# UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

## FACULTAD DE INGENIERÍA

DESARROLLO DE UNA APLICACIÓN
DE INGENIERÍA EN CLUSTERS
DE ALTO DESEMPEÑO

## TESIS PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

P R E S E N T A N:

ÁVILA JIMÉNEZ EDUARDO
CANTERA RUBIO FERNANDO NAHÚ
CASTAÑEDA CABALLERO ERIC RICARDO

DIRECTORA DE TESIS ING. LAURA SANDOVAL MONTAÑO
CODIRECTOR DE TESIS M en I. AURELIO ADOLFO MILLÁN NÁJERA

CUIDAD UNIVERSITARIA MÉXICO, D.F. 2005

Universidad Nacional
Autónoma de México

**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.

EDUARDO ÁVILA JIMÉNEZ
FERNANDO NAHÚ CANTERA RUBIO
ERIC RICARDO CASTAÑEDA CABALLERO

# Desarrollo de una Aplicación de Ingeniería en Clusters de Alto Desempeño

ABRIL 2005

## *Agradecimientos*

Merecen un agradecimiento para aquellas personas, pues sin su participación explícita o implícita, voluntaria o involuntaria, este proyecto no existiría:

- A la Ing. Laura Sandoval Montaño, cuya colaboración en la realización del seminario que desembocó en este proyecto, su dirección durante la realización del mismo y su apoyo, entusiasmo y paciencia durante el tiempo que tomó completar el proyecto, fueron esenciales.

- A la Ing. Elba Karén Sáenz García, responsable del Laboratorio de Telemática del Departamento de Ingeniería en Computación de la Facultad de Ingeniería, por dar asilo al Cluster y mantener en buen funcionamiento de éste.

- Al Ing. Agustín Calderón Lara, gerente de Integradores de Soluciones Empresariales en Tecnología de Información, por prestarnos sus instalaciones para poder realizar este proyecto.

- A nuestros amigos, por habernos brindarnos su apoyo para la realización de este proyecto.

Para poder llegar hasta este punto en el que me encuentro ahora, muchas personas me han ayudado y/o apoyado. Tal vez no pueda mencionarlas a todas, pero quiero agreceder principalmente:

*A mi padre*

Por que eres un hombre maravilloso y por que gracias a ti me has dado la oportunidad de vivir y la tarea de ser alguien en la vida.

*A mi familia*

A ustedes que siempre tienen una palabra oportuna en todo momento y me han apoyado siempre.

*A mis compañeros de Tesis*

Por que sin ellos no hubiese podido realizar el presente trabajo.

*A Liliana G. E.*

Por haberme apoyado durante la realización del presente trabajo y por haber compartido conmigo muchas victorias y no menos derrotas.

*A mis amigos*

A todos aquellos que me consideraron su amigo y que desinteresadamente me brindaron su amistad y su apoyo.

Y a todas las personas que de una u otra manera me han ayudado a llegar hasta aquí.

*Eduardo Ávila Jiménez*

El haber llegado al punto a donde hoy me encuentro es reflejo del apoyo que hasta hoy me han brindado muchas personas. Pero en especial muchas gracias:

*A mis padres*

Con los que siempre estaré en deuda por el hecho de haberme dado la vida. Que nunca dudaron de mi capacidad de superación y me infundieron siempre la confianza para seguir adelante a pesar de las adversidades. Gracias también por todos sus esfuerzos, sacrificios y no menos meritorias desveladas para lograr que mis hermanas y yo seamos personas independientes y responsables.

*A mis hermanas*

Agradezco el empeño y responsabilidad con que cada una lleva a cabo sus actividades, siempre buscando ser mejor cada día; ello fue un factor decisivo para la conclusión de mi carrera y ahora de este trabajo.

*A mis compañeros de Tesis*

Por brindarme la oportunidad de trabajar con ellos y ayudarme a llevar a buen término este trabajo.

Merecen una mención especial aquellas personas cuya intervención directa o indirecta hicieron posible la conclusión de mis estudios y ahora la terminación de este trabajo de tesis:

Ing. Laura Sandoval Montaño sin cuyo apoyo este proyecto no hubiera sido posible.

Mis compañeros y amigos del Colegio de Ciencias y Humanidades Plantel Azcapotzalco con los que he compartido muchas victorias y no menos derrotas. Gracias a todos ustedes. Y a todas las personas que de una u otra manera me han ayudado a llegar hasta aquí.

*Fernando Nahú Cantera Rubio*

*A mis padres*

Por su invaluable apoyo, paciencia y tolerancia, ya que siempre escuché de ustedes palabras de aliento y esperanza que me dieron las fuerzas necesarias para continuar con mi camino. Por nunca haber dudado en sacrificar parte de su vida para formarme y educarme.

*A mi Hermana y Hermanos*

Porque siempre han formado parte de mi vida, me han comprendido y dado lo mejor de ustedes sin esperar nada a cambio, porque me han sabido escuchar y brindar su ayuda cuando es necesario.

*A mis compañeros de Tesis*

Por haberme dado la oportunidad de participar con ustedes en la elaboración de este trabajo y compartir un momento especial en nuestras vidas.

*A mis amigos*

Por su incansable apoyo, por haberme enseñado muchas cosas, por aguantarme tantas cosas y por creer en mí.

*Eric Ricardo Castañeda Caballero*

## *Reconocimientos*

***Metodología orientada al procesamiento paralelo***
Eduardo Ávila Jiménez, Fernando Nahú Cantera Rubio,
Eric Ricardo Castañeda Caballero, Ing. Laura Sandoval Montaño.
Conferencia basada en el capítulo II del presente proyecto.
V Semana de Supercómputo DGSCA, UNAM.
12 de Noviembre del 2004.

***Metodología orientada al procesamiento paralelo***
Ing. Laura Sandoval Montaño.
Artículo que hace referencia al capítulo II del presente proyecto.
Boletín de Ingeniería en Computación, Volumen 2 Nº 3, pp. 7-8.
Facultad de Ingeniería, UNAM.
Octubre, 2004.

# Índice

**Índice** ix

**Introducción** xiii

**1. Antecedentes** 1
1.1. SISTEMAS DE CÓMPUTO PARALELO 4
*1.1.1. Arquitecturas de computadoras paralelas* 8
*1.1.1.1. SISD* 9
*1.1.1.2. SIMD* 9
*1.1.1.3. MIMD* 10
*1.1.1.3.1. Sistemas de Memoria Compartida* 11
*1.1.1.3.2. Sistemas de Memoria Distribuida* 13
*1.1.1.3.3. Sistemas de Memoria Compartida-Distribuida* 16
*1.1.1.4. MISD* 18
1.2. SISTEMAS DE CÓMPUTO DISTRIBUIDO 18
*1.2.1. Hardware en los Sistemas Distribuidos* 22
*1.2.2. Clasificación de los Sistemas Distribuidos* 23
*1.2.2.1. Modelo de Minicomputadoras* 23
*1.2.2.2. Modelo de Estación de Trabajo* 24
*1.2.2.3. Modelo de Microprocesadores de Pooling* 24
*1.2.3. Modelos de procesamiento distribuido* 25
*1.2.3.1. Procesamiento distribuido basado en la entrada y salida* 25
*1.2.3.2. Procesamiento distribuido basado en llamadas a procedimientos remotos* 25
*1.2.3.3. Procesamiento distribuido basado en objetos distribuidos* 25
*1.2.3.4. Procesamiento distribuido basado en memoria compartida* 25
*1.2.4. Comunicación en los Sistemas Distribuidos* 25
1.3. CÓMPUTO PARALELO/DISTRIBUIDO EN CLUSTERS 31
*1.3.1. Arquitectura* 33
*1.3.2. Clasificación* 35
*1.3.3. Componentes de Hardware* 37
*1.3.3.1. Hardware de nodos* 38
*1.3.3.2. Hardware de red* 40
*1.3.4. Lenguajes y compiladores* 44
*1.3.5. Entornos de programación y Sistemas Operativos* 46
*1.3.6. Equilibrado de cargas* 47
*1.3.7. Cluster tipo Beowulf* 49

**2. Metodología orientada al procesamiento paralelo** 53
2.1. FUNDAMENTOS EN SISTEMAS DE CÓMPUTO PARALELO 55
*2.1.1. ¿Qué es paralelismo y qué es cómputo paralelo?* 56
*2.1.2. ¿Qué es computación paralela y qué aspectos involucra?* 56
*2.1.3. ¿Qué es una computadora paralela?* 56
*2.1.4. ¿Cuándo hay que paralelizar?* 56

*2.1.5. ¿Qué se necesita para paralelizar?* 57
2.2. NIVELES DE PARALELISMO 58
*2.2.1. Granularidad* 58
*2.2.1.1. Teoría de la paralelización de grano fino* 59
*2.2.1.2. Teoría de la paralelización de grano medio* 59
*2.2.1.3. Teoría de la paralelización de grano grueso* 59
*2.2.1.4. Teoría de la paralelización de grano muy grueso* 60
*2.2.1.5. Teoría de la paralelización de grano independiente* 60
2.3. MODELOS DE PARALELIZACIÓN 60
*2.3.1. Maestro-Esclavo* 61
*2.3.2. Divide y Vencerás* 66
2.4. ETAPAS EN LA CREACIÓN DE PROGRAMAS PARALELOS 67
*2.4.1. Particionamiento* 69
*2.4.1.1. Descomposición* 69
*2.4.1.2. Aglomeración (Asignación)* 70
*2.4.2. Orquestación* 71
*2.4.2.1. Sincronización* 71
*2.4.2.2. Comunicación* 72
*2.4.3. Mapeo* 73
2.5. PARADIGMAS DE LA PROGRAMACIÓN PARALELA BASADO EN LA NATURALEZA DEL ALGORITMO 74
*2.5.1. Teoría de paralelización homogénea (homoparalelismo)* 74
*2.5.2. Teoría de paralelización heterogénea(hetereoparalelismo)* 74
2.6. BIBLIOTECAS DE PROGRAMACIÓN PARALELA 75
*2.6.1. Biblioteca de paso de mensajes* 76
*2.6.1.1. PVM* 76
*2.6.1.1.1. Historia* 76
*2.6.1.1.2. Diseño* 77
*2.6.1.1.3. Implementación* 78
*2.6.1.2. MPI* 79
*2.6.1.2.1. Historia* 81
*2.6.1.2.2. Diseño* 81
*2.6.1.2.3. Implementación* 83
*2.6.1.3. Comparación entre MPI y PVM* 85

***3. Desarrollo de una aplicación de ingeniería*** 87
3.1. PLANTEAMIENTO DEL PROBLEMA, PROCESAMIENTO DIGITAL DE UNA IMAGEN 90
3.2. ALTERNATIVA DE SOLUCIÓN, SUAVIZADO DE IMÁGENES 91
3.3. MEDIOS A UTILIZAR 95
*3.3.1. Cluster tipo Beowulf* 95
*3.3.1.1. Hardware* 95
*3.3.1.1.1. Comunicación entre nodos* 96
*3.3.1.1.2. Consideraciones para equipos sin disco duro* 97
*3.3.1.1.3. Integración de hardware* 98
*3.3.1.2. Software* 100
3.4. DISEÑO DE LA APLICACIÓN EN PARTICULAR 105
*3.4.1. Lectura de la imagen* 107

*3.4.2. Selección del filtro* 109
*3.4.3. Transformada de Fourier* 109
*3.4.3.1. Transformada de Fourier en matrices* 110
*3.4.4. Multiplicación de matrices* 111
*3.4.5. Implementación paralela* 111
*3.4.5.1. Elección de organización* 112
*3.4.5.2. El algoritmo en paralelo* 112
*3.4.5.3. Implementación* 114
*3.4.6. Compilación* 116
*3.4.7. Ejecución* 117

**4. Resultados** 119

**Conclusiones** 131

***Bibliografía*** 137

***Apéndice*** 145

# Introducción

La principal motivación para realizar esta tesis es introducir el empleo del procesamiento paralelo como herramienta para reducir tiempos de cómputo en aplicaciones tradicionalmente costosas de análisis, de diseño, simulación e instrumentación de procesos industriales. En particular, se proyecta trabajar sobre un ambiente de procesamiento distribuido, donde diferentes estaciones de trabajo colaboran para resolver un problema en común. Este enfoque constituye la tendencia actual en materia de procesamiento paralelo pues involucra menores costos de inversión, permite distribuir mejor las tareas hacia las estaciones de trabajo más apropiadas, aprovecha los recursos existentes y brinda mejores posibilidades de expansión.

En el Capítulo I se da una descripción de la evolución de la computación pasando de la era secuencial a la paralela, se introducen conceptos de procesamiento paralelo y distribuido, se expone la clasificación clásica de las máquinas paralelas, propuesta por M. Flynn, así como el de las máquinas contemporáneas, en donde se introduce el concepto de Cluster. Por lo que entendemos como Cluster un sistema de procesamiento paralelo o distribuido que consiste en una colección de computadoras autónomas interconectadas trabajando unidas como un solo recurso.

Existen diferentes tipos de Cluster y su clasificación depende del sistema operativo instalado, arquitectura de computadoras utilizada e interconexión de red empleada por las mismas. Por lo que es importante conocer el tipo de hardware a utilizar para su construcción y las diferentes plataformas bajo las que pueden trabajar los diferentes sistemas operativos. Dentro de la clasificación de Cluster encontramos al Cluster tipo Beowulf, el cual será la base del presente trabajo.

En el Capítulo II se propone una metodología o etapas que se deben de considerar para poder realizar una aplicación, desde su contemplación hasta su programación mediante bibliotecas de programación, cuya ejecución será en un Cluster de alto desempeño. También colocamos una serie de condiciones para saber si el problema a resolver se puede paralelizar y/o ejecutar en un Cluster.

El segundo Capítulo empieza con una serie de preguntas que se deben de hacer para saber si un problema se puede paralelizar y en qué momento hay que hacerlo, al mismo tiempo se abordan diversos modelos de programación que permite entender cómo se pueden desarrollar diferentes programas y colocarlos en la máquina paralela, así como el nivel de paralelismo que se puede tener. Posteriormente se presentan algunas etapas que se deben considerar para la creación de programas paralelos y bibliotecas de programación que nos ayudarán a poder implementar un programa de forma paralela a través de algún lenguaje de programación como C, C++ y Fortran.

En el Capítulo III se propone el diseño de una aplicación en ingeniería, suavizado de una imagen con ruido, utilizando un equipo de alto desempeño de tipo Cluster y definiendo una metodología orientada a procesamiento distribuido y paralelo. Esta aplicación fue escogida con base en que se requiere el manejo de cantidades masivas de datos y cálculos que al combinar el poder de muchas

máquinas del tipo estación de trabajo se pueden alcanzar niveles de rendimiento similares a los de las supercomputadoras, pero a menor costo.

El tercer Capítulo empieza con una serie de argumentos del por qué se decidió realizar un procesamiento digital de imágenes como una aplicación de ingeniería, al mismo tiempo se enfocó, en este mismo rubro, a definir en particular cuál sería la aplicación en desarrollar. Una vez definida la aplicación a desarrollar, se prosiguió a explicar de manera más detallada nuestro instrumento de trabajo, un Cluster de alto desempeño tipo *Beowulf*. Posteriormente se desarrolló un análisis minucioso sobre la aplicación a desarrollar en forma paralela, con base en la metodología expuesta en el Capítulo II. En este diseño exponemos las condiciones básicas de la aplicación y en qué términos se realizará, además de todos los pasos a seguir para lograr el suavizado de una imagen.

Finalmente en el Capítulo IV presentamos una evaluación de desempeño de la aplicación que se desarrolló, suavizado de imágenes.

## OBJETIVO

Desarrollar el procesamiento de una imagen por medio de la aplicación de un filtro para su restauración, donde la imagen contará con algún tipo de ruido, a través del uso de un equipo de alto desempeño de tipo Cluster, definiendo una metodología orientada a procesamiento distribuido y paralelo.

## RESULTADOS ESPERADOS

Obtener una aplicación de ingeniería de alta calidad que trabaje en un Cluster de alto desempeño.

Contar con una metodología a seguir para el desarrollo de aplicaciones en sistemas de procesamiento paralelo.

Al ejecutar la aplicación de ingeniería, los resultados de las pruebas de desempeño deben permitir llegar a la conclusión de que el Cluster de alto desempeño tipo *Beowulf* brinda un desempeño mayor que el de una solución de un solo procesador.

# Capítulo I
## Antecedentes

# *Antecedentes*

En el mundo de los sistemas de cómputo existen diversas arquitecturas de computadoras en paralelo, pero se diferencian en cómo se organizan sus procesadores, memoria e interconexiones. Los sistemas de cómputo más comunes son:

- Massive Parallel Processors (MPP).
- Symmetric Multiprocessors (SMP).
- Cache-Coherent Nonuniform Memory Access (CC-NUMA).
- Sistemas Distribuidos.
- Clusters.

Un MPP es un gran sistema de procesamiento paralelo con una arquitectura que típicamente está formada de varios cientos de elementos de procesamiento que están interconectados a través de una red de interconexión de alta velocidad. Cada elemento suele disponer de una memoria principal y de uno o más procesadores. En cada nodo se ejecuta una copia separada del sistema operativo.

Un sistema SMP suele disponer de 2 a 64 procesadores y en su arquitectura todos los elementos son compartidos. En estos sistemas, todos los procesadores comparten todos los recursos disponibles. Es característico, además, que sólo se ejecuta una copia del sistema operativo.

CC-NUMA es un sistema escalable multiprocesador que dispone de una memoria de acceso no uniforme. Al igual que los SMP, cada procesador en un CC-NUMA tiene una visión global de la memoria. Este tipo de sistema recibe su nombre debido a que el acceso a memoria requiere distintos tiempos de acceso según la distancia a la que se encuentra la parte de memoria a la que se accede.

Un Sistema Distribuido es una colección de computadoras independientes de sus recursos conectados por una red de comunicaciones convencional, que el usuario percibe como un solo sistema y puede acceder a los recursos remotos de la misma forma en que accede recursos locales. Cada máquina ejecuta su propio sistema operativo.

Un Cluster, también llamado NOW[1] ó COW[2], es una colección de estaciones de trabajo[3] o computadoras personales que están interconectados mediante alguna tecnología de red. Generalmente las redes que se forman mediante la utilización de Clusters son de alta velocidad. La principal característica de un Cluster es que trabaja como una colección integrada de recursos y dispone de una única imagen del sistema que es compartida por todos sus nodos.

El uso de Clusters para desarrollar, depurar y ejecutar aplicaciones paralelas está ganando popularidad y se está convirtiendo en una gran alternativa para el uso de arquitecturas más

[1] NOW = *Network Of Workstations*, Red de estaciones de trabajo.
[2] COW = *Cluster Of Workstations*, Cluster de estaciones de trabajo.
[3] Computadoras para usuarios finales.

especializadas. Un factor importante que ha hecho que el uso de Clusters sea más práctico es la estandarización de numerosas herramientas y utilidades usadas en aplicaciones paralelas. Algunos de los estándares más importantes son la librería de paso de mensajes MPI (*Message Passing Interface*) y el lenguaje paralelo de datos HPF (*High Performance Fortran*). Esta estandarización permite que las aplicaciones sean desarrolladas y probadas en Clusters y en una etapa posterior a la programación puedan ser llevadas, con ligeras modificaciones, a plataformas paralelas especializadas.

La tabla 1.1 muestra una comparación de las arquitecturas y características funcionales de las máquinas mencionadas anteriormente, originalmente propuesta por Hwang y Xu[4].

| Características | MPP | SMP-NUMA | Cluster | Distribuido |
|---|---|---|---|---|
| Número de Nodos | 100 -1000 | 10 - 100 | 100 o menos | 10 -1000 |
| Complejidad del nodo | Grano fino o medio | Grano fino o tosco | Grano medio | Rango amplio |
| Comunicación entre nodos | Paso de Mensajes/Variables compartidas para una memoria distribuida y compartida | Memoria compartida, centralizada y distribuida | Paso de Mensajes | Archivos compartidos, RPC, Paso de Mensajes e IPC |
| Calendarización de trabajos | Una sola cola en el host | Una sola cola principal | Múltiples colas, pero coordinadas | Colas independientes |
| Soporte para SSI[5] | Parcial | Siempre en SMP y en algunas NUMA | Deseado | No |
| Copias del Sistema Operativo en los nodos y tipo | N micro – Kernel's monolíticos o sistemas operativos de capas | Para los SMP uno monolítico y para las NUMA varios | N plataformas para los sistemas operativos homogéneos o micro Kernel | N plataformas de los sistemas operativos homogéneos |
| Espacio de direccionamiento | Múltiple. Sencillo para DSM | Sencillo | Múltiple o sencillo | Múltiple |
| Seguridad entre los nodos | No necesaria | No necesaria | Requerido si está expuesto | Requerido |
| Propietarios | Una organización | Una organización | Una o más organizaciones | Muchas organizaciones |

***Tabla 1.1 Comparación de las arquitecturas y características de máquinas paralelas***

## 1.1. SISTEMAS DE CÓMPUTO PARALELO

Los sistemas de cómputo en paralelo se refieren a los conceptos de acelerar la ejecución de un programa dividiendo el programa en múltiples fragmentos que se pueden ejecutar simultáneamente, cada uno con su propio procesador. Un programa ejecutado a través de *x*

[4] Rajkumar Buyya
***High Performance Cluster Computing***: *Architectures and System*, Vol. 1, 1/e
School of Computer Science and Software Engineering.
Monash University, Melbourne, Australia.
Copyright 1999, pp 881.

[5] SSI = *Single System Image*, Sistema Simple de Imagen. Ilusión, creada por software o hardware, que hace parecer al cluster como una sola máquina para el usuario, las aplicaciones y la red (aporta transparencia). Cada nodo puede acceder a un periférico o disco sin necesidad de saber su ubicación física.

procesadores va a ejecutar la misma operación $x$ veces más rápido que si se usara un solo procesador.

A pesar de que el uso de múltiples procesadores puede acelerar muchas operaciones, la mayoría de las aplicaciones no pueden aún beneficiarse de este proceso, básicamente el procesamiento en paralelo es apropiado si y solo si la aplicación tiene las instrucciones de paralelismo necesario para hacer uso de múltiples procesadores. En parte, es muy importante identificar las porciones del programa que pueden ser ejecutadas independientemente y simultáneamente en procesadores separados.

El alto desempeño de los sistemas de cómputo en paralelo es logrado por repartición de tareas grandes y complejas a los múltiples procesadores. Durante la Segunda Guerra Mundial, antes de la llegada de la computadora electrónica, una técnica similar fue usada para llevar a cabo grandes cálculos asociado con el diseño de la bomba atómica del proyecto *Manhattan*. Para reducir la cantidad de tiempo significativo se resolvió un problema matemático grande, cada una de sus partes del problema fueron realizadas por diferentes personas. Hoy en día, las computadoras electrónicas pueden trabajar en la armonía para resolver problemas científicos no soñados hace algunas décadas.

A continuación se presenta una cronología de la evolución del procesamiento en paralelo:

- **1955**
  La *IBM 704* usa circuitos aritméticos paralelos binarios junto con una unidad de punto flotante que aceleraban significativamente el desarrollo de operaciones numéricas frente a las tradicionales unidades aritmético-lógicas. A pesar de su velocidad, aproximadamente 5 kFLOPS[6], las operaciones de E/S resultaban lentas y representaban un cuello de botella. Como solución a este problema la IBM decide incorporar procesadores de E/S independientes, llamados canales, en modelos posteriores de la 704 y su sucesor, la *IBM 709*.

- **1956**
  IBM inicia el proyecto 7030, llamado *STRETCH*, para producir una supercomputadora para el Laboratorio Nacional Los Álamos. Su meta era crear una máquina 100 veces más poderosa que las de su época.

  Se inicia el proyecto LARC (*Livermore Automatic Research Computer*) para el diseño de una supercomputadora para el Laboratorio Nacional Livermore.

  El proyecto *Atlas* comienza como una aventura conjunta entre la Universidad de Manchester y Ferranti Ltd.

- **1958**
  Bull anuncia la *Gamma 60* con múltiples unidades funcionales e instrucciones *fork* y *join* en su conjunto de instrucciones. Llegaron a construirse diecinueve.

[6] FLOPS = *FLoating point OPeration per Second*, Una operación de punto flotante por segundo.

John Cocke y Daniel Slotnick discuten el uso del paralelismo en cálculos numéricos en un memorandum de la IBM. Posteriormente Slotnick propone *SOLOMON*, una máquina SIMD[7] con 1024 elementos de procesamiento de 1 bit, cada uno con memoria para 128 valores de 32 bits. La máquina nunca se construye pero es el punto de arranque para trabajos posteriores.

- **1959**
  Sperry Rand entrega el primer sistema *LARC*, el cual dispone de un procesador de E/S independiente que operaba en paralelo con una o dos unidades de procesamiento. Sólo se construyeron dos.

  IBM entrega su primera *STRETCH* que presentaba la anticipación de instrucciones y corrección de errores. Se construyen ocho. La tecnología es reutilizada en la *IBM 7090*.

  La primera *IBM 7090* es entregada. Esta es la versión transistorizada de la *IBM 709*.

- **1960**
  Control Data inicia el desarrollo de su *CDC 6600*.

  E. V. Yevreinov en el Instituto de Matemáticas en Novosibirsk comienza sus trabajos en arquitecturas fuertemente acopladas de paralelismo burdo con interconexiones programables.

- **1962**
  Control Data Corporation entrega su primera *CDC 1604*, máquina similar a la *IBM 7090* caracterizada por palabras de 48 bits y ciclos de memoria de 6μs.

  La computadora *Atlas* es operacional. Es la primera máquina en usar memoria virtual y paginación, su ejecución de instrucciones es un oleoducto (pipeline) y contiene unidades aritméticas de punto flotante y punto fijo separadas. Su desempeño es de aproximadamente 200 kFLOPS.

  C. A. Petri describe las *Redes de Petri*, un concepto teórico para la descripción y análisis de las propiedades de sistemas concurrentes.

  Burroughs introduce su *multiprocesador MIMD simétrico D825*. Cuenta de 1 a 4 CPU's que acceden a 1 ó 16 módulos de memoria usando un conmutador de baraje cruzado (crossbar switch). Los CPU's[8] son similares al posterior *B5000*, el sistema operativo es simétrico con una cola compartida.

- **1964**
  Control Data Corporation empieza a producir la *CDC 6600*, la primera supercomputadora en ser un éxito técnico y comercial. Cada máquina tiene un CPU de 60 bits y 10 unidades

[7] Ver subtema *1.1.1.2 SIMD*.
[8] CPU = *Central Processing Unit*, Unidad Central de procesamiento.

periféricas de procesamiento. El CPU utiliza un marcador para manejar la dependencia de instrucciones.

IBM inicia el diseño del *Advanced Computer System* (ACS), capaz de manejar hasta siete instrucciones por ciclo. El proyecto fue cerrado en 1969 pero muchas de las técnicas fueron incorporadas en posteriores computadoras.

Daniel Slotnick propone la construcción de una computadora paralela masiva para el Laboratorio Nacional Lawrence Livermore, pero la Comisión de Energía Atómica da el contrato a CDC, que construye la *STAR-100*. Slotnick consigue el financiamiento de la U.S. Air Force y su diseño evoluciona a la *ILLIAC-IV*. La máquina es construida en la Universidad de Illinois, con Burroughs y Texas Instruments como principales subcontratistas. La *Advanced Scientific Computer* (ASC) de la Texas Instruments crece junto a esta iniciativa.

- **1965**
  General Electric, el Instituto Tecnológico de Masachusetts y AT&T Bell Laboratories comienzan a trabajar en *Multics*. El objetivo del proyecto es la construcción de un sistema operativo de propósito general de memoria compartida, multiprocesamiento y tiempo compartido.

  Edsger Dijkstra describe y nombra el *Problema de las Regiones Críticas*. Mucho del trabajo posterior en sistemas concurrentes es dedicado a encontrar eficientes y seguras formas de manejar regiones críticas.

  James W. Cooley y John W. Tukey describen el *Algoritmo de la Transformada Rápida de Fourier*, que es posteriormente uno de los más grandes consumidores de ciclos de punto flotante.

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

- **1986**
  La *Cray X-MP*, una máquina con 4 procesadores vectoriales, alcanzó una velocidad de 713 MFLOPS.

- **1997**
  La ASCI Red, construida por Intel, alcanzó la marca de 1 TFLOPS.

[9] Un procesador de arquitectura vectorial contiene un procesador matricial. Los procesadores matriciales realizan operaciones sobre matrices o vectores de números en punto flotante. La clave dentro de la arquitectura vectorial es la utilización de ALU's segmentadas o ALU's paralelas.

- **2000**
  La computadora más rápida del mundo, la *ASCI White*, construida por IBM en el Laboratorio Nacional Lawrence Livermore alcanzó un rendimiento de alrededor de 4 TFLOPS.

- **2001**
  La *ASCI White* se expandió y alcanzó un rendimiento sostenido de 7.2 TFLOPS.

Esta tendencia de crecimiento continuo en años siguientes, ya que para el año 2002 se crearon planes para la creación de computadoras con un rendimiento de 30 TFLOPS y de 100 TFLOPS para el año 2004.

La idea básica detrás del procesamiento paralelo es que varios procesadores, ejecutando simultánea y coordinadamente las tareas, pueden rendir más que un único dispositivo. El problema fundamental son las innovaciones tecnológicas que se requieren para obtener ese rendimiento mejorado.

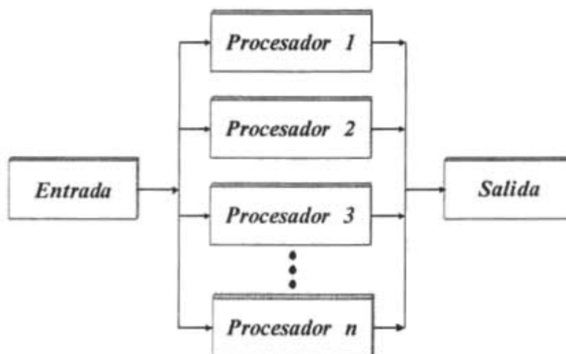


*Fig. 1.1 Principio básico de un procesamiento paralelo*

### *1.1.1. Arquitecturas de computadoras paralelas*

En 1966, Michael Flynn propuso un mecanismo clásico para la clasificación de las computadoras, y aunque no cubre todas las posibles arquitecturas, sí proporciona una importante penetración en varias arquitecturas de computadoras modernas. Esta clasificación está basada en el número de instrucciones y secuencia de datos que la computadora utiliza para procesar la información. Puede haber secuencias de instrucciones sencillas o múltiples y secuencias de datos sencillas o múltiples. Dando lugar a 4 tipos de computadoras, de las cuales sólo dos son aplicables a las computadoras paralelas.

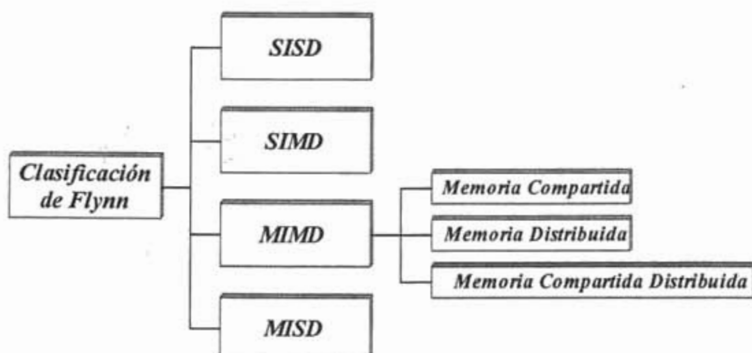


*Fig. 1.2 Clasificación de Flynn para las computadoras paralelas*

### *1.1.1.1. SISD*

El SISD (*Single Instruction stream, Single Data stream*) es el modelo tradicional de computación secuencial, donde una unidad de procesamiento recibe sólo una secuencia de instrucciones que opera en una secuencia de datos, es decir, la primera instrucción se extrae de la memoria y se ejecuta, a continuación se extrae la segunda y se lleva a cabo, y así sucesivamente.

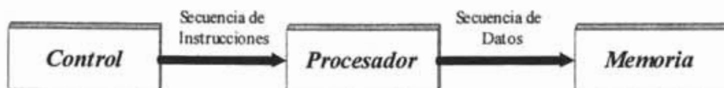


*Fig. 1.3 Modelo SISD*

En cada paso, la unidad de control emite una instrucción que opera sobre datos obtenido de la unidad de memoria. Éste es el modelo Von Neumann, al cual corresponden la mayoría de las computadoras actuales, no presenta ningún paralelismo. Aún así puede lograrse cierto paralelismo si se extrae la siguiente instrucción antes de terminar el procesamiento de la que está en curso.

### *1.1.1.2. SIMD*

A diferencia del SISD, el SIMD (*Single Instruction stream, Multiple Data stream*) tiene múltiples procesadores que sincronizadamente ejecutan la misma secuencia de instrucciones, pero en diferentes datos. La memoria es distribuida.

Esta arquitectura se refiere a un modelo de ejecución en paralelo en el cual todos los procesadores ejecutan la misma operación al mismo tiempo, pero cada procesador se le permite operar sobre sus propios datos. Este modelo naturalmente encaja en el concepto de ejecutar la misma operación en cada elemento del arreglo, y es por lo tanto asociada con vectores de arreglo de manipulación de datos. Todas las operaciones están sincronizadas, las interacciones entre procesadores SIMD tiende a ser más fácil si son eficientemente implementadas.

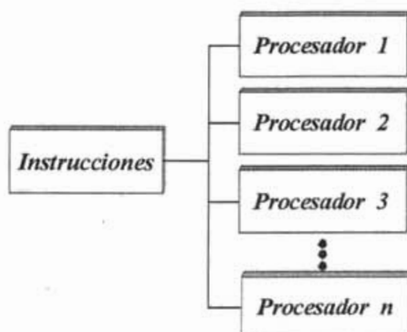


*Fig. 1.4 Modelo SIMD*

Hay *n* secuencias de datos, una por procesador. Cada procesador ejecuta la misma instrucción con diferentes datos. Los procesadores operan sincronizadamente y utilizan un reloj global para asegurar esta operación.

Las computadoras SIMD operan perfectamente sobre arrays[10] y poseen múltiples unidades funcionales para paralelizar el trabajo. Su virtud es que todas estas unidades funcionales de ejecución en paralelo están sincronizadas y todas responden adecuadamente a una sola instrucción que proviene de un único contador de programa.

El desarrollo de SIMD viene motivado por el reducido espacio de memoria que necesita, dado que sólo requiere la zona de programa en ejecución en memoria. Los métodos de intercambio de datos se basan en redes de interconexión.

Para aprovechar un sistema SIMD se debe pensar en ejecución de bucles sobre arrays, por ejemplo, bucles *for*. Los sistemas SIMD han quedado en plano de investigación o estudio dada su poca flexibilidad.

### *1.1.1.3. MIMD*

Esta computadora también es paralela como la SIMD, pero el MIMD (*Multiple Instruction stream, Multiple Data stream*) es asíncrono. No tiene un reloj central. Cada procesador en un sistema MIMD puede ejecutar su propia secuencia de instrucciones y tener sus propios datos. Esta característica es la más general y poderosa de esta clasificación.

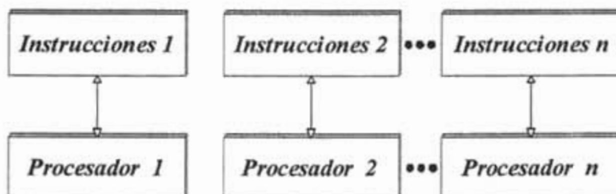


*Fig. 1.5 Modelo MIMD*

[10] Vectores de datos.

Este modelo encaja naturalmente en el concepto de descomponer un programa para ejecuciones paralelas en bases funcionales. Éste es un modelo más flexible que la ejecución SIMD, pero se consigue el riesgo de encontrar errores llamados condiciones de carrera, en la cual un programa intermitentemente falla debido a variaciones de tiempo reordenando las operaciones de un procesador relativas con la de otro.

Se tienen *n* procesadores, *n* secuencias de instrucciones y *n* secuencias de datos. Cada procesador ejecuta su propia secuencia de instrucciones con diferentes datos. Gracias al diseño del software escalable, las computadoras MIMD pueden soportar pérdidas de procesadores por fallo, es decir, si un procesador falla, el sistema obtendrá como resultado a ese fallo un servicio de *n-1* procesadores. En las computadoras MIMD los procesadores operan sincrónicamente; pueden estar haciendo diferentes cosas en diferentes datos al mismo tiempo.

Los sistemas MIMD se clasifican en:

- Sistemas de Memoria Compartida.
- Sistemas de Memoria Distribuida.
- Sistemas de Memoria Compartida Distribuida.

### *1.1.1.3.1. Sistemas de Memoria Compartida*

Las computadoras MIMD con memoria compartida son conocidas como *Sistemas de Multiprocesamiento Simétrico* (SMP), donde múltiples procesadores comparten un mismo sistema operativo y memoria. Otro término con que se le conoce es *Máquina Firmemente Acoplada* o *de Multiprocesadores*.

En este sistema cada procesador tiene acceso a toda la memoria, es decir, hay un espacio de direccionamiento compartido. Los tiempos de acceso a memoria son uniformes, pues todos los procesadores se encuentran igualmente comunicados con la memoria principal, sus lecturas y escrituras tienen los mismos tiempos de latencia[11], y el acceso a memoria es por medio de un canal común, es decir, los procesadores comparten la misma memoria y el bus del sistema.

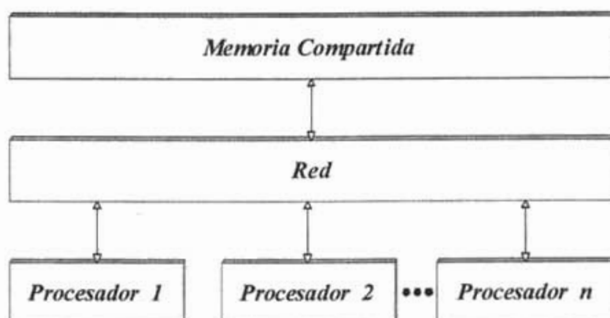


*Fig. 1.6 Sistema de Memoria Compartida*

[11] Tiempo mínimo para transmitir un objeto.

La presencia de un solo espacio de memoria simplifica tanto el diseño del sistema físico (*hardware*) como la programación de las aplicaciones (*software*). Esa memoria compartida permite que un sistema operativo con multiconexión distribuya las tareas entre varios procesadores, o que una aplicación obtenga toda la memoria que necesita para una simulación compleja. La memoria globalmente compartida también vuelve fácil la sincronización de los datos.

En esta configuración, debe asegurarse que los procesadores no tengan acceso simultáneo a regiones de memoria que provoquen algún error. Actualmente los sistemas operativos de estos equipos controlan todo esto mediante mecanismos llamados *semáforos*.

Un *Sistema de Memoria Compartida* es uno de los diseños de procesamiento paralelo más maduros. Sin embargo, la memoria global contribuye al problema más grande de un *Sistema de Memoria Compartida*: conforme se añaden procesadores, el tráfico en el bus de memoria se satura. Al añadir memoria caché a cada procesador se puede reducir algo del tráfico en el bus.

Al manejarse ocho o más procesadores, el cuello de botella se vuelve crítico, inclusive para los mejores diseños, por lo que un sistema de memoria compartida es considerado una tecnología poco escalable[12].

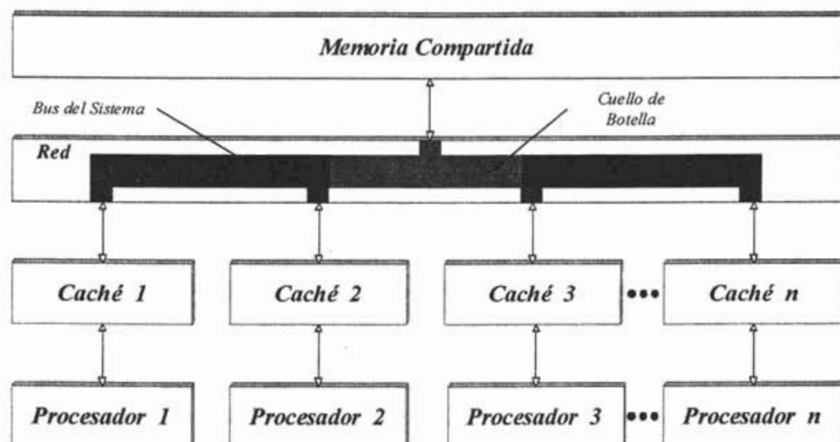


*Fig. 1.7 Sistema de Memoria Compartida con reducción de tráfico*

[12] Capacidad de crecimiento.

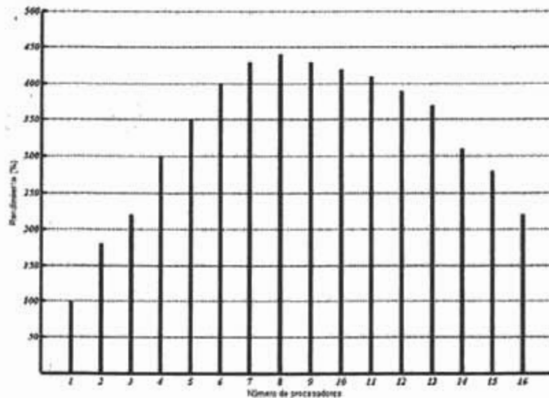


***Fig. 1.8 Rendimiento de un Sistema de Memoria Compartida***

- *Ventajas*
  - Más fácil programar que en sistemas de memoria distribuida.

- *Desventajas*
  - El acceso simultáneo a memoria es un problema.
  - Todos los CPU's comparten el camino a memoria.
  - Un CPU que acceda la memoria, bloqueando el acceso de todos los otros CPU's.

Los primeros componentes utilizados en un *Sistema de Memoria Compartida* fueron procesadores RISC, en la actualidad, debido a su bajo costo, los procesadores CISC avanzados como Pentium III y Pentium IV son empleados con mayor frecuencia.

Un *Sistemas de Multiprocesamiento Simétrico* tiene un diseño simple, efectivo y económico.

### *1.1.1.3.2. Sistemas de Memoria Distribuida*

Las computadoras MIMD de memoria distribuida son conocidas como *Sistemas de Procesamiento Masivamente Paralelo* (MPP), donde múltiples procesadores trabajan en diferentes partes de un programa, usando su propio sistema operativo y memoria. También se les conoce como *Multicomputadoras, Máquinas Libremente Juntas* o *Cluster*.

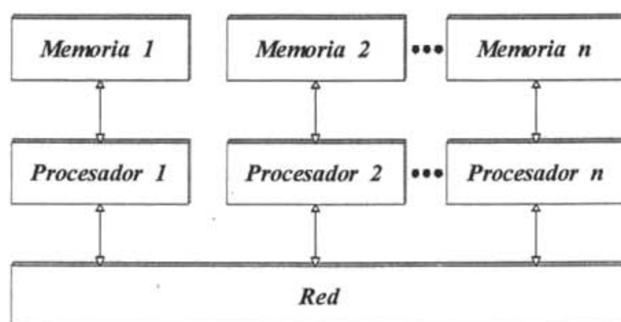


*Fig. 1.9 Sistema de Memoria Distribuida*

Estos sistemas tienen su propia memoria local. Se comparte información sólo a través de mensajes, es decir, si un procesador requiere datos contenidos en la memoria de otro, deberá enviar un mensaje solicitándolos. Esta comunicación se le conoce como *Paso de Mensajes*.

La otra forma de comunicación entre los procesadores es a través de una red de interconexión. En este modelo la memoria es dividida entre el conjunto de procesadores, para su acceso local. Además, cada procesador es conectado con sus vecinos a través de una línea bidireccional de comunicación, la cual le permite enviar o recibir datos en cualquier instante de tiempo, habiéndose desarrollado una amplia variedad de topologías que permiten abarcar una gran cantidad de problemas de manera eficiente, tales como: el *arreglo lineal*, el *anillo*, la *malla*, el *toroide*, el *árbol*, el *fat-tree* y el *hipercubo*, que tienen una baja cantidad de enlaces entre procesadores, de manera tal que cuando sea necesario comunicar un mensaje entre dos procesadores que no tienen conexión directa, debe encaminarse o enrutarse dicho mensaje por procesadores intermedios entre éstos dos.

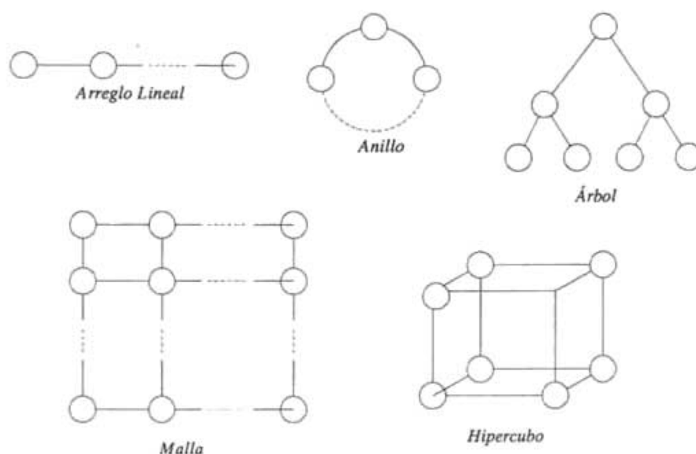


*Fig. 1.10 Topologías de comunicación*

Este sistema reduce el tráfico del bus, debido a que cada sección de memoria interactúa únicamente con aquellos accesos que le están destinados, en lugar de interactuar con todos los accesos a memoria, como ocurre en un sistema SMP. Esto permite la construcción de *Sistemas de Memoria Distribuida* de gran tamaño, con cientos y aún miles de procesadores, por lo que MPP es una tecnología altamente escalable.

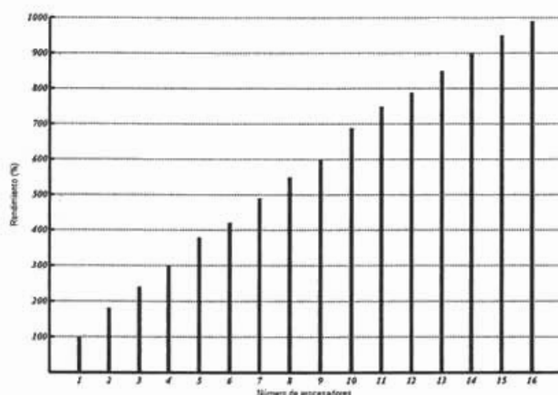


***Fig. 1.11 Rendimiento de un Sistema de Memoria Distribuida***

Para evitar los cuellos de botella en el bus de memoria, este sistema no utiliza memoria compartida, en su lugar, distribuye equitativamente la memoria entre los procesadores de modo que se asemeja a una red, es decir, cada procesador con su memoria distribuida asociada es similar a una computadora dentro de una red de procesamiento distribuido.

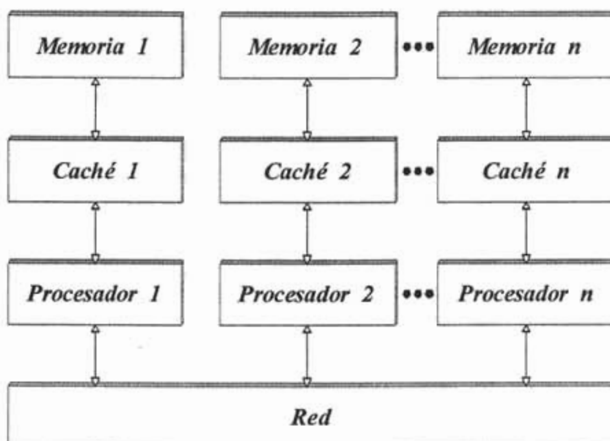


***Fig. 1.12 Sistema de Memoria Distribuida con reducción de tráfico***

La parte negativa del *Sistema de Memoria Distribuida*, desde el punto de vista tecnológico, es que la programación se vuelve difícil, debido a que la memoria se rompe en pequeños espacios separados. Sin la existencia de un espacio de memoria globalmente compartido, ejecutar una aplicación que requiere una gran cantidad de memoria, comparada con la memoria local, puede ser difícil. La sincronización de datos entre tareas ampliamente distribuidas también se complica, particularmente si un mensaje debe pasar por muchos componentes de hardware hasta alcanzar la memoria del procesador destino.

Escribir una aplicación como un *Sistema de Procesamiento Masivamente Paralelo* también requiere estar al tanto de la organización de la memoria manejada por el programa. Donde sea necesario, se deben insertar comandos de paso de mensajes dentro del código del programa. Además de complicar el diseño del software, tales comandos pueden crear dependencias de hardware en las aplicaciones. Sin embargo, la mayor parte de vendedores de computadoras han salvaguardado la portabilidad de las aplicaciones adoptando, sea un mecanismo de dominio público para paso de mensajes conocido como PVM (*Máquina Virtual Paralela*) o un estándar llamado MPI (*Interfaz de Paso de Mensajes*).

El costo de las soluciones basadas en un *Sistema de Procesamiento Masivamente Paralelo* es mucho más alto que el costo por procesador de las soluciones SMP, por lo que su uso sólo se justifica cuando la necesidad de procesamiento es muy alta.

El *Procesamiento Masivamente Paralelo* es una arquitectura computacional de alto rendimiento.

### *1.1.1.3.3. Sistemas de Memoria Compartida-Distribuida*

Las computadoras MIMD de memoria compartida distribuida son conocidas como *Sistemas de Procesamiento Paralelo Escalable* (SPP). Este tipo de sistemas es un híbrido de SMP y MPP, que utiliza una memoria jerárquica de dos niveles para alcanzar la escalabilidad.

Es un conjunto de procesadores que tienen acceso a una memoria compartida pero sin un canal compartido. Esto es, físicamente cada procesador posee su memoria local y se interconecta con otros procesadores por medio de un dispositivo de alta velocidad, y todos ven las memorias de cada uno como un espacio de direcciones globales.

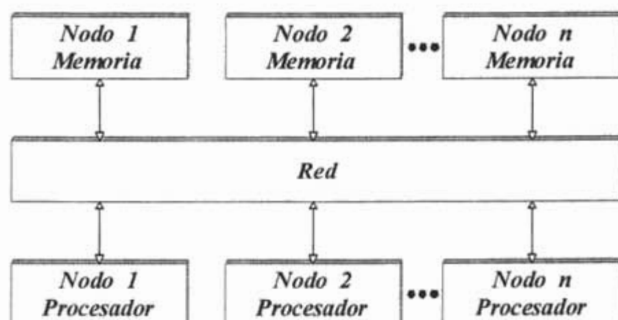


*Fig. 1.13 Sistema de Memoria Compartida-Distribuida*

El acceso a memoria se realiza bajo el esquema de NUMA (*Acceso a Memoria No Uniforme*), la cual toma menos tiempo en acceder a la memoria local de un procesador que acceder a memoria remota de otro procesador.

Este tipo de sistema contiene dos capas: la primera capa consiste de componentes de memoria distribuida que son esencialmente parte de sistemas MPP completos, con múltiples nodos (nodo = procesador + memoria distribuida) y el segundo nivel de memoria está globalmente compartido al estilo SMP.

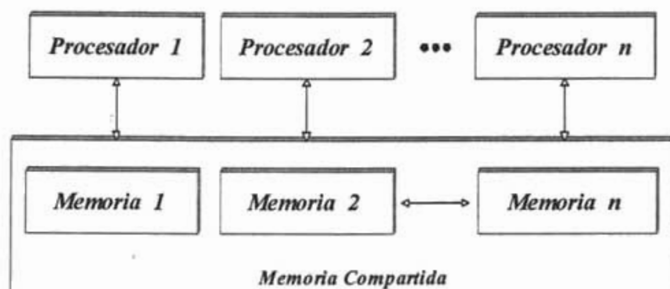


*Fig. 1.14 Capas de un Sistema de Memoria Compartida-Distribuida*

Se construyen *Sistemas de Procesamiento Paralelo Escalable* grandes interconectando dos o más nodos a través de la segunda capa de memoria, de modo que esta capa aparece, lógicamente, como una extensión de la memoria individual de cada nodo.

La memoria de dos niveles reduce el tráfico de bus debido a que solamente ocurren actualizaciones para mantener coherencia de memoria. Por tanto, un *Sistema de Procesamiento Paralelo Escalable* ofrece la facilidad de programación del modelo SMP, a la vez que provee una escalabilidad similar a la de un diseño MPP.

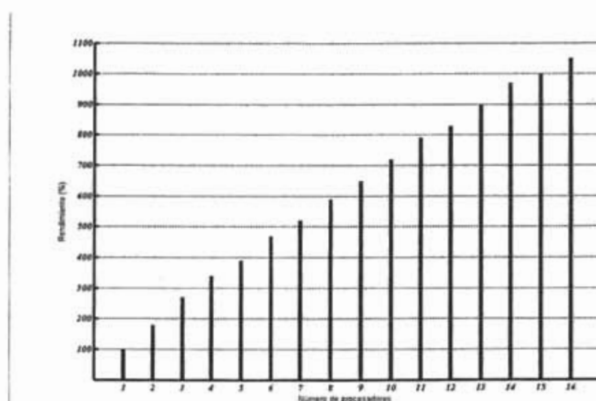

*Fig. 1.15 Rendimiento de un Sistema de Memoria Compartida-Distribuida*

- *Ventajas*
  - Escalable como en los *Sistemas de Memoria Distribuida.*
  - Fácil de programar como en los *Sistemas de Memoria Compartida.*
  - No existe el cuello de botella, como en máquinas de sólo memoria compartida.

### *1.1.1.4. MISD*

En el modelo MISD (*Multiple Instruction stream, Single Data stream*) las secuencias de instrucciones pasan a través de múltiples procesadores. Diferentes operaciones son realizadas en *n* procesadores, cada uno con su propia unidad de control comparten una memoria común.

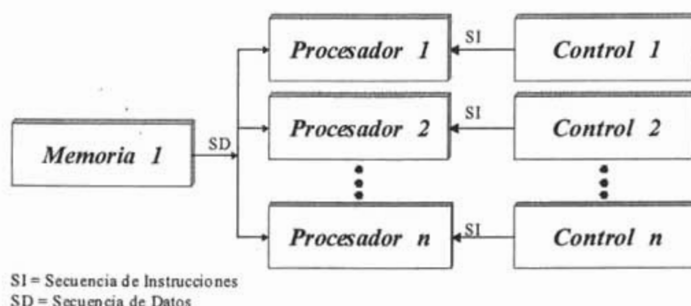


*Fig. 1.16 Modelo MISD*

Existen *n* secuencias de instrucciones y una secuencia de datos. Cada procesador ejecuta diferentes secuencias de instrucción, al mismo tiempo, con el mismo dato. Las computadoras MISD son útiles donde la misma entrada está sujeta a diferentes operaciones. No existe ningún equipo comercial basado en este tipo de arquitectura.

## 1.2. SISTEMAS DE CÓMPUTO DISTRIBUIDO

Un sistema distribuido se define como una colección de computadoras autónomas conectadas por una red de comunicaciones convencional, que el usuario percibe como un solo sistema, las cuales cooperan en la realización de una tarea. El usuario accede a los recursos remotos de la misma manera en que accede a los recursos locales.

En otras palabras, un sistema distribuido es aquel en el que dos o más máquinas colaboran para la obtención de un resultado. En todo sistema distribuido se establecen una o varias comunicaciones siguiendo un protocolo prefijado mediante un esquema.

Los sistemas distribuidos deben de ser muy confiables, ya que si un componente del sistema se descompone otro componente debe de ser capaz de reemplazarlo.

El tamaño de un sistema distribuido puede ser muy variado, ya sean decenas de *hosts*[13] por medio de una LAN[14], centenas de *hosts* en una MAN[15] y miles o millones de *hosts* por medio de Internet.

Algunas características que contienen los sistemas distribuidos son:

- Cada elemento de cómputo tiene su propia memoria y su propio sistema operativo.
- Control de recursos locales y remotos.
- Sistemas abiertos.
- Plataformas no estándar.
- Medios de comunicación.
- Capacidad de Procesamiento en paralelo.
- Dispersión y parcialidad.

Algunos factores que han afectado el desarrollo de los sistemas distribuidos se presentan a continuación:

- Avances tecnológicos.
- Nuevos requerimientos.
- Globalización
- Aspectos externos, como culturales, políticos y económicos.
- Integración de varias redes y sistemas operativos.
- Aspectos externos, como culturales, políticos y económicos.
- Integración de varias redes y sistemas operativos.

Los sistemas distribuidos están basados en las ideas básicas de transparencia, eficiencia, flexibilidad, escalabilidad y fiabilidad. Sin embargo estos aspectos son en parte contrarios, y por lo tanto los sistemas distribuidos han de cumplir en su diseño el compromiso de que todos los puntos anteriores sean solucionados de manera aceptable.

- ***Transparencia***

  El concepto de transparencia de un sistema distribuido va ligado a la idea de que todo el sistema funcione de forma similar en todos los puntos de la red, independientemente de la posición del usuario. Queda como labor del sistema operativo el establecer los mecanismos que oculten la naturaleza distribuida del sistema y que permitan trabajar a los usuarios como si de un único equipo se tratara.

  En un sistema transparente, las diferentes copias de un archivo deben aparecer al usuario como un único archivo. Queda como labor del sistema operativo el controlar las copias, actualizarlas en caso de modificación y en general, la unicidad de los recursos y el control de la concurrencia.

---

[13] Es una computadora conectada a una red que permite a los usuarios comunicarse con otras computadoras dentro de red. Es una computadora que actúa como fuente de información.
[14] LAN = *Local Area Network*, Red de Área Local.
[15] MAN = *Metropolitan Area Network*, Red de Área Metropolitana.

El que el sistema disponga de varios procesadores debe lograr un mayor rendimiento del sistema, pero el sistema operativo debe controlar que tanto los usuarios como los programadores vean el núcleo del sistema distribuido como un único procesador. El paralelismo es otro punto clave que debe controlar el sistema operativo, que debe distribuir las tareas entre los distintos procesadores como en un sistema multiprocesador, pero con la dificultad añadida de que esta tarea hay que realizarla a través de varias computadoras.

- ***Eficiencia***
  La idea base de los sistemas distribuidos es la de obtener sistemas mucho más rápidos que las computadoras actuales. Es en este punto cuando nos encontramos de nuevo con el paralelismo.

  Para lograr un sistema eficiente hay que descartar la idea de ejecutar un programa en un único procesador de todo el sistema, y pensar en distribuir las tareas a los procesadores libres más rápidos en cada momento.

  La idea de que un procesador vaya a realizar una tarea de forma rápida es bastante compleja, y depende de muchos aspectos concretos, como la propia velocidad del procesador, pero también la localidad del procesador, los datos, los dispositivos, etc. Se han de evitar situaciones como enviar un trabajo de impresión a una computadora que no tiene conectada una impresora de forma local.

- ***Flexibilidad***
  Un proyecto en desarrollo como el diseño de un sistema operativo distribuido debe estar abierto a cambios y actualizaciones que mejoren el funcionamiento del sistema. Esta necesidad ha provocado una diferenciación entre las dos diferentes arquitecturas del núcleo del sistema operativo: el núcleo monolítico[16] y el micronúcleo[17]. Las diferencias entre ambos son los servicios que ofrece el núcleo del sistema operativo. Mientras el núcleo monolítico ofrece todas las funciones básicas del sistema integradas en el núcleo, el micronúcleo incorpora solamente las fundamentales, que incluyen únicamente el control de los procesos y la comunicación entre ellos y la memoria. El resto de servicios se cargan dinámicamente a partir de servidores en el nivel de usuario.

[16] Como ejemplo de sistema operativo de núcleo monolítico está UNIX. Estos sistemas tienen un núcleo grande y complejo, que engloba todos los servicios del sistema. Está programado de forma no modular, y tiene un rendimiento mayor que un micronúcleo. Sin embargo, cualquier cambio a realizar en cualquier servicio requiere la parada de todo el sistema y la recompilación del núcleo.

[17] La arquitectura de micronúcleo ofrece la alternativa al núcleo monolítico. Se basa en una programación altamente modular, y tiene un tamaño mucho menor que el núcleo monolítico. Como consecuencia, el refinamiento y el control de errores son más rápidos y sencillos. Además, la actualización de los servicios es más sencilla y ágil, ya que sólo es necesaria la recompilación del servicio y no de todo el núcleo. Como contraprestación, el rendimiento se ve afectado negativamente. En la actualidad la mayoría de sistemas operativos distribuidos en desarrollo tienden a un diseño de micronúcleo. Los núcleos tienden a contener menos errores y a ser más fáciles de implementar y de corregir. El sistema pierde ligeramente en rendimiento, pero a cambio consigue un gran aumento de la flexibilidad.

- ***Escalabilidad***
  Un sistema operativo distribuido debería funcionar tanto para una docena de computadoras como varios millares. Igualmente, debería no ser determinante el tipo de red utilizada ni las distancias entre los equipos, etc.

  Aunque este punto sería muy deseable, puede que las soluciones válidas para unas cuantas computadoras no sean aplicables para varios miles. Del mismo modo el tipo de red condiciona tremendamente el rendimiento del sistema, y puede que lo que funcione para un tipo de red, para otro requiera un nuevo diseño.

  La escalabilidad propone que cualquier computadora individual ha de ser capaz de trabajar independientemente como un sistema distribuido, pero también debe poder hacerlo conectado a muchas otras máquinas.

- ***Fiabilidad***
  Una de las ventajas claras que nos ofrece la idea de sistema distribuido es que el funcionamiento de todo el sistema no debe estar ligado a ciertas máquinas de la red, sino que cualquier equipo pueda suplir a otro en caso de que uno se estropee o falle.

  La forma más evidente de lograr la fiabilidad de todo el sistema está en la redundancia. La información no debe estar almacenada en un solo servidor de archivos, sino en por lo menos dos máquinas. Mediante la redundancia de los principales archivos o de todos evitamos el caso de que el fallo de un servidor bloquee todo el sistema, al tener una copia idéntica de los archivos en otro equipo.

  Otro tipo de redundancia más compleja se refiere a los procesos. Las tareas críticas podrían enviarse a varios procesadores independientes, de forma que el primer procesador realizaría la tarea normalmente, pero ésta pasaría a ejecutarse en otro procesador si el primero hubiera fallado.

- ***Comunicación***
  La comunicación entre procesos en sistemas con un único procesador se lleva a cabo mediante el uso de memoria compartida entre los procesos. En los sistemas distribuidos, al no haber conexión física entre las distintas memorias de los equipos, la comunicación se realiza mediante la transferencia de mensajes.

A continuación presentamos algunas ventajas y desventajas que presentan los sistemas distribuidos:

- *Ventajas*
  - Procesadores más poderosos y a menos costos
    - Desarrollo de estaciones con más capacidades
    - Las estaciones satisfacen las necesidades de los usuarios.
    - Uso de nuevas interfaces.
  - Avances en la tecnología de comunicaciones.
    - Disponibilidad de elementos de Comunicación.

  - Desarrollo de nuevas técnicas.
- Compartición de recursos.
  - Dispositivos (Hardware).
  - Programas (Software).
- Eficiencia y Flexibilidad.
  - Respuesta rápida.
  - Ejecución concurrente de procesos.
  - Empleo de técnicas de procesamiento distribuido.
- Disponibilidad y Confiabilidad.
  - Sistema poco propenso a fallas.
  - Mayores servicios que elevan la funcionalidad.
- Crecimiento Modular.
  - Es inherente al crecimiento.
  - Inclusión rápida de nuevos recursos.
  - Los recursos actuales no afectan.

• *Desventajas*
  - Requerimientos de mayores controles de procesamiento.
  - Velocidad de propagación de información muy lenta a veces.
  - Servicios de replicación de datos y servicios con posibilidades de fallas.
  - Mayores controles de acceso y proceso.
  - Administración más compleja.
  - Costo más alto.

### *1.2.1. Hardware en los Sistemas Distribuidos*

Aun cuando todos los sistemas distribuidos están compuestos de múltiples CPU's, existen diferentes formas en que el hardware se encuentra organizado, particularmente en la manera en que están interconectados y cómo se combinan.

Varios esquemas de clasificación para describir sistemas de cómputo con múltiples CPU's han sido propuestos a través de los años, pero sólo consideraremos a los sistemas construidos a partir de computadoras independientes y están divididos en dos grupos: aquellas que tienen memoria compartida usualmente llamadas *multiprocesadores* o de *mulprocesamiento* y aquellas que no implementan memoria compartida y que son llamadas con frecuencia *multicomputadoras*.

♦ ***Multiprocesadores***[18]
  Tienen memoria compartida, es decir, tienen un espacio de direcciones virtuales compartido por todos los procesadores, de manera que si se realiza una modificación en una localización, ésta es vista desde los demás procesadores.

♦ ***Multicomputadoras***[19]
  Una multicomputadora es una máquina con multiprocesadores de memoria distribuida en la cual los procesadores y la red son físicamente cercanos uno del otro (comúnmente en el

[18] Ver subtema *1.1.1.3.1 Sistemas de Memoria Compartida.*
[19] Ver subtema *1.1.1.3.2 Sistemas de Memoria Distribuida.*

mismo gabinete). Por esta razón una multicomputadora también es conocida como una *máquina fuertemente acoplada*[20]. Cada procesador tiene su propia memoria, de manera que modificaciones en una localización sólo es vista por aquel procesador que tiene acceso a esa memoria en particular. Una multicomputadora es usada por una o al menos unas cuantas aplicaciones al mismo tiempo y cada aplicación usa un conjunto de procesadores dedicados.

### *1.2.2. Clasificación de los Sistemas Distribuidos*

Los sistemas distribuidos son una de las grandes áreas de las Ciencias de la Computación, la cual se dedica al estudio y solución de los problemas que se presentan en el desarrollo de sistemas de cómputo distribuidos. Esta área se divide en las siguientes subareas:

- Bases de datos distribuidas.
- Sistemas operativos distribuidos y de red.
- Sistemas cliente-servidor.
- Sistemas multimedia distribuidos.
- Cómputo paralelo.
- Sistemas de tiempo real distribuidos.
- Sistemas de control distribuido.

A partir de esta clasificación de subáreas de los sistemas distribuidos, se pueden crear varios modelos de computadoras mediante los cuales se puede realizar el desarrollo de sistemas de cómputo distribuido.

#### *1.2.2.1. Modelo de Minicomputadoras*

Varias computadoras soportan diferentes usuarios sobre cada una y provee acceso a recursos remotos. Por lo menos debe haber un usuario por cada computadora.

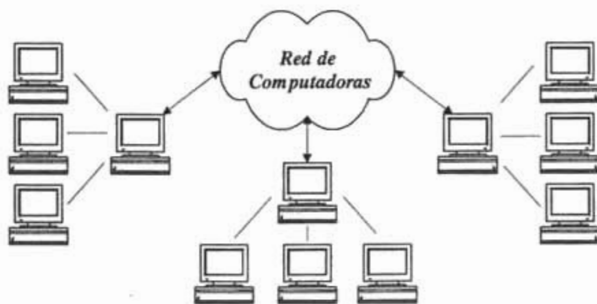


*Fig. 1.17 Modelo de Minicomputadoras*

[20] Los procesadores se comunican a través de una memoria principal y compartida, de esta manera la razón de la velocidad a la cual los procesadores pueden comunicarse de un procesador a otro es del orden del ancho de banda de la memoria. Se tolera un alto grado de interacciones entre las tareas sin un significativo deterioro de su desempeño.

### *1.2.2.2. Modelo de Estaciones de Trabajo*

Varias estaciones, por lo general cientos, donde cada usuario cuenta con una estación de trabajo y realiza en ella todo su trabajo. Requiere de un sistema operativo que soporte funciones de acceso y control remoto. Un microprocesador por cada usuario, por cada computadora. Las estaciones de trabajo cuentan con interfaces gráficas, CPU potentes y memorias propias.

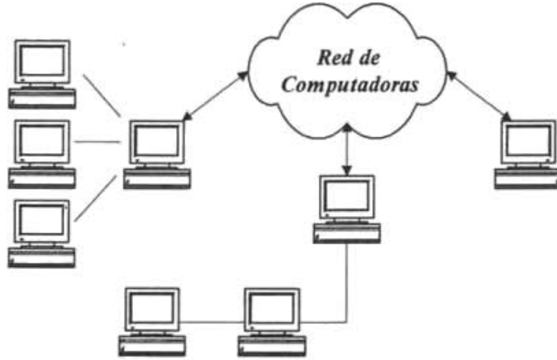


*Fig. 1.18 Modelo de Estación de Trabajo*

### *1.2.2.3. Modelo de Microprocesadores de Pooling*

Trata de utilizar uno o más microprocesadores dependiendo de las necesidades de los usuarios. Primero los procesadores completan su tarea y posteriormente regresan a esperar una nueva asignación. El número de microprocesadores normalmente es mayor a uno por cada usuario.

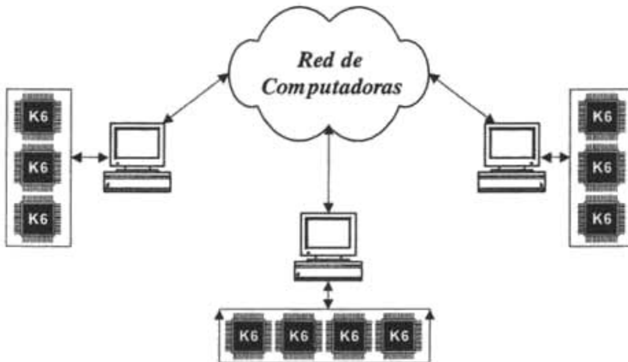


*Fig. 1.19 Modelo de Microprocesadores en Pooling*

### *1.2.3. Modelos de procesamiento distribuido*

A partir de esta clasificación de subáreas mostradas en el subtema anterior sobre los sistemas distribuidos, se pueden realizar varios tipos de procesamiento de acuerdo a las necesidades de los sistemas.

#### *1.2.3.1. Procesamiento distribuido basado en la entrada y salida*

- Comunicarse con un proceso remoto es similar a leer o escribir a un archivo.
- La biblioteca de sockets utiliza este modelo.
- Enviar y recibir mensajes es realmente Entrada/Salida.
- Es un enfoque de nivel relativamente bajo.

#### *1.2.3.2. Procesamiento distribuido basado en llamadas a procedimientos remotos*

- Comunicarse con un proceso remoto es similar a invocar un procedimiento.
- El procedimiento invocado no reside en el proceso que invoca sino en otro proceso (posiblemente en otra máquina).
- Los procedimientos reciben parámetros y devuelven resultados.
- Es un enfoque de nivel más alto que el orientado a entrada/salida.
- Los detalles de enviar y recibir mensajes quedan ocultos al programador.

#### *1.2.3.3. Procesamiento distribuido basado en objetos distribuidos*

- Comunicarse con un proceso remoto es similar a invocar un método de un objeto.
- El objeto al cual se hace la solicitud no reside en el proceso que invoca el método sino en otro proceso (posiblemente en otra máquina).
- Los métodos en POO[21] reciben parámetros y devuelven resultados.
- Es un enfoque de nivel más alto que los anteriores.
- Los detalles de enviar y recibir mensajes quedan ocultos al programador.

#### *1.2.3.4. Procesamiento distribuido basado en memoria compartida*

- Comunicarse con un proceso consiste en leer y escribir datos de una memoria común.
- El sistema de comunicación subyacente se encarga de duplicar el bloque de memoria común en las diferentes computadoras que forman parte del sistema.

### *1.2.4. Comunicación en los Sistemas Distribuidos*

La comunicación en los sistemas distribuidos se basa en la transferencia de mensajes. Si el proceso A quiere comunicarse con el proceso B, construye un mensaje en su propio espacio de direcciones; luego ejecuta una llamada al sistema para que éste tome el mensaje y lo envíe a través de la red hacia B. Para conseguir la comunicación, A y B deben entender exactamente lo mismo.

[21] Programación Orientada a Objetos.

La ISO (*International Standards Organization*) desarrolló el modelo de referencia para interconexión de sistemas abiertos (heterogéneos) o modelo OSI (*Open Systems Interconnection*).

El modelo de referencia OSI es la arquitectura de red actual más prominente. El OSI es un modelo de 7 capas, donde cada capa define los procedimientos y las reglas (protocolos normalizados) que los subsistemas de comunicaciones deben seguir, para poder comunicarse con sus procesos correspondientes de los otros sistemas. Esto permite que un proceso que se ejecuta en una computadora, pueda comunicarse con un proceso similar en otra computadora, si tienen implementados los mismos protocolos de comunicaciones de capas OSI.

Cada una de las 7 capas o niveles tiene funciones definidas, que se relacionan con las funciones de las capas siguientes. Los niveles inferiores se encargan de acceder al medio, mientras que los superiores, definen cómo las aplicaciones acceden a los protocolos de comunicación.

| | *Programa de aplicación del usuario* | |
|---|---|---|
| *Nivel 7* | ***Aplicación*** | *Provee servicios generales sobre aplicaciones* |
| *Nivel 6* | ***Presentación*** | *Formato de datos* |
| *Nivel 5* | ***Sesión*** | *Coordina la iteración en la sesión con los usuarios* |
| *Nivel 4* | ***Transporte*** | *Provee una transmisión de datos* |
| *Nivel 3* | ***Red*** | *Enruta unidades de información* |
| *Nivel 2* | ***Enlace*** | *Intercambio de datos entre dispositivos* |
| *Nivel 1* | ***Física*** | *Transmite un flujo de bits sobre un medio físico* |

*Fig. 1.20 Capas del Modelo OSI*

Las características generales de las capas son las siguientes:

- Una capa se creará en situaciones en donde se necesite un nivel de diferencia abstracto.
- Cada capa deberá efectuar una función bien definida.
- La función que realizará cada capa, deberá seleccionarse con la intención de definir protocolos normalizados internacionalmente.
- Los límites de las capas, deberán seleccionarse tomando en cuenta la minimización del flujo de información, a través de las interfaces.
- El número de capas debe ser lo suficientemente grande, para que cada una no realice más de una función y lo suficientemente pequeña para que la arquitectura sea manejable.
- Los servicios proporcionados por cada nivel son utilizados por el nivel superior.
- Existe una comunicación virtual entre 2 mismas capas, de manera horizontal.
- Existe una comunicación vertical entre una capa de nivel N y la capa de nivel N+1.
- La comunicación física se lleva a cabo entre las capas de nivel 1.

- ***Capa Física***

  La capa física define cómo un medio de transmisión se conecta a una computadora y cómo se transfiere la información eléctrica por este medio. Esta capa relaciona la

agrupación de circuitos físicos a través de los cuales los bits[22] son movidos y que encierran las características físicas, eléctricas funcionales y procedimentales, para el envío y recepción de bits.

- Define las características físicas (componentes y conectores mecánicos).
- Define las características eléctricas (niveles de tensión).
- Define las características funcionales de la interfaz (establecimiento, mantenimiento y liberación del enlace físico).
- Solamente reconoce bits individuales, no reconoce caracteres ni tramas multicarácter. Por ejemplo RS-232 y RS-449.
- Transmisión de flujo de bits a través del medio. No existe estructura alguna.
- Maneja voltajes y pulsos eléctricos.
- Especifica cables, conectores y componentes de interfaz con el medio de transmisión.

Interfaces de nivel físico para comunicaciones incluyen EIA RS – 232.

- ***Capa de Enlace***

  Este nivel se relaciona con el envío de bloque de datos sobre una comunicación física, determina el principio y el fin de un bloque de datos transmitido, detecta errores de transmisión, controla muchas máquinas que comparten un circuito físico para que sus transmisiones no sufran mezclas, direcciona un mensaje a una máquina entre varias.

- Detección y control de errores.
- Control de secuencia.
- Control de flujo.
- Control de enlace lógico.
- Control de acceso al medio.
- Sincronización de la trama.
- Estructura el flujo de bits bajo un formato predefinido llamado trama.
- Para formar una trama, el nivel de enlace agrega una secuencia especial de bits al principio y al final del flujo inicial de bits.
- Transfiere tramas de una forma confiable libre de errores (utiliza reconocimientos y retransmisión de tramas).
- Provee control de flujo.

A continuación se presenta una lista de protocolos que ocupan este nivel:

- Control de enlace de datos de alto nivel (High-level Data Link Control HDLC).
- Manejadores y métodos de acceso de LAN, como Ethernet o Token Ring.
- ATM para redes de área extensa WAN de transmisión rápida.
- Network Driver Interface Specification (NDIS) de Microsoft.
- Open Datalink Interface (NODI) de Novell.

[22] Unidad mínima de información utilizada por un equipo. Un bit expresa un 1 o un 0 en un numeral binario, o una condición lógica verdadera o falsa.

- ***Capa de Red***

  Esta capa relaciona los circuitos virtuales, estos circuitos son imaginarios y aunque no existen figuran e interactúan con los niveles más altos y dan la impresión de existencia en este nivel están los procedimientos de interfaces estándar para el circuito virtual y los mecanismos complejos de operación están ocultos a los niveles más altos de software tanto como sea posible. En esta capa se determina el establecimiento de la ruta.

  - Esta capa mira las direcciones del paquete para determinar los métodos de conmutación y enrutamiento
  - Realiza control de congestión.
  - Divide los mensajes de la capa de transporte en paquetes y los ensambla al final.
  - Utiliza el nivel de enlace para el envío de paquetes: un paquete es encapsulado en una trama.
  - Enrutamiento de paquetes.
  - Envía los paquetes de nodo a nodo usando ya sea un circuito virtual o datagramas.
  - Control de Congestión.

  A continuación se muestra una lista de protocolos que ocupan este nivel:

  - Protocolo de Internet (IP).
  - Protocolo X.25.
  - Internetwork Packet Exchange (IPX) de Novell.
  - VINES Intenet Protocol (VIP).

- ***Capa de Transporte***

  Controla la interacción entre procesos usuarios, incluye controles de integración entre usuarios de la red para prevenir pérdidas o doble procesamiento de transmisiones, controla el flujo de transacciones y direccionamiento de máquinas a procesos de usuario. Esta capa asegura que se reciban todos los datos y en el orden adecuado. Realiza un control de extremo a extremo. Algunas de las funciones realizadas son:

  - Acepta los datos del nivel de sesión, fragmentándolos en unidades más pequeñas en caso necesario y los pasa al nivel de red o viceversa.
  - Multiplexaje.
  - Regula el control de flujo del tráfico de extremo a extremo.
  - Reconoce los paquetes duplicados.
  - Establece conexiones punto a punto sin errores para el envío de mensajes.
  - Permite multiplexar una conexión punto a punto entre diferentes procesos del usuario (puntos extremos de una conexión).
  - Provee la función de difusión de mensajes a múltiples destinos.
  - Control de Flujo.

  Los siguientes protocolos pueden estar en este nivel:

  - Internet Transport Protocol (TCP).
  - Internet User Datagram Protocol (UDP).

- Sequenced Packed Exchange (SPX).
- NetBios/NetBEUI.

◆ ***Capa de Sesión***

Este nivel estandariza el proceso de establecimiento y terminación de una sesión, si por algún motivo esta sesión falla éste restaura la sesión sin pérdida de datos o si esto no es posible termina la sesión de una manera ordenada chequeando y recuperando todas sus funciones. Establece las reglas o protocolos para el diálogo entre máquinas y así poder regular quién habla y por cuánto tiempo o si hablan en forma alterna es decir las reglas del diálogo que son acordadas. Provee mecanismos para organizar y estructurar diálogos entre procesos de aplicación. Actúa como un elemento moderador capaz de coordinar y controlar el intercambio de los datos. Controla la integridad y el flujo de los datos en ambos sentidos. Algunas de las funciones que realiza son las siguientes:

- Establecimiento de la conexión de sesión.
- Intercambio de datos.
- Liberación de la conexión de sesión.
- Sincronización de la sesión.
- Administración de la sesión.
- Permite a usuarios en diferentes máquinas establecer una sesión.
- Una sesión puede ser usada para efectuar un login a un sistema de tiempo compartido remoto, para transferir un archivo entre 2 máquinas, etc.
- Controla el diálogo.
- Función de sincronización.

◆ ***Capa de Presentación***

Sus funciones están relacionadas con el conjunto de caracteres o códigos de datos que son usados, o la manera como van a ser presentados en pantalla o como van a ser impresos, cuando un conjunto de caracteres llega a una pantalla, se dan ciertas acciones para una presentación buena de la información. Esta capa también tiene que ver con el conjunto de caracteres que debe presentar una edición de datos, salto de línea, colocación de datos en columnas, adición de encabezados fijos para las columnas, etc. En esta capa se realizan las siguientes funciones:

- Se da formato a la información para visualizarla o imprimirla.
- Se interpretan los códigos que estén en los datos (conversión de código).
- Se gestiona la encripción de datos.
- Se realiza la compresión de datos.
- Establece una sintaxis y semántica de la información transmitida.
- Se define la estructura de los datos a transmitir (define los campos de un registro: nombre, dirección, teléfono, etc.).
- Define el código a usar para representar una cadena de caracteres (ASCII, EBCDIC, etc).
- Compresión de datos.
- Criptografía.

- ***Capa de Aplicación***
  Relacionado con las funciones de más alto nivel que proporcionan soporte a las aplicaciones o actividades del sistema. Por ejemplo, control de transferencia de archivos, soporte al operador funciones de diálogo de alto nivel, actividades de bases de datos de alto nivel. Los tres primeros niveles proporcionan una variedad de servicios que son empleados en la sesión de los usuarios, a este subconjunto se le denomina subsistema de la sesión de servicios. Se definen una serie de aplicaciones para la comunicación entre distintos sistemas, las cuales gestionan:

  - Transferencia de archivos (FTP).
  - Intercambio de mensajes (correo electrónico).
  - Login remoto (rlogin, telnet).
  - Acceso a bases de datos, etc.

  A continuación se listan algunos protocolos utilizados en este nivel:

  - Terminal Virtual
  - File Transfer Access and Management (FTAM)
  - Distributed Transaction Processing (DTP)

La comunicación según el modelo OSI siempre se realizará entre dos sistemas. Supongamos que la información se genera en el nivel 7 de uno de ellos, y desciende por el resto de los niveles hasta llegar al nivel 1, que es el correspondiente al medio de transmisión (por ejemplo el cable de red) y llega hasta el nivel 1 del otro sistema, donde va ascendiendo hasta alcanzar el nivel 7. En este proceso, cada uno de los niveles va añadiendo a los datos a transmitir la información de control relativa a su nivel, de forma que los datos originales van siendo recubiertos por capas datos de control.

De forma análoga, al ser recibido dicho paquete en el otro sistema, según va ascendiendo del nivel 1 al 7, va dejando en cada nivel los datos añadidos por el nivel equivalente del otro sistema, hasta quedar únicamente los datos transmitidos. La forma, pues de enviar información en el modelo OSI tiene una cierta similitud con enviar un paquete de regalo a una persona, donde se ponen una serie de papeles de envoltorio, una o más cajas, hasta llegar al regalo en sí.

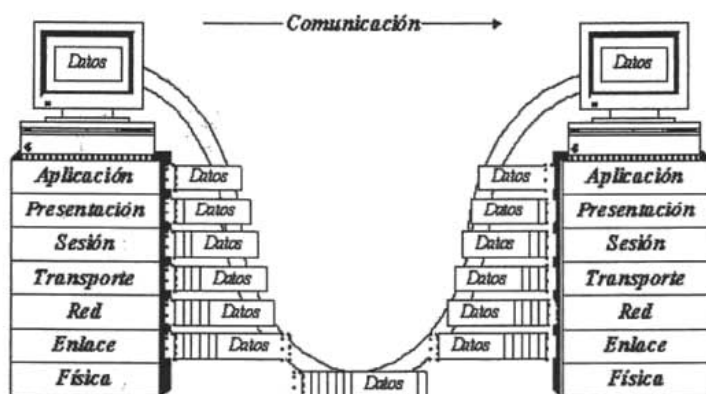


*Fig. 1.21 Comunicación según el modelo OSI*

Los niveles OSI se entienden entre ellos, es decir, el nivel 5 enviará información al nivel 5 del otro sistema (lógicamente, para alcanzar el nivel 5 del otro sistema debe recorrer los niveles 4 al 1 de su propio sistema y el 1 al 4 del otro), de manera que la comunicación siempre se establece entre niveles iguales, a las normas de comunicación entre niveles iguales es a lo que llamaremos protocolos. Este mecanismo asegura la modularidad del conjunto, ya que cada nivel es independiente de las funciones del resto, lo cual garantiza que a la hora de modificar las funciones de un determinado nivel no sea necesario reescribir todo el conjunto.

## 1.3. CÓMPUTO PARALELO/DISTRIBUIDO EN CLUSTERS

El término Cluster se aplica no sólo a computadoras de alto rendimiento sino también a los conjuntos de computadoras, construidos utilizando componentes de hardware comunes y software libre.

Los Clusters juegan hoy en día un papel muy importante en la solución de problemas de las ciencias, las ingenierías y en el desarrollo y ejecución de muchas aplicaciones comerciales. Los Clusters han evolucionado para apoyar actividades en aplicaciones que van desde supercómputo hasta el software adaptado a misiones críticas, pasando por los servidores web, el comercio electrónico y las bases de datos de alto rendimiento.

La computación basada en Clusters surge gracias a la disponibilidad de microprocesadores de alto rendimiento más económicos y de redes de alta velocidad, y también gracias al desarrollo de herramientas de software para cómputo distribuido de alto rendimiento; todo ello frente a la creciente necesidad de potencia de cómputo para aplicaciones en las ciencias y en el ámbito comercial, así como de disponibilidad permanente para algunos servicios. Por otro lado, la evolución y estabilidad que ha alcanzado el sistema operativo GNU/Linux, ha contribuido de forma importante al desarrollo de muchas tecnologías nuevas.

Según la aplicabilidad de los Clusters, se han desarrollado diferentes líneas tecnológicas. La primera surge frente a la necesidad de supercómputo para determinadas aplicaciones, lo que se persigue es conseguir que un gran número de máquinas individuales actúen como una sola máquina muy potente. A este tipo de Cluster se le conoce como *Clusters de alto desempeño.* Este tipo de Clusters se aplica mejor en problemas grandes y complejos que requieren una cantidad enorme de potencia computacional, como el pronóstico numérico del estado del tiempo, astronomía, investigación en criptografía, simulación militar, simulación de recombinaciones entre moléculas naturales y el *análisis de imágenes.*

Un segundo tipo de tecnología de Clusters, es el destinado al balanceo de carga. Surge el concepto de *Cluster de servidores virtuales*, Cluster que permite que un conjunto de servidores de red compartan la carga de trabajo y de tráfico de sus clientes, aunque aparezcan para estos clientes como un único servidor. Al balancear la carga de trabajo en un conjunto de servidores, se mejora el tiempo de acceso y la confiabilidad. Además como es un conjunto de servidores el que atiende el trabajo, la caída de uno de ellos no ocasiona una caída total del sistema. Este tipo de servicio es de gran valor para compañías que trabajan con grandes volúmenes de tráfico y trabajo en sus webs y servidores de correo.

El último tipo importante de tecnología de Clusters trata del mantenimiento de servidores que actúen entre ellos como respaldos de la información que sirven. Este tipo de Clusters se conoce como *Clusters de alta disponibilidad* o *Clusters de redundancia.* La flexibilidad y robustez que proporcionan este tipo de Clusters, los hacen necesarios en ambientes de intercambio masivo de información, almacenamiento de datos sensibles y allí donde sea necesaria una disponibilidad continua del servicio ofrecido.

Los *Clusters de alta disponibilidad* permiten un fácil mantenimiento de servidores. Una máquina de un Cluster de servidores se puede sacar de línea, apagarse y actualizarse o repararse sin comprometer los servicios que brinda el Cluster. Cuando el servidor vuelva a estar listo, se reincorporará y volverá a formar parte del Cluster.

Además del concepto de Cluster, existe otro concepto más amplio y general que es el Cómputo en Malla (*Grid Computing*). Una Malla, dentro de una NOW, es un tipo de sistema paralelo y distribuido que permite compartir, seleccionar y añadir recursos que se encuentran distribuidos a lo largo de dominios administrativos *múltiples.* Si los recursos distribuidos se encuentran bajo la administración de un sistema central único de programación de tareas, entonces nos referiremos a un Cluster. En un Cluster, todos los nodos trabajan en cooperación con un objetivo y una meta común y la asignación de recursos la lleva a cabo un solo administrador centralizado y global. En una Malla, cada nodo tiene su propio administrador de recursos y política de asignación.

- *Ventajas*
  - Cada uno de los nodos es una máquina completa que se puede utilizar como una estación de trabajo normal, al mismo tiempo se empleará como una máquina paralela.
  - Debido a que puede estar formado por PC's normales de gama media o incluso baja (Pentium, Pentium 2 ó 3), su precio es relativamente bajo (causado por el gran mercado que hay actualmente).
  - Si no se van a emplear cada uno de los nodos como estación de trabajo, solo bastará con un teclado y una consola para todo el sistema, pudiéndonos evitar la compra

incluso de la tarjeta de vídeo ya que el sistema operativo Linux puede ejecutarse sin todos estos elementos.
- La construcción de un supercomputadora de *n* procesadores[23] es complejo, caro y en muchos casos irrealizable. Sin embargo, crear una red con 16, 100 o incluso más nodos es, en comparación, algo trivial.
- Cualquier problema en una máquina de este tipo es relativamente sencillo de resolver, ya que si por ejemplo se estropea un procesador en un nodo, nos bastará con sustituirlo por otro para resolverlo ya que es infinitamente más sencillo encontrar un nuevo procesador.
- Las estaciones de trabajo están ganando en potencial de procesamiento. Su potencial se ha incrementado drásticamente en los últimos años, y se dobla cada 18-24 meses. Y todo indica a que esta tendencia continuará por varios años.
- El ancho de banda[24] de la comunicación entre estaciones de trabajo se está incrementando y el tiempo de latencia se decrementa según se implementan las nuevas redes y protocolos en LAN.
- Los Clusters de estaciones de trabajo son más fáciles de integrar en redes existentes que computadoras paralelas específicas. Usualmente los usuarios de estaciones de trabajo no aprovechan toda la potencia de cómputo que éstas les ofrecen.
- Las herramientas de desarrollo para estaciones de trabajo están más *maduras* en comparación con las herramientas de desarrollo para computadoras paralelas, esto se debe principalmente a la naturaleza no estándar de los distintos sistemas paralelos.
- Los Clusters de estaciones de trabajo son baratos y son una alternativa lista y disponible a las plataformas especializadas de computación paralela.
- Los Clusters son fácilmente escalables. La capacidad de los nodos se puede ampliar de forma sencilla, añadiendo memoria o procesadores adicionales.

• *Desventajas*
- Las redes de computadoras no están diseñadas para el procesamiento en paralelo (al menos no en principio), siendo su ancho de banda muchas veces demasiado bajo y su latencia demasiado alta al menos en comparación con máquinas SMP.
- No hay mucho software que sea capaz de tratar un Cluster como si fuese un único sistema.

### *1.3.1. Arquitectura*

Un Cluster es un tipo de sistema de procesamiento paralelo o distribuido, que consiste en una colección de PC's autónomas interconectados trabajando unidos como un solo recurso de computación.

Un nodo del sistema puede ser un sistema con uno o más procesadores con memoria, recursos de Entrada/Salida y un sistema operativo. Un Cluster en general se refiere a dos o más computadoras

[23] Se toma el valor de *n* superior de 4.
[24] El ancho de banda es la máxima cantidad de datos que pueden ser transmitidos en una unidad de tiempo una vez que la transmisión de datos ha comenzado. Un buen ancho de banda puede transmitir grandes bloques de datos entre procesadores.

conectadas juntas. Estos nodos pueden coexistir en una misma caja, o bien estar físicamente separados y conectados a través de una LAN.

Un Cluster de computadoras interconectadas mediante una LAN puede parecer como un sistema único a los usuarios y las aplicaciones. Un sistema como éste ofrece un mecanismo con buena relación costo-rendimiento que permite obtener características que históricamente sólo estaban disponibles en sistemas muy caros.

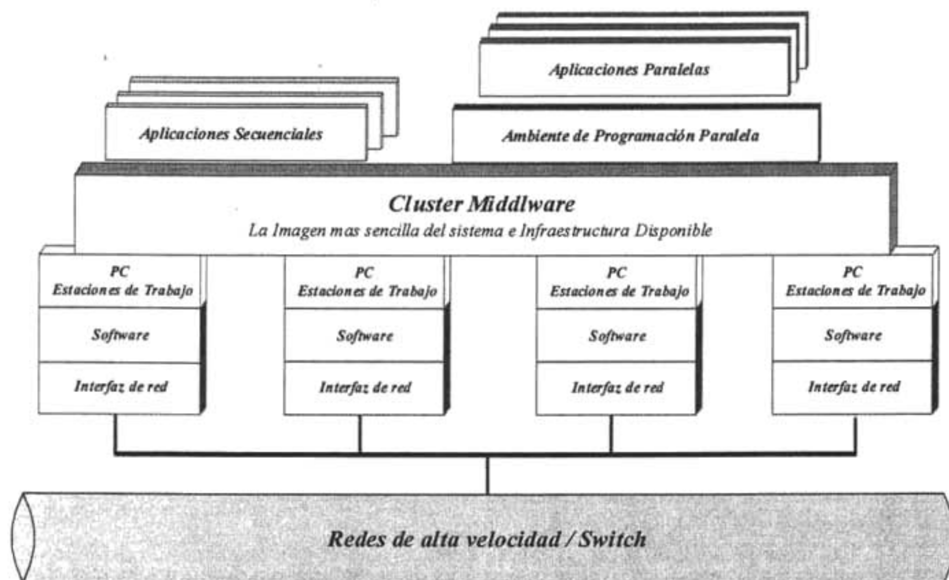


*Fig. 1.22 Arquitectura de un Cluster*

Los siguientes componentes generales conforman una computadora tipo Cluster:

- Múltiples computadoras de alto desempeño (PC's y SMP's).
- Sistemas operativos hechos casi en estado de arte [State of the art OS], basados en varios niveles o basados en Micro Kernel.
- Redes/Switches de alto desempeño, como son Gigabit Ethernet y Myrinet.
- Tarjetas de interfaz de red (NIC).
- Protocolos y servicios de rápida comunicación, como son Mensajes Activos y Rápidos (*Active and Fast Messages*).
- Cluster Middleware, Single System Image (SSI) y System Availability Infrastructure[25] (SAI).
    - Hardware, como son el Canal de Memoria (DEC) Digital, hardware DSM y técnicas SMP.
    - Kernel del sistema operativo o niveles adheridos, como son Solares MC y GLUnix.

[25] Esta capa proporciona al cluster servicios como: Checkpoints y recuperación de fallos y tolerancia a fallos.

  - Aplicaciones y subsistemas.
    - Aplicaciones.
    - Sistemas en tiempo de ejecución.
    - Manejo de recursos y software de calendarización.
- Ambientes y herramientas de programación paralela, como son los compiladores, PVM (*Parallel Virtual Machine*) y MPI (*Message Passing Interface*).
- Aplicaciones.
  - Secuencial.
  - Paralelos o distribuidos.

El hardware de interfaz de red actúa como un procesador de comunicaciones y es el responsable de transmitir y recibir los paquetes de datos entre los nodos del Cluster vía una Red/Switch.

El software de comunicación ofrece una significativa rápida y confiable comunicación de datos entre los nodos del Cluster y el mundo exterior, con una Red/Switch especial, como Myrinet, usan protocolos de comunicación como son mensajes activos para una más rápida comunicación entre los nodos. El sistema operativo cuenta con potenciales carreteras de comunicación y esto remueve los sobre picos que vienen directamente del acceso del nivel de usuarios hacia la interfaz de red.

Los nodos del Cluster pueden trabajar colectivamente, como un recurso integrado de cómputo, pueden operar como computadoras individuales. El Middleware del Cluster es el responsable de ofrecer la ilusión de una imagen de un sistema unificado (la imagen de un solo sistema) y la disponibilidad de la colección o las independencias pero interconectadas computadoras.

Los ambientes de programación pueden ofrecer herramientas portables, eficientes y de fácil uso para el desarrollo de aplicaciones. Esto incluye la librería de paso de mensaje, depuraciones y perfiladores (*profilers*). No se debe olvidar que los Clusters deben ser usados para le ejecución de aplicaciones secuenciales o paralelas.

### *1.3.2. Clasificación*

Los Clusters ofrecen las siguientes características a un relativo bajo costo:

- Alto desempeño.
- Expansibilidad y estabilidad.
- Alta productividad.
- Alta disponibilidad.

La tecnología de Clusters permite a las organizaciones impulsar su poder de procesamiento usando tecnologías estándar (componentes de conveniencia de hardware y software) que pueden ser adquiridos o comprados a relativo bajo costo. Esto provee expansibilidad y da una manera de actualización que permite a las organizaciones incrementar su poder de cómputo mientras preserven su inversión y sin incurrir en gastos extras.

El desempeño de las aplicaciones mejora con el soporte de un ambiente de software escalable. Otro beneficio de los Clusters es la capacidad de soporte a fallas que permite que otra computadora tome las tareas de otra que ha fallado en el Cluster.

Los Cluster son clasificados en diferentes categorías basados en varios factores que son indicados a continuación:

- ***Objetivo de la aplicación***
  Ciencias computacionales o aplicaciones de misión crítica.

  - Clusters de alto rendimiento (*High Performance*).
  - Clusters de alta disponibilidad (*High Availability*).

- ***Uso de cada nodo***
  Los nodos pertenecen a un Cluster individual o a uno dedicado.

  - Clusters dedicados.
  - Clusters no dedicados.

  En los primeros, se utilizan todos los recursos para la computación paralela, mientras que en los segundos, cada computadora ejecuta sus propios programas y el Cluster roba los ciclos en los que cada nodo no hace nada y los utiliza para el procesamiento paralelo.

  La distinción entre estos dos tipos de Clusters está basada en la propiedad que se tiene de los nodos. En el caso de los Clusters dedicados un individuo en particular no es propietario de las estaciones de trabajo; los recursos son compartidos para que el cómputo paralelo pueda ser desarrollado en el Clusters completo. La alternativa no dedicada es cuando hay individuos que son propietarios de las estaciones de trabajo y las aplicaciones son ejecutadas en los ciclos libres de CPU. La motivación para este hecho de que la mayoría de los CPU's de las estaciones de trabajo son utilizados totalmente algunas veces durante ciertas horas. El cómputo paralelo que está en un conjunto que cambia dinámicamente de estaciones de trabajo no dedicadas se le llama *cómputo paralelo adaptivo*.

  En los Clusters no dedicados, existe una tensión entre los propietarios de las estaciones de trabajo y los usuarios remotos que necesitan correr sus aplicaciones en las máquinas. El primero espera una respuesta interactiva y rápida de la aplicación que se ejecuta en su estación, mientras que el segundo sólo puede esperar que el primero desocupe un poco de tiempo del CPU para que pueda ejecutarse su aplicación remota.

- ***Hardware de cada nodo***

  - Clusters de PC's (CoPs) o Pilas de PC's (PoPs).
  - Clusters de estaciones de trabajo (COW's)
  - Clusters de SMP's (CLUMP's)

- ***Sistema Operativo de los nodos***

  - Clusters basados en Linux, por ejemplo Beowulf.
  - Clusters basados en Solaris, por ejemplo Berkeley NOW.
  - Clusters basados en NT, por ejemplo HPVM.
  - Clusters basados en AIX, por ejemplo IBM SP2.
  - Clusters Digitales VMS.
  - Clusters basados en HP-UX.
  - Clusters Microsoft Wolfpack.

- ***Configuración de los nodos***
  Arquitectura del nodo y el tipo de sistema operativo que tiene.

  - Clusters Homogéneos.- Todos los nodos tienen una arquitectura similar y corren bajo el mismo sistema operativo.
  - Clusters Heterogéneos.- Todos los nodos tienen diferentes arquitecturas y corren bajo diferentes sistemas operativos.

- ***Niveles de agrupamiento (clustering)***
  Basados en la localización de los nodos y su número.

  - Grupo de Clusters (de 2 a 99 nodos).- Los nodos son conectados por SAN's[26] conocidas como Myrinet y están apiladas en un frame o que existe en un centro.
  - Clusters Departamentales (de 10 a 100 nodos).
  - Clusters Organizacionales (más de 100 nodos).
  - Metacomputadoras nacionales (WAN/Internet) (gran cantidad de nodos en comparación con los dos anteriores).
  - Metacomputadoras internaciones (Internet) (de 1000 nodos a muchos millones).

Los Clusters pueden ser interconectados para formar grandes sistemas (Clusters de Clusters), y de hecho, la Internet puede ser usada como un Clusters de computadoras.

El uso de los recursos de una red amplia para cómputo de alto desempeño ha conducido al surgimiento de un nuevo campo llamado *Metacomputación*.

### 1.3.3. *Componentes de Hardware*

La clave para integrar un Cluster de componentes comerciales, fáciles de adquirir y a un costo considerable, es tener:

- Gran desempeño en los nodos
- Cómputo y una interconexión de red dedicada para proveer comunicación de datos entre los nodos.

[26] SAN = System *Area Network*, Red de Área del Sistema.

#### 1.3.3.1. *Hardware de nodos*

El nodo de un Cluster provee un sistema de cómputo y la capacidad almacenamiento. A diferencia de nodos que tienen un sistema completamente integrado como las MPP que es derivada de subsistemas de cómputo operacional completamente autosuficientes comúnmente comercializadas como sistemas servidor o desktop.

Los sistemas o componentes que deben integrar un nodo son los siguientes:

- ***Procesador***
  Actualmente la unidad central de proceso es un complejo subsistema que incluye, la esencia de un elemento de procesamiento, dos niveles de caché y controladores de bus externos. Hay procesadores de 32 y 64 bits, ambos disponibles en arquitecturas populares como la familia Intel Pentium Pro, Pentium II/III y la Compaq Alpha 21264. Otros incluyendo la IBM PowerPC, la Sun Microsystem Super Sparc III y AMD K7 Athlon.

  Los procesadores Intel son los de uso común en las computadoras basadas en PC's. Incluyen los Pentium Pro y II/III. Estos procesadores, aun cuando no se consideran como procesadores de gran desempeño como los actuales, alcanzan el desempeño de una estación de trabajo mediana. En operaciones enteras el Pentium Pro es mejor que la Sun Ultra Sparc, pero no lo es en operaciones de punto flotante.

  Actualmente tenemos los procesadores Intel Pentium III y IV con mayores velocidades, Pentium IV arriba de los 2.4 GHz, lo que nos permite un mayor desempeño.

  Los Pentium II Xerox y Pentium II usan un bus de memoria de 100 MHz, pueden tener de 512 KB a 2 MB de caché L2. El Xeon puede soportar buses PCI de 64 bits que pueden ser interconectados con una red de Giga bit.

  Otros procesadores populares son AMD x86, Cyrix x86, Digital Alpha, IBM PowerPC, Sun Sparc, SGI MIPS y HP PA.

- ***Memoria***
  Originalmente la memoria que se encontraba en la computadora era de 64 Kbits. Hoy una PC puede contener entre 128, 256, 512 Mbytes o más, instalados en los slots de la computadora de tipo SIMM (*Standar Industry Memory Module*) o DIMM.

  El sistema de una computadora puede usar varios tipos de memoria entre las que se incluye EDO (*Extended Data Out*) y *fase page*. EDO permite empezar el próximo acceso mientras los datos previos están siendo leídos, y *fase page* permite que los múltiples accesos adyacentes sean realizados más eficientemente.

  La cantidad de memoria que necesita un Cluster depende del objetivo de la aplicación a la que se enfoque. Los programas a ser paralelizados deberán estar distribuidos de manera que tanto la memoria como el procesamiento estén distribuidos entre los procesadores con fines de estabilidad. No es necesario tener RAM que contenga todo el problema en

memoria pero sí debería haber suficiente como para evitar el exceso de *swapping* (intercambio) de bloques de memoria (*page-misses*) a disco.

El acceso a DRAM (*Dynamic Random Acces Memory*) es extremadamente lento comparado con la velocidad del procesador, ocupa en magnitud más tiempo que el ciclo de reloj de un CPU. La memoria caché es utilizada para mantener los bloques de memoria recientemente usados para un acceso más rápido, en el caso de que el CPU se refiera de nuevo a alguna palabra que esté en el bloque.

La memoria caché permite tener accesos más cercanos a la velocidad del CPU, pero es extremadamente cara. El tamaño total de la memoria caché oscila entre 8 KB y 2 MB por procesador.

En los procesadores Pentium es común hallar un bus de memoria de 64 bits y el soporte de 2 MB caché externo. Esta mejora fue necesaria para aprovechar más los procesadores Pentium y hacer una arquitectura con memoria similar a la de las Workstation de UNIX.

La DRAM ha sido más comercial porque provee alta densidad con moderado tiempo de acceso a costos considerables.

La capacidad de memoria del nodo principal debe ser de 64 Mbytes a mayores de 1 Gbytes, entrados en los SIMM's, DIMM's o RIMM's para Pentium IV. Los chips pueden contener individualmente 64 Mbits y emplear algunos posibles protocolos de interfaz. SDRAM es más ampliamente usada para proveer un gran ancho de banda de memoria.

- ***Almacenamiento Secundario y E/S***

Las mejores en el tiempo de acceso al disco no mantienen un ritmo en el desempeño de los microprocesadores, los cuales han mejorado en un 50% o más por año. A pesar de que se ha incrementado la capacidad de los discos y las tecnologías, el tiempo de acceso al disco sigue siendo muy lento.

La ley de Amdahl indica que la velocidad de un sistema está limitada por la velocidad de su componente más lento y éste, en un sistema de computación, es la E/S.

Una forma de mejorar el desempeño de la E/S es llevando a cabo las operaciones de E/S en paralelo, esto es soportado por sistemas de archivos paralelos basados en *software* o *hardware* RAID. Dado que el hardware RAID es caro, el software RAID puede ser construido utilizando los discos asociados con cada estación de trabajo en el Cluster.

Los discos duros son la principal forma de almacenamiento secundario, los más populares son los SCSI II (*Small Computer Systems Interface*, Interfaz para Sistemas de Pequeñas Computadoras) y EIDE (*Enhanced Integrate Drive Electronics*, Electrónica de Unidad Integrada Mejorada). Los SCSI proveen mayor capacidad y un superior acceso de ancho de banda (*bandwidth*), mientras que la ventaja que tienen los EIDE es su bajo costo.

Las actuales generaciones de disco proveen entre 20 y 100 Gbytes de almacenamiento con un tiempo de acceso de milisegundos. Los CD-ROM y CD-RW han llegado a

considerarse otros medios de almacenamiento debido a su bajo costo, manejo de código y datos y apoyo de almacenamiento.

- ***Interfaz Externa (Buses del sistema)***
  Los canales de interfaz estándar nos permiten conectar a un nodo dispositivos externos y también conectarnos a una red. Existen los canales o ranuras más conocidas y usadas EISA, ISA y PCI, pero muchos nodos que todavía incluyan puertos EISA en las tarjetas madre son muy lentos.

  El primer bus usado en PC es el AT, ahora conocido como bus ISA, iba a 5 MHz y tenía un ancho de 8 bits.

  Las PC's son sistemas modulares en los que el procesador y la memoria se localizaban en una *motherboard* mientras que otros componentes se encuentran en tarjetas hijas que se conectan por el bus del sistema.

  El bus ISA luego se extendió a un ancho de 16 bits con un reloj de 13 MHz. Sin embargo esto no era suficiente para evitar el cuello de botella generado por la diferencia de velocidad con el CPU.

  En la actualidad todavía hay placas bases, incluso para procesadores de última generación cuyo diseño integra 2 ó 3 slots ISA. Aunque debido a la actualización de tarjetas de este tipo, tiende a desaparecer en breve, siendo sustituido por el bus PCI.

  Actualmente se tiene que el bus PCI desarrollado por Intel, es de 32 bits, posee 124 conectores, su frecuencia es de 33 MHz y su velocidad de transferencia es de 132 a 264 Mbytes por segundo. Existe una extensión de 64 bits que añade otros 60 contactos. Entre las características especiales se encuentra la configuración automática de tarjetas, lo que se conoce como PnP (*Plug and Play*). PCI también ha sido adoptado para plataformas no basadas en Intel.

  Actualmente el bus PCI es un estándar en las placas base, se extiende a ampliar el número de estas ranuras y a reducir el de ranuras ISA. El máximo de slots permitidos por la especificación del bus es de seis.

#### 1.3.3.2. *Hardware de red*

La construcción del Cluster es posible gracias a las tecnologías de red adecuadas que permiten la interconexión entre los nodos. Estas redes de interconexión interactuando con el *hardware* y *software* adecuados permiten que los paquetes de datos se transfieran entre los nodos o procesadores que conforman el Cluster.

En las máquinas paralelas un elemento importante a tener en cuenta es la interfaz que utilizan para la comunicación, y antes de adquirirse se debe pensar para qué se quiere el Cluster y el tráfico de datos que se espera tener. Una vez tenido en cuenta esto, se seleccionara aquella interfaz que se ajuste más a lo deseado dentro de lo posible.

| Tipo de Red | Ancho de banda | Tiempo de Latencia (ping) | Interfaz/Puerto |
|---|---|---|---|
| ATM | 155 [Mb/seg] | 120 [ms] | PCI |
| Ethernet | 10 [Mb/seg] | 100 [ms] | PCI |
| Fast Ethernet | 100 [Mb/seg] | 80 [ms] | PCI |
| Gigabit Ethernet | 1000 [Mb/seg] | 300 [ms] | PCI |
| Myrinet | ~2 [Gb/seg] | < 7 [µs] | PCI |
| Token Ring | 100 [Mb/seg] – 1 [Gb/seg] | | PCI |
| SCI | 5 [Gb/seg] | 2.7 [ms] | PCI |

*Tabla 1.2 Interfaz para comunicación*

La comunicación entre nodos de un Cluster utiliza redes de alta velocidad utilizando un protocolo estándar como lo es TCP/IP o un protocolo de un nivel más bajo como lo es *Active Messages* (Mensajes Activos). Lo más común y más fácil de implementar es una interconexión vía Ethernet. En términos de desempeño, latencia y ancho, esta tecnología muestra su edad. Sin embargo, Ethernet es más barata y es un camino más fácil para poder compartir archivos e impresoras. Una simple conexión Ethernet no puede ser tomada en serio como la base en Cluster, su ancho de banda y su latencia no están balanceados en comparación con el poder computacional de las estaciones de trabajo ahora disponibles. Típicamente, se puede esperar que el ancho de banda de una interconexión en un Cluster exceda los 10 Mbytes por segundo y tenga latencias en los mensajes no mayores a 100 µs. Un gran número de tecnologías de red de alto desempeño están disponibles en el mercado.

Además de las LAN Ethernet, existen otros tipos de interconexión, la Token Ring y FDDI, que también son populares, pero que no son usadas tan comúnmente en una interconexión de nodos en un Cluster.

A continuación se da una breve descripción de las redes de interconexión más comúnmente usadas en un Cluster.

- ***Ethernet, Fast Ethernet y Gigabit Ethernet***

  El término Ethernet se refiere a la familia de implementación de LAN que incluye tres categorías principales:

  1) El estándar Ethernet o IEEE 802.3 ha sido en algunas ocasiones sinómino de redes con estaciones de trabajo. Esta tecnología se usa ampliamente en sectores tanto académicos como comerciales. Sin embargo, su ancho de banda de 10 Mbps no es suficiente para ser usado en ambientes donde los datos transferidos por los usuarios tienen un gran tamaño o la densidad del tráfico es muy alta. IEEE 802.3 opera a 10 Mbps sobre cable coaxial y par trenzado.

  2) Fast Ethernet o IEEE 802.3u provee un ancho de banda de 100 Mbps y ha sido diseñada para actualizar las ya existentes redes Ethernet. La tecnología Fast Ethernet y la estándar no pueden coexistir en un mismo cable, aunque utilizan el mismo tipo. IEEE 802.3u opera sobre par trenzado y fibra óptica.

3) Gigabit Ethernet o IEEE 802.3z es una extensión a las normas de 10 Mbps y 100 Mbps, ofreciendo un ancho de banda de 1000 Mbps. Gigabit Ethernet mantiene compatibilidad completa con la base instalada de nodos Ethernet. Gigabit Ethernet extiende Ethernet y corre en ambos modos de operación, half y full duplex. Gigabit Ethernet promete ser atractiva desde el punto de vista del desempeño, el costo y por su compatibilidad con las estructuras de red actuales. IEEE 802.3z opera sobre par trenzado y fibra óptica.

- ***ATM (Asynchronous Transfer Mode)***

  Fue desarrollada por la industria de las telecomunicaciones. Intenta unificar los estilos de LAN y WAN.

  Es una tecnología de switching basada en unidades de datos de un tamaño fijo de 53 bytes llamadas celdas o células. ATM opera en modo orientada a conexión, esto significa que cuando dos nodos desean transferir deben primero de establecer un canal o conexión por medio de un protocolo de llamada o señalización. Una vez establecida la conexión, las celdas de ATM incluyen información que permiten identificar la conexión a la cual pertenecen.

  ATM está diseñada teniendo en mente que las celdas puedan ser transferidas a través de diferentes medios de comunicación. Esto hace que tenga diferentes niveles de desempeño.

  En una red ATM las comunicaciones se establecen a través de un conjunto de dispositivos intermedios llamados switches.

- ***SCI (Scalable Coherent Interface)***

  Es el equivalente moderno de un bus Procesador-Memoria-E/S y una red LAN combinados y trabajando en paralelo para soportar multiproceso distribuido con un elevado ancho de banda, muy baja latencia y una arquitectura escalable que permite diseñar grandes sistemas evitando la utilización de grandes bloques.

  SCI surge como una ramificación del proyecto *IEEE Standard Futurebus* en 1988, cuando se hizo evidente que los futuros procesadores serían pronto demasiados rápidos para cualquier bus y para soportar un costo razonable en configuraciones multiproceso. El grupo SCI buscó una nueva solución que proporcionara al usuario servicios propios de un bus, evitando los cuellos de botella, estabilidad en diseños con supercomputadoras y soporte efectivo de software para sistemas y aplicaciones de procesamiento paralelo.

  Las siglas de SCI corresponden con:

  ***Escalabilidad*** (*Scalability*).- Un sistema es escalable cuando su funcionamiento es independiente del número de procesadores que en él intervengan. La estabilidad de la arquitectura de un sistema posee varias dimensiones que atienden a eficiencia, costo económico, direccionamiento, independencia de software, etc.

***Coherencia*** (*Coherence*).- Utilización de forma eficiente y coherente de la memoria caché en los sistemas multiprocesador con memoria compartida.

***Interfaz*** (*Interface*).- Disponer de una arquitectura de comunicación abierta que permita la utilización de múltiples productos de diversos fabricantes.

Las principales características que contiene SCI son:

- Es un estándar que permite a los sistemas crecer con componentes modulares de diferentes fabricantes.
- Flujo de datos en el anillo de hasta 1 Gbytes por segundo por nodo.
- Memoria compartida.
- Opcionalmente coherencia caché basada en directorios distribuidos.
- Mecanismos de paso de mensajes.
- Escalable hasta 64 Kprocesadores.
- Enlaces de datos unidireccionales de 16 bits cada 2 ns.
- Disponibilidad de interfaz *Single Chip* que incluye todos los transceivers[27], FIFO's y lógica de protocolo con el consiguiente ahorro de costo y simplicidad.
- Enlaces de fibra óptica o coaxial de 1 Gbit por segundo, también posibles para aplicaciones LAN con protocolos eficientes de memoria compartida.
- Mecanismos de interfaz a otros buses comerciales.
- Arquitectura CSR (Control State Registers, IEEE std 1212-1991).

♦ ***Myrinet***

Myrinet es un red rápida LAN que conecta estaciones de trabajo o PC's de alto rendimiento mediante un enlace de 1.28 Gbits por segundo full duplex, con bajo tiempo de latencia y también es un paquete enrutador de alta tecnología que usa redes de tipo SAN y LAN.

Myrinet está basada en la tecnología usada para comunicaciones de paquetes y supercomputadoras paralelas. Redes convencionales como las Ethernet pueden ser usadas para construir Clusters, pero no proveen las características y desempeño requeridas por los Clusters de alta disponibilidad y alto desempeño.

- Enlace full duplex 1.28 + 1.28 Gbits por segundo con puertos de switches y puertos de interfaz.
- Control de flujo y control de errores en cada estación.
- Baja latencia.
- Interfaz en las estaciones de trabajo que permiten ejecutar un programa de control que maneja directamente la interacción entre las estaciones, los buffers, los mapeos de la red y el monitoreo de la misma.
- Las redes Myrinet pueden escalar de 10 a 100 hosts con una tasa de datos de Tbits por segundo y puede proveer direcciones de comunicación alternativa entre los hosts.

[27] Dispositivo que transmite y recibe señales digitales o analógicas. Generalmente reside en la tarjeta de red. Se encarga de colocar datos al cable de red y detectar y recibir los mismos.

- La conexión de la red es una arquitectura abierta, incluyendo las arquitecturas que permiten múltiples conexiones para desempeño y redundancia. Myrinet mapea la interfaz y permite la comunicación entre procesos.

El software desarrollado por Myrinet para la red reporta cortas latencias entre procesos de Unix, menor a cinco microsegundos, esto es mejor que el mejor sistema de memoria distribuida y requiere de una tasa de transmisión de 1 Gbit por segundo.

Ya es posible implementar sobre Myrinet muchos de los sistemas de cómputo paralelo como MPI (*Mensaje Passing Interface*) y PVM (*Parallel Virtual Memory*). El MPI y PVM están implementados directamente en la capa UDP/IP de Myrinet. A través de éstos se logra un mejor desempeño.

Además Myrinet puede implementarse como componentes sencillos conectados a una SAN. Myrinet provee acceso desde cualquier estación de trabajo conectada a una LAN Myrinet.

### *1.3.4. Lenguajes y compiladores*

No encontraremos ningún lenguaje de programación para máquinas paralelas tan potente como el versátil C compilado con *gcc*. Aún así, existen lenguajes de más alto nivel adecuados para este tipo de programación.

- ***Fortran***
  Fortran, tal y como es hoy, es un lenguaje de alto nivel, con numerosas mejoras con respecto al ANSI de 1966, (objetos como los de C++, instrucciones para procesamiento en paralelo, etc.).

  De hecho podemos encontrar compiladores capaces de generar código para arquitecturas paralelas tipo SMP y Clusters entre otras.

- ***Mentat***
  Mentat es un lenguaje orientado a objetos para procesamiento en paralelo, funciona en máquinas Clusters y se encuentra disponible para Linux. Su sintaxis es similar a la C++.

- ***mpC Programming Language***
  mpC es un lenguaje paralelo de alto nivel, una extensión del C, diseñado especialmente para desarrollar aplicaciones adaptables y portables para redes heterogéneas de computadoras. La idea principal subyacente en mpC es que cada aplicación define una red abstracta y distribuye datos, computación y comunicaciones sobre la red. El sistema de programación mpC usa esta información para reasignar la red abstracta sobre cualquier red real de forma que se asegura la eficiencia.

  El sistema mpC encapsula una plataforma particular de comunicación, actualmente un subconjunto de MPI, asegurando la independencia de la plataforma del resto de componentes del sistema.

- ***Cilk***
  Cilk es un lenguaje para programación paralela multihebrada basado en C. Está diseñado para utilizarse como un lenguaje de propósito general, pero es específicamente efectivo para paralelismo asíncrono, que puede ser difícil de escribir utilizando un estilo de paso de mensajes.

- ***Jade/SAM (dialecto concurrente de C) sobre PVM***
  Jade es un lenguaje de programación paralelo, una extensión de C, para explotar la concurrencia en programas secuenciales de grano grueso. Provee la conveniencia del modelo de memoria compartida permitiendo a cada tarea acceder a los objetos compartidos de forma transparente. Los programadores de Jade aumentan sus programas secuenciales con construcciones que descomponen la computación en tareas y declaran cómo esas tareas acceden a los objetos compartidos.

  La implementación de Jade interpreta de forma dinámica la información para ejecutar el programa paralelo mientras preserva la semántica secuencial si hay dependencia de datos entre tareas, las tareas se ejecutan en el mismo orden en el que lo harían en una ejecución secuencial. La estructura del paralelismo producida por un programa Jade es un grafo acíclico dirigido de tareas, donde cada arco entre nodos representa las restricciones de dependencia de los datos. Debido a que la construcción de la declaración del acceso a datos es ejecutada dinámicamente, este grafo de tareas puede ser dinámico y puede por tanto expresar la concurrencia disponible por la dependencia de datos sólo en tiempo de ejecución.

- ***Parallaxis (data parallel Modula-2) sobre PVM***
  Parallaxis es un lenguaje de programación estructurado para programación de datos paralelos en sistemas SIMD. El lenguaje está basado en Modula-2, pero ampliado por constructores paralelos independientes de la máquina. En Parallaxis, la abstracción se consigue declarando una configuración de procesador en forma funcional, especificando un número, el orden, y la conexión entre los procesadores.

- ***pC++ (data parallel C++) sobre PVM***
  pC++ es un C++ paralelo portable para computadoras de alto rendimiento. pC++ es una extensión de C++ que permite operaciones del estilo de datos-paralelos usando *colecciones de objetos* desde alguna base elemental de clase. Las funciones miembro de esta clase elemental se pueden aplicar a la totalidad de la colección en paralelo. Esto permite a los programadores componer estructuras paralelas de datos con una semántica de ejecución paralela. Estas estructuras distribuidas pueden estar alineadas y distribuidas sobre la jerarquía de memoria de la máquina paralela tanto como en HPF (*High Performance Fortran*). pC++ también incluye un mecanismo para el encapsulamiento al estilo de computación SPMD en un modelo de computación basado en hebras.

- ***ZPL (lenguaje basado en vectores) sobre PVM***
  ZPL es un lenguaje de programación de vectores diseñado principalmente para una rápida ejecución en computadoras secuenciales y paralelos. A causa de que ZPL se beneficia de

la reciente investigación en compilación paralela, suministra un mecanismo de alto nivel para supercomputadoras con una eficiencia comparable al paso de mensajes codificado *a pelo*. Usuarios con experiencia en computación científica pueden aprender ZPL en pocas horas.

- ***NESL***
  NESL es un lenguaje paralelo que integra varias ideas de la comunidad teórica, algoritmos paralelos, de la comunidad de lenguajes y de la comunidad de sistemas.

- ***Orca Parallel Programming Language***
  Orca es un lenguaje de programación paralela en sistemas distribuidos, basado en el modelo de objetos de datos compartido. Este modelo es una forma portable y simple del modelo basado en objetos de memoria distribuida compartida.

### *1.3.5. Entornos de programación y Sistemas Operativos*

Un Cluster se divide en dos partes principalmente. Primero, el hardware de interconexión de los elementos del Cluster y segundo, la parte software: un sistema operativo adaptado, por ejemplo GNU Linux con un kernel modificado, compiladores especiales y aplicaciones en general, elementos software que permitan a los programas que se ejecutan sobre el Cluster aprovechar todas sus ventajas.

A continuación se presentan algunos entornos de programación y sistemas operativos que permiten que el Cluster tenga un mayor desempeño.

- ***MPI*** (*Message Passing Interface*)
  El objetivo básico de MPI es desarrollar un estándar de amplia utilización para escribir programas de paso de mensajes. Esta interfaz intenta establecer un práctico, portable, eficiente y flexible estándar para el paso de mensajes.

  En el diseño del MPI se utilizó en gran medida el *MPI Forum* de forma que se tomaba en cuenta la opinión de los programadores. Ha sido influenciado en gran medida por el trabajo en el IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, y PARMACS. Otras contribuciones importantes provienen de Zipcode, Chimp, PVM, Chamaleon y PICL.

  La principal ventaja de establecer un estándar en el paso de mensajes es la portabilidad y la facilidad de uso. En un sistema de memoria compartida en el cual las rutinas de más alto nivel y las abstracciones están construidas sobre una capa de bajo nivel de paso de mensajes los beneficios de la estandarización son particularmente aparentes. Además permite que se ofrezca soporte hardware y se aumente la escalabilidad.

- ***PVM*** (*Parallel Virtual Machine*)
  PVM no es más que un API GNU de programación C para máquinas paralelas, capaz de crear una máquina virtual paralela, es decir, emplea los recursos libres de cada uno de los nodos sin tener que preocuparnos del cómo, haciéndonos creer que estamos ante una

única supermáquina. Un punto a destacar de PVM es que si disponemos de una red UNIX no tenemos más que instalar el software y ya dispondremos de un supercomputadora paralela con un costo nulo.

Además, PVM está portado para otras máquinas además de Linux, entre las que se encuentran distintos tipos de UNIX e incluso existe una versión para NT aunque cabe decir que alcanzan unas velocidades bastantes inferiores a las que alcanza con Linux.

- ***Inferno***
  Inferno es un nuevo sistema operativo y un entorno de programación para repartir contenido en un rico entorno de redes heterogéneas, clientes y servidores. El sistema Inferno incluye el kernel Inferno, el lenguaje de programación Limbo, y API's de referencia que incluyen interfaces para redes, gráficos, protocolos de red, seguridad y autentificación y varios kits de herramientas.

- ***PGDBG***
  PGDBG es depurador para programas paralelos en MPI y en OpenMP pensado para ser utilizado en Clusters con Linux.

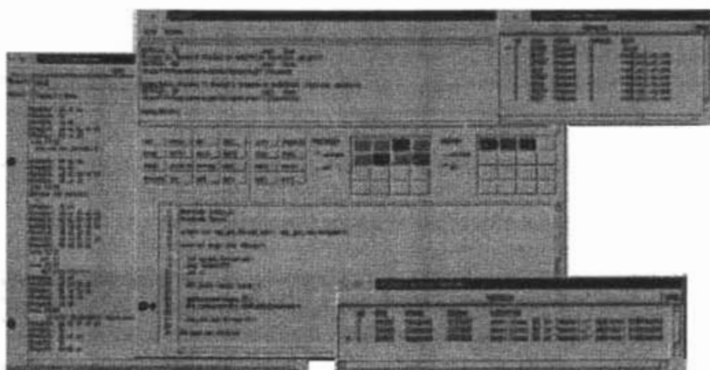

*Fig. 1.23 Imágenes de PGDBG*[28]

### *1.3.6. Equilibrado de cargas*

Con el crecimiento de Internet en los últimos años el tráfico en la red ha aumentado de forma radical y con él, la carga de trabajo que ha de ser soportada por los servidores de servicios, especialmente por los servidores web. Para soportar estos requerimientos hay dos soluciones: el servidor se basa en una máquina de altas prestaciones, que a largo plazo probablemente quede obsoleta por el crecimiento de la carga o bien se encamina la solución a la utilización de la

[28] Oscar Rafael García Regis, Enrique Cruz Martínez.
***Paralelización en la Supercomputadora Cray Origin 2000***
Cómputo Aplicado, DGSCA, y Facultad de Ciencias.
UNAM.

tecnología de clustering para mantener un Cluster de servidores. Cuando la carga de trabajo crezca, se añadirán más servidores al Cluster.

Hay varias formas de construir un Cluster de balanceo de carga. La mejor solución es utilizar un balanceador de carga para distribuir ésta entre los servidores de un Cluster. En este caso el balanceo queda a nivel de conexión, con una granularidad más fina y con mejores resultados. Además, se podrán enmascarar más fácilmente las posibles caídas de los nodos del Cluster. El balanceo de carga puede hacerse a dos niveles:

- De aplicación.
- A nivel IP.

El balanceo a nivel de aplicación puede provocar efectos de cuello de botella si el número de nodos es grande.

A continuación presentamos algunas herramientas utilizadas en un Cluster para poder realizar el equilibrado de cargas.

- ***Mosix***

  Mosix es una herramienta desarrollada para sistemas tipo UNIX, cuya característica resaltante es el uso de algoritmos compartidos, los cuales están diseñados para responder al instante a las variaciones en los recursos disponibles, realizando el balanceo efectivo de la carga en el Cluster mediante la migración automática de procesos o programas de un nodo a otro en forma sencilla y transparente.

  El uso de Mosix en un Cluster de PC's hace que éste trabaje de manera tal que los nodos funcionan como partes de un sola computadora. El principal objetivo de esta herramienta es distribuir la carga generada por aplicaciones secuenciales o paralelizadas.

  A diferencia de paquetes como MPI o PVM, Mosix realiza la localización automática de los recursos globales disponibles y ejecuta la migración dinámica *online* de procesos o programas para asegurar el aprovechamiento al máximo de cada nodo.

- ***Condor***

  Condor es un manejador de carga del sistema especializado para trabajos de computación intensiva. Los usuarios envían sus trabajos a Condor, el cual los coloca en una cola, elige cuándo y dónde ejecutar los trabajos, y finalmente informa al usuario que se han completado.

  Puede usarse para manejar un Cluster dedicado de nodos de computadoras, como los Cluster tipo *Beowulf*.

### *1.3.7. Cluster tipo Beowulf*

En el verano de 1994, Thomas Sterling y Donald Becker, trabajando en el CESDIS[29] (*Center of Excellence in Space Data and Information Sciences*) bajo la tutela del proyecto ESS (*Earth and Space Sciences*), construyeron un Cluster computacional consistente en procesadores de tipo x86 comerciales conectados por una red Ethernet de 10Mb. Llamaron a su máquina *Beowulf*, nombre de un héroe de la mitología danesa relatado en el libro *La Era de las Fábulas*, del autor norteamericano Thomas Bulfinch (Beowulf derrotó al monstruo gigante Grendel).

Inmediatamente, aquello fue un éxito y su idea de basar sistemas en el concepto de COTS (*Commodity Off The Shelf*) pronto se difundió a través de la NASA y las comunidades académicas y de investigación. El desarrollo de esta máquina pronto se vio enmarcado dentro de lo que hoy se conoce como *The Beowulf Project*. Los Clusters Beowulf están hoy reconocidos como un tipo de Clusters dentro de los HPC[30].

*Wiglaf* fue el primer Cluster de tipo Beowulf. Era un Cluster de 16 nodos con procesadores DX4 a 100 MHz (un híbrido entre el 80486 y el Intel P5 Pentium). La placa base estaba basada en el chipset SiS 82471, que era el de más altas prestaciones en bajo costo en aquel momento. Cada procesador disponía de 16 M de DRAM algo más rápida y cara que la usual en el mercado (60 ns). Y cada nodo poseía también 540 M de 1G EIDE disk. Además el sistema contenía tres tarjetas de red Ethernet a 10 Mbps.

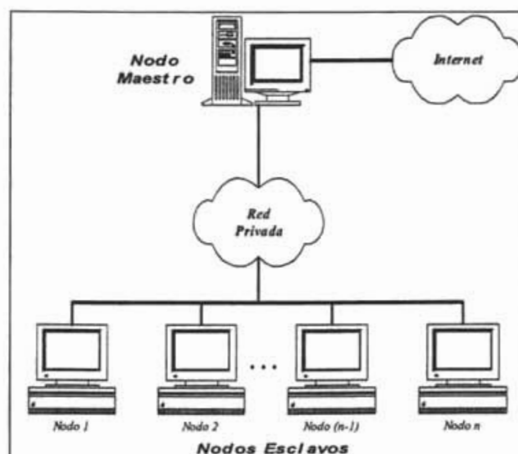


*Fig. 1.24 Diagrama general de un Cluster Beowulf*

[29] El CESDIS es una división de la USRA (*University Space Research Association*) localizada en Maryland. El CESDIS trabajaba para la NASA en el proyecto ESS. Este proyecto determinó la aplicabilidad de las computadoras MPP a los problemas que surgían en aquel momento de la investigación de la tierra y el espacio.
[30] Clusters de Alto Rendimiento.

Un Cluster de tipo Beowulf no es más que una colección de computadoras personales interconectadas por medio de una red privada de alta velocidad, corriendo algún sistema operativo libre: Linux, Fedora, etc.

Los nodos en el Cluster están dedicados exclusivamente a ejecutar tareas asignadas al Cluster. Por lo general el Cluster se encuentra comunicado al mundo exterior a través de un solo nodo, llamado nodo maestro, el cual también está reservado para acceder, compilar y manejar las aplicaciones a ejecutar. Si bien hay que señalar que en los Clusters de computación se elige la rapidez frente a la seguridad con lo que se relajan las medidas de seguridad entre nodos hasta el punto de hacer poco recomendable la conexión del Cluster a Internet; incluso el sistema utilizado para mantenimiento y administración debe estar desconectado de la red.

A medida que la tecnología ha ido avanzando se han podido construir Clusters con procesadores más rápidos, mejores tecnologías de red y costo más bajo frente a las prestaciones. Sin embargo, una característica importante de los Clusters Beowulf es que todos estos cambios no modifican el modelo de programación. A esta independencia del hardware ha contribuido en gran medida la madurez de GNULinux y de la utilización del paso de mensajes vía PVM y MPI.

La crítica histórica de que el software para computadoras de alta computación es dedicado y poco reciclable deja de ser real desde el momento en que, con Beowulf, se escapa de la dependencia de los modelos de software forzados por los fabricantes del hardware de alta computación.

Si intentáramos clasificar los Clusters Beowulf, podríamos decir que se encuentran en un punto intermedio entre los MPP y las NOW's. Los Clusters Beowulf se benefician de distintas cualidades de ambos tipos de sistemas. Los MPP's (nCube, CM5, Convex SPP, Cray T3D, Cray T3E, etc.) son más grandes y con menos latencia en la red de interconexión que los Clusters Beowulf. Sin embargo, los programadores deben preocuparse acerca de la localidad, el balanceo de carga, la granularidad y los overheads[31] de comunicación para obtener los mejores resultados. Incluso en máquinas con memoria compartida, muchos programadores, por comodidad, desarrollan sus programas en un estilo de paso de mensajes. Este último tipo de programas se podría portar perfectamente a Clusters de tipo Beowulf.

Por otro lado, en una NOW se suelen ejecutar programas poco ajustados, tolerantes a problemas de balanceo de carga y alta latencia de comunicación. Cualquier programa que funcione sobre una NOW va a funcionar, al menos con la misma eficiencia, en un Cluster Beowulf. Sin embargo, las diferencias de concepto entre un Cluster Beowulf y una NOW son significativas. Primero, los nodos en un Cluster están dedicados exclusivamente al Cluster, es decir, la labor de cada nodo no está sujeta a factores externos, lo que facilita la resolución de problemas de balanceo de carga. Además, como la red de interconexión de un Cluster está aislada del resto de la red, el tráfico que soporta es sólo el generado por la aplicación que está siendo ejecutada sobre el Cluster. Esto evita el problema de latencia impredecible que se da en las NOW's. Además, el aislamiento de la red evita problemas de seguridad, que sí han de ser tenidos en cuenta en las NOW's. Asimismo, en un Cluster se pueden realizar ajustes en parámetros del sistema operativo para mejorar el funcionamiento de los nodos en el Cluster, cosa que no se puede hacer en las

[31] Se utiliza para describir el tiempo empleado en la preparación para la realización de alguna tarea, procedimiento necesario pero que no forma parte de la tarea como tal.

NOW ya que los nodos realizan otras tareas independientes y propias. Por último, en las NOW's, el trabajo se replica en los nodos para luego ser contrastado. Esto se lleva a cabo por cuestiones de seguridad ya que los nodos forman parte de entornos de administración distintos.

# Capítulo II
## Metodología orientada al procesamiento paralelo

# *Metodología orientada al procesamiento paralelo*

Durante la última década hemos sido testigos del crecimiento explosivo en las capacidades y rendimiento de los sistemas de cómputo. Esos mejoramientos se deben a dos tipos de cambios: tecnológicos y arquitecturales.

Los cambios arquitecturales se basan fundamentalmente en organizaciones nuevas que permiten realizar nuevas funciones o las funciones anteriores con mayor velocidad. Ambos aspectos, arquitecturales y tecnológicos, están estrechamente ligados entre sí; mejoras tecnológicas promueven cambios arquitecturales y éstos, a su vez, demandan más capacidades a los circuitos.

Uno de los aspectos que mejor representan estos cambios es sin lugar a dudas la aparición de las computadoras paralelas. Una computadora paralela es una colección de elementos de procesamiento que se comunican y cooperan para resolver problemas grandes de manera rápida.

La programación paralela se refiere a la forma de construir las aplicaciones que trabajen en computadoras paralelas.

## 2.1. FUNDAMENTOS EN SISTEMAS DE CÓMPUTO PARALELO

Desde el modelo propuesto por Von Neuman, la computación secuencial ha sido el modo habitual de computación. Sin embargo, existe una demanda permanente de mayor rendimiento computacional, por ende se recurre al cómputo paralelo, pues la solución en máquinas secuenciales no sería óptimo o posible en tiempos razonables. El uso del cómputo paralelo comprende una amplia gama de aplicaciones, tales como: la predicción del clima, modelado de la biosfera, exploración petrolera, procesamiento de imágenes, fusión nuclear, modelado de océanos, sincronización de osciladores entre otras.

Este modelo de programación ofrece:

- Alternativas a los relojes rápidos de mejorar el rendimiento.
- Aplicación a todos los niveles del diseño de un sistema de cómputo.
- Mayor importancia en aplicaciones que demandan alto rendimiento.
- Una insaciable necesidad de computadoras rápidas.
- Tendencias en la tecnología, en la arquitectura y en la economía.
- Apoyo para el cómputo científico: física, química, biología, oceanografía, astronomía, etc., y para el aceleramiento del cómputo de propósito general: video, CAD[32], bases de datos, procesamiento de transacciones, etc.
- Una forma natural de mejorar el rendimiento.
- Explotación a muchos niveles.
  - A nivel de instrucciones.
  - Servidores multiprocesadores.

[32] CAD = *Computer Assisted Design*, Diseño Asistido por Computadora.

- Computadoras paralelas masivas.

### *2.1.1. ¿Qué es paralelismo y qué es cómputo paralelo?*

El *paralelismo* es la realización de varias actividades al mismo tiempo que tienen una interrelación.

El *cómputo paralelo* es la ejecución de más de un cómputo (cálculo) al mismo tiempo usando más de un procesador.

Se trata de reducir al mínimo el tiempo total de cómputo, distribuyendo la carga de trabajo entre los procesadores disponibles. Obtener un alto rendimiento o mayor velocidad al ejecutar un programa es una de las razones principales para utilizar el paralelismo en el diseño de hardware o software.

### *2.1.2. ¿Qué es computación paralela y qué aspectos involucra?*

Es el proceso de información que enfatiza la manipulación concurrente de elementos de datos pertenecientes a uno o más procesos resolviendo un problema común.

Entre los aspectos más importantes relacionados con la computación paralela están:

- Diseño de Computadoras Paralelas.
- Diseño de Algoritmos Eficientes.
- Métodos para Evaluar Algoritmos Paralelos.
- Programación Automática de Computadoras Paralelas.
- Lenguajes de Programación Paralela.
- Herramientas para Programación Paralela.
- Portabilidad de Programas Paralelos.

### *2.1.3. ¿Qué es una computadora paralela?*

Una computadora paralela es un conjunto de procesadores capaces de trabajar cooperativamente para resolver un problema computacional.

Las computadoras paralelas son interesantes porque ofrecen recursos computacionales potenciales. Otra definición sería una colección de elementos de procesamiento que se comunican y cooperan entre sí para resolver problemas grandes de manera rápida.

### *2.1.4. ¿Cuándo hay que paralelizar?*

La respuesta a esta pregunta depende del análisis que se debe efectuar tomando en cuenta el hardware y software necesario, para determinar si vale la pena paralelizar el programa o viceversa.

En términos del hardware, paralelizar código involucra conocer la arquitectura de la supercomputadora o Cluster donde se pretende paralelizar, conocer el número de procesadores con los que cuenta, la cantidad de memoria, espacio en disco, los niveles de memoria disponible, el medio de interconexión, etc.

En términos de software, se debe conocer qué sistema operativo está manejando, si los compiladores instalados permiten realizar aplicaciones con paralelismo, si se cuenta con herramientas como PVM o MPI.

Para decidir si es conveniente paralelizar un código, podemos ubicar el problema considerando los siguientes puntos:

- *Necesidad de respuesta inmediata de resultados*
  Al ejecutar varias veces un programa en una máquina secuencial con diferentes datos de entrada, donde cada tiempo de ejecución es considerable, es inapropiado o costoso esperar tanto tiempo para volver a realizar otra ejecución, afectando notablemente el desempeño de la máquina. Esto, sin considerar las modificaciones al código o fallas en la ejecución del modelo. La paralelización y la ejecución del programa en una máquina paralela permiten realizar más análisis en menos tiempo.

- *Problema de Gran Reto*
  Cuando un problema se tiene que resolver mediante algoritmos de cálculo científico intensivo que demandan grandes recursos de cómputo, la frase *divide y vencerás* tiene un significado más amplio. Las tareas se dividen entre varios procesadores, se ejecutan en paralelo y se obtiene una mejora en la relación costo-desempeño. Actualmente las arquitecturas de cómputo incorporan paralelismo en los más altos niveles de sus sistemas, satisfaciendo las exigencias de los problemas de gran reto.

- *Elegancia de programación*
  Es decisión del programador escribir un algoritmo paralelo aunque no se justifique ni por tiempo ni por uso intensivo de otros recursos. Para paralelizar un programa es importante identificar en el código:

  - *Tareas independientes*.- Que existan ciclos *for* o *do* independientes, y rutinas o módulos independientes. De tal forma que no existan dependencias de datos que dificulten la paralelización.
  - *Zonas donde se efectúa la mayor carga de trabajo*.- Estas zonas deben de consumir la mayor parte de tiempo de ejecución. Para detectar dichas zonas existen herramientas que permiten obtener una perspectiva o perfil del programa.

### 2.1.5. *¿Qué se necesita para paralelizar?*

Es necesario disponer de un *ambiente paralelo*, constituido de:

- Hardware de Multiprocesamiento.
- Soporte del sistema operativo para paralelizar.

- Herramientas de desarrollo de software paralelizable.

## 2.2. NIVELES DE PARALELISMO

Normalmente la computadora se concibe como una máquina secuencial, donde el CPU lee la instrucción de la memoria y la ejecuta una por una. De la misma manera, al especificar algoritmos se concibe una secuencia de instrucciones a ser ejecutadas una a una.

Aunque internamente, las computadoras secuenciales presentan un paralelismo en diversas funciones (paralelismo a nivel de instrucciones), por mencionar algunas: el traslape de algunas operaciones, mientras un proceso está escribiendo a disco, otro proceso esta ejecutándose en el CPU; a nivel de micro operación, señales de control se generan y viajan al mismo tiempo a través de canales paralelos, como el caso de la comunicación a una impresora conectada al puerto paralelo.

En computadoras modernas de tipo SISD el paralelismo viene implementado en la arquitectura del procesador, conocido como *paralelismo a nivel instrucción, Pipeline*. Éste consiste en ejecutar al mismo tiempo diversas etapas de instrucciones del programa; mientras en una etapa se hace la ejecución de una instrucción, simultáneamente en otra etapa se está realizando una lectura de la siguiente instrucción.

Una característica de procesadores recientes es que poseen varias ALU's[33] para realizar operaciones de suma, resta, multiplicación y división en forma paralela. El paralelismo a nivel de programación puede ser complicado o sencillo, depende de la arquitectura, del modelo para programar en paralelo e inclusive del programa en sí.

### *2.2.1. Granularidad*

Para paralelizar una aplicación es necesario contar con un lenguaje o biblioteca de programación que brinde las herramientas necesarias para realizar esto. Dependiendo de la herramienta con que se cuente, se particionará el código en pedazos para que se ejecuten en paralelo en varios procesadores. De aquí surge el término de granularidad.

Granularidad es el tamaño de las piezas en que se divide una aplicación. Dichas piezas pueden ser desde una sentencia de código, una función hasta un proceso en sí que se ejecutarán en paralelo.

La granularidad de sincronización, o frecuencia, entre procesos en el sistema, es una buena manera de caracterizar multiprocesadores y ubicarlos en un contexto con otras arquitecturas. Se pueden distinguir cinco categorías de paralelismo que difieren en el grado de granularidad. Estas categorías se encuentran resumidas en la siguiente tabla.

[33] ALU = Aritmetic Logic Unit, Unidad Aritmética Lógica.

| Tamaño de grano | Descripción | Intervalo de sincronización (instrucciones) |
|---|---|---|
| Fino | Paralelismo inherente en un único flujo de instrucciones. | < 20 |
| Medio | Procesamiento paralelo o multitarea dentro de una aplicación individual. | 20-200 |
| Grueso | Multiprocesamiento de procesos concurrentes en un entorno multiprogramado. | 200-2000 |
| Muy Grueso | Proceso distribuido por los nodos de un red para formar un solo entorno de computación | 2000-1M |
| Independiente | Varios procesos no relacionados. | (N/A) |

*Tabla 2.1 Grados de Granularidad*

El paralelismo de grano fino y grueso se puede presentar en sistemas de memoria compartida sólo que el de grano grueso es más complicado de programar que el de grano fino.

### *2.2.1.1. Teoría de la paralelización de grano fino*

Se presenta cuando el código se divide en una gran cantidad de piezas pequeñas. A nivel de sentencia, o en un ciclo cuando se divide en varios subciclos que se ejecutan en paralelo. También se le conoce como *Paralelismo de Instrucción.*

El paralelismo de grado fino representa un uso mucho más complejo del paralelismo que es encontrado en el uso de hebras. Aunque muchos trabajos han sido hechos en aplicaciones altamente paralelas, es un área especializada y fragmentada, con muchos enfoques diferentes.

### *2.2.1.2. Teoría de la paralelización de grano medio*

Una aplicación puede ser efectivamente implementada como una colección de hebras con un paralelismo simple. En este caso, el paralelismo potencial de una aplicación debe ser explícitamente especificado por el programador. Generalmente se necesitará un alto grado de coordinación e interacción entre las hebras de una aplicación, levando a un nivel medio de sincronización.

Mientras que el paralelismo independiente, de grano grueso y de grano muy grueso pueden verse respaldados tanto en un monoprocesador multiprogramado como en un multiprocesador con poco o ningún impacto sobre la función de planificación, se debe revisar la planificación cuando se trata de la planificación de hilos. Puesto que los diversos hilos de una planificación interactúan de forma muy frecuente, las decisiones de planificación que involucran a un hilo pueden afectar al rendimiento de la aplicación completa.

### *2.2.1.3. Teoría de la paralelización de grano grueso*

Se realiza a nivel de subrutinas o segmentos de código, donde las piezas son pocas y de cómputo más intensivo que las de grano fino. También se le conoce como *Paralelismo de Tareas.*

El nivel más alto de paralelismo de grano grueso se presenta cuando se detectan tareas independientes y estas se ejecutan como procesos independientes en más de un procesador. Este esquema es común en las aplicaciones de Productor-Consumidor, Lector-Escritor, Maestro-Esclavo y Cliente-Servidor. En los modelos de Memoria Distribuida sólo se implementa paralelismo de grano grueso.

#### 2.2.1.4. *Teoría de la paralelización de grano muy grueso*

Con esta clase de paralelismo existe sincronización entre procesos pero a nivel muy difícil. Esta clase de situación es fácilmente entendible como un grupo de procesos concurrentes ejecutándose en un monoprocesador multiprogramado y puede ser soportado en un multiprocesador con un pequeño o no cambio al software del usuario.

En general, cualquier conjunto de procesos concurrentes que necesiten comunicarse o sincronizarse puede aprovechar el uso de las arquitecturas de los multiprocesadores. Un sistema distribuido puede ofrecer un soporte adecuado en caso de interacciones poco frecuentes entre los procesos. Sin embargo, si la interacción es algo más frecuente, el sobrecargo de comunicaciones a través de la red puede anular parte de la posible aceleración. En este caso, la organización del multiprocesador ofrece el soporte más efectivo.

#### *2.2.1.5. Teoría de la paralelización de grano independiente*

Entre los procesos no existe una sincronización explícita. Cada uno representa una separación, una aplicación independiente. El uso típico de este tipo de paralelismo es en los sistemas de tiempo compartido.

Cada usuario está ejecutando una aplicación en particular, como un procesador de textos o una hoja de cálculo. El multiprocesador ofrece el mismo servicio que un procesador multiprogramado. Como hay más de un procesador disponible, el tiempo medio de respuesta a los usuarios será menor.

Es posible alcanzar un aumento similar de rendimiento proporcionado a cada usuario una computadora personal o una estación de trabajo. Si van a compartirse archivos o alguna información, entonces se deben conectar los sistemas individuales en un sistema distribuido soportado por una red. Por otro lado, un único sistema multiprocesador ofrece, en muchos casos, un costo mejor que un sistema distribuido, pudiendo así mejorar los discos y otros periféricos.

## 2.3. MODELOS DE PARALELIZACIÓN

En la planificación de un procesamiento paralelo se debe de tener en cuenta la forma de asignación de los procesos a los procesadores. La asignación puede ser estática o dinámica.

Si se asigna un proceso a un procesador de forma permanente, desde su activación hasta su terminación, debe mantenerse una cola a corto plazo dedicada para cada procesador .Una ventaja

de este método es que puede existir una sobrecarga menor en la función de planificación porque la asignación al procesador se realiza una sola vez y para siempre.

Una desventaja de la asignación estática es que un procesador puede estar desocupado, con cola vacía, mientras que otro procesador tiene trabajos pendientes. Para prevenir esta situación, se puede usar una cola común. Todos los procesos van a una cola global y son ejecutados en cualquier procesador que esté disponible. De este modo, durante la vida de un proceso, este se puede ejecutar en diferentes procesadores en momentos diferentes y la información de contexto de todos los procesos se encuentra disponible para todos los procesadores y, por lo tanto el costo de la planificación de un proceso será independiente de la identidad del procesador en el que fue planificado.

La forma de asignar los procesos a los procesadores puede ser por medio de dos modelos:

- Maestro-Esclavo.
- Divide y Vencerás.

### *2.3.1. Maestro-Esclavo*

Como es usualmente en el caso de la computación paralela, se presume que la comunicación es más lenta a la hora de la detección de una colisión. Esta suposición implica la representación del diseño de la comunicación para asegurar que los procesadores de la computadora estén ocupados y no se pierda tiempo en ellos. Una estrategia para obtener este objetivo es usando un modelo llamado Maestro-Esclavo.

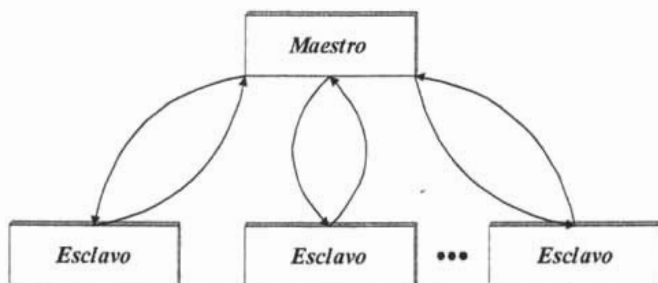


***Fig. 2.1 Concepto Maestro-Esclavo***

- *Ventajas*
  - Facilidad de cambio.
  - Los Esclavos son independientes del Maestro.
  - Si introducimos una clase abstracta Esclavo, tenemos diferentes implementaciones de los Esclavos.
  - Separación de funciones.
  - Eficiencia si hay procesamiento en paralelo.

- *Desventajas*
  - Un fallo en el Maestro hace caer todo el sistema.

- El maestro puede llegar a ser un cuello de botella del rendimiento.
- Alto consumo de recursos de la máquina que puede llegar a ser prohibido en ocasiones.
- Dependencia de la arquitectura de la máquina en caso de procesamiento paralelo.
- Dificultad de implementación. Es difícil definir el mecanismo de división del trabajo, cooperación y evaluación conjunta de resultados.
- Hay que definir qué ocurre si fallan los componentes individuales.
- Poco portable por la fuente de dependencia de la máquina que implica el procesamiento paralelo.

Esta estructura se basa en las organizaciones del mundo real, la forma de trabajo consiste en que el Maestro, controlador de la estructura, envía iterativamente datos a los Esclavos, realizadores del trabajo real, y recibe resultados de éstos. Usualmente, los Esclavos son generados como requisitos para resolver un problema.

La implementación de este modelo es el siguiente:

1. *Dividir el trabajo.*
2. *Combinar los resultados de las subtareas.*
3. *Definir cómo es la cooperación entre el Maestro y los Esclavos.*
   Especificar la interfaz del subservicio que dan los esclavos al maestro. Existen distintas posibilidades. Una opción es que el Maestro arranque subtareas en cada Esclavo mediante un parámetro en la llamada al Esclavo, que tendría la forma de una llamada a función. Otra es que exista un almacén de datos global en el que el Maestro escribe sus peticiones y los Esclavos las recogen para atenderlas. El Esclavo puede tener sus propios datos o compartirlos con los demás Esclavos. Los Esclavos pueden devolver su resultado en forma de parámetro de retorno de una función, o lo pueden escribir en una variable global cuyo valor lee el Maestro para hacer sus cálculos. Para elegir la solución más adecuada, hay que tener en cuenta los siguientes factores: Costo de duplicar estructuras de datos en términos de espacio en memoria. Costo de las llamadas a los Esclavos en términos de tiempo de proceso. Si los Esclavos sólo leen ciertos datos y no los modifican, esos datos pueden ser globales.
4. *Implementar los esclavos.*
5. *Implementar el maestro.*

Este modelo de programación puede usar balance de cargas estático o balance de cargas dinámico. En el primer caso, la distribución de tareas es toda desarrollada al inicio del cómputo, lo cual permite al Maestro participar en el cómputo después de que a cada Esclavo se le ha proporcionado una fracción del trabajo. La distribución de las tareas puede ser hecha una sola vez o de manera cíclica.

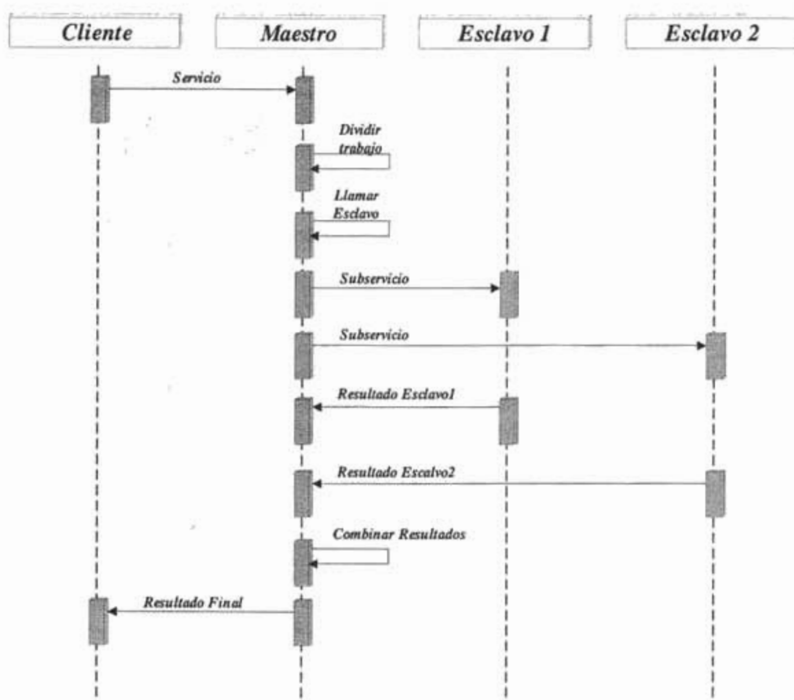


*Fig. 2.2 Implementación del modelo Maestro-Esclavo en forma Estática*

La otra manera es usar un balance de cargas dinámico. Es conveniente usarlo cuando:

- El número de tareas excede el número de procesadores disponibles.
- El número de tareas es desconocido al inicio de la aplicación.
- Los tiempos de ejecución no son predecibles.
- Cuando se está negociando con problemas no balanceados.

Una característica importante del balanceo de cargas dinámico es la capacidad de que la aplicación se pueda adaptar por sí misma a cambios dentro del sistema, lo que hace posible una reconfiguración de los recursos.

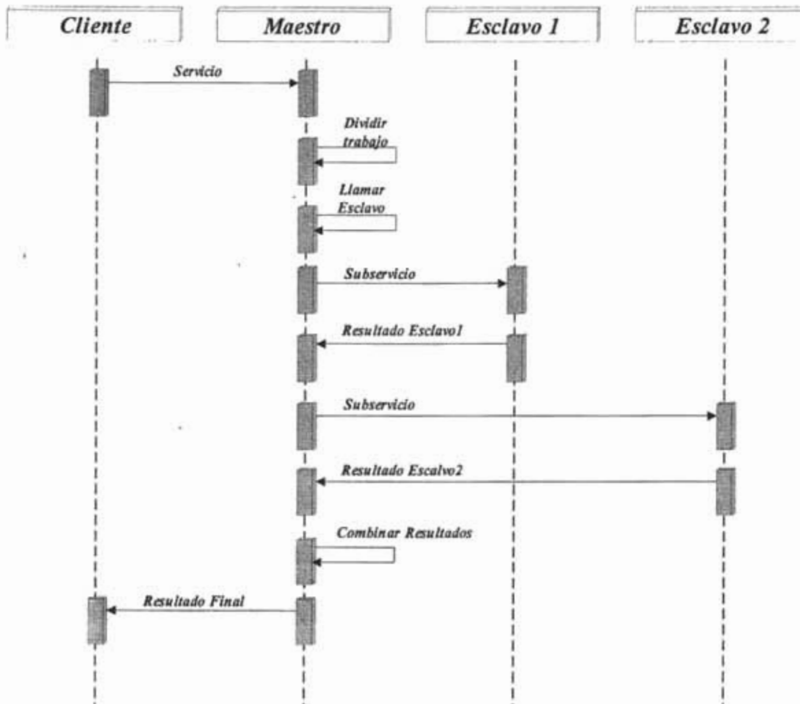


*Fig. 2.3 Implementación del modelo Maestro-Esclavo en forma Dinámica*

Con esta característica el modelo responde bien si hay fallas en algunos procesadores lo cual simplifica la posible creación de aplicaciones robustas que son capaces de sobrevivir a la pérdida de algunos Esclavos o inclusive hasta la pérdida del Maestro.

De acuerdo a las dependencias de datos se dan dos casos de dependencias entre iteraciones:

- *Iteraciones dependientes*
  El maestro necesita los resultados de todos los esclavos para generar un nuevo conjunto de datos.

- *Entradas de datos independientes*
  Los datos llegan al maestro, que no necesita resultados anteriores para generar un nuevo conjunto de datos.

El trabajo del Maestro es dividir el trabajo entre los esclavos disponibles, guardando cada uno de ellos sin usar demasiada la comunicación. Si se usa la detección de colisión directa entre un número $n$ de objetos en una escena, el trabajo del Maestro es muy fácil. En la figura 2.4 se muestran algunas combinaciones que se pueden tener en un objeto que se encuentra en cola y espera a un Esclavo disponible. Si un esclavo está disponible y hay trabajo para hacer, el Maestro

asigna el trabajo al Esclavo, mientras que el resto se ponen en cola y esperan hasta que se les asigne un trabajo.

*Fig. 2.4 Combinaciones del objeto en cola*

El Maestro hace más que un solo trabajo. El Maestro está en una posición excelente para mirar el trabajo que se hará globalmente. Después de haber realizado este chequeo de alto nivel, sólo los trabajos necesarios se crean y ponen en cola. Estos trabajos tienen que ser realizados y se ejecutarán por los Esclavos.

Un aspecto importante para la Paralelización es el control de la Granularidad[34]. Cuando presentamos un trabajo al Maestro, éste realiza una búsqueda por nodos limitado por la profundidad de la búsqueda máxima por el descubrimiento de la colisión. Ahora, cuando la profundidad máxima se ha alcanzado, el Esclavo puede someter un nuevo trabajo al Maestro. El Maestro lo pone en la cola de espera como cualquier otro trabajo y espera un Esclavo para ejecutarlo. Esta técnica permite controlar, dinámicamente, la granularidad.

Por otro lado, el Esclavo hace la interferencia que verifica y puede generar los nuevos trabajos para el Maestro. Si un trabajo es sometido a un Esclavo que contendrá dos nodos de arranque de los árboles. Antes de empezar el descubrimiento de la colisión, el esclavo tendrá que localizar estos nodos en su espacio de memoria. Una posibilidad es especificar los nodos por el camino de arranque a la raíz del árbol. Esto requeriría especificando dos valores: un bit String que contiene las *instrucciones* left/right para el camino y un entero que contiene el número de bits que son válidos en el bit String para que no se descienda demasiado profundo en el árbol. Este acercamiento utiliza poca memoria y es lento.

Este modelo, Maestro-Esclavo, es tolerante frente a fallos, ya que si falla un esclavo, se tienen más. Permite el procesamiento en paralelo, pues cada esclavo trabaja independientemente de los demás. Además, el Maestro valida la exactitud del resultado final comparando las salidas de los diferentes Esclavos.

[34] Ver el subtema 2.2 Niveles de Paralelismo.

Este modelo puede lograr una velocidad de cómputo alta y un interesante grado de escalabilidad. Sin embargo, para un gran número de procesadores el control centralizado del proceso Maestro puede causar cuellos de botella en la aplicación. Se puede mejorar la estabilidad del paradigma teniendo en lugar de un solo Maestro un grupo de Maestros, cada uno de ellos controlando a diferentes grupos de procesos Esclavos.

Entre las posibles aplicaciones de este modelo se encuentran la multiplicación de matrices (cada fila la calcula un esclavo), codificación de imágenes, cálculo de la correlación de dos señales, entre otras.

### *2.3.2. Divide y Vencerás*

La técnica de diseño de algoritmos de este tipo de modelo consiste en descomponer el problema original en varios subproblemas más sencillos (fase de dividir), para luego resolver éstos independiente mediante un cálculo sencillo (fase de conquistar). Por último, se combinan los resultados de cada subproblema para obtener la solución del problema original (fase de combinar).

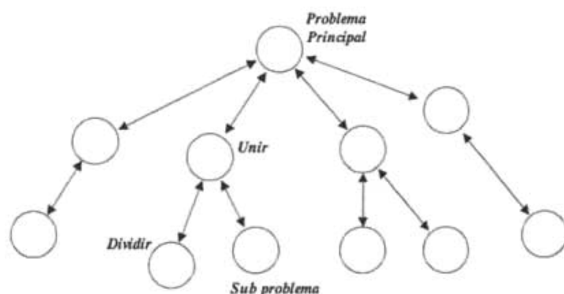


*Fig. 2.5 Concepto Divide y Vencerás*

Los requisitos para aplicar del modelo de Divide y Vencerás son los siguientes:

- Necesitamos un método de resolver los problemas de tamaño pequeño.
- El problema original debe poder dividirse fácilmente en un conjunto de subproblemas, del mismo tipo que el problema original pero con una resolución más sencilla.
- Los subproblemas deben ser disjuntos, la solución de un subproblema debe obtenerse independientemente de los otros.
- Es necesario tener un método de combinar los resultados de los subproblemas.

La aplicación del modelo Divide y Vencerás radica en la forma de definir funciones genéricas, mediante un razonamiento inductivo.

- *Pequeño*
  Determina cuándo el problema es pequeño para aplicar la resolución directa.
- *Solución Directa*
  Método alternativo de resolución para tamaños pequeños.

- *Dividir*
  Función para descomponer un problema grande en subproblemas.
- *Combinar*
  Método para obtener la solución al problema original, a partir de las soluciones de los subproblemas.

La subdivisión del trabajo puede hacerse entre todos los procesadores existentes. Cada uno de los subproblemas en que se dividió el problema global puede asignarse a un procesador. Cada uno de los procesadores recibe una parte de los datos y los procesa.

El tiempo de ejecución de un algoritmo de Divide y Vencerás, $T(n)$, viene dado por la suma de dos elementos:

- El tiempo que tarda en resolver los A subproblemas en los que se divide el original, $AT(\frac{n}{B})$, donde $\frac{n}{B}$ es el tamaño de cada subproblema.
- El tiempo necesario para combinar las soluciones de los subproblemas para hallar la solución del original; normalmente es $O(n^k)$.

El tiempo total es $T(n) = AT(\frac{n}{B}) + O(n^k)$. La solución de esta ecuación, si A es mayor o igual que 1 y B es mayor que 1, es:

$$\begin{array}{ll} \text{Si } A > B^k, & T(n) = O(n^{\log_B A}) \\ \text{Si } A = B^k, & T(n) = O(n^k \log n) \\ \text{Si } A < B^k, & T(n) = O(n^k) \end{array}$$

Uno de los aspectos que hay que tener en cuenta en los algoritmos de Divide y Vencerás es dónde colocar el umbral, esto es, cuándo se considera que un subproblema es suficientemente pequeño como para no tener que dividirlo para resolverlo. Normalmente esto es lo que hace que un algoritmo de Divide y Vencerás sea efectivo o no.

## 2.4. ETAPAS EN LA CREACIÓN DE PROGRAMAS PARALELOS

El diseño de algoritmos paralelos no se reduce a simples recetas de cocina, sino que se requiere tener creatividad y sobre todo entender el problema que se desea resolver mediante dicho algoritmo. Sin embargo puede darse un enfoque metódico que permita maximizar el rango de opciones consideradas, brindar mecanismos para evaluar las alternativas que se tienen, y reducir el costo de *backtracking*[35] por malas elecciones.

[35] Costo referido al retroceso de un proceso.

Las etapas de diseño que se describen a continuación tratan de ser lo más generales posibles, y representan las bases del trabajo que aquí se va a proponer para solucionar un problema mediante programación paralela.

Estas etapas están orientadas a resolver un problema en específico, como se verá en el siguiente capítulo, pero no por ello deja de ser válida para muchos otros problemas, pues se busca plantearla de manera general, y posteriormente emplearla en la solución del problema específico que se tratará aquí. Consta de 3 etapas:

- Particionamiento.
- Orquestación.
- Mapeo y Calendarización.

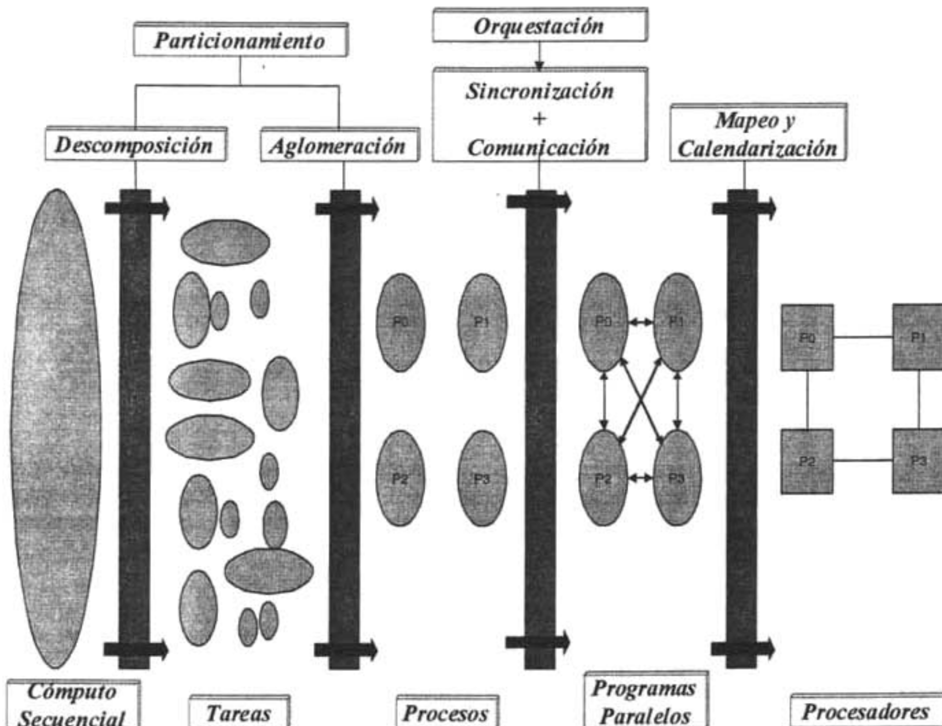


*Fig. 2.6 Etapas para la creación de programas paralelos*

El diseño que se está proponiendo aparentemente es una actividad secuencial, pues consta de 3 pasos básicos, sin embargo, en la práctica se podrá ver que es un proceso altamente paralelo y con muchos temas a considerar en donde tal vez será necesario realizar *backtracking*.

### *2.4.1. Particionamiento*

Esta etapa consiste en resaltar las posibilidades de ejecución paralela. Se concentra en la definición de un gran número de pequeñas tareas a fin de producir lo que se conoce como la descomposición de un problema en grano fino. Esta etapa se subdivide en 2 partes:

- Descomposición.
- Aglomeración.

Un buen particionamiento divide tanto los cálculos asociados con el problema como los datos sobre los cuales opera.

#### *2.4.1.1. Descomposición*

Hay alternativas en esta subetapa:

- *Descomposición del dominio*
  Se concentra en el particionamiento de los datos. Primero, se busca descomponer los datos asociados al problema. Se trata de dividir los datos en piezas del mismo tamaño. Después, se dividen los cálculos que serán realizados. Asociando cada operación con los datos sobre los cuales opera. Los datos a ser descompuestos pueden ser:

  - La entrada del programa
  - La salida computada por el programa.
  - Los valores intermedios mantenidos por el programa.

  Una regla que se tiene que seguir es concentrarse en la estructura de datos más grandes o la que se accede con mayor frecuencia.

- *Descomposición funcional*
  Se concentra en la descomposición de funciones o tareas. Primero, se concentra en los cálculos que serán realizados. Se divide el procesamiento en tareas disjuntas. Después, se procede a examinar los datos que serán utilizados por esas tareas. Si los datos son disjuntos, el particionamiento es completo. Si los datos no son disjuntos, se requiere replicar los datos o comunicarlos entre tareas diferentes.

El objetivo de la descomposición es definir al menos un orden de magnitud de más tareas que procesadores en la computadora paralela, pues en caso contrario, se tendrá poca flexibilidad en las etapas siguientes.

Debe evitar cálculos y almacenamientos redundantes. En caso contrario, difícilmente se logrará un algoritmo escalable.

Debe generar tareas de tamaño comparable. En caso contrario, puede ser difícil asignar a cada procesador cantidades de trabajo similares. Debe generar tareas escalables con el tamaño del

problema, ya que un incremento en el tamaño del problema debe incrementar el número de tareas en lugar del tamaño de las tareas individuales.

Es recomendable considerar varias alternativas de descomposición. En las etapas siguientes se puede obtener mayor flexibilidad teniendo varias alternativas.

### *2.4.1.2. Aglomeración (Asignación)*

Para reducir comunicaciones y explorar la vecindad de los cálculos es conveniente considerar si es útil aglomerar o combinar las tareas identificadas en la fase de descomposición. Puede ser útil también replicar datos para evitar comunicaciones.

El número de procesos producidos en la fase de aglomeración, aunque reducido, aún puede ser mayor que el número de procesadores. Por tal razón, el diseño de un programa paralelo en la fase de aglomeración aún permanece abstracto.

En la fase de aglomeración se persiguen los siguientes objetivos:

- Balancear la carga de trabajo.
  - Límite de la aceleración
  - $Speedup \leq \dfrac{TrabajoSecuencial}{MaximoTrabajoEnCualquier\,\mathrm{Pr}\,ocesador}$
  - El trabajo incluye acceso a datos y otros costos.
  - El trabajo no sólo debe ser repartido equitativamente sino también debe de realizarse al mismo tiempo.
  - Identificar suficiente paralelismo.
  - Decidir cómo manejar la aglomeración.
  - Determinar la granularidad del paralelismo.
  - Reducir las tareas seriales y los costos de sincronización.
- Incrementar la granularidad del cómputo y de la comunicación.
- Retener flexibilidad con respecto a las decisiones de escalabilidad y mapeo.
- Reducir los costos de desarrollo.

Una aglomeración se puede manejar de dos maneras:

- *Estática*
  - La aglomeración se basa en la entrada y en el problema.
  - Los requerimientos de cómputo deben ser predecibles.
  - Es preferible a otros enfoques.
  - No aplicable en ambientes heterogéneos.

- *Dinámica*
  - Para casos en donde la distribución de trabajo o el medio ambiente es impredecible.
  - Se adapta en tiempo de ejecución para balancear la carga de trabajo.
  - Puede incrementar las necesidades de comunicación y reducir la localidad.
  - Puede incurrir en un trabajo adicional para manejar a las tareas.

- Basada en perfiles.

### *2.4.2. Orquestación*

En la fase de orquestación se establecen los patrones de:

- Comunicación.
- Sincronización del programa paralelo.

La arquitectura, el modelo de programación y el lenguaje de programación juegan un papel importante. Se requieren mecanismos para:

- Nombrar y/o identificar datos.
- Intercambiar datos con otros procesadores, comunicación.
- Sincronización entre todos los procesadores.
- Decisiones acerca de la organización de estructuras de datos, la calendarización de tareas, mezclar o dividir mensajes, traslapar cómputo con comunicación.

Algunos objetivos de la orquestación son:

- Reducción de costos de comunicación y sincronización.
- Promover la localidad de referencias a datos.
- Facilitar la calendarización de tareas para evitar tiempos latentes.
- Reducción del trabajo adicional para controlar el paralelismo.

#### *2.4.2.1. Sincronización*

Una buena sincronización de actividades paralelas se logra mediante las actividades de calendarización y mapeo.

- *Calendarización*
  Determinar el momento en que se realiza cada una de las actividades.

- *Mapeo*
  Determinar quién ejecutará el trabajo a realizar.

La calendarización y el mapeo en la fase de orquestación trabajan a nivel de procesos. La calendarización y el mapeo de procesadores se realizan en la fase final del diseño de un programa paralelo.

Durante la orquestación se deben determinar actividades seriales provocadas en la fase de particionamiento. Existen dos aspectos fundamentales que se deben de considerar:

- *Reducir el uso de sincronización conservadora.*
  Utilizar conexión punto a punto en lugar de barreras y cuidar la granularidad de comunicaciones punto a punto.

- *Exclusión Mutua*
  Usar candados diferentes para datos diferentes, candados por tarea en una cola de tareas, no por la cola total y reducir el tamaño de las secciones críticas de código.

### *2.4.2.2. Comunicación*

La comunicación se puede establecer en dos fases:

- *Canales de comunicación entre emisores y receptores*
  Lógicos o físicos

- *Estructura de los mensajes entre emisores y receptores*
  Patrones de comunicación.

Dependiendo de la tecnología de programación, puede no ser necesario crear lógica y explícitamente esos canales de comunicación, ya que definir un canal de comunicación involucra un costo intelectual y transmitir un mensaje por un canal de comunicación implica un costo físico.

La comunicación se puede clasificar como:

- *Local vs Global*
  En la comunicación *local* cada tarea se comunica con un pequeño conjunto de tareas vecinas. En la comunicación global se requiere que cada tarea se comunique con muchas tareas.

- *Estructurada vs No Estructurada*
  En la comunicación estructurada se establecen patrones regulares de comunicación. En la comunicación no estructurada no existen patrones definidos.

- *Estática vs Dinámica*
  En la comunicación estática los participantes no cambian con el tiempo. En la comunicación dinámica los patrones de comunicación se determinan en tiempo de ejecución.

- *Síncrona vs Asíncrona*
  Los participantes de una comunicación síncrona lo hacen de manera coordinada. En la comunicación asíncrona es posible solicitar datos sin el acuerdo de un emisor o productor de datos.

Como objetivos de la comunicación se tiene:

- Se debe balancear las comunicaciones entre todas las tareas.
- Un desbalance en las comunicaciones conduce a un programa no escalable.
- Distribuir estructuras de datos frecuentemente accedidas entre las tareas.

- Siempre que se pueda, se debe tratar de tener comunicaciones locales.
- Las operaciones de comunicación deben promover el paralelismo.

Las operaciones de comunicación deben permitir que las tareas del cómputo se realicen en forma concurrente.

### 2.4.3. *Mapeo y Calendarización*

En la fase de mapeo se hace una asignación de procesos a procesadores físicos. El objetivo es minimizar el tiempo de ejecución de un programa paralelo.

Las estrategias de mapeo son:

- Colocar procesos que son capaces de ejecutarse concurrentemente en procesadores diferentes.
- Colocar procesos que se comunican frecuentemente en el mismo procesador para fomentar la vecindad (espacial y temporal).

El problema del mapeo no aparece en computadoras con un solo procesador o en computadoras con memoria compartida. En multiprocesadores, mecanismos de hardware o del sistema operativo hace una calendarización de procesos automática. No existen, hasta el momento, mecanismos de mapeo de propósito general.

Las interacciones entre procesos puede expresarse por tres modelos diferentes.

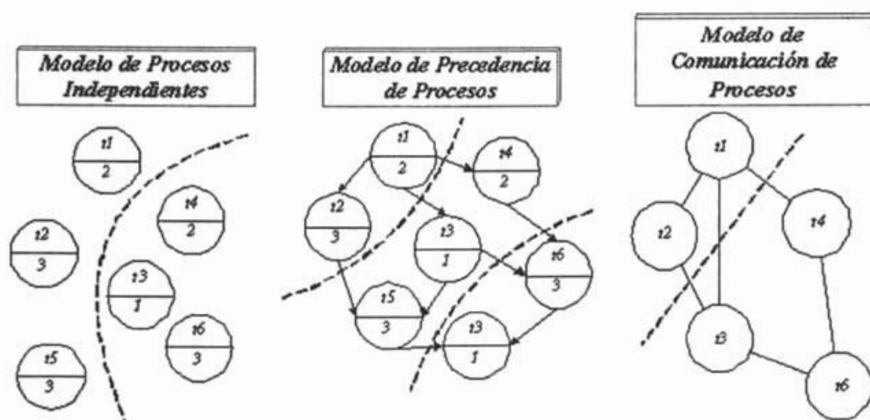


*Fig. 2.7 Modelos de mapeo*

## 2.5. PARADIGMAS DE LA PROGRAMACIÓN PARALELA BASADO EN LA NATURALEZA DEL ALGORITMO

La naturaleza del algoritmo determina el modelo apropiado para su paralelización. Existen dos tipos de algoritmos:

- *Homogéneos*
  Aplican el mismo código a múltiples elementos de datos.

- *Heterogéneos*
  Aplican múltiples códigos a múltiples elementos de datos.

Estas dos alternativas describen diferentes formas de generar tareas que pueden ser ejecutadas en paralelo. La independencia entre las tareas asegura que una tarea no modificará una variable que otra tarea pueda estar leyendo o modificando simultáneamente.

### *2.5.1. Teoría de paralelización homogénea (homoparalelismo)*

Se lleva a cabo este tipo de paralelismo cuando el trabajo realizado por el algoritmo puede ser descompuesto en tareas idénticas, homogéneas, cada una es una porción del trabajo total. En este caso todas las tareas se ejecutarán y terminarán al mismo tiempo y solo tendrán dos puntos de sincronización, al inicio y al final.

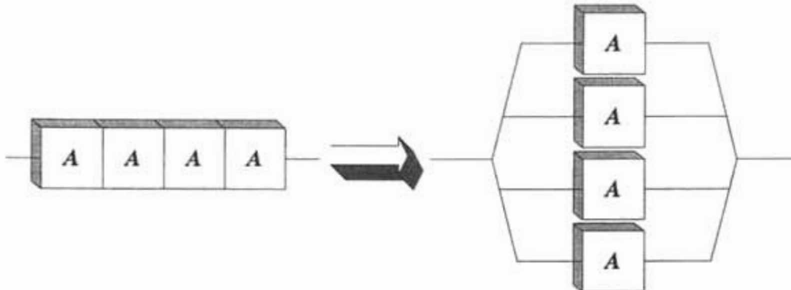


***Fig. 2.8 Paralelización homogénea***

### *2.5.2. Teoría de paralelización heterogénea (hetereoparalelismo)*

Es posible cuando el trabajo realizado por el algoritmo puede ser distribuido en tareas diferentes, siendo cada tarea una porción del algoritmo total. En este caso solo tendrán un punto de sincronización al inicio de las tareas y posteriormente terminarán a diferentes tiempos dependiendo del tamaño de las tareas.

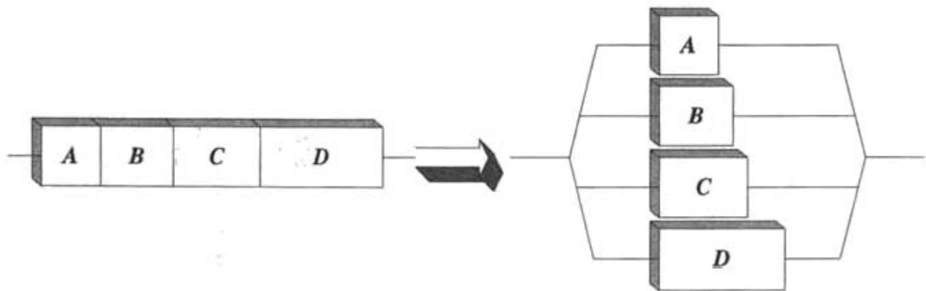


*Fig. 2.9 Paralelización heterogénea*

## 2.6. BIBLIOTECAS DE PROGRAMACIÓN PARALELA

Cuando a una máquina paralela de cualquier arquitectura y un algoritmo adecuado a dicha arquitectura y al problema que deseamos resolver, se requiere una herramienta que nos permita desarrollar un programa para atacar el problema, es decir, una forma de expresar el algoritmo de manera que la máquina pueda ejecutarlo, haciendo uso de las capacidades de paralelismo del equipo.

No se puede esperar que una computadora tradicional *adivine* la tarea que se desea realizar, es necesario indicarlo explícitamente a la computadora por medio de un programa expresado de una manera que puede variar desde el código de máquina hasta lenguajes de alto nivel. En el caso del cómputo científico se prefieren lenguajes de medio o alto nivel, pues permiten una expresividad mayor así como enfocarse en el algoritmo y no en detalles relevantes al hardware u otros aspectos.

De manera similar, dado un programa escrito para una computadora secuencial es imposible suponer que al ejecutarlo en una máquina paralela se aprovechen automáticamente sus capacidades. El algoritmo debe diseñarse y expresarse explícitamente para aprovechar la capacidad de paralelismo en el hardware con que se cuenta.

Se han diseñado lenguajes de programación especializados para arquitecturas paralelas. Sin embargo una desventaja de estos lenguajes es que son útiles únicamente en la arquitectura paralela para los que fueron creados, limitando su utilidad fuera de este ámbito. Como ejemplo se puede mencionar el lenguaje Occam que está diseñado para programar computadoras del tipo Transputer.

Otro enfoque es el de ampliar lenguajes existentes a fin de adecuarlos a una arquitectura paralela. De esta forma se crean variantes del lenguaje original con modificaciones. En cada caso se cuenta con primitivas para hacer uno de paralelismo. Una desventaja es que este enfoque no tiene en cuenta la arquitectura de la máquina donde se está ejecutando, lo cual puede redundar en implementaciones no óptimas.

Finalmente, el enfoque más socorrido es de emplear un lenguaje existente, donde C y Fortran son los utilizados, y recurrir a funciones proporcionadas por una biblioteca para cómputo paralelo. La

popularidad de este enfoque se debe a que unidamente se requiere que el programador se familiarice con algunas nuevas funciones, sin dejar atrás su proficiencia con el lenguaje que utiliza. Además, ya que las funciones de biblioteca son específicas para la arquitectura paralela que se esta trabajando, en la mayoría de los casos se asegura que el rendimiento que se puede obtener de las funciones paralelas de la arquitectura será el máximo posible.

Este enfoque tiene algunas desventajas. Un lenguaje creado específicamente para un arquitectura paralela permite la expresión, de manera natural y como parte de su semántica, de algoritmos paralelos. Por otro lado, en un lenguaje tradicional, el uso de funciones de biblioteca para expresar el uso de las facilidades de paralelismo del sistema se realiza de manera adicional, y por tanto *artificial*, lo cual puede dificultar la concepción y expresión de un algoritmo paralelo. Además, la ventaja de contar con funciones de biblioteca específicas para cada arquitectura puede también convertirse en una desventaja, pues si para cada arquitectura se tiene un juego de funciones diferente, rápidamente puede volverse complicado el dominar todas estas bibliotecas, esto dificulta la transformación del programa de una arquitectura a otra.

### *2.6.1. Biblioteca de paso de mensajes*

Inicialmente la tecnología para producir máquinas paralelas era compleja y regularmente cada una de las compañías que producía esta clase de equipos tomaba un enfoque propio y diferente, ya que el desarrollo en general era cerrado y no existía comunicación y cooperación entre las compañías, cada una desarrolló herramientas para aprovechar las capacidades de los equipos. Dichas herramientas eran compatibles entre sí, si bien algunas de ellas, en particular las correspondientes a equipos MPP que se programaba empleando alguna variante del esquema de paso de mensajes.

Dada su aplicación en el cómputo científico, los principales usuarios de equipos paralelos son las instituciones académicas y laboratorios gubernamentales, particularmente en Estados Unidos.

#### *2.6.1.1. PVM*

Uno de los primeros esfuerzos para crear una biblioteca de paso de mensajes con especificación abierta fue PVM (*Parallel Virtual Machine*). El desarrollo de PVM comenzó en un contexto en donde su naturaleza abierta lo convirtió en un estándar de facto en paso de mensajes.

PVM es un sistema de programación con paso de mensajes portable, diseñado para enlazar varios equipos para formar una máquina paralela virtual, que es un recurso de cómputo único y administrable.

##### *2.6.1.1.1. Historia*

El desarrollo de PVM comenzó en 1989 como parte de un proyecto de investigación sobre cómputo distribuido en ambientes heterogéneos en el Laboratorio Nacional Oak Ridge (ORNL), en Estados Unidos. Como un producto de este proyecto se desarrolló el concepto de una *Máquina Paralela Virtual* y se desarrollaron herramientas de programación internas para realizar experimentos sobre estos conceptos.

En 1991, con la idea de permitir el uso de estos desarrollos por entidades externas al ORNL, se reescribió el sistema PVM apareciendo así PVM 2.0. Con esta versión, el uso de PVM se extendió rápidamente, particularmente entre científicos que se dieron cuenta de la utilidad de este *software* para realizar investigaciones sobre cómputo.

En 1993 se liberó PVM 3.0 tratándose de un rediseño total del software para responder a las necesidades de los ya varios millones de usuarios PVM. En la actualidad PVM es uno de los estándares de paso de mensajes más utilizado y su desarrollo continúa aún bajo el auspicio de los proyectos de cómputo distribuido en el ORLN.

### *2.6.1.1.2. Diseño*

El diseño de PVM se centra alrededor del concepto de la *Máquina Paralela Virtual*, que es una colección dinámica de recursos de cómputo que a través de PVM se puede administrar como un solo sistema paralelo. El concepto de *Máquina Paralela Virtual* es esencial ya que proporciona la base para la heterogeneidad, portabilidad y encapsulamiento de funciones en PVM. Es también el aspecto más único de PVM la capacidad de *agregar* recursos de cómputo de plataformas disímiles en una entidad que permite aprovechar dichos recursos en la realización de una tarea común.

Si bien PVM fue inicialmente diseñado para un ambiente distribuido y heterogéneo, que puede visualizar como una serie de equipos o nodos interconectados por una red local, también está disponible para equipos MPP comerciales como los Intel iPSC, Paragon y CM-5 de Thinking Machines, algunos con arquitectura de memoria compartida de Sequent, IBM, SGI, DEC y Sun, e incluso supercomputadoras como la Cray T-3D[36].

Más aún, si bien la versión pública de PVM se puede ejecutar en todas estas arquitecturas, los fabricantes pueden implementar la API[37] de PVM sobre las funciones de *hardware* específicas de sus equipos, aprovechando sus capacidades y obteniendo mayor rendimiento. Esto se da particularmente en equipos inherentemente multiprocesador, que pueden ya contar con facilidades para intercambio de mensajes de muy alto rendimiento.

Como una consecuencia de su naturaleza multiplataforma, una característica importante de PVM es que permite el desarrollo portable de aplicaciones paralelas con paso de mensajes, utilizando la misma API, para un número considerable de arquitecturas.

El diseño de PVM refleja ciertos principios, obedeciendo a su desarrollo para una *Máquina Paralela Virtual* que pueda estar compuesta por nodos con distintas arquitecturas y el uso del paradigma de paso de mensajes.

PVM proporciona facilidades para crear la *Máquina Paralela Virtual* a partir de uno o más *hosts* disponibles. El usuario tiene la posibilidad de especificar los *hosts* que formarán parte de su

[36] Equipo masivamente paralelo con memoria distribuida.
[37] API = *Application Program Interface*, Interfaz para Programas de Aplicación. El juego de funciones externas que una biblioteca presenta a los programadores para posibilitar el uso de su funcionalidad.

máquina virtual y esta configuración puede modificarse al estar ejecutando el programa; de hecho, un programa para PVM puede por sí solo agregar y eliminar *hosts* de la máquina virtual.

Tradicionalmente, en máquinas masivamente paralelas cada nodo cuenta con exactamente la misma configuración de *hardware* y el mismo tipo de CPU; en las máquinas virtuales en las que se ejecutan los programas en PVM, se tiene la posibilidad de tener nodos con distintas configuraciones y distintas arquitecturas o tipos de CPU. A fin de proporcionar máxima versatilidad y aprovechar estas características, PVM provee lo que se denomina *acceso translúcido* al hardware. El programador tiene la opción de considerar a la máquina virtual como un conjunto de nodos similares, sin atributos particulares; o bien identificar cada nodo, explorando sus capacidades específicas y asignar las tareas a los nodos más apropiados.

PVM va más allá de la portabilidad[38], implementa el concepto de heterogeneidad, es decir, en una aplicación PVM pueden interactuar programas ejecutándose en nodos con arquitecturas diferentes. PVM logra esto empleando tipos de datos opacos y funciones que convierten los tipos de datos específicos de cada nodo al equivalente opaco para su empleo en mensajes que se distribuyen entre los nodos.

Una aplicación de PVM se compone de tareas, *tasks*. La tarea es la unidad de trabajo básica en PVM, un flujo de control independiente y secuencial que alterna entre cálculo y comunicaciones con otras tareas. Las tareas se comunican por medio de paso de mensajes explícito.

##### *2.6.1.1.3. Implementación*

A nivel funcional, PVM está implementado bajo una arquitectura cliente-servidor. Cada nodo ejecuta un demonio[39] *pvmd* que es el servidor, y se encarga de arbitrar los recursos del nodo y comunicarse con el resto de los nodos para formar la máquina virtual.

Los clientes son los programadores de usuario que hacen uso de las facilidades de PVM para aprovechar la máquina virtual. Una aplicación para PVM se compone de uno o más programas secuenciales, normalmente escritos en C o Fortran, que realizan llamadas a las funciones de biblioteca de PVM. Cada programa corresponde a una tarea de la aplicación.

Para ejecutar una aplicación, el usuario debe iniciar el demonio PVM en cada nodo, e indicar al sistema PVM cuáles nodos formarán parte de la máquina virtual. Una vez configurada la máquina virtual, se invoca al programa inicial de la aplicación. Este programa se encarga de iniciar las demás tareas que componen la aplicación. Eventualmente se tiene una colección de tareas que se encontrarán realizando cálculos localmente e intercambiando información por medio de llamadas a las funciones de PVM para resolver algún problema.

De acuerdo a esto, PVM incluye dos componentes de *software* esenciales: el demonio *pvmd* y un juego de bibliotecas que proporcionan funciones para paso de mensajes.

[38] Capacidad de compilar el mismo programa sin cambios en varias arquitecturas distintas.
[39] *Daemon*, es un programa que está en ejecución continua y existe para manejar peticiones de servicio periódicas recibidas por el equipo.

PVM proporciona los siguientes tipos de funciones:

- Paso de mensajes (envió y recepción).
- Empacado y desempacado de mensajes.
- Funciones de agrupación de procesos.
- Control de procesos (iniciar y detener procesos).
- Obtención de información.
- Control de opciones.
- Configuración dinámica de máquina virtual.

#### *2.6.1.2. MPI*

MPI (*Message Passing Interface*), el otro gran estándar de programación con paso de mensajes existente actualmente, es posterior a la aparición de PVM, cuyo hecho contó con algunas características en su diseño que hacen que comúnmente se considere un estándar muy avanzado. A diferencia de PVM, MPI fue diseñado por un comité de académicos e industriales con el fin de convertirse en un estándar de programación con paso de mensajes bien definido.

La meta de MPI es el desarrollo de un estándar ampliamente utilizado para escribir programas con paso de mensajes. Como tal, se busca que la interfaz establezca un estándar práctico, portable, eficiente y flexible para paso de mensajes.

El objetivo principal de MPI es lograr la portabilidad a través de diferentes máquinas, tratando de obtener un grado de portabilidad comparable al de un lenguaje de programación que permita ejecutar, de manera transparente, aplicaciones sobre sistemas heterogéneos; objetivo que no debe ser logrado a expensas del rendimiento.

De manera simple, el objetivo de MPI es desarrollar un estándar para ser ampliamente usado que permita escribir programas usando paso de mensajes. Por lo tanto, la interfaz debería establecer un estándar práctico, portable, eficiente y flexible. A continuación se presentan los objetivos de MPI:

- Diseñar una API.
- Hacer eficiente la comunicación.
- Permitir que las aplicaciones puedan usarse en ambientes heterogéneos.
- Soportar conexiones de la interfaz con C y Fortran 77, con una semántica independiente del lenguaje.
- Proveer una interfaz de comunicación confiable.
- No diferir significativamente de algunas implementaciones tales como PVM, p4, NX, Express, etc., extendiendo la flexibilidad de éstas.
- Definir una interfaz implementable en plataformas de diferentes proveedores sin tener que hacer cambios significativos.

A continuación se presentan las características MPI:



ESTA TESIS NO SALE
DE LA BIBLIOTECA

- *Generales*
  - Los comunicadores combinan procesamiento en contexto y por grupo para seguridad de los mensajes.
  - Seguridad en la manipulación de aplicaciones con hebrado.
- *Manejo de ambiente.* MPI incluye definiciones para:
  - Organización por medio de comunicaciones.
  - Inicializar y finalizar.
  - Control de errores.
  - Interacción con el ambiente de ejecución.
- *Comunicación punto a punto*
  - Heterogeneidad para *buffers* estructurados y tipos de datos derivados.
  - Varios modos de comunicación
    - Normal, con y sin bloqueo.
    - Síncrono.
    - Listo, para permitir acceso a protocolos rápidos.
    - Retardados, utilizando *buffers.*
- *Comunicación colectivas*
  - Capacidad de manipulación de operaciones colectivas con operaciones propias o definidas por el usuario.
  - Gran número de rutinas para el movimiento de datos.
  - Los subgrupos pueden definirse directamente o por la topología.
- *Topologías por procesos*
  - Soporte incluido para topologías virtuales para procesos (mallas y grafos).
- *Caracterización de la Interfaz*
  - Se permite al usuario interceptar llamadas MPI para instalar sus propias herramientas.

A continuación se presentan las condiciones en las cuales se podrá utilizar el estándar MPI:

- *Cuándo usar MPI*
  - Si la portabilidad es necesaria. Si se requiere una aplicación paralela portable, MPI será una buena elección.
  - En el desarrollo de bibliotecas paralelas.
  - La interrelación de los datos es dinámica o irregular y no se ajusta a un modelo de datos paralelos. MPI, y en general el esquema de paso de mensajes, facilitan la programación en estos casos.
- *Cuándo No utilizar MPI*
  - Si es posible usar HPF (*High Performance Fortran*) o Fortran 90 paralelo.
  - Si es posible usar una biblioteca de más alto nivel.

El estándar MPI es extenso en cuanto al número de rutinas que se especifican; contiene alrededor de 129 funciones muchas de la cuales tienen numerosas variantes y parámetros. Para aprovechar las funcionalidades extendidas de MPI se requieren muchas funciones. Esto no implica una relación directamente proporcional entre la complejidad y el número de funciones; muchos programas paralelos pueden ser escritos usando sólo 6 funciones básicas. Sin embargo, MPI permite flexibilidad cuando sea requerida, además de que no se debe ser un experto en todos los aspectos de MPI para usarlo satisfactoriamente.

Es importante notar que MPI es un estándar de *facto*, al igual que PVM. A diferencia de este último, en el diseño de MPI participaron empresas e instituciones importantes en el ámbito del cómputo paralelo, con el objetivo específico de llegar a un estándar que pudieran implementar en sus productos. Por lo tanto, el uso de MPI se ha difundido rápidamente y comienza a reemplazar a PVM como la interfaz de paso de mensajes más utilizada.

### *2.6.1.2.1. Historia*

Antes de 1992, existían varias especificaciones de paso de mensajes dependientes de cada fabricante e incompatibles entre sí, siendo la más popular de estas especificaciones PVM, que está íntimamente ligado a la implementación existente. Sin embargo, fuera de PVM, no existía manera de crear aplicaciones con paso de mensajes que fueran posibles entre distintas plataformas de *hardware*. Aún en el caso de PVM, existían limitantes en cuanto al *hardware* en que se podía ejecutar un programa utilizando PVM, lo cual restringía su utilidad en algunas plataformas.

El proceso de estandarización de MPI ha involucrado a más de 80 personas de 40 organizaciones, principalmente en los Estados Unidos y Europa. Este proceso comenzó en abril de 1992 con el *Workshop on Standards for Message Passing in Distributed Memory Enviroment* patrocinado por el Center for Research on Parallel Computing en Williamsburg, Virginia. En este taller se discutieron las características básicas de una interfaz de paso de mensajes estandarizada y se formó un grupo de trabajo para definir el estándar.

Una versión preliminar del estándar, conocida como MPI 1.0 o MPI1, se presentó en noviembre de ese mismo año en Minneapolis, terminándose la revisión en febrero de 1993. En la conferencia de Supercómputo, en 1993, se presentó dicha versión preliminar del estándar y se constituyó el MPI Forum como un organismo encargado de supervisar la evolución del estándar. El MPI Forum es un foro cuya membresía está abierta a cualquier interesado en el cómputo de alto rendimiento.

Luego de un periodo de comentarios y revisiones, que resultaron en algunos cambios a MPI, la versión 1.0 fue liberada el 5 de mayo de 1994. Comenzando en marzo de 1995, el MPI Forum convino en corregir errores y hacer aclaraciones de la versión 1.0. Tales discusiones resultaron en la versión 1.1 liberada en junio de 1995. Actualmente la especificación más reciente es MPI2. Desde que fue completado en junio 1994, MPI ha ido creciendo tanto en aceptación como en uso. Se han realizado implementaciones en una gran variedad de plataformas que van desde máquinas masivamente paralelas hasta redes de estaciones de trabajo. Incluso, grandes compañías que fabrican supercomputadoras incluyen a MPI como una de sus herramientas fundamentales.

### *2.6.1.2.2. Diseño*

Para el diseño de MPI se buscó implementar las mejores características de algunos sistemas de paso de mensajes existentes, entre ellos PVM; esto a diferencia de procesos habituales en los cuales se selecciona un sistema existente y se adopta como estándar.

MPI busca el diseño de una interfaz para programar la aplicación. El objetivo del proyecto es exclusivamente la definición de una interfaz, sin involucrar en detalles de implementación de la misma. Se buscó que la interfaz sea genérica y versátil a fin de maximizar su audiencia posible. La interfaz debería poder implementarse en equipos de distintos fabricantes, sin requerir cambios significativos en el *software* de sistema y comunicaciones del equipo. Se buscó también que sea eficiente evitando operaciones innecesarias, permitiendo la superposición de comunicación y cálculos y el uso de *hardware* auxiliar de comunicaciones. Sin embargo, la interfaz también debe funcionar eficientemente si no se cuenta con dicho *hardware*. A fin de extender aún más el rango de equipos en que se puede utilizar, la interfaz debe permitir que se realicen implementaciones en ambientes heterogéneos, se buscó que sea semánticamente similar a otras opciones existentes, como PVM, y que no sea independiente de algún lenguaje de programación en particular.

El estándar MPI únicamente proporciona la definición de las interfaces, se deja a cada fabricante la opción de implementar esta especificación de manera más conveniente. De esta forma cada fabricante es libre de aprovechar las facilidades del *hardware* para que se implemente MPI, siempre que se respete la semántica de la interfaz. En este diseño de MPI se tuvo cuidado de mantener compatibilidad semántica con las operaciones que puede realizar el *hardware* de alto rendimiento de algunos fabricantes.

MPI es un API para paso de mensajes, junto con especificaciones tanto semánticas como de protocolo, sobre cómo deben comportase esas características. MPI incluye paso de mensajes punto a punto y operaciones (globales) como *broadcast*, dispersión/recolección (*scatter/gather*) y reducción de datos distribuidos.

El diseño de MPI es orientado a objetos. Dicha orientación es a nivel funcional, ya que MPI no requiere un lenguaje orientado a objetos, de hecho las API's más usadas están en C y Fortran. MPI utiliza extensamente objetos opacos con constructores y destructores bien definidos. Entre los objetos definidos se incluyen los grupos que son los contenedores de procesos fundamentales, los comunicadores que contienen grupos y son utilizados como argumentos para llamadas de comunicaciones y objetos de petición para operaciones asíncronas.

MPI especifica conversión de datos heterogénea y transparente, requiriendo especificación del tipo de datos para todas las operaciones de comunicaciones permitiendo a las implementaciones realizar la conversión a un formato común. Se dice que MPI tiene un diseño fuertemente tipado. La especificación proporciona definiciones para los tipos de datos más comunes, así como posibilidad de especificar tipos de datos nuevos. El requerir la especificación de tipo de datos en los datos predefinidos y de usuario permite la comunicación en ambientes heterogéneos.

Un programa de MPI consta de procesos autónomos, ejecutando su propio código en un esquema MIMD. Los procesos se comunican por medio de llamadas a primitivas de comunicación de MPI. Típicamente cada proceso se ejecuta en su propio espacio de direcciones de memoria, aunque es factible una implementación de MPI en memoria compartida.

La API de MPI proporciona funciones para realizar las siguientes operaciones:

- Organización de procesos: manipulación de grupos y rangos.

- Paso de mensajes.
  - Envió y recepción.
  - *Buffers* de mensajes.
  - Mensajes bloqueantes y no bloqueantes.
- Comunicaciones entre procesos.
  - Organización por medio de comunicaciones.
  - Comunicación entre grupos no relacionados.
  - Comunicaciones de una vía (*Remote Memory Access*): *put* y *get*.
  - Mecanismos de sincronización: *fence* y *lock*.
- Operaciones colectivas.
  - Reducción.
  - Dispersión/Recolección (*Scatter/ Gather*).
- Manipulación de archivos optimizada para máquinas paralelas.

En esta lista se nota la ausencia de funciones para manipulación de procesos, por ejemplo: se mencionó que PVM proporciona métodos para lanzar el programa inicial de la aplicación y dicho programa se encarga de invocar a los demás programas que la componen. MPI no cuenta, estrictamente hablando, con la capacidad para realizar esta clase de manejo de procesos.

Conviene recordar que MPI es únicamente una especificación de las características de una API de paso de mensajes. Al diseñar esta especificación se puso particular énfasis en no dictar detalles de implementación, únicamente en algunos casos el documento del estándar MPI contempla sugerencias a los implementadores.

Así pues, MPI no especifica cómo arrancar y detener procesos, dejando este detalle a cada implementación en particular. Dada la gran cantidad de plataformas en las que corre MPI para cuestiones de implementación es importante dictar un mecanismo estandarizado. En general, MPI evita dictar políticas o mecanismos en instancias en las cuales esto no es factible por no estar definido el comportamiento que se va a tener en la práctica.

### *2.6.1.2.3. Implementación*

Como se vio anteriormente, no existe la implementación de MPI, existe quizá varias decenas de implementaciones que se adhieren a la especificación MPI pero que pueden ser operativamente diferentes. Algunas de las implementaciones del dominio público de MPI son:

- *MPICH* (*MPI/Chameleon*).- Producida por el Argonne Nacional Laboratory y la Universidad del Estado de Mississippi.
- *LAM* (*Local Area Multicomputer*).- Es un ambiente de programación y un sistema de desarrollo sobre redes de computadores heterogéneas. Esta implementación es del Centro de Supercomputación de Ohio.
- *CHIMP* (*Common High-level Interface to Message Passing*).- Desarrollado por El Centro de Computación Paralela de Edinburgo.
- *UNIFY*.- Provee un subconjunto de MPI dentro del ambiente PVM sin sacrificar las llamadas ya disponibles de PVM. Fue desarrollado por la Universidad del Estado de

Mississipi. Corre sobre PVM y provee al programador el un uso de API dual; un mismo programa puede contener código de PVM y MPI.

- *MPICH/NT*.- Es una implementación de MPI para una red de estaciones con Windows NT. Está basada en MPICH y está disponible de la Universidad del Estado de Mississipi.
- *W32MPI*.- Implementación de MPI para un conglomerado con MS-Win32 basado también en MPICH y está disponible del Instituto de Ingeniería Superior de Coimbra, Portugal.
- *WinMPI*.- WinMPI es la primera implementación de MPI para MS-Windows 3.1. Éste corre sobre cualquier PC. Fue desarrollado por la Universidad de Nebraska en Omaha.
- *MPI-FM*.- *MPI*-FM es un puerto de alto rendimiento de MPICH para un conglomerado de SPARCstation interconectadas por Myrinet. Está basado en mensajes rápidos y viene de la Universidad de Illinois en Urbana, Champaign.

El Cluster que se utilizará para realizar nuestro proyecto está implementado bajo MPICH de MPI. MPICH es una de las implementaciones más robustas de MPI habiendo evolucionado junto con el estándar y estando disponible para una gran cantidad de plataformas.

Tradicionalmente, un estándar como MPI involucra un proceso de definición y una vez que la especificación está bien definida se procede a la implementación, existiendo considerable retardo entre la terminación del estándar y la aparición de implementaciones funcionales.

En el caso de MPI, un par de científicos de la División de Matemáticas y Computación del Argonne Nacional Laboratory se ofrecieron como voluntarios, durante la creación del MPI Forum, para realizar una implementación inmediata que siguiera el desarrollo de la implementación y permitiera exponer rápidamente los problemas que la implementación pudiera plantear. Partiendo del *software* existente en el momento, MPICH implementó la primera pre-especificación de MPI en unos cuantos días. MPICH ha seguido el desarrollo de la especificación MPI y actualmente está disponible de manera portable para una gran cantidad de plataformas, entre las que se incluyen sistemas Unix comerciales (Solaris, HP UX, AIX e IRIX), máquinas masivamente paralelas (Intel Paragon y Cray) y variantes libres de Unix (Linux y BSD). MPICH soporta arquitecturas SMP, MPP, redes de estaciones y *Clusters*.

A nivel implementación MPICH proporciona una biblioteca de funciones que implementa la API de MPI. MPICH está diseñado por capas, permitiendo gran portabilidad sin sacrificar el rendimiento. A niveles altos MPICH implementa las funciones de MPI, comunicándose con la capa inferior por medio de una interfaz conocida como *interfaz de canal*. La capa inferior implementa, de manera específica para cada plataforma, funciones para intercambiar información entre procesos según el canal de comunicaciones que se tenga, desde memoria compartida hasta una red local.

MPICH también proporciona los medios para iniciar una aplicación en MPI. Éste es un detalle específico a la implementación. En el caso de MPICH, se proporciona un comando *mpirun*, al que se le puede especificar el número de procesos a iniciar. Éste debe hacerse desde el inicio porque no existen funciones de MPI que permitan iniciar más procesos. El comando *mpirun* encapsula todo el proceso necesario para determinar la arquitectura del equipo en que se está

ejecutando, prepara la interfaz de comunicaciones y lanza los procesos que componen la aplicación.

Las características más importantes de esta librería son:

- Soporta paralelismo del tipo SPMD[40] y MPMD[41].
- La transferencia de mensajes se realiza de forma cooperativa, tanto el proceso emisor como en el receptor participan en el traspaso.
- Permite establecer comunicación punto a punto o comunicación colectiva.
- Proporciona cuatro modos de comunicación:
  - *Standard*.- La emisión de un mensaje se completa una vez que ha sido enviado, haya o no llegado al receptor.
  - *Synchronous*.- El proceso de transferencia no se completa hasta que el emisor recibe la confirmación que el receptor ha recibido el mensaje.
  - *Buffered*.- El mensaje es copiado en un *buffer* del sistema para transmitirlo más tarde si fuese necesario. En este modo de transferencia se completa inmediatamente.
  - *Ready*.- El emisor deja el mensaje dentro de la red de comunicación y supone que el receptor lo estará esperando. Este modo también se completa inmediatamente.

MPICH permite trabajar con los lenguajes:

- C
- C++
- Fortran

### *2.6.1.3. Comparación entre MPI y PVM*

El desarrollo de las principales bibliotecas de computación paralela se ha basado, fundamentalmente en una *máquina virtual* o bien en el *paso de mensajes*.

PVM se desarrolló basándose en el concepto de *máquina virtual*, esto es, una colección de recursos computacionales manejados como una computadora paralela. Por su parte, MPI se centró en el *paso de mensajes*, y aunque no tiene el concepto de *máquina virtual*, sí hace una abstracción de todos los recursos en términos de topología de paso de mensajes.

Mediante la implementación de un algoritmo utilizando tanto PVM como MPI se puede realizar una comparación de características de ambas bibliotecas, desde el punto de vista del desarrollo de programas con ellas.

[40] SPMD = *Single Program, Multiple Data*, Un solo Programa con Múltiples Datos. Es una versión restringida de MIMD en la cual todos los procesadores corren el mismo programa. No como SIMD en donde cada procesador que ejecuta código SPMD puede tomar un camino de control diferente hacia el programa.
[41] MPMD = *Multiple Program, Multiple Data*, Varios Programas con Múltiples Datos. Es un estilo de programación paralela. En muchos casos es fácil convertir un programa MPMD a un solo programa que usa una línea del proceso para invocar a una rutina diferente, haciendo más fácil la programación paralela. Si no es posible convertir de MPMD se debe utilizar el comando *mpich* para ejecutar el programa. Sin embargo, no se podrá utilizar el comando *mpirun* para ejecutar el programa debido a que hay que seguir las instrucciones para cada dispositivo.

La implementación de un algoritmo, utilizando líneas de código fuente efectivas, muestra que MPI es ligeramente más compacto, requiriendo menos código que PVM. Apoyando esta observación, se puede indicar que en general PVM es un poco más laborioso de utilizar que MPI, requiriendo en ocasiones más pasos para lograr el mismo resultado. Por ejemplo, la realización de un *broadcast* en MPI requiere únicamente una llamada a función, mientras que en PVM se requieren dos (una para empacar la información y otra para realizar el envío).

En general, MPI proporciona una API más limpia y mejor planeada. Se requieren un menor número de llamadas a funciones, dichas funciones están mejor organizadas, y es obvio el hecho de que al momento de planear la API se tienen en cuenta la mayoría de las posibilidades de comunicación por paso de mensajes.

PVM es un proyecto con más antigüedad, y esto es obvio en algunas de sus funciones, en particular las funciones de manejo de grupo, dando la impresión de que dichas funciones se agregaron *al vapor* y un tanto sobre la marcha, sin dar mucha importancia a la plantación y enfatizando el lograr una implementación utilizable de la biblioteca. Por otro lado, la API de PVM es un tanto engorrosa, en ocasiones requiriendo un número de llamadas mayor para lograr funciones relativamente sencillas, y algunos comportamientos no están bien documentados.

Cabe mencionar que ambas bibliotecas proporcionan aproximadamente la misma funcionalidad. A continuación se presenta una tabla comparativa donde se describen las tareas a realizar por medio de la biblioteca de paso de mensajes, así como las funciones de MPI y PVM, respectivamente, que realizan dicha tarea.

| Tarea | Función en MPI | Función en PVM |
|---|---|---|
| Inicialización de bibliotecas paralelas | MPI_Init | Implícito |
| Determinación de mi número de proceso | MPI_Comm_rank | pvm_mytid y pvm_joingroup |
| Determinación de número de procesos en mi grupo o comunidad | MPI_Comm_size | pvm_gsize |
| Obtención de nombre de procesador | MPI_Get_processor_name | No aplica |
| Inicialización de buffers de envío | No aplica | pvm_initsend |
| Empacado me mensajes | No aplica | pvm_pk* |
| Desempacado de mensajes | No aplica | pvm_upk* |
| Envió de Broadcast | MPI_Broadcast | pvm_bcast |
| Recepción de Broadcast | MPI_Broadcast | pvm_recv |
| Envío de dato | MPI_Send | pvm_send |
| Recepción del resultado | MPI_Recv | pvm_recv |
| Terminación de sesión paralela | MPI_Finalize | pvm_exit |

*Tabla 2.2 Comparación entre funciones de MPI y PVM*

Hasta la fecha, PVM había venido siendo la biblioteca estándar para computación distribuida. Sin embargo, MPI está siendo cada vez más utilizada, convirtiéndose en el nuevo estándar para este tipo de programación.

φ

# Capítulo III
## Desarrollo de una aplicación de ingeniería

# *Desarrollo de una aplicación de ingeniería*

El mismo término de Clusters se aplica a los conjuntos o conglomerados de computadoras, construidos utilizando componentes de hardware común y software libre; éstos juegan hoy en día, un papel importante en la solución de problemas de las ciencias, las ingenierías y aplicaciones comerciales.

El cómputo en Clusters surge como resultado de la convergencia de varias tendencias que incluyen, la disponibilidad de microprocesadores de alto rendimiento más económicos y redes de alta velocidad, el desarrollo de herramientas de software para cómputo distribuido de alto rendimiento, y la creciente necesidad de potencia computacional para aplicaciones en las ciencias computacionales y comerciales.

Los Clusters han evolucionado para apoyar actividades en aplicaciones que van desde supercómputo y software de misiones críticas, servidores web, comercio electrónico y bases de datos de alto rendimiento.

Por otro lado, la evolución y estabilidad que ha alcanzado el sistema operativo Linux, ha contribuido importantemente al desarrollo de muchas tecnologías nuevas, entre ellas la de Clusters. La tecnología de Clusters de alto rendimiento para linux más conocida es la tecnología *Beowulf*.

Entre las aplicaciones más comunes de Clusters de alto rendimiento se encuentran:

- Aplicaciones científicas y de ingeniería
  - Pronóstico del tiempo.
  - *Procesamiento de imágenes.*
  - Investigación en criptografía.
  - Astronomía.
- Simulaciones
- Aplicaciones comerciales
  - Comercio electrónico (www.amazon.com, www.ebay.com).
  - Bases de datos (Oracle en Clusters).
  - Sistema de soporte para la toma de decisiones (ruteo, optimización).
- Aplicaciones de internet
  - Servicios web / buscadores.
  - Infowares (www.yahoo.com, www.aol.com).
  - ASP's (Applications Service Providers).
  - Correo electrónico, chat electrónico, librería electrónica, banco electrónico, sociedad electrónico, etc.
  - Portales que integren algunas de las aplicaciones mencionadas.
- Aplicaciones críticas
  - Sistemas de control críticos (control de reactor nuclear).
  - Bancos.

- Proyectos militares.

Los beneficios de construir un Cluster se dan en varios aspectos en una variedad de aplicaciones y ambientes:

- Incremento de velocidad de procesamiento ofrecido por los Clusters de alto rendimiento.
- Incremento del número de transacciones o velocidad de respuesta ofrecido por los Cluster de balance de carga.
- Incremento de confiabilidad ofrecido por los Clusters de alta disponibilidad.

Por ejemplo, en el *procesamiento de imágenes* se requiere el manejo de cantidades masivas de datos y cálculos. Al combinar el poder de muchas máquinas del tipo estación de trabajo se pueden alcanzar niveles de rendimiento similares a los de las supercomputadoras, pero a menor costo, pues éstas requieren de hardware y software especializado muy caro.

Otro ejemplo sería el de la situación de un sitio web de mucho tráfico. Si no se cuenta con un plan de alta disponibilidad, cualquier problema menor de una tarjeta de red, puede hacer que un servidor quede completamente inutilizado. Pero al contar con servidores redundantes y servidores de respaldo instantáneos, se puede reparar el problema mientras el sitio funciona con el plan de respaldo, sin suspensión de servicio.

## 3.1. PLANTEAMIENTO DEL PROBLEMA, PROCESAMIENTO DIGITAL DE UNA IMAGEN

El procesamiento de imágenes digitales concierne a un amplio y variado elenco de campos de aplicación. No existe un acuerdo unánime de cuáles son los tópicos que cubre el procesamiento de imágenes digitales y cuáles son sus interrelaciones con otras áreas.

El procesamiento de imágenes tiene como objetivo mejorar el aspecto de las imágenes y hacer más evidentes en ellas ciertos detalles que se desean hacer notar. El procesamiento de imágenes se puede hacer por medio de métodos ópticos, o bien por medio de métodos digitales, en una computadora.

Desde los años sesenta del siglo pasado, el procesamiento de imágenes digitales se ha convertido gradualmente en una de las áreas de investigación científica más importantes. Sin embargo, como cualquier algoritmo de procesamiento de imágenes requiere una vasta capacidad de procesamiento, su desarrollo limitado ha estado en las manos de unos pocos expertos. Pero con el desarrollo rápido de las computadoras, muchas personas han ido apuntando su gran interés por el procesamiento de imágenes. El desarrollo del procesamiento de imágenes está siendo acelerado aún más con el rápido avance de las tecnologías relacionadas con la computación en paralelo, la maximizada capacidad de memoria de los chips y el sistema de visualización en color de alta-resolución.

A mediados de 1980 el costo de un equipo de procesamiento digital de imágenes, era comparable al de un microscopio de polarización o de una herramienta de laboratorio de costo medio. A partir

del advenimiento de las computadoras personales, cambia radicalmente el entorno en el estudio digital de las imágenes. A fines de la década de los 90, del siglo pasado, el mayor desempeño logrado en los procesadores, permite manipular datos digitales con muchas posibilidades en el tratamiento de imágenes. En la actualidad la generación PC's brinda prestaciones más eficientes a los centros de procesamiento de imágenes a un costo muy bajo.

El procesamiento de imágenes digitales, en términos generales, envuelve al reconocimiento de imágenes 2D, 3D y secuencias de imágenes, análisis, manipulación, transmisión y otras áreas relacionadas. Partes cubiertas por esta área son: transformaciones de intensidad y filtros, procesamiento en el dominio de la frecuencia, restauración de imágenes, procesamiento del color, compresión de imágenes digitales, procesamiento morfológico, segmentación, representación y descripción, reconocimiento de formas y objetos e interpretación.

## 3.2. ALTERNATIVA DE SOLUCIÓN, SUAVIZADO DE IMÁGENES

Una imagen digital puede ser definida como una función bidimensional $f(x,y)$, donde $x$ e $y$ son coordenadas espaciales, y la amplitud de $f$ en cualquier par de coordenadas se denomina intensidad o nivel de gris de la imagen en el punto. Cuando $x$, $y$ y los valores de la amplitud de $f$ son todos finitos, cantidades discretas, estaremos ante una imagen digital. Hay que hacer notar que una imagen digital está compuesta de un número finito de elementos, píxeles[42], cada uno de ellos teniendo una particular localización y valor, dependiendo de los valores de $x$ y $y$.

Una vez almacenada en forma de matriz, una imagen digital es susceptible de serle aplicada cualquier transformación que conduzca a mejorar su calidad o acentuar la información que contiene; por ejemplo, los métodos de suavizado de imágenes. El suavizado de imágenes se utiliza para dar una apariencia borrosa o reducir el ruido en imágenes.

Las técnicas de suavizado de imágenes suelen dividirse en dos grandes categorías: los métodos en el dominio del espacio (espaciales) y los métodos en el dominio de la frecuencia (frecuenciales). El dominio espacial se refiere al plano mismo de la imagen y las técnicas en esta categoría están basadas en la manipulación de las píxeles de una imagen. Las técnicas en el dominio de la frecuencia están basadas en la manipulación de la Trasformada Discreta de Fourier de una imagen.

El filtrado es una técnica para modificar o mejorar a una imagen, es decir, puede resaltar o atenuar algunas características. La técnica de filtraje es una transformación de la imagen píxel a píxel, que no dependen solamente del nivel de gris de un determinado píxel, sino también del valor de los niveles de gris de los píxeles vecinos, en la imagen original.

Al realizar la operación de filtrado, el valor de cada píxel dado en la imagen procesada se calcula mediante la aplicación de un filtro, matriz cuadrada de $m \times n$, sobre los valores de los píxeles de la matriz de la imagen original. El filtro se desplaza sobre la imagen de tal forma que el elemento central del filtro coincida con cada uno de los píxeles de la imagen. En cada posición, se multiplica el valor de cada píxel de la imagen, que coincide en posición con un elemento del

[42] La palabra píxel proviene de la abreviatura de *picture element*.

filtro, por el valor de éste y el píxel de la imagen, que coincide con el elemento central del filtro, es substituido por la suma de los productos.

Existen muchos tipos de filtros para realizar el filtrado de una imagen. Los *filtros paso bajas* tienen un aspecto visual de *suavización* y *reducción del número de niveles de gris* de la imagen. Los filtros paso bajas atenúan o eliminan las componentes con altas frecuencias de la imagen indicando que las transiciones abruptas son atenuadas, es decir, los bordes de la imagen se atenuarán. La suavización tiende a minimizar ruidos y origina una imagen menos nítida, con niveles de gris más difuminados. El filtro paso bajas más sencillo e intuitivo es aquel que tiene coeficientes unitarios en todos los elementos.

$$FPB = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

*Fig. 3.1 Filtro paso bajas elemental de 3x3*

- ***Filtro de mediana***

  El filtro de la mediana consiste en definir una matriz de dimensión $n$ x $n$ sobre el píxel filtrado, de tal manera que el valor de la intensidad del píxel de interés en la imagen $f(x,y)$, sea la mediana de los valores de las intensidades de los píxeles de la imagen original que caen dentro del filtro. El filtro de mediana permite eliminar el ruido tipo sal y pimienta, es decir, elimina puntos blancos y negros presentes en la imagen.

  El filtro de mediana utiliza como base el *filtro paso bajas elemental* de dimensión $n$ x $n$ para formar la vecindad[43], conformada por el acoplamiento de la imagen y el filtro, y así aplicar el algoritmo para poder suavizar la imagen.

  Este filtro ordena los píxeles en la vecindad por sus valores y asigna el valor de la mediana (valor en la posición media de los valores ordenados) al píxel central de la vecindad. En una secuencia de números $x_1$, $x_2$, ...$x_N$, la mediana es aquel valor que cumple que $\frac{(N-1)}{2}$ elementos tienen un valor menor o igual a ella y que $\frac{(N-1)}{2}$ tiene un valor mayor o menor que la mediana. La mediana se obtiene ordenando las intensidades de los píxeles de menor a mayor, y el píxel que se encuentra en $\frac{(N-1)}{2}$ es la mediana.

[43] El grupo de píxeles alrededor de un píxel de interés se denomina *vecindad* del píxel considerado. Éstos forman una matriz de valores de píxeles con una dimensión que tiene un número impar de elementos, ejemplo 3x3, 5x5, etc. El píxel de interés es el que se encuentra en el centro.

| | | | | |
|---|---|---|---|---|
| 15 | 17 | 21 | 20 | 21 |
| 20 | 18 | 19 | 17 | 19 |
| 20 | 19 | 6 | 17 | 18 |
| 19 | 19 | 18 | 17 | 19 |
| 19 | 18 | 18 | 19 | 20 |

{6, 17, 17, 17, 18, 18, 19, 19, 19}

*Fig. 3.2 Obtención de la mediana en una vecindad*

Este filtro es particularmente efectivo para remover ruido impulsivo, y cuando los cambios fuertes en nivel de gris característicos de las orillas de un objeto dentro de una imagen, se desean preservar.

Como ocurre con todas las técnicas de mejoramiento y realce de imágenes, es obligado controlar los resultados de aplicar el filtro de la mediana a una imagen específica, ya que en algunos casos pueden ser contrarios a los objetivos buscados. El filtro de mediana tiene muy buenas propiedades de preservación de bordes. Por el contrario, como los bordes tienen un alto contenido de frecuencia, son destruidos por otro tipo de filtros como lo sería un filtro promediador.

- ***Filtro promediador***

  Un filtro promediador se utiliza para el suavizado de imágenes reduciendo la variación de intensidad entre un píxel y el siguiente. Se utiliza para reducir el ruido en las imágenes. El valor de cada celda de un filtro promediador es el promedio aritmético de las celdas en la matriz que conforma al filtro.

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \qquad \frac{1}{25}\begin{bmatrix}1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\end{bmatrix}$$

3x3 5x5

*Fig. 3.3 Ejemplos de filtros promediadores*

El promedio ejecutado por el filtro remueve las características de alta frecuencia, bordes, mientras que permite a las características de baja frecuencia, contraste e intensidad promedio, pasar a través del filtro inalterable. Éste tiene el efecto de suavizar la imagen enfatizando su tendencia de brillo a gran escala.

Si el suavizamiento producido por el filtro es beneficioso o no depende de las características de la imagen de entrada. Si las características de alta frecuencia representan ruidos introducidos que le quitan atractivo a la calidad de la imagen global, el suavizamiento del filtro promediador mejorará la imagen. Sin embargo, el suavizamiento puede manchar los márgenes, degradar el detalle útil de la imagen, y reducir el contraste.

El aumento en el tamaño de la ventana de filtro extiende los efectos del filtro a menores frecuencias espaciales. Para el filtro promediador significa que el grado de suavizamiento aumenta con el aumento del tamaño de la ventana de filtro. El límite entre los rasgos de alta frecuencia removidos y los rasgos de baja frecuencia retuvieron los movimientos hacia las frecuencias bajas.

El filtrado digital de imágenes se basa en la operación de *convolución*[44] entre la imagen y la función de filtro. El cambio de dominio espacial de la imagen al frecuencial permite sustituir las convoluciones por productos, con claras ventajas debido a que se simplifica el cálculo matemático y el tiempo de procesamiento. El filtrado en el dominio de la frecuencia permite mayor flexibilidad al ser posible seleccionar no solo la dirección del filtrado, sino también los intervalos de frecuencia que requieran ser eliminados.

La convolución es una herramienta clave de los cálculos que se realizan para el análisis y extracción de toda la información que contiene una imagen. Intuitivamente podemos mirar a la convolución de dos funciones $f(x)$ y $g(x)$ como la función resultante que aparece después de efectuar los siguientes pasos:

- Girar respecto del origen los valores de una de ellas, es decir $g(z) = g(-z)$ para todo $z$ desde $-\infty$ a $\infty$.
- Ir trasladando la función girada sobre la otra $f(z)g(x-z)$.
- En cada punto $x$ calculamos el valor que resulta de sumar los productos obtenidos de multiplicar para todos los $z$ los correspondientes valores de las funciones $f(z)$ y $g(x-z)$.

En esencia estamos calculando para cada valor de $x$ una especie de valor ponderado de una de las funciones $f(x)$ con los valores de la otra $g(x)$. En el caso de que el área encerrada por la curva de $g(x)$ fuese igual 1 entonces estaríamos calculando para $x$ una media ponderada. Matemáticamente la expresión para esta operación es $f(x) * g(x) = h(x) = \int_{-\infty}^{\infty} f(z)g(x-z)dz$.

De la expresión anterior puede verse cómo para un valor fijo de $x$ los orígenes de las funciones $g$ y $f$ están desplazados justamente en ese valor $x$. Los valores de $f$ para $z$ crecientes van siendo multiplicados por valores de $g$ para $(x-z)$ decrecientes.

En el caso discreto esta visión intuitiva de la convolución queda más clara. La gran importancia de esta operación radica en el hecho de que la transformada de Fourier de un producto de

[44] Es una de las dos operaciones espaciales de mayor importancia en el procesamiento digital de imágenes.

convolución de dos funciones es igual al producto de las transformadas de Fourier de dichas funciones, es decir, $\Im\{f(x) * g(x)\} = F(u) \cdot G(u)$

Este resultado denominado *Teorema de Convolución* implica que podemos calcular un producto de convolución de dos funciones multiplicando sus correspondientes transformadas de Fourier y al resultado aplicarle la transformada de Fourier inversa. En el caso de señales discretas las distintas longitudes que pudieran tener las sucesiones de puntos de cada una de las funciones son posibles causas de errores en el cálculo final de la convolución, es por ello que ambas funciones han de definirse en una misma cantidad de puntos por cada eje.

Para lograr esto consideremos que la función $f(x)$ ha sido muestreada sobre un conjunto de puntos de longitud $A$ y la función $g(x)$ lo ha sido sobre un conjunto de longitud $B$, entonces ambas funciones se rellenarán con ceros hasta que cada una de ellas quede definida en $M = A + B - 1$ valores. La fórmula de rellenar con ceros los valores que faltan no es la única manera que existe de fijar dichos valores aunque sí es la más comúnmente usada. Una vez que las dos funciones tienen el mismo rango de definición la convolución se puede calcular por $f(x) * g(x) = h(x) = \sum_{m=0}^{M-1} f(m)g(x-m)$ para $x = 0,1,...,M-1$.

## 3.3. MEDIOS A UTILIZAR

Algunos programas utilizados para el procesamiento de las imágenes requieren gran capacidad computacional, por ello sería de gran utilidad utilizar un *Cluster* y así intentar reducir el tiempo de procesamiento.

Un Cluster es un *Multiprocesador Masivamente Paralelo*[45] a un bajo costo, en donde sus nodos se conectan por una red como Ethernet.

### *3.3.1. Cluster tipo Beowulf*

Para poder lograr el objetivo de este proyecto es necesario contar con un Cluster. Como se mencionó en capítulos anteriores, un Cluster tipo *Beowulf* está compuesto por equipos y componentes disponibles comercialmente. Dentro de los componentes básicos que forman un Cluster, equipos que fungen como nodos de procesamiento, medios de comunicación entre nodos, software del sistema y aplicaciones, se pueden realizar diversas elecciones que resultan en un sinfín de combinaciones posibles. Obviamente estas elecciones deben de tener en cuenta dos factores básicos: costo y rendimiento. En general, aquellos componentes de mayor costo tienen un rendimiento más elevado.

#### *3.3.1.1. Hardware*

Debido a que el Cluster con el que se cuenta fue realizado por medio de un proyecto de naturaleza académica, en la cual sólo se busca probar teorías y técnicas de los Clusters para

[45] Ver el subtema 1.1.1.3.2 Sistemas de Memoria Distribuida.

obtener un rendimiento mayor al de soluciones uniprocesador, no se contó con un presupuesto para la adquisición de equipo. Por lo tanto, se reunió el equipo que estaba en desuso, evaluar las características y posibilidades del mismo, y de acuerdo a esto se generó una configuración tanto en software como en hardware que permitiera contar con un Cluster utilizable como plataforma de cómputo.

En la construcción del Cluster se consiguió un equipo AMD Athlon con 128 MB en RAM, disco duro de 29.89 GB y dos tarjetas de red. Este equipo se designó como el nodo central del Cluster. Se cuenta con dos tarjetas de red, donde una proporciona el acceso a la red pública[46] y la otra está dedicada exclusivamente a la red de interconexión del Cluster. La red de comunicaciones dedicada es una característica importante de los Cluster tipo *Beowulf.* Desde este equipo los usuarios pueden crear y ejecutar sus programas.

Adicionalmente se consiguieron los siguientes equipos:

- 4 equipos HP Vectra 486/66 con 12 a 16 MB en RAM y tarjeta de red 3Com 3C509.
- 8 equipos HP Vectra 486/50 con 12 a 16 MB en RAM y tarjeta de red 3Com 3C509.
- 2 equipos Dell Optiplex 486/66 con 16 MB en RAM y tarjeta de red 3Com 3C509.
- 1 equipo Digital 486/33 con 16 MB en RAM y tarjeta de red 3Com 3C509.
- 1 equipo Pentium/120 con 16 MB en RAM y tarjeta de red RTL8029.

Estos equipos se utilizan como nodos o elementos de procesamiento en el Cluster. El conocer las características de estos equipos definió la estructura del Cluster así como parte de la configuración de software requerida para el mismo.

#### *3.3.1.1.1. Comunicación entre nodos*

Un elemento básico en todo Cluster es la red o canal de comunicaciones entre nodos. Las características de los equipos con que se cuenta definieron esta elección. Por el tipo de tarjetas de red con las que se contaba, se decidió utilizar una red Ethernet de 10 Mbps para unir a los nodos entre sí. Se empleó un concentrador sencillo así como cableado UTP categoría 5 y conectores RJ-45.

Desde el punto de vista del rendimiento una red Ethernet a 10 Mbps no es una muy buena elección para un Cluster. Los tiempos de latencia son relativamente elevados, andando en el orden de milisegundos. El ancho de banda de 10 Mbps es poco para aplicaciones que requieran un mayor nivel de comunicaciones entre nodos. Y la arquitectura de bus proporciona un canal de comunicaciones compartido con retransmisión que tiende a saturarse muy rápidamente a medida que la cantidad de mensajes pasados entre nodos se incrementa.

Sin embargo, el uso de la red Ethernet tiene ciertas ventajas y características interesantes. Una de ellas es su facilidad de instalación y bajo costo. Por otro lado, la popularidad de la tecnología Ethernet ha llevado a desarrollos que permiten incrementar el desempeño según crezcan las necesidades. Un Cluster puede beneficiarse con el uso de switches que segmentan el tráfico,

[46] Internet.

reducen la saturación y colisiones en la red de comunicación. Y por último se puede contar con incrementos de desempeño inmediatos utilizando Fast Ethernet y Gigabit Ethernet.

### *3.3.1.1.2. Consideraciones para equipos sin disco duro*

Dado que los nodos de procesamiento no cuentan con disco duro, se configuraron como estaciones sin disco duro. El uso de estaciones *diskless*[47], como se conocen comúnmente, está bastante difundido pues permite un desempeño aceptable para terminales que normalmente fungen como despliegue del trabajo realizado en un servidor multiusuario. Las terminales *diskless* requieren un mínimo de trabajo de mantenimiento y configuración a través de un servidor central.

La técnica de arranque *diskless* proporciona ventajas como son la centralización de todos los archivos de los nodos en un servidor central y cierta economía en los requerimientos del equipo, pues evita la necesidad de contar con disco duro en cada uno de ellos.

El uso de esta técnica es una extensión del uso del sistema de archivos NFS[48]. NFS normalmente se emplea para compartir los directorios de usuario en redes de estaciones de trabajo y en un Cluster suele emplearse para facilitar la distribución de los programas a ejecutar. En el Cluster tipo *Beowulf*, los sistemas de archivos de los nodos se encuentran en el servidor central.

En la figura que se muestra a continuación se da una descripción de cómo está acomodado el sistema de archivos para los nodos sin disco en el servidor. Los nodos arrancan desde un disco de 3 ½ configurado con el paquete Etherboot. Con la ayuda de DHCP[49], TFTP[50] y NFS se logra la comunicación de los nodos con el servidor.

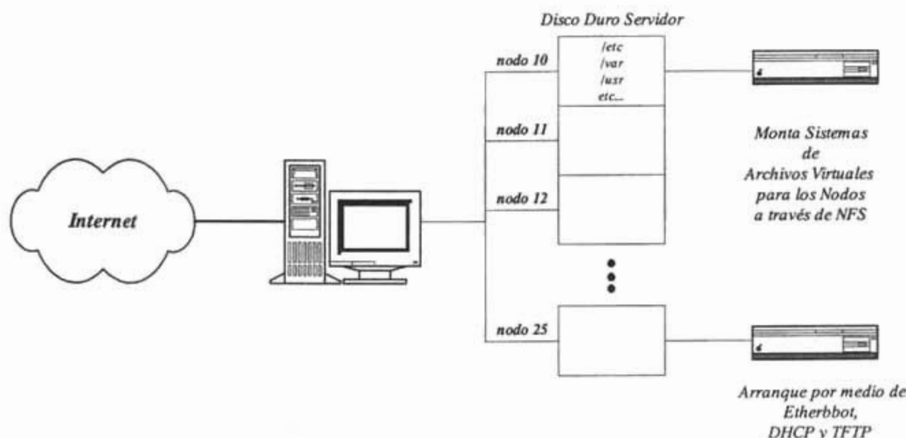


*Fig. 3.4 Sistema de archivos del Cluster*

[47] Sin disco duro.
[48] NFS = *Network File System*, Sistema de Archivos por Red.
[49] DHCP = *Dynamic Host Configuration Protocol*, Protocolo de Configuración de Nodos en forma Dinámica.
[50] TFTP = *Trivial File Transfer Protocol*, Protocolo Trivial de Transferencia de Archivos.

El uso de esta técnica presenta dos desventajas básicas. La primera es el incremento del uso del disco duro en el servidor central. En la configuración final del Cluster se requieren aproximadamente 15 MB de espacio por cada nodo agregado, esto comprende los archivos que no pueden compartirse entre nodos y por lo tanto deben de mantenerse separados, tales como directorios necesarios para el arranque y los archivos de configuración.

La segunda desventaja es un bajo desempeño en el acceso a archivos por parte de los nodos. Como los nodos no cuentan con almacenamiento secundario local, todo intento de acceso a disco se realiza a través de la red, y como la red no es bastante rápida, estos accesos pueden tomar bastante tiempo. El hecho de que el acceso a archivos es lento para los nodos debe tomarse en cuenta al momento de diseñar los programas a ejecutar en el Cluster y se debe tener precaución con el acceso a archivos en los procesos que se ejecutan en los nodos.

En general, la consideración en cuanto al desempeño del acceso a archivos tendrá un impacto que debe tomarse en cuenta en el overhead de arranque de los procesos. Si se diseñan cuidadosamente los programas esto no repercutirá en el desempeño durante la realización de cálculos.

### *3.3.1.1.3. Integración de hardware*

Habiendo tenido en cuenta los factores anteriores, la integración del hardware fue sencilla. El Cluster se ensambló en la instalación compartida del Laboratorio de Telemática y el Grupo de Usuarios de Linux de la Facultad de Ingeniería.

De los equipos que componen el Cluster, uno cuenta con gabinete estilo minitorre. Los 16 restantes tienen gabinetes de tipo escritorio. El equipo minitorre se colocó en el centro como servidor central. Los nodos tipo escritorio se organizaron en dos torres a cada lado del servidor central. El monitor y teclado del servidor se ubican al centro.

*Fig. 3.5 Servidor central*

*Fig. 3.6 Vista frontal del Cluster*

Los concentradores para la red Ethernet forman la columna vertebral del mecanismo de conexión. Se requirieron 17 nodos, por lo cual se utilizaron dos concentradores: uno de 16 nodos y el otro de 8 puertos, los cuales se colocan en cascada. En los concentradores se realiza la conexión de los nodos.

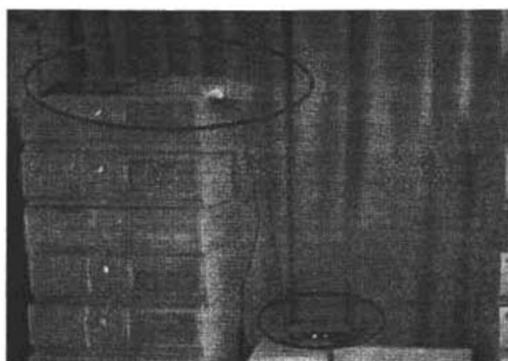

*Fig. 3.7 Vista de los concentradores*

Para la conexión eléctrica se utiliza un regulador de corriente al que se conectan tres multicontactos. Cada multicontacto cuanta con 6 enchufes polarizados y aterrizados de manera que se tiene la posibilidad de conectar 18 nodos. El monitor del servidor central se conecta en el espacio restante en el regulador que cuenta con 4 contactos.

*Fig. 3.8 Aspecto final del Cluster*

### *3.3.1.2. Software*

Partiendo de la plataforma de hardware descrita anteriormente, la instalación y configuración de todo el software que se implementó en el sistema puede visualizarse en las siguientes etapas:

- Instalación y arranque del sistema operativo en el servidor central.
- Instalación y configuración de software de iniciación en los nodos.
- Configuración de servidor y nodos para operación con NFS y NIS[51].
- Configuración y organización de sistemas de archivos individuales para los nodos.
- Configuración y organización de área de archivos compartidos.
- Instalación y configuración de software requerido para aplicaciones paralelas.

El sistema operativo en el servidor central sirvió como base para la creación de los directorios de los sistemas de archivos para los nodos. Este servidor cuenta con el software para proporcionar los servicios requeridos para el arranque y operación de los nodos con la configuración *disklees*. Se cuenta con el software para la programación paralela así como herramientas de desarrollo, editor de texto, depurador y compilador, para que el usuario pueda desarrollar en el servidor los programas a ejecutar en el Cluster.

Uno de los puntos clave para la elección de Linux como sistema operativo para este Cluster fue el hecho de que todo el software requerido estaba disponible. Algunos de estos componentes de software son elementos que se desarrollaron para permitir que Linux fuera una alternativa viable para el uso de redes tradicionales. Algunos otros componentes de software fueron desarrollados por el proyecto *Beowulf* para el fin explícito de tener un Cluster basado en Linux.

Entre las distribuciones de Linux que se pueden utilizar se eligió la distribución Red Hat versión 6.2. Esta elección se debe a que dicha distribución es una de las más establecidas de Linux en

[51] NIS = *Network Information System*, Sistema de Información de Red.

Internet[52] y cuenta con el sistema RPM[53]. Además era la que más soportaba el hardware del Cluster.

Se crearon dos particiones principales: una partición pequeña que contiene el cargador de arranque y el *kernel*, que se montó bajo el directorio */boot*, y una partición que abarca el resto del disco duro y se montó bajo el directorio raíz /. Además se especificó una partición de intercambio de memoria virtual requerida para el funcionamiento del sistema.

El programa de instalación permitió instalar una serie de grupos de paquetes que están organizados dependiendo de su funcionalidad, los cuales son:

- X Windows System.
- GNOME.
- Mail/WWW/News tools.
- Graphics Manipulation.
- Networked Workstation.
- NFS Server.
- Authoring/Publishing.
- Emacs.
- Development.
- Kernel Development.
- Clustering.
- Utilities.

Algunos paquetes que se agregaron explícitamente son:

- DHCP
  Para la configuración de la primera interfaz se eligen los datos de acuerdo a la red externa en la que se encuentra el servidor. Para este caso se utilizaron los siguientes datos, según se asignaron al servidor por el administrador de la red externa:

| | |
|---|---|
| ***IP Addreess*** | 192.168.1.3 |
| ***Netmask*** | 255.255.255.0 |
| ***Default Gateway*** | 192.168.1.1 |
| ***Nameserver*** | 132.248.10.2 |
| | 132.248.204.1 |

  Arbitrariamente se seleccionó la subred 192.168.10.0 con máscara de red de 24 bits, 255.255.255.0. Esto proporciona 253 direcciones IP utilizables de la 192.168.10.1 a la

[52] La distribución de Red Hat versión 6.2 se obtuvo de manera gratuita a través de Internet, por eso se dice que es de una de más establecidas en la red.
[53] RPM = *Red Hat Package Manager*, Sistema de Administración de Paquetes. Este sistema maneja la instalación y desinstalación automática de software del sistema, a diferencia de algunas distribuciones que requieren obtener los programas en código fuente, compilarlos e instalarlos manualmente.

192.168.10.254. Véase que esta subred está dentro de las redes especificadas como privadas, para uso interno, en el RFC 1918[54].

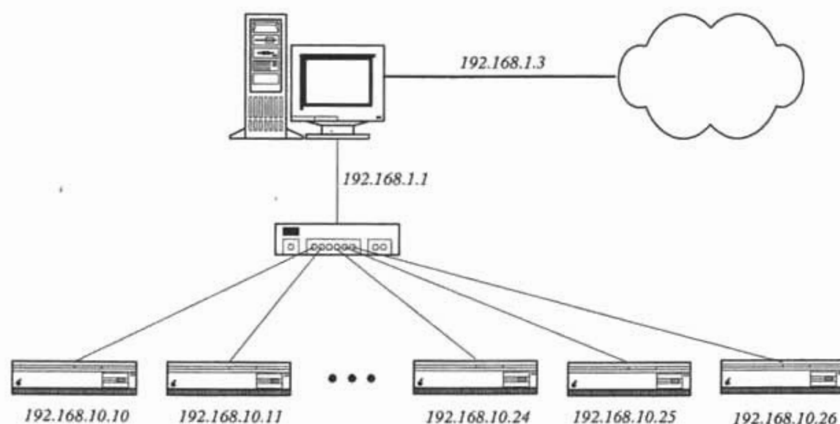


***Fig. 3.9 Esquema de direccionamiento utilizado***

El arranque remoto de estaciones sin disco duro involucra 4 etapas:

- Al arrancar la computadora se carga un programa conocido como *arrancador de red*. Éste es un programa que tradicionalmente reside en una ROM de arranque que se encuentra en la tarjeta de red.
- El arrancador de red obtiene su dirección IP de un servidor utilizando los protocolos BOOTP[55] y DHCP. Con los datos entregados por el servidor, el arranque de red realiza configuración básica de la tarjeta de red para hacer transferencias TCP/IP.
- El arrancador de red utiliza el protocolo TFTP para transferir un archivo desde el servidor, cuya ubicación normalmente se especifica como parte de la configuración recibida por BOOTP y DHCP. Este archivo comúnmente es el *kernel* que se debe cargar la estación para realizar su arranque.
- Una vez cargado el *kernel*, termina el trabajo de arrancador de red. El *kernel* se carga normalmente y realiza su procedimiento de inicio.

Tanto el protocolo BOOTP como el DHCP permiten la asignación de información de configuración para estaciones de trabajo desde un servidor central. En ambos casos el cliente realiza una transmisión de *broadcast* con su MAC[56]. El servidor BOOTP o DHCP toma esta petición y regresa al cliente la información requerida, que básicamente consta de la dirección IP que el cliente deberá utilizar. El Cluster utiliza el servidor de DHCP debido a que es un protocolo

[54] Rekhter, Moskowitz, Karrenberg, de Groot, Lear.
***RFC 1918:*** *Address Allocation for Private Internets*
Copyright 1996.
[55] BOOTP = *Bootstrap Protocol*, Protocolo de Arranque.
[56] MAC = *Media Access Control*, Control de Acceso al Medio. Todo dispositivo de red debe contar con una dirección MAC única para identificación.

más sofisticado y su configuración es más clara que la de BOOTP. El DHCP proporciona la posibilidad de enviar más información al cliente que BOOTP y cuenta con algunas características como asignación dinámica de direcciones.

El protocolo TFTP es un protocolo basado en UDP, que permite bajar archivos de un servidor. Su principal utilidad es proporcionar archivos de arranque a equipos que no cuentan con almacenamiento local. TFTP no cuenta con ninguna clase de control de acceso, contraseñas o nombres de usuarios.

El programa encargado de iniciar la interfaz de red obtiene los datos de configuración básicos y carga por medio de TFTP el archivo especificado en esta configuración. El cargador de arranque libre que se utilizó fue Etherboot. Etherboot utiliza manejadores internos y genera una imagen ROM para cada tipo de tarjeta de red soportada. El uso de manejadores internos, desarrollados con autoconfiguración para tarjetas de tipo ISA, permite que la imagen tenga un tamaño muy reducido que no da problemas con ninguna tarjeta de red soportada. La versión de Etherboot utilizada fue la 4.6.7.

Cada nodo requiere un sistema de archivos raíz que utiliza para el arranque. Estos directorios se exportan a través de NFS y deben contener los archivos necesarios para el arranque del sistema. El servidor de NFS permite acceder a archivos ubicados en sistemas remotos tal como si se encontraran localmente. En este caso es de gran importancia ya que a través de NFS se proporcionan los sistemas de archivos raíz y un área compartida para los nodos del Cluster. El demonio NFS, que se debe de habilitar, requiere un archivo de configuración que le indique qué sistemas de archivos y directorios debe exportar o hacer disponibles, así como varios parámetros que controlan el acceso que los clientes tendrán a estos sistemas de archivos. Los directorios que se exportan son:

- /bin
- /dev
- /etc
- /lib
- /proc
- /root
- /sbin
- /tmp
- /var

Como acompañante de NFS, Sun Microsystem desarrolló el protocolo NIS. NIS permite compartir información mantenida en un servidor central y que se hace accesible a los clientes, de una manera que se presta idóneamente para sincronizar, entre otras cosas, la información del archivo */etc/passwd*. En la arquitectura NIS se tiene entidades desconocidas como dominios. Para cada dominio, existe un servidor maestro que es el que mantiene la información autoritaria y comparte con los clientes. Los clientes se unen al dominio localizando al servidor correspondiente y realizando peticiones cuando necesitan la información requerida.

NIS proporciona facilidades de acceso con base en datos sustentados en llaves. Se observa que una propiedad interesante de los archivos que se pueden compartir por NIS es que regularmente el acceso se realiza por medio de una llave, especificando un *uid*[57] o nombre de usuario. NIS también se utiliza para compartir la tabla de *hosts* del archivo */etc/hosts*. Dando como llave un *hostname*, el sistema NIS podría devolver su dirección IP y viceversa.

A fin de que un nodo cliente pueda compartir la información de un servidor NIS, se requiere ejecutar un programa, *ypbind*, que lo enlace al dominio NIS correspondiente de modo que cuando algún programa solicite información de las bases de datos compartidas, ésta pueda obtenerse del servidor NIS y no de los archivos locales. De esta manera se asegura que existe consistencia de información entre los clientes del dominio NIS, en particular para asegurar que la información de los permisos sea la misma para todos los nodos.

El archivo */etc/hosts* contiene la lista de mapeos de nombres a direcciones IP. Esta información es necesaria para la correcta operación del sistema. Adicionalmente este archivo es uno de los archivos compartidos a través de NIS de modo que únicamente se necesita modificar en el servidor central para que todos los nodos tengan la misma información

Después de la instalación del sistema operativo y algunos paquetes con su debida configuración, en este momento los nodos están listos para realizar su arranque.

Las bibliotecas de programación paralela que se agregaron al Cluster fueron tanto MPICH y PVM, las cuales tienen una licencia que permite la distribución libre y la modificación de su código. Para la instalación y configuración de PVM y MPICH se requiere que esté configurada la ejecución de programas remotos de *rsh*[58] en cada nodo. La página de internet de PVM, mantenida por el *Oak Ridge Nacional Labs*, se encuentra en *http://www.epm.ornl.gov/pvm/*. Aquí se encuentra documentación, información miscelánea y un enlace a la distribución de PVM 3.4.3 en código fuente que se encuentra en *http://www.netlib.org/pvm3/pvm3.4.3.tgz*, la cual se instaló en el Cluster. El paquete RPM utilizado en la instalación se obtuvo en: *ftp://ftp.rpmfind.net/linux/redhat/6.2/en/os/i386/RedHat/RPMS/pvm-3.4.3-4.i386.rpm*. Esta versión de PVM se liberó el 29 de marzo del 2000.

La versión que se instaló de MPICH fue la 1.2.2 y fue liberada el 20 de agosto del 2001. La página en Internet de MPICH, mantenida por *Argonne Nacional Labs*, se encuentra en *http://www.-unix.mcs.anl.gov/mpi/mpich*. Junto con toda la documentación en particular de MPICH así como enlaces a la definición del estándar MPI, se encuentra un enlace a la distribución en código fuente de MPICH en *ftp://ftp.mcs.anl.gov/pub/mpi/mpich.tar.gz*. El paquete RPM utilizado en esta instalación se obtuvo en: *ftp://ftp.rpmfind.net/linux/redhat/6.2/en/powertools/i386/i386/mpich-1.2.0-5.i386.rpm*.

[57] Cada usuario es identificado por un valor numérico conocido como *uid* o *User id*.
[58] RSH = *Remote Shell*, Shell Remoto. Permite la ejecución de algún comando arbitrario en un equipo remoto. La sintaxis en su forma más básica es la siguiente: # rsh usuario@servidor comando. De esta manera se ejecuta *comando* en un equipo llamado *servidor* como el usuario *usuario*. A fin de permitir la ejecución de comandos remotos por parte de algún otro equipo cada nodo debe tener un archivo */etc/hosts.equiv*. Aquí se colocan los nombres o direcciones de los equipos remotos a los que se dará autorización para ejecutar comandos.

## 3.4. DISEÑO DE LA APLICACIÓN EN PARTICULAR

El suavizado de una imagen es una técnica para modificar o mejorar a una imagen. La técnica de suavizado de imagen es una transformación de la imagen píxel a píxel, que no dependen solamente del nivel de gris de un determinado píxel, sino también del valor de los niveles de gris de los píxeles vecinos, en la imagen original. Al realizar el suavizado de una imagen es necesario hacer una operación de filtrado.

*Fig. 3.10 Estructura general de un filtrado*

El diagrama anterior lo tomaremos como base para poder elaborar nuestro programa que realiza el suavizado de una imagen.

Los pasos de diseño para el suavizado de imágenes que se presentan a continuación tratan de cubrir las etapas generales del proceso de elaboración del programa paralelo que se utilizará para dar solución al problema que se plantea durante la elaboración de este proyecto. En este caso se cubre la elaboración de un programa que nos permita llevar a cabo el suavizado de una imagen.

El primer paso consiste en ubicar el mayor número de tareas, con el fin de poder realizar un particionamiento. En este caso los datos que pasan por el proceso de descomposición son:

- Los datos de entrada (imagen y filtro).

Además se lleva a cabo un proceso de descomposición de funciones o tareas. De una manera general podemos definir estas tareas de la siguiente manera:

- Lectura de la imagen.
- Generación del filtro.
- Transformada discreta de Fourier.
- Transposición de matrices.
- Multiplicación de matrices.
- Transformada discreta de Fourier inversa.
- Generación imagen suavizada.
- Guardar datos en memoria.
- Obtener datos de memoria.

Posteriormente tratamos de reducir el número de tareas seriales a realizar, esto se logra mediante la repartición de tareas en cada uno de los procesadores, buscando la igualdad entre tareas a realizar por cada procesador y además buscando que dichas tareas se realicen al mismo tiempo.

Dentro del programa que se realizará se podrá observar esta etapa cuando el servidor central del Cluster repartirá los datos de entrada entre los diferentes nodos, donde cada nodo deberá efectuar una operación matemática y posteriormente éstos deberán enviar de regreso el resultado de la operación realizada al servidor central para que éste pueda reorganizar los datos obtenidos por los nodos. La comunicación entre el servidor central del Cluster y los nodos que forman a éste estará dada en forma estática y síncrona.

El lenguaje de programación a utilizar juega un papel muy importante debido a que mediante éste se llevará a cabo el intercambio de datos entre los diferentes procesadores. Dentro del programa a realizar podremos observar la importancia de este paso, ya que para poder obtener un mejor desempeño es importante mantener todos los datos a intercambiar sincronizados. La sincronización de datos es importante debido a que no se podría dar inicio a la siguiente tarea sin antes tener realizada por completo la tarea anterior.

A continuación se presenta un diagrama que refleja lo expuesto anteriormente.

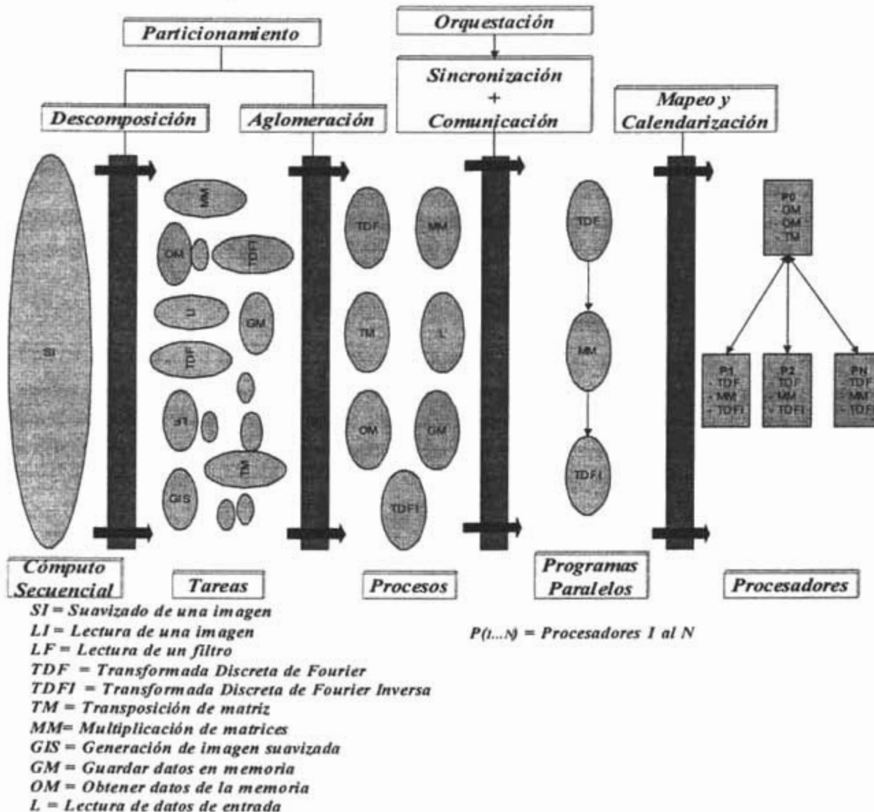


*Fig. 3.11 Pasos para paralizar el suavizado de una imagen*

### *3.4.1. Lectura de la imagen*

Una imagen es una función continua *f(x,y)*, que expresa la variación en el espacio de la cantidad de luz emitida o reflejada. Esta cantidad de luz o energía procedente de la imagen es lo que comúnmente denominamos brillo, intensidad o nivel de gris de los distintos objetos o detalles contenidos en la imagen.

Lo que se conoce como formación de una imagen digital o digitalización de una imagen, no es más que la conversión en una serie de valores discretos de la función continua *f(x,y)* por muestreo tanto del plano espacial como de la intensidad. Así, una imagen digital se representa como una tabla de valores o matriz bidimensional, en la que cada elemento corresponde al valor de intensidad de cada píxel de la imagen.

Generalmente las imágenes con niveles de gris se codifican de tal forma que cada píxel ocupa un byte[59] de memoria, es decir, que cada píxel puede tener un valor entero entre 0 y 255. Sin embargo algunos sistemas actuales utilizan otro tipo de codificaciones más comprimidas (1, 2 o 4 bits por píxel) o de mayor resolución (10 o 12 bits por píxel).

Las dimensiones virtuales de la imagen, número de píxeles, dependerán del número de filas y columnas que tenga la matriz que la define. Así una imagen de 600 por 500 píxeles, corresponderá una matriz de 600 columnas y 500 filas, con un tamaño total de 300,000 píxeles.

Un formato de imagen es una forma normalizada de almacenar los datos que constituyen la misma, de modo que los datos almacenados por un sistema puedan ser leídos por cualquier otro sistema. La variedad de formatos existentes y las diferencias entre ellos provienen, aparte de las razones comerciales o históricas, por los diferentes fines para los cuales almacenamos una imagen y por los diferentes medios de presentación de las mismas.

- ***RAW***

  Este formato es el que almacena la imagen en bruto, es decir, la imagen está sin procesamiento y generalmente sin formato, no teniendo información de ***header*** (***cabecera***). Su tamaño es bastante grande debido a que se trata de información sin procesar y requiere sistemas de almacenamiento de gran capacidad. Éste es el formato de almacenamiento más simple.

  Este formato es de fácil lectura y escritura y son almacenados en 8 bits por píxel, variando de 0 a 255 niveles de gris. Así, si tenemos una imagen de niveles de gris en la que cada píxel se representa con un byte, una imagen *128 x 128* se almacenaría en un archivo de 16,384 bytes. El archivo, desde el punto de vista del programador es un archivo binario.

[59] Unidad de información utilizada por las computadoras. Cada byte está compuesto por ocho bits.

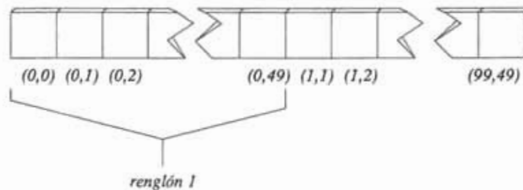


*Fig. 3.12 Almacenamiento de una imagen de 100 x 50 píxeles en de niveles de grises formato raw*

- ***TIFF (Tagged Image File Format)***

  Es el resultante de procesar la imagen una vez que ha sido captada por el sensor CCD pero sin compresión, por lo que no existe pérdida de calidad. Para su visualización no es necesaria la intervención de programas especiales ya que se trata de un estándar. Este formato es el utilizado en los medios editoriales ya que permite su tratamiento múltiples veces al no existir pérdidas de calidad. Al no existir compresión, es necesario tener gran capacidad de almacenamiento.

  TIFF es un formato muy flexible con o sin pérdida. Los detalles del algoritmo de almacenamiento de la imagen se incluyen como parte del archivo. En la práctica, TIFF se usa casi exclusivamente como formato de almacenamiento de imágenes sin pérdidas y sin ninguna compresión. Consecuentemente, los archivos en este formato suelen ser muy grandes.

- ***JPEG (Joint Photographic Experts Group)***

  Es el más utilizado en el mundo de la fotografía digital por que los archivos resultantes son de menor tamaño. Esto es debido que se trata de un algoritmo de compresión de imagen, que además podemos regular a nuestro gusto, en función de la calidad que queramos.

  El inconveniente es que no se pueden editar en muchas ocasiones ya que cada vez que lo hagamos perderemos calidad. No requiere grandes sistemas de almacenamiento dado que el tamaño de los archivos es bastante reducido. El más recomendable para los usuarios domésticos.

Entre los formatos de imágenes, anteriormente explicados, vamos a utilizar el formato *raw* para poder leer la imagen que vamos a suavizar. Escogemos el formato raw por que es el más simple de todos ya que no se guarda información acerca de la imagen, tan solo se guardan los valores de luminosidad y además se puede leer como un archivo binario.

Una restricción que tendrá la imagen que se leerá radica en que su dimensión será *256 x 256* o menor. Esta restricción está dictada por la memoria disponible en los nodos, para aquellos que cuentan con únicamente 12 MB de memoria debido a que no es posible el manejo de una imagen más grande. Otra restricción que tendrá la imagen es que ésta contendrá ruido gaussiano para poder observar un resultado para la aplicación del filtro que se aplicará.

En el apartado del Apéndice, se pueden observar un programa que se realizó como prueba[60] para la lectura de una imagen tipo *raw*. Posteriormente se modificó para poder dar pauta a una parte del programa que utilizamos para suavizar la imagen.

### *3.4.2. Selección del filtro*

Como vimos anteriormente, los filtros existentes para poder realizar el suavizado de una imagen son los filtros paso bajas, entre los que se encuentran el promediador y el de mediana. Para determinar cuál es el filtro que vamos a utilizar, debemos saber cuáles son las características de la imagen.

Dado que la imagen a leer tendrá ruido, el filtro que más se adecua a nuestra necesidad es el promediador debido a que:

- El suavizado de la imagen no depende de las características de la imagen.
- Atenúa ruido aditivo Gaussiano de manera efectiva.
- Mejora la imagen, debido a la presencia del ruido.

Otra razón por la cual se escogió el filtro promediador fue por que simplifica el cálculo matemático y el tiempo de procesamiento.

### *3.4.3. Transformada de Fourier*

La Transformada de Fourier es una operación matemática importante en la materia que nos ocupa. La comprensión de su operativa nos proporciona la posibilidad de incursionar en:

- Proceso de Imágenes.
- Filtros.
- Convolución.
- Obtención de Espectros.

La transformada de Fourier de una función continua e integrable de una variable real $x$ se define por $F(u) = \int_{-\infty}^{\infty} f(x)e^{-2\pi iux}dx$. Observemos que la transformada de una función real es una función compleja, es decir, $F(u) = R(u) + I(u)i$ donde $R(u)$ e $I(u)$ son la parte real e imaginaria de $F(u)$, respectivamente. La variable u recibe el nombre de variable de frecuencia.

La transformada de Fourier de una función discreta de una variable real $x$ se define por $F(u) = \sum_{n=0}^{N-1} f(x)e^{-\frac{2\pi iux}{N}}$, donde $N$ es el número de elementos que conforma a la función *f(x)*, es decir, la longitud del rango de los valores que puede tener la función *f(x)*. Mientras que la

[60] Ver el Apéndice, sección B Lectura de una imagen con formato *raw*.

transformada de Fourier inversa de una función discreta de una variable de frecuencia $u$ se define por $f(x)=\sum_{u=0}^{N-1} F(u)e^{-\frac{2\pi i x u}{N}}$.

### 3.4.3.1. *Transformada de Fourier en matrices*

En el tratamiento de imágenes, la transformada de Fourier bidimensional toma un papel fundamental. En una imagen, cada píxel tiene coordenadas de posición *(x,y)* y se le asocia un valor de brillo determinado. El brillo o intensidad de la imagen dependerá del punto donde se la mida por lo tanto se podrá definir una función intensidad de las coordenadas *(x,y)*: *f(x,y)*, función de dos dimensiones.

En el caso de funciones de dos dimensiones el par de transformadas de Fourier discretas vendrán dadas por las siguientes expresiones: $F(u,v)=\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-2i\pi(\frac{ux}{M}+\frac{vy}{N})}$ para $u=0,1,\ldots,M-1$ y $v=0,1,\ldots,N-1$, y $F(x,y)=\frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} f(u,v)e^{2i\pi(\frac{ux}{M}+\frac{vy}{N})}$ para $x=0,1,\ldots,M-1$ y $y=0,1,\ldots,N-1$.

En las expresiones anteriores hemos considerado que usamos un muestreo con distinto paso en la dirección de *x*, $\Delta x$, y en la dirección de *y*, $\Delta y$. Así mismo el número de muestras tomadas según la dirección de *x* es *M* y el número siguiendo la dirección de *y* es *N*. De igual forma se puede aplicar a la función $F(u,v)$.

- ***Separabilidad***

  Esta propiedad de la transformada de Fourier discreta está relacionada con la posibilidad de calcular la transformada de Fourier discreta de una función bidimensional como una combinación de dos transformadas de Fourier discretas, calculando primero una transformada de Fourier discreta sobre la variable de uno de los ejes y al resultado aplicarle de nuevo la transformada de Fourier discreta sobre la variable del otro eje.

  La ventaja que aporta esta propiedad es el hecho de poder obtener la transformada *f(x,y)* o la inversa *f(x,y)* en dos pasos, mediante la aplicación de la transformada de Fourier en una dimensión.

  La transformada de Fourier discreta bidimensional se puede escribir de forma separada como: $$F(u,v)=\sum_{x=0}^{M-1} e^{-2i\pi(\frac{ux}{M})}\sum_{y=0}^{N-1} f(x,y)e^{-2i\pi(\frac{vy}{N})}$$ y

  $$F(x,y)=\frac{1}{MN}\sum_{u=0}^{M-1} e^{2i\pi(\frac{ux}{M})}\sum_{v=0}^{N-1} f(u,v)e^{2i\pi(\frac{vy}{N})}.$$

En el apartado del Apéndice, se pueden observar un programa que se realizó como prueba[61] para realizar la transformada de Fourier discreta en dos dimensiones. Posteriormente se modificó para poder dar pauta a una parte del programa que utilizamos para suavizar la imagen.

### *3.4.4. Multiplicación de matrices*

La multiplicación de matrices no es conmutativa. Ésta sólo se puede realizar si las matrices cumplen cierta condición. Si el número de columnas en la matriz $B$ es igual al número de renglones en la matriz $A$, las matrices se denominan conformables, y pueden multiplicarse en el orden $B \times A$. Específicamente, si $B$ es una matriz de dimensiones $(q, n)$ y $A$ tiene dimensiones $(m, p)$ el producto $BA$ será una matriz $(q, p)$, es decir: $(q, n) \times (m, p) = (q, p)$

Una vez conocidas las dimensiones de la matriz resultado, para obtener el elemento $i$ en la columna $j$ del producto $P = BA$, se selecciona el *iésimo* renglón de $B$ y la *jésima* columna de $A$, y se suman los productos de sus elementos correspondientes, iniciando en el extremo izquierdo y la parte superior, respectivamente. Esto pudiéndose denotar como $P_{ij} = \sum_{r=1}^{n} B_{ir} A_{rj}$.

Para el producto de matrices BA es necesario que B sea de dimensiones $(q, n)$ y que A sea de dimensiones $(m, p)$, sin embargo, podemos utilizar lo que se conoce como *multiplicación punto a punto* y multiplicar cada elemento $(i, j)$ de la matriz B por el correspondiente elemento $(i, j)$ de la matriz A. Este tipo de multiplicación es utilizada en el Procesamiento Digital de Imágenes.

La multiplicación de matrices es un problema muy fácilmente paralelizable y se encuentra en la categoría de problemas conocidos como *embarrassingly parallel*[62]. Esto es porque cada nodo que participe en el cálculo únicamente necesita conocer, antes de iniciar, los valores de las dos matrices a multiplicar y en ningún momento requerirá comunicarse con los demás nodos para realizar su trabajo.

### *3.4.5. Implementación paralela*

La idea principal de este trabajo consiste en aplicar un modelo de paralelismo a un algoritmo de un procesamiento digital de imágenes, en este caso el suavizado de imágenes, basado en la propuesta de simplificar al máximo posible el diseño del algoritmo paralelo. El modelo diseñado aplica el paralelismo en cálculos matemáticos independientes e introduce un operador central que controla todas las operaciones.

Dado una vez el algoritmo del suavizado de una imagen, se implementó el programa en MPI, utilizando cálculos matemáticos como la transformada de Fourier, multiplicación de matrices y la transposición de matrices. Esta implementación está con base en una inicialización, una utilización y una finalización de la librería del paso de mensajes.

---

[61] Ver el Apéndice, sección C Transformada de Fourier Discreta en una dimensión.
[62] Problemas vergonzosamente paralelizables.

### *3.4.5.1. Elección de organización*

Uno de los aspectos más útiles del cómputo paralelo es el de la elección de la organización lógica de los procesos que participarán en el cálculo. La elección de esta organización junto con el algoritmo a emplear para resolver el proceso es esencial para obtener un rendimiento óptimo del equipo paralelo.

Existen varias maneras tradicionales de organizar la división de trabajo en un cálculo en equipos paralelos. Una de ellas es el uso de un esquema *Maestro–Esclavo*. En este esquema uno de los procesos se dedica a arbitrar el trabajo de los demás, sin participar realmente en el cálculo. Este proceso se conoce como *maestro* mientras que los demás se denominan *esclavos*. En general el maestro divide el problema en unidades de trabajo, asigna estas unidades a los procesos esclavo, recoge los resultados entregados por los mismos, y consolida dichos resultados parciales para obtener una respuesta final.

Otro esquema muy utilizado es el de *Divide y Vencerás*. Este esquema consiste en descomponer el problema original en varios subproblemas más sencillos, para luego resolver éstos independiente mediante un cálculo específico. Por último, se combinan los resultados de cada subproblema para obtener la solución del problema original.

En general es complicado dar una idea de cuál es la manera más eficiente de organizar y dividir el trabajo. Ésta está determinada en gran medida por el algoritmo con que se va a atacar el problema. Tanto la elección del algoritmo como de la organización y división de trabajo son tareas que requieren experiencia e intuición, proporcionando el aspecto más *artístico* del cómputo en paralelo.

Tanto para el caso de la transformada de Fourier y como el de la multiplicación de matrices se optó por una organización *Maestro–Esclavo*.

### *3.4.5.2. El algoritmo en paralelo*

Se asume que en el cálculo participarán al menos dos nodos. La unidad de trabajo básica será el renglón, de forma que los nodos calculan los elementos resultado hasta completar un renglón, que se envía al nodo maestro para su consolidación procediendo el nodo esclavo a calcular otro renglón.

El primer nodo se constituye como maestro, obteniendo las matrices correspondientes a la imagen y al filtro aplicado y distribuyéndolas a los procesos esclavos por medio de un mensaje de *broadcast.*

Los esclavos reciben la matriz que conforma la imagen para realizar la transformada discreta de Fourier y determinan el número de procesos que participan en el cálculo, así como su lugar en la lista de procesos. Con esta información cada proceso puede decidir cuáles renglones debe resolver. En particular, los nodos se distribuyen equitativamente los renglones, dejando al último nodo los renglones restantes o residuo.

Como un ejemplo, si se tiene una matriz de *100* x *100*, y en el cálculo participan 7 nodos, se realiza la operación $\frac{100}{7}$, descartando la parte fraccionaria, se obtiene 14 renglones por cada nodo. Los dos renglones residuales se asignan automáticamente al último nodo. De esta forma la asignación de trabajo queda como sigue:

| Nodo | Renglones |
|---|---|
| 1 | 0 – 13 |
| 2 | 14 – 27 |
| 3 | 28 – 41 |
| 4 | 42 – 55 |
| 5 | 56 – 69 |
| 6 | 70 – 83 |
| 7 | 84 – 99 |

*Tabla 3.1 Asignación de trabajo para los nodos*

Una vez recibida la matriz a operar y conociendo el bloque de renglones asignados, cada nodo esclavo procede a calcular, entregando los resultados parciales al nodo maestro, estos resultados se envían a través de un mensaje punto a punto. El nodo maestro recibe estos resultados, contabilizando el número de renglones resueltos, y cuando se tiene la matriz semitransformada, el proceso se da por terminado. Como la transformada discreta de Fourier sólo se hizo en una sola dimensión, renglones, el maestro obtiene la matriz transpuesta de la matriz semitransformada y vuelve a enviar esta matriz a los nodos para que realicen nuevamente la transformada discreta de Fourier y así se obtenga la transformada discreta de Fourier de la matriz en dos dimensiones. Cuando el maestro recibe el resultado de la última matriz que envió, éste vuelve a obtener la matriz transpuesta para obtener la matriz de la imagen transformada.

El procedimiento anterior se realiza exactamente igual para la matriz que conforma al filtro que se le aplicará a la imagen.

Una vez realizadas, ambas matrices están transformadas; el maestro distribuye dichas matrices entre los procesos esclavos. Los esclavos reciben las matrices a multiplicar y determinan el número de procesos que participan en el cálculo, así como su lugar en la lista de procesos. Los nodos se distribuyen equitativamente los renglones, dejando al último nodo los renglones restantes o residuo. Una vez habiendo recibido las matrices a operar, cada nodo esclavo procede a calcular y posteriormente a entregar los resultados al nodo maestro.

El maestro, una vez que obtuvo la matriz resultante de la multiplicación, distribuye a los procesos esclavos esta matriz por medio de un mensaje de *broadcast* para que realicen la transformada discreta de Fourier inversa.

Los esclavos reciben la matriz para realizar la transformada discreta de Fourier inversa. Una vez recibida la matriz a operar, cada nodo esclavo procede a calcular y a entregar los resultados parciales al nodo maestro. El nodo maestro recibe estos resultados, contabilizando el número de renglones resueltos, y cuando se tiene la matriz semitransformada, el proceso se da por terminado. Como la transformada discreta de Fourier inversa sólo se hizo en una sola dimensión, renglones, el maestro obtiene la matriz transpuesta de la matriz semitransformada y vuelve a

enviar esta matriz a los nodos para que realicen nuevamente la transformada discreta de Fourier inversa y así se obtenga la transformada discreta de Fourier inversa de la matriz en dos dimensiones. Cuando el maestro recibe el resultado de la última matriz que envió, éste vuelve a obtener la matriz transpuesta para obtener la matriz transformada. Una vez realizada esta tarea, se tiene la matriz completamente resuelta, el proceso se da por terminado, y se puede decir que la imagen ha sido suavizada.

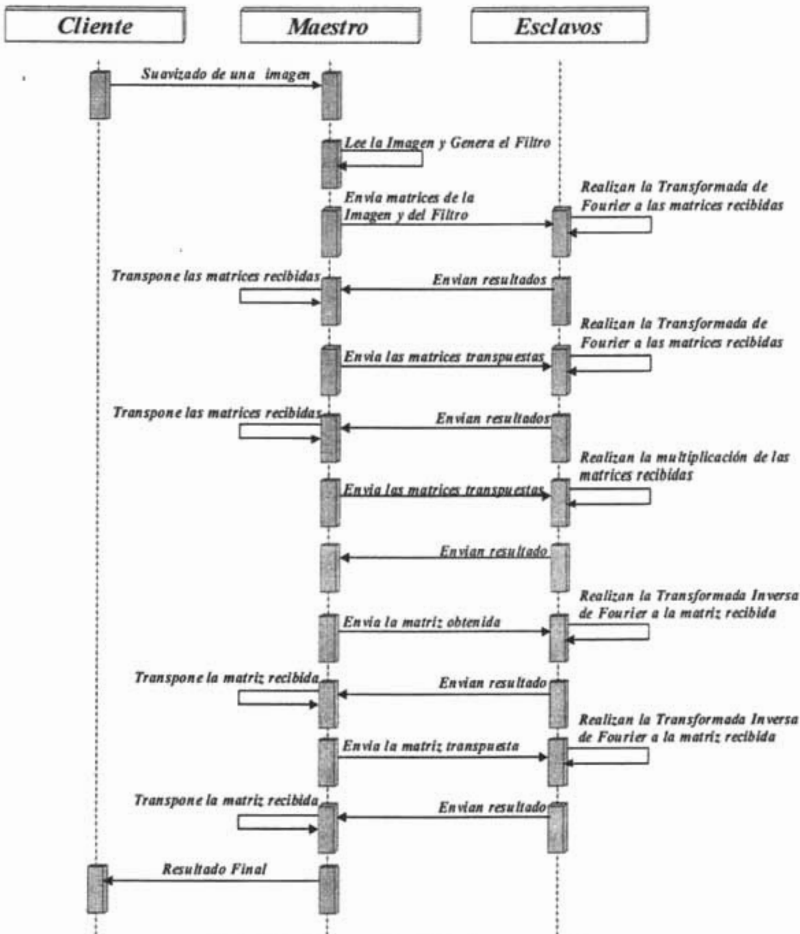


*Fig. 3.13 Implementación del algoritmo en paralelo*

### *3.4.5.3. Implementación*

El primer paso para utilizar MPI en un programa es inicializar el mecanismo de paso de mensajes. Éste se hace por medio de la llamada *MPI_Init*. A continuación el proceso determina el

tamaño de su comunicador[63], cuántos procesos lo componen, con una llamada *MPI_Comm_size*. Para la organización y distribución de trabajo, se utiliza el comunicador *MPI_COMM_WORLD*, que es el comunicador al que pertenecen inicialmente todos los procesos. Obviamente cada proceso puede posteriormente cambiar a otro comunicador, pero en este caso es suficiente el uso de *MPI_COMM_WORLD*.

Se determina también el rango del proceso dentro del comunicador[64] y el nombre del procesador en que se está ejecutando, llamando a *MPI_Comm_rank* y *MPI_Get_processor_name*. El rango es de particular utilidad para determinar cuáles renglones debe resolver cada proceso, según se describió en el subtema 3.4.5.2 *El algoritmo en paralelo*.

A continuación se realiza la bifurcación de trabajo en el proceso, es decir, se designa un proceso para que asuma el papel de maestro, mientras los demás se configuran como esclavos. Basándose en el parámetro *localid*, el rango dentro del comunicador se encuentra el trabajo que realiza el proceso maestro.

En este caso, el criterio de decisión es designar como maestro al proceso que tiene *localid* de 0. En MPI esta decisión es un tanto arbitraria, en realidad se puede escoger cualquier proceso como maestro, sin embargo el utilizar al proceso 0 es por convención y comodidad. Esto obedece al hecho de que en MPI no se tiene el concepto inherente de *proceso padre* como sí lo tiene PVM.

El proceso padre obtiene la imagen a partir de una archivo .raw y el filtro, transmite la imagen a todos los procesos del comunicador *MPI_COMM_WOLRD* para que se le aplique la transformada discreta de Fourier. Esto se hace por medio de una llamada a *MPI_Broadcast*. Aquí se indica la dirección de los datos a transmitir, la cantidad que se desea enviar, el rango del proceso raíz y el comunicador. El parámetro de proceso raíz indica cuál proceso va a iniciar el broadcast.

Nótese que en general, las llamadas a primitivas de comunicación en MPI requieren especificar la dirección de los datos a enviar o recibir, así como el tamaño o cantidad de información a comunicar.

Una vez enviada la imagen, el proceso padre no realiza ninguna tarea de cálculo, únicamente espera a recibir los renglones completos por parte de los nodos. Éste se hace en un ciclo que contabiliza el número de renglones recibidos y en cada iteración realiza una llamada a *MPI_Recv*. Esta llamada bloquea en espera de recepción de un mensaje de cualquier proceso del comunicador, parámetros *MPI_ANY_SOURCE* y *MPI_COMM_WORLD*.

Al terminar de recibir los renglones el proceso padre realiza la transpuesta de la matriz que conforma la imagen. Una vez realizada la matriz transpuesta, el proceso padre vuelve a transmitir la matriz a los nodos como se mencionó y queda en espera de recibir los renglones completos por parte de los nodos. Cuando el proceso padre termina de recibir los renglones, éste realiza la

---

[63] El comunicador, o contexto de comunicaciones, es el grupo de trabajo básica en MPI. En general, un comunicador delimita el alcance de llamadas a funciones grupales, como *broadcast* y *scatter/gatter*.
[64] Qué posición ocupa de entre los procesos que componen el comunicador.

transpuesta de la matriz para obtener la imagen transformada. Este proceso se hace de la misma manera para poder obtener el filtro transformado.

Cuando el proceso padre obtiene la imagen transformada y el filtro transformado, transmite ambos datos a todos los procesos del comunicador *MPI_COMM_WOLRD* para que se multipliquen. Esto se hace por medio de dos llamadas consecutivas a *MPI_Broadcast*. Una vez enviados los datos, el proceso padre no realiza ninguna tarea de cálculo, únicamente espera a recibir los renglones completos por parte de los nodos.

El proceso padre, una vez obtenida la matriz resultado de la multiplicación, transmite dicha matriz a todos los procesos del comunicador *MPI_COMM_WOLRD* para que se le aplique la transformada discreta de Fourier inversa, por medio de *MPI_Broadcast*. Una vez enviada la matriz, el proceso padre no realiza ninguna tarea de cálculo, únicamente espera a recibir los renglones completos por parte de los nodos. Al terminar de recibir los renglones el proceso padre realiza la transpuesta de la matriz que conforma la matriz. Una vez realizada la matriz transpuesta, el proceso padre vuelve a transmitir la matriz a los nodos como se mencionó y queda en espera de recibir los renglones completos por parte de de los nodos. Cuando el proceso padre termina de recibir los renglones, éste realiza la transpuesta de la matriz para obtener la matriz transformada. Al terminar de recibir los renglones el proceso padre envía el resultado a un archivo .raw y termina su ejecución.

Los procesos esclavos, con *localid* diferente de 0, asignan memoria para las matrices a operar y reciben sus valores por medio de broadcast. Obsérvese que en MPI la llamada a *MPI_Broadcast* es igual para recibir información. Los procesos cuyo *localid* sea diferente al especificado en la llamada, recibirán la información del proceso que inició el *broadcast*.

Una vez teniendo la información necesaria, los nodos comienzan a resolver su porción de matriz, guardando el resultado parcial de cada renglón en un arreglo y enviándolo al proceso maestro cuando se ha completado un renglón. Esto se hace por medio de una llamada a *MPI_Send*. Esta función toma como parámetros la dirección y tamaño de la información a enviar, el tipo de datos, el proceso destino, en este caso 0, una bandera identificadora de mensaje y el comunicador al que se debe enviar el mensaje.

Los nodos continúan con este proceso hasta terminar el cálculo de sus renglones asignados, en este momento terminan su ejecución.

### *3.4.6. Compilación*

Uno de los binarios incluidos en la distribución de MPICH es un *front end* para compilar, que se encarga de agregar las rutas necesarias para los archivos incluidos y las bibliotecas, así como el ligado final del ejecutable. Éste reside en */usr/share/mpi/bin* y se invoca de la siguiente manera:

*mpicc suavizado.c -lm*

Esto genera el ejecutable de *suavizado.c* y lo deja listo para su ejecución.

### *3.4.7. Ejecución*

En sentido estricto, MPI no cuenta con una manera de ejecutar aplicaciones que utilicen sus funciones. Este detalle se deja a cada implementación particular. MPICH provee el comando *mpirun*, que se encarga de inicializar los mecanismos de comunicación necesarios, así como los programas a ejecutar y realizar la corrida de los mismos.

*mpirun* toma un parámetro *–np* para indicar en cuántos procesadores se va a ejecutar el programa. En un Cluster se tiene una lista de máquinas, cada una de las cuales cuenta como un procesador, de modo que si se especifica *–np 5* se indica que se ejecutará el programa de MPI utilizando 5 nodos. Los procesos se inician en el orden que se especifica en el archivo de lista de nodos.

Entonces, para ejecutar el programa de suavizado de imágenes en MPI, se debe indicar el siguiente comando:

*mpirun –np 7 suavizado lena.raw 256 256 3*

Este comando especifica ejecutar el programa utilizando 7 nodos. Como parámetros adicionales se coloca el nombre de la imagen, la dimensión de la imagen, *256 x 256*, y la del filtro promediador, que en este caso es de una dimensión de *3 x 3*. El comando *mpirun* realiza las tareas de inicialización y arranque del número de procesos necesarios.

La ejecución del programa en MPI fue relativamente limpia. No se proporciona un mecanismo de control para los procesos que se ejecutan en paralelo, lo cual supone desventaja, aunque por otro lado, en caso de complicaciones, el detener el proceso inicial automáticamente elimina los procesos generados en los nodos.

En caso de ocurrir algún error, MPI envía mensajes que en ocasiones no resultan tan obvios pero permiten suponer que ha ocurrido alguna complicación, en cuyo caso se puede detener el proceso restaurando el estado del sistema.

$\varphi$

# Capítulo IV
## Resultados

# *Resultados*

Una vez contando con el programa de suavizado de imágenes, se realizaron una serie de pruebas a fin de poder tener una idea del desempeño y características de la aplicación en paralelo.

En general existen tres parámetros que afectan directamente el rendimiento que se puede esperar del equipo:

- La dimensión de la imagen a utilizar.
- La dimensión del filtro paso bajas a utilizar.
- El número de elementos de procesamiento, nodos, que participan en el cálculo.

A fin de poder observar el comportamiento del Cluster al variar estos tres parámetros se decidió realizar pruebas con varios tamaños de filtros y varias imágenes, variando el número de nodos, y tomando el tiempo empleado en la realización de los cálculos

Para cada tamaño del filtro se tomaron los tiempos de solución con la versión uniprocesador, valores que en la siguiente tabla aparecen como "1 nodo", y con la versión paralela utilizando MPI, desde 2 hasta 17 nodos. De esta manera se puede comparar directamente la diferencia de desempeño entre el uso de un solo procesador, sin recurrir a las bibliotecas paralelas y el uso de varios nodos para realizar el cálculo.

Para cada combinación de tamaño del filtro y número de nodos se realizaron tres corridas del programa, descartando la primera y promediando las dos restantes para obtener el valor final. Esto se hizo para evitar posibles valores *picos* o valores extraños, que podrían presentarse tomando sólo una lectura; el descartar la primera lectura permite eliminar posibles variaciones en el tiempo de ejecución por los mecanismos de *caching* de disco con que cuentan los nodos, y puede influir al momento de ejecutar un proceso diferente.

Dada la cantidad de corridas que debieron efectuarse para la realización de las pruebas, éstas abarcaron un periodo de aproximadamente cuatro días, y los resultados se muestran continuación.

- ***Prueba 1***
  En la primera prueba realizamos el suavizado de una imagen de dimensiones *256 x 256*, *lena.raw*, desde 1 hasta 17 nodos, con tamaños del filtro de *3 x 3* y *5 x 5*. Para el nodo 1, el resultado es realmente por el programa de suavizado en uniprocesador. De 2 nodos en adelante, se empleó el programa de suavizado con MPI. Todos los tiempos están en segundos.

***Fig. 4.1 Imagen lena.raw***

| Nodos | Tamaño Filtro 3 x 3 | Tamaño Filtro 5 x 5 |
|---|---|---|
| 1 | 811.567279 | 811.642757 |
| 2 | 1357.099026 | 1357.087611 |
| 3 | 1099.130580 | 1099.335742 |
| 4 | 825.416795 | 831.012732 |
| 5 | 676.495152 | 677.212210 |
| 6 | 597.557932 | 600.977209 |
| 7 | 700.739183 | 701.777209 |
| 8 | 512.559596 | 571.925669 |
| 9 | 507.533688 | 507.233346 |
| 10 | 561.307664 | 564.164529 |
| 11 | 481.706843 | 481.916250 |
| 12 | 468.998182 | 468.132665 |
| 13 | 525.272334 | 526.044144 |
| 14 | 601.596406 | 600.208903 |
| 15 | 509.482303 | 482.072751 |
| 16 | 472.864789 | 461.280239 |
| 17 | 438.195624 | 441.301652 |

***Tabla 4.1 Resultados obtenidos variando el número de nodos y el tamaño del filtro para lena.raw***

*Fig. 4.2 lena.raw suavizada con un filtro de 3x3*

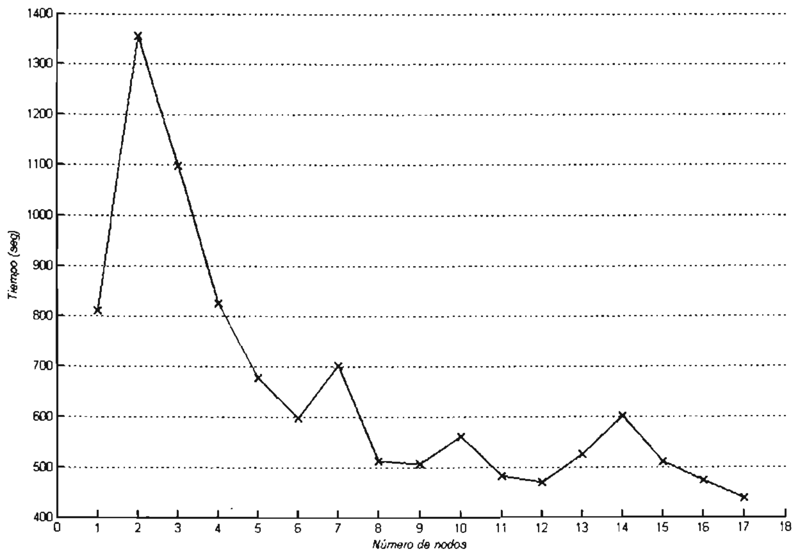


*Fig. 4.3 Gráfica de rendimiento del suavizado de lena.raw con filtro de 3x3*

*Fig. 4.4 lena.raw suavizada con un filtro de 5x5*

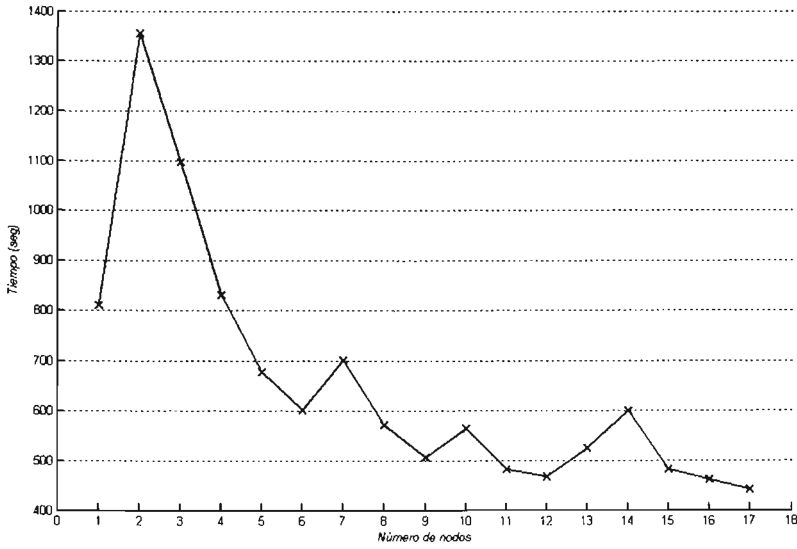


*Fig. 4.5 Gráfica de rendimiento del suavizado de lena.raw con filtro de 5x5*

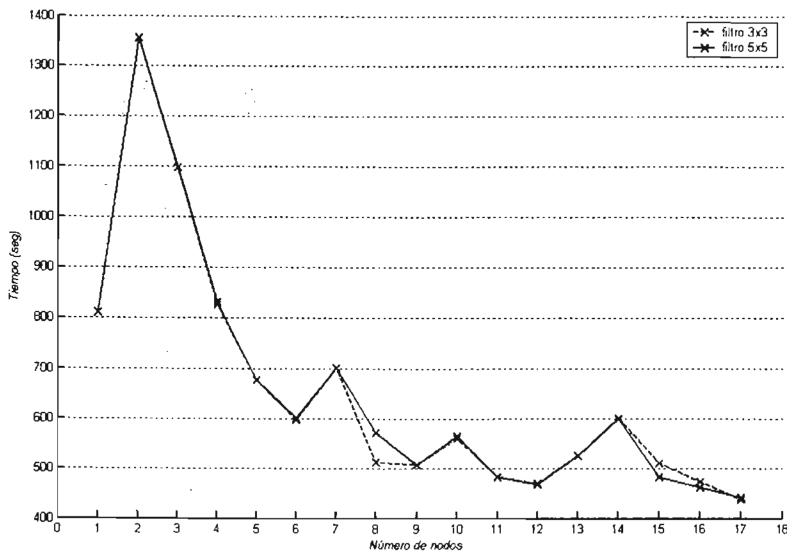


***Fig. 4.6 Gráfica de comparación al variar el número de nodos y el tamaño del filtro para lena.raw***

Las gráficas de rendimiento para el suavizado de la imagen *lena.raw*, de los filtros de *3 x 3* y *5 x 5* presentan un comportamiento similar. Se puede observar que los mejores tiempos de cálculo se logran utilizando el nodo 17 para la versión en MPI. El comportamiento general es de mejoramiento del rendimiento al agregar más nodos, aunque en algunos puntos hay diversos picos derivados del tráfico de la red del Cluster.

Las reducciones del desempeño entre 6-7, 9-10 y 12-14 se deben al hecho de que los equipos utilizados en el Cluster no tienen un rendimiento similar. Al introducir los equipos 486/50, que tienen un rendimiento menor a los 486/66, se genera un *cuello de botella* donde el equipo más lento es el que determina la terminación del cálculo que se está realizando.

- ***Prueba 2***

  En la segunda prueba realizamos el suavizado de una imagen de dimensiones *119 x 119*, *televisa.raw*, desde 1 hasta 17 nodos, con tamaños del filtro de *3 x 3* y *5 x 5*. Para el nodo 1, el resultado es realmente por el programa de suavizado en uniprocesador. De 2 nodos en adelante, se empleó el programa de suavizado con MPI. Todos los tiempos están en segundos.

*Fig. 4.7 Imagen televisa.raw*

| Nodos | Tamaño Filtro 3 x 3 | 5 x 5 |
|---|---|---|
| 1 | 83.577927 | 82.33564 |
| 2 | 139.573889 | 139.446730 |
| 3 | 115.787427 | 116.348664 |
| 4 | 92.205551 | 93.135616 |
| 5 | 80.694239 | 80.223345 |
| 6 | 72.796485 | 72.549323 |
| 7 | 70.393710 | 70.890917 |
| 8 | 90.004155 | 87.113375 |
| 9 | 58.371055 | 58.680461 |
| 10 | 85.162992 | 84.485401 |
| 11 | 82.162992 | 82.591177 |
| 12 | 98.087528 | 86.663108 |
| 13 | 55.826102 | 56.076777 |
| 14 | 97.384813 | 96.969225 |
| 15 | 54.596068 | 55.068641 |
| 16 | 56.158593 | 58.684894 |
| 17 | 58.245399 | 61.228725 |

*Tabla 4.2 Resultados obtenidos variando el número de nodos y el tamaño del filtro para televisa.raw*

*Fig. 4.8 televisa.raw suavizada con un filtro de 3x3*

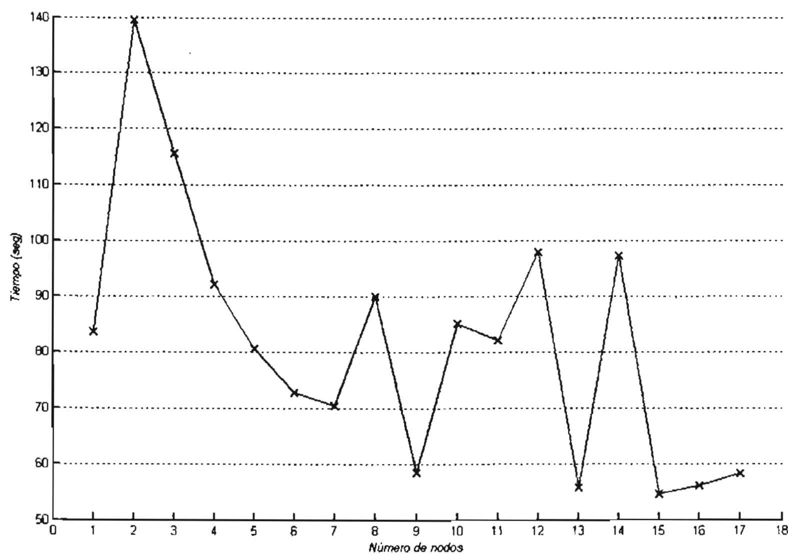


*Fig. 4.9 Gráfica de rendimiento del suavizado de televisa.raw con filtro de 3x3*

*Fig. 4.10 televisa.raw suavizada con un filtro de 5x5*

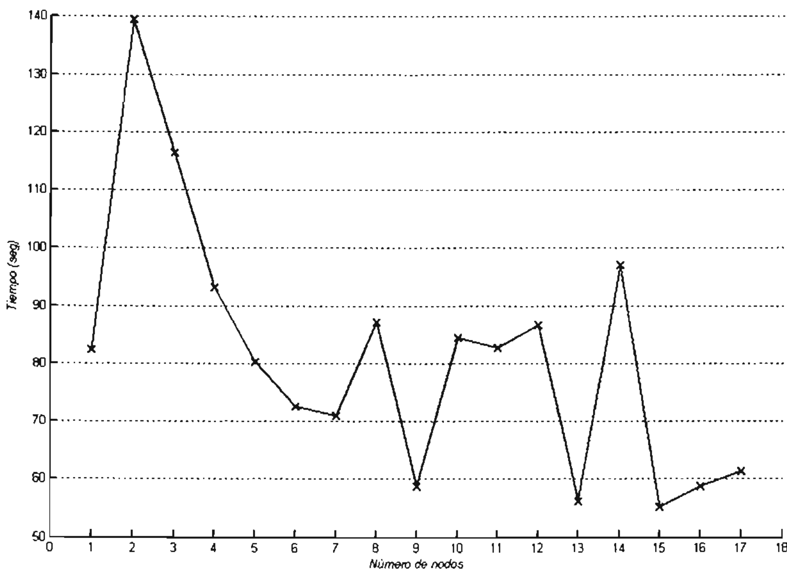


*Fig. 4.11 Gráfica de rendimiento del suavizado de televisa.raw con filtro de 5x5*

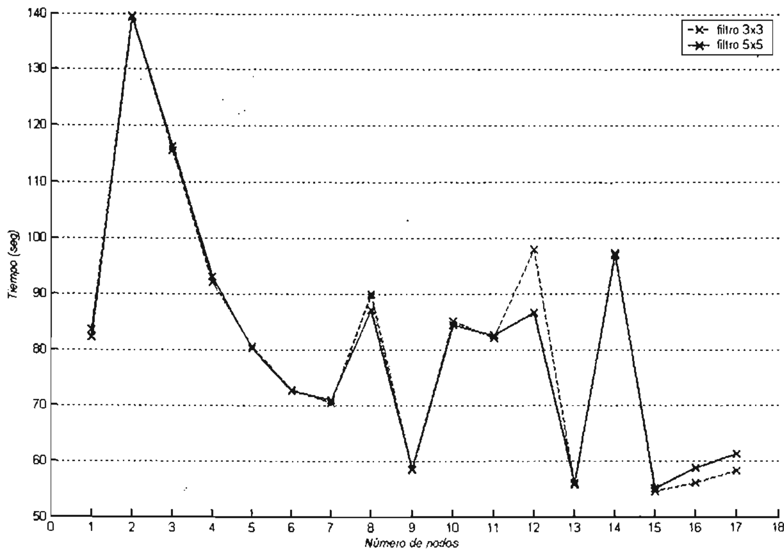


***Fig. 4.12 Gráfica de comparación al variar el número de nodos y el tamaño del filtro para televisa.raw***

Las gráficas de rendimiento para el suavizado de la imagen *televisa.raw*, de los filtros de *3 x 3* y *5 x 5* presentan un comportamiento similar. Se puede observar que los mejores tiempos de cálculo se logran utilizando el nodo 15 para la versión en MPI.

Las fluctuaciones más o menos violentas del desempeño para esta imagen se deben a la poca duración del cálculo, de forma que se introducen las variaciones debido a características del arranque del proceso en el sistema y el comportamiento aleatorio de las comunicaciones de red por el arranque de los procesos remotos.

El filtro promediador deja pasar las componentes de frecuencias bajas de la imagen y estas componentes son las responsables del contraste y la intensidad de la imagen. Por lo tanto, al aplicarle el filtro a las imágenes de *lena.raw* y *televisa.raw* variamos el contraste y los valores de los píxeles promedios de la imagen.

Al aumentar el tamaño del filtro promediador produce una mayor variación en el contraste y los valores de los píxeles promedios de la imagen, es decir, la imagen obtenida tiene una menor resolución (borrosa). La condición esencial de este tipo de filtros es la afinación de algunas zonas de la imagen, especialmente en donde hay ruido.

Comparando las gráfica para las imágenes de *lena.raw* y *televisa.raw* notamos que mientras mayor sea la carga de trabajo, es decir, mayor es la dimensión de la imagen, el tiempo que tarda la ejecución del programa se incrementa.

En general, para imágenes de tamaño más grandes se obtiene un mayor beneficio al utilizar el número máximo de nodos, pero a medida que la imagen se reduce de tamaño el mejor rendimiento se alcanza con menos nodos.

φ

# Conclusiones

# Conclusiones

El proyecto de investigación que se presenta en este documento nos permitió acercarnos al mundo del cómputo paralelo. Este acercamiento se dio tanto en la parte referente al hardware, como en el aspecto del software, al entender y manejar los conceptos de programación propios de este ambiente.

De esta manera podemos concluir que al querernos introducir al mundo de la computación paralela y resolver problemas que se requirieran de este tipo de cómputo, necesitamos de una máquina paralela y un lenguaje de programación como C, C++ o Fortran que interactúe con las diversas herramientas de paralelización.

Un Cluster es un sistema paralelo y distribuido, al cual se interconecta un grupo de computadoras, pudiendo funcionar como un solo recurso computacional unificado. La idea principal en el uso de los Clusters es el de dividir grandes problemas computacionales en tareas más pequeñas, que puedan correr e intercomunicarse, cada una en diferentes computadoras.

Como el Cluster es un sistema distribuido, no hay puntos únicos de falla, es decir, si uno de los computadoras presenta anomalías, el resto del Cluster puede seguir trabajando, a diferencia de un sistema con máquinas SMP, en donde los procesadores comparten la memoria, el bus de entrada/salida y el disco, por lo que un problema con cualquiera de estos componentes afecta a todos los procesadores simultáneamente.

Podemos decir que la descomposición del problema del procesamiento digital de imágenes, suavizado de imágenes, se llevó acabo de manera satisfactoria siguiendo los pasos planteados por la metodología descrita en páginas anteriores.

Para paralelizar la aplicación es necesario detectar las partes en que la aplicación secuencial puede dividirse sin afectar el resultado y cada parte de esta división pueda hacerse de manera independiente para al final juntarlos y obtener el resultado deseado. Se partió de un problema general enfocado a resolver un procesamiento digital de imágenes que se encarga de eliminar ruido gaussiano de la imagen. El siguiente paso fue fragmentar el problema en el mayor número de tareas independientes para así poder realizar una mejor paralelización de tareas, aquí es importante mencionar que para realizar el suavizado de la imagen se pueden seguir dos caminos, el primero es hacer el suavizado directamente sobre la imagen, datos, en lo que se conoce como dominio del tiempo y el segundo camino consiste en pasar los datos al dominio de la frecuencia utilizando la transformada de Fourier. Se optó por el segundo camino debido que nos permite una paralelización más sencilla debido a la propiedad de separabilidad de la transformada de Fouier.

Después de obtener las tareas independientes a realizar, se agruparon en procesos que permitieran el correcto funcionamiento del algoritmo de suavizado mediante la comunicación entre los procesos esclavos y el proceso maestro, debido a que los esclavos son los encargados de realizar los cálculos y enviarlos al maestro, el cual sólo los agrupa y decide qué datos volver a enviar a los esclavos para que sigan trabajando.

Al concluir el análisis según la metodología llegamos a una arquitectura que se implementó en el lenguaje de programación C auxiliándonos de la librería de paso de mensajes MPI. El paso de mensajes es útil en el procesamiento paralelo, principalmente cuando una aplicación no requiere que haya una comunicación continua entre los procesadores. Si dicha aplicación requiere que sus procesos se estén comunicando entre sí constantemente, depende de la velocidad de la red si esta comunicación se realiza eficientemente.

Puesto que los resultados obtenidos de la ejecución del programa son favorables, podemos concluir que la metodología es funcional y perfectamente aplicable a un problema que sea factible de paralelizar por que simplifica el proceso de análisis permitiendo una rápida decisión de si se puede descomponer en varias tareas o no. Además de que permite una visualización a nivel global de la estructura que se tendrá el programa de aplicación incluso antes de su implementación.

Con el análisis del desempeño del Cluster llegamos a los siguientes puntos claves:

- El desempeño del Cluster se puede ver afectado por la máquina más lenta que forme parte de éste.
- El desempeño también está delimitado por la velocidad de las máquinas que sea el común denominador, esto es, si hay máquinas más rápidas que el resto del grupo, estas máquinas tendrán que esperarse a que las demás terminen.
- Dependiendo de las velocidades de las máquinas que conforman el Cluster, si tenemos una más veloz que las demás, ésta podría desempeñar el mismo trabajo que 2, 3 o más máquinas juntas de menor velocidad.
- No tiene caso utilizar una máquina paralela o paralelizar un problema, si éste no va a realizar un cálculo considerable, ya que de ser así el Cluster tardaría más tiempo en realizar la comunicación entre las máquinas que el cálculo especificado. Por lo tanto, es importante evaluar la dimensión del problema a resolver antes de decidir si se obtendrá un beneficio utilizando un Cluster.
- Las fluctuaciones más o menos violentas del desempeño del Cluster para los problemas pequeños se deben a la poca duración del cálculo, de forma que se introducen las variaciones debido a características del arranque del proceso en el sistema y el comportamiento aleatorio de las comunicaciones de red por el arranque de los procesos remotos.
- En general para los problemas grandes se obtiene un mayor beneficio al utilizar el número máximo de nodos, pero a medida que el problema es más pequeño el mejor rendimiento se alcanza con menos nodos. En el extremo de esta lógica se observa que para problemas muy pequeños el uso de un solo procesador da un menor rendimiento.

Al aplicar un filtro promediador dejamos pasar las componentes de frecuencias bajas de la imagen y estas componentes son las responsables del contraste y la intensidad de la imagen. Por lo tanto, al aplicarle el filtro promediador a una imagen variamos el contraste y los valores de los píxeles promedios de la imagen.

Al aumentar el tamaño del filtro promediador produce una mayor variación en el contraste y los valores de los píxeles promedios de la imagen, es decir, la imagen obtenida tiene una menor resolución (borrosa).

La condición esencial de este tipo de filtro es la afinación de algunas zonas de la imagen, especialmente en donde hay ruido y dar pie a que podamos apreciar mejor la imagen. Ésta es la principal ventaja del filtro promediador.

Finalmente, se considera que los objetivos planteados al inicio de este trabajo se alcanzaron satisfactoriamente, y se espera que el resultado del siguiente trabajo pueda proveer una metodología orientada al procesamiento paralelo, tomando en cuenta todos los factores que influyen para la creación, utilización y ejecución de una aplicación de ingeniería en Clusters de alto desempeño.

φ

# Bibliografía

# Bibliografía

## LIBROS

***High Performance Cluster Computing:*** *Architectures and System,* Vol. 1, 1/e
Rajkumar Buyya
School of Computer Science and Software Engineering
Monash University, Melbourne, Australia.
Copyright 1999.

*Tesis* ***Análisis del desempeño de un Cluster Beowulf en diversos algoritmos de tipo concurrente***
Juárez Sosa Julio Cesar, Sáenz García Elba Karén y Valdez Casillas Oscar René
Facultad de Ingeniería, UNAM.
México, 2001.

*Tesis* ***Construcción y evaluación de desempeño de un Cluster Beowulf para cómputo de alto rendimiento***
Manrique Martínez Daniel.
Facultad de Ingeniería, UNAM.
México, 2001.

***Parallel Computer Architecture:*** *A Hardware & Software Approach*
Culler David, Singh J. P. y Gupta A.
Morgan Kauffman Publishers, 1998.
ISBN 1-55860-343-3.

***Parallel Computing:*** *Theory and Practice*
Quinn, Michael J.
McGraw-Hill, 1994.
ISBN 0-07-051294-9.

***Diccionario de Informática.*** *Traducción del Dictionary of Computing, 3rd Edition*
De Mendizábal Allende Blanca, Oxford University.
Ediciones Díaz de Santos, 1993, España.
ISBN 84-7978-068-1.

***Operating System,*** *2nd Edition*
Harvey M. Deitel.
Addison-Wesley, 1990, USA.
ISBN 0-201-50939-3.

***IEEE Spectrum***
The Institute of Electrical and Electronic Engineers Inc.
USA, Enero 1991.

*Timeline of Computing History. The Computer Society*
The Institute of Electrical and Electronic Engineers Inc.
USA, 1996.

*PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*
Al Geist, Adam Beguelin, Jack Dongarra, Weigheng Jiang, Robert Manchek, Vaidy Sunderman.
MIT Press, 1994.

## ARTICULOS

*Paralelización en la Supercomputadora Cray Origin 2000*
Oscar Rafael García Regis. Laboratorio de Dinámica No Lineal, Facultad de Ciencias, UNAM.
Enrique Cruz Martínez. Cómputo Aplicado, DGSCA, UNAM.

*Procesamiento Paralelo*
Boletín 5.
Facultad de Informática y Multimedia, Universidad Internacional del Ecuador (UIDE-BITS).
Ecuador.

*Procesamiento Paralelo 2003*
Clase 4.
Facultad de Informática, Universidad Nacional de la Plata.
Argentina.

*Arquitectura e Ingeniería de Computadoras*
José M. Claver.
UJI.

*Introducción a las tecnologías clustering en GNU/Linux*
Rosa Maria Yánez Gómez.
Versión 1.0.

*Los Clusters como plataforma de procesamiento paralelo*
Oscar Pino Morillas, Roberto Francisco Arroyo Moreno y Francisco Javier Nievas Muñoz.

*Introducción a MPI*
Francisco Hidrovo, Herbert Hoeger.
Centro Nacional de Calculo Científico (CeCalCULA), Universidad de los Andes.
Venezuela.

*Introducción al procesamiento digital de imágenes*
Departamento Postgrado.
Facultad de Ciencias Naturales y Museo, Universidad Nacional de la Plata.
Argentina.

*Filtrando Imágenes*
Randall B. Smith, Ph.D.
GeoVectra S.A.

*Filtrado de imágenes en el dominio de la frecuencia*
C. Pinilla, A. Alcalá y F. J. Ariza.
Departamento de Ingeniería Cartográfica, Geodésica y Fotogrametría. Universidad de Jaén.

*Procesando en dos dimensiones*
Miguel Ángel Lagunas.

## DIRECCIONES WEB

http://www.cpmputer.org/computer/timeline/

http://www.dei.uc.edu.py/tai2002/SD/discom.htm

http://www.r-associates.com/spanish.html

http://www.fing.edu.uy/inco/grupos/cecal/hpc/proyectos/

http://numerix.us.es/pers/denk/_parallel/modelo.html

http://homepage.mac.com/eravila/histpara.html

http://www.uns.edu.ar/secretarias/Tecnologia/Agencia/PICTPonzoni.htm

http://es.tldp.org/Manuales-LuCAS/doc-cluster-computadoras/doc-cluster-computadoras-html/node9.html

http://www.acm.org/crossroads/espanol/xrds8-3/fineGrained.html

http://telematica.cicese.mx/computo/super/cicese2000/paralelo/Part4.html

http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/MonogSO/PLAPRO02_archivos/granularidad.htm

http://parallel.vub.ac.be/documentation/pvm/Example/Marc_Ramaekers/node8.html

http://parallel.vub.ac.be/documentation/pvm/Example/Marc_Ramaekers/node9.html

http://parallel.vub.ac.be/documentation/pvm/Example/Marc_Ramaekers/node1.html

http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/MonogSO/PLAPRO02_archivos/elemento__de__diseno.htm

http://polaris.dit.upm.es/~jcduenas/patrones/MASTER.htm

http://parallel.vub.ac.be/documentation/pvm/Example/Marc_Ramaekers/node10.html

http://www.lania.mx/biblioteca/newsletters/2000-otono-invierno/linea-inv-vsanchez.htm

http://studies.ac.upc.es/EPSC/TFC/Beowulf/beowulf.html

http://www.ac.upc.es/lcac/i/plantilla1/index,es.html

http://www.r-associates.com/clusters-es.html

http://delta.cs.cinvestav.mx/~adiaz/ParProg2000/ParProg.html

http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpichman-chp4/node25.htm

http://flanagan.ugr.es/oep/node12.html

http://www.linti.unlp.edu.ar/trabajos/tesisDeGrado/tutorial/redes/modosi.htm

http://www.enete.us.es/docu_enete/varios/redes/osi.html

http://www.ceres.ugr.es/~alumnos/redrs232/osi.htm

http://www.fuac.edu.co/autonoma/servicios/estudiantes/tele/Osi/osi.html

http://supercomputo.izt.uam.mx/inicio/masinformacionclustes.htm

http://www.blando.info/luis/thesis_cuc/glosario.html

http://www.dpi.inpe.br/spring/usuario_spa/c_filtro.htm

http://www.dpi.inpe.br/spring/usuario_spa/raw_format.htm

http://www.us.es/gtocoma/pid/introduccion.html

http://www.us.es/gtocoma/pid/pid6/pid61.htm

http://paleo.ija.csic.es/tele/TUTORIAL%20A.I/enhancement/filtros.htm

http://gente.pue.udlap.mx/~mramirez/notaspdi/contenido.htm

http://www.vinuesa.com

http://paleo.ija.csic.es/tele/TUTORIAL%20A.I/enhancement/filtropb.htm

http://paleo.ija.csic.es/tele/TUTORIAL%20A.I/introduccion/imdig_1.htm

http://www.ing.ula.ve/~abravo/document/tutorial/imagenes/capitulo%2010.html

http://www.ing.ula.ve/~abravo/document/tutorial/imagenes/capitulo%2016.html

http://www.ing.ula.ve/~abravo/document/tutorial/imagenes/capitulo%2011.html

http://www.lfcia.org/~cipenedo/cursos/Ip/Tema4/nodo4_2.html

http://www.ii.uam.es/~taao1/practica/practica_Op2.html

http://www.ii.uam.es/~siguenza/procesamiento.ppt

http://www2.ing.puc.cl/~dmery/iman/

http://iie.fing.edu.uy/investigacion/grupos/gti/timag/trabajos/2004/segmentacion_sar/

http://www.etsimo.uniovi.es/vision/intro/node30.html

http://www.etsimo.uniovi.es/vision/intro/node23.html

http://www.wfu.edu/~matthews/misc/graphics/formats/formats.html

http://mecfunnet.faii.etsii.upm.es/difraccion/filtrado.html

φ

# Apéndice

# Apéndice

## A. *Programa* **HolaMundo.c**
*Prueba de funcionalidad del Cluster*

```
1   /**
2    *
3    * FileName: HolaMundo.c
4    * @authors: Eduardo Ávila Jiménez, Fernando Nahu Cantera Rubio, Eric R. Castañeda Caballero
5    * Date Created: Oct 29, 2004
6    *
7    **/
8
9   #include <stdio.h>
10  #include <mpi.h>
11
12  //Despliega en pantalla el mensaje "HOLA"
13  int main(int argc,char **argv)
14   {
15    MPI_Init(&argc,&argv);
16     printf("HOLA\n");
17    MPI_Finalize();
18  }
```

## B. *Programa* **LeeImagen.c**
*Lectura de una imagen con formato raw*
*Versión de prueba*

```
1   /**
2    *
3    * FileName: LeeImagen.c
4    * @authors: Eduardo Ávila Jiménez, Fernando Nahu Cantera Rubio, Eric R. Castañeda Caballero
5    * Date Created: Nov 8, 2004
6    *
7    **/
8
9   #include <stdio.h>
10  #include <malloc.h>
11  #include <stdlib.h>
12
13  //Asigna memoria float
14  float ** alloc_dmatrix(long ren,long col)
15  {
16   long i;
17   float **m;
18   m=(float **)malloc((ren*sizeof(float *)));
19   if(!m)
20    return NULL;
21   m[0]=(float *)malloc((ren*col)*sizeof(float));
22   if(!m[0])
23    return NULL;
24   for(i=1;i<ren;i++)
25    m[i]=m[i-1]+col;
26   return m;
27  }
```

```
28
29    //Libera memoria de float
30    void free_dmatrix(float **m)
31    {
32     free(m[0]);
33     free(m);
34    }
35
36    //Asigna memoria unsigned char
37    unsigned char * asg(unsigned char * s)
38    {
39     unsigned char *t;
40     if(s!=NULL)
41      {
42       t=(unsigned char *)malloc(strlen(s)+1);
43       strcpy(t,s);
44      }
45     return t;
46    }
47
48    //Lee la imagen de un archivo .raw
49    unsigned char ** lee(char *nombre,int M,int N)
50    {
51     unsigned char buffer[N];
52     unsigned char **img;
53     int i,j;
54     FILE *f=fopen(nombre,"rb");
55     i=0;
56     img=(unsigned char **)malloc((M*sizeof(unsigned char *)));
57     while(1)
58      {
59       fseek(f,(i*sizeof(unsigned char)*N),SEEK_SET);
60       if(fread(buffer,sizeof(unsigned char),N,f)==0)
61         break;
62       img[i]=asg(buffer);
63       i=i+1;
64      }
65     img[i]='\0';
66     fclose(f);
67     return img;
68    }
69
70    //Formato a tipo Double
71    float ** dobles(unsigned char **image,int M,int N)
72    {
73     int i,j;
74     float **ims;
75     ims=alloc_dmatrix((long)M,(long)N);
76     for(i=0;i<M;i++)
77      {
78       for(j=0;j<N;j++)
79         ims[i][j]=(float)((float)image[i][j]/(float)255);
80      }
81     return ims;
82    }
83
84    //Liberar memoria de unsigned char
85    void liberachar(unsigned char ** img)
86    {
87     int i;
88     i=0;
89     while(img[i]!=(unsigned char)NULL)
```

```
90      {
91       free(img[i]);
92       i=i+1;
93      }
94     free(img);
95    }
96
97    //Escribe la imagen a un archivo
98    void escribe(char * nombre,unsigned char **img,int M,int N)
99    {
100    FILE *f=fopen(nombre,"wb");
101    int i;
102    for(i=0;i<M;i++)
103     fwrite(img[i],sizeof(unsigned char),N,f);
104    fclose(f);
105   }
106
107   main()
108   {
109    unsigned char **im;
110    float **imx;
111    int i,j;
112    im=lee("lena256.raw",256,256); //Se lee la imagen
113    for(i=0;i<256;i++) //Se manda imprimir en pantalla la imagen
114     {
115      printf("\n");
116      for(j=0;j<256;j++)
117       printf("%d ",im[i][j]);
118      if(i==3)
119       break;
120     }
121
122    escribe("lena.raw",im,256,256); //Se manda escribir a un archivo
123    imx=dobles(im,256,256);
124    liberachar(im);
125    for(i=0;i<256;i++)//Se manda imprimir en pantalla el archivo
126     {
127      printf("\n");
128      for(j=0;j<256;j++)
129       printf("%.4f ",imx[i][j]);
130      if(i==3)
131       break;
132     }
133
134    free_dmatrix(imx);
135    printf("\n");
136   }
```

## C. *Programa* **Transformadafd.c**
### *Transformada de Fourier Discreta en una dimensión*
### *Versión de prueba*

```
1    /**
2     *
3     * FileName: Transformadafd.c
4     * @authors: Eduardo Ávila Jiménez, Fernando Nahu Cantera Rubio, Eric R. Castañeda Caballero
5     * Date Created: Dic 1, 2004
6     *
7     **/
8
9    #include <stdio.h>
10   #include <math.h>
11   #include <stdlib.h>
12
13   #define dospi 6.2831
14
15   //Obtener obtiene de memoria matrix de float
16   float m_g_c_f(float * matrix,long cols,long x,long y)
17   {
18    long valor_lineal;
19    valor_lineal=((y*cols)+x);
20    return matrix[valor_lineal];
21   }
22
23   //Obtener guarda en memoria matrix de float
24   int m_s_c_f(float *matrix,long cols,long x,long y,float val)
25   {
26    long valor_lineal;
27    valor_lineal=(y*cols+x);
28    matrix[valor_lineal]=val;
29    return 0;
30   }
31
32   //Se calcula la Transforma Discreta de Fouier
33   void fourier(long renini,long renfin, long col,float *a,float *b,float *x,float *y)
34   {
35    long i,j,k;
36    float tempr;
37    float tempi;
38    float w=0;
39    float p,p1;
40
41    for(i=renini;i<renfin;i++)
42     {
43      for(k=0;k<col;k++)
44       {
45        w=(float)((6.2832*k)/col);
46        tempr=0; tempi=0;
47        for(j=0;j<col;j++)
48         {
49           p=(float)((m_g_c_f(a,col,j,i)*cos((double)w*j)) + (m_g_c_f(b,col,j,i)*sin((double)w*j)));
50          p1=(float)((m_g_c_f(b,col,j,i)*cos((double)w*j)) - (m_g_c_f(a,col,j,i)*sin((double)w*j)));
51          tempr= tempr + p;
52          tempi=tempi + p1;
53         }
54        m_s_c_f(x,col,k,i,tempr);
55        m_s_c_f(y,col,k,i,tempi);
```

```
56          }
57        }
58    }
59
60    main()
61    {
62
63    //Matrices que se utilizaran para el calculo de la Transformada
64      float *m; //Matrix parte real
65      float *mi; //Matrix parte imaginaria
66      float *res;
67      float *resi;
68
69    //Se aparta memoria para el calculo
70      m=(float *)malloc(16*sizeof(float));
71      mi=(float *)malloc(16*sizeof(float));
72      res=(float *)malloc(16*sizeof(float));
73      resi=(float *)malloc(16*sizeof(float));
74
75      m[0]=0;
76      m[1]=4;
77      m[2]=0;
78      m[3]=-4;
79      m[4]=0;
80      m[5]=0;
81      m[6]=0;
82      m[7]=0;
83      m[8]=0;
84      m[9]=0;
85      m[10]=0;
86      m[11]=0;
87      m[12]=0;
88      m[13]=0;
89      m[14]=0;
90      m[15]=0;
91
92      mi[0]=0;
93      mi[1]=0;
94      mi[2]=0;
95      mi[3]=0;
96      mi[4]=0;
97      mi[5]=0;
98      mi[6]=0;
99      mi[7]=0;
100     mi[8]=0;
101     mi[9]=0;
102     mi[10]=0;
103     mi[11]=0;
104     mi[12]=0;
105     mi[13]=0;
106     mi[14]=0;
107     mi[15]=0;
108     fourier(0,4,4,m,mi,res,resi); //Se realiza la transformada de Fourier
109     printf("hola %f %f imaginarios: %f  %f",res[1],res[3],resi[1],resi[3]); //Imprimir
110   }
```

## D. *Programa* **funtions.c**

*Lectura de una imagen con formato raw, generación de filtro paso bajas, funciones de guardado y obtención de valores de memoria, transformada de Fourier discreta en una dimensión, transformada inversa discreta de Fourier en una dimensión y transpuesta de una matriz y envió y recepción de datos en forma paralela*
*Versión final*

```
1   /**
2    *
3    * FileName: funtions.c
4    * @authors: Eduardo Ávila Jiménez, Fernando Nahu Cantera Rubio, Eric R. Castañeda Caballero
5    * Date Created: Dic 16, 2004
6    *
7    **/
8
9   #include <stdio.h>
10  #include <stdlib.h>
11  #include <math.h>
12  #include "mpi.h"
13  #define PI2 6.2831
14
15  //Obtiene valores de la memoria unsigned char
16  unsigned char matrix_get_cell(unsigned char *matrix,int rows,int cols,int x, int y)
17  {
18   int valor_lineal;
19   valor_lineal=(y*cols+x);
20   return matrix[valor_lineal];
21  }
22
23  //Obtiene valores de la memoria float
24  float matrix_get_cell_float(float *matrix,int rows,int cols,int x,int y)
25  {
26   int valor_lineal;
27   valor_lineal=(y*cols+x);
28   return matrix[valor_lineal];
29  }
30
31  //Guardar valores unsigned char en la memoria
32  int matrix_set_cell(unsigned char *matrix,int rows,int cols,int x,int y,unsigned char val)
33  {
34   int valor_lineal;
35   valor_lineal=(y*cols+x);
36   matrix[valor_lineal]=val;
37   return 0;
38  }
39
40  //Guardar valores float en la memoria
41  int matrix_set_cell_float(float *matrix,int rows,int cols,int x,int y,float val)
42  {
43   int valor_lineal;
44   valor_lineal=(y*cols+x);
45   matrix[valor_lineal]=val;
46   return 0;
47  }
48
49  //Genera filtro promediador
50  void matrix_get_filtro(float *matrix,int tam,int tam2 ,int val)
51  {
```

```
52      int i,j,x,y=0,z;
53      float tmp;
54      tmp=(float)(val*val);
55      for(i=0;i<tam;i++)
56       {
57       j=0;
58       if(i<val)
59        {
60          for(x=0;x<val;x++)
61           {
62             matrix_set_cell_float(matrix,tam,tam2,j,i,((float)1.0/tmp));
63             j++;
64             y=1;
65          }
66        }
67       if(y==1)
68          z=tam2-val;
69       else
70          z=tam2;
71       for(x=0;x<z;x++)
72        {
73          matrix_set_cell_float(matrix,tam,tam2,j,i,0);
74           j=j+1;
75        }
76       }
77     }
78
79     //Genera una matriz de ceros
80     void matrix_zero(float *matrix, int rows, int cols)
81     {
82      int i, j;
83      for(i=0; i<rows; i++)
84       {
85       for(j=0; j<cols; j++)
86        {
87         matrix_set_cell_float(matrix, rows, cols, j, i,(float)0);
88        }
89       }
90     }
91
92     //Transpone una matriz
93     void matrix_transpone(float *origen, float *destino, int rows, int cols)
94     {
95      int i, j;
96      for(i=0; i<rows; i++)
97       {
98       for(j=0; j<cols; j++)
99        {
100         matrix_set_cell_float(destino, rows, cols, i, j, matrix_get_cell_float(origen, rows, cols, j, i));
101       }
102      }
103    }
104
105    //Convierte una matriz de float a unsigned char
106    void matrix_get_char(float *origen, unsigned char *destino, int rows, int cols)
107    {
108     unsigned char tmp;
109     int i, j;
110     for(i=0; i<rows; i++)
111      {
112      for(j=0; j<cols; j++)
113       {
```

```
114        tmp=(unsigned char)matrix_get_cell_float(origen, rows, cols, j, i);
115        matrix_set_cell(destino, rows, cols, j, i, tmp);
116       }
117     }
118   }
119
120   //Transformada Discreta de Fourier
121   void fourier(int renini, int renfin, int rows, int col, float *a, float *b, float *x, float *y)
122   {
123    long i, j, k;
124    float tempr;
125    float tempi;
126    float w=0;
127    float p, p1;
128
129    for(i=renini; i<=renfin; i++)
130     {
131      for(k=0; k<col; k++)
132       {
133        w=(float)((PI2*k)/col);
134        tempr=0; tempi=0;
135         for(j=0; j<col; j++)
136         {
137          p=(float)((matrix_get_cell_float(a, rows, col, j, i)*cos((double)w*j))+(matrix_get_cell_float(b, rows, col,
138   j, i)*sin((double)w*j)));
139          p1=(float)((matrix_get_cell_float(b, rows, col, j, i)*cos((double)w*j))-(matrix_get_cell_float(a, rows, col,
140   j, i)*sin((double)w*j)));
141         tempr=tempr+p;
142         tempi=tempi+p1;
143        }
144       matrix_set_cell_float(x, rows, col, k, i, tempr);
145       matrix_set_cell_float(y, rows, col, k, i, tempi);
146      }
147    }
148   }
149
150   //Transformada Discreta Inversade Fourier
151   void ifourier(int renini, int renfin, int rows, int col, float *a, float *b, float *x, float *y)
152   {
153    long i, j, k;
154    float tempr;
155    float tempri;
156    float w=0;
157    float p, p1;
158
159    for(i=renini; i<=renfin; i++)
160     {
161      for(k=0; k<col; k++)
162       {
163        w=(float)((PI2*k)/col);
164        tempr=0; tempri=0;
165        for(j=0; j<col; j++)
166         {
167          p=(float)((matrix_get_cell_float(a, rows, col, j, i)*cos((double)w*j))-(matrix_get_cell_float(b, rows, col, j,
168   i)*sin((double)w*j)));
169          p1=(float)((matrix_get_cell_float(a, rows, col, j, i)*sin((double)w*j))+(matrix_get_cell_float(b, rows, col,
170   j, i)*cos((double)w*j)));
171         tempr=tempr+p;
172         tempri=tempri+p1;
173        }
174      tempr=(tempr/(float)col);
175      tempri=(tempri/(float)col);
```

```
176      matrix_set_cell_float(x, rows, col, k, i, tempr);
177      matrix_set_cell_float(y, rows, col, k, i, tempri);
178     }
179    }
180   }
181
182   //Obtiene el valor absoluto de una matriz
183   void matrix_get_abs(float *m, float *r, int rows, int cols)
184   {
185    int i, j;
186    for(i=0; i<rows; i++)
187     {
188      for(j=0; j<cols; j++)
189      {
190       matrix_set_cell_float(r, rows, cols, j, i, fabs(matrix_get_cell_float(m, rows, cols, j, i)));
191      }
192     }
193   }
194
195   //Lee imagen
196   unsigned char * lee(char *nombre,int M,int N)
197   {
198    unsigned char *img;
199    FILE *f=fopen(nombre,"rb");
200    img=(unsigned char *)malloc((M*N*sizeof(unsigned char)));
201    fseek(f,(0*sizeof(unsigned char)*N),SEEK_SET);
202    if(fread(img,sizeof(unsigned char),(M*N),f)==0)
203     fclose(f);
204    return img;
205   }
206
207   //Convierte una matriz de unsigned char a float
208   float * dobles(unsigned char *image,int M,int N)
209   {
210    int i,j;
211    float *ims;
212    ims=(float *)malloc((M*N*sizeof(float)));
213    for(i=0;i<M;i++)
214     {
215      for(j=0;j<N;j++)
216       matrix_set_cell_float(ims,M,N,j,i,(float)((float)matrix_get_cell(image,M,N,j,i)/(float)1)+(float)1);
217     }
218    return ims;
219   }
220
221   //Escribe una matriz a un archivo
222   void escribe(char * nombre,unsigned char *img,int M,int N)
223   {
224    FILE *f=fopen(nombre,"wb");
225    int i;
226    fwrite(img,sizeof(unsigned char),M*N,f);
227    fclose(f);
228   }
229
230   //Envia datos a los esclavos para hacer un calculo y Recibe los resultados de esos calculos
231   void envia_recibe(float *mer,float *mei,float *mrr,float *mri,int row,int col)
232   {
233    int completerows=0;
234    int rv=1,i;
235    float *resultrow;
236    MPI_Status status;
237
```

```
238    rv=MPI_Bcast(mer,row*col,MPI_FLOAT,0,MPI_COMM_WORLD);
239    printf("Padre, Broadcast dice %d\n",rv);
240    rv=MPI_Bcast(mei,row*col,MPI_FLOAT,0,MPI_COMM_WORLD);
241    printf("Padre, Broadcast dice %d\n",rv);
242
243    resultrow=malloc((col+1)*sizeof(float));
244    while(completerows < (row*2))
245     {
246
247    MPI_Recv(resultrow,col+1,MPI_FLOAT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&s
248    tatus);
249      if(status.MPI_TAG==1)
250       {
251        completerows++;
252        for(i=0;i<col;i++)
253          matrix_set_cell_float(mrr,row,col,i,(int)resultrow[0],resultrow[i+1]);
254       }
255      else
256       {
257        completerows++;
258        for(i=0;i<col;i++)
259          matrix_set_cell_float(mri,row,col,i,(int)resultrow[0],resultrow[i+1]);
260       }
261     }
262    if(resultrow != NULL)
263     free(resultrow);
264    }
265
266    //El esclavo recibe los datos, realiza la Transformada Discreta de Fourier y envia los datos al maestro
267    void calcula_envia(float *mrr,float *mri,float *mer,float *mei,int row,int col,int firstrow,int lastrow)
268    {
269     int rv,i,x;
270     float *resultrow,*resultrow1;
271
272     rv=MPI_Bcast(mrr,row*col,MPI_FLOAT,0,MPI_COMM_WORLD);
273     rv=MPI_Bcast(mri,row*col,MPI_FLOAT,0,MPI_COMM_WORLD);
274
275     resultrow=malloc((col+1)*sizeof(float));
276     resultrow1=malloc((col+1)*sizeof(float));
277     fourier(firstrow,lastrow,row,col,mrr,mri,mer,mei);
278
279     for(i=firstrow;i<=lastrow;i++)
280      {
281       for(x=0;x<col;x++)
282        {
283         resultrow[x+1]=matrix_get_cell_float(mer,row,col,x,i);
284         resultrow1[x+1]=matrix_get_cell_float(mei,row,col,x,i);
285        }
286       resultrow[0]=(float)i;
287       resultrow1[0]=(float)i;
288       MPI_Send(resultrow,col+1,MPI_FLOAT,0,1,MPI_COMM_WORLD);
289       MPI_Send(resultrow1,col+1,MPI_FLOAT,0,2,MPI_COMM_WORLD);
290      }
291
292     if(resultrow != NULL)
293      free(resultrow);
294     if(resultrow1 != NULL)
295      free(resultrow1);
296    }
297
298    //El esclavo recibe los datos, realiza la Transformada Discreta de Fourier Inversa y envia los datos al maestro
299    void calcula_envia_inversa(float *mrr,float *mri,float *mer,float *mei,int row,int col,int firstrow,int lastrow)
```

```
300  {
301  int rv,i,x;
302  float *resultrow,*resultrow1;
303
304  rv=MPI_Bcast(mrr,row*col,MPI_FLOAT,0,MPI_COMM_WORLD);
305  rv=MPI_Bcast(mri,row*col,MPI_FLOAT,0,MPI_COMM_WORLD);
306
307  resultrow=malloc((col+1)*sizeof(float));
308  resultrow1=malloc((col+1)*sizeof(float));
309  ifourier(firstrow,lastrow,row,col,mrr,mri,mer,mei);
310
311  for(i=firstrow;i<=lastrow;i++)
312   {
113   for(x=0;x<col;x++)
314    {
315     resultrow[x+1]=matrix_get_cell_float(mer,row,col,x,i);
316     resultrow1[x+1]=matrix_get_cell_float(mei,row,col,x,i);
317    }
318   resultrow[0]=(float)i;
319   resultrow1[0]=(float)i;
320   MPI_Send(resultrow,col+1,MPI_FLOAT,0,1,MPI_COMM_WORLD);
321   MPI_Send(resultrow1,col+1,MPI_FLOAT,0,2,MPI_COMM_WORLD);
322  }
323
324  if(resultrow != NULL)
325   free(resultrow);
326  if(resultrow1 != NULL)
327   free(resultrow1);
328  }
```

## E. *Programa* **unisuavizado.c**
### *Suavizado de una imagen*
### *Versión final, uniprocesador*

```
1    /**
2     *
3     * FileName: unisuavizado.c
4     * @authors: Eduardo Ávila Jiménez, Fernando Nahu Cantera Rubio, Eric R. Castañeda Caballero
5     * Date Created: Dic 16, 2004
6     *
7     **/
8
9    #include <stdio.h>
10   #include <time.h>
11   #include <malloc.h>
12   #include <ctype.h>
13   #include "funtions.c"
14
15   main(int argc,char **argv)
16   {
17     clock_t comienzo;
18     unsigned char *im; //contendra la imagen leida en formato de enteros
19     unsigned char *imfinal;
20     //se definen los valores para el tamaño de imagen
21     int ROWS, COLS;
22
23     //para la parte real
24     float *filtro;
25     float *imx;//contendra la imagen leida en formato de flotantes
26
```

```
27     //para los valores imaginarios
28     float *fimg;
29     float *imagenimg;
30
31     //para los resultados
32     //filtro
33     float *fresreal;
34     float *fresim;
35
36      //imagen
37     float *imagenreal;
38     float *imagencom;
39
40      //resultado final
41     float *final;
42     float *finalimg;
43     float *ift;
44     float *ifti;
45
46   //para  las transpuestas
47      float *filtrotrans;
48      float *ftransimg;
49      float *imtransreal;
50      float *imtransimg;
51
52     int i,j,k;
53     int tamf;
54     char *nombre;
55
56   //para la multiplicacion
57   float elem1,elem2,elem3,elem4,suma,sumai;
58
59   //se da tratamiento a los argumentos del main
60   if(argc==5)
61   {
62      nombre=argv[1];
63      ROWS =atoi(argv[2]);
64      COLS =atoi(argv[3]);
65      tamf = atoi(argv[4]);
66   }
67   else//Valores por default
68   {
69      nombre="lena256.raw";
70      ROWS =256;
71      COLS =256;
72      tamf=3;
73   }
74
75     //se lee la imagen
76     im=lee(nombre,ROWS,COLS);
77
78    //apartados de memoria para el filtro
79    filtro=(float *)malloc((ROWS*COLS*sizeof(float)));
80    fimg=(float *)malloc((ROWS*COLS*sizeof(float)));
81
82     //apartados de memoria para los resultados
83    fresreal=(float *)malloc(ROWS*COLS*sizeof(float));
84     fresim=(float *)malloc(ROWS*COLS*sizeof(float));
85
86   //se comienza el conteo de segundos
87   comienzo=clock();
88
```

```
89    matrix_zero(fimg,ROWS,COLS);//se llena de ceros la parte imaginaria del filtro
90    matrix_get_filtro(matrix1,COLS,COLS,tamf); //genera el filtro
91
92    fourier(0,256,ROWS,COLS,filtro,fimg,fresreal,fresim);//transformada de fourier en una dimension
93
94    //liberando memoria del filtro
95    free(filtro);
96    free(fimg);
97
98    //apartado de memoria para las transpuestas
99    filtrotrans=(float *)malloc(ROWS*COLS*sizeof(float));
100   ftransimg=(float *)malloc(ROWS*COLS*sizeof(float));
101
102   //transponiendo matrices
103   matrix_transpone(fresreal,filtrotrans,ROWS,COLS);
104   matrix_transpone(fresim,ftransimg,ROWS,COLS);
105
106   //aplicando nuevamente la transformada
107
108   //se aparta memoria para los resultados de la transformada
109   filtro=(float *)malloc((ROWS*COLS*sizeof(float)));
110   fimg=(float *)malloc((ROWS*COLS*sizeof(float)));
111
112   fourier(0,256,ROWS,COLS,filtrotrans,ftransimg,filtro,fimg); //transformada de fourier en una dimension
113
114   free(filtrotrans);
115   free(ftransimg);
116
117   //Memoria para transponer matices
118   filtrotrans=(float *)malloc(ROWS*COLS*sizeof(float));
119   ftransimg=(float *)malloc(ROWS*COLS*sizeof(float));
120
121   //transponiendo para tener las matrices correctas
122
123   matrix_transpone(filtro,filtrotrans,ROWS,COLS);
124   matrix_transpone(fimg,ftransimg,ROWS,COLS);
125
126   free(filtro);
127   free(fimg);
128
129   //la transformada del filtro esta en filtrotrans y ftransimg
130
131   //apartado de espacio para parte imaginaria de la imagen
132   imagenimg =(float *)malloc((ROWS*COLS*sizeof(float)));
133
134   matrix_zero(imagenimg,ROWS,COLS); //se llena de ceros la parte imaginaria de la imagen
135   imx=dobles(im,ROWS,COLS);//de la imagen leida se pasa a flotantes
136
137   //resultados
138   imagenreal=(float *)malloc(ROWS*COLS*sizeof(float));
139   imagencom=(float *)malloc(ROWS*COLS*sizeof(float));
140
141   //sigue transformada de fourier
142   fourier(0,256,ROWS,COLS,imx,imagenimg,imagenreal,imagencom);//transformada de fourier en una
143   dimension
144   free(imx);
145   free(imagenimg);
146
147   //memoria para las transposiciones
148   imtransreal=(float *)malloc(ROWS*COLS*sizeof(float));
149   imtransimg=(float *)malloc(ROWS*COLS*sizeof(float));
150
```

```
151   //transponiendo matrices
152   matrix_transpone(imagenreal,imtransreal,ROWS,COLS);
153   matrix_transpone(imagencom,imtransimg,ROWS,COLS);
154
155   //aplicando nuevamente la transformada
156
157   //apartado de memoria para el resultado de la transformada
158   imx=(float *)malloc((ROWS*COLS*sizeof(float)));
159   imagenimg=(float *)malloc((ROWS*COLS*sizeof(float)));
160
161   fourier(0,256,ROWS,COLS,imtransreal,imtransimg,imx,imagenimg);//transformada en una dimension
162
163   free(imtransreal);
164   free(imtransimg);
165
166   //memoria para la transposicion de la imagen
167   imtransreal=(float *)malloc(ROWS*COLS*sizeof(float));
168   imtransimg=(float *)malloc(ROWS*COLS*sizeof(float));
169
170   //transponiendo para tener las matrices correctas
171   matrix_transpone(imx,imtransreal,ROWS,COLS);
172   matrix_transpone(imagenimg,imtransimg,ROWS,COLS);
173
174   free(imx);
175   free(imagenimg);
176
177   //la transformada de la imagen esta en imtransreal e imtransimg
178
179   //memoria para el resultado final de multiplicar el filtro por la imagen (parte real e imaginaria)
180   final=(float *)malloc(ROWS*COLS*sizeof(float));
181   finalimg=(float *)malloc(ROWS*COLS*sizeof(float));
182
183   //sigue multiplicacion de matrices
184   for(i=0;i<ROWS;i++)
185     {
186      for(j=0;j<COLS;j++)
187       {
188          suma=0;
189          sumai=0;
190          elem1=matrix_get_cell_float(imtransreal,ROWS,COLS,j,i);
191          elem2=matrix_get_cell_float(imtransimg,ROWS,COLS,j,i);
192          elem3=matrix_get_cell_float(filtrotrans,ROWS,COLS,j,i);
193          elem4=matrix_get_cell_float(ftransimg,ROWS,COLS,j,i);
194          suma=(elem1*elem3)-(elem2*elem4);
195          sumai=(elem1*elem4)+(elem2*elem3);
196          matrix_set_cell_float(final,ROWS,COLS,j,i,suma);
197          matrix_set_cell_float(finalimg,ROWS,COLS,j,i,sumai);
198       }
199     }
200
201   free(filtrotrans);
202   free(ftransimg);
203   free(imtransreal);
204   free(imtransimg);
205
206   //memoria necesaria para transponer el resultado de la multiplicacion
207   imtransreal=(float *)malloc(ROWS*COLS*sizeof(float));
208   imtransimg=(float *)malloc(ROWS*COLS*sizeof(float));
209
210   //transponiendo matrices
211   matrix_transpone(final,imtransreal,ROWS,COLS);
212   matrix_transpone(finalimg,imtransimg,ROWS,COLS);
```

```
213
214 free(final);
215 free(finalimg);
216
217 //memoria para el resultado de la transformada inversa de fourier
218 ift=(float *)malloc(ROWS*COLS*sizeof(float));
219 ifti=(float *)malloc(ROWS*COLS*sizeof(float));
220
221 //transformada inversa de fourier
222 ifourier(0,256,ROWS,COLS,imtransreal,imtransimg,ift,ifti);
223
224 free(imtransreal);
225 free(imtransimg);
226
227 //memoria para transponer
228 imtransreal=(float *)malloc(ROWS*COLS*sizeof(float));
229 imtransimg=(float *)malloc(ROWS*COLS*sizeof(float));
230
231 //transponiendo las matrices
232 matrix_transpone(ift,imtransreal,ROWS,COLS);
233 matrix_transpone(ifti,imtransimg,ROWS,COLS);
234
235 free(ift);
236 free(ifti);
237
238 //memoria para la transformada inversa
239 ift=(float *)malloc(ROWS*COLS*sizeof(float));
240 ifti=(float *)malloc(ROWS*COLS*sizeof(float));
241
242 //transformada inversa de fourier
243 ifourier(0,256,ROWS,COLS,imtransreal,imtransimg,ift,ifti);
244
245 free(imtransreal);
246 free(imtransimg);
247
248 free(im);
249 //memoria para guardar el resultado del procesamiento
250 im=(unsigned char *)malloc(ROWS*COLS*sizeof(unsigned char));
251
252 //memoria para el valor absoluto del resultado
253 imtransreal=(float *)malloc(ROWS*COLS*sizeof(float));
254 //obtiene el valor absoluto
255 matrix_get_abs(ift,imtransreal,ROWS,COLS);
256
257 free(ift);
258
259 //se convierte de formato de flotantes a enteros
260 matrix_get_char(imtransreal,im,ROWS,COLS);
261
262 //escribe la imagen en char a un archivo .raw
263 escribe("total.raw",im,ROWS,COLS);
264
265
266 free(im);
267 //se imprime el tiempo de calculo
268 printf("Tiempo: %g s\n", (clock()-comienzo)/(double)CLOCKS_PER_SEC);
269
270 printf("\nllego al final \n");
271 }
```

## F. *Programa* **suavizado.c**
### *Suavizado de una imagen*
### *Versión final, utilizando MPI*

```
1   /**
2    *
3    * FileName: suavizado.c
4    * @authors: Eduardo Ávila Jiménez, Fernando Nahu Cantera Rubio, Eric R. Castañeda Caballero
5    * Date Created: Feb 7, 2005
6    *
7    **/
8
9   #include "mpi.h"
10  #include "funtions.c"
11  #include <stdio.h>
12  #include <math.h>
13  #include <stdlib.h>
14  #include <unistd.h>
15
16  int main(int argc, char **argv)
17  {
18   int localid, numprocs, namelen, rv, row, col,tamf;
19   //Matrices que utilizaremos para realizar los calculos
20   float *matrix1=NULL;
21   float *matrix2=NULL;
22   float *matrix3=NULL;
23   float *matrix4=NULL;
24   float *matrix5=NULL;
25   float *matrix6=NULL;
26
27  //Variables auxiliares
28   unsigned char *im=NULL;
29   double startwtime, endwtime;
30   char processor_name[MPI_MAX_PROCESSOR_NAME];
31   int x, y, i, k;
32   int j;
33   int partitions, firstrow, lastrow, rowstodo;
34   float *resultrow,*resultrow1;
35   int elem1, elem2, suma;
36   int completerows;
37   int dimension=256;
38   char optchar;
39   int opt_print=0;
40   int seed=(unsigned int) (time (0) / 2);
41   MPI_Status status;
42   char *nombre;
43
44   //Variables auxiliares para la multiplicacion
45   float elem11,elem22,elem3,elem4,suma1,sumai;
46
47   //Inicializar MPI
48   MPI_Init(&argc, &argv);
49   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
50   MPI_Comm_rank(MPI_COMM_WORLD, &localid);
51   MPI_Get_processor_name(processor_name, &namelen);
52
53   printf("soy el proceso %d de %d en %s \n", localid, numprocs, processor_name);
54
55   //Se verifican los argumentos de la linea de comandos
56   if(argc==5)
57   {
```

```
58      nombre=argv[1];
59      col=atoi(argv[2]);
60      row=atoi(argv[3]);
61      tamf=atoi(argv[4]);
62     }
63     else //Valores por default
64     {
65      nombre="lena256.raw";
66      col=256;
67      row=256;
68      tamf=3;
69     }
70
71    //Inicia el MAESTRO
72    if(localid==0)
73     {
74      srandom((unsigned int) (time(0) / 2));
75      //Se aparta memoria para leer la imagen y generar el filtro
76      matrix1=malloc(row * col * sizeof(float));
77      matrix2=malloc(row * col * sizeof(float));
78      matrix3=malloc(row * col * sizeof(float));
79      matrix4=malloc(row * col * sizeof(float));
80
81      //Se libera memoria
82      if(matrix1 == NULL || matrix2 == NULL || matrix3==NULL || matrix4==NULL)
83       {
84        MPI_Finalize();
85        if(matrix1 !=NULL) free(matrix1);
86        if(matrix2 !=NULL) free(matrix2);
87        if(matrix3 !=NULL) free(matrix3);
88        if(matrix4 !=NULL) free(matrix4);
89        exit(1);
90       }
91
92      // --------------------------------- FILTRO ---------------------------------
93      //Se genera el filtro
94      matrix_get_filtro(matrix1,row,col,tamf);
95      //se genera una matriz de cero para fungir como parte imaginaria del filtro
96      matrix_zero(matrix2,row,col);
97
98      startwtime=MPI_Wtime();//Tiempo
99
100     //Se le envia el filtro a los esclavos para que realicen la Transformada Discreta de Fourier y recibe
101     //los resultados del calculo
102     envia_recibe(matrix1,matrix2,matrix3,matrix4,row,col);
103
104     //Se libera memoria
105     if(matrix1 !=NULL) free(matrix1);
106     if(matrix2 !=NULL) free(matrix2);
107
108     //Se aparta memoria
109     matrix1=malloc(row * col * sizeof(float));
110     matrix2=malloc(row * col * sizeof(float));
111
112     //Se transponen las matrices que se reciben
113     matrix_transpone(matrix3,matrix1,row,col);
114     matrix_transpone(matrix4,matrix2,row,col);
115
116     //Se libera memoria
117     if(matrix3 !=NULL) free(matrix3);
118     if(matrix4 !=NULL) free(matrix4);
119
```

```
120    //Se aparta memoria
121    matrix3=malloc(row * col * sizeof(float));
122    matrix4=malloc(row * col * sizeof(float));
123
124    //Se le envia la matriz transpuesta a los esclavos para que realicen de nuevo la Transformada Discreta de
125   Fourier y recibe los resultados del calculo
126    envia_recibe(matrix1,matrix2,matrix3,matrix4,row,col);
127
128    //Se libera memoria
129    if(matrix1 !=NULL) free(matrix1);
130    if(matrix2 !=NULL) free(matrix2);
131
132    //Se aparta memoria
133    matrix1=malloc(row * col * sizeof(float));
134    matrix2=malloc(row * col * sizeof(float));
135
136    //Se transponen las matrices que se reciben
137    matrix_transpone(matrix3,matrix1,row,col);
138    matrix_transpone(matrix4,matrix2,row,col);
139
140    //El resultado de la transformada del filtro en matrix1 y matrix2, parte real e imaginaria respectivamente
141
142    // ---------------------------------- IMAGEN ----------------------------------
143    //Se libera memoria
144    if(matrix3 !=NULL) free(matrix3);
145    if(matrix4 !=NULL) free(matrix4);
146
147    //Se aparta memoria para la imagen
148    matrix3=malloc(row * col * sizeof(float));
149    matrix4=malloc(row * col * sizeof(float));
150    matrix5=malloc(row * col * sizeof(float));
151    matrix6=malloc(row * col * sizeof(float));
152
153    //Se aparta la memoria para leer la imagen
154    im=(unsigned char *)malloc(row*col*sizeof(unsigned char));
155    //Se lee la iamgen
156    im=lee(nombre,row,col);
157    //La convertimos en float
158    matrix3=dobles(im,row,col);
159
160    if(im !=NULL)
161      free(im);
162
163    //se genera una matriz de cero para fungir como parte imaginaria de la imagen
164    matrix_zero(matrix4,row,col);
165
166    //Se le envia la imagen a los esclavos para que realicen la Transformada Discreta de Fourier y recibe
167    //los resultados del calculo
168    envia_recibe(matrix3,matrix4,matrix5,matrix6,row,col);
169
170    //Se libera memoria
171    if(matrix3 !=NULL) free(matrix3);
172    if(matrix4 !=NULL) free(matrix4);
173
174    //Se aparta memoria
175    matrix3=malloc(row * col * sizeof(float));
176    matrix4=malloc(row * col * sizeof(float));
177
178    //Se transponen las matrices que se reciben
179    matrix_transpone(matrix5,matrix3,row,col);
180    matrix_transpone(matrix6,matrix4,row,col);
181
```

```
182     //Se libera memoria
183     if(matrix5 !=NULL) free(matrix5);
184     if(matrix6 !=NULL) free(matrix6);
185
186     //Se aparta memoria
187     matrix5=malloc(row * col * sizeof(float));
188     matrix6=malloc(row * col * sizeof(float));
189
190     //Se le envia la matriz transpuesta a los esclavos para que realicen de nuevo la Transformada Discreta de
191    Fourier y recibe
192     //los resultados del calculo
193     envia_recibe(matrix3,matrix4,matrix5,matrix6,row,col);
194
195     //Se libera memoria
196     if(matrix3 !=NULL) free(matrix3);
197     if(matrix4 !=NULL) free(matrix4);
198
199     //Se aparta memoria
200     matrix3=malloc(row * col * sizeof(float));
201     matrix4=malloc(row * col * sizeof(float));
202
203     //Se transponen las matrices que se reciben
204     matrix_transpone(matrix5,matrix3,row,col);
205     matrix_transpone(matrix6,matrix4,row,col);
206
207     //Se libera memoria
208     if(matrix5 !=NULL) free(matrix5);
209     if(matrix6 !=NULL) free(matrix6);
210
211     //El resultado de la transformada de la imagen en matrix3 y matrix4, parte real e imaginaria respectivamente
212
213     // --------------------------------- MULTIPLICACION ---------------------------------
214
215     //Se aparta memoria para guardar el resultado ahi
216     matrix5=malloc(row * col * sizeof(float));
217     matrix6=malloc(row * col * sizeof(float));
218
219     completerows=0;
220     //Se envia el filtro y la imagen a los esclavos
221     rv=MPI_Bcast(matrix1, row * col, MPI_FLOAT, 0, MPI_COMM_WORLD);
222     printf("Root, Broadcast said %d\n", rv);
223     rv=MPI_Bcast(matrix2, row * col, MPI_FLOAT, 0, MPI_COMM_WORLD);
224     printf("Rott, Broadcast said %d\n", rv);
225     rv=MPI_Bcast(matrix3, row * col, MPI_FLOAT, 0, MPI_COMM_WORLD);
226     printf("Root, Broadcast said %d\n", rv);
227     rv=MPI_Bcast(matrix4, row * col, MPI_FLOAT, 0, MPI_COMM_WORLD);
228     printf("Rott, Broadcast said %d\n", rv);
229
230     //Se obtienen los valores enviados por los esclavos y se almacenan en la memoria apartada
231     resultrow= malloc((col+1) * sizeof(float));
232     while(completerows < (row*2))
233      {
234       MPI_Recv(resultrow,col+1,MPI_FLOAT,MPI_ANY_SOURCE,MPI_ANY_TAG, MPI_COMM_WORLD,
235    &status);
236       if(status.MPI_TAG==1)//Se obtiene parte real
237        {
238         completerows++;
239         for(i=0; i<col; i++)
240          {
241           matrix_set_cell_float(matrix5, row, col, i,(int)resultrow[0],resultrow[i + 1]);
242          }
243        }
```

```
244     else//Se obtiene parte imaginaria
245     {
246       completerows++;
247       for(i=0; i<col; i++)
248       {
249         matrix_set_cell_float(matrix6, row, col, i,(int)resultrow[0],resultrow[i + 1]);
250       }
251     }
252   }
253
254   //Se libera memoria
255   if(matrix1 != NULL) free(matrix1);
256   if(matrix2 != NULL) free(matrix2);
257   if(matrix3 != NULL) free(matrix3);
258   if(matrix4 != NULL) free(matrix4);
259   if(resultrow != NULL) free(resultrow);
260
261   // ---------------------------------- OBTIENER IMAGEN SUAVIZADA ----------------------------------
262
263   //Se aparta memoria
264   matrix1=malloc(row * col * sizeof(float));
265   matrix2=malloc(row * col * sizeof(float));
266
267   //Se transponen las matrices que se reciben de la multiplicacion
268   matrix_transpone(matrix5,matrix1,row,col);
269   matrix_transpone(matrix6,matrix2,row,col);
270
271   //Se libera memoria
272   if(matrix5 != NULL) free(matrix5);
273   if(matrix6 != NULL) free(matrix6);
274
275   //Se aparta memoria
276   matrix3=malloc(row * col * sizeof(float));
277   matrix4=malloc(row * col * sizeof(float));
278
279   //Se le envia la multyiplicacion a los esclavos para que realicen la Transformada Discreta de Fourier y recibe
280   //los resultados del calculo
281   envia_recibe(matrix1,matrix2,matrix3,matrix4,row,col);
282
283   //Se libera memoria
284   if(matrix1 !=NULL) free(matrix1);
285   if(matrix2 !=NULL) free(matrix2);
286
287   //Se aparta memoria
288   matrix1=malloc(row * col * sizeof(float));
289   matrix2=malloc(row * col * sizeof(float));
290
291   //Se transponen las matrices que se reciben
292   matrix_transpone(matrix3,matrix1,row,col);
293   matrix_transpone(matrix4,matrix2,row,col);
294
295   //Se libera memoria
296   if(matrix3 !=NULL) free(matrix3);
297   if(matrix4 !=NULL) free(matrix4);
298
299   //Se aparta memoria
300   matrix3=malloc(row * col * sizeof(float));
301   matrix4=malloc(row * col * sizeof(float));
302
303   //Se le envia la matriz transpuesta a los esclavos para que realicen de nuevo la Transformada Discreta de
304 Fourier y recibe
305   //los resultados del calculo
```

```
306    envia_recibe(matrix1,matrix2,matrix3,matrix4,row,col);
307
308    //Se libera memoria
309    if(matrix1 !=NULL) free(matrix1);
310    if(matrix2 !=NULL) free(matrix2);
311    if(matrix4 !=NULL) free(matrix4);
312
313    //Se obtiene el valor absoluto de la matriz que contiene a la imagen suavizada
314    matrix4=malloc(row*col*sizeof(float));
315    matrix_get_abs(matrix3,matrix4,row,col);
316
317    //Se libera memoria
318    if(matrix3 !=NULL) free(matrix3);
319
320    //Se obtiene los valores de la imagen en char
321    im=(unsigned char *)malloc(row*col*sizeof(unsigned char));
322    matrix_get_char(matrix4,im,row,col);
323
324    //Se manda es escribir la imagen
325    escribe("total.raw",im,row,col);
326
327    endwtime=MPI_Wtime();//Tiempo
328
329    printf("wall clock time= %f \n", endwtime - startwtime);
330
231    //Se libera memoria
332    if(matrix1 !=NULL) free(matrix1);
333    if(matrix2 !=NULL) free(matrix2);
334    if(matrix3 !=NULL) free(matrix3);
335   }
336   else //Empieza el ESCLAVO
337   {
338    //Se aparta memoria para los calculos a realizar
339    matrix1=malloc(row * col * sizeof(float));
340    matrix2=malloc(row * col * sizeof(float));
341    matrix3=malloc(row * col * sizeof(float));
342    matrix4=malloc(row * col * sizeof(float));
343
344    //Se libera memoria
345    if(matrix1 == NULL || matrix2 == NULL || matrix3==NULL || matrix4==NULL)
346     {
347      MPI_Finalize();
348      if(matrix1 != NULL) free(matrix1);
349      if(matrix2 != NULL) free(matrix2);
350      if(matrix3 != NULL) free(matrix3);
351      if(matrix4 != NULL) free(matrix4);
352      exit(1);
353     }
354
355    //Se calcula cuantos renglones van a operar cada esclavo
356    partitions = numprocs - 1;
357    rowstodo = (int) (row / partitions);
358    firstrow=rowstodo * (localid - 1);
359    lastrow=firstrow + rowstodo - 1;
360    if(localid==numprocs - 1)
361     {
362      lastrow=lastrow + (row % partitions);
363     }
364
365    //Obtiene las matrices del maestro, realiza la Transformada Discreta de Fourier y envia
366    //los resultados del calculo
367    calcula_envia(matrix1,matrix2,matrix3,matrix4,row,col,firstrow,lastrow);
```

```
368
369     //Se libera memoria
370     if(matrix1!=NULL) free(matrix1);
371     if(matrix2!=NULL) free(matrix2);
372     if(matrix3!=NULL) free(matrix3);
373     if(matrix4!=NULL) free(matrix4);
374
375     //Tiempo muerto para los esclavos
376     for(i=0;i<(row*col);i++)
377      {
378       ;
379      }
380
381     //Se aparta memoria para los calculos a realizar
382     matrix1=malloc(row * col * sizeof(float));
383     matrix2=malloc(row * col * sizeof(float));
384     matrix3=malloc(row * col * sizeof(float));
385     matrix4=malloc(row * col * sizeof(float));
386
387     //Se libera memoria
388     if(matrix1 == NULL || matrix2 == NULL || matrix3==NULL || matrix4==NULL)
389      {
390       MPI_Finalize();
391       if(matrix1 != NULL) free(matrix1);
392      if(matrix2 != NULL) free(matrix2);
393      if(matrix3 != NULL) free(matrix3);
394      if(matrix4 != NULL) free(matrix4);
395      exit(1);
396      }
397
398     //Obtiene las matrices del maestro, realiza la Transformada Discreta de Fourier y envia
399     //los resultados del calculo
400     calcula_envia(matrix1,matrix2,matrix3,matrix4,row,col,firstrow,lastrow);
401
402     //Tiempo muerto para los esclavos
403     for(i=0;i<(row*col);i++)
404      {
405       ;
406      }
407
408     //Se libera memoria
409     if(matrix1!=NULL) free(matrix1);
410     if(matrix2!=NULL) free(matrix2);
411     if(matrix3!=NULL) free(matrix3);
412     if(matrix4!=NULL) free(matrix4);
413
414     //Se aparta memoria para los calculos a realizar
415     matrix1=malloc(row * col * sizeof(float));
416     matrix2=malloc(row * col * sizeof(float));
417     matrix3=malloc(row * col * sizeof(float));
418     matrix4=malloc(row * col * sizeof(float));
419
420     //Se libera memoria
421     if(matrix1 == NULL || matrix2 == NULL || matrix3==NULL || matrix4==NULL)
422      {
423       MPI_Finalize();
424       if(matrix1 != NULL) free(matrix1);
425       if(matrix2 != NULL) free(matrix2);
426       if(matrix3 != NULL) free(matrix3);
427       if(matrix4 != NULL) free(matrix4);
428       exit(1);
429      }
```

```
430
431     //Obtiene las matrices del maestro, realiza la Transformada Discreta de Fourier y envia
432     //los resultados del calculo
433     calcula_envia(matrix1,matrix2,matrix3,matrix4,row,col,firstrow,lastrow);
434
435     //Tiempo muerto para los esclavos
436     for(i=0;i<(row*col);i++)
437      {
438       ;
439      }
440
441     //Se libera memoria
442     if(matrix1!=NULL) free(matrix1);
443     if(matrix2!=NULL) free(matrix2);
444     if(matrix3!=NULL) free(matrix3);
445     if(matrix4!=NULL) free(matrix4);
446
447     //Se aparta memoria para los calculos a realizar
448     matrix1=malloc(row * col * sizeof(float));
449     matrix2=malloc(row * col * sizeof(float));
450     matrix3=malloc(row * col * sizeof(float));
451     matrix4=malloc(row * col * sizeof(float));
452
453     //Se libera memoria
454     if(matrix1 == NULL || matrix2 == NULL || matrix3==NULL || matrix4==NULL)
455      {
456       MPI_Finalize();
457       if(matrix1 != NULL) free(matrix1);
458       if(matrix2 != NULL) free(matrix2);
459       if(matrix3 != NULL) free(matrix3);
460       if(matrix4 != NULL) free(matrix4);
461        exit(1);
462      }
463
464     //Obtiene las matrices del maestro, realiza la Transformada Discreta de Fourier y envia
465     //los resultados del calculo
466     calcula_envia(matrix1,matrix2,matrix3,matrix4,row,col,firstrow,lastrow);
467
468     //Tiempo muerto para los esclavos
469     for(i=0;i<(row*col);i++)
470      {
471       ;
472      }
473
474     //Se libera memoria
475     if(matrix1!=NULL) free(matrix1);
476     if(matrix2!=NULL) free(matrix2);
477     if(matrix3!=NULL) free(matrix3);
478     if(matrix4!=NULL) free(matrix4);
479
480     //Se aparta memoria para los calculos a realizar
481     matrix1=malloc(row * col * sizeof(float));
482     matrix2=malloc(row * col * sizeof(float));
483     matrix3=malloc(row * col * sizeof(float));
484     matrix4=malloc(row * col * sizeof(float));
485
486     //Se libera memoria
487     if(matrix1 == NULL || matrix2 == NULL || matrix3==NULL || matrix4==NULL )
488      {
489       MPI_Finalize();
490       if(matrix1 != NULL) free(matrix1);
491       if(matrix2 != NULL) free(matrix2);
```

```
492      if(matrix3 != NULL) free(matrix3);
493      if(matrix4 != NULL) free(matrix4);
494      exit(1);
495     }
496
497     //Se obtiene las matrices del maestro para realizar la multplicacion de estas
498     rv= MPI_Bcast(matrix1, row * col, MPI_FLOAT, 0, MPI_COMM_WORLD);
499     rv= MPI_Bcast(matrix2, row * col, MPI_FLOAT, 0, MPI_COMM_WORLD);
500     rv= MPI_Bcast(matrix3, row * col, MPI_FLOAT, 0, MPI_COMM_WORLD);
501     rv= MPI_Bcast(matrix4, row * col, MPI_FLOAT, 0, MPI_COMM_WORLD);
502
503     //Se aparta memoria para los resultados de la multiplicacion
504     resultrow = malloc((col + 1) * sizeof(float));
505     resultrow1= malloc((col+1) * sizeof(float));
506
507     //Se realiza la multplicacion
508     for(i=firstrow; i <= lastrow; i++)
509      {
510       for(x=0; x< col; x++)
511       {
512        suma1=0;
513        sumai=0;
514        elem11=matrix_get_cell_float(matrix3,row,col,x,i);
515        elem22=matrix_get_cell_float(matrix4,row,col,x,i);
516        elem3=matrix_get_cell_float(matrix1,row,col,x,i);
517        elem4=matrix_get_cell_float(matrix2,row,col,x,i);
518        suma1=(elem11*elem3)-(elem22*elem4);
519        sumai=(elem11*elem4)+(elem22*elem3);
520        resultrow[x+1]=suma1;
521        resultrow1[x+1]=sumai;
522       }
523      resultrow[0]=(float)i;
524      resultrow1[0]=(float)i;
525      //Se envian los resultados al maestro
526      MPI_Send(resultrow, col + 1, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
527      MPI_Send(resultrow1,col+1,MPI_FLOAT,0,2,MPI_COMM_WORLD);
528     }
529
530     //Se libera memoria
531     if(matrix1!=NULL) free(matrix1);
532     if(matrix2!=NULL) free(matrix2);
533     if(matrix3!=NULL) free(matrix3);
534     if(matrix4!=NULL) free(matrix4);
535     if(resultrow!=NULL) free(resultrow);
536     if(resultrow1 != NULL) free(resultrow1);
537
538     //Tiempo muerto para los esclavos
539     for(i=0;i<(row*col);i++)
540      {
541       ;
542      }
543
544     //Se aparta memoria para los calculos a realizar
545     matrix1=malloc(row * col * sizeof(float));
546     matrix2=malloc(row * col * sizeof(float));
547     matrix3=malloc(row * col * sizeof(float));
548     matrix4=malloc(row * col * sizeof(float));
549
550     //Se libera memoria
551     if(matrix1 == NULL || matrix2 == NULL || matrix3==NULL || matrix4==NULL)
552      {
553       MPI_Finalize();
```

```
554     if(matrix1 != NULL) free(matrix1);
555     if(matrix2 != NULL) free(matrix2);
556     if(matrix3 != NULL) free(matrix3);
557     if(matrix4 != NULL) free(matrix4);
558     exit(1);
559    }
560
561   //Obtiene las matrices del maestro, realiza la Transformada Discreta de Fourier Inversa y envia
562   //los resultados del calculo
563   calcula_envia_inversa(matrix1,matrix2,matrix3,matrix4,row,col,firstrow,lastrow);
564
565   //Se libera memoria
566   if(matrix1!=NULL) free(matrix1);
567   if(matrix2!=NULL) free(matrix2);
568   if(matrix3!=NULL) free(matrix3);
569   if(matrix4!=NULL) free(matrix4);
570
571   //Tiempo muerto para los esclavos
572   for(i=0;i<(row*col);i++)
573    {
574     ;.
575    }
576
577   //Se aparta memoria para los calculos a realizar
578   matrix1=malloc(row * col * sizeof(float));
579   matrix2=malloc(row * col * sizeof(float));
580   matrix3=malloc(row * col * sizeof(float));
581   matrix4=malloc(row * col * sizeof(float));
582
583   //Se libera memoria
584   if(matrix1 == NULL || matrix2 == NULL || matrix3==NULL || matrix4==NULL)
585    {
586     MPI_Finalize();
587     if(matrix1 != NULL) free(matrix1);
588     if(matrix2 != NULL) free(matrix2);
589     if(matrix3 != NULL) free(matrix3);
590     if(matrix4 != NULL) free(matrix4);
591     exit(1);
592    }
593
594   //Obtiene las matrices del maestro, realiza la Transformada Discreta de Fourier Inversa y envia
595   //los resultados del calculo
596   calcula_envia_inversa(matrix1,matrix2,matrix3,matrix4,row,col,firstrow,lastrow);
597
598   //Se libera memoria
599   if(matrix1!=NULL) free(matrix1);
600   if(matrix2!=NULL) free(matrix2);
601   if(matrix3!=NULL) free(matrix3);
602   if(matrix4!=NULL) free(matrix4);
603
604  }//Termina es ESCLAVO
605  MPI_Finalize();
606  return 0;
607  }//Fin del main
```