Universidad Michoacana de San Nicolás de Hidalgo

Facultad de Ingeniería Eléctrica

SOLUCION DE SISTEMAS DE ECUACIONES Ax=b BAJO UN ESQUEMA DE PROCESAMIENTO EN PARALELO

TESIS

Que para obtener el Titulo de
INGENIERO ELECTRICISTA

Presenta
Luis Alberto Patricio Martínez

Asesor
Dr. Norberto García Barriga

Morelia, Michoacán, Junio de 2007

Agradecimientos

Quiero agradecer a todas las personas que hicieron posible que el presente trabajo llegara a su fin:

- A mi asesor de tesis: Dr. Norberto García Barriga, por todo tu apoyo, tiempo y sus invaluables enseñanzas.
- A mis padres: Miguel Patricio Ambriz, por su apoyo y Maria Luz Martínez Espinoza, por darme la vida, el cariño y el apoyo que me sigues brindando, y a pesar de mis errores me sigue dando ánimos para seguir adelante, gracias mama por ser mi madre.
- A mis hermanos: Ramiro, por creer en mí. Cesar, por el apoyo que siempre me has dado y por tus consejos que me ayudaron a crecer como persona. Ana Lilia, por el apoyo y comprensión en los momentos más difíciles que tuve en algún tiempo, así como también, a sus hijas Maritza y Litzy Lucero. Miguel, por su apoyo y enseñanzas de hermano. Ilifonsa, por apoyarme. Y a mis hermanos Los Cuates (Carlos y Armando), por ser más que hermanos, mis amigos incondicionales, he aprendido mucho de ustedes y siempre seremos mas que hermanos los mejores amigos.
- A mis amigos: Alejandro (A.S.V., Chaflán), Alfredo (Tapia), Jonatan, Rene, Gibran, Juan Carlos (Charly), Luis (Lucho), Lucero, Mio Sid (Sid), Aldo Iván, Moisés (mi cuñado), Miguel Ángel Garrido y Familia, y por todos los demás que me apoyan de algún modo.

Dedicatoria

Dedico el presente trabajo con todo mi cariño a:

- **A mi madre**: por darme esta herencia que es lo más grande que pueda recibir un hijo.

Gracias madrecita…

Resumen

En este trabajo se propone la implementación de las operaciones matriciales básicas tales como inversión, multiplicación, suma/resta y transposición usando una plataforma de procesamiento en paralelo. El desempeño de estas rutinas en paralelo se evalúa en problemas de pequeña y grandes dimensiones que requieren precisamente de operaciones matriciales. En particular, estas operaciones matriciales en paralelo se utilizan para mostrar su utilidad en cálculos como la solución de sistemas de ecuaciones algebraicas de la forma **Ax**=**b**. El tipo de programación que se utiliza es la programación con *múltiples-hilos*, la cual permite trabajar directamente con cada uno de los procesadores de una computadora de memoria compartida con más de un procesador. En este trabajo se realiza la inversión de una matriz de dimensiones *n*x*n* basada en una descomposición LU del tipo Crout, mientras que las operaciones suma/resta, multiplicación y transpuesta se realizan seccionando en bloques de renglones y columnas las matrices involucradas en los cálculos. Se reportan casos de estudio con resultados obtenidos en una computadora con dos procesadores.

Contenido

Lista de Figuras

Lista de Tablas

Lista de Símbolos y Abreviaturas

t	Tiempo.
seg.	Segundo.
M	Mega.
G	Giga.
Hz	Hertz.
Gb	Gigabytes.
Mb	Megabytes.
PC	Computadora Personal.
CPU	Unidad de Procesamiento Central.
A	Matriz A.
L	Matriz L (Matriz triangular inferior).
U	Matriz U (Matriz triangular superior).
$\mathbf{A}^{-1}$	Matriz Inversa.
I	Matriz identidad.
b	Vector b.
d	Vector d.
x	Vector x.
SISD	Flujo de Instrucciones Simple, Flujo de Datos Simple (*Single Instruction Stream, Single Data Stream*).
SIMD	Flujo de Instrucciones Simple, Flujo de Datos Múltiple (*Single Instruction Stream, Multiple Data Stream*).
MISD	Flujo de Instrucciones Múltiple, Flujo de Datos Simple (*Multiple Instruction Stream, Single Data Stream*).
SIMD	Flujo de Instrucciones Múltiple, Flujo de Datos Múltiple (*Multiple Instruction Stream, Multiple Data Stream*).
PVM	Máquina Virtual Paralela (Parallel Virtual Machine).
MPI	Interfaz de Envió de Mensajes (*Message Passing Interface*).

SPMD	Programa Multiple Dato Multiple (*Multiple Program Mutiple Data*).
MPP	Multiprocesador.
LWPs	Procesos ligeros.
POSIX	Sistema Operativo de Interfase Portátil (*Portable Operating System Interface*).
GUI	Interfase Gráfica del Usuario (*Interface Graphics User*).

Capítulo 1

Introducción

En muchos problemas de ingeniería la solución de un sistema involucra y la aplicación de diversas operaciones matriciales básicas y métodos de álgebra lineal. Por ejemplo, la solución del sistema de ecuaciones **Ax**=**b** es un problema genérico que ha recibido gran atención y para el cual se han planteado diferentes alternativas de solución como lo es la eliminación de Gauss, Descomposición LU, entre otros. En áreas específicas como los circuitos eléctricos, la solución de un sistema práctico involucra una gran cantidad de variables de estado, lo cual implica un esfuerzo computacional considerable. Es por ello que la aplicación de técnicas de procesamiento en paralelo para contar con herramientas de cálculo matricial en paralelo permite reducir considerablemente el tiempo de cómputo demandado.

1.1 Antecedentes

Sin duda, el rápido auge de la tecnología de procesamiento en paralelo durante las últimas décadas ha tenido un gran impacto tanto en la academia como en la industria. En el ámbito de la ingeniería eléctrica es aplicable en áreas tales como teoría de circuitos [1], sistemas de control [2], máquinas eléctricas [3] y sistemas de potencia [4]. En muchos de estos problemas de ingeniería la solución de un sistema involucra la solución de un sistema de ecuaciones que normalmente se representa como **Ax**=**b**. En áreas específicas como los sistemas eléctricos de potencia, la solución de un sistema práctico involucra miles de variables de estado, lo cual implica una demanda computacional muy importante. En problemas ampliamente conocidos como flujos de potencia [5], o la determinación de estado estable por medio de técnicas de aceleración [6], se utiliza el método de Newton-Raphson como un elemento primordial de estas soluciones. Dicho método Newton requiere la solución de un sistema de ecuaciones algebraicas.

Debido a los requerimientos crecientes de muchos sistemas, muchos controladores modernos encontrados en áreas de control digital necesitan recursos de cómputo poderosos

para alcanzar el desempeño requerido. Por ejemplo, el diseño de nuevas estructuras y la mejora del desempeño de aeronaves requiere de leyes de control más complejas que deben ser calculadas a altas relaciones de muestreo con el objeto de poder utilizarlas dentro de un lazo de control en tiempo rea. El control de un motor eléctrico requiere de intervalos de muestreo cortos debido a las constantes de tiempo pequeñas que se utilizan y requiere que los cálculos asociados al control se realicen en un intervalo de tiempo relativamente corto. Los sistemas muti-variables agregan una complejidad adicional a los cálculos del control debido a que varias señales de control se deben calcular simultáneamente. En donde la confiabilidad es esencial por ejemplo, en aplicaciones aeroespaciales y nucleares, la incorporación de características asociadas a las tolerancias de fallas agrega más complejidad a los programas. Esto ha estimulado la investigación para explotar el potencial de nuevas arquitecturas de computadoras de bajo costo y alta velocidad para ser aplicadas a la solución de estos problemas [7].

Es por ello que en este trabajo se propone la implementación de una librería de operaciones matriciales desarrolladas en un ambiente de programación en paralelo. Este conjunto de operaciones matriciales incluye la inversión, la transposición, suma/resta y multiplicación, as cuales pueden ser utilizadas en aplicaciones que demanden un esfuerzo computacional importante. Esta librería en paralelo de operaciones matriciales permite aprovechar la ventaja de contar actualmente con equipos de cómputo de una gran velocidad de procesamiento, capacidad de memoria y multiprocesadores. Se proveen as rutinas en paralelo de las operaciones matriciales utilizando un lenguaje de nivel medio como lo es el C++ y programación con *multi-hilos* –el desempeño de estas sub-rutinas en paralelo se evalúa en problemas de pequeña y gran escala que requieren precisamente de operaciones matriciales como inversión, transpuesta, multiplicación y suma/resta.

1.2 Objetivos

El objetivo general de esta tesis es implementar por medio de herramientas de álgebra lineal y programación en paralelo las operaciones matriciales básicas. Además, la implementación de las operaciones matriciales en paralelo de forma modular permitirá resolver problemas como el sistema de ecuaciones $\mathbf{Ax}=\mathbf{b}$.

Los objetivos particulares de este trabajo son:

- Analizar las posibilidades de paralelizar métodos tales como el Gauss, Gauss-Jordan, descomposición LU, etc.
- Aplicar programación con *hilos* para paralelizar la descomposición LU para calcular la matriz inversa $\mathbf{A}^{-1}$.
- Implementar las operaciones de multiplicación, suma/resta y transposición matricial en un esquema de programación en paralelo.
- Minimizar los tiempos de cómputo asociadas a las herramientas de algebra lineal utilizadas para la solución de un sistema de ecuaciones algebraicas.

1.3 Justificación

Los cálculos matriciales es un problema importante y fundamental en el algebra lineal. Problemas de algebra lineal tales como sistemas de ecuaciones lineales, problemas de mínimos cuadrados y problemas de valores característicos, son fundamentales para determinar las soluciones de ecuaciones diferenciales y problemas de optimización. De esta manera, por ejemplo, un algoritmo más rápido para multiplicar matrices implica contar con algoritmos más rápidos para todos estos problemas. Hablando de la multiplicación matricial esta operación se encuentra también en el cálculo de la transformación de coordenadas para robots y graficas de computadora.

Tomando en cuenta que los cálculos se realizan normalmente con un solo procesador en una computadora digital, el tiempo demandado para realizar operaciones matriciales de grandes dimensiones puede ser considerable. En cambio, con el procesamiento en paralelo se puede disminuir este tiempo dependiendo de las partes del problema que se puedan paralelizar y del número de procesadores con los que se trabaje.

Por otra parte, hoy en día los equipos de cómputo que se pueden adquirir cuentan con poderosos circuitos integrados con múltiples procesadores en un sólo núcleo. Es decir, la tendencia que se percibe es que la computadora personal que posee un usuario promedio tenga por lo menos un par de elementos de procesamiento, lo cual requiere de herramientas de cómputo adecuadas que exploten al máximo los recursos de cómputo disponibles. Para lograr esto, se aplican técnicas de procesamiento en paralelo, entre las cuales la programación con *múltiples-hilos* es una de las más adecuadas.

1.4 Metodología

La metodología seguida en este trabajo se basa primeramente en la implementación de métodos de algebra lineal como la descomposición de Crout para hacer la descomposición de la matriz **A** en **L** y **U**. El desarrollo de operaciones matriciales tales como multiplicación adición/sustracción y transposición bajo un ambiente de programación en paralelo se basa en la partición de las matrices en bloques de renglones y columnas, las cuales se asignan a diferentes procesadores. La elección del lenguaje de programación C++ para implementar todas las rutinas asociadas a este trabajo ofrece la posibilidad de contar con un lenguaje de nivel medio, el cual es eficiente y portable. Además, este lenguaje de programación cuenta con la librería *pthread* la cual se necesita para realizar la programación en paralelo. La programación en paralelo se lleva a cabo mediante la programación con *hilos* ya que representa una opción eficiente de explotar los recursos de una PC con más de un procesador de memoria compartida. Los casos de estudio que se analizan en este trabajo para representar sistemas de grandes dimensiones se crean mediante una rutina de valores aleatorios. Dichos sistemas de grandes dimensiones se usan para invertir su correspondiente matriz **A** y realizar las operaciones de suma/resta, multiplicación y transposición. Además, se usan dichas operaciones matriciales en paralelo para obtener la solución de un sistema de ecuaciones **Ax**=**b**.

1.5 Descripción de los Capítulos

En el Capítulo 1 se presenta una introducción a este trabajo y se resaltan los beneficios que se obtienen en el análisis de diversos problemas cuando se aplica procesamiento en paralelo. Así mismo, se describe el objetivo, justificación y metodología de este trabajo.

En el Capítulo 2 se presentan los conceptos básicos para la solución de sistemas de ecuaciones algebraicas lineales de la forma **Ax**=**b**, así como también los métodos para resolver estos sistemas de ecuaciones.

En el Capítulo 3 se da una explicación de las arquitecturas de computadoras, y se presentan tres diferentes formas de programación en paralelo. Además se implementa la

descomposición LU con *hilos* y las operaciones de multiplicación, suma/resta y transposición matricial en un ambiente de programación en paralelo.

En el Capítulo 4 se presentan casos de estudio para la inversión, multiplicación, suma/resta y transposición de una matriz de *n*x*n*. Asimismo, se presenta un caso de estudio en donde se aplican las operaciones matriciales en paralelo para solucionar un sistema de ecuaciones **Ax**=**b**.

En el Capítulo 5 se presentan las conclusiones generales y trabajos futuros.

Capítulo 2

Solución de Ecuaciones Algebraicas Lineales

Este trabajo se enfoca a la solución de sistemas de ecuaciones **Ax**=**b** en donde **A** es no-singular y el rango de **A** es *n*. Los casos indeterminados (en donde el número de ecuaciones *M* es menor que el número de incógnitas *N*) y sobre-determinados (en donde

$M > N$) no se consideran en esta tesis. Estos casos no se consideran en esta tesis ya que su solución requiere la utilización de técnicas tales como descomposición en valores singulares de la matriz **A**, lo cual cae fuera de los alcances de dicha tesis. Una primera alternativa para solucionar el sistema de ecuaciones consiste en calcular $\mathbf{A}^{-1}$, donde $\mathbf{A}^{-1}$ denota la matriz inversa de **A** y después multiplicar ambos lados por $\mathbf{A}^{-1}$ obteniendo $\mathbf{A}^{-1}\mathbf{Ax}=\mathbf{A}^{-1}\mathbf{b}$, $\mathbf{A}^{-1}\mathbf{A}=\mathbf{I}$ siendo **I** la matriz identidad. Es decir, $\mathbf{x}=\mathbf{A}^{-1}\mathbf{b}$, el cual representa el vector solución. Sin embargo, esta alternativa sufre de inestabilidad numérica producido por errores de redondeo en los casos donde se presenten números con punto flotante en lugar de enteros. La alternativa usada es la descomposición LU, la cual es más estable numéricamente (es decir, no existe el acarreo de errores) y aproximadamente 3 veces más rápida que la alternativa anterior implementado con el método Gauss-Jordan para una matriz **A** de cualquier dimensión [7]. Una de las ventajas de determinar la descomposición LU para una matriz **A** radica en el hecho de que el sistema de ecuaciones se puede resolver más fácilmente utilizando matrices triangulares [8].

En este capítulo se presentan las ecuaciones algebraicas lineales simultáneas que en general se pueden representar como la Ecuación (2.1),

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \ldots + a_{0n}x_n &= b_0 \\ a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ \vdots \qquad\quad \vdots \qquad\quad & \vdots \\ a_{n0}x_0 + a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned} \tag{2.1}$$

en donde a y b son coeficientes constantes.

Las técnicas de algebra lineal que se describen en este capítulo son la eliminación Gaussiana, eliminación Gauss-Jordan y la descomposición LU, en las cuales se involucra la combinación de ecuaciones para eliminar las incógnitas [9].

2.1 Eliminación Gaussiana

La técnica sistemática de eliminación hacia adelante y la sustitución hacia atrás representan los pasos principales involucrados en la eliminación Gaussiana. Para implementar esta técnica en computadoras digitales se requiere de algunas modificaciones para obtener un algoritmo confiable. En particular, el programa debe de evitar la división

entre cero y, por lo tanto, la eliminación gaussiana simple no evita este problema. En esta sección se presentan características adicionales necesarias para tener un programa de cómputo efectivo.

Considere el siguiente conjunto de *n* ecuaciones, representado por la Ecuación (2.1). La eliminación Gaussiana consiste en dos fases: la eliminación hacia adelante de las incógnitas y su solución hacia atrás. La eliminación hacia adelante consiste en reducir el conjunto de ecuaciones en un sistema triangular superior (vea la Figura 2.1).

$$a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \ldots + a_{0n}x_n = b_0 \tag{2.2a}$$

$$a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \tag{2.2b}$$

$$\begin{array}{ccc} . & . & . \\ . & . & . \\ . & . & . \end{array}$$

$$a_{n0}x_0 + a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \tag{2.2c}$$

El paso inicial consiste en eliminar la primera incógnita x_0, desde la segunda hasta la *n*-ésima ecuación. Para esto, se multiplica la Ecuación (2.2a) por a_{10}/a_{00} para obtener,

$$a_{10}x_0 + \frac{a_{10}}{a_{00}}a_{01}x_1 + \ldots + \frac{a_{10}}{a_{00}}a_{0n}x_n = \frac{a_{10}}{a_{00}}b_0 \tag{2.3}$$

Ahora, esta ecuación se puede restar de la Ecuación (2.2b) para obtener,

$$\left(a_{11} - \frac{a_{10}}{a_{00}}a_{01}\right)x_1 + \ldots + \left(a_{1n} - \frac{a_{10}}{a_{00}}a_{0n}\right)x_n = b_1 - \frac{a_{10}}{a_{00}}b_0 \tag{2.4}$$

ó bien,

$$a'_{11}x_1 + \ldots + a'_{1n}x_n = b'_1 \tag{2.5}$$

en donde el superíndice prima indica que los elementos han cambiado sus valores originales.

Este procedimiento se repite para las ecuaciones restantes. Por ejemplo, la Ecuación (2.2b) se puede multiplicar por a_{20}/a_{00} y el resultado se resta a la tercera ecuación. Se repite el procedimiento para las ecuaciones restantes y da como resultado el siguiente sistema modificado [9].

$$\left[\begin{array}{ccc|c} a_{00} & a_{01} & a_{02} & b_0 \\ a_{10} & a_{11} & a_{12} & b_1 \\ a_{20} & a_{21} & a_{22} & b_2 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} a_{00} & a_{01} & a_{02} & b_0 \\ & a'_{11} & a'_{12} & b'_1 \\ & & a''_{22} & b''_2 \end{array}\right] \Rightarrow$$

Eliminación hacia adelante

$$\begin{aligned} x_2 &= b''_2 / a''_{22} \\ x_1 &= \left(b'_2 - a'_{12}\right)x_2 / a'_{11} \\ x_0 &= \left(b_1 - a_{01}x_1 - a_{02}x_2\right)/ a_{00} \end{aligned}$$

Sustitución hacia atrás

Figura 2.1 Fases de la eliminación Gaussiana.

$$a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \ldots + a_{0n}x_n = b_0 \tag{2.6a}$$

$$a'_{11}x_1 + a'_{12}x_2 + \ldots + a'_{1n}x_n = b'_1 \tag{2.6b}$$

$$a'_{21}x_1 + a'_{22}x_2 + \ldots + a'_{2n}x_n = b'_2 \tag{2.6c}$$

$$\begin{array}{cc} . & . \\ . & . \\ . & . \end{array}$$

$$a'_{n1}x_1 + a'_{n2}x_2 + \ldots + a'_{nn}x_n = b'_n \tag{2.6d}$$

Para los pasos anteriores, la Ecuación (2.2a) es llamada la ecuación pivote y a_{00} es llamado el coeficiente pivote o elemento**.** El proceso de multiplicación del primer renglón por a_{10}/a_{00} es equivalente a dividirla entre a_{00} y multiplicarla por a_{10}. Algunas veces la operación de división es referida como normalización. Se hace esta distinción porque un elemento pivote cero puede interferir con la normalización y causar una división entre cero.

Ahora se repite el procedimiento antes descrito para eliminar la segunda incógnita de la Ecuación (2.6c) hasta (2.6d). Para realizar esto, se multiplica la Ecuación (2.6b) por $a_{21}^{'}/a_{11}^{'}$ y se resta el resultado a la Ecuación (2.6c). Se realiza en forma similar la eliminación para las ecuaciones restantes de tal manera que se obtiene,

$$a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + ... + a_{0n}x_n = b_0 \tag{2.7a}$$

$$a_{11}^{'}x_1 + a_{12}^{'}x_2 + ... + a_{1n}^{'}x_n = b_1^{'} \tag{2.7b}$$

$$a_{22}^{''}x_2 + ... + a_{2n}^{''}x_n = b_2^{''} \tag{2.7c}$$

$$\begin{array}{cc} . & . \\ . & . \\ . & . \end{array}$$

$$a_{n2}^{''}x_2 + ... + a_{nn}^{''}x_n = b_n^{''} \tag{2.7d}$$

donde el superíndice bi-prima indica que los elementos se han modificado dos veces.

El procedimiento se puede continuar usando las ecuaciones pivotes restantes. El procedimiento final en la secuencia es el uso de la (n-1)-ésima ecuación para eliminar el término x_{n-1} de la n-ésima ecuación. En este punto el sistema se transforma en un sistema triangular superior [9].

$$a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + ... + a_{0n}x_n = b_0 \tag{2.8a}$$

$$a_{11}^{'}x_1 + a_{12}^{'}x_2 + ... + a_{1n}^{'}x_n = b_1^{'} \tag{2.8b}$$

$$a_{22}^{''}x_2 + ... + a_{2n}^{''}x_n = b_2^{''} \tag{2.8c}$$

$$\begin{array}{cc} . & . \\ . & . \\ . & . \end{array}$$

$$a_{nn}^{(n-1)}x_n = b_n^{(n-1)} \tag{2.8d}$$

El procedimiento de sustitución hacia atrás consiste básicamente en que la Ecuación (2.8d) se resuelva para x_n, es decir,

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}} \tag{2.9}$$

Este resultado se puede sustituir hacia atrás en la (n-1)-ésima ecuación y resolverse para x_{n-1}. El procedimiento, que se repite para evaluar las x´s restantes, se puede representar mediante la expresión,

$$x_i = \frac{b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)} x_j}{a_{ii}^{(i-1)}} \tag{2.10}$$

para i=n-1, n-2, …, 0

2.2 Eliminación Gauss-Jordan

El método Gauss-Jordan es una variación de la eliminación de Gauss. La principal diferencia consiste en que cuando una incógnita se elimina en el método de Gauss-Jordan, ésta se elimina de todas las ecuaciones excepto su elemento pivote de cada renglón. Además, todos los renglones se normalizan al dividirlos entre su elemento pivote. De esta forma, el paso de eliminación genera una matriz identidad en vez de una triangular (véase Figura 2.2). En consecuencia, no es necesario usar la sustitución hacia atrás para obtener la solución, ya que como se puede observar en la Figura 2.2 se tiene directamente la solución del sistema de ecuaciones.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \mid b_0 \\ a_{10} & a_{11} & a_{12} \mid b_1 \\ a_{20} & a_{21} & a_{22} \mid b_2 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} 1 & 0 & 0 \mid b_0^{(n)} \\ 0 & 1 & 0 \mid b_1^{(n)} \\ 0 & 0 & 1 \mid b_2^{(n)} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} x_0 & 0 & 0 \mid b_0^{(n)} \\ 0 & x_1 & 0 \mid b_1^{(n)} \\ 0 & 0 & x_2 \mid b_2^{(n)} \end{bmatrix}$$

Figura 2.2 Representación grafica del método Gauss-Jordan.

2.3 Descomposición LU

La descomposición LU es una de las técnicas más atractivas para resolver sistemas de ecuaciones lineales, ya que el paso de eliminación que demanda un esfuerzo de cómputo importante se puede reformular de tal manera que involucre sólo operaciones sobre los coeficientes de la matriz **A**. De esta forma, esta descomposición es muy adecuada para aquellas situaciones donde se deben evaluar diferentes vectores **b** del lado derecho de la igualdad para una misma matriz **A**.

Tanto la eliminación Gaussiana como la eliminación Gauss-Jordan comparten la desventaja que debe conocerse con anticipación los vectores del lado derecho de la igualdad. Por su parte, la descomposición LU no comparte esta desventaja. Un motivo adicional para utilizar la descomposición LU se basa en el hecho de que proporciona un medio eficiente para calcular la matriz inversa. Como se describió en la sección anterior, la eliminación de Gauss está diseñada para resolver sistemas de ecuaciones algebraicas lineales como el siguiente,

$$\mathbf{Ax} = \mathbf{b} \tag{2.11}$$

Recordando que la eliminación de Gauss implica dos pasos (Eliminación hacia adelante y sustitución hacia atrás, véase en la Figura 2.1) es precisamente el paso de eliminación hacia adelante el que demanda un esfuerzo computacional mayor. Esto es particularmente importante y evidente para sistemas de ecuaciones de grandes dimensiones [9].

La Ecuación (2.11) se puede reordenar para obtener,

$$\mathbf{Ax} - \mathbf{b} = \mathbf{0} \tag{2.12}$$

Supóngase que (2.12) se puede expresar como un sistema triangular superior,

$$\begin{bmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \tag{2.13}$$

Se observa que ésta es similar a la manipulación que ocurre en el primer paso de la eliminación de Gauss. Es decir, se usa la eliminación para reducir el sistema a una forma triangular superior. La Ecuación (2.13) puede también ser expresada en notación matricial y reordenada para tener,

$$\mathbf{Ux} - \mathbf{d} = \mathbf{0} \tag{2.14}$$

Ahora bien, supóngase que existe una matriz diagonal inferior con números 1 en la diagonal,

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ l_{10} & 1 & 0 \\ l_{20} & l_{21} & 1 \end{bmatrix} \tag{2.15}$$

que tiene la propiedad que cuando se multiplica por la Ecuación (2.14), el resultado es la Ecuación (2.12). Esto es,

$$\mathbf{L}\left(\mathbf{Ux} - \mathbf{d}\right) = \mathbf{Ax} - \mathbf{b} \tag{2.16}$$

Si esta ecuación se cumple, de las reglas de multiplicación de matrices se obtiene:

$$\mathbf{LU} = \mathbf{A} \tag{2.17}$$

y

$$\mathbf{Ld} = \mathbf{b} \tag{2.18}$$

Una estrategia de dos pasos (véase la Figura 2.3) para obtener soluciones tiene su base en las Ecuaciones (2.14), (2.17) y (2.18):

1.- Paso de la descomposición LU. Se factoriza ó descompone **A** en matrices triangulares, inferior **L** y superior **U**.

2.- Paso de sustitución. **L** y **U** se usan para determinar una solución **x** dado un vector **b**. Este paso, a su vez, se divide en dos. Primero, la Ecuación (2.18) se usa para generar un vector intermedio **d** por sustitución hacia adelante. Luego, el resultado es sustituído en la Ecuación (2.14), la que se resuelve por sustitución hacia atrás para **x**.

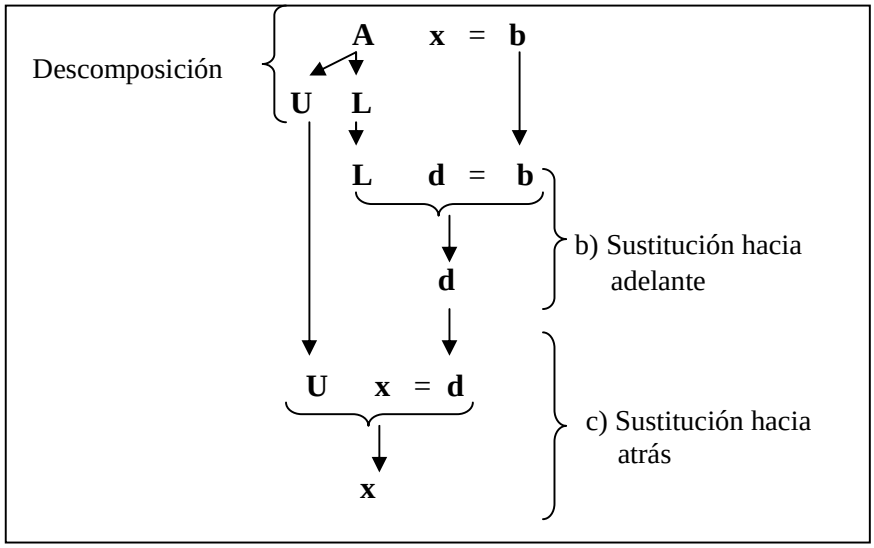

Figura 2.3 Procedimiento para la solución de **Ax**=**b** con la descomposición LU.

2.3.1 Descomposición LU con Base en la Eliminación de Gauss

A primera vista podría parecer que la descomposición LU no está relacionada con la eliminación de Gauss, pero puede usarse esta última para descomponer **A** en **L** y **U**. Esto puede verse fácilmente para **U**, el cual es un producto directo de la eliminación hacia adelante. Es precisamente en el paso que corresponde a esta última eliminación en donde se pretende reducir la matriz de coeficientes **A** a la forma,

$$\mathbf{U} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ 0 & a_{11}^{'} & a_{12}^{'} \\ 0 & 0 & a_{22}^{''} \end{bmatrix} \tag{2.19}$$

la cual esta en la forma triangular superior deseada.

Aunque podría no ser tan claro, la matriz **L** se produce durante este paso. Esto se puede ilustrar en forma fácil para un sistema de tres ecuaciones,

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} \tag{2.20}$$

El primer paso en la eliminación de Gauss es multiplicar el renglón 1 por un factor (Ecuación (2.3)),

$$f_{10} = \frac{a_{10}}{a_{00}} \tag{2.21}$$

y restando el resultado del segundo renglón para eliminar a_{10}. De forma similar, el renglón 1 se multiplica por,

$$f_{20} = \frac{a_{20}}{a_{00}} \tag{2.22}$$

y restando el resultado del tercer renglón para eliminar a_{20}. El paso final es multiplicar el segundo renglón modificado por,

$$f_{21} = \frac{a_{21}^{'}}{a_{11}^{'}} \tag{2.23}$$

y restar el resultado del tercer renglón para eliminar $a_{21}^{'}$.

Ahora suponga que se realizan todas esas operaciones en la matriz **A**. Resulta claro que si no se quiere cambiar la ecuación, se tiene que realizar lo mismo para el lado derecho **b**. Pero no existe ninguna razón para realizar las operaciones en forma simultánea. Entonces, se podría salvar las *f* y después trabajar en **b**. Después de la eliminación, la matriz **A** puede, por lo tanto, escribirse como,

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ f_{10} & a'_{11} & a'_{12} \\ f_{20} & f_{21} & a''_{22} \end{bmatrix} \tag{2.24}$$

De hecho, esta matriz representa un almacenamiento eficiente de la descomposición LU de **A**,

$$\mathbf{A} \rightarrow \mathbf{LU} \tag{2.25}$$

donde,

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ f_{10} & 1 & 0 \\ f_{20} & f_{21} & 1 \end{bmatrix} \tag{2.26}$$

$$\mathbf{U} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ 0 & a'_{11} & a'_{12} \\ 0 & 0 & a''_{22} \end{bmatrix} \tag{2.27}$$

Después de descomponer la matriz, se puede generalizar una solución para un vector particular **b** del lado derecho de la igualdad. Esto se hace en dos procedimientos. Como se muestra en la Figura 2.3.

El procedimiento de sustitución hacia adelante mostrado en la Figura 2.3 se puede representar en forma concisa como,

$$d_i = d_i - \sum_{j=i}^{i-1} L_{ij} b_j \tag{2.28}$$

para i=2,3,…,n

El segundo procedimiento, entonces, solo toma en cuenta la implementación de la sustitución hacia atrás, como en la Ecuación (2.14). Así en forma similar las Ecuaciones (2.9) y (2.10). El procedimiento de sustitución hacia atrás se puede representar en forma concisa como:

$$x_n = \frac{d_n}{U_{nn}} \tag{2.29}$$

$$x_i = \frac{d_i - \sum_{j=i+1}^{n} U_{ij} d_j}{U_{ii}} \tag{2.30}$$

para $i=n-1, n-2, \ldots,0$

2.3.2 Descomposición de Crout

Los métodos de descomposición LU separan el esfuerzo computacional consumido en la eliminación de la matriz **A** y el demandado por manipulaciones del lado derecho **b**. Entonces, una vez que la matriz **A** ha sido descompuesta, los múltiples vectores del lado derecho **b** se pueden evaluar de manera eficiente. En este trabajo se implementó la descomposición LU mediante a versión de la descomposición de Crout. Una representación esquemática de las evaluaciones necesarias en la descomposición LU de Crout, se muestra en la Figura 2.4. La aproximación de la descomposición de Crout se puede implementar con la siguiente serie de formulas concisas [9],

$$l_{i1} = a_{i1} \qquad \text{para i=1,2…,n} \tag{2.31}$$

$$u_{ij} = \frac{a_{1j}}{l_{11}} \qquad \text{para j=2,3…,n} \tag{2.32}$$

Para j=2,3…,n

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \qquad \text{para i=j, j+1, j+2, …,n} \tag{2.33}$$

$$u_{jk} = \frac{a_{jk} - \sum_{i=1}^{j-1} l_{ji} u_{ik}}{l_{jj}} \quad \text{para k=j, j+1, j+2, …,n} \qquad (2.34)$$

y

$$l_{nn} = a_{nn} - \sum_{k=1}^{j-1} l_{nk} u_{kn} \qquad (2.35)$$

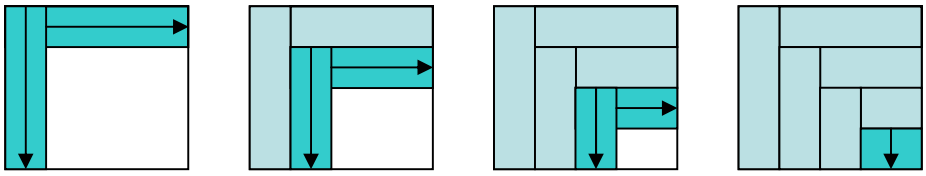

Figura 2.4 Representación esquemática de la descomposición de Crout.

En la Figura 2.5 muestra el pseudocódigo del programa que se implementó en esta tesis basada en la descomposición de Crout. Como se puede observar en la Figura 2.5, la descomposición LU se hace en forma secuencial. Primeramente, en la Fase 1 se iguala la matriz **A** = **U** y **L** = **I**, en donde **I** es la matriz identidad. Enseguida, en la Fase 2 se realiza la eliminación hacia adelante, la cual queda guardada en la matriz **U**. El ciclo externo mueve hacia abajo la matriz del renglón pivote al siguiente. El ciclo intermedio mueve hacia abajo el renglón pivote para cada renglón subsecuente y es aquí donde ocurre la eliminación y se calculan los elementos de la matriz **L**. Finalmente, el ciclo más interno avanza por las columnas para eliminar o transformar los elementos de un renglón particular y calcular los elementos de la matriz **U**.

descomposicion_LU(A, n)

Fase 1

U = A y L = I

Fase 2

for *i*=0,1,2,…, *n*-1

for *j*=*i*+1, … , *n*

L*ij* → U*ji* / U*ii*

```
            for  k=i, …, n
              Hacer (Ujk  →  Ujk  -  Lji Uik)
            end
          end
        end
end descomposicion_LU
```

Figura 2.5 Pseudocódigo de la descomposición de Crout.

Ejemplo:

Usando la descomposición LU de Crout mostrada en la Figura 2.5, la siguiente matriz **A** se descompone en **L** y **U**.

$$\begin{bmatrix} 3.0000 & -0.1000 & -0.2000 \\ 0.1000 & 7.0000 & -0.3000 \\ 0.3000 & -0.2000 & 10.0000 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7.85 \\ -19.3 \\ 71.4 \end{bmatrix}$$

Implementando la eliminación hacia adelante se tiene una matriz triangular inferior **L** y se obtiene también una matriz triangular superior **U**.

$$= \begin{bmatrix} \mathbf{3.000000} & \mathbf{-0.100000} & \mathbf{-0.200000} \\ 0.033333 & \mathbf{7.003333} & \mathbf{-0.293333} \\ 0.100000 & -0.027130 & \mathbf{10.012042} \end{bmatrix}$$

ó bien,

$$= \begin{bmatrix} 1.000000 & 0.000000 & 0.000000 \\ 0.033333 & 1.000000 & 0.000000 \\ 0.100000 & -0.027130 & 1.000000 \end{bmatrix}$$

$$= \begin{bmatrix} 3.000000 & -0.100000 & -0.200000 \\ 0.000000 & 7.003333 & -0.293333 \\ 0.000000 & 0.000000 & 10.012042 \end{bmatrix}$$

Haciendo la sustitución hacia adelante en la matriz **L** se obtiene:

$$d[0] = 7.850000$$
$$d[1] = -19.561667$$
$$d[2] = 70.084293$$

Ahora la sustitución hacia atrás en la matriz **U** se obtiene:

$$x[0] = 3.000000$$
$$x[0] = -2.500000$$
$$x[0] = 7.000000$$

Una forma de verificar si está bien hecha la descomposición es hacer la multiplicación matricial **L*U**. Se puede observar a continuación que la matriz **A** obtenida es exactamente la matriz **A** original.

```
 >> L=[1.000000       0.000000       0.000000
         0.033333       1.000000       0.000000
         0.100000      -0.027130       1.000000]
L =
                1.0000       0              0
                0.0333       1.0000         0
                0.1000      -0.0271         1.0000

 >> U=[3.000000       -0.100000      -0.200000
         0.000000       7.003333       -0.293333
         0.000000       0.000000       10.012042]
U =
              3.0000    -0.1000     -0.2000
                0         7.0033    -0.2933
                0           0         10.0120

>> A=L*U
A =
              3.0000   -0.1000   -0.2000
              0.1000    7.0000   -0.3000
              0.3000   -0.2000   10.0000
```

2.4 Aplicación de la Descomposición LU

Una de las aplicaciones de la descomposición LU es la inversión de una matriz, ya que teniendo descompuesta la matriz **A** se facilita el cálculo de la inversa de esta matriz. Esto se debe a que la inversa de la matriz **A** se puede calcular en una forma columna-por-columna

mediante la generación de soluciones con vectores unitarios que están en el lado derecho de la igualdad.

Sea una matriz **D** la inversa de la matriz **A**. Entonces se debe cumplir que $\mathbf{A}\cdot\mathbf{D}=\mathbf{I}$, es decir, la multiplicación de la matriz **A** por su inversa **D** da como resultado la matriz identidad **I**. Esto es,

$$\mathbf{A}\cdot\mathbf{D}=\mathbf{I}=\begin{bmatrix}1 & 0 & 0 & \cdots & 0\\ 0 & 1 & 0 & \cdots & 0\\ 0 & 0 & 1 & \cdots & 0\\ \vdots & \vdots & \vdots & & \vdots\\ 0 & 0 & 0 & \cdots & 1\end{bmatrix} \tag{2.36}$$

Supóngase que la primer columna de la matriz **D** es $\left[d_{11}d_{21}\cdots d_{n1}\right]^T$**,** entonces,

$$\mathbf{A}\times\begin{bmatrix}d_{11}\\ d_{21}\\ \vdots\\ d_{n1}\end{bmatrix}=\begin{bmatrix}1\\ 0\\ \vdots\\ 0\end{bmatrix} \tag{2.37}$$

De donde se deduce que la primer columna de la matriz inversa **D** se obtiene resolviendo la expresión (2.37). De manera similar para la segunda columna de **D**, se tiene,

$$\mathbf{A}\times\begin{bmatrix}d_{11}\\ d_{21}\\ \vdots\\ d_{n1}\end{bmatrix}=\begin{bmatrix}0\\ 1\\ \vdots\\ 0\end{bmatrix} \tag{2.38}$$

En donde la solución del sistema de ecuaciones (2.38) proporciona la segunda columna de la matriz inversa **D**.

En la Figura 2.6 se muestra el diagrama flujo del programa para realizar la inversión de una matriz de *n*x*n* dimensiones. Primeramente se hace la descomposición de la matriz **A** en **L** y **U**, posteriormente se crea un vector **b** a partir de una columna de la matriz identidad para utilizarlo haciendo sustitución hacia adelante con la matriz **L** y después haciendo sustitución hacia atrás con la matriz **U**. Hecho esto se guarda el resultado en una columna

de la matriz inversa. En el apéndice A se presenta el listado completo del programa implementado en C++ para realizar la inversión de una matriz.

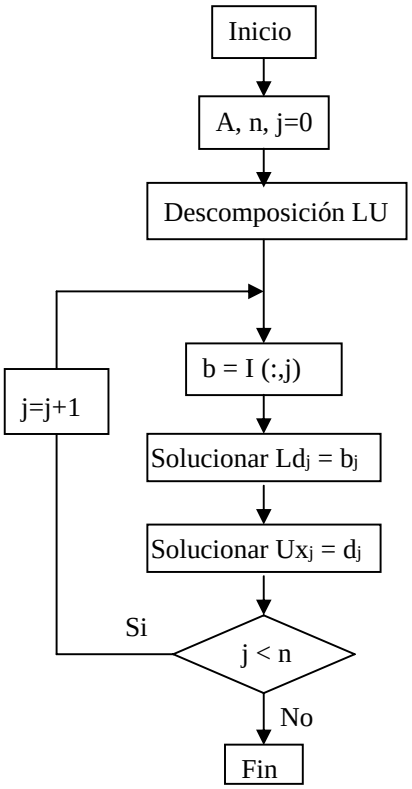

Figura 2.6 Diagrama de flujo de la inversión de una matriz *n*x*n*.

2.5 Ejemplo de Aplicación del Método

Del ejemplo presentado anteriormente se tiene que la descomposición de Crout es,

$$= \begin{bmatrix} \mathbf{3.000000} & \mathbf{-0.100000} & \mathbf{-0.200000} \\ 0.033333 & \mathbf{7.003333} & \mathbf{-0.293333} \\ 0.100000 & -0.027130 & \mathbf{10.012042} \end{bmatrix}$$

Tomado la matriz triangular inferior **L** y el vector **b** constituido por la primer columna de la matriz identidad **I**, como se muestra a continuación.

$$\begin{bmatrix} 1.000000 & 0.000000 & 0.000000 \\ 0.033333 & 1.000000 & 0.000000 \\ 0.100000 & -0.027130 & 1.000000 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Resolviendo con la sustitución hacia adelante queda **d** = [1 -0.03333 -0.1009]^T. Este vector se puede usar en el lado derecho con la forma de la Ecuación. (2.13), y ahora tomando la matriz triangular superior **U**.

$$\begin{bmatrix} 3.000000 & -0.100000 & -0.200000 \\ 0.000000 & 7.003333 & -0.293333 \\ 0.000000 & 0.000000 & 10.012042 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -0.03333 \\ -0.1009 \end{bmatrix}$$

la cual se puede resolver por sustitución hacia atrás para **x** = [0.332489 -0.005182 -0.010078]^T, lo cual es la primera columna de la matriz inversa,

$$^{-1} = \begin{bmatrix} 0.332489 & 0 & 0 \\ -0.005182 & 0 & 0 \\ -0.010078 & 0 & 0 \end{bmatrix}$$

Para determinar la segunda columna, el vector **b** representa la segunda columna de la matriz identidad, entonces,

$$\begin{bmatrix} 1.000000 & 0.000000 & 0.000000 \\ 0.033333 & 1.000000 & 0.000000 \\ 0.100000 & -0.027130 & 1.000000 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

y se puede resolver para **d**. El vector **b** se usa en la Ecuación (2.13) para determinar **x** = [0.004944 0.142903 0.002710]^T, en donde este vector **x** representa la segunda columna de la matriz inversa.

$$^{-1} = \begin{bmatrix} 0.332489 & 0.004944 & 0 \\ -0.005182 & 0.142903 & 0 \\ -0.010078 & 0.002710 & 0 \end{bmatrix}$$

Por ultimo, los procedimientos de sustitución hacia adelante y hacia atrás pueden ser implementados con **b** = [0 0 1]^T, y resolviendo para **x** queda, **x** = [0.006798 0.004183 0.099880]^T

$$^{-1} = \begin{bmatrix} 0.332489 & 0.004944 & 0.006798 \\ -0.005182 & 0.142903 & 0.004183 \\ -0.010078 & 0.002710 & 0.099880 \end{bmatrix}$$

Reportando la matriz inversa con 20 dígitos significativos se tiene,

$$^{-1} = \begin{bmatrix} 3.324887213398430e-001 & 4.944070205796923e-003 & 6.798096532970769e-003 \\ -5.181765888767929e-003 & 1.429026446021688e-001 & 4.183444020289704e-003 \\ -1.007829695797065e-002 & 2.709730785869468e-003 & 9.987972598441668e-002 \end{bmatrix}$$

La validez de este resultado se puede comprobar al verificar mediante la expresión **A*****A**$^{-1}$ = **I**.

```
>> A=[3.000000      -0.100000      -0.200000
      0.100000       7.000000      -0.300000
      0.300000      -0.200000      10.000000]
A =
    3.0000   -0.1000   -0.2000
    0.1000    7.0000   -0.3000
    0.3000   -0.2000   10.0000

>> Ainversa=[
 3.324887213398430e-001   4.944070205796923e-003   6.798096532970769e-003
-5.181765888767929e-003   1.429026446021688e-001   4.183444020289704e-003
-1.007829695797065e-002   2.709730785869468e-003   9.987972598441668e-002]

>> I=A*Ainversa
 1.000000000000000e+000  -1.084202172485504e-019                   0
-7.806255641895632e-018   1.000000000000000e+000   3.469446951953614e-018
                       0                        0   1.000000000000000e+000
```

Como se puede observar al calcular la matriz identidad **I=A A**$^{-1}$, los elementos fuera de la diagonal son menos que a 1x10^{-17}. Por lo tanto, se puede concluir que el cálculo de la matriz inversa como se presentó en esta sección permite obtener errores menores de 1x10^{-17}.

2.6 Conclusiones

En este capítulo se presentó en forma breve las bases del método de Gauss-Jordan y la eliminación de Gauss. Se presentaron las opciones de descomposición basadas en la eliminación Gaussiana y en la descomposición de Crout. Esta alternativa basada en la descomposición de Crout se utilizó en este trabajo para descomponer la matriz **A** en **L** y **U** e implementar posteriormente la inversión de la matriz **A**. Se mostró un ejemplo de descomposición de una matriz **A** con este método, así como también su inversa y su solución al sistema de ecuaciones **Ax**=**b**.

Capítulo 3

Programación en Paralelo

En este capítulo se presentan las principales plataformas de programación en paralelo. Se mencionan tres tipos de programación en paralelo como son el PVM (*Máquina Virtual Paralela*), MPI (*Interface de Envió de Mensajes*) y se detalla la programación con *hilos*. En particular, se le dedica más espacio en está tesis a la descripción de la programación con *hilos* ya que este tipo de programación es la que se utiliza en este trabajo. Algunas razones por las cuáles se eligió la programación con *hilos* para realizar este trabajo son: *i.*- se cuenta con computadoras de memoria compartida, las cuáles son ideales para programarse con *mútiples-hilos*, *ii.*- la programación con *hilos* tiene menos necesidades de comunicación para transferir datos a los diferentes procesos y, *iii.*- la programación con *hilos* no requiere el uso de una red de Internet ó ethernet para comunicarse entre los diferentes elementos de procesamiento. La aplicación de éstas técnicas de programación en paralelo y procedimientos de algebra lineal permite contar con herramientas poderosas para solucionar problemas prácticos de grandes dimensiones como flujos de potencia, estabilidad transitoria, etc. Existen otros tipos de programación en paralelo como UNIX, PCN, C-Linda, EPT y CHARM, pero no son considerados en este trabajo.

3.1 Introducción

Los sistemas de cómputo con procesamiento en paralelo surgen de la necesidad de resolver problemas complejos con tiempos de cómputo menores, utilizando las ventajas de memoria, velocidad de los procesadores, formas de interconexión de éstos y distribución de las tareas a realizar. Para resolver un problema en particular, se aplica una arquitectura ó combinación de múltiples arquitecturas, ya que cada una ofrece ventajas y desventajas que tienen que ser revisadas antes de implementar la solución del problema en una arquitectura en particular. También es necesario tomar en cuenta algunos aspectos básicos para paralelizar un algoritmo, la partición del problema en múltiples tareas y como distribuir estas según la arquitectura en particular con que se trabaje.

La Figura 3.1 describe las diferentes etapas que se deben seguir en el diseño de un algoritmo implementado en paralelo. Se puede apreciar de esta figura que básicamente se deben de seguir 4 fases de diseño, y de las cuales a continuación se presenta una breve descripción [8].

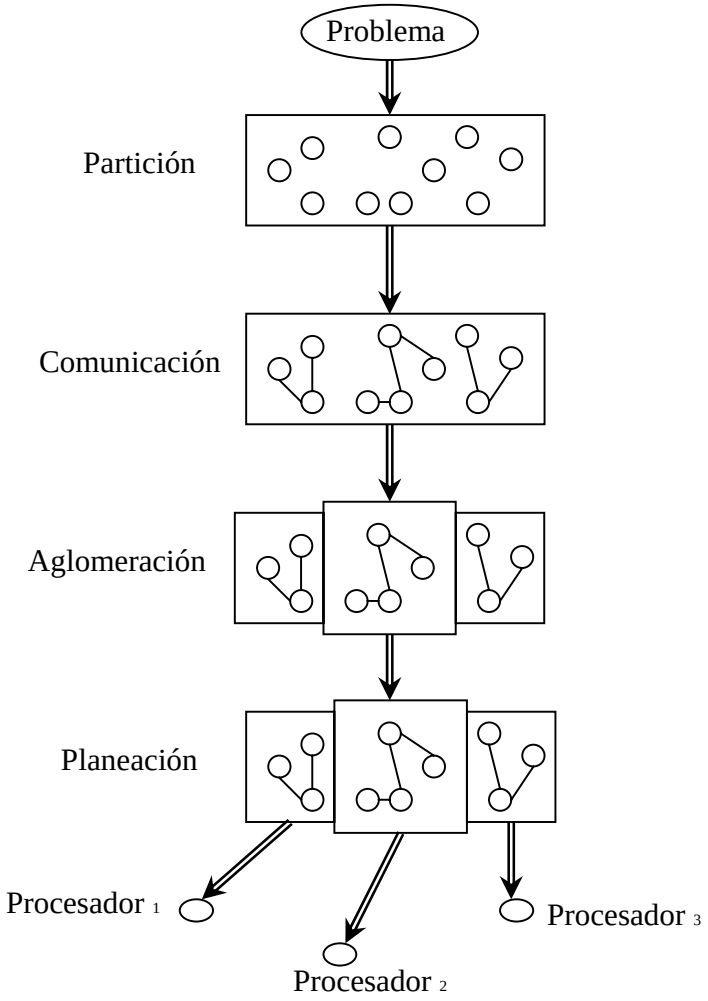

Figura 3.1 Etapas del diseño de un algoritmo implementado en paralelo.

1) Partición: La etapa de partición tiene como objetivo mostrar las posibilidades de ejecución en paralelo mediante la descomposición de todos los cálculos en pequeñas tareas. Aspectos tales como el número de procesadores en el tipo de arquitectura en paralelo que se va a utilizar son ignorados en esta etapa y, por lo tanto, la atención se enfoca en definir un gran número de pequeñas tareas con el objeto de descomponer el problema con una alta resolución.

2) Comunicación: La etapa de comunicación pretende establecer una estructura de algoritmos de comunicación adecuados para coordinar la ejecución de las tareas. En este sentido el diseño de la fase de comunicación debe asegurar que la información se transfiera entre las tareas y, de esta manera, permitir que procedan los cálculos.
3) Aglomeración: la etapa de aglomeración se plantea para evaluar la relación que existe entre la realización de la tarea y los requerimientos de comunicación en términos de desempeño y de los costos de implementación. Es decir, para una arquitectura en paralelo determinada sería ineficiente el crear muchas más tareas que el número de procesadores disponibles. Entonces, en esta etapa de aglomeración se debe de hacer una revisión de los resultados arrojados por las etapas de partición y comunicación. Un aspecto crítico en el buen desempeño de una implementación en paralelo son los costos de comunicación por lo cual, en la etapa de aglomeración se deben de reducir los tiempos utilizados en comunicación.
4) Planeación: La etapa de planeación tiene como objetivo especificar en donde se ejecutarán cada una de las tareas. Esto es, a cada procesador se le asigna una tarea de tal forma que se maximice la utilización de cada procesador y se minimicen los costos de comunicación.

3.2 Arquitectura de las Computadoras

Las principales diferencias entre computadoras son en la forma en que los módulos de procesamiento son conectados. Los elementos convencionales y mínimos en un sistema de cómputo son el procesador y la memoria. El procesador manipula y almacena los datos en la memoria mediante instrucciones. Las instrucciones son almacenadas en los módulos de memoria y siempre hay un flujo de datos de la memoria al procesador. El flujo de datos es bi-direccional, de manera que los datos puedan ser leídos o escritos en los módulos de memoria. La Figura 3.2 muestra la interconexión entre el procesador y la memoria, la cual representa el modelo conocido como de Von Newmann [8].

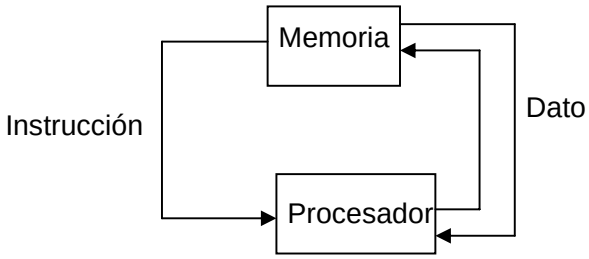

Figura 3.2 Interconexión procesador-memoria

A continuación se presentan dos clasificaciones para las computadoras digitales en las cuáles se toma en cuenta el número de procesadores que contienen y la arquitectura que utilizan para comunicar los elementos de procesamiento y los elementos de almacenaje de la información. El objetivo de esta sección es presentar dos tipos de clasificación para las computadoras actuales, así como cuáles son las ventajas e inconvenientes que cada uno ostenta. Es importante conocer estas características ya que es muy común que al resolver un problema particular se usen una o más arquitecturas de computadoras interconectadas.

Clasificación de Flynn

La clasificación clásica de arquitecturas de computadoras que hace alusión a sistemas con uno o varios procesadores la publicó Flynn por primera vez en 1966. Esta clasificación se basa en el flujo que siguen los datos dentro de la máquina y de las instrucciones sobre esos datos. Se define como flujo de instrucciones al conjunto de instrucciones secuenciales que son ejecutadas por un único procesador y como flujo de datos se define al flujo secuencial de datos requeridos por el flujo de instrucciones [8]. Con estas consideraciones, Flynn clasifica los sistemas en cuatro categorías:

- Flujo de Instrucciones Simple, Flujo de Datos Simple SISD (*Single Instruction Stream, Single Data Stream*).

- Flujo de Instrucciones Simple, Flujo de Datos Múltiple SIMD (*Single Instruction Stream, Multiple Data Stream*).
- Flujo de Instrucciones Múltiple, Flujo de Datos Simple MISD (*Multiple Instruction Stream, Single Data Stream*).
- Flujo de Instrucciones Múltiple, Flujo de Datos Múltiple MIMD (*Multiple Instruction Stream, Multiple Data Stream*).

1. SISD

Estas computadoras tienen un CPU que ejecuta una instrucción a la vez (*Single Instrucción Stream*) y estrategias de datos almacenados uno a la vez (*Single Data Stream*). La Figura 3.3 muestra la estructura general de funciones de la arquitectura SISD. Todas las computadoras SISD utilizan un registro simple, llamado contador de programa, con el cual se implementa una serie de ejecuciones de instrucciones.

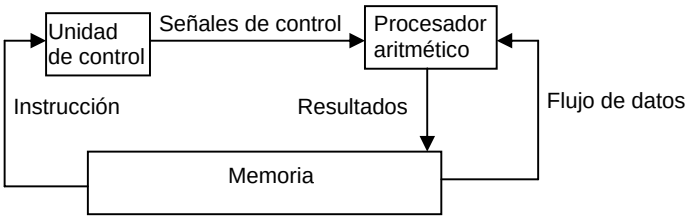

Figura 3.3 Modelo de una arquitectura SISD.

2. SIMD

Estas máquinas tienen una unidad de control que ejecuta un sólo flujo de instrucciones pero cuenta con más de un elemento de procesamiento. La unidad de control genera las señales de control para todos los procesadores, los cuales ejecutan la misma operación con diferentes datos. Ejemplos de este tipo de computadoras son la ILLIAC IV, DAP y máquinas de conexión CM-2. La Figura 3.4 muestra la estructura general de una arquitectura SIMD, la cual se reduce a una máquina Von Newmann cuando únicamente esta activo un elemento de procesamiento.

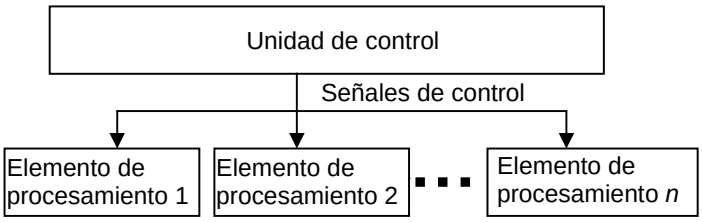

Figura 3.4 Modelo de una arquitectura SIMD.

3. MISD

En esta categoría de computadora se pueden ejecutar diferentes programas conjuntamente utilizando los mismos datos. La Figura 3.5 muestra la estructura MISD [8].

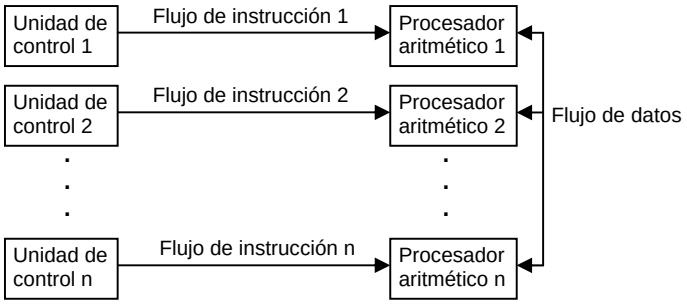

Figura 3.5 Modelo de una arquitectura MISD.

4. MIMD

Estas máquinas son llamadas multiprocesadores, las cuales tienen más de un procesador y pueden ejecutar diferentes tipos de programas (*Múltiple Instrucción Stream*) con su propio flujo de datos. En la mayoría de los sistemas MIMD, cada uno de los procesadores tiene acceso a una memoria global, con la cual se puede reducir el retraso de la comunicación del procesador. Adicionalmente, cada procesador posee una memoria local, con lo cual se evitan problemas para acceder la memoria. Algunos modelos comerciales de estas computadoras son la BBN Butterfly, la serie Alliant FX y la serie Intel Corporation’s iPSC. La Figura 3.6 representa la estructura de una arquitectura MIMD.

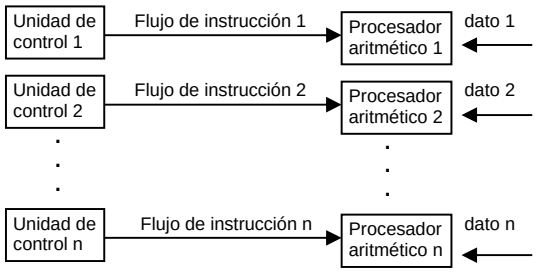

Figura 3.6 Modelo de una arquitectura MIMD.

Una segunda clasificación para las computadoras en basé al número de procesadores se define de la siguiente manera:

Micro-computadoras. Es una computadora funcional que sirve sólo para un usuario. Esta puede ser usada en el hogar y la oficina, su tamaño está basado en su memoria, capacidad de disco duro y la velocidad en un solo procesador.

Mini-computadoras. Su aplicación es más general ya que éstas pueden atender a más de un usuario. Sirven para supervisar instrumentos, equipo de prueba de laboratorios y en la actualidad cuentan con más de un procesador.

Súper-computadoras. Este tipo de computadoras ofrecen velocidades y capacidades de almacenamiento mayores que la mini ya que cuentan con un número mayor de procesadores. Se utilizan como computadoras centrales de redes.

Ultra-computadoras. Estas supercomputadoras están diseñadas para procesar aplicaciones que requieren un esfuerzo de cómputo importante y están construidas con múltiples procesadores [10].

3.3 Programación PVM

El sistema PVM (*Parallel Virtual Machine*) permite interactuar colectivamente entre estaciones de trabajo y estructuras de computadoras diferentes en un ambiente de procesamiento en paralelo. PVM fue un proyecto de colaboración entre el Laboratorio Nacional *Oak Ridge*, la Universidad de Tennessee, la Universidad Emory y la Universidad Carnegie-Mellon el cual lo comenzó Vaidy Sunderam y Al Geist en 1989 en el Laboratorio Nacional de *Oak Ridge*. El sistema PVM permite la conexión de un sistema heterogéneo de

computadoras UNIX interconectadas entre si a través de una red para ser vista por el usuario como una sola computadora.

El sistema PVM esta compuesto principalmente por dos partes:

- Un demonio llamado **pvmd** ó **pvm3** el cual reside en todas la computadoras para permitir la construcción de una máquina virtual. Por ejemplo, un programa de correo que se encarga de manejar los mensajes de entrada y salida en una computadora es un programa demonio.
- Una librería de **interfaces de rutinas** pvm. Esta librería contiene un conjunto de funciones primitivas que son necesarias para la cooperación entre tareas de una aplicación. Estas librerías contienen rutinas de envió de mensajes, coordinación de tareas y modificación de la máquina virtual.

En general, un usuario que desea ejecutar una aplicación PVM primeramente crea una máquina virtual para comenzar la programación PVM. La aplicación PVM puede entonces comenzarse desde una terminal UNIX de cualquier *hosts*. Se pueden integrar diversos usuarios a la máquina virtual y ejecutar varias aplicaciones PVM simultáneamente [8].

3.4 Programación MPI

MPI (*Message Passing Interface*) es un interfaz estandarizado para la realización de aplicaciones paralelas basadas en el envío de mensajes. MPI ofrece una gran colección de funciones que sirven para diseñar programas paralelos y no obliga a seguir una implementación en especial [11].

El modelo de programación que se utiliza con MPI es MIMD (*Multiple Instruction Streams, Múltiple Data Streams*) aunque se dan facilidades especiales para la utilización del modelo SPMD (*Single Program Multiple Data*), el cual representa un caso particular de MIMD en el que todos los procesos ejecutan el mismo programa aunque no necesariamente la misma instrucción al mismo tiempo. En este modelo de programación MPI, el programa ejecutado consiste de uno o más procesos comunicados a través de llamadas a rutinas de librerías para mandar y recibir mensajes a otros procesos. En la mayoría de las implementaciones de MPI, se crea un conjunto fijo de procesos al inicializar el programa y un proceso es creado por cada tarea. Sin embargo, estos procesos pueden ejecutar diferentes programas. De ahí que, el modelo de programación MPI es algunas veces referido como

MPMD (*Multiple Program Multiple Data*) para distinguirlo del modelo SPMD (*Simple Program Multiple Data*), en el cual cada procesador ejecuta el mismo programa [11].

Debido a que el número de procesos en una operación de MPI es normalmente fijo, se puede enfatizar en el uso de los mecanismos para comunicar datos entre procesos. Los procesos pueden utilizar operaciones de comunicación punto a punto para mandar mensajes de un proceso a otro. Estas operaciones pueden ser usadas para implementar comunicaciones locales y no estructuradas. Un grupo de procesos puede llamar colectivamente operaciones de comunicación para realizar tareas globales tales como la función transmitir (*broadcast*). La habilidad de MPI para enviar mensajes da como resultado el poder soportar comunicaciones asíncronas. Probablemente una de las características más importantes del MPI es el soporte para la programación modular. Un mecanismo llamado comunicador permite al programador del MPI definir módulos que encapsulan estructuras internas de comunicación (estos módulos pueden ser combinados secuencialmente y paralelamente).

Los algoritmos paralelos ejecutan llamadas a operaciones para coordinar la comunicación en múltiples procesos. Por ejemplo, todos los procesos pueden necesitar cooperación para invertir una matriz distribuida o para sumar un conjunto de números distribuidos en cada proceso. Claramente, estas operaciones globales pueden ser implementadas por un programador usando las funciones *send* y *receive*. Por conveniencia y para permitir la optimización, MPI provee un conjunto especializado de funciones colectivas de comunicación que proveen operaciones de este tipo [11].

3.5 Programación con *Hilos*

El desarrollo del concepto de programación con *múltiples-hilos* tuvo su auge en los años 60s. Su implementación en los sistemas UNIX comenzó a mediados de los 80s. Mientras hay acuerdo acerca de lo que son los *múltiples-hilos* y las características necesarias para soportarlo, las interfaces usadas para implementar *múltiples-hilos* han disentido grandemente.

La palabra *múltiples-hilos* puede ser traducida como *múltiples hilos de control* ó *flujos múltiples de control.* Mientras un proceso tradicional UNIX siempre contiene un sólo *hilo*

de control, el *múltiples-hilos* (MT) separa un proceso de muchos *hilos*, ejecutándolos cada cual corriendo independientemente [12].

La aplicación con *múltiples-hilos* tiene las siguientes ventajas:

- **Versatilidad**

  Cualquier programa en el que muchas actividades no son dependientes una de otra pueden ser rediseñada a fin de que cada actividad sea definida con un *hilo*. Por ejemplo, el usuario con *múltiples-hilos* de una Interfaz Gráfica del Usuario (GUI) no tiene que esperar para que una actividad se complete antes de comenzar otra.

- **Uso eficiente de multiprocesadores**

  Típicamente, las aplicaciones que tienen requerimientos de concurrencia con *hilos* no necesitan tomar en cuenta el número de procesadores disponibles. El desempeño de la aplicación mejora directamente con los procesadores adicionales. Las aplicaciones y algoritmos numéricos con un alto grado de paralelismo, como multiplicaciones matriciales, pueden ejecutarse más rápidamente cuando son implementados con *hilos* en un multiprocesador.

3.5.1 Conceptos Asociados a la Programación con *Hilos*

A continuación se mencionan algunos conceptos básicos involucrados en la programación con *hilos* [12]:

- **Concurrencia y paralelismo**

En un proceso de *múltiples-hilos* con un sólo procesador, el procesador puede cambiar los recursos de ejecución entre los *hilos*, resultando en una ejecución concurrente. En el mismo proceso los *múltiples-hilos* en un ambiente de multiprocesadores con memoria compartida, cada *hilo* durante el proceso puede ejecutarse al mismo tiempo en un procesador separado resultando en una ejecución paralela. Cuando el proceso tiene menos ó el mismo número de *hilos* que de procesadores, el sistema de soporte de los *hilos* en conjunto con el ambiente operativo asegura que cada *hilo* se ejecute en un procesador diferente.

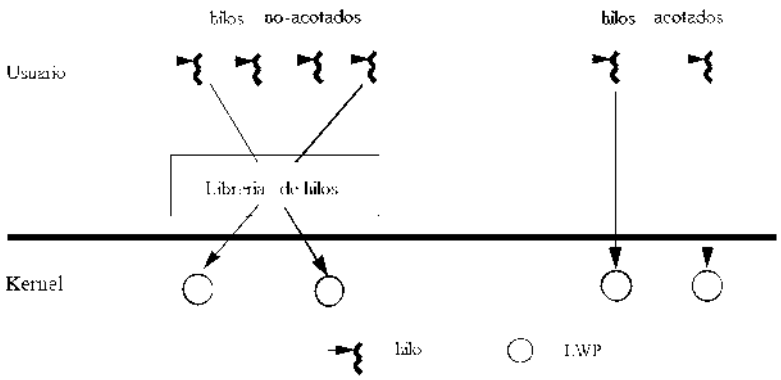

Figura 3.7 *Hilos* nivel-usuario y procesos ligeros

- **Los *hilos* nivel-usuario**

  - ✓ Este tipo específico de *hilos* tienen entre otras características que son más económicos, por que desde el punto de vista de demanda de recursos de cómputo, se les puede ver como bits de memoria virtual que son ubicados en la dirección del espacio corriendo en el tiempo.
  - ✓ Son funciones que se sincronizan rápidamente por que la sincronización esta dada para la aplicación y no a través del conjunto de programas que controlan el funcionamiento de la computadora, esto es, el sistema operativo.
  - ✓ Facilita usar la librería de *hilos*, ya sea *libpthread* o *libthread***.**

- **Procesos Ligeros (LWP´s)**

La librería de *hilos* soporta *hilos* de control llamados *procesos ligeros*. Se puede definir el LWP como en un CPU virtual que ejecuta las llamadas de código ó del sistema. En la Figura 3.7 se muestra la relación entre los *hilos* nivel-usuario y los procesos ligeros, cuando se definen *hilos* acotados ó no acotados. Los *hilos* acotados son *hilos* que son atendidos directamente por los LWP's, mientras que para los *hilos* no acotados, los recursos se obtienen a través de la librería de *hilos*.

- **Cancelación**

La cancelación del *hilo* es una opción que permite al programador tener la posibilidad de que un *hilo* termine la ejecución de cualquier otro *hilo* en el proceso. Es decir, es un recurso de programación disponible para los casos en donde el programa especifico que se este implementando así lo requiera. El *hilo* objetivo puede mantener pendiente la cancelación y puede ejecutar la aplicación específica de limpieza posteriormente.

La característica de cancelación de la librería *pthread* permite la terminación de *hilos* asíncronos ó diferir la terminación de un *hilo*. La cancelación asíncrona puede ocurrir en cualquier momento; la cancelación diferida puede ocurrir sólo en los puntos definidos. La cancelación diferida es el tipo pre-establecido.

- **Sincronización**

La sincronización permite controlar el flujo de programa y accesar a datos compartidos por los *hilos* que se ejecutan concurrentemente. Los tres modelos de sincronización existentes son**:** *mutex locks*, variables de control y semáforos.

- ✓ Los candados *mutex* permiten sólo un *hilo* a la vez para ejecutar un código específico ó acceder a datos específicos.
- ✓ Las variables control bloquean los *hilos* hasta que una condición particular sea cierta.
- ✓ Las comunicaciones por semáforos coordinan el acceso a recursos. El contador es el límite de cuántos *hilos* pueden tener el acceso al semáforo. Cuando la cuenta se alcanza, el semáforo se bloquea.

- **Creación de *Hilos***

Para crear un *hilo* se usa la función *pthread_create* de POSIX, la cual tiene la siguiente sintaxis:

pthread(&tid, NULL, rutina, arg);

en donde *tid* se define por medio de la instrucción,

pthread_t tid

la *rutina* representa una función implementada por el usuario en la cual se realiza la tarea específica encomendada al *hilo*. El argumento *arg* es utilizado por la función *rutina*.

Cuando el *hilo* fue creado exitosamente con la función *pthread_create*, se le asigna un identificador localizado en la variable *tid*. La función *pthread_create* regresa un cero cuando este se completa exitosamente. Cualquier otro valor señala que ocurrió un error.

3.5.2 Uso de candados de exclusión mutua

En la Figura 3.8 se muestra el proceso de utilización de las variables de control en conjunto con los candados *mutex lock* y *mutex unlock*. Como se observa en esta figura los candados *mutex lock* permiten que sólo un *hilo* a la vez ejecute la tarea específica asignada ó acceder a datos específicos. En el caso de esta tesis estos datos específicos son las matrices **L** y **U**.

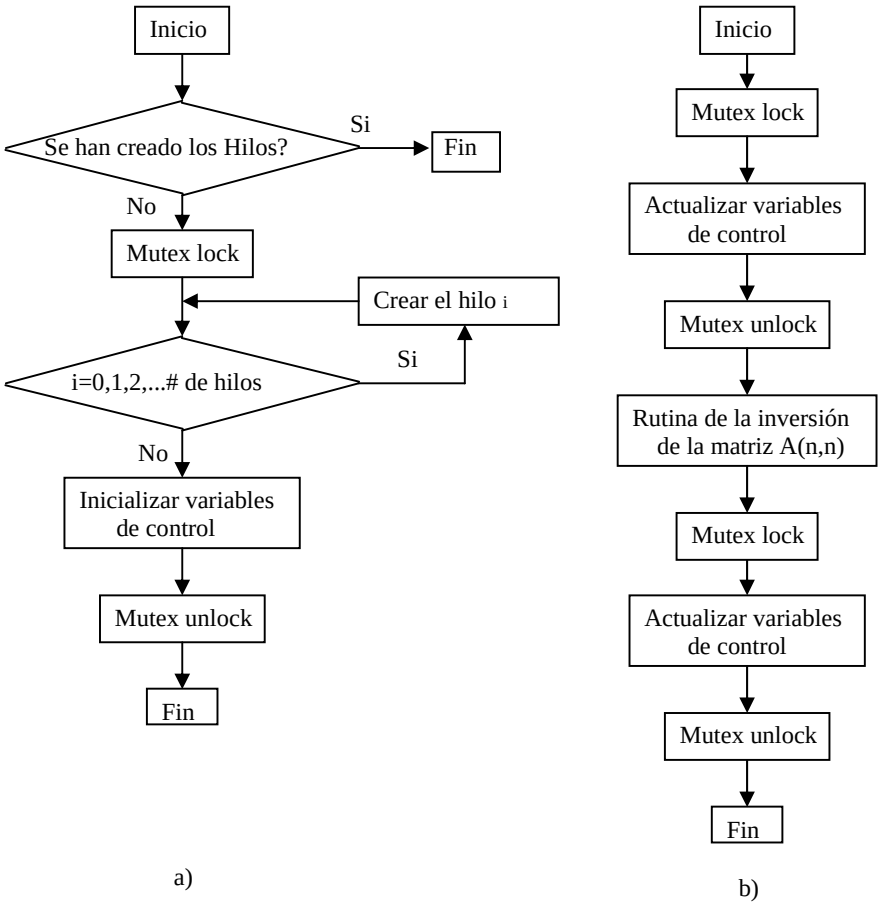

Figura 3.8 Sincronización de los *hilos* con candados de exclusión mutua.

En la Figura 3.8 (a) se muestra la creación de los *hilos*, así como también la inicialización de las variables de control usando los candados *mutex lock* y *mutex unlock*. En la Figura 3.8 (b) se muestra la utilización de los candados *mutex lock* para asignarle a cada *hilo* sólo una tarea con la ayuda de las variables de control. Cuando se ha asignado la tarea se libera el *hilo* con *mutex unlock* y realiza la tarea asignada. Una vez terminada la tarea se vuelve a adquir el candado *mutex lock*, se actualizan las variables de control y se libera nuevamente el candado *mutex unlock*. Finalmente, la función devuelve el resultado de la tarea.

Cada *hilo* calcula una columna de la matriz inversa y actualiza las variables de control para que otro *hilo* calcule la siguiente columna de la matriz inversa. Las columnas calculadas en paralelo por cada *hilo* se guardan en una matriz vacía llamada matriz inversa, hasta calcular la última columna de esta matriz inversa.

3.6 Ley de Amdahl

La ley de Amdahl [13] es un modelo matemático para mostrar los límites del procesamiento en paralelo. La expresión de la ley de Amdahl que indica el factor de aceleración de los cálculos es,

$$R(f) = \frac{1}{\frac{f}{R_H} + \frac{(1-f)}{R_L}} \tag{3.1}$$

en donde:

1. Se definen relaciones de alta ejecución, R_H y de baja ejecución R_L.
2. f es una fracción de los resultados generados en una relación alta de ejecución y (1-f) es una fracción de los resultados en una relación baja de ejecución.

Dicho en otras palabras, **f** es la parte del programa que se ejecuta secuencialmente y (1-f) es la parte del programa ejecutado en paralelo. Si **f** es 1, el factor de aceleración R(f) es igual a R_H y con **f**=0 el factor de aceleración es R_L.

Esta expresión matemática es útil para analizar el desempeño de un programa a partir de las relaciones individuales y de ejecución, tales como un conjunto de operaciones realizadas en paralelo y serialmente.

La expresión (3.1) también puede tomar la forma,

$$R(f) = \frac{p}{B_1 p + (1 - B_1)} \tag{3.2}$$

en donde B_1 y $(1 - B_1)$ representan la parte serial y paralela del programa, respectivamente. La variable *p* representa el número de procesadores.

3.7 Implementación de la Matriz Inversa con *Hilos*

Las etapas de diseño del algoritmo en paralelo que se presentaron en la Figura 3.1 se particularizan ahora para invertir una matriz, como se muestra en la Figura 3.9. A manera de ejemplo, la Figura 3.9 muestra las etapas que se deben de seguir para invertir una matriz **A** de dimensiones 3x3. Primeramente, la etapa de partición muestra las posibles tareas que se podrían paralelizar, las cuales son descomposición LU, tres sustituciones hacia adelante (SAD) y tres sustituciones hacia atrás (SAT). Posteriormente, se establece la etapa de comunicación en donde se observa que se requiere coordinar la ejecución de las diferentes operaciones de SAD y SAT. Enseguida, se continúa a la etapa de aglomeración en la cual se determina que no hay necesidad de comunicación y transferencia de datos entre los pares de procesos SAD y SAT, lo cual reduce los tiempos y costos de los cálculos. Finalmente, la etapa de planeación asigna a cada procesador una tarea específica para minimizar los costos de comunicación, la cual consiste en calcular una columna de la matriz inversa haciendo SAD y SAT. Cabe mencionar que la descomposición LU y la 1er tarea SAD-SAT se hace con el procesador 1, esto es debido a que la tarea de la descomposición LU se realiza secuencialmente ya no se paraleliza. La 2da tarea SAD-SAT se le asigna al procesador 2 y finalmente la 3er tarea SAD-SAT se le asigna al procesador 3.

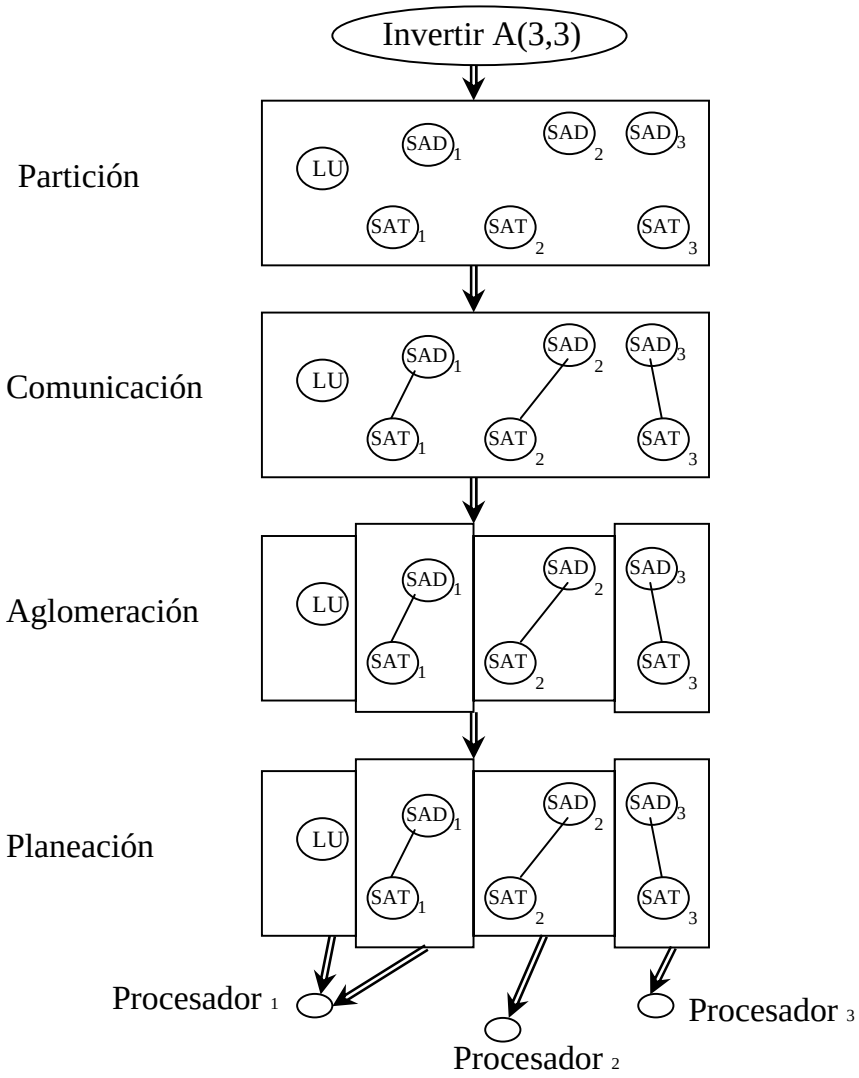

Figura 3.9 Etapas del diseño del algoritmo implementado en paralelo.

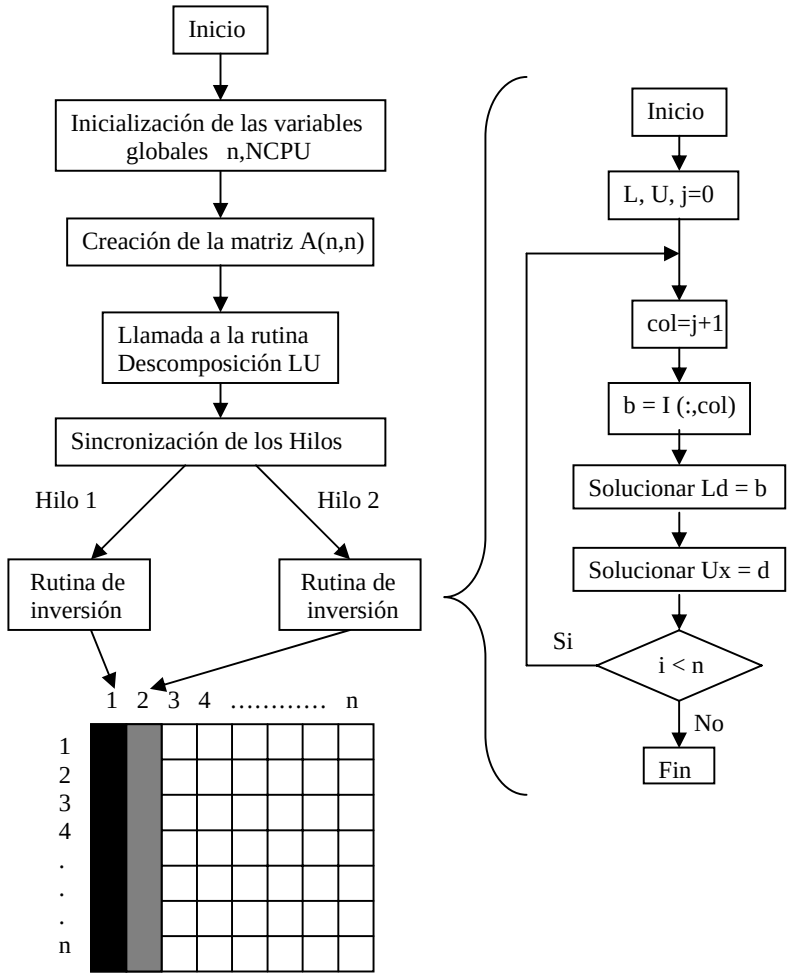

Figura 3.10 Diagrama de flujo de la matriz inversa con *Hilos*.

En la Figura 3.10 se describe el procedimiento que se debe seguir para invertir una matriz **A** de *n*x*n* dimensiones. Primeramente se inicializan algunas variables de control y se procede a realizar la descomposición de la matriz **A** en **L** y **U**. Posteriormente, se inicia el proceso del cálculo de la inversa de la matriz **A** columna por columna mediante el proceso de sincronización de los *hilos* mostrados en la Figura 3.8. A cada *hilo* se le asigna la tarea de calcular una columna de la matriz inversa mediante la ejecución de la rutina de inversión. Adicionalmente, la Figura 3.10 muestra la rutina que ejecuta cada *hilo* para invertir la matriz por columnas.

3.8 Implementación de la Multiplicación de dos Matrices con *Hilos*

El producto de una matriz **A** de *nxn* $\mathbf{A} = (a_{ij})_{nxn}$ y una matriz **B** de *nxn* $\mathbf{B} = (b_{ij})_{nxn}$, da como resultado una matriz $\mathbf{C} = (c_{ij})_{nxn}$ cuyos elementos se definen por,

$$\mathbf{C}_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj} \tag{3.3}$$

en donde $0 \le i \le n-1$ y $0 \le j \le n-1$. Este algoritmo de multiplicación matricial secuencial requiere *nxnxn* sumas y *nxnxn* multiplicaciones, en donde las sumas y las multiplicaciones de los *n* términos se realiza secuencialmente [8].

Para llevar a cabo la implementación en paralelo de la multiplicación matricial se propone que las matrices **A** y **B** se dividan en diferentes grupos de renglones y columnas, en donde cada procesador se encarga de los cálculos relacionados con un grupo de renglones de la matriz **A** y un grupo de columnas de la matriz **B**. La Figura 3.11 muestra el esquema de partición implementado en este trabajo para realizar la multiplicación matricial por columnas.

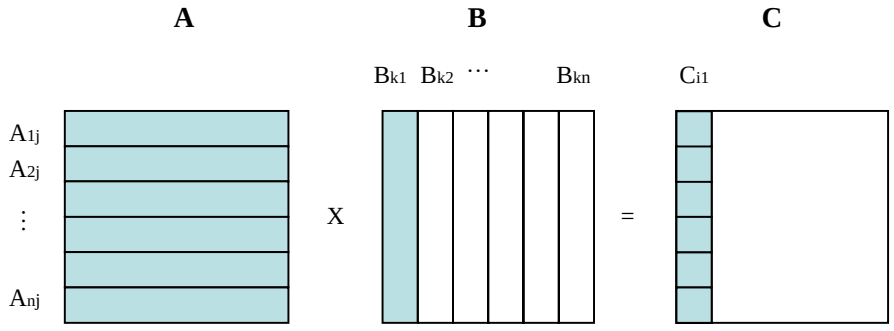

Figura 3.11 Esquema de partición para realizar la multiplicación **C**=**A**x**B**.

En esta Figura 3.11 se presenta el ejemplo de calcular una columna completa de la matriz resultante **C** usando uno de los procesadores disponibles. Como se puede apreciar, el cálculo de los *n* elementos que integran la columna resultante lo realiza un solo procesador mediante la multiplicación de cada uno de los renglones de la matriz **A** por la primer columna de la matriz **B**.

La multiplicación de una matriz $\mathbf{A}_{nxn}$ por un vector $\mathbf{b}_{nx1}$ es un caso particular de la multiplicación de dos matrices. Dicha operación se define por, por un vector $\mathbf{b}_{nx1}$.

$$\mathbf{c}_{i1} = \sum_{k=0}^{n-1} A_{ik} b_{kj} \tag{3.4}$$

en donde dicha operación matricial demanda n^2 sumas y n^2 multiplicaciones. A continuación en la Figura 3.12 se muestra el esquema de partición propuesto para llevar a cabo los cálculos del producto de la matriz **A** por el vector **b** en paralelo. En este caso se plantea asignar la tarea de multiplicar un renglón de la matriz **A** por el vector **b** a un procesador y así sucesivamente al resto de los procesadores disponibles.

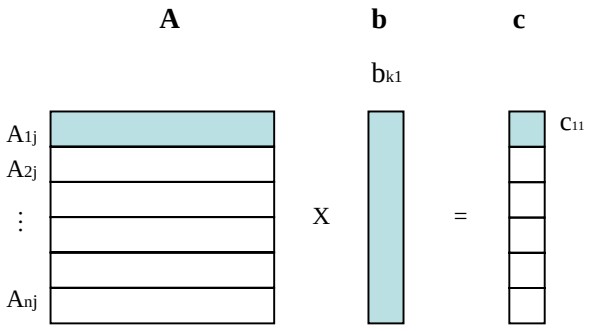

Figura 3.12 Esquema de partición para realizar la multiplicación **c**=**A**x**b**.

3.9 Implementación de la Suma/Resta de dos Matrices con *Hilos*

La suma de una matriz **A** de *nxn* $\mathbf{A} = \left(a_{ij}\right)_{nxn}$ y una matriz **B** de *nxn* $\mathbf{B} = \left(b_{ij}\right)_{nxn}$, se obtiene al sumar términos correspondientes de cada matriz y da como resultado una matriz $\mathbf{C} = \left(c_{ij}\right)_{nxn}$ es decir,

$$\mathbf{C}_{ij} = A_{ij} + B_{ij} \tag{3.5}$$

en donde $0 \le i \le n-1$ y $0 \le j \le n-1$. De manera similar, la resta de dos matrices de una matriz **A** de *nxn* $\mathbf{A} = \left(a_{ij}\right)_{nxn}$ y una matriz **B** de *nxn* $\mathbf{B} = \left(b_{ij}\right)_{nxn}$, se obtiene de la siguiente forma,

$$\mathbf{C}_{ij} = A_{ij} - B_{ij} \tag{3.6}$$

Este algoritmo de suma/resta matricial secuencial requieren *nxn* sumas/restas, en donde las sumas/restas de los *n* términos se realiza secuencialmente.

Para realizar la implementación en paralelo de la operación suma/resta matricial se propone que las matrices **A** y **B** se dividan en diferentes grupos de columnas, en donde cada procesador se encarga de los cálculos relacionados con una columna de la matriz **A** y una columna de la matriz **B**. La Figura 3.13 muestra el esquema de partición implementado en este trabajo para realizar la suma/resta matricial por columnas. En esta Figura se presenta el ejemplo de calcular una columna completa de la matriz resultante **C** usando uno de los procesadores disponibles. Como se puede apreciar, el cálculo de los *n* elementos que integran la columna lo realiza un solo procesador mediante la suma/resta de la columna de la matriz **A** con la columna de la matriz **B**.

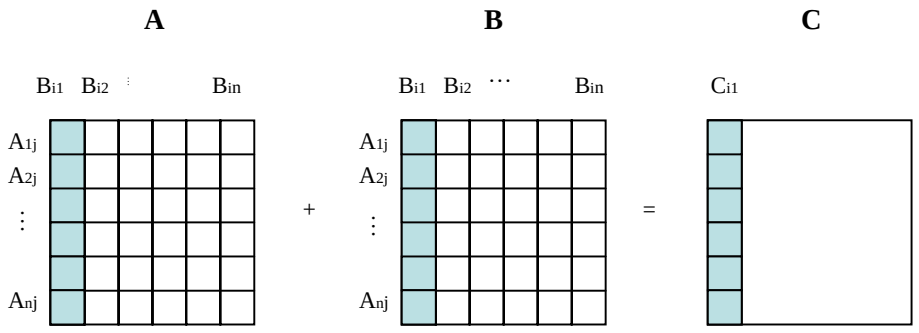

Figura 3.13 Esquema de partición para realizar la suma matricial C=A+B.

3.10 Implementación de la transpuesta de una matriz con *hilos*

La transpuesta de una matriz **A** de *nxn* $\mathbf{A} = \left(a_{ij}\right)_{nxn}$ se define como $\mathbf{A}^T = \left(\mathrm{a}_{\mathrm{ji}}\right)_{nxn}$. Por ejemplo, para una matriz de 4x4,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{3.7}$$

cuyos elementos de esta matriz transpuesta, designada por $\mathbf{A}^T$ donde el subíndice *T* designa la transpuesta, esta definida como,

$$\mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix} \tag{3.8}$$

En otras palabras, el elemento a_{ij} de $\mathbf{A}^T$ es igual al elemento a_{ji} de la matriz **A** original. Esta operación de transposición requiere de n^2 operaciones, en donde cada operación representa $a_{ij} = a_{ji}$.

Para llevar a cabo la implementación en paralelo de la transpuesta de la matriz **A** se propone que se dividan en diferentes grupos de columnas, en donde cada procesador se encarga de los intercambios relacionados con un grupo de columnas de la matriz **A** a un grupo de renglones de la matriz $\mathbf{A}^T$. La Figura 3.14 muestra el esquema de partición implementado en este trabajo para realizar la transpuesta de la matriz **A** intercambiando las columnas por renglones. Esta Figura presenta el ejemplo de determinar un renglón completo de la matriz resultante $\mathbf{A^T}$ usando uno de los procesadores disponibles. Como se puede observar, la determinación de los *n* elementos que integran el renglón lo realiza un solo procesador mediante el intercambio de los elementos de la columna de **A**.

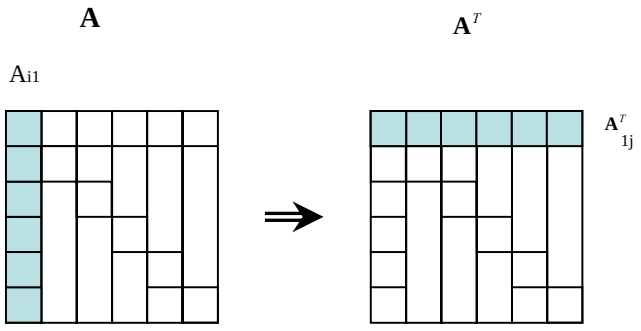

Figura 3.14 Esquema de partición para realizar la matriz transpuesta.

3.11 Implementación del Programa en C++

En seguida se muestra en la Figura 3.15 un listado simplificado del programa codificado en C++ para implementar la inversión de una matriz con *hilos*. Como se puede observar en este listado existen varias líneas de programa que son clave para la compilación y buen funcionamiento de dicho programa, entre las cuales se puede resaltar las siguientes:

1.- Línea 3. En esta línea se incluye la librería de *hilos*.

2.- Línea 9. En esta línea se define el número de *hilos*.

3.- Líneas 11-16. En estas líneas se declaran las variables que van a utilizar los *hilos* en un bloque llamado *trabajo* para tener flexibilidad en el manejo de estas variables. En donde:

candado variable asociada al método de sincronización *mutex lock.*

ini_hilo variable para indicar el inicio de la operación con *hilos*.

fin_hilo variable de trabajo para indicar el final de la operación con *hilos*.

mL se asigna una localidad de memoria asociada a la matriz **L**.

mU se asigna una localidad de memoria asociada a la matriz **U**.

mAinv se asigna una localidad de memoria asociada a la matriz **Ainv**.

col variable asociada a la columna con la que se va a trabajar.

trab_xhacer variable asociada a trabajo por hacer.

trab_nrealzd variable asociada a trabajo por no realizado.

crear_hilos variable asociada para la creación de los *hilos*.

4.- Línea 17. Variable asociada a la sincronización que se va a utilizar con los *hilos*, del tipo *mutex lock* a través de la variable *mul_candado*.

5.- Línea 19. Función de la descomposición de la matriz **A** en **L** y **U**.

6.- Línea 20. Función de la inversa de la matriz **A** con *hilos*, Esta función trabaja en conjunto con la función *trabajador1* línea 21.

7.- Línea 22. Función que crea una matriz (*n*x*n*).

8.- Línea 23. Función que crea el vector (*n*x*1*).

9.- Líneas 25 y 26. Variables globales que son la matriz **A** y el vector **b**.

10.- Línea 28. Declaración de la variable *tid* que identifica a cada uno de los *hilos*.

11.- Líneas 29-35. Es la función *main()* en la cual se llaman a las funciones para realizar las tareas necesarias y se declaran e inicializan las variables que se utilizarán.

12.- Líneas 37-39. Función de la descomposición LU en forma secuencial.

13.- Líneas 41-61. Es la función que calcula la inversa de la matriz **A** con *hilos*. Primeramente, en las líneas 47-50 se crean los *hilos* que se utilizarán con la función *pthread_create()*, la cual ya tiene definido sus argumentos que principalmente son el identificador *tid* y la función *trabajador1* para el cálculo por columnas de $\mathbf{A}^{-1}$. En la línea 55 se igualan las variables *trab_nrealzd* y *trab_xhacer* al número *n*, para sincronizar las tareas que se tengan que hacer. En la línea 56 la función *broadcast* habilita a los *hilos* para que estos comiencen a trabajar. En las líneas 57 a 60 la función *cond_wait* espera hasta que se realiza la última tarea y termina el programa.

14.- Líneas 72-75. La línea 72 decrementa el número de tareas a realizar, se cargan los datos de las matrices **L** y **U** en variables locales *mL* y *mU* en la línea 73, se carga en la variable *col* en número de columna sobre la cual se va a trabajar en la línea 74 y, finalmente en la línea 75 se actualiza la variable *trabajo.col* para que el siguiente *hilo* no realice las misma tarea y prosiga al cálculo de la siguiente columna.

15.- Líneas 76-79. Una vez que se determina la columna sobre la cual se va a trabajar, el *hilo* realiza su tarea y al terminar actualiza el trabajo realizado (línea 79).

16.- Las líneas 80-85. Si ya no existen tareas por hacer la función *pthread_cond_signal* indica que ese *hilo* ya término.

```
1   /* Este programa hace la descomposición de la matriz A en L y U, y   */
2   /*  calcula la inversa de la matriz A para  matrices no-singulares.      */
3   #include <pthread.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <math.h>
7
8   #define n 500
9   #define NCPU 1
```

Figura 3.15 Listado simplificado del programa en C++ (cont.).

```
10
11  struct {
12  pthread_mutex_t candado;
13  pthread_cond_t ini_hilo, fin_hilo;
14  double (*mL)[n][n],(*mU)[n][n],(*mAinv)[n][n];
15  int col, trab_xhacer, trab_nrealzd, crear_hilos ;
16  } trabajo;
17  pthread_mutex_t mul_candado;
18  //                    F   U   N   C   I   O   N   E   S
19  void des_LU(double (*mL)[n][n],double (*mU)[n][n]);
20  void Inversa(double (*mL)[n][n],double (*mU)[n][n],double (*mAinv)[n][n]);
21  void *trabajador1(void *);
22  void Crea_mat();
23  void Crea_Vb();
24
25  double mA1[n][n];                  //  Variables globales
26  double V_b1[n];
27
28  pthread_t tid[NCPU];               // Variable que identifica a los hilos.
29  main(){
30    Crea_mat();                      // Crea la matriz A de (nxn)
31    Crea_Vb();                       //  Crea el vector b de (nx1)
32    des_LU(&mL1,&mU1);               //  Descomposición de la matriz  A en L y U
33    Inversa(&mL1,&mU1,&mAinv1);      //  Matriz inversa de A
34  return 0;
35  }
36
37  void des_LU(double (*mL)[n][n],double (*mU)[n][n]){
38      ⋮
39  }
```

Figura 3.15 Listado simplificado del programa en C++ (cont.).

```
40
41  void Inversa(double (*mL)[n][n],double (*mU)[n][n],double (*mAinv)[n][n]){
42  int i;
43
44  pthread_mutex_lock(&mul_candado);
45  pthread_mutex_lock(&trabajo.candado);
46
47    if (trabajo.crear_hilos == 0) {
48       for (i = 0; i < NCPU; i++) {
49         pthread_create(&tid[i], NULL, trabajador1, (void *)NULL);
50       }
51    trabajo.crear_hilos = NCPU;
52    }
53    trabajo.mL=mL; trabajo.mU=mU; trabajo.mAinv=mAinv;
54    trabajo.col = 0;
55    trabajo.trab_xhacer = trabajo.trab_nrealzd = n;
56    pthread_cond_broadcast(&trabajo.ini_hilo);
57  while (trabajo.trab_nrealzd)
58    pthread_cond_wait(&trabajo.fin_hilo, &trabajo.candado);
59    pthread_mutex_unlock(&trabajo.candado);
60    pthread_mutex_unlock(&mul_candado);
61  }
62
63  void *trabajador1(void *){
64  double (*mL)[n][n],(*mU)[n][n],(*mAinv)[n][n];
65  double Vb[n],Vx[n];
66  int col,i,j,k,t,s;
67
68  while (1) {
```

Figura 3.15 Listado simplificado del programa en C++ (cont.).

```
69      pthread_mutex_lock(&trabajo.candado);
70    while (trabajo.trab_xhacer == 0)
71      pthread_cond_wait(&trabajo.ini_hilo, &trabajo.candado);
72    trabajo.trab_xhacer--;
73    mL=trabajo.mL; mU=trabajo.mU; mAinv=trabajo.mAinv;
74    col = trabajo.col;
75    trabajo.col++;
76      pthread_mutex_unlock(&trabajo.candado);
77                    ⋮
78      pthread_mutex_lock(&trabajo.candado);
79      trabajo.trab_nrealzd--;
80      if (trabajo.trab_nrealzd == 0)
81          pthread_cond_signal(&trabajo.fin_hilo);
82      pthread_mutex_unlock(&trabajo.candado);
83    }
84   return (NULL);
85   }
```

Figura 3.15 Listado simplificado del programa en C++ (cont.).

3.12 Conclusiones

En este capítulo se analizaron las arquitecturas de computadoras y las clasificaciones existentes para éstas. Además, se presentaron tres tipos de programación en paralelo, como son el *PVM*, *MPI* y *hilos*. Se presenta el procedimiento para implementar la programación con *hilos*, realzando los aspectos claves en este tipo de programación como la creación de los *hilos* y sincronización. El método de sincronización detallada en esta tesis es basado en candados de exclusión mutua. Se presentó la implementación de la inversa de una matriz **A** de dimensiones *n*x*n* con este tipo de programación con *hilos*. Utilizando los candados *mutex lock* en conjunto con las variables de control y la rutina de sincronización hecha para asignar a cada *hilo* una tarea. El cálculo de la matriz inversa se realiza columna por columna, es decir, a cada *hilo* se le asigna la tarea de calcular una columna de la matriz

inversa y actualiza las variables de control. Una vez que los *hilos* terminan sus tareas almacenan el resultado en una localidad compartida asociada a la matriz inversa.

El proceso de paralelizacion con *hilos* presentado en este capítulo esta asociada exclusivamente al proceso repetitivo de las tareas de solución de los sistemas de ecuaciones **Ld**=**b** y **Ux**=**d**. En este trabajo no se paralelizó el proceso de descomposición LU de la matriz **A**.

Capítulo 4

Casos de Estudio

En este capítulo se presentan varios casos de estudio en donde se muestra la aplicación de la programación en paralelo con *hilos* para solucionar problemas de grandes dimensiones que requieren el uso de operaciones matriciales. Los casos de estudio que se presentan involucran la aplicación de la matriz inversa y la multiplicación matricial en la solución de un sistema **Ax=b** de grandes dimensiones. Todas las simulaciones reportadas en este trabajo se realizaron en una computadora HP Proliant ML 350 con dos procesadores Xeon, con una velocidad de 3.0 GHz y con una memoria compartida de 1Gb.

4.1 Solución del sistema de Ax=b en problemas diversos

A continuación se presentan dos ejemplos asociados a problemas de probabilidad-estadística y sistemas de control de pequeñas dimensiones en los cuales se requieren implementar la solución de un sistema de ecuaciones algebraicas **Ax**=**b**.

Ejemplo 1.- En química analítica, los análisis de flourescencia con rayos X son una herramienta para calcular porcentajes de ingredientes en mezclas multicomponentes. En muchas ocasiones, la estimación de las concentraciones depende en gran medida de la habilidad del usuario para ajustar modelos de regresión adecuados. Considérese las concentraciones de los ingredientes en las mezclas para producir estándares de calibración que se muestran a continuación en la Tabla 4.1 [14].

Tabla 4.1 Resultados de mezclas hechas en el laboratorio.

i	y	x_1	x_2	x_3	x_4
1	0.5514	1.1240	0.8980	0.8219	0.9906
2	0.4426	0.9285	0.8872	0.9308	0.9944
3	0.5631	1.1214	0.8030	0.7668	1.1221
4	0.5624	1.1635	0.8706	0.9272	0.9832
5	0.4505	0.9415	0.8064	0.9026	1.1127
6	0.5290	1.0712	0.8404	0.8662	1.0836
7	0.4702	0.9561	0.8731	0.8206	1.0290
8	0.5001	1.0186	0.8431	0.8346	1.0591
9	0.4425	0.9039	0.8314	0.7596	1.0994

La respuesta y_i es la concentración medida de un ingrediente A. El valor medido x_1 es la razón de intensidad de los rayos X asociada con el ingrediente A y los valores x_2, x_3, y x_4 son las razones de intensidad de los componentes adicionales de la mezcla. Con estos datos se plantea la solución de un modelo de regresión lineal múltiple de la forma,

$$\mu_Y \mid x_1, x_2, x_3, x_4 = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 \tag{4.1}$$

Con los datos proporcionados en la Tabla 4.1 y con $n = 9$, se sustituye en las ecuaciones normales relacionadas con el modelo de regresión lineal [14] y se obtiene el siguiente sistema de ecuaciones,

$$\begin{bmatrix} 9 & 9.2287 & 7.6532 & 7.6303 & 9.4741 \\ 9.2287 & 9.5394 & 7.8510 & 7.8257 & 9.7037 \\ 7.6532 & 7.8510 & 6.5172 & 6.4943 & 8.0421 \\ 7.6303 & 7.8257 & 6.4943 & 6.5015 & 8.0182 \\ 9.4741 & 9.7037 & 8.0421 & 8.0182 & 9.9974 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \end{bmatrix} = \begin{bmatrix} 4.5118 \\ 4.6663 \\ 3.8375 \\ 3.8226 \\ 4.7456 \end{bmatrix} \tag{4.2}$$

Descomponiendo la matriz **A** en **L** y **U**,

Matriz **L**

****

1.0000 0.0000 0.0000 0.0000 0.0000

1.0254 1.0000 0.0000 0.0000 0.0000

0.8504 0.0436 1.0000 0.0000 0.0000

0.8478 0.0198 0.6327 1.0000 0.0000

1.0527 -0.1463 -1.5106 -0.1778 1.0000

****

Matriz **U**

****

9.0000 9.2287 7.6532 7.6303 9.4741

0.0000 0.0762 0.0033 0.0015 -0.0111

0.0000 -0.0000 0.0091 0.0058 -0.0138

0.0000 -0.0000 0.0000 0.0288 -0.0051

-0.0000 0.0000 0.0000 -0.0000 0.0009

****

y haciendo SAD y SAT para encontrar la solución del sistema **Ax**=**b** se obtiene,

Sustitución hacia adelante

****

d_0 = 4.5118

d_1 = 0.0399

d_2 = -0.0009

d_3 = -0.0028

d_4 = 0.0001

****

Sustitución hacia atrás

****

β_0 = -0.3307

β_1 = 0.5398

β_2 = 0.1935

β_3 = -0.0680

β_4 = 0.1630

****

A manera de comprobación, la solución del sistema de ecuaciones (4.2) usando MATLAB es,

```
>> x=inv(A)*b
x =
```

$$\beta = \begin{bmatrix} -0.3307 \\ 0.5398 \\ 0.1935 \\ -0.0680 \\ 0.1630 \end{bmatrix}$$

de dónde se puede observar que los resultados son iguales.

La solución de este conjunto de ecuaciones proporciona los coeficientes *β's* definidos en el modelo de regresión lineal. Por lo tanto, la ecuación de regresión queda,

$$\hat{y} = (-0.3307) + (0.5398)x_1 + (0.1935)x_2 + (-0.0680)x_3 + (0.1630)x_4 \qquad (4.3)$$

Para una mezcla específica de rayos X con x_1 = 1.091, x_2 = 0.855, x_3 = 0.758, y x_4 = 1.005, se tiene una concentración estimada de,

$$y = -0.3307 + (0.5398)(1.091) + (0.1935)(0.855) + (-0.0680)(0.758) + (0.1630)(1.005)$$

$$y = 0.5359$$

Ejemplo 2.- Sea la planta [15],

$$G_p(s) = \frac{N_p(s)}{D_p(s)} = \frac{1}{s^3 + s} \tag{4.4}$$

a la cuál se le desea incorporar un controlador $G_c(s)$ definido como,

$$D_c(s) = s^2 + d_1 s + d_0 \tag{4.5}$$

$$N_c(s) = c_2 s^2 + c_1 s + c_0 \tag{4.6}$$

La determinación de los coeficientes d_1, d_0, c_2, c_1 y c_0, del controlador se realiza por medio de la siguiente ecuación Diophantine,

$$D_p(s)D_c(s) + N_c(s)N_p(s) = D_d(s) \tag{4.7}$$

$$\begin{aligned} D_p(s)D_c(s) + N_p(s)N_c(s) &= s^5 + d_1 s^4 + (1 - d_0)s^3 + d_1 s^2 + d_0 s + c_2 s^2 + c_1 s + c_0 \\ &= s^5 + d_1 s^4 + (1 - d_0)s^3 + (d_1 + c_2)s^2 + (d_0 + c_1)s + c_0 \end{aligned} \tag{4.8}$$

Suponiendo que se quieren ubicar los polos en lazo cerrado en $s = -3 + j3$, $s = -3 - j3$, $s = -5$, $s = -5$ y $s = -10$.

$$D_p(s) = (s + 3 - j3)(s + 3 + j3)(s + 5)(s + 5)(s + 10) \tag{4.9}$$

$$= s^5 + 26s^4 + 263s^3 + 1360s^2 + 3750s + 4500 \tag{4.10}$$

Por lo tanto, igualando la ecuación (4.8) y (4.10) queda en forma matricial,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d_2 \\ d_1 \\ d_0 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 26 \\ 263 \\ 1360 \\ 3750 \\ 4500 \end{bmatrix} \tag{4.11}$$

en donde la matriz de la izquierda se conoce como *Silvester*. La solución de este sistema de ecuaciones es la siguiente:

Matriz **L**

****

1.0000 0.0000 0.0000 0.0000 0.0000 0.0000

0.0000 1.0000 0.0000 0.0000 0.0000 0.0000

1.0000 0.0000 1.0000 0.0000 0.0000 0.0000

0.0000 1.0000 0.0000 1.0000 0.0000 0.0000

0.0000 0.0000 1.0000 0.0000 1.0000 0.0000

0.0000 0.0000 0.0000 0.0000 0.0000 1.0000

****

Matriz **U**

****

1.0000 0.0000 0.0000 0.0000 0.0000 0.0000

0.0000 1.0000 0.0000 0.0000 0.0000 0.0000

0.0000 0.0000 1.0000 0.0000 0.0000 0.0000

0.0000 0.0000 0.0000 1.0000 0.0000 0.0000

0.0000 0.0000 0.0000 0.0000 1.0000 0.0000

0.0000 0.0000 0.0000 0.0000 0.0000 1.0000

****

Ahora haciendo SAD y SAT para encontrar la solución del sistema **Ax**=**b**,

Sustitución hacia adelante

****

d_0 = 1.0000

d_1 = 26.0000

d_2 = 262.0000

d_3 = 1334.0000

d_4 = 3488.0000

d_5 = 4500.0000

****

Sustitución hacia atrás

****

d_2 = 1.0000

d_1 = 26.0000

d_0 = 262.0000

c_2 = 1334.0000

c_1 = 3488.0000

c_0 = 4500.0000

****

La solución del sistema de (4.11) por medio de MATLAB es,

```
>> x=inv(A)*b
x =
```

$$x = \begin{bmatrix} 1 \\ 26 \\ 262 \\ 1334 \\ 3488 \\ 4500 \end{bmatrix}$$

lo cual demuestra que se obtienen los mismos resultados con la descomposición LU.

Finalmente, los coeficientes calculados definen el siguiente controlador,

$$\frac{N_c(s)}{D_c(s)} = \frac{1334s^2 + 3488s + 4500}{s^2 + 26s + 262} \qquad (4.12)$$

Ejemplo 3.- Considere una red cuya solución en términos de los voltajes nodales esta definida por [16],

$$\left(\mathbf{A}\mathbf{Y}_b\mathbf{A}^T\right)\mathbf{v}_n = \mathbf{A}\left(\mathbf{j} - \mathbf{Y}_b\mathbf{e}\right) \tag{4.13}$$

ó bien,

$$\mathbf{Y}_n\mathbf{v}_n = \mathbf{j}_n \tag{4.14}$$

en donde **A** es la matriz de conectividad, $\mathbf{Y}_b$ es la matriz de admitancias, **j** es el vector de fuentes de corriente, **e** es el vector de fuentes de voltaje y $\mathbf{v}_n$ es el vector de voltajes nodales.

Asumiendo que,

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & -1 \end{bmatrix}$$

$$\mathbf{Y}_b = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}$$

$$\mathbf{j} = \begin{bmatrix} 2 \\ 7 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} 5 \\ 0 \\ 0 \\ 0 \\ 0 \\ -6 \\ 0 \end{bmatrix}$$

Entonces la matriz de admitancia nodal $\mathbf{Y}_n$ y el vector de fuentes de corriente $\mathbf{j}_n$ tienen los siguientes valores. La matriz transpuesta queda,

Transpuestas de la matriz **A**

****

1.0000 0.0000 0.0000

0.0000 1.0000 0.0000

0.0000 0.0000 1.0000

1.0000 -1.0000 0.0000

1.0000 -1.0000 0.0000

0.0000 1.0000 -1.0000

1.0000 0.0000 -1.0000

****

La multiplicación de $\mathbf{Y}_b\mathbf{A}^T$

Multiplicación de la matriz **Y***At

****

2.0000 0.0000 0.0000

0.0000 5.0000 0.0000

0.0000 4.0000 -4.0000

0.0000 8.0000 -8.0000

10.0000 -10.0000 0.0000

0.0000 3.0000 -3.0000

0.5000 0.0000 -0.50000

****

Obteniendo ahora la multiplicación matricial $\mathbf{A}\mathbf{Y}_b\mathbf{A}^T$

Multiplicación de la matriz **A***Y***A**t

****

12.5000 -2.0000 -8.5.0000

-10.0000 10.0000 5.0000

-0.5.0000 1.0000 -0.5.0000

****

$$\mathbf{Y}_n = \mathbf{A}\mathbf{Y}_b\mathbf{A}^T = \begin{bmatrix} 12.5 & -2 & -8.5 \\ -10 & 10 & 5 \\ -0.5 & 1 & -0.5 \end{bmatrix}$$

Ahora obteniendo la multiplicación de $\mathbf{Y}_b\mathbf{e}$

Multiplicación de **Y**b***e**

****

10.0000

0.0000

-24.0000

-48.0000

0.0000

-18.0000

0.0000

****

Ahora restamos $\mathbf{j} - \mathbf{Y}_b\mathbf{e}$, y después multiplicamos por la matriz **A**,

Resta de **j**-**Y**b***e**

****

-8.0000

7.0000

23.0000

48.0000

0.0000

18.0000

0.0000

****

Multiplicación de **A**(**j**-**Y**b***e**)

****

40.0000

-23.0000

5.0000

****

$$\mathbf{j}_n = \mathbf{A}\left(\mathbf{j} - \mathbf{Y}_b\mathbf{e}\right) = \begin{bmatrix} 40 \\ -23 \\ 5 \end{bmatrix}$$

Ahora, solucionando el sistema de ecuaciones mediante descomposición de Crout queda,

Impresión de la matriz **L**

****

1.0000 0.0000 0.0000

-0.8000 1.0000 0.0000

-0.0400 0.1095 1.0000

****

Impresión de la matriz **U**

****

12.5000 -2.0000 -8.5000

0.0000 8.4000 -1.8000

0.0000 0.0000 -0.6429

****

Sustitución hacia adelante y sustitución hacia atrás

$\mathbf{d}_0 = 40.0000$

$\mathbf{d}_1 = 9.0000$

$\mathbf{d}_2 = 5.6143$

$\mathbf{x}_0 = -2.8667$

$\mathbf{x}_1 = -0.8000$

$\mathbf{x}_2 = -8.7333$

por lo tanto, la solución del sistema de ecuaciones descrito por a expresión (4.14) es,

$$\mathbf{v}_n = \begin{bmatrix} -2.8667 \\ -0.8000 \\ -8.7733 \end{bmatrix}$$

4.2 Cálculo de la Inversa de una Matriz A^{-1} de Grandes Dimensiones con *Hilos*

Para realizar este caso de estudio se requiere una función que genere una matriz **A** no-singular de grandes dimensiones. Por lo tanto, en este trabajo esta matriz se genera automáticamente con la función *random* para crear una matriz arbitraria no-singular. A continuación se muestra en la Figura 4.1 el diagrama de flujo para dicha función *ramdom*.

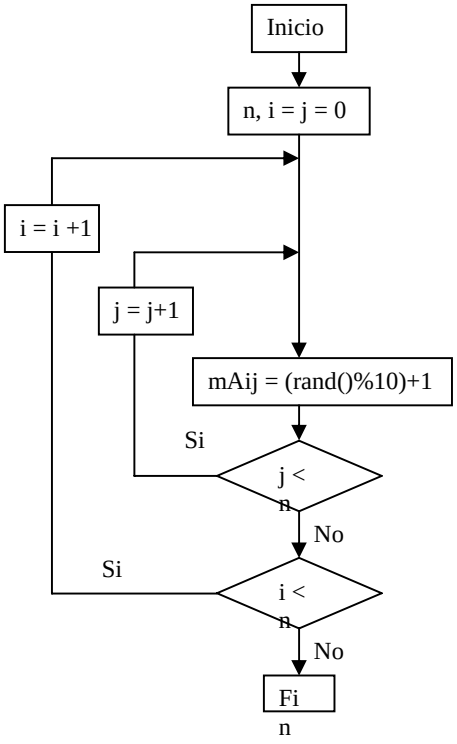

Figura 4.1 Creación de la matriz **A** de dimensiones *n*x*n*.

Como se puede observar en la Figura 4.1 se crea una matriz **A** con la función (rand()%10)+1), en donde toma valores entre 1 y 9. Después de crear la matriz de dimensiones *n*x*n* se hace la descomposición de **A** en **L** y **U**. Teniendo estas dos funciones en nuestro programa se puede agregar el método de la inversión de la matriz *n*x*n* programado con *hilos* como se muestra en la Figura 3.10.

Como se observa en el diagrama de flujo presentado en la Figura 3.10, primeramente se inicializan las variables globales que básicamente son *n* y **NCPU,** las cuales representan las dimensiones del arreglo matricial y el número de *hilos*, respectivamente. Posteriormente se genera la matriz **A**(*n*,*n*) y se llama a la rutina de descomposición LU, la cual descompone la matriz **A** en **L** y **U** para posteriormente utilizarlas en la rutina de la inversa. Haciendo uso del método de sincronización con candados *mutex lock* se le asigna a cada *hilo* la tarea de calcular una columna de la matriz inversa con la rutina implementada para este fin. Una vez que se han asignado las tareas a cada uno de los *hilos*, entonces los cálculos se realizan en paralelo.

A continuación se muestra en la Tabla 4.2 los resultados obtenidos para el cálculo de la matriz inversa $\mathbf{A}^{-1}$ considerando matrices de diferentes dimensiones. La segunda columna de la Tabla 4.2 muestra el tiempo requerido para realizar la descomposición LU, la cual se realiza de forma secuencial y en este trabajo no se trato de paralelizar. Por ejemplo, se puede observar que para una matriz de dimensiones de 3000x3000 se requieren 186.627 segundos para realizar la descomposición. La tercera y cuarta columna de la Tabla 4.2 muestran los tiempos de cómputo necesarios para realizar la inversión de la matriz cuando se especifican 1 ó 2 *hilos*, respectivamente.

Tabla 4.2 Tiempos de resultados obtenidos de $\mathbf{A}^{-1}$.

N	Descomposición **A**=LU(seg.)	Matriz $\mathbf{A}^{-1}$ 1 *Hilo*(seg.)	Matriz $\mathbf{A}^{-1}$ 2 *Hilos*(seg.)
3000	186.627	789.758	538.561
2500	124.256	427.713	291.450
2000	51.637	166.581	113.265
1500	23.151	75.291	49.952
1000	6.676	19.975	13.616
500	0.878	2.832	1.893

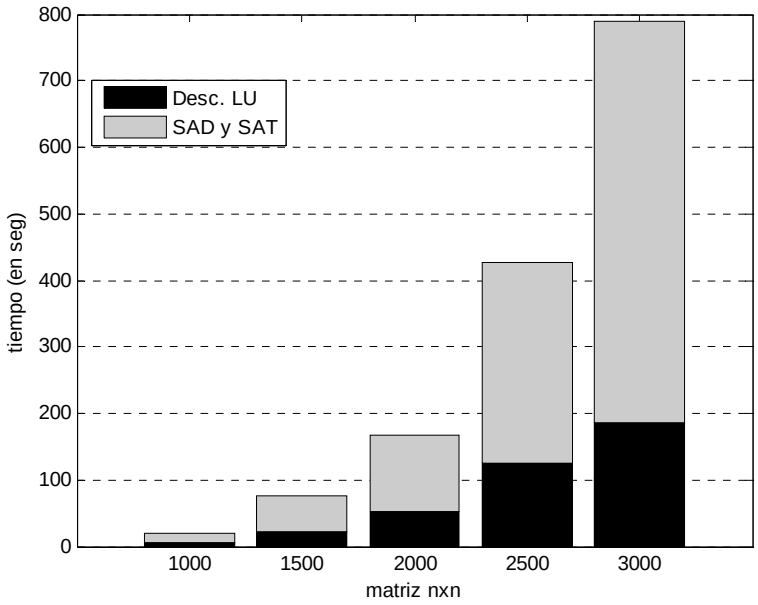

Figura 4.2 Tiempos de cómputo requeridos para determinar $\mathbf{A}^{-1}$ con 1 *hilo*.

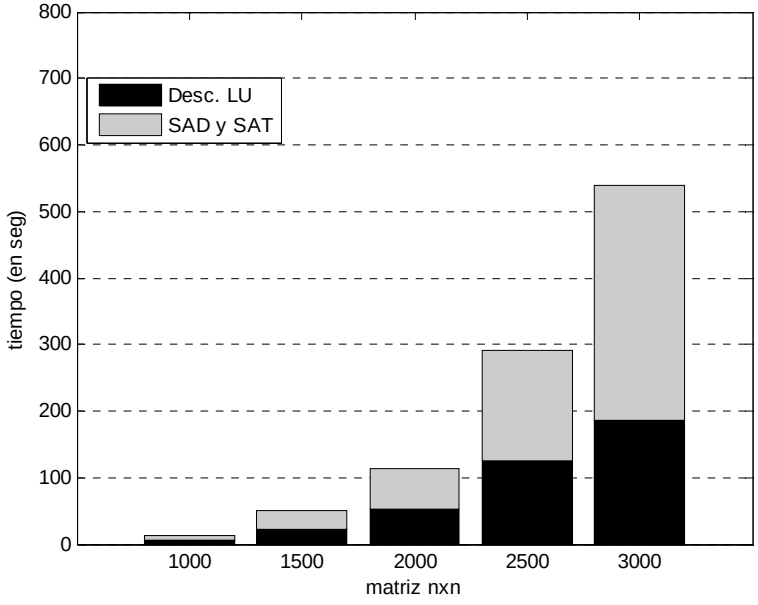

Figura 4.3 Tiempos de cómputo requeridos para determinar $\mathbf{A}^{-1}$ con 2 *hilos*.

En la Figura 4.2 y 4.3 se muestran los tiempos demandados para determinar la matriz inversa cuando se especifican 1 ó 2 *hilos*, respectivamente. En estas dos figuras la parte de la barra en color negro representa el conjunto de operaciones realizadas secuencialmente, asociadas con la descomposición LU, mientras que la parte de la barra de tono gris esta relacionada con sustitución hacia adelante (SAD) y sustitución hacia atrás (SAT), las cuales representan el conjunto de operaciones que se realizan en paralelo. Como se observa en estas figuras, la reducción en tiempo que se obtiene al invertir la matriz de 3000x3000 es la diferencia de tiempo que tarda en invertir la matriz con 1 *hilo* (789.758 segundos) y con 2 *hilos* (538.561 segundos), lo cual representa aproximadamente una tercera parte menor de tiempo con dos *hilos* respecto a un solo *hilo*.

La Figura 4.4 muestra como se comporta el factor de aceleración para el proceso de inversión de la matriz, el cual se calcula con el cociente de los tiempos obtenidos con uno y dos *hilos*, es decir,

$$\text{S} = \text{Tiempo con 1 } \textit{hilo} \ / \ \text{Tiempo con 2 } \textit{hilos}. \tag{4.15}$$

Se puede apreciar de la Figura 4.4 que el factor de aceleración se encuentra alrededor de 1.5 para cualquier dimensión de la matriz **A**.

El hecho de que los factores de aceleración no alcanzan magnitudes mayores a 1.5 se explica de la siguiente manera. Debido a que el proceso de inversión de la matriz **A** involucra básicamente tres tareas principales, las cuales son la descomposición LU, sustitución hacia delante y sustitución hacia atrás, entonces si algunas de estas tareas no se paraleliza el factor de aceleración no podrá ser 2 para el caso que se utilicen 2 procesadores. Es decir, la complejidad (O) de la tarea asociada a la descomposición LU es [7],

$$O = \frac{1}{3} N^3 \tag{4.16}$$

Para la tarea definida como sustitución hacia delante (SAD) tiene la complejidad,

$$O = \frac{1}{2} N^3 \tag{4.17}$$

Por su parte para la sustitución hacia atrás (SAT) el número de operaciones necesarias es,

$$O = \frac{1}{6} N^3 \tag{4.18}$$

Por lo tanto, la complejidad asociada a la inversión de la matriz **A** es,

$$O\left(\frac{1}{3}N^3 + \frac{1}{2}N^3 + \frac{1}{6}N^3\right) = O\left(N^3\right) \tag{4.19}$$

de donde se puede observar que la parte del proceso para calcular $\mathbf{A}^{-1}$ que se realiza de forma secuencial requiere de un número de operaciones igual a $\frac{1}{3}N^3$.

Ahora bien, de la ley de Amdahl se determina el factor de aceleración máximo que se podría obtener utilizando **p** procesadores. Para el caso planteado en esta tesis en donde se utilizo una computadora con 2 procesadores, se tiene,

$$S = \frac{p}{B_1 S + (1 - B_1)} \tag{4.20}$$

$$S = \frac{2}{\left(\frac{1}{3}\right)(2) + \left(\frac{2}{3}\right)}$$

$$S = 1.5038$$

Dicho resultado corrobora que el factor de aceleración máximo que pudiéramos obtener es 1.5038, la cual coincide con los resultados reportados en la Figura 4.4.

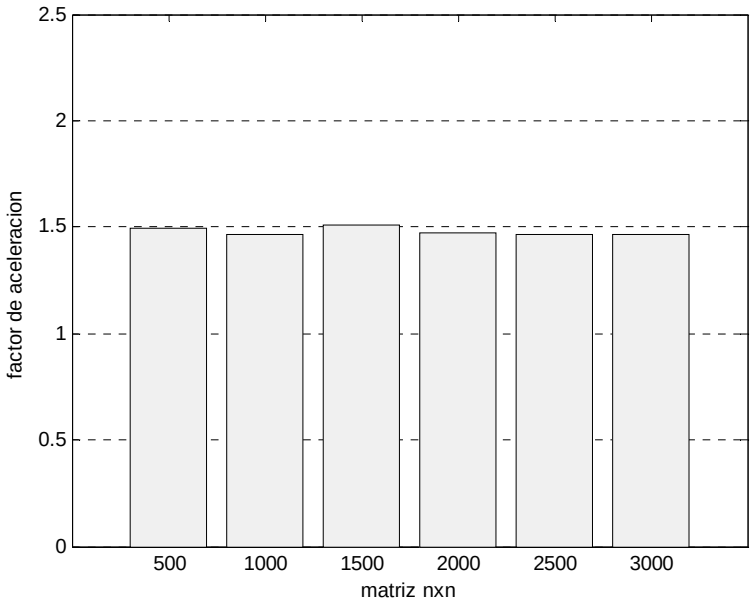

Figura 4.4 Factor de aceleración para el cálculo de $\mathbf{A}^{-1}$ con 2 procesadores.

4.3 Cálculos de la Multiplicación de dos Matrices con *Hilos*

El algoritmo secuencial para la multiplicación de dos matrices **A** y **B** de dimensiones *n*x*n* requiere n^3 operaciones de multiplicación y suma, es decir, la complejidad de la ejecución es $O(n^3)$. Por lo tanto, para los resultados presentados en esta tesis con una computadora con dos procesadores el tiempo total de ejecución esta asociado a $O = \left(\frac{n^3}{2} \right)$.

A continuación se muestra en la Figura 4.5 los resultados obtenidos para el cálculo de la multiplicación de las matrices **A**x**B** considerando matrices de dimensiones de 500x500 a 3000x3000.

Esta figura muestra como se comporta el factor de aceleración para el proceso de la multiplicación matricial **C**=**A**x**B**. Como se puede apreciar, los factores de aceleración obtenidos se encuentran cerca de 2 el cual representa el máximo factor esperado en esta operación matricial ya que las simulaciones se realizaron en dos procesadores.

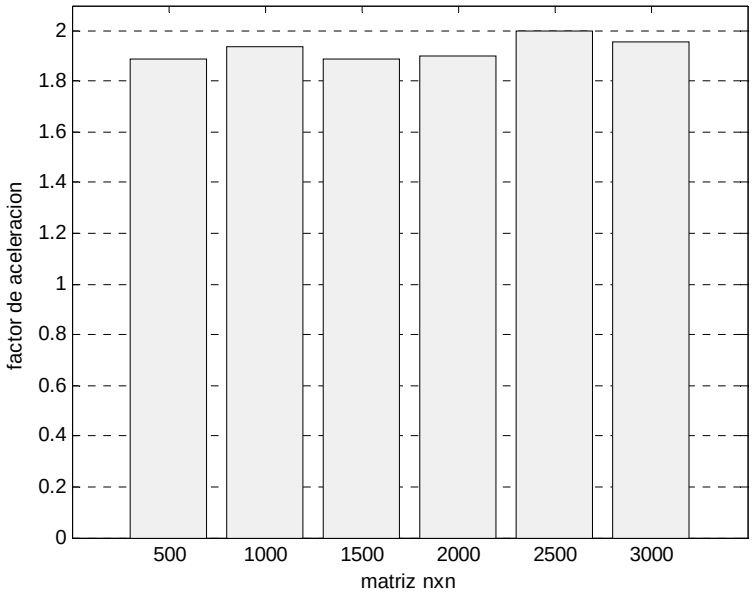

Figura 4.5 Factor de aceleración del producto matricial **A**x**B**.

4.4 Cálculos de la Suma de dos Matrices con *Hilos*

A continuación se resumen los resultados obtenidos para el cálculo de la suma de dos matrices con diferentes dimensiones, que van desde 5000x5000 hasta 10000x10000. La Figura 4.6 muestra como se comporta el factor de aceleración para el proceso de suma de la matriz **C**=**A**+**B**. Se puede apreciar que el factor de aceleración se encuentra alrededor de 2 para cualquier dimensión de la matriz **A**.

El algoritmo secuencial para sumar/restar dos matrices tiene una complejidad de $O(n^2)$, en donde n representa una operación de adición/sustracción. Es por ello que para una computadora con 2 procesadores el tiempo de ejecución esta asociado a $O=\left(\frac{n^2}{2}\right)$. Además, si se toma en cuenta que la tarea de sumar/restar dos matrices es 100% paralelizable, entonces el factor de aceleración máximo esperado es de 2. Esto se corrobora en los resultados reportados en la Figura 4.6.

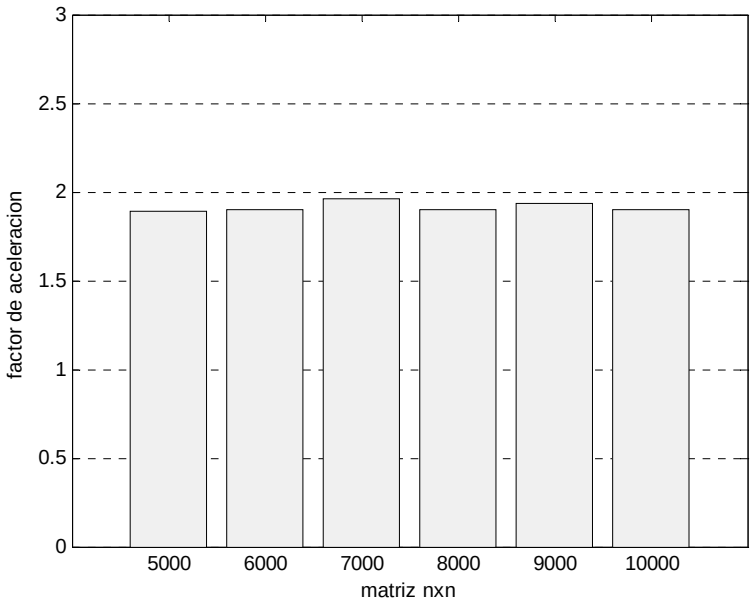

Figura 4.6 Factor de aceleración de la suma **A**+**B**.

4.5 Cálculo de la Transpuesta de una Matriz con *Hilos*

A continuación se muestran los resultados obtenidos para el cálculo de la transpuesta de la matriz **A** considerando matrices de dimensiones 5000x5000 a 10000x10000. La Figura 4.7 resume los resultados en términos del factor de aceleración para el cálculo de la transpuesta de la matriz **A**.

De manera similar a las operaciones de suma/resta, esta operación de transposición es 100% paralelizable y por lo tanto el factor de aceleración máximo esperado con una computadora de 2 procesadores es 2. Por lo tanto, los resultados reportados en la Figura 4.7 oscilan precisamente alrededor de 2.

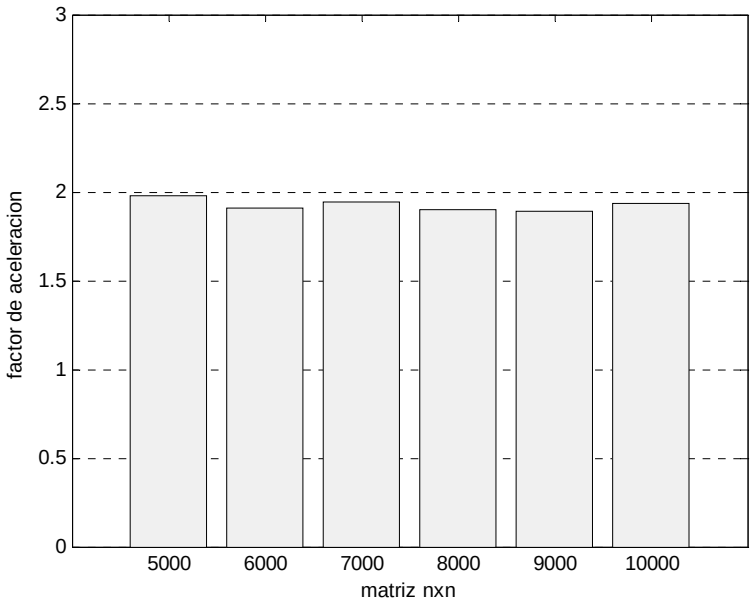

Figura 4.7 Factor de aceleración de la transpuesta de una matriz.

4.6 Solución del Sistema Ax=b de Grandes Dimensiones con *Hilos*

En esta sección se presenta la solución del sistema de ecuaciones **Ax=b** de grandes dimensiones. Se considera que se conoce de antemano $\mathbf{A}^{-1}$ y se procede simplemente multiplicar dicha matriz para el vector **b** haciendo uso de una multiplicación en paralelo con *hilos*.

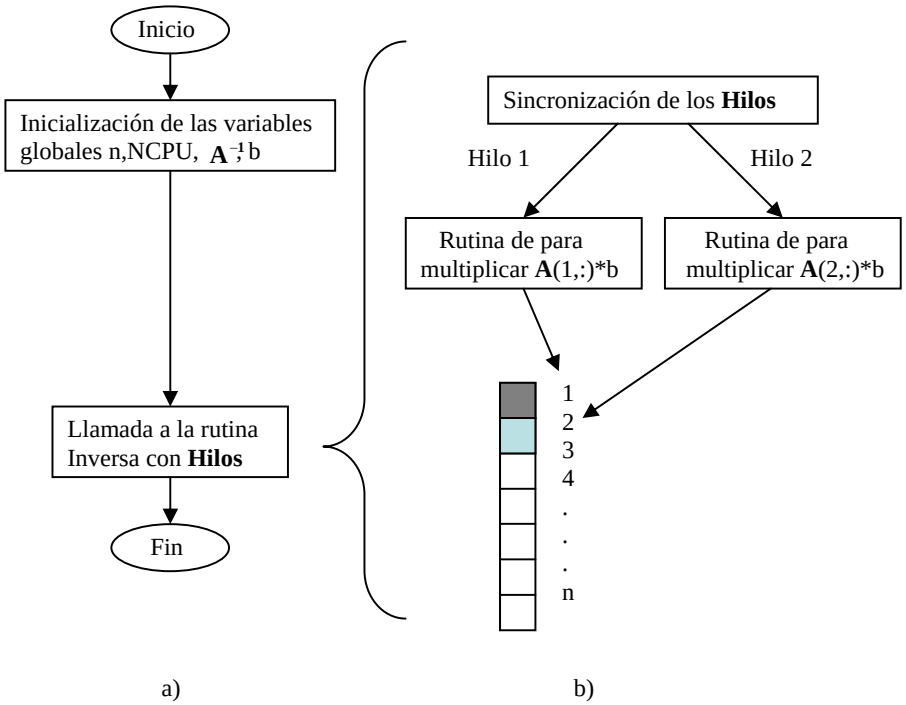

Figura 4.8 Calculo del producto $\mathbf{A}^{-1}\ \mathbf{b}$ con *hilos*.

Como se observa en la Figura 4.8a, primeramente se inicializan las variables globales que básicamente son *n* y **NCPU**. La matriz $\mathbf{A}^{-1}$ y el vector **b** se proporcionan también como datos de entrada. Utilizando la sincronización de candados *mutex lock* (Figura 4.8b) se le asigna a cada *hilo* la tarea de calcular el producto de un renglón de $\mathbf{A}^{-1}$ por el vector **b**, es decir, **A**(1,:)***b**, **A**(2,:)***b**, etc. Al final los *n* elementos calculados representan el vector resultante de la solución $\mathbf{x} = \mathbf{A}^{-1}\ \mathbf{b}$. Los tiempos de cómputo demandados para realizar la solución de **Ax**=**b** con esta alternativa se reportaron en la Tabla 4.3.

A continuación se muestran en la Tabla 4.3 los resultados obtenidos de la solución del sistema **Ax**=**b**.

Tabla 4.3 Tiempos de resultados obtenidos de $\mathbf{x} = \mathbf{A}^{-1}\ \mathbf{b}$.

N	1hilo	2*hilos*
5000	0.395	0.211
6000	0.450	0.247
7000	0.619	0.328
8000	0.819	0.440
9000	0.998	0.538
10000	1.244	0.678

La Figura 4.9 muestra como se comporta el factor de aceleración para la solución del sistema **Ax**=**b**. Tomando en cuenta que se conoce la matriz inversa denotada por $\mathbf{A}^{-1}$ y el vector **b** también. Como se puede observar los factores de aceleración que se obtienen se encuentran cercanos a 2.

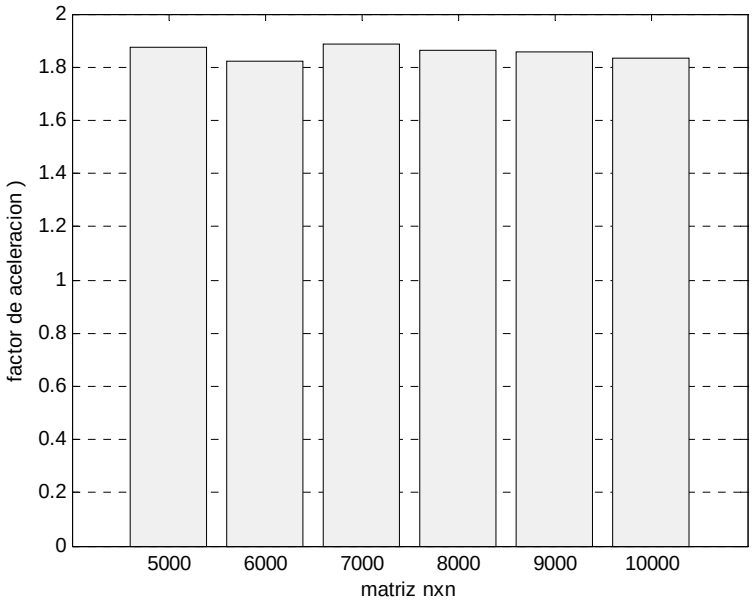

Figura 4.9 Factor de aceleración para la solución $\mathbf{x}=\mathbf{A}^{-1}\,\mathbf{b}$ con 2 *hilos*.

4.7 Conclusiones

En este capítulo se presentó la implementación del cálculo de la matriz inversa y las operaciones matriciales de multiplicación suma/resta y transposición. La solución de sistemas de ecuaciones **Ax**=**b** de grandes dimensiones que se presentó en este capitulo se realizó utilizando las operaciones matriciales en paralelo, tales como inversión y multiplicación matricial. El enfoque desarrollado en esta tesis se basa en el uso de programación con *múltiples-hilos* con el objeto de explotar los recursos de una computadora con dos procesadores y memoria compartida.

Se reportan factores de aceleración de 1.5 para la inversión matricial, los cuales coinciden con los factores esperados según la ley de Amdahl. Las operaciones matriciales de multiplicación, suma/resta y transposición presentaron factores de aceleración cercanos

a 2. Estos valores son mayores a los reportados para la inversión matricial debido a que todo el trabajo de cómputo en estas operaciones de multiplicación, suma/resta y transposición es 100% paralelizable.

La solución de sistemas de ecuaciones **Ax=b** de grandes dimensiones que se presentó en este capitulo se realizó utilizando las operaciones matriciales en paralelo, tales como inversión y multiplicación matricial.

La reducción en los tiempos de cómputo reportados en este capítulo es importante para aquellos problemas que involucran la utilización de operaciones básicas en matrices de grandes dimensiones y que operan en tiempo real. Esto es, si los tiempos de respuesta del problema que se analice son críticos para tomar una decisión, entonces la reducción en tiempo proporcionado por la programación en paralelo coadyubará para lograr el objetivo de tener una respuesta en tiempo requerido.

Capítulo 5

Conclusiones y Trabajos Futuros

En este capítulo se presentan las conclusiones de este trabajo y se indican algunas sugerencias para trabajos futuros.

5.1 Conclusiones

En este trabajo se ha presentado la implementación de las operaciones matriciales básicas bajo una plataforma de procesamiento en paralelo. Se ha propuesto una implementación basada en procesamiento en paralelo usando programación con *múltiples-hilos* para el cálculo de la matriz inversa de **A** y la implementación de las operaciones de multiplicación matricial, multiplicación de una matriz por un vector, suma/resta y transposición. En particular se desarrolló un esquema de programación con *hilos* específicamente para la tarea asociada a la sustitución hacia adelante y hacia atrás. El proceso de descomposición LU no fue paralelizado en este trabajo.

Se analizaron los alcances y limitaciones de esta metodología desde el punto de vista de los procesadores, así como también desde el punto de vista del grado de paralelizacion que se alcanzó en este trabajo.

La mayor parte de este trabajo se dedicó a la determinación de la inversa de una matriz **A** de dimensiones *n*x*n* con este tipo de programación con *hilos*. Se reportan factores de aceleración de 1.5, los cuales coinciden con los factores esperados según la ley de Amdahl. Las operaciones matriciales de multiplicación, suma/resta y transposición presentaron factores de aceleración cercanos a 2. Estos valores son mayores a los reportados para la inversión matricial debido a que todo el trabajo de cómputo en estas operaciones de multiplicación, suma/resta y transposición es 100% paralelizable.

Se utilizaron las implementaciones en paralelo de las principales operaciones matriciales como inversión y multiplicación para mostrar sus beneficios en el cálculo de la solución del problema **Ax**=**b**. Se reportaron resultados para sistemas **Ax**=**b** de pequeñas

dimensiones asociados a problemas encontrados en áreas tales como ingeniería química, probabilidad y estadística, sistemas de control y circuitos eléctricos.

Los resultados obtenidos corroboran los niveles de desempeño esperados considerando el número de procesadores y el nivel de paralelismo encontrado en las diferentes tareas. Actualmente, los avances en los equipos de cómputo han reducido los costos de los equipos con más de un procesador a niveles tales que ya no representa una desventaja importante para hacer desarrollos con este tipo de equipos.

Los programas desarrollados en esta tesis permiten la solución de matrices no-singulares. Para el caso de matrices singulares no se pueden resolver con los programas de esta tesis. Para poder solucionar estos casos especiales se requieren incorporar a estos programas algunas rutinas adicionales para la determinación del grado de condicionamiento de las matrices y técnicas de algebra lineal que permitan manipular dichas matrices singulares ó cercanas a ser singulares.

La reducción en los tiempos de cómputo reportados en este capítulo son importantes para aquellos problemas que involucran alguna operación matricial como el cálculo de una matriz inversa ó la multiplicación matricial y que operan en tiempo real. Esto es, si los tiempos de respuesta del problema que se analice son críticos para tomar una decisión, entonces la reducción en tiempo proporcionado por la programación en paralelo coadyuvará para lograr el objetivo de tener una respuesta en tiempo requerido.

5.2 Trabajos Futuros

Las propuestas de trabajos futuros basados en los resultados obtenidos en este trabajo son los siguientes:

- Aplicación de procesamiento en paralelo a la descomposición LU.
- Determinación de los factores de aceleración utilizando computadoras con más de 2 procesadores.

Apéndice A

A continuación se presenta el listado completo del programa desarrollado en C++.

```
/* Este programa hace la descomposición de la matriz A en L y U, y   */
/*  calcula la inversa de la matriz A para  matrices no-singulares.      */

#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define n 500
#define NCPU 1

struct {
  pthread_mutex_t candado;
  pthread_cond_t ini_hilo, fin_hilo;
  double (*mL)[n][n],(*mU)[n][n],(*mAinv)[n][n];
  int col, trab_xhacer, trab_nrealzd, crear_hilos ;
} trabajo;
pthread_mutex_t mul_candado;
//                                   F U N C I O N E S
void des_LU(double (*mL)[n][n],double (*mU)[n][n]);
void Inversa(double (*mL)[n][n],double (*mU)[n][n],double (*mAinv)[n][n]);
void *trabajador1(void *);
void Crea_mat();
void Crea_Vb();
//                                   V a r i a b l e s   g l o b a l e s
double mA1[n][n];
double V_b1[n];
//                                  I d e n t i f i c a c i ó n   d e   l o s   h i l o s
pthread_t tid[NCPU];

main()
{
   int i,j,k;
   double mL1[n][n],mU1[n][n],mAinv1[n][n],V_x1[n];

    Crea_mat();                          // Crea la matriz A de (nxn).
    Crea_Vb();                           // Crea el vector b de (nx1).

    des_LU(&mL1,&mU1);                   // Descomposición de la matriz  A en L y U.

    Inversa(&mL1,&mU1,&mAinv1);    // Matriz inversa de A.
```

```
    return 0;

}

void Crea_mat()
{
  int i,j;

   for(i=0; i<n; i++)
   {
    for (j=0; j<n; j++)
    {
      mA1[i][j]=(rand()%10)+1;
    }
   }
}

void Crea_Vb()
{
   int i;

    for (i=0; i<n; i++)
    {
      V_b1[i]=(rand()%10)+1;
    }
}

void des_LU(double (*mL)[n][n],double (*mU)[n][n])
{
  int i,j,k;

   for (i=0; i<n; i++){
     for (j=0; j<n; j++){
       (*mU)[i][j]=mA1[i][j];
     }
   }
   for (i=0; i<n; i++){
     for (j=0; j<n; j++){
       if (i==j)
       (*mL)[i][j]=(*mU)[i][j]/(*mU)[i][j];
       else
       (*mL)[i][j]=0.0;
     }
   }
   for (i=0; i<n-1; i++)
   {
```

```
      for (j=i+1; j<n; j++)
      {
        (*mL)[j][i]=(*mU)[j][i]/(*mU)[i][i];
        for (k=i; k<n; k++)
        {
          (*mU)[j][k]=(*mU)[j][k]-(*mL)[j][i]*(*mU)[i][k];
        }
      }
    }
}

void Inversa(double (*mL)[n][n],double (*mU)[n][n],double (*mAinv)[n][n])
{
     int i;

    pthread_mutex_lock(&mul_candado);
    pthread_mutex_lock(&trabajo.candado);

   if (trabajo.crear_hilos == 0) {
     for (i = 0; i < NCPU; i++) {
      pthread_create(&tid[i], NULL, trabajador1, (void *)NULL);
     }
     trabajo.crear_hilos = NCPU;
   }
   trabajo.mL=mL; trabajo.mU=mU; trabajo.mAinv=mAinv;
   trabajo.col = 0;
   trabajo.trab_xhacer = trabajo.trab_nrealzd = n;
   pthread_cond_broadcast(&trabajo.ini_hilo);
  while (trabajo.trab_nrealzd)
    pthread_cond_wait(&trabajo.fin_hilo, &trabajo.candado);
    pthread_mutex_unlock(&trabajo.candado);
    pthread_mutex_unlock(&mul_candado);
}

void *trabajador1(void *)
{
  double (*mL)[n][n],(*mU)[n][n],(*mAinv)[n][n];
  double Vb[n],Vx[n];
  double su_U=0;
  int col,i,j,k,t,s;

  while (1) {
    pthread_mutex_lock(&trabajo.candado);
   while (trabajo.trab_xhacer == 0)
     pthread_cond_wait(&trabajo.ini_hilo, &trabajo.candado);
    trabajo.trab_xhacer--;
    mL=trabajo.mL;  mU=trabajo.mU;  mAinv=trabajo.mAinv;
```

```
    col = trabajo.col;
    trabajo.col++;
    pthread_mutex_unlock(&trabajo.candado);

    for(j=0; j<n; j++)
    {
      Vb[j]=0;
    }
    Vb[col]=1;

    for (k=0; k<n-1; k++)
    {
      for (j=k+1; j<n; j++)
      {
        Vb[j]=Vb[j]-(*mL)[j][k]*Vb[k];
      }
    }

    for (t=n-1; t>=0; t--)
    {
      Vx[t]=Vb[t]/(*mU)[t][t];
    }
    for (t=n-1; t>=0; t--)
    {
      su_U=0;
      for (s=t+1; s<n; s++)
      {
        su_U = su_U + (*mU)[t][s]*Vx[s];
      }
      Vx[t] =(Vb[t] - su_U) / (*mU)[t][t];
    }

    for(k=0; k<n; k++)
    {
      (*mAinv)[k][col]=Vx[k];
    }

    pthread_mutex_lock(&trabajo.candado);
    trabajo.trab_nrealzd--;
    if (trabajo.trab_nrealzd == 0)
      pthread_cond_signal(&trabajo.fin_hilo);
    pthread_mutex_unlock(&trabajo.candado);
  }
  return (NULL);
}
```

A continuación se presenta el listado de programa del *trabajador2* que se implementó para la multiplicación matricial **A**x**B**.

```
void *trabajador2(void *)
{
  double (*mA)[n][n],(*mB)[n][n],(*mC)[n][n];
  double result;
  int col2,i,j;

  while (1) {
    pthread_mutex_lock(&trabajo2.candado2);
    while (trabajo2.trab_xhacer2 == 0)
      pthread_cond_wait(&trabajo2.ini_hilo2, &trabajo2.candado2);
    trabajo2.trab_xhacer2--;
    mA=trabajo2.mA; mB=trabajo2.mB; mC=trabajo2.mC;
    col2 = trabajo2.col2;
    trabajo2.col2++;

   if(trabajo2.col2 == n){
     trabajo2.col2==0;
   }

    pthread_mutex_unlock(&trabajo2.candado2);

   for(i=0; i<n; i++)
   {
    result=0;
    for (j=0; j<n; j++)
    {
     result +=(*mA)[i][j]*(*mB)[j][col2];
    }
 (*mC)[i][col2]=result;
   }

    pthread_mutex_lock(&trabajo2.candado2);
    trabajo2.trab_nrealzd2--;
    if (trabajo2.trab_nrealzd2 == 0)
      pthread_cond_signal(&trabajo2.fin_hilo2);
    pthread_mutex_unlock(&trabajo2.candado2);
  }
  return (NULL);
}
```

A continuación se presenta el listado de programa del *trabajador3* que se implementó para el producto matricial **c**=**A***__b__.

```
void *trabajador(void *)
{
 double (*mA)[n][n],(*V_b)[n], (*V_c) [n];
 double result;
 int col,i,j;

 while (1) {
  pthread_mutex_lock(&trabajo.candado);
  while (trabajo.trab_xhacer == 0)
   pthread_cond_wait(&trabajo.ini_hilo, &trabajo.candado);
  trabajo.trab_xhacer--;
  mA=trabajo.mA;  V_b=trabajo. V_b;  V_c=trabajo.V_c;
  col = trabajo.col;
  trabajo.col++;
  pthread_mutex_unlock(&trabajo.candado);

 for(i=0; i<n; i++)
 {
  result  + = (*mA)[col][i] * (*V_b)[i];
 }
  (*V_c) [col]=result;

  pthread_mutex_lock(&trabajo.candado);
  trabajo.trab_nrealzd--;
  if (trabajo.trab_nrealzd == 0)
   pthread_cond_signal(&trabajo.fin_hilo);
  pthread_mutex_unlock(&trabajo.candado);
 }
```

```
 return (NULL);
}
```

A continuación se presenta el listado de programa del *trabajador3* que se implementó para la suma matricial **A**+**B**.

```
void *trabajador3(void *)
{
 double (*mA)[n][n],(*mB)[n][n], (*mC)[n][n];
 double result3;
 int col3,i,j;

 while (1) {
  pthread_mutex_lock(&trabajo3.candado3);
  while (trabajo3.trab_xhacer3 == 0)
   pthread_cond_wait(&trabajo3.ini_hilo3, &trabajo3.candado3);
  trabajo3.trab_xhacer3--;
  mA=trabajo3.mA; mB=trabajo3.mB; mC=trabajo3.mC;
  col3 = trabajo3.col3;
  trabajo3.col3++;

  pthread_mutex_unlock(&trabajo3.candado3);

  for(i=0; i<n; i++)
  {
   result3=0;
   result3 = (*mA)[i][col3] + (*mB)[i][col3];

  (*mC)[i][col3]=result3;
  }

  pthread_mutex_lock(&trabajo3.candado3);
  trabajo3.trab_nrealzd3--;
  if (trabajo3.trab_nrealzd3 == 0)
   pthread_cond_signal(&trabajo3.fin_hilo3);
  pthread_mutex_unlock(&trabajo3.candado3);
 }
 return (NULL);
}
```

A continuación se presenta el listado de programa del *trabajador4* que se implementó para la transpuesta de la matriz **A**.

```
void *trabajador4(void *)
{
 double (*mA)[n][n];
 double result,result2;
 int col4,i,j;

 while (1) {
  pthread_mutex_lock(&trabajo4.candado4);
  while (trabajo4.trab_xhacer4 == 0)
   pthread_cond_wait(&trabajo4.ini_hilo4, &trabajo4.candado4);
  trabajo4.trab_xhacer4--;
  mA=trabajo4.mA;
  col4 = trabajo4.col4;
  trabajo4.col4++;

 if(trabajo4.col4 == n){
  trabajo4.col4==0;
 }

  pthread_mutex_unlock(&trabajo4.candado4);

  for (i=0; i<n; i++)
  {
    result = (*mA)[i][col4];
   (*mA)[col4][i]=result;
  }

  pthread_mutex_lock(&trabajo4.candado4);
  trabajo4.trab_nrealzd4--;
  if (trabajo4.trab_nrealzd4 == 0)
   pthread_cond_signal(&trabajo4.fin_hilo4);
  pthread_mutex_unlock(&trabajo4.candado4);
 }
 return (NULL);
}
```

Referencias

[1] R. A. Saleh, k. A. Gallivan, M. C. Chang, I. N. Hajj, D. Smart, y T. N. Trick, "Parallel circuits simulation on supercomputers". Proc. IEEE, vol. 77, pp. 1915-1931, Dec. 1989.

[2] W. M. Schubert y R. F. Stengel, "Parallel synthesis of robust control systems." IEEE Trans. Control Syst. Technol., vol. 6 pp. 701-706, Nov. 1998.

[3] G. M. Asher y M. Summer, "Parallelism and the transputer for real-time high-performance control of AC induction motors." Proc. Inst. Elect. Eng. (IEE), vol. 137, pp. 179-188, July 1990.

[4] Taks Force of the Computer and Analytical Methods Subcommittee of the Power Systems Engineering Committee, "Parallel processing in power systems computation." IEEE Trans. Power Syst., vol. 7, pp. 629-638. May 1992.

[5] G. W. Stagg y A. H. El-Abiad, Computer Methods in Power System Analysis, EUA: McGraw-Hill, 1968.

[6] N. Garcia y A. Medina "Swift Time Domain Solution of Electric System Including SUSs", IEEE Transactions on Power Delivery, vol. 18, no 3, págs. 921-927, Julio 2003.

[7] D. Fabian Garcia Nocetti y Peter J. Fleming, Parallel Processing in Digital Control, Springer-Verlag, 1992.

[8] W. H. Press, S.A. Teukolsky, W. T. Vetlerling y B. P. Flaanery, Numerical Recipes in C++: The art of Scienfic Computing, Cambridge University Press, Reino Unido, 2002.

[9] Seyed H. Roosta, Paralell Procesing and Parallel Algorithms (Theory and Computation), New York: Springer Verlag, 2000.

[10] Steven C. Chapra y Raymond P. Canale, Métodos Numéricos para Ingenieros, México D. F.: Mc Graw Hill, 2003.

[11] Donal H. Sanders, Informática Presente y Futuro, México D. F.: Mc Graw Hill, 1990.

[12] Samuel Garrido Daniel "Programación Paralela", CINVESTAV-IPN, Departamento de Ingeniería Eléctrica (Sección de Computación), México, D.F. 2/Agosto/2004.

[13] SUN Microsystems, "Multithreaded Programming Guide" EU: SUN, 1998.

[14] G. M. Amdahl, "Validity of the Single-Processor Approach to Achieving Large-Scale Computer Capabilities," AFIPS Conference Proceedings, Vol. 30, págs. 483-485, 1967.

[15] R. E. Walpole y R. H. Myers, Probabilidad y Estadística, México D. F.: Interamericana, 1989.

[16] Charles E. Rohrs, James L. Melsa y Donald G. Schultz, Sistemas de Control Lineal, México D. F.: Mc Graw Hill, 1994.

[17] Leon O. Chua y Pen-Min Lin, Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques, E.U. Nueva Jersey, 1975.