UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTA DE INGENIERÍA

ELABORACIÓN DE PRÁCTICAS DE PROGRAMACIÓN DISTRIBUIDA EN EL CLUSTER TIPO BEOWULF DE LA FACULTAD DE INGENIERÍA

TESIS PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

P R E S E N T A N

CHRISTIAN GONZÁLEZ FLORES
LISSETTE GONZÁLEZ FLORES
ERYK ROGELIO RAMÍREZ PONCE

DIRECTORA DE TESIS: ING. LAURA SANDOVAL MONTAÑO

CIUDAD UNIVERSITARIA MÉXICO, D.F., 2004

A **DIOS**, por brindarme la gran oportunidad de ser su hijo, por haber caminado a mi lado durante toda la vida. Estoy convencido que hacen falta más que palabras para poder plasmar lo inmensamente agradecido que estoy contigo y con la Virgen María Santísima. Gracias por todo, Señor.

A la **Virgen María**, por darme paz en los momentos difíciles, por su infinito amor y por su constante intercesión ante Dios.

A mis amados padres, Carmen Flores y José González, por ser los mejores, por sus consejos, su cariño, sus cuidados y su compañía; a mis hermanas, Lissette y Stephany, porque siempre han sido un ejemplo para mí.

A la Ing. Laura Sandoval Montaño, por la paciencia y el apoyo que me ha dado desde que la conozco.

A mis primos, Alejandra Ríos y Rafael González, y a mis tíos, Norma García y Gerardo Flores, por su ayuda incondicional.

-Christian González Flores.

A Dios

Porque ha estado a mi lado en cada paso de mi vida, a Él le dedico este trabajo, le doy gracias por apoyarme y entregarme su amor incondicional como sólo Él lo sabe entregar. Le doy gracias por todas las cosas que ha hecho de mí y porque sé que Él nunca me va a abandonar y siempre va a estar a mi lado así sea en la felicidad como en las dificultades, con Él quiero compartir toda mi vida.

A la Virgen María

Porque ella sabe lo mucho que la quiero, y con todo el corazón le doy gracias por escuchar mis ruegos cada noche cuando platicamos e interceder por mí ante su hijo. Por ser una madre buena, cariñosa y un ejemplo de mujer a la que admiro, a ella le dedico todo mi esfuerzo, mi trabajo y mi vida.

A mis Padres

A mi mamá Carmen, por apoyarme y desde niña impulsarme y guiarme para ser una persona responsable y de bien, además de consentirme y darme todo su cariño, estar siempre a mi lado y olvidarse de sí misma para darme lo mejor. Además de ser ella mi mejor amiga con quien comparto tantas cosas.

A mi papá José de Jesús, porque además de apoyo económico me ha dado su amor y me ha demostrado que las cosas que realmente valen se ganan con esfuerzo, él es un ejemplo de trabajo y dedicación porque durante toda su vida se ha esforzado por darle siempre lo mejor a mi familia.

A mis padres les dedico este trabajo porque es gracias a su apoyo que he llegado a concluirlo.

A mis Hermanos

A mi hermanita Stephany, porque ella es la alegría de la casa y desde que Dios nos la regaló sólo nos ha traído felicidad en mi hogar, porque nos permite distraernos del trabajo y pensar en volver a ser niños como ella. Porque todos la queremos mucho y sin ella mi familia no estaría completa.

A mi hermano Christian, por ser bueno y el más cariñoso de la familia, además por haber compartido desde niños todo, los juegos, la escuela e inclusive este trabajo de tesis que gracias a su apoyo logramos terminarlo.

A mis hermanos les dedico mi trabajo porque los quiero.

A mi novio

Eryk, porque hemos compartido ya mucho tiempo de nuestra vida, y porque lo conocí en una época muy especial de mi vida y es maravilloso que hayamos iniciado esta carrera juntos y la hayamos concluido amándonos como en el principio. A él le dedico mi trabajo porque lo amo.

A mi profesora

Laura Sandoval, por todo el apoyo, ayuda y comprensión para guiarnos en el desarrollo de la tesis, además de las pláticas amenas que teníamos en su cubículo, por ser una profesora que en realidad se interesa porque sus alumnos aprendan y por haber confiado en mí y preocuparse no sólo por apoyarnos en el campo académico sino también ponernos en contacto con gente en el campo laboral.

A mis tíos y a mis primos

A mis tíos Gerardo Flores, Norma García y Rodrigo Flores y a mis primos Rafael González y Alejandra Ríos, por su apoyo y cariño, además de confiar siempre en mí y por esto ayudarme con lo que estaba dentro de sus medios.

A mi Abuelita

Toña, porque aunque ya no esté aquí sé que ella está orgullosa de que haya logrado alcanzar esta meta, y porque cuando estuvo a mi lado siempre me dio su amor y cariño, además de momentos muy divertidos.

A todos ellos pero especialmente a Dios les dedico este trabajo.

-Lissette González Flores.

A Dios:

Por permitirme llegar hasta este momento tan importante de mi vida y lograr una meta más de las que me he propuesto alcanzar como parte de mi proyecto de vida. A Él le dedico mi trabajo, mi esfuerzo y los resultados que durante todos estos años he obtenido.

En Él me he apoyado en muchas ocasiones cuando he sentido que no encuentro la forma de resolver ciertos problemas que se me han presentado, y con su ayuda, siempre he encontrado la mejor manera de solucionarlos.

A mis padres:

Por todo lo que me han dado, pues sin su gran amor, su dedicación hacia mí y su forma de guiarme en los primeros pasos de mi vida, no hubiera podido realizar los sueños que hasta el día de hoy he alcanzado, y que me servirán para encontrar la manera de consumar los sueños futuros.

Gracias a mis hermanos, por su comprensión ante mi ausencia en casa. La búsqueda por conseguir mis metas me ha alejado un poco de ustedes, pero sé que me comprenden, y que siempre estamos unidos a pesar de la distancia.

A mi novia:

Por estar a mi lado cuando más la he necesitado, por su apoyo, por su comprensión y su cariño que me ha brindado en todo el tiempo que hemos compartido nuestras vidas. Su compañía, su forma de ser y su carácter me han enseñado muchas cosas. Sin ella, toda mi vida hubiera sido diferente y le agradezco estar a mi lado todo este tiempo y tomarme de la mano para seguir juntos por la vida.

También le agradezco a toda su familia, que me ha dado su amistad y su confianza, ya que me han hecho sentir como un miembro más de su familia, de manera muy particular a su hermano con el que he establecido una gran amistad.

A la ingeniera Laura Sandoval Montaño:

Por todo el apoyo que me dio durante el desarrollo de este trabajo, además de todo lo que aprendí con ella como maestra. La considero una de las mejores profesoras que he tenido en toda mi vida académica además de ser una excelente persona dentro y fuera del salón de clases.

-Eryk Rogelio Ramírez Ponce.

Contenido.

Introducción

La introducción de las computadoras en nuestras vidas trajo consigo una gran revolución que, en la actualidad, no sólo concierne al ámbito tecnológico. Existen muchas cosas que serían inconcebibles sin la ineludible utilización de estos procesadores de información.

La versatilidad que tienen los ordenadores ha permitido que su uso se extienda a muy diversas áreas de aplicación, es por ello que la complejidad de las tareas que éstos deben realizar así como la cantidad de información que manejan son dos factores que tienden a incrementarse, y que exigen una demanda directa de poder computacional. Una solución inmediata ante tal demanda es la adquisición de equipo de cómputo más potente. La anterior, a pesar de ser una posible solución, no suele ser siempre la mejor ya que los nuevos equipos están relacionados con costos elevados y en ocasiones hasta los sistemas más avanzados pueden verse saturados por la gran demanda de trabajo que exigen las nuevas aplicaciones.

Las actuales tendencias en cómputo apuntan hacia la cooperación entre sistemas en lugar de optar por la utilización de mejores equipos, pues resulta más redituable tener varios sistemas pequeños y económicos trabajando en conjunto y en paralelo que contar con un solo equipo centralizado de gran capacidad pero de elevado costo.

La cooperación entre sistemas tiene sus bases en las redes computacionales ya que son éstas las que se encargan de proveer el medio propicio para la intercomunicación de los diferentes equipos. En esta modalidad de trabajo se pretende que varias computadoras colaboren en la solución de un problema en común, realizando su labor en **paralelo** y coordinándose mediante la comunicación en red. Los anteriores sistemas constituyen una buena opción para brindar un mayor poder computacional debido a que son económicos, pueden construirse con equipo ya existente y no necesariamente moderno, se pueden expandir a gran escala y con facilidad, su obsolescencia es más lenta que la de los sistemas centralizados, y comúnmente cuando se dañan no hay que reemplazar todo el sistema sino sólo la parte averiada, además, si ésta no es suplida, el sistema puede seguirse utilizando aunque no tendrá la misma capacidad computacional.

Si bien estos sistemas parecen tener los mejores atributos, hay que tener en cuenta una cuestión: "la cooperación entre equipos, la distribución del trabajo, la aligación de resultados, la coordinación entre máquinas y el trabajo en paralelo, son elementos no inherentes a este tipo de sistemas".

Por esta razón, es conveniente tener siempre en mente que los sistemas cooperativos requieren de un software más especializado, complejo y robusto que haga que varios sistemas pequeños puedan ser vistos como un solo sistema de gran capacidad.

El principal objetivo de esta tesis es el de incursionar en el ámbito de esta clase de sistemas cooperativos a través de una serie de técnicas y paradigmas que nos permitan mostrar claramente la forma en que trabajan y, más precisamente, presentar una metodología que permita desarrollar aplicaciones eficientes que puedan implantarse de manera práctica en uno de estos sistemas. Para tal efecto, utilizaremos un sistema distribuido: **el cluster tipo Beowulf de la Facultad de Ingeniería**.

Comenzaremos esta tesis con una introducción a los sistemas distribuidos con el objeto de poder comprender qué es un cluster y específicamente qué es un cluster tipo Beowulf, que es en el que desarrollaremos algunas aplicaciones. Analizaremos también las características que posee el cluster de la Facultad de Ingeniería.

Antes de abordar las metodologías de programación paralela que nos permitirán desarrollar aplicaciones en el cluster, veremos algunos aspectos importantes relacionados con el paralelismo y que son básicos para poder entender dichas metodologías.

Existen distintas herramientas que permiten desarrollar aplicaciones en un cluster tipo Beowulf, en este trabajo de tesis decidimos utilizar la llamada MPI (Message-Passing Interface - Interfaz de Paso de Mensajes) pues es de las más poderosas y eficientes en su ramo, está en continuo desarrollo, cuenta con varias distribuciones con características especializadas según el sistema, todas las distribuciones siguen un estándar, y está bajo la licencia del software libre en afinidad con el sistema operativo del cluster, Linux. La distribución de MPI empleada en esta tesis es MPICH, esta distribución junto con LAM-MPI son las más utilizadas en clusters de tipo Beowulf.

Las técnicas de programación paralela expuestas en esta tesis están pensadas de manera general, no para casos particulares, con la intención de que sean, en la medida de lo posible, independientes de la plataforma empleada y puedan ser utilizadas con cualquier herramienta para la programación paralela y no sólo con la MPI.

Las aplicaciones que aquí presentamos están hechas con base en las técnicas y metodologías antes mencionadas y están implementadas con la ayuda de la MPI.

Esperamos que el presente trabajo de tesis represente una buena fuente que sirva de apoyo a todo aquel que esté interesado en la programación en paralelo sobre sistemas distribuidos.

1. Antecedentes.

1.1 El auge de los sistemas distribuidos.

Hoy en día ¿quién no ha usado un sistema distribuido? El teléfono, un cajero del banco o una operación en bolsa, la TV por cable, un viaje en avión o en metro, el correo electrónico o la visita de alguna página Web, todos ellos son sistemas distribuidos, pero ahora bien ¿qué es un sistema distribuido? Según Andrew Tanenbaum es un conjunto de computadores independientes que el usuario percibe como un solo computador.

El concepto de sistema distribuido surge a partir de la necesidad de tener redes dinámicas. Esto quiere decir que ahora las redes no se limitarán a ser sistemas diseñados para servir a usuarios geográficamente dispersos, sino que aunado a esto serán redes que permitirán el crecimiento o decrecimiento de un sistema según se requiera. De esta manera, el sistema será capaz de tolerar la adición de nuevos dispositivos o que sean retirados aquellos que ya no le sean necesarios. Lo importante de este concepto es que se lleve a cabo de una forma sencilla; es decir, (y aquí es donde entra el concepto básico de un sistema distribuido) que sea transparente y que se aplique desde el programador del sistema hasta el usuario final.

Por otro lado es fundamental tener presente que en un sistema distribuido las tareas se dividen entre los componentes que integran la red. La responsabilidad del procesamiento y las aplicaciones recaerá sobre las entidades que integran la red. Un sistema distribuido permite tener distintos elementos de procesamiento y muchos elementos de almacenamiento, todo conectado a través de una red.

1.1.1 Componentes de un sistema distribuido.

Los sistemas distribuidos tienen componentes tanto de hardware como de software y éstos aparecen listados en la tabla 1.1.

Componentes físicos	Componente lógico
Procesadores	Software
Enlaces de comunicación	
Relojes	
Almacenamiento	

Tabla 1.1. Componentes de un sistema distribuido en general.

Los componentes de software se encuentran en el sistema tal como se muestra en la figura 1.1.

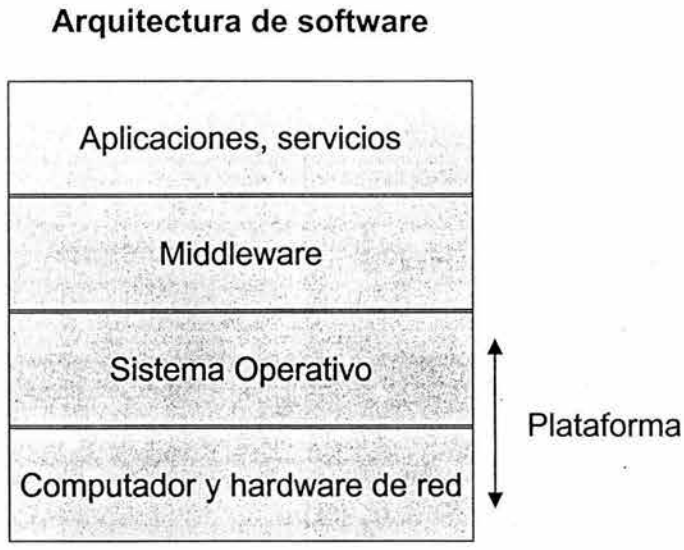


Figura 1.1. Disposición de los componentes de un sistema distribuido.

El término arquitectura de software se refería inicialmente a la estructuración del software como capas en un único computador. Recientemente las capas son uno o varios procesos, localizados en el mismo o diferentes computadores, que ofrecen y solicitan servicios.

En esta arquitectura las capas más bajas proporcionan servicio a las superiores y su implementación es dependiente de cada computador.

Según estos niveles es como se encuentran los componentes de software, en la capa más baja se encuentra el hardware de red y el computador, en la siguiente capa se encuentra el sistema operativo, posteriormente se encuentra el middleware, y las aplicaciones y servicios.

El middleware es el nivel encargado de enmascarar la heterogeneidad del sistema distribuido. Además proporciona un modelo de programación a los programadores de aplicaciones. Y soporta abstracciones como:

- Llamadas a procedimientos remotos o invocación remota.
- Comunicación entre grupos de procesos.
- Notificación de eventos.
- Replicación de datos.
- Servicios de nombres.

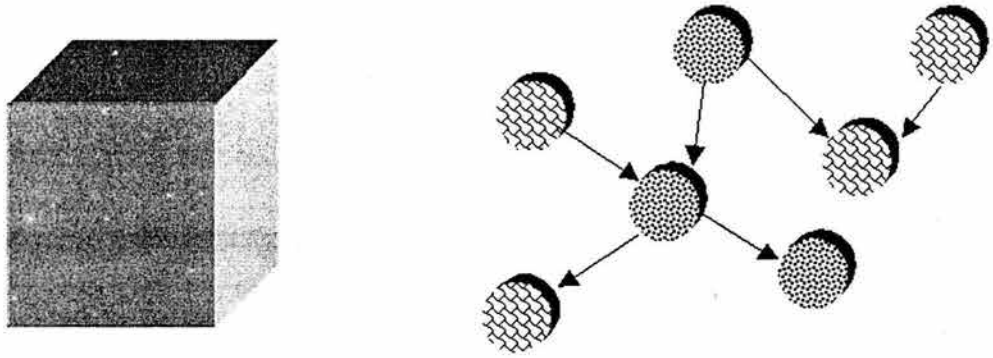
1.1.2 Tipos de sistemas distribuidos.

Existen dos tipos de sistemas distribuidos, los síncronos y los asíncronos, los asíncronos son aquellos cuya frecuencia de reloj no se encuentra sincronizada y por tal motivo el tiempo en que tarda en ser entregado un mensaje no se encuentra definido, en cambio en los sistemas síncronos los relojes se encuentran sincronizados y con ello se logra que el tiempo de entrega de un mensaje sea acotado, esto trae una ventaja sobre los sistemas asíncronos porque teniendo acotado el tiempo del mensaje puede usarse la técnica denominada *timeout*.

1.1.3 Ventajas y desventajas de los sistemas distribuidos sobre los centralizados.

Desde el punto de vista físico, un sistema centralizado es aquel cuya filosofía de instalación obliga a cablear todo hasta y desde un controlador principal o unidad central. Por ello podemos afirmar que un sistema centralizado siempre tiene una unidad central, pero debemos tener muy claro que un sistema con unidad central no es siempre centralizado, podemos tener un sistema con unidad central pero con módulos de entradas y saiidas.

Una manera de ilustrar cómo son los sistemas centralizados en comparación con los distribuidos se observa en la figura 1.2.



Centralizado (Monolítico)

Distribuido (Multicomponentes)

Figura 1.2. Representación esquemática de un sistema centralizado y de un sistema distribuido.

Generalmente, los sistemas distribuidos proporcionan mayores beneficios a los usuarios que los centralizados. Sus ventajas y desventajas se muestran en las tablas 1.2 y 1.3, respectivamente.

Ventajas	
Economía	Los microprocesadores ofrecen mejor proporción precio/desempeño que los *mainframe*
Velocidad	Un sistema distribuido puede tener mayor poder de cómputo que un *mainframe*
Distribución inherente	Algunas aplicaciones envuelven espacialmente máquinas separadas
Confiabilidad	Si una máquina falla, el sistema puede sobrevivir
Crecimiento incremental	Poder computacional puede ser añadido en pequeños incrementos

Tabla 1.2. Ventajas de un sistema distribuido en comparación con un sistema centralizado.

Desventajas	
Software	Más complejo
Red	La red puede congestionarse o causar otros problemas
Seguridad	Consideraciones sobre acceso a información privada

Tabla 1.3. Desventajas de un sistema distribuido en comparación con un sistema centralizado.

1.1.4 Desafíos de los sistemas distribuidos.

Debido al auge de los sistemas distribuidos, la tecnología siempre tiende a superar retos aún mayores, y hoy en día se quieren incrementar algunas características de los sistemas distribuidos para que éstos sean aún más eficientes, entre los desafíos que se desean alcanzar están:

- La **heterogeneidad** aplicada en los siguientes elementos:
  - Redes.
  - Hardware de computadores.
  - Sistemas operativos.
  - Lenguajes de programación.
  - Implementaciones de diferentes desarrolladores.

- **Código móvil o portable:** es el código que puede enviarse desde un computador a otro y ejecutarse en este último, en otras palabras, es un código que puede ser ejecutado en diferentes sistemas sin que necesite ser modificado. El concepto de máquina virtual ofrece un modo de crear código ejecutable sobre cualquier hardware.

- **Extensibilidad:** es la característica que determina si el sistema puede extenderse de varias maneras. Un sistema puede ser abierto o cerrado con respecto a extensiones de hardware o de software. Para lograr la extensibilidad es imprescindible que las interfaces clave sean publicadas.

  Los Sistemas Distribuidos Abiertos pueden extenderse a nivel de hardware mediante la inclusión de computadoras a la red y a nivel de software por la introducción de nuevos servicios y la reimplementación de los antiguos. Otro beneficio de los sistemas abiertos es su independencia de proveedores concretos.

- **Seguridad.**

  La seguridad tiene tres componentes:

  - **Confidencialidad**: protección contra individuos no autorizados.
  - **Integridad:** protección contra la alteración o corrupción.
  - **Disponibilidad:** protección contra la interferencia que impide el acceso a los recursos.

  Existen dos características que no han sido resueltas en su totalidad:

  - Ataques de denegación de servicio.
  - Seguridad del código móvil.

- **Escalabilidad.** Se dice que un sistema es escalable si conserva su efectividad cuando ocurre un incremento significativo en el número de recursos y en el número de usuarios.

  El diseño de sistema distribuido escalable presenta los siguientes retos:

  - **Controlar la degradación del rendimiento:** por ejemplo los algoritmos que emplean estructuras jerárquicas se comportan mejor frente al crecimiento de la escala, que los algoritmos que emplean estructuras lineales.

  - **Evitar cuellos de botella:** los algoritmos deberían ser descentralizados.

  - **Prevenir el desbordamiento de los recursos de software.**

- **Detección de fallos.**

  Se pueden utilizar sumas de comprobación (checksums) para detectar datos corruptos en un mensaje. Al emplear el enmascaramiento de fallos, los mensajes pueden retransmitirse, o utilizar la réplica de los datos.

- **Tolerancia de fallos.**

  Los programas clientes de los servicios pueden diseñarse para tolerar ciertos fallos. Esto implica que los usuarios tendrán también que tolerarlos.

- **Recuperación de fallos:** implica el diseño de software en el que, tras una caída del servidor, el estado de los datos puede reponerse o retractarse (rollback) a una situación anterior.

- **Redundancia:** emplear componentes redundantes. Tener componentes adicionales para un mismo recurso garantiza la disponibilidad del mismo y minimiza las pérdidas en caso de fallo. Existe la posibilidad de acceso concurrente a un mismo recurso. La concurrencia en los servidores se puede lograr a través de threads (hilos). Cada objeto que represente un recurso compartido debe responsabilizarse de garantizar que opera correctamente en un entorno concurrente. Para que un objeto sea seguro en un entorno concurrente, sus operaciones deben sincronizarse de forma que sus datos permanezcan consistentes.

- **Transparencia.**

  - **Transparencia de acceso:** permite acceder a los recursos locales y remotos empleando operaciones idénticas.

  - **Transparencia de ubicación:** permite acceder a los recursos sin conocer su localización.

  - **Transparencia de concurrencia:** permite que varios procesos operen concurrentemente sobre recursos compartidos sin interferencia mutua.

  - **Transparencia de replicación:** permite replicar los recursos sin que los usuarios y los programadores necesiten su conocimiento.

  - **Transparencia frente a fallos:** permite ocultar fallos.

  - **Transparencia de movilidad:** permite la reubicación de recursos y clientes en un sistema sin afectar la operación de los usuarios y los programas.

1.2 Equipos para el cómputo paralelo.

1.2.1 Clasificación de Flynn.

En 1966 Flynn propuso el siguiente modelo de clasificación basado en la ejecución de instrucciones paralelas y el flujo de datos en los procesadores.

1.2.1.1 SISD.

Single Instruction-Single Data. Flujo único de Instrucciones-Flujo único de Datos. Es el procesador convencional.

- Las instrucciones se ejecutan secuencialmente, pero pueden estar traslapadas en las etapas de ejecución.
- Puede tener más de una unidad funcional bajo supervisión de una sola Unidad de Control.
- Es la organización de la mayoría de los computadores serie actuales.

En la figura 1.3 se muestra la arquitectura SISD.

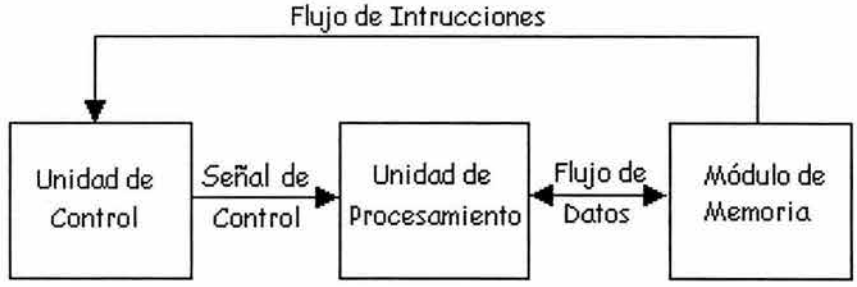


Figura 1.3. Organización de la arquitectura SISD.

1.2.1.2 SIMD.

Single Instruction-Multiple Data. Flujo único de Instrucciones-Múltiples flujos de Datos. Las computadoras SIMD poseen una estructura semejante a los anteriores, pero operan sobre vectores de datos y presentan la ventaja de que todas las unidades de ejecución paralela están sincronizadas y todas responden a una sola instrucción que emana de un único contador de programa (PC).

Estos procesadores se muestran muy eficientes a la hora de tratar con vectores en ciclos **for** y en situaciones de paralelismo de datos (masivos), sin embargo su rendimiento decrece en las sentencias de selección, al tener que ejecutar diferentes caminos para distintas opciones.

- Múltiples unidades de ejecución o elementos de proceso (EP) supervisados por una única Unidad de Control.
- Todos los EP reciben la misma instrucción emitida por la UC (Unidad de Control) pero operan sobre diferentes conjuntos de datos procedentes de flujos distintos.
- Esta organización corresponde a los procesadores matriciales.

En la figura 1.4 a continuación se muestra la arquitectura SIMD.

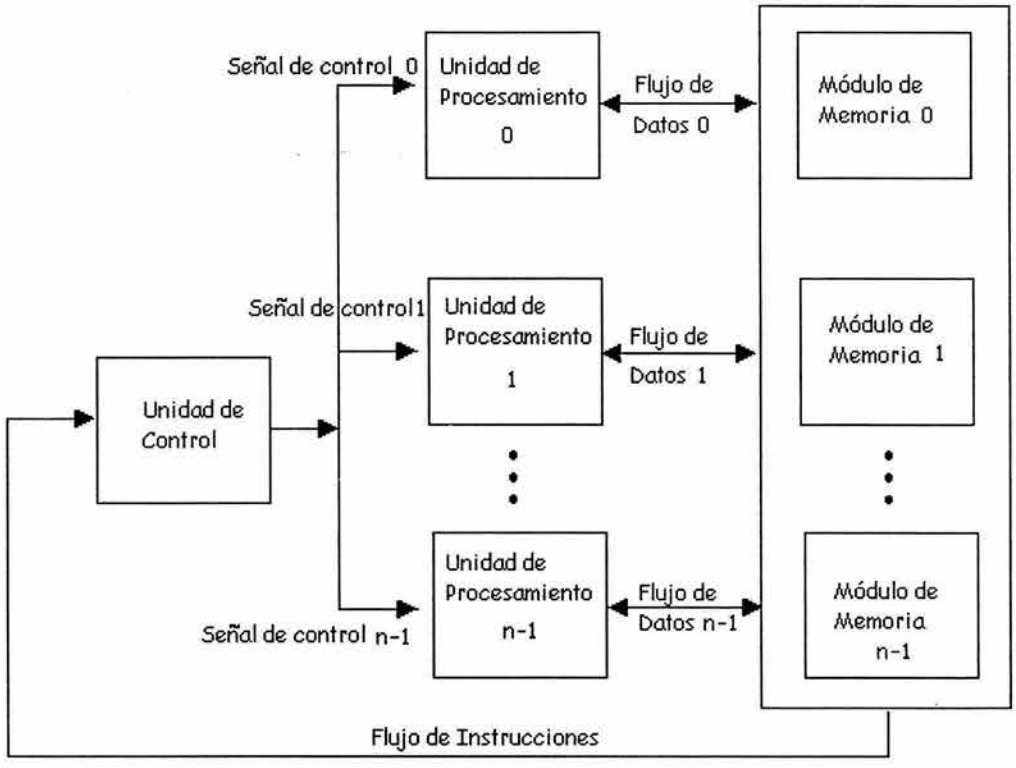


Figura 1.4. Organización de la arquitectura SIMD.

1.2.1.3 MISD.

Multiple Instructions-Single Data. Múltiple flujo de Instrucciones-Flujo único de Datos.

- Existen **n** unidades procesadoras.
- Cada una recibe distintas instrucciones que operan sobre el mismo flujo de datos y sus derivados. Los resultados (la salida) de un procesador pasan a ser la entrada (los operandos) del siguiente procesador.
- Esta clasificación no es un sistema viable para la computación.

En la figura 1.5 se especifica el esquema de la arquitectura MISD.

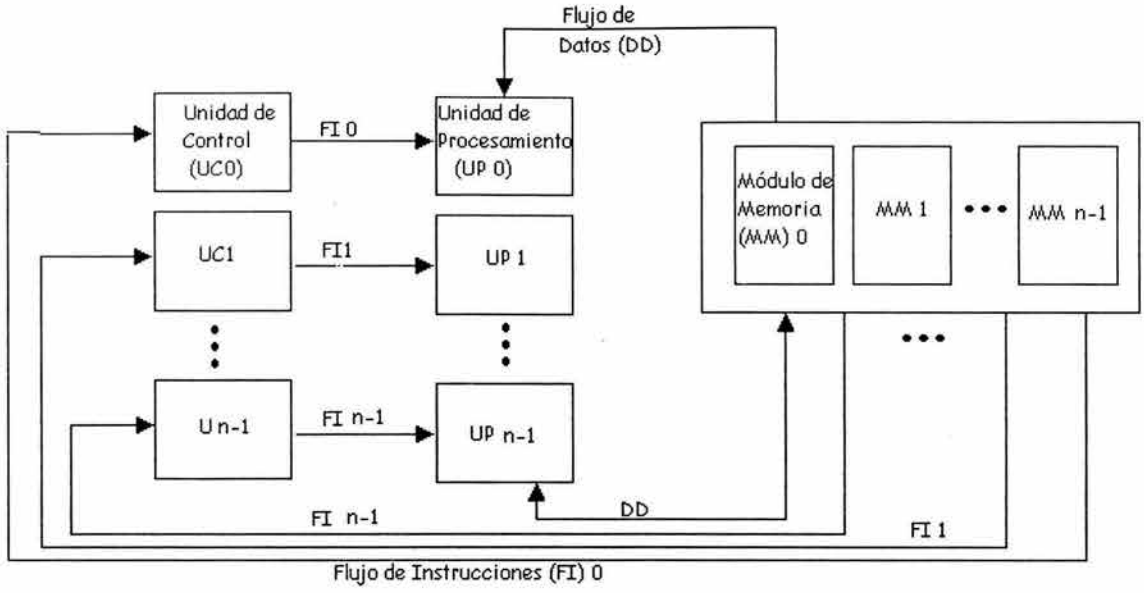


Figura 1.5. Organización de la arquitectura MISD.

1.2.1.4 MIMD.

Multiple Instructions-Multiple Data. Múltiple flujo de Instrucciones-Múltiple flujo de Datos. Se basan en la idea de crear potentes computadores conectando otros más pequeños que cooperan entre sí frente al costo de crear un supercomputador SISD. Presentan, además de la ventaja del costo, un gran soporte ante fallos (si disponemos de **n** procesadores siempre podemos continuar la unidad de procesamiento con **n-1**), lo cual es una garantía en máquinas a las que se es exige un servicio continuo.

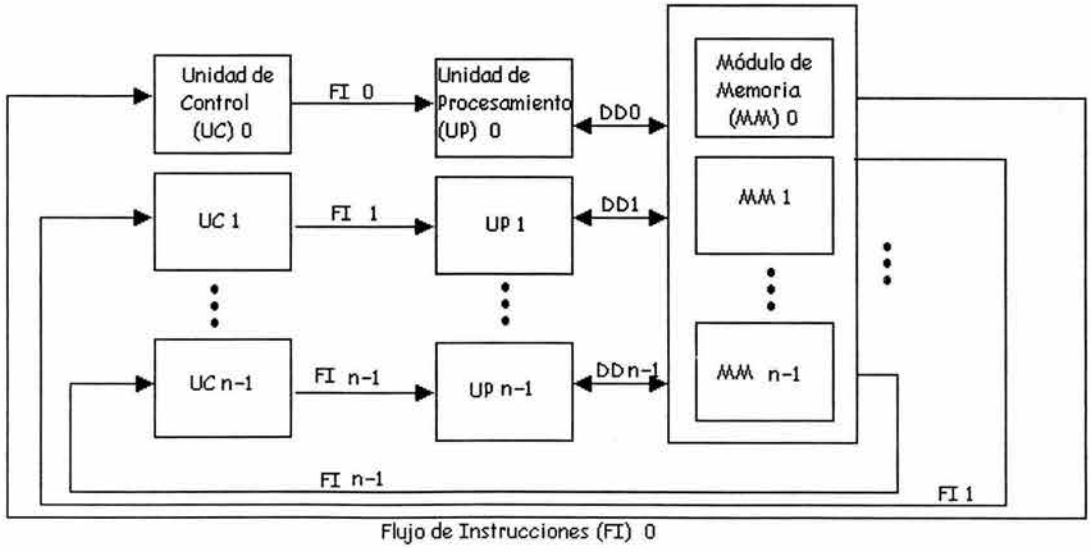


Figura 1.6. Organización de la arquitectura MIMD.

- Ésta incluye a sistemas con más de un procesador capaz de ejecutar varios programas Multiprocesador.
- Multiprocesadores.
- Multicomputadoras.

La figura 1.6 muestra el diseño de la arquitectura MIMD.

1.2.2 Arquitecturas paralelas.

Algunas de las arquitecturas paralelas que no se encuentran incluidas en la clasificación de Flynn se encuentran agrupadas en las siguientes categorías:

- **SMP** (Symmetric Multiprocessors): este término incluye solamente a los procesadores simétricos los cuales se encuentran conectados a la memoria compartida de manera uniforme. Los procesadores son aquellos que tienen igual oportunidad de acceso a los recursos de memoria compartida. Es una arquitectura de computadores que provee un rápido desempeño haciendo que múltiples CPU estén disponibles para completar procesos individuales simultáneamente (multiprocesamiento).

- **MPP** (Massively Parallel Processor) es una arquitectura de computadores que provee un rápido desempeño haciendo que múltiples CPU estén disponibles para completar procesos individuales simultáneamente (multiprocesamiento).

- **La arquitectura ccNUMA** (Cache Coherent Nonuniform Memory Access) es una extensión de SMP. Diseñada para superar los cuellos de botella inherentes de SMP, la arquitectura ccNUMA deja a los proveedores construir servidores en gran escala. ccNUMA ofrece todos los mejores beneficios de SMP y MPP, sin ninguna de sus desventajas.

  Ventajas Arquitecturales de ccNUMA

  - **Modelo de memoria compartida.** Como los sistemas SMP, presenta un solo, global y unificado modelo de memoria.

  - **Multi-CPU, Multiprocesamiento.** Como SMP y MPP, soporta multi-CPU y multiprocesamiento en configuraciones de cierto número de CPU.

  - **Distribución de la carga.** Como MPP, distribuye E/S y accesos a memoria por múltiples subsistemas, pero difiere de MPP en que la carga se balancea automáticamente.

- **Operaciones de E/S concurrentes.** Como MPP, soporta múltiples operaciones concurrentes al disco usando independientes, pero totalmente conectados, subsistemas de E/S.

- **Sistemas Distribuidos:** se refieren a redes de computadoras independientes, son combinaciones de SMP, MPP, clusters y computadoras individuales.

- **Clusters:** son agrupaciones de PC unidos por una red de alta velocidad. Es un conjunto de computadoras las cuales trabajan en conjunto para resolver una tarea. Un cluster está conformado por varias computadoras las cuales se comunican por medio de una conexión a red trabajando en un proyecto el cual seria muy largo para una sola computadora, resolviéndolo en un tiempo razonable.

  Existen varios tipos de clusters uno de los más comunes es el cluster computacional: éste generalmente es utilizado procesar grandes cantidades de datos. Otro tipo de cluster es el de base de datos ilamado libre de falla o alta disponibilidad se ocupa para almacenar bases de datos las cuales deben estar siempre en funcionamiento.

1.2.3 Clasificación de los clusters.

Un cluster es un tipo de sistema paralelo de procesamiento distribuido, el cual consiste en una colección de computadoras interconectadas para trabajar en conjunto como una sola, integrando sus recursos de cómputo.

Los clusters son clasificados en diferentes categorías basadas en diferentes factores:

1. **Propósito de la aplicación:** Son utilizados en las ciencias computacionales o aplicaciones de misión crítica.

   - Clusters de alto desempeño (High Performance(HP)).
   - Clusters de alta disponibilidad (High Availability (HA)).

2. **Propiedad de los nodos:** Los nodos pertenecen a un cluster individual o a uno dedicado.

   - Cluster dedicado.
   - Cluster no dedicado.

   La distinción entre estos dos tipos de clusters está basada en la propiedad que se tiene de los nodos en el cluster. En el caso de los cluster 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 cluster completo. La alternativa no dedicada es cuando hay individuos que son propietarios de las estaciones de trabajo y las aplicaciones son ejecutadas en ciclos libres de CPU.

3. **Hardware de los nodos:** PC, estaciones de trabajo, o SMP.

   - Cluster de PC (CoPC) o pilas de PCs (Pops).
   - Cluster de estaciones de trabajo (COWs).
   - Cluster de SMPs (CLUMPs).

4. **Sistema operativo del nodo:** Linux, NT, Solaris, AIX, etc.

   - Cluster con base en Linux
   - Cluster con base en Solaris.
   - Cluster con base en NT.
   - Cluster con base en AIX.
   - Cluster digitales VMS.
   - Clusters con base en HP-UX.
   - Cluster Microsoft Wolfpack.

5. **Configuración del nodo:** Arquitectura del nodo y tipo de sistema operativo que tiene.

   - Clusters homogéneos: todos los nodos tienen una arquitectura similar y corren bajo el mismo sistema operativo.
   - Cluster heterogéneos: todos los nodos tienen diferentes arquitecturas y corren bajo diferentes sistemas operativos.

6. **Niveles de agrupamiento (clustering):** con base en la localización de los nodos y su número.

   - Grupo de clusters (de 2 a 9 nodos): los nodos son conectados por SANs (System Area Network).
   - Clusters departamentales (de 10 a 100 nodos).
   - Clusters organizacionales (de más de 100 nodos).
   - Metacomputadoras nacionales (WAN/basadas en Internet).
   - Metacomputadoras internacionales de 1000 nodos a muchos millones.

1.3 Los clusters tipo Beowulf.

Un cluster tipo Beowulf es una colección de computadoras personales o de estaciones de trabajo interconectadas por alguna topología de red, constituido principalmente por componentes comerciales de hardware, y que utiliza un sistema operativo de libre distribución como Linux o FreeBSD. Los nodos que lo conforman están totalmente dedicados a las tareas asignadas al cluster y utilizan

una red privada para comunicarse. Generalmente, el cluster se conecta al mundo exterior a través de un solo nodo –el servidor–. La programación en un Beowulf comúnmente es realizada por medio del paso de mensajes utilizando lenguajes como C y FORTRAN.

El primer cluster de este tipo fue desarrollado por Donald Becker y Thomas Sterling en 1994, desde entonces, esta solución popular ha sido ampliamente aceptada en varios ambientes de producción, principalmente en laboratorios de investigación y en sitios académicos.

De acuerdo a la clasificación establecida en el tema 1.2.3, los clusters tipo Beowulf poseen las siguientes características:

1. **Propósito de la aplicación:**
   - ➢ Clusters de alto desempeño (High Performance(HP)).
2. **Propiedad de los nodos:**
   - ➢ Cluster dedicado.
3. **Hardware de los nodos:**
   - ➢ Cluster de PC (CoPC) o pilas de PCs (Pops) y de estacioes de trabajo (COWs).
4. **Sistema operativo del nodo:**
   - ➢ Cluster con base en Linux o en algún otro sistema operativo de libre distribución.
5. **Configuración del nodo:**
   - ➢ Clusters heterogéneos u homogéneos.
6. **Niveles de agrupamiento (clustering):**
   - ➢ Clusters departamentales (de 10 a 100 nodos).

1.3.1 Ventajas y desventajas de los clusters tipo Beowulf.

Existen diversas ventajas y desventajas en el empleo de un cluster tipo Beowulf, cabe mencionar que éstas están íntimamente relacionadas con el tipo de aplicación en la que se desea emplear el cluster, a pesar de ello, de manera general podemos mencionar que las que consideramos más relevantes son las que listamos a continuación.

Ventajas de un cluster tipo Beowulf.

- Una de las ventajas obvias de un cluster tipo Beowulf es su economía, ya que está constituido por computadoras personales, componentes comerciales y además utiliza un sistema operativo de código libre.

- Como bien sabemos la construcción de un supercomputador de varios procesadores, es compleja y costosa. Sin embargo, crear una red con 10, 100 o incluso más nodos es más económico y fácil de realizar sobre todo por el hecho de trabajar con componentes que están al alcance de cualquier usuario.

- Debido a que cada uno de sus nodos es una máquina completa, podemos usarlos individualmente o sincronizarlos de tal forma de hacerlos ver ante el usuario como un único sistema de cómputo de mayor capacidad de procesamiento.

- Un Beowulf, por lo general, tiene nodos cliente que resultan muy económicos ya que sólo poseen lo mínimo para funcionar: no tienen tarjeta de video, ni monitor, mucho menos teclado, ratón o periféricos. En ocasiones los clientes no cuentan siquiera con disco duro.

- La principal ventaja de tener una arquitectura tipo Beowulf sin discos duros es la flexibilidad que se adquiere al agregar nuevos nodos. Debido a que el cliente no contiene información local, al agregar el nuevo nodo, sólo se deben modificar unos pocos archivos en el nodo servidor y correr un guión de comandos (*script*) que dará de alta al nuevo cliente.

- Otra gran ventaja de la arquitectura sin discos es el hecho de que no hay que instalar un sistema operativo o algún otro software en los clientes ya que el servidor es el encargado de proveer todos los archivos a los clientes. En esta configuración, los nodos son accedidos a través de conexiones remotas desde el servidor.

- Las herramientas de desarrollo para las computadoras comerciales que se utilizan como nodos en un Beowulf están más "maduras" en comparación con las herramientas de desarrollo para ordenadores paralelos, esto se debe principalmente a la naturaleza no estándar de los distintos sistemas paralelos.

- Los clusters tipo Beowulf son fácilmente escalables. La capacidad de los nodos se puede ampliar de forma sencilla, añadiendo más memoria o procesadores adicionales.

- El ancho de banda de las redes LAN se ha incrementado notablemente del mismo modo que el tiempo de latencia ha disminuido. Con las nuevas

implementaciones y protocolos de las Redes de Área Local, es posible tener una rápida interconexión entre computadoras y a bajo precio.

- Los clusters tipo Beowulf son más fáciles de integrar en redes existentes que los ordenadores paralelos específicos.

- Se pueden utilizar diversas topologías de red para la conexión de los nodos de un Beowulf: anillo, estrella, malla, anillo-estrella, hipercubo-estrella, etc.

- No es necesario que todos los nodos tengan la misma capacidad de procesamiento, es decir, es posible conformar un Beowulf con máquinas de diferentes capacidades de cómputo.

- 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, esto es importante tenerlo en cuenta, ya que es infinitamente más sencillo encontrar un nuevo procesador Pentium que cualquiera de los componentes de un complejo supercomputador Fujitsu, por ejemplo.

  Además, las computadoras personales están incrementando su potencial de procesamiento. Su capacidad se ha incrementado drásticamente en los últimos años y todo indica a que esta tendencia continuará así.

- Los Beowulf no requieren de excesivas medidas de seguridad cuando son conectados en una red. La política de seguridad para los clusters tipo Beowulf debe ser tal que todos los nodos dentro del cluster deben tener plena confianza entre ellos. La razón por la que los Beowulf resultan ser tan seguros es porque ninguno de los clientes está directamente conectado al mundo exterior y todos los nodos son prácticamente iguales.

  Asimismo, el servidor puede confiar casi por completo en los nodos clientes ya que es prácticamente imposible para alguien acceder a los nodos clientes sin estar en la consola o haciéndolo por medio del nodo servidor primero. Las principales ventajas de tener una buena seguridad dentro del cluster son la flexibilidad, y la facilidad de uso y administración.

- Un cluster tipo Beowulf puede ser tan inteligente como para poder notar si uno de sus nodos clientes no está disponible o ha dejado de funcionar correctamente y así continuar son su labor empleando sólo a los nodos disponibles.

- Pueden modificarse más fácilmente que los sistemas SMP (Symmetric Multiprocessors). Estos sistemas se vuelven obsoletos rápidamente, mientras que un Beowulf es para siempre, las partes que lo forman evolucionan por sí solas y es posible agregar nuevos componentes cuando se requiera.

Así, los nuevos dispositivos pueden ser incorporados en un Beowulf antes que en un sistema MPP (Massively Parallel Processor). En contraste, tan pronto como un nuevo procesador es colocado en la tarjeta madre de una computadora personal, éste está listo para ser instalado en un Beowulf 24 horas después.

- Cada vez encontramos más software diseñado para aumentar el rendimiento de nuestro Beowulf, así como para facilitar su programación. También, día a día, aparecen nuevas redes diseñadas específicamente para este tipo de arquitectura, asimismo el hardware de red de interconexión ha venido experimentando un constante decremento de precio.

Desventajas de un cluster tipo Beowulf.

- Es un poco complicada su instalación y su administración, los clusters que son armados a partir de componentes comerciales no son fáciles de configurar y manejar.

- Otra limitante consiste en que las aplicaciones que pueden tener verdadera ventaja de la capacidad de procesamiento del cluster son aquellas que tienen naturaleza predominantemente paralela.

  Para obtener auténticas ventajas de un Beowulf, las tareas y aplicaciones deben ser repartidas de alguna forma para poder distribuirlas en los múltiples procesadores. Esta tarea no es nada trivial, el programador es quien tiene que decidir qué partes de un proceso deben correr en diferentes nodos clientes y qué partes no, en qué momento deben comunicarse los procesadores y cómo deben reunirse los resultados individualmente obtenidos para conseguir un resultado global.

- Como cada procesador tiene su propio reloj, la excesiva comunicación entre procesadores hace que la ejecución del cluster se vea disminuida de manera general ya que no resulta fácil la sincronización entre procesadores por lo que se debe evitar al máximo la comunicación entre ellos.

- Las redes de computadoras, no están especializadas para el procesamiento en paralelo (al menos en un principio) por lo que a veces no se tienen los resultados esperados al añadir más y más nodos al cluster debido a que existe un *overhead* muy grande a medida que más máquinas se comunican entre sí, por lo que en muchas ocasiones, al agregar más procesadores, en lugar de mejorar el desempeño del Beowulf se disminuye.

- La posible diferencia de potencia entre las distintas máquinas que constituyen el Beowulf plantea un problema: la necesidad de equilibrar la carga de trabajo entre los distintos nodos, ya que si la distribución de carga fuese equitativa, los nodos de mayor potencia finalizarían antes su tarea y

tendrían que esperar a los nodos de menor potencia, disminuyendo bastante la eficiencia que tiene el cluster.

Una aproximación de balanceo de carga es realizada por los usuarios a la hora de asignar los diferentes procesos de un trabajo paralelo a cada nodo, habiendo hecho una revisión previa de forma manual del estado de dichos nodos.

PVM y MPI, por ejemplo, disponen de herramientas para la asignación inicial de procesos a cada nodo, sin considerar la carga existente en los mismos ni la disponibilidad de la memoria libre de cada uno. Estos paquetes corren en el ámbito de usuario como aplicaciones ordinarias, es decir, son incapaces de activar otros recursos o de distribuir la carga de trabajo en el cluster dinámicamente. La mayoría de las veces es el propio usuario el responsable del manejo de los recursos en los nodos y de la ejecución manual de la distribución o migración de programas. Sin embargo, afortunadamente existen programas que realizan el equilibrado de carga de forma dinámica y sin que el usuario tenga que preocuparse de ello.

1.4 El cluster tipo Beowulf de la Facultad de Ingeniería.

La máquina de procesamiento distribuido creada en la Facultad está constituida de manera general por un nodo central o servidor, una serie de nodos, y el hardware y software necesarios para la interconexión y configuración de cada uno de estos elementos, los cuales se explicarán más adelante.

Una característica importante de los nodos, es que ninguno posee teclado, ni monitor, ni ratón, debido a que en este tipo de máquinas es muy conveniente que los nodos sean lo más mudos posible, esto es, mientras menos puertos de entrada estén generando procesos particulares a cada nodo, la capacidad de procesamiento distribuido es más grande, ya que mayor parte del procesamiento esté enfocada a la resolución del problema. El único nodo que tiene conectado los periféricos mencionados es el servidor.

El cluster Beowulf de la Facultad de Ingeniería ofrece las siguientes características:

- Expansibilidad y escalabilidad.
- Alta disponibilidad.
- Componentes económicos.
- Una comunicación dedicada.

La tecnología y componentes con los que está diseñado el cluster permiten incrementar su poder de procesamiento mediante el uso de tecnologías estándar, y gracias a esto se puede lograr que en el momento en que se disponga de mejor

hardware se puedan hacer los ajustes al equipo para incrementar su poder de procesamiento de una manera relativamente sencilla.

A los componentes con los que está diseñado el cluster de la Facultad de Ingeniería de la UNAM se les denomina componentes *commodity*, debido a que son comerciales, fáciles de adquirir y poseen un costo considerablemente bajo.

La clave para integrar un cluster con componentes *commodity* es tener:

- Un buen desempeño en los nodos.
- Una interconexión de red dedicada para proveer comunicación de datos entre los nodos, logrando un procesamiento eficiente.

La descripción de cada una de las partes constitutivas del cluster es la siguiente:

El servidor.

Su nombre es scluster, cuenta con un procesador AMD Athlon (K7) a 850 MHz, memoria RAM de 128MB, un disco duro de 29.89 GB, una unidad de CD-ROM, una unidad de disco de 3 ½ , un monitor SVGA, teclado, ratón, 2 tarjetas de red 3Com 3c509 (una para la transferencia de información interna y otra para poder conectarse a una red externa) y finalmente una tarjeta de video.

Los nodos.

Son 14 nodos en total, sus nombres son nodo10, nodo11, ..., nodo22 y nodo23. Éstos tienen en común las siguientes características: 16 MB de memoria RAM, una unidad de disco de 3 ½ " indispensable para el arranque –los nodos no poseen disco duro y tienen que inicializarse por medio de discos flexibles por lo que son conocidos como estaciones *diskless*–, una tarjeta de red 3Com 3c509 y una tarjeta de video. En cuanto a CPU, los nodos no son homogéneos, el nodo10 posee un Intel Pentium a 100 MHz, del nodo11 al nodo15 tienen un procesador Intel 80486 a 66 MHz, del nodo16 al nodo22 tienen un Intel 80486 a 50 MHz y el nodo23 cuenta con un Intel 80486 a 33 MHz.

La red.

Para la conexión de cada uno de los nodos con el servidor se maneja una red LAN con tecnología Ethernet a 10 Mbps y un esquema de direccionamiento IP Clase C de tipo privado que tiene el identificador de red 192.168.10.0 y máscara 255.255.255.0. A continuación mostramos el archivo /etc/hosts para puntualizar las direcciones IP de los nodos y del servidor.

```
127.0.0.1 localhost.localdomain    localhost
#servidor
192.168.10.1    scluster
#nodos
192.168.10.10  nodo10
192.168.10.11  nodo11
192.168.10.12  nodo12
192.168.10.13  nodo13
192.168.10.14  nodo14
192.168.10.15  nodo15
192.168.10.16  nodo16
192.168.10.17  nodo17
192.168.10.18  nodo18
192.168.10.19  nodo19
192.168.10.20  nodo20
192.168.10.21  nodo21
192.168.10.22  nodo22
192.168.10.23  nodo23
```

Para interconectar los equipos se utilizan dos concentradores (*hubs*) en cascada unidos mediante una conexión cruzada (*crossover*) y toda la red está cableada con par trenzado UTP categoría 3 con conectores RJ-45.

El software.

En cuanto a software se refiere, se tiene como base para la configuración y administración del cluster al sistema operativo Linux con una distribución Red Hat versión 6.2. Este sistema se carga en el servidor y desde ahí, cada nodo crea una imagen kernel del mismo para poder arrancar. Para esto se han creado previamente discos de arranque mediante los paquetes Etherboot y Mini-Netboot.

La programación en paralelo en el cluster es realizada con base en la PVM (Parallel Virtual Machine, Máquina Virtual Paralela) y la MPI (Message-Passing Interface, Interfaz de Paso de Mensajes) empleando la distribución MPICH.

La información se comparte entre los nodos a través del sistema de archivos NFS (Network File System, Sistema de archivos de Red), que permite montar sistemas de archivos a través de una red y que es completamente transparente para los usuarios. Además, el NFS también se usa para facilitar la distribución de los programas que se van a ejecutar.

Esquema del cluster.

A continuación, en la figura 1.7, se muestra la forma en que se encuentra conectado el cluster y el direccionamiento que utiliza.

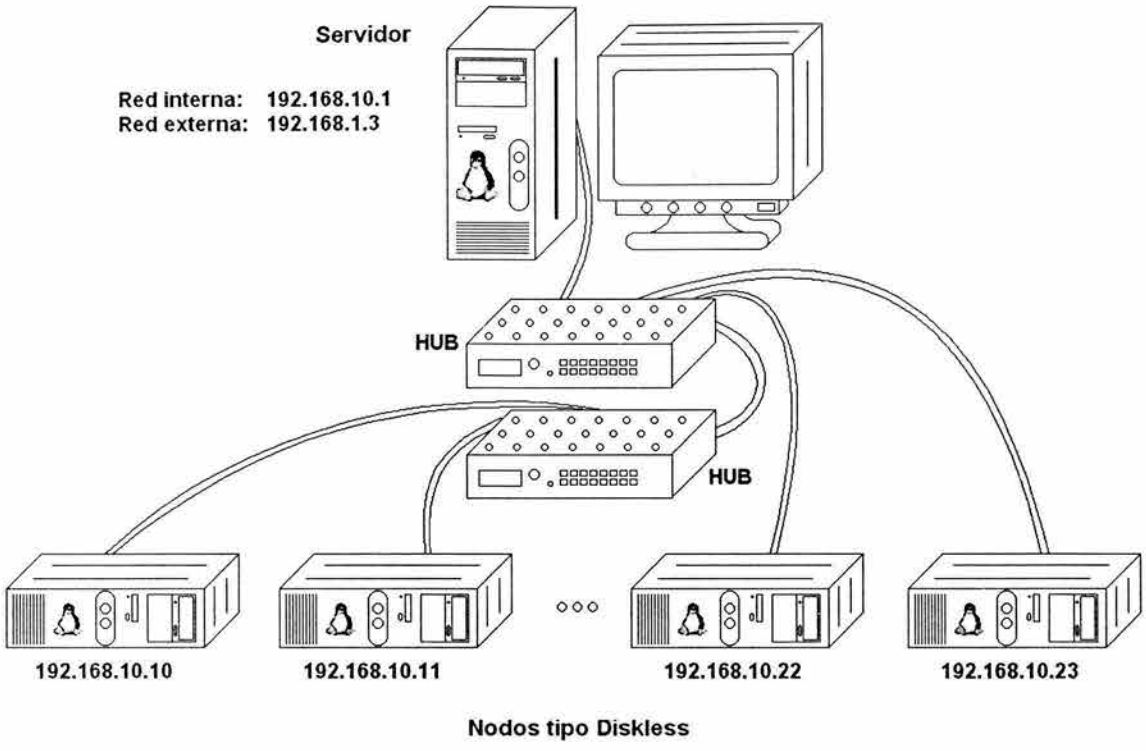


Figura 1.7. Esquema de la interconexión del cluster tipo Beowulf de la Facultad de Ingeniería.

2. La programación distribuida y paralela.

2.1 Niveles de paralelismo.

En la actualidad, la mayoría de las computadoras, a pesar de que sólo cuenten con un procesador, implementan algún tipo de paralelismo. La idea de la inclusión del paralelismo en los sistemas computacionales surge con el afán de poder incrementar la velocidad de procesamiento de éstos.

El paralelismo se clasifica en diversos niveles atendiendo principalmente al número de instrucciones que son ejecutadas de forma simultánea. Los diversos niveles en los que se divide el paralelismo serán expuestos a continuación.

Paralelismo a nivel de tarea o trabajo (*Job-level parallelism*) .

Este tipo de paralelismo se presenta cuando diferentes tareas (programas enteros) están corriendo al mismo tiempo pero en diferentes procesadores.

Características:

- Es el más alto nivel de paralelismo que se tiene.
- No existe gran interacción entre procesadores ya que, por lo general, las tareas son independientes entre sí.

Entre mayor número de procesadores se tengan es posible tener más trabajos siendo procesados al mismo tiempo.

Paralelismo a nivel de programa (*Program-level parallelism*) .

Consiste en dividir un programa en sus partes constitutivas y procesar cada una de éstas en diferente procesador, por lo que, partes del mismo programa estarán siendo ejecutadas en paralelo en los diversos procesadores del sistema.

Características:

- Existe más comunicación entre los procesadores que en el paralelismo a nivel de tarea.
- Se busca dividir al programa en partes que no sean tan dependientes para minimizar la interacción entre los procesadores y así agilizar la ejecución del programa.
- En ocasiones no es posible fraccionar un programa tan fácilmente ya que algunas de sus partes dependen del resultado previamente obtenido por la ejecución de otras.

- La partición de los programas requiere de las técnicas y los algoritmos de la programación concurrente y paralela.

Paralelismo a nivel de instrucción (Instruction-*level parallelism*) .

Dentro de un mismo proceso o tarea se pueden ejecutar instrucciones independientes en forma simultánea. Esto significa que múltiples instrucciones están siendo ejecutadas por diferentes procesadores o incluso por uno solo aunque, en este caso, no de forma paralela sino que el procesador da un determinado tiempo de procesamiento a cada una de ellas. Lo anterior es posible debido a que los procesadores son más rápidos que los dispositivos periféricos, por lo que pueden estar atendiendo a uno de ellos mientras esperan la respuesta de los otros.

Existen dos tipos de paralelismo a nivel de instrucción:

1. Las instrucciones independientes se ejecutan al mismo tiempo en diferentes procesadores.

2. Una determinada instrucción se descompone en subinstrucciones y éstas son ejecutadas en paralelo.

Características:

- La dependencia entre operaciones debe ser verificada.
- Las instrucciones que son independientes de las otras, pueden ser calendarizadas para que sean ejecutadas en un determinado tiempo de forma simultánea.

Paralelismo a nivel de bit (Bit-*level parallelism*) .

Éste es el paralelismo de más bajo nivel y es invisible para el usuario ya que se logra a través del hardware.

El paralelismo a nivel de bit concierne a los diseñadores de los microprocesadores y en especial a los que diseñan su Unidad Aritmético-Lógica ya que ellos pueden lograr que las operaciones a nivel de bits se realicen o no en paralelo, por ejemplo, al efectuar una suma, se puede optar por predecir los acarreos de forma simultánea al cálculo de la suma o simplemente se pueden ir acarreando los bits conforme se realiza ésta.

Características:

- El usuario no puede controlar este tipo de paralelismo ya que todo se maneja por hardware.
- Debido a que los procesadores que implementan este tipo de paralelismo requieren de mayor tecnología en su elaboración, tiene precios más

elevados que los que no utilizan paralelismo a nivel de bit, pero a pesar de ello, su desempeño práctico es mucho mejor.

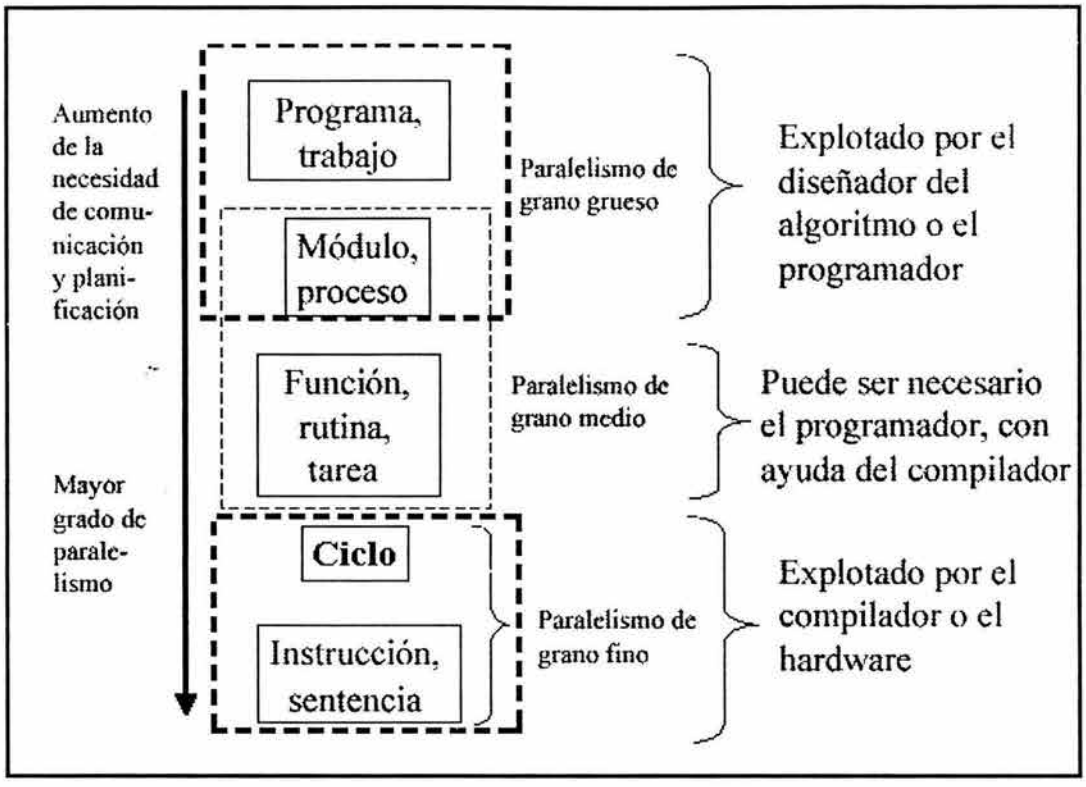


Figura 2.1. Niveles de paralelismo y su relación con los grados de granularidad.

Granularidad.

Se le conoce como granularidad al tamaño relativo de las unidades de cómputo (programas, subprogramas, funciones, subrutinas, ciclos, instrucciones o bytes) que son ejecutadas al mismo tiempo, es decir, de forma paralela. En otras palabras, la granularidad se mide de acuerdo al grosor o la fineza con que se dividen las tareas en un sistema de cómputo.

Se distinguen los siguientes grados de granularidad:

Paralelismo de Grano Fino: El paralelismo de grano fino representa el más complejo nivel de paralelismo. En este grado de paralelismo, el tamaño del grano es fino ya que se trabaja a nivel de ciclos e instrucciones.

Paralelismo de Grano Medio: Aquí, una aplicación es dividida en una colección de tareas, funciones, o rutinas que se ejecutan de forma paralela en diferentes procesadores. Generalmente se necesitará un alto grado de

coordinación e interacción entre las partes constitutivas de la aplicación que está siendo ejecutada.

Paralelismo de Grano Grueso: Esta clase de granularidad es fácilmente entendible como un grupo de procesos o aplicaciones completamente independientes ejecutándose al mismo tiempo por diferentes procesadores. En esta clase de granularidad no existe comunicación entre las diferentes aplicaciones.

En la figura 2.1 se muestran los niveles de paralelismo y la relación que tienen con los grados de granularidad.

2.2 Detección del paralelismo para la solución de problemas.

No todos los problemas son aptos para resolverse mediante programación distribuida o paralela, y en aquellos problemas que puedan ser atacados con estas técnicas se deben tener en cuenta los siguientes aspectos: la granularidad de los flujos en que se dividirá la aplicación paralela, el tipo de sincronización que existirá entre los procesos, los mecanismos de comunicación que se utilizarán y la arquitectura sobre la que se deberá de trabajar.

2.2.1 Diseño de algoritmos paralelos.

El diseño de algoritmos no se reduce a simples recetas, sino que es necesaria la existencia y el uso de la creatividad. Lo que se puede hacer es dar un enfoque metódico para maximizar el rango de opciones consideradas, brindar mecanismos para evaluar las alternativas, y reducir el costo de tener que volver a empezar (backtracking) por malas elecciones.

Pueden distinguirse 4 etapas:

- Particionamiento (descomposición en tareas).
- Comunicación (estructura necesaria para coordinar la ejecución).
- Aglomeración (evaluación de tareas y estructura con respecto a costo, combinando tareas para mejorar el desempeño).
- Mapeo (asignación de tareas a procesadores).

2.2.2 Particionamiento.

Esta etapa intenta exponer oportunidades para la ejecución paralela. Se intenta definir un gran número de pequeñas tareas para obtener una descomposición de grano medio, para brindar la mayor flexibilidad a los algoritmos paralelos potenciales.

Una buena partición divide tanto el procesamiento como los datos. Podemos emplear técnicas de descomposición de dominio (determinar una división de los datos y luego asociarle la programación paralela) o técnicas de descomposición funcional (primero descompone la programación y luego trata de adaptar los datos). Cuando se va adquiriendo experiencia, el particionamiento puede realizarse de manera más minuciosa con el objetivo de reducir costos.

2.2.3 Comunicación.

Marca las transferencias de datos entre tareas o procesos. Pueden marcarse dos etapas: primero definir una estructura de canales que enlacen las tareas que cooperan y luego especificar los mensajes que viajan sobre ello.

En problemas de descomposición de dominio puede ser difícil de determinar la forma de comunicación. En algoritmos obtenidos por descomposición funcional es casi directo: corresponde a flujos de datos entre procesos.

Los patrones de comunicación pueden pertenecer a diferentes tipos de clasificaciones como lo son:

- Local o global.
- Estructurada o no estructurada.
- Estática odinámica.
- Sincrónica o asincrónica.

2.2.4 Aglomeración.

El algoritmo resultante de las etapas anteriores es abstracto en el sentido de que no es especializado para ejecución eficiente en una máquina particular.

En esta etapa se revisan las decisiones tomadas con la visión de obtener un algoritmo que ejecute en forma eficiente en la máquina paralela que se tenga. En particular, se considera si es útil combinar o aglomerar las tareas para obtener otras de mayor tamaño y lograr un comportamiento paralelo más eficaz.

Al concluir esta fase, la cantidad de tareas puede ser mayor que la cantidad de procesadores, por lo tanto es conveniente pensar cómo se distribuirá el trabajo entre los procesadores que se tengan, lo cual se analiza en la fase de mapeo.

Hay tres objetivos, a veces conflictivos, que guían las decisiones de aglomeración:

- **Incremento de la granularidad.** Intenta reducir la cantidad de comunicaciones combinando varias tareas relacionadas en una sola.

- **Preservación de la flexibilidad.** Al juntar tareas puede limitarse la escalabilidad del algoritmo. La capacidad para crear un número variante de tareas es crítica si se busca un programa portable y escalable.
- **Reducción de costos.** Se intenta evitar cambios extensivos, por ejemplo, reutilizando rutinas existentes.

La razón de que puedan ser conflictivos es que se da comúnmente el caso de que, para lograr mayor granularidad el costo tiende a ser mayor; o que al querer preservar la flexibilidad la granularidad puede verse afectada de manera negativa.

2.2.5 Mapeo.

Se especifica dónde ejecuta cada tarea. Este problema no existe en máquinas uniprocesadores o máquinas de memoria compartida con programador de tareas automático.

El objetivo es minimizar tiempo de ejecución. Aquí hay dos estrategias que aparecen en esta parte que a veces causan conflictos entre ellas: ubicar tareas que pueden ejecutarse en paralelo en distintos procesadores o poner las tareas que se comunican con frecuencia en el mismo procesador para incrementar la localidad y, por lo tanto, hacer menor el tiempo de ejecución.

2.2.6 Áreas de aplicación.

Las áreas de aplicación del cómputo paralelo son muchas y muy variadas. Algunas de ellas son las siguientes:

- Tratamiento de imágenes, con aplicaciones médicas, militares, visión por computadora, etc.
- Simulación de fenómenos físicos y químicos: dinámica molecular, física de partículas, etc.
- Métodos de análisis de elementos finitos (simulación de formaciones de metal).
- Sensado remoto, procesamiento de datos satelitales.
- Oceanografía (simulación de circulación oceánica).
- Aerodinámica computacional (simulación de túneles de viento, diseño y dinámica de vehículos).
- Reconocimiento de patrones en ADN.
- Inteligencia artificial, aprendizaje en redes neuronales, consultas en grandes Bases de Datos, predicción del clima, etc.

Las aplicaciones llevan, en muchos casos, a algoritmos comunes como:

- Manejo de arreglos (ordenación, búsqueda, reducciones, etc.)

- Algoritmos matriciales (multiplicación, trasposición, resolución de ecuaciones lineales).
- Problemas en árboles y grafos (búsquedas, camino crítico, ruteo de paquetes).

Muchos de éstos son problemas *Grand Challenge*: problemas fundamentales en ciencia e ingeniería con gran impacto económico y científico, y cuya solución puede obtenerse aplicando técnicas y recursos de computación de alto nivel.

3. MPI: Interfaz Estándar de Paso de Mensajes.

3.1 Introducción.

El paso de mensajes es una tarea ampliamente usada en ciertas clases de máquinas paralelas, especialmente en aquellas que cuentan con memoria distribuida. Aunque existen muchas variaciones, el concepto básico en el proceso de comunicación mediante mensajes está bien establecido. En los últimos 10 años, se ha logrado un avance substancial en convertir aplicaciones significativas hacia este tipo de tareas. Más recientemente, diferentes sistemas han demostrado que una interfaz de paso de mensajes puede ser implementada eficientemente y con un alto grado de portabilidad.

Al diseñarse MPI, se tomaron en cuenta las características más atractivas de los sistemas existentes para el paso de mensajes, en vez de seleccionar uno solo de ellos y adoptarlo como el estándar. Resultando así, en una fuerte influencia para MPI los trabajos hechos por 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, Chameleon y PICL.

La meta de la MPI o Message-Passing Interface (Interfaz de Paso de Mensajes) es desarrollar un estándar (que sea ampliamente usado) para escribir programas que implementen el paso de mensajes. Por lo cual, la interfaz intenta establecer un estándar práctico, portable, eficiente y flexible.

El esfuerzo para estandarizar a la MPI involucró a cerca de 60 personas de 40 organizaciones diferentes; principalmente de E.U.A. y de Europa. La mayoría de los vendedores de computadoras concurrentes estaban involucrados con MPI, junto con investigadores de diferentes universidades, laboratorios del gobierno e industrias. El proceso de estandarización comenzó en el Taller de Estándares para el Paso de Mensajes en un ambiente con memoria distribuida, patrocinado por el Centro para la Investigación en Computación Paralela en Williamsburg, Virginia (abril 29-30 de 1992). Se llegó a una propuesta preliminar conocida como MPI1, enfocada principalmente en comunicaciones punto a punto sin incluir rutinas para comunicación colectiva y no presentaba tareas seguras. El estándar final de la MPI fue presentado en la conferencia de Supercomputación 93, en noviembre de 1993, constituyéndose así el Foro para la MPI (MPI Forum).

En un ambiente de comunicación con memoria distribuida en el que las rutinas de más alto nivel se cimientan sobre rutinas de paso de mensajes del más bajo nivel, los beneficios de la estandarización son particularmente notorios. La

principal ventaja al establecer un estándar para el paso de mensajes es la portabilidad y el ser fácil de utilizar.

MPI es una interfaz compleja, comprende alrededor de 129 funciones, de las cuales la mayoría tiene muchos parámetros y variantes.

La interfaz es compatible con el uso de programas completamente generales del tipo MIMD, tanto como lo es con aquellos escritos en el estilo más restricto de SPMD.

Distribuciones de la MPI.

MPI es un estándar que posee varias implementaciones o distribuciones empleadas en distintas áreas de aplicación.

Algunas de las implementaciones de dominio público de MPI son:

- **CHIMP** (Common High-level Interface to Message Passing). Desarrollado por el Centro de Computación Paralela de Edimburgo.

- **LAM** (Local Area Multicomputer). Es un ambiente de programación y un sistema de desarrollo sobre redes de computadoras heterogéneas. Esta implementación es del Centro de Supercomputación de Ohio, de la Universidad de Indiana y de la Universisdad de Notre Dame.

- **MPICH** (MPI/Chameleon). Producida por el Laboratorio Nacional de Argonne y la Universidad del Estado de Mississippi. Su principal característica es su portabilidad, pues puede instalarse en un gran número de plataformas, tanto de estaciones de trabajo como de supercomputadoras. Otra característica importante es que permite definir los mecanismos de comunicación e incluso que diferentes mecanismos coexistan en una sola aplicación.

- **MPICH/NT**. Es una implementación de MPI para una red de estaciones con Windows NT. Está hecha con base en MPICH y es de la Universidad del Estado de Mississipi.

- **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.

- **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 de un uso de API dual; un mismo programa puede contener código de PVM y MPI.

- **W32MPI**. Implementación de MPI para un conglomerado con MS-Win32 basado también en MPICH. Fue creada por Instituto de Ingeniería Superior de Coimbra -Portugal y por la Universidad de Coimbra.

- **WinMPI**. WinMPI es la primera implementación de MPI para MS-Windows 3.1. Ésta corre sobre cualquier PC. Fue desarrollado por la Universidad de Nebraska en Omaha.

Además de las distribuciones de dominio público, también existen otras implementaciones de la MPI:

- **CRI/EPCC MPI para la Cray/T3D**. El Centro de Computación Paralela de Edimburgo (EPCC) ha trabajado con Cray Research Inc. (CRI) para desarrollarar una versión de MPI sobre la Cray T3D.

- **Implementación de MPI de SGI**.

- **MPI para la Fujitsu AP1000.** David Sitsky de la Universidad Nacional de Australia desarrolló una implementación de MPI para la Fujitsu AP1000.

- **MPIF para la IBM SP1/2.** MPIF es una implementación eficiente de IBM para la IBM SP1/2.

3.2 Metas de la MPI.

- Diseñar una interfaz de programación aplicable (no necesariamente para compiladores o sistemas que implementan una librería).

- Permitir una comunicación eficiente. Evitando el copiar de memoria a memoria y permitiendo donde sea posible la sobreposición de computación y comunicación, además de aligerar la comunicación con el procesador.

- Permitir implementaciones que puedan ser utilizadas en un ambiente heterogéneo.

- Permitir enlaces convenientes en C y Fortran 77.

- Asumir una interfaz de comunicación segura. El usuario no debe lidiar con fallas de comunicación. Tales fallas son controladas por el subsistema de comunicación interior.

- Definir una interfaz que no sea muy diferente a los actuales, tales como PVM, NX, Express, p4, etc., y proveer de extensiones para permitir mayor flexibilidad.

- Definir una interfaz que pueda ser implementada en diferentes plataformas, sin cambios significativos en el software y las funciones internas de comunicación.

- La semántica de la interfaz debe ser independiente del lenguaje.

- La interfaz debe ser diseñada para producir tareas seguras.

3.3 Modelo de programación.

En el modelo de programación MPI, un programa está constituido por uno o más procesos comunicados a través de llamadas a rutinas de librerías para enviar (*send*) y recibir (*receive*) mensajes a otros procesos. En la mayoría de las implementaciones de MPI, se crea un conjunto fijo de procesos al inicializar el programa, y un proceso es creado por cada tarea. Sin embargo, estos procesos pueden ejecutar diferentes programas. De ahí que, el modelo de programación de la MPI se adecua al que Flynn propone como MIMD.

Debido a que el número de procesos en un programa de MPI es normalmente fijo, se puede enfatizar en el uso de los mecanismos para comunicar datos entre procesos. Los procesos pueden utilizar operaciones de comunicación punto a punto para mandar mensajes de un proceso a otro, estas operaciones pueden ser usadas para implementar comunicaciones locales y no estructuradas.

Un grupo de procesos puede llamar colectivamente operaciones de comunicación para realizar tareas globales tales como transmisión, difusión, recolección, etc. La habilidad de MPI para probar mensajes da como resultado el soportar comunicaciones asíncronas. Probablemente una de las características más importantes de la MPI es el soporte para la programación modular. Un mecanismo llamado comunicador permite al programador de la MPI definir módulos que encapsulan estructuras internas de comunicación (estos módulos pueden ser combinados secuencial y paralelamente).

Un programa de MPI consiste en un grupo de procesos, ejecutando su propio código, con un estilo MIMD. Los códigos ejecutados por los diferentes procesos no necesariamente son idénticos. Los distintos procesos se coordinan por medio de llamadas a las primitivas de comunicación de MPI.

3.4 Convenciones y términos de la MPI estándar.

3.4.1 Especificación de los parámetros de una función.

Las funciones de la MPI estándar intentan usar una notación independiente del lenguaje de programación utilizado. Es por eso que los argumentos de las funciones utilizan la siguiente notación:

- **IN:** cuando la llamada a la función utiliza al argumento pero no lo modifica.
- **OUT:** cuando la llamada a la función modifica al argumento.
- **INOUT:** cuando la llamada a la función utiliza y modifica al argumento.

3.4.2 Términos semánticos.

Al hablar de los procedimientos de MPI, los siguientes términos semánticos son comúnmente usados.

- **Desbloqueo:** si el procedimiento puede regresar y permitir que se ejecuten otras instrucciones antes de que la operación iniciada por éste haya sido completada y antes de que el usuario esté en condiciones de reutilizar los recursos especificados al momento de la llamada al procedimiento.
- **Bloqueo:** si el regreso del procedimiento indica que el usuario puede reutilizar los recursos que fueron especificados en la llamada.

3.4.3 Estructuras internas.

La MPI maneja diferentes estructuras en su modelo de programación, a continuación se explica cada una de ellas.

Objetos opacos y manejadores.

MPI opera con dos clases de memoria: la *memoria del sistema* y la *memoria del usuario*. La memoria del sistema es utilizada como buffer para la transmisión de mensajes y para el almacenamiento de representaciones internas de varios objetos de MPI como lo son los grupos, los comunicadores, los tipos de datos, etc. Ya que la memoria del sistema no es directamente accesible al usuario se dice que los objetos almacenados en ésta son **opacos**: su forma y su tamaño no pueden ser conocidos por el usuario. Los objetos opacos son accesibles a través del uso de **manejadores**, los cuales se definen en la memoria del usuario. Por lo tanto, las funciones de MPI que operan sobre objetos opacos requieren de estos manejadores para poder acceder a tales objetos. Además de ser usados por MPI para poder tener acceso a los objetos opacos, los manejadores pueden participar en asignaciones y comparaciones.

Los objetos opacos son asignados y liberados por funciones que son específicas de cada tipo de objeto. Estas funciones aceptan un argumento manejador del tipo respectivo. En una función de asignación el manejador es un argumento OUT que regresa una referencia válida al objeto opaco. En una llamada de liberación el manejador es un argumento INOUT que regresa con un valor de "manejador nulo". MPI proporciona una constante "manejador nulo" para cada tipo de objeto. Las comparaciones con esta constante se utilizan para probar la validez de un manejador.

Una llamada de liberación invalida el manejador y marca el objeto opaco asociado a éste para ser liberado. El objeto ya no es accesible para el usuario después de la llamada. A pesar de ello, MPI no necesariamente libera al objeto inmediatamente después de la llamada. Cualquier operación que haya quedado pendiente e involucre a dicho objeto será completada de manera normal y el objeto será liberado después de ello.

Un objeto opaco y su manejador son significativos sólo en el proceso en el que el objeto fue creado, y no pueden ser trasferidos a otros procesos.

Constantes.

MPI define un conjunto de valores constates utilizados por algunas funciones. Estas constantes tienen un significado especial, por ejemplo, **MPI_ANY_TAG** es un valor utilizado como comodín para la etiqueta de un mensaje. MPI define también algunas constantes como valores válidos para manejadores como en el caso de **MPI_COMM_WORLD** que es un manejador de un objeto que representa a todos los procesos disponibles al momento de la iniciación de MPI y permite comunicarse con cualquiera de ellos.

Tipo de dato "choice".

En la descripción de algunas funciones de MPI se especifican argumentos de un tipo denominado **choice**, esta descripción indica que distintas llamadas a una de estas funciones pueden tener argumentos de distintos tipos. Por ejemplo, en el envío de mensajes se utiliza una misma función para transmitir datos de diferentes tipos, tales como enteros, de punto flotante, de doble precisión, etc. En C, para proveer este tipo de argumentos utilizamos **(void *)**,

3.4.4 Enlaces con el lenguaje C.

Todos los nombres de MPI utilizan el prefijo **MPI_**, las constantes predefinidas utilizan letras mayúsculas, y los tipos definidos así como las funciones tienen una letra mayúscula después del prefijo y después continúan con letras minúsculas. Por lo anterior, debe evitarse la declaración de variables, constantes o funciones cuyos nombres comiencen con el prefijo **MPI_** para evitar posibles colisiones con los nombres.

Casi todas las funciones regresan un código de error. El código de retorno exitoso es **MPI_SUCCESS**, pero los códigos de falla son dependientes de la implementación.

Las banderas lógicas son enteros con valor igual a cero para indicar un valor lógico *false* y con un valor distinto de cero para indicar *true*.

3.5 Funciones básicas.

Antes de utilizar cualquier función de comunicación se debe hacer una llamada a la función **MPI_Init**, con el fin de iniciar el ambiente de MPI. Al finalizar el programa, debe hacerse una llamada a la función de finalización **MPI_Finalize**. La sintaxis de ambas se muestra enseguida.

MPI_Init.

MPI_Init(argc, argv)

IN	**argc**	Apuntador al número de argumentos
IN	**argv**	Apuntador al vector de argumentos

int MPI_Init(int *argc, char ***argv)

MPI_Init permite iniciar el ambiente de MPI para poder, posteriormente, utilizar las funciones de esta interfaz. Explícitamente el estándar no especifica la forma en que un programa es arrancado o qué debe hacer el usuario con su ambiente para lograr ejecutar un programa en MPI. Sin embargo, una implementación puede requerir cierta configuración previa antes de poder ejecutar llamadas a las rutinas de MPI. Para esto, cada diferente implementación incluye una rutina de iniciación **MPI_Init**.

La MPI estándar no especifica argumentos en la línea de órdenes, pero lo permite a través de algunas implementaciones mediante los argumentos **argc** y **argv** de **MPI_Init**. En el caso de MPICH, los dichos argumentos son ignorados e inalterados.

La MPI estándar no especifica lo que un programa puede hacer antes de **MPI_Init** o después de **MPI_Finalize**. En la implementación de MPICH, uno debe realizar lo mínimo posible. En particular, se deben evitar los cambios al estado externo del programa, como abrir archivos, leer de la entrada estándar o escribir en ésta.

MPI_Finalize.

MPI_Finalize termina el ambiente de ejecución de MPI. Todos los procesos deben llamar a esta rutina antes de terminar. Es mejor no ejecutar mucho más que un return rc después de llamar a esta rutina. Su prototipo es el siguiente:

int MPI_Finalize(void)

Ejemplo: Inicio y fin de un cómputo con la MPI.

```
int main(int argc,char **argv){
     MPI_Init(&argc,&argv);
     /*Programa principal que usa las funciones de la MPI.*/
     MPI_Finalize();
     return 0;
}
```

Por otra parte, existen datos importantes acerca de un programa paralelo como lo son: el número de procesos que ejecutan el programa y el identificador o rango de cada proceso. Estos datos pueden ser obtenidos mediante las funciones **MPI_Comm_size** y **MPI_Comm_rank**.

MPI_Comm_size.

MPI_Comm_size(comm, size)

IN	**comm**	Comunicador (handle)
OUT	**size**	Número de procesos en el grupo de **comm** (entero)

int MPI_Comm_size(MPI_Comm comm, int *size)

Determina el tamaño del grupo de procesos asociado con un determinado comunicador.

MPI_Comm_rank.

MPI_Comm_rank(comm, rank)

IN	**Comm**	Comunicador (handle)
OUT	**Rank**	Rango del proceso que llama la función en el grupo de **comm** (entero)

int MPI_Comm_rank(MPI_Comm comm, int *rank)

Determina el rango del proceso que invoca la función en un determinado comunicador.

Ejemplo: Determinación del rango de cada proceso y el número de procesos que conforman al grupo.

```
int main(int argc,char *argv[]){
     int rango,tamanioGrupo;
     MPI_Init(&argc,&argv);
          MPI_Comm_size(MPI_COMM_WORLD,&tamanioGrupo);
          MPI_Comm_rank(MPI_COMM_WORLD,&rango);
          printf("Soy el proceso %d
                  de un total de %d\n",rango,tamanioGrupo);
     MPI_Finalize();
```

```
    return 0;
}
```

MPI_Wtime

Conocer el tiempo de ejecución de un programa o de partes de su código resulta de gran importancia para medir el desempeño del mismo y para poder realizar comparaciones con otros programas. Es por ello que MPI brinda una función, llamada **MPI_Wtime**, que permite cuantificar el tiempo, su sintaxis es:

double MPI_Wtime(void)

MPI_Wtime regresa un valor de doble precisión que corresponde al tiempo transcurrido desde un tiempo arbitrario en el pasado hasta el momento en que es invocada la función. MPI garantiza que este "tiempo arbitrario en el pasado" no cambie durante la vida del proceso. El tiempo regresado es medido en segundos y es local al nodo en que fue invocada la función.

Ejemplo: La función **MPI_Wtime** puede ser utilizada como se muestra en el siguiente código para realizar mediciones de tiempo de ejecución.

```
{
   double inicio,fin;
   inicio=MPI_Wtime();
   /* Segmento de código que se desea cronometrar */
   fin=MPI_Wtime();
   printf("Transcurrieron %lf segundos.\n",fin-inicio);
}
```

MPI_Abort

En ocasiones es necesario concluir de forma no ordinaria el ambiente de ejecución de la MPI, es decir, forzar su terminación. Para esto puede ser utilizada la función **MPI_Abort**, su sintaxis es:

MPI_Abort(comm, errorcode)

IN	**comm**	Comunicador de los procesos que serán abortados (handle)
IN	**errorcode**	Código de error que será regresado al abortar el ambiente (entero)

int MPI_Abort(MPI_Comm comm, int errorcode)

MPI_Abort finaliza todos los procesos asociados al comunicador comm.

Ejemplo: Código que utiliza a la función **MPI_Abort**.

```
int main(int argc,char *argv[]){
     MPI_Init(&argc,&argv);
          if(argc!=3){
               printf("Formato de ejecución:\n
                       prog <argumento1> <argumento2>\n");
               MPI_Abort(MPI_COMM_WORLD,1);
          }
          ...
     MPI_Finalize();
     return 0;
}
```

3.6 Comunicación punto a punto.

El envío y la recepción de mensajes entre procesos es el mecanismo básico de comunicación de MPI. Mediante estas operaciones los procesos que intervienen en la realización de una tarea conjunta pueden intercambiar información y sincronizarse.

3.6.1 Operación de bloqueo *Send*.

La sintaxis de la operación de bloqueo *send* se muestra a continuación.

MPI_Send(buf, count, datatype, dest, tag, comm)

IN	**buf**	Dirección inicial del buffer de envío (choice)
IN	**count**	Número de elementos en el buffer de envío (entero no negativo)
IN	**datatype**	Tipo de dato de cada elemento del buffer de envío (handle)
IN	**dest**	Rango del destino (entero)
IN	**tag**	Etiqueta o identificador del mensaje (entero)
IN	**comm**	Comunicador (handle)

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Datos del mensaje.

En este esquema de comunicación utilizado por MPI, el mensaje se considera dividido en dos partes: los datos del mensaje y el sobre del mensaje. En esta sección explicaremos los pormenores relacionados con los datos del mensaje enviado por medio de la función **MPI_Send**.

El buffer de envío **buf** especificado por la operación **MPI_Send** consiste en **count** entradas sucesivas del tipo indicado por **datatype**, iniciando con la entrada que se encuentra en la dirección apuntada por **buf**.

La parte de datos del mensaje consiste en una secuencia de **count** valores, cada uno de tipo **datatype**. **count** puede ser cero y en este caso la parte de datos del mensaje está vacía. Los tipos de datos básicos que pueden ser utilizados para el argumento **datatype** se muestran en la tabla 3.1.

Tipo de dato de MPI	Tipo de dato de C
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	Int
MPI_LONG	long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Tabla 3.1. Correspondencia entre los tipos de datos de C y los tipos que define la MPI.

Como puede observarse en la tabla anterior, todos los tipos de datos del lenguaje C tienen un correspondiente tipo de dato bajo MPI, el uso de tipos de datos propios de la interfaz permite la realización de programas portables entre diferentes plataformas.

Los tipos de datos **MPI_BYTE** y **MPI_PACKED** no corresponden a algún tipo de datos de C. Un valor del tipo **MPI_BYTE** consiste en un byte (8 bits). Un byte es diferente de un carácter ya que diferentes plataformas pueden tener distintas representaciones para los caracteres o pueden usar más o menos bits para representar los caracteres. Por otra parte, un valor **MPI_BYTE** tiene la ventaja de ser igualmente representado entre máquinas distintas y siempre tendrá un valor de 8 bits. El tipo de dato **MPI_PACKED** no será abordo en esta tesis.

Sobre del mensaje.

Además de la parte de datos, un mensaje cuenta con información que es usada para distinguirlo y así poder seleccionarlo para enviarlo al receptor adecuado. Esta información consiste en un número de parámetros fijo que en

conjunto forman lo que se conoce como el sobre del mensaje. Estos parámetros son: fuente, destino, etiqueta y comunicador.

La fuente del mensaje se determina de forma implícita identificando al emisor del mensaje. Los otros campos se especifican por medio de los argumentos de la operación **MPI_Send.**

El destino del mensaje es especificado por medio del valor del argumento **dest**.

La etiqueta o identificador del mensaje se especifica por medio del argumento **tag**. El rango de valores válidos para **tag** es **0, 1, ..., LS**, donde **LS** depende de la implementación de MPI utilizada. A pesar de esto, MPI requiere que **LS** no sea menor que 32767.

El argumento **comm** especifica el comunicador que será utilizado para la operación de envío. Un comunicador especifica el contexto de la comunicación que será utilizado en la operación. Cada contexto de comunicación provee un "universo de comunicación" separado: los mensajes son siempre recibidos dentro del contexto donde fueron enviados y dos mensajes enviados en diferentes contextos no pueden tener interferencia mutua.

El comunicador también especifica el conjunto de procesos que comparten el mismo contexto de comunicación. Este grupo de procesos está ordenado y los procesos se identifican por medio de su rango dentro del grupo. El rango de valores válidos para el parámetro **dest** es **0,1,..., n-1**; donde **n** es el número de procesos en el grupo.

Un comunicador predefinido **MPI_COMM_WORLD** es provisto por MPI. Éste permite la comunicación con todos los procesos que son accesibles después de la inicialización de MPI y los procesos se identifican por su rango en el grupo de **MPI_COMM_WORLD**.

3.6.2 Operación de bloqueo *Receive.*

La sintaxis de la operación de bloqueo *receive* se muestra a continuación.

MPI_Recv(buf, count, datatype, source, tag, comm, status)

OUT	**buf**	Dirección inicial del buffer de recepción (choice)
IN	**count**	Número de elementos en el buffer de recepción (entero)
IN	**datatype**	Tipo de dato de cada elemento del buffer de recepción (handle)
IN	**source**	Rango de la fuente (entero)
IN	**tag**	Etiqueta o identificador del mensaje (entero)
IN	**comm**	Comunicador (handle)

OUT **status** Objeto estado (Status)

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

El buffer de recepción **buf** es un almacenamiento que contiene **count** elementos consecutivos del tipo especificado por **datatype**, iniciando en la dirección **buf**. La longitud del mensaje recibido debe ser menor o igual a la longitud del buffer de recepción. Un error de sobre flujo ocurrirá si no todo el mensaje recibido se ajusta, sin truncamiento, al buffer de recepción.

Si el mensaje recibido es más corto que el buffer de recepción, entonces sólo las localidades de memoria correspondientes al espacio que ocupa el mensaje serán modificadas.

La selección de un mensaje por una operación de recepción se realiza a través del sobre del mensaje. Un mensaje podrá ser recibido por la función **MPI_Recv** si y sólo si su sobre tiene los mismos valores de **source**, **tag** y **comm** que la operación de recepción. El receptor puede especificar un valor comodín **MPI_ANY_SOURCE** para **source**, o un valor comodín **MPI_ANY_TAG** para **tag**, indicando que cualesquiera **source** o **tag** son aceptables. No existen comodines para el parámetro **comm**.

En conclusión, un proceso podrá recibir un mensaje mediante **MPI_Recv** si dicho mensaje fue dirigido a él y ambos, emisor y receptor, utilizan el mismo comunicador, usan el mismo **source** (o **MPI_ANY_SOURCE** en el receptor) y poseen la misma etiqueta **tag** (o **MPI_ANY_TAG** en el receptor).

La etiqueta del mensaje se especifica por medio del argumento **tag**. El argumento **source**, si es diferente de **MPI_ANY_SOURCE**, se especifica como uno de los rangos dentro del grupo de procesos asociados con el mismo comunicador. Así, el rango válido para el argumento source es {0,1,...,n-1}$\cup${**MPI_ANY_SOURCE**}, donde n es el número de procesos en el grupo.

Es posible que un proceso pueda enviarse un mensaje a sí mismo, pero es inseguro ya que esto podría llevar a un interbloqueo si se utilizan las operaciones envío y recepción de bloqueo.

El argumento status.

La fuente y la etiqueta de un mensaje recibido no podrían conocerse si alguno de los valores comodín es utilizado en la operación de recepción. De igual forma, si múltiples operaciones son efectuadas por una misma función de MPI, un código de error diferente necesitará ser devuelto.

Toda esta información es regresada por el argumento **status** de **MPI_Recv**, el tipo de dato de **status** es **MPI_Status**. Las variables de este tipo necesitan ser

asignadas explícitamente por el usuario ya que no son objetos de sistema. En el contexto del lenguaje C, **status** es una estructura que contiene tres campos llamados **MPI_Source**, **MPI_Tag** y **MPI_Error**; la estructura puede contener campos adicionales dependiendo de la implementación de MPI que se esté manejando. Así **status.MPI_Source**, **status.MPI_Tag** y **status.MPI_Error** contienen la fuente, la etiqueta y el código de error, respectivamente, del mensaje recibido.

En general, el campo que contiene el código de error, es modificado únicamente por funciones que regresan múltiples códigos de error ya que las funciones que sólo regresan uno lo hacen a través del valor de su valor de retorno.

Ejemplo: Operaciones de envío y recepción de datos.

```
int main(int argc,char *argv[]){
     int rango,tamanioGrupo;
     MPI_Status status;
     char c='A';
     MPI_Init(&argc,&argv);
          MPI_Comm_size(MPI_COMM_WORLD,&tamanioGrupo);
          MPI_Comm_rank(MPI_COMM_WORLD,&rango);
          c+=rango;
          if(rango<tamanioGrupo-1)
               MPI_Send(&c,1,MPI_CHAR,
                                   rango+1,0,MPI_COMM_WORLD);
          else
               MPI_Send(&c,1,MPI_CHAR,0,0,MPI_COMM_WORLD);
          MPI_Recv(&c,1,MPI_CHAR,MPI_ANY_SOURCE,
                         MPI_ANY_TAG,MPI_COMM_WORLD,&status);
          printf("Proceso %3d:
                  recibí el carácter '%c'
                  del proceso %3d\n",
                  rango,c,status.MPI_SOURCE);
     MPI_Finalize();
     return 0;
}
```

El argumento **status** también provee información sobre la longitud del mensaje recibido. Sin embargo, esta información no está directamente disponible como un campo de la variable **status**, por lo que se requiere una llamada a **MPI_Get_Count** para poder decodificar esta información.

La sintaxis de la función **MPI_Get_count** se muestra a continuación.

MPI_Get_count(status, datatype, count)

IN	**status**	Parámetro status regresado por la operación de recepción (Status)
IN	**datatype**	Tipo de dato de cada elemento del buffer de recepción (handle)
OUT	**count**	Número de elementos recibidos (entero)

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

Esta función regresa el número de entradas recibidas (se cuenta el número de entradas, cada una del tipo *datatype*, no los bytes recibidos). El argumento **datatype** debe ser el mismo que el argumento utilizado en la llamada de recepción que produjo la variable **status**.

3.7 Comunicaciones colectivas.

3.7.1 Introducción.

Una comunicación colectiva se define como una función que involucra a un grupo determinado de procesos. Las funciones de este tipo provistas por MPI son las siguientes:

- Barrera de sincronización a través de todos los miembros de un grupo.
- Transmisión de datos de un miembro a todos los miembros de un grupo.
- Recolección de datos de los miembros de un grupo en un miembro.
- Dispersión de los datos de un miembro a todos los miembros de un grupo.
- Una variante de la recolección donde todos los miembros del grupo reciben el resultado.
- Dispersión y recolección de datos de todos los miembros a todos los miembros de un grupo.
- Operaciones globales de reducción como suma, máximo u operaciones definidas por el usuario, donde el resultado se regresa a todos los miembros del grupo y una variante donde el resultado se regresa sólo a un miembro.
- Una operación que combina la operación de reducción y la operación de dispersión.
- Exploración a través de todos los miembros del grupo.

Las funciones de comunicaciones colectivas se ilustran, en la figura 3.1, para un grupo de seis procesos. Cada renglón de un recuadro representa localidades de datos pertenecientes a un proceso. Así, por ejemplo, el renglón número dos representa localidades de datos del proceso con rango igual a dos, de igual manera, cada elemento del renglón dos representa a un dato de este proceso, dicho dato puede ser primitivo o derivado.

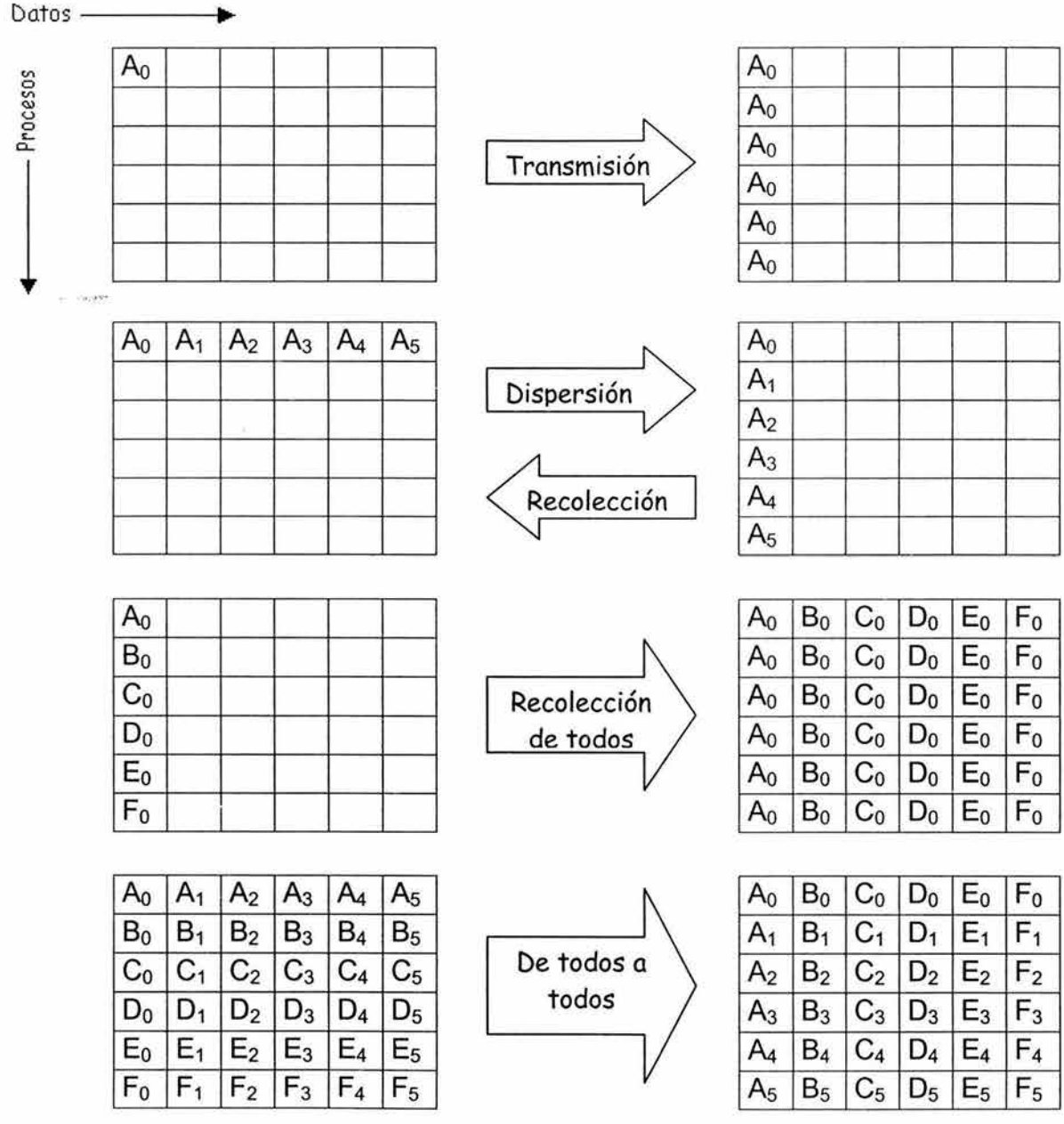


Figura 3.1. Comunicaciones colectivas.

Una operación colectiva se ejecuta cada vez que todos los procesos dentro de un grupo llaman a la misma rutina de comunicación con argumentos relacionados. La sintaxis y la semántica de las operaciones colectivas están

definidas para ser consistentes con la sintaxis y la semántica de las operaciones punto a punto. Por eso, uno de los argumentos clave de estas operaciones es un comunicador que define el grupo de procesos participantes y provee un contexto para la operación.

Varias rutinas colectivas tienen un proceso transmisor o receptor único que se conoce como raíz. Algunos argumentos de las funciones colectivas son sólo significativos para el proceso raíz, es decir que, son ignorados por todos los procesos participantes excepto por él.

La concordancia entre el tipo de dato enviado por el emisor y el tipo de dato empleado por receptor en las comunicaciones colectivas es más estricta que en las comunicaciones punto a punto. Es decir, para operaciones colectivas, la cantidad de datos enviada debe coincidir exactamente con la cantidad de datos especificada en el receptor.

La completitud de una función colectiva indica que el proceso que la llamó es libre de acceder a las localidades que constituyen al buffer de comunicación. Esto no indica que otros procesos en el grupo hayan terminado o incluso empezado la operación. Entonces, una comunicación colectiva puede o no tener el efecto de sincronizar a los procesos que la llamaron, por lo tanto, es importante no crear programas que se basen en la suposición de tal efecto.

3.7.2 Argumento de comunicación.

El concepto clave de las funciones colectivas es tener un grupo de procesos participantes. Las rutinas de este tipo de comunicación no tienen un identificador de grupo como argumento explícito. En su lugar, tiene un comunicador. Con el propósito de comprender las comunicaciones colectivas, un comunicador puede ser visto como un identificador de grupo ligado a un contexto.

3.7.3 Barrera de sincronización.

La sintaxis de esta función se muestra a continuación:

MPI_Barrier(comm)

IN **comm** Comunicador (handle)

int MPI_Barrier(MPI_Comm comm)

La función **MPI_Barrier** bloquea al proceso que la llama hasta que todos los miembros del grupo hayan hecho la llamada a esta función. La función puede finalizarse en un proceso sólo después de que todos lo miembros del grupo hayan entrado en ella.

Ejemplo: Uso de la barrera de sincronización.

```
MPI_Comm com;
int rango;
...
MPI_Comm_rank(com,&rango);
if(!rango){
     /* Segmento de código 1 */
     ...
     MPI_Barrier(com);
     /* Segmento de código 2 */
     ...
}else{
     MPI_Barrier(com);
     /* Segmento de código 3 */
     ...
}
```

En el ejemplo anterior, debido a la función de sincronización, los procesos pertenecientes al comunicador **com** no podrán comenzar a ejecutar los segmentos de código 2 y 3 hasta que todos los procesos hayan realizado la llamada a la barrera de sincronización.

3.7.4 Transmisión.

La sintaxis de la función **MPI_Bcast** se muestra a continuación.

MPI_Bcast(buffer, count, datatype, root, comm)

INOUT	**buffers**	Dirección inicial del buffer (choice)
IN	**count**	Número de elementos en el buffer (entero)
IN	**datatype**	Tipo de dato de cada elemento del buffer (handle)
IN	**root**	Identificador del proceso raíz (entero)
IN	**comm**	Comunicador (handle)

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

MPI_Bcast transmite un mensaje desde el proceso con el identificador o el rango de **root** a todos los procesos en el grupo, incluyéndose a él mismo. La función debe ser llamada por todos los procesos del grupo usando los mismos argumentos para **comm** y **root**. Como resultado de la llamada, el contenido del buffer de comunicación del proceso **root** es copiado a todos los procesos.

Los tipos de datos derivados son permitidos para **datatype**. La firma **count**, **datatype** en cualquier proceso debe ser igual a la firma **count**, **datatype** del

proceso raíz. Lo anterior implica que la cantidad de datos enviada debe ser igual a la cantidad de datos recibida, entre cada proceso y el proceso raíz. **MPI_Bcast** y todas las otras rutinas colectivas de movimiento de datos tienen esta restricción. Distintos tipos de mapeos de memoria son permitidos entre el emisor y el receptor.

```
MPI_Comm com;
int rango,raiz;
double param;
...
MPI_Comm_rank(com,&rango);
if(rango==raiz)param=-97.124;
MPI_Bcast(&param,1,MPI_DOUBLE,raiz,com);
```

Después de la llamada a la función **MPI_Bcast**, el valor de la variable **param** de todos los procesos pertenecientes al grupo del comunicador **com** será igual a **-97.124**.

3.7.5 Recolección.

La sintaxis de la función **MPI_Gather** se muestra a continuación.

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
IN	**sendcount**	Número de elementos en el buffer de envío (entero)
IN	**sendtype**	Tipo de dato de cada elemento del buffer de envío (handle)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice, significativo sólo para el proceso raíz)
IN	**recvcount**	Número de elementos para cada recepción (entero, significativo sólo para el proceso raíz)
IN	**recvtype**	Tipo de dato de los elementos del buffer de recepción (handle, significativo sólo para el proceso raíz)
IN	**root**	Identificador del proceso raíz (entero)
IN	**comm**	Comunicador (handle)

int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Cada proceso (incluyendo al proceso raíz) envía el contenido de su buffer de envío al proceso raíz. El proceso raíz recibe los mensajes y los almacena de acuerdo al rango de cada proceso. El resultado es equivalente al que se obtendría si cada uno de los n procesos en el grupo (incluyendo al proceso raíz) hubiera ejecutado una llamada a

MPI_Send(sendbuf, sendcount, sendtype, root, ...),

y el proceso raíz hubiera ejecutado n llamadas a

MPI_Recv(recvbuf+i*recvcount*extent(recvtype), recvcount, recvtype, i,...),

donde extent(recvtype) es la extensión del tipo de dato obtenida por medio de una llamada a la función MPI_Type_extent().

Una descripción alternativa es que los n mensajes enviados por los procesos del grupo son concatenados en orden con respecto al rango del proceso que los envía y el mensaje resultante es recibido por el proceso raíz mediante una llamada a:

MPI_Recv(recvbuf, recvcount*n, recvtype, ...)

El buffer de recepción es ignorado por todos los procesos, excepto por el proceso raíz.

Tipos de datos derivados son permitidos para **sendtype** y **recvtype**. La firma **sendcount**, **sendtype** en el proceso iésimo debe ser igual a la firma **recvcount**, **recvtype** del proceso raíz. Lo anterior implica que la cantidad de datos enviada debe ser igual a la cantidad de datos recibida, entre cada proceso y el proceso raíz. Distintos tipos de mapeos de memoria son permitidos entre el emisor y el receptor.

Todos los argumentos de la función son significativos para el proceso raíz, mientras que, para los otros procesos sólo los argumentos **sendbuf**, **sendcount**, **sendtype**, **root** y **comm** son significativos. Los argumentos **root** y **comm** deben tener valores idénticos para todos los procesos.

Nótese que el argumento **recvcount** del proceso raíz indica el número de elementos que éste recibe de cada proceso, no el número total de elementos que recibe.

Ejemplo: El proceso raíz recolecta 100 enteros de cada proceso del grupo, ver figura 3.2.

```
MPI_Comm com;
int rango,tamGrupo,raiz,envio[100],*recepcion;
...
MPI_Comm_rank(com,&rango);
if(rango==raiz){
     MPI_Comm_size(com,&tamGrupo);
     recepcion=(int *)malloc(100*tamGrupo*sizeof(int));
}
MPI_Gather(envio,100,MPI_INT,recepcion,100,MPI_INT,raiz,com);
```

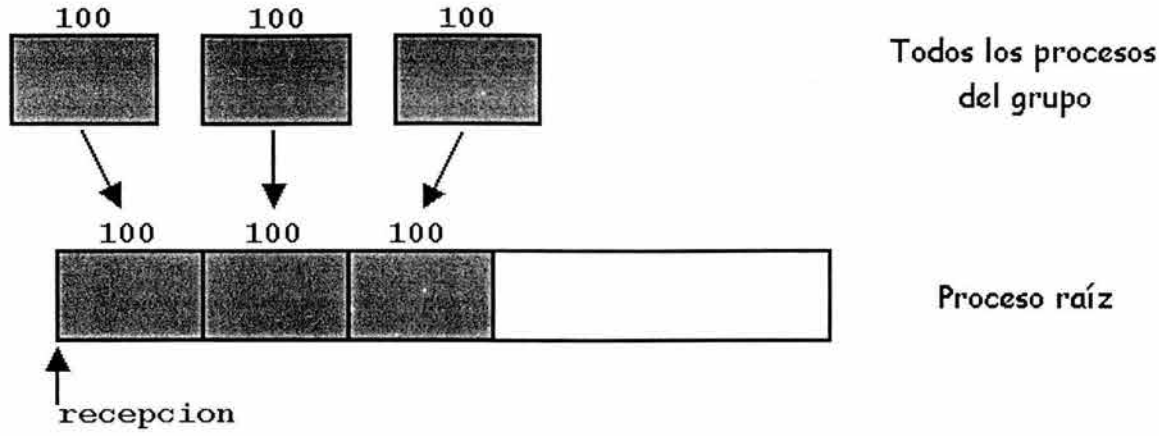


Figura 3.2. El proceso raíz recolecta 100 enteros de cada proceso del grupo.

Existe otra función que provee MPI para realizar la recolección de datos llamada **MPI_Gatherv**, su sintaxis se muestra a continuación.

MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
IN	**sendcount**	Número de elementos en el buffer de envío (entero)
IN	**sendtype**	Tipo de dato de cada elemento del buffer de envío (handle)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice, significativo sólo para el proceso raíz)
IN	**recvcounts**	Arreglo de enteros (de tamaño igual al total de elementos del grupo) que contiene el número de elementos que son recibidos por parte de cada proceso (significativo sólo para el proceso raíz)
IN	**displs**	Arreglo de enteros (de tamaño igual al total de elementos del grupo). El iésimo elemento especifica el desplazamiento, relativo a **recvbuf**, a partir del que se colocarán los datos de entrada del proceso iésimo (significativo sólo para el proceso raíz)
IN	**recvtype**	Tipo de dato de los elementos del buffer de recepción (handle, significativo sólo para el proceso raíz)
IN	**root**	Identificador del proceso raíz (entero)
IN	**comm**	Comunicador (handle)

int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)

MPI_Gatherv extiende la funcionalidad de **MPI_Gather** al permitir que cada proceso envíe diferente cantidad de datos al proceso raíz, ya que **recvcounts** es

ahora un arreglo. Esta función también brinda mayor flexibilidad ya que permite indicar dónde serán colocados los datos en el proceso raíz gracias a que posee un nuevo argumento llamado **displs**.

El resultado es equivalente al que se obtendría si cada uno de los n procesos en el grupo (incluyendo al proceso raíz) hubiera ejecutado una llamada a

MPI_Send(sendbuf, sendcount, sendtype, root, ...),

y el proceso raíz hubiera ejecutado n llamadas a

MPI_Recv(recvbuf+displs[i]*extent(recvtype), recvcounts[i], recvtype, i,...),

donde extent(recvtype) es la extensión del tipo de dato obtenida por medio de una llamada a la función MPI_Type_extent().

Los mensajes son colocados en el buffer de recepción del proceso raíz ordenados de acuerdo a su rango, esto es, los datos enviados por el proceso **j** son colocados en la j-ésima porción del buffer de recepción **recvbuf** del proceso raíz. La j-ésima porción de **recvbuf** comienza después de un desplazamiento de **displs[j]** elementos (en términos de **recvtype**) dentro de **recvbuf**.

El buffer de recepción es ignorado por todos los procesos, excepto por el raíz. Tipos de datos derivados son permitidos para **sendtype** y **recvtype**. La firma **sendcount**, **sendtype** en el proceso iésimo debe ser igual a la firma **recvcounts[i]**, **recvtype** del proceso raíz. Lo anterior implica que la cantidad de datos enviada debe ser igual a la cantidad de datos recibida, entre cada proceso y el proceso raíz. Distintos tipos de mapeos de memoria son permitidos entre el emisor y el receptor.

Todos los argumentos de la función son significativos para el proceso raíz, mientras que, para los otros procesos sólo los argumentos **sendbuf**, **sendcount**, **sendtype**, **root** y **comm** son significativos. Los argumentos **root** y **comm** deben tener valores idénticos para todos los procesos.

Ejemplo: Cada proceso enviará **TAM** enteros al proceso raíz y éste los colocará separados entre sí un número de enteros igual a **sep**. Es necesario mencionar que el programa será incorrecto si **sep < TAM**.Ver figura 3.3.

```
#define TAM 25
...
MPI_Comm com;
int sep,i;
int rango,tamGrupo,raiz;
int envio[TAM],*recepcion;
int *desplazamientos,*cantidades;
...
```

```
MPI_Comm_rank(com,&rango);
if(rango==raiz){
     MPI_Comm_size(com,&tamGrupo);
     recepcion=(int *)malloc(sep*tamGrupo*sizeof(int));
     desplazamientos=(int *)malloc(tamGrupo*sizeof(int));
     cantidades=(int *)malloc(tamGrupo*sizeof(int));
     for(i=0;i<tamGrupo;i++){
          desplazamientos[i]=i*sep;
          cantidades[i]=TAM;
     }
}
MPI_Gatherv(envio,TAM,MPI_INT,recepcion,
            cantidades,desplazamientos,MPI_INT,raiz,com);
```

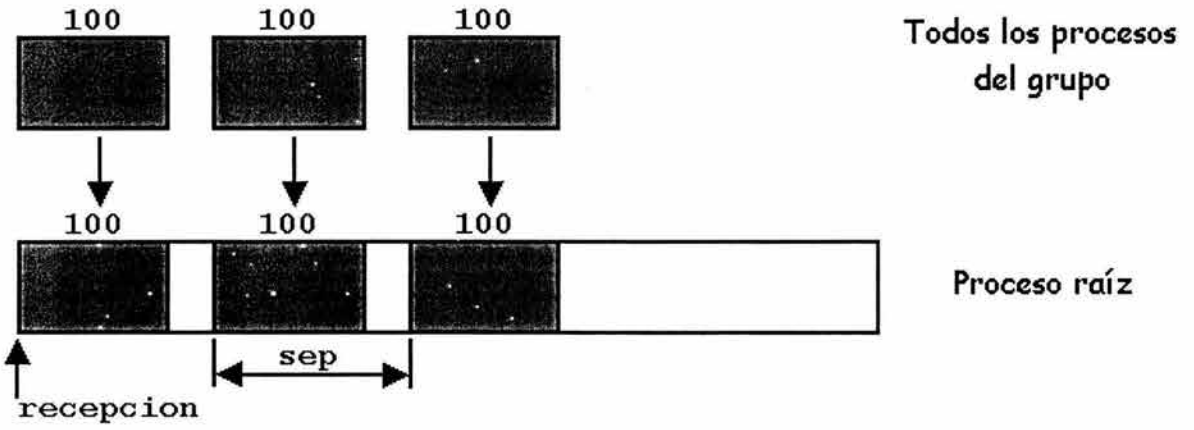


Figura 3.3. El proceso raíz recolecta 100 enteros de cada proceso del grupo y los coloca separados entre sí un número de enteros igual a **sep**.

3.7.6 Dispersión.

La sintaxis de la función **MPI_Scatter** se muestra a continuación.

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice, significativo sólo para el proceso raíz)
IN	**sendcount**	Número de elementos enviados a cada proceso (entero, significativo sólo para el proceso raíz)
IN	**sendtype**	Tipo de dato de cada elemento del buffer de envío (handle, significativo sólo para el proceso raíz)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice)
IN	**recvcount**	Número de elementos en el buffer de recepción (entero)

IN	**recvtype**	Tipo de dato de los elementos del buffer de recepción (handle)
IN	**root**	Identificador del proceso raíz (entero)
IN	**comm**	Comunicador (handle)

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

MPI_Scatter es la operación inversa de **MPI_Gather**. El resultado de su aplicación es equivalente al que se obtendría si el proceso raíz hubiera ejecutado n operaciones de envío,

MPI_Send(sendbuf+i*sendcount*extent(sendtype), sendcount, sendtype, i, ...),

donde extent(recvtype) es la extensión del tipo de dato obtenida por medio de una llamada a la función MPI_Type_extent(), y cada proceso hubiera ejecutado una operación de recepción,

MPI_Recv(recvbuf, recvcount, recvtype, i,...).

Una descripción alternativa es que el proceso raíz envía un mensaje por medio de

MPI_Send(sendbuf, sendcount*n, sendtype, ...),

dicho mensaje es dividido en n segmentos iguales, el iésimo segmento es enviado al iésimo proceso del grupo y cada proceso recibe este mensaje con la operación de recepción inmediata anterior.

El buffer de envío es ignorado por todos los procesos, excepto por el proceso raíz.

La firma **sendcount**, **sendtype** del proceso raíz debe ser igual a la firma **recvcount**, **recvtype** de todos los procesos. Lo anterior implica que la cantidad de datos enviada debe ser igual a la cantidad de datos recibida, entre el proceso raíz y cada proceso. Distintos tipos de mapeos de memoria son permitidos entre el emisor y el receptor.

Todos los argumentos de la función son significativos para el proceso raíz, mientras que, para los otros procesos sólo los argumentos **recvbuf**, **recvcount**, **recvtype**, **root** y **comm** son significativos. Los argumentos **root** y **comm** deben tener valores idénticos para todos los procesos.

Ejemplo: El proceso raíz envía 100 enteros a cada proceso del grupo, ver figura 3.4.

```
MPI_Comm com;
```

```
int rango,tamGrupo,raiz,*envio,recepcion[100];
...
MPI_Comm_rank(com,&rango);
if(rango==raiz){
     MPI_Comm_size(com,&tamGrupo);
     envio=(int *)malloc(100*tamGrupo*sizeof(int));
     ...
}
MPI_Scatter(envio,100,MPI_INT,recepcion,
                                        100,MPI_INT,raiz,com);
```

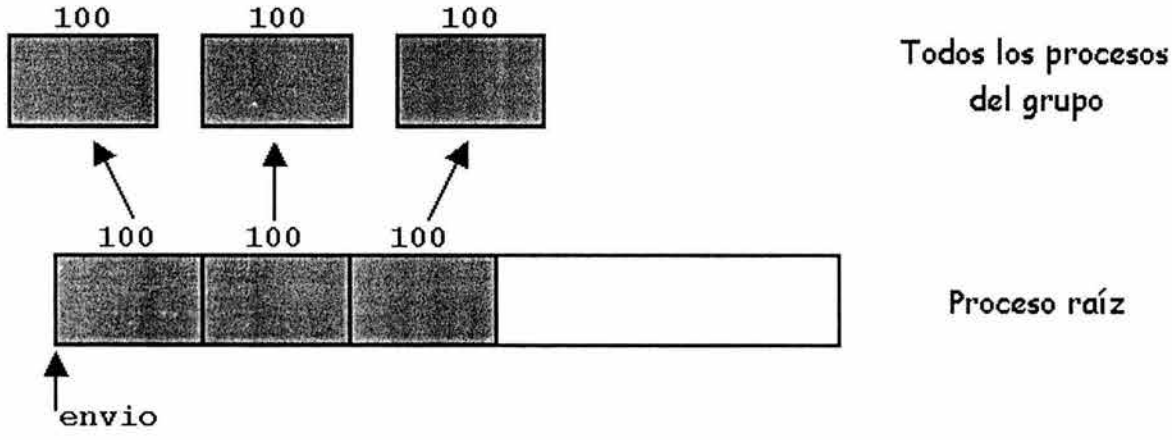


Figura 3.4. El proceso raíz envía 100 enteros a cada proceso del grupo.

Existe otra función que provee MPI para realizar la dispersión de datos llamada **MPI_Scatterv**, su sintaxis se muestra a continuación.

MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice, significativo sólo para el proceso raíz)
IN	**sendcounts**	Arreglo de enteros (de tamaño igual al total de elementos del grupo) que contiene el número de elementos que son enviados a cada proceso (significativo sólo para el proceso raíz)
IN	**displs**	Arreglo de enteros (de tamaño igual al total de elementos del grupo). El iésimo elemento especifica el desplazamiento, relativo a **sendbuf**, a partir del que se toman los datos que se envían al proceso iésimo (significativo sólo para el proceso raíz)
IN	**sendtype**	Tipo de dato de cada elemento del buffer de envío (handle)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice)
IN	**recvcount**	Número de elementos en el buffer de recepción (entero)

IN	**recvtype**	Tipo de dato de los elementos del buffer de recepción (handle)
IN	**root**	Identificador del proceso raíz (entero)
IN	**comm**	Comunicador (handle)

int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

MPI_Scatterv es la operación inversa de **MPI_Gatherv**.

MPI_Scatterv extiende la funcionalidad de **MPI_Scatter** al permitir que se envíe diferente cantidad de datos a cada proceso, ya que **sendcounts** es ahora un arreglo. Esta función también brinda mayor flexibilidad ya que permite indicar de dónde serán leídos los datos del proceso raíz gracias a que posee un nuevo argumento llamado **displs**.

El resultado de su aplicación es equivalente al que se obtendría si el proceso raíz hubiera ejecutado n operaciones de envío,

MPI_Send(sendbuf+displs[i]*extent(sendtype), sendcounts[i], sendtype, i,...),

donde extent(recvtype) es la extensión del tipo de dato obtenida por medio de una llamada a la función MPI_Type_extent(), y cada proceso hubiera ejecutado una operación de recepción,

MPI_Recv(recvbuf, recvcount, recvtype, i,...).

El buffer de envío es ignorado por todos los procesos, excepto por el proceso raíz.

La firma **sendcounts[i]**, **sendtype** del proceso raíz debe ser igual a la firma **recvcount**, **recvtype** del proceso iésimo. Lo anterior implica que la cantidad de datos enviada debe ser igual a la cantidad de datos recibida, entre el proceso raíz y cada proceso. Distintos tipos de mapeos de memoria son permitidos entre el emisor y el receptor.

Todos los argumentos de la función son significativos para el proceso raíz, mientras que, para los otros procesos sólo los argumentos **recvbuf**, **recvcount**, **recvtype**, **root** y **comm** son significativos. Los argumentos **root** y **comm** deben tener valores idénticos para todos los procesos.

Ejemplo: El proceso raíz envía una colección de `TAM` enteros a cada proceso del grupo y cada colección, en el buffer de envío, está separada entre sí un número de enteros igual a `sep`. El programa será incorrecto si `sep < TAM`.Ver figura 3.5.

```
#define TAM 25
...
MPI_Comm com;
int sep,i;
int rango,tamGrupo,raiz;
int *envio,recepcion[TAM];
int *desplazamientos,*cantidades;
...
MPI_Comm_rank(com,&rango);
if(rango==raiz){
     MPI_Comm_size(com,&tamGrupo);
     envio=(int *)malloc(sep*tamGrupo*sizeof(int));
     ...
     desplazamientos=(int *)malloc(tamGrupo*sizeof(int));
     cantidades=(int *)malloc(tamGrupo*sizeof(int));
     for(i=0;i<tamGrupo;i++){
          desplazamientos[i]=i*sep;
          cantidades[i]=TAM;
     }
}
MPI_Scatterv(envio,cantidades,desplazamientos,
                    MPI_INT,recepcion,TAM,MPI_INT,raiz,com);
```

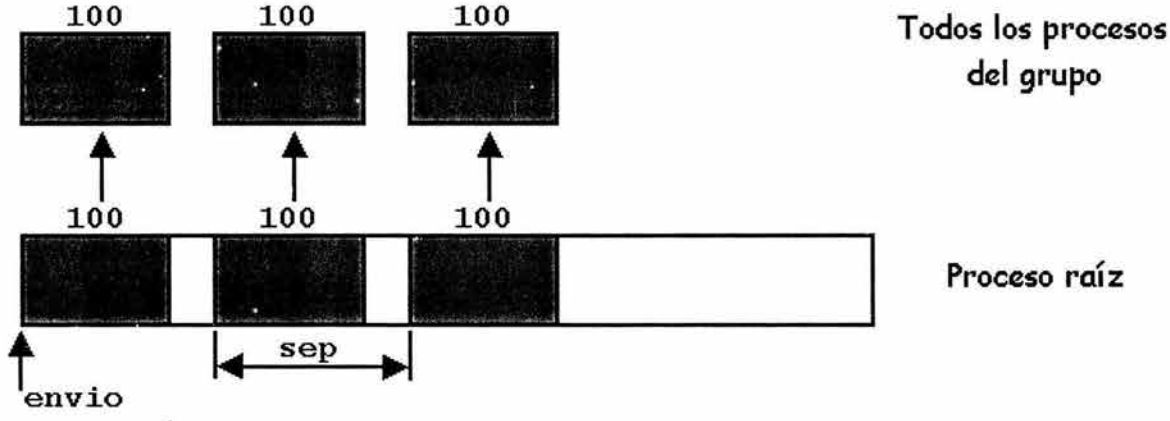


Figura 3.5. El proceso raíz envía **TAM** enteros a cada proceso del grupo, éstos están separados entre sí un número de enteros igual a **sep**.

3.7.7 Recolección de todos los procesos.

La sintaxis de la función **MPI_Allgather** se muestra a continuación.

MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
IN	**sendcount**	Número de elementos en el buffer de envío (entero)

IN	**sendtype**	Tipo de dato de cada elemento del buffer de envío (handle)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice)
IN	**recvcount**	Número de elementos recibidos de un solo proceso (entero)
IN	**recvtype**	Tipo de dato de los elementos del buffer de recepción (handle)
IN	**comm**	Comunicador (handle)

int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

MPI_Allgather puede ser vista como **MPI_Gather**, con la diferencia de que todos los procesos reciben el resultado en lugar de que sólo lo reciba el proceso raíz. El bloque de datos enviado por el j-ésimo proceso es recibido por todos los procesos y colocado en el j-ésimo bloque del buffer de recepción **recvbuf**.

La firma **sendcount**, **sendtype** de un proceso debe ser igual a la firma **recvcount**, **recvtype** de cualquier otro proceso.

El resultado de su aplicación es equivalente al que se obtendría si todos los procesos ejecutaran n llamadas a

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm),

para root = 0 , 1, ..., n-1. Las reglas para el correcto uso de **MPI_Allgather** pueden ser fácilmente inferidas de las reglas correspondientes a **MPI_Gather**.

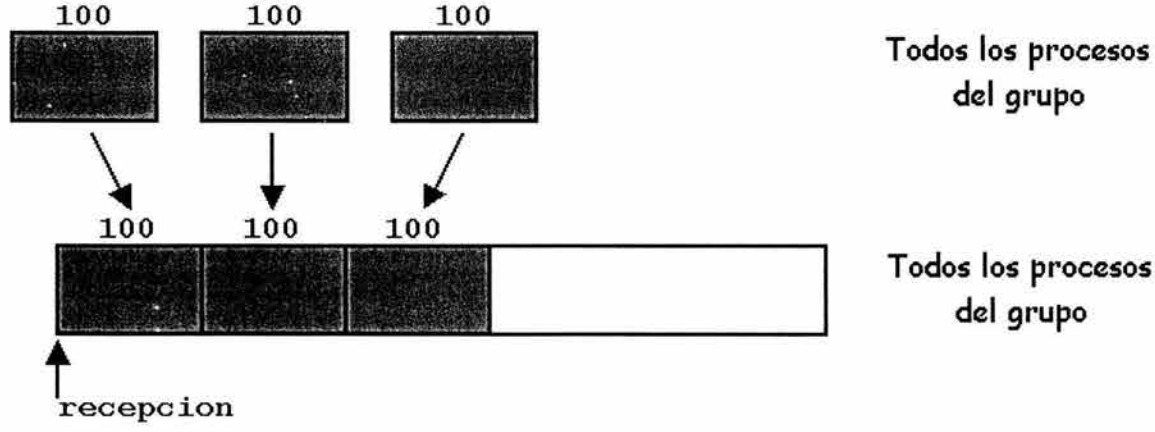


Figura 3.6. Todos los procesos recolectarán 100 enteros de cada uno de los miembros del grupo.

Ejemplo: Todos los procesos recolectarán 100 enteros de cada uno de los miembros del grupo. Ver figura 3.6.

```
MPI_Comm com;
int tamGrupo,envio[100],*recepcion;
...
MPI_Comm_size(comm,&tamGrupo);
recepcion=(int *)malloc(100*tamGrupo*sizeof(int));
MPI_Allgather(envio,100,MPI_INT,recepcion,100,MPI_INT,com);
```

Al igual que con las otras funciones de comunicaciones colectivas también existe una función llamada **MPI_Allgatherv**, su sintaxis se muestra a continuación.

MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
IN	**sendcount**	Número de elementos en el buffer de envío (entero)
IN	**sendtype**	Tipo de dato de cada elemento del buffer de envío (handle)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice)
IN	**recvcounts**	Arreglo de enteros (de tamaño igual al total de elementos del grupo) que contiene el número de elementos que son recibidos por parte de cada proceso
IN	**displs**	Arreglo de enteros (de tamaño igual al total de elementos del grupo). El iésimo elemento especifica el desplazamiento, relativo a **recvbuf**, a partir del que se colocan los datos que envía el proceso iésimo
IN	**recvtype**	Tipo de dato de los elementos del buffer de recepción (handle)
IN	**comm**	Comunicador (handle)

int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)

MPI_Allgatherv puede ser vista como **MPI_Gatherv**, con la diferencia de que todos los procesos reciben el resultado en lugar de que sólo lo reciba el proceso raíz. El bloque de datos enviado por el j-ésimo proceso es recibido por todos los procesos y colocado en el j-ésimo bloque del buffer de recepción **recvbuf**. Dichos bloques no necesitan ser del mismo tamaño como con **MPI_Allgather**.

La firma **sendcount**, **sendtype** del proceso j debe ser igual a la firma **recvcounts[j]**, **recvtype** de cualquier otro proceso.

El resultado de su aplicación es equivalente al que se obtendría si todos los procesos ejecutaran n llamadas a

MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm),

para root = 0 , 1, ..., n-1. Las reglas para el correcto uso de **MPI_Allgatherv** pueden ser fácilmente inferidas de las reglas correspondientes a **MPI_Gatherv**.

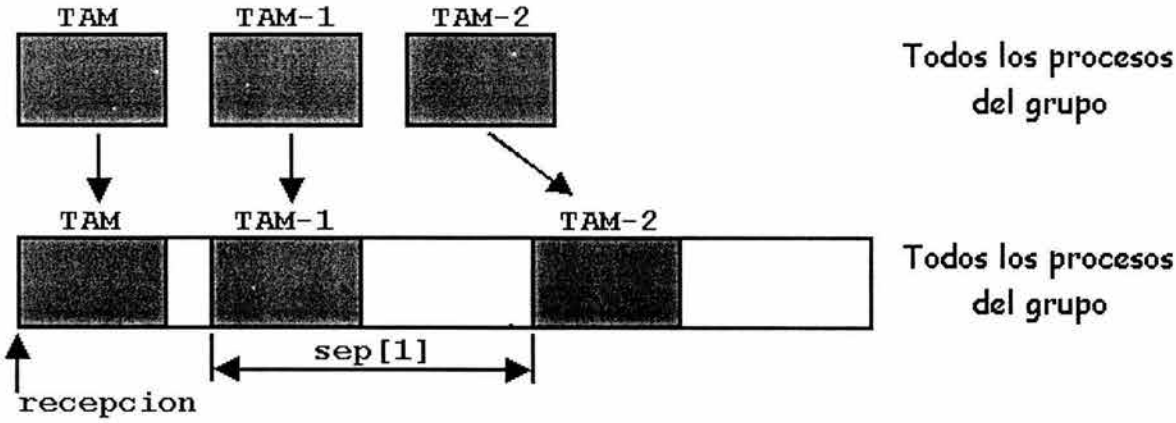


Figura 3.7. La función MPI_Allgatherv.

Ejemplo: El proceso i-ésimo enviará **TAM-i** enteros a todos los miembros del grupo y éstos serán colocados en el buffer de recepción separados entre sí un número de enteros igual a **sep[i]**. Es necesario mencionar que el programa será incorrecto si **sep[i] < TAM-i**. Ver figura 3.7.

```
#define TAM 25
...
MPI_Comm com;
int *sep,i;
int rango,tamGrupo;
int *envio,*recepcion,tamRec;
int *desplazamientos,*cantidades,acum;
...
MPI_Comm_rank(com,&rango);
MPI_Comm_size(com,&tamGrupo);
envio=(int *)malloc((TAM-rango)*sizeof(int));
sep=(int *)malloc(tamGrupo*sizeof(int));
...
desplazamientos=(int *)malloc(tamGrupo*sizeof(int));
cantidades=(int *)malloc(tamGrupo*sizeof(int));
for(i=0,acum=0;i<tamGrupo;i++){
     desplazamientos[i]=acum;
     acum+=sep[i];
     cantidades[i]=TAM-i;
}
tamRec=desplazamientos[tamGrupo-1]+cantidades[tamGrupo-1];
```

```
recepcion=(int *)malloc(tamRec*sizeof(int));
MPI_Allgatherv(envio,TAM-rango,MPI_INT,recepcion,
                    cantidades,desplazamientos,MPI_INT,com);
```

3.7.8 Dispersión-recolección (de todos a todos).

La sintaxis de la función **MPI_Alltoall** se muestra a continuación.

MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
IN	**sendcount**	Número de elementos enviados a cada proceso (entero)
IN	**sendtype**	Tipo de dato de cada elemento del buffer de envío (handle)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice)
IN	**recvcount**	Número de elementos recibidos de un solo proceso (entero)
IN	**recvtype**	Tipo de dato de los elementos del buffer de recepción (handle)
IN	**comm**	Comunicador (handle)

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

MPI_Alltoall es una extensión de **MPI_Allgather** para el caso en que cada proceso envía distintos datos a cada receptor. El j-ésimo bloque de datos enviado por el proceso i es recibido por el proceso j y es colocado en el iésimo bloque del buffer de recepción **recvbuf**.

La firma **sendcount**, **sendtype** de un proceso debe ser igual a la firma **recvcount**, **recvtype** de cualquier otro proceso. Esto implica que la cantidad de datos enviados debe ser igual a la cantidad de datos recibidos entre cada par de procesos. Sin embargo, distintos tipos de mapeos de memoria son permitidos entre el emisor y el receptor.

El resultado de su aplicación es equivalente al que se obtendría si cada proceso hubiera ejecutado n operaciones de envío,

MPI_Send(sendbuf+i*sendcount*extent(sendtype), sendcount, sendtype, i, ...),

y los datos fueran recibidos por los procesos mediante n llamadas a

MPI_Recv(recvbuf+i*recvcount*extent(recvtype), recvcount, recvtype, i, ...),

en ambos casos para i = 0 , 1, ..., n-1. Todos los argumentos son significativos para cualquiera de los procesos. El argumento **comm** debe tener idénticos valores en todos los procesos.

Por otra parte, la función MPI_Alltoallv posee la siguiente sintaxis.
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
IN	**sendcounts**	Arreglo de enteros (de tamaño igual al total de elementos del grupo) que contiene el número de elementos que son enviados a cada proceso
IN	**sdispls**	Arreglo de enteros (de tamaño igual al total de elementos del grupo). El j-ésimo elemento especifica el desplazamiento, relativo a **sendbuf**, a partir del que se toman los datos que se envían al proceso j-ésimo
IN	**sendtype**	Tipo de dato de cada elemento del buffer de envío (handle)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice)
IN	**recvcounts**	Arreglo de enteros (de tamaño igual al total de elementos del grupo) que contiene el número de elementos que son recibidos por parte de cada proceso
IN	**rdispls**	Arreglo de enteros (de tamaño igual al total de elementos del grupo). El iésimo elemento especifica el desplazamiento, relativo a **recvbuf**, a partir del que se colocan los datos que envía el proceso iésimo
IN	**recvtype**	Tipo de dato de los elementos del buffer de recepción (handle)
IN	**comm**	Comunicador (handle)

int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

MPI_Alltoallv agrega flexibilidad a **MPI_Alltoall** ya que permite especificar las localidades que serán leídas durante el envío de datos mediante el empleo del argumento **sdispls**, además de que las localidades donde se colocarán los datos recibidos se definen por medio del argumento **rdispls**.

El j-ésimo bloque de datos enviado por el proceso i es recibido por el proceso j y es colocado en el iésimo bloque del buffer de recepción **recvbuf**. Dichos bloques no necesariamente son del mismo tamaño.

La firma **sendcount[j]**, **sendtype** del proceso i debe ser igual a la firma **recvcount[i]**, **recvtype** del proceso j. Esto implica que la cantidad de datos enviados debe ser igual a la cantidad de datos recibidos entre cada par de

procesos. Sin embargo, distintos tipos de mapeos de memoria son permitidos entre el emisor y el receptor.

El resultado de su aplicación es equivalente al que se obtendría si cada proceso hubiera ejecutado n operaciones de envío,

MPI_Send(sendbuf+sdispls[i]*extent(sendtype), sendcounts[i], sendtype, i, ...),

y los datos fueran recibidos por los procesos mediante n llamadas a

MPI_Recv(recvbuf+rdispls[i]*extent(recvtype), recvcounts[i], recvtype, i, ...),

en ambos casos para i = 0 , 1, ..., n-1. Todos los argumentos son significativos para cualquiera de los procesos. El argumento **comm** debe tener idénticos valores en todos los procesos.

3.7.9 Operaciones globales de reducción.

Las funciones de esta sección realizan operaciones globales de reducción (como suma, máximo, *and* lógico, etc.) a lo largo de todos los miembros de un grupo. La operación de reducción puede ser seleccionada de la lista predefinida de operaciones o puede ser creada por el propio usuario.

Operación de reducción	Comportamiento
Reducción	Regresa el resultado de la operación a un solo proceso llamado raíz.
Reducción para todos	Regresa el resultado de la operación a todos los procesos del grupo.
Reducción-Dispersión	Combina la funcionalidad de una operación reducción con la de una operación de dispersión.
Exploración	Cada miembro del grupo recibe la reducción de los valores contenidos en el buffer de envío de los miembros con rango menor o igual al suyo.

Tabla 3.2. Descripción de los tipos de operaciones globales de reducción.

Las operaciones globales de reducción tienen cuatro presentaciones cuya funcionalidad se resume en la tabla 3.2.

3.7.9.1 Reducción.

La sintaxis de la función **MPI_Reduce** es la siguiente:

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice, significativo sólo para el proceso raíz)
IN	**count**	Número de elementos en el buffer de envío (entero)
IN	**datatype**	Tipo de dato de cada elemento del buffer de envío (handle)
IN	**op**	Operación de reducción (handle)
IN	**root**	Identificador del proceso raíz (entero)
IN	**comm**	Comunicador (handle)

int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

MPI_Reduce combina los elementos provistos en el buffer de entrada de cada proceso en el grupo, usando la operación **op**, y regresa el valor combinado en el buffer de salida del proceso con rango igual a **root**. El buffer de entrada está definido por los argumentos **sendbuf**, **count** y **datatype**; el buffer de salida está definido por los argumentos **recvbuf**, **count** y **datatype**; ambos tienen el mismo número de elementos, con el mismo tipo de datos. La rutina es llamada por los miembros del grupo usando los mismos argumentos para **count**, **datatype**, **root** y **comm**. De esta forma, todos los procesos proveen buffers de entrada y salida del mismo tamaño y con elementos del mismo tipo. Cada proceso puede proveer un elemento o una secuencia de elementos, en este último caso la operación es efectuada componente a componente entre posiciones iguales de los buffers de entrada de los procesos. Por ejemplo, si la operación utilizada es **MPI_MAX** y el buffer de envío contiene dos elementos que son números de punto flotante (**count** = 2 y **datatype** = **MPI_FLOAT**), entonces, después de ejecutarse la función de reducción, la variable contenida en la dirección apuntada por **recvbuf** tendrá el máximo valor de entre todos los elementos **sendbuf(0)** de los procesos del grupo; de igual forma, **recvbuf+1** apuntará al máximo valor de entre todos los elementos **sendbuf(1)** de los procesos del grupo.

La operación **op** debe ser asociativa. Todas las operaciones predefinidas son además conmutativas. Las operaciones definidas por el usuario deben ser asociativas pero no necesariamente deben ser conmutativas. El orden de evaluación de las operaciones de reducción está determinado por el rango de los procesos del grupo, sin embargo, las distintas implementaciones de MPI pueden tomar ventaja de la asociatividad o de la conmutatividad para cambiar dicho orden. Por lo anterior, es responsabilidad del usuario la creación de nuevas operaciones pues el resultado de una función de reducción puede cambiar si la operación que utiliza no es estrictamente asociativa o conmutativa.

3.7.9.2 Operaciones de reducción predefinidas.

En la tabla 3.3, se listan las operaciones predefinidas que pueden utilizarse como el argumento **op** de los cuatro tipos de funciones globales de reducción descritos anteriormente (véase tabla 3.2).

Nombre de la operación	Función realizada
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
MPI_LAND	And lógico
MPI_BAND	And a nivel de bits
MPI_LOR	Or lógico
MPI_BOR	Or a nivel de bits
MPI_LXOR	Xor lógico
MPI_BXOR	Xor a nivel de bits
MPI_MAXLOC	Máximo y su posición
MPI_MINLOC	Mínimo y su posición

Tabla 3.3. Operaciones predefinidas por la MPI, para utilizarse con las funciones globales de reducción.

Las operaciones **MPI_MINLOC** y **MPI_MAXLOC** se describen posteriormente. Para las otras operaciones predefinidas, enumeramos a continuación las combinaciones permitidas para los argumentos **op** y **datatype**. Primero, definiremos grupos para los tipos básicos de MPI de la siguiente forma: Entero = {**MPI_INT**, **MPI_LONG**, **MPI_SHORT**, **MPI_UNSIGNED_SHORT**, **MPI_UNSIGNED**, **MPI_UNSIGNED_LONG**}; Punto flotante = {**MPI_FLOAT**, **MPI_DOUBLE**, **MPI_LONG_DOUBLE**}; Byte = {**MPI_BYTE**}. Ahora, los tipos de datos básicos para cada operación se muestran en la tabla 3.4.

Op	Tipos permitidos
MPI_MAX, MPI_MIN	Entero, Punto flotante
MPI_SUM, MPI_PROD	Entero, Punto flotante
MPI_LAND, MPI_LORD, MPI_LXOR	Entero
MPI_BAND, MPI_BOR, MPI_BXOR	Entero, Byte

Tabla 3.4. Tipos de datos permitidos para las distintas operaciones predefinidas.
Ejemplo: Cálculo del producto punto de dos vectores que están distribuidos a lo largo de los miembros de un grupo de procesos, el resultado será recibido por el proceso raíz.

```
#define TAM 500
```

```
...
MPI_Comm com;
float v1[TAM],v2[TAM],*prodPun,sumParcial;
int i,raiz;
/* Los vectores se distribuyen entre los diferentes procesos.
 */
...
for(i=0,sumParcial=0.;i<TAM;i++)
     sumParcial+=v1[i]*v2[i];
MPI_Reduce(&sumParcial,prodPun,1,MPI_FLOAT,MPI_SUM,raiz,com);
```

Las operaciones MINLOC y MAXLOC.

El operador **MPI_MINLOC** obtiene el mínimo global así como el índice asociado a éste. **MPI_MAXLOC**, de manera similar, obtiene el máximo global y su índice.

La operación que define a **MPI_MAXLOC** es:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

donde

$$w = máx(u, v)$$

y

$$k = \begin{cases} i & si\ u > v \\ mín(i, j) & si\ u = v \\ j & si\ u < v \end{cases}$$

La operación **MPI_MINLOC** se define de manera similar:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

donde

$$w = mín(u, v)$$

y

$$k = \begin{cases} i & si\ u < v \\ mín(i, j) & si\ u = v \\ j & si\ u > v \end{cases}$$

Además, ambas operaciones son asociativas y conmutativas.

Hay que notar que si la operación **MPI_MAXLOC** es aplicada a una secuencia de parejas $(u_0, 0)$, $(u_1, 1)$, ..., $(u_{n-1}, n-1)$, entonces el valor de retorno es

(u, r), donde u = $máx_i u_i$ y r es el índice del primer máximo global de la secuencia. Así, una aplicación de esta operación podría ser la de encontrar el máximo global y el rango del proceso que contiene dicho valor. Si cada proceso suministra un valor y su rango dentro del grupo, entonces la función de reducción con **op** = **MPI_MAXLOC** regresará el valor máximo y el rango del primer proceso del grupo con ese valor. Similarmente, **MPI_MINLOC** puede ser usada para encontrar el mínimo global y su índice. Nótese que el anterior es sólo un ejemplo pues el índice de las parejas no necesariamente tiene que estar relacionado con el rango del proceso e incluso éste puede ser negativo.

Estas dos operaciones están definidas para trabajar sobre operandos que consisten en un par: valor e índice. Para utilizar **MPI_MINLOC** y **MPI_MAXLOC** en una función de reducción, uno debe proporcionar un argumento **datatype** que represente a este par. MPI posee varios tipos de datos predefinidos para tal efecto, las operaciones **MPI_MINLOC** y **MPI_MAXLOC** pueden ser utilizadas con cualquiera de los tipos descritos en la tabla 3.5

Nombre del tipo de dato	Descripción
MPI_FLOAT_INT	float e int
MPI_DOUBLE_INT	double e int
MPI_LONG_ING	long e int
MPI_2INT	int e int
MPI_SHORT_INT	short e int
MPI_LONG_DOUBLE_INT	long double e int

Tabla 3.5. Tipos de datos predefinidos para ser utilizados con las operaciones **MPI_MINLOC** y **MPI_MAXLOC.**

Ejemplo: Se posee una secuencia de valores distribuida en arreglos entre los miembros de un grupo. Encontrar el máximo valor y el rango del proceso que lo contiene.

```
struct{
     double val;
     int indice;
}maxLocal,maxGlobal;
double *valor;
int rango,raiz,cantidad;
...
valor=(double *)malloc(cantidad*sizeof(double));
...
/* Cálculo del máximo local y su índice. */
for(i=1,maxLocal.val=valor[0];i<cantidad;i++)
     if(maxLocal<valor[i])
          maxLocal.val=valor[i];
MPI_Comm_rank(MPI_COMM_WORLD,&rango);
maxLocal.indice=rango;
```

```
/* Cálculo del máximo global y su índice. */
MPI_Reduce(&maxLocal,&maxGlobal,1,MPI_DOUBLE_INT,
                                MPI_MAXLOC,raiz,MPI_COMM_WORLD);
```

3.7.9.3 Operaciones definidas por el usuario.

La sintaxis de la función **MPI_Op_create** es la siguiente:

MPI_Op_create(function, commute, op)

IN	**function**	Función definida por el usuario (function)
IN	**commute**	true si es conmutativa; false en otro caso
OUT	**op**	Operación (handle)

int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)

MPI_Op_create vincula una operación global definida por el usuario con un manejador **op** que puede ser utilizado posteriormente en cualquiera de los cuatro tipos de funciones globales de reducción de la tabla 3.2. Si **commute** = true, entonces la operación debe ser conmutativa y asociativa. Si **commute** = false, entonces el orden de los operandos se fija y define para que sea ascendente, establecido con respecto al rango de los procesos del grupo y comenzando por el proceso cero. El orden de evaluación puede ser cambiado, tomando ventaja de la asociatividad de la operación. Si **commute** = true entonces el orden de evaluación puede ser cambiado, tomando ventaja de la conmutatividad y de la asociatividad.

El argumento **function** es la función definida por el usuario, ésta debe tener los siguientes argumentos: **invec**, **inoutvec**, **len** y **datatype**.

El prototipo, en ANSI-C, de la función definida por el usuario es el siguiente:

typedef void MPI_User_function(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype);

La función de reducción definida por el usuario debe ser escrita de tal forma que cumpla con lo siguiente: sean **u[0]**, ..., **u[len-1]** los **len** elementos en el buffer de comunicación descrito por los argumentos **invec**, **len** y **datatype** en el momento en que se invoca la función; sean **v[0]**, ..., **v[len-1]** los **len** elementos del buffer de comunicación descrito por **inoutvec**, **len** y **datatype** en el momento en que se invoca la función; sean **w[0]**, ..., **w[len-1]** los **len** elementos del buffer de comunicación descrito por **inoutvec**, **len** y **datatype** una vez que la función ha retornado; entonces **w[i] = u[i]** $\circ$ **v[i]**, para **i = 0**, ..., **len-1**, donde $\circ$ es la operación de reducción que la función realiza.

Tipos de datos generales pueden pasarse a la función definida por el usuario, sin embargo, el uso de tipos de datos que no son contiguos es probable que conduzca a ineficiencias en la ejecución de la operación de reducción.

Ninguna función de comunicación de MPI puede ser llamada dentro de una función definida por el usuario. **MPI_Abort** será llamada dentro de la función en caso de que ocurra un error.

Del mismo modo que existe una función para crear operaciones de reducción, existe también una función para liberarlas, su sintaxis se presenta a continuación:

MPI_Op_free(op)

INOUT **op** Operación (handle)

int MPI_Op_free(MPI_Op *op)

Esta función marca a una operación de reducción definida por el usuario para que sea liberada y asigna a **op** el valor de **MPI_OP_NULL**.

Ejemplo: Este ejemplo utiliza una operación definida por el usuario para poderle calcular una norma vectorial a cada uno de los **TAM** vectores distribuidos a lo largo de los procesos de un grupo. La norma que en este caso se utilizará está definida para un determinado vector $\bar{x}$ de P componentes como sigue:

$$N(\bar{x}) = \sum_{i=0}^{P-1} |x_i|$$

```
#define TAM 10
...
void normaVectorial(double*,double*,int*,MPI_Datatype*);
...
int raiz;
double vec[TAM],normas[TAM];
MPI_Op miOper;
...
MPI_Op_create(normaVectorial,1,&miOper);
...
MPI_Reduce(vec,normas,TAM,MPI_DOUBLE,
                                  miOper,raiz,MPI_COMM_WORLD);
...
MPI_Op_free(&miOper);
...
```

```
void normaVectorial(double *in,double *inout,
                            int *len,MPI_Datatype *datatype){
     int i;
     for(i=0;i<*len;i++,inout++,in++)
          *inout=fabs(*inout)+fabs(*in);
}
```

3.7.9.4 Reducción para todos.

MPI incluye algunas variantes de la operación **MPI_Reduce**, una de ellas es **MPI_Allreduce** cuya sintaxis es la siguiente.

MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice)
IN	**count**	Número de elementos en el buffer de envío (entero)
IN	**datatype**	Tipo de dato de cada elemento del buffer de envío (handle)
IN	**op**	Operación de reducción (handle)
IN	**comm**	Comunicador (handle)

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_Allreduce realiza casi lo mismo que **MPI_Reduce** con la diferencia que el resultado aparece en el buffer de recepción de todos los miembros del grupo en lugar de sólo aparecer en el proceso raíz.

La función **MPI_Allreduce** puede ser implementada por medio de una función **MPI_Reduce** seguida de una función **MPI_Bcast**, sin embargo, la primera está optimizada y, por lo tanto, su ejecución es mucho mejor a la obtenida por medio de la reducción seguida de la transmisión.

3.7.10 Reducción-dispersión.

Otra variante de **MPI_Reduce** es **MPI_Reduce_scatter**, esta función distribuye el resultado obtenido a lo largo de todos los procesos del grupo.

MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice)

IN	**recvcounts**	Arreglo de enteros que especifica el número de elementos del resultado que se distribuye a cada proceso. Este arreglo debe ser idéntico en todos los procesos que ejecutan la llamada a esta función.
IN	**datatype**	Tipo de dato de cada elemento del buffer de entrada (handle)
IN	**op**	Operación de reducción (handle)
IN	**comm**	Comunicador (handle)

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_Reduce_scatter primero efectúa una reducción componente a componente sobre el vector de **count = Σ_i recvcounts[i]** elementos del buffer de entrada definido por **sendbuf**, **count** y **datatype**. Después, el vector resultado es dividido en **n** segmentos, donde **n** es el número de miembros del grupo. El segmento **i** contiene **recvcounts[i]** elementos. El iésimo segmento se envía al proceso **i** y se almacena en el buffer de recepción definido por **recvbuf**, **recvcounts[i]** y **datatype**.

La rutina **MPI_Reduce_scatter** es funcionalmente equivalente a: una operación **MPI_Reduce** con **count** igual a la suma de **recvcounts[i]** seguida de **MPI_Scatterv** con **sendcounts** igual a **recvcounts**. Sin embargo, la implementación directa tendrá una mejor ejecución que la anterior combinación.

3.7.11 Exploración.

MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)

IN	**sendbuf**	Dirección inicial del buffer de envío (choice)
OUT	**recvbuf**	Dirección inicial del buffer de recepción (choice)
IN	**count**	Número de elementos en el buffer de entrada (entero)
IN	**datatype**	Tipo de dato de cada elemento del buffer de entrada (handle)
IN	**op**	Operación de reducción (handle)
IN	**comm**	Comunicador (handle)

int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_Scan se usa para efectuar una reducción de los datos distribuidos a través del grupo con respecto al índice de cada proceso. Esta función regresa, en el buffer de recepción del proceso con rango igual a **i**, la reducción de los valores contenidos en el buffer de envío de los procesos con rango igual a **0, ..., i**. El tipo de operaciones soportadas, su semántica y las restricciones de los buffers de envío y recepción son iguales que para la función **MPI_Reduce**.

3.8 Compilación y ejecución de programas con MPICH.

Para la creación de aplicaciones paralelas, en esta tesis, utilizamos MPICH –una de las distribuciones que tiene la MPI estándar–. En esta sección vermos cómo se compilan y ejecutan los programas con el ambiente de MPICH; para más detalles acerca del ambiente de trabajo sobre el cluster, puede consultarse el apéndice A.

Los programas que utilizan las librerías de MPI deben ser compilados antes de poder ser ejecutados, el comando que permite compilar programas de MPI con base en el lenguaje C se llama **mpicc**, mientras que para Fortran 77 se llama **mpif77**.

La sintaxis completa de **mpicc** puede ser consultada en las páginas del manual del comando, enseguida mostramos una versión simplificada, acorde a nuestras necesidades.

```
mpicc -o <nombre del programa de salida>  <nombre del programa en C>
```

Después, para ejecutar el programa de salida que se obtuvo mediante **mpicc**, podemos utilizar el guión de comandos **mpirun**, su sintaxis completa puede consultarse en las páginas del manual del comando, aquí damos una simplificación.

```
mpirun [opciones_de_mpirun...] <nombre del programa> [argumentos...]
```

Opciones de mpirun.

```
-machinefile <archivo con nombre de las máquinas>
      -Toma la lista de máquinas, en las que se deberá ejecutar el
      programa, del archivo <archivo con nombre de las máquinas>.
      <archivo con nombre de las máquinas> es un archivo de texto
      que debe contener el nombre de las máquinas, una por renglón,
      en las que el programa ha de ejecutarse.
-np <número de procesos>
      -Especifica el número de procesos a ejecutar. Si <número de
      procesos> excede al total de procesadores del sistema, los
      procesos se ejecutan de forma concurrente.
-nolocal
      -No ejecuta procesos en la máquina local.
```

4. Técnicas básicas de programación paralela.

4.1 Divide y Conquista

La técnica de programación paralela Divide y Conquista tiene un principio muy simple: dividir un problema en subproblemas, resolver cada uno de manera independiente, y mediante la solución a cada subproblema se puede conseguir la solución generalizada. Este esquema es sencillo y adecuado para convertir un algoritmo secuencial en un algoritmo paralelo; además, se puede considerar que todos los programas paralelos siguen este diseño.

En programación secuencial, esta técnica es ampliamente empleada, ya sea para estructurar el código, como para el análisis del problema y para lograr una programación modular. La ventaja que se tiene al implementar esta técnica en sistemas paralelos es que cada uno de los subproblemas puede ser distribuido a cada uno de los procesos, teniendo así un proceso para cada subproblema, lo cual disminuye el tiempo de obtención de resultados.

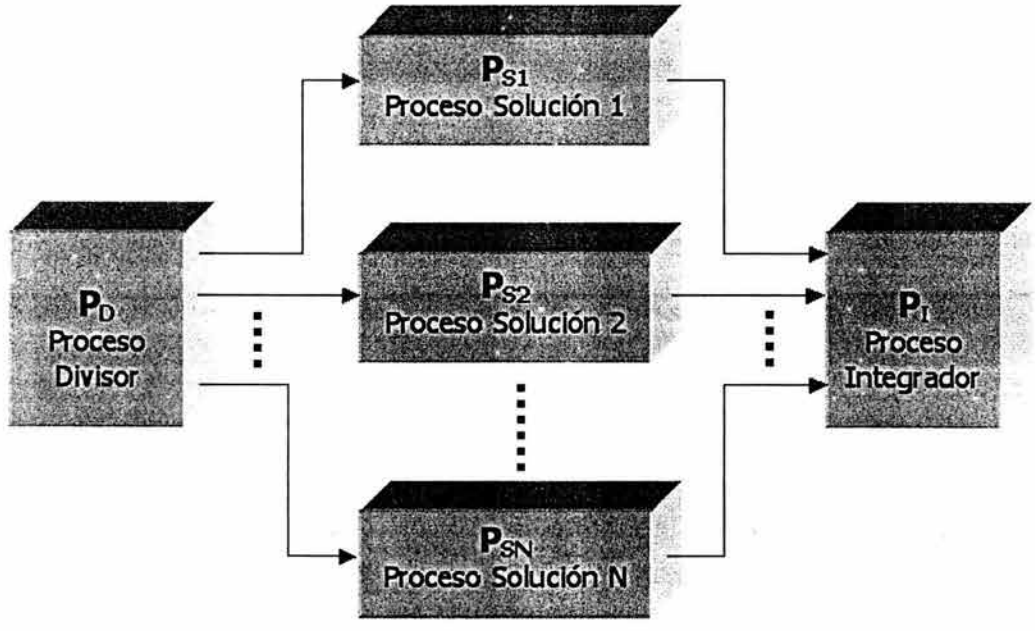


Figura 4.1. Diagrama característico de la técnica Divide y Conquista.

Para la solución paralela de problemas mediante esta técnica se debe tener en cuenta que deben existir procesos tanto para resolver cada subproblema como para realizar el particionamiento de tareas y de datos, y también para la integración de las soluciones parciales en una solución final. En ocasiones, los

subproblemas generados del problema original son pequeñas réplicas de éste, esto puede conducir a una solución recursiva del problema.

Descripción:

La manera en que este algoritmo realiza su trabajo se puede representar mediante el esquema de la figura 4.1.

Como se puede apreciar en la figura 4.1, el Proceso Divisor es el encargado de generar y distribuir las tareas, los Procesos Solución son aquellos encargados de procesar cierta información y generar resultados parciales que irán hacia el Proceso Integrador el cual conjuntará el trabajos de los demás y se proporcionará un resultado final.

Para detallar las actividades que realiza cada tipo de proceso nos enfocarnos en ellos de manera individual, una vez entendidas las actividades que desempeñarán, estudiaremos la forma en que se da la cooperación y la comunicación entre dichos procesos.

Proceso Divisor P_D

Las actividades que realiza el Proceso Divisor son:

- El proceso divisor se encargará de determinar la distribución de los datos entre los diferentes procesos solución.
- Este proceso se encarga además de enviar los datos correspondientes a los Procesos Solución o, en su defecto, les indicará dónde obtenerlos.
- Adicionamente, podrá transmitir diversas señales (inicio del cómputo, espera, paro, etc.) a los Procesos Solución, con el objeto de coordinarlos.

El envío de datos es lo más común en la paralelización de procesos pero, en muchas ocasiones, el proceso Divisor sólo indica el punto en que debe comenzar el cómputo de cada proceso, sin la necesidad de transmitir alguna información adicional.

De manera general, no hay comunicación entre los procesos solución, ya que es conveniente que los procesos estén encargados de resolver problemas independientes entre sí.

La forma en que se da la comunicación entre procesos, y el tipo de tareas que realiza cada uno de ellos, puede identificarse con la estructura MIMD (Multiple Instruction-Multiple Data) propuesta por Flynn, por la variedad de formas que pueden tener las actividades de cada proceso y del problema en su conjunto.

Proceso solución P_{Sx}

Las actividades que realiza cada Proceso Solución son:

- Recepción de los datos a procesar o de la ubicación de éstos (proviene del Proceso Divisor).
- Procesamiento de los datos y generación de una solución parcial.
- Envío de resultados hacia el Proceso Integrador.

Los datos que llegan a cada Proceso Solución son leídos, una vez obtenidos, se realizan sobre ellos las operaciones necesarias para generar un resultado, éste último será enviado al Proceso Integrador para que reúna todos los resultados parciales y consiga el resultado final.

Proceso Integrador P_I

Las actividades que realiza el Proceso Integrador son:

- Recepción de resultados de cada Proceso Solución.
- Procesamiento final y entrega de resultados.

Antes de comenzar a pensar en la codificación del algoritmo en un lenguaje de programación, es importante tener en cuenta los tiempos en que la comunicación de procesos se va a dar.

Cada problema puede tener un esquema de comunicaciones diferente e incluso pueden haber problemas en los que se necesite que todos los procesos se comuniquen con más de uno en particular y en varias ocasiones a lo largo del procesamiento de información.

Tomando en cuenta las consideraciones de independencia de los subproblemas especificadas anteriormente, se plantea el siguiente esquema, mostrado en la figura 4.2, en el cual se aprecian los tiempos de duración o ciclo de vida de cada uno de los procesos así como la forma en que deben comunicarse.

En el esquema de la figura 4.2, la línea punteada separa las fases de envío y recepción de datos de la fase de procesamiento de datos en cada uno de los procesos. De igual forma, se puede ver cómo el Proceso Divisor termina su ciclo de vida cuando finaliza su comunicación con los Procesos Solución; los Procesos Solución colaboran con procesar su información y enviarla al Proceso Integrador, y el Proceso Integrador interviene sólo hasta que los Procesos Solución comienzan a enviarle resultados, cuando termina de recibirlos, suministra la solución total y concluye su ciclo de vida.

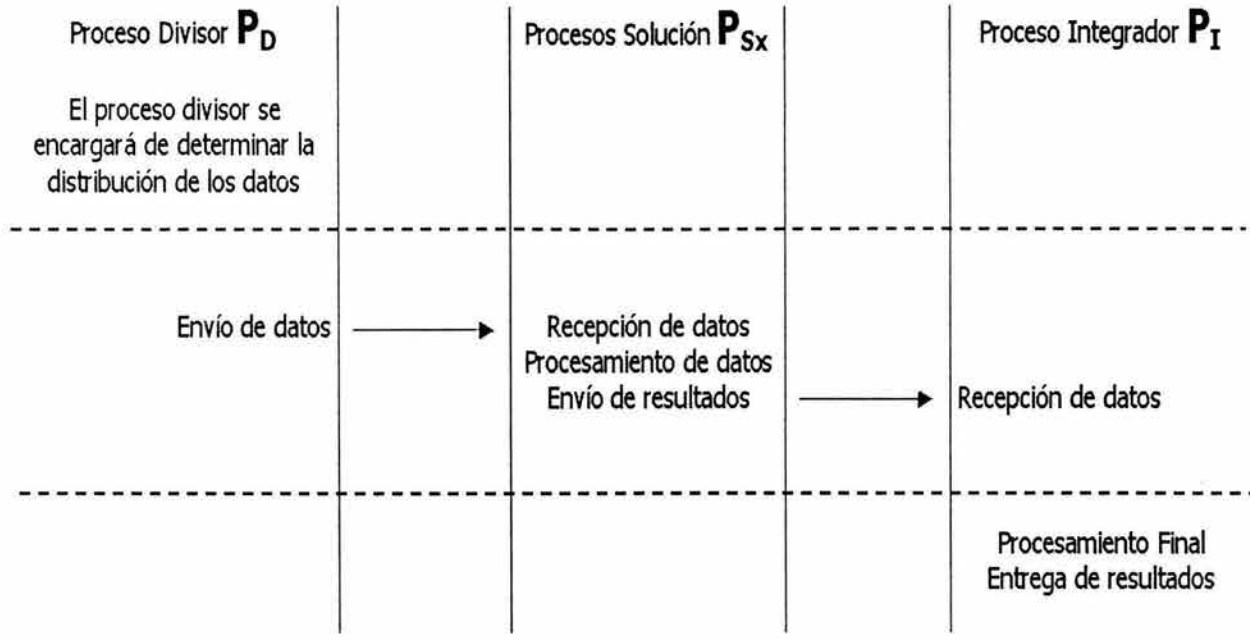


Figura 4.2. Diagrama de comunicación entre procesos para la técnica Divide y Conquista.

4.2 Entubamiento de Datos (Data Pipelining).

Esta técnica permite realizar una división del trabajo en distintas funciones, es un paralelismo en el que se busca encontrar la forma de descomponer el problema en fases, buscando las diferentes tareas del algoritmo que se pueden ejecutar de manera concurrente de tal forma que cada procesador efectúe una parte del algoritmo total.

Podemos comparar la técnica con el cauce de una tubería en la que cada proceso realiza su función y pasa sus resultados al siguiente proceso en la línea. Aunque esta técnica parece tener un carácter secuencial, el procesamiento paralelo puede observarse una vez que el cauce de la tubería se encuentra lleno, ya que de forma simultánea se puede estar recibiendo datos de entrada y produciendo resultados a la salida; pues todos los procesos que conforman la tubería trabajan en paralelo.

Al igual que en una línea de producción, una vez que un proceso terminó su función, el siguiente proceso en la línea se encarga de continuar el trabajo de su antecesor y así hasta dar un resultado final.

Descripción

El esquema que representa este tipo de procesamiento paralelo se muestra en la figura 4.3.

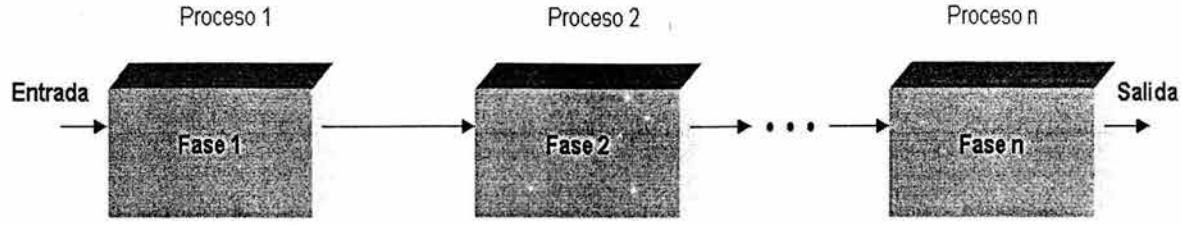


Figura 4.3. Diagrama característico de la técnica de Entubamiento de Datos.

En la figura 4.3 podemos observar que cada proceso corresponde a una parte de la tubería y es responsable de una tarea particular. El flujo de los datos va de una fase a otra y la comunicación se lleva a cabo de manera asíncrona. La eficiencia de esta técnica depende de la distribución de las cargas de manera que cada una de ellas se encuentre balanceada dentro de las fases de la tubería.

La descripción de cada uno de los procesos es la siguiente:

Proceso 1 P_1

Las actividades que realiza el Proceso 1 son:

- Este proceso se encarga de recibir los datos iniciales del problema que aún no se han procesado, es decir, obtiene los datos de la fuente primaria.
- Posteriormente, procesa un grupo de datos de entrada.
- Una vez procesados los datos, se encarga de enviarlos a su proceso sucesor en la línea para que continúen en la siguiente fase del algoritmo.
- Repite continuamente el anterior procedimiento hasta que se terminen los datos.

Proceso X P_x X=2,...,n-1

Las actividades que realiza el Proceso X son:

- Recibir datos de su proceso antecesor.
- Procesar los datos realizando la tarea que le corresponda.
- Enviar datos ya procesados a su sucesor.
- Este procedimento es repetido hasta que deje de recibir datos de su proceso predecesor.

Proceso n P_n

Las actividades que realiza el Proceso n son:

- Recibe los datos procesados que le envía el proceso n-1.
- Procesa los datos.
- Por ser el último en la línea de procesamiento, una vez que ha procesado su información, va generando la solución total.
- Repite este procedimento hasta que deje de recibir datos del proceso n-1, con los últimos datos recibidos computa el resutado final.

El tiempo aproximado de entrega de resultados por parte del algoritmo de Entubamiento de Datos puede estimarse con el tiempo de ejecución de la tarea más tardada dentro de la línea de procesos, ya que esta tarea será la que aporte un mayor tiempo de procesamiento y, por lo tanto, los procesos que se encuentren delante del proceso más tardado deberán esperar a que éste termine su trabajo para poder tomar su salida y utilizarla.

Para entender esto, podemos pensar en el siguiente ejemplo: supóngase un problema resuelto mediante Entubamiento de Datos, el cual consta de tres procesos, los cuales tienen duraciones de 1, 3 y 1 unidades de tiempo respectivamente, tal y como se muestra en la figura 4.4

Figura 4.4. Utilización de la técnica de Entubamiento de Datos en la resolución de un problema que consta de tres fases.

Al iniciarse la ejecución, el primer resultado tardará en generarse 5 segundos (1 + 3 + 1 segundos), pero el siguiente resultado se generará ya no dentro de otros 5 segundos sino que tardará tan solo 3 segundos –tiempo que tarda en ser realizada la tarea más compleja– ya que el cauce de datos de la línea ya tiene datos procesados. El diagrama de tiempos asociado a este problema podemos visualizarlo en la figura 4.5.

Tiempo [s]	Proceso 1		Proceso 2		Proceso 3	Entrega de resultados
1	Datos (1)					
2	Datos (2)		Datos (1)			
3			Datos (1)			
4			Datos (1)			
5	Datos (3)		Datos (2)		Datos (1)	Resultado (1)
6	⋮		Datos (2)			
7			Datos (2)			
8			Datos (3)		Datos (2)	Resultado (2)
9			Datos (3)			
10			Datos (3)			
11			⋮		Datos (3)	Resultado (3)
12					⋮	⋮
13						

Figura 4.5. Diagrama de tiempos para un problema que consta de tres fases con duraciones de 1, 3 y 1 segundos, respectivamente.

Aquí podemos apreciar que cada proceso realiza su trabajo en su tiempo correspondiente y que la entrega de resultados se hace de manera constante cada 3 segundos en toda la ejecución; con excepción del primer resultado que demora un poco más por no estar lleno el cauce, pues es el primer procesamiento dentro de la ejecución.

El diagrama de tiempos en que existe cada proceso es el de la figura 4.6.

Datos de entrada	→	Proceso 1	Proceso 2	…	Proceso n		
	Fase 1	Datos de 1					
	Fase 2		Datos de 2				
	⋮			⋱			
	Fase n				Datos de n	→	Datos de salida
T0		T1	T2	…	Tn		

Figura 4.6. Diagrama de tiempos para la técnica de Entubamiento de Datos.

De manera general, toda la secuencia permanece constante durante la ejecución, de esta forma se puede ver cómo la técnica de Entubamiento de Datos puede beneficiar en gran medida al procesamiento de información. Existe aún una mejora sustancial a esta técnica que incrementa su rendimiento, pero dificulta su implementación, ésta es la de tener varios cauces de datos o tuberías para la solución del problema.

4.3 Maestro-Esclavo.

La técnica Maestro-Esclavo es una manera muy eficiente de dividir el trabajo a realizar entre los diferentes procesos paralelos que contribuyen a la solución de un problema. Como su nombre lo indica, existe un proceso que controla la labor de los otros y es el encargado de dar el efecto de unidad al trabajo que realicen los esclavos.

Descripción:

La manera en que este algoritmo realiza su trabajo se puede representar mediante el esquema de la figura 4.7.

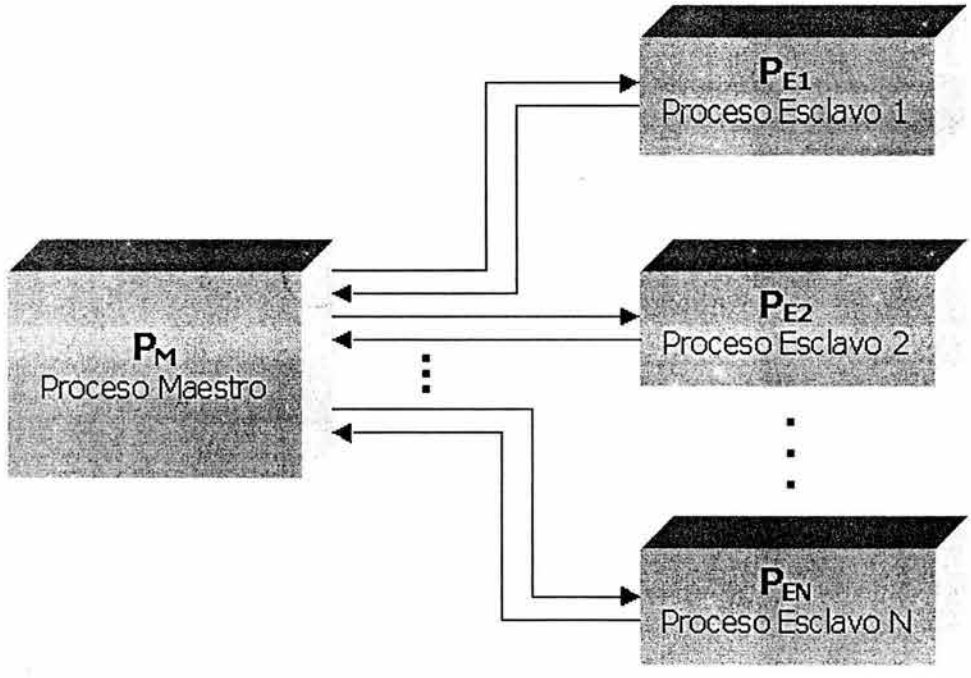


Figura 4.7. Diagrama característico de la técnica Maestro-Esclavo.

En esta figura 4.7 se visualiza que el Proceso Maestro tiene que realizar un envío de información hacia los Esclavos y éstos, al procesar la información obtenida, regresan su resultado al Maestro. Finalmente, cuando el Maestro unifique los resultados, producirá una solución general.

Para poder entender la forma en que se llevan a cabo las actividades de cada proceso podemos enfocarnos en ellos de manera individual, una vez entendidas las actividades que desempeñarán, veremos cómo interactúan al trabajar en paralelo.

Proceso Maestro P_M

Las actividades que realiza el Proceso Maestro son:

- Realizar la distribución de los datos entre los diferentes Procesos Esclavo.
- En caso de requerirse, establece comunicación con los Procesos Esclavo con el afán de coordinarlos.
- Recepción de las soluciones parciales generadas por los Esclavos.
- Procesamiento final y entrega de resultados.

Proceso Esclavo P_{Ex}

Las actividades que realiza cada Proceso Esclavo son:

- Recepción de los datos a procesar o de la ubicación de los mismos, según lo indique el Proceso Maestro.
- Procesamiento de los datos y generación de resultados.
- Envío de la solución parcial obtenida (hacia el Maestro).

El número de Procesos Esclavo depende de la capacidad del sistema y también de la decisión del usuario para la solución del problema; en un esquema generalizado, la realización de pruebas sobre el algorimto es la que habitualmente muestra el número de Procesos Esclavo que deben emplearse. Por otra parte, la manera en que se distribuyen los datos está en función del problema que se desee resolver.

Es importante hacer notar que en este paradigma de programación no hay comunicación entre Procesos Esclavo; pues, estos últimos, sólo se comunican con el Maestro. Además, la forma en que se realiza el envío, procesamiento y recepción de datos se identifica con la estructura SIMD (Single Instruction-Multiple Data) propuesta por Flynn, debido a que todos los Procesos Esclavo realizan la misma actividad (misma secuencia de instrucciones) sobre los datos que reciben (que para cada Esclavo son diferentes).

Una vez que hemos entendido las funciones específicas que desarrollará cada proceso, tenemos que analizar la forma en que éstos se comunicarán, el

ciclo de vida que tendrán, y los tiempos a partir de los cuales pueden actuar sobre los datos. El siguiente diagrama, mostrado en la figura 4.8, representa lo anterior.

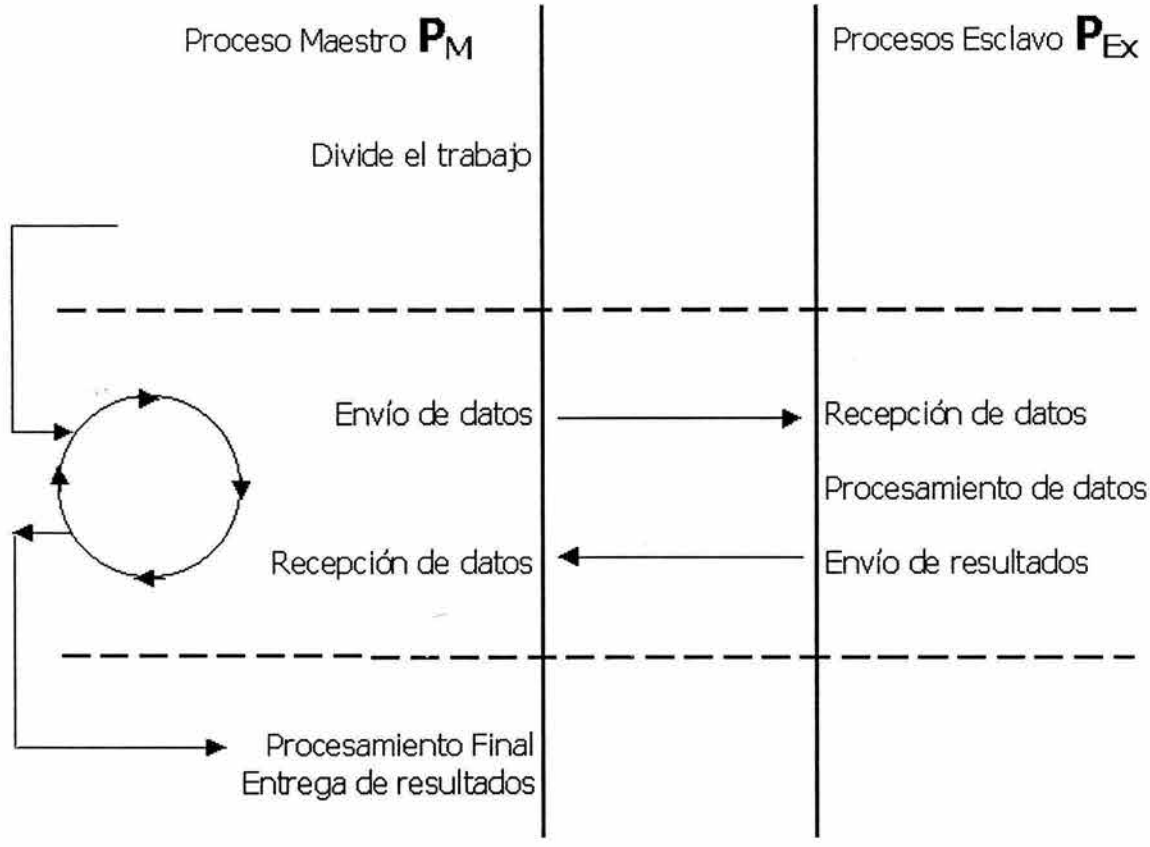


Figura 4.8. Diagrama de comunicación entre procesos para la técnica Maestro-Esclavo.

Si tomamos en consideración que al iniciarse el algoritmo se comienza a ejecutar la parte superior del diagrama (figura 4.8), al principio, el Maestro es el único proceso en realizar cálulos; mientras, los Esclavos esperan a que éste realice la repartición del trabajo.

La verdadera utilidad de los Procesos Esclavo se presenta una vez que el Maestro ha dividido el trabajo, en este momento, pueden comenzar con la parte del procesamiento que les corresponde, generan resultados y los transmiten de vuelta al Maestro. Todos los Procesos Esclavo tienen su campo de acción después de la primer línea punteada, y su tiempo de vida es hasta terminar el envío de sus resultados al Maestro, es decir, hasta antes de la segunda línea punteada.

Del mismo modo, mientras los Esclavos están procesando de manera paralela los datos que reciben, el Maestro sólo realiza la función de esperar por los resultados que devolverá cada uno. Finalmente, después de contar con todos los resultados parciales, el Maestro da solución total al problema.

4.4 SPMD (Single Program-Multiple Data).

En este paradigma de programación, al igual que en los vistos anteriormente, se busca dividir el problema principal en subprogramas que realicen tareas simples. La diferencia fundamental entre esta técnica y las anteriores radica en que los diferentes procesos pueden comunicarse entre ellos, es decir, no necesariamente realizan la tarea asignada con los datos que el proceso divisor les proporciona, sino que pueden comunicarse entre sí durante su ejecución; lo anterior es con la intención de proveer un mayor grado de coordinación entre los procesos participantes.

Descripción:

La manera en que este algoritmo realiza su trabajo se puede representar mediante el esquema de la figura 4.9.

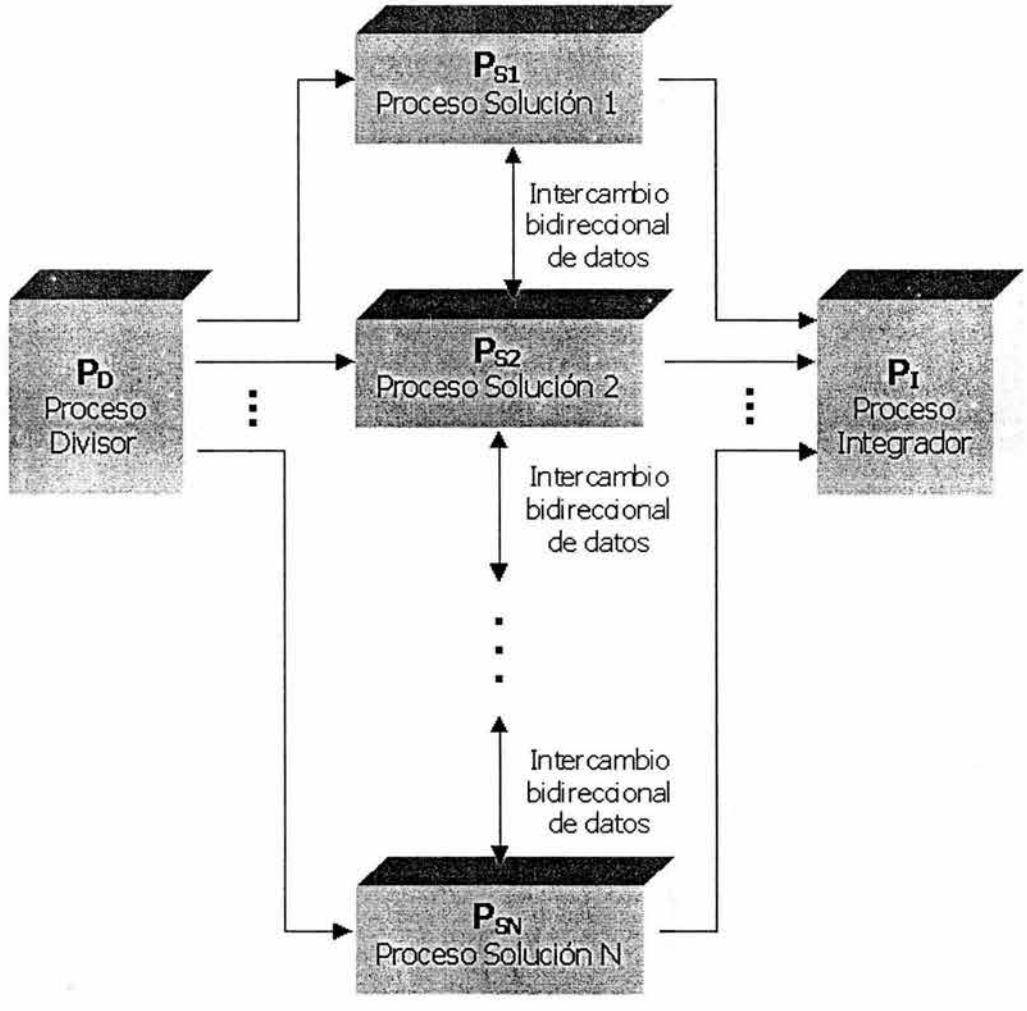


Figura 4.9. Diagrama característico de la técnica SPMD.

De acuerdo con lo mostrado en la figura 4.9, existe un Proceso Divisor que es el encargado distribuir el conjunto de datos entre los diferentes Procesos Solución; cada Proceso Solución trabaja con el conjunto de datos que el Proceso Divisor le proporcionó y puede, de así requerirlo, establecer comunicación con alguno de los otros Procesos Solución que están aún en ejecución para generar una solución parcial. Las soluciones parciales son utilizadas por el Proceso Integrador que se encarga de procesarlas para poder proporcionar una solución total.

De forma más específica, las actividades que realiza cada proceso participante se listan a continuación:

Proceso Divisor P_D

Las actividades que realiza el Proceso Divisor son:

- Tomar el conjunto total de datos involucrados en el problema y determinar la forma en que serán divididos entre los diferentes Procesos Solución. Esta división puede o no ser de forma equitativa.
- Enviar los datos correspondientes a cada Proceso Solución o, en su caso, indicar cuál es su ubicación.
- En ocasiones, se opta por que el Proceso Divisor, después de enviar los datos a cada Proceso Solución, se incorpore al trabajo, convirtiéndose en un Proceso Solución más; de esta forma no se desperdician los recursos del sistema utilizados por este proceso.

El número de Procesos Solución que deben utilizarse para resolver un determinado problema, depende de la complejidad de éste y de las características del sistema en que se vaya a implementar.

El estilo de programación de cada persona puede influir en la manera de distribuir las tareas, e incluso de organizarlas, por esta razón es muy factible el poder hacer que las funciones del Proceso Divisor y las del Proceso Integrador se unan en un solo proceso, o también que cualquiera de los procesos pueda ayudar al procesamiento de datos de otro. Estas cuestiones siempre tendrán repercusiones en la forma de resolver el problema y harán que el algoritmo sea más óptimo o también podrían complicarlo.

Proceso solución P_{Sx}

Las actividades que realiza cada Proceso Solución son:

- Recibir los datos que son enviados por el Proceso Divisor.

- Procesamiento de dichos datos.
- De ser necesario, el proceso establecerá comunicación con alguno de los otros Procesos Solución.
- Envío de los datos procesados o de la solución parcial al Proceso Integrador.

El hecho de que un Proceso Solución pueda establecer comunicación con otro de éstos, brinda una gran versatilidad a esta técnica ya que esto permite una mejor coordinación entre los procesos al momento de realizar el trabajo en conjunto, además mediante la comunicación entre procesos puede lograrse la sincronización que muchas veces se necesita sobre todo en los casos en los que los procesos están compitiendo por un determinado recurso. Mediante dicha comunicación, es factible evitar los problemas de concurrencia que podrían darse cuando diversos procesos compiten por un determinado recurso.

Proceso Integrador P_I

Las actividades que realiza el Proceso Integrador son:

- Recibir los resultados que envía cada Proceso Solución.
- Procesar los resultados parciales para poder proporcionar una solución completa al problema.

En la figura 4.10 se pueden apreciar los tiempos de duración o ciclo de vida de cada uno de los procesos así como la forma en que deben comunicarse.

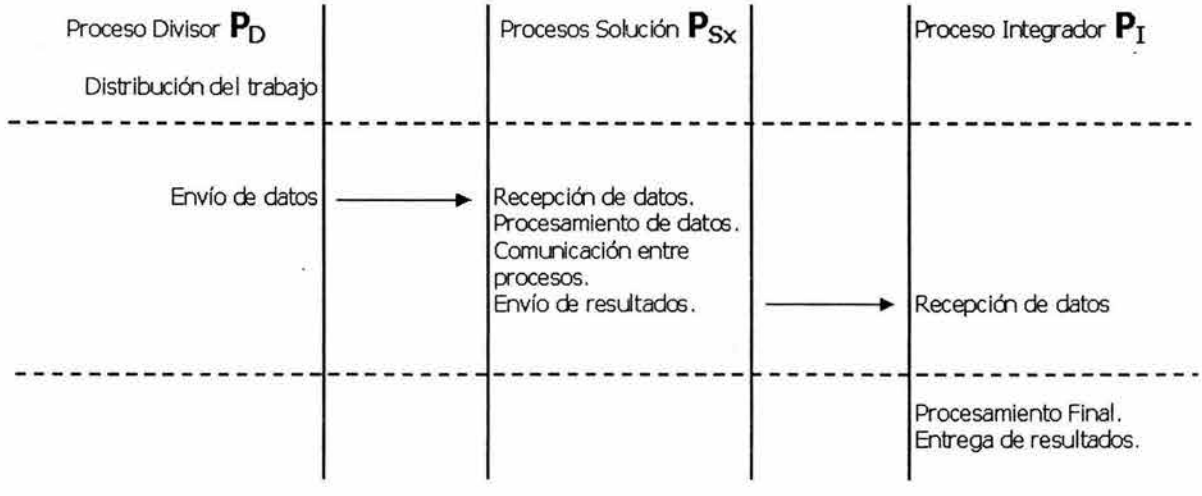


Figura 4.10. Diagrama de comunicación entre procesos para la técnica SPMD.

En la figura 4.10, la línea punteada separa las fases de envío y recepción de la fase de procesamiento de datos en cada uno de los procesos. Como ya lo mencionamos, el Proceso Divisor puede incorporarse al cálculo que realizan los Procesos Solución después que ha terminado de distribuir el trabajo. Los Procesos Solución intervienen sólo mientras procesan la información que les fue dada y terminan su ciclo de vida una vez que envían su solución parcial al Proceso Integrador. El Proceso Integrador coopera hasta que los Procesos Solución comienzan a enviarle resultados; estos úlitmos son unificados para conformar una solción total al problema.

4.5 Árboles Binarios.

Esta técnica se basa en el concepto de árbol, por lo tanto, es importante definir qué es un árbol binario para entender cómo funciona. Un árbol es una colección de nodos que puede estar vacía o no. Si no está vacía, el árbol estará formado por un nodo raíz y cero o más (sub)árboles que están unidos a la raíz por otros arcos. La figura 4.11 es un ejemplo de árbol: el nodo A es conocido como raíz; los nodos B, C, D, F y H son los padres, y los restantes son las hojas del árbol.

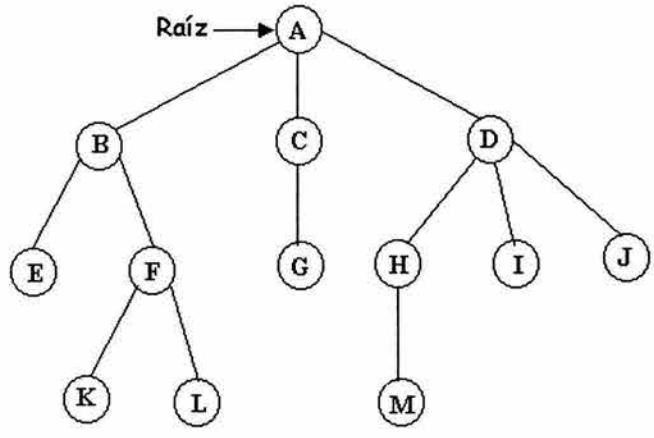


Figura 4.11. Representación gráfica de un árbol.

Para que un árbol sea binario es requisito indispensable que el número máximo de hijos que tenga cada nodo sea 2. Por lo tanto, el árbol anterior no es un árbol binario, ya que los nodos A y D tienen 3 hijos.

Suponiendo que tenemos **x** datos, correspondientes a cada una de las hojas de un árbol binario completo. Cada dato será obtenido de manera paralela y los resultados parciales se irán encontrando al ir recorriendo el árbol comenzando por las hojas. El proceso usado es de carácter ascendente ya que comienza a obtener resultados en las hojas (parte baja del árbol) hasta obtener el resultado final en la raíz.

Descripción:

La manera en que este algoritmo realiza su trabajo se puede representar mediante el esquema de la figura 4.12.

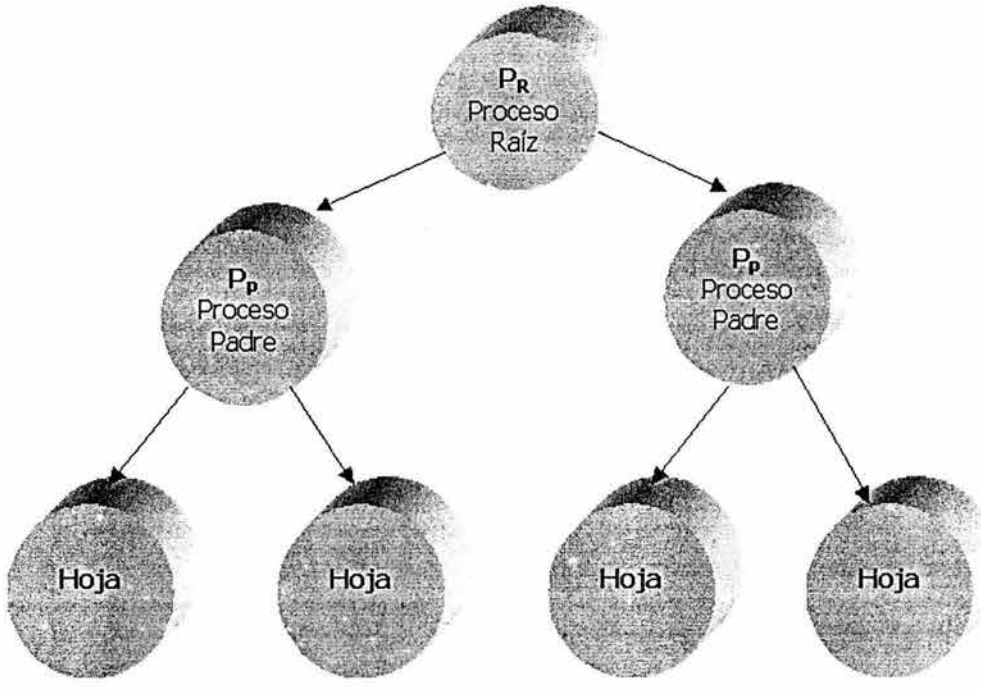


Figura 4.12. Diagrama característico de la técnica de Árboles Binarios.

En la figura 4.12 podemos observar cómo funciona la técnica de los Árboles Binarios. El proceso se realiza de manera ascendente, iniciando en las hojas, quienes comienzan a procesar sus datos, una vez que éstas han obtenido su resultado parcial envían datos a su proceso padre. Posteriormente, los padres comienzan a procesar la información que ha sido enviada por las hojas, al terminar la envían a su respectivo padre y así sucesivamente hasta llegar al proceso raíz que es el que obtiene el resultado o solución final.

A continuación describiremos las actividades que cada proceso realizará con el fin de mostrar de una manera más clara la técnica de Árboles Binarios.

Proceso Hoja **Hoja**

Las actividades que realiza el Proceso Hoja son:

- Obtener los datos de su fuente original (éstos deben ser divididos entre todas las hojas).
- Procesar los datos.

- Enviar a su correspondiente proceso padre los resultados parciales obtenidos.

Proceso Padre P_p

Las actividades que realiza el Proceso Padre son:

- Recibir datos provenientes de sus dos hijos.
- Procesar los datos recibidos.
- Enviar el resultado parcial obtenido a su correspondiente proceso padre.

Proceso Raíz P_R

Las actividades que realiza el Proceso Raíz son:

- Recibir datos provenientes de sus subárboles tanto izquierdo como derecho.
- Procesar los datos recibidos.
- Mostrar el resultado final.

Una vez que se ha pensado cada una de las actividades que realizarán los procesos es necesario tener en mente el diagrama de comunicación del algoritmo paralelo que deseamos desarrollar.

En el caso de la técnica de Árboles Binarios ya hemos comentado que el proceso se desarrolla de manera ascendente. Es por esta razón que, en el primer tiempo únicamente las hojas son los procesos activos, los cuales se encuentran realizando operaciones sobre sus datos, mientras que todos los demás procesos se encuentran en espera de recibir un resultado parcial para poder comenzar a generar el propio. Una vez que alguna de las hojas ha enviado un resultado parcial a su padre, éste comienza a procesar su propia información. El Proceso Raíz debe esperar a que los demás terminen su trabajo y le envíen sus resultados para poder proveer la solución total. Este proceso lo ejemplificamos en la figura 4.13.

En el esquema vemos cómo cada proceso tiene su propio tiempo para ir desarrollando los cálculos. Se observa también el paralelismo que existe cuando los Procesos Hoja se encuentran trabajando simultáneamente, de igual forma, una vez que éstos han terminado su procesamiento, todos los procesos padres inician sus cálculos y finalmente el proceso raíz es el que se encarga de reunir la información para mostrar el resultado final.

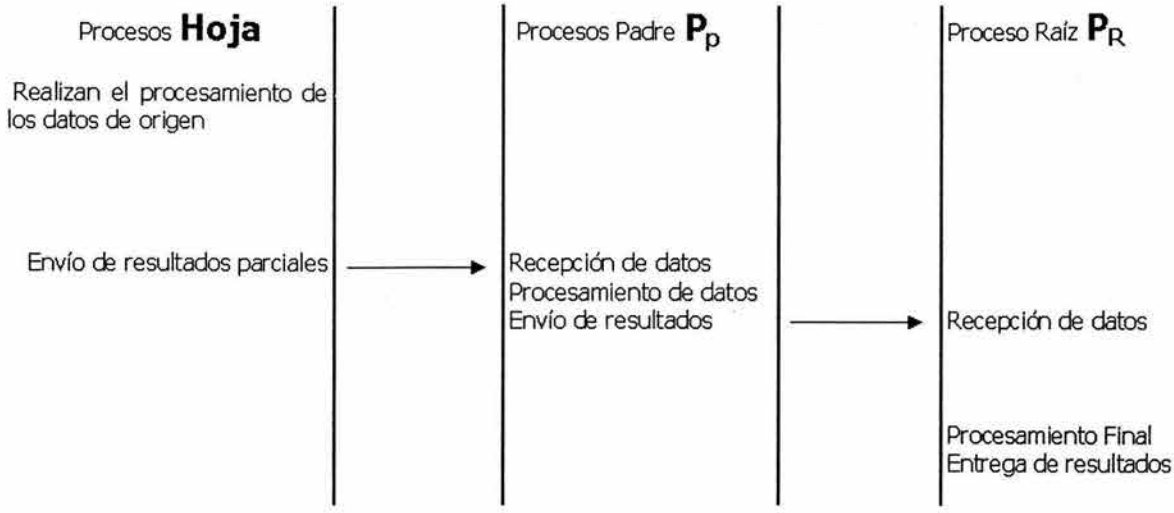


Figura 4.13. Diagrama de comunicación entre procesos para la técnica de Árboles Binarios.

5. Prácticas de programación distribuida con MPI.

En este capítulo desarrollaremos varias prácticas que nos servirán para ilustrar la manera en que se pueden aplicar las distintas técnicas de programación paralela en la creación de algunos programas que muestran la forma en que la carga de trabajo de un determinado problema puede ser distribuida entre varios procesos sobre un cluster heterogéneo mediante el empleo de la interfaz MPI.

5.1 Medidas Estadísticas.

5.1.1 Introducción.

Actualmente muchas áreas o disciplinas se apoyan en la estadística para realizar inferencias sobre algún tipo de fenómeno. En aplicaciones reales la cantidad de datos que tiene que ser procesada para obtener dicho tipo de inferencias es muy grande. Es por ello que la utilización de sistemas secuenciales se ha visto desplazada por los sistemas paralelos.

Existen muchas medidas estadísticas que pueden obtenerse de un conjunto de datos. En el presente trabajo decimos, a manera de ejemplo, calcular una medida de tendencia central (la media aritmética) y una medida de dispersión (la variancia).

Antes de comenzar a explicar cómo se estructurará el programa, es conveniente primero recordar cómo se calculan dichas medidas.

La media aritmética o promedio, es la suma de los datos que se quieren promediar, dividida entre el número de éstos. La expresión para su cálculo se muestra a continuación

$$\bar{x} = \frac{\sum_{i=0}^{n-1} x_i}{n}; \qquad (5.1\text{-}1)$$

donde: x_i es el i-ésimo dato, y n es el número de datos.

La variancia es el promedio de las desviaciones al cuadrado entre los datos y su media aritmética. Mide el grado de dispersión de los datos numéricos con respecto a su media aritmética. La expresión que nos permite calcularla es

$$\sigma^2 = \frac{\sum_{i=0}^{n-1}(x_i - \bar{x})^2}{n} = \frac{\sum_{i=0}^{n-1} x_i^2 - \frac{\left(\sum_{i=0}^{n-1} x_i\right)^2}{n}}{n}; \qquad (5.1\text{-}2)$$

donde: x_i es el i-ésimo dato, $\bar{x}$ es la media aritmética del conjunto de datos, y n es el número de datos.

La primera expresión es la definición de la variancia y la segunda es resultado de desarrollar el binomio al cuadrado de la primera fórmula y aplicarle las propiedades de la sumatoria. En el programa paralelo emplearemos la segunda expresión debido a que no depende del cálculo previo de la media de los datos para su obtención y por lo tanto agiliza la obtención de resultados y facilita la división del trabajo entre los procesos.

5.1.2 Planteamiento del problema.

Para la realización de este programa emplearemos la técnica SPMD, ya vista anteriormente. Antes de comenzar a explicar el código del programa, mostraremos de forma general cómo se distribuirá el trabajo y la forma en que se comunicarán los procesos.

Una vez que se ha decidido qué técnica se va a utilizar, es necesario determinar el papel que desempeñará cada proceso del grupo. En principio, puede seleccionarse cualquier proceso para que desempeñe cualquier papel, es decir, para que sea Proceso Divisor, Proceso Solución o Proceso Integrador, sin embargo, una buena elección de los roles de los procesos puede simplificar en gran medida los cálculos, la división del trabajo y la comunicación, obteniendo así un programa más claro y eficiente.

La asignación de papeles que decidimos emplear para un grupo de **n** procesos con rangos **0, 1, ..., n-1** se muestra en la tabla 5.1.

Tipo de proceso	Rango o identificador del proceso
Proceso Divisor	n-1
Proceso Solución	0, 1, ..., n-1
Proceso Integrador	n-1

Tabla 5.1. Asignación de papeles seleccionada en la aplicación paralela de medidas estadísticas.

Como se puede ver, el último proceso del grupo desempeña el papel de los tres tipos de procesos que conforman a esta técnica. Más adelante veremos que

esta asignación permite que los datos sean obtenidos por cada proceso con mayor facilidad, cabe destacar que la elección de roles no depende de la técnica sino del problema que se está resolviendo, en otras palabras, la asignación hecha en la tabla 5.1 no tiene porque ser siempre la misma para la técnica SPMD.

En las siguientes líneas describiremos con palabras la forma en que el programa realizará el cálculo de las medidas estadísticas y posteriormente analizaremos el código fuente.

Etapa 1 División del trabajo.

Esta etapa es realizada por el Proceso Divisor, éste es el encargado de obtener el número total de datos para determinar con cuántos datos trabajará cada proceso, una vez que lo ha determinado, envía un mensaje a los Procesos Solución que corresponde al número de datos con los que debe trabajar cada uno de ellos.

La distribución del trabajo que realiza el Proceso Divisor es la siguiente: los **n-1** procesos del grupo con rangos **0, 1, ..., n-2**, trabajarán con la misma cantidad de datos, dicha cantidad es el resultado de dividir el número total de datos entre **n-1**. El Proceso Divisor al momento de formar parte de los Procesos Solución trabajará con los datos sobrantes que no fueron asignados a los otros procesos, es decir, trabajará con un número de datos igual al residuo de la división hecha con anterioridad.

Etapa 2 Solución parcial.

Las partes de las expresiones (5.1-1) y (5.1-2) que se van a calcular en esta etapa son las siguientes:

$$\sum_{i=0}^{n-1} x_i \,, \tag{5.1-3}$$

$$\sum_{i=0}^{n-1} x_i^2 \,. \tag{5.1-4}$$

Una vez que los Procesos Solución saben el número de datos con el que deben trabajar, acceden a ellos y calculan la parte que les corresponde de las expresiones (5.1-3) y (5.1-4). Después de esto, deben comunicarse entre ellos para obtener una solución parcial formada por las expresiones (5.1-3) y (5.1-4).

Etapa 3 Obtención del resultado final.

En esta etapa, el Proceso Integrador recibe la solución parcial y obtiene el resultado final. Calcula la media sustituyendo la expresión (5.1-3) en (5.1-1), y sustituye (5.1-3) y (5.1-4) en (5.1-2) para obtener la variancia.

5.1.3 Código de la aplicación paralela.

Enseguida se muestra el código fuente del programa que calcula las anteriores medidas estadísticas.

```
/*************************************************************/
/*            TÉCNICAS DE PROGRAMACIÓN PARALELA              */
/*                                                           */
/*PROGRAMA:    estadistica.c                                 */
/*                                                           */
/*AUTORES:     Christian González,                           */
/*             Lissette González y                           */
/*             Eryk Ramírez.                                 */
/*                                                           */
/*DESCRIPCIÓN: Obtiene la media y la variancia de un         */
/*             conjunto de datos empleando la técnica de     */
/*             programación paralela SPMD.                   */
/*                                                           */
/*************************************************************/

/* Bibliotecas */
//mpi.h es necesaria para utilizar las funciones de la MPI.
#include <mpi.h>
#include <stdio.h>
#include <math.h>

/* Prototipos de función */
void divisorSolucionIntegrador(int,char**,int,int);
void solucion(int,int,char**);
FILE *abre(char*,char*);
void cierra(FILE*,char*);

/* Función principal */
int main(int argc,char *argv[]){
     int rango,tamanioGrupo;
     //Iniciamos un cómputo con MPI.
     MPI_Init(&argc,&argv);
          //Cada proceso obtiene su rango y el número de
          //procesos del grupo.
          MPI_Comm_rank(MPI_COMM_WORLD,&rango);
          MPI_Comm_size(MPI_COMM_WORLD,&tamanioGrupo);
```

```
        //Se le asigna una función a cada proceso
        //dependiendo del valor de su rango.
        if(rango==tamanioGrupo-1)
            divisorSolucionIntegrador(argc,argv,
                                      rango,tamanioGrupo);
        else
            solucion(rango,tamanioGrupo,argv);
    //Finalizamos el cómputo de la MPI.
    MPI_Finalize();
    return 0;
}

/* El proceso que ejecuta esta función se encarga de dividir
 * el trabajo, ayuda en el cálculo de la solución e integra
 * las soluciones parciales para obtener el resultado final.
 */
void divisorSolucionIntegrador(int argc,char **argv,int
                                    rango,int tamanioGrupo){
    double dato,sum=0.,sum2=0.,sumTotal,sum2Total;
    int i,tamBloque,residuo,totalDatos;
    FILE *archivoDeDatos;
    //Se comprueba que el número de argumentos sea el
    //correcto, de lo contrario se finaliza el programa.
    if(argc!=2){
        printf("Formato de ejecución:\n
                        estadistica <archivo_de_datos>\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    //Abrimos el archivo que contiene los datos a procesar.
    archivoDeDatos=abre(*(argv+1),"r");
    //Cálculo del tamaño del bloque de datos con los que
    //deberán trabajar los Procesos Solución.
    fseek(archivoDeDatos,0L,SEEK_END);
    totalDatos=ftell(archivoDeDatos)/sizeof(double);
    tamBloque=totalDatos/(tamanioGrupo-1);
    //Transmisión del número de datos que tiene que manejar
    // cada proceso.
    MPI_Bcast(&tamBloque,1,MPI_INT,rango,MPI_COMM_WORLD);
    //Este proceso trabaja con los datos que sobraron.
    residuo=totalDatos%(tamanioGrupo-1);
    fseek(archivoDeDatos,
                    rango*tamBloque*sizeof(double),SEEK_SET);
    for(i=0;i<residuo;i++){
        fread(&dato,sizeof(double),1,archivoDeDatos);
        //Obtención de una parte de la expresión(5.1-3)
        sum+=dato;
        //Obtención de una parte de la expresión(5.1-4)
```

```
            sum2+=pow(dato,2);
        }
        cierra(archivoDeDatos,*(argv+1));
        //Cálculo completo de la expresión (5.1-3)
        MPI_Reduce(&sum,&sumTotal,1,
                        MPI_DOUBLE,MPI_SUM,rango,MPI_COMM_WORLD);
        //Cálculo completo de la expresión (5.1-4)
        MPI_Reduce(&sum2,&sum2Total,1,
                        MPI_DOUBLE,MPI_SUM,rango,MPI_COMM_WORLD);
        //Se integran las solciones parciales.
        printf("media=%lf\nvariancia=%lf\n",sumTotal/totalDatos,
            (sum2Total-pow(sumTotal,2)/totalDatos)/totalDatos);
}

/* Los Procesos Solución ejecutan esta función para calcular
 * un resultado parcial del problema.
 */
void solucion(int rango,int tamanioGrupo,char **argv){
        double dato,sum=0.,sum2=0.;
        int tamBloque;
        register i;
        FILE *archivoDeDatos;
        //Los procesos reciben el número de datos con el que
        //deben trabajar.
        MPI_Bcast(&tamBloque,1,MPI_INT,
                                tamanioGrupo-1,MPI_COMM_WORLD);
        archivoDeDatos=abre(*(argv+1),"r");
        //Cada proceso se dispone a acceder a la parte de la
        //información que le corresponde dentro del archivo.
        fseek(archivoDeDatos,
                        rango*tamBloque*sizeof(double),SEEK_SET);
        for(i=0;i<tamBloque;i++){
            fread(&dato,sizeof(double),1,archivoDeDatos);
            //Obtención de una parte de la expresión(5.1-3)
            sum+=dato;
            //Obtención de una parte de la expresión(5.1-4)
            sum2+=pow(dato,2);
        }
        cierra(archivoDeDatos,*(argv+1));
        //Cálculo completo de la expresión (5.1-3)
        MPI_Reduce(&sum,NULL,1,MPI_DOUBLE,
                        MPI_SUM,tamanioGrupo-1,MPI_COMM_WORLD);
        //Cálculo completo de la expresión (5.1-4)
        MPI_Reduce(&sum2,NULL,1,MPI_DOUBLE,
                        MPI_SUM,tamanioGrupo-1,MPI_COMM_WORLD);
}
```

```
/* Esta función abre un archivo y verifica que no existan
 * errores al abrirlo.
 */
FILE *abre(char *nombre,char *acceso){
     FILE *archivo;
     if((archivo=fopen(nombre,acceso))==NULL){
          printf("Ocurrió un error
                          al abrir el archivo: %s\n",nombre);
          MPI_Abort(MPI_COMM_WORLD,1);
     }
     return archivo;
}

/* Esta función cierra un archivo y verifica que no existan
 * errores al cerrarlo.
 */
void cierra(FILE *archivo,char *nombre){
     if(fclose(archivo)==EOF){
          printf("Ocurrió un error al
                            cerrar el archivo: %s\n",nombre);
          MPI_Abort(MPI_COMM_WORLD,1);
     }
}
```

5.1.4 Comentarios adicionales.

La función principal (main) es ejecutada por todos los procesos del grupo, éstos obtienen su rango y el número total de procesos participantes en el cálculo y, con base en ello, determinan qué función deben ejecutar.

El archivo de datos es distribuido entre los **n** procesos del grupo como se muestra en la figura 5.1.

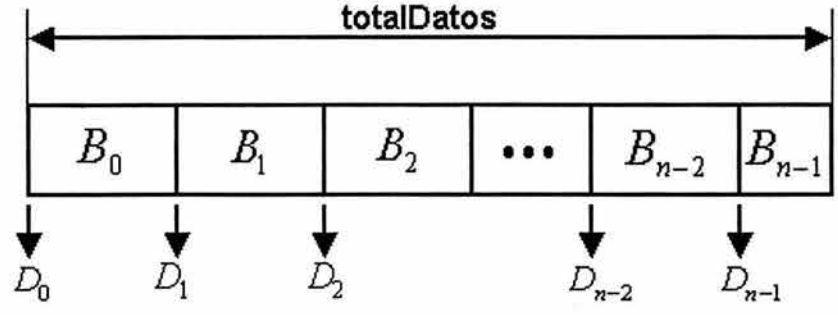


Figura 5.1. Distribución de los datos de entrada entre los procesos del grupo.

Con tal disposición, el proceso con rango igual a **i** trabaja con el bloque de datos B_i que se encuentra a D_i elementos del inicio del archivo que contiene un número de elementos igual a `totalDatos` (véase figura 5.1).

Por otra parte, el bloque de datos B_i comienza después de D_i=`i*tamBloque` elementos y posee `tamBloque` elementos, para **i<n-1**. El último bloque de datos, B_{n-1}, comienza después de `(n-1)*tamBloque` elementos y posee `residuo` elementos.

El proceso con rango igual a **i** manipula sus datos para obtener las expresiones (5.1-5) y (5.1-6), para **i<n-1**.

$$sum_i = \sum_{j=D_i}^{D_i+tamBloque-1} Dato[j] \tag{5.1-5}$$

$$sum2_i = \sum_{j=D_i}^{D_i+tamBloque-1} (Dato[j])^2 \tag{5.1-6}$$

donde: $Dato[j]$ es el j-ésimo elemento del archivo.

El proceso con rango igual a **n-1** manipula sus datos para obtener:

$$sum_{n-1} = \sum_{j=D_{n-1}}^{D_{n-1}+residuo-1} Dato[j] \tag{5.1-7}$$

$$sum2_{n-1} = \sum_{j=D_{n-1}}^{D_{n-1}+residuo-1} (Dato[j])^2 \tag{5.1-8}$$

donde: $Dato[j]$ es el j-ésimo elemento del archivo.

Después del cálculo de las expresiones (5.1-5)-(5.1-8), los procesos se comunican entre sí para obtener la solución parcial formada por las expresiones (5.1-3) y (5.1-4); en otras palabras, obtienen:

$$sumTotal = \sum_{i=0}^{n-1} sum_i \tag{5.1-9}$$

$$sum2Total = \sum_{i=0}^{n-1} sum2_i \tag{5.1-10}$$

Las expresiones (5.1-9) y (5.1-10) se obtienen fácilmente por medio de la invocación de la función `MPI_Reduce`.

Con las soluciones parciales, el proceso integrador calcula finalmente la media y la variancia del conjunto de datos.

5.1.5 Evaluación del desempeño.

Ya que hemos comprendido la forma en que opera esta aplicación paralela, ahora analizaremos cómo es su desempeño al ser ejecutada en el cluster. De manera general, el desempeño de una aplicación paralela dependerá en gran medida de la capacidad computacional del sistema donde se ejecute, entre mayor sea ésta, mejor será el desempeño de la aplicación.

En este análisis tomaremos en cuenta dos factores que influyen en el comportamiento de cualquier aplicación paralela: la carga de trabajo y el número procesos que participan en el cálculo. Para ello, mediremos los tiempos de ejecución de la aplicación al ir aumentando el número de procesos para tres distintas cargas de trabajo: una pequeña, una mediana y una grande.

Las cargas de trabajo con las que probamos el programa fueron de 500, 2500000 y 8000000 datos de entrada. En cuanto a la variación en el número de procesos, hicimos pruebas con 2, 3, ..., y 14 procesos. El menor número de procesos que utilizamos fue 2, pues es el mínimo número que puede utilizar la técnica SPMD. El máximo, 14 procesos, corresponde al número de nodos que componen el cluster –es posible ejecutar más de un proceso por nodo pero, debido a las limitantes de memoria del cluster y la disposición del algoritmo, ésta no resulta una buena opción–.

Para las mediciones de tiempos utilizamos la función que para tal efecto provee la MPI, **MPI_Wtime**. Es importante destacar que, las secciones de código que permiten medir el tiempo de ejecución de un programa deben ser colocadas entre el código del algoritmo de modo que no afecten su comportamiento de alguna forma y verificando que en realidad se esté midiendo bien el tiempo total de ejecución, en otras palabras, lo idóneo es que un solo proceso realice la medición, éste deberá ser el último en finalizar. Es conveniente realizar un análisis del programa para determinar qué proceso es el que finaliza al último, si se tienen problemas para determinarlo con claridad, se pueden utilizar otras funciones como **MPI_Barrier**, para forzarlo.

Para cada condición distinta –de carga de trabajo y de número de procesos– se realizaron 5 corridas del programa y se obtuvo el tiempo promedio de ejecución con el objeto de obtener un valor más representativo.

Debido a que el servidor del cluster es mucho más potente que cualquiera de sus nodos y, por lo tanto, las contribuciones de los nodos sólo pueden ser bien

apreciadas si no participa en el cálculo, en las corridas de esta aplicación ningún proceso fue creado en el servidor, es decir que, se le pasó la opción `nolocal` al comando **mpirun**.

Otro punto importante que se debe tomar en cuenta cuando se trabaja con un cluster heterogéneo es la forma en que se asignarán los procesos en las distintas máquinas que lo conforman, pues esta asignación repercute directamente en el desempeño de la aplicación paralela. Para el caso particular de este programa, el proceso que posee el mayor rango es el que, en general, realiza menos trabajo, por esta razón resulta conveniente que se ejecute en alguno de los equipos de menor capacidad dejando así los mejores equipos del cluster para los demás procesos. Por eso fue que empleamos la opción `machinefile` del comando **mpirun**. También debemos considerar que, al utilizar la opción `nolocal`, el arranque de procesos remotos en los nodos ya no será realizado por el servidor y por lo tanto debe seleccionarse a uno, de entre los mejores equipos, para que realice esta labor; la selección de un equipo lento para este fin, produce un incremento en el tiempo de ejecución.

5.1.5.1 Resultados.

La tabla 5.2 muestra los tiempos de ejecución obtenidos al correr la aplicación paralela para los distintos tamaños de carga y número de procesos.

	Tiempos de ejecución para:		
Número de procesos	**500 datos [s]**	**2500000 datos [s]**	**8000000 datos [s]**
2	0.046378	97.422729	315.137676
3	0.059954	59.945153	166.433860
4	0.058236	49.524591	118.102689
5	0.097335	36.079392	102.223430
6	0.115384	30.068865	88.202201
7	0.127469	26.370065	84.830608
8	0.129121	24.443453	79.796829
9	0.154681	22.694439	76.862207
10	0.172986	21.458896	73.872031
11	0.215236	20.839941	73.420916
12	0.239569	20.489942	70.800158
13	0.228315	19.523830	62.567552
14	0.189774	20.558833	55.337932

Tabla 5.2. Resultados obtenidos con las pruebas realizadas a la aplicación paralela de medidas estadísticas.

En la figura 5.2, mostramos la gráfica de los tiempos de ejecución obtenidos al ir aumentando el número de procesos con una carga pequeña de trabajo.

Podemos observar que, el tiempo de ejecución se tiende a incrementar conforme se utilizan más procesos para realizar el cálculo. El comportamiento anterior es común en las aplicaciones paralelas cuando los cálculos involucrados no son considerables, pues es mayor el tiempo invertido en comunicación, coordinación, sincronización y otras cuestiones inherentes de estas aplicaciones que el tiempo utilizado en la resolución del problema. Por lo tanto, no resulta benéfico utilizar la aplicación cuando no es muy grande el procesamiento de datos.

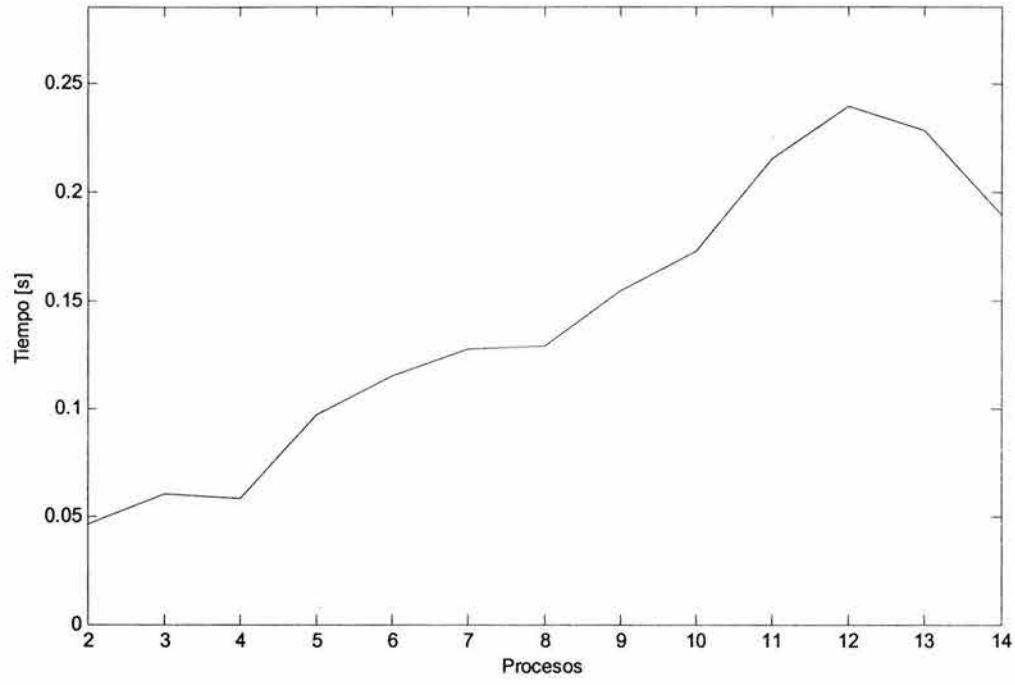


Figura 5.2. Gráfica de tiempo de ejecución contra número de procesos, con una carga constante de trabajo de 500 datos.

El comportamiento que tiene el programa para una carga de trabajo moderada se muestra en la figura 5.3. En este caso, debido a que el programa es eminentemente paralelo –son pocas las partes del mismo en las que el código resulta secuencial–, se aprecia que, conforme se incrementa el número de procesos, se reduce el tiempo de ejecución.

Finalmente, en la figura 5.4, vemos la gráfica del desempeño de la aplicación ante una carga considerada como grande. De nuevo se ve reducido el tiempo de ejecución conforme se suman más procesos a la resolución del problema. Podemos comentar que esta aplicación mostró un buen desempeño, éste se debe principalmente a que el algoritmo requiere de poca comunicación entre procesos y, por lo tanto, la mayor parte del tiempo es dedicada a la resolución del problema. También es necesario aclarar que, debido a que el algoritmo no es muy complejo, fue posible crear una aplicación con poca dependencia entre procesos, pero esto no siempre es posible, especialmente cuando la complejidad del algoritmo es mayor.

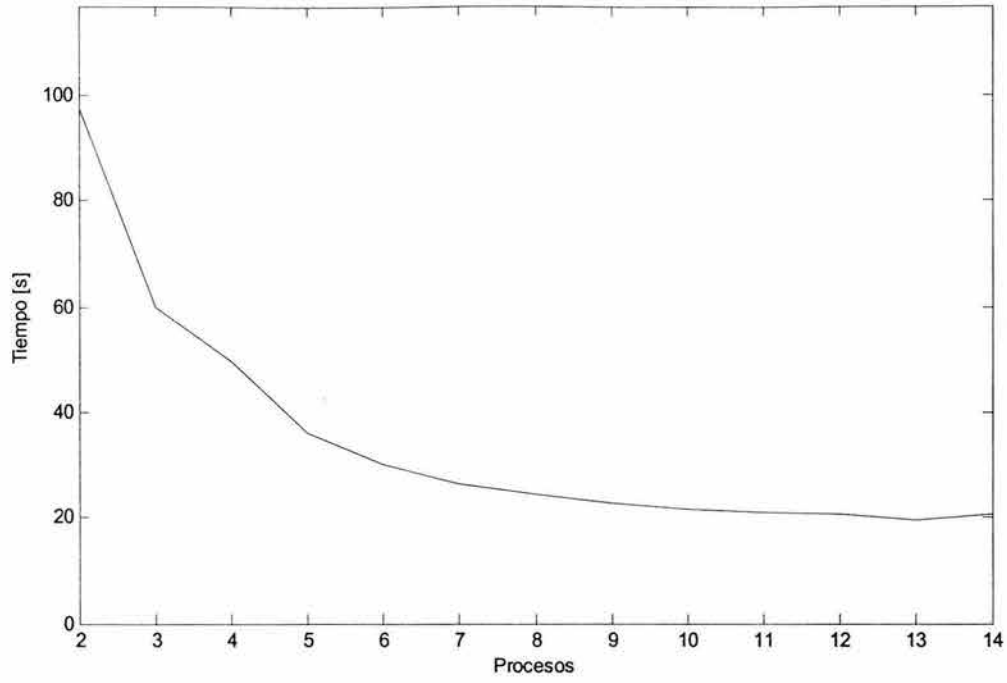


Figura 5.3. Gráfica de tiempo de ejecución contra número de procesos, con una carga constante de trabajo de 2500000 datos.

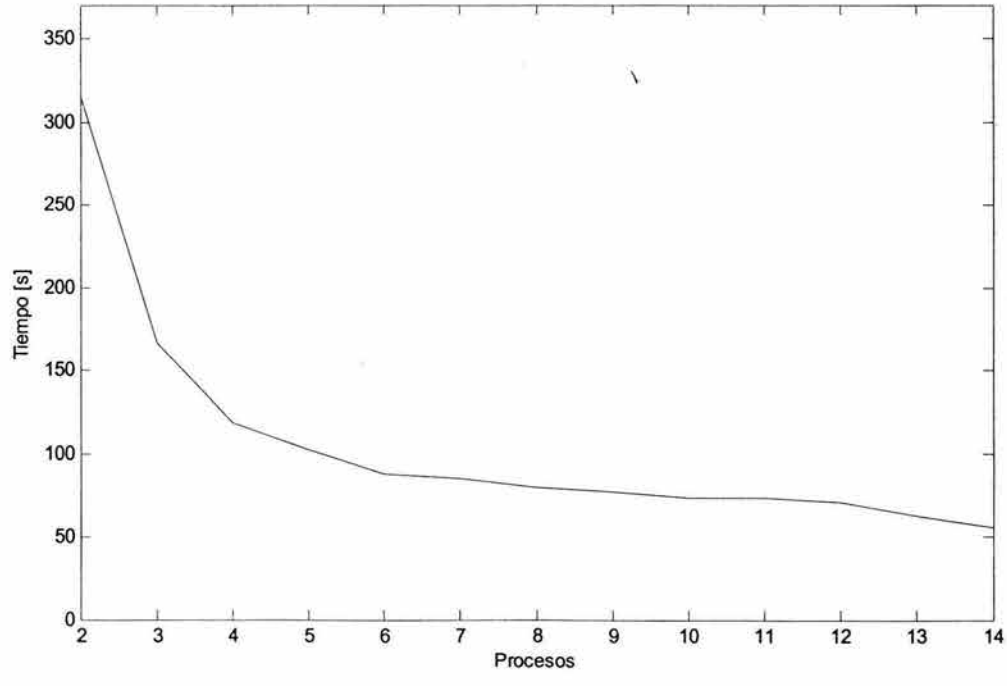


Figura 5.4. Gráfica de tiempo de ejecución contra número de procesos, con una carga constante de trabajo de 8000000 datos.

5.2 Ordenamiento de datos.

5.2.1 Introducción.

El ordenamiento de datos está presente en un sinnúmero de aplicaciones, es de gran utilidad pues los datos que están ordenados pueden ser accedidos con mayor rapidez y facilidad.

Debido a la importancia que tiene el ordenamiento, se han desarrollado muchos algoritmos con el afán de ordenar los datos lo más rápido posible. Cuando se trabaja con cantidades pequeñas de datos, cualquier algoritmo resulta conveniente, pero para ordenar gran cantidad de datos el método seleccionado puede significar una gran mejora en tiempo, del mismo modo, otro factor que puede influir en el decremento en tiempo es la capacidad computacional de la arquitectura en la que se ejecuta el programa de ordenamiento.

En nuestro caso, optamos por realizar un programa de ordenamiento en paralelo utilizando dos algoritmos: uno de ellos es el Quicksort y el otro es conocido como Intercalación y consiste en una mezcla o fusión de dos grupos de datos previamente ordenados.

Antes de comenzar a explicar cómo funciona nuestra implementación de ordenamiento en paralelo, veremos en qué consisten los algoritmos antes mencionados.

5.2.1.1 Ordenamiento por el método Quicksort.

La idea central de este algoritmo es la siguiente:

1. Se toma un elemento X de una posición cualquiera del arreglo.
2. Se trata de ubicar a X en la posición correcta del arreglo, de tal forma que todos los elementos que se encuentren a su izquierda sean menores o iguales a X y todos que se encuentren a su derecha sean mayores o iguales a X.
3. Se repiten los pasos anteriores pero ahora para los conjuntos de datos que se encuentran a la izquierda y a la derecha de la posición correcta de X.
4. El proceso termina cuando todos los elementos se encuentran en su posición correcta, es decir, están ordenados.

Debe seleccionarse entonces un elemento X. Este valor se puede escoger aleatoriamente; el valor óptimo es aquel que está precisamente en medio del rango de valores de los datos. A pesar de ser este último el mejor valor para X, si se elige un valor cualquiera del conjunto de datos, el algoritmo funciona correctamente. Además, para una gran cantidad de datos es posible que el esfuerzo computacional requerido para encontrar el valor óptimo de X haga que la ventaja en tiempo con dicha selección no sea tan sorprendente. Por lo anterior,

nosotros optamos por seleccionar como X al primer elemento del conjunto de datos que se va a ordenar.

Una vez que se ha seleccionado a X, se procede a colocarlo en el lugar que le corresponde. Existen diferentes maneras en que puede conseguirse tal resultado, nosotros lo haremos como sigue: comenzamos a recorrer el arreglo de derecha a izquierda comparando si los elementos son mayores o iguales a X. Si un elemento no cumple con esta condición, se intercambian los mismos y almacenamos la posición del elemento intercambiado. Se inicia nuevamente el recorrido pero ahora de izquierda a derecha, comparando si los elementos son menores o iguales a X. Si un elemento no cumple con esta condición, entonces se intercambian los mismos y se almacena la posición del elemento intercambiado. Se repiten los pasos anteriores hasta que el elemento X encuentra su posición correcta en el arreglo. Finalmente, se realiza el mismo procedimiento pero con los grupos de datos que están a la izquierda y a la derecha de X.

A continuación, mostraremos un ejemplo que ilustra lo descrito con anterioridad.

Ejemplo. Supongamos que se desea ordenar el siguiente grupo de elementos empleando el método Quicksort:

2	45	-3	9	2	0	7

Seleccionamos el primer elemento para comenzar las comparaciones, por lo tanto X=2.

1) PRIMERA PASADA.

a) Recorrido de derecha a izquierda.

$7 \geq 2$ Sí se cumple, por lo tanto, no hay intercambio.
$0 \geq 2$ No se cumple, por lo tanto, sí hay intercambio.

0	45	-3	9	2	2	7
↑					↑	

b) Recorrido de izquierda a derecha.

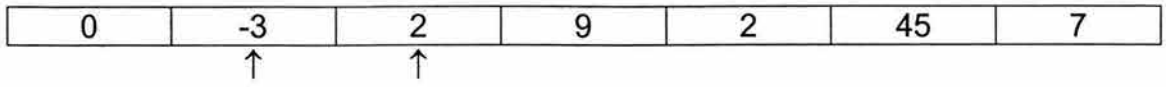
$45 \leq 2$ No se cumple, por lo tanto, sí hay intercambio.

0	2	-3	9	2	45	7
	↑				↑	

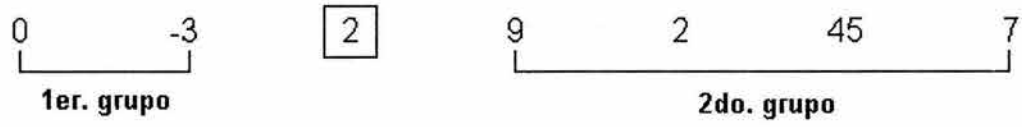
2) SEGUNDA PASADA.

a) Recorrido de derecha a izquierda.

$2 \geq 2$ Sí se cumple, por lo tanto, no hay intercambio.
$9 \geq 2$ Sí se cumple, por lo tanto, no hay intercambio.
$-3 \geq 2$ No se cumple, por lo tanto, sí hay intercambio.



Como el recorrido de izquierda a derecha debería iniciarse en la misma posición en donde se encuentra el elemento X, el proceso se termina debido a que éste ya se encuentra en la posición correcta.



Obsérvese que los elementos que forman parte del primer grupo son menores o iguales a X, y los que forman parte del segundo grupo son mayores o iguales a X. Este proceso de particionamiento aplicado para localizar la posición correcta de un elemento X se repite cada vez que queden grupos formados por dos o más elementos. En la figura 5.5 se presenta la ubicación que van tomando los diferentes elementos conforme se sigue aplicando el algoritmo, los elementos que están en un recuadro son los que ya se encuentran en la posición correcta.

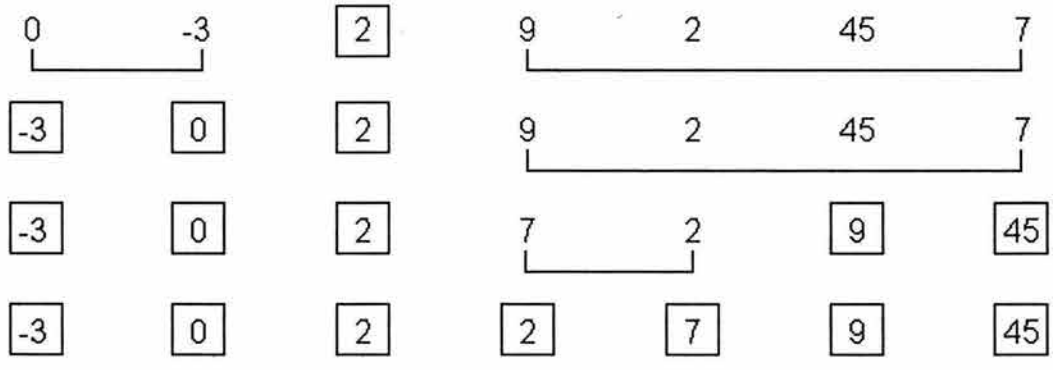


Figura 5.5. Progresión de resultados obtenidos al aplicar el algoritmo Quicksort.

5.2.1.2 Ordenamiento por Intercalación.

Este algoritmo es comúnmente utilizado cuando se poseen dos grupos de elementos, cada uno de ellos ordenado, y se desea obtener un tercer grupo formado por los elementos de ambos grupos y también ordenado.

El ordenamiento por Intercalación puede resumirse de la siguiente manera:

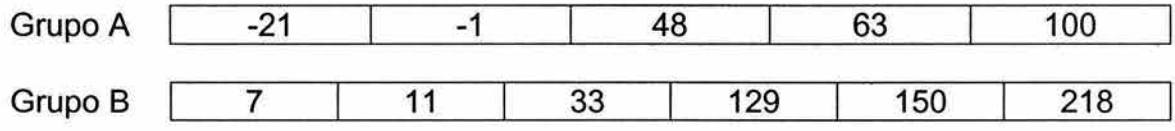
- Se toman dos elementos, uno de cada grupo, se comparan y el menor es colocado en el tercer grupo.

- Se toma otro elemento del grupo que aportó el último elemento del tercer grupo y se compara con el elemento que no fue seleccionado en la anterior comparación. El menor es colocado en el tercer grupo.

- El proceso del paso anterior continúa hasta que se terminen los elementos de uno de los grupos, después de esto, los elementos sobrantes del otro grupo son colocados en el tercer grupo.

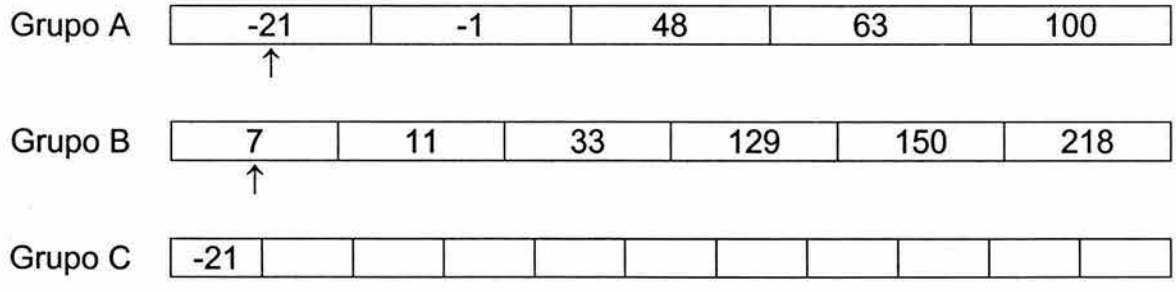
Al comenzar el algoritmo, la primera comparación se realiza entre el primer elemento de cada grupo. A continuación mostraremos un ejemplo que demuestra la aplicación de este algoritmo.

Ejemplo. Se tienen dos grupos de elementos ordenados, A y B, y se desea unirlos en un tercer grupo, C, de elementos ordenados empleando el algoritmo de intercalación.

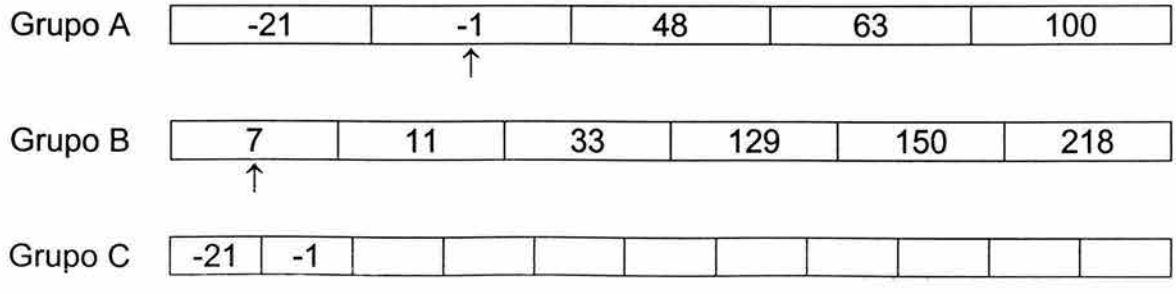


Las comparaciones y asignaciones que deben realizarse para formar al grupo C son las siguientes:

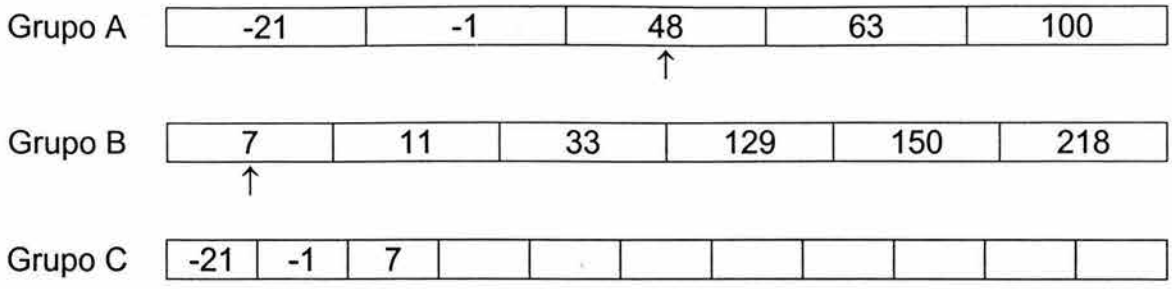
-21 < 7 Sí se cumple, por lo tanto, -21 es colocado en el grupo C.



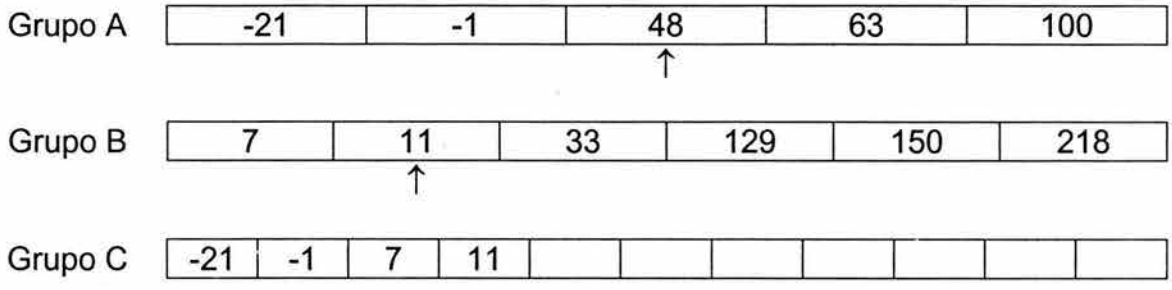
-1 < 7 Sí se cumple, por lo tanto, -1 es colocado en el grupo C.



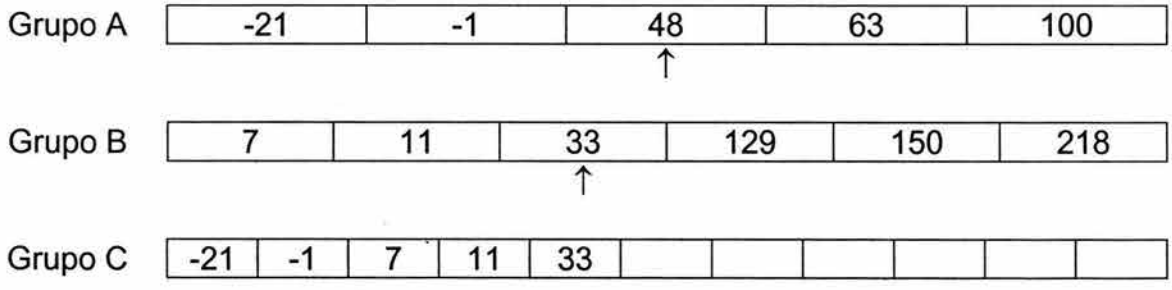
48 < 7 No se cumple, por lo tanto, 7 es colocado en el grupo C.



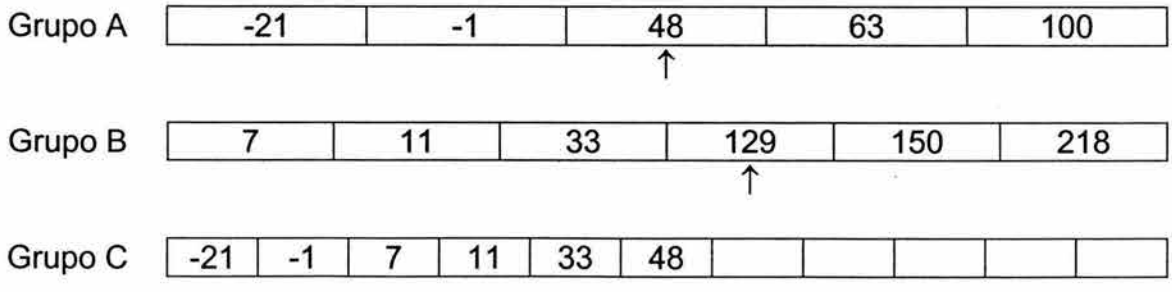
48 < 11 No se cumple, por lo tanto, 11 es colocado en el grupo C.



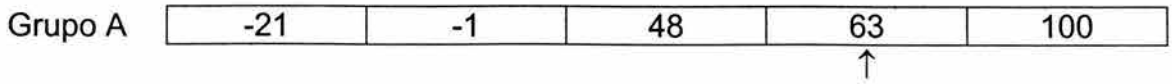
48 < 33 No se cumple, por lo tanto, 33 es colocado en el grupo C.

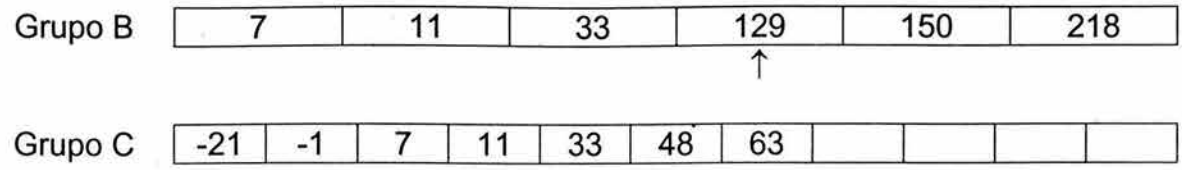


48 < 129 Sí se cumple, por lo tanto, 48 es colocado en el grupo C.

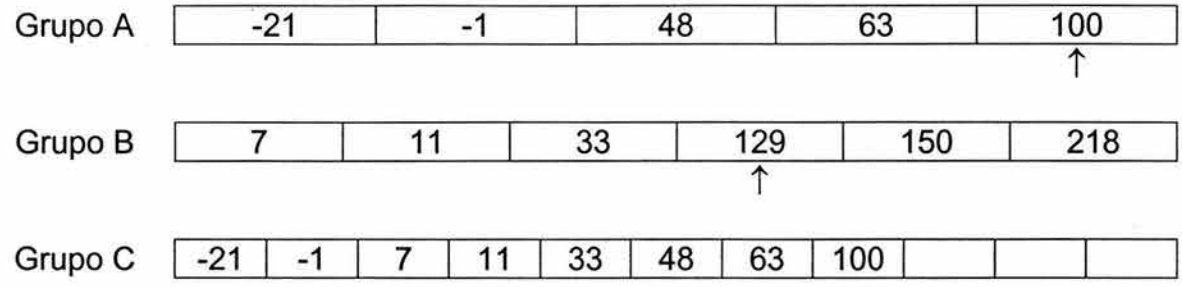


63 < 129 Sí se cumple, por lo tanto, 63 es colocado en el grupo C.

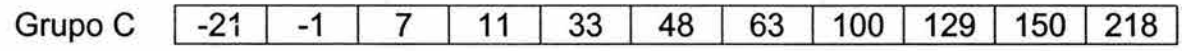




100 < 129 Sí se cumple, por lo tanto, 100 es colocado en el grupo C.



Como se terminaron los elementos del grupo A, se copian al grupo C los elementos sobrantes del grupo B y finaliza el algoritmo.



5.2.2 Planteamiento del problema.

Para desarrollar este programa emplearemos la técnica de Árboles Binarios. Antes de comenzar a explicar el código del programa, mostraremos de forma general cómo se distribuirá el trabajo y la forma en que se comunicarán los procesos para brindar una mayor claridad.

Una vez que se ha decidido qué técnica se va a utilizar, es necesario determinar el papel que desempeñará cada proceso del grupo. En principio, puede seleccionarse cualquier proceso para que desempeñe cualquier papel, es decir, para que sea Proceso Raíz, Proceso Padre o Proceso Hoja, sin embargo, una buena elección de los roles de los procesos puede simplificar en gran medida los cálculos, la división del trabajo y la comunicación, obteniendo así un programa más claro y eficiente.

La asignación de papeles que decidimos utilizar para un grupo de $\mathbf{n = 2^m-1}$ procesos con rangos **0, 1, ..., n-1** se muestra en la tabla 5.3.

Para completar la información de la tabla 5.3, en la figura 5.6 se muestra cómo decidimos distribuir los procesos según su rango en el árbol binario.

Tipo de proceso	Rango o identificador del proceso
Proceso Raíz	0
Proceso Padre	$1 \leq \text{rango} < 2^{m-1}-1$
Proceso Hoja	$2^{m-1}-1 \leq \text{rango} < n$

Tabla 5.3. Asignación de papeles seleccionada en la aplicación paralela de ordenamiento de datos.

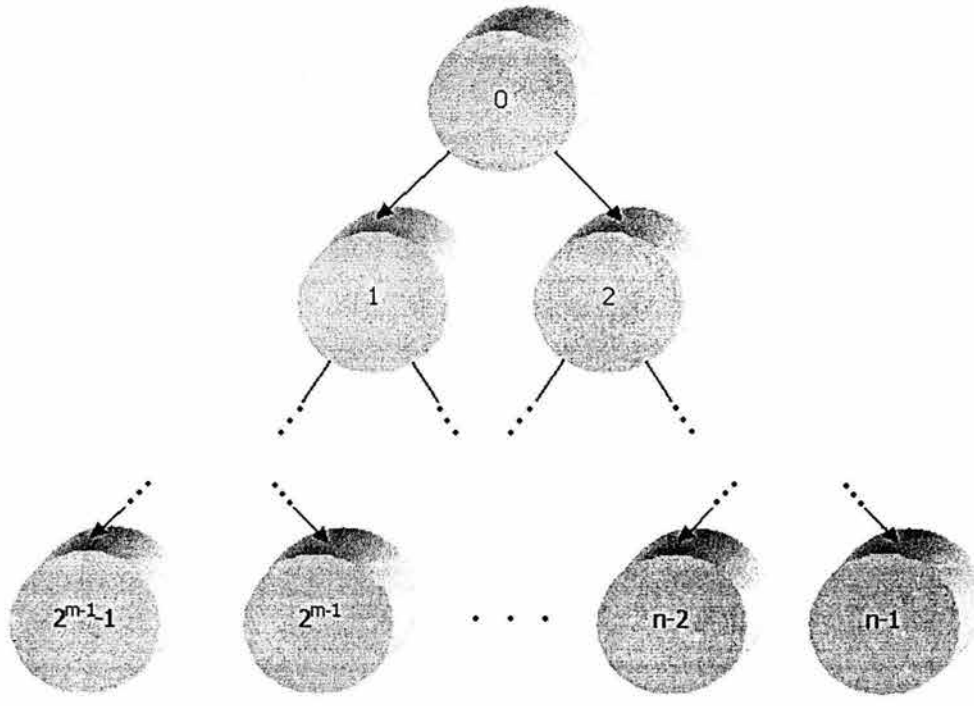


Figura 5.6. Distribución de los procesos en el árbol binario, según su rango.

Con esta asignación, cada proceso tiene un papel bien definido y ninguno desempeña más de un papel. Además, esta elección permite, como más adelante veremos, encontrar una función sencilla para comunicar a cada hijo con su padre y viceversa.

Para este desarrollo, asumiremos que la fuente de datos es un archivo formado por elementos que son números de punto flotante. Lo anterior, nos permitirá mostrar nuestra implementación de ordenamiento en paralelo pero hay que tener presente que el ordenamiento de datos no sólo se refiere a los números sino a cualquier tipo de objeto que pueda ser comparado con otros mediante una relación de desigualdad.

A continuación describimos la forma en que decidimos dividir el trabajo y cómo el programa lleva a cabo el ordenamiento de los datos.

Ordenamiento en paralelo.

Los Procesos Hoja.

El primer paso que realizamos es la división de los datos, si recordamos, la técnica de Árboles Binarios propone la obtención de resultados en forma ascendente, es decir, los Procesos Hoja son los primeros en trabajar para poder enviar sus resultados a sus respectivos Procesos Padre. Debido a esta característica, los Procesos Hoja son los que realizan la división del trabajo.

Para distribuir el conjunto de datos, cada Proceso Hoja debe calcular el número de elementos que componen al archivo y, después de ello, conociendo el número de Procesos Hoja existentes, debe calcular el número de datos que le corresponden y la ubicación de éstos en el archivo.

El número de elementos con los que trabajará cada Proceso Hoja es obtenido como sigue: se divide el número total de elementos entre el total de Procesos Hoja, esto proporciona el número de elementos con los que debe trabajar cada Proceso Hoja, si existen elementos sobrantes debido a que existió un residuo en la división anterior, éstos se reparten uno por uno entre las distintas Hojas comenzando la repartición por el Proceso Hoja que tiene el menor rango.

Una vez que cada Proceso Hoja cuenta con sus propios datos, los ordena utilizando el algoritmo Quicksort y, una vez ordenados, los envía a su padre.

Como podemos notar, al final de la labor de los Procesos Hoja se obtienen varios grupos de elementos ya ordenados y éstos son transmitidos a sus padres para que comiencen su trabajo.

Los Procesos Padre.

Los Procesos Padre reciben por medio de sus hijos la información con la que deberán trabajar, como estamos utilizando árboles binarios, un Proceso Padre sólo tiene dos hijos. Cada Proceso Padre recibe entonces dos grupos de datos ordenados mismos que fusiona en un tercer grupo ordenado mediante el ordenamiento por intercalación. Una vez que obtiene este tercer grupo debe enviarlo a su respectivo padre.

Como podremos ver, la solución se va obteniendo al ir ascendiendo en el árbol, obteniéndose grupos de datos ordenados cada vez mayores conforme se van subiendo los niveles del árbol.

El Proceso Raíz.

El Proceso Raíz recibe los dos últimos grupos de datos ordenados, los fusiona y los almacena en un archivo de salida que constituye el resultado final de este programa.

El siguiente es el código fuente del programa que realiza el ordenamiento de datos en paralelo.

5.2.3 Código de la aplicación paralela.

```
/*************************************************************/
/*            TÉCNICAS DE PROGRAMACIÓN PARALELA              */
/*                                                           */
/*PROGRAMA:    ordena.c                                      */
/*                                                           */
/*AUTORES:     Christian González,                           */
/*             Lissette González y                           */
/*             Eryk Ramírez.                                 */
/*                                                           */
/*DESCRIPCIÓN: Ordena un grupo de datos por medio de los     */
/*             algoritmos Quicksort e Intercalación,         */
/*             mediante la técnica de programación paralela*/
/*             de Árboles Binarios.                          */
/*                                                           */
/*************************************************************/

/* Bibliotecas */
//mpi.h es necesaria para utilizar las funciones de la MPI.
#include <mpi.h>
#include <stdio.h>

/* Prototipos de función */
void raiz(char**);
void padre(int);
void hoja(int,char**,int,int);
void quickSort(float*,int);
void auxQuickSort(float*,int,int);
void intercala(float*,int,float*,int,float*);
void cierra(FILE*,char*);
void *reservaMemoria(size_t);
FILE *abre(char*,char*);

/* Función principal */
int main(int argc,char *argv[]){
     int rango,tamanioGrupo,numHojas=1,numNodos=1;
     //Iniciamos un cómputo con MPI.
     MPI_Init(&argc,&argv);
          //Cada proceso obtiene su rango y el número de
          //procesos del grupo.
          MPI_Comm_size(MPI_COMM_WORLD,&tamanioGrupo);
          MPI_Comm_rank(MPI_COMM_WORLD,&rango);
          //Verificamos que el número de procesos del grupo
```

```
        //sea un valor n=2^m-1, de lo contrario el árbol no
        //binario no estaría completo.
        while(numNodos<tamanioGrupo)numNodos+=numHojas<<=1;
        if(!rango&&
                (numNodos!=tamanioGrupo||tamanioGrupo==1)){
            printf("El número de
                    procesos creados debe ser 2^m-1,
                                m>1,con 'm' entero\n");
            MPI_Abort(MPI_COMM_WORLD,1);
        }
        //Se le asigna una función a cada proceso
        //dependiendo del valor de su rango.
        if(!rango)
            raiz(argv);
        else if(rango<numNodos-numHojas)
            padre(rango);
        else
            hoja(argc,argv,rango,numHojas);
    //Finalizamos el cómputo de MPI.
    MPI_Finalize();
    return 0;
}

/* Esta función es ejecutada por el Proceso Raíz, se encarga
 * de recibir los datos provenientes de sus hijos, los
 * fusiona y coloca el resultado final en un archivo de
 * salida.
 */
void raiz(char **argv){
    MPI_Status status;
    float *secuencia,*secuencia1,*secuencia2;
    int tamBloque,tamBloque1,tamBloque2;
    FILE *archivoOrdenado;
    //Recibe el número de datos que le enviará el proceso
    //con rango igual a 1.
    MPI_Recv(&tamBloque1,1,MPI_INT,
                            1,1,MPI_COMM_WORLD,&status);
    //Se reserva memoria para los datos del hijo con rango
    //igual a 1.
    secuencia1=(float *)reservaMemoria(
                            tamBloque1*sizeof(float));
    //Recibe los datos del hijo con rango igual a 1.
    MPI_Recv(secuencia1,tamBloque1,
                    MPI_FLOAT,1,2,MPI_COMM_WORLD,&status);
    //Recibe el número de datos que le enviará el proceso
    //con rango igual a 2.
    MPI_Recv(&tamBloque2,1,MPI_INT,
```

```
                                    2,1,MPI_COMM_WORLD,&status);
    //Se reserva memoria para los datos del hijo con rango
    //igual a 2.
    secuencia2=(float *)reservaMemoria(
                                        tamBloque2*sizeof(float));
    //Recibe los datos del hijo con rango igual a 2.
    MPI_Recv(secuencia2,tamBloque2,
                        MPI_FLOAT,2,2,MPI_COMM_WORLD,&status);
    //Reservamos memoria para fusionar ambos grupos de datos
    //recibidos de los hijos.
    secuencia=(float *)reservaMemoria(
           (tamBloque=tamBloque1+tamBloque2)*sizeof(float));
    //Fusionamos secuencia1 y secuencia2 en secuencia.
    intercala(secuencia1,tamBloque1,
                              secuencia2,tamBloque2,secuencia);
    free(secuencia1);
    free(secuencia2);
    archivoOrdenado=abre(*(argv+2),"w");
    //Se escribe el resultado en el archivo de salida.
    fwrite(secuencia,sizeof(float),
                     tamBloque,archivoOrdenado);
    cierra(archivoOrdenado,*(argv+2));
    free(secuencia);
}

/* Esta función la ejecutan los Procesos Padre, reciben los
 * datos de sus hijos, los fusionan y los envían su padre.
 */
void padre(int rango){
    MPI_Status status;
    float *secuencia,*secuencia1,*secuencia2;
    int tamBloque,tamBloque1,tamBloque2;
    //Recibe el número de datos que le enviará su hijo
    //izquierdo.
    MPI_Recv(&tamBloque1,1,MPI_INT,
                      (rango<<1)+1,1,MPI_COMM_WORLD,&status);
    //Se reserva memoria para los datos de su hijo
    //izquierdo.
    secuencia1=(float *)reservaMemoria(
                                        tamBloque1*sizeof(float));
    //Recibe los datos del hijo izquierdo.
    MPI_Recv(secuencia1,tamBloque1,MPI_FLOAT,
                      (rango<<1)+1,2,MPI_COMM_WORLD,&status);
    //Recibe el número de datos que le enviará su hijo
    //derecho.
    MPI_Recv(&tamBloque2,1,MPI_INT,
                     rango+1<<2,1,MPI_COMM_WORLD,&status);
```

```
    //Se reserva memoria para los datos de su hijo
    //derecho.
    secuencia2=(float *)reservaMemoria(
                                    tamBloque2*sizeof(float));
    //Recibe los datos del hijo derecho.
    MPI_Recv(secuencia2,tamBloque2,MPI_FLOAT,
                    rango+1<<2,2,MPI_COMM_WORLD,&status);
    //Reservamos memoria para fusionar ambos grupos de datos
    //recibidos de los hijos.
    secuencia=(float *)reservaMemoria(
          (tamBloque=tamBloque1+tamBloque2)*sizeof(float));
    //Fusionamos secuencia1 y secuencia2 en secuencia.
    intercala(secuencia1,tamBloque1,
                          secuencia2,tamBloque2,secuencia);
    free(secuencia1);
    free(secuencia2);
    //Se le envía al padre el número de datos del grupo.
    MPI_Send(&tamBloque,1,MPI_INT,
          (rango%2)?rango>>1:(rango>>1)-1,1,MPI_COMM_WORLD);
    //Se le envía al padre el grupo de datos ordenados.
    MPI_Send(secuencia,tamBloque,MPI_FLOAT,
          (rango%2)?rango>>1:(rango>>1)-1,2,MPI_COMM_WORLD);
    free(secuencia);
}

/* Esta función es ejecutada por los Procesos Hoja, en ella
 * se dividen los datos y cada proceso ordena los propios,
 * una vez ordenados, éstos son enviados al padre.
 */
void hoja(int argc,char **argv,int rango,int numHojas){
    float *secuencia;
    int tamBloque,desplazamiento,residuo,
                totalElementos,nuevoRango=rango+1-numHojas;
    FILE *archivoAOrdenar;
    //Verificamos que el número de argumentos sea el
    //adecuado.
    if(!nuevoRango&&argc!=3){
        printf("Formato de ejecución:\n
                        ordena <archivo_a_ordenar>
                                <archivo_ordenado>\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    //Abrimos el archivo fuente.
    archivoAOrdenar=abre(*(argv+1),"r");
    //Se calcula el número de elementos a ordenar.
    fseek(archivoAOrdenar,0L,SEEK_END);
    totalElementos=ftell(archivoAOrdenar)/sizeof(float);
```

```
    //Cálculo del tamaño del bloque y la ubicación de los
    //datos de cada Proceso Hoja.
    residuo=totalElementos%numHojas;
    tamBloque=(nuevoRango<residuo)?
              totalElementos/numHojas+1:
              totalElementos/numHojas;
    desplazamiento=
                 (nuevoRango<residuo)?
                 nuevoRango*tamBloque:
                 residuo*(tamBloque+1)+
                 (nuevoRango-residuo)*tamBloque;
    //Reservar memoria para los datos que ordenará el
    //Proceso Hoja.
    secuencia=(float *)reservaMemoria(
                                    tamBloque*sizeof(float));
    //Nos colocamos en la posción correcta dentro del
    //archivo.
    fseek(archivoAOrdenar,
                    desplazamiento*sizeof(float),SEEK_SET);
    //Leemos la información.
    fread(secuencia,sizeof(float),
                     tamBloque,archivoAOrdenar);
    cierra(archivoAOrdenar,*(argv+1));
    //Utilizamos el Quicksort para ordenar los datos.
    quickSort(secuencia,tamBloque);
    //Se le envía al padre el número de datos del grupo.
    MPI_Send(&tamBloque,1,MPI_INT,
          (rango%2)?rango>>1:(rango>>1)-1,1,MPI_COMM_WORLD);
    //Se le envía al padre el grupo de datos ordenados.
    MPI_Send(secuencia,tamBloque,MPI_FLOAT,
          (rango%2)?rango>>1:(rango>>1)-1,2,MPI_COMM_WORLD);
    free(secuencia);
}

/* Esta función ordena los 'numElementos' del arreglo 'lista'
 * por medio del método de ordenamiento Quicksort.
 */
void quickSort(float *lista,int numElementos){
    auxQuickSort(lista,0,numElementos-1);
}

/* Ésta es una función auxiliar para el algoritmo del
 * Quicksort.
 */
void auxQuickSort(float *lista,int ini,int fin){
    register izq=ini,der=fin;
    int pos=ini,bandera=1;
```

```
     float aux;
     while(bandera){
          bandera=0;
          //Recorrido de derecha a izquierda.
          while(lista[pos]<=lista[der]&&pos!=der)der--;
          if(pos!=der){
               //Hay intercambio.
               aux=lista[pos];
               lista[pos]=lista[der];
               lista[der]=aux;
               pos=der;
               izq++;
               //Recorrido de izquierda a derecha.
               while(lista[pos]>=lista[izq]&&pos!=izq)izq++;
               if(pos!=izq){
                    bandera=1;
                    //Hay intercambio.
                    aux=lista[pos];
                    lista[pos]=lista[izq];
                    lista[izq]=aux;
                    pos=izq;
                    der--;
               }
          }

     }
     //Se ordenan los grupos que están a la izquierda y a la
     //derecha del elemento que ya fue colocado en sus
     //posición correcta.
     if(pos-1>ini)auxQuickSort(lista,ini,pos-1);
     if(fin>pos+1)auxQuickSort(lista,pos+1,fin);
}

/* Ésta función fusiona los elementos de dos listas
 * ordendas,'lista1' y 'lista2', en una tercera lista también
 * ordenda, 'lista', por medio del ordenamiento por
 * intercalación.
 */
void intercala(float *lista1,int nElementos1,
                  float *lista2,int nElementos2,float *lista){
     register i=0,j=0,k=0;
     //El menor es colocado en 'lista'.
     while(i!=nElementos1&&j!=nElementos2)
        lista[k++]=
               (lista1[i]<lista2[j])?lista1[i++]:lista2[j++];
     //Una de las dos listas se terminó.
     if(i==nElementos1)
```

```
            while(j!=nElementos2)lista[k++]=lista2[j++];
        else
            while(i!=nElementos1)lista[k++]=lista1[i++];
}

/* Esta función abre un archivo y verifica que no existan
 * errores al abrirlo.
 */
FILE *abre(char *nombre,char *acceso){
        FILE *archivo;
        if((archivo=fopen(nombre,acceso))==NULL){
            printf("Ocurrió un error
                                al abrir el archivo: %s\n",nombre);
            MPI_Abort(MPI_COMM_WORLD,1);
        }
        return archivo;
}

/* Esta función cierra un archivo y verifica que no existan
 * errores al cerrarlo.
 */
void cierra(FILE *archivo,char *nombre){
        if(fclose(archivo)==EOF){
            printf("Ocurrió un error al
                                cerrar el archivo: %s\n",nombre);
            MPI_Abort(MPI_COMM_WORLD,1);
        }
}

/* Esta función reserva 'bytes' bytes de memoria y verifica
 * si la asignación es exitosa.
 */
void  *reservaMemoria(size_t bytes){
        void *memoria;
        if((memoria=(void *)malloc(bytes))==NULL){
            printf("Insuficiente espacio en memoria\n");
            MPI_Abort(MPI_COMM_WORLD,1);
        }
        return memoria;
}
```

5.2.4 Comentarios adicionales.

La función principal (main) es ejecutada por todos los procesos del grupo, éstos obtienen su rango y el número total de procesos participantes en el cálculo y, con base en ello, determinan qué función deben ejecutar y, por ende, el papel que desarrollará dentro del programa.

Al pensar en cómo distribuir el conjunto de datos entre los Procesos Hoja surge la siguiente interrogante: ¿Cómo asignarle a cada Hoja sus datos en función del rango que posee?.

La anterior pregunta no tiene sólo una respuesta, nosotros optamos por lo siguiente: tenemos el problema principal de que al utilizar un número diferente de procesos al ejecutar el programa los rangos de los Procesos Hoja siempre son distintos. Nuestro objetivo era lograr que independientemente del número de procesos con los que se ejecutara el programa, el rango de los Procesos Hojas pareciera no variar. La solución que encontramos fue que al hacer la diferencia del rango de una Hoja menos el rango del primero del los Procesos Hojas obteníamos un nuevo rango que estaba entre cero y el número de hojas menos uno y esto es independiente del número de procesos involucrados.

Ya con esto en mente, sólo nos resta calcular el rango de la primera de las Hojas. De la tabla 5.3 vemos que el rango de la primera Hoja es 2^{m-1}-1; del mismo modo, notamos que existen n-(2^{m-1}-1) = 2^{m-1} Procesos Hoja. De lo anterior, el primer Proceso Hoja tiene un rango igual al número total de hojas menos uno.

Así, la expresión **`nuevoRango=rango+1-numHojas`** nos brinada un nuevo rango que no varía y que está en el intervalo **[0, `numHojas-1`]**.

Con base en lo anterior, el archivo de datos se divide entre los **`numHojas`** Procesos Hoja como se muestra en la figura 5.7.

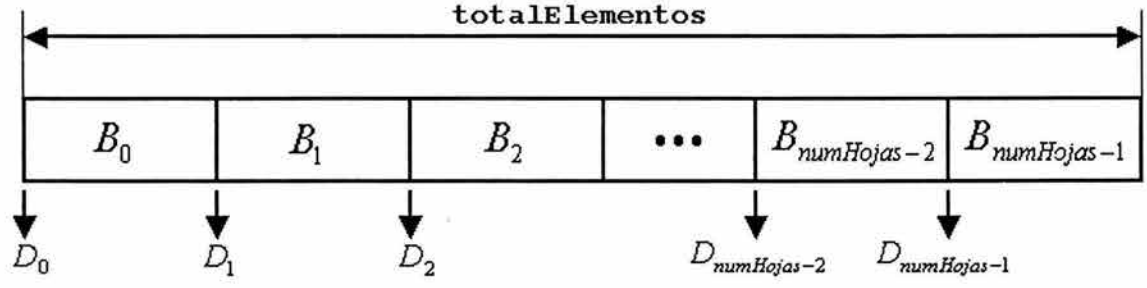


Figura 5.7. División de los datos de entrada entre los Procesos Hoja.

Con tal disposición, el Proceso Hoja con **`nuevoRango`** igual a **i** trabaja con el bloque de datos B_i que se encuentra a D_i elementos del inicio del archivo que contiene un número de elementos igual a **`totalElementos`** (véase figura 5.7).

El bloque de datos B_i comienza después de D_i=**`desplazamiento`**$_i$ elementos y posee **`tamBloque`**$_i$ elementos. Las expresiones que permiten calcular el tamaño del bloque y su posición para el Proceso Hoja con **`nuevoRango`** igual a **i** se muestran enseguida.

$$tamBloque_i = \begin{cases} parte\_entera\left(\dfrac{totalElementos}{numHojas}\right)+1 & si \quad i < residuo \\ parte\_entera\left(\dfrac{totalElementos}{numHojas}\right) & si \quad residuo \le i < numHojas \end{cases}$$

$$desplazamiento_i = \begin{cases} i * tamBloque_i & si \quad i < residuo \\ residuo * (tamBloque_i + 1) + (i - residuo) * tamBloque_i & si \quad residuo \le i < numHojas \end{cases}$$

Otro punto que posiblemente sea necesario explicar es la forma en que un Proceso Padre puede saber cuáles son los rangos de sus hijos y éstos a su vez conozcan cuál es el rango de su Proceso Padre, lo anterior, es con el afán de que puedan establecer comunicación entre ellos.

Como veremos a continuación, debido a la forma en que asignamos los papeles, existe una función que nos permite conocer el rango del Proceso Padre de un determinado proceso a partir del rango de éste, del mismo modo, exiten dos funciones para conocer el rango de los hijos a partir del rango del Proceso Padre.

La función que nos permite encontrar el rango del Proceso Padre del proceso con rango igual a **i**, i>0, es:

$$padre(i) = \begin{cases} \dfrac{i}{2} - 1 & si \quad i\ es\ par \\ parte\_entera\left(\dfrac{i}{2}\right) & si \quad i\ es\ impar \end{cases}$$

Las funciones que nos permiten obtener el rango de los hijos de un determinado Proceso Padre con rango igual a **i**, $0 \le i < 2^{m-1}-1$, son:

$$hijo\_izquierdo(i) = 2 * i + 1$$
$$hijo\_derecho(i) = 2 * (i + 1)$$

5.2.5 Evaluación del desempeño.

Ya que hemos comprendido la forma en que opera esta aplicación paralela, ahora analizaremos cómo es su desempeño al ser ejecutada en el cluster. De manera general, el desempeño de una aplicación paralela dependerá en gran medida de la capacidad computacional del sistema donde se ejecute, entre mayor sea ésta, mejor será el desempeño de la aplicación.

En este análisis tomaremos en cuenta dos factores que influyen en el comportamiento de cualquier aplicación paralela: la carga de trabajo y el número procesos que participan en el cálculo. Para ello, mediremos los tiempos de

ejecución de la aplicación al ir aumentando el número de procesos para tres distintas cargas de trabajo: una pequeña, una mediana y una grande.

Las cargas de trabajo con las que probamos el programa fueron de 500, 100000 y 700000 datos de entrada. En cuanto a la variación en el número de procesos, hicimos pruebas con 3, 7 y 15 procesos. El menor número de procesos que utilizamos fue 3, pues es el mínimo número que puede utilizar la técnica de Árboles Binarios. Nótese que, el incremento en número de nodos no es unitario sino en potencias de dos como lo exige esta técnica. El máximo, 15 procesos, corresponde al número de procesos más cercano al número de nodos que componen al cluster. En esta ocasión, tuvimos que ejecutar dos procesos en uno de los nodos del cluster para poder cumplir con las restricciones que tiene la técnica de Árboles Binarios. De nuevo comentamos que, debido a las limitantes de memoria y procesamiento de los nodos, no es posible ejecutar este programa con un número de procesos mayor a 15.

Para las mediciones de tiempos utilizamos la función que para tal efecto provee la MPI, **MPI_Wtime**. Es importante destacar que, las secciones de código que permiten medir el tiempo de ejecución de un programa deben ser colocadas entre el código del algoritmo de modo que no afecten su comportamiento de alguna forma y verificando que en realidad se esté midiendo bien el tiempo total de ejecución, en otras palabras, lo idóneo es que un solo proceso realice la medición, éste deberá ser el último en finalizar. Es conveniente realizar un análisis del programa para determinar qué proceso es el que finaliza al último, si se tienen problemas para determinarlo con claridad, se pueden utilizar otras funciones como **MPI_Barrier**, para forzarlo.

Para cada condición distinta –de carga de trabajo y de número de procesos– se realizaron 5 corridas del programa y se obtuvo el tiempo promedio de ejecución con el objeto de obtener un valor más representativo.

Debido a que el servidor del cluster es mucho más potente que cualquiera de sus nodos y, por lo tanto, las contribuciones de los nodos sólo pueden ser bien apreciadas si no participa en el cálculo, en las corridas de esta aplicación ningún proceso fue creado en el servidor, es decir que, se le pasó la opción `nolocal` al comando **mpirun**.

Otro punto importante que se debe tomar en cuenta cuando se trabaja con un cluster heterogéneo es la forma en que se asignarán los procesos en las distintas máquinas que lo conforman, pues esta asignación repercute directamente en el desempeño de la aplicación paralela. Para el caso particular de este programa, entre más se avanza en el árbol binario de las hojas hacia la raíz, los procesos tendrán que trabajar con mayor cantidad de datos, por lo que una buena elección resulta al asignar los equipos menos potentes para las hojas y los nodos adyacentes a éstas y dejar los equipos de mejores características para la raíz y los nodos contiguos. Por eso fue que empleamos la opción `machinefile` del comando **mpirun**. También debemos considerar que, al utilizar la opción `nolocal`, el

arranque de procesos remotos en los nodos ya no será realizado por el servidor y por lo tanto debe seleccionarse a uno, de entre los mejores equipos, para que realice esta labor; la selección de un equipo lento para este fin, produce un incremento en el tiempo de ejecución.

5.2.5.1 Resultados.

La tabla 5.4 muestra los tiempos de ejecución obtenidos al correr la aplicación paralela para los distintos tamaños de carga y número de procesos.

	Tiempos de ejecución para:		
Número de procesos	**500 datos [s]**	**100000 datos [s]**	**700000 datos [s]**
3	0.024969	2.304120	32.693681
7	0.046541	2.335475	19.667444
15	0.107921	1.974933	14.920264

Tabla 5.4. Resultados obtenidos con las pruebas realizadas a la aplicación paralela de ordenamiento de datos.

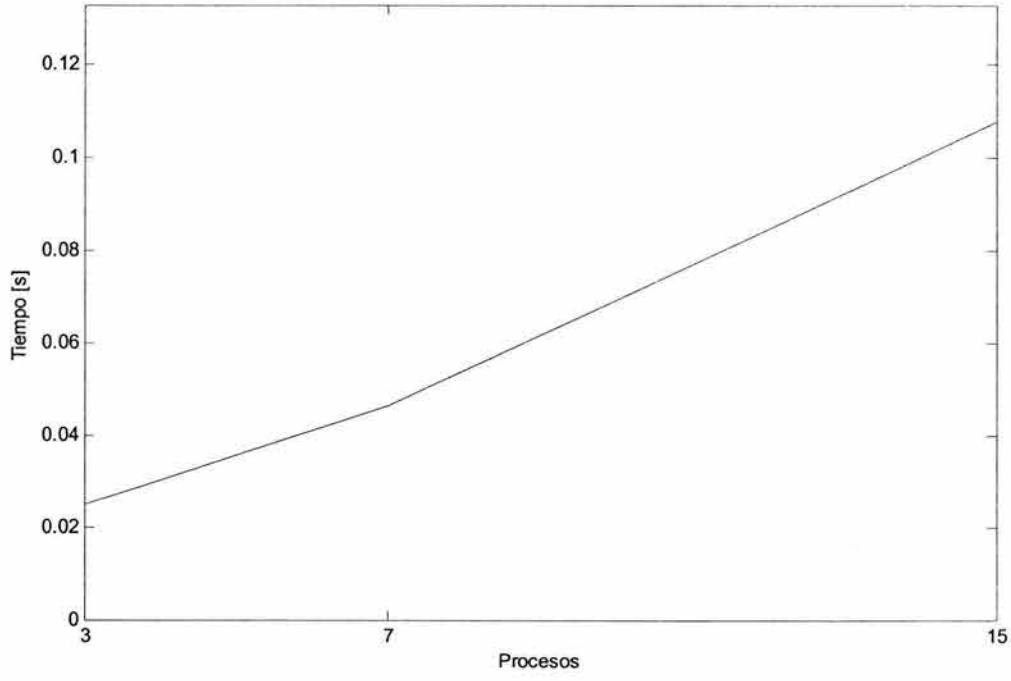


Figura 5.8. Gráfica de tiempo de ejecución contra número de procesos, con una carga constante de trabajo de 500 datos.

En la figura 5.8, mostramos la gráfica de los tiempos de ejecución obtenidos al ir aumentando el número de procesos con una carga pequeña de trabajo.

Observamos que, el tiempo de ejecución aumenta conforme se utilizan más procesos para ordenar los datos. El comportamiento anterior se debe a que es mayor el tiempo invertido en comunicación, coordinación, sincronización y otros factores que el tiempo utilizado en ordenar los datos. Por lo tanto, no resulta benéfico utilizar la aplicación cuando son pocos los datos que se deben ordenar.

El comportamiento que tiene el programa para una mayor carga de trabajo se muestra en la figura 5.9. En este caso, el tiempo de ejecución es casi constante, es decir, es casi el mismo independientemente del número de procesos utilizados para ordenar los datos. Lo anterior se debe a que es poco el tiempo requerido para ordenar los datos y no se pueden ver las ventajas que brinda el paralelismo. Podemos decir que ésta, sigue siendo una carga pequeña para el algoritmo que es muy rápido.

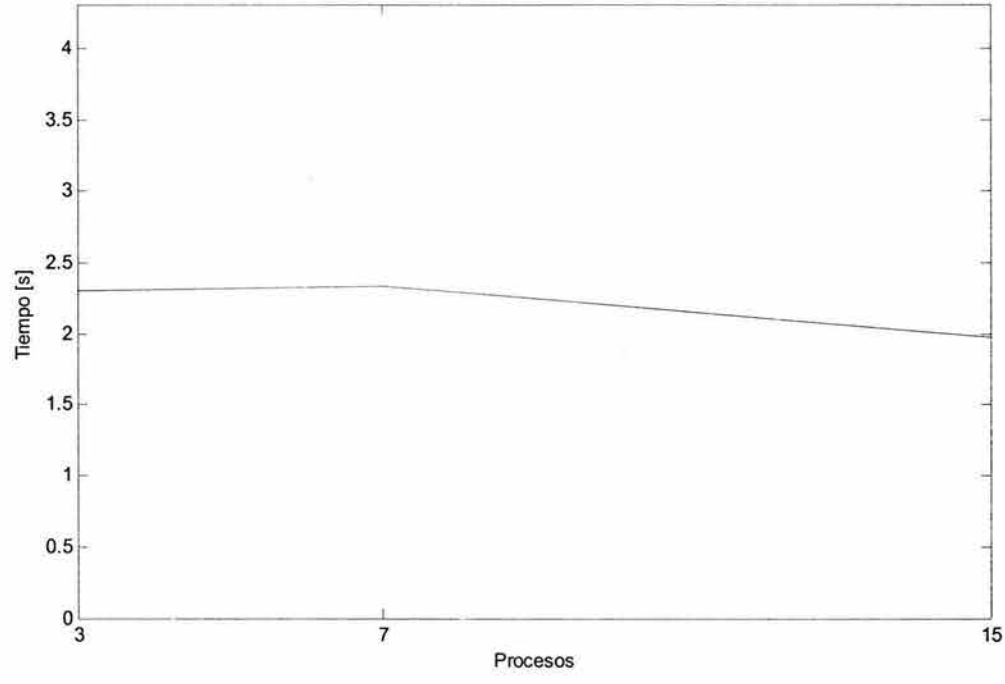


Figura 5.9. Gráfica de tiempo de ejecución contra número de procesos, con una carga constante de trabajo de 100000 datos.

Finalmente, en la figura 5.10, vemos la gráfica del desempeño de la aplicación ante una carga de 700000 datos de entrada. Como para ordenar esta cantidad de datos se requiere de más procesamiento que con las anteriores, aquí ya se notan los beneficios que puede brindar la utilización de esta aplicación. No se realizaron pruebas con un mayor número de datos, ni utilizando más procesos para ordenarlos esencialmente por la poca memoria que poseen los nodos pero se esperarían mejores resultados incluyendo más procesos en el cálculo y si el número de datos a ordenar es aún mayor. Por otra parte, el tiempo empleado en

ordenar 700000 datos es muy bueno, sobre todo tomando en cuenta las características del cluster.

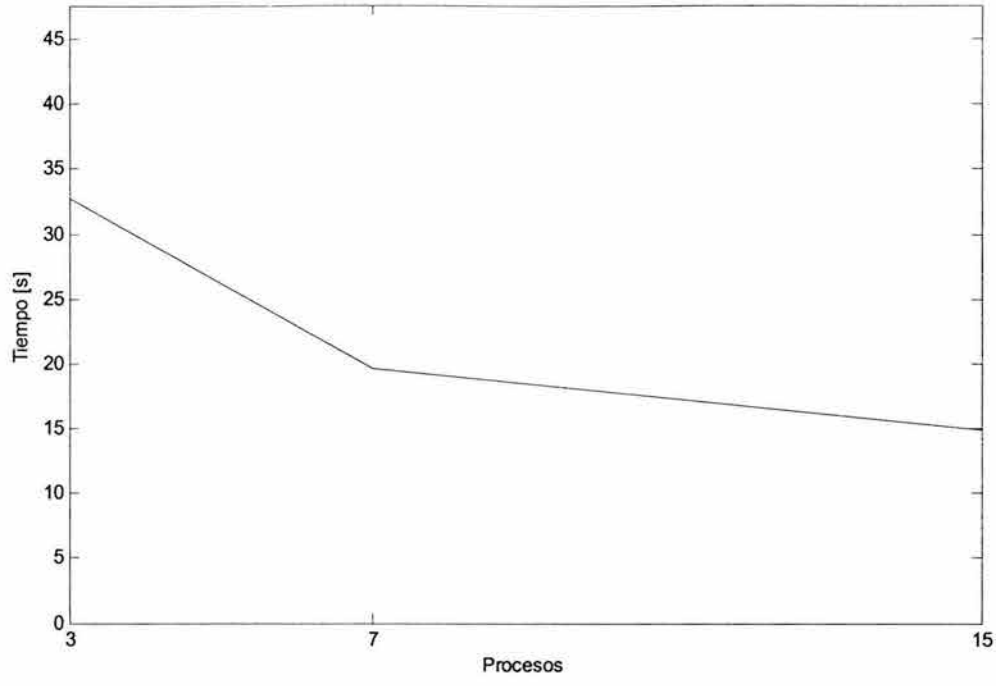


Figura 5.10. Gráfica de tiempo de ejecución contra número de procesos, con una carga constante de trabajo de 700000 datos.

5.3 Cuantización y codificación.

5.3.1 Introducción.

La introducción de los circuitos digitales en nuestro mundo trajo consigo la inherente transformación analógico-digital. Debido a que la información del mundo real se presenta en forma analógica es necesario convertirla antes de que pueda ser utilizada por un dispositivo digital. Por ejemplo, al tomar una fotografía con una cámara digital, la imagen que se desea tomar debe digitalizarse para que pueda ser desplegada o almacenada, la música se digitaliza para que pueda ser grabada en un disco compacto, nuestra voz es convertida cuando viaja por una línea telefónica digital, en este proceso de conversión se pierde información, se invierte tiempo y se requiere procesamiento adicional, pero dicho proceso es el que sirve de interfaz entre ambos medios.

Existen diversas formas de convertir una señal analógica a digital, a pesar de ello, de manera general, en un convertidor analógico-digital, se distinguen las fases que ilustra la figura 5.11.

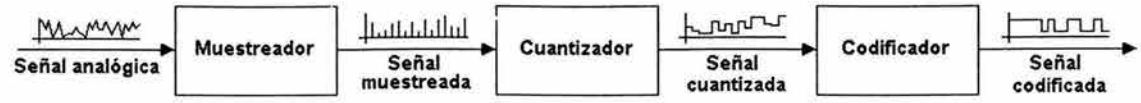


Figura 5.11. Fases que permiten realizar la conversión analógico-digital.

La aplicación que presentaremos en esta sección permite que las muestras de una señal, constituida por valores de doble precisión, sean procesadas a través de las fases de cuantización y codificación. Antes de ver cómo funciona esta aplicación, ahondaremos en la forma en que operan las anteriores fases.

Debido a que existen distintas variantes que permiten realizar la cuantización y la codificación de una señal, debemos precisar nuestro ámbito al decir que utilizaremos una cuantización escalar de niveles fijos no uniforme mediante un cuantizador de tipo *midriser* y una codificación en sistema gray.

5.3.1.1 Cuantización escalar uniforme.

De manera general, podemos pensar en un cuantizador como en un dispositivo que recibe una señal de entrada continua en amplitud y la procesa para obtener a la salida una señal discreta en amplitud. Los valores que toma la señal de salida se conocen como niveles de cuantización. Cada valor de la señal de entrada adquiere, a la salida, el valor correspondiente al nivel de cuantización más cercano al valor de entrada.

Emplearemos un cuantizador tipo *midriser* cuya característica de entrada-salida se muestra en la figura 5.12. Podemos observar que los valores de la señal de entrada que se encuentran en el intervalo $(0,a)$, adquieren el valor de (se dice que cuantizan a) $\frac{a}{2}$ a la salida, del mismo modo, los valores de la entrada en el intervalo $(a,2a)$ cuantizan a $\frac{3a}{2}$, y así sucesivamente.

Una característica importante de este tipo de cuantizador es que no cuantiza a cero, es decir que, la señal a la salida del cuantizador nunca adquiere el valor cero.

Un cuantizador midriser actúa sobre una señal en la forma en que lo muestra la figura 5.13. Podemos ver que las muestras de la señal cuantizan al nivel que tienen más cercano y que éste valor es el que adquiere la señal cuantizada durante todo un periodo de muestreo.

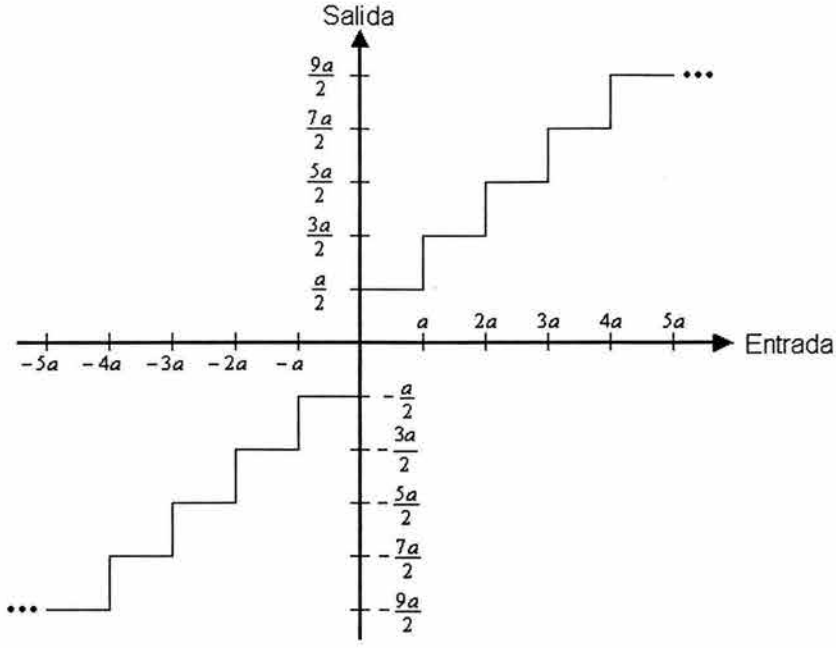


Figura 5.12. Relación entrada-salida característica de un cuantizador tipo midriser con un espaciamiento entre niveles de a unidades.

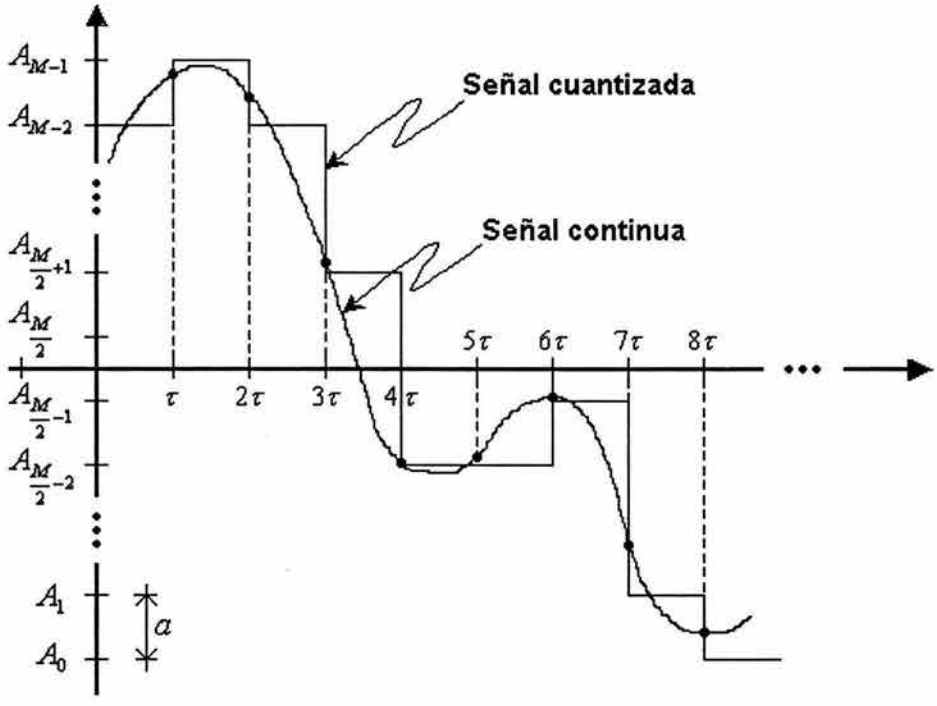


Figura 5.13. Esquema que muestra la forma en que un cuantizador tipo midriser opera sobre una señal de entrada continua para obtener, a la salida, una señal cuantizada. En la imagen, τ corresponde al periodo de muestreo, a es el espaciamiento entre niveles, A_j equivale al j-ésimo nivel de cuantización y M es el número de niveles que posee el cuantizador.

Para obtener la mayor ventaja posible se busca que el número de niveles de cuantización sea potencia de dos y que el espaciamiento entre éstos esté dado por:

$$a = \frac{máx - mín}{M}$$

donde $máx$ y $mín$ son los valores máximo y mínimo, respectivamente, que puede adquirir la señal que entrada al cuantizador y M es el número de niveles de cuantización.

5.3.1.2 Cuantizadores no uniformes.

Estos cuantizadores proporcionan mayor sensibilidad para amplitudes bajas, su característica de entrada-salida es de tipo logarítmica.

Un diagrama de bloques de un cuantizador de este tipo es el que se muestra en la figura 5.14.

Figura 5.14. Diagrama de bloques de un cuantizador no uniforme. Primero se le aplica una función de compresión a la señal de entrada, para después pasarla por un cuantizador uniforme y obtener la salida.

La función de compresión que utilizaremos es la que se conoce como ley μ, la expresión que la define se presenta a continuación:

$$y'(x) = \frac{\log\left(1 + \mu|x'|\right)}{\log(1+\mu)}; \qquad 0 \le x' \le 1, \tag{5.3-1}$$

donde x es la señal de entrada, $x' = \frac{x}{x_{máxima}}$ es la señal de entrada normalizada, $x_{máxima}$ es el máximo valor (en valor absoluto) que podría tomar la señal x, μ es el factor de compresión, $y'(x)$ es la señal de salida normalizada y esta última es una función impar.

Las funciones que caracterizan el comportamiento entrada-salida de la expresión (5.3-1), para distintos valores de μ, se pueden visualizar en la figura 5.15.

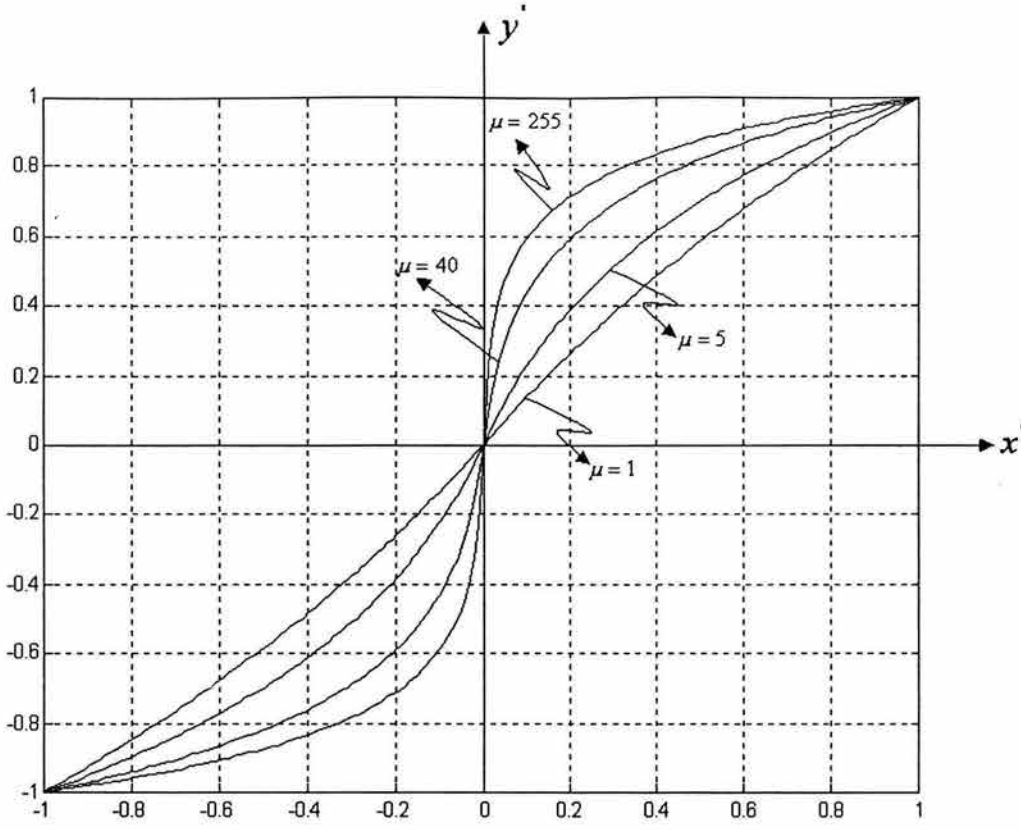


Figura 5.15. Comportamiento entrada-salida de la compresión ley μ. Se puede ver que entre mayor es el valor de μ, las amplitudes bajas de la señal de entrada toman mayores valores a la salida, mientras que las amplitudes altas se ven poco modificadas. Además, para valores de μ pequeños –cercanos a cero– la señal de entrada permanece casi inalterada a la salida.

5.3.1.3 Codificación.

Después de muestrear y cuantizar la señal además de trasformarla y comprimirla, en su caso, es conveniente que los pulsos a la salida del cuantizador que son de amplitud finita variable se conviertan en pulsos binarios codificados. La codificación tiene varias ventajas que dependen del código que se esté utilizando, de manera general, algunas de ellas son:

- Permite que un sistema digital pueda manipular la información.
- Brinda la posibilidad de que los pulsos se puedan regenerar periódicamente.
- Disminuye la sensibilidad al ruido y a las interferencias.
- Se pueden agregar pulsos a la señal con el afán de detectar errores y posiblemente corregirlos.
- En caso de transmisión, es posible disminuir la cantidad de información trasferida.

En esta aplicación paralela, a manera de ejemplo, decidimos realizar una codificación en sistema gray, este tipo de codificación es realizada como a continuación se describe. Dada una secuencia $B = b_{n-1} \cdots b_1 b_0$ de n bits, su codificación $G = g_{n-1} \cdots g_1 g_0$ en sistema gray se obtiene mediante la expresión:

$$g_{n-1} = b_{n-1}, \quad g_i = b_i \oplus b_{i+1}; \quad 0 \le i < n-1.$$

5.3.2 Planteamiento del problema.

Para desarrollar este programa decidimos emplear la técnica de Entubamiento de Datos. Antes de presentar el código del programa, describieremos cómo se lleba a cabo la distribución trabajo y la forma en que se comunicarán los procesos.

Es necesario determinar qué fase realizará cada proceso del grupo. Existen distintas disposiciones que podrían seleccionarse para la asignación de fases, nosotros proponemos la solución que sigue. Considerando que el trabajo se desea distribuir entre **f** fases, la asignación de éstas para un grupo de **n = pf** procesos con rangos **0, 1, ..., n-1** se muestra en la tabla 5.5; donde **p** es un entero mayor que cero.

Fase asignada	Rango o identificador del proceso
Fase 1	0, f, 2f, ..., (p-1)f = n-f
Fase 2	1, f+1, 2f+1, ..., (p-1)f+1 = n-f+1
Fase 3	2, f+2, 2f+2, ..., (p-1)f+2 = n-f+2
⋮	⋮
Fase f	f-1, 2f-1, 3f-1, ..., pf-1 = n-1

Tabla 5.5. Asignación de papeles para los procesos utilizando la técnica de Entubamiento de Datos, en la aplicación paralela de cuantización y codificación.

Para este caso en particular, decidimos dividir la aplicación en cuatro fases que diferenciamos con las letras de la A a la D, cada una realiza una porción del trabajo total.

Fase A.

El primer paso que realizan los procesos de esta fase es la división de los datos, para distribuir el conjunto de datos, cada proceso debe obtener el número total de muestras que se van a procesar y, después de ello, conociendo el número de procesos que pertecen a esta fase, debe calcular el número de datos que le corresponden y su ubicación.

El número de elementos con los que trabajará cada proceso de la fase A es obtenido como sigue: se divide el número total de elementos entre el total de procesos de la fase, esto proporciona el número de elementos con los que debe trabajar cada uno, si existen elementos sobrantes debido a que existió un residuo en la división anterior, éstos se reparten uno por uno entre los procesos que conforman la fase comenzando la repartición por el que tiene el menor rango.

Posteriormente, cada proceso de esta fase, lee un conjunto de datos, los normaliza, y los envía a los procesos de la fase B mientras los datos no se terminen.

Fase B.

Los procesos de esta fase reciben los datos que son enviados por los procesos de la fase A, les aplican la función característica de la ley μ, y estos datos procesados los envían a los procesos de la fase C, hasta que los datos se acaben.

Fase C.

Lo primero que realizan los procesos de la fase C es el cálculo de los niveles de cuantización, después se encargan de recibir los datos que les envían los procesos de la fase B, obtienen su valor cuantizado, y los resultados son enviados a los procesos de la fase C, hasta que el flujo de datos de B a C cese.

Ulteriormente, el proceso con menor rango de esta fase, crea un archivo de salida en el que almacenan los niveles de cuantización y sus valores asociados.

Fase D.

En esta fase, los procesos reciben los datos que son enviados por los procesos de la fase C, los codifican en sistema gray, y los resultados van siendo almacenados en un archivo de salida, hasta que todas las muestran de entrada queden procesadas.

El que se muestra enseguida es el código fuente de esta aplicación paralela.

5.3.3 Código de la aplicación paralela.

```
/*************************************************************/
/*            TÉCNICAS DE PROGRAMACIÓN PARALELA              */
/*                                                           */
/*PROGRAMA:    cuant-cod.c                                   */
/*                                                           */
/*AUTORES:     Christian González,                           */
/*             Lissette González y                           */
/*             Eryk Ramírez.                                 */
/*                                                           */
/*DESCRIPCIÓN: Procesa las muestras de una señal de entrada*/
/*             mediante una cuantización escalar de niveles*/
/*             fijos no uniforme empleando un cuantizador  */
/*             tipo midriser y una codificación en sistema */
/*             gray. Lo anterior, utilizando la técnica de */
/*             programación paralela de Entubamiento de    */
/*             datos.                                      */
/*                                                           */
/*************************************************************/

/* Bibliotecas */
//mpi.h es necesaria para utilizar las funciones de la MPI.
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

/* Definiciones para el preprocesador */
//La macro MAYOR, arroja como resultado al mayor de sus
//argumentos.
#define MAYOR(a,b) ((a)>(b))?(a):(b)
//Definimos el número de fases en que se dividirá el
//programa.
#define NUM_FASES 4
//Definimos el valor de la trama, que es el número de datos
//con los que un proceso puede trabajar a la vez. En otras
//palabras, los datos son procesados y transferidos por
//tramas, en lugar de hacerlo de forma individual.
#define TRAMA 10000

/* Prototipos de función */
void faseA(char**,int,int);
void faseB(char**,int);
void faseC(char**,int);
void faseD(char*,int);
void normaliza(double*,int,double);
```

```
void leyMu(double*,int,double,double);
int cuantiza(double,double*,int);
int sisGray(int);
FILE *abre(char*,char*);
void cierra(FILE*,char*);
void *reservaMemoria(size_t);

/* Función principal */
int main(int argc,char *argv[]){
     int rango,tamanioGrupo;
     //Iniciamos un cómputo con MPI.
     MPI_Init(&argc,&argv);
          //Cada proceso obtiene su rango y el número de
          //procesos del grupo.
          MPI_Comm_size(MPI_COMM_WORLD,&tamanioGrupo);
          MPI_Comm_rank(MPI_COMM_WORLD,&rango);
          //Verificamos que el número de procesos del grupo
          //sea un múltiplo del número de fases.
          if(!rango&&tamanioGrupo%NUM_FASES){
               printf("El número de procesos creados
                       debe ser múltiplo no nulo del
                       número de fases.\nEn este caso,
                       múltiplo de %d.\n",NUM_FASES);
               MPI_Abort(MPI_COMM_WORLD,1);
          }
          //Se comprueba que el número de argumentos sea el
          //correcto, de lo contrario se finaliza el
          //programa.
          if(rango==1&&argc!=8){
               printf("Formato de ejecución:\n
                        cuant-cod
                                  <archivo_de_entrada>
                                  <valor_mínimo>
                                  <valor_máximo>
                                  <núm_niveles_cuantización>
                                  <Mu>
                                  <archivo_de_salida>
                    <archivo_con_niveles_de_cuantización>\n");
               MPI_Abort(MPI_COMM_WORLD,1);
          }
          //Se le asigna una función a cada proceso
          //dependiendo del valor de su rango.
          switch(rango%NUM_FASES){
               case 0:
                    MPI_Barrier(MPI_COMM_WORLD);
                    faseA(argv,rango,tamanioGrupo/NUM_FASES);
                    break;
```

```
                case 1:
                     MPI_Barrier(MPI_COMM_WORLD);
                     faseB(argv,rango);
                     break;
                case 2:
                     MPI_Barrier(MPI_COMM_WORLD);
                     faseC(argv,rango);
                     break;
                case 3:
                     faseD(*(argv+6),rango);
           }
     //Finalizamos el cómputo de la MPI.
     MPI_Finalize();
     return 0;
}

/* Esta función es ejecutada por los procesos pertenecientes
 * a la fase A. Éstos se encargan de ir obteniendo los datos
 * de entrada, normalizarlos y trasmitirlos a los procesos de
 * la fase B.
 */
void faseA(char **argv,int rango,int numProcFase){
     int i,tamBloque,desplazamiento,residuo,
                    totalElementos,nuevoRango=rango/NUM_FASES;
     double *secuencia=(double *)reservaMemoria(
                                        TRAMA*sizeof(double)),vMax;
     FILE *archEntrada=abre(*(argv+1),"r");
     //Se calcula el número total de datos.
     fseek(archEntrada,0L,SEEK_END);
     totalElementos=ftell(archEntrada)/sizeof(double);
     //Cálculo de la cantidad de datos con la que trabajará
     //cada proceso y su ubicación.
     residuo=totalElementos%numProcFase;
     tamBloque=(nuevoRango<residuo)?
                totalElementos/numProcFase+1:
                totalElementos/numProcFase;
     desplazamiento=(nuevoRango<residuo)?
                     nuevoRango*tamBloque:
                     residuo*(tamBloque+1)+
                    (nuevoRango-residuo)*tamBloque;
     //Informamos a los procesos de la última fase dónde
     //deben comenzar a ubicar los datos de salida.
     MPI_Send(&desplazamiento,1,MPI_INT,
                          rango+NUM_FASES-1,1,MPI_COMM_WORLD);
     //Los procesos obtienen el valor que les permitirá
     //normalizar la señal de entrada.
     vMax=MAYOR(fabs(atof(*(argv+2))),fabs(atof(*(argv+3))));
```

```
     //Nos colocamos en la posción correcta dentro del
     //archivo.
     fseek(archEntrada,
                    desplazamiento*sizeof(double),SEEK_SET);
     //Los datos comienzan a procesarse por tramas.
     for(i=TRAMA;i<=tamBloque;i+=TRAMA){
          //Se lee una trama.
          fread(secuencia,sizeof(double),TRAMA,archEntrada);
          //Se normaliza.
          normaliza(secuencia,TRAMA,vMax);
          //Enviamos resultados a la fase B.
          MPI_Send(secuencia,TRAMA,MPI_DOUBLE,
                                    rango+1,0,MPI_COMM_WORLD);
     }
     //Procesamos los datos sobrantes.
     if(residuo=tamBloque%TRAMA){
          //Leemos los datos restantes.
          fread(secuencia,sizeof(double),
                                         residuo,archEntrada);
          //Los normalizamos.
          normaliza(secuencia,residuo,vMax);
          //Enviamos los resultados a la fase B.
          MPI_Send(secuencia,residuo,MPI_DOUBLE,
                                    rango+1,0,MPI_COMM_WORLD);
     }
     //Enviamos a los procesos de la fase B una señal de
     //paro.
     MPI_Send(NULL,0,MPI_DOUBLE,rango+1,0,MPI_COMM_WORLD);
     //Se cierra el archivo de datos.
     cierra(archEntrada,*(argv+1));
     //Liberamos memoria.
     free(secuencia);
}

/* Esta función es ejecutada por los procesos pertenecientes
 * a la fase B. Éstos se encargan de ir recibiendo los datos
 * que les envían los procesos de la fase A, les aplican la
 * función característica de la ley mu y los datos procesados
 * son trasmitidos a la fase C.
 */
void faseB(char **argv,int rango){
     int numElem,mu=atof(*(argv+5));
     double *secuencia=(double *)
                         reservaMemoria(TRAMA*sizeof(double)),
            vMax=MAYOR(fabs(atof(*(argv+2))),
                                         fabs(atof(*(argv+3))));
     MPI_Status status;
```

```
        while(1){
            //Se reciben los datos provenientes de la fase A.
            MPI_Recv(secuencia,TRAMA,MPI_DOUBLE,
                                rango-1,0,MPI_COMM_WORLD,&status);
            //Se obtiene el número de elementos recibidos.
            MPI_Get_count(&status,MPI_DOUBLE,&numElem);
            //Si no se recibieron datos, el proceso termina su
            //trabajo.
            if(numElem){
                //Aplicamos la ley mu a los datos recibidos.
                leyMu(secuencia,numElem,vMax,mu);
                //Enviamos los datos procesados a la fase C.
                MPI_Send(secuencia,numElem,MPI_DOUBLE,
                                        rango+1,0,MPI_COMM_WORLD);
            }else{
                //Enviamos a los procesos de la fase C una
                //señal de paro.
                MPI_Send(NULL,0,MPI_DOUBLE,
                                        rango+1,0,MPI_COMM_WORLD);
                //Liberamos memoria.
                free(secuencia);
                return;
            }
        }
}

/* Esta función es ejecutada por los procesos pertenecientes
 * a la fase C. Éstos se encargan de ir recibiendo los datos
 * que les envían los procesos de la fase B, obtienen su
 * valor cuantizado correspondiente y los resultados los
 * trasmiten a la fase D. El proceso de menor rango en esta
 * fase crea un archivo con los niveles de cuantización y sus
 * valores codificados asociados.
 */
void faseC(char **argv,int rango){
        int i,numElem,M=atoi(*(argv+4)),
            *valorCuantizado=(int *)
                                reservaMemoria(TRAMA*sizeof(int));
        double *secuencia=(double *)
                                reservaMemoria(TRAMA*sizeof(double)),
               *nivel=(double *)
                                reservaMemoria(M*sizeof(double)),
               min=atof(*(argv+2)),
               max=atof(*(argv+3)),
               a=(max-min)/M;
        MPI_Status status;
        //Los procesos calculan los niveles de cuantización.
```

```
    for(i=1,nivel[0]=(max+min-(M-1)*a)/2.;i<M;i++)
        nivel[i]=nivel[i-1]+a;
    while(1){
        //Se reciben los datos provenientes de la fase B.
        MPI_Recv(secuencia,TRAMA,MPI_DOUBLE,
                        rango-1,0,MPI_COMM_WORLD,&status);
        //Se obtiene el número de elementos recibidos.
        MPI_Get_count(&status,MPI_DOUBLE,&numElem);
        if(numElem){
            //Los datos recibidos son cuantizados.
            for(i=0;i<numElem;i++)
                valorCuantizado[i]=
                        cuantiza(secuencia[i],nivel,M);
            //Enviamos resultados a la fase C.
            MPI_Send(valorCuantizado,numElem,MPI_INT,
                                rango+1,0,MPI_COMM_WORLD);
        }else{
            //Enviamos a los procesos de la fase C una
            //señal de paro.
            MPI_Send(NULL,0,MPI_INT,rango+1,
                                        0,MPI_COMM_WORLD);
            //El proceso de menor rango crea el archivo
            //con los niveles de cuantización y el valor
            //asociado a cada nivel en sistema gray.
            if(!(rango/NUM_FASES)){
                FILE *archNiveles=abre(*(argv+7),"w");
                int gray;
                for(i=0;i<M;i++){
                    gray=sisGray(i);
                    fwrite(&gray,sizeof(int),
                                        1,archNiveles);
                    fwrite(&nivel[i],sizeof(double),
                                        1,archNiveles);
                }
                cierra(archNiveles,*(argv+7));
            }
            //Liberamos memoria.
            free(nivel);
            free(secuencia);
            free(valorCuantizado);
            return;
        }
    }
}

/* Esta función es ejecutada por los procesos pertenecientes
 * a la fase D. Éstos se encargan de ir recibiendo los datos
```

```
 * que les envían los procesos de la fase C, los convierten a
 * sistema gray y almacenan los resultados en un archivo de
 * salida.
 */
void faseD(char *nomArch,int rango){
     int i,numElem,desplazamiento,
         *secuencia=(int *)reservaMemoria(TRAMA*sizeof(int));
     FILE *archSalida;
     MPI_Status status;
     //El proceso de menor rango crea el archivo de salida
     //-si es que éste no exite- y los otros lo abren para
     //leer y escribir.
     if(rango/NUM_FASES){
          MPI_Barrier(MPI_COMM_WORLD);
          archSalida=abre(nomArch,"r+");
     }else{
          archSalida=abre(nomArch,"w");
          MPI_Barrier(MPI_COMM_WORLD);
     }
     //Cada proceso recibe la indicación de dónde comenzar a
     //ubicar los datos de salida que vaya generando.
     MPI_Recv(&desplazamiento,1,MPI_INT,
                  rango-NUM_FASES+1,1,MPI_COMM_WORLD,&status);
     //Nos colocamos en la posción correcta dentro del
     //archivo.
     fseek(archSalida,desplazamiento*sizeof(int),SEEK_SET);
     while(1){
          //Se reciben los datos provenientes de la fase C.
          MPI_Recv(secuencia,TRAMA,MPI_INT,
                           rango-1,0,MPI_COMM_WORLD,&status);
          //Se obtiene el número de elementos recibidos.
          MPI_Get_count(&status,MPI_INT,&numElem);
          //Si no se recibieron datos, el proceso termina su
          //trabajo.
          if(numElem){
               //Los datos recibidos son convertidos a
               //sistema gray.
               for(i=0;i<numElem;i++)
                    secuencia[i]=sisGray(secuencia[i]);
               //Almacenamos resultados.
               fwrite(secuencia,sizeof(int),
                                          numElem,archSalida);
          }else{
               //Liberamos memoria.
               free(secuencia);
               //Cerramos el flujo de datos de salida.
               cierra(archSalida,nomArch);
```

```
                return;
            }
        }
}

/* Esta función, normaliza una secuencia de 'tam' datos
 * utilizando el valor 'vMax'.
 */
void normaliza(double *secuencia,int tam,double vMax){
        int i;
        for(i=0;i<tam;i++)
            secuencia[i]/=vMax;
}

/* Esta función, calcula la ley mu a secuencia de 'tam'
 * datos, utilizando el valor de mu indicado y escalando la
 * secuencia de salida con un factor 'vMax'.
 */
void leyMu(double* secuencia,int tam,double vMax,double mu){
        int i;
        for(i=0;i<tam;i++)
            secuencia[i]=
                    (secuencia[i]<0.?-1.:1.)*vMax*
                    log10(1+mu*fabs(secuencia[i]))/log10(1+mu);
}

/* 'cuantiza', obtiene el índice asociado al valor cuantizado
 * que le corresponde a 'val', de acuerdo a los 'M' niveles
 * de cuantización contenidos en el arreglo 'nivel'. Dicho de
 * otra forma, 'cuantiza', obtiene el índice asociado al
 * elemento del arreglo 'nivel' que está más próximo de 'val'
 * que los demás.
 */
int cuantiza(double val,double *nivel,int M){
        int li=0,ls=M-1,mitad;
        if(val<=nivel[li])
            return li;
        if(val>=nivel[ls])
            return ls;
        while(li+1<ls){
            mitad=li+ls>>1;
            if(val>=nivel[mitad])
                li=mitad;
            else
                ls=mitad;
        }
```

```
    return abs(val-nivel[li])<abs(val-nivel[ls])?li:ls;
}

/* sisGray, expresa el entero 'bin' en sistema gray.
 */
int sisGray(int bin){
    return bin^bin>>1;
}

/* Esta función abre un archivo y verifica que no existan
 * errores al abrirlo.
 */
FILE *abre(char *nombre,char *acceso){
    FILE *archivo;
    if((archivo=fopen(nombre,acceso))==NULL){
        printf("Ocurrió un error
                        al abrir el archivo: %s\n",nombre);
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    return archivo;
}

/* Esta función cierra un archivo y verifica que no existan
 * errores al cerrarlo.
 */
void cierra(FILE *archivo,char *nombre){
    if(fclose(archivo)==EOF){
        printf("Ocurrió un error al
                            cerrar el archivo: %s\n",nombre);
        MPI_Abort(MPI_COMM_WORLD,1);
    }
}

/* Esta función reserva 'bytes' bytes de memoria y verifica
 * si la asignación es exitosa.
 */
void  *reservaMemoria(size_t bytes){
    void *memoria;
    if((memoria=(void *)malloc(bytes))==NULL){
        printf("Insuficiente espacio en memoria\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    return memoria;
}
```

5.3.4 Comentarios adicionales.

La función principal (main) es ejecutada por todos los procesos del grupo, éstos obtienen su rango y el número total de procesos participantes en el cálculo, verifican que el número de argumentos y procesos creados sean los correctos y determinan qué función deben ejecutar, es decir, a qué fase pertenecerán.

Debido a que los procesos de la fase A son los que tienen contacto con los datos de origen, son los encargados de repartirlos entre ellos. Tomando en cuenta la asignación que decidimos usar en la tabla 5.5, los procesos de esta fase realizan la repartición como sigue.

Utilizamos la expresión **`nuevoRango=rango/NUM_FASES`** que nos brinda un nuevo rango que está en el intervalo **`[0, numProcFase-1]`**. Con base en lo anterior, el archivo de entrada es dividido entre los **`numProcFase`** procesos de la fase A como se muestra en la figura 5.16.

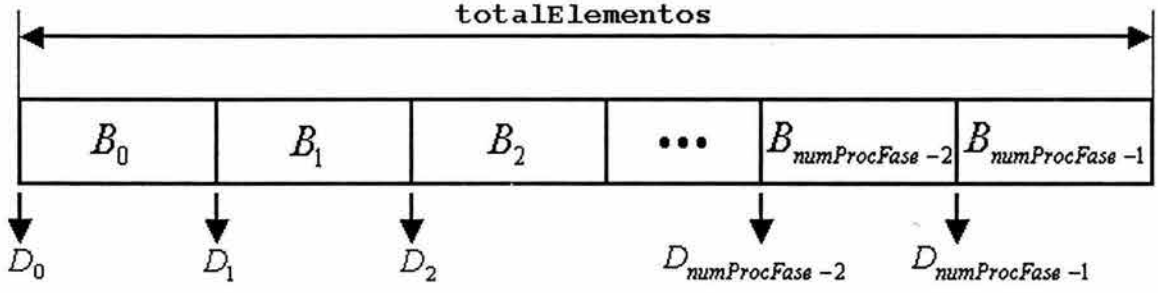


Figura 5.16. Distribución del archivo de entrada entre los procesos de la fase A.

Con la disposición anterior, el proceso de la fase A con **`nuevoRango`** igual a **i** trabaja con el bloque de datos B_i que se encuentra a D_i elementos del inicio del archivo que contiene un número de elementos igual a **`totalElementos`** (véase figura 5.16).

El bloque de datos B_i comienza después de D_i=**`desplazamiento`**$_i$ elementos y posee **`tamBloque`**$_i$ elementos. Las expresiones que permiten calcular el tamaño del bloque y su posición para el proceso de la fase A con **`nuevoRango`** igual a **i** se muestran a continuación.

$$tamBloque_i = \begin{cases} parte\_entera\left(\dfrac{totalElementos}{numProcFase}\right)+1 & si \quad i < residuo \\ parte\_entera\left(\dfrac{totalElementos}{numProcFase}\right) & si \quad residuo \le i < numProcFase \end{cases}$$

$$desplazamiento_i = \begin{cases} i * tamBloque_i & si \quad i < residuo \\ residuo * (tamBloque_i + 1) + (i - residuo) * tamBloque_i & si \quad residuo \leq i < numProcFase \end{cases}$$

Otro punto importante es que, debido a la forma en que se lleva a cabo la asignación de fases, la comunicación entre éstas se facilita pues un proceso con rango igual a **i**, se puede comunicar con la siguiente fase si envía datos al proceso con rango igual a **i+1** y con la fase anterior estableciendo contacto con el proceso con rango igual **i-1**. De manera general, un proceso con rango igual a **i** puede comunicarse con la fase X que está **r** fases delante de la suya a través del proceso con rango **i+r**, del mismo modo podrá comunicarse con la fase Y que está **s** fases detrás de la suya por medio del proceso con rango **i-s**. De esta forma, si el número de procesos del grupo es mayor al número de fases, no sólo se tendrá un cauce de datos (*data pipeline*), sino que tendremos varios flujos de datos en paralelo, que a su vez, trabajan individualmente en paralelo.

Concluimos esta serie de observaciones diciendo que los datos del archivo de salida son colocados en el mismo orden en que están sus valores asociados en el archivo de entrada, con la diferencia de que los valores de salida son enteros, en lugar de reales en doble precisión. Es por esa razón que, los procesos de la primera fase envían el valor de su desplazamiento a los procesos de la última fase con la intención de facilitarles los cálculos e indicarles la posición en la que deben comenzar a ubicar los datos de salida que vayan generando.

5.3.5 Ejemplos de aplicación.

Para observar, de forma gráfica, cómo es que esta aplicación paralela –programa cuant-cod.c– actúa sobre una determinada señal de entrada, mostraremos algunos ejemplos.

Primero, veremos cómo se ven modificadas las muestras de una función después de pasar por un proceso de cuantización y codificación. La función que en este caso utilizamos es la llamada *sinc*, definida como sigue:

$$sinc(x) = \frac{sen(\pi x)}{\pi x}.$$

En la figura 5.17, podemos observar la forma en que se comporta la señal de salida al variar el valor del parámetro μ. Los resultados fueron obtenidos usando muestras de la función *sinc* en el rango [-5,5] con espaciamientos constantes de 0.1 y empleando 256 niveles de cuantización. Se puede observar claramente que las amplitudes de la señal de salida son mayores entre más grande es el valor de μ, asimismo, las amplitudes más pequeñas se ven incrementadas en mayor proporción que las grandes. A pesar de esto, la señal de salida conserva la misma forma que posee la entrada.

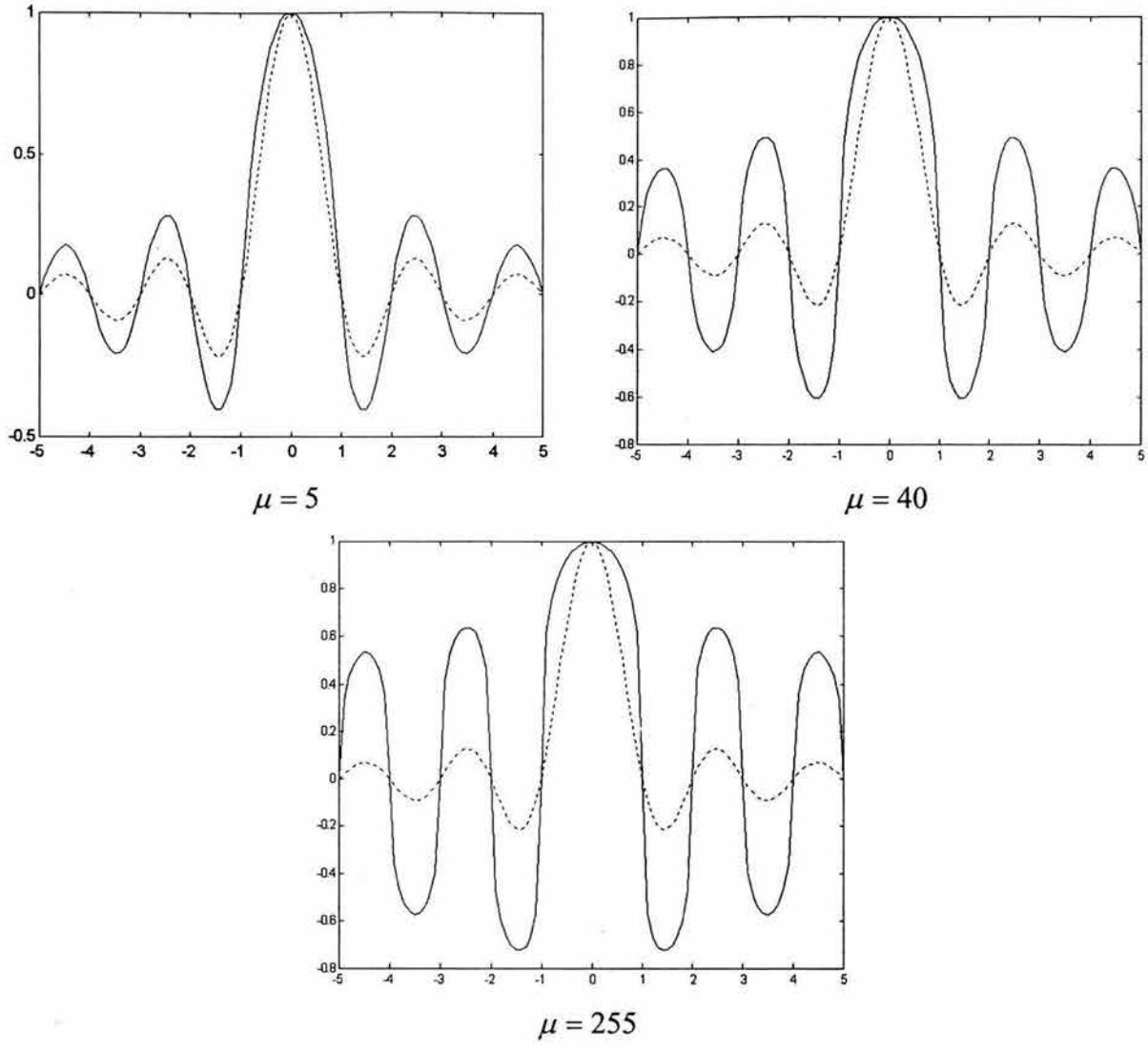


Figura 5.17. Comportamiento de la señal de salida para distintos valores de μ. En cada uno de los casos, la señal en línea continua corresponde a la salida y la señal en línea punteada corresponde a la entrada.

Por otra parte, en la figura 5.18, podemos ver los resultados que se obtienen al utilizar distinto número de niveles de cuantización. Se utilizaron las muestras de entrada que ya fueron detalladas en el anterior ejemplo y un valor constante $\mu = 5$. Podemos observar que la señal de salida se ve más deformada entre menor es el número de niveles de cuantización utilizados, esto se debe principalmente a que se pierde información debido al proceso de cuantización. Entre mayor es el número de niveles empleados en el proceso de cuantización, menor es la pérdida de información, pero se requiere mayor número de bits para representar a cada nivel.

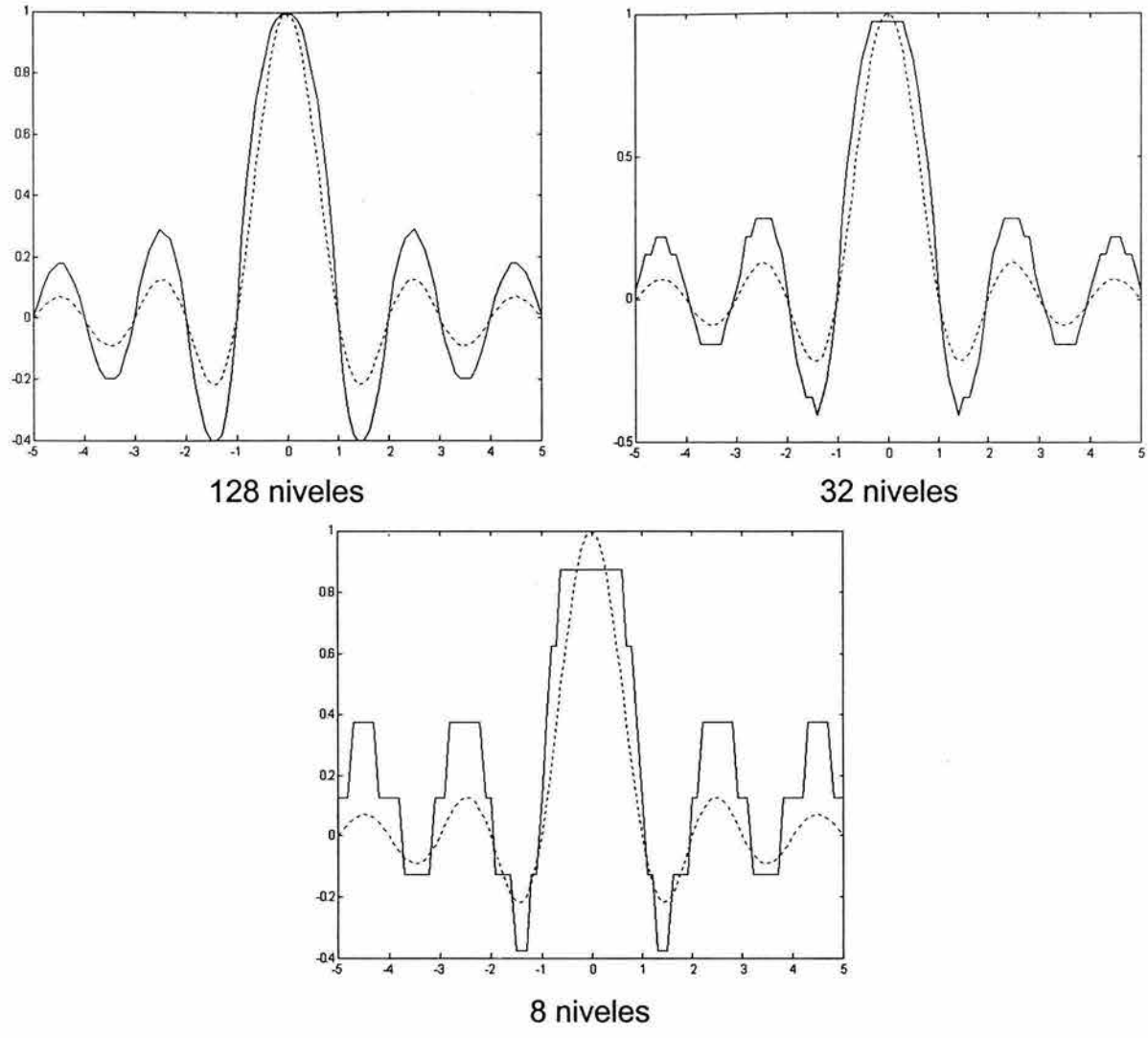
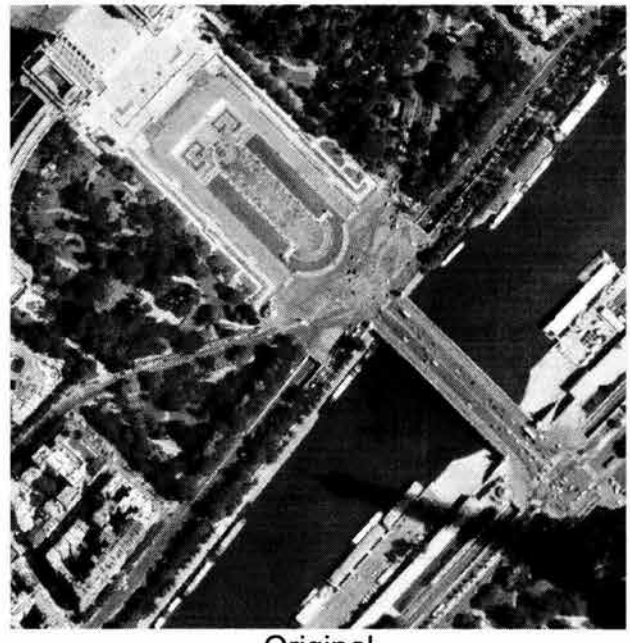


Figura 5.18. Comportamiento de la señal de salida utilizando distinto número de niveles de cuantización. En cada una de las gráficas, la señal en línea continua corresponde a la salida y la señal en línea punteada corresponde a la entrada.

Las muestras de la señal de entrada pueden pertenecer a diversos tipos, por ejemplo, pueden corresponder a una señal de audio, a una señal de video, a una imagen, a los valores de que adquiere una determinada variable involucrada en un proceso físico, etc. Enseguida mostramos los resultados obtenidos al proporcionar una imagen como entrada de esta aplicación paralela.

En la figura 5.19, podemos ver cómo se modifica la imagen de entrada al variar el valor de μ. El número de niveles de cuantización empleados fue 128. Corroboramos que las amplitudes de la señal de salida son mayores entre más grande es el valor de μ, sólo que en este caso esto se manifiesta presentando una imagen de salida más clara, pues las intensidades tienden al color blanco que

es la amplitud más alta. También podemos confirmar que, esencialmente, la salida conserva la misma forma que posee la entrada.



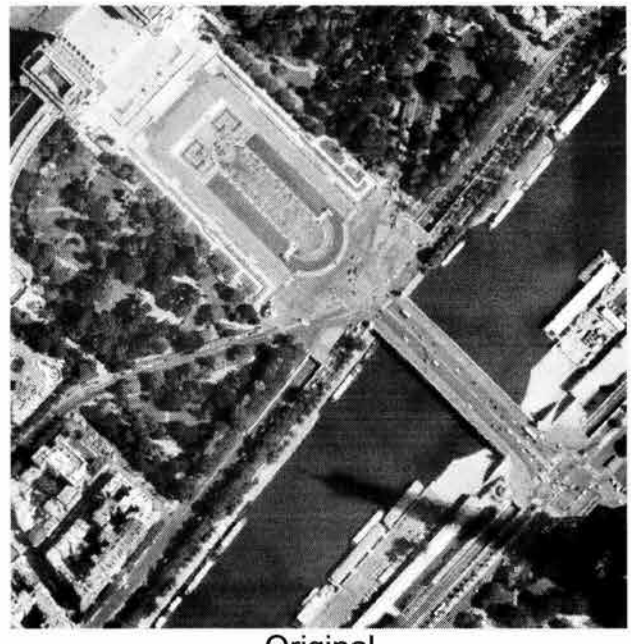
Original

$\mu = 10$ $\mu = 500$

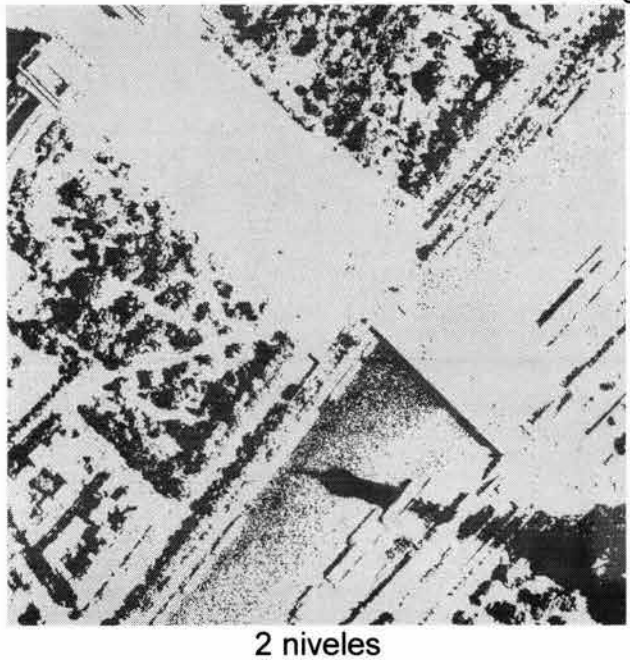
Figura 5.19. Comportamiento de la imagen de salida para distintos valores de μ. Arriba, imagen original: vista aérea de Paris, Francia. Abajo, imágenes de salida para dos diferentes valores de μ.

Para finalizar esta serie de ejemplos, en la figura 5.20, podemos ver los resultados que se obtienen al utilizar distinto número de niveles de cuantización

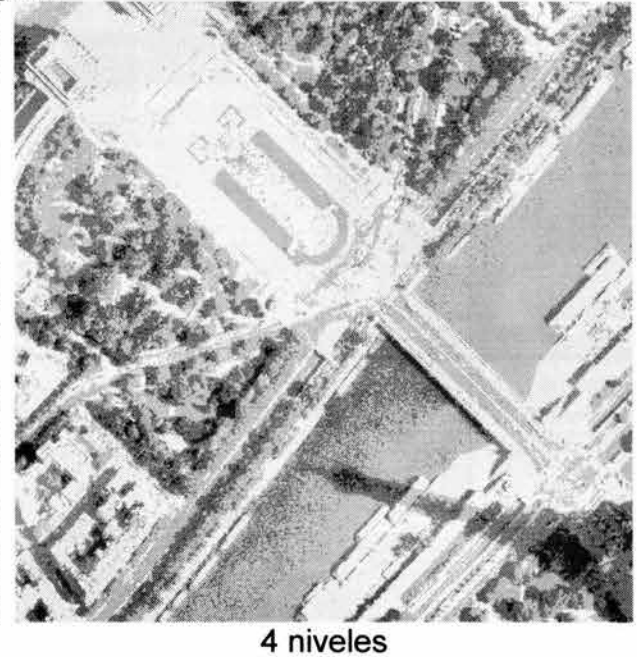
para el caso de una imagen. Utilizamos un valor constante $\mu = 10$. Observamos de nuevo que la salida se ve más degradada entre menor es el número de niveles de cuantización utilizados.



Original



2 niveles



4 niveles

Figura 5.20. Comportamiento de la imagen de salida utilizando distinto número de niveles de cuantización. Arriba, imagen original: vista aérea de Paris, Francia. Abajo, imágenes de salida para dos diferentes números de niveles de cuantización.

5.3.6 Evaluación del desempeño.

Ahora, analizaremos cómo es el desempeño de esta aplicación al ser ejecutada en el cluster. De manera general, el desempeño de una aplicación paralela dependerá en gran medida de la capacidad computacional del sistema donde se ejecute, entre mayor sea ésta, mejor será el desempeño de la aplicación.

En este análisis tomaremos en cuenta dos factores que influyen en el comportamiento de cualquier aplicación paralela: la carga de trabajo y el número procesos que participan en el cálculo. Para ello, mediremos los tiempos de ejecución de la aplicación al ir aumentando el número de procesos para tres distintas cargas de trabajo: una pequeña, una mediana y una grande.

Las cargas de trabajo con las que probamos el programa fueron de 500, 100000 y 5000000 datos de entrada. En cuanto a la variación en el número de procesos, hicimos pruebas con 4, 8, 12 y 16 procesos. El menor número de procesos que utilizamos fue 4, pues es el mínimo número que se puede utilizar en la técnica de Entubamiento de Datos para 4 fases. El máximo, 16 procesos, fue establecido de tal forma que cumpliera con las restricciones propias de la técnica de Entubamiento de Datos y pensando en no exceder las capacidades del sistema.

Para las mediciones de tiempos utilizamos la función que para tal efecto provee la MPI, **MPI_Wtime**. Es importante destacar que, las secciones de código que permiten medir el tiempo de ejecución de un programa deben ser colocadas entre el código del algoritmo de modo que no afecten su comportamiento de alguna forma y verificando que en realidad se esté midiendo bien el tiempo total de ejecución, en otras palabras, lo idóneo es que un solo proceso realice la medición, éste deberá ser el último en finalizar. Es conveniente realizar un análisis del programa para determinar qué proceso es el que finaliza al último, si se tienen problemas para determinarlo con claridad, se pueden utilizar otras funciones como **MPI_Barrier**, para forzarlo.

Para cada condición distinta –de carga de trabajo y de número de procesos– se realizaron 5 corridas del programa y se obtuvo el tiempo promedio de ejecución con el objeto de obtener un valor más representativo.

Debido a que el servidor del cluster es mucho más potente que cualquiera de sus nodos y, por lo tanto, las contribuciones de los nodos sólo pueden ser bien apreciadas si no participa en el cálculo, en las corridas de esta aplicación ningún proceso fue creado en el servidor, es decir que, se le pasó la opción `nolocal` al comando **mpirun**.

Otro punto importante que se debe tomar en cuenta cuando se trabaja con un cluster heterogéneo es la forma en que se asignarán los procesos en las

distintas máquinas que lo conforman, pues esta asignación repercute directamente en el desempeño de la aplicación paralela. Antes de realizar una distribución de procesos, cuando se emplea la técnica de Entubamiento de Datos es conveniente efectuar un análisis para determinar cuál de las fases por las que atraviesa el cauce de datos es la que requiere de mayor procesamiento para asignarle a los procesos que la conforman los mejores equipos del cluster. En el caso particular de este programa y más concretamente de las pruebas que realizamos –en éstas, decidimos utilizar un número de niveles de cuantización igual al que se usa en la música con calidad de CD, es decir, representando cada nivel con 16 bits–, la fase que precisa mayor procesamiento es la que designamos como fase C, así que ejecutamos los procesos pertenecientes a ésta en las mejores máquinas del cluster. Por eso fue que empleamos la opción `machinefile` del comando **mpirun**. También debemos considerar que, al utilizar la opción `noloca1`, el arranque de procesos remotos en los nodos ya no será realizado por el servidor y por lo tanto debe seleccionarse a uno, de entre los mejores equipos, para que realice esta labor; la selección de un equipo lento para este fin, produce un incremento en el tiempo de ejecución.

5.3.6.1 Resultados.

La tabla 5.6 muestra los tiempos de ejecución obtenidos al correr la aplicación paralela para los distintos tamaños de carga y número de procesos.

	Tiempos de ejecución para:		
Número de procesos	**500 datos [s]**	**100000 datos [s]**	**5000000 datos [s]**
4	1.778473	7.924378	309.763843
8	2.049393	5.536177	192.880183
12	2.168820	5.106911	173.669792
16	2.400194	4.756488	113.607874

Tabla 5.6. Resultados obtenidos con las pruebas realizadas a la aplicación paralela de cuantización y codificación.

En la figura 5.21, mostramos la gráfica de los tiempos de ejecución obtenidos al ir aumentando el número de procesos con una carga pequeña de trabajo. Con pocos datos que procesar, el tiempo de ejecución aumenta si se utilizan más procesos para dar solución al problema. En casos con poca carga de trabajo el problema será resuelto más rápido utilizando menos nodos.

El comportamiento que tiene el programa para una carga de trabajo moderada se muestra en la figura 5.22. Con esta carga de trabajo, se aprecia que, ya comienzan a obtenerse ventajas utilizando más nodos en el procesamiento. También notamos que es conveniente tener más de un proceso por fase.

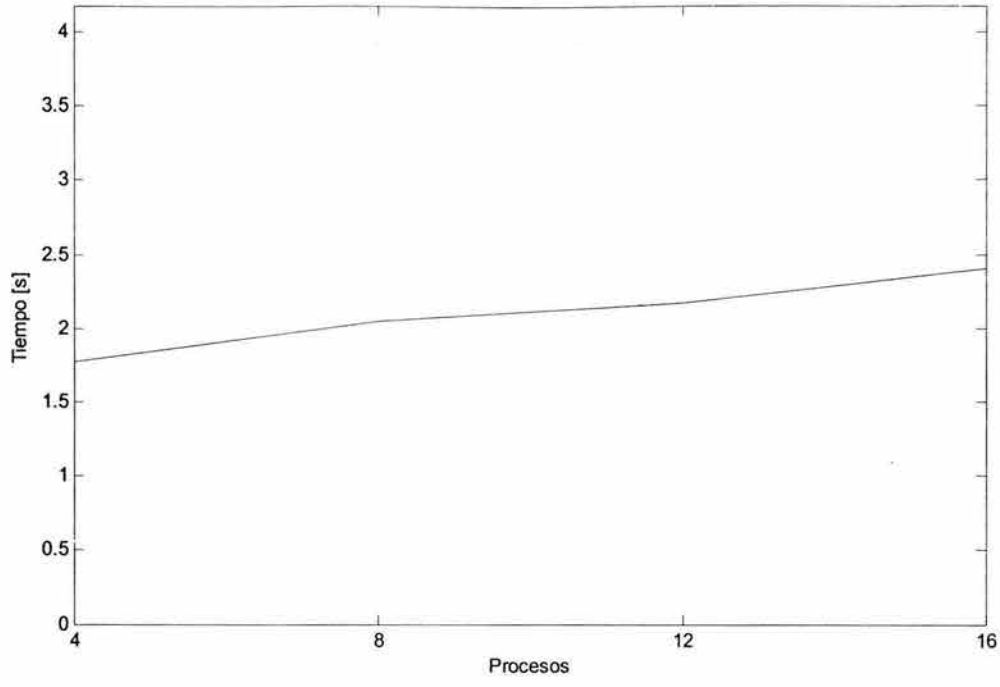


Figura 5.21. Gráfica de tiempo de ejecución contra número de procesos, con una carga constante de trabajo de 500 datos.

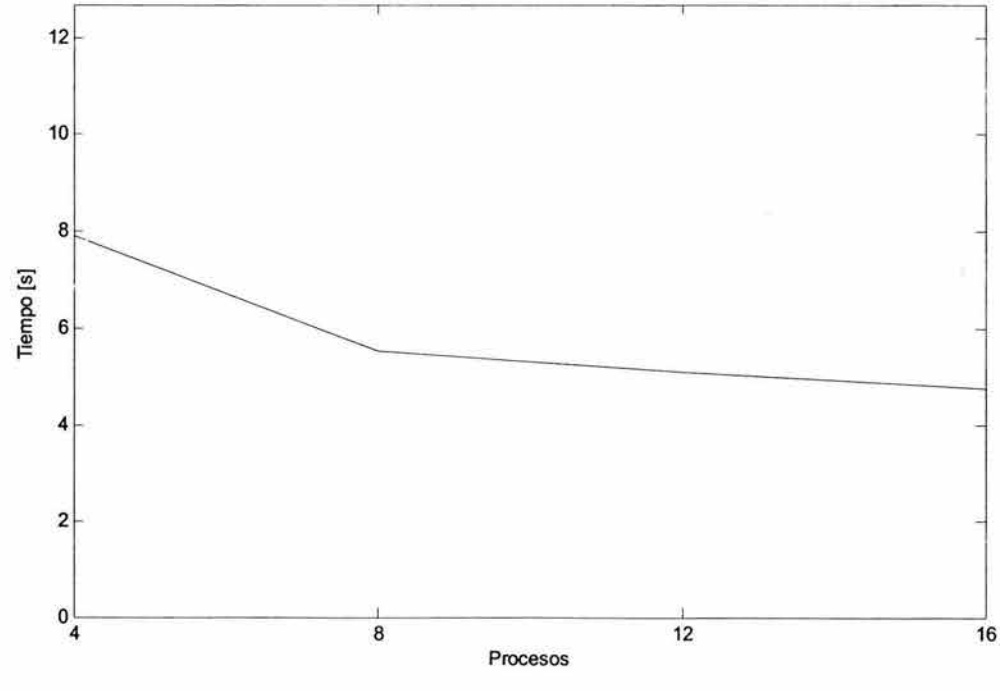


Figura 5.22. Gráfica de tiempo de ejecución contra número de procesos, con una carga constante de trabajo de 100000 datos.

Finalmente, en la figura 5.23, vemos la gráfica del desempeño de la aplicación ante una carga considerada como grande. Podemos visualizar que se va obteniendo una reducción en tiempo de ejecución conforme se involucran más procesos en la solución del problema. Asimismo, debido a que –por haber utilizado la técnica de Entubamiento de Datos–, se requiere que los procesos estén en una constante comunicación y por lo tanto se tiene que invertir en esta parte del tiempo, la verdadera ventaja que posee el algoritmo se logra al tener cauces paralelos de datos, trabajando a su vez, en paralelo.

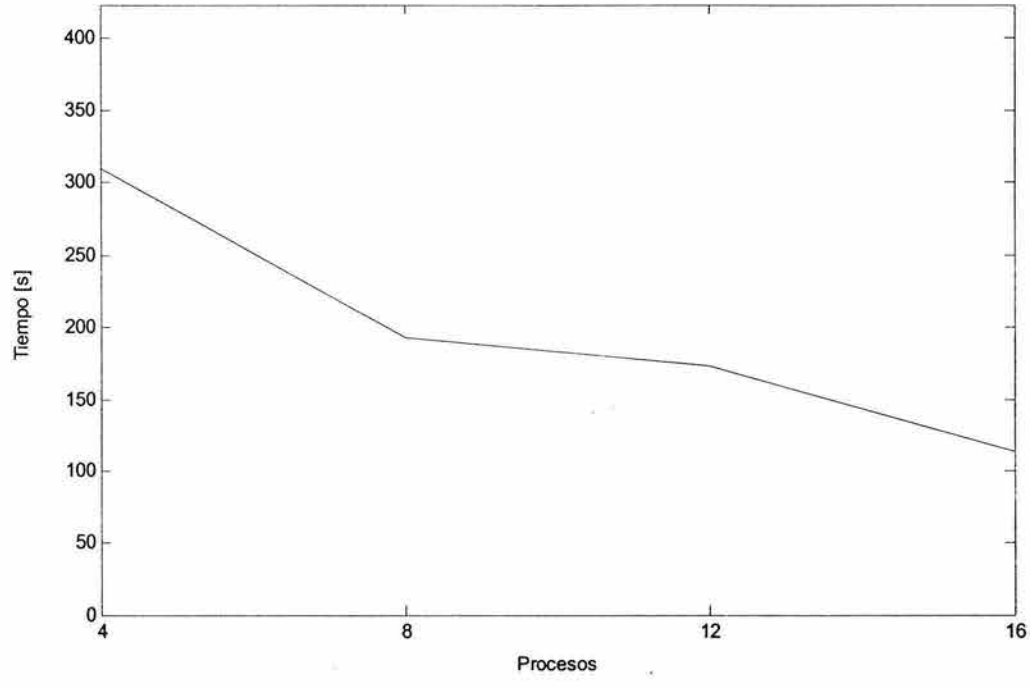


Figura 5.23. Gráfica de tiempo de ejecución contra número de procesos, con una carga constante de trabajo de 5000000 datos.

5.4 Procesamiento de imágenes.

5.4.1 Introducción.

En la actualidad, las imágenes digitales han adquirido un papel preponderante en diferentes áreas que va más allá de su simple visualización en entornos multimedia. Hoy en día, se les emplea en medicina, meteorología, cartografía, geología, biología, astronomía, aplicaciones militares, arqueología, teledetección de recursos naturales a partir de las imágenes que envían los satélites artificiales, criminología, etc. En ocasiones, el trabajo computacional requerido por el Procesamiento Digital de Imágenes (PDI) puede ser muy demandante, esto depende de la complejidad del tratamiento que se dará a las imágenes y de la cantidad de información que debe manipularse cuando se utilizan imágenes de alta resolución.

En esta sección abordaremos dos de las técnicas que pertenecen al mejoramiento de imágenes y presentaremos una solución para su implementación sobre un ambiente distribuido mediante la MPI.

Las dos técnicas antes citadas son: el Procesamiento Puntual y el Filtrado Espacial. Antes de mostrar cómo operan estas técnicas sobre las imágenes es conveniente revisar algunos aspectos básicos del PDI.

5.4.1.1 Aspectos básicos.

Concepto de imagen.

Una imagen puede definirse como una función bidimensional f(x, y); donde x, y son coordenadas espaciales. La amplitud de f para cualquier par (x, y) se conoce como intensidad o nivel de gris de la imagen en ese punto (véase figura 5.24).

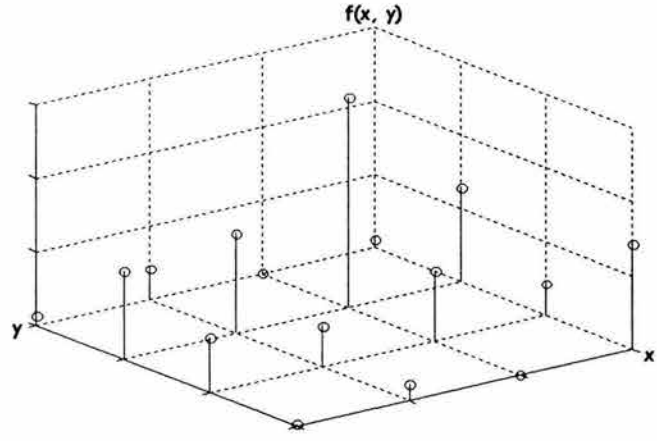


Figura 5.24. Representación de la amplitud de una imagen en función de sus coordenadas espaciales.

En una imagen digital, el par (x, y) y los valores de la amplitud de f, son finitos y discretos.

La amplitud, intensidad o nivel de gris para cualquier punto de una imagen monocromática puede expresarse como:

$$l = f(x,y) \quad donde \quad L_{\min} \le l \le L_{máx}$$

El intervalo $[L_{min}, L_{máx}]$ se conoce como escala de grises. En las distintas aplicaciones, este intervalo comúnmente se traslada a $[0, L-1]$ con L potencia de dos. De este modo, $l = 0$ es considerado negro y $l = L-1$ es considerado blanco

en dicha escala. Todos los valores intermedios son matices del gris que van variando del negro al blanco.

Representación de imágenes digitales monocromáticas.

Una imagen digital monocromática f(x, y) de resolución MxN, puede definirse como:

$$f(x,y)=\begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & & \vdots \\ f(M-1,0) & f(M-1) & \cdots & f(M-1,N-1) \end{bmatrix}. \qquad (5.4\text{-}1)$$

Cada elemento del lado derecho de la expresión (5.4-1) es conocido elemento de la imagen o píxel.

Es necesario mencionar que los desarrollos que en esta tesis se abordarán están diseñados para aplicarse sobre imágenes monocromáticas, por lo tanto, no es posible aplicarlos directamente a imágenes a color. A pesar de ello, la utilización del modelo de color llamado HSI (Hue Saturation Intensity) hace posible la aplicación de las técnicas para imágenes monocromáticas en imágenes de color. Así, cualquier técnica de mejoramiento de imágenes monocromáticas puede ser utilizada como herramienta en el procesamiento de imágenes a color. Simplemente se requiere convertir la imagen a color al formato HSI, procesar la componente de intensidad, y convertir el resultado al formato RGB. La explicación de los anteriores modelos de color, sus correspondencias y conversiones son temas que trascienden a este trabajo de investigación.

5.4.2 Procesamiento puntual.

5.4.2.1 Introducción.

El procesamiento puntual es una de las técnicas de mejoramiento de imágenes, sus métodos operan directamente sobre los píxeles de una imagen y su objetivo es modificar una imagen para que el resultado sea más adecuado para una determinada aplicación. Las funciones utilizadas en el procesamiento puntual tienen la forma general:

$$s = T(r) \qquad (5.4\text{-}2)$$

donde: T es la función de transformación y r, s son variables que denotan el nivel de gris de la imagen en el píxel (x, y) antes y después de la trasformación, respectivamente. Como el mejoramiento de la imagen en cualquier punto depende sólo del nivel de gris en ese punto, este tipo de técnicas se conocen como procesamiento puntual.

Como lo muestra la expresión (5.4-2), existen un sinnúmero de transformaciones puntuales que pueden ser aplicadas a los píxeles de una imagen. Para ejemplificar cómo trabaja el procesamiento puntual llevaremos a cabo la implementación distribuida de una trasformación muy conocida: el negativo de una imagen.

Los negativos de las imágenes digitales son útiles en diferentes aplicaciones, como en el despliegue de imágenes médicas y en la fotografía. El negativo de una imagen digital se obtiene utilizando la función de transformación $s = T(r)$ que se muestra en la figura 5.25.

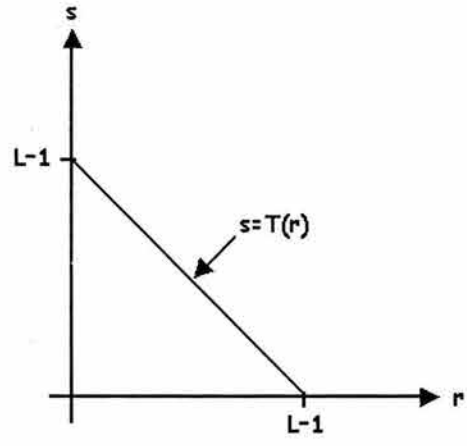


Figura 5.25. Negativo de una imagen: L representa el número de niveles de gris.

La idea es revertir el orden del negro al blanco de tal forma que la intensidad del píxel de la salida decrezca a medida que la intensidad del píxel de entrada crezca y viceversa.

5.4.2.2 Planteamiento del problema.

Con las bases anteriores, comenzaremos la explicación de cómo decidimos realizar el programa para la obtención del negativo de una imagen. En éste emplearemos la técnica de programación paralela Maestro-Esclavo.

Es necesario determinar el papel que desempeñará cada proceso del grupo. La asignación de papeles que decidimos emplear para un grupo de **n** procesos con rangos **0, 1, ..., n-1** se muestra en la tabla 5.7.

Tipo de proceso	Rango o identificador del proceso
Maestro	0
Esclavo	1, ..., n-1

Tabla 5.7. Asignación de papeles seleccionada en la aplicación paralela de procesamiento puntual de imágenes.

En este programa, la fuente de datos serán archivos de imagen en formato crudo (*.raw). Este tipo de archivos está compuesto de una secuencia de bytes que representan el nivel de gris de cada píxel de la imagen. A diferencia de otros formatos como BMP, TIFF, GIF, JPEG, PSD, PNG, entre otros muchos, que almacenan información acerca de la imagen y algunos de ellos realizan una compresión de la misma, el formato crudo sólo guarda los valores de los niveles de gris de los píxeles de la imagen.

En las siguientes líneas, describimos de forma general la forma en que los procesos se coordinan para obtener el negativo de una imagen.

Proceso Maestro.

Lo primero que debe realizar el Proceso Maestro es obtener el número total de píxeles que forman a la imagen para determinar con cuántos de ellos trabajará cada Proceso Esclavo, una vez que lo ha determinado, envía un mensaje a los Procesos Esclavo que corresponde al número de píxeles con los que trabajará cada uno. Depués, el Maestro debe enviarle a cada Esclavo los píxeles de la imagen que le corresponden según su rango en el grupo, esperar a que éstos los procesen para posteriormente recibir el resultado y finalmente crear el archivo de salida que contendrá el negativo de la imagen.

El Proceso Maestro obtiene el número de elementos de imagen con los que trabajará cada Proceso Esclavo como sigue: se divide el número total de elementos entre el total de Procesos Esclavo, esto proporciona el número de píxeles con los que debe trabajar cada Esclavo, si existen elementos sobrantes debido a que existió un residuo en la división anterior, éstos se reparten uno por uno entre los distintos Esclavos comenzando la repartición por el Proceso Esclavo que tiene el menor rango.

Procesos Esclavo.

Una vez que cada Esclavo cuenta con la parte de la imagne con la que le tocó trabajar y el número de píxeles que la constituyen, se dedica a obtener su negativo. Una vez procesada, la parte de la imagen, es enviada al Proceso Maestro para que una los resultados parciales y genere el archivo de salida.

El código fuente del programa que obtiene el negativo de una imagen es el de las siguientes líneas:

5.4.2.3 Código de la aplicación paralela.

```
/*************************************************************/
/*            TÉCNICAS DE PROGRAMACIÓN PARALELA              */
/*                                                           */
/*PROGRAMA:    negativo.c                                    */
/*                                                           */
/*AUTORES:     Christian González,                           */
/*             Lissette González y                           */
/*             Eryk Ramírez.                                 */
/*                                                           */
/*DESCRIPCIÓN: Obtiene el negativo de una imagen con         */
/*             formato crudo (formato RAW) mediante la       */
/*             técnica de programación paralela              */
/*             Maestro-Esclavo.                              */
/*                                                           */
/*************************************************************/

/* Bibliotecas */
//mpi.h es necesaria para utilizar las funciones de la MPI.
#include <mpi.h>
#include <stdio.h>

/* Definiciones para el preprocesador */
//Definimos el número de niveles de gris.
#define L 256

/* Prototipos de función */
void maestro(int,char**);
void esclavo(int);
void *reservaMemoria(size_t);
void cierra(FILE*,char*);
FILE *abre(char*,char*);
unsigned char negativo(unsigned char);

/* Función principal */
int main(int argc,char *argv[]){
     int rango;
     //Iniciamos un cómputo con MPI.
     MPI_Init(&argc,&argv);
          //Cada proceso obtiene su rango dentro del grupo.
          MPI_Comm_rank(MPI_COMM_WORLD,&rango);
          //Se le asigna una función a cada proceso
          //dependiendo del valor de su rango.
          if (rango)
               esclavo(rango);
          else
```

```
            maestro(argc,argv);
    //Finalizamos el cómputo de la MPI.
    MPI_Finalize();
    return 0;
}

/*****************************************************/
/*  Maestro                                          */
/*****************************************************/

/* El Proceso Maestro ejecuta esta función y se encarga de
 * dividir el trabajo entre los distintos Esclavos, une las
 * soluciones parciales y obteniene el resultado final.
 */
void maestro(int argc,char **argv){
    unsigned char *imagen;
    int i,tamBloque,*tamBloques,
        desplazamiento, *desplazamientos,
        residuo,tamanioGrupo,totalPixeles;
    FILE *archivoImagen,*archivoImagenNegativo;
    //Se comprueba que el número de argumentos sea el
    //correcto, de lo contrario se finaliza el programa.
    if(argc!=3){
        printf("Formato de ejecución:\n
                    negativo <archivo_de_imagen>
                                <archivo_negativo_imagen>\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    //Abrimos el archivo de la imagen.
    archivoImagen=abre(*(argv+1),"r");
    //Calculamos el total de píxeles que forman a la imagen.
    fseek(archivoImagen,0L,SEEK_END);
    totalPixeles=ftell(archivoImagen);
    //Colocamos a la imagen en memoria.
    imagen=(unsigned char *)reservaMemoria(totalPixeles);
    rewind(archivoImagen);
    fread(imagen,1,totalPixeles,archivoImagen);
    cierra(archivoImagen,*(argv+1));
    //Se determina qué parte de la imagen se enviará a cada
    //Proceso Esclavo.
    MPI_Comm_size(MPI_COMM_WORLD,&tamanioGrupo);
    tamBloque=totalPixeles/(tamanioGrupo-1);
    residuo=totalPixeles%(tamanioGrupo-1);
    tamBloques=(int *)reservaMemoria(
                                    sizeof(int)*tamanioGrupo);
    desplazamientos=(int *)reservaMemoria(
                                    sizeof(int)*tamanioGrupo);
```

```
    for (i=1,tamBloques[0]=
                    desplazamientos[0]=0;i<tamanioGrupo;i++){
        tamBloques[i]=(i>residuo)?tamBloque:tamBloque+1;
        desplazamientos[i]=
                            desplazamientos[i-1]+
                                        tamBloques[i-1];
    }
    //Le enviamos un mensaje a cada Esclavo para indicarle
    //el número de píxeles que tiene la parte de la imagen
    //con la que trabajará.
    MPI_Scatter(tamBloques,1,MPI_INT,
                      &tamBloque,1,MPI_INT,0,MPI_COMM_WORLD);
    //Enviamos la parte de la imagen que le corresponde a
    //cada Esclavo.
    MPI_Scatterv(imagen,tamBloques,
                 desplazamientos,MPI_UNSIGNED_CHAR,
                 NULL,0,MPI_UNSIGNED_CHAR,0,MPI_COMM_WORLD);
    //Recolectamos los resultados obtenidos por cada
    //Proceso Esclavo para formar el negativo total de la
    //imagen.
    MPI_Gatherv(NULL,0,MPI_UNSIGNED_CHAR,
                imagen,tamBloques,desplazamientos,
                        MPI_UNSIGNED_CHAR,0,MPI_COMM_WORLD);
    free(tamBloques);
    free(desplazamientos);
    //Creación del archivo de salida.
    archivoImagenNegativo=abre(*(argv+2),"w");
    fwrite(imagen,1,totalPixeles,archivoImagenNegativo);
    cierra(archivoImagenNegativo,*(argv+2));
    free(imagen);
}

/*************************************************************/
/*  Esclavo                                                  */
/*************************************************************/

/* Esta función será ejecutada por todos los Procesos Esclavo
 * permite que éstos obtengan la parte de la imagen que les
 * corresponde, la procesen y la envíen ya procesada al
 * Maestro.
 */
void esclavo(int rango){
    unsigned char *parteDeImagen;
    int tamBloque;
    register i;
    //Cada Esclavo obtiene el número de píxeles que le
    //corresponde procesar.
```

```
    MPI_Scatter(NULL,0,MPI_INT,&tamBloque,
                                    1,MPI_INT,0,MPI_COMM_WORLD);
    //Los Esclavos obtiene la parte de la imagen de la que
    //que deben obtener el negativo.
    parteDeImagen=(unsigned char *)
                                     reservaMemoria(tamBloque);
    MPI_Scatterv(NULL,NULL,NULL,MPI_UNSIGNED_CHAR,
                 parteDeImagen,tamBloque,
                 MPI_UNSIGNED_CHAR,0,MPI_COMM_WORLD);
    //Cada Esclavo obtiene el negativo de su porción de
    //imagen.
    for(i=0;i<tamBloque;
             parteDeImagen[i++]=negativo(parteDeImagen[i]));
    //La parte de imagen procesada es enviada de vuelta al
    //Maestro.
    MPI_Gatherv(parteDeImagen,tamBloque,
                MPI_UNSIGNED_CHAR,NULL,NULL,NULL,
                MPI_UNSIGNED_CHAR,0,MPI_COMM_WORLD);
}

/* Esta función le aplica la trasformación puntual negativo
 * a un elemento de imagen.
 */
unsigned char negativo(unsigned char pixel){
    return (-pixel+L-1);
}

/* Esta función abre un archivo y verifica que no existan
 * errores al abrirlo.
 */
FILE *abre(char *nombre,char *acceso){
    FILE *archivo;
    if((archivo=fopen(nombre,acceso))==NULL){
        printf("Ocurrió un error
                          al abrir el archivo: %s\n",nombre);
        MPI_Abort(MPI_COMM_WORLD,1);
    }
    return archivo;
}

/* Esta función cierra un archivo y verifica que no existan
 * errores al cerrarlo.
 */
void cierra(FILE *archivo,char *nombre){
    if(fclose(archivo)==EOF){
        printf("Ocurrió un error al
                            cerrar el archivo: %s\n",nombre);
```

```
            MPI_Abort(MPI_COMM_WORLD,1);
        }
}

/* Esta función reserva 'bytes' bytes de memoria y verifica
 * si la asignación es exitosa.
 */
void  *reservaMemoria(size_t bytes){
        void *memoria;
        if((memoria=(void *)malloc(bytes))==NULL){
            printf("Insuficiente espacio en memoria\n");
            MPI_Abort(MPI_COMM_WORLD,1);
        }
        return memoria;
}
```

5.4.2.4 Comentarios adicionales.

La función principal (main) es ejecutada por todos los procesos del grupo, éstos obtienen su rango y es así como podemos determinar qué función debe ejecutar cada uno.

Debido a que la transformación del negativo es una técnica de procesamiento puntual, podemos trabajar con la imagen como si fuera una secuencia de elementos y no una matriz. En otras palabras, una imagen con resolución de **M** renglones por **N** columnas puede considerarse como un arreglo lineal constituido por **MxN** elementos.

Con base en lo anterior, la imagen es distribuida entre los Procesos Esclavo como se muestra en la figura 5.26.

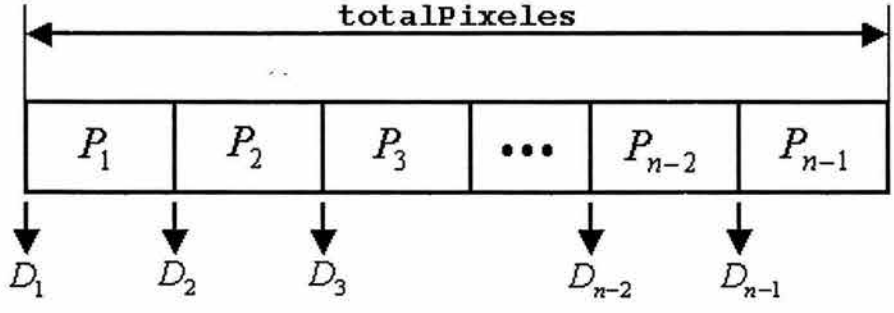


Figura 5.26. División de la imagen de entrada entre los Procesos Esclavo.

Así, el Proceso Esclavo con rango igual a **i** trabaja con la porción de imagen P_i que se encuentra a D_i píxeles del inicio de la imagen que contiene un número de elementos de imagen igual a `totalPixeles` (véase figura 5.26).

La porción de imagen P_i comienza después de D_i=**desplazamientos**$_i$ elementos de imagen y posee **tamBloques**$_i$ píxeles. Las expresiones que permiten calcular los anteriores parámetros para el proceso con rango igual a **i**, **i>0**, de un conjunto de **n** son:

$$tamBloques_i = \begin{cases} parte\_entera\left(\frac{totalPixeles}{\text{n - 1}}\right)+1 & si \quad i < residuo \\ parte\_entera\left(\frac{totalPixeles}{n-1}\right) & si \quad residuo \le i < n \end{cases},$$

$$desplazamientos_i = \begin{cases} 0 & si \quad i = 1 \\ desplazamientos_{i-1} + tamBloques_{i-1} & si \quad 1 < i < n \end{cases}.$$

5.4.2.5 Ejemplos de aplicación.

Enseguida, en la figura 5.27, mostramos algunas imágenes y sus negativos, estos últimos fueron obtenidos mediante el programa ejecutable asociado al fuente negativo.c. Es importante mencionar que por la naturaleza de la trasformación correspondiente al negativo, este programa puede ser utilizado tanto en imágenes monocromáticas como en imágenes a color.

Figura 5.27. Obtención del negativo de una imagen. Arriba, imagen del interior de la catedral de Worcester, Massachusetts, EE.UU. Abajo, el parque nacional de Yosemite, California, EE.UU. A la derecha, las imágenes después de aplicarles la trasformación correspondiente al negativo.

5.4.2.6 Evaluación del desempeño.

En esta sección analizaremos el desempeño que tiene esta aplicación sobre el cluster. De manera general, el desempeño de una aplicación paralela

dependerá en gran medida de la capacidad computacional del sistema donde se ejecute, entre mayor sea ésta, mejor será el desempeño de la aplicación.

Al igual que con las otras aplicaciones, también analizaremos el comportamiento del programa al variar la carga de trabajo y el número procesos que participan en el cálculo. Para ello, mediremos los tiempos de ejecución de la aplicación al ir aumentando el número de procesos para tres distintas cargas de trabajo: una pequeña, una mediana y una grande.

Las cargas de trabajo con las que probamos el programa fueron matrices de entrada compuestas de 400, 1000000 y 2500000 elementos. En cuanto a la variación en el número de procesos, hicimos pruebas con 2, 3, ..., y 14 procesos. El menor número de procesos que utilizamos fue 2, pues es el mínimo número que permite utilizar la técnica Maestro-Esclavo. El máximo, 14 procesos, corresponde al número de nodos que componen el cluster –es posible ejecutar más de un proceso por nodo pero, debido a las limitantes de memoria del cluster y la disposición del algoritmo, ésta no resulta una buena opción–.

Para las mediciones de tiempos utilizamos la función que para tal efecto provee la MPI, **MPI_Wtime**. Es importante destacar que, las secciones de código que permiten medir el tiempo de ejecución de un programa deben ser colocadas entre el código del algoritmo de modo que no afecten su comportamiento de alguna forma y verificando que en realidad se esté midiendo bien el tiempo total de ejecución, en otras palabras, lo idóneo es que un solo proceso realice la medición, éste deberá ser el último en finalizar. Es conveniente realizar un análisis del programa para determinar qué proceso es el que finaliza al último, si se tienen problemas para determinarlo con claridad, se pueden utilizar otras funciones como **MPI_Barrier**, para forzarlo.

Para cada condición distinta –de carga de trabajo y de número de procesos– se realizaron 5 corridas del programa y se obtuvo el tiempo promedio de ejecución con el objeto de obtener un valor más representativo.

Debido a que el servidor del cluster es mucho más potente que cualquiera de sus nodos y, por lo tanto, las contribuciones de los nodos sólo pueden ser bien apreciadas si no participa en el cálculo, en las corridas de esta aplicación ningún proceso fue creado en el servidor, es decir que, se le pasó la opción `nolocal` al comando **mpirun**.

Otro punto importante que se debe tomar en cuenta cuando se trabaja con un cluster heterogéneo es la forma en que se asignarán los procesos en las distintas máquinas que lo conforman, pues esta asignación repercute directamente en el desempeño de la aplicación paralela. Para el caso particular de este programa, el proceso que realiza el papel de Maestro es el que debe ser ejecutado en uno de los mejores equipos ya es el encargado de distribuir y reunir el trabajo, en la medida que lo haga más rápido los resultados tardarán menos en ser obtenidos. Por eso fue que empleamos la opción `machinefile` del comando

mpirun. También debemos considerar que, al utilizar la opción `nolocal`, el arranque de procesos remotos en los nodos ya no será realizado por el servidor y por lo tanto debe seleccionarse a uno, de entre los mejores equipos, para que realice esta labor; la selección de un equipo lento para este fin, produce un incremento en el tiempo de ejecución.

5.4.2.6.1 Resultados.

La tabla 5.8 muestra los tiempos de ejecución obtenidos al correr la aplicación paralela para los distintos tamaños de carga y número de procesos.

	Tiempos de ejecución para una imagen de:		
Número de procesos	**400 píxeles [s]**	**1000000 píxeles [s]**	**2500000 píxeles [s]**
2	0.066218	5.931518	18.198148
3	0.064898	4.071542	18.094540
4	0.092231	3.700142	18.338830
5	0.084440	3.667868	16.888039
6	0.097071	3.727915	14.040310
7	0.098060	3.692672	11.806343
8	0.109392	3.749596	11.162897
9	0.097977	3.704166	12.176946
10	0.092929	3.636598	11.569855
11	0.098790	3.740165	11.747224
12	0.098604	4.091677	12.182251
13	0.127216	4.036547	11.948602
14	0.126411	3.824308	12.457748

Tabla 5.8. Resultados obtenidos con las pruebas realizadas a la aplicación paralela de procesamiento puntual de imágenes.

En la figura 5.28, mostramos la gráfica de los tiempos de ejecución obtenidos al ir aumentando el número de procesos con una carga pequeña de trabajo. De manera general, el tiempo de ejecución no mejora con la utilización de más procesos. La carga de trabajo es muy pequeña como para que se puedan obtener ventajas con la utilización de un mayor número de procesos. Por lo tanto, para poco procesamiento resulta mejor utilizar menos procesos en el cálculo.

El comportamiento que tiene el programa para una carga de trabajo moderada se muestra en la figura 5.29. Con una carga de trabajo como ésta, se nota que se invierte más tiempo utilizando el mínimo número de procesos, para un número mayor que el mínimo, el tiempo de ejecución es casi constante.

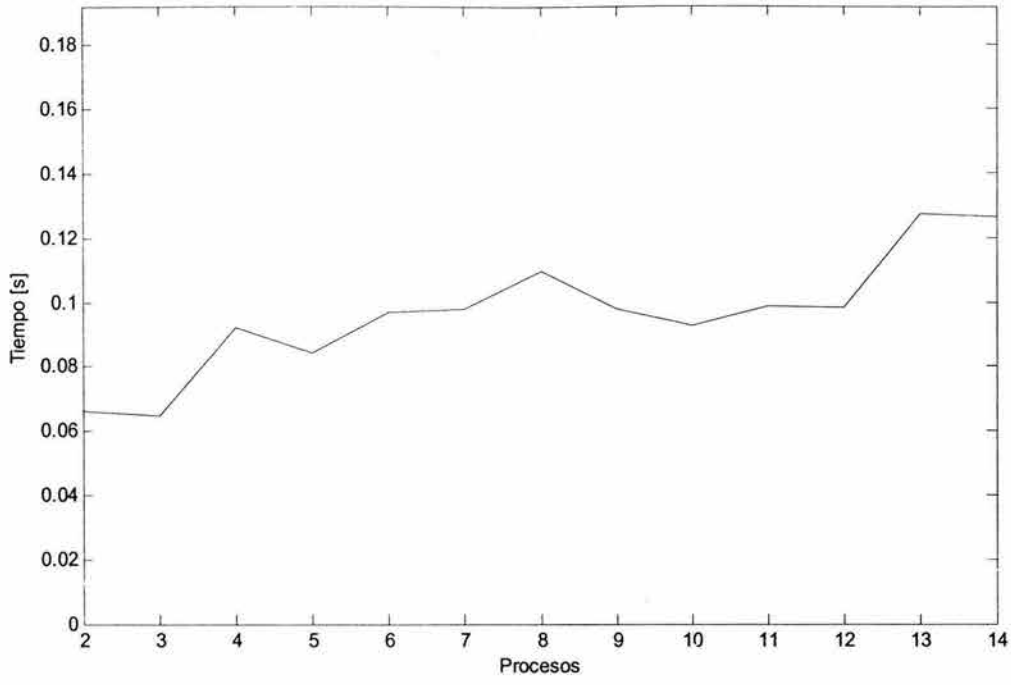


Figura 5.28. Gráfica de tiempo de ejecución contra número de procesos, utilizando una imagen de entrada formada por 400 píxeles.

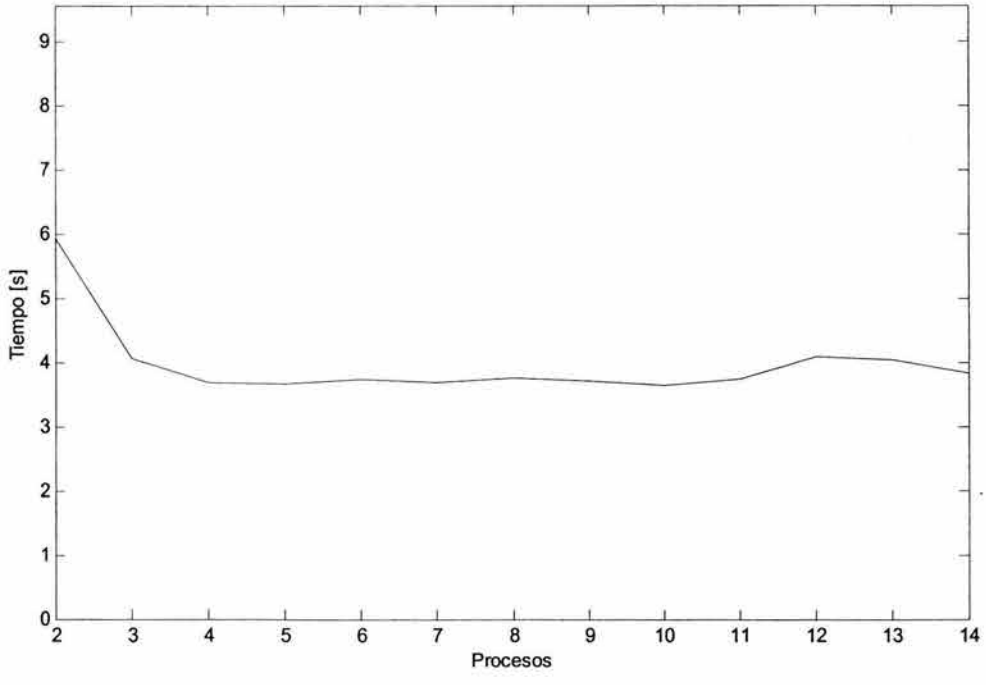


Figura 5.29. Gráfica de tiempo de ejecución contra número de procesos, utilizando una imagen de entrada formada por 1000000 píxeles.

Finalmente, en la figura 5.30, vemos la gráfica del desempeño de la aplicación ante una carga considerada como grande. En esta ocasión, el tiempo de ejecución disminuye al utilizar más procesos pero hasta un límite, pues se observa que de 2 a 8 procesos se consigue reducir el tiempo de ejecución, pero al ejecutar la aplicación con un número de procesos mayor a 8 ya no se consiguen mejoras e incluso el tiempo es ligeramente mayor usando más procesos. Este comportamiento se debe primordialmente a que la transformación del negativo no es una operación muy complicada y la utilización de más procesos retrasa la obtención de resultados, pues empleando más nodos se invertirá más tiempo en la distribución y colección de datos que en el cálculo en sí. Otras trasformaciónes puntuales más complejas tendrán mejores resultados.

Podemos ultimar que, lo mejor es utilizar alrededor de 8 procesos para manipular a imágenes de tamaño grande. No se hicieron pruebas con imágenes de mayor tamaño por las restricciones del cluster, pero esperaríamos un comportamiento similar para imágenes más grandes.

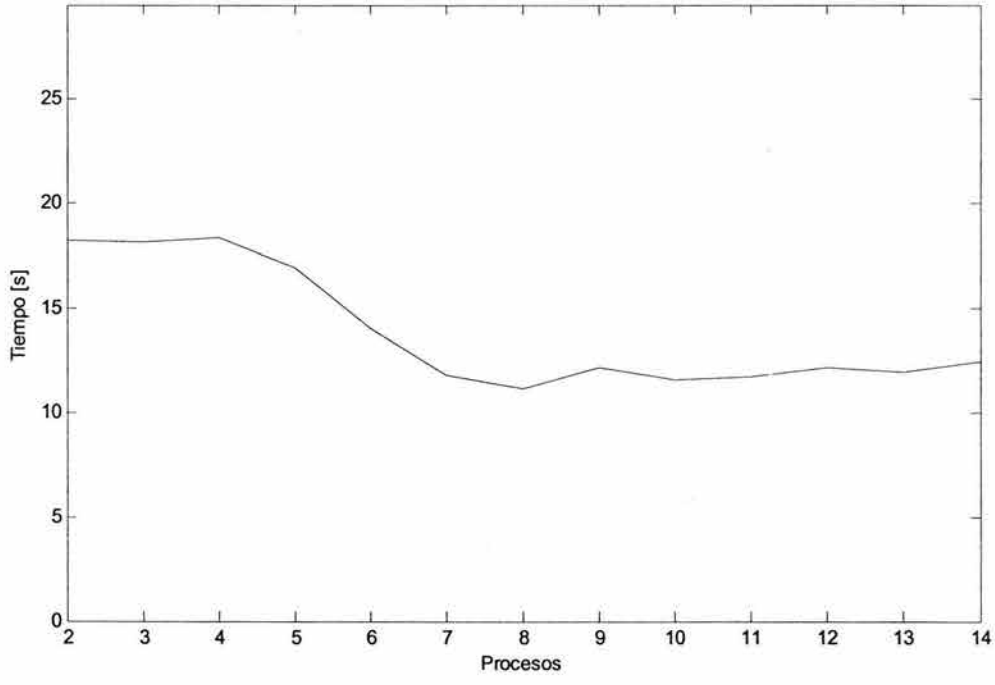


Figura 5.30. Gráfica de tiempo de ejecución contra número de procesos, utilizando una imagen de entrada formada por 2500000 píxeles.

5.4.3 Filtrado espacial.

5.4.3.1 Introducción.

En el filtrado, a diferencia del procesamiento puntual cuyos métodos operan individualmente sobre cada píxel de una imagen, se trabaja sobre vecindarios. Un

vecindario de un píxel es un grupo de píxeles contiguos a éste y puede ser visto como una subimagen dentro de la imagen a procesar.

El hecho de considerar los píxeles de una vecindad hace que las técnicas de procesamiento basadas en una región tengan un mayor costo de cálculo numérico que las técnicas basadas en un solo punto. Este costo dependerá del tamaño de la vecindad a considerar y del tamaño de la imagen, es por ello que en el en filtrado comúnmente se utilizan implementaciones paralelas que reduzcan el tiempo de cálculo.

En la implantación de estas técnicas de procesamiento regional la convolución bidimensional juega un papel preponderante ya que permite llevar a cabo la aplicación de un filtro sobre una imagen. En procesamiento de imágenes, la convolución corresponde a la extensión del caso unidimensional, mediante la cual una señal cualquiera podía ser procesada con un filtro arbitrario con una respuesta al impulso conocida. En este ámbito los filtros, máscaras, o ventanas son básicamente arreglos bidimensionales en los que los valores de los coeficientes determinan la naturaleza de la transformación.

La convolución permite que las imágenes sean procesadas por medio de filtros, esta operación está formulada de la siguiente manera. Para una imagen $i(x,y)$ de dimensiones AxB y un filtro $f(x,y)$ de dimensiones CxD, asumiendo que ambos arreglos son periódicos con periodos M en la dirección "x" y N en la dirección "y", tales que M ≥ A+C-1 y N ≥ B+D-1. La convolución discreta bidimensional de $i(x,y)$ y $f(x,y)$ está definida por:

$$i(x,y)*f(x,y)=\frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}i(m,n)f(x-m,y-n), \qquad (5.4\text{-}3)$$

para los valores de x = 0, 1, ..., M-1 y de y = 0, 1, ..., N-1. La matriz de dimensiones MxN de la expresión (5.4-3) es un periodo de la convolución discreta bidimensional.

La implantación de la expresión (5.4-3) se hace de manera directa cuando las dimensiones del filtro y de la imagen son pequeñas, pues en tales casos el costo computacional no es exagerado, sin embargo, cuando se tienen filtros o imágenes de gran tamaño, lo más recomendable es implantar esta ecuación de convolución mediante la utilización de la transformada rápida de Fourier.

Nuestra implementación paralela utiliza la transformada rápida de Fourier (TRF) para realizar el cálculo de la convolución discreta bidimensional, antes de abordar el código del programa, explicaremos brevemente la trasformada de Fourier, la TRF y la forma en que la convolución puede ser obtenida a través de la TRF.

5.4.3.1.1 La transformada discreta de Fourier.

Antes de abordar el caso bidimensional, analizaremos el caso de funciones de una sola variable, para ello, consideremos que una función continua f es muestreada a intervalos espaciados Δx, obteniéndose la secuencia de N muestras $\{f(x_0), f(x_0+\Delta x), f(x_0+2\Delta x), \ldots, f(x_0+(N-1)\Delta x)\}$. Así, si definimos $f(x)=f(x_0+x\Delta x)$ para x = 0, 1, ..., N-1, entonces $\{f(0), f(1), \ldots, f(N-1)\}$ representa alguna secuencia de N muestras uniformemente espaciadas de la correspondiente función continua.

Tomando en cuenta lo anterior, la trasformada discreta de Fourier de $f(x)$, denotada $F\{f(x)\}$, está definida por

$$F\{f(x)\}=F(u)=\sum_{x=0}^{N-1} f(x)\exp\left(\frac{-j2\pi ux}{N}\right) \qquad (5.4\text{-}4)$$

para u = 0, 1, ..., N-1.

Dada $F(u)$, $f(x)$ puede ser obtenida mediante la trasformada inversa de Fourier

$$F^{-1}\{F(u)\}=f(x)=\frac{1}{N}\sum_{u=0}^{N-1} F(u)\exp\left(\frac{j2\pi ux}{N}\right) \qquad (5.4\text{-}5)$$

para x = 0, 1, ..., N-1.

Para el caso de funciones de dos variables, supongamos que la función discreta $f(x,y)$ es obtenida como resultado de muestrear una función continua f en una red o malla bidimensional, con espaciamientos de Δx y Δy en los ejes "x" y "y", respectivamente. En otras palabras, $f(x,y)$ representa las muestras de la función continua en $f(x_0+x\Delta x, y_0+y\Delta y)$ para x = 0, 1, ..., M-1 y para y = 0, 1, ..., N-1. La trasformada de Fourier de $f(x,y)$ está definida por

$$F\{f(x,y)\}=F(u,v)=\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\exp\left(-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)\right) \qquad (5.4\text{-}6)$$

para u = 0, 1, ..., M-1, v = 0, 1, ..., N-1.

Dada $F(u,v)$, $f(x,y)$ puede ser obtenida mediante la trasformada inversa de Fourier

$$F^{-1}\{F(u,v)\}=f(x,y)=\frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)\exp\left(j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)\right) \qquad (5.4\text{-}7)$$

para x = 0, 1, ..., M-1, y = 0, 1, ..., N-1.

5.4.3.1.2 La trasformada rápida de Fourier (TRF).

El número de multiplicaciones y adiciones complejas requeridas para implementar la expresión (5.4-4) es proporcional a N^2. Esto es, para cada uno de los N valores de u, se requieren N multiplicaciones de $f(x)$ por la exponencial compleja y N-1 sumas de los anteriores productos.

Una correcta descomposición de la expresión (5.4-4) puede hacer que el número de multiplicaciones y adiciones necesarias para su cálculo sea proporcional a $N \log_2 N$. El procedimiento de descomposición es conocido como el algoritmo de la TRF. Obviamente la TRF representa una gran ventaja en cuanto a cálculo se refiere sobre la implementación directa de la trasformada de Fourier, particularmente cuando N es relativamente grande. Por ejemplo, pensemos que el cálculo de la TRF de una función discreta compuesta por 2048 muestras toma 6 segundos en una cierta computadora. La misma máquina necesitaría alrededor de 18 minutos en realizar el cálculo de la transformada de Fourier de la misma función utilizando la expresión (5.4-4).

A continuación mostraremos el desarrollo del algoritmo de la TRF para el caso unidimensional (1-D), posteriormente veremos que la TRF en dos dimensiones (2-D) puede obtenerse a partir de la TRF 1-D.

Para mayor claridad en el desarrollo, llevaremos la expresión (5.4-4) a la forma

$$F(u) = \sum_{x=0}^{N-1} f(x) W_N^{ux} \tag{5.4-8}$$

donde

$$W_N = \exp\left(\frac{-j2\pi}{N}\right) \tag{5.4-9}$$

y con la restricción de que se debe cumplir que

$$N = 2^n$$

donde n es un entero positivo. Ya que N puede ser expresado como

$$N = 2M \tag{5.4-10}$$

donde M es también un entero positivo. La sustitución de la expresión (5.4-10) en la expresión (5.4-8) conduce a

$$F(u)=\sum_{x=0}^{2M-1} f(x)W_{2M}^{ux}=\sum_{x=0}^{M-1} f(2x)W_{2M}^{u(2x)}+\sum_{x=0}^{M-1} f(2x+1)W_{2M}^{u(2x+1)}. \qquad (5.4\text{-}11)$$

De la expresión (5.4-9) sabemos que $W_{2M}^{2ux}=W_M^{ux}$, así (5.4-11) puede ser expresada en la forma

$$F(u)=\sum_{x=0}^{M-1} f(2x)W_M^{ux}+\sum_{x=0}^{M-1} f(2x+1)W_M^{ux}W_{2M}^{u}. \qquad (5.4\text{-}12)$$

Al definir

$$F_{par}(u)=\sum_{x=0}^{M-1} f(2x)W_M^{ux} \qquad (5.4\text{-}13)$$

para u = 0, 1, ..., M-1, y

$$F_{impar}(u)=\sum_{x=0}^{M-1} f(2x+1)W_M^{ux} \qquad (5.4\text{-}14)$$

para u = 0, 1, ..., M-1, la expresión (5.4-12) se reduce a

$$F(u)=F_{par}(u)+F_{impar}(u)W_{2M}^{u}. \qquad (5.4\text{-}15)$$

Además, ya que $W_M^{u+M}=W_M^u$ y $W_{2M}^{u+M}=-W_{2M}^u$, las expresiones (5.4-13)-(5.4-15) llevan a

$$F(u+M)=F_{par}(u)-F_{impar}(u)W_{2M}^{u}. \qquad (5.4\text{-}16)$$

Un análisis cuidadoso de las expresiones (5.4-13)-(5.4-16) revela algunas propiedades interesantes de éstas. Una transformada de N puntos puede ser calculada al dividir la expresión original en dos partes, como la indican las expresiones (5.4-15) y (5.4-16). El cálculo de la primera mitad de $F(u)$ requiere la evaluación de dos trasformadas de N/2 puntos dadas por las expresiones (5.4-13) y (5.4-14). Los valores resultantes de $F_{par}(u)$ y $F_{impar}(u)$ son sustituidos en la expresión (5.4-15) para obtener $F(u)$ para u = 0, 1, ..., (N/2-1). La otra mitad puede ser directamente obtenida de la expresión (5.4-16) sin cálculo de trasformadas adicionales.

Implementación computacional de la TRF.

En las siguientes líneas mostraremos una forma en que la anterior teoría de la TRF puede ser llevada a la práctica, en la actualidad existen muchas formas de hacerlo, la que aquí mostraremos resulta conveniente para sistemas con poca memoria, no requiere de datos preprocesados, y la complejidad de su desarrollo con relación a su velocidad de cálculo es equilibrada.

El principal punto a considerar es que los datos de entrada, que forman un arreglo, deben ser reordenados para poder realizar las aplicaciones sucesivas de las expresiones (5.4-13) y (5.4-14). Para comprender mejor la forma en que los datos deben ser reacomodados consideraremos un ejemplo. Supongamos que deseamos calcular la trasformada de Fourier de una función de 8 puntos, $\{f(0), f(1), \ldots, f(7)\}$, utilizando las expresiones (5.4-13)-(5.4-16). La expresión (5.4-13) utiliza las muestras con argumentos pares, $\{f(0), f(2), f(4), f(6)\}$, y la expresión (5.4-14) usa las muestras con argumentos impares, $\{f(1), f(3), f(5), f(7)\}$. Sin embargo, cada transformada de 4 puntos se calcula como dos trasformadazas de 2 puntos, las que también requieren del uso de (5.4-13) y (5.4-14). De esta forma, para calcular la TRF del muestrario $\{f(0), f(2), f(4), f(6)\}$, debemos dividirlo en su parte par $\{f(0), f(4)\}$ y su parte impar $\{f(2), f(6)\}$. De manera análoga, el otro muestrario debe dividirse en $\{f(1), f(5)\}$ para la expresión (5.4-13) y en $\{f(3), f(7)\}$ para la expresión (5.4-14). Ya no es necesario otro reacomodo ya que cada uno de los grupos de dos muestras resultantes se considera que posee un elemento par y uno impar. La combinación de estos resultados sugiere que los datos de entrada deben ser reorganizados en la forma $\{f(0), f(4), f(2), f(6), f(1), f(5), f(3), f(7)\}$. La TRF opera sobre este último grupo de muestras de la manera que se muestra en la figura 5.31. En el primer nivel se calculan cuatro trasformadas de 2 puntos utilizando: $\{f(0), f(4)\}$, $\{f(2), f(6)\}$, $\{f(1), f(5)\}$ y $\{f(3), f(7)\}$. El siguiente nivel, utiliza estos resultados para formar dos transformadas de 4 puntos, y el último nivel usa estos dos últimos resultados para calcular la trasformada originalmente deseada.

El procedimiento generalizado que permite reorganizar el arreglo de datos entrada sigue una regla simple conocida como *inversión de bits*. Si x representa algún argumento válido para $f(x)$, el argumento correspondiente en el arreglo reordenado se obtiene al expresar a x en binario e invertir sus bits. Por ejemplo, si $N = 2^3$, el quinto elemento en el arreglo original, $f(4)$, se convierte en el segundo elemento del arreglo reordenado, esto debido a que 4 = 100_2 se convierte en 001_2 = 1 cuando se invierten sus bits. La tabla 5.9 ilustra este procedimiento para $N = 8$. Si este arreglo reordenado es utilizado en el cálculo de la TRF, el resultado arrojado por ésta será la trasformada de Fourier del arreglo de entrada original en el orden correcto.

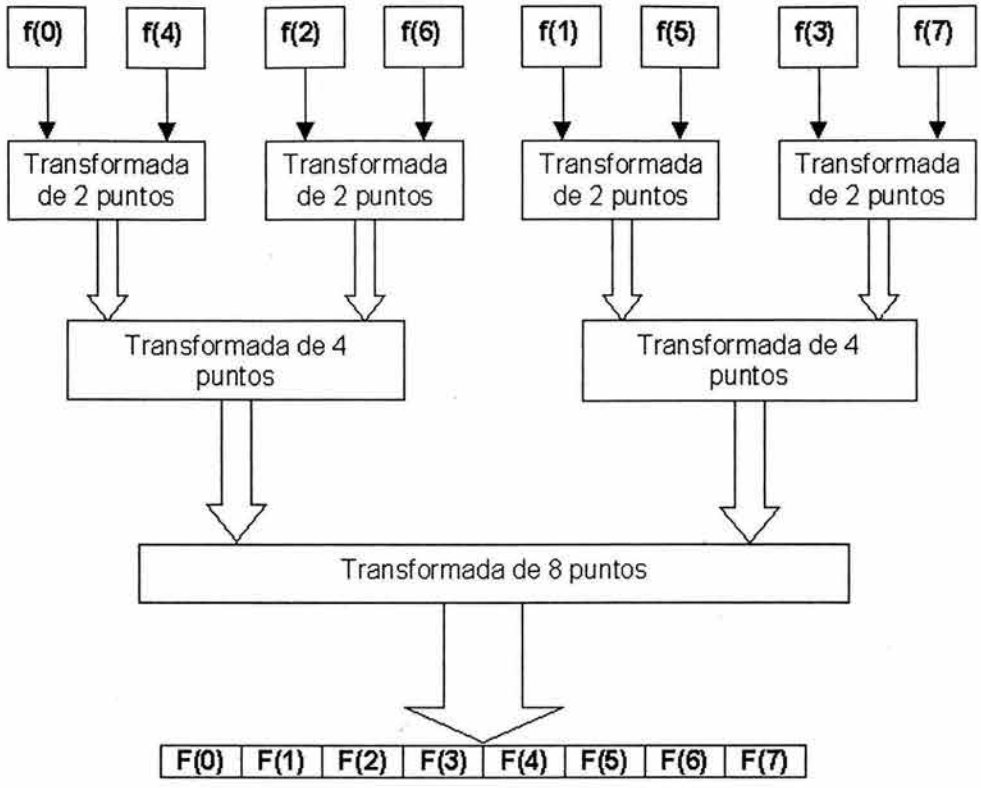


Figura 5.31. Forma en que opera el algoritmo de la TRF sobre el arreglo de entrada reordenado.

Argumento original	Arreglo de entrada	Argumento de bits invertidos	Arreglo reordenado
000	$f(0)$	000	$f(0)$
001	$f(1)$	100	$f(4)$
010	$f(2)$	010	$f(2)$
011	$f(3)$	110	$f(6)$
100	$f(4)$	001	$f(1)$
101	$f(5)$	101	$f(5)$
110	$f(6)$	011	$f(3)$
111	$f(7)$	111	$f(7)$

Tabla 5.9. Ejemplo del reacomodo del arreglo de entrada para realizar el cálculo de la trasformada de Fourier a través de la TRF.

La extensión de este algoritmo de la TRF a 2-D se basa principalmente en una propiedad de la trasformada de Fourier conocida como **propiedad de separación** y, debido a que la trasformada de Fourier y la TRF arrojan los mismos resultados para funciones cuyo número de muestras es potencia de dos, esta propiedad puede extenderse y ser utilizada para el cálculo de la TRF 2-D.

5.4.3.1.3 Propiedad de separación de la trasformada de Fourier.

La trasformada discreta de Fourier en 2-D de la expresión (5.4-6) puede rescribirse en la forma

$$F(u,v) = \sum_{x=0}^{M-1} \exp\left(\frac{-j2\pi ux}{M}\right) \sum_{y=0}^{N-1} f(x,y) \exp\left(\frac{-j2\pi vy}{N}\right) \qquad (5.4\text{-}17)$$

para u = 0, 1, ..., M-1, v = 0, 1, ..., N-1.

La ventaja principal de la propiedad de separación es que $F(u,v)$ o $f(x,y)$ pueden ser obtenidas en dos pasos por medio de aplicaciones sucesivas de la trasformada de Fourier 1-D o de su inversa, respectivamente. Para mayor claridad, expresaremos (5.4-17) como sigue

$$F(u,v) = \sum_{x=0}^{M-1} F(x,v) \exp\left(\frac{-j2\pi ux}{M}\right) \qquad (5.4\text{-}18)$$

donde

$$F(x,v) = \sum_{y=0}^{N-1} f(x,y) \exp\left(\frac{-j2\pi vy}{N}\right). \qquad (5.4\text{-}19)$$

Para cada valor de x, la expresión (5.4-19) es una trasformada en 1-D, con valores de frecuencia v = 0, 1, ..., N-1. Por lo tanto, la función bidimensional $F(x,v)$ se obtiene al calcular la trasformada de cada uno de los renglones de $f(x,y)$. El resultado deseado, $F(u,v)$, será conseguido al calcular la transformada de cada una de las columnas de $F(x,v)$, según lo indica la expresión (5.4-18). Todo este desarrollo realizado para la transformada, es válido para la transformada inversa dada por la expresión (5.4-7) con la diferencia que después de la transformación por renglones es necesario multiplicar a la función bidimensional resultante por el valor de N antes de realizar la trasformación por columnas.

5.4.3.1.4 La TRF inversa.

Hasta el momento, no hemos hablado de la TRF inversa, la razón de ello es que cualquier algoritmo que implemente la TRF también puede ser usado, con mínimos cambios en la entrada, para llevar a cabo el cálculo de la inversa. Para

mostrar esto, obtendremos el complejo conjugado de la expresión (5.4-5) y multiplicaremos ambos lados por N:

$$Nf^*(x) = \sum_{u=0}^{N-1} F^*(u)\exp\left(\frac{-j2\pi ux}{N}\right). \qquad (5.4\text{-}20)$$

donde * denota el complejo conjugado.

Al comparar este resultado con (5.4-4), es posible notar que el miembro de la derecha de la expresión (5.4-20) tiene la forma de la trasformada de la expresión (5.4-4). Por lo que, si proporcionamos a la función $F^*(u)$ como entrada al algoritmo que calcula la TRF conseguiremos como resultado $Nf^*(x)$. Si tomamos el complejo conjugado y dividimos entre N tendremos la transformada inversa deseada.

5.4.3.1.5 Teorema de la convolución.

Este teorema establece una la relación muy importante que existe entre la transformada de Fourier y la convolución, para 2-D que es el caso que nos interesa, su enunciado es el siguiente. Si $f(x,y)$ tiene por trasformada de Fourier a $F(u,v)$ y $g(x,y)$ tiene por transformada de Fourier a $G(u,v)$, entonces $f(x,y)*g(x,y)$ tiene a $F(u,v)G(u,v)$ por transformada de Fourier. Este teorema de la convolución comúnmente es expresado como

$$f(x,y)*g(x,y) \Leftrightarrow F(u,v)G(u,v)$$

lo que indica que la convolución puede ser obtenida también al calcular la trasformada inversa del producto $F(u,v)G(u,v)$.

5.4.3.1.6 Cálculo de la convolución a través de la TRF.

La realización de un algoritmo que calcule la convolución en 2-D por medio de la TRF en 1-D se puede deducir fácilmente con base en las definiciones, los teoremas y las propiedades que hasta el momento hemos visto. En las siguientes líneas veremos la forma en que unificamos estos conocimientos.

La implementación paralela que programamos para aplicar los procesos de filtrado a las imágenes mediante la convolución, utiliza la TRF 1-D de forma reiterada para obtener la transformada de Fourier en 2-D y mediante el teorema de la convolución calcula la trasformación que el filtro realiza a la imagen.

De forma somera, los pasos a seguir en el cálculo de la convolución a través de la TRF son[†]:

1) Calcular la transformada de Fourier en 2-D de la imagen.
2) Calcular la transformada de Fourier en 2-D del filtro.
3) Multiplicar, de forma puntual, las transformadas obtenidas en los puntos (1) y (2).
4) Calcular la transformada inversa de Fourier en 2-D del producto conseguido en el del punto (3).

Para conseguir el punto (1) nuestra implementación realiza el siguiente procedimiento.

1.1) Obtiene la TRF 1-D de cada renglón de la imagen.
1.2) Transpone la matriz conseguida en (1.1).
1.3) Obtiene la TRF 1-D de cada renglón de la matriz del punto (1.2).
1.4) Transpone la matriz conseguida en (1.3).

Para el punto (2) se procede como con el (1), sólo que con el filtro en lugar de con la imagen.

El punto (3) requiere una multiplicación matricial punto a punto, ésta puede ser expresada de la siguiente forma. Dadas dos matrices, $f(x,y)$ y $g(x,y)$, ambas con dimensión AxB, su multiplicación puntual es otra matriz $p(x,y)$ dada por

$$p(x,y) = f(x,y)g(x,y)$$

para x = 0, 1, ..., A-1, y = 0, 1, ..., B-1.

En el punto (4) nuestra implementación realiza lo siguiente:

4.1) Obtiene el complejo conjugado de cada elemento del arreglo bidimensional proporcionado.
4.2) Al arreglo resultante del punto (4.1) le aplica el mismo proceso que a la imagen en el punto (1).
4.3) Divide cada a cada elemento del arreglo obtenido en (4.2) entre MN, donde MxN es la resolución de la matriz de convolución ampliada que se obtiene a partir de la modificación que se efectúa en las matrices de entrada con el objeto de que cumplan con las restricciones impuestas por la TRF y por la propia convolución.

[†] Es necesario aclarar que como el algoritmo de la TRF tiene la restricción de que el número de muestras de la función a la que se la va a calcular la trasformada debe ser potencia de dos, nuestra implementación paralela agrega, de ser necesario, ceros a los arreglos de entrada para poder cumplir con esta restricción y con la impuesta por la propia operación de convolución con respecto al periodo de los las funciones. El agregar ceros no altera de forma global el resultado de obtener la convolución por medio de la TRF.

La matriz del punto (4.3), no es otra sino la resultante de la convolución de la imagen con el filtro.

Hasta este punto, todas las aplicaciones que hemos desarrollado se adecuan a una de las técnicas de programación paralela. En la medida que un determinado algoritmo requiere una comunicación más versátil de la que provee una determinada técnica, es necesario crear aplicaciones híbridas que mezclen y coordinen más de una de estas técnicas para llevar a cabo su implementación.

Para la realización de este programa utilizamos dos técnicas de programación paralela, el resultado es una aplicación híbrida de las técnicas: Divide y Conquista y Maestro-Esclavo.

Independientemente de la técnica utilizada, decidimos distribuir el trabajo entre los distintos procesos conforme a la demanda, en otras palabras, no se asigna más trabajo a un proceso hasta que termine con el que inicialmente se le asignó.

Ya con estos antecedentes, pasamos directamente a inspeccionar el código fuente de esta implementación paralela que adjuntamos en las próximas líneas. Después de éste discutiremos algunos comentarios adicionales para reafirmarán algunos puntos de interés.

5.4.3.2 Código de la aplicación paralela.

```
/*************************************************************/
/*            TÉCNICAS DE PROGRAMACIÓN PARALELA              */
/*                                                           */
/*PROGRAMA:    conv2D.c                                      */
/*                                                           */
/*AUTORES:     Christian González,                           */
/*             Lissette González y                           */
/*             Eryk Ramírez.                                 */
/*                                                           */
/*DESCRIPCIÓN: Calcula la convolución bidimensional de dos   */
/*             matrices mediante la aplicación de la         */
/*             Transformada Rápida de Fourier utilizando     */
/*             las técnicas de Programación Paralela:        */
/*             Maestro-Esclavo y Divide y Conquista.         */
/*                                                           */
/*************************************************************/

/* Bibliotecas */
//mpi.h es necesaria para utilizar las funciones de la MPI.
#include <mpi.h>
#include <stdio.h>
```

```
#include <stdlib.h>

/* Definiciones para el preprocesador */
//Definimos el valor de Pi.
#define PI 3.141592654

/* Declaraciones de tipos */
//El tipo de dato Resolucion se utiliza para almacenar las
//dimensiones de las matrices que intervienen el cálculo de
//la convolución y el tipo Exponente se usa para guardar los
//exponentes correspondientes a las potencias de dos más
//pequeñas que deberían tener por dimensión las matrices
//involucradas en la convolución para poder aplicar el
//algoritmo de la TRF.
typedef struct{
     int M,N;
}Resolucion,Exponente;
//El tipo de dato Complejo lo utilizamos para facilitar el
//manejo de números complejos.
typedef struct{
     double real,imag;
}Complejo;

/* Prototipos de función */
void conv2D(char**,Resolucion*,Resolucion*,
                          Resolucion*,Resolucion*,Exponente*);
void divideYConquistaDivisor1(char*,Resolucion*,int);
void divideYConquistaDivisor2(FILE*,int,int,int);
void divideYConquistaDivisor3(char*,Resolucion*,int);
void divideYConquistaSolucion1(int,int,int,int);
void divideYConquistaSolucion2(int,int,int);
void divideYConquistaSolucion3(int,int,int,int);
void divideYConquistaSolucion4(int,int,int);
void divideYConquistaSolucion5(int,int,int,int,int);
FILE *divideYConquistaIntegrador1(int,int,int,int);
void divideYConquistaIntegrador2(char*,
                                        char*,int,int,int,int);
FILE *maestroEsclavoMaestro(FILE*,FILE*,Resolucion*,int);
void maestroEsclavoEsclavo(int,int);
void tRF(Complejo*,int);
Complejo prodComplejo(Complejo,Complejo);
unsigned char redondeo(double);
FILE *abre(char*,char*);
void cierra(FILE*,char*);
void *reservaMemoria(size_t);
/* Función principal */
int main(int argc,char *argv[]){
```

```
Resolucion resImagen,resFiltro,resConv,resDeseada={1,1};
Exponente exp={0,0};
int rango;
//Iniciamos un cómputo con MPI.
MPI_Init(&argc,&argv);
     //Cada proceso obtiene su rango.
     MPI_Comm_rank(MPI_COMM_WORLD,&rango);
     //Se comprueba que el número de argumentos sea el
     //correcto, de lo contrario se finaliza el
     //programa.
     if(!rango&&argc!=9){
          printf("Formato de ejecución:\n
                  conv2D
                         <archivo_de_imagen>
                         <num_renglones_imagen>
                         <num_columnas_imagen>\n
                         <archivo_de_filtro>
                         <num_renglones_filtro>
                         <num_columnas_filtro>
                         <archivo_imagen_de_salida>
                         <archivo_conv2D>\n");
          MPI_Abort(MPI_COMM_WORLD,1);
     }
     //Obtenemos el tamaño de la imagen y del
     //filtro.
     resImagen.M=atoi(*(argv+2));
     resImagen.N=atoi(*(argv+3));
     resFiltro.M=atoi(*(argv+5));
     resFiltro.N=atoi(*(argv+6));
     //Calculamos las dimensiones que tendrá la matriz
     //resultado de la convolución de la imagen con el
     //filtro.
     resConv.M=resImagen.M+resFiltro.M-1;
     resConv.N=resImagen.N+resFiltro.N-1;
     //Se determinan las dimensiones mínimas que deben
     //tener las matrices que intervienen en la
     //convolución para que sea posible aplicar el
     //algoritmo de la TRF.
     while(resDeseada.M<resConv.M)
               exp.M++,resDeseada.M<<=1;
     while(resDeseada.N<resConv.N)
               exp.N++,resDeseada.N<<=1;
     //Convolución en 2-D.
     conv2D(argv,&resImagen,&resFiltro,
                              &resConv,&resDeseada,&exp);
//Finalizamos el cómputo de MPI.
MPI_Finalize();
```

```
    return 0;
}

/* Esta función es ejecutada por todos los procesos, en ella
 * se lleva a cabo el cálculo de la convolución en 2-D con la
 * ayuda de las técnicas Maestro-Esclavo y Divide y
 * Conquista .
 */
void conv2D(char **argv,Resolucion *resImagen,
            Resolucion *resFiltro,Resolucion *resConv,
                    Resolucion *resDeseada,Exponente *exp){
    FILE *temporal[4];
    int rango,tamanioGrupo;
    //Cada proceso obtiene su rango y el número de
    //procesos del grupo.
    //Dependiendo del valor de su rango, cada proceso
    //determina qué funciones deberá ejecutar.
    MPI_Comm_size(MPI_COMM_WORLD,&tamanioGrupo);
    MPI_Comm_rank(MPI_COMM_WORLD,&rango);

    /*****************************************************/
    /*                  Divide y Conquista               */
    /*****************************************************/
    //En la siguiente sección condicional se aplica cuatro
    //veces la técnica Divide y Conquista, las dos primeras
    //obtienen la trasformada de Fourier en 2-D de la imagen
    //de entrada (una aplicación para obtener la TRF por
    //renglones y otra para obtener la TRF por columnas) y
    //las otras hacen lo mismo pero con el filtro.
    //Cada aplicación de la técnica obtiene una trasformada
    //1-D.
    if(!rango){
        divideYConquistaDivisor1(*(argv+1),resImagen,
                                          tamanioGrupo-1);
        MPI_Barrier(MPI_COMM_WORLD);
        temporal[2]=divideYConquistaIntegrador1(
                          resDeseada->N,resDeseada->M,
                          tamanioGrupo-1,tamanioGrupo-2);
        MPI_Barrier(MPI_COMM_WORLD);
        divideYConquistaDivisor3(*(argv+4),
                                  resFiltro,tamanioGrupo-1);
        MPI_Barrier(MPI_COMM_WORLD);
        temporal[4]=divideYConquistaIntegrador1(
                          resDeseada->N,resDeseada->M,
                          tamanioGrupo-1,tamanioGrupo-2);
    }else if(rango<tamanioGrupo-1){
        divideYConquistaSolucion1(resImagen->N,
```

```
                    resDeseada->N,exp->N,tamanioGrupo-1);
     MPI_Barrier(MPI_COMM_WORLD);
     divideYConquistaSolucion2(resDeseada->M,exp->M,0);
     MPI_Barrier(MPI_COMM_WORLD);
     divideYConquistaSolucion3(resFiltro->N,
                    resDeseada->N,exp->N,tamanioGrupo-1);
     MPI_Barrier(MPI_COMM_WORLD);
     divideYConquistaSolucion2(resDeseada->M,exp->M,0);
}else{
     temporal[1]=divideYConquistaIntegrador1(
                         resImagen->M,
                         resDeseada->N,0,tamanioGrupo-2);
     MPI_Barrier(MPI_COMM_WORLD);
     divideYConquistaDivisor2(
                            temporal[1],resDeseada->N,
                            resImagen->M,tamanioGrupo-1);
     MPI_Barrier(MPI_COMM_WORLD);
     temporal[3]=divideYConquistaIntegrador1(
                         resFiltro->M,
                         resDeseada->N,0,tamanioGrupo-2);
     MPI_Barrier(MPI_COMM_WORLD);
     divideYConquistaDivisor2(
                            temporal[3],resDeseada->N,
                            resFiltro->M,tamanioGrupo-1);
}

/*****************************************************/
/*                  Maestro-Esclavo                  */
/*****************************************************/
//En la siguiente sección condicional se aplica la
//técnica Maestro-Esclavo para realizar la
//multiplicación puntual entre las trasformadas de
//Fourier 2-D de la imagen y del filtro. Adicionalmente,
//se toma el complejo conjugado de este producto para
//que la matriz resultante sea la adecuada para
//obtenerle su transformada inversa.
MPI_Barrier(MPI_COMM_WORLD);
if(rango)
     maestroEsclavoEsclavo(resDeseada->N,0);
else
     temporal[1]=maestroEsclavoMaestro(
                                 temporal[2],temporal[4],
                                 resDeseada,tamanioGrupo);


/*****************************************************/
/*                  Divide y Conquista               */
```

```
     /*******************************************************/
     //En esta última sección se aplica dos veces la técnica
     //Divide y Conquista para conseguir la trasformada
     //inversa de Fourier en 2-D de la matriz resultante de
     //la técnica Maestro-Esclavo. La primera aplicación
     //obtiene la TRF por renglones y la segunda la calcula
     //por columnas. Finalmente, el resultado es dividido
     //entre la constante MN resultando así la matriz
     //convolución de la imagen con el filtro.Cada aplicación
     //de la técnica obtiene una trasformada 1-D.
     MPI_Barrier(MPI_COMM_WORLD);
     if(!rango){
          divideYConquistaDivisor2(
                                temporal[1],resDeseada->M,
                                resDeseada->N,tamanioGrupo-1);
          MPI_Barrier(MPI_COMM_WORLD);
          divideYConquistaIntegrador2(
                               *(argv+7),*(argv+8),
                               resConv->N,resConv->M,
                               tamanioGrupo-1,tamanioGrupo-2);
     }else if(rango<tamanioGrupo-1){
          divideYConquistaSolucion4(resDeseada->N,
                                    exp->N,tamanioGrupo-1);
          MPI_Barrier(MPI_COMM_WORLD);
          divideYConquistaSolucion5(
                        resDeseada->M,resConv->M,
                        exp->M,resDeseada->M*resDeseada->N,0);
     }else{
          temporal[2]=divideYConquistaIntegrador1(
                              resDeseada->M,
                              resDeseada->N,0,tamanioGrupo-2);
          MPI_Barrier(MPI_COMM_WORLD);
          divideYConquistaDivisor2(
                                temporal[2],resConv->N,
                                resDeseada->M,tamanioGrupo-1);
     }
}

/* Esta función, es ejecutada por un Proceso Divisor, en
 * conjunción con otras dos funciones (una integradora y otra
 * solución), permite utilizar la técnica Divide y Conquista.
 * Se encarga de enviar trabajo a los Procesos Solución. El
 * Proceso Integrador es el que le indica qué Procesos
 * Solución se encuentran libres para enviarles trabajo.
 */
void divideYConquistaDivisor1(char *nomArch,
                              Resolucion *res,int mayorRango){
```

```
    MPI_Status status;
    FILE *archivo=abre(nomArch,"r");
    int i=1,libre;
    unsigned char *ren;
    //Reserva memoria para envío de datos a los Procesos
    //Solución.
    ren=(unsigned char *)reservaMemoria(res->N);
    //Envía un renglón de la matriz a cada uno de los
    //Procesos Solución.
    while(i<mayorRango&&i<=res->M){
        fread(ren,1,res->N,archivo);
        MPI_Send(ren,res->N,MPI_UNSIGNED_CHAR,
                                    i,i-1,MPI_COMM_WORLD);
        i++;
    }
    i--;
    //Envía renglones individuales a los Procesos Solución
    //que ya han terminado su trabajo (están libres) hasta
    //que se terminen los renglones de la matriz.
    while(i<res->M){
        //Recibe, por parte del Proceso Integrador, el
        //rango de un Proceso Solución que se encuentra
        //libre.
        MPI_Recv(&libre,1,MPI_INT,MPI_ANY_SOURCE,
                        MPI_ANY_TAG,MPI_COMM_WORLD,&status);
        fread(ren,1,res->N,archivo);
        //Envía trabajo al Proceso Solución libre.
        MPI_Send(ren,res->N,MPI_UNSIGNED_CHAR,
                                libre,i++,MPI_COMM_WORLD);
    }
    //Envía la señal de paro a cada uno de los Procesos
    //Solución.
    for(i=1;i<mayorRango;i++)
        MPI_Send(NULL,0,MPI_UNSIGNED_CHAR,
                                    i,0,MPI_COMM_WORLD);
    //Se libera el buffer de envío.
    free(ren);
    //Se cierra el archivo de datos.
    cierra(archivo,nomArch);
}

/* Esta función, es ejecutada por un Proceso Divisor, en
 * conjunción con otras dos funciones (una integradora y otra
 * solución), permite utilizar la técnica Divide y Conquista.
 * Se encarga de enviar trabajo a los Procesos Solución. El
 * Proceso Integrador es el que le indica qué Procesos
 * Solución se encuentran libres para enviarles trabajo.
```

```
 */
void divideYConquistaDivisor2(FILE *archivo,int numRen,
                                    int numCol,int mayorRango){
     MPI_Status status;
     Complejo *ren;
     int i=1,libre;
     //Reserva memoria para envío de datos a los Procesos
     //Solución.
     ren=(Complejo *)reservaMemoria(numCol*sizeof(Complejo));
     //Coloca el apuntador al principio del archivo.
     rewind(archivo);
     //Envía un renglón de la matriz a cada uno de los
     //Procesos Solución.
     while(i<mayorRango&&i<=numRen){
          fread(ren,sizeof(Complejo),numCol,archivo);
          MPI_Send(ren,numCol<<1,MPI_DOUBLE,
                                        i,i-1,MPI_COMM_WORLD);
          i++;
     }
     i--;
     //Envía renglones individuales a los Procesos Solución
     //que ya han terminado su trabajo (están libres) hasta
     //que se terminen los renglones de la matriz.
     while(i<numRen){
          //Recibe, por parte del Proceso Integrador, el
          //rango de un Proceso Solución que se encuentra
          //libre.
          MPI_Recv(&libre,1,MPI_INT,MPI_ANY_SOURCE,
                         MPI_ANY_TAG,MPI_COMM_WORLD,&status);
          fread(ren,sizeof(Complejo),numCol,archivo);
          //Envía trabajo al Proceso Solución libre.
          MPI_Send(ren,numCol<<1,MPI_DOUBLE,
                                    libre,i++,MPI_COMM_WORLD);
     }
     //Envía la señal de paro a cada uno de los Procesos
     //Solución.
     for(i=1;i<mayorRango;i++)
          MPI_Send(NULL,0,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
     //Se libera el buffer de envío.
     free(ren);
}

/* Esta función, es ejecutada por un Proceso Divisor, en
 * conjunción con otras dos funciones (una integradora y otra
 * solución), permite utilizar la técnica Divide y Conquista.
 * Se encarga de enviar trabajo a los Procesos Solución. El
 * Proceso Integrador es el que le indica qué Procesos
```

```
 * Solución se encuentran libres para enviarles trabajo.
 */
void divideYConquistaDivisor3(char *nomArch,
                              Resolucion *res,int mayorRango){
     MPI_Status status;
     FILE *archivo=abre(nomArch,"r");
     int i=1,libre;
     double *ren;
     //Reserva memoria para envío de datos a los Procesos
     //Solución.
     ren=(double *)reservaMemoria(res->N*sizeof(double));
     //Envía un renglón de la matriz a cada uno de los
     //Procesos Solución.
     while(i<mayorRango&&i<=res->M){
          fread(ren,sizeof(double),res->N,archivo);
          MPI_Send(ren,res->N,MPI_DOUBLE,
                                        i,i-1,MPI_COMM_WORLD);
          i++;
     }
     i--;
     //Envía renglones individuales a los Procesos Solución
     //que ya han terminado su trabajo (están libres) hasta
     //que se terminen los renglones de la matriz.
     while(i<res->M){
          //Recibe, por parte del Proceso Integrador, el
          //rango de un Proceso Solución que se encuentra
          //libre.
          MPI_Recv(&libre,1,MPI_INT,MPI_ANY_SOURCE,
                          MPI_ANY_TAG,MPI_COMM_WORLD,&status);
          fread(ren,sizeof(double),res->N,archivo);
          //Envía trabajo al Proceso Solución libre.
          MPI_Send(ren,res->N,MPI_DOUBLE,
                                    libre,i++,MPI_COMM_WORLD);
     }
     //Envía la señal de paro a cada uno de los Procesos
     //Solución.
     for(i=1;i<mayorRango;i++)
          MPI_Send(NULL,0,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
     //Se libera el buffer de envío.
     free(ren);
     //Se cierra el archivo de datos.
     cierra(archivo,nomArch);
}

/* Esta función, es ejecutada por un Proceso Solución, en
 * conjunción con otras dos funciones (una divisora y otra
 * integradora), permite utilizar la técnica Divide y
```

```
 * Conquista.
 * Recibe los vectores que son enviados por el Proceso
 * Divisor, les agrega los ceros necesarios para poder
 * aplicar el algoritmo de la TRF, les calcula su
 * transformada de Fourier y envía los resultados al Proceso
 * Integrador.
 */
void divideYConquistaSolucion1(int numElem,int tam,
                               int log2tam,int rangoIntegrador){
     Complejo *renComplejo;
     unsigned char *ren;
     MPI_Status status;
     int i;
     //Reserva memoria para recibir los datos que le envía el
     //Proceso Divisor.
     ren=(unsigned char *)reservaMemoria(numElem);
     //Se reserva memoria para enviar los resultados al
     //Proceso Integrador.
     renComplejo=(Complejo *)reservaMemoria(
                                        tam*sizeof(Complejo));
     while(1){
          //Recibe el vector de datos al que deberá
          //calcularle su TRF.
          MPI_Recv(ren,numElem,MPI_UNSIGNED_CHAR,
                        MPI_ANY_SOURCE,
                        MPI_ANY_TAG,MPI_COMM_WORLD,&status);
          //Se obtiene el número de elementos recibidos.
          MPI_Get_count(&status,MPI_UNSIGNED_CHAR,&numElem);
          //Si no se recibieron datos, el proceso termina su
          //trabajo.
          if(numElem){
               //El vector recibido debe expresarse como un
               //vector de números complejos para poder
               //llamar a la función tRF.
               for(i=0;i<numElem;i++)
                    renComplejo[i].real=ren[i],
                                        renComplejo[i].imag=0.;
               //Se agregan ceros para cumplir con la
               //restricción de la TRF que exige que el
               //número de muestras de la función de la que
               //se quiere obtener su trasformada sea
               //potencia de dos.
               for(;i<tam;i++)
                    renComplejo[i].real=
                                        renComplejo[i].imag=0.;
               //Cálculo de la TRF del vector 'renComplejo'.
               tRF(renComplejo,log2tam);
```

```
                //Envío de resultados al Proceso Integrador.
                MPI_Send(renComplejo,tam<<1,
                                MPI_DOUBLE,rangoIntegrador,
                                status.MPI_TAG,MPI_COMM_WORLD);
          }else{
                //Se libera el buffer de recepción.
                free(ren);
                //Se libera el buffer de envío.
                free(renComplejo);
                return;
          }
     }
}

/* Esta función, es ejecutada por un Proceso Solución, en
 * conjunción con otras dos funciones (una divisora y otra
 * integradora), permite utilizar la técnica Divide y
 * Conquista.
 * Recibe los vectores que son enviados por el Proceso
 * Divisor, les agrega los ceros necesarios para poder
 * aplicar el algoritmo de la TRF, les calcula su
 * transformada de Fourier y envía los resultados al Proceso
 * Integrador.
 */
void divideYConquistaSolucion2(int tam,int log2tam,
                                          int rangoIntegrador){
     MPI_Status status;
     Complejo *ren;
     int i,num;
     //Reserva memoria para envío y recepción de datos.
     ren=(Complejo *)reservaMemoria(tam*sizeof(Complejo));
     while(1){
          //Recibe el vector de datos al que deberá
          //calcularle su TRF.
          MPI_Recv(ren,tam<<1,MPI_DOUBLE,
                          MPI_ANY_SOURCE,
                          MPI_ANY_TAG,MPI_COMM_WORLD,&status);
          //Se obtiene el número de elementos recibidos.
          MPI_Get_count(&status,MPI_DOUBLE,&num);
          //Si no se recibieron datos, el proceso termina su
          //trabajo.
          if(num){
                //Se agregan ceros para cumplir con la
                //restricción de la TRF que exige que el
                //número de muestras de la función de la que
                //se quiere obtener su trasformada sea
                //potencia de dos.
```

```
                for(i=num>>1;i<tam;i++)
                    ren[i].real=ren[i].imag=0.;
                //Cálculo de la TRF del vector 'ren'.
                tRF(ren,log2tam);
                //Envío de resultados al Proceso Integrador.
                MPI_Send(ren,tam<<1,
                                MPI_DOUBLE,rangoIntegrador,
                                status.MPI_TAG,MPI_COMM_WORLD);
            }else{
                //Liberamos el buffer de comunicación.
                free(ren);
                return;
            }
        }
}

/* Esta función, es ejecutada por un Proceso Solución, en
 * conjunción con otras dos funciones (una divisora y otra
 * integradora), permite utilizar la técnica Divide y
 * Conquista.
 * Recibe los vectores que son enviados por el Proceso
 * Divisor, les agrega los ceros necesarios para poder
 * aplicar el algoritmo de la TRF, les calcula su
 * transformada de Fourier y envía los resultados al Proceso
 * Integrador.
 */
void divideYConquistaSolucion3(int numElem,int tam,
                               int log2tam,int rangoIntegrador){
    Complejo *renComplejo;
    double *ren;
    MPI_Status status;
    int i;
    //Reserva memoria para recibir los datos que le envía el
    //Proceso Divisor.
    ren=(double *)reservaMemoria(numElem*sizeof(double));
    //Se reserva memoria para enviar los resultados al
    //Proceso Integrador.
    renComplejo=(Complejo *)reservaMemoria(
                                        tam*sizeof(Complejo));
    while(1){
        //Recibe el vector de datos al que deberá
        //calcularle su TRF.
        MPI_Recv(ren,numElem,MPI_DOUBLE,
                        MPI_ANY_SOURCE,
                        MPI_ANY_TAG,MPI_COMM_WORLD,&status);
        //Se obtiene el número de elementos recibidos.
        MPI_Get_count(&status,MPI_DOUBLE,&numElem);
```

```
            //Si no se recibieron datos, el proceso termina su
            //trabajo.
            if(numElem){
                  //El vector recibido debe expresarse como un
                  //vector de números complejos para poder
                  //llamar a la función tRF.
                  for(i=0;i<numElem;i++)
                        renComplejo[i].real=ren[i],
                                          renComplejo[i].imag=0.;
                  //Se agregan ceros para cumplir con la
                  //restricción de la TRF que necesita que el
                  //número de muestras de la función debe ser
                  //potencia de dos.
                  for(;i<tam;i++)
                        renComplejo[i].real=
                                          renComplejo[i].imag=0.;
                  //Cálculo de la TRF del vector 'renComplejo'.
                  tRF(renComplejo,log2tam);
                  //Envío de resultados al Proceso Integrador.
                  MPI_Send(renComplejo,tam<<1,
                                    MPI_DOUBLE,rangoIntegrador,
                                    status.MPI_TAG,MPI_COMM_WORLD);
            }else{
                  //Se libera el buffer de recepción.
                  free(ren);
                  //Se libera el buffer de envío.
                  free(renComplejo);
                  return;
            }
      }
}

/* Esta función, es ejecutada por un Proceso Solución, en
 * conjunción con otras dos funciones (una divisora y otra
 * integradora), permite utilizar la técnica Divide y
 * Conquista.
 * Recibe los vectores que son enviados por el Proceso
 * Divisor, les calcula su transformada de Fourier y envía
 * los resultados al Proceso Integrador.
 */
void divideYConquistaSolucion4(int tam,int log2tam,
                                          int rangoIntegrador){
      MPI_Status status;
      Complejo *ren;
      int num;
      //Reserva memoria para la comunicación.
      ren=(Complejo *)reservaMemoria(tam*sizeof(Complejo));
```

```
     while(1){
          //Recibe el vector de datos al que deberá
          //calcularle su TRF.
          MPI_Recv(ren,tam<<1,MPI_DOUBLE,
                         MPI_ANY_SOURCE,
                         MPI_ANY_TAG,MPI_COMM_WORLD,&status);
          //Se obtiene el número de elementos recibidos.
          MPI_Get_count(&status,MPI_DOUBLE,&num);
          //Si no se recibieron datos, el proceso termina su
          //trabajo.
          if(num){
               //Cálculo de la TRF del vector 'ren'.
               tRF(ren,log2tam);
               //Envío de resultados al Proceso Integrador.
               MPI_Send(ren,num,
                              MPI_DOUBLE,rangoIntegrador,
                              status.MPI_TAG,MPI_COMM_WORLD);
          }else{
               //Liberamos el buffer de envío y recepción.
               free(ren);
               return;
          }
     }
}

/* Esta función, es ejecutada por un Proceso Solución, en
 * conjunción con otras dos funciones (una divisora y otra
 * integradora), permite utilizar la técnica Divide y
 * Conquista.
 * Recibe los vectores que son enviados por el Proceso
 * Divisor, les calcula su transformada de Fourier con base
 * en ésta obtiene otros dos vectores: uno que es un vector
 * de reales que corresponde al resultado de realizar la
 * convolución del filtro con la imagen y otro cuyos
 * elementos son trasformados a su equivalente entero al
 * redondear los elementos del primer vector. Este último
 * vector no es exactamente el resultado de la convolución
 * pero es el que permite que la imagen tenga el mismo
 * formato que la imagen de entrada, es decir, un formato
 * crudo. Estos dos vectores son enviados al Proceso
 * Integrador.
 */
void divideYConquistaSolucion5(int tamRec,int tamEnv,
                               int log2tamEnv,
                               int MN,int rangoIntegrador){
     Complejo *renComplejo;
     unsigned char *ren;
```

```
    double *renDoble;
    MPI_Status status;
    int i,num;
    //Reserva memoria para recibir los datos que le envía el
    //Proceso Divisor.
    renComplejo=(Complejo *)reservaMemoria(
                                    tamRec*sizeof(Complejo));
    //Se reserva memoria para enviar los resultados al
    //Proceso Integrador.
    renDoble=(double *)reservaMemoria(
                                      tamEnv*sizeof(double));
    ren=(unsigned char *)reservaMemoria(tamEnv);
    while(1){
        //Recibe el vector de datos al que deberá
        //calcularle su TRF.
        MPI_Recv(renComplejo,tamRec<<1,
                    MPI_DOUBLE,MPI_ANY_SOURCE,
                    MPI_ANY_TAG,MPI_COMM_WORLD,&status);
        //Se obtiene el número de elementos recibidos.
        MPI_Get_count(&status,MPI_DOUBLE,&num);
        //Si no se recibieron datos, el proceso termina su
        //trabajo.
        if(num){
            //Cálculo de la TRF del vector 'renComplejo'.
            tRF(renComplejo,log2tamEnv);
            //Obtención de los vectores que junto a los
            //otros conformarán el resultado de la
            //convolución y la imagen de salida.
            for(i=0;i<tamEnv;i++)
                ren[i]=redondeo(
                    renDoble[i]=renComplejo[i].real/MN);
            //Envío de resultados al Proceso Integrador.
            MPI_Send(ren,tamEnv,MPI_UNSIGNED_CHAR,
                        rangoIntegrador,
                        status.MPI_TAG,MPI_COMM_WORLD);
            MPI_Send(renDoble,tamEnv,MPI_DOUBLE,
                        rangoIntegrador,
                        status.MPI_TAG,MPI_COMM_WORLD);
        }else{
            //Liberamos los buffers de comunicación.
            free(renComplejo);
            free(renDoble);
            free(ren);
            return;
        }
    }
}
```

```
/* Esta función, es ejecutada por un Proceso Integrador, en
 * conjunción con otras dos funciones (una divisora y otra
 * solución), permite utilizar la técnica Divide y Conquista.
 * Recibe los vectores que son enviados por los Procesos
 * Solución, envía un mensaje al Proceso Divisor para
 * indicarle que el proceso que acaba de enviarle datos está
 * libre, y acomoda los vectores de datos de tal forma que el
 * resultado sea la matriz transpuesta de la matriz
 * distribuida por el Proceso Divisor involucrado en la
 * aplicación de esta técnica.
 */
FILE *divideYConquistaIntegrador1(int numRen,int tam,
                                  int rangoDivisor,int numProcSol){
     MPI_Status status;
     Complejo *ren;
     int i,des=numRen-1;
     //Se utiliza un archivo para almacenar resultados
     //parciales.
     FILE *archTemp=tmpfile();
     //Reserva memoria para comunicación.
     ren=(Complejo *)reservaMemoria(tam*sizeof(Complejo));
     while(numRen){
          //Recibe un vector que es enviado por uno de los
          //Procesos Solución.
          MPI_Recv(ren,tam<<1,
                        MPI_DOUBLE,MPI_ANY_SOURCE,
                        MPI_ANY_TAG,MPI_COMM_WORLD,&status);
          //Indicamos al Proceso Divisor qué proceso se
          //encuentra libre.
          if(numProcSol<numRen)
               MPI_Send(&status.MPI_SOURCE,1,
                        MPI_INT,rangoDivisor,0,MPI_COMM_WORLD);
          //El vector de datos se almacena en el archivo en
          //el orden pertinente para la obtención de la
          //mencionada matriz transpuesta.
          fseek(archTemp,status.MPI_TAG*sizeof(Complejo),
                                                  SEEK_SET);
          fwrite(ren,sizeof(Complejo),1,archTemp);
          for(i=1;i<tam;i++){
               fseek(archTemp,des*sizeof(Complejo),SEEK_CUR);
               fwrite(ren+i,sizeof(Complejo),1,archTemp);
          }
          numRen--;
     }
     //Liberamos la memoria de comunicación.
     free(ren);
     return archTemp;
```

```
}

/* Esta función, es ejecutada por un Proceso Integrador, en
 * conjunción con otras dos funciones (una divisora y otra
 * solución), permite utilizar la técnica Divide y Conquista.
 * Recibe los vectores que son enviados por los Procesos
 * Solución, envía un mensaje al Proceso Divisor para
 * indicarle que el proceso que acaba de mandarle datos está
 * libre, y acomoda los vectores de datos en los archivos de
 * tal forma que el resultado corresponda a las matrices
 * transpuestas de las matrices distribuidas por el Proceso
 * Divisor involucrado en la aplicación de esta técnica.
 */
void divideYConquistaIntegrador2(char *nomArch1,
                    char *nomArch2,int numRen,
                    int tam,int rangoDivisor,int numProcSol){
     MPI_Status status;
     double *renDoble;
     unsigned char *ren;
     int i,des=numRen-1;
     //Se utilizan dos archivos para almacenar los
     //resultados.
     FILE *arch1=abre(nomArch1,"w"),
                                    *arch2=abre(nomArch2,"w");
     //Reserva memoria para comunicación.
     ren=(unsigned char *)reservaMemoria(tam);
     renDoble=(double *)reservaMemoria(tam*sizeof(double));
     while(numRen){
          //Recibe los vectores que son enviados por uno de
          //los Procesos Solución.
          MPI_Recv(ren,tam,
                         MPI_UNSIGNED_CHAR,MPI_ANY_SOURCE,
                         MPI_ANY_TAG,MPI_COMM_WORLD,&status);
          MPI_Recv(renDoble,tam,
                        MPI_DOUBLE,status.MPI_SOURCE,
                        status.MPI_TAG,MPI_COMM_WORLD,&status);
          //Indicamos al Proceso Divisor qué proceso se
          //encuentra libre.
          if(numProcSol<numRen)
               MPI_Send(&status.MPI_SOURCE,1,
                        MPI_INT,rangoDivisor,0,MPI_COMM_WORLD);
          //Los vectores de datos se almacenan en los
          //archivos en el orden pertinente para la obtención
          //de las mencionadas matrices transpuestas.
          fseek(arch1,status.MPI_TAG,SEEK_SET);
          fwrite(ren,1,1,arch1);
          fseek(arch2,status.MPI_TAG*sizeof(double),
```

```
                                                    SEEK_SET);
            fwrite(renDoble,sizeof(double),1,arch2);
            for(i=1;i<tam;i++){
                  fseek(arch1,des,SEEK_CUR);
                  fwrite(ren+i,1,1,arch1);
                  fseek(arch2,des*sizeof(double),SEEK_CUR);
                  fwrite(renDoble+i,sizeof(double),1,arch2);
            }
            numRen--;
      }
      //Liberamos la memoria de comunicación.
      free(ren);
      free(renDoble);
      //Se cierran los archivos
      cierra(arch1,nomArch1);
      cierra(arch2,nomArch2);
}

/* La siguiente función es utilizada en la técnica Maestro-
 * Esclavo y es ejecutada por el proceso Maestro. El
 * resultado de la aplicación de esta técnica es la obtención
 * del complejo conjugado de la multiplicación puntual entre
 * las matrices contenidas en los archivos 'arch1' y 'arch2'.
 * El Maestro se encarga de distribuir, renglón por renglón,
 * las dos matrices entre los Esclavos. Una vez que recibe
 * resultados por parte de un Esclavo vuelve a enviarle datos
 * hasta concluir con el trabajo.
 */
FILE *maestroEsclavoMaestro(FILE *arch1,FILE *arch2,
                            Resolucion *res,int tamanioGrupo){
      Complejo *renRec,*renEnv;
      int i=1,renRecibidos=0;
      MPI_Status status;
      //Archivo para almacenamiento de resultados.
      FILE *conjMulPun=tmpfile();
      //Reserva memoria para la recepción de datos.
      renRec=(Complejo *)reservaMemoria(
                                    res->N*sizeof(Complejo));
      //Se reserva memoria para el envío de datos.
      renEnv=(Complejo *)reservaMemoria(
                                    res->N*sizeof(Complejo));
      //Colocamos los apuntadores al principio de los
      //archivos.
      rewind(arch1);
      rewind(arch2);
      //Envía un renglón de ambas matrices a cada uno de los
      //Procesos Esclavos.
```

```
while(i<tamanioGrupo&&i<=res->M){
     fread(renEnv,sizeof(Complejo),res->N,arch1);
     MPI_Send(renEnv,res->N<<1,
                        MPI_DOUBLE,i,i-1,MPI_COMM_WORLD);
     fread(renEnv,sizeof(Complejo),res->N,arch2);
     MPI_Send(renEnv,res->N<<1,
                        MPI_DOUBLE,i,i-1,MPI_COMM_WORLD);
     i++;
}
i--;
//Recibe resultados y envía nuevos datos hasta finalizar
//el trabajo.
while(i<res->M){
     //Recibe resultados de un Proceso Esclavo.
     MPI_Recv(renRec,res->N<<1,MPI_DOUBLE,
                    MPI_ANY_SOURCE,
                    MPI_ANY_TAG,MPI_COMM_WORLD,&status);
     renRecibidos++;
     //Envía nuevos datos al Proceso Esclavo que le
     //acaba de mandar resultados para que siga
     //trabajando.
     fread(renEnv,sizeof(Complejo),res->N,arch1);
     MPI_Send(renEnv,res->N<<1,MPI_DOUBLE,
                    status.MPI_SOURCE,i,MPI_COMM_WORLD);
     fread(renEnv,sizeof(Complejo),res->N,arch2);
     MPI_Send(renEnv,res->N<<1,MPI_DOUBLE,
                   status.MPI_SOURCE,i++,MPI_COMM_WORLD);
     //Se van almacenando los resultados.
     fseek(conjMulPun,
                 status.MPI_TAG*res->N*sizeof(Complejo),
                                                SEEK_SET);
     fwrite(renRec,sizeof(Complejo),res->N,conjMulPun);
}
//Recibe los resultados faltantes.
while(renRecibidos<res->M){
     MPI_Recv(renRec,res->N<<1,MPI_DOUBLE,
                    MPI_ANY_SOURCE,
                    MPI_ANY_TAG,MPI_COMM_WORLD,&status);
     renRecibidos++;
     fseek(conjMulPun,
                 status.MPI_TAG*res->N*sizeof(Complejo),
                                                SEEK_SET);
     fwrite(renRec,sizeof(Complejo),res->N,conjMulPun);
}
//Envía una señal de paro a cada uno de los Procesos
//Esclavos.
for(i=1;i<tamanioGrupo;i++)
```

```
            MPI_Send(NULL,0,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
        //Se libera el buffer de recepción.
        free(renRec);
        //Se libera el buffer de envío.
        free(renEnv);
        return conjMulPun;
}

/* La siguiente función es utilizada en la técnica Maestro-
 * Esclavo y es ejecutada por los Procesos Esclavos.
 * Los Procesos Esclavos, mientras no reciban la señal de
 * paro, toman los dos vectores enviados por el Maestro,
 * calculan el complejo conjugado de su multiplicación punto
 * a punto y envían de nuevo el resultado al Maestro.
 */
void maestroEsclavoEsclavo(int tam,int rangoMaestro){
        Complejo *ren1,*ren2;
        MPI_Status status;
        int i,num;
        //Se reserva memoria para recibir los datos del Maestro.
        ren1=(Complejo *)reservaMemoria(tam*sizeof(Complejo));
        ren2=(Complejo *)reservaMemoria(tam*sizeof(Complejo));
        while(1){
            //Se recibe el primer vector.
            MPI_Recv(ren1,tam<<1,MPI_DOUBLE,
                            MPI_ANY_SOURCE,
                            MPI_ANY_TAG,MPI_COMM_WORLD,&status);
            //Se obtiene el número de elementos recibidos.
            MPI_Get_count(&status,MPI_DOUBLE,&num);
            //Si no se recibieron datos, el proceso termina su
            //trabajo.
            if(num){
                //Se recibe el segundo vector.
                MPI_Recv(ren2,tam<<1,MPI_DOUBLE,
                        MPI_ANY_SOURCE,
                        status.MPI_TAG,MPI_COMM_WORLD,&status);
                //Calculamos el complejo conjugado de la
                //multiplicación puntual de los vectores
                //recibidos.
                for(i=0;i<tam;i++){
                    ren1[i]=prodComplejo(ren1[i],ren2[i]);
                    ren1[i].imag=-ren1[i].imag;
                }
                //Envío de resultados.
                MPI_Send(ren1,tam<<1,
                                MPI_DOUBLE,rangoMaestro,
                                status.MPI_TAG,MPI_COMM_WORLD);
```

```
          }else{
               //Liberamos la memoria de comunicación.
               free(ren1);
               free(ren2);
               return;
          }
     }
}

/* tRF calcula la trasformada de Fourier del arreglo 'F' cuyo
 * número de componentes es 2^log2N, mediante la aplicación
 * del algoritmo de la TRF.
 */
void tRF(Complejo *F,int log2N){
     Complejo w,u,temp;
     int N=1<<log2N,NE2=N>>1,NM1=N-1,i,j,k,n,m,ip;
     //Reordenamos el arreglo de entrada con la regla de
     //inversión de bits.
     for(i=1,j=NE2;i<NM1;i++,j+=k){
          if(i<j){
               temp=F[j];
               F[j]=F[i];
               F[i]=temp;
          }
          k=NE2;
          while(k<=j){
               j-=k;
               k>>=1;
          }
     }
     //Ya con el arreglo reordenado, calculamos la
     //trasformada de Fourier.
     for(k=1;k<=log2N;k++){
          n=1<<k;
          m=n>>1;
          w.real=1.;
          w.imag=0.;
          u.real=cos(PI/m);
          u.imag=-sin(PI/m);
          for(j=0;j<m;j++){
               for(i=j;i<N;i+=n){
                    temp=prodComplejo(F[ip=i+m],w);
                    F[ip].real=F[i].real-temp.real;
                    F[ip].imag=F[i].imag-temp.imag;
                    F[i].real+=temp.real;
                    F[i].imag+=temp.imag;
               }
```

```
                w=prodComplejo(w,u);
            }
        }
}

/* prodComplejo obtiene el producto de multiplicar a los
 * números complejos 'a' y 'b'.
 */
Complejo prodComplejo(Complejo a,Complejo b){
     return (Complejo){a.real*b.real-a.imag*b.imag,
                                  a.real*b.imag+a.imag*b.real};
}

/* redondeo obtiene la representación a entero, en el rango
 * [0-255], del número real 'valor'. Lo anterior es
 * conseguido al redondear 'valor' y realizar un posible
 * truncamiento del resultado si éste no está dentro del
 * anterior rango.
 */
unsigned char redondeo(double valor){
     return ((int)(10*valor)%10<5)?valor:valor+1;
}

/* Esta función abre un archivo y verifica que no existan
 * errores al abrirlo.
 */
FILE *abre(char *nombre,char *acceso){
     FILE *archivo;
     if((archivo=fopen(nombre,acceso))==NULL){
          printf("Ocurrió un error
                              al abrir el archivo: %s\n",nombre);
          MPI_Abort(MPI_COMM_WORLD,1);
     }
     return archivo;
}

/* Esta función cierra un archivo y verifica que no existan
 * errores al cerrarlo.
 */
void cierra(FILE *archivo,char *nombre){
     if(fclose(archivo)==EOF){
          printf("Ocurrió un error al
                                cerrar el archivo: %s\n",nombre);
          MPI_Abort(MPI_COMM_WORLD,1);
     }
}
```

```
/* Esta función reserva 'bytes' bytes de memoria y verifica
 * si la asignación es exitosa.
 */
void  *reservaMemoria(size_t bytes){
     void *memoria;
     if((memoria=(void *)malloc(bytes))==NULL){
          printf("Insuficiente espacio en memoria\n");
          MPI_Abort(MPI_COMM_WORLD,1);
     }
     return memoria;
}
```

5.4.3.3 Comentarios adicionales.

La entrada de datos del programa está conformada por dos archivos. Uno de ellos es la imagen a la que se le desea aplicar el proceso de filtrado, ésta debe estar almacenada en formato crudo y cada píxel que la constituye debe estar representado por 1 byte. El otro archivo es el que corresponde al filtro y debe estar constituido por elementos de doble precisión en formato binario.

Todos los procesos comienzan a ejecutar la función principal, el proceso con rango igual a cero verifica que el número de argumentos en la línea de órdenes sea el correcto. Los procesos del grupo determinan distintos valores involucrados en el cálculo de la convolución en 2-D: **`resImagen`**, **`resFiltro`**, **`resConv`** y **`resDeseada`**.

Las variables **`resImagen`** y **`resFiltro`** almacenan la resolución o tamaño de las matrices que representan a la imagen y al filtro, respectivamente. La variable **`resConv`** almacena la resolución que tendrá la matriz resultado de efectuar la convolución entre la imagen y el filtro, más claramente, para una imagen de dimensiones AxB y un filtro de dimensiones CxD, asumiendo que ambos arreglos son periódicos con periodos M en la dirección "x" y N en la dirección "y", tales que M = A+C-1 y N = B+D-1. La matriz resultante de tal convolución discreta bidimensional tendrá resolución MxN. Como podemos notar de la expresión (5.4-3), los valores de los periodos de la imagen y el filtro están regidos por una relación de desigualdad, para el diseño de este programa, decidimos optar por la igualdad ya que ésta brinda las dimensiones más pequeñas que pueden tener las matrices para que el resultado sea exactamente un periodo completo de la convolución, sin traslapes ni cálculos extra. Por otro lado, **`exp`** almacena los valores más pequeños de **r**, **s** y **`resDeseada`** los más pequeños de **R**, **S** que satisfacen las expresiones:

$$M \leq 2^r = R, \tag{5.4-21}$$

$$N \leq 2^s = S. \tag{5.4-22}$$

Otro punto importante que salta a la vista y justifica las expresiones (5.4-21) y (5.4-22) es el hecho de tener que modificar las matrices de entrada agregándoles los ceros necesarios para poder aplicar el algoritmo de la TRF en el cálculo de la convolución en 2-D. En otras palabras, podemos considerar a `resDeseada` como la resolución mínima que deben tener las matrices de las que se desea obtener la convolución para que sea posible el cálculo de ésta a través de la TRF, por lo tanto, ambas matrices deben ser rellenadas con ceros según éstos se requieran hasta que su resolución sea igual a `resDeseada`.

Aclaramos también que, en esta implementación paralela, el relleno antes mencionado no lo efectuamos antes de comenzar con el proceso de la convolución como podría pensarse, sino que, éste se va dando a la par con los cálculos con la intención de minimizar las operaciones y disminuir el tiempo de ejecución. Los Procesos Solución, en las distintas veces que se aplica la técnica Divide y Conquista, son los que van sucesivamente completando con ceros las matrices en donde éstas lo requieren.

Finalizamos los comentarios del código de este programa con la tabla 5.10, en ella se muestra una visión general de esta aplicación paralela: las técnicas que intervienen, los resultados obtenidos después la utilización de cada una de éstas, así como la forma en la que decidimos realizar la asignación de papeles dentro de cada paradigma.

Técnica utilizada	Índice†	Pasos realizados‡	Tipo de proceso	Función que ejecuta	Rango del proceso
Divide y Conquista	1	(1.1) y (1.2)	Divisor	divideYConquistaDivisor1	0
			Solución	divideYConquistaSolución1	1, ..., n-2
			Integrador	divideYConquistaIntegrador1	n-1
	2	(1.3) y (1.4)	Divisor	divideYConquistaDivisor2	n-1
			Solución	divideYConquistaSolución2	1, ..., n-2
			Integrador	divideYConquistaIntegrador1	0
	3	(2.1) y (2.2)	Divisor	divideYConquistaDivisor3	0
			Solución	divideYConquistaSolución3	1, ..., n-2
			Integrador	divideYConquistaIntegrador1	n-1
	4	(2.3) y (2.4)	Divisor	divideYConquistaDivisor2	n-1
			Solución	divideYConquistaSolución2	1, ..., n-2
			Integrador	divideYConquistaIntegrador1	0
Maestro-Esclavo	1	(3) y (4.1)	Maestro	maestroEsclavoMaestro	0
			Esclavo	maestroEsclavoEsclavo	1, ..., n-1
Divide y Conquista	5	(4.2) y (4.3)	Divisor	divideYConquistaDivisor2	0
			Solución	divideYConquistaSolución4	1, ..., n-2
			Integrador	divideYConquistaIntegrador1	n-1
	6		Divisor	divideYConquistaDivisor2	n-1
			Solución	divideYConquistaSolución5	1, ..., n-2
			Integrador	divideYConquistaIntegrador2	0

Tabla 5.10. Forma de utilización de las técnicas Maestro-Esclavo y Divide y Conquista en la aplicación paralela para el cálculo de la convolución en discreta bidimensional.

† Debido a que la técnica Divide y Conquista se usa más de una vez en el programa, los números de esta columna nos sirven para indicar por precedencia a qué aplicación de la técnica nos estamos refiriendo, de esta manera podemos establecer que el valor de 1 representa la primera vez que se utilizó esta técnica en el programa y, del mismo modo, el número 6 representa a la última.

‡ Aquí nos referimos a la serie de pasos, que ya anteriormente establecimos y, que permiten llevar a cabo el cálculo de la convolución en 2-D a través de la TRF.

5.4.3.4 Ejemplos de aplicación.

Existen muy diversos filtros que pueden ser aplicados a una imagen con la intención de resaltar o mejorar una determinada característica de ésta. Por lo tanto, según el área de aplicación, varían el tamaño de la ventana y los valores de los elementos del arreglo que constituye una determinada máscara de filtrado.

Con la intención de mostrar el tipo de resultados que se pueden obtener con la aplicación paralela que realizamos para el filtrado espacial –programa conv2D.c–, mostramos algunas de las máscaras más utilizadas en el procesamiento digital de imágenes.

5.4.3.4.1 Filtros para atenuar el ruido.

La mayoría de los filtros que son utilizados para disminuir el ruido en imágenes son filtros paso-bajas pues éstos dejan el contenido de baja frecuencia inalterado mientras que atenúan las componentes de alta frecuencia, como el ruido está asociado con las altas frecuencias este tipo de filtros resulta adecuado para reducir el ruido presente en una imagen. Los filtros paso-bajas se caracterizan por poseer componentes positivos o nulos, uno de los más conocidos es el llamado *promediador*, éste se distingue por la máscara de la expresión (5.4-23), para una resolución de MxN.

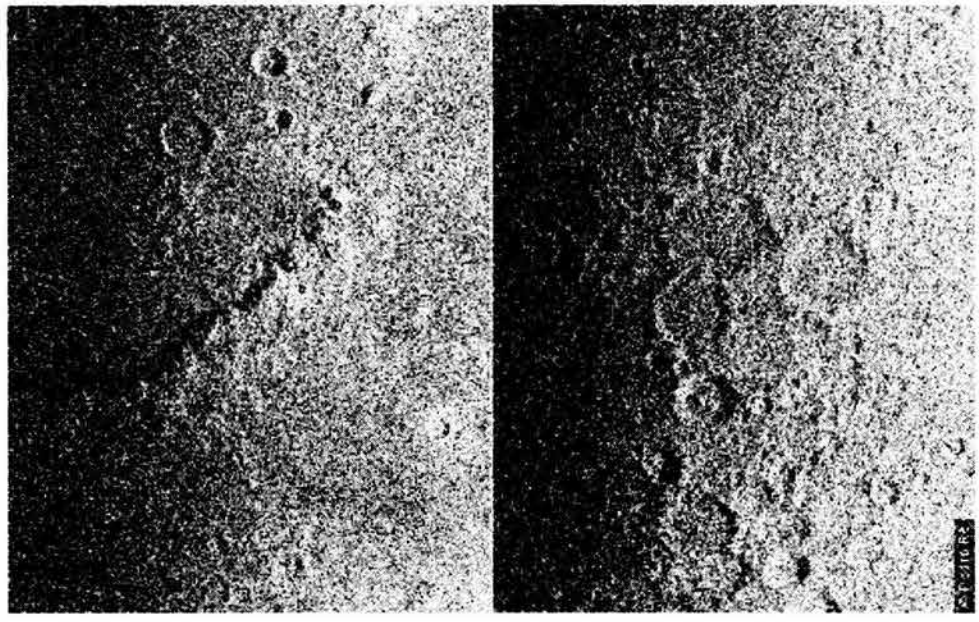
$$prom_{MxN} = \frac{1}{MN}\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}_{MXN} \qquad (5.4\text{-}23)$$

Aparte del filtro promediador, existen otras máscaras que al igual que éste permiten atenuar el ruido, algunos ejemplos son los de la figura 5.32.

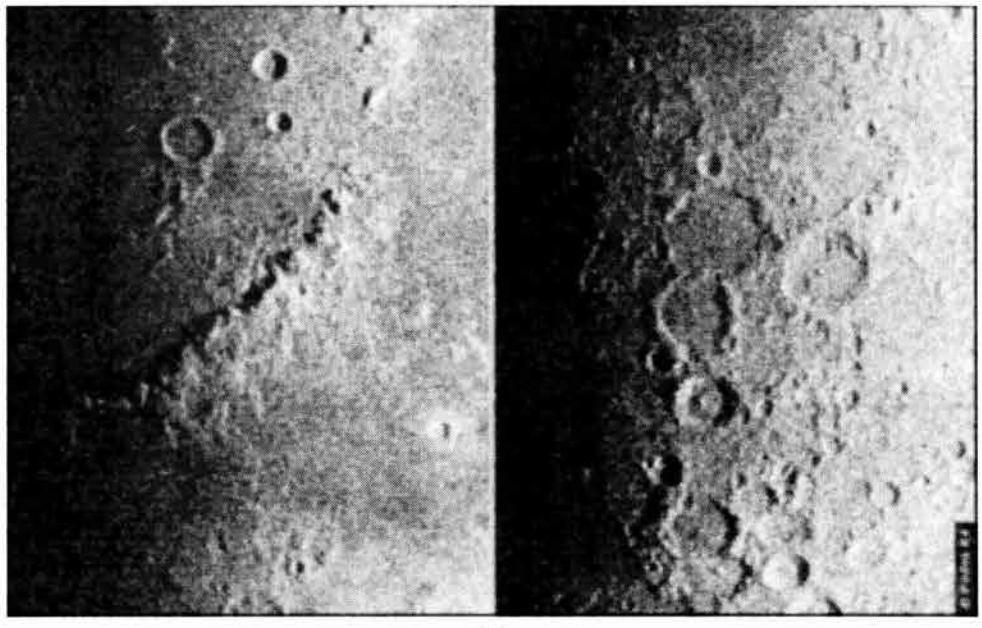
$$\frac{1}{5}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \frac{1}{6}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \frac{1}{10}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figura 5.32. Cuatro máscaras que permiten el filtrado paso-bajas de una imagen.

En las figuras 5.33 y 5.34, se muestran dos imágenes que han sido procesadas con la intención de disminuir el ruido.

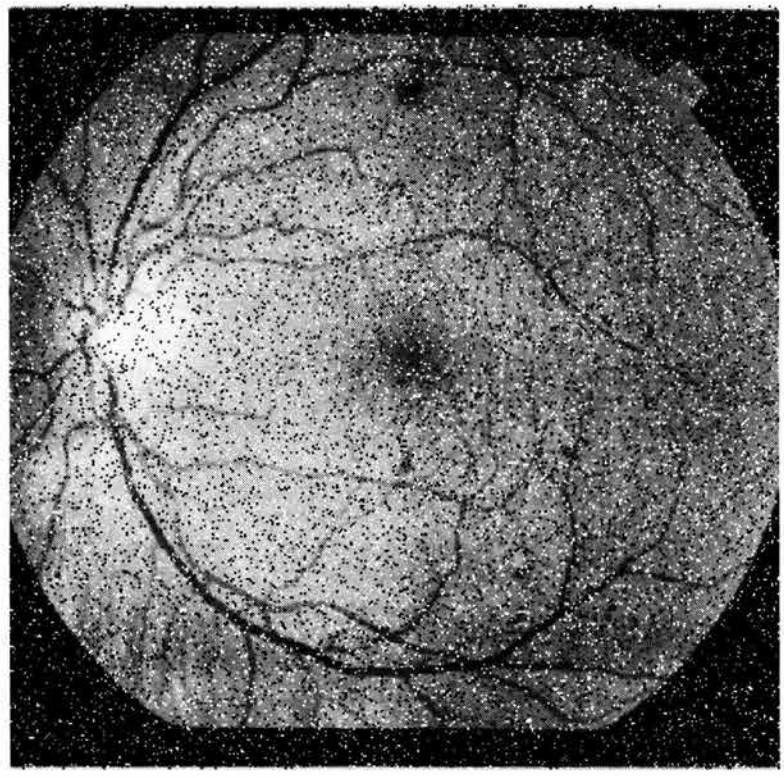


(a)

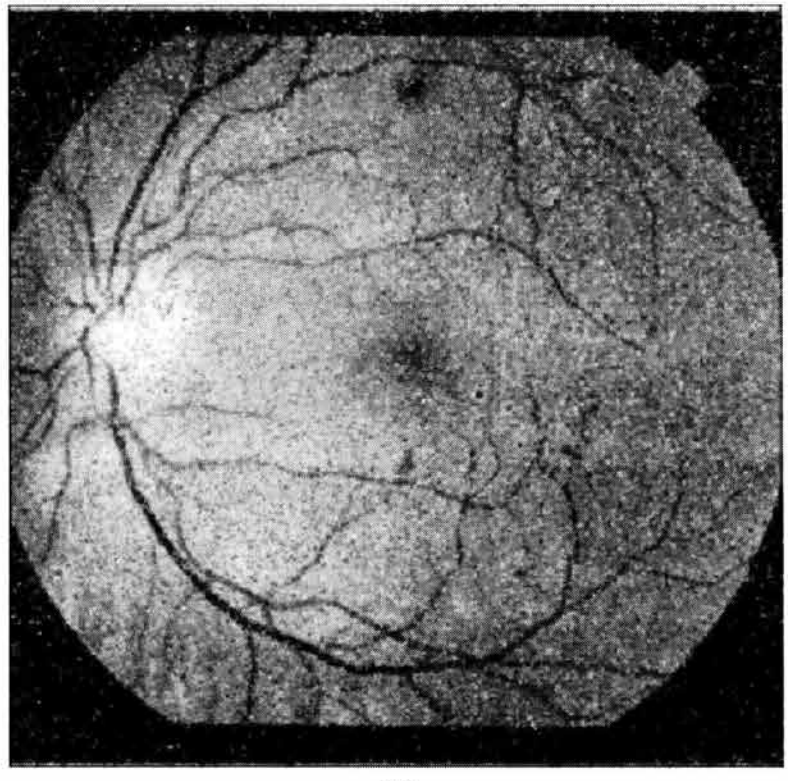


(b)

Figura 5.33. Atenuación del ruido de una imagen. (a) Imagen de la superficie lunar corrupta por ruido. (b) Resultado de procesar la imagen ruidosa con un filtro promediador 5x5.



(a)



(b)

Figura 5.34. Atenuación del ruido de una imagen. (a) Imagen ruidosa de la retina ocular. (b) Resultado de procesar (a) con la última máscara de la figura 5.32.

5.4.3.4.2 Filtros para la detección de bordes.

Los filtros para la detección de bordes se utilizan comúnmente para enfatizar detalles finos o contornos de una imagen. En este rubro se encuentran los filtros paso-altas y los filtros derivativos.

Los filtros paso-altas –como su nombre lo indica– tienen la propiedad de acentuar los detalles de alta frecuencia de una imagen y atenuar las componentes de frecuencia baja. Normalmente, las porciones de una imagen que presentan componentes de alta frecuencia serán resaltadas mediante la utilización de niveles de gris más claros, mientras que aquéllas con componentes de baja frecuencia serán más oscuras. Uno de los efectos indeseados de estos filtros es que pueden acentuar el ruido de la imagen si es que ésta lo posee. En la figura 5.35 se muestran tres máscaras de convolución representativas de un filtro paso-altas.

$$\frac{1}{9}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \frac{1}{6}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \frac{1}{10}\begin{bmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

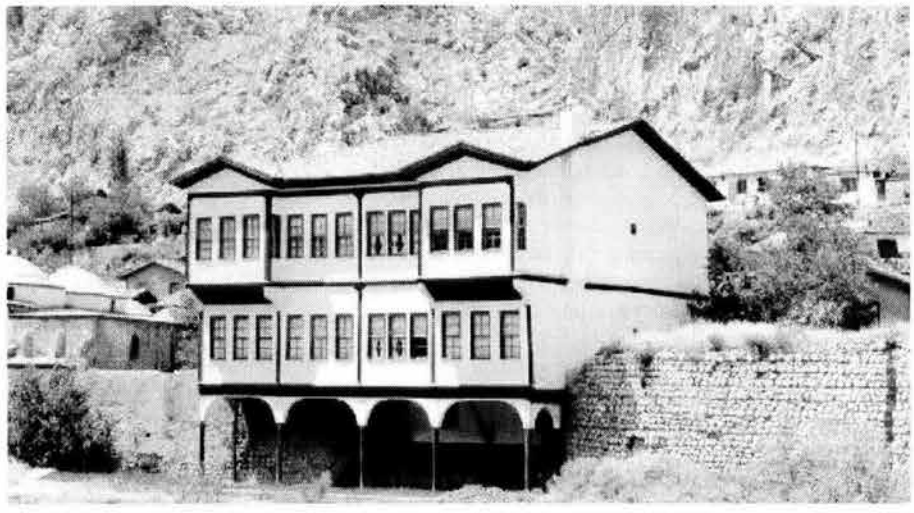
Figura 5.35. Algunas máscaras características de filtros paso-altas.

En la figura 5.36 se muestra el resultado de procesar una imagen con un filtro paso-altas.

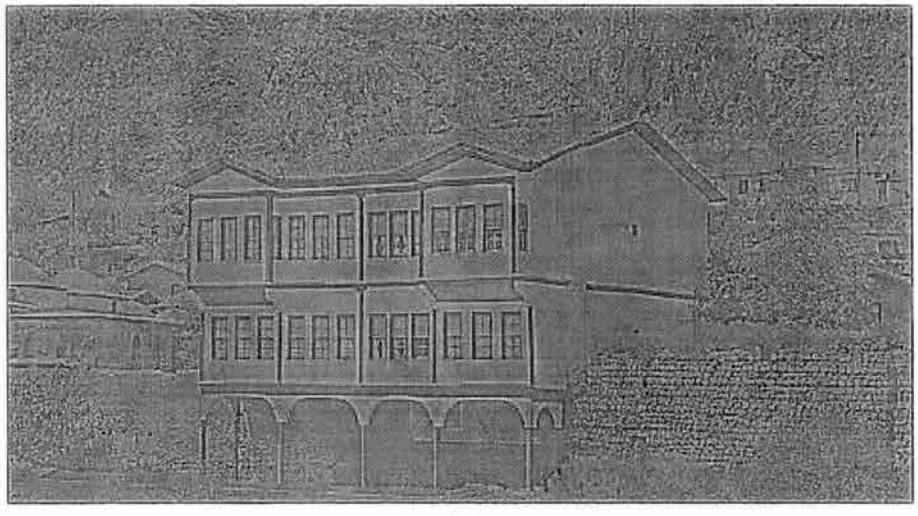
Por otro lado, los llamados filtros derivativos utilizan una determinada aproximación a la magnitud del gradiente de la señal con el objeto de enfatizar los bordes de una imagen. Una discusión acerca de la obtención de las máscaras de este tipo de filtros sale del contexto de esta tesis. A continuación mostramos cómo actúan sobre una imagen este tipo de filtros.

Operador de Sobel.

El operador de Sobel obtiene una aproximación de la magnitud del gradiente con las máscaras de la figura 5.37, en la figura 5.38 podemos ver una imagen procesada con este operador.



(a)



(b)

Figura 5.36. Detección de bordes de una imagen por medio de un filtro paso-altas. (a) Imagen original de una casa. (b) Resultado obtenido al procesar la imagen con el primer filtro paso-altas aparece en la figura 5.35.

$$\frac{1}{8}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Figura 5.37. Máscaras del operador de Sobel. La máscara de la izquierda detecta los bordes en dirección horizontal y la de la derecha lo hace en dirección vertical.

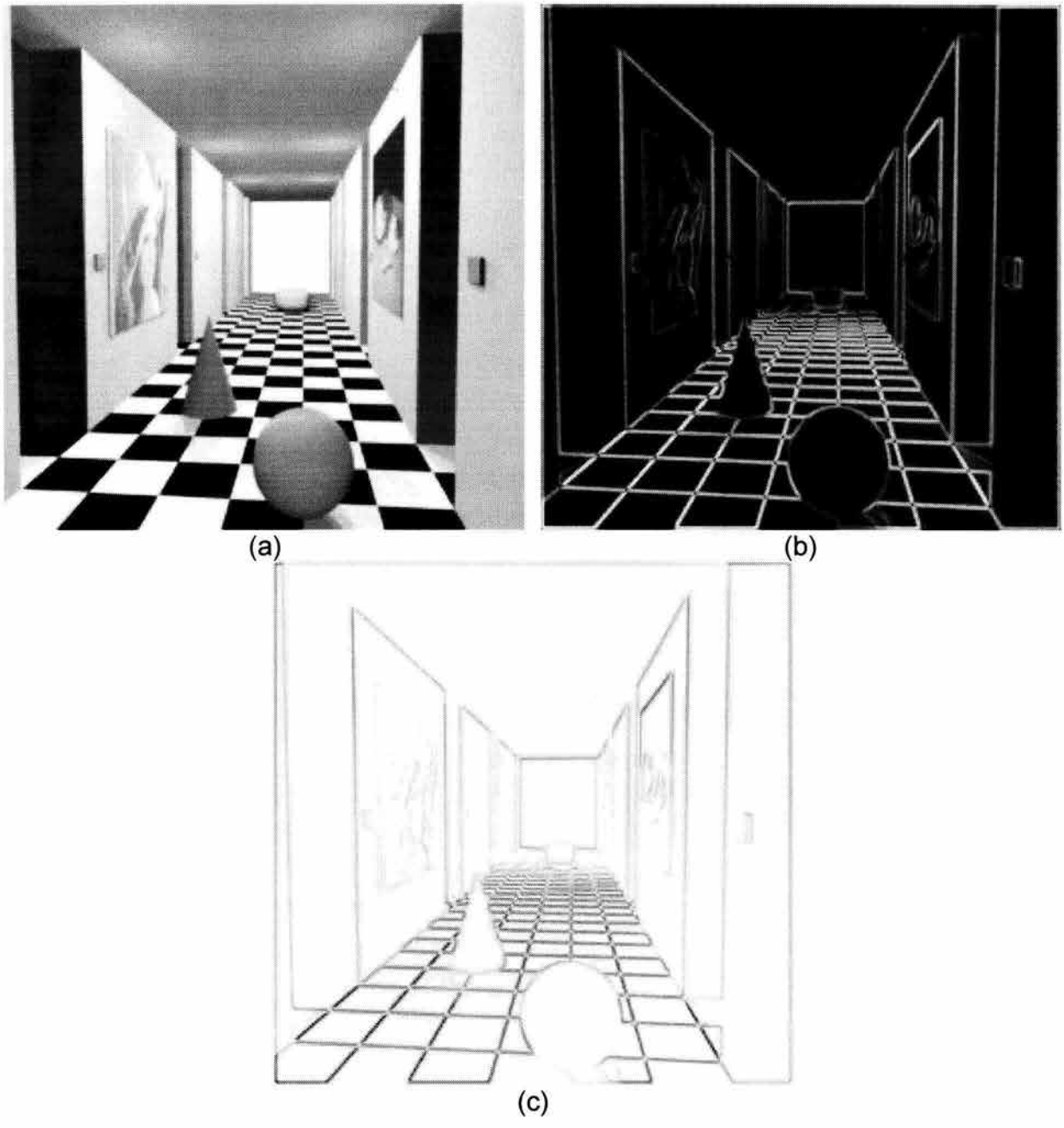


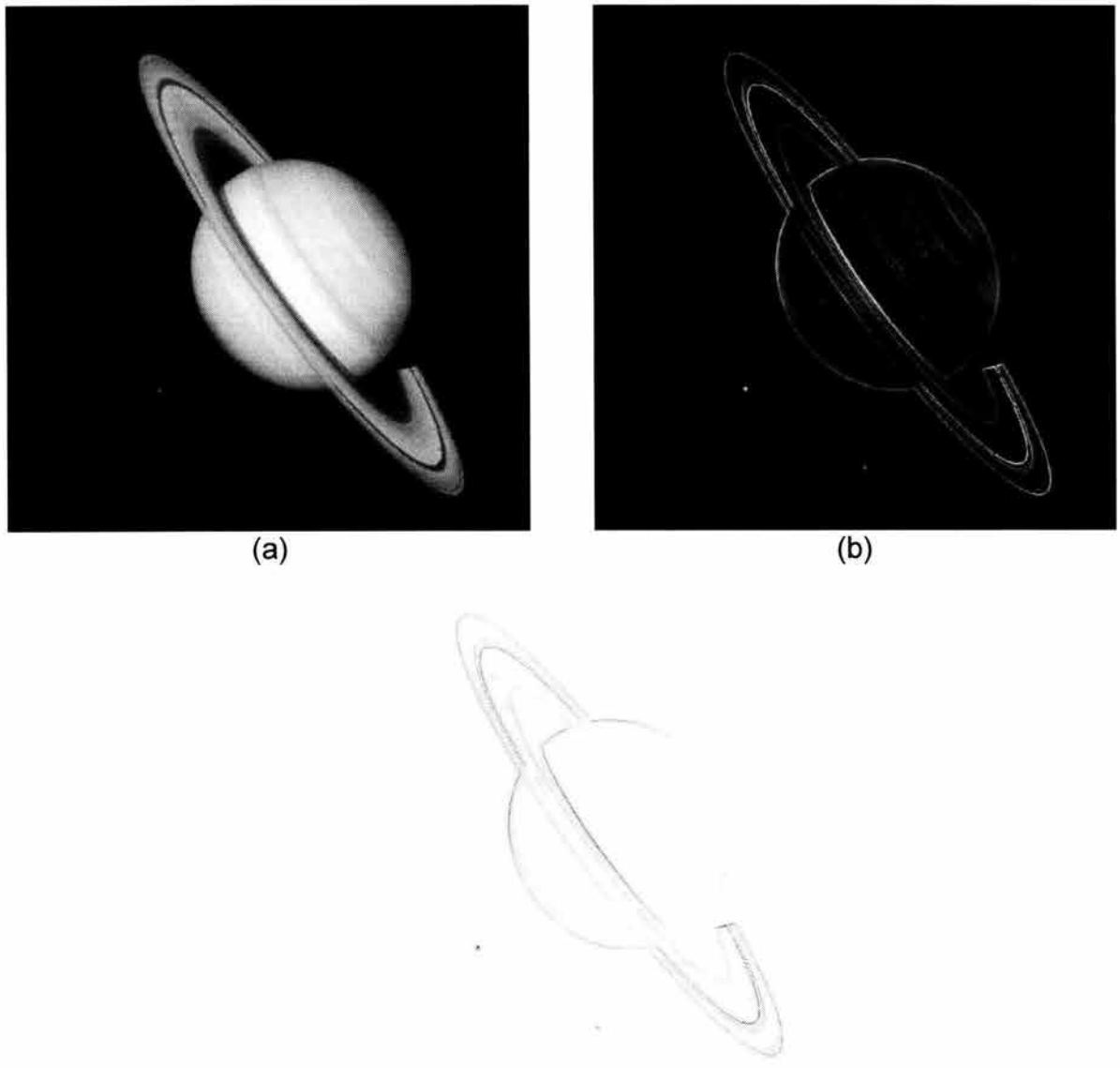
Figura 5.38. Detección de los bordes de una imagen utilizando el operador de Sobel. (a) Imagen de una escena virtual. (b) Resultado obtenido al procesar la imagen con el operador de Sobel. (c) Imagen negativo de la imagen procesada con Sobel.

Operador de Roberts.

Esta clase de operador aproxima la magnitud del gradiente mediante las máscaras de la figura 5.39, en la figura 5.40 podemos ver el resultado de su aplicación en una imagen.

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} \qquad \frac{1}{\sqrt{2}}\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix}$$

Figura 5.39. Máscaras correspondientes al operador de Roberts. La máscara de la izquierda resalta los bordes a 45^0 y la máscara de la derecha realza los bordes a -45^0.

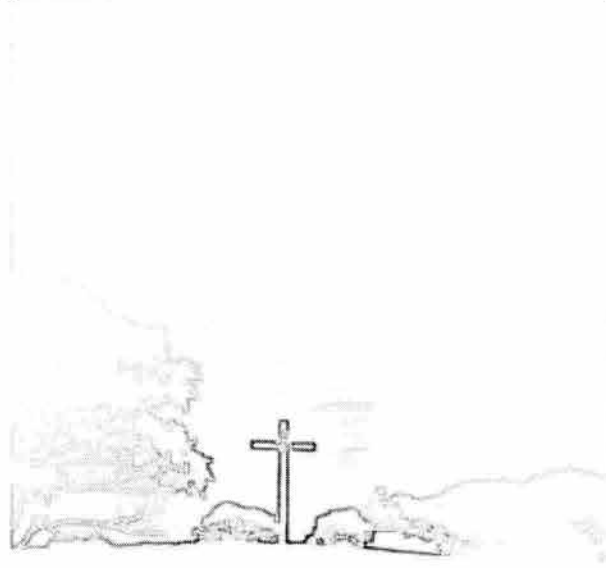


(a) (b)

(c)

Figura 5.40. Detección de los bordes de una imagen utilizando el operador de Roberts. (a) Imagen del planeta Saturno. (b) Resultado que se obtiene al filtrar la imagen con el operador de Roberts. (c) Imagen negativo de la imagen procesada con Roberts.

(a) (b)



(c)

Figura 5.42. Detección de los bordes de una imagen utilizando el operador de Prewitt. (a) Imagen de crepúsculo en Kwajalein, Micronesia. (b) Resultado obtenido al procesar (a) con el operador de Prewitt. (c) Imagen negativo de (b).

Operador de Prewitt.

Las operaciones de Prewitt obtienen una aproximación a la magnitud del gradiente con las máscaras de la figura 5.41, en la figura 5.42 podemos ver una imagen filtrada con este operador.

$$\frac{1}{6}\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \frac{1}{6}\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Figura 5.41. Máscaras del operador de Prewitt. La primera máscara detecta los bordes en dirección horizontal y la segunda lo hace en dirección vertical.

5.4.3.5 Evaluación del desempeño.

A continuación, evaluaremos el desempeño de esta aplicación sobre el cluster de la Facultad de Ingeniería. De manera general, el desempeño de una aplicación paralela dependerá en gran medida de la capacidad computacional del sistema donde se ejecute, entre mayor sea ésta, mejor será el desempeño de la aplicación.

En este análisis tomaremos en cuenta dos factores que influyen en el comportamiento de cualquier aplicación paralela: la carga de trabajo y el número procesos que participan en el cálculo. Para ello, mediremos los tiempos de ejecución de la aplicación al ir aumentando el número de procesos para tres distintas cargas de trabajo: una pequeña, una mediana y una grande.

Las cargas de trabajo con las que probamos el programa se especifican en la tabla 5.11. En cuanto a la variación en el número de procesos, hicimos pruebas con 3, 4, ..., y 14 procesos. El menor número de procesos que utilizamos fue 3, pues es el mínimo número que puede utilizar esta aplicación híbrida. El máximo, 14 procesos, corresponde al número de nodos que componen el cluster.

Tipo de carga	Tamaño de la imagen [píxeles]	Tamaño del filtro [elementos]
Carga 1	3600	9
Carga 2	14400	64
Carga 3	250000	144

Tabla 5.11. Cargas de trabajo utilizadas para las pruebas de rendimiento de la aplicación paralela de filtrado espacial.

Para las mediciones de tiempos utilizamos la función que para tal efecto provee la MPI, **MPI_Wtime**. Es importante destacar que, las secciones de código que permiten medir el tiempo de ejecución de un programa deben ser colocadas entre el código del algoritmo de modo que no afecten su comportamiento de alguna forma y verificando que en realidad se esté midiendo bien el tiempo total de ejecución, en otras palabras, lo idóneo es que un solo proceso realice la medición, éste deberá ser el último en finalizar. Es conveniente realizar un análisis del programa para determinar qué proceso es el que finaliza al último, si se tienen problemas para determinarlo con claridad, se pueden utilizar otras funciones como **MPI_Barrier**, para forzarlo.

Para cada condición distinta –de carga de trabajo y de número de procesos– se realizaron 3 corridas del programa y se obtuvo el tiempo promedio de ejecución con el objeto de obtener un valor más representativo.

Debido a que el servidor del cluster es mucho más potente que cualquiera de sus nodos y, por lo tanto, las contribuciones de los nodos sólo pueden ser bien apreciadas si no participa en el cálculo, en las corridas de esta aplicación ningún proceso fue creado en el servidor, es decir que, se le pasó la opción `nolocal` al comando **mpirun**.

Otro punto importante que se debe tomar en cuenta cuando se trabaja con un cluster heterogéneo es la forma en que se asignarán los procesos en las distintas máquinas que lo conforman, pues esta asignación repercute directamente en el desempeño de la aplicación paralela. Para el caso particular de este programa, los procesos que realizan funciones de distribución de trabajo y recolección de resultados son los que deberán ejecutarse en los mejores equipos del cluster. Por eso fue que empleamos la opción `machinefile` del comando **mpirun**. También debemos considerar que, al utilizar la opción `nolocal`, el arranque de procesos remotos en los nodos ya no será realizado por el servidor y por lo tanto debe seleccionarse a uno, de entre los mejores equipos, para que realice esta labor; la selección de un equipo lento para este fin, produce un incremento en el tiempo de ejecución.

5.4.3.5.1 Resultados.

La tabla 5.12 muestra los tiempos de ejecución obtenidos al correr la aplicación paralela para los distintos tamaños de carga y número de procesos.

En la figura 5.43, mostramos la gráfica de los tiempos de ejecución obtenidos al ir aumentando el número de procesos con una carga pequeña de trabajo. Podemos observar que, aquí si se tuvieron mejores resultados al ejecutar el programa con un mayor número de procesos a pesar de que la carga de trabajo es pequeña. Que se lleguen a obtener ventajas al usar más procesos con una carga pequeña indica que es mucho el procesamiento que se realiza con los datos de entrada, esto también se visualiza si comparamos los tiempos de ejecución de

esta aplicación con los tiempos obtenidos por la de procesamiento puntual, que es la demanda menor procesamiento.

	Tiempos de ejecución para:		
Número de procesos	**Carga 1 [s]**	**Carga 2 [s]**	**Carga 3 [s]**
3	36.133504	161.180489	657.384034
4	38.718495	146.444621	632.520983
5	35.819008	147.898936	601.529097
6	39.599038	146.910525	594.294071
7	35.939276	161.277646	595.802023
8	39.299574	161.416540	478.728507
9	35.858173	145.246246	450.113392
10	30.399185	125.657761	366.181978
11	33.038642	134.153810	382.739107
12	30.357761	124.262860	314.369090
13	27.964960	112.773464	293.350740
14	25.348456	92.953291	277.519819

Tabla 5.12. Resultados obtenidos con las pruebas realizadas a la aplicación paralela de filtrado espacial.

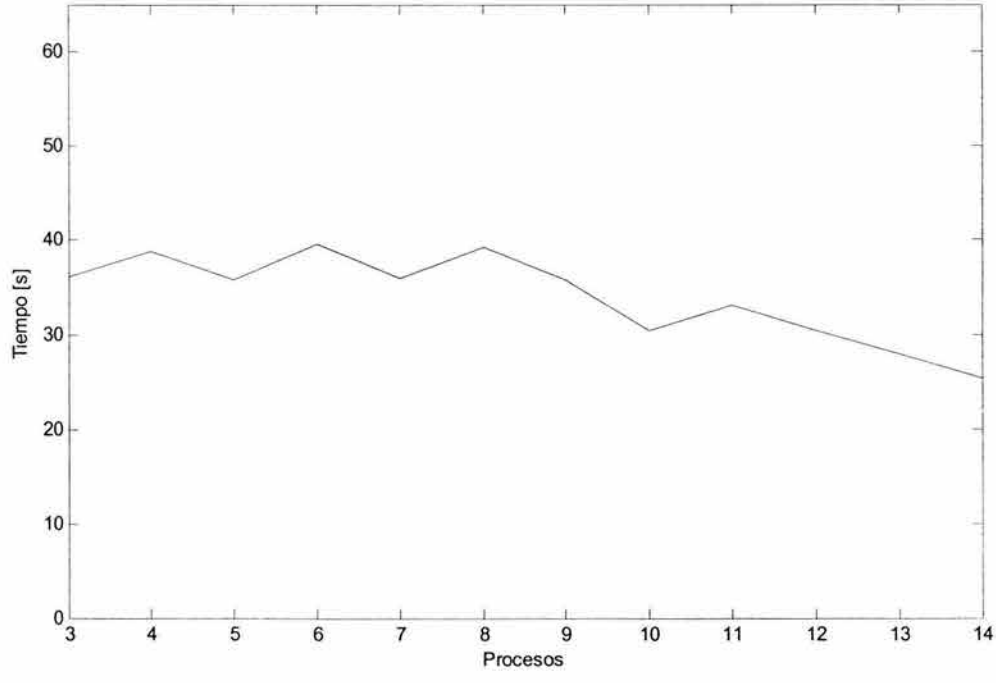


Figura 5.43. Gráfica de tiempo de ejecución contra número de procesos, para una carga del tipo "carga 1".

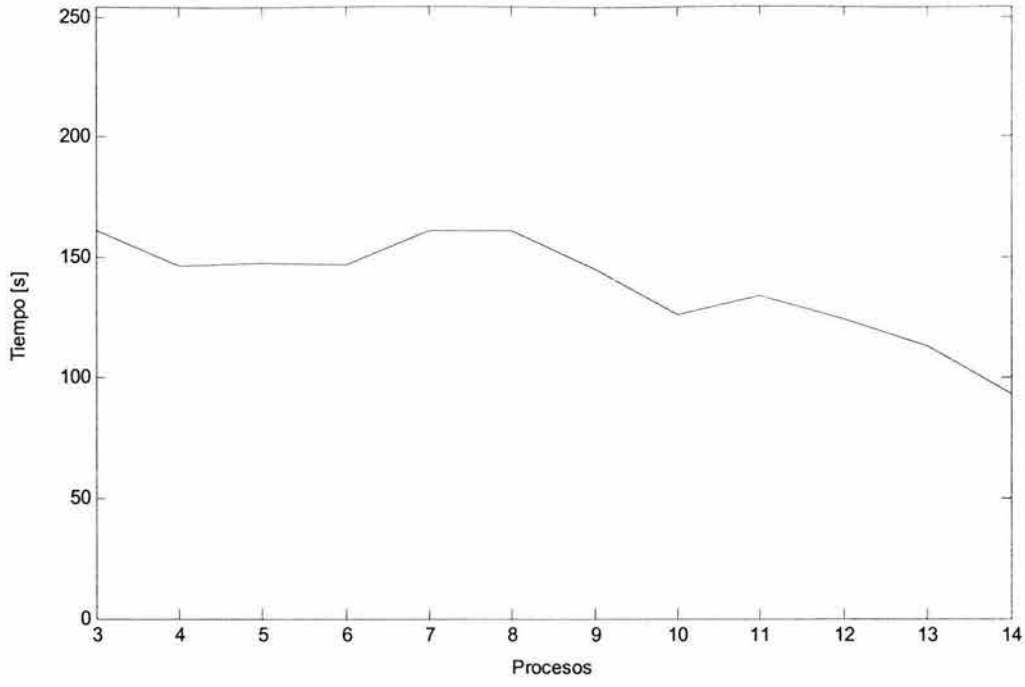


Figura 5.44. Gráfica de tiempo de ejecución contra número de procesos, para una carga del tipo "carga 2".

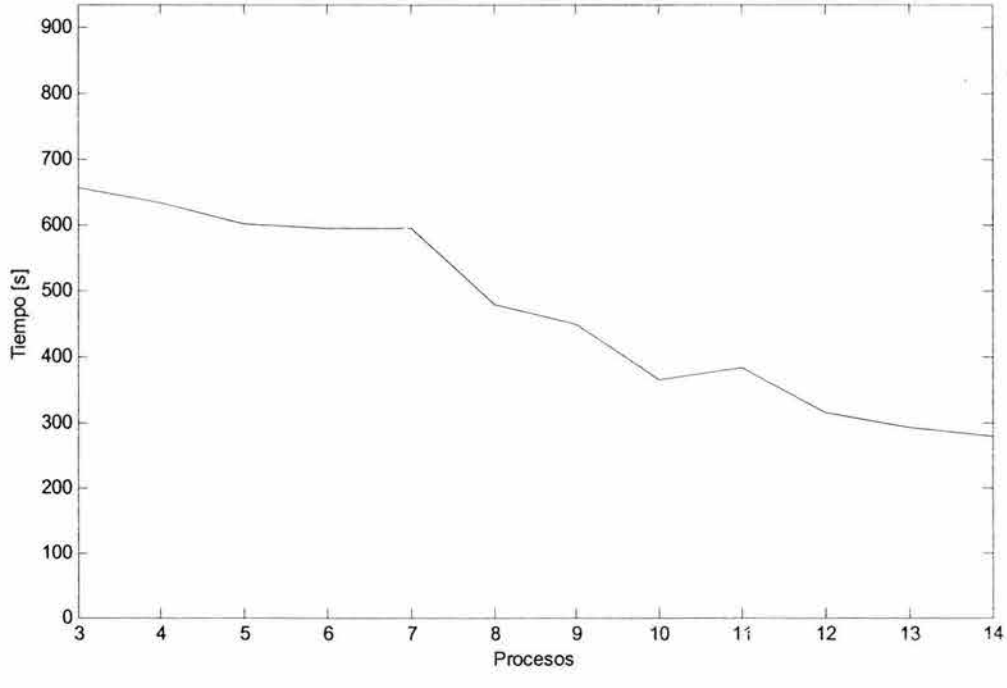


Figura 5.45. Gráfica de tiempo de ejecución contra número de procesos, para una carga del tipo "carga 3".

El comportamiento que tiene el programa para una carga de trabajo moderada se muestra en la figura 5.44. Se aprecia que, conforme se incrementa el número de procesos, se reduce el tiempo de ejecución. Para este tamaño de carga se obtienen más ventajas que para la pequeña.

Finalmente, en la figura 5.45, vemos la gráfica del desempeño de la aplicación ante una carga considerada como grande. De nuevo se ve reducido el tiempo de ejecución conforme se suman más procesos a la resolución del problema. La complejidad de este algoritmo es grande, se requiere de mucha comunicación para lograr que los procesos colaboren y se coordinen, sin importar esto, se obtivieron buenas ventajas al volverlo paralelo porque la cantidad de cálculos involucrada en la resolución del problema es muy grande y la distribución del trabajo es realizada conforme éste se demanda.

Conclusiones.

El trabajo de investigación que hemos realizado en esta tesis nos ha provisto de una visión general acerca del cómputo paralelo; pues el proceso de creación de una aplicación paralela contempla una serie de consideraciones que involucran a los desarrolladores en diversos aspectos: el conocimiento de las características de la arquitectura en la que se implantará, el dominio de un ambiente de desarrollo de aplicaciones acorde con la arquitectura y el empleo de los paradigmas básicos de programación paralela que existen sobre estos sistemas.

Con esta experiencia de aprendizaje, nos percatamos realmente de la complejidad que puede presentar la conversión de un algoritmo secuencial en un algoritmo paralelo. Es evidente que es mucho más sencillo desarrollar aplicaciones en un sistema monoprocesador que en un ambiente paralelo; porque, en este último, hasta una pequeña aplicación requiere de un análisis profundo y de conocimientos que van más allá de la utilización de un lenguaje de programación. El hecho de que la complejidad de crear un programa paralelo sea mayor a la de crear el mismo programa para un ambiente secuencial se debe a que, en un sistema paralelo no centralizado, el usuario es el responsable de dividir el trabajo, especificar la manera en que deberán comunicarse y coordinarse los procesos e indicar cómo deberán unificarse los resultados parciales para encontrar la solución total del problema que se desea resolver.

También vemos que sin importar que exista una metodología general para la programación paralela, en la implementación de una algoritmo paralelo sobre un sistema específico se debe tomar en consideración la forma en que éste está constituido, es decir, cuál es el hardware que lo conforma y el esquema de comunicación que utiliza. En el caso específico de un cluster, se deben conocer las características de cada uno de los nodos, las especificaciones y el desempeño que tiene la red de interconexión y las limitantes del propio sistema; lo anterior con el objeto de obtener los mayores beneficios de las aplicaciones creadas para correr en el mismo. En otras palabras, los paradigmas y las técnicas que emplea la programación paralela son generales, pueden ser aplicados para diseñar una aplicación que funcione adecuadamente en cualquier arquitectura paralela. Por otro lado, no se puede programar una aplicación para un determinado ambiente paralelo si se desconocen sus características y sus principios de funcionamiento.

Con respecto a las técnicas de programación paralela, se hizo evidente la versatilidad y la generalidad que éstas ostentan. Constituyen la base que permite realizar la planificación del desarrollo de aplicaciones paralelas. Además, en la mayoría de los casos, existe más de una técnica con la que puede un algoritmo convertirse en paralelo. Si obtenemos varias versiones de un mismo algoritmo utilizando distintas técnicas para volverlo paralelo, todas atacarán el problema de distinta forma pero al final obtendrán el mismo resultado. Los programas asociados a estas versiones del algoritmo paralelo tendrán distintas

características: algunos podrán ser más rápidos o eficientes bajo algunas condiciones, otros serán más fáciles de programar o implementar, otros realizarán una mejor distribución del trabajo, otros manejarán mejor las comunicaciones entre procesos, etc. Por esta razón, un desarrollador deberá contemplar cuáles son las ventajas y desventajas que tendrá el algoritmo al volverlo paralelo con una determinada técnica, a fin de poder establecer qué paradigma deberá utilizarse para implementarlo.

Otro punto que debemos destacar es que las técnicas de programación paralela que aquí presentamos no son únicas, existen muchas variaciones y adaptaciones de éstas, e incluso otras nuevas. Sin embargo, si se dominan las técnicas básicas y se comprende la forma en que operan y distribuyen el trabajo, es posible combinarlas o modificarlas de manera que se adecuen más al problema que se está atacando y faciliten su implementación. De este modo, podemos crear variantes de una misma técnica con características especiales que son más acordes con la aplicación y con el sistema donde se desea implantarla.

Una característica importante de las técnicas que presentamos es que no son excluyentes, es posible combinarlas para crear aplicaciones híbridas. El algoritmo que desarrollamos para el filtrado espacial es un ejemplo de aplicación híbrida, está formado por una mezcla de las técnicas SPMD y Maestro-Esclavo. Por lo general, cuando se desarrollan aplicaciones de gran tamaño, surge la necesidad de utilizar varias técnicas debido a que no existe una que pueda, por sí sola, dar solución al problema. La creación de una aplicación híbrida requiere de mayor experiencia para conseguir que todas las técnicas empleadas en su realización, ya sean básicas o variaciones de éstas, colaboren entre sí y se coordinen adecuadamente cuando trabajan en conjunto. Además, la utilización de más técnicas también permite realizar una programación de tipo modular.

De acuerdo a los resultados obtenidos con las pruebas realizadas sobre las aplicaciones que desarrollamos, existen varias cuestiones que debemos puntualizar:

- En un cluster heterogéneo, la correcta asignación de los procesos en los distintos equipos que lo componen puede suscitar una mejora importante en el rendimiento general del algoritmo.

- En general, si la aplicación demanda gran cantidad de procesamiento, agregar más nodos a la solución del problema decrementará el tiempo de ejecución.

- Por el contrario, si una aplicación no requiere de gran cantidad de procesamiento y se suman más nodos al cálculo, esto empeorará su desempeño. En estos casos, es necesario encontrar el número máximo de procesos para el que se obtienen las mejores ventajas.

Las pruebas realizadas evidencian los beneficios que pueden conseguirse con una aplicación paralela, es de esperarse que si se utiliza un Beowulf de mejores características en cuanto a la capacidad de los nodos y la velocidad de la red, se obtenga un mejor rendimiento y mayores ventajas con el empleo de más nodos en el cálculo.

Creemos que el cluster tipo Beowulf en el que realizamos las pruebas, si bien tiene muchas limitantes debidas esencialmente a la falta de presupuesto y por estar formado por un equipo en desuso, esto no impidió que pudiéramos estudiar el comportamiento de diversos algoritmos paralelos sobre esta plataforma distribuida. Lo principal fue que logramos ver cómo varios equipos independientes pueden trabajar en unidad para emular un solo sistema de mayor poder computacional.

Por otra parte, quedó de manifiesto que el sistema operativo Linux representa una plataforma estable que, si se maneja adecuadamente, puede brindarnos las herramientas necesarias y el ambiente de trabajo idóneo para el desarrollo de aplicaciones paralelas robustas. De igual manera, la MPI mostró ser una interfaz confiable, eficiente y poderosa; la gran variedad de funciones que posee pone a nuestra disposición un conjunto de herramientas que permiten elaborar toda clase de aplicaciones paralelas. Al parecer, éstas y otras razones han convertido actualmente al software libre en la principal elección para el cómputo de alto rendimiento.

Durante el desarrollo de esta tesis, una de las dificultades que más nos costó superar fue la de comprender el paralelismo. No resulta tarea fácil la familiarización con el paralelismo, sobre todo porque estamos acostumbrados a pensar de forma secuencial congénitamente. No es sino después de adentrarse en los conceptos relacionados con el paralelismo y de la constante práctica con esta clase de sistemas que uno comienza a entenderlo. Es en este momento cuando comienzan a relucir cuestiones como la sincronización, la concurrencia y la aleatoriedad que describen los procesos que se ejecutan de forma simultánea; los anteriores, son factores que no están presentes en los algoritmos secuenciales y que comúnmente pasamos por alto cuando comenzamos a desarrollar nuestros primeros programas paralelos.

Finalmente, no está de más comentar que de nada sirve conocer todas las funciones que puede proveer una plataforma como la MPI, si se carece de una metodología para el desarrollo de aplicaciones, se desconocen los aspectos básicos relacionados con el paralelismo y sobre todo si no se ha adquirido aún la habilidad más importante:

PENSAR EN PARALELO.

Apéndice A

Especificaciones del ambiente de trabajo.

A.1 Introducción.

En este apéndice incluimos algunos aspectos, relacionados con el ambiente de trabajo del cluster de la Facultad de Ingeniería, que es necesario conocer para poder llevar a cabo el desarrollo de aplicaciones paralelas utilizando la MPI, en su distribución MPICH.

A diferencia de un computador ordinario que se enciende y es posible comenzar a trabajar con él, el cluster necesita de un proceso previo de inicialización para poder ser utilizado. Por esta razón, indicaremos los procedimientos básicos que permiten trabajar e interactuar con el cluster.

A.2 Creación de aplicaciones.

El servidor del cluster –scluster– permite administrarlo por completo, a pesar de que está conformado por varios sistemas independientes –los nodos–, desde el servidor es posible tener el control sobre el cluster en su totalidad. Es por ello que, el primer paso es iniciar una sesión de usuario en el servidor. Posteriormente, se puede utilizar cualquier editor de texto para crear una aplicación paralela utilizando las funciones que provee la MPI estándar.

Para compilar nuestro código fuente, la forma más sencilla de hacerlo es utilizar el comando **mpicc** que provee MPICH. Podemos proporcionar la ruta completa del mismo o incluirla en nuestro *path*, añadiendo en el archivo $HOME/.bash_profile la línea:

```
export PATH=$PATH:/usr/share/mpich-1.2.5/bin
```

Es necesario resaltar que, para compilar un programa con base en la MPI, sólo se requiere del servidor, en otras palabras, los nodos no intervienen en este proceso y por lo tanto pueden permanecer apagados hasta que se necesite de su intervención.

A.3 Ejecución de aplicaciones.

Una vez que compilamos nuestra aplicación paralela, lo común es que deseemos ejecutarla en el cluster. El ambiente distribuido por MPICH provee un guión de comandos para facilitar la ejecución de programas en arquitecturas paralelas. Dicho guión es el ya mencionado **mpirun**, situado en el directorio de

binarios de MPICH –ya incluido en el *path*–. También necesitaremos agregar a nuestro *path* la ubicación de las bibliotecas de la MPI:

/usr/share/mpich-1.2.5/include
/usr/share/mpich-1.2.5/lib

Otro punto importante es que, antes de ejecutar cualquier programa, debemos especificar de alguna forma los nombres de las máquinas sobre las que se desea correr la aplicación. Esto puede realizarse de diversas formas. Una es mediante el paso del parámetro `machinefile` al comando **mpirun**, tal y como lo vimos en la sección 3.8. Otra manera de hacerlo, la más común, es por medio de la edición del archivo /usr/share/mpich-1.2.5/share/machines.xxxx, que deberá contener los nombres de las máquinas de la arquitectura xxxx –en nuestro caso, la arquitectura es LINUX–. De esta forma, cada vez que **mpirun** es ejecutado, el número requerido de nodos será seleccionado de este archivo. Así por ejemplo, para correr todos los procesos de la MPI sobre una misma máquina, sólo es necesario hacer que ésta aparezca en todas las líneas del mencionado archivo. Un ejemplo de este archivo se muestra a continuación.

```
# Change this file to contain the machines that you want to use
# to run MPI jobs on.  The format is one host name per line, with either
#    hostname
# or
#    hostname:n
# where n is the number of processors in an SMP.  The hostname should
# be the same as the result from the command "hostname"
scluster
nodo10
nodo11
nodo12
nodo13
nodo14
nodo15
nodo16
nodo17
nodo18
nodo19
nodo20
nodo21
nodo22
nodo23
```

Los nombres de los nodos deben ser provistos en el mismo formato que los proporciona el comando **hostname**.

En este momento, MPICH ya cuenta con la información necesaria para poder ejecutar nuestra aplicación; pero antes, requerimos inicializar los nodos con del cluster con los que deseamos trabajar –no es necesario que todos los nodos estén encendidos, sólo hay que inicializar aquéllos con los que se desea trabajar–.

El proceso de inicialización de los nodos se debe a que éstos no poseen disco duro y, por lo tanto, requieren que la imagen del sistema operativo con el que trabajarán sea cargada por medio de la red. Para tal efecto requerimos de un disco de arranque, que debemos crear con la ayuda del paquete Etherboot ejecutando:

```
$ /tftpboot/etherboot-5.0.6/src/make bin32/tipo_de_tarjeta.fd0
```

esta acción grabará en un disco flexible la información que se necesita para iniciar la tarjeta de red, con esto, el nodo podrá solicitar su dirección IP al servidor DHCP y con ello se le enviará lo necesario para poder trabajar. En nuestro caso, el tipo de tarjeta de red utilizada es 3c509.

Con el disco de arranque creado, sólo es cuestión de insertarlo en la unidad de discos de 3.5" y esperar a que en nodo sea iniciado propiamente por medio de la red.

Después que los nodos están listos para trabajar, es posible comenzar a ejecutar programas con el comando **mpirun**. No está por demás aclarar que, por razones de seguridad del sistema, los programas no pueden ejecutarse como superusuario.

A.4 Solución de problemas.

Enseguida, abordaremos algunas cuestiones que pueden suscitar dificultades y que es común que se presenten al momento de trabajar con este sistema distribuido.

Es posible que un nodo se inicialice incorrectamente debido a que no se esperó el suficiente tiempo como para que los distintos pasos que componen el proceso de inicio concluyeran. Si un nodo no se inició bien, obtendremos un error a la hora de intentar ejecutar cualquier proceso sobre el mismo. Es engorroso determinar si un nodo ya está listo para ser utilizado, por eso, una mejor opción es la de consultar el archivo /var/log/messages, en éste podemos ver el progreso del proceso de inicio de un nodo. Podemos, además, utilizar el comando **tail** para darle seguimiento a dicho proceso como lo muestra la siguiente línea:

```
# tail -f /var/log/messages
```

una vez que se monta el directorio **$HOME** por medio de NFS, el nodo se encuentra listo para ser utilizado. Otra forma de verificación puede ser el uso del comando **ping**, pero la primera opción es más útil.

Otro problema que puede presentarse es que, los servicios de red o el servidor DHCP no estén levantados o no se hayan iniciado correctamente. Si existen problemas como los anteriores, aunque encendamos los nodos con el disco de arranque, no se les transferirá la información que necesitan para poder trabajar.

Para trabajar con los demonios **inetd** y **dhcpd**, utilizamos los guiones de comandos /etc/rc.d/init.d/inet y /etc/rc.d/init.d/dhcpd, respectivamente. Su uso es el siguiente:

```
inet {start|stop|status|restart|reload}

dhcpd {start|stop|restart|status}
```

Para poder trabajar con el cluster, deben estar corriendo los demonios **inetd** y **dhcpd**

```
$ /etc/rc.d/init.d/dhcpd status
dhcpd (pid 1110) is running...

$ /etc/rc.d/init.d/inet status
inetd (pid 1195) is running...
```

si no están corriendo o presentan algún problema es necesario reiniciarlos.

También se pueden producir problemas en tiempo de ejecución, no sería posible nombrarlos pues dependen en gran medida, o casi en su totalidad, del código que se está ejecutando. El usuario es el que debe aprender a identificarlos y a solucionarlos. A pesar de esto, un error en tiempo de ejecución que puede ocurrir a menudo en este cluster es el siguiente:

```
p0_5943: (13.819058) p4_shmalloc returning NULL; request = 4194344 bytes
You can increase the amount of memory by setting the environment variable
P4_GLOBMEMSIZE (in bytes); the current size is 4194304
p0_5943:  p4_error: alloc_p4_msg failed: 0
```

Este error se debe a que uno o más nodos solicitan más memoria de la que tienen asignada. En la resolución de este problema con el cluster, no hay mucho que hacer –la poca memoria de los nodos impide una asignación mayor–; simplemente nos indica que nuestro programa ha excedido las capacidades del sistema y que no podremos incrementar más la carga de trabajo en los nodos.

En un cluster de mejores características, podemos solucionar el problema incrementando la cantidad de memoria asignada a los nodos; esto último se logra al establecer el valor de la variable de entorno **P4_GLOBMEMSIZE**, añadiendo en el archivo $HOME/.bashrc la línea

```
export P4_GLOBMEMSIZE=cantidad_de_memoria_en_bytes
```

donde cantidad_de_memoria_en_bytes debe ser mayor a la petición de memoria que hizo el programa.

Finalmente, es posible que requiramos trabajar de forma directa con las máquinas que componen el cluster. Su sistema de archivos está disponible en el directorio /tftpboot del servidor bajo una carpeta con el nombre del nodo. Así, por ejemplo, para la máquina nodo10 su sistema de archivos se desprenderá a partir de /tftpboot/nodo10. De esta forma podemos realizar las modificaciones deseadas sobre una máquina en particular.

Si, por algún motivo, es necesario ingresar directamente en una máquina, se puede realizar una conexión remota con un nodo en particular mediante el comando **rsh**. Una vez que establecida una sesión remota, podemos trabajar en él en la forma que habitualmente lo hacemos con Linux; para salir de la sesión utilizamos el comando **logout**.

Referencias

Textos.

1. Almasi, George S. y Gottlieb, Allan. "Highly parallel computing". The Benjamin/Cummings Publishing Company, Inc. E.U.A., 1994.

2. Bertsekas, Dimitri P. y Tsitsiklis, John N. "Parallel and distributed computation". Prentice Hall. E.U.A., 1989.

3. Boyanov, K. "Parallel and distributed processing". Elsevier Science Publishers. Holanda, 1991.

4. Brawer, Steven. "Introduction to parallel programming". Academic Press, Inc. E.U.A., 1989.

5. Cairó, Osvaldo y Guardati, Silvia. "Estructuras de datos". McGraw-Hill. México, 1993.

6. Ceballos Sierra, Francisco. "Curso de programación con C". RA-MA Editorial. México, 1990.

7. Codenotti, Bruno y Leoncini, Mauro. "Introduction to parallel processing". Addison-Wesley. Gran Bretaña, 1993.

8. Cooley, J.W.; Lewis , P. A. W. y Welch, P. D. "The fast Fourier transform and its applications". IEEE Trans. Educ. E.U.A., 1969.

9. Foster, Ian. "Designing and building parallel programs". Addison-Wesley. E.U.A., 1995.

10. Gonzalez, Rafael C. y Woods, Richard E. "Digital image processing". Addison-Wesley. E.U.A., 1992.

11. Grammatikakis, Miltos; *et al.* "Parallel system interconnections and communications". CRC Press. E.U.A., 2001.

12. Jájá, Joseph. "An introduction to parallel algorithms". Addison-Wesley. E.U.A., 1992.

13. Juárez Sosa, Julio César; Saénz García, Elba Karén y Valdez Casillas, Oscar René. Trabajo de tesis: "Análisis del desempeño de un cluster Beowulf en diversos algoritmos de tipo concurrente". Directora de tesis: Ing. Laura Sandoval Montaño. Facultad de Ingeniería. UNAM, 2001.

14. Kumar, Vipin; *et al.* "Introduction to parallel computing: desing and analysis of algorithms". The Benjamin/Cummings Publishing Company, Inc. E.U.A., 1994.

15. Manrique Martinez, Daniel. Trabajo de tesis: "Construcción y evaluación del desempeño de un cluster tipo Beowulf para cómputo de alto rendimiento". Directora de tesis: Ing. Laura Sandoval Montaño. Facultad de Ingeniería. UNAM, 2001.

16. Message Passing Interface Forum. "MPI: A message-passing interface". Universidad de Tennessee, Knoxville, Tenn., 1995.

17. Mikloško, J.; *et al.* "Fast algorithms and their implementation on specialized parallel computers". VEDA. Checoslovaquia, 1989.

18. Modi, Jagdish J. "Parallel algorithms and matrix computation". Oxford University Press. E.U.A., 1990.

19. Pitas, Ioannis. "Parallel algorithms: for digital image processing, computer vision and neural networks". Wiley: series in parallel computing. Gran Bretaña, 1993.

20. Ritchie, Dennis M. y Kernighan, Brian W. "The C programming language". Prentice Hall. E.U.A., 1988.

21. Stevens, W. Richard. "Advanced programming in the UNIX environment". Addison-Wesley. E.U.A., 1993.

22. Tanenbaum, Andrew S. "Distributed operating systems". Prentice Hall. E.U.A., 1995.

23. Xavier, C. e Iyengar, S. S. "Introduction to parallel algorithms". John Wiley & Sons, Inc. E.U.A., 1998.

Direcciones electrónicas.

24. Argonne National Laboratory. "MPICH-A Portable Implementation of MPI". Universidad de Chicago.
http://www-unix.mcs.anl.gov/mpi/mpich/

25. Bravo V., Antonio J. "Procesamiento Digital de Imágenes".
http://www.ing.ula.ve/~abravo/document/tutorial/imagenes/indice.html

26. Departamento de Supercómputo de la DGSCA. "Clusters en México". UNAM.
http://clusters.unam.mx/

27. Dietz, Hank. "Linux Parallel Processing HOWTO". Purdue University School of Electrical and Computer Engineering.
http://suparum.rz.uni-mannheim.de/Linux/parallel/pphowto.html
http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html

28. Gropp, William y Lusk, Ewing. "Installation and User's Guide to MPICH, a Portable Implementation of MPI Version 1.2.6. The ch_p4 device for Workstation Networks". Universidad de Chicago, Laboratorio Nacional de Argonne.
http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpichman-chp4.pdf

29. IEEE Computer Society. "IEEE Distributed Systems Online".
http://dsonline.computer.org/

30. Jonson, Don. "Amplitude Quantization".
http://cnx.rice.edu/content/m0051/latest/

31. Little Unix Programmers Group. "Parallel Programming - Basic Theory For The Unwary".
http://users.actcom.co.il/~choo/lupg/tutorials/parallel-programming-theory/parallel-programming-theory.html

32. Message Passing Interface Forum.
http://www.mpi-forum.org/

33. Moreno, Asunción. "Cuantificación". Universidad Politécnica de Cataluña.
http://gps-tsc.upc.es/veu/personal/asuncion/curso/Tema5.ppt

34. Schaller, Nan C. "Nan's Parallel Computing Page". Rochester Institute of Technology, Computer Science Department.
http://www.cs.rit.edu/~ncs/parallel.html

35. SP Parallel Programming Workshop. "Parallel programming introduction".
http://www.mhpcc.edu/training/workshop/parallel_intro/MAIN.html

36. Sterling, Thomas y Savarese, Daniel. "A Coming of Age for Beowulf-class Computing". Center for Advanced Computing Research, California Institute of Technology.
http://www.cacr.caltech.edu/Publications/techpubs/PAPERS/cacr175/cacr175.html

37. Texas Advanced Computing Center.
http://www.tacc.utexas.edu/

38. The Beowulf project (CESDIS).
http://www.beowulf.org.

39. Young, I.T.; Gerbrands, J.J. y van Vliet, L.J. "Image Processing Fundamentals". http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Contents.html