Vniversidad Nacional Avtonoma de Mexico

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"Análisis comparativo del desempeño de patrones arquitectónicos para programación paralela"

T E S I S

Que para obtener el grado de
Maestro en Ciencias de la Computación

PRESENTA:

RODRIGO ISRAEL NOVELO CERVERA

Director de Tesis: Dr. Jorge Luis Ortega Arjona

MÉXICO, DF, 2011

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Índice general

Resumen **V**

1. Introducción **1**
1.1. Contexto de la tesis 1
1.2. Problemática 2
1.3. Hipótesis 3
1.4. Objetivo 4
1.5. Aproximación 4
1.6. Contribuciones 4
1.7. Metodología 5
1.8. Estructura de la tesis 6

2. Antecedentes **8**
2.1. Cómputo paralelo 8
2.1.1. La clasificación de Flynn 9
2.1.2. Arquitecturas Paralelas Reales 14
2.2. Programación Paralela 17
2.2.1. Modelos de algoritmos paralelos 18
2.3. Patrones Arquitectónicos 18
2.4. Patrones Arquitectónicos para Programación Paralela 18
2.4.1. Patrón *Manager-Workers* 19
2.4.2. Patrón *Parallel Pipes & Filters* 20
2.4.3. Patrón *Parallel Layers* 20
2.4.4. Patrón *Communicating Sequential Elements* 21
2.4.5. Patrón *Shared Resource* 22
2.5. Paso de Mensajes 23
2.5.1. Interface de Paso de Mensajes (MPI) 23
2.6. Métricas de desempeño 25
2.6.1. Tiempo de Ejecución 26

2.6.2. *Speedup* 26
2.6.3. Eficiencia 28
2.7. Resumen 28

3. Trabajo relacionados **29**
3.1. Arquitecturas paralelas para procesamiento de imágenes 29
3.1.1. Paralelismo Pipeline 29
3.1.2. Paralelismo de Datos 30
3.2. Resumen 32

4. Análisis comparativo **34**
4.1. Representación de una imagen digital 34
4.2. Imágenes a escala de grises 35
4.3. Imágenes a color en RGB 36
4.3.1. Detección de bordes en imágenes a color 36
4.4. Algoritmo de detección de bordes de Canny 38
4.5. Paralelización del algoritmo de Canny 41
4.5.1. Aproximación *Manager-Workers* 43
4.5.2. Aproximación *Communicating Sequential Elements* 44
4.6. Comparación de patrones arquitectónicos para programación paralela 46
4.7. Resumen 47

5. Evaluación experimental **48**
5.1. Supuestos 48
5.2. Variables de los Experimentos 49
5.2.1. Imagen de entrada 49
5.2.2. Número de Procesadores 49
5.2.3. Número de Procesos por procesador 49
5.3. Número de Experimentos 50
5.4. Resultados y Análisis 52
5.4.1. Tiempos de ejecución promedio 52
5.4.2. Análisis del *Speedup* y Eficiencia 59
5.5. Resumen 66

6. Conclusiones y Trabajo futuro **67**
6.1. Conclusiones generales 67
6.2. Trabajo futuro 69

Bibliografía **70**

Apéndices **73**

A. Intervalos de Confianza **73**
A.1. Construyendo un intervalo de confianza 73
A.2. Intervalo de confianza para la media de la población 74

B. Paso de mensajes con MPI **76**
B.1. Ejemplo: 76
B.2. Paso de mensajes bloqueante 77
B.2.1. Código fuente en C/MPI utilizando MPI_Send y MPI_Recv 77
B.3. Paso de mensajes no bloqueantes 79
B.3.1. Código fuente en C/MPI utilizando MPI_Isend y MPI_Irecv 80

Índice alfabético **82**

Agradecimientos

Esta tesis no hubiera sido posible sin la ayuda de muchas personas que se vieron involucradas directa o indirectamente, leyendo, opinando, teniéndome paciencia, dandome ánimo, acompañandome en los momentos de crisis y en los momentos de felicidad.

Agradezco el apoyo incondicional de mis padres, hermanos y sobrinos, porque a pesar de mi ausencia siempre estuvieron cerca de mi.

Gracias a mis grandes amigas Paty y Fátima por brindarme su amistad, cariño y mucha paciencia a lo largo de este tiempo y sobretodo por hacerme sentir como en casa.

Al Dr. Jorge Ortega y al Dr. Héctor Benitez por haber confiado en mi, por su apoyo y gran ayuda en este trabajo, y sobre todo por haberme brindado su amistad.

A todos mis amigos, por el gran apoyo que me han dado, particularmente a: Rocio, Miri, Magali, Martha, Ireri, Sergio, Ger, al Nax, al Padrino y al TJ.

Y sobre todo a ti Maylie, por haber creido en mi desde el principio, tu impulsaste este gran sueño que hoy se hace realidad, Gracias.

Gracias a todos.

Resumen

Hoy en día existen muchos problemas computacionales que requieren de gran poder de cómputo y/o manejo de grandes cantidades de información. Estos tipos de problemas pueden demorarse días, meses o hasta años para poder obtener resultados. El cómputo paralelo puede ser utilizado para minimizar el tiempo de procesamiento de meses a horas o incluso a minutos.

Existen diversos patrones arquitectónicos para programación paralela que se pueden implementar en un programa paralelo, entre los más importantes están: *Manager-Workers*, *Communicating Sequential Elements*, *Parallel Layers*, *Parallel Pipes and Filters* y *Shared Resource*.

Uno de los principales retos al diseñar un programa paralelo es poder determinar de antemano cuál patrón arquitectónico para programación paralela se debe de implementar.

Una de las formas para lograrlo es expresar el problema en términos de algoritmo y datos. Después se analizan ambas partes para poder determinar si son paralelizables. El resultado del análisis se pasa a través de un criterio que selecciona entre los posibles patrones arquitectónicos para programación paralela. Para algunos casos la decisión no es contudente, es decir, el análisis puede ofrecer al menos dos distintos patrones arquitectónicos.

Para estos casos no existe un criterio complementario para decidir cuál es el patrón arquitectónico que mejor conviene, sino que se basan en la experiencia para poder elegir uno del otro. Es por esta razón que se realiza un análisis comparativo del desempeño como criterio de desempate entre los patrones arquitectónicos resultantes.

El análisis comparativo ayuda a determinar cuál patrón arquitectónico para programación paralela ofrece mejor desempeño para un problema dado, y bajo qué circunstancias se cumple.

Capítulo 1
Introducción

1.1. Contexto de la tesis

En las últimas décadas, la utilización de la computación paralela se ha incrementado debido a la gran cantidad de información que se requiere procesar en el menor tiempo posible. La computación paralela consiste principalmente en la división de una carga de trabajo en tareas pequeñas, las cuales se distribuyen para su realización en diferentes procesadores. La implementación es un programa paralelo.

Las aplicaciones científicas siguen siendo una de las principales razones detrás del desarrollo del cómputo paralelo, que es necesaria para la manipulación de grandes bases de datos, aceleración en la ejecución del código, y la resolución de problemas que requieren una cantidad significante de tiempo de procesamiento [FLD00].

Esta tesis se encuentra situada en la intersección de dos disciplinas computacionales: Cómputo Paralelo e Ingeniería de Software. En esta intersección se localizan los patrones arquitectónicos para programación paralela, que son los elementos básicos iniciales de esta tesis, (Figura 1.1). Cada uno de estos patrones proveen diferentes maneras de coordinación entre los programas paralelos.

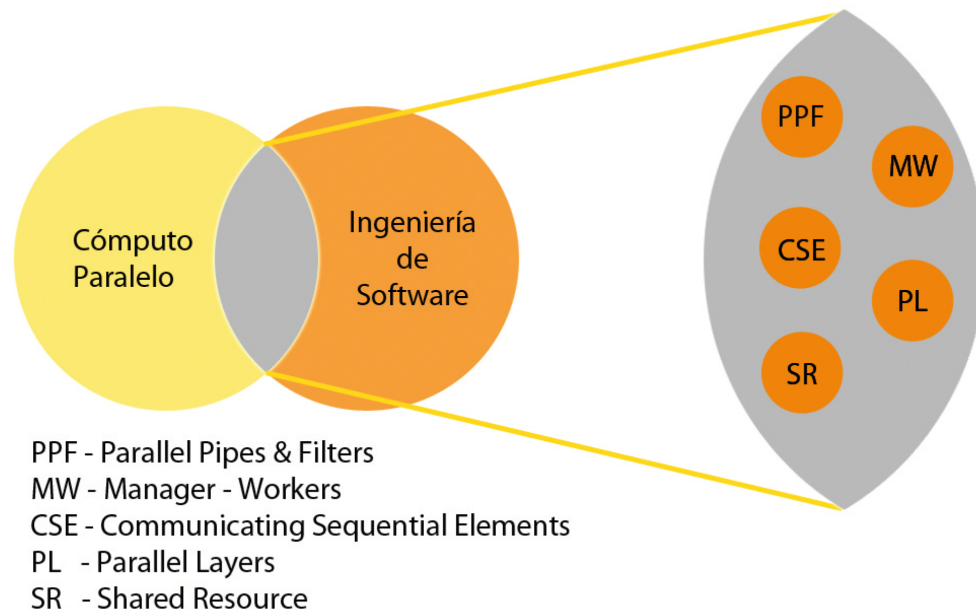

Figura 1.1: Contexto de la tesis.

1.2. Problemática

Hoy en día se pueden encontrar muchos problemas computacionales con grandes volúmenes de datos que se requieren procesar, una gran cantidad de iteraciones en el código o simplemente un gran número de operaciones computacionales que consumen mucho tiempo del procesador. Este tipo de problemas es prácticamente difícil de resolver utilizando una única computadora convencional con un programa secuencial. Por ejemplo, cuando dado un problema el tamaño de la memoria y/o la capacidad de procesamiento son limitadas.

Cuando un programa no puede ejecutarse debido a las limitantes anteriores se justifica el uso del cómputo paralelo. Este tipo de cómputo puede ser la solución. Sin embargo, existe la posibilidad de que un inadecuado análisis conlleve a una mala implementación del programa en paralelo.

Este análisis consiste en considerar principalmente al problema expresado en términos de algoritmo y datos. Al analizar el algoritmo se debe de observar si este es posible de dividir en varios bloques independientes, donde cada bloque pudiera ser procesado independientemente sin afectar la salida del algoritmo. Por otro lado, el análisis de los datos ayuda a determinar la forma en la que los datos pudieran ser distribuidos a lo largo de todos los procesadores del sistema, y así, lograr que se procesen independientemente.

El criterio anterior permite determinar qué tipo de organización ó patrón arquitectónico para programación paralela se debe utilizar para un determinado problema. Este criterio de selección se puede observar en la Tabla 1.1. Los posibles tipos de paralelismo en donde la mayoría de los programas paralelos caen son [OA10]: paralelismo funcional, paralelismo de dominio y paralelismo de actividad.

Tabla 1.1: Criterio para determinar el tipo de paralelismo.

Particionamiento		Tipo de Paralelismo
Algoritmo	Datos	
No	No	Secuencial
No	Si	Dominio
Si	No	Funcional
Si	Si	Actividad

Dependiendo del tipo de paralelismo obtenido por este criterio es el patrón arquitectónico para programación paralela que se puede utilizar en la implementación del programa paralelo. Un programa paralelo se puede organizar de diferentes maneras, de acuerdo al orden observable en el algoritmo a utilizar y los datos.

Sin embargo, existe la posibilidad de que este criterio arroje para un cierto tipo de problema más de un patrón arquitectónico para programación paralela, y la selección del patrón depende generalmente de la experiencia del desarrollador. Esto es debido a que no se tiene una manera cuantitativa para poder respaldar tal elección.

1.3. Hipótesis

Dado un problema expresado en términos de algoritmo y datos, un conjunto de patrones arquitectónicos para programación paralela, un número variable de procesadores y procesos por procesador, ¿Es posible determinar con un análisis comparativo del desempeño de los patrones cuál ofrece un mayor desempeño sobre el otro?

1.4. Objetivo

Determinar entre los patrones arquitectónicos para programación paralela *Communicating Sequential Elements* y *Manager-Workers* cuál ofrece un mejor desempeño para un problema determinado. Para ello es necesario dividir el código secuencial del programa y mantenerlo en ambas implementaciones, para así poder inferir que los cambios en los tiempos de ejecución dependen únicamente del tipo de coordinación entre los procesos paralelos especificados en ambos patrones arquitectónicos para programación paralela.

1.5. Aproximación

Dada la problemática descrita en la sección 1.2, es necesario agregar otro criterio para determinar cuál de los patrones arquitectónicos es el más adecuado para organizar un problema dado bajo las condiciones de ejecución presentes (plataforma de HW, plataforma de SW, lenguaje de programación).

En esta tesis se propone utilizar al desempeño (en términos de tiempo de ejecución en segundos) de las aplicaciones paralelas como criterio complementario para la selección de los patrones arquitectónicos para programación paralela. Entonces, dado un problema expresado por un algoritmo y datos, se pretende realizar un análisis comparativo que cuantifique en una aplicación el tiempo de ejecución, y así utilizarlo para seleccionar una organización de procesos paralelos.

1.6. Contribuciones

La principal contribución de la presente tesis es proveer un análisis comparativo entre las implementaciones de dos o más patrones arquitectónicos para programación paralela, analizando comparativamente sus desempeños con el fin de elegir el mejor patrón para un problema dado, expresado en términos de algoritmo y datos.

En la presente tesis se determina qué patrón arquitectónico para programación paralela (*Manager-Workers* ó *Communicating Sequential Elements*) es el ofrece mejor desempeño para la detección de bordes en imágenes digitales a color en RGB en formato BMP de alta definición bajo el algoritmo de Canny.

Además, en la presentación de los resultados se ofrece un análisis comparativo de los dos patrones arquitectónicos para programación paralela implementados el cual describe porqué

un patrón es más adecuado que el otro y bajo qué condiciones.

1.7. Metodología

Para realizar el análisis de desempeño se utiliza un caso de estudio: la detección de bordes en imágenes digitales a color en RGB en formato BMP de alta definición. Para este caso de estudio se elige el algoritmo de Canny de detección de bordes, debido a que es un algoritmo conocido, sencillo de implementar y con las imágenes resultantes se puede verificar si el procesamiento es el correcto.

La metodología para determinar el patrón arquitectónico que se debe aplicar para un caso de estudio en el área de procesamiento digital de imágenes es:

1. Desarrollar un programa secuencial para el caso de estudio. Se realiza un programa secuencial utilizando el algoritmo de Canny para la detección de bordes, con el objetivo de tener una referencia para el desempeño de los programas paralelos.
2. Implementación de cada uno de los patrones arquitectónicos para programación paralela. Para el problema de procesamiento de imágenes utilizando el algoritmo de Canny, comúnmente se utilizan los patrones *Manager-Workers* y *Comunnicating Sequential Elements*.
3. Medición de los tiempos de ejecución para cada patrón, variando el número de procesadores, el número de procesos por procesador y para dos versiones de la misma imagen (con diferente resolución). Una vez implementados los programas paralelos, se ejecutan independientemente utilizando un mismo conjunto de características (arquitectura, lenguaje de programación, número de procesadores, tamaño del problema, granularidad) para obtener los tiempos de procesamiento. A partir de los tiempos de ejecución, se obtienen valores estadísticos como la media (μ), desviación estándar (σ) y la varianza (σ^2) para los tiempos de ejecución de cada programa con cada combinación de características.
4. Obtener las métricas de desempeño: tiempo de ejecución, *Speedup* y eficiencia.
5. Realizar un análisis del desempeño, *Speedup* y Eficiencia a partir de los resultados obtenidos en los puntos anteriores.
6. Realizar los intervalos de confianza para las medias de los tiempos de procesamiento.
7. Analizar los resultados obtenidos de la aplicación secuencial respecto a los resultados de las aplicaciones paralelas.

8. Una vez teniendo todos los resultados anteriores, se procede a realizar el análisis comparativo entre los dos patrones implementados, para poder determinar cuál ofrece mejor desempeño para el caso de estudio tratado en esta tesis.

1.8. Estructura de la tesis

La tesis está organizada de la siguiente manera:

- **Capítulo 2. Antecedentes.** En este capítulo se encuentra la descripción de algunos conceptos importantes para la realización de esta tesis. También se describen los tipos de comuncación entre procesadores, cómo se clasifican las arquitecturas paralelas reales y cómo se clasifican utilizando la clasificación de Flynn [Fly66].

- **Capítulo 3. Trabajo Relacionado.** En este capítulo se presenta un trabajo de procesamiento de imágenes utilizando dos diferentes tipos de paralelismo: paralelismo *pipeline* y paralelismo de datos. También se describe la forma en que los datos pueden ser distribuidos a lo largo de los procesadores del sistema paralelo, así como cuál sería la mejor forma de distribuir los datos dependiendo del problema a resolver. Los dos tipos principales de distribución que se tocan en este artículo es la distribución de datos estática y dinámica.

- **Capítulo 4. Análisis Comparativo.** En este capítulo se describe a detalle el caso de estudio que se utiliza para poder realizar el análisis comparativo del desempeño de patrones arquitectónicos para programación paralela. Se describe el problema que se va a resolver y luego se describe cómo es su implementación, utilizando dos patrones arquitectónicos para programación paralela: *Manager-Workers* y *Communicating Sequential Elements*.

- **Capítulo 5. Evaluación Experimental.** En este capítulo se describen los factores que intervienen en los experimentos. Se establecen las suposiciones que se deben de tener para que los resultados obtenidos de los experimentos reflejen las diferencias entre los dos tipos de coordinaciones tratados en esta tesis. Se describen los experimentos que se realizan para poder obtener los datos que puedan ser analizados para determinar cuál de los dos patrones implementados ofrece mejor desempeño. Se realiza un análisis estadístico sobre los tiempos de ejecución para poder determinar un intervalo de confianza para las medias de estos tiempos. Se describen los resultados obtenidos mediante tablas y gráficas. También se muestran los tiempos de ejecución promedio de todos los experimentos realizados, así como también gráficas en 3D para poder observar cómo cambia el tiempo de ejecución cuando cambian parámetos como número de procesadores, número de procesos por procesdor y tamaño del trabajo. Al fijar el número de

procesadores y el número de procesos por procesador, y sobre todo establecer que cada coordinación ejecuta el mismo código secuencial es posible comparar únicamente los desempeños. Estos desempeños son afectados directamente por el tipo de patrón utilizado.

Además, se realiza el análisis del desempeño, del *Speedup* y la Eficiencia. Este análisis sirve para determinar cuál patrón arquitectónico para programación paralela que ofrece mejor desempeño para el caso de estudio de esta tesis, y también para obtener información acerca de la implementación del programa paralelo.

- **Capítulo 6. Conclusiones y Trabajo futuro.** Este capítulo presenta las conclusiones generales y específicas del presente trabajo de tesis. Para lo anterior se retoma la hipótesis planteada en el capítulo 1 y se contrasta con los resultados obtenidos. Además se describe el trabajo futuro para dar seguimiento a este trabajo.

Capítulo 2
Antecedentes

El objetivo de este capítulo es presentar una revisión de (a) Cómputo Paralelo, qué es, cómo se clasifica, cómo se puede medir su desempeño, (b) conceptos acerca de la programación paralela y (c) descripción de cada uno de los patrones arquitectónicos para programación paralela.

2.1. Cómputo paralelo

El cómputo paralelo es *la especificación de un conjunto de procesos ejecutándose simultáneamente, y que se comunican entre sí para alcanzar un objetivo común* [OA09]. Un aspecto general de este concepto se puede ver en la Figura 2.1. El principal objetivo es acelerar el desempeño de los programas que se ejecutan sobre una computadora secuencial.

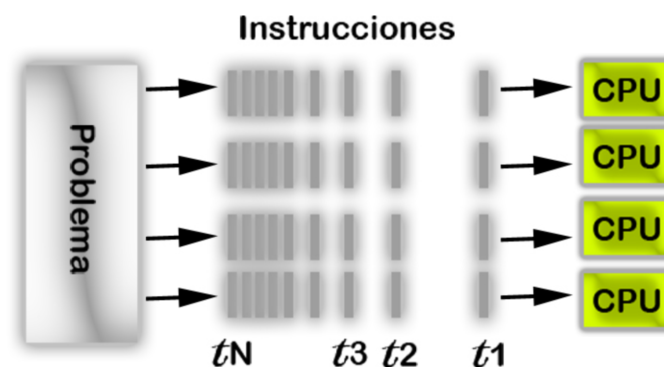

Figura 2.1: Cómputo paralelo.

El cómputo paralelo puede ser alcanzado a través de computadoras paralelas, las cuales cuentan con varias unidades de procesamiento que trabajan simultáneamente y de forma coordinada para resolver un mismo problema computacional. Las unidades de procesamiento pueden estar incluidas en una misma computadora (sistema multiprocesador) o en computadoras independientes interconectadas a través de una red de conexión (sistema multicomputadora). Estos dos sistemas se describen en la sección a continuación.

2.1.1. La clasificación de Flynn

En 1966, Michael J. Flynn clasificó a las computadoras paralelas. Esta clasificación se basó en dos principales aspectos [DC98]:

- **Flujo de instrucciones** Es la secuencia de instrucciones que es ejecutada por una computadora.
- **Flujo de datos** Es la secuencia de datos que es llamada por el flujo de instrucciones.

A partir de estas dos conceptos, Flynn organizó a las computadoras de la siguiente manera [Fly66]:

	Simple Flujo de Datos	Múltiple Flujo de Datos
Simple Flujo de Instrucciones	SISD	SIMD
Múltíple Flujo de Instrucciones	MISD	MIMD

Figura 2.2: Clasificación de Flynn de procesamiento paralelo.

Simple flujo de Instrucciones, Simple flujo de Datos (SISD)

Esta arquitectura representa la arquitectura convencional de una computadora de Von Neumann [DC98]. Esta arquitectura SISD representa a las computadoras uniprocesador ordinarias [Par99].

Simple flujo de Instrucciones, Múltiple flujo de Datos (SIMD)

Este tipo de arquitectura se desarrolló debido a la necesidad de desempeñar un mismo conjunto de instrucciones sobre diferentes arreglos de datos, como por ejemplo en el procesamiento de imágenes, donde a cada imagen se le puede ver como un arreglo de pixeles y a cada imagen se le pueden aplicar las mismas técnicas de procesamiento, como pueden ser filtros, transformaciones, etc.

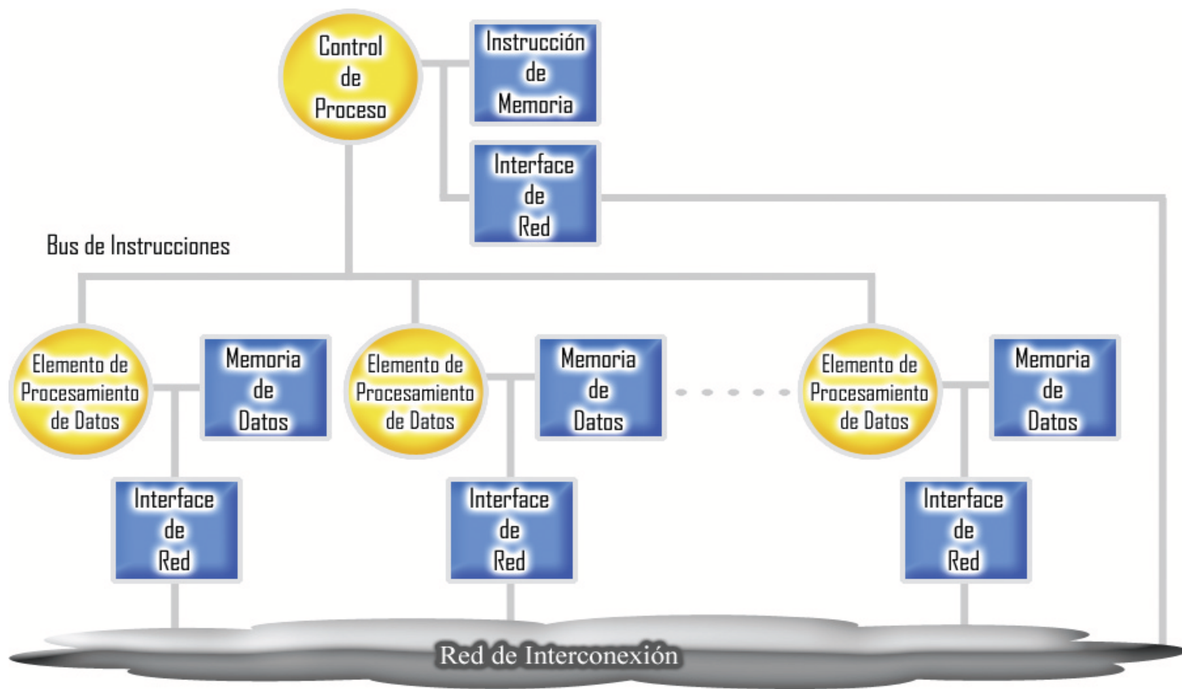

Figura 2.3: Diagrama SIMD.

Múltiple flujo de Instrucciones, Simple flujo de Datos (MISD)

A este tipo de computadora paralela no se ha encontrado alguna aplicación. La razón más importante es que la mayoría de las aplicaciones no se mapean fácilmente a este tipo de arquitectura. Sin embargo, se puede considerar una arquitectura tipo MISD de procesadores paralelos para ciertas aplicaciones [Par99].

En la Figura 2.4 se muetra un ejemplo de un procesador paralelo con arquitectura tipo MISD. Donde un simple flujo de datos entra a una máquina con cinco procesadores.

Cada procesador realiza varias operaciones sobre un cierto dato antes de pasarlo al siguiente procesador.

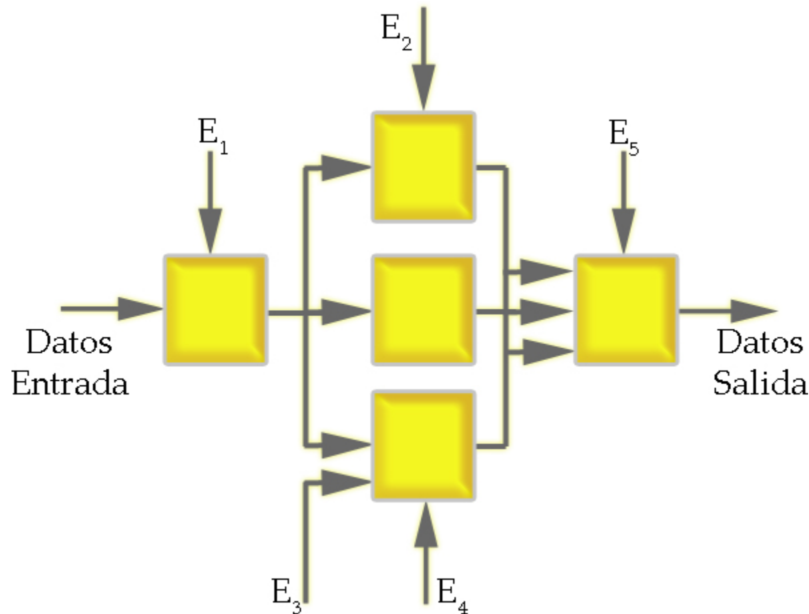

Figura 2.4: Múltiple flujo de instrucciones operando sobre un simple flujo de datos (MISD).

Múltiple flujo de Instrucciones, Múltiple flujo de Datos (MIMD)

Las computadoras paralelas de este tipo son computadoras individuales que se pueden clasificar en dos tipos dependiendo del hardware que viene a daterminar la manera con que las computadoras cooperan entre sí.

La arquitectura MIMD se clasifica en:

• **Sistema Multiprocesador con Memoria Compartida**:
En una computadora convencional el procesador ejecuta un programa que se encuentra alojado en memoria principal. Para acceder a una localidad específica de la memoria se localiza a través de un número llamado dirección de memoria. El direccionamiento empieza en 0 y termina en 2^n-1 cuando hay n bits que direccionar [WA04].

El modelo uniprocesador se puede extender agregando procesadores, para lo cual se necesita intercomunicar los módulos de memoria de cada procesador. Si cada procesador puede

acceder a cualquier módulo de memoria se le denomina *Memoria Compartida*. Los procesadores comparten una memoria en común donde todos los procesadores pueden leer y escribir [BJD00].

Los procesadores pueden acceder a múltiples módulos de memoria que se encuentran interconectados. El conjunto de todos los módulos forman una sola memoria global y tienen un *Único Espacio de Direcciones*, lo que significa que cada locación de memoria en cada módulo tiene una única dirección, como se muestra en la Figura 2.5.

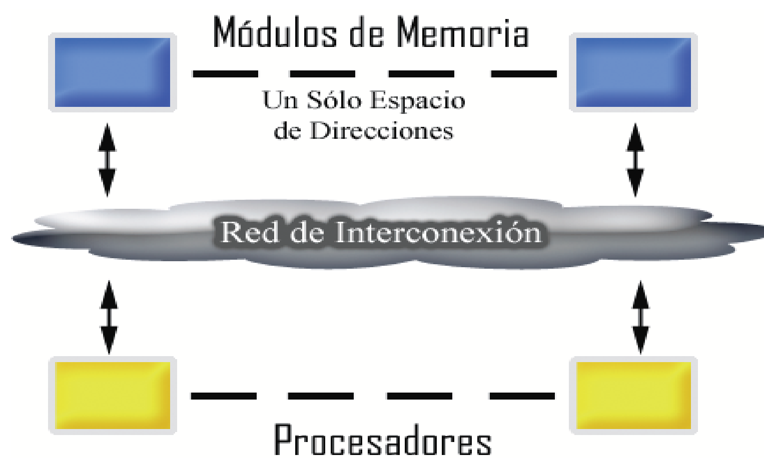

Figura 2.5: Modelo Multiprocesador con memoria compartida.

• **Multicomputadoras con Memoria Distribuida**:
En un sistema multicomputadora, la memoria se encuentra distribuida a lo largo de las computadoras, donde cada computadora posee su propio espacio de direcciones [WA04]. Cada procesador sólo puede acceder a su propio espacio de direcciones. Para que los procesadores se puedan comunicar, se envían mensajes entre ellos a través de la red de interconexión. Los mensajes pueden contener datos de otros procesadores que se requieren para realizar alguna operación computacional. A este tipo de sistema se le conoce comúnmente como *Multiprocesadores*, o simplemente *Multicomputadoras*, el cual se muestra en la Figura 2.6.

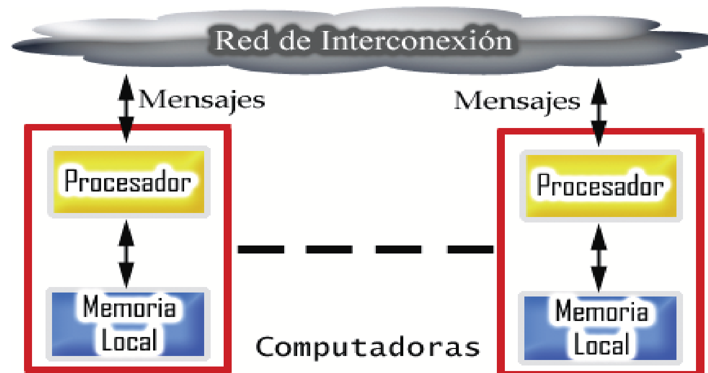

Figura 2.6: Modelo Multicomputadoras con memoria distribuida.

Memoria Compartida Distribuida En el *Sistema de Memoria Compartida Distribuida*, la memoria se encuentra distribuida a lo largo de las computadoras al igual que en el *Sistema Multicomputadoras*. La diferencia es que en esta arquitectura, se tiene un *Único Espacio de Direcciones* para acceder a todas las memorias.

Cuando un procesador trata de acceder a una localidad de memoria que no está dentro de su memoria local, necesita usar mensajes para pasar datos que están alojados en la memoria local de otro procesador. Para el procesador es transparente la ubicación de la localidad de memoria que quiere acceder, debido a que da la ilusión de *Memoria Compartida* aunque esté de manera *distribuida*, a esta idea se le llamó *Memoria Virtual Compartida* [WA04], mostrado en Figura 2.7.

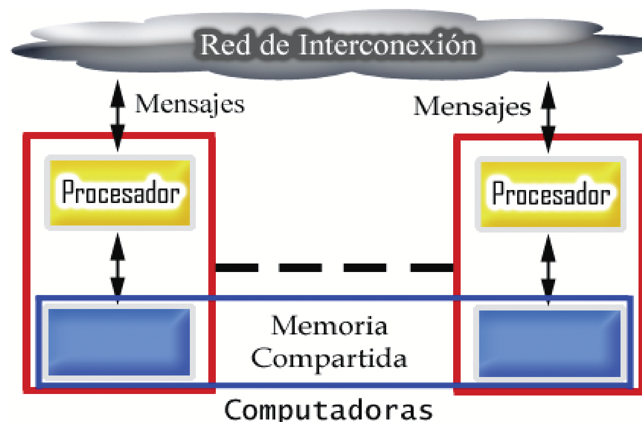

Figura 2.7: Modelo Multicomputadoras con Memoria Compartida.

2.1.2. Arquitecturas Paralelas Reales

Se pueden encontrar diferentes tipos de arquitecturas paralelas reales, entre las más comunes están [HX98]:

Symmetric Multiprocessor (SMP)

Entre los sistemas que tienen este tipo de arquitectura se encuentran: la IBM R50, el SGI Power Callenge, y la DEC Alpha server 8400 [HX98]. Esta arquitectura se puede observar en la Figura 2.8.

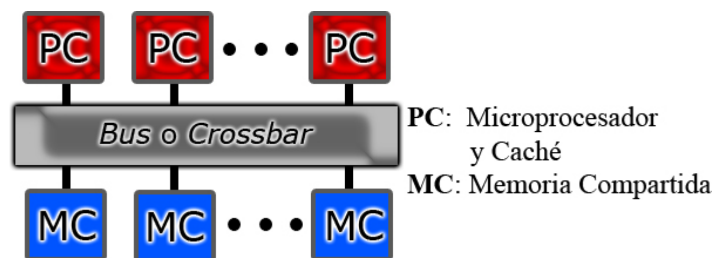

Figura 2.8: Arquitectura *Symmetric Multiprocessor*.

Este tipo de arquitectura posee memoria compartida (MC); esto es la capacidad de acceso múltiple a la misma locación de memoria por los procesadores (PC)[GL95].

Parallel Vector Processor (PVP)

La Cray C-90, Cray T-90, y NEX SX-4 son ejemplos de computadoras con esta arquitectura [HX98]. La arquitectura PVP se puede observar en la Figura 2.9.

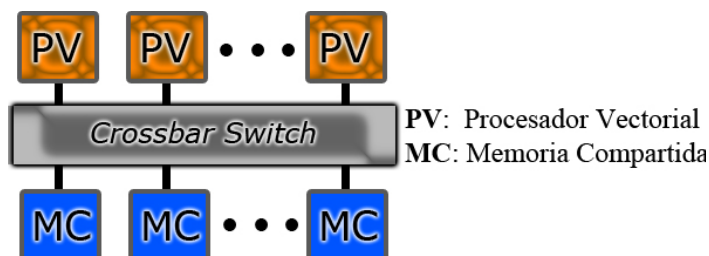

Figura 2.9: Arquitectura *Parallel Vector Processor*.

El *crossbar switch* permite a los procesadores vectoriales (PV) acceder a los múltiples módulos de memoria compartida.

Distributed Shared Memory machine (DSM)

Esta es una arquitectura similar a la SMP. La diferencia es que la memoria está distribuida entre los nodos del sistema. Sin embargo, se cuenta con un sistema de hardware y software que hace que la memoria se vea como una sola, es decir, se tiene un mismo espacio de direcciones [HX98]. La arquitectura DSM se puede observar en la figura 2.10.

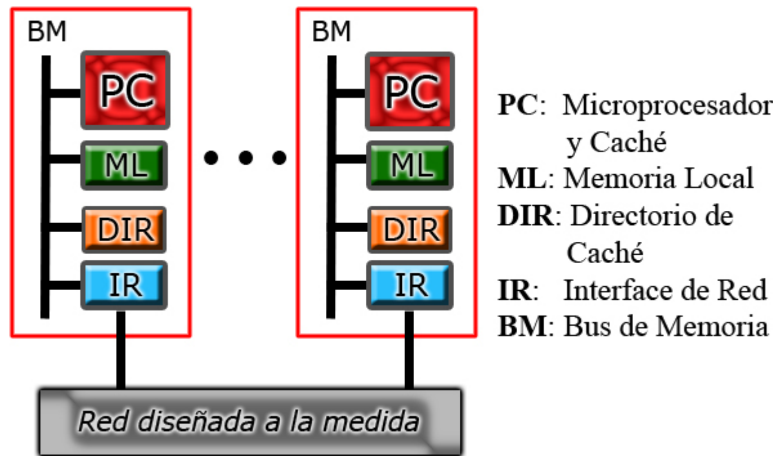

Figura 2.10: Arquitectura *Distributed Shared Memory Machine*.

Massively Parallel Processor (MPP)

Las computadoras con esta arquitectura cuentan con las siguientes características [HX98]:

- Cada nodo cuenta con su propio microprocesador.
- Usa memoria distribuida.
- Utiliza interconecciones con alto ancho de banda y baja latencia.
- Puede escalar a cientos o hasta miles de procesadores.
- Los procesos son sincronizados mediante funciones de paso de mensajes bloqueantes.
- El programa cuenta con múltiples procesos, los cuales tienen su propia memoria privada.

La estructura de la arquitectura MPP se puede observar en la Figura 2.11.

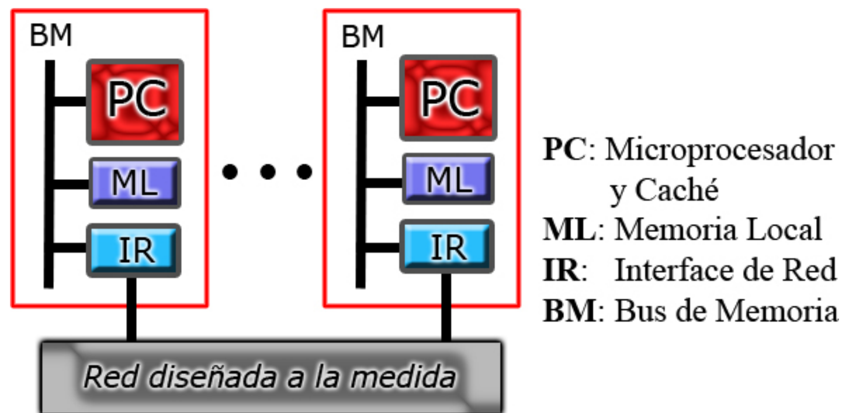

Figura 2.11: Arquitectura *Massively Parallel Processor*.

Cluster of Workstations (COW)

Algunos ejemplos de sistemas con esta arquitectura son: Digital's TruCluser, IBM SP2, y la Berkeley NOW. Esta arquitectura es una variación de los MPP pero de bajo costo. Las principales distinciones de los *clusters* son [HX98]:

- Cada nodo es un workstation completo, pero sin algunos periféricos como teclado, mouse, monitor, etc.
- Los nodos están conectados a través de una red de bajo costo, como Ethernet, FDDI, canal de fibra, etc.
- Se cuenta con un disco local.
- Cada nodo cuenta con su propio sistema operativo, mientras que en los MPPs sólo tienen un microkernel.

La arquitectura COW se muestra en la Figura 2.12.

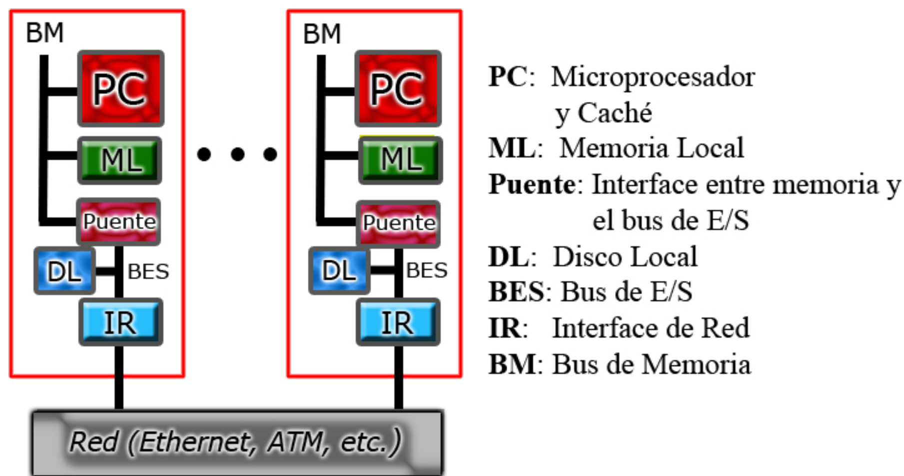

Figura 2.12: Cluster of Workstations.

2.2. Programación Paralela

Es la especificación de un conjunto de procesos que se ejecutan simultáneamente y que se comunican entre sí obtener un objetivo común. El objetivo es resolver el problema computacional más rápido [And00].

Actualmente, el desarrollo del software paralelo se encuentra rezagado con respecto al avance tecnológico en el hardware paralelo. Este rezago es debido a que la programación paralela es más compleja a comparación de la programación secuencial [HX98].

Existen ciertas limitantes en la construcción de programas paralelos, entre las cuales están [HX98]:

- Existencia de diferentes arquitecturas paralelas.
- Los compiladores y los *debugger* son herramientas mucho más avanzadas para programas secuenciales.
- La experiencia obtenida en la programación secuencial es mucho mayor a la que se tiene en la programación paralela.

2.2.1. Modelos de algoritmos paralelos

En la descripción del problema, se identifica el orden de los datos y de las operaciones que lleva a diseñar el software, es decir, indica cómo el cómputo paralelo tiene que ser realizado. De acuerdo con la descripción del problema, es posible considerar que la mayoría de las aplicaciones paralelas caen en alguna de las siguientes formas de paralelismo [OA10]:

Paralelismo de Dominio Este tipo de paralelismo divide el flujo de datos entre los procesadores del sistema en tareas que se asignan a diferentes procesadores.

Paralelismo Funcional Este tipo de paralelismo se basa en tener distintos bloques funcionales en la aplicación. Estos bloques se logran al descomponer el algoritmo, y estos bloques pueden ser asignados a diferentes procesadores [Roo99].

En el paralelismo funcional todas las tareas inician simultáneamente, esperando que cada uno de los procesadores ya cuenten con su conjunto respectivo de datos que debe procesar. En cada paso del procesamiento se realizan cambios sobre los datos.

Paralelismo de Actividad Este tipo de paralelismo particiona tanto a los datos como al algoritmo [CG89] [Pan96]. El algoritmo se divide en diferentes tareas, donde cada tarea puede considerarse como un "*worker*"que es capaz de tomar una partición de los datos y procesarla.

2.3. Patrones Arquitectónicos

Un patrón arquitectónico es una descripción organizacional fundamental de una estructura de alto nivel observada en un grupo de sistemas de software. Pueden ser vistas como plantillas que expresan y especifican propiedades estructurales de los subsistemas, así como las responsabilidades y relaciones existentes entre estas.

2.4. Patrones Arquitectónicos para Programación Paralela

Los patrones arquitectónicos para programación paralela son las estructuras organizacionales básicas comúnmente utilizadas en la programación paralela para componer sistemas de software paralelo [OA10].

La selección de un patrón arquitectónico es considerado como una decisión fundamental durante el diseño de la estructura de un sistema de software [BMR+96], [Sha95]. Esta selección del patrón para programación paralela depende de la partición del algoritmo y/o los datos [OA10].

A continuación, se describen brevemente estos patrones arquitectónicos.

2.4.1. Patrón *Manager-Workers*

Descripción Este patrón es usado particularmente cuando el problema contiene paralelismo de actividad. Realiza las mismas operaciones simultánea e independientemente sobre distintos conjuntos de datos. Es una variante del patron *Master-Slave* para sistemas paralelos [BMR+96] [OA10].

El *Manager* se encarga de toda la gestión de los datos, es decir, se encarga de la segmentación de los datos, de proveer a cada *worker* el segmento correspondiente de datos que debe procesar, así como también de la obtención de los datos resultantes de cada *worker*.

Elementos *Manager* y *Workers*.

Estructura La estructura de este patrón consiste de un *Manager* encargado de proveer los datos en un cierto orden a los *Workers*. Estos *workers* realizan el mismo conjunto de operaciones (Figura 2.13).

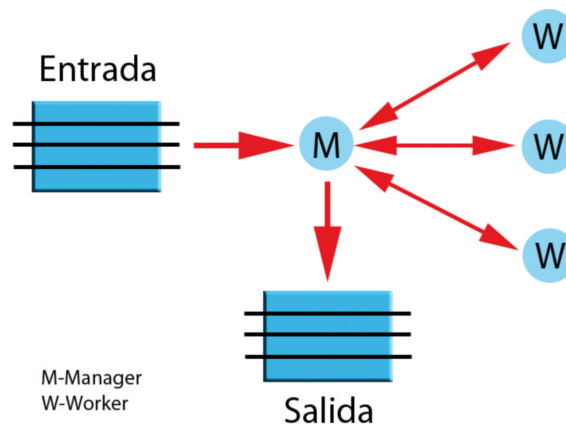

Figura 2.13: Patrón *Manager-Workers*.

2.4.2. Patrón *Parallel Pipes & Filters*

Descripción Es un patrón arquitectónico de paralelismo funcional para programación paralela. Los *Filters* realizan simultáneamente distintas operaciones sobre diferentes conjuntos de datos de entrada. Al terminar cada *filter* con sus operaciones respectivas, envía sus datos resultantes al siguiente *filter* a través de los *Pipes* [OA10].

Elementos *Pipes* y *Filters*.

Estructura La estructura de este patrón es una serie de *filters* interconectados de forma serial mediante *pipes* que se encargan proveer los datos de entrada a cada *filter* (Figura 2.14).

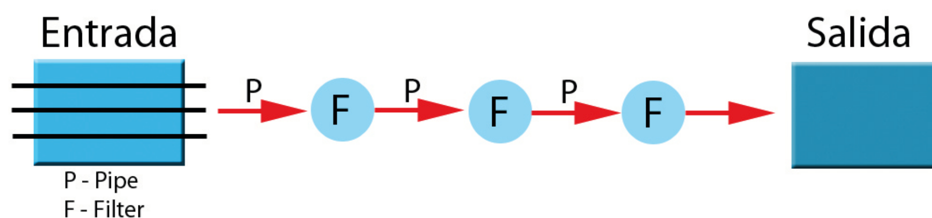

Figura 2.14: Patrón *Pipes & Filters*.

2.4.3. Patrón *Parallel Layers*

Descripción El patrón arquitectónico para programación paralela *Parallel Layers* resuelve problemas con paralelismo funcional. El paralelismo se realiza cuando dos o más componentes de una capa realizan las mismas operaciones simultáneamente sobre conjuntos de datos independientes [OA10].

Los componentes de cada capa pueden ser creados estáticamente al empezar la aplicación paralela o también pueden ser creados dinámicamente cuando alguna de las capas superiores solicitan su creación.

Elementos *Layers*.

Estructura La estructura de este patrón es en forma arborecente, y tiene un comportamiento jerárquico (Figura 2.15).

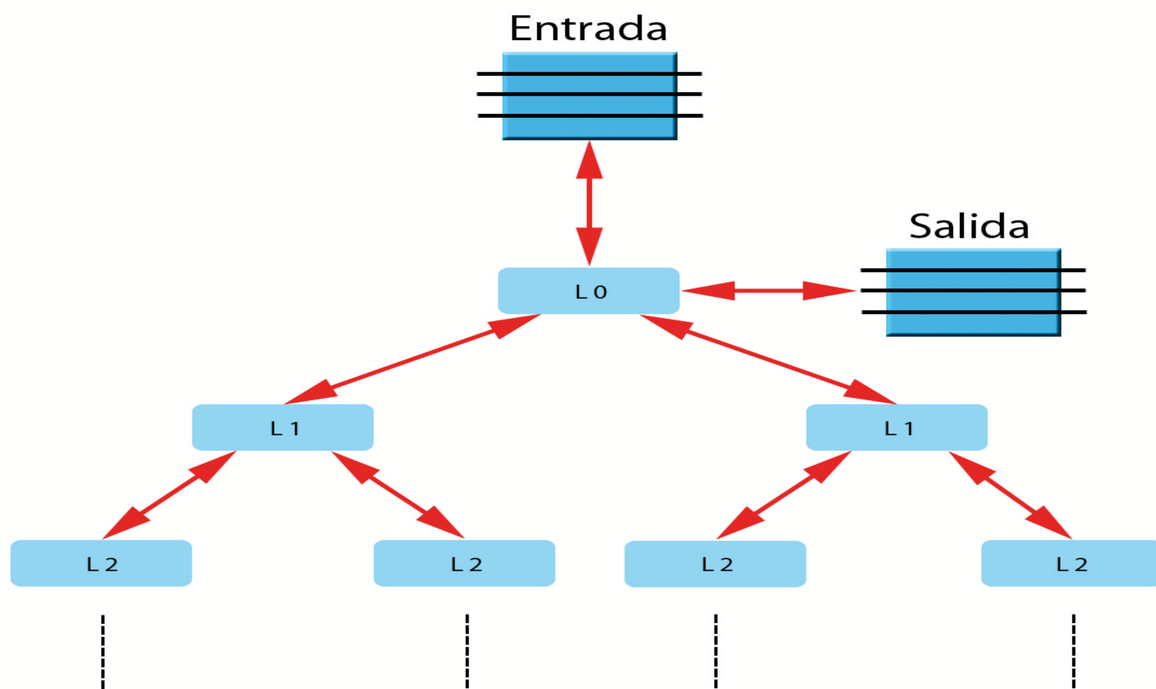

Figura 2.15: Patrón *Parallel Layers*.

2.4.4. Patrón *Communicating Sequential Elements*

Descripción Este patrón se usa para solucionar problemas de paralelismo de dominio. Cada elemento de procesamiento realiza las mismas operaciones sobre diferentes segmentos de datos de manera simultánea, donde cada elemento de procesamiento depende de los resultados parciales o finales de sus vecinos [OA10].

Elementos Elementos de procesamiento y canales de comunicación.

Estructura Existen varias formas con las que se puede estructurar este patrón, la estructura depende principalment de la forma en la que se dividen los datos, y de la información adicional que se necesita de los demás elementos de procesamiento (Figura 2.16).

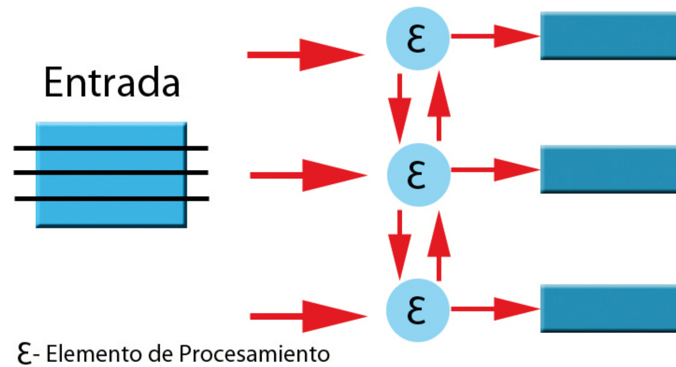

Figura 2.16: Patrón *Communicating Sequential Elements*.

2.4.5. Patrón *Shared Resource*

Descripción El patrón Shared Resource se utiliza cuando el problema contiene paralelismo de actividad. El recurso compartido es dividido lógicamente en segmentos que los procesos accionistas podrán acceder sin ningún tipo de restricción. Es por esto que los procesos accionistas deben de tener bien definido su dominio de trabajo para que no pueda leer y/o escribir en algún segmento de datos que no le corresponda. Cada proceso accionistas realiza operaciones diferentes e independientes sobre el segmento de datos que le corresponde [OA10].

Elementos Recurso compartido y Procesos accionistas.

Estructura El recurso compartido es accesible por todos los procesos accionistas, cada quien es responsable de trabajar exclusivamente en su correspondiente segmento de datos (Figura 2.17).

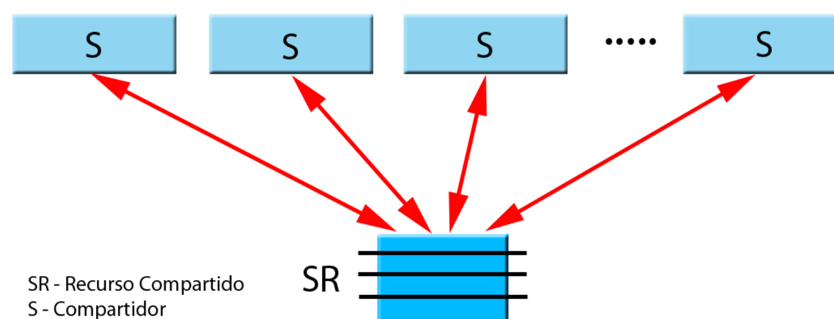

Figura 2.17: Patrón *Shared Resource*.

2.5. Paso de Mensajes

Paso de mensajes es la transferencia de datos entre procesos o hilos mediante operaciones de envío y recepción entre los procesos en comunicación [GL95]. Las operaciones de envío y recepción se detallan en la Sección 2.5.1, y pueden realizarse de diferentes maneras, como se describe a continuación:

- **Comunicación Asíncrona** Es un estilo de comunicación en la cual los nodos son libres de enviar mensajes a cualquier otro nodo en cualquier momento sin considerar la interconexión física.

- **Comunicación Bloqueante** En este estilo de comunicación las operaciones de envio de mensajes bloquea el programa hasta que el mensaje se haya recibido, y para las operaciones de recepción, se bloquea hasta que en nodo transmisor emita el mensaje que se especifica.

- **Comunicación No Bloqueante** En la comunicación no bloqueante las operaciones de envio depositan en mensaje en un buffer y continúan con la ejecución del programa, mientras que la recepción de mensajes realiza una lectura del estado del *buffer* que indica si se ha recibido o no el mensaje.

- **Comunicación Colectiva** Realizan las operaciones de envio y recepción a grupos de nodos, pueden ser por *broadcast* o por *multicast*.

2.5.1. Interface de Paso de Mensajes (MPI)

MPI es una biblioteca donde se encuentran contenidas rutinas de paso de mensajes. Existen lenguajes de programación secuencial que no cuentan con este tipo de funciones, como *C* y *Fortran*. La inclusión de MPI permite la comunicación y sincronización entre procesos de un programa distribuido.

Cada procesador ejecuta una copia del mismo programa. En el programa están definidas las acciones que debe seguir cada procesador por medio de su identificador. Cuando se requieren comunicar los procesos, utilizan funciones de MPI. Las comunicaciones entre procesos pueden ser punto a punto, grupales (*multicast*) ó global (*broadcast*).

Una de las características principales de MPI es la habilidad de ejecución en sistemas heterogéneos, es decir, puede haber comunicación entre procesadores aún cuando estén dentro de arquitecturas diferentes [SO98].

Funciones Básicas

En todo programa distribuido con MPI se necesita usar ciertas funciones para poder determinar ciertas características del ambiente en donde se va a trabajar Las principales funciones son [SO98]:

- `MPI_Init`.- Esta función debe ser ejecutada antes de utilizar alguna otra función de la biblioteca MPI.
- `MPI_COMM_WORLD`.- Es el comunicador por defecto donde define un dominio de comunicación inicial para todos los procesos relacionados en el cómputo.
- `MPI_Comm_size`.- Función que sirve para obtener el número de procesos dentro de un grupo local.
- `MPI_Comm_rank`.- Función que devuelve la posición del proceso dentro de un grupo local.
- `MPI_Finalize`.- Esta función limpia los estados de MPI. Después de haber limpiado el ambiente de MPI no se puede volver a invocar a la función `MPI_Init`.

A continuación se muesta un ejemplo de un programa general en C utilizando las operaciones básicas de MPI

```
#include <stdio.h>
#include <mpi.h>
int main (argc, argv){
     int argc;
     char *argv[];
     int rank, size;
     /* inicia MPI */
     MPI_Init (&argc, &argv);
     /* se obtiene el ID del proceso actual */
     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
     /* se obtiene el # número de procesos */
     MPI_Comm_size (MPI_COMM_WORLD, &size);
     printf( "Hola mundo del proceso %d de %d\n", rank, size );
     /* termina MPI */
     MPI_Finalize();
     return 0;
}
```

Al ejecutar este programa en un *cluster* de computadoras se obtiene en la pantalla de salida de la computadora que ejecutó el programa un conjunto de lineas generadas por todos los procesos creados a lo largo del cluster con la leyenda *"Hola mundo del proceso "* más su respectivo identificador y *"de "* más el número de procesos totales.

Funciones para enviar y recibir mensajes

Entre las funciones que provee la biblioteca MPI para el envío y recepción de mensajes están:

- `MPI_Bcast`.- Realiza la emisión de un dato desde el nodo raíz a todos los demás procesos dentro de un grupo específico.
- `MPI_Send`.- Función para enviar datos punto a punto.
- `MPI_Recv`.- Función de envío de datos de un modo bloqueante.
- `MPI_Isend`.- Función para el envío de datos de modo no bloqueante.
- `MPI_Irecv`.- Función para el recepción de datos de modo no bloqueante.

Para realizar la comunicación (envío/recepción), es necesario determinar el tipo de dato que se envía o recibe, además de una etiqueta asociada a este dato (TAG) [SO98].

La diferencia entre las funciones bloqueantes y no bloqueantes es que en las bloqueantes se suspende la ejecución del programa cuando se llega a una función de envío o recepción, hasta que en el otro proceso (con el que quiere comunicar) ejecute una instrucción de recepción o envío. Para el caso de las funciones no bloquentes, se utiliza un *buffer* donde se aloja el mensaje que se quiere transmitir cuando se ejecuta una función de envío y continúa con la ejecución del programa, y si es el caso de recibir un dato, el proceso receptor verifica que en el *buffer* esté el mensaje que requiere, si está lo toma y continúa, y si no está sigue con la ejecución del programa.

En el Apéndice B se observan dos ejemplos utilizando paso de mensajes bloqueante y no bloqueante.

2.6. Métricas de desempeño

Existen distintas maneras para medir el desempeño de un programa paralelo. Entre las métricas más comunes están: el tiempo de ejecución, el *Speedup* y la eficiencia [KF90].

2.6.1. Tiempo de Ejecución

El tiempo de ejecución serial de un programa es el tiempo trancurrido entre el inicio y el final del programa en una máquina secuencial, denotado como T_s. En contraste, el tiempo de ejecución paralelo es el tiempo trancurrido desde el momento de que se inicia el primer procesador hasta que el último termine la ejecución, denotado como T_p [GKKG03].

2.6.2. *Speedup*

Para calcular el *Speedup* (Sp) se necesita medir el tiempo de ejecución de un mismo programa con diferente número de procesadores.

Sp está dada por [FP92]:

$$Sp = \frac{T(o)}{T(p)},$$

donde T_0 es el tiempo de ejecución del programa serial más rápido sobre un sólo procesador. En esta definición de *Speedup* se observa que se compara a un programa paralelo con el programa serial más rápido para un mismo problema. Esta medida ofrece la ganancia que se obtiene de llevar un programa de una máquina secuencial con un procesador a un sistema paralelo con p procesadores idénticos. Si es el mismo algoritmo empleado para los dos tipos de programas, se espera que el tiempo de ejecución (T_1) del programa paralelo sobre un procesador sea mayor que el tiempo de ejecución de la implementación secuencial (T_0) sobre un procesador debido al *overhead* asociado en la ejecución de los procesos paralelos [FP92].

Speedup Algorítmico Algorítmico

El *Speedup* algorítmico es igual al tiempo de ejecución de un programa paralelo en un procesador entre el tiempo de ejecución en n procesadores [KF90] [FP92].

$$Sp = \frac{T(1)}{T(p)}$$

donde Sp es el *Speedup* y p es el número de procesadores.

Cuando el *overhead* incrementa, el *Speedup* decrece.

Speedup Ideal o Lineal Ideal o Lineal

Esta clase de *Speedup* es alcanzado cuando $Sp = p$, es decir, cuando el *Speedup* es igual al número de procesadores (p) utilizados en el procesamiento.

Speedup Super Lineal Super Lineal

Cuando el *Speedup* alcanzado es mayor que n sobre un sistema de n procesadores se dice que se obtuvo un *Speedup* super lineal. Existen dos principales razones que explican el porqué ciertas aplicaciones pueden presentar un *Speedup* superlineal [Car01]:

- Incremento del tamaño de la memoria caché: En un sistema multiprocesador cada procesador puede tener tanta memoria caché como la tiene el procesador en un sistema uniprocesador. Entonces, la memoria caché total en un sistema multiprocesador puede ser más grande que la memoria caché de un sistema uniprocesador.
- Mejor estructura: Algunos programas desempeñan menor trabajo cuando es ejecutado sobre un sistema multiprocesador, esto permite alcanzar el *Speedup* superlineal.

Limitaciones en el *Speedup*

En un sistema multiprocesador ideal se debería de obtener un *Speedup* lineal. Sin embargo en la práctica solo se puede esperar este comportamiento cuando se utilizan una cantidad pequeña de procesadores. Cuando el número de procesadores se incrementa, la curva del *Speedup* se aleja del *Speedup* lineal de forma decresiente. Existen tres principales razones de la obtención de un menor *Speedup* que el *Speedup* lineal [Car01]:

- La comunicación interprocesador: En un sistema multiprocesador existe la necesidad de comunicar datos y resultados del cómputo realizado de un procesador a algún otro, y esta comunicación toma un cierto tiempo.
- La sincronización: La sincronización se encarga de garantizar que todos los procesadores hayan terminado una determinada fase del procesamiento antes de comenzar alguna otra.
- El balanceo de carga: En muchas aplicaciones paralelas es necesario dividir la carga del trabajo con respecto al número de procesadores tal que cada procesador procese su trabajo asignado en un tiempo similar a los demás. Cuando esta división no puede llevarse a cabo, pueden existir procesadores que terminen su trabajo correspondiente antes que otros, y mientras se quedan en un estado de ocio esperando que todos los demás procesadores terminen. Esta distribución desigual de trabajos incrementa el tiempo de ejecución total del procesamiento, debido a que existen procesadores sin usarse en parte del proceso.

Adicionalmente se agrega un nuevo que puede ayudar en la obtención de un menor *Speedup*:

- La granularidad: La granularidad está definida por el tamaño de los segmentos en las que los datos originales se particionaron. Una granularidad fina podría aumentar el costo de las comunicaciones, mientras que una granularidad gruesa aumenta el costo de cómputo. Entonces se debe encontrar un balance entre estos dos tipos de granularidades.

2.6.3. Eficiencia

Eficiencia es una medida de la fracción de tiempo en la cual un elemento de procesamiento es útilmente empleado [GKKG03]. Se calcula con la razón del *Speedup* respecto al número de procesadores p.

$$e = \frac{T(1)}{pT(p)} = \frac{Sp}{p}$$

donde e representa la eficiencia.

Overhead

El overhead es la cantidad de tiempo que se va utilizando en el programa paralelo para realizar operaciones de comunicación, E/S, mediciones de tiempos, llamadas al sistema o simplemente en ocio y no para la resolución del problema.

2.7. Resumen

En este capítulo se presenta una introducción del cómputo paralelo, una descripción de la clasificación según Flynn así como también una revisión de algunas arquitecuras paralelas reales como son *Symmetric Multiprocessors*, *Parallel Vector Processor*, *Distributed Shared Memory machine*, entre otras.

Además se describen las funciones básicas de la biblioteca MPI para el paso de mensajes entre computadoras y los diferentes tipos de comunicación que puede haber.

También se describe cada uno de los patrones arquitectónicos para programación paralela [OA10]. Además, se revisan las métricas *Speedup* y Eficiencia que se deben utilizar para poder medir el desempeño de las aplicaciones paralelas.

Capítulo 3
Trabajo relacionados

En este capítulo se describe: (a) una aproximación de paralelismo *Pipeline* para el procesamiento digital de imágenes, (b) una aproximación de paralelismo de datos para el procesamiento digital de imágenes, así como una revisión del paralelismo estático de datos contra el paralelismo dinámico de datos.

3.1. Arquitecturas paralelas para procesamiento de imágenes

En este artículo se encuentra un estudio de dos técnicas de paralelismo aplicadas en programas de procesamiento de imágenes [DC98]:

- Paralelismo Pipeline.
- Paralelismo de Datos.

Cada uno de estos tipos de paralelismo tienen sus ventajas propias, pero a menudo se puede hacer una combinación de estos para obtener aún mejores resultados. La combinación consiste en particionar la aplicación del procesamiento de imágenes en un *pipeline* con etapas concurrentes, después se realiza el paralelismo de datos al segmentar la o las imágenes en pedazos de datos y estos serán los que se procesen dentro del *pipeline* con el fin que sean procesados de manera concurrente [DC98].

3.1.1. Paralelismo Pipeline

En paralelismo *pipeline* se basa en la premisa de que el procesamiento digital de imágenes consta de un conjunto de diferentes tareas ordenadas que se aplican secuencialmente sobre una imagen. A veces, cada una de estas tareas puede realizarse de forma concurrente por distintos procesadores, como se puede observar en la Figura 3.1. Cada distinta tarea puede ser mapeada a un distinto procesador [DC98].

Figura 3.1: Paralelismo Pipeline.

Para procesar una colección de imágenes, primero se envia la primera imagen al primer procesador del *pipeline* que realiza la primea tarea. Este procesador, al terminar su respectiva tarea para la primer imagen, envía sus resultados al siguiente procesador, mientras que está en espera de recibir la siguiente imagen de la colección, hasta se procesen todas las imagenes [DC98].

Este tipo de paralelismo da una mayor rapidez en el procesamiento, debido a que cuando está lleno el pipeline se van arrojando resultados parciales en la etapa siguiente. Sin embargo, este tipo de paralelismo presenta dos principales desventajas [DC98]:

- la tasa de rendimiento está determinada por la tarea más lenta del *pipeline* y
- la aplicación no es escalable, debido a que es imposible segmentar la imagen para que pueda ser procesada por un número arbitrario de tareas concurrentes.

3.1.2. Paralelismo de Datos

En este tipo de paralelismo, todos los elementos de procesamiento realizan las mismas tareas. En el caso de procesamiento de imágenes, se divide la o las imágenes en subimágenes, y se distribuyen a través de todos los elementos de procesamiento. Estos aplican las mismas tareas en un orden específico sobre la subimágen que se le determinó procesar. A veces es implementada sobre una red de procesadores (MIMD). En el procesamiento de imágenes a bajo nivel, predominan las operaciones a nivel de pixel, por ejemplo, las operaciones con los pixeles vecinos; este tipo de operaciones son bien ubicadas en el dominio de la imagen, lo que permite que los procesadores puedan operar más o menos independiente y en paralelo [DC98].

Utilizar el paralelismo de datos para el procesamiento de imágenes a través de un conjunto de procesadores, implica pensar en la forma en la cual se pueden distribuir tales datos. Esta distribución de datos puede ser básicamente de dos formas: Estática y Dinámica [DC98].

Distribución Estática de Datos

En la distribución estática de datos, se divide la imagen original en segmentos, como se muestra en la Figura 3.2. El número de segmentos está determinado por el número de procesadores en el sistema paralelo, con la finalidad de que cada segmento sea asignado a un distinto procesador. Así, cada

procesador realiza el procesamiento sobre su segmento de imagen (estático) correspondiente [DC98].

Esta aproximación del paralelismo de datos es probablemente la más sencilla. Cada segmento puede procesarse de forma independiente con poca comunicación entre los procesadores. Sin embargo, existe un problema cuando se requiere el procesamiento de la primera ó la última fila de un segmento. Este procesamiento realiza operaciones donde se necesitan datos de otro segmento, debido a que los datos que se necesitan están alojados en la memoria de otro procesador. Este problema se resuelve duplicando las filas necesarias en los procesadores vecinos (lógicamente). Por ejemplo, si en la última fila se requiere realizar una operacion usando una máscara de 3×3 se necesita únicamente agregar una fila de más que viene siendo la primera fila del segmento siguiente [DC98].

Sin embargo, los problemas pueden incrementarse en este tipo de distribución de datos debido a que la información contenida de la imagen puede estar distribuida de forma desigual en la misma. Por ejemplo, si se requeire procesar en paralelo características de una imagen como lineas ó regiones en vez de sólo pixeles se puede encontrar el problema de balanceo de cargas. Es debido a este problema que vale la pena considerar una alternativa más dinámica de distribución de datos [DC98].

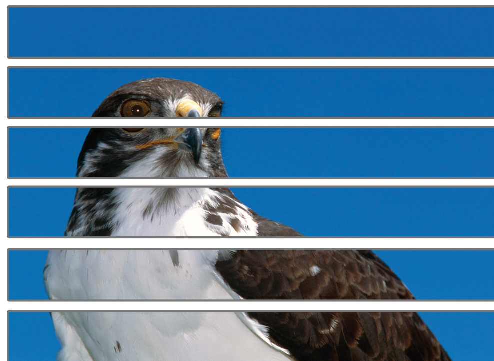

Figura 3.2: Particionamiento de una imagen en segmentos estáticos horizontales.

Distribución Dinámica de Datos

A este tipo de distribución también es conocida como *farming*. Consiste principalmente en tener un procesador encargado del control del *farming*. Este controlador mantiene la imagen original y la divide en pequeños paquetes de trabajo [DC98]. Estos paquetes serán enviados a un conjunto de procesadores trabajadores como se muestra en la Figura 3.3.

Los procesadores trabajadores regresan paquetes de resultados y espera a que les sea enviado un nuevo paquete de trabajo para procesar. Si un trabajador recibe un paquete de trabajo con muy poca carga computacional, significa que pronto terminará su trabajo y estará disponible para recibir un nuevo paquete de trabajo, a diferencia otro trabajador que ha recibido un trabajo de más cómputo

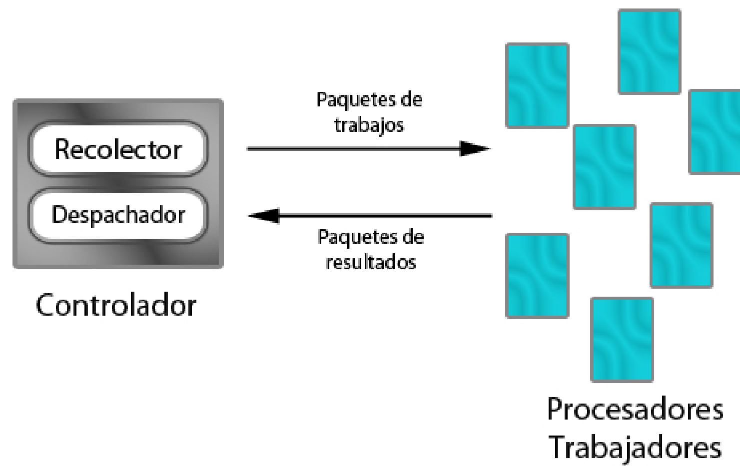

Figura 3.3: Balanceo de carga dinámico utilizando un *farming* de procesadores.

que se mantendrá ocupado por mayor tiempo, por lo que procesará menos paquetes de trabajo [DC98].

Sin embargo, cuando se tienen muchos paquetes de trabajo de poco procesamiento implica que se realizarán muchas comunicaciones. Esto implica un incremento en el *overhead* en el envío de los paquetes de trabajos y resultados lo que puede superar considerablemente el ahorro del balanceo de cargas [DC98].

La elección de la distribución de datos estática ó dinámica depende de la cantidad de cómputo requerido en los paquetes de trabajo y del desbalance de cargas [DC98].

3.2. Resumen

En este capítulo se presenta una introducción de paralelismo *pipeline*, la cual describe la estructura y comportamiento de este tipo de paralelismo aplicado a procesamiento de imágenes. También se mencionan sus principales desventajas que son: la velocidad del procesamiento total depende de la tarea más lenta del *pipeline*, además de que este tipo de paralelismo no es escalable debido a que no se puede descomponer el problema en un número determinado de tareas para que sea asignado cada una a un procesador diferente.

Sin embargo, se ofrece una alernativa a este tipo de procesamiento en paralelo la cual es la distribución de datos; su principal función es dividir los datos en segmentos los cuales puedan ser asignados a diferentes procesadores que pueden realizar las mismas tareas de forma concurrente.

También se describen dos formas de distribuir los datos, una es la distribución de datos estática y la otra es dinámica. La manera de elegir una de otra depende de dos factores, la cantidad de trabajo y del desnivel del balanceo de cargas.

Capítulo 4
Análisis comparativo del desempeño de patrones arquitectónicos para programación paralela.

En este capítulo se presenta: (a) una introducción acerca de procesamiento digital de imágenes, (b) descripción del algoritmo de detección de Canny y (c) las implementaciones de la paralelización del algoritmo de Canny utilizando dos patrones arquitectónicos para la programación paralela: *Manager-Workers* y *Communicating Sequential Elements*.

4.1. Representación de una imagen digital

El sentido de la vista en los humanos es muy importante para la percepción de nuestro entorno, aunque sólo se pueda distinguir una cierta banda del espectro electromagnético (EM).

Una imagen digital en general se representa por una función, $E(x, y)$, donde x y y son las coordenadas cartesianas de un plano bidimensional, y el valor de E en cualquier par ordenado (x, y) es la intensidad que hay en ese punto [GW06]. Cuando x, y y E son cantidades discretas y finitas, se llaman imágenes digitales, y a cada punto (x, y) es denominado *elemento de la imagen* mejor conocido como *pixel*[1].

En la actualidad, existen varios algoritmos para el procesamiento de imágenes digitales. Entre los principales objetivos de estos algoritmos están: el mejoramiento, restauración, segmentación, compresión, etc.

En el área médica se necesita que los algoritmos de procesamiento de imágenes médicas sean implementados eficientemente, y además, que tengan la capacidad de poder procesar y analizar las imágenes obtenidas a través de PET (tomografía por emisión de positrones), CT (Tomografía Compu-

[1] *Picture Element*

tarizada de rayos X), MRI (imágenes de resonancia magnética) y microscopía.

Los algoritmos de procesamiento de imágenes más utilizados en las aplicaciones médicas son [Sri05] :

- Ecualización del Histograma
- Filtros de suavización - *Smoothening filter*
- Convolución y Morfología
- Mejoramiento del contraste en el punto de transformación - *Contrast enhancement by point processing*
- Detección de bordes.
- Interpolación de la imagen (Zooming) - *Image interpolation (Zooming)*

Los tiempos de ejecución de los algoritmos dependen del tamaño de las imágenes digitales a procesar. Es decir, para imágenes de resolución 256×256 de 8 bits por *pixel*, el tiempo de ejecución puede no ser significativo y pueden implementarse en computadoras personales, DSPs o FPGAs. Pero para imágenes muy grandes o imágenes de alta resolución, el tiempo empieza a ser considerablemente grande.

Para el caso donde las imágenes digitales son de gran tamaño, o cuando el tratamiento a aplicar es complejo, el realizar el procesamiento exigirá mayor poder de cómputo de lo que una computadora convencional puede proporcionar. Sin embargo, no significa que el trabajo no se pueda realizar. Lo único que implica es que le tomaría una mayor cantidad de tiempo en realizar el cómputo.

Ahora bien, para poder alcanzar los beneficios que proporciona el paradigma de paralelismo, se debe realizar un análisis de las características del problema. Esto para poder determinar si es conveniente invertir tiempo y esfuerzos en paralelizar el problema.

4.2. Imágenes a escala de grises

Una imagen a escala de gris E tiene asignado un valor $E(p) = E(x, y)$ en cada *pixel* $p = (x, y)$, este valor numérico determina el tono de gris u de un específico *pixel*. Los valores de x e y representan las coordenadas de cada *pixel* en la imagen [KA08].

Las dimensiones de la imagen están determinadas de antemano, es decir, se conoce su resolución. El tamaño de la imagen se determina al multiplicar el ancho de la imagen M con la altura N (medidos en *pixeles*). Los valores de M y N determinan la resolución de la imagen. El dominio de x es

$1 \leq x \leq N$, y el de y es $1 \leq y \leq M$.

Los tonos de gris que pueden utilizarse para representar cada pixel están limitados a cierto número de valores de gris. A esto se le denomina cuantización. Cada posible valor de gris puede ser representado por valores enteros positivos consecutivos acotados por un valor máximo. Por lo anterior tenemos:

$$0 \leq u \leq G_{max}$$

El valor estándar de G_{max} para imágenes a escala de grises es de 255.

4.3. Imágenes a color en RGB

La principal diferencia en una imagen digital a escala de gris y una a color es el número de valores que se le asigna a cada *pixel*; a una imagen en escala de grises se le asigna un valor a cada *pixel*, mientras que a una de color en formato RGB se le asignan tres valores[KA08].

Por ejemplo, a una imagen a color C de tres canales se le asocia a cada pixel (x, y) un vector de tres componentes u_1, u_2, u_3:

$$C(x, y) = (u_1(x, y), u_2(x, y), u_3(x, y))^t = (u_1, u_2, u_3)^t$$

Cada vector tiene valores enteros positivos:

$$0 \leq u_1, u_2, u_3 \leq G_{max}$$

La definición de una imagen a color de tres canales se puede expandir a n-canales. A estas imágenes se les denominan imágenes multicanal, multibanda o multiespectrales. Se pueden aplicar los mismos métodos que se aplican a las imágenes a escala de gris a los diferentes canales de la imagen a color [Rus06].

4.3.1. Detección de bordes en imágenes a color

Como cada *pixel* de la imagen es representado por tres valores que corresponden a la cantidad de rojo, verde y azul en una imagen en formato RGB, la luminicencia está dada por la suma ponderada:

$$Y = a_1u_1 + a_2u_2 + a_3u_3$$

donde las a_i son constantes que dependen del espectro característico de los colores.

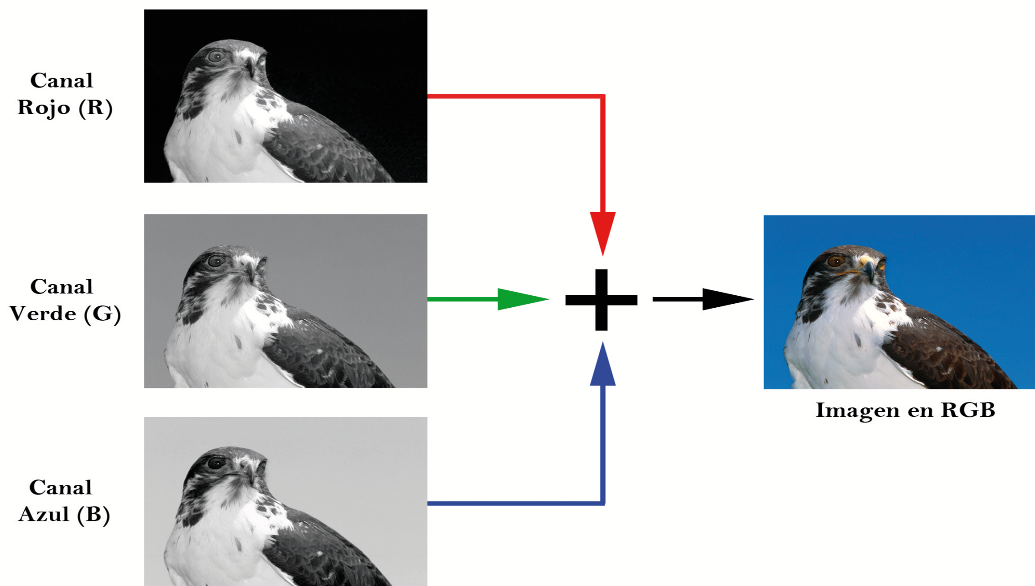

Figura 4.1: Imagen a color RGB.

Existen diversas definiciones para determinar cuando existe un borde en una imagen a color, entre ellas están [Pra01]:

- Existe un borde en la imagen a color si y solo si existe un borde en el campo de luminicencia.
- Si existe un borde en cualquier canal, la imagen a color presenta un borde.
- Esta otra definición, se basa en una suma en los gradientes de cada color:

$$G(x, y) = G_1(x, y) + G_2(x, y) + G_3(x, y)$$

  Y se dice que existe un borde si el valor $G(x, y)$ excede un umbral.

- Existe otra función basada de igual manera en la suma de los gradientes:

$$G(x, y) = \sqrt{[G_1(x, y)]^2 + [G_2(x, y)]^2 + [G_3(x, y)]^2}$$

  Y el criterio para determinar si hay bordes es el mismo que el caso anterior.

4.4. Algoritmo de detección de bordes de Canny

La filosofía de esta técnica es primero calcular las primeras derivadas parciales sobre la imagen suavizada con respecto a x e y. Una vez calculadas las derivadas, se encuentra la magnitud y dirección del borde [KA08].

La variación en el color del pixel en (x,y) es descrita por la siguiente ecuación:

$$\Delta \mathbf{C} = \Delta \mathbf{J}(x, y)$$

En la matriz jacobiana **J** se encuentran las derivadas parciales de cada componente del vector de color. Por ejemplo, para el espacio de color RGB se tiene [KA08]:

$$\mathbf{J} = \begin{pmatrix} R_x & R_y \\ G_x & G_y \\ B_x & B_y \end{pmatrix} = (C_x, C_y)$$

Los subíndices x e y en la matriz anterior representan las derivadas parciales de las funciones.

El algoritmo de detección de bordes de Canny se puede subdividir en tres pasos [VR07]:

1. Calcular las derivadas parciales (Obtención del Gradiente).
2. Calcular las direcciones y magnitudes de los gradientes.
3. Implementar la supresión no-máximo.

En 1986, Canny propuso un método para la detección de bordes basado en tres criterios [Can86]:

- Detección: Este criterio sirve para que no se eliminen bordes importantes y no se agreguen bordes falsos.
- Localización: La distancia entre el borde real y el borde calculado sea mínima.
- Una respuesta: Aglomerar las respuestas múltiples en una sola para generar un único borde.

Los algoritmos para cada uno de los pasos mencionados anteriormente son [VR07]:

Algoritmo: Obtención de Gradiente.

Entrada: imagen I, máscara de convolución H, con media cero y desviación estándar sigma.

Salida: imagen E_m de la magnitud del gradiente
imagen E_o de la orientación del gradiente.

1. Suavizar la imagen I con H mediante un filtro gaussiano para obtener J.
2. Para cada pixel de J, obtener la magnitud y orientación del gradiente: El gradiente de una imagen $f(x,y)$ en el pixel (x,y) se define como un vector bidimencional dado por la ecuación:

$$G[f(x,y)] = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} f(x,y) \\ \frac{\partial}{\partial y} f(x,y \end{bmatrix}$$

siendo un vector perpendicular al borde, donde el vector G apunta en la dirección de variación máxima de f en el punto (x,y) por unidad de distancia, con la magnitud y dirección dadas por:

$$|G| = \sqrt{G_x^2 + G_y^2} = |G_x| + |G_y|\,,$$

$$\phi(x,y) = tan^{-1}\frac{G_y}{G_x}$$

3. Obtener E_m a partir de la magnitud y E_o a partir de la orientación, de acuerdo a las expresiones anteriores.

La máscara de convolución H para aplicar el filtro gaussiano a la imagen original para su suavizamiento es [VR07]:

$$H = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Algoritmo: Supresión No máxima.

Entrada: imagen E_m de la magnitud del gradiente
imagen E_o de la orientación del gradiente

Salida: imagen In

Considerar: cuatro direcciones d_1, d_2, d_3, d_4 identificadas por las direcciones 0°, 45°, 90° y 135° con respecto al eje horizontal.

1. Para cada pixel (i, j)

   a) Encontrar la dirección d_k, que mejor se aproxime a la dirección $E_o(i, j)$, que viene a ser perpendicular al borde.

   b) Si $E_m(i, j)$ es más pequeño que al menos uno de sus dos vecinos en la dirección d_k, al pixel (i, j) de In se le asigna el valor 0, $I(i, j) = 0$ (supresión), de otro modo $In(i, j)$ = $E_m(i, j)$.

2. Devolver In

La imagen resultante al aplicar el algoritmo de detección de bordes de Canny se puede ver en la Figura 4.2.

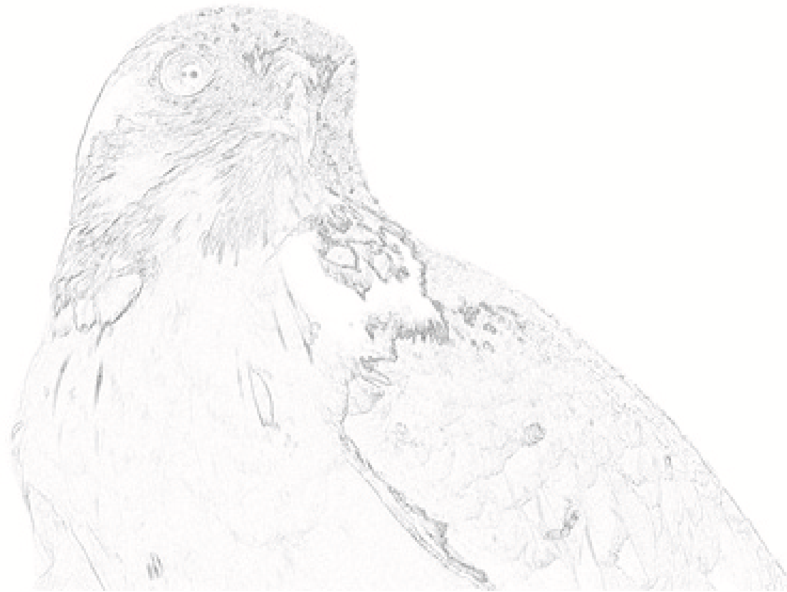

Figura 4.2: Imagen resultante de un filtro de detección de bordes.

4.5. Paralelización del algoritmo de Canny

Para la paralelización de este algoritmo se implementaron dos tipos de patrones arquitectónicos para la programación paralela: *Manager-Workers* (Sección 2.4.1) y *Communicating Sequential Elements* (Sección 2.4.4).

Las dos implementaciones tienen prácticamente el mismo código secuencial, la diferencia es únicamente en el tipo de coordinación. Recuerdese que el objetivo de esta tesis es implementar estos dos patrones para encontrar diferencias significativas entre sus tiempos de ejecución variando ciertos parámetros descritos en la Sección 5.2.

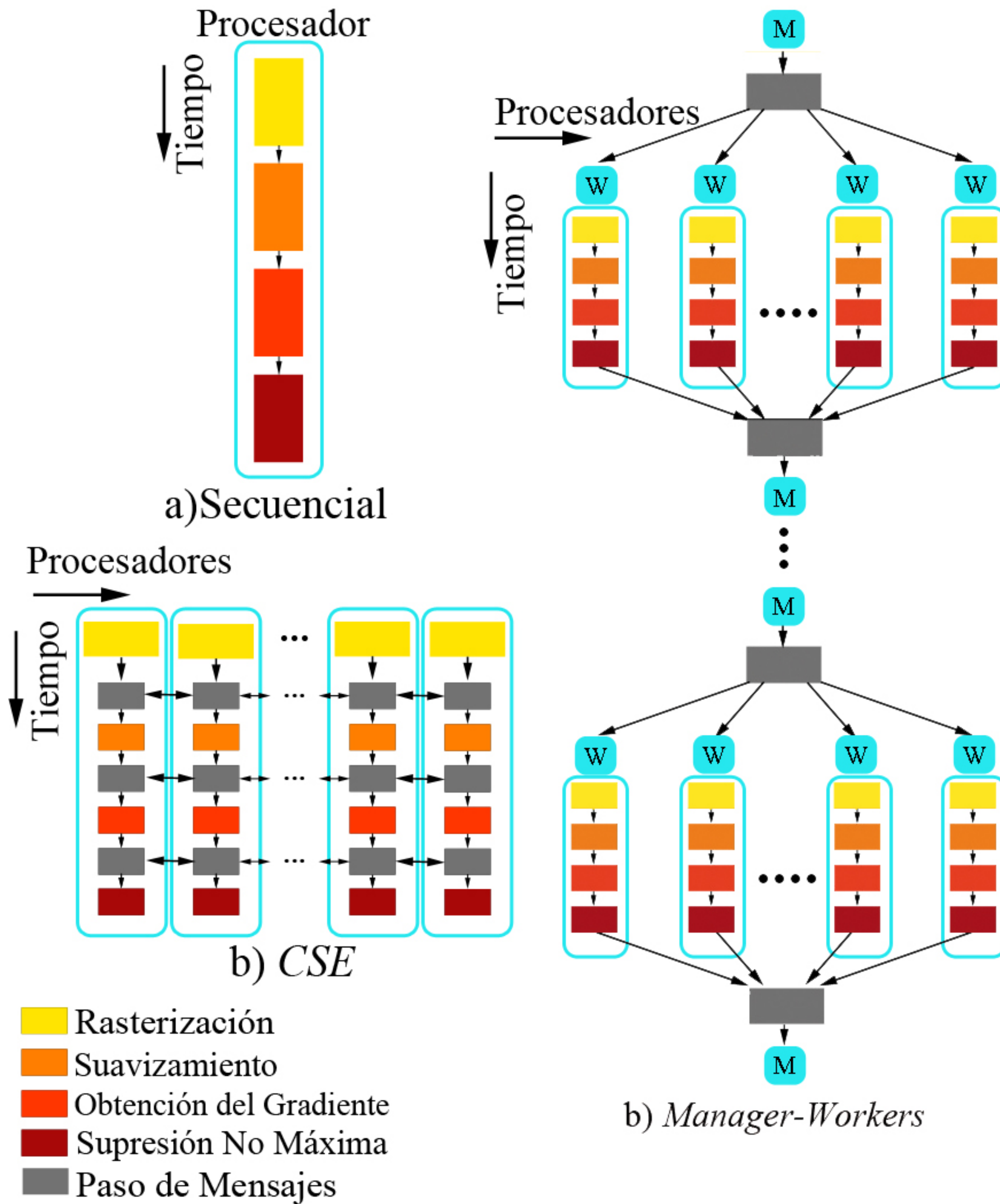

Figura 4.3: a) Procesamiento Secuencial, b) Procesamiento por *Manager-Workers* y c) Procesamiento por *CSE*

En la Figura 4.3 se observa el orden como se realiza el procesamiento de la detección de bordes con el programa secuencial, el programa paralelo utilizando *Manager-Workers* y el programa paralelo utilizando *CSE*. Primero, se divide el problema en tareas identicas: rasterización, suavizamiento,

Obtención del gradiente y supresión no máxima. Estas tareas están codificadas idénticamente para los tres programas.

En el programa secuencial (a) el procesamiento se lleva a cabo tarea por tarea en un determinado orden sobre el conjunto de datos completo.

Para la implementación del patrón arquitectónico para programación paralela *MW* (b), se realizan varias veces el procesamiento completo de las tareas sobre un determinado conjunto de datos indicado previamente por el proceso *Manager*. Al finalizar, cada proceso *Worker* envía el resultado al proceso *Manager* y espera a que se le asigne un nuevo segmento de datos para procesar o la notificación de terminar con su trabajo. La comunicación entre procesos (*Manager-Worker*) es de manera síncrona. El proceso *Manager* recolecta todos los resultados de los procesos *Workers* para que al final se tenga un solo resultaddo final.

Por último se observa la implementación del patrón *CSE* (c) donde se realizan las tareas siguiendo el mismo orden que en el procesamiento secuencial pero con la diferencia que al finalizar cada una de estas tareas se realizan intercambios de información con los procesos que sean necesarios para poder obtener la información necesaria para poder realizar la siguiente tarea. Para el intercambio de información se utiliza comunicación asíncrona. Cada proceso escribe en un respectivo orden y de manera sincronizada su propio resultado.

4.5.1. Aproximación *Manager-Workers*

Una aproximación para resolver un problema expresado en términos de algoritmo y datos es el *Manager-Workers*. Este tipo de patrón es implementado para el caso de estudio que trata esta tesis, que es la detección de bordes en paralelo para imágenes a color de alta definición en formato RGB, utilizando como base el algoritmo de Canny.

En este patrón arquitectónico para programación paralela hay un nodo principal (*Manager*), que divide la imagen en segmentos horizontales independientes; y asigna cada segmento a un determinado nodo *Worker* para su procesamiento. A cada segmento se le agregan cuatro líneas superiores y cuatro inferiores para para que se pueda realizar el procesamiento sin la necesidad de requerir el paso de información con otros nodos.

La forma de segmentación depende del número total de procesos que se generan en la ejecución del programa paralelo. En tiempo de ejecución se divide la imagen lógicamente en segmentos horizontales, formando subimágenes que serán asignadas a un determinado nodo *Worker*.

En la Figura 4.4 se observa el k-ésimo segmento de una imagen, en la cual las líneas *a, b, c y d* corresponden a las líneas *e, f, g y h* del segmento k-1, y las líneas *e, f, g y h* corresponden a las líneas *a, b, c y d* del segmento k+1.

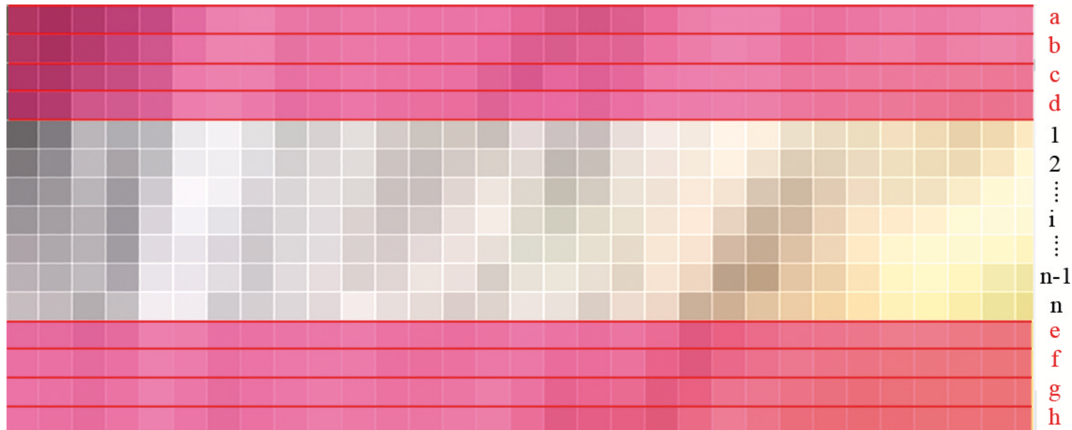

Figura 4.4: Ejemplo del *k*-ésimo segmento y sus líneas replicadas.

El primer segmento no cuenta con un segmento superior para poder replicar líneas, para este caso las filas *a, b, c y d* se convierten en copias de la línea 1. Para el último segmento ocurre el mismo inconveniente, pero por no tener un segmento inferior. Entonces las filas *e, f, g y h* se convierten en copias de la línea n.

Al terminar cada nodo *worker* su respectivo trabajo, envía la información de los bordes detectados al nodo *Manager*, y éste es el encargado de escribir la imagen resultante ensamblando todos los resultados de los demás *Workers*.

La comunicación entre el *Manager* y los *Workers* es de forma síncrona. Esta comunicación síncrona es un tipo de comunicación bloqueante. Esto implica que para que pueda haber comunicación, ambas partes deben estar uno enviando y el otro recibiendo la información. Si un nodo llega antes que el otro a la instrucción enviar o recibir en el código, éste se queda esperando a que el otro nodo llegue a su contraparte, es decir, espera hasta que lleguen al estado enviar-recibir y se pueda realizar la comunicación. Una vez realizada, cada parte continúa con su respectivo procesamiento.

El número de segmentos en que la imagen de entrada se divide (n) para esta coordinación está dada por:

$$n = (\text{Núm. procesadores}) \times (\text{Núm. procesos por procesador})$$

4.5.2. Aproximación *Communicating Sequential Elements*

El otro enfoque que trata esta tesis es el patrón arquitectónico para programación paralela *Communicating Sequential Elements*. En este tipo de coordinación existen varios elementos de procesamiento

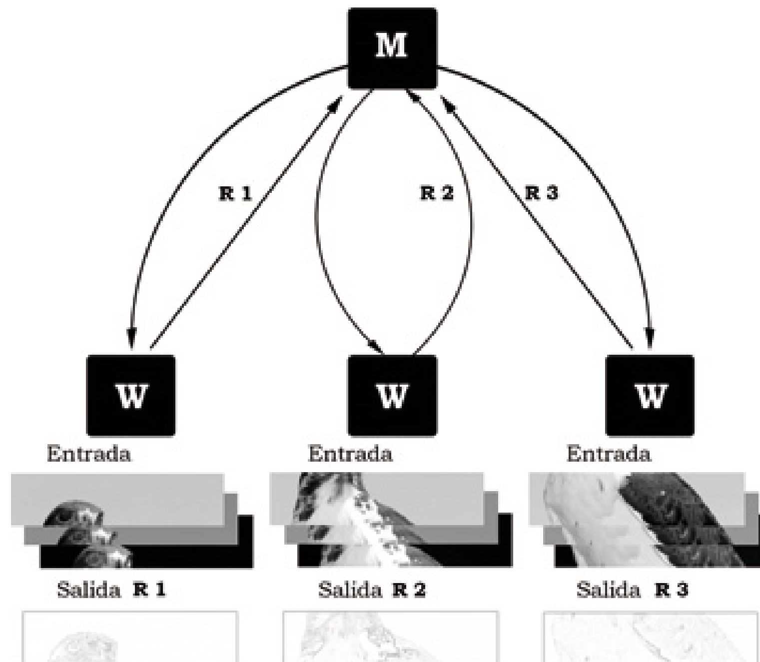

Figura 4.5: Enfoque *Manager-Workers*

que intercambian información que no es posible obtener con su propia información local. Debido a que cada proceso tiene un identificador único (un número entero $x \geq 0$, y consecutivos) es posible tener un cierto orden entre los procesos. Este orden permite la asignación de segmentos contígüos de la imagen a procesos con identificadores tambien consecutivos.

Los elementos se comunican a través de una comunicación asíncrona. Esto permite una mayor flexibilidad en las comunicaciones debido a que no es necesario que el nodo emisor y el receptor se localicen en la misma sección del código. Este tipo de comunicación implementa un *buffer* donde el nodo emisor puede alojar los datos que desea transmitir. El nodo receptor tomará estos datos del *buffer* cuando el así los necesite.

El número de segmentos en que la imagen de entrada se divide (n) para esta coordinación está dada por:

$$n = [(\text{Núm. procesadores}) \times (\text{Núm. procesos por procesador})] - 1$$

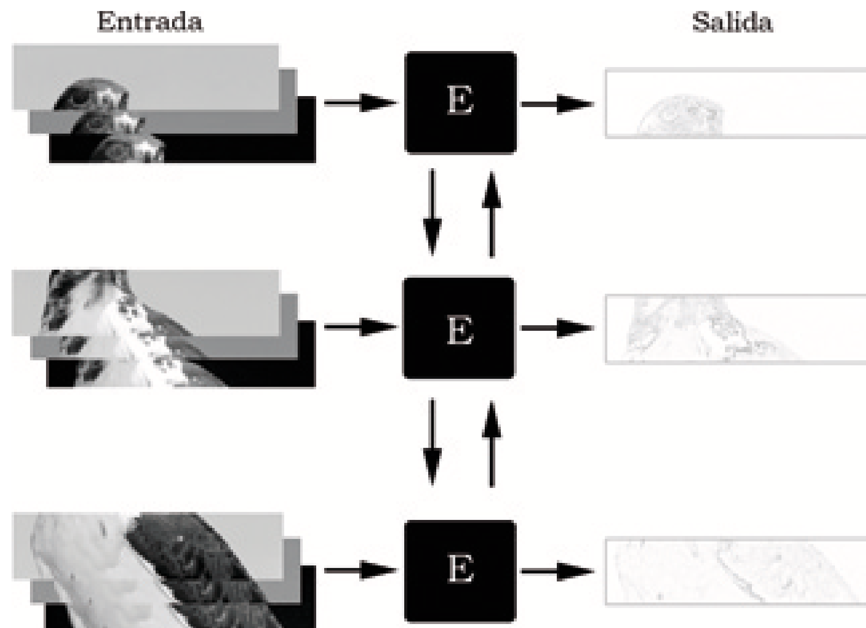

Figura 4.6: Enfoque *Communicating Sequential Elements*

4.6. Comparación de patrones arquitectónicos para programación paralela

Las implementaciones de los patrones arquitectónicos para programación paralela *Manager-Workers* y *Communicating Sequential Elements* son necesarias para poder determinar cuál ofrece mejor desempeño.

Las dos implementaciones tienen el mismo conjunto de tareas: suavizamiento, obtención del gradiente y la orientación, y supresión no máxima.

La principal diferencia entre las dos implementaciones es el tipo de comunicación. Los elementos del patrón *Manager-Workers* utilizan comunicación síncrona mientras que el patrón *Communicating Sequential Elements* asíncrona.

En la Sección 5 se describe a detalle como se lleva a cabo la comparación de estos dos patrones arquitectónicos para programación paralela.

4.7. Resumen

En este capítulo se describen las características principales de una imagen digital a escala de grises y a color en formato RGB. También se describe el algoritmo de detección de bordes de Canny, principalmente en los pasos: cálculo de las derivadas parciales, cálculo de las direcciones y megnitudes de los gradientes y la implementación de la supresión no-máxima.

Además, se describe cómo se puede paralelizar el algoritmo de Canny de detección de bordes utilizando dos patrones arquitectónicos para la programación paralela: *Manager-Workers* y *Communicating Sequential Elements*.

Entre las principales diferencias entre los patrones *CSE* y *MW* está el tipo de comunicación. *CSE* utiliza comunicación asíncrona entre procesos paralelos mientras que *MW* comunicación síncrona.

Además, *CSE* se comunica exclusivamente con los procesos que tengan información que le sea útil, mientras que en el *MW* solo exite comunicación entre el *Manager* y los *Workers*.

Capítulo 5
Evaluación experimental

En este capítulo se describen los experimentos necesarios para para poder realizar la comparación de los dos patrones arquitectónicos para programación paralela en los que se enfoca esta tesis. Estos experimentos varían principalmente en: número de procesadores, número de procesos por procesador, tamaño del problema y tipo de coordinación aplicada para resolver el problema. También se describen los elementos que no cambian a lo largo de los experimentos y se enuncian los supuestos necesarios para las ejecuciones de los programas paralelos.

5.1. Supuestos

Para el desarrollo de los experimentos se cuenta con un conjunto de supuestos que se consideran a lo largo de las ejecuciones de los programas paralelos.

- **Supuesto 1** La red de interconexión que se uiliza para la comunicación entre los nodos del sistema paralelo no varía durante el desarrollo de los experimentos.
- **Supuesto 2** Cada nodo que interviene en el procesamiento es visto por todos los demás nodos del sistema durante la ejecución de los programas paralelos.
- **Supuesto 3** Se mantiene la misma partición de datos para las dos implementaciones paralelas.
- **Supuesto 4** La plataforma de HW paralelo utilizada para la ejecución de los programas paralelos está dedicada exclusivamente para la realización de los experimentos.
- **Supuesto 5** Cada experimento se realiza de forma ordenada e independiente, es decir, en el sistema paralelo sólo se ejecuta un experimento a la vez.
- **Supuesto 6** El código secuencial en las dos implementaciones es el mismo (suavizamiento, cálculo de la magnitud y la orientación, supresión no máxima). La única diferencia es tipo de coordinación que se utiliza para la resolución del problema.

5.2. Variables de los Experimentos

A continuación se describen los tres elementos fundamentales que intervienen en el desarrollo de los experimentos.

Para la realización de los experimentos se utilizan los seis procesadores. En cada experimento se ejecuta un programa paralelo donde se van tomando todas las posibles combinaciones que se pueden obtener al variar el número de procesadores, el número de procesos por procesador, el tipo de coordinación y la imagen de entrada.

5.2.1. Imagen de entrada

Para la realización de los distintos experimentos se utilizan dos imágenes de alta definición a color (RGB) en formato BMP. Las dimensiones de estas dos imágenes se muestran a continuación:

Imagen	Ancho	Largo
Halcon	1600 px	1200 px
Halcon2	5333 px	4000 px

Se utilizan estas dos imágenes con el objetivo de observar el desempeño de los programas paralelos cuando el tamaño del problema escala.

5.2.2. Número de Procesadores

El sistema paralelo que se utiliza es un *cluster* con seis procesadores con su respectiva memoria principal. Cada uno de estos procesadores se encuentran físicamente embebidos en un nodo independiente en el *cluster*, y la comunicación entre ellos es a través de una red Ethernet.

5.2.3. Número de Procesos por procesador

En cada experimento se asigna un número determinado de procesos a cada procesador. Por ejemplo, en el primer experimento con dos procesadores se le asigna a cada procesador un proceso, mientras que para el siguiente experimento con los mismos dos procesadores se le asignan dos procesos, y así consecutivamente. El número máximo de procesos por procesador está determinado, y es de siete procesos por procesador.

En un procesador cada proceso puede ser visto como el número de instancias del programa paralelo que se ejecutan de manera concurrente. Estos procesos utilizan la misma memoria principal.

5.3. Número de Experimentos

El número de experimentos totales está dado por el número de veces que se utiliza cierto número de procesadores (7), número de procesos por procesador (7), número de patrones arquitectónicos implementados (2, MW y CSE), el número de imágenes de entrada (2) y el número de ejecuciones de un mismo experimento (10).

De lo anterios se puede deducir que el número de ejecuciones que se realizaron fue de $7 \times 7 \times 2 \times 2 \times 10 = 1960$ experimentos ejecutados sobre el *cluster*.

El sistema *cluster* donde se realizan los experimentos cuanta con las siguientes características:

`Hardware`

- Un Servidor:
  - Dos procesadores Intel Xeon, 2.6GHz
  - Motherboard SE7501BR2 Intel dual Xeon
  - 1GB de memoria RAM
  - Disco duro SCSI Cheeta, 80GB
- 6 Nodos:
  - Un procesador Intel Pentium 4, 2.6GHz
  - Motherboard Intel Pentium 4 BOSC845GBSRL
  - 512 MB de memoria RAM
  - Disco duro Seagate de 40 GB
- Un Switch 3com Superstack 3 4226T
  - 24 puertos 10/100
  - 2 puertos 10/100/1000

`Software`

- Sistema Operativo:
  - GNU-Linux Debian

- Compilador:
  - GNU project C and C++ compiler (gcc, g++)
- Ambientes de programación paralela:
  - Message Passing Interface (MPI) version 2; MPICH version 0.971

Se realiza un análisis estadístico sobre los tiempos de ejecución para poder conocer principalmente el valor medio de los tiempos de ejecución (μ), así como su desviación estándar (σ) y la varianza (σ^2). La necesidad de estas medidas son debido a que no es posible determinar con exactitud el tiempo de ejecución promedio de cada experimento debido al no determinísmo que existe en los programas paralelos. Esto significa que cada vez que ejecutamos un programa paralelo se obtiene un tiempo de ejecución diferente a todos los anteriores.

La media ($\hat{\mu}$) ofrece un valor que representa a los 10 tiempos de ejecución, pero no a todas las ejecuciones posibles, es decir, $\hat{\mu} \neq \mu$. Ahora, si se realizara un experimento más no sería posible determinar de antemano que tan mayor o menor es este nuevo valor con respecto a la media estimada, ni cuál sería la probabilidad de que el nuevo tiempo de ejecución esté considerablemente cerca de la media, ni mucho cuán confiable es el estimador $\hat{\mu}$.

Para las incertidumbres anteriores se usan los intervalos de confianza [Bar07].

En la Tabla 5.1 se pueden observar los elementos que pueden variar en cada experimento. Se realizan los mismos experimentos para cada tipo de coordinación (MW - CSE), variando el número de procesadores (1 - 7) y el número de procesos por procesador (1 - 7). Estos experimentos se realizan para las dos imágenes de entrada definidas anteriormente.

Coordinación	Imagen	Núm. CPU	Núm. Procs
MW	Halcon.bmp	1	1
CSE	Halcon2.bmp	2	2
		3	3
		4	4
		5	5
		6	6
		7	7

Tabla 5.1: Elementos variables en los experimentos

5.4. Resultados y Análisis

En esta sección se muestra en tablas los tiempos de ejecución promedio de las implementaciones de los dos patrones arquitectónicos para la programación paralela.

Los tiempos presentados son el promedio de los 10 experimentos realizados para cada configuración. Además, se presentan gráficas en 3D para observar con más detalle los cambios en los tiempos de ejecución promedio cuando varía el número de procesadores y el número de procesos por procesador.

También se muestran los resultados obtenidos con los mismos experimentos anteriores, pero ahora escalando el tamaño de la imagen de entrada.

5.4.1. Tiempos de ejecución promedio

Los tiempos de ejecución se miden independientemente al momento de ejecutarse el programa paralelo. Una vez tomados los 10 tiempos de ejecución de cada configuración se realiza un promedio para tener un valor que los represente. Estos tiempos de ejecución promedio son necesarios para realizar el análisis comparativo entre los dos patrones tratados en esta tesis.

Sin embargo, debido a la variación que se observa en los tiempos de ejecución se realiza un análisis estadístico, el cuál consiste en estimar los intervalos de confianza para las medias de los tiempos de ejecución que permita determinar un rango de posibles valores con un 99.73 % de confianza.

Experimentos sobre la imagen halcon.bmp

CPU \ Procesos/CPU	1	2	3	4	5	6	7
1	8.93	5.83	7.13	8.53	9.68	11.32	12.43
2	5.89±0.077	4.44±0.048	3.99±0.022	4.05±0.062	5.44±0.118	6.32±0.128	7.13±0.21
3	3.63±0.008	3.32±0.095	3.13±0.082	3.25±0.121	4.46±0.096	5.37±0.17	6.35±0.234
4	3.06±0.083	2.72±0.061	2.79±0.088	2.74±0.04	4.07±0.119	4.69±0.12	5.89±0.097
5	2.64±0.097	2.52±0.006	2.52±0.114	2.41±0.051	3.84±0.137	4.87±0.261	5.00±0.071
6	2.44±0.001	2.34±0.013	2.17±0.073	2.27±0.101	3.33±0.149	4.80±0.075	5.00±0.071
7	2.15±0.004	2.28±0.108	2.08±0.012	2.22±0.127	3.47±0.068	4.49±0.146	5.22±0.095

Tabla 5.2: Tiempos de ejecución promedio en segundos para el patrón *Manager-Workers* con la imagen halcon.bmp

CPU \ Procesos/CPU	1	2	3	4	5	6	7
1	15.21	13.77	13.48	13.62	13.42	13.56	13.85
2	10.23±0.188	5.79±0.117	4.38±0.114	4.32±0.198	5.06±0.154	6.16±0.113	6.62±0.105
3	5.73±0.164	3.58±0.042	3.07±0.093	3.38±0.143	4.03±0.164	5.34±0.139	6.21±0.193
4	2.61±0.109	3.08±0.1	3.18±0.229	2.71±0.057	3.93±0.128	5.10±0.15	5.74±0.23
5	2.45±0.005	2.59±0.564	2.52±0.674	2.44±0.099	3.99±0.095	4.79±0.081	5.19±0.101
6	2.29±0.005	2.29±0.068	2.19±0.156	2.37±0.166	3.63±0.086	4.61±0.136	4.99±0.118
7	2.64±0.138	2.37±0.138	2.05±0.048	2.31±0.132	3.66±0.077	4.69±0.113	5.57±0.137

Tabla 5.3: Tiempos de ejecución promedio en segundos para el patrón *Communicating Sequential Elements* con la imagen halcon.bmp

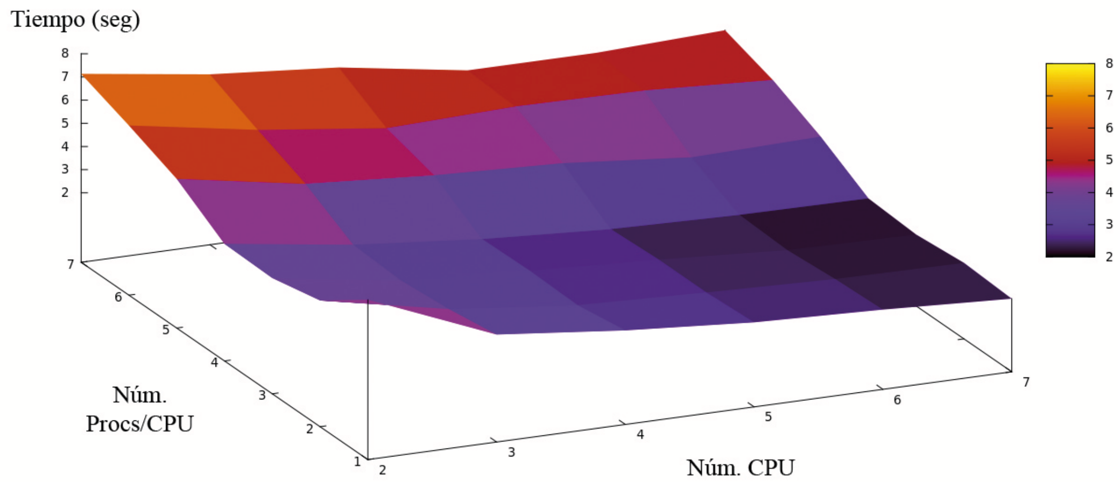

Figura 5.1: Gráfica de los tiempos de ejecución promedio en segundos para el patrón *Manager-Workers* con la imagen halcon.bmp

.

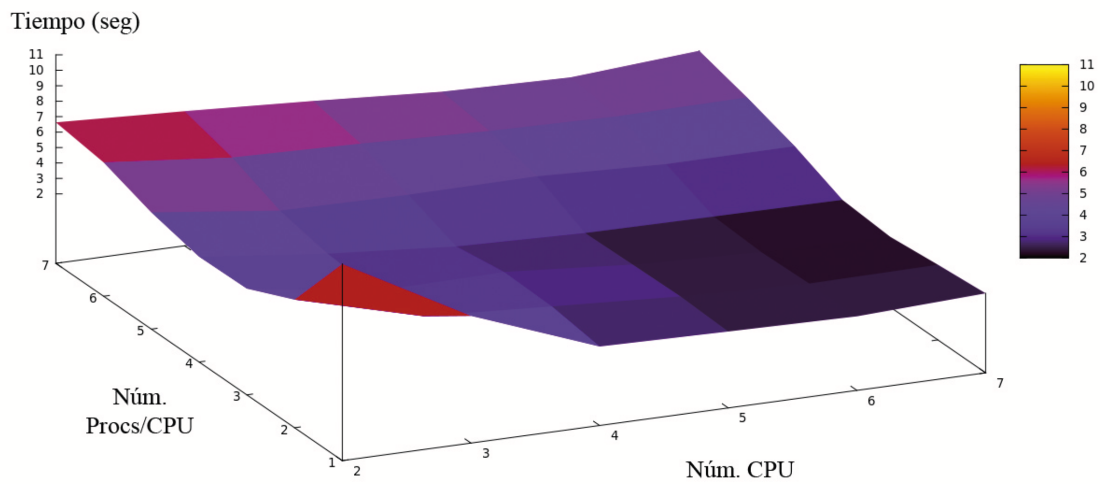

Figura 5.2: Gráfica de los tiempos de ejecución promedio en segundos para el patrón *CSE* con la imagen halcon.bmp

.

Para poder realizar el análisis comparativo entre los desempeños de los programas paralelos se divide la tabla de datos en cuatro regiones, como se observa en la tabla 5.4.

La primera región (A) comprende los tiempos de ejecución promedio de las implementaciones paralelas utilizando 2, 3 y 4 procesadores con 1, 2, 3 y 4 procesos por procesador; la segunda región (B) utilizando 2, 3 y 4 procesadores con 5, 6 y 7 procesos por procesador; la tercera región (C) utilizando 5, 6 y 7 procesadores con 1, 2, 3 y 4 procesos por procesador; la última región (D) utilizando 5, 6 y 7 procesadores con 5, 6 y 7 procesos por procesador.

<table>
<tr><th>Procesos/CPU
CPU</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>2</td><td colspan="4" rowspan="3">A</td><td colspan="3" rowspan="3">B</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td><td colspan="4" rowspan="3">C</td><td colspan="3" rowspan="3">D</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
</table>

Tabla 5.4: División de la tabla de los tiempos promedios de ejecución en cuatro regiones.

En las tablas 5.2 y 5.3 están marcados de color gris los mejores desempeños para cada coordinación. Esta comparación de desempeños se lleva a cabo fijando el número de procesadores, el número de procesos por procesador, el tamaño del trabajo y el procesamiento secuencial. Por lo tanto, la variación en los desempeños estará determinado por el tipo de coordinación que existe entre los componentes de cada patrón, debido a que es lo único que cambia.

Al comparar los desempeños de las dos coordinaciones en la región A, se observa que con la coordinación *Manager-Workers* se obtienen los mejores resultados que por el *Communicating Sequential Elements*. La implementación del patrón *M-W* tiene 9 mejores resultados, mientras que el *CSE* únicamente obtuvo 3. Esto es debido a que existe una mayor cantidad de comunicaciones en el *CSE*, y al ser estas comunicaciones de manera síncrona vuelven a esta coordinación tardada. La mejor coordinación para la región A es *Manager Workers*.

Ahora se comparan los desempeños en la región B. En esta región hay más procesos por procesador que procesadores involucrados en el procesamiento. La coordinación *M-W* presenta sólo 1 resultado favorable mientras que los obtenidos por *CSE* son 8. La mejor coordinación para la región B es *Communicating Sequential Elements*. Esto es debido a que aunque la coordinación *Communicating Sequential Elements* realice una mayor cantidad de comunicaciones, éstas se realizan a través del *software*; lo que hace que sean más rápidas.

Para la región C, 7 de los mejores resultados son obtenidos por la coordinación *M-W*, y 5 por la coordinación *CSE*. En esta región se comienza a ver un emparejamiento en el número de resultados favorables, esto es debido a que los tiempo de procesamiento del trabajo para los componentes de la coordinación *M-W* llegan a tener poca diferencia con respecto al tiempo de trabajo de la coordinación *CSE* más el tiempo requerido por las comunicaciones síncronas de los componentes. Pero aún asi, la coordinación *Manager-Workers* ofrece mejores resultados.

Por último se analizan los desempeños de la región D. En esta región la coordinación *CSE* tiene únicamente 3 mejores resultados, mientras que *M-W* obtiene 6. Entonces la mejor coordinación para este tipo de configuraciones es definitivamente la *Manager-Workers*.

Experimentos sobre la imagen halcon2.bmp

CPU \ Procesos/CPU	1	2	3	4	5	6	7
1	79.23	61.58	76.84	88.91	103.24	114.22	127.79
2	42.17±0.361	40.55±0.126	41.53±0.49	42.07±0.652	60.62±0.34	71.44±0.263	79.38±0.51
3	34.17±0.478	33.12±0.22	34.67±0.488	34.34±0.685	50.29±0.25	62.42±0.358	70.36±0.436
4	34.65±0.379	31.07±0.484	31.61±0.4	33.25±0.66	46.47±0.289	52.89±0.424	65.91±0.483
5	30.44±0.479	28.81±0.387	28.69±0.685	29.26±0.562	43.91±0.294	54.77±0.333	58.34±0.335
6	28.89±0.217	26.95±0.241	27.78±0.383	28.21±0.421	39.88±0.355	53.49±0.365	56.88±0.493
7	25.33±0.299	25.59±0.283	26.24±0.222	26.86±0.378	41.27±0.238	52.21±0.417	60.57±0.268

Tabla 5.5: Tiempos de ejecución promedio en segundos para el patrón *Manager-Workers* con la imagen halcon2.bmp

CPU \ Procesos/CPU	1	2	3	4	5	6	7
1	83.37	80.50	77.23	67.71	78.09	79.95	84.68
2	42.42±0.632	40.62±0.172	41.47±0.776	41.95±0.557	56.26±0.337	68.34±0.411	77.51±0.522
3	33.95±0.251	32.83±0.738	34.05±0.415	33.27±0.688	49.85±0.469	59.90±0.339	69.74±0.342
4	30.47±0.44	29.44±0.26	30.66±0.613	31.25±0.221	45.92±0.42	55.86±0.416	60.14±0.335
5	28.07±0.212	27.05±0.183	28.26±0.419	29.44±0.253	44.40±0.372	55.06±0.395	59.93±0.529
6	31.65±0.362	27.59±0.296	28.55±0.329	28.38±0.222	40.58±0.222	54.21±0.371	57.88±0.371
7	28.25±0.324	24.96±0.15	27.50±0.35	26.44±0.373	43.25±0.384	53.34±0.457	58.25±0.286

Tabla 5.6: Tiempos de ejecución promedio en segundos para el patrón *Communicating Sequential Elements* con la imagen halcon2.bmp

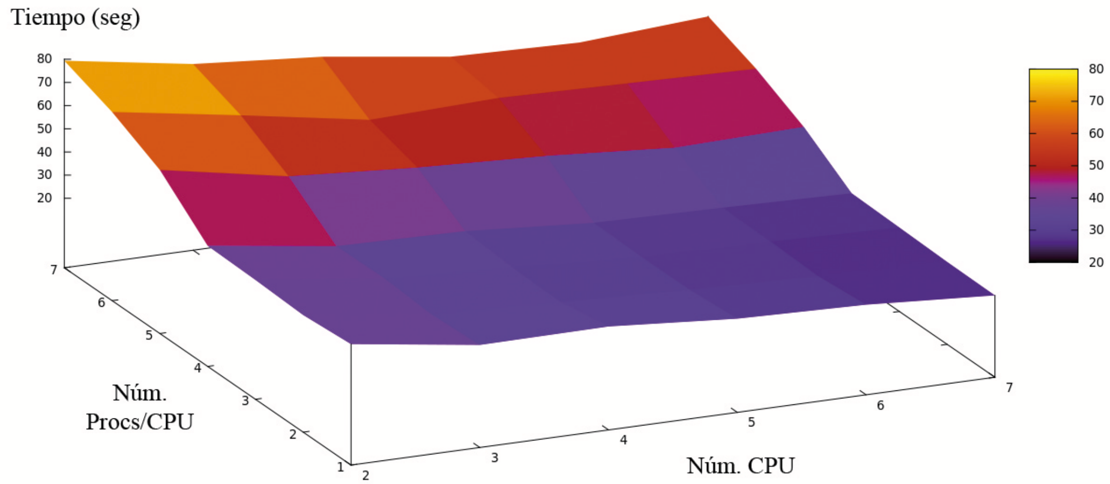

Figura 5.3: Gráfica de los tiempos de ejecución promedio en segundos para el patrón *Manager-Workers* con la imagen halcon2.bmp

.

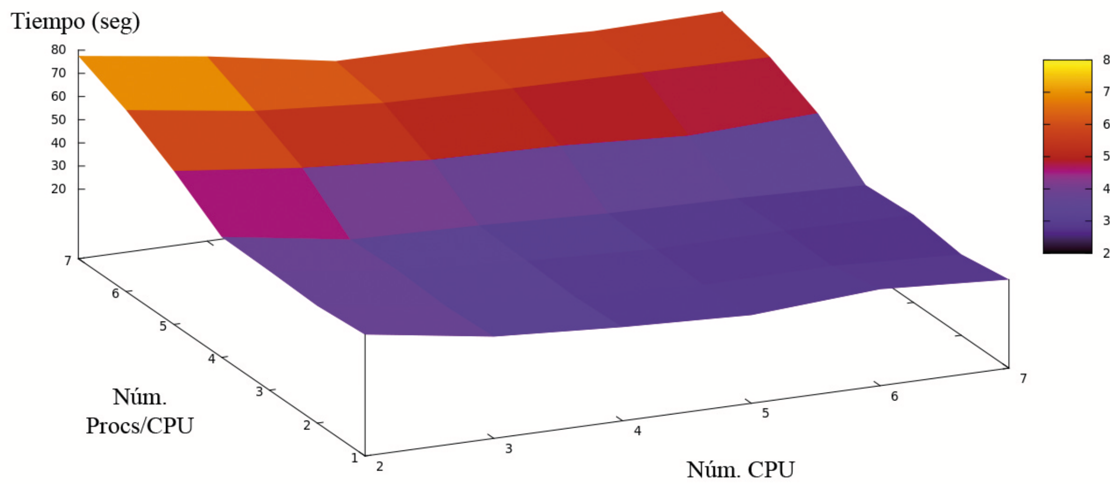

Figura 5.4: Gráfica de los tiempos de ejecución promedio en segundos para el patrón *CSE* con la imagen halcon2.bmp

.

Utilizando la misma división por regiones como se observa en la Tabla 5.4, se analizan los resultados obtenidos de los tiempos de ejecución promedio al escalar el tamaño de la imagen. Se realiza la misma comparación con los desempeños como se observó anteriormente, pero ahora con las tablas 5.5 y 5.6.

Los mejores resultados de la región A son obtenidos por la coordinación *Communicating Sequential Elements*. Con 10 resultados favorables en contra de 2 de la coordinación *Manager Workers*. Por lo tanto, la coordinación más adecuada para las configuraciones de la región A es *Communicating Sequential Elements*.

También se observa esta misma tendencia en la región B. Con 8 resultados favorables de la coordinación *Communicating Sequential Elements* contra 1 de la coordinación *Manager-Workers*. Este comportamiento en estas dos regiones es debido a que se utilizan pocos procesadores y a cada componente de la coordinación *CSE* le corresponde una parte más pequeña del trabajo para procesar debido a que la imagen es dividida dependiendo del número de procesos, mientras que para la coordiación *M-W* se divide la imagen entre el número de procesos menos uno.

En la región C y D, la coordinación que ofrece mejor desempeño es la de *Manager-Workers* con 10 desempeños favorables y 2 en contra para la región C, y para la región D tiene 8 a favor y 1 en contra. Por lo tanto, para 5, 6 y 7 procesadores la coordinación *Manager-Workers* ofrece mejores resultados independientemente del número de procesos implicados en el procesamiento.

5.4.2. Análisis del *Speedup* y Eficiencia

Además del análisis del desempeño se realiza el anális del *Speedup* así como también el de la Eficiencia. Dichas métricas son calculadas conforme a la Sección 2.6.

Los datos mostrados en las Tablas 5.7 y 5.8 son los resultados obtenidos al calcular la métrica *Speedup* para los patrones arquitectónicos para programación paralela *Manager-Workers* y *Communicating Sequential Elements*. Además, se muestra una gráfica del *Speedup* calculado para cada uno de los patrones mencionados.

CPU \ Procs/CPU	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1.51	1.31	1.78	2.10	1.77	1.79	1.74
3	2.46	1.75	2.27	2.62	2.17	2.10	1.95
4	2.91	2.13	2.55	3.11	2.37	2.41	2.11
5	3.38	2.31	2.82	3.53	2.52	2.32	2.48
6	3.65	2.49	3.28	3.75	2.90	2.35	2.48
7	4.15	2.55	3.42	3.84	2.78	2.52	2.38

Tabla 5.7: *Speedup* para el patrón *Manager-Workers* con la imagen halcon.bmp

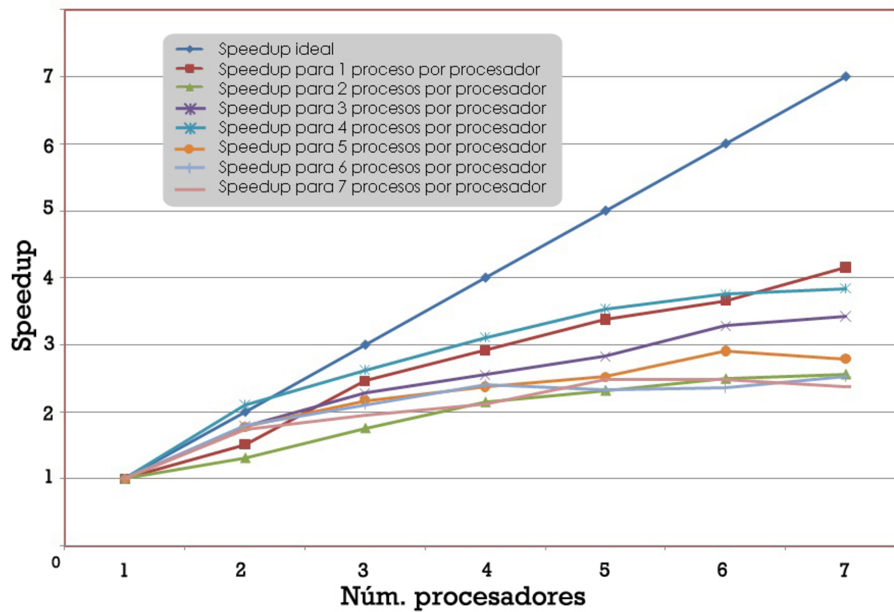

Figura 5.5: Gráfica de *Speedup* del patrón *Manager-Workers* para la imagen halcon.bmp

Al analizar los valores del *Speedup* de la Tabla 5.7, se puede observar que en general los mayores *Speedups* se encuentran cuando se realiza el procesamiento paralelo con cuatro procesos por procea-dor sin importar el número de procesadores. Este resultado es debido a que cuando se le agregan más procesos por procesador los procesadores empiezan a requerir más memoria principal de la disponible y entonces necesitan mover algún proceso poco activo de la memoria principal a la memoria de intercambio (disco duro) lo que hace que el procesamiento paralelo disminuya su desempeño, lo cual repercute directamente en el *Speedup*.

Sin embargo, al observar los valores por número de procesadores, se encuentra que en promedio se obtienen los mejores resultados al utilizar siete procesadores.

Cuando se realiza el cálculo del *Speedup* cuando se realiza el procesamiento paralelo con siete procesadores y un proceso por procesador, se obtiene un *Speedup* de 4.15. Este valor indica cuantas veces es más rápido el programa paralelo con los parámetros mencionados anteriormente que el programa secuencial. Esto es debido a que cada procesador únicamente procesa un único segmento de datos y el único intercambio de información es cuando el proceso *Worker* le envía el resultado de su trabajo al proceso *Manager*, lo que hace que el procesamiento total se realice en un menor tiempo.

CPU \ Procs/CPU	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1.48	2.37	3.07	3.15	2.65	2.20	2.09
3	2.65	3.84	4.39	4.02	3.33	2.53	2.23
4	5.82	4.47	4.23	5.02	3.41	2.65	2.41
5	6.20	5.31	5.34	5.58	3.36	2.83	2.66
6	6.64	6.01	6.15	5.74	3.69	2.94	2.77
7	5.76	5.81	6.57	5.89	3.66	2.89	2.48

Tabla 5.8: *Speedup* para el patrón *Communicating Sequential Elements* con la imagen halcon.bmp

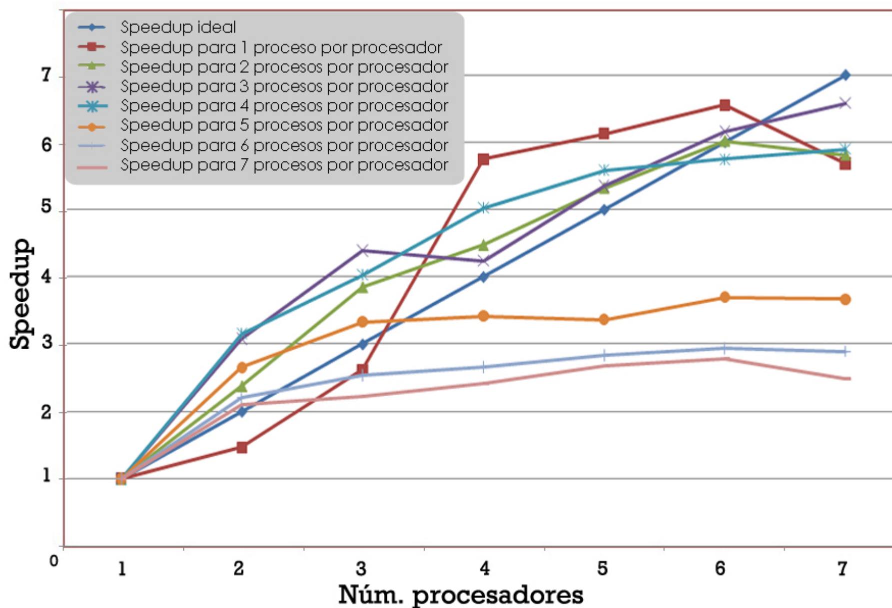

Figura 5.6: Gráfica de *Speedup* del patrón *Communicating Sequential Elements* para la imagen halcon.bmp

En la Figura 5.6 se observa que existen valores por encima del *Speedup* lineal, esto es debido a una combinación entre el tamaño del segmento de datos asignado a cada proceso y al tipo de comunicación que existe entre los procesos. Se utiliza comunicación asíncrona, lo que permite la continuación del programa sin que se tengan que sincronizar las partes receptora y transmisora.

Los mejores resultados de *Speedup* se obtuvieron en los procesamientos con a lo más cuatro procesos por procesador. Utilizando la coordinación *CSE* se realizan más comunicaciones entre procesos, lo que implica que si se le agrega más procesos por procesador se consumirá más tiempo en el paso de mensajes que en el procesamiento, por lo tanto, el *Speedup* disminuye. El mejor *Speedup* se logró con seis procesadores con un proceso por procesador, y es 6.64 veces más rápido que el programa secuencial.

CPU \ Procs/CPU	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1.87	1.51	1.85	2.11	1.7	1.59	1.6
3	2.31	1.85	2.21	2.58	2.05	1.82	1.81
4	2.28	1.98	2.43	2.67	2.22	2.15	1.93
5	2.6	2.13	2.67	3.03	2.35	2.08	2.19
6	2.74	2.28	2.76	3.15	2.58	2.13	2.24
7	3.12	2.4	2.92	3.31	2.5	2.18	2.1

Tabla 5.9: Speedup en segundos para el patrón *Manager-Workers* con la imagen halcon2.bmp

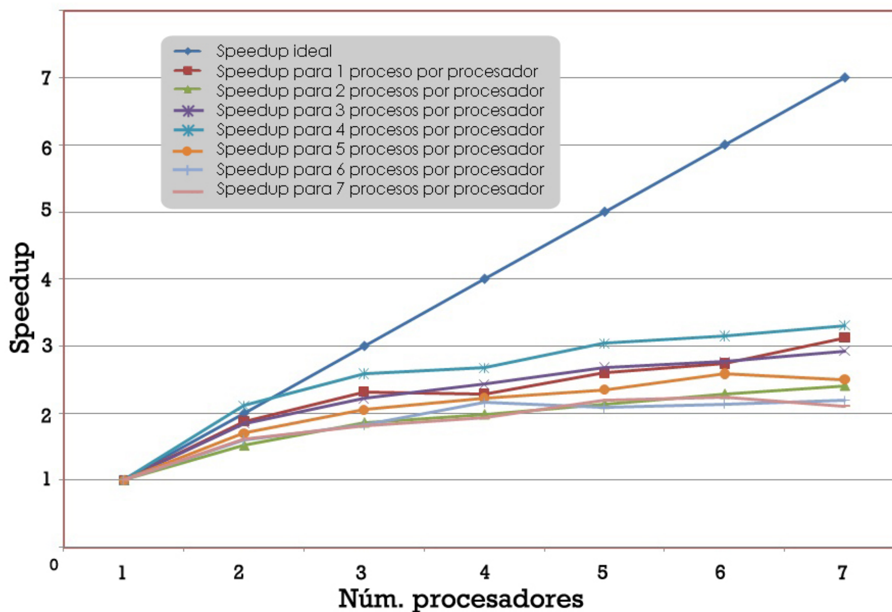

Figura 5.7: Gráfica de *Speedup* del patrón *Manager-Workers* para la imagen halcon2.bmp

Al escalar el tamaño del trabajo, se observa que existe una gran similitud entre los valores de la Tabla 5.7 y la Tabla 5.9. Los mejores resultados obtenidos se localizan cuando en el procesamiento paralelo se utilizan cuatro procesos por procesador. Esto nos indica que cada procesador puede realizar a lo más cuatro procesos concurrentes.

Sin embargo, se mantiene el mismo comportamiento sin importar que el tamaño del problema haya escalado. El máximo *Speedup* se alcanzó realizando el procesamiento con siete procesadores y con cuatro procesos por procesador, y es 3.31 veces más rápido que el programa secuencial.

CPU \ Procs/CPU	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1.96	1.98	1.86	1.61	1.38	1.16	1.09
3	2.45	2.45	2.26	2.03	1.56	1.32	1.21
4	2.73	2.73	2.51	2.12	1.7	1.42	1.4
5	2.96	2.97	2.73	2.29	1.75	1.45	1.41
6	2.63	2.91	2.7	2.38	1.89	1.47	1.46
7	2.95	3.22	2.8	2.56	1.8	1.49	1.45

Tabla 5. … imagen hal-con2.bm

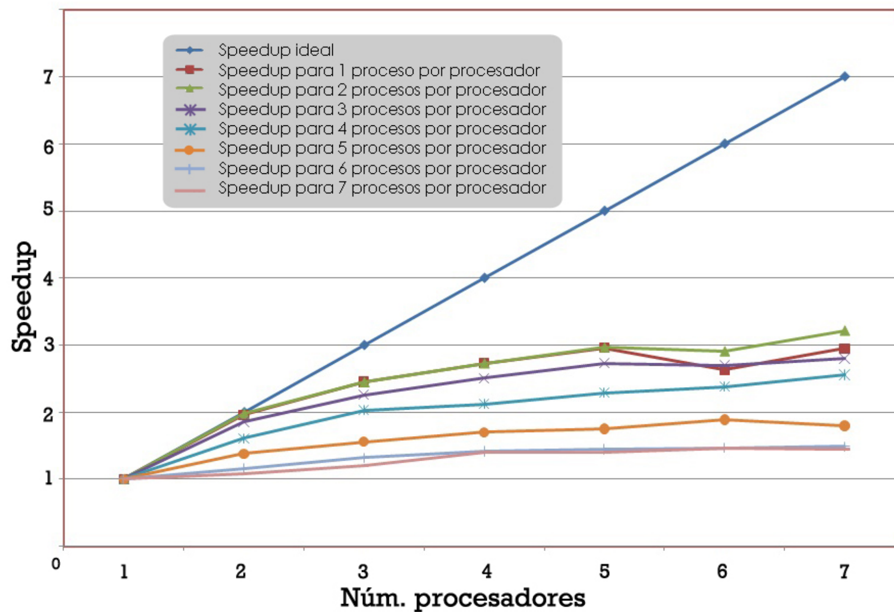

Figura 5.8: Gráfica de *Speedup* del patrón *Communicating Sequential Elements* para la imagen halcon2.bmp

El *Speedup* decreció conforme el tamaño del problema escaló con el patrón arquitectónico para programación paralela. Los mejores resultados de *Speedup* se obtuvieron al utilizar dos procesos por procesador. Sin embargo, se esperaba un mejor resultado que los obtenidos en utilizando la imagen Halcon.bmp.

De los resultados obtenidos hasta ahora se puede observar que cada patrón es mejor que el otro bajo ciertos parámetros. El patrón *MW* es mejor cuando el tamaño del problema escala utilizando cuatro procesos por procesador, pero el patrón *CSE* es mejor cuando el tamaño del problema es menor utilizando dos procesos por procesador. Esto está definido por la cantidad de comunicaciones que se realizan y del tamaño del mensaje que se envia a través de la red de interconección.

El *Speedup* más grande se alcanzó al realizar el procesamiento paralelo con siete procesadores y dos procesos por procesador, y fué 3.22 veces más rápido que el programa secuencial.

Eficiencia:

CPU \ Procs/CPU	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	3.02	2.62	3.57	4.21	3.55	3.58	3.48
3	7.38	5.26	6.83	7.87	6.51	6.32	5.87
4	11.67	8.57	10.22	12.45	9.51	9.65	8.44
5	16.91	11.56	14.14	17.69	12.6	11.62	12.43
6	21.95	14.94	19.71	22.54	17.44	14.15	14.91
7	29.07	17.89	23.99	26.89	19.52	17.64	16.66

Tabla 5.11: Eficiencia para el patrón *Manager-Workers* con la imagen halcon.bmp

En la Tabla 5.11 se observa que el máximo porcentaje de Eficiencia en el programa paralelo es de 29.07 %. Este porcentaje indica que los recursos computacionales no se están utilizando eficientemente. Sin embargo, el resultado depende directamente de la comunicación entre procesos, principalmente del tamaño de la información que se envía y el número de intercambios de información ente los procesos *Workers* con el proceso *Manager*.

CPU \ Procs/CPU	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	2.97	4.75	6.15	6.3	5.3	4.4	4.18
3	7.96	11.53	13.17	12.08	9.99	7.61	6.69
4	23.31	17.88	16.95	20.1	13.65	10.63	9.65
5	31.04	26.58	26.74	27.9	16.81	14.15	13.34
6	39.85	36.07	36.93	34.48	22.18	17.64	16.65
7	40.32	40.67	46.02	41.27	25.66	20.23	17.4

Tabla 5.12: Eficiencia para el patrón *Communicating Sequential Elements* con la imagen halcon.bmp

En la Tabla 5.12 se observa un grupo de configuraciones que en su mayoría ofrecen una mayor Eficiencia que la Eficiencia máxima de la Tabla 5.11. Este grupo está determinado por los procesamientos en paralelo con 5, 6 y 7 procesadores con 1, 2 , 3 y 4 procesos por procesador. Esto significa que los recursos son utilizados de una mejor manera en el patrón *CSE* que en el patrón *MW*.

En la implementación del patrón arquitectónico para programación paralela *CSE* se obtiene una Eficiencia máxima de 46.02 %.

CPU \ Procs/CPU	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	3.75	3.03	3.7	4.22	3.4	3.19	3.21
3	6.95	5.57	6.64	7.76	6.15	5.48	5.44
4	9.14	7.92	9.72	10.69	8.88	8.63	7.75
5	13.01	10.68	13.39	15.19	11.75	10.42	10.95
6	16.45	13.7	16.59	18.91	15.53	12.81	13.47
7	21.89	16.84	20.49	23.17	17.51	15.31	14.76

Tabla 5.13: Eficiencia para el patrón *Manager-Workers* con la imagen halcon2.bmp

Al escalar el tamaño del trabajo se puede observar que bajo la implementación del patrón arquitectónico para programación paralela *MW* se obtuvo una máxima Eficiencia de 23.17 % al realizar el procesamiento paralelo con siete procesadores y cuatro procesos por procesador.

CPU \ Procs/CPU	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	3.92	3.96	3.72	3.22	2.76	2.32	2.18
3	7.35	7.35	6.78	6.09	4.68	3.96	3.63
4	10.92	10.92	10.04	8.48	6.8	5.68	5.6
5	14.8	14.85	13.65	11.45	8.75	7.25	7.05
6	15.78	17.46	16.2	14.28	11.34	8.82	8.76
7	20.65	22.54	19.6	17.92	12.6	10.43	10.15

Tabla 5.14: Eficiencia para el patrón *Communicating Sequential Elements* con la imagen halcon2.bmp

Para la implementación de patrón arquitectónico para programación paralela *CSE* se obtuvo una Eficiencia máxima de 22.54 % con siete procesadores con dos procesos por procesador.

La diferencia en las Eficiencias de ambas implementaciones es menor del 1 %. Lo que se puede decir es que ambas implementaciones no utilizan a los procesadores de una manera eficiente, esto es debido a que la mayoría del tiempo de procesamiento es utilizado para el paso de mensajes.

Para mejorar el *Speedup* y la Eficiencia, es necesario disminuir el número y el tamaño de los mensajes que se envían entre los procesos. Esto con el fin de que los procesadores estén mayormente trabajando sobre el problema y no en ocio esperando por información de otros procesos.

5.5. Resumen

En este capítulo se muestran los resultados de los experimentos realizados que sirven para el análisis comparativo del desempeño de los patrones arquitectónicos para programación paralela. Estos resultados se obtuvieron de las diferentes configuraciones de los parámetros (variando en número de procesadores, número de procesos por procesador, tamaño del trabajo y tipo de coordinación).

Además, se realizan tres diferentes tipos de análisis. El primer análisis es el que se encarga de comparar los desempeños de los patrones arquitectónicos; el cual ayuda determinar cuál patrón arquitectónico es el mejor para el caso de estudio y porqué. El segundo análisis es del *Speedup* y sirve para determinar cuantas veces es más rápido el programa paralelo que el programa secuencial. Por último, se encuentra el análisis de la Eficiencia, el cual estima qué tan bien se utilizan los procesadores en la solución del problema.

Los últimos dos análisis ayudan a determinar cómo se podrían optimizar los programas paralelos. Principalmente se observa que disminuyendo las comunicaciones entre los procesos y el tamaño de la información se puede mejorar el desemeño, el *Speedup* y la Eficiencia.

Capítulo 6
Conclusiones y Trabajo futuro

Este capítulo se describen a) las conclusiones generales y específicas de esta tesis. Para esto es necesario retomar la hipótesis planteada en el Capítulo 1, para poder contrastarla con los resultados obtenidos. Además, se determinan algunas condiciones en los experimentos que no se estudian en esta tesis que forman parte del trabajo futuro.

6.1. Conclusiones generales

La hipótesis dice:

Dado un problema expresado en términos de algoritmo y datos, un conjunto de patrones arquitectónicos para programación paralela, un número variable de procesadores y procesos por procesador, ¿Es posible determinar con una análisis comparativo del desempeño de los patrones cuál ofrece un mayor desempeño sobre el otro?.

Para contestar esta pregunta primero hay que observar los resultados generales del análisis comparativo:

halcon.bmp		halcon2.bmp	
MW	CSE	MW	CSE
19	23	24	18

Tabla 6.1: Número de desempeños ganadores globalmente.

Observando los resultados de la tabla 6.1 no se puede determinar un patrón ganador global y que sea contundente.

Sin embargo, se puede realizar un análisis local de los resultados ganadores. En cada región en las que se dividen las tablas de los tiempos de ejecución promedios se puede observar que si se puede

determinar un patrón arquitectónico para la programación paralela ganador.

<table>
<tr><th></th><th colspan="2">A</th><th colspan="2">B</th><th colspan="2">C</th><th colspan="2">D</th></tr>
<tr><th></th><th>MW</th><th>CSE</th><th>MW</th><th>CSE</th><th>MW</th><th>CSE</th><th>MW</th><th>CSE</th></tr>
<tr><td>halcon.bmp</td><td>9</td><td>3</td><td>1</td><td>8</td><td>7</td><td>5</td><td>6</td><td>3</td></tr>
<tr><td>halcon2.bmp</td><td>2</td><td>10</td><td>1</td><td>8</td><td>7</td><td>5</td><td>8</td><td>1</td></tr>
</table>

Tabla 6.2: Número de desempeños ganadores por región.

En la tabla 6.2 se observan el número de resultados ganadores por cada región. A partir de esta nueva tabla se contruyen dos nuevas tablas donde se observa la coordinación ganadora por región.

<table>
<tr><th>Procs/CPU
CPU</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>2</td><td colspan="4" rowspan="3">MW</td><td colspan="3" rowspan="3">CSE</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td><td colspan="4" rowspan="3">MW</td><td colspan="3" rowspan="3">MW</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
</table>

Tabla 6.3: Coordinaciones ganadoras por región con imagen de entrada halcon.bmp.

Al ver esta tabla uno no podría afirmar que el patrón arquitectónico para programación paralela *Manager-Workers* ofrece mejor desempeño que el *Communicating Sequential Elements*. No existe una conclusión general para este caso. Lo que se pudiera inferir es que el patrón *M-W* ofrece mejor desempeño que el *CSE* para las regiones A, C y D con la imagen de entrada halcon.bmp.

Al aumentar el tamaño del trabajo, se obtuvieron los siguientes resultados:

Se puede observar que cuando el tamaño del trabajo escala se mantienen las mismas coordinaciones ganadoras en las regiones B, C y D. Sin embargo, para la región A se observa que el tamaño del trabajo es más importante; cuando la carga de trabajo va en aumento la coordinación que ofrece mejores resultados es la *Communicating Sequential Elements*.

Con la ayuda del análisis del *Speedup* y la Eficiencia, se determina que los programas paralelos implementados pueden obtener un mejor desempeño. Esto es debido a que en el procesamiento utilizan mucho tiempo para el intercambio de información entre procesos en vez de la resolución del problema

<table>
<tr><th>CPU \ Procs/CPU</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>2</td><td colspan="4" rowspan="3">CSE</td><td colspan="3" rowspan="3">CSE</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td><td colspan="4" rowspan="3">MW</td><td colspan="3" rowspan="3">MW</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
</table>

Tabla 6.4: Coordinaciones ganadoras por región con imagen de entrada halcon2.bmp.

en sí. Por lo tanto, se deben de reducir las comunicaciones y el tamaño de los mensajes para poder obtener un mayor desempeño.

6.2. Trabajo futuro

Este trabajo es la primera aproximación que trata el problema de elección entre al menos dos tipos de patrones arquitectónicos para programación paralela. Existen diversas variaciones que se podrían agregar al trabajo, por ejemplo:

- Cambiar la arquitectura de HW y/o SW
    - Utilizar un sistema de memoria compartida en vez de un sistema de memoria distribuida.
    - Utilizar otro lenguaje de programación u otra biblioteca para el paso de mensajes.
    - Utilizar una diferente red de interconexión alámbrica.
- Aumentar el número de procesadores.
- Aumentar la cantidad de procesos por procesador.
- Escalar aún más el tamaño del problema.
- Cambiar el particionamiento de las tareas.
- Cambiar el tipo de aplicación.

Además, se puede agregar otros tipos de análisis como:

- Análisis de la comunicación entre procesos.
- Análisis de la concurrencia de los procesos.

Bibliografía

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 2000.

[Bar07] Michael Baron. *Probability and Statistics for Computer Scientists*. Chapman & Hall/-CRC, 2007.

[BJD00] Plateu B. Blazewicz J., Ecker K. and Trystram D. *Handbook on Parallel and Distributed Processing*. Sprinter-Verlag, Heidelberg, Berlin, GERMANY, 2000.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[Can86] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, November 1986.

[Car01] Nicholas Carter. *Computer Architecture*. McGraw-Hill, Inc., 2001.

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Comput. Surv.*, 21(3):323–357, 1989.

[DC98] A. Downton and D. Crookes. Parallel architectures for image processing. *Electronics and Communication Engineering Journal*, 10(3):139–151, Jun 1998.

[FLD00] Ghassan Fadlallah, Michel Lavoie, and Louis-A. Dessaint. Parallel computing environments and methods. *Parallel Computing in Electrical Engineering, International Conferenceon*, 0:2, 2000.

[Fly66] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec. 1966.

[FP92] T. L. Freeman and C. Phillips. *Parallel numerical algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[GKKG03] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, January 2003.

[GL95] W. W. Gropp and E. L. Lusk. A taxonomy of programming models for symmetric multiprocessors and smp clusters. In *In Proceedings of 1995 Programming Models for Massively Parallel Computers*, pages 2–7, 1995.

[GW06] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[HX98] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing: Technology,Architecture,Programming*. McGraw-Hill, Inc., New York, NY, USA, 1998.

[KA08] Andreas Koschan and Mongi A. Abidi. *Digital Color Image Processing*. Wiley-Interscience, New York, NY, USA, 2008.

[KF90] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, 1990.

[OA09] Jorge Luis Ortega Arjona. *Architectural Patterns for Parallel Programming: Models for Performance Estimation*. VDM Verlag, 2009.

[OA10] Jorge Luis Ortega Arjona. *Patterns for Parallel Software Design*. John Wiley and Sons, 2010.

[Pan96] Cherri M. Pancake. Is parallelism for you? *IEEE Comput. Sci. Eng.*, 3(2):18–37, 1996.

[Par99] Behrooz Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[Pra01] William K. Pratt. *Digital Image Processing: PIKS Inside*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[Roo99] Seyed H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[Rus06] John C. Russ. *The Image Processing Handbook, Fifth Edition (Image Processing Handbook)*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

[Sha95] Mary Shaw. *Patterns for software architectures*, pages 453–462. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[SO98] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[Sri05] V. Srinivasan, N. et Vaidehi. Application of cluster computing in medical image processing. *Broadband Networks, 2005. BroadNets 2005. 2nd International Conference*, 2005.

[VR07] Jorge Valverde Rebaza. Detección de bordes mediante el algoritmo de canny. *Escuela Académico Profesional de Informática, Universidad Nacional de Trujillo*, 2007.

[WA04] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition edition, 2004.

Apéndice A
Intervalos de Confianza

Definición 1.-Un intervalo $[a, b]$ es un $(1-\alpha)100\,\%$ intervalo de confianza para el parámetro θ si este contiene el parámetro con probabilidad $(1-\alpha)$,

$$\boldsymbol{P} a \leq \theta \leq b = 1 - \alpha$$

La probabilidad de cobertura $(1-\alpha)$ es también llamada nivel de confianza.

A.1. Construyendo un intervalo de confianza

Dado una muestra de datos (10 tiempos de ejecución de un determinado experimento) y un nivel de confianza deseado $(1-\alpha)$, se puede construir un intevalo de confianza $[a,\ b]$ que satisfaga la condición de cobertura definición 1

$$\boldsymbol{P} a \leq \theta \leq b = 1 - \alpha.$$

Para llegar a contruir dicho intervalo primero se estima el parámetro μ. Se supone que exsiste un estimador (Assume there is an unbiased estimator) $\hat{\mu}$ que tiene una distribución Normal. Al estandarizarlo se obtiene una variable estándar Normal

$$Z = \frac{\hat{\mu} - \mathbf{E}(\hat{\mu})}{\text{Std}(\hat{\mu})} = \frac{\hat{\mu} - \mu}{\text{Std}(\hat{\mu})}.$$

Esta variable cae entre los cuartiles $q_{\alpha/2}$ y $q_{1-\alpha/2}$, cuya notación está dada por

$$-z_{\alpha/2} - q_{\alpha/2}$$
$$z_{\alpha/2} - q_{1-\alpha/2}$$

con la probabilidad $(1-\alpha)$, como se observa en la figura A.1.

Entonces,

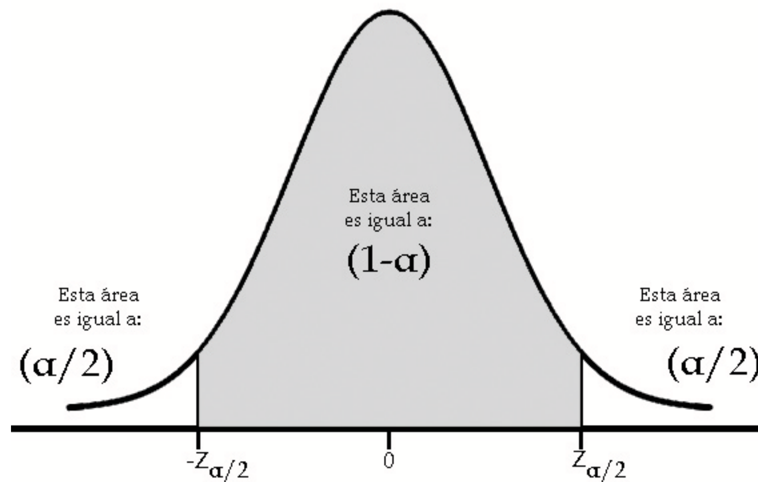

Figura A.1: Stándard Normal quantiles $\pm z_{\alpha/2}$ y particion del área bajo la curva de densidad.

$$\boldsymbol{P}\{-z_{\alpha/2} \leq \tfrac{\hat{\mu}-\mu}{\text{Std}(\hat{\mu})} \leq z_{\alpha/2}\} = 1-\alpha.$$

Resolviendo la desigualdad se tiene

$$\boldsymbol{P}\{\hat{\mu} - z_{\alpha/2}\text{Std}(\hat{\mu}) \leq \mu \leq \hat{\mu} + z_{\alpha/2}\text{Std}(\hat{\mu})\} = 1-\alpha.$$

Por lo tanto ya se tienen los dos valores del intervalo

$$a = \hat{\mu} - z_{\alpha/2}\text{Std}(\hat{\mu})$$
$$b = \hat{\mu} + z_{\alpha/2}\text{Std}(\hat{\mu})$$

tal y como

$$\boldsymbol{P} a \leq \theta \leq b = 1-\alpha.$$

En las ecuaciones anteriorers, $\hat{\mu}$ es el centro del intervalo, y $z_{\alpha/2}\text{Std}(\hat{\mu})$ es el margen de error.

A.2. Intervalo de confianza para la media de la población

Se construye el intervalo de confianza para la media de la población

$$\theta = \mu = \mathbf{E}(X).$$

Se define el estimador,

$$\hat{\theta} = \bar{X} = \tfrac{1}{n}\sum_{i=1}^{n} X_i$$

Suponiendo que los tiempos de ejecución $\boldsymbol{X} = X_1, X_2, ..., X_n$ tienen una distribución normal entonces la regla 9.3 puede ser aplicada

$$E(\bar{X}) = \mu$$
$$Std(\bar{X}) = \tfrac{\sigma}{\sqrt{n}}$$

Entonces la regla 9.3 se reduce al siguiente $(1-\alpha)100\,\%$ intervalo de confianza para μ.

Intervalo de confianza para la media; con σ conocida [Bar07]

$$\bar{X} \pm z_{\alpha/2}\tfrac{\sigma}{\sqrt{n}}$$

Apéndice B

Paso de mensajes con MPI

B.1. Ejemplo:

Objetivo: Este ejemplo muestra la diferencia entre comunicación bloqueante (síncrona) y no bloqueante (asíncrona).

Descripción: Cada tarea transfiere un vector de números aleatorios (senddbuff) a la siguiente tarea (taskid+1). La última tarea transfiere el vector a la tarea 0. Consecuentemente, cada tarea recibe un vector de la tarea precedente y lo almacena en recvbuff.

Autor: Carol Gauthier
Centre de Calcul scientifique
Universite de Sherbrooke

```
                sendbuff   recvbuff       sendbuff   recvbuff
                ########   ########       ########   ########
            0   #  AA  #   #      #       #  AA  #   #  DD  #
                #      #   #      #       #      #   #      #
                ########   ########       ########   ########
        T       #      #   #      #       #      #   #      #
            1   #  BB  #   #      #       #  BB  #   #  AA  #
        a       #      #   #      #       #      #   #      #
                ########   ########       ########   ########
        r       #      #   #      #       #      #   #      #
            2   #  CC  #   #      #       #  CC  #   #  BB  #
        e       #      #   #      #       #      #   #      #
                ########   ########       ########   ########
        a       #      #   #      #       #      #   #      #
            3   #  DD  #   #      #       #  DD  #   #  CC  #
                ########   ########       ########   ########
                          ANTES                   DESPUES
```

B.2. Paso de mensajes bloqueante

Este código fuente muestra que MPI_Send y MPI_Recv no son las funciones más eficientes para desempeñar este trabajo. Debido a que trabaja de modo bloqueante. Para poder recibir su vector, cada tarea debe de realizar un MPI_Recv correspondiente al MPI_Recv del remitente, y entonces esperar que la recepción sea completada antes de poder enviar su sendbuff a la siguiente tarea. Para evitar un *deadlock* [1], una de las tareas debe de iniciar la comunicación realizando un MPI_Recv después su MPI_Send. En este ejemplo, la última tarea inicia el proceso en cascada donde cada tarea consecutiva se encuentra esperando por la recepción completa de la tarea preceddente antes de empezar a enviar su sendbuff a la siguiente tarea. El proceso completo termina cuando la última tarea ha terminado su recepción.

Antes de la comunicación, la tarea 0 calcula la suma de todos los vectores a enviar de cada tarea, y las imprime. Similarmente despues de todas las comunicaciones, la tarea 0 calcula la suma de todos los vectores recibidos para cada tarea y las imprime junto con los tiempos de comunicación.

El tamaño del vector (buffsize) está dado por un argumento al ejecutar el programa.

B.2.1. Código fuente en C/MPI utilizando MPI_Send y MPI_Recv

```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "math.h"
#include "mpi.h"

int main(int argc,char** argv){
    int           taskid, ntasks;
    MPI_Status    status;
    MPI_Request   req[1024];
    int           ierr,i,j,itask,recvtaskid;
    int           buffsize;
    double        *sendbuff,*recvbuff;
    double        sendbuffsum,recvbuffsum;
    double        sendbuffsums[1024],recvbuffsums[1024];
    double        inittime,totaltime,recvtime,recvtimes[1024];

    /* Comienzo del programa, despues declaracion de variables.        */
    MPI_Init(&argc, &argv);
    /* Obtiene el numero de tareas de MPI y el identificador de esta tarea.      */
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
    /* Obtiene el valor del buffsize desde los argumentos del programa.               */
    buffsize=atoi(argv[1]);
```

[1]Un conjunto de procesos está en estado de *deadlock* cuando cada proceso del mismo esta esperando un evento que sólo puede ser causado por otro proceso que pertenece a ese conjunto

```
/* Imprime la descripcion del ejemplo.                    */
if ( taskid == 0 ){
  printf("\n\n\n");
  printf("##################################################\n\n");
  printf("␣Example␣1␣\n\n");
  printf("␣Comunicacion␣Punto␣a␣Punto:␣MPI_Send␣MPI_Recv␣\n\n");
  printf("␣Tamaño␣del␣vector:␣%d\n",buffsize);
  printf("␣Nummero␣de␣tareas:␣%d\n\n",ntasks);
  printf("##################################################\n\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-->␣ANTES␣DE␣LAS␣COMUNICACIONES␣<--\n\n");
}

/* Alojamiento de la memoria                              */
sendbuff=(double *)malloc(sizeof(double)*buffsize);
recvbuff=(double *)malloc(sizeof(double)*buffsize);

/* Inicializacion de los Vectors y/o matrices.            */
srand((unsigned)time( NULL ) + taskid);
for(i=0;i<buffsize;i++){
  sendbuff[i]=(double)rand()/RAND_MAX;
}

/* Imprime antes de las comunicaciones.                   */
sendbuffsum=0.0;
for(i=0;i<buffsize;i++){
  sendbuffsum += sendbuff[i];
}
ierr=MPI_Gather(&sendbuffsum,1,MPI_DOUBLE,
                sendbuffsums,1, MPI_DOUBLE,
                0,MPI_COMM_WORLD);

if(taskid==0){
  for(itask=0;itask<ntasks;itask++){
    recvtaskid=itask+1;
    if(itask==(ntasks-1))recvtaskid=0;
    printf("Tarea␣%d␣:␣La␣suma␣de␣los␣vectores␣enviados␣a␣%d␣=␣%e\n",
           itask,recvtaskid,sendbuffsums[itask]);
  }
}

/* Comunicacion.                                          */
inittime = MPI_Wtime();

if ( taskid == 0 ){
  ierr=MPI_Recv(recvbuff,buffsize,MPI_DOUBLE,
          ntasks-1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  recvtime = MPI_Wtime();
  ierr=MPI_Send(sendbuff,buffsize,MPI_DOUBLE,
          taskid+1,0,MPI_COMM_WORLD);
}
else if( taskid == ntasks-1 ){
  ierr=MPI_Send(sendbuff,buffsize,MPI_DOUBLE,
          0,0,MPI_COMM_WORLD);
  ierr=MPI_Recv(recvbuff,buffsize,MPI_DOUBLE,
          taskid-1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  recvtime = MPI_Wtime();
}
else{
  ierr=MPI_Recv(recvbuff,buffsize,MPI_DOUBLE,
```

```
            taskid -1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
    recvtime = MPI_Wtime();
    ierr=MPI_Send(sendbuff,buffsize,MPI_DOUBLE,
            taskid+1,0,MPI_COMM_WORLD);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  totaltime = MPI_Wtime() - inittime;
  /* Imprime despues de las comunicaciones.                        */
  recvbuffsum=0.0;
  for(i=0;i<buffsize;i++){
    recvbuffsum += recvbuff[i];
  }
  ierr=MPI_Gather(&recvbuffsum,1,MPI_DOUBLE,
                recvbuffsums,1, MPI_DOUBLE,
                0,MPI_COMM_WORLD);
  ierr=MPI_Gather(&recvtime,1,MPI_DOUBLE,
                recvtimes,1, MPI_DOUBLE,
                0,MPI_COMM_WORLD);
  if(taskid==0){
    printf("\n");
    printf("##########################################################\n\n");
    printf("                --> DESPUES DE LAS COMUNICIONES <-- \n\n");
    for(itask=0;itask<ntasks;itask++){
      printf("Tarea %d : La suma de los vectores recibidos = %e : Tiempo= %f segundos\n",
            itask,recvbuffsums[itask],recvtimes[itask]);
    }
    printf("\n");
    printf("##########################################################\n\n");
    printf(" Tiempo de comunicacion : %f segundos\n\n",totaltime);
    printf("##########################################################\n\n");
  }
  /* Liberacion de la memoria                                      */
  free(recvbuff);
  free(sendbuff);
  /* Finalizacion de MPI                                           */
  MPI_Finalize();
}
```

B.3. Paso de mensajes no bloqueantes

Este ejemplo muestra que MPI_Isen y MPI_Irecv son mucho más apropiadas para realizar este trabajo.

MPI_Isen y MPI_Irecv son no bloqueantes, lo que significa que la función que se llama regrese antes que la comunicación sea completada. El *deadlock* entonces se vuelve imposible con la comunicación no bloqueante, pero otras precauciones deben ser tomaddas cuando se usan. En particular, se tiene que asegurar en un cierto punto que los datos han llegado. Para lo anterior se coloca una llamada a MPI_Wait para cada envio y/o recepción cuando se requieran los datos para poder continuar con el programa.

Es claro que usando llamadas no bloqueantes para este ejemplo, todos los intercambios de datos entre dos tareas ocurren al mismo tiempo.

B.3.1. Código fuente en C/MPI utilizando MPI_Isend y MPI_Irecv

```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "math.h"
#include "mpi.h"

int main(int argc,char** argv){
   int          taskid, ntasks;
   MPI_Status   status;
   MPI_Request  send_request,recv_request;
   int          ierr,i,j,itask,recvtaskid;
   int          buffsize;
   double       *sendbuff,*recvbuff;
   double       sendbuffsum,recvbuffsum;
   double       sendbuffsums[1024],recvbuffsums[1024];
   double       inittime,totaltime,recvtime,recvtimes[1024];

   /* Comienzo del programa, despues declaracion de variables.        */
   MPI_Init(&argc, &argv);
   /* Obtiene el numero de tareas de MPI y el identificador de esta tarea.      */
   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
   /* Obtiene el valor del buffsize desde los argumentos del programa.             */
   buffsize=atoi(argv[1]);

   /* Imprime la descripcion del ejemplo.                */
   if ( taskid == 0 ){
     printf("\n\n\n");
     printf("##########################################\n\n");
     printf("␣Ejemplo␣2␣\n\n");
     printf("␣Comunicacion␣Punto␣a␣Punto:␣MPI_Isend␣MPI_Irecv␣\n\n");
     printf("␣Tamaño␣del␣vector:␣%d\n",buffsize);
     printf("␣Nummero␣de␣tareas:␣%d\n\n",ntasks);
     printf("##########################################\n\n");
     printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣−−>␣ANTES␣DE␣LAS␣COMUNICACIONES␣<−−\n\n");
   }

   /* Alojamiento de la memoria                              */
   sendbuff=(double *)malloc(sizeof(double)*buffsize);
   recvbuff=(double *)malloc(sizeof(double)*buffsize);

   /* Inicializacion de los Vectors y/o matrices.                */
   srand((unsigned)time( NULL ) + taskid);
   for(i=0;i<buffsize;i++){
     sendbuff[i]=(double)rand()/RAND_MAX;
   }

   /* Imprime antes de las comunicaciones.                         */
   sendbuffsum=0.0;
   for(i=0;i<buffsize;i++){
     sendbuffsum += sendbuff[i];
   }
   ierr=MPI_Gather(&sendbuffsum,1,MPI_DOUBLE,
                   sendbuffsums,1, MPI_DOUBLE,
                   0,MPI_COMM_WORLD);
```

```
if(taskid==0){
  for(itask=0;itask<ntasks;itask++){
    recvtaskid=itask+1;
    if(itask==(ntasks-1))recvtaskid=0;
    printf("Tarea %d : La suma de los vectores enviados a %d = %e\n",
            itask,recvtaskid,sendbuffsums[itask]);
  }
}

/* Comunicacion.                                          */
inittime = MPI_Wtime();

if ( taskid == 0 ){
  ierr=MPI_Isend(sendbuff,buffsize,MPI_DOUBLE,
          taskid+1,0,MPI_COMM_WORLD,&send_request);
  ierr=MPI_Irecv(recvbuff,buffsize,MPI_DOUBLE,
          ntasks-1,MPI_ANY_TAG,MPI_COMM_WORLD,&recv_request);
  recvtime = MPI_Wtime();
}
else if( taskid == ntasks-1 ){
  ierr=MPI_Isend(sendbuff,buffsize,MPI_DOUBLE,
          0,0,MPI_COMM_WORLD,&send_request);
  ierr=MPI_Irecv(recvbuff,buffsize,MPI_DOUBLE,
          taskid-1,MPI_ANY_TAG,MPI_COMM_WORLD,&recv_request);
  recvtime = MPI_Wtime();
}
else{
  ierr=MPI_Isend(sendbuff,buffsize,MPI_DOUBLE,
          taskid+1,0,MPI_COMM_WORLD,&send_request);
  ierr=MPI_Irecv(recvbuff,buffsize,MPI_DOUBLE,
          taskid-1,MPI_ANY_TAG,MPI_COMM_WORLD,&recv_request);
  recvtime = MPI_Wtime();
}
ierr=MPI_Wait(&send_request,&status);
ierr=MPI_Wait(&recv_request,&status);

totaltime = MPI_Wtime() - inittime;

/* Imprime despues de las comunicaciones.                          */
recvbuffsum=0.0;
for(i=0;i<buffsize;i++){
  recvbuffsum += recvbuff[i];
}

ierr=MPI_Gather(&recvbuffsum,1,MPI_DOUBLE,
                recvbuffsums,1, MPI_DOUBLE,
                0,MPI_COMM_WORLD);

ierr=MPI_Gather(&recvtime,1,MPI_DOUBLE,
                recvtimes,1, MPI_DOUBLE,
                0,MPI_COMM_WORLD);

if(taskid==0){
  printf("##################################################\n\n");
  printf("                --> DESPUES DE LAS COMUNICIONES <-- \n\n");
  for(itask=0;itask<ntasks;itask++){
    printf("Tarea %d : La suma de los vectores recibidos= %e : Tiempo=%f segundos\n",
            itask,recvbuffsums[itask],recvtimes[itask]);
  }
```

```
        printf("\n");
        printf("###################################################\n\n");
        printf("␣Tiempo␣de␣comunicacion:␣%f␣segundos\n\n",totaltime);
        printf("###################################################\n\n");
    }
    /* Liberacion de la memoria                                            */
    free(recvbuff);
    free(sendbuff);
    /* Finalizacion de MPI                                                 */
    MPI_Finalize();
}
```

Índice alfabético

Arquitecturas paralelas reales, 14

Cómputo paralelo, 1, 2, 8, 18, 28
Canny, 38
Cluster of Workstations, 16
Communicating Sequential Elements, 21, 44, 46, 55, 59, 68
Comunicación Asíncrona, 23, 45
Comunicación Bloqueante, 23
Comunicación Colectiva, 23
Comunicación no Bloqueante, 23
Comunicación Síncrona, 44

Deadlock, 77, 79
Detección de bordes, 38
Distributed Shared Memory machine, 15, 28

Eficiencia, 28

Flynn, 6, 9, 28

Imágenes a color, 36
Imágenes a escala de grises, 35
Intervalos de confianza, 5, 73

Manager-Workers, 19, 43, 46, 55, 56, 68
Massively Parallel Processor, 15
Memoria compartida, 11–13
Memoria compartida distribuida, 13
Memoria Distribuida, 12
MIMD, 11
MISD, 10
MPI, 23, 24
Multicomputadora, 9, 12
Multiprocesador, 9, 11, 12

Overhead, 26, 28

Paralelismo de Actividad, 18
Paralelismo de Dominio, 18
Paralelismo Funcional, 18
Parallel Layers, 20
Parallel Vector Processor, 14, 28
Paso de mensajes, 15, 23
Patrones Arquitectónicos para Programación Paralela, 18

Shared Resource, 22
SIMD, 10
SISD, 9
Speedup, 7, 25–27, 59, 60, 62, 63, 68
Symmetric Multiprocessor, 14
Symmetric Multiprocessors, 28

Tiempo de Ejecución, 26