UNIVERSIDAD NACIONAL
AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

INDUCCIÓN AL CÓMPUTO DE ALTO DESEMPEÑO

TESIS PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE INGENIERO EN COMPUTACIÓN

PRESENTAN

RAMOS PALACIOS GUADALUPE SUSANA
ROSALES MADRIGAL SERVANDO

DIRECTORA DE TESIS

ING. LAURA SANDOVAL MONTAÑO

CIUDAD UNIVERSITARIA. MÉXICO, D.F., 2006

Agradecimientos:

A todas aquellas personas que nos han brindado el apoyo a lo largo de nuestra carrera, especialmente a nuestros padres, que con amor y paciencia, nos han ayudado a superar momentos difíciles permitiéndonos llegar a esta importante meta que se ve alcanzada con la realización de esta tesis, de todo corazón MIL GRACIAS.

Gracias también a todos aquellos profesores que no sólo nos han transmitido sus conocimientos, sino que además con sus grandes palabras de sabiduría y experiencia, nos han formado el orgulloso sentimiento de formar parte de esta gran Universidad lo cual nos ha tatuado profundamente el escudo de la UNAM en nuestros corazones y que permanecerá ahí por siempre.

A todos, GRACIAS! ☺.

Atte: Susana y Servando

ÍNDICE

INTRODUCCIÓN

A lo largo de la historia se ha podido apreciar el inevitable desarrollo de la tecnología que en un comienzo fue presentándose poco a poco, aportando gran parte de las herramientas para adquirir conocimiento y transformarlo en beneficio o prejuicio de la humanidad, pero no es sino hasta estos últimos 20 años, en que este rápido avance en la tecnología, ha generado la tendencia a buscar en cualquier área, algo más rápido y más pequeño a lo que se tiene; un ejemplo claro es la computación, que en muy poco tiempo puede generar muchísimos cambios en cuanto al hardware y software se refiere, y debido a este crecimiento tan acelerado, es posible tener nuevas tecnologías en corto tiempo, desarrollando computadoras cada vez más rápidas y más pequeñas. Es por esta razón que no sólo su capacidad de hardware se incrementa, sino que proporcionalmente a esto, lo hacen sus exigencias computacionales y aplicaciones que puede ésta realizar, y que con mayor razón, se espere un alto desempeño por parte de estos equipos.

La versatilidad que tienen las computadoras 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. En algunos casos esta exigencia puede verse resuelta con potentes equipos o supercomputadoras, pero ésta no es la solución pues el costo de estos equipos es elevado y a veces estos sistemas tan avanzados pueden llegar a saturarse por la gran demanda de trabajo que exigen las nuevas aplicaciones.

Las actuales tendencias en cómputo apuntan hacia la operación 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.

Es por esta razón que los sistemas paralelos/distribuidos no sólo se aplican en problemas grandes y complejos que requieren una cantidad enorme de potencia computacional, sino que debido a su economía, a su capacidad de comportarse como un único equipo, a la gran ventaja de expansión con gran facilidad y flexibilidad y con una alta capacidad de funcionar a pesar de que alguno de los equipos falle, han venido a brindar una solución factible y poderosa a los equipos centralizados de gran capacidad pero de mayor costo.

El objetivo principal de esta tesis es elaborar un compendio que explique de forma clara y suficiente el tema de cómputo de alto desempeño que permita incursionar en el ámbito de los sistemas paralelos/distribuidos a través de cuatro grandes capítulos.

Comenzaremos en esta tesis con una introducción a los sistemas de alto desempeño en donde se describirán los sistemas de cómputo paralelo, distribuidos y paralelos/distribuidos que permitirán obtener los conocimientos básicos del cómputo de alto desempeño.

En el segundo capítulo, se abordarán conceptos, métodos y técnicas en la programación paralela/distribuida que permitirá desarrollar aplicaciones en los sistemas paralelos/distribuidos. Describirá los diversos niveles de paralelismo que se pueden presentar en códigos secuenciales; los distintos paradigmas de la programación paralela que muestran la estrecha relación que existe entre las arquitecturas y el código paralelo, sus ventajas y desventajas; así como cada una de las etapas para la creación de programas paralelos que permitan un mejor desempeño de los sistemas.

En el tercer capítulo se describirán los distintos entornos de programación paralela y se profundizará en herramientas para desarrollar programas de procesamiento paralelo/distribuido. Se abordaran principalmente las bibliotecas de MPI y PVM describiendo su funcionamiento y las funciones básicas de cada una de ellas.

En el cuarto capítulo se aplicarán los lenguajes y herramientas de programación paralela descritos en el tercer capítulo para desarrollar aplicaciones. Se abordan aplicaciones de tipo numérico, manipulación de datos, cómputo basado en la web y balanceador de cargas.

Finalmente, se ha incluido en este trabajo un anexo con prácticas de laboratorio. Estas prácticas, permitirán no sólo reforzar y aplicar los conocimientos básicos de la teoría, si no que además permitirán realizar ejercicios e investigaciones que la complementarán. Las prácticas abordan temas como la construcción y configuración de un cluster tipo Beowulf, la instalación de las bibliotecas paralelas, así como el uso de los diferentes métodos y técnicas del procesamiento paralelo para la creación de programas paralelos para después hacer uso de las bibliotecas paralelas con ejemplos sencillos que permitan comprender el funcionamiento de cada una de ellas. Las últimas prácticas abordarán temas novedosos de la aplicación del cómputo de alto desempeño que reflejan la importancia de estos sistemas hoy en día.

Esperamos que este trabajo de tesis represente una buena fuente que sirva de apoyo a todas aquellas personas interesadas en los sistemas paralelos/distribuidos.

CAPÍTULO 1

INTRODUCCIÓN A LOS SISTEMAS DE ALTO DESEMPEÑO

1.1 Sistemas de cómputo paralelo

1.1.1 Paralelismo

En los últimos 40 años han ocurrido incrementos dramáticos en la velocidad de cómputo. Mucho de ello se debe al uso de componentes electrónicos inherentemente más rápidos. Así, desde los tubos de rayos catódicos, pasando por transistores y luego por la integración de pequeña, media y gran escala, hemos atestiguado el crecimiento en tamaño y rango de los problemas computacionales susceptibles de ser resueltos. Así, el supercómputo hasta finales de los años ’80 tenía como punteras a máquinas con veloces procesadores vectoriales. Un procesador vectorial es una máquina diseñada específicamente para realizar de forma eficiente operaciones en las que se ven involucrados elementos de matrices, denominados vectores. Estos procesadores resultan especialmente útiles para ser utilizados en el cálculo científico de alto rendimiento (*high performance computing*), donde las operaciones con vectores y con matrices son ampliamente utilizadas. Sin embargo la velocidad de estos procesadores, si bien ha crecido en los últimos años, tiene un límite físico dado por la velocidad de la luz en el vacío. Esta velocidad es aproximadamente igual a 3×10^8 [m/s]. Si deseamos que un aparato electrónico realice 10^{12} operaciones por segundo, encontraremos que le toma más tiempo a la señal viajar entre dispositivos situados medio milímetro de distancia, que lo que le toma a cada uno de ellos procesar la señal. Desgraciadamente, la física impone también límites al tamaño y distancias mínimas entre dispositivos, pues hay un punto en que comienzan a interaccionar reduciendo no solo su velocidad sino también su confiabilidad. Por ello, pareciera que la única solución al problema es el paralelismo. Aquí la idea es que si varias operaciones se realizan simultáneamente en diversos procesadores, entonces el tiempo de realización de una tarea completa es reducido significativamente distribuyendo la carga de trabajo entre los procesadores disponibles.

El cómputo paralelo ha crecido gracias a la necesidad insaciable por poder de cómputo útil en muchas industrias:

- Petróleo: análisis de reservas.
- Automotriz: simulación de choques, análisis de arrastre, eficiencia de combustión.
- Aeronáutica: análisis de flujo de aire, eficiencia de motores, mecánica estructural, electromagnetismo.
- CAD (Diseño Asistido por computadora).
- Farmacéuticas: modelado molecular.
- Visualización: para todo lo anterior, entretenimiento (películas), Arquitectura (simulaciones 3D y dibujos a color).

- Bancos: Bases de Datos, procesamiento de transacciones en línea, soporte de decisiones, minería de datos.

Paralelizar un proceso consiste en mejorar el tiempo de ejecución de un programa específico mediante la minimización de un problema en problemas más simples, es decir, separar una tarea en subtareas para que éstas puedan tratarse por varios procesadores de forma paralela y simultánea.

Este tiempo de ejecución del programa está dado por una unidad de medida llamada flops, los cuales son la cantidad de operaciones de punto flotante que puede realizar un microprocesador (*floating-point operation*). Incluye cualquier operación que involucre números fraccionarios debido a que este tipo de operaciones requiere de mayor tiempo para ejecutarse que las operaciones con enteros.

La mayoría de los procesadores modernos incluyen una unidad de punto flotante (FPU), que es la encargada de la ejecución de las operaciones con punto flotante, por lo cual, la medida flops, específicamente mide la velocidad de la FPU. La prueba más común que se aplica para medir los flops es la linpack.

La prueba de linpack, desarrollada por Jack J. Dongarra del Argonne National Laboratory de los Estados Unidos, se ha convertido en los últimos años en la prueba de referencia para medir la eficiencia de sistemas informáticos dedicados esencialmente al cálculo científico y técnico, es decir, a procesos de tratamiento numérico.

La prueba consiste en la resolución de un sistema, denso y aleatorio, de 100 ecuaciones lineales con 100 incógnitas, utilizando subrutinas FORTRAN de los paquetes linpack y BLAS. El tiempo de ejecución -tiempo de Unidad Central- se divide por el número de operaciones (adiciones y multiplicaciones) en punto flotante (flops), para dar una medida de la potencia de cálculo del sistema, evaluada en Mflops/seg.

La importancia y significación de esta prueba, para los sistemas dedicados a la computación, se debe fundamentalmente a las siguientes características:

- Proporciona una métrica -Mflops/seg- que es independiente de la arquitectura y de la configuración del sistema y depende sólo del proceso numérico a realizar.
- Es una medida real, obtenida a partir de una prueba y no a partir de características teóricas del fabricante, como tiempos de ciclo, tamaño de memoria caché, etc.
- Al ser una prueba realizada con un programa FORTRAN, incluye una medida de la eficiencia del compilador, por lo que no sólo se mide la capacidad de la Unidad Central, sino de todo el sistema.
- Es un programa de utilización masiva en todos los campos de la ciencia y de la técnica. Pues analizando algunos programas reales utilizados frecuentemente en la ciencia e ingeniería, permitieron determinar que las operaciones más frecuentes y responsables de la mayor parte del tiempo consumido, eran análogas a la operación, realizado en la prueba de linpack.

- En todo caso, esta prueba puede considerarse como una indicación de orientación de la potencia de un sistema, pero no debe ser usada para efectuar -sobre su única base- un proceso de selección de sistemas. Aunque puede, y probablemente debe, formar parte de un paquete de pruebas que simule la carga de trabajo en un entorno dado.

1.1.2 Procesamiento concurrente

Concurrencia

Los procesos son concurrentes si existen simultáneamente. Los procesos concurrentes pueden funcionar en forma totalmente independiente unos de otros, o pueden ser asíncronos, lo cual significa que en ocasiones requieren cierta sincronización y cooperación.

La concurrencia la podemos encontrar de dos formas:

- *Concurrencia implícita:* Es la concurrencia interna al programa, por ejemplo cuando un programa contiene instrucciones independientes que se pueden realizar en paralelo, o existen operaciones de E/S que se pueden realizar en paralelo con otros programas en ejecución. Está relacionada con el paralelismo hardware.
- *Concurrencia explícita:* Es la concurrencia que existe cuando el comportamiento concurrente es especificado por el diseñador del programa. Está relacionada con el paralelismo software.

Una forma de ver la concurrencia es como un conjunto de actividades que se desarrollan de forma simultánea. Cada una de esas actividades se suele llamar proceso.

Procesamiento Paralelo y Concurrente

El paralelismo es un caso particular de la concurrencia. Se habla de paralelismo cuando:

- Ocurre la ejecución simultánea de instrucciones (*pipeline*) en donde se reduce el número de ciclos de máquina necesarios para la ejecución de la instrucción, ya que esta técnica permite que una instrucción pueda empezar a ejecutarse antes de que haya terminado la anterior.
- Arquitecturas paralelas en donde existen múltiples procesadores que operan sobre múltiples flujos de datos.
- Procesamiento paralelo en donde varios procesadores ejecutando simultánea y coordinadamente instrucciones, pueden rendir más que un solo procesador.
- Algoritmos paralelos, es decir, un algoritmo que puede ser ejecutado por partes en el mismo instante de tiempo por varias unidades de procesamiento, para finalmente unir todas las partes y obtener el resultado correcto.
- Programación paralela, técnica de programación basada en la ejecución simultánea, bien sea en un misma computadora con uno o varios procesadores.

Existen ocasiones en las que se confunde el término concurrencia con el término paralelismo. La concurrencia se refiere a un paralelismo potencial, que puede o no darse; por lo que es habitual encontrar el término de programa concurrente en el mismo contexto que el de programa paralelo o distribuido. Existen diferencias sutiles entre estos conceptos:

- *Programa concurrente:* Es aquél que define acciones que pueden realizarse simultáneamente.
- *Programa paralelo:* Es un programa concurrente diseñado para su ejecución en un hardware paralelo.
- *Programa distribuido:* Es un programa paralelo diseñado para su ejecución en una red de procesadores autónomos que no comparten la memoria.

El término concurrente es aplicable a cualquier programa que presente un comportamiento paralelo actual o potencial. En cambio el término paralelo o distribuido es aplicable a aquel programa diseñado para su ejecución en un entorno específico. Cuando se emplea un solo procesador para la ejecución de programas concurrentes se habla de seudoparalelismo.

Características de la concurrencia

Los procesos concurrentes tienen las siguientes características:

1. *Indeterminismo:* Las acciones que se especifican en un programa secuencial tienen un orden total en el conjunto de tareas (instrucciones) que establece, pero en un programa concurrente el orden es parcial, ya que existe una incertidumbre sobre el orden exacto de ocurrencia de ciertos sucesos, esto es, existe un indeterminismo en la ejecución. La presencia de esta propiedad en un programa puede crear problemas al programador ya que puede producir fallos provenientes de errores transitorios. Un error transitorio es aquél que puede ocurrir dependiendo del orden en que se ejecutan las tareas en una activación concreta del programa.

2. *Interacción entre procesos:* Los programas concurrentes implican interacción entre los distintos procesos que los componen. En general, la interacción entre procesos se produce en tres circunstancias diferentes:

   - Cuando los procesos compiten por acceder a un recurso compartido.
   - Cuando los procesos necesitan suspender temporalmente su ejecución.
   - Cuando los procesos se comunican entre sí para intercambiar datos.

   En estas situaciones se necesita que los procesos sincronicen su ejecución, para evitar conflictos o establecer contacto para el intercambio de datos. La interacción entre procesos se logra mediante variables compartidas o bien mediante el paso de mensajes. Además la interacción puede ser explícita, si aparece en la descripción del programa, o implícita, si aparece durante la ejecución del programa.

3. *Gestión de recursos:* Los recursos compartidos necesitan una gestión especial. Un proceso que desee utilizar un recurso compartido debe solicitar dicho recurso, esperar a adquirirlo, utilizarlo y después liberarlo. Si el proceso solicita el recurso pero no puede adquirirlo en ese momento, es suspendido hasta que el recurso está disponible. La gestión de recursos compartidos es problemática y se debe realizar de tal forma que se eviten situaciones de retraso indefinido (esperar indefinidamente por un recurso) y de *deadlock* (bloqueo indefinido o abrazo mortal).

4. *Comunicación:* La comunicación entre procesos puede ser síncrona, cuando los procesos necesitan sincronizarse para intercambiar los datos, o asíncrona, cuando un proceso que suministra los datos no necesita esperar a que el proceso receptor los recoja, ya que los deja en un buffer de comunicación temporal.

Problemas de la concurrencia

1. *Violación de la exclusión mutua:* Cuando un proceso usa un recurso del sistema, realiza una serie de operaciones sobre el recurso y después lo deja de usar. A la sección de código que usa ese recurso se le llama región crítica. La condición de exclusión mutua establece que solamente se permite a un proceso estar dentro de la misma región crítica. Esto es, que en cualquier momento solamente un proceso puede usar un recurso a la vez. Para logar la exclusión mutua generalmente se usan algunas técnicas para lograr entrar a la región crítica: semáforos, monitores, el algoritmo de Dekker y Peterson o candados.

2. *Bloqueo mutuo o deadlock:* Un proceso se encuentra en estado de *deadlock* si está esperando por un suceso que puede o no ocurrir. Se puede producir en la comunicación de procesos y más frecuentemente en la gestión de recursos. Existen cuatro condiciones necesarias para que se pueda producir *deadlock:*

   a) Los procesos necesitan acceso exclusivo a los recursos.
   b) Los procesos necesitan mantener ciertos recursos exclusivos mientras esperan por otros.
   c) Los recursos no se pueden obtener de los procesos que están a la espera.
   d) Existe una cadena circular de procesos en la cual cada proceso posee uno o más de los recursos que necesita el siguiente proceso en la cadena. Por ejemplo, suponga que el proceso A tiene asignado el recurso R1 y el proceso B tiene asignado el recurso R2. En ese momento al proceso A se le ocurre pedir el recurso R2 y al proceso B el recurso R1. Ahí se forma una espera circular entre estos dos procesos que se puede evitar quitándole a la fuerza un recurso a cualquiera de los dos procesos.

3. *Retraso indefinido o starvation:* Un proceso se encuentra en *starvation* si es retrasado indefinidamente esperando un suceso que nunca ocurrirá. Esta situación se puede producir si la gestión de recursos emplea un algoritmo en el que no se tenga en cuenta el tiempo de espera del proceso. Por ejemplo, si un proceso ocupa un recurso y lo marca como “ocupado” y termina sin marcarlo como “desocupado” y hay algún otro proceso que pida ese mismo recurso, este proceso lo verá “ocupado” y esperará indefinidamente a que se “desocupe”.

4. *Injusticia o unfairness:* Se pueden dar situaciones en las que exista cierta injusticia en relación a la evolución de un proceso. Se deben evitar estas situaciones de tal forma que se garantice que un proceso evoluciona y satisface sus necesidades sucesivas en algún momento.

5. *Espera ocupada:* Esta condición consiste en que un proceso pide un recurso que ya está asignado a otro proceso y dicho recurso no puede arrebatársele por ningún motivo, y estará disponible hasta que el proceso lo suelte por su voluntad. Entonces el primer proceso estará gastando o desperdiciando el resto de su tiempo de ejecución revisando si el recurso fue liberado. La solución más común a este problema consiste en que el sistema operativo se dé cuenta de esta situación y mande a una cola de espera al proceso, otorgándole inmediatamente el turno de ejecución a otro proceso.

1.1.3 Breve historia del paralelismo

1955 El IBM 704 usa circuitos aritméticos paralelos binarios junto con una unidad de punto flotante que aceleraban significativamente el desarrollo de operaciones numéricas frente a las tradicionales unidades aritmético-lógicas. Su velocidad era de 5kFLOPS aproximadamente, pero las operaciones de E/S resultaban lentas y representaban un cuello de botella, así que IBM incorpora procesadores de E/S independiente, la IBM 709.

1956 IBM inicia el proyecto 7030 (también llamado STRETCH) para crear una supercomputadora con 100 veces mayor velocidad.

1958 Bull anuncia la Gamma 60 con múltiples unidades funcionales e instrucciones *fork* y *join* en su conjunto de instrucciones. Posteriormente Slotnick propone la SOLOMON, una máquina SIMD con 1024 elementos de procesamiento de 1 bit, cada uno con memoria para 128 valores de 32 bits. La máquina nunca se construye pero es el punto de arranque para trabajos posteriores.

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

IBM entrega su primera STRETCH, que presentaba la anticipación de instrucciones y corrección de errores.

1960 Control Data inicia el desarrollo de su CDC 6600.
E.V. Yevreinov en el Instituto de Matemáticas en Novosibirsk (IMN) comienza sus trabajos en arquitecturas fuertemente acopladas de paralelismo burdo con interconexiones programables.

1962 La computadora Atlas es operacional. Es la primera máquina en usar memoria virtual y paginación, su ejecución de instrucciones es en entubamiento *(pipeline*), y contiene unidades aritméticas de punto flotante y punto fijo separadas. Su desempeño es de aproximadamente 200 Kflops.

Sale un multiprocesador MIMD simétrico Cuenta de 1 a 4 CPUs que acceden a 1 ó 16 módulos de memoria usando un conmutador de barraje cruzado (*crossbar switch*).

1964 CDC 6600, la primera supercomputadora en ser un éxito técnico y comercial. Cada máquina tiene una CPU de 60 bits y 10 unidades periféricas de procesamiento (PPUs). La CPU utiliza un marcador para manejar la dependencia de instrucciones.

1965 General Electric, el MIT, y AT&T Bell Laboratories comienzan la construcción de un sistema operativo de propósito general de memoria compartida, multiprocesamiento y tiempo compartido.

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

1984 Hasta este momento el paralelismo estaba limitado a entubamientos (*pipelining*) y procesamiento de vectores, o cuando mucho a algunos procesadores compartiendo trabajos.

Introducción de máquinas con cientos de procesadores que podían estar trabajando de manera simultánea en diferentes partes de un mismo programa.

Utilización de redes de computadoras y estaciones de trabajo de un solo usuario.

1985-1990 Se desarrolló un tercer tipo de procesadores paralelos conocidos por paralelo-datos o SIMD, en los que podían existir varios miles de procesadores muy simples trabajando coordinadamente con una misma unidad de control; esto es, todos los procesadores trabajaban en la misma tarea con variables locales.

1990 Sistemas paralelos compiten con los procesadores de vectores en términos del poder total de cómputo. Combinaciones de arquitecturas paralelas / vector se establecen. Se establecen metas para alcanzar los Teraflops 10^{12} operaciones aritméticas por segundo.

1990 a la fecha Las estaciones de trabajo siguen mejorándose con el diseño de procesadores que utilizan combinaciones de RISC, entubamiento (*pipelining*) y procesadores paralelos.

1.1.4 Procesamiento paralelo

El procesamiento paralelo es un término que se usa para denotar un grupo de técnicas significativas que se usan para proporcionar tareas simultáneas de procesamiento de datos con el fin de aumentar la velocidad computacional de un sistema de computadoras. En lugar de procesar cada instrucción en forma secuencial como en una computadora convencional, un sistema de procesamiento paralelo puede ejecutar procesamiento concurrente de datos para conseguir un menor tiempo de ejecución. Por ejemplo, cuando se ejecuta una instrucción en la ALU, puede

leerse la siguiente instrucción de la memoria. El sistema puede tener 2 o más ALUS y ser capaz de ejecutar dos o más instrucciones al mismo tiempo. Además, el sistema puede tener dos o más procesadores operando en forma concurrente.

El propósito del procesamiento paralelo es acelerar las posibilidades de procesamiento de la computadora y aumentar su eficiencia, esto es, la capacidad de procesamiento que puede lograrse durante un cierto intervalo de tiempo. La cantidades de circuitería aumenta con el procesamiento paralelo y, con él, también el costo del sistema. Sin embargo, los descubrimientos tecnológicos han reducido el costo de la circuitería a un punto en donde las técnicas de procesamiento paralelo son económicamente factibles.

El procesamiento paralelo puede considerarse de diversos niveles de complejidad. En el nivel más bajo, distinguimos entre operaciones seriales y paralelas mediante el tipo de registros que utilizan. Los registros de corrimiento operan en forma serial un bit a la vez, mientras que los registros con carga paralela operan con todos los bits de la palabra en forma simultánea. Puede obtenerse procesamiento paralelo a un nivel más alto de complejidad al tener múltiples unidades funcionales que ejecuten operaciones idénticas o diferentes, de manera simultánea. El procesamiento paralelo se establece al distribuir los datos entre las unidades funcionales múltiples.

Existen varias maneras de clasificar el procesamiento paralelo. Puede considerarse a partir de la organización interna de los procesadores, desde la estructura de interconexión entre los procesadores o desde del flujo de información a través del sistema.

1.1.5 Arquitecturas paralelas

El procesamiento paralelo puede ocurrir en el flujo de instrucciones, en el flujo de datos o en ambos. La clasificación clásica de arquitecturas de computadoras que hace alusión a sistemas con uno o varios procesadores es la de *J.M.Flynn*. La publicó por primera vez en 1966 y por segunda vez en 1970. Esta taxonomía se basa en el número de flujos que siguen los datos dentro de la máquina y del número de instrucciones sobre esos datos.

Se define como flujo de instrucciones al conjunto de instrucciones secuenciales que son ejecutadas por un único procesador y como flujo de datos al flujo secuencial de datos requeridos por el flujo de instrucciones.

Flynn propuso un esquema de clasificación de los sistemas en cuatro categorías:

- SISD *Single Instruction, Single Data.*
- SIMD *Single Instruction, Multiple Data*.
- MISD *Multiple Instruction, Single Data*.
- MIMD *Multiple Instruction, Multiple Data*.

SISD (*Single Instruction, Single Data*)

Un sistema con un único flujo de instrucciones sobre un único flujo de datos se llama SISD (*Single Instruction, Single Data*). Como lo muestra la figura 1.1, se ejecuta una instrucción detrás de otra. Todas las computadoras tradicionales de un procesador caen dentro de esta categoría (máquinas secuenciales).

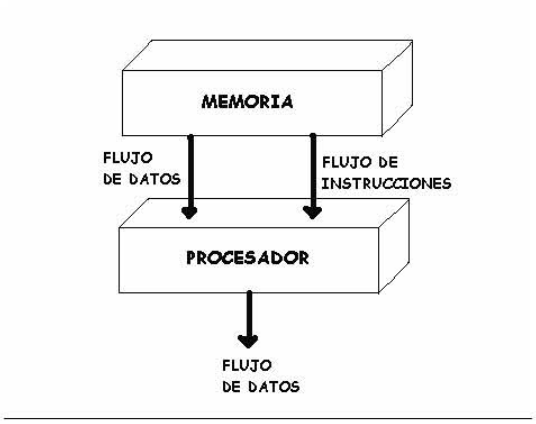


Figura 1.1 Arquitectura SISD *Single Instruction Single Data.*

Por ejemplo, para procesar la suma de *N* números a_1, a_2, ... $a_{N..}$ el procesador necesita acceder a memoria N veces consecutivas para recibir un número, también son ejecutadas en secuencia N-1 adiciones.

SIMD (*Single Instruction, Multiple Data*)

La siguiente categoría es SIMD (*Single Instruction, Multiple Data*). Estos sistemas tienen un único flujo de instrucciones que operan sobre múltiples flujos de datos. A diferencia de SISD, en este caso se tienen múltiples procesadores que sincronizadamente ejecutan la misma secuencia de instrucciones, pero en diferentes datos.

Estas máquinas son útiles para las operaciones que repiten los mismos cálculos en un conjunto de datos (vectores). Por ejemplo, para sumar dos vectores de 64 entradas, se envían 64 flujos de datos a 64 ALUs para formar 64 sumas en un solo ciclo.

Las virtudes de SIMD son que todas las unidades de ejecución paralela están sincronizadas donde un reloj global se encarga de esto, y todas responden a una sola instrucción que emana de un único contador de programa (PC):

- Una memoria central contiene los programas y una unidad de control centralizada se encarga de extraer cada instrucción y ejecutarla. Aunque cada unidad ejecutará la misma instrucción, cada unidad de ejecución tiene sus registros de dirección propios y por lo tanto cada unidad puede tener diferentes direcciones de datos.

- La motivación original para SIMD fue amortizar el coste de la unidad de control mediante varias unidades de ejecución.
- Otra ventaja es el tamaño reducido de la memoria para almacenar el programa, ya que sólo se necesita una copia. La memoria virtual y el incremento de la capacidad de los chips de memoria han reducido la importancia de esta ventaja.
- Habitualmente, hay un procesador central SISD que realiza las operaciones secuenciales como los saltos.
- Las instrucciones SIMD se envían a todas las unidades de ejecución, cada una con su propio conjunto de registros y memoria.
- Las unidades de ejecución dependen de la red de interconexión para intercambiar datos.
- SIMD funciona mejor cuando trata con vectores en bucles *for*.
- SIMD tiene su peor rendimiento en las sentencias *case* o *switch*, donde cada unidad de ejecución debe ejecutar una operación diferente con sus datos, dependiendo de los datos que tenga. Básicamente, estas unidades tienen un rendimiento de una *n*-ésima parte, siendo *n* el número de casos del condicional.

La figura 1.2 muestra ésta arquitectura:

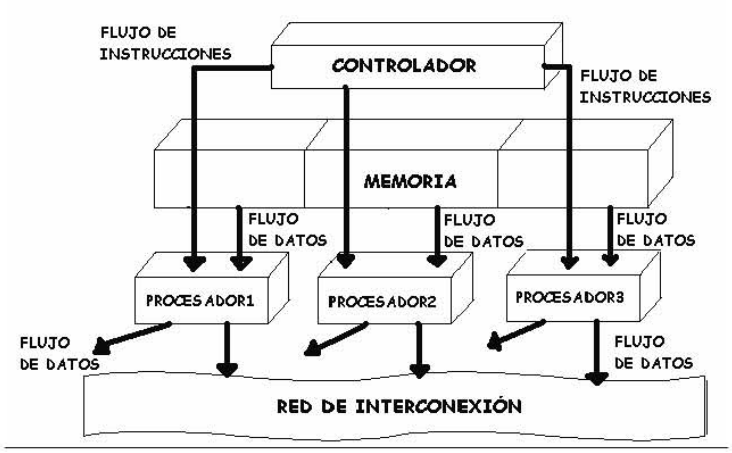


Figura 1.2 Arquitectura SIMD *Single Instruction Multiple Data*

La SIMD más conocida es la Illiac IV, sin duda el proyecto de supercomputadora más desastroso, aunque aportó mucha tecnología útil para futuros proyectos. Los costes subieron de los 8 millones de dólares presupuestados en 1966 a los 31 millones en 1972, a pesar de que sólo se construyó una cuarta parte de la máquina proyectada. El rendimiento máximo fue de 15 Mflops comparado con las predicciones iniciales de 1000 Mflops para el sistema completo.

Entregado a la NASA en 1972, pasaron 3 años hasta que el sistema fue operacional. Los sucesores SIMD del Illiac IV incluyen el ICL DAP, el Goodyear MPP, las Thinking Machines CM-1 y CM-2 y los Maspar MP-1 y MP-2.

Un modelo relacionado con los SIMD es el procesamiento vectorial. Es una arquitectura sólidamente establecida. Ha popularizado considerablemente los supercomputadores y es la alternativa SIMD más usada. La diferencia con un SIMD clásico es que los procesadores vectoriales dependen de unidades funcionales segmentadas que suelen operar con unos pocos elementos del vector por ciclo, mientras que las máquinas SIMD normalmente operan todos los elementos a la vez.

Por ejemplo, para sumar dos matrices $A + B = C$, siendo A y B de orden 2 y teniendo 4 procesadores:

$$A_{11} + B_{11} = C_{11} \qquad A_{12} + B_{12} = C_{12}$$
$$A_{21} + B_{21} = C_{21} \qquad A_{22} + B_{22} = C_{22}$$

La misma instrucción es ejecutada en los 4 procesadores (sumando dos números) y los 4 ejecutan las instrucciones simultáneamente. Esto toma un paso en comparación con cuatro pasos en una máquina secuencial.

MISD (*Multiple Instruction, Single Data*)

La siguiente categoría es MISD (*Multiple Instruction, Single Data*), sistemas con múltiples instrucciones que operan sobre un único flujo de datos. Los sistemas MISM se contemplan de dos maneras distintas:

1. Varias instrucciones operando simultáneamente sobre un único dato.
2. Secuencias de instrucciones que pasan a través de múltiples procesadores, es decir, se realizan diversas operaciones en diversos procesadores. N procesadores, cada uno con su propia unidad de control comparten una memoria común. Aquí hay N secuencias de instrucciones (algoritmos /programas) y una secuencia de datos. Cada procesador recibe distintas instrucciones que operan sobre el mismo flujo de datos y sus derivados. Los resultados de un procesador pasan a ser la entrada (los operandos) del siguiente procesador. El paralelismo es alcanzado dejando que los procesadores realicen diferentes cosas al mismo tiempo en el mismo dato.

La figura 1.3 muestra ésta arquitectura:

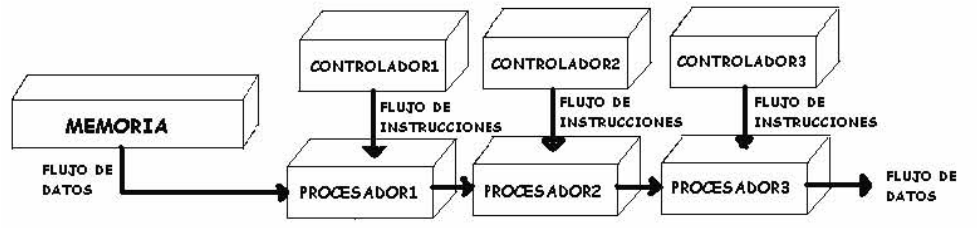


Figura 1.3 Arquitectura MISD *Multiple Instruction Single Data*

Ejemplos de estos tipos de sistemas son los *arrays* sistólicos o *arrays* de procesadores.

MIMD (*Multiple Instruction, Multiple Data*).

Por último, están los sistemas MIMD (*Multiple Instruction, Multiple Data*), es decir, un grupo de computadoras independientes cada una de ellas con su propio contador de programa y sus propios datos. Son sistemas con flujo de múltiples instrucciones que operan sobre múltiples datos. La diferencia con estos sistemas y los SIMD, es que MIMD es asíncrono. No tiene un reloj central. Cada procesador en un sistema MIMD puede ejecutar su propia secuencia de instrucciones y tener sus propios datos. Se tienen N procesadores, N secuencias de instrucciones y N secuencias de datos.

La idea de los MIMD es crear potentes computadoras conectando sencillamente otras muchas pero pequeñas. Presentan la ventaja de tolerancia a fallos, es decir, si disponemos de n procesadores siempre podemos continuar la unidad de procesamiento con *n-1*, lo cual es garantía en máquinas a las que se les exige un servicio continuo.

Se puede dividir a los MIMD en dos familias:

1 *Multiprocesadores de memoria compartida.*

Usan una memoria centralizada (puede estar dividida en módulos) para almacenar los datos y para sincronizarse. La dirección 100 es la misma para todos los procesadores. Como el acceso a la memoria puede ser conflictivo, normalmente cada procesador dispone de una memoria caché, la cual suele traer problemas de coherencia.

Los procesadores con memoria compartida tienen dos variantes:

Multiprocesadores UMA (acceso uniforme a memoria) o SMP (multiprocesadores simétricos), con un tiempo de acceso a memoria igual para todos los procesadores y todas las direcciones de memoria. En particular, la red que interconecta áreas de memoria es una colección de árboles de switches y memorias. Consecuentemente cada procesador tiene algunas memorias que están cercanas y otras que son lejanas. Esta organización elimina la congestión que podría ocurrir si hubiera un solo bus o switch compartido.

Multiprocesadores NUMA (acceso no uniforme a memoria). El tiempo de los accesos a memoria es no uniforme. Cada procesador tiene su propia memoria caché; si dos procesadores referencian diferentes localidades de memoria los contenidos pueden ser colocados de manera segura en los cachés de ambos procesadores. Pero los problemas pueden crecer cuando dos procesadores referencian a la misma localidad de memoria al mismo tiempo.

Si ambos procesadores leen la misma localidad de memoria cada uno puede cargar una copia del dato que contiene en su caché local. Pero si un procesador escribe en la misma localidad de memoria se presenta un problema de consistencia del caché: el caché del otro procesador no contiene el valor correcto. Así pues, cualquiera de los otros cachés necesitarán ser invalidadas. Cada procesador tiene que implementar un protocolo de consistencia de caché en

su hardware. Un método para ello consiste en tener cada caché vigilado en el bus de direcciones de memoria observando las referencias hechas a localidades de este caché.

La escritura también conduce a problemas de consistencia en memoria: ¿cuándo es actualizada realmente la memoria primaria? Por ejemplo, si un procesador escribe en una localidad de memoria y otro procesador lee posteriormente de esa localidad, ¿el segundo procesador tiene la garantía de que verá el nuevo valor? Existen varios modelos de consistencia diferentes.

El problema de la consistencia de la memoria plantea una elección entre la facilidad en la programación y la redundancia en la implementación. El programador intuitivamente espera que las actualizaciones de memoria ocurran en algún orden secuencial y que cada procesador vea el mismo orden, debido a que un programa lee y escribe variables sin respetar en donde serán almacenados los valores dentro de la máquina.

En particular cuando a un proceso se le asigna una variable, el programador espera que el resultado de esa asignación sea vista inmediatamente por todos los demás procesos en el programa.

Consecuentemente los multiprocesadores típicamente implementan un modelo más débil de consistencia y dejan al programador la tarea de insertar instrucciones de sincronización de memoria. Los compiladores y bibliotecas con frecuencia toman cuidados especiales en este aspecto de tal manera que el programador de aplicaciones no tiene por que hacerlo.

Imagine por ejemplo que las variables *x* y *y* ocupan una palabra cada una, y son almacenadas como palabras adyacentes en la memoria, que es mapeada en la misma línea de caché. Supongamos que algún proceso está ejecutándose en el procesador 1 de un multiprocesador, leyendo y escribiendo la variable *x*. Finalmente supongamos que otro proceso se está ejecutando en el procesador 2, escribiendo y leyendo la variable *y*. Entonces en cualquier momento que el procesador 1 accede a la variable *x* la línea de caché de ese procesador también contendrá una copia de la variable *y*; y un proceso similar ocurrirá con el procesador 2.

El escenario es una falsa compartición, el proceso en realidad no comparte las variables *x* y *y*, pero el hardware trata a ambas variables como una unidad. Consecuentemente, cuando el proceso 1 actualiza *x*, la línea de caché conteniendo a ambas *x* y *y* en el procesador 2 tiene que ser invalidada o actualizada. En forma similar, cuando el procesador 2 actualiza *y* la línea de caché conteniendo *x* y *y* en el procesador 1 tiene que ser actualizada o invalidada.

Las invalidaciones o actualizaciones en caché disminuyen la velocidad del sistema de memoria, así que el programa se ejecutará mucho más lento que el caso en el que las dos variables se ubiquen en distintas líneas de caché. El punto clave es que el programador entienda que los procesos no comparten variables porque en realidad el sistema de memoria implementa la redundancia y la compartición para lidiar con ese tipo de problemas. Para evitar la falsa compartición, el programador debe asegurarse que las variables que se escriben para diferentes procesos no deben estar en localidades de memoria adyacentes.

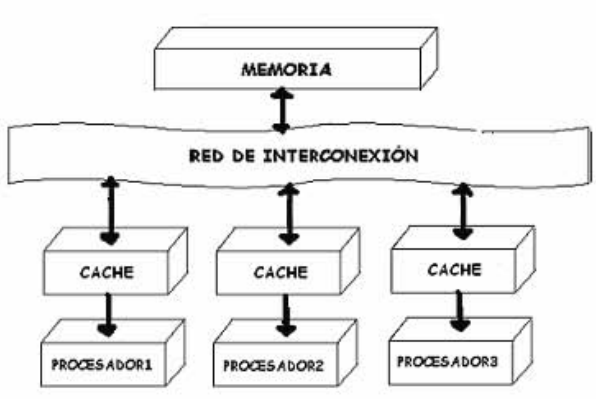


Figura 1.4 Multiprocesadores con memoria compartida

2 *Multiprocesadores con memoria no común (distribuida).*

Se presenta de nuevo una red de interconexión, pero cada procesador tiene su propia memoria privada, no accesible desde otro procesador y se comunican mediante paso de mensajes en lugar de lectura/escritura de memoria, es decir, si un procesador requiere los datos contenidos en la memoria de otro procesador, deberá enviar un mensaje solicitándolos. Cada procesador tiene su propia caché, pero debido a que la memoria no es compartida no existen problemas de consistencia en memoria o caché.

Las computadoras MIMD de memoria distribuida son conocidas como sistemas de procesamiento en paralelo masivo (MPP) donde múltiples procesadores trabajan en diferentes partes de un programa, usando su propio sistema operativo y memoria. También se les conoce como multicomputadoras ya que cada uno de ellos puede actuar como una computadora independiente. Un ejemplo común es una colección de computadoras personales conectadas mediante una red. La red de interconexión provee alta velocidad y gran ancho de banda como ruta de comunicación entre procesadores.

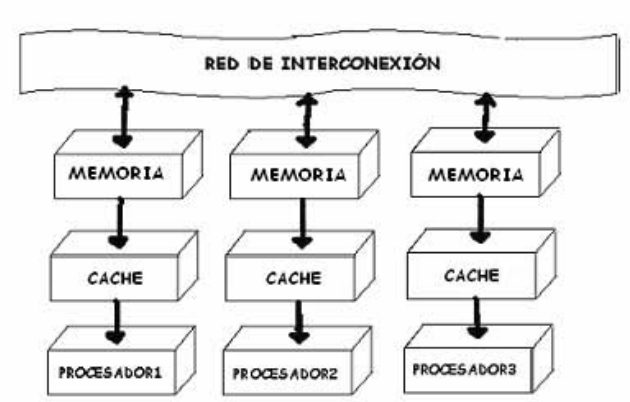


Figura 1.5 Arquitectura MIMD *Multiple Instruction Multiple Data*

Una multicomputadora es usada por una o al menos unas cuantas aplicaciones al mismo tiempo, y cada aplicación usa un conjunto de procesadores dedicados.

Se dice que una computadora del tipo MIMD es fuertemente acoplada (*tightly coupled*) si el grado de interacciones entre sus procesadores es alto. De no ser así se considera débil acoplada (*loosely coupled*).

Los *sistemas de multiprocesamiento débilmente acoplados* no tienen la cantidad de conflictos con la memoria como aquellos que experimentan los sistemas fuertemente acoplados. En estos sistemas de acoplamiento débil, cada procesador tiene un conjunto de dispositivos de entrada/salida y una gran cantidad de memoria local a través de la cual acceden a la mayor parte de las instrucciones y datos.

El grado se acoplamiento de tales sistemas es muy débil. El factor que determina el grado de acoplamiento es la topología de comunicación del sistema de transferencia de mensajes asociado. Los sistemas débilmente acoplados son usualmente eficientes cuando las interacciones entre tareas son mínimas.

Un ejemplo de una arquitectura débilmente acoplada está en los clusters tipo Beowulf. En donde cada nodo es un sistema de cómputo (computadora por separado) interconectado a través de un canal de datos (comúnmente Ethernet).

En un *sistema de multiprocesadores fuertemente acoplado* los multiprocesadores se comunican a través de una memoria principal y compartida, de esta manera la razón de la velocidad a la cual los procesadores pueden comunicarse de un procesador a otro es el orden del ancho de banda de la memoria.

Los sistemas fuertemente acoplados pueden tolerar un alto grado de interacciones entre las tareas sin un significativo deterioro en su desempeño. Un ejemplo de máquinas de multiprocesamiento fuertemente acoplado es el *Transputer*, una máquina en donde dos o más procesadores comparten la misma memoria.

El *Transputer* es un microprocesador diseñado especialmente para computación paralela. Reúne en un solo chip un procesador, una memoria local y canales de comunicación que proporcionan una conexión punto-a-punto con otros *Transputers*. El control complejo se realiza por microcódigo. En particular, cada *Transputer* puede soportar varios procesos que se ejecutan concurrentemente. El *Transputer* que más difusión ha tenido hasta el momento es el T8000. Para posibilitar las redes de comunicación de alta velocidad, hace pocos años se ha desarrollado la segunda generación de *Transputer*s, denominado T9000. Sobre todo a nivel de la Comunidad Económica Europea, el diseño de máquinas basadas en el *Transputer* es muy importante.

El *Transputer* es un claro ejemplo de las computadoras de un solo chip, la cual, generó las facilidades para la creación de sistemas de procesamiento paralelo. Más de cuatro ligas de comunicación por *Transputer* forman redes de *Transputer*s que son construidas por conexiones punto a punto con ninguna lógica externa. Las ligas soportan la velocidad de operación estándar,

que es de 10 MB por segundo, pero también operan desde 5 a 20 MB por segundo. El *Transputer* es un microcomputador monochip dotado de su propia memoria local que incorpora diversos enlaces para facilitar la conexión con otros *Transputer*. Esta familia de dispositivos puede utilizarse bien en sistemas monoprocesadores y en redes para facilitar y permitir la construcción de sistemas concurrentes de altas prestaciones utilizando un sistema de comunicación punto a punto.

En cuanto a su programación, el *Transputer* es un componente que puede programarse para realizar una función especializada, por lo que puede ser considerado como una caja negra. A nivel interno, un *Transputer* está formado por una memoria, un procesador y su correspondiente sistema de comunicación conectado a un bus de 32 bits. Este, a su vez, se encarga de realizar la conexión con la interface de memoria externa, característica que facilita la utilización de memoria adicional. El procesador, la memoria y el sistema de comunicaciones, ocupan cada uno aproximadamente el 25% del total del área de silicio, mientras que el resto esta asignado a la distribución de la señal de alimentación, los generadores de reloj y las conexiones externas.

Los *Transputer*s están dirigidos hacia dos usos fundamentales:

1. Ser el elemento básico para construir prototipos de sistemas concurrentes de uso específico, sirviendo para modelar procesos que, en última instancia, serán realizados en hardware. En particular, está dotado para construir sistemas multiprocesador basados en paso de mensajes con topología de interconexión en malla.

2. Ser el bloque fundamental para construir sistemas multiprocesador de elevadas prestaciones. El único sistema que lo utiliza es la serie T, se trata de un sistema multiprocesador de gran paralelismo orientado hacia el cálculo científico. Sus estructuras básicas son los elementos de proceso (canales de comunicación, procesador de control, memoria, unidad de proceso de vectores) conectados entre sí a través de una red de tipo hipercubo. Otras aplicaciones van desde estaciones de trabajo a Inteligencia Artificial, pasando por procesamiento de imágenes, telecomunicaciones, robótica, bases de datos distribuidas, etc.

1.2 Sistemas de cómputo distribuidos

1.2.1 Sistemas distribuidos

Antecedentes los Sistemas Distribuidos

Los sistemas distribuidos aparecen a raíz del desarrollo de las redes computacionales y de las computadoras a multiprocesadores. Si bien la aparición de estos dos eventos a sido crucial en el desarrollo de la informática, la gestión de todos los recursos, tanto en una área como en la otra requiere de conocimientos a tomar en cuenta. Los sistemas distribuidos tienden a reunir todos estos conocimientos.

Desde un punto de vista histórico, las primeras primitivas de comunicación y los primeros protocolos que se usaron para el intercambio de información interproceso, proporcionaban paso de mensajes entre dos procesos (comunicación uno a uno), con diferentes calidades de servicio (*Quality of service -QoS*) que especificaban el comportamiento en caso de fallos de comunicación. Más adelante, con la adopción del modelo cliente-servidor, se desarrollaron protocolos uno a uno optimizados para la interacción con RPCs, especificando alguna *QoS* en caso de fallos de proceso.

Pero cuando comenzaron a considerarse aplicaciones tolerantes a fallos, replicadas y de tipo difusión, se reconoció la importancia de enviar mensajes a un conjunto de procesos (comunicación uno a varios) como un paradigma útil. En este escenario, los modelos de envío de mensajes útiles son muchos, y las garantías que son útiles dependen mucho de la aplicación.

A continuación se muestra una lista de la evolución de los sistemas distribuidos:

- Alto, Dorado (Xerox, 70s).
- Estaciones Sun, Macintosh, PCs (80s).
- Sistemas Distribuidos “primitivos”: Xerox Distributed System, Cambridge Distributed Computing System, Locus, Apollo Domain, Newcastle Connection, Grapevine, Cedar, etc. (finales de los 70).
- Modelo cliente/servidor. Sistema donde el cliente es una máquina que solicita un determinado servicio y se le denomina servidor a la máquina que lo proporciona. Los servicios pueden ser la ejecución de un determinado programa, acceso a un determinado banco de información o acceso a un dispositivo de hardware.
- RPC (*Remote Procedure Call* -Llamada a Procedimiento Remoto). Protocolo que permite a un programa ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos. Permite que los programas llamen a subrutinas que se ejecutan en un sistema remoto. El programa llamador, denominado *client* envía un mensaje de llamada al proceso servidor (*server*) y espera por un mensaje de respuesta. La llamada incluye los parámetros del procedimiento y la respuesta los resultados.
- Sprite (mediados 80): caché coherente de ficheros.
- Coda, Echo (final 80, 90s): sistema de ficheros replicados.
- Amoeba (80s): capacidad, simplicidad.

- Andrew y Athena (finales 80): Servicio a comunidades grandes.
- OSF DCE: conjunto completo de interfaces normalizados para un sistema distribuido.
- Corba: Normalización con intención comercial.

Los factores que han ayudado el desarrollo de los sistemas distribuidos son:

1.Avances Tecnológicos.

Desarrollo de microprocesadores más poderosos, más pequeños y más económicos, con una mayor disponibilidad y accesibilidad.

Desarrollo de redes LAN y de comunicación, que permiten conectar computadoras y transferir datos a alta velocidad

2.Ingeniería de software

Disminución de costos

Nuevos requerimientos.

La complejidad de las tareas así como la cantidad de información que se maneja actualmente son dos factores que tienden a incrementarse, y que exigen una demanda directa de poder computacional.

3.Globalización.

La alianza, el crecimiento y la idea de avanzar exigen la incorporación de nuevas instituciones en el mercado, y con ello se debe dar una integración en los sistemas de computo que no serán los mismos en todo el mundo.

4.Integración.

El poder trabajar sin problemas de un sistema operativo a otro es un factor que se busca pues en el mundo existen varios sistemas que se quieren unir y no todos van a contar con el mismo tipo de sistema operativo.

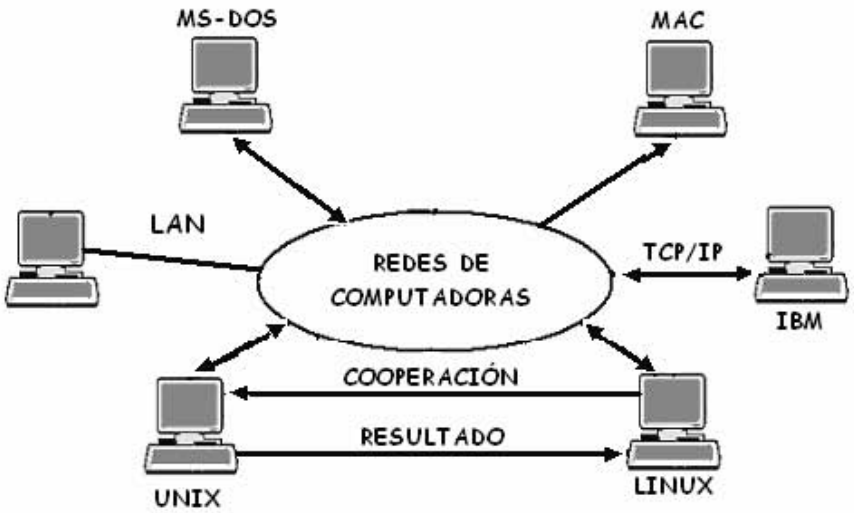


Figura 1.6 Integración en los sistemas distribuidos.

Definición

Un sistema distribuido es un conjunto de hardware y software autónomos que se encuentran físicamente separados y no comparten una memoria común, están conectados a una red de alta velocidad la cual les permite la comunicación y coordinación de acciones mediante el paso de mensajes, para el logro de un objetivo.

Por las características que distinguen a los sistemas distribuidos, se exige para su desarrollo el uso de algoritmos especiales exclusivamente diseñados para estos sistemas. Estos algoritmos reciben el nombre de algoritmos distribuidos. Los algoritmos distribuidos difieren de los algoritmos convencionales, éstos últimos se usan para sistemas de cómputo local, mientras que los primeros están enfocados a ser usados en sistemas distribuidos.

El término cómputo local se refiere a los programas que están confinados a un sólo espacio de direcciones, mientras que el término de cómputo distribuido se refiere a programas que hacen llamadas a otros espacios de direcciones, posiblemente en otra máquina. Es decir, los algoritmos secuenciales y paralelos pueden ser vistos como recipientes que transforman entradas (datos), proporcionadas al principio, en salidas producidas al final del algoritmo. Los algoritmos distribuidos se ejecutan en procesos de sistemas cuyos elementos de procesamiento se encuentran físicamente separados, y sirven para garantizar un comportamiento deseado del sistema.

Los componentes de un sistema distribuido son:

- *Hardware.* Contempla todos los componentes físicos del sistema como son: microprocesadores, memorias, dispositivos de almacenamiento, monitores, teclados, ratones, hubs, routers, switches, etc.

- *Sistema Operativo.* Cada maquina posee su sistema operativo, pero ya trabajando en conjunto se puede decir que el sistema operativo es distribuido lo que implica que los servicios de gestión del sistema de archivos, el control de acceso a recursos (archivos, dispositivos, etc.), el manejo de periféricos (discos, teclados, pantallas, cintas,...), la autenticación de usuarios, la asignación de memoria y CPU, así como la creación y planificación de procesos, pueden ser proporcionados por varios procesos en varias máquinas, por lo que la idea de sistema operativo se difumina a: cliente (pide un servicio), servidor (ofrece un servicio), y comunicaciones.

  Por medio del sistema operativo adecuado, las computadoras mantienen su capacidad de procesamiento de tareas local, mientras constituyen elementos cooperativos de procesamiento en el ambiente distribuido. El elemento de interconexión indica que debe existir el mecanismo de transporte de información entre los componentes de manera que sea factible el intercambio de mensajes entre nodos cooperativos tal que no se violente la transparencia de una transacción.

  Todos los servicios del sistema, los directorios y programas de aplicación frecuentemente usados deben tener el mismo efecto sin importar el punto donde sean ejecutados, y ser

llamados de la misma forma independientemente de la ubicación y notación interna del nodo, a esto se le llama consistencia semántica. El principio de autonomía cooperativa establece que todos los elementos involucrados tienen igual oportunidad en la toma de decisiones que les afectan como un todo. En un momento dado, la toma de decisión se hace sólo entre los procesadores involucrados en una tarea, en tanto que los demás permanecen ajenos a esta decisión.

- *Software Especializado (middleware).* Dentro de este componente, tenemos todo aquel software que nos permite el correcto intercambio de información y la perfecta comunicación entre los distintos equipos del sistema.
- *Aplicaciones y Servicios.* Contempla todas la aplicaciones y servicios que se pueden proporcionar por medio de los sistemas distribuidos.

- *Enlaces de Comunicación.* Todo medio de comunicación que permita el intercambio de información entre los distintos equipos del sistema (cables UTP, fibra óptica, satélites).

Arquitectura en un Sistema Distribuido

La arquitectura que presenta el software en un sistema distribuido se encuentra formada en la capa más externa por las aplicaciones y servicios, enseguida tenemos al *middleware* que es la capa encargada de enmascarar la heterogeneidad del sistema, éste interactúa con la siguiente capa que es el sistema operativo y por último en la capa más interna se encuentra el procesador y el hardware de red.

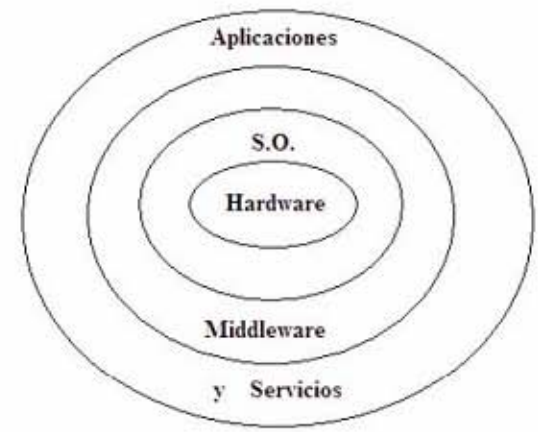


Figura 1.7 Arquitectura de un sistema distribuido

Características de los Sistemas Distribuidos.

a) *Concurrencia:* permite que los recursos disponibles en la red puedan ser utilizados simultáneamente por los usuarios y/o agentes que interactúan en la red.

b) *Transparencia:* La transparencia puede aplicarse a distintos conceptos:

- Heterogeneidad: Los recursos se acceden del mismo modo independientemente de la arquitectura de los nodos, arquitectura de la red, sistema operativo, lenguaje de programación y fabricante del software (p.ej. Middleware como Corba o J2EE).
- Acceso: Recursos locales y remotos se acceden del mismo modo (p. ej. Modelo de objetos remotos de Corba).
- Ubicación: Los recursos pueden ser accedidos sin conocer su ubicación física (p. ej. Mediante servicios de nombrado o de descubrimiento de recursos).
- Replicación: Permite utilizar varios ejemplares de cada recurso. Un recurso replicado puede accederse del mismo modo que uno no replicado.
- Fallos: Los recursos siguen pudiéndose acceder del mismo modo a pesar de que haya fallos.
- Movilidad: Los recursos pueden desplazarse sin afectar a su operación.
- Rendimiento: El sistema equilibra la carga *(load balancing)* sin que el acceso a los recursos se vea afectado.

c) *Escalabilidad.* Parámetros importantes:

- Rendimiento (*throughput*): Número de operaciones por unidad de tiempo.
- Tiempo de respuesta (*response time*): Tiempo desde que el cliente solicita un servicio hasta que recibe la respuesta.
- Fiabilidad (*reliability*): Fiabilidad del sistema con respecto a la fiabilidad de los componentes.

Un sistema distribuido es escalable si su capacidad de procesamiento puede crecer añadiendo nodos adicionales:

- Aumenta el rendimiento con un número creciente de nodos (idealmente de forma lineal).
- El tiempo de respuesta decrece con un número creciente de nodos.
- La fiabilidad el sistema aumenta con número creciente de nodos (idealmente de forma logarítmica).

d) *Carencia de reloj global.* La transferencia de mensajes entre los componentes para la realización de una tarea, no tienen una temporización general, esta más bien distribuida a los componentes.

e) Tolerancia a Fallos.

- Los fallos en un sistema distribuido son principalmente parciales, es decir, los fallos parciales no deben afectar al sistema que debe seguir ofreciendo servicio, quizás en modo degradado (*graceful degradation*).

Las dos propiedades más importantes son:

- Disponibilidad *(availability):* Un recurso permanece disponible a pesar de que haya fallos.

- Atomicidad *(atomicity):* Un recurso mantiene su consistencia a pesar de que haya fallos.

f) *Fallos independientes de los componentes*. Cada componente del sistema puede fallar independientemente, con lo cual los demás pueden continuar ejecutando sus acciones. Esto permite el logro de las tareas con mayor efectividad, pues el sistema en su conjunto continua trabajando. Cada elemento de cómputo tiene su propia memoria y su propio sistema operativo.

g) *Control de recursos locales y remoto.* administra y aloja los recursos así como controla la ejecución de los programas de usuarios y las operaciones de los dispositivos de entrada/salida.

h) *Sistemas Abierto:* Facilidades de cambio tanto de equipo como de componentes del mismo; facilidad de crecimiento.

i) *Plataforma no estándar*. Es posible incorporar distintos tipos de plataformas como son Unix, NT, Intel, RISC, etc.

j) Capacidad de procesamiento en paralelo.

k) *Sistemas débilmente acoplados.* Cada procesador tiene su propia memoria local y el procesador se comunica con los demás procesadores mediante líneas de comunicación, buses de alta velocidad y líneas telefónicas.

Protocolos en los sistemas distribuidos

En un sistema distribuido los procesos están repartidos en distintas máquinas, e intercambian información mediante paso de mensajes sobre algún sistema de comunicación. Cuando se les compara con los sistemas tradicionales, "centralizados", surgen nuevos problemas, debidos a la propia distribución: retardos, particiones de la red, fallo independiente de máquinas, etc. Pero también hay nuevas posibilidades, como por ejemplo la replicación de procesos en diferentes máquinas para incrementar la tolerancia a fallos. Todo esto hace que el diseño y la implementación de sistemas distribuidos sea una tarea compleja y difícil. Así, es particularmente útil encontrar formas de simplificar los problemas, y proporcionar módulos o paradigmas probados y listos para usar, que resuelvan algún aspecto difícil de la construcción de sistemas distribuidos. Esto se aplica en particular a los protocolos de comunicación que se usan: dependiendo de las características que proporcione un protocolo, la construcción de algún tipo de sistema distribuido sobre él puede ser mucho más fácil.

Un protocolo de comunicación es un conjunto de reglas y formatos que se utilizan para la comunicación entre procesos que realizan una determinada tarea. Se requieren dos partes:

- Especificación de la secuencia de mensajes que se han de intercambiar.
- Especificación del formato de los datos en los mensajes.

Permite que componentes heterogéneos de sistemas distribuidos puedan desarrollarse independientemente, y por medio de módulos de software que componen el protocolo, haya una comunicación transparente entre ambos componentes. Estos componentes del protocolo deben estar tanto en el receptor como en el emisor. Entre los protocolos más utilizados en los sistemas distribuidos de acuerdo con el modelo OSI son:

- IP: Protocolo de Internet. Protocolo de la capa de Red, que permite definir la unidad básica de transferencia de datos y se encarga del direccionamiento de la información para que llegue a su destino.
- TCP: Protocolo de Control de Transmisión. Protocolo de la capa de Transporte, que permite dividir y ordenar la información a transportar en paquetes de menor tamaño para su transporte y recepción.
- HTTP: Protocolo de Transferencia de Hipertexto. Protocolo de la capa de Aplicación, que permite el servicio de transferencia de páginas de hipertexto entre el cliente Web y los servidores.
- SMTP: Protocolo de Transferencia de Correo Simple. Protocolo de la capa de Aplicación, que permite el envío de correo electrónico por la red.
- POP3: Protocolo de Oficina de Correo. Protocolo de la capa de aplicación, que permite a una estación de trabajo recuperar los correos que están almacenados en el servidor.

1.2.2 Sistemas de procesamiento distribuido

Kernels de Sistemas Distribuidos con mínima funcionalidad. Reciben el nombre de microkernels dado que proveen un conjunto casi mínimo de primitivas para la administración de procesos, manejo de memoria, intercambio de mensajes y la administración de la memoria de respaldo.

Sistemas Integrados. Estos tipos de sistemas permiten la ejecución de software estándar ampliamente configurable en cualquier computadora, y provee a cada máquina de las ayudas y servicios que necesita para tal función. Actualmente están basados fundamentalmente en el sistema operativo UNIX.

Sistemas Orientados a Objetos Distribuidos. En los sistemas Cliente/Servidor, un objeto distribuido es aquel que esta gestionado por un servidor y sus clientes invocan sus métodos utilizando un "método de invocación remota". El cliente invoca el método mediante un mensaje al servidor que gestiona el objeto, se ejecuta el método del objeto en el servidor y el resultado se devuelve al cliente en otro mensaje.

Sistemas Orientados a Base de Datos Distribuidas. Es una colección de datos (base de datos) construida sobre una red que pertenecen a un solo sistema distribuido.

- La información de la base de datos esta almacenada físicamente en diferentes sitios de la red.
- En cada sitio de la red, la parte de la información, se constituye como una base de datos en sí misma.

- Las bases de datos locales tienen sus propios usuarios locales, sus propios DBMS y programas para la administración de transacciones, y su propio administrador local de comunicación de datos.

Sistemas Basados en el Modelo banco de servidores. Utilizan un servidor con los servicios disponibles para cada usuario.

Ventajas de los sistemas distribuidos

Con respecto a los sistemas centralizados se tiene:

- Economía, pues es mucho más barato, añadir servidores y clientes cuando se requiere aumentar la potencia de procesamiento.
- El trabajo en conjunto. Por ejemplo: en una fábrica de ensamblado, los robots tienen sus CPU's diferentes y realizan acciones en conjunto, dirigidos por un sistema distribuido.
- Tienen una mayor confiabilidad. Al estar distribuida la carga de trabajo en muchas máquinas la falla de una de ellas no afecta a las demás, el sistema sobrevive como un todo.
- Capacidad de crecimiento incremental. Se puede añadir procesadores al sistema incrementando su potencia en forma gradual según sus necesidades.

Con respecto a PC's Independientes se tiene:

- Se pueden compartir recursos, como programas y periféricos, muy costosos. Ejemplo: Impresora Láser, dispositivos de almacenamiento masivo, etc.
- Al compartir recursos, satisfacen las necesidades de muchos usuarios a la vez. Ejemplo: Sistemas de reservas de aerolíneas.
- Se logra una mejor comunicación entre las personas. Ejemplo: el correo electrónico.
- Tienen mayor flexibilidad**,** la carga de trabajo se puede distribuir entre diferentes computadoras.

Desventajas de los sistemas distribuidos

- El principal problema es el software, el diseño, implantación y uso del software distribuido, pues presenta numerosos inconvenientes. Las principales interrogantes son los siguientes: ¿Qué tipo de S. O., lenguaje de programación y aplicaciones son adecuados para estos sistemas?, ¿Cuánto deben saber los usuarios de la distribución?, ¿Qué tanto debe hacer el sistema y qué tanto deben hacer los usuarios?
- Otro problema tiene que ver con las redes de comunicación: pérdida de mensajes, saturación en el tráfico, etc.
- Un problema que puede surgir al compartir datos es la seguridad de los mismos.

En general se considera que las ventajas superan a las desventajas, si estas últimas se administran seriamente.

Aplicaciones de los sistemas distribuidos

- Servicios Comerciales: Sistemas de reservas de líneas aéreas, aplicaciones bancarias, cajas y gestión de grandes almacenes.
- Redes WAN: Correo electrónico, servicio de noticias, transferencia de archivos, World Wide Web.
- Aplicaciones Multimedia: videoconferencia, televigilancia, juegos multiusuarios, enseñanza asistida por computadora.
- Áreas de la informática aplicada a los Sistemas Distribuidos. Comunicaciones, sistemas operativos distribuidos, Base de datos distribuidas, servidores distribuidos de archivos, lenguajes de programación distribuidos, sistemas de tolerancia de fallos.

1.2.3 Asociación con arquitecturas paralelas (Flynn) y modelos de red.

No existe una clasificación general que sea totalmente aceptada para poder ordenar los ambientes de cómputo existentes y dentro de ellos, a los sistemas distribuidos en particular pero caen dentro de la clasificación de *Flynn* fundamentada en la multiplicidad de los flujos de instrucciones y de los flujos datos dentro del sistema como ya se describió anteriormente.

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

- SMP.
- MPP.
- Arquitectura ccNUMA.
- Sistemas Distribuidos.
- Clusters.

SMP (*Symmetric Multiprocessors).* En el Multiprocesamiento Simétrico se tiene un diseño simple, efectivo y económico. En SMP, muchos procesadores comparten las misma memoria RAM y el bus del sistema. La presencia de un solo espacio de memoria simplifica tanto el diseño del sistema físico (hardware) como la programación de las aplicaciones (software). Esa memora compartida permite que un sistema operativo con multiconexión distribuya las tareas entre varios procesadores, o que una aplicación obtenga toda la memora que necesita para una simulación compleja. La memoria globalmente compartida también vuelve fácil la sincronización de datos.

SMP es uno de los diseños de procesamiento paralelo más maduros. Sin embargo, la memoria global contribuye al problema más grande de SMP: conforme se añaden procesadores, el tráfico en el bus de memoria se satura. Al añadir memoria caché a cada procesador se puede reducir algo del tráfico en el bus. Al manejarse ocho o más procesadores, el cuellos de botella se vuelve crítico, inclusive para los mejores diseños, por los que SMP es considerada una tecnología poco escalable.

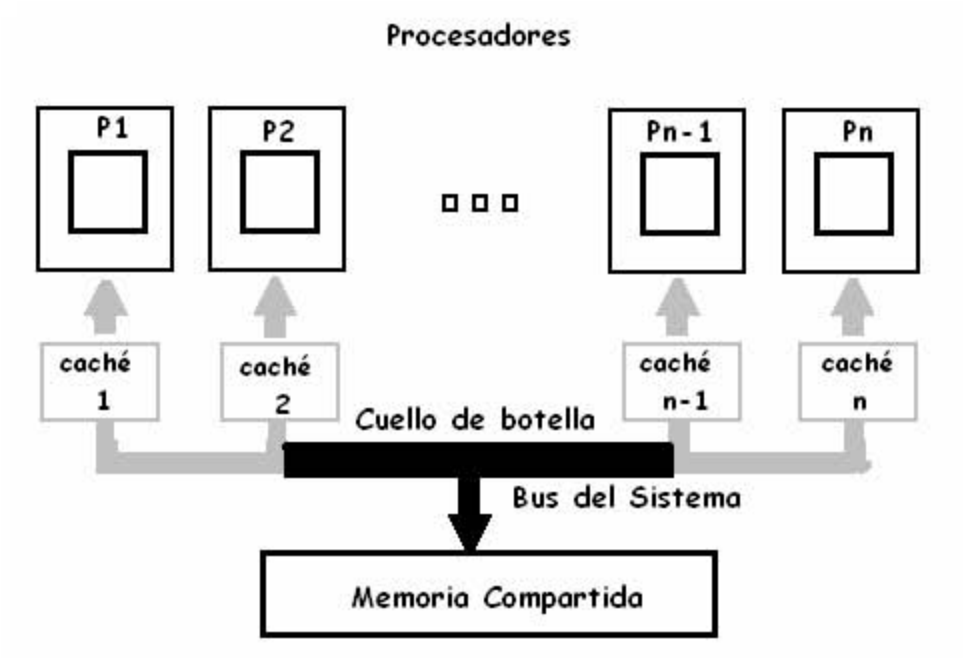


Figura 1.8 Multiprocesamiento Simétrico (SMP)

MPP (*Massively Parallel Processor*). Las computadoras MPPs (Procesadores Masivamente Paralelos), son actualmente las más poderosas en el mundo. Ésta máquina combina desde algunas decenas, hasta algunas cientas de CPUs en un único gabinete con cientos de Gigabytes de memoria. Los MPPs ofrecen un enorme poder de cómputo y son comúnmente usados para ejecutar enormes cálculos, resolviendo problemas que constituyen verdaderos retos computacionales.

El costo de las grandes MPPs es típicamente mayor a 1 millón de dólares, y en contraste, los usuarios del cómputo distribuido observan un costo mucho menor al usar un conjunto de computadoras con las que ya cuentan para solucionar sus problemas, si bien no es común para los usuarios del cómputo de las grandes MPPs, por lo menos sí son capaces de resolver problemas varias veces más complejos de los que pueden solucionarse usando una sola de sus computadoras de propósito general.

Para evitar los cuellos de botella en el bus, MPP no utiliza memoria compartida, en su lugar, distribuye equitativamente la memoria RAM entre los procesadores de modo que se asemeja a una red (cada procesador con su memoria distribuida asociada).

Para tener acceso a las áreas de memoria fuera de su propia RAM, los procesadores utilizan un esquema de paso de mensajes análogo a los paquetes de datos en redes. Este sistema reduce el tráfico del bus, debido a que cada sección de memoria interactúa únicamente con aquellos accesos que le están destinados en lugar de interactuar con todos los accesos a memoria, como ocurre con un sistema SMP. Esto permite la construcción de sistemas MPP de grana tamaño con cientos y aún miles de procesadores por lo que MPP es una tecnología altamente escalable.

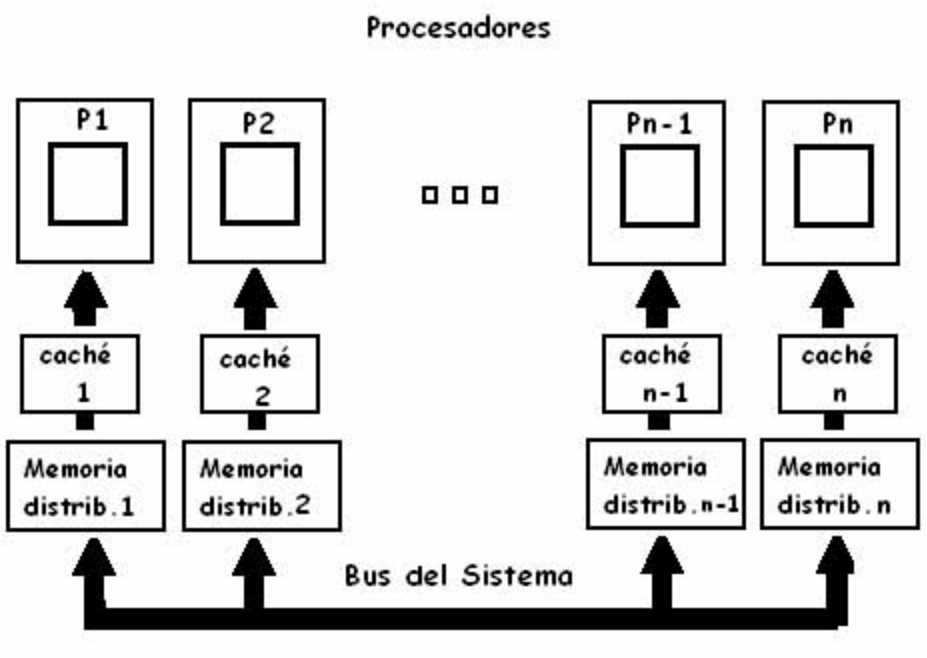


Figura 1.9 Procesadores Masivamente Paralelos (MPP)

La parte negativa de MPP es que la programación se vuelve difícil, debido a que la memoria se rompe en pequeños espacios separados. Sin la existencia de un espacio de memoria globalmente compartido, ejecutar una aplicación que requiere una gran cantidad de RAM, que puede ser difícil. La sincronización de datos también se complica, particularmente si un mensaje debe pasar por muchos componentes de hardware hasta alcanzar la memoria del procesador destino.

Arquitectura ccNUMA (*Caché Coherent Non uniform 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.

Las ventajas de esta arquitectura son:

- 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 multiprocesamiento en configuraciones de cierto número de CPUs.
- Distribución de 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 un 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 para procesar grandes cantidades de datos. Otro tipo de cluster es el de base de datos llamado libre de falla o alta disponibilidad, que se ocupa para balancear bases de datos las cuales deben estar siempre en funcionamiento.

La figura 1.10 muestra la relación entre las arquitecturas paralelas.

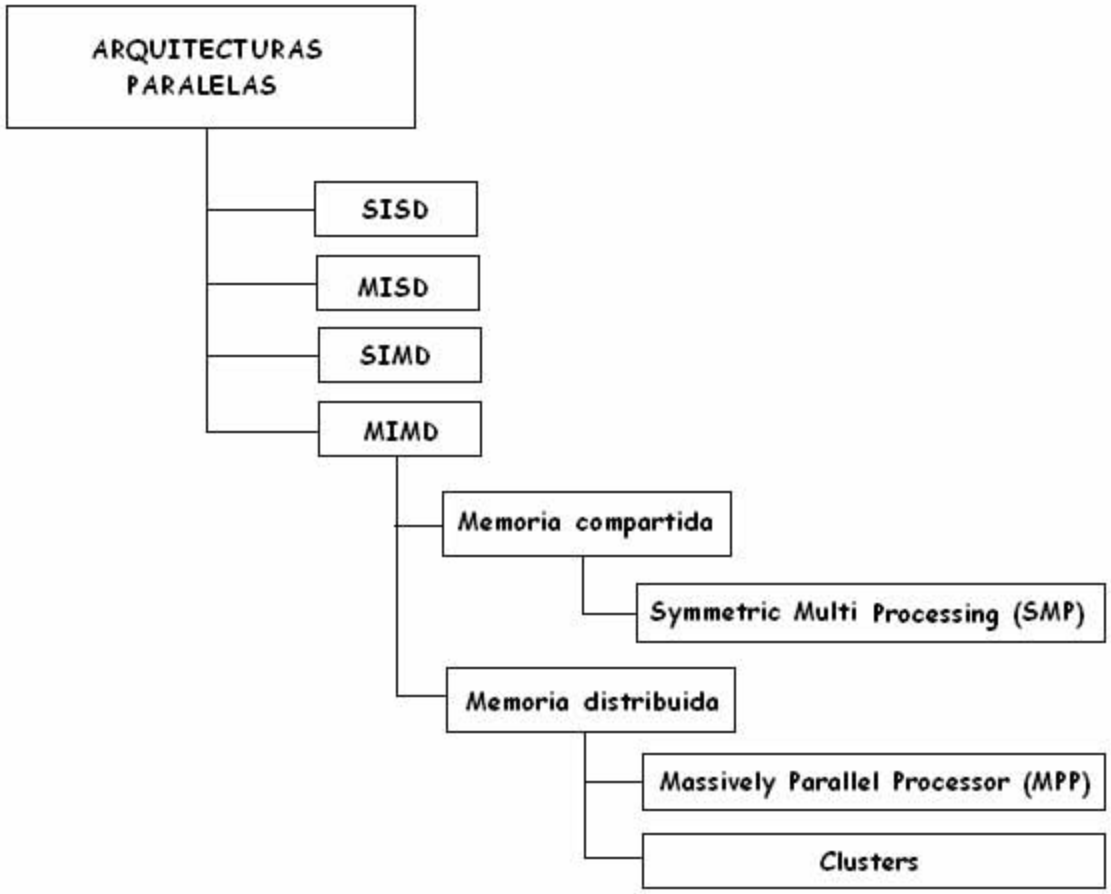


Figura 1.10 Arquitecturas paralelas

Modelos de red

Existen varios modelos fundamentales que se usan para la creación de sistemas de cómputo distribuido, los cuales se clasifican en diferentes categorías:

- Modelo de minicomputadora.
- Modelo de estaciones de trabajo.
- Modelo estación de trabajo – servidor.
- Modelo de banco de procesadores.
- Modelo híbrido.

Modelo de Minicomputadora

En 1960 surgió la minicomputadora, una versión más pequeña de la macrocomputadora. Al ser orientada a tareas específicas, no necesita de muchos periféricos como los que necesita un mainframe, y esto ayudó a reducir el precio y costos de mantenimiento.
Las macrocomputadoras son también conocidas como mainframes. Los mainframes son grandes, rápidos y caros, capaces de controlar cientos de usuarios simultáneamente, así como cientos de dispositivos de entrada y salida. Los mainframes tienen un costo que va desde 350,000 dólares hasta varios millones de dólares. De alguna forma los mainframes son más poderosos que las supercomputadoras porque soportan más programas simultáneamente, pero las supercomputadoras pueden ejecutar un solo programa más rápido que un mainframe.

En el pasado, los mainframes ocupaban cuartos completos o hasta pisos enteros de algún edificio, hoy en día, un mainframe es parecido a una hilera de archiveros en algún cuarto con piso falso, esto para ocultar los cientos de cables de los periféricos, y su temperatura tiene que estar controlada.

Las minicomputadoras, en tamaño y poder de procesamiento, se encuentran entre los mainframes y las Estaciones de trabajo, y por lo tanto, es frecuentemente referida como de rango medio (*midrange*). En general, una minicomputadora, es un sistema multiproceso (varios procesos en paralelo) capaz de soportar de 10 hasta 200 usuarios simultáneamente. Actualmente se usan para almacenar grandes bases de datos, automatización industrial y aplicaciones multiusuario.

El modelos de minicomputadora es simplemente la extensión del modelo de equipo centralizado. Este esquema consiste de varias minicomputadoras conectadas por una red de comunicación. Cada minicomputadora tiene su propio grupo de usuarios quienes pueden tener acceso a otros recursos no presentes en su sistema a través de la red de datos. Este modelo se usa cuando existe la necesidad de compartir recursos de diferentes tipos a través de acceso remoto.

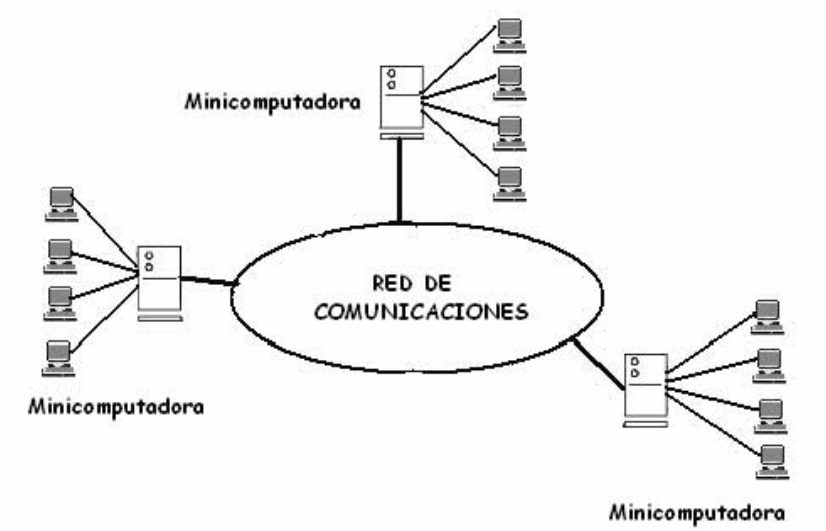


Figura 1.11 Modelo de Minicomputadora

Modelo de Estación de Trabajo

Las estaciones de trabajo son un tipo de computadoras que se utilizan para aplicaciones que requieran de poder de procesamiento moderado y relativamente capacidades de gráficos de alta calidad. Son usadas para:

- Aplicaciones de Ingeniería
- CAD (Diseño asistido por Computadora)
- CAM (Manufactura Asistida por Computadora)
- Publicidad
- Creación de Software

El modelo de Estación de Trabajo, consiste en varias Estaciones de Trabajo interconectadas por una red. Cada usuario se "da de alta" en su computadora de inicio o "computadora *home*" y luego desde ahí puede enviar trabajos para ejecución. Cada estación tiene su propio disco duro, y su propio sistema de archivos. La implementación mostrada tiene la desventaja de que se puede desperdiciar tiempo de procesamiento si una o varias computadoras no están haciendo uso de su CPU. Si el sistema determina que la estación del usuario no posee la suficiente capacidad de proceso para una tarea, transfiere el trabajo de forma automática hacia el equipo que tiene menos actividad en ese momento y por último regresa la salida del trabajo a la estación del usuario.

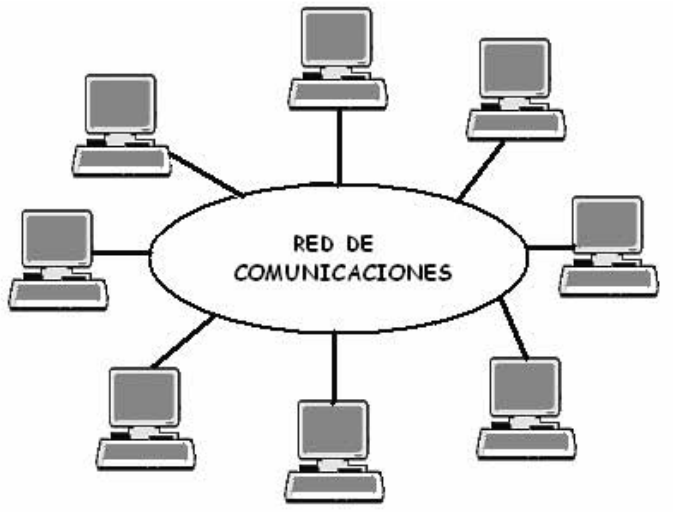


Figura 1.12 Modelo de Estación de Trabajo

Modelo Estación de Trabajo – Servidor

Las microcomputadoras o Computadoras Personales (PC´s) tuvieron su origen con la creación de los microprocesadores. Un microprocesador es "una computadora en un chip", o sea un circuito integrado independiente. Las PC´s son computadoras para uso personal y relativamente son baratas, actualmente se encuentran en las oficinas, escuelas y hogares.
El término PC se deriva de que para el año de 1981 , IBM, sacó a la venta su modelo "IBM PC", la cual se convirtió en un tipo de computadora ideal para uso "personal", de ahí que el término "PC" se estandarizó y los clones que sacaron posteriormente otras empresas fueron llamados "PC y compatibles", usando procesadores del mismo tipo que las IBM , pero a un costo menor y pudiendo ejecutar el mismo tipo de programas.

Existen otros tipos de microcomputadoras , como la Macintosh, que no son compatibles con la IBM, pero que en muchos de los casos se les llaman también "PC´s", por ser de uso personal.

En la actualidad existen variados tipos en el diseño de PC´s:

a) Computadoras personales, con el gabinete tipo minitorre, separado del monitor.
b) Computadoras personales portátiles *Laptop* o *Notebook*. Las computadoras *laptops* son aquellas computadoras que están diseñadas para poder ser transportadas de un lugar a otro. Se alimentan por medio de baterías recargables , pesan entre 2 y 5 kilos y la mayoría trae integrado una pantalla de LCD (*Liquid Crystal Display*).
c) Computadoras personales más comunes, con el gabinete horizontal, separado del monitor.
d) Computadoras personales que están en una sola unidad compacta el monitor y la CPU.
e) Estaciones de trabajo o *Workstations*.

El modelo de estación de Trabajo-Servidor, posee varias microcomputadoras y varias estaciones de trabajo (algunas de las cuales pueden ser estaciones sin disco o *diskless*) comunicadas

mediante red. Ahora que existe la potencialidad de tener estaciones sin disco, el sistema de archivos a usar por estas computadoras puede ser el de alguna computadora completa o alguno compartido de las diferentes minicomputadoras del sistema. Algunas otras microcomputadoras se pueden usar para proveer otros tipos de servicios, como base de datos, impresión, etc. Estas máquinas cumplen ahora nuevas tareas especializadas para proveer y administrar acceso a los recursos, a los que se les llama servidores (*servers*).

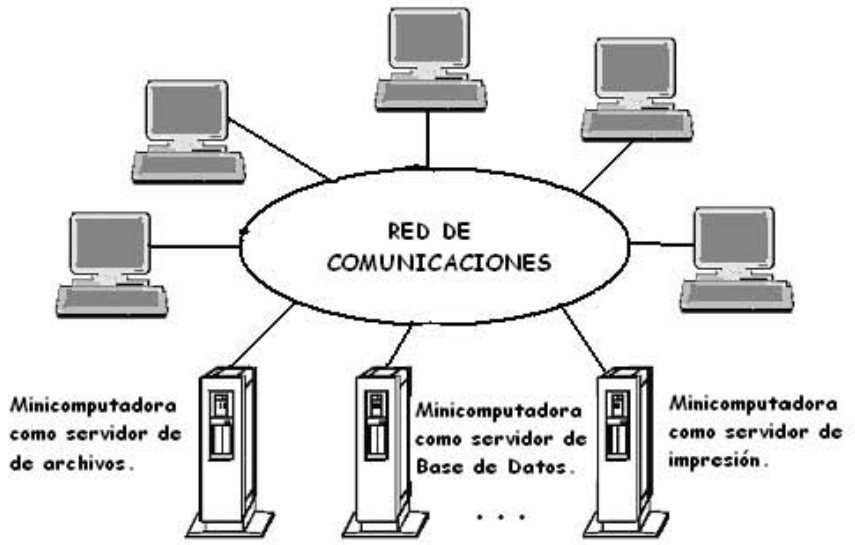


Figura 1.13 Modelo de Estación de Trabajo-Servidor

Por razones de escalabilidad y confiabilidad, es común distribuir un recurso entre varios servidores, lo cual incrementa la confiabilidad y rapidez en la solución de fallas en caso de algún mal funcionamiento de alguno de ellos. Por ejemplo, el servicio de correo electrónico puede tenerse en diferentes equipos del conjunto, de manera tal que si uno fallara momentáneamente, fuese posible para todos los demás usuarios continuar trabajando con dicho servicio a través de los equipos redundantes.

En este modelo el usuario se da de alta normalmente en su computadora y para ciertas tareas utiliza los recursos locales. Cuando necesita otro servicio que no tiene instalado, las peticiones de aplicaciones específicas se encaminan hacia los servidores, quienes procesan las solicitudes y regresan a usuario el archivo o programa solicitado. Otra característica es que en éste modelo los procesos de usuario no necesitan ser migrados hacia los servidores para obtener los beneficios del procesamiento adecuado.

Como observaciones al modelo, se tiene:

1. En general es más barato usar unas pocas minicomputadoras equipadas con grandes discos de alta velocidad que tener a todas las estaciones de trabajo con un disco duro instalado más lento y más pequeño.
2. Desde el punto de vista de mantenimiento, es mejor tener estaciones sin disco que todo el sistema completo.

3. El usuario tiene en general la libertad de elegir en cuál estación trabajar, a diferencia del modelo de estación de trabajo, donde tiene su cuenta y clave de acceso.
4. Su operación se basa en el paradigma cliente-servidor, donde un proceso cliente envía una petición de procesamiento al servidor. El servidor ejecuta la petición y regresa a la estación el resultado de la misma.

Modelo de Banco de Procesadores

Éste se basa en el hecho de que los usuarios en promedio no requieren capacidad de procesamiento durante un buen rato, pero existen instantes en los que la actividad y los programas que ejecutan demandan potencia de trabajo en alto grado. A diferencia del modelo anterior en el que cada persona tiene su servidor asignado, en éste se dispone de un conjunto de servidores que son compartidos y asignados conforme a demanda. Cada procesador en el banco tiene su propia memoria y ejecuta un programa de sistema o de aplicación que le permite participar en el ambiente de cómputo distribuido.

En la figura 1.14 se puede apreciar que el modelo no soporta la conexión de estaciones directamente a los servidores, sino sólo por medio de la red de interconexión. Las terminales usadas pueden ser estaciones sin disco o terminales gráficas. Se tiene instalado un servidor especial llamado “de ejecución” el cual se dedica a la administración y asignación de recursos conforme a la demanda. Cuando se recibe una solicitud de una persona, este equipo asigna temporalmente el número de servidores que deberán ser dedicados al trabajo de tal petición, y luego de atenderla, regresan para ser liberados y quedar a disposición de la siguiente tarea. Otra diferencia importante con otros esquemas, es que aquí no existe el concepto de “computadora principal”, de manera que el usuario no se da de alta en alguna máquina en particular, pero sí percibe al sistema como un todo.

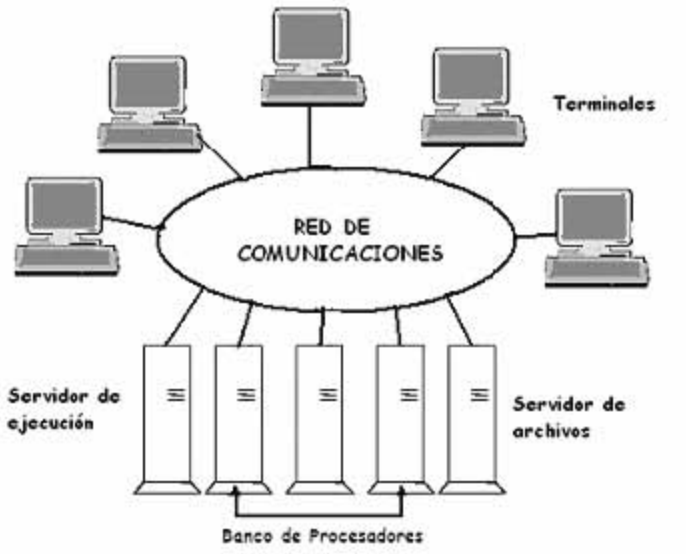


Figura 1.14 Modelo de Banco de Procesadores

Comparado con el modelo de Estación de Trabajo-Servidor, éste permite una mejor utilización del poder de cómputo disponible en el ambiente distribuido, dado que dicho poder computacional está disponible para todos a diferencia de Estación de Trabajo-Servidor, donde varios equipos pueden estar desocupados en algún momento pero su capacidad no se puede asignar a otros. Por otra parte, esta metodología provee gran flexibilidad ya que el sistema se puede expandir agregando procesadores que operarán como servidores adicionales para soportar una carga extra de trabajo originada por incremento en el número de usuarios o por la implantación de nuevos servicios.

Sin embargo, este modelo no se considera recomendado para programas de cálculo complejo como aplicaciones intensivas de alto rendimiento o aplicaciones basadas en gráficas. Esto se debe principalmente a la lentitud de las redes de conexión que se utilizan para la comunicación entre procesos. Para este tipo de tareas se considera mejor el modelo Estación de Trabajo-Servidor.

Modelo híbrido

De los cuatro modelos descritos anteriormente, el de Estación de Trabajo-Servidor es el más ampliamente utilizado para la construcción de sistemas de cómputo distribuido. Esto se debe a que un usuario promedio solamente ejecuta tareas interactivas simples como editar archivos, enviar correos electrónicos, etc. El modelo se ajusta perfectamente a especificaciones tan sencillas. Para ambientes donde hay grupos de usuarios de mayor potencia que realizan tareas masivas con alta necesidad de poder computacional el modelo Banco de Procesadores es más adecuado.

El modelo híbrido permite combinar las mejores características de Estación de Trabajo-Servidor y Banco de Procesadores, esencialmente agregando a la red de estaciones de trabajo un Banco de servidores que pueden ser asignados dinámicamente para trabajos que son muy extensos para máquinas individuales o que necesitan varios equipos concurrentemente para una ejecución adecuada. Esta variante tiene la ventaja de garantizar el nivel de respuesta a trabajos interactivos dado que permite la ejecución en la misma computadora de la persona que lo solicita. Por otra parte, su principal desventaja estriba en que el coste de implantación se eleva puesto que requiere mayor número de componentes para su construcción.

1.3 Cómputo paralelo/ distribuido en clusters

1.3.1 Clusters

Los cluster son agrupaciones de PC's unidos por una red de alta velocidad. Es un conjunto de computadoras las cuales trabajan en conjunto para resolver una tarea. Un cluster está formado por varias computadoras las cuales se comunican por medio de una conexión a red, trabajando en un proceso, el cual sería muy largo para una sola computadora, ejecutándolo en un tiempo razonable.

Clasificación de los clusters

1. *Alta disponibilidad (high availability, HA) o "cluster de redundancia".*

Un cluster de alta disponibilidad es un conjunto de dos o más máquinas, que se caracterizan porque comparten los discos de almacenamiento de datos, y porque están constantemente monitorizándose entre sí. Si se produce un fallo del hardware o de las aplicaciones de alguna de las máquinas del cluster, el software de alta disponibilidad es capaz de rearrancar automáticamente los servicios que han fallado en cualquiera de las otras máquinas del cluster (debido a que existen servidores que actúan entre ellos como respaldos vivos de la información que sirven). Y cuando la máquina que ha fallado se recupera, los servicios son nuevamente migrados a la máquina original. Esta capacidad de recuperación automática de servicios nos garantiza la integridad de la información, ya que no hay pérdida de datos, y además evita molestias a los usuarios, que no tienen por qué notar que se ha producido un problema.

2. *Alto rendimiento (high performance, HP).*

La idea detrás del concepto de "cluster de alto rendimiento" es hacer que un número grande de máquinas individuales actúen como una sola máquina muy potente. Este tipo de clusters se aplica en problemas grandes y complejos que requieren una cantidad enorme de potencia computacional. Entre las aplicaciones más comunes de clusters de alto rendimiento se encuentra el pronóstico numérico del estado del tiempo, astronomía, investigación en criptografía, análisis de imágenes, y más.

Su finalidad es disponer del máximo número de *flops*, unidad de medida para las operaciones de punto flotante realizadas por segundo de tiempo.

No hay que confundir los clusters de alta disponibilidad con los clusters de alto rendimiento. Un cluster de alto rendimiento es una configuración de equipos diseñada para proporcionar capacidades de cálculo mucho mayores que la que proporcionan los equipos individuales, mientras que los clusters de alta disponibilidad están diseñados para garantizar el funcionamiento ininterrumpido de ciertas aplicaciones.

3. *Balanceo de carga (load balancing) o "cluster de servidores virtuales".*

Existe una jerarquía de maestros y esclavos. Los maestros reciben la petición del servicio y mandan la tarea a los esclavos para que la ejecuten. Éstos devuelven el resultado y el servidor responde.

Este tipo de cluster permite que un conjunto de servidores de red compartan la carga de trabajo de tráfico de sus clientes. Al balancear la carga de trabajo de tráfico en un arreglo de servidores, mejora el tiempo de acceso y confiabilidad. Además como es un conjunto de servidores el que atiende el trabajo, la falla de uno de ellos no ocasiona una falla catastrófica total. Este tipo de servicio es de gran valor para compañías que experimentan grandes volúmenes de tráfico en sus sitios Web.

Además del concepto de Cluster, existe otro concepto más amplio y general que es el Cómputo en Malla (*Grid Computing*). Una Malla es un tipo de sistema paralelo y distribuido que permite compartir, seleccionar y añadir recursos que se encentran distribuidos a lo largo de dominios administrativos "múltiples" basados en su disponibilidad, capacidad, rendimiento, costo y calidad de servicio que requiere un usuario.

Si los recursos distribuidos se encuentran bajo la administración de un sistema central único de programación de tareas, entonces nos referiremos a un Cluster. En un Cluster, todos los nodos trabajan en cooperación con un objetivo y una meta común pues la asignación de recursos los ejecuta un solo administrador centralizado y global. En una Malla, cada nodo tiene su propio administrador de recursos y política de asignación.

Configuración de los clusters

Los clusters se presentan en tres tipos de configuración: activo/activo, activo/standby, y tolerante a fallos.

- *Activo/activo*: Todos los nodos en el cluster realizan un trabajo significativo. Si algún nodo cae, el nodo restante (o nodos restantes) continúan realizando su trabajo y además el trabajo del nodo que ha caído. El tiempo de recuperación está entre 15 y 90 segundos.

- *Activo/standby*: Un nodo (el nodo primario) realiza trabajo, y el otro espera a que suceda una caída del nodo primario. Si el primer nodo falla, la solución *clustering* transfiere el trabajo del nodo primario al nodo *standby* y finaliza la sesión de los usuarios o cualquier trabajo en el nodo *standby*. El tiempo de recuperación está entre los 15 y los 90 segundos.

- *Tolerante a Fallos*: Un cluster tolerante a fallos es un sistema completamente redundante (disco y CPU) cuyo objetivo es estar disponible el 99.999% del tiempo. Este objetivo se traduce en menos de 6 minutos fuera de servicio por año. Ambos nodos del cluster tolerante a fallos realizan simultáneamente tareas idénticas; el trabajo de los nodos es redundante. El tiempo de recuperación es menos de 1 segundo.

1.3.2 Características de los sistemas paralelos / distribuidos en clusters

Transparencia

El ambiente debe volver “invisibles” (transparentes) a todas las máquinas que lo integran y con las cuales trabajan las personas. El requisito es que cada persona trabajando con éste conjunto de máquinas conectadas por una red de comunicaciones “sientan” que están trabajando con un único procesador.

Tipos de transparencia

- *Transparencia de acceso*. El usuario no distingue si un recurso es remoto o local, es decir, si utiliza un recurso debe ser un mecanismo igual sin importar la localización física del ente.
- *Transparencia de localización*. Los recursos tienen nombre únicos dentro del sistema y no están vinculados con una localización física de los mismos; por lo tanto el usuario puede acceder a ellos desde cualquier punto.
- *Transparencia de replicación.* El sistema crea réplicas de los procesos tanto de los archivos como de los recursos en diferentes nodos del sistema de forma automática a fin de lograr un mejor desempeño y confiabilidad.
- *Transparencia a fallas*. El sistema deberá continuar operando aún en el caso de fallas en alguno de su componentes.
- *Transparencia de migración*. Los recursos pueden ser cambiados de localización sin alterar su nombre.
- *Transparencia de concurrencia*. El sistema provee la facilidad de operación tal para compartir sus recursos que cada usuario no percibe la presencia de otros que también están haciendo uso de los mismos.
- *Transparencia de desempeño*. La capacidad de procesamiento del sistema debe ser uniformemente distribuida, es decir, existe un balanceo y una asignación de cargas automáticamente.
- *Transparencia de escalamiento*. Propiedad que permite expandir en escala al sistema sin interrumpir los trabajos de usuarios.

Confiabilidad

Se refiere a la capacidad que tiene un sistema de funcionar a pesar de la existencia de varios problemas, es decir, sigue funcionando aunque fallen algunos de sus componentes. A estos sistemas se les llama sistemas tolerantes a fallas.
Un aspecto clave de la tolerancia a fallas es la redundancia. Si falla un componente, otro equivalente toma el relevo. Algunos sistemas emplean una estrategia de mancomunidad, en la cual recursos idénticos funcionan en paralelo (redundancia de hardware). Otros recursos activan recursos redundantes sólo cuando hay una falla.

Los sistemas tolerantes a fallas ofrecen una degradación paulatina; cuando fallan componentes en un sistema así, éste sigue dando servicio, pero a niveles reducidos. Los sistemas tolerantes a fallas están diseñados para poder desconectar un componente, repararlo y volverlo a conectar, todo mientras el sistema sigue dando servicio ininterrumpido.

Flexibilidad

Esta característica se refleja principalmente en la facilidad de modificación, es decir poder reemplazar componentes y en la expansión del sistema, poder añadir componentes.

Escalabilidad

Se refiere a la capacidad del sistema para adaptarse a una demanda incrementada de servicio sin causar interrupción a los demás.

1.3.3 Cluster de alto rendimiento tipo Beowulf

Breve historia

En los últimos años, el personal académico de diversas universidades y centros de investigación se han dado a la tarea de aprender a construir sus propias supercomputadoras conectando computadoras personales y desarrollando software para enfrentar tales problemas extraordinarios. En 1994, se integró el primer cluster de PCs en el Centro de Vuelos Espaciales Goddard de la NASA, para resolver problemas computacionales que aparecen en las ciencias de la Tierra y el Espacio. Los pioneros de este proyecto fueron Thomas Sterling, Donald Becker y otros científicos de la NASA. El cluster de PCs desarrollado tuvo una eficiencia de 70 megaflops (millones de operaciones de punto flotante por segundo). Los investigadores de la NASA le dieron el nombre de *Beowulf* a este cluster, en honor del héroe de las leyendas medievales, quien derrotó al monstruo gigante Grendel.

En 1996, hubo también otros dos sucesores del proyecto Beowulf de la NASA. Uno de ellos es el proyecto *Hyglac* desarrollado por investigadores del Instituto Tecnológico de California (CalTech) y el Laboratorio de Propulsión Jet (JPL), y el otro, el proyecto *Loki* construido en el Laboratorio Nacional de Los Alamos, en Nuevo México. Cada cluster se integró con 16 microprocesadores Intel Pentium Pro y tuvieron un rendimiento sostenido de más de un gigaflop con un costo menor a $50,000 dólares.

En 1996, en el Laboratorio Nacional de Oak Ridge en Tennessee, se enfrentaban al problema de elaboración de un mapa de las condiciones ambientales del territorio de Estados Unidos. El territorio fue dividido en 7.8 millones de celdas de 1Km. Cada celda contenía la información de 25 variables, desde la precipitación promedio mensual hasta el contenido de Nitrógeno del suelo. Ninguna estación de trabajo o PC podría con esta tarea. Se requería una supercomputadora de

procesamiento paralelo. En la actualidad cuentan con un cluster de 130 PCs para trabajar en la elaboración del mapa de eco regiones.

Definición

Un Cluster tipo Beowulf es una conexión 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. 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), el cual controla el cluster completo y proporciona archivos a los nodos cliente. La programación en un Beowulf comúnmente es realizada por medio del paso de mensajes utilizando lenguajes como C y FORTRAN.

Ventajas

- Una de las ventajas obvias de un cluster tipo Beowulf es su economía, ya que está constituido por computadoras personales, componentes comerciales y además en la mayoría de los casos utiliza un sistema operativo de código libre.
- Como bien sabemos, la construcción de una supercomputadora de varios procesadores es complejo y costoso. Sin embargo, crear una red con 16,100 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 usarlas individualmente o sincronizarlos de tal forma que sean vistos ante el usuario como un único sistema de cómputo de mayor capacidad de procesamiento.
- Un Beowulf, por lo general, tienen 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 algunos archivos en el nodo servidor que darán 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.
- Los cluster 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 protocolo de las redes LAN, es posible tener una rápida interconexión entre computadoras y a bajo precio.

- 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 es relativamente sencillo de resolver ya que, si por ejemplo se estropea un procesador en un nodo, nos bastará con sustituirlo por otro. Esto es importante tenerlo en cuenta ya que es infinitamente más sencillo encontrar un nuevo procesador que cualquiera de los componentes de una compleja supercomputadora.
- 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 es porque ninguno de los clientes está directamente conectado al mundo exterior y todos los nodos son prácticamente iguales. Así mismo, el servidor puede confiar 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 modo servidor primero.
- 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 con su labor empleando sólo a los nodos disponibles.

Desventajas

- Es un poco complicada su instalación y su administración, los cluster 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 parte no, en qué momento deben comunicarse los procesadores y cómo deben reunirse los resultados individualmente obtenidos para conseguir un resultado global.
- Las redes de computadoras no están especializadas para el procesamiento en paralelo 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 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. Afortunadamente existen programas que realizan el equilibrio de carga de forma dinámica y sin que el usuario tenga que preocuparse de ello.

CAPÍTULO 2

METODOLOGÍA ORIENTADA AL PROCESAMIENTO PARALELO/DISTRIBUIDO

A pesar de los avances tecnológicos conseguidos en los últimos años, la tecnología del silicio está llegando a su límite. Si se quieren resolver problemas más complejos y de mayores dimensiones se deben buscar nuevas alternativas tecnológicas. Una de estas alternativas en desarrollo es el paralelismo. Mediante el paralelismo se pretende conseguir la distribución del trabajo entre las diversas CPU, que conformen una máquina paralela como es el caso de los cluster, de forma que realice el trabajo simultáneamente, con el objetivo de aumentar considerablemente el rendimiento total.

Para poder ejecutar una aplicación en paralelo en varias CPUs, debe ser explícitamente dividida en partes concurrentes. Una aplicación normal diseñada para una única CPU no se ejecutará más rápido en un sistema multiprocesador. Existen varias utilidades y compiladores que pueden dividir los programas, pero paralelizar el código no es una operación sencilla. Dependiendo de la aplicación, paralelizar el código puede ser fácil, muy difícil y, en algunos casos, imposible debido a las dependencias del algoritmo.

Programación Concurrente

El término concurrente es aplicable a cualquier programa que presente un comportamiento paralelo actual o potencial (viable). En cambio el término paralelo o distribuido es aplicable a aquel programa diseñado para su ejecución en un entorno específico. Cuando se emplea un solo procesador para la ejecución de programas concurrentes se habla de seudoparalelismo.

Programación concurrente es el nombre dado a las notaciones y técnicas empleadas para expresar el paralelismo potencial (viable) y para resolver los problemas de comunicación y sincronización resultantes. La programación concurrente proporciona una abstracción sobre el estudio del paralelismo sin tener en cuenta los detalles de implementación. Esta abstracción ha demostrado ser muy útil en la escritura de programas claros y correctos empleando las facilidades de los lenguajes de programación modernos.

El problema básico en la escritura de un programa concurrente es identificar qué actividades pueden realizarse concurrentemente. Además, la programación concurrente es mucho más difícil que la programación secuencial clásica por la dificultad de asegurar que el programa concurrente es correcto.

2.1 Niveles de paralelismo

Por lo general la computadora se visualiza como una máquina secuencial, en la cual la CPU lee la instrucción de la memoria y la va ejecutando una por una. El programador sigue ese mismo enfoque al especificar algoritmos como una secuencia de instrucciones para que las ejecute la computadora.

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

En computadoras modernas de tipo SISD el paralelismo viene implementado en la arquitectura del procesador, conocido como paralelismo a nivel instrucción, por ejemplo: *pipeline* o procesamiento en línea de instrucciones. Consiste en ejecutar al mismo tiempo en diversas etapas las instrucciones del programa; mientras en una etapa se hace la ejecución de una instrucción, simultáneamente en otra etapa se está realizando una lectura de la siguiente instrucción, es un esquema muy similar al de una línea de ensamble de autos.

Otra característica de procesadores recientes es que poseen varias ALUs (Unidad Aritmética Lógica) para realizar operaciones de suma, resta, multiplicación y división en forma paralela.

El paralelismo a nivel de programación resulta algunas veces complicado y otras veces sencillo dependiendo de la arquitectura de la supercomputadora, del modelo o técnica empleada para programar en paralelo e inclusive del programa en sí.

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 son:

- Paralelismo a nivel de tarea o trabajo.
- Paralelismo a nivel de programa.
- Paralelismo a nivel de instrucción.
- Paralelismo a nivel de bit.

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 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 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.

2.1.1 Granularidad

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

Granularidad es el tamaño de las piezas en que se divide una aplicación. Dichas piezas pueden ser una sentencia de código, una función o un proceso en sí que se ejecutarán en paralelo. El término granularidad se usa como el mínimo componente del sistema que puede ser preparado para ejecutarse de manera 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.

La granularidad se divide en grano fino, grano medio y grano grueso.

- De *grano fino:* representa el más complejo nivel de paralelismo. Es cuando el código se divide en una gran cantidad de piezas pequeñas. Es a un nivel de sentencia donde un ciclo se divide en varios subciclos que se ejecutarán en paralelo. Se le conoce además como *Paralelismo de Datos.*

  El paralelismo de datos se presenta en aplicaciones en las que los datos están sujetos a idéntico procesamiento. Es más apropiado para ejecutar en máquinas SIMD aunque también se pueden ejecutar en computadores MIMD. Se requiere sincronización global después de cada instrucción, lo que resulta en un código ineficiente.

Los lenguajes de paralelismo de datos ofrecen construcciones de alto nivel para compartir información y manejar concurrencia; los programas son más fáciles de escribir y comprender, aunque el código generado por estas construcciones no es tan eficiente como el obtenido usando primitivas de bajo nivel.

El paralelismo de datos /de control es la ejecución simultánea de cadenas de instrucciones diferentes. Las instrucciones se pueden aplicar sobre la misma cadena de datos aunque normalmente se aplican a cadenas de datos diferentes. Adecuados para mapear en MIMD ya que el paralelismo de control requiere múltiples cadenas de instrucciones.

Los lenguajes de paralelismo de datos facilitan al programador la tarea de expresar el paralelismo disponible en un programa de manera independiente de la arquitectura.
Se genera una sola cadena de instrucciones; la ejecución de las instrucciones es síncrona; es más fácil escribir y depurar programas de paralelismo de datos puesto que los bloqueos son imposibles; el programador especifica el paralelismo en el código.

- De *grano medio:* es cuando se divide en módulos, rutinas y tareas o funciones 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. El paralelismo de grano medio en general es explotado por el programador o el compilador. Dentro de él también se encuentran diversas librerías como pueden ser PVM o MPI. El hardware normalmente también se prepara para poder aprovechar este tipo de paralelismo, por ejemplo, los procesadores pueden disponer de instrucciones especiales para ayudar en el cambio de una tarea a otra que realiza el sistema operativo.

- De *grano grueso:* es a nivel de segmentos de código, módulos, procesos o programas, donde las piezas son pocas y de cómputo más intensivo que las de grano fino. Se le conoce como *Paralelismo de Tareas.* El paralelismo de grano grueso se presenta cuando en la aplicación se detectan tareas independientes y éstas se ejecutan como procesos independientes en más de un procesador. En esta clase de granularidad no existe comunicación entre las diferentes aplicaciones. Este esquema es común en las aplicaciones de Productor/Consumidor, Lector/Escritor, Maestro/Esclavo y Cliente /Servidor. En los modelos de Memoria Distribuida (paso de mensajes) sólo se implementa paralelismo de grano grueso.

  El paralelismo de grano grueso es el que explota el programador mediante programas que no tienen por qué requerir la utilización de ninguna librería externa, sino solamente el uso de conocimientos de programación para paralelizar un algoritmo. Se basa principalmente en cualquier tipo de medio que utilice el programador para crear un programa, que solucione un problema de manera paralela, sin tener que hacer uso más que de su habilidad de programador y de un buen algoritmo. Son los más limitados al carecer de métodos específicos para comunicación entre nodos o procesadores.

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

Nivel de Aplicación	Grano muy grueso Decenas de miles de instrucciones Explotado por el S. O. multiprogramado
Nivel de Programa	Grano grueso Miles de instrucciones Explotado por el programador
Nivel de Procedimiento	Grano medio Mil instrucciones Explotado por el programador con ayuda del preprocesador
Nivel de bucle	Grano fino 500 instrucciones Explotado por el compilador y el microprocesador

Figura 2.1 Niveles de granularidad.

Paralelismo de grano fino (paralelismo en el lenguaje máquina).

Es la granularidad más pequeña y basa prácticamente todo su funcionamiento en propiedades del hardware. El hardware puede ser suficientemente inteligente para que el programador no tenga que hacer mucho por soportar esta granularidad, por ejemplo el hardware puede aportar reordenamiento de instrucciones. Es una paralelización automática, se pueden obtener buenas eficiencias en poco tiempo.

Paralelismo de grano medio (paralelismo en threads).

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

El hardware normalmente también se prepara para poder aprovechar este tipo de paralelismo, por ejemplo, los procesadores pueden disponer de instrucciones especiales para ayudar en el cambio de una tarea a otra que realiza el sistema operativo.

Paralelismo de grano grueso (paralelismo de procesos).

El paralelismo de grano grueso es el que explota el programador mediante programas que no tienen por qué requerir la utilización de ninguna librería externa, sino solamente el uso de conocimientos de programación para paralelizar un algoritmo. Se basa principalmente en cualquier tipo de medio que utilice el programador para crear un programa, que solucione un problema de manera paralela, sin tener por que hacer uso más que de su habilidad de programador y de un buen algoritmo.

Son los más limitados al carecer de métodos específicos para comunicación entre nodos o procesadores, se dan en sistemas muy débilmente acoplados.

A continuación se muestra la relación entre los niveles de paralelismo y los grados de granularidad:

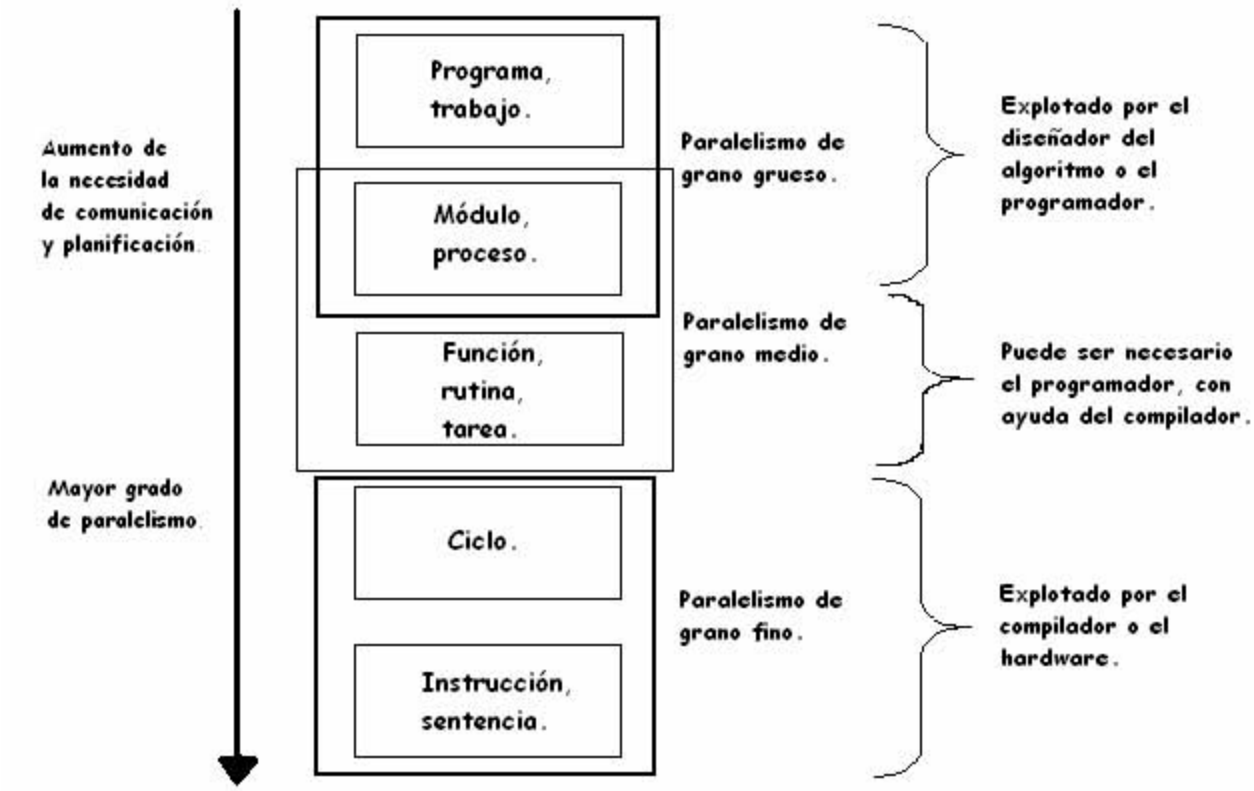


Figura 2.2 Relación entre niveles de paralelismo y granularidad.

2.1.2 Modelos de paralelismo

Uno de los mayores problemas que existen en la creación de programas que hagan uso de paralelización, es la transparencia en la programación de dichos programas. A la hora de crear un programa que resuelva un problema mediante el uso de un algoritmo que explote de alguna

manera la paralelización hay que conocer el sistema de ejecución. Dependiendo del sistema elegido y teniendo en cuenta que por norma general se paralelizan tareas, procesos, procedimientos, rutinas o componentes distribuidos que realizan este trabajo, hay dos modelos de implementación:

1. *Modelo de programación explícita.* El algoritmo paralelo debe especificar explícitamente cómo cooperan los procesadores para resolver un problema específico. La tarea del compilador es sencilla, en cambio la del programador es bastante difícil.

   En este modelo se requiere de una biblioteca de funciones especiales que se encarga tanto de realizar la comunicación como de los métodos de migración y demás factores que en un principio no deben afectar al programador, el cual debe abstraerse de esta capa. Este tipo de modelo es el que utiliza RPC, MPI o PVM. Requiere un conocimiento especial de dicha biblioteca, lo que limita la velocidad de desarrollo del software y además lo hace más costoso debido al tiempo que se debe gastar en el conocimiento de las funciones.

2. *Modelo de programación implícita.* Se usa un lenguaje de programación secuencial y el compilador inserta las instrucciones necesarias para ejecutar el programa en una computadora paralela. El compilador tiene que analizar y comprender las dependencias para asegurar un mapeo eficiente.

   Un programa secuencial normalmente se utiliza en sistemas de memoria compartida. Se paralelizan bucles, es decir, dividen el trabajo en los bucles entre los distintos procesadores. Por ejemplo, si se tiene un bucle de 500 iteraciones, se puede paralelizar el bucle dividiendo el número de iteraciones entre los distintos procesadores. Así, si se tienen 5 procesadores, cada uno realizaría 100 iteraciones como se muestra a continuación:

   ```
   para i=1 to 100
        a[i]=b[i]+a[i-1]
   finpara
   ```

   Si puede haber dependencia de datos no se paraleliza aunque se puede forzar la paralelización con opciones de compilación, o con directivas (OpenMP), generando ficheros con información de bucles paralelizados y no paralelizados.

   Es el modelo quizá más atractivo. Basa todo su funcionamiento en que el programador sepa lo mínimo del sistema para paralelizar sus procesos. Generalmente este modelo lo explotan los compiladores especiales de un sistema particular.

   Por otro lado se suele depender de macros o funciones especiales que delimitan la granularidad de los procesos a migrar. Un caso especial de este tipo es *OpenMosix*: la programación y migración de procesos en *OpenMosix* no requiere de conocimiento del usuario respecto al cluster de máquinas. Lo ideal para obtener transparencia en la programación será programar de manera completamente implícita y que al mismo tiempo el sistema implantado fuese lo menos intruso posible en lo que se refiere a comportamiento con el usuario final.

2.2 Paradigmas de programación paralela

Maestro-Esclavo

Este es uno de los paradigmas de comunicación más sencillo. Es un modelo en el cual se generan un conjunto de subproblemas y procesos que los resuelven. Existe un proceso distinguido llamado maestro y uno o varios procesos idénticos llamados esclavos. El proceso maestro controla a los procesos esclavos, mientras que los procesos esclavos procesan los datos.

El paradigma Maestro-Esclavo funciona de la siguiente manera: el proceso maestro lanza a los esclavos y les envía datos que deben procesar; luego de establecida la relación maestro/esclavo, la jerarquía impone la dirección del control del programa (del maestro sobre los esclavos); la única comunicación desde los esclavos hacia el maestro es para enviar los resultados de la tarea asignada.

Habitualmente no hay dependencias fuertes entre las tareas realizadas por los esclavos, es decir existe poca o nula comunicación entre ellos.

Algunas de las aplicaciones son: técnicas de simulación Monte Carlo, aplicaciones criptográficas, algoritmos de optimización (basados en poblaciones), modelos de partición sencilla de tareas y datos.

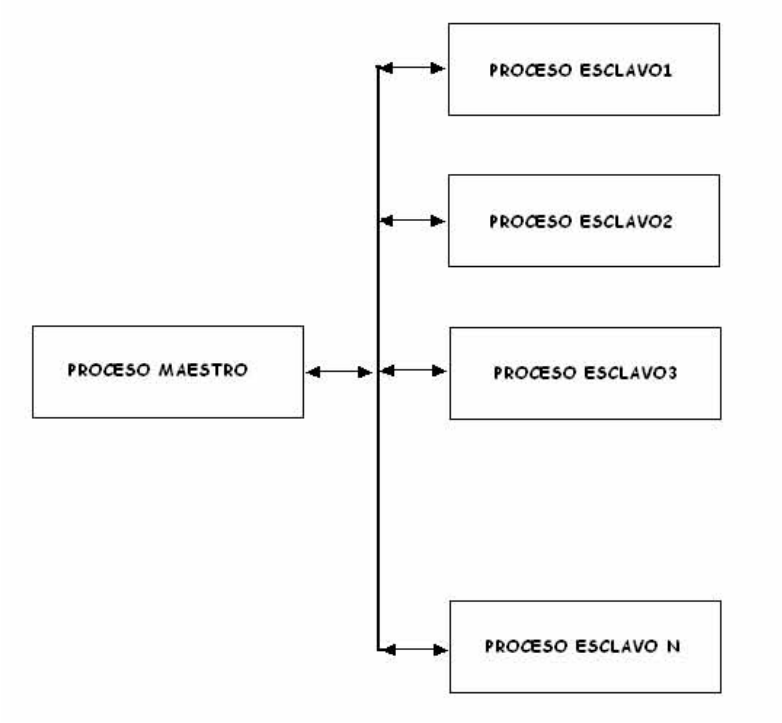


Figura 2.3 Paradigma Maestro-Esclavo.

En la figura 2.3, 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.

Proceso Maestro.
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.
Las actividades que realiza el proceso Esclavo son:

- Recepción de datos a procesar y 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 dependen 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 algoritmo es la que habitualmente muestra el número de procesos esclavo que deben emplearse.

EJEMPLO: Proceso Maestro

```
main ( ) {
      Inicializar;
      Fijar número de esclavos (num_esclavos);
      para i=0 hasta i=num_esclavos {
            datos=Determinar_datos(i);
            Lanzar_tarea(i,datos);
      }
      respuestas=0;
      para i=0 hasta i=num_esclavos {
            res=Obtener_Respuesta();
            resultado=Procesar_resultado(res);
      }
      Desplegar_resultado(resultado);
}
```

EJEMPLO: Proceso Esclavo

```
main() {
/* El proceso esclavo se crea con una referencia al proceso
master, o puede obtenerla mediante una invocación a una
función específica */
     datos=Esperar_datos(master);
     Procesar(datos);
     /* No hay comunicación entre procesos esclavos */
     Enviar_Resultado(master);
}
```

SMPD (*Simple Program Multiple Data*)

En este paradigma de programación, se busca dividir el problema principal en subprogramas que realicen tareas simples. La diferencia fundamental entre esta técnica y la anterior 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.

La manera en que este algoritmo realiza su trabajo se puede representar mediante el esquema:

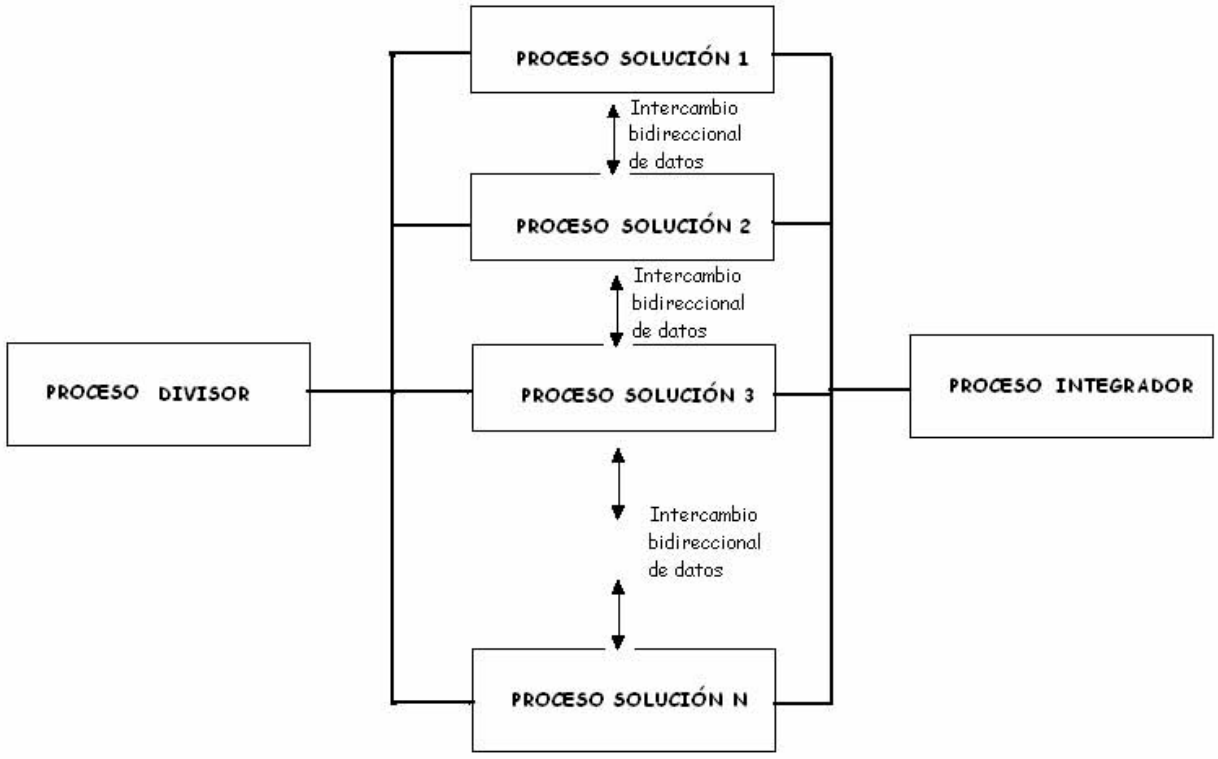


Figura 2.4 Paradigma SMPD.

De acuerdo con lo mostrado en la figura 2.4, existe un proceso divisor que es el encargado de 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 manera más específica, las actividades que realiza cada proceso participante se listan a continuación:

Proceso divisor.
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 porque 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 puede ayudar al procesamiento de datos de otro. Estas cuestiones siempre tendrán repercusiones de la forma de resolver el problema y harán que el algoritmo sea más óptimo o también complicarlo.

Proceso solución.
Las actividades que realiza cada proceso solución son:

- Recibir los datos que son enviados por el proceso divisor.
- Procesamiento de dichos datos.
- 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 determinando recurso.

Proceso integrador
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.

Entubamiento de datos

En este paradigma se genera una sola cadena de instrucciones que se ejecutan de manera síncrona. Es más fácil escribir y depurar programas de paralelismo de datos puesto que los bloqueos son imposibles, y es el programador quien especifica el paralelismo en el código. Asocia un procesador virtual con una unidad fundamental de paralelismo. El programador expresa la computación en términos de las operaciones realizadas por los procesadores virtuales.

El entubamiento de datos permite que cada procesador tenga acceso a las posiciones de memoria de otros procesadores.

Los compiladores para los lenguajes de paralelismo de datos deben mapear los procesadores virtuales en procesadores físicos, generar código para comunicar datos y esforzarse para la ejecución síncrona de instrucciones.

Los procesadores virtuales son emulados por procesadores físicos. Si el número de procesadores virtuales es mayor que el número de procesadores físicos, cada procesador físico emula varios procesadores virtuales.

Algunos lenguajes de paralelismo de datos contienen primitivas que permiten al programador especificar el mapeo deseado de los procesadores virtuales en los físicos.

El siguiente esquema representa este tipo de procesamiento paralelo:

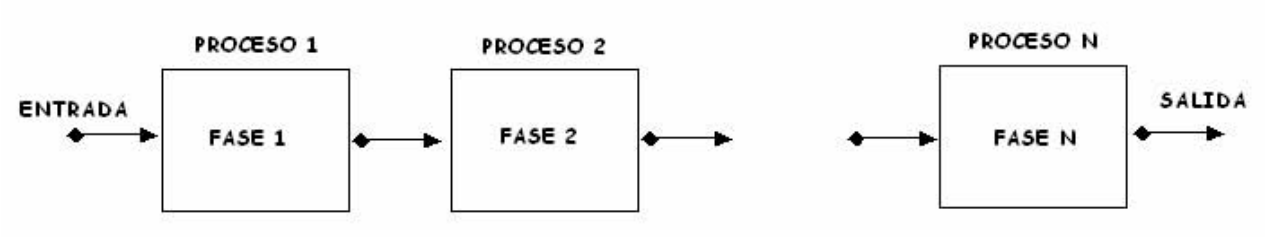


Figura 2.5 Paradigma de Entubamiento de datos.

En la figura 2.5, podemos observar que cada proceso corresponde a una parte de la tubería y es responsable de una tarea en particular. El flujo de los datos va de un fase a otra y la comunicación se lleva a cabo de manera síncrona. 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.
Las actividades que realiza el proceso 1 son:

- 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 hasta que se terminen los datos.

Proceso *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 procedimiento es repetido hasta que deje de recibir datos de su proceso predecesor.

Proceso *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 línea de procesamiento, una vez que ha procesado su información, va generando la solución total.
- Repite este procedimiento hasta que deje de recibir datos del proceso *n-1*. Con los últimos datos recibidos computa el resultado 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 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.

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 causes de datos o tuberías para la solución del problema.

Divide y Conquista.

Este paradigma divide un problema *p* en subproblemas $p_1,p_2,...p_n$; resuelve cada uno de los subproblemas p_i obteniendo s_i (solución *i*). Después combina las soluciones parciales $s_1,s_2,...,sn$ para obtener la solución global de *p*. El éxito del método depende de que se pueda hacer la división y la combinación de forma eficiente.

La solución de los subproblemas se puede hacer en paralelo, es decir, la división del problema *p* debe producir subproblemas de coste balanceado ya que la división y la combinación implicarán comunicaciones y sincronización.

Este paradigma es el esquema más adecuado para paralelizar. Se puede considerar que todos los programas paralelos siguen este esquema.

La manera en que este algoritmo realiza su trabajo se pude representar de la siguiente forma:

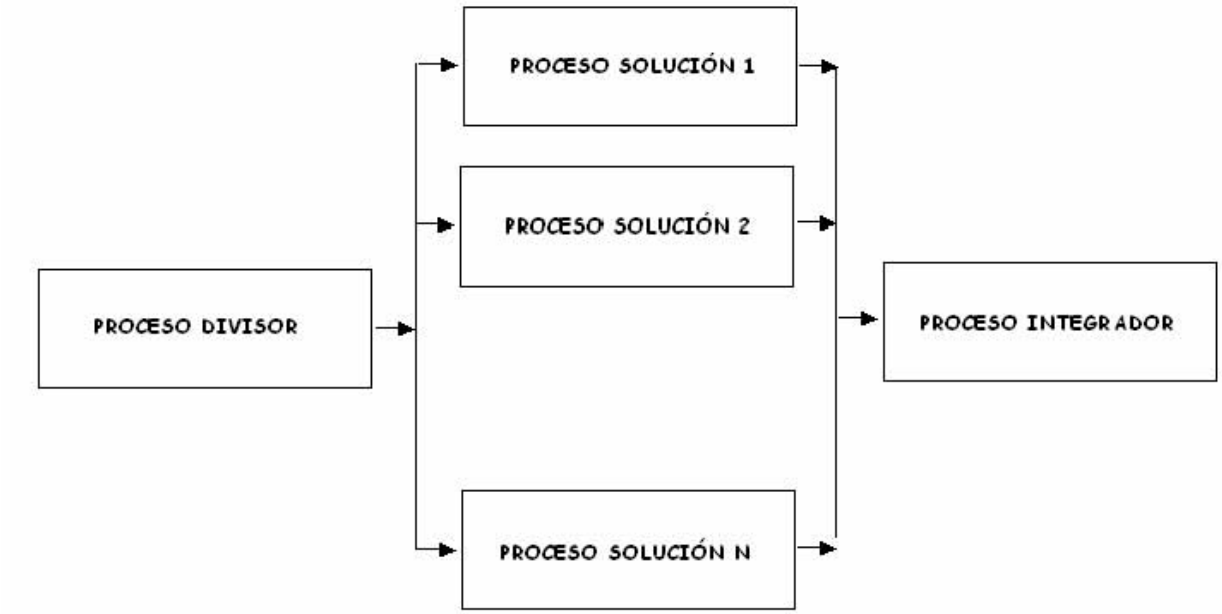


Figura 2.6 Paradigma divide y conquista.

Como se puede apreciar en la figura 2.6, 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 los trabajos de los demás y se proporcionará un resultado final.

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

Proceso divisor.
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.
- Adicionalmente, 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 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 propuesta por Flynn, por la variedad de formas que pueden tener las actividades de cada proceso y del problema en su conjunto.

Proceso solución.
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.
Las actividades que realiza el proceso integrador son:

- Recepción de resultados de cada proceso solución
- Procesamiento final y entrega de resultados.

2.3 Etapas en la creación de programas paralelos

2.3.1 Ley de Amdahl

En cualquier programa paralelizado existen dos tipos de código: el código paralelizado y el código secuencial. Como es sabido existen ciertas secciones de código que ya sea por dependencias, por acceso a recursos únicos o por requerimientos del problema no pueden ser paralelizadas. Estas secciones conforman el código secuencial, que debe ser ejecutado por un solo elemento procesador. Es pues lógico afirmar que la mejora de rendimiento de un programa dependerá completamente de:

1. El tiempo en el que se ejecuta el código serie.
2. El tiempo en el que se ejecuta el código paralelizable.
3. El número de operaciones ejecutadas de forma paralela.

Ésta es la llamada ley de Amdahl y fue descrita por Gene Amdahl en 1967.

Las implicaciones que trae esta ecuación son, a pesar de que no tenga en cuenta las características de cada sistema en concreto:

- El rendimiento no depende completamente del número de procesadores que posea el sistema: en la mayoría de los casos dependerá del número de procesadores máximo que se aprovecharán simultáneamente para ejecutar un programa.
- Cuanto mejor paralelizado esté un programa más susceptible será de aumentar su *speedup* y por tanto explotar el rendimiento del sistema paralelo que lo ejecute.

Supongamos ahora que tenemos un programa que inicialmente no hemos paralelizado, cuyos tiempos de ejecución son 12% y 88%, en serie y en paralelo respectivamente.

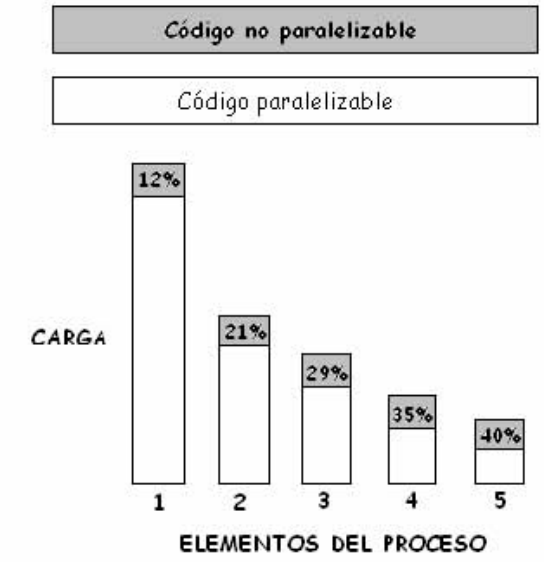


Figura 2.7 Ley de Amdahl.

Como se puede ver en la figura 2.7, la parte no paralelizable del código impide que se pueda escalar de forma lineal, llegará un momento que añadir nuevos procesadores no añadirá una ventaja real al sistema, porque todo lo que estará en ejecución será código secuencial. Por lo tanto para maximizar el aprovechamiento de los sistemas paralelos debe tenerse mucho cuidado con la forma de paralelizar las aplicaciones: cuanto más código secuencial tengan, más problemas de escalabilidad se tendrán.

Como se ha visto, la ley de Amdahl pone límites en lo que se refiere al incremento del rendimiento de cualquier sistema en el que se utilicen fragmentos de código no paralelizable, es decir, de todos los sistemas que actualmente se conocen. Una vez asumida dicha limitación, no queda más opción que optimizar los programas para que el rendimiento de los mismos en el sistema sea el mejor posible. Esto a menudo implica añadir una fase más en la vida de nuestros programas.

Así pues a las fases de análisis, diseño e implementación hay que añadir una fase de paralelización, en la que se deberá tener en cuenta las necesidades del programa y del problema a resolver, así como de los medios de los que se disponen para poder hacer que el programa aumente el rendimiento a medida que aumenta la capacidad de procesamiento de sistema.
Existen otras muchas limitaciones además de la ley de Amdahl que afectan al desarrollo de aplicaciones paralelas, comenzando por el sistema elegido para paralelizar aplicaciones, hasta la granularidad del paralelismo.

2.3.2 Paralelización de programas

Se ha visto que la ley de Amdahl limita el incremento de rendimiento de los programas cuando éstos se ejecutan en sistemas multiprocesadores. También que la optimización de los programas suele depender en la mayoría de los casos del conocimiento del sistema más que del algoritmo que se utilice. Comprendiendo esto puede empezarse a definir cómo debe ser el software a nivel de aplicación para explotar al máximo el paralelismo de estos sistemas. Por un lado se intentará, basándonos en la ley de Amdahl, que el aumento de rendimiento (o *speedup*) de los programas sea el máximo posible, esto implica que los fragmentos de código no paralelizable deben ser los mínimos posibles. Por otro lado deberán conocerse las limitaciones y características del sistema para el que programaremos.

Teóricamente se puede paralelizar cualquier cosa que se haya diseminado mediante técnicas de divide y conquista, procesos, módulos, rutinas, o algoritmos completamente paralelos.

Existen dos formas bien conocidas y fáciles de comprender de paralelismo:

1. *El paralelismo funcional*: divide las aplicaciones en funciones. Se podría ver como paralelismo de código. Por ejemplo puede dividirse en: entrada, preparación del problema, solución del problema, preparación de la salida, salida y mostrar la solución. Esto permite a todos los nodos producir una cadena. Esta aproximación es como la segmentación en funciones de un procesador.

Aquí se sigue la misma idea pero a nivel de software, se dividen los procesos formando una cadena y dependiendo uno del siguiente. Aunque al principio no se logre el paralelismo, una vez que se ponen todos los nodos a trabajar (i.e. cuando hayamos ejecutado N veces lo que estemos ejecutando, siendo N el número de pasos en los que hemos dividido la aplicación) se consiguen que todos los nodos estén trabajando a la vez si todas las funciones tardan el mismo tiempo en completarse. Sino, las demás funciones tienen que esperar a que se complete la función más lenta.
La idea es exactamente la misma que la arquitectura *pipeline*. Esta forma de paralelizar no es ampliamente usada puesto que es muy difícil dividir en funciones que tarden el mismo tiempo en ejecutarse. Sólo se usa en casos concretos donde se ve claramente que es una buena opción por ser fácilmente implementable.

2. *El paralelismo de datos*: se basa en dividir los datos que se tienen que procesar. Típicamente los procesos que están usando esos datos son idénticos entre sí y lo único que hacen es dividir la cantidad de información entre los nodos y procesarla en paralelo. Esta técnica es más usada debido a que es más sencillo realizar el paralelismo.
   Un programa paralelizado no debe obtener peor tasa de eficiencia que un programa secuencial, lo cual no siempre se cumple ni es posible debido a transferencias de comunicación entre los procesos, módulos, nodos o elementos del cluster que generalmente suele ser el cuello de botella del sistema. Es por esto que los elementos de procesos a paralelizar deben ser lo más independientes y ortogonales posibles.

Al paralelizar programas se ve que se obtiene como ventaja un incremento del rendimiento. Este incremento puede ser decremento si estamos comunicando continuamente nuestros procesos a través de una red, que generalmente suele ser el cuello de botella del sistema. Es por esto que los elementos de procesos a paralelizar deben ser lo más independientes y ortogonales posibles, en el sentido de que no deben interferir entre sí.

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

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

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

3. *Elegancia de programación.* Es decisión del programador escribir un algoritmo paralelo aunque no justifique ni por tiempo ni por uso intensivo de otros recursos.

Para paralelizar un programa es importante identificar en el código:

- Tareas independientes, por ejemplo, que existan ciclos *for* o *do* independientes, y rutinas y módulos independientes. De tal modo que no existan dependencias de datos que dificulten la paralelización.
- Zonas donde se efectúa la mayor carga de trabajo y que por lo tanto consumen la mayor parte de tiempo de ejecución.

2.3.3 Etapas en la creación de programas paralelos

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

   Esta etapa se subdivide en dos partes:

   a) Descomposición, ya sea del dominio, donde se trata de dividir los datos en piezas del mismo tamaño y dividir los cálculos que serán realizados asociando a cada operación con los datos sobre los cuales opera, o bien, una descomposición funcional, la cual se concentra en los cálculos que serán realizados (funciones o tareas), para después examinar los datos que serán utilizados por esas tareas; si los datos son disjuntos, el particionamiento será completo, pero si no lo son, se requiere replicar los datos o comunicarlos entre tareas diferentes.

      Los objetivos de la descomposición son:

      - Definir al menos un orden de magnitud de más tareas que procesadores en la computadora paralela.
      - Debe evitar cálculos y almacenamientos redundantes. En caso contrario, difícilmente se logrará un algoritmo escalable.
      - Debe generar tareas de tamaño comparable, pues puede ser difícil asignar a cada procesador cantidades de trabajo similares.

   b) Aglomeración. Para reducir comunicaciones y explorar la vecindad de los cálculos es conveniente considerar si es útil aglomerar o combinar las tareas identificadas en la fase de descomposición.

      Los objetivos de la aglomeración son: balancear la carga de trabajo, determinar la granularidad de paralelismo, reducir las tareas seriales y los costos de sincronización.

2. *Orquestación*. En esta etapa se establecen los patrones de comunicación y sincronización del programa paralelo.

   Los objetivos de la orquestación son:

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

   Sincronización. Una buena sincronización de actividades paralelas se logra mediante las actividades de calendarización y mapeo. La calendarización determina el momento en que se realiza cada una de las actividades, y el mapeo, determina quién ejecutará el trabajo a realizar.

   Existen dos aspectos fundamentales que se deben de considerar:
   a) Reducir el uso de sincronización conservadora. La sincronización de grano fino es más difícil de programar y requiere más operaciones.
   b) Exclusión mutua. Usar candados por tarea en una cola de tareas, no por la cola total y reducir el tamaño de las secciones dentro de una sección crítica.

3. *Mapeo.* En esta etapa se hace una asignación de procesos a procesadores físicos. El objetivo es minimizar el tiempo de ejecución de una programa paralelo.

   Las estrategias son:

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

4. *Desarrollar código*. El código puede venir influenciado por la arquitectura de la máquina sobre la que se trabaja, se elige un modelo de programación, se determinan las comunicaciones entre procesos que deben llevarse a cabo añadiendo el código necesario para llevar a cabo las tareas de control y comunicación entre procesos.

5. *Compilar, probar y depurar*.

2.3.4 Ejemplos de problemas paralelizables

Generalmente los programas convencionales no cuentan con ninguna manera de paralelizar su código. Es decir, sería genial poder contar con algún ejemplo de programa que comprima de formato DVD a DivX en paralelo y poder comprimir una película en un cluster o poder ejecutar programas de compresión en varios nodos a la vez. La realidad es otra. La mayoría de los programas son pensados y implementados de manera secuencial. Si bien existen muchos campos donde el desarrollo de programas que hagan uso de una manera u otra de paralelismo, está creciendo cada vez más.

Actualmente, entre estos campos contamos con:

- Procesos matemáticos: multiplicación de matrices, suma de vectores por componentes, multiplicación de matrices o vectores por escalares o funciones escalares, integrales definidas (creando intervalos más pequeños como elementos de proceso).
- Compresión.
- Aplicación a nuevos paradigmas inherentemente paralelos: Redes Neuronales, Algoritmos Genéticos.

En el campo científico es en el que más se están haciendo desarrollos de este tipo de programas ya que es el campo en el que generalmente es más fácil encontrarse problemas a paralelizar por la naturaleza de éstos. Actualmente otras nuevas vías de desarrollo están siendo bastante utilizadas. Un ejemplo real es el de estudio oceanográfico realizado por el instituto nacional de oceanografía español, en este estudio, se utilizaban matrices descomunales para poder controlar el movimiento de las partículas comprendidas en una región de líquido, que tenía en cuenta viscosidad, temperatura, presión, viento en la capa superficial del liquido, y otras muchas variables.

EJEMPLO: multiplicación de dos matrices.

Se pretende realizar la implementación del problema del producto de dos matrices rectangulares con una aproximación paralela.

Modelo secuencial:

$A\Re \in nxm, B\Re \in mxq, X\Re \in nxq$

$X = A \cdot B$

```
PARA i=0 HASTA n-1
      PARA j=0 HASTA m-1
            x[i, j] = 0
            PARA k=0 HASTA q-1
                  x[i, j] = x[i, j] + a[i, k] * b[k, j]
            FINPARA
      FINPARA
FINPARA
```

El producto se puede analizar considerando una distribución de la matriz *A* por bloques de filas y la matriz *B* replicada completa en cada procesador.

En el caso simplificado de una distribución por filas, cada procesador puede realizar de forma independiente el cálculo del resultado correspondiente a cada fila. Para ello necesita la fila correspondiente de la matriz *A* y todas las columnas de la matriz *B*. De esta forma puede establecerse el siguiente patrón:

- Procesador 0: calcula los elementos desde $x_{0,0}$ hasta $x_{0,n-1}$, utilizando la fila 0 de *A* y todas las columnas de *B*.
- Procesador 1: calcula los elementos desde $x_{1,0}$ hasta $x_{1,n-1}$, utilizando la fila 1 de *A* y todas las columnas de *B*.
- Procesador *k*: calcula los elementos desde $x_{k,0}$ hasta $x_{k,n-1}$, utilizando la fila *k* de *A* y todas las columnas de *B*.

Sin embargo, es mucho más eficiente considerar que cada procesador realice el cálculo de un bloque de filas consecutivas, reduciendo la granularidad del algoritmo.

Considerando que el número de filas (*nb = n/p*) es divisible entre el número de procesadores, puede establecerse el siguiente patrón:

- Procesador 0: calcula desde $x_{0,0}$ hasta $x_{nb-1,n-1}$, a partir de las filas 0 a *nb-1* de *A* y las columnas 0 a *n-1* de *B*.
- Procesador 1: calcula desde $x_{nb,0}$ hasta $x_{2(nb-1),n-1}$, a partir de las filas 1 a *2(nb-1)* de *A* y las columnas 0 a *n-1* de *B*.
- Procesador *k*: calcula desde $x_{k*nb,0}$ hasta $x_{(k+1)*nb-1,n-1}$, a partir de las filas *k*nb* a *(k+1)*nb-1* de *A* y las columnas 0 a *n-1* de *B*.

El programa deberá tener en cuenta los siguientes aspectos:

- Creación de las matrices *A* y *B* (aleatorias) en el Procesador 0.
- Distribución de las mismas por bloques de filas a todos los procesadores.
- Realización del producto parcial en todos los procesadores (incluyendo el Procesador 0).

EJEMPLO: Paralelización del cálculo de número *pi*.

El objetivo es calcular el número *pi* mediante integración numérica, teniendo en cuenta que la integral en el intervalo [0,1] de la derivada del arco tangente de *x* es $\pi / 4$.

$$\left.\begin{array}{l} arctg'(x) = \dfrac{1}{1+x^2} \\ arctg(1) = \dfrac{\pi}{4} \\ arctg(0) = 0 \end{array}\right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} = arctg(x)\Big|_0^1 = \frac{\pi}{4} - 0$$

Vamos a basarnos en una versión secuencial. En ella se aproxima el área en cada subintervalo utilizando rectángulos en los que la altura es el valor de la derivada del arco tangente en el punto medio.

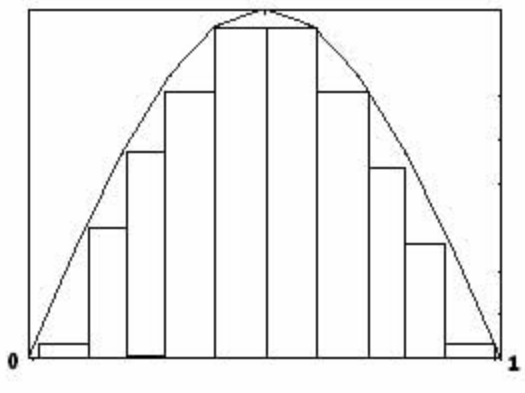
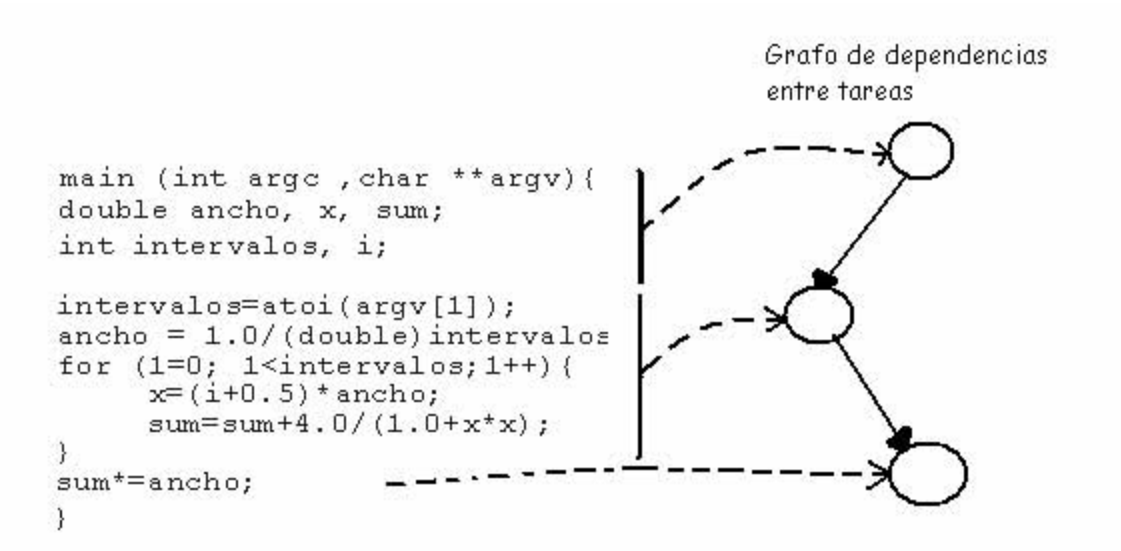


Figura 2.8 Cálculo del número *pi*.

Versión secuencial.



Versión SPMD paralelo.

Supongamos que tenemos un cluster Beowulf con 8 PCs. En cada procesador se ejecutará un proceso distribuyendo la carga de trabajo por igual en cada uno de ellos.
Para realizarlo escribiremos el código paralelo usando MPI (paso de mensajes) en cual se verá más a fondo en el capítulo 3.

```
Incluir bibliotecas     #include <mpi.h>
                        main (int argc, char **argv) {
                        double ancho,x,sum, tsum=0;
Declarar variables      int intervalos, i, nproc,iproc;

                        if (MPI_Init(&argc, &argv)=MPI_SUCCESS)
                             exit(1);
Crear procesos          MPI_Comm_size(MPI_COMM_WORLD, &nproc);
                        MPI_Comm_rank(MPI_COMM_WORLD, &iproc);

                        intervalos=atoi(argv[1]);
                        ancho=1.0/(double)intervalos;
Asignar/localizar       for (i=iproc; i<intervalos;i+=nproc) {
                             x=(i+0.5)*ancho;
                             sum=sum+4.0/(1.0+x*x);
                        }
                        sum*=ancho;
Comunicar/sincronizar   MPI_Reduce(&sum,    &tsum,    1,    MPI_DOUBLE,
                        MPI_SUM,0,MPI_COMM_WORLD);
Terminar procesos       MPI_Finalize();
                        }
```

Crear y terminar procesos:

- `MPI_Init()` crea el proceso que lo ejecuta dentro del mundo de MPI, es decir, dentro del grupo de procesos denominado `MPI_COMM_WORLD`. Una vez que se ejecuta esta función se pueden utilizar el resto de las funciones MPI.
- `MPI_Finalize()` se debe llamar antes de que un proceso creado en MPI acabe su ejecución.
- `MPI_Comm_size(MPI_COMM_WORLD, &nproc)` es una función se pregunta a MPI el número de procesos creados en el grupo `MPI_COMM_WORLD`, se devuelve en `nproc`.
- `MPI_Comm_rank(MPI_COMM_WORLD, &iproc)` es una función que devuelve al proceso su identificador (`tproc`) dentro del grupo.

Asignar tareas a procesos y localizar paralelismo:

- Se reparten las iteraciones del bucle entre los diferentes procesos MPI de forma explícita utilizando un turno aleatorio.

Comunicación y sincronización:

- Los procesos se comunican y sincronizan para sumar las áreas parciales calculadas por los procesos.
- Usamos la operación cooperativa `MPI_Reduce` realizada entre todos los miembros de comunicador para obtener un único resultado final.
- Los procesos envían al proceso 0 el contenido de su variable local `sum`, los contenidos de `sum` se suman y se guardan en el proceso 0 en la variable `tsum`.

CAPÍTULO 3

ENTORNOS Y HERRAMIENTAS DE PROGRAMACIÓN PARALELA

Asegurar el paralelismo en los programas de aplicación para alcanzar ganancias de órdenes de magnitud en el rendimiento ha sido un reto que el cómputo ha atacado desde los 70. Las supercomputadoras vectoriales, los procesadores en arreglos SIMD y los multiprocesadores MPP(Massively Parallel Processors) han explotado formas variables de algoritmos de paralelización a través de una combinación de mecanismos de hardware y software. Los resultados han sido variados. Los más altos grados de rendimiento alcanzados han sido a través de cómputo paralelo. Pero en muchos casos, los rendimientos observados han sido bajos y las dificultades para alcanzarlos han sido grandes.

Los clusters armados, debido a su ventaja superior en precio-rendimiento para un amplio rango de problemas, sufren ahora de gran presión para atacar los problemas que sus predecesores confrontaron. Mientras se han buscado múltiples modelos de programación, un paradigma ha surgido como la forma predominante, al menos a corto plazo. El modelo de "comunicación de procesos secuenciales" comúnmente llamado modelo de "envío de mensajes" ha evolucionado a través de varias implementaciones distintas resultando un estándar de la comunidad: MPI o Interfase de envío de mensajes. MPI no es un lenguaje completo sino una biblioteca que permite a los usuarios de C y Fortran acceder a las bibliotecas de envío de mensajes entre procesos concurrentes en los nodos de procesamiento separados pero interconectados.

Para que la programación paralela sea efectiva requiere algo más que un grupo de construcciones. Se necesita de un ambiente y herramientas para entender el comportamiento operacional de un programa, para corregir los errores en el cálculo y mejorar el rendimiento. El estado de dichas herramientas para los clusters está aún en la infancia aunque muchos equipos han hecho un esfuerzo significativo.

Herramientas para generar programas paralelos

Bibliotecas de funciones para la programación. El programador utiliza un lenguaje secuencial (por ejemplo C o Fortran), para escribir el cuerpo de los procesos y las hebras. También, usando el lenguaje secuencial, distribuye las tareas entre los procesos. El programador, utilizando las funciones de la biblioteca:

- Crea y gestiona los procesos.
- Implementa la comunicación y sincronización.
- Incluso puede elegir la distribución de los procesos entre los procesadores (siempre que lo permitan las funciones).
  Ventajas:
- Los programadores están familiarizados con los lenguajes secuenciales.

- Las bibliotecas están disponibles para todos los sistemas paralelos.
- Las bibliotecas están más cercanas al hardware y dan al programador un control a bajo nivel.
- Podemos usar las bibliotecas tanto para programar con hebras como con procesos. Ejemplos: MPI, PVM, OpenMP y Pthread.

Lenguajes paralelos y directivas del compilador. Se utilizan lenguajes paralelos o lenguajes secuenciales con directivas del compilador, para generar los programas paralelos. Los lenguajes paralelos utilizan:

- Construcciones propias del lenguaje, como por ejemplo FORALL para el paralelismo de datos (por bucles). Estas construcciones, además de distribuir la carga de trabajo, pueden crear y terminar procesos.
- Directivas del compilador para especificar las máquinas virtuales de los procesadores, o cómo se asignan los datos a dichas máquinas.
- Funciones de biblioteca, que implementan en paralelo algunas operaciones típicas. Ventajas:
- Facilidad de escritura y claridad. Son más cortos.

Compiladores paralelos. Un compilador, a partir de un código secuencial, extrae automáticamente el paralelismo a nivel de bucle (de datos) y a nivel de función (de tareas, en esto no es muy eficiente).

PVM / MPI

PVM (Parallel Virtual Machine) es un paquete de software diseñado para correr en una colección heterogénea de computadoras conectadas por una o más redes. PVM es un conjunto de herramientas del Oak Ridge National Laboratory para la paralelización con memoria distribuida en FORTRAN y C.

La meta es que PVM sea capaz de ver a todas estas máquinas como una sola y utilizarla como una máquina paralela de cómputo. En consecuencia, los problemas que impliquen cómputo a gran escala, pueden ser resueltos de manera más efectiva, a través del uso de la potencia y memoria de varias computadoras. El software es bastante portable. El código fuente del mismo, se encuentra disponible en forma gratuita a través de *netlib* y ha sido compilado en distintas arquitecturas desde *laptops* hasta *CRAYs*.

PVM permite a los usuarios explotar su hardware existente, para resolver problemas mucho más grandes a un costo adicional mínimo. En general programadores y científicos están utilizando PVM para resolver problemas tanto de tipo científico, industrial y médico. Adicionalmente se está utilizando PVM como una herramienta educacional para la enseñanza de programación paralela.

MPI es una interfaz de paso de mensajes (*Message Passing Interface*). El paso de mensajes es un modelo de comunicación ampliamente usado en computación paralela.
El objetivo principal de MPI es lograr la portabilidad a través de diferentes máquinas, tratando de obtener un grado comparable al de un lenguaje de programación que permita ejecutar de manera transparente, aplicaciones sobre sistemas heterogéneos.

Las principales características de MPI son:

- Estándar formalmente especificado.
- Código fuente 100% portable.
- Permite bibliotecas externas (desarrolladas por terceros).
- Tipos de datos derivados arbitrariamente para minimizar *overhead*.
- Topologías de procesos para ganar eficiencia en MPP.
- Permite solapamiento completo en la comunicación.
- Comunicación en grupo extensa.

3.1 MPI (*Message Passing Interface*).

3.1.1 Introducción a MPI

MPI (*Message Passing Interface*) es una interfaz estandarizada para la realización de aplicaciones paralelas basadas en paso de mensajes. El modelo de programación que subyace tras MPI es MIMD (*Multiple Instruction, Multiple Data*) aunque se dan especiales facilidades para la utilización del modelo SPMD (*Single Program Multiple Data*), un caso particular de MIMD en el que todos los procesos ejecutan el mismo programa, aunque no necesariamente la misma instrucción al mismo tiempo. MPI es, como su nombre lo indica, una interfaz, lo que quiere decir que el estándar no exige una determinada implementación del mismo.

Lo importante es dar al programador una colección de funciones para que éste diseñe su aplicación, sin que tenga necesariamente que conocer el hardware concreto sobre el que se va a ejecutar, ni la forma en la que se han implementado las funciones que emplea.

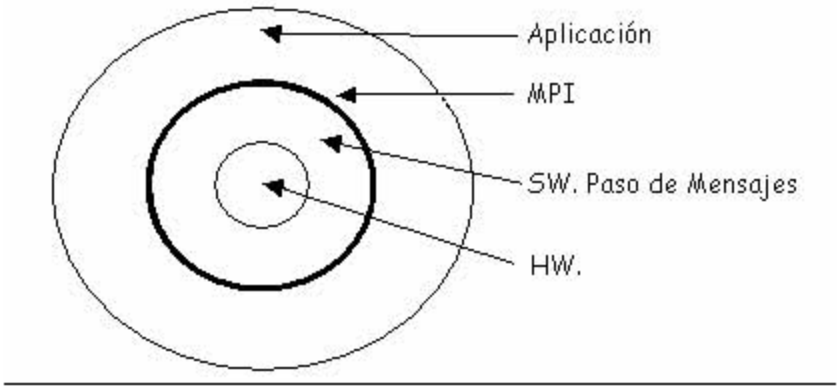


Figura 3.1 Ubicación de MPI en el proceso de programación de aplicaciones paralelas

MPI ha sido desarrollado por el MPI Forum, un grupo formado por investigadores de universidades, laboratorios y empresas involucrados en la computación de altas prestaciones. Los objetivos fundamentales del MPI Forum son los siguientes:

1. Definir un entorno de programación único que garantice la portabilidad de las aplicaciones paralelas.
2. Definir totalmente la interfaz de programación, sin especificar cómo debe ser la implementación del mismo
3. Ofrecer implementaciones de calidad, de dominio público, para favorecer la extensión del estándar.
4. Convencer a los fabricantes de computadoras paralelas para que ofrezcan versiones de MPI optimizadas para sus máquinas (lo que ya han hecho fabricantes como IBM y Silicon Graphics).

Los elementos básicos de MPI son una definición de una interfaz de programación independiente de lenguajes, más una colección de *bindings* o concreciones de esa interfaz para los lenguajes de programación más extendidos en la comunidad usuaria de computadoras paralelas: C y FORTRAN.

Un programador que quiera emplear MPI para sus proyectos trabajará con una implementación concreta de MPI, que constará de, al menos, estos elementos:

- Una biblioteca de funciones para C, más el archivo de cabecera mpi.h con las definiciones de esas funciones y de una colección de constantes y macros.
- Una biblioteca de funciones para FORTRAN + mpif.h.
- Comandos para compilación, típicamente `mpicc o mpif77`, que son versiones de los comandos de compilación habituales (`cc`, f77) que incorporan automáticamente las bibliotecas MPI.
- Comandos para la ejecución de aplicaciones paralelas, típicamente `mpirun`.
- Herramientas para monitorización y depuración.

MPI no es, evidentemente, el único entorno disponible para la elaboración de aplicaciones paralelas. Existen muchas alternativas, destacamos las siguientes:

- Utilizar las bibliotecas de programación propias de la computadora paralela disponible: NX en el Intel Paragon, MPL en el IBM SP2, etc.
- PVM (*Parallel Virtual Machine*): de características similares a MPI, se desarrolló con la idea de hacer que una red de estaciones de trabajo funcionase como una multicomputadora. Funciona también en multicomputadoras, normalmente como una capa de software encima del mecanismo de comunicaciones nativo.
- Usar, si es posible, lenguajes de programación paralelos (FORTRAN 90) o secuenciales (C, FORTRAN 77) con directivas de paralelismo.
- Usar lenguajes secuenciales junto con compiladores que paralelicen automáticamente.

MPI está aún en sus comienzos, y aunque se está haciendo un hueco creciente en la comunidad de programadores de aplicaciones científicas paralelas, no es probable que desplace a corto plazo a los entornos de programación ya existentes (como los anteriormente citados) o impida la aparición de otros nuevos. El MPI Forum es consciente de que MPI todavía adolece de algunas limitaciones, e incluso ha identificado bastantes de ellas:

- Entrada/salida: no se establece un mecanismo estandarizado de E/S paralela.
- Creación dinámica de procesos. MPI asume un número de procesos constante, establecido al arrancar la aplicación.
- Variables compartidas. El modelo de comunicación estandarizado por MPI sólo tiene en cuenta el paso de mensajes.
- *Bindings* para otros lenguajes, además de C y FORTRAN. Se piensa, en concreto, en C++ y ADA.
- Soporte para aplicaciones de tiempo real. MPI no recoge en ningún punto restricciones de tiempo real.
- Interfaces gráficos. No se define ningún aspecto relacionado con la interacción mediante GUIs con una aplicación paralela.

MPI está especialmente diseñado para desarrollar aplicaciones SPMD. Al arrancar una aplicación se lanzan en paralelo *N* copias del mismo programa (procesos). Estos procesos no avanzan sincronizados instrucción a instrucción sino que la sincronización, cuando sea necesaria, tiene que ser explícita. Los procesos tienen un espacio de memoria completamente separado. El intercambio de información, así como la sincronización, se hacen mediante paso de mensajes.

Se dispone de funciones de comunicación punto a punto (que involucran sólo a dos procesos), y de funciones u operaciones colectivas (que involucran a múltiples procesos). Los procesos pueden agruparse y formar *comunicadores*, lo que permite una definición del ámbito de las operaciones colectivas, así como un diseño modular.

La estructura típica de un programa MPI, usando el *binding* para C, es la siguiente:

```
# include "mpi.h"
main (int argc, char **argv) {
int nproc; /* Número de procesos */
int yo; /* Mi dirección: 0<=yo<=(nproc-1) */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &yo);
/* CUERPO DEL PROGRAMA */
MPI_Finalize();
}
```

Este segmento de código ya nos presenta cuatro de las funciones más utilizadas de MPI:

`MPI_Init()` para iniciar la aplicación paralela, `MPI_Comm_size()` para averiguar el número de procesos que participan en la aplicación, `MPI_Comm_rank()` para que cada proceso averigüe su dirección (identificador) dentro de la colección de procesos que componen la aplicación, y `MPI_Finalize()` para dar por finalizada la aplicación.

El ejemplo nos sirve también para que prestemos atención a algunas convenciones de MPI. Los nombres de todas las funciones empiezan con "`MPI_`", la primera letra que sigue siempre es mayúscula, y el resto son minúsculas. La mayor parte de las funciones MPI devuelven un entero, que es un diagnóstico. Si el valor devuelto es `MPI_SUCCESS`, la función se ha realizado con éxito. No se han estandarizado otros posibles valores.

La palabra clave `MPI_COMM_WORLD` hace referencia al comunicador universal, un comunicador predefinido por MPI que incluye a todos los procesos de la aplicación.

Todas las funciones de comunicación de MPI necesitan como argumento un comunicador.

Comunicación Punto a Punto

Un buen número de funciones de MPI están dedicadas a la comunicación entre pares de procesos. Existen múltiples formas de intercambiar un mensaje entre dos procesos, en función del *modelo* y el *modo* de comunicación elegido.

3.1.2 Modelos y modos de comunicación

MPI define dos modelos de comunicación: bloqueante (*blocking*) y no bloqueante (*nonblocking*).

El modelo de comunicación tiene que ver con el tiempo que un proceso pasa bloqueado tras llamar a una función de comunicación, sea ésta de envío o de recepción. Una función bloqueante mantiene a un proceso bloqueado hasta que la operación solicitada finalice.

Una no bloqueante supone simplemente "encargar" al sistema la realización de una operación, recuperando el control inmediatamente. El proceso tiene que preocuparse, más adelante, de averiguar si la operación ha finalizado o no.

Queda una duda, sin embargo: ¿cuándo damos una operación por finalizada? En el caso de la recepción está claro: cuando tengamos un mensaje nuevo, completo, en el buffer asignado al efecto. En el caso de la emisión es más compleja: se puede entender que la emisión ha terminado cuando el mensaje ha sido recibido en destino, o se puede ser menos restrictivo y dar por terminada la operación en cuanto se ha hecho una copia del mensaje en un buffer del sistema en el lado emisor.

MPI define un envío como finalizado cuando el emisor puede reutilizar, sin problemas de causar interferencias, el buffer de emisión que tenía el mensaje. Dicho esto, podemos entender que tras hacer un send (envío) bloqueante podemos reutilizar el buffer asociado sin problemas, pero tras hacer un send no bloqueante tenemos que ser muy cuidadosos con las manipulaciones que se realizan sobre el buffer, bajo el riesgo de alterar inadvertidamente la información que se está enviando.

Al margen de si la función invocada es bloqueante o no, el programador puede tener un cierto control sobre la forma en la que se realiza y completa un envío. MPI define, en relación a este aspecto, 4 modos de envío: básico (*basic*), con buffer (*buffered*), síncrono (*synchronous*) y listo (*ready*).

Cuando se hace un envío *con buffer* se guarda inmediatamente, en un buffer al efecto en el emisor, una copia del mensaje. La operación se da por completa en cuanto se ha efectuado esta copia. Si no hay espacio en el buffer, el envío fracasa.

El modo de envío *básico* no especifica la forma en la que se completa la operación: es algo dependiente de la implementación. Normalmente equivale a un envío con buffer para mensajes cortos y a un envío síncrono para mensajes largos. Se intenta así agilizar el envío de mensajes cortos a la vez que se procura no perder demasiado tiempo realizando copias de la información.

En cuanto al envío en modo *listo*, sólo se puede hacer si antes el otro extremo está preparado para una recepción inmediata. No hay copias adicionales del mensaje (como en el caso del modo con buffer), y tampoco podemos confiar en bloquearnos hasta que el receptor esté preparado.

Si se hace un envío *síncrono*, la operación se da por terminada sólo cuando el mensaje ha sido recibido en destino. Este es el modo de comunicación habitual en los sistemas basados en *Transputers*. En función de la implementación elegida, puede exigir menos copias de la información conforme ésta circula del buffer del emisor al buffer del receptor.

Comunicación con buffer

Uno de los problemas que tiene el envío básico es que el programador no tiene control sobre cuánto tiempo va a tardar en completarse la operación. Puede que se tome más tiempo, si es que el sistema se limita a hacer una copia del mensaje en un buffer, que saldrá más tarde hacia su destino; pero también puede que mantenga al proceso bloqueado un largo tiempo, esperando a que el receptor acepte el mensaje.

Para evitar el riesgo de un bloqueo no deseado se puede solicitar explícitamente que la comunicación se complete copiando el mensaje en un buffer, que tendrá que asignar al efecto el propio proceso. Así, se elimina el riesgo de bloqueo mientras se espera a que el interlocutor esté preparado.

La función correspondiente es `MPI_Bsend()`, que tiene los mismos argumentos que `MPI_Send()`:

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm  comm);
```

Para que se pueda usar el envío con buffer es necesario que el programador asigne un buffer de salida. Para ello que hay reservar previamente una zona de memoria (de forma estática o con `malloc()`) y luego indicarle al sistema que la emplee como buffer. Esta última operación la hace `MPI_Buffer_attach()`.

Ese buffer se puede recuperar usando `MPI_Buffer_detach()`.

```
int MPI_Buffer_attach(void* buffer, int size);

int MPI_Buffer_detach(void* buffer, int* size);
```

Una peculiaridad de los envíos con buffer es que fracasan en el caso de que el buffer no tenga suficiente espacio como para contener un mensaje, y el resultado es que el programa aborta. El programador puede decidir entonces que el buffer asignado es muy pequeño, y asignar uno más grande para una ejecución posterior, o bien cambiar el modo de comunicación a otro con menos requerimientos de buffer pero con más riesgo de bloqueo.

El resultado de la combinación de dos modelos y cuatro modos de comunicación nos da 8 diferentes funciones de envío. Funciones de recepción sólo hay dos, una por modelo.

3.1.3 Prototipos de las funciones más habituales

Empezamos con `MPI_Send()` y `MPI_Recv()` que son, respectivamente, las funciones de envío y recepción básicas bloqueantes.

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

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

El significado de los parámetros es como sigue. `Buf, count` y `datatype` forman el mensaje a enviar o recibir, `count` número de copias de un dato del tipo `datatype` que se encuentran (o se van a dejar) en memoria a partir de la dirección indicada por `buf`. `Dest` es, en las funciones de envío, el identificador del proceso destinatario del mensaje. `Source` es, en las funciones de recepción, el identificador del emisor del cual esperamos un mensaje. Si no nos importa el origen del mensaje, podemos poner `MPI_ANY_SOURCE.` `Tag` es una etiqueta que se puede poner al mensaje. El significado de la etiqueta lo asigna el programador. Suele emplearse para distinguir entre diferentes clases de mensajes. El emisor pone siempre una etiqueta a los mensajes, y el

receptor puede elegir entre recibir sólo los mensajes que tengan una etiqueta dada, o aceptar cualquier etiqueta, poniendo `MPI_ANY_TAG` como valor de este parámetro. `Comm` es un comunicador; en muchas ocasiones se emplea el comunicador universal `MPI_COMM_WORLD`. *Status* es un resultado que se obtiene cada vez que se completa una recepción, y nos informa de aspectos tales como el tamaño del mensaje recibido, la etiqueta del mensaje y el emisor del mismo.

La definición de la estructura `MPI_Status` es la siguiente:

```
typedef struct {
int MPI_SOURCE;
int MPI_TAG; /* otros campos no accesibles */
} MPI_Status;
```

Podemos acceder directamente a los campos. `MPI_SOURCE` y `MPI_TAG`, pero a ningún otro más. Si queremos saber el tamaño de un mensaje, lo haremos con la función `MPI_Get_count()` como se muestra a continuación:

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

`MPI_Isend()` y `MPI_Irecv()` son las funciones de emisión/recepción básicas no bloqueantes.

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Wait(MPI_Request *request, MPI_Status *status);

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);

int MPI_Cancel(MPI_Request *request);
```

Las funciones no bloqueantes manejan un objeto `request` del tipo `MPI_Request`. Este objeto es una especie de “recibo” de la operación solicitada. Se podrá utilizar este recibo para saber si la operación ha terminado o no.

La función `MPI_Wait()` toma como entrada un recibo, y bloquea al proceso hasta que la operación correspondiente termina. Por lo tanto, hacer un `MPI_Isend()` seguido de un `MPI_Wait()` equivale a hacer un envío bloqueante. Sin embargo, entre la llamada a la función de envío y la llamada a la función de espera, el proceso puede haber estado haciendo cosas útiles, es decir, consigue solapar parte de cálculo de la aplicación con la comunicación.

Cuando no interesa bloquearse, sino simplemente saber si la operación ha terminado o no, podemos usar `MPI_Test()`. Esta función actualiza una bandera que se le pasa como segundo parámetro. Si la función ha terminado, esta bandera toma el valor 1, y si no ha terminado pasa a valer 0.

Por último, `MPI_Cancel()` nos sirve para cancelar una operación de comunicación pendiente, siempre que ésta aún no se haya completado.

Recepción por encuesta

Las funciones de recepción de mensajes engloban en una operación la sincronización con el emisor (esperar a que haya un mensaje disponible) con la de comunicación (copiar ese mensaje).

Sin embargo, conviene separar ambos conceptos. Por ejemplo, podemos estar a la espera de mensajes de tres clases, cada una asociada a un tipo de datos diferente, y la clase nos viene dada por el valor de la etiqueta. Por lo tanto, nos gustaría saber el valor de la etiqueta antes de leer el mensaje. También puede ocurrir que nos llegue un mensaje de longitud desconocida, y resulte necesario averiguar primero el tamaño para así asignar dinámicamente el espacio de memoria requerido por el mensaje.

Las funciones `MPI_Probe()` y `MPI_Iprobe()` nos permiten saber si tenemos un mensaje recibido y listo para leer, pero sin leerlo. A partir de la información de estado obtenida con cualquiera de estas funciones, podemos averiguar la identidad del emisor del mensaje, la etiqueta del mismo y su longitud. Una vez hecho esto, podemos proceder a la lectura real del mensaje con la correspondiente llamada a `MPI_Recv()` o `MPI_Irecv()`.

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
MPI_Status *status);

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

`MPI_Probe()` es bloqueante: sólo devuelve el control al proceso cuando hay un mensaje listo.

`MPI_Iprobe()` es no bloqueante: nos indica en el argumento *flag* si hay un mensaje listo o no—es decir, realiza una encuesta.

Tipos de datos

Los mensajes gestionados por MPI son secuencias de *count* elementos del tipo *datatype*. MPI define una colección de tipos de datos primitivos, correspondientes a los tipos de datos existentes en C. Hay otra colección distinta para FORTRAN.

Tipos MPI	Tipos C equivalentes
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIDED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	sin equivalente

Figura 3.2 Equivalencia de datos MPI y C

Aunque una aplicación desarrollada en C trabaja con los tipos de datos habituales, cuando se realice un paso de mensajes habrá que facilitar a MPI un descriptor del tipo equivalente, tal como lo define MPI.

La idea de fondo es la siguiente: los procesadores que participan en una aplicación MPI no tienen por qué ser todos iguales. Es posible que existan diferencias en la representación de los datos en las distintas máquinas. Para eliminar los problemas que puedan surgir, MPI realiza, si son necesarias, transformaciones de sintaxis que posibilitan la comunicación en entornos heterogéneos. La excepción la constituyen los datos de tipo `MPI_BYTE`, que se copian sin más de una máquina a otra. Aparte de los tipos simples definidos de forma primitiva por MPI, se permite la definición de tipos de usuario, más complejos.

Etiquetas y comunicadores

Todo mensaje que se envía con MPI va etiquetado: parámetro *tag*.

El proceso receptor puede elegir entre aceptar mensajes con una cierta etiqueta (parámetro *tag* con un valor concreto), o decir que acepta un mensaje cualquiera (`MPI_ANY_TAG`). Tras la recepción, se puede saber la etiqueta concreta accediendo a `status.MPI_TAG`.

Las etiquetas permiten diferenciar las diferentes clases de información que pueden intercambiarse los procesos.

La comunicación se produce dentro de un *comunicador* (entorno de comunicación), que es básicamente un conjunto de procesos. El comunicador universal `MPI_COMM_WORLD` está siempre definido, e incluye a todos los procesos que forman parte de la aplicación. Los comunicadores permiten que diferentes grupos de procesos actúen sin interferir, así como dividir la aplicación en fases no solapadas. MPI garantiza la entrega ordenada de mensajes dentro de un mismo comunicador.

3.1.4 Operaciones colectivas

Muchas aplicaciones requieren de la comunicación entre más de dos procesos. Esto se puede hacer combinando operaciones punto a punto, pero para que le resulte más cómodo al programador, y para posibilitar implementaciones optimizadas, MPI incluye una serie de operaciones colectivas:

- Barreras de sincronización
- Difusión (*broadcast*)
- Recolección (*gather*)
- Distribución (*scatter*)
- Operaciones de reducción (suma, multiplicación, mínimo, etc.)
- Combinaciones de todas ellas

Una operación colectiva tiene que ser invocada por *todos* los participantes, aunque los roles que jueguen no sean los mismos. La mayor parte de las operaciones requieren la designación de un proceso como *root* o raíz de la operación.

Barreras y broadcast

Dos de las operaciones colectivas más comunes son las barreras de sincronización (`MPI_Barrier()`) y el envío de información en modo difusión (`MPI_Broadcast()`).

La primera de estas operaciones no exige ninguna clase de intercambio de información. Es una operación puramente de sincronización, que bloquea a los procesos de un comunicador hasta que todos ellos han pasado por la barrera. Suele emplearse para dar por finalizada una etapa del programa, asegurándose de que todos han terminado antes de dar comienzo a la siguiente.

```
int MPI_Barrier(MPI_Comm comm);
```

`MPI_Broadcast`() sirve para que un proceso, el raíz, envíe un mensaje a todos los miembros del comunicador. Esta función ya muestra una característica peculiar de las operaciones colectivas de MPI: todos los participantes invocan la misma función, designando al mismo proceso como raíz de la misma. Una implementación alternativa sería tener una función de envío especial, en modo *broadcast*, y usar las funciones de recepción normales para leer los mensajes.

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

Este esquema representa el significado de un *broadcast*. En las filas se representan los procesos de un comunicador, y en las columnas los datos que posee cada proceso. Antes del *broadcast*, el procesador raíz (se supone que es el 0) tiene el dato A0. Tras realizar el *broadcast*, todos los procesos (incluyendo el raíz) tienen una copia de A0.

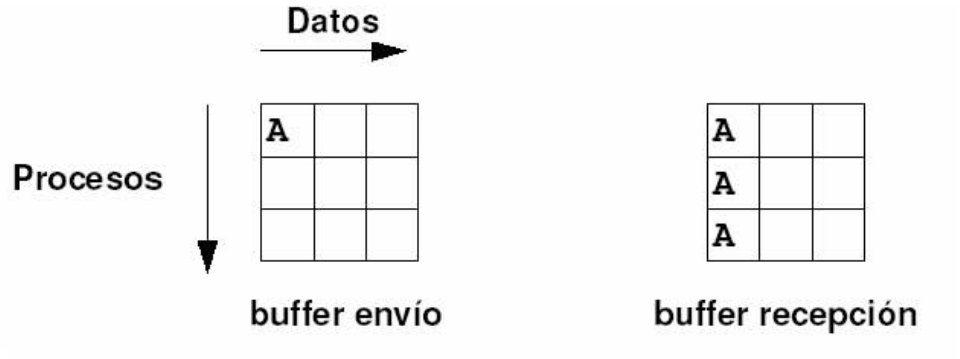


Figura 3.3 Esquema de la operación colectiva MPI_Broadcast().

"Buffer envío" indica los contenidos de los *bufferes* de envío antes de proceder a la operación colectiva. "Buffer recepción" indica los contenidos de los *bufferes* de recepción tras completarse la operación.

Reducción

Una reducción es una operación realizada de forma cooperativa entre todos los procesos de un comunicador, de tal forma que se obtiene un resultado final que se almacena en el proceso raíz.

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

MPI define una colección de operaciones del tipo `MPI_Op` que se pueden utilizar en una reducción: `MPI_MAX` (máximo), `MPI_MIN` (mínimo), `MPI_SUM` (suma), `MPI_PROD` (producto), `MPI_LAND` (and lógico), `MPI_BAND` (and bit a bit), `MPI_LOR` (or lógico), `MPI_BOR` (or bit a bit), `MPI_LXOR` (xor lógico), `MPI_BXOR` (xor bit a bit), etc.

En ocasiones el programador puede necesitar otra operación de reducción distinta, no predefinida. Para ello MPI ofrece la función `MPI_Op_create()`, que toma como argumento de entrada una función de usuario y devuelve un objeto de tipo `MPI_Op` que se puede emplear con `MPI_Reduce`(). Ese objeto se puede destruir más adelante con `MPI_Op_free()`.

```
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op
);
int MPI_Op_free(MPI_Op *op);
```

La operación a realizar puede ser cualquiera. En general se emplean operaciones conmutativas y asociativas, es decir, cuyo resultado no depende del orden con el que se procesan los operandos. Eso se indica con el valor 1 en el parámetro `commute`. Si la operación no es conmutativa

(`commute` = 0) entonces se exige que la reducción se realice en orden de dirección (se empieza con el proceso 0, luego con el 1, con el 2, etc.).

En ocasiones resulta útil que el resultado de una reducción esté disponible para todos los procesos. Aunque se puede realizar un *broadcast* tras la reducción, podemos combinar la reducción con la difusión usando `MPI_Allreduce()`. Si el resultado de la reducción es un vector que hay que distribuir entre los procesadores, podemos combinar la reducción y la distribución usando `MPI_Reduce_scatter()`.

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

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

Una variante más de la reducción nos la da `MPI_Scan()`. Es similar a `MPI_Allreduce()`, excepto que cada proceso recibe un resultado parcial de la reducción, en vez del final: cada proceso *i* recibe, en vez del resultado de `OP`(*0 ... n-1*) - siendo *n* el número total de procesos - el resultado de `OP`*(0 ... i)*.

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

Modularidad

MPI permite definir grupos de procesos. Un grupo es una colección de procesos, y define un espacio de direcciones (desde 0 hasta el tamaño del grupo menos 1). Los miembros del grupo tienen asignada una dirección dentro de él. Un proceso puede pertenecer simultáneamente a varios grupos, y tener una dirección distinta en cada uno de ellos.

Un comunicador es un universo de comunicación. Básicamente consiste en un grupo de procesos, y un contexto de comunicación. Las comunicaciones producidas en dos comunicadores diferentes nunca interfieren entre sí.

El concepto de comunicador se introdujo para facilitar la elaboración de bibliotecas de programas: el programa de un usuario se ejecuta en un comunicador, y las funciones de la biblioteca en otro diferente (posiblemente, con el mismo grupo de procesos, pero con un contexto diferente). Así no hay riesgo de que se mezclen mensajes del usuario con mensajes privados de las funciones de la biblioteca. Además, así tampoco hay problemas a la hora de usar etiquetas.

Descripción de algunas de las funciones sobre comunicadores más comunes. Dos de ellas ya han sido presentadas: `MPI_Comm_size()` para averiguar el tamaño (número de procesos) de un comunicador, y `MPI_Comm_rank()` para que un proceso obtenga su identificación dentro del comunicador.

```
int MPI_Comm_size(MPI_Comm comm, int *size);

int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

La función `MPI_Comm_dup()` permite crear un nuevo comunicador, con el mismo grupo de procesos, pero diferente con texto. Se puede usar antes de llamar a una función de una biblioteca. La figura 3.4 (izquierda) muestra cómo puede ocurrir que un mensaje del usuario (flecha fina) interfiera con los mensajes propios de una función de biblioteca (flechas gruesas). Si la biblioteca trabaja en un comunicador diferente (derecha) entonces no hay problemas de interferencia.

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);
```

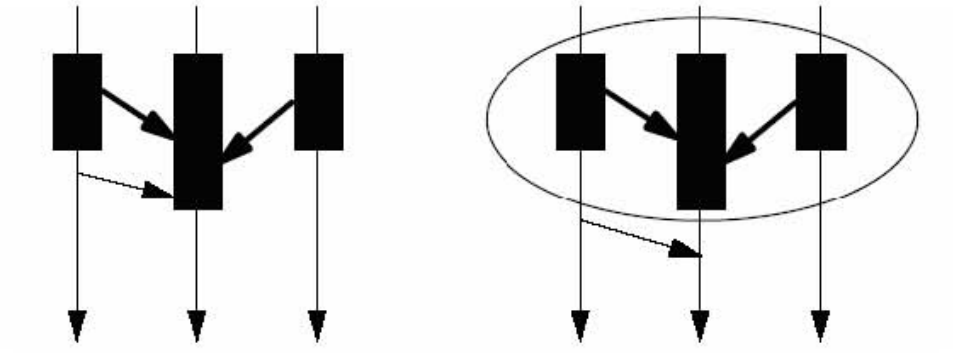


Figura 3.4 Utilización de un comunicador
para aislar una función de biblioteca

…

```
MPI_Comm_dup (MPI_COMM_WORLD, &newcomm); /* llamada a función, que se ejecutará
dentro de newcomm */
MPI_Comm_free (&newcomm);
```

…

La operación `MPI_Comm_dup()`, así como todas las demás que crean comunicadores, son operaciones colectivas: tienen que ser invocadas por todos los procesos implicados, con argumentos compatibles.

`MPI_Comm_free()` destruye un comunicador que ya no se va a emplear. Por su parte, `MPI_Comm_split`() crea, a partir de un comunicador inicial, varios comunicadores diferentes, cada uno con un conjunto disjunto de procesos. El *color* determina en qué comunicador queda cada uno de los procesos.

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm
*newcomm);
```

EJEMPLO: creación de comunicadores con `MPI_Comm_split`().

Partiendo del comunicador universal, definimos dos comunicadores: uno con aquellos procesos que tienen dirección par en el comunicador `MPI_COMM_WORLD`, y otro con los procesos impares. Cada proceso pertenecerá a dos comunicadores: al universal y al recién creado, y tendrá por lo tanto dos direcciones, una por comunicador.

```
int myglobalid, myotherid, color;
MPI_Comm newcom;
MPI_Comm_rank (MPI_COMM_WORLD, &myglobalid);
if (myglobalid%2 == 0) color = 0;
else color = 1;
MPI_Comm_split (comm, color, myglobalid, &newcom);
MPI_Comm_rank (newcom, &myotherid);
/* operaciones entre pares e impares, por separado */
MPI_Comm_free (&newcom);
```

La figura 3.5 muestra una colección de seis procesos que forman inicialmente parte del comunicador universal y que, tras `MPI_Comm_split()` pasan a formar dos nuevos comunicadores disjuntos.

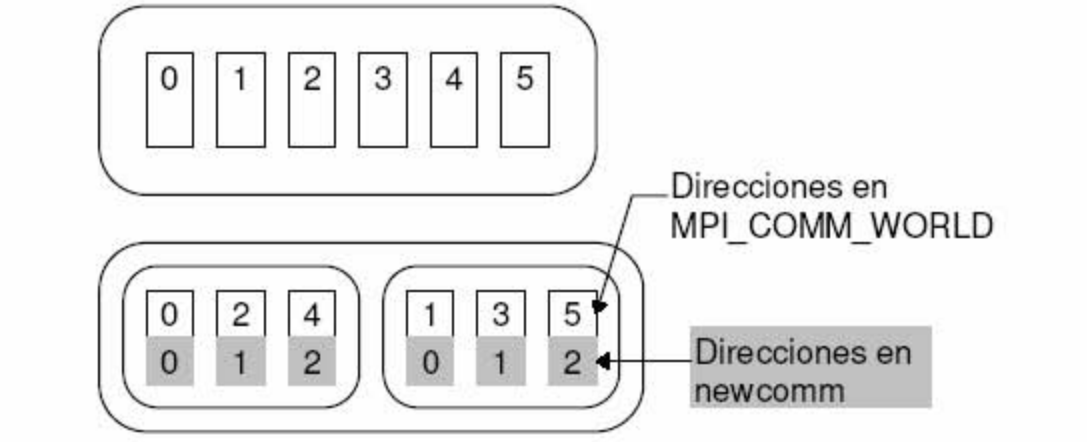


Figura 3.5. Arriba: una colección de procesos en el comunicador `MPI_COMM_WORLD`. Abajo, dos comunicadores: uno con los procesos con dirección global par, y otro con los procesos con dirección global impar.

Los procesos que se encuentran en dos grupos diferentes no pueden comunicarse entre sí, a no ser que se cree un intercomunicador (los comunicadores vistos hasta ahora se denominan intracomunicadores). La función `MPI_Intercomm_create()` nos permite crear un intercomunicador.

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm
*newintercomm);
```

Todos los procesos de los dos comunicadores que quieren interconectarse deben hacer una llamada a `MPI_Intercomm_create()` con argumentos compatibles. Interpretar los argumentos de esta función no es trivial. En primer lugar, cada proceso tiene que facilitar el comunicador (*local_comm*), y nombrar a un proceso local como lider del intercomunicador (*local_leader*).

Luego hay que facilitar un comunicador que englobe a los dos comunicadores que queremos intercomunicar (*peer_comm*; `MPI_COMM_WORLD` siempre resulta válido), y la dirección de un proceso del otro comunicador que va a actuar como líder (*remote_leader*), teniendo en cuenta que esa dirección corresponde al comunicador global. Con esto ya se tiene tendido un "puente" entre los dos comunicadores. También hay que facilitar un número de etiqueta (*tag*) que esté libre y que el programador no debe emplear para ningún otro propósito. El intercomunicador lo obtenemos en *newintercomm*. Una vez que se tiene un intercomunicador, se puede usar para acceder al comunicador que está "al otro lado". Las direcciones de los procesos receptores que hay que facilitar en las funciones de envío, o las de los procesos emisores que se obtienen tras una función de recepción, hacen referencia a ese "otro lado".

El siguiente ejemplo toma como punto de partida los dos comunicadores creados en el ejemplo anterior: uno con los procesos que en `MPI_COMM_WORLD` tienen dirección par, y otro con los que tienen en ese comunicador dirección impar. Cada proceso tiene almacenado en `newcom` el comunicador al que pertenece, y en `myotherid` su dirección en ese comunicador. Tras crear, de forma colectiva, el intercomunicador, cada proceso del comunicador "par" envía un mensaje a su semejante en el comunicador "impar".

```
if (myglobalid%2 == 0)
{
     MPI_Intercomm_create (newcom, 0, MPI_COMM_WORLD, 1, 99,
     &intercomm);
     MPI_Send(msg, 1, type, myotherid, 0, intercomm);
}
else
 {
     MPI_Intercomm_create (newcom, 0, MPI_COMM_WORLD, 0, 99,
     &intercomm);
     MPI_Recv(msg, 1, type, myotherid, 0, intercomm, &status);
 }
```

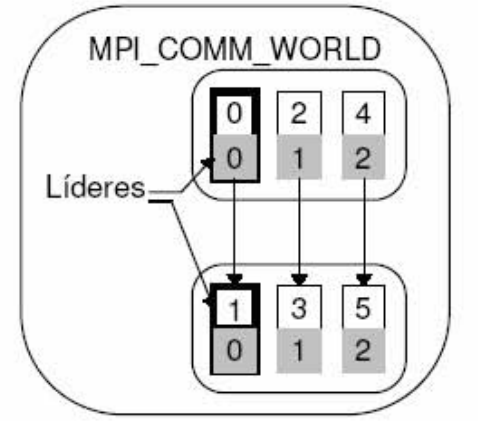


Figura 3.6. Creación de un intercomunicador entre los dos comunicadores de la Figura 3.9

3.2 PVM (*Parallel Virtual Machine*)

3.2.1 Introducción a PVM

PVM es un conjunto integrado de herramientas y bibliotecas de software que emulan un marco de trabajo de computación concurrente de propósito general, flexible y heterogéneo. Todo esto sobre un conjunto de computadoras de distintas o iguales arquitecturas interconectadas entre sí. El principal objetivo del sistema PVM, es permitir que tal conjunto de máquinas, sean usadas en forma cooperativa para hacer computación concurrente o paralela.

PVM crea una nueva abstracción, que es la máquina paralela virtual, empleando los recursos computacionales libres de todas las máquinas de la red bajo la cual se encuentra. Al utilizar programación distribuida, se emplean los recursos de hardware de dicho paradigma; pero se programa el conjunto de máquinas como si fuera una sola máquina paralela.

Este modelo de trabajo permite usar un conjunto de librerías para crear una máquina multiprocesador totalmente escalable, a la cual se le puede aumentar y disminuir el número de procesadores en tiempo de ejecución. Para ello, la máquina oculta la red que estamos empleando, así como las máquinas de la red y sus características específicas. Este conjunto de máquinas constituye una capa que se sitúa por encima del sistema operativo. La figura 3.7 ilustra la estructura de capas de una máquina virtual PVM.

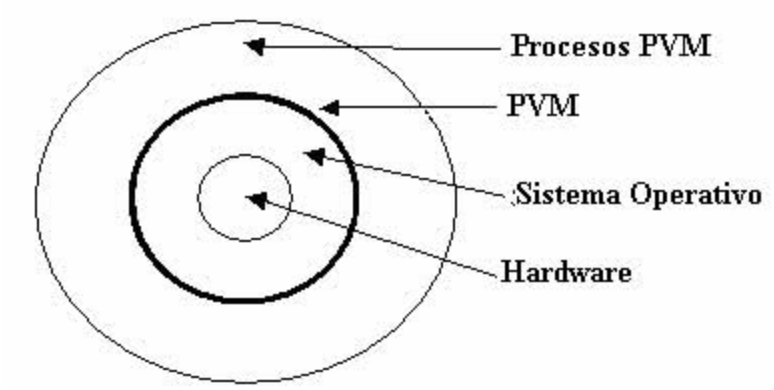


Figura 3.7. Esquema de las distintas capas que integran una maquina virtual

PVM es un software de dominio público desarrollado por el Oak Ridge National Laboratory que promete cumplir los requerimientos que necesita la computación distribuida del futuro, permitiendo escribir programas que utilicen una serie de recursos capaces de operar independientemente y que son coordinados para obtener un resultado global que depende de los cálculos realizados en todos ellos.

El planteamiento anteriormente descrito tiene ventajas con respecto al uso de una supercomputadora. Las más destacadas son:

- *Precio*. Así como es mucho más barata una computadora tradicional que una paralela, un conjunto de computadoras de mediana o baja potencia es muchísimo más barata que la computadora paralela de potencia equivalente. Al igual que ocurre con el caso de computadora paralela, van a existir factores -fundamentalmente, la lentitud de la red frente a la velocidad del bus de la computadora paralela- que van a ser necesarias más computadoras de pequeña potencia que las teóricas para igualar el rendimiento. Sin embargo, aún teniendo esto en cuenta, la solución es mucho más barata.

- *Flexibilidad*. Se adapta a diversas arquitecturas. Permite combinaciones de redes locales con las de área extendida. Cada aplicación "decide" dónde y cuándo sus componentes son ejecutados y determina su propio control y dependencia. Se pueden programar diferentes componentes en diferentes lenguajes. Es fácil la definición y la posterior modificación de la propia Máquina Virtual Paralela.

- *Disponibilidad del equipo que soporte a PVM*. La máquina paralela puede construirse con base en hardware que ya existe. Por lo que con el equipo que se tenga se puede construir un cluster que puede programarse por medio de PVM.

- *Tolerancia a fallos*. Si por cualquier razón falla una de las computadoras que conforman la máquina virtual bajo PVM y el programa fue correctamente diseñado, la aplicación

puede seguir funcionando sin problemas. Esto permite que aplicaciones requieren de una alta disponibilidad del sistema la tengan.

- *Heterogeneidad*. Se puede crear una máquina paralela virtual a partir de computadoras de cualquier tipo. PVM nos va a abstraer la topología de la red, la tecnología de la red, la cantidad de memoria de cada máquina, el tipo de procesador y la forma de almacenar los datos. Esto es importante ya que no se requiere de rutinas de transformación entre los datos que se reciben y se envían. Por último, permite incluir en PVM hasta máquinas paralelas. Una máquina paralela en PVM se puede comportar tanto como una sola máquina secuencial -caso, por ejemplo, del soporte SMP de Linux- o, como ocurre en muchas máquinas paralelas, presentarse a PVM como un conjunto de máquinas secuenciales.

- *Compatibilidad con diferentes plataformas.* La máquina virtual que crea PVM se encuentra para más de una plataforma: Linux, PowerPC con AIX o Sun con Solaris.

El uso de PVM tiene muchas ventajas, pero también tiene una gran desventaja: si el paralelismo que se va a implantar requiere de una fuerte comunicación entre los procesadores, y la comunicación es a través de una red Ethernet, simplemente la red va a dejar de funcionar para todas las aplicaciones -incluido PVM- de la cantidad de colisiones que se van a producir. Si disponemos de una red de tecnología más avanzada, es decir, más cara como ATM, el problema es menor mas sigue existiendo.

La primera versión de las PVM fueron las 1.0, un prototipo diseñado por Vaidy Sunderam y Al Geist en el *Oak Ridge National Laboratory* en 1989. Esta versión fue un desarrollo interno que no se llegó a hacer pública. La versión 2 de la PVM fue realizada en la Universidad de Tennessee y se hizo pública en 1991. En un año se realizaron gran cantidad de aplicaciones para la PVM, lo que produjo una gran realimentación, con las siguientes versiones mejoradas.

Surgieron cuatro versiones posteriores, entre las 2.1 y la 2.4, hasta que en 1993 se reescribió completamente la biblioteca, completando la versión 3 en Febrero de 1993. Hoy en día, la versión actualizada es la 3.3, aunque podemos encontrar la beta de la 3.4.

Más formalmente, podríamos definir PVM como un conjunto de programas y librerías que convierten un conjunto de máquinas heterogéneas en un único multicomputador o *máquina virtual*. Sus ventajas radican en la portabilidad, pues los programas son en lo que a PVM se refiere, portables, y en la diversidad de máquinas que se pueden utilizar. La visión del programador no cambia tanto si la máquina subyacente es un multicomputador de memoria compartida, un cluster de workstations, un multiprocesador de memoria distribuida o cualquier combinación de los anteriores.

No obstante, la portabilidad supone que el código funcionará correctamente independiente de la plataforma que utilicemos para la máquina virtual. Sin embargo, las prestaciones, y por tanto el objetivo de diseño en paralelo, dependerán del hardware subyacente.

3.2.2 El modelo de paso de mensajes.

Aunque la máquina física pueda tener cualquier tipo de estructura, el modelo sobre el que se programa es el de paso de mensajes. Así, un programa paralelo va a consistir de varios procesos independientes que intercambian información a través de mensajes.

El programador ve todos los *hosts* conectados entre si como diferentes nodos independientes.

Cualquier proceso que esté en cualquier nodo podrá enviar un mensaje a otro proceso en cualquier otro. La topología de la *máquina virtual*, de cara al programador, es de tipo totalmente conectada, aunque sus prestaciones correspondan a la arquitectura física subyacente.

Básicamente, PVM permite usar las primitivas básicas; comunicación asíncrona con recepción bloqueante o no, así como multidifusión o "*broadcast*"

3.2.3 Componentes de PVM

PVM se compone de un conjunto de programas y librerías. Los programas integran los procesos demonios gestores de los procesos PVM, la consola que arranca y monitoriza el sistema y diversas utilidades para la compilación y configuración del sistema. Las librerías ofrecen la interfaz para el acceso a los recursos PVM por parte de los programas paralelos PVM.

- **Demonios y librerías.**

Un proceso demonio o *daemon* es un proceso que se arranca en un momento dado y permanece activo sirviendo una serie de peticiones. PVM dispone de un proceso de este tipo por máquina (*pvmd3)*. Estos procesos demonios son los que van a permitir la comunicación entre los procesos PVM independientes del programa paralelo que estemos ejecutando. Cada *daemon* mantiene una tabla de configuración e información de los procesos relativa a su máquina virtual (`/tmp/pvmd.uid`). Los procesos PVM independientes se comunican con el *daemon* local vía la librería de funciones de interfaz. Luego el *daemon* local manda/recibe mensajes de/a *daemons* de los *hosts* remotos.

La creación y puesta en marcha de estos procesos es totalmente automática, activándose en cada *host* con la conexión de éste a la máquina virtual.

La comunicación entre dos procesos difiere en función de la ubicación de éstos. Sin embargo, esto es totalmente transparente para el programador, que no tendrá por qué conocer dónde se encuentra un proceso. No obstante, es interesante saberlo para aprovechar mejor las prestaciones de la máquina.

La figura 3.8 ilustra el mecanismo empleado para la comunicación entre procesos y demonios.

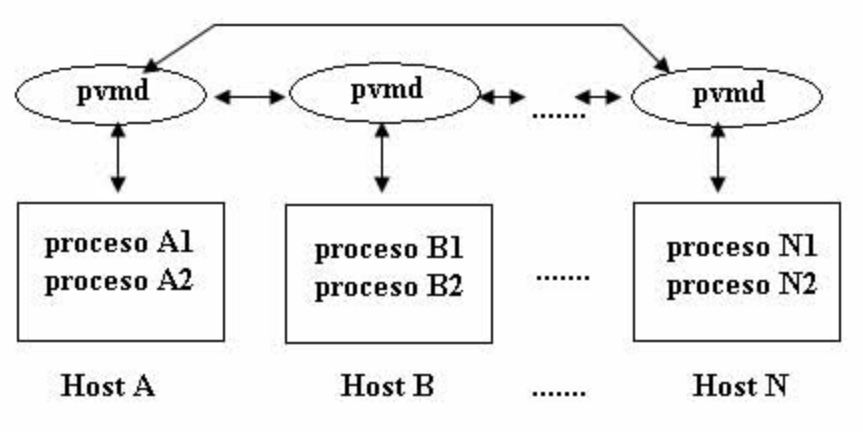


Figura 3.8. Comunicación entre procesos

La comunicación entre dos procesos instalados en un mismo *host* (A1 y A2) se produce de forma directa, sin que intervengan otros procesos. Sin embargo, cuando la comunicación se produce entre dos procesos ubicados en diferentes *hosts*, se realiza a través de los procesos demonios. Esto es debido a que son estos procesos los que conocen el mapeo de los procesos sobre la arquitectura subyacente. El proceso "A1" desconoce el *host* en el que se encuentra el proceso "N1", y debe poder acceder a él de forma transparente. Para ello, se produce la transferencia de datos al demonio de la máquina local. Éste se encargará de transmitir la información al demonio de la máquina remota. Una vez el mensaje se encuentre en el demonio de la máquina destino, el mensaje será transmitido al proceso "N1".

Como dato de interés, la comunicación entre las tareas y los procesos demonios locales se realiza por *sockets*, (en el caso de la implementación estándar). Sin embargo la comunicación entre los demonios se realiza mediante UDP.

Todas las acciones de un entorno de programación paralela de paso de mensajes se realizan mediante llamadas a funciones. Para poder utilizar estas funciones se deben utilizar las librerías que acompañan al paquete. Las librerías permiten escribir programas en "C" y en FORTRAN.

Existe una librería para cada lenguaje (la de "C" permite enlazar con "C++") y una tercera librería para la gestión de grupos. Junto con las librerías se suministran varios ficheros de definiciones (ficheros de cabecera) para la definición de constantes en FORTRAN y "C".

La librería de rutinas de interfaz (`libpvm3.a, libfpvm3.a y libgpvm3.a`)

- *libpvm3.a* - Librería en lenguaje C de rutinas de interfaz: Son simples llamadas a Funciones que el programador puede incluir en la aplicación paralela. Proporciona capacidad de:

Iniciar y terminar procesos, empaqueta, envía y recibe mensajes, sincronizar mediante barreras, conocer y dinámicamente cambiar la configuración de la máquina virtual paralela.
Las rutinas de la librería no se conectan directamente con otros procesos, si no que mandan y reciben la configuración de la máquina virtual paralela.

- *libgpvm3.a*- Se requiere para usar grupos dinámicos.

- *libfpvm3.a*- Para programas en Fortran

En el directorio `/tmp` de cada máquina, se encuentran dos ficheros por usuario que son utilizados por PVM para depositar en ellos cierta información:

*Fichero de Sockets***.** Se nombra como `pvms.número_usuario`. El número de usuario corresponde a un campo dentro del fichero de usuarios del sistema y es único y permanente para cada usuario. Este fichero es necesario para la gestión del *socket* que comunica el demonio con los procesos PVM de la máquina.

Fichero de log. Se nombra como `pvml.número_usuario`. Este fichero es de tipo texto y contiene información relativa a los errores que se han producido en el sistema desde su puesta en marcha. Además permite recoger ciertos mensajes que pueden ser introducidos por el usuario.

Estos dos ficheros se borran cuando se apaga y reanuda el sistema.

- **La consola de PVM**

Otro elemento que compone el entorno PVM es la consola. La consola es un programa que nos permite tanto arrancar y apagar el entorno, así como configurar y monitorizar el sistema, actuando como intérprete de comandos entre el usuario y el *daemon pvmd3*. Utilizando una serie de comandos se puede obtener información sobre el estado de los procesos, las máquinas conectadas, los errores detectados. También permite arrancar o eliminar procesos y añadir o eliminar una máquina al entorno.

La versión X de PVM (*xpvm*) proporciona una serie de herramientas gráficas que permiten configurar la red, arrancar procesos y monitorizar la ejecución de los procesos en el tiempo. *Xpvm* exige la existencia de un servidor de grupos de procesos, *pvmgs*.

La vía más cómoda para obtener la PVM es a través de *netlib*. *Netlib* es tanto una biblioteca de cálculo científico como el *site* que la mantiene. Allí encontraremos la PVM, gran cantidad de rutinas de cálculo numérico y la implementación de algunas de las rutinas de cálculo numérico empleando la PVM.

3.2.4 Modelos de comunicaciones.

El entorno PVM permite el uso de una gran variedad de modelos diferentes de comunicaciones a través de primitivas básicas. El resto de modelos puede implementarse utilizando una combinación de las mismas. La comunicación en PVM permite mensajes de cualquier longitud y no está limitado el tamaño de los *buffers* de recepción.

PVM permite asociar una etiqueta a los mensajes, para poder hacer una recepción selectiva de únicamente los mensajes de un determinado tipo. Igualmente se permite la recepción de los mensajes provenientes de una tarea concreta o bien de cualquier tarea.

El orden de los mensajes está garantizado, es decir, si un proceso envía a otro dos mensajes, el proceso receptor siempre recibirá primero el primer mensaje que el emisor haya enviado.

En cuanto al tipo de mensajes, se pueden clasificar de dos formas distintas

Mensajes en función del número de Participantes.

Atendiendo al número de participantes, PVM permite cualquier combinación. Existen primitivas que permiten el envío de mensajes tanto a un único receptor como a varios receptores. Igualmente, la primitiva de recepción admite recibir de varios emisores distintos, con lo que se puede implementar incluso recepciones *multicast*, es decir, varios emisores y varios receptores.

La figura 2-7 ilustra todas estas posibilidades.

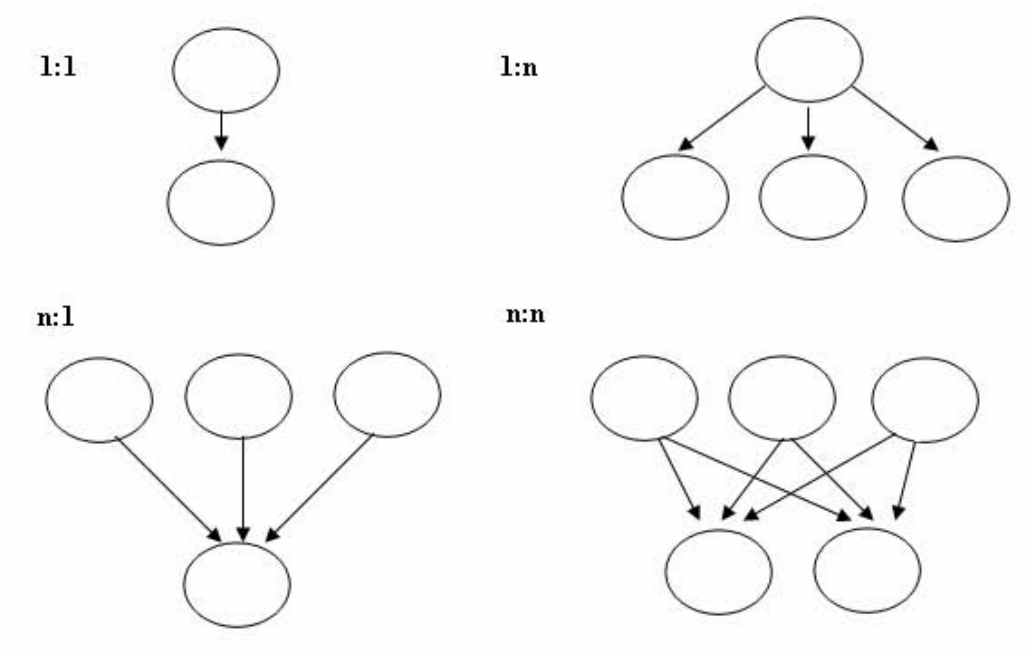


Figura 3.9. Diferentes modos de comunicación en función del número de participantes

Mensajes en función del Receptor.

Atendiendo al comportamiento del proceso receptor, se puede diferenciar varios tipos de comunicación, dependiendo de si éste queda bloqueado indefinidamente esperando el mensaje, espera durante un tiempo máximo o simplemente no espera.

- **Comunicación con Receptor Síncrono.**

El proceso que envía los datos reanuda su ejecución inmediatamente después de haber ejecutado la primitiva de envío del mensaje. La ejecución de la primitiva de recepción supone sin embargo, la suspensión del proceso hasta la llegada del mensaje. Si el mensaje llega con anterioridad a la ejecución de la primitiva, no se produce la espera del proceso (figura 3.10). Es el modelo más clásico de comunicaciones y se utiliza cuando el receptor necesita de los datos para continuar la ejecución.

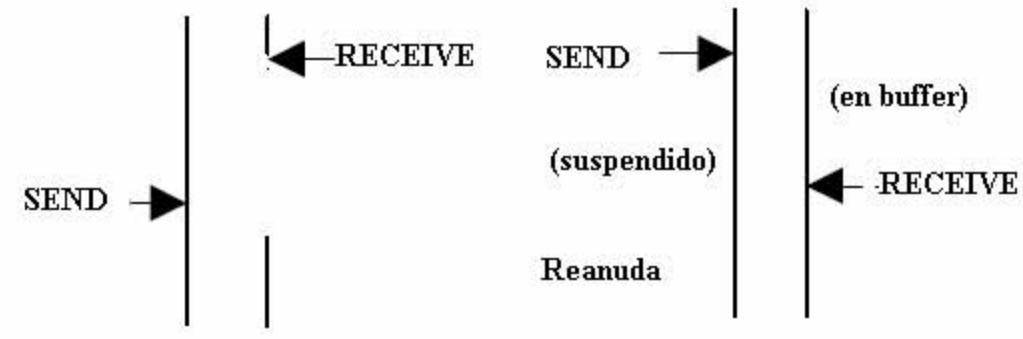


Figura 3.10. Diferentes modos de comunicación en función del número de participantes

- **Comunicación con Receptor Asíncrono.**

El inconveniente de la recepción síncrona es que el receptor queda bloqueado si el mensaje no ha llegado. Podría ser interesante que el proceso receptor no se suspendiera, sino que retornara inmediatamente en el caso de que no se hubiera recibido ningún mensaje.

Esta aproximación podría dejar un poco desatendidos a los procesos clientes. Una forma de reducir este efecto consiste en conjuntar este modelo con el anterior, de forma que la recepción fuera bloqueante pero sólo durante un cierto tiempo, transcurrido el cual, se retornaría a atender el trabajo local.

La figura 3.11 ilustra un esquema de comunicación asíncrono y con vencimiento o timeout.

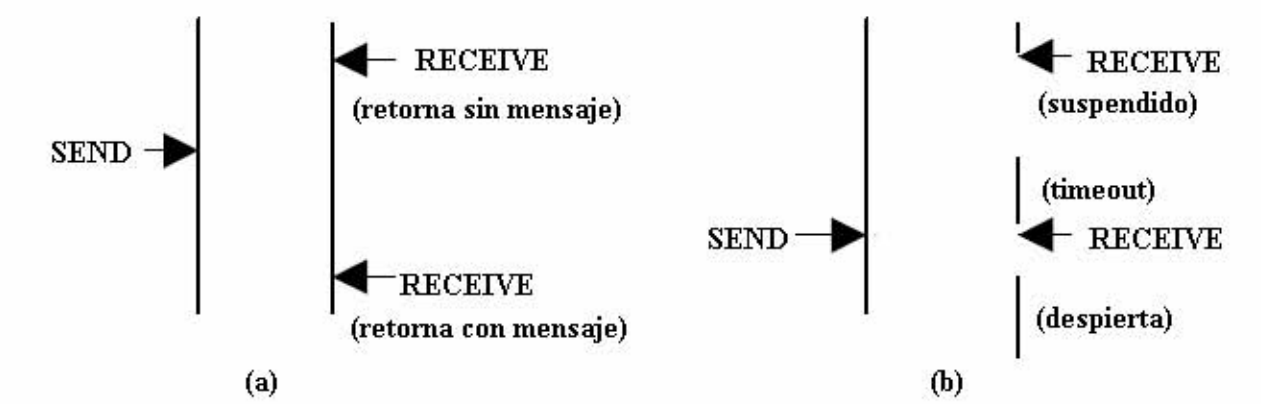


Figura 3.11. (a) Recepción Asíncrona y con timeout (b)
Figura 3.11. (a) Recepción Asíncrona y con timeout (b)

- **Comunicación con Emisión Síncrona.**

PVM no proporciona primitivas para una comunicación síncrona, es decir en la que el emisor queda bloqueado hasta que el receptor efectivamente recibiese o consultase el mensaje. Esto se debe principalmente a que la comunicación entre tareas PVM se realiza a través de un enlace TCP. Este protocolo proporciona una comunicación fiable, con lo que no requiere un mensaje de reconocimiento de que el mensaje ha llegado. Sin embargo , se puede añadir un reconocimiento utilizando primitivas PVM.

3.2.5 Ejemplo programa paralelo en PVM

Este programa intenta calcular el producto escalar de dos vectores de forma paralela. El proceso maestro dispone de los dos vectores y los distribuye a los demás procesos. Éstos, realizan el producto escalar de la parte que les corresponde retornando el resultado al primer proceso. Es un típico ejemplo del modelo de computación masiva. La figura 3.12 ilustra el comportamiento del programa:

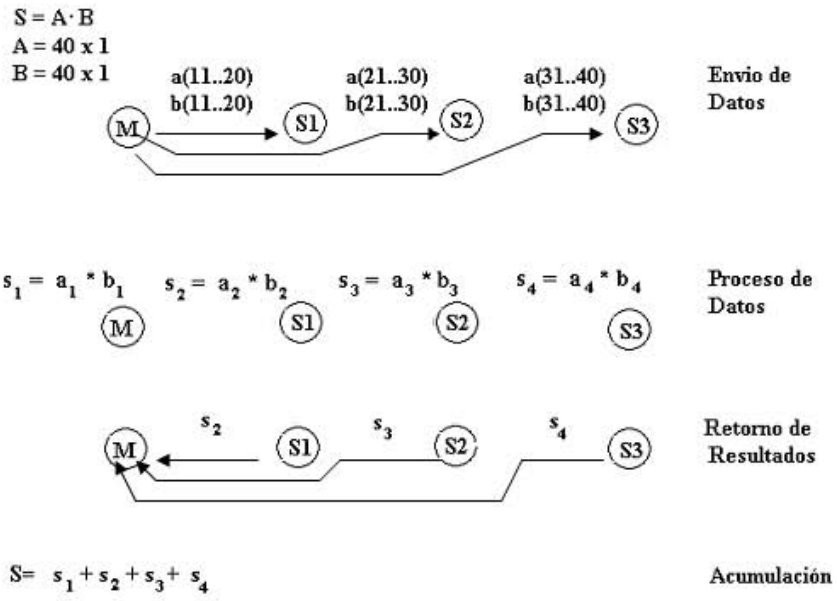


Figura 3.12. (a) Recepción Asíncrona y con *timeout* (b)

Modelo Básico.

Proceso 1:

```
Conectarse_a_PVM;
Crear_Procesos;
Enviar_Datos;
Procesar_Datos;
Recibir_Resultados;
```

Proceso 2:

```
Conectarse_a_PVM;
Recibir_Datos;
Procesar_Datos;
Enviar_Resultados;
```

Modelo Completo en Pseudocódigo.

Proceso 1:

```
mi_tid = Conectarse_a_PVM;
procesos = Crear_Procesos(N, Proceso_2);
PARA i=1,N HACER
Enviar_Datos(IND_DATO, dato[i],procesos[i]);
FPARA
resultado = Procesar_Datos(dato, i);
Recibir_Datos(IND_RESULT, resultado);
```

Proceso 2:

```
mi_tid, padre_tid = Conectarse_a_PVM;
Recibir_Datos(IND_DATO, dato,i);
resultado = Procesar_Datos(dato, i);
Enviar_Datos(IND_RESULT, resultado, padre_tid);
```

Código completo en PVM.

Proceso 1:

```
int mi_tid, procesos, id, i, buffer, rc, tids[N];
double a[N*BLK], b[N*BLK], Resultado, ResAux†;
char *Proceso_2, *argv[];
// Obtener el tid propio y conectarse a PVM
mi_tid = pvm_mytid();
// Crear ëNí Procesos esclavos.
procesos=pvm_spawn(Proceso_2,argv,PvmTaskDefault,NULL,N,tids);
// Enviar los datos iniciales a cada proceso.
for (i=0;i<N;i++)
{
     buffer = pvm_initsend(PvmTaskDefault);
     rc = pvm_pkint( &BLK, 1, 1);
     rc = pvm_pkdouble( &(a[(i+1)*BLK]), BLK, 1);
     rc = pvm_pkdouble( &(b[(i+1)*BLK]), BLK, 1);
     rc = pvm_send( tids[i], IND_DATO);
}
// Procesar la parte de datos propia.
for (i=0;i<BLK;i++)
{
     Resultado = a[i] * b[i];
}
// Recogida de los resultados finales.
for (i=0;i<N;i++)
{
```

```
      buffer = pvm_recv( -1, IND_RESULT);
      rc = pvm_upkdouble( &(ResAux), 1, 1);
      Resultado = Resultado + ResAux;
}
```

Proceso 2:

```
// Obtener tid propio y del padre. Conexion.
mi_tid = pvm_mytid();
padre_tid = pvm_parent();
// Recibir Datos
buffer = pvm_recv( padre_tid, IND_DATO);
rc = pvm_upkint( &BLK, 1, 1);
rc = pvm_upkdouble( ai, BLK, 1);
rc = pvm_upkdouble( bi, BLK, 1);
// Procesar la parte de datos propia.
for (i=0;i<BLK;i++)
{
      miresultado = ai[i] * bi[i];
}
// Enviar los Resultados.
buffer = pvm_initsend(PvmTaskDefault);
rc = pvm_pkdouble( miresultado, BLK, 1);
rc = pvm_send( padre_tid, IND_RESULT);
```

3.2.6 Funciones Básicas.

Se presentan algunas de las funciones de librería más utilizadas en el diseño de programas paralelos utilizando PVM.

Conexión y Desconexión a PVM.

La primera llamada a la librería PVM provoca la conexión del proceso al entorno PVM. Generalmente la primera llamada que se realiza corresponde con la obtención del identificador de tarea que es la clave para referenciar a un proceso en cualquier función PVM.

Todas las funciones PVM retornan un código de error**.** El valor de un código de error siempre es negativo, correspondiendo un valor positivo o cero a una terminación satisfactoria.

```
int tid = pvm_mytid( void )
call pvmfmytid( tid )
```

Un proceso puede conocer su identificador PVM y el identificador del proceso que lo creo. Si el proceso se hubiera lanzado desde la consola, obtendría como identificador del proceso padre, el identificador de la misma. Si el proceso hubiera sido lanzado desde el *prompt* del UNIX, obtendría un código de error (*PvmNoParent*) que le indica acerca de dicha situación.

```
int tid = pvm_parent( void )
call pvmfparent( tid )
```

Cuando un proceso ha acabado de realizar todo el procesamiento PVM que requería, debe abandonar el sistema. Si bien no es necesario, es recomendable que los procesos abandonen PVM antes de finalizar. Esto facilita la gestión de recursos y realiza un uso más óptimo del sistema.

```
int tid = pvm_exit( void )
call pvmfexit( info )
```

La función *pvmfparent* puede aprovecharse para generar un código único para todos los procesos, y establecer un mecanismo cliente-servidor dentro del mismo. El siguiente bloque de pseudocódigo muestra esta idea.

```
call pvmfmytid(tid)
call pvmfparent(dtid)
if (dtid .LT. 0) then
C *** Crea procesos
else
C *** Recibe Datos
endif
Call ProcesaDatos
if (dtid .LT. 0) then
C *** Recibe Resultados
else
C *** Envia Resultados
Endif
```

Creación de procesos hijos.

PVM permite que los procesos paralelos que componen el programa sean creados de forma dinámica durante la ejecución del programa maestro o inicial. La creación dinámica por tanto no requiere que el número de procesos sea especificado ni durante la etapa de diseño ni la carga en el computador, pudiendo incluso variar durante la ejecución para adaptarse a las características del sistema. Mediante la rutina *pvmfspawn* se pude crear un número determinado de instancias de un proceso, especificando además el modo de ejecución y el lugar donde se quiere que se planifiquen estos procesos.

```
int numt = pvm_spawn( char *task, char **argv, int flag, char *donde,
                      int numero, int *tids)

call pvmfspawn( task, flag, donde, numero, tids, numt )
```

La rutina pvmfspawn ejecuta un *numero* **de** procesos utilizando el ejecutable *task* y almacenando los identificadores de las tareas en el vector *tids*. Esta rutina regresa el número de tareas creadas satisfactoriamente, almacenando en el primer elemento erróneo del vector *tids* el código de error en caso de que se produjera.

La creación de procesos no tiene por qué limitarse a una única llamada a la rutina pvmfspawn, puede utilizarse cualquier combinación de ellas.

El parámetro flag nos permite diferenciar:

- Modo de ejecución. Seleccionable mediante las constantes definidas:

  PvmTaskDefault. Modo normal de ejecución.
  PvmTaskDebug. Modo depuración.
  PvmTaskTrace. Modo traza.

- Ubicación de las tareas en la máquina virtual.

  PvmTaskDefault. PVM se encarga de realizar el balance de carga de forma que se distribuyan los procesos de forma equilibrada.
  PvmTaskHost. El argumento ‘donde’ indica el nombre del nodo PVM sobre el que se ejecutarán los procesos creados con dicha orden.
  PvmTaskCompl. El argumento ‘donde’ indica el nombre del nodo PVM que se excluirá a la hora de repartir equilibradamente los procesos en la máquina virtual.
  PvmTaskArch. El argumento ‘donde’ indica la arquitectura a la que pertenecen los nodos donde se ejecutarán los procesos creados distribuyéndose de forma equitativa.

Para especificar varias opciones en la misma llamada a pvmfspawn. Se debe separarlas utilizando el operador **+**, o bien el operador **|**, dependiendo si se trata de FORTRAN o “C” respectivamente.

Todas estas constantes, como cualquier otra presente en el sistema, se encuentran definidas en los ficheros de cabecera del sistema, **`pvm3/include/pvm3.h`** o **`pvm3/include/fpvm3.h`**.

Eliminación de un proceso.

La eliminación abrupta de un proceso puede realizarse mediante la función *pvmfkill*.

Esta función de librería permite eliminar un proceso pvm especificando su identificador. Cualquier proceso puede eliminar a cualquier otro, siempre y cuando pertenezcan al mismo usuario.

```
int info = pvm_kill( int tid )
call pvmfkill( tid, info )
```

Mediante esta función se puede eliminar cualquier proceso independientemente de su estado, liberando los recursos en uso. No obstante, generalmente siempre existe una solución más estructurada que ésta, como la utilización de un mensaje con una etiqueta reservada para este caso.

La tarea que realiza la llamada a *pvmfkill* no puede especificar como parámetro su propio identificador. Deberá recurrir al método estándar de abandono de PVM.

A continuación se presenta un fragmento de código que describe la creación de procesos del programa paralelo para la ordenación en árbol.

```
PROGRAM PARSORT
INTEGER mitid, padretid, htids(2), rc
INTEGER v(MAXN), v1(MAXN/2), v2(MAXN/2)
INTEGER n
CALL pvmfmytid(mitid)
CALL pvmfparent(padretid)
CALL RecDatos(v,n)
IF (n .GT. TAMAÑO_MINIMO) THEN
v1 = v[1:n/2]
v2 = v[n/2+1:n/2]
CALL pvmfspawn(“PARSORT”, PVMTASKDEFAULT,
& “”, 2, htids, rc)
IF (rc .EQ. 2) THEN
```

```
CALL SendDatos(htids(1),v1,n/2)
CALL SendDatos(htids(2),v2,n/2)
CALL RecRes(htids(1),v[1:n/2],n/2)
CALL RecRes(htids(2),v[n/2+1:n/2],n/2)
CALL MergeRes(v,1,n/2+1,n/2)
CALL SendRes(padretid,v,n)
ELSE
WRITE *,”Error creando tareas”
ENDIF
ELSE
CALL QUICKSORT(v,n)
CALL SendRes(padretid,v,n)
ENDIF
CALL pvmfexit(rc)
END
```

Mensajes.

PVM dispone de una rica variedad de primitivas para la comunicación entre procesos.
Estas primitivas permiten comunicación tanto punto a punto como multipunto (difusión), al mismo tiempo que la recepción síncrona y asíncrona.

La emisión de un mensaje siempre conlleva tres etapas: Inicialización del buffer, empaquetado de los datos y envío. La recepción en cambio se divide solamente en dos fases:

Recepción y desempaquetado de los datos. Los mensajes en PVM contienen información de tipo heterogéneo, es decir, se puede empaquetar en un mensaje elementos con tipos de datos diferentes.

Elementos Constitutivos.

Los elementos involucrados en un proceso de comunicación PVM son los siguientes:

- *Tareas.*
  Los puntos de origen y destino de cualquier comunicación PVM son tareas. Estas se especifican mediante su identificador (o varios de ellos en caso de una difusión). El identificador se necesita tanto en la emisión (se indica a qué tarea se envía el mensaje) como en la recepción (se indica desde qué tarea se aceptan los mensajes). En el caso de la recepción

es posible hacer referencia a mensajes provenientes de cualquier tarea indicando como identificador el comodín -1.

- *Buffers.*

Para permitir la recepción de varios mensajes PVM dispone de un sistema de *buffering* o almacenamiento temporal de capacidad ilimitada (limitada por la capacidad de memoria del sistema). Un buffer es un almacén temporal de datos que tiene cabida para un mensaje y se encuentra identificado por un número entero. Los mensajes son accedidos por los procesos PVM a través de estos *buffers*. La recepción de múltiples mensajes es posible gracias a las colas de *buffers* que el sistema PVM gestiona internamente.

Un proceso PVM siempre dispone de forma visible, de como mínimo dos *buffers*, los *buffers* de emisión y recepción por defecto. Estos *buffers* son los que almacenan tanto los mensajes entrantes como el mensaje saliente que se está elaborando. El buffer de recepción por defecto contiene siempre el último mensaje recibido. Cuando se produce una operación de recepción, pasa a almacenar el siguiente, y así sucesivamente. Generalmente no es necesario utilizar mas *buffers* que los por defecto. No obstante, el sistema permite crear más *buffers*, que pueden sustituir a los *buffers* por defecto en un momento dado.

- *Etiquetas.*

Para facilitar la tarea de procesamiento de los mensajes, éstos pueden ser nombrados mediante una etiqueta consistente en un número entero. Esta etiqueta se especifica en la etapa de emisión, y puede ser utilizada en la recepción para que sólo se reciban mensajes correspondientes a una etiqueta determinada. Por supuesto, también podemos hacer uso del comodín para indicar que se reciba cualquier tipo de mensaje.

El uso de comodines en la recepción permite la implementación de esquemas cliente-servidor de manera muy eficiente. Estos esquemas consisten en un proceso servidor que atiende las diferentes peticiones que le llegan en forma de mensajes desde los procesos clientes. Una implementación utilizando pseudocódigo de este sistema viene reflejada en las siguientes líneas.

```
C *** Recibe un msg. De cualquier tipo y procedencia.
Call RecibeMensaje( -1, -1 )
Call ObtenerEtiqueta(EtiquetaMsg)
if (EtiquetaMsg .eq. DATO_REQUERIDO) then
C *** Envia dato
else
if (EtiquetaMsg .eq. RESULTADO) then
C *** Procesar resultado
else
if (EtiquetaMsg .eq. ERROR) then
C *** Tratar Error
```

```
else
...
```

Envío de datos.
La etapa de envío de datos se divide en tres subetapas, completadas con tres tipos de primitivas PVM diferentes.

- **Inicialización del buffer.**

Con anterioridad al empaquetamiento de los datos en el mensaje, se debe vaciar el buffer de emisión para que no contenga datos pertenecientes a mensajes anteriores. La inicialización del buffer se realiza mediante la función *pvmfinitsend*.

```
int bufid = pvm_initsend(int codificacion)
call pvmfinitsend(codificacion, bufid)
```

Los buffers almacenan la información que se va a enviar en el mensaje. Dependiendo del formato con el que se quiera que se almacene dicha información, se dispone de tres modos de almacenamiento, seleccionables a través del argumento **`codificacion`**. Los tres valores que puede tomar dicho argumento son:

PvmDataDefault. Puesto que PVM permite máquinas virtuales compuestas por nodos heterogéneos, es posible encontrarse con nodos que dispongan de diferente representación de los datos e incluso diferente longitud de palabra. Para evitar malas interpretaciones, PVM dispone del formato independiente de la arquitectura XDR.

PvmDataRaw. La codificación en XDR consume bastante tiempo. Cuando ésta no es necesaria, ya sea porque las máquinas son de la misma arquitectura o bien porque tienen el mismo formato para la representación de datos, es más eficiente evitar la codificación, empaquetando los datos en su formato original.

PvmDataInPlace. El proceso de envío copia inicialmente los datos del mensaje saliente al buffer particular del demonio. Los mensajes son leídos por el demonio y copiados en su momento a los *buffers* de emisión del sistema operativo. Si los mensajes que se están enviando contienen *arrays* largos, se esta perdiendo eficiencia al realizar copias innecesarias de los datos. Mediante esta opción, los demonios PVM almacenan los punteros a los datos en lugar de éstos mismos. De esta forma se ahorra gran parte del tiempo en el primer copiado de datos. El problema que esto conlleva es que al pasar los punteros y no los datos, y puesto que la emisión es asíncrona, no se puede modificar los datos que hemos empaquetado hasta que éstos no se encuentren en el buffer de comunicaciones del sistema operativo. Esto no es fácil de precisar, con lo que generalmente no deberán de ser modificados hasta la próxima sincronización. No todos los programas admiten una restricción de este tipo.

- **Empaquetado de datos.**

La fase de empaquetamiento de los datos comienza tras la inicialización del buffer de emisión. Los datos que queremos enviar son empaquetados uno a uno especificando su tipo, número de elementos y la distancia o paso entre elementos. La sintaxis de las rutinas para el empaquetado de datos varían ligeramente entre "C" y FORTRAN.

```
int rc = pvm_pkbyte ( char *cp, int nitem, int paso )
int rc = pvm_pkcplx ( float *xp, int nitem, int paso )
int rc = pvm_pkdcplx ( double *zp, int nitem, int paso )
int rc = pvm_pkdouble( double *dp, int nitem, int paso )
int rc = pvm_pkfloat ( float *fp, int nitem, int paso )
int rc = pvm_pkint ( int *np, int nitem, int paso )
int rc = pvm_pklong ( long *np, int nitem, int paso )
int rc = pvm_pkshort ( short *np, int nitem, int paso )
int rc = pvm_pkstr ( char *cp, int nitem, int paso )
int rc = pvm_packf( const char *formato, ...)
call pvmfpack( tipo, xp, nitem, paso, info )
```

Cada primitiva "C" contiene en su nombre el tipo de datos que empaqueta, mientras que en FORTRAN sólo existe una primitiva. Esta primitiva admite un argumento parametrizado para especificar el tipo de los datos a empaquetar. Esto permite la revisión de tipos del compilador en los parámetros de una llamada "C". Los valores de las constantes para los tipos en FORTRAN son:

```
STRING BYTE1 INTEGER2 INTEGER4 REAL4 COMPLEX8 REAL8 COMPLEX16
```

Adicionalmente, hay una primitiva "C" que permite el empaquetamiento de datos utilizando el formato de las rutinas de entrada y salida estándar como *printf*.

El incremento o paso utilizado para referenciar los elementos de un *array* puede resultar muy provechoso en el envío de elementos que no se encuentran almacenados de forma contigua, como las filas de una matriz en un programa FORTRAN. En el siguiente fragmento de código se ilustra el empaquetamiento de la fila i-ésima de una matriz.

```
Real matriz(NFilas, NCols)
...
call pvmfpack(REAL8, matriz(i,1), NCols, NFilas, rc )
...
```

- ***Envío del mensaje generado.***

Una vez ya se han empaquetado los datos pertinentes, el mensaje puede ser enviado al destino o destinos que se especifique. Es éste el momento para asignarle una etiqueta para un mejor procesamiento en su recepción.

PVM aporta dos primitivas diferentes en función del número de destinos para la emisión de mensajes: La primitiva *pvmfsend* permite la comunicación punto a punto y la primitiva *pvmfmcast* la comunicación multi-punto.

```
int info = pvm_send( int tid, int etiqueta )
call pvmfsend( tid, etiqueta )
int info = pvm_mcast( int *tids, int ntask, int etiq. )
call pvmfmcast(ntask, tids, etiqueta, info )
```

A partir del momento de la ejecución de cualquiera de estas primitivas, el mensaje se envía al demonio, con lo que el buffer de emisión por defecto queda libre para un nuevo mensaje.

El envío por difusión, aunque no consiste en una implementación totalmente óptima, si produce resultados mejores que el envío individual de datos. Hasta en un 75% menos de tiempo comparado con los correspondientes tiempos de los envíos individuales.

Recepción de datos.

El proceso de recepción de datos se divide en dos etapas bien diferenciadas: La recepción, síncrona o asíncrona del mensaje y el desempaquetamiento de los datos. Ésta segunda etapa tiene una estructura muy similar a la ofrecida por el empaquetamiento de los datos.

- **Recepción del Mensaje.**

La recepción de un mensaje presenta diferentes posibilidades:

Recepción Síncrona.

El proceso que ejecuta la orden de recepción queda suspendido hasta que el mensaje llega (si éste no había sido recibido con anterioridad a la recepción). Es el proceso más habitual por la sincronización que conlleva.

```
int pvm_recv( int tid, int etiqueta)
call pvmfrecv( tid, etiqueta, rc )
```

En la recepción puede especificar de qué proceso y qué tipo de mensaje se espera recibir, permitiéndose el uso del comodín -1 para especificar cualquier instancia en cada caso.

Recepción Asíncrona.

Si el proceso que realiza la recepción de datos no requiere de los datos para continuar la ejecución, puede realizar otras tareas mientras espera la llegada de los datos. Para ello puede utilizar dos tipos distintos de primitivas PVM.

```
int pvm_nrecv( int tid, int etiqueta)
call pvmfnrecv( tid, etiqueta, rc )
```

```
int pvm_probe( int tid, int etiqueta)
call pvmfprobe( tid, etiqueta, rc )
```

La primitiva *pvmfnrecv* retorna inmediatamente ya sea con el mensaje recibido, o bien con un código de error que indica que no había ningún mensaje en el buffer. La primitiva *pvmfprobe* retorna inmediatamente indicando si se ha recibido un mensaje o no, pero sin 'recibirlo', es decir, se necesita una llamada a *pvmfrecv* ó *pvmfnrecv* posteriormente para poder empezar a procesar el mensaje. Esta última primitiva permite que se llame varias veces para un mismo mensaje.

Para encontrar un ejemplo que aclare el punto anterior, debemos complicar un poco el sistema. Las siguientes líneas de código corresponden a una recepción con prioridades.

El planteamiento es el siguiente: Se dispone de dos prioridades de mensajes, los mensajes con la etiqueta A con prioridad 1 y los mensajes con etiquetas B y C con prioridad 2. Si sólo hay un mensaje, se recibirá éste independientemente de su prioridad. Si hay un mensaje de cada prioridad, el de más prioridad primero, y si hay dos de la misma prioridad, se escogerá al azar uno cualquiera de los dos.

```
...
CALL pvmfprobe(-1, ETQ_B, bufB)
CALL pvmfprobe(-1, ETQ_C, bufC)
IF ((bufC .GT. 0) .AND. (bufB .GT. 0)) THEN
IF (azar(0,1) .EQ. 0) THEN
CALL pvmfrecv(-1,ETQ_B, bufid)
ELSE
CALL pvmfrecv(-1,ETQ_C, bufid)
ENDIF
ELSE
IF (bufB .GT. 0) THEN
CALL pvmfrecv(-1,ETQ_B, bufid)
ELSE
IF (bufC .GT. 0) THEN
CALL pvmfrecv(-1,ETQ_C, bufid)
ENDIF
ENDIF
ELSE
CALL pvmfprobe(-1, ETQ_A, bufA)
IF (bufA .GT. 0) THEN
```

```
CALL pvmfrecv(-1,ETQ_A, bufid)
ENDIF
ENDIF
...
```

En definitiva, `pvmfprobe` permite decidir qué mensaje de los recibidos va a ser procesado primero.

Recepción Asíncrona con Vencimiento

Como alternativa a la recepción totalmente asíncrona o totalmente síncrona, PVM ofrece un tercer modelo de recepción mixto mediante la función *pvmftrecv*. El proceso que realiza una llamada a esta función se suspende, ya sea hasta que llega un mensaje del tipo y emisor requerido, o bien hasta que transcurre un determinado tiempo especificado en la llamada. De esta forma, puede dar una prioridad relativa al procesamiento de datos locales frente al proceso de las comunicaciones.

```
int buf = pvm_trecv(int origen, int etiqueta,struct timeval
          *tiempo)

call pvmftrecv(origen, etiq, sec, usec, buf)
```

La recepción con *timeout* resulta muy interesante cuando el proceso receptor tiene que realizar antes de continuar con su ejecución a la finalización de la recepción, algún tipo de postprocesamiento de los resultados que está recibiendo. Si los procesos emisores tienen problemas de sincronización y producen tiempos inactivos en el proceso receptor, se puede utilizar el esquema mostrado en las líneas de código siguientes correspondientes a un proceso maestro que recibe el resultado generado por N procesos esclavos. Si tras esperar un tiempo de un segundo no se ha recibido ningún mensaje, el proceso maestro pasa a aplicar el postprocesamiento al primero de los bloques de datos que hubiera recibido, alternando espera con postprocesamiento.

```
IndPend = 1
Pendientes = 0
i = 1
100 continue
if (pendientes .eq. 0) then
call pvmfrecv( -1, IND_RES, bufid)
call pvmfunpack( XXX, resultado(i), rc)
i = i + 1
pendientes = pendientes + 1
```

```
else
call pvmftrecv( -1, IND_RES, 1, 0, bufid)
if (bufid .eq. 0) then
call procesa(resultado(indPend))
pendientes = pendientes - 1
indPend = indPend+1
else
call pvmfunpack( XXX, resultado(i), rc)
i = i + 1
pendientes = pendientes + 1
endif
endif
if (i .lt. N) goto 100
do i = indPend,N
call procesa(resultado(indPend))
enddo
```

Desempaquetado.

La etapa de desempaquetado consiste en la recuperación de la información contenida en el mensaje que ha sido recibido en el buffer de recepción por defecto. Su funcionamiento es análogo al de la etapa de empaquetado. La sintaxis de las primitivas es idéntica.

```
int rc = pvm_upkbyte ( char *cp, int nitem, int paso)
int rc = pvm_upkcplx ( float *xp, int nitem, int paso)
int rc = pvm_upkdcplx ( double *zp, int nitem, int paso)
int rc = pvm_upkdouble( double *dp, int nitem, int paso)
int rc = pvm_upkfloat ( float *fp, int nitem, int paso)
int rc = pvm_upkint ( int *np, int nitem, int paso)
int rc = pvm_upklong ( long *np, int nitem, int paso)
int rc = pvm_upkshort ( short *np, int nitem, int paso)
int rc = pvm_upkstr ( char *cp, int nitem, int paso)
int rc = pvm_upackf( const char *formato, ...)
call pvmfunpack( tipo, xp, nitem, paso, info )
```

Es importante asegurarse de que los datos contenidos en el mensaje son desempaquetados especificando el mismo orden, número y tipo de datos, puesto que puede producirse una mala interpretación del contenido del mensaje si utilizamos primitivas de diferente tipo o con tamaños diferentes. De nuevo, los valores de las constantes para los tipos en FORTRAN son:

```
STRING BYTE1 INTEGER2 INTEGER4 REAL4 COMPLEX8 REAL8 COMPLEX16
```

El proceso de recepción genera pocos problemas en cuanto a lo que PVM se refiere. La mayoría de los problemas aparecen como consecuencia de fallos en el diseño que producen interbloqueos o situaciones similares.

CAPÍTULO 4

DESARROLLO DE APLICACIONES

4.1 Aplicaciones Numéricas

4.1.1 Multiplicación de Matrices

Debido a la sencillez que se tiene al utilizar el algoritmo de la multiplicación de matrices, se podrá llevar acabo la implementación de este algoritmo utilizando tanto PVM como MPI. En general la multiplicación de matrices no es conmutativa y ésta sólo se puede realizar si las matrices cumplen cierta condición. Si el número de columnas en la matriz *B* es igual al número de renglones en la matriz *A*, las matrices se denominan conformables, y pueden multiplicarse en el orden *B x A.* Específicamente, si B es una matriz de dimensiones *(q,n)* y *A* tiene dimensiones *(m,p)* el producto será una matriz *(q,p)*, es decir:

$$(q,n)\ x\ (m,p) = (q,p)$$

Una vez conocidas las dimensiones de la matriz resultado, para obtener cada elemento de la misma se procede como sigue:

Para obtener el elemento *i* en la columna *j* del producto *P = BA*, se selecciona el *iésimo* renglón de *B* y la *iésima* columna de *A*, y se suman los productos de sus elementos correspondientes, iniciando en el extremo izquierdo y la parte superior, respectivamente. Así :

$$P_{ij} = \sum_{r=1}^{n} B_{ir} A_{rj}$$

Se observa que la multiplicación de matrices es un problema muy fácilmente paralelizable; de hecho, se encuentra en la categoría de problemas conocidos como "vergonzosamente paralelizables". Esto es porque cada nodo que participe en el cálculo únicamente necesita conocer, antes de iniciar, los valores de las dos matrices a multiplicar; y en ningún momento requerirá comunicarse con los demás nodos para realizar su trabajo.

La implementación más sencilla involucra comunicación con un proceso "maestro o padre" que recoge y consolida los resultados parciales de los nodos, pero en ningún momento se requerirá que los nodos detengan su cálculo para esperar información de otro nodo.

Una sencilla implementación de multiplicación de matrices en un solo procesador

La siguiente implementación de la multiplicación dos matrices, tanto el programa secuencial como los programas de MPI y PVM, fue realizada por el Ing. Manrique Martínez Daniel, alumno de la Facultad de Ingeniería, UNAM, y fue presentada en su tesis que lleva por título *"Construcción y evaluación de desempeño de un cluster tipo Beowulf para cómputo de alto rendimiento"* en el año 2001.

El programa llamado *matrix.c* contiene las rutinas para el manejo de las matrices. El programa llamado *unimatrix.c* contiene el cálculo de la multiplicación y únicamente puede operar sobre matrices cuadradas de igual dimensión, que contengan elementos enteros. Adicionalmente, el programa no acepta datos de entrada, sino que genera las matrices de forma aleatoria.

Aún con estas restricciones, el programa es suficiente para presentar una implementación de la multiplicación de matrices, así como evaluar su desempeño. El programa acepta parámetros en su línea de comandos para definir la dimensión de las matrices, lo cual es importante para variar la cantidad de trabajo que se debe realizar; así como para imprimir a pantalla tanto las matrices a multiplicar como su resultado. Esto es básicamente con fines de verificación de resultados, y es poco práctico para matrices de dimensiones superiores a 10 x 10 pues es difícil visualizar éstas en una pantalla común.

A continuación se muestran los códigos secuenciales escritos en lenguaje C para la multiplicación de dos matrices cuadradas:

matrix.c

```
1    /*Funciones para el manejo de matrices*/
2    /* Daniel Manrique */
3
4    #include <stdio.h>
5    #include <stdlib.h>
6
7    //rellena una matriz aleatoriamente
8    void randomatrix(int *matrix, int row, int col, int maxval)
9
10   {
11      int i;
12
13      for(i=0; i<(row * col); i++){
14           *(matrix + i)= 1 (int)((float)maxval*random()/
15           (RAND_MAX + 1.0));
16
17      // printf("elemento %d es %d\n",i,*(matrix + i));
18
19      }
20
21   }
22
```

```
23
24   //dado un arreglo lineal lo trata como matriz y regresa el
25   //elemento specificado. row, col son iniciados en 0 es decir
26   //si el primer elemento es 0,0
27
28   int matrix_get_cell(int *matrix, int rows, int cols, int x,
29                       int y)
30   {
31      int valor_lineal;
32      valor_lineal = (y * cols + x);
33      return matrix[valor_lineal];
34   }
35
36   //similar pero el valor del elemento dado en la matriz
37
38   int matrix_set_cell(int *matrix, int rows, int cols, int x,
39                        int y, int val)
40
41   {
42      int valor_lineal;
43      valor_lineal=(y * cols + x);
44      matrix[valor_lineal] = val;
45      return 0;
46   }
47
48
49   // presentación de matrices
50   int matrix_print_linear(int *matrix, int rows, int cols)
51
52   {
53      int x,y;
54      for( x=0; x<cols; x++){
55           for(y=0; y< rows; y++){
56                printf("%d,%d es %d\n",x,y,
57                       matrix_get_cell(matrix,rows,cols,x,y));
58           }
59      }
60      return 0;
61   }
62
63   int matrix_print(int *matrix, int rows, int cols)
64   {
65      int x,y;
66      for(y=0; y < rows; y++){
67           for(x=0; x< cols; x++){
68           printf("%d7d",matrix_get_cell(matrix, rows,cols,x,y));
69           }
70           printf("\n");
71      }
72      return 0;
73   }
```

unimatrix.c

```
1    /*Programa multiplicación de matrices uniprocesador*/
2    /* Daniel Marique */
3
4    #include "matrix.c"
5    #include <stdio.h>
6    #include <math.h>
7    #include <stdlib.h>
8    #include <unistd.h>
9    #include <sys/time.h>
10
11   int main(int argc, char **argv)
12   {
13      int rv, row, col;
14      int *matrix1=NULL;
15      int *matrix2=NULL;
16      int *matrix3=NULL;
17      int x, y, i, k;
18      int elem1, elem2, suma;
19      int dimension;
20      int seed = (unsigned int) (time(0)/2);
21      char optchar;
22      struct timeval starttime, endtime;
23      double dstart, dend;
24      int opt_print = 0;
25
26
27   //Este ciclo procesa las opciones de la línea de
28   //comandos y fija las banderas necesarias
29   //del programa.
30
31   do {
32      optchar= getopt(argc, argv, "fpd:");
33      switch (optchar){
34           case 'p':
35                printf("imprimiendo matrices \n");
36                opt_print=1;
37                break;
38           case 'd':
39                dimension = atoi(optarg);
40                printf("dimension %d \n", dimension);
41                break;
42           case 'f':
43                printf("matriz fija\n");
44                seed = 1;
45                break;
46      }//switch
47   }//do
48   while (optchar != -1);
49
```

```
50      if(dimension == 0){
51           printf("dimension 0, pos no funcionara \n");
52           return 0;
53      }//if
54    row = dimension;
55    col = dimension;
56
57
58    srandom (seed);
59
60    //Asignar memoria para tres matrices de row x col
61
62    matrix1= malloc (row * col * sizeof(int));
63    matrix2= malloc (row * col * sizeof(int));
64    matrix3= malloc (row * col * sizeof(int));
65      if (matrix1 == NULL || matrix2 == NULL
66          || matrix3 == NULL){
67           if(matrix1!=NULL) free(matrix1);
68           if(matrix2!=NULL) free(matrix2);
69           if(matrix3!=NULL) free(matrix3);
70           exit(1);
71      }
72
73    // llenamos las dos primeras matrices aleatoriamente
74    // la definición de la función randomatrix está
75    // en el archivo matrix.c
76    randomatrix(matrix1, row, col ,5);
77    randomatrix(matrix2,row, col, 5);
78
79    //obtener tiempo de inicio
80    rv= gettimeofday(&starttime,NULL);
81
82    // mostrar matrices
83      if (opt_print){
84           matrix_print(matrix1, row, col);
85           printf("\n");
86           matrix_print(matrix2, row, col);
87      }
88      // Ir calculando renglón por renglón
89      for (i=0;i < row; i++){
90           printf("calculando row %d \n",i);
91           /*el elemento que se está calculando tiene las
92           coordenadas y,x entonces lo que varía es x
93           porque y es fija por ser el reglón*/
94           y= i;
95      // aquí se calcula cada celda
96           for (x=0; x < col; x++){
97                //printf("calculando elemento (%d,%d)\n",x,y);
98
99                suma = 0;
100               for(k=0; k< col; k++){
```

```
101                     elem1 =
102                             matrix_get_cell(matrix1, row, col,
103                                             k,y);
104                     elem2 =
105                             matrix_get_cell(matrix2, row, col,
106                                             x,k);
107                     suma += elem1 * elem2;
108               // printf ("%d* %d + ", elem1,elem2);
109
110               }//for
111               matrix_set_cell(matrix3,row, col, x, i, suma);
112     }//for
113  }//if
114
115  // se termina el calculo y se obtiene el tiempo
116  rv = gettimeofday(&endtime, NULL);
117
118  // Imprimir matrices
119     if (opt_print){
120          matrix_print(matrix3, row, col);
121     }//if
122  //Calculamos el tiempo que tardamos y  mostramos
123
124     dend = (double) endtime.tv_sec;
125     dend += (double) endtime.tv_usec * 0.000001;
126
127     dstart = (double) starttime.tv_sec;
128     dend += (double) starttime.tv_usec * 0.000001;
129
130     printf("wall clock time = %f\n",(dend-dstart));
131     if(matrix1 != NULL) free (matrix1);
132     if(matrix2 != NULL) free (matrix2);
133     if(matrix3 != NULL) free (matrix3);
134     return 0;
135  }//main
```

La descripción del programa *unimatrix.c* se describe a continuación:

Una vez asignando espacio para las dos matrices generadas aleatoriamente (líneas 62-77), el programa toma una muestra de la hora actual, con precisión hasta microsegundos (línea 80). Posteriormente procede a recorrer cada renglón de la matriz (ciclo que inicia en la línea 89). Para cada renglón, otro ciclo realiza el resultado en cada celda (línea 96), empleando un contador (línea 100) para multiplicar cada elemento del renglón de la primera matriz por su correspondiente en la columna de la segunda matriz y acumulando para finalmente obtener el resultado de la celda correspondiente. Los resultados se almacenan en una tercera matriz. Al finalizar el cálculo se toma otra muestra de la hora actual (línea 116), se calcula el tiempo de operación a partir de las dos muestras tomadas, y se despliega este valor (líneas 124-130).

Una corrida ejemplo de este programa, con una matriz de dimensión 150 (es decir, 150 x 150, o 22,500 elementos) es como sigue:

```
$ gcc unimatrix.c
$ ./a.out -d 150
dimension 150
calculando row 0
calculando row 1
calculando row 2
calculando row 3
calculando row 4
calculando row 5
.
.
.
calculando row 145
calculando row 146
calculando row 147
calculando row 148
calculando row 149
wall clock time = 3.795671
```

Implementación paralela. Elección de organización

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

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

Otro esquema muy utilizado es el cálculo en horda (*herd computing*). Aquí, todos los procesos son jerárquicamente iguales y comparten información entre ellos para alcanzar la solución final. Comúnmente la consolidación de resultados se realiza al final del cálculo, ya que todos los procesos han completado sus unidades de trabajo.

También está presente la división funcional de trabajo, en la cual cada nodo realiza una tarea diferente, a diferencia de los esquemas anteriores donde los nodos ejecutan básicamente el mismo proceso, pero sobre distintas porciones de los datos a manipular. El esquema de división funcional puede equipararse burdamente a una línea de producción industrial, donde en cada estación se realiza una tarea diferente de las demás.

Obsérvese la especificación del tamaño de la matriz por medio del parámetro `-d`. En esta corrida ejemplo se aprecia que el tiempo de ejecución es de 3.79 segundos.

En general es complicado dar una idea de cuál es la manera más eficiente de organizar y dividir el trabajo. Esto está determinado en gran medida por el algoritmo con que se va a atacar el problema, que podrá prestarse a alguna de las formas de organización antes mencionada. De hecho, tanto la elección del algoritmo como de la organización y división de trabajo son tareas que requieren experiencia e intuición, proporcionando el aspecto más "artístico" del cómputo en paralelo.

En el caso del problema de la multiplicación de matrices propuesto, se optó por una organización maestro-esclavo. La operación detallada del algoritmo paralelo se describe en la siguiente sección.

El algoritmo en paralelo

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

El primer nodo se constituye como maestro, generando las matrices aleatorias y distribuyéndolas a los procesos esclavos por medio de un mecanismo de *broadcast*.

Los esclavos reciben las matrices a multiplicar, y determinan el número de procesos que participan en el cálculo, así como su lugar en la lista de procesos. Con esta información cada proceso puede decidir cuáles renglones debe resolver. En particular, los nodos se distribuyen equitativamente los renglones, dejando al último nodo los renglones restantes o residuo.

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

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

Figura 4.1 Tabla de asignación de trabajo
para la multiplicación de matrices.

Una vez que se han recibido las matrices a operar y se conoce el bloque de renglones asignados, cada nodo esclavo procede a calcular, entregando los resultados parciales al nodo maestro; estos resultados se envían a través de un mensaje punto a punto (mecanismo `send`). El nodo maestro recibe estos resultados por medio de un mecanismo `receive`, contabilizando el número de renglones resueltos, y cuando se tiene la matriz completamente resuelta, el proceso se da por terminado.

Este programa realiza el mismo proceso de medición de tiempo de cálculo que se efectúa en la versión uniprocesador, a fin de tener valores para comparar el desempeño. Cabe hacer notar que el muestreo de tiempo toma en cuenta el tiempo empleado en la transmisión de las matrices (*broadcast*) pues este tiempo efectivamente forma parte de la realización del cálculo con paralelismo.

Implementación

Se implementó el programa de multiplicación de matrices tanto en PVM como en MPI. En ambos casos el algoritmo utilizado para el cálculo es el mismo, y únicamente cambian las porciones relevantes a la inicialización, utilización y terminación de la biblioteca de paso de mensajes correspondiente.

Implementación con MPI

El programa de multiplicación de matrices paralelo utilizando MPI se muestra al final. El primer paso para utilizar MPI en un programa es inicializar el mecanismo de paso de mensajes. Esto se hace por medio de la llamada `MPI_Init` (línea 31). A continuación el proceso determina el tamaño de su comunicador (cuántos procesos lo componen) con una llamada a `MPI_Comm_size` (línea 36) . El comunicador, o contexto de comunicaciones, es el grupo de trabajo básico de MPI; en general, un comunicador delimita el alcance de llamadas a funciones grupales, como *broadcasts* y *scatterlgather*. Esto facilita la organización y distribución de trabajo. En este caso, se utiliza el comunicador `MPI_COMM_WORLD`, que es el comunicador al que pertenecen inicialmente todos los procesos. Obviamente cada proceso puede posteriormente cambiar a otro comunicador, pero en este caso es suficiente el uso de `MPI_COMM_WORLD`

Se determina también el rango del proceso dentro del comunicador (es decir, qué posición ocupa de entre los procesos que componen el comunicador) y el nombre del procesador en que se está ejecutando, llamando a `MPI_Comm_rank` y `MPI_Get_processor_name` (líneas 36 y 37). El rango es de particular utilidad para determinar cuáles renglones debe resolver cada proceso.

A continuación se realiza la bifurcación de trabajo en el proceso; es decir, se designa un proceso para que asuma el papel de maestro, mientras los demás se configuran como esclavos. Basándose en el parámetro `localid`, el rango dentro del comunicador, a partir de la línea 74 y hasta la 169 se encuentra el trabajo que realiza el proceso maestro; de la línea 170 a la 247 se encuentran las instrucciones para los procesos esclavo.

En este caso el criterio de decisión es designar como maestro al proceso que tiene localidad de 0. En MPI esta decisión es un tanto arbitraria, en realidad se puede escoger cualquier proceso como maestro, sin embargo el utilizar al proceso 0 es por convención. Esto obedece al hecho de que en MPI no se tiene el concepto inherente de "proceso padre".

El proceso padre genera las matrices aleatorias y las transmite a todos los procesos del comunicador `MPI_COMM_WORLD`. Esto se hace por medio de dos llamadas consecutivas a `MPI_Broadcast`. Aquí se indica la dirección de los datos a transmitir, la cantidad de información que se desea enviar, el rango del proceso raíz, y el comunicador. El parámetro de proceso raíz indica cuál proceso va a iniciar el *broadcast*.

Nótese que en general, las llamadas a primitivas de comunicación en MPI requieren especificar la dirección de los datos a enviar o recibir, así como el tamaño o cantidad de información a comunicar.

Una vez enviadas las matrices, el proceso padre no realiza ninguna tarea de cálculo, únicamente espera a recibir los renglones completos por parte de los nodos. Esto se hace en un ciclo que contabiliza el número de renglones recibidos (línea 29), Y en cada iteración realiza una llamada a `MPCRecv` (línea 135). Esta llamada bloquea en espera de recepción de un mensaje de cualquier proceso del comunicador (parámetros `MPI_ANY _SOURCE` y `MPCCOMM- WORLD`).

Al terminar de recibir los renglones el proceso padre muestra el tiempo empleado en el cálculo y termina su ejecución.

A partir de la línea 170, los procesos esclavo (con localidad diferente de 0) asignan memoria para las matrices a operar y reciben sus valores por medio de *broadcast*. Obsérvese que en MPI la llamada a `MPCBroadcast` es igual para recibir información. Los procesos cuyo `localid` sea diferente al especificado en la llamada, recibirán la información del proceso que inició el *broadcast*.

En las líneas 200-209, los procesos esclavo determinan la sección de la matriz que deben resolver.

Una vez teniendo la información necesaria, los nodos comienzan a resolver su porción de la matriz, guardando el resultado parcial de cada renglón en un arreglo y enviándolo al proceso maestro cuando se ha completado un renglón. Esto se hace por medio de una llamada a `MPI_Send` (línea 238). Esta función toma como parámetros la dirección y tamaño de la información a enviar, el tipo de datos (en este caso `MPI_INT`), el proceso destino (en este caso 0, el proceso padre), una bandera identificadora de mensaje y el comunicador al que se debe enviar el mensaje.

Los nodos continúan con este proceso hasta terminar el cálculo de sus renglones asignados, en este momento terminan su ejecución.

MatrixMPI.c

```
1  /* Programa de multiplicación de matrices paralelizado
2  utilizando MPI. Daniel Manrique */
3
4  #include "mpi.h"
5  #include "matrix.c"
6  #include <stdio.h>
7  #include <math.h>
8  #include <stdlib.h>
9  #include <unistd.h>
10
11 int main(int argc, char **argv)
12 {
13   int localid, numprocs, namelen, rv, row, col;
14   int *matrix1 = NULL;
15   int *matrix2 = NULL;
16   int *matrix3 = NULL;
17   double startwtime, endwtime;
18   char processor_name[MPI_MAX_PROCESSOR_NAME];
19   int x, y , i ,k;
20   int partitions, firstrow, lastrow, rowstodo;
21   int *resultrow;
22   int elem1, elem2, suma;
23   int completerows;
24   int dimension;
25   char optchar;
26   int opt_print = 0;
27   int seed = (unsigned int) (time (0)/2);
28   MPI_Status status;
29
30 // Inicializar MPI
31   MPI_Init(&argc,&argv);
32
33 /* determinar el numero total de procesos e identificador,
34  así como en que procesador (nodo) se esta ejecutando */
35
36   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
37   MPI_Comm_rank(MPI_COMM_WORLD,&localid);
38   MPI_Get_processor_name(processor_name,&namelen);
```

```
39
40   printf("soy el proceso %d de %d en %s \n",localid, numprocs,
41         processor_name);
42 //Este ciclo procesa las opciones de la línea de comandos
43 //y fija las banderas necesarias del programa
44
45   do {
46         optchar = getopt(argc, argv, "fpd:");
47         switch (optchar) {
48         case 'p':
49               printf("imprimiendo matrices\n");
50               opt_print = 1;
51               break;
52         case 'd':
53               dimension = atoi(optarg);
54               printf("dimension %d\n",dimension);
55               break;
56         case 'f':
57               printf("matriz fija\n");
58               seed = 1;
59               break;
60         }//switch
61   } while (optchar != -1);
62
63   if (dimension ==0) {
64         printf("dimension 0, pos 0 , no funcionara\n");
65         return 0;
66   }
67   row = dimension;
68  col= dimension;
69
70
71 /* En MPI el proceso inicial (padre) tiene rank de 0,
72  esta es la sección donde el proceso padre
73 realiza su trabajo*/
74   if (localid==0){
75         srandom((unsigned int )(time (0)/2));
76         //Asignar memoria para tres matrices
77         //de row * col
```

```
78          matrix1= malloc (row * col * sizeof(int));
79          matrix2= malloc (row * col * sizeof(int));
80          matrix3= malloc (row * col * sizeof(int));
81          if (matrix1 == NULL || matrix2 == NULL ||
82              matrix3 == NULL){
83                MPI_Finalize();
84                if(matrix1!=NULL) free(matrix1);
85                if(matrix2!=NULL) free(matrix2);
86                if(matrix3!=NULL) free(matrix3);
87                exit(1);
88          }
89
90          /*llenamos las dos primeras matrices
91           aleatoriamente.. la definición de la función
92           randomatrix está en el archivo matrix.c */
93          randomatrix(matrix1, row, col, 5);
94          randomatrix(matrix2, row, col, 5);
95
96          /* Determinar el momento de inicio de ejecución,
97             aquí medimos cuando comenzamos a hacer
98             los cálculos*/
99          startwtime = MPI_Wtime();
100
101         //mostrar matrices si el usuario lo solicito
102
103         if (opt_print){
104               matrix_print(matrix1, row, col);
105               printf("\n");
106               matrix_print(matrix2, row, col);
107         }
108         //registro de cuáles renglones ya están completos
109
110         completerows =0;
111
112         /* el proceso con rank 0 transmite las dos matrices
113          a los demás, esto se hace por medio de un
114          broadcast de MPI a todos los miembros
115          del comunicador (MPI_COMM_WORLD)*/
116          rv=MPI_Bcast(matrix1, row * col, MPI_INT, 0,
```

```
117                       MPI_COMM_WORLD);
118
119       printf("Root, Broadcast said %d\n",rv);
120       rv= MPI_Bcast(matrix2, row * col, MPI_INT, 0,
121                       MPI_COMM_WORLD);
122
123       printf("Root, Broadcast said %d\n",rv);
124
125       //Asignar un row temporal para resultados
126       resultrow = malloc ((col + 1) * sizeof(int));
127       //esperar a tener todos los resultados completos
128
129       while (completerows < row){
130
131            /*Esta llamada espera a recibir un renglón
132            ya resuelto, de cualquier proceso hijo.
133            El parámetro MPI_ANY_SOURCE indica que
134            se pueden recibir valores de cualquier proceso*/
135
136            MPI_Recv(resultrow, col + 1, MPI_INT,
137                       MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&status);
138
139            printf("recibido renglon %d de %d \n",
140                    resultrow[0],status.MPI_SOURCE);
141
142            completerows++;
143
144  //este ciclo pone el renglón recibido en
145  //la matriz de resultado
146
147            for (i=0;i<col;i++ ) {
148                 matrix_set_cell(matrix3,row,col,i,
149                                  resultrow[0],resultrow[i + 1]);
150
151            }
152       }//while
153
154       //se termina el cálculo
155      // anotar tiempo al terminar
```

```
156         endwtime=MPI_Wtime();
157         // mostrar la matriz de resultados
158         // si fue solicitada
159         if (opt_print) {
160              matrix_print(matrix3, row, col);
161         }
162         //Y mostrar el tiempo total de cálculo
163         printf("wall clock time = %f \n",
164                endwtime - startwtime);
165         if(matrix1 != NULL) free (matrix1);
166         if(matrix2 != NULL) free (matrix2);
167         if(matrix3 != NULL) free (matrix3);
168
169    }//if
170    else {
171         /*Aquí comienza lo que ejecutan los procesos
172          con rank distinto de cero, es decir los procesos hijos.
173         Asignamos espacio para dos matrices*/
174         matrix1= malloc (row * col * sizeof(int));
175         matrix2= malloc (row * col * sizeof(int));
176         if (matrix1 == NULL || matrix2 == NULL){
177              MPI_Finalize();
178              if(matrix1!=NULL) free(matrix1);
179              if(matrix2!=NULL) free(matrix2);
180              if(matrix3!=NULL) free(matrix3);
181              exit(1);
182         }
183
184         /*En MPI, si mi rank no es 0, una llamada al broadcast
185         (notar el parámetro 4 que es de 0) indica
186         recibir el broadcast del proceso con ese rank*/
187
188         rv = MPI_Bcast(matrix1, row * col, MPI_INT,0,
189                        MPI_COMM_WORLD);
190
191         rv = MPI_Bcast(matrix2, row * col, MPI_INT,0,
192                        MPI_COMM_WORLD);
193
194         /*De acuerdo a las dimensiones de  las matrices
```

```
195         y al número de procesos, calcular número de
196         particiones, renglones por partición,
197         el primer renglón que tiene que resolver este proceso,
198         y el último renglón */
199
200        partitions = numprocs - 1;
201        rowstodo = (int) (row / partitions);
202        firstrow = rowstodo * (localid - 1);
203        lastrow = firstrow + rowstodo -1;
204
205        // el último proceso amplía su limite para
206        // tomar los huerfanitos
207        if (localid== numprocs -1) {
208             lastrow= lastrow + (row % partitions);
209        }
210
211
212        // asignar un row temporal
213        resultrow=malloc((col + 1) * sizeof(int));
214
215        // calcular cada row del grupo que le toca.
216        for (i= firstrow; i<= lastrow ;i++ )
217        {
218             y= i;
219             for (x=0; x<col ;x++ )
220             {
221             suma =0;
222             for (k=0;k<col ;k++ )
223             {
224                  elem1 =
225                         matrix_get_cell(matrix1,row,
226                                        col, k, y);
227                  elem2 =
228                         matrix_get_cell(matrix2,row,
229                                        col, x, k);
230
231                  suma += elem1 * elem2;
232             }
233
```

```
234             resultrow[x+1]= suma;
235       }
236
237       resultrow[0]=i;
238       MPI_Send( resultrow, col+1,
239                  MPI_INT, 0, 1, MPI_COMM_WORLD);
240
241  }
242
243       if(matrix1!=NULL) free(matrix1);
244       if(matrix2!=NULL) free(matrix2);
245       if(matrix3!=NULL) free(matrix3);
246
247  } //termina sección de calculo proceso hijo
248
249  //Terminamos la sesión de MPI
250  MPI_Finalize();
251  return 0;
252
253  }//main
```

Una corrida ejemplo de este programa, con una matriz de dimensión 150 (es decir, 150 x 150, o 22,500 elementos) es como sigue:

```
$ mpicc -o m1 matrix1.c
$ mpirun -np 5 ./m1 -d 150
soy el proceso 0 de 5 en euridice
dimension 150
Root, Broadcast said 0
Root, Broadcast said 0
recibido renglon 111  de 4
recibido renglon 112  de 4
recibido renglon 113  de 4
.
.
.
recibido renglon 108  de 3
recibido renglon 109  de 3
recibido renglon 110  de 3
```

```
wall clock time = 0.615142
soy el proceso 1 de 5 en hercules
dimension 150
soy el proceso 2 de 5 en euridice
dimension 150
soy el proceso 3 de 5 en hercules
dimension 150
soy el proceso 4 de 5 en euridice
```

Implementación con PVM

El programa de multiplicación de matrices paralelo utilizando PVM se muestra al final.

La inicialización de PVM es hasta cierto punto implícita; el llamar a cualquier función de PVM automáticamente enrola al proceso en la máquina virtual, si es que no se encontraba en ella anteriormente. Por convención la primera llamada debe ser a la función `pvm_mytid` (línea 43), con la que también se obtiene el `TID` (identificador de tarea o t*ask identifier*) del proceso. A continuación se obtiene el `TID` del proceso padre con una llamada a `pvm-parent`. La necesidad de conocer el `TID` del proceso padre será obvia más adelante.

Entre la línea 80 y 112, el proceso padre (identificado por el hecho de que su proceso padre tiene el valor `PvmNoParent`, que se verifica en la línea 80) debe iniciar o engendrar a los demás procesos que tomarán parte en el cálculo.

La función `pvm_spawn` inicia, con una sola llamada, todos los procesos necesarios (el número de procesos a iniciar se pasa como un parámetro a la función). La función permite especificar dónde se deben iniciar los procesos, o bien dejar que PVM decida dónde hacerlo; también permite pasar parámetros de línea de comandos a los procesos (segundo parámetro) a fin de que su inicialización pueda ser controlada adecuadamente. La función devuelve un arreglo con los `TID` de los procesos que se iniciaron.

Una vez concluida la inicialización de procesos, todos los procesos deben unirse a un grupo, que arbitrariamente se nombró `matrix-world` (línea 116), obteniendo al mismo tiempo su posición en el grupo. El grupo cumple una función similar a la del comunicador en MPI, es decir, permite organizar procesos en grupos de trabajo para restringir el alcance de algunas funciones de comunicación. Una vez unido al grupo, el proceso determina el tamaño del grupo (línea 119). El tamaño del grupo y la posición del proceso se utilizan para determinar el rango de renglones a resolver.

Los grupos no son una función intrínseca de PVM. La funcionalidad de grupos está provista por una biblioteca adicional, así como un demonio que arbitra la comunicación en grupos. La biblioteca y el demonio se incluyen con la distribución de PVM así que no se requiere

configuración adicional para usarlos, sin embargo téngase en mente que la funcionalidad es adicional a las funciones básicas de PVM.

A continuación se realiza la bifurcación de trabajo en el proceso, designando al proceso maestro. En PVM hay un concepto directo de proceso padre, y por convención el proceso padre será el maestro. Basándose en aquél proceso que no tiene padre, el proceso maestro realiza su trabajo de la línea 136 a la 267, y los procesos esclavo de la línea 268 a la 384.

El proceso padre genera las matrices aleatorias y las transmite a todos los procesos hijo por medio de un *broadcast*.

Las operaciones de comunicaciones en PVM involucran algunos pasos. El primero es inicializar el buffer de transmisión, con una llamada a `pvm_initsend` (línea 182). Posteriormente, se debe empacar el mensaje a enviar en dicho buffer. Esto representa un paso adicional, pero también permite empacar varios mensajes en el buffer y enviarlos todos simultáneamente. El empaque se realiza con una llamada a `pvm-pkint3` indicando la dirección y tamaño del dato a enviar. Finalmente, el envío se realiza con una llamada a la función correspondiente, en este caso se trata de un *broadcast* y la función es `pvm_bcast`, a la que le especificamos el grupo al que se envía y una etiqueta identificadora de mensaje. La etiqueta es en realidad un valor entero, aunque se acostumbra definir nombres significativos para las constantes de etiqueta (líneas 16-17).

Antes de transmitir la segunda matriz se debe llamar a `pvm_initsend` nuevamente, a fin de limpiar el buffer, ya que éste es acumulativo.
Finalmente el proceso padre espera la recepción de los resultados enviados por los procesos hijo. Utilizando una recepción de mensaje bloqueante (`pvm_recv`), el proceso espera en la línea 221 hasta recibir un mensaje de cualquier proceso, y con cualquier etiqueta de identificación (indicado por los valores -1 que especifican recepción tipo *wildcard*). Al recibir el mensaje (un renglón resuelto) se desempaca (`pvm_upkint`, línea 228), y se integra al resultado final.

Al terminar de recibir los renglones el proceso padre muestra el tiempo empleado en el cálculo y termina su ejecución.

A partir de la línea 170, los procesos esclavos (para los cuales la variable `pvmparent` es distinto de la constante `pvmNoParent`, indicando que sí tienen un proceso padre) asignan memoria para las matrices a operar y reciben sus valores por medio de *broadcast*. En PVM la recepción se hace con una llamada a `pvm_recv`, independientemente de si el origen fue un *broadcast* o un envío punto a punto. Nótese que en las llamadas a `pvm_recv` en las líneas 293 y 304 se especifica la etiqueta de mensaje `MATRIX_TAG`, que corresponde a la etiqueta que se empleó al enviar el *broadcast*.
Los procesos deben desempacar las matrices recibidas por medio de la función `pvm_upkint`, después de lo cual determinan los renglones que deben calcular (líneas 317 a 325) y comienzan a realizar los cálculos.
Al completar cada renglón, lo envían al proceso padre, inicializando su buffer de mensajes (`pvm_initsend`, línea 353), empacando el resultado (`pvm-pkint`, línea 355) y finalmente

enviándolo al proceso padre (`pvm_send`, línea 366). En la llamada a `pvm_send`, nótese el primer parámetro `myparent`, que indica enviar el mensaje al proceso padre.

Los nodos continúan con este proceso hasta terminar el cálculo de sus renglones asignados, en este momento terminan su ejecución.

MatrixPVM.c

```
1     /*Programa de multiplicación de matrices paralelizado
2     utilizando PVM. Daniel Manrique */
3
4     #include "matrix.c"
5     #include <stdio.h>
6     #include <math.h>
7     #include <stdlib.h>
8     #include <unistd.h>
9     #include <pvm3.h>
10    #include <sys/time.h>
11
12    /*Definir algunas banderas de mensaje
13    PVM las utiliza para distinguir entre
14    Distintos tipos de mensajes */
15
16    #define MATRIX_TAG 1
17    #define ROW_TAG 2
18
19    int main(int argc, char **argv)
20    {
21       int mytid, myparent;
22       int rcvbuf;
23       int localid, numprocs, info;
24       int *child;
25       int ntask;
26       int *resultrow;
27       int rv, row, col;
28       int *matrix1 = NULL;
29       int *matrix2 = NULL;
30       int *matrix3 = NULL;
31       int x, y , i ,k;
32       int partitions, firstrow, lastrow, rowstodo;
33       int elem1, elem2, suma;
34       int completerows;
35       int dimension;
36       char optchar;
```

```
37      int opt_print = 0;
38      int seed = (unsigned int) (time (0)/2);
39      struct timeval starttime, endtime;
40      double dstart, dend;
41
42      //obtener la task id del proceso padre y del hijo
43      mytid = pvm_mytid();
44      myparent = pvm_parent();
45      /* Este ciclo procesa las opciones de la línea
46       de comandos y fija las banderas necesarias
47       del programa. En particular nos interesa que
48       esté en este punto de programa por que necesitamos
49       1)  que el padre conozca cuantos hijos
50       va a tener (parámetro t)
51       2) que los hijos conozcan la dimensión de la matriz
52       (parámetro d)*/
53      do {
54           optchar = getopt(argc, argv, "fpd:t:");
55           switch (optchar){
56           case 'p' :
57                printf("imprimiendo matrices \n");
58                opt_print = 1;
59                break;
60           case  'd':
61                dimension= atoi (optarg);
62                printf("dimension %d \n", dimension);
63                break;
64           case 't':
65                ntask = atoi (optarg);
66                if (ntask < 1)
67                    ntask = 1;
68                printf("% d proceso \n", ntask);
69                break;
70           case 'f':
71                printf("matriz fija");
72                seed =1;
73                break;
74           }
75      } while (optchar != -1);
76
77
78
79      // Si es el proceso padre, crea a los hijos
```

```
80      if (myparent == PvmNoParent) {
81           //Asignar el arreglo que tendrá
82           // los tid de los procesos hijo
83           child = (int *) malloc (sizeof (int)*ntask);
84
85           /* La llamada pv_Spawn cre a los hijos, hace
86           esto solicitando a la máquina virtual que cree más
87           procesos. La máquina virtual decide dónde iniciarlos,
88           normalmente hace un round-robin entre
89           los nodos, a menos que en esta llamada se le
90           especifique dónde iniciarlos. Importante: como
91           las invocación de los hijos la hace esta llamada,
92           hay que pasar la lista de parámetros
93           (desde argv[1]) a los hijos, para que la procesen bien,
94           en particular nos interesa que tomen el parámetro –d
95           para que puedan dimensionar su matriz
96           del mismo tamaño que del padre. */
97
98           info= pvm_spawn(argv[0],&argv[1],
99                           PvmTaskDefault,(char *)NULL,
100                          ntask,child);
101
102          // imprimir los tid de cada hijo
103          for (i=0; i< ntask ; i++ )
104               // imprime el error code en decimal
105               if (child [i]< 0)
106                    printf("%d",child[i]);
107               else
108               // imprime el tid en hexadecimal
109               printf("t%x\t",child[i]);
110               putchar('\n');
111     }
112     /* Todos los procesos deben unirse al grupo
113     matrix_World. La llamada devuelve el número
114     de instancia dentro del grupo; se considera
115     equivalente al comm_rank de MPI */
116     localid=pvm_joingroup ("matrix_world");
117
118     // obtener el núemro de procesos en el grupo
119     numprocs = pvm_gsize("matrix_world");
120
121     printf("soy el proceso %d  de %d \n",
122           localid, numprocs);
```

```
123     printf("mu tid is %d, my parent %d \n",
124           mytid,myparent);
125
126     if (dimension == 0) {
127          printf("dimension 0, pos no funcionará \n");
128          return 0;
129     }
130     row= dimension;
131     col = dimension;
132
133     /* Esta es la sección donde el proceso
134     padre realiza su trabajo */
135
136     if (myparent == PvmNoParent) {
137          printf("padre\n");
138          srandom((unsigned int) (time (0) / 2));
139          // Asignar memoria para las tres matrices
140          // de row * col
141          matrix1 = malloc (row * col *sizeof(int));
142          matrix2 = malloc (row * col *sizeof(int));
143          matrix3 = malloc (row * col *sizeof(int));
144          if (matrix1 == NULL || matrix2 == NULL ||
145               matrix3 == NULL){
146               pvm_exit();
147               if(matrix1!=NULL) free(matrix1);
148               if(matrix2!=NULL) free(matrix2);
149               if(matrix3!=NULL) free(matrix3);
150
151               exit(1);
152          }
153
154          /* llenamos las dos primeras matrices
155          aleatoriamente.. la definición de la función
156          randomatrix esta en el archivo
157          matrix.c */
158          randomatrix(matrix1,row,col,5);
159          randomatrix(matrix2,row,col,5);
160
161          /* Determinar el momento de inicio de ejecución,
162          aquí medimos cuándo comenzamos realmente
163          a hacer los cálculos */
164          rv= gettimeofday(&starttime,NULL);
165
```

```
166          // imprimir matrices
167          if (opt_print) {
168               matrix_print(matrix1,row,col);
169               printf("\n");
170               matrix_print(matrix2,row,col);
171          }
172          // registro de cuáles renglón ya están completos
173
174          completerows=0;
175          /* el proceso padre transmite las dos matrices
176          a los demás, esto se hace por medio de un
177          broadcast a todos los miembros del grupo
178          (matriz_world) */
179
180          /* limpiar e inicializar el buffer
181          de transmisión de PVM */
182          pvm_initsend(PvmDataDefault);
183
184          /* Importante: en PVM debemos de empacar
185          los datos en un buffer de mensaje antes
186          de enviarlos.  La llamanda a pkint empaca
187          el dato al que apuntaremos en el buffer
188          de mensajes actual. */
189
190          rv = pvm_pkint(matrix1,row * col, 1);
191          if (rv < 0) {
192               pvm_perror("pkint matrix1");
193          }
194          /* pvm_bcast transmite el buffer actual (matriz)
195          a todos los miembros del grupo matrix_world.
196          Le pone la bandera MATRIX_TAG, una bandera
197          definida por el usuario para identificarlo */
198
199          rv= pvm_bcast("matrix_world",MATRIX_TAG);
200          printf("Root,Broadcast said %d \n",rv);
201
202          // segunda matriz, notese que
203          // se limpió el buffer antes
204          pvm_initsend(PvmDataDefault);
205          pvm_pkint(matrix2,row * col,1);
206
207          rv= pvm_bcast("matrix_world",MATRIX_TAG);
208          printf("Root , Broadcast said %d \n",rv);
```

```
209          // asignar un renglón temporal para resultados
210          resultrow= malloc((col +1) * sizeof(int));
211          // esperar a obtener todos los
212          // resultados completos
213          while (completerows < row) {
214          /* Esta llamada espera a recibir un renglón
215          ya resuelto, de cualquier proceso hijo.
216          Nótese los valores –1 que PVM identifica como
217          “wildcards” y nos idican aceptar valores con
218          cualquier bandera, y de cualquier proceso.
219          rcvbuf nos indica el buffer donde se recibió
220          el mensaje */
221          rcvbuf = pvm_recv(-1,-1);
222          if (rcvbuf < 0) {
223               pvm_perror("rcvbuf row");
224          }
225          /* Bajo PVM se necesita desempacar el mensaje,
226          sacarlo del buffer y ponerlo en una
227          variable utilizable */
228          rv= pvm_upkint(resultrow,col + 1,1);
229          if (rv < 0) {
230               pvm_perror("upk row");
231          }
232          printf("recibido renglon %d \n",resultrow[0]);
233
234          completerows++;
235          // Este ciclo pone el renglón recibido en
236          // la matriz de resultado
237          for (i=0;i< col ; i++ ) {
238               matrix_set_cell(matrix3,row,col,
239                               i,resultrow[0],
240                               resultrow[i+1]);
241
242          }
243
244     }
245
246     /* se termina el cálculo.
247     Se anota el tiempo al terminar  */
248     rv = gettimeofday(&endtime,NULL);
249     // imprime la matriz de resultados
250     // si es que fue solicitada
251     if (opt_print) {
```

```
252          matrix_print(matrix3,row,col);
253     }
254     // mostrar el tiempo total del cálculo
255     dend = (double) endtime.tv_sec;
256     dend += (double) endtime.tv_usec * 0.000001;
257
258     dstart = (double) starttime.tv_sec;
259     dend += (double) starttime.tv_usec * 0.000001;
260
261     printf("wall clock time = %f \n", dend-dstart);
262
263    //    if(matrix1 != NULL) free (matrix1);
264    //    if(matrix2 != NULL) free (matrix2);
265    //    if(matrix3 != NULL) free (matrix3);
266
267     } //Aquí termina la ejecución del proceso padre
268     else {
269          printf("Hijo \n");
270          fflush(stdout);
271
272     /* Aquí comienza lo que ejecutan los porcesos hijo */
273
274          // Asigno espacio para dos matrices
275          matrix1= malloc(row * col * sizeof(int));
276          matrix2= malloc(row * col * sizeof(int));
277
278          if (matrix1 == NULL || matrix2 == NULL) {
279               pvm_exit();
280               if(matrix1 != NULL) free (matrix1);
281               if(matrix2 != NULL) free (matrix2);
282               if(matrix3 != NULL) free (matrix3);
283               exit(1);
284          }
285
286
287          /* Se espera recibir un mensaje del proceso padre
288          (myparent), con bandera MATRIX_TAG,
289          entonces se sabrá que es una de las dos matrices.
290          Nótese que en PVM es distinto si se está
291          recibiendo un broadcast o un ,mensaje directo */
292
293          rcvbuf= pvm_recv(-1,MATRIX_TAG);
294          if (rcvbuf < 0){
```

```
295               pvm_perror("rcvbuf matrix1");
296          }
297          // Desempacar directo en el espacio de la matriz
298
299          rv = pvm_upkint(matrix1, row * col, 1);
300          if (rv < 0) {
301               pvm_perror("upv amtrix1");
302          }
303          // repetimos para la siguiente matriz..
304          rcvbuf= pvm_recv(-1,MATRIX_TAG);
305          if (rcvbuf < 0) {
306               pvm_perror("rcvbuf matrix2");
307          }
308          rv = pvm_upkint(matrix2,row *col,1 );
309          if (rv < 0) {
310               pvm_perror("recv matrix2");
311          }
312          /* De acuerdo a las dimensiones de las
313          matrices y al número de procesos, calcular
314          número de particiones, renglones por partición,
315          el primer renglón que tiene que resolver este
316          proceso, y el último renglón */
317          partitions= numprocs -1;
318          rowstodo = (int) (row / partitions);
319          firstrow = rowstodo * (localid -1);
320          lastrow = firstrow + rowstodo -1;
321          // el último proceso amplia su límite
322          //para tomar los huerfanitos
323          if (localid == numprocs -1 ) {
324               lastrow= lastrow + (row % partitions);
325          }
326
327          //asignar un renglón temporal
328          resultrow = malloc((col +1 ) * sizeof(int));
329
330          // calcular cada renglón del grupo
331          for (i= firstrow; i<= lastrow ; i++ ) {
332          /* El elemento que se está calculando
333          se encuentra en x,y; esntonces lo que
334          varía más rápido es x porque y es el renglón */
335               y=i;
336               for (x=0; x<col ;x++ ) {
337                    suma =0;
```

```
338                     for (k=0;k<col ;k++ ) {
339                          elem1 =
340                                 matrix_get_cell(matrix1,row,
341                                                      col, k, y);
342                          elem2 =
343                                 matrix_get_cell(matrix2,row,
344                                                      col, x, k);
345                          suma += elem1 * elem2;
346                     }
347                     resultrow[x+1]= suma;
348               }
349               /* Ahora que ya se tiene el renglón calculado
350               hay que mandarlo al proceso padre*/
351               resultrow[0]=i;
352               // limpiar el buffer
353               pvm_initsend(PvmDataDefault);
354               // empacar
355               pvm_pkint(resultrow,col + 1,1);
356
357               /* Enviar el paquete. Este es un envío
358               directo a un proceso, por medio de un sed.
359               Mandamos el buffer de mensajes actual
360               directamente al proceso padre con una
361               bandera de ROW_TAG para significar lo
362               que estamos enviando (renglón),
363               aunque en este caso el padre ignora
364               el tipo de bandera */
365
366               pvm_send(myparent,ROW_TAG);
367
368          } // termina proceso de un renglón
369     } // termina el cálculo del proceso hijo
370
371     pvm_exit();
372     if(matrix1 != NULL) free (matrix1);
373     if(matrix2 != NULL) free (matrix2);
374     if(matrix3 != NULL) free (matrix3);
375     return 0;
376  }//main
```

Una corrida ejemplo de este programa, con una matriz de dimensión 150 (es decir, 150 x 150, o 22,500 elementos) es como sigue:

```
$     cc     /usr/share/pvm3/examples/$fuente     -o     $objeto     -I
/usr/share/pvm3/include -L /usr/share/pvm3/lib/LINUX -lgpvm3 -lpvm3 -lm
$ ./m2 -t 5 -d 150
 5 proceso
dimension 150
t40011  t40012  t40013  t8000b  t8000c
soy el proceso 5  de 6
mu tid is 262160, my parent -23
padre
Root,Broadcast said 0
Root , Broadcast said 0
recibido renglon 0
recibido renglon 1
recibido renglon 2
recibido renglon 3
recibido renglon 4
recibido renglon 5
.
.
.
recibido renglon 107
recibido renglon 108
recibido renglon 624
recibido renglon 625
wall clock time = 22.705273
```

Comparación de funciones: PVM vs MPI

Se puede apreciar que MPI es ligeramente más compacto, requiriendo menos código que PVM. Obteniendo las líneas de código fuente efectivas para cada archivo, se determinó que la implementación en MPI requiere 141 líneas de código, mientras que la implementación en PVM utiliza 192 líneas. Apoyando esta observación, se puede indicar que en general PVM es un poco más laborioso de utilizar que MPI, requiriendo en ocasiones más pasos para lograr el mismo resultado. Por ejemplo, la realización de un *broadcast* en MPI requiere únicamente una llamada a función, mientras que en PVM se requieren dos (una para empacar la información y otra para realizar el envío).
En general, se considera que MPI proporciona una API más limpia y mejor planeada. Se requieren un menor número de llamadas a funciones, dichas funciones están mejor organizadas, y es obvio el hecho de que al momento de planear la API se tuvieron en cuenta la mayoría de las posibles necesidades de comunicación por paso de mensajes.

PVM es un proyecto con más antigüedad, y esto es obvio en algunas de sus funciones (en particular las funciones de manejo de grupo), dando la impresión de que dichas funciones se agregaron sobre la marcha, sin dar mucha importancia a la planeación y enfatizando el lograr una implementación utilizable de la biblioteca. Por otro lado, la API de PVM es un tanto engorrosa,

en ocasiones requiriendo un número de llamadas mayor para lograr funciones relativamente sencillas, y algunos comportamientos no están bien documentados.

A continuación se muestra una tabla de tiempos que se obtuvieron al hacer diferentes pruebas con diferentes matrices de 20 x 20, 150 x 150 y 650 x 650, utilizando los programas *uniprocesador.c*, *matrixMPI.c* y *matrixPVM.c*.

	Tamaño de la matriz		
Núm. Procesos	20x20	150x150	650x650
1 (Uniprocesador)	0.418219	3.795671	42.503326
5 (MPI)	0.096106	0.615142	22.057449
5 (PVM)	1.905265	1.814607	22.705273

Figura 4.2 Tabla de tiempos de ejecución

Comparando estos resultados, se puede observar que los tiempos del programa escrito con PVM son ligeramente mayores, indicando un menor desempeño, que posiblemente se pueda atribuir a una implementación menos eficiente de los mecanismos de comunicación de red de PVM; esta observación se basa en el comportamiento de la red, que al ejecutar el programa en PVM presenta un gran número de colisiones y aparente saturación, lo cual es visible en el comportamiento de los indicadores en los concentradores. Aún cuando la diferencia es menor, se debe tener en cuenta su existencia al momento de seleccionar la biblioteca de paso de mensajes a utilizar. Estos resultados refuerzan la recomendación de utilizar MPI en la medida posible. Aún así, para aplicaciones grandes, PVM muestra un mejor desempeño que un solo procesador.

4.1.2 *High Performance* LINPACK: un *benchmark* ampliamente reconocido

El *benchmark* de LINPACK es una prueba de rendimiento, o *benchmark*, de uso muy difundido en la comunidad de cómputo de alto rendimiento. Este *benchmark* no pretende medir el desempeño general de un sistema. En vez de ello, evalúa el desempeño en un área muy específica: el cálculo de sistemas de ecuaciones lineales de alta densidad. Esta medida resulta útil para conocer las capacidades de equipo de cómputo de alto rendimiento, pues esta clase de equipos generalmente se utilizan para resolver ese tipo de problemas, de forma que el *benchmark* de LINPACK proporciona una idea bastante acertada del desempeño que el equipo tendrá en aplicaciones reales.

El uso del *benchmark* de LINPACK fue introducido en 1979 por Jack Dongarra, investigador de la Universidad de *Tennessee* en *Knoxville*, como parte del paquete LINPACK, un juego de bibliotecas matemáticas en FORTRAN para solución de sistemas de ecuaciones lineales. En general, el *benchmark* realiza la resolución de un sistema de ecuaciones generado aleatoriamente, expresado como una matriz de coeficientes que en la computadora están representados con números de punto flotante, normalmente a una precisión de 64 bits. El paso crucial de esta solución es la descomposición LU con pivoteo parcial de la matriz de coeficientes. Obtener la

solución requiere 2/3n3 + 2n2 operaciones de punto flotante, donde n es la dimensión de la matriz. Normalmente se emplean valores de *n=100* y *n=1000*.

Finalmente el *benchmark* entrega un valor de rendimiento expresado en MFlops (millones de operaciones de punto flotante por segundo). Este valor es el que se suele emplear al hacer comparaciones entre diversos equipos.

LINPACK fue originalmente implementado en lenguaje FORTRAN para máquinas uniprocesador, vectoriales y SMP. A medida que los equipos MPP empezaron a ser reconocidos en el ámbito de desarrollo científico, se hizo necesario el contar con una manera de comparar su desempeño, de forma que se desarrolló el benchmark HPL (*High Performance* LINPACK). HPL es una implementación del benchmark LINPACK escrita en lenguaje C, que puede ejecutarse en cualquier equipo que cuente con una implementación de MPI, lo cual incluye a prácticamente cualquier equipo MPP comercial y, desde luego, *clusters* tipo Beowulf.

HPL es el *benchmark* utilizado para medir el desempeño de las computadoras más rápidas del mundo, gracias a su portabilidad, su dependencia en otras bibliotecas que están ampliamente disponibles, particularmente MPI y BLAS7, y la capacidad de alterar fácilmente los parámetros de cálculo a fin de determinar la configuración óptima para obtener el mejor rendimiento.

Es interesante el realizar pruebas con HPL en un Beowulf, básicamente para comparar el incremento de rendimiento al utilizar toda la capacidad del mismo. Desde luego, no se espera que el rendimiento alcance los niveles presentados por las computadoras más rápidas del mundo; (la computadora más rápida alcanza un rendimiento sostenido de 7226 GFlops). Sin embargo el comparar ambos equipos con el mismo mecanismo de medición puede resultar interesante.

Requisitos previos

HPL tiene como requisitos previos la presencia en el sistema de alguna implementación de MPI, con la que ya se cuenta; y de la biblioteca BLAS para cálculos algebraicos.

Instalación de biblioteca ATLAS

Primeramente se requiere instalar la biblioteca BLAS o alguna que proporcione funcionalidad similar. BLAS (*Basic Linear Algebra System*) proporciona funciones de álgebra lineal en FORTRAN, definiendo para ello una API que otras bibliotecas más modernas y eficientes también han implementado. En este caso se seleccionó la biblioteca ATLAS (*Automatically Tuned Linear Algebra Software*), desarrollada por la Universidad de *Tennessee* en *Knoxville*. ATLAS está escrito en C, y es un software que al momento de compilar, determina automáticamente las características de la arquitectura del equipo y genera bibliotecas utilizando los algoritmos más eficientes para las funciones que proporciona. Su instalación es sencilla pero, debido al proceso de pruebas para determinar los parámetros que resultan en un mejor rendimiento, suele tomar bastante tiempo.

La biblioteca ATLAS se obtiene de *http://www.netlib.org/atlas/atlas3.2.0.tgz*. Una vez contando con este archivo se descompacta y se pasa al subdirectorio de ATLAS con los siguientes comandos:

```
$ tar -zxvf atlas3.2.0.tgz
$ cd ATLAS
```

Posteriormente se ejecuta el siguiente comando para generar la configuración de ATLAS:

```
$ make config
```

En este paso se deben responder algunas preguntas sobre el sistema con que se cuenta. En todas las preguntas se puede dar la respuesta predeterminada, o presionar ENTER, salvo en la siguiente opción, donde se pide el tipo de máquina. Aquí se debe seleccionar el procesador correcto:

```
Enter your machine type:

1. Other/UNKNOWN
2. AMO Athlon
3. Pentium PRO
4. Pentium II
5. Pentium III
6. Pentium
7. Pentium MMX
8. IA-64 Itanium
Enter machine number [1]:
```

Una vez concluida la configuración se puede proceder a la compilación e instalación con el siguiente comando:

```
$ make install arch=Linux_PS
```

Nótese que el valor para el parámetro `arch` se compone del nombre de la arquitectura del sistema operativo (en este caso Linux) y del procesador con que se cuenta (en este caso PS o Pentium).
Esto inicia la autoconfiguración, compilación e instalación de ATLAS. En el servidor central del *cluster*, este proceso tomó alrededor de una hora.

HPL *(High Performance LinkPack)*

El paquete de HPL se obtiene de *http://www.netlib.org/benchmark/hpl/hpl.tgz*. Una vez contando con este archivo se descompacta y se pasa al subdirectorio HPL con los siguientes comandos:

```
$ tar -zxvf hpl.tar.gz
```

HPL proporciona varios *Makefiles* con la configuración adecuada para distintos sistemas bajo el directorio setup. Se copia el más adecuado al directorio HPL:

```
$ cp setup/Make.Linux_PII_CBLAS_gm ./
```

Posteriormente se modifica el *Makefile* para corresponder con la configuración de nuestro sistema. Únicamente se debe modificar el valor de las siguientes variables, para indicar que se va a emplear la biblioteca ATLAS:

```
LAdir = /usr/lib
LAinc =
LAlib = $ (LAdir)/libcblas.a $ (LAdir)/libatlas.a
```

Una vez concluida la configuración se puede proceder a la compilación con el siguiente comando:

```
$ make arch=Linux_PII_CBNAS_gm
```

Esto genera los archivos ejecutables en *binILinux_PICCBLAS_gm*. Los archivos necesarios para correr el *benchmark* son dos, *xhpl* y el archivo de configuración *HPL.dat*.

Configuración

La configuración de parámetros para HPL, como el tamaño del problema a resolver, las dimensiones de la matriz de procesos, y otros, se definen en el archivo *HPL.dat*. El archivo empleado para la realización de las pruebas se muestra al final del capítulo.
Básicamente, el archivo permite especificar distintos juegos de parámetros a experimentar; por ejemplo, permite definir varios tamaños de problema a probar, así como número y organización de los procesos que participarán en el cálculo. Se prueban todas las combinaciones posibles. El archivo mostrado especifica probar problemas de dimensión 650 y 1000, con un proceso, 16 procesos (organizados en arreglos de 4 x 4 y 8 x 2) y 17 procesos (en un arreglo de 1 x 17).

Ejecución

Una vez creado el archivo, se procede a correr la prueba:

```
$ mpirun -np 17 ./xhpl
```

El número de procesos debe ser suficiente para incluir al número dado por las matrices de proceso. La prueba generará los resultados para los problemas especificados, enviando bloques de información como sigue:
Los datos de interés son *N, NB, P* y *Q* que indican las características del problema a resolver; el tiempo de ejecución, y el desempeño en GFlops.

```
T/V                N     NB     P     Q       Time                 Gflops
--------------------------------------------------------------------------------
W10L2L2           1000    16    17     1     164.65              4.058E-03
--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV-
Max aggregated wall time rfact .  .  . :                          77.71
+ Max aggregated wall time pfact  .  . :                          77.61
+ Max aggregated wall time mxswp  .  . :                          77.49
Max aggregated wall time update   .  . :                          105.31
+ Max aggregated wall time laswp  .  . :                          86.77
Max aggregated wall time up tr sv .  . :                          0.67
--------------------------------------------------------------------------------
```

Figura 4.3 Ejecución del *Benchmark*

Tras realizar las pruebas con los valores especificados, se obtuvieron las siguientes tablas:

Dimensión	Nodos	P	Q	Tiempo (s)	GFLOPS
650	1	1	1	4.79	3.698e-02
650	16	2	8	19.56	9.392e-03
650	16	4	4	22.69	8.098e-03
650	17	1	17	31.95	5.750e-03
1000	1	1	1	127.73	5.231e-03
1000	16	2	8	45.04	1.484e-02
1000	16	4	4	49.47	1.351e-02
1000	17	1	17	73.20	9.128e-03

Figura 4.4 Tabla de Resultados del *Benchmark*

El archivo de configuración HPL permite especificar distintos parámetros para el cálculo, el más importante de los cuales es el arreglo de procesos. El manual de HPL [20] indica que, para arreglos de comunicaciones punto a punto, lo cual incluye redes que emplean *switches* para segmentar el tráfico, el mejor tipo de arreglo es lo más cuadrado posible (como un ejemplo, distribuir 16 nodos en un arreglo de 4 x 4), mientras que en un esquema de comunicaciones por bus, como el que se utiliza en este caso, se prefiere un arreglo plano (un arreglo de 8 x 2 se considera más "plano"). Por ello es que se especificaron estas dos combinaciones al utilizar 16 nodos. En el caso de 17 nodos, se tiene únicamente una posibilidad de organización, el arreglo de 1 x 17.

Para cada tamaño de problema se tienen, entonces, 4 juegos de resultados, indicando tiempo de resolución y desempeño en GFlops.

Para la matriz de coeficientes de 650 x 650, se aprecia que el mejor tiempo y mejor desempeño se obtienen con un solo procesador. Esto sugiere, nuevamente, que la complejidad de la realización del cálculo a este tamaño de problema niega la ventaja de utilizar más nodos. El mejor desempeño para este tamaño de problema es de 36.98 MFlops. Nótese que, entre las soluciones que emplearon 16 y 17 nodos, la más rápida fue la que utiliza una matriz de procesos de 8 x 2, siendo incluso más rápida que emplear 17 nodos. Esto permite ver que la correcta organización de la matriz de procesos es importante para tener un mejor desempeño, y da la idea de que, para esta configuración particular, la matriz de 8 x 2 es la que dará mejor rendimiento.

Con una matriz de coeficientes de 1000 x 1000, la situación cambia radicalmente. En este caso la solución con un procesador es la más lenta y la que reporta menor rendimiento, y el mejor resultado se obtiene con 16 nodos en matriz de 8 x 2, alcanzando para este problema un rendimiento de 14.84 MFlops. Este rendimiento es casi 3 veces superior al alcanzado con un solo procesador, que es de 5.23 MFlops.

Cabe mencionar nuevamente que utilizando 17 nodos, debido a la organización menos eficiente de la matriz de procesos, el rendimiento fue menor, únicamente de 9.12 MFlops.

En este momento resulta de interés comentar que, comparando el desempeño de la computadora más rápida del mundo, a 7226 GFlops, contra el mejor resultado obtenido por el *cluster*, de 14.84 MFlops, resulta que el *cluster* es 486927 veces más lento que la computadora *ASCI White*. Esta comparación se realizó utilizando el tamaño de matriz de 1000 x 1000, que es el utilizado en *http://www.top500.org* para comparar el desempeño de las supercomputadoras más rápidas.

Si bien la diferencia de desempeño es casi cómica, es también de interés el hecho de que efectivamente la solución se obtiene más rápidamente usando todo el *cluster* que utilizando sólo un nodo. Esto sugiere, nuevamente, que el uso del *cluster* proporciona un rendimiento mejor para problemas relativamente grandes.

4.2 Aplicaciones de manipulación de datos

Técnicas como la generación (*rendering*) de imágenes fotorrealistas por computadora también son buenos candidatos para su aceleración por medio de un *cluster*. En general esta clase de problemas pueden particionarse de manera que no exista necesidad de comunicación entre los elementos de procesamiento, lo cual redunda en un buen rendimiento en sistemas paralelos. El uso de *render farms* (granjas de trazado), en las cuales se emplean grandes cantidades de computadoras para trazar cuadros de animación generada por computadora en tiempos cortos, está tomando auge a medida que la industria cinematográfica tiende al uso de las computadoras para generación de efectos especiales e incluso películas completas.

Procesos más afines al uso de la computadora propiamente dichos también se pueden beneficiar del uso de *clusters*. La compilación de programas grandes y complejos puede acelerarse por medio del uso de compilación en paralelo, para lo cual existen utilerías como `pmake`, que se encargan de distribuir el trabajo entre los nodos.

En general, el contar con un equipo más rápido, como un *cluster*, abre las puertas hacia nuevas aplicaciones de computación. Sin embargo, se debe tener en cuenta las características y limitaciones que estos sistemas podrían llegar a tener y así se pueda ejercer un juicio cuidadoso al momento de elegir un *cluster* como plataforma para correr una aplicación determinada.

4.2.1 Ray tracing: una aplicación real

Una posible aplicación de un *cluster*, es el trazado de imágenes por computadora. Una de las técnicas más utilizadas para generar imagen por computadora de alta calidad es el *ray tracing* (trazado o seguimiento de rayos). El *ray tracing* es seleccionado como la aplicación real para la cual se puede utilizar el *cluster*, ya que cumple con una serie de características que la hacen interesante, útil y atractiva.

La técnica de ray tracing

El algoritmo de *ray tracing* fue propuesto por primera vez en 1968, por Appel9, aunque inicialmente su uso era únicamente en la detección de superficies ocultas. El *ray tracing* es actualmente la simulación más completa de un modelo de iluminación y reflexión por computadora. El algoritmo básicamente supone que un observador ve un punto en una superficie como resultado de la interacción de dicha superficie en ese punto con rayos que emanan de otros puntos de la escena.

El *ray tracing* va más allá que otras técnicas, pues en vez de considerar únicamente la interacción de los puntos de la superficie con la iluminación directa de las fuentes de luz, se toma en cuenta que en general, un rayo de luz puede alcanzar la superficie indirectamente por reflexión en otras superficies, transmisión a través de objetos parcialmente transparentes, o una combinación de ambos. Esto se denomina iluminación global, pues la luz se origina del ambiente de la escena, en vez de hacerlo por interacción local de la superficie con la iluminación directa de las fuentes de luz.
Este método tiene algunas desventajas, la más importante de las cuales es su gran requerimiento de procesamiento, tomando tiempos muy largos (horas o días) para calcular imágenes. Sin embargo, su ventaja más significativa a nivel algoritmo de graficación es que representa una solución parcial al problema de la iluminación global, combinando en un solo modelo la remoción de superficies ocultas, el sombreado debido a iluminación tanto directa como global, y el cálculo de sombras. Una ventaja adicional es que el algoritmo es muy fácilmente paralelizable.
El algoritmo de *ray tracing* trabaja completamente en el espacio de objetos. En un punto del plano de imagen, se obtienen las superficies visibles, así como el color e intensidad en el punto, trazando un rayo hacia atrás, desde el ojo o cámara, a través del punto de interés, y hacia la escena.

Si el rayo intersecta un objeto, entonces se realizan cálculos locales para determinar el color que resulte de la iluminación directa en el punto. Si es parcialmente reflejante, parcialmente transparente, o ambos, el color del punto en el plano de la imagen incluirá una contribución de los rayos reflejados y transmitidos. Éstos deben trazarse hacia atrás a partir del punto de intersección

para descubrir sus contribuciones. Determinar el color de estos rayos puede a su vez requerir el trazado de más rayos en intersecciones con objetos. Para determinar completamente el color del punto original en el plano visible, este juego de rayos debe trazarse hacia atrás dentro de la escena. El trazado de un rayo en particular termina cuando no se intersectan más objetos (en cuyo caso se asigna al rayo un color de fondo), o cuando el rayo está separado del observador por un número tal de intersecciones que su contribución de color se considera despreciable.

Se lanza un rayo desde el observador hacia cada punto a definir dentro del plano de imagen, pasando por el centro de dicho punto o *pixel*. Esto significa que la escena será muestreada en el espacio de objeto por rayos infinitamente delgados. En el proceso se tiene coherencia espacial de cero, es decir, todos los rayos se trazan independientemente sin utilizar información de rayos vecinos). Esto produce *aliasing*, lo cual representa una desventaja; por otro lado, hace que la implementación paralela sea trivial, y el proceso fácilmente distribuidle.

Existen una gran cantidad de programas y paquetes, tanto comerciales como gratuitos, que emplean el *ray tracing*, así como variantes y combinaciones con otras técnicas como la radiosidad y el trazado por líneas de rastreo.

Si bien, como ya se describió, el algoritmo de *ray tracing* es relativamente sencillo a nivel básico, en realidad éste puede complicarse, según se incrementa la cantidad de comportamientos visuales que el programa debe manejar, sus capacidades de manipulación de objetos, y la eficiencia de la implementación.

4.2.2 POV-Ray

POV-Ray (*Persistence of Vision Raytracer*) es un programa para *ray tracing* que ha estado en desarrollo durante varios años. Como tal, es una herramienta confiable, eficiente y robusta, cuyo código fuente está disponible, si bien no bajo una licencia libre. A pesar de ello, el uso de este código fuente no representa problemas ya que la versión modificada para uso en máquinas paralelas no se va a distribuir bajo términos que contravengan la licencia de POV-Ray. El punto central de distribución e información sobre POV-Ray es el sitio en Internet *http://www.povray.org.* POV-Ray cuenta con un lenguaje de descripción de escenas fácil de utilizar. La geometría y características de materiales, texturas, iluminación y cámaras de la escena a trazar se describen por medio de este lenguaje, posteriormente POVRay se encarga de interpretar la descripción y generar la imagen a partir de dicha información. Gracias a su popularidad y la facilidad de uso del lenguaje de descripción, existen grandes cantidades de escenas disponibles públicamente, así como descripciones de objetos que pueden utilizarse para componer nuevas escenas.

POV-Ray proporciona una serie de primitivas básicas como esferas, cajas, cilindros, cuerpos cuádráticos, conos, triángulos y planos; figuras más complejas como toroides, curvas cuárticas, texto, texturas fractales, prismas, polígonos, superficies de revolución y algunas otras. Adicionalmente éstas se pueden combinar por medio de geometría sólida constructiva (Constructive Solid Geometry, o CSG) para formar nuevas figuras. A estas figuras pueden asignarse patrones y propiedades de materiales que confieren a éstos texturas. A fin de poder

iluminar y visualizar estas escenas, el programa proporciona varios tipos de cámaras, entre ellos una cámara panorámica, una cámara con perspectiva, una "ojo de pescado", ortográfica y otras; fuentes luminosas cilíndricas, cónicas o de reflector y de área; se puede emplear iluminación ínter difusa para obtener efectos más reales en áreas cerradas o de interiores, efectos atmosféricos como niebla, neblina y arco iris, modelos de partículas para efectos como nubes, polvo o fuego, y sombreado y reflejos Phong y especulares.

POV-Ray puede entregar el trazado de la imagen con una profundidad de color hasta de 48 bits, en formatos TGA, PNGI2 Y PPMI3, entre otros.

Como se puede apreciar, la funcionalidad que puede tener un programa de *ray tracing* es extensa, y su implementación constituye un problema no trivial. Adicionalmente, ya que POV-Ray proporciona algoritmos de alta calidad y eficiencia para realizar estas tareas, se considera que su utilización permite ahorrar tiempo y obtener resultados de buena calidad.

Utilizando POV-Ray

POV-Ray es un programa para línea de comandos. Como mínimo, se debe indicar el nombre del archivo que contiene la descripción de la escena. Dicho archivo consta simplemente de texto en formato ASCII, en el lenguaje de descripción de escenas de POV-Ray y que normalmente, por convención, tiene la extensión .pov. El programa traza la escena y genera la salida como una imagen, tradicionalmente en formato TGA, si bien el formato PNG también es utilizado.
Así pues, una invocación típica sería:

```
$ x-povray -i skyvase.pov
```

Esto utiliza los parámetros por omisión, generando un archivo de salida con el mismo nombre base que el de entrada, en formato PNG (por lo tanto la salida quedará en `skyvase.png`). La resolución, si no se especifica, será de 320 x 240 pixeles.
Adicionalmente, en este caso se cuenta con MPI-POVRay, cuya invocación, desde luego, se debe realizar a través de `mpirun`:

```
$ mpirun -np 17 mpi-x-povray -i skyvase.pov
```

Esta invocación genera la imagen con los mismos parámetros y opciones que la anterior, la diferencia es que se utilizarán 17 nodos para procesamiento paralelo.

Una vez contando con el software para *ray tracing* en paralelo, inmediatamente se piensa en probar su ejecución y desempeño trazando una escena real. Para este propósito, una excelente opción es realizar el *benchmark* oficial de POV-Ray, lo que permitirá evaluar el funcionamiento con una escena de uso común, así como obtener más datos sobre el rendimiento del *cluster*.

El *benchmark* de POV-Ray

Desde 1994, se desarrolló una metodología para comparar el rendimiento de trazado con POV-Ray en distintos sistemas. Esta metodología consiste en trazar una escena en particular, con parámetros y opciones bien definidos, y tomar el tiempo de trazado de dicha escena. Este tiempo se compara con el obtenido por otros sistemas para darse una idea del desempeño relativo en esta aplicación particular.

El sitio oficial del *benchmark* de POV-Ray, mantenido por *Andrew Haveland Robinson*, está en la página de Intemet *http://www.haveland.com/povbench/index.htm*. La escena consiste en una vasija, con una textura basada en una imagen de nubes, sobre un pedestal, frente a un escenario de paredes reflejantes. La escena es sencilla pero prueba todos los elementos básicos de un programa de *ray tracing*, como son geometría sólida constructiva, aplicación de texturas, y propiedades de materiales tales como reflexión y transparencia. Dicha escena se muestra, con fines de referencia, en la figura 4.5.

Figura 4.5 Escena POV-Ray

4.3 Cómputo paralelo basado en la web

El éxito del WWW ha traído consigo un crecimiento exponencial de la población de usuarios que utiliza el Internet, el número de computadoras conectadas en la misma, y por consiguiente el tráfico que circula sobre ella. Este comportamiento ha producido nuevas demandas de infraestructura para difundir, y acceder a información que el Internet pensado en los años 70 no cubre. Las nuevas demandas que han surgido debido a ese crecimiento, están enfocadas precisamente para solucionar problemas tales como, grandes congestiones en la red, un bajo ancho de banda, altas latencias al momento de extraer información, sobrecarga en servidores, accesos simultáneos masivos a un servidor con temas atractivos que producen partición de la red, entre otros.

Para aminorar los problemas anteriores se ha recurrido al uso de técnicas como *caching* aplicado al Web. El *caching* en el Web reduce las cargas en la red, en los servidores, y las latencias al extraer un documento. Sin embargo, los sistemas de *caching* presentan varias problemáticas y cuestiones de diseño que no son triviales de resolver, y que deben tenerse en cuenta para no producir resultados contradictorios al momento de utilizar cachés en el Web. Algunas de las cuestiones son: la consistencia en la información, la escalabilidad en los sistemas de distribución/acceso a la misma, la organización de los elementos que conforman los sistemas de distribución/acceso en sistemas a gran escala. Éstos son tópicos claves en el acceso a información distribuida, y por tanto son la motivación principal de este proyecto.

El objetivo particular de este proyecto se enfoca en el análisis y diseño de una arquitectura escalable para la distribución de documentos, que ofrezca rapidez de acceso a documentos en el Web, y la mejor consistencia posible. Se busca una arquitectura propuesta a partir de perfiles estudiados de comportamiento que tienen algunos sistemas de caché cooperativo, que mezclan distintos mecanismos de cooperación entre cachés, con diferentes mecanismos de validación de la consistencia de los documentos y estresados con cargas de trabajo en la Web que presentan variantes en el comportamiento.

Dentro de los análisis hechos durante este proyecto, se ha visto la posibilidad de definir una arquitectura de distribución/acceso de documentos, que trabajando bajo un entorno determinado de cooperación entre cachés, un cierto mecanismo de validación de documentos, y un tipo particular de carga de trabajo, ofrezcan resultados excelentes en cuanto a consistencia de documentos y eficiencia percibida por los clientes. Sin embargo, al cambiar alguno de los parámetros anteriores (mecanismo de cooperación, consistencia, o carga de trabajo) pueden producir efectos patológicos que pueden degradar de manera notoria los beneficios de la caché, lo cual puede llegar a ser perjudicial su uso.

La idea de esta arquitectura, no es la de ofrecer los mejores resultados para todo tipo de sistema de cooperación entre cachés, y para cualquier tipo de mecanismo de validación de consistencia de documentos, bajo cualquier tipo de tráfico (situación que según los resultados obtenidos en trabajos que se han realizado con anterioridad no parece posible), si no más bien, es tratar de conciliar diferentes mecanismos de cooperación entre cachés, utilizando diferentes mecanismos de validación de consistencia entre documentos, basado en diferentes comportamientos de cargas de trabajo, que produzca resultados beneficiosos en mayor o menor grado, pero beneficiosos, independientemente de que cambie alguno de los parámetros anteriores. Una parte importante de esta arquitectura es que pueda ser compatible con protocolos existentes para distribución de documentos en el Web (HTTP, NNTP, etc.).

Ejemplo del *Cluster* de Google

Hoy en día, los buscadores de Internet necesitan brindar un buen servicio a las millones de personas que diariamente se conectan a la Web buscando una gran variedad y cantidad de información de todo el mundo. Es por esto, que los buscadores de Internet se han dado a la tarea de buscar nuevas formas para satisfacer a todas esas personas. A continuación se muestra una tabla con algunos de los buscadores de Internet más consultados que muestra el enorme incremento de demanda así como la gran cantidad de información que manejan.

Año	Págs. Web indexadas (millones)	Peticiones por día (millones)	Buscador
1994	0.11	0.0015	WWW Word
1997	2	-	WebCrawler
1997	100	20	AltaVista
2000	1327	70	Google
2003	3083	200	Google

Figura 4.6 Buscadores de Internet

Los principales retos que tienen los buscadores de Internet son:

- Ordenar los resultados por relevancia (*ranking*).
- Tiempos de respuesta cortos (latencia menor de 0.5 segundos).
- Fiabilidad y disponibilidad.
- Actualización regular o frecuente.

Google realiza la petición de búsqueda de la siguiente manera:

- Selecciona al servidor *Google* que va a responder a la petición (balanceo de carga basado en DNS entre varios *cluster*s).
- Busca en un índice invertido que empareja cada palabra de búsqueda con una lista de identificadores de documentos.
- Los servidores de índice intersectan las diferentes listas y calculan un valor de relevancia para cada documento referenciado (establece el orden de los resultados).
- Los servidores de documentos extraen información básica (título, URL, sumario, etc.) de los documentos a partir de los identificadores anteriores (ya ordenados).

La figura 4.7 muestra cómo es que Google realiza la petición de búsqueda:

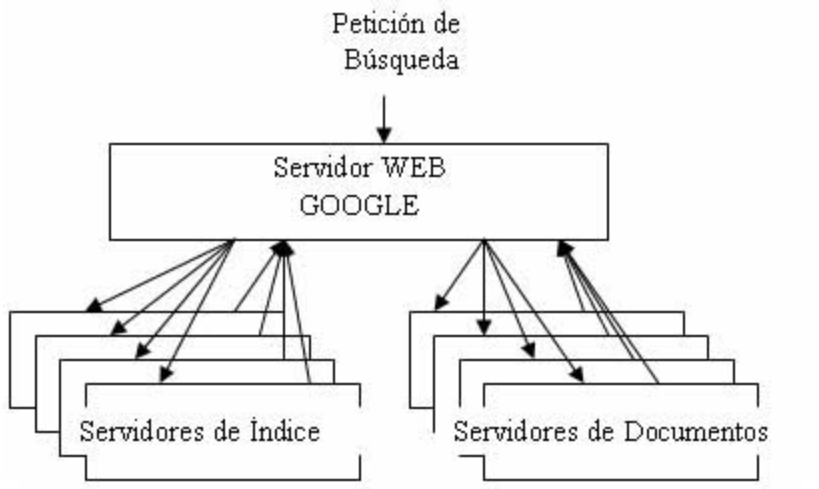


Figura 4.7 Petición de búsqueda de *Google*

Las características de *Google* son:

1. Paralelismo multinivel:

   - A nivel de petición: se resulten en paralelo peticiones independientes.
   - A nivel de término de búsqueda: cada término de búsqueda puede chequearse en paralelo en los servidores de índice.
   - A nivel de índice: el índice de los documentos está dividido en piezas que comprenden una selección elegida aleatoriamente de documentos.

2. Principios de diseño de *Google*.

   - Fiabilidad hardware: existe redundancia a diferentes niveles, como alimentación eléctrica, discos magnéticos en arreglo *RAID*, componentes de alta calidad.
   - Alta disponibilidad y servicio: mediante la replicación masiva de los servicios críticos.
   - Preferencia en precio/rendimiento frente a rendimiento pico: múltiples servidores hardware de bajo coste configurados en *cluster*.
   - PC convencionales: uso masivo de PCs convencionales frente a hardware especializado de alto rendimiento.

3. Infraestructura.

   - Cuenta con múltiples *cluster*s situados principalmente en California y Virginia.
   - Tiene más de 6000 PCs y 12000 discos magnéticos.
   - Las PCs van desde Intel *Celeron* a 533 MHz hasta Intel *Pentium* III a 1.4 GHz con 256 MB DRAM.

- Cada PC cuenta con 1 o varios discos duros de 40 u 80 GB.
- Todos los PCs se organizan en *racks*.
- La red entre PCs dentro de cada *rack* es *Fast Ethernet* (100 Mbps) y de *Gigabit Ethernet* (1000 Mbps) entre diferentes *racks*.
- Cuentan con el sistema operativo Linux.

Un *cluster* tiene la siguiente infraestructura como se muestra en la figura 4.8:

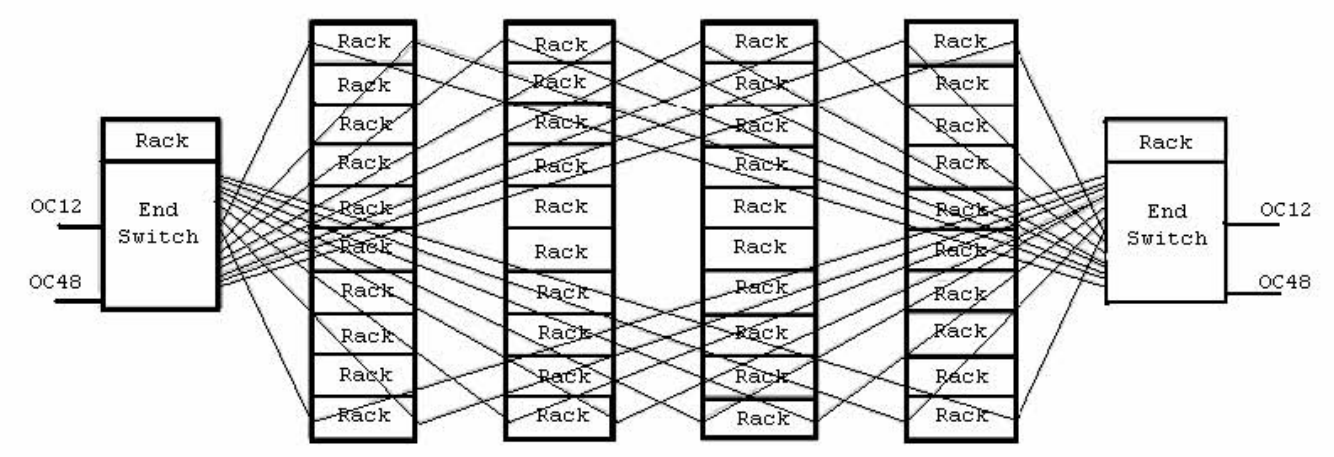


Figura 4.8 Infraestructura de un *cluster*.

- Tiene 40 *racks*, cada uno con 80 PCs (un total de 3600 PCs).
- Conexión a Internet vía un enlace 0C48 (2488 Mbps).
- Conexión a sistema redundante vía un enlace 0C12 (622 Mbps).
- Conexión entre enlaces y *racks* mediante un conmutador de 128 canales *Gigabit Ethernet*.
- Los nodos del *cluster* pueden configurarse para diferentes necesidades: nodos *host* (punto de entrada de *login*), nodos de computación (ejecutan procesos de cálculo) y nodos de E/S (sirven peticiones de E/S).

El *cluster* cuenta con una imagen única que ofrece al usuario la visión de un *cluster* único en vez de una colección de nodos independientes: tiene un punto único de entrada, pues permite al usuario conectarse al *cluster* (*login*) como un único *host* virtual, aunque el *cluster* pueda tener varios nodos físicos que den el servicio de *login*. El directorio *home* del usuario puede residir en un área de almacenamiento compartida o estar distribuida entre los nodos *host*. La figura 4.9 muestra cómo es que un usuario logra acceder al sistema:

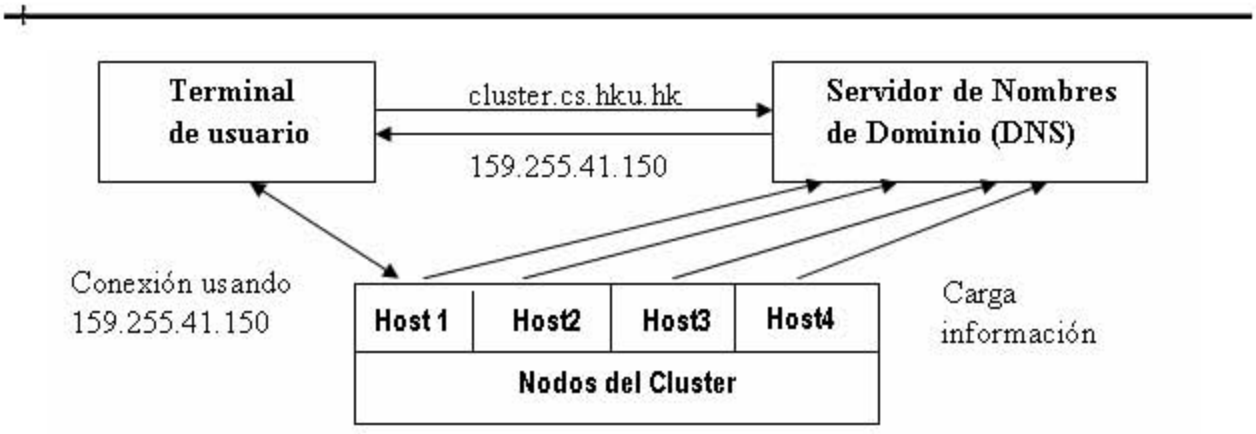


Figura 4.9 Punto de entrada de *login*.

Cuenta con una jerarquía unificada de archivos, de gestión de sistema, del espacio de memoria, de la interfaz de usuario y del planificador de trabajos. Permite la ilusión de un gran sistema de archivos unificado, que de forma transparente, integra los discos magnéticos locales y globales, así como otros sistemas, como las unidades de cinta.

Es un *cluster* de alta disponibilidad pues cuenta con redundancia de *hardware*: salva periódicamente el estado de los procesos (*checkpoint*) en un almacenamiento estable, así, tras el fallo, el sistema se reconfigura para asilar el componente erróneo, recupera el último *checkpoint* y reanuda las operaciones, es decir que reconstruye el estado de ejecución anterior; también cuenta con migración de procesos pues cuenta con un balanceador de cargas dinámico; es de fácil mantenimiento tanto a nivel software como *hardware* y es fiable pues es tolerante a fallos ya que todos los componentes operan en redundancia (ejecutan los mismos procesos). Además cuando uno de los nodos falla, entra otro en su lugar o bien su trabajo se reparte entre los demás nodos.

4.4 Balanceo de cargas y planificación de procesos

Existen dos estrategias de supercomputación con características y requerimientos distintos. La computación de alto rendimiento HPC (*High Performance Computing*) donde se trata de minimizar el tiempo de ejecución de una aplicación utilizando técnicas de programación paralela. Y la computación de alta capacidad de procesamiento HTC (*High Throughput Computing*) cuyo objetivo principal es la ejecución del mayor número de trabajos posibles por unidad de tiempo. Hasta hace pocos años, la comunidad científica utilizaba exclusivamente supercomputadoras para llevar a cabo estos dos tipos de estrategias, lo que requería que un número elevado de grupos de trabajo y personas unieran sus recursos financieros para permitirse una máquina de este tipo. Los usuarios tenían que esperar su turno, y no podían hacer uso de la supercomputadora por tiempo ilimitado. Mientras que este entorno no es apropiado para el usuario, la utilización de recursos es de casi el 100% del tiempo.

Cuando las computadoras se hicieron más pequeñas, más rápidas y más baratas, los usuarios comenzaron a utilizar computadoras personales, así, un solo individuo podía permitirse un recurso de computación que tenía disponible todo el tiempo con acceso exclusivo, pero

obviamente una computadora personal es menos potente que una gran máquina centralizada y además un solo individuo no puede capitalizar toda su potencia computacional, por lo que en una institución habrá muchas máquinas desocupadas durante muchos periodos de tiempo. En la última década se está produciendo una nueva revolución en la computación, fusionando las ventajas de las dos plataformas de trabajo mencionadas, por un lado se tienen computadoras personales de acceso exclusivo y por otro lado, uniendo estas computadoras se consiguen potentes supercomputadoras distribuidas, es lo que se conoce como metacomputación. Un paso adelante en esta estrategia ha sido la de unir intencionadamente con una red de altas prestaciones un conjunto de computadoras personales (*commodity* o también *off-the-shelf technology*), que junto con un software que aglutine las distintas unidades y dé de ellas una única imagen, constituye lo que se conoce como *cluster computing*.

La utilización de este tipo de sistemas es similar a la de las grandes supercomputadoras, y una forma de aprovechar al máximo sus recursos es implementando sobre él una estrategia HPC con una interfaz de paso de mensajes de tipo MPI o PVM y/o una HTC con un sistema de procesado en *batch*, también llamado sistema de colas, que además contribuirá a dar una imagen única del conjunto de procesadores.

Los sistemas *batch* proporcionan un mecanismo para encolar, controlar y ejecutar trabajos en un recurso compartido como lo es la multicomputadora, proporcionando un acceso centralizado a una serie de recursos distribuidos. Sin embargo, deben hacer mucho más que proporcionar una visión global del sistema, además deben procurar utilizar los recursos de cálculo de una manera inteligente, segura, justa y eficaz. Estas complejidades pueden conducir a la degradación de las prestaciones y a desigualdades significativas en uso. Por ello, un buen sistema de colas debe ser tolerante al fallo de los trabajos, adaptarse a la carga de las CPU y poder migrar procesos. Todo ello se verá complementado con un planificador (*scheduler*) que será el encargado de determinar cuándo, dónde, y cómo los trabajos se ejecutan para maximizar el rendimiento de la máquina.

4.4.1 Planificación de procesos

Dos de las partes más críticas de un *kernel* son el subsistema de memoria y el planificador. Esto es debido a que estas dos piezas influyen en el diseño y afectan al comportamiento de prácticamente todos los demás elementos del *kernel* y del sistema operativo. Ésta es también la razón por la que es deseable que estas dos piezas tengan un funcionamiento absolutamente correcto y un comportamiento óptimo. El *kernel* de Linux se utiliza en todo tipo de máquinas, desde pequeños dispositivos embebidos hasta grandes computadoras (*mainframes*). No importa lo bueno que sea el diseño del *kernel*, siempre habrá quien piense que algunas categorías de procesos han tenido un trato injusto.

Para entender cómo el planificador asigna el tiempo de CPU y administra la ejecución de los programas, es necesario entender el concepto de proceso. Un proceso es un conjunto de instrucciones que el procesador ejecuta secuencialmente. Uno o más procesos pueden constituir un programa.

Los procesos pueden tener uno de cinco estados: en ejecución (*running*), interrumpible (*interruptible*), no *interrumpible* (*uninterruptible*), zombi (*zombie*) o detenido (*stopped*). Un proceso en el estado de ejecución, está o bien siendo ejecutado por el procesador, o almacenado en la memoria esperando ser ejecutado. Un proceso interrumpible se encuentra actualmente bloqueado. El bloqueo ocurre cuando un proceso voluntariamente cede el control de la CPU hasta que se cumpla determinada condición. Esto típicamente ocurre cuando un proceso está esperando una entrada del usuario, disco duro, red u otro recurso externo.

Los procesos pueden tener varios niveles de prioridad. El planificador de Linux utiliza un algoritmo eficiente para favorecer a los procesos de alta prioridad, a la vez que permite ejecutarse a los procesos de baja prioridad. El planificador mantiene una lista de los niveles de prioridad. Cuando llega el momento de seleccionar un proceso, el planificador busca el nivel de prioridad más alto que contenga procesos disponibles. Luego selecciona un proceso del nivel de prioridad deseado. Dado que el número de niveles de prioridad es fijo, el planificador siempre demora una cantidad constante de tiempo.

El planificador tiene otra capacidad importante: expropiación (*preemption*). Esto permite que un proceso pueda ser detenido en cualquier momento, permitiendo que un proceso de mayor prioridad pueda ser iniciado. Esto es muy importante desde el punto de vista del usuario, ya que permite que los procesos que interactúan con el usuario sean ejecutados cuando sea necesario, mientras permite que los procesos de baja prioridad que no interactúan con el usuario se ejecuten en segundo plano (*background*). Por ejemplo, imagine que está ejecutando un cliente de computación distribuida. Este programa usa la mayor parte de su tiempo de CPU cuando usted no está frente a su computadora. En el momento en que usted comienza a utilizar un programa tal como un navegador Web, este último desaloja al cliente de computación distribuida. Su sistema no parecerá lento o pesado, incluso si usted tiene un proceso computacionalmente intensivo ejecutándose en segundo plano. Los beneficios de la expropiación son tan grandes que los desarrolladores del *kernel* decidieron hacer que el *kernel* mismo sea expropiable (*preemptible*). Esto permite a una tarea del *kernel*, tal como E/S de disco ser expropiado, por ejemplo, por un evento de teclado. Esto le permite al sistema tener mejores tiempos de respuesta a las demandas del usuario. El *kernel* es capaz de administrar eficientemente sus propias tareas además de los procesos de usuario.

El planificador de Linux intenta conseguir varios objetivos:

1. *Ecuanimidad*: El planificador debe repartir la CPU entre todos los procesos de forma ecuánime. Se ha trabajado en el *kernel* para garantizar un reparto imparcial del tiempo de CPU entre los procesos.

2. *Volumen de producción y eficiencia*: El planificador debe intentar maximizar tanto el volumen de producción (*throughput*) como la eficiencia (utilización de la CPU). La forma habitual de incrementar la utilización de la CPU es incrementando el nivel de multiprogramación. Pero esto es beneficioso sólo hasta un cierto punto, a partir del cual se vuelve contraproducente.

3. *Mínima Sobrecarga*: Un planificador debe estar en ejecución el menor tiempo posible. La latencia del planificador debe ser mínima. Pero ésta es la parte difícil. Generalmente la planificación en sí se considera trabajo no útil. Ahora bien, si la planificación se hace de forma correcta aún a costa de consumir algo más de tiempo puede valer la pena. Pero ¿cómo decidimos cuál es el punto óptimo? La mayoría de los planificadores resuelven este problema utilizando políticas heurísticas.

4. *Hacer cumplir una planificación basada en prioridades*: Planificación basada en prioridades significa que algunos procesos tienen precedencia sobre otros. Como mínimo, el planificador debe distinguir entre procesos limitados por E/S y procesos limitados por CPU. Además, debe implementarse algún mecanismo que tenga en cuenta la edad de los procesos para evitar que algunos no tengan acceso a la CPU (inanición). Linux respeta las prioridades y distingue entre varias categorías de procesos. El *kernel* de Linux distingue entre trabajos planificados vía *batch* y tareas interactivas. Todos ellos obtienen una cuota de CPU de acuerdo a sus prioridades. Probablemente alguien ha utilizado alguna vez el comando `nice` para cambiar la prioridad de un proceso.

5. *Tiempo de turnaround (tiempo de espera):* El tiempo de *turnaround* es la suma del tiempo de servicio y el tiempo de espera en la cola de ejecución (*ready*). El planificador intenta reducir ambos.

6. *Tiempo de respuesta y variabilidad:* El tiempo de respuesta de un programa debería ser lo menor posible. Además, otro factor importante, que a menudo se olvida, es la variabilidad entre los tiempos de respuesta. No es aceptable que el tiempo de respuesta medio sea bajo pero que algún usuario obtenga ocasionalmente un retraso de, digamos, diez segundos ejecutando un programa interactivo es aceptable.

7. *Miscelánea:* El planificador intenta también cumplir otros objetivos como la predictibilidad. El comportamiento del planificador debe ser predecible para un conjunto dado de procesos con prioridades asignadas. El rendimiento del planificador debe decaer suavemente al aumentar la carga. Esto es especialmente importante en el caso de Linux al ser muy popular como servidor, ya que los servidores tienden a resultar sobrecargados durante los picos de mayor tráfico.

La habilidad de manipular distintas cantidades de recursos computacionales, desde pequeñas CPUs empotradas a supercomputadoras masivas de 64 procesadores es llamada escalabilidad; y es una de las mayores fortalezas del *kernel*. Uno podría pensar que un planificador diseñado para administrar una CPU nunca podría ser adaptado para administrar 64 CPUs; sin embargo, el planificador de Linux puede administrar sistemas de muchos procesadores a través del uso de listas especiales llamadas colas de ejecución. Una cola de ejecución almacena información acerca de los procesos que se están ejecutando en una sola CPU; habiendo una cola de ejecución por cada CPU en el sistema. La información contenida en las colas de ejecución le permiten al planificador transferir el control de una CPU hacia otra, usando un método denominado balance de carga.

El balance de carga es una manera de asegurar que los recursos de una CPU no serán desperdiciados mientras otra CPU se encuentra sobrecargada. Si el planificador encuentra que

una cola de ejecución tiene muchos más procesos en ella que otra, uno o más procesos pueden ser movidos desde la cola más larga hacia la más pequeña. El balanceador de carga es invocado cada vez que se vacía una cola de ejecución; si no hay colas de ejecución vacías, es invocado periódicamente. El temporizador periódico le permite al sistema mantener un balance de carga razonable a través de varias CPUs sin tener que dedicar demasiado tiempo a mover procesos de una CPU hacia otra. Balancear la carga cuando una cola de ejecución se vacía permite al planificador asegurar que el precioso poder de CPU nunca va a desperdiciarse. Hay una excepción a esta regla del balanceo: algunos procesos especiales pueden ser fijados a una cierta cola de ejecución. Este atributo es llamado afinidad de hilos (*thread affinity*), un hilo (*thread*) es simplemente otro nombre para proceso. Es importante notar que si el soporte de Multiprocesamiento Simétrico (*SMP, symmetric multiprocessing*) no se encuentra habilitado cuando se compila el *kernel*, no será habilitado el código de balance de carga y no se desperdiciará tiempo tratando de balancear la carga de una única CPU.

La cantidad de tiempo de CPU que el planificador asigna a un proceso particular es llamada una porción de tiempo (*timeslice*). Luego de que la porción de tiempo de un proceso ha sido utilizada, el proceso es detenido para que pueda ejecutarse el siguiente. Es importante recordar que un proceso puede ser detenido en medio de su porción de tiempo; este es el propósito de la expropiación. A distintos procesos les son asignadas distintas porciones de tiempo basadas en la prioridad; los procesos de alta prioridad se ejecutan una mayor cantidad de tiempo que los de baja prioridad. La prioridad no es un concepto unitario; cada proceso tiene una prioridad estática y una dinámica.

La prioridad estática, o amabilidad (*niceness*) en la terminología tradicional de UNIX es la medida de cuán importante es un proceso. Puede ser fijada por el usuario o por otros programas. A los procesos con un valor bajo de amabilidad se les asigna una porción de tiempo mayor; a aquellos con valores altos, se les asignan porciones de tiempo mas pequeñas. El hecho de que los procesos de mayor prioridad tengan valores de amabilidad más baja puede resultar confuso si uno ve a la amabilidad como una medida de prioridad; en cambio, piense en la amabilidad como una medida de la disposición del proceso a ceder ante otros. Los valores de amabilidad pueden ser fijados desde la línea de comando con el comando `nice`, o a través de algunos programas de monitoreo del sistema.

La prioridad dinámica de un proceso es determinada por el planificador monitoreando su comportamiento durante la ejecución. Los procesos que gastan mucho de su tiempo bloqueándose son conocidos como procesos cota de E/S (*I/O bound*); su comportamiento está acotado por la entrada y la salida. Cuando el planificador reconoce a un proceso cota de E/S se le asigna una bonificación negativa, y por lo tanto una mayor porción de tiempo. En contraste, a los procesos cota de CPU se les asigna una bonificación positiva, y por lo tanto una porción de tiempo menor. Esto previene que los procesos cota de CPU controlen el procesador, y permite a las entradas y salidas desarrollarse sin problemas. A veces, un proceso puede variar entre comportamientos cota de E/S y cota de CPU.

Por ejemplo, un proceso puede leer cierta información desde el teclado y realizar algunas computaciones basadas en esa entrada. El planificador necesita constantemente estar alerta del comportamiento del proceso; las prioridades dinámicas pueden ser reajustadas de acuerdo a esto.

El planificador lleva un registro de todas las porciones de tiempo con dos listas de procesos. La primera lista contiene a los procesos que todavía disponen de tiempo; la segunda contiene aquellos procesos que ya han agotado su tiempo. Cuando un proceso usa su porción de tiempo, el planificador calcula una nueva porción añadiendo la bonificación de prioridad dinámica a la prioridad estática. El proceso es luego insertado en la segunda lista. Cuando la primera lista se vacía, la segunda lista la reemplaza y viceversa. Esto permite al planificador calcular continuamente las porciones de tiempo con una sobrecarga computacional mínima.

¿Por qué debe ser tan cuidadoso el planificador cuando calcula porciones de tiempo? Un buen planificador debe alcanzar un balance apropiado entre caudal y latencia. Caudal (*throughput*) es la cantidad de datos que pueden ser transferidos desde una ubicación hacia otra. Latencia es el tiempo que le toma a un proceso responder a una entrada. Este balance es alcanzado ajustando las porciones de tiempo. Un proceso cota de E/S necesita buen caudal para cumplir sus tareas rápidamente. Esta es la razón por la cual el planificador le da a este tipo de procesos porciones de tiempo mayores; ellos tienen que hacer y responder a los requerimientos de E/S, y no tiene que esperar demasiado para que otros procesos se ejecuten. Dado que casi todos los procesos pueden beneficiarse de un caudal superior. ¿Por qué no darles a todos los procesos una porción de tiempo mayor? Si un planificador hiciera esto, sufriría la latencia. Dado que cada proceso toma un gran tiempo para completar su tarea, otros procesos no serán capaces de responder rápidamente a la entrada del usuario. Un buen balance entre caudal y latencia llevan a una sensación de respuesta veloz para el usuario con el suficiente caudal.

4.4.2 OpenMosix

El sistema *openMosix* es una extensión del *kernel* del sistema operativo GNU/Linux. Permite que varios equipos compartan la carga de sus respectivas aplicaciones de forma totalmente transparente al usuario y que amplía grandemente sus posibilidades en computación paralela, a la vez que implementa una arquitectura SSI. Hay muchas variedades de *clusters* y un *cluster* SSI tiene copias de un único *kernel* de sistema operativo en varios nodos que permiten ver el conjunto como un todo. Esto aporta ventajas a diversos niveles, entre los cuales se cuenta una mayor facilidad de uso y transparencia frente al usuario, procesos y los propios nodos.

El *openMosix* es una tecnología que permite transparencia total en la ejecución de aplicaciones en ambientes paralelos. Este entorno permite la administración de los procesos. Su planificación se basa en los recursos del *cluster* (tanto de procesador, como de memoria).

Un *cluster* puede verse desde el usuario y/o el programador como un conjunto de máquinas trabajando juntas para un mismo propósito computacional, esto pues significa que se tiene un especial esfuerzo de los desarrolladores para generar código fuente paralelizado, un esfuerzo de los usuarios, y esfuerzo del administrador del sistema. Los *clusters* están en auge en las áreas científicas y universidades ya que requieren gran poder computacional y al menor precio posible. Usar un *cluster* es un poco más complicado que usar una sola máquina. Para la mayor parte de la gente el hito es adquirir una máquina más potente, más rápida; pero siempre llegará al punto en el que el mercado no disponga de un procesador más rápido del que ya tenemos. Aquí es donde entran en juego los equipos multiprocesador. Administrar y utilizar un sistema multiprocesador es

muy fácil, pero las máquinas multiprocesador no escalan muy bien. De hecho, debido a los accesos a memoria y al acceso a otros recursos, no pueden escalar del todo. Aquí es donde aparecen los sistemas NUMA. Existen varias diferencias con las arquitecturas multiprocesador, básicamente, cada procesador tiene su memoria local. NUMA tiene numerosas ventajas sobre las máquinas normales, la mayoría de ellas son más poderosas que los sistemas de un solo procesador y que los multiprocesadores normales. Así, tenemos que ni los multiprocesadores ni los monoprocesadores son suficientemente rápidos para las tareas que queramos ejecutar, pero NUMA es caro. La solución es *openMosix*, un proyecto de *clustering* SSI libre. La idea principal del SSI es poder ver nuestro sistema como una sola computadora, donde se necesita de un administrador del *cluster*, pero el sistema se ofrece al programador como un único sistema monoprocesador.

Primero fue MOSIX desde los años 70. MOSIX permite SSI puro en entornos POSIX, con migración transparente de tareas y equilibrado de carga automático Pero la política del profesor A. Barak, quien ingenió MOSIX no permitió la colaboración de nadie que quisiera hacerlo, y la incredulidad de muchos desarrolladores estancaron *openMosix* en las sombras durante 25 largos años. Cuando MOSIX se portó a Linux debido a la estructura vertical de la GPL, la parte del *kernel* de MOSIX fue licenciado bajo GPL. Un tiempo después, debido a diversas presiones el profesor Barak licenció también la parte del área de usuario (herramientas de usuario) del proyecto MOSIX. Durante los siguientes años y hasta el 2001, MOSIX fue un proyecto muy jerarquizado, pero su licencia era libre y gratuita. El profesor Barak nunca aceptó ninguna ayuda externa ni parches. La apertura del proyecto le dio mucha fama y la posibilidad de mucho dinero de la industria de IT.

Éste fue el proyecto de su vida, y lo hizo libre, después de 2 años, teniendo el control sobre MOSIX fue cada vez más duro. Mientras tanto, el alumno más aventajado del profesor Barak, Dr. Moshe Bar, empezó a estar más y más descontento sobre el futuro comercial de MOSIX, Moshe Bar empezó un nuevo proyecto de *clustering* alzando la empresa Qlusters, Inc. en la que el profesor A. Barak decidió no participar ya que no quería poner MOSIX bajo licencia GPL y rechazó más ayudas de la industria y a más gente que quería ayudarle. Finalmente el proyecto MOSIX dejó el mundo libre, ahora MOSIX es gratuito, pero no es libre. Como había una significativa base de usuarios clientes de la tecnología MOSIX (unas 1000 instalaciones a lo ancho del planeta) Moshe Bar decidió continuar el desarrollo de MOSIX pero bajo otro nombre, *openMosix*, totalmente bajo licencia GPL2. El Dr. Moshe Bar es ahora el líder del proyecto *openMosix* y tiene a muy buenos programadores trabajando en el proyecto. *OpenMosix* está respaldado y siendo desarrollado por personas muy competentes y respetadas en el mundo del open source, trabajando juntas en todo el mundo.

El *cluster openMosix* tiene la función de repartir la carga de procesos entre los nodos. Tiene la capacidad de ampliarse fácilmente añadiendo más PCs al *cluster*. Además, ante la caída de alguna de las computadoras del *cluster* el servicio se puede ver mermado, pero mientras haya computadoras en funcionamiento, éste seguirá.

El algoritmo interno de balanceo de carga migra, transparentemente para el usuario, los procesos entre los nodos del *cluster*. La principal ventaja es una mejor compartición de recursos entre nodos, así como un mejor aprovechamiento de los mismos. El *cluster* escoge por sí mismo la

utilización óptima de los recursos que son necesarios en cada momento, y de forma automática. Esta característica de migración transparente hace que el *cluster* funcione a todos los efectos como un gran sistema SMP (*Symmetric Multi Processing*) con varios procesadores disponibles. Su estabilidad ha sido ampliamente probada aunque todavía se está trabajando en diversas líneas para aumentar su eficiencia. El punto fuerte de este proyecto es que intenta crear un estándar en el entorno del *clustering* para todo tipo de aplicaciones HPC.

OpenMosix es un conjunto de parches que se aplican al núcleo o *kerne*l de Linux de manera que los procesos, en principio no todos realmente, sólo aquellos que no utilizan memoria compartida y los que no usan recursos locales al nodo (esto se soluciona con este parche), son capaces de migrar entre una serie de nodos conectados entre sí por red. Una de ellas es *plumpos*, esta distribución permite añadir nodos a un *cluster openMosix* aunque tiene la limitación de que necesita una instalación previa de un nodo *openMosix* que no sea *plumpos* que actúe como lanzador de procesos, uno de sus puntos fuertes es su reducido tamaño. *Plumpos* incluye las herramientas básicas de administración de un nodo o*penMosix*, así como las más básicas herramientas Linux. Otra de las distribuciones es *clusterKnoppix* que no es más que una distribución Knoppix modificada para que arranque con un *kernel* con los parches *openMosix* aplicados, esta distribución no necesita un nodo con una distribución distinta, pero tiene el inconveniente de ser una distribución de más de 600 MB. Esta última distribución incluye un amplio abanico de aplicaciones de gestión del *cluster*, tanto las que ya incluía *plumpos* como las últimas versiones de herramientas gráficas (en parte por el hecho de que *clusterKnoppix* incluye XWindow al contrario que *plumpos*). Quizás una mezcla de estas dos distribuciones puede permitir el montaje de un *cluster openMosix* de una manera sencilla, ambas distribuciones soportan el descubrimiento de nuevos nodos en automático, por lo que la configuración necesaria es mínima. Aunque estas distribuciones hacen bien lo que se espera de ellas, añadir/montar nodos *openMosix* rápido y fácilmente, nada como un *cluster* configurando los nodos uno a uno y ajustado a las características particulares del *cluster* que se intenta crear.

Knoppix es una distribución de Linux, basada en GNU Debian Linux, pero con la cualidad, que puede correr desde un *cdrom*, sin necesidad de ser instalada, ni de utilizar un disco rígido. Knoppix es un CD *booteable* con una colección de programas GNU y detección automática de hardware. Knoppix puede ser usado como una demo de Linux, CD educacional, sistema de rescate, o adaptado y usado como plataforma comercial de demos de productos. No es necesario instalar nada en el disco duro. Debido a la descompresión on-demand, el CD tiene casi 2 GB de programas ejecutables instalados en él.

Una vez que arranca, nos deja en una sesión de Xfree, con KDE 3.1, y cantidad de programas para usar, entre ellos OpenOffice y gran variedad de herramientas para configuración de red, GIMP (herramienta similar a Photoshop), varios clientes de correo electrónico, mensajeros instantáneos, etc. Por lo tanto, uno puede ir, agarrar una máquina sin disco rígido, con un poco de RAM, arrancar con el CD de Knoppix, y en cuestión de minutos tener una terminal Xwindows, con todas las aplicaciones necesarias para trabajar normalmente en una oficina.

CONCLUSIONES

La computación es sin duda una de las mejores herramientas que el hombre ha sido capaz de desarrollar a lo largo de su historia, y ha sido una de las principales fuentes de investigación y constante desarrollo para poder aplicarla en prácticamente cualquier entorno en donde el hombre busque incrementar las capacidades que éste le pueda proporcionar.

Es por tal motivo que la computación está en constante evolución, cada día se descubre algo nuevo, o se realizan mejoras a los desarrollos ya implementados, para poder proporcionar un mayor desempeño en las actividades para las cuales cada una de estas aplicaciones fue desarrollada. Uno de los grandes avances que ha mejorado a la computación ha sido lo que ya conocemos hasta esta parte del presente documento, la computación paralela, que ha venido a revolucionar prácticamente el concepto que se tenía de programación, dándole un cambio a la forma en que las aplicaciones pueden ser procesadas.

El cómputo paralelo si bien ya tiene algunos años desde que fue inventado, aún existen muchas personas que no conocen respecto al tema o las cuales no les queda muy claro cómo es que éste trabaja, qué aplicaciones y ambientes nos proporcionan un cómputo paralelo, es por lo anterior que nosotros hemos realizado este compendio que puede ayudar no sólo a nuestra comunidad estudiantil de la Facultad de Ingeniería sino a todas aquellas personas que estén interesadas en aprender y comprender conceptos básicos, aplicaciones sencillas y entornos de programación en donde el tema principal es el cómputo paralelo.

Este documento proporciona una visión más detallada de lo que es el cómputo paralelo, así como los conceptos fundamentales que giran entorno al cómputo de alto desempeño, para brindar una fuerte noción a aquellos que posean interés en comprender o estudiar a fondo el cómputo de alto desempeño, pues a partir de la lectura de este documento, el lector será capaz de comprender y si así se quiere profundizarse en actividades más complejas del cómputo de alto desempeño.

Lo que se realizó fue explicar de una forma sencilla y entendible los principios y fundamentos para generar el interés por el cómputo de alto desempeño o en su defecto, para poder proporcionar un material didáctico que ayude como una guía al alumno y a cualquier otra persona que quiera incursionar en el campo del cómputo de alto desempeño.

Este trabajo nos permite conocer mejor el ambiente del cómputo paralelo que hasta el momento ha tenido gran éxito en las aplicaciones y procesos que necesitan de una mayor capacidad de cómputo, un alto rendimiento y una alta disponibilidad, así como también de una mayor flexibilidad para construir sistemas computacionales paralelos que nos permitan contar con estas características sin la necesidad de recurrir a adquirir un equipo de un costo elevado. Así como a tener el contacto con lo entornos que nos permiten realizar este tipo de aplicaciones, comprendiendo sus beneficios, identificando sus limitaciones, pero sobre todo induciendo a profundizar en el tema.

Para generar un valor agregado y hacer de este material algo atractivo, también se realizaron distintas prácticas que se presentan en el anexo, para dar a entender de forma sencilla y clara el proceso de comenzar a generar un entorno en donde el lector pueda de una forma inmediata observar los beneficios de los sistemas paralelos, es por ese simple hecho que muchas veces no se encuentra en otros documentos que se detalla con precisión cada paso por sencillo que éste sea, llevando de la mano al lector para poder realizar diversas prácticas relacionadas con el cómputo paralelo.

Estas prácticas nos permiten desarrollar desde cero, el cluster más económico y sencillo que puede haber de tan sólo dos nodos, un cliente y un servidor, proporcionando también una explicación para poder convertir este sencillo sistema en un verdadero sistema paralelo, con una capacidad de cómputo grande, pues también se indica cómo conectar más equipos y así mejorar el pequeño pero ilustrativo cluster tipo Beowulf.

En otras de las prácticas, se detallan la utilización de los entornos y librerías de programación que nos permiten tener la posibilidad de ejecutar y programar aplicaciones de forma paralela, dándonos la oportunidad de verdaderamente comprobar que las aplicaciones están siendo ejecutadas en más de un procesador, induciendo así al lector a continuar investigando y enriqueciéndose de más información en lo que a la programación paralela se refiere así como el tipo de aplicaciones que se pueden desarrollar y paralelizar .

Si bien la programación paralela es en parte complicada, es posible hacerla atractiva una vez que se comienza a tener práctica en la misma, y es por eso que a través de este material, se ayuda a reducir esa complejidad brindando ejemplos prácticos que permitan ver en acción al cómputo paralelo.

ANEXO

Debido a que el objetivo principal de esta tesis es proporcionar un material didáctico que sirva de apoyo a todas aquellas personas interesadas en los sistemas paralelos/distribuidos, hemos decidido incluir en este trabajo un anexo con prácticas de laboratorio ya que el temario de la asignatura de Cómputo de Alto Desempeño no las contempla y creemos que son importantes.

Las prácticas de laboratorio permitirán al alumno reforzar y aplicar los conocimientos básicos de la teoría realizando ejercicios e investigaciones que complementarán lo aprendido en clase.

La primera práctica consiste en la construcción de un cluster tipo Beowulf bajo el sistema operativo Linux la cual no sólo permite conocer la configuración e instalación de un cluster, sino que, además de que los alumnos se relacionen con sistemas paralelos/distribuidos, permitirá ver cuáles son las ventajas por las cuales la tendencia a utilizar clusters como una alternativa a la gran demanda de poder de cómputo se ha incrementado. La segunda práctica muestra la instalación y configuración de bibliotecas paralelas que permitirán trabajar con ambientes paralelos/distribuidos, siendo el complemento de la primera práctica.

En la tercera práctica el alumno podrá aplicar los paradigmas de la programación paralela para aprender a paralelizar programas secuenciales. Así, en la cuarta práctica, los alumnos tendrán que aplicar cada una de las etapas en la creación de programas paralelos teniendo en cuenta las ventajas y desventajas de cada uno de los paradigmas.

Las prácticas 5 y 6 introducen al alumno a la programación paralela, utilizando algunas de las funciones básicas de MPI, una de las bibliotecas de programación paralela instaladas en la práctica número 2 y que permiten que el alumno trabaje y comprenda el funcionamiento de los sistemas de alto desempeño.

En las prácticas 7 y 8 el alumno investigará sobre temas importantes en la aplicación de los clusters como lo son las Grids de computadoras o la renderización de imágenes, la integración de varios sistemas o brindar una solución a la gran demanda de cómputo, temas que dentro del cómputo de alto desempeño son novedosos y muy utilizados hoy en día.

PRÁCTICA # 1
CONSTRUCCIÓN DE UN CLUSTER TIPO BEOWULF (PARTE I)

OBJETIVO

El alumno conocerá la configuración e instalación de un cluster tipo Beowulf utilizando software libre.

Al final de esta práctica el alumno:
Construirá un cluster tipo Beowulf.
Estará familiarizado con los ambientes paralelos/distribuidos.

CUESTIONARIO PREVIO

1. ¿Qué es un cluster?
2. ¿Qué es una dirección IP?
3. ¿Para qué sirve un tarjeta de red?
4. ¿Qué es el servicio NFS?
5. ¿Qué es el servicio NIS?

Requerimientos para la realización de la práctica:

- Si se va a trabajar con tan solo dos máquinas, necesitarán estar conectadas a través de un cable cruzado para poder comunicarse; una de ellas será configurada como servidor y la otra como cliente. Si se va a trabajar con más de dos máquinas, necesitarán estar todas conectadas a un *switch*; una máquina será configurada como servidor y las restantes como clientes.
- Las máquinas deberán tener instalado alguna versión de *Red Hat* o *Fedora Core*.
- Se deberán contar con los discos de la distribución Linux instalada.

EJERCICIOS PROPUESTOS

1. ¿Qué es un cluster tipo Beowulf?
2. Menciona las ventajas y desventajas del cluster tipo Beowulf.
3. Menciona algunas aplicaciones del los clusters.
4. Configuración del Servidor y los Clientes.

El sistema operativo Linux está conformado por varios directorios los cuales se encuentran organizados jerárquicamente, permitiendo una mayor funcionalidad al sistema operativo. Dentro de esta estructura podemos encontrar directorios especiales, los cuales contienen archivos que permiten realizar funciones específicas para el sistema operativo. La razón por la cual están ordenados los archivos en directorios especiales es para que el sistema sepa en

qué lugar están archivos para tareas específicas como bibliotecas, documentación, manuales, etc., y pueda acceder e ellos libremente.

El directorio */etc*, contiene a los archivos de configuración locales. Para construir el cluster, primero hay que configurar la tarjeta de red dentro de uno de los archivos de este directorio. Linux, contiene dentro de este directorio archivos para la configuración de cada una de las interfaces de red, ya sea para activarlas o desactivarlas. Estos archivos se encuentran dentro del directorio */etc/sysconfig/network-scripts* y en él, se encuentra el archivo *ifcfg-eth0*, archivo de configuración de la tarjeta red.

Generalmente este archivo contiene:

```
DEVICE=eth0
BOOTPROTO=dhcp
ONBOOT=yes
```

Para visualizar el contenido de este archivo utilizamos el comando *cat* de la siguiente manera:

```
$ cat /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE=eth0
BOOTPROTO=dhcp
ONBOOT=yes
```

Para nuestra práctica modificaremos este archivo con el comando *vi* de la siguiente manera:

```
$ vi /etc/sysconfig/network-scripts/ifcfg-eth0
```

El archivo deberá contener las siguientes líneas:

```
DEVICE= eth0
ONBOOT=yes
BOOTPROTO=static
BROADCAST=192.168.1.255
IPADDR=192.168.1.1
NETMASK=255.255.255.0
NETWORK=192.168.1.0
```

donde:

`DEVICE= eth0` → identifica al dispositivo por el cual se conectará a la red. En este caso de utilizará la tarjeta de red.

`ONBOOT=yes` → la interfaz se activa al momento de arrancar el sistema.

`BOOTPROTO=static` → la dirección IP se asigna manualmente.
`BROADCAST=192.168.1.255` → dirección IP de Broadcast a todas las máquinas de a subred 1.
`IPADDR=192.168.1.1` → dirección IP de la máquina 1.
`NETMASK=255.255.255.0` → identifica la máscara de red.
`NETWORK=192.168.1.0` → identificador de red.

Para modificar el archivo debemos entrar al modo de inserción presionando la tecla *i*. Para guardar los cambios y salir del editor *vi*, hay que presionar la tecla *esc*, después *:wq*, y presionar la tecla de *enter*.

Una vez guardados los cambios, dentro del directorio */etc/sysconfig* se encuentra el archivo *network*. La función de este archivo es proporcionar información sobre la configuración de la red deseada.

Generalmente, este archivo contiene:

```
NETWORKING=yes
HOSTNAME=localhost.localdomain
```

Para nuestra práctica modificaremos este archivo, el cual deberá contener las siguientes líneas:

```
NETWORKING=yes
HOSTNAME=miservidor
GATEWAY=192.168.1.1
```

donde:

`NETWORKING=yes` → se trabajará con una conexión a red.
`HOSTNAME=miservidor` → nombre canónico con el que se identificará a cada máquina.
`GATEWAY=192.168.1.1` → dirección IP de la máquina que se utilizará como puerta de enlace, generalmente es la IP del servidor.

Dentro del directorio */etc* se encuentra el archivo *hosts*. Este archivo se usa para resolver nombres de hosts, es decir, establece un relación de cada host que integra la red con su dirección IP que la identifica dentro de la misma. Para hacer esto asocia en un renglón el nombre canónico de cada máquina con su dirección IP. Para nuestra práctica, este archivo debe contener todas las direcciones IP de las máquinas que conformarán el cluster:

```
$  vi /etc/hosts
192.168.1.1 Miservidor
192.168.1.2 cliente1
192.168.1.3 cliente2
192.168.1.4 cliente3
```

Una vez configurados los archivos de red, hay que reiniciar el servicio de red para que tomen los nuevos valores. Linux cuenta con un script que levanta el servicio de red; este archivo se encuentra en */etc/rc.d/init.d* y se llama *network*. La siguiente línea muestra como se reinicia el servicio:

```
$ /etc/rc.d/init.d/network restart
Interrupción de la interfaz de loopback:                   [  OK  ]
Configurando parámetros de red:                            [  OK  ]
Activación de la interfaz de loopback:                     [  OK  ]
Activando interfaz eth0:                                   [  OK  ]
```

Los archivos */etc/sysconfig/network-scrips/ifcfg-eth0*, */etc/sysconfig/network* y */etc/hosts*, deben ser modificados en <u>todas las máquinas</u> que formarán parte del cluster (servidor y clientes).

Para verificar los cambios, se ejecuta el comando *ifconfig* de la siguiente manera:

```
$ ifconfig
```

```
eth0      Link encap:Ethernet  HWaddr 00:10:B5:82:EF:97
          inet addr:192.168.1.2  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:240 (240.0 b)
          Interrupt:5 Base address:0x5000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:28778 errors:0 dropped:0 overruns:0 frame:0
          TX packets:28778 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1965510 (1.8 Mb)  TX bytes:1965510 (1.8 Mb)
```

Para ver si todas las máquinas se pueden comunicar, Linux cuenta con el comando *ping* el cual se ejecuta de la siguiente manera:

```
$ ping 192.168.1.2
PING cliente1 (192.168.1.2) 56(84) bytes of data.
64 bytes from cliente1 (192.168.1.1): icmp_seq=1 ttl=64 time=0.205 ms
64 bytes from cliente1 (192.168.1.1): icmp_seq=2 ttl=64 time=0.084 ms
64 bytes from cliente1 (192.168.1.1): icmp_seq=3 ttl=64 time=0.088 ms
64 bytes from cliente1 (192.168.1.1): icmp_seq=4 ttl=64 time=0.087 ms
64 bytes from cliente1 (192.168.1.1): icmp_seq=5 ttl=64 time=0.085 ms

--- cliente1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3996ms
rtt min/avg/max/mdev = 0.084/0.109/0.205/0.049 ms
```

con el comando anterior se hace un *ping* al cliente 1.

Una vez que todas las máquinas logran comunicarse, hay que levantar el servicio NFS. El servicio NFS (*Network File System*) monta en una máquina Linux un sistema de archivos vía red. NFS permite centralizar archivos en una localización, mientras se permite su acceso continuo a los usuarios autorizados. Una de las ventajas principales de esta aplicación es el ahorro de espacio en disco de las máquinas que tienen que acceder frecuentemente a un conjunto de información importante para realizar variadas tareas.

Para levantar este servicio, en la máquina que será el servidor, hay que modificar el archivo *exports* dentro del directorio */etc*. Este archivo nos permite especificar qué archivos o directorios serán los que podrá exportar el servidor NFS y también se indican las opciones de exportación. Si no existe el archivo, hay que crearlo de la siguiente manera:

```
$ vi /etc/exports
```

Para nuestra práctica, este archivo debe contener la siguiente línea:

```
/home192.168.1.0/255.255.255.0(rw,no_root_squash)
```

donde:

`/home` → indica el directorio que se exportará.
`192.168.1.0/255.255.255.0` → indica las máquinas que podrán importar el directorio (Identificador de red/mascara de red).
`rw` → indica que el sistema de archivos se montará con permisos de lectura y escritura.
`no_root_squash` → permite que usuarios *root* remotos tengan todos los privilegios.

Una vez modificado el archivo *exports*, hay que levantar el servicio NFS. Linux cuenta con un script que nos permite realizar esto de la siguiente manera:

```
$ /etc/rc.d/init.d/nfs start
Inicio de los servicios NFS:                               [  OK  ]
Starting NFS quotas:                                       [  OK  ]
Inicialización del demonio NFS:                            [  OK  ]
Inicialización de NFS mountd:                              [  OK  ]
```

NFS utiliza el servicio *portmap* el cual permite llevar a acabo los procedimientos necesarios para dirigir las peticiones entre servidores y clientes. Hay que reiniciar este servicio de la siguiente manera:

```
$/etc/rc.d/init.d/portmap restart
Parando portmapper:                                        [  OK  ]
Iniciando portmapper:                                      [  OK  ]
```

En cada una de las máquinas clientes, hay que montar el sistema de archivos que se importará. En este caso se montará el sistema de archivos del servidor. Hay que modificar el archivo *fstab* que se encuentra dentro del directorio */etc*. Este archivo es leído cuando arranca el sistema y cualquier sistema de archivos NFS listado será montado. Para este ejemplo, se debe agregar la siguiente línea:

```
192.168.1.1:/home     /home     nfs     defaults 0 0
```

donde:

`192.168.1.1:/home` → indica el servidor NFS que exportará el sistema de archivos y el directorio exportado.
`/home` → indica el directorio local donde se montará el sistema de archivos.
`nfs` → especifica el tipo de sistema de archivos que será montado.
`Defaults 0 0` → especifica cómo el sistema de archivos es montado.

Una vez que se ha declarado el directorio a importar, hay que levantar el servicio NFS. Linux cuenta con un script que nos permite realizar esto de la siguiente manera:

```
$ /etc/rc.d/init.d/nfs start
```

Al igual que en el servidor, debe reiniciarse el servicio *portmap*:

```
$/etc/rc.d/init.d/portmap restart
```

Una vez modificado el archivo *fstab*, hay que montar el sistema de archivos del servidor. Linux cuenta con el comando *mount* que nos permite hacer esto de la siguiente manera:

```
$ mount –t nfs 192.168.1.1:/home /home
```

donde:

`nfs` → tipo de sistema de archivos que será montado.
`192.168.1.1:/home` → dirección IP del servidor NFS y el directorio a importar.
`/home` → directorio donde se montará el sistema de archivos.

Para ver si se montó el sistema de archivos, basta con ejecutar el comando *mount*:

```
$ mount
/dev/hda2 on / type ext3 (rw)
none on /proc type proc (rw)
usbdevfs on /proc/bus/usb type usbdevfs (rw)
none on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda7 on /home type ext3 (rw)
none on /dev/shm type tmpfs (rw)
/dev/hda9 on /tmp type ext3 (rw)
/dev/hda6 on /usr type ext3 (rw)
/dev/hda3 on /usr/local type ext3 (rw)
/dev/hda5 on /var type ext3 (rw)
192.168.1.1:/home on /home type nfs (rw,addr=192.168.1.1)
```

Una vez que está montado el sistema de archivos, hay que levantar el servicio NIS. El servicio NIS (*Network Information Service*) distribuye información de un servidor entre las máquinas que conforman una red. Dichos datos consisten generalmente de información de usuarios y procesos.

En todas las máquinas, hay que ver si se tienen instalados los paquetes *ypserv*, *ypbind y yptools*. Para ver si estos paquetes se encuentran instalados, hay que ejecutar el siguiente comando:

```
$ rpm –qa | grep yp
yp-tools-2.7-5
freetype-2.1.3-6
XFree86-truetype-fonts-4.3.0-2
ypbind-1.11-4
ypserv-2.6-2
```

el cual deberá desplegar una lista con los paquetes instalados que contengan *yp*.

Si no se encuentran instalados, el CD de Linux cuenta con un directorio donde se encuentran todos los paquetes RPM. Sólo basta con montar el CD, cambiarse a la carpeta de RPMS y ejecutar los siguientes comandos:

```
$ rpm –ivh ypserv-2.6-2.rpm
$ rpm –ivh ypbind-1.11-4.rpm
$ rpm –ivh yptools-2.7-5.rpm
```

Una vez que se encuentran instalados, hay que definir un nombre de dominio bajo el cual se identificarán todas las máquinas que van a tener acceso a los archivos que va a compartir el servidor maestro. Es a través de este dominio como los clientes NIS van a buscar un servidor para solicitarle los archivos de autenticación que estarán en un formato especial; y es por esta razón que tanto clientes como el servidor deben pertenecer al mismo dominio.

Para definir el nombre del dominio de manera permanente, dentro del archivo *network* en el directorio */etc/sysconfig* hay que agregar la siguiente línea:

```
NISDOMAIN=Beowulf
```

Editar el archivo *yp.conf* dentro del directorio */etc*, archivo de configuración de *ypserv*. Agregar las siguientes líneas:

```
ypserver 192.168.1.1
domain Beowulf broadcast
```

lo cual indica la dirección IP del servidor NIS y el nombre del dominio al que pertenecen.

En el servidor, hay que ejecutar el comando *make* que se encuentra en el directorio */var/yp*. Con esto se creará un directorio con el nombre del dominio NIS en el que se encontrarán los archivos que consultarán los clientes NIS:

```
$ cd /var/yp/
```

```
$ make
gmake[1]: Cambiando a directorio `/var/yp/(none)'
Updating passwd.byname...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
passwd.byuid...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
group.byname...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
group.bygid...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
hosts.byname...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
hosts.byaddr...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
rpc.Byname...
```

```
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
rpc.bynumber...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
services.byname...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
services.byservicename...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
netid.byname...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
protocols.bynumber...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
protocols.byname...
failed to send 'clear' to local ypserv: RPC: Program not registeredUpdating
mail.aliases...
failed to send 'clear' to local ypserv: RPC: Program not
registeredgmake[1]: Sal
iendo directorio `/var/yp/(none)'
```

En el servidor, hay que iniciar *ypserv* de la siguiente manera:

```
$ /etc/rc.d/init.d/ypserv start
Configuración del nombre de dominio NIS Beowulf:           [  OK  ]
Iniciando los servicios del servidor YP:                   [  OK  ]
```

NIS ocupa *portmap*, por lo que se debe reiniciar este programa:

```
$/etc/rc.d/init.d/portmap restart
Parando portmapper:                                        [  OK  ]
Iniciando portmapper:                                      [  OK  ]
```

Para configurar al sistema como servidor NIS, en el servidor, hay que ejecutar el siguiente comando y agregar el nombre del servidor de la siguiente manera:

```
$ /usr/lib/yp/ypinit –m
At this point, we have to construct a list of the hosts which will run NIS
servers.  miservidor is in the list of NIS server hosts.  Please continue
to add the names for the other hosts, one per line.  When you are done with
the list, type a <control D>.
     next host to add:  miservidor
     next host to add:

   The current list of NIS servers looks like this:

   miservidor

   Is this correct?  [y/n: y]  y
   We need a few minutes to build the databases...
   Building /var/yp/Beowulf/ypservers...
```

```
   Running /var/yp/Makefile...
   gmake[1]: Cambiando a directorio `/var/yp/Beowulf'
   Updating passwd.byname...
   Updating passwd.byuid...
   Updating group.byname...
   Updating group.bygid...
   Updating hosts.byname...
   Updating hosts.byaddr...
   Updating rpc.byname...
   Updating rpc.bynumber...
   Updating services.byname...
   Updating services.byservicename...
   Updating netid.byname...
   Updating protocols.bynumber...
   Updating protocols.byname...
   Updating mail.aliases...
   gmake[1]: Saliendo directorio `/var/yp/Beowulf'

   miservidor has been set up as a NIS master server.

Now you can run ypinit -s miservidor on all slave server.
```

En las máquinas cliente, hay que levantar *ypbind* de la siguiente manera:

```
$ /etc/rc.d/init.d/ypbind start
Configuración del nombre de dominio NIS Beowulf:           [  OK  ]
Enlazando al dominio NIS:                                  [  OK  ]
Escuchando por un servidor de dominio NIS.
```

Para que el servicio se levante cada vez que se inicia el sistema, en las máquinas cliente, hay que crear las siguientes ligas:

```
$ ln –s /etc/rc.d/init.d/ypbind /etc/rc.d/rc3.d/S27ypbind
$ ln –s /etc/rc.d/init.d/nfs /etc/rc.d/rc3.d/S28nfs
$ ln –s /etc/rc.d/init.d/ypbind /etc/rc.d/rc5.d/S27ypbind
$ ln –s /etc/rc.d/init.d/nfs /etc/rc.d/rc5.d/S28nfs
```

Para el servidor, hay que crear las siguientes ligas:

```
$ ln –s /etc/rc.d/init.d/ypbind /etc/rc.d/rc3.d/S27ypserv
$ ln –s /etc/rc.d/init.d/nfs /etc/rc.d/rc3.d/S28nfs
$ ln –s /etc/rc.d/init.d/ypbind /etc/rc.d/rc5.d/S27ypserv
$ ln –s /etc/rc.d/init.d/nfs /etc/rc.d/rc5.d/S28nfs
```

En las máquinas cliente, hay que editar el archivo *nsswitch.conf* dentro del directorio */etc* y agregar *nis* a las líneas: *passwd, shadow* y *group*. El archivo debe quedar de la siguiente manera:

```
passwd:   nis files
shadow:   nis file
group:    nis file
```

para indicar que estos archivos serán autocompletados por medio de NIS.

En el servidor, agregar un nuevo usuario al sistema con *password*, por medio del cual se conectará con las máquinas de la siguiente manera:

```
$ adduser usuario1
```

```
$ passwd usuario1
Changing password for user usuario1.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

Para que el usuario sea actualizado, hay que ejecutar los siguientes comandos:

```
$ cd /var/yp
```

```
$ make
gmake[1]: Cambiando a directorio `/var/yp/Beowulf'
Updating passwd.byname...
Updating passwd.byuid...
Updating group.byname...
Updating group.bygid...
Updating netid.byname...
gmake[1]: Saliendo directorio `/var/yp/Beowulf'
```

Importante: cada vez que se agregue un nuevo usuario al sistema, debe ejecutase el comando *make* para actualizar los archivos de autenticación de los usuarios.

Para que las máquinas que conforman el cluster puedan comunicarse entre ellas, hay que tener habilitado el servicio *rsh* que permite conexiones remotas de forma insegura.

Para todas las máquinas, primero hay que ver que el paquete se encuentre instalado:

```
$ rpm –qa | grep rsh
rsh-0.17-14
rsh-server-0.17-14
```

Si no se encuentra instalado, hay que montar el CD de Linux, cambiarse a la carpeta de RPMS y ejecutar el siguiente comando:

```
$ rpm –ivh rsh-0.17-14.rpm
$ rpm –ivh rsh-server-0.17-14.rpm
```

Para habilitar este servicio, hay que editar los archivos *rsh, rlogin y rexec* que se encuentran dentro del directorio */etc/xinetd* y cambiar el valor de la variable *disable* a *no* de la siguiente manera:

```
disable=no
```

Después, hay que reiniciar el servicio *xinetd* de la siguiente manera:

```
$ /etc/rc.d/init.d/xinetd restart
Parando xinetd:                                            [  OK  ]
Iniciando xinetd:                                          [  OK  ]
```

Para conectarse a alguna de las máquinas, hay que ejecutar el siguiente comando:

```
$  rsh -l usuario1 192.168.1.4
```

para entrar al cliente número 3.

En el servidor, crear el archivo *.rhosts* dentro del *home* del usuario que se conectará a las máquinas de la siguiente manera:

```
$ vi /home/usuario1/.rhosts
```

Este archivo debe contener la dirección IP del servidor y el nombre del usuario de la siguiente manera:

```
192.168.1.1 usuario1
```

Crear en todas las máquinas el archivo *hosts.equiv* dentro del directorio */etc* y agregar las direcciones IP de todas las máquinas que trabajan con PVM:

```
$ vi /etc/hosts.equiv

192.168.1.1
192.168.1.2
192.168.1.3
192.168.1.4
```

Finalmente, conectarse a una terminal y entrar como el usuario que se conectará a todas las máquinas, una vez dentro, realizar un *rsh* a alguna de las máquinas de la siguiente manera:

```
$ su – usuario1
$ rsh 192.168.1.4
```

PRÁCTICA # 2
CONSTRUCCIÓN DE UN CLUSTER TIPO BEOWULF
(PARTE II)

OBJETIVO

El alumno conocerá la instalación y configuración de bibliotecas paralelas utilizando software libre (PVM y MPI) en sistemas operativos Linux.

Al final de esta práctica el alumno podrá:
Trabajar en ambientes paralelos/distribuidos utilizando una herramienta de programación paralela.

CUESTIONARIO PREVIO

6. ¿Qué herramientas de programación paralela existen?
7. ¿Qué es PVM?
8. ¿Qué es MPI?
9. ¿Qué ventajas tienen estas bibliotecas?

Requerimientos:

- Tener instalado un compilador: `gcc,gcc-c++,gcc-g77.`
- Tener instalado y funcionando el servicio *rsh*.

EJERCICIOS PROPUESTOS

1. Instalación y configuración de PVM.

   En <u>todas las máquinas</u>, primero hay que instalar PVM. Revisar que el paquete de PVM esté instalado:

   ```
   $ rpm –qa | grep pvm
   ```

   Si no se encuentra instalado, se debe montar el CD de Linux, cambiarse al directorio RPMS y ejecutar el siguiente comando:

   ```
   $ rpm -ivh pvm-3.4.4-12.i386.rpm
   warning: pvm-3.4.4-12.i386.rpm: V3 DSA signature: NOKEY, key ID
   db42a60e
   Preparing...      ########################################### [100%]
      1:pvm          ########################################### [100%]
   ```

   Para saber el directorio en donde se ha instalado PVM, ejecutar el comando *whereis*, el cual nos permitirá conocer la ruta de instalación PVM de la siguiente manera:

```
$ whereis pvm
pvm:    /usr/share/pvm3
```

Una vez instalado y sabiendo el directorio de instalación, en el servidor, modificar el archivo *.bashrc* dentro del directorio *home* del usuario que se conectará a todas las máquinas (usuario1) y agregar las variables de trabajo de PVM. Para nuestra práctica, este archivo debe contener las siguientes líneas:

```
export PVM_ROOT=/usr/share/pvm3
export PVM_ARCH=LINUX
```

donde:

`export PVM_ROOT=/usr/share/pvm3` → indica el directorio de instalación de PVM.
`export PVM_ARCH=LINUX` → indica el tipo de arquitectura.

Para el servidor, crear el archivo *skel.bashrc* dentro del directorio */etc* y agregar las mismas líneas:

```
export PVM_ROOT=/usr/share/pvm3
export PVM_ARCH=LINUX
```

En todas las máquinas, crear la siguiente liga:

```
$ ln –s /usr/share/pvm3/lib/LINUXI386
      /usr/share/pvm3/lib/LINUX
```

En una terminal, entrar como el usuario que se conecta a todas las máquinas (usuario1). Entrar a la consola de PVM y agregar las máquinas que trabajarán bajo la máquina virtual. Para esto, sólo hay que ejecutar el comando *pvm* desde el *home* del usuario de la siguiente manera:

```
$ su – usuario1
$ pwd
  /home/usuario1
$ pvm
      pvm>
```

Una vez que se entró a la consola, el comando *add* nos permitirá agregar las máquinas que trabajarán dentro de la máquina virtual. Agregar los nombres de las máquinas clientes de la siguiente manera:

```
pvm> add cliente1
add cliente1
1 successful
                    HOST     DTID
                cliente1  180000

pvm> add cliente2
add cliente2
1 successful
                    HOST     DTID
                Cliente2  180001

pvm> add cliente3
add cliente3
1 successful
                    HOST     DTID
                Cliente3  180002

pvm> add cliente4
add cliente4
1 successful
                    HOST     DTID
                cliente4 180003
pvm> quit
```

Para compilar un programa paralelo escrito en PVM, hay que entrar como el usuario que se creó para conectarse con todas las máquinas (usuario1) y ejecutar el siguiente comando:

```
$ su – usuario1
$ pwd
   /home/usuario1

$ cc prog_fuente –o prog_exe –I /usr/share/pvm3/include –L
/usr/share/pvm3/lib/LINUX –lgpvm3 -pvm3 –lm
```

donde:

`prog_fuente` → es el nombre del archivo del programa fuente.
`-o prog_exe` → opción que nos permite crear el archivo ejecutable, `prog_exe` es el nombre del archivo ejecutable que será creado.
`–I /usr/share/pvm3/include` → opción que nos permite indicar dónde se encuentran las bibliotecas de PVM.

`–L /usr/share/pvm3/lib/LINUX` → opción que nos permite indicar dónde se ligarán las bibliotecas de PVM.
`-lgpvm3` → biblioteca de grupos.
`-lpvm3` → biblioteca básica para C.

Para correr el programa solo hay que ejecutar el siguiente comando:

```
$ ./prog_exe –t 10
```

donde:

`prog_exe` → es el archivo ejecutable creado en la compilación
`-t 10` → parámetro que nos permite especificar el número de procesos que serán atendidos por el cluster.

2. Instalación y configuración de MPI.

El paquete de MPI puede bajarse de la siguiente dirección:

http://www-unix.mcs.anl.gov/mpi/mpich/

Para instalar MPI, hay que descomprimir y desempaquetar el paquete en todas las máquinas de la siguiente manera:

```
$ tar –xvzf mpchi-xxx.tar.gz
```

Al desempaquetar, se creará una carpeta con el mismo nombre *mpich-xxx*. Entrar a esa carpeta y ejecutar el script *configure* de la siguiente manera:

```
$ cd mpich-xxx
$ ./configure –-prefix=/usr/local/mpi –-with-arch=LINUX
```

donde:

`–-prefix=/usr/local/mpi` → indica la ruta donde se instalará MPI.
`–-with-arch=LINUX` → indica la arquitectura de la máquina.

Para seguir con la compilación de MPI, ejecutar los siguientes comandos:

```
$ make
$ make install
```

Para configurarlo, editar el archivo *profile* que se ubica en el directorio */etc*, y agregar la siguiente línea:

```
export PATH=$PATH:/usr/local/mpi/bin:/usr/local/mpi/include
```

Editar el archivo *machines.LINUX* para listar los nombres de las máquinas que conforman el cluster. Este archivo se encuentra dentro del directorio */usr/local/mpi/share*. Para nuestra práctica este archivo debe contener:

```
$ vi /usr/local/mpi/share/machines.LINUX

miservidor
cliente1
cliente2
cliente3
cliente4
```

En una terminal, entrar como el usuario que se conectará a todas las máquinas (usuario1) y agregar al PATH las rutas de los archivo binarios y las bibliotecas de MPI de la siguiente manera:

```
$ su – usuario1
$ pwd
   /home/usuario1

$ export PATH=$PATH:/usr/local/mpi/bin:/usr/local/mpi/include
```

Para comprobar que todas las máquinas que trabajarán con MPI se comunican, existe un script que nos permite realizar un test a todas las máquinas. El script se llama *tstmachines* y se encuentra en el directorio */usr/local/mpi/sbin/*. Para realizar el test, hay que logearse como el usuario que se conectará a todas las máquinas (usuario1) y desde su *home* ejecutar el script de la siguiente manera:

```
$ su - usuario1

$ pwd
   /home/usuario1

$ /usr/local/mpi/sbin/tstmachines -v
Trying true on miservidor ...
Trying true on cliente1 ...
Trying true on cliente2 ...
```

```
Trying true on cliente3 ...
Trying true on cliente4 ...
Trying ls on miservidor ...
Trying ls on cliente1 ...
Trying ls on cliente2 ...
Trying ls on cliente3 ...
Trying ls on cliente4 ...
Trying user program on miservidor ...
Trying user program on cliente1 ...
Trying user program on cliente2 ...
Trying user program on cliente3 ...
Trying user program on cliente4 ...
```

Si el test no marca ningún error, MPI está listo para correr programas en todas las máquinas dadas de alta en el archivo *machines.LINUX*.

Para compilar un programa paralelo escrito en MPI, hay que entrar como el usuario que se conecta a todas las máquinas (usuario1) y desde su *home* ejecutar el siguiente comando:

```
$ su – usuario1
$ pwd
   /home/usuario1

$ mpicc –o hola hola.c
```

donde:

`mpicc` → es el compilador de MPI
`-o hola` → es el nombre del archivo ejecutable que se generará al momento de la compilación.
`hola.c` → es el nombre del archivo fuente.

Para correr el nuevo archivo ejecutable, hay que teclear el siguiente comando:

```
$ mpirun –np 10 hola
```

donde:

`mpirun` → es el comando que nos permite ejecutar el programa.
`-np 10` → es el número de procesos que serán atendidos por el cluster.
`hola` → es el archivo ejecutable.

PRÁCTICA # 3
PARADIGMAS DE LA PROGRAMACIÓN PARALELA

OBJETIVO

El alumno conocerá y aplicará diferentes paradigmas de la programación paralela (SMPD, Maestro-Esclavo y Entubamiento de Datos).

Al final de esta práctica el alumno podrá:
Aplicar un paradigma de la programación paralela para paralelizar un problema (SMPD, Maestro-Esclavo y Entubamiento de Datos), teniendo en cuenta las ventajas y desventajas de cada uno de ellos.

CUESTIONARIO PREVIO

1. ¿En qué consiste el paradigma SMPD?

2. Describa las funciones que realizan cada uno de los procesos que participan en el paradigma Maestro-Esclavo.

3. ¿En qué consiste el paradigma de Entubamiento de Datos?

4. Menciona las ventajas y desventajas del paradigma de Entubamiento de Datos.

EJERCICIOS PROPUESTOS

1. Escriba el pseudocódigo de la suma de *n* datos utilizando el paradigma SMPD, describiendo cada una de las funciones que realizan los procesos que intervienen en el paradigma.

2. Escriba el pseudocódigo de la suma de *n* datos utilizando el paradigma Maestro-Esclavo, describiendo cada una de las funciones que realizan los procesos que intervienen en el paradigma.

3. ¿Qué diferencias existen entre los paradigmas SMPD y Maestro-Esclavo?

4. Mencione las ventajas y desventajas de cada paradigma.

PRÁCTICA # 4
PARALELIZACIÓN DE UN PROGRAMA

OBJETIVO

El alumno conocerá y aplicará las etapas de la programación paralela .

Al final de esta práctica el alumno podrá:
Aplicar cada una de las etapas de la programación paralela para paralelizar un problema secuencial.

CUESTIONARIO PREVIO

1. ¿Cuáles son las etapas en la creación de programas paralelos?

2. Describa cada una de las etapas.

EJERCICIOS PROPUESTOS

1. Identificar y describir cada una de las etapas en la creación de programas paralelos aplicado al producto de 2 matrices rectangulares de acuerdo con el siguiente modelo secuencial:

$A\Re \in nxm, B\Re \in mxq, X\Re \in nxq$

$X = A \cdot B$

```
PARA i=0 HASTA n-1
     PARA j=0 HASTA m-1
          x[i, j] = 0
          PARA k=0 HASTA q-1
               x[i, j] = x[i, j] + a[i, k] * b[k, j]
          FINPARA
     FINPARA
FINPARA
```

2. Escribir el pseudocódigo del modelo secuencial anterior utilizando un paradigma de la programación paralela.

3. Mencione cuáles son los puntos a considerar para paralelizar o no un programa.

PRÁCTICA # 5
PROGRAMACIÓN EN MPI

OBJETIVO

El alumno conocerá y aplicará las funciones básicas de una biblioteca de programación paralela (MPI).

Al final de esta práctica el alumno podrá:
Aplicar las funciones básicas de MPI para comunicar procesos en la paralelización de un problema.

CUESTIONARIO PREVIO

1. ¿Qué es MPI?
2. Describe cada una de las siguientes funciones:
   `MPI_Init()`
   `MPI_COMM_WORLD()`
   `MPI_Comm_size()`
   `MPI_Comm_rank()`
   `MPI_Reduce()`
   `MPI_Finalice()`

EJERCICIOS PROPUESTOS

1. De acuerdo con lo investigado anteriormente, describa lo que realiza el siguiente código paralelo “hola_mundo.c”:

```
#include <stdio.h>
#include "mpi.h"

#define MAX 255
main(int argc, char **argv)
{
      char car;
      int node,size;
      int tam = MAX;
      char name[MAX];

      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD, &size );
      MPI_Comm_rank(MPI_COMM_WORLD, &node);
      MPI_Get_processor_name(name, &tam);

      printf("Hola Mundo del proceso %d de %d procesos (%s)\n",node,
size,name);
      MPI_Finalize();
}
```

2. ¿Qué paradigma representa en código anterior?

3. Ejecuta el programa “hola_mundo.c” y escribe su salida.

PRÁCTICA # 6
PROGRAMACIÓN EN MPI:
CÁLCULO DEL NÚMERO PI.

OBJETIVO

El alumno conocerá y aplicará las funciones básicas de una biblioteca de programación paralela (MPI).

Al final de esta práctica el alumno podrá:
Aplicar las funciones básicas de MPI para comunicar procesos en la paralelización de un problema.

CUESTIONARIO PREVIO

1. ¿Qué ventajas tiene la librería de MPI?
2. Describe cada una de las siguientes funciones:
   - `MPI_Get_processor_name()`
   - `MPI_MAX_PROCESSOR_NAME`
   - `MPI_Bcast()`
   - `MPI_Wtime()`

EJERCICIOS PROPUESTOS

1. De acuerdo con lo investigado anteriormente, describa lo que realiza el siguiente código paralelo “pi.c”:

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

double f( double );
double f( double a )
{
    return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
```

```
        fprintf(stderr,"Process %d on %s\n", myid, processor_name);
        n = 0;
        while (!done)
        {
            if (myid == 0)
            {
/*
              printf("Enter the number of intervals: (0 quits) ");
              scanf("%d",&n);
*/
               if (n==0)
                     n=100;
               else n=0;

              startwtime = MPI_Wtime();
            }
            MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
            if (n == 0)
                done = 1;
            else
            {
                h   = 1.0 / (double) n;
                sum = 0.0;
                for (i = myid + 1; i <= n; i += numprocs)
                {
                    x = h * ((double)i - 0.5);
                    sum += f(x);
                }
                mypi = h * sum;

                 MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
                 0,MPI_COMM_WORLD);

                if (myid == 0)
                {
                    printf("pi is approximately %.16f, Error is %.16f\n",
                           pi, fabs(pi - PI25DT));
                     endwtime = MPI_Wtime();
                     printf("wall clock time = %f\n", endwtime-startwtime);
                }
            }
        }
        MPI_Finalize();

        return 0;
    }
```

2. ¿Qué paradigma representa en código anterior?

3. Ejecuta el programa “pi.c” y escribe su salida.

PRÁCTICA # 7
GRID DE COMPUTADORAS

OBJETIVO

El alumno investigará una aplicación de los sistemas paralelos/distribuidos (GRID de computadoras).

Al final de esta práctica el alumno:
Conocerá una de las aplicaciones del cómputo paralelo.

EJERCICIOS PROPUESTOS

1. ¿Qué es una GRID?
2. ¿Cuál es la configuración de un GRID?
3. ¿Cuáles son las ventajas de una GRID de computadoras?
4. ¿Qué es Globus?
5. Menciona algunas de sus aplicaciones.

PRÁCTICA # 8
APLICACIÓN DE LOS CLUSTERS: RENDERIZACIÓN DE IMÁGENES

OBJETIVO

El alumno investigará una aplicación de los sistemas paralelos/distribuidos dentro del diseño asistido por computadora.

Al final de esta práctica el alumno:
Conocerá uno de los campos de aplicación del cómputo paralelo.
Sabrá la importancia del cómputo paralelo.

EJERCICIOS PROPUESTOS

1. ¿Qué es renderización de imágenes?
2. ¿A qué se refiere el término de granjas de renderización?
3. ¿Qué software utiliza?
4. Menciona algún proyecto en el que se haya utilizado un cluster para la renderización de imágenes.
5. ¿Cuáles son las ventajas de utilizar sistemas paralelos/distribuidos en la renderización de imágenes?

BIBLIOGRAFÍA

1) Balderas Jiménez, Roy, Moreno Sánchez, Jeannette y Placido Mojica, Martín. **Administración del cluster tipo Beowulf de la Facultad de Ingeniería**. UNAM. Facultad de Ingeniería. 2003.

2) González Flores, Christian, González Flores, Lissette y Ramírez Ponce, Erik Rogelio. **Elaboración de prácticas de programación distribuida en el cluster tipo Beowulf de la Facultad de Ingeniería.** UNAM. Facultad de Ingeniería. 2004.

3) Juárez Sosa, Julio Cesar, Saenz García, Elba Karen y Valdez Casillas, Oscar Rene. **Análisis del desempeño de un cluster Beowulf en diversos algoritmos de tipo concurrente**. UNAM. Facultad de Ingeniería. 2001.

4) Manrique Martínez, Daniel. **Construcción y evaluación de desempeño de un cluster tipo Beowulf para cómputo de alto rendimiento.** UNAM. Facultad de Ingeniería. 2001.

REFERENCIAS ELECTRÓNICAS

5) The Beowulf Project. http://www.beowulf.org

6) Argonne National Laboratory. “MPICH-A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/

7) García Leiva, Rafael A. Universidad Autónoma de Madrid. Instalación y Configuración de un cluster de alta disponibilidad bajo Linux. http://es.tldp.org/Manuales-LuCAS/doc-instalacion-cluster-alta-disponibilidad/instalacion-cluster-alta-disponibilidad/node1.html

8) Vega Luna, José Ignacio, Sánchez González, Roberto y Salgado Guzmán, Gerardo. Universidad Autónoma Metropolitana. Arquitectura RISC Vs CISC. http://www.azc.uam.mx/publicaciones/enlinea2/num1/1-2.htm

9) Laboratorio de Sistemas Distribuidos y Redes de Computadoras. Programa de Ciencias de la Computación. Universidad de Sonora. http://clusters.fisica.uson.mx/clusters_de_Linux.htm

10) Encyclopedia dedicated to computer technology. http://www.webopedia.com/TERM/FLOPS.html

11) Araujo Cárdenas, Alfonso. Sistemas Distribuidos.
http://mx.geocities.com/alfonsoaraujocardenas/sistemasdistribuidos.html

12) Clusters para todos.
http://www.windowstimag.com/atrasados/1997/12_sept97/revista/Art12.htm

13) Plata, Oscar. Universidad de Málaga. Arquitecturas distribuidas.
http://www.ac.uma.es/educacion/cursos/informatica/ArqDist/pdfs/01-Modelos_Arq_Escal.pdf

14) Enciclopedia libre Wikipedia. Computación concurrente.
http://es.wikipedia.org/wiki/Programaci%C3%B3n_concurrente

15) Perales Fabian, Víctor. Universidad Señor de Sipan. Arquitecturas paralelas.
http://www.elrinconcito.com/articulos/ArquitecturaParalela/ArquitecturaParalela.htm

16) Enciclopedia libre Wikipedia. Algoritmos paralelos.
http://es.wikipedia.org/wiki/Algoritmos_paralelos

17) Enciclopedia libre Wikipedia. Programación paralela.
http://es.wikipedia.org/wiki/Programaci%C3%B3n_paralela

18) Sistemas Operativos. Administración de procesos.
http://www.tau.org.ar/base/lara.pue.udlap.mx/sistoper/capitulo5.html

19) Los algoritmos distribuidos.
http://webdia.cem.itesm.mx/ac/rogomez/interAlgosDist.html

20) Domínguez Aguilar, Guillermo. Sistemas distribuidos.
http://ccbas.uaa.mx/~guido/portal/files/portfolio/SORyM/Sistemas_Distribuidos.PDF

21) Enciclopedia libre Wikipedia. RPC.
http://es.wikipedia.org/wiki/RPC

22) Martínez, David. Comunicación en los sistemas distribuidos.
http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/SO8.htm#Intro

23) Clusters de Linux. Universidad de sonara.
http://www.fisica.uson.mx/carlos/LinuxClusters/

24) Pruebas de eficiencia: Test de linpack.
http://www.csi.map.es/csi/silice/Ctciabs5.html

25) Introducción a la computadora. Tipos de computadora.
http://coqui.lce.org/cedu6320/ssegarra/tipocomp.html

26) Instituto tecnológico de la paz. Tutorial de sistemas distribuidos.
http://www.itlp.edu.mx/publica/tutoriales/sistsdist1/

27) Jiménez Peris, Ricardo y Patiño Martínez, Marta. Distributed Systems Laboratory. Universidad Politécnica de Madrid. Introducción a los sistemas distribuidos.
http://lsd.ls.fi.upm.es/lsd/asignatura-distribuidos/intro-color.pdf

28) Tutorial y descripción técnica de TCP/IP.
http://ditec.um.es/laso/docs/tut-tcpip/3376fm.html

29) Grid Computing.
http://www.iquiqueweb.com/disco/publicos/trabajo%20grid%20computing.doc

30) Martín Llorente, Ignacio. Universidad Complutense Madrid. Niveles de Paralelismo dentro de un programa.
http://asds.dacya.ucm.es/nacho/pp_archivos/Niveles%20Paralelismo.pdf

31) Programación paralela. Lenguajes y modelos de programación paralela.
http://servinf.dif.um.es/~domingo/apuntes/ProgPar/trasparencias/Lenguajes.ppt#339,1,%20PROGRAMACIÓN%20PARALELA%20%20Tema%202:%20Lenguajes%20y%20modelos%20de%20programación%20paralela

32) García Leiva, Rafael A. Universidad Autónoma de Madrid. Paralelismo.
http://es.tldp.org/almacen/Manuales-LuCAS/doc-manual-openMosix-1.0/doc-manual-openMosix_html-1.0/node9_ct.html

33) Giménez Cánovas, Domingo. Programación paralela.
http://dis.um.es/~domingo/progpar.html

34) Departamento de sistemas informáticos y computación. Programación paralela bajo en entorno MPI.
http://www-copa.dsic.upv.es/docencia/iblanque/lcp/Boletines/Boletin2_2004.pdf

35) Departamento de sistemas informáticos y computación. Entorno de programación PVM.
http://www-copa.dsic.upv.es/docencia/iblanque/lcp/Boletines/Boletin1_2004.pdf

36) García Carballeira, Félix. Arquitectura de computadores.
http://arcos.inf.uc3m.es/~ii_ac2/descripcion.shtml

37) Redes para clusters.
http://icaro.eii.us.es/descargas/Sistemas%20Multiprocesadores%202004-2005%20parte%203.pdf

38) Departamento de lenguajes y sistemas informáticos. Universidad de Granada. Programación distribuida y paralela.
http://www-lsi.ugr.es/~jmantas/pdp/pdp1.pdf

39) Pino Morillas, Óscar, Arroyo Moreno, Roberto Francisco y Nievas Muñoz, Francisco Javier. Los clusters como plataforma de procesamiento. http://64.233.179.104/search?q=cache:h2XZw9WE_5kJ:usuarios.lycos.es/lacaraoculta/descargas/Clusters_definitivo.pdf+lenguajes+y+entornos+de+programaci%C3%B3n+paralela&hl=es

40) Departamento de sistemas informáticos y computación. Lenguajes y entornos de la programación paralela: el entorno PVM. http://alumnat.upv.es/pla/visfit/2606/AAAGNXAAXAAAD5MACS/casi1.doc

41) Alonso, José Miguel. Programación de aplicaciones paralelas con MPI. http://www.sc.ehu.es/acwmialj/edumat/mpi.pdf

42) Introducción a PVM http://www.ii.uam.es/%7Ecortiz/docencia/Paralelo/prac0.html#Introduccion

43) Descripción de los sistemas distribuidos. http://www.cenidet.edu.mx/subaca/web-dcc/web-sd/LabSisDis/DescripEspSD.html

44) El planificador de Linux. http://www.espaciolinux.com/artitecid-9.html

45) Samldone, Javier. Dentro del planificador de Linux. http://72.14.203.104/search?q=cache:ZbvkMroiyb0J:www.smaldone.com.ar/gnulinux/docs/linux2.6_es.pdf+balanceo+de+cargas+y+planificaci%C3%B3n+de+procesos&hl=es

46) Rafael A. García Leiva. Universidad Autónoma de Madrid. ¿Qué es realmente openMOXIS? http://es.tldp.org/almacen/Manuales-LuCAS/doc-manual-openMosix-1.0/doc-manual-openMosix_html-1.0/node21_ct.html

47) Enciclopedia libre Wikipedia. Cluster de balanceo de carga. http://es.wikipedia.org/wiki/Cluster_de_balanceo_de_carga

48) Plata, Oscar. Universidad de Málaga. Clusters arquitecturas distribuidas. http://www.ac.uma.es/educacion/cursos/informatica/ArqDist/pdfs/04-ClustersBW.pdf

49) Universidad Internacional del Ecuador. Procesamiento Paralelo. http://www.internacional.edu.ec/academica/informatica/creatividad/uide-bits/uide-bits-05-2003.pdf