



BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

**PROGRAMACIÓN Y MONITOREO DE
APLICACIONES PARALELAS EN UN
CLUSTER**

TESIS

**QUE PARA OBTENER EL GRADO DE
LICENCIADO EN CIENCIAS DE LA
COMPUTACIÓN**

PRESENTA:

GUADALUPE ESPARRAGOZA MARTÍNEZ

ASESOR:

DR. MANUEL MARTIN ORTÍZ

ENERO, 2008

INDICE

INTRODUCCIÓN.....	4
ARQUITECTURA FUNCIONAL DE COMPUTADORAS Y LENGUAJES DE PROGRAMACIÓN..	6
1.1 TAXONOMIA DE FLYNN.....	6
1.1.1 SISD.....	6
1.1.2 MISD.....	7
1.1.3 SIMD.....	7
1.1.4 MIMD.....	8
1.1.5 Multiprocesadores o Sistemas de Memoria Compartida.....	9
1.1.6 Multicomputadoras o Sistemas de Memoria Distribuida.....	10
1.1.7 Sistemas de Memoria Compartida Distribuida.....	11
1.2 CLASIFICACIÓN DE LOS MULTIPROCESADORES EN FUNCIÓN DE LA ORGANIZACIÓN DE MEMORIA.....	12
1.2.1 UMA.....	12
1.2.2 NUMA.....	13
1.2.3 MPM.....	13
1.3 LENGUAJES DE PROGRAMACIÓN.....	14
1.3.1 Lenguajes Imperativos.....	14
1.3.2 Lenguajes Funcionales.....	15
1.3.3 Lenguajes Lógicos.....	15
1.3.4 Lenguajes Orientados a Objetos.....	16
1.3.5 Lenguajes Concurrentes, Paralelos y Distribuidos.....	17
CLUSTER.....	18
2.1 DEFINICIÓN DE CLUSTER.....	18
2.2 COMPONENTES DE UN CLUSTER.....	18
2.2.1 Nodos.....	18
2.2.2 Sistema Operativo.....	18
2.2.3 Conexiones de Red.....	19
2.2.4 Middleware.....	19
2.2.5 Protocolos de Comunicación.....	20
2.3 CARACTERÍSTICAS GENERALES DE UN CLUSTER.....	20
2.3.1 Software a nivel de aplicación.....	20
2.3.2 Software a nivel de sistema.....	20
2.3.3 Acoplamiento de un cluster.....	20
2.3.4 Control del Cluster.....	22
2.3.5 Escalado vertical.....	22
2.3.6 Escalado horizontal.....	22
2.3.7 Disponibilidad.....	22
2.3.8 Fiabilidad.....	22
2.3.9 Tolerancia a fallos.....	23
2.4 TIPOS DE CLUSTER SEGÚN EL TIPO DE NODO.....	23
2.4.1 Homogéneo.....	23
2.4.2 Semi-homogéneo.....	23
2.4.3 Heterogéneo.....	23
2.5 TIPOS DE CLUSTER SEGÚN SU APLICABILIDAD.....	23
2.5.1 Cluster de Alta Disponibilidad.....	23
2.5.2 Cluster de Alto Rendimiento.....	24
2.5.3 Clusters de Alta Confiabilidad.....	26
2.6 FUNCIONAMIENTO DE UN CLUSTER.....	26
2.7 CLUSTER BEOWULF.....	27
2.8 BENEFICIOS DE LOS CLUSTERS.....	29

PARALELISMO	30
3.1 PROGRAMACIÓN PARALELA: LIBRERIAS DE PASO DE MENSAJES	30
3.1.1 Paso de mensajes síncrono	30
3.1.2 Paso de mensajes asíncrono	31
3.2 MPI	31
3.3 PVM.....	33
3.4 DIFERENCIAS ENTRE PVM Y MPI	35
3.5 VENTAJAS DE PVM SOBRE SUPERCOMPUTADORAS	36
3.6 GRANULARIDAD DE UN PROGRAMA	37
3.7 TÉCNICAS BÁSICAS DE PROGRAMACIÓN CON PVM.....	38
3.7.1 Modelo Crowd	38
3.7.2 Modelo de Árbol	39
3.7.3 Modelo Híbrido.....	39
3.8 FUNCIONAMIENTO DE PVM.....	39
3.9 FUNCIONAMIENTO DE XPVM.....	42
3.10 INSTALACIÓN Y CONFIGURACIÓN DE PVM.....	43
3.10.1 Pasos para la instalación de PVM.....	44
3.10.2 Configuración del Entorno PVM y Compilación de Ejemplos	44
3.11 INSTALACIÓN Y CONFIGURACIÓN DE XPVM	47
3.12 COMANDOS FUNDAMENTALES DEL PROGRAMA PVM.....	49
3.13 INTERFAZ DE USUARIO DE PVM.....	50
APLICACIONES	61
4.1 INFIERNO.....	62
4.1.1 Maestro	62
4.1.2 Esclavo.....	66
4.2 SOPA DE LETRAS	69
4.2.1 Maestro.....	69
4.2.2 Esclavo.....	72
4.3 PRUEBAS Y RESULTADOS	74
4.3.1 Infierno.....	7574
4.3.2 Sopa de letras.....	8884
CONCLUSIONES.....	9389
Trabajo a futuro.....	9490
APENDICE A	9591
LINUX.....	9591
Comandos básicos de Linux.....	9793
Fedora.....	9995
APENDICE B	10197
LENGUAJE C	10197
Elementos generales de un programa en C	10298
BIBLIOGRAFIA	104100

INTRODUCCIÓN

Los clusters de computadoras juegan hoy en día un papel muy importante en la solución de problemas de las ciencias, las ingenierías, en el desarrollo y ejecución de muchas aplicaciones comerciales, debido a que son máquinas poderosas que pueden desarrollar varios miles de millones de operaciones por segundo.

La tecnología de clusters ha evolucionado en apoyo de actividades que van desde aplicaciones de supercómputo y software de misiones críticas, servidores Web y comercio electrónico, hasta bases de datos de alto rendimiento, entre otros usos.

Por su alto costo, el uso de supercomputadoras verdaderas está limitado a organismos gubernamentales, militares y grandes centros de investigación, en donde tienen aplicaciones científicas, como en la simulación de procesos naturales (previsión del tiempo, análisis de cambios climáticos, entre otros procesos), modelaje molecular, simulaciones físicas como túneles de viento, criptoanálisis, etc.

En los últimos años, el personal académico de diversas universidades y centros de investigación se han dado a la tarea de construir sus propias supercomputadoras conectando computadoras personales y desarrollando software para dichos sistemas, ya que representan una solución de bajo costo.

Por otro lado, la programación paralela es una técnica de programación en que se dividen grandes problemas en componentes más pequeños, llamados tareas o procesos, que pueden resolverse en paralelo basándose en la ejecución simultánea de dichos procesos, bien sea en una misma computadora (con uno o varios procesadores) o en un cluster de computadoras, en cuyo caso se denomina computación distribuida. Debido a que la programación paralela es el resultado de la necesidad de contar con una mayor velocidad en cuanto a la ejecución de procesos, la paralelización de programas provee resultados en una menor cantidad de tiempo.

Es por tal motivo que, este trabajo de tesis presenta un análisis de la operabilidad y funcionamiento de un cluster bajo PVM, adquirido en abril de 2004 por la Facultad de Ciencias de la Computación de la Benemérita Universidad Autónoma de Puebla, basado en componentes hardware de bajo costo y software libre.

Esta tesis pretende introducir al lector en las herramientas de software que pueden ser utilizadas para la explotación de sus recursos. Específicamente hablando de LINUX y PVM.

Cabe señalar que se proporciona información resumida del cómo han surgido los clusters, según su aplicabilidad los distintos servicios que ofrece, además de su funcionamiento, arquitectura, sistema operativo que lo compone, sus diferentes formas de programación, entre otras cosas.

En particular, la documentación es desarrollada en torno a la programación orientada a aplicaciones paralelas basada en paso de mensajes a través de las librerías de PVM (Pa-

rallel Virtual Machine) Máquina Virtual Paralela; ésta es una interfaz de paso de mensajes la cual representa la disponibilidad de un software altamente eficiente y portable para satisfacer las necesidades actuales en la computación de alto rendimiento.

Además de tratar temas tales como su instalación, programación, compilación de programas, así como presentar la ejecución de algunos ejercicios asociados con la programación paralela, de modo que se pueda observar el funcionamiento de PVM sobre un cluster para obtener el incremento de rendimiento o ganancia de velocidad que se consigue añadiendo nuevos nodos a la resolución del problema y esto a su vez, sirva cómo base para la ejecución y resolución de problemas de mayor complejidad.

CAPÍTULO I

ARQUITECTURA FUNCIONAL DE COMPUTADORAS Y LENGUAJES DE PROGRAMACIÓN

El concepto de Arquitectura de computadoras se define como el funcionamiento, estructura y diseño de computadoras. Dentro de este término se engloban aspectos como formato de instrucción, modo de direccionamiento, conjunto de instrucciones, entre otros.

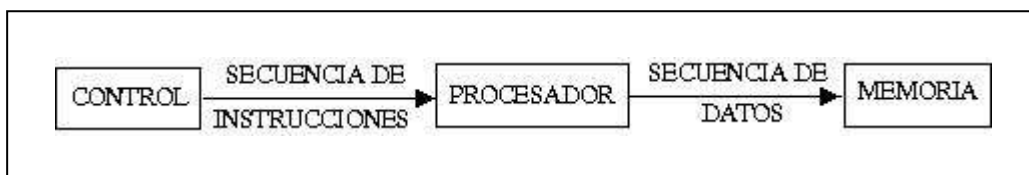
La mayoría de las computadoras que se utilizan actualmente tienen arquitectura John Von Neumann (arquitectura secuencial) para el procesamiento de la información; esto es que una instrucción o tarea debe ser completada antes de iniciar la siguiente. En esta arquitectura se utiliza el mismo dispositivo de almacenamiento tanto para las instrucciones como para los datos.

1.1 TAXONOMIA DE FLYNN

A diferencia de Neumann, Michel J. Flynn considera la organización de un sistema de computadora mediante la cantidad de instrucciones y unidades de datos que se manipulan en forma simultánea. La operación normal de una computadora es recuperar instrucciones de la memoria y ejecutarlas en el procesador. La secuencia de instrucciones leídas de la memoria constituye un flujo de instrucciones. Las operaciones ejecutadas sobre los datos en el procesador constituyen un flujo de datos.

El procesamiento paralelo puede ocurrir en el flujo de instrucciones, en el flujo de datos o en ambos. La clasificación de Flynn divide a las computadoras en cuatro grupos principales de la manera siguiente:

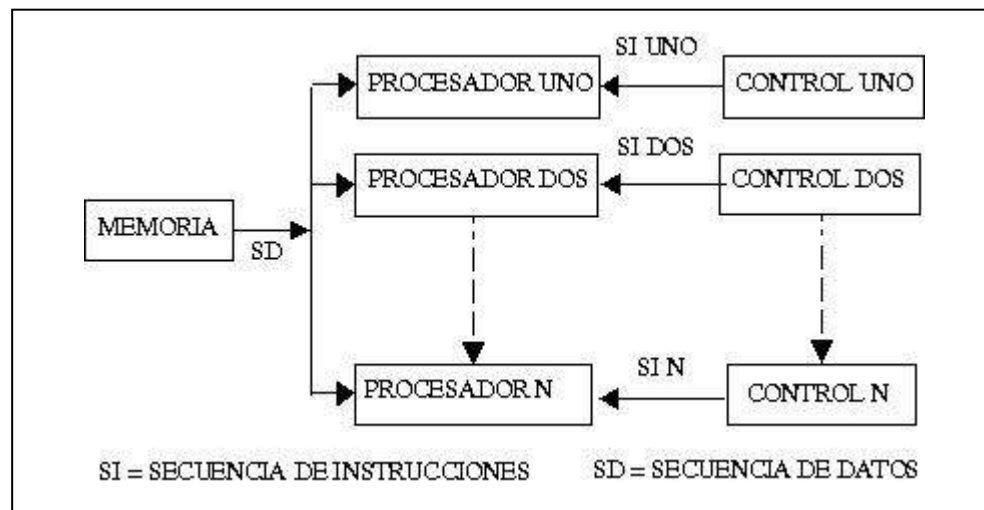
1.1.1 SISD (Single Instruction Single Data). Constituye el modelo clásico de arquitectura de computadoras en el que se basaron los diseños a lo largo de muchos años. Representa la organización de una computadora única que contiene una unidad de control, una unidad de procesador y una unidad de memoria. Las instrucciones se ejecutan en forma secuencial y el sistema puede tener o no tener posibilidades de procesamiento paralelo. En este caso el procesamiento paralelo puede lograrse mediante unidades funcionales múltiples o mediante una arquitectura paralela.



Modelo **SISD** (Single Instruction Single Data).

1.1.2 MISD (Multiple Instruction Single Data). Esta categoría corresponde a configuraciones en las que un tren de unidades de procesamiento se surte por uno de sus extremos de un dato extraído de memoria y devuelve un resultado a la misma memoria por el extremo opuesto, en tanto cada unidad del tren toma por entrada al dato procesado por la unidad anterior y da por salida el que será tomado como entrada por la unidad siguiente. La característica (MI) está en el hecho de que cada unidad de procesamiento va ejecutando una operación distinta sobre un único dato (SD) que va avanzando hasta convertirse en el resultado.

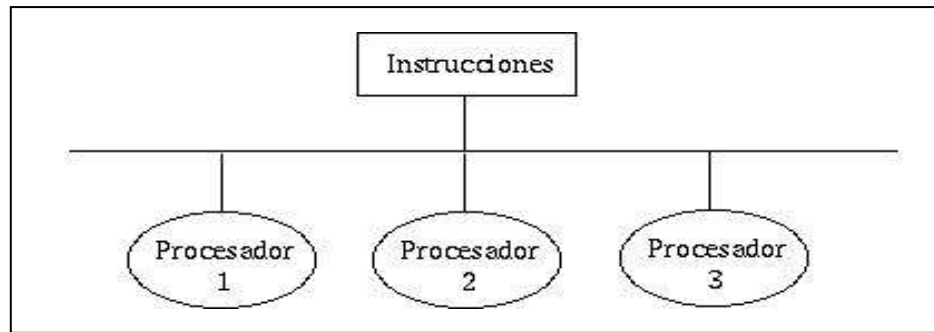
En este tipo de modelos el paralelismo es alcanzado dejando que los procesadores realicen diferentes cosas al mismo tiempo en el mismo dato.



Modela **MISD** (Multiple instruction Single Data).

Las máquinas MISD son útiles en cómputos donde la misma entrada esta sujeta a diferentes operaciones.

1.1.3 SIMD (Single Instruction Multiple Data). Aquí se consideran los sistemas que a partir de una única instrucción son capaces de procesar un conjunto o array de datos en forma simultánea, con la particularidad de ejecutar la misma instrucción sobre cada uno de sus elementos. Estos sistemas se componen de varias unidades funcionales idénticas (Unidades Aritmético-Lógicas, Registros de Direcciones de Memoria, etc.) comandadas todas ellas desde una única unidad de control encargada de recibir y procesar cada instrucción a ejecutar. Podría considerarse que, aunque en forma restringida, esta arquitectura incorpora en cierta medida el paralelismo desde el momento que es capaz de ejecutar "algunas tareas" (no "cualquier tarea") en forma simultánea, concretamente puede realizar N operaciones O, sobre N datos D, al mismo tiempo. Para que esta modalidad pueda ser implementada, la arquitectura correspondiente deberá combinar necesariamente las prestaciones SIMD con las SISD. El tipo de memoria que estos sistemas utilizan es distribuida.



Modelo **SIMD**(Single Instruction Multiple Data).

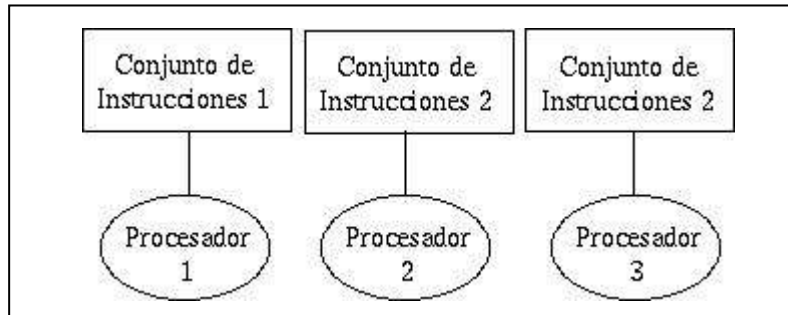
1.1.4 MIMD (Multiple Instruction Multiple Data). Este tipo de computadora es paralela al igual que las SIMD, la diferencia con estos sistemas es que MIMD es asíncrono. No tiene un reloj central. Cada procesador en un sistema MIMD puede ejecutar su propia secuencia de instrucciones y tener sus propios datos. Esta característica es la más general y poderosa de esta clasificación.

En este modelo cada procesador actúa en forma independiente, realizando cada uno de ellos un trabajo específico sobre una parte de la tarea completa. Esto es lo que se denomina descomposición de control, donde cada procesador puede estar ejecutando un programa distinto. Mucho más habitual es la descomposición de datos en la que los datos del problema se reparten entre los distintos procesadores y todos ejecutan el mismo programa sobre su fragmento de datos.

Esta versión restringida del modelo se denomina SPMD (Single Program, Multiple Data), donde la diferencia con el modelo SIMD reside en que cada procesador que ejecuta un programa SPMD puede fluir a través de ramas de código diferentes, resultando un modelo mucho más flexible. Para lograr este efecto, se debe recurrir a la programación de hebras (threads) ^[1]. Esencialmente en el modelo MIMD los datos se intercambian entre procesadores mediante memoria compartida o por medio del paso de mensajes. Los procesadores de los sistemas con memoria compartida comparten físicamente la memoria, accediendo todos al mismo espacio de direcciones. A la combinación de MIMD y memoria compartida se le denomina SMP (Symmetric Multi-Processing). En el caso de utilizar el paso de mensajes, los sistemas están compuestos por procesadores que disponen cada uno de su propia memoria independiente de la del resto. Existen herramientas de software que permiten programar una arquitectura de memoria distribuida como si fuera de memoria compartida, sin preocuparse de en qué procesador reside cada dato. A este modelo se lo denomina de memoria compartida virtual. Los sistemas con memoria distribuida pueden ser una única computadora con múltiples CPUs comunicadas por un bus de datos o bien múltiples computadoras, cada una con su procesador, enlazadas por una red de datos. Más comúnmente se implementan los sistemas constituidos por varias máquinas conectadas entre sí, lo que de forma genérica se conoce como cluster (grupo) de estaciones de trabajo. Las implementaciones de

¹ Una **hebra** (o *proceso ligero*) es la unidad básica de utilización de la CPU (unidad de expedición). Consta de: Contador de programa, Conjunto de registros, Espacio de pila, Estado.

las arquitecturas MIMD suelen constituirse por procesadores potentes y grandes cantidades de memoria.



Modelo **SIMD**(Single Instruction Multiple Data).

Los sistemas MIMD se clasifican en:

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

1.1.5 Multiprocesadores o Sistemas de Memoria Compartida. En este tipo de sistemas cada procesador tiene acceso a toda la memoria, es decir hay un espacio de direccionamiento compartido. Se tienen tiempos de acceso a memoria uniformes ya que todos los procesadores se encuentran igualmente comunicados con la memoria principal y las lecturas y escrituras de todos los procesadores tienen exactamente las mismas latencias; y además el acceso a memoria es por medio de un ducto común. En esta configuración, debe asegurarse que los procesadores no tengan acceso simultáneamente a regiones de memoria de una manera en la que pueda ocurrir algún error. No utiliza ninguna librería y solo hace uso de compiladores.

Desventajas:

- El acceso simultáneo a memoria es un problema.
- Poca escalabilidad de procesadores, debido a que se puede generar un cuello de botella al incrementar el número de CPU's.

En computadoras vectoriales como Crays,

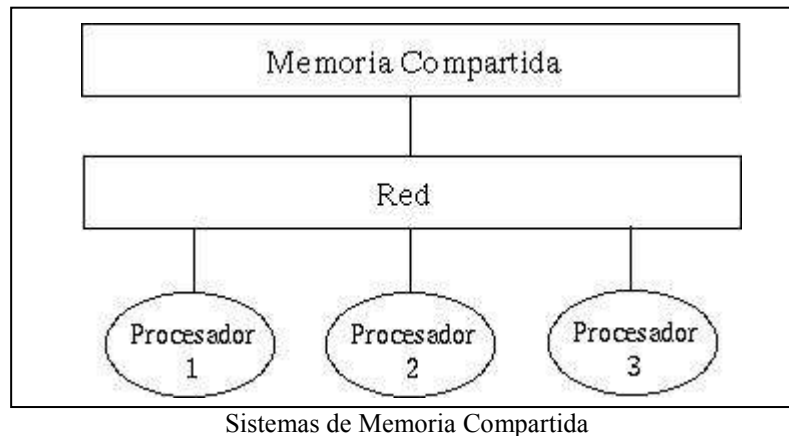
Todos los CPUs tienen un camino libre a la memoria.
No hay interferencia entre CPUs.

La razón principal por el alto precio de Cray es la memoria.

Ventaja:

- La facilidad de la programación. Es mucho más fácil programar en estos sistemas que en sistemas de memoria distribuida.

Las computadoras MIMD con memoria compartida son sistemas conocidos como de multi-procesamiento simétrico (SMP) donde múltiples procesadores comparten un mismo sistema operativo y memoria. Otro término con que se le conoce es máquinas firmemente juntas o de multiprocesadores. Ejemplos son: SGI/Cray Power Challenge, SGI/Cray C90, SGI/Onyx, ENCORE, MULTIMAX, SEQUENT y BALANCE, entre otras.



1.1.6 Multicomputadoras o Sistemas de Memoria Distribuida. Estos sistemas tienen su propia memoria local. Los procesadores pueden compartir información solamente enviando mensajes, es decir, si un procesador requiere los datos contenidos en la memoria de otro procesador, deberá enviar un mensaje solicitándolos. Esta comunicación se le conoce como Paso de Mensajes. Las librerías de paso de mensajes que utilizan son: PVM, MPI o SHMEM.

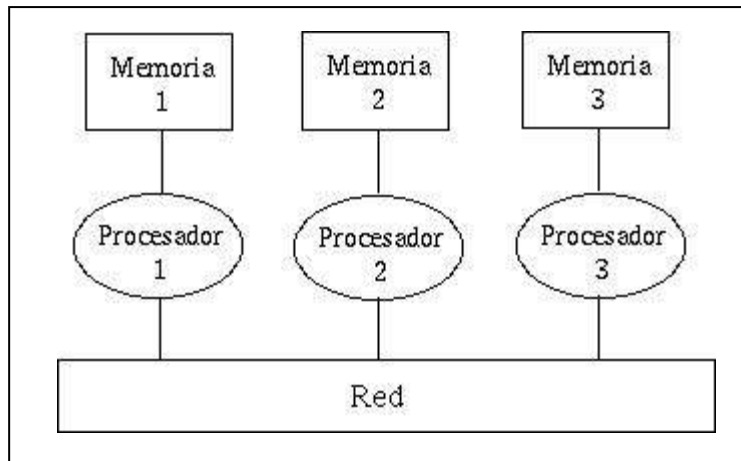
Desventajas:

- El acceso remoto a memoria es lento.
- La programación puede ser complicada.

Ventajas:

- La escalabilidad. Las computadoras con sistemas de memoria distribuida son fáciles de escalar, mientras que la demanda de los recursos crece, se puede agregar más memoria y procesadores.

Las computadoras MIMD de memoria distribuida son conocidas como *sistemas de procesamiento en paralelo masivo* (MPP) donde múltiples procesadores trabajan en diferentes partes de un programa, usando su propio sistema operativo y memoria. Además se les llama multicomputadoras, máquinas libremente juntas o *cluster*. Algunos ejemplos de este tipo de máquinas son IBM SP2 y SGI/Cray T3D/T3E.



Sistemas de Memoria Distribuida.

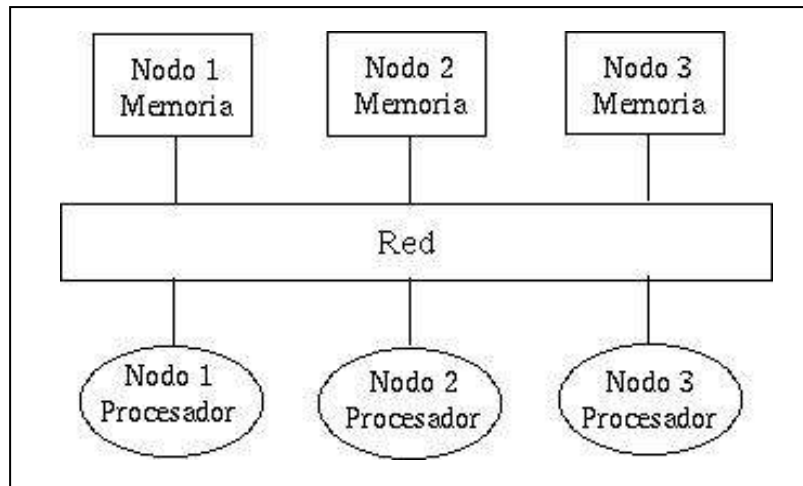
1.1.7 Sistemas de Memoria Compartida Distribuida. Es un cluster o una partición de procesadores que tienen acceso a una memoria compartida común pero sin un canal compartido. Esto es, físicamente cada procesador posee su memoria local y se interconecta con otros procesadores por medio de un dispositivo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

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

Algunos ejemplos de este tipo de sistemas son HP/Convex SPP-2000 y SGI/Cray Origin2000.

Ventajas:

- Presenta escalabilidad como en los sistemas de memoria distribuida.
- Es fácil de programar como en los sistemas de memoria compartida.
- No existe el cuello de botella que se puede dar en máquinas de sólo memoria compartida.



Sistemas de Memoria Compartida Distribuida.

1.2 CLASIFICACIÓN DE LOS MULTIPROCESADORES EN FUNCIÓN DE LA ORGANIZACIÓN DE MEMORIA

En general, vamos a distinguir tres tipos de organización de la memoria en sistemas multiprocesador. Aunque los sistemas comerciales van a emplear modelos mixtos, esta clasificación sencilla, ayuda suficientemente a comprender las implicaciones que se derivan de cada modelo.

Esta clasificación atiende al criterio del tiempo de acceso a la memoria desde los procesadores, existen dos posibles tipos de sistemas: aquellos en que el acceso a la memoria se produce en un tiempo (latencia) fijo, denominados sistemas UMA (Uniform Memory Access), y aquellos en que estos tiempos no son homogéneos sino que varían en función de qué procesador y qué porción de memoria consideramos y que se denominan sistemas NUMA (Non Uniform Memory Access)

1.2.1 UMA (Uniform Memory Access). En estos sistemas la memoria física es global para todos los procesadores y los accesos tienen los mismos retardos para todos los nodos, es decir, la memoria es uniformemente compartida. Requiere un hardware sencillo aunque su principal desventaja es el cuello de botella en que se convierte el acceso a memoria y los problemas de escalabilidad que ello supone.

Los multiprocesadores son sistemas fuertemente acoplados (tightly-coupled), dado el alto grado de compartición de los recursos (hardware o software) y el alto nivel de interacción entre procesadores, lo que hace que un procesador dependa de lo que hace otro.

Este modelo es conveniente para aplicaciones de propósito general y de tiempo compartido por varios usuarios.

Algunas categorías de sistemas UMA son:

Sistema Simétrico. Cuando todos los procesadores tienen el mismo tiempo de acceso a todos los componentes del sistema (incluidos los periféricos), reciben el nombre de sistemas multiprocesador simétrico.

Los procesadores tienen el mismo dominio (prioridad) sobre los periféricos y cada procesador tiene la misma capacidad para procesar.

Sistema Asimétrico. Los sistemas multiprocesador asimétrico, son sistemas con procesadores maestros y procesadores esclavos, en donde sólo los primeros pueden ejecutar aplicaciones y dónde en tiempo de acceso para diferentes procesadores no es el mismo. Los procesadores esclavos (attached) ejecutan código usuario bajo la supervisión del maestro, por lo tanto cuando una aplicación es ejecutada en un procesador maestro dispondrá de una cierta prioridad.

1.2.2 NUMA (Non Uniform Memory Access). Un sistema multiprocesador NUMA es un sistema de memoria compartida donde el tiempo de acceso varía según donde se encuentre localizado el acceso.

El acceso a memoria, por tanto, no es uniforme para diferentes procesadores. Existen memorias locales asociadas a cada procesador y estos pueden acceder a datos de su memoria local de una manera más rápida que a las memorias de otros procesadores, debido a que primero debe aceptarse dicho acceso por el procesador del que depende el módulo de memoria local.

Todas las memorias locales conforman la memoria global compartida y físicamente distribuida y accesible por todos los procesadores. El hardware de la red es complejo ya que se van a necesitar interconexiones especiales que minimicen los retardos en el acceso a memoria remota. Pero, por otra parte, se mejora la escalabilidad.

Cluster Jerárquico. Otro modelo NUMA que nace como la mezcla del modelo UMA y el modelo NUMA anterior, es el cluster jerárquico en el que se combinan las memorias locales y las globales obteniendo una cierta escalabilidad del sistema.

Los procesadores aparecen distribuidos en clusters (1 sistema UMA o un 1 sistema NUMA). Estos clusters están conectados a la memoria global compartida. El sistema en su totalidad es un sistema NUMA, ya que el acceso a memoria es no uniforme por parte de los clusters. La ventaja de estos sistemas con respecto a los sistemas UMA, es que el acceso a memoria local es mucho más rápido.

1.2.3 MPM (Message Passing Machines). En este caso la memoria además de pertenecer a cada procesador, va a ser vista mediante un mapa de direccionamiento distribuido. Cada procesador es un computador independiente del resto, esto implica que la comunicación entre procesadores sea mediante el envío y recepción de mensajes. Se mejora la escalabilidad notablemente, siendo la red la parte más compleja de escalar. El overhead de comunicación, en cambio, es el más alto. Las arquitecturas MPM son las que se relacionan con los clusters.

Ventajas:

- Adaptación a las Arquitecturas. El modelo de paso de mensajes encaja bien en los supercomputadores paralelos y en los Clusters de Workstations.
- Funcionalidad. El modelo de paso de mensajes oferta un conjunto completo de funciones para expresar algoritmos paralelos, proporcionando un control no disponible en otros modelos.
- Rendimiento y Escalabilidad. La localidad en el acceso a los datos proporciona un alto rendimiento y un elevado grado de escalabilidad

Desventajas:

- Responsabilidad en manos del programador. Debe expresar de forma explícita el esquema de distribución de datos, así como todas las comunicaciones entre procesos y las sincronizaciones. Además, debe resolver todas las dependencias de datos y evitar las condiciones de bloqueo

Algunos ejemplos de librerías de Paso de Mensajes que son paquetes de dominio público y que no están dirigidos a una máquina concreta son: PVM, MPI, PICTL, PARMACS, P4, etc.

1.3 LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación es una técnica estándar de comunicación que permite expresar las instrucciones que han de ser ejecutadas en una computadora. Consiste en un conjunto de reglas sintácticas y semánticas que definen un lenguaje informático.

Un lenguaje de programación permite a un programador especificar de manera precisa: sobre qué datos una computadora debe operar, cómo deben ser estos almacenados y transmitidos y qué acciones debe tomar bajo una variada gama de circunstancias. Todo esto, a través de un lenguaje que intenta estar relativamente próximo al lenguaje humano o natural, tal como sucede con el lenguaje Léxico.

En esta sección se expone un breve panorama de los más importantes tipos de lenguajes de programación.

1.3.1 Lenguajes Imperativos. En este tipo de lenguajes, cuyo origen está ligado a la propia arquitectura de Von Neumann, la arquitectura consta de una secuencia de celdas, llamadas memoria, en la cual se pueden guardar en forma codificada, lo mismo datos que instrucciones; y de un procesador, el cual es capaz de ejecutar de manera secuencial una serie de operaciones, principalmente aritméticas y booleanas, llamadas comandos. En general, un lenguaje imperativo ofrece al programador conceptos que se traducen de forma natural al modelo de la máquina.

Los lenguajes imperativos más destacados de la historia han sido: FORTRAN, Algol, Pascal, C, Modula-2, Ada.

El programador, al utilizar un lenguaje imperativo, por lo general tiene que traducir la solución abstracta del problema a términos muy primitivos, cercanos a la máquina. La distancia entre el nivel del razonamiento humano y lo expresable por los lenguajes imperativos causa que sus programas sean más "comprensibles" para la máquina que para el hombre. Esta desventaja para nosotros, reflejada en la dificultad que tenemos al construir programas en un lenguaje imperativo, se vuelve una ventaja en el momento de la generación del código. El programa está expresado en términos tan cercanos a la máquina, que el código generado es relativamente parecido al programa original, lo que permite cierta eficiencia en la ejecución.

1.3.2 Lenguajes Funcionales. Los matemáticos desde hace un buen tiempo están resolviendo problemas usando el concepto de función. Una función convierte ciertos datos en resultados. Así, algunos matemáticos inventaron los lenguajes de programación funcionales. Además, aprovecharon la posibilidad que tienen las funciones para manipular datos simbólicos, y no solamente numéricos, y la propiedad de las funciones que les permite componer, creando de esta manera, la oportunidad para resolver problemas complejos a partir de las soluciones a otros más sencillos. También se incluyó la posibilidad de definir funciones recursivamente.

El lenguaje funcional más antiguo, y seguramente el más popular hasta la fecha, es LISP, diseñado por McCarthy en la segunda mitad de los años 50. Su área de aplicación es principalmente la Inteligencia Artificial. En la década de los 80 hubo una nueva ola de interés por los lenguajes funcionales, añadiendo la tipificación y algunos conceptos modernos de modularización y polimorfismo, como es el caso del lenguaje ML.

Programar en un lenguaje funcional significa construir funciones a partir de las ya existentes. Por lo tanto es importante conocer y comprender bien las funciones que conforman la base del lenguaje, así como las que ya fueron definidas previamente. De esta manera se pueden ir construyendo aplicaciones cada vez más complejas. La desventaja de este modelo es que resulta bastante alejado del modelo de la máquina de von Neumann y, por lo tanto, la eficiencia de ejecución de los intérpretes de lenguajes funcionales no es comparable con la ejecución de los programas imperativos precompilados.

1.3.3 Lenguajes Lógicos. Otra forma de razonar para resolver problemas en matemáticas se fundamenta en la lógica de primer orden. El conocimiento básico de las matemáticas se puede representar en la lógica en forma de axiomas, a los cuales se añaden reglas formales para deducir cosas verdaderas (teoremas) a partir de los axiomas. Gracias al trabajo de algunos matemáticos, de finales de siglo XIX y principios del XX, se encontró la manera de automatizar computacionalmente el razonamiento lógico --particularmente para un subconjunto significativo de la lógica de primer orden-- que permitió que la lógica matemática diera origen a otro tipo de lenguajes de programación, conocidos como lenguajes lógicos. También se conoce a estos lenguajes, y a los funcionales, como *lenguajes declarativos*, porque el programador, para solucionar un problema, todo lo que tiene que hacer es describirlo vía axiomas y reglas de deducción en el caso de la programación lógica y vía funciones en el caso de la programación funcional.

En los lenguajes lógicos se utiliza el formalismo de la lógica para representar el conocimiento sobre un problema y para hacer preguntas que, si se demuestra que se pueden deducir a partir del conocimiento dado en forma de axiomas y de las reglas de deducción estipuladas, se vuelven teoremas. Así se encuentran soluciones a problemas formulados como preguntas. Con base en la información expresada dentro de la lógica de primer orden, se formulan las preguntas sobre el dominio del problema y el intérprete del lenguaje lógico trata de encontrar la respuesta automáticamente. El conocimiento sobre el problema se expresa en forma de predicados (axiomas) que establecen relaciones sobre los símbolos que representan los datos del dominio del problema.

El PROLOG es el primer lenguaje lógico y el más conocido y utilizado. Sus orígenes se remontan a los inicios de la década de los 70 con los trabajos del grupo de A. Colmerauer en Marsella, Francia. También en este caso, las aplicaciones a la Inteligencia Artificial mantienen el lenguaje vivo y útil.

En el caso de la programación lógica, el trabajo del programador se restringe a la buena descripción del problema en forma de hechos y reglas. A partir de ésta se pueden encontrar muchas soluciones dependiendo de como se formulen las preguntas (metas), que tienen sentido para el problema. Si el programa está bien definido, el sistema encuentra automáticamente las respuestas a las preguntas formuladas. En este caso ya no es necesario definir el algoritmo de solución, como en la programación imperativa, en cambio, lo fundamental aquí es expresar bien el conocimiento sobre el problema mismo. En programación lógica, al igual que en programación funcional, el programa, en este caso los hechos y las reglas, están muy alejados del modelo Von Neumann que posee la máquina en la que tienen que ser interpretados; por lo tanto, la eficiencia de la ejecución no puede ser comparable con la de un programa equivalente escrito en un lenguaje imperativo. Sin embargo, para cierto tipo de problemas, la formulación del programa mismo puede ser mucho más sencilla y natural (para un programador experimentado, por supuesto).

1.3.4 Lenguajes Orientados a Objetos. A mediados de los años 60 se empezó a vislumbrar el uso de las computadoras para la simulación de problemas del mundo real. Pero el mundo real está lleno de objetos, en la mayoría de los casos complejos, los cuales difícilmente se traducen a los tipos de datos primitivos de los lenguajes imperativos. Así es que a dos noruegos, Dahl y Nygaard, se les ocurrió el concepto de objeto y sus colecciones, llamadas clases de objetos, que permitieron introducir abstracciones de datos a los lenguajes de programación. La posibilidad de reutilización del código y sus indispensables modificaciones, se reflejaron en la idea de las jerarquías de herencia de clases. A ellos también les debemos el concepto de polimorfismo introducido vía procedimientos virtuales. Todos estos conceptos fueron presentados en el lenguaje Simula 67, desde el año 1967. Aunque pensado como lenguaje de propósito general, Simula tuvo su mayor éxito en las aplicaciones de simulación discreta, gracias a la clase SIMULATION que facilitaba considerablemente la programación.

La comunidad informática ha tardado demasiado en entender la utilidad de los conceptos básicos de Simula 67, que hoy identificamos como conceptos del modelo de objetos. Tuvi-
mos que esperar hasta los años 80 para vivir una verdadera ola de propuestas de lenguajes de programación con conceptos de objetos encabezada por Smalltalk, C++, Eiffel, Modula-

3, Ada 95 y terminando con Java. La moda de objetos se ha extendido de los lenguajes de programación a la Ingeniería de *Software*.

El modelo de objetos, y los lenguajes que lo usan, parecen facilitar la construcción de sistemas o programas en forma modular. Los objetos ayudan a expresar programas en términos de abstracciones del mundo real, lo que aumenta su comprensión. La clase ofrece cierto tipo de modularización que facilita las modificaciones al sistema. La reutilización de clases previamente probadas en distintos sistemas también es otro punto a favor. Sin embargo, el modelo de objetos, a la hora de ser interpretado en la arquitectura Von Neumann conlleva un excesivo manejo dinámico de memoria debido a la constante creación de objetos, así como a una carga de código fuerte causada por la constante invocación de métodos. Por lo tanto, los programas en lenguajes orientados a objetos siempre pierden en eficiencia, en tiempo y memoria, contra los programas equivalentes en lenguajes imperativos.

1.3.5 Lenguajes Concurrentes, Paralelos y Distribuidos. La necesidad de ofrecer concurrencia en el acceso a los recursos computacionales se remonta a los primeros sistemas operativos. Mientras que un programa realizaba una operación de entrada o salida otro podría gozar del tiempo del procesador para sumar dos números, por ejemplo. Aprovechar al máximo los recursos computacionales fue una necesidad apremiante, sobre todo en la época en que las computadoras eran caras y escasas; el sistema operativo tenía que ofrecer la ejecución concurrente y segura de programas de varios usuarios, que desde distintas terminales utilizaban un solo procesador, y así surgió la necesidad de introducir algunos conceptos de programación concurrente para programar los sistemas operativos.

Posteriormente, cuando los procesadores cambiaron de tamaño y de precio, se abrió la posibilidad de contar con varios procesadores en una máquina y ofrecer el *procesamiento en paralelo*, es decir, procesar varios programas al mismo tiempo. Esto dio el impulso a la creación de lenguajes que permitían expresar el paralelismo. Finalmente, llegaron las redes de computadoras, que también ofrecen la posibilidad de ejecución en paralelo, pero con procesadores distantes, lo cual conocemos como la *programación distribuida*.

En resumen, el origen de los conceptos para el manejo de concurrencia, paralelismo y distribución está en el deseo de aprovechar al máximo la arquitectura von Neumann y sus modalidades reflejadas en conexiones paralelas y distribuidas.

Históricamente en la literatura se encuentran soluciones conceptuales y mecanismos tales como: semáforos, regiones críticas, monitores, envío de mensajes (CSP), llamadas a procedimientos remotos (RPC), que posteriormente se incluyeron como partes de los lenguajes de programación en Concurrent Pascal, Modula, Ada, OCCAM, y últimamente en Java.

Uno de los ejemplos más importantes es el modelo de envío de mensajes de CSP de Hoare, para las arquitecturas paralelas y distribuidas, el cual no solamente fructificó en una propuesta del lenguaje de programación OCCAM, sino dio origen a una nueva familia de procesadores, llamados "transputers", que fácilmente se componen en una arquitectura paralela. Hoy en día, la programación de sistemas complejos tiene que incluir las partes de comunicaciones, programación distribuida y concurrencia.

CAPÍTULO II

CLUSTER

2.1 DEFINICIÓN DE CLUSTER (Supercomputadora Virtual)

Un cluster es un conjunto de equipos de cómputo independientes interconectados entre si por medio de una red, que trabajan conjuntamente en la solución de un problema, de manera que ejecutan una serie de aplicaciones de forma conjunta; distribuyéndose la tarea entre ellas, y que además aparecen ante usuarios y aplicaciones como un solo sistema. Estos sistemas constituyen una solución flexible, de bajo costo y gran escalabilidad para aplicaciones que requieren una elevada capacidad de cómputo y memoria.

A diferencia de las supercomputadoras tradicionales^[2], los sistemas en cluster requieren de una inversión inicial menor, ya que pueden construirse a partir de PC's de uso común. Por esta razón, los costos de mantenimiento y actualización se reducen notablemente. Debido a las supercomputadoras, sus antecesoras, los cluster son también llamados supercomputadoras virtuales.

2.2 COMPONENTES DE UN CLUSTER

En general, un cluster necesita de varios componentes de software y hardware para poder funcionar. A saber:

- Nodos (las computadoras).
- Sistemas Operativos.
- Conexiones de Red.
- Middleware (capa de abstracción entre el usuario y los sistemas operativos).
- Protocolos de Comunicación y servicios.
- Aplicaciones (pueden ser paralelas o no).

2.2.1 Nodos. Pueden ser simples computadoras, sistemas multi-procesador o estaciones de trabajo (workstations).

2.2.2 Sistema Operativo. Debe ser de fácil uso y acceso. Además permitir múltiples procesos así como múltiples usuarios. Algunos ejemplos de Sistemas Operativos pueden ser:

GNU/Linux

Unix: Solaris / HP-Ux / Aix

Windows NT / 2000 / 2003 Server

Mac OS X

² Estas máquinas con un alto costo están diseñada para procesar enormes cantidades de información en poco tiempo y son dedicadas a una tarea específica.

- Cluster OS's especiales

2.2.3 Conexiones de Red. Los nodos de un cluster pueden conectarse mediante una simple red Ethernet con tarjetas comunes (network adapters o NIC'S), o utilizarse tecnologías especiales de alta velocidad como Fast Ethernet, Gigabit Ethernet, Myrinet, Infiniband, SCI, etc.

	ANCHO DE BANDA
Ethernet	10 Mbps
Fast Ethernet	100 Mbps
Giganet (GigE)	800 Mbps
Gigabit Ethernet	1 Gbps
Myrinet	2 Gbps
Infiniband	2.5 Gbps

2.2.4 Middleware. El middleware es un software de conectividad que permite ofrecer un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas. Generalmente actúa entre el sistema operativo y las aplicaciones con la finalidad de proveer a un cluster lo siguiente:

- Una interfaz única de acceso al sistema, denominada SSI (*Single System Image*), la cual genera la sensación al usuario de que utiliza una única computadora muy potente.
- Herramientas para la optimización y mantenimiento del sistema: migración de procesos, *checkpoint-restart* (congelar uno o varios procesos, mudarlos de servidor y continuar su funcionamiento en el nuevo host), balanceo de carga, tolerancia a fallos, etc.
- Escalabilidad: debe poder detectar automáticamente nuevos servidores conectados al cluster para proceder a su utilización.
-

Existen diversos tipos de middleware, como por ejemplo: MOSIX, OpenMOSIX, Condor, OpenSSI, etc.

El middleware recibe los trabajos entrantes al cluster y los redistribuye de manera que el proceso se ejecute más rápido y el sistema no sufra sobrecargas en un nodo. Esto se realiza mediante políticas definidas en el sistema (automáticamente o por un administrador) que le indican dónde y cómo debe distribuir los procesos, por un sistema de monitorización, el cual controla la carga de cada CPU y la cantidad de procesos en él.

El middleware también debe poder migrar procesos entre servidores con distintas finalidades:

- Balancear la carga: si un nodo está muy cargado de procesos y otro está ocioso, pueden transferirse procesos a este último para liberar de carga al primero y optimizar el funcionamiento.

- Mantenimiento de nodos: si hay procesos corriendo en un nodo que necesita mantenimiento o una actualización, es posible migrar los procesos a otro nodo y proceder a desconectar del cluster al primero.
- Priorización de trabajos: en caso de tener varios procesos corriendo en el cluster, pero uno de ellos de mayor importancia que los demás, puede migrarse este proceso a los servidores que posean más o mejores recursos para acelerar su procesamiento.

2.2.5 Protocolos de Comunicación. Los Protocolos de Comunicación o también Protocolos de red son el conjunto de reglas que permiten el intercambio de datos u órdenes durante la comunicación entre las entidades que forman parte de una red.

2.3 CARACTERISTICAS GENERALES DE UN CLUSTER

Debido a que un cluster es un conjunto de computadoras conectadas entre si por una red local, utilizadas todas ellas para funcionar como un solo sistema es importante que cuente con características tales como software de control especializado, escalado vertical y horizontal, tener un alto grado de disponibilidad y confiabilidad así como también contar con tolerancia a fallos.

Para empezar, parte de este software se debe dedicar a la comunicación entre las computadoras (nodos).

2.3.1 Software a nivel de aplicación. Software generado por bibliotecas especiales dedicadas a clusters. Este tipo de software se sitúa a nivel de aplicación, se utilizan generalmente bibliotecas de carácter general que permiten la abstracción de un nodo a un sistema conjunto, permitiendo crear aplicaciones en un entorno distribuido de manera lo más abstracta posible. Este tipo de software suele generar elementos de proceso del tipo rutinas, procesos o tareas, que se ejecutan en cada nodo del cluster y se comunican entre sí a través de la red.

2.3.2 Software a nivel de sistema. Este tipo de software se sitúa a nivel de sistema, suele estar implementado como parte del sistema operativo de cada computadora, o ser la totalidad de esta.

Es más crítico y complejo, por otro lado suele resolver problemas de carácter más general que los anteriores y su eficiencia, por norma general, es mayor.

A pesar de esta división existen casos en los cuales se hace uso de un conjunto de piezas de software de cada tipo para conformar un sistema cluster completo. Son implementaciones híbridas donde un cluster puede tener implementado a nivel de kernel parte del sistema y otra parte estar preparada a nivel de usuario.

2.3.3 Acoplamiento de un cluster. Dependiendo del tipo de software, el sistema puede estar más o menos acoplado.

Se entiende por acoplamiento del software a la *integración que tengan todos los elementos software que existan en cada nodo* (nivel de colaboración que une los elementos del cluster). Gran parte de la integración del sistema la produce la comunicación entre los nodos, y

es por esta razón por la que se define el acoplamiento; otra parte es la que implica cómo de crítico es el software y su capacidad de recuperación ante fallos.

Se debe destacar que todo esto depende de si el sistema es centralizado o distribuido. En cualquier caso, el acoplamiento del software es una medida subjetiva basada en la integración de un sistema cluster a nivel general.

Se distinguen 3 tipos de acoplamiento:

- Acoplamiento fuerte.
- Acoplamiento medio.
- Acoplamiento débil.

Acoplamiento fuerte. El software que entra en este grupo es software cuyos elementos se interrelacionan mucho unos con otros y posibilitan la mayoría de las funcionalidades del cluster de manera altamente cooperativa. El caso de acoplamiento más fuerte que se puede dar es que solamente haya una imagen del kernel del sistema operativo, distribuida entre un conjunto de nodos que la compartirán. Por supuesto algo fundamental es poder acceder a todas las partes de este sistema operativo, estrechamente relacionadas entre sí y distribuidas entre los nodos.

Este caso es el que se considera como más acoplado, de hecho no está catalogado como cluster, sino como sistema operativo distribuido.

Otro ejemplo son los cluster SSI (Single System Image), en estos clusters todos los nodos ven una misma imagen del sistema, pero todos los nodos tienen su propio sistema operativo, aunque estos sistemas están estrechamente relacionados para dar la sensación a las aplicaciones que todos los nodos son idénticos y se acceda de una manera homogénea a los recursos del sistema total.

Si arranca o ejecuta una aplicación, ésta verá un sistema homogéneo, por lo tanto los kernels tienen que conocer los recursos de otros nodos para presentarle al sistema local los recursos que encontraría si estuviera en otro nodo. Por supuesto se necesita un sistema de nombres único, manejo de sistema distribuido o centralizado y un mapeo de los recursos físicos a este sistema de nombres.

Acoplamiento medio. A este grupo pertenece un software que no necesita un conocimiento tan exhaustivo de todos los recursos de otros nodos, pero que sigue usando el software de otros nodos para aplicaciones de muy bajo nivel. Como ejemplos hay openMosix y Linux-HA (High Availability o Alta Disponibilidad).

Un cluster openMosix necesita que todos los kernels sean de la misma versión. Por otro lado no está tan acoplado como el caso anterior: no necesita un sistema de nombres común en todos los nodos, y su capacidad de dividir los procesos en una parte local y otra remota consigue que por un lado se necesite el software del otro nodo donde está la parte del fichero que falta en el nodo local y por otro que no se necesite un SSI para hacer otras tareas.

Acoplamiento débil. Generalmente se basan en aplicaciones construidas por bibliotecas preparadas para aplicaciones distribuidas. Es el caso de por ejemplo PVM, MPI o CORBA. Éstos por sí mismos no funcionan en modo alguno con las características que antes se han descrito (como Beowulf) y hay que dotarles de una estructura superior que utilice las capacidades del cluster para que éste funcione.

2.3.4 Control del Cluster. El parámetro de control implica el modelo de gestión que propone el cluster. Este modelo de gestión hace referencia a la manera de configurar el cluster y es dependiente del modelo de conexionado o colaboración que surgen entre los nodos. Puede ser de dos tipos:

Control centralizado. Se hace uso de un nodo maestro desde el cual se puede configurar el comportamiento de todo el sistema. Este nodo es un punto crítico del sistema aunque es una ventaja para una mejor gestión del cluster.

Control descentralizado. En un modelo distribuido donde cada nodo debe administrarse y gestionarse. También pueden ser gestionados mediante aplicaciones de más alto nivel de manera centralizada, pero la mayoría de la gestión que hace el nodo local es leer archivos de configuración de su propio nodo.

Es propio de sistemas distribuidos, como ventaja tiene que presenta más tolerancia a fallos como sistema global, y como desventajas que la gestión y administración de los equipos requiere más tiempo.

La escalabilidad es la capacidad de un equipo para hacer frente a volúmenes de trabajo cada vez mayores sin, por ello, dejar de prestar un nivel de rendimiento aceptable.

2.3.5 Escalado vertical. El escalado vertical implica aumentar los recursos del sistema (como procesadores, memoria, discos y adaptadores de red) al hardware existente o reemplazar hardware existente por otro con mayores recursos de sistema. Por ejemplo, si el hardware actual no ofrece un rendimiento adecuado para los usuarios, puede considerar la posibilidad de agregar memoria RAM o unidades centrales de procesamiento (CPU) a los nodos (computadoras) del clúster para satisfacer esa demanda.

2.3.6 Escalado horizontal. El escalado horizontal implica agregar equipos de cómputo para atender a la demanda, esto significa agregar nodos al clúster. El escalado horizontal también es adecuado cuando se desea mejorar el tiempo de respuesta de los clientes con los servidores.

2.3.7 Disponibilidad. Es la calidad de estar presente, listo para su uso, a mano, accesible. La métrica empleada para medir la disponibilidad es el porcentaje de tiempo que un sistema es capaz de realizar las funciones para las que está diseñado.

2.3.8 Fiabilidad. Es la probabilidad de un funcionamiento correcto. Las medidas de confiabilidad suelen utilizarse para calcular la probabilidad de que se produzca un error. Un sistema es más confiable si es tolerante a errores.

2.3.9 Tolerancia a fallos. Es la capacidad de un sistema para seguir funcionando cuando se produce un error en parte del sistema. Para conseguir tolerancia a fallos hay que diseñar el sistema con un alto grado de redundancia de hardware. Si se produce un error en un único componente, el componente redundante asumirá su función sin que se produzca un tiempo de inactividad apreciable.

Cuando se trata de resolver un problema en paralelo, el software utilizado en un cluster debe ser capaz de dividir el problema en tareas más pequeñas, repartirlas entre los nodos y elaborar los resultados. Puesto que las subtarefas van a ejecutarse en paralelo se consigue un aumento de velocidad, aunque hay que tener en cuenta el retardo que la división, reparto y transmisión de mensajes (resultado, coherencia, estados...) supone.

2.4 TIPOS DE CLUSTER SEGÚN EL TIPO DE NODO

2.4.1 Homogéneo. Las computadoras tienen todas las mismas configuraciones hardware y el mismo sistema operativo. Todos los nodos tienen una arquitectura y recursos similares, de manera que no existen muchas diferencias entre cada nodo.

2.4.2 Semi-homogéneo. Diferente rendimiento pero con arquitecturas y sistemas operativos similares.

2.4.3 Heterogéneo. Diferente hardware y sistema operativo lo que hace mas fácil y económica su construcción. Algunas de estas distinciones pueden estar en los siguientes puntos:

- Tiempos de acceso distintos
- Arquitectura distinta
- Sistema operativo distinto
- Rendimiento de los procesadores o recursos sobre una misma arquitectura distintos

El uso de arquitecturas distintas, o distintos sistemas operativos, impone que exista una biblioteca que haga de interfaz, e incluso una sintaxis de notación abstracta del tipo ASN.1 o XDR en la capa de presentación que utilice la interfaz de comunicación de nuestro sistema distribuido o cluster. Esto hace que este tipo de clusters se consideren implementados a nivel de aplicación.

2.5 TIPOS DE CLUSTER SEGÚN SU APLICABILIDAD

Según la *aplicabilidad* de los clusters, se han desarrollado diferentes líneas tecnológicas.

2.5.1 Cluster de Alta Disponibilidad (HA – High Availability). Tanto en el mundo empresarial, como en el mundo académico, existen ciertas aplicaciones que dada su naturaleza deben proporcionar un servicio ininterrumpido de 24 horas al día, 7 días a la semana (24*7). Para conseguir estos niveles de disponibilidad se suele utilizar una configuración avanzada de hardware y software denominado *Cluster de Alta Disponibilidad* o *Clusters de Redundancia*, los cuales se hacen necesarios en ambientes de intercambio masivo de información, almacenamiento de datos sensibles y allí donde sea necesaria una disponibilidad

del servicio ofrecido durante el mayor tiempo posible. Para poder dar estos servicios los clusters de este tipo se implementan en base a tres factores.

- Fiabilidad.
- Disponibilidad.
- Dotación de servicio.

Mediante estos tres tipos de actuaciones y los mecanismos que lo implementan se asegura que un servicio esté el máximo tiempo disponible y que éste funcione de una manera fiable. Respecto al tercer punto, se refiere a la dotación de uno de estos clusters de un servicio que provea a clientes externos.

Un cluster de alta disponibilidad es un conjunto de dos o mas máquinas, que se caracterizan porque comparten los discos de almacenamiento de datos, y porque están constantemente monitorizándose entre sí. En caso de producirse un fallo del hardware o de las aplicaciones de alguna de las máquinas del cluster, el software de alta disponibilidad es capaz de reiniciar automáticamente los servicios que han faltado en cualquiera de las otras máquinas del cluster. Y cuando la máquina que ha fallado se recupera, los servicios son nuevamente migrados a la máquina original. Esta capacidad de recuperación automática de servicios nos garantiza la integridad de la información, ya que no hay pérdida de datos.

Entre los problemas que solucionan se encuentran:

- Sistemas de información redundante
- Sistemas tolerantes a fallos
- Balanceo de carga entre varios servidores
- Balanceo de conexiones entre varios servidores

En general todos estos problemas se ligan en dos fuentes de necesidad de las empresas u organizaciones.

- Tener un servicio disponible
- Ahorrar económicamente todo lo que sea posible

2.5.2 Cluster de Alto Rendimiento (HP – High Performance). Este tipo de cluster surge frente a la necesidad de súper computo para determinadas aplicaciones, lo que se persigue es conseguir que un gran número de máquinas individuales actúen como una sola máquina muy potente. Este tipo de clusters se aplica mejor en problemas grandes y complejos que requieren una cantidad enorme de potencia computacional. La misión o el objetivo de este tipo de cluster es, como su propio nombre indica mejorar el rendimiento en la obtención de la solución de un problema, en términos bien del tiempo de respuesta o bien de su precisión.

Un *cluster de alto rendimiento* tiene como objetivo mejorar el rendimiento en la obtención de la solución de un problema. Cualquier operación que necesite altos tiempos de CPU puede ser utilizada en un cluster de alto rendimiento, siempre que se encuentre un algoritmo que sea paralelizable. Dentro de esta definición no englobamos ningún tipo de problema

en especial. Esto supone que cualquier cluster que haga que el rendimiento del sistema general aumente respecto al de uno de los nodos individuales puede ser considerado cluster de alto rendimiento. Generalmente, los problemas que se le plantean a una computadora, suelen ser de carácter computacional, por esto, este tipo de clusters suele ser denominado clusters de alto rendimiento de cómputo, pero esto no implica que no solucione otros problemas. Se podría definir estos sistemas como sistemas distribuidos (en cada nodo) en los cuales se resuelve de manera distribuida un problema de cómputo.

Generalmente estos problemas de cómputo suelen estar ligados a:

- Cálculos matemáticos.
- Renderizaciones de gráficos.
- Compilación de programas.
- Compresión de datos.
- Descifrado de códigos.
- Rendimiento del sistema operativo, (incluyendo en él, el rendimiento de los recursos de cada nodo).

Existen otros muchos problemas más que se pueden solucionar con clusters de alto rendimiento, donde cada uno aplica de una manera u otra las técnicas necesarias para habilitar la paralelización del problema, su distribución entre los nodos y obtención del resultado.

Existen clusters que pueden ser denominados de alto rendimiento tanto a *nivel de sistema* como a *nivel de aplicación*. A nivel de sistema tenemos openMosix, mientras que a nivel de aplicación se encuentran otros como MPI, PVM, Beowulf y otros muchos. En cualquier caso, estos clusters hacen uso de la capacidad de procesamiento que pueden tener varias máquinas.

Los clusters implementados a nivel de aplicación no suelen implementar balanceo de carga. Suelen basar todo su funcionamiento en una política de localización que sitúa las tareas en los diferentes nodos del cluster, y las comunica mediante las librerías abstractas. Resuelven problemas de cualquier tipo como los mencionados anteriormente, pero se deben diseñar y codificar aplicaciones propias para cada tipo para poderlas utilizar en estos clusters.

Por otro lado están los sistemas de alto rendimiento implementados a nivel de sistema. Estos clusters basan todo su funcionamiento en comunicación y colaboración de los nodos a nivel de sistema operativo, lo que implica generalmente que son clusters de nodos de la misma arquitectura, con ventajas en lo que se refiere al factor de acoplamiento, y que basan su funcionamiento en compartición de recursos a cualquier nivel, balanceo de la carga de manera dinámica, funciones de planificación especiales y otros tantos factores que componen el sistema.

Entre las limitaciones que existen actualmente está la incapacidad de balancear la carga dinámica de las librerías PVM o la incapacidad de openMosix de migrar procesos que hacen uso de memoria compartida. Una técnica que obtiene mayor ventaja es cruzar ambos sistemas: PVM + openMosix. Se obtiene un sistema con un factor de acoplamiento elevado que presta las ventajas de uno y otro, con una pequeña limitación por desventajas de cada uno.

2.5.3 Clusters de Alta Confiabilidad (HR – High Reliability). Estos clusters tratan de aportar la máxima confiabilidad en un entorno, en la cual se necesite saber que el sistema se va a comportar de una manera determinada. Puede tratarse por ejemplo de sistemas de respuesta a tiempo real. Este tipo de clusters son los más difíciles de implementar. No se basan solamente en conceder servicios de alta disponibilidad, sino en ofrecer un entorno de sistema altamente confiable. Esto implica muchísima sobrecarga en el sistema, son también clusters muy acoplados.

Dar a un cluster SSI (Single System Image) capacidad de alta confiabilidad implica gastar recursos necesarios para evitar que aplicaciones caigan. Esto es, en los clusters de alta disponibilidad generalmente una vez que el servicio ha caído éste se relanza, y no existe manera de conservar el estado del servidor anterior, más que mediante puntos de parada o *checkpoints*, pero que en conexiones en tiempo real no suelen ser suficientes. Por otro lado, los clusters confiables tratan de mantener el estado de las aplicaciones, no simplemente de utilizar el último checkpoint del sistema y relanzar el servicio.

Generalmente este tipo de clusters suele ser utilizado para entornos de tipo empresarial y esta funcionalidad solamente puede ser efectuada por hardware especializado. Al balancear la carga de trabajo en un conjunto de servidores, se mejora el tiempo de acceso y la confiabilidad. Además como es un conjunto de servidores el que atiende el trabajo, la caída de uno de ellos no ocasiona una caída total del sistema. Este tipo de servicio es de gran valor para quienes trabajan con grandes volúmenes de tráfico y trabajo en sus Web, servidores de correo... Por el momento no existe ninguno de estos clusters implementados como software de manera eficiente en tiempo real. Esto se debe a limitaciones de la latencia^[3] de la red, así como a la complejidad de mantener los estados.

2.6 FUNCIONAMIENTO DE UN CLUSTER

Desde un punto de vista general, un cluster consta de dos partes.

* La primera es el **software**: un sistema operativo confeccionado especialmente para esta tarea (modificaciones al kernel Linux), compiladores y aplicaciones especiales, que permite que los programas que se ejecutan en el sistema exploten todas las ventajas del cluster. En el sistema operativo de GNU/Linux (software libre)^[4] hay que destacar las aplicaciones GNU: PVM (Paralell Virtual Machine) y la MPI (Message Passing Interface), librerías que abstraen la componente hardware de la componente software.

* La segunda componente es la **interconexión de hardware** entre las máquinas (nodos) del cluster. Se han desarrollado interfaces de interconexión especiales muy eficientes, pero es común realizar las interconexiones mediante una red Ethernet dedicada de alta velocidad.

³ Lapso necesario para que un paquete de información viaje desde la fuente hasta su destino. La latencia y el ancho de banda, juntos, definen la capacidad y la velocidad de una red.

⁴ GNU es un acrónimo recursivo que significa "GNU No es Unix".

"Software Libre" se refiere a la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software.

Gracias a esta red de interconexión los nodos del cluster intercambian entre sí las tareas, las actualizaciones de estado y los datos del programa. En un cluster abierto, existirá una interfaz de red que conecte al cluster con el mundo exterior (Internet).

Los nodos en un cluster están dedicados exclusivamente a ejecutar tareas asignadas al cluster. Por lo general el cluster se encuentra comunicado al mundo exterior a través de un solo nodo, llamado nodo maestro, el cual también está reservado para acceder, compilar y manejar las aplicaciones a ejecutar.

Los servidores de un cluster de alta disponibilidad normalmente no comparten la carga de procesamiento que tiene un cluster de Alto Rendimiento. Tampoco comparten la carga de tráfico como lo hacen los clusters de Balanceo de Carga. Su función es la de estar preparados para entrar inmediatamente en funcionamiento en caso de que falle algún otro servidor.

2.7 CLUSTER BEOWULF

El proyecto Beowulf comenzó con un cluster en Linux^[5] y surgió en la NASA (desarrollado por Thomas Sterling y Donald Becker, en el Centro de Vuelos Espaciales Goddard) en el año de 1994. Lo que dio origen a esta supercomputadora virtual fue el hecho de que se requería una computadora en la cual pudieran procesarse grandes cantidades de información, sin que su costo fuera muy elevado.

Es por ello que un cluster Beowulf es creado con un conjunto de nodos minimalistas (con lo mínimo para funcionar) basado en componentes hardware de bajo costo y software libre para su gestión y programación. Es decir, no es más que una colección de computadoras personales interconectadas por medio de una red privada de alta velocidad. Cabe mencionar que la red de trabajo siempre ha sido un factor limitante desde sus inicios, ya que una vez que se mueve entre los 100 y 200 nodos, la red de trabajo fácilmente se satura. El estándar Ethernet de banda ancha de 1 Gbps o más, eliminará el problema, y permitirá escalar a un gran número de nodos para aplicaciones que puedan incorporar mucho paralelismo.

Los clusters Beowulf están basados en el sistema operativo Linux; ya que este sistema operativo funciona en una variedad de computadoras. Estos clusters se utilizan para cualquier tarea que requiera enormes cantidades de cómputo: data mining, simulaciones científicas, renderización de gráficos, modelado meteorológico. La manera en que efectúan el procesamiento de datos es la siguiente: la máquina o nodo principal distribuye en paralelo el trabajo al resto de las PC's, posteriormente éstas envían a la máquina principal los resultados obtenidos individualmente para convertirlos en uno solo. Siendo sólo el "nodo maestro" visible desde el exterior. El nodo maestro está reservado para acceder, compilar y manejar las aplicaciones a ejecutar. Por otro lado, cuando cualquier nodo del cluster se daña, todos los otros siguen funcionando normalmente absorbiendo la carga total del sistema y de esta forma se obtiene una tolerancia a fallas superior. En cuanto a la escalabilidad, es muy sim-

⁵ Beowulf, nombre de un héroe de la mitología danesa relatado en el libro La Era de las Fábulas, del autor norteamericano Thomas Bulfinch.

Este cluster consistente en procesadores de tipo x86 comerciales conectados por una red Ethernet de 10Mb, utilizando este software libre.

ple incorporar nuevos nodos en un cluster y expandirlo progresivamente, en la medida de las posibilidades locales.

En la taxonomía de las computadoras paralelas, el cluster Beowulf cae en alguna parte entre MPP (Procesadores Masivamente Paralelos), como nCube, CM5, Convex SPP, Cray T3D, Cray, T3E, etc. y NOWs (red de estaciones de trabajo). El proyecto Beowulf se beneficia del desarrollo de ambas clases de arquitectura.

La disponibilidad gratuita, alta confiabilidad y la eficiencia de Linux, ha sido una gran fortuna para las ciencias computacionales desde la década de los 90's y su popularidad crece cada día en los medios académicos. Los científicos utilizan Linux para convertir PC's en estaciones de trabajo UNIX efectivas para la realización de un gran número de tareas, incluyendo cómputo numérico de modelos complejos. El desarrollo de los clusters de computadoras tipo Beowulf, ha extendido la utilidad de Linux al campo del cómputo paralelo de alto rendimiento. Adicionalmente, la naturaleza de fuente abierta del sistema Linux ha permitido a los programadores, añadir directamente características adicionales al sistema operativo para responder a las necesidades en los clusters de computación.

Aunque por la naturaleza de los problemas a los que dan solución este tipo de clusters lo normal es desarrollar aplicaciones específicas, existen también algunas soluciones generales.

Dentro del proyecto Beowulf se puede encontrar un kernel modificado y algunas herramientas y librerías usadas para presentar el cluster como un único sistema. La idea es que los procesos que se ejecutan en los nodos esclavos son manejados por un nodo maestro, dando la impresión de que el cluster es un único sistema.

Hay otros proyectos en los que también se utiliza la idea de un único sistema. OpenMosix es un conjunto de extensiones del kernel estándar, así como herramientas desarrolladas para que el uso del cluster sea más eficiente.

SCE (Scalable Cluster Environment) es un conjunto de herramientas que permiten construir y usar un cluster Beowulf. Bproc, el programa núcleo del proyecto Beowulf tiene la habilidad de presentar el sistema como único y es usado en ClubMask, Kickstart, cfengine, the Maui scheduler, LAM/MPI.

Existen otros clusters computacionales que no modifican la forma en que funciona el kernel. Éstos utilizan otros medios para ejecutar tareas y mostrar información sobre ellas. Cplant, el KaClustering Toolkit, y OSCAR permiten construir, usar y administrar clusters de este modo.

Con la madurez y la robustez ^[6] de Linux además de la estandarización de envío de mensajes vía PVM y MPI, los programadores ahora tienen la garantía de que los programas que

⁶ Un programa es maduro cuando consigue la calidad de software. Un programa es robusto cuando no falla, es decir que reacciona adecuadamente ante los eventos.

escriban correrán en un futuro en un Cluster Beowulf sin importar quien hace los procesadores o las redes de trabajo.

2.8 BENEFICIOS DE LOS CLUSTERS

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

- Cada uno de sus nodos es una máquina completa que se puede utilizar como una estación de trabajo normal, al mismo tiempo que es empleada para máquina paralela.
- Gran capacidad de expansión y fiabilidad. Debido a que éste puede estar formado por PC's normales de gama media o incluso baja (Pentium, Pentium II o III), su precio es relativamente bajo.
- Alto rendimiento, además del incremento de velocidad de procesamiento ofrecido por los clusters de alto rendimiento e incremento del número de transacciones o velocidad de respuesta ofrecida por los cluster de balance de carga.
- Incremento de confiabilidad ofrecido por los clusters de alta disponibilidad. Por ejemplo, en las investigaciones meteorológicas y pronóstico numérico del estado del tiempo, se requiere el manejo de cantidades masivas de datos y cálculos muy complejos. Al combinar el poder de muchas máquinas del tipo estación de trabajo o servidor, se pueden alcanzar niveles de rendimiento similares a los de las supercomputadoras, pero a menor costo (pues estas requieren de hardware y software especializado muy caro, así como personal de soporte técnico dedicado).
- Debido a que la construcción de una supercomputadora de n procesadores (tomando n como un valor superior a 4) es complejo, caro y en muchos casos irrealizable.
- Los clusters de estación de trabajo son baratos y son una alternativa lista y disponible a las plataformas especializadas de computación paralela. Los clusters son fácilmente escalables. La capacidad de los nodos se puede ampliar de forma sencilla, añadiendo memoria o procesadores adicionales.

CAPÍTULO III

PARALELISMO

3.1 PROGRAMACIÓN PARALELA: LIBRERÍAS DE PASO DE MENSAJES

Las librerías de paso de mensajes se pueden definir como un conjunto de funciones de comunicación que permite implementar el modelo de paso de mensajes.

Algunos de sus componentes son:

- Rutinas de gestión de procesos: inicio, fin, identificación, etc.
- Rutinas de comunicación punto a punto: envío y recepción entre 2 procesadores.
- Rutinas de comunicación colectiva: Operaciones entre varios procesos y de sincronización de procesos.

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

Tanto PVM como MPI se basan en el concepto de paso de mensajes ^[7]. Los mensajes son pasados entre los procesos para conseguir que se ejecuten de manera colaborativa y de forma sincronizada. Se ha elegido mensajes pues se puede implementar de forma más o menos efectiva en un cluster, los mensajes se pueden enviar en forma de paquete IP y el ordenador destino desempaqueta el mensaje y decide a que proceso va dirigido. Una vez hecho esto, envía la información al proceso en cuestión. MPI en particular se necesita conocer de forma básica los mecanismos de paso de mensajes. Hay tres mecanismos básicos de paso de mensajes:

3.1.1 Paso de mensajes síncrono. Cuando un proceso P ejecuta un envío síncrono a un proceso Q, tiene que esperar hasta que el proceso Q ejecuta el correspondiente recibo de información síncrono. Ambos procesos no volverán del envío o el recibo hasta que el mensaje está a la vez enviado y recibido.

Cuando el enviar y recibir acaban, el mensaje original puede ser inmediatamente sobrescrito por el nuevo mensaje de vuelta y éste puede ser inmediatamente leído por el proceso que envió originariamente el mensaje. No se necesita un buffer extra en el mismo buffer donde se encuentra el mensaje de ida, se escribe el mensaje de vuelta.

⁷ El paso de mensajes es una tarea ampliamente usada en ciertas clases de máquinas paralelas, especialmente aquellas que cuentan con memoria distribuida.

Enviar/Recibir bloqueante. Un envío bloqueante es ejecutado cuando un proceso lo alcanza sin esperar el recibo correspondiente. Esta llamada bloquea hasta que el mensaje es efectivamente enviado, lo que significa que el mensaje (el buffer donde se encuentra el mensaje) puede ser reescrito sin problemas. Cuando la operación de enviar ha acabado no es necesario que se haya ejecutado una operación de recibir. Sólo sabemos que el mensaje fue enviado, puede haber sido recibido o puede estar en un buffer del nodo que lo envía, o en un buffer de algún lugar de la red de comunicaciones o puede que esté en el buffer del nodo receptor.

Un recibo bloqueante es ejecutado cuando un proceso lo alcanza, sin esperar a su correspondiente envío. Sin embargo no puede acabar sin recibir un mensaje. Quizás el sistema esté proveyendo un buffer temporal para los mensajes.

Envío/recibo no bloqueante. Un envío no bloqueante es ejecutado cuando un proceso lo alcanza, sin esperar al recibo. Puede acabar inmediatamente tras notificar al sistema que debe enviar el mensaje. Los datos del mensaje no están necesariamente fuera del buffer del mensaje, por lo que es posible incurrir en error si se sobrescriben los datos.

Un recibo no bloqueante es ejecutado cuando un proceso lo alcanza, sin esperar el envío. Puede volver inmediatamente tras notificar al sistema que hay un mensaje que se debe recibir. El mensaje puede que no haya llegado aún, puede estar todavía en transito o puede no haber sido enviado aún.

3.1.2 Paso de mensajes asíncrono. En el paso de mensajes asíncrono, el proceso que envía, no espera a que el mensaje sea recibido, y continúa su ejecución, siendo posible que vuelva a generar un nuevo mensaje y a enviarlo antes de que se haya recibido el anterior. Por este motivo se suelen emplear buzones, en los que se almacenan los mensajes a espera de que un proceso los reciba. Generalmente empleando este sistema, el proceso que envía mensajes solo se bloquea o para, cuando finaliza su ejecución, o si el buzón está lleno.

3.2 MPI

(Message Passing Interface)

En el modelo de programación MPI, un cómputo comprende de uno o más procesos comunicados a través de llamadas a rutinas de librerías para mandar (send) y recibir (receive) mensajes a otros procesos. En la mayoría de las implementaciones de MPI, se crea un conjunto fijo de procesos al inicializar el programa, y un proceso es creado por cada tarea. Sin embargo, estos procesos pueden ejecutar diferentes programas. De ahí que, el modelo de programación MPI es algunas veces referido como *MPMD* (multiple program multiple data) para distinguirlo del modelo *SPMD*, en el cual cada procesador ejecuta el mismo programa.

Debido a que el número de procesos en un cómputo de MPI es normalmente fijo, se puede enfatizar en el uso de los mecanismos para comunicar datos entre procesos. Los procesos pueden utilizar operaciones de comunicación punto a punto para mandar mensajes de un proceso a otro, estas operaciones pueden ser usadas para implementar comunicaciones loca-

les y no estructuradas. Un grupo de procesos puede llamar colectivamente operaciones de comunicación para realizar tareas globales tales como broadcast, etc. La habilidad de MPI para probar mensajes da como resultado el soportar comunicaciones asíncronas. Probablemente una de las características más importantes del MPI es el soporte para la programación modular. Un mecanismo llamado comunicador permite al programador del MPI definir módulos que encapsulan estructuras internas de comunicación (estos módulos pueden ser combinados secuencialmente y paralelamente).

MPI consigue portabilidad proveyendo una librería de paso de mensajes estándar independiente de la plataforma y de dominio público. La especificación de esta librería está en una forma independiente del lenguaje y proporciona funciones para ser usadas con C y Fortran. Abstrae los sistemas operativos y el hardware. Hay implementaciones MPI en casi todas las máquinas y sistemas operativos. Esto significa que un programa paralelo escrito en C o Fortran usando MPI para el paso de mensajes, puede funcionar sin cambios en una gran variedad de hardware y sistemas operativos. Por estas razones MPI ha ganado gran aceptación dentro de la computación paralela.

Además tiene que ser implementado sobre un entorno que se preocupe del manejo de los procesos y la E/S por ejemplo, puesto que MPI sólo se ocupa de la capa de comunicación por paso de mensajes. Necesita un ambiente de programación paralelo nativo.

Todos los procesos son creados cuando se carga el programa paralelo y están vivos hasta que el programa termina. Hay un grupo de procesos por defecto que consiste en todos esos procesos, identificado por `MPI_COMM_WORLD`.

Los procesos MPI son procesos como se han considerado tradicionalmente, del tipo pesados, cada proceso tiene su propio espacio de direcciones, por lo que otros procesos no pueden acceder directamente al las variables del espacio de direcciones de otro proceso. La intercomunicación de procesos se hace vía paso de mensajes.

Las desventajas de MPI, realmente son desventajas del modelo de paso de mensajes y de la implementación en espacio de usuario. Además aunque es un estándar y debería tener un API ^[8] estándar, cada una de las implementaciones varía, no en las llamadas sino en el número de llamadas implementadas (MPI tiene unas 200 llamadas). Esto hace que en la práctica los diseñadores del sistema y los programadores tengan que conocer el sistema particular de MPI para sacar el máximo rendimiento. Además como sólo especifica el método de paso de mensajes, el resto del entorno puede ser totalmente distinto en cada implementación con lo que otra vez se impide esa portabilidad que teóricamente tiene.

Existen implementaciones fuera del estándar que son tolerantes a fallos, no son versiones demasiado populares porque causan mucha sobrecarga.

⁸ Application Program Interface es un conjunto de rutinas, protocolos, y herramientas para construir aplicaciones de software.

3.3 PVM (Parallel Virtual Machine)

Es un conjunto de herramientas y librerías para el cómputo paralelo en un sistema distribuido de computadoras de manera que emulan un entorno de propósito general compuesto de nodos interconectados de distintas arquitecturas (computadoras heterogéneas). El objetivo es conseguir que ese conjunto de nodos pueda ser usado de forma colaborativa para el procesamiento paralelo. Esto es, comparta sus recursos de cómputo (como el procesador y la memoria RAM) con el fin de aprovechar esto para disminuir el tiempo de ejecución de un programa al distribuir la carga de trabajo en varias computadoras.

El modelo en el que se basa PVM es dividir las aplicaciones en distintas tareas (igual que ocurre con openMosix). Son los procesos los que se dividen por las máquinas para aprovechar todos los recursos. Cada tarea es responsable de una parte de la carga que conlleva esa aplicación. PVM soporta tanto paralelismo en datos, como funcional o una mezcla de ambos.

PVM permite que las tareas se comuniquen y sincronicen con las demás tareas de la máquina virtual, enviando y recibiendo mensajes, muchas tareas de una aplicación pueden cooperar para resolver un problema en paralelo. Cada tarea puede enviar un mensaje a cualquiera de las otras tareas, sin límite de tamaño ni de número de mensajes.

El sistema PVM se compone de dos partes. La primera es un demonio, llamado *pvmd* que reside en todos los nodos que forman parte de la máquina virtual. Cuando un usuario quiere ejecutar una aplicación PVM, primero crea una máquina virtual para arrancar PVM. Entonces se puede ejecutar la aplicación PVM en cualquiera de los nodos. Muchos usuarios pueden configurar varias máquinas virtuales aunque se mezclen unas con las otras y se pueden ejecutar varias aplicaciones PVM simultáneamente. Cada demonio es responsable de todas las aplicaciones que se ejecutan en su nodo.

Así el control está totalmente distribuido excepto por un demonio maestro, que es el primero que se ejecuta a mano por el usuario, los demás nodos fueron iniciados por el maestro y son esclavos. En todo momento siempre hay un *pvmd* maestro. Por tanto la máquina virtual mínima es de un miembro, el maestro.

La segunda parte del sistema es la *librería de PVM*. Contiene un repertorio de primitivas que son necesarias para la cooperación entre los procesos o threads de una aplicación. Esta librería contiene rutinas para inicialización y terminación de tareas, envío y recepción de mensajes, coordinar y sincronizar tareas, broadcast, modificar la máquina virtual.

Cuando un usuario define un conjunto de nodos, PVM abstrae toda la complejidad que tenga el sistema y toda esa complejidad se ve como un gran computador de memoria distribuida llamada máquina virtual. Esta máquina virtual es creada por el usuario cuando se comienza la operación. Es un conjunto de nodos elegidos por el usuario. En cualquier momento durante la operación puede elegir nuevos nodos para la máquina virtual. Esto puede ser de gran ayuda para mejorar la tolerancia a fallos pues se tiene unos cuantos nodos de reserva (PVM no tiene migración) para si alguno de los nodos fallara. O si se ve que un

conjunto de nodos de una determinada red están fallando se pueden habilitar nodos de otra red para solucionarlo. Para conseguir abstraer toda la complejidad de las diferentes configuraciones, soporta la heterogeneidad de un sistema a tres niveles:

- Aplicaciones: las subtareas pueden estar hechas para aprovechar las arquitecturas sobre la que funcionan. Por tanto como se puede elegir en que conjunto de nodos se ejecutarán unas tareas específicas, podemos hacer nuestras aplicaciones con la arquitectura al máximo por lo que se puede optimizar y hacer que funcionen aplicaciones hechas para arquitecturas específicas con PVM.
- Máquinas: nodos con distintos formatos de datos están soportados, incluyendo arquitecturas secuenciales, vectoriales, SMP. Abstrae little endian y big endian.
- Redes: la máquina virtual puede ser interconectada gracias a distintas tecnologías de red. Para PVM, bajo él existe una red punto a punto, no fiable y no secuencial. Esto abstrae cualquier tecnología de red. Utiliza UDP (User Datagram Protocol) e implementa toda la confiabilidad y todas las demás operaciones como broadcast en la propia librería PVM.

Tiene un conjunto de interfaces que está basado en la observación de las necesidades de la mayoría de las aplicaciones, que están escritas en C y Fortran. Los enlaces para C y C++ para la librería PVM están implementados como funciones, siguiendo las reglas usadas por la mayoría de los sistemas que usan C, incluyendo los sistemas operativos tipo UNIX. Los enlaces para Fortran están implementados como subrutinas más que funciones.

Todas las tareas están identificadas con un único identificador de tarea TID (Task Identifier). Los mensajes son enviados y recibidos por TIDs. Son únicos en toda la máquina virtual y están determinados por el *pvm*d local y no se pueden elegir por el usuario. Varias funciones devuelven estos TIDs (*pvm_mytid()*, *pvm_parent()*, *etc.*) para permitir que las aplicaciones de los usuarios conozcan datos de las otras tareas. Existen grupos nombrados por los usuarios, que son agrupaciones lógicas de tareas. Cuando una tarea se une al grupo, a ésta se le asigna un único número dentro de ese grupo. Estos números empiezan en 0 y hasta el número de tareas que disponga el grupo. Cualquier tarea puede unirse o dejar cualquier grupo en cualquier momento sin tener que informar a ninguna otra tarea del grupo. Los grupos se pueden superponer y las tareas pueden enviar mensajes multicast a grupos de los que no son miembro.

Cuando una tarea se quiere comunicar con otra ocurren una serie de cosas, los datos que la tarea ha enviado con una operación *send*, son transferidos a su demonio local quien decodifica el nodo de destino y transfiere los datos al demonio destino. Este demonio decodifica la tarea destino y le entrega los datos. Este protocolo necesita 3 transferencias de datos de las cuales solamente una es sobre la red. También se puede elegir una política de encaminado directo (dependiente de los recursos disponibles). En esta política tras la primera comunicación entre dos tareas los datos sobre el camino a seguir por los datos son guardados en una caché local. Las siguientes llamadas son hechas directamente gracias a esta información. De esta manera las transferencias se reducen a una transferencia sobre la red. Para comunicar entre sí los demonios *pvm*d se usa UDP pues es mucho más sencillo, sólo consume un

descriptor de fichero, y con un simple socket UDP se puede comunicar a todos los demás demonios. Además es muy sencillo colocar temporizadores sobre UDP para detectar fallos de nodo, *pvm* o red. La comunicación entre las tareas y los *pvm* es mediante TCP puesto que se necesita tener la seguridad de que los datos llegarán. En el caso de que sólo se haga una transferencia ésta es TCP por lo que hay que establecer la conexión primero por lo que realmente tampoco es tan beneficioso.

3.4 DIFERENCIAS ENTRE PVM Y MPI

PVM nació en 1989 como un proyecto universitario y ha evolucionado desde entonces beneficiándose de las experiencias de sus usuarios. El concepto clave en su diseño es la máquina virtual: un conjunto de procesadores conectados por una red es percibido por el usuario como una única computadora paralela. El objetivo de este diseño es la portabilidad y flexibilidad, pero con un posible impacto en el rendimiento. Por el contrario, MPI es un estándar formal desarrollado en 1993 por un comité de 60 expertos. El objetivo de MPI era definir un estándar que cada fabricante pudiera implantar para lograr el máximo rendimiento posible en su hardware. Los distintos orígenes de PVM y MPI así como sus distintas filosofías de diseño se traducen en un conjunto de diferencias:

- *Estándar*: PVM es el estándar de hecho para el procesamiento en paralelo, MPI es el estándar formal. Trabajar con un estándar formal puede ser una ventaja en algunos contextos.
- *Tolerancia a fallos*: PVM es tolerante a fallos en el sentido de que si un nodo falla los demás pueden seguir adelante con la ejecución del programa. Este mecanismo de recuperación no existe en MPI-1.
- *Heterogeneidad*: La máquina virtual hace que PVM funcione muy bien en clusters heterogéneos (aquel construido con computadoras distintas entre sí). MPI espera que el hardware sea homogéneo, una máquina MPP o, al menos, un Beowulf de computadoras idénticas.
- *Rendimiento*: Cabe esperar que MPI sea de mejor rendimiento que PVM, especialmente en máquinas MPP.
- *Base instalada*: Al ser bastante posterior, el número de programas que hacen uso de MPI es significativamente inferior a los que emplean PVM.
- *Control de ejecución y gestión de recursos*: PVM especifica cómo ejecutar los programas independientemente del hardware y del sistema operativo. Asimismo, incluye un mecanismo para la gestión de recursos. MPI ni lo uno ni lo otro. Se espera que las implementaciones de MPI-2 haya incorporado esta funcionalidad.
- *Topología*: con MPI es posible especificar la topología de conexión entre elementos de proceso, lo que permite optimizar las comunicaciones. PVM no dispone de esta capacidad.
- *API*: Al ser posterior a PVM, MPI aprendió de sus errores y ofrece un API más sencilla y eficiente.

- *Interoperabilidad:* Distintas aplicaciones de MPI pueden no cooperar correctamente entre sí. Más aún, MPI no fuerza la interoperabilidad entre lenguajes, de tal forma que un programa en C puede no poder comunicarse con otro en Fortran. PVM carece de estos problemas.

- *Complejidad:* PVM muestra un propósito definido en todo su diseño. En la especificación de MPI 2 se han incluido toda una serie de capacidades al margen del paso de mensajes, como acceso remoto a memoria o un sistema paralelo de entrada/salida. Estas cosas son útiles, pero hacen de MPI una librería compleja.

- *Flexibilidad en las comunicaciones:* el conjunto de funciones de comunicación de MPI es más rico que el de PVM.

En conclusión, teniendo en cuenta la lista anterior, se ve que PVM se encuentra en mayor cantidad de implementaciones y es muy común encontrarlo en la mayoría de los casos de implementaciones de clusters Beowulf. En particular, su mejor comportamiento en entornos heterogéneos y su tolerancia a fallos puede que hagan de PVM la mejor alternativa para la programación de clusters. Por otro lado, MPI aparece como un estándar —lo cual garantiza cierta continuidad en el tiempo y que todas las implementaciones sean intercambiables—. Además, MPI-2 estandariza la generación de procesos dinámicamente lo cual agregaría a MPI la funcionalidad básica de una implementación de PVM. En la práctica, muchas implementaciones utilizan PVM y MPI combinadas, PVM para la administración de procesos y nodos del cluster y MPI para la comunicación entre procesos o paso de mensajes.

3.5 VENTAJAS DE PVM SOBRE SUPERCOMPUTADORAS

La máquina paralela virtual es una máquina que no existe, pero un API apropiado permite programar como si existiese. El modelo abstracto que permite usar el API de la PVM consiste en una máquina multiprocesador completamente escalable (es decir, que se pueda aumentar y disminuir el número de procesadores *en caliente*). Para ello, se va a ocultar la red que se este empleando para conectar las máquinas, así como las máquinas de la red y sus características específicas. Este planteamiento tiene numerosas ventajas respecto a emplear un supercomputador, de las cuales, las más destacadas son:

- *Precio.* Así como es mucho más barata una computadora paralela que la computadora tradicional equivalente, un conjunto de maquinas de mediana o baja potencia es muchísimo más barato que una maquina paralela de potencia equivalente. Al igual que ocurrirá 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 hacer de que sean necesarios más equipos de pequeña potencia que los teóricos para igualar el rendimiento. Sin embargo, aun teniendo esto en cuenta, la solución es mucho más barata. Además, al no ser la PVM una solución que necesite de máquinas dedicadas (es decir, el *daemon* de PVM corre como un proceso más), se puede emplear en el proceso los tiempos muertos de los procesadores de todas las máquinas de nuestra red a las que se tenga acceso. Por ello, si ya se tiene una red Unix montada, el costo de tener una supercomputadora paralela va a ser cero ya que se dispone de las máquinas,

no se tiene que comprar nada nuevo, y además la biblioteca PVM es software libre, por lo que no hay que pagar para usarla.

- *Disponibilidad.* Todo centro de cálculo tiene un mínimo de una docena de máquinas arrumbadas en una esquina, y que nadie sabe qué hacer exactamente ya con ellas. Con esa docena que hace seis años que ya no corren ni la última versión del Word para Windows, podemos instalar Linux, la PVM y añadirlo al supercomputador paralelo virtual que conforma las máquinas que ya tendríamos en red.
- *Tolerancia a fallos.* Si por cualquier razón falla uno de los nodos que conforman nuestra PVM y el programa que la usa está razonablemente bien hecho. La aplicación puede seguir funcionando sin problemas. En un caso, en el que la aplicación va a estar corriendo durante meses, es crítico que la aplicación sea tolerante a fallos. Siempre hay alguna razón por la que alguna máquina puede fallar, y la aplicación debe continuar haciendo los cálculos con aquel hardware que continúe disponible.
- *Heterogeneidad.* Se puede crear una máquina paralela virtual a partir de computadoras de cualquier tipo. La PVM 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. Este último punto es de extrema importancia, ya que el principal problema que se tendría en los *sockets* sería la programación de rutinas de conversión de formato de datos entre todas las computadoras de la red, puesto que la codificación, tanto de enteros como de flotantes, puede ser distinta. Por último, una máquina paralela en una 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 la PVM como un conjunto de máquinas secuenciales.
- *Disponibilidad.* La disponibilidad de la PVM es completa. Se encuentra con facilidad para PowerPC con AIX, Sun con Solaris y PC 80x86 con Linux.

3.6 GRANULARIDAD DE UN PROGRAMA

Es el tamaño de las piezas en que se divide una aplicación. Dichas piezas puede ser una sentencia de código, una función o un proceso en sí que se ejecutarán en paralelo. La granularidad es categorizada en paralelismo de grano fino y paralelismo de grano grueso.

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

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

Un paralelismo de grano grueso en el nivel más alto se presenta cuando en la aplicación se detectan tareas independientes y estas se ejecutan como procesos independientes en más de un procesador. Este esquema es común en las aplicaciones de Productor/Consumidor, Lec-

tor/Escritor, Maestro/Esclavo y Cliente /Servidor. En los modelos de Memoria Distribuida (paso de mensajes) solo se implementa paralelismo de grano grueso.

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

Creado en el verano de 1989 por el Oak Ridge National Laboratory y desarrollado, posteriormente, junto con la Universidad de Tennessee en los EUA. PVM es una biblioteca de envío de mensajes, totalmente libre, capaz de trabajar en redes homogéneas y heterogéneas, que hace uso de los recursos existentes en algún centro de trabajo para poder construir una máquina paralela de bajo costo, obteniendo su mejor rendimiento en "horas muertas".

PVM maneja transparentemente el ruteo de todos los mensajes, conversión de datos y calendarización de tareas a través de una red de arquitecturas heterogéneas. PVM está diseñado para conjuntar recursos de cómputo y proveer a los usuarios de una plataforma paralela para correr sus aplicaciones, independientemente del número de computadoras distintas que utilicen y dónde se encuentren localizadas. El modelo computacional de PVM es simple y además muy general. El usuario escribe su aplicación como una colección de tareas cooperativas. Las tareas accesan los recursos de PVM a través de una biblioteca de rutinas. Estas rutinas permiten la inicialización y terminación de tareas a través de la red, así como la comunicación y sincronización entre tareas.

Los constructores de comunicación incluyen aquellos para envío y recepción de estructuras de datos así como primitivas de alto nivel, tales como emisión, barreras de sincronización y sumas globales.

3.7 TÉCNICAS BÁSICAS DE PROGRAMACIÓN CON PVM

El desarrollo de aplicaciones en PVM sigue el paradigma tradicional de programación de multiprocesadores de memoria distribuida. Las técnicas básicas son similares en aspectos logísticos de programación y en el desarrollo de algoritmos. Sin embargo existen diferencias significativas en términos de:

- La administración de tareas, especialmente en cuanto a la creación, nombramiento y direccionamiento de los procesos dinámicos.
- Fases de inicialización anteriores al cómputo actual.
- Opciones de granularidad.
- Heterogeneidad.
- Paradigmas comunes de programación paralela.

Al usar PVM en el cómputo paralelo lo podemos describir desde tres puntos de vista fundamentales, basados en la organización del procesamiento de las tareas. Estos son modelo Crowd, modelo de Árbol (Tree) y modelo Híbrido.

3.7.1 Modelo Crowd. El modelo más común para las aplicaciones PVM puede ser llamado procesamiento "Crowd": Una colección de procesos más o menos relacionados, general-

mente ejecutando el mismo código, llevan a cabo procesamiento en diferentes porciones de la carga de trabajo, usualmente involucrando el intercambio periódico de resultados intermedios. Este paradigma puede ser subdividido en dos categorías:

Modelo Maestro – Esclavo. En el modelo Maestro-Esclavo un programa de control separado llamado maestro es el responsable de la expansión de los procesos, inicialización, colección y desplegado de los resultados y quizá de tomarles el tiempo a las tareas, mientras que los programas esclavo llevan a cabo el procesamiento involucrado en la tarea y su carga de trabajo es alojada por el maestro o por ellos mismos.

Modelo de Un-Solo-Nodo (Node-Only). Es donde se ejecutan múltiples estancias de un programa en un solo nodo, con un proceso encargado de las responsabilidades independientes al procesamiento pero que contribuye con el.

3.7.2 Modelo de Árbol (Tree). Este es otro modelo que soporta PVM, aquí los procesos son expandidos (dinámicamente conforme avanza el procesamiento) en forma de árbol estableciendo una relación padre- Hijo. Este paradigma aunque es menos usado, en aplicaciones donde se desconoce desde el principio la carga de trabajo es mas adecuado, por ejemplo en algoritmos de búsqueda y algoritmos recursivos “divide y vencerás”.

3.7.3 Modelo Híbrido. Este modelo puede ser visto como una combinación de los anteriores. Este paradigma posee una estructura de expansión arbitraria: esto es en cualquier punto durante la ejecución de la aplicación, la estructura de la relación de proceso puede generar una grafica arbitraria y cambiante.

Estas tres clasificaciones están hechas en base a la relación de los procesos aunque frecuentemente también corresponden a las topologías de comunicación, mas sin embargo, en las tres es posible que los procesos interactúen y se sincronicen entre si. La elección del modelo depende de la aplicación y se debe elegir la que mejor se adapte a la estructura natural del programa paralelizado.

3.8 FUNCIONAMIENTO DE PVM

La arquitectura de la pvm se compone de dos partes. La primera parte es el daemon, llamado *pvmd*. En la versión actual de la PVM -la 3-, el nombre es *pvmd3*. El daemon ha de estar funcionando en todas las máquinas que vayan a compartir sus recursos computacionales con la máquina paralela virtual. A diferencia de otros daemons y programas del sistema, el daemon de la PVM puede ser instalado por el usuario en su directorio particular (de hecho, la instalación por defecto es así). Esto nos va a permitir hacer supercomputación como usuarios, sin tener que discutir con el administrador de la red que programas vamos a poder ejecutar. Una vez que un usuario (o superusuario) instaló en un directorio la PVM, todos los usuarios pueden hacer uso de esa instalación con el requisito de que el directorio donde esté instalada la PVM sea de lectura al usuario que quiera hacer uso de ella.

En muchos centros de computación, el administrador prefiere instalar él mismo la PVM; con lo que, además de evitar que un usuario pueda borrarla sin consultar a los demás, va a permitir que todos los usuarios tengan la PVM instalada por defecto; y, lo que es más im-

portante, que el administrador puede determinar el valor de *nice* (prioridad del daemon) con el que va a ser lanzado el daemon pvmd3 y así, si este valor de nice es lo suficientemente alto, permite que la máquina ejecute la PVM solamente en los momentos ociosos.

Este daemon pvmd3 es el responsable de la máquina virtual de por sí, es decir, de que se ejecuten los programas para la 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 la 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 la PVM. Esto va a evitar el problema que se con los Sockets ya que se tenía que hacer una rutina de codificación y otra de decodificación, al menos, por cada arquitectura distinta del sistema.

Cada usuario, arrancará el daemon como si de un programa normal se tratase, 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 ha de enlazar a la biblioteca para poderse ejecutar después. Se tendrán tres ficheros de bibliotecas, la *libpvm3.a* (biblioteca básica en C), la *libgpvm3.a* (biblioteca de tratamiento de grupos) y la *libfpvm3.a* (biblioteca para Fortran).

Un programa para PVM va a ser un conjunto de tareas que cooperan entre si. Las tareas se van a intercambiar información empleando paso de mensajes. La PVM, de forma transparente al programador, va a ocultar las transformaciones de tipos asociadas al paso de mensajes entre máquinas heterogéneas. Toda tarea de la PVM puede incluir o eliminar máquinas, arrancar o parar otras tareas, mandar datos a otras tareas o sincronizarse con ellas.

Cada tarea en la PVM tiene un número que la identifica unívocamente, denominado **TID** (Task Identification Number). Es el número al que se mandan los mensajes habitualmente. Sin embargo, no es el único método de referenciar una tarea en la PVM. Muchas aplicaciones paralelas necesitan hacer el mismo conjunto de acciones sobre un conjunto de tareas. Por ello, la PVM incluye una abstracción nueva, *el grupo*. Un grupo es un conjunto de tareas a las que se pueden referir con el mismo código, el identificador de grupo. Para que una tarea entre o salga de un grupo, basta con avisar de la salida o entrada al grupo. Esto va a dotar de un mecanismo muy cómodo y potente para realizar programas empleando modelos *SIMD* (Single Instruction, Multiple Data), en el que se han de dividir los datos en muchos datos pequeños que sean fáciles de tratar, y después codificar la operación simple y replicarla tantas veces como datos unitarios se tengan de dividir el problema. Para trabajar con grupos, además de enlazar la biblioteca de la PVM (*libpvm3.a*) se tiene que enlazar también la de grupos (*libgpvm3.a*).

Habitualmente para arrancar un programa para la PVM, se lanzará manualmente desde un ordenador contenido en el conjunto de máquinas una tarea madre. La tarea se lanzará con el comando *spawn* desde un monitor de la máquina virtual, que a su vez se activará 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), bien desde alguna subrutina invocada por

ella. Para lanzar 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.

Para evitar el engorro de andar realizando 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 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 si. Esto no es trivial, ya que empleando UDP puede que se envíen 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, tiene el inconveniente que establecer las conexiones entre nodos empleando TCP supone, si se tienen n nodos, se realizaran un mínimo de $(n)(n-1)$ conexiones TCP activas. Provocando esto que hasta para números ridículos de n no hayan mas puertos por éste planteamiento. Establecer conexiones TCP entre procesos en lugar de entre nodos es peor todavía, por las mismas razones que en el caso de los nodos.

La comunicación de las tareas con el daemon se hace empleando TCP. Esto se debe a que, al ser comunicaciones locales, la carga derivada de la apertura y cierre de un canal es muy pequeño. Además, no se van a tener tantas conexiones como en el caso de la conexión entre daemons, ya que las tareas no se conectan entre sí ni con nada fuera del nodo, por lo que sólo hablan directamente con su daemon. Esto determina que serán n conexiones TCP, que sí es una cifra razonable.

La recepción de los mensajes se puede hacer mediante primitivas bloqueantes, no bloqueantes o con un tiempo máximo de espera. La PVM dotará de primitivas para realizar los tres tipos de recepción. En principio serán más cómodas las bloqueantes, ya que darán un mecanismo de sincronización bastante cómodo. Las de tiempo máximo de espera serán útiles para trabajar con ellas como si fuesen bloqueantes, mas dando soporte al hecho de que puede que el que tiene que mandar el mensaje se haya colgado. Por último, la recepción de mensajes mediante primitivas no bloqueantes hace de la sincronización un dolor de cabeza. De cualquier forma, en los tres casos anteriormente citados la misma PVM se encargará de

decir cuándo una tarea acabó. Para informarse de lo que pasa, emplea un mecanismo de eventos asíncronos.

La 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 se trabaja 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 se desea 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 se puede invocar la función; bien directamente si el lenguaje lo permite, bien haciendo alguna pequeña rutina para adaptar el tipo de los datos, el formato de llamada a función o cualquiera de las restricciones que imponga el lenguaje que se emplee para invocar funciones en C.

Se ha de destacar que toda función en C *pvm_ alguna cosa* tiene como equivalente en Fortran *pvmfalguna cosa*, y viceversa.

3.9 FUNCIONAMIENTO DE XPVM

La interfaz gráfica de PVM se denomina XPVM y nos permite analizar los resultados a través de pantallas que hacen más fácil la interpretación de los resultados.

En muchas ocasiones, es muy útil tener una representación gráfica de la configuración de la máquina virtual que se está utilizando, así como una codificación visual de la actividad llevada a cabo en cada host de la máquina virtual, qué mensajes se están enviando, quién los envía y a dónde. La interfaz gráfica de usuario de PVM (XPVM) permite realizar todas estas funciones.

XPVM combina las funciones de la consola básica PVM con un monitor de seguimiento de actividades y un debugger en una interfaz tipo X-Windows. XPVM está escrito en C, usando el toolkit TCL/TK.

Para ejecutar XPVM, hay que asegurarse de que el daemon no está ya corriendo y que no haya ficheros temporales relacionados con PVM.

La consola se compone de varias vistas de tamaño reconfigurable y una serie de ventanas que son utilizados por XPVM para mostrar mensajes de estado o de ayuda (Status y Help). Por defecto, la consola inicialmente muestra la vista de red (Network View) y la vista de representación temporal de tareas (Space-Time).

El menú Hosts permite añadir un nuevo host a la máquina virtual, seleccionado de entre todos los hosts listados en el fichero *.xpvhosts*.

En este caso se van a añadir varios hosts. Cada vez que se añade uno aparece un nuevo símbolo de host conectado a los símbolos existentes.

A través del menú Tasks-SPAWN pueden lanzarse tareas en cualquiera de los hosts que compone la máquina.

La vista de Representación de Tareas muestra el estado de todas las tareas que se están ejecutando en la máquina virtual en un momento dado. Para que las tareas se muestren, el botón PLAY que se ve en la parte superior de la ventana de visualización de tareas. La visualización puede ser interrumpida o terminada en cualquier momento, utilizando los botones PAUSE y STOP. Una vez detenida la visualización se mover hacia el pasado o el futuro de las tareas utilizando los botones REWIND y FORWARD.

La vista de Representación de Tareas se compone de dos ventanas. La ventana izquierda contiene el nombre del host y el de la tarea ejecutada en el mismo. Las tareas aparecen ordenadas alfabéticamente. El número de tareas mostradas en una ventana puede aumentarse utilizando el botón de compresión de tareas que aparece a la izquierda de los botones anteriormente mencionados.

La ventana derecha muestra, para cada proceso, el estado de dicha tarea en cada momento, así como líneas rojas que emanan de cada proceso y que corresponden a envíos de mensajes entre procesos. El código de colores muestra el estado del proceso, que puede estar ejecutando tareas propias (verde), rutinas PVM (amarillo) o esperando mensajes (blanco).

El usuario puede recabar información detallada sobre un estado determinado o un mensaje, seleccionando con el botón izquierdo un estado u mensaje. Si se selecciona una barra de tarea, se obtiene su estado así como el tiempo de comienzo y fin de la tarea y la última llamada a PVM que se hubiera generado. Si se selecciona una línea de mensaje, la ventana que aparece mostrará el tiempo de envío y recepción, así como el número de bytes enviado y el identificador de mensaje.

La representación de tareas en la ventana derecha puede ampliarse o reducirse (zooming) utilizando simultáneamente los dos botones del ratón (simula el botón central de un ratón de tres botones) y el botón derecho, respectivamente.

Existe una ventana de salida de tareas, accesible a través del menú VIEWS, que actúa como salida standard para los procesos.

Finalmente, existe una ventana de utilización de recursos, accesible también a través del menú VIEWS. Esta ventana, que está sincronizada con la ventana de representación de tareas, muestra el número de tareas que están ejecutándose, ejecutando funciones PVM o en espera en cada momento.

3.10 INSTALACIÓN Y CONFIGURACIÓN DE PVM

Para la instalación se deberán seguir los pasos indicados en la propia descripción del PVM. El resultado del ejercicio será un diario de acciones que permitirán repetir fácilmente la instalación en cualquier otro sistema. La instalación y configuración de PVM y XPVM se realizarán sobre las máquinas del cluster.

3.10.1 Pasos para la instalación de PVM

1. Se deberán descargar los códigos fuente de PVM desde la página oficial de PVM en Oak Ridge Laboratorios (<http://www.epm.ornl.gov/pvm>).
2. El fichero pvm3.4.4.tgz se descargará y desempaquetará (`tar -xzf pvm3.4.4.tgz`) sobre el directorio \$HOME de cada máquina que integra el cluster.
3. Una vez desempaquetado, se editará el fichero de arranque (.bashrc) siguiendo los pasos descritos en el apartado siguiente.
4. La instalación se completará con la compilación y ejecución de los siguientes ejemplos incluidos en el código de PVM:

- Ejemplo hello / hello_other
- Ejemplo master1 slave1

3.10.2 Configuración del Entorno PVM y Compilación de Ejemplos

A continuación se detallan una serie de pasos necesarios para ajustar el funcionamiento sobre cada uno de los nodos. Los pasos a seguir para iniciar el entorno son:

1. *Ajustar las variables de entorno PVM.* Introducir en el fichero .bashrc (.profile para el shell ksh) las variables de trabajo PVM. (Verificar que el directorio final de instalación del pvm sea =/usr/share/pvm3, para ello, comprobar la existencia de este directorio).

El fichero .bashrc (que se encuentra en el directorio \$HOME) debe contener las siguientes entradas. Si no están presentes, deberán añadirse al final del fichero. Para actualizar el fichero .bashrc, utilizar cualquier editor de texto (pico, vi, nano, etc.).

```
export PVM_ROOT=/usr/share/pvm3
export PVM_ARCH=LINUX
export PATH=$PATH:$PVM_ROOT/lib:$HOME/pvm3/bin/$PVM_ARCH
export PVM_DPATH=$PVM_ROOT/lib/pvmd
export PVM_RSH=/usr/bin/ssh
export MANPATH=$MANPATH:$PVM_ROOT/man}
```

Salir de la sesión (logout) y volver a entrar (login) ejecutando la orden: `pvm` Si todo ha ido bien, el sistema presentará el prompt de pvm:

pvm>

Para salir de la consola de la PVM teclear el comando `halt`:

pvm>halt

2. *Crear un directorio propio de trabajo.* Es necesario crear un subdirectorio de trabajo donde estén tanto los programas que se realicen como los ejecutables. Para ello hay que ejecutar:

```
$ mkdir -p $HOME/pvm3/bin/LINUX
```

3. Copiar los ejemplos de PVM a nuestro directorio:

```
$ cp -r $PVM_ROOT/examples $HOME/pvm3/examples
```

4. *Prueba de los ejemplos.* Se trata de acceder a los ejemplos de pvm, para aprender a compilar y ejecutar:

```
$ cd $HOME/pvm3/examples
```

Modificar el fichero de Compilación (Makefile) para poder hacer referencia a los directorios actuales. Para ello se debe editar este fichero y modificar las siguientes líneas:

```
#
# $Id: Makefile.aimk,v 1.16 1998/11/20 19:46:41 pvmsrc Exp $
#
# Makefile.aimk for PVM example programs.
#
...
#
SHELL = /bin/sh
# CAMBIO 1
#PVMDIR = ../..
# se sustituye por
PVMDIR = /usr/share/pvm3/
#CAMBIO 2
#SDIR = $(PVMDIR)/examples
#se sustituye por
SDIR = $(HOME)/pvm3/examples
# CAMBIO 3
#BDIR = $(PVMDIR)/bin
se sustituye por
BDIR = $(HOME)/pvm3/bin
XDIR = $(BDIR)/$(PVM_ARCH)
CFLOPTS = -g
CFLAGS = $(CFLOPTS) -I$(PVMDIR)/include $(ARCHCFLAGS)
##### Need to select some architecture-specific libraries to link
tasks
# The default setup is for workstations
### Select PVMLIB depending on architecture.
# For MPPs (PGON and AIX4SP2/SP2MPI) select:
# PVMLIB = -lpvm3pe
# For Workstations and shared memory architectures select:
# PVMLIB = -lpvm3
PVMLIB = -lpvm3
```

Una vez modificado, se compila el ejemplo hello hello_other:

```
$ aimk hello hello_other
```

Ejecutar:

```
$ hello
```

5. El sistema PVM permite el uso de varias máquinas como si se tratara de un único computador. Para poder ejecutar comandos de forma remota, PVM utiliza 'ssh'. Las conexiones vía 'ssh' son seguras y la información viaja encriptada. Sin embargo, esto requiere el suministro de la contraseña en cada operación que se realiza, lo que resulta bastante ineficiente.

Para ello se utilizará el agente de claves de ssh. Para poder utilizarlo hay que crear una clave DSA en cada máquina. Como todas comparten el directorio 'home' basta con crearla en una máquina.

```
$ ssh-keygen -t dsa
```

A la pregunta del 'passphrase' se puede suministrar una clave adicional o no. Por comodidad se recomienda NO añadir una clave adicional. Tras esta operación, se generan un par de ficheros (id_dsa.pub y id_dsa) en el directorio \$HOME/.ssh. El fichero id_dsa.pub debe añadirse al final del fichero .ssh/authorized_keys (cat id_dsa.pub >> authorized_keys) para que la clave quede activada.

A partir de este momento, es posible activar el agente de claves con las operaciones:

```
$ ssh-agent bash
```

```
$ ssh-add
```

6. Hasta el momento, se dispone de una única computadora en la máquina paralela. Lógicamente, se desean añadir más computadoras para poder realizar procesamiento paralelo. Para ello, se debe utilizar el comando *add* desde la consola. Para comprobar el correcto funcionamiento del sistema, añadir una máquina con el comando add y repetir la ejecución de los ejemplos anteriores. Probar con varias máquinas.

Ejemplo:

```
pvm> add node7
```

A continuación, comprobar que la máquina ha sido correctamente añadida mediante el comando conf.

```
pvm> conf
```

Finalmente, se puede eliminar alguna máquina con el comando *delete*.

pvm> delete node7

7. Al terminar la sesión se deberá cerrar pvm con el comando *halt*. Este comando termina todos los demonios de todas las máquinas conectadas. Si se olvida hacer halt y cualquiera de las máquinas que se tienen conectadas (o la propia) se apaga, no se podrá poner en marcha el pvm.
8. Para no tener que manualmente añadir todas las máquinas, podemos crear un fichero de configuración que contenga los nombres de las máquinas que se quieran añadir a la máquina virtual y poner en marcha el pvm con este fichero. Un fichero ejemplo se muestra a continuación:

node1	//Mestro 148.228.xx.xx
node2	//Nodo2 S.O. RedHat XX 148.228.xx.xx
node3	//Nodo3 S.O. RedHat XX 148.228.xx.xx
node4	//Nodo4 S.O. RedHat XX 148.228.xx.xx
node5	//Nodo5 S.O. RedHat XX 148.228.xx.xx
node6	//Nodo6 S.O. Suse XX 148.228.xx.xx
node7	//Nodo7 S.O. Suse XX 148.228.xx.xx
node8	//Nodo8 S.O. Suse XX 148.228.xx.xx
node9	//Nodo9 S.O. Suse XX 148.228.xx.xx
node10	//Nodo10 S.O. Fedora XX 148.228.xx.xx
node11	//Nodo11 S.O. Fedora XX 148.228.xx.xx
node12	//Nodo12 S.O. Fedora XX 148.228.xx.xx
node13	//Nodo13 S.O. Fedora XX 148.228.xx.xx
node14	//Nodo14 S.O. Fedora XX 148.228.xx.xx
node15	//Nodo15 S.O. Fedora XX 148.228.xx.xx

Para poder activar esta configuración, el demonio de pvm debe de estar inactivo y ejecutarlo desde la línea de comandos:

\$ pvm hostfile

Siendo hostfile el fichero anterior.

Una vez que se han efectuado todos estos pasos, ya se pueden empezar a desarrollar los programas a ejecutar.

3.11 INSTALACIÓN Y CONFIGURACIÓN DE XPVM

1. Descargar los archivos tcl y tk en este caso la versión de ambos es la 8.4.9; desde la pagina <http://tcl.sourceforge.net>
2. Los dos archivos anteriores se descomprimen y se ubican en los directorios:
\$HOME/usr/share/tcl8.4.9 **para tcl**
\$HOME/usr/share/tk8.4.9 **para tk**

3. Dentro de cada directorio se ejecutaran las siguientes instrucciones:

```
$./configure  
$make  
$make install
```

4. Editar el fichero .bashrc (que se encuentra en el directorio \$HOME) añadiendo al final del fichero las siguientes entradas:

```
export TCL_LIBRARY=/usr/local/lib/tcl8.4  
export TCL_LIBRARY=/usr/local/lib/tk8.4
```

Salir y entrar nuevamente a la sesión para que se puedan generar las variables de entorno.

5. Editar “xpvm/src/Makefile.aimk” que se encuentra dentro del directorio de pvm

```
SHELL          =      /bin/sh  
# CAMBIO 1  
PVMVERSION     =      -DUSE_PVM_33  
# Se sustituye por:  
PVMVERSION     =      -DUSE_PVM_34  
  
PVMGLIB        =  
CELOPTS       =      -O2  
CC             =      gcc  
CFLAGS        =      $(CFLOPTS) $(PVMVERSION) -DIMA_$(PVM_ARCH)  
$(ARCHFLAGS)  
LFLAGS        =      $(ARCHFLAGS)  
AIMK          =      $(PVM_ROOT)/lib/aimk  
  
#  
#Tcl/Tk Settings  
#  
#CAMBIO 2  
TCKTKHOME     =      /usr/local/lib  
TCLINCL       =      -I$(TCKTKHOME)/tcl/generic  
TKINCL        =      -I$(TCKTKHOME)/tk/generic  
TCLLIBDIR     =      -L$(TCLTKHOME)/tcl/unix/$(PVM_ARCH)  
TKLIBDIR      =  
L$(TCLTKHOME)/tk/unix/$(PVM_ARCH)  
# Se sustituye por:  
TCKTKHOME     =      /usr/share  
TCLINCL       =      -I$(TCKTKHOME)/tcl8.4.9/generic  
TKINCL        =      -I$(TCKTKHOME)/tk8.4.9/generic  
TCLLIBDIR     =      -L$(TCLTKHOME)/tcl8.4.9/unix/$(PVM_ARCH)  
TKLIBDIR      =      -L$(TCLTKHOME)/tk8.4.9/unix/$(PVM_ARCH)  
  
#Tcl / Tk 8.2  
#CAMBIO 3  
TCLLIB        =      -ltcl  
TKLIB         =      -ltk  
# Se sustituye por
```

```

TCLLIB      =      -ltcl8.4
TKLIB       =      -ltk8.4

#X11
XINCL       =      -I/usr/X11R6/incluye
#CAMBIO 4
XLIBDIR     =      -L/usr/X11R6/lib
# Se sustituye por
XLIBDIR     =      -L/usr/X11R6/lib64

XLIB        =      -lX11 -lXpm

# Non-Shared Library Systems
# CAMBIO 5
SHLIB       =
# Se sustituye por
# SHLIB     =
# Linux, BSD, SunOS Shared Library Systems
# CAMBIO 6
# SHLIB     =      -ldl
# Se sustituye por
SHLIB       =      -ldl
#IBM Shared Library Systems
# SHLIB     =      -lld
# HP, Some Linux Shared Library Systems
# SHLIB     =      -ldld
.
.
.

```

Una vez modificado el archivo se procede a hacer un make en el directorio:

\$HOME/usr/share/pvm3/xpvm

6. Ejecutar:

\$ xpvm

3.12 COMANDOS FUNDAMENTALES DEL PROGRAMA PVM

El programa PVM corresponde al intérprete de comandos de la 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 la PVM, elimina el *daemon* de forma ordenada y sale del programa pvm. Es la forma de salir *en condiciones*, y mucho mejor que matar el *daemon* con kill.

- **help:** lista los comandos del programa. Tremendamente útil en los momentos de desesperación.
- **id:** imprime el TID de la consola.
- **jobs:** genera un listado de los trabajos en ejecución.
- **kill:** mata un proceso de la PVM.
- **mstat:** muestra el estado de una máquina de las pertenecientes a la PVM.
- **pstat:** muestra el estado de un proceso de los pertenecientes a la PVM.
- **reset:** inicializa la máquina. Eso supone matar todos los procesos de la 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 que se han trazado.
- **alias:** define un alias predefinido, es decir, un atajo para teclear un comando.
- **unalias:** elimina un alias predefinido.
- **versión:** imprime la versión usada de la 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, en cualquier número, y en cualquier máquina asociada a una máquina paralela virtual determinada.

3.13 INTERFAZ DE USUARIO DE PVM

En PVM todas las tareas están identificadas por un número entero que provee el demonio de inicialización del PVM (el programa local *pvm*). En las descripciones futuras éste identificador de tareas será llamado **TID**. Es similar al identificador de procesos (PID) usado en el sistema operativo **UNIX** y asume ser opaco para el usuario, es decir, el valor del TID no tiene significado para él. De hecho, PVM codifica información en el TID para su uso interno únicamente.

Todas las rutinas del PVM están escritas en C. Aplicaciones de Fortran pueden llamar a éstas rutinas a través de una interfaz de Fortran 77 proveída por PVM. Esta interfaz traduce argumentos, los cuales son pasados por referencia en Fortran a sus valores si es necesario, por las rutinas de C.

El modelo de comunicación de PVM asume que cualquier tarea puede enviar un mensaje a cualquier otra tarea en PVM y que no hay límite a su tamaño o número de tales mensajes. Mientras que todos los anfitriones (hosts) tienen limitaciones de memoria física que limita espacio potencial de buffer, el modelo de comunicación no se restringe a las limitaciones de una máquina en particular y asume que hay suficiente memoria disponible. El modelo de comunicación PVM provee funciones de envío asíncrono de bloques, funciones para recibir de forma asíncrona bloques de información, recibo de particionado en bloques. Esto es, el envío de bloque regresa en cuanto el buffer de envío se encuentra libre para ser rehusado,

envío asincrónico no depende de un llamado del recibo a una pareja recibo antes de poder el envío regresar.

Un recibo no particionado inmediatamente regresa ya sea con datos o una bandera indicando que los datos aun no han llegado, mientras que un recibo particionado regresa únicamente cuando los datos se encuentran en el buffer de recibo.

El modelo de PVM garantiza que el orden de los mensajes es preservado. Si la tarea 1 envía un mensaje A a la tarea 2, después la tarea 1 envía el mensaje B a la tarea 2, el mensaje A llegará a la tarea 2 antes que el mensaje B.

Los buffers de mensajes son asignados dinámicamente. El tamaño máximo del mensaje que se puede enviar o recibir es limitado únicamente por la cantidad de memoria disponible de un anfitrión dado.

Control de Proceso

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

La rutina *pvm_mytid()* regresa el TID de este proceso y puede ser llamado varias veces. Prepara éste proceso en PVM si ésta es la primer llamada de PVM.

```
int info = pvm_exit( void )
```

La rutina *pvm_exit()* le dice al pvmd local que éste proceso se retira de PVM. Esta rutina no mata el proceso, la cual puede seguir en ejecutando tareas al igual que cualquier proceso en UNIX.

```
int numt=pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)
```

La rutina *pvm_spawn()* empieza *ntask* copias de un archivo de tareas ejecutables en la máquina virtual. *argv* es un apuntador a un arreglo de argumentos a procesar con el final del arreglo especificado con **NULL**. Si *task* no tiene argumentos, entonces *argv* es **NULL**. El argumento *flag* es usado para especificar las siguientes opciones:

Valor	Opción	Descripción
0	PvmTaskDefault	PVM elige en donde reproducir procesos.
1	PvmTaskHost	Aquí, el argumento <i>where</i> es un anfitrión en particular a reproducir la tarea.
2	PvmTaskArch	En este caso el argumento <i>where</i> es un PVM_ARCH a reproducirse
4	PvmTaskDebug	Inicializa tareas bajo un depurador
8	PvmTaskTrace	Un seguimiento de datos es generado

16	PvmMppFront	Se inicializa tareas en MPP front-end.
32	PvmHostCompl	Es el complemento de anfitrión definido en <i>where</i> .

De regreso, *numt* es un conjunto de tareas que lograron reproducirse o un código de error si ninguna tarea se pudo ejecutar. Si tareas fueron comenzadas, entonces *pvm_spawn()* regresa un vector de los tid's de las tareas reproducidas; y si algunas tareas no pudieron ser ejecutadas, los códigos de error correspondiente son puestos en los ultimos *ntask - numt* posiciones del vector.

```
int info = pvm_kill( int tid )
```

La rutina *pvm_kill()* mata alguna otra tarea PVM identificada por el TID. Esta rutina no está diseñada para matar a la tarea llamada, en la cual deberá ser acompañada llamando a *pvm_exit()* seguido de *exit()*.

```
int info = pvm_catchout( FILE *ff )
```

Por omisión PVM escribe el *stderr* y *stdout* de las tareas reproducidas al archivo */tmp/pvml.*. La rutina *pvm_catchout* causa a la tarea llamada pescar la salida de tareas posteriormente reproducidas. Caracteres impresos en *stdout* o en *stderr* en tareas hijas son colectadas por los pvmds y enviados en mensajes de control a la tarea padre, en la cual va a marcar cada línea y agregarla al archivo especificado (en C) o a la salida estándar (en Fortran).

Información

```
int info = pvm_parent( void )
```

La rutina *pvm_parent()* regresa el TID del proceso que reprodujo esta tarea o el valor de *PvmNoParent* si no fue creado por *pvm_spawn()*.

```
int info = pvm_tidtohost( int tid )
```

La rutina *pvm_tidtohost()* regresa el TID *dtid* del demonio corriendo en el mismo anfitrión que TID. Esta rutina es útil para determinar en que anfitrión está ejecutándose cierta tarea.

```
int info = pvm_config( int *nhosts, int *narch, struct pvmhostinfo **hostp )
```

La rutina *pvm_config()* regresa información acerca de la máquina virtual incluyendo el número de anfitriones, *nhost*, el número de los distintos formatos de datos, *narch*. *hostp* es un apuntador a un arreglo declarado por el usuario de estructuras *pvmhostinfo*. El arreglo debe ser por lo menos de tamaño *nhost*. De regreso, cada estructura *pvmhostinfo* contiene el *TID* *pvmd*, nombre del anfitrión, nombre de la arquitectura, velocidad relativa de CPU para ese anfitrión en la configuración.

```
int info = pvm_tasks( int which, int *ntask,
```

```
struct pvmtaskinfo **taskp )
```

La rutina *pvm_tasks()* regresa información acerca de las tareas PVM corriendo en la máquina virtual. El entero *which* especifica acerca de cual de las tareas regresar información.

El número de tareas es regresado en *ntask*. *taskp* es un apuntador a un arreglo de estructuras *pvmtaskinfo*. El arreglo es de tamaño *ntask*. Cada estructura *pvmtaskinfo* contiene el TID, pvmd TID, TID padre, bandera de status, y el nombre del archivo reproducido.

Configuración Dinámica

```
int info = pvm_addhosts( char **hosts, int nhost, int *infos )  
int info = pvm_delhosts( char **hosts, int nhost, int *infos )
```

Las rutinas de C agregan o borran un conjunto de anfitriones de la máquina virtual. En fortran agregan o borran un solo anfitrión en la máquina virtual. En la versión de C *info* es devuelto como el valor de anfitriones agregados exitosamente. El argumento *infos* es un arreglo de longitud *nhost* que contiene el código de estado para cada anfitrión individual siendo agregado o borrado. Esto permite al usuario revisar ya sea si únicamente un anfitrión dentro de un conjunto causó un problema en vez de intentar de nuevo agregar o borrar el conjunto entero de anfitriones.

Estas rutinas son usadas algunas veces para configurar la máquina virtual, pero es más común que sean usadas para incrementar la flexibilidad y la tolerancia de fallas de una aplicación grande. Estas rutinas permiten a la aplicación incrementar la potencia de cómputo (agregando anfitriones) si determina que el problema se está poniendo más complejo para resolverse.

Señalamiento

```
int info = pvm_sendsig( int tid, int signum )  
int info = pvm_notify( int what, int msgtag, int cnt, int tids )
```

La rutina *pvm_sendsig()* envía una señal *signum* a otra tarea de PVM identificada por el TID. La rutina *pvm_notify()* solicita a PVM a notificar a quien hace la llamada la detección de ciertos eventos. Las opciones presentes son las siguientes:

- *PvmTaskExit*: notifica si una tarea se retira.
- *PvmHostDelete*: notifica si un anfitrión es borrado (o no responde).
- *PvmHostAdd*: notifica si un anfitrión es agregado.

En respuesta a una solicitud de notificación, cierto número de mensajes son enviados de vuelta por PVM a la tarea que hace la llamada.

Definiendo y obteniendo opciones

```
int oldval = pmv_setopt( int what, int val )
```

```
int val = pvm_getopt( what, val )
```

La rutina *pvm_setopt()* es una función de propósito general que permite al usuario definir u obtener opciones en el sistema PVM. *pvm_setopt()* puede ser usada para definir varias opciones, incluyendo impresión automática de mensajes de error, nivel de depuración, y el método de enrutamiento de comunicación para todas las llamadas posteriores de PVM. *pvm_setopt()* regresa el valor previo del conjunto en *oldval*. *what* puede tener los siguientes valores:

Opción.	Valor.	Descripción.
PvmRoute	1	Póliza de enrutamiento.
PVMDebugMask	2	Enmascaramiento del depurador.
PvmAutoErr	3	Esta opción hace un reporte de error automático.
PvmOutputTid	4	Destino stdout para los hijos.
PvmOutputCode	5	Salida de <i>msgtag</i> .
PvmTraceTid	6	Seguimiento de destino para los hijos.
PvmTraceCode	7	Seguimiento de <i>msgtag</i> .
PvmFragSize	8	Tamaño del fragmento del mensaje.
PvmResvTids	9	Permitir que los mensajes reserven etiquetas y tids.
PvmSelfOutputTid	10	Destino stdout para si mismo.
PvmSelfOutputCode	11	Salida de <i>msgtag</i> .
PvmSelfTraceTid	12	Seguimiento de destino para si mismo.
PvmSelfTraceCode	13	Seguimiento de <i>msgtag</i> .

El uso más popular de *pvm_setopt* es el de activar una ruta de comunicación directa entre las tareas del PVM. Como una regla general, el ancho de banda de comunicación de PVM sobre una red se dobla haciendo la llamada.

```
pvm_setopt( PvmRoute, PvmRouteDirect );
```

La desventaja es que éste método rápido de comunicación no es escalable bajo UNIX; por consiguiente, puede que no funcione si la aplicación involucra más de 60 tareas que se comunican aleatoriamente entre si. Si no funciona, PVM automáticamente hace el cambio de vuelta al método de comunicación por omisión. Puede ser llamado multiples veces durante la aplicación para definir selectivamente las ligas de comunicación directa entre tareas, pero comunmente se llama una sola vez después de la llamada inicial a *pvm_mytid()*.

Envío de mensajes

El envío de mensajes en PVM esta compuesto por tres pasos; primero, un buffer de envío debe ser inicializado con una llamada a *pvm_initsend()* o *pvm_mkbuf()*; segundo, el mensa-

je debe de ser encapsulado a éste buffer cualquier número de veces y en combinaciones de las rutinas *pvm_pk**(); y tercero, el mensaje completo es enviado a otro proceso llamando a la rutina *pvm_send()* o repartición multiple con la rutina *pvm_mcast()*.

Un mensaje es recibido ya sea llamando a una rutina de recibo de bloques particionados o no particionados y después expandir cada uno de los paquetes del buffer de recibo. Las rutinas de recibo pueden estar puestas para recibir cualquier tipo de mensajes, o cualquier mensaje de una fuente específica, o cualquier mensaje con una marca de mensaje especificada, o únicamente mensajes con una cierta marca de mensaje dado de una fuente dada.

Buffers de mensajes

```
int bufid = pvminitsend( int encoding )
```

Si el usuario está usando únicamente un solo buffer (este es el caso típico) entonces *pvm_initsend()* es la única rutina para el buffer que se requiere. Es llamada antes de encapsular un mensaje nuevo en un buffer. La rutina *pvm_initsend* limpia el buffer de envío y crea uno nuevo para encapsular el mensaje nuevo. El esquema de codificación usado para la encapsulación es definido por *encoding*. El identificador del buffer nuevo es devuelto en *bufid*.

Las opciones de codificación son las siguientes:

- *PvmDataDefault*: Codificación XDR es empleada por omisión porque PVM no puede determinar si el usuario va a agregar una máquina heterogénea antes de que el mensaje sea enviado. Si el usuario sabe que el siguiente mensaje va a ser enviado solo a una máquina que conoce el formato nativo, entonces se puede usar la codificación *PvmDataRaw* y ahorrarse costos de codificación.
- *PvmDataRaw*: No se hace codificación alguna. Mensajes son enviados en su formato original. Si el proceso de recibo no puede leer este formato, regresará un error durante la expansión.
- *PvmDataInPlace*: El buffer contiene solo tamaño y apuntadores de los mensajes que se van a enviar. Cuando *pvm_send()* es llamado, los mensajes son leídos directamente de la memoria del usuario.

Las rutinas de mensajes de buffer siguientes son requeridas solo si el usuario desea manejar múltiples buffers de mensaje dentro de la aplicación. Por lo general, múltiples buffers de mensajes no son necesarios para el envío de mensajes entre procesos. Solo hay un buffer de envío activo y uno buffer de recibo activo por proceso en cualquier tiempo dado. El des-arrrollador puede crear varios buffer de mensajes y cambiarse de uno a otro para la encapsulación y envío de datos. Las rutinas encapsulación, envío, recibo, y expansión solo afectan a los buffers activos.

```
int bufid = pvm_mkbuf( int encoding )
```

La rutina *pvm_mkbuf* crea un nuevo buffer de envío vacío y especifica el método de codificación usado para la encapsulación de mensajes. Regresa un identificador de buffer *bufid*.

```
int info = pvm_freebuf( int bufid )
```

La rutina *pvm_freebuf()* desecha el buffer con el identificador *bufid*. Este debe de hacerse después de que un mensaje ha sido enviado y que ya no se ocupará. Llama a *pvm_mkbuf()* para crear un buffer para un mensaje nuevo si lo requiere. Ninguna de estas llamadas son necesarias cuando se hace la llamada a *pvm_initsend()*, la cual ejecuta estas llamadas para el usuario.

```
int bufid = pvm_getsbuf( void )  
int bufid = pvm_getrbuf( void )
```

La función *pvm_getsbuf()* regresa el identificador del buffer de envío activo, mientras que la función *pvm_getrbuf()* regresa el identificador del buffer de recibo activo.

```
int oldbuf = pvm_setsbuf( int bufid )  
int oldbuf = pvm_setrbuf( int bufid )
```

Estas rutinas definen el buffer de envío(o recibo) activo en *bufid*, guardar el estado del buffer anterior, y regresar el identificador del buffer activo previo *oldbuf*.

Si *bufid* es puesto a 0 en *pvm_setsbuf()* o *pvm_setrbuf()*, entonces el presente buffer es guardado y sin buffer activo. Esta característica puede ser utilizada para guardar el presente estado de los mensajes de la aplicación para que las librerías de matemáticas o gráficos que también envían mensajes no interfieran con el estado de los buffers de la aplicación.

Es posible el envío de mensajes sin tener que volver a encapsularlos usando las rutinas de buffer de mensajes. Esto se ilustra en el siguiente fragmento.

```
bufid = pvm_rcv( src, tag );  
oldid = pvm_setsbuf( bufid );
```

Encapsulamiento de datos

Cada una de las siguientes rutinas en C encapsula un arreglo del tipo de datos dado en un buffer de envío activo. Pueden ser llamadas varias veces para encapsular datos en un solo mensaje. No hay límite en la complejidad de los mensajes encapsulados, pero una aplicación debe de expandir los mensajes exactamente como fueron encapsulados.

Los argumentos para cada una de las rutinas es un apuntador al primer dato a ser encapsulado, *nitem* es el número total de datos a encapsular dentro del arreglo, y *STRIDE* es el *STRIDE* a usar en la encapsulación. Un *STRIDE* de 1 significa que un vector contiguo es encapsulado, un *STRIDE* de 2 significa todos los demás datos son encapsulados, y así sucesivamente. Una excepción es *pvm_pkstr()* que por definición encapsula una cadena con un caracter terminador NULL y no requiere del parámetro *nitem* o *stride*.

```
int info = pvm_pkbyte( char *cp, int nitem, int stride )
```

```

int info = pvm_pkcplx( float *xp,  int nitem, int stride )
int info = pvm_pkdcplx( double *zp, int nitem, int stride )
int info = pvm_pkdouble( double *dp, int nitem, int stride )
int info = pvm_pkfloat( float *fp,  int nitem, int stride )
int info = pvm_pkint( int *np,      int nitem, int stride )
int info = pvm_pklong( long *np,    int nitem, int stride )
int info = pvm_pkshort( short *np,  int nitem, int stride )
int info = pvm_pk( char *cp )

```

Todas las variables son pasadas como direcciones *count* y *stride* son especificadas; del caso contrario, a las variables se asume que son valores.

Envío y recibo de datos

```

int info = pvm_send( int tid, int msgtag )
int info = pvm_mcast( int *tids, int ntask, int msgtag )

```

La rutina *pvm_send()* etiqueta el mensaje con un identificador entero *msgtag* y lo envía al proceso identificado por TID.

La rutina *pvm_mcast()* etiqueta el mensaje con un identificador entero *msgtag* y transmite el mensaje a todas las tareas especificadas en el arreglo de tids (excepto a si mismo). El arreglo de los tids es de tamaño *ntask*.

```

int info = pvm_psend( int tid, int msgtag, void *vp, int cnt, int type)

```

La rutina *pvm_psend()* encapsula y envía un arreglo de tipo de datos especificado a la tarea identificada por TID. Los tipos de datos definidos pueden ser cualquiera de los siguientes:

PVM_STR	PVM_FORMAT
PVM_BYTE	PVM_CPLX
PVM_SHORT	PVM_DOUBLE
PVM_INT	PVM_DCPLX
PVM_LONG	PVM_UINT
PVM_USHORT	PVM_ULONG

```

int bufid = pvm_recv( int tid, int msgtag )

```

Esta rutina de recibo de bloques va a esperar hasta que un mensaje con la etiqueta *msgtag* ha llegado de TID. Un valor de -1 en *msgtag* o el TID se empareja con cualquier cosa. Entonces pone el mensaje en un nuevo buffer de recibo que es creado. El buffer de recibo previo es vaciado al menos que haya sido guardado con una llamada a *pvm_setrbuf()*.

```

int bufid = pvm_nrecv( int tid, int msgtag )

```

Si el mensaje solicitado aún no ha llegado, entonces el recibo sin particiones *pvm_nrecv()* regresa *bufid* = 0. Esta rutina puede ser llamada varias veces para el mismo mensaje para verificar si ha llegado o no, mientras ejecuta trabajo útil entre llamadas. Si un mensaje con la etiqueta *msgtag* ha llegado de TID, *pvm_nrecv()* coloca el mensaje en un nuevo buffer de recibo activo (el cual crea) y regresa el identificador del buffer. El buffer de recibo activo previo es vaciado al menos de que haya sido guardado con la llamada a ***pvm_setrbuf()***.

```
int bufid = pvm_probe( int tid, int msgtag )
```

Si el mensaje solicitado aún no ha llegado, entonces *pvm_probe()* regresa *bufid* = 0. Del caso contrario, regresa un *bufid* para ese mensaje, pero no lo recibe. Esta rutina puede ser llamada varias veces para el mismo mensaje para verificar si ha llegado o no, mientras ejecuta trabajo útil entre llamadas. En adición, *pvm_bufinfo()* puede ser llamada con el *bufid* devuelto para determinar información acerca del mensaje antes de recibirlo.

```
int bufid = pvm_trecv( int tid, int msgtag, struct timeval *tmout )
```

Además PVM suministra una versión de recibo con expiración de tiempo. Considere el caso donde un mensaje nunca llegará a su destino (debido a un error o falla); la rutina *pvm_recv* se bloquearía para siempre. Para evitar tales situaciones, el usuario puede retirarse después de un lapso de tiempo fijo. La rutina *pvm_trecv()* permite al usuario especificar el tiempo de espera. Si el tiempo de espera es puesto bien largo, entonces *pvm_trecv* actúa como *pvm_recv*. Si el tiempo es puesto a 0, entonces *pvm_trecv* actúa como *pvm_nrecv*.

```
int info = pvm_bufinfo( int bufid, int *bytes, int *msgtag, int *tid )
```

La rutina *pvm_bufinfo()* regresa *msgtag*, TID fuente, y la longitud en bytes del mensaje identificado por *bufid*. Puede ser usada para determinar la etiqueta y fuente de un mensaje que fue recibido con comodines (wildcards) especificados.

```
int info = pvm_prevcv( int tid, int msgtag, void *vp, int cnt, int type, int *rtid, int *rtag, int *rcnt)
```

La rutina *pvm_prevcv()* combina las funciones de un recibo de bloques y expande el buffer de recibo. No regresa un *bufid*. En su lugar, regresa los valores actuales de *TID*, *msgtag*, y *cnt*.

```
int (*old)() = pvm_recvf( int (*new)() ( int buf, int tid, int tag ))
```

La rutina *pvm_recvf()* modifica el contexto de recibo causado por las funciones de recibo y puede ser empleado para extender PVM.

Expansión de datos

Las siguientes rutinas en C expanden tipos de datos del buffer de recibo activo. En una aplicación deben de corresponder a las rutinas equivalentes de encapsulación en tipo, nú-

mero de bloques, y progreso. *nitem* es un número de elementos del tipo dado a expandir, y *stride* es el progreso.

```
int info = pvm_upkbyte( char *cp, int nitem, int stride )
int info = pvm_upkcplx( float *xp, int nitem, int stride )
int info = pvm_upkdcplx( double *zp, int nitem, int stride )
int info = pvm_upkdouble( double *dp, int nitem, int stride )
int info = pvm_upkfloat( float *fp, int nitem, int stride )
int info = pvm_upkint( int *np, int nitem, int stride )
int info = pvm_upklong( long *np, int nitem, int stride )
int info = pvm_upkshort( short *np, int nitem, int stride )
int info = pvm_upkstr( char *cp )
int info = pvm_unpackf( const char *fmt, ... )
```

La rutina *pvm_unpackf()* usa un tipo de formato similar de expresión printf para especificar que datos expandir y como expandir los datos del buffer de recibo.

Grupos de procesos dinámicos

Las funciones del grupo de procesos dinámicos están construidas en el núcleo de las rutinas de PVM. Una librería separada libgpvm3.a debe de ser encadenada a los programas de los usuarios para hacer uso de cualquiera de las funciones del grupo. El *pvm*d no ejecuta las funciones del grupo. Esta tarea es manejada por el grupo servidor que es automáticamente empezada cuando el primer grupo de funciones es invocada.

Siguiendo la filosofía de PVM, las funciones de grupo fueron diseñadas para ser muy generales y transparentes al usuario. Cualquier tarea de PVM puede unirse o retirarse de cualquier grupo en cualquier instante sin tener que informarle a las demás tareas en los grupos afectados. Las tareas pueden transmitir mensajes a grupos en la cual no pertenecen. En general, cualquier tarea de PVM puede hacer un llamado a cualquiera de las funciones del grupo en cualquier instante dado. Las excepciones son *pvm_lvgroup()*, *pvm_barrier()*, y *pvm_reduce()*, que por su naturaleza requieren que la tarea que llama sea miembro de un grupo especificado.

```
int inum = pvm_joyngroup( char *group )
int info = pvm_lvgroup( char *group )
```

Estas rutinas permiten que una tarea se una o deje un grupo de usuarios. La primer llamada a *pvm_joyngroup()* crea un grupo con el nombre del grupo y coloca a la tarea que llama en este grupo. *pvm_joyngroup()* regresa la instancia del número (*inum*) del proceso de este grupo. Instancias de número tienen un rango de 0 hasta el número de miembros en el grupo menos 1. Una tarea puede unirse a múltiples grupos.

Si un proceso se retira de un grupo y después regresa, el proceso puede recibir una instancia de número diferente. Las instancia de números son ciclados para que una tarea que se une al grupo reciba la instancia de número más bajo disponible.

```
int tid = pvm_gettid( char *group, int inum )
int inum = pvm_getinst( char *group, int tid )
int size = pvm_gsize( char *group )
```

La rutina *pvm_gettid()* regresa el TID de un proceso dado nombre de un grupo y la instancia de número. *pvm_gettid()* permite que dos tareas sin conocimiento de la otra obtenga el TID de la otra con el simple hecho de unirse al grupo. La rutina *pvm_getinst()* regresa la instancia de número dado el nombre del grupo y el TID. La rutina *pvm_gsize()* regresa el número de miembros en el grupo especificado.

```
int info = pvm_barrier( char *group, int count )
```

En la llamada a *pvm_barrier()* el proceso se bloquea hasta *count* miembros del grupo hayan llamado *pvm_barrier()*. En general *count* deberá de ser el total de miembros en el grupo. Una cuenta es necesaria ya que en grupos de procesos dinámicos PVM no puede determinar cuantos miembros pertenecen al grupo en un instante dado.

```
int info = pvm_bcast( char *group, int msgtag )
```

La función *pvm_bcast()* etiqueta el mensaje con un identificador entero *msgtag* y transmite el mensaje a todas las tareas en el grupo específico excepto a si misma (si es que es del mismo grupo). Para *pvm_bcast()* "all tasks" es definido ser todas aquellas tareas que el grupo servidor cree que están dentro del grupo cuando la rutina es llamada. Si una tarea se une al grupo mientras una transmisión, puede que no reciba el mensaje. Si una tarea se retira del grupo mientras una transmisión se lleva a cabo, una copia del mensaje va a ser enviada a la tarea.

```
int info = pvm_reduce( void (*func)(), void *data, int nitem,
int datatype, int msgtag, char *group, int root )
```

La función *pvm_reduce()* realiza operaciones globales aritméticas en el grupo, por ejemplo, global sum o global max. El resultado de la operación de reducción aparece en root. PVM provee cuatro funciones predefinidas que el usuario pueda colocar en *func*. Estas son:

- PvmMax
- PvmMin
- PvmProduct

CAPÍTULO IV

APLICACIONES

A través del paso del tiempo el uso de computadoras ha crecido de manera considerable y la información que actualmente se maneja en los sistemas informáticos es en algunos casos muy difícil de procesar. Para dichos problemas que requieren grandes recursos por la cantidad de información a procesar existen como anteriormente se ha mencionado los cluster que proveen el servicio necesario para la solución de los mismos.

Por este motivo se plantean a continuación dos tipos de problemas, el primero tiene que ver con la simulación del tráfico que se genera en una población en donde dada un área específica se tengan que mover de un lado a otro a dos grupos de individuos. A dicho problema se le ha nombrado *Infierno*; este problema se plantea como un parámetro evolutivo. Y la solución de este en forma paralela pretende primordialmente obtener menores tiempos en su ejecución y observar el caos que se puede generar en un momento determinado por la saturación del espacio ocupado.

El segundo problema tiene que ver con el manejo de datos, donde en un océano de información tenemos que encontrar un dato específico y sus incidencias en caso de existir; este es más bien un problema de búsqueda y se propone el problema de la Sopa de Letras. Los resultados que arroje la ejecución del mismo mostrarán el comportamiento en paralelo en cuanto a los tiempos obtenidos.

Ambos problemas son desarrollados utilizando C bajo Linux para su realización y bajo PVM para su ejecución. Los cuales tienen como fin principal demostrar la viabilidad que tiene la paralelización para la resolución de problemas que requieren una cantidad de tiempo considerable de CPU buscando una reducción de tiempo para su ejecución.

Cabe mencionar que se eligieron estos ejemplos ya que son aplicables a la solución de problemas que actualmente existen debido a la gran cantidad de datos a procesar.

Las características del cluster utilizado para realizar dichas pruebas son las siguientes:

16 nodos unidos mediante una red ethernet con un ancho de banda de 1 Gbps.
Donde cada nodo tiene las siguientes características,

HARDWARE:

Dos procesadores AMD Opteron de 64 bits.

Memoria RAM de 2 GB.

Un Disco Duro SCSI de 36 GB.

SOFTWARE:

Sistema Operativo Fedora Core 5

Cada aplicación es ejecutada de forma paralela incrementando y decrementando el número de nodos de manera que se pueda hacer un comparativo de dichos corrimientos. La técnica en la cual están diseñadas dichas aplicaciones es la de Maestro-Esclavo en el cual existe un programa Maestro que se encarga de invocar tantos procesos Esclavos sea necesario crear para realizar dicha tarea, repartándose el trabajo. Estos se ejecutan en diferentes nodos del clúster y se encargan de procesar la información enviada por el Maestro. El procesamiento de la información consiste en movimientos en el caso del Infierno y en búsquedas para la Sopa de Letras. Una vez realizadas dichas funciones los resultados son enviados al programa Maestro, este se encarga de unificar los resultados parciales de cada uno de los Esclavos, almacenarlos en archivos y así el usuario consulte el resultado final.

El funcionamiento de estas dos aplicaciones se describe a continuación.

4.1 INFIERNO

La primera aplicación desarrollada es la simulación de un flujo, en este caso se pueden considerar dos masas de personas que se mueven en direcciones opuestas. De aquí que se deben tener en cuenta algunas consideraciones, la primera es que no todas las personas avanzan al mismo destino ni a la misma velocidad, se debe tomar en cuenta también que si hay alguna obstrucción que les impida avanzar deberán tomar direcciones alternativas para su llegada, además que en ningún caso podrán retroceder y cada uno de sus movimientos será para lograr su objetivo.

4.1.1 Maestro

El programa es llamado **maestro_infierno.c** desarrollado en lenguaje C bajo Linux. Es el encargado de crear la matriz con la cual trabajarán los Esclavos, también se encarga de crear el número de Esclavos que serán utilizados para la realización de dicho trabajo, cabe mencionar que este número es variable. Crear las particiones para después enviarlas a los Esclavos y esperar a que estos devuelvan los resultados, una vez recibidos estos se hace la unificación necesaria para después almacenar dicha información en un archivo con extensión data.

El código del programa se presenta a continuación.

En la siguiente figura se muestran las librerías que son utilizadas en el programa maestro.

```
#include <stdio.h>
#include <pvm3.h>
#include <string.h>
```

```
#include <malloc.h>
#include <math.h>

#define ENTRADA "ENTRA.DATA" /*Se almacena la matriz original*/
#define SALIDA1 "RESP1.DATA" /*Movimientos de las personas (A)*/
#define SALIDA2 "RESP2.DATA" /*Movimientos de las personas (B)*/
#define SALIDA3 "RESP3.DATA" /*Se almacena matriz final*/
```

Librerías incluidas en el programa Maestro

Entre las variables empleadas en el programa Maestro son declaradas dos constantes, la primera es utilizada para indicar el numero de particiones que se crearan y la segunda almacena la dimensión de la matriz, además de crear 6 apuntadores de tipo entero para hacer referencia a las matrices utilizadas, 2 arreglos que definen el grupo de personas para saber el sentido al que se dirigen, además de otras variables que sirven para saber el tamaño de la partición que será enviada a cada uno de los Esclavos y otras que recibirán los datos resultantes de los mismos, así como algunas variables auxiliares que son utilizadas por el Maestro.

```
int N_NODOS=100; /*Numero de particiones de la matriz*/
int Dimension=1000; /*Tamaño de la matriz*/

/* Apuntadores de tipo carácter y entero a las matrices*/
char *Tabla,*Tabla3,punto,Puntos[5]={'O','A','A','A','O'};
char Puntos2[5]={'O','B','B','B','O'},c3;
int *Tabla1,*Tabla2,*Tabla11,*Tabla22;
/*Arreglos para identificar a las personas y determinar su dirección*/
/*Variables auxiliares para las particiones de la matriz*/
int
tid,cont,aux,nodos,jrecorre,j2,modulo,limite,parx,modulox,auxiliar,k;
int random,x,y,c,c1,c2,c3,recorre2,ite,x2,punto2,cont2,ypar,jj,ii;
```

Descripción de variables del programa Maestro

En la figura se muestra el código utilizado para la creación de la matriz que almacena los dos grupos de personas. De manera general se expresa que el grupo de personas identificadas con la letra A se colocaran en el extremo izquierdo y moverán al derecho y el grupo asignado con la letra B se coloca en el extremo derecho y con movimientos para alcanzar el izquierdo. La distribución de cada persona se hace de forma aleatoria en un rango definido directamente en el programa.

```
/*Rango que define el tamaño donde serán colocadas las personas de manera
aleatoria, este puede ser variable*/
limite=(x*0.10);
x2=x-limite;

/*Clico donde se genera la matriz, con los dos bloques de personas en ambos
extremos*/
cont=0;
cont2=0;
```

```

    for (i=0;i<x;i++){
        for (j=0;j<y;j++){
            if (i<limite){
/*Se determina de manera aleatoria la ocupación de la casilla, si será
ocupada por una persona o estará vacía de acuerdo a los arreglos*/
                random=rand()%5;
                punto=Puntos[random];
                if (punto=='A'){
                    cont=cont+1;
                }
            }
            if (i>=x2){
                random=rand()%5;
                punto=Puntos2[random];
                if (punto=='B'){
                    cont2=cont2+1;
                }
            }
/*Si ambos limites han sido alcanzados el resto de la matriz se llena de
letras 'O'*/
            if ((limite<=i) && (i<x2)){
                punto=0;
            }
/*Se asigna el valor en la posición (i,j) de la matriz y se guarda en ar-
chivo*/
            Tabla[i*y+j]=punto;
            fprintf(FileIn,"%c",punto);
        }
        fprintf(FileIn,"\n");
    }

    fclose(FileIn);

```

Creación de matriz de datos – Personas A y B

Una vez declarada la matriz con que trabajarán los procesos Esclavos se realizan las llamadas de las funciones para lanzar nuevas tareas y habilitarlas para el envío de datos, así también para la recepción de los resultados obtenidos. Esta sección consta de un ciclo el cual permite hacer el envío de las particiones a los Esclavos para que procesen dicha información, también se encuentra otro ciclo para la recepción de los resultados parciales que son devueltos por los Esclavos.

```

tid=pvm_mytid(); /*Devuelve el identificador del Maestro*/
/*Levanta al proceso Esclavo*/
pvm_spawn("esclavo_infierno",0,PvmTaskDefault,NULL,N_NODOS,&arraytids[0]);
jrecorre=0;
nodos=0;
/*Envia particiones a esclavos*/
for (nodos=0;nodos<N_NODOS;nodos++){
/*Limpia el buffer de envío y crea uno nuevo para empaquetar el mensaje*/
    pvm_initsend(PvmDataDefault);
    if ((nodos+1)==N_NODOS){
        aux=aux+modulo;
        j2=y;
    }
}

```

```

    }
/* Datos de tipo entero empaquetados para su envío*/
    pvm_pkint(&x,1,1); //Tamaño de matriz en x
    pvm_pkint(&aux,1,1); //Tamaño de matriz en y
    for (i=0;i<x;i++){
        for (j=jrecorre;j<j2;j++){
/*Elemento de la matriz en la posición(i,j)*/
            pvm_pubyte(&Tabla[i*x+j],1,1);
        }
        jrecorre=j2;
        j2=j2+aux;
/*El mensaje completo es enviado al proceso Esclavo*/
        pvm_send(arraytids[nodos],1);

/*Recibe datos de los esclavos*/
        jrecorre=0;cont=0;cont2=0;
        j2=y/N_NODOS;
        aux=y/N_NODOS;
        modulo=y%N_NODOS;
        nodos=0;
/*Ciclo que hace la recolección de los resultados enviados por cada uno de
los esclavos*/
        for (nodos=0;nodos<N_NODOS;nodos++){
            pvm_rcv(arraytids[nodos],-1);
            if ((nodos+1)==N_NODOS){
                aux=aux+modulo;
                j2=y;
            }
            for (i=0;i<x;i++){
                for (j=jrecorre;j<j2;j++){
/* Almacena todos los movimientos hechos por el grupo de personas de A*/
                    pvm_upkint(&Tabla1[i*x+j],1,1);
/* Almacena todos los movimientos hechos por el grupo de personas de B*/
                    pvm_upkint(&Tabla2[i*x+j],1,1);
                }
            }
            jrecorre=j2;
            j2=j2+aux;
        }
    }

```

Envío y recepción de información Maestro - Esclavo

Una vez obtenidos todos los resultados parciales enviados por los Esclavos, se unifican los datos dentro del Maestro. Para ello el ciclo hace las pasadas necesarias recopilando los datos y almacenándolos posteriormente en las matrices.

```

//Hacer suma de los acumulados de las dos matrices
    jrecorre=0;cont=0;cont2=0;
    j2=y/N_NODOS;
    aux=y/N_NODOS;
    nodos=0;
/*Ciclo donde se hace la suma de cada una de las particiones*/
    for (nodos=0;nodos<N_NODOS;nodos++){

```

```

ii=0;
for (i=0;i<x;i++){
    jj=0;
    for (j=jrecorre;j<j2;j++){
        Tabla11[ii*x+jj] = Tabla11[ii*x+jj] + Tabla1[i*x+j];
        Tabla22[ii*x+jj] = Tabla22[ii*x+jj] + Tabla2[i*x+j];
        jj++;
    }
    ii++;
}
jrecorre=j2;
j2=j2+aux;
}

```

Suma de los resultados parciales

Finalmente, se guardan en archivos las matrices de resultados; movimientos para A, B, la matriz final y el programa Maestro indica que se retira de PVM, lo cual no indica que mate al proceso, este puede seguir ejecutando otras tareas.

```

for (i=0;i<x;i++){
    for (j=0;j<yvar;j++){
        c1=Tabla11[i*x+j];
        c2=Tabla22[i*x+j];
        fprintf(FileOut1,"%d ",c1);
        fprintf(FileOut2,"%d ",c2);
    }
    fprintf(FileOut1,"\n");
    fprintf(FileOut2,"\n");
}
fclose(FileOut1);
fclose(FileOut2);

pvm_exit();

```

Almacenamiento de información y cierre de programa Maestro

4.1.2 Esclavo

De manera general se da una visión de la estructura del programa Esclavo, este es denominado **esclavo_infierno.c**, la estructura de todos los programas invocados es idéntica.

La figura muestra las librerías incluidas para el funcionamiento de este programa.

```

#include <stdio.h>
#include <pvm3.h>
#include <string.h>
#include <malloc.h>
#include <math.h>

```

Librerías del programa Esclavo

Las variables utilizadas en el programa Esclavo incluyen las estructuras que contienen la información de la dirección a la que se moverá cada persona, los apuntadores que harán referencia a cada una de las matrices que se utilizan así como otras variables auxiliares que son utilizadas por este programa.

```
int DirX[5]={1,0,0,1,1}; /*Dirección para mover A en x*/
int DirY[5]={0,1,-1,1,-1}; /*Dirección para mover A en y*/
int DirX2[5]={-1,0,0,-1,-1}; /*Dirección para mover B en x*/
int DirY2[5]={0,1,-1,-1,1}; /*Dirección para mover B en y*/
int *Tabla1,*Tabla2,limite2,ypar;
int n,x,y,k,xin,yin,xfin,yfin,cont,a,b,c,limite,pasos,parx,k,modx;
int Paso[2]={0,1},Paso2[2]={0,-1};
int i,j,tid,padre; int ranx,rany,dirx,diry,dx,dy;
int letras,i,k,j,m,inicio,bandera,ban,dir,temporal,temporal2;
char *Tabla,aux,otro,blanco;
int auxx,auxy,auxdir,temp,c,c2,i2,j2,coordi,coordj;
int cont1,cont2,auxx2,auxy2,temp2,ban2;
```

Descripción de variables del programa Esclavo

A continuación se muestra la estructura para la recepción y envío de datos del Maestro al Esclavo y viceversa.

```
/*Recibe datos del Maestro y envía el resultado obtenido*/
int main(){
    tid=pvm_mytid();/*Devuelve el identificador del Maestro*/
    /*Regresa el identificador del proceso que reprodujo esta tarea*/
    padre=pvm_parent();
    /*Se inicia la recepción de datos enviados por el Maestro*/
    pvm_recv(padre,-1);
    /*Desempaqueta el valor de x*/
    pvm_upkint(&x,1,1);
    /*Desempaqueta el valor de y*/
    pvm_upkint(&y,1,1);
    /*Ciclo para la recepción de los datos de la matriz*/
    for (i=0;i<x;i++){
        cont=0
        for (j=0;j<y;j++){

            pvm_upkbyte(&Tabla[i*y+j],1,1);
        }
    }
    /*Llamado a la función Mover*/
    Mover();
    /*Ciclo que inicia el envío de los resultados al Maestro*/
    pvm_initsend(PvmDataDefault);
    for (i=0;i<x;i++){
        for (j=0;j<y;j++){
            /*Empaqueta información del grupo A en la posición (i,j)*/
            pvm_pkint(&Tabla1[i*y+j],1,1);
            /*Empaqueta información del grupo B en la posición (i,j)*/
```

```

        pvm_pkint(&Tabla2[i*y+j],1,1);
    }
}
/*El mensaje completo es enviado al proceso Maestro*/
pvm_send(padre,1);
/*Indica que el proceso Esclavo se retira de pvm*/
pvm_exit();
}

```

Esclavo - Recepción y envío de datos

La siguiente función se utiliza para hacer los movimientos de las personas A y B de un extremo a otro respectivamente. Para ello se emplea un ciclo que hace n pasadas para mover cada uno de los elementos de la matriz. Dentro de este se valida el movimiento de la persona para saber si está disponible o no la posición que desea ocupar. La elección de la dirección a la que se mueve la persona se hace de manera aleatoria.

```

/*Función donde se hacen los movimientos de A y B*/
void Mover(){
    limite=x-5;
    coordi=0;coordj=0;
    for (otro=0;otro<ypar;otro++){
        i=x-1;
        j=y-1;
        for (i2=0;i2<x;i2++){
            if (coordi==x){ coordj++; coordi=0;}
            cont1=0;cont2=0;
            for (j2=0;j2<y;j2++){
/* Inicia movimiento de A */
                ban=0;
/*Indica de que forma se moverá A, existen cinco posibilidades: izquierda, derecha, frente, diagonal izquierda y diagonal derecha*/
                dir=rand()%5;//Random para elegir la dirección
                pasos=rand()%2;//Numero de pasos que avanza, 1 ó 2 a la vez
                auxdir=0;
                c=Tabla[i*y+j];
                if (c=='A'){
                    cont1++;
                }
            }
/*Si no se a alcanzado llegar al limite
            if ((c==1) && (i<limite)){
                while (ban==0){
/*A auxx se le asigna la posición en i mas la dirección que se ha elegido
para avanzar, en cuyo caso se decrementara o incrementara el valor de i
en 1 o quedara igual. Lo mismo sucede para la variable auxy que pertenece
al movimiento de j*/
                    auxx=i+dir;
                    auxy=j+dir;
/*Si el valor de dir es diferente a cero se moverán tantos pasos se indiquen para x*/
                    if (DirX[dir]!=0){auxx=auxx+Paso[pasos];}
/*Si el valor de dir es diferente a cero se moverán tantos pasos se indiquen para y*/
                    if (DirY[dir]!=0){auxy=auxy+Paso[pasos];}

```

```

        temp=Tabla[auxx*y+auxy];
/*Se valida si está ocupada por otra persona la posición, en caso de no
ser así a la posición actual se le asigna vacío y a la nueva se le coloca
el valor de la persona*/
        if (temp=='O'){
            Tabla[i*y+j]='O';
            Tabla[auxx*y+auxy]='A';
            ban=1;
        }
/*En caso de que la posición seleccionada no se encuentre vacía, se hace
un recorrido de las cinco direcciones posibles para validar si hay alguna
que pueda ser ocupada*/
        else{
            if (auxdir<5){
                dir=auxdir;
                auxdir++;
            }
/*Si no existe ninguna dirección que pueda ser ocupada la persona no se
mueve hasta el siguiente ciclo*/
            else{
                if (auxdir==5)
                    ban=1;
            }
        }
    }
    Tabla1[coordi*y+coordj]=cont1;
    j--;
/* Termina movimiento de A */

/* Para mover a B se sigue el mismo concepto */

```

Rutina donde se generan los movimientos de las Personas A y B

4.2 SOPA DE LETRAS

La segunda aplicación desarrollada es el típico juego de sopa de letras el cual consiste en dada una lista de palabras se esconden en un conjunto de letras desordenadas para su búsqueda. En términos generales el programa es capaz de hacer la búsqueda de dichas palabras e indicar la posición en la cual se encuentran.

4.2.1 Maestro

El programa es llamado **maestro_sopa.c** desarrollado este también en lenguaje C bajo Linux. Es el encargado de leer desde un archivo predeterminado el diccionario de palabras para colocarlas de manera oculta en la matriz y así posteriormente trabajen los Esclavos con dicha matriz, también se encarga de crear el número de Esclavos que serán utilizados para las búsquedas de las palabras. Crear las particiones para después enviarlas a los Esclavos y esperar a que estos devuelvan los resultados, una vez recibidos estos se hace la unificación necesaria para después almacenar dicha información en un archivo con extensión data.

La figura muestra las librerías incluidas para el funcionamiento de este programa.

```

#include <stdio.h>
#include <pvm3.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
/* Contiene las palabras ocultas dentro del conjunto de letras desordena-
das */
#define GENERA "SOP.DATA"
/* Listado de palabras junto con las coordenadas en donde fueron escritas
originalmente */
#define PALABRA "PALABRAS.DATA"
/* Es el diccionario ya establecido de donde se tomaran las palabras a
colocar en la sopa */
#define DICC "DICC.DATA"
/* Archivo donde se encuentran las palabras y letras, propiamente la sopa
de letras */
#define ENTRADA "SOP.DATA"
/* Guarda todas las coordenadas donde se repite una palabra dentro de la
matriz sopa */
#define SALIDA "RESP.DATA"

```

Librerías incluidas en el programa Maestro

Entre las variables empleadas en el programa Maestro es declarada una constante la cual es utilizada para indicar el numero de particiones que se crearan, otra que almacena la dimensión de la matriz, además de crear 4 apuntadores de tipo carácter para hacer referencia a las matrices utilizadas, un arreglo que contiene los 26 caracteres del alfabeto, además de otras variables que sirven para saber el tamaño de la partición que será enviada a cada uno de los Esclavos y otras que recibirán los datos resultantes de los mismos, así como algunas variables auxiliares que son utilizadas por el Maestro.

```

int N_NODOS=10; /*Número de particiones o esclavos*/
int n = 100; /*Dimensión de la matriz*/

/*Arreglo que contiene todas las letras del alfabeto para su búsqueda*/
char Letras[26]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N',
'O','P','Q','R','S','T','U','V','W','X','Y','Z'};
/*Apuntadores de las matrices y archivos, entre otras variables*/
char *Tabla,*Tabla1, Diccionario[20],Palabra[10],Pal[10],a,letra;
FILE *FileIn,*FileOut,*FilePal,*FileDic;
int n,x,y,i,j,random,c,maxfil,max,tid,aux,nodos,jrecorre,j2,modulo;
int xin,yin,xfin,yfin,recorre2,iniaux,finaux,tampal ;

```

Descripción de variables del programa Maestro

Extracto del menú principal, en este se hace un llamado a dos funciones la primera crea la matriz donde se ocultaran las palabras a buscar y la segunda genera la posición y escribe dentro de la matriz las palabras. Además de ser escritas en la matriz se guarda la posición inicial y final de cada palabra en un archivo PALABRAS.DATA. El maestro es el encarga-

do de particionar la matriz y enviar cada pedazo a un esclavo junto con la palabra a buscar después recibe del esclavo las coordenadas de la palabra encontrada, es decir, la posición donde empieza y donde termina la palabra.

```
int main(){
    /*Archivo donde se encuentra la matriz sopa de letras*/
    FileIn=fopen(ENTRADA,"r");
    /*Archivo que hace referencia de las palabras a buscar*/
    FileDic=fopen(DICC,"r");
    /*Archivo donde se escribirán las coordenadas de las palabras encontra-
das*/
    FileOut=fopen(SALIDA,"w");
    Genera();
    Palabrass();

    tid=pvm_mytid();/*Devuelve el identificador del Maestro*/
    /*Levanta al proceso Esclavo*/
    pvm_spawn("esclavo_sopas22",NULL,0,NULL,N_NODOS,arraytids);
    jrecorre=0;
    j2=y/N_NODOS;
    aux=y/N_NODOS;
    modulo=y%N_NODOS;
    nodos=0;

    while (nodos <= N_NODOS){
        if (nodos < N_NODOS){
            /*Envia particiones a esclavos*/
            for (nodos=0;nodos<N_NODOS;nodos++){
                /*Limpia el buffer de envío y crea uno nuevo para empaquetar el mensaje*/
                pvm_initsend(PvmDataDefault);
                if ((nodos+1)==N_NODOS){
                    aux=aux+modulo;
                    j2=y;
                }
            }
            /*Numero de caracteres de la palabra a buscar*/
            pvm_pkint(&n,1,1);
            pvm_pkint(&x,1,1); //Valor de la matriz en x
            pvm_pkint(&aux,1,1); //Valor de la matriz en y

            for (i=0;i<x;i++){
                for (j=jrecorre;j<j2;j++){
                    /*Elemento de la matriz en la posición(i,j)*/
                    pvm_pkbyte(&Tabla[i*x+j],1,1);
                    c=Tabla[i*x+j];
                }
            }
            jrecorre=j2;
            j2=j2+aux;
            /*El mensaje completo es enviado al proceso Esclavo*/
            pvm_send(arraytids[nodos],1);
        }

        jrecorre=0;
        j2=y/N_NODOS;
        aux=y/N_NODOS;
    }
}
```

```

        modulo=y%N_NODOS;
/*Se lee la palabra a buscar y se almacena en un arreglo para su envío a
los esclavos*/
        for ( k=0 ; k<n ; k++) {
                fscanf(FileDic,"%s",&Palabra); /*Palabra a buscar*/
                for (nodos=0;nodos<N_NODOS;nodos++){
/*Limpia el buffer de envío y crea uno nuevo para empaquetar el mensaje*/
                        pvm_initsend(PvmDataDefault);
/*Almacena toda la palabra para su envío*/
                        pvm_pkbyte(Palabra,10,1);
/*El mensaje completo es enviado al proceso Esclavo*/
                        pvm_send(arraytids[nodos],1);

/*Recive datos de los esclavos*/
                        pvm_recv(arraytids[nodos],1);
/*Desempaqueta la posición donde inicia la palabra en X*/
                        pvm_upkint(&xin,1,1);
/*Desempaqueta la posición donde inicia la palabra en Y*/
                        pvm_upkint(&yin,1,1);
/*Desempaqueta la posición donde termina la palabra en X*/
                        pvm_upkint(&xfin,1,1);
/*Desempaqueta la posición donde termina la palabra en Y*/
                        pvm_upkint(&yfin,1,1);
/*Desempaqueta la palabra buscada*/
                        pvm_upkbyte(Pal,10,1);
                        if ((xin>=0) && (yin>=0)){
                                iniaux=((y/N_NODOS)*nodos);
                                yin=iniaux+yin;
                                yfin=iniaux+yfin;
/*Se almacena las coordenadas enviadas por los esclavos en el archivo de
salida*/
                                fprintf(FileOut, "%s (%d,%d) -> (%d,%d) \n",
                                        Palabra,xin+2,yin+1,xfin+2,yfin+1);
                        }
                }
        }
        nodos=N_NODOS;
}
pvm_exit();
}

```

Envío y recepción de información Maestro - Esclavo

4.2.2. Esclavo

De manera general se da una visión de la estructura del programa Esclavo, este es denominado **esclavo_sopa.c**, la estructura de todos los programas invocados es idéntica.

La figura muestra las librerías incluidas para el funcionamiento de este programa.

```

#include <stdio.h>
#include <pvm3.h>
#include <string.h>
#include <malloc.h>
#include <math.h>

```

Librerías del programa Esclavo

Las variables utilizadas en el programa Esclavo incluyen dos estructuras que contienen la información de los movimientos que se deben generar para la búsqueda y ubicación de cada una de las palabras, las variables que harán referencia a cada una de las coordenadas de las palabras encontradas así como otras auxiliares que son utilizadas por este programa.

```
int DirX[8]={1,-1,0,0,1,-1,-1,1};
int DirY[8]={0,0,1,-1,1,-1,1,-1};
char *Tabla;
char Palabra[10];
int n,x,y,k,xin,yin,xfin,yfin,a,b;
```

Descripción de variables del programa Esclavo

A continuación se muestra la estructura para la recepción y envío de datos del Maestro al Esclavo y viceversa.

```
/*Recibe datos del Maestro y envía el resultado obtenido*/
int main(){
    tid=pvm_mytid();()/*Devuelve el identificador del Maestro*/
/*Regresa el identificador del proceso que reprodujo esta tarea*/
    padre=pvm_parent();
/*Se inicia la recepción de datos enviados por el Maestro*/
    pvm_recv(padre,1);
/*Desempaqueta el valor de n*/
    pvm_upkint(&n,1,1);
/*Desempaqueta el valor de x*/
    pvm_upkint(&x,1,1);
/*Desempaqueta el valor de y*/
    pvm_upkint(&y,1,1);
/*Ciclo para la recepción de los datos de la matriz*/
    for (i=0;i<x;i++){
        for (j=0;j<y;j++){
            pvm_upkbyte(&Tabla[i*y+j],1,1);
        }
    }
/*Ciclo para la recepción de cada una de las palabras a buscar*/
    for ( k=0 ; k<n ; k++) {
        pvm_recv(padre,1);
        pvm_upkbyte(Palabra,10,1);
/*Llamado a la función Mover*/
        Buscar();
/*Ciclo que inicia el envío de los resultados al Maestro*/
        pvm_initsend(PvmDataDefault);
/*Empaqueta coordenada inicial de la palabra en X*/
        pvm_pkint(&xin,1,1);
/*Empaqueta coordenada inicial de la palabra en Y*/
        pvm_pkint(&yin,1,1);
/*Empaqueta coordenada final de la palabra en X*/
        pvm_pkint(&xfin,1,1);
```

```

/*Empaqueta coordenada final de la palabra en Y*/
    pvm_pkint(&yfin,1,1);
/*Empaqueta la palabra buscada*/
    pvm_pkbyte(Palabra,10,1);
/*El mensaje completo es enviado al proceso Maestro*/
    pvm_send(padre,1);
}
/*Indica que el proceso Esclavo se retira de pvm*/
    pvm_exit();
}

```

Esclavo - Recepción y envío de datos

La siguiente función se utiliza para hacer la búsqueda de cada palabra indicada por el maestro. Para ello se emplea un ciclo que hace n pasadas para revisar cada una de las posibles direcciones donde puede estar escrita dicha palabra. Dentro de este se valida cada una de las letras hasta encontrar la cadena correcta que forme la palabra. De manera que al ubicar la palabra se almacena la coordenada inicial y final de esta (xin, yin) (xfin, yfin).

```

void Buscar(){
    int i,j,d,p,letras;
    int fin=0;
/*Longitud de la palabra*/
    letras=strlen(Palabra);
    for (i=0;i<x;i++){
        for (j=0;j<y;j++){
/*Ciclo donde se genera la busqueda de la palabra*/
            for (d=0;d<8 && !fin;d++){
                for(a=i,b=j,p=1;
                    Palabra[p-1]==Tabla[a*y+b] && p<=letras && a>=0 &&
a<x && b>=0 &&b<y;
                    p++,a+=DirX[d],b+=DirY[d])
/*Si la palabra es encontrada se almacenan las coordenadas*/
                        if (p==letras){
                            xin=i; yin=j;
                            xfin=a; yfin=b;
                            fin=1;
                        }
                    }
                }
            }
/*En caso de no ubicar a la palabra todas las coordenadas tomaran el va-
lor de -1*/
            if (fin==0){
                xin=-1; yin=-1;
                xfin=-1; yfin=-1;
            }
        }
    }
}

```

Rutina donde se genera la búsqueda de palabras

4.3 PRUEBAS Y RESULTADOS

4.3.1 Infierno

En esta sección se muestran cuatro tablas con los diferentes tiempos arrojados por cada uno de los corrimientos hechos manipulando diferentes datos. La primera columna contiene el tamaño de las matrices utilizadas, la segunda el número de particiones hechas a cada una de las matrices, en la tercera columna el número de nodos empleados para dichos corrimientos, siguiendo con la manipulación de la cantidad de personas A y B, finalmente cada uno de los tiempos registrados en segundos. Es importante mencionar que se hicieron cuatro diferentes muestras en la población de manera esquemática, a continuación se muestra cada manipulación hecha.

Figura 1 - Caso A

	FRON	TERA	
A			B

Figura 2 – Caso B

	FRON	TERA	
A			B

Figura 3 – Caso C

	FRON	TERA	
A			B

Figura 4 – Caso D

	FRO	NTERA	
A			B

La prueba consiste en que dados dos grupos de personas A y B, deben moverse al otro extremo de la matriz, es decir las personas de la región A deben avanzar hacia la región donde se encuentran inicialmente las de B y estas a su vez avanzar a la región de A. Para llevar acabo los movimientos de cada persona existen dos restricciones la primera es que podrán moverse en cualquier dirección excepto hacia atrás y la segunda es que dos personas no pueden ocupar el mismo lugar por lo cual deben verificar si el espacio esta vacío o no.

Los tiempos registrados dependen de dos variables principalmente la primera del tamaño de los grupos A, B la matriz (si la matriz crece también crece el número de personas) y la segunda del número de nodos utilizados.

A continuación se muestran los resultados en tablas obtenidos para cada uno de los cuatro casos mencionados anteriormente (Caso A, B, C y D).

Caso A

T. MATRIZ	No. DE PART.	No. NODOS	No. DE PERSONAS		TIEMPO
			A	B	
2000*2000	10	1	239842	239927	7.955s
2000*2000	10	2	239842	239927	7.774s
2000*2000	10	3	239842	239927	5.518s
2000*2000	10	4	239842	239927	4.142s
4000*4000	10	1	959462	960691	52.811s
4000*4000	10	2	959462	960691	52.650s
4000*4000	10	3	959462	960691	2736.592798s
4000*4000	10	4	959462	960691	228.243355s
6000*6000	10	1	2159299	2158811	2m 50.907s136.476s
6000*6000	10	2	2159299	2158811	2m 48.315117.182s
6000*6000	10	3	2159299	2158811	1m 44.659s83.392s
6000*6000	10	4	2159299	2158811	1m 24.223s65.559s
8000*8000	10	1	3839894	3840794	6m 36.429396.429s
8000*8000	10	2	3839894	3840794	4m 40.930244.108s
8000*8000	10	3	3839894	3840794	3m 21.688183.658s
8000*8000	10	4	3839894	3840794	2m 27.677145.232s
10000*10000	10	1	5999804	5998581	12m 43.319763.319s
10000*10000	10	2	5999804	5998581	8m 08.288488.288s
10000*10000	10	3	5999804	5998581	5m 36.331271.621s
10000*10000	10	4	5999804	5998581	4m 16.528235.88s

Caso B

T. MATRIZ	No. DE PART.	No. NODOS	No. DE PERSONAS		TIEMPO
			A	B	
2000*2000	10	1	479768	480186	8.37995s
2000*2000	10	2	479768	480186	8.308s
2000*2000	10	3	479768	480186	5.757880s
2000*2000	10	4	479768	480186	4.559s
4000*4000	10	1	1919292	1918857	56.998s
4000*4000	10	2	1919292	1918857	56.406854s
4000*4000	10	3	1919292	1918857	39.49336.798s
4000*4000	10	4	1919292	1918857	30.33228.355s
6000*6000	10	1	4319871	4319334	2m 16.476170.907s
6000*6000	10	2	4319871	4319334	2m 02.463142.463s
6000*6000	10	3	4319871	4319334	4m 53.524113.524s
6000*6000	10	4	4319871	4319334	4m 29.46276.961s
8000*8000	10	1	7678810	7678241	7m 04.658424.658s
8000*8000	10	2	7678810	7678241	5m 42.864280.93s
8000*8000	10	3	7678810	7678241	3m 28.603201.688s
8000*8000	10	4	7678810	7678241	2m 41.445152.677s
10000*10000	10	1	12001569	12000368	13m 43.753823.753s
10000*10000	10	2	12001569	12000368	8m 38.736508.452s
10000*10000	10	3	12001569	12000368	6m 01.766336.331s
10000*10000	10	4	12001569	12000368	4m 34.698256.528s

Caso C

T. MATRIZ	No. DE PART.	No. NODOS	No. DE PERSONAS		TIEMPO
			A	B	
2000*2000	10	1	959462	960691	9.519s
2000*2000	10	2	959462	960691	8.993s

2000*2000	10	3	959462	960691	6.334s
2000*2000	10	4	959462	960691	5.172s
4000*4000	10	1	3839894	3840794	^{1m} 04.29964.299s
4000*4000	10	2	3839894	3840794	56.48059.851s
4000*4000	10	3	3839894	3840794	27.59241.493s
4000*4000	10	4	3839894	3840794	22.24333.559s
6000*6000	10	1	8639969	8638309	^{3m} 28.519208.519s
6000*6000	10	2	8639969	8638309	^{1m} 57.182182.947s
6000*6000	10	3	8639969	8638309	^{1m} 23.392154.659s
6000*6000	10	4	8639969	8638309	^{1m} 05.559114.233s
8000*8000	10	1	15360642	15359842	^{8m} 08.703488.703s
8000*8000	10	2	15360642	15359842	^{4m} 24.108322.864s
8000*8000	10	3	15360642	15359842	^{3m} 03.658218.603s
8000*8000	10	4	15360642	15359842	^{2m} 25.232165.445s
10000*10000	10	1	24000740	23994917	^{15m} 39.945939.945s
10000*10000	10	2	24000740	23994917	^{8m} 28.452539.107s
10000*10000	10	3	24000740	23994917	^{5m} 56.787371.766s
10000*10000	10	4	24000740	23994917	^{4m} 34.993286.993s

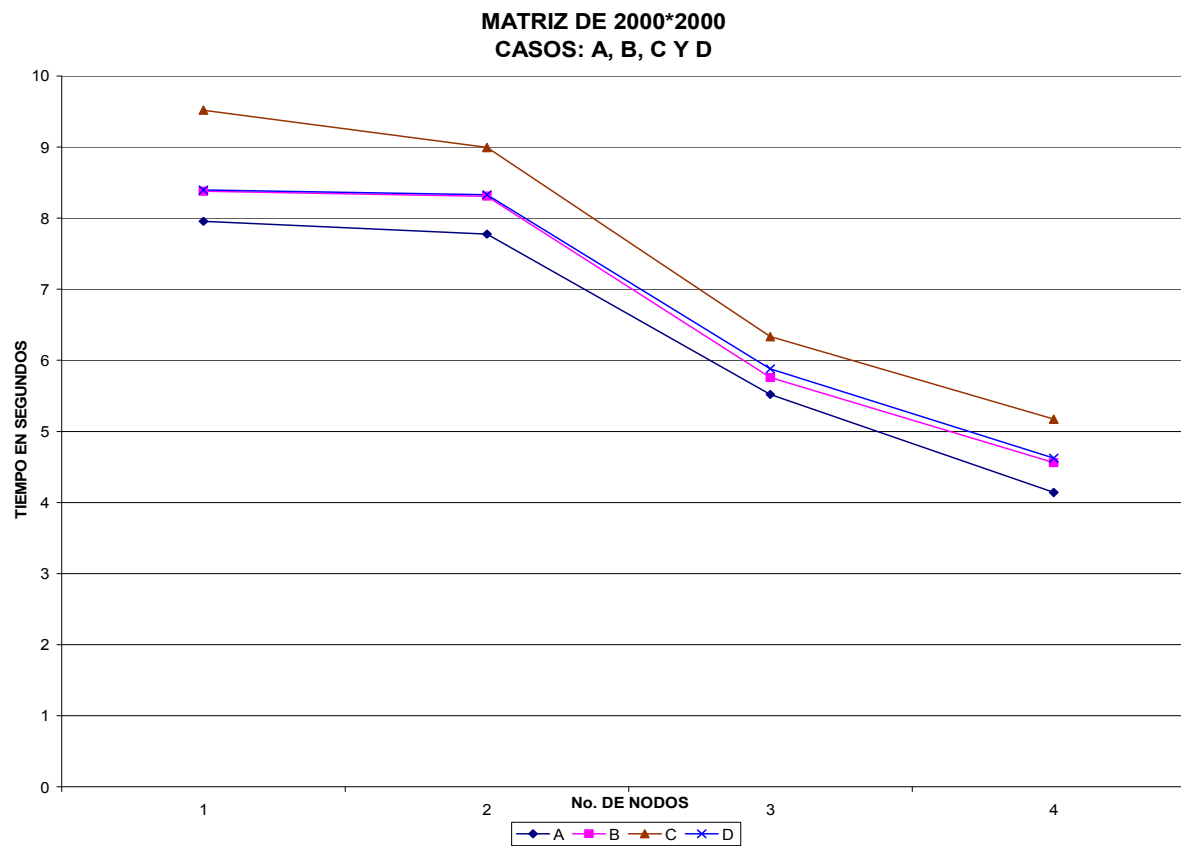
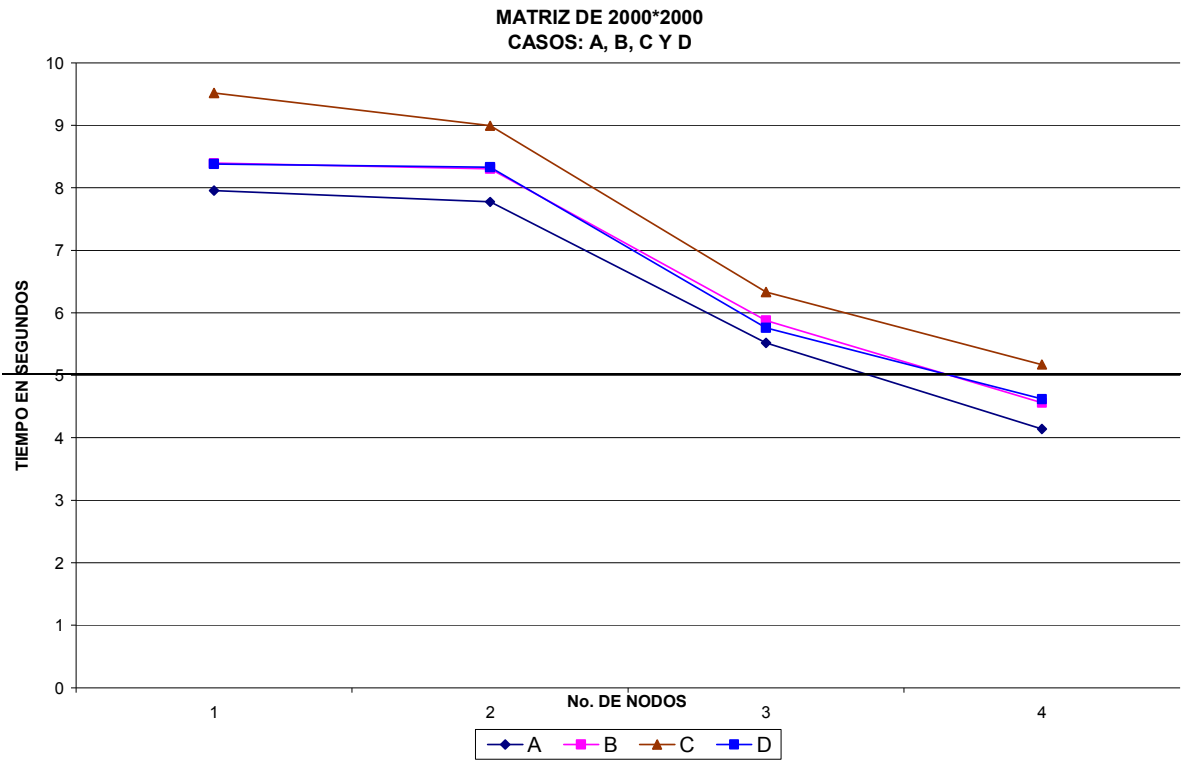
Caso D

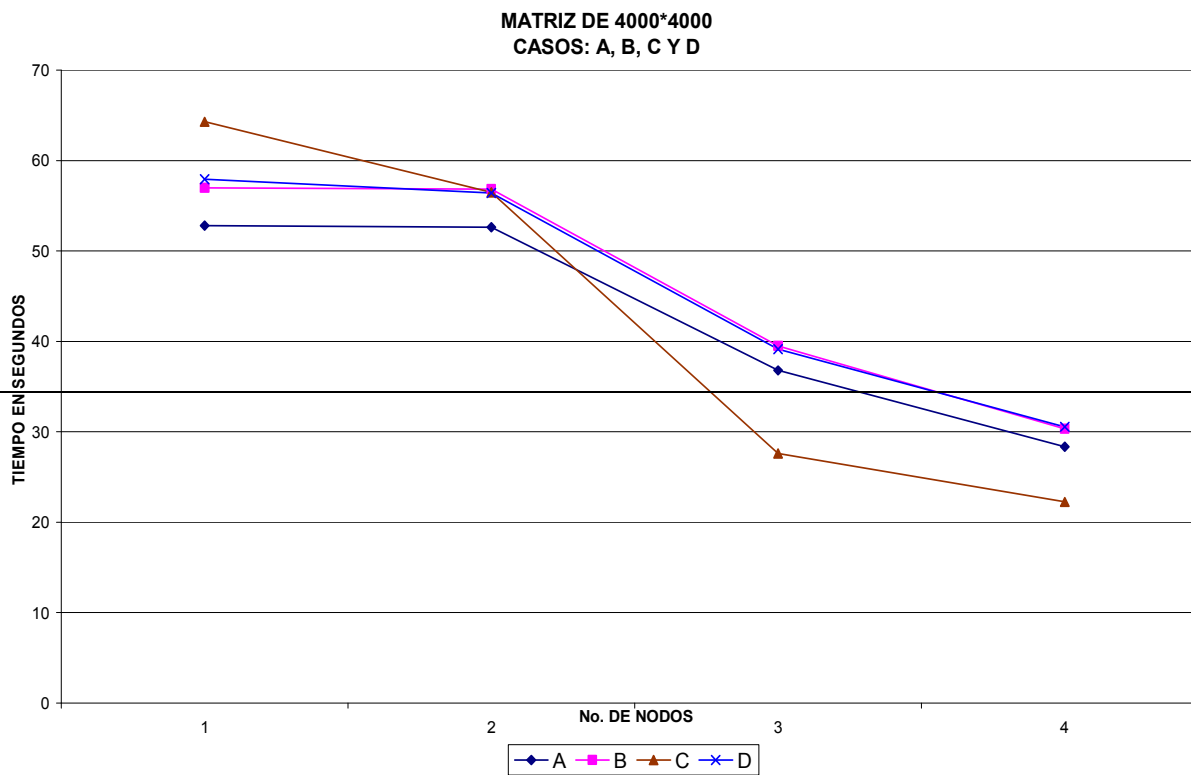
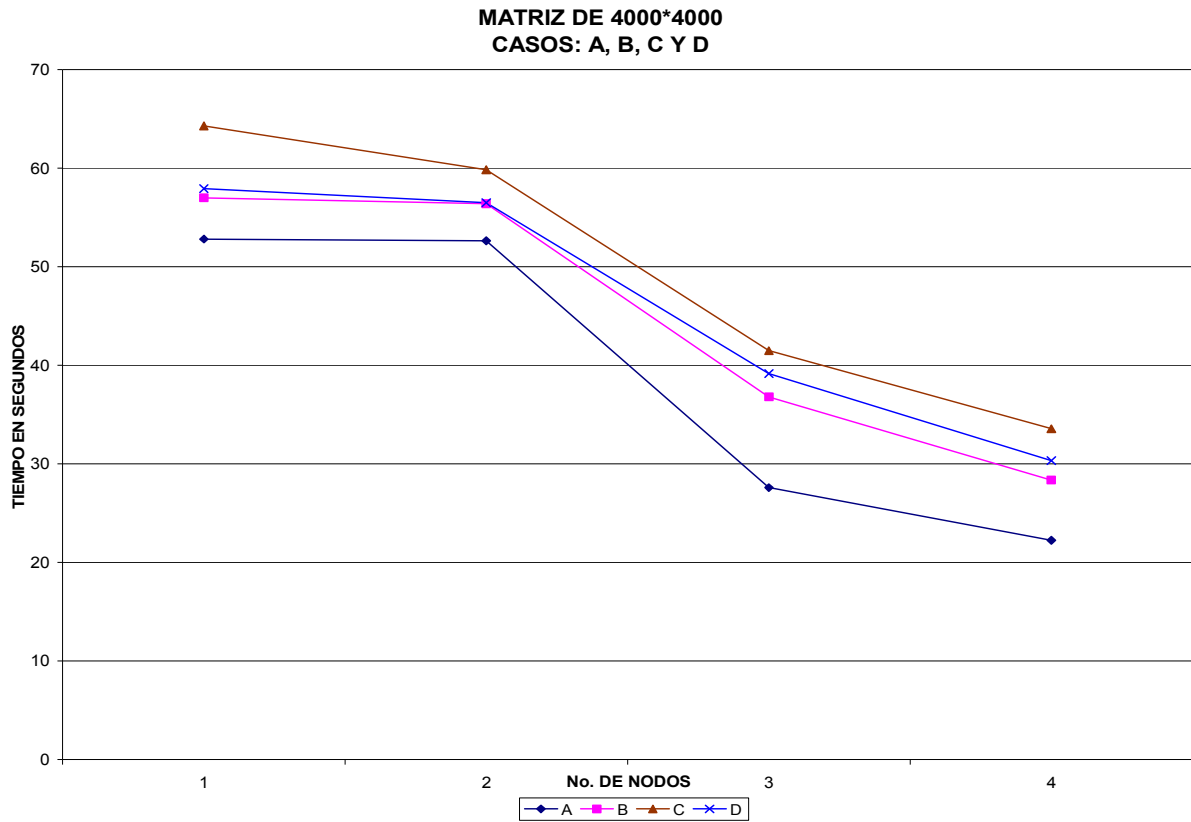
T. MATRIZ	No. DE PART.	No. NODOS	No. DE PERSONAS		TIEMPO
			A	B	
2000*2000	10	1	959462	480356	8.3795s
2000*2000	10	2	959462	480356	8.329s
2000*2000	10	3	959462	480356	5.880757s
2000*2000	10	4	959462	480356	4.621s
4000*4000	10	1	3839894	1920021	57.9838s
4000*4000	10	2	3839894	1920021	56.49006s
4000*4000	10	3	3839894	1920021	39.149s
4000*4000	10	4	3839894	1920021	30.332559s
6000*6000	10	1	8639969	4319702	^{3m}

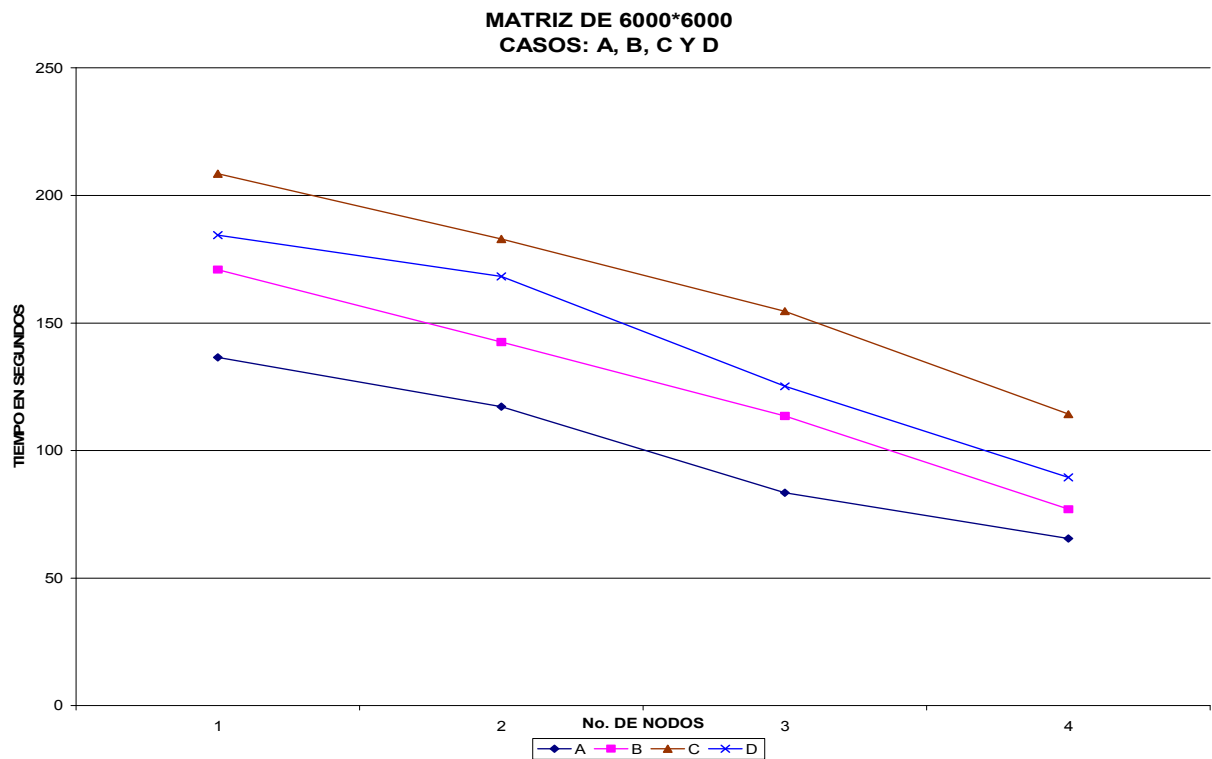
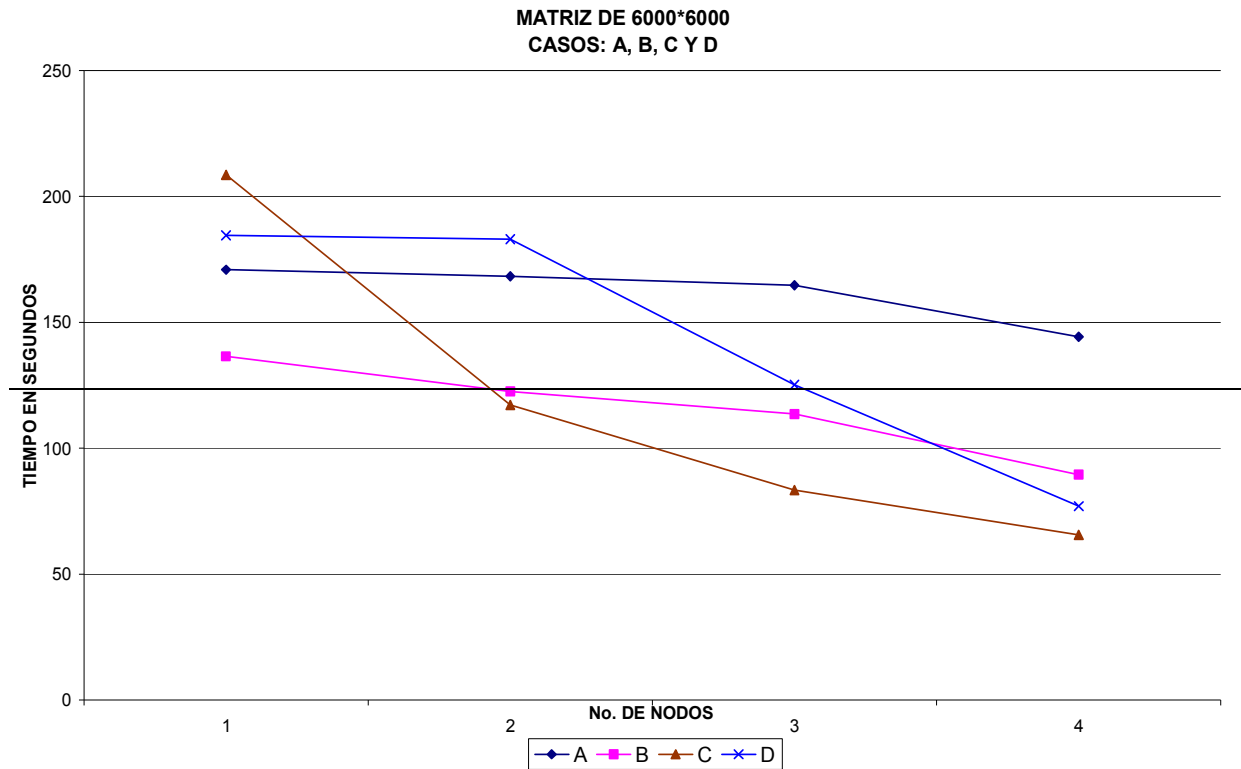
					04.467184.467s
6000*6000	10	2	8639969	4319702	~~03.947168.315s~~ 3m
6000*6000	10	3	8639969	4319702	~~05.145125.145s~~ 2m
6000*6000	10	4	8639969	4319702	~~26.96189.462s~~ 1m
8000*8000	10	1	15360642	7678973	~~05.871425.871s~~ 7m
8000*8000	10	2	15360642	7678973	~~53.707293.707s~~ 4m
8000*8000	10	3	15360642	7678973	~~25.405205.405s~~ 3m
8000*8000	10	4	15360642	7678973	~~39.926159.926s~~ 2m
10000*10000	10	1	24000740	11998430	~~49.516829.516s~~ 13m
10000*10000	10	2	24000740	11998430	~~39.107518.736s~~ 8m
10000*10000	10	3	24000740	11998430	~~31.621356.787s~~ 4m
10000*10000	10	4	24000740	11998430	~~55.880274.698s~~ 3m

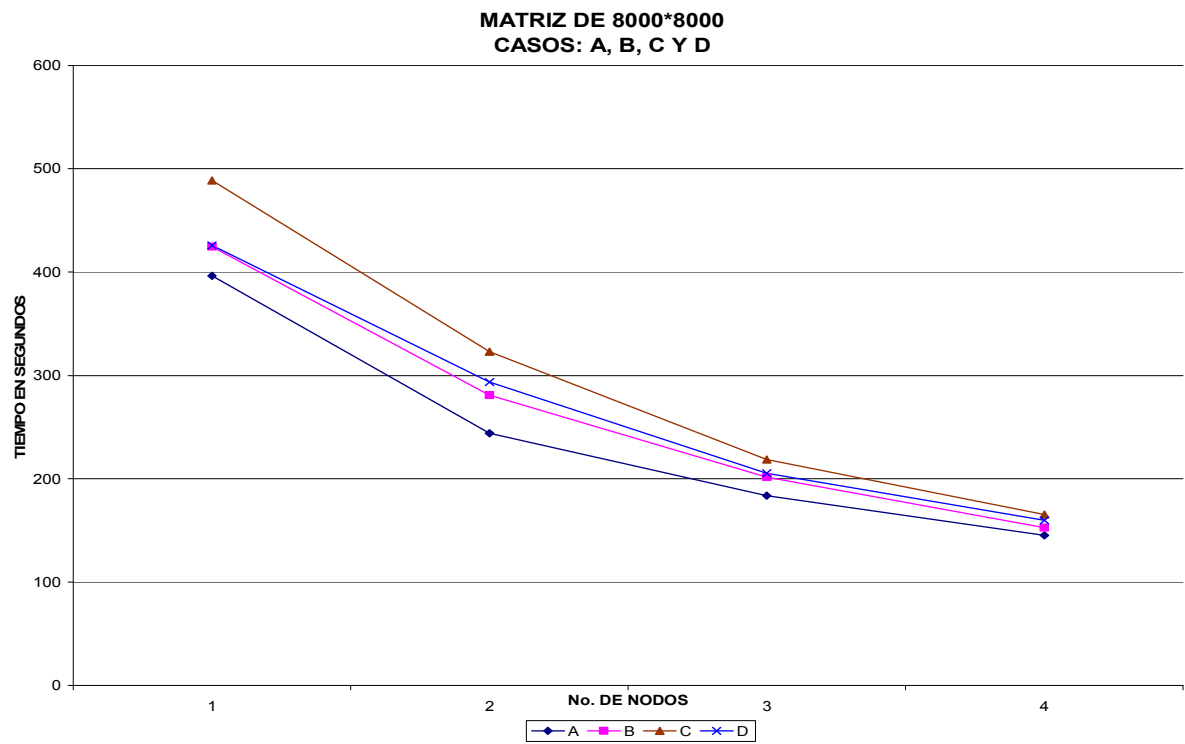
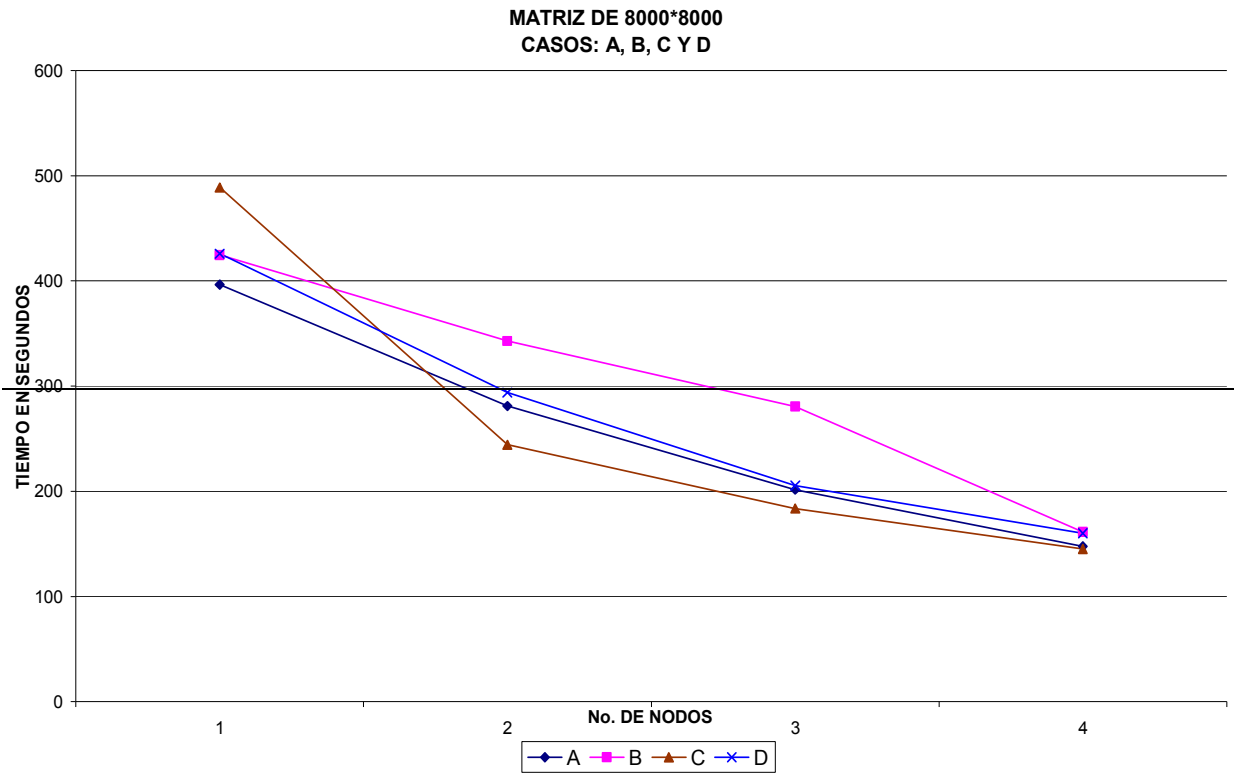
A continuación se muestra cada una de las graficas con los tiempos registrados de los cuatro casos contemplados para dicho ejercicio; considerando que dichos tiempos son variables debido a que se debe tomar en cuenta que la población de personas para el caso A no es del mismo tamaño que para el de B, C o D debido a que es variable por la densidad de la misma.

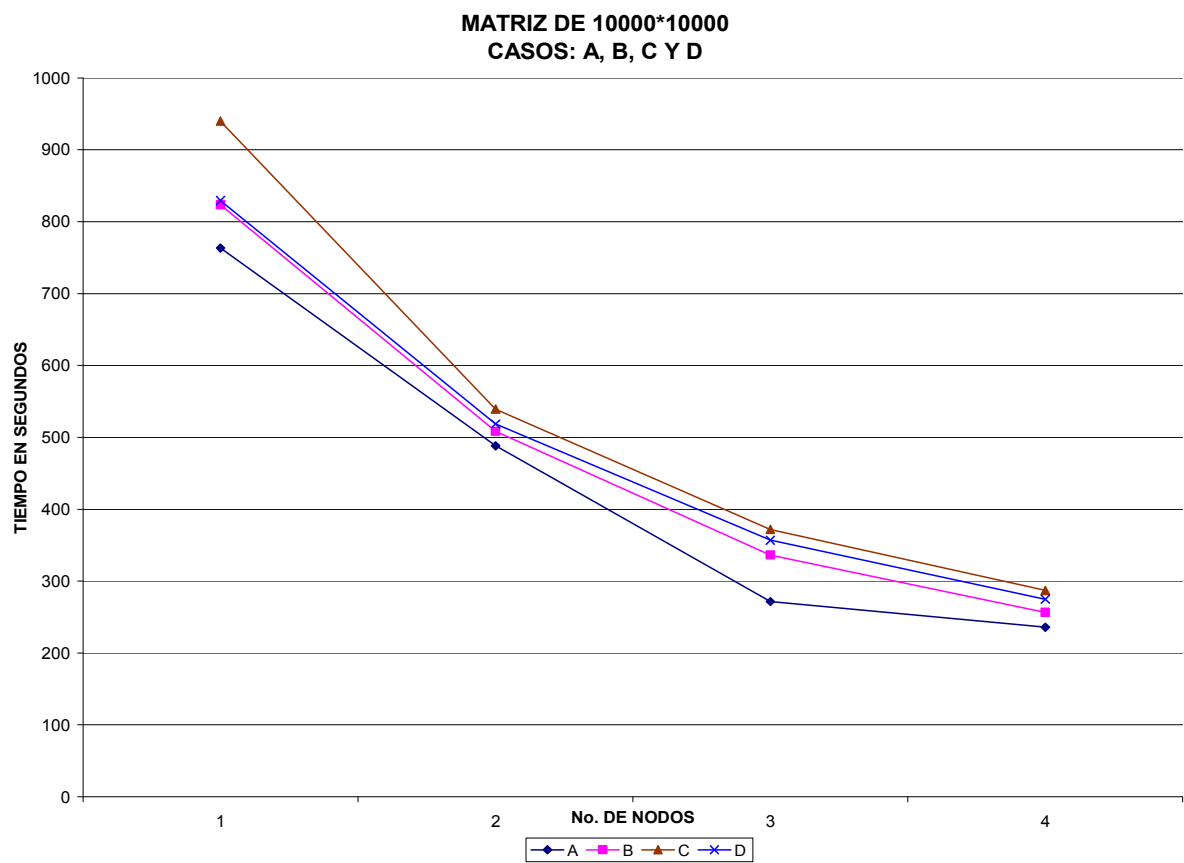
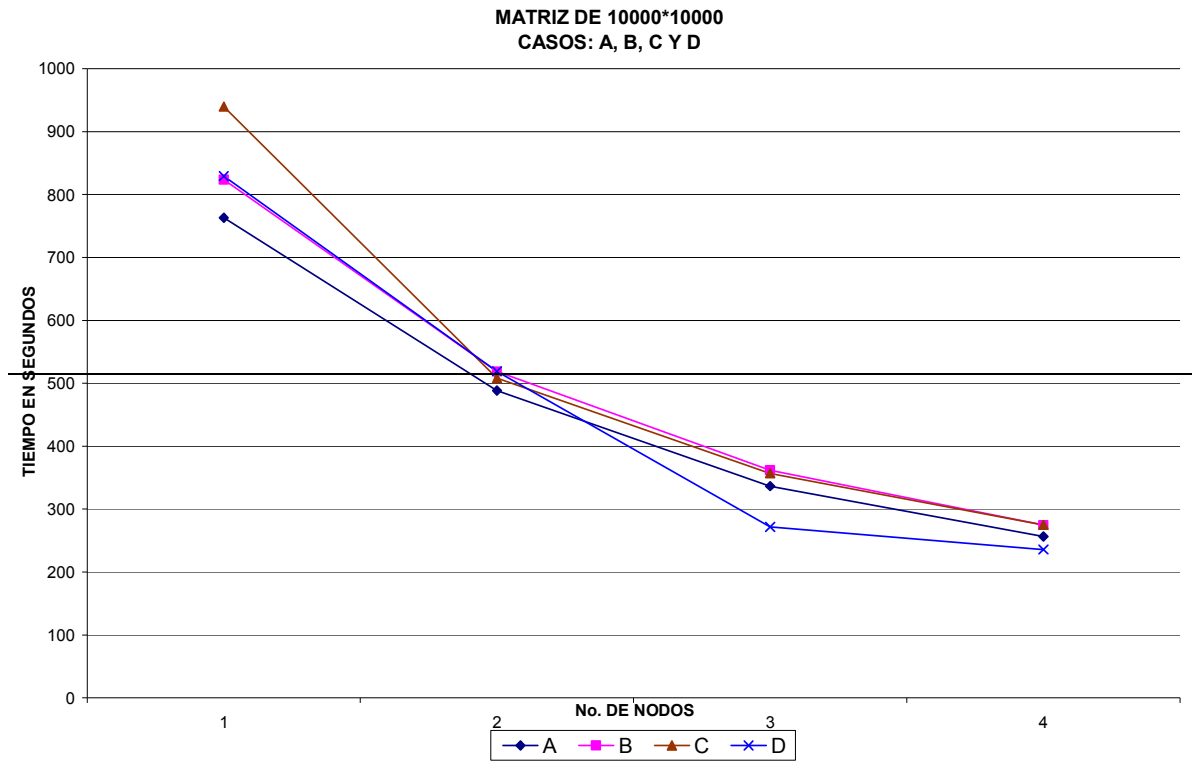
$$Densidad = (No. de personas) / (Tamaño de región)$$









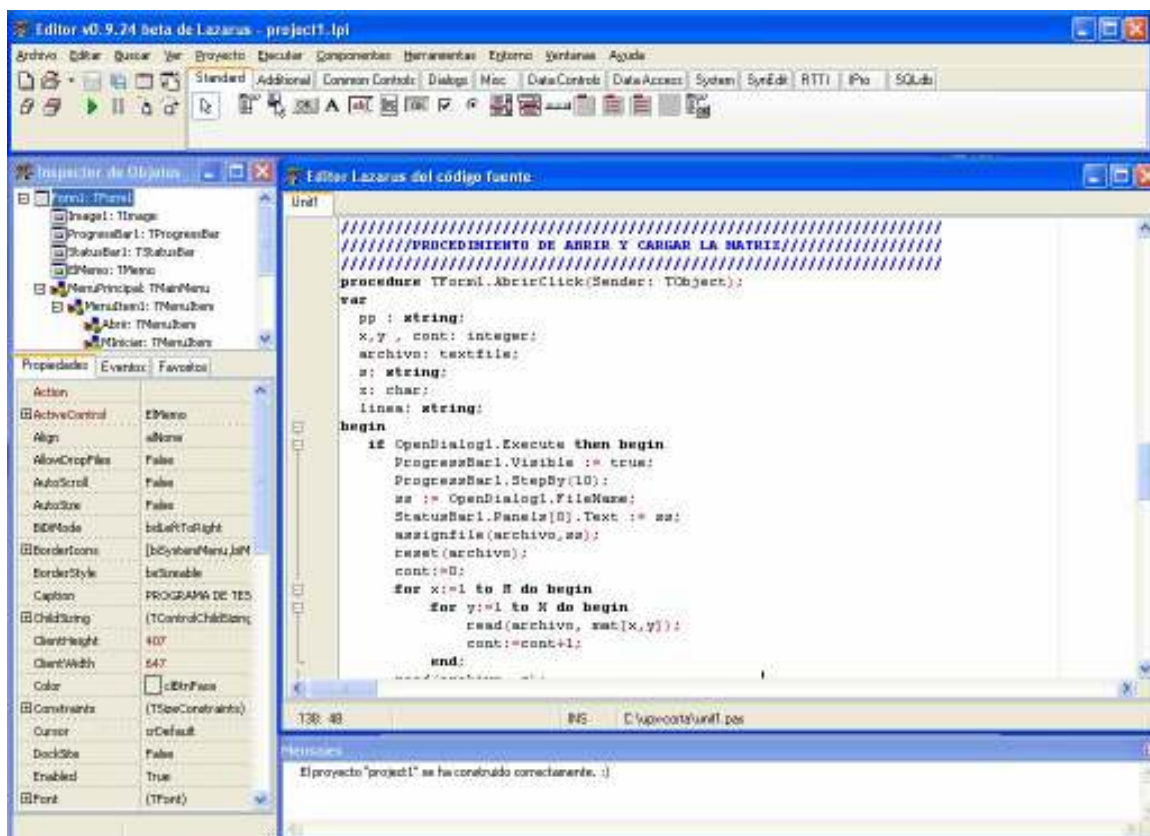


Para la simulación de los movimientos de cada uno de los casos expuestos anteriormente se desarrollo un software utilizando el programa Lazarus, el cual dado el archivo con los resultados obtenidos de los movimientos de las personas A y B genera una imagen de salida en la cual por cada persona A coloca un píxel de color rojo, para las personas B coloca un píxel de color verde y en el caso de los espacios libres coloca un píxel de color blanco. Así se obtiene el resultado de manera gráfica.

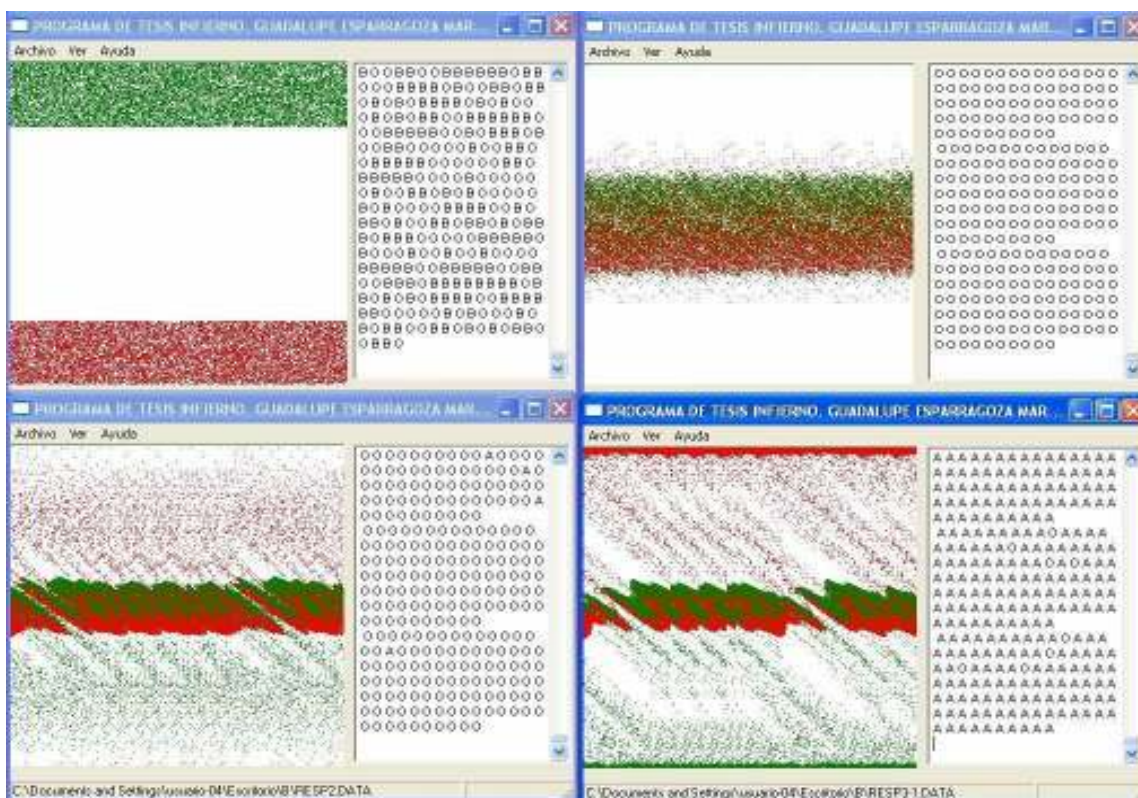
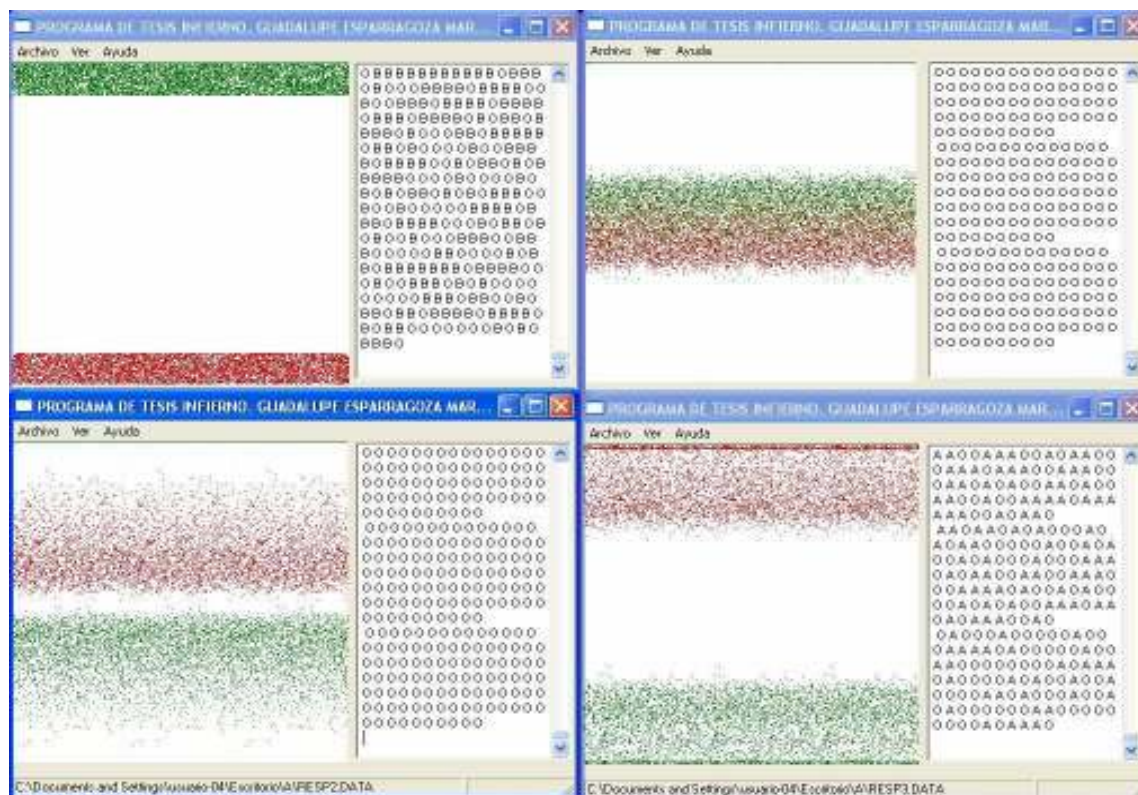
Lazarus es una herramienta visual para el desarrollo de aplicaciones en el lenguaje de programación Pascal, disponible para los sistemas operativos Windows, GNU/Linux y Mac OS X.

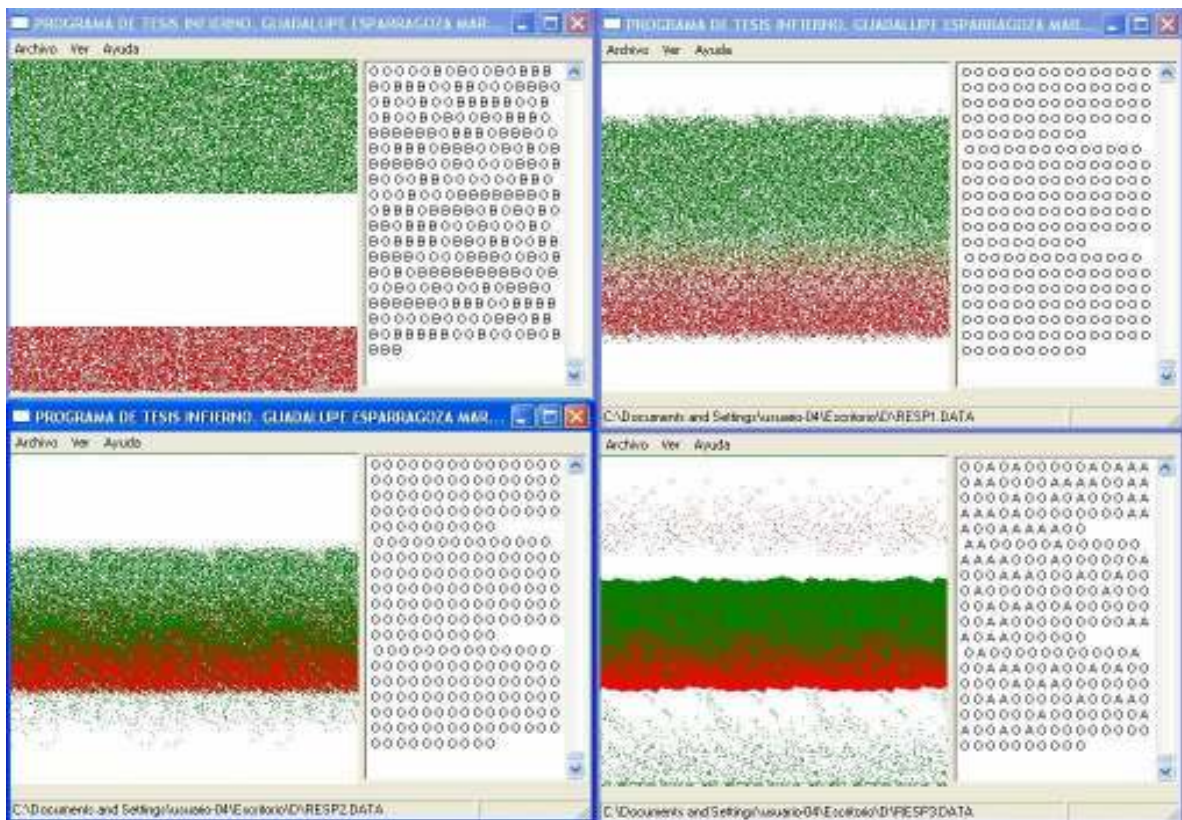
Se trata de una alternativa gratuita a Delphi, desarrollada como proyecto de software libre a partir de Free Pascal. La web y la mayoría de la documentación están en inglés, pero el entorno de desarrollo (IDE) sí está traducido al español en gran parte.

Se muestra una imagen de cómo se presenta este programa frente al usuario.



La simulación se presenta usando la aplicación desarrollada en Lazarus para los casos descritos.





La siguiente figura muestra como *pvm* añade los nodos que el usuario indica para la ejecución del programa. Los pasos a seguir son:

1. Abrir una consola de Linux
2. Escribir *pvm* y teclear Enter
3. Una vez dentro agregamos cuantos nodos deseemos con la palabra *add*. Es importante tomar en cuenta que por cada nodo se debe dar la instrucción.
4. Finalmente para visualizar cuales nodos fueron agregados tecleamos la palabra *conf*.

```

3:cluster.cs.buap.mx - cluster - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles

pvm> ps -a
ps -a
      HOST      TID  FLAG 0x COMMAND
nodo2.cluster.fcc 4001a 4/c -
nodo2.cluster.fcc 4001b 6/c,f esclavo_sopas22
nodo2.cluster.fcc 4001c 6/c,f esclavo_sopas22
nodo4      c0019 6/c,f esclavo_sopas22
nodo4      c001a 6/c,f esclavo_sopas22
nodo3      80019 6/c,f esclavo_sopas22
nodo3      8001a 6/c,f esclavo_sopas22
nodo5     100019 6/c,f esclavo_sopas22
nodo5     10001a 6/c,f esclavo_sopas22

pvm>
Connected to cluster.cs.buap.mx  SSH2 - aes128-cbc - hmac-md5 - none 80x27

```

4.3.2 Sopa de letras

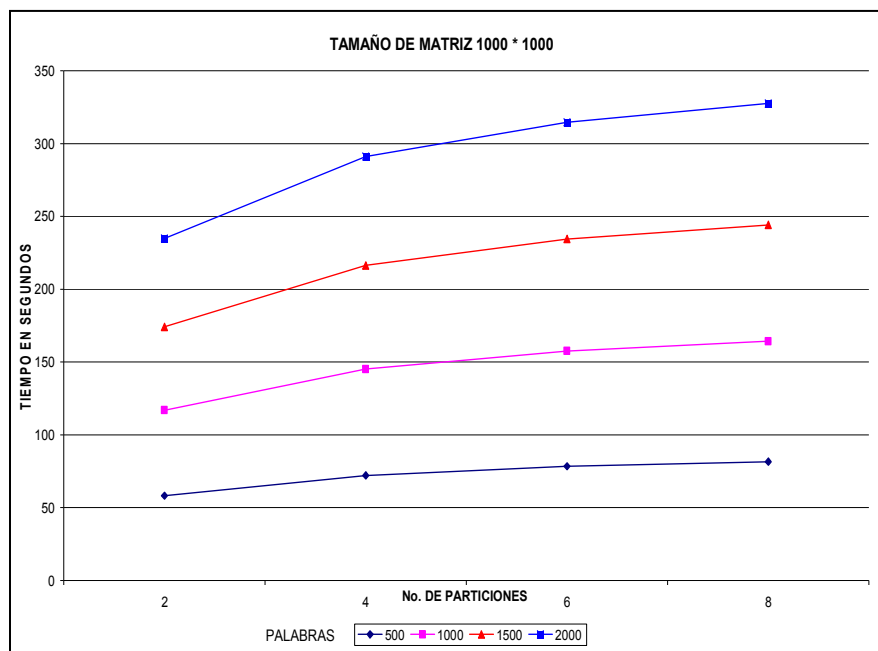
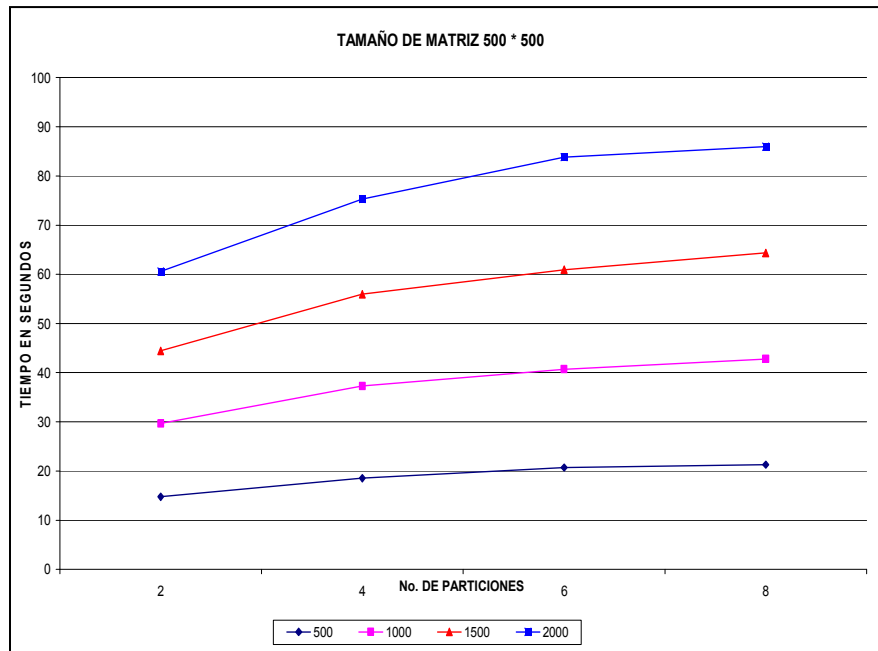
Los resultados arrojados por cada uno de los corrimientos hechos para este programa manipulando diferentes datos se muestran en las siguientes tablas. En la primera columna tenemos la cantidad de palabras que se ocultaron para la sopa de letras, en la segunda columna el tamaño de las matrices utilizadas, en la tercera el número de nodos empleados para dichos corrimientos, siguiendo con cada uno de los tiempos registrados en segundos que tardó el programa en ejecutarse. Es importante mencionar que se hicieron cuatro diferentes muestras de acuerdo a la cantidad de palabras, tamaño de la matriz y número de nodos utilizados.

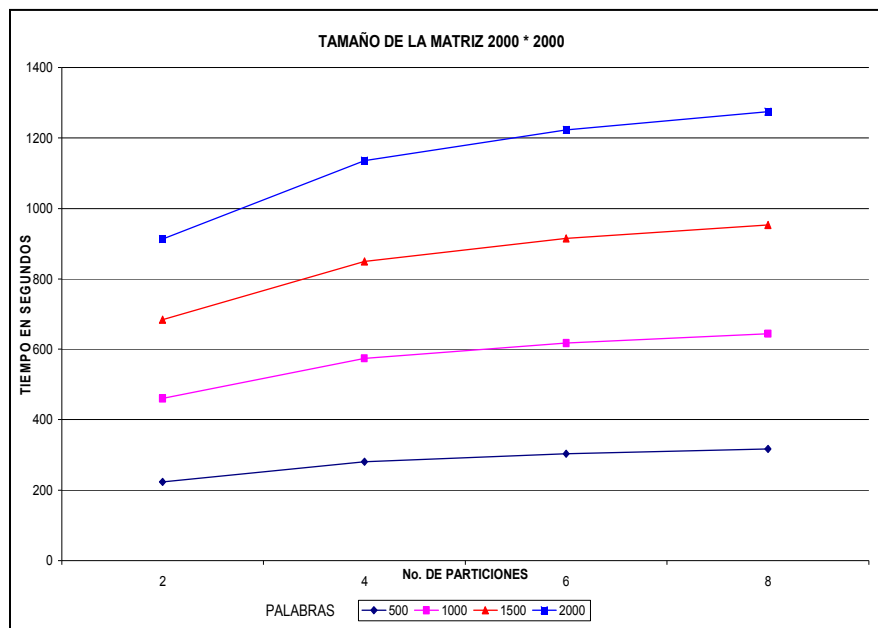
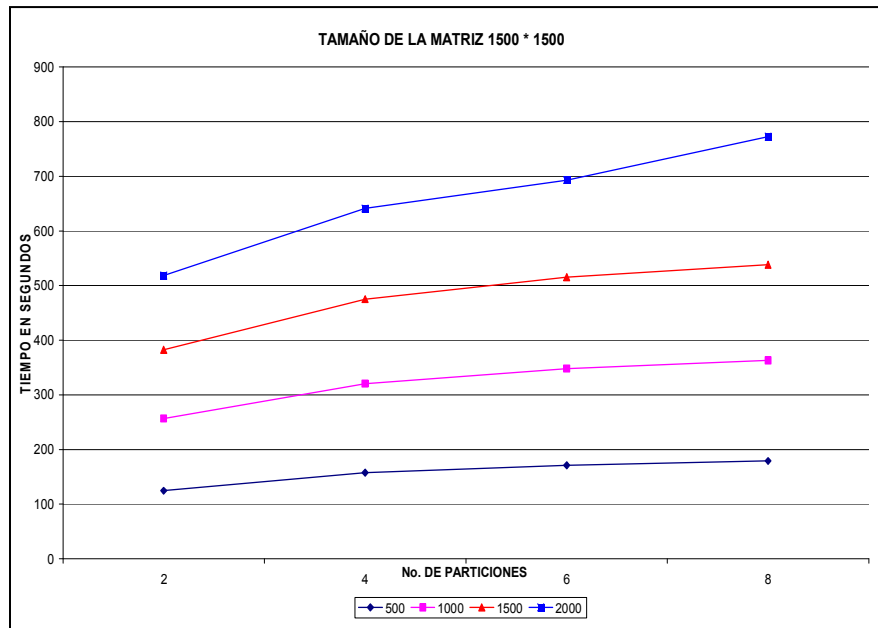
PALABRAS	TAMAÑO DE MATRIZ	PART.	TIEMPO	PART.	TIEMPO
500	500*500	2	14,773 s	4	18,520 s
1000	500*500	2	29,660 s	4	37,271 s
1500	500*500	2	44,446 s	4	55,944 s
2000	500*500	2	1m 0,574 s	4	1m 2,274 s
500	1000*1000	2	58,238 s	4	1m 2,133 s
1000	1000*1000	2	1m 9,934 s	4	2m 4,278 s
1500	1000*1000	2	2m 9,129 s	4	3m 6,346 s
2000	1000*1000	2	3m 9,853 s	4	4m 8,162 s
500	1500*1500	2	2 m 0,759 s	4	2m 6,764 s

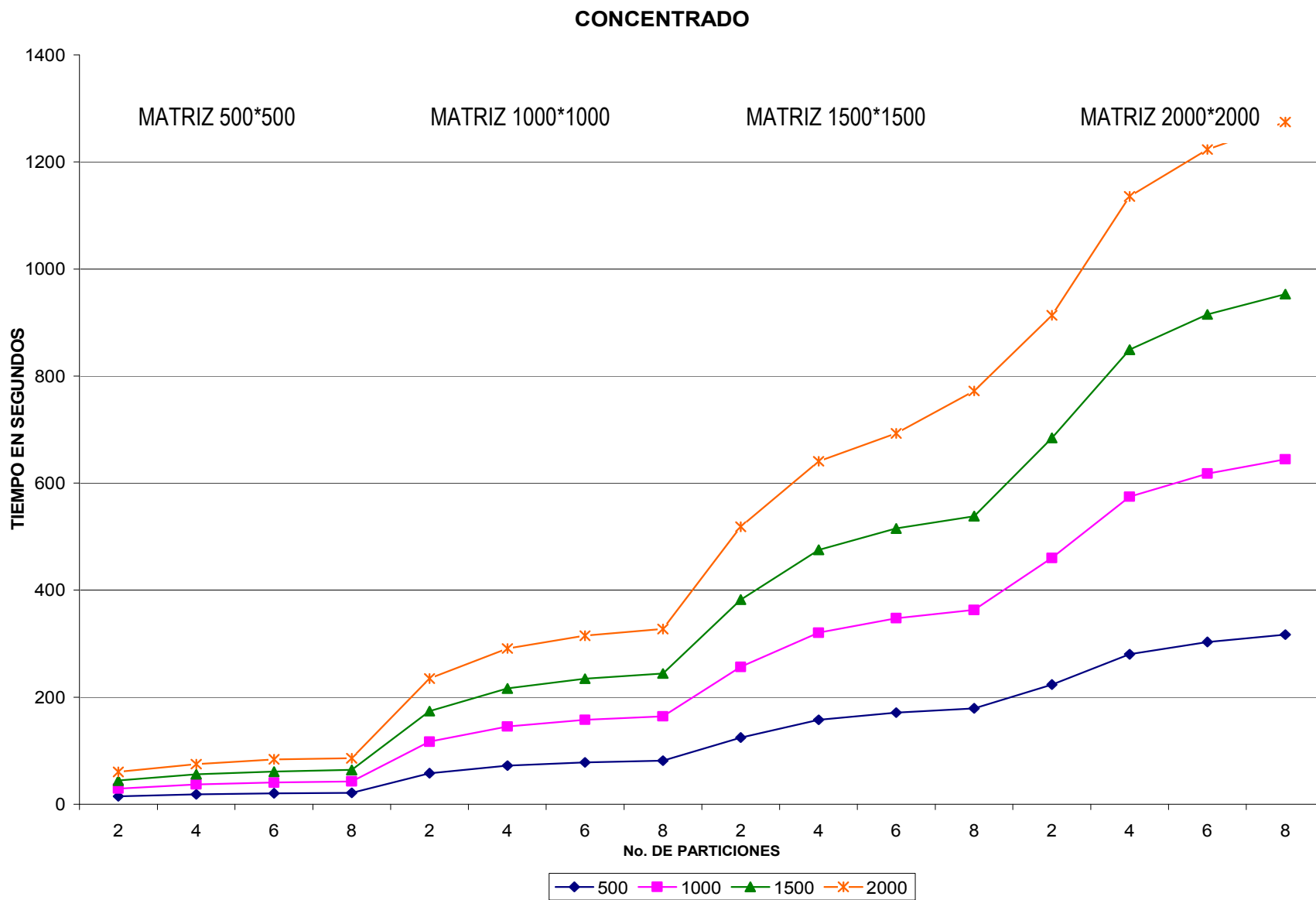
1000	1500*1500	2	4 m 2,670 s	4	5m 3,490 s
1500	1500*1500	2	6 m 3,346 s	4	7m 9,194 s
2000	1500*1500	2	8 m 6,529 s	4	10m 6,901 s
500	2000*2000	2	3m 7,527 s	4	4m 6,186 s
1000	2000*2000	2	7 m 6,269 s	4	7m 5,580 s
1500	2000*2000	2	11 m 4,326 s	4	14m 1,406 s
2000	2000*2000	2	15m 2,523 s	4	18m 9,474 s

PALABRAS	TAMAÑO DE MATRIZ	PART.	TIEMPO	PART.	TIEMPO
500	500*500	6	20,686 s	8	21,285 s
1000	500*500	6	40,685 s	8	42,795 s
1500	500*500	6	1m 0,931 s	8	1m 1,726 s
2000	500*500	6	1m 3,795 s	8	1m 4,920 s
500	1000*1000	6	1m 3,396 s	8	1m 3,580 s
1000	1000*1000	6	2m 6,695 s	8	2m 7,421 s
1500	1000*1000	6	3m 9,565 s	8	4m 0,234 s
2000	1000*1000	6	5m 2,775 s	8	5m 4,644 s
500	1500*1500	6	2m 8,107 s	8	2m 9,179 s
1000	1500*1500	6	5m 7,844 s	8	6m 0,991 s
1500	1500*1500	6	8m 5,386 s	8	8m 9,124 s
2000	1500*1500	6	11m 5,970 s	8	12m 0,265 s
500	2000*2000	6	5m 0,353 s	8	5m 2,823 s
1000	2000*2000	6	10m 2,678 s	8	10m 7,395 s
1500	2000*2000	6	15m 2,952 s	8	15m 8,973 s
2000	2000*2000	6	20m 3,127 s	8	21m 2,406 s

A continuación se presentan de forma gráfica los resultados obtenidos, es importante notar que en este programa específicamente la reducción del tiempo con respecto a la paralelización del programa no se reduce y al contrario crece de forma proporcional a medida que crece el número de particiones.







CONCLUSIONES

En base al conocimiento y a las aplicaciones desarrolladas bajo la librería de PVM en un cluster tipo Beowulf se concluye que:

PVM emula una máquina virtual paralela con un conjunto de computadoras heterogéneas conectadas entre si a través de una red local.

Al observar el rendimiento de los programas, en el caso del problema de la *Sopa de Letras*, podemos observar que debido a la cantidad de comunicación que se necesita en PVM, el tiempo de ejecución del programa aumenta al aumentar los nodos o computadoras, esto debido a que es una aplicación de grano fino y la cantidad de comunicación entre los nodos a través de la red aumentan, lo cual deriva en el incremento global del tiempo.

En el caso del *Infierno* que representa un flujo, el resultado obtenido era el esperado ya que entre mas densa sea la población mayor es el tiempo y la cantidad de movimientos aumenta para pasar de un lado a otro y en un momento dado si la población crece se puede llegar a un caos en el cual nadie puede avanzar. También, observamos que entre mayor sea la cantidad de nodos menor será el tiempo de procesamiento de datos dada una cantidad de ciclos o pasadas suficientes en cada caso para llegar a su destino.

La solución de un problema en paralelo permite alcanzar tiempos de ejecución mucho menores, en una escala, no lineal, relacionada al número de computadoras miembros de la red.

Como podemos observar en los resultados del programa de la Sopa de Letras no todos los problemas que requieran una gran cantidad de tiempo de ejecución es recomendable paralelizar. La decisión de paralelizar en un sistema como el de la máquina virtual paralela a través de una red dependerá de varios factores:

- En primer lugar dependerá del problema que estamos tratando, y de si es verdaderamente necesario o útil la aplicación de un algoritmo paralelo.
- Es importante también el análisis del algoritmo paralelo que se vaya a utilizar o la paralelización de un algoritmo ya existente; porque esto significará un cambio importante en la lógica de programación misma a utilizar.
- El tipo de paralelización a usar también dependerá de los recursos en Hardware que se tenga disponibles y del problema mismo al que se le quiere dar solución.
- Con estos factores considerados será posible establecer la eficiencia de una implementación en paralelo comparada a una solución en serie.

Trabajo a futuro

El caso del *Infierno* se puede implementar para la simulación de algoritmos evolutivos para observar el crecimiento de una población y el tamaño de espacio destinado para su desarrollo.

En protección civil para determinar si las salidas de una ciudad son suficientes en caso de alguna evacuación a causa de alguna inundación, erupción de volcán o cualquier otra eventualidad no controlada.

Para ciudades muy pobladas se puede hacer la estimación de cuanto pueden soportar la sobrepoblación antes de que se colapsen.

La *Sopa de Letras* puede ser utilizada para determinar que búsquedas son las más frecuentes en internet de acuerdo a las palabras claves.

Tomando como principio la búsqueda de cada palabra se puede implementar en algoritmos de cifrado.

APENDICE A

LINUX

La historia de Linux está fuertemente vinculada a la del proyecto GNU. El proyecto GNU (iniciado en 1983) tiene como objetivo el desarrollo de un sistema Unix completo compuesto enteramente de software libre ^[9]. Hacia 1991, cuando la primera versión del núcleo Linux fue liberada, el proyecto GNU había producido varios de los componentes del sistema, incluyendo un intérprete de comandos, una biblioteca C y un compilador.

El proyecto GNU aún no contaba (y aún no lo hace) con el núcleo que definiría un sistema operativo. Sin embargo, el núcleo creado por Linus Torvalds ^[10], llenó el hueco final que el sistema operativo GNU exigía.

Cabe mencionar que a pesar de que el núcleo de Linux se libera bajo los términos de la licencia GPL ^[11], no es parte oficial del proyecto GNU.

El término *Linux* estrictamente se refiere al núcleo Linux ^[12], pero es más comúnmente utilizado para describir al sistema operativo tipo Unix (de estándar POSIX) ^[13], que utiliza primordialmente filosofía y metodologías libres (también conocido como GNU/Linux) y que está formado mediante la combinación del núcleo Linux con las bibliotecas y herramientas del proyecto GNU y de muchos otros proyectos/grupos de software (libre o no). El núcleo no es parte oficial del proyecto GNU (el cual posee su propio núcleo en desarrollo, llamado Hurd), pero es distribuido bajo los términos de la licencia GPL (GNU General Public License).

También es utilizado para referirse a las distribuciones Linux, las cuales suelen contener grandes cantidades de software además del núcleo. El software que suelen incluir consta de una enorme variedad de aplicaciones, como: entornos gráficos, suites ofimáticas, servidores Web, servidores de correo, servidores FTP, etcétera.

Desde su lanzamiento, Linux ha incrementado su popularidad en el mercado de servidores. Su gran flexibilidad ha permitido que sea utilizado en un rango muy amplio de sistemas de cómputo y arquitecturas: computadoras personales, supercomputadoras, dispositivos portátiles, etc.

⁹ Es el software que, una vez obtenido, puede ser usado, copiado, estudiado, modificado y redistribuido libremente.

¹⁰ Hacker finlandés que pretendía obtener un sistema operativo libre similar a Unix que funcionara con microprocesadores Intel 80386.

¹¹ Licencia pública general; es una licencia creada por la Free Software Foundation y orientada principalmente a los términos de distribución, modificación y uso de software. Su propósito es declarar que el software cubierto por esta licencia es software libre.

¹² En informática, el **kernel** (también conocido como **núcleo**) es la parte fundamental de un sistema operativo. Es el software responsable de facilitar a los distintos programas acceso seguro al hardware de la computadora

¹³ Es un sistema operativo portable, multitarea y multiusuario. POSIX es el acrónimo de **P**ortable **O**perating **S**ystem **I**nterface, viniendo la **X** de UNIX (Sistema Operativo Portable basado en UNIX).

Los sistemas Linux funcionan sobre más de 20 diferentes plataformas de hardware, entre ellas las más comunes son las de los sistemas compatibles con PC, computadoras Macintosh, procesadores PowerPC, Sparc y MIPS.

Aquí se tiene una lista con las características de LINUX:

Multitarea: La palabra multitarea describe la habilidad de ejecutar varios programas al mismo tiempo.

LINUX utiliza la llamada *multitarea preventiva*, la cual asegura que todos los programas que se están utilizando en un momento dado serán ejecutados, siendo el sistema operativo el encargado de ceder tiempo de microprocesador a cada programa.

Características:

- *Multiusuario:* Muchos usuarios usando la misma maquina al mismo tiempo.
- *Multiplataforma:* Las plataformas en las que en un principio se puede utilizar Linux son 386-, 486-, Pentium, Pentium Pro, Pentium II, Amiga y Atari, también existen versiones para su utilización en otras plataformas, como Alpha, ARM, MIPS, PowerPC y SPARC.
- *Multiprocesador:* Soporte para sistemas con mas de un procesador esta disponible para Intel y SPARC.
- Funciona en *modo protegido 386*.
- *Protección de la memoria* entre procesos, de manera que uno de ellos no pueda colgar el sistema.
- *Carga de ejecutables por demanda:* Linux sólo lee del disco aquellas partes de un programa que están siendo usadas actualmente.
- Política de copia en escritura para la compartición de páginas entre ejecutables: esto significa que 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.
- *Memoria virtual usando paginación* (sin intercambio de procesos completos) a disco. A una partición o un archivo en el sistema de archivos, o ambos, con la posibilidad de añadir más áreas de intercambio sobre la marcha. Un total de 16 zonas de intercambio de 128Mb de tamaño máximo pueden ser usadas en un momento dado con un límite teórico de 2Gb 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.
- *Librerías compartidas* de carga dinámica (DLL's) y *librerías 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.
- Compatible con *POSIX*, System V y BSD a nivel fuente.
- Emulación de *iBCS2*, casi completamente compatible con SCO, SVR3 y SVR4 a nivel binario.

- Todo el *código fuente está disponible*, incluyendo el núcleo completo y todos los drivers, las herramientas de desarrollo y todos los programas de usuario; además todo ello se puede distribuir libremente. Hay algunos programas comerciales que están siendo ofrecidos para Linux actualmente sin código fuente, pero todo lo que ha sido gratuito sigue siendo gratuito.
- *Control de tareas* POSIX.
- *Pseudo-terminales* (pty's).
- *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 coma flotante), esta será usada en lugar de la emulación, pudiendo incluso compilar tu propio kernel sin la emulación matemática y conseguir un pequeño ahorro de memoria.
- Soporte para muchos teclados nacionales o adaptados y es bastante fácil añadir nuevos dinámicamente.
- *Consolas virtuales múltiples*: varias sesiones de login a través de la consola entre las que se puede cambiar con las combinaciones adecuadas de teclas (totalmente independiente del hardware de video). Se crean dinámicamente y puedes tener hasta 64.
- Soporte para *varios sistemas de archivo* comunes, incluyendo minix-1, Xenix y todos los sistemas de archivo típicos de System V, y tiene un avanzado sistema de archivos propio con una capacidad de hasta 4 Tb y nombres de archivos de hasta 255 caracteres de longitud.
- Acceso transparente a particiones MS-DOS (o a particiones OS/2 FAT) mediante un sistema de archivos especial: no es necesario ningún comando especial para usar la partición MS-DOS, esta parece un sistema de archivos normal de Unix (excepto por algunas restricciones en los nombres de archivo, permisos, y esas cosas). Las particiones comprimidas de MS-DOS 6 no son accesibles en este momento, y no se espera que lo sean en el futuro. El soporte para VFAT, FAT32 (WNT, Windows 95/98) se encuentra soportado desde la versión 2.0 del núcleo y el NTFS de WNT desde la versión 2.2 (Este último solo en modo lectura).
- Un sistema de archivos especial llamado UMSDOS que permite que Linux sea instalado en un sistema de archivos DOS.
- Soporte en sólo lectura de HPFS-2 del OS/2 2.1
- Sistema de archivos de CD-ROM que lee todos los formatos estándar de CD-ROM.
- *TCP/IP*, incluyendo ftp, telnet, NFS, etc.
- *AppleTalk*.
- Software cliente y servidor *Netware*.
- *Lan Manager / Windows Native (SMB)*, software cliente y servidor.
- Diversos *protocolos de red* incluidos en el kernel: TCP, IPv4, IPv6, AX.25, X.25, IPX, DDP, Netrom, etc.

Comandos básicos de Linux

Los comandos son esencialmente los mismos que cualquier sistema UNIX. En el siguiente listado se muestran los comandos mas frecuentes.

Información del sistema

- **hostname:** Devuelve el nombre de la maquina.
- **uptime:** Devuelve la cantidad de tiempo transcurrido desde la ultima vez que se arranco el sistema, la cantidad de usuarios trabajando en el sistema y el load average (carga del sistema).
- **uname -a:** Información sobre el sistema operativo de la maquina.
- **dmesg|more:** Imprime el "ring buffer" del kernel.
- **free -tm:** Información sobre la cantidad de memoria disponible y usada.
- **df -h:** Información sobre todo los dispositivos montados en la maquina.
- **du -bh /|more:** Información sobre el espacio ocupado por cada subdirectorio, comenzando en el directorio raíz /.
- **ps:** Información sobre los procesos del actual usuario, que se están ejecutando.
- **ps axu:** Información sobre todos los procesos que se están ejecutando en la maquina.
- **top:** Información sobre el uso de CPU y memoria de los procesos del sistema.
- **cat /proc/cpuinfo:** Información sobre el microprocesador.
- **cat /proc/interrupts:** Información sobre las interrupciones en uso.
- **cat /proc/dma:** Información sobre dma en uso.
- **cat /proc/filesystems:** Información sobre los sistemas de archivos que se pueden utilizar (compilados en el kernel).
- **/sbin/lsmmod:** Información sobre los módulos en uso.
- **finger usuario:** Información sobre el usuario *usuario*.
- **who:** Información sobre los usuarios usando el sistema.
- **id usuario:** Información sobre UID, GID y GROUPS del usuario *usuario*.
- **last:** Información sobre los últimos usuarios que han usado el sistema.
- **set|more:** Información sobre el entorno de usuario actual.
- **ping maquina:** Para comprobar si hay contacto con la máquina *maquina* (*maquina*: nombre o IP)
- **/sbin/route:** Tabla de enrutamiento del sistema.
- **/sbin/ifconfig:** Información sobre los distintos dispositivos de red.
- **netstat:** Información valiosa sobre la conexión de red (Este comando tiene muchas posibilidades, ejecutar `man netstat`

Administración

- **su:** Hace la conversión a administrador (root) después de introducir la clave de acceso. Ideal para realizar alguna tarea de administración sin necesidad de salir del sistema y entrar de nuevo como root.
- **/usr/sbin/adduser usuario:** Registra y crea una cuenta de usuario.
- **/usr/sbin/userdel usuario:** Borra la cuenta de usuario *usuario*.
- **passwd:** Cambia la clave de acceso para el usuario actual. Root puede cambiar la clave de cualquier usuario `passwd usuario`
- **/usr/sbin/groupadd grupo:** Crea un nuevo grupo.
- **/usr/sbin/groupdel grupo:** Borra un grupo.
- **more /etc/passwd:** Muestra el fichero de claves del sistema. (Si no se usa shadow password)

- **more /etc/group:** Muestra los grupos registrados en el sistema.
- **chmod *permisos fichero/directorio*:** Cambia los permisos de ficheros/directorios
- **cp *fichero1 fichero2*:** Copia fichero1 como fichero2.
- **rm *fichero*:** Borra fichero
- **rm -R *directorio*:** Borra el contenido completo (ficheros/subdirectorios) de directorio.
- **mv *fichero1 fichero2*:** Cambia el nombre de fichero1 a fichero2. También sirve para mover un archivo(s) a un directorio.
- **mkdir *directorio*:** Crea un subdirectorio
- **rmdir *directorio*:** Borra un subdirectorio
- **shutdown -t3 -r now:** Reinicializa el sistema (hay que hacerlo como root).
- **shutdown -t3 -h now:** Apaga el sistema (hay que hacerlo como root).
- **cat *fich1 [...fichN]*:** Concatena y muestra un archivo.
- **cd [*dir*]:** Cambia de directorio.
- **chown *usuario/grupo fich*:** Cambia el dueño un archivo.
- **Diff [-e] *arch1 arch2*:** Encuentra diferencia entre archivos.
- **file *arch*:** Muestra el tipo de un archivo.
- **find -name "*arch*":** Encuentra archivos
- **grep [-cilmv] *exp archivos*:** Busca patrones en archivos.
- **head -count *fich*:** Muestra el inicio de un archivo.
- **less / more *fich(s)*:** Visualiza página a página un archivo; less acepta comandos vi.
- **ln [-s] *fich acces*:** Crea un acceso directo a un archivo.
- **ls:** Lista el contenido del directorio.
- **pwd:** Muestra la ruta del directorio actual.
- **rm -r *dir*:** Borra todo un directorio.
- **rmdir *dir*:** Borra un directorio vacío.
- **tail -count *fich*:** Muestra el final de un archivo.
- **vi *fich*:** Edita un archivo.

Fedora

Fedora Core (también conocida como Fedora Linux) es una distribución Linux desarrollada por la comunidad Fedora y promovida por la compañía estadounidense Red Hat Linux ^[14].

El objetivo de Fedora es conseguir un sistema operativo de propósito general y basado exclusivamente en software libre con el apoyo de la comunidad Linux.

Originalmente, Red Hat Linux fue desarrollado exclusivamente dentro de Red Hat, con la sola realimentación de informes de usuarios que recuperaban fallos y contribuciones a los paquetes de software incluidos; y no contribuciones a la distribución como tal. Esto cam-

¹⁴ Es la compañía responsable de la creación y mantenimiento de la distribución del sistema operativo Linux que lleva el mismo nombre: Red Hat Linux. Es famoso en todo el mundo por los diferentes esfuerzos orientados a apoyar el movimiento del software libre.

bió, cuando Red Hat Linux se derivó dando origen al Proyecto Fedora que está orientado a la comunidad de usuarios y así mismo, sirve de base para que Red Hat Enterprise Linux se desarrolle con más efectividad y adopte las nuevas características que se añaden en el Proyecto Fedora.

Este modelo es similar a la relación entre Netscape Communicator y Mozilla, o entre StarOffice y OpenOffice.org, aunque en este caso, el producto comercial resultante es software libre.

Se llama Core porque incluye solo las partes más básicas del sistema operativo. Para incluir nuevo software, hay que usar los llamados Repositorios^[12]. Actualmente hay dos familias de repositorios incompatibles entre si:

- **Livna**
- **RPMForge**: FreshRPM, Dag, NewRpms y Dries

En Livna están aquellos paquetes que, aunque legales, solo pueden ser descargados por el usuario final, como códecs para MP3 y otros formatos. La segunda familia no tiene los paquetes clasificados por licencias sino por funcionalidad. Los paquetes de ATrpms pueden causar problemas de incompatibilidad pero se engloban en la familia de RPMForge.

Actualmente ambas familias son incompatibles entre ellas y se culpan mutuamente de la incompatibilidad.

La herramienta habitual en Fedora para usar los repositorios es la aplicación en línea de comandos Yum, aunque hay un entorno gráfico para ella llamado Yumex.

Un **repositorio** es un sitio centralizado donde se almacena y mantiene información, habitualmente bases de datos o archivos informáticos. Está preparado para distribuirse habitualmente sirviéndose de una red informática como Internet o en un medio físico como un disco compacto.

APENDICE B

LENGUAJE C

C es un lenguaje de programación de propósito general que ofrece economía sintáctica, control de flujo y estructuras sencillas y un buen conjunto de operadores. No es un lenguaje de muy alto nivel y más bien un lenguaje pequeño, sencillo y no está especializado en ningún tipo de aplicación. Esto lo hace un lenguaje potente, con un campo de aplicación ilimitado y sobre todo, se aprende rápidamente. En poco tiempo, un programador puede utilizar la totalidad del lenguaje.

Este lenguaje ha sido estrechamente ligado al sistema operativo UNIX, puesto que fueron desarrollados conjuntamente. Sin embargo, este lenguaje no está ligado a ningún sistema operativo ni a ninguna máquina concreta. Se le suele llamar *lenguaje de programación de sistemas* debido a su utilidad para escribir compiladores y sistemas operativos, aunque de igual forma se pueden desarrollar cualquier tipo de aplicación.

La base del C proviene del BCPL, escrito por Martin Richards, y del B escrito por Ken Thompson en 1970 para el primer sistema UNIX en un DEC PDP-7. Estos son lenguajes sin tipos, al contrario que el C que proporciona varios tipos de datos. Los tipos que ofrece son caracteres, números enteros y en coma flotante, de varios tamaños. Además se pueden crear tipos derivados mediante la utilización de punteros, vectores, registros y uniones. El primer compilador de C fue escrito por Dennis Ritchie para un DEC PDP-11 y escribió el propio sistema operativo en C.

C trabaja con tipos de datos que son directamente tratables por el hardware de la mayoría de computadoras actuales, como son los caracteres, números y direcciones. Estos tipos de datos pueden ser manipulados por las operaciones aritméticas que proporcionan las computadoras. No proporciona mecanismos para tratar tipos de datos que no sean los básicos, debiendo ser el programador el que los desarrolle. Esto permite que el código generado sea muy eficiente y de ahí el éxito que ha tenido como lenguaje de desarrollo de sistemas. No proporciona otros mecanismos de almacenamiento de datos que no sea el estático y no proporciona mecanismos de entrada ni salida. Ello permite que el lenguaje sea reducido y los compiladores de fácil implementación en distintos sistemas. Por contra, estas carencias se compensan mediante la inclusión de funciones de librería para realizar todas estas tareas, que normalmente dependen del sistema operativo.

Originariamente, el manual de referencia del lenguaje para el gran público fue el libro [1] de Kernighan y Ritchie, escrito en 1977. Es un libro que explica y justifica totalmente el desarrollo de aplicaciones en C, aunque en él se utilizaban construcciones, en la definición de funciones, que podían provocar confusión y errores de programación que no eran detectados por el compilador. Como los tiempos cambian y las necesidades también, en 1983 ANSI establece el comité X3J11 para que desarrolle una definición moderna y comprensible del C. El estándar está basado en el manual de referencia original de 1972 y se desarrolla con el mismo espíritu de sus creadores originales. La primera versión de estándar se publicó en 1988 y actualmente todos los compiladores utilizan la nueva definición. Una aportación muy importante de ANSI consiste en la definición de un conjunto de librerías

que acompañan al compilador y de las funciones contenidas en ellas. Muchas de las operaciones comunes con el sistema operativo se realizan a través de estas funciones. Una colección de ficheros de encabezamiento, *headers*, en los que se definen los tipos de datos y funciones incluidas en cada librería. Los programas que utilizan estas bibliotecas para interactuar con el sistema operativo obtendrán un comportamiento equivalente en otro sistema.

Algunos de las C existentes son:

- Quick C
- C++
- Turbo C
- Turbo C ++
- Borland C
- Borland C++
- Microsoft C
- etc.

C es un lenguaje de programación de nivel medio ya que combina los elementos del lenguaje de alto nivel con la funcionalidad del ensamblador.

Su característica principal es ser portable, es decir, es posible adaptar los programas escritos para un tipo de computadora en otra.

Otra de sus características principales es el ser estructurado, es decir, el programa se divide en módulos (funciones) independientes entre sí.

El lenguaje C inicialmente fue creado para la programación de:

- Sistemas operativos
- Intérpretes
- Editores
- Ensambladores
- Compiladores
- Administradores de bases de datos.

Actualmente, debido a sus características, puede ser utilizado para todo tipo de programas.

Elementos generales de un programa en C

Aunque cada uno de los programas es distinto, todos tienen características comunes. Los elementos de un programa en C son los siguientes:

Comentarios

Inclusión de archivos

main()

{

variables locales

flujo de sentencias

}

Definición de funciones creadas por el programador utilizadas en main()

A continuación se explica brevemente en que consiste cada uno:

Comentarios: Se identifican porque van entre diagonales y asterisco. Sirve para escribir información que referencia al programa pero que no forme parte de él. Por ejemplo especificar que hace el programa, quien lo elaboró, en que fecha, que versión es, etc.

Inclusión de archivos: Consiste en mandar llamar a la o las bibliotecas donde se encuentran definidas las funciones de C (instrucciones) que se utilizan en el programa.

En realidad, la inclusión de archivos no forma parte de la estructura propia de un programa sino que pertenece al desarrollo integrado de C. Se incluye para no olvidar que se debe mandar llamar a los archivos donde se encuentran definidas las funciones estándar que se van a utilizar.

main(): En C, todo está constituido a base de funciones. El programa principal no es la excepción. *main()* indica el comienzo de la función principal del programa la cual se delimita con llaves.

Variables locales: Antes de realizar alguna operación en el programa, se deben declarar la(s) variable(s) que se utilizarán en el programa.

Flujo de sentencias: Es la declaración de todas las instrucciones que conforman el programa.

Definición de funciones creadas por el programador utilizadas en main(): Finalmente, se procede a definir el contenido de las funciones utilizadas dentro de *main()*. Estas contienen los mismos elementos que la función principal.

Importante:

Después de cada asignación o función es imprescindible colocar un punto y coma (;) ya que éste es un terminador de proposiciones. En caso de que no se escriba, se marcará un error a la hora de compilar el programa.

En C, los comandos deben ser escritos con letras minúsculas. En el caso de las variables o las funciones definidas por el usuario la situación es similar:

No es lo mismo: *Apellido* que, *apellido* que, *APELLIDO*

Aunque para nosotros es lo mismo, el compilador de C, los toma como tres nombres distintos. Por tanto, se debe asegurar mandar a llamar las variables o funciones exactamente de la misma forma en que las declaró.

BIBLIOGRAFIA

[1] COMPONENTES DE UN CLUSTER

[http://es.wikipedia.org/wiki/Cluster_\(inform%C3%A1tica\)](http://es.wikipedia.org/wiki/Cluster_(inform%C3%A1tica))

[2] CARACTERISTICAS GENERALES DE UN CLUSTER

http://es.tldp.org/almacen/Manuales-LuCAS/doc-manual-openMosix-1.0/doc-manual-openMosix_html-1.0/node16_ct.html#SECTION00712000000000000000

[3] FUNCIONAMIENTO DE UN CLUSTER

<http://talika.eii.us.es/~rosa/clustering.pdf>
E.T.S.I.I. Sevilla

[4] ARQUITECTURA DE COMPUTADORAS

<http://telematica.cicese.mx/computo/super/cicese2000/paralelo/Part3.html>

[5] PVM Y XPV

<http://es.tldp.org/Manuales-LuCAS/doc-cluster-computadoras/doc-cluster-computadoras-html/node48.html>

[6] CONFIGURACION DEL ENTORNO Y EJEMPLOS

http://www-copa.dsic.upv.es/docencia/iblanque/lcp/Boletines/Boletin1_2004.pdf

[7] Torres, J. and E. Rodríguez, Conceptos de cómputo paralelo, Ed ITESM Universidad Virtual. México, D.F. Trillas.

[8] UNAM. Conceptos básicos de los Clusters Departamento de Super Cómputo

<http://clusters.unam.mx/Conceptos/Index.php>

[9] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, 1975.

[10] R. Babb II (ed.), Programming Parallel Processors, Addison-Wesley Inc, 1988.

[11] Geist, A., et al., PVM 3 User's Guide and Reference Manual, Rep. ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee USA, 1993. En
<http://www.netlib.org/pvm3/>

[12] PVM Version 3.4, Manual en línea,

<http://venus.arcride.edu.ar/calculoparalelo/manpvm.html>

[13] Top 500 SuperComputer, <http://www.top500.org/>

[14] Ohio Supercomputer Center training resources, <http://oscinfo.osc.edu/training>

[15] J. Dongarra, <http://www.netlib.org/utk/people/JackDongarra/courses.html>

- [16] PVM and MPI: a Comparison of Features, G. A. Geist, J. A. Kohl, P. M. May 30. 1996
- [17] Mónica Rivera de la Rosa, PROGRAMACION PARALELA, Descomposición LU DE GRANO FINO Versión con PVM, Centro de Investigación y Estudios Avanzados del IPN. 20-Julio-2004
- [18] Armando Fox; Steven Gribble; Yatin Chawathe; Eric Brewer; Paul Gauthier. Cluster-based scalable network services. 1997
- [19] Francisco Javier Jiménez Gómez 2003-02-26
<http://personales.alumno.upv.es/~frajigo/proyectos/cluster/documentos/libro-pocluster/node1.html>
- [20] Universidad de Talca Facultad de Ingeniería, Computación Paralela y Distribuida PVM: Parallel Virtual Machina, <http://ing.otalca.cl/~fmeza/cursos/2007-2/ppd/docs/pvm.pdf>
- [21] Hoffman, F and W. Hargrove, 1999, Computación Paralela con Linux, ACM Cross-roads Students Magazine
- [22] Jim Pruyne and Miron Livny, Providing resource management services to parallel applications. In Proceedings of the Second Workshop on Environments and Tools for Parallel Scienti_c Computing, 1991, <http://www.cs.wisc.edu/condor/publications.html>
- [23] A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, B.; Sunderam, V , PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Network Parallel Computing Geist,. The MIT Press, 1994