S_{k_1}, ..., S_{k_l}$ se guarda a través de la estructura de dato en la variable en la línea 2 del código 4.6.

La línea 2 del código 4.6, es una variable que guarda los pares de índices del subconjunto S_{k_i}, donde i es el i-ésimo nodo donde se está ejecutando este código.

```
1 // All points vector tuple
2 std::vector<std::tuple<size_t, size_t>> local_points;
3 ...
```

Código 4.6: Variable donde se guarda el i subconjunto S_{k_i} en el i-ésimo nodo.

El particionamiento S_k en l subconjuntos se realiza con el particionamiento de índices que realiza OpenMP al paralelizar un ciclo for. En la línea 16 del código 4.7 se muestra como se guarda la variable mencionada en el código 4.6.

```
1  #pragma omp parallel for num_threads(size) private(p, p_trans, q_trans)
2  for (size_t q = m_ordering - k + 1; q <= n - k; ++q) {
3    if (m_ordering - k + 1 <= q && q <= (2 * m_ordering) - (2 * k)) {
4      p = ((2 * m_ordering) - (2 * k) + 1) - q;
5    } else if ((2 * m_ordering) - (2 * k) < q && q <= (2 * m_ordering) - k - 1) {
6      p = ((4 * m_ordering) - (2 * k)) - q;
7    } else if ((2 * m_ordering) - k - 1 < q) {
8      p = n;
9    }
10
11   // Translate to (0,0)
12   p_trans = p - 1;
13   q_trans = q - 1;
14
15   if(rank == omp_get_thread_num()){
16     local_points.emplace_back(p_trans,q_trans);
17     ...
18   }
19 }
```

Código 4.7: Fragmento de código fuente donde se muestra como se guardan los pares de índices en las variables mencionadas en el código 4.6.

También se desea tener un conjunto ordenado de los índices, estas estructuras de datos se generan en las líneas del código 4.8.

La línea 2 del código 4.8, es una variable que guarda los índices ordenados del subconjunto S_{k_i}, donde i es el i-ésimo nodo donde se está ejecutando este código.

La línea 5 del código 4.8, es una variable que guarda los índices ordenados de todos los subconjuntos $S_{k_1}, ..., S_{k_l}$ en el nodo raíz o el nodo 1 que distribuirá datos a todos los nodos.

```
1 // Ordered set of coordinates
2 std::set<size_t> local_set_columns;
3 ...
4 // ordered set of points by rank. To use for data distribution and extraction.
5 std::vector<std::set<size_t>> root_set_columns_by_rank(size);
```

Código 4.8: Variables donde se guardan los l subconjuntos de índices ordenados $S_{k_1}, ..., S_{k_l}$ con $l = size$.

Estos subconjuntos ordenados de índices se realizan con el particionamiento de índices que realiza OpenMP al paralelizar un ciclo for. En las líneas $17-18$ y $23-24$ del código 4.7 se muestra como se guardan las variables mencionadas en el código 4.6.

```
1  #pragma omp parallel for num_threads(size) private(p, p_trans, q_trans)
2  for (size_t q = m_ordering - k + 1; q <= n - k; ++q) {
3    if (m_ordering - k + 1 <= q && q <= (2 * m_ordering) - (2 * k)) {
4      p = ((2 * m_ordering) - (2 * k) + 1) - q;
5    } else if ((2 * m_ordering) - (2 * k) < q && q <= (2 * m_ordering) - k - 1) {
6      p = ((4 * m_ordering) - (2 * k)) - q;
7    } else if ((2 * m_ordering) - k - 1 < q) {
8      p = n;
9    }
10
11   // Translate to (0,0)
12   p_trans = p - 1;
13   q_trans = q - 1;
14
15   if(rank == ROOT_RANK){
16     ...
17     root_set_columns_by_rank[omp_get_thread_num()].insert(p_trans);
18     root_set_columns_by_rank[omp_get_thread_num()].insert(q_trans);
19   }
20
21   if(rank == omp_get_thread_num()){
22     ...
23     local_set_columns.insert(p_trans);
24     local_set_columns.insert(q_trans);
25   }
26 }
```

Código 4.9: Fragmento de código fuente donde se muestra como se guardan los índices ordenados en las variables mencionadas en el código 4.8.

4.4.1. Ejemplo de distribución de matriz

Supongamos una matriz simétrica $B \in \mathbb{R}^{10\times10}$ definida por A^TA con $A \in \mathbb{R}^{n\times n}$ una matriz arbitraria, en la Figura 4.3 se muestra la representación de su ordenamiento paralelo.

Tomando la secuencia paralela $S_1 = \{(4,5),(3,6),(2,7),(1,8),(9,10)\}$, se particionan las columnas de A a partir de B para 3 nodos, es decir para 1 nodo raíz y 2 nodos esclavos como $S_{1_1} = \{(4,5)\}, S_{1_2} = \{(3,6),(2,7)\}, S_{1_2} = \{(1,8),(9,10)\}$.

x	4	8	3	7	2	6	1	5	9
	x	3	7	2	6	1	5	9	8
		x	2	6	1	5	9	4	7
			x	1	5	9	4	8	6
				x	9	4	8	3	5
					x	8	3	7	4
						x	7	2	3
							x	6	2
								x	1
									x

Figura 4.3: Representación del ordenamiento paralelo de la matriz A.

En la ecuación 4.1 se visualiza los contenidos de la variable root_set_columns_by_rank, en la ecuación 4.2, 4.3 y 4.4 se visualizan los contenidos de las variables local_set_columns en los nodos 1, 2 y 3, respectivamente.

$$root\_set\_columns\_by\_rank = \{\{4,5\},\{2,3,6,7\},\{1,8,9,10\}\} \tag{4.1}$$
$$local\_set\_columns = \{4,5\} \tag{4.2}$$
$$local\_set\_columns = \{2,3,6,7\} \tag{4.3}$$
$$local\_set\_columns = \{1,8,9,10\} \tag{4.4}$$

Con las variables root_set_columns_by_rank y local_set_columns se construyen las matrices A_1, A_2, A_3 sobre las cuales operarán cada nodo. En la Figura 4.4 se muestra la repartición de columnas entre nodos con los que se definen las matrices A_1, A_2, A_3.

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}	a_{110}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}	a_{210}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figura 4.4: Las columnas pertenecientes al nodo 1 se muestran en color azul, las columnas pertenecientes al nodo 2 en color naranja y las columnas pertenecientes al nodo 3 en color verde

Las matrices A_1, A_2, A_3 quedan como en las ecuaciones 4.5, 4.6 y 4.7 respectivamente.

$$A_1 = \begin{bmatrix} a_{14} & a_{15} \\ a_{24} & a_{25} \\ \vdots & \vdots \end{bmatrix} \tag{4.5}$$

$$A_2 = \begin{bmatrix} a_{12} & a_{13} & a_{16} & a_{17} \\ a_{22} & a_{23} & a_{26} & a_{27} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \tag{4.6}$$

$$A_2 = \begin{bmatrix} a_{11} & a_{18} & a_{19} & a_{110} \\ a_{21} & a_{28} & a_{29} & a_{210} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \tag{4.7}$$

En el código 4.10 se detalla como se crean estas matrices, para después distribuirlas en los nodos excepto el nodo raíz.

```
1  std::vector<double> tmp_matrix;
2   for(auto column_index: root_set_columns_vector_by_rank[index_rank]){
3     tmp_matrix.insert(tmp_matrix.end(), A.elements + m*column_index, A.elements
          + (m*(column_index + 1)));
4   }
5   MatrixMPI A_rank(tmp_matrix, m, root_set_columns_by_rank[index_rank].size());
```

Código 4.10: Fragmento de código fuente donde se realiza la creación de las matrices a distribuir.

Para realizar cálculos sobre estas matrices locales, se necesita un diccionario que mapee los índices encontrados en local_points a los índices de las matrices locales. En el código 4.11 se describe como crear la estructura de datos con variable column_index_to_local_column_index que hace este mapeo.

```
1  // convert local set to vector
2  local_set_to_vector = std::vector<size_t>(local_set_columns.begin(),
       local_set_columns.end());
3  // map coordinates to local column indices
4  size_t local_set_to_vector_size = local_set_to_vector.size();
5  for(auto i = 0; i < local_set_to_vector_size; ++i){
6    column_index_to_local_column_index[local_set_to_vector[i]] = i;
7  }
```

Código 4.11: Fragmento de código fuente donde se realiza el mapeo de índices en local_points a índices de las matrices locales.

En las ecuaciones 4.8 y 4.9 se visualizan los contenidos de las variables column _index_to_local_column_index en los nodos 2 y 3, respectivamente.

$$column\_index\_to\_local\_column\_index = \{\{2:1\},\{3:2\},\{6:3\},\{7:4\}\} \tag{4.8}$$

$$column\_index\_to\_local\_column\_index = \{\{1:1\},\{8:2\},\{9:3\},\{10:4\}\} \tag{4.9}$$

Realizados los cálculos sobre las matrices A_1, A_2, A_3, los nodos 2 y 3 regresan sus respectivas matrices al nodo 1. Y se sobreescribe la matriz A en el nodo 1. En el

código 4.12 se describe como se realiza el escritura de cada matriz recibida A_gather sobre A.

```
1  // Create local matrix
2  MatrixMPI A_gather(m, root_set_columns_by_rank[index_rank].size());
3
4  ...
5
6  size_t root_set_columns_vector_by_rank_for_index_rank_size =
       root_set_columns_vector_by_rank[index_rank].size();
7  #pragma omp parallel for
8  for(size_t index_column = 0; index_column <
       root_set_columns_vector_by_rank_for_index_rank_size; ++index_column){
9    for(size_t index_row = 0; index_row < m; ++index_row){
10     A.elements[iteratorC(index_row, root_set_columns_vector_by_rank[index_rank
           ][index_column], lda)] = A_gather.elements[iteratorC(index_row,
           index_column, lda)];
11   }
12 }
```

Código 4.12: Fragmento de código fuente donde se realiza la escritura de las matrices recibidas A_gather sobre A.

4.5. Implementación híbrida paralela

El pseudo-algoritmo 4.8 realiza la descomposición SVD distribuyendo el ordenamiento paralelo S_k en r nodos, donde cada nodo paraleliza en l hilos con el uso de fork/join.

Algorithm 4.8 Una iteración del algoritmo de Jacobi por un lado híbrido paralelo

Require: $A \in \mathbb{R}^{m \times n}, V \in \mathbb{R}^{n \times n}$

1: $m\_ordering \leftarrow \lfloor (n+1)/2 \rfloor$
2: **for** $k = 1, ..., m\_ordering - 1$ **do**
3: $S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
4: **for** $q = m\_ordering - k + 1, ..., n - k$ **do**
5: **if** $m\_ordering - k + 1 \leq q \leq m\_ordering - 2k$ **then**
6: $p \leftarrow (2m\_ordering - 2k + 1) - q$
7: **else if** $2m\_ordering - 2k < q \leq 2m\_ordering - k - 1$ **then**
8: $p \leftarrow (4m\_ordering - 2k) - q$
9: **else if** $2m\_ordering - k - 1 < q$ **then**
10: $p \leftarrow n$
11: $S_k \cup \{(p, q)\}$ ▷ Agregar el par (p, q) a la secuencia paralela S_k
12: Particionar S_k equitativamente entre r subconjuntos, i.e. $S_{k_1}, ..., S_{k_r}$.
13: Crear matrices $A_1, ..., A_r$ de los vectores columna de A de los índices de los puntos en $S_{k_1}, ..., S_{k_r}$, respectivamente.
14: Distribuir las matrices $A_1, ..., A_r$ con sus respectivos conjuntos de puntos $S_{k_1}, ..., S_{k_r}$ en los r nodos.
15: **for** $t = 1, ..., r$ nodos **do**
16: Particionar S_{k_t} equitativamente entre l subconjuntos, i.e. $S_{k_{t1}}, ..., S_{k_{tl}}$.
17: Fork l hilos.

18: **for** $s = 1, ..., l$ hilos **do**
19: **for** $(i, j) \in k_{ts}$ **do**
20: $a \leftarrow \sum_{k=1}^{n} a_{ki}^2$ ▷ Calcular los elementos $\begin{bmatrix} a & c \\ c & b \end{bmatrix} \equiv (i, j) \in A^T A$
21: $b \leftarrow \sum_{k=1}^{n} a_{kj}^2$
22: $c \leftarrow \sum_{k=1}^{n} a_{ki} a_{kj}$
23:
24: $\zeta \leftarrow (b - 1)/(2c)$ ▷ Calcular la rotación de Jacobi
25: $t \leftarrow sign(\zeta)/(|\zeta| + \sqrt{1 + \zeta^2})$
26: $cs \leftarrow 1/\sqrt{1 + t^2}$
27: $sn \leftarrow cs * t$
28:
29: **for** $k = 1, ..., m$ **do** ▷ Actualizar las columnas i y j de A
30: $tmp \leftarrow a_{ki}$
31: $a_{ki} \leftarrow cs * tmp - sn * a_{kj}$
32: $a_{kj} \leftarrow sn * tmp + cs * a_{kj}$
33: **for** $k = 1, ..., n$ **do** ▷ Actualizar las columnas i y j de V
34: $tmp \leftarrow v_{ki}$
35: $v_{ki} \leftarrow cs * tmp - sn * v_{kj}$
36: $v_{kj} \leftarrow sn * tmp + cs * v_{kj}$
37: Join los l hilos.
38: **for** $k = m\_ordering, ..., 2m\_ordering - 1$ **do**
39: $S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
40: **for** $q = 4m\_ordering - n - k, ..., 3m\_ordering - k - 1$ **do**
41: **if** $q < 2m\_ordering - k + 1$ **then**
42: $p \leftarrow n$
43: **else if** $2m\_ordering - k + 1 \leq q \leq 4m\_ordering - 2k - 1$ **then**
44: $p \leftarrow (4m\_ordering - 2k) - q$
45: **else if** $4m\_ordering - 2k - 1 < q$ **then**
46: $p \leftarrow (6m\_ordering - 2m\_ordering - 1) - q$
47: $S_k \cup \{(p, q)\}$ ▷ Agregar el par (p, q) a la secuencia paralela S_k
48: ▷ Se repiten las líneas 12-37
49: ▷ Los valores singulares son las normas de los vectores columnas en A
50: ▷ Los vectores singulares por la izquierda son las columnas normalizadas de A con los valores singulares

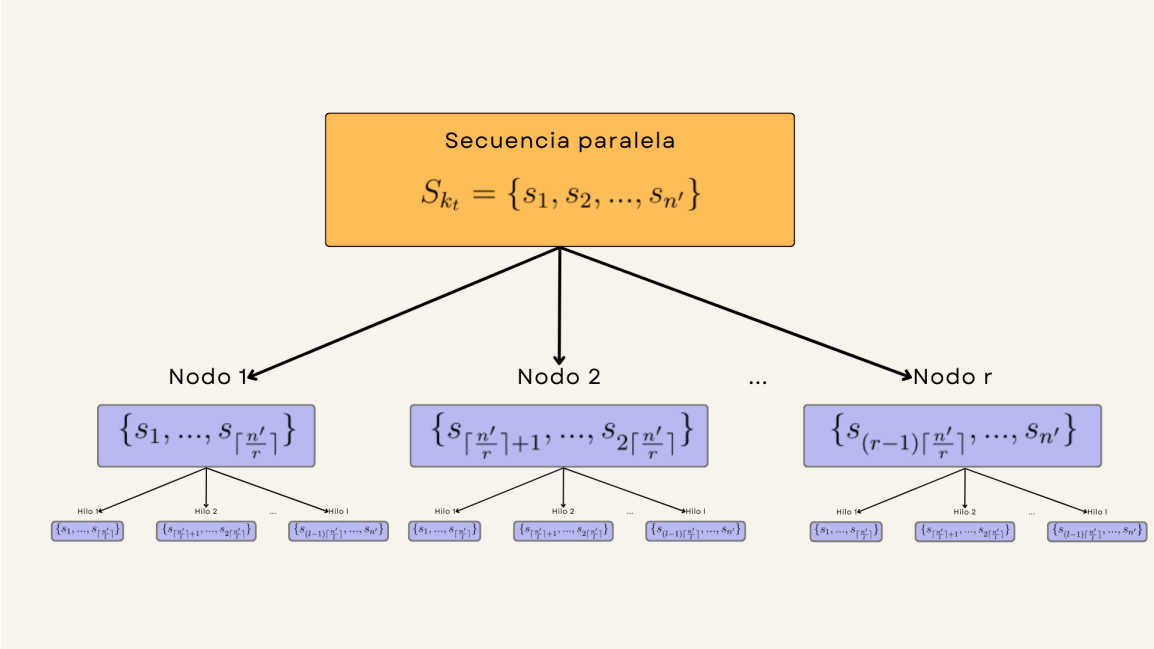

Figura 4.5: Visualización de implementación híbrida-paralela de la secuencia paralela S_{k_t} en r nodos con l hilos.

4.5.1. Implementación

La distribución de las matrices con MPI se realizó como se indica en la sección anterior 4.4, donde se presentó el esquema de distribución y partición de la matriz. Además, se paralelizó las rotaciones sobre las particiones en cada nodo como se explica en la sección 4.2, siguiendo la paralelización del ordenamiento paralelo con OpenMP. Así, se implementa el algoritmo bajo el modelo híbrido paralelo mezclando las herramientas MPI y OpenMP.

4.6. Implementación paralela híbrida heterogénea

El pseudo-algoritmo 4.9 realiza la descomposición SVD distribuyendo el ordenamiento paralelo S_k en r nodos, donde cada nodo paraleliza en l hilos con el uso de fork/join y se usa el dispositivo para realizar las rotaciones de Jacobi.

Algorithm 4.9 Una iteración del algoritmo de Jacobi por un lado híbrido paralelo heterogéneo

Require: $A \in \mathbb{R}^{m \times n}, V \in \mathbb{R}^{n \times n}$
1: $m\_ordering \leftarrow \lfloor (n+1)/2 \rfloor$
2: **for** $k = 1, ..., m\_ordering - 1$ **do**
3: $S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
4: **for** $q = m\_ordering - k + 1, ..., n - k$ **do**
5: **if** $m\_ordering - k + 1 \leq q \leq m\_ordering - 2k$ **then**

$p \leftarrow (2m\_ordering - 2k + 1) - q$
else if $2m\_ordering - 2k < q \leq 2m\_ordering - k - 1$ **then**
$p \leftarrow (4m\_ordering - 2k) - q$
else if $2m\_ordering - k - 1 < q$ **then**
$p \leftarrow n$
$S_k \cup \{(p, q)\}$ ▷ Agregar el par (p, q) a la secuencia paralela S_k
Particionar S_k equitativamente entre r subconjuntos, i.e. $S_{k_1}, ..., S_{k_r}$.
Crear matrices $A_1, ..., A_r$ de los vectores columna de A de los índices de los puntos en $S_{k_1}, ..., S_{k_r}$, respectivamente.
Distribuir las matrices $A_1, ..., A_r$ con sus respectivos conjuntos de puntos $S_{k_1}, ..., S_{k_r}$ en los r nodos.
for $t = 1, ..., r$ nodos **do**
Particionar S_{k_t} equitativamente entre l subconjuntos, i.e. $S_{k_{t1}}, ..., S_{k_{tl}}$.
Fork l hilos.
for $s = 1, ..., l$ hilos **do**
for $(i, j) \in k_{ts}$ **do**
$a \leftarrow \sum_{k=1}^{n} a_{ki}^2$ ▷ Calcular los elementos $\begin{bmatrix} a & c \\ c & b \end{bmatrix} \equiv (i, j) \in A^T A$
$b \leftarrow \sum_{k=1}^{n} a_{kj}^2$
$c \leftarrow \sum_{k=1}^{n} a_{ki} a_{kj}$

$\zeta \leftarrow (b - 1)/(2c)$ ▷ Calcular la rotación de Jacobi
$t \leftarrow sign(\zeta)/(|\zeta| + \sqrt{1 + \zeta^2})$
$cs \leftarrow 1/\sqrt{1 + t^2}$
$sn \leftarrow cs * t$

Cargar los vectores columna i, j de A a la GPU en a'_i, a'_j.
Cargar los vectores columna i, j de V a la GPU en v'_i, v'_j.

$RotacionJacobi(a'_i, a'_j, n, cs, sn)$
$RotacionJacobi(v'_i, v'_j, m, cs, sn)$

Descargar a'_i, a'_j de la GPU en los vectores columna i, j de A.
Descargar v'_i, v'_j de la GPU en los vectores columna i, j de V.
Join los l hilos.
for $k = m\_ordering, ..., 2m\_ordering - 1$ **do**
$S_k \leftarrow \emptyset$ ▷ Crear el conjunto vacio S_k
for $q = 4m\_ordering - n - k, ..., 3m\_ordering - k - 1$ **do**
if $q < 2m\_ordering - k + 1$ **then**
$p \leftarrow n$
else if $2m\_ordering - k + 1 \leq q \leq 4m\_ordering - 2k - 1$ **then**
$p \leftarrow (4m\_ordering - 2k) - q$
else if $4m\_ordering - 2k - 1 < q$ **then**

46: $p \leftarrow (6m\_ordering - 2m\_ordering - 1) - q$
47: $S_k \cup \{(p, q)\}$ ▷ Agregar el par (p, q) a la secuencia paralela S_k
48: ▷ Se repiten las líneas 12-37
49: ▷ Los valores singulares son las normas de los vectores columnas en A
50: ▷ Los vectores singulares por la izquierda son las columnas normalizadas de A con los valores singulares

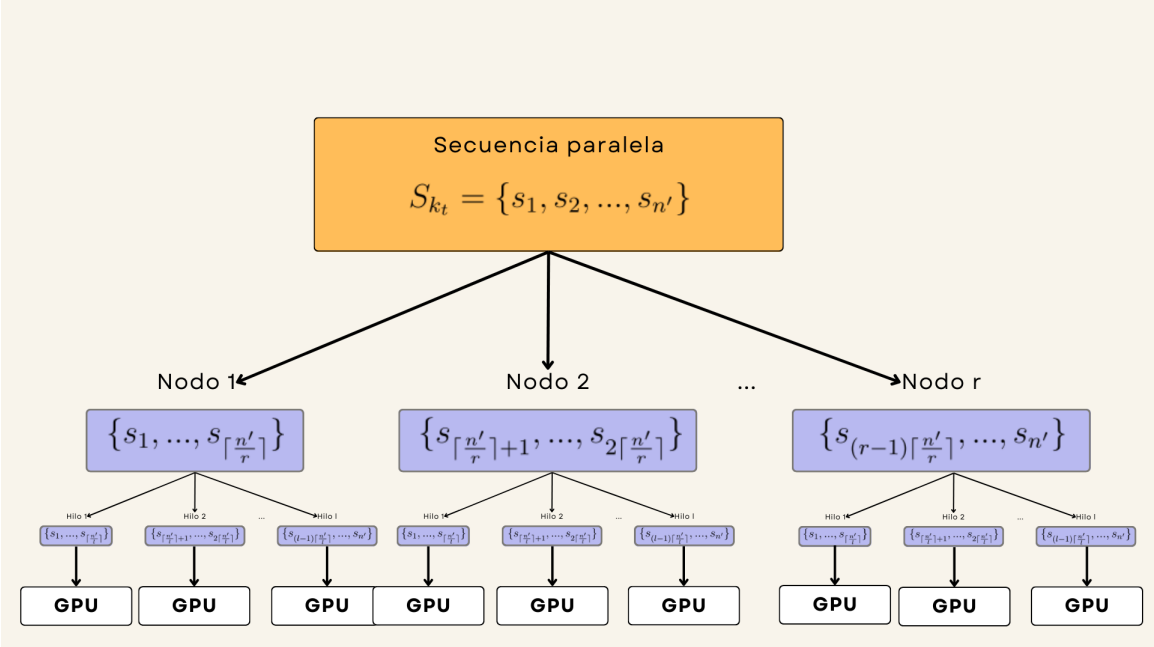

Figura 4.6: Visualización de implementación híbrida paralela heterogénea de la secuencia paralela S_{k_t} en r nodos con l hilos y un GPU por nodo.

4.6.1. Implementación

La distribución de las matrices con MPI se realizó como se indica en la sección anterior 4.4 donde se describió el esquema de distribución y partición de las columnas de un ordenamiento paralelo, igualmente se realizaron las rotaciones de manera paralela heterogénea como se describió en la sección 4.3, donde se describe como paraleliza un ordenamiento paralelo y cada hilo realiza rotaciones de Jacobi en la GPU.

Capítulo 5

Resultados

En este capítulo se presentan los resultados experimentales de las implementaciones paralelas. Se probaron las siguientes implementaciones:

- Paralela en memoria compartida (OpenMP).
- Paralela heterogénea (OpenMP+CUDA).
- Híbrida paralela (MPI+OpenMP).
- Paralela híbrida heterogénea (MPI+OpenMP+CUDA).

Algunas implementaciones realizadas se comparan contra la implementación del método de Jacobi por un lado de `LAPACKE_dgesvj` de la biblioteca LAPACK de Intel MKL. Las pruebas se realizaron en dos diferentes equipos de cómputo, en uno personal y en la supercomputadora Leo Atrox. Se muestran gráficas comparativas del tiempo de ejecución de las diferentes implementaciones y el error absoluto entre la matriz original y la descomposición para múltiples dimensiones de matrices cuadradas así como una descripción de estas figuras. El capítulo se encuentra separado en tres secciones, la primera describe las características de las pruebas que se realizaron y las dos secciones siguiente se separan por equipo.

5.1. Características de las pruebas

Todas las pruebas se realizaron sobre matrices triangulares superiores y cuadradas. Sus elementos fueron seleccionados aleatoriamente con valores entre 0.0 y 1.0 como se muestra en el código 5.1.

```
1 // Create R matrix
2 std::random_device random_device;
3 std::mt19937 mt_19937(random_device());
4 std::default_random_engine e(seed);
5 std::uniform_real_distribution<double> uniform_dist(0.0, 1.0);
6 for (size_t indexRow = 0; indexRow < std::min<size_t>(A_height, A_width); ++
      indexRow) {
```

```
7     for (size_t indexCol = indexRow; indexCol < std::min<size_t>(A_height,
          A_width); ++indexCol) {
8       double value = uniform_dist(mt_19937);
9       A.elements[iteratorC(indexRow, indexCol, A_height)] = value;
10      A_copy.elements[iteratorC(indexRow, indexCol, A_height)] = value;
11    }
12  }
```

Código 5.1: Se llena la triangular superior con valores aleatorios entre 0 y 1.

Se midió el tiempo de ejecución del algoritmo de una iteración del método de Jacobi por un lado, en el código 5.2 se muestra como se midió el tiempo de ejecución para la implementación paralela en memoria compartida, esta medición se realiza de la misma manera en todas las implementaciones.

```
1  // Calculate SVD decomposition
2  double ti = omp_get_wtime();
3  Thesis::omp_dgesvd(Thesis::AllVec,
4                     Thesis::AllVec,
5                     A.height,
6                     A.width,
7                     Thesis::COL_MAJOR,
8                     A,
9                     A_height,
10                    s,
11                    V,
12                    A_width);
13 double tf = omp_get_wtime();
14 double time = tf - ti;
15 time_avg += time;
16 file_output << "SVD OMP time with U,V calculation: " << time << "\n";
17 std::cout << "SVD OMP time with U,V calculation: " << time << "\n";
```

Código 5.2: Medición de tiempo de ejecución de la implementación paralela.

Se midió el error absoluto entre A y su descomposición SVD $U\Sigma V^T$ generado por el algoritmo de una iteración del método de Jacobi por un lado, es decir $||A - U\Sigma V^T||_F$. En el código 5.3 se muestra de ejemplo como se calcula para la implementación paralela en memoria compartida, esta medición se realiza de la misma manera en todas las implementaciones.

```
1  // A - A*
2  #pragma omp parallel for
3  for (size_t indexRow = 0; indexRow < A_height; ++indexRow) {
4    for (size_t indexCol = 0; indexCol < A_width; ++indexCol) {
5      double value = 0.0;
6      for (size_t k_dot = 0; k_dot < A_width; ++k_dot) {
7        value += A.elements[iteratorC(indexRow, k_dot, A_height)] * s.elements[
             k_dot]
8            * V.elements[iteratorC(indexCol, k_dot, A_height)];
9      }
10     A_copy.elements[iteratorC(indexRow, indexCol, A_height)] -= value;
11   }
12 }
13 // Calculate frobenius norm
14 double frobenius_norm = 0.0;
15 #pragma omp parallel for reduction(+:frobenius_norm)
16 for (size_t indexRow = 0; indexRow < A_height; ++indexRow) {
17   for (size_t indexCol = 0; indexCol < A_width; ++indexCol) {
18     double value = A_copy.elements[iteratorC(indexRow, indexCol, A_height)];
19     frobenius_norm += value*value;
20   }
21 }
```

```
22  file_output << "||A-USVt||_F: " << sqrt(frobenius_norm) << "\n";
23  std::cout << "||A-USVt||_F: " << sqrt(frobenius_norm) << "\n";
```

Código 5.3: Norma de frobenius del error absoluto $||A - U\Sigma V^T||_F$ en la implementación híbrida paralela.

5.2. Resultados en equipo personal

En este equipo se realizaron las pruebas de `LAPACKE_dgesvj` de la biblioteca LAPACK de Intel MKL. Además se probaron dos implementaciones OpenMP y OpenMP+CUDA.

5.2.1. Características del equipo

El equipo personal tiene un procesador AMD Ryzen 3700X de 8 núcleos y 16 hilos con 16 GB de memoria RAM, una tarjeta de video NVIDIA GTX 1660 con capacidad de aceleración CUDA. Todas las pruebas se realizaron con 16 hilos.

5.2.2. Tiempos de ejecución

Los tiempos de ejecución era una métrica que se tenía de objetivo minimizar. Las pruebas fueron realizadas todas con las características descritas con anterioridad. En general, se muestra en la Figura 5.1 que la aceleración de la rotación de Jacobi en una GPU minimiza el tiempo de ejecución de la implementación acelerando hasta casi 5 veces comparado con la implementación paralela (OMP).

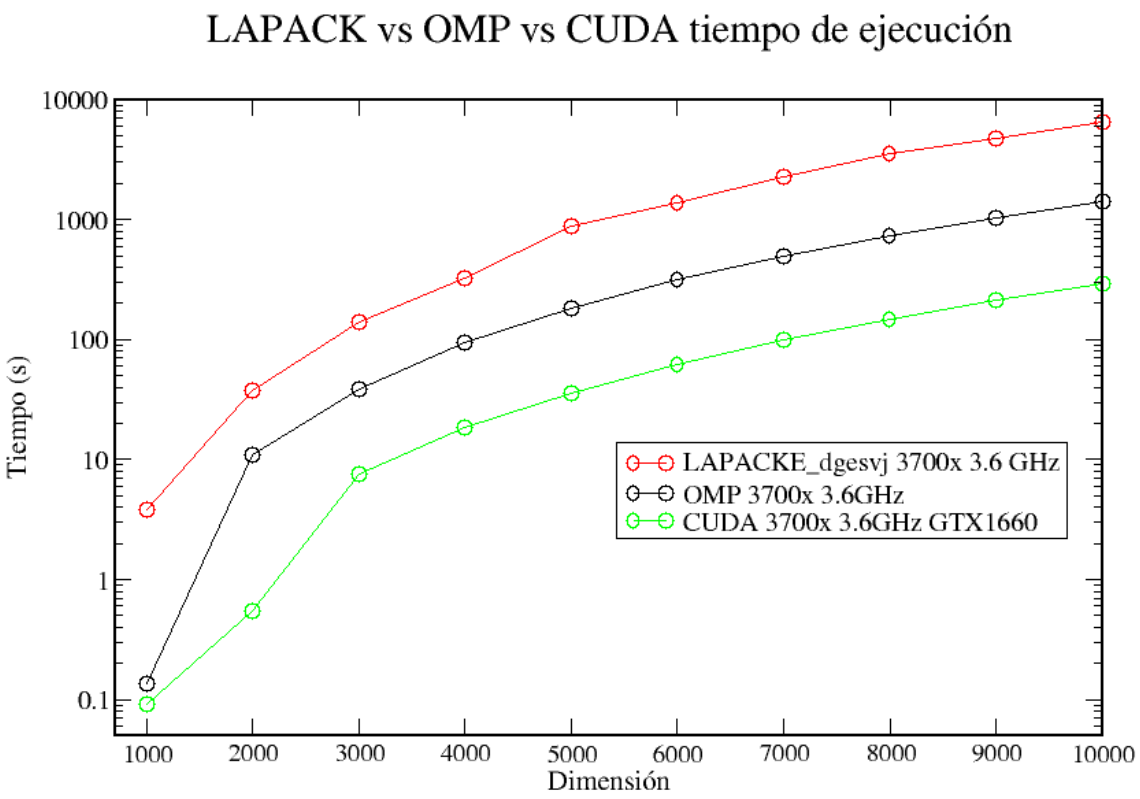

Figura 5.1: Tiempo de ejecución en segundos de las implementaciones paralela (con leyenda OMP), paralela heterogénea (con leyenda CUDA) y la función `LAPACKE_dgesvj` de LAPACK contra dimensiones de 1000 a 10000 de matrices cuadradas.

Se observa en la Figura 5.1 que la función `LAPACKE_dgesvj` (que es la descomposición en valores singulares usando el método de Jacobi por un lado) de LAPACK es la más lenta, esto se debe a que la función implementa el ordenamiento de los valores singulares y vectores singulares, mientras nosotros solo implementamos una iteración del método de Jacobi especificado en [35]. La versión paralela heterogénea (con leyenda CUDA) que acelera la rotación de Jacobi es la más rápida acelerando 4.9 veces el tiempo de ejecución contra la implementación paralela en memoria compartida (con leyenda OMP).

5.2.3. Error absoluto

El error absoluto es una métrica importante en los métodos numéricos, aunque no fue un objetivo en este trabajo. Sin embargo, se muestra en la Figura 5.2 que la aceleración de la rotación de Jacobi en una GPU minimizó el error absoluto, siendo la implementación paralela heterogénea (con leyenda CUDA) con menor error en las pruebas en el equipo personal.

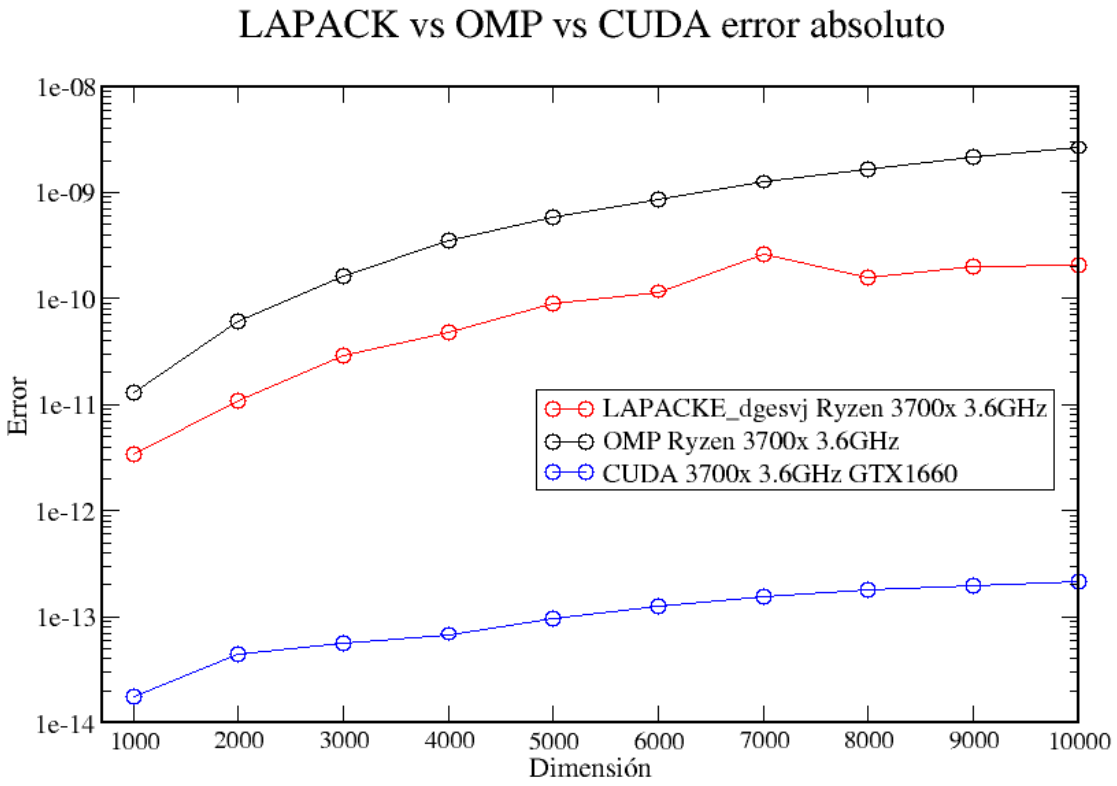

Figura 5.2: Error absoluto de las implementaciones paralela en memoria compartida (OMP), paralela heterogénea en memoria compartida (CUDA) y la función `LAPACKE_dgesvj` de LAPACK contra dimensiones de 1000 a 10000 de matrices cuadradas.

Se observa en la Figura 5.2 que la implementación paralela en memoria compartida (con leyenda OMP) es la que tiene el mayor error absoluto. La versión paralela heterogénea (con leyenda CUDA) que acelera la rotación de Jacobi tiene el menor error absoluto con orden de magnitud de 10^{-14}.

5.3. Resultados en la supercomputadora Leo Atrox de la Universidad de Guadalajara

Las pruebas realizadas en la supercomputadora Leo Atrox de la Universidad de Guadalajara fueron las implementaciones OpenMP, OpenMP+CUDA, MPI+OpenMP y MPI+OpenMP+CUDA.

5.3.1. Características de los nodos

Todos los nodos incluidos los que tienen GPU tienen un procesador Intel Xeon Gold 6140 con 36 núcleos, 36 hilos repartidos en dos sockets, con una red de interconexión Omnipath de Intel. Los dos nodos que cuentan con acelerador GPU, tienen una GPU NVIDIA Tesla P100. Todas la pruebas se realizaron con 36 hilos.

En las pruebas de la versión heterogénea se corrieron para las dimensiones de matrices cuadradas 11000, 21000 y 31000 porque se hizo una corrida para una matriz de dimensión 1000 para tener cache en la GPU.

5.3.2. Tiempos de ejecución

En la Figura 5.3 se ofrece una gráfica comparativa de tiempos de ejecución entre las implementaciones paralela en memoria compartida (con leyenda OMP), paralela heterogénea en memoria compartida (con leyenda CUDA), híbrida paralela en memoria distribuida y compartida (con leyenda MPI+OMP) e paralela híbrida heterogénea (con leyenda MPI+OMP+CUDA) para diferentes dimensiones de matrices cuadradas.

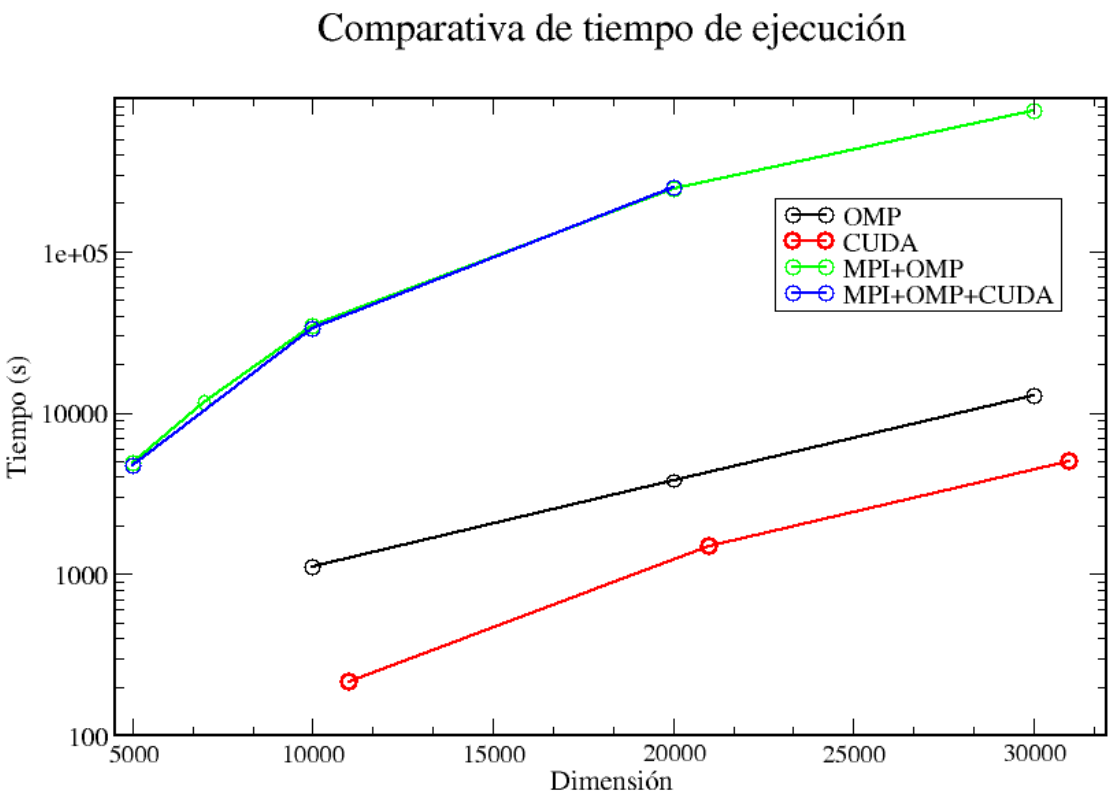

Figura 5.3: Tiempo de ejecución en segundos de las implementaciones paralela en memoria compartida (con leyenda OMP), paralela heterogénea en memoria compartida (con leyenda CUDA), híbrida paralela en memoria distribuida y compartida (con leyenda MPI+OMP) y paralela híbrida heterogénea (con leyenda MPI+OMP+CUDA) contra dimensiones de matrices cuadradas.

Se observa en la Figura 5.3 que la implementación heterogénea (con leyenda CUDA) es la más rápida, las implementaciones no híbridas fueron más rápidas que las versiones híbridas. Los tiempos de ejecución de las implementaciones híbridas fueron similares y altas a comparación de la versión no híbridas. Al realizar las pruebas de la implementación híbrida paralela (con leyenda MPI+OMP) se revisó con el equipo de monitoreo de la supercomputadora que las pruebas se estaban realizando correctamente, este no fue el caso para la implementación paralela híbrida heterogénea (con leyenda MPI+OMP+CUDA), ya que la estancia de investigación había acabado y no es posible pedir esa información fuera del departamento de monitoreo.

5.3.3. Error absoluto

En la Figura 5.4 se muestra una figura comparativa del error absoluto entre las implementaciones paralela en memoria compartida (con leyenda OMP), paralela heterogénea en memoria compartida (con leyenda CUDA), híbrida paralela en memoria

distribuida y compartida (con leyenda MPI+OMP) y paralela híbrida heterogénea (con leyenda MPI+OMP+CUDA) para diferentes dimensiones de matrices cuadradas densas.

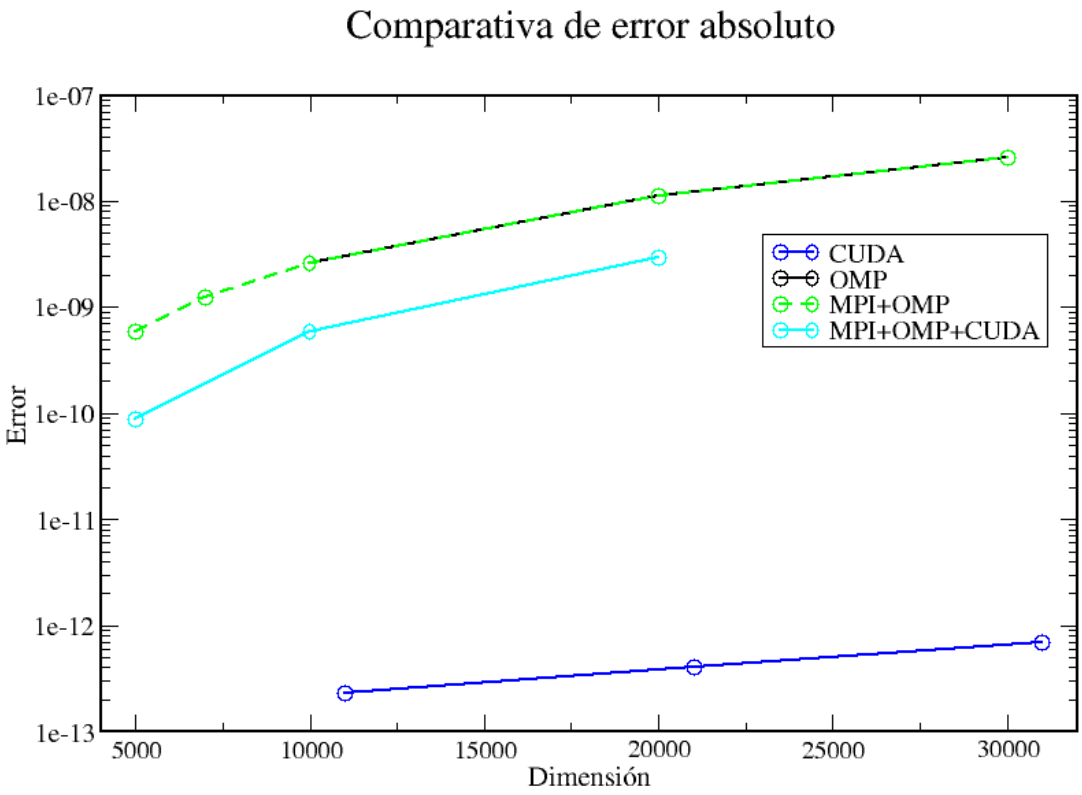

Figura 5.4: Error absoluto de las implementaciones paralela en memoria compartida (con leyenda OMP), paralela heterogénea en memoria compartida (con leyenda CUDA), híbrida paralela en memoria distribuida y compartida (con leyenda MPI+OMP) y paralela híbrida heterogénea (con leyenda MPI+OMP+CUDA) contra dimensiones de matrices cuadradas.

En la Figura 5.4 se muestra que la implementación paralela heterogénea (con leyenda CUDA) es la que tiene el menor error absoluto en el orden de 10^{-13}, la implementación paralela en memoria compartida (con leyenda OMP) tiene el segundo menor error absoluto y las versiones híbridas tienen el mismo orden de error absoluto en las dimensiones 10000, 20000 y 30000.

Capítulo 6

Conclusiones

En este trabajo se revisaron varios indicadores de desempeño de la implementación como tiempo de ejecución, aceleración, entre otros. Se notó que los mejores indicadores para nuestro trabajo son el tiempo de ejecución y el error absoluto. El tiempo de ejecución fue seleccionado como una indicador porque no se pudieron realizar pruebas por complicaciones técnicas de la función `LAPACKE_dgesvj` de LAPACK ni tampoco la implementación SVD de SCALAPACK en la supercomputadora Leo Atrox. El error absoluto fue la métrica elegida para medir la correctitud de las implementaciones.

En este trabajo se revisaron en la sección 2.4 diferentes métodos numéricos basados en la diagonalización de matrices simétricas de Jacobi o también llamado el método de Jacobi para descomposición espectral para la descomposición SVD. Y se notó que el método de Jacobi por un lado es el que tiene menor condición de número y error relativo [35].

En la tesis, se describieron en varias secciones las diferentes ventajas de paralelización del método seleccionado. En particular, en la sección 2.4.7 se definió por primera vez y demostramos el paralelismo de los ordenamientos paralelos. En muchos artículos se tomaba este concepto como arbitrario por su facilidad de entendimiento para un experto en paralelismo, sin embargo se propuso realizar una definición formal, junto con un lema sobre el paralelismo y su demostración. Los ordenamientos paralelos nos permitieron describir de manera sencilla e intuitiva las cargas sobre los diferentes modelos paralelos implementados.

Se desarrollaron múltiples implementaciones del método de Jacobi por un lado para arquitecturas paralelas: paralela en memoria compartida, paralela heterogénea, híbrida paralela, paralela híbrida heterogénea. Para así tener una comparación de métricas entre varios modelos de programación paralela.

Se analizaron bajo las métricas seleccionadas: tiempo de ejecución y error absoluto las diferentes implementaciones realizadas en este trabajo. Además de realizar el mismo análisis sobre las mismas métricas a la función `LAPACKE_dgesvj` de la biblioteca del estado del arte LAPACK. En las figuras 5.1, 5.2, 5.3 y 5.4 se encontró que la implementación del algoritmo del método de Jacobi por un lado más rápido y con menor error absoluto fue la implementación paralela heterogénea. Sin embargo, Los altos tiempos de las implementaciones híbridas son causadas por la red de

interconexión que usa Leo Atrox, la cual es OmniPath de Intel que ya está en desuso.

En la tabla 6.1 se muestra una comparativa de tiempo de ejecución entre implementaciones y equipo usado. Se observa que la implementación paralela heterogénea es la más rápida en los dos equipos que se realizaron la prueba y que la más lenta en la supercomputadora fueron las versiones híbridas; y en el equipo personal fue `LAPACKE_dgesvj` de la biblioteca del estado del arte LAPACK.

	Implementación				
Equipo	OpenMP	OpenMP+CUDA	MPI+OpenMP	MPI+OpenMP+CUDA	LAPACK
Personal			X	X	
Supercomputadora Leo Atrox					X

Tabla 6.1: Tabla comparativa de tiempo de ejecución entre implementaciones y equipo usado. Los recuadros con X son aquellas implementaciones que no se probaron en el equipo que se indica. Los recuadros en rojo son las implementaciones con mayor tiempo de ejecución y los recuadros en diferentes tonalidades de verde son las implementaciones con menores tiempos de ejecución, de verde oscuro a verde claro, representando de menor a mayor tiempo de ejecución respectivamente.

6.1. Contribuciones

La contribución principal de este trabajo de tesis es demostrar que es posible extraer rendimiento en algoritmos de métodos numéricos implementados en bibliotecas del estado del arte. Se demostró que los ambientes paralelos heterogéneos pueden tener mejor rendimiento aún con solo realizar una operación en el dispositivo heterogéneo, es decir que no es necesario trasladar todo el algoritmo o todos los datos al dispositivo heterogéneo sino solo los datos necesarios para una operación que acelere un algoritmo.

	Hardware		Modelo			Tecnología			
Biblioteca	CPU	GPU	Híbrido	Paralelo	Heterogéneo	MPI	OpenMP	CUDA	Kernel BLAS
LAPACK (MKL)	X			X					X
SCALAPACK	X		X	X		X			X
MAGMA	X			X			X		
SLATE	X		X	X		X	X		
Implementación paralela (OpenMP)	X			X			X		
Implementación paralela (OpenMP+CUDA)	X	X		X	X		X	X	
Implementación paralela (OpenMP+MPI)	X		X	X		X	X		
Implementación paralela (OpenMP+MPI+CUDA)	X	X	X	X	X	X	X	X	

Figura 6.1: Comparación entre 4 bibliotecas del estado del arte y las implementaciones realizadas contra el hardware que usan, modelo de programación y tecnología (biblioteca o marco de trabajo)

6.2. Trabajo futuro

Las implementaciones paralelo heterogénea y paralela híbrida heterogénea no se realizaron con streams en CUDA, este podría ser un trabajo a futuro ya que de realizar

una implementación con multiples streams y múltiples hilos podría hacer más rápida en tiempos de ejecución ya que se volvería una ejecución asíncrona de funciones en el dispositivo heterogéneo, contrario a la ejecución secuencial de funciones en el dispositivo heterogéneo que se realiza en las implementaciones heterogéneas e híbrida paralela heterogénea.

Por la manera en que se realizaron las implementaciones híbridas, haciendo pruebas con emulación local en MPI mapeando a cores antes de realizarse en la supercomputadorea Leo Atrox, es decir no se probaron mas esquemas de distribución de datos, por ejemplo asíncrona, al vuelo, entre otras.

Además, también se puede usar MPI en ambientes de un solo nodo pero que cuentan con múltiples procesadores en diferentes sockets y tienen memoria compartida por socket. Se podría realizar como trabajo a futuro una implementación paralela híbrida heterogénea que trabaje en esos ambientes con una o más GPU's y que podría ser más rápida que un comunicador entre varios nodos.

Capítulo 7

Código fuente

A continuación se comparte el repositorio GitHub acastellanos95/SVD-Implementations. Se encuentran todos los códigos, scripts y datos extraídos del equipo personal y de la supercomputadora.

Bibliografía

[1] "Image compression with singular value decomposition." http://timbaumann.info/svd-image-compression-demo/. Last accessed 8 August 2023.

[2] D. J. Evans and M. Gusev, "Systolic SVD and QR decomposition by Householder reflections," *International Journal of Computer Mathematics*, vol. 79, pp. 417–439, 2002.

[3] "Inside pascal: Nvidia's newest computing platform." https://developer.nvidia.com/blog/inside-pascal/, 2016. Last accessed 27 Jule 2023.

[4] T. Lyche, *Numerical Linear Algebra and Matrix Factorizations*. Springer International Publishing, 2020.

[5] Intel, "Developer reference for intel® oneapi math kernel library - c." https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html#gs.f6breo, 2022. Last accessed 7 November 2022.

[6] H. Meuer, E. Stohmaier, J. Dongarrra, H. Simon, and M. Meuer, "Top500 - june 2022." https://www.top500.org/lists/top500/2022/06/.

[7] "Wayback machine: Lumi supercomputer - documentation - high performance libraries." https://web.archive.org/web/20220213101321/https://docs.lumi-supercomputer.eu/development/compiling/libraries/.

[8] "About fugaku." https://www.r-ccs.riken.jp/en/fugaku/about/. Last accessed 27 Jule 2023.

[9] "Frontier - oak ridge national laboratory." https://www.olcf.ornl.gov/frontier/. Last accessed 7 November 2022.

[10] Y. Jaradat, M. Masoud, I. Jannoud, A. Manasrah, and M. Alia, "A tutorial on singular value decomposition with applications on image compression and dimensionality reduction," in *2021 International Conference on Information Technology (ICIT)*, pp. 769–772, 2021.

[11] Z. Kuang, "Singular-value decomposition and its applications," Master's thesis, University of California San Diego, California USA, 2017.

[12] Y. Wang and L. Zhu, "Research and implementation of SVD in machine learning," in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, pp. 471–475, 2017.

[13] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, vol. 17, pp. 395–416, 2007.

[14] D. Ghoshdastidar and A. Dukkipati, "Spectral clustering using multilinear SVD: Analysis, approximations and applications," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

[15] M. Aharon, M. Elad, and A. Bruckstein, "K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation," *IEEE Transactions on Signal Processing*, vol. 54, pp. 4311–4322, Nov. 2006.

[16] R. Ñanculef, P. Radeva, and S. Balocco, "Training convolutional nets to detect calcified plaque in IVUS sequences," in *Intravascular Ultrasound* (S. Balocco, ed.), pp. 141–158, Elsevier, 2020.

[17] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*. Cambridge University Press, 3 ed., 2020.

[18] "Scopus." `https://www.scopus.com/`.

[19] E. J. Kontoghiorghes, ed., *Handbook of Parallel Computing and Statistics*. Chapman and Hall/CRC, December 2005.

[20] L. Han and M. Neumann, "Linear algebra," in *Handbook of Linear Algebra*, pp. 41–678, Chapman and Hall/CRC, nov 2013.

[21] "Parallel programming models for dense linear algebra on heterogeneous systems," *Supercomputing Frontiers and Innovations*, vol. 2, 4 2016.

[22] M. Gates, A. Charara, J. Kurzak, A. YarKhan, M. Al Farhan, D. Sukkari, and J. Dongarra, "SLATE users' guide, SWAN no. 10," Tech. Rep. ICL-UT-19-01, Innovative Computing Laboratory, University of Tennessee, July 2020. revision 07-2020.

[23] D. Liu, R. Li, D. J. Lilja, and W. Xiao, "A divide-and-conquer approach for solving singular value decomposition on a heterogeneous system," in *Proceedings of the ACM International Conference on Computing Frontiers - CF '13*, ACM Press, 2013.

[24] M. Narwaria and W. Lin, "SVD-based quality metric for image and video using machine learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 42, no. 2, pp. 347–364, 2012.

[25] N. C. for Biotechnology Information, "Geo accession viewer." `https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE32490`, Oct 2012.

[26] S. L. Brunton and J. N. Kutz, *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

[27] D. S. Watkins, *Fundamentals of Matrix Computations*. Wiley-Interscience, 2008.

[28] R. A. van de Geijn and M. Myers, *Advanced Linear Algebra Foundations to Frontiers*. 2023.

[29] A. Cauchy, I. royale, and D. freres, *Cours d'analyse de l'Ecole royale polytechnique; par m. Augustin-Louis Cauchy ... 1.re partie. Analyse algébrique*. de l'Imprimerie royale, 1821.

[30] G. W. Stewart, *Matrix Algorithms*. Society for Industrial and Applied Mathematics, January 1998.

[31] G. H. Golub and C. F. Van Loan, *Matrix computations*. John Hopkins University press, 4 ed., 2013.

[32] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Society for Industrial and Applied Mathematics, 1996.

[33] Å. Björck, *Numerical Methods in Matrix Computations*. Springer International Publishing, 2015.

[34] J. H. Wilkinson, "Note on the quadratic convergence of the cyclic Jacobi process," *Numerische Mathematik*, vol. 4, pp. 296–300, Dec. 1962.

[35] J. Demmel and K. Veselić, "Jacobi's method is more accurate than QR," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 4, pp. 1204–1245, 1992.

[36] S. Kurgalin and S. Borzunov, *A practical approach to high-performance computing*. Springer International Publishing, 2019.

[37] A. H. Sameh, "On Jacobi and Jacobi-like algorithms for a parallel computer," *Mathematics of Computation*, vol. 25, no. 115, pp. 579–590, 1971.

[38] T. Sterling, M. Anderson, and M. Brodowicz, *High Performance Computing*. Elsevier, 2018.

[39] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, eds., *Sourcebook of Parallel Computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[40] W. Hwu, D. Kirk, and I. E. Hajj, *Programming massively parallel processors: A hands-on approach*. Morgan Kaufmann, 2023.

[41] M. Geshi, ed., *The Art of High Performance Computing for Computational Science, Vol. 1.* Springer Singapore, 2019.

[42] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A quantitative approach.* Morgan Kaufmann, 2019.

[43] B. Lewis and D. J. Berg, *Threads primer: A guide to multithreaded programming.* SunSoft, 1996.

[44] B. Barney, "Posix threads programming." `https://hpc-tutorials.llnl.gov/posix/`.

[45] V. Eijkhout, *Parallel Programming for Science and Engineering. The Art of HPC.*, vol. 2. 3 ed., 2022.

[46] M. Zahran, *Heterogeneous computing: Hardware and software perspectives.* Association for Computing Machinery, 2019.

[47] "Cuda c++ programming guide release 12.1." `https://docs.nvidia.com/cuda/archive/12.1.0/cuda-c-programming-guide/index.html`. Last accessed 27 Jule 2023.

[48] B. Barney, "Openmp." `https://hpc-tutorials.llnl.gov/openmp/`.

[49] B. Barney, "Mpi." `https://hpc-tutorials.llnl.gov/mpi/`.