Universidad Michoacana de San Nicolás de Hidalgo
Facultad de Ingeniería Eléctrica

# TÉCNICAS DE PROCESAMIENTO EN PARALELO BASADAS EN OPENMP COMO UNA HERRAMIENTA PARA EL ANÁLISIS DE SISTEMAS ELÉCTRICOS

**Tesis**

**Que para obtener el Titulo de**
**INGENIERO** EN **ELECTRÓNIC**A

**Presenta**
**Eustaquio García Ramírez**

**Asesor de Tesis**
**Dr. Antonio Ramos Paz**

**Agosto del 2010**

# Agradecimientos

Primero que a nadie, agradezco a Dios por haberme ayudado en todos los momentos a que todo se conjuntara para que yo pudiera terminar la carrera.

Agradezco de una manera muy especial a mis padres que, a pesar de tantos obstáculos que se pusieron en el camino jamás dijeron “no se puede”, porque siempre hicieron hasta el último esfuerzo para que yo no dejara de estudiar.

A toda mi familia, especialmente a mi hermana María porque siempre tuvo la voluntad de ayudarnos a todos. Generalmente agradezco a todos mis hermanos y hermanas porque de una u otra manera siempre me han apoyado para que yo pudiera concluir la licenciatura de ingeniería electrónica.

Por último agradezco a alguien que estuvo conmigo, durante toda la carrera, a la persona con la que compartí mis humildes logros, y mis momentos difíciles, a quien cada fin de semana me esperaba con gusto y me deseaba un buen inicio de semana cuando me despedía de ella para venirme a seguir con mis estudios, amiga y novia, Andrea Elizabeth.

# Dedicatoria

A mis padres:

Dorotea Ramírez

Damián García

# Resumen

Esta tesis presenta la propuesta e implementación de un esquema de procesamiento en paralelo basado en OpenMP aplicado a operaciones matriciales de grandes dimensiones.

Los algoritmos paralelos desarrollados son comparados contra los algoritmos secuenciales en donde se muestra la ventaja del uso de las técnicas de procesamiento en paralelo. Los algoritmos presentados son programados en lenguaje C y fueron ejecutados en una computadora de 8 núcleos.

Esta tesis hace también una descripción detallada de los principales conceptos asociados con el procesamiento en paralelo, analiza las principales arquitecturas computacionales usadas para el procesamiento en paralelo así como las plataformas operativas aplicadas en el desarrollo de programas en paralelo.

La eficiencia de los esquemas de procesamiento en paralelo propuestos, aplicados en operaciones con matrices de grandes dimensiones, se mide en términos de eficiencia relativa (*speed up*).

Finalmente se realiza una serie de recomendaciones para trabajos futuros asociados con las aplicaciones del procesamiento en paralelo en el análisis de redes eléctricas de gran escala.

# Contenido


Agradecimientos…….…… ii

Dedicatoria…….……... iii

Resumen……..…….. iv

Lista de Figuras……..…….. viii

Lista de Tablas…….. xi

Lista de Símbolos y Abreviaciones…….. xii

**Capítulo 1. Introducción** 1

1.1 Antecedentes…….……... 3

1.2 Objetivos de la Tesis..……... 5

1.3 Justificación…….. 6

1.4 Metodología…….. 7

1.5 Descripción de los Capítulos……..…….. 7

**Capítulo 2. Arquitecturas de procesamiento en paralelo** 9

2.1 Introducción……... 9

2.2 Arquitectura Multicomputadoras…….. 9

2.2.1 Taxonomía de los ambientes de cómputo según Flynn..…… 10

2.2.2 Multicomputadoras con base en buses…… 14

2.2.3 Multicomputadoras con conmutador……. 14

2.3 Arquitectura multiprocesador……..... 15

2.3.1. Multiprocesador con base en buses…… 16

2.3.2. Multiprocesador con conmutador……... 16

2.4 Arquitectura híbrida……..……..... 17

2.5 Conclusiones……. 18

**Capítulo 3. Plataformas de procesamiento en paralelo** 19
3.1 Introducción…….... 19
3.2. Interfaz de paso de mensajes, MPI……..... 19
3.2.1. Comunicación punto a punto…..……..... 20
3.2.2. Comunicaciones colectivas……. 22
3.3. Máquina virtual paralela…… 25
3.3.1. El sistema PVM…….. 25
3.3.2. Estructura de un programa con PVM……... 26
3.4. Pthreads…… 28
3.4.1. Creación de threads……... 28
3.4.2 Atributos de threads…….. 29
3.4.3 Terminando threads……... 30
3.5. OpenMP…….. 31
3.5.1. Elementos básicos……..... 32
3.5.2. Compilación y ejecución…… 32
3.5.3. Paralelización de ciclos…… 33
3.5.4. Regiones paralelas……... 35
3.5.5. Variables privadas y variables públicas……..... 37
3.6 Conclusiones…… 38

**Capitulo 4. Propuesta de paralelización para las operaciones matriciales** 39
4.1 Introducción……..... 39
4.2 Operaciones matriciales…… 39
4.2.1 Suma de matrices……..... 39
4.2.2 Algoritmo secuencial para la suma de matrices……..... 40
4.2.3 Resta de matrices……..... 41
4.2.4 Algoritmo secuencial para la resta de matrices……... 42
4.2.5 Multiplicación de una matriz por un escalar……... 42
4.2.6 Algoritmo secuencial para la multiplicación matriz escalar……... 43
4.2.7 Producto punto……..... 43

4.2.8 Algoritmo secuencial para el producto punto…… 44
4.2.9 Producto de una matriz por un vector…….... 44
4.2.10 Algoritmo secuencial para el producto matriz-vector…… 44
4.2.11 Producto de matrices……... 45
4.2.12 Algoritmo secuencial para el producto de matrices…… 46
4.3 Propuesta de paralelización de las operaciones matriciales…….. 47
4.3.1 Algoritmo paralelo para la suma de matrices…….. 47
4.3.2 Algoritmo paralelo para la resta de matrices…….. 48
4.3.3 Algoritmo paralelo para la multiplicación de una matriz por un escalar 49
4.3.4 Algoritmo paralelo para el producto punto……. 49
4.3.5 Algoritmo paralelo para el producto matriz vector…… 50
4.3.6 Algoritmo paralelo para el producto de matrices…… 51
4.4 Conclusiones…….. 52

**Capitulo 5. Casos de estudio.** 53
5.1 Introducción……... 53
5.2 Caso de estudio 1. Suma de matrices…… 54
5.3 Caso de estudio 2. Resta de matrices…… 57
5.4 Caso de estudio 3. Multiplicación matriz-escalar……..... 59
5.5 Caso de estudio 4. Producto punto…… 61
5.6 Caso de estudio 5. Producto matriz-vector…… 62
5.7 Caso de estudio 6. Producto de matrices…… 64

**Capítulo 6. Conclusiones y Trabajos Futuros** 67
6.1. Conclusiones ……... 67
6.2. Trabajos Futuros…….. 68
**Referencias** ……... 69

# Lista de Figuras


2.1. Categoría SISD……..……...……..... 11
2.2. Categoría SIMD……... 12
2.3. Categoría MISD…….……...……... 12
2.4. Categoría MIMD……. 13
2.5. Taxonomía de los equipos de cómputo…… 13
2.6. Multicomputadoras con base en buses……. 14
2.7. Topología Matriz para multicomputadoras…….. 14
2.8. Topología hipercubo de dimensión 4 para multicomputadoras……..... 15
2.9. Topología hipercubo de dimensión 5 para multicomputadoras……... 15
2.10. Multiprocesadores con base en buses…….. 16
2.11. Modelo Cross-bar……. 17
2.12. Memoria compartida distribuida…….. 17

3.1 Ejemplo de comunicación punto a punto……. 20
3.2 Modelo de PVM……... 26
3.3 Estructura de un programa en PVM……. 27
3.4 Creación de hilos a partir de un hilo existente……. 29
3.5 Modelo de memoria compartida……..……... 32
3.6 Paralelización de un ciclo for……... 35
3.7 Declaración de variable compartida……. 37
3.8 Declaración de una variable privada……..... 38

4.1 Código secuencial para la suma de matrices……..... 40
4.2 Disposición en memoria del almacenamiento de la matriz……. 41
4.3 Recorrido del procesador a través de le memoria……..... 41
4.4 Código secuencial para la resta de matrices…… 42
4.5 Código secuencial para el producto matriz escalar…….. 43
4.6 Código secuencial para el producto punto……... 44
4.7 Código secuencial para el producto matriz vector……..……. 44

4.8 Algoritmo secuencial para el producto de matrices…….. 46
4.9 Propuesta de paralelización de la suma de matrices…….. 47
4.10 Repartición de los datos entre los procesadores para la suma de matrices……... 47
4.11 Propuesta de paralelización para la resta de matrices……... 48
4.12 Repartición de los datos entre los procesadores para la resta……... 48
4.13 Algoritmo paralelo para el producto matriz-escalar……. 49
4.14 Algoritmo paralelo para el producto punto…….. 49
4.15 Algoritmo paralelo del producto matriz-vector…… 50
4.16 Algoritmo paralelo para el producto de matrices……... 51

5.1 Equipo de trabajo *Intel(R) Xeon(R) CPU E5405 @ 2.00GHz*……. 53
5.2 Vista de los procesadores……. 54
5.3 Vista de las memorias……. 54
5.4 Tiempo de ejecución de la suma de matrices…… 56
5.5 Gráfica de Speed-Up para la suma de matrices……... 56
5.6 Tiempo de ejecución para la resta de matrices……....……... 58
5.7 *Speed-Up* para la resta de matrices…….. 58
5.8 Tiempo de ejecución para la operación matriz-escalar…… 60
5.9 Speed-Up para la operación matriz-escalar……. 60
5.10 Tiempo de ejecución para el producto punto…….. 61
5.11 Speed-Up para el producto punto…… 62
5.12 Gráfica del tiempo de ejecución del producto matriz-vector …….. 63
5.13 Gráfica del Speed-Up para para el producto matriz-vector…….. 64
5.14 Gráfica del tiempo de ejecución del producto de matrices……. 66
5.15 *Speed-Up* para el producto de matrices…… 66

# Lista de Tablas


3.1 Tipos de datos para MPI……..... 22

3.2 Datos admitidos para las funciones MPI…… 23

5.1 Resultados obtenidos de la suma de matrices del orden de 100000000..…... 55

5.2 Tiempo de ejecución de la suma de matrices……. 55

5.3 Speed-Up para la suma de matrices…… 55

5.4 Tiempo de ejecución de la resta de matrices del orden de 100000000…….. 57

5.5 Tiempo de ejecución promedio de la resta de matrices……. 57

5.6 Speed-Up para la suma de matrices…… 57

5.7 Tiempo de ejecución del producto matriz-escalar con 10000 elementos…... 59

5.8 Tiempo de ejecución promedio del producto matriz-escalar…….. 59

5.9 *Speed-Up* para el producto matriz-escalar…….. 59

5.10 Tiempo de ejecución promedio del producto punto del orden de 100000000 61

5.11 *Speed-Up* para el producto punto…… 61

5.12 Tiempo de ejecución del producto matriz-vector……... 62

5.13 Tiempo de ejecución promedio del producto matriz-vector……... 63

5.14 *Speed-Up* del producto matriz-vector del orden de 100000000 elementos… 63

5.15 Tiempo de ejecución del producto de matrices……... 65

5.16 Tiempo de ejecución promedio del producto de matrices…….. 65

5.17 *Speed-Up* para el producto de matrices……. 65

# Lista de Símbolos y Abreviaturas

MPI Interfaz de Paso de Mensajes

PVM Máquina Virtual Paralela

API Interfaz de Programación de Aplicaciones

SISD Una sola instrucción un solo dato

SIMD Una instrucción, múltiples datos

MISD Múltiples instrucciones, un solo dato

MIMD Múltiples instrucciones, múltiples datos

DSM Memoria compartida distribuida

DVSM Memoria virtual compartida distribuida

NUMA Acceso a Memoria no Uniforme

CC-NUMA Coherencia de Cache, con Acceso a Memoria no Uniforme

COMA Arquitectura de memoria solamente cache

# Capítulo 1

# Introducción

El cómputo paralelo consiste en el uso de varios procesadores para que trabajen de forma conjunta en la solución de un problema computacional. Una de las aplicaciones de la programación en paralelo es el procesamiento de imágenes, donde existe un procesador maestro que se encarga de la administración de las imágenes y se comunica con los procesadores esclavos a través del envío de mensajes necesarios para la repartición de las mismas entre los procesadores participantes [M. Aguilar-Cornejo 2007]. Así como este proyecto existen otros más, como la paralelización del método *Solver* Frontal utilizado para la solución de sistemas de ecuaciones, aplicados a ecuaciones diferenciales en derivadas parciales en un modelo de corriente.

Generalmente cada uno de los procesadores realiza una parte del trabajo y se comunica con los demás procesadores mediante intercambio de datos. Dependiendo de la manera en que se realice el intercambio se tienen dos modelos distintos de procesamiento en paralelo. Los dos casos básicos son:

- Donde la memoria es compartida, a la que todos los procesadores pueden acceder directamente. En este caso que es el que nos ocupa en esta tesis se tiene el modelo de Memoria Compartida (*Shared Memory Model*), cada procesador puede leer o escribir directamente a todos los módulos de memoria, tanto si los tiene directamente asociados como si están asociados a otros procesadores. Las plataformas de programación para este modelo de programación son *pthreads [Pthreads 2010]* y *OpenMP [OpenMP 2008]*, considerado éste último como el estándar para las arquitecturas de memoria compartida.

- Por otra parte se tiene el modelo de Memoria distribuida. En esta arquitectura cada procesador tiene asociado un bloque de memoria y ningún procesador puede acceder

directamente a bloques de memoria que correspondan a otros procesadores. De manera que para comunicarse entre ellos será necesario que el procesador que necesite datos de otro procesador realice un llamado hacia ese elemento que contiene los datos que el primero necesita. Este es el modelo de Paso de Mensajes. Existen varios entornos para el Paso de Mensajes tales como la Máquina Paralela Virtual (PVM) [Geist 1994], la Interface de Paso de Mensajes (MPI) *[MPI 2009]*.

Algunos de las aplicaciones donde se utiliza el procesamiento en paralelo son: predicciones y estudios meteorológicos y estudio del genoma humano, por ejemplo. En el campo de la ingeniería eléctrica ha sido utilizado en la solución de problemas de flujos de potencia, análisis de estabilidad, respuesta a la frecuencia análisis del estado estable, etc.

A continuación se enlista un resumen acerca de las diferentes librerías de programación en paralelo:

- La Interfaz de Paso de Mensajes (conocido ampliamente como MPI, siglas en inglés de *Message Passing Interface*) es una especificación estándar para procesamiento en paralelo mediante paso de mensajes. Establece el modelo de programación en paralelo de paso de mensajes, en el que un dato es movido del espacio de direcciones de un proceso a otro proceso a través de operaciones cooperativas. Todas las operaciones de MPI son expresadas como funciones, subrutinas, o métodos para lenguajes de programación como C, C++, Fortran-77, y Fortran-95. El estándar ha sido definido mediante procesos abiertos por una comunidad de investigadores de procesamiento en paralelo, científicos de la computación y aplicaciones desarrolladas *[MPI 2009]*.

- PVM (*Parallel Virtual Machine*, Máquina Virtual Paralela) es otro entorno de programación en paralelo, dentro del cual se pueden desarrollar programas de una manera eficiente usando hardware existente. PVM hace que una colección heterogénea de computadoras sea vista como una sola máquina virtual paralela. El usuario escribe su aplicación como una colección de tareas cooperativas. Las tareas accesan a los recursos de PVM a través de rutinas de la interfaz. Estas rutinas permiten la inicialización y

terminación de tareas a través de la red así como comunicación y sincronización entre las tareas. Las estructuras de comunicación incluyen aquellas para enviar y recibir estructuras de datos tal como *broadcast*, barreras de sincronización y suma global *[Geist 1994]*.

- En arquitecturas de multiprocesadores de memoria compartida, tal como SMPs, los hilos pueden ser usados para implementar paralelismo. Históricamente, los fabricantes de hardware han implementado sus propias versiones de hilos, haciendo la portabilidad un interés primordial para desarrolladores de software. Para sistemas UNIX, una interface de programación de hilos en lenguaje C estandarizado ha sido especificado por la norma IEEE POSIX 1002.1c. Las implementaciones que son adheridas a este estándar son llamados POSIX threads, o Pthreads *[Pthreads 2010].*

- OpenMP es una API (*Application Program Interface*) que posibilita la programación paralela en entornos multiprocesador con memoria compartida, como es el caso de la mayoría de los procesadores actualmente en el mercado. Utilizando modificaciones en los compiladores, esta tecnología permite el desarrollo incremental de aplicaciones paralelas a partir de código fuente serie. Es el estándar actual para programación con memoria compartida, incluye los sistemas multicore y computadoras de altas prestaciones con memoria virtual compartida. Típicamente se trata de sistemas con un número no demasiado alto de procesadores: desde 2 en un dual core hasta alrededor de 128 en una supercomputadora. También hay versiones de OpenMP para otros tipos de sistemas, como GPU y clusters de procesadores, que es, precisamente el equipo que se estará utilizando en este proyecto *[Chapman 2008]*.

## 1.1 ANTECEDENTES

El procesamiento en paralelo ha tenido éxito en la solución de problemas donde se involucran grandes cantidades de procesamiento de información y donde se requieren velocidades de respuesta considerables, lo cual es prácticamente imposible de realizar con una computadora de uso convencional.

Este tipo de procesamiento data desde la aparición de los primeros ordenadores en los años 50. En 1955, el IBM 704 incluye un hardware para procesamiento de números de punto flotante (co-procesador). En 1956, IBM lanza el proyecto 7030 (conocido como STRETCH) para producir una "supercomputadora" para Los *Alamos National Laboratory* (LANL). El objetivo en esta época era construir una máquina con capacidad computacional 100 veces mayor que cualquier otra máquina disponible en ese tiempo. En el mismo año, el proyecto LARC (*Livermore Automatic Research Computer*) comienza a proyectar otra "supercomputadora" para el *Lawrence Livermore National Laboratory* (LLNL). Estos dos proyectos tardarán tres años para producir los dos primeros productos: los supercomputadores conocidos como STRETCH y LARC *[Chapman 2008]*.

En 1960 *Control Data* inicia el desarrollo del proyecto CDC 6600. En el año de 1962 CDC entrega su primera CDC 1604, máquina similar a la IBM 7090 caracterizada por palabras de 48 bits y ciclos de memoria de 6μs. Es la primera máquina en usar memoria virtual y paginación, su ejecución de instrucciones es en oleoducto (pipelined), y contiene unidades aritméticas de punto flotante y punto fijo separadas. Su desempeño es de aproximadamente 200kFLOPS. C. A.

En 1964 *Control Data Corporation* empieza a producir la CDC 6600, la primer supercomputadora en ser un éxito técnico y comercial. Cada máquina tiene una CPU de 60 bits y 10 unidades periféricas de procesamiento (PPUs). La CPU utiliza un marcador para manejar la dependencia de instrucciones. IBM inicia el diseño del *Advanced Computer System* (ACS), capaz de manejar hasta siete instrucciones por ciclo.

En el año de 1965 *General Electric*, el *MIT*, y *AT&T Bell Laboratories* comienzan a trabajar en *Multics*. El objetivo del proyecto es la construcción de un sistema operativo de propósito general de memoria compartida, multiprocesamiento y tiempo compartido. *James W. Cooley* y *John W. Tukey* describen el Algoritmo de la Transformada Rápida de Fourier, que es posteriormente uno de los más grandes consumidores de ciclos de punto flotante.

En Marzo de 1976 La *Cray-1* es la primera computadora en usar el procesamiento vectorial y tenía una capacidad de procesamiento pico de 100 MFLOPS. Frecuencia de reloj 110 MHz; 9 ns ciclo del núcleo.

Aparecieron diferentes entornos de programación en paralelo como MPI cuyo estándar final fue presentado en la conferencia de Supercómputo en noviembre de 1993, constituyéndose así el foro para el MPI. Otro es PVM, la primera versión fue escrita en ORNL en 1989, y después de ser modificado por la Universidad de Tennessee, fue lanzada la versión 2 en marzo de 1991. La versión 3 fue lanzada en marzo de 1993 con mejoras en la tolerancia a fallas y portabilidad. El estándar de Threads surgió en el año de 1995.

OpenMP fue basado en los inicios del trabajo del PCF (*Parallel computing Forum*). La primera versión consiste de una serie de directivas, la cual fue pensada para ser usada en Fortran. Fue introducida para el público en 1997. Los compiladores OpenMP empezaron a aparecer después de ese año. En el margen de ese tiempo apareció para C y fue introducido también para C++, y la serie de características ha sido extendida. Los compiladores son ahora válidos para todas las plataformas SMPs.

OpenMP fue diseñado para habilitar la programación paralela de memoria compartida portable. Está dirigido para ayudar a la paralelización de aplicaciones de muchas disciplinas. Sus creadores pensaron en proporcionar un medio que fuera relativamente fácil de aprender y de aplicar. La API está diseñada para permitir un enfoque incremental para paralelizar un código existente, inicialmente en su forma secuencial. Este es el contraste marcado para la conversión todo-o-nada de un programa entero en un solo paso, que es típicamente requerido por otro paradigma de programación en paralelo.

## 1.2 Objetivo

Objetivo general. Aplicar técnicas de procesamiento en paralelo basadas en OpenMP en la programación de operaciones matriciales de gran escala.

Objetivos particulares:

1. Implementación de las operaciones matriciales de suma, resta, multiplicación de un vector por un escalar, multiplicación de una matriz pro otra matriz, producto punto usando procesamiento secuencial.
2. Implementación de las operaciones matriciales de suma, resta, multiplicación de un vector por un escalar, multiplicación de una matriz pro otra matriz, producto punto usando procesamiento en paralelo basado en OpenMP.

## 1.3 Justificación

La necesidad del procesamiento en paralelo se origina por las limitaciones de las computadoras personales, es decir computadoras que cuentan con un solo procesador.

Las principales rezones del uso de la programación paralela son disminuir el tiempo total de ejecución de una aplicación.

También se podría pensar en los límites físicos de la computación de una máquina secuencial, que actualmente ya están en la frontera de lo que puede ser practicable en términos de velocidad interna de transmisión de datos y de velocidad de CPU. Además de esto, podemos decir que durante los últimos diez años, las nuevas tecnologías de procesadores avanzan hacia la presencia constante del procesamiento en paralelo.

Generalmente las aplicaciones del procesamiento en paralelo son problemas de alto costo y problemas de no tan alto costo, pero de gran dimensión, como son las operaciones matriciales que es el tema principal del presente proyecto de tesis. De la misma manera es importante la aplicación del procesamiento en paralelo en problemas de tiempo real, en los que se necesita respuesta en un tiempo mínimo, por ejemplo en estudios meteorológicos.

En ingeniería eléctrica, algunas de las áreas donde hace sentir la necesidad del procesamiento en paralelo es en los sistemas de potencia, donde el análisis se realiza por medio de un modelo matemático, es decir, una representación matricial cuyas dimensiones son de miles o millones de elementos. Estas proporciones son las que, en un momento dado provocan lentitud a la hora de monitorear, e incluso tener un control sobre el sistema (en tiempo real), lo que pondría en riesgo la seguridad del mismo. Por ello es que en este proyecto de tesis se presenta una propuesta basada en procesamiento en paralelo aplicado a operaciones matriciales de grandes dimensiones.

## 1.4 Metodología

1. Durante la primera etapa de este proyecto de investigación se realizará un estudio acerca del procesamiento en paralelo y algunas de las principales aplicaciones.
2. Posteriormente se realizará una revisión bibliográfica acerca del procesamiento en paralelo basado en la plataforma operativa de OpenMP.
3. En esta etapa, se programaran las operaciones matriciales básicas presentadas en esta tesis utilizando procesamiento en paralelo basado en OpenMP.
4. De manera paralela a todos estos elementos se realizara la escritura de la tesis.

## 1.5 Descripción de los Capítulos

En el Capítulo 1 se da una breve introducción acerca de lo que es el procesamiento en paralelo.

En el Capítulo 2 se describen las principales arquitecturas computacionales usadas en el procesamiento en paralelo.

En el Capítulo 3 se realiza una descripción detallada de las principales plataformas operativas usadas para el procesamiento en paralelo.

En el Capítulo 4 se hace una propuesta de aplicación de procesamiento en paralelo en las operaciones matriciales de suma, resta y multiplicación.

En el Capítulo 5 se realizan varios estudios donde se analiza el efecto del procesamiento en paralelo en el tiempo de ejecución de los programas que realizan las operaciones matriciales descritas en al Capítulo 4.

En el Capítulo 6 se presentan las conclusiones generales obtenidas de esta tesis, así como también se presentan propuestas para trabajos futuros de investigación sobre esta misma línea de investigación.

# Capítulo 2

# Arquitecturas de procesamiento en paralelo

## 2.1 Introducción

Las arquitecturas de procesamiento en paralelo han sido utilizadas en numerosas aplicaciones, y el área de los problemas numéricos ha sido normalmente el punto de partida.

Se han implementado diversos métodos para la solución de los problemas numéricos, generalmente tomando en cuenta dos aspectos [Fernando G. 2003]:

1. Estabilidad numérica, cuando se resuelven con aritmética que involucra algún tipo de error.
2. Implementación directa o con costo mínimo en algún lenguaje de programación.

Los tres tipos de arquitecturas para el procesamiento en paralelo más ampliamente utilizadas son:

1. Arquitectura multicomputadora
2. Arquitectura multiprocesador
3. Arquitectura híbrida

A continuación se detallan las características principales de cada una de estas arquitecturas.

## 2.2 Arquitectura multicomputadora

La arquitectura de multicomputadoras, también conocido como sistema distribuido lo constituyen un arreglo de computadoras interconectadas y un sistema de transmisión de mensajes que en conjunto resuelven tareas de manera colaborativa.

El canal con el cual están conectadas las computadoras indica que debe existir el tráfico de mensajes entre los componentes.

La unidad de la modalidad de paso de mensajes proporciona las funciones para que los procesos de un sistema de memoria distribuida puedan ejecutarse de manera integrada, coherente y sincronizada, con el objetivo de que la ejecución de cada uno, realice su cooperación de acuerdo a la terea común de la aplicación para la que haya sido implementada.

Como la forma en que se comunican los procesos es por paso de mensajes, este funcionamiento se apoya en dos operaciones básicas, una para enviar mensajes, *send* y otra para recibirlos, *receive*. La idea es que cualquiera que sea la topología de la red que dé soporte al paso de mensajes, se establezcan canales a través de los cuales dos procesos *P* y *Q* puedan intercambiar información, por ejemplo del siguiente modo: SEND(*P,m*); RECEIVE(*Q,m*). En este caso el proceso *Q* envía el mensaje *m* al proceso *P* mediante la emisión a través del canal establecido de la sentencia SEND dirigida explícitamente al proceso *P*, el cual recibe el mensaje *m* del canal del proceso *Q* y causa recibo de él a modo de confirmación mediante la emisión a través del mismo canal de la sentencia RECEIVE, en la que especifica el destinatario de la misma (no de m), o sea Q. Este tipo de comunicación implica el establecimiento de un canal (en tiempo y espacio) entre procesos para lo cual ambos deben estar simultáneamente en condiciones de comunicarse, de ahí el nombre de síncrono. Su contrapartida es el modo asíncrono en el que la comunicación se realiza a través de buffers que actúan como buzones de correo. En este caso el proceso que envía el mensaje se despreocupa respecto a si el destinatario está o no en condiciones de recibirlo. No obstante las demoras pueden sobrevenir mientras el proceso que haya enviado un mensaje no reciba la confirmación de parte del proceso que lo hubiese recibido. Los mensajes pueden ser datos, información de sincronización, señales de interrupción, etc.

De esta arquitectura se deriva la taxonomía propuesta por Flynn, de las arquitecturas computacionales para el procesamiento en paralelo [iniciación al PVM, 2009].

### 2.2.1 Taxonomía de los ambientes de cómputo según Flynn

La clasificación de los ambientes de cómputo establecida por *Flynn* en el año de 1996 está basada en la multiplicidad de los flujos de instrucciones y de los flujos de datos dentro del sistema. Esta clasificación propone cuatro esquemas. El proceso de computación es la ejecución de una secuencia de instrucciones sobre un flujo de datos. La palabra *flujo* se utiliza para referirse a una secuencia de elementos (que pueden ser instrucciones o datos) que ejecuta un solo procesador y a su vez las instrucciones o los datos se definen con respecto a la máquina referenciada.

Un flujo de instrucciones es una secuencia de instrucciones ejecutadas por la máquina; un flujo de datos es una secuencia de datos que incluye los datos de entrada y los resultados parciales, totales, solicitados o producidos por el flujo de instrucciones.

Las diferentes organizaciones de computadoras se caracterizan por la multiplicidad de hardware provisto para atender a los flujos de instrucciones y de datos. La categorización de Flynn depende de la multiplicidad de procesos simultáneos que ocurren en los componentes del sistema, aunque conceptualmente sólo son necesarios tres tipos de equipos de dicha taxonomía. En seguida se presentan estas categorías:

**SISD:** *Single Instruction, Single Data*.

En este tipo de máquinas sólo existe un procesador por lo que las instrucciones se ejecutan de manera secuencial. En la Fig. 2.1 aparece un esquema de este tipo de sistemas Los significados de las anotaciones en las figuras son: FD = flujo de datos, FI = flujo de instrucciones, UC = unidad de control, UP = unidad de procesamiento, MM= módulo de memoria.

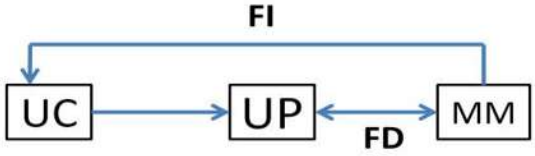


Figura 2.1 Categoría SISD.

**SIMD:** *Single Instruction, Multiple Data*.

En estos sistemas existen *n* unidades de proceso supervisadas por una sola unidad de control. En esta configuración las unidades de proceso reciben la misma unidad de instrucciones pero actúan sobre datos distintos provenientes de diferentes flujos de datos como se ve en la Figura 2.2.

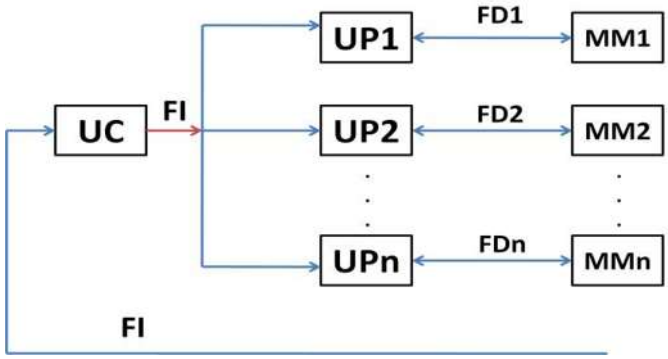


Figura 2.2 Categoría SIMD.

**MISD:** Multiple Instructions, Single Data.

En este caso existen *n* unidades de procesamiento como lo muestra la Figura 2.3. Cada unidad recibe diferentes instrucciones que operan sobre el mismo flujo de datos. Los resultados que se obtiene en un procesador se pasan al siguiente por lo que se tienen varias instrucciones con un único flujo de datos lo que resulta poco práctico al querer obtener velocidad de procesamiento.

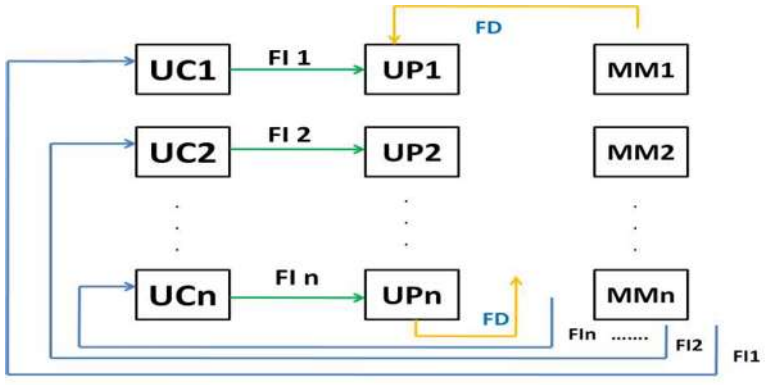


Figura 2.3 Categoría MISD.

**MIMD:** *Multiple Instructions, Multiple Data*.

Los sistemas de memoria compartida y memoria distribuida se incluyen en esta categoría, cuyo diagrama de comunicación aparece en la Figura 2.4. Entonces se divide la categoría MIMD como se muestra en la Figura 2.5 en aquellas que tienen una memoria común y las que tienen una memoria distribuida.

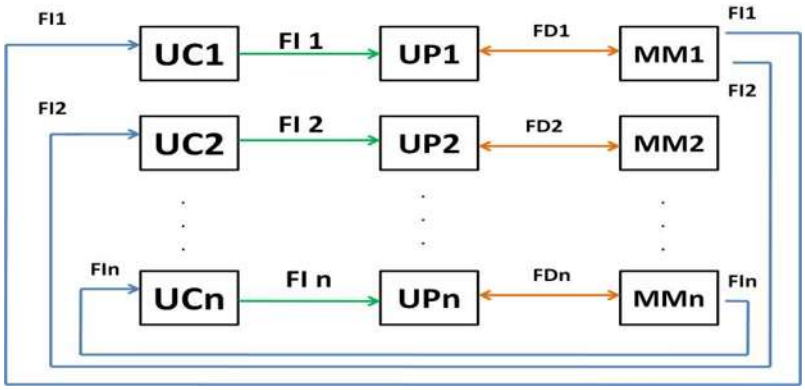


Figura 2.4. Categoría MIMD.

Cada uno de los esquemas se puede subdividir de acuerdo a la red de interconexión. Estas subdivisiones están clasificadas en bus y conmutador como aparece en la Figura 2.5.

En la subcategoría de conexión con bus existe un medio que conecta a todas las máquinas.

Por otra parte en la que se conecta por conmutador las máquinas están conectadas con cables individuales y patrones de cableados específicos.

Figura 2.5 Taxonomía de los equipos de cómputo.

### 2.2.2 Multicomputadoras con base en buses

En este sistema las máquinas no comparten la memoria, más bien cada una de las computadoras posee su propia memoria local como aparece en la Figura 2.6. El medio con el que se conectan favorece al tráfico de información en la comunicación cpu-memoria a diferencia de la configuración de multiprocesador.

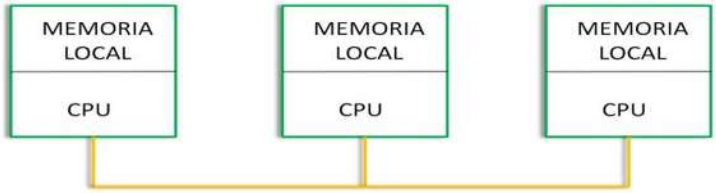


Figura 2.6 Multicomputadoras con base en buses.

Se puede encontrar este tipo de arquitecturas en las redes locales de computadoras (LAN) en las cuales, las computadoras están conectadas por medio de un cableado con una velocidad de 10 a 155 Mbps.

### 2.2.3 Multicomputadora con conmutador

A continuación se muestran las topologías más comunes de esta conexión que es la topología matriz y el hipercubo. La primera aparece en la Figura 2.7, y es utilizada en el procesamiento de imágenes y visión robótica.

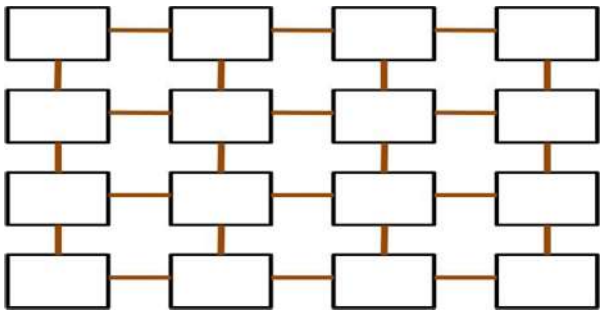

Figura 2.7 Topología Matriz para multicomputadoras.

La segunda está representada en la Figura 2.8, cuyo modelo consiste en un cubo *n* dimensional. En la Figura 2.8 por ejemplo se tiene un cubo de dimensión 4, puesto que los vértices de cada uno de los cubos están conectados con otros cuatro vértices. Se tiene entonces que cada vértice es visto como un procesador.

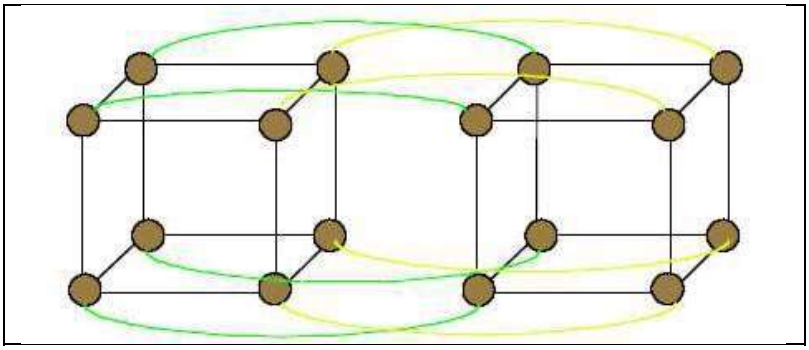

Figura 2.8 Hipercubo de dimensión 4.

Por ejemplo en la Figura 2.9 aparece el esquema de un hipercubo de dimensión 5. Una de las desventajas de las desventajas del crecimiento en el número de procesadores en estos casos es que el tiempo de transferencia de información aumenta de manera exponencial conforme crece el número de procesadores.

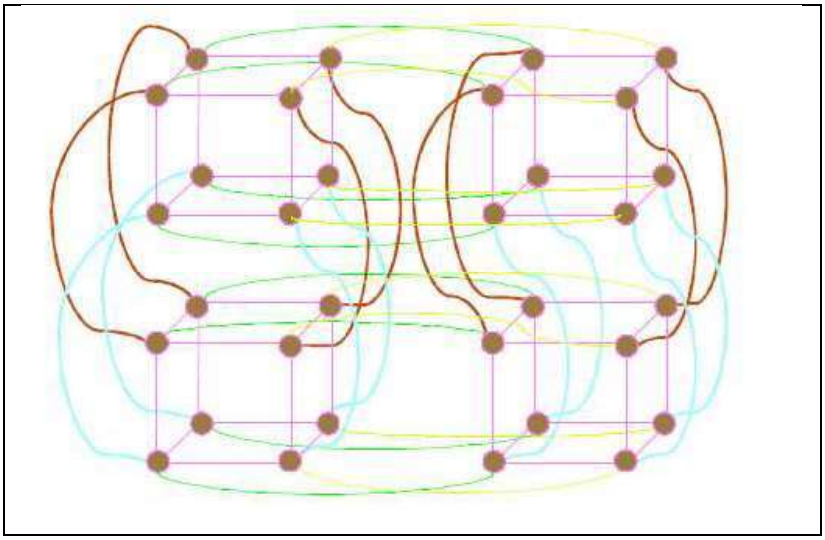

Figura 2.9 Hipercubo de dimensión 5.

## 2.3 Arquitectura multiprocesador

Esta arquitectura difiere de la de paso de mensajes. En este caso se basa en una forma de comunicación más restringida, pero mucho más rápida, cuyo funcionamiento opera

mediante la lectura/escritura compartidas de memoria en las que se almacenan variables compartidas por todos los procesos.

### 2.3.1 Multiprocesadores con base en buses

Este esquema contiene varios procesadores conectados a un bus común, generalmente un *backplane* junto con un módulo de memoria que es compartido por todos los procesadores.

La memoria que posee esta categoría pone a disposición de todos los procesadores lo que otro procesador haya puesto en ella de manera inmediata. Para obtener localidad es necesario aumentar memoria caché a los procesadores para que así haya menor necesidad de que los procesadores no tengan que recurrir al canal de comunicación y puedan usar memoria caché para los datos que cada uno ocupe en un momento dado. Este procedimiento es conocido como *caché monitor*. En la Figura 2.10 se muestra un diagrama de tal arquitectura.

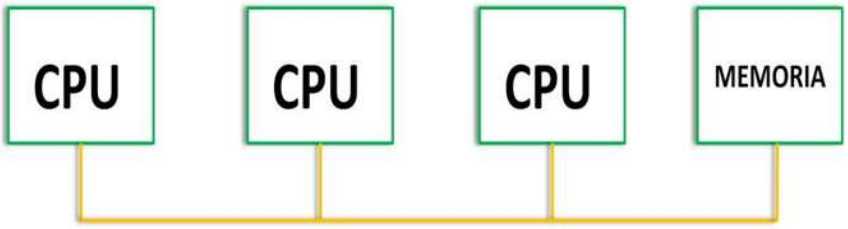


Figura 2.10 Multiprocesador con base en Bus.

### 2.3.2 Multiprocesadores con conmutador

Como se mencionó anteriormente la limitación de la arquitectura multiprocesador crece a medida que aumenta el número de procesadores. En sí al sobre pasar más de 64 procesadores se complica la comunicación entre el procesador y la memoria.

Una ventaja importante es que muchos procesadores pueden accesar a la memoria al mismo tiempo, aunque si dos de ellos desean usar la misma memoria en forma simultánea uno de ellos deberá esperar. Se han hecho modificaciones al cross-bar para minimizar este problema de bloqueo, como el uso de memoria caché en los puntos de cruce o el uso de planificación. La principal problemática del sistema es su tamaño, puesto que con $n$ CPU's y $n$ memorias, hacen falta $n_2$ conmutadores en los puntos de cruce. Si $n$ es grande la

posibilidad de construcción del equipo se complica. Véase en la Figura 2.11 un ejemplo de Cross-bar de 6 x 6.

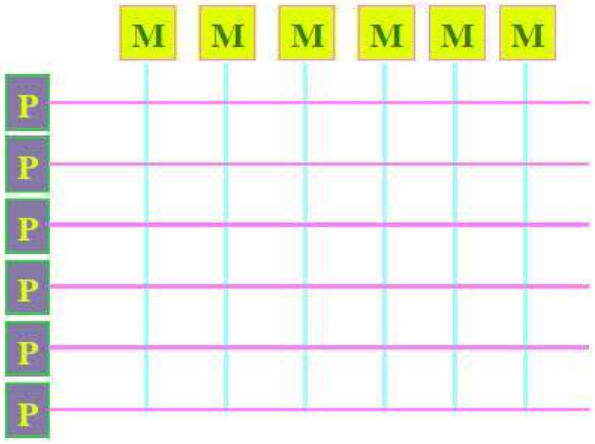


Figura 2.11 cross-bar 6x6

## 2.4 Arquitectura híbrida

Los sistemas híbridos, DSM son una combinación de las dos arquitecturas vistas en las secciones 2.2 y 2.3.

El concepto de memoria compartida distribuida combina las mejores características de las multicomputadoras y de los multiprocesadores por lo que es considerado como una arquitectura con mayor escalabilidad que los dos primeros.

Se puede ver a la arquitectura híbrida como un arreglo de procesadores con una memoria de dos niveles. La primera capa de la memoria es un nodo que consiste en un sistema de memoria compartida. Para complementar este sistema, se interconectan varios de estos nodos por medio del segundo nivel de memoria, siendo este nivel globalmente compartido por todos los nodos y por ende, por todos los procesadores.

En la Figura 2.12 se muestra un esquema típico de este modelo de procesamiento en paralelo.

Figura 2.12 Memoria compartida distribuida

Los sistemas DSM pueden clasificarse en las siguientes arquitecturas:

**DVSM** (*Distributed Virtual Shared Memory*). En este tipo de sistemas el espacio compartido de direcciones es proporcionado por el sistema operativo o por el compilador.

**NUMA** (*Non Uniform Memory Access*). En este caso cada uno de los nodos que están formando al sistema de memoria compartida distribuida contienen uno o varios procesadores con una memoria cache y un módulo de memoria que almacena una parte del espacio compartido de direcciones. Esta parte de direcciones puede accederse desde cualquier nodo, sin embargo el tiempo de acceso variará de acuerdo a si éste se produce desde el nodo local o desde un nodo remoto.

**CC-NUMA** (*Cache-coherent Non Uniform Memory Access*). En estos sistemas se busca reducir el tiempo de acceso a los datos utilizando en un tercer nivel memoria cache para almacenar datos remotos y así poder atender a las solicitudes de nodos remotos y agilizar el tiempo de acceso.

**COMA** (*Cache Only Memory Architecture*). En este sistema los módulos de memoria almacenan todo el conjunto de trabajo utilizado por cada procesador, de tal manera que se pueden ver como grandes memorias cache. Al contrario de lo que sucede con las arquitecturas NUMA y CC-NUMA, en este sistema el bloque se traslada hacia los nodos que requieran algún dato que esté dentro del bloque.

## 2.5 Conclusiones

Este capítulo ha descrito en forma detallada las principales arquitecturas computacionales usadas en la implementación en programas paralelos. Se han mostrado las principales ventajas así como las principales desventajas de cada una de ellas.

# Capítulo 3

# Plataformas para procesamiento en paralelo

## 3.1 Introducción

La utilización de máquinas paralelas en el campo de científico y de ingeniería se ha incrementado aceleradamente en los últimos años. Muchas aplicaciones secuenciales de altos requerimientos computacionales están siendo migradas a plataformas paralelas. Cada una de las plataformas de procesamiento en paralelo tienen una peculiaridad en el trabajo desarrollado sobre las aplicaciones que las hace distinguirse entre sí además, se deben tomar algunas consideraciones antes de pensar en qué plataforma usar, como por ejemplo qué hardware está disponible [Chapman. 2008]. En este capítulo se presentan nociones básicas de cuatro plataformas para este tipo de procesamiento las cuales son:

- MPI
- PVM
- THREADS
- OPENMP

En el presente proyecto de tesis se presentan los elementos básicos de las funciones de MPI, PVM y Threads, mientras que para la implementación paralela de las operaciones matriciales se utiliza OpenMP, por lo que el estudio acerca de ésta Interfaz se hace de una manera más detallada que las demás.

## 3.2 Interfaz de paso de mensajes, MPI

La primera versión de MPI fue publicada en mayo de 1994. Esta especificación fue desarrollada por el *MPI Forum.* A partir de la especificación aparecieron varias implementaciones libres, de las que las más difundidas son MPICH y LAM-MPI. En el proceso de la estandarización de MPI participaron 60 personas de 40 organizaciones,

principalmente de Estados Unidos y de Europa. La interfaz ha evolucionado y en la actualidad está disponible la versión MPI2 y OpenMPI, que es una distribución de código abierto MPI2 [MPI Forum 2009].

### 3.2.1 Comunicaciones punto a punto

Para que los procesos se comuniquen entre sí se utilizan las comunicaciones punto a punto. En el código que se muestra en la Figura 3.1 aparece un ejemplo de este tipo de comunicaciones.

```
#include "mpi.h“
main( argc, argv )
int argc;
char **argv;
        {
        char message[20];
        int myrank;
        MPI_Status status;
        MPI_Init( &argc, &argv );
        MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
        if (myrank == 0) /* código para proceso cero */
                {
                strcpy(message,"Hello, world");
        MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
                }
                else /* código para el proceso 1*/
                {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
              printf("recibido :%s:\n", message);
                 }
            MPI_Finalize();
        }
```

Figura 3.1. Ejemplo de comunicación punto a punto.

En este ejemplo el proceso cero (*myrank = 0*) envía un mensaje al proceso uno usando la función MPI_SEND. La operación especifica que un dato será enviado desde la memoria. El buffer de envío consiste de un almacenamiento conteniendo la variable *message* en la memoria del proceso cero. La localización, tamaño y tipo de buffer de envío están especificados por los primeros tres parámetros de la operación de envío.

El proceso uno (*myrank = 1*) recibe este mensaje con la operación *MPI_RECV*. En este caso, el buffer receptor consiste de un almacenamiento conteniendo la cadena de caracteres *message* en la memoria del proceso uno. Los primeros tres parámetros de la operación de recepción especifican la localización, tamaño y tipo del buffer receptor. La sintaxis de las funciones es:

- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Los parámetros tienen el siguiente significado:

- *buf* contiene el inicio de la zona de memoria de la que se van a tomar los datos a enviar o donde se van a almacenar los datos que se reciben.
- *count* contiene el número de datos a enviar o el espacio disponible para recibir.
- *datatype* es el tipo de los datos a transferir, y tiene que ser un tipo mpi.
- (*MPI_Datatype*), que en este caso es MPI_CHAR.
- *dest* y *source* son el identificador del proceso al que se envía y del que se recibe el mensaje.
- El parámetro *tag* se utiliza para diferenciar entre mensajes, y tiene que coincidir su valor en el proceso que envía y el que recibe.
- *comm* es el comunicador dentro del cual se hace la comunicación.
- *status* hace referencia a una variable de tipo *MPI_Status.*

En los mensajes hay que utilizar tipos de datos *mpi*. Los posibles tipos de datos se muestran en la Tabla 3.1, donde se puede observar la comparación con el tipo de datos del lenguaje C.

Tabla 3.1 Tipos de datos para MPI.

| Tipo MPI | Tipo C |
|---|---|
| MPI_CHAR | Signed char |
| MPI_SHORT | Signed short int |
| MPI_INT | Signed int |
| MPI_LONG | Signed long int |
| MPI_UNSIGNED_CHAR | Unsigned char |
| MPI_UNSIGNED_SHORT | Unsigned short |
| MPI_UNSIGNED | Unsigned int |
| MPI_UNSIGNED_LONG | Unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | Long double |
| MPI_BYTE | |
| MPI_PACKED | |

MPI proporciona dos tipos de comunicación: bloqueante y no bloqueante. Esta modalidad está relacionada con el tiempo que un proceso tiene que esperar al haber solicitado una función de comunicación. Si una función mantiene a un proceso bloqueado hasta que la operación solicitada termine, se le llama función bloqueante. Por otro lado, si la función termina inmediatamente después de haber iniciado la comunicación se le llama función no bloqueante.

Las funciones para la comunicación no bloqueante son MPI_Isend y MPI_Irecv. En algunas ocasiones el proceso receptor debe recibir el mensaje para poder continuar con la ejecución del proceso. Para poder determinar cuánto tiempo debe esperar existe la función MPI_Wait y MPI_Test para comprobar que la operación ha sido completada.

### 3.2.2 Comunicaciones colectivas

Las comunicaciones colectivas están definidas como comunicaciones que involucran a un grupo o grupos de procesos. En este tipo de operaciones la eficiencia de las aplicaciones aumenta considerablemente con respecto a la comunicación punto a punto.

Las funciones de este tipo proporcionadas por MPI están a continuación:

- *int MPI_Barrier(MPI_Comm comm)* establece una barrera. Los procesos esperan a que todos lleguen a la barrera, para continuar la ejecución.
- *int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)* realiza una operación de *broadcast* (comunicación uno a todos), donde se mandan *count* datos del tipo *datatype* desde el proceso raíz *(root)* al resto de procesos en el comunicador.
- *int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)* realiza una reducción todos a uno.

El tipo de operaciones que se admiten en las comunicaciones colectivas están en la Tabla 3.2.

Tabla 3.2 Datos admitidos por las funciones MPI.

| **Operación** | **Significado** | **Tipos permitidos** |
|---|---|---|
| MPI_MAX | Máximo | Enteros y punto flotante |
| MPI_MIN | Mínimo | Enteros y punto flotante |
| MPI_SUM | Suma | Enteros y punto flotante |
| MPI_PROD | Producto | Enteros y punto flotante |
| MPI_LAND | AND lógico | Enteros |
| MPI_LOR | OR lógico | Enteros |
| MPI_LXOR | XOR lógico | Enteros |
| MPI_BAND | Bitwise AND | Enteros y Bytes |
| MPI_BOR | Bitwise OR | Enteros y Bytes |
| MPI_BXOR | Bitwise XOR | Enteros y Bytes |
| MPI_MAXLOC | Máximo y localización | Parejas de tipos |
| MPI_MINLOC | Máximo y localización | Parejas de tipos |

- Cuando todos los procesos tienen que recibir el resultado de la operación se usa la función *int MPI_Allreduce(void *sendbuf,void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm).*
- Para enviar desde un proceso mensajes distintos al resto de procesos se puede usar *int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount,MPI_Datatype recvtype, int root, MPI_Comm comm)*. En el

proceso raíz el mensaje se divide en segmentos de tamaño *sendcount*, y el segmento *i*-ésimo se envía el proceso *i*.

- La inversa de la función *MPI_Scatter es int MPI_Gather( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm).* Todos los procesos (incluido el raíz) envían al proceso raíz *sendcout* datos de *sendbuf*, y el raíz los almacena en *recvbuf* por el orden de los procesos.
- Para mandar bloques de datos de todos a todos los procesos se usa *int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm),* donde el bloque enviado por el *i*-ésimo proceso se almacena como bloque *i*-ésimo en *recvbuf* en todos los procesos. Para enviar bloques de tamaños distintos se usa *MPI_Allgatherv.*
- Para mandar bloques de datos distintos a los distintos procesos se utiliza *int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount,MPI_Datatype recvtype, MPI_Comm comm)*. De cada proceso *i*, el bloque *j* se envía al proceso *j*, que lo almacena como bloque *i* en *recvbuf*. Existe el correspondiente *MPI_Alltoallv*.

Uno de los argumentos claves en una función para comunicaciones colectivas es un comunicador que define un grupo o grupos de procesos participantes y proporciona un contexto para las operaciones. Existen dos tipos de comunicadores:

1. Intracomunicador. Se puede ver como un identificador para un solo grupo de procesos unido a un contexto.
2. Intercomunicador. Identifica a dos grupos distintos de procesos unidos a un contexto.

## 3.3 Máquina virtual paralela, PVM

El proyecto PVM empezó en el verano de 1989 en el *Oak Ridge National Laboratory.* El prototipo del sistema PVM 1.0, fue construido por *Vaidy Sunderam* y *All Geist;* Esta versión del sistema fue usado internamente en el laboratorio y no fue publicado.

La versión 2 de PVM fue escrita en la Universidad de Tennessee y publicada en Marzo de 1991. Durante el siguiente año, PVM comenzó a usarse en varias aplicaciones científicas. Después de una revisión y algunos cambios surgió la versión de PVM 2.1-2.4, fue entonces cuando una completa reescritura fue emprendida, y la versión 3 fue completada en Febrero de 1993. La versión 3.3 es la que se describe en este capítulo, a la que se hace referencia simplemente como PVM. El software ha sido distribuido libremente y ha comenzado a usarse en aplicaciones científicas alrededor del mundo.

Las tareas PVM pueden poseer control arbitrario y estructuras de dependencia. En otras palabras, en cualquier punto de la ejecución en una aplicación concurrente, cualquier tarea en existencia puede iniciar o parar otras tareas, agregar o borrar computadoras de la máquina virtual. Cualquier proceso puede comunicarse y/o sincronizarse con otro. El control específico y estructura de dependencia pueden implementarse bajo el sistema PVM mediante un uso adecuado de estructuras de PVM.

### 3.3.1 El sistema PVM

El sistema PVM se compone de dos partes. La primera parte es un demonio *(daemon),* abreviado como *pvmd* que está en todas las computadoras de la red haciendo una máquina paralela virtual. La segunda parte del sistema es una librería de rutinas de la interfaz de PVM. Contiene un repertorio funcionalmente completo de funciones que son necesarias para la cooperación entre tareas en una aplicación [Geist 1994]. Los objetivos generales de este proyecto son investigar problemas y desarrollar soluciones para el procesamiento concurrente en arquitecturas de memoria distribuida. Esta librería contiene rutinas que permite al usuario el paso de mensajes, coordinación de tareas y modificación de la máquina virtual.

El modelo de procesamiento de PVM está basado en la noción de que una aplicación consiste de varias tareas. Cada una de las cuales es responsable de una parte del trabajo de la aplicación computacional. Un método más común de la paralelización de una

aplicación es conocido como “paralelismo de datos”. En este método todas las tareas son una sola, pero cada una solo conoce y resuelve una pequeña parte del problema entero. Un diagrama del modelo de procesamiento de PVM se muestra en la Figura 3.3.

En cada una de las computadoras donde se está ejecutando PVM, y por cada uno de los usuarios, existe un proceso *daemon,* que se comunica con todos los procesos PVM de ese usuario que están ejecutándose en esa computadora en cada momento. En el caso de que se quieran comunicar procesos que se estén ejecutando en el mismo procesador lo harán a través de su proceso *pvmd.* Si los procesos se ejecutaran en procesadores diferentes, cada proceso se comunicará con su *pvmd,* y luego los diferentes procesos *pvmd* correspondientes a los diferentes procesadores, se comunicarán a través de la red.

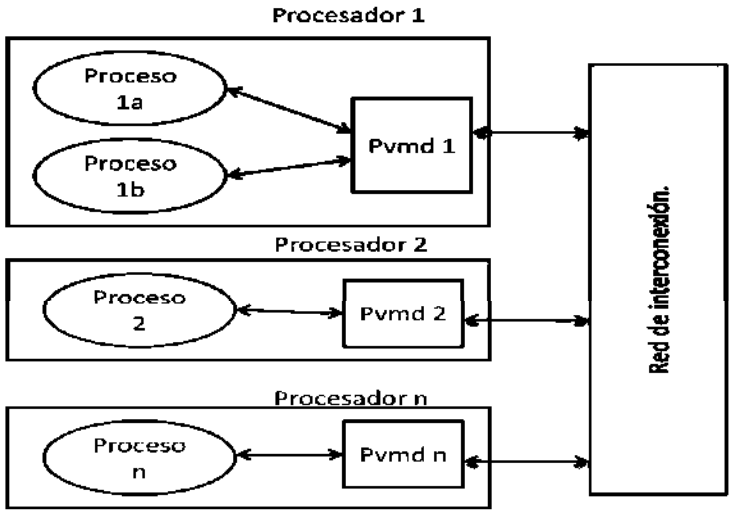


Figura 3.2. Modelo de PVM.

Cada proceso que funciona bajo PVM posee un número que le caracteriza; ese número, llamado identificador del proceso, puede obtenerse llamando a la función *(pvm_mytid()).* Este identificador de proceso es necesario para que los demás procesos se comuniquen con él, enviando o recibiendo mensajes. Las rutinas de envío y recepción de mensajes emplean a los identificadores para seleccionar el proceso de destino u origen del mensaje respectivamente.

### 3.3.2. Estructura de un programa con PVM

Para implementar un programa en PVM primero que nada se debe incluir la librería *pvm3.h.*

Una vez que se ha incluido el archivo de cabecera *pvm3.h* se debe inicializar PVM. Para hacerlo primeramente se deben de relacionar los procesos de PVM. Esto se hace conociendo el identificador de proceso mediante la función *pvm_mytid()*.

Posteriormente se deben de poner en marcha todos los procesos que van a trabajar en paralelo. Esto se consigue mediante la función *pvm_spawn(),* con lo que se retorna un vector con los identificadores de los procesos creados.

Por cada mensaje que se envía se debe de reiniciar el buffer de mensaje enviado mediante la llamada a *pvm_initsend()*. Luego se tienen qué desempaquetar todos los mensajes que se vayan a enviar con la función *pvm_pk<tipo>* donde *tipo* es uno de los tipos de datos del Lenguaje C.

La función utilizada para recibir los mensajes es *pvm_recv()*. Después es necesario desempaquetar el mensaje con la función *pvm_upk<tipo>*. Por último terminar la ejecución del programa utilizando la función *pvm_exit()*. Para ejemplificar lo dicho anteriormente se presenta el código de la Figura 3.3.

```
#include "pvm3.h"
        main()
        {
                int cc, tid, msgtag;
                char buf[100];
                printf ("i' m t%x\n", pvm_mytid());
                cc = pvm_spawn("hello_other", (char**) 0, 0 "",1, &tid);
                        if (cc == 1)
                        {
                        msgtag = 1;
                        pvm_recv (tid, msgtag);
                        pvm_upkstr (buf);
                        printf ("from t%x: %s\n", tid, buf);
                        }
                                else
                                printf("can't start hello_other\n");
                                pvm_exit();
        }
```

Figura 3.3. Estructura de un programa en PVM.

Este programa está hecho para ser invocado manualmente; después de imprimir su identificador *id* correspondiente a la tarea, obtenido con *pvm_mytid()*. Esta función inicia una copia de otro programa llamado *hello_other* usando la llamada *pvm_spawn()*. Esto causa que el programa ejecute un recibimiento bloqueante usando *pvm_recv.* Después recibiendo el mensaje, el programa imprime el mensaje enviado por su contador de partes, así como su identificador de tarea; el buffer es extraído desde el mensaje usando *pvm_upkstr.* Finalmente el programa termina con la función *pvm_exit.*

## 3.4 Pthreads

*Pthreads* es una plataforma de procesamiento en paralelo que ha sido desarrollada para entornos multiprocesador.

La librería de Pthreads es una librería que cumple los estándares POSIX y que permite trabajar con distintos hilos de ejecución (*threads*) al mismo tiempo.

La diferencia entre un thread y un proceso es que los procesos no comparten memoria entre sí a no ser que se haya declarado explícitamente usando alguno de los mecanismos de IPC (*InterProcess Communication*) de *Unix*, mientras que los threads sí comparten totalmente la memoria entre ellos. Para crear threads se usan las funciones de la librería *pthread* o de cualquier otra que soporte threads mientras que para crear procesos se usará la llamada al sistema *fork()*, que se encuentra en todos los sistemas *unix* [Pthreads 2010].

### 3.4.1 creación de threads

El programa implementado en paralelo inicia como un solo thread. Así que para crear un thread se necesita la función *pthread_create* de la librería y de la estructura de datos *pthread_t* que identifica cada thread diferenciándolo de los demás y que contiene todos sus datos.

La sintaxis de la función es la siguiente:

*pthread_create()*

La sintaxis es:

*pthread_create (thread,attr,start_routine,arg)*

- *pthread_create* crea un nuevo hilo y hace su ejecutable. Esta rutina puede ser llamada cualquier número de veces dentro del código desde donde sea.

- Argumentos de *pthread_create:*
    - *thread:* es un identificador para el nuevo hilo retornado por la subrutina.
    - *attr:* Es un objeto que puede ser usado para establecer ciertas características ó atributos a los hilos. Entonces se pueden especificar las características que debe de tener un hilo o se le puede asignar atributos predeterminados por medio de la instrucción *NULL.*
    - *start_routine:* Es una rutina de C que será ejecutada una vez que se haya creado el hilo.
    - *arg:* Es un argumento que puede ser pasado a *start_routine.* Debe ser pasado a referencia como puntero al parámetro que se le pasará la función.

- El máximo número de hilos que puede ser creado por un proceso es una implementación dependiente.
- Una vez creados, estos pueden crear otros hilos. No existe jerarquía implícita o dependencia entre los hilos como se muestra en la Figura 3.4

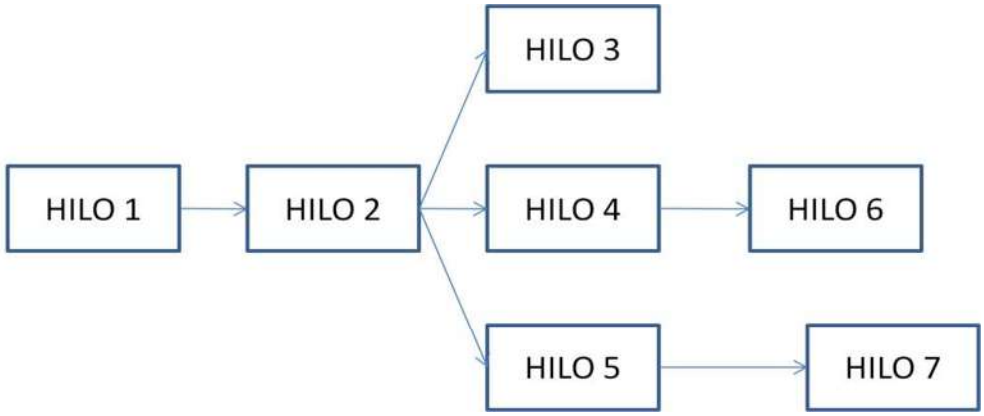


Figura 3.4 Creación de hilos a partir de un hilo existente.

### 3.4.2 Atributos de threads

De manera predeterminada, un hilo es creado con ciertos atributos. Algunos de estas características pueden ser modificadas por el programador por medio del objeto de atributos de los hilos. Las funciones *pthread_attr_init* y *pthread_attr_destroy* son usadas para inicializar y destruir objetos de atributos respectivamente.

### 3.4.3 Terminando threads

Existen varias maneras en las cuales un hilo puede ser terminado:

- El hilo regresa a su rutina inicial, es decir a su rutina de origen.
- El hilo hace un llamado a la subrutina *pthread_exit.*
- El hilo es cancelado por medio de otro hilo usando la función *pthread_cancel.*
- El proceso entero se termina debido a una llamada a las subrutinas de salida.

La función *pthread_exit* es usada para explícitamente terminar un hilo. Típicamente, la rutina es llamada cuando un hilo ha terminado con su trabajo y ya no es necesario que siga existiendo en la aplicación.

Si *main()* finaliza antes que los hilos que ha creado, y existe con *pthread_exit(),* los otros hilos continuarán su ejecución. Por otra parte ellos serán terminados cuando *main* finalice.

El programador puede opcionalmente especificar un estado de terminación, el cual es guardado como un indicador nulo para cualquier hilo que pueda unir el hilo de la llamada.

*Cleanup*: la rutina *pthread_exit()* no cierra archivos; cualquier archivo abierto dentro del hilo permanecerá abierto después de que los hilos hayan terminado.

Para esperar a que todos los hilos terminen su ejecución respectiva se utiliza la función *pthread_join()*. La sintaxis de la función es

```
int pthread_join (pthread_t th, void **thread_return)
```

Los parámetros de la función `pthread_join` son:

*th:* Es el identificador del thread que se quiere esperar, y es el mismo que se obtuvo al crearlo con *pthread_create.*

*thread_return:* es un puntero a puntero, que apunta (valga la redundancia) al resultado devuelto por el thread que estamos esperando cuando terminó su ejecución. Si este parámetro es *NULL*, quiere decir que se le está indicando a la librería que no importa el resultado.

Por otra parte la función *pthread_tpthread_self()* devuelve al thread que la llama su información, en forma de variable de tipo *pthread_t.* Es útil si el propio thread que se está

ejecutando quiere cambiar sus atributos, hacerse él mismo un pthread_detach, etc. La sintaxis de esta función es:

```
pthread_t pthread_self (void)
```

## 3.5 OPENMP

OpenMP realiza un modelo de programación de memoria-compartida. Este modelo asume, como su nombre lo indica, que los programas serán ejecutados en uno o más procesadores que comparten parcial o totalmente la memoria permitida como se muestra en la Figura 3.5. Los programas de memoria compartida son típicamente ejecutados por múltiples hilos independientes (estados de ejecución que pueden procesar una instrucción de cadena); los hilos comparten datos, pero también tienen algunos adicionales, datos privados. El enfoque de memoria compartida de programación en paralelo debe proporcionar, en adición a una normal rama de instrucciones, un medio para poner en marcha los hilos, asignando trabajo a ellos, y coordinando su acceso a datos compartidos, incluyendo asegurar que ciertas operaciones son realizadas por un solo hilo a la vez [Chapman 2008].

Antes de continuar con el estudio de OpenMP es conveniente mostrar qué es un thread. Técnicamente un thread está definido como una cadena independiente de instrucciones que pueden ser programadas para ejecutar sobre el sistema operativo.
Un proceso en un sistema operativo tradicional es un espacio de direcciones y un hilo de control. Un proceso es un contenedor de recursos con un hilo en ejecución.

Un hilo se ejecuta dentro de un proceso. Los hilos son entidades panificables para su ejecución sobre un procesador. Los hilos añaden a los procesos la capacidad de ejecutar varios conjuntos de instrucciones de forma concurrente dentro de un mismo entorno de procesamiento.

Las dos principales formas de aprovechamiento para asignación de trabajo en threads, son:

- Niveles de ciclos paralelos.
- Regiones paralelas.

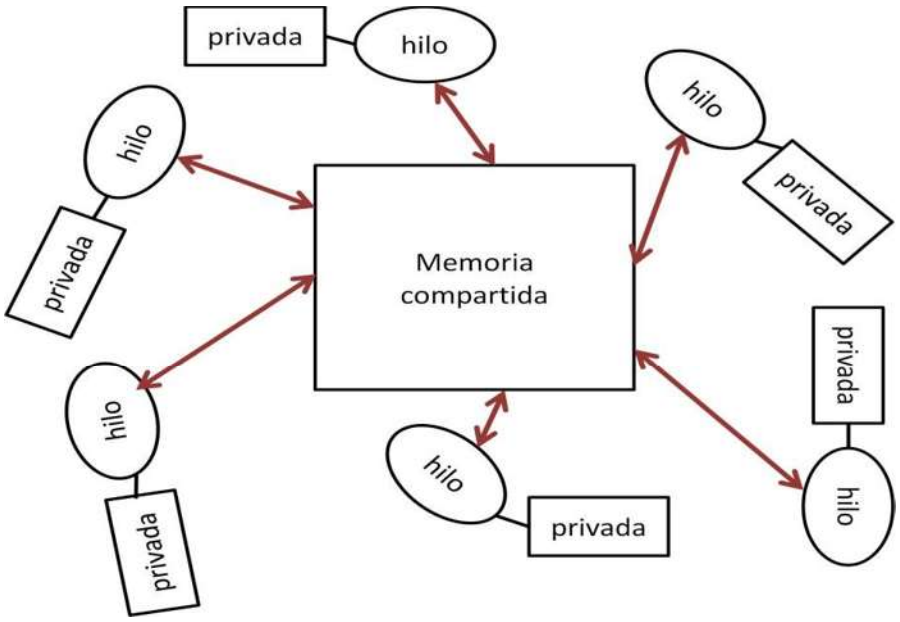


Figura 3.5 Modelo de memoria compartida.

### 3.5.1 Elementos básicos

OpenMP comprende de tres componentes complementarios:

- Un conjunto de directivas de compilación usadas para la comunicación.
- Una librería de funciones en tiempo de ejecución que habilita los parámetros paralelos que se van a usar, tal como número de threads que van a participar y el número de cada threads.
- Un número limitado de las variables de entorno que pueden ser usadas para definir en tiempo de ejecución parámetros del sistema en paralelo tales como el número de threads.

### 3.5.2 Compilación y ejecución

Para la compilación y ejecución de los programas implementados con OpenMP es necesario disponer de un compilador que pueda interpretar los pragmas necesarios para la paralelización de aplicaciones. (El gcc es el que se utilizó para el desarrollo de esta tesis,

por lo que en este caso se hará referencia precisamente a cómo se compila y ejecuta con gcc). La sintaxis de la compilación de los programas es

```
gcc –o nombre_del_codigo nombre_del_codigo.c -fopenmp
```

Con esto se genera nombre_del_codigo, que se podrá ejecutar en paralelo poniendo en marcha varios hilos. La ejecución se realiza de la misma manera en todos los programas escritos con OpenMP, pero hay que determinar cuántos hilos intervendrán en el ciclo o región paralela. Hay una variable de entorno `OMP_NUM_THREADS` que indica ese número. Si no se inicializa esa variable tendrá un valor predeterminado, que suele coincidir con el número de núcleos con que se esté trabajando. Antes de la ejecución se puede establecer el valor de la variable. Una posibilidad de utilizar esta variable es `OMP_NUM_THREADS=6` con lo que se está estableciendo que el número de hilos en la región paralela será de 6, independientemente del número de núcleos de que se disponga.

A continuación se presentan algunas otras variables de entorno, con la función que realizan en tiempo de ejecución:

- *int omp_get_num_threads(void)* Devuelve cuántos threads se están utilizando
- *int omp_get_num_procs(void)* Devuelve el número de procesadores accesibles
- *void omp_set_num_threads(int)* Indica el número de threads a utilizar
- *int omp_get_max_threads(void)* Devuelve el máximo posible de threads

### 3.5.3 Paralelización de ciclos

La directiva #pragma omp for prepara las iteraciones que están justo debajo de ella para que sean ejecutadas en paralelo. En tiempo de ejecución las iteraciones son distribuidas entre los hilos. La sintaxis para c/c++ se muestra en la siguiente figura.

```
#pragma omp for [clause][,][clause]….]
```

Las cláusulas que se pueden utilizar son las siguientes:

- private(lista de variables)
- firstprivate(lista de variables)
- lastprivate(lista de variables)
- reduction(operador:lista de variables)

En seguida se muestra una explicación breve de lo que realizan las cláusulas:

- *private(lista)*: las variables de la lista son privadas a los threads, lo que quiere decir que cada thread tiene una variable privada con ese nombre. Las variables no se inicializan antes de entrar y no se guarda su valor al salir.
- *firstprivate(lista)*: las variables son privadas a los threads, y se inicializan al entrar con el valor que tuviera la variable correspondiente.
- *lastprivate(lista):* son privadas a los threads, y al salir quedan con el valor de la última iteración si se encuentra en un bucle for paralelo.
- *reduction(operador:lista)*: las variables de la lista se obtienen por la aplicación del operador, que debe ser asociativo.

▪ Puede aparecer una cláusula Schedule para indicar la forma cómo se dividen las iteraciones del for entre los threads, que puede ser:

- *schedule(static,tamaño):* Las iteraciones se dividen según el tamaño que se indica.
- *schedule(dynamic,tamaño):* Las iteraciones se agrupan según el tamaño y se asignan a los threads dinámicamente cuando van finalizando las tareas que le fueron asignadas su trabajo.
- *schedule(guided,tamaño):* Las iteraciones se asignan dinámicamente a los threads pero con tamaños decrecientes hasta llegar al tamaño que se indica (si no se indica nada el valor por defecto es 1).
- *schedule(runtime):* Deja la decisión para el tiempo de ejecución, y se obtiene de la variable de entorno OMP_SCHEDULE.

La sintaxis del ciclo for se muestra enseguida

for (*init-expr; var relop b*; *incr-expr*)

En la Figura 3.6 se presenta un ejemplo de cómo se paraleliza un ciclo for:

```
#pragma omp parallel shared (n) private (i)
        {
        pragma omp for
        for (i=0;i<n;i++)
        printf("thread %d executes loop iteration %d\n",omp_get_thread_num(), i);
        }/*--end of parallel region --*/
```

Figura 3.6. Paralelización de un ciclo for

En esta ocasión se utiliza la función de OpenMP *omp_get_thread_num()* para obtener e imprimir el número de hilo ejecutante en cada iteración. Así como se utiliza esta función que define el número de hilos en tiempo de ejecución, existen también algunas otras que se verán en la siguiente sección.

### 3.5.4 Regiones paralelas

Anteriormente se habló de la paralelización de ciclos. En esta sección se hablará de los elementos que se requieren para paralelizar una región de código, que previamente se haya escrito en forma secuencial. Al mismo tiempo se explicarán algunas de las características de estos bloques de código.

Como se describió en la sección asociada con la paralelización de ciclos, estos comienzan con *#pragma omp for*. En general OpenMP comienza a través de directivas, con funciones propias (que pueden ser condicionalmente compiladas) y variables de entorno, que pueden modificar el comportamiento en tiempo de ejecución.

La directiva de formato que utiliza OpenMP es *#pragma omp* seguida de una palabra específica que identifica el nombre de la directiva, con posiblemente una o más cláusulas, como sigue:

```
#pragma omp directive-name [clause[[,]clause]….] new line
```

La cláusula puede ser:

- if(expresión escalar)
- private(lista de variables)
- firstprivate(lista de variables)

- default(shared | none)
- shared(lista de variables)
- copyin(lista de variables)
- reduction(operador:lista de variables)
- num_threads(expresión entera)

Entre las cláusulas que se pueden utilizar, las más usadas son:

- *private(lista de variables)* En esta la lista de variables puede ser separada por "," y para cada una de ellas se genera una copia en cada thread, esta copia no tiene relación con la original y no es inicializada a menos que se utilice *firstprivate*.
- *shared(lista de variables)* En esta las variables de la lista son comunes a todos los threads y cada uno de ellos puede modificarla afectándola en forma global.
- *threadprivate(lista de variables)* Hace que la lista sea privada a cada thread pero globales dentro de un thread.
- *reduction(operador:lista de variables)* Realiza una operación de reducción sobre las variables que aparecen en la lista utilizando el operador/intrínseco especificado. El operador puede ser: +, *, -, &(and), |(or), ^(eqv), &&(neqv). El intrínseco puede ser: || (max), (min), (and), (or).

Al final de una región paralela hay una sincronización implícita. Solo el thread master continúa la ejecución. Cuando un thread encuentra un constructor paralelo, un grupo de threads es creado si uno de los siguientes casos es verdadero:

- No está la cláusula if presente.
- Si al evaluar la expresión de if tiene un valor diferente de cero.

Este thread se convierte en el thread maestro de un grupo, con el número de thread 0, y todos los threads, incluyendo el maestro ejecutan la región paralela. Si el valor de la expresión *if* se hace cero la región es serializada nuevamente. Para determinar el número de threads necesarios, las siguientes reglas deben ser consideradas en orden. La primera regla que cumple la condición es aplicada:

1. Si la cláusula *num_threads* está presente, entonces la expresión entera en la cláusula es el número de threads requerido.

2. Si la función *omp_set_num_threads* de la librería es llamada, entonces el valor del argumento es el número de threads requeridos.
3. Si la variable de entorno *OMP_NUM_THREADS* es definida entonces el valor de esta variable de entorno es el número de threads requeridos.
4. Si ninguno de los métodos anteriormente mencionados es usado, entonces el número de threads requeridos en el definido por defecto.

### 3.5.5 Variables privadas y variables públicas

La forma en que se declara una variable privada es con la cláusula *shared (list)*. En este caso todos los elementos de *list* serán compartidos entre los diferentes hilos en el equipo.

En la Figura 3.7 se muestra un fragmento de código para ejemplificar cómo se declara una variable como compartida.

```
#pragma omp parallel for shared (a)
        for (i=0; i<n; i++)
                a[i] += i;
        /*-- end of parallel for--*/
```

Figura 3.7 Declaración de variable compartida.

En este ejemplo el vector *a* está declarado para ser compartido, esto implica que todos los hilos pueden leer y escribir sobre el vector *a*. Cada uno de los hilos podrá tener acceso al valor anterior de *a[i]*.

Por otra parte la sintaxis para declarar una variable como privada es *prívate (list).* Cada variable en *list* es reproducida, así que cada hilo en el equipo de hilos tiene acceso exclusivo a una copia local de esta variable. Los cambios hechos por un hilo en el dato, no son vistos por otros hilos.

Un ejemplo de cómo se declara como privada una variable se muestra en la siguiente parte de código.

```
#pragma omp parallel for prívate (i, a)
        for (i = 0; i < n; i++)
          {
            a  = i + 1;
            printf("thread %d has a value of a = %d for i = %d\n", omp_get_thread_num (), a, i);
          }
       /*-- end of parallel for --*/
```

Figura 3.8. Declaración de una variable privada.

En este ejemplo existen dos variables que están declaradas como privadas las variables ***i*** y ***a*** si la variable ***a*** ha sido declarada con una cláusula compartida (*shared*) múltiples hilos podrían intentar actualizar la misma variable con diferentes valores de una manera incontrolable.

### 3.6 conclusiones.

En este capítulo se han descrito así como analizado las principales características de las diversas plataformas operativas utilizadas en la implementación de programas en paralelo. Se han mostrado sus ventajas así como desventajas.

# Capítulo 4

# Propuesta de paralelización para las operaciones matriciales

## 4.1 Introducción

En análisis de problemas de ingeniería frecuentemente se utilizan diversos tipos de representaciones para los mismos. Una de las representaciones más comúnmente utilizada es la representación matricial. Por ejemplo un sistema eléctrico puede ser representado mediante una matriz de admitancias la cual representa la conectividad que existe entre los diversos elementos que forman el sistema eléctrico. Una vez que se tiene la representación matricial, el ingeniero busca solucionar el problema. Esta solución generalmente implica el uso de operaciones matriciales básicas tales como sumas, restas, multiplicaciones, etc.

Generalmente las matrices asociadas a la representación de los sistemas, particularmente los eléctricos, son de grandes dimensiones, máxime si se tiene una representación muy a detalle del sistema. Esta representación detallada implica desde el punto de vista computacional dos grandes retos: el manejo de memoria y el tiempo de cómputo. En esta tesis nos enfocaremos en el segundo reto.

## 4.2 Operaciones matriciales

A continuación se describen las principales operaciones matriciales. Una vez descritas estas operaciones se proponen algoritmos computacionales para realizarlas en forma secuencial y posteriormente en forma paralela usando la plataforma operativa de procesamiento en paralelo de OpenMP.

### 4.2.1 Suma de matrices

En la operación referente a la suma de matrices se suma cada uno de los elementos de la matriz *A* con el respectivo número de elemento de la matriz *B*. Dicho con otras palabras,

supóngase que $A = (a_{ij})$ y $B = (b_{ij})$ son dos matrices de *m x n*. La suma de *A* y *B* resulta en una matriz de $m \times n$ dada por

$$A + B = \left(a_{ij} + b_{ij}\right) = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \dots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \dots & a_{mn} + b_{mn} \end{pmatrix} \tag{4.1}$$

La suma de dos matrices está definida solamente cuando ambas matrices tienen el mismo tamaño. La complejidad de la suma de matrices es $O(n^2)$.

### 4.2.2 Algoritmo secuencial para la suma de matrices

Como primer algoritmo se muestra la suma de matrices de manera secuencial en la Figura 4.1. Cabe mencionar que en este proyecto, para ejemplificar cada uno de los algoritmos, se utilizará la *matriz A* y la *matriz B* como matrices de lectura y la *matriz C* será la matriz resultante de la operación correspondiente de las primeras dos. Para el caso de la suma de matrices se leen los elementos de la matriz *A* y *B,* se suman y se almacenan en la *matirz C* como se explicó en la sección referente a la operación suma. El primer ciclo *for* recorre los renglones de las matrices y el segundo las columnas.

```
for (i=0;i<N;i++)
        {
            for (j=0;j<N;j++)
            C[i*N+j]=A[i*N+j]+B[i*N+j];
        }
```

Figura 4.1 Código secuencial para la suma de matrices.

Los módulos en memoria reservados para un vector son adyacentes, mientras que los módulos reservados para una matriz no necesariamente tienen que serlo. Por lo que es mejor reservar espacio en memoria para una matriz como si fuera un vector, (así como se declaró en el código anterior) que declararlo como una serie de vectores indexados.

En la Figura 4.2 se muestra la forma en cómo se ha declarado la matriz como vector para obtener una mayor facilidad de acceso a los datos. Por otra parte la manera en que el procesador recorre los datos aparece en la Figura 4.3.

Nota: En cada uno de los algoritmos presentados a continuación se realiza el acceso a los datos en memoria tal como se mostró para el algoritmo de la suma de matrices que se escribió previamente.

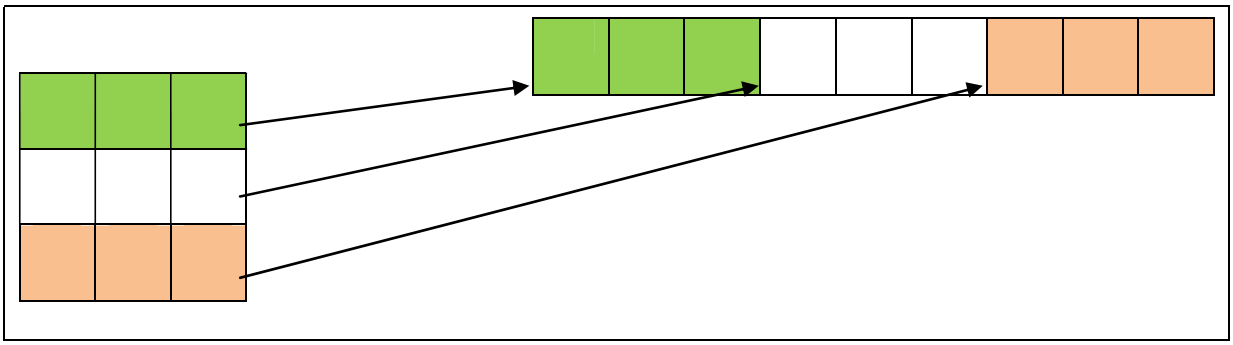

Figura 4.2. Disposición en memoria del almacenamiento de la matriz.

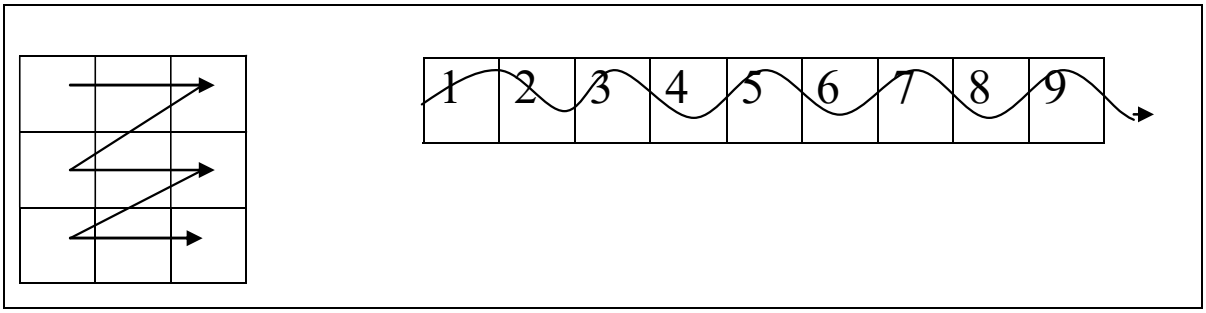


Figura 4.3 Recorrido del procesador a través de le memoria.

### 4.2.3 Resta de matrices

En la resta de matrices se hace de la misma manera que en la suma. Si suponemos dos matrices A y B, entonces se dice que para encontrar la matriz resultante, es decir la resta de estas dos, se procede como se muestra en la siguiente fórmula.

$$A - B = (a_{ij} - b_{ij}) = \begin{pmatrix} a_{11} - b_{11} & a_{12} - b_{12} & \dots & a_{1n} - b_{1n} \\ a_{21} - b_{21} & a_{22} - b_{22} & \dots & a_{2n} - b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} - b_{m1} & a_{m2} - b_{m2} & \dots & a_{mn} - b_{mn} \end{pmatrix} \quad (4.2)$$

Existen las mismas restricciones para la resta de matrices que para la suma de matrices, es decir, las matrices a sumar deben de ser del mismo tamaño. Por otro lado la complejidad de este tipo de operaciones es también $O(n^2)$ como en el caso de la suma, por lo que se esperan resultados similares a la hora de optimizar dichos códigos.

### 4.2.4 Algoritmo secuencial para la resta de matrices

A continuación se presenta una fracción de código que muestra la parte de la operación para la resta de matrices para las dos matrices inicialmente existentes.

```
for (i=0; i<N;i++)
{
        for (j=0;j<N;j++)
        C[i*N+j]=A[i*N+j]-B[i*N+j];
}
```

Figura 4.4 Código secuencial para la resta de matrices.

Como se aprecia en la Figura 4.4, la operación de la resta de matrices es similar a la suma, excepto porque en el segundo caso se realiza la sustracción en lugar de la adición de datos. Nuevamente el primer ciclo *for* realiza el recorrido las filas de las matrices y el segundo, el de las columnas.

### 4.2.5 Multiplicación de una matriz por un escalar

En este tipo de operación se realiza la multiplicación de cada uno de los elementos de la matriz por el escalar, es decir, si $A = (a_{ij})$ es una matriz de $m\ x\ n$ y $\alpha$ es un escalar, entonces la matriz $\alpha A$ de $m\ x\ n$ está dada por:

$$\propto A=\left(\propto a_{ij}\right) = \begin{pmatrix} \propto a_{11} & \propto a_{12} & \dots & \propto a_{1n} \\ \propto a_{21} & \propto a_{22} & \dots & \propto a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \propto a_{m1} & \propto a_{m2} & \dots & \propto a_{mn} \end{pmatrix} \tag{4.3}$$

Para la operación de la multiplicación de una matriz por un escalar la complejidad es simplemente igual al número de elementos de la matriz a multiplicar es decir $O(n)$.

### 4.2.6 Algoritmo secuencial para la multiplicación de una matriz por un escalar

El código secuencial para la multiplicación de una matriz por un escalar se muestra en la Figura 4.5, se puede ver que el ciclo que está inmediatamente encima de la operación de multiplicación está recorriendo las columnas de la matriz. Por su parte el otro ciclo está haciendo el barrido de los renglones de la misma matriz, y en cada iteración se va realizando la multiplicación del escalar $B$ un elemento de la matriz $A$ hasta que se han registrado todos.

```
for (i=0;i<N;i++)
        for(j=0;j<N;j++)
        C[i*N+j] = B*A[i*N+j];
```

Figura 4.5. Código secuencial para el producto matriz-escalar

### 4.2.7 Producto punto

El producto punto de dos vectores se encuentra al multiplicar el elemento i-ésimo del vector $A$ por el elemento i-ésimo del vector $B$ y posteriormente realizar la sumatoria de estos productos hasta el n-ésimo como se muestra en la ecuación (4.4). El resultado es un escalar.

$$\begin{aligned} A \cdot B &= (a_1 + a_2 \ldots + a_n) \cdot (b_1 + b_2 \ldots b_n) \\ &= a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \\ &= \sum a_i \cdot b_i \end{aligned} \tag{4.4}$$

### 4.2.8 Algoritmo secuencial para el Producto Punto

El algoritmo para la operación del producto punto es un tanto sencillo puesto que sólo es necesario multiplicar los mismos elementos en cada matriz, para encontrar el resultado. El código del programa secuencial aparece en la Figura 4.6.

```
for (i=0; i<N; i++)
        c[i] = a[i]*b[i];
```

Figura 4.6. Código secuencial para el producto punto

### 4.2.9 Producto de una matriz por un vector

Para que el producto de una matriz *A* por un vector *B* exista es necesario que el número de columnas de la matriz *A* sea igual al número de renglones del vector *B*. Por ejemplo supóngase que *A* es una matriz de *n* columnas y *B* un vector de *n* renglones. El renglón *i* de *A* es $[a_{i1}\ a_{i2}\ a_{i3}\ \ldots\ a_{in}]$ y la columna de B es $\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$. Si $C = A\ B$, entonces el producto de la matriz por el vector está dado por la ecuación (4.5)

$$c_{ij} = a_{i1}b_1 + a_{i2}b_2 + a_{i3}b_3 + \cdots + a_{in}b_n \tag{4.5}$$

### 4.2.10 Algoritmo secuencial para el producto matriz-vector

Antes de comenzar con el tema relativo a la multiplicación de matrices se presenta en esta sección la multiplicación de una matriz por un vector, para lo que se presenta el código de la Figura 4.7.

```
for(i=0;i<N;i++)
        {
                A[i]= 0;
                for (j=0;j<N;j++)
                A[i]+=B[i*N+j]*C[j];
        }
```

Figura 4.7. Código secuencial para el producto matriz-vector.

### 4.2.11 Producto de matrices

Dada una matriz $A = (a_{ij})$ de $m\ x\ n$ donde cada uno de sus elementos se denota como $a_{ij}$ con $1 \leq i \leq m$, y $1 \leq j \leq r$; y una matriz $B = (b_{ij})$ de $n\ x\ p$ donde cada uno de sus elementos se denota como $b_{ij}$ con $1 \leq i \leq n$ y $1 \leq j \leq p$; la matriz resultante de la operación de multiplicación de las matrices A y B, C = A x B, es tal que cada uno de sus elementos se denota como $c_{ij}$ con $1 \leq i \leq m$, y $1 \leq j \leq p$ y se calcula de la siguiente manera

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{4.6}$$

Como en la mayoría de las operaciones básicas de álgebra lineal, la cantidad de operaciones entre escalares, necesaria para el cálculo de la matriz resultante es conocida (calculable) de manera exacta. Tal como ha sido definida previamente la multiplicación, son necesarias exactamente

$$cant_{op} = m \cdot p \cdot (2n - 1) \tag{4.7}$$

Multiplicaciones y sumas entre escalares. Por razones de simplicidad, normalmente se hace todo el análisis en función de matrices cuadradas de orden $n$ y de esta manera se llega a que la cantidad de operaciones básicas entre escalares es exactamente

$$cant_{op} = 2n^3 - n^2 \tag{4.8}$$

Y así se encuentra que la complejidad para la multiplicación de matrices es $2n^3 - n^2$. La cual determina el tiempo de ejecución necesario para ser resuelto por la computadora. La complejidad de estas operaciones también se puede denotar como O ($n^3$), es decir "de orden $n^3$"

### 4.2.12 Algoritmo secuencial para el producto de matrices

Se sabe de los cursos de algebra lineal, que el producto de dos matrices existen si el número de columnas de la matriz *A* es igual al número de renglones de la matriz *B*. Para nuestros

casos de análisis se supondrán solo matrices cuadradas de las mismas dimensiones. En la Figura 4.8 se muestra el algoritmo para la implementación de la multiplicación de matrices de manera secuencial, en donde se puede observar un buen grado de paralelización.

```
for (m=0;m<N;m++)
                {
                        for (i=0;i<N;i++)
                        {
                                aux=0;
                                for (j=0;j<N;j++)
                                {
                                C[m*N+i]+=(A[m*N+j]*B[j*N+i]);
                                C[m*N+i]=aux;
                                }
                        }
                }
```

Figura 4.8 Algoritmo secuencial para el producto de matrices.

Si se asigna un número de ciclo a cada uno de los que aparecen en el programa comenzando con el que se encuentra por encima de las operaciones de suma y multiplicación; el ciclo número uno, recorre las filas de la matriz *B* y las columnas de la matriz *A*, suma el resultado de los productos y los almacena en la matriz *C*. El ciclo número dos va cambiando las columnas de la matriz *B* y de *C*. Por último el ciclo número tres va recorriendo las filas de la matriz *C* y de la matriz *A* como se vaya requiriendo.

## 4.3 Propuesta de paralelización de las operaciones matriciales

Una vez que se tiene el conocimiento de cómo resolver problemas que involucren operaciones matriciales, es de mucha ayuda construir un programa que realice este tipo de operaciones de una manera rápida.

A continuación se presenta una propuesta de paralelización de las operaciones básicas matriciales.

### 4.3.1 Algoritmo paralelo para la suma de matrices

En la Figura 4.9 se presenta la propuesta de paralelización de la suma de matrices

```
#pragma omp parallel shared(A,B,C) private(i,j)
{
        #pragma omp for
        for (i=0;i<N;i++)
        {
                for (j=0;j<N;j++)
        C[i*N+j]=A[i*N+j]+B[i*N+j];
        }/*fin de la región paralela*/
}
```

Figura 4.9. Propuesta de paralelización de la suma de matrices.

En este caso se ha declarado la región paralela en la cual se están declarando las matrices *A, B,* y *C* como variables compartidas para que así puedan todos los hilos tener acceso en cualquier momento a los datos que se encuentran dentro. Las variables *i* y *j* se hacen privadas, así cada hilo cuenta con una variable *i* y una variable *j*.

En la Figura 4.10 se observa cómo es que se realiza la repartición del trabajo entre los hilos. El hilo 0 recorre un bloque de filas que son sumados con el bloque de filas registrado por el hilo 4. El hilo 1 hace lo mismo y suma sus datos con los datos del hilo 5, y así sucesivamente. Después de la suma termina la región paralela.

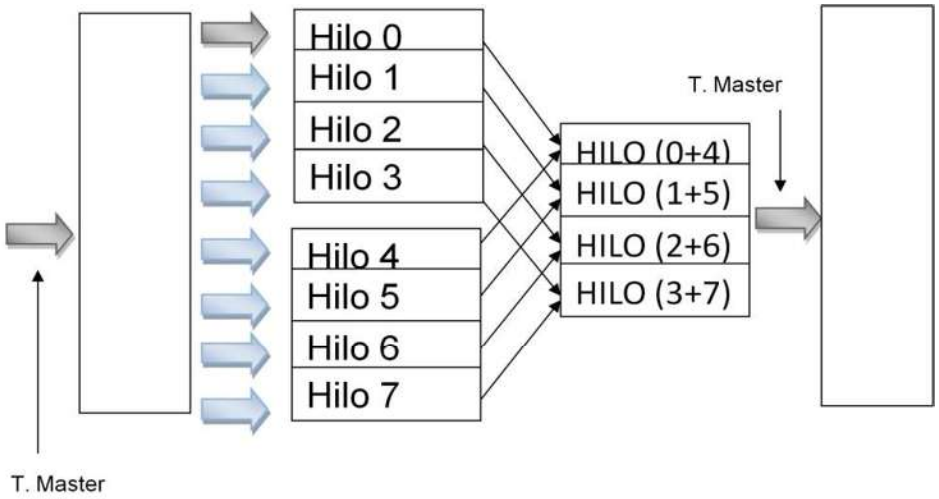


Figura 4.10 Repartición de los datos entre los procesadores para la suma de matrices.

### 4.3.2 Algoritmo paralelo para la resta de matrices

La propuesta que se tiene para la resta es similar que la de la suma. En seguida se muestra el código paralelizado.

```
#pragma omp parallel shared(A,B,C) private(i,j)
{
        #pragma omp for
        for (i=0;i<N;i++)
{
        for (j=0;j<N;j++)
            C[i*N+j]=A[i*N+j]-B[i*N+j];

    }/*fin de la región paralela*/
}
```

Figura 4.11. Propuesta de paralelización para la resta de matrices.

El procedimiento para la ejecución de la resta de matrices de manera paralela es similar a la de la suma de matrices. Se declaran las matrices *A, B,* y *C* como variables compartidas para todos los hilos y las variables *i* y *j* como variables privadas.

El procedimiento de la resta de matrices paralelizado aparece en la Figura 4.12. Es parecido al de la suma, a diferencia de que en este último caso se realiza la resta en lugar de sumar los datos.

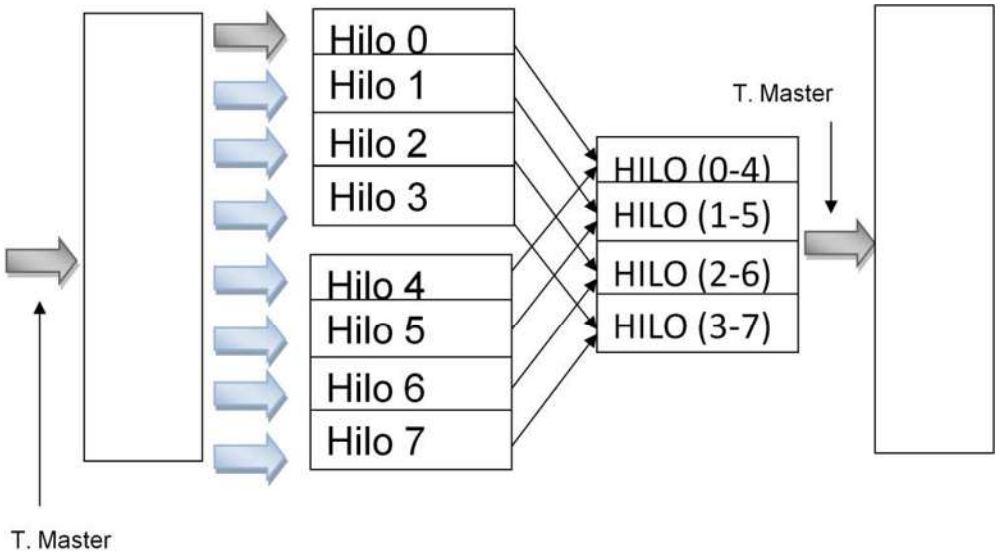


Figura 4.12 Repartición de los datos entre los procesadores para la resta.

### 4.3.3 Algoritmo paralelo para la multiplicación de una matriz por un escalar

El código paralelo de este algoritmo es el que aparece en la Figura 4.13. En este caso se están haciendo públicos los vectores la matriz *A, B*, el escalar *B* y la condición de paro *N* para que todos los hilos que los necesiten tengan acceso a ellos. Además se hizo una asignación dinámica de dos, a los hilos del equipo para que se repartan el trabajo conforme vayan terminando.

```
#pragma omp parallel shared(A,B,C,N) private (i,j)
#pragma omp for schedule(dynamic,2)
        for (i=0;i<N;i++)
                for(j=0;j<N;j++)
                C[i*N+j] = B*A[i*N+j];
```

Figura 4.13. Algoritmo paralelo para el producto matriz-escalar.

### 4.3.4 Algoritmo paralelo para el Producto Punto

Para el programa paralelizado del producto punto se propone el código que se encuentra en la Figura 4.14. Donde se tienen declarados a los vectores *a, b, c, nthreads, chunk,* y *N* como públicos a todos los hilos. La variable *i* se hace privada igual que *tid.* La variable *chunk*, es igual al número de procesadores que se tienen disponibles. En este caso entonces se divide el número de columnas entre ocho, y se realiza una asignación dinámica al equipo de hilos participantes en esta tarea

```
#pragma omp parallel shared(a,b,c,nthreads,chunk,N) private(i,tid)
        {
        #pragma omp for schedule(dynamic,chunk)
                for (i=0; i<N; i++)
                    c[i] = a[i]*b[i];
        }  /* end of parallel section */
```

Figura 4.14. Algoritmo paralelo para el producto punto.

### 4.3.5 Algoritmo paralelo para el producto matriz-vector

En la Figura 4.15 aparece el código paralelizado que se propone para realizar esta operación del producto de una matriz por un vector. Como se puede observar, vuelve a incluirse la cláusula *schedule* con la cual se está haciendo una repartición de la mitad del número de filas de la matriz *B* y entonces se realizará el producto correspondiente para encontrar el resultado de la operación.

```
#pragma omp parallel shared(N,A,B,C) private (i,j)
{
      #pragma omp for schedule(static,2)
           for(i=0;i<N;i++)
           {
             A[i]= 0;
                 for (j=0;j<N;j++)
                    A[i]+=B[i*N+j]*C[j];
           }
}
```

Figura 4.15. Algoritmo paralelo del producto matriz-vector.

### 4.3.6 Algoritmo paralelo para el producto de matrices

El caso de estudio del producto de una matriz por otra matriz es muy apropiado parser implementado en un ambiente paralelo dado que cada uno de los elementos que se calculan de la matriz resultado *C*, $c_{ij}$ es independiente de todos los demás elementos. Esta independencia es sumamente útil puesto que permite trabajar a todos los núcleos al mismo tiempo en diferentes bloques de filas.

En seguida se muestra la propuesta para la paralelización del algoritmo de la multiplicación de matrices en la Figura 4.16.

```
#pragma omp shared (aux,A,B,N) private(m,i,j)
        {
                #pragma omp parallel for
                for (m=0;m<N;m++)
                {
                        for (i=0;i<N;i++)
                        {
                                aux=0;
                        #pragma omp parallel for
                                for (j=0;j<N;j++)
                                        aux+=(A[m*N+j]*B[j*N+i]);
                        }
                }
        }
```

Figura 4.16. Propuesta de paralelización para la multiplicación de matrices.

Comenzando a contar los ciclos en orden descendente, se propone paralelizar el primer y el tercer ciclo puesto que no existe independencia en los cálculos realizados, además en el tercer ciclo es donde existe un mayor número de operaciones, lo cual es una razón más para paralelizarlo. La administración de los hilos quedará a cargo de OpenMP.

## 4.4 Conclusiones

En este capítulo se han descrito las operaciones matriciales básicas así como la programación secuencial y paralela de los algoritmos para su realización.

# Capítulo 5

# Casos de estudio

## 5.1 Introducción

Una vez que han sido propuestos y codificados algoritmos para la realización de diversas operaciones matriciales, a continuación se presentan algunos casos de estudio en donde se muestra el uso del procesamiento en paralelo basado en OpenMp.

Para la realización de los casos de estudio que se muestran a continuación se utilizó una computadora modelo *Intel(R) Xeon(R) CPU E5405 @ 2.00GHz.* La arquitectura de esta computadora, está conformada por ocho procesadores, con una velocidad de 2000.194 MHz cada uno. La Figura 5.1 muestra una vista frontal del equipo de cómputo utilizado.

Figura 5.1. Equipo de trabajo *Intel(R) Xeon(R) CPU E5405 @ 2.00GHz*

La Figura 5.2 muestra el interior de la computadora en donde se pueden observar los dos procesadores que contiene, cada uno de los cuales está formado por cuatro núcleos.

Figura 5.2 vista de los procesadores.

Figura 5.3. Vista de las memorias.

La Figura 5.3 muestra los módulos de memoria de la computadora utilizada en el desarrollo de esta tesis. Cada uno de estos módulos tiene una capacidad de 512Mbytes.

Durante las pruebas se hizo la variación del número de hilos en cada una de las operaciones y se registró la medición del *speed-up* para cada una de ellas.

El *speed-up* es una medida de la ejecución de un mismo programa en un número variado de procesadores. También se puede decir entonces, que es el tiempo necesario transcurrido en un procesador, dividido entre el tiempo necesario en *p* procesadores:

$$s = \frac{T(1)}{T(p)} \tag{5.1}$$

Donde:

S= speed-up

T(1) = tiempo de ejecución con un procesador.

T(p)= tiempo de ejecución con p elementos del proceso.

## 5.2 Caso de estudio 1. Suma de matrices

En este primer caso de estudio se realizó la suma de dos matrices de 10000 por 10000, es decir con 100,000,000 de elementos cada una. Esta operación se realizó utilizando desde un núcleo hasta 8 núcleos. La Tabla 5.1 resume los tiempos de ejecución obtenidos en 8 pruebas para cada número de núcleos utilizados.

Tabla 5.1. Resultados obtenidos de la suma de matrices con 100, 000,000 de elementos cada una. Tiempo medido en segundos.

| Número de hilos | prueba 1 | prueba 2 | prueba 3 | prueba 4 | prueba 5 | prueba 6 | prueba 7 | prueba 8 |
|---|---|---|---|---|---|---|---|---|
| 1 núcleo | 7.44 | 7.44 | 7.48 | 7.43 | 7.45 | 7.44 | 7.44 | 7.45 |
| 2 núcleos | 6.57 | 6.54 | 6.57 | 6.60 | 6.78 | 6.58 | 6.58 | 6.59 |
| 3 núcleos | 6.25 | 6.24 | 6.37 | 6.27 | 6.24 | 6.24 | 6.26 | 6.27 |
| 4 núcleos | 6.10 | 6.07 | 6.08 | 6.29 | 6.09 | 6.08 | 6.07 | 6.38 |
| 5 núcleos | 6.05 | 6.04 | 6.03 | 6.06 | 6.03 | 6.02 | 6.04 | 6.04 |
| 6 núcleos | 6.02 | 6.01 | 6.02 | 6.01 | 6.03 | 6.02 | 6.01 | 6.03 |
| 7 núcleos | 6.01 | 6.00 | 6.01 | 6.01 | 6.00 | 6.001 | 6.01 | 6.00 |
| 8 núcleos | 6.01 | 6.01 | 6.00 | 6.01 | 6.02 | 6.02 | 6.00 | 6.00 |

De los tiempos mostrados en la Tabla 5.1 se eliminó el tiempo más pequeño y el más grande y se promediaron los tiempos restantes para dar el tiempo promedio de ejecución para cada número de núcleos, los cuales se muestran en la Tabla 5.2.

Tabla 5.2 Tiempo de ejecución de la suma de matrices

| Número de núcleos | Promedio (segundos) |
|---|---|
| 1 | 7.44 |
| 2 | 6.6 |
| 3 | 6.27 |
| 4 | 6.15 |
| 5 | 6.04 |
| 6 | 6.02 |
| 7 | 6.01 |
| 8 | 6.01 |

Tabla 5.3. Speed-Up

| Número de núcleos | Speed-Up |
|---|---|
| 1 | 1 |
| 2 | 1.12727 |
| 3 | 1.1866 |
| 4 | 1.23794 |
| 5 | 1.20976 |
| 6 | 1.23588 |
| 7 | 1.23794 |
| 8 | 1.23794 |

La Tabla 5.3 muestra el *speed-up* obtenido con el uso desde uno, hasta 8 núcleos. De este registro puede apreciarse que utilizando 8 núcleos la suma matricial se realiza 1.23 más rápido que su contraparte serie.

En la gráfica mostrada en la Figura 5.1 se puede notar cómo es que al aumentar el número de núcleos en la tarea que se está realizando, el tiempo de ejecución del programa disminuye. Al inicio cuando solo se está utilizando un núcleo el tiempo es de 7.44 segundos, y al ir incrementando el número de núcleos disminuye hasta un tiempo de 6.01 segundos con el uso de 8 núcleos.

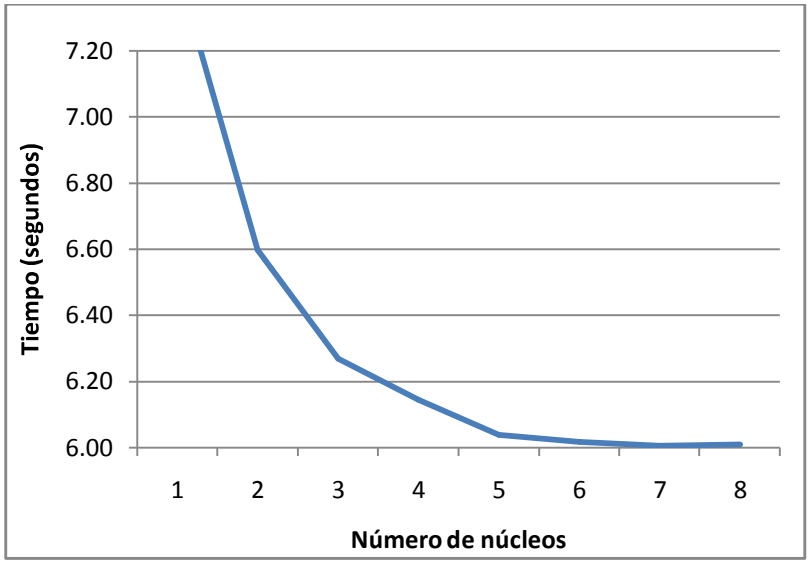


Figura 5.4 Tiempo de ejecución de la suma de matrices.

La Figura 5.2 muestra la gráfica del *speed-up* obtenida con el uso de uno hasta 8 núcleos correspondiente a la suma matricial. Se puede apreciar de esta Figura que el *speed-up* se mantiene prácticamente constante para 5,6,7 y 8 núcleos, además de que tiene un alto crecimiento con el uso de 2, 3 y 4 núcleos.

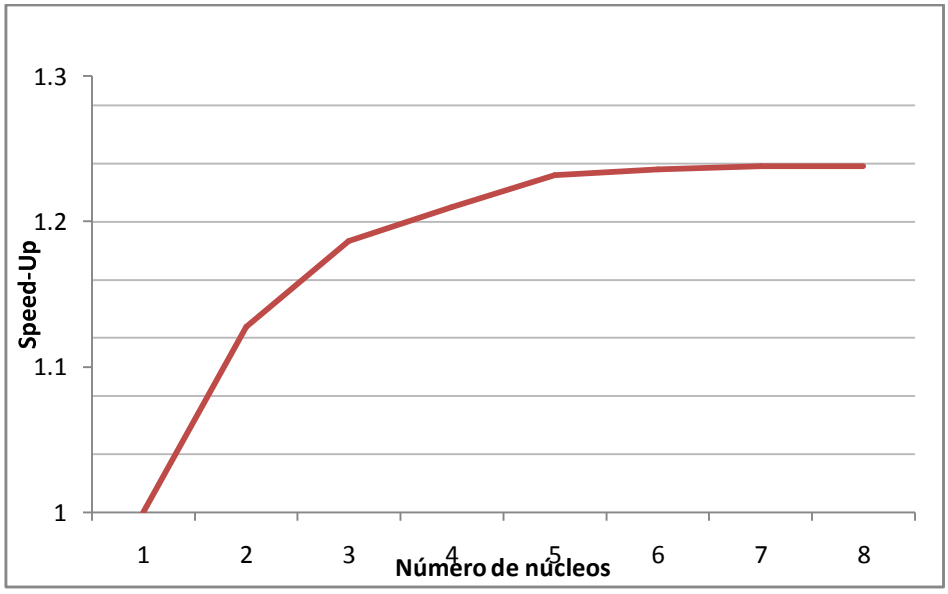


Figura 5.5. Speed Up para la suma de matrices.

## 5.3 Caso de estudio 2. Resta de matrices

En la tabla 5.4 se muestran las pruebas asociadas con la resta de matrices. Nuevamente se realizó una variación del número de núcleos desde uno hasta 8 obteniéndose resultados significativos.

Tabla 5.4. Tiempo de ejecución de la resta de matrices con 100, 000,000 de elementos cada una. Tiempo medido en segundos.

| Número de hilos | prueba 1 | prueba 2 | prueba 3 | prueba 4 | prueba 5 | prueba 6 | prueba 7 | prueba 8 |
|---|---|---|---|---|---|---|---|---|
| 1 núcleo | 7.11 | 7.09 | 7.08 | 7.10 | 7.11 | 7.09 | 7.10 | 7.08 |
| 2 núcleos | 6.38 | 6.43 | 6.40 | 6.41 | 6.41 | 6.40 | 6.39 | 6.39 |
| 3 núcleos | 6.15 | 6.15 | 6.15 | 6.16 | 6.15 | 6.16 | 6.16 | 6.73 |
| 4 núcleos | 6.06 | 6.04 | 6.04 | 6.07 | 6.08 | 6.05 | 6.62 | 6.04 |
| 5 núcleos | 6.05 | 6.03 | 6.03 | 6.05 | 6.32 | 6.05 | 6.06 | 6.05 |
| 6 núcleos | 6.03 | 6.02 | 6.06 | 6.31 | 6.02 | 6.02 | 6.02 | 6.02 |
| 7 núcleos | 6.01 | 6.01 | 6.01 | 6.01 | 6.62 | 6.01 | 6.01 | 6.04 |
| 8 núcleos | 5.99 | 6.00 | 6.03 | 6.02 | 6.02 | 6.02 | 6.01 | 6.01 |

En la Tabla 5.5 y 5.6 se presentan el tiempo de ejecución promedio y el *speed-up* respectivamente. Puesto que la resta de matrices es muy parecida a la suma, el resultado esperado en los siguientes datos debe también tener una similitud con los obtenidos en la suma de matrices.

Tabla 5.5. Tiempo de ejecución promedio de la suma de matrices

| Número de núcleos | Tiempo promedio |
|---|---|
| 1 | 7.09 |
| 2 | 6.4 |
| 3 | 6.23 |
| 4 | 6.12 |
| 5 | 6.08 |
| 6 | 6.06 |
| 7 | 6.02 |
| 8 | 6.01 |

Tabla 5.6. Speed-up para la suma de matrices

| número de núcleos | Speed-up |
|---|---|
| 1 | 1 |
| 2 | 1.10781 |
| 3 | 1.13804 |
| 4 | 1.1585 |
| 5 | 1.16612 |
| 6 | 1.16997 |
| 7 | 1.1642 |
| 8 | 1.1797 |

En la gráfica de la Figura 5.3 se muestra el tiempo de ejecución contra el número de núcleos, donde se onbserva un mayor decrecimiento en tiempos cuando se utilizan 2 y 3 núcleos. Para 4,5,6,7 y 8 es menor.

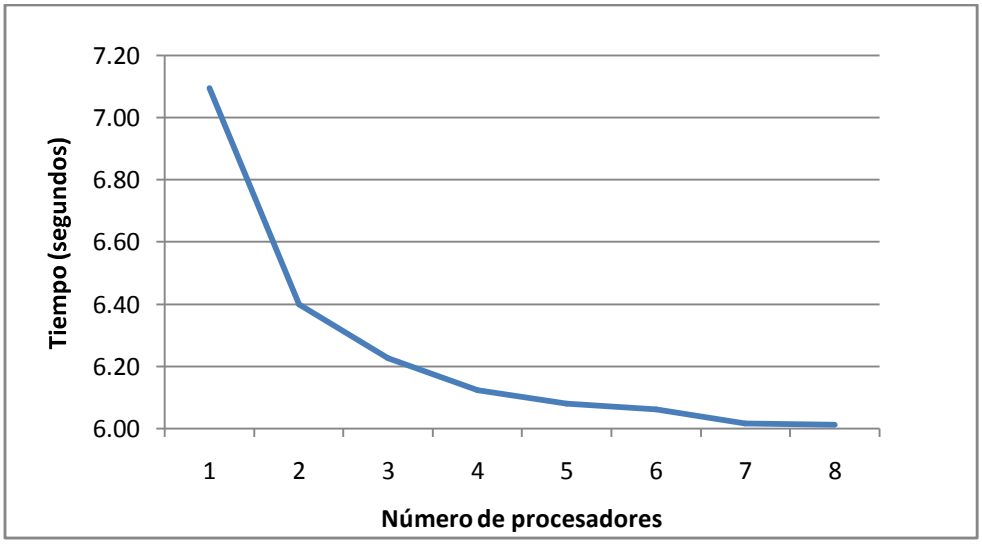


Figura 5.6. Tiempo de ejecución para la resta de matrices.

En la Figura 5.6 se muestra el *speed-up* para el caso de la resta de matrices. El resultado demuestra un mayor crecimiento al incrementar de 1 a 2 núcleos, mientras que para 7 y 8 el *speed-up* es constante, lo que indica que el código ha llegado a su máximo grado de paralelismo.

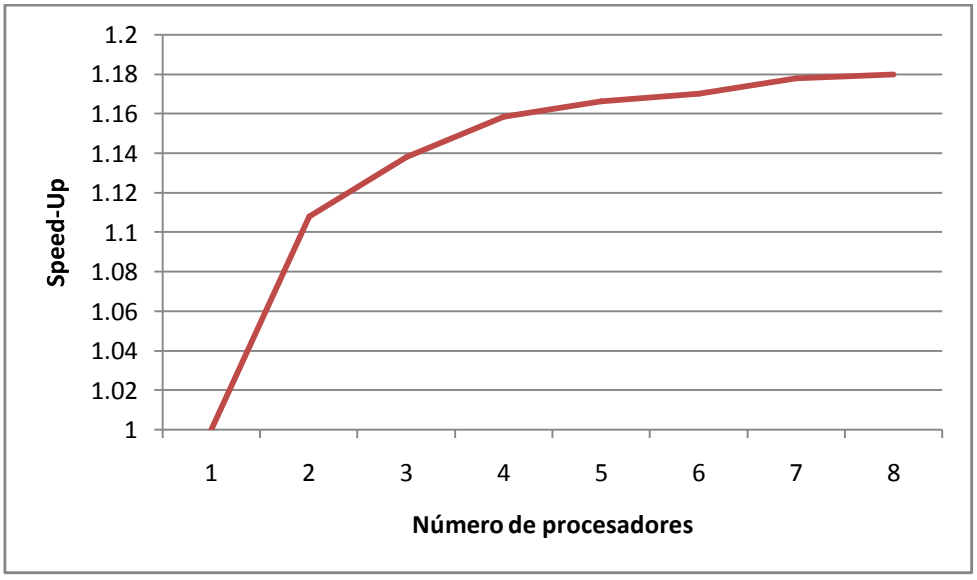


Figura 5.7. Speed-Up para la resta de matrices.

## 5.4 Caso de estudio 3. Multiplicación de matriz-escalar

La Tabla 5.7 contiene los datos registrados para el producto de una matriz por un escalar.

Tabla 5.7. Datos para la operación matriz-escalar con 10000 elementos. Tiempo medido en segundos.

| Número de hilos | prueba 1 | Prueba 2 | prueba 3 | prueba 4 | prueba 5 | Prueba 6 | prueba 7 | prueba 8 |
|---|---|---|---|---|---|---|---|---|
| 1 núcleo | 4.163 | 4.191 | 4.156 | 4.163 | 4.171 | 4.16 | 4.162 | 4.178 |
| 2 núcleos | 3.605 | 3.618 | 3.614 | 3.622 | 3.613 | 3.624 | 3.617 | 3.75 |
| 3 núcleos | 3.374 | 3.383 | 3.376 | 3.644 | 3.377 | 3.382 | 3.381 | 3.375 |
| 4 núcleos | 3.271 | 3.282 | 3.276 | 3.285 | 3.272 | 3.278 | 3.303 | 3.287 |
| 5 núcleos | 3.246 | 3.248 | 3.236 | 3.249 | 3.233 | 3.245 | 3.245 | 3.241 |
| 6 núcleos | 3.236 | 3.236 | 3.234 | 3.237 | 3.235 | 3.236 | 3.237 | 3.237 |
| 7 núcleos | 3.226 | 3.239 | 3.229 | 3.23 | 3.238 | 3.27 | 3.24 | 3.227 |
| 8 núcleos | 3.240 | 3.234 | 3.241 | 3.233 | 3.229 | 3.238 | 3.243 | 3.236 |

A partir de esta tabla se obtienen los resultados promedios para proceder a graficarlos.

Tabla 5.8. Tiempo de ejecución promedio del producto: matriz escalar

| número de núcleos | Promedio (segundos) |
|---|---|
| 1 | 4.1689 |
| 2 | 3.6328 |
| 3 | 3.4115 |
| 4 | 3.2817 |
| 5 | 3.2428 |
| 6 | 3.2678 |
| 7 | 3.2456 |
| 8 | 3.1168 |

Tabla 5.9. Speed-up para el producto: matriz escalar

| número de núcleos | speed-up |
|---|---|
| 1 | 1.000 |
| 2 | 1.147 |
| 3 | 1.222 |
| 4 | 1.272 |
| 5 | 1.285 |
| 6 | 1.275 |
| 7 | 1.284 |
| 8 | 1.337 |

En la Figura 5.5 se presentan gráficamente los resultados para el tiempo de ejecución, Visualizando los datos de la Tabla 5.8.

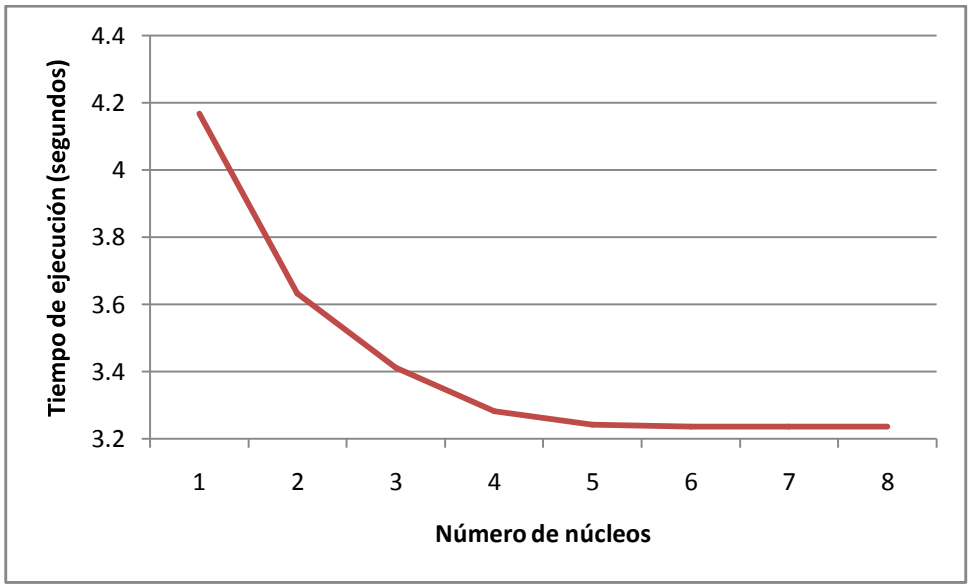


Figura 5.8 Tiempo de ejecución para la operación matriz-escalar

Se puede observar notoriamente en qué proporciones disminuye el tiempo de ejecución. De la información mostrada en la gráfica se deduce que el cambio negativo en el tiempo de ejecución se da para 2,3 y 4 núcleos.

Por otra parte la gráfica del *speed-up* muestra una línea casi recta de 5 núcleos en adelante. Se tiene pues con 8 núcleos una velocidad 1.337 veces más rápida que con un solo núcleo.

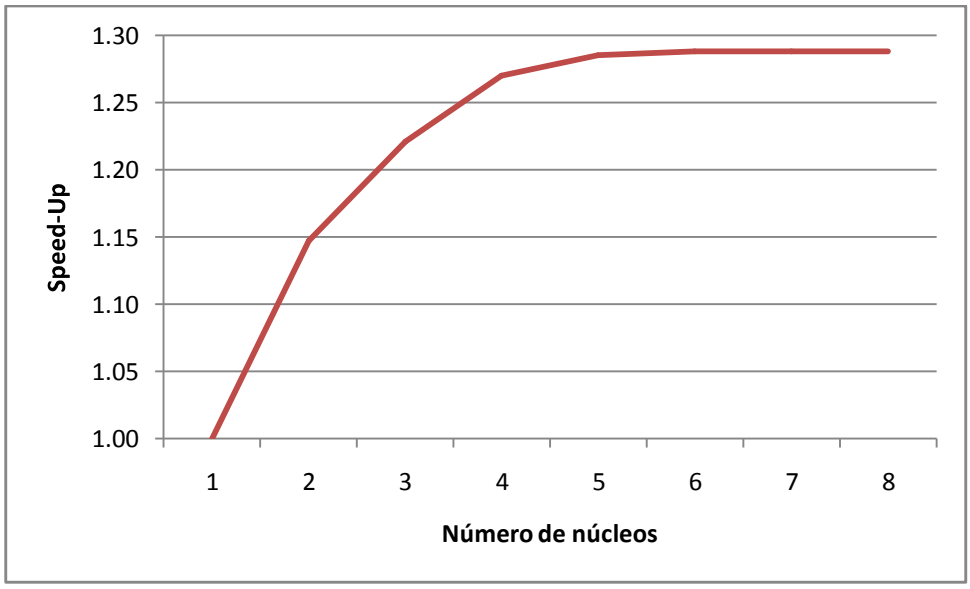


Figura 5.9 Speed-Up para la operación matriz-escalar

## 5.5 Caso de estudio 4. Producto punto

En las Tablas 5.10 y 5.11 se presentan los datos para el producto punto. En este caso se utilizaron 100, 000,000 elementos para el análisis.

Tabla 5.10. Tiempo de ejecución promedio del producto punto

| número de núcleos | Promedio (segundos) |
|---|---|
| 1 | 2.789667 |
| 2 | 2.342333 |
| 3 | 2.158333 |
| 4 | 2.101667 |
| 5 | 2.085333 |
| 6 | 2.073667 |
| 7 | 2.067333 |
| 8 | 2.068976 |

Tabla 5.11. Speed-up para el producto punto.

| número de núcleos | speed-up |
|---|---|
| 1 | 1 |
| 2 | 1.19097 |
| 3 | 1.29251 |
| 4 | 1.32735 |
| 5 | 1.33775 |
| 6 | 1.34528 |
| 7 | 1.34940 |
| 8 | 1.34896 |

Al revisar las Tablas se puede ver que la reducción en tiempo de ejecución es de más de 7 décimas de segundo al utilizar el número máximo de procesadores que corresponde a un *Speed-Up* de 1.34. Entonces se puede concluir que se está haciendo el trabajo un 34 % más rápido con esta propuesta de paralelización.

En la Figura 5.9 se presenta el tiempo de ejecución para el producto punto.

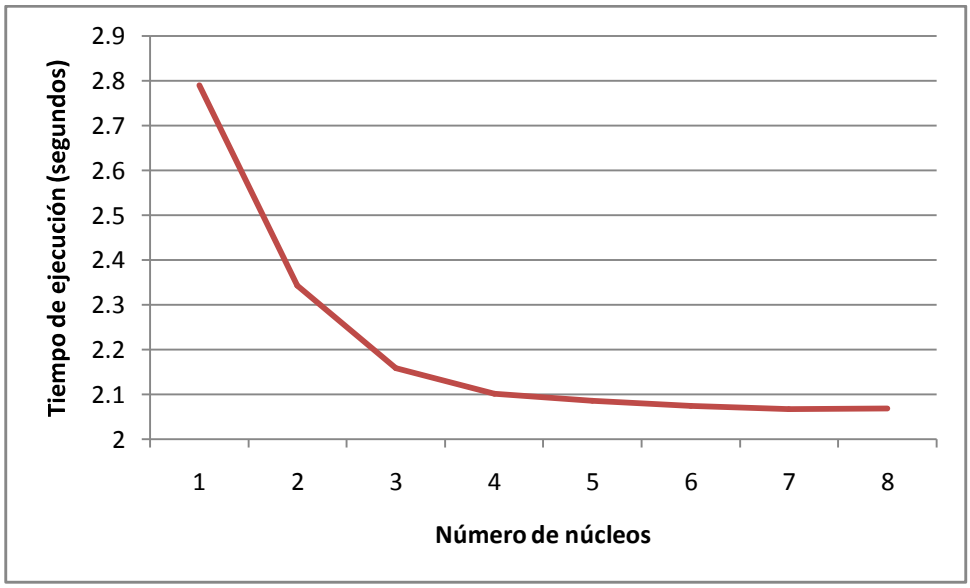


Figura 5.10 tiempo de ejecución para el producto punto.

En la Figura 5.10 el *Speed-Up* del producto punto.

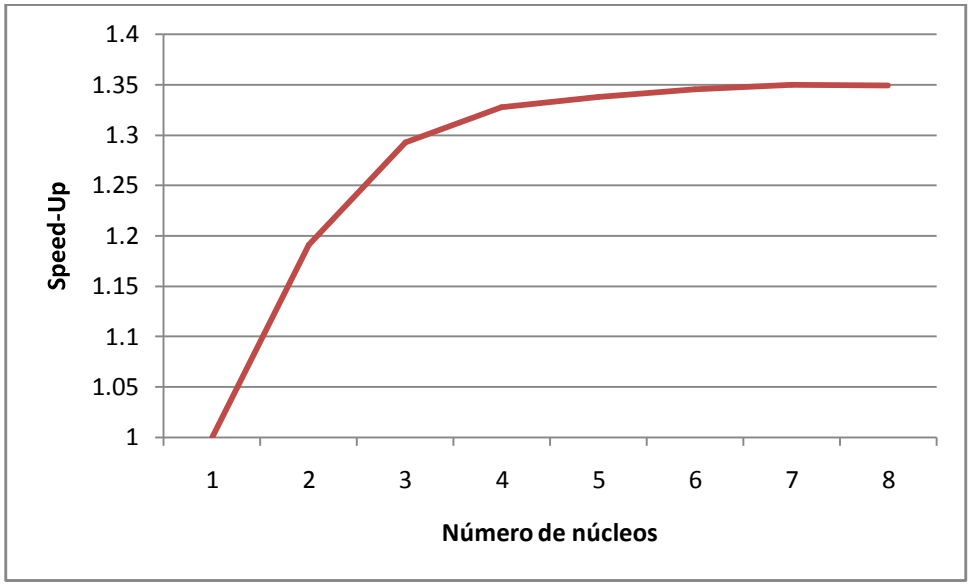


Figura 5.11 Speed-Up para el producto punto.

De esta representación gráfica se observa que los mayores incrementos del *speed-up* tienen lugar hasta llegar a 3 núcleos. Después de ahí comienza a disminuir la ganancia en velocidad hasta llegar a ser prácticamente constante para 7 y 8 núcleos.

## 5.6 Caso de estudio 5. Producto matriz-vector

En seguida se muestra la Tabla 5.12 donde se registraron todos los datos relativos a la operación de la multiplicación matriz por vector.

Tabla 5.12 tiempo de ejecución del producto matriz por vector con 100, 000,000 elementos. Tiempo medido en segundos.

| Número de hilos | prueba 1 | prueba 2 | prueba 3 | prueba 4 | prueba 5 | prueba 6 | prueba 7 | prueba 8 |
|---|---|---|---|---|---|---|---|---|
| 1 núcleo | 3.76 | 3.76 | 3.76 | 3.76 | 3.76 | 3.76 | 3.76 | 3.76 |
| 2 núcleos | 3.22 | 3.52 | 3.22 | 3.22 | 3.22 | 3.22 | 3.21 | 3.22 |
| 3 núcleos | 3.06 | 3.46 | 3.04 | 3.06 | 3.06 | 3.05 | 3.04 | 3.04 |
| 4 núcleos | 2.97 | 2.96 | 2.97 | 2.96 | 2.96 | 2.96 | 2.96 | 2.98 |
| 5 núcleos | 2.94 | 2.92 | 2.93 | 2.91 | 2.93 | 2.94 | 2.93 | 2.93 |
| 6 núcleos | 2.92 | 2.93 | 2.94 | 2.92 | 2.92 | 2.92 | 2.93 | 2.92 |
| 7 núcleos | 2.91 | 2.93 | 2.92 | 2.91 | 2.91 | 2.92 | 2.92 | 2.91 |
| 8 núcleos | 2.91 | 2.91 | 2.93 | 2.91 | 2.91 | 2.92 | 2.91 | 2.91 |

Una vez que se realizaron todas las pruebas, se procede nuevamente a encontrar los parámetros promedio para realizar la graficación de los mismos y demostrar la eficiencia que se ha tenido al paralelizar el código en las Tablas 5.13 y 5.14.

Tabla 5.13. Tiempo de ejecución promedio del producto: matriz vector

| número de núcleos | Promedio (segundos) |
|---|---|
| 1 | 3.76 |
| 2 | 3.26 |
| 3 | 3.10 |
| 4 | 2.96 |
| 5 | 2.93 |
| 6 | 2.92 |
| 7 | 2.92 |
| 8 | 2.91 |

Tabla 5.14. Speed-up para el producto: matriz vector

| número de núcleos | speed-up |
|---|---|
| 1 | 1 |
| 2 | 1.533 |
| 3 | 1.212 |
| 4 | 1.270 |
| 5 | 1.283 |
| 6 | 1.287 |
| 7 | 1.287 |
| 8 | 1.292 |

Haciendo una comparación del tiempo promedio contra la ganancia en velocidad, lo que indican las Tablas es que para dos núcleos, el tiempo que tardó la aplicación fue de 3.26 segundos, con un *speed-up* de 1.1533, haciendo una disminución de 1.5 segundos comparando los tiempos de un núcleo contra 2 núcleos. Esto se ve claramente en la gráfica de la Figura 5.11.

Figura 5.12. Tiempo de ejecución del producto: matriz vector.

La gráfica del *speed-up* está representada por la Figura 5.12.

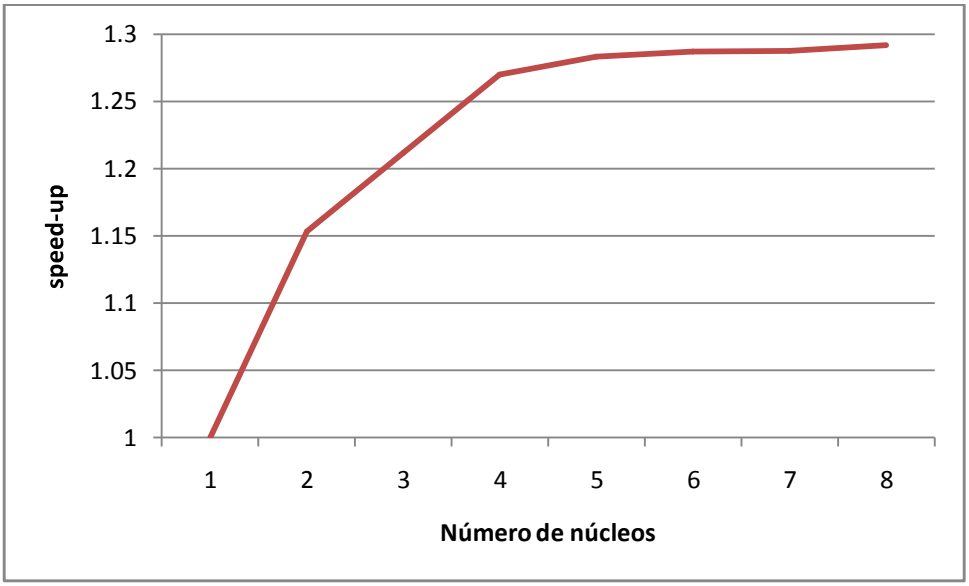


Figura 5.13. Speed-up para el producto: matriz vector.

Finalmente, al utilizar el máximo de hilos disponibles, es decir 8 núcleos se obtiene un *speed-up* de 1.29096, que corresponde a un tiempo de 2.91 segundos. Si se realiza una comparación nuevamente, pero esta vez contra el tiempo de 2 núcleos se obtiene una diferencia de .35 segundos. Esto indica que la operación del producto de una matriz por un vector, no tiene un grado muy elevado de paralelismo como lo tiene la multiplicación de dos matrices como se verá a continuación.

## 5.7 Caso de estudio 6. Producto de matrices

Por último, se ejecutó la aplicación paralelizada del producto de matrices registrándose los tiempos que aparecen en la Tabla 5.15. Antes de continuar con la descripción de este caso de estudio, es importante señalar que durante la ejecución de los casos de estudio anteriores, no se registró un *speed-up* mayor a 1.35 que representaría una velocidad 35% más rápida si se comparara contra la velocidad alcanzada por un solo núcleo.

Con este último caso se concluye qué operación tiene el mayor grado de paralelismo dentro de las propuestas establecidas en esta tesis para la arquitectura de memoria compartida.

En la Tabla 5.15 se registraron los tiempos correspondientes a la multiplicación de matrices.

Tabla 5.15 Tiempo de ejecución de la multiplicación de matrices con 1000000 elementos. Tiempo medido en segundos.

| Número de hilos | Prueba 1 | prueba 2 | prueba 3 | prueba 4 | prueba 5 | prueba 6 | prueba 7 | prueba 8 |
|---|---|---|---|---|---|---|---|---|
| 1 núcleo | 29.99 | 30.07 | 30.22 | 30.20 | 30.20 | 32.19 | 30.22 | 32.87 |
| 2 núcleos | 14.96 | 14.59 | 14.96 | 14.89 | 15.33 | 15.31 | 15.30 | 15.26 |
| 3 núcleos | 10.20 | 10.21 | 10.38 | 10.61 | 10.58 | 10.60 | 10.62 | 10.61 |
| 4 núcleos | 8.22 | 8.07 | 8.06 | 8.09 | 8.12 | 8.07 | 7.97 | 8.00 |
| 5 núcleos | 6.10 | 6.41 | 6.39 | 6.33 | 6.27 | 6.40 | 6.40 | 6.47 |
| 6 núcleos | 5.55 | 5.55 | 5.59 | 5.43 | 5.56 | 5.51 | 5.63 | 5.59 |
| 7 núcleos | 4.92 | 4.86 | 4.90 | 4.93 | 4.86 | 4.82 | 4.83 | 4.87 |
| 8 núcleos | 4.62 | 4.56 | 4.56 | 4.52 | 4.70 | 4.71 | 4.55 | 4.70 |

Ahora se presentan las Tablas 5.16 y 5.17 con los resultados promedio de la Tabla 5.15.

Tabla 5.13. Tiempo de ejecución promedio del productode matrices

| número de núcleos | Promedio (segundos) |
|---|---|
| 1 | 30.75 |
| 2 | 15.07 |
| 3 | 10.47 |
| 4 | 8.08 |
| 5 | 6.35 |
| 6 | 5.55 |
| 7 | 4.87 |
| 8 | 4.61 |

Tabla 5.14. Speed-up para el producto de matrices

| número de núcleos | speed-up |
|---|---|
| 1 | 1 |
| 2 | 2.04048 |
| 3 | 2.93696 |
| 4 | 3.80569 |
| 5 | 4.84252 |
| 6 | 5.54054 |
| 7 | 6.31417 |
| 8 | 6.67028 |

Los datos registrados en las Tablas 5.13 y 5.14 permiten conocer el nivel de paralelización de la multiplicación de matrices en ambientes multiprocesador, específicamente utilizando OpenMp. Si se realiza una comparativa con las operaciones anteriores se puede notar de una manera categórica que efectivamente, la multiplicación de matrices tiene el mayor grado de paralelización al obtener un tiempo de 30.75 segundos utilizando un núcleo y uno de 4.61 segundos al utilizar el máximo de 8 núcleos. El tiempo

mínimo de 4.61 segundos equivale a un *speed-up* de 6.670 que confirma lo dicho anteriormente.

En la Figura 5.13 se grafica el tiempo de ejecución contra el número de núcleos obteniéndose una curva, que indica una disminución constante en el tiempo utilizado por la aplicación.

Figura 5.14. Gráfica del tiempo de ejecución del producto de Matrices.

Ahora se muestra la gráfica del *speed-up* casi como una recta, que respalda un crecimiento constante en velocidad y que es proporcional al número de núcleos utilizados.

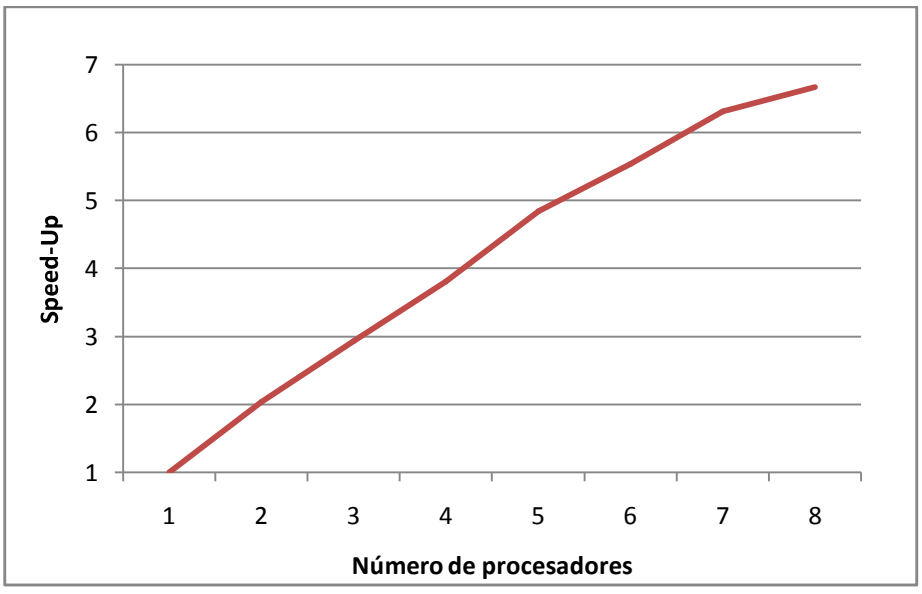


Figura 5.15. Speed-UP para le multiplicación de matrices.

# Capítulo 6

# Conclusiones y recomendaciones para trabajos futuros

## 6.1 Conclusiones

- Se han implementado las operaciones matriciales de manera secuencial usando el lenguaje de programación c++ y se realizaron pruebas con diferente número de elementos en cada uno de los casos de estudio.
- Se ha logrado paralelizar las principales operaciones matriciales utilizando OpenMP de una manera eficiente y al mismo tiempo satisfactoria, al hacer una comparación de tiempos en la programación secuencial, es decir la implementación con un procesador contra la programación en paralelo con dos o más hilos como se mostró en las tablas de los casos de estudio.
- Se ha mostrado el poderío de OpenMP y la forma en que proporciona facilidad en la implementación de programas paralelos ejecutados sobre arquitecturas multiprocesador, más concretamente en este proyecto, sobre operaciones matriciales.
- Se ha evidenciado la gran capacidad del procesamiento en paralelo para resolver problemas de ámbito numérico con velocidades de respuesta considerables.
- En cuanto a las operaciones matriciales se puede decir que la operación con más grado de paralelismo es la de la multiplicación de matrices. Esta conclusión vierte del análisis del *Speed-Up*, y de la gráfica del tiempo de ejecución. Dicho de otro modo, la eficiencia del uso de los núcleos es mayor en el caso de la multiplicación de matrices que en el de la suma y resta, incluso mayor que en el caso de la multiplicación de la matriz por el vector, por lo que se comprueba que el producto de matrices es ideal para arquitecturas de memoria compartida.

## 6.2 Trabajos futuros

Queda abierta la propuesta para una investigación futura acerca de los siguientes temas:

- Aplicación de técnicas de procesamiento en paralelo para la solución de sistemas de ecuaciones lineales de grandes dimensiones y del problema característico $Ax = B$.
- Aplicación de técnicas de procesamiento en paralelo para el análisis de sistemas eléctricos.
- Aplicación de técnicas de procesamiento en paralelo en el análisis de flujos de potencia.

# Referencias

**Libro:**

[Chapman 2008]

Barbara Chapman. Using OpenMP: Portable Shared Memory Parallel Programming. Massachusetts Institute of Technology: Edited by Janusz Kowalik. 2008.

[Shonkwiler 2006]

Ronald Shonkwiler and Lew Lefton. An Introduction to Parallel and Vector Scientific Computing. Cambridge: Editorial Board. 2006.

[Geist 1994]

Al Geist. PVM: Parallel Virtual Machine. Cambridge, Massachusetts, London, England. 1994.

**Internet:**

[Pthreads 2010]

Pthreads. Tutoriales. Estados Unidos. 13 de Enero de 2010. https://computing.llnl.gov/tutorials/pthreads/

[MPI Forum 2009]

MPI. Documentos de MPI. Estados Unidos. 29 de Julio de 2009. http://www.mpi-forum.org/docs/docs.html

[OpenMP 2008]

OpenMP. Página principal. Estados Unidos 24 de Noviembre de 2009. http://openmp.org/wp/

[OpenMP V3.0 2008]

OpenMP. Documentos. Estados Unidos 25 de Noviembre de 2009. http://www.openmp.org/mp-documents/spec30.pdf

[Universidad de Valladolid 2008]

Universidad de Valladolid. Iniciación al PVM. 30 de Noviembre de 2009. http://www.infor.uva.es/~bastida/Arquitecturas%20Avanzadas/pvm.pdf

[Cronología del procesamiento en paralelo]
http://homepage.mac.com/eravila/histpara.html

[Universidad Metropolitana, 2007]

Departamento de Ing. Eléctrica. Área de Computación y Sistemas. 12 de Enero de 2010.
http://www.izt.uam.mx/contactos/n64ne/imag-med.pdf

[Facultad de Ingeniería. Universidad de la República de Urugay, 2009]

Área de trabajo e Investigación en Procesamiento en Paralelo y sus Aplicaciones
http://www.fing.edu.uy/inco/grupos/cecal/hpc/proyectos/

[Universidad de Murcia, 2009]

Programación Paralela y Distribuída, Departamento de Informática y Sistemas, Universidad de Murcia, España.
http://servinf.dif.um.es/~domingo/09/ERBASE/minicursoespanol.pdf

[Fernando G. Tinetti, 2003]

Universidad Autónoma de Barcelona. Barcelona 2003.

http://www.tdr.cesca.es

[Instituto Nacional de Astrofísica, Óptica y Electrónica, 2009]

Instituto Nacional de Astrofísica, Óptica y Electrónica. Sistemas Distribuidos. México. 5 de Febrero de 2010.

http://ccc.inaoep.mx/~lamorales/distribuidos/Sistemas_Distribuidos.pdf