30

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

ANÁLISIS DEL DESEMPEÑO DE UN CLUSTER BEOWULF EN DIVERSOS ALGORITMOS DE TIPO CONCURRENTE

TESIS PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

P R E S E N T A N

JUÁREZ SOSA JULIO CESAR
SÁENZ GARCÍA ELBA KARÉN
VALDEZ CASILLAS OSCAR RENÉ

DIRECTOR DE TESIS M en I. LAURA SANDOVAL MONTAÑO

CIUDAD UNIVERSITARIA MÉXICO. D. F.. 2001

A Dios

Porque siempre ha estado conmigo y me ha abierto las puertas de mis ilusiones y metas; me ha concedido en todos los aspectos la salud y sabiduría para seguir siempre adelante. Porque me ha permitido llevarlo a él y a Jesús siempre en mi corazón.

A mis padres

Cristina García García

Porque siempre me ha brindado su amor, apoyo, comprensión y muchas otras cosas. Que me han ayudado a tener confianza y seguir siempre adelante. Agradezco todos sus esfuerzos por conseguir que mis hermanos y yo seamos felices y cumplamos nuestras metas.

Jorge Roberto Sáenz Vázquez

Porque siempre ha sido mi admiración y un ejemplo a seguir. Le agradezco el quererme, darme su cariño y su apoyo.

A los dos que han disfrutado mis logros y triunfos les dedico este trabajo con todo mi amor.

A mi hermanita

América

Que siempre me ha apoyado y se ha preocupado por mí, y a quien quiero muchisimo.

A mi hermanito

Jorge Raymundo

Que quiero mucho y que deseo pronto cumpla también esta meta.

A los dos les agradezco el quererme y cuidarme.

A Oscar

Porque juntos hemos compartido y disfrutado momentos tristes y alegres durante nuestra estancia en la facultad y juntos hemos realizado este trabajo. Por apoyarme, comprenderme y por ese cariño tan especial que nos tenemos.

A mis amigos

A todos ellos que me consideraron su amiga, y que desinteresadamente me brindaron su amistad y apoyo.

A todas aquellas personas

Que durante mi vida me han querido y se han preocupado por mí.

Elba Karén Sáenz García

Para lograr llegar hasta este punto en que ahora estoy, muchas personas me han ayudado.
Tal vez no pueda mencionarlas a todas, pero quiero agradecer principalmente a:

A Dios.

A mi mamá y mi papá, por permitirme ser su hijo y seguir su ejemplo de trabajo y dedicación.

A mamá Margarita, por el amor que siempre nos ha brindado.

A Juan y Diego, mis hermanos queridos, de los cuales también he aprendido.

Agradecer a Laura, por apoyarnos en este proyecto.

Agradecerles también a Karén y Julio por permitirme trabajar junto con ellos.

A Karén, por su cariño.

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

Oscar.

Agradecimientos::

Al hombre que durante mucho tiempo se preocupo por mí y que hoy ya no se encuentra entre nosotros, pero que siempre estará en mi mente y alma, porque gracias a él soy parte de esta vida. A mi padre dedico este trabajo.

A mis padres, Rosalinda Sosa de Juárez y Adelfo Juárez Pérez que por mucho tiempo se ocuparon de mis estudios y gracias a ellos hoy cumplo un objetivo más.

A mis hermanos, por todo el apoyo que me han dado.

A mis compañeros de tesis, Oscar y Karen, que sin ellos no hubiese podido realizar el presente trabajo.

A UNICA (Unidad de Computo Académico), por todas y cada una de las facilidades otorgadas y en especial a la Ing. Beatriz Barrera Hernández e Ing. Enrique Barranco Vite

Julio César

ÍNDICE

INTRODUCCIÓN

Actualmente existen diversidad de aplicaciones como el análisis del movimiento de la tierra, pronostico del tiempo, procesamiento de imágenes entre otras que necesitan de cómputo paralelo, y por lo tanto vemos la necesidad de una máquina paralela, debido a que si dichas aplicaciones son ejecutadas en maquinas secuenciales tardan días o varios meses en obtener un resultado, y en una máquina paralela podríamos reducir ese tiempo.

Existen máquinas con varios procesadores integrados pero su costo es muy elevado, es por ello que como día con día las estaciones de trabajo y computadoras personales tienen mayor poder de cómputo y las redes de comunicación mayor ancho de banda, se vuelve cada vez más atractivo y factible realizar cómputo paralelo en redes de computadoras con la ayuda de software que nos permita la paralelización de los procesos. De hecho, en muchas ocasiones es la única forma de realizar cómputo paralelo ya que como se menciono las máquinas paralelas tienen un alto costo.

En nuestro caso, para poder tener una máquina paralela de bajo costo, se recurrió a equipo que se consideraba de desecho, adecuándolo para su correcto funcionamiento dentro de una red también de bajo costo.

En cuanto a la programación, ya no es llevada a cabo de la misma forma que se hacia tradicionalmente. La paralelización se lleva a cabo haciendo una fragmentación del problema y dividiéndolo manualmente en diversas computadoras por lo que es necesario contar con una máquina paralela para agilizar dicho proceso.

Este trabajo tiene un enfoque con fines didácticos. Es por ello que el objetivo de este trabajo es dar una referencia para la construcción de un tipo de máquina paralela, en este caso un Cluster Beowulf sin discos, este tipo de máquina nos permite utilizar todos y cada uno de los equipos con los que se cuenta. Así como conocer los algoritmos, paradigmas básicos de programación paralela y cómo programarlos a través de PVM y C, que servirán de apoyo a quien pretenda construir y realizar una aplicación paralela.

A continuación se presenta una breve descripción de cada uno de los capítulos que conforman este trabajo de investigación.

El Capítulo I da una descripción de la evolución de la computación pasando de la era secuencial a la paralela, se introducen conceptos de procesamiento paralelo así como algunos ejemplos que nos permitirán comprender los conceptos de paralelización, continuando con la clasificación clásica de las máquinas paralelas propuesta por Flynn, así como el de las máquinas paralelas contemporáneas, en donde se introduce el concepto de Cluster, para después profundizar un poco más en él. Por lo que entenderemos como Cluster el sistema paralelo de procesamiento distribuido, el cual consiste en una

colección de computadoras interconectadas para trabajar en conjunto como una sola, integrando sus recursos de cómputo.

Existen diferentes tipos de cluster, y su clasificación depende del sistema operativo instalado, arquitectura de computadoras utilizada e interconexión de red utilizada por las mismas, cada una de estas clasificaciones se explican en detalle. Por lo que es importante conocer el tipo de hardware a utilizar para su construcción y las diferentes plataformas bajo las que pueden trabajar los diferentes sistemas operativos. Dentro de las clasificaciones de Cluster encontramos al Cluster Beowulf, donde deberá tomar gran atención ya que forma la base del presente trabajo de investigación, por lo que se explica en el último apartado de este capítulo.

El Capítulo II fue destinado al sistema operativo Linux, iniciando con una breve historia del Sistema Operativo, para seguir con una descripción de sus características y su funcionamiento. Esto nos permitirá entender porqué la elección del Sistema Operativo Linux, como base para la creación de un cluster de tipo Beowulf dedicado.

Como cualquier sistema operativo, Linux tiene un núcleo o Kernel el cual permite la comunicación entre las aplicaciones y el hardware. Éste se puede configurar de acuerdo a los requerimientos del sistema. En este capítulo se muestran las tres diferentes formas de configurar el Kernel (línea de comandos, menú texto y entorno gráfico), así como las opciones que encontraremos al momento de configurarlo y una breve explicación de éstas.

Es importante conocer cómo ha de funcionar el Kernel ya que de él dependerá el funcionamiento de los nodos.

El Capítulo III se ocupa de la construcción del Cluster Beowulf sin discos. Se da una visión completa de los componentes ocupados, instalación, comandos utilizados para su configuración y la administración del mismo, entre los que se encuentra Mini-Netboot, Etherboot y RSH (Remote Shell). Así como se presentan todos y cada uno de los archivos de configuración ocupados. Adicionalmente se presentan las herramientas utilizadas para iniciar una sesión en cada uno de los nodos, recordando que sólo ocuparemos tiempo de procesador, es decir, sólo utilizaremos el CPU.

El Capítulo IV, comienza con la descripción del cómputo paralelo; conceptos como concurrencia y paralelismo son utilizados para comprender de forma más clara el desarrollo de algoritmos paralelos, al mismo tiempo se abordan los diversos paradigmas de programación que nos permitieron entender cómo se pueden desarrollar diferentes programas y colocarlos en una máquina paralela. Los programas aquí expuestos explican tan solo algunos de los paradigmas de programación ya que el tratar de abarcarlos constituiría otro trabajo de investigación. Existe también un apartado para PVM (Parallel Virtual Machine), que como herramienta de paralelización es indispensable mostrar cómo se configura y se utiliza junto con el lenguaje de programación C, en el

desarrollo de cada uno de los programas que se presentan. Además se explican las sentencias básicas con las que trabaja y las más importantes para el programador. Al finalizar la sección presentamos algunos programas realizados con PVM y C, así como el análisis de su desempeño.

Finalmente en el Capítulo V, presentamos gráficas de desempeño del Cluster en los diferentes algoritmos utilizados, así como el análisis correspondiente de cada uno.

Capítulo I

Antecedentes

1. ANTECEDENTES

1.1. ERAS DE LA COMPUTACIÓN

La industria de la computación es una de las que está creciendo más rápidamente y esto se debe al rápido desarrollo tecnológico en las áreas del cómputo en hardware y software. Los avances tecnológicos en hardware incluyen desarrollo y fabricación en tecnologías de chips, microprocesadores rápidos y baratos, así como también un alto ancho de banda y baja latencia en interconexión de redes. Además de esto, los recientes avances en tecnologías VLSI (*Very Large Scale Integration*) han jugado un buen papel en el desarrollo de poderosas computadoras secuenciales y paralelas. En lo que se refiere al software también se está desarrollando rápidamente. Software maduro como sistemas operativos, lenguajes de programación, metodologías de desarrollo y herramientas ya están ahora disponibles. Esto ha posibilitado el desarrollo y aparición de múltiples aplicaciones en diversas ciencias de ingeniería y en necesidades comerciales.

Debe hacerse notar que grandes aplicaciones como pronósticos del tiempo y análisis del movimiento de la tierra han llegado a ser una de las razones del desarrollo de poderosas computadoras paralelas.

Una manera de ver a la computación es en dos importantes eras de desarrollo:

- Era de la computación secuencial
- Era de la computación paralela.

Cada era de la computación empieza con la etapa de desarrollo en arquitectura de hardware, seguidas por sistemas de software (particularmente en el área de compiladores y sistemas operativos) y aplicaciones que han alcanzando su cenit con el aumento en PSE'S (*Problem Solving Enviroments* - Ambiente de solución de problemas)

Cada una de las etapas anteriores pasa por tres fases

1. Investigación y desarrollo
2. Comercialización
3. Mercancías ("artículos de tienda a bajo costo")

Lo anterior se representa en la figura 1.1

La tecnología y el desarrollo de componentes en las etapas de la era secuencial han madurado y desarrollos similares todavía están sucediendo en

la era paralela. Esto es, la tecnología de computación paralela necesita desarrollarse ya que todavía no llega a su auge.

La principal razón para crear y usar computadoras paralelas es porque el paralelismo es una de las mejores maneras de superar las velocidades y los cuellos de botella que suelen suceder con un solo procesador. Además se puede decir que el precio a razón del desempeño de un *Cluster* basado en cómputo paralelo en comparación a una microcomputadora es mucho muy pequeño y consecuentemente sería menor su costo. Un corto desarrollo y producción de sistemas de moderada velocidad usando arquitecturas paralelas es más barato que el equivalente desarrollo en un sistema secuencial.

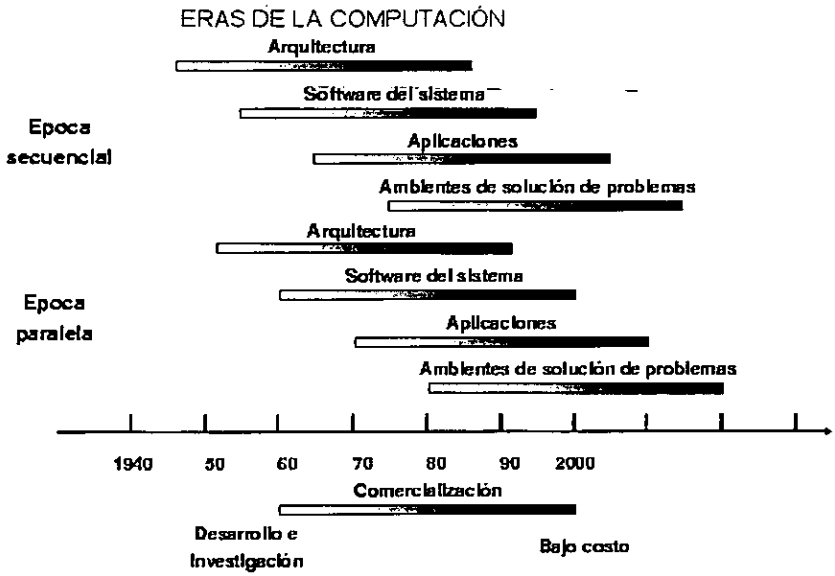


Figura 1. 1 Cuadro comparativo entre las diferentes eras de la computación.

1.1.1. EVOLUCIÓN

El cómputo paralelo es un problema central e importante en muchas aplicaciones de cómputo intensivo, tales como procesamiento de imágenes y robótica entre otras.

Dado que es un problema, el cómputo paralelo es el proceso de dividir el problema en varios subproblemas, resolviendo los subproblemas simultáneamente, y combinando la solución de los subproblemas para dar una solución al problema original.

Cuando la arquitectura de la computadora fue propuesta por Von Neumann, él visualizaba la máquina como una calculadora rápida. Él sugiere 5 unidades de sistema para la computadora.

1. Unidad de entrada
2. Unidad de Salida
3. Memoria
4. Unidad Aritmética – Lógica (ALU) y
5. Unidad de Control

La arquitectura es representada por un diagrama de bloques en la figura 1.2

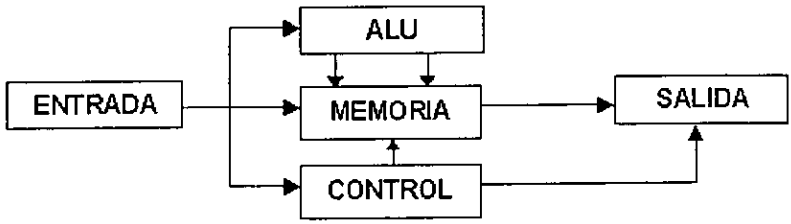


Figura 1. 2 Diagrama de bloques de la arquitectura Von Neumann

La unidad de control, es la que dirige el sistema, las líneas de control se muestran en líneas punteadas, de manera diferente a las líneas de datos, en líneas continuas. La unidad de entrada toma cada uno de los trabajos y los envía a la computadora. Entre los dispositivos utilizados por la unidad de entrada se encuentran las tarjetas perforadas, cintas de papel perforado, cintas magnéticas, discos magnéticos o flexibles, y teclados, siendo éstos los más utilizados en la historia de la computación. La unidad de salida toma los trabajos procesados y los envía hacia su salida. Impresoras y plotters son algunos de los más populares dispositivos de salida. La unidad aritmética y

lógica se encargan de procesar cada uno de los trabajos, mientras que la memoria almacena cada uno de los trabajos temporalmente, hasta que se les da salida.

La primera generación (1945-1955) : Las primeras computadoras empleaban válvulas y su programación se realizaba directamente mediante conexiones o lenguaje máquina. No existían lenguajes de programación, ni sistemas operativos.

La segunda generación (1955-1965) : Las computadoras se construían empleando transistores. Los programas se escribían en un lenguaje de programación (Ensamblador o Fortran) para después pasarlo a tarjetas perforadas. Se desarrollaron los primeros sistemas de procesamiento por lotes, que consistían en agrupar una serie de trabajos en un lote y pasárselos a la computadora para su resolución de forma secuencial, uno tras otro. Las grandes computadoras de esta época se dedicaban al cálculo científico y de ingeniería. Los sistemas operativos más habituales eran FMS (Fortran Monitor System) e-IBSYS, el sistema operativo de IBM para la 7094 (la computadora más conocida de la época)

La tercera generación (1965-1978) : La aparición de los circuitos integrados a pequeña escala favoreció la producción de máquinas más potentes, de dimensiones más reducidas, y de menor precio. Se desarrollaron familias de computadoras, como el sistema 360 de IBM, con el fin de lograr la compatibilidad entre las distintas máquinas de la familia, y poder emplear los mismos programas, incluso el sistema operativo, en toda la familia. Esto originó un sistema operativo monstruoso, el OS/360, que contenía miles de errores, y requería continuas revisiones que arreglaban una serie de errores, introduciendo muchos otros. Se generalizó la técnica de multiprogramación, que consistía en dividir la memoria en varias partes, con un trabajo distinto en cada una. Mientras un trabajo esperaba por E/S, otro podía emplear la CPU, lo cual mejoraba la utilización de la máquina. Para aislar cada una de las particiones se empleaba un hardware especial. Se inventó la técnica de spooling (Simultaneous Peripheral Operation On Line, Operación simultánea y en línea de periféricos), que se empleó para las entradas y salidas. Sin embargo, aunque los sistemas operativos de esta época eran adecuados para los cálculos científicos y el procesamiento de datos comerciales, seguían siendo esencialmente sistemas de procesamiento por lotes. A pesar de ello comenzaron los primeros desarrollos de sistemas de tiempo compartido, variante de la multiprogramación, como el sistema CTSS del MIT. Un diseño posterior de sistema de estas características fue MULTICS (MULTiplexed Information and Computing Service) cuyo objetivo era proporcionar una máquina para cientos de usuarios empleando tiempo compartido. MULTICS fue un sistema que fracasó y se abandonó. Durante esta época, se desarrolla la familia de mini computadoras DEC PDP-1 hasta PDP-11. Ken Thompson

escribió una versión especial de MULTICS para un usuario en una PDP-7, que más tarde daría origen a UNIX.

La cuarta generación (1978-1991) : Se desarrollaron los circuitos integrados del tipo LSI (*Large Scale Integration - Alta escala de integración*) y VLSI (*Very Large Scale Integration - Muy alta escala de integración*) que contenían miles y hasta millones de transistores en un centímetro cuadrado, lo que abarató los costos de producción. Se inventó de esta forma el microprocesador, que permitía a un usuario tener su propia computadora personal. Los sistemas operativos desarrollados han ido siendo cada vez más amigables, proporcionando entornos gráficos de trabajo. Los sistemas operativos más difundidos en estos años han sido MS-DOS de Microsoft funcionando en los chips de Intel, y UNIX funcionando en todo tipo de máquinas (especialmente con chips RISC). A mediados de los 80, aparecen las primeras arquitecturas con computadoras en paralelo que emplean memoria compartida o distribuida, y hardware vectorial, así como los primeros sistemas operativos de multiproceso, lenguajes y compiladores especiales para estos sistemas.

La quinta generación (1990-) : La tecnología del proceso paralelo comienza a madurar y se inicia la producción de máquinas comerciales. Los sistemas MPP (Massively Paralell Processing) son los sistemas paralelos en los que se basa la investigación. El desarrollo progresivo de las redes de computadoras, y la idea de la interconexión e intercambio de recursos, ha promovido el desarrollo de sistemas operativos de red y sistemas operativos distribuidos. En los primeros, el usuario es consciente de la existencia de otras máquinas a las que puede conectarse de forma remota. Cada máquina ejecuta su propio sistema operativo local y tiene su propio grupo de usuarios. En cambio, en un sistema operativo distribuido, el usuario percibe el sistema como un todo, sin conocer el número de máquinas, ni su localización, de esta forma los usuarios no son conscientes del lugar donde se ejecuta su programa, ni dónde se encuentran sus archivos físicamente; todo ello es manejado de forma automática por el sistema operativo. Los sistemas operativos de red no tienen grandes diferencias respecto a los sistemas operativos de un solo procesador, ya que sólo necesitan una serie de características adicionales que no modifican la estructura esencial del sistema operativo. En cambio, los sistemas operativos distribuidos necesitan una serie de modificaciones de mayor importancia, ya que esencialmente se diferencian de los sistemas centralizados en la posibilidad de cómputo en varios procesadores a la vez.

Un equipo de científicos japoneses pensó que para satisfacer las necesidades de la futura sociedad en el campo de la inteligencia artificial, las posibilidades de las computadoras con la presente arquitectura podrían ser inadecuadas. En 1979, el gobierno japonés nombra un comité encabezado por Tohru Moto Oka, imaginó las necesidades de la sociedad en los 90s. El comité se separó en tres subcomités. Los 10 primeros miembros del comité, encabezados por Hajime Karatsu, estudian el tipo de computadora necesario para el futuro. Los 12

miembros del segundo subcomité, encabezados por Hideo Aiso, estudian la arquitectura de la computadora y sus necesidades. Los siguientes 13 miembros del subcomité, encabezados por Hazuhiro Fuchi, trabajaron con el concepto de la computadora del futuro. Estos tres subcomités enviaron sus propuestas, y Chairman Oka formuló la propuesta final de la computadora del futuro, que se conoce como la quinta generación de computadoras. El gobierno japonés entendió dicha implicación y organizó la primera conferencia de la quinta generación de computadoras en octubre de 1981. En la conferencia, la quinta generación de computadoras propuestas fue discutida en detalle por científicos de todo el mundo quienes aceptaron la propuesta. Cuando la quinta generación de computadoras ingresó al mercado, las presentes computadoras no cayeron en desuso ya que para resolver los problemas de las áreas más comunes de la computación siguen siendo las más utilizadas. Sin embargo, éstas no pueden ser usadas para manejar grandes cantidades de información como son el Procesamiento Natural del lenguaje, reconocimiento de imágenes y patrones y otras aplicaciones de inteligencia artificial. La arquitectura de las computadoras de quinta generación es diferente de la presente arquitectura de Von Neumman. El gobierno japonés ha establecido el instituto para la nueva generación de computadoras (New Generation Computer Technology (ICOT)), para la construcción de la quinta generación de computadoras.

Los datos y las instrucciones son almacenados en la memoria de la computadora. Cuando un proceso tiene lugar, los datos son recuperados de la memoria.

Por ejemplo, considera un médico en la atención a un paciente. Como se muestra en la figura 1.3 el médico reúne los síntomas (datos) del paciente. El médico conoce todas las enfermedades y sus síntomas. Todo está listo en memoria. Ahora el médico compara los síntomas del paciente con los datos registrados en memoria y diagnostica la enfermedad.

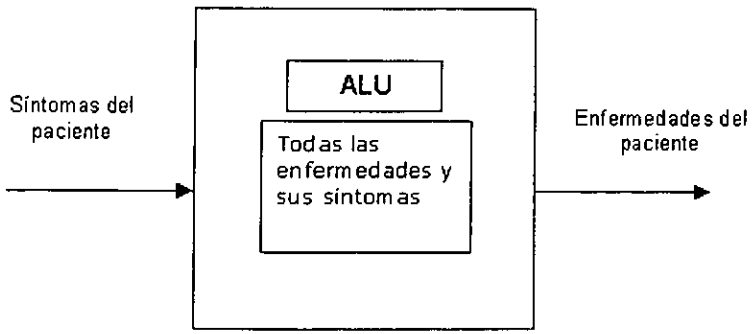


Figura 1. 3 Esquema de diagnostico de enfermedades de un paciente.

En la ausencia de un médico, ¿puede ser usada una computadora? En otras palabras, ¿puede una computadora remplazar un médico? Científicos han tomado esto como un reto. Este tipo de aplicaciones necesita de un largo procesamiento de trabajo hecho en fracciones de segundo. Estos problemas pertenecen al área de Inteligencia Artificial. Sistemas computacionales que resuelven problemas de inteligencia artificial son llamados sistemas expertos. Las siguientes son algunas de las áreas en el campo de la inteligencia artificial.

- Procesamiento Natural del Lenguaje (NLP)
- Procesamiento y Reconocimiento de Imágenes.
- Reconocimiento de Patrones.
- Reconocimiento de caracteres.
- Reconocimiento de la voz
- Pronóstico del tiempo
- Diagnósticos Médicos y
- Máquinas Inteligentes

1.1.2. COMPUTADORAS PARALELAS

En la sección previa se ha visto que las computadoras secuenciales actuales no son tiene la capacidad necesaria para el procesamiento de algunas aplicaciones. En esta sección introduciremos el concepto de computadoras paralelas, las cuales serán muy útiles para realizar el procesamiento que las máquinas secuenciales no pueden realizar o les toma demasiado tiempo en terminar. Un modelo simple de computadora se da en la figura 1.4.

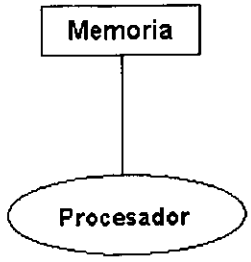


Figura 1. 4 Modelo simple de computadora.

Las operaciones de entrada y/o salida no son consideradas aquí, por simplicidad. El procesador tiene que acceder a memoria. Los valores almacenados en la memoria son leídos por el procesador, y después procesados por el mismo, los resultados son almacenados nuevamente en la memoria. A causa de las grandes invenciones en la tecnología VLSI, tenemos

un gran rendimiento en los procesadores, con mas de 10^6 compuertas, disponibles hasta el momento. Éstos tienen líneas con un ancho menor a los 0.5 micrones (1 microns = 10^{-6} metros), y tienen una densidad de almacenamiento de 1000 KB por centímetro. Los procesadores más recientemente incorporados al mercado alcanzan ya las velocidades de 1 GHz a 1.5 GHz. Con estos chips, alcanzamos llamadas con un alto grado de respuesta, pero científicos dudan en el tiempo de respuesta del dispositivo al ser construido. Desafortunadamente, estos veloces procesadores no pueden satisfacer las necesidades de la computación en áreas como preedición del tiempo, oceanografía, astrofísica, aerodinámica, procesamiento de imágenes y sensaciones remotas, las cuales se han mencionado en la sección anterior.

En una computadora ordinaria sólo se tiene un procesador. Al tratar de alcanzar una mayor velocidad, se podrían tener más procesadores en una computadora. Tales computadoras son llamadas computadoras paralelas. Computadoras con un procesador son llamadas computadoras secuenciales. El paralelismo puede ser explicado usando la aplicación de reconocimiento de caracteres; ver figura 1.5

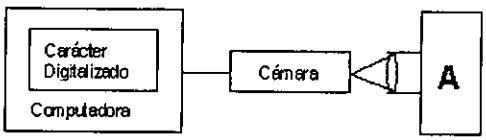


Figura 1. 5 Computadora de un solo procesador.

El alfabeto es escrito en diferentes estilos y por diferentes personas, la gente tiene la capacidad de reconocerlos. Una computadora puede entender y determinar que tipo de carácter es a través de métodos probabilísticos.

Cuando se usa una computadora secuencial, las probabilidades P(A), P(B), ..., P(n); son calculadas una antes de la otra, por un procesador. En su lugar, podemos usar varios procesadores para calcular cada una de estas probabilidades simultáneamente. Esto se ilustra en la figura.1.6.

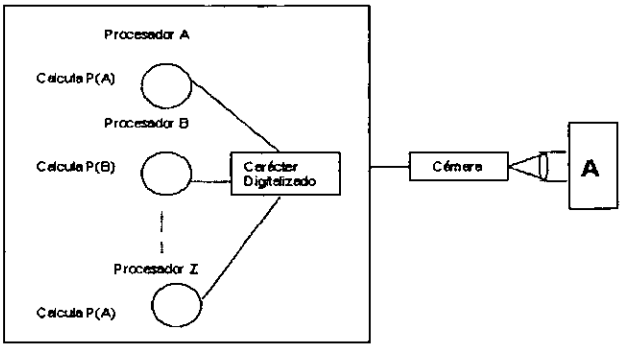


Figura 1. 6 Computadora de más de un procesador.

1.1.3. CONCEPTOS DE PROCESAMIENTO PARALELO

Al resolver un problema usando cómputo paralelo, uno descompone el problema en pequeños subproblemas, los cuales pueden ser resueltos en paralelo. Entonces estos resultados son combinados de manera eficiente para dar el resultado final del problema principal. No es tan fácil descomponer cada problema grande en subproblemas. Porque de los datos que se emplean dependen unos de otros, los procesadores deben comunicarse entre sí. El punto importante aquí es el tiempo (frecuencia) que toman para comunicarse, usualmente el tiempo de comunicación entre dos procesadores, es mayor cuando lo comparamos con el tiempo de procesamiento. Dado este factor, el esquema de comunicación deberá estar muy bien planeado, para dar un buen algoritmo paralelo.

La descomposición y la dependencia de datos son ilustrados por un simple ejemplo de la vida real. Consideremos un contratista (en construcción) quien tiene mano de obra y otros recursos para llevar a cabo la construcción de una casa en un periodo de 5 meses.

Asumiremos que el contratista no puede llevar la construcción de más que una casa simultáneamente. Si al contratista le es asignada la construcción de 100 casas por una junta de viviendas, éste tomara 500 meses para completar el trabajo. La figura 1.7 ilustra el proceso para cuatro casas solamente.

Casa 4				Casa 4	
Casa 3			Casa 3		
Casa 2		Casa 2			
Casa 1	Casa 1				
Tiempo en meses	0 - 5	5 - 10	10 - 15	15 - 20	

Figura 1. 7 Proceso de construcción de cuatro casas solamente.

Si la junta de viviendas busca mayor velocidad en el proceso, ésta pudiese emplear 100 diferentes constructoras y llevar la construcción de las 100 casas simultáneamente. En este caso, la construcción de todas las 100 casas será completada en 5 meses. Si 100 contratistas no están disponibles y solo hay 10 disponibles, el trabajo será completado en 50 meses. La figura 1.8 ilustra la terminación de 6 casas por dos contratistas en 15 meses.

Casa 6			Contratista 2	
Casa 5			Contratista 1	
Casa 4		Contratista 2		
Casa 3		Contratista 1		
Casa 2	Contratista 2			
Casa 1	Contratista 1			
Tiempo en meses	0 - 5	5 - 10	10 - 15	15 - 20

Figura 1. 8 Proceso de construcción con dos contratistas.

Éste es un ejemplo en el cual el trabajo es descompuesto fácilmente. Aquí, cada casa es independiente una de la otra, y entonces el trabajo principal es descompuesto fácilmente.

Ahora consideremos el trabajo de construcción de una casa. Dividiremos el trabajo en 5 pasos diferentes.

1. Construcción de cimientos.
2. Construcción de la estructura.
3. Completando la carpintería, tal como ventanas, puertas, etc.
4. Instalando el sistema eléctrico y
5. Pintado y acabados.

Nótese que todos estos pasos no pueden ser llevados simultáneamente, ya que cada paso iniciará sólo si el anterior paso es completado. Diríamos que éstos son inherentemente secuenciales. En este caso, dividiendo el trabajo en subgrupos es una dificultad. Sin embargo, si el contratista tiene 5 diferentes grupos de gente para llevar estos 5 pasos, él o ella pueden completar su trabajo más eficientemente. Asumiremos que cada grupo hace un paso en un mes. Cuando el primer paso es terminado para la primera casa, el segundo grupo iniciará el segundo paso de la primera casa. En el mismo tiempo, el primer grupo iniciará el trabajo del paso 1 en la segunda casa. Como se muestra en la figura.1.9

Casa 1 Lista
Casa 2 Lista
Casa 3 Lista

Paso 5					Casa 1	Casa 2	Casa 3
Paso 4				Casa 1	Casa 2	Casa 3	Casa 4
Paso 3			Casa 1	Casa 2	Casa 3	Casa 4	Casa 5
Paso 2		Casa 1	Casa 2	Casa 3	Casa 4	Casa 5	
Paso 1	Casa 1	Casa 2	Casa 3	Casa 4	Casa 5		
Tiempo en meses	1	2	3	4	5	6	7

Figura 1. 9 Construcción de las casas con cinco grupos de trabajo diferentes.

Este procedimiento de paralelización se le conoce como *pipelining* (entubamiento). En el ejemplo anterior de *pipelining*, durante el quinto mes todos los grupos están trabajando en diferentes casas.

Grupo 1 esta haciendo el paso 1 para la casa 5.
Grupo 2 esta haciendo el paso 2 para la casa 4.
Grupo 3 esta haciendo el paso 3 para la casa 3.
Grupo 4 esta haciendo el paso 4 para la casa 2.
Grupo 5 esta haciendo el paso 5 para la casa 1.

Al término del quinto mes, la primera casa está terminada. A partir del4 quinto mes una casa es completada en cada subsiguiente mes. En este caso, el contratista es capaz de entregar una casa completada al término de cada mes.

Veamos un ejemplo más para ilustrar la descomposición de un problema y la interdependencia de subproblemas. Consideremos una compañía en cual 100 máquinas de algún tipo deben ser instaladas. Si hay un grupo de ingenieros y ellos pueden instalar solamente una máquina a la vez, entonces el trabajo puede ser completado en 100 unidades de tiempo. Si estos son 100 diferentes grupos de ingenieros quienes pueden trabajar simultáneamente, el trabajo será hecho en una unidad de tiempo. Si cada grupo contiene tres subgrupos para hacer el trabajo apropiado (ingenieros mecánicos), suministrando instalación eléctrica (ingenieros eléctricos) y probando (controladores de procesos), entonces el trabajo puede ser paralelizado utilizando la técnica de *pipelined*.

En la presente discusión hemos visto el concepto de *pipelining*. Otro concepto importante en el procesamiento paralelo es el multiprocesamiento. La palabra multiprocesamiento se refiere a procesos simultáneos de más de una tarea por diferente procesador. Esto en general se divide en dos subcategorías.

- Multiprocesos
- Multicomputadoras

En un sistema multiprocesos, muchos procesadores trabajan simultáneamente y comparten una memoria en común. En las multicomputadoras, se tiene un grupo de procesadores, en los cuales cada uno tiene suficiente conocimiento de la memoria local. La comunicación entre procesadores es a través de mensajes. Esto es también conocido como procesamiento distribuido. O sea, ninguna memoria en común ni siquiera un reloj común.

1.1.4. SUPERCOMPUTADORAS

Por los intensivos avances en tecnologías, el paso de cuatro y media décadas han visto abundantes diseños de hardware, los cuales han ayudado a la industria de la computación a experimentar generaciones de precedente crecimiento y desarrollo, físicamente marcadas por un rápido cambio en la construcción de computadoras desde los relevadores y tubos de vacío (1940s - 1950s) hasta la presente ultra escala de circuitos integrados (ICs)

Como resultado de estos avances tecnológicos y diseños mejorados, las computadoras han pasado por una notable metamorfosis desde los lentos uniprocesadores de los años 50 hasta nuestros días con máquinas de alto desempeño, en las cuales se incluyen supercomputadoras, cuyo pico en desempeño y velocidad son miles de ordenes de magnitud superior a las primeras computadoras. Los requerimientos de ingenieros y científicos por tener siempre computadoras más poderosas, han sido la principal fuerza que ha dado dirección al desarrollo de las computadoras digitales. La realización de

computadoras rápidas es uno de los más grandes retos. La industria de la computación responde a estos retos de manera tremenda, y el resultado es una notable evolución en las computadoras en sólo cuatro décadas.
La transformación tecnológica ha sido acelerada por el revolucionario diseño, la cual frecuentemente ocurre junto con una mejora tecnológica. Las técnicas más comunes de diseño han incorporado algunas características paralelas dentro de las modernas computadoras. Desde 1960, literalmente cientos de estructuras paralelas han sido propuestas, y muchas han sido construidas y puestas en operación en los años 70.

La Illiac IV fue puesta en operación en la NASA´s Ames Research en 1972, la primera Texas Instruments Inc. Advance Scientific Computer (TI - ASC) fue usada en Europa en 1972; la primera Control Data Corp. Star-100 fue desarrollado en Lawrence Livermore National Laboratory en 1974; la primera Cray Research Inc. Cray - 1 fue puesta en servicio en Los Álamos Nacional Laboratory en 1976.

Las máquinas mencionadas anteriormente no sólo fueron las pioneras en diseño e innovación, sino que estaban dotadas con un poder sin precedente en cómputo, pero fueron entonces las predecesoras de los más poderos sistemas computacionales que hoy existen. Por lo tanto, mientras su fama decaía, dieron pronto camino a otras generaciones de las más poderosas computadoras, que culmina hoy en día con sistemas de supercómputo. Así de esta manera la Illiac IV cesó operaciones en 1981; desde 1976, la Star - 100 fue envuelta dentro de la CDC Cyber 203 (no hubo mayor producción), y entonces también envuelta en la Cyber 205, la cual señala a CDC´s a entrar en el campo de las supercomputadoras. La Cray -1 (pipelined uní procesador) fue envuelta dentro de la Cray - 1S, quien tiene considerablemente más memoria que su antecesora la Cray - 1. Las siguientes son algunas de las más grandes supercomputadoras.

- Cray XMP4 (4 procesadores, 128 Mword supercomputadora, con un pico alrededor de los 840 MFLOPS);
- Cray - 2 (256 Mword, 4 procesadores reconfigurables, supercomputadora, con 2 GFLOPS pico); y
- Cray - 3 (16 procesadores, 2 Gword, supercomputadora, con 16 GFLOPS pico)

Entre otras supercomputadoras producidas en 1980 tenemos la Eta -10, Fujitso VP - 200, Hitachi S - 810, y la IBM 3090 400/VF. De esta manera la velocidad de los procesadores construidos entre 1960 y 1970 se tienen envueltos en las supercomputadoras de los años 80 y 90.

Otras computadoras de interés histórico, aunque su principal propuesta no fue la computación numérica, tenemos la Goodyear Corparation´s STARAN y la C.mmp system de Carnegie - Mellon University.

La Illiac – 4 tenía sólo 64 procesadores. Otras computadoras consistían de un gran número de procesadores, incluida la Denelcor HEP y la International Computers Ltd. Dta Array Processor (ICL DAP), la máquina de elementos finitos de NASA´s Langley Research Center; MIDAS de Lawrence Berkeley Laboratory; Cosmic Cube de California Institute of Tecnology, TRAC de la University of Texas, CM* de at Carnegie – Mellon University, ZMOB de la University of Maryland, Pringle de la University of Washington y Purdue University y la Massively Parallel Processor (MPP) de NASA´s Goddard Space Flight Center. Sólo unas cuantas son diseñadas primordialmente para computación numérica, mientras que otras son para propósitos de investigación.

1.2. ARQUITECTURAS PARALELAS

Uno de los problemas a los que se enfrenta este campo de la computación, es el de los acrónimos de tecnología. En un principio el cómputo paralelo usó gran variedad de acrónimos que hacen referencia a algún tipo de máquina paralela. Más adelante con el desarrollo que han tenido las computadoras paralelas, las nuevas arquitecturas no siempre entran en las categorías ya definidas.

Históricamente el esquema más común de clasificación para computadoras paralelas es el ideado por Flynn[1]. Está basado en un mnemónico de cuatro letras, donde las dos primeras se relacionan con el número de instrucciones en los datos y las segundas dos letras se refieren al tipo de flujo de datos. Estas son:

SISD (Single Instruction-Single Data): Este es el modelo tradicional de la computadora de Von Newman, donde un conjunto de instrucciones se ejecutan secuencialmente sobre un flujo de datos.

SIMD (Single Instruction-Multiple Data):En esta clasificación se localizan las primeras computadoras paralelas como la Conection Machine (CM-2) , la cual fue construida por una compañía llamada Thinking Machine. En el modelo SIMD múltiple procesos individuales o un solo vector de procesos ejecutan la misma instrucción sobre diferentes conjuntos de datos.

MISD (Multiple Instruction-Single Data): Esta clasificación no es un sistema viable para la computación.

MIMD(Multiple Instruction-Multiple Data): En este modelo múltiples procesadores pueden ejecutar diferentes instrucciones en sus propios datos, y comunicarse cuando lo necesiten. Hay dos subclases de maquinas MIMD: de memoria compartida (SM) o memoria distribuida (DM).

Más adelante se da una explicación con más detalle de cada clasificación.

[1] **Industrial Strength Parallel Computing**
Alice E. Konigues
Morgan Kaufmann Publishes
2000 pag.3,4

En la figura 1.10 se muestran las diferencias básicas entre **MIMD-SM** y **MIMD-DM**.

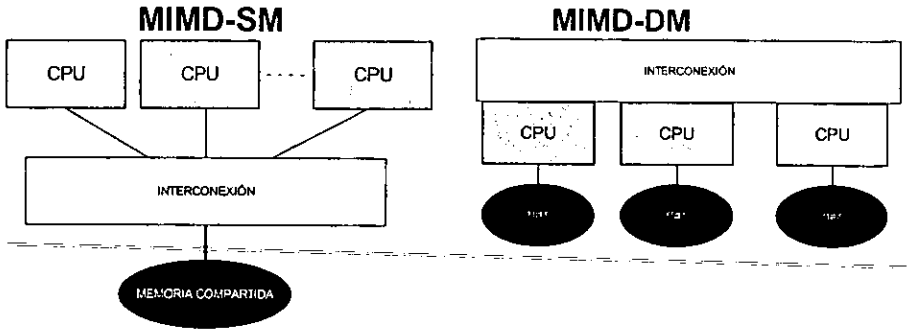


Figura 1. 10 Diferencias entre MIMD-SM y MIMD-DM.

Otro posible modelo de programación se basa en el modelo SIMD, en el cual todos los procesadores son controlados para ejecutar en *modo de paso asegurado (lock-step fashion)*. El controlador *lock-step* maneja a cada una de sus unidades de procesamiento (EP). En la figura siguiente se observa que no tienen CPUs, tienen elementos de procesamiento (EPs) la diferencia que existe con la MIMD es que en ésta cada CPU utiliza sus propias instrucciones y tiene su propio *Program Counter* que apunta a las instrucciones que van a ser utilizadas en un momento dado. Un elemento de procesamiento es alimentado por su unidad de control, la cual es la única que tiene un *Program Counter*.

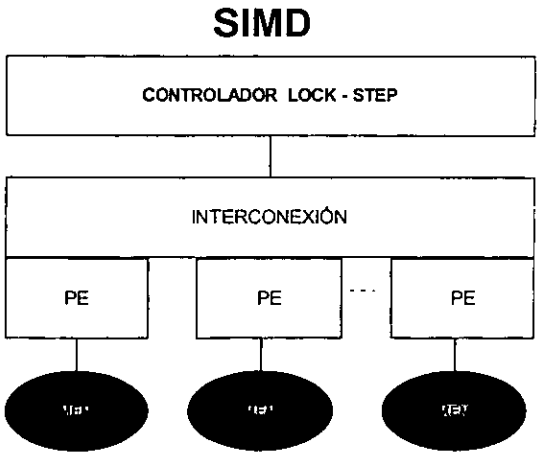


Figura 1. 11 Arquitectura de tipo SIMD.

A continuación se da con detalle la clasificación de Flynn.

1.2.1. TAXONOMÍA DE FLYNN:

Cualquier computadora, ya sea secuencial o paralela, ejecuta instrucciones sobre datos. Un flujo de instrucciones (el programa) le indica a la computadora qué hacer en cada paso, y estas instrucciones afectan a un flujo de datos (las entradas del programa). De acuerdo a la forma según la cual el conjunto de datos es afectado por el conjunto de instrucciones, pueden ser definidos diferentes modelos de computadoras, y es así como Flynn definió los siguientes cuatro modelos, dependiendo de si existen uno o múltiples flujos de instrucciones o datos, ejecutados u operados por un procesador:

a.) Un solo flujo de Instrucciones –Un solo flujo de Datos (SISD)

b.)Varios flujos de Instrucciones – Un solo flujo de Datos (MISD)

c) Un solo flujo de Instrucciones - Varios flujos de Datos (SIMD)

d.) Varios flujos de Instrucciones - Varios flujos de Datos (MIMD)

a) Computadoras SISD

Una computadora SISD consiste de una sola unidad de procesamiento recibiendo una secuencia de instrucciones que opera sobre una secuencia de datos. En cada paso, la unidad de control emite una instrucción que opera sobre datos obtenido de la unidad de memoria. Éste es el modelo Von Neumann, al cual corresponden la mayoría de los computadoras actuales, no presenta ningún paralelismo ver figura 1.12.

Figura 1. 12 Computadora SISD

b) Computadoras MISD

En este modelo p procesadores ($p>1$), cada uno con su propia unidad de control, comparten una unidad de memoria común donde residen los datos. Existen p secuencias de instrucciones y una secuencia de datos. En cada paso, un dato recibido desde la memoria es operado por todos los procesadores

simultáneamente, cada uno con las instrucciones que el procesador recibe de su unidad de control. Así el paralelismo es alcanzado realizando diferentes operaciones sobre el mismo dato, ver figura.1.13. No existen computadoras comerciales que se ajusten a este modelo.

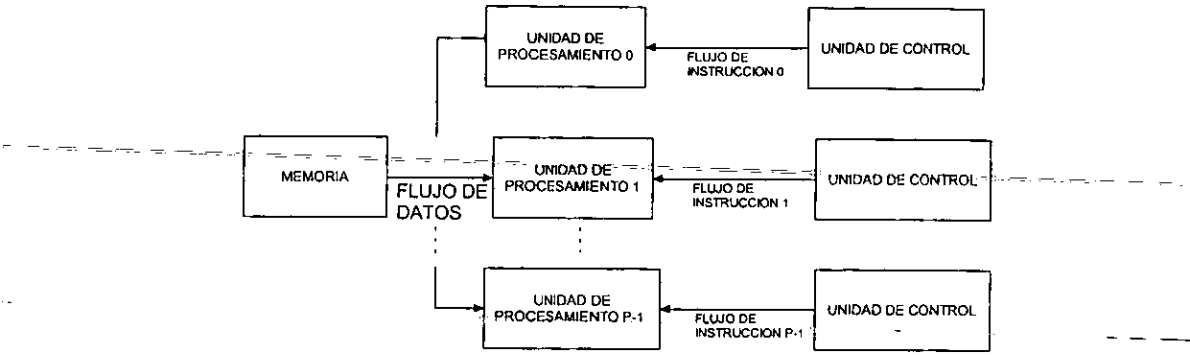


Figura 1. 13 Computadora MISD

c) Computadoras SIMD

En este modelo una computadora paralela consiste de p procesadores idénticos ($p>1$), cada uno operando con su propia memoria local. Todos los procesadores operan bajo el control de una sola secuencia de instrucciones emitida por una unidad de control central. Existen p secuencias de datos, una por procesador, los cuales operan de manera síncrona. En cada paso, todos los procesadores ejecutan la misma instrucción cada uno sobre un dato diferente.
En la mayoría de las aplicaciones de interés que se quieran resolver sobre este tipo de computadoras, es deseable que los procesadores puedan comunicarse entre sí durante el cálculo a fin de intercambiar datos o resultados intermedios. Esto puede ser realizado de dos maneras diferentes, a través de una memoria común (computadoras SIMD de memoria compartida) o a través de una red de interconexión (computadoras SIMD de memoria distribuida).

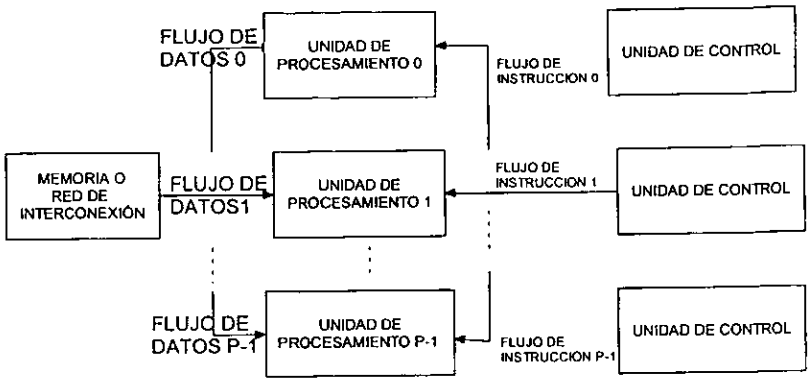


Figura 1. 14 Computadora SIMD.

d) Computadoras MIMD

Esta clase de computadoras es la más general y más poderosa en el paradigma de la computación paralela que las clasifica de acuerdo a la secuencia de datos y/o instrucciones. Este tipo de computadoras poseen p procesadores ($p>1$), cada uno operando bajo el control de una secuencia de instrucciones emitida por su propia unidad de control. Así los procesadores están potencialmente todos ejecutando diferentes programas sobre datos diferentes. Esto significa que los procesadores operan de manera asíncrona, sin embargo, los algoritmos asíncronos son difíciles de diseñar, evaluar e implantar.

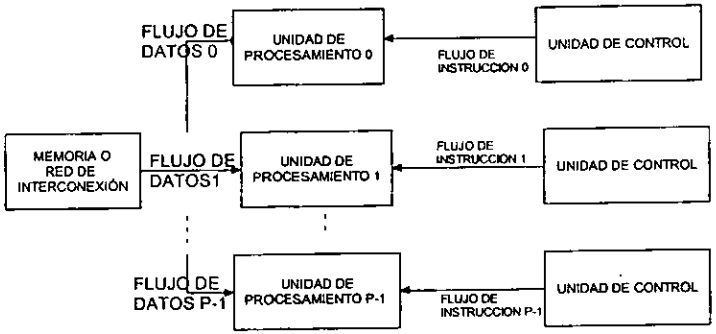


Figura 1. 15 Computadora MIMD.

Los procesadores se comunican entre sí ya sea a través de una memoria común, o por medio de una red de interconexión, y este hecho también subdivide esta clase de computadoras en otras dos, a saber:

1) Arquitecturas MIMD de Memoria Compartida

Esta clase de computadoras es también conocida como modelo de máquina paralela de acceso aleatorio (PRAM). Aquí p procesadores (p>1) comparten una memoria común. Cuando dos procesadores quieren comunicarse lo hacen a través de esta memoria común. Si se desea transmitir un dato desde el procesador *Pi* al procesador *Pj*, esto lo realiza en dos pasos, en el primero, el procesador *Pi* escribe el dato en una dirección de memoria conocida por el procesador *Pj*. En el segundo paso el procesador *Pj* lee esa localidad.

El modelo básico permite a todos los procesadores acceder la memoria compartida simultáneamente, si la posición de memoria que ellos están tratando de escribir es diferente. Sin embargo, el tipo de memoria puede dividir este modelo en cuatro subclases dependiendo de si dos o más procesadores pueden acceder a la misma posición de memoria simultáneamente, estas son:

1. *Lectura-Exclusiva, Escritura-Exclusiva*:
2. *Lectura-Concurrente, Escritura-Exclusiva*
3. *Lectura-Exclusiva, Escritura-Concurrente*
4. *Lectura-Concurrente, Escritura-Concurrente*

El permitir múltiples lecturas simultáneas sobre la misma posición de memoria no debe ocasionar ningún problema. Conceptualmente, si cada procesador requiere leer desde una misma posición de memoria copia el contenido de esa posición y lo almacena en su memoria local. Sin embargo, si varios procesadores requieren escribir simultáneamente diferentes datos sobre la misma posición de memoria, debe existir una manera determinística de especificar el contenido de esa posición de memoria una vez realizadas las escrituras. En estos casos (subclases 3 y 4), los conflictos de escritura se resuelven por hardware, mientras que para las subclases 1 y 2, es el sistema operativo que los resuelve, dando como resultado que en diferentes ejecuciones de un mismo programa pueden tener resultados distintos, dada la aleatoriedad descrita.

2) Arquitecturas MIMD de Memoria Distribuida

La otra forma de comunicación entre los procesadores es a través de una red de interconexión. En este modelo la memoria es dividida entre el conjunto de procesadores, para su acceso local. Además, cada procesador es conectado con sus vecinos a través de una línea bidireccional de comunicación, la cual le permite enviar o recibir datos en cualquier instante de tiempo, habiéndose

desarrollado una amplia variedad de topologías que permiten abarcar una gran cantidad de problemas de manera eficiente, tales como: el *arreglo lineal,* el *anillo*, la *malla*, el *toroide*, el *árbol*, el *fat-tree*, y el *hipercubo*, que tienen una baja cantidad de enlaces entre procesadores, de manera tal que cuando sea necesario comunicar un mensaje entre dos procesadores que no tienen conexión directa, debe encaminarse o enrutarse dicho mensaje por procesadores intermedios entre estos dos. Estas topologías se muestran en la siguiente figura.

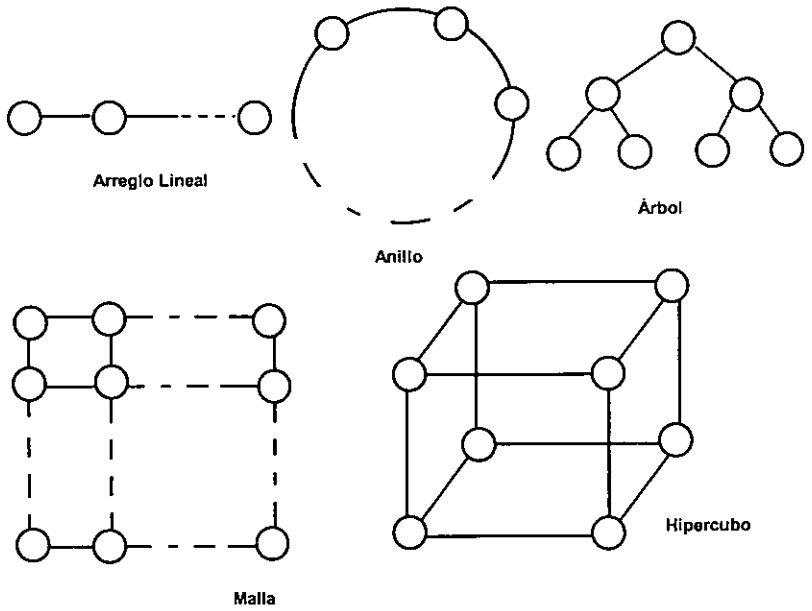


Figura 1. 16 Topologías.

Sin embargo, la clasificación de Flynn, aunque provee una clasificación muy útil, es insuficiente para clasificar otros modelos de computadoras. Por ejemplo, los procesadores vectoriales de *pipeline* merecen una inclusión dentro de las arquitecturas paralelas, debido a que ellos exhiben una substancial ejecución de aritmética concurrente y pueden manipular cientos de vectores de elementos en paralelo. Sin embargo, ellos no están situados en ninguna de las clasificaciones de Flynn, debido a la dificultad de acomodarlos dentro de la taxonomía ,como estas computadoras carecen de procesadores que ejecuten la misma instrucción que un ciclo SIMD, fallan también al tratar de poseer la autonomía de la categoría MIMD. Debido a esta deficiencia en la taxonomía de Flynn, se ha intentado extender ésta para acomodar las modernas computadoras paralelas.

1.2.2. ARQUITECTURAS PARALELAS CONTEMPORÁNEAS

Como ya se mencionó anteriormente la clasificación de Flynn no ha permitido incluir a nuevas máquinas paralelas o existen desacuerdos en qué arquitectura se les habría de incluir, por lo que han surgido gran cantidad de acrónimos (clasificaciones). En este documento no hablaremos de todas ellas sino sólo las que se consideraron son más usadas.

Las computadoras de alto desempeño hasta nuestros días se agrupan en tres principales acrónimos SMP (Symmetric Multiprocessors), MPP (Massively Parallel Processors) y NUMA(nonunyform memory access) la cual es una generalización de la SMP. Pero desde la pasada década han surgido diversos sistemas de computación de alto desempeño. Su taxonomía se basa en sus procesadores, memoria e interconexión. Los sistemas más comunes son:

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

El término SMP es referido exclusivamente a procesadores simétricos, refiriéndose a la manera uniforme de en la cual los procesadores son conectados a la memoria compartida. La mejor manera de entender el concepto de procesadores simétricos es pensando que todos los procesadores tienen igual acceso a los recursos compartidos, es decir, memoria compartida y funcionalidades de I/O. El tiempo requerido para que un procesador acceda a la memoria debe ser el mismo para todos los procesadores. Actualmente estos sistemas tienen de 2 a 64 procesadores, y una sola copia del sistema operativo corre en ese sistema.

Los sistemas MPP que antes de los noventa eran considerados como MIMD-DM, se generalizan como la interconexión de estaciones de trabajo (workstations) y de PCs rápidas las cuales se basan en microprocesadores RISC. Para simplificar la descripción de este tipo de máquinas podemos dividirla en procesadores, memoria con varios niveles de acceso incluyendo cache, una red de interconexión o switch y soporte de hardware (figura 1.17)

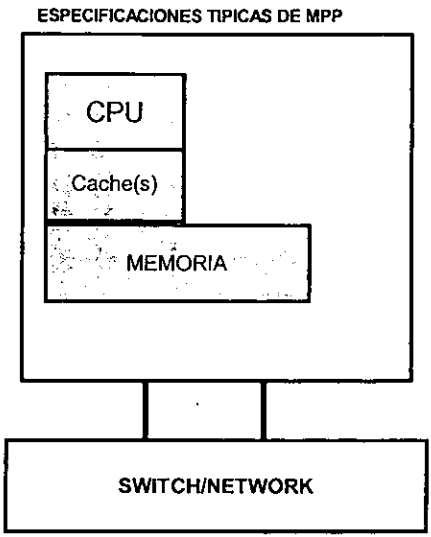


Figura 1. 17 Especificaciones típicas de MPP.

Los nodos especiales pueden tener conectados periféricos como discos o sistema backup. En cada nodo corre una copia separada del sistema operativo.

Los sistemas considerados más poderosos en aplicaciones hasta ahora son los SMP y MPP. En MPP la memoria es distinta (así el desempeño es muy escalable). En las SMP los procesadores comparten una área global de memoria RAM. El propósito de esta arquitectura es evitar problemas asociados con la comunicación entre procesadores. Sin embargo el diseño de memoria compartida del SMP puede traer como consecuencia cuellos de botella, si se agregan más procesadores (por ejemplo un bus o switch usado para acceder a la memoria puede ocasionar una sobrecarga). Es por eso que los sistemas SMP no tienen el mismo potencial de escalabilidad como los MPP. Una manera de evitar problemas en la escalabilidad en los sistemas SMP es reintroduciendo NUMA en el sistema. NUMA agrega otro nivel de memoria, compartida entre un subconjunto de procesadores y se puede acceder a ella sin usar el bus de interconexión de red.

En esta extensa generalización de las arquitecturas SMP algunas veces encontramos a la **CC-NUMA** que es un sistema multiprocesador escalable que tiene un coherente cache de no uniforme acceso a memoria (*cache coherency*). En este sistema todos los procesadores comparten todos los recursos globales disponibles. Como en una SMP, todos los procesadores en una CC- NUMA tienen una vista global de toda la memoria. Y obtiene su nombre del no

uniforme tiempo para acceder a las más cercanas y remotas partes de la memoria.

Los **sistemas distribuidos** pueden ser considerados convencionalmente como redes de computadoras independientes. Tienen múltiples imágenes del sistema, cada nodo corre su propio sistema operativo y una máquina individual en este tipo de sistemas puede ser por ejemplo combinaciones de MPPs, SMPs, Clusters y computadoras individuales.

Los **Clusters** es una colección de estaciones de trabajo o PCs que se interconectan por medio de alguna tecnología de red. Para propósito de computación paralela, un Cluster del alto desempeño puede consistir de la interconexión generalmente de workstation o PCs por medio de una red de alta velocidad. Un *Cluster* trabaja como una colección integrada de recursos y puede tener una única imagen del sistema dispersa (spanning) a través de todos sus nodos.
La tabla 1.1 muestra una comparación de las arquitecturas y características funcionales de las máquinas mencionadas anteriormente, originalmente propuesta por Hwang y Xu[2].

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

Tabla 1.1 Comparación de las arquitecturas y características contemporáneas.

[2] **High Performance Cluster Computing**: *Architectures and Systems*, Vol. 1, 1/e
Rajkumar Buyya
School of Computer Science and Software Engenering
Monash University, Melbourne,Australia
Copyright 1999, 881 pp
Cita capítulo 1 pag

1.3. CLUSTERS

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

Una computadora nodo puede ser un sistema con un solo procesador o con multiprocesadores (PCs, WorkStation o SMPs), con memoria, facilidades de I/O y un sistema operativo. Un Cluster generalmente se refiere a dos o más computadoras (nodos) conectadas entre sí. Los nodos pueden estar en un solo gabinete o separados físicamente conectados vía LAN. Un Cluster de computadoras interconectadas (basadas en LAN) pueden parecer un solo sistema a usuarios y aplicaciones. Igualmente el sistema puede proveer una forma de costo eficiencia para ganar en las características y beneficios que han sido históricamente encontrados en costosos sistemas propietarios de memoria compartida.

La arquitectura típica de un Cluster se muestra en la figura 1.18.

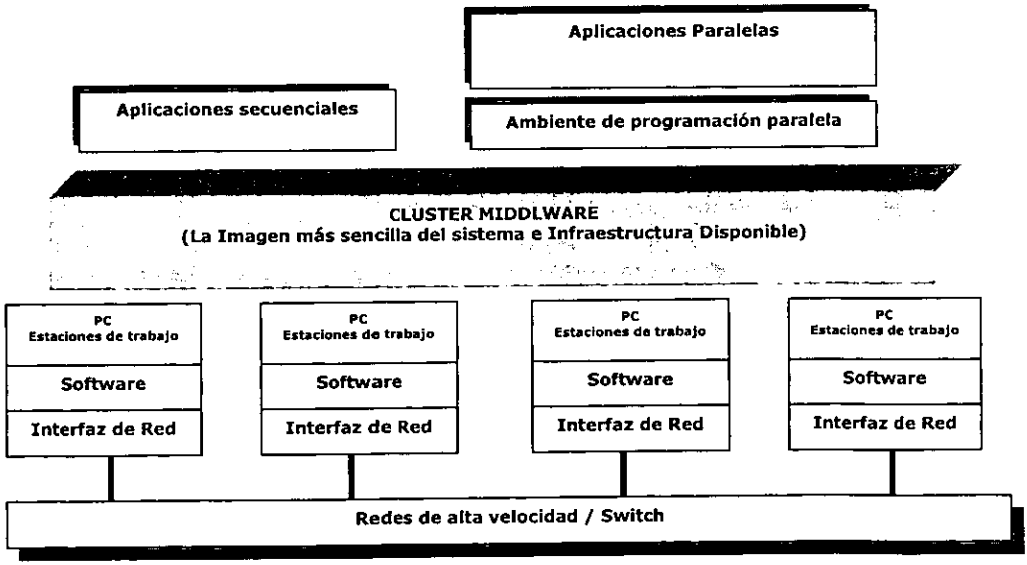


Figura 1. 18 Arquitectura típica de un Cluster.

Los siguientes componentes generales de una computadora tipo Cluster:

- Múltiples computadoras de alto desempeño (PCs, WS, SMPs)
- Sistemas operativos hechos casi en estado de arte [State-of-the-art OS], (en sistemas basados en varios niveles o basados en Micro Kernel)
- Redes/Switches de alto desempeño (como son Gigabit Ethernet y Myrinet).
- Tarjetas de interfaz de red (NIC)
- Protocolos y servicios de rápida comunicación, (como son Mensajes Activos y Rápidos, [Active and Fast Messages])
- Cluster Middleware (Single System Image (SSI) y System Availability Infraestructure)
  - Hardware (como son Canal de Memoria (DEC) Digital, Hardware DSM, y técnicas SMP)
  - Kernel del Sistema operativo o niveles adheridos [Gluing Layer], (Como son SOLARIS MC y GLUnix)
  - Aplicaciones y subsistemas
    - Aplicaciones (como herramientas de manejo del sistema y formas electrónicas)
    - Sistemas en tiempo de ejecución (como el software DSM y sistemas de archivos paralelos)
    - Manejo de Recursos y Software de calendarización (como el LSF (Load Sharing Facility) y CODINE (Computing in Distributed Networked Enviroments))
- Ambientes y herramientas de programación paralela (Como son los compiladores, PVM (Parallel Virtual Machine) y MPI (Message Passing Interface))
- Aplicaciones
  - Secuencial
  - Paralelos o distribuidos

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

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

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

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

1.3.1. CLASIFICACIÓN DE CLUSTERS

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

- Alto desempeño
- Expansibilidad y escalabilidad
- Alta productividad
- Alta disponibilidad

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

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

Los Clusters son clasificados en diferentes categorías basadas en varios factores que son indicados a continuación

1. Propósito de la aplicación: Ciencias computacionales o aplicaciones de misión crítica

- Clusters de Alto Desempeño (*High Performance (HP)*)
- Clusters de Alta disponibilidad (*High Availability (HA)*)

2. Propiedad de los Nodos: Los nodos pertenecen a un Cluster individual o a uno dedicado

- Cluster Dedicado
- Cluster no dedicado

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

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

3. Hardware de los Nodos: Pc, estaciones de trabajo (workstation) o SMP.
- Cluster de PC (CoPs) o Pilas de PCs (PoPs)
- Clusters de estaciones de trabajo (COWs).
- Clusters de SMPs (CLUMPs)

4. Sistema Operativo del Nodo: Linux, NT, Solaris, AIX, etc.
- Clusters basados en Linux (Por ejemplo Beowulf)
- Clusters basados en Solaris (Por ejemplo Berkeley NOW)
- Clusters basados en NT (Por ejemplo HPVM)
- Clusters basados en AIX (Por ejemplo IBM SPS2)
- Clusters Digitales VMS
- Clusters basados en HP-UX
- Clusters Microsoft Wolfpack.

5. Configuración del Nodo: Arquitectura del nodo y el tipo de sistema operativo que tiene.
- Clusters Homogéneos. Todos los nodos tienen una arquitectura similar y corren bajo el mismo sistema operativo.
- Clusters Heterogéneos. Todos los nodos tienen diferentes arquitecturas y corren bajo diferentes sistemas operativos.

6. Niveles de Agrupamiento (Clustering): Basados en la localización de los nodos y su número.

- Grupo de Clusters (Número de nodos: 2 – 9): Los nodos son conectados por SANs (System Area Network) conocidas como Myrinet y están apiladas en un frame o que existen en un centro.
- Clusters Departamentales (Número de nodos: 10 – 100).
- Clusters Organizacionales (Número de nodos: más de 100).
- Metacomputadoras Nacionales (WAN/basadas en internet): Número de nodos: gran cantidad de los dos anteriores.
- Metacomputadoras internacionales (Basadas en Internet) de 1000 nodos a muchos millones.

Los Clusters pueden ser interconectados para formar grandes sistemas (Clusters de Clusters), y de hecho, la Internet puede ser usada como un Cluster de computadoras.
El uso de los recursos de una red amplia para cómputo de alto desempeño ha conducido al surgimiento de un nuevo campo llamado METACOMPUTACIÓN.

1.3.2. COMPONENTES DE HARDWARE

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

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

HARDWARE DE NODOS

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

Los subsistemas o componentes que deben integrar un nodo son los siguientes (se da una breve descripción):

PROCESADOR: Actualmente la unidad central de proceso es un complejo subsistema que incluye, la esencia de un elemento de procesamiento, dos niveles de cache, y controladores de bus externos. Hay procesadores de 32 y 64 bits ambos disponibles en arquitecturas populares como la familia INTEL Pentium pro, Pentium II/III y la COMPAQ Alpha 21264. Otros incluyendo la IBM PowerPC, la Sun Microsystems Super Sparc III, y AMD K7 Athlon.

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

Actualmente tenemos los procesadores Intel Pentium III y IV con mayores velocidades (Pentium IV de hasta 1.7 GHz) , lo que nos permitiría un mayor desempeño.

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

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

MEMORIA: Originalmente la memoria que se encuentra ya en la computadora es de 64 Kbits. Hoy una PC puede contener entre 32, 64, 128 Mbytes y más, instalados en los slots de la computadora de tipo SIMM (Standar Industry Memory Module) o DIMM.

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

La cantidad de memoria que necesita un Cluster depende del objetivo de aplicación a la que se enfoque. Los programas a ser paralelizados deberán estar distribuidos de manera que tanto la memoria como el procesamiento esté distribuido entre los procesadores con fines de escalabilidad. No es necesario tener RAM que mantenga todo el problema en memoria pero sí debería haber suficiente como para, evitar el exceso de *swapping (intercambio)* de bloques de memoria(*page -misses*) a disco.

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

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

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

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

La capacidad de memoria del nodo principal debe ser de 64 Mbytes a mayores de 1 Gbyte, encontrados en los SIMMs , DIMMs o RIMM para pentium IV . Los chips pueden contener individualmente 64 Mbits y emplear algunos posibles protocolos de interfaz. SDRAM es más ampliamente usada para proveer un gran ancho de banda de memoria.

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

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

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

Los discos duros son la principal forma de almacenamiento secundario, los más populares son los SCSI II (*Small Computer Systems Interface — Interfaz para sistemas de pequeñas computadoras*) y EIDE (*Enhanced Integrate Drive Electronics – Electrónica de unidad integrada mejorada*). Los SCSI proveen mayor capacidad y un superior acceso de ancho de banda (*bandwidth*), mientras que la ventaja que tienen los EIDE es su bajo costo.

Las actuales generaciones de discos proveen entre 20 y 100 Gbytes de almacenamiento con un tiempo de acceso de milisegundos. Los CD-ROM y CD-RW han llegado a considerarse otros medios de almacenamiento, debido a su bajo costo, manejo de código y datos y apoyo de almacenamiento.

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

El primer bus usado en *PC* (AT o ahora conocido como bus ISA) iba a 5MHz y tenía un ancho de 8 bits.

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

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

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

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

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

HARDWARE DE REDES PARA CLUSTERS

La construcción de Clusters *commodity* es posible gracias a las tecnologías de red adecuadas que permiten la interconexión entre los nodos. Estas redes de interconexión interactuando con el hardware y software adecuados permiten que los paquetes de datos se transfieran entre los nodos o procesadores que conforman el Cluster.
Los Clusters *commodity* incorporan una o más redes dedicadas para la comunicación de paquetes de mensajes en sistemas distribuidos

INTERCONEXIONES EN LOS CLUSTERS

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

Además de las LAN (Redes de Área Local) Ethernet, existen otros dos tipos de interconexión, la Token Ring y FDDI, que también son populares, pero que no son usadas tan comúnmente en una interconexión de nodos en un Cluster.
A continuación se da una breve descripción de las redes de interconexión más comúnmente usadas en un Cluster.

ETHERNET, FAST ETHERNET Y GIGABIT ETHERNET

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

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

2) Una versión, llamada Fast Ethernet, provee un ancho de banda de 100Mbps (opera a 100Mbps sobre cable par trenzado) y ha sido diseñada para actualizar las ya existentes redes Ethernet. La tecnología Fast Ethernet y la estándar no pueden coexistir en un mismo cable, aunque utilizan el mismo tipo.

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

ATM (Asynchronous Transfer Mode)

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

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

ATM está diseñada teniendo en mente que las celdas (células) puedan ser transferidas a través de diferentes medios (cobre, fibra óptica, etc.). Esto hace que tenga diferentes niveles de desempeño.
En una red ATM las comunicaciones se establecen a través de un conjunto de dispositivos intermedios llamados switches.

SCI (Scalable Coherent Interfaz):

Es el equivalente moderno de un bus Procesador - Memoria - E/S y una red de Área local, combinados y trabajando en paralelo para soportar multiproceso distribuido con un elevado ancho de banda, muy baja latencia y una arquitectura escalable que permite diseñar grandes sistemas evitando la utilización de grandes bloques.
SCI surge como una ramificación del proyecto IEEE Standard Futurebus en 1988, cuando se hizo evidente que los futuros Procesadores serían pronto demasiado rápidos para cualquier bus y para soportar un coste razonable en configuraciones multiproceso. El grupo SCI buscó una nueva solución que proporcionara al usuario servicios propios de un bus pero evitando los cuellos de botella inherentes en los buses; escalabilidad en diseños con supercomputadores; y soporte efectivo de software para sistemas y aplicaciones de procesamiento paralelo.
Las siglas de SCI se corresponden con:

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

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

Interfaz: Disponer de una arquitectura de comunicación abierta que permita la utilización de múltiples productos de diversos fabricantes.

Principales características de SCI

- Es un estándar que permite a los sistemas crecer con componentes modulares de diferentes fabricantes.
- Flujo de datos en el anillo de hasta 1 GByte/s por nodo.
- Memoria compartida.
- Opcionalmente coherencia *cache* basada en directorios distribuidos.
- Mecanismos de paso de mensajes también posibles.
- Escalable hasta 64K procesadores.
- Enlaces de datos unidireccionales de 16 bits cada 2 ns.

- Disponibilidad de interfaz Single-chip que incluyen todos los *transceivers (transductores)*, FIFOs y lógica de protocolo con el consiguiente ahorro de costos y simplicidad.
- Enlaces de fibra óptica o coaxial de 1 Gbit/s también posibles para aplicaciones LAN (*Local Area Network*) con protocolos eficientes de memoria compartida.
- Mecanismos de interfaz a otros buses comerciales (VME, SBUS, PCI,...).
- Arquitectura CSR (*Control State Registers*, IEEE Std 1212-1991).

MYRINET

Myrinet es una red rápida de área local (LAN) que conecta estaciones de trabajo o PC's de alto rendimiento mediante un enlace de 1.28 GigaBits/s full-duplex, con baja latencia, y también es un paquete enrutador de alta tecnología que usa redes del tipo SAN (*System-Area*) y redes LAN (*Local-Area*).

Myrinet está basada en la tecnología usada para comunicación de paquetes y supercomputadoras paralelas. Redes convencionales como Ethernet pueden ser usadas para construir Clusters, pero no proveen las características y desempeño requeridas por los Clusters de alta disponibilidad y alto desempeño (high-performance o high-availability Clustering).

Las características que la distinguen son:

- Enlace full-duplex 1.28+1.28 GigaBits/s, con puertos de switches y puertos de interfaz.
- Control de flujo y control de errores en cada conexión.
- Baja latencia.
- Interfaz en las estaciones de trabajo que permiten ejecutar un programa de control que maneja directamente la interacción entre las estaciones, los buffers, los mapeos de la red y el monitoreo de la misma
- Las redes Myrinet pueden escalar de 10 a 100 host, con una tasa de datos de Terabits por segundo, y puede proveer direcciones de comunicación alternativa entre los hosts.
- La conexión de la red es de arquitectura abierta, incluyendo las arquitecturas que permiten múltiples conexiones para desempeño y redundancia. Myrinet mapea la interfaz y permite la comunicación entre procesos.

El software desarrollado por Myrinet para la red, reporta cortas latencias entre procesos de Unix (menor a cinco microsegundos), esto es mejor que el mejor sistema de memoria distribuida (MPP) conocido con anterioridad, y requiere de una tasa de transmisión de 1 GigaBit/s. Ya es posible implementar sobre Myrinet muchos de los sistemas de cómputo paralelos actuales (MPP), como MPI y PVM.

El Menssage-Passing Interfaz (MPI) y Parallel Virtual Memory (PVM), están implementados directamente en la capa UDP/IP de Myrinet. A través de estos se logra un mejor desempeño (performance).

Este tipo de conexión es la favorita para las MPP. A diferencia de los de las conexiones anteriores para MPP's, los basados en Myrinet son implementados con plataformas heterogéneas (las más comunes son Pentium Pro, Sun SPARC y DEC Alfa, todas workstation de alto rendimiento), cada uno con su propio sistema operativo y aplicaciones seguras.

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

1.3.3. SISTEMAS OPERATIVOS

Un sistema operativo moderno provee dos servicios fundamentales para los usuarios. Primero, hace que el hardware de la computadora sea fácil de usar. Crea una máquina virtual que difiere notablemente de la máquina real. De hecho, la revolución que ha habido en las computadoras en las últimas dos décadas es debido, en parte, al éxito que los sistemas operativos han logrado al proteger a los usuarios de las oscuridades que hay en el manejo del hardware. Segundo, un sistema operativo comparte los recursos de hardware con los usuarios. Uno de los más importantes recursos es el procesador. Un sistema operativo multitarea, como lo es UNIX o Windows NT, divide el trabajo que se necesita para que sea ejecutado entre los procesadores, dando a cada procesador memoria, recursos del sistema, con al menos un hilo de ejecución y una unidad de ejecución sin procesos. El sistema operativo corre un hilo por un tiempo corto y luego hace el cambio hacia otro hilo, corriendo uno a la vez. Incluso en sistemas de un solo procesador, la multitarea es extremadamente útil debido a que habilita a la computadora a desarrollar múltiples tareas a la vez. Por ejemplo, un usuario que edita un documento mientras otro documento es impreso en *background* o mientras un compilador traduce un programa muy extenso. Cada proceso obtiene lo necesario para trabajar, y al usuario le parece que todos los programas corren simultáneamente.

Aparte de los beneficios mencionados anteriormente, el nuevo concepto en los servicios de los sistemas operativos es el soporte a múltiples hilos de control en los mismos procesos. Este concepto se ha sumado a una nueva dimensión del procesamiento paralelo, el paralelismo sin procesos, en lugar de que sea a través de programas. En la siguiente generación de Kernels para los sistemas operativos, las direcciones de espacio y los hilos son separados hacia un solo espacio de direccionamiento que pueda tener múltiples hilos de ejecución. La programación de procesos con múltiples hilos de control es conocida como multithreading (multihilaje). La interfaz de hilos que proporciona POSIX es un estándar de ambiente de programación para crear concurrencia / paralelismo sin procesos.

Un número de tendencias de diseño que afectan a los sistemas operativos se han visto en los últimos años, la mayoría de estas tendencias con dirección hacia la modularidad. Sistemas Operativos como Windows de Microsoft, OS/2 de IBM, y otros, han sido descompuestos dentro de componentes discretos, cada uno siendo pequeño, con una interfaz bien definida, y cada uno comunicándose con otros por medio de una interfaz de mensajes entre tareas. El nivel más profundo es el micro-Kernel, el cual provee solamente los servicios esenciales del sistema operativo, como es el intercambio de contexto o context switching. Windows NT, por ejemplo, también incluye una capa de abstracción de hardware, (*Hardware abstraction layer (HAL)*), debajo del micro-Kernel, lo que posibilita al resto del sistema operativo desarrollarse

independientemente de la capa adyacente del procesador. Este alto nivel de abstracción de la portabilidad de los sistemas operativos es una fuerza que los está llevando hacia la modularidad, basada en un micro-Kernel. Otros servicios se ofrecen a los subsistemas construidos en la parte superior del micro-Kernel. Por ejemplo, los servicios para los archivos pueden ser ofrecidos por el servidor de archivos, el cual es construido como un subsistema en la parte superior del micro-Kernel.

Esta parte se enfocará a los diferentes sistemas operativos disponibles para las estaciones de trabajo y las PCs. La tecnología ha madurado y puede ser fácilmente extendida a nuevos subsistemas que pueden ser adheridos sin modificar la estructura del sistema operativo. Los sistemas operativos modernos soportan multihilaje (multithreading) a nivel de Kernel y un alto desempeño en el nivel de usuario en los sistemas multithilaje que puede ser construido sin la intervención del Kernel. La mayoría de los sistemas operativos para PC son estables y tienen un soporte multitarea, multihilaje, y soporte para red.

UNIX y sus variantes (como son Solaris de Sun, AIX de IBM, HP UX) son de los más usados en las estaciones de trabajo. En esta sección, se discutirá un poco los tres sistemas operativos más populares que son usados en los nodos de los Clusters de PCs o de Estaciones de trabajo (WS).

LINUX.

Linux es un sistema operativo similar a UNIX el cual fue inicialmente desarrollado por Linus Torvalds, cuando era estudiante en los años de 1991-92. Las versiones originales de Linux habían sido realizadas con base en el sistema operativo Minix; sin embargo, los esfuerzos de un numeroso grupo de programadores han resultado en el desarrollo e implementación de un sistema robusto, confiable y hecho conforme a POSIX.

A pesar de que Linux fue desarrollado por un solo autor inicialmente, un gran número de autores están involucrados en su desarrollo. Una gran ventaja de este desarrollo distribuido es visto en el amplio panorama de herramientas de software, librerías y utilidades disponibles. Esto es debido al hecho de que cualquier programador capaz tiene acceso al código fuente del sistema operativo y puede implementar la característica que él desee. El control de calidad de Linux es mantenido sólo por las versiones del Kernel permitidas desde un solo punto, y está disponible vía Internet, dando una rápida realimentación acerca de los *bugs* (errores) y otros problemas. Los siguientes puntos muestran ventajas de usar Linux:

- Linux corre sobre plataformas x86, baratas, ofreciendo el poder y la flexibilidad de UNIX.
- Linux está disponible en el Internet y puede ser descargado sin costo.

- Los *bugs* que contenga son fácilmente reparados y se puede mejorar el desempeño.
- Los usuarios pueden desarrollar manejadores para el hardware que permitan un mejor aprovechamiento del mismo y pueden ser disponibles fácilmente para otros usuarios.

Linux provee las características típicas encontradas en las implementaciones UNIX, como son:

Multitarea apropiativa (preemptive multitasking), paginación por demanda de la memoria virtual, multiusuario y soporte multiprocesador. La mayoría de las aplicaciones escritas para UNIX requiere solamente una recompilación. En adición al Kernel de Linux, una gran cantidad de aplicaciones /sistemas de software son disponibles sin costo, incluyendo software GNU y Xfree86, un servidor basado en X de dominio público.

Solaris.

El sistema operativo Solaris de SunSoft es un sistema basado en multihilaje y en multiusuario. Este sistema soporta las plataformas Intel x86 y basadas en SPARC. En su soporte de red incluye el conjunto de protocolos TCP/IP y características basadas en capas como son las Remote Procedure Calls (RPC, Llamadas Remotas a Procesos), y su Sistema de Archivos de Red (Network File System (NFS)). El ambiente de programación de Solaris incluye los compiladores para C y C++ de ANSI, así como herramientas para desarrollar y depurar programas multitareas.

El Kernel de Solaris soporta multihilaje, multiprocesamiento, y características de calendarización en tiempo real que son críticas para aplicaciones multimedia. Solaris soporta dos tipos de hilos: Light Weight Processes (LWPs, Procesos de peso ligero) e hilos a nivel de usuario. Los hilos deben ser proyectados para que sean suficientemente ligeros así que pueda manejar cientos y que la sincronización y el cambio de contexto pueda ser llevado a cabo sin la intervención del Kernel

Solaris, junto con el sistema de archivos BSD, también soporta varios tipos de sistemas de archivos no BSD, lo que incrementa su desarrollo y facilidad de uso. Para su desempeño tiene tres nuevos sistemas de archivos: CacheFS, AutoClient, y TmpFS. El CacheFS guarda un sistema de archivos permitiendo al disco local ser usado y manejado por otro sistema remoto NFS o sistema de archivos, desde un disco o CD-ROM. Con AutoClient y CacheFS, un disco local puede ser completamente usado como cache. El TmpFS, es un sistema de archivos temporal que usa la memoria principal para contener un sistema de archivos. Además, hay otros sistemas de archivos como *Proc* y *Volume* que mejoran la utilidad del sistema.

Solaris soporta cómputo distribuido y está habilitado para almacenar y recuperar información distribuida para describir el sistema y los usuarios a través del Network Information Service (NIS), y bases de datos. La interfaz gráfica de Solaris, Solaris GUI, Open Windows, es una combinación del X111R5 y el sistema Adobe PostScript, lo que permite que aplicaciones que se corren de manera remota se puedan desplegar de manera local con algunas aplicaciones.

Microsoft Windows NT.

Microsoft Windows NT(New Technology), es el sistema operativo dominante en el mercado de las computadoras personales. Es apropiativo, multitarea, multiusuario, y un sistema operativo de 32 bits. NT soporta múltiples CPUs y provee multitarea al usar multiprocesamiento simétrico. Cada aplicación para NT de 32 bits opera en su propia dirección de memoria virtual. De manera diferente a las versiones anteriores (como Windows for Workgroups y Windows 95/98), NT es un sistema operativo completo, no una mejora del DOS. NT soporta diferentes CPUs y máquinas multiprocesadores con hilos. También tiene un modelo de seguridad basado en objetos y su propio sistema de archivos especial (NTFS) que permite cambiar los permisos en un archivo con base en los permisos en el directorio.
Un diagrama esquemático de la arquitectura del NT se muestra a continuación. NT tiene protocolos de red y servicios integrados con la base del sistema operativo.

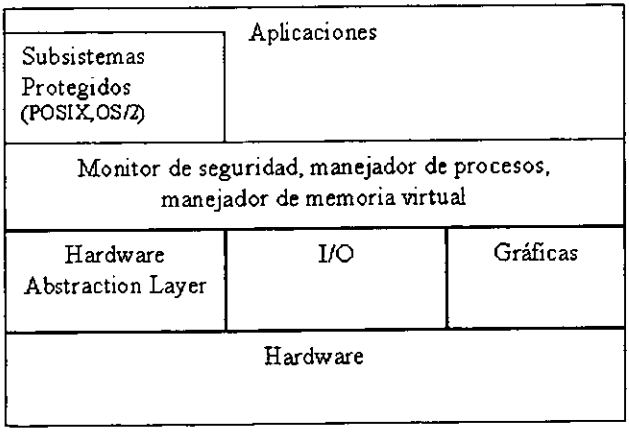


Figura 1. 19 Diagrama de la arquitectura de Windows NT.

Empaquetados con el Windows NT hay algunos protocolos de red que han sido incluidos dentro de él, como son IPX/SPX, TCP/IP, y NetBEUI con sus APIs, como son NetBIOS, DCE RPC, y Windows Sockets, (Winsock). La aplicaciones de TCP/IP usan los Winsock para comunicarse a través de una red basada en TCP/IP.

1.4. BEOWULF

¿QUIÉN FUE Y QUÉ ES UN CLUSTER BEOWULF?

BEOWULF es un poema épico que fue escrito en el siglo VIII y que es uno de los textos sobrevivientes once siglos después de lo viejos manuscritos ingleses lo cual lo hace la primera pieza conocida de la literatura inglesa. Esto cuenta la historia del guerrero Geat Beowulf, quien salva al señor de los Daneses y su corte, de las venganzas del malvado monstruo Grendel. Este poema no contiene referencias al cómputo paralelo debido que fue escrito hace cientos de años antes de que las computadoras se conocieran y se inventaran, pero ha inspirado el desarrollo de las computadoras tipo Beowulf durante los años finales del segundo milenio.

El héroe Beowulf libera a los Daneses del monstruo Grendel, el cual visitaba el salón Heorot cada noche y aterrorizaba a su gente. De la misma manera los sistemas Beowulf han liberado a los científicos de recurrir a tareas opresivas para desarrollar sus códigos y nuevas arquitecturas cada año, compartiendo recursos de cómputo con cientos de otros usuarios.

El primer Beowulf fue construido en la NASA. En el verano de 1994 Thomas Sterling y Don Becker, quienes trabajaban bajo contrato en la NASA, construyeron una computadora de procesamiento paralelo consistente de 16 procesadores 486 DX4. A partir de la construcción del primer Beowulf se empezaron a construir diversas máquinas paralelas consideradas como Beowulf , se tomaron las bases del primer Beowulf y se fueron mejorando.

Existen varias definiciones de BEOWULF entre las que encontramos las siguientes:

a) Es una colección de Computadoras Personales interconectadas por alguna tecnología de red y que trabaja bajo un SISTEMA OPERATIVO de código abierto (LINUX y variantes) que son similares al sistema operativo UNIX.

b) Un Cluster Beowulf está esencialmente construido por un conjunto de PCs conectadas en una red, relativamente rápida.

c) Es un grupo de máquinas LINUX dedicadas a un propósito específico y puede ser llamado Cluster. Un Cluster Beowulf necesita tener un nodo central que coordine a las demás máquinas.

La siguiente definición es la que consideramos más completa ya que se toman en cuenta todas las características:
d) Una arquitectura multi-computadora que puede ser usado para computación paralela. Es un sistema que generalmente consiste de un nodo servidor y uno o varios nodos clientes conectados a través de una red

Ethernet u otro tipo de interconexión de red. Es un sistema construido mediante hardware de tipo *commodity*, como cualquier PC capaz de ejecutar Linux, tarjetas de red Ethernet y cables. No contiene ningún componente de hardware específico y es fácilmente reproducible. Beowulf también usa un software libre, como el sistema operativo Linux, *PVM, MPI* y actualmente *MOSIX* . El nodo servidor controla el Cluster completo y maneja (sirve a) los directorios de los nodos cliente; también hace de consola del Cluster y es su enlace con el mundo exterior. Los grandes sistemas Beowulf pueden tener más de un nodo servidor, y también algunos nodos dedicados a tareas específicas, como consolas o puestos de monitorización. En la mayoría de los casos, los nodos clientes de un sistema Beowulf son mudos, y cuanto más mudos, mejor. Los nodos son configurados y controlados por el nodo servidor, y sólo hacen lo que se les dice que hagan. En las configuraciones donde los clientes no tienen discos, los nodos clientes ni siquiera conocen su IP hasta que el servidor les dice cuál es. Una de las principales diferencias entre Beowulf y los Clusters de Estaciones de Trabajo (*COW*) es el hecho de que Beowulf se comporta más como una única máquina que como varias estaciones de trabajo. En la mayoría de los casos, los nodos clientes no tienen teclado ni monitor, y sólo se puede acceder a ellos a través de acceso remoto o tal vez por medio de terminales serie. Puede pensarse en un nodo Beowulf como un conjunto CPU + memoria que puede ser conectado al Cluster, del mismo modo que una CPU o un módulo de memoria puede ser conectado a una placa base.

A pesar de que hay muchos paquetes de software, como modificaciones del núcleo, bibliotecas para PVM y MPI y herramientas de configuración que hacen más rápida, fácil de configurar y de usar la arquitectura Beowulf, cualquiera puede construir una máquina de tipo Beowulf usando una distribución normal de Linux sin ningún software adicional. Si se tienen dos PC con Linux en red que comparten el sistema de directorios */home* vía NFS y se permiten ejecutar shells remotos *(rsh),* entonces podría decirse que usted tiene una máquina Beowulf simple de dos nodos.

ARQUITECTURA BEOWULF

Con la definición anterior de Beowulf nos podemos dar una idea de qué consiste un Beowulf, pero en este apartado se trata de explicar cómo estaría conformada la arquitectura. Por lo que para describir mejor la arquitectura de un cluster Beowulf es usar un ejemplo que es muy similar a un auténtico Beowulf, y familiar para la mayoría de los administradores de sistemas. El ejemplo que más se acerca a una máquina Beowulf es una sala de máquinas Unix, con un servidor y varios clientes. Todos los clientes tienen instalada una copia local del sistema operativo Digital Unix 4.0, pero obtienen el espacio de directorio de usuario *(/home)* y */usr/local* del servidor a través de NFS (Network File System). Cada cliente tiene una entrada para el servidor y para todos los demás clientes en su directorio */etc/hosts.equiv*, de manera que todos los clientes puede ejecutar un shell remoto *(rsh)* en cualquiera de las

otras máquinas. Una persona puede sentarse en la consola de uno de los nodos y acceder al sistema y tener el mismo entorno que si hubiese entrado a través del servidor. La razón de que todos los clientes tengan el mismo aspecto y comportamiento (*look and feel*) es que el sistema operativo está instalado y configurado de la misma manera en todas las máquinas y que tanto el */home* como el */usr/local* están físicamente en el servidor y son accedidos por el cliente a través de NFS.
Pero ¿Cuál sería la diferencia entre un Beowulf y Cluster formado con estaciones de trabajo?.

Diferencia entre Beowulf y un COW

La sala de máquinas descrita arriba es un ejemplo perfecto de un Cluster de estaciones de trabajo (COW). En realidad un Beowulf tiene unas pocas características únicas. Entre las que encontramos las siguientes:

- En la mayoría de los casos los nodos clientes de un Cluster Beowulf no tienen teclado, ratón, tarjeta gráfica o monitor.
- Todos los accesos a los nodos cliente son realizados a través de conexiones remotas desde el nodo servidor.
- Dado que no hay necesidad para los nodos clientes de acceder a máquinas ajenas al Cluster, ni al revés, es una práctica muy frecuente utilizar direcciones IP privadas para los clientes, como las familias 10.0.0.0/8 o 192.168.0.0/16
- Normalmente, la única máquina que está conectada al mundo exterior utilizando una segunda tarjeta de red es el nodo servidor.
- Las formas más usuales de usar el sistema es accediendo a la consola del servidor directamente o a través de telnet o acceso remoto al servidor desde una estación de trabajo personal. Una vez en el nodo servidor, los usuarios pueden editar y compilar su código, y también lanzar las tareas en todos los nodos del Cluster.
- En muchos casos, se usan COWs para computación paralela durante la noche y fines de semana, cuando la gente no está usando sus estaciones de trabajo para el trabajo diario, aprovechando los ciclos libres de CPU. Beowulf, en cambio, es una máquina generalmente dedicada a la computación paralela, y optimizada para este propósito.
- Beowulf también ofrece una relación precio/rendimiento mejor, ya que está construida con componentes *commodity* y ejecuta principalmente software libre.

- Beowulf es un único sistema que ayudan a los usuarios a ver el Cluster como un único sistema de computación

Si desea conocer los mejores clusters basados en Linux, visite la siguiente página: http://clusters.top500.org/

Capítulo II

Sistema Operativo Linux

2. EL SISTEMA OPERATIVO LINUX

2.1. HISTORIA

Linux fue creado originalmente por Linus Torvalds en la Universidad de Helsinki en Finlandia, siendo él estudiante de informática. Linus originalmente inició el desarrollo del núcleo como su proyecto favorito, inspirado por su interés en Minix, un pequeño sistema Unix desarrollado por Andy Tannenbaum. Él se propuso crear lo que en sus propias palabras seria un "mejor Minix que el Minix".

2.1.1. EVOLUCIÓN

- Linus nunca anunció la versión 0.01 de Linux (agosto 1991), esta versión no era ni siquiera ejecutable, solamente incluía los principios del núcleo del sistema, estaba escrita en lenguaje ensamblador y asumía que uno tenía acceso a un sistema Minix para su compilación
- El 5 de octubre de 1991, Linux anunció su primera versión "oficial" de Linux, versión 0.02 aparece como evolución de MINIX para el 80386. Con esta versión Linus pudo ejecutar *Bash* (GNU *Bourne Again Shell*) y *gcc* (El compilador GNU de C) pero no funcionaba mucho más
- En menos de un año, más de 100 programadores colaboran.
- Los fuentes (versiones) se difunden con la máxima frecuencia y cualquiera que quiera modificar o criticar algo, lo hace. Los más conocedores de esa área (los que la han programado) deciden si es útil y si lo es, lo incorporan.
- Se portan las herramientas GNU de la FSF (*Free Software Foundation*) *gcc, gdb, bash, emacs, etc.*
- Linus incrementó el número de versión hasta la 0.95 (Marzo 1992)
- Un año después (diciembre 1993) el núcleo del sistema estaba en la versión 0.99.
- En marzo de 1994 aparece Linux 1.0 en forma de "distribución".
- En junio de 1996 se distribuye Linux 2.0, ya competitivo con otros UNIX. Aparecen varias distribuciones (*RedHat, Caldera, S.U.S.E,*

Slackware, Debian, etc., algunas con soporte oficial para empresas)

- La FSF (*Free Software Fundation*) adopta Linux como Sistema Operativo Oficial: GNU/Linux.
- Interés por parte de las empresas (*Intel, Sun, Netscape, Lotus, Adobe, Corel, Oracle, Informix, Sysbase*, etc.)
- En enero de 1999 aparece *Linux 2.2* con muchas mejoras y soporte para nuevos tipos de hardware. El primer número cambia cuando se da una evolución importante, el segundo es la versión y el tercero la revisión. Las versiones pares son estables y las impares inestables.
- En enero de 2001 aparece Linux 2.4 con mejoras en el soporte a multiprocesadores, dispositivos como el *USB*, y acceso directo al HW gráfico (2D, 3D).

Año	Usuarios	Versión y capacidades	Líneas de código
1991	1	0.01. Linus Torvalds diseña un Kernel y unos manejadores de teclado y pantalla, dejándolos al alcance de todos por FTP.	10.000
1992	1.000	0.96. Linux empieza a ser funcional y se incorpora un interfaz gráfico.	40.000
1993	20.000	0.99. Cientos de programadores aportan cambios y mejoras al código. Torvalds delega en un grupo de 5 personas la tarea de revisión y selección de código.	100.000
1994	100.000	1.0. Se añade soporte de red.	170.000
1995	500.000	1.2. Se transporta a las plataformas Digital y Sun SPARC. Aparece la revista *Linux Journal* en EE.UU. con una tirada de 10.000 ejemplares.	250.000
1996	1,5 mill.	2.0. Se añade soporte para multiproceso.	400.000
1997	3,5 mill.	2.1. Aparecen revistas dedicadas a Linux en España, Japón, Polonia, Alemania, Yugoslavia, el Reino Unido y otros países. Se anuncian nuevas versiones del Kernel prácticamente cada semana.	800.000
1999	9 mill	2.2 Linux se afianza como segundo sistema operativo en Internet, en número de servidores.	-
2001	15 mill.	2.4 Última versión con más controladores (*drivers*) y prestaciones.	-

Tabla 2. 1 Evolución del S.O. Linux con respecto al número de usuarios, versiones y líneas de código.

2.2. LINUX

Linux es un sistema operativo multiplataforma, basado en UNIX, y desarrollado según el modelo OSS ("*Open Source Software*" o Software de Fuentes Abiertos o públicos) por la comunidad internacional bajo la licencia GPL ("*GNU Public License*").

Linux es un sistema operativo diseñado por cientos de programadores de todo el planeta, aunque el principal responsable del proyecto, como ya hemos mencionado es Linus Torvalds. Su objetivo inicial es propulsar el software de libre distribución junto con su código fuente para que pueda ser modificado por cualquier persona. El hecho de que el sistema operativo incluya su propio código fuente expande enormemente las posibilidades de este sistema. Este método también es aplicado en numerosas ocasiones a los programas que corren en el sistema, lo que hace que podamos encontrar muchos programas útiles totalmente gratuitos y con su código fuente.

2.2.1. ESTRUCTURA INTERNA DEL SISTEMA OPERATIVO LINUX

- Sistema monolítico (hasta antes de la versión 2.2.x del Kernel): el Sistema Operativo (S.O.) es un solo programa.
- Sistema modular
- Sistema estructurado en capas: el S.O. está dividido en procesos o bloques funcionales.
- Comunicaciones dentro del sistema operativo:
  - Modelo cliente servidor: comunicación por mensajes.
  - Modelo biblioteca: llamadas a procedimiento.
- Comunicaciones entre procesos y sistema operativo (llamadas al sistema).
  - Mediante interrupción software.
  - Llamada a procedimiento.

2.2.2. ARCHIVOS Y SISTEMAS DE ARCHIVOS

ARCHIVOS Y TIPOS DE ARCHIVOS

Una característica especial de este sistema operativo e igualmente de UNIX, es que en él todo es un archivo. Aunque algunos sistemas operativos tienen notaciones o referencias especiales para manejar los directorios y dispositivos especiales, Linux los considera como archivos

Linux divide a los archivos en tres tipos, ver tabla 2.2:

TIPO DE ARCHIVO	DESCRIPCIÓN
Archivo Ordinario	Archivos que contienen datos o programas
Archivo de Directorio	Archivos que contienen los nombres de otros archivos (incluyendo otros directorios)
Archivo especial	Archivo que representa un dispositivo de Hardware

Tabla 2. 2 Tipos de archivos del S.O. Linux.

En Linux también existen archivos ocultos, conocidos también como archivos de punto, porque llevan antes del nombre un punto, estos son usados por las aplicaciones para almacenar información de configuración específica del usuario. Esto evita que la información sea eliminada o modificada accidentalmente.

El nombre de un archivo en Linux, no puede incluir espacios, ni caracteres que representen un separador de campo; no debe incluir caracteres de significado especial para el *shell* ni " / " ya que éste se utiliza para indicar nombres de rutas de acceso; se permiten hasta 255 caracteres en el nombre siendo Linux sensible a mayúsculas y minúsculas.

TÉRMINO SISTEMA DE ARCHIVOS

Un sistema de archivos es una estructura de directorios que se utiliza para organizar y almacenar archivos. El término sistema de archivos se emplea de diferentes formas:

a) Para describir un tipo particular de sistema de archivos: basado en disco, basado en red, o virtual.
b) Para describir un árbol de archivos completo, desde el directorio raíz hacia abajo. Para describir la estructura de datos de una parte del disco o de otro medio de almacenamiento.
c) Para describir una porción de una estructura de árbol de archivos que se adjunta a un punto de montaje en el árbol de archivos principal, para que sea accesible.

La administración del sistema de archivos es una de las tareas más importantes de administración del sistema, ya que los sistemas de archivos gestionan toda la información del sistema, incluida la que utiliza el propio sistema operativo para su operación. Todos nuestros programas, librerías, archivos del sistema y archivos de usuario de Linux están organizados en sistemas de archivos.

EL SISTEMA DE ARCHIVOS

El sistema de archivos que vemos los usuarios de Linux está basado en una estructura en árbol, con la raíz en la parte superior. En la figura 2.1 podemos ver un ejemplo simplificado de esta estructura. Aunque nos parezca ver una estructura simple formada por directorios y archivos, puede ocurrir que muchos de los archivos o directorios de este árbol tengan una ubicación remota, es decir, estén localizados físicamente en distintas particiones de un disco, en discos distintos, o incluso, en distintas máquinas. Veremos cómo estos directorios remotos se asociarán a este árbol de archivos mediante un punto de montaje, para empezar a formar parte de él, y configurar el sistema de archivos que ven los usuarios.

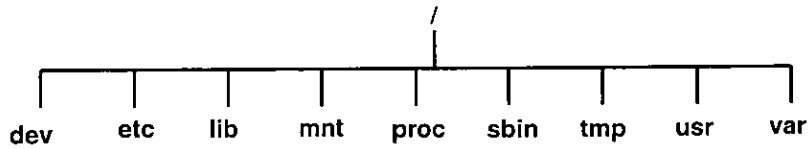


Figura 2. 1 Ejemplo de un árbol de directorios de Linux.

El árbol completo de directorios y archivos que tengamos dependerá de cómo se haya hecho la instalación. Por lo general, la mayor parte del sistema operativo reside en dos sistemas de archivos: el sistema de archivos raíz (/), y el sistema de archivos montado bajo */usr*.

El directorio */bin* contiene programas ejecutables conocidos como binarios. Estos programas son archivos de sistema esenciales para operar con Linux. Muchos comandos clásicos, como *ls,* son programas ubicados en este directorio.

El directorio */sbin* también se utiliza para guardar archivos binarios de sistema, pero más orientados a la administración del sistema.

El directorio */etc* contiene muchos de los archivos de configuración del sistema. Estos archivos van a modelar el sistema Linux de forma particular. Por ejemplo, el archivo que contiene la lista de los sistemas de archivos que hay

que montar al arrancar (*fstab*) está en este directorio. Además, contiene los importantes *scripts* (guiones) de arranque para Linux, la lista de máquinas con direcciones IP que queremos tener registradas, y más información de configuración.

En el directorio */lib* se encuentran las bibliotecas de funciones compartidas que utilizan los programas cuando se ejecutan.

El directorio */dev* contiene archivos especiales conocidos como archivos de dispositivos, que se utilizan para tener acceso al hardware del sistema. Por ejemplo, el archivo, */dev/mouse*, se utiliza para la lectura de entrada del ratón. Así, la interfaz hacia un dispositivo de hardware parece un archivo. Algunos de los dispositivos usados más frecuentemente en el directorio */dev* son: */dev/console* que se refiere a la consola del sistema; */dev/hd* es la interfaz de dispositivo para las unidades de disco duro IDE; */dev/sd* se refiere a la interfaz de dispositivo para los discos SCSI; etc.

El directorio */proc* es un sistema de archivos virtual. Contiene una lista de los procesos activos y un conjunto de archivos especiales con información sobre la utilización de los recursos por parte de los procesos.

El directorio */tmp* se utiliza para guardar archivos temporales que se crean al ejecutar distintos programas, por eso es conveniente, si se crea gran cantidad de archivos temporales grandes, tener */tmp* como un sistema de archivos aparte, para evitar que el sistema de archivos raíz se llene.

El directorio */home* se utiliza como base para albergar los directorios personales de los usuarios. Casi siempre se configura como un sistema de archivos aparte, para que los usuarios puedan disponer de una gran cantidad de espacio para sus archivos. Incluso si tenemos en nuestro sistema distintos tipos de usuarios podríamos separar varios sistemas de archivos, como */home/admin*, */home/invest*, etc., y que nos permitirá montar cada uno de esos directorios como sistema de archivos distintos, para después crear los directorios personales de los usuarios debajo de ellos

El directorio */var* guarda los archivos que tienden a cambiar de tamaño a lo largo del tiempo. Por ejemplo, */var/spool*, y sus subdirectorios, se utilizan para guardar datos como el correo y las noticias recibidas o puestas en cola para transmisión a otro sitio.

El directorio */usr* y sus subdirectorios tienen también un papel relevante en el sistema. Contiene varios directorios con algunos de los programas más importantes del sistema, como por ejemplo:

- */usr/bin*, que guarda muchos de los programas ejecutables del sistema
- */usr/man*, contiene las diversas páginas del manual para los programas de Linux
- */usr/local*, diseñado para la personalización local del sistema, por lo que gran parte del software local se instala en sus subdirectorios. Lo podemos organizar como: */usr/local/bin* para los binarios, */usr/local/etc* para los archivos de configuración, */usr/local/lib* para las bibliotecas. etc.
- */usr/src*, directorio que contiene el código fuente de los diferentes programas del sistema.
- */usr/lib*, con diversas bibliotecas que usarán los programas.
- */usr/etc*, directorio que contiene gran cantidad de archivos de configuración del sistema.
- */usr/include*, en este directorio se encuentran todos los archivos de inclusión para el compilador C.

ESTRUCTURAS DE DATOS UTILIZADAS POR EL SISTEMA DE ARCHIVOS

La estructura jerárquica de directorios que acabamos de repasar nos presenta una imagen uniforme de los dispositivos de almacenamiento, combinando los conceptos de directorio y archivo. De este modo, los usuarios acceden a la información que necesitan mediante un conjunto de comandos con los que pueden moverse por el árbol de directorios y mediante los cuales pueden manipular archivos, ejecutar aplicaciones, etc.

De forma genérica, el usuario accede a la información mediante los nombres de ruta, esto es, una cadena de caracteres que especifica de forma unívoca la ubicación del archivo en el sistema de archivos con independencia del dispositivo.

Para que el sistema soporte esa interfaz, necesita mantener la referencia entre los bloques de datos que forman los archivos de los usuarios y los nombres de esos archivos. Para el sistema de archivos los dispositivos no son más que un conjunto de bloques de tamaño fijo.

Es aquí donde intervienen los directorios. Como se mencionó, un directorio no es más que un archivo especial del sistema que contiene una entrada por cada archivo contenido en el directorio. Para cada archivo contenido en el directorio se emplean dos campos:

1) la cadena de caracteres que forma el nombre del archivo
2) y un número.

Este número es el identificador de una estructura de datos que contiene información sobre el archivo (tamaño, fechas de creación, modificación y acceso, propietario, derechos de acceso, contador de enlaces, punteros a los bloques de datos, etc.). Dicha estructura de datos se conoce con el nombre de *nodo-i o inodo*. Todos los archivos en el sistema tienen asociado un *inodo*. Debido a los enlaces físicos puede ocurrir que dos entradas distintas de directorio tengan asociado el mismo *inodo* y, por consiguiente se accede a la misma información a través de dos nombres diferentes, lo que permite que no se duplique la información.

Así los sistemas de archivos que son fundamentalmente estructuras de árbol, en Linux, con EXT2, permiten al sistema de archivos formar grafos dirigidos como se muestra en la figura 2.2.

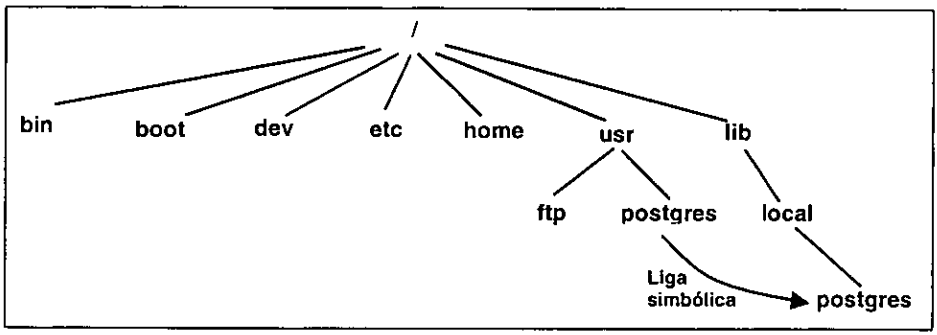


Figura 2. 2 Representación de un grafo dirigido en una estructura de árbol por medio de una liga simbólica.

Esto es posible gracias a los enlaces llamados ligas duras y ligas simbólicas. Una liga dura es un mecanismo por el cual un archivo físico puede poseer uno o más nombres lógicos, cuando un archivo es borrado, el archivo continua existiendo bajo uno de sus otros nombres. Con cada enlace duro creado se incrementa la cuenta de referencia. Cuando se borra un enlace duro, se decrementa la cuenta de referencia, sólo cuando la cuenta llega a cero el archivo puede ser borrado completamente. Un enlace duro no puede existir entre dos archivos de particiones separadas. Esto es debido a que el enlace duro se refiere al archivo original por *inodo*, y el *inodo* de un archivo difiere entre sistemas de archivos.

A diferencia de los enlaces duros, los cuales apuntan a un archivo por su *inodo*, en un enlace simbólico apunta a otro archivo por su nombre. Esto permite que

los enlaces simbólicos apunten a archivos localizados en otras particiones , incluso en otras unidades de red.

Cuando se realiza una operación sobre un archivo (abrir, leer, escribir, etc.) el sistema de archivos debe localizar los bloques de datos sobre los que se desea operar. Para ello, debe examinar la ruta (absoluta o relativa) del nombre del archivo con el objetivo de obtener el número de *inodo*. Al realizar el análisis de las rutas de los archivos, aparecen los dispositivos montados en el sistema de archivos raíz. A partir del *inodo*, teniendo en cuenta el desplazamiento, obtenemos los bloques de datos necesarios para realizar la transferencia.

Además, si el archivo necesita un nuevo bloque de datos, porque se está añadiendo más información, el bloque se obtiene de la lista de bloques libres. El número del nuevo bloque se anota en el *inodo* del archivo al que se asigna.

Hay que señalar que todas las unidades (dispositivos, particiones, etc.) que utilizan el sistema de archivos de Linux disponen de un bloque especial con información sobre la unidad y que se denomina súper bloque.

El sistema operativo suele implementar una estructura de datos en memoria para almacenar los bloques más utilizados de los dispositivos. Esta estructura de datos se conoce con el nombre de *cache de bloques*. Los bloques de disco más utilizados permanecen en memoria - dentro de este cache - para hacer más eficiente el acceso a la información.

Al localizar los bloques de un archivo se debe tener en cuenta la presencia de este cache de bloques. De este modo, una vez que se obtiene el *inodo* del archivo, y después de calcular los bloques que se necesitan, se comprueba si estos bloques están localizados en el cache, para hacer más rápido su acceso.

EL SISTEMA DE ARCHIVOS VIRTUAL (VFS) DE LINUX

Seguramente, pocas veces nos ponemos a pensar cómo es posible unificar conceptos tan dispares como una partición de MS-DOS con un directorio de un servidor de red, de manera que podemos utilizar los mismos comandos y, de la misma forma, en estructuras tan distintas. La razón de esta aparente magia reside en un conjunto de estructuras que el núcleo Linux mantiene y que sirve para unificar de cara al usuario todos y cada uno de los posibles tipos de estructuras de almacenamiento de datos. Esta capa intermedia se denomina sistema de archivos virtual, *VFS (Virtual File System)*.

Linux soporta multitud de tipos de sistemas de archivos, cada uno de ellos sobre diversos soportes físicos. Además, Linux permite programar e insertar dentro del núcleo muchos otros más, incluso los programados por el usuario. Todo ello exige un API unificado y un modelo de programación. El *VFS* proporciona métodos genéricos para las diversas operaciones que se pueden

realizar con *inodos* y archivos, funciones que luego son personalizadas para cada sistema en particular. La filosofía de funcionamiento es la siguiente:

1. Se produce una llamada al sistema que implica un acceso a un sistema de archivos

2. El *VFS* determina a qué sistema de archivos pertenece.

3. Se comprueba si existe ya una entrada en la cache y si dicha entrada es válida.

4. En el caso de que se requiera, se accede a la función del sistema de archivos específico, que realiza dicha función a través de las estructuras del VFS.

5. La función puede, o bien completar la llamada, o requerir una operación de *entrada/salida* sobre un dispositivo.

El *VFS* proporciona también mecanismos de bloqueo y control de accesos. No obstante, estas operaciones dependen de cada sistema de archivos en particular.

El VFS se puede consultar en el archivo */proc/filesystem* para obtener la lista de los sistemas de archivos registrados y que los comprende. Los diversos sistemas de archivos se agrupan como una lista encadenada de estructuras que definen cada sistema de archivos. En la siguiente tabla se ilustra esta lista.

Basado en el medio	**De red**	**Especiales**
ext2 – nativo de Linux	NFS	Procfs - /proc
Usf – BSD	AFS	Umsdos – Unix en DOS
fat – DOS	Coda	Userfs – redirigido al usuario
Vfat – Win 95	Smbfs - LanManager	
Minix	Ncpfs - Novell	
Isofs – CD-ROM		
Sysv – System V Unix		
Hfs – Macintosh		
Affs – NT		
Adfs		

Tabla 2. 3 Organización de la lista de sistema de archivos.

Cuando queramos saber en qué sistema de archivos está ubicado un directorio concreto, podemos usar el comando *df*. La salida muestra el sistema de archivos, así como el espacio libre disponible.

Montaje de los sistemas de archivos

Los sistemas de archivos se pueden adjuntar a la jerarquía de directorios disponibles en el sistema. A este proceso se le llama montaje. Un sistema de archivos montado se adjunta al árbol de directorios del sistema en un punto de montaje especificado (un directorio concreto), y de esta manera, se hace disponible al sistema. El sistema de archivos raíz *(/)* siempre se monta. Cualquier otro puede conectarse o desconectarse del sistema de archivos raíz. Antes de empezar con la operación de montaje necesitamos los siguientes requisitos:

- Ser superusuario
- Que exista ya el punto de montaje en el sistema local
- El nombre del recurso del sistema de archivos que se va a montar (por ejemplo, */usr*)

Además, también hay que determinar los siguientes asuntos:

- Si el sistema de archivos se va a incluir en */etc/fstab* (tabla de sistemas de archivos), para que sea montado cada vez que se levanta el sistema.
- Si el sistema de archivos se puede montar de forma apropiada utilizando un auto- montador.
- Si el sistema de archivos sólo se utilizará temporalmente.
- Si el sistema de archivos se puede montar desde la línea de comandos, usando el comando mount.

El archivo /etc/fstab

El */etc/fstab* es un archivo de configuración que usa *mount*. Este archivo contiene una lista de todas las particiones conocidas del sistema. Durante el proceso de arranque se lee esta lista y sus elementos se montan de forma automática.

Éste es el formato de las entradas del archivo */etc/fstab*

/dev/dispositivo /directorio/a/montar fstype parámetros fs_freq fs_passno

A continuación presentamos en la tabla 2.4 cada elemento en una entrada del */etc/fstab* .

Entradas del archivo /etc/fstab	Descripción
/dev/dispositivo	La partición que está montada (por ejemplo, /dev/hda3).
/directorio/a/montar	El directorio donde está montada la partición
Fstype	El tipo de sistema de archivos
Parámetros	Los parámetros proporcionados con la opción -o del comando *mount*.
fs_freq	Dice al comando *dump* con cuanta frecuencia se necesita hacer copia de seguridad del sistema de archivos
fs_passno	Llama al programa *fsck* en tiempo de arranque para determinar el orden en el cual se comprueban los sistemas de archivos.

Tabla 2. 4 Elementos de entrada del *letc/fstab* y descripción.

2.2.3. PROCESOS E HILOS

La mayor parte de los sistemas operativos modernos manejan tanto procesos como hilos. Aunque la diferencia precisa entre los dos términos suele variar de una implementación a otra, podemos definir la distinción principal: los procesos representan la ejecución de programas individuales, mientras que los hilos representan contextos de ejecución individuales pero concurrentes de un solo proceso que ejecuta un solo programa.

Dos procesos individuales cualesquiera tienen su propio espacio de direcciones independiente, aún si están usando memoria compartida para compartir parte del contenido (pero no todo) de su memoria virtual. En contraste, dos hilos dentro del mismo proceso comparten el mismo espacio de direcciones (no sólo espacios de direcciones similares: cualquier cambio que un hilo haga a la organización de la memoria virtual será visible de inmediato para los demás hilos del proceso, porque en realidad sólo hay un espacio de direcciones en el que todos se están ejecutando).

Hay varias formas distintas de implementar los hilos. Se puede implementar un hilo en el núcleo del sistema operativo como objeto propiedad de un proceso, o puede ser una entidad totalmente independiente. Los hilos no tienen que implementarse en el núcleo; es posible hacerlo enteramente dentro del código de una aplicación o biblioteca con la ayuda de interrupciones de temporizador suministradas por el núcleo.

El núcleo de Linux maneja de forma sencilla la diferencia entre procesos e hilos: utiliza exactamente la misma representación interna para todos. Un hilo no es más que un proceso nuevo que por casualidad comparte el mismo espacio de direcciones que su padre. La distinción entre un proceso y un hilo se hace sólo cuando se crea un hilo nuevo con la llamada al sistema *done* (hacer). Mientras que *fork* crea un proceso nuevo que tiene su propio contexto totalmente nuevo, done crea un proceso nuevo que tiene identidad propia pero que puede compartir las estructuras de datos de su padre.

Esta distinción se puede hacer porque Linux no guarda todo el contexto de un proceso dentro de la estructura de datos de proceso principal; más bien, guarda el contexto dentro de subcontextos independientes. El contexto de sistema de archivos, la tabla de descriptores de archivo, la tabla de manejadores de señales y el contexto de memoria virtual de un proceso se guardan en estructuras de datos distintas. La estructura de datos del proceso simplemente contiene punteros a esas otras estructuras, con lo que cualquier cantidad de procesos puede compartir fácilmente uno de estos subcontextos apuntando a los mismos subcontextos según sea apropiado.

La llamada al sistema *done* acepta un argumento que le dice cuáles subcontextos debe copiar y cuáles debe compartir al crear el proceso nuevo. Éste siempre recibe una identidad nueva y un nuevo contexto de planificación, pero según los argumentos que se pasen podría crear nuevas estructuras de datos de subcontextos con valores iniciales que son una copia de los del padre, o configurar el nuevo proceso de modo que use la misma estructura de datos de subcontexto que el padre está usando. La llamada al sistema *fork* no es más que un caso especial de *done* que copia todos los subcontextos y no comparte ninguno. El uso de *done* confiere a una aplicación un control fino sobre qué exactamente comparten dos hilos.

Los grupos de trabajo *POSIX* han definido una interfaz de programación, y la han especificado en la norma *POSIX.1c*, para que las aplicaciones puedan ejecutar múltiples hilos. Las bibliotecas de sistema de Linux manejan dos mecanismos distintos que implementan esta norma única de diferentes maneras. Una aplicación puede optar por emplear el paquete de hilos basado en el modo de usuario de Linux o bien el basado en el núcleo. La biblioteca de hilos en modo de usuario evita el gasto extra de planificación del núcleo y llamadas al sistema del núcleo cuando los hilos interactúan, pero está limitado por el hecho de que todos los hilos se ejecutan en un solo proceso. La biblioteca de hilos apoyada por el núcleo usa la llamada al sistema *done* para implementar la misma interfaz de programación, pero, dado que se crean múltiples contextos de planificación, tiene la ventaja de que permite a una aplicación ejecutar hilos en múltiples procesadores a la vez en un sistema multiprocesador. Además, esta biblioteca permite que varios hilos estén ejecutando llamadas al sistema del núcleo simultáneamente.

Programas y procesos

Los programas son archivos con código ejecutable que se encuentra almacenado en el disco. Cuando ejecutamos este código, se lleva una copia de él a memoria y se pone en ejecución. A este código en memoria que se está ejecutando se le llama proceso. Un proceso se ejecuta en el procesador con unas características propias y con autonomía respecto al resto de procesos en memoria. Existe un proceso del núcleo llamado *sched* que se encarga de repartir el tiempo de procesador entre los distintos procesos que se ejecutan en memoria, recordemos que Linux es un sistema operativo multitarea. Existe un comando llamado *ps* que nos presenta información de los procesos que se encuentran en ejecución, como el propietario, el tiempo que lleva en ejecución o un código numérico de identificación que le asigna el núcleo del sistema operativo cuando lo pone en ejecución, llamado *PID (Identificador de Proceso).*

Una vez se está ejecutando un comando, el Intérprete de comandos no aceptará más órdenes y el usuario deberá esperar a que esa orden concluya para poder realizar otra nueva operación.

Procesos demonio de Linux

Linux lanza a memoria una serie de hilos de ejecución que son parte de su núcleo y que son despertados por el sistema con cierta periodicidad, denominados demonios o *daemons*. Estos procesos se encargan de tareas muy específicas como atender a una nueva conexión, gestionar las páginas de memoria o distribuir la CPU entre los distintos procesos que compiten por ella. Estos procesos permanecen dormidos, por lo que generalmente no los veremos con una orden *ps*, pero si están despiertos, sí será posible con el comando *ps -e.*

En la tabla 2.5 se muestran algunos de los principales procesos de Linux.

Nombre	Descripción	PID
Sched	Es el encargado de gestionarlos procesos en memoria para su ejecución, se puede decir que es el gerente de la CPU. Puede desalojar procesos completos de memoria en caso de ser necesario y cargar otros.	0
INIT	Lanza procesos de inicio en las conexiones. Posee un fichero donde se va colocando la información de ejecución de todos los procesos llamado /etc/inittab, en él se determinan las características de ejecución de los procesos.	1
kswapd	Gestiona el espacio de intercambio de memoria, procurando que el número de páginas de memoria libres no sea muy bajo, en cuyo caso desaloja información de memoria a área de intercambio *swap*, que suelen ser las menos recientemente usadas. Si no consigue espacio libre rápidamente, llama al proceso *sched* para que intervenga. Este proceso es un hilo del núcleo lanzado por el demonio init.	2
fsflush	Se encarga de actualizar los *buffers* a disco para la entrada-salida con buffer.	3
kundaemon	Recupera páginas de memoria que hay desechadas.	-
cron	Es parte del sistema de Linux que se encarga de ejecutar tareas periódicas del sistema	-
timon	Monitoriza los procesos tty que gestionan los terminales asociados al sistema.	-

Tabla 2. 5 Principales procesos de Linux.

2.2.4. CARACTERÍSTICAS DE LINUX

Linux es un sistema operativo de estilo UNIX, con una diversidad de características como por ejemplo multitarea, multiusuario, con planificación expulsiva, multiplataforma, con protección de memoria entre procesos, memoria virtual, capacidad de multiproceso simétrico, capacidad de funcionamiento en "*Cluster*", soporte de múltiples sistemas de archivos, de múltiples protocolos de red, sofisticado manejo de subsistemas de almacenamiento, entre otras. Estudiaremos algunas características que se consideran importantes.

MULTITAREA

- Linux es un sistema operativo clónico UNIX y del mismo modo permite la multitarea. Con Linux podremos estar ejecutando varios programas (realmente procesos) que compartirán el tiempo de procesador dando la sensación de que se ejecutan simultáneamente. La multitarea en Linux es además muy robusta y no produce errores en las ejecuciones debido a la planificación de las mismas.

  La palabra multitarea describe la capacidad de ejecutar muchos programas al mismo tiempo sin detener la ejecución de cada aplicación. Se le denomina multitarea prioritaria cuando cada programa tiene garantizada la oportunidad de ejecutarse, y se ejecuta hasta que el sistema operativo da prioridad a otro programa para que se ejecute. Este tipo de multitarea es exactamente lo que hace Linux. A diferencia de Linux, otros sistemas operativos no admiten la multitarea prioritaria; admiten una forma de multitarea denominada cooperativa. Con ésta, los programas se ejecutan hasta que permiten voluntariamente que se ejecuten otros programas o no tienen nada que hacer por el momento.

  Para comprender mejor la capacidad multitarea de Linux, se puede examinar bajo otro punto de vista. El microprocesador de la computadora sólo puede hacer una cosa a la vez, pero es capaz de realizar esas tareas individuales en períodos tan cortos que se escapan a nuestra comprensión. Por ejemplo, los microprocesadores habituales en la actualidad funcionan a velocidades de *500* a *1500 MHZ* (*Megahertz*) o más. Esto significa que son capaces de transferir de 500 a 1500, o más, millones de bits por segundo; al procesar un conjunto de instrucciones completo, las velocidades son mucho mayores, generalmente *30 nanosegundos* (billonésimas de segundo). La mente humana no puede detectar la diferencia entre un lapso tan corto y algo que se produzca simultáneamente. En resumen, parece que las tareas se están realizando al mismo tiempo.

Es fácil ver las ventajas de disponer de posibilidades de multitarea prioritaria. Además de reducir el tiempo muerto (tiempo en el que no puede seguir trabajando en una aplicación porque un proceso aún no ha finalizado), la flexibilidad de no tener que cerrar las ventanas de las aplicaciones antes de abrir y trabajar en otras es infinitamente mucho más cómoda.

Linux y otros sistemas operativos de multitarea prioritaria consiguen el proceso de prioridad supervisando los procesos que esperan para ejecutarse, así como los que se están ejecutando. El sistema programa entonces cada proceso para que disponga de las mismas oportunidades de acceso al microprocesador. El resultado es que las aplicaciones abiertas parecen estar ejecutándose al mismo tiempo (en realidad, hay una demora de billonésimas de segundo entre el momento en que el procesador ejecuta una serie de instrucciones de una aplicación y el momento programado por Linux para volver a dedicar tiempo a dicho proceso).

MULTIUSUARIO

- Linux es un sistema operativo multiusuario con capacidad de simular multiprocesamiento y procesamiento no interactivo. Debido a que es un sistema multitarea, permite que varios usuarios accedan a la computadora y ejecuten programas que compartirá la CPU. Además la ejecución de los procesos de cada usuario, su memoria, archivos etc.. estarán protegidos de modo que cada usuario pueda decidir quién accede a sus recursos.

  La idea de que varios usuarios pudieran acceder a las aplicaciones o a la capacidad de proceso de una única PC era una utopía hace relativamente pocos años. Linux ayudó a convertir ese sueño en realidad. El mismo diseño que permite multitarea, ha derivado en la posibilidad de permitir a varios usuarios el acceso a una misma computadora al mismo tiempo, ejecutando cada uno de ellos una o más aplicaciones, todo esto de manera completamente transparente al usuario de manera que cada usuario crea que él o ella, es la única persona que trabajan en la computadora. La característica más remarcable de Linux y sus funciones de multiusuario y multitarea es que más de una persona puede trabajar con la misma versión de la misma aplicación al mismo tiempo, desde el mismo terminal o desde terminales distintos. No se debe confundir con el hecho de que muchos usuarios puedan actualizar el mismo archivo simultáneamente, característica que es potencialmente confusa, potencialmente peligrosa y decididamente indeseable.

Aunque puede que esta característica no sea muy útil en casa, en un entorno de empresa o universitario permite que muchas personas accedan a los mismos recursos al mismo tiempo, sin duplicar la inversión en máquinas costosas. Incluso desde un domicilio particular se encontrará muy útil la posibilidad de entrar en cuentas separadas en lo que se denominan terminales virtuales. También desde una casa se puede ofrecer su propio servicio *on-line* utilizando Linux y varios módems.

SHELLS PROGRAMABLES

- Un *shell* conecta las ordenes de un usuario con el Kernel de Linux (el núcleo del sistema), y al ser programables se puede modificar para adaptarlo a tus necesidades. Por ejemplo, es muy útil para realizar procesos en segundo plano.

  En los años 1960, cuando Dennis Ritchie y Ken Thompson estaban diseñando Unix en AT&T, quisieron crear una forma en la cual los humanos podrían comunicarse con el sistema. Ya en aquella época las computadoras tenían intérpretes de comandos, que podían tomar instrucciones del usuario y ejecutarlas. Pero Ritchie y Thompson querían algo más. Así nació el *Bourne Shell* (o simplemente *sh*). Desde la creación del *Bourne Shell*, muchos más han sido desarrollados, como el *C Shell (csh)* o el *Korn Shell (ksh)*, y cuando la *Free Software Foundation* decidió crear un *shell* libre, los programadores se inspiraron del *Bourne Shell* y de ciertas funcionalidades de otros. El resultado fue el *Bourne Again Shell* o *bash*. El *shell* por defecto en Linux es *bash*, aunque también hay otros disponibles.

- Un *shell* programable es una característica que posee todo sistema Unix y, en consecuencia, Linux; y hace que Unix, y por tanto Linux, sea lo que es; el sistema operativo más flexible de los existentes. El *shell* es básicamente el indicador del sistema y desde ahí se realiza la dirección y la planeación de un sistema.

- La programación del *shell* que Linux ofrece puede ser usada para una variedad de funciones que al usuario se le ocurra. Muchos utilizan esta característica para personalizar su sistema y hacerlo más amigable. Otros lo encuentran muy útil para simplificar muchas de las aplicaciones que se ejecutan, permitiéndoles realizar una serie de procesos en segundo plano para poder trabajar en otros.

INDEPENDENCIA DE DISPOSITIVOS

- Linux admite cualquier tipo de dispositivo (módems, impresoras) gracias a que una vez instalado uno nuevo, se añade al Kernel el enlace o controlador necesario con el dispositivo, haciendo que el Kernel y el enlace se fusionen. Linux posee una gran adaptabilidad y no se encuentra limitado como otros sistemas operativos.

  A simple vista, quizá no parezca importante que los periféricos de un sistema de cómputo puedan funcionar en forma autónoma o independiente. Sin embargo, desde el punto de vista del entorno Linux multiusuario, se convierte en un factor decisivo en un lugar de trabajo productivo.

- Linux evita los problemas de agregar nuevos dispositivos contemplando cada periférico como un archivo aparte. A medida que se necesitan nuevos dispositivos, el administrador del sistema añade al Kernel el enlace necesario. Este enlace, también denominado controlador de dispositivo (*driver*), garantiza que el Kernel y el dispositivo se fusionen del mismo modo cada vez que se solicita el servicio del dispositivo.

  A medida que se van desarrollando y facilitando periféricos mejores para el usuario, el sistema operativo Linux permite un acceso inmediato y sin limitaciones a sus servicios, una vez que se han enlazado los dispositivos con el Kernel. La clave de independencia de los dispositivos estriba en la adaptabilidad del Kernel. Otros sistemas operativos sólo permiten un determinado número de un tipo determinado de dispositivo. Linux puede admitir cualquier cantidad de dispositivos de cualquier tipo, porque cada dispositivo se contempla de forma independiente a través de su enlace individual con el Kernel.

MULTIPLATAFORMA

- El primer Kernel de Linux fue programado en 80386,y las plataformas en las que en un principio se pudo utilizar Linux son *386, 486. Pentium, Pentium Pro, Pentium II, AMD Kx, Amiga y Atari*, sin embargo debido a su buen funcionamiento, Linux ha sido portado a una multitud de plataformas entre las que destacan *Motorola 68000, Digital Alpha, MIPS, Motorola/IBM PowerPC, ARM, Sun SPARC* y *SPARC64*, lo que hace que el Kernel de Linux pueda ser usado en múltiples arquitecturas. Existen versiones especiales para otras arquitecturas, no integradas en el Kernel "oficial".

2.2.5. OTRAS CARACTERÍSTICAS

ARQUITECTURA

- Soporta optimizaciones para los procesadores *386, 486/Cx486, 586 / K5 / 5x86 / 6x86 / 6x86MX, Pentium, PentiumPro/Pentium II, Pentium III, K6/K6-II/K6-III, Athlon* y *Crusoe.*

- Emulación de 387 en el núcleo, de tal forma que los programas no tengan que hacer su propia emulación matemática, cualquier máquina que ejecute Linux parecerá dotada de coprocesador matemático. Por supuesto, si el ordenador ya tiene una *FPU* (unidad de punto flotante), ésta será usada en lugar de la emulación, pudiendo incluso compilar su propio Kernel sin la emulación matemática y conseguir un pequeño ahorro de memoria.

- Funciona totalmente en modo protegido 386, y ejecuta exclusivamente código de 32 bits, incluso existen versiones de 64 bits para los procesadores de las PCs del "futuro", utiliza todo el poder de los procesadores 386 y superiores.

- Cuenta con soporte para sistemas con más de un procesador, disponible para *Sparc* e *Intel.*

- El límite al número de procesos en ejecución lo presenta la cantidad de memoria RAM de nuestro equipo y se puede modificar en tiempo de ejecución.

- El sistema es capaz de aprovechar placas base con varios procesadores Pentium Pro o Pentium II, según el modelo *SMP* de multiproceso simétrico de Intel, dirigiendo los procesos al procesador menos ocupado en cada momento. También existe un uso muy extendido de Linux en "Clusters" de estaciones de trabajo y PC´s, dedicadas exclusivamente o no, formando verdaderos supercomputadores paralelos a bajo costo.

- Bibliotecas compartidas de carga dinámica (*DLL's*) y bibliotecas estáticas.

- Se realizan volcados de estado (*core dumps*) para posibilitar los análisis post-mortem, permitiendo el uso de depuradores sobre los programas no sólo en ejecución sino también tras abortar éstos por cualquier motivo.

- Se puede actualizar el sistema operativo (el mismo Kernel) sin necesidad de reinicializar. Esto significa que se ahorran muchas horas/hombre.

- Pseudo-terminales (*pty's*).

- Terminales virtuales múltiples: varias sesiones de *login* a través de la terminal entre las que se puede cambiar con las combinaciones adecuadas de teclas (totalmente independiente del hardware de vídeo). Se crean dinámicamente y puede tener hasta 64.

- Control de tareas POSIX.

BASADO EN NORMAS Y ESTÁNDARES

- Desde sus inicios, Linus Torvalds toma los *estándares POSIX* para la escritura de este sistema operativo, de modo que sabemos que Linux ha seguido y sigue unos estándares para su escritura y funcionamiento

- Compatible con *POSIX (IEEE), System V* y *BSD* a nivel fuente.

- Emulación de *iBCS2*, casi completamente compatible con *SCO, SVR3* y *SVR4* a nivel binario.

- Soporte de lenguajes nativos, teclados nacionales, etc.

GRATUITO

- Linux es *completamente gratis* y puede ser bajado de Internet, además la gran mayoría de aplicaciones que corren sobre él también lo son, de modo que Linux constituye una alternativa muy económica para cualquier proyecto.

SEGURIDAD

El modelo de seguridad de Linux está íntimamente relacionado con los mecanismos de seguridad típicos de UNIX. Los problemas de seguridad se pueden clasificar en dos grupos:

- Validación. Asegurarse de que nadie pueda tener acceso al sistema sin antes de mostrar que tiene derecho de entrar.

- Control de acceso. Proporcionar un mecanismo para verificar si un usuario tiene derecho de acceder a un objeto dado, y evitar el acceso a objetos si es necesario.

Validación

La validación en UNIX tradicionalmente se ha realizado con la ayuda de un archivo de contraseñas que el público puede leer. La contraseña de un usuario se combina con un valor o "semilla" aleatorio y el resultado se codifica con una función de transformación unidireccional y se almacena en el archivo de contraseñas. El uso de la función unidireccional implica que no es posible deducir la contraseña original a partir del archivo de contraseñas, como no sea por prueba y error. Cuando un usuario presenta una contraseña al sistema, ésta se recombina con el valor de la "semilla" almacenado en el archivo de contraseñas y se le aplica la misma transformación unidireccional. Si el resultado coincide con el contenido del archivo de contraseñas, la contraseña se acepta.

Control de acceso

El control de acceso en los sistemas UNIX, y también en Linux, se realiza con la ayuda de identificadores numéricos únicos. Un identificador de usuario *(uid)* identifica un solo usuario o un solo conjunto de derechos de acceso. Un identificador de grupo *(gid)* es un identificador adicional que puede servir para identificar derechos pertenecientes a más de un usuario.

El control de acceso se aplica a diversos objetos del sistema. Cada archivo disponible en el sistema está protegido por los mecanismos de control de acceso estándar. Además, otros objetos compartidos, como secciones de memoria compartida y semáforos, utilizan el mismo sistema de acceso.

Todo objeto de un sistema UNIX que está sujeto a control de acceso por usuario y grupo tiene asociado un solo *uid* y un solo *gid*. Los procesos de usuario también tienen un solo *uid*, pero podrían tener más de un *gid*. Si el *uid* de un proceso coincide con el de un objeto, el proceso tiene derechos de *usuario* o de *propietario* sobre ese objeto; en caso contrario, si cualquiera de los *gid* del proceso coincide con el gid del objeto, se confieren derechos de *grupo;* en caso contrario, el proceso tiene derechos de *mundo* sobre el objeto.

Linux efectúa el control de acceso asignando a los objetos una *máscara de protección,* que especifica cuáles modos de acceso - lectura, escritura o ejecución - habrán de otorgarse a procesos con acceso de propietario,

acceso a grupos o al mundo. Así, el dueño de un objeto podría tener pleno acceso para leer, escribir y ejecutar un archivo; otros usuarios de cierto grupo podrían tener derecho de lectura pero no de escritura; y el resto de los usuarios podría no tener ningún acceso.

La única excepción es el *uid* privilegiado *root*. Un proceso con este *uid* especial tiene acceso automático a cualquier objeto del sistema, pasando por alto las verificaciones de acceso normales. A tales procesos se otorga también permiso para realizar operaciones privilegiadas como leer cualquier región de la memoria física o abrir *sockets* de red reservados. Este mecanismo permite al núcleo impedir a los usuarios normales el acceso a estos recursos: la mayor parte de los recursos internos clave del núcleo son propiedad implícita de este *uid* raíz.

Linux implementa el mecanismo *setuid* estándar de UNIX. Este mecanismo permite a un programa ejecutarse con privilegios distintos de los que tiene el usuario que ejecuta el programa: por ejemplo, el programa *lpr* (que coloca un trabajo en una cola de impresión) tiene acceso a las colas de impresión del sistema incluso si el usuario que ejecuta ese programa no lo tiene. La implementación de *setuid* en UNIX distingue entre el *uid real* y el *uid efectivo* de un Proceso: el *uid* real es el del usuario que ejecuta el programa, pero el *uid* efectivo es el del propietario del archivo.

Algunas otras características de seguridad que presenta Linux son las que a continuación se mencionan :

- Linux proporciona Seguridad. Linux no solo proporciona el sistema de protección entre procesos y entre directorios, también provee métodos de protección de red como *firewalls*, sistemas de encriptación de información y más.
- Protección de la memoria entre procesos, de manera que uno de ellos no pueda colgar el sistema. Esto significa que si se presenta un "error" o algo se "cae", no es necesario reiniciar todo el sistema, bastará con "matar" y reiniciar la aplicación, programa o servicio. Linux no presenta las pantallas conocidas como "pantallas azules de la muerte" en Windows, que le indican que algo salió mal y que debe reiniciar su equipo.
- Los métodos de seguridad de Linux son mejores que los de los otros sistemas operativos, tanto de forma local como remota, por lo que es menos probable que sea víctima de un "*Cracker*" o que se filtre información fuera de su computadora sin su autorización. En Linux el acceso a los directorios y archivos, así como la capacidad de borrar o modificar éstos, depende de los permisos de usuario que éstos tengan.

- No hay virus que realmente representen una amenaza para Linux.
- Linux le permitirá, utilizando las herramientas apropiadas, controlar el acceso a los puertos y bloquear intrusiones.

ESTABILIDAD

- Linux es estable. *Es* un sistema operativo en el que no se dan habitualmente cuelgues generales del sistema por causa del mismo, (como se mencionó en la parte de seguridad). Aísla los procesos y su ejecución de manera que no alteren a la ejecución y funcionamiento del sistema operativo. Linux es indicado para ser usado como servidor por que es un sistema con escasas caídas. Desaparece en Linux la idea tan conocida en otros sistemas operativos de la necesidad de reiniciar el servidor después de un tiempo de su funcionamiento.

ALTA PORTABILIDAD E INTEROPERABILIDAD

- Linux es muy rápido en cuanto al uso de sus recursos, mucho más rápido en comparación que otros sistemas del mercado.
- La característica de Linux de utilizar un sistema de archivos virtual le permite trabajar con diversos sistemas de archivos sin que varíe su comportamiento. Entre los sistemas de archivos soportados se encuentran: *FAT16* y *FAT32*, de DOS y Windows; *NTFS*, de Windows NT; Minixfs, de MINIX; *ExtFS* y *Ext2FS*, propios; y algunos más.
- Linux es absolutamente compatible con Unix ya que cualquier programa escrito para *SCO, Solaris, HP-UX, Unixware*, puede ser compilado en Linux, además de encontrarnos con ambientes totalmente idénticos a cualquier Unix comercial
- El Kernel de Linux proporciona a cada proceso una visión de la máquina como propia y exclusiva. Un proceso de usuario no puede, ni siquiera intencionadamente, interferir con los otros ni acceder a memoria fuera de su propio espacio de direcciones. El tiempo se reparte de acuerdo a un esquema de prioridades entre los diferentes procesos.

MANEJO DE MEMORIA

- Linux proporciona memoria virtual a los procesos, que ven un espacio de direccionamiento continuo de 3Gbytes. Para ello utiliza paginación por demanda, almacenando en disco, bien sea en una o más particiones, o en uno o varios archivos de "*swap*", páginas de memoria central.
- La memoria virtual usada en este tipo de paginación (sin intercambio de procesos completos) a disco, a una partición o un archivo en el sistema de archivos, o ambos, tiene la posibilidad de añadir más áreas de intercambio sobre la marcha. Un total de 32 zonas de intercambio de 128MB de tamaño máximo pueden ser usadas en un momento dado con un límite teórico de 4Gb para intercambio. Este límite se puede aumentar fácilmente con el cambio de unas cuantas líneas en el código fuente
- La memoria se gestiona como un recurso unificado para los programas de usuario y para el caché de disco, de tal forma que toda la memoria libre puede ser usada para caché y ésta puede a su vez ser reducida cuando se ejecuten grandes programas.
- Varios procesos pueden usar la misma zona de memoria para ejecutarse. Cuando alguno intenta escribir en esa memoria, la página (4Kb de memoria) se copia a otro lugar. Esta política de copia en escritura tiene dos beneficios: aumenta la velocidad y reduce el uso de memoria.
- Linux sólo lee del disco aquellas partes de un programa que están siendo usadas actualmente

SOPORTE DE HARDWARE

- Soporta arquitecturas *ISA, ISA PnP, EISA, VLB, PCMCIA, PCI* y *USB* en el Kernel.
- Soporta todas las tarjetas *IDE/EIDE* y *SCSI* en el mercado. Se pueden tener hasta 20 dispositivos *IDE* conectados a nuestro equipo.
- Soporta la mayoría de las tarjetas de red conocidas.
- Soporta la mayoría de las tarjetas de sonido y CD-ROMs en el mercado.

CONECTIVIDAD

- Linux incorpora conectividad. Linux ha sido en algunos ámbitos "el sistema operativo de red", esto es debido a su buena disposición en usos como servidor y por las características de *conectividad* que incorpora que le lleva a soportar protocolos como TCP/IP, IPX, Appletalk, SMB (de Windows), NFS de Sun, etc... , que le permite la comunicación con otras plataformas y con multitud de protocolos.

- Cuando se trata de interconectar varias redes, Linux puede trabajar como ruteador ("*router*"). Puede trabajar con tarjetas Ethernet (de 100 y 10 Mbits), *Arcnet* y *Tokenring*. Aparte del enrutamiento, también es posible llevar una administración de la red o redes que se manejen

Todas estas características dependen en un 90% del corazón del sistema operativo, el Kernel.

2.3. EL KERNEL DE LINUX

Los sistemas operativos necesitan un Kernel (núcleo) que facilite las operaciones básicas necesarias para el correcto funcionamiento del sistema.
El Kernel de cualquier sistema operativo es el núcleo de todo el software del sistema. La única cosa más importante que el Kernel es el propio hardware.

El Kernel tiene muchas tareas. Lo esencial de su trabajo es abstraer (separar) al software del hardware que tiene por debajo y proporcionar un entorno para el software de aplicación, ver figura 2.3.
Específicamente, el entorno debe manejar tareas tales como:

- Trabajo en red
- Acceso a disco
- La gestión de la memoria y la ejecución de procesos
- Manejo de periféricos
- Proporcionar una interfaz de comunicación entre los programas y los dispositivos
- y toda una serie de operaciones imprescindibles para conseguir que funcionen todos los mecanismos propios de un sistema operativo como plataforma básica , para la ejecución de programas.

El Kernel es lo primero que se carga cuando se arranca un sistema Linux. Si el Kernel no trabaja bien es improbable que el resto del sistema arranque. Así entonces el Kernel forma la definición pura de Linux.

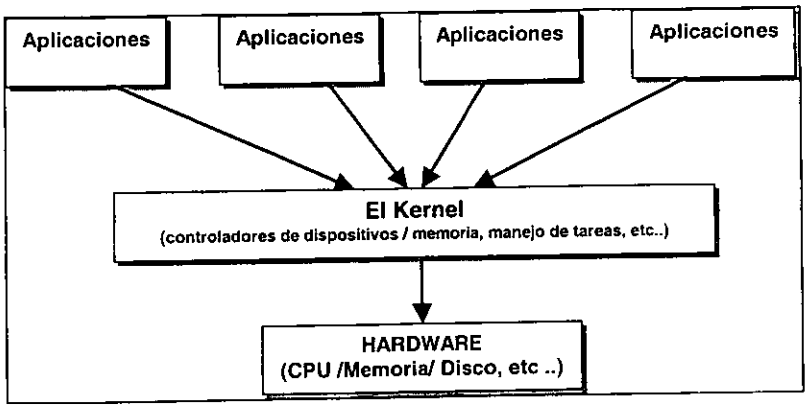


Figura 2. 3 Representación visual de cómo el kernel de Linux se ajusta a un sistema completo.

2.3.1. ADAPTABILIDAD DEL KERNEL

Cuando se instala una distribución de Linux automáticamente se copia un Kernel en el directorio */boot*. Este Kernel será el utilizado en el arranque del sistema por defecto. Inicialmente se puede usar el Kernel copiado automáticamente por la instalación. Sin embargo, puede ser necesaria una recompilación (reconfiguración) del núcleo por diversas razones:

- Añadir al núcleo soporte para cierto hardware instalado en nuestra máquina y para el cual no hay soporte por defecto en el núcleo instalado.

- Eliminar las partes redundantes del núcleo. Es decir, eliminar el código para todas las funcionalidades que son soportadas por el núcleo y que, sin embargo, en nuestra máquina no serán utilizadas. Por ejemplo si no se tiene un subsistema *SCSI*. ¿Para que gastar tanta memoria en soportarlo?

- Actualizar el núcleo del sistema a una nueva versión. Esto permitirá que el Kernel sea capaz de «entenderse» con un mayor número de dispositivos o que pueda ejecutarse más rápidamente que versiones anteriores. Pero normalmente se actualiza el Kernel para obtener soporte para un nuevo hardware y corregir pequeños fallos.

La filosofía de diseño de Linux nos permite decidir acerca de las partes importantes de su Kernel. Para este trabajo sólo necesitamos recordar, para los siguientes capítulos, que se recompilará o reconstruirá el Kernel para poder eliminar partes redundantes del sistema que no se necesitarán o agregar controladores de dispositivos (en este caso principalmente para tarjetas de red) necesarios.

2.3.2. LOS MÓDULOS CARGABLES DEL KERNEL

Originalmente, el núcleo de Linux era del tipo denominado *monolítico*. Esto significa que todas las funcionalidades del núcleo eran enlazadas en un solo archivo ejecutable al ser compilado el Kernel. Esto daba lugar a tener cargado el soporte para dispositivos que sólo se usaban de muy tarde en tarde, pero que sin embargo ocupaban memoria. Además, e incluso más grave, era la necesidad de recompilar y reiniciar el sistema para añadir el controlador para un nuevo dispositivo.

Esto fue remediado en versiones posteriores, y actualmente existen muchos controladores y funcionalidades que pueden ser enlazadas con el archivo

principal del núcleo o cargadas en tiempo de ejecución. En la creación de la configuración de las fuentes del núcleo para su posterior compilación puede seleccionarse el modo que se prefiera, como parte del Kernel o como módulo.

Los módulos son guardados en el directorio */lib/modules/<versión>*, donde versión corresponde a la versión actual del Kernel.

El cargador de módulos del Kernel (kmod)

El uso de los módulos ha sido simplificado a partir de la versión *2.2.x* del Kernel. El «antiguo» *Kernel* ha sido sustituido por el *Kernel module loader*. Este demonio es cargado al inicio del sistema y se encarga de cargar automáticamente los módulos conforme son necesarios por medio de *modprobe*. Para poder hacer uso del *Kernel module loader* hay que habilitarlo en la configuración del núcleo (la opción *(CONFIG_KMOD)*).

Los controladores para acceder a las particiones de arranque, no pueden estar configurados como módulos. Es decir, no es aconsejable configurar como módulos el controlador *SCSI*, el controlador para el sistema de archivos ext2 (sistema de archivos más habitual en las particiones Linux) o el controlador para el *IDE*.

2.3.3. CONFIGURACIÓN DEL KERNEL

La configuración del núcleo consiste en responder a una serie de preguntas sobre cómo se quiere personalizar el núcleo. Ello requiere unos conocimientos avanzados en ciertos apartados, si no se entiende alguna opción simplemente se debe dejar con su valor por defecto.

Antes de configurar o personalizar el Kernel es aconsejable saber con precisión cuáles serán las características que se desean tener. Normalmente esto significa que se necesita una lista de hardware; con esta lista de hardware ya se está listo para proceder a la configuración del Kernel. Esta configuración puede ser realizada de tres formas distintas y se debe estar en la ruta

/usr/src/Linux/ :

1) En la línea de comando: no es muy aconsejable, ya que se tendría que responder a demasiadas preguntas, a menudo sobre cosas cuya configuración no deseamos cambiar. Para realizar la configuración de esta forma se escribe:

 # make config

La pantalla que se obtiene se muestra en la figura 2.4

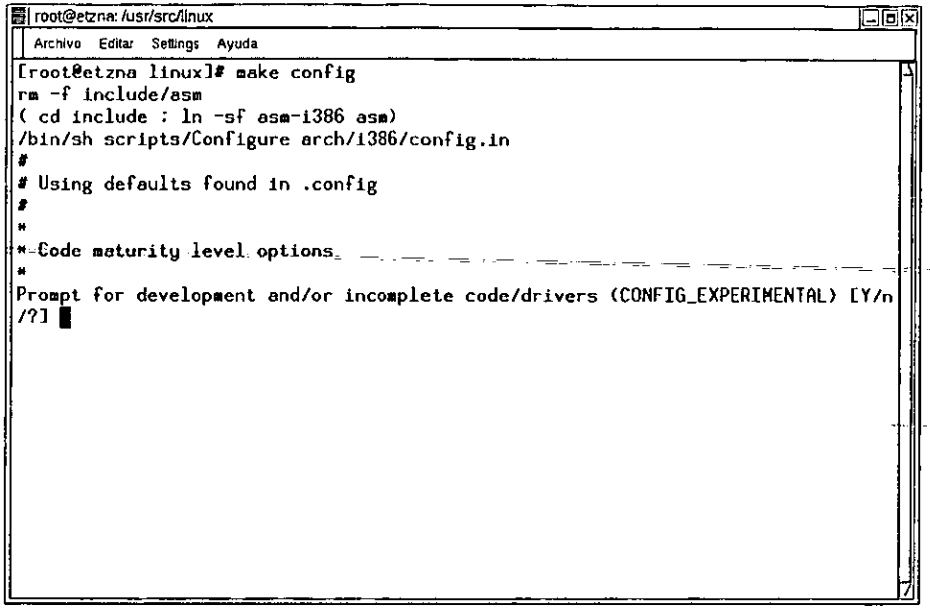


Figura 2. 4 Pantalla de la opción de configuración del kernel (línea de comando).

A continuación, aparecen una serie de preguntas en las que existen dos o tres respuestas. Dichas respuestas pueden ser *'y'* (si), *'n'* (no) o 'm' módulo. Si se pulsa cualquier otra tecla aparece una ayuda breve sobre la opción actual como se muestra en la figura 2.5.

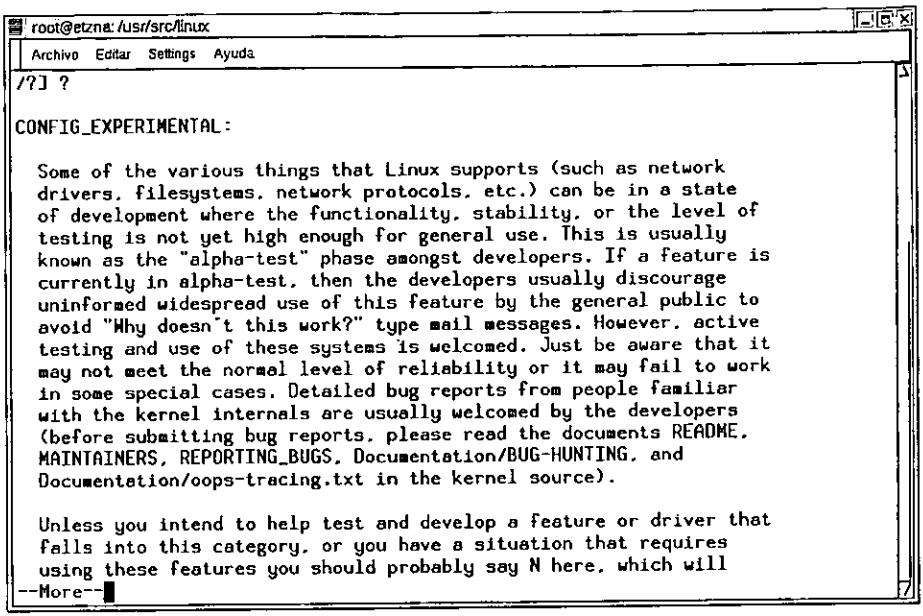


Figura 2. 5 Pantalla de la ayuda en el modo de configuración del kernel (línea de comando).

2) Desde un menú en modo texto: es una opción bastante más cómoda que la anterior, ya que está bastante estructurada por categorías y podremos configurar sólo lo que queramos, ignorando las opciones que no nos interesen. En este caso se debe escribir:

#make menuconfig

En la figura 2.6 se muestra la salida a pantalla el menú de configuración modo texto.

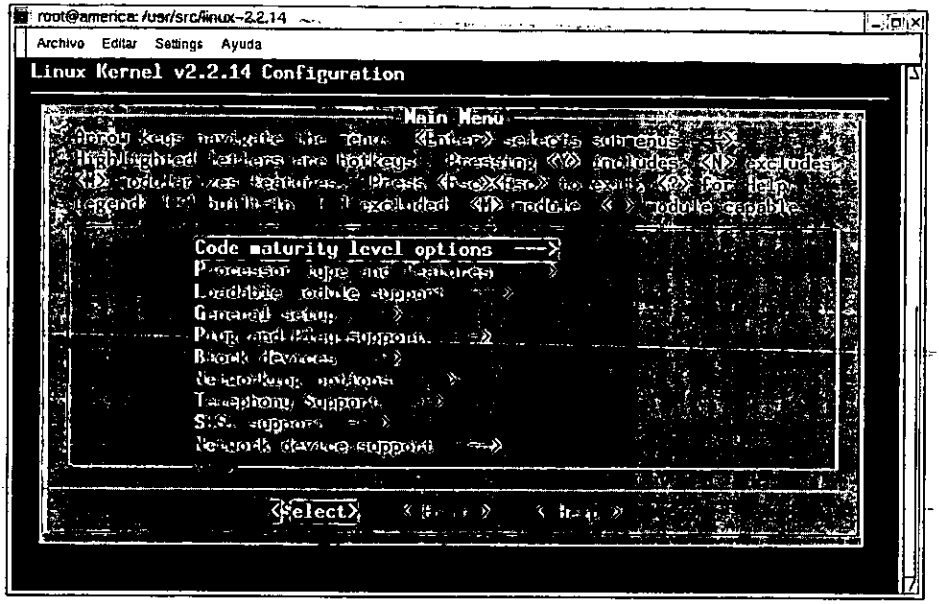


Figura 2. 6 Pantalla modo texto para la configuración del kernel.

3) Desde el entorno X-Windows: Es la opción más cómoda, y la más aconsejable si el sistema que va a configurar tiene instalado X-Windows. Para ello se escribe en la línea de comandos:

 # make xconfig

Y la salida que se obtendrá es la siguiente (ver figura 2.7)

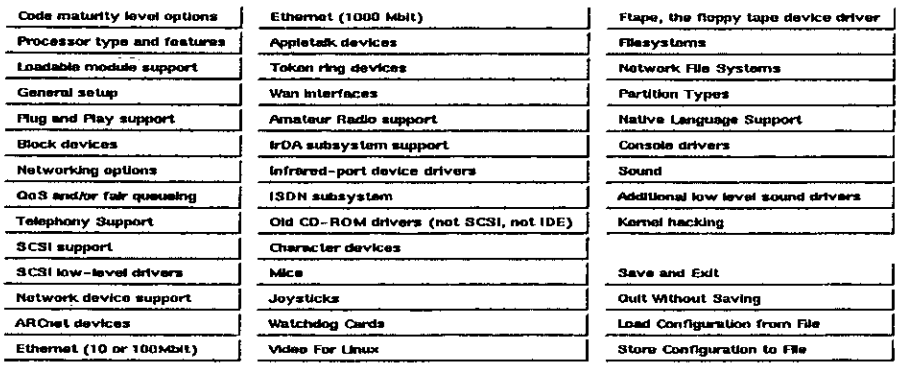


Figura 2. 7 Pantalla de configuración del kernel en el entorno X-Windows.

Naturalmente, no todo se puede elegir para compilar como un módulo. El Kernel debe saber unas pocas cosas antes de poder cargar y descargar módulos, tales como el modo de acceder al disco duro y el manejo de los sistemas de archivos. Si no se está seguro de si una característica en particular puede compilarse como módulo, puede probar cómo funciona o simplemente dejar los valores predeterminados.

A continuación se describen algunas de las opciones del Kernel, que son comunes a las diferentes distribuciones y que se podrían encontrar al momento de configurar el Kernel (Se presentan los términos en ingles y en español para una mejor comprensión):

CODE MATURITY LEVEL OPTIONS; NIVEL DE MADUREZ DE CÓDIGO

Nos lleva a un menú donde se decide si se desea configurar las partes del núcleo que todavía están en fase de pruebas o no. Esta opción la activan normalmente los desarrolladores para probar la estabilidad de los nuevos controladores, pero no es aconsejable que la utilicen los usuarios normales, a no ser que se requiera el soporte de un nuevo dispositivo que se necesite usar. Obviamente, en un servidor de producción no debería ejecutarse código experimental. Por otra parte, si se planea jugar con un sistema de escritorio y se quieren probar nuevas posibilidades, este código experimental podría ofrecerle algunas funciones interesantes y beneficiosas. (En general, incluso el código experimental es lo bastante estable para el trabajo diario. Raramente una característica experimental provoca la caída de un sistema, pero puede ocurrir.)

PROCESSOR TYPE AND FEATURES; TIPO DE PROCESADOR Y CARACTERÍSTICAS

Ésta es otra lista de elementos configurables prácticamente es auto explicativo y sólo se debe responder a una serie de preguntas.

Processor Family. Familia a la que pertenece el procesador.

En está opción se puede seleccionar el tipo de procesador para el que se va a compilar el núcleo.

Disable the PII/PIII serial number at bootup, Deshabilitar el identificador único del procesador

En los últimos procesadores de Intel se tiene un identificador para cada procesador. Con esta opción se habilita o deshabilita.

Enable PII/PIII Extended/Fast FPU save and restore support

Activa el uso de las nuevas características de los procesadores Intel y es imprescindible para el uso de registros *XMM* de 128 bits del *PIII*.

Math emulation

Con esta opción se brinda emulación matemática. (Sólo necesaria para ejecutar el Kernel en sistemas *386, 386SX y 486SX*). Se puede responder que sí *('y')* a esta pregunta, ya que si el sistema tiene coprocesador será usado y se ignorará el emulador de coprocesador.

MTRR (Memory Type Range Registers)

Esta opción permite en gran medida acelerar las operaciones con los buses *PCI/AGP*. Es aconsejable activarla si posee un procesador *Pentium Pro, Pentium II* o superior, aunque también se soportan los *AMD K6-II, K6-III, Cyrix 6x86* y *WinChip C6*.

Simetric multi-processing support

La opción *Simetric multi-processing support* (Soporte de Multiprocesamiento Paralelo, SMP) no es aconsejable activarla a no ser que se disponga de una máquina multiprocesador. Si se responde que sí a esta pregunta y no se tiene una máquina SMP puede que se ejecute correctamente en una máquina uniprocesador (como casi todas las PC), pero en cualquier caso lo hará más lento que si responde no.

Maximun Physical Memory

Es el límite de memoria, donde se puede elegir 1Gb o 2Gb.

LOADABLE MODULE SUPPORT; SOPORTE PARA LOS MÓDULOS CARGABLES

En esta pantalla se puede activar el soporte para los módulos cargables (*Loadable modules*) Cuando se vuelve a recompilar el núcleo también se suelen recompilar todos lo módulos. Sin embargo, existe la posibilidad de que se use un módulo que no ha sido generado a partir de las fuentes del núcleo. Para hacerlo, hay que activar la opción *Set version information on all symbols for modules*. Se debe tener en cuenta que si la utilidad *genksyms* no está dentro del directorio *modutils* la compilación del núcleo fallará. Los módulos pueden ser cargados con *insmod* o *modprobe*. Pero también pueden ser cargados automáticamente por el núcleo si se activa la siguiente opción, *Kernel module loader.* El núcleo cargará los módulos conforme le hagan faltan haciendo uso de la utilidad *modprobe*.

GENERAL SETUP; CONFIGURACIÓN GENERAL

Este menú es crítico en el gran esquema de la configuración del Kernel. Sus decisiones de configuración aquí activarán o desactivarán otros menús, así que se debe tener mucho cuidado. La pantalla de configuración general permite ajustar los parámetros relativos al soporte de red, los tipos de archivos binarios soportados por el núcleo y la gestión de ahorro de energía con *APM (Advanced Power Management).*

Uso de la memoria más allá del primer Gigabyte.

Hoy en día es algo bastante desorbitado, así que se responde no. Se puede ahora decidir si se quiere que haya soporte para red o no. Es muy aconsejable que se responda positivamente a esta pregunta, pues incluso aunque la computadora no esté conectada a otras computadoras a través de una red es necesario para el correcto funcionamiento de un buen número de programas.

Soporte a archivos de tipo binario

No hace demasiado tiempo, los programas compilados se almacenaban en un formato de archivo conocido como *a.out*. Desafortunadamente, este esquema tenía limitaciones graves cuando llamaba a bibliotecas compartidas, las cuales crecieron cuando evolucionó UNIX. El formato *ELF (Executable and Linking Format)* es un formato binario originalmente desarrollado por *USL (UNIX System Laboratories)* y usado en *Solaris* y *System V Release 4*, se creó para superar estas limitaciones. Actualmente Linux usa el formato binario *ELF* para todos sus programas. Es poco probable que necesite soporte *a.out* y, seguramente, se puede decir *No* en esta opción.

El soporte para el formato *ELF*

Es imprescindible, ya que prácticamente todos los ejecutables actuales son distribuidos en este formato ejecutable. Se recomienda responder *'y'* para optimizar la velocidad del sistema. Se puede configurar el sistema de forma que sea posible el registro de distintos formatos ejecutables (*java, python,...*) que serán ejecutados como si se tuviera soporte nativo en el sistema, es decir, simplemente escribiendo su nombre en la línea de comandos y pulsando *ENTER*. También se puede configurar como módulo. En este caso el módulo se llamará *binfmt_misc.o*. Para ello se activa la opción Kernel *support for MISC binaries*. Finalmente, también se puede incluir soporte para la ejecución de programas java. Sin embargo, no funcionará si no tiene instalado el *JRE (Java Runtime Environment)*

Soporte para la ejecución de distintos tipos de formatos de archivos ejecutables

El formato *a.out* ha quedado bastante obsoleto, aunque se puede configurar como modulo. No se aconseja responder *'n'* puesto que todavía se puede encontrar algún programa cuyo ejecutable esté generado en formato *a.out*. El módulo generado en este caso se denomina *binfint_aout.o*.

PCI

Es el método utilizado para la comunicación a través del bus del sistema. Hoy en día prácticamente todas las máquinas personales tienen este tipo de bus. En cuanto al medio de acceso a los dispositivos PCI *(PCI access mode)* es bastante seguro dejar la selección *Any*. De este modo Linux tratará de acceder de forma directa, y si no lo consigue lo hará a través de la *BIOS* del sistema. En caso de que disponga de una BIOS defectuosa, no será capaz de crear una configuración correcta u óptima. Para corregir este problema active la opción *PCI quick*. No hace falta que la active si su *BIOS* funciona perfectamente.

MCA

Es un tipo de bus del sistema como *PCI* o *ISA* que aparece en algunas máquinas *PS/2* de IBM. No lo seleccione a no ser que se disponga de una de estas máquinas.

SGI Visual Workstation Support

Se debera activar esta opcion si se dispone de una *SGI* 320 o 540, pero se debe saber que un núcleo compilado con esta opción no funcionará en otra placa sin los chips de *SGI* y viceversa.

IPC (Inter Process Communication)

Es una librería para la comunicación de distintos programas en ejecución. Si no se activa está opción se encontrará que algunos programas no se ejecutan, entre ellos el emulador de DOS*: dosemu*.

BSD Process Accounting

Permite que los programas de usuario puedan almacenar ciertas estadísticas sobre la ejecución de los programas. La información exacta guardada está descrita en el archivo */usr/src/Linux/include/Linux/acct.h*. Suele ser acertado activar este soporte.

Modificación de parámetros sin reiniciar el sistema o recompilarlo

El núcleo de Linux permite la modificación de ciertos parámetros sin recompilarlo y sin reiniciar el sistema. Esto se realiza a través de *sysctl*. Si se ha activado el sistema de archivos */proc*, donde se describe el sistema, sysctl creará algunas entradas en */proc* que pueden ser modificadas dinámicamente. Esta opción se debe activar, sólo que el sistema ande muy corto de memoria.

Soporte para el puerto paralelo

Es necesario para el uso de impresoras, unidades ZIP, etc. Puede ser configurado como módulo, porque no es algo que se utilice normalmente, constantemente. En las *PC* y *Alpha* se debe activar también *PC-style hadware* para habilitar el uso del tipo de puerto paralelo que se encuentra en este tipo de máquinas. Si se desea usar otro tipo de puertos paralelos hay que responder *'y'* o *'n'* a *Supportforeing hardware*. Esta selección produce pérdida en el rendimiento, por lo que a no ser que sea estrictamente necesario se debeá responder *'n'* .

APM

Permite el ahorro de energía en sistemas cuyo *BIOS* soporte este estándar. Es muy útil en el caso de computadoras portátiles. Está, en la práctica, totalmente deshabilitado en sistema con varias CPU. *Enabled PM at boot time* permite que la máquina pueda entrar en estado suspendido y otras funciones de ahorro de energía cuando se realizan llamadas del tipo «procesador ocioso». Para que Linux ponga el procesador en estado ocioso *(idle)* cuando el sistema no está realizando ninguna tarea, y que el *APM* de la *BIOS* se encargue de controlar el ahorro de energía de los dispositivos, también hay que responder *'y'* a *Make CPU idle calls when idle.*

Apagado Automático

Si se desea que cuando se cierre el sistema se desconecte la computadora automáticamente se debe contestar *'y'* a *Power off on Shutdown*. Para apagar el equipo de forma segura sólo se tendrá que escribir *halt.*

PLUG AND PLAY SUPPORT; SOPORTE PARA CONECTAR Y USAR

El soporte Linux para dispositivos *PnP* (*Plug and Play*, conectar y usar) funciona. Sin embargo, la naturaleza del hardware de las PC lo hace irrealizable. En general, definir lo que se necesita del sistema es un modo de actuar más seguro que confiar en que *PnP* lo encuentre.

Para poder utilizar los dispositivos *Plug and Play* se debe activar el soporte para ellos. El soporte para *Plug and Play* no puede ser en forma de módulo. Esto permitirá al sistema operativo configurar automáticamente un buen número de dispositivos. Existe también un tipo de dispositivos conectados al puerto paralelo que pueden ser identificados por el sistema. Para ello se debe contestar *'y'* a *Auto-probe for parallel devices.*

BLOCK DEVICES; DISPOSITIVOS DE BLOQUE

Los dispositivos de bloques son aquéllos cuya unidad de información es el bloque, y no el carácter como ocurre en las impresoras o las pantallas. Ejemplos de dispositivos de bloques son los controladores de los discos duros o las disqueteras.
Las opciones más comunes que hay al momento de configurar un Kernel son:

Normal PC floppy disk support

Aporta el uso de las disqueteras. Este soporte también puede configurarse como módulo. En caso de que desee configurarlo como módulo recuerde que el módulo generado será denominado *floppy.o.*

Enhanced *IDE*/MFM/RLL disk/cdrom/tape/floppy support

Esta opción permite tener soporte para los dispositivos de tipo *IDE.* No es necesario este módulo si el sistema sólo utiliza dispositivos *SCSI.* Una observación importante al configurar el Kernel es que nunca se configure como módulo un controlador para un dispositivo que inicia el arranque. Si el sistema arranca desde una unidad de disco *IDE* jamás se debe configurar esta opción como módulo. Se Podría configurar como módulo en caso de que el sistema arrancara desde una unidad de disco *SCSI.*

Use old disk-only driver on primary interface

En algunos sistemas antiguos es necesario el uso del viejo controlador para el interfaz primario *IDE.* A no ser que se tenga un sistema realmente antiguo, no se necesita activar.

Soporte para dispositivos *IDE*/ATAPI:

Si se dispone de un CD-ROM que utiliza el protocolo *IDE/ATAPI,* (prácticamente todos los CD-ROM actuales) la opción include *IDE/ATAPI CDROM support* se debe incorporar al Kernel. De igual forma si se dispone de una dispositivo de cinta o una disquetera (*Iomega Zip, LS 120,...*) que utiliza el interfaz *IDE* se

puede marcar *'y'* o *'n'* en las opciones *include IDE/ATAPI TAPE support* e *include IDE/ATAPI FLOPPY support*.

Emulación de soporte *SCSI*

SCSI emulation support permite el uso de un controlador genérico *SCSI* para un dispositivo *IDE* para el que todavía no existe soporte nativo. Esto puede usarse con algunas grabadoras. Para hacer uso de esta opción también es necesario activar el soporte genérico *SCSI* (*Generic SCSIsupport*).

Opciones específicas para los *chipsets*.

- CMD640
  Dentro de las placas 486 y Pentium era muy común encontrar este *chipset*. Este *chipset* presenta una serie de fallos que Linux intenta corregir de forma automática. La opción que brinda soporte y corrección de errores es Active CMD640 *chipset bugfix/support*. Si se tiene este *chipset* hay que activar también *CMD640 enhanced support* para optimizar los accesos con este *chipset*.

- PC- Technologies RZ1000
  Al igual que el caso del *chipset* CMD640, el chipset PC-Technologies RZ1000 también era común en las antiguas placas base (tarjetas madre) para 486 y Pentium y contiene un fallo que puede provocar que se corrompan los datos. Esto se puede evitar respondiendo *'y'* a *RZ1000 chipset bugfix/support*. Se añadirá soporte para dicho *chipset* así como se corregirá el fallo antes citado. La corrección del fallo produce un ligero descenso en el rendimiento del acceso a disco, pero es totalmente fiable.
  Si la placa (tarjeta madre) utiliza un *chipset* distinto, se puede activar tanto *Other chipsets support* como la opción correspondiente al *chipset* que utilice la placa base. Se recomienda consultar las especificaciones de la placa base en caso de no estar seguro. El uso del controlador exacto para el *chipset* de la placa base redunda en una optimización del rendimiento de su sistema, permitiendo a menudo aumentar la velocidad de las transferencias.

Soporte PCI.

La opción *Generic PCI IDE support* ayuda a la detección de los dispositivos *IDE* en sistemas con PCI. La opción *Generic PCI bus-master DMA support* reduce la carga que la CPU tiene que soportar en las transferencias de datos. Es soportado por prácticamente todas las placas base Pentium y superiores.

Controlador de disco externo.

Si se utiliza un controlador de disco externa (*off-board*) en una tarjeta PCI los canales *IDE* adicionales serán asignados a *IDE2* e *IDE3*, mientras que los canales *IDE* del controlador de la placa base (*on-board*) son asignados a *IDE1* e *IDE2*. Para invertir esta situación debe activar *boot off-board chipsets first support*. Hay que tener en cuenta que al hacer esto el nombrado de las unidades de disco cambia.

DMA

Use DMA by default when avaible activa el uso del *DMA* por defecto. En versiones anteriores del núcleo éste era el comportamiento por defecto, pero debido a problemas provocados por fallos en el hardware que han causado problemas, ahora no se utiliza como opción por defecto. El uso del *DMA* acelera el acceso a disco, así que se debería activar a no ser que el hardware presente algún tipo de problema con esta opción.

Uso de archivos como dispositivos de bloque.

En Linux se puede utilizar un archivo normal como si fuera un dispositivo de bloques. De esta forma se puede crear un sistema de archivos sobre el dispositivo de bloques y usarlo igual que cualquier otro sistema de archivos que pueda ser montado. Esto es muy útil para comprobar la corrección de una imagen *ISO9660* creada antes de grabarla sobre un disco virgen. Para poder usar estas funcionalidades debe responder *'y'* a *loopback device support*. Normalmente se responderá *'n'* a esta opción.

Soporte para RAID.

Multiple devices driver support permite combinar distintas particiones de disco en un dispositivo de bloques lógico. Esto puede ser utilizado para obtener los beneficios de la tecnología *RAID*. *RAID* permite, dependiendo del nivel de RAID, obtener tolerancia a fallos en disco. Si en el sistema se tiene soporte *RAID* por hardware no se necesita responder *'y'* a esta opción. La combinación entre las distintas particiones puede ser realizada de distintas maneras. Se puede simplemente añadir una partición tras otra de forma lineal. Para ello se utiliza la opción *Linear (append) mode*.

Los distintos niveles de la tecnología *RAID* determinan diferentes formas de realizar la combinación entre las particiones de disco. *RAID-0* va escribiendo los datos repartiéndolos por la distintas particiones. De este modo se consigue, si las particiones residen en distintas unidades de disco, una aceleración en las transferencias de datos. *RAID-1* por su parte, hace que varias unidades de disco sean copias exactas unas de otras. Esto permite que si una unidad de

disco falla, el sistema siga funcionando normalmente usando las restantes copias que son réplicas exactas de la que ha fallado.

Tanto *RAID-4* como *RAID-5* soportan tolerancia a fallos respecto al fallo en una sola unidad. Esto se realiza manteniendo información redundante (paridad) para poder ser capaz de recuperar información en caso de fallo en una de las unidades. *RAID-4* mantiene toda la información de paridad en una sola unidad, mientras que *RAID-5* la distribuye por distintas unidades.

Discos virtuales en memoria RAM.

RAM disk support permite usar parte de la memoria *RAM* como si fuera un dispositivo de bloques. A partir de ese momento se podrán crear sistemas de archivos y leer o escribir exactamente igual que si se tratara de una unidad de disco real. Es usado habitualmente en el arranque de la instalación de las distintas distribuciones de Linux. Se usa en conjunción con *Initial RAM disk (initrd) support* para permitir que se utilice en el arranque del sistema para cargar ciertos módulos que son necesarios para montar otros sistemas de archivos. Normalmente no se necesitan los discos RAM, así que puede responder *'n'* para reducir el tamaño del núcleo.

Discos XT

Se muestra también la posibilidad de configurar soporte para discos XT. Este soporte sólo es necesario si se tiene un controlador de discos de 8 bits, de la época de los 8086.

Soporte para hardware conectado al puerto paralelo.

Parallel port IDE device support aporta el soporte para el numeroso hardware que se conecta al puerto paralelo pero que realmente son dispositivos *IDE*. Es el denominado *subsistema PARIDE*. Algunos de estos dispositivos son *CD-ROM, CD-R, CD-RW o Iomega ZIP*. Esta opción sólo aporta el soporte genérico para este tipo de dispositivos. También necesita activar el controlador exacto para que funcione el dispositivo en cuestión.

Controlador Compaq Smart Array.

Si se usa una computadora con un controlador *Compaq Smart Array* no olvide responder *'y'* a *Compaq SMART2 support.*

NETWORKING OPTIONS; OPCIONES DE RED

Como mínimo, se deberían habilitar las opciones *UNIX Domain Sockets (Sockets de dominio UNIX)* y *TCP/IP Netwrorking (Red TCP/IP).*

Uno de los grandes beneficios que ha lanzado Linux a la fama es su soporte a complejas configuraciones de red. De hecho, el código de Linux es examinado por compañías de desarrollo de equipos para redes comerciales como ruteadores y conmutadores (switches).

Opciones de Red

Hay aplicaciones que se comunican directamente con los dispositivos de red sin pasar por un protocolo de red implementado por el núcleo. Para ello es necesario responder *'y'* a *Packet socket*. Si se decide compilarlo como módulo, dicho módulo se denomina *af_packet.o*.

Kernel/User netlink socket permite comunicación bidireccional entre el núcleo y los procesos de usuario. Esta comunicación es utilizada entre otras aplicaciones por el *firewall* (cortafuegos) y el demonio de *ARP*. Para ello también se debe responder *'y'* a *IP : firewall packet netlink device*. También es utilizado por el dispositivo *ethertap* que permite a los programas de usuario acceder directamente a los paquetes *Ethernet*. También se necesita el controlador *network link*. Se puede responder *'y'* si no se está seguro.

Los programas de usuario pueden conseguir información sobre el enrutamiento en la red, siempre que cree un archivo especial de dispositivo de caracteres llamado *«/dev/route»* cuyo número mayor sea 36 y su número menor sea 0. Para ello debe utilizar el comando *mknod*. Todo esto se realiza al responder *'y'* a *Routing messages. Netlink device emulation* sólo se conserva por compatibilidad con versiones anteriores del núcleo y se eliminará en próximas versiones. De momento se debe marcar *'y'* .

Firewall.

Un *firewall* suele ser una computadora que protege una red local de posibles ataques. Su funcionamiento es sencillo: examina los paquetes que le llegan produciendo un filtrado de dichos paquetes atendiendo a su tipo, origen o destino. Si va a utilizar su computadora como «filtro» de una red local marque *'y'* en *Networkfirewalls*. Si su red además, funciona bajo el protocolo TCP/IP, adicionalmente debe marcar 'y' en *IP :firewalling*.

También es necesario para el enmascaramiento de *IPs (IP masquerading)*, que oculta las direcciones IP de las máquinas dentro de una red de forma que desde fuera sólo es visible la dirección del *firewall*. Siguiendo este método no será necesario asignar una dirección IP válida a cada una de las máquinas de las red, sino que se podrán usar direcciones ficticias.

Opciones para ruteo.

Al marcar *'y'* en esta opción asegúrese de marcar *'n'* en *Fast Switching.* Normalmente sólo se responderá sí a esta opción en la computadora de la red que utilice como *router* (encaminador o rutador).
Las últimas opciones de configuración de red se refieren a la utilidad de los *firewall* y del enmascaramiento de IP.

Filtrado de sockets

Socket filtering permite que los programas de usuario enlacen un filtro a un *socket* de forma que puedan indicarle al núcleo qué tipo de datos pueden o no pueden pasar. Funciona con todos los tipos de *sockets* excepto con los TCP. Los *sockets* son el mecanismo estándar para acceder a la red. Sin embargo, también los utilizan distintos programas aunque la computadora no esté conectada a una red, entre ellos *X-Windows*. Para habilitar marque *'y'* en *Unix domain sockets*.

Protocolo TCP/IP (Transmisión Control Protocol/Internet Protocol)

Una opción importante es *TCP/IP networking*. Hay que recordar que la pila de protocolos TCP/IP es utilizada por distintas aplicaciones aunque su sistema no esté conectado a la red.

Si se piensa utilizar el sistema como *router* debe responder *'y'* a *IP : advanced router.* Esto le permitirá redistribuir los paquetes por la red.

IP: policy routing, IP: equal cost multipath, IP: use TOS value as routing key, IP: verbose routing monitoring, IP: large routing tables y IP: fast network address translation le permitirán configurar la forma en la que se redireccionan los paquetes.

BOOTP (BOOTstrap Protocol - Protocolo de Arranque por red) y RARP (Reverse Address Resolution Protocol – Protocolo de resolución inversa de direcciones)

Una de las opciones que en este trabajo es de las más importante es ésta. La configuración de red IP de las máquinas pueden realizarse de forma automática utilizando los protocolos *BOOTP o RARP* .

Para ello marque *'y'* en *IP: Kernel level autoconfiguration.* Para usar esta opción necesitará un servidor de *BOOTP o RARP* funcionado en la red.

QoS AND FAIR QUEUEING; ENCOLAMIENTO Y QoS. QoS (Quality of Service; Calidad del servicio)

Es una de las características avanzadas de red del Kernel de Linux. Permite especificar cómo quiere que vaya priorizado cada paquete antes de enviarlo, dejándole designar varios niveles de servicio. A menos que se sepa exactamente qué opciones se quieren en esta lista de características y las repercusiones de usarla, es mejor dejar las configuraciones iguales.

A menudo, el núcleo tiene en su cola varios paquetes por enviar por la red. Por lo tanto, se tendrá que decidir qué paquete se envía en primer lugar al dispositivo de red. Este problema se agrava en caso de que algunos de los paquetes por enviar correspondan a conexiones en tiempo real, las cuales requieren una cierta cantidad mínima de datos para garantizar la calidad de servicio (QoS).

Si se desea utilizar el algoritmo de planificación por defecto responda *'n'*, e *'y'* en caso contrario. Si se responde *'y'* tendrá a su disposición toda una serie de planificadores que utilizan un amplia gama de algoritmos de planificación con los que podrá experimentar

TELEFONÍA EN LINUX.

Existe soporte para la telefonía en Linux, aunque actualmente sólo se soporta la tarjeta de *QuickNet*. Se pueden tener varias líneas simplemente añadiendo más tarjetas de telefonía. Si se dispone de una de estas tarjetas y se quiere conocer el estado actual de la telefonía en Linux puede leer el archivo *ixj.txt* en el directorio *' Documentation/telephony'* de las fuentes del núcleo.

***SCSI* SUPPORT; SOPORTE *SCSI*.**

El soporte *SCSI* permite la utilización de toda la familia de dispositivos *SCSI* (discos duros, unidades de cinta, CD-ROM, ...) además de otros tipos de dispositivos que, sin ser *SCSI*, requieren de él para su utilización como, por ejemplo, las unidades Iomega ZIP para el puerto paralelo.

Si se desea soporte para discos duros *SCSI* se responderá *'y'* a *SCSI disk support*. Si se necesita soporte para las unidades de cinta para copias de seguridad *SCSI* responda *'y'* a *SCSI tape support*. Sin embargo, ya que las unidades de cinta no se utilizan constantemente, es aconsejable, para reducir el tamaño del núcleo, configurarlas como módulo. El módulo generado se denomina *st.o*. Si se tiene un CD-ROM *SCSI* se necesita adicionalmente marcar *'y'* en la opción *SCSI CD-ROM support*.

Si se desea usar cualquier otro tipo de dispositivo *SCSI* (escáners, grabadoras de Cds, ...) que no sean discos duros, unidades de cinta o CD-ROM responda *'y'* a *Generic SCSI support*. Esta operación no es suficiente para poderse usar, sino que se necesitará un controlador específico para cada dispositivo.

Si se utiliza un dispositivo *SCSI* que ocupa más de un *LUN* (número de unidad lógico), como, por ejemplo, un *CD-Jukebox* se necesita responder *'y'* a *Probe all Luns on each SCSI device*. Normalmente responde *'n'* a esta pregunta.

Se puede hacer que los mensajes de error producidos por los dispositivos *SCSI* sean más explícitos marcando *'y'* en *Verbose: SCSI error reporting*.

SCSI LOW-LEVEL DRIVERS; CONTROLADORES *SCSI* A BAJO NIVEL.

Los controladores *SCSI* de bajo nivel aportan el soporte para el hardware *SCSI* específico. Hay que revisar la lista que aparece y activar las opciones correspondientes a los dispositivos *SCSI* presentes en el sistema.

NETWORK DEVICE SUPPORT; SOPORTE A DISPOSITIVOS DE RED.

En esta opción encontramos:

Network device support

Nos sirve para poder conectarse a otra computadora a través de una red ya que al igual que en el subsistema *SCSI*, el subsistema de red necesita tener un controlador configurado para la tarjeta de red. Es necesario conocer el tipo de tarjeta de red que tiene el sistema para poder tener soporte para ese tipo de tarjeta de red.
Si en un futuro se llegara a necesitar una segunda tarjeta de interfaz de red de un fabricante diferente, o si se quiere anticipar a posibles sustituciones de la tarjeta, se puede elegir soporte para varias tarjetas. Cuando se cargue el Kernel, automáticamente probará todas las tarjetas que pueda soportar y activará los controladores para cada tarjeta que encuentre.

Dummy net driver support

Si se piensa utilizar los protocolos PPP o SLIP se necesitará responder *'y'* a esta opción.

Ethertap network tap

Si se respondió afirmativamente a *Kernel/User network link driver* ahora también se puede responder de igual manera a *Ethertap network tap*. De esta

forma se permitirá que los programas de usuario puedan acceder a las tramas ethernet directamente. Para ello, debe crear un nuevo archivo especial de caracteres llamado *'/dev/tap0'* cuyo número mayor sea 36 y su número menor sea 16. Utilice el comando *mknod*.

FDDI driver support

Añade soporte para las conexiones que utilizan los distintos tipo de fibra óptica. La conexión con fibra óptica es usada en redes de alta velocidad. Para conectarse a este tipo de redes de alta velocidad se utilizan tarjetas del tipo de la *DEFEA* de Digital.

High Performance Parallel Interface (HIPPI)

Es un estándar para conexión punto a punto bidireccional de alta velocidad (800Mbit/s o 1600Mbit/s). Para poder utilizar la tarjeta para la conexión a un sistema HIPPI se debe también incluir soporte para su tarjeta de red HIPPI.

A continuación, puede seleccionar qué protocolos desea instalar.

PLIP (Parallel Line Internet Protocol)

Se utiliza para conectar dos máquinas a través del puerto paralelo. Las dos computadoras que se están interconectando deben tener esta opción activada en el núcleo.

SLIP (Serial Line IP), CSLIP (SLIP comprimido) o PPP (Point to Point Protocol)

Se utiliza para conectarse a un proveedor de Internet, para conectarse a otra máquina UNIX o para montar un servidor *SLIP/CSLIP*. Es un protocolo utilizado para transmitir una conexión IP a través de una línea serie (líneas telefónicas, cables de módem nulo, ...). SLIP ha caído en desuso y ha sido sustituido por PPP (*Point to Point Protocol*). PPP es usado con la misma finalidad que SLIP. Es usado en conjunción con *pppd*.

ETHERNET (10 ó 100 MBITS)

Ethernet es el tipo de tarjeta de red local más extendida. Existe una gran variedad de tipos de tarjetas de red Ethernet, variando principalmente el tipo de cable utilizado y su velocidad de transmisión. Hoy en día el tipo de cable más utilizado es el par trenzado, ya que entre otras muchas cosas, es mucho más sencillo de instalar, debido al reducido tamaño de su sección.

En esta pantalla aparece un gran número de tipos de tarjetas, incluidos algunos que están en desuso. Sería extraño que alguna tarjeta de red no apareciera en esta lista (o un tipo compatible con su tarjeta de red). Se debe marcar el tipo de tarjeta de red que se corresponde con la tarjeta de red instalada en el sistema.

ETHERNET (1000 MBITS)

Las tarjetas Ethernet más comunes hoy en día son de 10Mbits o 100Mbits. Sin embargo, también existen tarjetas Ethernet de 1000 Mbits. Este tipo de tarjetas son utilizadas a menudo en redes de estaciones de trabajo (*NOW*), donde es necesaria una red de altas prestaciones.

DISPOSITIVOS TOKEN RING

Token Ring fue un tipo de red diseñada por IBM. Si está conectado a una red que utiliza Token Ring active esta opción y seleccione el tipo de tarjeta de red que utiliza para conectarse a la red.

ISDN (RDSI)

Hoy en día es común conectarse a Internet a través de *ISDN* (o *RDSI* en español). Este tipo de conexiones proporcionan un acceso mucho más rápido que un módem tradicional. Otras de las ventajas por la que se ha extendido en empresas es que se pueden mantener simultáneamente más de una conexión a Internet. Para ello es necesario disponer de una tarjeta *RDSI* y contratar el servicio con una de las compañías telefónicas que operen en la correspondiente zona.

AMATEUR RADIO SUPPORT; SOPORTE A RADIOAFICIONADO

Con esta opción se puede permanecer conectado a la red a través de los controladores de radioaficionado. Estas configuraciones permiten configurar un enlace punto a punto sobre los que se asienta TCP/IP.

IRDA SUPPORT; SOPORTE IRDA

Aquí encontramos el soporte para dispositivos infrarrojos que ahora es parte estándar de Linux, lo cual es especialmente útil para los propietarios de sistemas que tengan un puerto *IR*.

INFRARED PORT DEVICE DRIVERS; CONTROLADORES DE DISPOSITIVOS DE PUERTOS INFRARROJOS

Éstos son los controladores necesarios si tiene activado el soporte *IRDA*.

ISDN SUBSYSTEM; SUBSISTEMA RDSI

Seleccione estas opciones si quiere utilizar un módem *RDSI*. A diferencia de algunos de los menús precedentes, las entradas de soporte RDSI no se separan de las entradas de controlador de dispositivo, así que sea cuidadoso con sus valores.

OLD CD-ROM DRIVERS; CONTROLADORES DE CD-ROM ANTIGUOS

Si se instala Linux en un sistema antiguo, este menú puede ser útil. Existe un buen número de CD-ROM que ni son *IDE* ni son *SCSI*. Entre ellos se encuentran los primeros lectores de Mitsumi, Sony y Phillips. Estos tipos de CD-ROM requieren un controlador específico para ese dispositivo. Para obtener soporte para este tipo de CD-ROM responda *'y'* a *Support non-SCSI/IDE/ATAPI CDROM drives* y responda afirmativamente también al correspondiente controlador para el dispositivo que está en el sistema.

CHARACTER DEVICES; DISPOSITIVOS DE CARÁCTER

Este menú cubre una clase genérica de dispositivos que se comunican con el sistema byte por byte (a diferencia de los dispositivos como discos duros, que se comunican mediante bloques de bytes). Encontrará soporte para puertos series, terminales y temporizadores de control.

Support for console on virtual terminal

Las terminales virtuales permiten ejecutar varias terminales virtuales en un mismo terminal físico. Esto es muy útil para poder tener diferentes sesiones abiertas simultáneamente e incluso para tener abiertas sesiones en modos texto y sesiones en X-Windows simultáneamente.

Habitualmente los mensajes producidos por el núcleo son mostrados en la terminal virtual activa. Para que el sistema proceda de esta forma, que es la más común, se debe responder *'y'* a *Support for console on virtual terminal*. En caso contrario los mensajes producidos por el núcleo sólo serán enviados a un puerto serie.

Standard/generic (dumb) serial support

Incluye soporte para los puertos serie estándar. Normalmente se responderá *'y'* a esta opción, a no ser que no piense utilizar los puertos serie de su máquina.

Support on console on serial port

Esta opción nos sirve si se va a utilizar alguna característica no estándar del controlador para los puertos serie como, por ejemplo, el soporte otros puertos *COM* más allá de los *COM 1,2,3* y *4* por defecto, etc.

MICE, RATONES

Este menú cubre el soporte para ratones típicos de una computadora. En este caso se podrá decidir sobre qué tipo de ratones son soportados. Si se conoce exactamente el tipo de ratón que se va a utilizar se recomienda añadir el controlador correspondiente y deje el resto en *'n'* o como módulo.

WATCHDOG CARDS; TARJETAS DE CONTROL

Las tarjetas de control son herramientas especiales que se conectan a la fuente de alimentación o al botón de reinicio del sistema. Internamente, ejecutan un pequeño temporizador que se reinicia antes de que expire; de otra forma, el sistema se reinicia (el reinicio del temporizador normalmente se hace por software). Este temporizador permite el reinicio automático en el caso de que haya una caída del sistema grave que el sistema no pueda corregir y se vuelve a ejecutar.

La tarjeta de control son recomendables para sistemas remotos de gran uso. Es recomendable asegurarse que este dispositivo esté incluido en este menú.

VIDEO FOR LINUX; VÍDEO PARA LINUX

No hay que confundir estos controladores con el soporte de vídeo normal necesario en X-Windows. Las tarjetas listadas en este menú proporcionan captura de vídeo y audio, y características avanzadas.

JOYSTICK SUPPORT; SOPORTE A JOYSTICK

Sólo se compilarán estas configuraciones si se piensa jugar en el sistema.

FTAPE, THE FLOPPY TAPE DEVICE DRIVER; EL CONTROLADOR DE DISPOSITIVO DE CINTA

Este es otro recuerdo de los sistemas antiguos. Antes de la llegada de los discos Zip y de las soluciones de copias de seguridad por red de computadora, cualquiera que quisiera hacer copia de seguridad del sistema DOS necesitaba una unidad de cinta unida al propio sistema. Las opciones más utilizadas se conectaban al sistema mediante una interfaz de controlador de disquete. Tales dispositivos todavía están en uso, pero puede ignorar este menú con seguridad si no necesita este controlador de dispositivo.

FILE SYSTEMS; SISTEMAS DE ARCHIVOS

Esta opción de Linux nos permite soportar un gran número de sistemas de archivos. Estos sistemas de archivos pueden ser de muy variados tipos: *FAT, VFAT, ext, ext2, minix,* etc. Linux permite el acceso a sistemas de archivos de otros sistemas operativos, entre ellos *OS/2, Windows '95/'98 y MS-DOS.*

Esto es especialmente útil en instalaciones que no sean en PC, como en *Sun SPARC o Apple Macintosh.* En general, si no reconoce un elemento en particular o si el sistema de archivos no es nativo de su hardware, puede deshabilitarlo. El único sistema de archivos que debe activar es el soporte para *ext2.*

También permite asignar una cuota máxima, de forma que los usuarios no puedan almacenar archivos que ocupen más espacio en el sistema de archivos del límite asignado en su cuota.

aviso: Aunque es una opción compilar *ext2* como módulo, ¡ NO LO HAGA! . El Kernel necesita saber cómo leer los módulos de un sistema de archivos *ext2* antes de poder cargar módulos adicionales.

NETWORK FILE SYSTEM; SISTEMAS DE ARCHIVOS DE RED

Estas opciones proporcionan soporte a sistemas de archivos de red, como *NFS, NCP , SMB(SAMBA) y CODA.*

PARTITION TYPES; TIPOS DE PARTICIONES.

Si Linux necesita compartir su disco con otros sistemas que usan su propio esquema de partición, necesitará activar el soporte para ese esquema en particular. Sin embargo, si su sistema Linux es un equipo aislado, puede ignorar estas opciones.

NATIVE LANGUAGE SUPPORT; SOPORTE A IDIOMA NATIVO

El soporte a idioma nativo (*NLS*) es útil cuando se necesita proporcionar un acceso fácil a los caracteres específicos de un lenguaje. El soporte de lenguaje por defecto es el inglés. A menos que se necesiten otras opciones, puede ignorar estas configuraciones.

CONSOLE DRIVERS; CONTROLADORES DE CONSOLA

Cuando Linux incrementó sus actitudes para entornos heterogéneos, se hicieron disponibles más opciones para varios tipos de consola. Por defecto, Linux soporta la consola *VGA*, la cual, normalmente, es la necesaria para las *PC* de la mayoría de la gente. Si necesita que Linux muestre otras consolas, seleccione el soporte en este menú.

SOUND AND ADDITIONAL LOW-LEVEL SOUND DRIVERS; SONIDO y CONTROLADORES DE SONIDO A BAJO NIVEL ADICIONALES

Estos dos menús contienen valores para tarjetas de sonido y sus características. Esto le permite utilizar el Real audio, escuchar archivos MP3, etc.
En Linux la configuración de la tarjeta de sonido requiere que se haya incluido soporte para dicha tarjeta en el núcleo, dentro del propio núcleo o como módulo. Linux soporta prácticamente todas las tarjetas de sonido más habituales del mercado, y será muy difícil que no haya soporte para alguna.

KERNEL HACKING; INVESTIGACIÓN DEL KERNEL

A menos que se planeen hacer desarrollos sobre el Kernel, se contestará a esta pregunta con *'N'*. *Kernel Hacking* permite aumentar el control que se tiene sobre la ejecución, incluso aunque el sistema se bloquee. No responda *'y'* a no ser que se tenga muy claro lo que se quiere hacer.

RECOMENDACIONES GENERALES DE LA CONFIGURACION DEL KERNEL

Cuando se configura el núcleo para su compilación se pueden encontrar algunas opciones que no se comprenden. Si no se está seguro de qué se debe responder a alguna de estas preguntas, simplemente hay que dejar el valor por defecto.

Los valores por defecto se fijan de forma que el núcleo sea lo más estable posible en todas las computadoras. La primera vez que se configure el

El poder interconectar PCs y formar una red que básicamente es lo que nos permite fabricar un Beowulf. Un Beowulf usualmente conecta sus nodos usando tarjetas Ethernet y un Hub o Switch a 10/100 Mbps, pero en realidad se puede usar cualquier dispositivo capaz de soportar TCP/IP.

En cuanto a los ambientes de programación que Linux ofrece, están los lenguajes que son estándares para la programación en general, tal y como las versiones libres de C/C++ (gcc), java, fortran, phyton, perl, etc. A estos ambientes de programación se les pueden agregar algunas librerías extras que permiten el trabajo en paralelo, lo que hace que el ambiente sea bastante flexible para la creación de diversas aplicaciones.

El permitir iniciar una máquina remotamente es una característica crucial para la administración de un Beowulf, ya que esto nos permite que únicamente desde el servidor se manejen todas las demás máquinas, dando la apariencia de tener sólo una máquina multiprocesador.

Estas características proveen de recursos a las necesidades que se tienen para este trabajo, y son las razones por las que se pensó en Linux como base para este desarrollo.

Capítulo III

Construcción del Cluster

3. CONSTRUCCIÓN DEL CLUSTER

3.1. DISEÑO DE UN CLUSTER BEOWULF

En este capítulo se da una visión completa de la construcción, instalación y administración de un Cluster Beowulf.

Antes de iniciar es importante considerar los aspectos de diseño y tomar ciertas decisiones entre las que destacan:

1) Escoger el tipo de los procesadores que van a conformar las CPUs que se van a utilizar y su velocidad de procesamiento.
2) La cantidad de memoria RAM, tanto para el servidor como para los nodos
3) Capacidades de Disco Duro, para el servidor y clientes o clientes sin disco.
4) Tipo de interconexión entre los nodos y el servidor.

En general, si se puede contar con el tipo de recursos que uno desea, sin importar el costo, entonces pensamos en la aplicación en que se va a trabajar y tomando esto como base se decide cuáles de las características antes mencionadas merecen una prioridad mayor, por ejemplo; puede que la aplicación necesite más memoria que disco duro o viceversa.

También existen casos en que no se cuenta con recursos muy amplios, por lo que el cluster se construye basándose en el equipo existente, y es en la parte de programación donde se puede o no ajustar la aplicación al sistema (dependiendo de la aplicación).

3.1.1. REQUERIMIENTOS MÍNIMOS DE UN NODO

CPU

Básicamente un nodo tendrá que tener como mínimo, una CPU 386 y una tarjeta madre compatible con Intel. Existe la posibilidad de trabajar con otras arquitecturas que soporten Linux, como es SPARC. Los procesadores Intel 386 funcionarían pero no tendrían un buen desempeño, debido a que sus velocidades de procesamiento están muy por debajo del requerido.

MEMORIA

Los requerimientos de memoria se deciden dependiendo del procesamiento de datos en la aplicación. Un nodo tendría que contener como mínimo 16MB de memoria.

DISCO DURO

Usando un espacio de disco centralizado, los nodos se pueden inicializar desde un disco de 3½, un pequeño disco duro o un sistema de archivos en red. Entonces los nodos pueden acceder a su partición raíz desde un servidor de archivos a través de la red, normalmente usando el Network File System (NFS). Esta configuración funciona mejor en un entorno con mucho ancho de banda en las conexiones y con un gran rendimiento en el servidor de archivos. Para un mejor rendimiento bajo otras condiciones, cada nodo tendría que tener suficiente espacio en el disco local para el sistema operativo, la memoria virtual y los datos. Cada nodo tendría que tener como mínimo 200 MB de espacio en disco.

RED DE INTERCONEXIÓN

Como mínimo cada nodo tiene que incluir una tarjeta de red Ethernet o Fast Ethernet. Alternativamente, interconexiones de más alto rendimiento, incluyendo las de los tipos Gigabit Ethernet y Myrinet, podrían ser usadas en conjunto con CPUs más rápidas.

OTROS

Una tarjeta de vídeo y una unidad de disco de 3½, completan un nodo funcional. Los teclados y los monitores sólo se necesitan para la carga y configuración inicial del sistema operativo

REQUERIMIENTOS SERVIDOR

El servidor es recomendable que tenga recursos mayores que los nodos, además de que es el único que contará con un teclado, monitor, mouse (opcional) así como otros dispositivos que se consideren necesarios.

3.2. CONSTRUCCIÓN DE UN BEOWULF SIN DISCOS

Para este trabajo los recursos con los que se contaban y fueron usados son los siguientes:

NODOS

Procesador Intel 80486
Tarjetas madre compatible con Intel para procesadores 80486
Memoria RAM de 16Mb
Sin disco duro
Unidad de disco de 3 ½
Tarjetas de red 3Com ISA 3c509 (combo)
Tarjeta de vídeo

SERVIDOR

Procesador Pentium II
Tarjeta madre compatible con Intel para procesador Pentium II
Memoria RAM de 98Mb
Disco Duro de 3 Gb
Unidad de CD-ROM
Unidad de disco de 3 ½
Monitor SVGA
Teclado
Mouse
Tarjetas de red 3Com ISA 3c509 (combo)
Tarjeta de vídeo

RED DE INTERCONEXIÓN

Tipo de Red : Área Local (LAN)
Tecnología: Ethernet
Esquema de direccionamiento: IP Clase C de tipo privado con la dirección de red :192.168.10.0 y mascara /24 255.255.255.0
Dispositivo de interconexión: Hub
Cables de interconexión: Par trenzado UTP Cat 3 y UTP Cat 5
Conectores RJ45

En el servidor se tienen dos tarjetas Ethernet, lo cual nos brinda la facilidad de tener salida hacia una red externa que nos pueda conducir a Internet y la otra permite conectarse a red interna 192.168.10.0

SOFTWARE

El software que se utilizó en este trabajo nos permite brindar los servicios necesarios del Nodo Central hacia los demás nodos, así como la comunicación entre todas las máquinas para establecer la máquina paralela.

Algunos de los servicios instalados son:

- NFS
- DHCP
- NIS
- OPEN SSH

El software que se utiliza para la programación en paralelo:

- PVM

Para el arranque de los nodos:

- Etherboot y Mini-Netboot

ESQUEMA DEL CLUSTER

En la figura 3.1 se muestra el esquema de interconexión del cluster, así como el esquema de direccionamiento utilizado.

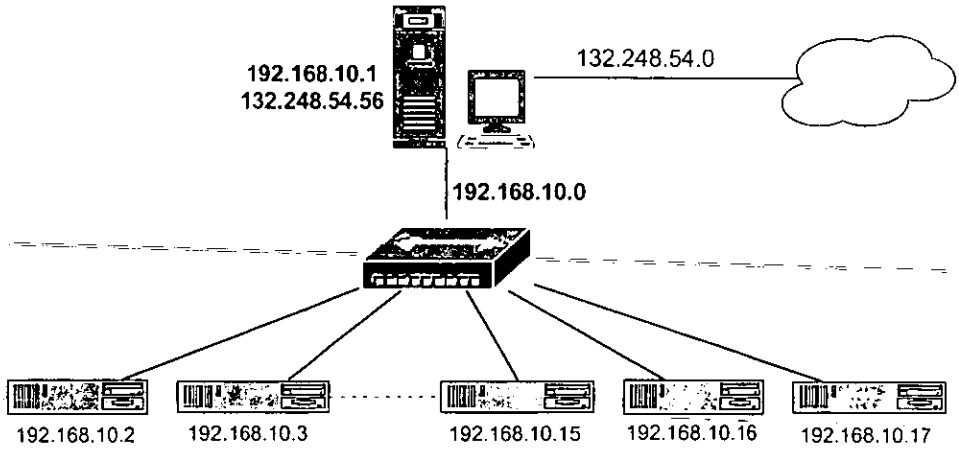


Figura 3. 1 Esquema de interconexión del cluster.

El sistema de archivos para los nodos se comparte por medio de NFS. En el esquema que se muestra a continuación (figura 3.2)se da una descripción de cómo está acomodado el sistema de archivos para los nodos sin disco en el servidor. Los nodos arrancan desde un disco de 3 ½ previamente configurado con el paquete Etherboot. Con la ayuda de DHCP, TFTP y NFS, se logra la comunicación de los nodos con el servidor.

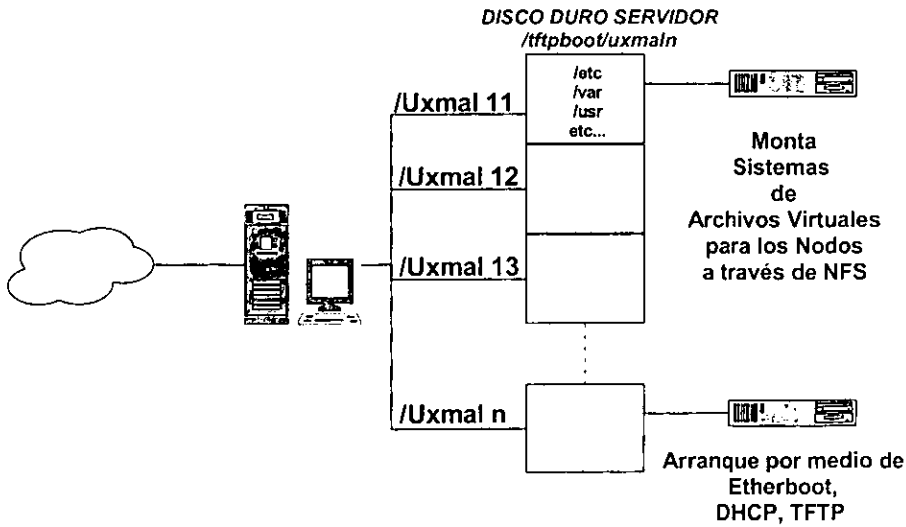


Figura 3. 2 Esquema del sistema de archivos del cluster.

3.3. CONFIGURACIÓN DEL SERVIDOR Y NODOS UTILIZANDO LINUX RED HAT 6.2

La instalación que brinda la distribución de Linux proporcionada por Red Hat en su versión 6.2, consiste en una serie de preguntas con las cuales se da una configuración inicial al sistema.

Antes de instalar el Sistema Operativo se tiene que planificar una serie de aspectos para poder contestar a todas las preguntas durante la instalación. Para realizar esta planificación podemos basarnos en lo siguiente:

- Medios con los que se cuenta, básicamente el hardware con el que contamos. En esta versión de Linux puede que se detecte automáticamente todos los periféricos sin necesidad de pedirnos información adicional, pero en algunos casos no, y es necesario especificarlos.

Se debe de tener a la mano las especificaciones relativas a:

- Modelo de la tarjeta de video y memoria que contiene.
- Tipo de monitor y frecuencias de barrido horizontal y vertical.
- Modelo de dispositivos y controladores, del ratón, tarjeta de red y, dispositivos hardware que tengamos y que se desee que funcionen en Linux en la computadora servidor.

- Nombre que se va a poner a la máquina Linux.

- Qué software y servicios se van a instalar. (Colocar las herramientas de programación, desarrollo y clustering)

- Definir cómo se va a distribuir el disco ya que al momento de la instalación se requiere y es mejor planearlo con anticipación. Se recomienda dejar el mayor espacio posible a (/) raíz, aquí se creará el sistema de archivos para los nodos y en segundo lugar a la partición */usr,* donde se instalarán las herramientas o software de paralelización como PVM.

En nuestro caso, la distribución del disco de 3 Gb se muestra en la tabla 3.1.. Para cualquier de disco se recomienda tener estas particiones.

/	1.7 GB
/usr	900 MB
/var	75 MB
/boot	20 MB
Swap	200 MB
/tmp	100MB

Tabla 3. 1

Después de haber planeado lo anterior se procede a la instalación del Sistema Operativo Linux Red Hat 6.2 en la computadora que va a ser nuestro servidor.

La instalación se hará por medio de una unidad de CD-ROM, si el sistema soporta CD-ROM de arranque, sólo hay que introducir el CD e iniciará el proceso de instalación; si la computadora no es tan reciente y no permite arrancar desde la unidad de CD-ROM entonces es necesario crear un disco de arranque.

Para crear el disco de arranque, desde el prompt de MS-DOS

1) Habrá que ingresar a la unidad de CD-ROM en la que se encuentra el CD de instalación de RedHat y entrar al directorio *D:\dosutils* (la letra D se refiere a la unidad de CD en la computadora)

2) En el directorio *D:\dosutils* del CD-ROM de RedHat encontrará la utilidad *rawrite*. Una vez situados en este directorio tecleamos *rawrite*.

   D:\dosutils> rawrite. Este programa genera un disco de inicio para la máquina.

3) El programa pide el origen de la imagen, la imagen está en el directorio *D:/images* del CD-ROM.

   Se teclea la imagen *\images\boot.img* y se introduce el disco (3 ½).

   boot.img permite arrancar el sistema de instalación de Red Hat desde el CD-ROM.

La instalación se puede realizar de dos maneras, ya sea desde un ambiente de tipo texto o desde un ambiente gráfico. En ambos casos, la información que se requiera se solicitará durante todo el proceso.

Durante esta configuración se aprovecharán las facilidades de manejo y configuración que proporciona tanto Linux como algunos de los paquetes que se manejan.

Ya instalado el sistema operativo se procede a la configuración del equipo, es decir, configuración de los ambientes de red y los diferentes servicios que prestara nuestro servidor a los nodos.

Antes de detallar cada una de la configuraciones necesarias para la construcción del cluster mostramos el siguiente cuadro sinóptico que las resume.

ESQUEMA DE CONSTRUCCIÓN DEL CLUSTER BEOWULF

* Configuración inicial del Servidor
  - Configuración de las Tarjetas deRed
    - Editar y modificar los archivos
      - ⇒ */etc/conf.modules*
      - ⇒ */etc/sysconfig/network – scripts*
  - Configurarción de la Red
    - Editar y modificar el archivo
      - ⇒ */etc/sysconfig/network*
  - Colocar IP y alias de las máquinas en el archivo */etc/hosts*
  - Asignar el nombre del Servidor
* Etherboot y Mini – Netboot
  - Instalación
  - Activar el servicio de TFTP
  - Ubicar el arranque de los nodos por Red
  - Creación del sistema de archivos de los Nodos
* Configuración del Kernel
  - Tarjetas de Red Soportadas
  - Opciones de Red
  - Soporte NFS
* Colocar la imagen del Kernel para los Nodos

* NFS
  - Configuración del Servidor
    - Exportar o compartir directorios en el archivo */etc/exports*
    - Inicializar o reinicializar el servicio
  - Configuración del Cliente
    - Montar sistemas de archivos compartidos en el archivo */etc/fstab*

* DHCP
  - Configuración del Servidor DHCP
    - Agregar datos necesarios en */etc/dhcpd.conf*
  - Configuración del Cliente
    - Modificar y editar el archivo
    - ⇒ */tftpboot/nodo_n/etc/sysconfig/network – scripts/ifconfig – eth0*

* NIS
  - Configuración del Servidor NIS maestro
    - Nombre del dominio
    - Arrancar *ypserv*
    - Editar el archivo *Makefile*
    - Ejecutar *ypinit*
  - Configuración de un Cliente NIS
    - Editar *yp.conf*
    - Configurar los servicios de arranque

* Secure Shell
  - Instalación de Zlib
  - Instalación de OpenSSL
  - Instalación OpenSSH

* Sintonización
  - Nodos
    - Eliminación de servicios no utilizados
    - Duplicación del sitema de archivos para cada uno
  - Servidor
    - Cerrar los puertos no utilizados en el archivo ⇒ */etc/services*

* PVM
  - Instalación
  - Configuración

3.3.1. CONFIGURACIÓN INICIAL

Durante esta configuración se aprovecharán las facilidades de manejo y configuración que proporciona tanto Linux como algunos de los paquetes que se manejan.

Las configuraciones siguientes son para la máquina servidor.

I. Configuración de la(s) tarjeta(s) de red.

I.I. Editar el archivo /etc/conf.modules
Incluir las siguientes líneas:
alias eth0 modulo_tarjeta
options eth0 io=0xdir_memoria irq=num_interrupción
io: Dirección de entrada y salida
irq: interrupción asignada a la tarjeta.

Ejemplo:
Con una tarjeta NE2000 o compatible se puede utilizar la siguiente configuración:

alias eth0 ne
options ne io=0x300 irq=5

Donde *ne* es el módulo de la tarjeta NE2000. Si es una tarjeta ISA se utiliza este módulo, además de que si es del tipo PnP hay que deshabilitar esta característica por medio del software proporcionado por el fabricante.

Para nuestro caso, utilizamos 2 tarjetas *3Com (3c509)* en el servidor con la siguiente configuración.

alias eth0 3c509

En este caso, no fue necesario declarar la línea marcada como *options.*

Se prueba que la tarjeta funcione correctamente con el comando *modprobe*

#modprobe mod_tarjeta

Si no se muestra ningún mensaje de error, esto indica que la tarjeta ha sido configurada correctamente.

I.II. Una vez probadas y configuradas las tarjetas, en el directorio */etc/sysconfig/network-scripts* debe existir un archivo con el

nombre *ifcfg-ethx*, donde se darán de alta las características propias de cada tarjeta.

El contenido del archivo será parecido al siguiente:

```
DEVICE="eth0"
IPADDR="dirección_ip_de_la_tarjeta"
NETMASK="mascara_de_la_dir_ip"
NETWORK="dirección_de_red"
BROADCAST="broadcast_de_la_red"
ONBOOT="yes/no"
```

Si el archivo no existe, se crea y para que el sistema reconozca la tarjeta, se utiliza el comando:

#if-up ethx

Donde *x* es el número del adaptador de red. En nuestro caso, al utilizar una tarjeta para la red interna del cluster, y otra para la salida hacia el exterior, se crearon 2 archivos:
ifcfg-eth0 e ifcfg-eth1

Donde *ifcfg-eth0* es el archivo asociado a la tarjeta que proporciona la salida al exterior, y *ifcfg-eth1* es el archivo asociado a la tarjeta para la red interna.

A continuación se muestra el contenido de los archivos, para nuestras tarjetas.

```
ifcfg-eth1:
DEVICE="eth1"
IPADDR="192.168.10.1"
NETMASK="255.255.255.0"
NETWORK="192.168.10.0"
BROADCAST="192.168.10.255"
ONBOOT="yes"

ifcfg-eth0:
DEVICE="eth0"
IPADDR="132.248.54.56"
NETMASK="255.255.255.0"
NETWORK="132.248.54.0"
BROADCAST="132.248.54.255"
ONBOOT="yes"
```

Para mostrar cómo quedaron configuradas cada una de las tarjetas utilizamos el siguiente comando:

#ifconfig

lo que devolverá una salida con la información referente a cada tarjeta.

```
eth0      Link encap:Ethernet  HWaddr 00:20:AF:39:C6:37
          inet addr:132.248.54.56  Bcast:132.248.54.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:38486 errors:0 dropped:0 overruns:0 frame:0
          TX packets:339 errors:0 dropped:0 overruns:0 carrier:0
          collisions:13 txqueuelen:100
          Interrupt:11 Base address:0x300

eth1      Link encap:Ethernet  HWaddr 00:20:AF:4D:A6:7E
          inet addr:192.168.10.1  Bcast:192.168.10.255  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:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:5 Base address:0x210

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:3924  Metric:1
          RX packets:1247 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1247 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

Para verificar si la tarjeta responde, se utiliza el comando *ping* para probar la respuesta de la tarjeta.

```
#ping eth0
64 bytes from eth0 (192.168.10.1): icmp_seq=0 ttl=255 time 0.3 ms
```

II. En el archivo */etc/sysconfig/network* podremos configurar el nombre de la máquina, el dominio y la puerta de enlace (*gateway*):

Ejemplo:

```
NETWORKING=yes
HOSTNAME="etzna"
GATEWAY="132.248.54.254"
GATEWAYDEV="eth0"
FORWARD_IPV4="no"
NISDOMAIN="cluster.beowulf"
```

III. Editar el archivo */etc/hosts*. En este archivo se colocarán la dirección IP, el nombre de la máquina con su dominio, y el alias por el cual se conocerá a la máquina. En los sitios pequeños sin acceso a Internet, es razonable que cada máquina mantenga su propia copia de la tabla con los nombres de las máquinas de la red local con sus correspondientes direcciones IP. Esta tabla se guarda en el archivo */etc/hosts*.

Ejemplo:

127.0.0.1 localhost.localdomain localhost # denominaciones locales
192.168.10.1 etzna etzna # denominación publica

IV. En el archivo */etc/resolv.conf* se define el dominio y la dirección del servidor DNS.

En nuestro caso, el nombre de dominio externo es *fi-a.unam.mx*. La dirección del servidor DNS que se utiliza es: *132.248.10.2, 132.248.1.3, 132.248.204.1*

Ejemplo:

search fi-a.unam.mx dgsca.unam.mx cluster.beowulf
nameserver 132.248.10.2
nameserver 132.248.204.1
nameserver 132.248.1.3

V. Por último se define el nombre del *host* tanto en el archivo */etc/sysconfig/network* como en el archivo *HOSTNAME*

Ejemplo:

etzna

3.3.2. ETHERBOOT Y MINI-NETBOOT

Como los nodos esclavos no tienen disco duro es necesario hacer uso de ciertos servicios y utilerías para hacerlos funcionar.

El arranque de los nodos por medio de la red se puede conseguir mediante el paquete de distribución gratuita *Etherboot*. Para empezar se debe comprobar en la documentación del paquete que la tarjeta de red a utilizar este incluida para que soporte *Etherboot*. El programa que genera este paquete se puede grabar en una *EPROM* (*Erasable Programmable Read Only Memory* - Memoria de sólo lectura, programable y borrable), aunque también funciona al simular el código que debería estar en la *EPROM* con un disco, que es la opción que vamos a utilizar.

Para crear el disco de inicio es necesario colocar un bloque especial de inicio que se proporciona en la distribución. Este pequeño programa de 512 bytes carga en memoria los bloques del disco que lo siguen, y comienza la ejecución. Por lo tanto, para construir este disco únicamente hace falta concatenar el bloque de inicio con el binario de *Etherboot* que contenga el controlador de la tarjeta de red que se tiene.

Un ejemplo podría ser: *#cat floppyload.bin 3c509.lzrom /dev/fd0*

En la versión de *Etherboot* que se manejó para este trabajo, esto se automatiza mediante un *shell script* (guión de comandos) que viene incluido con el paquete. De esta manera, ahora se crea la imagen para el disco al teclear el comando:

#make bin32/identificador_tarjeta.fd0

Antes de poner el disco de arranque en red, es necesario configurar tres servicios en Linux: DHCP, TFTP y NFS. Los servidores de DHCP y NFS se explican más adelante en cuanto a funcionamiento y configuración.

Instalación de Etherboot y Mini-Netboot

Se crea un directorio en raíz (/) llamado */tftpboot*. Dentro de este directorio se instalan tanto el Etherboot, como el Mini-Netboot.

El paquete Etherboot nos ayuda a crear los discos de arranque de las tarjetas de red. Contiene las imágenes del software que inician a la tarjeta de red.

El paquete Mini-Netboot nos proporciona las utilerías necesarias para crear la imagen que se proporcionará a los nodos.

Para la instalación del Etherboot:

Se descomprime dentro del directorio */tftpboot*, utilizando el comando *tar*.

Dentro del directorio que se crea (*Etherboot-versión*), dentro de la carpeta *src* se escribe en la línea de comandos:

#/tfpboot/Etherboot-5.0.1/src/make

con lo cual se compilan e instalan las imágenes de las tarjetas de red.

Para crear los discos de inicio para las tarjetas de red, dentro del directorio *Etherboot-versión/src* se teclea:

$make bin32/identificador_tarjeta.fd0.

Esto enviará al disco de 3 ½ la información necesaria para que inicie la tarjeta de red en el nodo y comience a buscar su dirección IP en el servidor DHCP, solicitando que se le envíe la imagen de arranque del sistema operativo con el que va a trabajar.

Para una tarjeta del tipo 3c509, la instrucción que se pone en la línea de comandos es la siguiente:

[/etherboot-version/src]$make bin32/3c509.fd0

Para las tarjetas del tipo NE2000, la instrucción es la siguiente:

[/etherboot-version/src]$make bin32/ne.fd0

Para la instalación del Mini-Netboot:

Se descomprime el archivo *Mini-Netboot-0.8.1.tar.gz* por medio del comando *tar*, dentro del mismo y en la línea de comandos se ejecuta:

#./configure;make; make install

con lo que se instalan las herramientas necesarias para hacer la imagen del Kernel transportable.

Servidor TFTP

Para poder lograr que los nodos obtengan su Kernel, es necesario usar un servidor TFTP, comúnmente viene con las distribuciones de Linux, aunque también se puede conseguir el paquete para compilarlo. Normalmente, se lanza el demonio *tftpd* desde *inetd* con una línea como ésta

tftp dgram udp wait root /usr/sbin/tcpd in.tftpd -s /tftpboot

en el archivo de configuración */etc/inetd.conf*.

Si este servicio se encuentra en el archivo */etc/inetd.conf*, se remueve el símbolo # de la línea que hace referencia a este servicio.

Una vez configurado, para no reiniciar la máquina se reinicia el servicio de *inetd* por medio de la orden */etc/rc.d/init.d/inet restart*.

Arranque de los nodos por red

Ahora se procede a arrancar el nodo con el disco que contiene la imagen del controlador de la tarjeta de red. Se debe de detectar la tarjeta Ethernet y lanzar una petición de BOOTP en forma de *broadcast*. Si todo va bien, el servidor DHCP responderá al nodo con la información requerida, aunque al no existir el archivo

/tftpboot/imagen_transportable_Kernel

fallará en cuanto intente cargar el mismo. Para lograr esto se necesita compilar un núcleo especial, uno que tenga la opción de montar el sistema de archivos raíz vía NFS activado. También hace falta habilitar en el núcleo la opción de obtener la dirección IP desde la respuesta BOOTP original (RARP).

Adicionalmente es necesario compilar el controlador para el adaptador de red en el núcleo en vez de como un módulo aparte.

Esto se hará más adelante en la compilación del Kernel, donde al momento de la configuración elegiremos estas opciones.

La imagen del *Kernel (bzImage)* que se crea al terminar la compilación no se puede enviar directamente. Debe ser convertida en una imagen marcada. Ésta es una imagen normal del núcleo con una cabecera especial que le dice al cargador de arranque en red dónde han de almacenarse los bytes en memoria y en qué dirección empieza el programa. Para crear esta imagen marcada se usa un programa llamado *mknbi-linux*, que puede ser encontrado en la distribución *Mini-Netboot*. Después de crear la imagen, habrá que colocarla en el directorio */tftpboot* con el nombre especificado en los archivos de configuración del DHCP.

La nueva imagen creada del Kernel después de la compilación está en */usr/src/Linux/arch/i386/boot* ; y se copia al directorio */tftpboot.*

Dentro del mismo se ejecuta el comando

#/usr/local/bin mknbi-linux-d/tftpboot bzImage /tftpboot/vmlinuz.boot

Con el cual se hace transportable el Kernel, permitiendo el inicio remoto de los nodos.

Al volver a arrancar el nodo con el disco de inicio, ahora se debe llegar hasta cargar la imagen del núcleo y empezar a ejecutarlo. El arranque continuará hasta el punto en que intenta montar un sistema de archivos desde la raíz. En este punto, el servidor NFS proporciona los sistemas de archivos a exportar que serán montados.

Por varias razones, no es una buena idea asignar el sistema archivos raíz del servidor como sistema de archivos raíz de los nodos. Una de ellas consiste sencillamente, en que existen diferentes archivos de configuración, de forma que el nodo leerá una información errónea. Otra razón es la seguridad. Es peligroso tener un mismo sistema de archivos para todos los nodos, ya que es indispensable que sólo lo maneje el servidor. Sin embargo, un sistema de archivos raíz para el nodo no ocupa demasiado espacio, solo unos 47 MB y que pueden ser duplicados fácilmente para el resto de los nodos.

Idealmente para construir un sistema de archivos raíz, hará falta saber qué archivos se esperan encontrar en la distribución del sistema operativo que se está usando. En este caso, los sistemas de archivos que busca la distribución Red Hat 6.2 son */home*, */var*, */usr*. Para arrancar son especialmente delicados los archivos de dispositivo y los archivos que se encuentran en */sbin* y en */etc*. Al hacer una copia del sistema de archivos raíz que tiene el servidor, se tiene la estructura de archivos que busca el *Kernel*, y modificando sobre ésta algunos de los archivos de configuración que requiera el nodo se tiene la configuración apropiada para los nodos. En la distribución de *Etherboot*, hay enlaces a un par de *shell scripts* que se encargan de crear este tipo de sistema

de archivos raíz para un nodo, partiendo del sistema de archivos raíz del servidor.

El núcleo de Linux preparado para los nodos espera encontrar sistema de archivos raíz en */tftpboot/<nombre del nodo>*, por ejemplo: */tftpboot/uxmal11*

Al volver a arrancar el nodo se deberá ver cómo el núcleo monta un sistema de archivos raíz y sigue arrancando hasta presentar el *login* de usuario. Al inicio del nodo, habrá configuraciones que no correspondan al nodo, y mediante la adecuada sintonización de los nodos se corrige este problema.

Creación del sistema de archivos para los nodos.

Para que los nodos trabajen sólo con su memoria y tarjeta de red; es necesario crear un sistema de archivos que se compartan por medio de red, para que el nodo trabaje como si tuviera su propio disco duro. En el servidor, en */tftpboot*, se crea el sistema de archivos para cada nodo, dentro de un directorio que tiene por nombre el mismo que el nodo.

Para dicho objetivo se utilizó un primer *script* proporcionado por el paquete Etherboot que permite la copia del sistema de archivos básico del servidor. Y la creación de otros directorios que más tarde se utilizarán para montar o compartir:

```
#!/bin/sh
if [ $# != 1 ]
then
        echo Usar: $0 Direccion_IP_Cliente_o_Nombre
        exit 1
fi
cd /
umask 022
mkdir -p /tftpboot/$1

# Aqui solo se crean los siguientes directorios
for d in home mnt proc tmp usr
do
        mkdir /tftpboot/$1/$d
done
chmod 1777 /tftpboot/$1/tmp
touch /tftpboot/$1/fastboot
chattr +i /tftpboot/$1/fastboot

# Aquí se hace una copia de los sig. directorios
cp -a bin lib sbin dev etc root var /tftpboot/$1
cat <<EOF
Ahora, en /tftpboot/$1/etc, edita
        sysconfig/network
        sysconfig/network-scripts/ifcfg-eth0
        fstab

y configurar
        rc.d/rc3.d
EOF
```

3.3.3. CONFIGURACIÓN DEL KERNEL PARA LOS NODOS ESCLAVOS

Para tener acceso a la configuración del Kernel, utilizaremos la opción del menú de texto, al cual se accede por medio del comando:

$make menuconfig; dentro del directorio */usr/src/Linux*

Dentro de esta configuración del nuevo Kernel para los nodos, es recomendable desactivar las opciones que no sean necesarias para el trabajo de los nodos, tales como serían los dispositivos infrarrojos, el audio (si es que no se va a utilizar en los nodos), y otras opciones que se pueden consultar en el capítulo anterior en donde se da una explicación de las diversas opciones del Kernel que se pueden escoger.

Hay que recordar que de la misma manera en que en este momento se va a crear un Kernel que cumpla con nuestras necesidades, se puede hacer lo mismo en otra ocasión en que se necesite hacer modificaciones por nuevo hardware que se integre a cada nodo y se quiera que se reconozca directamente desde el Kernel.

Las opciones que son indispensables que estén presentes en el Kernel que se va dar a los nodos son las siguientes:

I. Network device support
  I.I. Ethernet 10 or 100 Mbit
    I.I.I. 3Com card
      Se cargan todas en el Kernel, no como módulos.

  I.II. Otras Tarjetas ISA
    I.II.I. NE2000/NE1000 support
    I.II.II. PCI NE2000

En el *cluster* se utilizan tarjetas 3Com, por lo que se activa su soporte dentro del Kernel.

Se pueden agregar también las tarjetas NE2000 para dar soporte a que se puedan agregar máquinas que tengan este tipo de NIC.

II. Networking Options
  II.I. TCP/IP Networking
  II.II. Ip: Kernel level autoconfiguration
    II.II.I. BOOTP support
    II.II.II. RARP support

Estas opciones proveen soporte para la pila de protocolos de TCP/IP y el soporte para poder iniciar de manera remota los clientes.

III. File System
    III.I. Network File System support
    III.II. Root file system on NFS

Proporciona soporte para compartir el sistema de archivos vía NFS por medio de la red y que inicie el montaje del sistema de archivos desde la raíz.

Se salva la configuración y se sale del menú.

Para compilar el Kernel, en la línea de comandos se escriben las siguientes instrucciones:

$make dep;make clean;make bzImage;make modules

Ya se tiene la imagen del sistema en */usr/src/Linux/arch/i386/boot* con el nombre de *bzImage*.

3.3.4. SERVIDOR NFS (NETWORK FILE SYSTEM, SISTEMA DE ARCHIVOS DE RED)

Como sabemos, la construcción del *cluster* está enfocada a un *cluster* con clientes (nodos) sin disco, por lo cual se utilizó un sistema de archivos de Red (NFS Network File System). NFS es un sistema que nos permite montar sistemas de archivos desde una computadora distinta sobre una red TCP/IP. Este tipo de sistema tiene muchas aplicaciones, pero sólo nos enfocamos en los sistemas de archivos que se comparten por este medio para los nodos sin disco. Para esto los clientes acceden a los archivos del servidor, montando los sistemas de archivos que comparte. En este caso el Servidor comparte los directorios:

/usr
/home
/tftpboot

a todas las máquinas cliente que pertenecen a la red 192.168.10.0 con máscara de red 255.255.255.0

Cuando el cliente monta el sistema de archivos remoto, no se hace una copia del sistema de archivos, en lugar de eso, el proceso de montaje usa una serie de Llamadas a Procedimientos Remotos (RPC, Remote Procedure Call) que habilita al cliente el acceso al sistema de archivos del disco del servidor, de forma transparente.

Configuración del servidor NFS

Para la configuración del servidor hay que seguir básicamente dos pasos:

El primer paso es crear el archivo */etc/exports* . Este archivo define qué partes del disco del servidor se comparten con el resto de la red, y las reglas mediante las cuales se comparten, es decir, lista las particiones compartidas, las máquinas con las que los comparte y con qué permisos

El formato de entrada de este archivo es el siguiente:

/dir/a/exportar *cliente1(permisos)* *cliente2(permisos)*

Donde:

I. */dir/a/exportar* :Es el directorio que se quiere compartir
II. *cliente1 cliente2*: Son las direcciones lógicas de los nodos clientes vía NFS
III. *(permisos)* : Son los permisos que corresponden a cada cliente

La siguiente tabla describe algunos de los permisos validos de cada cliente:

OPCIONES	SIGNIFICADO
secure	El número de puerto desde el cual el cliente realiza su petición debe ser menor de 1024. Este permiso es por defecto
ro	Permite acceso sólo de lectura sobre la partición.
noaccess	Al cliente se le denegará el acceso a todos los directorios por debajo de /dir/a/exportar. Esto le permite exportar el directorio /dir al cliente y después especificar /dir/a como inaccesible sin tener que dar acceso a algo como /dir/desde.
no_root_squash	Una simple medida de seguridad que provoca que el servidor ignore, por defecto, las peticiones hechas por el usuario root sobre una partición montada por NFS. Si se quiere deshabilitar esto y permitir que el usuario root de las máquinas clientes accedan al directorio montado por NFS, necesitará exportar el directorio con el permiso root_squash.
Squash_uids=lista-uid	Cuando se hace una petición de NFS, se incluye el UID del usuario que hace la petición. Esto permite que el servidor NFS haga cumplir los permisos. Se usa esta opción cuando se quiere restringir el acceso a la partición compartida de ciertos UID. La variable lista-uid es una lista separada por comas de los UID que quiere denegar. También se aceptan rangos, por ejemplo: squash_uids=5,10-15,20,23.
Squash_gips=lista-gid	Funciona como squash-uids, excepto que se aplica sobre GID en lugar de sobre UID
rw	Acceso de lectura/escritura normal.

Tabla 3. 2 Permisos validos pala los clientes especificados en el archivo *letc/export*.

Para la configuración de nuestro servidor NFS necesitamos:

- Exportar o compartir los directorios */usr*, */home*, */tftpboot*
- Los nombres de las máquinas cliente NFS, deben ser todas aquellas máquinas que pertenezcan a la red 192.168.10.0 con máscara 255.255.255.0 .
- Y los permisos serán que se necesitan básicamente son *rw* y *no_root_squash*

Por lo que el archivo /etc/exports de nuestro servidor es el siguiente:

/tftpboot 192.168.10.0/255.255.255.0(rw,no_root_squash)
/home 192.168.10.0/255.255.255.0(rw,no_root_squash)
/usr 192.168.10.0/255.255.255.0(rw,no_root_squash)

Después es necesario que el servidor NFS lea el archivo, para ello debemos iniciar o reiniciar el servicio de NFS, ya sea reiniciando la máquina o por medio de la línea
#/etc/rc.d/init.d/nfsd start o restart ; según sea el caso.

CONFIGURACIÓN DEL CLIENTE

Los clientes NFS son fáciles de configurar bajo Linux debido a que no requieren ningún software nuevo o adicional para cargarse. El único requerimiento es que el Kernel sea compatible con el soporte NFS, y como al compilar el nuevo Kernel para los nodos se activó esta opción entonces la imagen del Kernel para nuestros nodos cumplen con este requerimiento.

Para montar un sistema de archivos, como ya se explicó en el capítulo anterior, es usando el comando *mount* y por medio del archivo */etc/fstab*

Para este caso se usa el archivo */etc/fstab*, el cual permite que se monten los sistemas de archivos remotos al arranque (automáticamente).

Éste es el formato de las entradas del archivo */etc/fstab*

/dev/dispositivo /directorio/a/montar fstype parámetros fs_freq fs_passno

En la Tabla 3.3 presentamos cada elemento en una entrada del */etc/fstab*.

Entradas del archivo /etc/fstab	Descripción
/dev/dispositivo	La partición que está montada (por ejemplo, /dev/hda3).
/directorio/a/montar	El directorio donde está montada la partición
fstype	El tipo de sistema de archivos
Parámetros	Los parámetros proporcionados con la opción -o del comando mount.
Fs_freq	Dice al comando dump con cuanta frecuencia se necesita hacer copia de seguridad del sistema de archivos
Fs_passno	Llama al programa fsck en tiempo de arranque para determinar el orden en el cual se comprueban los sistemas de archivos.

Tabla 3. 3

En el archivo *fstab* del nodo se declaran las siguientes líneas.

192.168.10.1:/tftpboot	*/*	*nfs*	*defaults*	*0 0*
192.168.10.1:/usr	*/usr*	*nfs*	*defaults*	*0 0*
192.168.10.1:/home	*/home*	*nfs*	*defaults*	*0 0*
none	*/proc*	*proc*	*defaults*	*0 0*
none	*/dev/pts*	*devpts*	*gid=5,mode=620*	*0 0*

Donde la primera columna especifica la partición o directorio que se monta desde el servidor remotamente, las particiones montadas son */tftpboot*, */usr* y */home* que corresponden al servidor, a quien le corresponde la IP 192.168.10.1. La segunda columna especifica el directorio donde está montada la partición o sistema de archivo (punto de montaje). La tercera columna nos dice el tipo de sistema de archivos, en este caso es *nfs* que nos indica que monta particiones o directorios desde sistemas remotos. La cuarta columna es una opción de montaje, en la cual colocamos default. En las dos últimas columnas los valores son cero debido a que no se necesitan hacer copias de seguridad ni comprobar los sistemas de archivos, ya que como los sistemas de archivos están en el servidor, éste hace la comprobación necesaria para cada uno.

3.3.5. CONFIGURACIÓN DHCP (DYNAMIC HOST CONFIGURATION PROTOCOL)

Esta herramienta es usada en el *cluster* para que el servidor asigne una dirección IP a los nodos que conforman el cluster. Cada máquina cliente (nodo) se configura para que obtenga una dirección IP de la red. Cuando el cliente inicia utiliza como dirección IP la 0.0.0.0 y envía una petición de *broadcast* (a la dirección 255.255.255.255, de la cual el servidor puede escuchar a los *host* que estén conectados en la red) por la red para una dirección IP. Entonces el servidor DHCP (que está ejecutando el demonio *dhcpd*) comprueba su base de datos local y responderá a esa petición, asignándole su correspondiente IP también en esa respuesta se puede incluir servidores de nombres, una máscara de red. A continuación se presenta el esquema básico de funcionamiento.

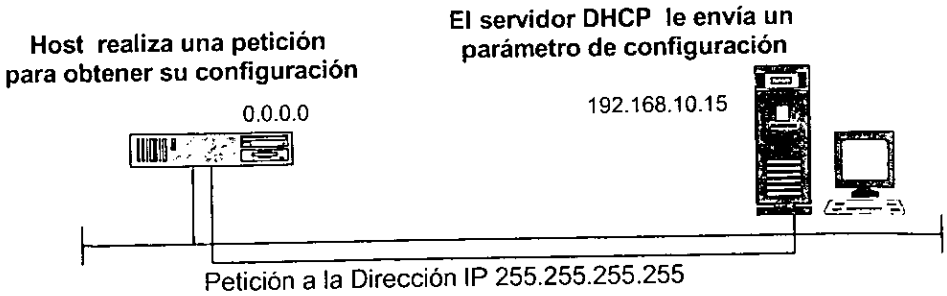


Figura 3. 3 Esquema básico del funcionamiento del DHCP.

Configuración del servidor DHCP

El archivo de configuración primario del servidor DHCP es por defecto */etc/dhcpd.conf.* Este archivo de configuración engloba básicamente dos ideas:

Declaraciones

Un conjunto de declaraciones de redes, máquinas o grupos unidos al sistema y posiblemente un rango de direcciones que se conceden a cada entidad. Se pueden usar varias declaraciones para describir varios grupos de clientes. Las declaraciones también se pueden anidar a otra cuando se necesiten varios conceptos para describir un conjunto de clientes o máquinas.

Parámetros

Un conjunto de parámetros que describen el comportamiento del servidor y configuran las respuestas apropiadas. Los parámetros pueden ser globales, o locales al conjunto de declaraciones.

El archivo de configuración */etc/dhcpd.conf,* es un archivo de texto y tiene la siguiente estructura:

Parámetros globales;
Declaración1 {
[parámetros relacionados con declaración1]
[Subdeclaración anidada]
}

Declaración2 {
[parámetros relacionados con declaración2]
[Subdeclaración anidada]
}

....

Para la configuración de nuestro servidor, el archivo de configuración utilizó los siguientes parámetros globales:

Server-Identifier: La sintaxis de este parámetro es la siguiente:
server-identifier *nombre-máquina;*

Parte de la respuesta DHCP es la dirección del servidor. En sistemas con varias interfaces, el servidor DHCP concede la dirección a la primera interfaz. Desafortunadamente, esta interfaz no se puede alcanzar por todos los clientes de un servidor o del ámbito de la declaración. En estos casos raros, este parámetro se puede usar para enviar la IP a la interfaz apropiada por la que el cliente se comunica con el servidor.

Option domain-name: Esta opción da el nombre del dominio que de debe usar como nombre de dominio local; la sintaxis es la siguiente:

Option domain-name "dominio";

Option domain-name-servers: Especifica cuáles son los servidores de nombres; su sintaxis es la siguiente:

Option domain-name-servers " dirección IP ";

Option broadcast-address: Especifica una dirección de una subred de cliente se especifica como dirección de *broadcast.*

Option host-name: La cadena usada para identificar el nombre del cliente (nodo). Especifica el nombre del *host.*

Option host-name "string"

En lo que concierne al conjunto de declaraciones de nuestro archivo de configuración tenemos las siguientes:

Subnet: Esta sentencia se usa para dar parámetros específicos de una subred. Además permite al servidor saber si una dirección dada pertenece o no a dicha subred. La declaración *subnet* se utiliza para aplicar un conjunto de parámetros y/o declaraciones a un conjunto de direcciones que coincidan con la descripción de esta declaración. La sintaxis es la siguiente:

subnet dirección-de-la-subred *netmask* máscara-de-red{
[Parámetros]
[sub_Declaraciones]
}

Shared-network: Esta sentencia se utiliza para especificar que varias subredes comparten físicamente la misma red. Los parámetros y declaraciones que se introduzcan aquí afectarán a todas las subredes que englobe.

Una declaración *shared-network* agrupa un conjunto de direcciones de miembros de una misma red física. Esto permite agrupar parámetros y declaraciones para propósitos administrativos. La sintaxis es:

shared-network etiqueta {
[parámetros]
[subdeclaraciones]

}

La *etiqueta* es el nombre definido por el usuario para la red compartida. *Parámetros* y *subdeclaraciones* son parte del conjunto de parámetros y declaraciones.

Podemos utilizar esta sentencia para dar los parámetros específicos de una subred, además esta sentencia le permite al servidor de *DHCP* saber si una dirección dada pertenece o no a dicha subred.

host: Una declaración *host* se usa para aplicar un conjunto de parámetros y declaraciones a una máquina en particular, además de los parámetros indicados al grupo. Se usa normalmente para fijar la dirección de arranque, o para clientes *BOOTP*. Deberá aparecer una sentencia *host* por cada equipo que deseamos que obtenga su dirección mediante *DHCP o BOOTP*, a no ser que definamos en el rango de direcciones que los clientes de *BOOTP* también obtengan sus direcciones *IP* de forma dinámica. La sintaxis de una declaración *host* es la siguiente:

host etiqueta {
[parámetros]
[subdeclaraciones]
}

La *etiqueta* es el nombre definido por el usuario para identificar a la máquina.

fixed-address: Este parámetro aparece sólo bajo la declaración *"host"*. Se utiliza para especificar la dirección que le corresponde a un host.
La sintaxis de este parámetro es la siguiente:

fixed-address dirección;

Hardware:

hardware tipo-de-hardware dirección-hardware.

Donde *tipo-de-hardware* puede ser o bien ethernet o bien token-ring y *dirección-hardware* es el identificador de la tarjeta de red y es un conjunto de octetos en hexadecimal (MAC).

Esta sentencia es necesaria para que los clientes que soliciten su configuración mediante bootp puedan obtener los datos correspondientes a la interfaz cuya dirección hardware es la especificada.

Filename: Esta sentencia se utiliza cuando el cliente(nodo) DHCP necesita saber el nombre de un archivo para usarlo en el arranque. Se combina con *next-server* para recuperar un archivo remoto para configurar la instalación o arrancar un cliente sin disco.

filename "archivo";

Next-server: Especifica el nombre o dirección del servidor que contiene el fichero de arranque (aquél que se especificó con la sentencia *filename*),
La sintaxis de este parámetro es la siguiente:

next-server nombre-servidor;

Después de haber creado el archivo *dhcpd.conf*, hace falta crear el archivo *dhcpd.leases* como root.

#touch /etc/dhcpd.leases

y empezar el demonio *dhcpd*

#/etc/rc.d/init.d/dhcpd restart

El archivo de configuración *dhcpd.conf* del servidor presenta la siguiente estructura:

```
server-identifier etzna;

option domain-name "fi-a.unam.mx";
option domain-name-servers 132.248.10.2;
        subnet 132.248.54.0 netmask 255.255.255.0 {
        }

shared-network SERVIDOR {
        subnet 192.168.10.0 netmask 255.255.255.0 {
                option broadcast-address 192.168.10.255;
        }
}

host uxmal10 {
        fixed-address 192.168.10.10;
        hardware ethernet 00:04:76:2A:D5:52;
        filename "/tftpboot/vmlinuz.boot";
        next-server 192.168.10.1;
        option host-name "uxmal10";
}

host uxmal11 {
        fixed-address 192.168.10.11;
        hardware ethernet 00:20:AF:4D:A6:37;
        filename "/tftpboot/vmlinuz.boot";
        next-server 192.168.10.1;
        option host-name "uxmal11";
}
...
```

Es importante que cada vez que se cree un nuevo nodo se agregue otra entrada *host* con los datos propios del nodo en este archivo.

Configuración DHCPD nodos(cliente)

Para configurar los nodos hay que abrir el archivo

/tftpboot/nodo_n/etc/sysconfig/network-scripts/ ifcfg-eth0

En lugar de proporcionar la dirección IP se especifica que ésta será proporcionada por un servidor DHCP, esto lo logramos modificando la línea

IPADDR="dhcp"

3.3.6. CONFIGURACIÓN DE RED EN EL NODO.

Como la configuración de red se hereda del servidor, es necesario ajustarla a las necesidades del nodo, por lo que es necesario modificar los siguientes archivos según lo visto anteriormente:
/tftpboot/nodo/etc/sysconfig/network:

```
NETWORKING=yes
HOSTNAME=""
GATEWAY="192.168.10.0"
GATEWAYDEV="eth0"
FORWARD_IPV4="no"
NISDOMAIN="cluster.beowulf"
```

/tftpboot/nodo/etc/sysconfig/network-scripts/ifcfg-eth0 :

```
DEVICE="eth0"
IPADDR="dhcp"
NETMASK="255.255.255.0"
ONBOOT="yes"
BOOTPROTO="none"
BROADCAST=192.168.10.255
NETWORK=192.168.10.0
USERCTL=no
```

Y para que inicie el nodo en modo texto modificar en el archivo

/tftpboot/nodo/etc/inittab

la línea que da el arranque por default.

```
# Default runlevel. The runlevels used by RHS are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
id:3:initdefault:
```

Para este trabajo no es necesario utilizar el ambiente gráfico en los nodos, y por esto se da el inicio en el modo de texto con soporte para red (nivel de ejecución 3).

3.3.7. SERVIDOR NIS.

Los servicios de NIS son simplemente una base de datos en un esquema cliente – servidor que los clientes pueden consultar. Consta de una serie de tablas independientes. Cada tabla se origina como un archivo de texto, como el */etc/passwd*, el cual tiene una naturaleza tabular y tiene al menos una columna que es la única por cada archivo (lo que permite generar una base de datos de valores *clave/par*). NIS accede a esas tablas por nombre y permite búsquedas de dos maneras:

- Listado de la tabla
- Presentación de una entrada específica basada en una búsqueda por una clave dada.

Una vez establecidas las bases de datos en el servidor, los clientes pueden buscar en el servidor las entradas de la base de datos. Normalmente esto ocurre cuando un cliente se configura para que busque en un mapa NIS cuando no encuentra una entrada en su base de datos local. Una máquina puede tener sólo las entradas que necesite para que el sistema trabaje en modo usuario único (cuando no hay conectividad por red), por ejemplo, con el archivo */etc/passwd*. Cuando un programa hace una petición de búsqueda de la información de contraseña de un usuario, el cliente comprueba su archivo *passwd* local y si el usuario no existe allí, entonces el cliente hace la petición a su servidor NIS para buscar la entrada correspondiente en la tabla de contraseñas. Si NIS tiene la entrada, volverá al cliente y al programa que pidió la información. El propio programa no se da cuenta de que ha usado NIS. Lo mismo es cierto si el mapa NIS devuelve la respuesta de que la entrada de contraseña del usuario no existe. El programa debería recibir la información sin que sepa qué ha ocurrido.

Esto se aplica a todos los archivos que digamos que se compartan por NIS.

Dentro de nuestro trabajo, NIS nos proporcionara la facilidad de poder tener una administración centralizada para la configuración de ciertos archivos para todas las máquinas que se encuentran en nuestra red.

Los archivos de configuración sobre los cuales se generan los mapas son:

/etc/group, /etc/hosts, /etc/netgroup, /etc/networks, /etc/passwd, /etc/protocols, /etc/rpc, /etc/services

Los mapas pueden generarse de diferentes maneras al ordenarlos por un registro diferente cada uno. Así, para el archivo *passwd*, se pueden generar los siguientes mapas:

passwd.byname (ordenado por nombre)
passwd.bynumber (ordenado por número)
passwd.byuid (ordenado por UID de usuario)

NIS puede tener un único servidor autorizado donde permanecen los archivos de datos originales. Este servidor autorizado se llama servidor NIS maestro. Si la carga en la red es muy grande, se puede distribuir en varios servidores NIS secundarios o esclavos.

El servidor NIS primario establece dominios que abarcan el área en que trabajan. El servidor de NIS sólo envía los datos que le pide el cliente, pero no realiza autenticación, de ese paso se encargan las máquinas individualmente.

Las características que presentan los dominios en NIS son las siguientes:

- Todas las máquinas pertenecientes a un dominio tienen el mismo nombre de dominio.
- Un dominio NIS tiene un único servidor maestro.
- Un dominio de NIS puede tener o no servidores esclavos.
- Los mapas con igual nombre pero distinto dominio son diferentes.
- El dominio de NIS es diferente a los dominios que se definen en Internet.
- Los dominios suelen aparecer en los sistemas como subdirectorios del directorio donde se localiza *yp*.

Configuración del servidor NIS maestro.

Las distribuciones de Linux normalmente vienen con NIS ya compilado e instalado. En ese caso sólo se deberá de habilitar el servicio y asegurarse que *ypserv* es parte del proceso de arranque de los niveles de ejecución 3 y 5.

Una vez que se ha instalado NIS, se necesita configurarlo. Este proceso se puede dividir en cuatro pasos:

I. Establecer el nombre del dominio
II. Arrancar el demonio *ypserv* que inicia NIS
III. Editar el archivo *Makefile*
IV. Ejecutar *ypinit* para crear la base de datos

I. Establecer el nombre del dominio.

La forma de definir un dominio NIS es con el comando *domainname*:

domainname dominio

Donde "dominio" es el dominio NIS empleado. Es necesario que cada vez que se rearranque el sistema, el dominio se establezca. Se puede añadir la siguiente línea al archivo */etc/sysconfig/network*:

NIS_DOMAIN=nombre.de.dominio

La configuración del nombre de dominio debe de ocurrir antes que se inicien los servicios del NIS.

El nombre del dominio que utilizamos lleva la denominación *cluster.beowulf.*

II. Arranque de *ypserv.*

EL programa *ypserv* es el responsable del manejo de las peticiones que se hacen hacia el servidor NIS. Para no reiniciar el servidor, es posible reiniciar el demonio de *ypserv* por medio de los comandos:

$/etc/rc.d/init.d/ypserv start

III. Edición del archivo *Makefile*

Para que el NIS funcione debidamente, es necesario crear los mapas de los cuales se servirá para realizar su función. Para esto, en el directorio */var/yp* se encuentra un archivo llamado *Makefile*. Este archivo lista los archivos que se compartirán mediante NIS, así como algunos parámetros adicionales sobre cómo compartirlos.

Algunas de las opciones que se presentan dentro de este archivo son:

UID y GID mínimos.

En esta sección, se pueden delimitar los UID y GID mínimos que se manejarán dentro de los mapas de NIS.

```
# We do not put password entries with lower UIDs (the root and system
# entries) in the NIS password database, for security. MINUID is the
# lowest uid that will be included in the password maps.
# MINGID is the lowest gid that will be included in the group maps.
MINUID=500
MINGID=500
```

En los comentarios que aparecen dentro del archivo, se nos menciona que no es conveniente colocar como UID mínimo el correspondiente a root ni los correspondientes a la administración del sistema. Esto es para que por medio de los mapas globales no se comparta el *password* de root, y sólo sean los mapas locales los que conozcan esta contraseña. Algunos otros UID referentes a la administración del sistema no se colocan por la misma razón, y por esto se define que el UID mínimo será

igual a 500, que es desde donde comienzan los UID de los usuarios del sistema.

Mezcla de contraseñas ocultas con contraseñas reales.

Si usa contraseñas shadow (*shadow password*), NIS automáticamente lo manejará tomando el campo encriptado del archivo */etc/shadow* y mezclándolo con la copia compartida de NIS del archivo */etc/passwd*. A menos que haya una razón específica por la que no se quiera activar la compartición de contraseñas encriptadas, *MERGE_PASSWD* se dejará a su valor por defecto. En nuestro caso, esta opción se dejó en su valor original.

```
# Should we merge the passwd file with the shadow file ?
# MERGE_PASSWD=true|false
MERGE_PASSWD=true
```

Mezcla de contraseñas de grupo ocultas con grupos reales.

De la misma forma en que el archivo */etc/passwd* permite contraseñas ocultas *(/etc/shadow)*, el archivo */etc/group* permite contraseñas de este tipo. Si se tiene un archivo *shadow* para las contraseñas de grupo, será necesario configurar *MERGE_GROUP* al valor de *false*.

En nuestro caso, no es necesario utilizar el *shadow* para los grupos, así que su valor se deja en *false*.

```
# Should we merge the group file with the gshadow file ?
# MERGE_GROUP=true|false
MERGE_GROUP=false
```

Designación de nombres de archivos

Dentro del *Makefile* se listan los archivos que se preconfigurarán para compartirse mediante NIS. Sin embargo, que se listen aquí no significa que se compartirán automáticamente. Este listado simplemente establece los nombres de archivos que más tarde se usarán al ejecutar el *Makefile*.

La mayoría de las entradas empiezan con *$(YPPWDDIR)* que es una variable que se configura antes de esta sección en el *Makefile*. El valor por defecto de esta variable es */etc*, lo cual significa que cualquier ocurrencia de *$(YPPWDDIR)* se sustituye por */etc* cuando se ejecute el *Makefile*. Así la entrada *GROUP* se convierte en */etc/group*, *PASSWD* convierte en */etc/passwd*, etc. La variable *$(YPSRCDIR)* también se configura hacia la ruta */etc*.

```
# These are the source directories for the NIS files; normally
# that is /etc but you may want to move the source for the password
# and group files to (for example) /var/yp/ypfiles. The directory
```

```
# for passwd, group and shadow is defined by YPPWDDIR, the rest is
# taken from YPSRCDIR.
#
YPSRCDIR = /etc
YPPWDDIR = /etc
YPBINDIR = /usr/lib/yp
YPSBINDIR = /usr/sbin
YPDIR = /var/yp
YPMAPDIR = $(YPDIR)/$(DOMAIN)

# These are the files from which the NIS databases are built. You may edit
# these to taste in the event that you wish to keep your NIS source files
# seperate from your NIS server's actual configuration files.
#
GROUP      = $(YPPWDDIR)/group
PASSWD     = $(YPPWDDIR)/passwd
SHADOW          = $(YPPWDDIR)/shadow
GSHADOW    = $(YPPWDDIR)/gshadow
ADJUNCT    = $(YPPWDDIR)/passwd.adjunct
#ALIASES    = $(YPSRCDIR)/aliases  # aliases could be in /etc or /etc/mail
ALIASES    = /etc/aliases
ETHERS     = $(YPSRCDIR)/ethers     # ethernet addresses (for rarpd)
BOOTPARAMS = $(YPSRCDIR)/bootparams # for booting Sun boxes (bootparamd)
HOSTS      = $(YPSRCDIR)/hosts
NETWORKS   = $(YPSRCDIR)/networks
PRINTCAP   = $(YPSRCDIR)/printcap
PROTOCOLS  = $(YPSRCDIR)/protocols
PUBLICKEYS = $(YPSRCDIR)/publickey
RPC        = $(YPSRCDIR)/rpc
SERVICES   = $(YPSRCDIR)/services
NETGROUP   = $(YPSRCDIR)/netgroup
NETID      = $(YPSRCDIR)/netid
AMD_HOME   = $(YPSRCDIR)/amd.home
AUTO_MASTER = $(YPSRCDIR)/auto.master
AUTO_HOME  = $(YPSRCDIR)/auto.home
YPSERVERS = $(YPDIR)/ypservers      # List of all NIS servers for a domain
```

Lo que se comparte.

Al final de la parte de configuración del archivo *Makefile*, existe una entrada que lleva la etiqueta *all*. Después de esta etiqueta se mencionarán todos los mapas que se comparten. Si no se quiere compartir algunos mapas, basta con anteponer un signo de # antes del nombre del mapa:

```
If you don't want some of these maps built, feel free to comment
# them out from this list.

all:  passwd group hosts rpc services netid protocols netgrp mail \
# shadow publickey # networks ethers bootparams printcap \
# amd.home auto.master auto.home passwd.adjunct
```

En nuestro caso, se eliminaron los servicios netgrp y mail, ya que no son necesarios.

IV. Uso de ypinit

Una vez que se tenga preparado el *Makefile* se inicializa el servidor YP (NIS) usando el comando *ypinit*:

/usr/lib/yp/ypinit -m

donde la opción *-m* le dice a *ypinit* que configure el sistema como servidor de NIS maestro. Ejecutándolo sobre el dominio *cluster.beowulf*, presenta una salida de la siguiente manera.

```
[root@etzna /root]# /usr/lib/yp/ypinit -m
At this point, we have to construct a list of the hosts which will run NIS
servers.  etzna 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:  etzna
        next host to add:

The current list of NIS servers looks like this:

etzna

Is this correct?  [y/n: y] y
We need some  minutes to build the databases...
Building /var/yp/cluster.beowulf/ypservers...
Running /var/yp/Makefile...
gmake[1]: Entering directory `/var/yp/cluster.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 netid.byname...
Updating protocols.bynumber...
Updating protocols.byname...
Updating mail.aliases...
gmake[1]: Leaving directory `/var/yp/cluster.beowulf'
[root@etzna /root]#
```

Si se presenta uno o varios errores que muestren la siguiente información:

```
[root@etzna /root]# /usr/lib/yp/ypinit -m At this point, we have to
construct a list of the hosts which will run NIS servers.  etzna 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:  etzna
      next host to add: The current list of NIS servers looks like this:

etzna

Is this correct?  [y/n: y] y
```

We need some minutes to build the databases...
Building /var/yp/cluster.beowulf/ypservers...
Running /var/yp/Makefile... gmake[1]: Entering directory `/var/yp/cluster.beowulf'
Updating passwd.byname...
failed to send 'clear' to local ypserv: RPC:Programa no registradoUpdating passwd.byuid...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating group.byname...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating group.bygid...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating hosts.byname...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating hosts.byaddr...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating rpc.byname...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating rpc.bynumber...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating services.byname...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating netid.byname...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating protocols.bynumber...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating protocols.byname...
failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating mail.aliases...
failed to send 'clear' to local ypserv: RPC: Programa no registrado
*gmake[1]: *** No rule to make target `/etc/netgroup', needed by*
`netgroup'. Stop. gmake[1]: Leaving directory `/var/yp/cluster.beowulf'
*make: *** [target] Error 2 Error running Makefile. Please try it by hand.*

etzna has been set up as a NIS master server.

Now you can run ypinit -s etzna on all slave server.
[root@Etzna /root]

nos indica dos cosas:

I. Los errores del tipo

failed to send 'clear' to local ypserv: RPC: Programa no registradoUpdating mail.aliases...
failed to send 'clear' to local ypserv: RPC: Programa no registrado

indican que el servidor de NIS no ha sido iniciado.

II. Los mensajes del tipo:

*gmake[1]: *** No rule to make target `/etc/netgroup', needed by*
`netgroup'. Stop. gmake[1]: Leaving directory `/var/yp/cluster.beowulf'

indican que se ha especificado un archivo a compartir que no existe. Se puede crear el archivo o volver a editar el archivo *Makefile* para no compartir el archivo, como se menciona en la sección "lo que se comparte".

Configuración de un cliente NIS

La configuración de los clientes es más sencilla que la del servidor. Los pasos a seguir en general serían los siguientes:

I. Editar el archivo */etc/yp.conf*

II. Configurar los servicios de arranque

I. Edición del archivo */etc/yp.conf*

El archivo */etc/yp.conf* contiene la información necesaria para que el demonio del cliente, *ypbind*, arranque y encuentre el servidor NIS. El cliente puede encontrar al servidor de diferentes maneras:

- El uso de *broadcast*
- La especificación de nombre de la máquina del servidor.

La configuración por *broadcast* es apropiada para cuando un cliente se mueve a varias subredes. En nuestro caso, las máquinas van a estar en una misma red y no va a haber desplazamiento del cliente.
La otra opción busca de manera ya predeterminada el servidor que le dará servicio de NIS.

Para utilizar esta última configuración, se edita el archivo */etc/yp.conf* y se le agrega la siguiente línea:

ypserver nombre-servidor

Ejemplo:

```
# /etc/yp.conf - ypbind configuration file
# Valid entries are
#
#domain NISDOMAIN server HOSTNAME
#       Use server HOSTNAME for the domain NISDOMAIN.
#
#domain NISDOMAIN broadcast
#       Use  broadcast  on  the local net for domain NISDOMAIN
#
#ypserver HOSTNAME
#       Use server HOSTNAME for the  local  domain.  The
#       IP-address of server must be listed in /etc/hosts.
#
ypserver 192.168.10.1
```

Donde *nombre-dominio* es el nombre de su dominio de NIS y *nombre-servidor* es el nombre de su servidor NIS al que hace referencia el cliente.

II. Configurar los servicios de arranque

El *ypbind* es el demonio que corre en los clientes NIS, pero no debemos olvidar que todo servidor NIS es cliente, porque necesita acceder a la información de los mapas, así que todas las máquinas del dominio deben correr el *ypbind*. Este demonio determina a qué servidor debe conectarse la máquina cliente para realizar sus peticiones de información.

Normalmente, éste se inicia en el script de arranque */etc/rc.d/init.d/ypbind*. Se debe de comprobar que se inicia este servicio tanto en el nivel de inicio 3 como en el nivel de inicio 5.

Actualización de mapas

Una vez que se tienen configurados el servidor y el cliente, se puede probar que funcionan correctamente por medio de la utilidad *ypcat*, que mostrara a pantalla el mapa que se comparte. Para esto se debe de escoger un mapa que se comparta en el dominio, como por ejemplo, *passwd*.

Una vez que se tiene todo esto configurado, al dar de alta a un usuario nuevo, se hará directamente desde el servidor, y ya no nodo por nodo. Para actualizar todos los mapas con los nuevos datos, solo se deberá ejecutar *make* dentro del directorio */var/yp* con lo que se actualizarán todos los mapas que se comparten.

3.3.8. SECURE SHELL

¿Que es Secure Shell?

Secure Shell (*ssh*) es un programa que permite realizar conexiones entre máquinas a través de una red abierta de forma segura, así como ejecutar programas en una máquina remota y copiar archivos de una máquina a otra, como se muestra en la siguiente figura.

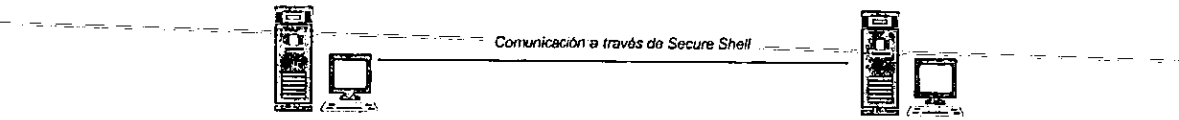


Figura 3. 4 Esquema comunicación del Secure Shell.

El SSH (Secure Shell) se define como un programa para conectarse a otros equipos a través de una red, para ejecutar comandos en una máquina remota y para mover archivos de una máquina a otra. Proporciona una exhaustiva autenticación y comunicaciones seguras en redes no seguras.

Ssh provee fuerte autenticación y comunicación segura sobre un canal inseguro y nace como un reemplazo a los comandos *telnet, ftp, rlogin, rsh*, y *rcp*, los cuales proporcionan gran flexibilidad en la administración de una red, pero sin embargo, presenta grandes riesgos en la seguridad de un sistema. Adicionalmente, *ssh* provee seguridad para conexiones de servicios X Windows y envío seguro de conexiones arbitrarias TCP.

Secure Shell admite varios algoritmos de cifrado entre los cuales se incluyen:

- Blowfish
- 3DES
- IDEA
- RSA

La ventaja más significativa de *ssh* es que no modifica mucho las rutinas. En todos los aspectos, iniciar una sesión de *ssh* es tan sencillo como(y similar a) iniciar una sesión de *telnet*. Tanto el intercambio de llaves, la autenticación, así como el posterior cifrado de sesiones son transparentes para los usuarios.

¿De qué previene Secure Shell?

Debido a la promiscuidad de la interfaz Ethernet, se genera una problemática sobre los siguientes servicios de red usados en la actualidad, tales como:

- telnet
- ftp
- http
- rsh
- rlogin
- rexec

Ello nos representa un problema importante, ya que, incluso en un entorno de red cerrado, debe existir como mínimo un medio seguro para poder desplazar archivos, hacer copia de archivos, establecer permisos, ejecutar archivos, scrips, etc., a través de medios seguros.

Por ello para evitar que determinadas personas capturen el tráfico diario de la red, es conveniente instalar el Secure Shell(SSH).

Entre los ataques más comunes que nos previenen Secure Shell están:

- Sniffering(Captura de tráfico)
- IP Spoofing
- MAC Spoofing
- DNS Spoofing
- Telnet Hickjacking
- ARP Spoofing
- IP Routing Spoofing
- ICMP Spoofing

Protocolos de Secure Shell

Existen actualmente dos protocolos desarrollados sobre *ssh*:

- SSH1: La última versión de *ssh* cliente/servidor para Unix que soporta este protocolo es la 1.2.31, ésta puede ser utilizada libremente para propósitos no comerciales y es ampliamente usada en ambientes académicos.

- SSH2: Provee licencias más estrictas que SSH1 ya que es de carácter comercial. La última versión de *ssh* cliente/servidor para Unix con este protocolo es la 2.4.0 y puede ser utilizada libremente respetando la licencia expresa.

Actualmente existe un proyecto llamado *OpenSSH*, el cual fue desarrollado inicialmente dentro del proyecto *OpenBSD*.

OpenSSH es una versión libre de los protocolos *SSH/SecSH* bajo licencia BSD y es totalmente compatible con los protocolos *SSH1* y *SSH2*.

Debido a que *OpenSSH* rompe la barrera de los protocolos que ha causado confusión entre diversos sectores, está herramienta esta siendo muy usada en la comunidad, tal es el caso de distribuciones como Linux que ya la incluyen dentro de su sistema operativo.

En el caso del *cluster* se realizó la instalación de *OpenSSH*, por el soporte que ofrece a ambos protocolos.

Instalación de OpenSSH

El proyecto *OpenSSH* incluye al programa *ssh*. De igual forma incluye el archivo *sshd* que es el demonio que controla del lado del servidor las peticiones realizadas, también incluye otras funciones básicas del *secure shell*, tales como *ssh-add, ssh-agent* y *ssh-keygen*.

Openssh hoy en día soporta ambos protocolos de comunicación segura como lo son los protocolos 1.3, 1.5 y 2.0.

Debido a que *OpenSSH* es desarrollado fuera de los Estados Unidos, usando código de diversos países es libre de uso sin restricciones bajo la licencia del tipo BSD. Hoy en día *OpenSSH* corre en diversas plataformas entre las cuales destacan:

- OpenBSD
- Linux
- Solaris
- AIX
- IRIX
- HP/UX
- FreeBSD
- NetBSD

Para poder configurar de manera apropiada el *OpenSSH* se deben cumplir con tres prerrequisitos básicos para una óptima configuración.

I. Instalación de Zlib
II. Instalación de OpenSSL
III. Instalación de OpenSSH

I. Instalación de Zlib

Zlib es un conjunto de librerías necesarias para el funcionamiento de *SSH* y *OpenSSH* no es la excepción al ejecutarlo.

Zlib se configura y se instala casi en todas las plataformas sin problema alguno, y para poderlo implementar se deben aplicar los siguientes pasos:

Después de obtener el programa de *Zlib* procedemos a desempacarlo:

```
# gunzip zlib.tar.gz
# tar -xvf zlib.tar
```

En este punto obtendremos un directorio *zlib-x.x.x* sobre la ruta donde desempacamos, ejecutaremos los siguientes comandos.

a) Configuración del entorno de compilación.

Dentro del directorio *zlib-x.x.x* se encuentra el script llamado *configure*, el cual lo indicaremos para que automáticamente nos detecte la configuración y plataforma de nuestro sistema.

```
# ./configure
```

b) Compilación de Zlib

Paso siguiente es realizar la compilación de las librerías de *Zlib* simplemente tecleando *make*.

```
# make
```

c) Instalación de *zlib*

Si no se presentó problema alguno el paso siguiente es como superusuario teclear la sentencia apropiada para la instalación de las librerías.

```
# make install
```

Si todo funcionó sin presentar errores procedemos a la instalación del software *OpenSSL*, descrito a continuación.

II. Instalación de OpenSSL

La última versión disponible del *OpenSSL* es la Versión 0.9.6.

OpenSSL se configura y se instala casi en todas las plataformas sin problema alguno, y para poderlo implementar se deben aplicar los siguientes pasos:

Después de obtener el programa de *OpenSSL* procedemos a desempacarlo:

```
# gunzip openssl-0.9.6.tar.gz
# tar -xvf openssl.0.9.6.tar
```

En este punto obtendremos un directorio *openssl-0.9.6* sobre la ruta donde desempacamos ejecutaremos los siguientes comandos.

a) Configuración del entorno de compilación.

Dentro del directorio *openssl-0.9.6* se encuentra el script llamado *config*, el cual automáticamente nos detectará la configuración y plataforma de nuestro sistema.

```
# ./config
```

b) Configuración de acuerdo a la plataforma

Una vez detectada la plataforma del sistema, procedemos a la configuración global de nuestra herramienta, bastará con solo teclear lo siguiente:

```
# ./configure PLATAFORMA
```

Donde Plataforma es la plataforma que *config*, previamente nos había detectado. Por ejemplo si *config* me detectó lo siguiente:

```
# ./config

Configuring for solaris-sparc-gcc
```

Significa que la plataforma a configurar es *solaris-sparc-gcc*, por lo que tecleariamos lo siguiente:

```
# ./configure solaris-sparc-gcc
```

c) Compilación de *OpenSSL*

Paso siguiente es realizar la compilación de las librerías de *OpenSSL* simplemente tecleando *make*

```
# make
```

d) Instalación de *OpenSSL*

Si no se presentaron problema alguno el paso siguiente es como superusuario teclear la sentencia apropiada para la instalación de las librerías.

```
# make install
```

Si la instalación se realizo sin errores procedemos a la instalación del software *OpenSSH*, descrito a continuación.

III. OpenSSH

Una vez realizada la instalación de *zlib* y *OpenSSL*, y si no se presentó problema alguno se procede a la instalación del software *OpenSSH*.

Instalación de OpenSSH

OpenSSH se configura y se instala casi en todas las plataformas sin problema alguno, y para poderlo implementar se deben aplicar los siguientes pasos:

Después de obtener el programa de OpenSSH procedemos a desempacarlo:

```
# gunzip openssh-x.x.x.tar.gz
# tar -xvf openssh-x.x.x.tar
```

En este punto obtendremos un directorio *openssh-x.x.x* sobre la ruta donde desempacamos; ejecutaremos los siguientes comandos.

a) Configuración del entorno de compilación.

Dentro del directorio *openssh-x.x.x* se encuentra el *script* llamado configure, el cual lo indicaremos para que automáticamente nos detecte la configuración y plataforma de nuestro sistema.

```
# ./configure
```

b) Compilación de *OpenSSH*

Paso siguiente es realizar la compilación de las librerías de *OpenSSH* simplemente tecleando *make*.

```
# make
```

c) Instalación de *OpenSSH*

Si no se presentó problema alguno el paso siguiente es como superusuario teclear la sentencia apropiada para la instalación de las librerías.

```
# make install
```

Si todo funcionó de manera adecuada ya tendremos en nuestro sistema instalado el *OpenSSH* funcionando y trabajando de forma apropiada.

Para comprobar la existencia de éste bastará con sólo teclear la sentencia:

```
# ssh -V

SSH Version OpenSSH_2.3.0p1, protocol versions 1.5/2.0.
Compiled with SSL (0x0090600f).
```

Lo cual nos indica que el *OpenSSH* está listo en el sistema aceptando los protocolos 1.5 y 2.0

d) Ya que se realizó la instalación en el equipo procedemos a configurar el sistema para poder ejecutar el demonio del servidor de *ssh* y permitir accesos por el puerto por defecto de Secure Shell (puerto 22).

Editar el archivo */etc/inetd.conf* e incluir la siguiente línea:

(Si no se tiene habilitado TCP-Wrapper)

ssh tcp root nowait /usr/local/sbin/sshd /usr/local/sbin/sshd -i

(Si se esta usando TCP-Wrapper)

ssh tcp root nowait /usr/local/etc/tcpd /usr/local/sbin/sshd -i

Editar el archivo */etc/services* y habilitar el puerto para Secure Shell usando la siguiente línea:

. . .

ssh 22/tcp Secure Shell
ssh 22/udp Secure Shell

. . .

Por último reiniciar el demonio de *inetd* usando la línea:

#/etc/rc.d/init.d/inetd restart

En este punto el sistema debe responder a las peticiones de conexión por Secure Shell.

3.3.9. SINTONIZACIÓN DE LOS NODOS.

Eliminación de servicios no utilizados

En el nodo, dentro de su directorio *etc/rc.d/rc3.d* se deshabiltan los servicios que no son necesarios en este nivel de ejecución del sistema. Esta eliminación se hace de acuerdo a las necesidades que se tengan y dejando sin cambio los servicios básicos del sistema.

En el directorio ya mencionado, aparece una lista de enlaces hacia diferentes programas que se ejecutan. Los servicios que están activados se identifican por una S mayúscula al inicio de su nombre, los servicios que se desactivan se le cambian en su nombre la S por una K al inicio. Así, al iniciar el sistema, sólo los servicios que inician con una S son los que se ejecutan.

La eliminación se lleva a cabo tomando en cuenta aquellos servicios que no se utilizan en el nodo, y sólo ocupan los recursos del mismo, siendo estos más necesarios para el cómputo.

Algunos de los servicios eliminados son:

lpd – servicio de impresión
xfs – servicio para la consola gráfica
apm – administración de energía, etc.

Módulos del sistema.

Debido a que el Kernel de Linux, en la plataforma que se está trabajando (Intel), puede trabajar de manera modular, es posible que al compilar un nuevo Kernel no todos los módulos que se encuentra instalados en el servidor serán usados. Al iniciar el sistema del nodo, lista los módulos que el Kernel no puede resolver; por lo que, para eliminar la presencia de esta lista será necesario eliminarlos o removerlos del directorio */lib/modules/version_Kernel/* del nodo. Con la ayuda del comando *depmod*, que muestra los nombres de los módulos que no se pueden resolver, es posible ir removiendo los módulos no utilizados junto con el comando *rm*.

Duplicación de los sistemas de archivos para los nodos.

Para que se puedan agregar más nodos al cluster, una vez que se han sintonizado los nodos, se procede a realizar la copia del sistema de archivos que se ha creado para el primer nodo. Para ello se ejecuta el *script* que se muestra a continuación.

```
#!/bin/sh
if [ $# != 2 ]
then
        echo Usage: $0 directorio_anterior nuevo_directorio
        exit 1
fi
cd /tftpboot
if [ ! -d $1 ]
then
        echo $1 no es un directorio
        exit 1
fi
umask 022
mkdir -p $2

# solamente crea estos
for d in home mnt proc tmp usr
do
        mkdir $2/$d
done
chmod 1777 $2/tmp
touch $2/fastboot
chattr +i $2/fastboot
```

```
# liga estos otros
for d in bin lib sbin
do
        (cd $1; find $d -print | cpio -pl ../$2)
done

# hace la copia de estos directorios
for d in dev etc root var
do
        cp -a $1/$d $2
done

cat <<EOF
Ahora, en /tftpboot/$2/etc, hay que editar

        sysconfig/network
        sysconfig/network-scripts/ifcfg-eth0
        fstab (tal vez)
        conf.modules (tal vez)

y configurar

        rc.d/rc3.d
EOF
```

El *script* anterior se obtuvo de la documentación de Etherboot.

Se agrega el nombre del nodo creado y su dirección IP en el archivo */etc/hosts y /tftpboot/uxmaln/etc/hosts*, para que sea reconocido por su nombre y su alias cada nodo.

3.3.10. SINTONIZACIÓN DEL SERVIDOR.

El servidor sólo requiere sintonización para cuestiones de seguridad.

De esta forma se cierran los puertos no utilizados y considerados inseguros. Para poder cerrarlos, en el archivo */etc/services* se le antepone un # para poder cancelar los servicios.

3.3.11. PVM (PARALLEL VIRTUAL MACHINE)

Al realizar la instalación del sistema operativo, entre los paquetes que se tienen dentro de las opciones para instalar, está PVM. Para verificar que el paquete ya está instalado, se puede utilizar el comando

```
#rpm -qi pvm
```

lo que devolverá una de dos posibles salidas:

1. La información acerca del paquete, o la lista de los paquetes instalados con los que coincide en su nombre la cadena dada.
2. Un mensaje que indica que el paquete no está instalado.

Cabe mencionar que este comando sólo nos proporcionará esta información si se realizó la instalación por medio de un paquete *rpm*.

Si no está instalado, habrá que conseguir el paquete, ya sea en forma de archivo comprimido, o en forma de *rpm*.

La instalación que en nuestro caso es más conveniente es por *rpm*.

#rpm –i pvm-3.4.3-4.i386.rpm
#rpm –i make_pvm-3.78.1-2.i386.rpm
 # rpm –i pvm-gui-3.4.3-4.i386.rpm

Una vez instalado entonces sólo hay que agregar las siguientes líneas en el archivo */etc/skel/.bashrc*

export PVM_ROOT=/usr/share/pvm3
export PVM_ARCH=LINUX

Al crearlo en */etc/skel/.bashrc* se permite que al agregar más usuarios al sistema, estas variables se copien a su directorio *Home*.

Si se tiene cuentas ya creadas, se agregan estas líneas al *.bashrc* de cada usuario.

Para que funcione el PVM, es necesario que se puedan ejecutar comandos de manera remota por medio del comando *rsh*. Para esto se agrega en el directorio */etc* el archivo *hosts.equiv*, que contiene los nombres y direcciones de las máquinas en las cuales puede ejecutarse comandos de manera remota. De esta manera, cada vez que se agrega un nuevo nodo habrá que agregar su nombre y dirección *IP* a este archivo.

Capítulo IV

Computación Paralela

4. COMPUTACIÓN PARALELA

Uno de los objetivos de este trabajo es hacer ver que 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 en nuestro caso un cluster Beowulf, 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.

Esta sección proporciona información general sobre los conceptos necesarios para poder entender cómo llevar a cabo la paralelización. No es una descripción exhaustiva o completa. Es una breve descripción de los temas que pueden ser importantes para un diseñador y programador que desee paralelizar una aplicación.

4.1. PARALELISMO Y CONCURRENCIA

Para conseguir un buen nivel de paralelismo es necesario que el hardware y el software se diseñen conjuntamente.

Existen dos visiones del paralelismo :

- **Paralelismo hardware** : Es el paralelismo definido por la arquitectura de la máquina.
- **Paralelismo software** : Es el paralelismo definido por la estructura del programa. Se manifiesta en las instrucciones que no tienen interdependencias.

El paralelismo se presenta, a su vez, en dos formas :

- **Paralelismo de control**: Se pueden realizar dos o más operaciones simultáneamente. Se presenta en los *pipelines* y las múltiples unidades

funcionales. El programa no necesita preocuparse de este paralelismo, pues se realiza a nivel hardware.

- **Paralelismo de datos**: Una misma operación se puede realizar sobre varios elementos simultáneamente.

En relación al paralelismo hardware, Michael Flynn realizó una clasificación de arquitecturas de computadoras (se mencionan en el capítulo 1).

4.1.1. CONCEPTO DE CONCURRENCIA

La concurrencia está presente en casi todas las actividades que realiza el ser humano. Normalmente, los primeros principios de programación se introducen estableciendo una analogía entre un programa y la consecución de tareas que se realizan en una actividad diaria cualquiera, como por ejemplo preparar una comida:

```
Abrir refrigerador
Si refrigerador está vacío entonces
  comer en el restaurante o comprar en la tienda
Si no
      preparar sopa
      preparar guisado
      preparar postre
      comer en casa
fin si
```

Esta analogía introduce el concepto de programa secuencial como la descripción de una secuencia de acciones. La ejecución de dicho programa la realiza un procesador y el patrón de funcionamiento resultante se conoce como procesador secuencial o simplemente proceso.

Un gran número de programas se puede expresar aceptablemente de una forma secuencial. Sin embargo, hay ciertas clases de problemas donde es esencial, o simplemente más apropiado, desarrollar un programa como un conjunto de *procesos cooperativos* que pueden ejecutarse en paralelo, o concurrentemente, para resolver dicho problema. En particular, es una buena idea el desarrollar un programa de esta forma, cuando la concurrencia de actividades es un aspecto esencial del problema a resolver.

Si consideramos en el ejemplo anterior que (ver Tabla 4.1):

Juan	*Pablo*
preparar sopa (1)	***preparar guisado (2)***
preparar postres (3)	***preparar la mesa (4)***
comer(5)	***comer (5)***

Tabla 4. 1.

Y si representamos mediante un grafo el orden en que se realizan las tareas anteriores, lo que se conoce como *diagrama o grafo de dependencias,* obtendremos (figura 4.1):

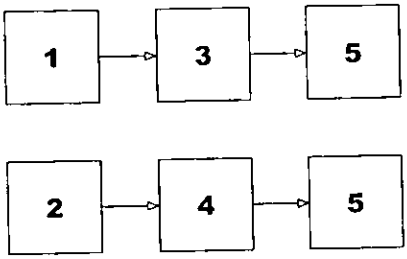


Figura 4. 1 Diagrama o grafo de dependencias.

Obviamente, el tiempo consumido es menor que en el caso secuencial, concretamente y, según la representación anterior, sería la mitad del caso secuencial. Sin embargo, para obtener esa reducción temporal, que es la máxima, ya que se está suponiendo que todas las tareas comienzan y terminan a la vez (igual duración), se ha duplicado el consumo de recursos. Por tanto, se debe establecer un compromiso entre el espacio y el tiempo para determinar si es o no adecuado el planteamiento concurrente frente al secuencial. Lo anterior puede aplicarse a algún programa secuencial determinando las partes concurrentes.

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.

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*. 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.

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. Así uno de los aspectos más importantes del diseño de programas concurrentes es expresar éstos de forma que se garantice su funcionamiento correcto independientemente del orden en el que se pueden realizar algunas de sus acciones individuales.

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 (espera 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

Los programas concurrentes a diferencia de los programas secuenciales tienen una serie de problemas muy particulares derivados de las características de la concurrencia, a continuación se da una breve descripción de los tipos de problemas:

1. **Violación de la exclusión mutua** : En ocasiones ciertas acciones que se realizan en un programa concurrente no proporcionan los resultados deseados. Esto se debe a que existe una parte del programa donde se realizan dichas acciones que constituye una región crítica, es decir, es una parte del programa en la que se debe garantizar que si un proceso accede a la misma, ningún otro podrá acceder. Se necesita pues garantizar la exclusión mutua.

2. **Bloqueo mutuo o *Deadlock*** : Un proceso se encuentra en estado de *deadlock* si está esperando por un suceso que no ocurrirá nunca. 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.

3. **Retraso indefinido o *starvation*** : Un proceso se encuentra en *starvation* si es retrasado indefinidamente esperando un suceso que puede no ocurrir nunca. 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.

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** : En ocasiones cuando un proceso se encuentra a la espera por un suceso, una forma de comprobar si el suceso se ha producido es verificando continuamente si el mismo se ha realizado ya. Esta solución de espera ocupada es muy poco efectiva, porque desperdicia tiempo de procesamiento, y se debe evitar. La solución ideal es suspender el proceso y continuar cuando se haya cumplido la condición de espera.

4.1.2. PROGRAMAS CONCURRENTES

En este trabajo utilizaremos el concepto de que un programa concurrente se puede definir como un programa que contiene partes que están diseñadas para ejecutarse en paralelo.

En general, hay dos razones para que un programador desarrolle un programa concurrente para resolver un problema dado:

1) Porque el problema a resolver sugiere de forma natural una solución concurrente, tal y como ocurre si el programa debe realizar operaciones que deben desarrollarse en paralelo, o si se deben gestionar eventos que se pueden producir en cualquier instante.

2) Porque el hardware de la computadora sobre la que se va a ejecutar el programa soporte ejecución de operaciones en paralelo y el tiempo de ejecución del programa pueda reducirse si éste se expresa de forma concurrente.

Como se ha comentado, un programa concurrente contiene un conjunto de procesos que pueden ejecutarse en paralelo. Si en la máquina sobre la que se va a ejecutar el programa están disponibles tantos procesadores como

procesos entonces la concurrencia puede realizarse completamente, en caso contrario, los procesos deben ejecutarse en tiempo compartido sobre el procesador o procesadores disponibles. En la siguiente figura se representa el tiempo de ejecución de los dos procesos que componen un programa concurrente en ambos casos.

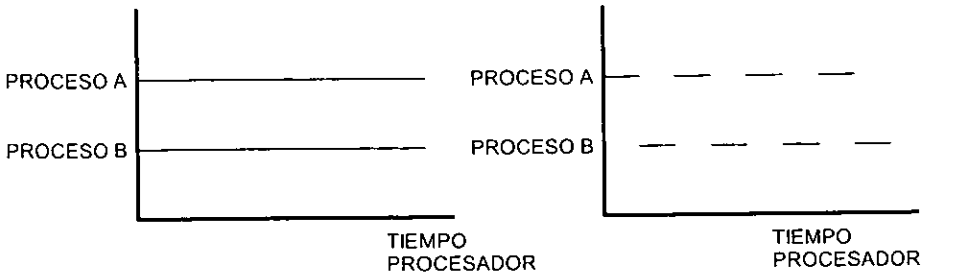


Figura 4. 2 Representación del tiempo de ejecución de dos procesos en uno y dos procesadores.

En nuestro caso el objetivo es diseñar programas que se ejecuten en menos tiempo y que cada procesador mantenga un proceso diferente.

La tarea del programador es determinar qué partes CONCURRENTES del programa DEBEN ser ejecutadas en PARALELO y cuáles NO DEBEN serlo. La respuesta a esto determinará la EFICIENCIA de la aplicación.

4.2. GRANULARIDAD DEL CÓDIGO Y NIVELES DE PARALELISMO

El paralelismo aparece en varios niveles de Hardware y Software: Señales, circuitos, componentes y niveles del sistema. En los niveles mas bajos, las señales viajan en paralelo a través de rutas de los datos paralelos. En los niveles más altos, múltiples unidades funcionales operan en paralelo para un desempeño más rápido, conocido como (*instruction level paralelism*).

Los niveles de señal y circuito son desarrollados implícitamente por técnicas de hardware llamado Paralelismo de Hardware .

Los niveles de componentes y del sistema de paralelismo son expresados implícita/explícitamente por varias técnicas de software, conocida como paralelismo de software.

Los niveles de paralelismo pueden estar basados en el tamaño del código (tamaño del grano), que puede ser un candidato potencial para el paralelismo. como se muestra en la siguiente tabla.

Tamaño del Grano	**Código Detallado**	**Paralelismo por:**
Muy fino	Instrucción	Procesador
Fino	Ciclos/ bloque de instrucciones	Compilador
Medio	Una página de funciones Estándar (Standard One Page Function)	Programador
Grande	Procesos pesados en programas separados	Programador

Tabla 4. 2 Niveles de paralelismo.

Todos los acercamientos de crear paralelismo basado en la granularidad del código, tiene como objetivo aumentar la eficiencia del procesador, reduciendo la latencia de duración de operación, como el acceso a memoria/disco.

El paralelismo en una aplicación puede ser detectado en varios niveles:

- Grano muy fino (múltiples instrucciones sencillas)
- Grano fino (niveles de datos)
- Grano medio (nivel de control)
- Grano grande (nivel de tareas)

Los diferentes niveles de paralelismo se muestran en la siguiente figura:

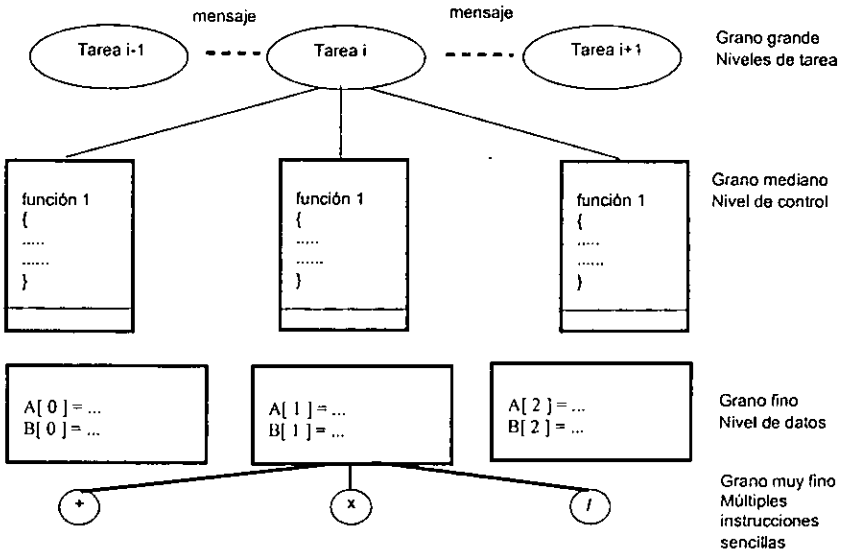


Figura 4. 3 Niveles de paralelismo.

4.3. DISEÑO DE ALGORITMOS PARALELOS

Una de las herramientas para el diseño de algoritmos paralelos es la creatividad del ser humano, que en muchos casos alcanza resultados verdaderamente asombrosos por su calidad. Otras formas, menos creativas, de alcanzar un resultado satisfactorio son utilizar teorías y modelos contrastados.

En este trabajo para la realización de los programas paralelos que se presentarán más adelante no nos basaremos en ninguna metodología de diseño establecida, sólo en el paradigma de programación (de los que se hablará en la siguiente sección) que se podría seguir. Se analizará el problema y se buscará la mejor solución, por lo que en la sección donde se describan los programas implementados, se dará una breve descripción del problema y de cómo se estructuró el programa que logrará la paralelización del mismo. Sin embargo cabe mencionar a grandes rasgos en qué consiste la metodología propuesta por Ian Foster[1].

Él sugiere una metodología que organice el diseño de procesos en 4 etapas. Efectúa un estudio del Particionado, de los requerimientos de las Comunicaciones, de las posibilidades de agrupamiento de tareas y del Mapeo de las tareas en los procesadores (Acrónimo PCAM).

> **Partición y Comunicación**: Se busca obtener algoritmos paralelos enfocando el problema desde el punto de vista de la Concurrencia y la Escalabilidad.
>
> **Agrupación y Mapeo**: Estudian aspectos relacionados con la Localidad y el Rendimiento.

PARTICIÓN

Descomposición de actividades computacionales y de los datos sobre los que se opera en pequeñas tareas.

En la etapa de partición se buscan oportunidades de paralelismo y se trata de subdividir el problema lo más finamente posible, es decir; que la granuralidad sea fina. Evaluaciones futuras podrán llevar a aglomerar tareas y descartar ciertas posibilidades de paralelismo.

Una buena partición divide tanto el cómputo como los datos. Hay dos formas de proceder con la descomposición.

---

[1] High Performance Cluster Computing: Programming and Aplication, Vol. 2
Rajkumar Buyya
Monash University, Australia.
P.p. 15.

Descomposición del dominio: El centro de atención son los datos. Se determina la partición apropiada de los datos y luego se trabaja en el cómputo asociado con los datos.

Descomposición funcional: es el enfoque alternativo al anterior. Primero se descompone el cómputo y luego se ocupa de los datos.

COMUNICACIÓN

Está enfocada al flujo de información y coordinación de la cantidad de tareas que son creadas durante la etapa de partición. La naturaleza del problema y el método de descomposición determina los patrones de comunicación entre las tareas corporativas de un programa paralelo.

Las tareas definidas en la etapa anterior, en general, pueden correr concurrentemente pero no independientemente. Los datos deben ser transferidos o compartidos entre tareas y esto es lo que se denomina la fase de comunicación.

AGRUPAMIENTO (AGGLOMERATION)

Los algoritmos obtenidos al aplicar las primeras etapas del proceso de diseño, particionado y análisis de las comunicaciones, se pueden considerar como algoritmos abstractos, en el sentido de que en su construcción no se han tenido en cuenta la eficiencia ni el tipo de computadora paralela donde se van a ejecutar. De hecho, pueden ser muy ineficientes si, por ejemplo, se crean más tareas que procesadores.

ASIGNACIÓN

En esta última etapa se determina en qué procesador se ejecutará cada tarea. Este problema no se presenta en máquinas de memoria compartida tipo UMA. Éstas proveen asignación dinámica de procesos y los procesos que necesitan de una CPU están en una cola de procesos listos. Cada procesador tiene acceso a esta cola y puede correr el próximo proceso. Por ahora no hay mecanismos generales de asignación de tareas para máquinas distribuidas. Esto continúa siendo un problema difícil y que debe ser atacado explícitamente a la hora de diseñar algoritmos paralelos.

La asignación de tareas puede ser estática o dinámica. En la asignación estática, las tareas son asignadas a un procesador al comienzo de la ejecución del algoritmo paralelo y corren ahí hasta el final. En la asignación dinámica se hacen cambios en la distribución de las tareas entre los procesadores a tiempo de ejecución, o sea, hay migración de tareas en tiempo de ejecución.

4.4. PARADIGMAS DE PROGRAMACIÓN PARALELA

Las aplicaciones paralelas pueden ser clasificadas dentro de los paradigmas de programación. Algunos paradigmas de programación son usados repetidamente para desarrollar programas paralelos. Cada paradigma es una clase de algoritmo que tiene una estructura de control.

La selección de paradigmas es determinada por los recursos de cómputo paralelo disponibles y por el tipo de paralelismo inherente en el problema. Los recursos de cómputo pueden definir el nivel de granularidad que puede ser soportado eficientemente en el sistema. El tipo de paralelismo refleja la estructura de la aplicación o de los datos y ambos tipos pueden existir en diferentes partes de la misma aplicación. El paralelismo que surge desde la estructura de la aplicación es llamado *paralelismo funcional*; en este caso diferentes partes del programa pueden desarrollar diferentes tareas de una manera concurrente y cooperativa. El paralelismo también se puede encontrar en la *estructura de los datos*; en este tipo de paralelismo se permite la ejecución de procesos paralelos con operaciones idénticas pero con diferentes partes de los datos.

4.4.1. ESCOGER UN PARADIGMA DE PROGRAMACIÓN

En el mundo de la computación paralela hay varios autores que presentan varias clasificaciones de paradigmas de programación, aquí solo presentaremos los más populares y que comúnmente son usados en programación paralela.

1) Maestro/ Esclavo (o Task-Farming)
2) Un solo programa–Múltiples datos SPMD(Single–Program Múltiple-Data)
3) Entubamiento de datos (Data Pipeling)
4) Divide y conquista
5) Paralelismo especulativo

Maestro / Esclavo (o Task-Farming): Este paradigma consta de dos entidades: un maestro y múltiples esclavos . El maestro es responsable de descomponer el problema en pequeñas tareas (y distribuir esas tareas entre una granja de procesos esclavos) , así también como para recoger los resultados parciales en orden para producir un resultado final del cómputo realizado. Los procesos esclavos ejecutan un ciclo muy simple: obtener un mensaje con la tarea, procesar la tarea y mandar el resultado al maestro. Usualmente la comunicación toma lugar sólo entre el maestro y los esclavos.

Este paradigma puede usar balanceo de carga estática, o balanceo de carga dinámica. En el primer caso, la distribución de tareas es toda desarrollada al inicio del cómputo, lo cual permite al maestro participar en el cómputo después de que a cada esclavo se le ha proporcionado una fracción del trabajo. La

distribución de las tareas puede ser hecha una sola vez o de una manera cíclica. En la figura 4.3 se representa el balanceo por carga estática.

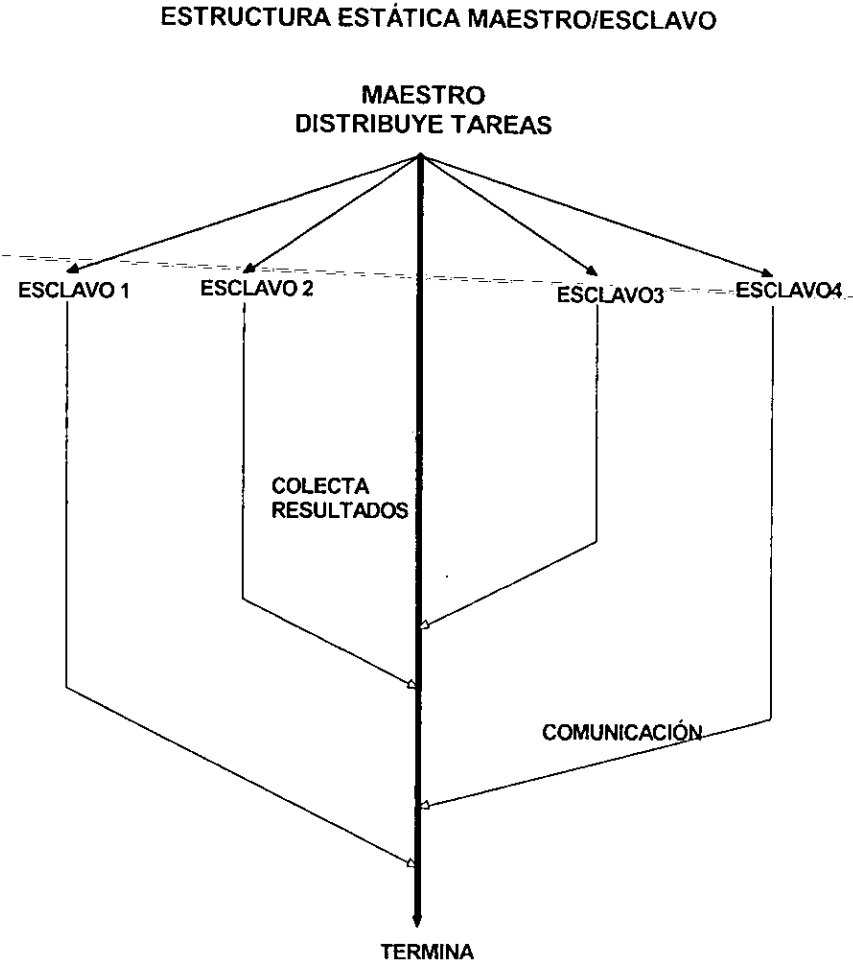


Figura 4. 4 Paradigma de programación.

La otra manera es usar un balanceo de carga dinámico, el cuál puede ser más conveniente cuando :

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

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

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

Hasta un cierto punto, este paradigma puede incluir algunas aplicaciones que están basadas en acercamientos triviales por descomposición: el algoritmo secuencial es ejecutado simultáneamente en diferentes procesadores, pero con diferentes entradas de datos. En algunas aplicaciones no hay dependencias entre diferentes corridas, así no hay necesidad de comunicación o coordinación entre procesos.

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

UN SOLO PROGRAMA – MULTIPLES DATOS (*Simple—Program Múltiple – Data)*: Este paradigma es el más comúnmente usado. Cada proceso ejecuta básicamente la misma pieza de código , pero en diferente parte de los datos. Esto involucra la división de los datos de la aplicación entre los procesadores disponibles. Este tipo de paralelismo es además referido como *paralelismo geométrico*, *descomposición de dominios*, o *paralelismo de datos*. En el siguiente esquema se representa este paradigma (ver figura 4.4).

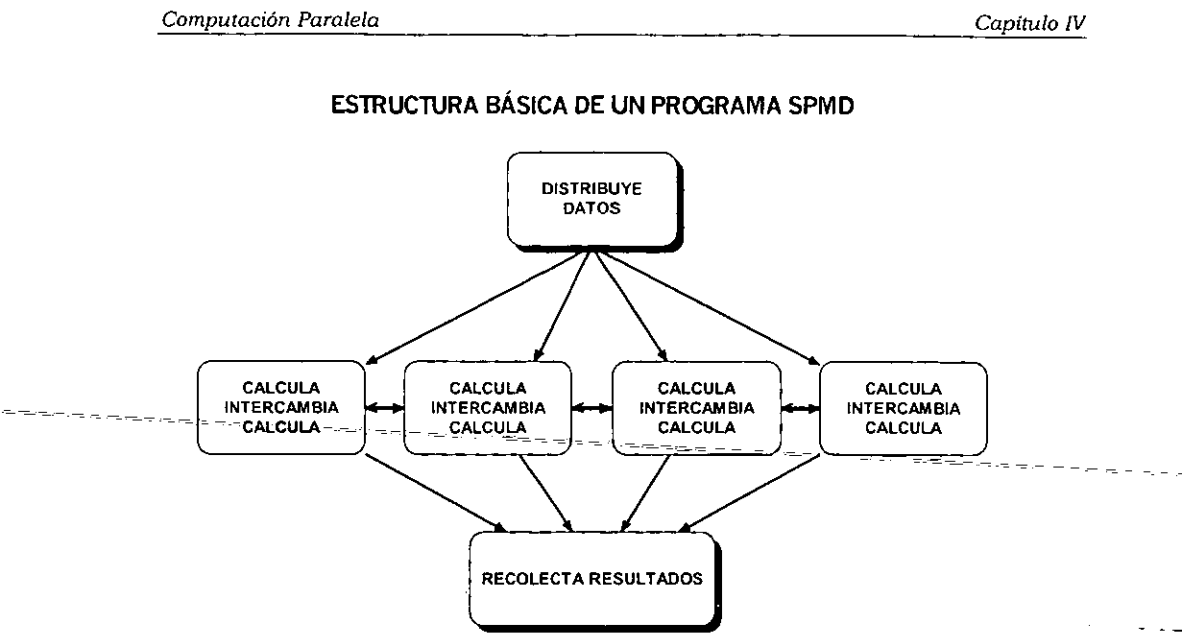


Figura 4. 5 Paradigma SPMD.

Muchos problemas físicos tienen como base la estructura de figuras geométricas regulares, con interacciones espacialmente limitadas. Esta homogeneidad permite a los datos ser distribuidos uniformemente entre los procesadores, donde cada uno será responsable de un área espacialmente definida. Los procesadores se comunican con sus vecinos y la carga de comunicación será proporcional al tamaño de los elementos que tiene definidos en su espacio, mientras la carga del cómputo será proporcional al volumen del elemento. Esto, además, podrá ser requerido para desempeñar una sincronización global entre todos los procesos. El patrón de comunicación es usualmente altamente estructurado y altamente predecible. Los datos podrán inicialmente ser generados ellos mismos por cada proceso o podrán ser leídos desde el disco durante la fase de inicialización.

Las aplicaciones ***SPMD*** pueden ser muy eficientes si los datos están bien distribuidos por los procesos, y el sistema es homogéneo. Si los procesos presentan diferentes cargas de trabajo o capacidades, entonces el paradigma requiere el soporte de algún esquema de balanceo de cargas que pueda adaptar el esquema de distribución de datos durante el tiempo de ejecución.

Este paradigma es altamente sensible a perder algunos procesos. Usualmente la pérdida de un solo proceso es la causa de un bloqueo en el cálculo, en el cual ninguno de los procesos puede avanzar más allá del punto de sincronización global.

Entubamiento de datos (*Data Pipeling*): Este es un paralelismo de grano más fino, el cual está basado en un acercamiento de descomposición de funciones: Las tareas del algoritmo, el cual es capaz de realizar una operación concurrente son identificadas y cada procesador ejecuta una parte del algoritmo total. El entubamiento es uno de los más simples y más populares paradigmas de descomposición de funciones. La figura 4.5 se presenta la estructura de este modelo.

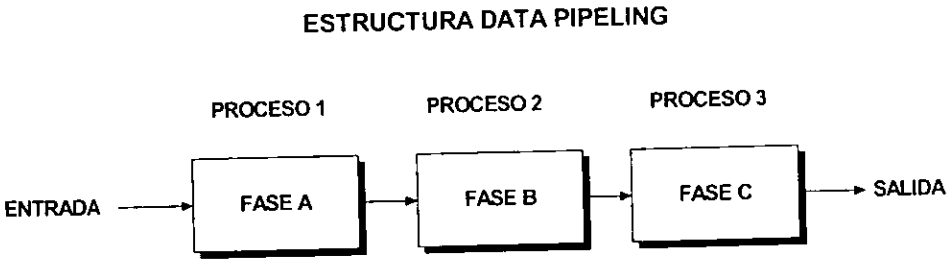


Figura 4. 6 Paradigma Data Pipeling.

Los procesos son organizados en forma de tubería; cada proceso corresponde a una parte de la tubería y es responsable de una tarea en particular. El patrón de comunicaciones es muy simple debido a que los flujos de datos se dan entre las partes de la tubería. Por esta razón, este tipo de paralelismo es llamado algunas veces como paralelismo de flujo de datos. La comunicación es totalmente asíncrona. La eficiencia de este paradigma depende directamente de la habilidad de balancear las cargas entre las fases de la tubería. La robustez de este paradigma en contra de reconfiguraciones del sistema puede ser lograda al proveer múltiples caminos independientes a través de las fases. Este paradigma es a menudo usado en la reducción de datos o en las aplicaciones de procesamiento de imágenes.

DIVIDE Y CONQUISTA: El acercamiento divide y conquista es bien conocido en el desarrollo de algoritmos secuenciales. Un problema es dividido en dos o más subproblemas. Cada uno de estos subproblemas es resuelto independientemente y sus resultados son combinados para obtener el resultado final. Algunas veces, los problemas más pequeños son pequeñas réplicas del original, lo que conduce a una solución recursiva. El procesamiento es requerido para dividir el procesamiento original o para combinar el resultado de los subproblemas. En la versión paralela del divide y conquista, los subproblemas pueden ser resueltos al mismo tiempo, dando suficiente paralelismo. Los procesos de reparticionamiento y recombinación también usan algo de paralelismo pero estas operaciones requieren algo de comunicación entre procesos. Sin embargo, debido a que los subproblemas son independientes, la comunicación no es necesaria entre los procesos que trabajan en diferentes subproblemas.

Se pueden identificar tres operaciones genéricas en el cómputo de divide y conquista: Separar, realizar el cómputo, y unir. La aplicación está organizada en forma de un árbol virtual: algunos de los procesos crean subtareas y tienen que combinar los resultados que producen las subtareas para dar un resultado común. Las tareas son realizadas por los procesos que se encuentran en las hojas del árbol virtual. La siguiente figura nos muestra esta ejecución:

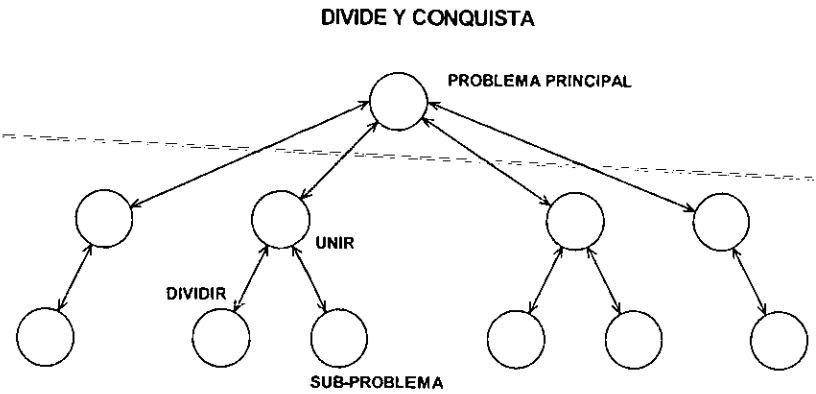


Figura 4. 7 Paradigma Divide y Conquista.

El paradigma de maestro/esclavo puede ser ligeramente modificado, degenerando en una especie de divide y conquista; por ejemplo, cuando el problema de la descomposición es realizado antes de que las tareas sean enviadas, las operaciones de separar y unir solamente son realizadas por el proceso maestro y todos los otros procesos son los responsables del cómputo.

En el modelo divide y conquista, las tareas pueden ser generadas durante el tiempo de ejecución y pueden ser agregadas a una cola de trabajos en el procesador maestro o distribuidas a través de algunas colas de trabajo en todo el sistema.

Los paradigmas de programación pueden ser caracterizados principalmente por dos factores: descomposición y distribución del paralelismo. Por ejemplo, en el paralelismo geométrico la descomposición y la distribución son estáticas. Lo mismo pasa con la descomposición y distribución de funciones del entubamiento de datos. En la granja de tareas el trabajo es estáticamente descompuesto pero dinámicamente distribuido. Finalmente, en el paradigma de divide y conquista la descomposición y la distribución son dinámicas.

PARALELISMO ESPECULATIVO: Este paradigma es empleado cuando es un poco difícil de obtener paralelismo a través de alguno de los paradigmas previos .Algunos problemas tienen dependencias de datos complejas, lo cual reduce las posibilidades de explotar la ejecución en paralelo. En estos casos,

una solución apropiada es ejecutar el problema en partes pequeñas pero usar algo de especulación o ejecución optimista para facilitar el paralelismo.

En algunos problemas asíncronos el sistema prueba una ejecución predictiva en relación a ciertas actividades y asumiendo que estas ejecuciones concurrentes no violan la consistencia de la ejecución del problema. Si se llega a dar, es necesario regresar al estado previo más consistente de la aplicación.

Otro uso de este paradigma, es usar diferentes algoritmos para el mismo problema. El primero que dé una solución es el que se escoge.

MODELOS HÍBRIDOS: Los límites entre los paradigmas son muchas veces difusos y, en algunas aplicaciones, se pueden necesitar mezclar elementos de diferentes paradigmas. Los métodos híbridos que incluyen más de un paradigma básico son usualmente observados en algunas aplicaciones paralelas de gran escala. Estas situaciones hacen que se mezcle el paralelismo de datos y tareas simultáneamente o en diferentes partes del mismo programa.

4.5. APLICACIONES PARALELAS

Las aplicaciones que pueden desarrollarse en un cluster son principalmente: Exigentes aplicaciones secuenciales y grandes desafíos en cómputo (GCAs Grand Challenge Applications) que son fundamentalmente problemas de ciencia e ingeniería con gran impacto económico y científico.

Un ejemplo típico de un gran desafío, es la simulación de algunos fenómenos que no pueden ser medidos a través de experimentos. Entre los GCAs podemos encontrar problemas de cristalografía y microtomografía estructural, análisis del tiempo entre otros en los que se emplee grandes cálculos (super cómputo de datos).

En el trabajo no se tratara una aplicación en específico, ya que para ello es necesario conocer sobre los temas que involucren el problema. Pero se analizarán pequeños algoritmos que pueden ayudar a la implementación de algoritmos más grandes para una cierto programa de aplicación.

4.6. HERRAMIENTAS DE PARALELIZACIÓN.

PVM

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 le 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

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:

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

PVM vs MPI

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

PVM se desarrolló basándose en el concepto de *máquina virtual*, esto es, una colección de recursos computacionales manejados como un computador paralelo.

Por su parte, MPI se centró en el *paso de mensajes*, y aunque no tiene el concepto de máquina virtual, sí hace una abstracción de todos los recursos en términos de topología de paso de mensajes.

El desarrollo de la biblioteca de programación paralela MPI (*Message Passing Interface*) ha causado que muchos programadores se planteen el desarrollar programas usando MPI o bien PVM (*Parallel Virtual Machine*).

MPICH

Es una implementación de la librería de paso de mensajes MPI (Message Passing Interface) escrita por el Argonne National Laboratory y la Mississipi State University. MPICH está especialmente optimizado para entornos Myrinet.

La principal característica de la librería MPICH es su adaptabilidad a gran número de plataformas, incluyendo clusters de estaciones de trabajo y procesadores masivamente paralelos (MPP).

Las características más importantes de esta librería, son:

- Soporta paralelismo del tipo SPMD (Single Program Multiple Data) y MPMD (Multiple Program Multiple Data).
- La transferencia de mensajes se realiza de forma cooperativa, tanto el proceso emisor como en el receptor participan en el traspaso.
- Permite establecer comunicación punto-a-punto (participan exactamente dos procesos) o comunicación colectiva (participan un número elevado de procesos simultáneamente).
- Proporciona cuatro modos de comunicación: *standard*, *synchronous*, *buffered*, *ready*.
    - *Standard (Normal)*: la emisión de un mensaje se completa una vez que ha sido enviado, haya o no llegado al receptor.
    - *Synchronous (Sincrono)*: el proceso de transferencia no se completa hasta que el emisor recibe la confirmación que el receptor ha recibido el mensaje.
    - *Buffered (En espera)*: el mensaje es copiado en un *buffer* del sistema para transmitirlo más tarde si fuese necesario. En este modo la transferencia se completa inmediatamente.

- *Ready (Listo)*: el emisor deja el mensaje dentro de la red de comunicación y supone que el receptor lo estará esperando. Este modo también se completa inmediatamente.

MPICH permite trabajar con lenguaje C, C++ y FORTRAN.

HPF

El lenguaje de programación HPF fue diseñado como un conjunto de extensiones y modificaciones al lenguaje Fortran estándar establecido en la *International Standard for Fortran*. Los objetivos fundamentales que están detrás del desarrollo del lenguaje HPF incluyen:

- Soporte para programación paralela de datos (threads simples, espacio de nombres globales, computación paralela no necesariamente síncrona).
- Portabilidad entre diferentes arquitecturas.
- Alto rendimiento en computadores paralelos con memoria no-uniforme (mientras no impida el desempeño en las demás máquinas).
- Utilización de Fortran estándar como lenguaje base (para HPF 2.0 el estándar es Fortran 95).
- Interfaces abiertas e interoperabilidad con otros lenguajes (por ejemplo, C) y otros bibliotecas de programación (como la librería de comunicación MPI).

Un objetivo importante de HPF es la de alcanzar la portabilidad de código entre una variedad de máquinas paralelas. Esto requiere que los programas de HPF compilen no sólo en todas las máquinas que se van a utilizar, sino también que un programa HPF que sea altamente eficiente en una máquina dada tenga un rendimiento comparable en otra máquina paralela con un número similar de procesadores. De otro modo, el esfuerzo que realiza el programador para optimizar un programa en una determinada arquitectura sería desperdiciado cuando ejecute su programa en otra arquitectura.

Entre los factores que afectan el desempeño de un programa paralelo están el grado de paralelismo disponible, manejo de la distribución de los datos y una selección y distribución adecuada de las tareas que deben ejecutarse. El lenguaje HPF provee mecanismos para que el programador le indique al compilador cómo manejar los factores anteriores.

Basado en el lenguaje Fortran, las características del lenguaje HPF se pueden dividir en:

- Directivas HPF
- Sintaxis de lenguaje nueva
- Nuevas rutinas de librería
- Cambios en el lenguaje y restricciones

PARMACS

PARMACS es una librería de paso de mensajes que permite configurar una red heterogénea de *workstations* como una máquina paralela. Altamente portable, puede ser utilizada desde FORTRAN y C. No obstante, ha quedado superada por MPI.

Las principales características de PARMACS son:

- Modelo de aplicación SPMD
- Paso de mensajes síncronos y asíncronos.
- Soporte para uno o más mensajes.
- Soporte para una topología de procesos lógicos.
- Operaciones Globales.
- Librerías de interfase.
- Procesos optimizados.
- Soporte para el *broadcast* de mensajes.
- Soporte para C y Fortran.

4.7. PVM (PARALLEL VIRTUAL MACHINE).

En el desarrollo de nuestro trabajo, para realizar la programación concurrente utilizamos la máquina virtual que proporciona PVM. La utilizamos debido a la facilidad que proporciona de poder ver al cluster como una sola máquina multiprocesador. El usar PVM y no otra herramienta de paso de mensajes (como es MPI) esta en relación con las facilidades que da PVM para entender el paso de mensajes, ya que con su interfaz gráfica es posible entender como se generan los procesos, su estado, y seguir la trayectoria que los mensajes siguen. Además que el ambiente de PVM incluye las librerías con las que se pueden realizar los programas, mientras que MPI es solo un estandar de programación, y el cual no permite aumentar y disminuir procesadores en tiempo de ejecución, esto es, no permite un balanceo de cargas dinámico.

A continuación procederemos a dar una explicación de las diferentes características de PVM.

¿Qué es PVM?

PVM es un conjunto integrado de herramientas y librerías 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 máquina 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.

Ventajas del uso de PVM.

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 la 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.

- *Disponibilidad de 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 que 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. En nuestro trabajo utilizaremos Linux, pero también existe para plataformas como son PowerPC con AIX o Sun con Solaris.

El uso de PVM tiene muchas ventajas, pero también tiene una gran desventaja: si el paralelismo que vamos a implementar 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.

Arquitectura de PVM

PVM se compone de dos partes:

La primera parte es el demonio (programa residente en memoria), que se va a llamar *pvmd*. En esta versión de PVM -la 3.4.3-, el nombre es *pvmd3*. El demonio debe estar funcionando en todas las máquinas que vayan a compartir sus recursos computacionales con la máquina paralela virtual. A diferencia de otros demonios y programas del sistema, el demonio de PVM puede ser instalado por el usuario en su directorio particular -de hecho, la instalación por defecto es así-. Esto va a permitir hacer Supercómputo como usuarios, aunque el instalarlo como *root* permite que los usuarios puedan tener acceso a la máquina virtual una vez que son agregados al sistema. Una vez que un usuario -o el administrador- instaló en un directorio PVM, todos los usuarios pueden hacer uso de esa instalación con el requisito -evidente- de que el directorio donde esté instalado PVM sea de lectura al usuario que quiera hacer uso de ella.
Este demonio *pvmd3* es el responsable de la máquina virtual, es decir, de que se ejecuten los programas para PVM y de administrar los mecanismos de comunicación entre máquinas, la conversión automática de datos y de ocultar la red al programador. Por ello, una vez que PVM esté en marcha, el paralelismo es independiente de la arquitectura de la máquina, y sólo depende de la arquitectura de la máquina virtual creada por PVM.

Cada usuario, arrancará el demonio como si se tratase de un programa normal, para ejecutar el código de PVM. Este programa se queda residente, realizando las funciones anteriores.

La segunda parte es la biblioteca de desarrollo. Contiene las rutinas para operar con los procesos, transmitir mensajes entre procesadores y alterar las propiedades de la máquina virtual. Toda aplicación se enlaza a la biblioteca para poderse ejecutar después. Tendremos tres archivos de bibliotecas, la *libpvm3.a* -biblioteca básica en C-, la *libgpvm3.a* -biblioteca de tratamiento de grupos- y la *libfpvm3.a* -biblioteca para Fortran-.

Filosofía de trabajo de un programa para PVM.

Un programa para PVM va a ser un conjunto de tareas que cooperan entre sí. Las tareas van a intercambiar información empleando paso de mensajes. PVM, de forma transparente al programador, va a ocultar las transformaciones de tipos asociadas al paso de mensajes entre máquinas heterogéneas. Toda tarea de PVM puede incluir o eliminar máquinas, arrancar o parar otras tareas, mandar datos a otras tareas o sincronizarse con ellas.

Cada tarea en PVM tiene un número que la identifica unívocamente, es su *TID -Task Identification Number-*. Es el número al que se mandan los mensajes habitualmente. Sin embargo, no es el único método de hacer referencia a una tarea en PVM. Muchas aplicaciones paralelas necesitan hacer el mismo conjunto de acciones sobre un conjunto de tareas. Por ello, PVM incluye una abstracción nueva, el grupo. Un grupo es un conjunto de tareas a las que nos podemos referir con el mismo código, el identificador de grupo. Para que una tarea entre o salga de un grupo, basta con que avise que entró o salió del grupo. Esto nos va a dotar de un mecanismo muy cómodo y potente para realizar programas empleando modelos *SIMD -Single Instruction, Multiple Data-*, en la que se pueden dividir los datos en muchos datos pequeños que sean fáciles de tratar, y después codificarlos en una operación simple y duplicarlos tantas veces como datos unitarios se tengan. Para trabajar con grupos, además de enlazar la biblioteca de PVM *-libpvm3.a-* se tiene que enlazar también la de grupos *-libgpvm3.a-*.

Habitualmente para arrancar un programa para PVM, se lanza manualmente desde una computadora contenida en el conjunto de máquinas, una tarea padre. La tarea se inicia con el comando *spawn* desde un monitor de la máquina virtual, que arrancaremos a su vez con el comando *pvm*. Esta tarea se encargará de iniciar todas las demás tareas, bien desde su función *main* -que va a ser la primera en ejecutarse-, o bien desde alguna subrutina invocada por ella. Para crear nuevas tareas se emplea la función *pvm_spawn*, que devolverá un código de error, asociado a si pudo o no crearla, y el TID de la nueva tarea. Esta función, así como otras relacionadas, se describen en una sección siguiente.

Activación y desactivación de PVM.

La forma más habitual de activar PVM es cargando el programa *pvm* y saliendo de él con el comando *quit*, lo que dejará activada PVM. Si queremos apagar PVM basta con cargar otra vez el programa *pvm* y emplear el comando *halt*. Obsérvese que, para que una computadora sea empleada como parte de PVM, tiene que estar inscrita en el conjunto de máquinas sobre las cuales se puede ejecutar PVM.

Hay que tener cuidado al detener por completo la máquina virtual. Antes de proceder a esto, habrá que eliminar de la máquina virtual a cada uno de los nodos. La razón de realizar esta operación es que al incluir una máquina en PVM, se crean archivos temporales. Si estos archivos no son removidos al detener PVM, al tratar de volver a incluir a dicho nodo se marcaran errores. Generalmente bastará con ir al directorio */tmp* de cada nodo y eliminar los archivos temporales de PVM.

Con qué lenguajes interacciona PVM.

PVM puede ser empleada de forma nativa como funciones en C y en C++, y como procedimientos en Fortran. Basta para ello con tomar las cabeceras necesarias -si trabajamos con C o C++ -; y, para los tres, enlazar con la biblioteca adecuada, que viene con la distribución estándar. En el caso de C es *libpvm3.a* y en el del Fortran *libfpvm3.a*.

Si deseamos trabajar en otros lenguajes puede ser un poco más complejo. Si el lenguaje permite incorporar funciones nativas en lenguaje C -como es el caso, por ejemplo, de Java- no hay ningún problema; ya que podemos invocar la función; bien directamente si el lenguaje lo permite, o haciendo alguna pequeña rutina para adaptar el tipo de los datos, el formato de llamada a función o cualquiera de las restricciones que nos imponga el lenguaje que empleemos para invocar funciones en C.

Toda función en C de la forma pvm_algo tiene como equivalente en Fortran pvmfalgo, y viceversa. El lenguaje de programación a seguir en este trabajo será el lenguaje C, por lo que vamos a emplear la nomenclatura de C para la descripción de las funciones propias de PVM.

Cómo obtener PVM.

La vía más cómoda para obtener PVM es a través de *netlib*, lo que podemos hacer en *http://www.netlib.org/pvm3/index.html*. *Netlib* es tanto una biblioteca de cálculo científico como el sitio que la mantiene. Allí encontraremos PVM, gran cantidad de rutinas de cálculo numérico y la implementación de algunas de las rutinas de cálculo numérico empleando PVM. El acceso a *netlib* se puede hacer a través de una herramienta, Xnetlib, que nos permite hacer una navegación automática por su sitio de Internet, vía FTP

anónimo -en netlib2.cs.utk.edu- o vía WWW a netlib o cualquiera de sus imágenes. Si se realiza una búsqueda en cualquiera de los buscadores existentes en Internet, veremos que la cantidad de software y documentación existente para PVM es bastante grande. También podemos encontrar gran cantidad de herramientas, sobre todo de depuración y monitorización en Internet. Si somos usuarios de Linux, podemos encontrar paquetes en formato rpm tanto del ejecutable de PVM como de los ejemplos y de muchas de las herramientas.

Versiones de PVM

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[2]. Esta versión fue un desarrollo interno que no se llegó a hacer pública. La versión 2 de PVM fue realizada en la Universidad de Tennessee, y se hizo pública en 1991. En un año se realizaron una gran cantidad de aplicaciones para PVM, lo que produjo una gran realimentación, con las consiguientes 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. Actualmente la versión estable es la 3.3, aunque se puede encontrar una versión beta de la 3.4.

Cómo compilar PVM

El mecanismo de compilación e instalación es sencillo, lo que fue una de las razones de la popularidad de la biblioteca.
Primero descomprimimos el paquete. Después son precisas definir dos variables de sistema: la variable *PVM_ROOT*, que ha de apuntar al directorio donde esta descomprimido el paquete de PVM, y *PVM_ARCH*, que ha de contener el nombre de la arquitectura. En el caso particular de Linux, esta variable ha de contener la cadena LINUX. Una vez definidas estas dos variables de entorno, ya sólo tenemos que entrar en el directorio *PVM_ROOT* y hacer *make*. El sistema emplea la variable *PVM_ARCH* para seleccionar el *Makefile* adecuado para la arquitectura, y realiza la compilación.

Si la compilación funcionó de forma correcta, en el subdirectorio *lib* del directorio apuntado por la variable de entorno *PVM_ROOT* se crearán los archivos *libpvm3.a* -biblioteca básica para C-, *libgpvm3.a* -biblioteca de grupos-, *libfpvm3.a* -biblioteca para Fortran-, *pvm* -monitor de PVM- y *pvmd3* -demonio de PVM-; y en el subdirectorio *bin* de dicha variable de entorno se creará el archivo *pvmgs*.

[2] PVM. A Users Guide and Tutorial for Networkerd Parallel Computing.
Al Geist
MIT Press.

Cómo instalar PVM

Hay dos maneras: el compilar PVM para nuestro sistema y plataforma, o por rpm.

1.Compilando PVM

La versión 3 de PVM -la actualmente empleada- va a intentar estar por defecto para cada usuario dentro de su directorio *$HOME/pvm3/bin/PVM_ARCH*, donde *PVM_ARCH* es la arquitectura de nuestro sistema -recordamos que la definimos en la etapa de compilación-; este es, el directorio donde tendremos que copiar los ejecutables. Un directorio tan rebuscado nos puede parecer extraño, mas, si tenemos en cuenta la capacidad de PVM de correr en entornos heterogéneos y que habitualmente el directorio donde va a estar instalada la biblioteca es compartido por todas las máquinas de la red, es un mecanismo que se revela indispensable -así podemos tener ejecutables para varias arquitecturas distintas de forma cómoda-. Además, se puede instalar como usuarios; de ahí que dependa de la ubicación de *$HOME*.

2.Instalación por rpm.

En caso de que estemos instalando el paquete rpm, basta con descomprimirlo con la instrucción -en el caso de la versión 3.4.3-

#rpm –i pvm-3.4.3-4.i386.rpm

y se instalará todo automáticamente, sólo que, en este caso, vamos a tener que ser el superusuario para hacerlo.

Un aspecto importante es la definición del conjunto de máquinas que formarán parte de PVM. Esto se hace dentro del archivo *.xpvm_hosts* - para el ambiente gráfico y con el archivo *pvmrc* para la consola- , archivo en el que incluiremos los nombres de todas las máquinas que pretendamos usar. En el caso de que sólo queramos indicar que podemos emplear la máquina, y no que sea inicializada PVM en la máquina cuando arrancamos el programa monitor, se incluye el signo & antes del nombre de la máquina en el archivo.

Por último, podemos añadir al archivo *.rhosts* el nombre de todas las máquinas que puede emplear. Esto crea un posible agujero de seguridad, así que debemos tener cuidado y llegar a un compromiso entre comodidad y seguridad. Otra manera de dar una lista de las máquinas en las que se puede ejecutar remotamente PVM es por medio de crear el archivo */etc/host.equiv* , el cual nos permite que ya no se agregue usuario por usuario el archivo *.rhosts* en su directorio *home*, sino que todos lo puedan usar.

Manejo del programa *pvm*

El programa *pvm* va a corresponder al núcleo de control de la máquina virtual, o, dicho de otra forma, al monitor de la máquina. A todos los efectos, se va a comportar como una consola dentro de la máquina virtual; por eso es denominado muchas veces, simplemente como consola. Cuando lo ejecutamos, entramos en el intérprete de comandos de nuestra máquina virtual. Él lanza el demonio de PVM en el caso de que no lo tengamos lanzado, y además nos permitirá ejecutar las funciones básicas de control de PVM. Un resumen de estas funciones se describe un poco más adelante.
Si invocamos al programa *pvm* de la forma *pvm archivo*, el archivo indicado será interpretado como la lista de máquinas que será incorporado a PVM.
El monitor tiene una versión gráfica, denominada *xpvm*, que nos va a permitir hacer exactamente lo mismo, pero de una forma más visual.

El TID

El TID -*Task Identifier*- es el identificador de tarea, y se comporta como el PID de un sistema Unix, sólo que de nuestra máquina virtual. Está definido como un entero de 32 bits para aumentar el rendimiento al máximo y, al mismo tiempo, permitir direccionar el mayor número de tareas posibles.
El TID está compuesto de 4 campos. Un primer campo es el bit S, que ocupa la posición de bit más significativo. Un segundo campo es el bit G, que ocupa la siguiente posición. Un tercer campo, de 12 bits, es el campo H; y un cuarto campo, formado por los 18 bits menos significativos es el campo L.

Los campos S, G y H son de representación independiente de la máquina que corra el demonio. Los campos S y G se van a emplear para informar qué significado tienen los campos H y L, según la tabla adjunta. El campo H va a ser un campo función exclusivamente de la máquina que asigna el TID, y el campo L va a ser asignado por la máquina que determine el campo H. Estos datos ya nos permiten determinar que podremos tener hasta un máximo de 4095 máquinas en una PVM y hasta un máximo de 262143 procesos corriendo simultáneamente en cada máquina. La máquina cero puede ser tanto el demonio local como el *shadow pvmd*, del cual se hablará a detalle más adelante.

Todos los TIDs se mantienen sincronizados con los valores de una tabla de asignación global. Hay que observar que no hay posibilidad de colisión entre dos máquinas distintas, ya que el campo H es un valor que define unívocamente la máquina dentro de PVM.

Los bits S y G

Los significados del TID y los rangos de los campos H y L según el valor de los bits S y G son (Tabla 4.3):

S	G	H	L	Uso
0	0	1..4096	1..262143	Identificador de tarea.
1	0	1..4096	0	Identificador de demonio de PVM.
1	0	0	0	Demonio de PVM local a una tarea, para la tarea.
1	0	0	0	Shadow pvmd para el demonio que genera las tareas.
0	1	1..4096	0..262143	Dirección de multienvío
1	1			Condición de error

Tabla 4. 3 Significado del TID de PVM.

En el caso de la condición de error, H y L se unen formando un número entero negativo. Como el número de errores distintos es pequeño, este número es pequeño.

Las clases de arquitectura

Para evitar el problema de realizar transformaciones continuas de datos, PVM define clases de arquitecturas. Antes de mandar un dato a otra máquina comprueba su clase de arquitectura. Si es la misma, no necesita convertir los datos, con lo que se tiene un gran incremento en el rendimiento. En caso que sean distintas las clases de arquitectura se emplea el protocolo XDR para codificar el mensaje.
Las clases de arquitectura están mapeadas en números de codificación de datos, que son los que realmente se transmiten y, por lo tanto, los que realmente determinan la necesidad de la conversión.

El modelo de paso de mensajes de PVM

El modelo de paso de mensajes es transparente a la arquitectura para el programador, por la comprobación de las clases de arquitectura y la posterior codificación con XDR de no coincidir las arquitecturas. Los mensajes son etiquetados al ser enviados con un número entero definido por el usuario, y pueden ser seleccionados por el receptor tanto por dirección de origen como por el valor de la etiqueta.

El envío de mensajes no es bloqueante. Esto quiere decir que el que envía el mensaje no tiene que esperar a que el mensaje llegue, sino que solamente espera a que el mensaje sea puesto en la cola de mensajes. La cola de mensajes, además, asegura que los mensajes de una misma tarea llegarán en orden entre sí. Esto no es trivial, ya que empleando UDP puede que enviemos dos mensajes y que lleguen fuera de orden -UDP es un protocolo no orientado a conexión-. TCP, por ser un protocolo orientado a la conexión, realiza una reordenación de los mensajes antes de pasarlos a la capa superior; sin embargo, recordamos que establecer las conexiones entre nodos empleando TCP supone, si tenemos n nodos, tener un mínimo de $(n)(n-1)$ conexiones TCP activas.

La comunicación de las tareas con el demonio sí se hace, en cambio, empleando TCP. Esto se debe a que, al ser comunicaciones locales, la carga derivada de la apertura y cierre de un canal es muy pequeña. Además, no vamos a tener tantas conexiones como en el caso de la conexión entre demonios, ya que las tareas no se conectan entre sí ni con nada fuera del nodo, por lo que sólo hablan directamente con su demonio. Esto determina que serán *n* conexiones TCP, que sí es una cifra razonable.

La recepción de los mensajes podemos hacerla mediante instrucciones bloqueantes, no bloqueantes o con un tiempo máximo de espera. PVM nos dotará de instrucciones para realizar los tres tipos de recepción. En principio nos serán más cómodas las bloqueantes, ya que nos darán un mecanismo de sincronización bastante cómodo. Las de tiempo máximo de espera nos serán útiles para trabajar con ellas como si fuesen bloqueantes, mas dando soporte al hecho de que puede que el que tiene que mandarnos el mensaje se haya colgado. Por último, la recepción de mensajes mediante instrucciones no bloqueantes hace de la sincronización un dolor de cabeza. El mecanismo más cómodo para sincronizar en estos casos será una sincronización de barrera, que comentaremos más adelante.

De cualquier forma, en los tres casos anteriormente citados la misma PVM se encargará de decirnos cuándo una tarea acabó o se queda sin terminar. Observamos que hablamos de que la tarea se quede procesando sin terminar. PVM no tiene forma de saber si una tarea demora en responder porque está realizando cómputo o porque está colgada; por ello, para detectar los cuelgues deberemos emplear las esperas con tiempo límite. Para informarnos de lo que pasa, emplea un mecanismo de eventos asíncronos.

Los eventos asíncronos

Los eventos asíncronos corresponden al equivalente de las señales del sistema en Linux. Se producen cuando acontece algo inaudito, y podemos pedir al sistema que nos interrumpa independientemente de lo que estemos haciendo para notificarnos de que algo anormal está aconteciendo. Los eventos pueden viajar de una máquina a otra, para informar a todas aquellas que lo precisen. Estos eventos corresponden a cosas tan traumáticas como una tarea que acaba o se queda trabada, una máquina es eliminada o igualmente se queda trabada, o la incorporación de máquinas.

Descriptores de mensaje de los eventos asíncronos

Los distintos descriptores de mensaje de los eventos asíncronos son:

- *PvmTaskExit*: Una tarea acaba o aborta. Una tarea colgada no puede ser detectada, por lo que si una tarea se cuelga no se generará un mensaje.
- *PvmHostDelete*: Una máquina es eliminada del conjunto de máquinas o se apaga.

- *PvmHostAdd*: Se incorpora una máquina nueva al conjunto de máquinas.

4.7.1. FUNCIONAMIENTO DE PVM

El demonio de PVM con detalle

El demonio de PVM puede ser arrancado bien por un monitor local, o por un monitor remoto. Lo normal es que en el caso del monitor local sea arrancado como maestro y en el caso del remoto sea arrancado como esclavo, mas esto puede ser modificado en la propia forma de invocación. Después de arrancar, crea los sockets que necesita para hablar con los otros demonios y abre un archivo de eventos de error en */tmp/pvml.uid*, donde *uid* es el identificador de usuario propietario de PVM. Cuando el demonio de PVM ve que su máquina es desconectada del conjunto de máquinas, mata todas las tareas conectadas a él mediante una señal *SIGTERM*. Después manda un mensaje a la tabla de máquinas para que elimine su entrada. En caso de salidas más salvajes - matando con el comando *kill* el demonio- PVM demorará un poco más en descubrir que un nodo no está funcionando, pero acabará descubriéndolo y eliminará el nodo de la tabla de máquinas.

Arranque de los demonio esclavos

El arranque es realizado por el *shadow pvmd*, que no es mas que un *fork()* que realiza *pvmd* para poder seguir atendiendo mensajes mientras que el *shadow pvmd* -el proceso hijo- se encarga de la creación de nuevos nodos, éste proceso es más lento. Se puede hacer mediante *rsh* o mediante *rexec*.

La elección de la forma de arranque presenta algunos problemas:

rsh es un programa que permite lanzar tareas remotas sin autentificación de usuario. Para que funcione, tenemos que tener actualizados en todas las máquinas el archivo */etc/hosts.equiv* para que todo el cluster aparezca en él, o incluir las máquinas en el archivo *.rhosts*.

Lo primero solamente tiene sentido en el caso de una red sin salida al exterior, o una red que la gestión de las claves sea centralizada, en cuyo caso no produce fallos en la seguridad el hecho de que sean definidas las máquinas como equivalentes. Esto ayuda en cluster de tipo dedicado.
La segunda posibilidad es creando un archivo *.rhosts* en la máquina remota. Esto sí es una mala idea por razones de seguridad, pero si se va a utilizar un conjunto de máquinas como cluster no dedicado es una opción viable, pero el administrador del sistema puede haber desactivado las utilidades remotas del sistema para evitar problemas de seguridad, en cuyo caso se debe de buscar una alternativa.

La alternativa es la función *rexec()* del demonio *pvmd*. El problema es que, si hacemos esto, o introducimos la clave desde teclado cada vez que activamos el nodo -lo que es realmente incomodo- o incluimos la clave en el archivo *.netrc*, lo que sí que es realmente la peor de las ideas. Desgraciadamente, demasiados sistemas tienen las claves de los usuarios en los archivos *.netrc* -con el consiguiente agujero de seguridad- porque el administrador considera que son demasiado inseguros los servicios remotos para estar funcionales en las máquinas.

4.7.2. COMANDOS DE PVM

Comandos fundamentales del programa PVM

El programa PVM corresponde al intérprete de comandos de nuestra máquina virtual. Algunos de los comandos más importantes son:

- *add* máquina: incorpora la máquina indicada a la máquina paralela virtual.
- *delete* máquina: elimina la máquina indicada del conjunto de máquinas asociadas a la máquina paralela virtual. Como es lógico, no podremos eliminar la máquina desde la que estamos ejecutando el intérprete de comandos.
- *conf*: configuración actual de la máquina paralela virtual.
- *ps*: listado de procesos de la máquina paralela virtual. *ps -a* lista todos los procesos.
- *halt*: apaga la máquina paralela virtual. Esto significa que mata todas las tareas de PVM, elimina el demonio de forma ordenada y sale del programa *pvm*. Es la forma de salir en condiciones, y mucho mejor que matar el demonio con *kill*.
- *help*: lista los comandos del programa.
- *id*: imprime el TID de la consola.
- *jobs*: genera un listado de los trabajos en ejecución.
- *kill*: mata un proceso de PVM.
- *mstat*: muestra el estado de una máquina de las pertenecientes a PVM.
- *pstat*: muestra el estado de un proceso de los pertenecientes a PVM.
- *reset*: inicializa la máquina. Eso supone matar todos los procesos de PVM salvo los programas monitores en ejecución, limpiar las colas de mensajes y las tablas internas y pasar a modo de espera todos los servidores.
- *setenv*: lista todas las variables de entorno del sistema.
- *sig* señal tarea: manda una señal a una tarea.

- *spawn*: arranca una aplicación bajo PVM. Es un comando bastante complejo cuyas opciones veremos en una sección aparte.
- *trace*: actualiza o visualiza la máscara de eventos trazados.
- *alias*: define un alias predefinido, es decir, un atajo para teclear un comando.
- *unalias*: elimina un alias predefinido.
- *version*: imprime la versión usada de PVM.

Cualquiera de estos comandos, o cualquier combinación de ellos puede ser definida en el archivo *.pvmrc*; dicho archivo será leído y ejecutado al arrancar cada monitor.

Los monitores pueden ser arrancados en cualquier momento.

Modificadores spawn

El comando de monitor *spawn* nos permite lanzar aplicaciones para PVM. Algunas de los modificadores que emplea son:

- *-count*: número de tareas involucradas. Por defecto es 1.
- *-host*: especifica la máquina para lanzar. Por defecto es *any*.
- *-ARCH*: lanza en máquinas de tipo ARCH (arquitectura).
- *-?*: Activa el modo depuración.
- ->: Redirecciona la salida de la tarea que se lanzará al monitor.
- ->archivo: Redirecciona la salida de la tarea que se lanzará al archivo indicado.
- ->>archivo: Redirecciona la salida de la tarea que se lanzará al archivo indicado, concatenando al final.
- -@: Traza el trabajo, redirigiendo la salida al monitor.
- *-@file*: Traza el trabajo, redirigiendo la salida al archivo indicado.

Opciones del archivo de máquinas

El archivo donde se especifican las máquinas es de extrema importancia, ya que define la arquitectura de nuestra máquina paralela virtual. Se compone de una lista de máquinas, que pueden llevar o no el símbolo & al principio según no sean o sean inicializadas al arrancar el monitor, respectivamente.

Las líneas de comentario son aquellas que comienzan con el símbolo #. Además de esto, cada máquina puede llevar asociada una serie de parámetros para determinar su configuración. Los más importantes son:

- *lo*=usuario: permite identificar un nombre de usuario distinto para la conexión a la máquina indicada. Por defecto, se emplea el nombre del usuario que lanzó el proceso en la máquina actual.
- *so*=clave: especifica la clave de la máquina remota. No es una buena idea emplear este campo, salvo que la red sea privada. Por defecto intenta una conexión con *rsh*, que sólo funcionará si en la máquina remota está la máquina local como una entrada al archivo *.rhosts* y las RPC están activadas.
- *dx*=ruta: donde se encuentra el *pvmd3*
- *ep*=rutas: conjunto de rutas, separadas por ";", donde se van a encontrar los ejecutables que se van a lanzar en la máquina. Si no se especifica, buscará en el directorio *$HOME/pvm3/bin/PVM_ARCH*.
- *sp*=valor: valor de potencia computacional relativa respecto a las otras máquinas. Los rangos posibles están entre 1 y 1000000. Es importante tener en cuenta la carga habitual de una máquina. Puede que una máquina de baja potencia sin carga tenga un valor más alto que una máquina de alta potencia con una carga excesiva. El valor por defecto es 1000.
- *bx*=ruta: localización del depurador. Sólo es activado si también se activa la opción de depuración.
- *wd*=ruta: directorio de trabajo. Es, pues, donde las tareas lanzadas se van a ejecutar. Por defecto apunta a *$HOME*.
- *ip*=dirección: especifica la dirección IP en la que se va a buscar la máquina.
- *so*=ms: significa que el servidor esclavo no va a ser lanzado con *rsh*, sino de forma manual. Sólo es útil si no disponemos de RPC o no queremos utilizarlo por razones de seguridad.

Un parámetro por defecto puede ser redefinido con una línea que comience por *. En ella se definen los parámetros como si el * fuese una máquina, y los parámetros definidos de esa forma serán los parámetros por defecto hasta nueva redefinición.

4.7.3. PROGRAMACIÓN BAJO PVM

Aunque en principio cómo esté codificado un programa que emplee PVM es algo muy personal de su programador, la estructura básica de un programa para PVM puede ser organizada en un conjunto de bloques estructurales que realizan determinadas funciones básicas.

Envío de datos

El envío y la recepción de datos son la piedra angular del modelo de PVM. Para enviar datos, vamos a emplear tres pasos:

- Inicialización del buffer
- Empaquetado del dato
- Enviar el dato

Recordamos que los datos van codificados con XDR, y que, por ello, es preciso empaquetar el dato para asegurarnos de que sea transmitido de una forma correcta.

Cuando comienza a ejecutarse un programa para PVM, tiene ya un buffer activo de emisión y otro de recepción que serán empleados para comunicarse con otros nodos. Si queremos crear algún buffer adicional, tenemos que emplear la función:

int bufid = pvm_mkbuf(int decod)

Donde *bufid* será el identificador de *buffer* y *decod* la forma de codificación. Habitualmente no es preciso crear *buffers* adicionales, por lo que muy raramente emplearemos esta instrucción.

Los *buffers* pueden ser eliminados con la instrucción:

int info = pvm_freebuf(int bufid)

Donde *bufid* es el identificador del buffer que se va a eliminar.
Hemos de recordar activar el buffer creado como buffer de envío.
Otra forma bastante más cómoda de la que disponemos para mandar datos es con la instrucción *pvm_psend()*, que permite mandar una matriz contigua de datos del mismo tipo. De cualquier forma, esta función tiene gran cantidad de limitaciones.

Inicialización del buffer

Siempre que vayamos a empaquetar un mensaje, tenemos que ejecutar la instrucción *pvm_initsend()*, que inicializará el buffer activo. En caso que no lo hagamos, los datos que se manden serán los que empaquetemos ahora, más los que teníamos ya empaquetados previamente en el buffer. La sintaxis es:

int bufid = pvm_initsend(int encod)

El identificador de buffer devuelto por la función será el del buffer inicializado -el activo-. El parámetro que se pasa a la función es la codificación, de definición similar a la del mismo parámetro en la instrucción de creación de un buffer nuevo.

Empaquetado de datos

En PVM los datos no se pueden mandar tal y como están. Es preciso empaquetar los datos en un buffer y después mandar el buffer. Aquí tenemos una gran diferencia entre C y Fortran. Fortran tiene una instrucción para empaquetar, mientras que C tiene catorce distintas. El formato en once de ellas es:

int info = pvm_pkXXX(YYY *que, int cuantos, int desde_donde)

donde *XXX* es el tipo en PVM de lo que vamos a mandar -*byte, cplx, long*...- e *YYY* el tipo en C relacionado. Para empaquetar una cadena no indicamos ni cuántos ni el desplazamiento, siendo la instrucción de la forma:

int info = pvm_pkstr(char *cadena)

Otra forma de empaquetamiento lo presenta la función:

int info=pvm_packf(const char *formato, ...)

que se invoca de la misma forma y con los mismos parámetros que el *printf* de C, con lo que se reducen las otras 13 funciones a una sola.

Bajo ningún concepto ningún mensaje puede ser mayor de 32 Mbytes.

Tipos de datos para empaquetado y desempaquetado

Tipo	Tipo C del 1er parámetro	1er Parámetro Fortran
byte	char *	BYTE1
Entero 1 byte	short *	BYTE1
Entero 2 bytes	int *	INTEGER2
Entero 4 bytes	long int *	INTEGER4
Flotante 4 bytes	float *	REAL4
Flotante 8 bytes	double *	REAL8
Complejo 8 bytes	float *	COMPLEX8
Complejo 16 bytes	double *	COMPLEX16
Cadena	Char *	STRING

Tabla 4. 4 Tipos de datos de desempaquetado en PVM.

Funciones para empaquetado y desempaquetado

Tipo	Empaquetado	Desempaquetado
byte	pvm_pkbyte	pvm_upkbyte
Entero 1 byte	pvm_pkshort	pvm_upkshort
Entero 2 bytes	pvm_pkint	pvm_upkint
Entero 4 bytes	pvm_pklong	pvm_upklong
Flotante 4 bytes	pvm_pkfloat	pvm_upkfloat
Flotante 8 bytes	pvm_pkdouble	pvm_upkdouble
Complejo 8 bytes	pvm_pkcplx	pvm_upkcplx
Complejo 16 bytes	pvm_pkdcplx	pvm_upkdcplx
Cadena	pvm_pkstr	pvm_upkstr

Tabla 4. 5 Funciones de desempaquetado y desempaquetado en PVM.

Envío de datos

Lo podemos realizar con la función:

int info= pvm_send(int TID, int etiqueta)

Esta rutina manda el mensaje que se guarda en el buffer activo al proceso en PVM identificado por el TID que se incluye. La etiqueta es usada para marcar el contenido del mensaje. Así el que recibe, al tener la misma etiqueta, sabe cuales datos le están llegando.

Para mandar el buffer por defecto a todo un conjunto de tareas, podemos emplear la función:

int info = pvm_mcast(int *TIDs, int número, int etiqueta)

donde el primer parámetro será una matriz de TIDs y el segundo el número de TIDs contenidos en la matriz.
Es importante que tengamos en cuenta que las comunicaciones se realizan mandando el mensaje solamente con la etiqueta de una tarea o de un conjunto de tareas. Si una tarea está bloqueada esperando otra etiqueta, no escuchará nuestro mensaje.

La emisión no es bloqueante para el emisor respecto a la tarea del receptor, mas sí respecto al demonio del emisor. Esto significa que la tarea cliente quedará bloqueada en la emisión hasta que todo el mensaje sea transmitido vía TCP para el demonio. Después, la tarea continuará ejecutándose, y el servidor intentará contactar con el demonio de la tarea receptora.

Recepción de datos

Para recibir datos, vamos a emplear dos pasos:

- Recibir el mensaje
- Desempaquetar el dato

Exactamente igual que el envío, se emplea un buffer, sólo que ahora no hay que limpiarlo antes de recibir cada dato.

Es importante que recordemos que el buffer de envío y de recepción al arrancar una tarea en PVM, son distintos; y tienen distintas instrucciones para ser activados.

Recepción del mensaje

Al recibir el mensaje, la tarea receptora queda bloqueada contactando con su servidor para una etiqueta determinada. Según el servidor esté con un mensaje listo en el buffer con una etiqueta para dicha tarea o no, y según el modo de recepción, la tarea quedará bloqueada o no esperando un mensaje. De cualquier forma, mientras el mensaje es transmitido entre el demonio receptor y la tarea receptora, la tarea receptora es bloqueada. El mensaje es transmitido entre demonio receptor y tarea receptora vía TCP. Además de todo esto, la espera entre emisión del mensaje y recepción puede ser bloqueante o no según la función de recepción. Estas son:

- ***int bufid=pvm_recv(int TID, int etiqueta)*** : bloqueante. El receptor esperará a que la tarea TID le mande un mensaje con la etiqueta indicada, entonces transferirá el mensaje al buffer de recepción por defecto -*bufid*-.

- ***int bufid=pvm_nrecv(int TID, int etiqueta)***: no bloqueante. Si encontramos en el servidor un mensaje de la tarea TID con la etiqueta adecuada ya lista, transferimos el mensaje de forma bloqueante al buffer de recepción por defecto -*bufid*-. Si no, la función devuelve como *bufid* el valor 0, y no queda bloqueada.

- ***int bufid=pvm_trecv(int TID,int etiqueta, struct timeval *tiempo)***: bloqueante con temporalización. El receptor estará bloqueado durante el tiempo indicado, tiempo que se comporta como la recepción bloqueante. Transcurrido ese tiempo, pasa a comportarse como recepción no bloqueante, por lo que, si aún no llegó mensaje, devolverá el valor 0.

Una función interesante es *pvm_probe*, que verifica si ha llegado un mensaje determinado. Su comportamiento es exactamente igual que el de *pvm_nrecv*, sólo que no recibe el paquete; por lo que para recibirlo sigue siendo precisa una instrucción de recepción de paquete. Su sintaxis es:

int bufid = pvm_probe(int TID, int etiqueta)

aún sin bajarlo, podemos obtener información detallada del mensaje con la función *pvm_bufinfo*, que da información sobre el mensaje recibido antes de descargarlo; basta con combinar su uso con *pvm_probe*.

De cualquier forma, si no estamos satisfechos con la semántica de la recepción, podemos emplear la función *pvm_recvf*, que nos permitirá redefinir el comportamiento de la recepción.

Cualquiera de las funciones de recepción antes mencionadas pueden tener como parámetro TID el valor -1, lo que significa cualquier tarea. Del mismo modo, una etiqueta con valor -1 significa cualquiera.

Envío y recepción entre dos tareas determinadas siempre se realizan en orden. Esto significa que si una tarea manda dos mensajes a otra, los mensajes serán recogidos por la tarea destinataria en el mismo orden en el que fueron mandados. Por otro lado, si dos tareas emiten cada una un mensaje a otra tercera, no tenemos forma de saber en qué orden serán recogidas por el receptor, aunque sepamos que una tarea envió mucho antes que otra. Los mensajes recibidos de distintas tareas se pueden recibir fuera de orden.

Desempaquetado de datos

El desempaquetado de datos es exactamente igual que el empaquetado de datos, con sus catorce funciones y con todos sus parámetros iguales, solo que cambiando *pvm_pk* por *pvm_upk*. Es decir, el formato en once de ellas es:

int info = pvm_upkXXX(YYY *que, int cuantos, int desde_donde)

donde *XXX* es el tipo en PVM de lo que vamos a mandar -*byte, cplx, long*...- e *YYY* el tipo en C relacionado, y para la cadena tampoco indicamos ni cuántos ni el desplazamiento, siendo la instrucción de la forma:

int info = pvm_upkstr(char *cadena)

Exactamente igual que con el empaquetado de datos, tenemos la función:

int info=pvm_upackf(const char *formato, ...)

que tiene el mismo formato que el *scanf* de C .

Por último, es importante destacar un error frecuente. Tenemos que desempaquetar los datos en el mismo orden que fueron empaquetados, porque en muchas aplicaciones recibiremos basura.

Sincronización de barrera

Este mecanismo de sincronización tiene como idea básica no dejar a ninguna tarea pasar hasta que un número suficiente de tareas estén esperando en la barrera. El punto de espera se define con la función *pvm_barrier*. El problema básico es que precisa saber cuántos procesos han de esperar en la barrera antes de dejarlos pasar a todos, información que el maestro conoce, mas los esclavos no; o, al menos, no basta con saber cuántos están en un grupo, ya que puede que algunos elementos todavía no hayan entrado. De ahí que formemos un grupo, y el maestro mande esta información a todo el grupo con *pvm_mcast*. El formato de la instrucción es:

int info=pvm_barrier(char *grupo, int cuantos)

Donde el primer parámetro es el grupo sobre el que estamos realizando la función, y el segundo cuántos tienen que esperar para pasar. En caso de que empleemos el grupo sólo para sincronía, podemos indicar en el segundo parámetro *-e*, lo que significa que tomará todos los miembros del grupo en ese momento *-función pvm_gsize-* y lo empleará como parámetro, por lo que no será preciso que el maestro nos mande la información -en teoría-. En la práctica, puede ocurrir que distintos procesos tengan distintos valores de *pvm_gsize* porque se ejecutan en distintos momentos, por lo que el -1 puede ser un desastre si no tenemos seguridad de que todos los miembros del grupo ya están registrados - para lo cual necesitaríamos otro bloqueo de barrera, por lo que estamos en las mismas, y acabamos haciendo el *pvm_mcast* -.

Salida de PVM

La salida es bastante fácil, basta con ejecutar la función *pvm_exit()*. Podríamos desde el maestro ir matando los esclavos con *pvm_kill*, pero es un procedimiento bastante poco recomendable.

Variables de entorno modificables por el usuario

Existen un conjunto de variables de entorno que podemos modificar. Estas son:

- *PVM_ROOT*: Ruta del directorio de instalación de PVM.
- *PVM_EXPORT*: Nombre de las variables del sistema que serán heredadas a los hijos lanzados con *spawn*.
- *PVM_DPATH*: Ruta del directorio de instalación de PVM en los esclavos, por defecto.
- *PVM_DEBUGGER*: Ruta del *script* de depuración empleado por *spawn*.

Variables de entorno no modificables por el usuario

Existen variables de entorno que no se pueden modificar, pero que es interesante conocer para poder leer sus valores. Estas son:

- *PVM_ARCH*: Nombre de la arquitectura de la máquina local.
- *PVMSOCK*: Dirección del *socket* del demonio de PVM local.
- *PVMEPID*: PID supuesto de la tarea que se va a lanzar con *spawn* esperado.
- *PVMTMASK*: Máscara de depuración de la *libpvm*.

Opciones del parámetro flag para lanzar tareas

- *PvmTaskDefault*: PVM decide en qué máquina va a lanzar las tareas.
- *PvmTaskHost*: El parámetro *host* indica en qué máquina se van a lanzar las tareas.
- *PvmTaskArch*: El parámetro *host* indica en qué arquitectura -conjunto de máquinas con la misma variable *$PVM_ARCH-* se van a lanzar las tareas.
- *PvmTaskDebug*: La tarea será lanzada junto con el depurador.
- *PvmTaskTrace*: Se mantendrá un histórico para las llamadas a las rutinas de PVM para esa tarea.
- *PvmMppFront*: Las tareas serán lanzadas con un *front-end* de MPP.
- *PvmHostCompl*: Las tareas serán lanzadas en aquellos nodos que no cumplan las características indicadas.

Para indicar varias opciones, lo hacemos asociándolas con el operador ||.

Opciones de codificación del buffer

El buffer admite varias formas distintas de codificación, seleccionables empleando *flags*. Estos son:

- *PvmDataRaw*: Los mensajes serán mandados sin codificar en *XDR*. Si al desempaquetar no es entendido el formato, generará un error.
- *PvmDataDefault*: Los mensajes serán mandados codificados con *XDR*.
- *PvmDataInPlace*: En el buffer tendremos punteros a datos y tamaños de datos, en lugar de datos. Acelera bastante algunas operaciones, pero se ha de tener cuidado con modificar los datos indiscriminadamente después de empaquetar.

Para indicar varias opciones, se asocian con el operador ||. *PvmDataRaw* y *PvmDataDefault* al mismo tiempo no tienen mucho sentido.

Otras funciones de mantenimiento de tareas

PVM nos permite desde el código controlar las distintas tareas con un conjunto de funciones. Estas son:

- Matar la tarea con el TID indicado. Para salir es mejor *pvm_exit* que matarse a sí mismo.

  int rc = pvm_kill(int tid)

- Devuelve el TID del proceso que lanzó la tarea, o *PvmNoParent* si fue lanzada directamente por el usuario.

  int tid = pvm_parent()

- Devuelve información sobre una tarea.

  int status = pvm_stat(int tid)

Funciones para operar sobre el conjunto de máquinas

Así como podemos operar sobre el conjunto de máquinas desde un archivo de configuración, también podemos hacerlo desde el propio programa. Para operar sobre el conjunto de máquinas desde el programa las funciones son:

- Añade un conjunto de máquinas de PVM.

  pvm_addhosts(char **hosts, int nhost, int *infos)

- Elimina un conjunto de máquinas de PVM.

  pvm_delhosts(char **hosts, int nhost, int *infos)

- Devuelve *PvmOk* si la máquina indicada está en PVM, *PvmNoHost* en cualquier otro caso. Útil para que la aplicación sea tolerante a fallos.

  int pvm_mstat(char *host)

- Devuelve información acerca del estado de la máquina paralela virtual. *Nhost* será el número de máquinas, *narch* el número de arquitecturas y la de hosts será un puntero a una matriz con los datos de todos los nodos -TID, nombre, arquitectura y velocidad relativa-.

  int pvm_config(int *nhost, int *narch, struct pvmhostinfo **hosts)

- Para la máquina paralela virtual.

  int rc = pvm_halt()

- Arranca el demonio para la máquina donde se ejecuta la tarea -si procede-.

int dtid = pvm_start_pvmd(int args, char **argv, int block)

- Devuelve el identificador de máquina de un proceso. Solamente extrae el dato del TID sin comprobar ni que exista ni que esté activo.

  int dtid = pvm_tidtohost(int tid)

Funciones para operaciones con buffers

Podemos realizar operaciones con los *buffers* empleando las funciones:

- Devuelve el identificador del buffer activo de recepción.

  int bufid=pvm_getrbuf()

- Devuelve el identificador del buffer activo de emisión.

  int bufid=pvm_getsbuf()

- Activa un buffer para emisión.

  int bufid=pvm_setsbuf()

- Activa un buffer para recepción.

  int bufid=pvm_setrbuf()

Otras operaciones con grupos de PVM

Además de las operaciones con grupos básicas ya explicadas, disponemos de otras operaciones adicionales bastante interesantes. Estas son:

- *pvm_gather*: recoge los datos de todo un grupo para almacenarlos en una matriz. Lo opuesto a *pvm_scatter*.
- *pvm_gettid*: devuelve el TID de una tarea para un número de instancia.
- *pvm_reduce*: Aplica un operador de reducción -sumatoria, producto, máximo, mínimo y otros definidos por el usuario- a los miembros de un grupo, generando un solo escalar.
- *pvm_scatter*: distribuye datos de una tarea a las restantes de un grupo. Los datos están originariamente en una matriz, y se mandan los *count* primeros a la primera, los *count* siguientes a la siguiente y así los restantes.
- *pvm_getinst*: devuelve el número de instancia de una tarea dentro de un grupo.

Las operaciones de grupo sólo se deben realizar con grupos estables. Por ello es recomendable bloquear con pvm_barrier después de la inscripción de las

tareas en un grupo, para que solamente pasen todas cuando todas estén inscritas.

4.7.4. COMPILACIÓN DE UN PROGRAMA EN PVM.

Una vez que ya se tiene el código del programa que se quiere ejecutar dentro de PVM, hace falta realizar algunas acciones más. Dentro del mismo programa se tiene que incluir la librería *pvm3.h*, que es la que se va a encargar de proporcionarnos las funciones de *PVM* que se estén utilizando.

Una vez verificado que se tiene esta cabecera en el programa, en el indicador del sistema se ejecuta un compilador de lenguaje C, indicando que ligue al programa las librerías de *PVM* como son *lpvm3* -biblioteca básica para C-, *lgpvm3* -biblioteca de grupos-. Para esto se le indica en dónde las va a localizar (*-I /usr/share/pvm3/include*) y desde dónde las va a ligar (*-L /usr/share/pvm3/lib/LINUX*). En caso de utilizar funciones matemáticas, se liga también las librerias de matemáticas *lm*:

cc prog_fuente.c -o prog_exe -I /usr/share/pvm3/include -L /usr/share/pvm3/lib/LINUX -lgpvm3 -lpvm3 -lm

4.8. EJEMPLOS DE PROGRAMACIÓN EN PVM

4.8.1. PROGRAMA HOLA

El siguiente programa, da a conocer el paradigma de programación maestro – esclavo. Partimos de un programa maestro hola.c que tomará el control de todas las tareas y otro llamado hola_desde.c que sirve de base para expandir cada una de las tareas.

El presente paradigma de programación tiene por objetivo mostrar cómo se lleva a cabo el paso de mensajes en un nivel básico entre el programa maestro y las tareas, haciendo uso de las funciones propias de PVM. El programa maestro enviará un mensaje a la consola indicando que ha iniciado su ejecución, después cada una de las tareas enviará un mensaje al programa maestro, que recogerá y mandará a desplegar.

Programa hola.c

```
/* ****************************************************************************
 El primer ejemplo es por supuesto el programa hola, este programa consiste de dos procesos
hola.c y hola_desde.c. El proceso hola se inicia en el servidor y genera los procesos hola_desde.c
en los nodos, y el proceso hola_desde.c envía un mensaje a hola.c que imprime en pantalla y
sale de PVM.
**************************************************************************** */

#include <stdio.h>
#include <stdlib.h>
#include "pvm3.h"
```

```
main(int argc, char *argv[])
{
 int cc, mytid, msgtag;
 int child[20];
 char buf[100];
 int myparent;
 int ntask=2;
 int i;
 mytid = pvm_mytid();                    /* Obtener el identificador del programa maestro */
 printf("Yo soy el papa t%x\n", mytid);  /* Imprimir el identificador del programa maestro */

 /*Verificacion, si no existo error al generar el programa maestro
  un valor menor que cero (-1), indica que no fue creado */

 if(mytid < 0) {
  pvm_perror(argv[0]);
  pvm_exit();
  exit(0);
 }

 myparent = pvm_parent();                /* Obtener el identificador del programa,
                                            determina si es maestro o esclavo */

 if ( (myparent < 0) && (myparent != PvmNoParent)){
  pvm_perror(argv[0]);
  pvm_exit();
  exit(0);
 }

 /* Verificar que el número de tareas a expandir no sea mayor al número
  de nodos que tenemos en ese momento, tenemos un rango de nodos de 0-20 */

 if ( myparent == PvmNoParent) {
  if(argc == 2) ntask = atoi(argv[1]);
  if((ntask < -1) || (ntask > 20)) {
   pvm_exit();
   exit(0);
  }
  /* Expandir las tareas a cada uno de los nodos */

  cc = pvm_spawn("hola_desde", (char**)0, PvmTaskDefault,(char*)0,
                         ntask, child);

 /* Obtenemos los mensajes de cada una de las tareas */

  for(i=0;i<ntask;i++){
   msgtag = 1;
   pvm_recv(child[i],msgtag);          /* Recibir el mensaje de acuerdo a su identificador */
   pvm_upkstr(buf);                    /* Lo colocamos en el bufer e imprimimos el mensaje */
   printf("desde t%x: %s\n", child[i], buf);   /* Imprimir mensaje en la consola */
  }
 }
}
```

Programa hola_desde.c

```
include "pvm3.h"
#include <string.h>
```

```
main()
{
 int ptid,msgtag;
 char buf[100];

 ptid = pvm_parent();              /* Obtiene su identificador, si este se genero
                                   a partir de un programa principal que realizo
                                   la tarea de spawn, regresa un tid entero y en
                                        caso contrario regresa PvmNoParent */

 strcpy(buf, "hola, mundo desde");  /* Copia el mensaje en el bufer */
 gethostname(buf + strlen(buf),64); /* Adiciona el nombre del host */
 msgtag = 1;                        /* Identificador del mensaje */
 pvm_initsend(PvmDataDefault);      /* Limpia el buffer */
 pvm_pkstr(buf);                    /* Coloca el mensaje en el bufer */
 pvm_send(ptid,msgtag);             /* Enviar el mensaje con su respectivo identificador */
 pvm_exit();                        /* Terminar el programa */

}
```

La figura 4.8 muestra el paso de mensajes de los programas anteriores, entre las maquinas que conforman el cluster, utilizando xpvm.

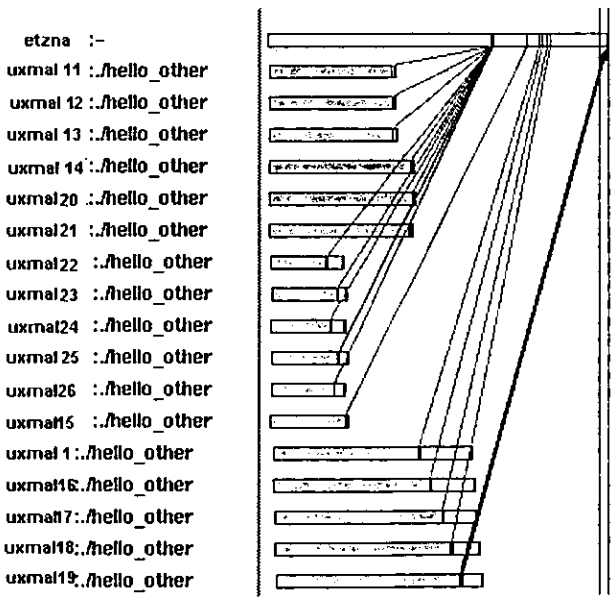


Figura 4. 8 Paso de mensajes en el programa hola.

4.8.2. PROGRAMA FORKJOIN

El ejemplo que a continuación se presenta lo podemos encontrar en el manual de PVM y se consideró importante colocarlo en este trabajo debido a que en él se ejemplifica claramente el paradigma de programación SPMD y ayuda a comprender algunas funciones principales de PVM que son útiles para la realización de este tipo de programas.

Aquí sólo tenemos un programa que es ejecutado primeramente por la maquina servidor, entonces tenemos a la tarea principal llamada padre, quien engendra o crea a las tareas hijos. Existen funciones de PVM que ayudan a obtener el identificador de la tarea y saber si ésta fue generada por alguna otra tarea o si ella es la tarea principal, esto nos ayuda a que al momento de que a cada máquina le toque ejecutar el programa sepa qué parte del código debe utilizar.

```
/*
        EJEMPLO Fork Join
        Demuestra como engendrar (crear) procesos e intercambiar mensajes
*/

/* libreria para definiciones y prototipos de PVM */

#include <pvm3.h>

/* Maximo numero de hijos que este programa engendrara */
#define MAXNHIJOS    20

/*La siguiente etiqueta se usa para el mensaje de union*/
#define JOINTAG   11


int main(int argc, char* argv[])
{
        /* significado de las variables */

/* numero de tareas a engendrar (crear) usando 3 como default */

  int ntask = 3;

/*codigo que retornan las llamadas a pvm*/
  int info;

 /* identificador de la terea ( my task id) */
  int mytid;

 /* identificador de la tarea padre ( my parents task id ) */
  int myparent;

 /* arreglo de identificadores de los hijos */
  int child[MAXNHIJOS];

/* otras variables */
  int i, mydata, buf, len, tag, tid;
```

```
        /*empieza codigo del programa */

 /* encuentra el numero de identificador (id) de mi tarea */

   mytid = pvm_mytid();

 /* verifica si hay un error ,esto es si el resultado de mytid es negativo */

 if (mytid < 0) {

   /* imprime un error que esta contenido en argv[0] */

   pvm_perror(argv[0]);

   /* sale del programa */

   return -1;

       }

 /* encuentra el (numero) identificador de tarea del padre */
   myparent = pvm_parent();

 /* sale si hay algun error ( esto es si el valor de myparent es negativo)descartando a
PVM_Noparent como resultado */

 if ((myparent < 0) && (myparent != PvmNoParent)) {
   pvm_perror(argv[0]);
   pvm_exit();
   return -1;
                                              }

 /* si yo no tengo padre entonces yo soy el padre. Esto es si myparent es igual a PvmNoParent
esto es que esta tarea no tiene padre y es el padre, entonces deberá crear a los hijos */

 if (myparent == PvmNoParent)
{

 /*Se entera y encuentra cuantas tareas hay que engendrar          (spawn) */
     if (argc == 2) ntask = atoi(argv[1]);

/* verifica que el numero de tareas sea legal, si no es legal sale */
     if ((ntask < 1) || (ntask > MAXNHIJOS))
     {
       pvm_exit();
       return 0;

     }

   /* engendra las tareas hijos */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0,
     ntask, child);

   /* imprime los identificadores de las tareas */
   for (i = 0; i < ntask; i++)
```

```
 if (child[i] < 0)
/* imprime el codigo del error en decimal*/
          printf(" %d", child[i]);

        else
        /* imprime el identificador de la tarea en hexadecimal*/
           printf("t%x\t", child[i]);
    putchar('\n');

    /* se asegura que el spawn haya sido exitoso*/
      if (info == 0) {
       pvm_exit();
        return -1;
            }
    /* solo espera respuesta de que el spawnwd sea correcto */
    ntask = info;
    for (i = 0; i < ntask; i++) {

      /* recibe un mensaje de uno de los procesos hijos */
      buf = pvm_recv(-1, JOINTAG);

       /* si buf es negativo imprime un error*/
      if (buf < 0)
        pvm_perror("calling recv");

      info = pvm_bufinfo(buf, &len, &tag, &tid);
       /* si info (información del bufer)es negativo imprime un error */
      if (info < 0)
        pvm_perror("llamando pvm_bufinfo");

       info = pvm_upkint(&mydata, 1, 1);

       if (info < 0)
        pvm_perror("calling pvm_upkint");

       if (mydata != tid)
          printf("This should not happen!\n");

      printf("Length %d, Tag %d, Tid t%x\n", len, tag, tid);
      }
      pvm_exit();
     return 0;

   } /* fin del primer if*/


  /* Soy un HIJO código ejecutado por el proceso hijo*/

  info = pvm_initsend(PvmDataDefault);
     if (info < 0) {
     pvm_perror("llamando pvm_initsend");
     pvm_exit();
     return -1;
           }
  info = pvm_pkint(&mytid, 1, 1);

  if (info < 0) {
   pvm_perror("llamando pvm_pkint");
   pvm_exit();
```

```
    return -1;
        }
    info = pvm_send(myparent, JOINTAG);
    if (info < 0) {
     pvm_perror("calling pvm_send");
     pvm_exit();
     return -1;
        }
     pvm_exit();
     return 0;
}
```

4.8.3. PROGRAMA ANILLO

El siguiente programa muestra el paradigma de programación SPMD, el cual sólo consta de un programa que en un principio ejecutará el servidor y se enviará una copia del mismo a los demás, para que ellos también ejecuten el programa, pero sólo la parte del código que les corresponde como tareas hijas.

Este programa utiliza el concepto de grupos, todas las tareas son inscritas en un grupo y así es más fácil saber quiénes son los vecinos de cada una de las tareas, lo que nos va a permitir pasar el *token* (o estafeta) de tarea en tarea en un orden hasta llegar de nuevo a la tarea que inicio.

```
/*
  Ejemplo paradigma SPMD "anillo"
  Se ilustran también las funciones de grupo
*/


#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include "pvm3.h"

#define NPROC 18  //numero de procesadores

void realizan_tareas();

main()
{
  int mytid;          /* id de tareas */
  int tids[NPROC];    /* arreglo de tids */
  int mi;             /* mi numero de proceso */
  int i;

  /* enrola en pvm, obtiene el tid */
  mytid = pvm_mytid();
```

```
/*****************************************************************
 junta las procesos llamados(tareas)hasta la tarea principal en un
grupo, si es la tarea principal mi =0 genera mas copias de el mismo
*****************************************************************/

 /* mi obtiene el tid de la tarea en el grupo */

  mi = pvm_joingroup( "grupo" );
  printf("\n mi = %d mytid = %d\n",mi,mytid);


  if( mi == 0 )
    pvm_spawn("./spmd", (char**)0, 0, "", NPROC-1, &tids[1]);

   /* espera a todos antes de seguir adelante*/

   pvm_freezegroup("grupo", NPROC);
   pvm_barrier("grupo", NPROC );
/*-----------------------------------------------------------------*/
   realizan_tareas(mi,NPROC );

   /* programa termidado todos dejan el grupo */
   pvm_lvgroup( "grupo" );
   pvm_exit();
   exit(1);
}
/*Ejemplo que muestra el paso de un token a traves de un anillo*/

void realizan_tareas(int mi,int nproc )
{
   int token;
   int fuente, destino;
   int msgtag = 4;

/* determina los vecinos en el anillo*/

   fuente = pvm_gettid("grupo", mi-1);

   destino= pvm_gettid("grupo", mi+1);

/*obtiene el tid del numero de proceso indicado en el grupo */
   if( mi == 0 )
   fuente = pvm_gettid("grupo", NPROC-1);

/*obtiene el tid del numero de proceso indicado en el grupo */
   if( mi == NPROC-1 )
   destino = pvm_gettid("grupo", 0);

   if( mi == 0 )
   {
    token = destino;
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&token,1,1);
    pvm_send(destino,msgtag );
    pvm_recv(fuente,msgtag);
    printf(" \n token recorrio el anillo\n");

   }
   else
```

```
  {

   pvm_recv(fuente,msgtag);
   pvm_upkint(&token,1,1);
   pvm_initsend(PvmDataDefault);
   pvm_pkint(&token,1,1 );
   pvm_send(destino,msgtag);

  }
}
```

La siguiente figura muestra el paso de mensajes entre las máquinas, y cómo la última tarea regresa, a la tarea principal para cerrar el anillo.

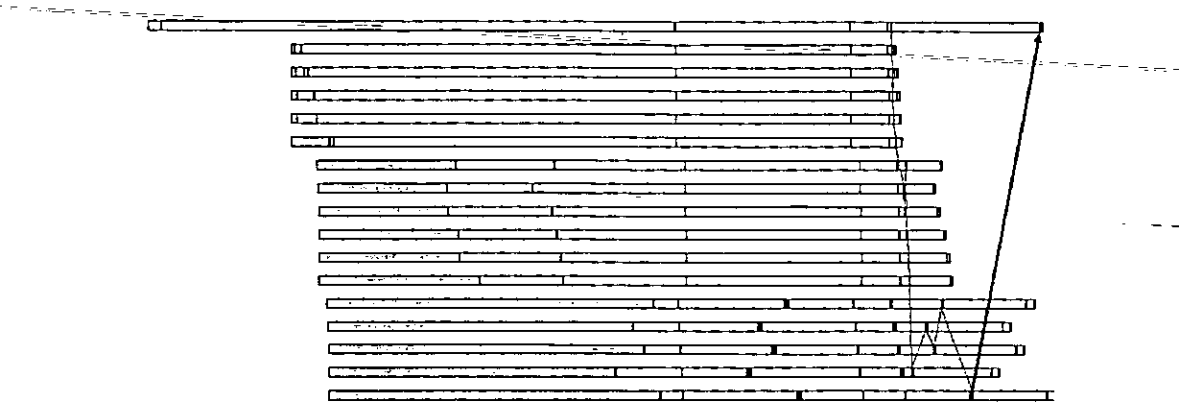


Figura 4. 9 Paso de mensajes en el programa anillo.

4.8.4. ARBOL

El siguiente código ejemplifica el recorrido de un árbol con una sola hoja. La máquina que ejecute por primera vez el programa será el nodo principal, éste a su vez generará al siguiente subnodo. El subnodo verifica la profundidad del árbol si todavía no es igual a la especificada, genera otro subnodo y así sucesivamente hasta llegar a la hoja, y hacer lo mismo de regreso hasta llegar al nodo padre.

```
/*
**************************************************************************
programa arbol.c

Este es un programa que recorre un arbol de ida y regreso
USANDO PVM
**************************************************************************
*/

#include <stdio.h>
#include "pvm3.h"
#define SAL 99
#define REGRESO 666
#define MAXNODOS 4 // Maximo de nodos encadenados
#define MAXPROCESOS 1       // Maximo numero de procesos por procesador
main()

{
 int nproc,              /* Numero de procesos*/
   numtareas,  /* Numero de tareas*/
   quien,              /* Quien Manda el mensaje */
   ultimo,             /* ultima hoja en el arbol*/
   profindidad, /* profundidad del arbol*/
   cont,         /* Contador para no nodo */
   padre,              /* Nodo padre*/
   mytid;        /* ID de la tarea*/

 FILE *fp;     /* archivo para guardar datos */
 char s[80];    /* para guardar el nombre del archivo*/

 int tids[MAXPROCESOS];      /* arreglo de tareas*/

 struct pvmhostinfo *hostp[MAXPROCESOS]; /* variables de ambiente*/

 nproc = MAXPROCESOS;

 cont = 0;

 /* enrrola la tarea en pvm, obtiene su identificador*/

 mytid = pvm_mytid();        /* obtengo mi tid */

 padre = pvm_parent();       /* obtengo el tid del padre */

 printf("No tiene padre = %d \n", PvmNoParent);
```

```
/*****************************************************************
        si no tiene padre el es el padre
        Es la primera llamada

*****************************************************************/

 if (padre == PvmNoParent)
 {

  printf("Introducir la profundidad del arbol, menor o igual a 4\n");
  scanf ("%d", &profundidad);

  if (profundidad > MAXNODOS) profundidad = MAXNODOS;

  printf("Empezando Primera Tarea.\n");

  /* Creando la tarea para hacer un nodo, se crea solo una tarea*/
  numtareas = pvm_spawn("./arbol", (char**)0, 0, "", nproc, tids);


  sprintf(s,"file.%dA",mytid);
  fp = fopen(s, "w");

  fprintf(fp, "Mi identificador: %d  Padre: %d \n", mytid, padre);
  fclose(fp);

       /* ¿Quien es el primer Nodo?*/

  quien = tids[cont];
  printf("Mandando Dato: %d %d \n", cont, quien);
       /* incrementamos el contador
  cont++;

  /* Manda mensaje de SALIDA (SAL) al primer nodo*/

  pvm_initsend(PvmDataDefault);      // Inicia Bufer
  pvm_pkint(&cont,1,1);        // Enpaqueta el contador actual
  pvm_pkint(&profundidad,1,1);              // Empaqueta la profundidad del arbol
  pvm_pkint(&mytid,1,1);             // Empaqueta mi identificador de tarea
  pvm_send(quien, SAL);              // manda mensaje

  /* Espera resultados de las tareas generadas en el arbol */
  printf("Esperando a recibir\n");
  pvm_recv( -1, REGRESO);            // Espera hasta recibir
  pvm_upkint( &cont, 1, 1 );   // obtiene contador final
  pvm_upkint( &ultimo, 1, 1); // anterior nodo en el arbol
  pvm_upkint( &quien, 1, 1);  //Quien mando el mensaje

  printf("Cont: %d Ultimo: %d  Quien: %d\n", cont, ultimo, quien);
 }/* fin if*/


/*****************************************************************************
**********
Esta parte es para un Subnodo
*****************************************************************************
**********/
else
 {
```

```
    /*Recive Mensaje que esta siendo mandado (SAL) desde el nodo padre*/
pvm_recv( -1, SAL );                    // Espera mensaje
pvm_upkint( &cont, 1, 1 );  // Obtiene actual contador
pvm_upkint(&depth, 1, 1 ); // Obtiene la maxima profundidad
pvm_upkint(&quien, 1, 1);  //Quien mando el mensaje

sprintf(s,"archivo.%dB",mytid);

fp = fopen(s, "w");
fprintf(fp, "Mi identificador de tarea: %d Contador: %d Profundidad: %d Quien: %d \n",
mytid, cont, depth,quien);

fclose(fp);
    /* ****************************************
    ¿Hemos alcanzado el ultimo nodo?
    ****************************************/

     /* Si no, se hacen nuevos nodos*/
if (cont < profundidad)     {
    numtareas = pvm_spawn("./arbol", (char**)0, 0, "", nproc, tids);
   cont++;

 /* Enviando mensaje de salida(SAL)al nodo hijo*/
 pvm_initsend(PvmDataDefault);
 pvm_pkint(&cont, 1, 1);    /* Empaqueta actual contador*/
 pvm_pkint(&depth, 1, 1);  /* Empaqueta profundidad actualdel arbol*/
 pvm_pkint(&mytid, 1, 1);  /* Empaqueto mi tid*/
 pvm_send(tids[0], SAL);   /* Mando mensaje */


          /* Espero a recibir mensaje (REGRESO) del nodo hijo*/
 pvm_recv( -1, REGRESO );          /* Espero mensaje*/
          /* obtengo los datos enviados del anterior nodo */
 pvm_upkint( &cont, 1, 1 ); /* Obtengo Contador*/
 pvm_upkint( &ultimo, 1, 1);        /*Obtengo anterior nodo en el arbol*/
 pvm_upkint( &quien, 1, 1 );        /* obtengo el id de quien mando */
          /* imprimo los datos en un archivo */
 sprintf(s,"archivo.%dC",mytid);
 fp = fopen(s, "w");
 fprintf(fp, "Mi identificador es: %d  Contador: %d  Quien %d: \n",
               mytid, cont, quien);
 fclose(fp);

    /* Manda Mensaje del tipo REGRESO al nodo padre el subnodo que lo creo*/
 pvm_initsend(PvmDataDefault);
 pvm_pkint(&cont, 1, 1);    /*Empaqueta el actual contador*/
 pvm_pkint(&ultimo, 1, 1);
 pvm_pkint(&mytid, 1, 1);  /* empaqueto mi identificador*/
 pvm_send(padre, REGRESO);       /* Se manda el mensaje*/
 }
    /*************************************************************************
*********
     Se llego a ser hoja, Ahora envio con mensaje REGRESO

     *************************************************************************
**********/
 else
  {
```

```
    sprintf(s,"archivo.%dD",mytid);
    fp = fopen(s, "w");
    fprintf(fp, "Ultimo!!! Mi identificador es: %d  Contador: %d  Quien%d: \n",mytid, cont,
quien);
    fclose(fp);

    /*************************************************************
   Envio con mensaje de REGRESO
        ************************************************************/
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&cont, 1, 1);  /* empaqueta Cuenta actual */
    pvm_pkint(&mytid, 1, 1);  /* Yo soy el ultimo nodo */
    pvm_pkint(&mytid, 1, 1);  /*empaqueta mi identificador de tarea*/
    pvm_send(padre, REGRESO); /* El envio va de regreso al padre*/
    }
  }

 /* El programa Termina*/
  pvm_exit();
}
```

La siguiente figura muestra el recorrido del árbol.

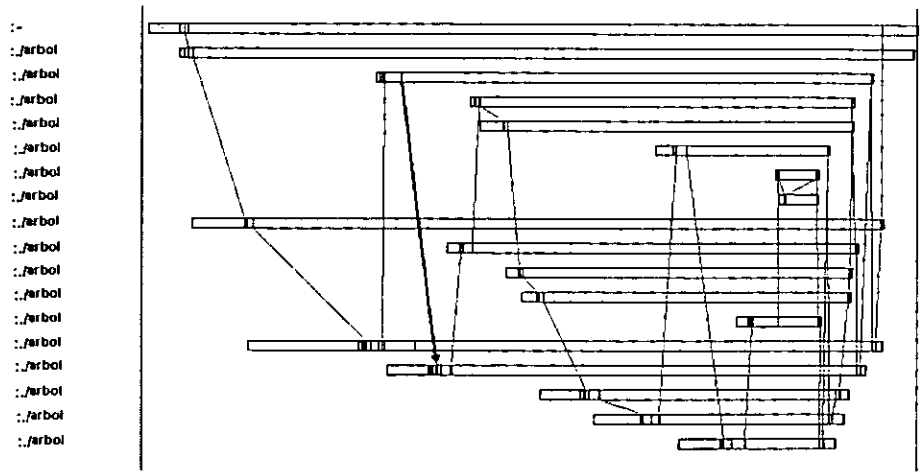


Figura 4. 10 Paso de mensajes en el programa del árbol.

4.8.5. CÁLCULO DE PI

Para desarrollar el cálculo del valor de π se recurre a la aproximación por medio de la integración numérica de la función $\frac{4}{(1+x^2)}$, entre los valores límites de 0 y 1. Esta aproximación se da al calcular el área de los rectángulos que de altura $\frac{4}{\left(1+\left(\frac{(k-0.5)}{N}\right)\times 2\right)}$ y anchura $w=\frac{1}{N}$, donde k es el índice que se va recorriendo desde 1 hasta N. N representa el número de subintervalos (rectángulos) empleados para calcular la integral.[3]

Para esto, el intervalo $[\,0,1\,]$ se divide en n subintervalos de longitud $\frac{1}{N}$.

Como la integral es una sumatoria, el sumar primero una parte y después agregarle otra suma realizada con datos de la curva una parte más adelante o más atrás, no afecta al resultado final. En esta parte es en donde se encuentra concurrencia dentro de este algoritmo. La idea que se tiene al desarrollar este programa es que la integral (la suma de áreas bajo la curva), se reparta de manera equitativa entre todos los procesadores, de esta manera cada uno de ellos realiza la suma de una región de tamaño pequeño, y el programa, en su parte de maestro, es quien se encarga de reunir el resultado y realizar la última suma, que corresponde al resultado final del proceso.

Para elaborar este programa se crea un código bajo el paradigma SPMD. Con este tipo de programación se crea un solo programa que contiene la información necesaria para el maestro y para el esclavo.

Dentro del presente programa, se utilizan 3 funciones para el manejo de los datos:

- Inicializar: Esta rutina se encarga de crear los procesos paralelos.

En la rutina de inicializar el maestro se identifica dentro de PVM. Una vez logrado esto redirecciona la salida de los procesos esclavos hacia el mismo. A continuación empieza la generación de los procesos esclavos. Una vez que los ha generado, empieza la distribución de los primeros datos para los esclavos, que son el número de tareas y los identificadores de todas las tareas esclavas.

Si al momento de identificarse dentro de PVM la tarea resulta ser un esclavo, entonces desempaqueta los datos recibidos e indica en què posición del arreglo de TIDs esta localizado el suyo.

- Solicitar: Pide el valor de N y lo distribuye a todos los procesos.

[3] Apuntes de Métodos Numéricos
Iriarte, Rafael
Editorial División de Ciencias Básicas

Igual que en la rutina anterior hay código que deberá seguir el maestro y código que corresponde al esclavo.

El maestro pide el número de subintervalos para calcular la integral. Una vez que tiene el valor de N lo manda a todos los esclavos.

En la parte de los esclavos, sólo desempaquetan la información proporcionada.

- Recoger: Recibe e imprime el resultado final.

En esta parte el maestro recibe la suma de cada esclavo y guarda e imprime el total.

Los esclavos mandan su suma al maestro.

Dentro de la rutina principal del programa se encuentran las operaciones que realiza cada nodo para obtener la suma que se le pide.

A continuación se muestra el listado del programa.

```
/*
 * Ejemplo de Paralelización usando PVM
 * pi_pvm.c */

/* librerias */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "pvm3.h"

#define f(x) ((double)(4.0/(1.0+x*x)))
#define pi ((double)(4.0*atan(1.0)))
#define MAXTAREAS 32

/* prototipos de funciones */
void inicializar (int *pmytid, int *pparent_tid, int *pntareas, int *indice, int tids[MAXTAREAS]);

double solicitar (int parent_tid, int ntareas, int tids[MAXTAREAS]);

void recoger(double sum, int parent_tid, int ntareas);

int main()
 {
 /* Este programa aproxima pi calculando pi = integral
  * de 0 a 1 de 4/(1+x*x)dx que a su vez se calcula por la suma
  * desde k=1 a N de los rectángulos de altura 4 / (1+((k-.5)/N)**2).
  * y anchura w=1.0/N.
  * El único parámetro de entrada es N y representa el numero
  * de subintervalos (rectángulos) empleados para calcular la integral
  */

 double sum, w;
 int i,rc;
 double N;
```

```
int mytid;          /* ID de tarea */
int parent_tid;       /* ID de tarea de la tarea padre */
int ntareas;        /* numero de tareas adicionales iniciadas */
int indice;         /* indice de la tarea en el arreglo de ID */
int tids[MAXTAREAS];     /* identificadores de las tareas adicionales */

 /*
 * La rutina inicializar se encarga de crear los procesos paralelos
  */

 inicializar(&mytid, &parent_tid, &ntareas, &indice, tids);

 /*
  * La rutina solicitar pide el valor de N y lo propaga a los proceso en la versión paralelo
  */

 N = solicitar(parent_tid, ntareas, tids);

  w = 1.0/(double)N;
  sum = 0.0;
  for (i = indice+1; i <= N; i+=ntareas+1)
   sum = sum + f(((double)i-0.5)*w);
  sum = sum * w;


  /*
   * La rutina recoger recibe e imprime el resultado
   */

  recoger(sum, parent_tid, ntareas);

 /*
  * Liberar enlace con PVM antes de salir del programa principal
  */

 rc = pvm_exit ();
 return (0);
 }/* fin del main*/

 /* ------------------------------------------------------------ */
 void inicializar (int *pmytid, int *pparent_tid, int *pntareas, int *pindice, int tids[MAXTAREAS])
 {
  int mytid, parent_tid, ntareas, indice, rc, i;

  /*
   * Obtiene su ID de tarea con pvm_mytid()
   */

  mytid = pvm_mytid();
  *pmytid = mytid;

  /*
   * Obtiene el identificador ID de la tarea padre. PvmNoParent indicara la tarea maestro
   */

  parent_tid = pvm_parent();
  *pparent_tid = parent_tid;

  /*
```

```
 * El maestro redirecciona la salida standard de los otros procesos
 */

if (parent_tid == PvmNoParent) {
 rc = pvm_catchout(stdout);

 /*
  * ... y con spawn inicia la ejecución de los procesos
  */


/* Numero de tareas a ejecutar*/
ntareas=2;

 while ((ntareas > MAXTAREAS)||(ntareas <= 0)) {
  printf("El numero de esclavos debe estar entre 1 y %2d...", MAXTAREAS);
  printf("Prueba otro numero!\n");
  scanf("%d", &ntareas);
 }
 *pntareas = ntareas;
 rc = pvm_spawn("./pi", NULL, PvmTaskDefault, NULL, ntareas, tids);
 if (rc != ntareas) {
  printf ("Error en spawn, solo %d\n tareas se han iconseguido levantar", rc);
  exit(0);
 }

 /*
  * ... asignarle su indice
  */

 indice = 0;
 *pindice = indice;

 /*
  * ... y enviar con multicasts el numero de tareas y los ID de los tareas esclavos
  */

 rc = pvm_initsend(PvmDataDefault);
 rc = pvm_pkint(&ntareas,1,1);
 rc = pvm_pkint(tids,ntareas,1);
 rc = pvm_mcast(tids, ntareas, 222);
}

/*
 * Los esclavos reciben y desempaquetan el mensaje enviados con multicast
 */

else {
 rc = pvm_recv(parent_tid, 222);
 rc = pvm_upkint(&ntareas, 1, 1);
 *pntareas = ntareas;
 rc = pvm_upkint(tids, ntareas, 1);

 /*
  * ... y pone su indice en el array de identifcadores de tarea
  */

 for (i = 0; i < ntareas; i++)
  if (mytid == tids[i]) indice=i+1;
```

```
  *pindice = indice;
 }
}


/* ---------------------------------------------------------- */
double solicitar (int parent_tid, int ntareas, int tids[MAXTAREAS])
{
 double N;
 int rc;

 /*
  * Solo el maestro pide información de entrada al usuario
  */

 if (parent_tid == PvmNoParent) {
N=pow(2,20)-1;
printf("\n mi valor es: %f \n",N);

  /*
   * ... y con un multicasts comunica esta entrada al resto de procesos
   */

  rc = pvm_initsend(PvmDataDefault);
  rc = pvm_pkdouble(&N,1,1);
  rc = pvm_mcast(tids, ntareas, 111);
 }

 /*
  * El proceso esclavo recibe el mensaje del maestro
  */

 else {
  rc = pvm_recv(parent_tid, 111);
  rc = pvm_upkdouble(&N, 1, 1);
 }
 return (N);
}
/* ---------------------------------------------------------- */
void recoger(double sum, int parent_tid, int ntareas)
{
 double err, x;
 int i, rc;

 /*
  * El maestro recibe la suma de cada esclavo y guarda el total
  */

 if (parent_tid == PvmNoParent) {
  for (i=0; i<ntareas; i++) {
   rc = pvm_recv(-1, 333);
   rc = pvm_upkdouble(&x, 1, 1);
   sum=sum+x;
  }
  /*
   * ... calcula una estimación del error cometido e imprime el resultado
   */
  err = sum - pi;
  printf("pi = ,  error = %7.5f, %10e\n", sum, err);
```

```
}
/*
 * Los esclavos mandan su suma al maestro
 */
else {
 rc = pvm_initsend(PvmDataDefault);
 rc = pvm_pkdouble(&sum, 1, 1);
 rc = pvm_send(parent_tid, 333);
}
}
```

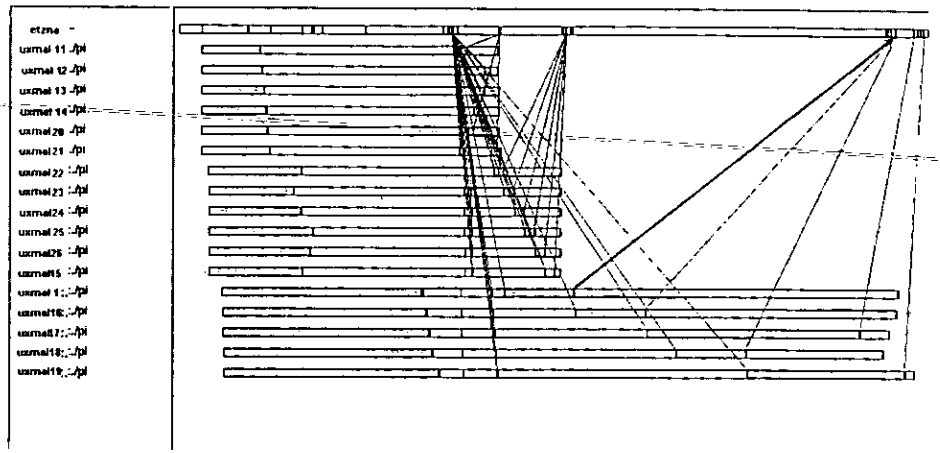


Figura 4. 11 Paso de mensajes del programa del cálculo de pi.

4.8.6. MULTIPLICACIÓN DE MATRICES

Las operaciones con matrices, son utilizadas muy a menudo para organizar o representar cierta información y poder manipularla mejor. Es por eso que consideramos importante un ejemplo con matrices, en este caso la multiplicación.

Vamos a mostrar primeramente en qué consiste el algoritmo para esta operación.

Si tenemos a las matrices A y B.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \ldots\ldots & a_{1m} \\ a_{21} & a_{22} & a_{23} \ldots\ldots & a_{2m} \\ a_{31} & a_{32} & a_{33} \ldots\ldots & a_{3m} \\ a_{n1} & a_{n2} & a_{n3} \ldots\ldots & a_{nm} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \ldots\ldots & b_{1h} \\ b_{21} & b_{22} & b_{23} \ldots\ldots & b_{2h} \\ b_{31} & b_{32} & b_{33} \ldots\ldots & b_{3h} \\ b_{m1} & b_{m2} & b_{m3} \ldots\ldots & b_{mh} \end{bmatrix}$$

La multiplicación se lleva a cabo renglón columna, como sigue:

$$C = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + \ldots + a_{1m}b_{m1} & a_{11}b_{12} + a_{12}b_{22} + \ldots + a_{1m}b_{m2} & \ldots. & a_{11}b_{1h} + a_{12}b_{2h} + \ldots + a_{1m}b_{mh} \\ a_{21}b_{11} + a_{22}b_{21} + \ldots + a_{2m}b_{m1} & a_{21}b_{12} + a_{22}b_{22} + \ldots + a_{2m}b_{m2} & \ldots. & \\ & & & \\ a_{31}b_{11} + a_{32}b_{21} + \ldots + a_{3m}b_{m1} & a_{31}b_{12} + a_{32}b_{22} + \ldots + a_{3m}b_{m2} & \ldots. & \\ & & & \\ a_{n1}b_{11} + a_{n2}b_{21} + \ldots + a_{nm}b_{m1} & a_{n1}b_{12} + a_{n2}b_{22} + \ldots + a_{nm}b_{m2} & \ldots. & a_{n1}b_{1h} + a_{n2}b_{2h} + \ldots + a_{nm}b_{mh} \end{bmatrix}$$

$$c[n][h] = \sum_{k=1}^{m} a_{nk}\, b_{kh}$$

Ahora que se conoce el algoritmo, de cierta manera el secuencial, hay que analizar de qué forma podemos hacerlo concurrente.

Como podemos darnos cuenta sólo existe una operación, aplicada a diferentes datos de la matriz. Entonces cada nodo puede realizar la misma operación sobre diferentes datos y lo que hay que analizar es cómo dividir los datos de manera que a cada nodo del cluster se le asignen datos diferentes.

Tenemos dos opciones:

1. Para obtener un elemento de la matriz resultante necesitamos un renglón de la matriz A y toda una columna de la matriz B.
2. Para obtener todo un renglón de la matriz resultante necesitamos un renglón de la matriz A y toda la matriz B.

Nosotros optamos por la segunda opción debido a que así la división de los datos entre los nodos sólo dependerá de la cantidad de renglones de la matriz A.

Entonces podemos asignar a cada nodo una cierta cantidad de renglones de A y toda la matriz B. El numero de renglones de A asignados a cada nodo lo obtenemos de la siguiente razón.

$$\text{renglones asignados} = \frac{\text{número de renglones de } A}{\text{número de nodos del cluster}}$$

¿Pero que pasa si esta razón no es entera?.Entonces no será posible asignar a cada nodo un número igual de renglones de A.

Para ello tenemos dos soluciones

1. Asignar los renglones sobrante a algún nodo
2. Distribuir uniformemente los renglones sobrantes entre los nodos

A nuestro parecer es mejor la segunda alternativa, para que sea más equitativo el trabajo y no se llegue a sobrecargar un solo nodo, ya que esto podría alentar el procesamiento.

Ya tenemos una opción para repartir el trabajo, y ahora , ¿Cómo se llevará acabo el cómputo?

Primero necesitamos que la máquina servidor reparta el trabajo a los nodos y ellos realicen los cálculos a los datos asignados y regresen el resultado correspondiente al servidor para que él organice los datos y muestre la matriz resultante.

En conjunto con lo mencionado anteriormente podemos resumir a grandes rasgos los pasos que debemos tomar en cuenta en la programación, en los siguientes:

- Definir matriz A y B.
- Obtener cuántos renglones de A serán enviados a cada nodo.
- Enviar datos correspondientes a cada nodo (renglones asignados de la matriz A y toda la matriz B), y esperar resultados.
- Cada nodo realiza sus operaciones correspondientes.
- Cada nodo envía su resultado al servidor.
- El servidor recolecta los resultados y los muestra ya organizados.

La programación de lo anterior puede ser realizada de diferentes maneras dependiendo de las ideas del programador.

Aquí presentaremos dos soluciones de programación utilizando el lenguaje C y PVM.

La primera es utilizando el paradigma de programación maestro-esclavo:

```
/*
*****************************************************************************
PVM Multiplicacion de Matrices A y B
Programa Maestro pvm_mm.maestro.c

En este codigo el programa maestro actua como el padre que crea
(spawned) (NPROC numero de procesos) las tareas que van ha realizar
los trabajadores

El programa maestro desarrolla la multiplicación de matrices
mandando toda la matriz B a cada tarea trabajador así como algunos
renglones de la matriz A , que corresponden a la cantidad total de
renglones dividida entre el numero de procesos y agregando los
adicionales a los primeros. Cada trabajador (tarea) realiza la
multiplicacion correspondiente a los renglones de la matriz A que le
tocaron y manda de regreso al maestro su respectivo resultado

*****************************************************************************
*/
```

```
/* librerias */
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include "pvm3.h"                /* incluye PVM versión 3.0 */

#define NPROC 2                  /* numero de procesos o tareas trabajadores a crear
(spawned) */
#define NRA 180                  /* numero de renglones en la matriz A */
#define NCA 20                   /* numero de columnas en la matriz A*/
#define NCB 20                   /* numero de columnas en la matriz B*/

main()
{
int maestro_tid;                 /* identificadir de la tarea maestro */
int trabajador_tids[NPROC];      /* arreglo de identificadores de trabajadores */
int tipo_mensaje;                /* tipo mensaje PVM */
int renglones;                   /* numero renglones de la matriz A a enviar a cada trabajador */
int div_renglones,extra,offset;  /* usados para determinar los renglones mandados a cada
trabajador */
int res,codigo_resultado,i=0,j;  /* otras variables */
int a[NRA][NCA];                 /* matriz A a multiplicar */
int b[NCA][NCB];                 /* matriz B a multiplicar*/
double c[NRA][NCB];              /* matriz C de resultado */
char este_host[35];              /* nombre del host seleccionado */

/* enrrola esta tarea en PVM, obtenemos tid maestro*/

maestro_tid = pvm_mytid();


/* La tarea maestro crea las tareas trabajadores llamando a pvm_spawn
Los identificadores unicos de las tareas son almacenados en el arreglo
trabajador_tids.
La primera tarea trabajador ( o proceso) es creada en una maquina especifica.
El codigo retornado nos dice el numero de tarea que se ha creado
satisfactoriamente*/

for(i=0;i<NPROC;i++) {

  res=pvm_spawn("./pvm_mm.esclavo", NULL, PvmTaskDefault,"", 1, &trabajador_tids[i]);
              /* asegura que el proceso sea creado exitosamente*/
        if (res != 1)
                {
                pvm_exit();
                return -1;
      } /* if*/

                  } /*for */

/* inicializa A y B */

for(i=0; i < NRA; i++){
  for (j=0; j < NCA; j++){
    a[i][j]= i+j;
                        }
             }
```

```
for (i=0; i<NCA; i++){
  for (j=0; j<NCB; j++){
    b[i][j]= i*j;
                    }
                }

/* obtenemos la razon de la cantidad de renglones a enviar */
  div_renglones = NRA/NPROC;

/* obtenemos los renglones sobrantes */
  extra = NRA%NPROC;

 offset = 0;

 tipo_mensaje = 1;
 /* manda los datos a las tareas trabajadores */

/* para cada una de las tareas */

for(i=0;i<NPROC;i++)
              {

  /*decision de numero de renglones a mandar de la matriz A */

  /* si i < extra renglones = div_renglones+1 en caso contrario
  renglones = div_renglones */

  renglones = (i < extra) ? div_renglones+1 : div_renglones;

  /* La proxima llamada inicializa el bufer de envio y especifica que el
   formato de datos a enviar sera el default XDR */

  /* significa que la conversion es solo en ambientes heterogeneos */
  codigo_resultado= pvm_initsend(PvmDataDefault);


  /* las 4 llamadas siguientes empaquetan los valores dentro el
   buffer de envio */

  /* empieza posicion en la matriz */
  codigo_resultado=pvm_pkint(&offset,1,1);

  /* numero de renglones de la matriz A a enviar */
  codigo_resultado=pvm_pkint(&renglones,1,1);


  /* algunos renglones de A, esta línea nos dice que la cantidad de
  elementos a enviar es igual a renglones*NCA, y los datos a enviar
  empiezan a partir de la posición a[offset][0] */

  codigo_resultado=pvm_pkint(&a[offset][0],renglones*NCA,1);

  /* se empaqueta toda la matriz B */
   codigo_resultado=pvm_pkint(*b,NCA*NCB,1);

  /* manda el contenido del bufer de envia a la tarea trabajador */
  codigo_resultado=pvm_send(trabajador_tids[i],tipo_mensaje);
```

```
/* desplazamiento de renglones en la matriz A*/
  offset = offset + renglones;

                }/*fin for*/

/*****************************************************************************/
/* espera resultados de todas las tareas trabajadores */

        /* pone tipo de mensaje, ahora ya es de recibir */
        tipo_mensaje = 2;

        for(i=0;i<NPROC;i++) {
         /* lo realiza para cada uno de los trabajadores */

        /*recive los mensajes del trabajador*/
        codigo_resultado=pvm_recv(-1,tipo_mensaje);

        /* empieza posicion en la matriz */
        codigo_resultado=pvm_upkint(&offset,1,1);

        /* numero de renglones enviados */
        codigo_resultado=pvm_upkint(&renglones,1,1);

         /*renglones para matriz C */
        codigo_resultado=pvm_upkdouble(&c[offset][0],renglones*NCB,1);

                        }/* fin for*/

/*****************************************************************************/

/* imprime resultados */

        for(i=0;i<NRA;i++) {
          printf("\n");
          for(j=0;j<NCB;j++)
           printf(" %6.2f  ",c[i][j]);

            }
        printf("\n");

        /* tareas que existen en PVM */
        codigo_resultado=pvm_exit();

} /* fin main*/
```

```
/**************************************************************************
**
* PVM Multiplicacion de matrices - Version en C
* Programa trabajador pvm_mm.esclavo.c

llamado por pvm_mm.maestro.c

**************************************************************************/
#include <stdio.h>
#include <malloc.h>
#include "pvm3.h"   /* Cabecera de PVM versión 3.0 */

#define NRA 180     /* Numero de renglones en la matriz A */
#define NCA 20      /* Numero de columnas en la Matriz A */
#define NCB 20      /* Numero de columnas en la Matriz B */

main() {

int  trabajador_tid; /* identificador de tarea de PVM de
                        este programa trabajador */
int  maestro_tid;    /* identificador de tarea de PVM
                        del proceso padre del programa maestro */
int  tipo_mensaje;   /* Tipo de mensaje PVM */
int  renglones;    /* Numero de renglones en la Matriz a enviados al trabajador */
int  offset;       /* Posición inicial en la matriz */
int  codigo_resultado, i, j, k;   /* varios */
int a[NRA][NCA];    /* Matriz A a ser multiplicada */
int b[NCA][NCB];    /* Matriz B a ser multiplicada */
double c[NRA][NCB];   /* Matriz C resultado */

/* incluye en PVM la tarea lenvantada por el trabajador,obtenemos el tid */
trabajador_tid = pvm_mytid();

/* Recibe el mensaje desde el maestro */

  /* cambia el tipo de mensaje */
  tipo_mensaje = 1;

  /* obtiene el identificador de tarea del proceso maestro */
  maestro_tid = pvm_parent();

  /* espera a recibir un mensaje desde el maestro*/
  codigo_resultado = pvm_recv(maestro_tid, tipo_mensaje);

  /* Posicion inicial en las matrices A y C*/
  codigo_resultado = pvm_upkint(&offset, 1, 1);

  /* numero de renglones de la Matriz A enviados */
  codigo_resultado = pvm_upkint(&renglones, 1, 1);

  /* nuestra parte compartida de la Matriz A */
  codigo_resultado = pvm_upkint(*a, renglones*NCA, 1);

  /* contenido Matriz B*/
  codigo_resultado = pvm_upkint(*b, NCA*NCB, 1);

  printf("trabajador: identificador de tarea = %d recibo %d renglones de
A\n", trabajador_tid, renglones);
```

```
/* hace la multiplicacion de las matrices */

        for (k=0; k<NCB; k++)
         for (i=0; i<renglones; i++) {
          c[i][k] = 0.0;
          for (j=0; j<NCA; j++)
           c[i][k] = c[i][k] + a[i][j] * b[j][k];
         }

/* Arma el mensaje a enviar al proceso maestro */

  /* cambia el tipo de mensaje */
        tipo_mensaje = 2;

  /* inicializa el buffer de envio */
        codigo_resultado = pvm_initsend(PvmDataDefault);

   /* posicion del resultado enviado en la matriz resultado */
    codigo_resultado = pvm_pkint(&offset, 1, 1);

   /* Numero de renglones que se envian */
    codigo_resultado = pvm_pkint(&renglones, 1, 1);

   /*muestra parte del resultado en la Matriz C */
   codigo_resultado = pvm_pkdouble(*c, renglones*NCB, 1);

   /* enviando al maestro */
   codigo_resultado = pvm_send(maestro_tid, tipo_mensaje);

   /* saliendo de PVM */
   codigo_resultado = pvm_exit();
} /* fin main */
```

La siguiente figura muestra cómo se comporta la ejecución de los programas anteriores, en todas la máquinas del Cluster

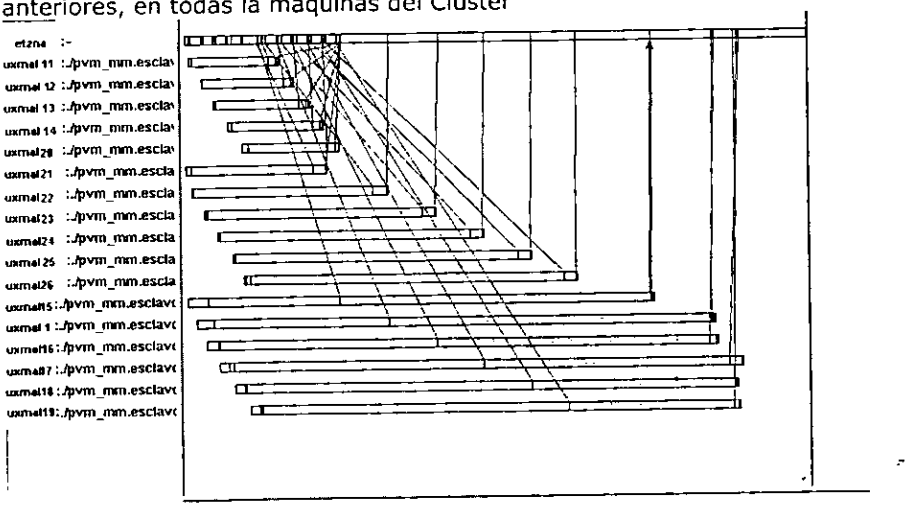


Figura 4. 12 Paso de mensajes en el programa de multiplicación de matrices (Paradigma maestro – esclavo).

La siguiente solución se implementó basándonos en el paradigma de programación SPMD, dónde soló vamos a tener un programa el cual ejecutará primeramente el servidor conteniendo él la tarea principal, esta tarea principal generará tareas esclavas las cuales serán asignadas a los nodos. En este programa usamos la sincronización de las tareas, juntándolas todas hasta la tarea principal en un grupo, lo que va a permitir que todas al mismo tiempo empiecen cierto proceso.

```
/**************************************************************************
Programa mul_mat1.c
Este programa se basa en el paradigma de programación SPMD
Tenemos a la tarea padre que genera NPROC tareas esclavas

El siguiente programa se basa en paradigma de programación SPMD, en el
que una tarea principal genera NPROC tareas esclavas. Todas las tareas
incluyendo la principal van pertenecer a un grupo común.
La tarea principal envía a todas las demás tareas toda la matriz B y
parte de la matriz A. Cada tarea desarrolla la operación de multiplicación
con sus respectivos datos, para que la tarea principal (padre) recolecte
los resultados.


**************************************************************************
*/

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include "pvm3.h"

#define NPROC 2 /* numero de tareas esclavas a generar */
#define NRA 5  /* numero de renglones de la matriz A ,multiple de NPROC
*/
#define NCA 5 /* numero de columnas de la matriz A */
#define NCB 5 /* numero de columnas de la matriz B */

#define RENGLONES (NRA/NPROC) /* NRA/NPROC numero de renglones que a cada
                                 tarea se le asignara */

#define BTIPO 17       /* Broadcast,tipo mensaje */


void main() {
 double A[NRA][NCA];           /* matriz A a multiplicar */
 double B[NCA][NCB];           /* matriz B a multiplicar */
 double C[NRA][NCB];           /*matriz resultante C */
 double ren_ent[RENGLONES][NCA];  /* parte de renglones de A que serán esparcidas */
 double ren_sal[RENGLONES][NCB]; /* renglones resultantes que van a ser enviados a la tarea
                                                              principal */

 int mtid;                     /* identificador de la tarea PVM maestro */
 int tids[NPROC];              /* arreglo de identificadores de tareas PVM
                               de los esclavos (workers)*/

 int mtipo;                    /* tipo mensaje */
 int parent_tid;         /* id tarea padre */
 int inum;                     /* identificador dentro del grupo */
```

```
int rcode, i, j, k;                /* misc */


printf("Matriz A %d x %d \n",NRA,NCA);
printf("Matriz B %d x %d \n",NCA,NCB);
printf("NPROC %d \n",NPROC);


/* enrola esta tarea en PVM, obtiene el identificador de la tarea */
mtid = pvm_mytid();

/*************************************************************************
 En la siguiente parte del programa se utiliza la funcion pvm_parent,
para saber si el proceso actual tiene padre o si el es el padre (proceso
original) quien creara (spawn) a todos los demas procesos esclavos
*************************************************************************/

 parent_tid = pvm_parent();

 if ((parent_tid == PvmNoParent) && (NPROC > 1)) {
  rcode = pvm_spawn("./mul_mat1", NULL, PvmTaskDefault, " ",
NPROC-1,tids);
                    }

        /*************************************************************************
         Incribimos a la tarea actual en un grupo de tareas, inum es el
    identificador de la tarea dentro del grupo
        *************************************************************************/

 inum = pvm_joingroup("grupo");
 printf("%x:del grupo #%d\n",mtid,inum);


        /* si el id en el grupo es 0 es la tarea principal */
  if (inum == 0)
{

  /* inicializa matrices A y B */

  printf("\n Inicia matriz A");
  for (i=0; i<NRA; i++){
   for (j=0; j<NCA; j++){
    A[i][j]= (i+1)/(NRA*1.0)+(j+1)/(NCA*1.0);
                  }
                              }
  printf("\n Matriz A terminada");

  printf("\n Inicia matriz B");

  for (i=0; i<NCA; i++){
   for (j=0; j<NCB; j++){
         B[i][j]= (i+1)/(NCA*1.0)*(j+1)/(NCB*1.0);
                         }
            }
   printf("\n Matriz B terminada");

     } /* fin if*/
```

```
printf("\n Termina de inicializar las matrices, inum %d \n",inum);


/* ************************************************************************
Aqui empleamos la sincronización por barrera esperar en la barrera hasta
 que todos los procesos esten listos(inscritos) en el grupo*/

 if (pvm_barrier("grupo", NPROC)) pvm_perror("barrier");

 printf("\n Termina Barrera\n");


/* si el id del grupo es la tarea principal inum=0 */
 if (inum == 0) {

/* Envia a todos los miembros del grupo la matriz B */
  rcode = pvm_initsend(PvmDataRaw);
  if (pvm_pkdouble(&B[0][0],NCA*NCB,1)) pvm_perror(NULL);
  rcode = pvm_bcast("grupo", BTIPO);
       }

 /************************************************************************
 no es la tarea principal, entonces espera datos
 ************************************************************************/

 else {

  rcode = pvm_recv(-1, BTIPO);
  if (pvm_upkdouble(&B[0][0],NCA*NCB,1)) pvm_perror(NULL);
 }

 /*
Si es la tarea principal.Se reparten los renglones de A a todas las
 demas tareas de el grupo */

 if (inum==0)

if(pvm_scatter(&ren_ents[0][0],&A[0][0],RENGLONES*NCA,PVM_DOUBLE,7,"grupo",0))
pvm_perror(NULL);

/* todos esperan en la barrera hasta que el scateer haya terminado
, se hayan repartido todos los renglones de A a todas las tateas
inccluyendo a la principal */

 if (pvm_barrier("grupo", NPROC)) pvm_perror("barrier");

 /* limpia la matriz auxiliar para salidas del resultado de la
multiplicacion */

 for (k=0; k<NCB; k++)
  for (i=0; i<RENGLONES; i++)
   ren_sal[i][k] = 0.0;

 /* Como todas las tareas tienen ya asignados los datos
correspondientes se lleva a cabo la operacion de multiplicacion con
cada quien sus respectivos datos*/

 for (i=0; i<RENGLONES; i++)
  for (j=0; j<NCA; j++)
```

```
  for (k=0; k<NCB; k++)
   ren_sal[i][k] = ren_sal[i][k] + ren_ent[i][j] * B[j][k];


/*****************************************************************************
La parte siguente del programa recoge los renglones resultantes y los
guarda en la matriz C
*****************************************************************************/

 if (pvm_barrier("grupo",NPROC)) pvm_perror("barrera");
 mtipo = 2;


 /* recoge los datos de todo el grupo y los almacena en la matriz de
 salida ren_sal*/
 rcode = pvm_gather(C, ren_sal, RENGLONES*NCB, PVM_DOUBLE, mtipo,"grupo", 0);


/* todos esperan en la barrera hasta que termine la funcion pvm_gather*/
 if (pvm_barrier("grupo",NPROC)) pvm_perror("barrier");


/* si es la tarea principal (padre)*/
 if (inum == 0) {
  /* imprime resultados */
  for (i=0; i<NRA; i++) {
   for (j=0; j<NCB; j++)
         printf("%6.2f  ", C[i][j]);
    printf("\n");
  }

 }

 /* las tareas dejan su grupo y salen de PVM */

 rcode = pvm_lvgroup("grupo");
 rcode = pvm_exit();

}
```

4.8.7. PROGRAMA PIPELINE

Este programa calcula la raíz cuadrada de un arreglo de números, para ello utilizamos un método numérico que se representa con la siguiente formula:

$$x_{j+1} = \frac{1}{2}\left(x_j + \frac{a}{x_j}\right)$$

Donde a es el número del que se quiere obtener la raíz
x_j es el valor actual [4].Este valor empieza en $x_0 = 1$
x_{j+1} es el valor siguiente de x_j.
La tolerancia es el error que se presenta de acuerdo al número de iteraciones que se realicen, entre más iteraciones se realicen es menor el error. La tolerancia se representa de la siguiente manera:

$$tol = \left|x_j - x_{j+1}\right|$$

La concurrencia la encontramos en que es la misma operación, pero para datos diferentes.

Para poder llevar a cabo varias iteraciones necesitamos el valor anterior obtenido, es decir x_j. Entonces podemos dividir el algoritmo en fases y que cada máquina desarrolle una de las fases . Así llegamos al paradigma de programación " *Data Pipeline*", donde cada uno de los números que pertenecen al arreglo pasará por la primera fase y el resultado obtenido pasará a la segunda fase y así sucesivamente, de esta manera todos los procesadores se encontrarán trabajando.

```
/*****************************************************************************
 *
 *Programa que ejemplifica el paradigma de programación Data Pipeline
 *
 *****************************************************************************/
#include<stdio.h>
#include "pvm3.h"
#define MAX 10

main(int argc, char *argv[])
{
int nproc;
int i,k;
int numtareas;
int quien;
int tipo_msg;
```

[4] Métodos Numéricos
Francis Scheid, segunda edición
McGraw-Hill

```
int cuenta;
int mi_tid;
int tids[MAX];
int x_0=1;
float x_j,x_j1;
float a[]={81,9,20,25};
float raiz=0,tolerancia;

struct pvmhostinfo *hostp[MAX];

printf("Programa pipeline \n");

nproc=atoi(argv[1]);

if((nproc<1)||(nproc>10))
nproc=MAX;

/* Obtiene el identificador de la tarea dentro de PVM*/

mi_tid=pvm_mytid();

/*Genera las tareas dentro de la máquina virtual*/

numtareas=pvm_spawn("./pipevk1",(char**)0,0,"",nproc,tids);

/*
* Imprime los identificadoires de las tareas
*/

for(i=0;i<nproc;i++)
{
printf("tarea[%d]: %d \n",i,tids[i]);
}

/*
* En este for lee el arreglo y lo envia a los nodos. Hasta que no termina
* de enviar no recibe mensajes
*/

for(k=0;k<4;k++){

        quien=mi_tid;

        cuenta=0;
/*
* Realiza la primera iteración del método con el dato leido del arreglo
*/
 x_j1=0.5*(x_0+(a[k]/x_0)); //se envian a y x_j1

/*
* Se establece la etiqueta con la que se identificaran los datos
*/
        tipo_msg=99;


/*
* Inicia el buffer y comienza el envio de los datos
*/
        printf("mandando dato: %d %d \n",cuenta,quien);
```

```
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&cuenta,1,1);
        pvm_pkint(&nproc,1,1);
        pvm_pkint(&quien,1,1);
        pvm_pkfloat(&a[k],1,1);
        pvm_pkfloat(&x_j1,1,1);
        pvm_pkint(tids,nproc,1);
        pvm_send(tids[cuenta],tipo_msg);
/*
* Cambia el tipo de mensaje para poder recibir
*/
        tipo_msg=100;
        printf("Esperando a recibir \n");
}

/*
* Ahora solo recibe hasta que se completa el número de datos que envio
*/
for(k=0;k<4;k++){
        pvm_recv(-1,tipo_msg);
        pvm_upkint(&cuenta,1,1);
        pvm_upkint(&quien,1,1);
  pvm_upkfloat(&tolerancia,1,1);
        pvm_upkfloat(&raiz,1,1);


        printf("cuenta: %d ultimo: %d Tolerancia %f Raiz: %f \n",cuenta,quien,raiz,tolerancia);

}
/*
* Sale de la máquina virtual
*/
pvm_exit();
}

/*****************************************************************************
* Segundo programa del paradigma Data Pipeline. Este programa se
* ejecuta en las tareas que genera el primer programa. Se encarga
* de realizar las iteraciones siguientes sobre el dato que se envia.
* Cada nodo que lo ejecuta se convierte en una iteración por la que
* pasan los datos.
*****************************************************************************/
#include<stdio.h>
#include <math.h>
#include "pvm3.h"
#define MAX 10

main()
{
int nproc;
int i,k;
int num_tarea,quien,padre,tipo_msg;
int cuenta;
int mi_tid;
float tolerancia;
float a,x_j,x_j1; //se agrega
int tids[MAX];
struct pvmhostinfo *hostp[MAX];
FILE *fp;
```

```
char s[50];

/* Recibe los datos que se envian desde el primer proceso
* o proceso padre. Con el for que sigue a continuación lee
* los datos como van llegando hasta acompletarlos.
*/

for(k=0;k<4;k++){

/*
* Obtengo el identificador de la tarea dentro de PVM
*/

mi_tid=pvm_mytid();

/*
* Creo un archivo con mi identificador. Este archivo es solo para
* verificar el funcionamiento de cada tarea y para depuración
*/

sprintf(s,"archivo.%d",mi_tid);
fp=fopen(s,"w");

/*
* Establezco la etiqueta con la cual puedo recibir los datos
*
*/

tipo_msg=99;

fprintf(fp,"Mi identifcador de tarea: % d \n",mi_tid);

/*
* Leo del buffer los mensajes que tengan la etiqueta que ya establecida
*/

pvm_recv(-1,tipo_msg);
pvm_upkint(&cuenta,1,1);
pvm_upkint(&nproc,1,1);
pvm_upkint(&quien,1,1);
pvm_upkfloat(&a,1,1);
pvm_upkfloat(&x_j,1,1);
pvm_upkint(tids,nproc,1);

/*
* Imprimo mi información en el archivo de depuración
*/

fprintf(fp,"cuenta: %d from: %d a: %f x_j+1: %f\n",cuenta,quien,a,x_j);
fprintf(fp,"Identificadores de las tareas\n");

for(i=0;i<nproc;i++)
{
 fprintf(fp,"Tarea [%d] : %d\n",i,tids[i]);
 }
```

```
/*
* Obtengo el TID del proceso que me creo
*/

padre=pvm_parent();
fprintf(fp,"padre: %d \n",padre);

/*
* realizo la iteración del método
*/

x_j1=0.5*(x_j+(a/x_j)); //se envian a y x_j1

tolerancia=(fabs(x_j-x_j1));
fprintf(fp,"tolerancia: %f",tolerancia);
fclose(fp);

/*
*Aumento la cuenta del número de procesos por los que
* ha pasado la información
*/

cuenta++;

/*
* En este if se contemplan dos casos:
* 1. Aún no soy el último proceso. Entonces establezco que los datos
* que voy a enviar tendran una etiqueta 99 y los mando hacia adelante.
* 2. Si soy el ultimo, establezco una etiqueta que indica que los
* datos van de regreso al padre.
*/

if (cuenta<nproc){
 tipo_msg=99;
 pvm_initsend(PvmDataDefault);
 pvm_pkint(&cuenta,1,1);
 pvm_pkint(&nproc,1,1);
 pvm_pkint(&mi_tid,1,1);
 pvm_pkfloat(&a,1,1);
 pvm_pkfloat(&x_j1,1,1);
 pvm_pkint(tids,MAX,1);
 pvm_send(tids[cuenta],tipo_msg);
 }
else
 {
 tipo_msg=100;
 pvm_initsend(PvmDataDefault);
 pvm_pkint(&cuenta,1,1);
 pvm_pkint(&mi_tid,1,1);
 pvm_pkfloat(&x_j1,1,1); //se agrega
 pvm_pkfloat(&tolerancia,1,1);
 pvm_send(padre,tipo_msg);
 }
}
pvm_exit();
}
```

A continuación se muestra cómo cada iteración es pasada por cada una de las fases. Todos los resultados son regresados a la tarea principal.

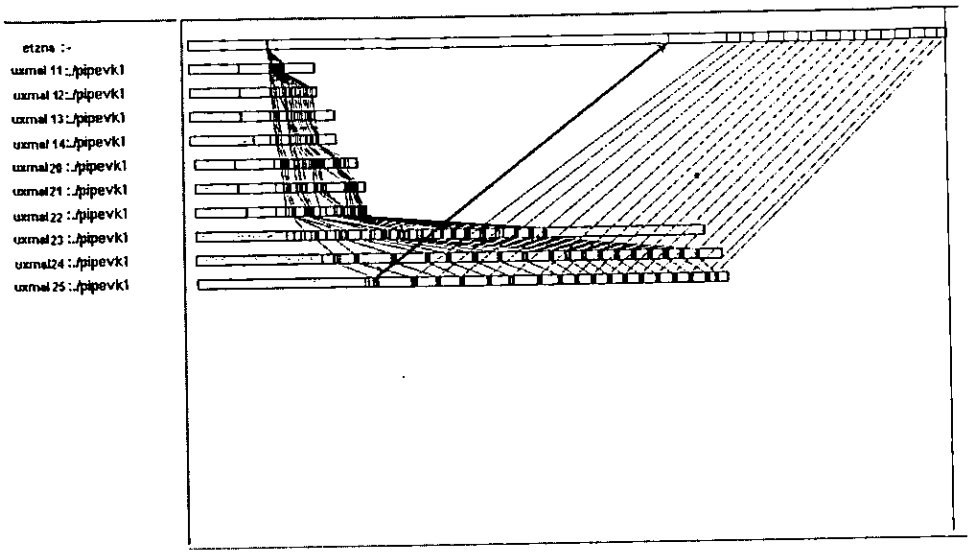


Figura 4. 13 Paso de mensajes del programa relativo al paradigma *Pipeline*.

En el siguiente programa se retoma el paradigma de *Data Pipeline*, sólo que ahora los datos no se leen directamente de memoria, sino de un archivo de texto que se ingresa como parámetro junto al número de procesadores con los que va a trabajar. En esta versión los procesos son terminados una vez que se encuentra la raíz del número dado. Así cada vez que se lee un nuevo dato se genera su "*pipe*" para que obtenga su raíz.

```
/*****************************************************************************
*
* Programa que inicia el pipeline del proceso.
*
*****************************************************************************/
#include<stdio.h>
#include "pvm3.h"
#define MAX 10

main(int argc, char *argv[])
{
int nproc,num=0;
int i,j,rc;
int num_tarea;
int quien;
int tipo_mensaje;
int cuenta;
int mi_tid;
int tids[MAX];
int termina;
int x_0=1;
```

```
float a,x_j,x_j1;
float raiz=0;
float tolerancia;
FILE *archivo;
struct pvmhostinfo *hostp[MAX];

 if((archivo=fopen(argv[2],"r"))==NULL) {
   printf("\n no se puede abrir %s \n\n",argv[2]);
   }

  else{ //else de lectura de archivo

   nproc=atoi(argv[1]);


/* ********************************************************
Agrego la lectura secuencial del archivo. Termina hasta
que encuentra fin de archivo.
******************************************************** */

while(!feof(archivo)){

/* ********************************************************
Declaro la tolerancia para todos los procesos empiezo la
lectura del archivo secuencial
******************************************************** */

tolerancia=0.001;
fscanf(archivo,"%f",&a);

printf("%d .: datos leidos desde el archivo-> a: %f ,tol: %f \n",num,a,tolerancia);

/* ********************************************************
Verifico que el número de procesos sea valido, si no lo es
tomo el máximo por default.
******************************************************** */

if((nproc<1)||(nproc>10))
nproc=MAX;

/* ********************************************************
Obtengo mi identificador dentr de PVM
******************************************************** */

mi_tid=pvm_mytid();

/* ********************************************************
Capturo la salida de todos los procesos hijos
******************************************************** */

rc = pvm_catchout(stdout);

/* ********************************************************
Genero los procesos hijos.
******************************************************** */

num_tarea=pvm_spawn("./pipev1",(char**)0,0,"",nproc,tids);
quien=mi_tid;
cuenta=0;
```

```
/* ********************************************************
Realizo la primera iteración del método
******************************************************** */

x_j1=0;
x_j1=(0.5*(x_0+(a/x_0))); //se envian a y x_j1

/* ********************************************************
Envio los datos que tengo.
******************************************************** */

tipo_mensaje=99;
printf("mandando dato: cuenta: %d quien:%d a: %f tolerancia: %f
\n",cuenta,quien,a,tolerancia);
pvm_initsend(PvmDataDefault);
pvm_pkint(&cuenta,1,1);
pvm_pkint(&nproc,1,1);
pvm_pkint(&quien,1,1);
pvm_pkfloat(&a,1,1);
pvm_pkfloat(&x_j1,1,1);
pvm_pkfloat(&tolerancia,1,1);
pvm_pkint(tids,nproc,1);
pvm_send(tids[cuenta],tipo_mensaje);

/* ********************************************************
Espero recibir resultados
******************************************************** */

tipo_mensaje=100;
printf("Esperando a recibir \n");
pvm_recv(-1,tipo_mensaje);
pvm_upkint(&cuenta,1,1);
pvm_upkint(&quien,1,1);
pvm_upkfloat(&raiz,1,1);


/* ********************************************************
Imprimo los resultados en pantalla
******************************************************** */

printf("num %d cuenta: %d last: %d Raiz: %f \n",num,cuenta,quien,raiz);


}//while

/* ********************************************************
Cierro el archivo secuencial
******************************************************** */

fclose(archivo);

/* ********************************************************
Salgo de PVM
******************************************************** */

pvm_exit();
 }// Se cierra else de lectura de archivo
} //main
```

```
/*****************************************************************************
*
* Segundo programa de pipeline.
*
*****************************************************************************/

#include<stdio.h>
#include <math.h>
#include "pvm3.h"
#define MAX 10

main()
{
int nproc;
int i;
int num_tarea,quien,padre,tipo_mensaje;
int cuenta;
int mi_tid,maestro;
int termina=0;
float tolerancia;
float a,x_j,x_j1;
float val_t;
int tids[MAX];
struct pvmhostinfo *hostp[MAX];
FILE *fp;
char s[50];


/* ************************************************
Obtengo mi identificado dentro de pvm
************************************************** */

mi_tid=pvm_mytid();

/************************************************
Creo el archivo en donde voy a guardar mis datos
************************************************ */
sprintf(s,"file.%d",mi_tid);
fp=fopen(s,"w");
fprintf(fp,"Mi identificador de tarea: % d \n",mi_tid);

/* ************************************************
Desempaqueto lo que recibo
************************************************ */
tipo_mensaje=99;
pvm_recv(-1,tipo_mensaje);
pvm_upkint(&cuenta,1,1);
pvm_upkint(&nproc,1,1);
pvm_upkint(&quien,1,1);
pvm_upkfloat(&a,1,1); //se agrega
pvm_upkfloat(&x_j,1,1); //se agrega
pvm_upkfloat(&val_t,1,1); //se agrega
pvm_upkint(tids,nproc,1);

printf("datos recibidos de : %d",quien);

/* ************************************************************
Imprimo en mi archivo lo que me llega
```

```
********************************************************************* */

fprintf(fp,"cuenta: %d from: %d a: %f x_j+1: %f tolerancia: %f\n",cuenta,quien,a,x_j,val_t);
fprintf(fp,"Identificadores de las tareas\n");
for(i=0;i<nproc;i++)
{
 fprintf(fp,"Tarea [%d] : %d\n",i,tids[i]);
 }

/* ****************************************************************
Obtengo el identificador de mi padre y lo imprimo.
**************************************************************** */
padre=pvm_parent();
fprintf(fp,"padre : %d \n",padre);

/* ****************************************************************
Operaciones para obtener el valor de la raiz
**************************************************************** */

x_j1=(0.5*(x_j+(a/x_j))); //se envian a y x_j1

printf("\n x_j1 de %d -> %f\n",mi_tid,x_j1);

/* ****************************************************************
Evaluamos la tolerancia y la escribimos
**************************************************************** */
tolerancia=(fabs(x_j-x_j1));
fprintf(fp,"tolerancia: %f",tolerancia);

/* ****************************************************************
Cierro el archivo.
**************************************************************** */

fclose(fp);

/* ****************************************************************
Incremento la cuenta del numero de procesos por el que ha pasado
**************************************************************** */
cuenta++;

/* ****************************************************************
Envio mis datos al siguiente proceso, verificando la tolerancia
**************************************************************** */
if (cuenta<nproc)
 {
 printf("empiezo envio de datos de %d a %d\n",mi_tid,tids[cuenta]);
 printf("cuenta: %d en %d \n",cuenta,mi_tid);
 tipo_mensaje=99;
 pvm_initsend(PvmDataDefault);
 pvm_pkint(&cuenta,1,1);
 pvm_pkint(&nproc,1,1);
 pvm_pkint(&mi_tid,1,1);
 pvm_pkfloat(&a,1,1); //se agrega
 pvm_pkfloat(&x_j1,1,1); //se agrega
 pvm_pkfloat(&val_t,1,1); //se agrega
 pvm_pkint(tids,MAX,1);
 pvm_send(tids[cuenta],tipo_mensaje);
 printf("Tolerancia igual a: %f\n",tolerancia);
 }
```

```
else
 {

/* ************************************************************
Envio los datos del resultado al padre.
************************************************************ */
 printf("De regreso al padre\n");
 printf("Tolerancia igual a: %f\n",tolerancia);
 tipo_mensaje=100;
 pvm_initsend(PvmDataDefault);
 pvm_pkint(&cuenta,1,1);
 pvm_pkint(&mi_tid,1,1);
 pvm_pkfloat(&x_j1,1,1); //se agrega
 pvm_send(padre,tipo_mensaje);
 }
pvm_exit();
}// main
```

En la siguente figura podemos apreciar la diferencia con el programa anterior, en este caso se vuelven a crear las tareas (fases) por las que pasarán los números almacenados en el archivo.

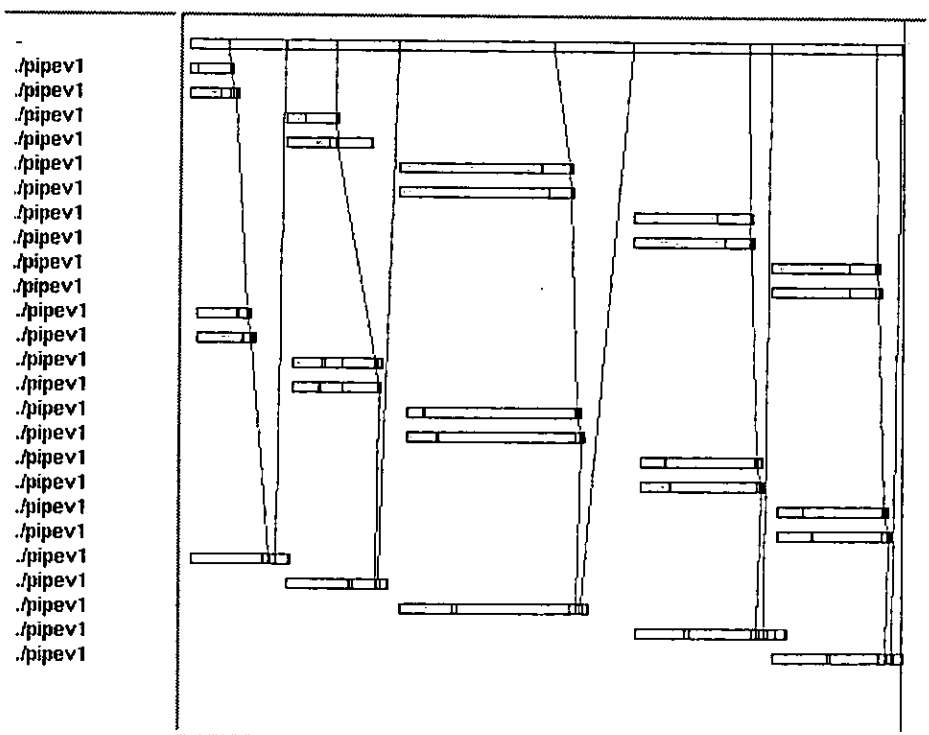


Figura 4. 14 Paso de mensajes del programa *Pipeline*.

Capítulo V

Análisis del Desempeño

5. ANÁLISIS DEL DESEMPEÑO

Para medir el rendimiento de equipos de cómputo, por lo regular se desarrollan programas llamados "*Benchmark*". Los "*Benchmarks*" son utilizados primordialmente para:

- Encontrar valores que demuestren la superioridad de un sistema o arquitectura.
- Estudiar el desempeño de la arquitectura, de computadoras paralelas. Este tipo de "*Benchmarks*" se conocen como "*workloads*".
- Realizar comparaciones entre arquitecturas de computadoras personales y de alto rendimiento.

Existe un gran número "*Benchmarks*" reconocidos a nivel mundial, sin embargo sólo unos pocos guardan relación real con programas, es decir, se supone que los resultados arrojados miden algo intrínseco a la máquina, pero en realidad es difícil identificar con seguridad qué es lo que está siendo medido. Esta situación es análoga a lo que ocurre cuando se trata de medir la inteligencia humana con un test. Los "*Benchmarks*" pueden ser clasificados como sigue:

Aplicación Real, es aquél en que se utiliza una aplicación a la cual se le ha agregado la medición del tiempo de ejecución del mismo.

Kernels, un kernel es un pequeño programa que encapsula un lazo interno de alguna aplicación. El objetivo aquí es gastar tiempo ejecutando un código kernel indefinidamente.

Sintéticos, son programas que tienen propiedades estadísticas las cuales se unen con extensas bases de datos de programas para tareas específicas.

Comerciales: la cantidad disponible es inmensa. La mayoría de estos programas han sido desarrollados para ayudar a tomar decisiones, por ejemplo los laboratorios BAPco y ZD poseen una serie de programas que tratan de medir el rendimiento de PC's compatibles, en aplicaciones reales. Otros como AIM, tratan de asistir en la toma de decisiones cuando se hacen comparaciones cruzadas de arquitectura.

Aplicación Específica: son *Benchmarks* que se construyen sobre la base de una especificación, y no sobre un programa real. Los desarrolladores tienen una gran libertad de elegir cómo construir el *Benchmark*, pero están obligados a cumplir con ciertas especificaciones del cliente.

Bechmaks juguete: este tipo de programas, realizan pruebas sobre algunos aspectos del rendimiento del sistema, uno de los más famosos es el

Benchmark de Integración de Stanford que fue usado para estudiar el rendimiento de máquinas RISC.

Cualquiera que sea el "*Benchmark*" escogido, siempre existe el problema de determinar qué es lo se está midiendo en realidad.

En nuestro caso podemos decir que nuestros "*Benchmarks*", son los programas realizados que efectúan cálculos, como son el calculo de *pi* ,la multiplicación de matrices y la raíz cuadrada de un arreglo de números y son los que usaremos para el análisis del desempeño del cluster .

Las pruebas que consideramos importantes para este análisis son las siguientes:

- Medir el tiempo que se tarda el programa en ejecutarse en cierto número de procesadores hasta llegar hasta el número total de máquinas que conforman el *cluster*, la carga de trabajo para los procesadores es constante.

- Medir el tiempo que tardan todas las máquinas del cluster en ejecutar el programa, pero variando la carga de trabajo en cada procesador.

Para obtener el tiempo total ejecutamos el programa 4 veces desde la línea de comandos la siguiente forma:

 # time ./nombre programa

Se consideran cuatro corridas para obtener un valor más estable sacando el promedio y con éste poder graficar.

En la primera ejecución el programa es cargado desde el disco a memoria, por lo que se tardará un poco más en ejecutarse. Una vez que se encuentra en memoria la ejecución se realiza de manera más rápida.

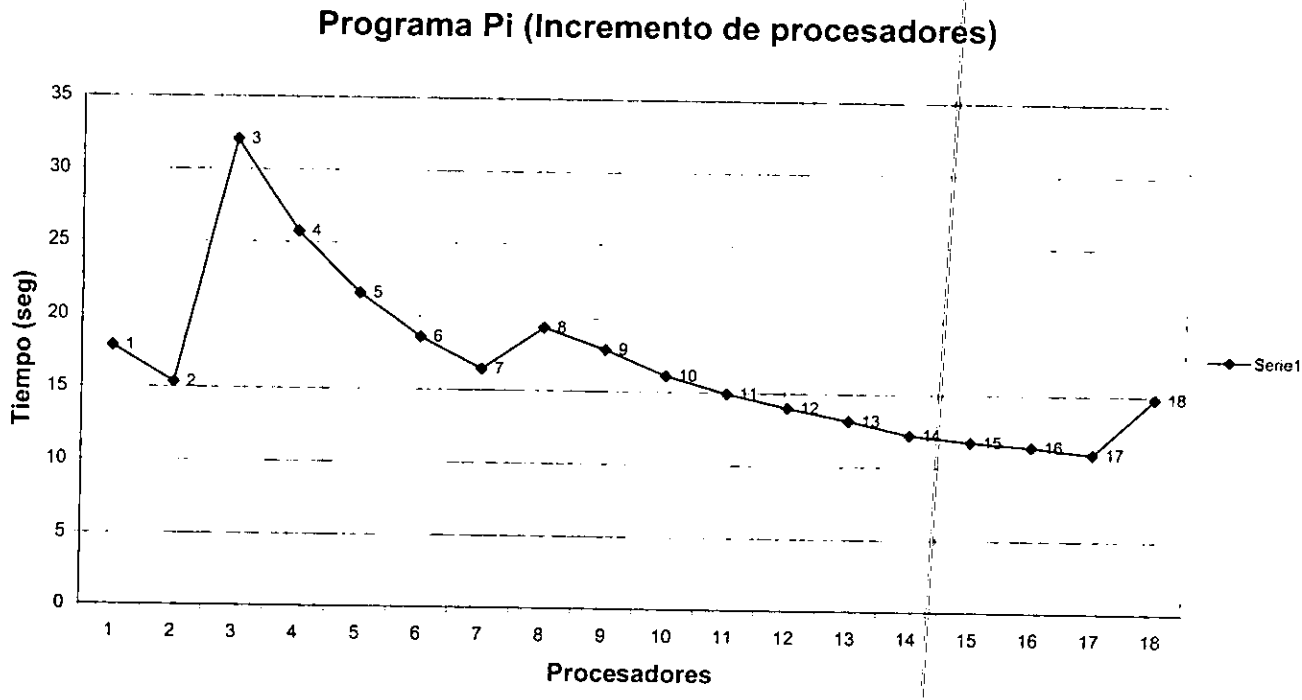
A continuación mostramos las tablas de los valores obtenidos en cada programa y tipo de prueba así como su respectiva gráfica y el análisis de la misma.

5.1. PROGRAMA DEL CÁLCULO DE *PI*

Primero veamos qué sucede con un número de iteraciones fijo, de $N = 2^{25}$, si ejecutamos el programa aumentando el número de procesadores hasta llegar a los que tenemos disponibles, los resultados se muestran en la siguiente tabla (ver Tabla 5.1) y se representan gráficamente (ver grafica 5.1).

NO. DE PROCESADORES	TIEMPO 1 [seg]	TIEMPO 2 [seg]	TIEMPO 3 [seg]	TIEMPO 4 [seg]	PROMEDIO [seg]
1	12,95	22,91	12,7	22,91	17,8675
2	15,42	15,34	15,33	15,33	15,355
3	32,19	32,02	31,95	31,94	32,025
4	26,01	25,61	25,6	25,61	25,7075
5	21,98	21,55	21,53	21,42	21,62
6	19,07	18,54	18,54	18,37	18,63
7	17,5	16,12	16,15	16,14	16,4775
8	19,74	19,22	19,23	19,15	19,335
9	19,16	17,44	17,45	17,45	17,875
10	16,96	15,8	15,96	15,79	16,1275
11	15,94	14,54	14,58	14,7	14,94
12	15,15	13,49	13,63	13,69	13,99
13	14,14	13,05	12,6	12,8	13,1475
14	13,43	12,43	11,47	11,47	12,2
15	13,05	11,35	11,36	11,29	11,7625
16	12,74	11,36	10,99	10,7	11,4475
17	13,02	10,24	10,21	10,43	10,975
18	15,77	14,31	14,3	14,97	14,8375

Tabla 5. 1 Tiempos obtenidos en la corrida del programa cálculo de pi al ir aumentando el número de procesadores.



Gráfica 5. 1 Procesadores vs tiempo del programa del cálculo de pi.

Antes de empezar con el análisis de esta gráfica cabe aclarar que las dos primeras máquinas que son el servidor y uno de los nodos (en el caso de esta gráfica el segundo), son las más rápidas que tenemos, y a partir del tercer procesador las velocidades promedio son entre 66 y 55 MHz hasta el procesador 17 y de 33 Mhz que es la última máquina agregada (el procesador número 18). Con lo anterior podemos ya podemos explicar el comportamiento de la gráfica:

- La máquina nodo que es la más rápida que las demás junto con el servidor tarda aproximadamente el mismo tiempo en la ejecución del programa que con 11 procesadores ,incluyéndolas.

- Si realizamos el análisis a partir del tercer procesador donde las velocidades son más o menos homogéneas podemos ver que a medida que se agreguen más procesadores el tiempo de ejecución decrementa.

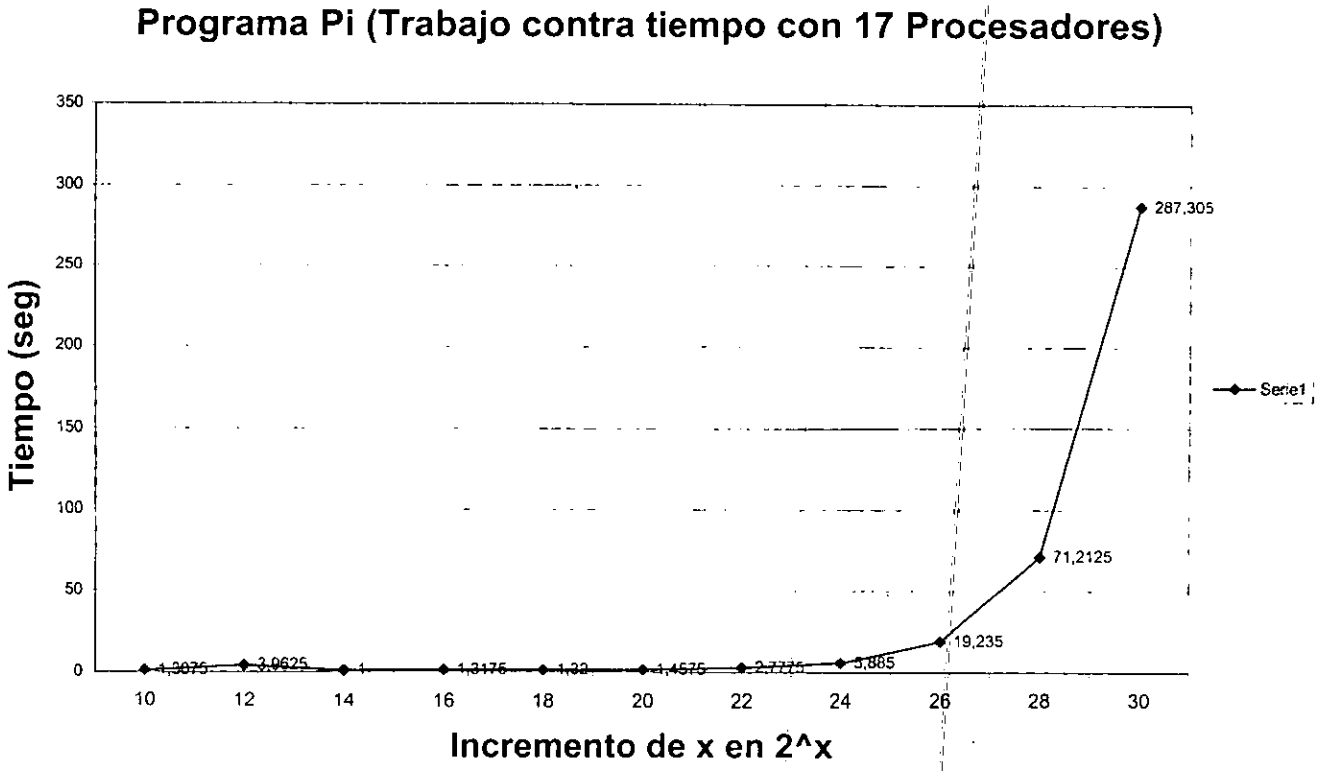
- Al agregar el nodo más lento, el número 18, el tiempo aumentó un poco, es decir disminuyó la capacidad de procesamiento del cluster.

Para el siguiente análisis, el cluster trabajo con 17 nodos, el numero 18 lo quitamos para evitar que alente la corrida de los programas y lo que vamos a ver es qué sucede al ir aumentando la carga de trabajo sobre el cluster, cálculos cada vez mas robustos, iteraciones de 2^x donde x es el valor que se fue aumentando.

En la tabla 5.2 se muestran los tiempos obtenidos en este análisis y en la grafica 5.2 se representa el comportamiento del tiempo al ir aumentado la carga de trabajo.

2 ^X	TIEMPO 1 [seg]	TIEMPO 2 [seg]	TIEMPO 3 [seg]	TIEMPO 4 [seg]	PROMEDIO [seg]
10	2,98	0,69	0,72	0,84	1,3075
12	3,27	1,13	1,09	10,36	3,9625
14	0,86	1,09	1,06	0,99	1
16	2,86	0,72	0,64	1,05	1,3175
18	2,81	0,7	1,03	0,74	1,32
20	2,75	0,98	0,94	1,16	1,4575
22	4,56	1,98	2,53	2,04	2,7775
24	6,89	5,9	5,34	5,41	5,885
26	20,46	18,64	19,15	18,69	19,235
28	72,81	70,52	70,54	70,98	71,2125
30	288,99	286,87	286,68	286,68	287,305

Tabla 5. 2 Tiempos obtenidos en la corrida del programa cálculo de pi al ir aumentando la carga de trabajo en 17 procesadores.



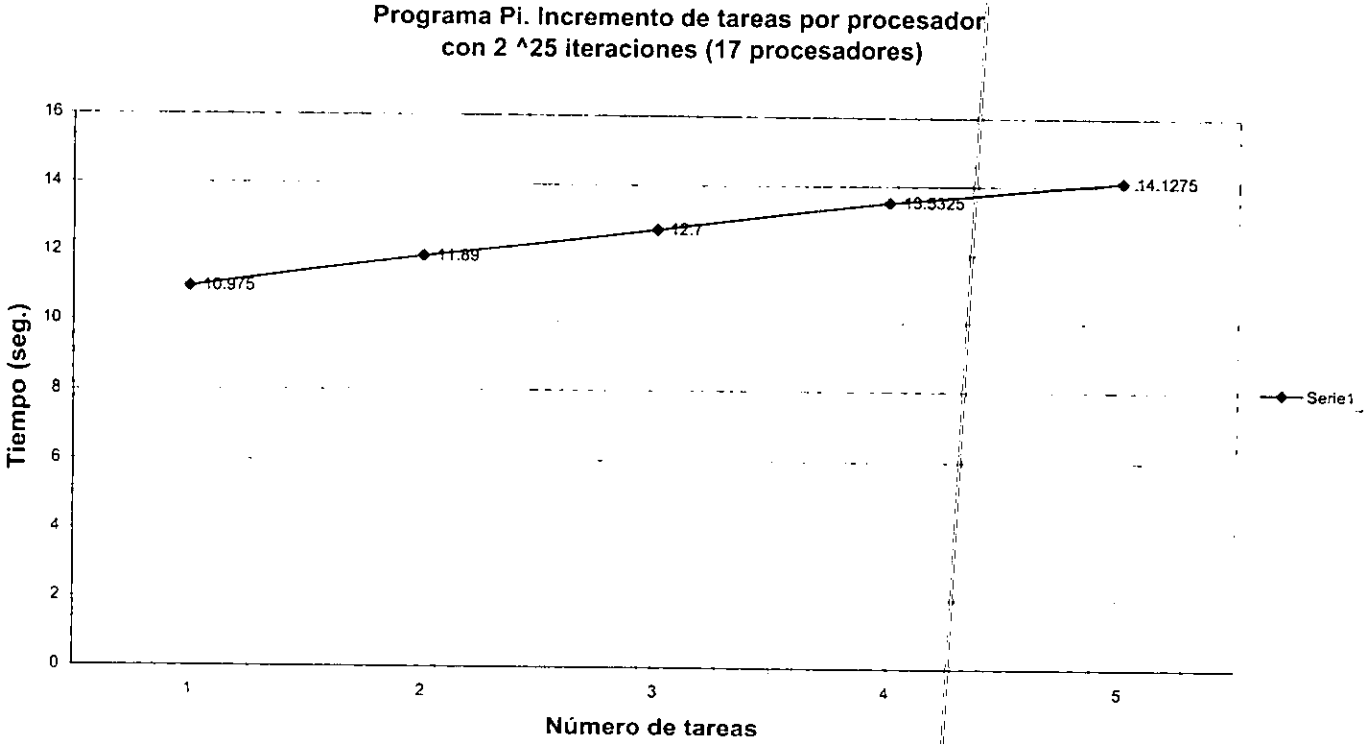
Gráfica 5. 2 Incremento de trabajo vs tiempo en el programa de cálculo de pi.

Como podemos ver la gráfica crece exponencialmente, y esto es debido a que el aumento de la carga de trabajo también es exponencial (2^x), por lo que podemos decir que el aumento del tiempo es proporcional al aumento de carga de trabajo.

Un tercer análisis para este programa es dejar la carga constante (2^{25}), utilizar 17 procesadores y variar el número de tareas asignadas a cada procesador. Los resultados de los tiempos obtenidos fueron los siguientes (ver tabla 5.3) y la representación de los mismos se muestra en la gráfica 5.3:

TAREAS POR PROCESADOR	TIEMPO 1 [seg]	TIEMPO 2 [seg]	TIEMPO 3 [seg]	TIEMPO 4 [seg]	PROMEDIO [seg]
1	13.02	10.24	10.21	10.43	10.975
2	13.36	11.48	11.37	11.35	11.89
3	13.5	12.42	12.41	12.47	12.7
4	14.75	13.05	13.13	13.2	13.5325
5	14.78	14.23	13.79	13.71	14.1275

Tabla 5. 3 Tiempos obtenidos al ir aumentando el número de tareas a cada procesador en el programa del cálculo de pi.



Gráfica 5. 3 Número de tareas en cada procesador vs tiempo en el programa del cálculo de pi.

En la gráfica podemos darnos cuenta que entre más tareas se le asignen a cada procesador mayor tiempo se tarda la ejecución del programa; vemos que no es conveniente dividir el problema (más tareas) en un número mayor con respecto al número de procesadores que tengamos, de todas maneras el numero de iteraciones es el mismo y lo que logramos es perder tiempo de cada procesador al prestar más atención en el envío y recepción de mensajes que en cada tarea asignada. Entonces es conveniente que cada tarea sea vista como un procesador o que a cada procesador se le asigne sólo una tarea.

5.2. MULTIPLICACIÓN DE MATRICES

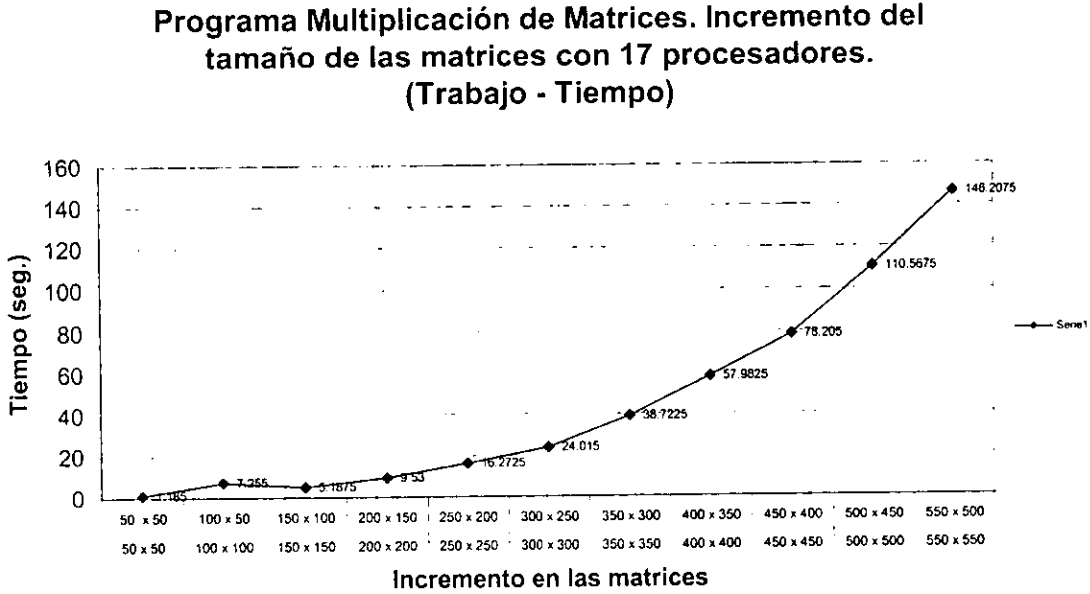
Con este primer análisis pretendíamos ver el comportamiento del cluster (utilizando 17 procesadores) al ir aumentando la carga de trabajo en él, esto es, ir incrementando el tamaño de las matrices.

A continuación se muestran los tiempos obtenidos, con el respectivo tamaño de matrices (ver tabla 5.4).

MATRIZ A REN X COL	MATRIZ B: REN X COL	TIEMPO 1 [seg]	TIEMPO 2 [seg]	TIEMPO 3 [seg]	TIEMPO 4 [seg]	PROMEDIO [seg]
50 x 50	50 x 50	0.36	1.37	1.22	1.79	1.185
100 x 100	100 x 50	4.77	1.93	2.02	20.3	7.255
150 x 150	150 x 100	6.58	4.67	4.82	4.68	5.1875
200 x 200	200 x 150	11.24	9.16	8.76	8.96	9.53
250 x 250	250 x 200	17.58	15.65	15.89	15.97	16.2725
300 x 300	300 x 250	25.24	23.87	23.49	23.46	24.015
350 x 350	350 x 300	39.74	38.77	38.51	37.87	38.7225
400 x 400	400 x 350	57.22	57.79	54.97	61.95	57.9825
450 x 450	450 x 400	80.84	77.71	76.28	77.99	78.205
500 x 500	500 x 450	107.3	115.19	109.89	109.89	110.5675
550 x 550	550 x 500	151.37	141.66	150.3	141.5	146.2075

Tabla 5. 4 Tiempos obtenidos al ir aumentando la carga de trabajo (tamaño de las matrices).

La gráfica 5.4 muestra el comportamiento del análisis anterior.



Gráfica 5. 4 Incremento del tamaño de las matrices vs tiempo, del programa multiplicación de matrices.

En la gráfica notamos que mientras mayor sea la carga de trabajo (matrices más grandes) el tiempo que tarda la ejecución del programa se incrementa, pero no en proporción con el tamaño de las matrices, sino a razones más pequeñas.

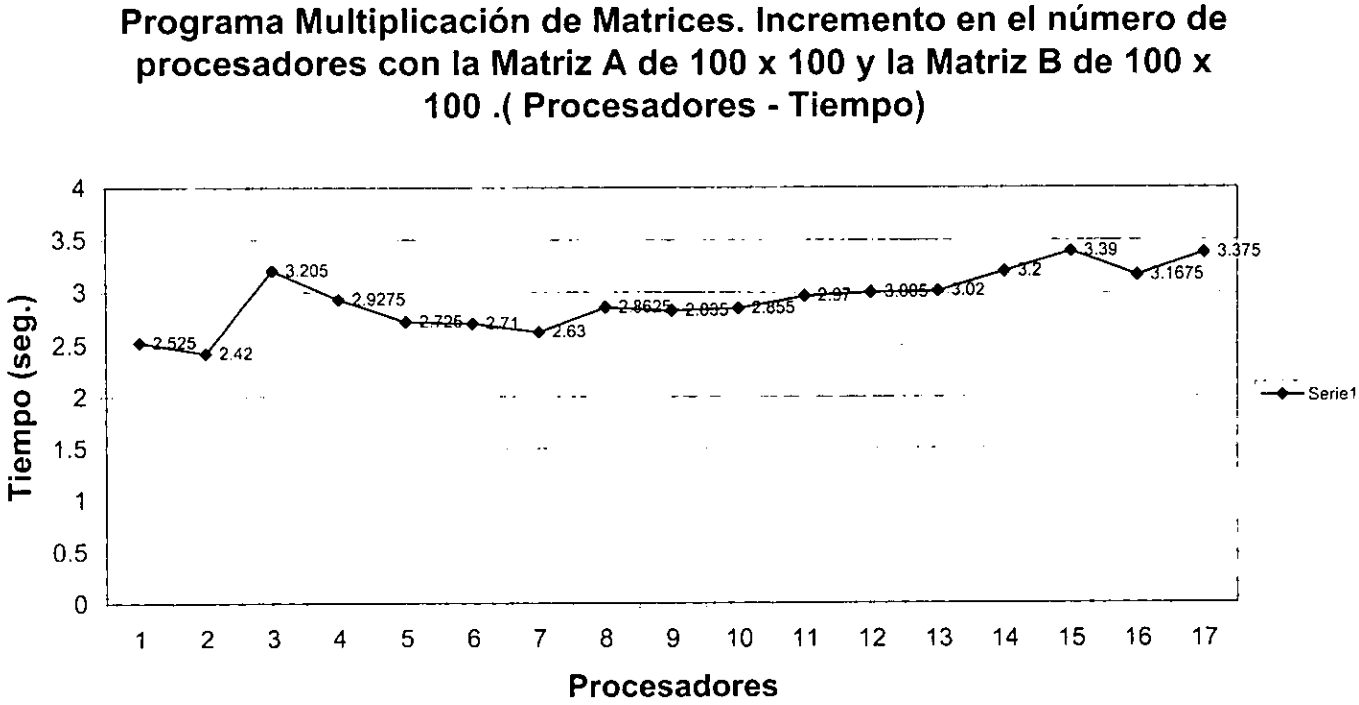
Ahora queremos verificar si paralelizar el algoritmo de multiplicación de matrices es una buena opción, y si el tener un mayor número de procesadores en el cluster nos va a ayudar a mejorar o disminuir el tiempo. Para ello realizamos la prueba de ir incrementando el número de procesadores y dejando una carga de trabajo fija. Esta carga de trabajo va ser de dos formas

a) Las dos matrices A y B de 100x100
b) Una de las matrices de 350x350 y la otra de 350x300

Los datos y el comportamiento del primer caso se muestran en la tabla 5.5 y la gráfica 5.5 respectivamente.

NO. DE PROCESADORES	TIEMPO 1 [seg]	TIEMPO 2 [seg]	TIEMPO 3 [seg]	TIEMPO 4 [seg]	PROMEDIO [seg]
1	2.63	2.51	2.49	2.47	2.525
2	2.6	2.49	2.3	2.29	2.42
3	3.35	3.08	3.09	3.3	3.205
4	3.1	2.82	2.82	2.97	2.9275
5	2.8	2.74	2.69	2.67	2.725
6	2.66	2.78	2.76	2.64	2.71
7	2.81	2.6	2.56	2.55	2.63
8	3.14	2.77	2.77	2.77	2.8625
9	3.01	2.8	2.78	2.75	2.835
10	3.13	2.78	2.79	2.72	2.855
11	3.39	2.82	2.83	2.84	2.97
12	3.38	2.91	2.84	2.89	3.005
13	3.38	2.89	2.89	2.92	3.02
14	3.42	2.99	3.36	3.03	3.2
15	4.12	3.1	3.19	3.15	3.39
16	3.54	2.96	3.04	3.13	3.1675
17	3.99	3.21	3.12	3.18	3.375

Tabla 5. 5 Tiempos obtenidos en la corrida del programa multiplicación de matrices al ir aumentando el número de procesadores con matrices fijas de 100x100.

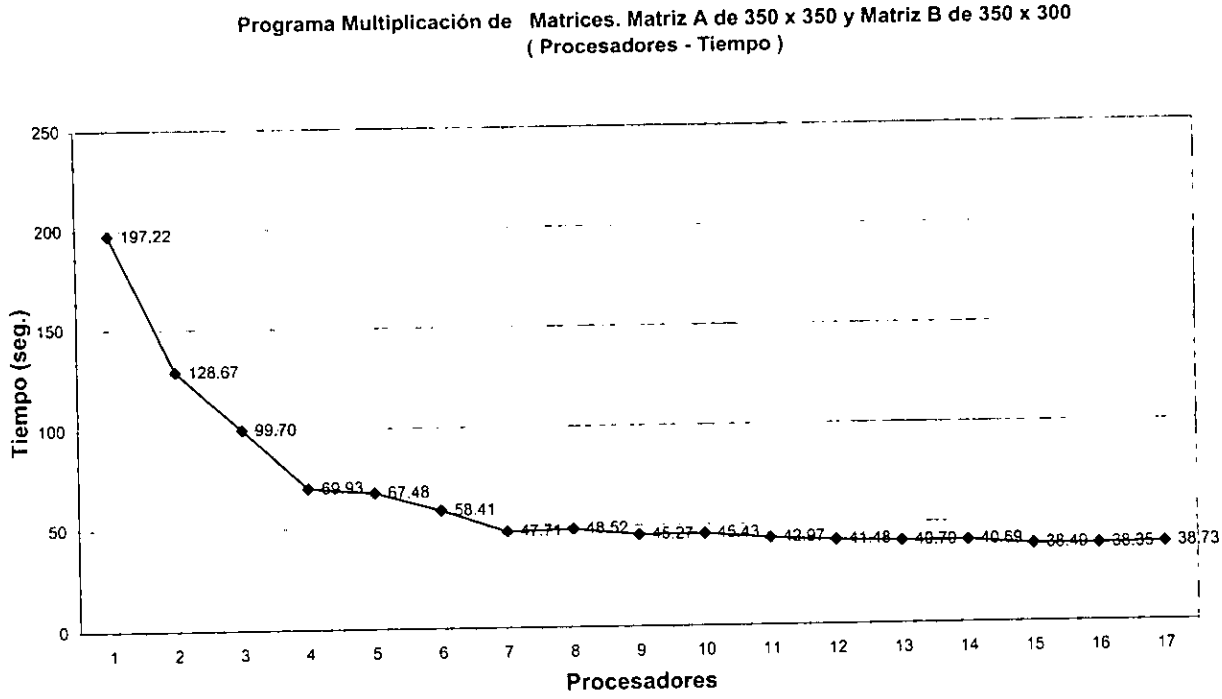


Gráfica 5. 5 Número de procesadores vs tiempo del programa multiplicación de matrices.

Los datos y gráfica del segundo caso. Son (ver tabla 5.6 y gráfica 5.6):

No. De Procesadores	TIEMPO 1 [seg]	TIEMPO 2 [seg]	TIEMPO 3 [seg]	TIEMPO 4 [seg]	PROMEDIO [seg]
1	193.32	198.85	198.4	198.3	197.2175
2	97.09	140.18	135.1	142.3	128.6675
3	99.95	99.49	99.6	99.75	99.6975
4	79.39	62.52	67.3	70.5	69.9275
5	67.76	67.36	67.4	67.38	67.475
6	58.27	58.82	58.3	58.25	58.41
7	54.37	45.04	45.7	45.72	47.7075
8	50.2	47.87	47.9	48.1	48.5175
9	48.11	44.25	44.5	44.2	45.265
10	45.39	45.45	45.46	45.4	45.425
11	43.06	42.82	43.1	42.9	42.97
12	42.37	40.83	41.8	40.9	41.475
13	41.45	40.54	40.3	40.5	40.6975
14	41.81	40.83	40.1	40.01	40.6875
15	38.84	38.36	38.4	38.35	38.4875
16	39.24	38.03	38.1	38.01	38.345
17	39.43	38.67	38.5	38.3	38.725

Tabla 5. 6 Tiempos obtenidos en la corrida del programa multiplicación de matrices al ir aumentando el número de procesadores con matriz A de 350x350 y matriz B de350x300.



Gráfica 5. 6 Procesadores vs tiempo, del programa multiplicación de matrices.

De los dos casos analizados notamos que si la carga de trabajo en el cluster es ligera, se mantiene el tiempo en un rango de 2.5 a 3.5 segundos aproximadamente, mientras que en el caso de una carga de trabajo más pesada el tiempo disminuye considerablemente entre más procesadores se empleen. En teoría debería ser igual para ambos casos, pero en el caso uno, como hay menos carga es más el tiempo que se emplea en la comunicación entre las máquinas que el empleado para la realización del cómputo. Entonces podemos decir que no es recomendable utilizar el cluster para realizar cálculos pequeños, ya que en lugar de aumentar el rendimiento disminuye.

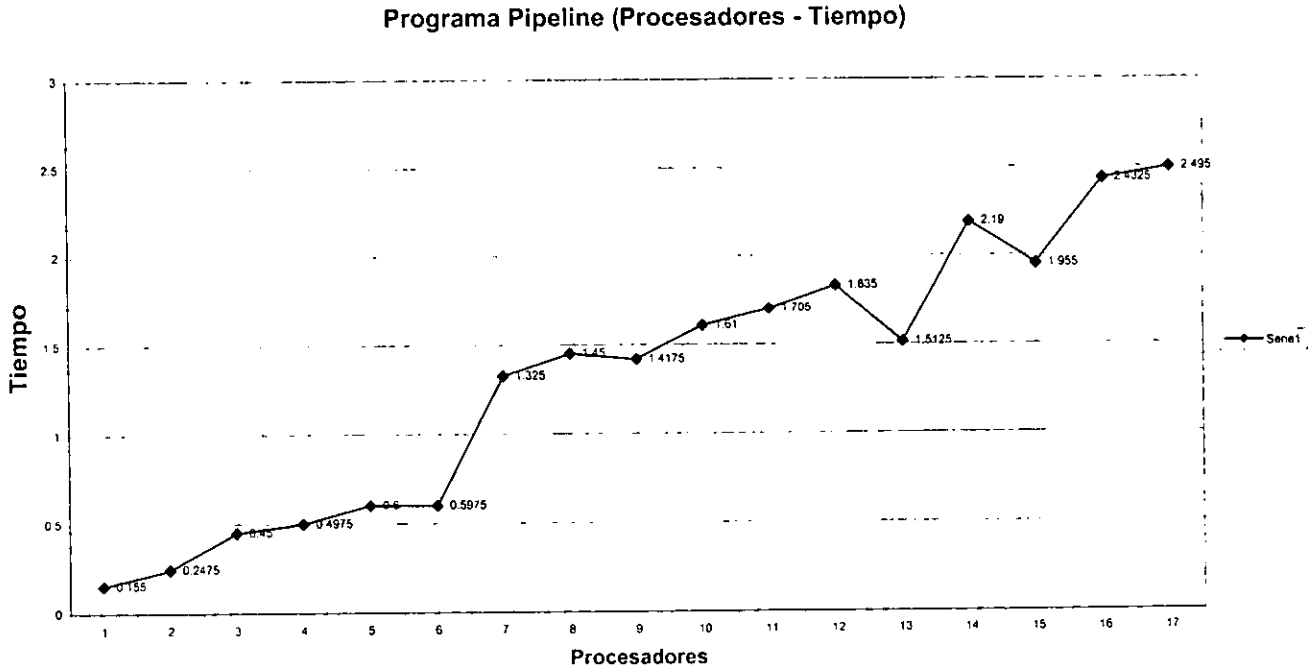
5.3. RAÍZ CUADRADA DE UN ARREGLO DE NÚMEROS

Como este programa trabaja por fases, y cada fase representa una aproximación a la raíz exacta del número proporcionado, entre más fases tengamos el error de aproximación es menor.

El análisis realizado en este programa fue ver qué sucede con el tiempo entre más procesadores tengamos en el cluster. Los datos y gráficas de este análisis se muestran en la tabla 5.7 y gráfica 5.7 respectivamente.

No. De Procesadores	TIEMPO 1 [seg]	TIEMPO 2 [seg]	TIEMPO 3 [seg]	TIEMPO 4 [seg]	PROMEDIO [seg]
1	0.16	0.14	0.16	0.16	0.155
2	0.25	0.24	0.25	0.25	0.2475
3	0.46	0.46	0.46	0.42	0.45
4	0.51	0.51	0.48	0.49	0.4975
5	0.82	0.52	0.55	0.51	0.6
6	0.81	0.52	0.54	0.52	0.5975
7	1.55	1.22	1.27	1.26	1.325
8	2.42	1.37	0.71	1.3	1.45
9	2.7	1.43	0.78	0.76	1.4175
10	1.77	2.14	1.47	1.06	1.61
11	1.79	1.49	2.01	1.53	1.705
12	1.84	2.28	1.62	1.6	1.835
13	1.63	1.72	1.64	1.06	1.5125
14	1.87	3.03	2.06	1.8	2.19
15	1.48	2.39	2.47	1.48	1.955
16	3.09	1.87	2.27	2.5	2.4325
17	2.53	2.32	2.54	2.59	2.495

Tabla 5. 7 Tiempos obtenidos en el aumento de procesadores en el programa de cálculo de la raíz cuadrada.



Gráfica 5. 7 Procesadores vs tiempo al aumentar el número de procesadores, en el programa cálculo de la raíz cuadrada.

Como era de esperarse entre más fases tengamos, el tiempo va a aumentar, y el acercamiento del resultado va a depender del número de procesadores que tenga el cluster. Entonces la ventaja que vamos a tener con este tipo de programa en el cluster es que podemos obtener resultados de cálculos de varios números a la vez.

Conclusiones

CONCLUSIONES

El trabajo de investigación que se presenta con este documento nos permitió acercarnos al mundo del cómputo paralelo. Este acercamiento se dio tanto en la parte referente al hardware (al construir una máquina paralela como lo es un cluster Beowulf), como en el aspecto del software, al entender y manejar los conceptos de programación propios de este ambiente.

De esta forma podemos concluir que al querernos introducir al mundo de la computación paralela y resolver problemas que requieran de este tipo de cómputo, necesitamos de una máquina paralela y un lenguaje de programación, como C, C++ o Fortran que interactúe con las diversas herramientas de paralelización

Al iniciar el desarrollo del trabajo, nos encontramos con el problema de escasez de equipo por lo que tuvimos que iniciar la búsqueda del mismo, una vez localizado éste se tuvo que reciclar, ya que las condiciones de dicho equipo eran consideradas como obsoletas por lo que la mayoría del equipo utilizado tuvo que ser reensamblado. Una vez que se termino con esta fase, se decidió crear un Cluster Beowulf sin discos.

Otra conclusión a la que llegamos en este trabajo, es darnos cuenta que es posible utilizar equipo considerado de desecho para construir una máquina paralela, la cual nos puede servir para aplicar y entender la programación paralela. Quizás una máquina de este tipo no sea considerada una supercomputadora pero para fines didácticos cumple con su cometido. Si dentro de un tiempo no muy lejano, las máquinas con las que ahora se cuentan fueran sustituidas por otras de mayor potencia, se tendría una máquina paralela de mejores condiciones.

Debido a que un cluster que sea considerado del tipo Beowulf debe llevar como Sistema Operativo a Linux, fue la elección por demás evidente. Además de trabajar en diversas plataformas, es más accesible a modificaciones o adecuaciones que otros sistemas operativos, tal es el caso de SUN, que requiere de hardware específico para ser instalado. Además debemos hacer notar que Linux es totalmente gratuito y esto no significa que no se tenga el soporte necesario para su administración y manejo, ya que se encuentra respaldado por un gran número de desarrolladores. En Internet es posible encontrar ayuda acerca de los diferentes aspectos que componen a Linux. Otra de las razones por las que se optó en trabajar en Linux, es que dicho sistema está enfocado al desarrollo, ya que al adquirir el disco de instalación, también adquiere las herramientas básicas de programación como C, g++, gcc, Perl, que pueden ser instaladas al inicio o anexadas posteriormente por el mismo disco de instalación.

En Linux observamos un gran desempeño y rendimiento durante la elaboración del cluster, no presentó problemas con el sistema operativo, y los archivos de configuración son fácilmente modificables.

El utilizar Linux nos permitió conocerlo mejor y aprender algunos aspectos sobre la instalación, administración y el entorno de red que maneja, ya que antes de iniciar este trabajo no teníamos idea de las posibilidades que éste nos podía ofrecer, sino que solo conocíamos algunos comandos básicos a nivel usuario.

En lo que se refiere al cluster Beowulf, encontramos las siguientes características:

- Componentes fáciles de obtener.
- Es fácil de ampliar.
- La gran mayoría del software es libre (tanto el sistema operativo como ambientes de programación).
- La relación rendimiento precio es buena.

En otro orden de ideas, la elección del ambiente de programación y desarrollo que iba a tener el cluster nos llevo a investigar en la literatura disponible las características de las diferentes herramientas de paralelización. Debido a que se quería que el cluster fuera visto como sólo una máquina paralela, se recurrió al uso de PVM. PVM permite ver al conjunto de máquinas como una sola y provee librerías de desarrollo par programar dentro de la misma. Como se mencionó en el capítulo IV, PVM trabaja a través del paso de mensajes entre los diferentes procesos y el intercambio de datos entre los mismos.

En lo que respecta a los programas realizados con PVM tenemos lo siguiente:

- Para paralelizar un algoritmo es necesario detectar las partes en que el algoritmo secuencial puede dividirse sin afectar el resultado, y cada parte de esta división pueda hacerse de manera independiente para al final juntarlos y obtener el resultado deseado.

- El uso de grupos en PVM en ocasiones puede afectar el rendimiento del programa, esto porque aunque hayan acabado varias tareas del grupo no se seguirá adelante hasta que terminen todas juntas.

- Un algoritmo a veces puede ser paralelizado utilizando varios paradigmas de programación y en algunos casos el código puede reducirse de manera notable de un paradigma a otro.

- Independientemente del paradigma de programación que se utilice siempre va a haber un proceso padre y uno o varios procesos hijos:
  - En el caso del *pipeline*, el primer eslabón de la cadena es quien genera los eslabones siguientes y recoge sus datos.
  - En el Maestro – Esclavo, el mismo paradigma indica que el Maestro distribuye las tareas a los procesos que él genera.
  - En el SPMD, aunque sólo existe un programa que se manda a todos los procesadores, existe tanto una parte "padre" como una parte "hijo".

Con el análisis del desempeño del cluster llegamos a las siguientes conclusiones:

- El desempeño del cluster se puede ver afectado por la máquina más lenta que forme parte de éste.
- El desempeño también está delimitado por la velocidad de las máquinas que sea el común denominador, esto es, si hay máquinas más rápidas que el grueso del grupo, estas máquinas tendrán que esperarse a que las demás terminen.
- Dependiendo de las velocidades de las máquinas que conforman el cluster, si tenemos una más veloz que las demás, ésta podría desempeñar el mismo trabajo que 2, 3 o más juntas de menor velocidad.
- No tiene caso utilizar una máquina paralela o paralelizar un programa, si éste no va a realizar un cálculo considerable, ya que de ser así el cluster tardaría más tiempo en realizar la comunicación entre las máquinas que el cálculo especificado.

Las mejoras a futuro sobre este trabajo se consideran en varios aspectos:

En cuanto al cluster en sí, su desempeño se puede mejorar al sustituir las máquinas que ahora lo conforman por otras de mayor capacidad de procesamiento. Las ya existentes se podrían incluir en su conjunto como una sola máquina del nuevo conjunto.

El aspecto de la comunicación entre servidor, nodos y entre los nodos mismos se puede mejorar al aumentar el ancho de banda de la red. Actualmente la red interna del cluster tiene un ancho de banda de 10Mbps y las tarjetas de red tienen capacidad de auto detectar el medio en el que están para poder trabajar en este esquema (Ethernet) o en un esquema similar pero de mayor velocidad conocido como Fast Ethernet, que brinda un ancho de banda de 100Mbps. Pero no sólo las tarjetas de red intervienen en mejorar este esquema, también hay

que mejorar los dispositivos de interconexión, en este caso los hubs, para que se pueda trabajar de esta manera, cambiándolos tal vez a switches (conmutadores).

En cuanto a la programación, se pueden probar más herramientas de paralelización, para poder tener un espectro más amplio en cuanto a las posibilidades de programación en paralelo.

Bibliografía

BIBLIOGRAFÍA

LIBROS

Industrial Strenhth Parallel Computing
Alice E. Konigues
Morgan Kaufmann Publishes
2000

Introduction to Parallell Algorithm
Jhon Willey
1998

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

High Performance Cluster Computing: *Programming and Applications*, Volume 2, 1/e
Rajkumar Buyya,
Monash University, Australia
Copyright 2000, 664 pp.

How to build a Beowulf,
T. Sterling. D. Becker, D.F. Savarese
from MIT Press
1999-05-13

PVM ,*A User`s Guide and Tutorial for Networked Parallel Computing*
Al Geist ,Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaid Sunderam
From MIT Press
Cambrige,Massachusetts
London,England

PVM 3 *USER`s GUIDE and reference Manual*
Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaid Sunderam
From MIT Press

LINUX , *Guía de Instalación y Administración*
Configuración y Programación de Servidores de Internet e Intranet
López Camacho Vicente , García Soler, Moreno Simarro, Tevar Marcilla, Lopez Jaquero, Gutiérrez de Juan, García García, Olivares Montes, Alfaro Cortés , Cortes Bragado
Osborne ,MacGrall-Hill
2001
Sistemas Operativos / Redes

Manual de Administración LINUX
Steve Shah
Osborne ,MacGrall-Hill
2001
LINUX /REDES

LINUX, Recursos para el usuario
James Mohr
Pearson Education
Categoría Sistemas Operativos

Edición Especial LINUX
Jack Tacket, Jr Steve Burnett
Cuarta Edición
Prentice Hall

Enciclopedia del Lenguaje C
Francisco Javier Cevallos
Alfaomega

Estructuras de Datos con C y C++
Langsam Yedidyah, Augenstein Moshe J.
Segunda Edición
Prentice Hall

Como programar en C/C++
Deitel H.M.,Deitel P.J.
Segunda Edición
Editorial Pearson, Prentice Hall

Métodos Numéricos
Francis Scheid
Segunda Edición
McGraw-Hill

Apuntes de Métodos Numéricos
Iriarte,Rafael
División de Ciencias Básicas
Departamento de Matemáticas Aplicadas

ARTICULOS

http://www.ieeetfcc.org/
Cluster Computing White Paper
Version 2.0
29 December 2000
Editor - Mark Baker, University of Portsmouth, UK

An Introduction to PC Cluster for High Performance Computing
Thomas Sterling-California Institute of Technologies and NASA Jet Propulsion Laboratory, USA

Network Technologies
Amy Apon, University off Arkansas , USA and Joachim Worringen, RWTH Aachen, University of Tecnology, Germany.

Linux LX Format, número 1
MC Ediciones
30 junio 2000

DIRECCIONES

http://www.infor.uva.es/

http://www.geotices.com/SiliconValley/Bay/9418/

http://www.cecalc.ula.ve/documentacion/tutoriales/beowulf

http://www.aic.uniovi.es/CyP/HTMs_de_PVM/Config_PVM.htm

http://www.aic.uniovi.es/CyP/HTMs_de_PVM/Install_PVM.htm

http://www.epm.ornl.gov/pvm/intro.html

http://golum.riv.csu.edu.au/~ialtas/module3/index.html

http://spd.ldc.ve/pvm

http://www-jics.cs.utk.edu/pvm_guide.html

http://www.ii.uam.es/~fjgomez/tutorial.html

http://www.cesca.es/esp/SERVICIOS/E_DESARROLLO/pvm.html

http://fscked.org/writings/clusters/cluster.html

http://fscked.org/writings/clusters/dhcpd.html

http://www.slug.org.au/etherboot/

http://www.linuxfocus.org/Castellano/September1998/article63.html

http://www.beowulf.org/intro.html

http://www.netlib.org/pvm3/faq_html/node3.html

http://strix.ciens.ucv.ve/~matcomp/algpar/CLASE1.html

http://agamenon.uniandes.edu.co/~c21437/ma-bedoy/

http://alumnat.upv.es/pla/visfit/2549/AAABNcAATAAAJVwAAs/consejospvm.htm

http://research.cem.itesm.mx/jesus/cursos/compd2/s3/s3.html

Beowulf HOWTO: [1999-05-13]
http://www.sci.usq.edu.au/staff/jacek/beowulf/BDP/HOWTO/

http://beowulf.gsfc.nasa.gov/howto/howto.html

http://lcdx00.wm.lc.ehu.es/~svet/beowulf/howto.html

http://www.cacr.caltech.edu/research/beowulf/tutorial/beosoft/

http://smile.cpe.ku.ac.th/smile/beotalk/index.htm

http://smile.cpe.ku.ac.th/beowulf/index.html

http://www.sci.usq.edu.au/staff/jacek/beowulf/BDP/BIAA-HOWTO/

http://www.linuxfocus.org/Castellano/January2000/article134.html

http://www.clustercomp.org/

http://www.aic.uniovi.es/CyP/indice.htm#ind1

ftp.fciencias.unam.mx

http://www.fis.unipr.it/pub/linux/rpm/redhat-6.2/

http://ftp.gul.uc3m.es/gul/raxi/curso/node15.html

http://www.lafacu.com/apuntes/informatica/sist_oper_unix2/default.htm

http://spisa.act.uji.es/~peralta/os/

http://wwwest.uniandes.edu.co/~ni-giral/llinux/node2.html

http://perlis.fciencias.unam.mx/~atellez/LinuxOS/

HOWTOs

Jacek Radajewski; Douglas Eadline.
Beowulf howto.
1.1.1, 1998.

Beowulf Instalation and Administration HOWTO
Jacek Radajewski; Douglas Eadline
1999

Apéndices

APÉNDICE A. PROGRAMA SUMA DE MATRICES

Para la realización del programa siguiente se tomó como base el de multiplicación de matrices, que podemos encontrar en el capítulo IV de este trabajo.

La suma de matrices es más sencilla, sólo asignamos a cada proceso un número proporcional tanto de la matriz A como de la matriz B, ya que ahora las operaciones son elemento a elemento.

```
/*
**************************************************************************
PVM suma de Matrices A y B
Programa  Maestro
ARCHIVO: pvm_sm.maestro.c
DESCRIPCION: En este codigo el programa maestro actua como el padre que crea
(spawned) (NPROC numero de procesos) las tareas que van  ha realizar los trabajadores

La primera tarea  o proceso trabajador es creada en una maquina especifica.

**************************************************************************
*/
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include "pvm3.h"              /* PVM version 3.0 include file */

#define NPROC 5                      /* numero de procesos o tareas
trabajadores a crear (spawned) */
#define NRA 5                  /* numero de renglones en la
matriz A */
#define NCA 5                  /* numero de columnas en la matriz A*/

main() {
int maestro_tid;                     /* identificadir de la tarea maestro */
int trabajador_tids[NPROC];          /* arreglo de identificadores de trabajadores */
int tipo_mensaje;                    /* tipo mensaje PVM */
int renglones;                 /* numero renglones de la matriz A a enviar  a cada trabajador
*/
int div_renglones,extra,offset;     /* usados para determinar los renglones mandados a cada
trabajador */
int res,codigo_resultado,i=0,j;      /* otros */
double a[NRA][NCA];            /* matriz  A */
double b[NRA][NCA];            /* matriz B */
double c[NRA][NCA];            /* matriz C  */
char este_host[35];            /* nombre del host seleccionado */

/* enrrola esta tarea en PVM*/

maestro_tid = pvm_mytid();

/* La tarea maestro crea  las tareas trabajadores llamando a pvm_spawn
Los identificadores unicos de las tareas  son almacenados en el arreglo trabajador_tids
La primera tarea trabajador ( o proceso) es creada en una maquina especifica.
```

```
El codigo retornado  nos dice el numero de tarea que se ha creado satisfactoriamente*/

for(i=0;i<NPROC;i++) {
  if (i==0) {
     printf ("Entra nombre del host seleccionado: ");
     scanf("%s",&este_host);
     res=pvm_spawn("./pvm_sm.worker", NULL, PvmTaskHost,este_host,1,&trabajador_tids[0]);
          /* asegura que el proceso sea creado exitosamente*/
printf("res%d",res);
if (res != 1)
                    {
                    pvm_exit();
                    return -1;
                    }

    }/*primer if*/
  else {
     res=pvm_spawn("./pvm_sm.esclavo", NULL, PvmTaskDefault,"", 1,&trabajador_tids[i]);
                    /* asegura que el proceso sea creado exitosamente*/
          if (res != 1)
                    {
                    pvm_exit();
                    return -1;
                    }/*segundo if*/

          }/*else*/
  }/*for */

/* inicializa  A y B */

for(i=0; i < NRA; i++){
   for (j=0; j < NCA; j++){
      a[i][j]= i+j;
                              }
                    }


for(i=0;i < NRA; i++){
printf("\n");
for(j=0;j < NCA ; j++){
  printf(" %6.2f", a[i][j]);
                              }
                         }
printf("\n");

getchar();
getchar();

for (i=0; i<NRA; i++){
   for (j=0; j<NCA; j++){
      b[i][j]= i*j;
                              }
                         }
for(i=0;i<NRA;i++){
printf("\n");
for(j=0;j < NCA;j++)
printf(" %6.2f  ",b[i][j]);
}
```

```
printf("\n");

div_renglones = NRA/NPROC;
extra = NRA%NPROC;
offset = 0;
tipo_mensaje = 1;

/* manda los datos a las tareas trabajadortes */
for(i=0;i<NPROC;i++) {                    /* para cada una de las tareas */

 renglones = (i < extra) ? div_renglones+1 : div_renglones;          /*decision de numero de
renglones a mandar de la matriz A */

/* si i < extra renglones = div_renglones+1 en caso contrario renglones = div_renglones */


  /* La proxima llamada inicializa el bufer de envio y especifica que el formato de datos a enviar
sera el default  XDR */
  /* significa que la conversion es solo en ambientes heterogeneos */

codigo_resultado= pvm_initsend(PvmDataDefault);
  /* las  4 llamadas siguientes empaquetan los valores dentro el bufer de envio */
  codigo_resultado=pvm_pkint(&offset,1,1);          /* empieza posicion  en la
matriz */
  codigo_resultado=pvm_pkint(&renglones,1,1);              /* numero de renglones
de la matriz A a enviar */
  codigo_resultado=pvm_pkdouble(&a[offset][0],renglones*NCA,1); /* algunos
renglones de A
, esta linea nos dice que la cantidad de elementos a enviar es  igual arenglones*NCA, y los datos
a enviar empiezan a partir de la pocision a[offset][0]  */
  codigo_resultado=pvm_pkdouble(&b[offset][0],renglones*NCA,1);          /* se
empaqueta toda la
matriz B */
 codigo_resultado=pvm_send(trabajador_tids[i],tipo_mensaje);  /* manda
el contenido del bufer  de envia  a la tarea trabajador */
 offset = offset + renglones;  /* desplazamiento derenglones en la matriz A*/
 }/*fin for*/

/* espera resultados de todas las tareas trabajadores */

 tipo_mensaje = 2;                                  /* pone tipo de mensaje, ahora ya es de
recivir */
 for(i=0;i<NPROC;i++) {
 /* lo realiza para cada uno de los trabajadores */
 codigo_resultado=pvm_recv(-1,tipo_mensaje);        /*recive los mensajes del trabajador*/
 codigo_resultado=pvm_upkint(&offset,1,1);          /* empieza posicion en la
 matriz */
 codigo_resultado=pvm_upkint(&renglones,1,1);             /* numero de renglones
 enviados */
 codigo_resultado=pvm_upkdouble(&c[offset][0],renglones*NCA,1); /*renglones
 para matriz C */
                        }/* fin for*/
 /* imprime resultados */
 for(i=0;i<NRA;i++) {
   printf("\n");
   for(j=0;j<NCA;j++)
     printf(" %6.2f   ",c[i][j]);

     }
```

```
printf("\n");

/* tareas que existen en PVM */
codigo_resultado=pvm_exit();
}


/*****************************************************************************
**
* PVM suma de matrices - Version en C
* Programa trabajador (Worker program)
* archivo: pvm_mm.worker1.c
* DESCRIPCION: ver pvm_sm.master.c
* VERSION PVM : 3.x
* ULTIMA REVISION : 27/04/01
*****************************************************************************
*/

#include <stdio.h>
#include <malloc.h>
#include "pvm3.h"      /* Cabecera de PVM version 3.0  */

#define NRA 5         /* Numero de renglones en la matriz  A */
#define NCA 5         /* Numero de columnas en la Matriz A */
/*trabajador_tid
maestro_tid
tipo_mensaje
renglones
codigo_resultado*/

main() {
int   trabajador_tid;           /* identificador de tarea de PVM de este programa trabajador */
int   maestro_tid;            /*  identificador de tarea de PVM del proceso padre del programa
maestro */
int   tipo_mensaje;           /* Tipo de mensaje PVM */
int   renglones;            /* Numero de renglones en la Matriz a enviados al trabajador */
int   offset;          /* Posición inicial en la matriz */
int   codigo_resultado, i, j, k;     /* varios */
double a[NRA][NCA];     /* Matriz A a ser multiplicada */
double b[NRA][NCA];     /* Matriz B a ser multiplicada */
double c[NRA][NCA];     /* Matriz C resultado */

/* incluye en PVM la tarea lenvanta por el trabajador */

trabajador_tid = pvm_mytid();

/* Recibe el mensaje desde el maestro */

tipo_mensaje = 1;                    /* cambia el tipo de mensaje */
maestro_tid = pvm_parent();              /* obtiene el identificador de tarea del proceso
                                          maestro */
codigo_resultado = pvm_recv(maestro_tid, tipo_mensaje);        /* espera a recibir un mensaje
desde el maestro*/
codigo_resultado = pvm_upkint(&offset, 1, 1);      /* Posicion inicial en las matrices  A y C*/
codigo_resultado = pvm_upkint(&renglones, 1, 1);       /* #renglones en la Matriz A enviados
*/
codigo_resultado = pvm_upkdouble(*a, renglones*NCA, 1); /* nuestra parte compartida de la
Matriz A */
codigo_resultado = pvm_upkdouble(*b, renglones*NCA, 1);  /* contenido
```

```
Matriz B*/

printf("trabajador: identificador de tarea = %d recibo %d renglones de A\n", trabajador_tid,
renglones);

/* hace la suma de las matrices */
/*for (k=0; k<NCA; k++)*/
  for (i=0; i<renglones; i++) {
/*    c[i][k] = 0.0; */
    for (j=0; j<NCA; j++)
      c[i][j] = a[i][j] + b[i][j];
  }


/* Arma el mensaje a enviar al proceso maestro */

tipo_mensaje = 2;                          /* cambia el tipo de mensaje */
codigo_resultado = pvm_initsend(PvmDataDefault);  /* inicializa el buffer  de envio */
codigo_resultado = pvm_pkint(&offset, 1, 1);      /* posicion del resultado enviado en la matriz
resultado */

      /* Numero de renglones que se nvian */

codigo_resultado = pvm_pkint(&renglones, 1, 1); codigo_resultado = pvm_pkdouble(*c,
renglones*NCA, 1);

   /* muestra parte del resultado en la Matriz C */

/* enviando al maestro */
codigo_resultado = pvm_send(maestro_tid, tipo_mensaje);

/* saliendo de PVM */
codigo_resultado = pvm_exit();
}
```

APENDICE B. *FOTOGRAFÍAS DEL CLUSTER BEOWULF ETZNA*

En este apéndice se mostrarán algunas fotografías tomadas al Cluster una vez ya construido.

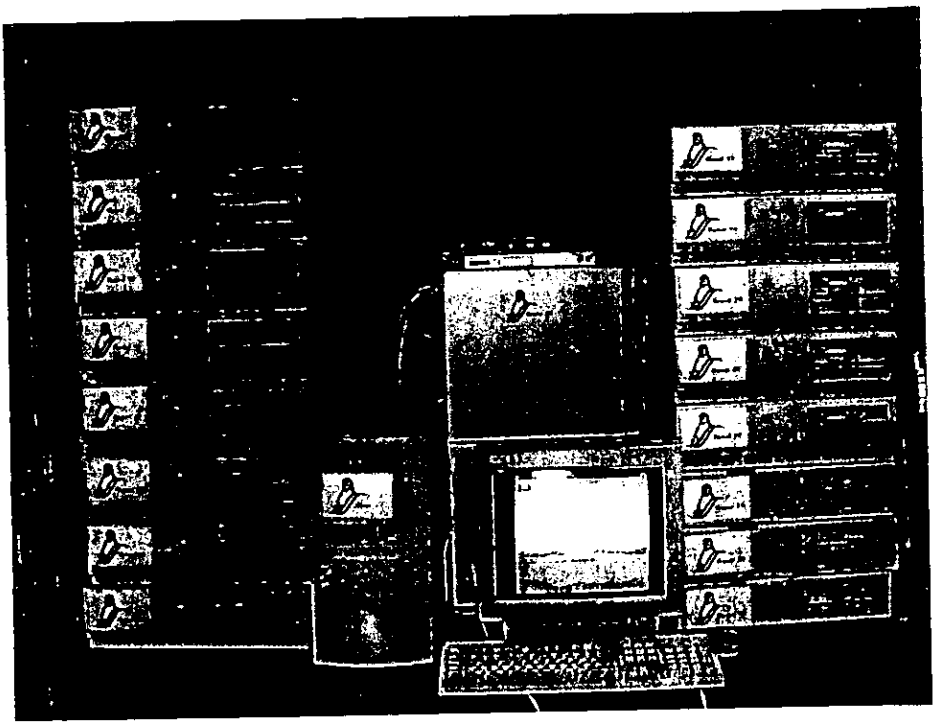


Figura B-1 Vista general del Cluster una vez terminado.

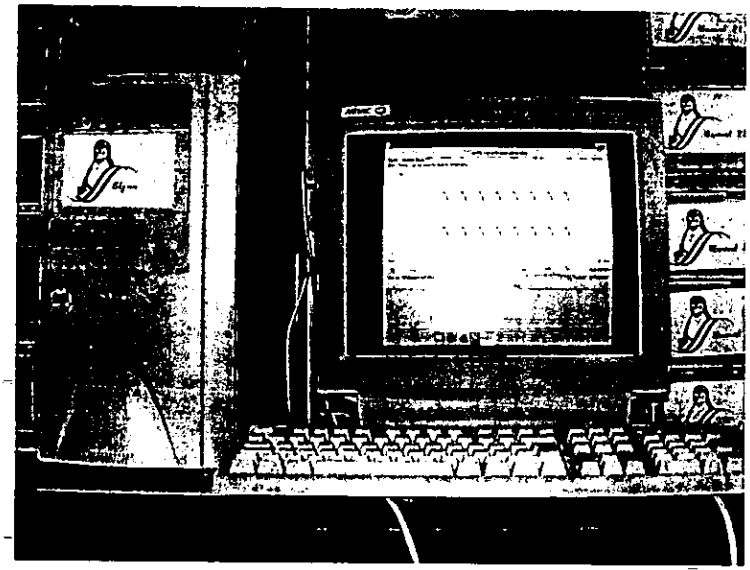


Figura B-2 Máquina servidor (etzna) ejecutando la interfaz gráfica de *PVM (XPVM)*, una vez que todos los nodos han sido agregados.

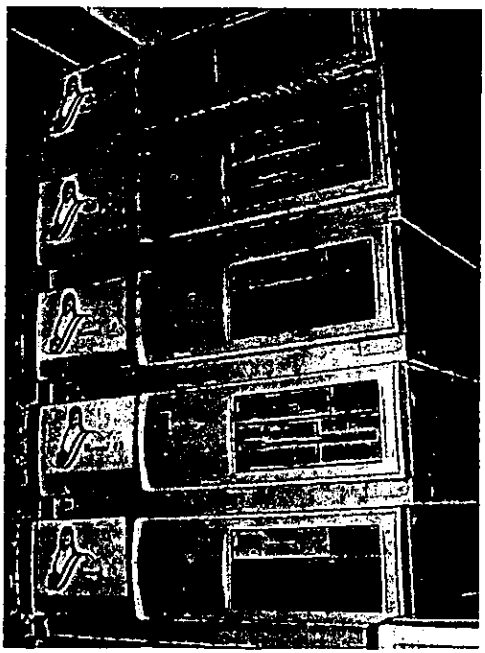


Figura B-3 Vista lateral de una de las torres de PC s.

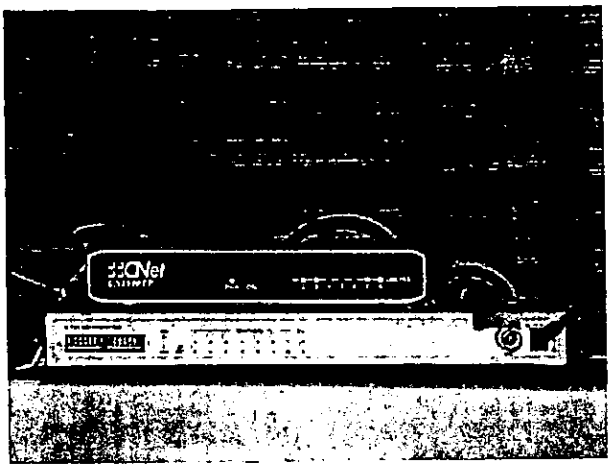


Figura B-4 Hubs (Concentradores) Ethernet utilizados para la interconexión de los nodos y el servidor.

Figura B-5 Vista posterior de los Hubs. Los cables se encuentran etiquetados en ambos extremos para su mejor y más fácil manejo.

Figura B-6 Vista posterior de una de las torres de PC s. Tanto los cables de red como los cables de alimentación se encuentran etiquetados en ambos extremos con el número que corresponde a cada nodo.

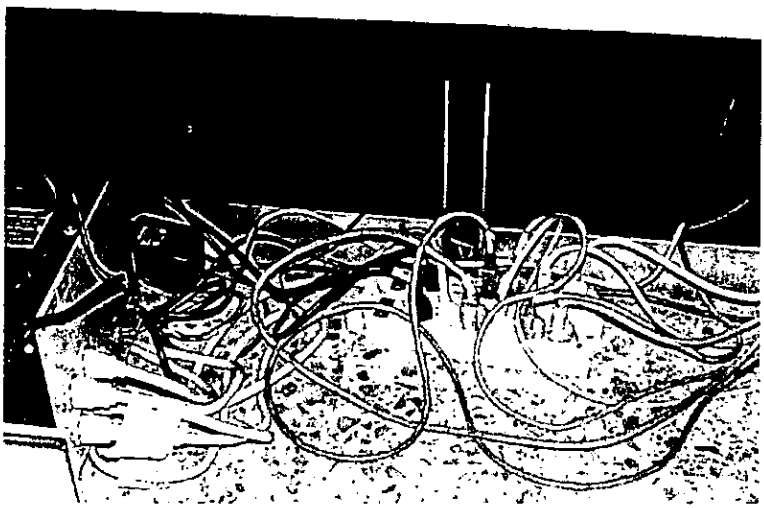


Figura B-7 Conexiones de alimentación de los nodos, servidor y Hubs.

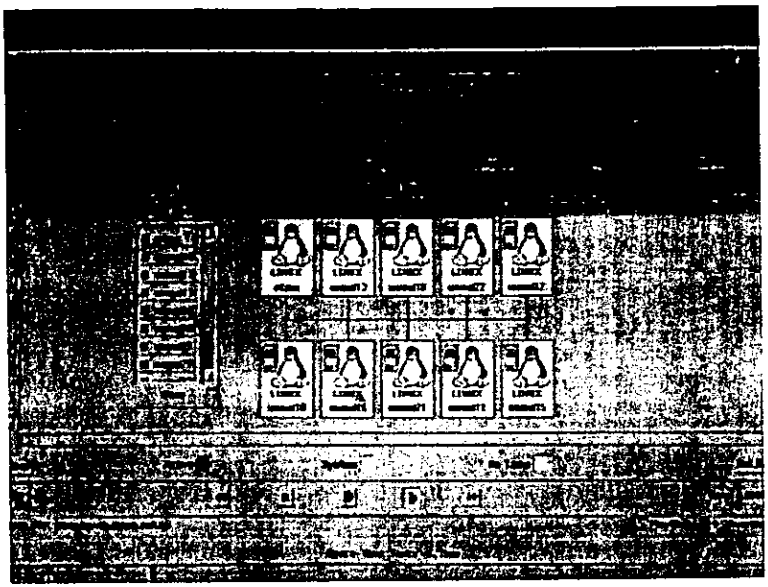


Figura B-8 Vista del *XPVM* al momento de agregar los nodos.