UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

Construcción y evaluación de desempeño de un *cluster* tipo *Beowulf* para cómputo de alto rendimiento

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A :

DANIEL MANRIQUE MARTINEZ

DIRECTORA: ING. LAURA SANDOVAL MONTAÑO.

MEXICO, D.F. 2001

Construcción y evaluación de desempeño de un *cluster* tipo *Beowulf* para cómputo de alto rendimiento

Daniel Manrique M.

24 de octubre de 2001

Índice General

Agradecimientos

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

- Ing. Laura Sandoval, cuya colaboración en la realización del seminario que desembocó en este proyecto, su dirección durante la realización del mismo y su apoyo, entusiasmo y paciencia durante el tiempo que tomó completar el proyecto, fueron esenciales.
- Los integrantes del Laboratorio de Telemática de la División de Ingeniería en Computación de la Facultad de Ingeniería, por dar asilo al *cluster* y soportar las infernales temperaturas generadas por el mismo.
- Todos los miembros (y no miembros) del Depto. de Física y Química Teórica de la Facultad de Química, en particular Alan Aspuru Guzik, culpables directos de mi interés por el cómputo paralelo y los *clusters*.

Introducción

El problema

Una de las características básicas de las computadoras es su capacidad para realizar grandes cantidades de operaciones o cálculos en tiempos muy breves, facilitando la realización de tareas que requieren de dichos cálculos. Como evidencia de la importancia de esta capacidad particular de las computadoras, se aprecia que muchos de los avances en tecnologías de computación están encaminados a incrementar la velocidad de procesamiento de las computadoras.

De entre los diversos enfoques o técnicas que existen para obtener un mejor desempeño en la realización de tareas de cómputo, destaca el concepto de *clusters*, que básicamente involucra cooperación entre sistemas completos y por lo demás independientes (nodos de procesamiento) para realizar una tarea común. Los *clusters* tienen ciertas ventajas y desventajas frente a otros esquemas de cómputo paralelo y otras técnicas utilizadas para obtener computadoras más rápidas (como son las supercomputadoras tradicionales de procesadores vectoriales).

De particular interés son los *clusters* tipo *Beowulf*, que se definen como "una clase de computadora masivamente paralela de alto rendimiento construida primordialmente con componentes comerciales ampliamente disponibles". En realidad esto consiste en un grupo de PCs dedicadas a ejecutar en conjunto tareas de cómputo que requieren alto rendimiento.

Tradicionalmente, un *Beowulf* se construye con PCs completamente independientes que se adquieren expresamente con el fin de construir un *cluster*. Sin embargo, ya que esto en ocasiones es costoso y puede no resultar factible el desembolso de fondos para un proyecto de esta naturaleza, resulta interesante el buscar

otras opciones para crear un *Beowulf*.

Se puede decir que el presente proyecto buscará la solución a las dos preguntas siguientes: ¿Se puede construir un *Beowulf* únicamente integrando equipo existente, disponible y en desuso? y ¿Qué clase de desempeño se puede obtener de un *cluster* con tales características?.

Objetivo

Se planteó como objetivo construir un *cluster* tipo *Beowulf* integrando el equipo disponible y realizar la evaluación de su desempeño en la resolución de un problema real.

Para alcanzar el primer objetivo, se requiere, primero, contar con equipo; se deberá, pues, localizar equipo que se encuentre en desuso, pero en condiciones operativas; una vez contando con dicho equipo, analizar los requerimientos para la construcción de un *Beowulf* y obtener los componentes necesarios, buscando que el costo sea el menor posible. Contando con la plataforma de *hardware* necesaria, se deberá conseguir, instalar y configurar el *software* requerido para obtener un *Beowulf* completo y funcional.

El segundo objetivo requiere la ejecución de programas que aprovechen los recursos del *Beowulf*, la comparación de desempeño variando sus parámetros operativos, y el análisis de estos datos para determinar exactamente el comportamiento del *Beowulf* en aplicaciones reales.

Resultados esperados

Se espera lograr integrar los equipos disponibles en un *cluster* tipo *Beowulf*, que efectivamente resulte una plataforma de cómputo estable y de alto rendimiento, y pueda utilizarse para resolver problemas por medio de software que aproveche las capacidades del *Beowulf*.

Idealmente, el *Beowulf* podrá utilizarse para correr diversas aplicaciones, de particular interés resulta su utilización para ejecutar alguna aplicación de utilidad en la resolución de un problema real. Al ejecutar esta aplicación, los resultados

de las pruebas de desempeño deberían permitir llegar a la conclusión de que el *Beowulf* brinda un desempeño mayor que el de una solución de un solo procesador.

Capítulo 1

Teoría

1.1 Contexto histórico

> "Una computadora es una máquina que puede programarse para manipular símbolos. Las computadoras pueden realizar procedimientos complejos y repetitivos de forma precisa y confiable, y pueden almacenar y recuperar rápidamente enormes cantidades de datos". [22]

> "Una computadora es un dispositivo que acepta información en la forma de datos digitales, y la manipula para obtener un resultado basado en un programa o secuencia de instrucciones que indican cómo procesar los datos". [23]

> "Un dispositivo que computa, especialmente una máquina electrónica programable que realiza operaciones lógicas o matemáticas a altas velocidades, o que ensambla, almacena, correlaciona o de alguna otra manera procesa la información". [24]

En la actualidad la presencia de las computadoras es inescapable, sin embargo muchas personas quizá nunca se preguntan exactamente qué es lo que hace una computadora. Las definiciones mostradas proporcionan una idea de lo que una computadora "debe ser". Es de interés resaltar dos conceptos centrales en estas definiciones: una computadora es un dispositivo que manipula o procesa informa-

ción de acuerdo a ciertas instrucciones; y la computadora realiza dichos procesos de manera confiable, precisa y a gran velocidad.

Estos aspectos han estado presentes desde el surgimiento de los primeros dispositivos auxiliares para realizar cálculos. Quizá el primer dispositivo mecánico de cálculo fue la sumadora de Pascal (1642). Junto con la máquina multiplicadora de Leibniz (1673), estos dispositivos permitían realizar cálculos con mayor velocidad.

La primera máquina creada específicamente para realizar cálculos de acuerdo a una secuencia de instrucciones definida fue la máquina analítica de Babbage. Esta máquina fue concebida en la primera mitad del siglo XIX, sin embargo nunca fue construida. Babbage tambien ideó la máquina de diferencias, cuyo propósito era calcular tablas numéricas para diversos propósitos. Nótese que el objetivo era calcular dichas tablas con rapidez y precisión.

A fines del siglo XIX, máquinas tabuladoras creadas por Herman Hollerith fueron utilizadas para procesar los datos del censo de 1890 en Estados Unidos. Si bien estas máquinas se utilizaban básicamente para contabilizar los datos, la reducción del tiempo requerido para procesar la información fue espectacular, ocupando sólo 2 años para una tarea que de otro modo habría tomado más de 10.

La aparición de componentes eléctricos y electrónicos dio lugar a avances en el campo de la automatización de cálculos en la primera mitad del siglo XX, cuando varios investigadores buscaban maneras de realizar cálculos aritméticos más rápida y eficientemente. La Segunda Guerra Mundial fue el catalizador que aceleró el desarrollo de las primeras computadoras digitales, sin embargo antes de la guerra Alan Turing introdujo el concepto de procesamiento simbólico, y la idea de una máquina universal (máquina de Turing) capaz de ejecutar cualquier algoritmo que pudiera describirse.

Durante la guerra se desarrollaron máquinas cuyo propósito era de apoyo a actividades bélicas, como desciframiento de mensajes en clave y cálculo de tablas de artillería. Durante este tiempo se iniciaron trabajos que darían como resultado, ya concluida la guerra, las primeras máquinas que pueden realmente llamarse computadoras, como la ENIAC, EDSAC, UNIVAC y otras.

Concluida la guerra, las computadoras se siguieron empleando para realizar cálculos rápidamente. También se comenzaron a emplear para propósitos no bélicos, como cálculos en empresas y apoyo a la investigación científica. A medida que

la demanda de equipos de cómputo se incrementaba y se requería mayor velocidad de procesamiento, se fueron desarrollando y refinando diseños que proporcionaban mayor velocidad de cálculo. Para 1955 la IBM 704 podía realizar 5 kFLOPS [1] gracias a su unidad de procesamiento de punto flotante.

Independientemente de la evolución generacional de las computadoras, es conveniente resaltar los incrementos en rendimiento que se fueron logrando, como consecuencia de la necesidad de realizar los cálculos de manera más rápida.

Para 1962, la computadora Atlas de la universidad de Manchester realizaba 200 kFLOPS, empleando unidades aritméticas separadas para punto fijo y flotante y *pipelining* de instrucciones.

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

En 1997, la ASCI Red, construida por Intel, alcanzó la marca de 1 TFLOPS.

En el año 2000, la computadora más rápida del mundo, la ASCI White, construida por IBM en el laboratorio nacional Lawrence Livermore, en Estados Unidos, alcanzó un rendimiento de alrededor de 4 TFLOPS con una velocidad pico de más de 12 TFLOPS; para el año 2001, dicho equipo ha sido expandido y alcanza un rendimiento sostenido de 7.2 TFLOPS.

Esta tendencia de crecimiento continuará en un futuro; recientemente se anunciaron planes para la creación de computadoras que alcancen los 30 TFLOPS para el año 2002, y 100 TFLOPS para el año 2004.

1.1.1 La necesidad de computadoras verdaderamente rápidas

¿De dónde viene la necesidad de contar con esta capacidad de procesamiento? Algunos cálculos sencillos darían la idea de que esta capacidad de cálculo es más que suficiente para aplicaciones empresariales, uno de los usos más frecuentes para una computadora.

En realidad la necesidad de computadoras realmente rápidas viene de la comunidad científica. En una entrevista, Seymour Cray[2] describe el ámbito del cómputo científico a mediados de los 60:

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

[2] Pionero en el campo del cómputo de alto rendimiento y fundador de la empresa *Cray Research*, el fabricante de supercomputadoras más conocido.

> (...) Tal como lo percibía, la comunidad científica apenas estaba descubriendo que podía resolver ecuaciones diferenciales parciales en computadoras por medio de un proceso iterativo, el análisis por elemento finito apenas comenzaba a apreciarse. Repentinamente, existe un requerimiento casi infinito de capacidad de cómputo, pues se vio claramente que entre más pasos se pudieran realizar en una solución iterativa, mejor sería la solución, por tanto al modelar algo como el clima o, en la milicia, las aplicaciones de modelado de reacciones nucleares, todas estas cosas requieren la solución de ecuaciones diferenciales donde podías dividirlos en cuantas unidades pequeñas pudieran imaginarse y únicamente se estaría limitado por el poder de cómputo para hacerlo a ese nivel de sofisticación (...) [25]

Así pues, la necesidad de contar con capacidad de realizar cálculos cada vez con mayor velocidad viene, básicamente, de aplicaciones científicas que no podrían realizarse de no contar con una manera rápida de realizar los cálculos involucrados. De hecho, al incrementarse el poder de las computadoras, las aplicaciones actuales pueden realizarse con más velocidad; pero también se abren las puertas a nuevas aplicaciones que con la capacidad anterior no eran factibles. Se tiene así un "círculo vicioso" donde los diseñadores de computadoras deben proporcionar equipos cada vez más rápidos que a su vez habilitan nuevas aplicaciones, que exigirán en cierto momento un nivel de desempeño mayor.

1.2 Diferentes enfoques para incrementar el rendimiento

A lo largo de la historia de las computadoras se han desarrollado y refinado técnicas para obtener un mayor rendimiento; es decir, realizar un mayor número de cálculos en el mismo tiempo. Dichas técnicas son de diversas índoles, algunas complejas, otras relativamente sencillas, algunas son diseños basados en tecnología existente mientras que otras se basan en la introducción de tecnología novedosa.

El primer salto tecnológico para mejorar el desempeño se dio con el advenimiento de las computadoras electrónicas, que gracias a su tecnología básica eran inherentemente mucho más rápidas que las mecánicas. Similarmente, al comenzar

a utilizarse semiconductores y, posteriormente, circuitos integrados, se ha obtenido mayor velocidad.

La evolución en tecnología semiconductora y de circuitos integrados ha sido el vehículo para incrementar el rendimiento de la manera más obvia posible: aumentando la frecuencia de operación de las computadoras. Esto naturalmente tiene el efecto de realizar mayor número de operaciones en el mismo tiempo, y ha sido posibilitado por el desarrollo de tecnologías y elementos capaces a operar a frecuencias cada vez mayores.

Sin embargo los desarrollos más obvios suelen resultar también costosos y complicados; a medida que se incrementa la frecuencia de operación se introducen mayores problemas eléctricos y térmicos en el circuito. La tecnología requerida para resolver estos problemas es costosa, de modo que en ocasiones se obtiene un incremento de rendimiento mínimo por un precio bastante elevado.

A raíz de esto se comenzaron a desarrollar otras técnicas que permitieran obtener un mayor rendimiento con menor costo y complicación.

Se han creado técnicas como la memoria de modo página, que explota el hecho de que las lecturas a memoria suelen ser lineales, reorganizando la memoria de forma que páginas que lógicamente están contiguas se encuentren físicamente en elementos separados, evitando así el efecto de latencia de lectura de la memoria; y las memorias caché, que aprovechan la propiedad de localidad de los programas, la cual indica que la mayoría del tiempo se emplea ejecutando ciertas porciones pequeñas del programa. Estas técnicas son relativamente simples de implementar y basan su funcionamiento en premisas que pueden no cumplirse en todos los casos; sin embargo son de costo relativamente bajo de modo que la relación costo/beneficio es grande y en la práctica redundan en incrementos de desempeño considerables.

La creación de arquitecturas o paradigmas de diseño nuevos también ha proporcionado mayor desempeño utilizando elementos tecnológicos existentes. La arquitectura RISC[3] plantea ciertas modificaciones a la manera tradicional como se hacían los microprocesadores. Estos incluyen la reducción del juego de instrucciones y el uso de instrucciones de longitud fija. Esto permite ejecutar el código a mucho mayor velocidad, de forma que una cantidad mayor de instrucciones se

[3] *Reduced Instruction Set Computer*, computadora con juego de instrucciones reducido.

ejecuta en un tiempo menor que el equivalente en un procesador CISC [4]. A cambio de este mayor rendimiento, es más complejo escribir código para procesadores RISC y dicho código suele ocupar más espacio de almacenamiento. Similarmente al enfoque tomado por las memorias caché, el enfoque RISC asume que el costo del almacenamiento es menor que el costo de un procesador más rápido.

El uso de *pipelines*[5] para aprovechar todas las etapas de un procesador (*fetch*, *decode*, *execute*) simultáneamente y procesar un promedio de una instrucción por ciclo de reloj (a diferencia del enfoque anterior que plantea utilizar sólo una etapa a la vez y, en un procesador de tres etapas, ejecutaría una instrucción en tres ciclos de reloj) es una técnica muy simple y que también proporciona una mejora de desempeño sustancial. Esta medida condujo directamente al desarrollo de procesadores con múltiples unidades funcionales (por ejemplo, un procesador con dos unidades lógico-aritméticas) y al uso de *superpipelining*, con el cual el procesador es capaz de ejecutar más de una instrucción por ciclo de reloj. Desde luego esto requiere "cooperación" de parte del código que se va a ejecutar; se necesita que el código esté organizado de manera que instrucciones contiguas se puedan ejecutar simultáneamente. Esto a su vez ha llevado a desarrollar varias sub-técnicas para reorganizar el código: se tienen procesadores que pueden reorganizar "al vuelo" el código existente, sin requerir modificaciones a los programas actuales, sin embargo este enfoque incrementa notablemente la complejidad de la electrónica de decodificación de instrucciones del procesador; también se han desarrollado nuevas arquitecturas como la VLIW[6], en las cuales el código se entrega al procesador en "paquetes" o *bundles* que el compilador ha identificado como ejecutables en paralelo. Esto reduce significativamente la complejidad de la electrónica y deja el trabajo "pesado" al compilador.

Otro enfoque ha sido el de crear *hardware* o unidades funcionales de propósito

[4] *Complex Instruction Set Computer*, computadora con juego de instrucciones complejo.

[5] Una traducción literal sería "tuberías", aunque teniendo en cuenta la función de un *pipeline* es más correcto identificarla como "línea de producción". Con esta técnica, cada unidad funcional comienza a trabajar en la siguiente instrucción tan pronto como ha realizado su trabajo y entregado la instrucción actual a la unidad funcional siguiente, a diferencia de la técnica tradicional donde cada unidad funcional realiza su trabajo y se mantiene inactiva hasta que se introduce la siguiente instrucción al procesador.

[6] *Very Long Instruction Word*, Palabra de instrucciones muy larga.

específico. Algunos de los primeros ejemplos fueron los procesadores vectoriales, hechos para operar simultáneamente en vectores y que se emplean primordialmente para cálculos físicos y matemáticos. Originalmente empleados en supercomputadoras como las Cray, en la actualidad se encuentran unidades de proceso vectorial en procesadores comerciales como el PowerPC G4, y versiones reducidas para cálculos SIMD (*Single Instruction on Multiple Data*) en casi cualquier procesador comercial actual (las extensiones MMX de Intel y 3DNow de AMD están basadas en tecnología de proceso vectorial).

1.3 Sistemas multiprocesador

El concepto de trabajo en equipo, el dividir una tarea entre varias unidades de ejecución para que pueda completarse más rápidamente, fue naturalmente llevado al mundo de las computadoras prácticamente desde sus inicios. David Slotnick, quien fuera colaborador de Von Neumann, le hizo la propuesta de una máquina que contara con varias unidades de procesamiento central; sin embargo, al sentir de Von Neumann, la tecnología de la época no permitía la realización de semejante proyecto. Aun así, Slotnick continuó con sus ideas, lo que eventualmente daría origen a la ILLIAC IV (1964), considerada una de las primeras computadoras masivamente paralelas de la historia. En la actualidad, las computadoras más rápidas del mundo son las máquinas masivamente paralelas.

El cómputo paralelo ofrece una serie de ventajas que lo hacen particularmente atractivo para los requerimientos de capacidad de cómputo, en particular los de la comunidad científica. Una de estas ventajas es económica. El uso de componentes comunmente disponibles, en grandes cantidades, permite ofrecer mayor rendimiento, a un precio menor que el de máquinas con procesadores especialmente diseñados (como por ejemplo las máquinas de procesadores vectoriales y de propósito específico). Adicionalmente, las computadoras paralelas son inherentemente escalables, permitiendo actualizarlas para adecuarlas a una necesidad creciente. Las arquitecturas "tradicionales" se actualizan haciendo los procesadores existentes obsoletos por la introducción de nueva tecnología a un costo posiblemente elevado. Por otro lado, una arquitectura paralela se puede actualizar en términos de rendimiento simplemente agregando más procesadores.

En ocasiones se menciona también la limitante física; existen factores que limitan la velocidad máxima de un procesador, independientemente del factor económico. Barreras físicas infranqueables, tales como la velocidad de la luz, efectos cuánticos al reducir el tamaño de los elementos de los procesadores, y problemas causados por fenómenos eléctricos a pequeñas escalas, restringen la capacidad máxima de un sistema uniprocesador, dejando la opción obvia de colocar muchos procesadores para realizar cálculos cooperativamente.

Como toda nueva arquitectura, las máquinas paralelas poseen características, y plantean ventajas y desventajas, que obligan a considerar cuidadosamente su utilización. También, dentro del mundo de las máquinas paralelas, existen dos enfoques con distintas características: la arquitectura SMP y la arquitectura MPP.

1.3.1 Arquitectura SMP

El enfoque más sencillo para una máquina con múltiples procesadores es el esquema SMP (*Symmetrical Multiprocessing*, multiproceso simétrico). Una arquitectura SMP es básicamente una expansión de una arquitectura tradicional pero con la adición de varios procesadores que comparten todos los demás recursos del sistema (memoria principal, almacenamiento secundario, periféricos de entrada y salida). En esta arquitectura no se establece distinción entre los procesadores; todos son jerárquicamente iguales y pueden ejecutar tareas indistintamente. De esta característica viene el nombre de "simétrica". Un diagrama de una arquitectura SMP genérica se muestra en la figura (1.1).

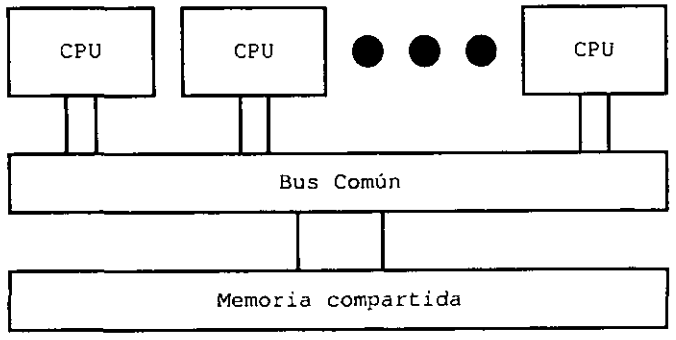

Figura 1.1: Arquitectura SMP

En general una arquitectura SMP tiene un equivalente en uniprocesador, y naturalmente un sistema SMP puede ejecutar simultáneamente varios programas o aplicaciones, que normalmente podrían ejecutarse en el sistema uniprocesador, de manera independiente. Sin embargo, para el uso de aplicaciones que aprovechen los múltiples procesadores para expeditar la realización de cálculos, nos interesa que dichos procesos no sean totalmente independientes, buscando entonces que cuenten con manera de comunicarse para distribuirse información, compartir y consolidar resultados.

Ya que un sistema SMP los procesadores comparten todos los periféricos y recursos, el esquema más obvio para comunicarse en una arquitectura SMP es el uso de memoria compartida. Como el nombre lo indica, en este esquema los procesadores tienen acceso a un espacio de direcciones común; esto puede ser todo el espacio de direcciones o únicamente un área designada para memoria compartida, permitiendo a cada proceso contar con un área exclusiva para sus requerimientos.

La memoria compartida es un esquema conceptualmente simple de utilizar. Sin embargo presenta ciertas limitaciones. Una de ellas, ya que se puede tener a dos o más procesadores manipulando la misma área de memoria, es que se puede caer en inconsistencias donde un procesador espera un dato que ha sido modificado por otro. Esto también puede provocar condiciones de competencia ("*race conditions*") y atoramientos ("*deadlocks*"), que son problemas clásicos de la sincronización de procesos, pero que no pueden dejar de tomarse en cuenta en una arquitectura SMP. Estas condiciones pueden resolverse utilizando mecanismos de sincronización de procesos, como semáforos, monitores y secciones críticas.

La limitación más importante de la arquitectura SMP, en términos del rendimiento máximo que puede alcanzarse, es la posibilidad de saturación de los buses del sistema. Ya que todos los procesadores tienen acceso al mismo bus de memoria, y en general a todos los periféricos que se encuentran comunicados comúnmente por buses, conforme se incrementa el número de procesadores se incrementa también el tráfico en dichos buses. Esto causa una saturación que finalmente termina por negar el incremento de rendimiento obtenido teniendo varios procesadores. Por esta razón una arquitectura SMP difícilmente puede escalar más allá de algunas decenas de procesadores.

1.3.2 Arquitectura MPP

Ya que el problema es el hecho de contar con memoria compartida y sus limitaciones, se propuso un esquema de una máquina paralela que consta de varias unidades de procesamiento básicamente independientes. En efecto cada una de estas unidades, conocida como "nodo", es prácticamente una computadora en sí misma, contando con su propio procesador, memoria no compartida, y que se comunica con las demás unidades de procesamiento a través de un canal provisto exclusivamente para este propósito. Este tipo de máquinas se conocen como computadoras masivamente paralelas o máquinas MPP (*Massively Parallel Processing*, procesamiento masivamente paralelo). Un diagrama de una arquitectura MPP genérica se muestra en la figura (1.2).

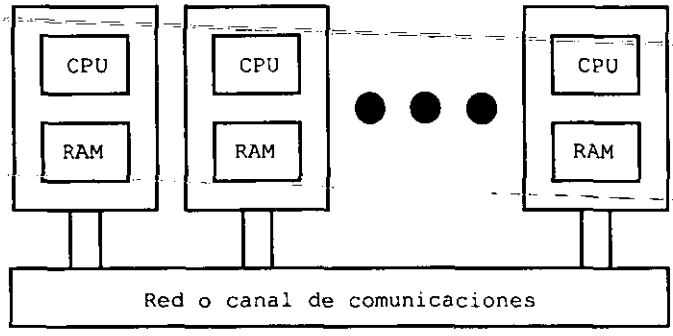

Figura 1.2: Arquitectura MPP

Una máquina MPP presenta una serie de consideraciones importantes derivadas de su arquitectura, que se deben tomar en cuenta al escribir programas que pretendan aprovechar su naturaleza multiprocesador. Obviamente la característica más importante es el hecho de que, en cada nodo, cada procesador opera básicamente como una computadora independiente, ejecutando su propio código independiente de los demás procesadores, y teniendo un área de memoria con datos también independientes.

Desde luego, para que esta organización redunde en un mayor desempeño, se requiere colaboración entre los nodos. Como se mencionó, una máquina MPP debe contar con un canal que permita a los nodos comunicarse entre sí, a fin de inter-

cambiar datos y coordinar sus operaciones. Ya que el objetivo principal de una máquina MPP es obtener alto rendimiento, se busca que este canal de comunicaciones sea lo *más eficiente posible, en términos tanto de ancho de banda como de* tiempo de latencia. En la mayoría de los casos este canal será un bus propietario, diseñado por el fabricante del equipo MPP.

Para tener acceso a información fuera de su propia área de memoria, los nodos se comunican entre sí, regularmente empleando un esquema de paso de mensajes. Esto resuelve el problema de saturación del bus de comunicaciones, pues éste sólo se emplea cuando se está realizando comunicación entre los nodos. De esta manera se tiene una arquitectura que puede escalarse a varios cientos o miles de procesadores (las máquinas MPP más grandes en la actualidad tienen alrededor de 10 mil procesadores).

Sin embargo el tener varias secciones de memoria independientes complica la programación en este tipo de arquitecturas. En una arquitectura MPP la distribución de trabajo entre los nodos es una consideración vital al diseñar cualquier aplicación. Se debe tomar en cuenta la sincronización de datos entre los nodos, y en toda comunicación entre ellos debe realizarse explícitamente por medio de llamadas al mecanismo de paso de mensajes.

1.3.3 Consideraciones de utilización para máquinas paralelas

Como se mencionó anteriormente, el incremento en capacidad de cómputo que ofrecen las máquinas paralelas conlleva ciertas restricciones y consideraciones que no pueden dejar de tenerse en cuenta. El contar con una máquina paralela de cualquier especie no es una panacea, ni redundará en un mayor desempeño para las aplicaciones, a menos que se tome en cuenta la naturaleza de su arquitectura y el *software* que se piensa ejecutar en ella se adapte o diseñe específicamente teniendo en mente las ventajas y limitaciones del esquema paralelo.

Memoria compartida

El enfoque paralelo más sencillo a nivel programación es el SMP. Ya que los procesadores comparten el mismo espacio de memoria, la programación es muy similar a la que se efectuaría en un sistema uniprocesador que soporte multiprogramación

o *multitasking*. Una máquina SMP se puede programar empleando técnicas conocidas, como bifurcación (*forking*) de procesos e hilos (*threads*). Desde luego, ya que es posible que dos procesos ejecutándose en distintos procesadores intenten acceder a la misma área de memoria simultáneamente, es muy importante planear los accesos a memoria compartida, identificando las secciones críticas en cada programa, y coordinando la entrada a dichas secciones con mecanismos de control tradicionales, como semáforos y monitores.

La relativa facilidad de programación para un esquema SMP tiene como desventaja la baja escalabilidad de esta arquitectura. Es decir, a cambio de contar con un paradigma de programación relativamente familiar y conceptualmente sencillo, un sistema SMP, como ya se ha mencionado, tendrá problemas para escalar más allá de cierto número de procesadores.

Paso de mensajes

El uso de arquitecturas MPP permite como principal ventaja una mayor escalabilidad-de-rendimiento por medio de la expansión del número de nodos. Sin embargo, debido a la arquitectura de un equipo MPP, se requiere un paradigma de programación diferente, que permita a los programas en cada nodo ejecutarse independientemente cuando lo necesiten, y explotar las facilidades de comunicación que son básicas para la arquitectura MPP cuando el diseño de la aplicación así lo requiera.

El esquema más utilizado en sistemas masivamente paralelos, o en general sistemas que constan de nodos independientes comunicados a través de algún medio, es el conocido como paso de mensajes.

Bajo el esquema de paso de mensajes, cada proceso se ejecuta independientemente, y únicamente se comunicará con otros procesos cuando el programa contenga instrucciones para hacerlo explícitamente. Esta comunicación se realiza a través de "mensajes" que contienen la información que requieran intercambiar los procesos. Un sistema de paso de mensajes proporciona funciones básicas, o "primitivas", para envío y recepción de mensajes. Es responsabilidad del programador el crear los mensajes con la información relevante, enviarlos a los procesos que requieren hacer uso de esta información, y asegurarse de que estos procesos empleen dicha información adecuadamente.

De estas características básicas se observa que el esquema de paso de mensajes

se presta a las características de una máquina masivamente paralela; cada proceso se ejecutará, normalmente, en un nodo, con un solo procesador y área de memoria; y la comunicación entre procesos se realiza únicamente cuando estos así lo requieran, de forma que el bus de comunicaciones se emplea únicamente cuando es necesario.

A cambio de esta mayor eficiencia en el uso del bus de comunicaciones, que en sistemas MPP es el elemento que más se presta a ser un "cuello de botella", el esquema de paso de mensajes es más complicado de programar que un esquema de memoria compartida. Una aplicación hecha con paso de mensajes, que se compone de varios procesos ejecutándose independientemente, debe realizarse teniendo en cuenta algunos factores inherentes a la arquitectura MPP. Todo intercambio de información entre procesos debe realizarse explícitamente y planearse cuidadosamente, teniendo en cuenta qué procesos tendrán cierta información, y qué otros procesos pueden requerirla, a fin de realizar el intercambio de esta información.

La gran mayoría de los problemas requieren, además de intercambio de datos entre los procesos, sincronización de los mismos, en casos en los cuales algún proceso requiere que otros terminen sus tareas antes de poder continuar. Al nivel más básico, el esquema de paso de mensajes no proporciona primitivas para estas operaciones, de modo que el programador tiene que implementarlas utilizando mensajes, esto incrementa la complejidad de la aplicación.

Finalmente, si se tiene en mente que la ventaja de una máquina MPP es su escalabilidad, se debe diseñar la aplicación teniendo en cuenta el aprovechamiento de un crecimiento en cuando al número de elementos de procesamiento en el equipo. Si no se planean cuidadosamente las comunicaciones entre procesos, es posible que la aplicación sature el bus de comunicaciones si se incrementa la cantidad de nodos o elementos de procesamiento.

Consideraciones generales

Tanto el paso de mensajes como la memoria compartida son paradigmas de comunicación interprocesos que en sí no dependen estrictamente de soporte a nivel lenguaje de programación, de alguna biblioteca de funciones o de algún soporte específico en el *hardware*. Es decir, es factible, en una máquina SMP, utilizar paso de mensajes para comunicarse entre procesos, ignorando las facilidades de memoria

compartida; también es posible implementar un esquema de memoria compartida en una máquina MPP, utilizando el bus de comunicaciones para simular la presencia de un área de memoria compartida. Sin embargo, ya que la meta de una máquina paralela de cualquier especie es tener mayor rendimiento, en general se busca emplear el esquema de programación adecuado a la arquitectura con que contamos, ya que esto permite aprovechar al máximo las facilidades que proporciona el equipo, así como emplear un esquema de programación que obliga a tener en cuenta las limitaciones de la arquitectura al momento de diseñar la aplicación.

Independientemente de la arquitectura de la máquina paralela con la que se cuente, siempre se debe tener en mente que el buen diseño es esencial para que una aplicación paralela realmente presente una mejora de rendimiento sobre un equivalente en uniprocesador. La dificultad de la resolución cooperativa de problemas, que es la idea central detrás del cómputo paralelo, es lo que hace de éste una rama entera de las ciencias de la computación, con un grado de complejidad y muchas sutilezas que deben tenerse en cuenta al considerar el empleo de este paradigma computacional.

En general se asume que un número N de tareas terminarán el trabajo N veces más rápido. Sin embargo esto regularmente no se cumple, y si bien inicialmente la lógica parece correcta, la afirmación siguiente muestra lo descabellado que esto puede resultar: "si un hombre puede cavar un agujero en un minuto, sesenta hombres pueden hacerlo en un segundo". Esta frase clásica nos hace detenernos a pensar y da un breve vistazo sobre la complejidad real de realizar tareas cooperativamente.

La Ley de Amdahl

Eugene Amdahl analizó este problema y en 1967 propuso lo que se conoce como la Ley de Amdahl [8], que indica la mejora de rendimiento que se puede esperar incrementando los elementos de procesamiento. La Ley de Amdahl toma en cuenta la parte "secuencial" del proceso, es decir, aquella que independientemente de cuántos elementos de procesamiento tengamos, puede ser realizada por uno solo de ellos; y el resto del cálculo no podrá continuar hasta que se haya completado la parte secuencial.

La Ley de Amdahl propone normalizar el tiempo que toma realizar la opera-

ción en un solo procesador al valor de 1. La fracción del cálculo que sólo se puede realizar secuencialmente será F, entonces la fracción paralelizable es $(1 - F)$. Con estos datos, el incremento de velocidad máximo que puede obtenerse con P elementos de procesamiento está dado por la fórmula:

$$\frac{1}{F + \frac{(1-F)}{P}}$$

Como un ejemplo, si nuestra aplicación no tiene sección secuencial (es decir, $F = 0$ y $(1 - F) = 1$), entonces el incremento de velocidad máximo estará dado exactamente por el número de elementos de procesamiento:

$$\frac{1}{\frac{1}{P}} = P$$

Por otro lado, si el 50% del código es secuencial (es decir, $F = 0.5$ y $(1 - F) = 0.5$), la ecuación queda:

$$\frac{1}{0.5 + \frac{(0.5)}{P}} = \frac{P}{0.5P} + \frac{0.5}{P}$$

Suponiendo un número infinito de procesadores, esta ecuación da como resultado 2. Si el 50% del código es secuencial, por más procesadores que se agreguen el rendimiento nunca será más de 2 veces mayor que una implementación uniprocesador.

En general el incremento de velocidad máximo si el número de procesadores tiende a infinito será de $1/F$. Si tenemos 10% de código secuencial, aumentar el número de procesadores no llevará un incremento de rendimiento mayor a 10 $(1/0.1)$. Similarmente, 90% de código secuencial significa que el rendimiento no podrá crecer más allá de un factor de 1.111 $(1/0.9)$.

Realizando una gráfica de número de procesadores contra incremento de rendimiento máximo, se observa que la gráfica es logarítmica, aproximándose al valor máximo determinado anteriormente, sin llegar a alcanzarlo nunca. Ya se determinó el factor de incremento máximo, sin embargo estas gráficas pueden ser una herramienta útil para decidir cuantos elementos de procesamiento se deben dedicar al problema. Dentro de los límites del incremento máximo ya mencionado, para cada valor de F la curva es diferente. Para algunos valores de F la curva se aproxima a

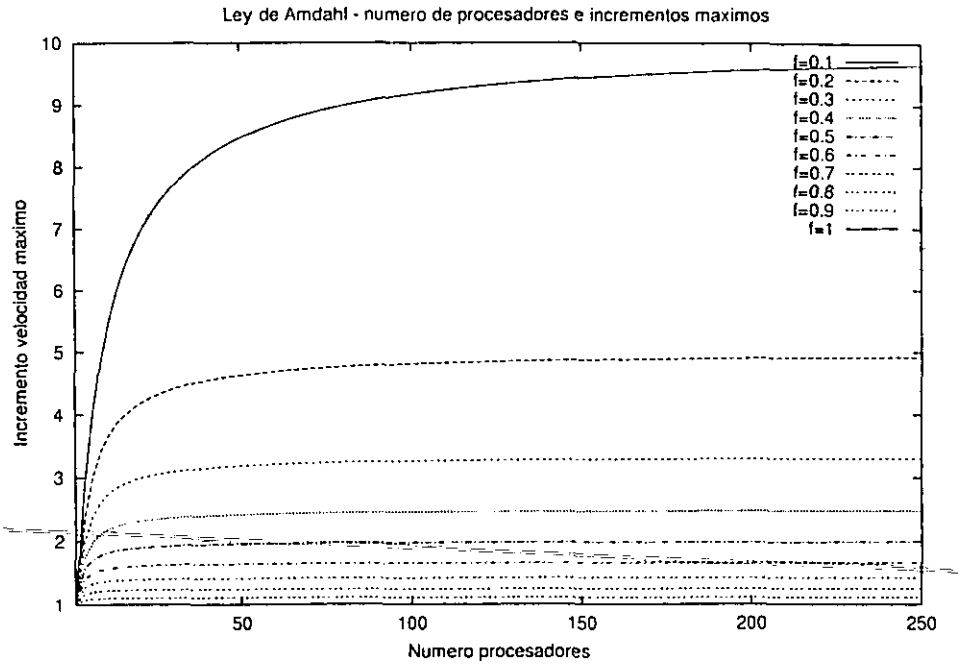

Figura 1.3: Gráfica Ley de Amdahl

la asíntota muy rápidamente según el número de elementos de procesamiento, en este caso puede no ser muy útil agregar más elementos. Para otros valores de F, la curva es menos pronunciada, acercándose más lentamente a la asíntota, y el incremento de rendimiento con más procesadores puede ser substancial hasta el límite propuesto por la Ley de Amdahl.

La figura (1.3) muestra gráficas de la fórmula de la Ley de Amdahl, para algunos valores de F entre 0.1 y 1, mostrando la relación entre el número de procesadores y el rendimiento que se puede obtener.

Así pues, la Ley de Amdahl deja de manifiesto un concepto esencial para la planeación y utilización de cualquier tipo de máquina paralela: el incremento en velocidad que podemos obtener está limitado por el algoritmo que emplee nuestra aplicación, y no por el número de procesadores. Como un ejemplo, de la gráfica (1.3) podemos observar que un algoritmo con 50% de código secuencial no podrá tener un incremento de velocidad mayor a 2 independientemente del número de procesadores; si se logra reducir el código secuencial a 40%, se obtiene un incre-

mento de velocidad de 2 utilizando únicamente 6 elementos de procesamiento. Si el problema lo permite, Se debe buscar implementar algoritmos que tengan la menor cantidad de código secuencial, pues como se aprecia en las curvas, este factor tiene mucha más importancia que el número de procesadores a utilizar.

1.3.4 Estándares de programación en máquinas paralelas

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

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

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

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

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

implementaciones no óptimas. Por ejemplo, un lenguaje que cuente con paralelismo a nivel datos depende para su óptimo rendimiento de encontrarse en un equipo SMP (memoria compartida) o MPP con un bus de comunicaciones muy amplio. Quizá el lenguaje ampliado más conocido es el HPF (High Performance Fortran), aunque también en esta categoría se cuentan Dataparallel-C y Compositional C++.

Finalmente, el enfoque más socorrido es el de emplear un lenguaje existente (donde C y Fortran son los más utilizados) y recurrir a funciones proporcionadas por alguna biblioteca para cómputo paralelo. La popularidad de este enfoque se debe a que únicamente se requiere que el programador se familiarice con algunas funciones nuevas, sin dejar atrás su proficiencia con el lenguaje que se utiliza. Además, ya que las funciones de biblioteca son específicas para la arquitectura paralela que se está trabajando, en la mayoría de los casos se asegura que el rendimiento que se puede obtener de las funciones paralelas de la arquitectura será el máximo posible.

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

1.3.5 Bibliotecas de paso de mensajes

Inicialmente la tecnología para producir máquinas paralelas era compleja y regularmente cada una de las compañías que producía esta clase de equipos tomaba un enfoque propio y diferente. Ya que el desarrollo en general era cerrado y no existía comunicación y cooperación entre las compañías, cada una desarrolló herramientas para aprovechar las capacidades de sus equipos. Dichas herramientas eran incompatibles entre sí, si bien algunas de ellas, en particular las correspondientes a

equipos MPP que se programaban empleando alguna variante del esquema de paso de mensajes, realizaban funciones muy similares.

Dada su aplicación en el cómputo científico, los principales usuarios de equipos paralelos son las instituciones académicas y laboratorios gubernamentales, particularmente en Estados Unidos. Se trata de comunidades de usuarios con una dinámica altamente cooperativa, y que en ocasiones cuentan con equipos producidos por diversos fabricantes. En particular, las decisiones de compra de equipo obedecen a criterios que pueden variar entre instituciones, por razones de presupuesto, de requerimientos tecnológicos y la tecnología disponible.

Por estas características, las principales comunidades de usuarios de máquinas paralelas se beneficiarían de tener métodos y mecanismos estandarizados para describir e implementar algoritmos paralelos, en cualquier equipo que cuente con estas capacidades.

1.3.6 PVM

Uno de los primeros esfuerzos para crear una biblioteca de paso de mensajes con especificación abierta fue PVM (*Parallel Virtual Machine*). Si bien, como se verá en un momento, el desarrollo de PVM comenzó en otro contexto, su naturaleza abierta lo convirtió en un estándar *de facto* en paso de mensajes.

PVM es "un sistema de programación con paso de mensajes portable, diseñado para enlazar varios equipos para formar una máquina paralela virtual, que es un recurso de cómputo único y administrable".[10]

Historia

El desarrollo de PVM comenzó en 1989, como parte de un proyecto de investigación sobre cómputo distribuido en ambientes heterogéneos, en el laboratorio nacional Oak Ridge (ORNL), en Estados Unidos. Como un subproducto de este proyecto, se desarrolló el concepto de una "máquina paralela virtual" y se desarrollaron herramientas de programación internas para realizar experimentos sobre estos conceptos.

En 1991, con la idea de permitir el uso de estos desarrollos por entidades externas al ORNL, se reescribió el sistema PVM, apareciendo así PVM 2.0. Con esta

versión, el uso de PVM se extendió rápidamente, particularmente entre científicos que se dieron cuenta de la utilidad de este *software* para realizar investigaciones sobre cómputo.

En 1993 se liberó PVM 3.0, tratándose de un rediseño total del software para responder a las necesidades de los ya varios miles de usuarios de PVM.

En la actualidad PVM es uno de los estándares de paso de mensajes más utilizados, y su desarrollo continúa, aún bajo el auspicio de los proyectos de cómputo distribuido en el ORNL.

Diseño

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

Si bien PVM fue inicialmente diseñado para un ambiente distribuido y heterogéneo, que se puede visualizar como una serie de equipos o nodos interconectados por una red local, también está disponible para equipos MPP comerciales como los Intel iPSC y Paragon, el CM-5 de Thinking Machines, algunos equipos con arquitectura de memoria compartida de Sequent, IBM, SGI, DEC y Sun, e incluso supercomputadoras como la Cray T-3D (un equipo masivamente paralelo con memoria distribuida). Esto tiene sentido si se recuerda que anteriormente (sección 1.3.3) se mencionó que es posible emplear un paradigma de paso de mensajes en una arquitectura de memoria compartida.

Más aún, si bien la versión pública de PVM se puede ejecutar en todas estas arquitecturas, los fabricantes pueden implementar la API [7] de PVM sobre las funciones de *hardware* específicas de sus equipos, aprovechando sus capacidades y obteniendo mayor rendimiento. Esto se da particularmente en equipos inherentemente multiprocesador, que pueden ya contar con facilidades para intercambio de

[7] *Application Program Interface*, interfaz para programas de aplicación, el juego de funciones externas que una biblioteca presenta a los programadores para posibilitar hacer uso de su funcionalidad.

mensajes de muy alto rendimiento.

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

El diseño de PVM refleja ciertos principios, obedeciendo a su desarrollo para una máquina paralela virtual que puede estar compuesta por nodos con distintas arquitecturas y el uso del paradigma de paso de mensajes.

PVM proporciona facilidades para crear la máquina paralela virtual a partir de uno o más *hosts* disponibles. El usuario tiene la posibilidad de especificar qué *hosts* formarán parte de su máquina virtual, y esta configuración puede modificarse al estarse ejecutando el programa; de hecho, un programa para PVM puede por sí solo agregar y eliminar hosts de la máquina virtual.

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

PVM va más allá de la portabilidad (capacidad de compilar el mismo programa sin cambios en varias arquitecturas distintas) e implementa el concepto de heterogeneidad, es decir, en una aplicación PVM pueden interactuar programas ejecutándose en nodos con arquitecturas diferentes. PVM logra esto empleando tipos de datos opacos, y funciones que convierten los tipos de datos específicos de cada nodo al equivalente opaco para su empleo en mensajes que se distribuyen entre los nodos.

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

Implementación

A nivel funcional, PVM está implementado bajo una arquitectura cliente-servidor. Cada nodo ejecuta un demonio[8] (*pvmd*) que es el servidor, y se encarga de arbitrar los recursos del nodo y comunicarse con el resto de los nodos para formar la máquina virtual.

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

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

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

PVM proporciona los siguientes tipos de funciones:

- Paso de mensajes (envío y recepción)
- Empacado y desempacado de mensajes
- Funciones de agrupación de procesos
- Control de procesos (iniciar, detener procesos)
- Obtención de información
- Control de opciones

[8] *daemon*, un programa que está en ejecución continua y existe para manejar peticiones de servicio periódicas recibidas por el equipo.

- Configuración dinámica de máquina virtual

1.3.7 MPI

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

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

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

Historia

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

En 1992, el Centro de Investigación en Cómputo Paralelo, ubicado en Williamsburg, Virginia, patrocinó un taller de estándares de paso de mensajes en ambientes de memoria distribuida. En este taller se discutieron las características básicas de una interfaz de paso de mensajes estandarizada, y se formó un grupo de trabajo para definir el estándar.

En la conferencia de Supercómputo, en 1993, se presentó una versión preliminar del estándar MPI, y se constituyó el MPI Forum como organismo encargado de supervisar la evolución del estándar. El MPI Forum es un foro cuya membresía está abierta a cualquiera interesado en el cómputo de alto rendimiento.

La especificación definitiva de MPI 1.0 se completó en 1994, y se ha revisado y actualizado constantemente. Actualmente la especificación más reciente es MPI-2.

Diseño

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

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

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

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

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

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

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

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

- Organización de procesos: manipulación de grupos y rangos
- Paso de mensajes
  - Envío y recepción
  - *Buffers* de mensajes
  - Mensajes bloqueantes y no bloqueantes
- Comunicaciones entre procesos

  - Organización por medio de comunicadores
  - Comunicación entre grupos no relacionados
  - Comunicaciones de una vía (Remote Memory Access): *put*, *get*
  - Mecanismos de sincronizacion: *fence*, *lock*

- Operaciones colectivas
  - Reducción
  - Dispersión/recolección (*scatter/gather*)

- Manipulación de archivos optimizada para máquinas paralelas

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

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

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

Implementación

Como se vio anteriormente, no existe **la** implementación de MPI; existen quizá varias decenas de implementaciones que se adhieren a la especificación MPI pero que pueden ser operativamente diferentes.

Para fines de este proyecto, se empleará la implementación MPICH de MPI. Los criterios tomados en cuenta para seleccionar esta implementación se describirán más adelante; sin embargo se considera adecuado describir a MPICH como un ejemplo de implementación de MPI. Aun así, téngase en mente que existen muchas implementaciones de MPI. Algunas de estas son MPI-BIP, LAM/MPI, W32MPI/p4, ScaMPI, y algunas implementaciones particulares de los fabricantes, como el MPI de Cray, IBM MPI (particularmente para sus sistemas *Scallable Parallel*), Digital MPI en equipos DEC/Compaq, MPI/Pro (una implementación comercial disponible para varias plataformas) y HP MPI (en particular para las supercomputadoras HP/Convex Exemplar).

MPICH es una de las implementaciones más robustas de MPI, habiendo evolucionado junto con el estándar, y estando disponible para una gran cantidad de plataformas.

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

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

A nivel implementación MPICH proporciona una biblioteca de funciones que implementan la API de MPI. MPICH está diseñado por capas, permitiendo gran portabilidad sin sacrificar el rendimiento. A niveles altos MPICH implementa las funciones de MPI, comunicándose con la capa inferior por medio de una interfaz

conocida como "interfaz de canal". La capa inferior implementa, de manera específica para cada plataforma, funciones para intercambiar información entre procesos, según el canal de comunicaciones que se tenga (desde memoria compartida hasta una red local)

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

1.3.8 Otros estándares de programación para máquinas paralelas

PVM y MPI son los estándares más utilizados en programación de máquinas paralelas, pues implementan el paradigma de paso de mensajes, que es el más escalable y con el que se tiene mayor experiencia, y ambos están ampliamente disponibles para una variedad de plataformas. Sin embargo no son los únicos disponibles. Existen una serie de definiciones de interfaces de paso de mensajes que no son ya muy utilizadas, habiendo sido superadas por PVM, y, en particular, por MPI; y un estándar para programación paralela, OpenMP, que en ocasiones se menciona como alternativa a PVM y MPI.

OpenMP

OpenMP es una especificación para una serie de directivas de compilador, funciones de biblioteca y variables de ambiente que pueden ser utilizadas para especificar paralelismo en memoria compartida en C/C++ y Fortran. OpenMP es un esfuerzo similar a MPI en su objetivo, que es el de crear un estándar para programación con memoria compartida, unificando las interfaces existentes, en las cuales se tienen semánticas similares pero implementaciones no compatibles. Existe un organismo, el OpenMP Architecture Review Board, que se encarga de mantener y revisar la especificación OpenMP; los fabricantes de equipos con memoria compartida pue-

den implementar la especificación OpenMP y proporcionarla a los usuarios para que utilicen sus equipos.

Como una implementación para memoria compartida, OpenMP no compite directamente con PVM y MPI. OpenMP es utilizable en equipos con memoria compartida, y aún empleando técnicas híbridas donde se tienen nodos de procesamiento con memoria distribuida y la memoria se comparte a través de canales de comunicación de alta velocidad especializados, tanto la arquitectura como el paradigma de programación para memoria compartida no alcanzan los niveles de escalabilidad de una arquitectura masivamente paralela. Por lo tanto el uso de OpenMP está hasta cierto punto restringido a equipos de menor capacidad, aunque si dichos equipos cuentan con memoria compartida, en ocasiones es más sencillo utilizar OpenMP para la programación. Los equipos MPP más complejos y poderosos siguen siendo el dominio casi exclusivo de PVM y MPI.

Especificaciones de Paso de Mensajes

Como se mencionó anteriormente, en un principio existían una gran cantidad de especificaciones e implementaciones de paso de mensajes, incompatibles entre sí y en general no muy portables. Estas se consideran predecesoras de MPI, y equivalentes a PVM, si bien la portabilidad y amplitud de PVM lo hizo el estándar dominante antes de la creación de MPI.

Algunos de estos son p4, Chameleon, Zipcode, Express, PARMACS (Parallel Macros), Chimp y PICL. Se tomaron elementos de diseño de algunos de éstos para la definición de MPI; y algunos de ellos, en particular Chameleon y p4, forman parte de la implementación portable de MPICH. En la actualidad estas especificaciones han caído en desuso.

1.4 *Clusters*: una clase de máquina paralela

En la sección (1.3.2) se describieron las características básicas que debe cumplir una máquina masivamente paralela. En resumen, una máquina MPP consta de nodos que contienen una unidad de procesamiento y memoria, y un canal de comunicaciones que permite intercambio de información entre nodos.

1.4.1 Algunas clases de máquinas paralelas

Existen varios diseños de *hardware* que corresponden a este paradigma. Las máquinas MPP más poderosas tradicionalmente emplean un bus de comunicaciones especializado y construido *ad hoc* para el sistema. Los elementos de procesamiento (conjuntos de CPU y memoria) se conectan directamente a este bus de comunicaciones.

La ILLIAC IV, considerada la primera máquina MPP de la historia, empleaba este mecanismo para comunicar sus 64 elementos de procesamiento, consistentes de un CPU trabajando a 13 MHz y 16 KB de memoria (para un total de 1 MB entre todos los elementos de procesamiento). La ILLIAC IV implementaba una organización lineal donde cada elemento de procesamiento tenía conexión a sus vecinos más cercanos.[21]

Este esquema sigue siendo muy utilizado; se emplea en equipos desde MPP comerciales como la Cray T3 hasta equipos únicos empleados en tareas de investigación como la ASCI White.

Este esquema proporciona el máximo rendimiento, ya que contando con control total sobre la organización y configuración física del bus de comunicaciones; se puede optimizar su desempeño, tanto en términos de latencia como de ancho de banda; de hecho, en equipos MPP modernos, el bus de comunicaciones es el componente más complejo y donde se pone el mayor énfasis. La desventaja es que la configuración del equipo puede llegar a ser inflexible; ya que el bus de comunicaciones no puede modificarse fácilmente, restringe el número máximo de elementos de procesamiento que pueden instalarse en el equipo, y aún si no se instalan todos los elementos posibles, se debe tener el bus de comunicaciones disponible para ellos. Además, el nivel de especialización del bus de comunicaciones hace que sea un componente sumamente costoso; el costo de un equipo MPP comercial puede alcanzar decenas de millones de dólares, y los equipos más poderosos, desarrollados para proyectos avanzados como la iniciativa ASCI, tienen un costo aún mayor.

Un escalón abajo se encuentran los sistemas que pueden operar independientemente, pero también se pueden unir para formar un equipo más poderoso, por medio de un canal de comunicaciones propietario, pero de propósito múltiple. Un ejemplo de este tipo de diseño está en los equipos Convex Exemplar (en la actualidad Convex es parte de Hewlett Packard). Convex denomina a su tecnología

SPP (*Scallable Parallel Processor*). Un solo equipo Exemplar consta de 1 a 8 procesadores. Si se desea escalar, se puede adquirir otro equipo Exemplar e interconectarlos por medio de un bus que Convex llama **ADN-ii**. De esta manera se obtiene una máquina más poderosa donde los procesadores pueden comunicarse a través del bus. En el caso particular de las Exemplar, se trata de un esquema híbrido pues cada equipo por sí solo se puede considerar como un sistema SMP con memoria compartida, y el conjunto se puede visualizar como un equipo MPP de memoria distribuida. Sin embargo, el concepto aplica en general a todo equipo que permite la conexión, por medio de un bus propietario y de alta velocidad, de cierta cantidad de equipos independientes para componer un equipo MPP de buena capacidad. Este enfoque es más económico que el anterior, aunque también, y dada la naturaleza del bus de comunicaciones, no se presta a un rendimiento tan elevado.

1.4.2 Los *clusters*

Los equipos MPP especializados suelen ser grandes y costosos. Como se trata de productos enfocados a un mercado sumamente especializado, es poco probable que esta tendencia se modifique, ya que como todo producto de nicho, los fabricantes de estos equipos deben mantener un costo elevado para contar con un margen de ganancia que permita justificar la venta de pocos equipos, así como financiar investigación y desarrollo para tecnologías futuras.

Fuera de estos mercados especializados, la tendencia en el ámbito de la computación es de avances espectaculares y reducciones de precio constantes. Gracias a este avance, y como es conocido ampliamente, las computadoras personales actuales, con precios que en ocasiones son menores a mil dólares, tienen más capacidad que las supercomputadoras que hace 20 años se consideraban las más poderosas del mundo, con un costo de varios millones de dólares.

El concepto del cómputo distribuido deriva directamente de la proliferación de computadoras relativamente potentes y de bajo costo, así como del advenimiento de la tecnología de redes de área local, que permite interconectar varios equipos independientes e intercambiar datos entre ellos.

Al hacerse más comunes estas tecnologías, y proliferar el tipo de instalaciones donde se tienen varias estaciones de trabajo comunicándose a través de una red de área local, rápidamente se cayó en cuenta que esta configuración corresponde, a

nivel básico, a la de un equipo de procesamiento paralelo.

De esta manera, y a mediados de la década de los 80, se comenzó a manejar el concepto de cómputo distribuido. Este concepto contempla la resolución de algún problema complejo, empleando varios equipos de cómputo independientes, que se comunican y cooperan para la resolución del problema, utilizando una red de área local que los interconecta. Este esquema resulta familiar pues es la estructura más básica de una máquina MPP, aunque las configuraciones empleadas en cómputo distribuido presentan una serie de diferencias, ventajas y desventajas frente a equipos MPP propiamente dichos.

Un grupo de computadoras interconectadas y que cooperan entre sí para realizar una tarea común, suele llamarse un *cluster* (racimo).

Conviene mencionar que el hecho de formar un *cluster* suele tener dos objetivos: el primero es lograr alta disponibilidad, y el segundo es tener alto rendimiento.

Un *cluster* de alta disponibilidad busca proporcionar un servicio de la manera más confiable posible. En esta configuración, uno o más equipos del *cluster* proporcionan el servicio, mientras que los demás funcionan como "respaldo". Normalmente efectúan "espejeo" (*mirroring*) de la información en los equipos de servicio, a fin de mantener una copia actualizada de dicha información. Si alguno de los equipos de servicio llega a fallar, el equipo de respaldo entra en su lugar, de esta manera logrando que el servicio no sea interrumpido.

El *cluster* de alto rendimiento busca la resolución de un problema, por medio de la cooperación entre los equipos que lo componen, en el menor tiempo posible. En este sentido es diferente al *cluster* de alta disponibilidad pues se busca que todos los equipos estén realizando alguna tarea componente de la solución al problema, y comunicándose con los demás nodos.

Los primeros proyectos para utilizar este tipo de configuración en cómputo de alto rendimiento se enfocaron a la explotación de recursos existentes, como estaciones de trabajo empleadas para diseño y cálculos de manera independiente, y las redes de área local que las interconectaban. Fue en estos primeros ambientes de cómputo distribuido heterogéneo que surgieron proyectos como PVM, que inicialmente fue producto de un proyecto de investigación sobre la utilización de esos recursos de cómputo no utilizados. Así pues, los primeros *clusters*, si bien ya pueden identificarse claramente como tales, se conocieron inicialmente como NOWs

(*Network Of Workstations*).

1.4.3 *Clusters* tipo *Beowulf*

Las computadoras personales tradicionalmente han estado un paso atrás de los equipos de alto nivel que entran en la categoría de estaciones de trabajo. Sin embargo, a consecuencia de la economía de mercado que impulsa el desarrollo tecnológico en el ámbito de las computadoras personales, éstas han evolucionado al grado que la tecnología disponible en la categoría de cómputo personal está casi a la par de las estaciones de trabajo, todo ello por un precio accesible, incluso al grado de poder dotar a cada persona de una organización con un equipo propio; la necesidad de las organizaciones de permitir colaboración y compartir información entre sus integrantes también ha incrementado la capacidad tecnológica y reducido el precio de la tecnología de redes locales. Fue, en particular, el incremento en rendimiento y reducción de costo de esta tecnología en redes locales la que permitió la creación, en 1994, de un nuevo tipo de *cluster*, denominado *Beowulf*.

Historia

En 1994, bajo el patrocinio del proyecto ESS (*Earth and Space Sciences*), un grupo de investigadores del CESDIS (*Center of Excellence in Space Data and Information Sciences*), que desarrolla proyectos para la NASA, construyeron un *cluster* consistente en 16 equipos con procesadores Intel DX4, interconectados por una red tipo Ethernet de canal múltiple. Esta máquina fue llamada *Beowulf*. En su honor, a los *clusters* de este tipo, y con las características que se mencionarán a continuación, se les conoce genéricamente como *clusters* tipo *Beowulf*, o, simplemente, *Beowulf*.

La motivación del proyecto que creó el primer *Beowulf* era el explorar la posibilidad de construir una plataforma de cómputo paralelo de alto rendimiento basándose en componentes comerciales comunes. Esto se hizo pensando en limitantes que existen en sistemas MPP comerciales. Estos sistemas, si bien en épocas recientes utilizan procesadores comercialmente disponibles, como el DEC Alpha, dependen en gran medida del bus de comunicaciones, que requiere microelectrónica especializada, y otros elementos que en ocasiones se construyen a propósito

para cada equipo. Un equipo con construcción muy especializada tiene un tiempo de vida limitado. Esto afecta al personal que trabaja en el mantenimiento de dicho equipo, así como a los usuarios del mismo, implicando un alto costo cuando el equipo, ya obsoleto, es reemplazado por uno más reciente. La utilización de tecnología más común en la construcción de equipos de alto rendimiento busca obtener una plataforma más genérica, que requiera menos cambios en el modelo y técnicas de programación, permitiendo una mayor continuidad en los proyectos que utilicen esta plataforma.

Beowulf fue un gran éxito, y los *clusters* de este tipo pronto se popularizaron dentro de la NASA, y más allá, convirtiéndose en una técnica extremadamente popular para obtener cómputo de alto rendimiento a bajo costo.

Actualmente existen infinidad de *Beowulfs* en toda clase de ambientes, desde instituciones de investigación hasta universidades, dependencias gubernamentales y empresas. Más allá del hecho de que la tecnología *Beowulf* pone el cómputo de alto rendimiento al alcance de las masas, cabe mencionar que el *Beowulf* más rápido del mundo, construido por IBM para la compañía Shell, alcanza velocidades pico de 1.037 TFLOPS, lo cual la coloca entre las 15 computadoras más rápidas del mundo. Este sistema, conocido como *Genesis Machine*, cuenta con 1038 procesadores, 512 GB de memoria total y 74 TB de almacenamiento secundario.

Características

Un *Beowulf* puede definirse como "una clase de computadora masivamente paralela de alto rendimiento construida primordialmente con componentes comerciales ampliamente disponibles". En realidad, como se vio anteriormente, esto consiste en un grupo de computadoras personales dedicadas a ejecutar en conjunto tareas de cómputo que requieren alto rendimiento.

Ya que las computadoras personales compatibles con IBM, basadas en procesadores Intel x86 y con alguna variante de red de área local Ethernet, son las más comunes en el mercado, la mayoría del desarrollo de *Beowulfs* se ha realizado bajo esta plataforma; sin embargo, la tecnología básica de un *Beowulf* puede emplearse bajo cualquier arquitectura de cómputo personal.

Un *Beowulf* consta de una serie de computadoras personales, o nodos. Cada una de estas computadoras es un sistema completo e independiente, contando con

su propio procesador, memoria y almacenamiento secundario. Estas computadoras se enlazan a través de alguna tecnología de red local disponible comercialmente. Gracias a su amplia disponibilidad, las redes tipo Ethernet, en sus tres variantes (Ethernet, Fast Ethernet y Gigabit Ethernet) son las más utilizadas, aunque también se pueden utilizar otro tipo de conexiones, como HIPPI (*HI-performance Parallel Port Interface*) y Myrinet (una tecnología de red de alta velocidad (2 Gbps), con conmutación y bajo tiempo de latencia (9 μs) creado por la empresa Myricom).

Cada uno de los nodos de un *Beowulf* funciona independientemente bajo algún sistema operativo, que tradicionalmente es alguna variante de Unix, y en la mayoría de los casos es el sistema operativo Linux.

Hay varios factores que determinan que Linux sea el sistema operativo más utilizado en un *Beowulf*. El desarrollo del primer *Beowulf* se hizo bajo el sistema Linux. Esto fue debido a la amplia disponibilidad de Linux, así como de su código fuente, que permitía a los investigadores del proyecto *Beowulf* realizar las modificaciones necesarias para alcanzar sus metas de rendimiento. De hecho, los controladores Ethernet que se utilizan, aún actualmente, bajo Linux, son derivados de los controladores creados por Donald Becker para el proyecto *Beowulf*, y Becker está aún muy involucrado con el desarrollo de controladores Ethernet de alto rendimiento, pues estos son básicos para el rendimiento en un *cluster*. Por otro lado, el desarrollo constante de Linux hace que este sistema suela tener un buen desempeño dentro de cada nodo, así como una gran estabilidad. Ya que el desarrollo normal de Linux sigue la evolución de las arquitecturas de cómputo personal más populares, no se requiere un esfuerzo adicional para mantener el sistema operativo al día con los avances tecnológicos. Finalmente, el utilizar un sistema tan difundido reduce la curva de aprendizaje de quienes desean crear un *Beowulf* por sí mismos.

El componente final para un *Beowulf* es el mecanismo que las aplicaciones utilizan para comunicación y cooperación entre nodos. Este vacío es llenado por implementaciones de los dos grandes estándares de comunicación por paso de mensajes, PVM y MPI. Una de las plataformas soportadas por PVM es Linux, empleando comunicación a través de TCP/IP. Por otro lado, varias implementaciones de MPI, entre ellas MPICH y LAM/MPI, soportan *Beowulfs* utilizando Linux como el sistema operativo. De esta manera, se aprovecha toda la experiencia previa de los

programadores utilizando paso de mensajes, y se asegura un gran nivel de compatibilidad, portabilidad e interoperabilidad de las aplicaciones paralelas que se desarrollen.

En general la organización de un *Beowulf* suena similar a la de un NOW. Existen, sin embargo, algunas características que diferencían claramente estos dos tipos de *clusters*. Los nodos de un *Beowulf* no realizan ninguna tarea que no sea relativa al *cluster*. También es necesario contar con una red dedicada a comunicación entre nodos del *cluster*. Estas dos características proporcionan una serie de ventajas. La primera es un mayor rendimiento ya que los nodos y la red no realizan tareas ajenas al cómputo del *cluster*. La segunda es una mayor predecibilidad del comportamiento tanto de los nodos como de la red; así se eliminan cargas impredecibles por procesos extraños en los nodos, y no se tienen problemas de latencia en la red pues en esta no existe más tráfico que el relativo a comunicaciones internodos. La restricción también redunda en una mayor facilidad de uso pues todo el trabajo se realiza sobre un nodo central, y deben existir mecanismos para distribución de programas y ejecución que sean transparentes al usuario. Adicionalmente se facilita el trabajo en cuanto a seguridad, ya que el nodo central actúa como "firewall" protegiendo a la red del *cluster* y reduciendo la necesidad de implementar políticas de seguridad en los nodos como tales.

Una característica importante de un *Beowulf* es que las actualizaciones de *hardware* más comunes, como son actualización de procesador, incremento de memoria, o mejora de velocidad de transferencia en la red, no cambian el modelo de programación utilizado. Por lo tanto, los usuarios de estos sistemas pueden contar con mejor compatibilidad con equipos futuros.

Dentro de la clasificación de equipos de cómputo paralelo, los *Beowulf* se pueden visualizar en un punto intermedio entre los equipos MPP propiamente dichos y las NOW. Un *Beowulf* comúnmente tiene menos procesadores que un equipo MPP; también, el mecanismo o bus de interconexión entre nodos es de menor rendimiento en un *Beowulf*; se tiene mayor tiempo de latencia y menor ancho de banda. Por otro lado, las NOW buscan utilizar la capacidad de cómputo no aprovechada en una red, mientras se busca que las características de la red no afecten la realización de cálculos. En una plataforma NOW se tienen consideraciones que no existen en un *Beowulf*, como son seguridad, balanceo de carga por trabajos extraños realiza-

dos en las estaciones, y latencia y ancho de banda variables en las comunicaciones entre nodos.

1.4.4 Implementación

¿Qué se necesita para tener un *Beowulf*? Como se ha mencionado, para un *Beowulf* se requieren los nodos como tales, así como una red local de interconexión; un sistema operativo en los nodos, que en la mayoría de los casos es Linux; y un método para que los programas aprovechen la naturaleza paralela del *Beowulf*.

Interesantemente, en la mayoría de los casos estos serán los únicos elementos necesarios. Desde el principio, el proyecto *Beowulf* ha buscado integrarse estrechamente con el desarrollo normal de Linux, así como interferir lo menos posible con una instalación de Linux tradicional.

Así, la mayoría del *software* requerido para construir un *Beowulf* se proporciona como una adición a alguna distribución públicamente disponible de Linux. El proyecto Beowulf se enfoca a la distribución Red Hat Linux, si bien sus componentes pueden instalarse en cualquier distribución. Cualquier distribución moderna incluye los componentes necesarios para la configuración del equipo como una estación de trabajo en red; esto incluye el *kernel* de Linux, el conjunto de utilerías y *software* GNU[9], y una serie de programas y aplicaciones como compiladores y herramientas de cómputo científico.

Aquellos elementos que un *Beowulf* requiere adicionar a la distribución, están disponibles como paquetes adicionales y bibliotecas de desarrollo. Esto incluye los ya mencionados PVM y MPICH, que en sentido estricto son las únicas adiciones necesarias para poder ejecutar una aplicación de PVM o MPI en un *Beowulf*. Sin embargo, a fin de proporcionar más facilidades, opcionalmente el proyecto *Beowulf* ha desarrollado utilerías, en particular BPROC, que proporciona un espacio de procesos unificado para todo el *Beowulf*, y está en desarrollo una biblioteca de programación con memoria compartida.

De esta manera, el proyecto *Beowulf* ha proporcionado todos los elementos para la construcción de un *cluster* de este tipo; únicamente se requiere la instalación de Linux, configuración del ambiente de red, y la instalación de los paquetes

[9] *GNU's Not Unix*, el proyecto GNU se describe con más detalle en la sección (1.5.3).

adicionales para contar con un *Beowulf* operativo. Sin embargo, como se verá en el capítulo 2, este procedimiento puede no ser trivial.

Cabe notar que, como producto del apoyo que el proyecto *Beowulf* ha dado al desarrollo de Linux, todas las mejoras a los controladores de red de Linux realizadas por los desarrolladores de *Beowulf* han sido incorporadas a cada nueva versión del *kernel* de Linux, de modo que estos controladores no necesitan obtenerse de manera externa.

1.5 Linux

Como se describió anteriormente (1.4.3), el sistema operativo Linux es un componente de *software* esencial para un *cluster* tipo *Beowulf*, dando la base para la operación de los nodos del mismo. Conocer la historia, filosofía y características de Linux es de interés para poder comprender el por qué de su elección como base para los *clusters* tipo *Beowulf*.

1.5.1 ¿Qué es Linux?

Linux es una implementación, o clon, de un *kernel* Unix, originalmente escrita desde cero y sin código propietario. Oficialmente, "Linux es un clon de Unix escrito desde cero por Linus Torvalds con ayuda de un equipo de *hackers* en el Internet. Uno de sus objetivos es lograr la certificación POSIX".[10]

El término Linux viene de "Linus Unix".

1.5.2 Algunas características de Linux

Linux fue desarrollado originalmente para procesadores i386 y compatibles. Actualmente, además del *release* general del kernel, mantenido por Linus Torvalds y colaboradores, existen diversas versiones o *ports*, mantenidas por otros equipos de programadores. Teniendo en cuenta todos los *ports* existentes, Linux está disponible para una gama extensa de arquitecturas de microprocesadores:

[10] La definición oficial se tomó de *http://www.kernel.org*, que es el sitio oficial de distribución del *kernel* de Linux.

- Intel x86 (i386, i486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4)
- Procesadores compatibles con Intel x86 (Texas Instruments, Cyrix, AMD K5, K6, K6-II, K6-III, Athlon, Duron, Sledgehammer, IDT WinChip).
- Intel IA-64
- DEC/Compaq Alpha AXP
- SPARC/UltraSPARC
- Motorola/IBM PowerPC, G3 y G4
- Motorola 68x00
- ARM
- Hitachi SuperH
- IBM S/390
- SGI MIPS
- HP PA-RISC

Linux implementa todas las características de un *kernel* Unix moderno: multitarea real, protección de memoria, memoria compartida, memoria virtual, carga bajo demanda, funciones de red y muchas más.

Bajo Linux pueden ejecutarse la gran mayoría de los programas existentes actualmente para Unix (asumiendo que se cuente con un binario en formato compatible o el código fuente). Gracias a esto, Linux puede ejecutar aplicaciones de productividad personal, entornos de programación, ambientes gráficos, servidores para una gran cantidad de servicios, utilerías de administración, etc.

1.5.3 Un poco de historia

Existen dos vertientes históricas que deben analizarse para comprender el surgimiento y filosofía de Linux: el sistema Unix y el *software* Libre.

Unix

Unix es un sistema operativo interactivo, multitarea, y de tiempo compartido. Fue inventado por Ken Thompson con ayuda de Dennis Ritchie (creador del lenguaje C) de AT&T Bell Labs en 1969 como un sucesor al difunto proyecto Multics. En el periodo 1972-1974, Unix fue reimplementado en C, convirtiéndolo en el primer sistema operativo con código fuente portable. Unix ha sido desarrollado y expandido por mucha gente, convirtiéndolo en un ambiente de desarrollo sumamente poderoso.

Originalmente Unix fue desarrollado por AT&T. A finales de los 70, AT&T cedió el código fuente a algunas instituciones educativas. Una de éstas fue la Universidad de California en Berkeley. Berkeley añadió extensiones de redes a Unix y lo liberó bajo el nombre BSD Unix (Berkeley System Distribution). El Unix de BSD fue durante mucho tiempo técnicamente superior al AT&T. Así se identifican las dos corrientes o "sabores" de Unix: AT&T System V y BSD. Ambos contienen diferencias sutiles pero significativas, sin dejar de ser Unix.

Software Libre

El proyecto GNU y la *Free Software Foundation* (FSF) fueron concebidos y creados por Richard Stallman.

En 1971, cuando Stallman inició su trabajo en el Laboratorio de Inteligencia Artificial en el Instituto de Tecnología de Massachusetts (MIT), prácticamente todo el *software* existente era libre. Los programadores eran libres de colaborar entre sí, e incluso las compañías comúnmente distribuían *software* libre.

Al iniciar la década de los 80, casi todo el *software* disponible era propietario. Esto significa que tenía dueños que prohibían la cooperación entre los programadores y los usuarios.

Stallman, molesto por las limitaciones impuestas por el *software* propietario, renunció al Laboratorio de IA para dedicarse a construir un ambiente de trabajo completamente libre. Así nace el proyecto GNU (GNU's Not Unix) y la Free *software* Foundation (la FSF fue oficializada en 1985, si bien Stallman había estado promoviendo el software libre, particularmente su editor de texto EMACS, desde tiempo atrás).

El término "libre" se refiere a libertad, más que a bajo costo. La licencia GNU especifica que con todo programa se debe distribuir el código fuente, permitir modificaciones a él, y obligar a que toda distribución de un trabajo derivado se acompañe también del código fuente.

El proyecto GNU buscaba construir un sistema operativo compatible con Unix y todas las utilerías necesarias para que éste funcionara correctamente. Se escogió a Unix por ser un diseño probado, portable, y su gran base de usuarios instalada facilitaría la migración a GNU.

Todo el desarrollo para el proyecto GNU debía realizarse bajo la GPL (General Public License), para asegurarse legalmente de la libertad del desarrollo.

Para 1990, el proyecto GNU tenía prácticamente todas las utilerías y aplicaciones necesarias para un sistema Unix, excepto el *kernel* (núcleo). Se estaba trabajando en un *kernel* conocido como HURD. Es en este momento que Linux entra en la escena.

1.5.4 Aparición de Linux

Linus Torvalds, en aquél entonces estudiante de la Universidad de Finlandia, trabajaba con el sistema operativo Minix, un mini-clon de Unix desarrollado por Andrew Tanenbaum para cursos de sistemas operativos. Linus decidió modificar Minix para eliminar algunas de sus limitaciones, y obtener más conocimiento de la arquitectura Intel 386. En cierto punto, Linus optó por comenzar desde cero y escribió un kernel que implementaba la mayoría de las funciones de un *kernel* Unix. Linus liberó su trabajo bajo la licencia GPL. Así, fue posible integrar el *kernel* Linux con las utilerías desarrolladas por el proyecto GNU.

Quizá el punto más importante en la historia del desarrollo de Linux, fue cuando Linus Torvalds decidió hacerlo público, bajo la licencia GPL, a fin de permitir que los interesados pudieran revisar el código y asistir en el desarrollo. Esto aceleró su desarrollo, además de servir como base para un modelo de desarrollo de *software* "cooperativo" que es clave para el crecimiento acelerado que ha tenido Linux.

En 1991, Linus publicó la versión experimental 0.01 del *kernel* de Linux. Su funcionalidad era mínima y estaba pensado básicamente para que los interesados pudieran echar un vistazo al código fuente.

La primera versión considerada estable, Linux 1.0, fue liberada en 1994. Con Linux se implementó un esquema de numeración *mayor.menor.revisión*, cuyo objetivo era indicar a los usuarios cuáles eran las versiones estables. Ya que Linux es un sistema operativo siempre en evolución, se mantienen dos ramas de código fuente, una estable y una en desarrollo. Las versiones de desarrollo están indicadas por un número *menor* impar (1.1, 1.3, 2.1), mientras que las estables tienen número *menor* par (1.0,1.2, 2.0). Dentro de cada *release*, las revisiones buscan no agregar nuevas características, sino corregir posibles errores en las existentes, o agregar nuevo soporte de *hardware*. Esto asegura que el *kernel* funciona correctamente al tiempo que se mantiene lo más actualizado posible. La siguiente tabla muestra las fechas de liberación de algunos *kernel*s estables de Linux:

Versión	Fecha
1.0	13/mar/1994
1.0.9	17/abr/1994
1.2	7/mar/1995
1.2.13	2/ago/1995
2.0	9/jun/1996
2.0.39	9/ene/2001
2.2.19	25/mar/2001
2.4	4/ene/2001
2.4.12	11/oct/2001

En la tabla se aprecia que al ir evolucionando el *kernel*, se incrementa su complejidad y por tanto el tiempo requerido entre una versión estable y la siguiente. Las versiones indicadas con números de revisión (1.0.9, 1.2.13, 2.0.39, 2.2.19 y 2.4.12) representan las últimas revisiones de cada rama estable. De las fechas de liberación se observa que un *kernel* se puede seguir actualizando para responder a necesidades de soporte de *hardware*, corrección de errores y seguridad, aún cuando ya no represente el desarrollo más reciente (el ejemplo ideal es el *kernel* 2.0.39, que contiene mejoras de seguridad importantes).

Recordando que un sistema Unix se compone del *kernel* y una serie de programas, utilerías, archivos de configuración y aplicaciones, es de suponerse que es necesario distribuir, además del *kernel*, todos estos aditamentos. Algunas personas

se dedicaron a empacar el *kernel* de Linux con las utilerías GNU, configuraciones prefabricadas y métodos de instalación. Estas son las conocidas como "distribuciones". Cada distribución tiene diferencias sutiles con las otras (por ejemplo, el estilo de los archivos de configuración, que pueden ser estilo BSD o System V), aunque todas están basadas en Linux.

Las distribuciones más populares son:

- Red Hat Linux. Creada por Red Hat *software*, es una de las distribuciones más populares, contando con desarrollo activo por parte de la compañía, una variante comercial que cuenta con soporte y asistencia técnica, una variante libre, avanzadas utilerías de configuración e instalación, y gran apego al espíritu GNU.

- Slackware Linux. Creada por Patrick Volkerding y una de las primeras distribuciones de Linux, si bien no cuenta con herramientas tan avanzadas como Red Hat y el desarrollo es un tanto lento.

- Debian GNU/Linux. La distribución patrocinada por la FSF, cuenta con una enorme comunidad de desarrollo, el respaldo tecnológico y filosófico de la FSF y algunas de las utilerías más avanzadas.

- Algunas distribuciones comerciales, como SuSE y Caldera, toman las distribuciones básicas (por ejemplo Red Hat) y las expanden con utilerías de configuración propias, ambientes de escritorio, versiones Linux de *software* comercial (WordPerfect, Netscape, CDE), y apoyo técnico.

1.5.5 Utilización de Linux en *clusters* tipo *Beowulf*

Como el sistema operativo utilizado en los nodos de un *Beowulf*, Linux es un componente crítico y esencial para la operación de este tipo de *cluster*. La elección de Linux para el proyecto *Beowulf* obedece a una serie de criterios; algunos de ellos se han descrito en la sección (1.4.3). En retrospectiva, el momento de la creación del primer *Beowulf* está determinado, por un lado, por un punto en la evolución del cómputo personal en que se hizo factible la incorporación de componentes comerciales comunes como plataforma de cómputo de alto rendimiento; y por otro lado, por la aparición del sistema operativo Linux, en 1994-1995.

Linux proporcionó a los creadores del proyecto *Beowulf* una plataforma muy prometedora para realizar el desarrollo de su proyecto. Bajo Linux se tenía un ambiente de trabajo y desarrollo Unix, que resultaba familiar para la mayoría de los usuarios de equipos de alto rendimiento, así como para los programadores que buscaban una plataforma para todos los elementos que componen un *Beowulf*. Adicionalmente, Linux contaba con las utilerías de trabajo y desarrollo de GNU, que eran también ampliamente utilizadas en otros ambientes Unix. Finalmente, ya que el código fuente de Linux estaba disponible libremente y bajo la licencia GNU GPL, se permitía a los programadores realizar modificaciones al mismo, liberarlas públicamente para su revisión y mejora por parte de una comunidad mundial, e incorporarlas posteriormente al sistema Linux para contribuir a su mejora.

Dada la naturaleza "hágalo usted mismo" del proyecto *Beowulf*, la elección de Linux sobre variantes de Unix comerciales resulta bastante obvia, teniendo en mente que el proyecto *Beowulf* buscaba evitar incurrir en altos costos, así como quedar "atrapados" con un solo fabricante, no sólo de *hardware* sino de *software*. Un sistema abierto y libre es, pues, la elección obvia.

Aún con la existencia de variantes libres del sistema BSD (FreeBSD 1.0 está disponible desde 1993; NetBSD 1.0, desde 1994), Linux fue la opción más razonable, probablemente debido al modelo de desarrollo altamente dinámico y cooperativo. Los BSD libres tienen un modelo de desarrollo más monolítico, basado en un "núcleo" (*core*) de programadores que tienen autorización para realizar cambios al código fuente del sistema, así como políticas de liberación más cerradas; aún teniendo una licencia del tipo libre, la licencia BSD es más vulnerable a apropiaciones indebidas del código que la GNU GPL.

El modelo de desarrollo de Linux se presta a grandes mejoras en rendimiento y estabilidad entre una versión del *kernel* y otra. En el desarrollo participan programadores con gran conocimiento en teoría de sistemas operativos, de manera que constantemente se están revisando y mejorando aspectos clave del *kernel*, redundando en incrementos de rendimiento, aprovechamiento de recursos y robustez. A lo largo de su desarrollo, se han mejorado significativamente aspectos como la programación de ejecución de procesos, administración de memoria, acceso a memoria y discos, protección de memoria, el subsistema de red de alto nivel, y los controladores de red de bajo nivel (que, como se mencionó, han tenido grandes

contribuciones de parte del proyecto *Beowulf*). Toda aplicación que requiera gran rendimiento, incluidas aquellas que se ejecutan en un *Beowulf*, se benefician de estas mejoras al sistema.

1.6 Resumen

En el capítulo 1 se proporcionó un breve recorrido por los avances de la tecnología de cómputo encaminados a aumentar el rendimiento de las computadoras, así como algunas ideas de por qué dicho rendimiento es necesario. Se describió la técnica del cómputo paralelo, sus variantes, ventajas, características y limitaciones. Se presentaron las características, técnicas y herramientas de programación empleadas en equipos masivamente paralelos. Se introdujo el concepto de *cluster* como una clase de máquina paralela, así como la historia e idea de los *clusters* tipo *Beowulf*, y las características, ventajas, desventajas y requerimientos, tanto de los *clusters* en general como de los *Beowulf*. Finalmente se dio un vistazo a las características, historia y filosofía del sistema operativo Linux, su papel dentro de la aparición de los *Beowulf*, y cómo las características de Linux lo hicieron la elección más conveniente para un *Beowulf*.

En el capítulo 2 se describirá la construcción de un *cluster* tipo *Beowulf* aplicando los conceptos y conocimientos que se presentaron en el presente capítulo.

Capítulo 2

Construcción del *cluster*

Parte del objetivo del proyecto es la construcción de un *cluster* integrando el equipo disponible. Como se mencionó anteriormente, un *cluster* tipo *Beowulf* está compuesto por equipos y componentes disponibles comercialmente. Dentro de los componentes básicos que forman un *cluster* (equipos que fungen como nodos de procesamiento, medio de comunicación entre nodos, y *software* de sistema y aplicación), se pueden realizar diversas elecciones que resultan en un sinfín de combinaciones posibles. Obviamente estas elecciones deben tener en cuenta dos factores básicos: costo y rendimiento. En general, aquellos componentes de mayor costo tienen un rendimiento más elevado.

En este caso se cuenta con cierta libertad en la selección y configuración del *software* a utilizar; Linux es versátil en cuanto a las características de los equipos en los que se va a ejecutar, de forma que se pudo contar con la libertad de adaptar el *software* según la plataforma de *hardware* con la que se contaba.

2.1 *Hardware*

Quizá la restricción más grande que se tuvo en este proyecto fue, precisamente, el hardware sobre el cual se va a construir el *cluster*. Por tratarse de un proyecto de naturaleza académica, en el cual se busca probar las teorías y técnicas de los *clusters* para obtener un rendimiento mayor al de soluciones uniprocesador, no se contó con presupuesto para adquisición de equipo. Se tuvo, entonces, que reunir

equipo que estaba en desuso, evaluar las características y posibilidades del mismo, y de acuerdo a esto generar una configuración tanto en *software* como en *hardware* que permitiera contar, finalmente con un *cluster* utilizable como plataforma de cómputo.

Para la construcción del *cluster* se consiguió, en primer lugar, un equipo Pentium con 16 MB RAM, disco duro de 10 GB, y dos tarjetas de red. Este equipo se designó como el nodo central del *cluster*. Desde este equipo los usuarios crearán y ejecutarán sus programas. Se cuenta con dos tarjetas de red para que una de ellas proporcione acceso a la red pública, y la otra esté dedicada exclusivamente a la red de interconexión del *cluster*; la red de comunicaciones dedicada es una característica importante de los *clusters* tipo *Beowulf*.

Adicionalmente se consiguieron los siguientes equipos:

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

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

2.1.1 Comunicación entre nodos

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

Desde el punto de vista del rendimiento una red Ethernet simple no es una muy buena elección para un *cluster*. Los tiempos de latencia son relativamente

elevados, del orden de algunos milisegundos. El ancho de banda de 10 Mbps es poco adecuado para aplicaciones que requieran un mayor nivel de comunicaciones entre nodos. Y la arquitectura de bus proporciona un canal de comunicaciones compartido con retransmisión (*broadcast*) que tiende a saturarse muy rápidamente a medida que la cantidad de mensajes pasados entre nodos se incrementa.

Sin embargo, el uso de la red Ethernet tiene ciertas ventajas y características interesantes. Una de ellas es su facilidad de instalación y bajo costo, que en particular fueron claves para su elección en este proyecto dadas las restricciones con que se realizó. Por otro lado, la popularidad de la tecnología Ethernet ha llevado a desarrollos que permiten incrementar el desempeño según crezcan las necesidades. Un *cluster* puede beneficiarse con el uso de *switches*, que segmentan el tráfico en el bus Ethernet y reducen la saturación y colisiones en el mismo. Y se puede contar con incrementos de desempeño inmediatos utilizando Fast Ethernet (100 Mbps) y Gigabit Ethernet (1 Gbps).

2.1.2 Consideraciones para equipos sin disco duro

Dado que los nodos de procesamiento no cuentan con disco duro, se decidió configurarlos como estaciones sin disco duro. El uso de estaciones *diskless* (sin disco), como se conocen comúnmente, está bastante difundido, pues permite un desempeño aceptable para terminales que normalmente fungen como despliegue del trabajo realizado en un servidor multiusuario. Las terminales *diskless* requieren un mínimo de trabajo de mantenimiento y configuración, y éstos se realizan básicamente en un servidor central, facilitando estas tareas.

En este caso, se da un enfoque un tanto diferente. El recurso de interés en las estaciones es su procesador y memoria, como elementos de trabajo básicos del *cluster*. Adicionalmente, no se pretende que los usuarios tengan aceso a estas estaciones directamente. La técnica de arranque *diskless* proporciona ventajas, como son la centralización de todos los archivos de los nodos en un servidor central, y cierta economía en los requerimientos de equipo, pues se evita la necesidad de contar con disco duro en cada uno de ellos.

El uso de esta técnica es una extensión del uso del sistema de archivos por red (Network File System o NFS). NFS normalmente se emplea para compartir los directorios de usuarios en redes de estaciones de trabajo, y en *clusters* suele

emplearse para facilitar la distribución de los programas a ejecutar.

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

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

En general esta consideración en cuanto al desempeño del acceso a archivos tendrá un impacto que debe tomarse en cuenta, en el *overhead*[1] de arranque de los procesos; si se diseñan cuidadosamente los programas, esto no repercutirá en el desempeño durante la realización de cálculos.

2.1.3 Integración de *hardware*

Habiendo tenido en cuenta los factores anteriores, la integración del *hardware* fue sencilla. Se busca que una vez ensamblado el *cluster* el diseño corresponda al esquema presentado en la figura (2.1).

El *cluster* se ensambló en instalaciones compartidas por el Laboratorio de Telemática y el Grupo de Usuarios de Linux de la Facultad de Ingeniería, en espacio donado para tal efecto.

De los equipos con que se dispone, dos cuentan con gabinete estilo minitorre. Los restantes 16 tienen gabinetes de tipo escritorio. Los equipos minitorre se dispusieron al centro, con el servidor en la parte inferior. Los nodos tipo escritorio se

[1] el término *overhead* habitualmente se usa para describir el tiempo empleado en la preparación para la realización de alguna tarea, procedimiento necesario pero que no forma parte de la tarea como tal.

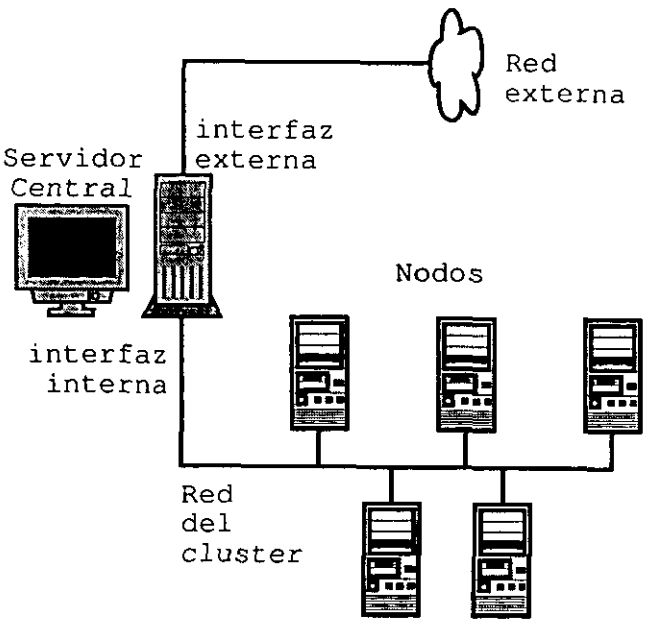

Figura 2.1: Esquema de construcción del *cluster*

organizaron en dos "torres" a cada lado del servidor central. El monitor y teclado del servidor se ubicaron al centro. Esto resulta en la disposición mostrada en la figura (2.2).

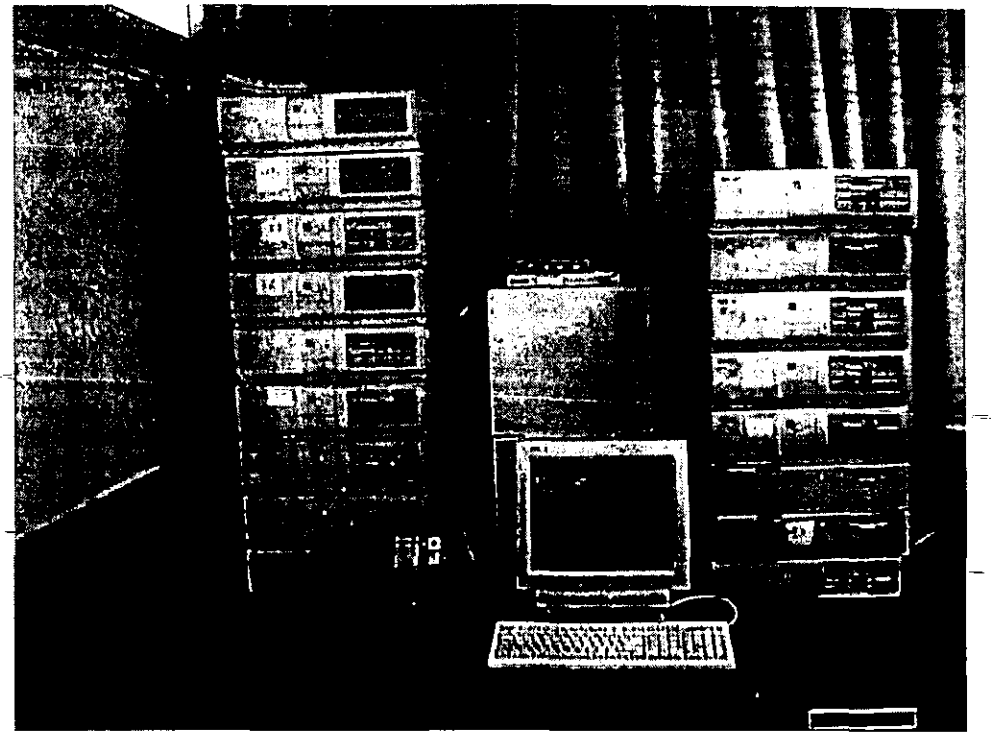

Figura 2.2: Vista frontal del *cluster*

Los concentradores para la red Ethernet forman la columna vertebral del mecanismo de interconexión. En este caso se requería un total de 18 puertos, por lo cual se utilizaron dos concentradores: uno de 16 puertos y otro de 8 puertos, los cuales se conectan en cascada, y en ellos se realiza la conexión de los nodos.

La figura (2.3) muestra el montaje de los concentradores sobre uno de los nodos minitorre, en la parte central del *cluster*. En este caso se aprecian los indicadores de actividad encendidos.

La figura (2.4) muestra el aspecto de las conexiones de red en la parte posterior de los concentradores. Se trata de un cable tipo UTP con conector RJ45 por cada nodo.

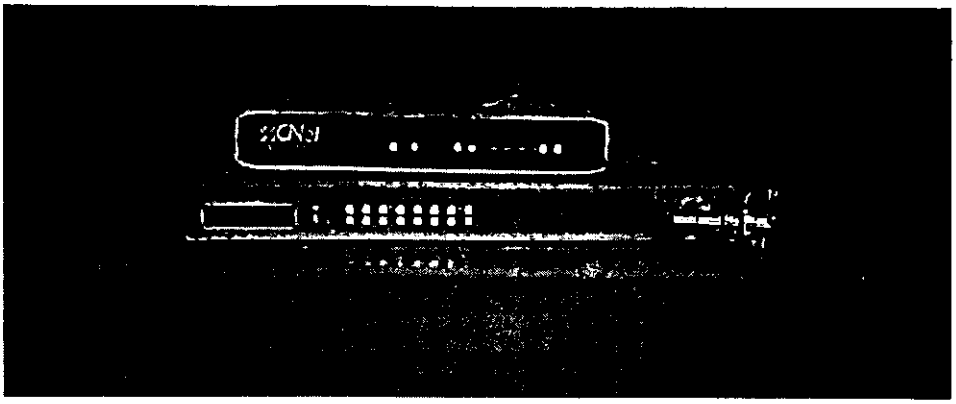

Figura 2.3: Vista frontal de los concentradores

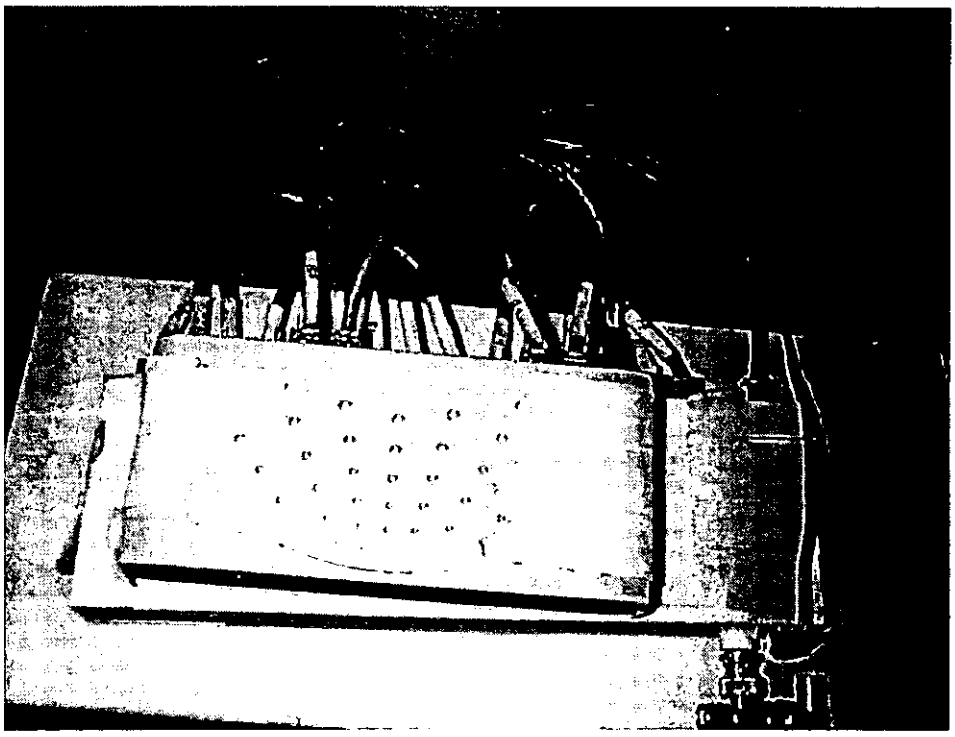

Figura 2.4: Conexiones de red

El servidor central cuenta con dos tarjetas de red, la que se designó para su uso en la red del *cluster* se conecta en estos concentradores. La otra tarjeta permite al servidor tener acceso a la red pública y se conecta en las facilidades provistas para ello por el Laboratorio.

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

El aspecto final de la conexión eléctrica se aprecia en la figura (2.5).

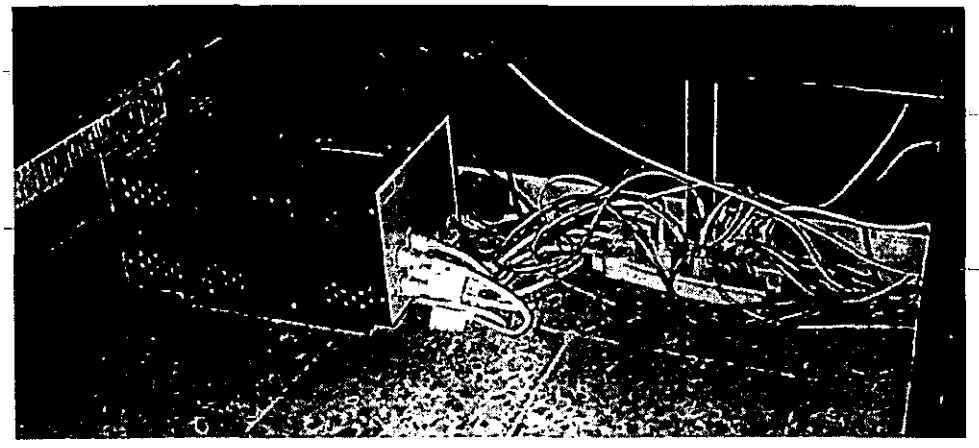

Figura 2.5: Conexión eléctrica

Una vez realizada la instalación física del equipo, se requiere únicamente asegurarse de que cada nodo funcione adecuadamente, y cuente con tarjeta de red y unidad de disco flexible. Esto deja una plataforma de *hardware* preparada para la instalación y configuración del *software*.

Figura 2.6: Aspecto final del *cluster*

2.2 *Software*

Partiendo de la plataforma de *hardware* descrita anteriormente, la instalación y configuración de todo el *software* que se empleará en el sistema puede visualizarse en etapas progresivas:

1. Instalación y arranque del sistema operativo en el servidor central.
2. Instalación y configuración de *software* de iniciación en los nodos.
3. Configuración de servidor y nodos para operación con sistema de archivos de red (NFS) y sistema de informacion de red (NIS).
4. Configuración y organización de sistemas de archivos individuales para los nodos.
5. Configuración y organización de área de archivos compartidos
6. Instalación y configuración de *software* requerido para aplicaciones paralelas.

2.2.1 Instalación y arranque del sistema operativo en el servidor central

El sistema operativo en el servidor central servirá como base para la creación de los directorios o sistemas de archivos para los nodos. Este servidor debe contar con el *software* para proporcionar los servicios requeridos para el arranque y operación de los nodos con la configuración *diskless*; ya que es el punto de entrada o *front end* para los usuarios del *cluster*, debe estar configurado para permitir la entrada a los usuarios al sistema. Se requiere que cuente con el *software* de programación paralela, para poder ejecutar los programas creados por los usuarios; y debe contar también con herramientas de desarrollo (editor de texto, compilador, depurador) para que los usuarios puedan desarrollar en el servidor los programas a ejecutar en el *cluster*.

Uno de los puntos clave para la elección de Linux como sistema operativo para este *cluster* fue el hecho de que todo el *software* requerido está disponible para Linux. Algunos de estos componentes de software (herramientas de programación,

servicios de archivos e información de red compartida como NFS y NIS, y herramientas para configuración y arranque de estaciones *diskless*) son elementos que se desarrollaron para permitir que Linux fuera una alternativa viable para su uso en redes tradicionales. Algunos otros componentes fueron desarrollados por el proyecto Beowulf para el fin explícito de tener un *cluster* basado en Linux. Por ello la elección de Linux es práctica en algunos aspectos y obvia en algunos otros.

De entre las distribuciones de Linux que se pueden utilizar, prácticamente cualquiera incluye el *software* necesario; aún en caso contrario, obtener el *software* a través de internet para su instalación es una tarea relativamente sencilla. Para este proyecto se eligió la distribución Red Hat, versión 6.2. Esta elección obedece a varios criterios. Se trata de una de las distribuciones más establecidas de Linux y está disponible de manera gratuita a través de internet. Red Hat cuenta con el sistema de administración de paquetes RPM [2]; este sistema maneja la instalación y desinstalación automática de software en el sistema, a diferencia de algunas distribuciones que requieren obtener los programas en código fuente, compilarlos e instalarlos manualmente. Si bien para el proyecto se necesita emplear software que sólo está disponible en forma de código fuente, el contar con un administrador de paquetes para el *software* que esté disponible en esta forma puede ahorrar tiempo y trabajo innecesario, permitiendo enfocar los esfuerzos a tareas más complicadas.

Red Hat Linux cuenta con una rutina de instalación muy sencilla. Para la instalación se requirió instalar una unidad de CD-ROM en el servidor central, que se utiliza únicamente durante la instalación. Se inicia el servidor con un diskette de arranque de Red Hat Linux. Esto inicia el programa de instalación.

El programa presenta tres modalidades de instalación: estación de trabajo, servidor, e instalación personalizada. Se eligió la instalación personalizada para contar con más control sobre el proceso de instalación, en particular respecto a la selección de los paquetes que se van a instalar, y que no es posible en los dos modos de instalación rápida, y a la creación de particiones en el sistema.

Se eligió crear dos particiones principales: una partición pequeña que contendrá el cargador de arranque y el *kernel*, que se monta bajo el directorio **/boot**; y una partición que abarca el resto del disco duro y se monta como directorio raíz **/**. Además se especificó una partición de intercambio (*swap*), requerida para el

[2] *Red Hat Package Manager*, administrador de paquetes de Red Hat.

funcionamiento del sistema.

El programa de instalación permite seleccionar grupos de paquetes; estos están organizados según la funcionalidad de los paquetes. Para la instalación se eligen los siguientes grupos:

- X Window System
- GNOME
- Mail/WWW/News tools
- Graphics manipulation
- Networked workstation
- NFS Server
- Authoring/Publishing
- Emacs
- Development
- Kernel Development
- Clustering
- Utilities

Esto instala la mayoría de los programas y servicios requeridos por el servidor.

A continuación el programa de instalación permite la selección individual de paquetes. Algunos de los paquetes requeridos se incluyen en la distribución Red Hat, sin embargo no se instalan a menos que se seleccionen explícitamente. En este caso se deben agregar los siguientes paquetes:

- dhcpd

El programa de instalación ofrece configurar la red, si el sistema cuenta con ella. Esto permite configurar la primera interfaz de red (téngase en mente que el

servidor cuenta con dos). La segunda interfaz, correspondiente a la red interna, se debe configurar explícitamente y esto se realiza una vez instalado el sistema.

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

- **IP Address**: 192.168.1.3
- **Netmask**: 255.255.255.0
- **Default Gateway**: 192.168.1.1
- **Primary Nameserver**: 132.248.10.2

Tras especificar estas opciones el programa de instalación realiza el formateo del disco duro e instala el sistema con la configuración especificada.

Una vez realizada la instalación del sistema, es necesario configurar la segunda interfaz de red. Esta segunda interfaz se denomina *eth1* (la primera interfaz que se configuró automáticamente se identifica como *eth0*).

A fin de que el sistema pueda cargar automáticamente el módulo o manejador correspondiente al *hardware* de la tarjeta de red, se agrega la siguiente línea en el archivo **/etc/conf.modules**:

```
alias eth1 ne2k-pci
```

Esto asocia el dispositivo *eth1* con el manejador de *hardware* ne2k-pci (que corresponde a la tarjeta de red utilizada), de modo que se cargue el manejador adecuado cuando el sistema lo solicita.

La interfaz *eth1* corresponde a la red interna para intercomunicación entre nodos. Ya que esta red está aislada de la red pública, se pueden asignar arbitrariamente los datos, tales como la red a utilizar, rango de direcciones IP, etc.

Arbitrariamente se seleccionó la subred 192.168.10.0 con máscara de subred de 24 bits (255.255.255.0). Esto proporciona 253 direcciones IP utilizables, de la 192.168.10.1 a la 192.168.10.254. Véase que esta subred está dentro de las redes especificadas como "privadas", para uso interno, en el RFC 1918 [3]. El uso de

estas direcciones para la red interna garantiza que no existirán conflictos al intentar revisar otros sitios de internet; si bien esta característica es importante únicamente para el servidor central, ya que es el único que cuenta con acceso a la red pública.

En el servidor se asigna la dirección 192.168.10.1 a la interfaz *eth1*. El establecer la configuración de red puede realizarse de dos maneras. Manualmente, y entrando al sistema como superusuario[3], se deben especificar los siguientes comandos:

```
# ifconfig eth1 192.168.1.0 netmask 255.255.255.0
# route add -net 192.168.1.0 netmask 255.255.255.0 dev eth1
```

Esto configura la interfaz y agrega una ruta a la subred correspondiente.

Opcionalmente, Red Hat proporciona un mecanismo de configuración automática, donde únicamente se requiere crear un archivo con la información relevante, y el sistema se encarga de levantar la interfaz al momento de iniciar. Para utilizar este mecanismo se requiere crear un archivo */etc/sysconfig/network-scripts/ifcfg-eth1* que contenga lo siguiente:

```
DEVICE=eth1
BOOTPROTO=none
IPADDR=192.168.10.1
NETMASK=255.255.255.0
ONBOOT=yes
```

En el servidor se optó por emplear el segundo mecanismo. De esta manera la red queda configurada tanto para entrada de usuarios por la interfaz *eth0* como para la comunicación con los nodos en la interfaz *eth1*.

2.2.2 Instalación y configuración de *software* de inicialización en los nodos

El arranque remoto de estaciones sin disco duro, técnica que se empleará para los nodos, puede emplearse para diversos sistemas operativos de red, como Novell y variantes de Unix. El método tradicional con redes Unix, que es el que se emplea en este caso, involucra 4 etapas:

[3]El "superusuario", o usuario *root*, es el que, en un sistema UNIX, cuenta con privilegios para realizar cualquier operación.

1. Al arrancar la computadora, se carga un programa conocido como "arrancador de red". Este es un programa que tradicionalmente reside en una ROM de arranque que se encuentra en la tarjeta de red.

2. El arrancador de red obtiene su dirección IP de un servidor, utilizando los protocolos BOOTP o DHCP. Con los datos entregados por el servidor el arrancador de red realiza configuración básica de la tarjeta de red para hacer transferencias por TCP/IP.

3. El arrancador de red utiliza el protocolo TFTP para transferir un archivo desde el servidor, cuya ubicación normalmente se especifica como parte de la configuración recibida por BOOTP o DHCP. Este archivo comúnmente es el *kernel* que debe cargar la estación para realizar su arranque.

4. Una vez cargado el *kernel*, termina el trabajo del arrancador de red; el *kernel* se carga normalmente y realiza su procedimiento de inicio.

Como se puede apreciar, esto involucra configuración de tres elementos básicos: el arrancador de red a ejecutar en los nodos, el servidor BOOTP o DHCP, y el servidor de TFTP; estos dos últimos elementos se configuran en el servidor.

Asignación automática de dirección IP

Tanto el protocolo BOOTP (*Bootstrap Protocol*) como el DHCP (*Dynamic Host Configuration Protocol*) permiten la asignación de información de configuración para estaciones de trabajo desde un servidor central. En ambos casos el cliente realiza una transmisión *broadcast* con su dirección de *hardware* (dirección MAC [4]). El servidor BOOTP o DHCP toma esta petición y regresa al cliente la información requerida, que básicamente consta de la dirección IP que el cliente deberá utilizar, y algunos otros datos. De particular importancia es un nombre de archivo que ayudará al cliente a realizar su arranque.

En este caso se optó por la utilización de DHCP, que es un protocolo más sofisticado y cuya configuración es más clara que la de BOOTP. DHCP proporciona la

[4] *Media Access Control*, control de acceso al medio. Todo dispositivo de red debe contar con una dirección MAC única para identificación.

posibilidad de enviar más información al cliente que BOOTP, y cuenta con algunas características como asignación dinámica de direcciones.

Red Hat incluye el servidor DHCP desarrollado por el ISC (*Internet Software Consortium*). Como una implementación de referencia, cuenta con todas las características del protocolo, que se especifican en el RFC 2031 [4]. Una vez instalado el paquete, se debe crear un archivo de configuración.

El archivo de configuración es relativamente sencillo, sin embargo es un tanto extenso ya que se requiere una sección *host* para cada nodo del *cluster*. El archivo completo se encuentra en el apéndice (B.1).

En este caso no se utiliza la capacidad de configuración dinámica de DHCP. Toda la configuración se especifica en las secciones *host*, en el modo que se conoce como "asignación manual", en la cual el administrador de red elige la configuración de cada estación y el protocolo únicamente se utiliza para enviar dicha información a las estaciones. Esto es conveniente pues se considera adecuado tener el control centralizado sobre la configuración de red de cada nodo.

En general el archivo consta de varias secciones *host* que tienen el formato mostrado en el siguiente ejemplo:

```
host tornado68 {
fixed-address 192.168.10.68;
hardware ethernet 00:60:08:0B:5A:9E;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado68";
}
```

Se puede apreciar que en cada sección *host* se asignan, con base en la dirección MAC (indicada por el parámetro *hardware ethernet*), las demás opciones de configuración. La más importante es la dirección IP de cada nodo (*fixed-address*). Éstas se asignan de manera progresiva y cuidando que sean únicas para cada host. Se decidió de manera arbitraria comenzar la asignación a partir de la dirección IP 192.168.10.68; en sentido estricto únicamente se requiere respetar la dirección IP del server (192.168.10.1) y todas las demás direcciones IP de la subred 192.168.10.0 están disponibles para su utilización. Otra opción de configuración

que es única para cada nodo es el nombre (*hostname*). Las dos opciones restantes para cada host son las mismas para todos: el nombre del archivo a cargar para el arranque (*filename*), que en este caso especifica la ruta, dentro del servidor TFTP, de un *kernel* Linux adecuado para el arranque de los nodos; y el servidor que entregará este archivo a los clientes (*next-server*). La razón de este último parámetro es que en ocasiones se puede tener un servidor TFTP que sea distinto del servidor DHCP; en la configuración default los clientes intentan cargar su archivo de arranque del mismo servidor que les entregó la configuración por DHCP, este parámetro permite alterar ese comportamiento.

Servidor de archivos de arranque TFTP

El protocolo TFTP (*Trivial File Transfer Protocol*) es un protocolo muy sencillo, basado en UDP, que permite bajar archivos de un servidor. Su principal utilidad es, precisamente, para proporcionar archivos de arranque a equipos que no cuentan con almacenamiento local. TFTP no cuenta con ninguna clase de control de acceso (contraseñas o nombres de usuario).

Red Hat Linux proporciona un servidor de tftp, contenido en el paquete *tftp-server*. Este paquete se encuentra instalado con la configuración de paquetes descrita anteriormente, sin embargo se encuentra normalmente deshabilitado. Para habilitarlo se debe agregar la siguiente línea en el archivo de configuración **/etc/inetd.conf**

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

esta es una línea de configuración tradicional del servidor *inetd*. En este caso se hace notar que el último parámetro (*/tftpboot*) indica el directorio que contiene los archivos a compartir por medio de TFTP.

Cargador de arranque

El programa encargado de iniciar la interfaz de red, obtener los datos de configuración básicos, y cargar por medio de TFTP el archivo especificado en esta configuración, es el cargador de arranque.

Para este proyecto se contaba, básicamente, con la elección de dos cargadores de arranque libres, que son los más utilizados en conjunto con redes Linux, y de

hecho son mencionados en la documentación sobre arranque de estaciones *diskless* [5]. Estos dos paquetes son Netboot y Etherboot.

Históricamente Netboot fue el primero en aparecer. Netboot utiliza los manejadores de paquetes (*packet drivers*) que se incluyen con casi cualquier tarjeta de red en el mercado, teniendo de esta manera gran compatibilidad con una extensa gama de tarjetas. Sin embargo, tiene dos desventajas básicas: la primera es que, por la inclusión de código externo y por tratarse de una base de código más antigua, es de tamaño un tanto grande, teniéndose en cuenta que el cargador de arranque normalmente se graba en una ROM que se adiciona a la tarjeta de red, las limitantes en tamaño de las ROM disponibles y el soporte a las mismas según cada tarjeta de red; la segunda es que, para tarjetas ISA, Netboot no realiza autoconfiguración, requiriéndose especificar los parámetros de la tarjeta (IRQ, puerto I/O) al momento de crear la imagen de arranque. Esto es importante porque representa una desventaja administrativa significativa, requiriendo mantener una imagen diferente para cada configuración de tarjeta, y ya que los equipos utilizados en el proyecto emplean tarjetas ISA, se considera esta limitación decisiva para no utilizar Netboot.

Etherboot es un desarrollo posterior, basado hasta cierto punto en Netboot, pero que ha sido reescrito proporcionando una base de código más limpia y compacta que la de Netboot. Etherboot utiliza manejadores internos y genera una imagen ROM para cada tipo de tarjeta de red soportada. El uso de manejadores internos permite que la imagen tenga un tamaño muy reducido que no da problemas con ninguna tarjeta de red soportada. Además, ya que los manejadores fueron desarrollados explícitamente para Etherboot, cuentan con autoconfiguración para tarjetas tipo ISA, lo que permite utilizar una sola imagen de arranque para cada tipo de tarjetas. El uso de Etherboot no se recomienda si se tienen tarjetas de red que no estén soportadas, ya que el soporte de Netboot es más extenso; en este caso se contaba únicamente con tarjetas que sí están soportadas por Etherboot, de manera que se decidió su uso para el arranque de los nodos.

Etherboot no se incluye en la distribución Red Hat Linux. Para utilizarlo se debe obtener el código fuente de la página *http://etherboot.sourceforge.net*. Para este proyecto se obtuvo la versión 4.6.7.

Para la instalación se debe desempacar el archivo obtenido con el siguiente comando:

```
# tar -zxvf etherboot-4.6.7.tar.gz
```

Esto crea un subdirectorio **etherboot-4.6.7**. A continuación cambiamos al subdirectorio **etherboot-4.6.7/src**. En este directorio se encuentra el código fuente. Aqui invocamos el comando **make**. Este compila todos los archivos fuente, generando las imágenes ROM y colocándolas en el directorio **etherboot-4.6.7/src/bin32**. En este directorio los archivos con extensión **.rom** son las imágenes ROM. En este caso son de particular interés los archivos **winbond940.rom** y **3c509.rom** pues estos corresponden a las tarjetas de red con que se cuenta.

En el paquete Etherboot se proporciona un pequeño cargador para discos flexibles, cuyo propósito es permitir probar las imágenes ROM antes de su grabado definitivo en una memoria EEPROM. Se decidió emplear este método pues es una manera sencilla de arrancar los nodos.

Para crear un diskette de arranque con Etherboot, se debe introducir un diskette limpio y formateado y utilizar el comando **make** como se indica:

```
# make bin32/3c509.fd0
# make bin32/winbond940.fd0
```

Cada uno de estos comandos creará un diskette de arranque para la tarjeta correspondiente.

Para utilizar el cargador de arranque se inserta el diskette correspondiente a cada nodo, según el tipo de tarjeta de red que cuenta. Se inicia el nodo con este diskette. El cargador de discos carga y ejecuta el programa Etherboot; éste solicita su dirección IP por medio de DHCP (Etherboot utiliza DHCP y si no obtiene respuesta intenta una petición BOOTP). Obtenida la dirección se comunica al servidor de TFTP y solicita el archivo especificado, que en este caso es **/tftpboot/vmlinuz-2.2-nbi**. Como hasta el momento este archivo no está disponible, el proceso falla y la carga se detiene. El paso siguiente será crear el archivo del *kernel* que utilizarán los nodos.

Creación del *kernel* para los nodos

El archivo que el servidor TFTP entregará a los nodos es un *kernel* de Linux funcional. Éste asume el control del sistema y realiza el arranque normal. Ya que la

configuración en las estaciones es bastante particular, el *kernel* debe contar internamente con las funciones necesarias para inicializar el dispositivo de red, obtener su configuración de un servidor remoto, y montar su sistema de archivos raíz a través de NFS. Una vez realizadas estas funciones, el *kernel* invoca al proceso init (funcionamiento tradicional en un sistema Unix) y el arranque prosigue normalmente.

La naturaleza modular del *kernel* de Linux permite una gran eficiencia y versatilidad en el manejo de los módulos que controlan a los dispositivos e implementan ciertas características a nivel kernel. Esto es práctico si se cuenta con almacenamiento local, pero en el caso de un nodo sin dichas facilidades, se requiere que el kernel contenga internamente todas las funciones necesarias para su arranque, al menos hasta el montaje del sistema de archivos raíz. En el ámbito de Linux, se dice que los módulos necesarios deben compilarse monolíticamente dentro del *kernel*. En este caso necesitamos compilar monolíticamente las siguientes opciones en el *kernel*:

- **Kernel level autoconfiguration**. Permite al *kernel* obtener su información de configuración a través de algún protocolo como DHCP o BOOTP. A continuación se seleccionan estos dos protocolos; el *kernel* lanza primero peticiones por DHCP y si no obtiene respuesta, efectúa la petición por medio de BOOTP. Se requiere que el *kernel* tenga esta opción, aún cuando el cargador de arranque (Etherboot) ya realizó esta petición, porque en general no existe un mecanismo para que el cargador de arranque pase la información obtenida al *kernel* al momento de iniciarlo. Esto efectivamente significa que para cada nodo, el servidor DHCP debe responder a dos peticiones iguales; sin embargo esto no supone ningún inconveniente para el sistema.

- **DHCP support**

- **BOOTP support**

- **NFS Filesystem Support**. Ya que todos los sistemas de archivos montados por los nodos residirán en un servidor NFS, esta opción es indispensable para la operación de los nodos.

- **Root File System on NFS**. Por medio de esta opción, el kernel monta un sistema de archivos en un servidor NFS como su sistema raíz. Basándose en la

información obtenida del servidor DHCP, el kernel intentará montar el directorio NFS **servidor:/tftpboot/hostname**. Los valores **servidor** y **hostname** son los enviados por el servidor DHCP (véase el parámetro **next-server** y **hostname** en el archivo de configuración para DHCP).

- **3c509/3c579 support**. Se requiere soporte monolítico en el *kernel* para las tarjetas de red que se piensan utilizar. Los manejadores realizan autodetección, de modo que no se necesita información adicional sobre las tarjetas, y se pueden compilar todos los que se necesitan pues el *kernel* cargará únicamente el que corresponda a la tarjeta con que cuenta cada nodo.

- **PCI NE2000 support**

Para configurar el *kernel* con las opciones requeridas, se pasa al directorio **/usr/src/linux**, que contiene el código fuente del kernel de Linux. Se puede utilizar la utilería **make menuconfig**, que presenta un menú fácil de utilizar, y se seleccionan las opciones especificadas (cuídese de especificar "y" para compilar las opciones en forma monolítica).

Otra posibilidad es editar directamente el archivo de configuración del *kernel* (**.config**), donde existe variables que especifican las opciones de configuración que se desean. Las opciones requeridas son las siguientes:

```
CONFIG_IP_PNP=y
CONFIG_IP_PNP_DHCP=y
CONFIG_IP_PNP_BOOTP=y
CONFIG_NET_VENDOR_3COM=y
CONFIG_EL3=y
CONFIG_NE2K_PCI=y
CONFIG_NFS_FS=y
CONFIG_ROOT_NFS=y
```

El archivo de configuración completo, utilizado para crear el *kernel* de los nodos, se encuentra en el apéndice (B.2)

Una vez realizada la configuración se procede a la compilación del kernel. Esto se realiza con los siguientes comandos, desde el directorio **/usr/src/linux**:

```
# make dep
# make clean
# make bzImage
```

Este proceso toma entre algunos minutos y unas horas, dependiendo de la capacidad del sistema donde se realice la compilación. Al término, el *kernel* está en el archivo
/usr/src/linux/arch/i386/bzImage.

Este *kernel* es adecuado para un sistema normal, pero para uso en un nodo del *cluster* se requiere un paso adicional. Se requiere que la imagen ejecutable cargada por el programa Etherboot contenga información adicional, que permita al cargador de arranque colocar la imagen en la locación de memoria correcta para su ejecución.

La distribución Etherboot contiene un programa para agregar esta información al *kernel*, denominado **mknbi**. Este programa permite crear archivos con formato NBI (Network Bootable Image). Para esto debemos pasar al directorio **etherboot-4.6.7/mknbi-1.0** y realizar lo siguiente:

```
# make
# make install
# /usr/local/lib/mknbi/mknbi --target=linux \
  --output=/tftpboot/vmlinuz-nbi-2.2 \
  /usr/src/linux/arch/i386/boot/bzImage
```

Estos comandos generan la utilería **mknbi** y sus archivos auxiliares y la instalan bajo **/usr/local/lib/mknbi**. El último comando invoca la utilería **mknbi**, especificando generar una imagen binaria para Linux, tomando el archivo de origen **/usr/src/linux/arch/i386/boot/bzImage**, que es el *kernel* que acabamos de compilar, y dejando la imagen en formato NBI en el archivo **/tftpboot/vmlinuz-nbi-2.2**. Nótese que esta ubicación corresponde a la especificada en el parámetro **filename** para todos los hosts, en el archivo de configuración de DHCP. Es decir, tras realizar este proceso la imagen del *kernel* queda lista para ser cargada por los nodos.

En este momento podemos volver a intentar el arranque de uno de los nodos. En esta ocasión el nodo obtendrá su configuración y logrará cargar el *kernel*. El

kernel realiza su proceso de arranque normal, pero se detendrá con un mensaje de error al intentar montar el sistema de archivos raíz por NFS. Se aprecia que el siguiente paso deberá ser configurar el servidor de NFS y proporcionar un directorio raíz adecuado para que el nodo lo monte y realice el resto del arranque de manera tradicional.

2.2.3 Organización de sistemas de archivos para NFS

Cada nodo requiere un sistema de archivos raíz que utilizará para el arranque. Estos directorios se exportarán a través de NFS y deben contener los archivos necesarios para el arranque del sistema.

La mayoría de las distribuciones de Linux, incluido Red Hat Linux, se adhieren a un estándar conocido como FHS (*Filesystem Hierarchy Standard*, estándar de jerarquía del sistema de archivos) [6]. El objetivo de contar con este estándar es el homologar la organización de los sistemas de archivos entre las distribuciones de Linux, para mejorar la interoperabilidad entre las aplicaciones, herramientas de administración de sistemas, herramientas de desarrollo y *scripts*, así como contar con una mayor uniformidad de uso y documentación para los sistemas que se adhieren el estándar.

Como se verá, la organización propuesta por el FHS se realizó teniendo en mente la posibilidad de sistemas *diskless* (como el caso de los nodos que se emplearán en el *cluster*), de modo que es conveniente revisar las premisas del FHS a fin de determinar cómo se pueden organizar nuestros sistemas de archivos raíz para los nodos, así como las áreas compartidas.

La especificación FHS indica que el contenido del directorio raíz debe ser adecuado para iniciar, restaurar, recuperar y/o reparar el sistema. En particular nos interesa la sección referente al arranque, que especifica que el contenido del sistema de archivos raíz debe incluir lo necesario para montar otros sistemas de archivos. Esto incluye utilerías, archivos de configuración, mecanismos de arranque, y otra información esencial para el inicio. Los directorios **/usr**, **/opt** y **/var** están organizados de modo que pueden estar ubicados en otras particiones o sistemas de archivos.

FHS intenta mantener la cantidad de archivos en el directorio raíz al mínimo (salvo para los directorios **/usr**, **/opt** y **/var** y **/home**) obedeciendo a algunos crite-

rios básicos. Uno de ellos es particularmente importante para estaciones *diskless*: el sistema de archivos raíz contiene muchos archivos de configuración específicos a cada sistema, como puede ser configuración de red o nombre del *host*. Esto implica que el sistema de archivos raíz no siempre se puede compartir entre sistemas en red. El mantener el sistema de archivos raíz lo más compacto posible minimiza el espacio desperdiciado por archivos no compartibles. También permite minimizar el tamaño del sistema de archivos inicial, sea local o remoto.

Los directorios de nivel superior, como lo especifica FHS, son los siguientes:

- **bin** binarios de comandos esenciales (uso público)
- **boot** archivos estáticos de arranque del sistema
- **dev** archivos de dispositivos
- **etc** configuración específica del sistema
- **home** directorios de usuarios
- **lib** bibliotecas compartidas esenciales
- **mnt** punto de montaje para otros sistemas de archivos
- **opt** *software* de aplicación adicional
- **root** directorio del superusuario
- **sbin** binarios esenciales del sistema
- **tmp** archivos temporales
- **usr** jerarquía secundaria
- **var** datos variables

Para los sistemas de archivos de los nodos, se omitirán los directorios **/usr** y **/home**, ya que estos serán compartidos entre todos los nodos y el servidor central.

A fin de generar el sistema de archivos para cada nodo, bajo el directorio **/tftpboot** se crean directorios con el *hostname* correspondiente a cada nodo, por

ejemplo: **/tftpboot/tornado68**. Bajo cada uno de estos se debe crear la jerarquía raíz para cada nodo. Para esto simplemente se copian los subdirectorios necesarios del servidor. Los directorios a copiar son:

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

Inicialmente se realiza únicamente una copia del directorio. Posteriormente la configuración por nodo se realiza en esta copia, y finalmente se crean tantas copias del primer directorio como nodos se tengan.

No se requiere copiar el directorio **/boot**, que contiene las imágenes ejecutables del *kernel* para el server, puesto que los nodos ya han cargado su *kernel* a través de TFTP. Se omite el directorio **/opt** pues este no existe inicialmente en Red Hat Linux y no se requerirá para este proyecto. Tampoco se requiere el directorio **/mnt** pues no se espera que los nodos vayan a montar sistemas de archivos no especificados. El directorio **/proc** contiene información de tiempo de ejecución del sistema y en Linux es requerido para el correcto funcionamiento; sin embargo es únicamente un directorio sin contenido pues su contenido se crea dinámicamente. **/usr** y **/home** también se omiten pues se montarán posteriormente como directorios compartidos.

El total del contenido de estos directorios abarca aproximadamente 15 MB de espacio. Cada nodo nuevo debe contar, en su propio subdirectorio, con una copia de todos estos archivos y directorios.

Para el directorio **/usr**, se compartirá directamente el directorio **/usr** del servidor. Según la especificación FHS, el directorio **/usr** debe contener únicamente información compartible y de sólo lectura. Esto nos garantiza que al compartirlo entre todos los nodos, no se tendrán problemas de inconsistencia o sincronización.

El directorio **/home** se comparte bajo el mismo mecanismo. De esta manera todas las entradas y administración de usuarios se realizan en el servidor central, los cambios y archivos de los usuarios se comparten entre todos los nodos.

2.2.4 Servidor NFS

El sistema de archivos en red (NFS) permite acceder a archivos ubicados en sistemas remotos tal como si se encontraran localmente. En este caso es de gran importancia ya que a través de NFS se proporcionarán los sistemas de archivos raíz y un área compartida para los nodos del *cluster*. El protocolo NFS fue desarrollado por Sun Microsystems, aunque está también publicado en el RFC 1094 [2], por lo tanto su uso está muy difundido como uno de los principales mecanismos para compartir archivos en redes de área local.

Linux cuenta con implementaciones NFS tanto para clientes como para servidores. Como se explicó en la página 66, el soporte para cliente NFS se compila directamente en el *kernel*. Así el kernel puede montar directamente sistemas de archivos que residen en otros servidores.

El *software* que permite a Linux fungir como servidor NFS está contenido en el paquete **nfs-utils**. Éste se incluye en la distribución Red Hat, sin embargo no se instala por default por lo que se debe agregar posteriormente. Tambien se debe habilitar el servicio NFS, de modo que al iniciar el sistema arranque el "demonio" NFS.

El demonio NFS requiere un archivo de configuración que le indique qué sistemas de archivos y directorios debe exportar, o hacer disponibles, así como varios parámetros que controlan el acceso que los clientes tendrán a estos sistemas de archivos.

El archivo de configuración que se debe crear es **/etc/exports**. Este archivo queda como sigue:

```
/tftpboot 192.168.10.0/255.255.255.0(rw,no_root_squash)
```

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

Cada línea indica el directorio a exportar, seguido de opciones que controlan el acceso al recurso. En este caso estamos especificando que los directorios solo podrán ser exportados a *hosts* con dirección IP dentro de la subred 192.168.10.0 y máscara 255.255.255.0 (lo cual corresponde precisamente a la subred que estamos empleando para los nodos del *cluster*). Los parámetros entre paréntesis indican los privilegios con que se exporta el recurso.

En este caso especificamos **rw** (*read/write*), lo cual indica que se permiten peticiones de lectura y escritura en el sistema exportado. Comúnmente se especifica la opción **ro** (*read only*), pero en este caso se requiere acceso total porque los nodos requerirán la capacidad de escribir en sus sistemas de archivos remotos.

El parámetro **no_root_squash** desactiva el "aplastamiento de *root*", que es el comportamiento por omisión al exportar por NFS. Normalmente, para evitar que un sistema remoto monte un sistema de archivos y el superusuario de ese sistema tenga acceso total a nuestro sistema, el NFS mapea las peticiones realizadas por el usuario con *uid*[5] 0 a un usuario anónimo con privilegios mínimos. En este caso se desea que los accesos con *uid* 0 no tengan un mapeo a un *uid* diferente, pues en los nodos sí se requieren accesos privilegiados. Esto no representa un riesgo de seguridad porque en este caso los accesos privilegiados están restringidos a los nodos, sobre los cuales se tiene bastante control administrativo.

2.2.5 Configuración servidor NIS

Una de las desventajas de NFS es que es un sistema de archivos remotos relativamente simplista. En general NFS exporta la información de permisos estilo Unix, así como el identificador numérico de usuario y grupo (*uid/gid*) de cada usuario, pero no realiza ninguna clase de control de acceso, dejando esta labor al sistema cliente. Se necesita entonces una manera de que el sistema cliente obtenga la información adecuada sobre la identidad del usuario y los permisos que el mismo debe tener sobre los archivos exportados.

[5]En sistemas UNIX, cada usuario es identificado por un valor numérico conocido como *uid* o *User ID*.

La opción más obvia es dar de alta los usuarios en todos los equipos en el mismo orden. Esto en teoría permitirá que el mapeo entre *uid* y nombres de usuario sea el mismo en todos los sistemas, ya que se están dando de alta en el mismo orden. Este enfoque es demasiado laborioso y se presta a errores, ya sea por falla del operador en el orden de alta de los usuarios o por la complejidad de realizar los cambios individualmente en cada una de las estaciones.

Otra posibilidad sería copiando el archivo **/etc/passwd** del servidor en cada cliente. De esta manera el mapeo entre *uid* y nombres de usuario sería consistente. Esto plantea un problema de sincronización pues es laborioso mantener las copias manualmente.

Como acompañante de NFS, Sun Microsystems desarrolló el protocolo NIS (*Network Information Service*, Servicio de Información de Red). NIS permite compartir información mantenida en un servidor central y que se hace accesible a los clientes, de una manera que se presta idóneamente para sincronizar, entre otras cosas, la información del archivo **/etc/passwd**.

En la arquitectura NIS se tienen entidades conocidas como dominios. Para cada dominio, existe un servidor maestro que es el que mantiene la información autoritaria y la comparte con los clientes. Los clientes se "amarran" al dominio, localizando al servidor correspondiente y realizando peticiones cuando requieren la información compartida.

El uso de dominios permite contar con varios grupos de servidores y clientes NIS, separando la información aún en el caso de que la red física sea la misma.

NIS proporciona facilidades de acceso a bases de datos basadas en llaves. Se observa que una propiedad interesante de los archivos que se pueden compartir por NIS es que regularmente el acceso se realiza por medio de una llave; especificado un *uid* o nombre de usuario, por ejemplo, se puede obtener el resto de su registro en el archivo **/etc/passwd**. NIS también puede utilizarse para compartir la tabla de *hosts* del archivo **/etc/hosts**. Dando como llave un *hostname*, el sistema NIS podría devolver su dirección IP, y viceversa.

El primer paso para montar NIS en una red es la configuración del servidor. La instalación que se realizó de Red Hat Linux ya contiene el *software* necesario.

Inicialmente se debe indicar al servidor el dominio NIS que va a atender. Esto se hace por medio de la utilería **domainname**, especificando el nombre del dominio

de la siguiente manera:

```
# domainname tornado
```

De esta manera se indica al sistema que pertenece al dominio **tornado**. Para no tener que realizar este procedimiento cada vez que se inicia el sistema se puede añadir la siguiente línea en el archivo **/etc/sysconfig/network**:

```
NISDOMAIN="tornado"
```

De esta manera los mecanismos de inicialización de Red Hat fijan automáticamente el dominio NIS.

Adicionalmente al server se le debe indicar que ejecute el demonio de NIS, conocido como **ypserv**. La mayoría de las utilerías que tienen que ver con NIS inician con las letras **yp**. Esto es por razones históricas; el sistema NIS también se conoce como *Yellow Pages*. Este último nombre cayó en desuso pues se trata de una marca registrada por otra compañía por lo que Sun se vio obligado a cambiar oficialmente el nombre del sistema a NIS; sin embargo las utilerías conservaron el prefijo **yp**.

Una vez configurado el nombre del dominio y funcionando el servidor **ypserv**, se deben crear los mapas en un formato que el servidor pueda manejar. Para esto se pasa al directorio **/var/yp**, donde están contenidos los archivos de información de NIS.

Se debe crear un subdirectorio para cada dominio que vayamos a servir (en este caso creamos un directorio **tornado**), y posteriormente ejecutar el comando **make**. En el directorio **/var/yp** se encuentra un *Makefile* que contiene información para crear los mapas a partir de los archivos originales. Una vez realizados estos pasos el servidor ya está compartiendo la información relevante.

2.2.6 Configuración por nodo

Montaje de los sistemas de archivos remotos

No es necesario tomar pasos adicionales para que cada nodo monte su sistema de archivos raíz. Como parte del arranque, el *kernel* montará el directorio NFS **192.168.10.1:/tftpboot/hostname** como su directorio raíz.

El archivo que indica los sistemas de archivos a montar una vez iniciado el sistema es el **/etc/fstab**. Ya que la configuración será la misma para todos los nodos, es conveniente realizar el cambio primero en el directorio **tornado68** que se creó para su posterior duplicación.

Para que los nodos monten los sistemas de archivos compartidos (**/home** y **/usr**), el archivo **/etc/fstab** debe quedar como sigue:

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

cada línea debe tener 6 elementos separados por espacios.

El primer elemento es el nombre o identificador del dispositivo a montar. El segundo elemento es el directorio sobre el cual se debe montar. El tercero indica el tipo de sistema de archivos que se encuentra en el dispositivo. El cuarto indica parámetros de montaje para el dispositivo. Los dos últimos elementos indican el orden en que se debe respaldar el sistema de archivos utilizando el comando **dump**.

En este caso las dos primeras líneas están configuradas de antemano. La primera indica el montaje del sistema de archivos **proc**, que contiene información de tiempo de ejecución del sistema y se genera dinámicamente. La segunda indica el montaje del sistema de archivos **devpts**, que genera dinámicamente las terminales virtuales para acceso remoto al sistema.

Las dos líneas siguientes indican el montaje de los sistemas **/usr** y **/home**. Éstos se montan a través de NFS (nótese el tercer parámetro especificando el tipo de sistema de archivos). La sintaxis para denotar el dispositivo a montar es **servidor:directorio**. Estas dos líneas montan los directorios **/usr** y **/home** del servidor en la misma ubicación en cada nodo.

Configuración cliente NIS

A fin de que un nodo cliente pueda compartir la información de un servidor NIS, se requiere ejecutar un programa que lo enlace al dominio NIS correspondiente, de modo que, cuando algún programa solicite información de las bases de datos compartidas, ésta pueda obtenerse del servidor NIS, y no de los archivos locales. De

esta manera se asegura que existe consistencia de información entre los clientes del dominio NIS, que como se mencionaba anteriormente, es necesaria, en particular para asegurar que la información de los permisos sobre los archivos sea la misma para todos los nodos.

El cliente NIS requiere fijar el nombre de dominio NIS al que va a pertenecer, de manera similar a la del servidor NIS, por medio del programa **domainname**:

```
# domainname tornado
```

De esta manera se indica al sistema que pertenece al dominio **tornado**. Para no tener que realizar este procedimiento cada vez que se inicia el sistema se puede añadir la siguiente línea en el archivo **/etc/sysconfig/network**:

```
NISDOMAIN="tornado"
```

Se observa que este procedimiento es igual tanto en el cliente como en el servidor.

Una vez que se ha fijado el valor de la variable NISDOMAIN, se requiere indicar el servidor que atenderá las peticiones NIS. Esto se configura en el archivo **/etc/yp.conf**. Se agrega la siguiente línea al archivo:

```
ypserver 192.168.10.1
```

Obsérvese que 192.168.10.1 es la dirección IP del servidor NIS.

Finalmente el cliente NIS debe ejecutar el programa que ejecuta las peticiones al servidor NIS, llamado **ypbind**. De nuevo se observa el prefijo **yp**, en este caso el nombre de la utilería indica que se "amarra" o "une" (*bind*) el cliente al dominio NIS previamente especificado.

2.2.7 Archivo /etc/hosts

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

El archivo contiene una lista de direcciones IP seguidas de nombres simbólicos. El contenido del archivo puede ser como sigue:

```
127.0.0.1 localhost
192.168.10.1 tornado
#nodos
192.168.10.68 tornado68
192.168.10.69 tornado69
192.168.10.70 tornado70
192.168.10.71 tornado71
```

Una vez agregada información al archivo, es importante recrear los mapas de NIS, como se menciona en la sección (2.2.5), ejecutando el comando **make** en el directorio **/var/yp**. De otro modo los nodos no tendrán acceso a esta información y el sistema no funcionará adecuadamente.

2.2.8 Toques finales

En este momento el nodo estará listo para realizar su arranque. Si en este momento encendemos la computadora nodo, debe realizarse el procedimiento de arranque remoto tal como se describio en la sección (2.2.2). Al terminar este proceso el nodo deberá presentar un *prompt* de entrada solicitando *login* y *password*. De hecho, cualquier usuario que tenga una cuenta en el servidor central debería poder tener acceso al nodo con sus datos habituales, así como poder tener acceso a sus archivos, ya que el directorio **/home** está compartido.

2.2.9 Configuración de archivo /etc/hosts.equiv

Bajo la estructura del *cluster* que se está construyendo, los usuarios tendrán acceso al servidor central, desde el cual ejecutarán sus aplicaciones escritas usando alguna biblioteca de paso de mensajes. Obviamente, se requiere algún mecanismo por medio del cual el servidor pueda iniciar programas en los nodos, que ya en ejecución colaboran entre sí para la resolución de algún problema.

En particular, y como se verá más adelante, MPICH y PVM emplean el mecanismo **rsh** (*remote shell*) para ejecutar los programas en los nodos.

rsh permite la ejecución de algún comando arbitrario en un equipo remoto. La sintaxis en su forma más básica es como sigue:

```
#  rsh usuario@sistema comando
```

De esta manera se ejecuta *comando* en un equipo llamado *servidor*, como el usuario *usuario*.

Desde luego, el servidor remoto debe dar autorización para esta ejecución, así como tener conocimiento de la existencia del usuario en cuestión.

En el *cluster*, el uso de NIS nos garantiza que todos los nodos tendrán conocimiento de los usuarios que entran al servidor central.

A fin de permitir la ejecución de comandos remotos por parte de algún otro equipo, sin embargo, cada nodo debe tener un archivo **/etc/hosts.equiv**. Aquí se colocan los nombres o direcciones de los equipos remotos a los que se dará autorización para ejecutar comandos.

Cabe hacer notar que este mecanismo no contempla ningun esquema de protección ni autenticación fuera del uso del archivo **/etc/hosts.equiv**. En redes públicas, el uso de **rsh** no es muy recomendado por su baja seguridad. Sin embargo, en un *Beowulf* se cuenta con una red aislada de la sección pública, por lo que el uso de **rsh** no conlleva un riesgo de seguridad inherente. Nótese que los únicos que permiten ejecución remota son los nodos; el servidor no requiere este mecanismo por lo cual no se cae en un riesgo de seguridad.

El archivo **/etc/hosts.equiv** en cada nodo queda como sigue:

```
tornado
192.168.10.1
```

Esto indica que se permite ejecución remota únicamente si se solicita desde la máquina **tornado** o **192.168.10.1**. En efecto, cada nodo permite así la ejecución de comandos remotos solicitados desde el servidor, que es el requisito para que PVM y MPICH puedan ejecutar programas en los nodos. Nótese que el uso del mecanismo **rsh** por parte de PVM y MPICH es totalmente transparente para el usuario. En general, y como se ha mencionado anteriormente, el usuario no interactúa directamente con los nodos, sino que realiza todo su trabajo en el servidor central.

Optimización de configuración

A fin de obtener una configuración óptima para el nodo, es recomendable entrar al nodo en modo superusuario, y utilizar la utilería **ntsysv** para deshabilitar todos los servicios que no son necesarios en el nodo. Esto minimiza el uso de recursos y los accesos innecesarios al sistema de archivos compartido.

Los únicos servicios que se deben dejar habilitados en cada nodo son:

- **inet** Super demonio de internet, necesario para admitir comandos **rsh** remotos.
- **netfs** Montaje de sistemas de archivos de red (NFS).
- **network** Inicialización de la configuración de red básica.
- **portmap** Requerido para el uso de NFS.
- **random** Generador de números aleatorios.
- **sshd** Demonio SSH, resulta útil para mantenimiento remoto de los nodos.
- **syslog** Demonio para registrar las actividades del sistema.
- **ypbind** Utilería mencionada en la sección (2.2.6).

Al concluirse estos pasos de configuración, el nodo está preparado para funcionar. Recuérdese que toda esta configuración se hizo exclusivamente para el primer nodo (tornado68). Dependiendo del número de nodos que se vayan a utilizar, se debe crear un directorio para cada uno bajo **/tftpboot**, de modo que cada nodo al arrancar pueda montar su directorio raíz de esta ubicación.

Para esto se utilizan los siguientes comandos:

```
# cd /tftpboot
# mkdir tornado69
# cp -a tornado68/* tornado69
```

Esto crea un directorio **/tftpboot/tornado69** y copia todos los archivos necesarios del directorio **/tftpboot/tornado68**. Utilizando este procedimiento se crean directorios para todos los nodos.

En este momento se debe poder iniciar todos los nodos con su correspondiente disco de arranque con Etherboot, debiendo todos ellos arrancar y quedar listos para su operación. Esto efectivamente constituye un *cluster*, como una plataforma de cómputo, sobre la cual aún resta instalar las bibliotecas que se emplearán para programar aplicaciones paralelas. Nótese que, una vez concluida la instalación por nodo, ya que el directorio **/usr** está compartido, todos los programas y bibliotecas que se instalen bajo dicho directorio quedarán también compartidos por todos los nodos; de manera que, si las instalaciones se realizan bajo **/usr** en el servidor, automáticamente los nodos podrán utilizar el *software* que ahí se encuentre.

2.2.10 Instalación y configuración de bibliotecas para programación paralela

Consideraciones previas

Tanto MPICH como PVM tienen una licencia que permite la distribución libre, así como la modificación del código. Adicionalmente ambas bibliotecas se distribuyen habitualmente en forma de código fuente, lo cual permite su compilación en prácticamente cualquier plataforma que cubra los requerimientos.

La manera más tradicional de instalarlos, pues, es la obtención del código fuente de los sitios de distribución en internet, descomprimir, compilar e instalar. Esto en la mayoría de los casos genera las bibliotecas y los archivos de encabezado necesarios para utilizarlas, y las instala en el lugar adecuado para su utilización.

En este caso se juzgó conveniente utilizar paquetes precompilados en formato RPM. Esta decisión obedece a algunos criterios. El uso de paquetes precompilados ahorra tiempo y recursos requeridos para la compilación de las bibliotecas, lo cual es una ventaja dado que la capacidad del servidor es limitada, y la compilación de paquetes complejos puede llevar un tiempo largo. Como un ejemplo, la compilación del *kernel* de Linux tomó alrededor de 40 minutos.

Por otro lado, el uso de paquetes RPM facilita la administración del *software* instalado en el sistema, en particular las tareas de instalación, desinstalación y actualización, lo cual es útil en el caso de bibliotecas como PVM y MPICH, en caso de que se generen problemas o se requiera actualizarlas.

Finalmente, el formato RPM cuenta con capacidad de reubicación; esto es,

proporciona facilidades para decidir bajo qué directorio se va a instalar el paquete. Esto es de utilidad pues se puede decidir que los paquetes queden instalados, por ejemplo, bajo **/usr**, que como se mencionó en la sección (2.2.9), es el directorio donde se puede instalar el *software* compartido por los nodos.

Restricciones para el superusuario

Es importante resaltar el hecho de que el superusuario no podrá hacer uso de programas escritos con PVM o MPI, dada la configuración actual del *cluster*. Este comportamiento obedece a las restricciones que, por seguridad, impone el mecanismo **rsh** al superusuario, quien no puede ejecutar comandos remotos con este comando. Ya que no se hace ninguna verificación sobre la identidad del usuario, no es conveniente permitir la ejecución de comandos remotos con privilegios de superusuario. Por lo tanto, el superusuario no podrá invocar programas escritos con PVM o MPI, pues éstos dependen del comando **rsh** para invocar procesos en los nodos.

Este comportamiento no supone inconveniente para el cluster; ya que la labor del superusuario se limita a instalación y administración de la plataforma del *cluster*, y en general todo el trabajo utilizando las bibliotecas de programación paralela se realiza bajo cuentas de usuarios normales. La estructura de este *cluster* está muy adecuada a este modo de trabajo, pues se espera que los usuarios escriban, compilen y ejecuten sus programas dentro de sus directorios, que están compartidos entre todos los nodos.

Configuración preliminar

Tanto PVM como MPICH requieren que esté configurada la ejecución de programas remotos por medio de **rsh** en cada nodo. Esta configuración se realizó en la sección (2.2.9), de modo que en este momento el sistema está preparado para la instalación de PVM o MPICH.

2.2.11 Instalación y configuración de PVM

La versión más actual de PVM es la 3.4.3, liberada el 29 de marzo del 2000.

La página en internet de PVM, mantenida por el Oak Ridge National Labs, se encuentra en *http://www.epm.ornl.gov/pvm/*. Aquí se puede encontrar documentación, información miscelánea, y un enlace a la distribución de PVM en código fuente, que se encuentra en *http://www.netlib.org/pvm3/pvm3.4.3.tgz*.

El paquete RPM utilizado en esta instalación se obtuvo en: *ftp://ftp.rpmfind.net/linux/redhat/6.2/en/os/i386/RedHat/RPMS/pvm-3.4.3-4.i386.rpm*.

La instalación se realiza con el siguiente comando, en modo superusuario:

```
# rpm -Uvh pvm-3.4.3-4.i386.rpm
```

Este paquete está creado de manera que la instalación se realiza bajo el directorio **/usr/share/pvm3**. Ya que esto queda bajo el directorio compartido **/usr**, no se requiere configuración adicional para que todos los nodos puedan utilizar PVM.

Configuración

Una vez instalado PVM, y a fin de poder utilizarlo, se requiere fijar dos variables de ambiente que permiten a PVM determinar la ubicación de sus archivos, así como la arquitectura de la máquina donde se está ejecutando. Dada la naturaleza homogénea de PVM, el conocer la arquitectura permitirá al sistema seleccionar el ejecutable adecuado a cada máquina.

En el archivo **$HOME/.bashrc**[6] se anexan las siguientes líneas:

```
export PVM_ROOT=/usr/share/pvm3
export PVM_ARCH=LINUX
```

En particular, **/usr/share/pvm3** es la ubicación donde el paquete RPM utilizado colocó los archivos de PVM.

Comprobación de funcionamiento

A fin de comprobar si PVM funciona correctamente, se inicia la consola PVM en el servidor central. Esto se debe realizar con una cuenta de usuario normal, por las razones mencionadas en la sección (2.2.10).

[6] **$HOME** representa el directorio hogar o *home directory* de cada usuario, es decir, cada usuario que desee utilizar PVM debe realizar esta adición a su archivo **.bashrc**.

```
$ pvm
```

Al ejecutar este comando, se inicia el demonio PVM en el servidor central y se nos presenta la consola de control de PVM, que aparece como sigue:

```
pvm>
```

Esto indica que la máquina virtual está lista para aceptar comandos. En este momento, la máquina virtual consta de únicamente un nodo, el servidor central. A fin de anexar nodos a la máquina virtual, para que puedan participar en las tareas colectivas, se utiliza el comando **add**, indicando el nombre de los nodos que queremos agregar. Si el comando es exitoso, se tendrá el despliegue siguiente:

```
pvm> add tornado70 tornado71 tornado72 tornado73
add tornado70 tornado71 tornado72 tornado73
4 successful
                    HOST     DTID
                tornado70   280000
                tornado71   2c0000
                tornado72   300000
                tornado73   340000
pvm>
```

Si por alguna razón la adición del nodo a la máquina virtual falla, el despliegue obtenido será como sigue:

```
pvm> add tornado70
0 successful

Auto-Diagnosing Failed Hosts...
tornado70...
Verifying Local Path to "rsh"...
Rsh found in /usr/bin/rsh - O.K.
Testing Rsh/Rhosts Access to Host "tornado70"...
Rsh/Rhosts Access is O.K.
Checking O.S. Type (Unix test) on Host "tornado70"...
```

```
Host tornado70 is Unix-based.
Checking $PVM_ROOT on Host "tornado70"...
$PVM_ROOT on tornado70 Appears O.K. ("/usr/share/pvm3")
Verifying Location of PVM Daemon Script on Host "tornado70"...
PVM Daemon Script Found ("/usr/share/pvm3/lib/pvmd")
Determining PVM Architecture on Host "tornado70"...
$PVM_ARCH on tornado70 set to
Manually Determining PVM Architecture on Host "tornado70"...
Could Not Determine $PVM_ARCH.
```

En general PVM nos informa cuántas adiciones fueron exitosas, y cuántas fallaron; en el caso de adiciones fallidas, PVM intenta determinar la causa del problema. Los problemas más comunes suelen ser: imposibilidad de invocar demonios PVM remotos por **rsh**, que se resuelve realizando la configuración descrita en la sección (2.2.9), y ausencia de la variable de ambiente **PVM_ROOT**, cuya configuración se describe en la sección (2.2.11).

Los diagnósticos realizados por PVM son relativamente completos, de modo que nos permiten resolver complicaciones rápidamente. Una vez que podemos agregar todos los nodos deseados a la máquina virtual, ésta queda lista para ejecución de programas paralelos. Un último recurso de comprobación de la correcta configuración de la máquina virtual es el comando de PVM *conf*, que muestra la configuración actual de la máquina virtual:

```
pvm> conf
7 hosts, 1 data format
                HOST     DTID     ARCH   SPEED       DSIG
             tornado    40000    LINUX    1000 0x00408841
           tornado70   280000    LINUX    1000 0x00408841
           tornado71   2c0000    LINUX    1000 0x00408841
           tornado72   300000    LINUX    1000 0x00408841
           tornado73   340000    LINUX    1000 0x00408841
           tornado85   380000    LINUX    1000 0x00408841
           tornado80   3c0000    LINUX    1000 0x00408841
pvm>
```

Aquí se puede comprobar el nombre y número de los hosts que componen la máquina virtual, así como la arquitectura, número base para identificadores de

tareas (DTID), velocidad relativa y DSIG.

2.2.12 Instalación y configuración de MPICH

La versión más actual de MPICH es la 1.2.2, liberada el 20 de agosto del 2001.

La página en internet de MPICH, mantenida por el Argonne National Labs, se encuentra en *http://www-unix.mcs.anl.gov/mpi/mpich/*. Junto con toda la documentación particular de MPICH, así como enlaces a la definición del estándar MPI, se encuentra un enlace a la distribución en código fuente de MPICH, en *ftp://ftp.mcs.anl.gov/pub/mpi/mpich.tar.gz*.

El paquete RPM utilizado en esta instalación se obtuvo en:
ftp://ftp.rpmfind.net/linux/redhat/6.2/en/powertools/i386/i386/mpich-1.2.0-5.i386.rpm.
La instalación se realiza con el siguiente comando, en modo superusuario:

```
# rpm -Uvh mpich-1.2.0-5.i386.rpm
```

Configuración

El correcto funcionamiento de MPICH requiere la configuración de una variable de ambiente, que agrega a la ruta de búsqueda la ubicación donde se encuentran los binarios de MPICH. Esta variable se puede fijar en el archivo **$HOME/.bashrc** de cada usuario, como sigue:

```
export PATH=$PATH:/usr/share/mpi/bin
```

Como se puede observar, esto es una simple adición a la ruta de búsqueda de cada usuario.

Ya que, como se mencionó en la sección (1.3.7), MPI como especificación no proporciona un mecanismo para control de procesos, la implementación particular de MPICH nos brinda dicho mecanismo, que cumple una funcionalidad un tanto similar a la adición de nodos a la máquina virtual por medio de la consola de PVM (sección (2.2.11). Se trata de un mecanismo muy sencillo que únicamente requiere agregar los nombres de los nodos que queremos que participen en las tareas colectivas, en un archivo ubicado en **/usr/share/mpi/share/machines.LINUX**. En general, MPICH cuenta con un archivo para cada arquitectura que deseemos utilizar, listando los nodos que corresponden a dicha arquitectura. Ya que en este caso

únicamente se cuenta con nodos Linux, únicamente es de interés el archivo mencionado. En la lista se agrega un nombre de nodo en cada línea, y puede quedar como sigue:

```
#Lista de nodos
tornado
tornado68
tornado69
tornado70
tornado71
tornado72
tornado73
tornado74
tornado75
```

Una vez completados estos dos pasos, está terminada la configuración de MPICH y se puede proceder a comprobar su operación.

Comprobación de funcionamiento

MPICH provee un *script* llamado **tstmachines** que verifica el funcionamiento correcto de todos los nodos descritos en el archivo **machines.LINUX**. Este *script* realiza pruebas completas de ejecución remota con **rsh** en todos los nodos, además de compilar y ejecutar un programa de prueba para verificar que todos los ejecutables estén accesibles. La prueba se debe realizar como un usuario normal, y ya que el *script* requiere permisos de escritura en el directorio actual, es conveniente cambiar al directorio **/tmp** antes de su ejecución. Los pasos necesarios para cambiar de directorio y realizar la prueba son como sigue:

```
$ cd /tmp
$ /usr/share/mpi/sbin/tstmachines -v
```

Nótese el parámetro **-v** que indica al *script* informar sobre los pasos que va realizando. De otra manera, se espera que, si no hay problemas, no se reciba ningún mensaje.

El *script* realiza su prueba nodo por nodo, y para los nodos donde el funcionamiento es correcto se tiene el siguiente despliegue:

```
Trying true on tornado68 ...
Trying ls on tornado68 ...
Trying user program on tornado68 ...
```

Si ocurre algún problema, el mensaje de diagnóstico indicará detalles sobre el mismo, así como posibles soluciones. Como un ejemplo, un nodo que no permita la ejecución por **rsh** presentará el siguiente diagnóstico:

```
Trying true on tornado75 ...
Errors while trying to run true
Unexpected response from tornado75:
--> Permission denied.
If your .cshrc, login, .bashrc, or other startup file
contains a command that generates any output when
logging in, such as fortune or hostname or even echo,
you should modify that startup file to only print such
a message when the process is attached to a terminal.
Examples of how to do this are in the Users Manual.  If
you do not do this, MPICH will still work, but this
script and the test programs will report problems
because they compare expected output from what the
programs produce.

    The test of rsh <machine> true failed on some
    machines.  This may be due to problems in your
    .login or .cshrc files; some common problems are
    described when detected.  Look at the output above
    to see what the problem is.

    If the problem is something like 'permission
    denied', then the remote shell command rsh does not
    allow you to run programs.  See the documentation
    about remote shell and rhosts.
```

```
1 errors were encountered while testing the machines
list for LINUX
```

Como se aprecia, el diagnóstico es bastante completo, y al final se presenta un resumen de la cantidad de nodos con error que se detectó. Al igual que con PVM, quizá los problemas más comunes son fallas en la configuración de **rsh** y de la ruta de búsqueda. Revisando que esta configuración se haya realizado correctamente en cada nodo que reporte error, se deben poder resolver los problemas rápidamente.

Una vez que todos los nodos se encuentran adecuadamente configurados, el *script* de prueba no reporta ningún error y el *cluster* queda listo para la ejecución de programas con MPI.

2.3 Resumen

En el capítulo 2 se describió la construcción del *cluster* tipo *Beowulf*, que constituye uno de los objetivos principales del proyecto. Se describió el *hardware* que compone el *cluster*, las elecciones de organización, interconexión y operación que se tomaron a partir del *hardware* con que se cuenta, la instalación y configuración del *software* que rige la operación del *cluster*, desde el sistema operativo Linux, pasando por la configuración necesaria para el arranque de nodos *diskless*, tanto en el server como en los propios nodos, la estructura de los sistemas de archivos compartidos, los servicios necesarios para compartir dichos sistemas de archivos, para finalmente contar con una plataforma de nodos interconectados por una red que constituye un *Beowulf*. Se describieron los procesos de instalación, configuración y verificación de funcionamiento de las bibliotecas PVM y MPI para cómputo paralelo. Al cabo de estos procesos se cuenta con un *cluster* tipo *Beowulf* funcional y listo para su utilización.

En el capítulo 3 se pondrá en uso el *cluster* a fin de conocer de forma práctica y experimental sus capacidades, limitaciones y características. Al realizar este proceso se detallarán las experiencias obtenidas al desarrollar y utilizar aplicaciones paralelas.

Capítulo 3

Utilización

Una vez contando con un *Beowulf* instalado y funcional, es posible comenzar a desarrollar y ejecutar aplicaciones paralelas. Dada la naturaleza experimental del *cluster*, y en particular teniendo en mente que los programas se pueden desarrollar utilizando cualquiera de las dos bibliotecas de programación paralelas instaladas, se consideró conveniente comenzar la utilización del *cluster* por medio de un programa sencillo que realiza una operación muy común y de fácil paralelización: la multiplicación de matrices.

3.1 Aplicación de evaluación: multiplicación de matrices

Ya que el algoritmo no es complicado, y se desea poder evaluar de manera experimental la creación de aplicaciones usando tanto PVM como MPI, así como el comparar el desempeño que se puede tener bajo cualquiera de estas dos bibliotecas, se optó por implementar el mismo algoritmo utilizando PVM y MPI. Esto permite familiarizarse con el uso de ambas bibliotecas; conocer y comparar sus filosofías de programación, la complejidad involucrada en su uso, la manera de compilar y ligar aplicaciones que las utilizan, y la manera de ejecutar dichas aplicaciones; y finalmente, permite comparar directamente el rendimiento de una aplicación cuya única diferencia es el uso de PVM o MPI, según sea el caso, y determinar si alguna de estas bibliotecas es más eficiente.

3.1.1 La operación de multiplicación de matrices

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

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

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

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

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

Se observa que la multiplicación de matrices es un problema muy fácilmente paralelizable; de hecho, se encuentra en la categoría de problemas conocidos como "vergonzosamente paralelizables"[1]. Esto es porque cada nodo que participe en el cálculo únicamente necesita conocer, antes de iniciar, los valores de las dos matrices a multiplicar; y en ningún momento requerirá comunicarse con los demás nodos para realizar su trabajo. Como se verá más adelante, la implementación más sencilla involucra comunicación con un proceso "maestro" o "padre" que recoge y consolida los resultados parciales de los nodos, pero en ningún momento se requerirá que los nodos detengan su cálculo para esperar información de otro nodo.

3.1.2 Una sencilla implementación de multiplicación de matrices

En el apéndice (A.1.2) se presenta un sencillo programa de multiplicación de matrices en C. Este programa se utilizará como punto de partida para las implementaciones paralelas. Como se trata de un programa con fines de pruebas, no está pensado

[1] *embarrassingly parallel.*

para tener una utilidad práctica real. El programa únicamente podrá operar sobre matrices cuadradas y de igual dimensión, que contengan elementos enteros. Adicionalmente, el programa no acepta datos de entrada, sino que genera las matrices de forma aleatoria.

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

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

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

```
$ ./unimatrix -d 150
dimension 150
calculado row 0
calculado row 1
calculado row 2
calculado row 3
...
calculado row 148
```

```
calculado row 149
wall clock time = 4.462505
```

Obsérvese la especificación del tamaño de la matriz por medio del parámetro **-d**. En esta corrida ejemplo se aprecia que el tiempo de ejecución es de 4.46 segundos.

3.2 Implementación paralela

3.2.1 Elección de organización

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

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

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

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

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

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

3.2.2 El algoritmo en paralelo

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

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

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

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

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

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

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

3.2.3 Implementación

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

MPI

El programa de multiplicación de matrices paralelo utilizando MPI se muestra en el apéndice (A.1.3).

El primer paso para utilizar MPI en un programa es inicializar el mecanismo de paso de mensajes. Esto se hace por medio de la llamada *MPI_Init* (línea 31). A continuación el proceso determina el tamaño de su comunicador (cuántos procesos

lo componen) con una llamada a *MPI_Comm_size* (línea 36) . El comunicador, o contexto de comunicaciones, es el grupo de trabajo básico de MPI; en general, un comunicador delimita el alcance de llamadas a funciones grupales, como *broadcasts* y *scatter/gather*. Esto facilita la organización y distribución de trabajo. En este caso, se utiliza el comunicador *MPI_COMM_WORLD*, que es el comunicador al que pertenecen inicialmente todos los procesos. Obviamente cada proceso puede posteriormente cambiar a otro comunicador, pero en este caso es suficiente el uso de *MPI_COMM_WORLD*.

Se determina también el rango del proceso dentro del comunicador (es decir, qué posición ocupa de entre los procesos que componen el comunicador) y el nombre del procesador en que se está ejecutando, llamando a *MPI_Comm_rank* y *MPI_Get_processor_name* (líneas 36 y 37). El rango es de particular utilidad para determinar cuáles renglones debe resolver cada proceso, según se describió en la sección (3.2.2).

A continuación se realiza la bifurcación de trabajo en el proceso; es decir, se designa un proceso para que asuma el papel de maestro, mientras los demás se configuran como esclavos. Basándose en el parámetro *localid*, el rango dentro del comunicador, a partir de la línea 74 y hasta la 169 se encuentra el trabajo que realiza el proceso maestro; de la línea 170 a la 262 se encuentran las instrucciones para los procesos esclavo.

En este caso el criterio de decisión es designar como maestro al proceso que tiene *localid* de 0. En MPI esta decisión es un tanto arbitraria, en realidad se puede escoger cualquier proceso como maestro, sin embargo el utilizar al proceso 0 es por convención. Esto obedece al hecho de que en MPI no se tiene el concepto inherente de "proceso padre". Como se verá en la sección 3.2.3, PVM sí cuenta con dicho concepto y existe un proceso que está identificado como padre de los demás.

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

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

Una vez enviadas las matrices, el proceso padre no realiza ninguna tarea de cálculo, únicamente espera a recibir los renglones completos por parte de los nodos. Esto se hace en un ciclo que contabiliza el número de renglones recibidos (línea 129), y en cada iteración realiza una llamada a *MPI_Recv* (línea 135). Esta llamada bloquea en espera de recepción de un mensaje de cualquier proceso del comunicador (parámetros *MPI_ANY_SOURCE* y *MPI_COMM_WORLD*).

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

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

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

Una vez teniendo la información necesaria, los nodos comienzan a resolver su porción de la matriz, guardando el resultado parcial de cada renglón en un arreglo y enviándolo al proceso maestro cuando se ha completado un renglón. Esto se hace por medio de una llamada a *MPI_Send* (línea 254). Esta función toma como parámetros la dirección y tamaño de la información a enviar, el tipo de datos (en este caso MPI_INT), el proceso destino (en este caso 0, el proceso padre), una bandera identificadora de mensaje y el comunicador al que se debe enviar el mensaje.

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

PVM

El programa de multiplicación de matrices paralelo utilizando PVM se muestra en el apéndice (A.1.4).

La inicialización de PVM es hasta cierto punto implícita; el llamar a cualquier función de PVM automáticamente enrola al proceso en la máquina virtual, si es

que no se encontraba en ella anteriormente. Por convención la primera llamada debe ser a la función *pvm_mytid* (línea 43), con la que también se obtiene el TID (identificador de tarea o *Task Identifier*) del proceso. A continuación se obtiene el TID del proceso padre con una llamada a *pvm_parent*. La necesidad de conocer el TID del proceso padre será obvia más adelante.

Entre la línea 80 y 112, el proceso padre (identificado por el hecho de que su proceso padre tiene el valor *PvmNoParent*, que se verifica en la línea 80) debe iniciar o engendrar[2] a los demás procesos que tomarán parte en el cálculo.

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

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

Los grupos no son una función intrínseca de PVM. La funcionalidad de grupos está provista por una biblioteca adicional, así como un demonio que arbitra la comunicación en grupos. La biblioteca y el demonio se incluyen con la distribución de PVM así que no se requiere configuración adicional para usarlos, sin embargo téngase en mente que la funcionalidad es adicional a las funciones básicas de PVM.

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

[2]del inglés *spawn*.

procesos esclavo de la línea 268 a la 384.

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

Las operaciones de comunicaciones en PVM involucran algunos pasos. El primero es inicializar el *buffer* de transmisión, con una llamada a *pvm_initsend* (línea 182). Posteriormente, se debe empacar el mensaje a enviar en dicho *buffer*. Esto representa un paso adicional, pero también permite empacar varios mensajes en el *buffer* y enviarlos todos simultáneamente. El empaque se realiza con una llamada a *pvm_pkint*[3] indicando la dirección y tamaño del dato a enviar. Finalmente, el envío se realiza con una llamada a la función correspondiente, en este caso se trata de un *broadcast* y la función es *pvm_bcast*, a la que le especificamos el grupo al que se envía y una etiqueta identificadora de mensaje. La etiqueta es en realidad un valor entero, aunque se acostumbra definir nombres significativos para las constantes de etiqueta (líneas 16-17).

Antes de transmitir la segunda matriz se debe llamar a *pvm_initsend* nuevamente, a fin de limpiar el *buffer*, ya que éste es acumulativo.

Finalmente el proceso padre espera la recepción de los resultados enviados por los procesos hijo. Utilizando una recepción de mensaje bloqueante (*pvm_recv*), el proceso espera en la línea 221 hasta recibir un mensaje de cualquier proceso, y con cualquier etiqueta de identificación (indicado por los valores -1 que especifican recepción tipo *wildcard*). Al recibir el mensaje (un renglón resuelto) se desempaca (*pvm_upkint*, línea 228), y se integra al resultado final.

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

A partir de la línea 170, los procesos esclavos (para los cuales la variable *pvmparent* es distinto de la constante *pvmNoParent*, indicando que sí tienen un proceso padre) asignan memoria para las matrices a operar y reciben sus valores por medio de *broadcast*. En PVM la recepción se hace con una llamada a *pvm_recv*, independientemente de si el origen fue un *broadcast* o un envío punto a punto. Nótese que en las llamadas a *pvm_recv* en las líneas 293 y 304 se especifica la etiqueta de

[3]PVM proporciona funciones para empacar diversos datos, que obligan a especificar el tipo de los mismos, logrando la conversión a tipos de datos opacos descrita en la sección (1.3.6), esencial para la operación en ambientes heterogéneos.

mensaje MATRIX_TAG, que corresponde a la etiqueta que se empleó al enviar el *broadcast*.

Los procesos deben desempacar las matrices recibidas por medio de la función *pvm_upkint*, después de lo cual determinan los renglones que deben calcular (líneas 317 a 325) y comienzan a realizar los cálculos.

Al completar cada renglón, lo envían al proceso padre, inicializando su *buffer* de mensajes (*pvm_initsend*, línea 371), empacando el resultado (*pvm_pkint*, línea 373) y finalmente enviándolo al proceso padre (*pvm_send*, línea 381). En la llamada a *pvm_send*, nótese el primer parámetro *myparent*, que indica enviar el mensaje al proceso padre.

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

3.2.4 Comparación de funciones: PVM vs. MPI

Tras haber implementado el algoritmo utilizando tanto PVM como MPI, se puede realizar una comparación de las características de ambas bibliotecas, desde el punto de vista del desarrollo de programas con ellas.

Se puede apreciar que MPI es ligeramente más compacto, requiriendo menos código que PVM. Obteniendo las líneas de código fuente efectivas[4] para cada archivo, se determinó que la implementación en MPI requiere 141 líneas de código, mientras que la implementación en PVM utiliza 192 líneas. Apoyando esta observación, se puede indicar que en general PVM es un poco más laborioso de utilizar que MPI, requiriendo en ocasiones más pasos para lograr el mismo resultado. Por ejemplo, la realización de un *broadcast* en MPI requiere únicamente una llamada a función, mientras que en PVM se requieren dos (una para empacar la información y otra para realizar el envío).

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

[4] Para este efecto se eliminaron los comentarios y formato de los archivos fuente en C, dejando el código lo más compacto posible.

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

Más allá de estos aspectos, se aprecia, tanto durante el desarrollo como por el hecho de que fue trivial el desarrollar la aplicación utilizando ambas bibliotecas, que ambas proporcionan las mismas primitivas básicas de paso de mensajes. Únicamente se emplearon las más elementales, como son *broadcast* y envío y recepción punto a punto, sin emplear mecanismos de sincronización ni llamadas de comunicaciones más complejas (como *scatter/gather*). Por otro lado, no se explotaron algunas de las características avanzadas de MPI, como manejo de archivos distribuido, mecanismos de control de comunicadores y comunicación entre ellos, y accesos directos a memoria, de forma que, si se requieren dichas características, quizá MPI es una mejor opción.

Con esta posible salvedad, cabe mencionar que ambas bibliotecas proporcionan aproximadamente la misma funcionalidad, de modo que la elección se puede dejar a criterio del programador; esto, desde luego, teniendo en mente que MPI presenta una implementación más limpia y mejor diseñada, y debería ser la elección primaria para proyectos nuevos.

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

Tarea	Función en MPI	Función en PVM
Inicialización de bibliotecas paralelas	MPI_Init	Implícito
Determinación de mi número de proceso	MPI_Comm_rank	pvm_mytid y pvm_joingroup
Determinación de número de procesos en mi grupo o comunicador	MPI_Comm_size	pvm_gsize
Obtención de nombre de procesador	MPI_Get_processor_name	no aplica
Inicialización de buffers de envío	no aplica	pvm_initsend
Empacado de mensajes	no aplica	pvm_pk*
Desempacado de mensajes	no aplica	pvm_upk*
Broadcast para matrices operando	MPI_Broadcast	pvm_bcast
Recepción de broadcast	MPI_Broadcast	pvm_recv
Recepción de renglones resueltos	MPI_Recv	pvm_recv
Envío de renglones resueltos	MPI_Send	pvm_send
Terminación de sesión paralela	MPI_Finalize	pvm_exit

3.2.5 Compilación

MPI

Uno de los binarios incluidos en la distribución de MPICH es un *front end* para compilar, que se encarga de agregar las rutas necesarias para los archivos incluidos y las bibliotecas, así como el ligado final del ejecutable. Este reside en **/usr/share/mpi/bin/** y se invoca de la siguiente manera:

```
mpicc matrix1.c
```

Esto genera el ejecutable **matrix1** y lo deja listo para su ejecución.

PVM

En el caso de PVM se requiere realizar la inclusión y ligado manualmente. Los archivos de encabezados **.h** se encuentran en **/usr/share/pvm3/include**; las bibliotecas para el ligado están en **/usr/share/pvm3/lib/LINUX**[5].

Adicionalmente se requiere indicar que se emplearán las bibliotecas *pvm3* y *gpvm3* (esta última se encarga del manejo de las funciones de grupo en PVM).

El comando usado para compilar el programa en PVM es como sigue:

```
gcc -o matrix2 matrix2.c -I /usr/share/pvm3/include \
    -L/usr/share/pvm3/lib/LINUX -lpvm3 -lgpvm3
```

A fin de facilitar la compilación de los programas de prueba, se creó un *Makefile* que automatiza las llamadas y parámetros de compilación, éste se incluye en el apéndice (C.3). En este caso simplemente se emplea el siguiente comando para generar todos los ejecutables:

```
make all
```

3.2.6 Ejecución

MPI

En sentido estricto, MPI no cuenta con una manera de ejecutar aplicaciones que utilicen sus funciones. Este detalle se deja a cada implementación particular. MPICH provee el comando **mpirun**, que se encarga de inicializar los mecanismos de comunicación necesarios, así como los programas a ejecutar, y realizar la corrida de los mismos.

mpirun toma un parámetro **-np** para indicar en cuántos procesadores se va a ejecutar el programa. Como se describió la sección (2.2.12), en un *cluster* se tiene una lista de máquinas, cada una de las cuales cuenta como un procesador, de modo que si se especifica **-np 5** se indica que se ejecutará el programa de MPI utilizando 5 nodos. Los procesos se inician en el orden que se especifica en el archivo de lista de nodos.

[5] en este caso, ya que se está utilizando la arquitectura LINUX; si se emplean diversas arquitecturas, el nombre del último subdirectorio cambia de acuerdo a ello.

Entonces, para ejecutar el programa de multiplicación de matrices en MPI, se debe indicar el siguiente comando:

```
$ mpirun -np 17 ./matrix1 -d 650
```

Este comando especifica ejecutar el programa utilizando 17 nodos, con una matriz de dimensión 650 × 650. El comando **mpirun** realiza las tareas de inicialización y arranque del número de procesos necesarios.

La ejecución del programa en MPI fue relativamente limpia. No se proporciona un mecanismo de control para los procesos que se ejecutan en paralelo, lo cual supone una desventaja, aunque por otro lado, en caso de complicaciones, el detener el proceso inicial (la invocación de **mpirun**) automáticamente elimina los procesos generados en los nodos.

En caso de ocurrir algún error, MPI envía mensajes que en ocasiones no resultan obvios pero permiten suponer que ha ocurrido alguna complicación, en cuyo caso se puede detener el proceso restaurando el estado del sistema.

PVM

Antes de poder iniciar un programa en PVM, debe inicializarse y configurarse la máquina virtual, para que al momento de realizar la ejecución, todos los nodos requeridos formen parte de ella.

La configuración de la máquina virtual se describe en la sección (2.2.11).

Una vez realizado esto, basta invocar al programa escrito utilizando PVM. Cabe observar que, ya que el programa se invoca directamente, y éste es el que se encarga de iniciar los procesos, se requirió un parámetro adicional para indicar al programa cuántos procesos se van a utilizar (parámetro **-t**).

```
$ ./matrix2 -d 650 -t 17
```

Este comando ejecuta el programa utilizando 17 nodos, o tareas, con una matriz de dimensión 650 × 650.

El uso de la máquina virtual supone una versatilidad que no está presente con MPI, pues además de permitir controlar precisamente la configuración de la máquina virtual, la consola contiene facilidades para administrar los procesos que corren

en la misma (comando **reset** para restaurar el estado de la máquina virtual, **ps** para visualizar el estado de los procesos, y algunos otros).

A pesar de ello, el uso de PVM resulta un poco menos limpio. El uso de la máquina virtual supone un paso adicional y exige al usuario un conocimiento más íntimo de la configuración de los nodos para poder agregarlos a la máquina virtual, tarea que en MPI se deja a la manipulación de archivos de configuración por parte del administrador.

Además, la terminación de procesos en PVM no es tan implícita como en MPI, y por lo tanto resulta menos limpia y un tanto inconveniente. Si algún proceso llega a fallar, los demás quedan en un estado indefinido, ocupando recursos en los nodos, y se hace necesario acceder a la consola de PVM y eliminar los procesos manualmente. En ocasiones, aún al concluir exitosamente la ejecución, algunos procesos quedan "colgados" requiriendo intervención del usuario para restaurar el estado de la máquina virtual.

En general la ejecución de procesos en PVM resulta hasta cierto punto más laboriosa y menos confiable que el que se observó con MPI.

Tanto con PVM como con MPI, la salida del comando debe ser similar a la presentada en la sección (3.1.2), ya que lo único que cambia es la sección donde se realizan los cálculos.

Número real de procesos iniciados

Es importante mencionar un aspecto del comportamiento del arranque de los programas en PVM y MPI que influye directamente en el número de procesos a iniciar para el cálculo.

En el caso de MPI, el parámetro **-np** especifica exactamente el número de procesos a iniciar. Si se especifica el valor de 5, por ejemplo, se iniciarán cinco procesos, uno de los cuales será el maestro y no participa en los cálculos directamente. Unicamente los cuatro restantes efectuarán cálculos.

En el caso de PVM, la invocación del programa inicia el primer proceso bajo la máquina virtual. A éste se le especifica cuántos procesos adicionales se deben iniciar, con el parámetro **-t**. De modo que si se especifica el valor de 5, primero se tendrá la instancia inicial del proceso, que a su vez engendrará otras cinco instancias, teniendo un total de seis; de estos, uno será el proceso padre, mientras que los

otros cinco efectúan cálculos.

Es de importancia tener en cuenta estas características para saber exactamente el número de procesos que se tendrán.

3.2.7 Realización de pruebas

Una vez contando con los programas de multiplicación de matrices, en uniprocesador, y en versión paralela utilizando PVM y MPI, se realizaron una serie de pruebas a fin de poder tener una idea del desempeño y características del *cluster*.

En general existen dos parámetros que afectan directamente el rendimiento que se puede esperar del equipo: el tamaño del problema, en este caso las dimensiones de las matrices a multiplicar; y el número de elementos de procesamiento, o nodos, que participan en el cálculo.

A fin de poder observar el comportamiento del *cluster* al variar estos dos parámetros, se decidió realizar pruebas con varios tamaños de matriz, variando el número de nodos, y tomando el tiempo empleado en la realización de los cálculos. Estos tiempos se comparan y grafican para permitir un análisis visual de los resultados.

Se emplearon matrices de 20×20, 50×50, 100×100, 200×200, 400×400, 600×600 y 650×650 elementos. Es interesante notar el límite superior (650×650). Este límite está dictado por la memoria disponible en los nodos; para aquellos que cuentan con únicamente 12 MB de memoria, no es posible el manejo de una matriz de dimensiones superiores.

Para cada tamaño de matriz, se tomaron los tiempos de solución con la versión uniprocesador (valores que en la tabla aparecen como "1 nodo") y con la versión paralela utilizando MPI, desde 2 hasta 17 nodos. De esta manera se puede comparar directamente la diferencia de desempeño entre el uso de un solo procesador, sin recurrir a las bibliotecas paralelas, y el uso de varios nodos en el cálculo.

Para cada combinación de tamaño de matriz y número de nodos, se realizaron tres corridas del programa, descartando la primera y promediando las dos restantes para obtener el valor final. Esto se hizo para evitar posibles "picos" o valores extraños, que podrían presentarse tomando sólo una lectura; el descartar la primera lectura permite eliminar posibles variaciones en el tiempo de ejecución por los mecanismos de *caching* de disco con que cuentan los nodos, y que pueden influir

al momento de ejecutar un proceso diferente.

La realización de las pruebas, así como la recolección y cálculo de resultados, se efectuaron por medio de un programa que automatiza la generación de parámetros tanto de número de nodos como de dimensión de las matrices, efectuando las corridas necesarias, generando automáticamente las tablas de resultados. De esta manera la realización de pruebas sólo requiere intervención humana cuando ocurre algún error. Este programa, escrito en lenguaje *Perl*[6] se incluye en el apéndice (C.4).

Dada la cantidad de corridas que debieron efectuarse para la realización de las pruebas, estas abarcaron un período de aproximadamente dos días, y los resultados se muestran a continuación.

3.2.8 Tablas de resultados

La primera tabla muestra los resultados de tiempo de ejecución, desde 1 hasta 17 nodos, con tamaños de matriz desde 20 × 20 hasta 650 × 650. Para 1 nodo, el resultado es realmente el entregado por la multiplicación de matrices en uniprocesador. De 2 nodos en adelante, se empleó el programa de multiplicación paralela con MPI. Todos los tiempos están en segundos.

[6] *Practical Extraction and Report Language*, lenguage práctico de extración y reportes.

Nodos	Tamaño de la matriz						
	20 × 20	50 × 50	100 × 100	200 × 200	400 × 400	600 × 600	650 × 650
1	0.0251	0.1286	0.8345	6.3813	58.0333	204.1593	264.1158
2	0.1349	0.1543	1.0468	7.8939	82.3824	300.8263	389.3297
3	0.0867	0.2728	1.6193	12.1640	100.5565	344.7163	446.3945
4	0.1053	0.2489	1.2775	8.8710	69.5401	235.8056	306.2163
5	0.1066	0.2106	1.0828	7.1068	54.0451	181.9883	233.2845
6	0.1056	0.2108	1.0222	6.2718	46.2985	150.9391	193.7154
7	0.0970	0.2474	1.0840	5.9078	40.2734	131.2392	167.5479
8	0.1456	0.3274	1.3211	7.1489	46.2516	154.6756	205.3675
9	0.1329	0.3531	1.1900	6.8588	42.8355	137.5202	174.3605
10	0.1610	0.3430	1.2580	6.7173	40.0890	128.6568	165.1221
11	0.1678	0.6157	1.4059	6.7373	38.9844	121.4581	153.9762
12	0.3596	0.6253	1.6434	7.2128	40.9458	124.0780	149.1919
13	0.3562	0.4338	1.6939	6.7146	37.7014	112.9040	144.1662
14	0.3654	0.4763	1.7718	8.4932	43.6856	111.4639	138.7395
15	0.5334	0.5373	1.6202	7.9568	41.6083	122.2964	144.6377
16	0.3766	0.8129	1.7602	8.1674	37.3292	106.0394	133.4445
17	0.3751	0.5270	1.7394	7.9168	38.2767	104.4315	131.8206

La segunda tabla muestra los tiempos de ejecución, utilizando 5 y 17 nodos, y tamaños de matriz de 50×50 y 650×650. En este caso se muestran los resultados empleando el programa de multiplicación paralela con PVM.

Nodos	Tamaño de la matriz	
	50 × 50	650 × 650
5	0.4915	272.3711
17	0.7166	151.3273

Comparando estos resultados con los obtenidos en la tabla anterior, se puede observar que los tiempos del programa escrito con PVM son ligeramente mayores, indicando un menor desempeño, que posiblemente se pueda atribuir a una implementación menos eficiente de los mecanismos de comunicación de red de PVM; esta observación se basa en el comportamiento de la red, que al ejecutar el programa en PVM presenta un gran número de colisiones y aparente saturación, lo cual es visible en el comportamiento de los indicadores en los concentradores. Aún cuando la diferencia es menor, se debe tener en cuenta su existencia al momento de

seleccionar la biblioteca de paso de mensajes a utilizar. Estos resultados refuerzan la recomendación de utilizar MPI en la medida de lo posible.

Posteriormente se graficaron los resultados para su análisis, información que se muestra a continuación.

3.2.9 Gráficas

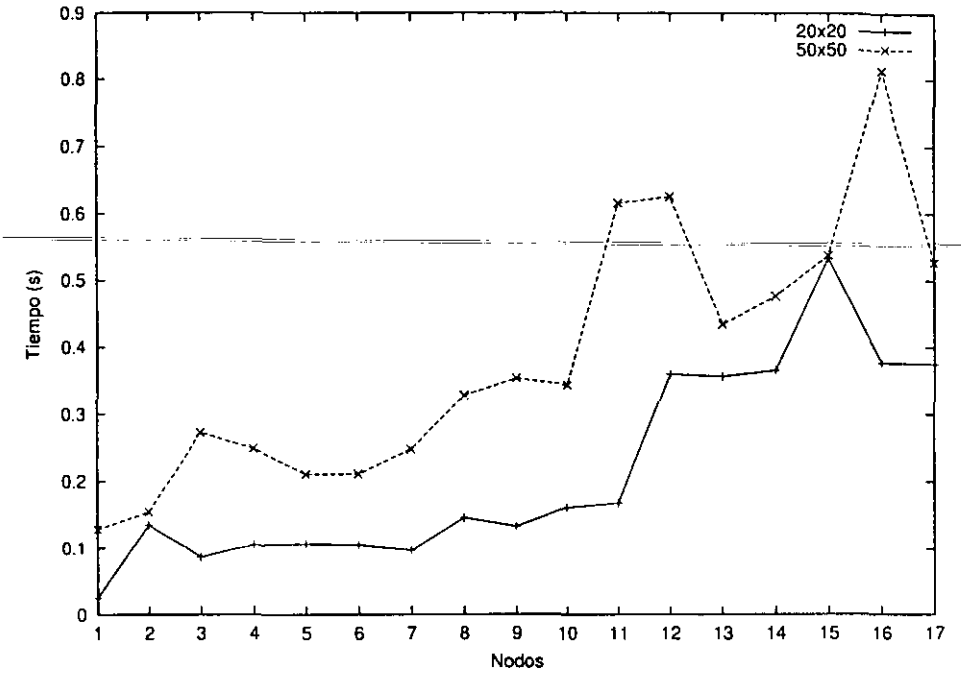

Figura 3.1: Multiplicación de matrices: 20×20 a 50×50

En la figura (3.1) se muestran los resultados para las matrices consideradas como "pequeñas", las de 20×20 y 50×50 elementos.

Se puede observar que, en ambas instancias, los mejores tiempos de cálculo se logran utilizando únicamente un nodo (versión uniprocesador). Más aún, en general, al incrementar el número de nodos el rendimiento empeora, lo cual se nota en el incremento del tiempo de cálculo.

Este comportamiento es esperado en problemas pequeños o de rápida resolución. En este caso, la implementación óptima es la uniprocesador. Debido a que la cantidad de operaciones y datos es pequeña, el tiempo de cálculo, aún en la versión

uniprocesador, es de algunas centésimas de segundo. El introducir más nodos a la operación es perjudicial, ya que el tiempo de arranque de los mecanismos de paralelización, sincronización y transmisión de datos, niegan la ventaja de tener más elementos de procesamiento. Esto es notorio por el hecho de que, por el contrario, el tiempo se incrementa al agregar más nodos.

Las fluctuaciones más o menos violentas del desempeño para los problemas pequeños se deben, precisamente, a la poca duración del cálculo, de forma que se introducen las variaciones observadas debido a características del arranque del proceso en el sistema, el comportamiento aleatorio de las comunicaciones en red por el arranque de los procesos remotos, y otros factores.

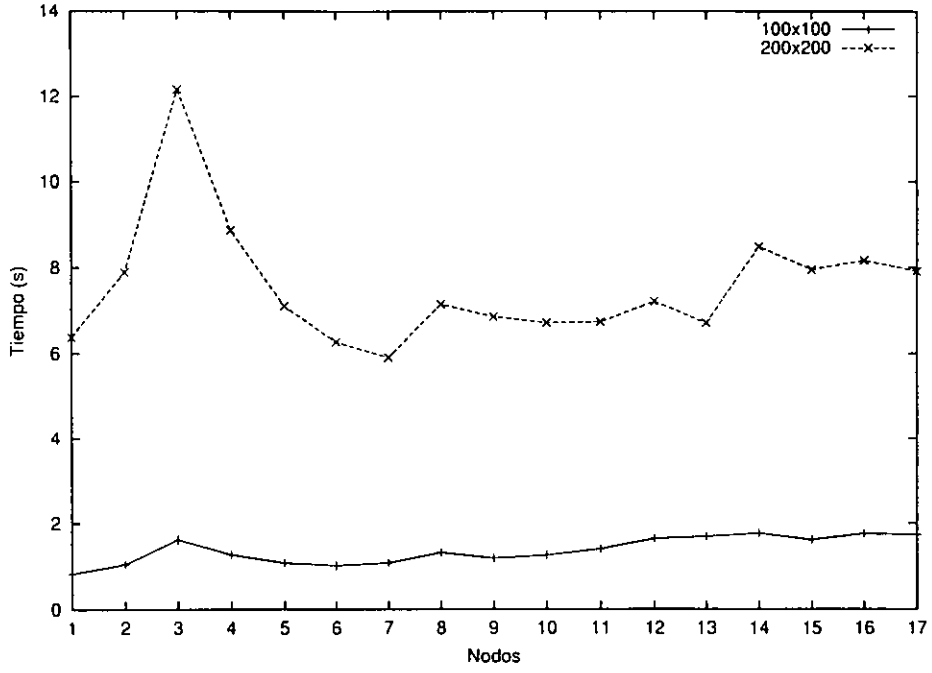

Figura 3.2: Multiplicación de matrices: 100 × 100 a 200 × 200

En la figura (3.2) se muestran los resultados para las matrices consideradas como "medianas", las de 100 × 100 y 200 × 200 elementos.

La matriz de 100 × 100 aún exhibe el comportamiento observado para las matrices pequeñas. El comportamiento general es de empeoramiento del rendimiento al agregar más nodos, si bien se aprecia que entre 3 y 7 nodos el rendimiento me-

jora, siendo el mejor tiempo de 5.9 segundos con 7 nodos. Esto sugiere que una matriz de este tamaño ya comienza a presentar un beneficio al utilizar múltiples elementos de procesamiento, sin embargo, nuevamente el beneficio se niega al ir más allá de 7 nodos. Esto implica que, para este tamaño de matriz, 7 nodos son los más adecuados para realizar el cálculo.

La matríz de 200 × 200 presenta un comportamiento similar, donde el uso de 6 y 7 nodos brinda un tiempo de ejecución menor que al utilizar un solo procesador.

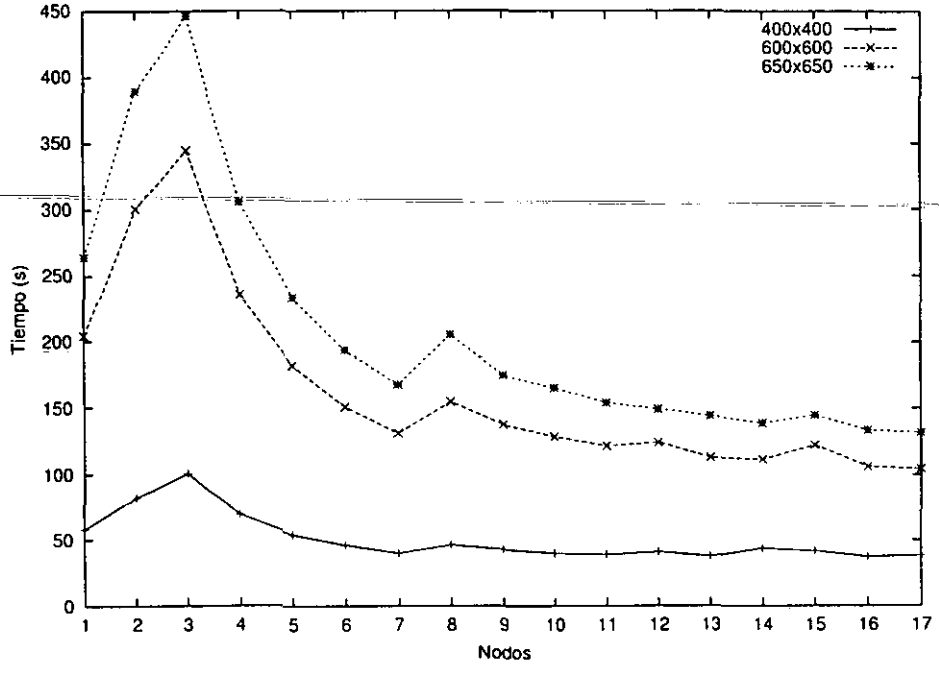

Figura 3.3: Multiplicación de matrices: 400 × 400 a 650 × 650

En la figura (3.3) se muestran los resultados para las matrices consideradas como "grandes", las de 400 × 400 a 650 × 650 elementos.

Las tres matrices presentan una gráfica de comportamiento similar. Se observa que el rendimiento empeora hasta alcanzar tres nodos, a partir de este momento se experimenta una tendencia a mejorar. En la matriz de 400 × 400 el rendimiento se estabiliza a partir de 10 nodos, mientras que en las de 600 × 600 y 650 × 650 el rendimiento sigue mejorando de modo que el mejor tiempo se obtiene utilizando el máximo de 17 nodos.

3.2.10 Discusión de resultados

Además de las observaciones presentadas, existen algunos factores que se observan por igual en todas las gráficas mostradas.

En la figura (3.3) se observa una reducción del desempeño entre 7 y 8 nodos. En general ésta y otras fluctuaciones en la forma esperada de la curva se deben al hecho de que los equipos utilizados en el *cluster* no tienen un rendimiento similar. Por lo tanto, al introducir los equipos 486/50, que tienen un rendimiento menor a los 486/66, se genera un "cuello de botella" donde el equipo más lento es el que determina la terminación del cálculo.

Al agregar más elementos de procesamiento, sin embargo, esta tendencia se revierte y como se comentó, con el mayor número de nodos posible se obtiene el mejor rendimiento para problemas grandes.

Esto se debe hasta cierto punto al algoritmo de asignación de trabajo empleado en la multiplicación de matrices, y descrito en la sección (3.2.2). Como se observa, la asignación de trabajo busca ser lo más equitativa posible, lo cual es adecuado en una máquina donde todos los nodos fueran exactamente iguales. Sin embargo en el sistema que se tiene, esto no se cumple, y asignar la misma cantidad de trabajo a un nodo más lento efectivamente lo convierte en un cuello de botella.

Esta situación puede remediarse si se implementa un mecanismo de asignación de trabajo diferente; en particular, la asignación del trabajo pendiente en unidades más pequeñas, y únicamente a los nodos que estén disponibles. De esta manera, por ejemplo, cada nodo resolvería un renglón a la vez, lo entregaría al servidor central, y éste indicaría al nodo el siguiente renglón que debe resolver.

Este es un ejemplo de las complicaciones que pueden surgir en un *cluster* donde el rendimiento por nodo es distinto, y da una idea de por qué normalmente se busca que los nodos de un *Beowulf* sean lo más parecidos posible, a fin de evitar esquemas de asignación de trabajo potencialmente más complicados.

A nivel general, de los resultados se observa que para cada tamaño de problema, puede o no resultar conveniente el uso de un *cluster*; asimismo, el número de nodos que se deben dedicar al proceso puede influir en el desempeño. En general para problemas grandes, que son los de más interés en un *cluster*, se obtiene beneficio dedicando el número máximo de nodos, pero a medida que el problema se reduce, el mejor rendimiento se alcanza con menos nodos. En el extremo de esta

lógica se observa que para problemas muy pequeños el uso de un solo procesador da un mejor rendimiento.

Por lo tanto es importante evaluar la dimensión del problema a resolver, antes de decidir si se obtendrá un beneficio utilizando un *cluster*.

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

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

El uso del *benchmark* de LINPACK fue introducido en 1979 por Jack Dongarra, investigador de la Universidad de Tennessee en Knoxville, como parte del paquete LINPACK, un juego de bibliotecas matemáticas en FORTRAN para solución de sistemas de ecuaciones lineales. En general, el *benchmark* realiza la resolución de un sistema de ecuaciones generado aleatoriamente, expresado como una matriz de coeficientes que en la computadora están representados con números de punto flotante, normalmente a una precisión de 64 bits. El paso crucial de esta solución es la descomposición LU con pivoteo parcial de la matriz de coeficientes. Obtener la solución requiere $2/3n^3 + 2n^2$ operaciones de punto flotante, donde n es la dimensión de la matriz. Normalmente se emplean valores de $n = 100$ y $n = 1000$.

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

LINPACK fue originalmente implementado en lenguaje FORTRAN para máquinas uniprocesador, vectoriales y SMP. A medida que los equipos MPP fueron

cobrando auge, se hizo necesario el contar con una manera de comparar su desempeño, de forma que se desarrolló el *benchmark* HPL (*High Performance LINPACK*). HPL es una implementación del *benchmark* LINPACK escrita en lenguaje C, que puede ejecutarse en cualquier equipo que cuente con una implementación de MPI, lo cual incluye a prácticamente cualquier equipo MPP comercial y, desde luego, *clusters* tipo *Beowulf*.

HPL es el *benchmark* utilizado para medir el desempeño de las computadoras más rápidas del mundo, gracias a su portabilidad, su dependencia en otras bibliotecas que están ampliamente disponibles, particularmente MPI y BLAS[7], y la capacidad de alterar fácilmente los parámetros de cálculo a fin de determinar la configuración óptima para obtener el mejor rendimiento.

Se consideró de interés el realizar pruebas con HPL en el *Beowulf*, básicamente para comparar el incremento de rendimiento al utilizar toda la capacidad del mismo. Desde luego, no se espera que el rendimiento alcance los niveles presentados por las computadoras más rápidas del mundo; como se mencionó en la sección (1.1), la computadora más rápida alcanza un rendimiento sostenido de 7226 GFLOPS. Sin embargo el comparar ambos equipos con el mismo mecanismo de medición puede resultar interesante.

3.3.1 Requisitos previos

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

3.3.2 Instalación

Biblioteca ATLAS

Primeramente se requiere instalar la biblioteca BLAS o alguna que proporcione funcionalidad similar. BLAS (*Basic Linear Algebra System*) proporciona funciones de álgebra lineal en FORTRAN, definiendo para ello una API que otras bibliotecas más modernas y eficientes también han implementado. En este caso se

[7] *Basic Linear Algebra System*, una biblioteca de cálculos algebraicos portable.

seleccionó la biblioteca ATLAS (*Automatically Tuned Linear Algebra Software*), desarrollada por la Universidad de Tennessee en Knoxville. ATLAS está escrito en C, y es un *software* que al momento de compilar, determina automáticamente las características de la arquitectura del equipo y genera bibliotecas utilizando los algoritmos más eficientes para las funciones que proporciona. Su instalación es sencilla pero, debido al proceso de pruebas para determinar los parámetros que resultan en un mejor rendimiento, suele tomar bastante tiempo.

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

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

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

```
# make config
```

En este paso se deben responder algunas preguntas sobre el sistema con que se cuenta. En todas las preguntas se puede dar la respuesta predeterminada, o presionar ENTER, salvo en las siguientes:

Tipo de máquina, aquí se debe seleccionar el procesador correcto, en este caso la opción 6 (Pentium).

```
Enter your machine type:
   1. Other/UNKNOWN

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

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

```
# make install arch=Linux_P5
```

Nótese que el valor para el parámetro **arch** se compone del nombre de la arquitectura del sistema operativo (en este caso Linux) y del procesador con que se cuenta (en este caso P5 o Pentium).

Esto inicia la autoconfiguración, compilación e instalación de ATLAS. En el servidor central del *cluster*, este proceso tomó alrededor de una hora.

HPL

El paquete de HPL se obtiene de *http://www.netlib.org/benchmark/hpl/hpl.tgz*.

Una vez contando con este archivo se descompacta y se pasa al subdirectorio HPL con los siguientes comandos:

```
# tar -zxvf hpl.tar.gz
# cd hpl
```

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

```
# cp setup/Make.Linux_PII_CBLAS_gm ./
```

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

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

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

```
# make  arch=Linux_PII_CBLAS_gm
```

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

3.3.3 Realización de pruebas

Configuración

La configuración de parámetros para HPL, como el tamaño del problema a resolver, las dimensiones de la matriz de procesos, y otros, se definen en el archivo **HPL.dat**. El archivo empleado para la realización de las pruebas se muestra en el apéndice (B.3).

Básicamente, el archivo permite especificar distintos juegos de parámetros a experimentar; por ejemplo, permite definir varios tamaños de problema a probar, así como número y organización de los procesos que participarán en el cálculo. Se prueban todas las combinaciones posibles. El archivo mostrado especifica probar problemas de dimensión 650 y 1000, con un proceso, 16 procesos (organizados en arreglos de 4×4 y 8×2) y 17 procesos (en un arreglo de 1×17).

Ejecución

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

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

El número de procesos debe ser suficiente para incluir al número dado por las matrices de proceso.

La prueba generará los resultados para los problemas especificados, enviando bloques de información como sigue:

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

```
||Ax-b||_oo / ( eps * ||A||_1  * N        ) =        0.9621179 ..... PASSED
||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  ) =        0.0233519 ..... PASSED
||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =        0.0056416 ..... PASSED
```

Los datos de interés son N, NB, P y Q que indican las características del problema a resolver; el tiempo de ejecución, y el desempeño en GFLOPS.

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

Dimensión	Nodos	P	Q	Tiempo (s)	GFLOPS
650	1	1	1	4.97	3.698e-02
650	16	2	8	19.56	9.392e-03
650	16	4	4	22.69	8.098e-03
650	17	1	17	31.95	5.750e-03
1000	1	1	1	127.73	5.231e-03
1000	16	2	8	45.04	1.484e-02
1000	16	4	4	49.47	1.351e-02
1000	17	1	17	73.20	9.128e-03

3.3.4 Discusión de resultados

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

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

Para la matriz de coeficientes de 650×650, se aprecia que el mejor tiempo y mejor desempeño se obtienen con un solo procesador. Esto sugiere, nuevamente, que la complejidad de la realización del cálculo a este tamaño de problema niega la

ventaja de utilizar más nodos. El mejor desempeño para este tamaño de problema es de 36.98 MFLOPS. Nótese que, entre las soluciones que emplearon 16 y 17 nodos, la más rápida fue la que utiliza una matriz de procesos de 8×2, siendo incluso más rápida que emplear 17 nodos. Esto permite ver que la correcta organización de la matriz de procesos es importante para tener un mejor desempeño, y da la idea de que, para esta configuración particular, la matriz de 8×2 es la que dará mejor rendimiento.

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

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

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

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

3.4 Uso del *cluster* en aplicaciones reales

Una vez teniendo una idea de las mejoras de desempeño que representa el uso del *cluster* sobre un equipo uniprocesador, es factible comenzar a utilizarlo en la resolución de una variedad de problemas reales. Esto incluye problemas de cálculos de

simulación por ecuaciones diferenciales y elemento finito, como se mencionó en la sección (1.1.1), que pueden aplicarse a distintas disciplinas, tales como ciencias nucleares, meteorología, astronomía, química, electrónica, diseño, y prácticamente cualquier disciplina que pueda requerir la observación del comportamiento de algún sistema físico, que sea factible de simularse con las técnicas descritas en la sección (1.1.1) y que se pueda beneficiar de la capacidad de un *cluster* para realizar grandes cantidades de cálculos.

Algunas técnicas de criptografía pueden beneficiarse del uso de un equipo con estas capacidades. Técnicas como el ataque por "fuerza bruta" a un mensaje encriptado[8] pueden realizarse en menor tiempo contando con un equipo más poderoso, como un *cluster*; esta clase de ataques son otro problema "vergonzosamente paralelizable".

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

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

En general, el contar con un equipo más rápido, como un *cluster*, abre las puertas hacia nuevas aplicaciones de computación. Sin embargo, las características y limitaciones que se han mencionado y observado obligan a ejercer un juicio cuidadoso al momento de elegir un *cluster* como plataforma para correr una aplicación determinada.

[8]En esta técnica se prueban, una por una, todas las llaves de desencriptación posibles, hasta encontrar aquella que decodifica el mensaje que se desea obtener.

3.4.1 *Ray tracing*: una aplicación real

En la sección (3.4) se mencionaron las posibles aplicaciones de un *cluster*, una de las cuales es el trazado de imágenes por computadora. Una de las técnicas más utilizadas para generar imagen por computadora de alta calidad es el *ray tracing* (trazado o seguimiento de rayos). Se seleccionó el *ray tracing* como la aplicación real para la cual se puede utilizar el *cluster*, ya que cumple con una serie de características que la hacen interesante, útil y atractiva.

3.4.2 La técnica de *ray tracing*

El algoritmo de *ray tracing* fue propuesto por primera vez en 1968, por Appel[9], aunque inicialmente su uso era únicamente en la detección de superficies ocultas.

El *ray tracing* es actualmente la simulación más completa de un modelo de iluminación y reflexión por computadora. El algoritmo básicamente supone que un observador ve un punto en una superficie como resultado de la interacción de dicha superficie en ese punto con rayos que emanan de otros puntos de la escena.

El *ray tracing* va más allá que otras técnicas, pues en vez de considerar únicamente la interacción de los puntos de la superficie con la iluminación directa de las fuentes de luz, se toma en cuenta que en general, un rayo de luz puede alcanzar la superficie indirectamente por reflexión en otras superficies, transmisión a través de objetos parcialmente transparentes, o una combinación de ambos. Esto se denomina iluminación global, pues la luz se origina del ambiente de la escena, en vez de hacerlo por interacción local de la superficie con la iluminación directa de las fuentes de luz.

Este método tiene algunas desventajas, la más importante de las cuales es su gran requerimiento de procesamiento, tomando tiempos muy largos (horas o días) para calcular imágenes.

Sin embargo, su ventaja más significativa a nivel algoritmo de graficación es que representa una solución parcial al problema de la iluminación global, combinando en un solo modelo la remoción de superficies ocultas, el sombreado debido a iluminación tanto directa como global, y el cálculo de sombras. Una ventaja

[9] Appel, A, *Some techniques for machine rendering of solids*, AFIPS conference proceedings 32, 37-45.

adicional es que el algoritmo es muy fácilmente paralelizable, como se verá más adelante.

El algoritmo de *ray tracing* trabaja completamente en el espacio de objetos. En un punto del plano de imagen, se obtienen las superficies visibles, así como el color e intensidad en el punto, trazando un rayo hacia atrás, desde el ojo o cámara, a través del punto de interés, y hacia la escena.

Si el rayo intersecta un objeto, entonces se realizan cálculos locales para determinar el color que resulte de la iluminación directa en el punto. Si es parcialmente reflejante, parcialmente transparente, o ambos, el color del punto en el plano de la imagen incluirá una contribución de los rayos reflejados y transmitidos. Estos deben trazarse hacia atrás a partir del punto de intersección para descubrir sus contribuciones. Determinar el color de estos rayos puede a su vez requerir el trazado de más rayos en intersecciones con objetos. Para determinar completamente el color del punto original en el plano visible, este juego de rayos deben trazarse hacia atrás dentro de la escena. El trazado de un rayo en particular termina cuando no se intersectan más objetos (en cuyo caso se asigna al rayo un color de fondo), o cuando el rayo está separado del observador por un número tal de intersecciones que su contribución de color se considera despreciable.

Se lanza un rayo desde el observador hacia cada punto a definir dentro del plano de imagen, pasando por el centro de dicho punto o *pixel*. Esto significa que la escena será muestreada en el espacio de objeto por rayos infinitamente delgados. En el proceso se tiene coherencia espacial de cero, es decir, todos los rayos se trazan independientemente sin utilizar información de rayos vecinos). Esto produce *aliasing*[10], lo cual representa una desventaja; por otro lado, hace que la implementación paralela sea trivial, y el proceso fácilmente distribuible.

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

Si bien, como ya se describió, el algoritmo de *ray tracing* es relativamente sen-

[10]Fenómeno de distorsión de información causado por muestrear a una frecuencia más baja de lo requerido; en graficación, la consecuencia es que el despliegue tiene una apariencia escalonada, ya que el proceso de muestreo digitaliza puntos coordenados de un objeto mapeándolos a posiciones de *pixel* enteras y discretas.

cillo a nivel básico, en realidad éste puede complicarse, según se incrementa la cantidad de comportamientos visuales que el programa debe manejar, sus capacidades de manipulación de objetos, y la eficiencia de la implementación.

Ya que el desarrollar un programa de *ray tracing* se considera fuera del alcance y enfoque de este proyecto, así como por consideraciones de tiempo, se optó por utilizar un paquete existente de *ray tracing* y buscar adaptarlo a una máquina paralela para mejorar su rendimiento.

3.4.3 Selección de *software*

Existe una gran variedad de programas para *ray tracing* disponibles, tanto libre como comercialmente. Por la naturaleza del proyecto se prefiere seleccionar algún paquete que sea gratuito y, preferentemente, cuyo código fuente esté disponible.

Ya que quizá el paquete de *ray tracing* más conocido es el POV-Ray, se investigó si el mismo cumple con las características requeridas.

3.4.4 POV-Ray

POV-Ray (*Persistence of Vision Raytracer*) es un programa para *ray tracing* que ha estado en desarrollo durante varios años. Como tal, es una herramienta confiable, eficiente y robusta, cuyo código fuente está disponible, si bien no bajo una licencia libre[11]. A pesar de ello, el uso de este código fuente no representa problemas ya que la versión modificada para uso en máquinas paralelas no se va a distribuir bajo términos que contravengan la licencia de POV-Ray. El punto central de distribución e información sobre POV-Ray es el sitio en internet *http://www.povray.org*.

POV-Ray cuenta con un lenguaje de descripción de escenas fácil de utilizar. La geometría y características de materiales, texturas, iluminación y cámaras de la escena a trazar se describen por medio de este lenguaje, posteriormente POV-Ray se encarga de interpretar la descripción y generar la imagen a partir de dicha información. Gracias a su popularidad y la facilida de uso del lenguaje de descripción, existen grandes cantidades de escenas disponibles públicamente, así como

[11] La licencia de POV-Ray impone restricciones sobre la utilización del código fuente y distribución de versiones modificadas que la hacen inaceptable bajo la definición de *software* libre oficial de la *Free Software Foundation*.

descripciones de objetos que pueden utilizarse para componer nuevas escenas.

POV-Ray proporciona una serie de primitivas básicas como esferas, cajas, cilindros, cuerpos cuádráticos, conos, triángulos y planos; figuras más complejas como toroides, curvas cuárticas, texto, texturas fractales, prismas, polígonos, superficies de revolución y algunas otras. Adicionalmente éstas se pueden combinar por medio de geometría sólida constructiva (*Constructive Solid Geometry*, o CSG) para formar nuevas figuras. A estas figuras pueden asignarse patrones y propiedades de materiales que confieren a éstos texturas. A fin de poder iluminar y visualizar estas escenas, el programa proporciona varios tipos de cámaras, entre ellos una cámara panorámica, una cámara con perspectiva, una "ojo de pescado", ortográfica y otras; fuentes luminosas cilíndricas, cónicas o de reflector y de área; se puede emplear iluminación interdifusa para obtener efectos más reales en áreas cerradas o de interiores, efectos atmosféricos como niebla, neblina y arcoiris, modelos de partículas para efectos como nubes, polvo o fuego, y sombreado y reflejos Phong y especulares.

POV-Ray puede entregar el trazado de la imagen con una profundidad de color hasta de 48 bits, en formatos TGA, PNG[12] y PPM[13], entre otros.

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

3.4.5 Paralelizando POV-Ray

El código fuente de POV-Ray abarca varios megabytes de espacio en disco, y es complejo y extenso. Sin embargo, se asume que cumple con las características generales del algoritmo de *ray tracing*, recordando en particular que dicho algoritmo es fácilmente paralelizable.

Se realizaron investigaciones en internet para localizar información que pudiera ser de ayuda en la modificación de POV-Ray para máquinas paralelas. Interesantemente, se encontró que ya existen dichas modificaciones al código fuente. Éstas

[12] *Portable Network Graphics.*

[13] *Portable Pixmap.*

se distribuyen de manera separada, como "parches" al código fuente de POV-Ray. Más aún, existen dos variantes de este código, implementadas en PVM y MPI respectivamente.

PVMPOV[14], escrito por Andreas Dilger, es el primero de estos paquetes. Se trata de modificaciones al código que proporcionan a POV-Ray "...la capacidad de distribuir el trazado a través de múltiples sistemas heterogéneos. El vehículo de la implementación es el sistema PVM ...".

La implementación de PVMPOV se centra en un esquema maestro-esclavo, correspondiendo exactamente a la organización maestro-esclavo descrita en la sección (3.2.1). El proceso maestro divide la imagen en pequeños bloques, que se asignan a los esclavos. Cuando éstos completan el trazado de sus bloques, se envían de regreso al maestro para combinarlos y formar la imagen terminada. El maestro no realiza ninguna tarea de trazado.

PVMPOV es una opción para obtener la versión paralela de POV-Ray. No obstante, se consideró que una implementación en PVM puede no resultar la más conveniente. Las razones más importantes para buscar otras opciones fueron el menor rendimiento que se observó con la implementación en PVM del programa de multiplicación de matrices, así como los problemas de estabilidad y confiabilidad exhibidos por el mismo.

Afortunadamente, también existe una implementación paralela de POV-Ray utilizando MPI. MPI-POVRay[15], desarrollado por Leon Verrall, "...consta de un parche al programa POV-Ray que distribuye el trabajo entre un número de elementos de procesamiento. La comunicación entre los elementos se logra por medio de paso de mensajes con MPI".

Como un testigo más de la compatibilidad funcional entre diferentes APIs de paso de mensajes, la página de Leon Verrall detalla que "...este parche está en gran parte basado en el excelente trabajo de Andreas Dilger en PVMPOV. He reimplementado el código de paso de mensajes en MPI pero el código de entrada/salida y asignación de bloques son en gran medida los mismos que en PVMPOV...".

Ya que MPI exhibió mejor rendimiento, mayor facilidad de uso, mejor estabilidad y capacidad de recuperación de errores, se consideró más apropiado utilizar

[14] *http://www-mddsp.enel.ucalgary.ca/People/adilger/povray/pvmpov.html*

[15] *http://www.verrall.demon.co.uk/mpipov*

MPI-Povray.

El utilizar código disponible públicamente permitió reducir el tiempo en el cual se logra tener una aplicación real operativa en el *cluster*.

3.4.6 Instalación de MPI-POV-Ray

Se debe obtener el paquete de POV-Ray para la arquitectura correcta, en este caso Linux. Este paquete contiene la distribución binaria de POV-Ray, incluyendo los archivos de datos y auxiliares que se requerirán, y se encuentra en *ftp://ftp.povray.org/pub/povray/Official/Linux/povlinux.tgz*. Se utiliza un paquete binario pues es más sencillo primero realizar la instalación de la versión uniprocesador de POV-Ray y posteriormente obtener el código fuente, las modificaciones necesarias para uso en equipos paralelos, y compilar únicamente el binario para uso paralelo.

A continuación se descompacta e instala el paquete por medio de los siguientes comandos:

```
# tar -zxvf povlinux.tar.gz
# cd povray31
# sh install
```

Una vez concluido este proceso, podemos utilizar la versión uniprocesador de POV-Ray. Ésta será útil para fines comparativos con el desempeño de la versión paralela.

Para poder utilizar POV-Ray en una máquina paralela se necesita contar con el código fuente a fin de poder aplicar las modificaciones necesarias al mismo, y posteriormente compilar la versión paralela.

El código fuente de POV-Ray se puede bajar de: *ftp://ftp.povray.org/pub/povray/Official/Unix/povuni_s.tgz*. Éste se coloca en el mismo subdirectorio que el archivo *povlinux.tgz* y se descompacta con el siguiente comando:

```
#tar -zxvf povuni_s.tgz
```

Esto genera un subdirectorio **povray31/source** conteniendo el código fuente.

El parche para utilizar MPI con POV-Ray se obtiene de:
http://www.verrall.demon.co.uk/mpipov/mpi-povray-1.0.patch.gz.

Este archivo se debe colocar en el directorio **povray31**. A continuación se aplica el parche con los siguientes comandos:

```
# gunzip mpi-povray-1.0.patch.gz
# patch -p1 < mpi-povray-1.0
```

Esto aplica las modificaciones necesarias a POV-Ray para que utilice MPI. A continuación se debe compilar el código fuente modificado. Normalmente, para ello se debe modificar el *Makefile* ubicado en **povray31/source/mpi-unix/**, sin embargo, ya que la configuración por omisión asume que se está utilizando MPICH, como es el caso, no se requieren modificaciones.

Para compilar la versión paralela de POV-RAY se cambia al directorio **povray31/source/mpi-unix** y se da el siguiente comando:

```
# make
```

Esto genera un ejecutable **mpi-x-povray**. Este ejecutable se coloca en **/usr/local/bin**, donde está accesible para todos los nodos.

3.4.7 Utilizando POV-Ray

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

Así pues, una invocación típica sería:

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

Esto utiliza los parámetros por omisión, generando un archivo de salida con el mismo nombre base que el de entrada, en formato PNG (por lo tanto la salida

quedará en **skyvase.png**). La resolución, si no se especifica, será de 320 × 240 *pixels*.

Adicionalmente, en este caso se cuenta con MPI-POVRay, cuya invocación, desde luego, se debe realizar a través de **mpirun**:

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

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

3.4.8 Algunas pruebas de rendimiento

Una vez contando con el *software* para *ray tracing* en paralelo, inmediatamente se piensa en probar su ejecución y desempeño trazando una escena real. Para este propósito, una excelente opción es realizar el *benchmark* oficial de POV-Ray, lo que permitirá evaluar el funcionamiento con una escena de uso común, así como obtener más datos sobre el rendimiento del *cluster*.

El *benchmark* de POV-Ray

Desde 1994, se desarrolló una metodología para comparar el rendimiento de trazado con POV-Ray en distintos sistemas. Esta metodología consiste en trazar una escena en particular, con parámetros y opciones bien definidos, y tomar el tiempo de trazado de dicha escena. Este tiempo se compara con el obtenido por otros sistemas para darse una idea del desempeño relativo en esta aplicación particular.

El sitio oficial del *benchmark* de POV-Ray, mantenido por Andrew Haveland-Robinson, está en la página de internet *http://www.haveland.com/povbench/index.htm*.

La escena consiste en una vasija, con una textura basada en una imagen de nubes, sobre un pedestal, frente a un escenario de paredes reflejantes. La escena es sencilla pero prueba todos los elementos básicos de un programa de *ray tracing*, como son geometría sólida constructiva, aplicación de texturas, y propiedades de materiales tales como reflexión y transparencia. Dicha escena se muestra, con fines de referencia, en la figura (3.4). La descripción de la escena empleada, que se obtuvo del sitio oficial, se incluye en el apéndice (C.1).

Figura 3.4: Aspecto final de la escena *skyvase.pov*

Las reglas para obtener un resultado válido en el *benchmark* especifican los argumentos empleados para la invocación de POV-Ray, a fin de que las corridas sean lo más homogéneas posible. Una invocación para este efecto es como sigue:

```
$ x-povray -i skyvase.pov +v1 -d +fn -x \
+a0.300 +r3 -q9 -w640 -h480 -mv2.0 +b1000
```

La invocación para MPI-POVRay es similar:

```
$ mpirun -np 17 mpi-x-povray -i skyvase.pov +v1 -d \
+fn -x +a0.300 +r3 -q9 -w640 -h480 -mv2.0 +b1000
```

El trazado de la escena se realizó, primero, con la versión uniprocesador, y posteriormente, con la versión paralela, variando el número de nodos para obtener la siguiente tabla de tiempos de trazado:

Nodos	Tiempo de trazado (s)
1	461
2	678
3	522
4	421
5	342
6	288
7	260
8	241
9	235
10	213
11	202
12	197
13	186
14	177
15	171
16	152
17	145

A partir de estos resultados se obtuvo la gráfica mostrada en la figura (3.5).

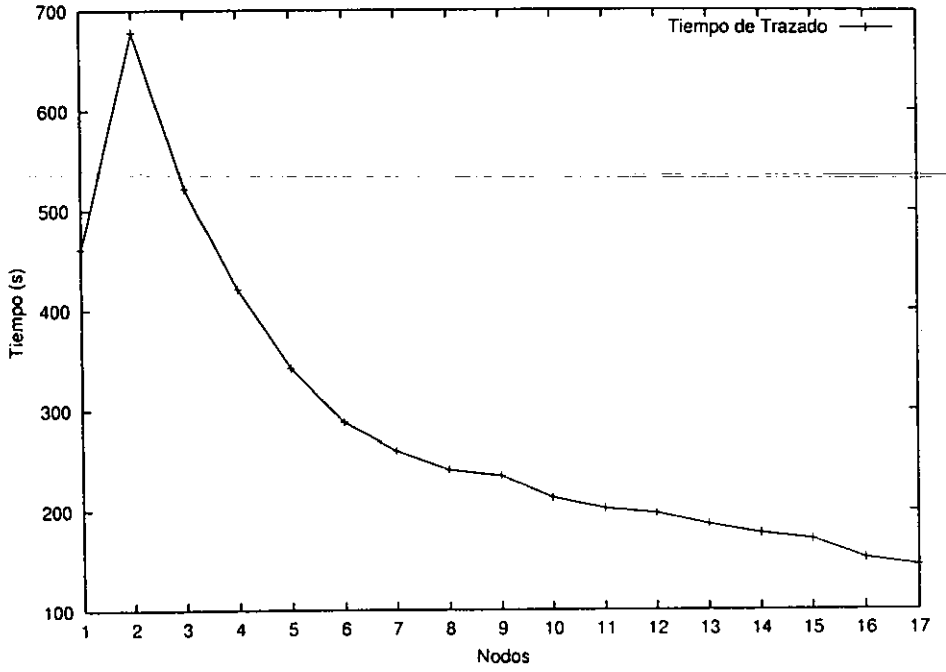

Figura 3.5: Tiempos de trazado de *skyvase.pov* con POV-Ray

Discusión de resultados

La gráfica obtenida en este caso es similar a las mostradas en la sección (3.2.9). Nuevamente se observa que al principio, utilizando pocos nodos, el rendimiento no mejora respecto al tiempo con un solo procesador. Sin embargo, en este caso se observa que el rendimiento mejora de manera más consistente al incrementar el número de nodos, alcanzando el desempeño de la versión uniprocesador más rápidamente y teniendo mejora constante de rendimiento al agregar más nodos.

Esto responde al algoritmo de asignación de trabajo utilizado en MPI-POVRay. En la sección (3.2.10) se describió una manera más eficiente de asignar el trabajo en bloques pequeños a los nodos que vayan concluyendo el trabajo previamente asignado y vayan quedando disponibles. MPI-POVRay utiliza este algoritmo de asignación, y como se aprecia, los resultados son bastante positivos.

Los resultados obtenidos pueden compararse con las tablas del *benchmark* oficial. El mejor tiempo obtenido por el cluster (145 segundos) se ubica alrededor de los tiempos logrados por equipos Pentium II a 266 y 300 MHz (entre 140 y 150 segundos). Por otro lado, el tiempo con un solo procesador (461 segundos) está entre los tiempos logrados por equipos Pentium a 120 y 133 MHz; esto tiene sentido si se considera que el servidor principal, donde se realizaron las pruebas uniprocesador, cuenta con un procesador Pentium a 133 MHz.

El *benchmark* oficial cuenta con una tabla de resultados para equipos paralelos, entre los que se incluyen gran cantidad de *clusters*. En esta tabla, un *cluster* con dos nodos Pentium MMX a 166 MHz obtuvo un tiempo de 145 segundos, exactamente el mismo que se obtuvo en este *cluster*.

De estos resultados pueden desprenderse dos observaciones, que refuerzan las conclusiones obtenidas en los anteriores *benchmarks* y pruebas realizadas en el *cluster*. La primera es que el rendimiento efectivamente se incrementa al utilizar más nodos, si bien el rendimiento máximo que se logró corresponde al de equipos uniprocesador que en la actualidad no se consideran avanzados (tómese el ejemplo del equipo Pentium II a 266 MHz). La segunda, y quizá más importante, es que, por medio de la cooperación entre sistemas que son ya francamente obsoletos, como los nodos empleados en este proyecto, se alcanza el rendimiento de un equipo que

está varias generaciones adelante de los mismos nodos[16].

3.4.9 Utilización en aplicaciones de *ray tracing*

Una vez realizadas estas pruebas, se determinó que el *ray tracing* es una aplicación que proporciona excelentes resultados en un *cluster*. Durante las pruebas se constató que la implementación de POV-Ray paralelo utilizada es eficiente y estable, así como totalmente compatible con la versión uniprocesador de POV-Ray.

La escena utilizada no es particularmente compleja, por ello no requiere mucha capacidad de memoria, que es una limitación importante en este *cluster*. Con la única salvedad de tener en cuenta los requerimientos de memoria para escenas complicadas y con muchos elementos, se puede utilizar el *cluster* de manera confiable para trazado de imágenes.

El *ray tracing* produce resultados vistosos pero cuya utilidad práctica no es inmediatamente obvia. Sin embargo esta técnica tiene una gran cantidad de aplicaciones prácticas.

El lenguaje de descripción de escenas es sencillo y es fácil generar las escenas automáticamente, a través de un programa que calcule las posiciones de los elementos, o bien importar escenas a partir de formatos usados por programas de diseño populares, como AutoCAD. Esto puede utilizarse, entre otras cosas, para generar vistas previas o incluso "paseos virtuales" de diseños de ingeniería o arquitectónicos, de utilidad cuando se requiere mostrar el aspecto que tendrá un proyecto terminado, y donde es importante que el resultado sea vistoso y de alta calidad. En esta aplicación es de importancia la gran calidad de imagen que se obtiene por medio de *ray tracing*.

Otra posible aplicación es en cómputo científico, en el cual es importante generar representaciones visuales de los problemas, sus soluciones y su comportamiento. De hecho el cómputo visual es una de las aplicaciones de computación con mayores requerimientos de cálculo. La capacidad de tener un equipo de bajo costo y buen rendimiento para generar tanto imágenes estáticas como animaciones puede permitir a científicos de distintas disciplinas visualizar y, con ello, comprender mejor, el comportamiento de algunos problemas.

[16] Los procesadores 486/66 de los nodos se consideraban modernos en 1993, mientras que el Pentium II a 266 MHz apareció unos 4 años después, en 1997.

Finalmente, no se debe descartar la utilidad del *ray tracing* con fines recreativos, sobre todo teniendo en cuenta que, por la naturaleza de esta aplicación, incluso su uso para recreación permite al usuario una mejor comprensión de la geometría espacial, las características de representación de formas, texturas e iluminación del *ray tracing*, y técnicas de graficación por computadora, a través de un mecanismo que proporciona retroalimentación visual rápidamente, trazando las imágenes en poco tiempo.

3.5 Resumen

En el capítulo 3 se comenzó la utilización propiamente dicha del *cluster*. Inicialmente se desarrolló una aplicación sencilla de multiplicación de matrices, con dos finalidades básicas: conocer y familiarizarse con la programación, compilación y ejecución de aplicaciones, tanto en MPI como en PVM, así como también realizar comparaciones entre éstas basándose en la experiencia adquirida en este desarrollo; y permitir la evaluación del desempeño del *cluster*, para lo cual se resolvieron problemas de distintas dimensiones, variando la capacidad del *cluster* utilizada en cada solución.

A continuación se consiguió, instaló y ejecutó un *benchmark* de uso muy difundido, el HPL, a fin de obtener una medida de desempeño en MFLOPS que se puede comparar con la de otros equipos paralelos existentes.

Finalmente, se propuso el uso del *cluster* para una aplicación real, en este caso generación de imágenes por *ray tracing*. Se localizó un programa para *ray tracing* públicamente disponible, el POV-Ray. Se consiguieron las modificaciones necesarias al código fuente del programa para su uso en máquinas paralelas, integrándolas al programa e instalando la versión resultante. Se ejecutaron los *benchmarks* oficiales de POV-Ray para evaluar el desempeño del *cluster* en esta aplicación. Durante este proceso se evaluó el comportamiento del *cluster* al utilizar esta aplicación real. Por último, se analizaron las posibilidades que otorga el contar con un equipo para realizar *ray tracing* con buen rendimiento y a bajo costo, y la utilidad práctica de esta aplicación.

Los resultados de todas las pruebas realizadas se presentaron en tablas y, cuando resultó apropiado, en gráficas para su visualización, discusión y análisis

Capítulo 4

Conclusiones

El presente proyecto resultó ser de un alcance bastante amplio, englobando una serie importante de aspectos, técnicas, y sub-disciplinas relativas a la computación. Esto fue particularmente notorio durante la fase de construcción del *cluster* (capítulo 2). Sólo en esta fase se requirieron conocimientos y técnicas de evaluación y ensamblado de *hardware*, implementación física de redes locales, mecanismos de arranque de equipos PC y compatibles, conocimientos generales de sistema operativo Linux, redes TCP/IP, una colección interesante de protocolos como NFS, DHCP, TFTP y NIS, organización de sistemas de archivos y compilación, instalación y ejecución de programas y bibliotecas.

Más allá de la gran cantidad de temas que de alguna u otra manera se tocaron durante la realización del proyecto, y que resultaron, además de un reto interesante y complejo, una excelente experiencia de aprendizaje, se pueden obtener una serie de conclusiones de la realización del proyecto.

4.1 Construyendo un *Beowulf*

4.1.1 *Hardware*: Un recurso limitado

El alcance del proyecto estuvo determinado de forma muy estricta por el *hardware* con el que se contaba. Los recursos, tanto de *hardware* disponible como de presupuesto para adquisición de los componentes faltantes, dictaron muchas decisiones de diseño e implementación que de otra forma hubieran sido diferentes, e

introdujeron algunos factores limitantes en la utilización del *cluster*.

El objetivo de construir un *cluster* es buscar alto rendimiento. En este caso, dados los recursos modestos con que se contó, el *cluster*, desde el comienzo, buscaba obtener *un rendimiento más alto* que el de un solo procesador. Por lo tanto, y como se verá a continuación, algunas decisiones de diseño fueron del tipo "esta opción es lo suficientemente buena para el enfoque y alcance del proyecto".

A continuación se presentan algunos comentarios y conclusiones sobre las capacidades y limitaciones del *hardware* que se utilizó en el proyecto.

Procesador y memoria

No se contó con mucho poder de elección en cuanto a los nodos que componen el *cluster*. Los equipos utilizados fueron "rescatados" de la Unidad de Cómputo Académico (UNICA). Su comportamiento resultó adecuado, sin embargo, su disparidad tanto en velocidad de procesador como en cantidad de memoria, introdujo una serie de problemas de "cuello de botella" (sección (3.2.10)) y tamaño máximo de problema trabajable (sección (3.2.7)). En particular, contar con más memoria hubiera sido de gran interés, pues hubiera permitido atacar problemas más grandes, que como se observó en las secciones (3.2.10) y (3.3.4), obtendrían un mejor desempeño del *cluster*, aún con los modestos procesadores con que se contaba.

Almacenamiento secundario

La falta de disco duro de los nodos dio lugar a una de las características centrales del proyecto, que resultó de gran interés, y cuya implementación fue interesante y desafiante: la configuración de nodos *diskless*. La configuración requerida para habilitar esta operación consumió gran parte del esfuerzo dedicado a la construcción del *cluster*. A pesar del logro técnico que esto representa, fue causado por una carencia que, en retrospectiva, resultó ser importante para el proyecto.

El compartir un mismo disco duro entre todos los nodos supuso un factor de bajo desempeño al arrancar los mismos, así como al realizar la carga de cualquier archivo del disco duro, ya sea programas ejecutables, bibliotecas o archivos de datos. Esto afecta adversamente al rendimiento, tanto del *cluster* en general como del servidor central, quien se ve forzado a soportar cargas de trabajo en cuanto a

entrada/salida de disco duro y red que idealmente deberían evitarse.

La solución, si bien adecuada para la plataforma con que se contó, debe evitarse en la medida de lo posible. El uso de sistemas de archivos compartidos debería restringirse a su uso en áreas estratégicas, como los archivos *home* de los usuarios, donde usar esta técnica es recomendable y no impacta adversamente al rendimiento.

Red local

La columna vertebral del *cluster* es la red local que interconecta a los nodos, y que permitió a los mismos el uso de sistemas de archivos remotos, así como las comunicaciones por paso de mensajes esenciales para su operación.

Esta red local, ensamblada a muy bajo costo, resultó también un factor a tomar muy en cuenta para el desempeño del *cluster*. Como se vio en la sección (4.1.1), el rendimiento de la red es esencial para permitir a los nodos un acceso eficiente a sus archivos. Adicionalmente, durante el paso de mensajes nos interesa contar con una red de alto rendimiento.

Desafortunadamente, la red con que se contaba tenía un desempeño más bien modesto. Tratándose de una red Ethernet normal, ésta se saturaba rápidamente, lo cual, aunado a su relativamente baja velocidad de 10 Mbits/s, la convirtió en un factor determinante para el desempeño del *cluster*.

Idealmente, se debe buscar contar con una red segmentada, lo cual ayudaría enormemente al rendimiento. El uso de un *switch* confina el tráfico a los segmentos que realmente participan en una transacción, eliminando la característica de bus de una red Ethernet. Adicionalmente, una red de 100 Mbits/s proporcionaría un desempeño considerablelmente mayor.

Implementar estas soluciones no fue posible debido al costo de un *switch* con las características mencionadas, y a que los nodos no son capaces de equiparse con tarjetas Fast Ethernet. Sin embargo es una consideración básica para otras implementaciones de *clusters* con mayor presupuesto.

4.1.2 *Software*: Adaptándose a las necesidades

Dadas las limitaciones mencionadas de la plataforma de *hardware*, se dejó la parte más complicada de la implementación al *software*, ya que éste es siempre más flexible y se puede adaptar a las necesidades de *hardware*. Esto se contrapone a la proposición de "escoger primero el *software* y después el *hardware*", sin embargo, dada la naturaleza del proyecto, el *software* tuvo que adaptarse a las condiciones presentes. Afortunadamente, aunque esto supuso bastante trabajo, la implementación final resultó adecuada, limpia y confiable.

La base para la "hazaña" de crear una estructura donde 17 nodos sin disco duro pueden cargar un sistema operativo completo tipo Unix y ejecutar aplicaciones sin requerir más que un disco flexible para su arranque es, en gran parte, mérito del sistema operativo Linux y las utilerías, programas y aplicaciones que lo componen, en particular aquellas que se emplearon directamente en la construcción y operación del *cluster* y que se describen en el capítulo 2. El hecho de que dicha operación haya sido posible, así como la excelente estabilidad y gran desempeño, dados los recursos existentes, es un testamento al logro de la comunidad de *software* libre que ha desarrollado el sistema Linux y las utilerías GNU.

La parte más compleja de la implementación de *software* fue el lograr sistemas relativamente independientes, pero que realizan arranque de un servidor de red. Normalmente, el arranque de red se realiza para terminales que comparten configuración y archivos. En este caso, el requerimiento adicional de tener cierta independencia operativa complicó las cosas, pues dicho procedimiento no estaba documentado en su totalidad, a diferencia de muchas otras tareas que se pueden realizar en un sistema Linux, y que están documentadas en el excelente *Linux Documentation Project*[1]. Finalmente la implementación a que se llegó fue operativa y estable, al grado de permitir la exitosa terminación del proyecto.

Una vez realizada esta tarea, la instalación de los componentes necesarios para utilizar el *cluster* para cómputo paralelo no supuso complicaciones, dando evidencia de que dicho *software* está ampliamente probado por una de las comunidades más exigentes en el ámbito de la computación, como es la comunidad del cómputo de alto rendimiento.

[1] *http://www.ldp.org*

4.2 Utilizando un *Beowulf*

Una vez contando con un *Beowulf* estable y funcional, se procede a utilizarlo; pues ésta es la finalidad de cualquier equipo de cómputo.

El primer paso de la utilización del *cluster* fue el desarrollo de una aplicación sencilla utilizando las dos bibliotecas de paso de mensajes disponibles, PVM y MPI. Esto permitió familiarizarse con la creación, compilación y ejecución de aplicaciones con estas bibliotecas. Adicionalmente, dicha aplicación fue probada exhaustivamente con diferentes parámetros, a fin de obtener una idea bastante clara sobre cómo se comporta un *cluster*, particularmente en cuanto a su desempeño al variar los parámetros de operación.

Al desarrollar la aplicación se confirmó que existen una serie de similitudes a nivel funcional entre PVM y MPI, y que ambas bibliotecas poseen una serie de ventajas y desventajas que se deberán tomar en cuenta al momento de seleccionar cuál utilizar. En este caso, se llegó a una preferencia por MPI, pues, como se detalló en la sección (3.2.2), PVM presenta algunos problemas que, si bien no son graves, pueden resultar engorrosos e incómodos; por otro lado, MPI presenta una API mucho más limpia, un diseño mejor planeado, mayor rendimiento y estabilidad. Estas observaciones apoyan el auge de MPI como la interfaz de paso de mensajes preferida de la comunidad de cómputo paralelo.

Las pruebas realizadas dieron los primeros resultados de desempeño del *cluster*, los cuales se presentan en la sección (3.2.8) y (3.2.9), y cuyo análisis se presenta en la sección (3.2.10). Aunque en este momento no se puede realizar una comparación con otros sistemas, estas primeras pruebas nos confirman una hipótesis importante que se tiene en todo *cluster*: el rendimiento se incrementa al utilizar más elementos de procesamiento.

Las pruebas realizadas con el *benchmark* HPL en la sección (3.3) nos permiten confirmar la observación anterior, además de permitirnos poner el desempeño del *cluster* en perspectiva. Si bien se enfrenta la primera noción de que el desempeño en términos reales no es particularmente espectacular, se tiene también un resultado más que respalda la observación de que, efectivamente, se obtiene una mejora de rendimiento al utilizar técnicas de cómputo paralelo.

Por último, la aplicación del *cluster* en la resolución de un problema real, en

particular el trazado de imágenes por *ray tracing* empleando una variante paralela del programa POV-Ray, descrito en la sección (3.4.1), permite apreciar, de primera mano, uno de los posibles usos de un *cluster*, en particular uno que es relativamente vistoso, lo cual ayuda a presentar el concepto de la utilidad del *cluster*. Como un beneficio adicional, se obtienen aún más resultados que permiten posicionar el rendimiento del *cluster* en términos reales a fin de realizar comparaciones.

4.3 ¿Para qué más sirve este *Beowulf*?

Se podría imaginar que, una vez concluido el proyecto, el *Beowulf* que se ha construido estará destinado a ser desmantelado. Más aún, aún en caso contrario, quizá esté destinado a ser un fósil sin utilidad práctica, más allá de servir como objeto de estudio y observación.

Sin embargo no está totalmente desprovisto de utilidad. La plataforma existe, y en realidad es sólo cuestión de que se le encuentre un uso, lo cual no es de ninguna manera imposible, como demuestra la aplicación práctica presentada en la sección (3.4.1).

Otro uso posible para el *Beowulf*, y uno que resulta de cabal importancia, es como plataforma de aprendizaje. El cómputo paralelo es una disciplina complicada, que requiere práctica, conocimiento de las características de una máquina paralela, y la capacidad para implementar algoritmos eficientes que aprovechen sus capacidades. Un *cluster* modesto, como el que se construyó en este proyecto, puede tener futuro como plataforma para permitir el estudio práctico de técnicas y herramientas de programación paralela, así como la experimentación con distintos algoritmos. Esto permite generar experiencia y conocimiento en esta área de la computación, lo cual contribuye a enriquecer la comunidad de cómputo de alto rendimiento con gente con mayor experiencia y capacitación.

4.4 "Concluyendo" las conclusiones

De los resultados obtenidos en este trabajo y detallados en esta sección, hay dos que son particularmente importantes pues se acoplan precisamente a los objetivos del proyecto.

Primero, se observa que la construcción de un *cluster* con equipo en desuso es posible, y la plataforma resultante es perfectamente utilizable y práctica.

Segundo, en todos los casos se observó que la teoría básica de operación de un *cluster* se cumple, obteniendo un mayor desempeño al agregar más nodos al cálculo, siempre que se esté consciente de las características del equipo, el problema a resolver, y se implemente un algoritmo adecuado y eficiente para atacar el problema. Es conveniente resaltar que, como se mencionó en las secciones (3.2.10) y (3.3.4), el tamaño del problema que se va a trabajar, así como el número de nodos del *cluster* que se emplearán en la solución del problema, son los factores más importantes para determinar el rendimiento máximo que se podrá alcanzar, por lo cual es vital tener en cuenta estos dos factores y balancearlos para aprovechar al máximo las capacidades del *cluster*.

Esto último es quizá la conclusión más importante, pues nos lleva a poder "proyectar" los resultados obtenidos en este proyecto a posibilidades de construcción de *clusters* futuras. Como se mencionó en la sección (3.4.8), el conjunto de varios equipos que ya se consideran antiguos, permite alcanzar el rendimiento de un equipo que se encuentra varias generaciones adelante. De esto se infiere que un *cluster* construido con equipos modernos será equivalente a un equipo que esté varias generaciones en el futuro. La implicación interesante de esto es que ¡dicho equipo aún no existe! Esto nos lleva a una observación que es conocida en el mundo del cómputo paralelo: un equipo paralelo puede ser significativamente mejor que uno uniprocesador *al momento de su aparición*. El aplicar las técnicas, teoría y observaciones que se han presentado en este trabajo, a la construcción de un *cluster* con equipo moderno y recursos suficientes, pueden permitir la creación de una máquina que alcance un rendimiento verdaderamente elevado a un precio accesible. Esto es, desde luego, la idea básica detrás de la creación de un *Beowulf*.

Finalmente, se considera que los objetivos planteados al inicio del proyecto se alcanzaron satisfactoriamente, y se espera que, en efecto, el resultado del presente proyecto pueda proveer una metodología para la construcción de un *cluster*, tomando en cuenta todos los factores que influyen para la creación, utilización y desempeño del mismo, así como dar una idea del desempeño que se puede lograr y las técnicas y principios que se deben seguir y respetar para ello.

Apéndice A

Programas aplicados

En este apéndice se presenta el código fuente de los programas desarrollados para realizar las pruebas de rendimiento con multiplicación de matrices, en uniprocesador, MPI y PVM.

A.1 Multiplicación de matrices

A.1.1 Rutinas de manejo de matrices

matrix.c

```
1  /* Funciones para manejo de matrices, 2001, Daniel
2     Manrique*/
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  // rellena una matriz aleatoriamente
8  void randomatrix(int *matrix,
9                   int row, int col, int maxval)
10 {
11     int i;
12
13     for (i = 0; i < (row * col); i++) {
14         *(matrix + i) = 1 + (int) ((float) maxval *
15                                    random() /
16                                    (RAND_MAX + 1.0));
17         //       printf ("elemento %d es
18         //       %d\n", i,*(matrix+i));
19     }
```

```
20
21  }
22
23
24  // dado un arreglo lineal lo trata como matriz y
25  // regresa el elemento especificado. row,col son
26  // iniciando en 0 es decir si quieres el primer
27  // elemento es 0,0 especificamos x, y
28  int matrix_get_cell(int *matrix,
29                      int rows, int cols, int x, int y)
30  {
31      int valor_lineal;
32      valor_lineal = (y * cols + x);
33      return matrix[valor_lineal];
34  }
35
36  // similar pero fija el valor del elemento dado en la
37  // matriz;
38  int matrix_set_cell(int *matrix,
39                      int rows,
40                      int cols, int x, int y, int val)
41  {
42      int valor_lineal;
43      valor_lineal = (y * cols + x);
44      matrix[valor_lineal] = val;
45      return 0;
46  }
47
48
49  // presentacion de matrices
50  int matrix_print_linear(int *matrix,
51                          int rows, int cols)
52  {
53      int x, y;
54      for (x = 0; x < cols; x++) {
55          for (y = 0; y < rows; y++) {
56              printf("%d,%d es %d\n",
57                     x,
58                     y,
59                     matrix_get_cell(matrix,
60                                     rows, cols, x, y));
61          }
62      }
63      return 0;
64  }
```

```
65
66 int matrix_print(int *matrix, int rows, int cols)
67 {
68     int x, y;
69     for (y = 0; y < rows; y++) {
70         for (x = 0; x < cols; x++) {
71             printf("%7d ",
72                     matrix_get_cell(matrix,
73                                     rows, cols, x, y));
74         }
75         printf("\n");
76     }
77     return 0;
78 }
```

A.1.2 Multiplicación de matrices - uniprocesador

unimatrix.c

```
1  /* Programa de multiplicación de matrices uniprocesador
2     2001, Daniel Manrique. */
3
4  #include <stdio.h>
5  #include <math.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <sys/time.h>
9
10
11 int main(int argc, char **argv)
12 {
13     int rv, row, col;
14     int *matrix1=NULL;
15     int *matrix2=NULL;
16     int *matrix3=NULL;
17     int x, y, i, k;
18     int elem1, elem2, suma;
19     int dimension;
20     int seed = (unsigned int) (time(0) / 2);
21     char optchar;
22     struct timeval starttime, endtime;
23     double dstart, dend;
24     int opt_print = 0;
25
26
```

```
27      // Este ciclo procesa las opciones de la línea de
28      // comandos y fija las banderas necesarias del
29      // programa.
30
31      do {
32          optchar = getopt(argc, argv, "fpd:");
33          switch (optchar) {
34          case 'p':
35              printf("imprimiendo matrices\n");
36              opt_print = 1;
37              break;
38          case 'd':
39              dimension = atoi(optarg);
40              printf("dimension %d\n", dimension);
41              break;
42          case 'f':
43              printf("matriz fija\n");
44              seed = 1;
45              break;
46          }
47      }
48      while (optchar != -1);
49
50      if (dimension == 0) {
51          printf("dimension 0, pos no funcionara\n");
52          return 0;
53      }
54      row = dimension;
55      col = dimension;
56
57
58      srandom(seed);
59
60      /* Asignar memoria para tres matrices de de row x
61         col */
62      matrix1 = malloc(row * col * sizeof(int));
63      matrix2 = malloc(row * col * sizeof(int));
64      matrix3 = malloc(row * col * sizeof(int));
65          if (matrix1 == NULL || matrix2 == NULL ||
66              matrix3 == NULL ){
67              if (matrix1!=NULL) free(matrix1);
68              if (matrix2!=NULL) free(matrix2);
69              if (matrix3!=NULL) free(matrix3);
70              exit(1);
71          }
```

```
72
73      /* llenamos las dos primeras matrices
74         aleatoriamente..  la definicion de la funcion
75         randomatrix esta en el archivo matrix.c */
76      randomatrix(matrix1, row, col, 5);
77      randomatrix(matrix2, row, col, 5);
78
79      // obtener inicio
80      rv = gettimeofday(&starttime, NULL);
81
82      // mostrar matrices..
83      if (opt_print) {
84          matrix_print(matrix1, row, col);
85          printf("\n");
86          matrix_print(matrix2, row, col);
87      }
88      // Ir calculando renglon por renglon
89      for (i = 0; i < row; i++) {
90          printf("calculado row %d\n", i);
91          /* el elemento que estoy calculando esta en y,x
92             entonces lo que va a variar va a ser la x
93             porque la y es fija por renglon */
94          y = i;
95          // aqui calculamos cada celda
96          for (x = 0; x < col; x++) {
97              //      printf ("calculando elemento
98              //      (%d,%d)\n",x,y);
99              suma = 0;
100             for (k = 0; k < col; k++) {
101                 elem1 =
102                     matrix_get_cell(matrix1, row, col,
103                                     k, y);
104                 elem2 =
105                     matrix_get_cell(matrix2, row, col,
106                                     x, k);
107                 suma += elem1 * elem2;
108                 //          printf ("%d*%d +
109                 //          ",elem1,elem2);
110             }
111             matrix_set_cell(matrix3, row, col, x, i,
112                             suma);
113         }
114     }
115     // Terminamos el calculo, obtener tiempo...
116     rv = gettimeofday(&endtime, NULL);
```

```
117
118     // Mostramos matrices..
119     if (opt_print) {
120         matrix_print(matrix3, row, col);
121     }
122     // Calculamos el tiempo que tardamos, mostramos y
123     // listo
124     dend = (double) endtime.tv_sec;
125     dend += (double) endtime.tv_usec * 0.000001;
126
127     dstart = (double) starttime.tv_sec;
128     dend += (double) starttime.tv_usec * 0.000001;
129
130     printf("wall clock time = %f\n", dend - dstart);
131     if (matrix1!=NULL) free(matrix1);
132     if (matrix2!=NULL) free(matrix2);
133     if (matrix3!=NULL) free(matrix3);
134     return 0;
135 }
```

A.1.3 Multiplicación de matrices paralelizada - MPI

matrix1.c

```
1  /* Programa de multiplicación de matrices paralelizado,
2     utilizando MPI.  2001, Daniel Manrique. */
3
4  #include "mpi.h"
5  #include "matrix.c"
6  #include <stdio.h>
7  #include <math.h>
8  #include <stdlib.h>
9  #include <unistd.h>
10
11 int main(int argc, char **argv)
12 {
13     int localid, numprocs, namelen, rv, row, col;
14     int *matrix1=NULL;
15     int *matrix2=NULL;
16     int *matrix3=NULL;
17     double startwtime, endwtime;
18     char processor_name[MPI_MAX_PROCESSOR_NAME];
19     int x, y, i, k;
20     int partitions, firstrow, lastrow, rowstodo;
21     int *resultrow;
```

```
22      int elem1, elem2, suma;
23      int completerows;
24      int dimension;
25      char optchar;
26      int opt_print = 0;
27      int seed = (unsigned int) (time(0) / 2);
28      MPI_Status status;
29
30      // Inicializar mpi
31      MPI_Init(&argc, &argv);
32
33      /* determinar numero total de procesos y cual
34         somos, asi como en que procesador (nodo) estamos
35         corriendo. */
36      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
37      MPI_Comm_rank(MPI_COMM_WORLD, &localid);
38      MPI_Get_processor_name(processor_name, &namelen);
39
40      printf("soy el proceso %d de %d en %s\n",
41             localid, numprocs, processor_name);
42      // Este ciclo procesa las opciones de la línea de
43      // comandos y fija las banderas necesarias del
44      // programa
45      do {
46          optchar = getopt(argc, argv, "fpd:");
47          switch (optchar) {
48          case 'p':
49              printf("imprimiendo matrices\n");
50              opt_print = 1;
51              break;
52          case 'd':
53              dimension = atoi(optarg);
54              printf("dimension %d\n", dimension);
55              break;
56          case 'f':
57              printf("matriz fija\n");
58              seed = 1;
59              break;
60          }
61      } while (optchar != -1);
62
63      if (dimension == 0) {
64          printf("dimension 0, pos no funcionara\n");
65          return 0;
66      }
```

```
67      row = dimension;
68      col = dimension;
69
70
71      /* En MPI el proceso inicial (padre) tiene rank
72         de 0, esta es la sección donde el proceso padre
73         realiza su trabajo */
74      if (localid == 0) {
75          srandom((unsigned int) (time(0) / 2));
76          // Asignar memoria para tres matrices de de row
77          // x col
78          matrix1 = malloc(row * col * sizeof(int));
79          matrix2 = malloc(row * col * sizeof(int));
80          matrix3 = malloc(row * col * sizeof(int));
81          if (matrix1 == NULL || matrix2 == NULL ||
82              matrix3 == NULL ){
83              MPI_Finalize();
84              if (matrix1!=NULL) free(matrix1);
85              if (matrix2!=NULL) free(matrix2);
86              if (matrix3!=NULL) free(matrix3);
87              exit(1);
88          }
89
90          /* llenamos las dos primeras matrices
91             aleatoriamente.. la definicion de la funcion
92             randomatrix esta en el archivo matrix.c */
93          randomatrix(matrix1, row, col, 5);
94          randomatrix(matrix2, row, col, 5);
95
96          /*Determinar el momento de inicio de ejecución,
97             aquí medimos cuando comenzamos realmente a
98             hacer los cálculos */
99          startwtime = MPI_Wtime();
100
101         // mostrar matrices si el usuario lo
102         // solicitó...
103         if (opt_print) {
104             matrix_print(matrix1, row, col);
105             printf("\n");
106             matrix_print(matrix2, row, col);
107         }
108         // mi registro de cuales rows ya estan
109         // completos
110         completerows = 0;
111
```

```
112         /* el proceso con rank 0 transmite las dos
113            matrices a los demás, esto se hace por medio
114            de un broadcast de MPI a todos los miembros de
115            mi comunicador (MPI_COMM_WORLD) */
116         rv = MPI_Bcast(matrix1,
117                        row * col,
118                        MPI_INT, 0, MPI_COMM_WORLD);
119         printf("Root, Broadcast said %d\n", rv);
120         rv = MPI_Bcast(matrix2,
121                        row * col,
122                        MPI_INT, 0, MPI_COMM_WORLD);
123         printf("Root, Broadcast said %d\n", rv);
124
125         // asignar un row temporal para resultados
126         resultrow = malloc((col + 1) * sizeof(int));
127         // esperar a tener todos los resultados
128         // completos
129         while (completerows < row) {
130             /* Esta llamada espera a recibir un renglón
131                ya resuelto, de cualquier proceso
132                hijo. Nótese el parametro MPI_ANY_SOURCE
133                que indica que se admiten valores de
134                cualquier proceso. */
135             MPI_Recv(resultrow,
136                      col + 1,
137                      MPI_INT,
138                      MPI_ANY_SOURCE,
139                      1, MPI_COMM_WORLD, &status);
140             printf("recibido renglon %d de %d\n",
141                    resultrow[0], status.MPI_SOURCE);
142             completerows++;
143             // Este ciclo "pone" el renglon recibido en
144             // mi matriz de resultado
145             for (i = 0; i < col; i++) {
146                 matrix_set_cell(matrix3,
147                                 row,
148                                 col,
149                                 i,
150                                 resultrow[0],
151                                 resultrow[i + 1]);
152             }
153         }
154         // terminamos el cálculo! anotar tiempo al
155         // terminar..
156         endwtime = MPI_Wtime();
```

```
157         // mostrar la matriz de resultados si se
158         // solicitó
159         if (opt_print) {
160             matrix_print(matrix3, row, col);
161         }
162         // Y mostrar el tiempo total de cálculo
163         printf("wall clock time = %f\n",
164                 endwtime - startwtime);
165         if (matrix1!=NULL) free(matrix1);
166         if (matrix2!=NULL) free(matrix2);
167         if (matrix3!=NULL) free(matrix3);
168
169     } // AQUI termina la ejecucion del proceso padre
170     else {
171         /* Aqui comienza lo que ejecuten procesos con
172            rank distinto de cero, es decir los procesos
173            hijo. Asigno espacio para dos matrices.. */
174         matrix1 = malloc(row * col * sizeof(int));
175         matrix2 = malloc(row * col * sizeof(int));
176         if (matrix1 == NULL || matrix2 == NULL){
177             MPI_Finalize();
178             if (matrix1!=NULL) free(matrix1);
179             if (matrix2!=NULL) free(matrix2);
180             if (matrix3!=NULL) free(matrix3);
181             exit(1);
182         }
183
184         /* En MPI, si mi rank no es 0, una llamada al
185            broadcast (notar el parametro 4 que es de 0)
186            indica recibir el broadcast del proceso con ese
187            rank. */
188         rv = MPI_Bcast(matrix1,
189                        row * col,
190                        MPI_INT, 0, MPI_COMM_WORLD);
191         rv = MPI_Bcast(matrix2,
192                        row * col,
193                        MPI_INT, 0, MPI_COMM_WORLD);
194
195         /*de acuerdo a las dimensiones de las matrices
196            y al numero de procesos, calcular numero de
197            particiones, renglones por particion, el
198            primer renglon que tiene que resolver este
199            proceso, y el ultimo renglon. */
200         partitions = numprocs - 1;
201         rowstodo = (int) (row / partitions);
```

```
202         firstrow = rowstodo * (localid - 1);
203         lastrow = firstrow + rowstodo - 1;
204
205         // el ultimo proceso amplia su limite para
206         // tomar los huerfanitos
207         if (localid == numprocs - 1) {
208             lastrow = lastrow + (row % partitions);
209         }
210
211 /*    printf ("me toca la particion %d\n", */
212 /*      localid); */
213 /*    printf ("en total hay %d particiones\n", */
214 /*      partitions); */
215 /*    printf ("cada particion tiene %d renglones, */
216 /*      ", rowstodo); */
217 /*    printf ("renglon inicial %d, ", */
218 /*      firstrow ); */
219 /*    printf ("renglon final %d\n", */
220 /*      lastrow); */
221
222         // asignar un row temporal
223         resultrow = malloc((col + 1) * sizeof(int));
224
225         // calcular cada row del grupo que me toca.
226         for (i = firstrow; i <= lastrow; i++) {
227             /* printf ("proceso %d[%s] haciendo row
228                %d\n", localid, processor_name, i); */
229             /* el elemento que estoy calculando esta en
230                y,x entonces lo que va a variar va a ser
231                la x porque la y es fija por renglon */
232             y = i;
233             for (x = 0; x < col; x++) {
234                 //              printf ("calculando
235                 //              elemento
236                 //              (%d,%d)\n", x, y);
237                 suma = 0;
238                 for (k = 0; k < col; k++) {
239                     elem1 = matrix_get_cell(matrix1,
240                                             row,
241                                             col, k, y);
242                     elem2 = matrix_get_cell(matrix2,
243                                             row,
244                                             col, x, k);
245                     suma += elem1 * elem2;
246                     //      printf ("%d*%d +
```

```
247                         //        ", elem1, elem2);
248                     }
249                     resultrow[x + 1] = suma;
250                 }
251                 // ya tengo calculado el renglon ahora se
252                 // lo tengo que mandar al proceso padre
253                 resultrow[0] = i;
254                 MPI_Send(resultrow,
255                          col + 1,
256                          MPI_INT, 0, 1, MPI_COMM_WORLD);
257                 //       printf ("enviado row %d\n", i);
258             }
259         if (matrix1!=NULL) free(matrix1);
260         if (matrix2!=NULL) free(matrix2);
261         if (matrix3!=NULL) free(matrix3);
262         } // termina seccion de calculo proceso hijo
263
264         // Terminamos la sesión de MPI
265         MPI_Finalize();
266
267         // y listo.
268         return 0;
269 }
```

A.1.4 Multiplicación de matrices paralelizada - PVM

matrix2.c

```
1  /* Programa de multiplicación de matrices paralelizado,
2     utilizando PVM.  2001, Daniel Manrique. */
3
4  //#include "mpi.h"
5  //#include "matrix.c"
6  #include <unistd.h>
7  #include <stdlib.h>
8  #include <stdio.h>
9  #include <math.h>
10 #include <pvm3.h>
11 #include <sys/time.h>
12
13 /* Definimos algunas banderas de mensaje. PVM usa esto
14 para distinguir entre distintos tipos de mensajes*/
15
16 #define MATRIX_TAG 1
17 #define ROW_TAG 2
```

```
18
19 int main(int argc, char **argv)
20 {
21     int mytid, myparent;
22     int rcvbuf;
23     int localid, numprocs, info;
24     int *child;
25     int ntask;
26     int *resultrow;
27     int rv, row, col;
28     int *matrix1=NULL;
29     int *matrix2=NULL;
30     int *matrix3=NULL;
31     int x, y, i, k;
32     int partitions, firstrow, lastrow, rowstodo;
33     int elem1, elem2, suma;
34     int completerows;
35     int dimension;
36     char optchar;
37     int opt_print = 0;
38     int seed = (unsigned int) (time(0) / 2);
39     struct timeval starttime, endtime;
40     double dstart, dend;
41
42     // obtener mi task id y la de mi proceso padre
43     mytid = pvm_mytid();
44     myparent = pvm_parent();
45
46     /* Este ciclo procesa las opciones de la línea de
47        comandos y fija las banderas necesarias del
48        programa. En particular nos interesa que esté en
49        este punto del programa porque necesitamos 1) que
50        el padre conozca cuantos hijos va a tener
51        (parametro t) y 2) que los hijos conozcan la
52        dimension de la matriz (parametro d) */
53     do {
54         optchar = getopt(argc, argv, "fpd:t:");
55         switch (optchar) {
56         case 'p':
57             printf("imprimiendo matrices\n");
58             opt_print = 1;
59             break;
60         case 'd':
61             dimension = atoi(optarg);
62             printf("dimension %d\n", dimension);
```

```
63              break;
64          case 't':
65              ntask = atoi(optarg);
66              if (ntask < 1)
67                  ntask = 1;
68              printf("%d procesos\n", ntask);
69              break;
70          case 'f':
71              printf("matriz fija\n");
72              seed = 1;
73              break;
74          }
75      } while (optchar != -1);
76
77
78
79      // Si soy el proceso padre, crear a los hijos
80      if (myparent == PvmNoParent) {
81          // Asignar el arreglo que va a tener
82          // los tid de los procesos hijo
83          child = (int *) malloc(sizeof(int) * ntask);
84
85          /* La llamada pvm_spawn crea a los hijos, hace
86             esto solicitando a la maquina virtual que cree
87             mas procesos. La maquina virtual decide donde
88             iniciarlos, normalmente hace un round-robin
89             entre los nodos, a menos que en esta llamada se
90             le especifique donde iniciarlos. OJO importante,
91             como la invocación de los hijos la hace esta
92             llamada, hay que pasar la lista de parametros
93             (desde argv[1]) a los hijos, para que la
94             procesen bien, en particular nos interesa que
95             agarren el parametro -d para que puedan
96             dimensionar su matriz del mismo tamaño que la
97             del padre. */
98          info = pvm_spawn(argv[0],
99                           &argv[1],
100                          PvmTaskDefault,
101                          (char *) NULL, ntask, child);
102         // Imprimir los TID de cada hijo
103         for (i = 0; i < ntask; i++)
104             if (child[i] < 0)    /* print the error code
105                                     in decimal */
106                 printf("%d ", child[i]);
107             else                 /* print the task id in
```

```
108                                     hex */
109                 printf("t%x\t ", child[i]);
110         putchar('\n');
111     }
112     /* Todos los procesos deben unirse al grupo
113        matrix_world. La llamada devuelve mi numero de
114        instancia dentro del grupo, se considera
115        equivalente al comm_rank de MPI */
116     localid = pvm_joingroup("matrix_world");
117
118     // Obtener el # de procesos en el grupo
119     numprocs = pvm_gsize("matrix_world");
120
121     printf("soy el proceso %d de %d\n",
122            localid, numprocs);
123     printf("my tid is %d, my parent %d\n", mytid,
124            myparent);
125
126     if (dimension == 0) {
127         printf("dimension 0, pos no funcionara\n");
128         return 0;
129     }
130     row = dimension;
131     col = dimension;
132
133     /* El proceso inicial (padre) no tiene padre (es
134        huerfano!), esta es la sección donde el proceso
135        padre realiza su trabajo */
136     if (myparent == PvmNoParent) {
137         printf("Padre\n");
138         srandom((unsigned int) (time(0) / 2));
139         // Asignar memoria para tres matrices de de row
140         // x col
141         matrix1 = malloc(row * col * sizeof(int));
142         matrix2 = malloc(row * col * sizeof(int));
143         matrix3 = malloc(row * col * sizeof(int));
144         if (matrix1 == NULL || matrix2 == NULL ||
145             matrix3 == NULL ){
146             pvm_exit();
147             if (matrix1!=NULL) free(matrix1);
148             if (matrix2!=NULL) free(matrix2);
149             if (matrix3!=NULL) free(matrix3);
150
151             exit(1);
152         }
```

```
153
154         /* llenamos las dos primeras matrices
155            aleatoriamente..  la definicion de la
156            funcion randomatrix esta en el archivo
157            matrix.c */
158         randomatrix(matrix1, row, col, 5);
159         randomatrix(matrix2, row, col, 5);
160
161         /*Determinar el momento de inicio de ejecución,
162            aquí medimos cuando comenzamos realmente a
163            hacer los cálculos */
164         rv = gettimeofday(&starttime, NULL);
165         // mostrar matrices..
166
167         if (opt_print) {
168             matrix_print(matrix1, row, col);
169             printf("\n");
170             matrix_print(matrix2, row, col);
171         }
172         // mi registro de cuales rows ya estan
173         // completos
174         completerows = 0;
175         /* el proceso padre transmite las dos matrices
176            a los demás, esto se hace por medio de un
177            broadcast a todos los miembros de mi grupo
178            (matrix_world) */
179
180         // limpiar e inicializar el buffer de
181         // transmision de PVM
182         pvm_initsend(PvmDataDefault);
183
184         /* OJO en PVM debemos empacar los datos en un
185            buffer de mensaje antes de poder
186            enviarlos. La llamada a pkint empaca el dato
187            al que apuntemos en el buffer de mensajes
188            actual. */
189
190         rv = pvm_pkint(matrix1, row * col, 1);
191         if (rv < 0) {
192             pvm_perror("pkint matrix1");
193         }
194         /*pvm_bcast transmite el buffer actual (matriz)
195            a todos los miembros del grupo
196            matrix_world. Le pone la bandera MATRIX_TAG,
197            una bandera definida por el usuario para
```

```
198             identificarlo */
199         rv = pvm_bcast("matrix_world", MATRIX_TAG);
200         printf("Root, Broadcast said %d\n", rv);
201
202         // segunda matriz, notese que limpiamos el
203         // buffer antes
204         pvm_initsend(PvmDataDefault);
205         pvm_pkint(matrix2, row * col, 1);
206
207         rv = pvm_bcast("matrix_world", MATRIX_TAG);
208         printf("Root, Broadcast said %d\n", rv);
209         // asignar un row temporal para resultados
210         resultrow = malloc((col + 1) * sizeof(int));
211         // esperar a tener todos los resultados
212         // completos
213         while (completerows < row) {
214             /* Esta llamada espera a recibir un renglón
215                ya resuelto, de cualquier proceso
216                hijo. Nótense los valores -1 que PVM
217                identifica como "wildcards" y nos indican
218                aceptar valores con cualquier bandera, y
219                de cualquier proceso. rcvbuf nos indica el
220                buffer donde se recibio el mensaje. */
221             rcvbuf = pvm_recv(-1, -1);
222             if (rcvbuf < 0) {
223                 pvm_perror("rcvbuf row");
224             }
225             /* Bajo PVM se necesita desempacar el
226                mensaje, sacarlo del buffer y ponerlo en
227                una variable utilizable. */
228             rv = pvm_upkint(resultrow, col + 1, 1);
229             if (rv < 0) {
230                 pvm_perror("upk row");
231             }
232             printf("recibido renglon %d\n",
233                     resultrow[0]);
234             completerows++;
235             // Este ciclo "pone" el renglon recibido en
236             // mi matriz de resultado
237             for (i = 0; i < col; i++) {
238                 matrix_set_cell(matrix3,
239                                 row,
240                                 col,
241                                 i,
242                                 resultrow[0],
```

```
243                                     resultrow[i + 1]);
244             }
245         }
246         // terminamos el cálculo! anotar tiempo al
247         // terminar..
248         rv = gettimeofday(&endtime, NULL);
249         //mostrar la matriz de resultados si se
250         //solicitó
251         if (opt_print) {
252             matrix_print(matrix3, row, col);
253         }
254         // Y mostrar el tiempo total de cálculo
255         dend = (double) endtime.tv_sec;
256         dend += (double) endtime.tv_usec * 0.000001;
257
258         dstart = (double) starttime.tv_sec;
259         dend += (double) starttime.tv_usec * 0.000001;
260
261         printf("wall clock time = %f\n",
262                 dend - dstart);
263         //if (matrix1!=NULL) free(matrix1);
264         //if (matrix2!=NULL) free(matrix2);
265         //if (matrix3!=NULL) free(matrix3);
266
267     } // AQUI termina la ejecucion del proceso padre
268     else {
269         printf("Hijo\n");
270         fflush(stdout);
271         /* Aqui comienza lo que ejecuten procesos que
272            sí tienen padre es decir los procesos hijo. */
273
274         // Asigno espacio para dos matrices
275         matrix1 = malloc(row * col * sizeof(int));
276         matrix2 = malloc(row * col * sizeof(int));
277
278         if (matrix1 == NULL || matrix2 == NULL){
279             pvm_exit();
280             if (matrix1!=NULL) free(matrix1);
281             if (matrix2!=NULL) free(matrix2);
282             if (matrix3!=NULL) free(matrix3);
283             exit(1);
284         }
285
286
287         /* Espero recibir un mensaje de mi proceso
```

```
288             padre (myparent), con bandera MATRIX_TAG,
289             entonces sabré que es una de las dos
290             matrices. Notese que en PVM es indistinto si
291             estoy recibiendo un broadcast o un mensaje
292             directo. */
293          rcvbuf = pvm_recv(-1, MATRIX_TAG);
294          if (rcvbuf < 0) {
295              pvm_perror("rcvbuf matrix1");
296          }
297          // Desempacar directo en el espacio de la
298          // matriz
299          rv = pvm_upkint(matrix1, row * col, 1);
300          if (rv < 0) {
301              pvm_perror("upk matrix1");
302          }
303          // repetimos para la siguiente matriz..
304          rcvbuf = pvm_recv(-1, MATRIX_TAG);
305          if (rcvbuf < 0) {
306              pvm_perror("rcvbuf matrix2");
307          }
308          rv = pvm_upkint(matrix2, row * col, 1);
309          if (rv < 0) {
310              pvm_perror("recv matrix2");
311          }
312          /*de acuerdo a las dimensiones de las matrices
313             y al numero de procesos, calcular numero de
314             particiones, renglones por particion, el
315             primer renglon que tiene que resolver este
316             proceso, y el ultimo renglon. */
317          partitions = numprocs - 1;
318          rowstodo = (int) (row / partitions);
319          firstrow = rowstodo * (localid - 1);
320          lastrow = firstrow + rowstodo - 1;
321          // el ultimo proceso amplia su limite para
322          // tomar los huerfanitos
323          if (localid == numprocs - 1) {
324              lastrow = lastrow + (row % partitions);
325          }
326
327  /*    printf ("me toca la particion %d\n", */
328  /*      localid); */
329  /*    printf ("en total hay %d particiones\n", */
330  /*      partitions); */
331  /*  printf ("cada particion tiene %d renglones, */
332  /*      ", rowstodo); */
```

```
333 /*  printf ("renglon inicial %d, ", */
334 /*     firstrow ); */
335 /*  printf ("renglon final %d\n", */
336 /*     lastrow ); */
337 /*   fflush(stdout); */
338         // asignar un row temporal
339         resultrow = malloc((col + 1) * sizeof(int));
340
341         // calcular cada row del grupo que me toca.
342         for (i = firstrow; i <= lastrow; i++) {
343             /* printf ("proceso %d[%s] haciendo row
344                %d\n", localid, processor_name, i); */
345             /* el elemento que estoy calculando esta en
346                y,x entonces lo que va a variar va a ser
347                la x porque la y es fija por renglon */
348             y = i;
349             for (x = 0; x < col; x++) {
350                 //              printf ("calculando
351                 //              elemento
352                 //              (%d,%d)\n", x, y);
353                 suma = 0;
354                 for (k = 0; k < col; k++) {
355                     elem1 = matrix_get_cell(matrix1,
356                                             row,
357                                             col, k, y);
358                     elem2 = matrix_get_cell(matrix2,
359                                             row,
360                                             col, x, k);
361                     suma += elem1 * elem2;
362                     //      printf ("%d*%d +
363                     //      ", elem1, elem2);
364                 }
365                 resultrow[x + 1] = suma;
366             }
367             // ya tengo calculado el renglon ahora se
368             // lo tengo que mandar al proceso padre
369             resultrow[0] = i;
370             // limpiar el buffer default
371             pvm_initsend(PvmDataDefault);
372             // empacar
373             pvm_pkint(resultrow, col + 1, 1);
374             /* Enviar, este es un envío directo a un
375                proceso, por medio de un send. Mandamos el
376                buffer de mensajes actual directamente al
377                proceso padre con una bandera de ROW_TAG
```

```
378                 para significar que lo que estamos
379                 enviando es un renglon, aunque en este
380                 caso el padre ignora el tipo de bandera. */
381               pvm_send(myparent, ROW_TAG);
382
383           } // termina proceso de un renglon
384       } // termina el calculo del proceso hijo
385
386       //terminamos la sesion de PVM
387       pvm_exit();
388       if (matrix1!=NULL) free(matrix1);
389       if (matrix2!=NULL) free(matrix2);
390       if (matrix3!=NULL) free(matrix3);
391
392       // listo
393       return 0;
394   }
```

Apéndice B

Archivos de configuración

B.1 Archivo de configuración demonio dhcp

```
server-identifier tornado;

#Opciones comunes a todas las subredes soportadas
option domain-name "unam.mx";
option domain-name-servers 132.248.10.2;

#asignacion dinamica..  notese que en este caso solo
#especificamos las subredes en las que se encuentra el
#servidor, puesto que no vamos a realizar asignación
#dinámica hay dos declaraciones de subred
#correspondientes a cada una de las interfases que
#tenemos.

shared-network TORNADO{
subnet 192.168.10.0 netmask 255.255.255.0 {
option broadcast-address 192.168.10.255;
}
}
subnet 192.168.1.0 netmask 255.255.255.0 {
}


# A partir de aquí especificamos la configuración por
# host.  cada bloque especifica un host, notar la
# dirección IP fija, el hostname que se asignará a
# cada nodo, y la direccion hardware ethernet que
# especifica a que direccion MAC se asignarán
# estos datos.
```

```
host tornado68 {
fixed-address 192.168.10.68;
hardware ethernet 00:60:08:0B:5A:9E;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado68";
}

host tornado69 {
fixed-address 192.168.10.69;
hardware ethernet 00:80:AD:30:49:D4;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado69";
}

host tornado70 {
fixed-address 192.168.10.70;
hardware ethernet 00:10:4B:35:74:A6;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado70";
}

host tornado71 {
fixed-address 192.168.10.71;
hardware ethernet 00:10:4B:35:75:B2;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado71";
}

host tornado72 {
fixed-address 192.168.10.72;
hardware ethernet 00:10:4B:35:75:89;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado72";
}

host tornado73 {
fixed-address 192.168.10.73;
hardware ethernet 00:10:4B:35:76:1E;
filename "/tftpboot/vmlinuz-nbi-2.2";
```

```
next-server 192.168.10.1;
option host-name "tornado73";
}

host tornado74 {
fixed-address 192.168.10.74;
hardware ethernet 00:60:97:35:83:DA;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado74";
}

host tornado75 {
fixed-address 192.168.10.75;
hardware ethernet 00:20:af:62:fa:af;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado75";
}

host tornado76 {
fixed-address 192.168.10.76;
hardware ethernet 00:60:97:34:d6:dd;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado76";
}

host tornado77 {
fixed-address 192.168.10.77;
hardware ethernet 00:20:af:4d:a6:20;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado77";
}

host tornado78 {
fixed-address 192.168.10.78;
hardware ethernet 00:20:af:4d:a6:24;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado78";
}

host tornado79 {
```

```
fixed-address 192.168.10.79;
hardware ethernet 00:20:af:70:5e:6e;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado79";
}

host tornado80 {
fixed-address 192.168.10.80;
hardware ethernet 00:20:af:4d:a6:48;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado80";
}

host tornado81 {
fixed-address 192.168.10.81;
hardware ethernet 00:20:af:4d:a6:40;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado81";
}

host tornado82 {
fixed-address 192.168.10.82;
hardware ethernet 00:20:af:52:fd:fb;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado82";
}

host tornado83 {
fixed-address 192.168.10.83;
hardware ethernet 00:20:af:62:fa:79;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado83";
}
host tornado84 {
fixed-address 192.168.10.84;
hardware ethernet 00:20:af:4d:a6:49;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado84";
}
```

```
host tornado85 {
fixed-address 192.168.10.85;
hardware ethernet 00:20:af:62:fa:33;
filename "/tftpboot/vmlinuz-nbi-2.2";
next-server 192.168.10.1;
option host-name "tornado85";
}
```

B.2 Archivo de configuración *kernel* Linux

```
#
# Automatically generated make config: don't edit
#

#
# Code maturity level options
#
CONFIG_EXPERIMENTAL=y

#
# Processor type and features
#
CONFIG_M386=y
# CONFIG_M486 is not set
# CONFIG_M586 is not set
# CONFIG_M586TSC is not set
# CONFIG_M686 is not set
# CONFIG_M686FX is not set
# CONFIG_X86_PN_OFF is not set
# CONFIG_X86_FX is not set
# CONFIG_X86_CPU_OPTIMIZATIONS is not set
CONFIG_MATH_EMULATION=y
CONFIG_MTRR=y
# CONFIG_SMP is not set
CONFIG_1GB=y
# CONFIG_2GB is not set

#
# Loadable module support
#
CONFIG_MODULES=y
CONFIG_MODVERSIONS=y
CONFIG_KMOD=y
```

```
#
# General setup
#
# CONFIG_BIGMEM is not set
CONFIG_NET=y
CONFIG_PCI=y
# CONFIG_PCI_GOBIOS is not set
# CONFIG_PCI_GODIRECT is not set
CONFIG_PCI_GOANY=y
CONFIG_PCI_BIOS=y
CONFIG_PCI_DIRECT=y
CONFIG_PCI_QUIRKS=y
# CONFIG_PCI_OPTIMIZE is not set
CONFIG_PCI_OLD_PROC=y
# CONFIG_MCA is not set
# CONFIG_VISWS is not set
CONFIG_SYSVIPC=y
CONFIG_BSD_PROCESS_ACCT=y
CONFIG_SYSCTL=y
CONFIG_BINFMT_AOUT=m
CONFIG_BINFMT_ELF=y
CONFIG_BINFMT_MISC=m
# CONFIG_BINFMT_JAVA is not set
CONFIG_PARPORT=m
CONFIG_PARPORT_PC=m
# CONFIG_PARPORT_OTHER is not set
# CONFIG_APM is not set

#
# Plug and Play support
#
CONFIG_PNP=y
CONFIG_PNP_PARPORT=m

#
# Block devices
#
CONFIG_BLK_DEV_FD=y
CONFIG_BLK_DEV_IDE=y

#
# Please see Documentation/ide.txt for help/info on IDE drives
#
# CONFIG_BLK_DEV_HD_IDE is not set
CONFIG_BLK_DEV_IDEDISK=y
```

```
CONFIG_BLK_DEV_IDECD=y
# CONFIG_BLK_DEV_IDETAPE is not set
# CONFIG_BLK_DEV_IDEFLOPPY is not set
# CONFIG_BLK_DEV_IDESCSI is not set
CONFIG_BLK_DEV_CMD640=y
# CONFIG_BLK_DEV_CMD640_ENHANCED is not set
CONFIG_BLK_DEV_RZ1000=y
CONFIG_BLK_DEV_IDEPCI=y
CONFIG_BLK_DEV_IDEDMA=y
# CONFIG_BLK_DEV_OFFBOARD is not set
# CONFIG_IDEDMA_AUTO is not set
# CONFIG_BLK_DEV_OPTI621 is not set
# CONFIG_BLK_DEV_TRM290 is not set
# CONFIG_BLK_DEV_NS87415 is not set
# CONFIG_BLK_DEV_VIA82C586 is not set
# CONFIG_BLK_DEV_CMD646 is not set
# CONFIG_IDE_CHIPSETS is not set

#
# Additional Block Devices
#
CONFIG_BLK_DEV_LOOP=m
CONFIG_BLK_DEV_NBD=y
# CONFIG_BLK_DEV_MD is not set
CONFIG_BLK_DEV_RAM=y
CONFIG_BLK_DEV_INITRD=y
# CONFIG_BLK_DEV_XD is not set
# CONFIG_BLK_DEV_DAC960 is not set
CONFIG_PARIDE_PARPORT=m
# CONFIG_PARIDE is not set
# CONFIG_BLK_CPQ_DA is not set
# CONFIG_BLK_DEV_HD is not set

#
# Networking options
#
CONFIG_PACKET=y
CONFIG_NETLINK=y
CONFIG_RTNETLINK=y
CONFIG_NETLINK_DEV=y
# CONFIG_FIREWALL is not set
CONFIG_FILTER=y
CONFIG_UNIX=y
CONFIG_INET=y
CONFIG_IP_MULTICAST=y
```

```
# CONFIG_IP_ADVANCED_ROUTER is not set
CONFIG_IP_PNP=y
CONFIG_IP_PNP_BOOTP=y
# CONFIG_IP_PNP_RARP is not set
# CONFIG_IP_ROUTER is not set
# CONFIG_NET_IPIP is not set
# CONFIG_NET_IPGRE is not set
# CONFIG_IP_MROUTE is not set
CONFIG_IP_ALIAS=y
# CONFIG_ARPD is not set
CONFIG_SYN_COOKIES=y

#
# (it is safe to leave these untouched)
#
CONFIG_INET_RARP=m
CONFIG_SKB_LARGE=y
# CONFIG_IPV6 is not set

#
#
#
# CONFIG_IPX is not set
# CONFIG_ATALK is not set
# CONFIG_X25 is not set
# CONFIG_LAPB is not set
# CONFIG_BRIDGE is not set
# CONFIG_LLC is not set
# CONFIG_ECONET is not set
# CONFIG_WAN_ROUTER is not set
# CONFIG_NET_FASTROUTE is not set
# CONFIG_NET_HW_FLOWCONTROL is not set
# CONFIG_CPU_IS_SLOW is not set

#
# QoS and/or fair queueing
#
# CONFIG_NET_SCHED is not set

#
# Telephony Support
#
# CONFIG_PHONE is not set
# CONFIG_PHONE_IXJ is not set
```

```
#
# SCSI support
#
# CONFIG_SCSI is not set

#
# Network device support
#
CONFIG_NETDEVICES=y

#
# ARCnet devices
#
# CONFIG_ARCNET is not set
CONFIG_DUMMY=m
CONFIG_BONDING=m
# CONFIG_EQUALIZER is not set
CONFIG_ETHERTAP=m
# CONFIG_NET_SB1000 is not set

#
# Ethernet (10 or 100Mbit)
#
CONFIG_NET_ETHERNET=y
CONFIG_NET_VENDOR_3COM=y
# CONFIG_EL1 is not set
# CONFIG_EL2 is not set
# CONFIG_ELPLUS is not set
# CONFIG_EL16 is not set
CONFIG_EL3=y
CONFIG_3C515=y
CONFIG_BC90X=y
CONFIG_VORTEX=y
# CONFIG_LANCE is not set
# CONFIG_NET_VENDOR_SMC is not set
# CONFIG_NET_VENDOR_RACAL is not set
CONFIG_RTL8139=y
CONFIG_NET_ISA=y
# CONFIG_AT1700 is not set
# CONFIG_E2100 is not set
CONFIG_DEPCA=y
# CONFIG_EWRK3 is not set
# CONFIG_EEXPRESS is not set
# CONFIG_EEXPRESS_PRO is not set
# CONFIG_FMV18X is not set
```

```
# CONFIG_HPLAN_PLUS is not set
# CONFIG_HPLAN is not set
# CONFIG_HP100 is not set
# CONFIG_ETH16I is not set
CONFIG_NE2000=y
# CONFIG_SEEQ8005 is not set
# CONFIG_SK_G16 is not set
CONFIG_NET_EISA=y
# CONFIG_PCNET32 is not set
# CONFIG_AC3200 is not set
# CONFIG_APRICOT is not set
# CONFIG_CS89x0 is not set
CONFIG_DM9102=y
CONFIG_DE4X5=y
# CONFIG_DEC_ELCP is not set
CONFIG_DEC_ELCP_OLD=m
# CONFIG_DGRS is not set
# CONFIG_EEXPRESS_PRO100 is not set
# CONFIG_LNE390 is not set
# CONFIG_NE3210 is not set
CONFIG_NE2K_PCI=y
# CONFIG_TLAN is not set
# CONFIG_VIA_RHINE is not set
# CONFIG_SIS900 is not set
# CONFIG_ES3210 is not set
# CONFIG_EPIC100 is not set
# CONFIG_ZNET is not set
# CONFIG_NET_POCKET is not set

#
# Ethernet (1000 Mbit)
#
# CONFIG_ACENIC is not set
# CONFIG_YELLOWFIN is not set
# CONFIG_SK98LIN is not set
# CONFIG_FDDI is not set
# CONFIG_HIPPI is not set
CONFIG_PLIP=m
CONFIG_PPP=m

#
# CCP compressors for PPP are only built as modules.
#
# CONFIG_SLIP is not set
# CONFIG_NET_RADIO is not set
```

```
#
# Token ring devices
#
# CONFIG_TR is not set
# CONFIG_NET_FC is not set
# CONFIG_RCPCI is not set
# CONFIG_SHAPER is not set

#
# Wan interfaces
#
# CONFIG_HOSTESS_SV11 is not set
# CONFIG_COSA is not set
# CONFIG_SEALEVEL_4021 is not set
# CONFIG_COMX is not set
# CONFIG_DLCI is not set
# CONFIG_SBNI is not set

#
# Amateur Radio support
#
# CONFIG_HAMRADIO is not set

#
# IrDA subsystem support
#
# CONFIG_IRDA is not set

#
# ISDN subsystem
#
# CONFIG_ISDN is not set

#
# Old CD-ROM drivers (not SCSI, not IDE)
#
# CONFIG_CD_NO_IDESCSI is not set

#
# Character devices
#
CONFIG_VT=y
CONFIG_VT_CONSOLE=y
CONFIG_SERIAL=y
```

```
CONFIG_SERIAL_CONSOLE=y
CONFIG_SERIAL_EXTENDED=y
CONFIG_SERIAL_MANY_PORTS=y
CONFIG_SERIAL_SHARE_IRQ=y
# CONFIG_SERIAL_DETECT_IRQ is not set
CONFIG_SERIAL_MULTIPORT=y
# CONFIG_HUB6 is not set
CONFIG_SERIAL_NONSTANDARD=y
# CONFIG_COMPUTONE is not set
# CONFIG_ROCKETPORT is not set
# CONFIG_CYCLADES is not set
# CONFIG_DIGIEPCA is not set
# CONFIG_DIGI is not set
# CONFIG_ESPSERIAL is not set
# CONFIG_MOXA_INTELLIO is not set
# CONFIG_MOXA_SMARTIO is not set
# CONFIG_ISI is not set
# CONFIG_RISCOM8 is not set
# CONFIG_SPECIALIX is not set
# CONFIG_SX is not set
# CONFIG_STALDRV is not set
# CONFIG_SYNCLINK is not set
# CONFIG_N_HDLC is not set
CONFIG_UNIX98_PTYS=y
CONFIG_UNIX98_PTY_COUNT=256
CONFIG_PRINTER=m
CONFIG_PRINTER_READBACK=y
# CONFIG_MOUSE is not set

#
# Joysticks
#
# CONFIG_JOYSTICK is not set
# CONFIG_QIC02_TAPE is not set
# CONFIG_WATCHDOG is not set
CONFIG_NVRAM=m
CONFIG_RTC=y
# CONFIG_AGP is not set

#
# Video For Linux
#
# CONFIG_VIDEO_DEV is not set
# CONFIG_DTLK is not set
```

```
#
# Ftape, the floppy tape device driver
#
# CONFIG_FTAPE is not set

#
# Filesystems
#
CONFIG_QUOTA=y
CONFIG_AUTOFS_FS=m
# CONFIG_ADFS_FS is not set
# CONFIG_AFFS_FS is not set
# CONFIG_HFS_FS is not set
CONFIG_FAT_FS=m
CONFIG_MSDOS_FS=m
CONFIG_UMSDOS_FS=m
CONFIG_VFAT_FS=m
CONFIG_ISO9660_FS=y
CONFIG_JOLIET=y
CONFIG_MINIX_FS=m
# CONFIG_NTFS_FS is not set
# CONFIG_HPFS_FS is not set
CONFIG_PROC_FS=y
CONFIG_DEVPTS_FS=y
# CONFIG_QNX4FS_FS is not set
CONFIG_ROMFS_FS=m
CONFIG_EXT2_FS=y
# CONFIG_SYSV_FS is not set
# CONFIG_UFS_FS is not set
# CONFIG_EFS_FS is not set

#
# Network File Systems
#
# CONFIG_CODA_FS is not set
CONFIG_NFS_FS=y
CONFIG_ROOT_NFS=y
CONFIG_NFSD=m
CONFIG_NFSD_SUN=y
CONFIG_SUNRPC=y
CONFIG_LOCKD=y
# CONFIG_SMB_FS is not set
# CONFIG_NCP_FS is not set

#
```

```
# Partition Types
#
CONFIG_BSD_DISKLABEL=y
CONFIG_MAC_PARTITION=y
CONFIG_SMD_DISKLABEL=y
CONFIG_SOLARIS_X86_PARTITION=y
CONFIG_UNIXWARE_DISKLABEL=y
CONFIG_NLS=y

#
# Native Language Support
#
CONFIG_NLS_CODEPAGE_437=m
CONFIG_NLS_CODEPAGE_737=m
CONFIG_NLS_CODEPAGE_775=m
CONFIG_NLS_CODEPAGE_850=m
CONFIG_NLS_CODEPAGE_852=m
CONFIG_NLS_CODEPAGE_855=m
CONFIG_NLS_CODEPAGE_857=m
CONFIG_NLS_CODEPAGE_860=m
CONFIG_NLS_CODEPAGE_861=m
CONFIG_NLS_CODEPAGE_862=m
CONFIG_NLS_CODEPAGE_863=m
CONFIG_NLS_CODEPAGE_864=m
CONFIG_NLS_CODEPAGE_865=m
CONFIG_NLS_CODEPAGE_866=m
CONFIG_NLS_CODEPAGE_869=m
CONFIG_NLS_CODEPAGE_874=m
CONFIG_NLS_ISO8859_1=m
CONFIG_NLS_ISO8859_2=m
CONFIG_NLS_ISO8859_3=m
CONFIG_NLS_ISO8859_4=m
CONFIG_NLS_ISO8859_5=m
CONFIG_NLS_ISO8859_6=m
CONFIG_NLS_ISO8859_7=m
CONFIG_NLS_ISO8859_8=m
CONFIG_NLS_ISO8859_9=m
CONFIG_NLS_ISO8859_14=m
CONFIG_NLS_ISO8859_15=m
CONFIG_NLS_KOI8_R=m

#
# Console drivers
#
CONFIG_VGA_CONSOLE=y
```

```
CONFIG_VIDEO_SELECT=y
# CONFIG_MDA_CONSOLE is not set
CONFIG_FB=y
CONFIG_DUMMY_CONSOLE=y
# CONFIG_FB_PM2 is not set
CONFIG_FB_VESA=y
# CONFIG_FB_VGA16 is not set
CONFIG_VIDEO_SELECT=y
# CONFIG_FB_MATROX is not set
# CONFIG_FB_ATY is not set
# CONFIG_FB_VIRTUAL is not set
# CONFIG_FBCON_ADVANCED is not set
CONFIG_FBCON_CFB8=y
CONFIG_FBCON_CFB16=y
CONFIG_FBCON_CFB24=y
CONFIG_FBCON_CFB32=y
# CONFIG_FBCON_FONTWIDTH8_ONLY is not set
# CONFIG_FBCON_FONTS is not set
CONFIG_FONT_8x8=y
CONFIG_FONT_8x16=y

#
# Sound
#
# CONFIG_SOUND is not set

#
# Kernel hacking
#
CONFIG_MAGIC_SYSRQ=y
```

B.3 Configuración HPL, HPL.dat

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
7            device out (6=stdout,7=stderr,file)
2           # of problems sizes (N)
650 1000  Ns
1            # of NBs
16     NBs
3            # of process grids (P x Q)
1 4 17       Ps
1 4 11      Qs
```

```
16.0            threshold
1               # of panel fact
0 1 2           PFACTs (0=left, 1=Crout, 2=Right)
1               # of recursive stopping criterium
2 4             NBMINs (>= 1)
1               # of panels in recursion
2               NDIVs
1               # of recursive panel fact.
0 1 2           RFACTs (0=left, 1=Crout, 2=Right)
1               # of broadcast
0 4 5               BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1               # of lookahead depth
1               DEPTHs (>=0)
2               SWAP (0=bin-exch,1=long,2=mix)
64              swapping threshold
0               L1 in (0=transposed,1=no-transposed) form
0               U in (0=transposed,1=no-transposed) form
1               Equilibration (0=no,1=yes)
8               memory alignment in double (> 0)
```

Apéndice C

Otros archivos

C.1 *skyvase.pov*, evaluación de rendimiento bajo POV-Ray

```
// Persistence Of Vision raytracer version 2.0 sample
// file.

// By Dan Farmer
//    Minneapolis, mn

//   skyvase.pov

// Vase made with Hyperboloid and sphere {, sitting on
// a hexagonal marble column.  Take note of the color
// and surface characteristics of the gold band around
// the vase.  It seems to be a successful combination
// for gold or brass.
//
// Contains a Disk_Y object which may have changed in
// shapes.dat


#include "shapes.inc"
#include "shapes2.inc"
#include "colors.inc"
#include "textures.inc"

#declare DMF_Hyperboloid = quadric {
/* Like Hyperboloid_Y, but more curvy */
   <1.0, -1.0,  1.0>,
   <0.0,  0.0,  0.0>,
   <0.0,  0.0,  0.0>,
```

```
   -0.5
}

camera {
   location <0.0, 28.0, -200.0>
   direction <0.0, 0.0, 2.0>
   up  <0.0, 1.0, 0.0>
   right <4/3, 0.0, 0.0>
   look_at <0.0, -12.0, 0.0>
}

/* Light behind viewer postion
   (pseudo-ambient light) */
light_source { <100.0, 500.0, -500.0> colour White }

union {
   union {
      intersection {
         plane { y, 0.7 }
         object { DMF_Hyperboloid
      scale <0.75, 1.25, 0.75> }
         object { DMF_Hyperboloid
      scale <0.70, 1.25, 0.70>
      inverse }
         plane { y, -1.0 inverse }
      }
      sphere { <0, 0, 0>, 1
    scale <1.6, 0.75, 1.6 >
    translate <0, -1.15, 0> }

      scale <20, 25, 20>

      pigment {
         Bright_Blue_Sky
         turbulence 0.3
         quick_color Blue
         scale <8.0, 4.0, 4.0>
         rotate 15*z
      }
      finish {
         ambient 0.1
         diffuse 0.75
         phong 1
         phong_size 100
         reflection 0.35
```

```
      }
   }

   sphere {  /* Gold ridge around sphere portion of
                vase*/
      <0, 0, 0>, 1
      scale <1.6, 0.75, 1.6>
      translate -7*y
      scale <20.5, 4.0, 20.5>

      finish { Metal }
      pigment { OldGold }
   }

   bounded_by {
      object {
         Disk_Y
         translate -0.5*y
     // Remove for new Disk_Y definition
         scale <34, 100, 34>
      }
   }
}

/* Stand for the vase */
object { Hexagon
/* Stand it on end (vertical)*/
   rotate -90.0*z
/* Turn it to a pleasing angle*/
   rotate -45*y
   scale <40, 25, 40>
   translate -70*y

   pigment {
      Sapphire_Agate
      quick_color Red
      scale 10.0
   }
   finish {
      ambient 0.2
      diffuse 0.75
      reflection 0.85
   }
}
```

```
union {
   plane { z, 50  rotate -45*y }
   plane { z, 50  rotate +45*y }

   pigment { DimGray }
   finish {
      ambient 0.2
      diffuse 0.75
      reflection 0.5
   }
}
```

C.2 *script* empleado para trazar *skyvase.pov*

```
#!/bin/bash
x-povray -i skyvase.pov +v1 -d +fn -x +a0.300 +r3 \
-q9 -w640 -h480 -mv2.0 +b1000
```

C.3 *Makefile* para automatizar la compilación de programas de multiplicación de matrices

```
PVMINCLUDE=/usr/share/pvm3/include
PVMLIBPATH=/usr/share/pvm3/lib/LINUX
PVMLIBS= -lpvm3 -lgpvm3

all: unimatrix matrix2 matrix1

matrix2.o: matrix2.c
gcc -c matrix2.c -I$(PVMINCLUDE)

unimatrix.o: unimatrix.c
gcc -c unimatrix.c

matrix.o: matrix.c
gcc -c matrix.c

matrix2: matrix2.o matrix.o
 gcc -o matrix2 matrix2.o matrix.o \
        -I$(PVMINCLUDE) -L$(PVMLIBPATH) $(PVMLIBS)

unimatrix: unimatrix.o matrix.o
 gcc -o unimatrix unimatrix.o matrix.o
```

```
matrix1: matrix1.c matrix.c
mpicc matrix1.c -o matrix1
clean:
rm -f *.o matrix1 matrix2 unimatrix
```

C.4 Tester.pl, automatización de pruebas

```
#!/usr/bin/perl

#numero maximo y minimo de nodos a utilizar
$minnodes=18;
$maxnodes=18;
@sizes_to_try=(20, 50, 100,
                200, 400, 500, 700,800);
#@sizes_to_try=(700,800);
# selector de comando
$whichcommand=0;

#constante para # de iteraciones
$hmconstant=100;

$file=$ARGV[0];

# abrir archivo para los resultados
print "Poniendo resultados en archivo $file\n";
if ($file eq ""){
    exit;
}


#para hacer autoflush y que el usuario no crea que esto
#se crasheo

$|=1;

#Este programa prueba todos los tamaños de matriz
#especificados en el arreglo sizes_to_try, para cada
#tamaño ejecuta el programa especificado al menos tres
#veces, aunque esto se decide segun el tamaño de la
#matriz.. Descartamos la primera ejecucion para
#eliminar lecturas falsas por cuestiones de caching, y
#las demás las promediamos, esta es la lectura que
#vamos a entregar.

```

```
38 sub get_time($$$){
39         my $dim=shift;
40         my $nodes=shift;
41         my $command=shift;
42 #comandos
43 $commands[0]="mpirun -np $nodes ./matrix1 -d $dim";
44 $commands[1]="\`pwd\`/matrix2 -d $dim";
45 $commands[2]="./unimatrix -d $dim";
46 $command=$commands[$whichcommand];
47         foreach $linea(`$command`){
48             print $linea;
49             if ($linea =~ m/wall clock time = (.*)/){
50                 return $1;
51             }
52         }
53         return 0;
54 }
55
56 # funcion que manda salida a pantalla y
57 #al archivo
58 sub imprimir($){
59     $l=1;
60     $toprint=shift;
61     print $toprint;
62 open F, ">>$file" or die (" no pude abrir $file");
63     print F $toprint;
64     close F;
65 }
66
67 # header
68 imprimir ("nodes\t");
69
70 foreach $size(@sizes_to_try){
71     imprimir ($size."\t");
72 }
73
74 imprimir ("\n");
75
76 imprimir ("tries\t");
77 foreach $size(@sizes_to_try){
78     $results_per_matrix=$hmconstant/$size;
79     if ($results_per_matrix < 3){
80         $results_per_matrix=3;
81     }
82     imprimir($results_per_matrix."\t");
```

```
83 }
84
85 imprimir ("\n");
86
87 for ($nodes=$minnodes;$nodes<=$maxnodes;$nodes++){
88     imprimir  ("$nodes\t");
89     foreach $size(@sizes_to_try){
90         $dimension=$size;
91         $totaltime=0;
92         $results_per_matrix=$hmconstant/$size;
93         if ($results_per_matrix < 3){
94             $results_per_matrix=3;
95         }
96         for ($iter=0;$iter<$results_per_matrix;$iter++){
97             print "start dim $size, iter $iter\n";
98             $thistime=get_time($dimension,
99                                $nodes,
100                                $command);
101            # si la lectura fue de cero, hay algun
102            #problema, notificarlo.
103            if ($thistime==0){
104                imprimir ("(!)");
105            }
106            # ignorar primera iteración para evitar
107            #broncas de
108            # pre-loading y cosas asi
109            if ($iter>0){
110                $totaltime+=$thistime;
111            }
112            print "end dim $size, iter $iter\n";
113        }
114        $avgtime=$totaltime/($results_per_matrix-1);
115        imprimir (sprintf( "%.4f\t",$avgtime));
116    }
117    imprimir "\n";
118 }
119
120 close F;
```

Bibliografía

[1] Bill Croft y John Gilmore, *RFC 951: Bootstrap Protocol (BOOTP)*, 1985

[2] Bill Nowicki, *RFC 1094: NFS: Network File System Protocol Specification*,1989

[3] Rekhter, Moskowitz, Karrenberg, de Groot, Lear, *RFC 1918: Address Allocation for Private Internets*, 1996

[4] R. Droms, *RFC 2031: Dynamic Host Configuration Protocol*, 1997

[5] Vasudevan, Nemkin, Gutshke, Yap, Kuhlmann, *Diskless Nodes HOWTO document for Linux*, 2001

[6] Daniel Quinlan, *Filesystem Hierarchy Standard*, 2000, http://www.pathname.com/fhs/2.1/fhs-toc.html

[7] Richard Stallman, *The GNU Project*, 1998, http://www.fsf.org/gnu/thegnuproject.html

[8] Amdahl, G.M. *Validity of the single-processor approach to achieving large scale computing capabilities*, AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, 1967.

[9] G.A. Geist, J.A. Kohl, P.M. Papadopoulos, *PVM and MPI: a comparison of features*, 1996

[10] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994

[11] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, 1994

[12] William Gropp, Ewing Lusk, *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*, Mississippi State University, 1994

[13] Leo Dagum, *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, 1997

[14] Phil Merkey, *Beowulf Introduction and Overview*,1998, http://www.beowulf.org/intro.html

[15] Daniel Ridge, Donald Becker, Phillip Merkey, Thomas Sterling Becker, *Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs*, Proceedings, IEEE Aerospace, 1997.

[16] R.A. Frazer, W.J. Duncan, A.R. Collar, *Elementary matrices and some applications to dynamics and differential equations*, Primera edición, séptima reimpresión, Cambridge University Press, 1960

[17] Alan Watt, *3D Computer Graphics*, Segunda edición, Addison Wesley, 1993

[18] Donald Hearn, M. Pauline Baker, *Computer Graphics - C Version*, Prentice Hall, 1997

[19] Jack J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software*, Depto. de Ciencias de la Computación, University of Tennessee, Knoxville, 2001

[20] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary, *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*, 2001, http://www.netlib.org/benchmark/hpl/

[21] *Especificaciones de la ILLIAC IV*, http://ed-thelen.org/comp-hist/illiac-iv.html

[22] *Free on-line dictionary of computing*, http://www.instantweb.com/foldoc

[23] *Whatis.com*, http://www.whatis.com

[24] *dictionary.com*, http://www.dictionary.com

[25] Entrevista a Seymour Cray, mayo 1995, http://americanhistory.si.edu/csr/comphist/cray.htm

[26] *Introduction to the T3D*, http://www.epcc.ed.ac.uk/epcc-tec/documents/intro-course/,Edinburgh Parallel Computing Centre

[27] Anuncio de prensa máquina petaflops, Applera Corporation, http://www.pecorporation.com/press/prccorp011901.html

[28] Anuncio de prensa ASCI White, LLNR, http://www.llnl.gov/asci/news/white_news.html

[29] Greg Almasi y Alan Gottlieb, *Highly Parallel Computing*, segunda edicion, Benjamin/Cummings Publishing, 1994

[30] *The official Red Hat Linux installation guide*, Red Hat, Inc, 2000.

[31] *Linux Kernel Version History*, http://www.memalpha.cx/Linux/Kernel/

[32] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, 1994

[33] *Just what is a Convex Exemplar?*, The ADN Connection, Publicación del UIC Academic Computing Center, University of Illinois at Chicago, Marzo/abril 1996.

[34] *Top 500 supercomputer sites*, http://www.top500.org

[35] The FreeBSD Documentation Project, *FreeBSD Handbook*, 2001, http://www.freebsd.org

[36] Equipo de desarrollo POV, *POV-Ray version 3.1g user's documentation*, http://www.povray.org, 1999