Centro de Investigación Científica y de Educación Superior de Ensenada

ESTUDIO Y SIMULACION DE LOS ALGORITMOS DE BALANCEO DE LA CARGA DE MOSIX Y "RATE OF CHANGE" EN UN CLUSTER DE PC's

TESIS
MAESTRIA EN CIENCIAS

ROMUALDO ZAYAS LAGUNAS

ENSENADA, B. C., NOVIEMBRE DEL 2000.

TESIS DEFENDIDA POR

Romualdo Zayas Lagunas

Y APROBADA POR EL SIGUIENTE COMITÉ

Dr. Andrei Tchernykh

Director del Comité

Dr. Isaac D. Scherson

Miembro del Comité

Dr. Oscar R. López Bonilla

Miembro del Comité

Dr. Ana I. Martínez García

Miembro del Comité

M. C.Jorge E. Preciado Velasco

Miembro del Comité

M. C. José Luis Briseño Cervantes

Jefe del Departamento de
Ciencias de la Computación

Dr. Federico Graef Ziehl

Director de Estudios de Posgrado

17 de Noviembre del 2000

Centro de Investigación Científica
y de Educación Superior de Ensenada

División de Física Aplicada
Departamento de Ciencias de la Computación
Posgrado en Ciencias de la Computación

ESTUDIO Y SIMULACIÓN DE LOS ALGORITMOS DE BALANCEO DE LA CARGA DE MOSIX Y "RATE OF CHANGE" EN UN CLUSTER DE PC's

Tesis

que para cubrir parcialmente los requisitos necesarios para
obtener el grado de Maestro en Ciencias presenta:

Romualdo Zayas Lagunas

Ensenada, Baja California, México. Noviembre del 2000

Resumen de la Tesis de Romualdo Zayas Lagunas, presentada como requisito parcial para la obtención del grado de MAESTRO EN CIENCIAS en CIENCIAS DE LA COMPUTACIÓN. Ensenada, Baja California, México. Noviembre del 2000

Estudio y Simulación de los Algoritmos
de Balanceo de la Carga de MOSIX y
"Rate of Change" en un cluster de PC's

Resumen aprobado por :

Dr. Andrei Tchernykh
Director de la tesis.

En un cluster de computadoras es necesario balancear la carga de trabajo entre todos los procesadores de tal forma que la ejecución de los procesos se lleve a cabo de manera eficiente. Para esto, existen algoritmos tradicionales como lo es el algoritmo que se utiliza en el Sistema Operativo Distribuido MOSIX y un nuevo algoritmo llamado "Rate of Change" recientemente propuesto en la Universidad de California-Irvine y del cual no existe una implantación real aún.

En este trabajo se presenta la construcción de un cluster de computadoras y las pruebas que se hicieron después de instalar MOSIX en cada uno de los nodos del cluster. También se presenta una simulación de los dos algoritmos de balanceo de la carga antes mencionados. Esta simulación tiene como fin estudiarlos, compararlos y poder plantear una nueva alternativa de balanceo.

Palabras clave: Balanceo de la carga, "Rate of Change", MOSIX, cluster de computadoras.

Abstract of the Thesis of Romualdo Zayas Lagunas, presented as a partial requirement for obtaining the degree of MASTER OF SCIENCE. Ensenada, Baja California, Mexico. November 2000.

Study and Simulation of
MOSIX's Load Balancing Algorithm and
Rate of Change Load Balancing Algorithm
in a PC Cluster

Abstract approved by :

Dr. Andrei Tchernykh
Thesis Director.

In a PC cluster it is necessary to balance the workload between all the processors, in order to execute the process in an efficient way. There are some traditional algorithms, like the algorithm used in the Distributed Operating System called MOSIX and a novel algorithm called "Rate of Change" recently proposed in the University of California-Irvine which still doesn't exist as a real implementation.

The present work deals the construction and tests of a PC cluster after installing MOSIX in each one of the cluster's nodes. Also, a simulation of both load balancing algorithms mentioned above is shown. The aim of this simulation is to study, compare and be able to raise a new alternative of load balancing.

Key words: Load Balancing, Rate of Change, MOSIX, PC clusters.

Agradecimientos

Quisiera agradecer a todas la personas e instituciones que de alguna forma contribuyeron para que pudiera culminar mis estudios de maestría.

A CONACYT por su apoyo económico sin el cual no hubiera sido posible la realización de estos estudios y a CICESE por haberme dado la oportunidad de ingresar a tan prestigiosa institución.

A mi esposa por su confianza, apoyo y ayuda.

A mis padres y a Dios por haberme dado la oportunidad de vivir, a mi también.

Contenido

Lista de Tablas

Lista de Figuras

Introducción

Los programas actuales requieren cada vez mayor poder de cómputo y rapidez en la generación de resultados. Existen varias alternativas para alcanzar el poder de cómputo deseado. Una primera alternativa es el uso de las "supercomputadoras", las cuales a su vez implican un "supercosto" en instalación, mantenimiento, actualización y licencias. Esta alternativa se convierte así, en una solución exclusiva para países desarrollados, para importantes centros de investigación y para grandes compañías internacionales. Otra alternativa es conectar varias computadoras de mediano poder de procesamiento por medio de una red local y utilizar un mecanismo de paso de mensajes para que puedan trabajar al mismo tiempo en la solución de un problema. Este tipo de sistemas es conocido como "Cluster de Computadoras" y está al alcance de cualquier compañía, departamento, universidad e incluso de algunas personas (Sterling et al, 99).

Una estrategia que puede complementar a la alternativa es el Balanceo de la Carga de Trabajo, que consiste en distribuir el trabajo equitativa y automáticamente entre todas las computadoras conectadas. La idea principal de esta estrategia es tener ocupados a todos los procesadores el mayor tiempo posible y que no haya tareas que estén esperando ejecución mientras existan procesadores ociosos (Xu y Lau, 1997).

La alternativa anterior parece muy atractiva y sencilla, pero la realidad es que no es tan simple hacer que una red de computadoras trabaje como un solo sistema y que además tenga un mecanismo que se encargue de balancear la carga de trabajo entre sus nodos, esto debido a que existen algunas dificultades físicas, técnicas y de metodologías.

Últimamente se ha apostado mucho a estas alternativas (Sterling et al, 1999) y se han dado los primeros pasos para lograr que un Cluster de Computadoras pueda ser visto como un solo sistema, eficiente, barato y sencillo de instalar y administrar (Barak et al, 1999). Estos pasos han consistido en desarrollar un software que sea el que se encargue de la administración de la red, de la administración de los procesos, del balanceo de la carga y otros (Rajkumar et al, 1999). Pero en realidad el software anterior tiene, entre otros, algunos inconvenientes como los siguientes:

- Los sistemas que permiten paso de mensajes y balanceo de la carga como PVM[1] (Geist et al, 1994) o MPI[2] (MPI Forum, 1995) hacen este trabajo de forma estática y no dinámica como podría requerirse en algunos problemas donde la creación y eliminación de procesos se hace dinámicamente (Schnor et al, 1996).

- La mayoría de los mecanismos de paso de mensajes son aplicaciones a nivel de usuario, lo que implica un uso poco eficiente de los recursos existentes (Baker et al, 1996).

- Los sistemas operativos que realizan el balanceo a nivel del núcleo o *kernel* son creados para un tipo de computadoras específicas, lo cual implica pérdida en la portabilidad (Petri et al, 1998), además de que son sistemas operativos comerciales en los cuales no se permiten modificaciones o adaptaciones al código para lograr un mejor rendimiento (Schnor et al, 1996).

- Para hacer un programa paralelo, a veces, es necesario aprender un lenguaje paralelo o la sintaxis particular de un mecanismo de paso de mensajes, lo cual es poco atractivo y por lo mismo no es tan fácil hacer programas portables (XPVM, 1994).

[1] *Parallel Virtual Machine*
[2] *Message Passing Interface*

En la Universidad Hebrea de Jerusalem han tratado de resolver, parcial o totalmente, algunos de los problemas anteriores. MOSIX es el nombre de la versión más reciente del Sistema Operativo Distribuido (SOD) que han construido en dicha Universidad. Este SOD permite a una red de computadoras aparecer ante los usuarios como un solo sistema, proporciona transparencia en el uso de recursos como lo son procesadores, la red, sistemas de archivos, etc., además realiza el trabajo de balanceo de la carga y migración de procesos a nivel del núcleo lo que hace que esta actividad sea eficiente, confiable y segura. El balanceo de la carga en MOSIX se lleva a cabo utilizando un algoritmo tradicional y probabilístico el cual trata de igualar el número de procesos en todos los procesadores, esto lo hace seleccionando aleatoriamente parejas de procesadores y migrando procesos del procesador que tiene más carga hacia el procesador que tiene menos carga. Este trabajo es completamente distribuido ya que todos los procesadores siguen los mismos algoritmos y la reducción de las diferencias de carga se hace independientemente por cualquier par de procesadores sin que exista un control central o maestro que se encargue de coordinar la actividad de balanceo y migración de procesos (Barak et al, 1999).

Durante los últimos años se han propuesto distintas estrategias o algoritmos de balanceo de la carga, las cuales tienen como objetivo realizar esta actividad de forma eficiente y así contribuir en un incremento en el rendimiento de un sistema. Este es el caso del algoritmo de Balanceo de la Carga *Rate of Change* (RoC-LB) que fue propuesto en la Universidad de California - Irvine y que se presenta como una propuesta totalmente innovadora que rompe con los esquemas tradicionales de balanceo de la carga de trabajo. Este algoritmo se basa en información local, como lo es la razón de cambio de la carga, para tomar decisiones que le permitan encontrar un equilibrio global y un rendimiento aceptable.

El presente trabajo es parte de un proyecto en el cual se quiere contribuir al avance y

evolución de los sistemas distribuidos al construir un SOD. En este proyecto se mezclan ideas como Balanceo de la Carga, *Rate of Change*, MOSIX, migración de procesos y se plantea una nueva alternativa que permita un incremento del rendimiento de un sistema de cómputo, en particular de un cluster de computadoras personales y principalmente que el cluster de computadoras presente la imagen de un solo sistema (ISS) hacia los usuarios.

La construcción de un SOD es un trabajo de gran tamaño y complicación, en el cual se tienen que involucrar conceptos no sólo de balanceo de la carga y migración de procesos, sino que también es necesario conocer políticas de administración de procesos, de memoria, de archivos, protocolos de red, conocimiento del hardware sobre el cual correrá el SOD y varios conceptos más (Tanenbaum, 1993). En el caso de que la construcción del SOD se base en la modificación de un sistema operativo existente, también se tendría que analizar el funcionamiento interno de dicho sistema y plantear una reestructuración y modificación.

La dimensión del proyecto de construcción de un SOD es bastante grande, de aquí que es necesario dividirlo en diferentes etapas. El presente trabajo representa la primera fase y tiene como objetivo general crear las bases y la infraestructura para poder continuar con el desarrollo del proyecto.

Los objetivos específicos que se persiguen en el presente trabajo son los siguientes:

1. Construir un cluster con computadoras personales el cual tenga como sistema operativo a Linux y que permita procesamiento en paralelo utilizando PVM y/o MPI.

2. Estudiar y comparar los algoritmos de balanceo de la carga de MOSIX y *Rate of Change.*

3. Simular los algoritmos de balanceo de la carga de MOSIX y *Rate of Change* en

un cluster de computadoras.

Estructura de la Tesis

Este documento está compuesto de 7 capítulos. Los primeros 5 capítulos describen las áreas, teorías y conceptos que se usaron para el desarrollo del trabajo. Los capítulos finales muestran el trabajo práctico que se realizó para verificar las teorías presentadas en los primeros capítulos y los resultados obtenidos de dicho trabajo.

En el capítulo I se describen a los Sistemas Distribuidos y un caso particular como lo son los clusters de computadoras, sistemas en los cuales está basado el trabajo.

En el capítulo II se presenta una metodología para comparar sistemas de cómputo, dicha comparación se hace por medio de programas especiales llamados *Benchmarks*. Estos programas también permiten verificar el funcionamiento de algún sistema y en particular los usamos para probar el funcionamiento del Balanceo de la Carga en MOSIX.

El Balanceo de la Carga de Trabajo, punto central en este trabajo, es presentado en el capítulo III. En este capítulo se muestra en qué consiste y los pasos necesarios para realizar dicha operación. Así mismo en los capítulos IV y V se describen los algoritmos de balanceo de la carga *Rate of Change* y de MOSIX, respectivamente.

La descripción de la simulación de los dos algoritmos anteriores se presenta en el capítulo VI y se muestran las funciones más importantes en dicha simulación.

Finalmente, el capítulo VII contiene una descripción del escenario en el cual se hacen los experimentos, se presentan los resultados obtenidos en la simulación y los resultados obtenidos al ejecutar un conjunto de *Benchmarks* en un cluster de computadoras.

Capítulo I

Sistemas Distribuidos

I.1 Introducción

Desde la década de los 40's, cuando comenzó la era de la computadora moderna, hasta cerca de 1985, las computadoras eran grandes y caras. Incluso las minicomputadoras costaban cientos de miles de dolares cada una. Como resultado de esto, la mayoría de las organizaciones tenían pocas computadoras y por carecer de una forma de conectarlas, éstas operaban en forma independiente entre sí.

Sin embargo, a partir de la mitad de los 80's, dos avances tecnológicos comenzaron a cambiar esta situación. El primero fue el desarrollo de poderosos microprocesadores. Muchos de ellos tenían el poder de cómputo de una computadora grande de ese tiempo, pero sólo costaban una fracción del precio de ésta.

El segundo desarrollo fue la invención de las redes de área local (LAN) de alta velocidad. Este desarrollo permitió conectar docenas, e incluso cientos de máquinas de tal forma que pudiesen transferir información entre ellas en muy poco tiempo.

El resultado neto de estas 2 tecnologías es que hoy en día no sólo es posible, sino fácil, reunir sistemas de cómputo compuestos por un gran número de procesadores conectados mediante una red de alta velocidad. Estos sistemas son los ahora conocidos Sistemas Distribuidos, los cuales prometen un rendimiento mayor o igual que sus

contrapartes centralizados.

Sólo que para obtener ese rendimiento existe un problema: el software para los Sistemas Distribuidos. Este software es radicalmente distinto al de los sistemas centralizados de un solo procesador y no sólo es distinto, sino que aún está en una etapa de desarrollo y de perfección.

En este capítulo daremos una definición de Sistema Distribuido, una clasificación de los Sistemas Distribuidos y finalmente describiremos un "cluster de computadoras" como un caso particular de los Sistemas Distribuidos.

I.2 Definición de un Sistema Distribuido

En la bibliografía existen diversas definiciones de lo que es un Sistema Distribuido. En algunas definiciones hacen énfasis en que el sistema está compuesto de varias computadoras y que no se tiene memoria compartida, pero esta definición deja fuera a las computadoras con multiprocesadores que en otras definiciones sí están contempladas. Otras definiciones hacen énfasis en el medio físico de comunicación, es decir, que la comunicación tiene que ser a través de medios de alta velocidad, líneas telefónicas, cable coaxial, etc. Algunas más toman en cuenta el nivel de acoplamiento que el sistema tiene, esto es, que si el sistema es débilmente o fuertemente acoplado (Melo, 1998).

Para nuestros propósitos, plantearemos una definición sencilla y a la vez muy general, que agrupa a las anteriores, esta es:

Sistema Distribuido: Es una colección de elementos que se comunican a través de una red de interconexión. Cada elemento está compuesto por uno o varios procesadores y por una o varias memorias (Melo, 1998).

Tomando en cuenta la definición anterior, en la cual tenemos agrupados a un conjunto de sistemas de cómputo, es conveniente hacer una clasificación de estos para entender con que tipo de sistema y cuáles son las características del sistema del cual

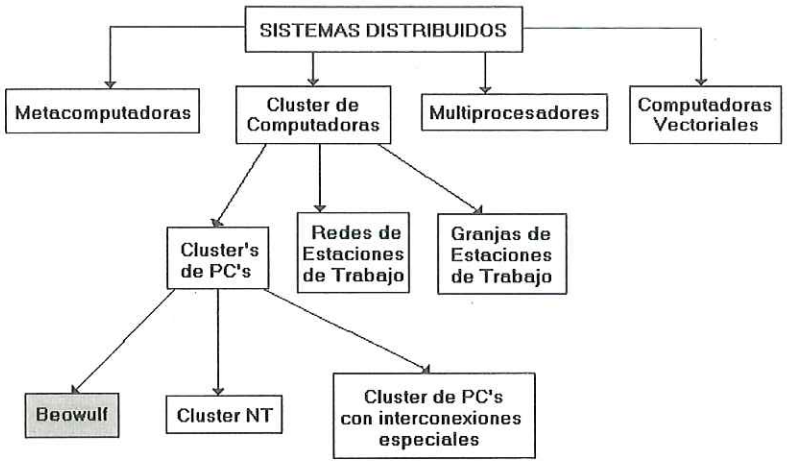

Figura 1: Clasificación de los sistemas distribuidos por su nivel de acoplamiento

se estará hablando en este documento.

I.3 Taxonomía de los Sistemas Distribuidos

Es posible crear diferentes taxonomías jerárquicas con los Sistemas Distribuidos, esta clasificación va a depender del punto de comparación, por ejemplo: la memoria, el tipo de procesadores, el nivel de acoplamiento, etc.

A continuación veremos una clasificación de los Sistemas Distribuidos, la cual está basada en el nivel de acoplamiento que tienen los mismos (Sterling et al, 1999), describiremos brevemente cada uno de ellos y nos enfocaremos en un tipo de Sistema Distribuido particular: *Cluster de Computadoras*, sistema en el cual está basado este trabajo y que más adelante describiremos con detalle.

En la figura 1 podemos ver la clasificación que se ha hecho de los Sistemas Distribuidos. En el nivel superior de esta taxonomía se encuentran, desde el más débilmente acoplado hasta el más fuertemente acoplado:

- Metacomputadoras.
- Clusters de Computadoras.
- Multiprocesadores.
- Computadoras Vectoriales.

Las Metacomputadoras aprovechan distintos sistemas de cómputo los cuales están separados y conectados por medio de redes de área amplia y cuya latencia[1] es de algunas decenas de milisegundos (Sterling et al, 1999).

Los Clusters de Computadoras, que normalmente son una colección de elementos de procesamiento independientes, están ubicados en el mismo departamento o dominio administrativo y su latencia está entre 10 a 100 microsegundos (Sterling et al, 1999).

Los Multiprocesadores están formados de varios procesadores altamente acoplados y vistos como una sóla computadora, con latencias entre medio microsegundo y 5 microsegundos (Sterling et al, 1999).

Finalmente las Computadoras Vectoriales que realizan cálculos en paralelo a nivel de hardware y cuya latencia está en el órden de las decenas de nanosegundos (Sterling et al, 1999).

En un siguiente nivel de la taxonomía encontramos que los clusters de computadoras se pueden agrupar en tres clases:

- Redes de Estaciones de Trabajo.
- Granjas de Estaciones de Trabajo.
- Clusters de PC's.

[1] El tiempo que se tarda en enviar y recibir un dato.

Las redes de estaciones de trabajo (NOWS) están formadas precisamente con estaciones de trabajo dedicadas a alguna tarea específica.

A diferencia de las redes de estaciones de trabajo, las granjas de estaciones de trabajo son sistemas formados con computadoras de escritorio, las cuales tienen un propietario o usuario y son conectadas o agrupadas para trabajar juntas cuando el propietario no las está utilizando.

Los clusters de PC's están formados con computadoras personales con procesadores 386, 486 o Pentium y con sistemas operativos como DOS, Windows NT, Linux, etc.

En el último nivel de la taxonomía podemos hacer una subdivisión de los clusters de PC's en tres clases: Beowulf, el cual será descrito más adelante; Cluster NT, usando el sistema operativo Windows NT de Microsoft y los clusters de PC's que utilizan interconexiones especiales como SCI[2] o como el proyecto SHRIMP de la Universidad de Princeton que proporciona soporte a nivel de hardware para dar una imagen de memoria distribuida compartida.

Dado que en el medio es muy común llamar solamente "cluster" a un "cluster de computadoras", nosotros también adoptaremos tal convención y con los dos nombres nos estaremos refiriendo al mismo concepto.

A continuación describiremos con más detalle el tipo de sistema que se utilizará en este trabajo.

I.4 Beowulf

Un Beowulf es una colección de computadoras personales interconectadas por un medio físico ampliamente disponible y trabajando con un sistema operativo libre como Linux. Los programas que se ejecutan en un Beowulf son usualmente escritos en C o

[2]Por sus siglas en inglés "Scalable Coherent Interface"

Fortran. Estos programas utilizan un mecanismo de paso de mensajes para realizar cómputo paralelo, normalmente este mecanismo es libre y puede ser PVM, MPI, etc (Sterling et al, 1999).

Actualmente los Beowulfs se utilizan en universidades, laboratorios nacionales, grupos industriales, centros de investigación, dependencias de gobierno y pequeñas compañías. Esta red de computadoras es simplemente el mejor de los sistemas tomando como punto de comparación la razón "costo/rendimiento" que ofrecen los sistemas de cómputo (Sterling et al, 1999).

Los Beowulf no se vuelven obsoletos con la aparición de un nuevo sistema o de un avance tecnológico, sino al contrario se ven beneficiados directamente porque dicho avance puede ser incorporado al sistema con la posibilidad de obtener un mejor rendimiento. Además, un Beowulf no tiene una arquitectura específica como pasa con muchos tipos de computadoras. Por tanto, existe un amplio rango de opciones entre procesadores, memoria y hardware que en general pueden ser utilizados para la construcción de un Beowulf. Esto nos lleva a otra característica de estos sistemas: la escalabilidad. Esto quiere decir que existe la posibilidad de conectar 1, 2, 3 o cientos de PC's en un Beowulf y el rendimiento no se verá disminuido, sino por el contrario se verá un aumento sustancial.

Como mencionamos antes, estos sistemas están compuestos por computadoras completas e independientes (nodos), que necesitan un medio de interconexión, que tienen un sistema operativo y que necesitan un modelo de programación paralela. En las siguientes secciones se describen cada una de estas características.

I.4.1 Nodos

En todos los clusters que mencionamos antes, el bloque fundamental de construcción es llamado "nodo", el cual es un sistema de cómputo independiente, con hardware completo y sistema operativo que puede manejar la ejecución de un programa de usuario

e interactuar con otros nodos del cluster.

Los nodos del cluster, pueden ser tan pequeños como una computadora de escritorio, costando mil dolares, hasta un sistema multiprocesador simétrico, que cuesta varios millones de dolares. Para un Beowulf, un nodo es usualmente una computadora personal.

I.4.2 Interconexión

Después del nodo, el subsistema más importante en un Beowulf es el sistema de interconexión de red. Los Beowulfs a veces, no siempre, emplean los protocolos *Ethernet* y TCP/IP desarrollados y vendidos para redes de área local. Los primeros Beowulfs usaron tarjetas *Ethernet* a 10 Mb, pero hoy se usa más *Ethernet* a 100 Mb o *Fast Ethernet*. Un mayor ancho de banda se ha logrado con *Myrinet* y *Gigabit Ethernet*, los cuales alcanzan índices de 1 Gb, con el único detalle de que son opciones muy caras que llegan a costar la mitad del precio total de todo el sistema. En contraste una tarjeta *Fast Ethernet* cuesta menos de 50 dólares.

La red es el medio por el cual los procesadores se comunican. Las redes de trabajo han evolucionado considerablemente desde el pasivo uso del cable coaxial que alguna vez conectó un número pequeño de computadoras. El cable coaxial fue reemplazado por activos dispositivos llamados *hubs* que pueden conectar hasta 16 nodos a un, relativamente, bajo costo pero con la diferencia de que es capaz de difundir el tráfico a todos los nodos. Los enrutadores de paquetes (*switches*) son más complejos, contienen una lógica que identifica el destino de cada paquete y envía el paquete al puerto apropiado. Así, cada nodo conectado a un enrutador de paquetes tiene el ancho de banda completo (100 Mb) disponible para su propia comunicación.

I.4.3 Sistemas Operativos

El sistema operativo es el ambiente de software que garantiza el acceso al procesador

y a la memoria, proporciona servicios a los programas de aplicación, presenta una interfaz al usuario y administra las interfaces externas a dispositivos externos. Los sistemas operativos más populares para una PC son los de Microsoft, pero existen otras opciones como Linux, FreeBSD y varias versiones comerciales de Unix. Los Beowulfs usan un sistema operativo libre y Linux es el más popular y usado.

Linux es un sistema operativo monolítico, multitareas, que soporta memoria virtual y que cumple con los estándares POSIX. Está disponible sin costo en Internet o a bajo costo en disco compacto. Linux viene con el código fuente completo y es una poderosa herramienta de investigación en ciencias de la computación. Linux soporta el ambiente de programación completo GNU incluyendo gcc y g++, g77, gdb, gprof y emacs, así como herramientas de compilación como lex, yacc y bison. Proporciona una interfaz de usuario X Windows y manejadores de ventanas. Soporta acceso remoto, llamadas a procedimientos y comunicación entre procesos remotos por medio de un mecanismo virtual (*socket*). Linux proporciona un completo sistema de archivos y un sistema de archivos distribuidos (*Network File System*).

I.5 Paso de Mensajes

Para ejecutar una aplicación de manera correcta en un Beowulf, ésta debe contener paralelismo, esto es, debe ser posible dividir un trabajo en varias partes, de tal forma que varios procesadores puedan unir sus esfuerzos y ejecutarlas al mismo tiempo para poder alcanzar una solución más rápido. Idealmente, las aplicaciones deben ser planeadas y estructuradas de tal forma que se puedan agregar elementos de procesamiento y el tiempo total de ejecución no se vea decrementado. Para escribir aplicaciones que tengan esta propiedad se debe utilizar un modelo de programación paralela. El modelo más usado es el de paso de mensajes o más formalmente, "Comunicación entre Procesos Secuenciales".

En este modelo, un proceso se ejecuta en cada nodo del sistema. El proceso realiza operaciones, primero sobre datos locales al nodo, es decir, datos que están almacenados en la memoria del nodo o en el disco del nodo. Para que el proceso continúe su ejecución puede requerir datos de otras partes del cálculo llevándose a cabo en otros procesadores. En este caso se tiene que enviar mensajes solicitando la información requerida, de aquí que el intercambio de datos y la sincronización entre procesos concurrentes es hecha de esta manera. La información sólo es compartida entre procesos o procesadores por medio de un intercambio explícito de mensajes.

Varias interfaces o ambientes de paso de mensajes para programas concurrentes se han desarrollado hasta la fecha. De todos estos, PVM y MPI son los más utilizados (Sterling et al, 1999).

MPI es el resultado de un gran esfuerzo para proporcionar un ambiente de programación estandarizado y está formado por una conjunto de rutinas que permiten el paso de mensajes para facilitar la comunicación entre los procesadores (MPI Forum, 1995).

PVM es un ambiente de software que permite que una colección de computadoras heterogéneas puedan ser usadas como un sistema de cómputo paralelo y flexible (Geist et al, 1994).

En los dos ambientes, PVM y MPI, el usuario puede hacer sus aplicaciones en lenguajes de programación como C, C++ y Fortran, donde los programas contienen un llamado a las rutinas que permiten el paso de mensajes entre procesos, esto con el fin de intercambiar datos y sincronizar procesos. Otra característica de estos ambientes de programación es que están disponibles en la red y son libres.

Además de estos dos ambientes de programación existe más software que puede ser utilizado para desarrollar aplicaciones de gran rendimiento y que explotan la característica de estar trabajando en un ambiente distribuido.

Capítulo II

Rendimiento en Sistemas de Cómputo

II.1 Introducción

Estamos interesados en el rendimiento que un cluster puede ofrecer utilizando un ambiente de paso de mensajes y una estrategia de balanceo de la carga de trabajo. Pero, para medir "algo" primero hay que saber lo que significa ese "algo", en nuestro caso tenemos que saber qué es el rendimiento para poder medirlo en nuestro sistema.

II.2 Definición de Rendimiento

Ahora estableceremos una definición general de lo que es el rendimiento (OECD, 1974; Miller et al, 1996).

Rendimiento: Es la forma en que un sistema realiza cierta actividad y existe una forma de medir el desempeño de dicha actividad. Además el resultado de la medición sirve como punto de comparación entre varios sistemas.

Los siguientes son ejemplos de lo que podemos medir y tomar como punto de comparación:

- Tiempo de ejecución de un programa.

- Número de operaciones que un procesador realiza en una unidad de tiempo (KIPs, MIPs, GIPs, MFLOPs, GFLOPs, TERAFLOPs).
- Tiempo que tarda un procesador en traer un dato de la memoria (latencia).
- Número de bits que se reciben en una unidad de tiempo (ancho de banda).
- Número de ciclos de reloj por segundo (MHertz).
- Número de programas que un sistema puede ejecutar por unidad de tiempo (Throughput Rate).
- Precio del sistema ($$).

En nuestro caso estamos interesados en medir el tiempo que se tarda un cluster en ejecutar una aplicación y el número de operaciones que se realizan, tomando en cuenta que en un sistema distribuido hay retardos debido a la comunicación entre procesos, la sincronización y por la distribución de la carga.

II.3 Razones para Evaluar el Rendimiento

Existen tres razones principales por las cuales se mide el rendimiento en un sistema de cómputo, estas razones son:

- Selección.
- Proyección.
- Modificación de Sistemas.

La selección es la razón más común para evaluar un sistema. Cuando alguien necesita comprar una computadora, el primer paso para esto es saber bien cuáles

son sus necesidades y posibilidades, a continuación tiene que establecer una forma de evaluación o comparación entre las distintas opciones. Finalmente tiene que escoger la mejor opción según su evaluación.

La proyección es la razón por la cual los fabricantes de computadoras necesitan evaluar el rendimiento. Cuando un fabricante ha diseñado una nueva computadora o ha hecho una modificación al procesador, a la memoria o a algún subsistema, entonces necesita conocer el rendimiento obtenido y compararlo con el rendimiento de los sistemas anteriores, de esta forma el fabricante sabrá si el nuevo producto es mejor que el producto anterior.

Cuando se quiere incrementar el rendimiento, a veces es posible hacer una modificación a nuestro sistema. Por ejemplo, conectar varias computadoras para que trabajen en paralelo, distribuir y balancear la carga de trabajo en las computadoras, cambiar de algoritmo de planificación, etc. Una vez hecha la modificación necesitamos medir el rendimiento para verificar el que paso con nuestro sistema después de la modificación. Cuando alguien propuso utilizar varias computadoras, conectarlas y hacer que todas cooperaran en la solución de un problema, se tuvo que evaluar el rendimiento y verificar si se lograba un incremento en dicho rendimiento (Miller et al, 1996). Éste es nuestro caso al tratar de incrementar el rendimiento en un cluster por medio de la implantación de un algoritmo de balanceo de la carga.

II.4 Formas de Evaluar el Rendimiento

Desde hace varios años se han tenido la necesidad de evaluar o medir el rendimiento de un sistema de cómputo. Para esto se han propuesto diversas estrategias que van desde los mediciones de alguna actividad en una computadora (tiempo de almacenamiento, ciclo de reloj, etc.), pasando por los *benchmarks* (que actualmente es la

estrategia más utilizada) hasta los vectores de rendimiento que garantizan sus resultados al combinar, por un lado, la solución de un sistema de ecuaciones y, por el otro lado, los tiempos obtenidos al medir ciertas actividades del sistema (Krishnaswamy, 1995).

A continuación describiremos algunos de los benchmarks más conocidos y utilizados en computadoras personales con sistema operativo del tipo Unix.

II.4.1 Benchmarks

Un benchmark es una programa o conjunto de programas que se ejecutan sobre diferentes computadoras para medir el rendimiento. Con estos resultados se puede hacer una comparación entre varias computadoras (OECD, 1974). De todos los benchmarks que existen, los que se describen a continuación son de los más conocidos y utilizados:

SIM: Programa que trabaja con enteros y que compara segmentos de DNA para encontrar similitudes.

Fhourstones (c4): Programa que trabaja con enteros y que busca soluciones al juego "connect-4".

Dhrystone: Programa sintético que trabaja con enteros y que proporciona un índice de MIPS (Millones de Instrucciones Por Segundo).

Nsieve: Programa el cual genera números primos basándose en la "Criba de Eratóstenes" y usando tamaños de arreglos desde 8 Kb hasta 1 Mb.

Heapsort: Es un programa que trabaja con enteros y usa el algoritmo *HeapSort* para ordenar un arreglo de enteros de hasta 2 Mb de tamaño. El resultado de este benchmark es un índice de MIPS.

Hanoi: Es un programa que trabaja con enteros y que resuelve el problema de "Las Torres de Hanoi" usando llamadas recursivas a funciones.

Queens: Programa que trabaja con enteros que resuelve el problema de las "14 Reinas", esto es, el programa encuentra todas las formas de colocar 14 reinas en un tablero de ajedrez de 14 × 14 de tal forma que ninguna reina pueda atrapar a otra.

Flops: Programa que calcula el índice de MFLOPS (Millones de Instrucciones de Punto Flotante) utilizando "mixes" de instrucciones.

FFT: Es un programa que calcula la Transformada Rápida de Fourier en 131,072 puntos.

TFFTDP: Es otro programa que calcula la Transformada Rápida de Fourier de 32 hasta 262,144 puntos.

Matrix Multiply: Este programa contiene 9 diferentes algoritmos para hacer una multiplicación de una matriz de 500 × 500. Los resultados muestran los efectos que tiene el cache.

BJ: Este programa sigue las reglas del juego "Las Vegas Strip BlackJack y es usado como prueba para sistemas de cómputo.

Shuffle: Este programa revuelve 4 paquetes de cartas de pocker usando un generador de números aleatorios.

Los benchmarks anteriores se encuentran disponibles en muchos sitios en Internet y están a la disposición de quien quiera tomarlos, instalarlos y ejecutarlos en su sistema. Un sitio donde pueden ser obtenidos es el siguiente:

`ftp://ftp.nosc.mil/pub/aburto`

Los siguientes, son otros benchmarks que también son de los más conocidos y también se encuentran disponibles en Internet.

BYTE: Este es un benchmark para ejecutarse en sistemas Unix y consiste de 8 subprogramas los cuales pueden ser divididos en dos clases: pruebas de bajo nivel (aritmética, llamadas al sistema, operaciones de memoria, operaciones de disco y dhrystone) y pruebas de alto nivel (operaciones con bases de datos, carga del sistema y pruebas al compilador de C instalado). Este benchmark se encuentra en:

`http://www.silkroad.com/bass/linux/linux-bm-java.html`

Linpack: Este programa se encarga de resolver un sistema de ecuaciones lineales y una matriz densa. Este benchmark se caracteriza por tener un número muy grande de instrucciones de punto flotante. Las rutinas usan algoritmos orientados a columnas y se apoyan en el conjunto de programas BLAS (Basic Linear Algebra Subprograms) del nivel 1, es decir, operaciones con arreglos unidimensionales o vectores (Dongarra, 1999).

II.5 Alternativas para Incrementar el Rendimiento

Durante la década de los 80's, la gente creía que el rendimiento podría ser incrementado creando procesadores más rápidos y más eficientes, en una palabra más "poderosos". Pero este incremento está limitado por procesos físicos de transmisión de señales y de otras dificultades tecnológicas. Entonces, no se puede esperar que el incremento del rendimiento dependa sólo de la posibilidad de reducir el tiempo de operación de los circuitos integrados.

Actualmente la idea de hacer procesadores muy veloces está siendo desplazada por el concepto de "clustering", el cual significa ligar dos o más computadoras para resolver un problema de cómputo. La meta de esto es desarrollar una infraestructura (hardware y software) que ofrezca un mejor rendimiento y que además los usuarios no necesiten saber en qué computadora están trabajando, esto es, hacer que múltiples computadoras trabajen juntas como un solo sistema y que además el trabajo sea transparente a los usuarios.

Alcanzar esta meta no es tarea fácil ya que existen muchos factores, tanto de tecnologías como de metodologías, que pueden hacer más difícil el camino hacia dicha meta, es decir, factores como la latencia en una computadora de memoria distribuida, los cuellos de botella en una computadora de memoria compartida, situaciones donde el número de tareas a ejecutar es mayor que el número de procesadores lo cual implica usar técnicas de balanceo de la carga (dinámicas o estáticas) que funcionan bien sólo para casos particulares y no para cualquier caso, el bajo nivel de los lenguajes paralelos que en realidad son sólo extensiones de lenguajes secuenciales y no lenguajes paralelos como tal y muchos factores más (Parallel C, 1989).

Es por eso que las técnicas de balanceo de la carga surgen como una alternativa para librar algunos de los obstáculos que se presentan en el camino hacia la meta planteada y mejorar el rendimiento o tiempo de ejecución de una aplicación en un sistema distribuido, en particular, en un cluster. El balanceo de la carga de trabajo, visto como una alternativa para incrementar el rendimiento, es el tema del siguiente capítulo.

Capítulo III

Balanceo de la Carga de Trabajo

III.1 Introducción

El poder de las computadoras se ha incrementado en los últimos años, pero la habilidad para aprovechar este poder no se ha incrementado debido a que estamos tan acostumbrados a resolver problemas secuencialmente, sin importar si la computadora es secuencial o paralela. Entonces podemos decir que hay una gran desventaja del hardware con respecto al software. Esta desventaja debe ser cubierta con software avanzado. Se han dedicado muchos esfuerzos por parte de algunos investigadores en todo el mundo para desarrollar técnicas de software distribuido o paralelo. Estos investigadores comparten la meta común de que el uso de las computadoras paralelas sea mucho más fácil y que permita al usuario explotar completamente su poder de cómputo. En otras palabras, tratan de dar al usuario la "Imagen de un Solo Sistema". Una técnica esencial es el Balanceo de la Carga de Trabajo (BCT), la cual se describirá en este capítulo.

En este capítulo hablaremos sobre el BCT en Sistemas Distribuidos en donde cada procesador tiene su propio espacio de direcciones y tiene que comunicarse con otros procesadores pasando mensajes a través de una red de interconexión punto a punto. Para ser más específicos, hablaremos del balanceo de la carga en un cluster de PC's

como el que se describió en la sección I.4. Los usuarios de estos sistemas no sólo necesitan escribir programas en lenguajes, con los que no están familiarizados y que a veces son poco claros, sino que también necesitan adaptar o modificar sus programas para que puedan ejecutarse sobre estos sistemas. Esto implica que la calidad del programa estará en función del conocimiento que el usuario tenga sobre la arquitectura de dicho sistema (Russ et al, 1997) .

La situación anterior ha provocado que se ponga mucha atención y se dediquen muchos esfuerzos a desarrollar técnicas de software avanzado para computadoras paralelas. Nuevos estándares de programación tales como *High-Performance Fortran* (HPF), PVM, MPI y compiladores que permiten una paralelización automática representan grandes pasos para resolver tal situación.

El BCT está dentro de los temas de mayor consideración en la programación paralela, en la paralelización automática y en el manejo de recursos en tiempo de ejecución. El BCT tiene como meta incrementar el rendimiento de los Sistemas Distribuidos repartiendo las tareas de los usuarios, apropiadamente, entre los procesadores. Desde un enfoque tradicional, el BCT trata de igualar las cargas de trabajo de los procesadores de manera automática durante la ejecución de un programa paralelo, o de varios programas secuenciales, de tal forma que el tiempo de ejecución o el tiempo de finalización de todos los programas sea minimizado.

III.2 El Problema del Balanceo de la Carga de Trabajo

Un programa paralelo está compuesto de varios procesos en donde cada uno realiza una o más tareas definidas en el programa. La creación de un programa paralelo involucra, primero, descomponer todo el trabajo computacional en tareas y luego asignar esas tareas a cada uno de los procesadores. Los pasos de descomposición y asignación,

juntos, son a veces llamados etapa de partición.

El problema de balancear la carga en programas paralelos consiste en desarrollar algoritmos de partición y mapeo con el propósito de lograr sus respectivos objetivos de optimización.

Por si fuera poco, los sistemas actuales de cómputo permiten que varios usuarios estén conectados y trabajando al mismo tiempo. Lo anterior complica la situación porque ahora no sólo hay que pensar en distribuir la carga durante la etapa de creación del programa, sino que hay que evitar que todos los usuarios ejecuten sus procesos en el mismo procesador, esto es muy difícil debido a que los usuarios ejecutan aplicaciones secuenciales y/o paralelas de manera aleatoria. Aquí, la única alternativa es que el sistema operativo sea el encargado de distribuir los procesos en los distintos procesadores.

Los usuarios generalmente inician sus procesos en su computadora principal o donde se conectaron al sistema. La creación aleatoria de procesos puede causar que algunas computadoras se sobrecarguen mientras que otras pueden estar ociosas o con poca carga. De tal forma que ahora el problema de compartir y balancear la carga implica desarrollar algoritmos de organización y administración de procesos, que transfieran de manera automática los procesos de computadoras sobrecargadas a computadoras con poca carga. Aquí la meta principal es asegurar que ningún procesador esté ocioso mientras que haya procesos esperando a ser atendidos en otros procesadores.

Por lo tanto, podemos concluir que el Balanceo de la Carga de Trabajo, tiene como objetivo incrementar el rendimiento de los Sistemas Distribuidos al igualar o nivelar las cargas de trabajo de los procesadores. Además en (Barak et al, 1993) se muestra que el BCT está influenciado por 3 factores:

- El ambiente en el cual se desea balancear la carga.
- Las herramientas de balanceo disponibles.

- La naturaleza de la carga.

Cada factor está influenciado por al menos uno de los otros factores. Por ejemplo, la complejidad de una herramienta de balanceo depende directamente del ambiente en el cual se esté trabajando.

El ambiente de un Sistema Distribuido está definido por factores como: la arquitectura de los procesadores del sistema, el tipo de recursos compartidos por los procesadores y la forma de interconexión de los procesadores.

Las herramientas de balanceo son procedimientos o programas que facilitan el BCT. Existen dos tipos de herramientas: las herramientas de información y las herramientas para procesos. Las primeras herramientas recolectan información de la disponibilidad y estado de los recursos, estas herramientas dependen del ambiente ya que en sistemas de memoria compartida se puede tener una tabla globales con la información de la carga de los procesadores, mientras que en un ambiente de paso de mensajes, se tendrán que usar tablas locales e independientes para cada procesador. Las herramientas de procesos ayudan en la transferencia de procesos entre los procesadores. En un sistema del tipo UNIX, la única herramienta para administrar los procesos es el núcleo mismo.

La naturaleza de la carga influye mucho en el balanceo ya que las tareas pueden ser clasificadas en: tareas que requieren Entrada/Salida (E/S), tareas que requieren tiempo en el procesador y tareas mixtas, es decir, requieren E/S y tiempo de procesador. Las tareas que requieren E/S no deberían de migrarse ya que operaciones de E/S remotas haría poco eficiente la obtención de los resultados. Por el contrario, las tareas que simplemente necesitan tiempo de procesador podrían migrarse indistintamente.

Una vez planteado el balanceo de la carga, procederemos a dar una clasificacón de las técnicas existentes para balancear la carga en un sistema.

III.3 Clasificación de los Algoritmos de BCT

En la literatura hay taxonomías y estudios sobre los algoritmos de BCT en sistemas de cómputo que se comunican entre ellos por medio de redes de área local. Una de estas taxonomías (Xu y Lau, 1997) dice que los algoritmos de BCT se pueden dividir en:

- Iterativos.
- Directos.

Los algoritmos *iterativos* se basan en sucesivas aproximaciones a la distribución global óptima, así que en cada operación sólo necesitan preocuparse por la dirección de la migración de la carga de trabajo. Estos algoritmos, a su vez, se pueden subdividir en:

- Determinísticos.
- Estocásticos.

La subdivisión anterior se hace tomando en cuenta la forma de realizar las operaciones de BCT. Los métodos iterativos determinísticos se basan en ciertas reglas de distribución predefinidas. ¿A qué vecino se va a trasladar la carga extra y cuánta carga se va trasladar?, son preguntas cuyas respuestas dependen de ciertos parámetros, tales como el estado de los procesadores más cercanos o algunos niveles preestablecidos.

Por el contrario, con los métodos iterativos estocásticos, las cargas de trabajo son redistribuidas de forma aleatoria, conforme al objetivo del balanceo de la carga.

En los algoritmos *directos* simplemente se utiliza información global de todo el sistema para tomar decisiones de la distribución de la carga total. El balanceo y distribución de la carga se hacen en un solo paso.

III.4 Balanceo Estático vs. Balanceo Dinámico

El BCT se puede realizar de manera *dinámica* o de manera *estática*, aunque hay algunos algoritmos de balanceo que, por naturaleza, son dinámicos y no pueden realizarse de otra manera.

Los algoritmos de balanceo estático distribuyen los procesos en tiempo de compilación, en la mayoría de los casos se basan en información de los procesos y del sistema, normalmente esta información es obtenida "a priori".

Por otro lado los algoritmos dinámicos distribuyen procesos en los procesadores en tiempo de ejecución. La información en la que se basan también es colectada dinámicamente.

Una ventaja importante de los algoritmos de balanceo estáticos es que ellos no provocarán ningún incremento en el tiempo final de ejecución. Los algoritmos de balanceo estáticos son atractivos para programas paralelos donde los tiempos de ejecución de los procesos y sus requerimientos pueden ser predecidos. En algunas situaciones, el balanceo estático es la única opción porque el tamaño de los procesos imposibilita la migración de estos durante la ejecución. Esta opción también es buena cuando se está hablando de un sistema monousuario, en donde hay procesos de un solo usuario y él puede decidir sobre la distribución y asignación de los procesos de manera eficiente.

Los algoritmos de balanceo estáticos se basan en los tiempos estimados, tanto de ejecución como de comunicación entre procesos. Esto no es bueno para programas paralelos que normalmente son dinámicos o impredecibles. Por ejemplo, en un programa de búsqueda combinatoria paralela, los procesos evalúan soluciones candidatas de un conjunto de posibles soluciones para encontrar una que satisfaga un criterio específico. Cada proceso busca una solución dentro de una porción del espacio de soluciones. La forma y tamaño del espacio de soluciones usualmente cambia conforme la búsqueda avanza. Las porciones del espacio de soluciones que contienen soluciones con altas

probabilidades serán expandidas y exploradas exhaustivamente, mientras que las porciones del espacio que no tengan soluciones serán descartadas en tiempo de ejecución. Lo anterior implica que se tienen que crear y destruir procesos en tiempo de ejecución sin seguir un patrón. Para asegurar eficiencia en el paralelismo, los procesos tienen que ser distribuidos en tiempo de ejecución, y por lo tanto los patrones de cambio en la carga de trabajo de los procesadores son difíciles de predecir.

Otra situación similar ocurre cuando el sistema es multiusuarios y no hay forma de que se puedan distribuir estáticamente los procesos de todos los usuarios debido a que esta actividad es 100% no determinística y por tanto dinámica.

El éxito de los algoritmos de balanceo de la carga dinámicos gira sobre la probabilidad de que siempre habrá procesadores con poca carga o que están ociosos durante la ejecución de un programa.

A su vez los algoritmos dinámicos se pueden dividir en:

- Sin interrupción de procesos.
- Con interrupción de procesos.

El balanceo dinámico sin interrupción implica que los procesos tienen que terminar su ejecución en donde iniciaron y que no pueden ser migrados a otro procesador, es decir, sólo procesos nuevos se toman en cuenta para la distribución o balanceo de la carga.

Por el otro lado, el balanceo dinámico con interrupción permite que un proceso, el cual está ejecutándose en un procesador, puede ser suspendido, migrado y reanudado en otro procesador en el mismo punto donde se suspendió. Esta política de balanceo es más flexible y más atractiva ya que en cualquier momento es posible hacer un reacomodo o redistribución de la carga.

III.5 Pasos para el Balanceo Dinámico de la Carga

La ejecución de un algoritmo de balanceo de la carga dinámico requiere algunos medios para mantener una visión consistente del estado del sistema en tiempo de ejecución y algunas políticas de negociación para la migración de procesos entre procesadores. Generalmente, un algoritmo de balanceo de la carga dinámico consiste de cuatro componentes (Xu y Lau, 1997):

- Regla de medida de la carga.
- Regla de intercambio de información.
- Regla de iniciación.
- Operación de balanceo de la carga.

III.5.1 Medida de la Carga

Los algoritmos de balanceo se basan en la información de carga de trabajo de los procesadores. La información de carga de trabajo es normalmente calculada por un índice de carga, una variable no negativa, que toma un valor de cero si el procesador está ocioso y toma valores crecientes positivos conforme la carga aumenta. Un índice de carga debería ser una buen estimador del tiempo de respuesta de los procesos ejecutados en un procesador. Por lo general, es imposible estimar este tiempo porque el tiempo de respuesta de los procesos, durante su ejecución, depende no sólo de sus necesidades de procesador y memoria, sino que también depende de la comunicación entre los procesos. Comúnmente, este índice de carga es tomado como el número de procesos que se encuentran en la cola de "listos" del sistema.

III.5.2 Intercambio de Información

La regla de intercambio de información especifica cómo colectar y mantener la información de la carga de trabajo en los procesadores, la cual es necesaria para hacer decisiones de balanceo de la carga. Idealmente, un procesador debería guardar un registro de la información más reciente de la carga de trabajo de otros procesadores. Sin embargo y desde el punto de vista práctico, esto no es tan recomendable en máquinas de memoria distribuida con paso de mensajes porque la comunicación interprocesador necesaria para colectar la información de la carga introduce retardos muy significativos. Este tiempo extra en la comunicación no permite que los procesadores intercambien su información frecuentemente. Por lo tanto, una buena regla del intercambio de información debe lograr un equilibrio razonable entre un bajo costo para recolectar la información de la carga de todo el sistema y mantener una vista exacta del estado del sistema. Este equilibrio se puede lograr con las siguientes reglas de intercambio:

A petición: Los procesadores colectan la información de la carga de trabajo de otros cuando una operación de balanceo de carga está cerca de comenzar o iniciar. Esta regla de intercambio de información por petición minimiza el número de mensajes pero pospone la recolección de la información de la carga de todo el sistema hasta que una operación de balanceo de la carga sea iniciada.

Periódica: Los procesadores periódicamente reportan su información de carga de trabajo a otros sin importarles si esa información les es útil a otros procesadores o no. Esta regla permite a los procesadores que necesiten balancear la carga iniciar la operación basándose en la información existente y sin ningún retardo. El problema con esta regla es cómo establecer el intervalo para el intercambio de información. Un intervalo corto implicaría retrasos de tiempo grandes para la comunicación, mientras que un intervalo grande podría sacrificar precisión de la

información de la carga de trabajo usada en tomar una decisión de balanceo.

En estado de cambio: Los procesadores reparten la información de su carga de trabajo cuando su estado rebasa cierto límite. Esta regla simplemente es un término medio de las reglas a petición y periódica.

III.5.3 Regla de Iniciación

Una regla de iniciación indica cuándo se va a iniciar una operación de balanceo de la carga. La ejecución de una operación de balanceo implica que habrá un tiempo extra que no puede despreciarse, la decisión de iniciación debe comparar la cantidad de tiempo extra contra el incremento de rendimiento esperado. Así, es necesario una política de iniciación para determinar cuando una operación de balanceo de carga será provechosa o no. Una política de iniciación óptima es deseable pero impráctica ya que su derivación podría ser complicada. En efecto, los índices de carga son sólo una estimación de las actuales cargas de trabajo y la carga que ellos representan podría estar no actualizada debido a los retardos de comunicación y recolección de información. En lugar de esto, se usan políticas de iniciación heurísticas.

III.5.4 Operación de Balanceo de la Carga

Una operación de balanceo de la carga especifica los detalles de la distribución o migración. Se especifican los participantes en la operación, la cantidad de tareas migradas, las tareas que han sidos seleccionadas para ser migradas, etc. El conjunto de procesadores que participan en una operación de balanceo es conocido como "dominio de balanceo".

La selección de los procesos a ser migrados puede ser sin interrupción o con interrupción de procesos. Con una selección que no permita interrupción siempre se

selecciona a los procesos creados recientemente, mientras que con una selección que permita interrupción se podría seleccionar un proceso que se está ejecutando, si esto fuera necesario.

La migración de un proceso con interrupción implica, primero, suspender el proceso, enviarlo a otro procesador y después reanudarlo de manera remota. El contexto del proceso también necesita ser transferido lo cual es, generalmente, más costoso que en una transferencia sin interrupción. Está demostrado que una transferencia sin interrupción es preferida, pero la transferencia con interrupción puede desempeñarse significativamente mejor que la transferencia sin interrupción en ciertos casos.

Además del tiempo extra que implica la transferencia de los procesos, la selección necesita tomar en cuenta el tiempo que implica la comunicación entre procesos después de la migración. Por ejemplo, al distanciar dos procesos fuertemente acoplados o que necesitan intercambiar mucha información se generarán altos requerimientos de comunicación y esto podría eliminar la ganancia en tiempo que da el balanceo de la carga. En principio una selección debería separar procesos débilmente acoplados.

Los algoritmos y las reglas anteriores dictan la forma tradicional de balancear la carga en un sistema distribuido, pero recientemente ha surgido una nueva idea para balancear la carga, esta nueva idea se basa en la variación de la carga de manera local y rompe con los esquemas tradicionales. Los detalles de esta nueva aproximación se darán en el siguiente capítulo.

Capítulo IV

Algoritmo "Rate of Change"

IV.1 Introducción

El algoritmo de balanceo de la carga de trabajo *Rate of Change* (RoC-LB) fue propuesto en la Universidad de California - Irvine (USA), este algoritmo rompe con los métodos tradicionales y se constituye como una nueva alternativa de balanceo con el fin de obtener un incremento en el rendimiento de un sistema de cómputo. Aquí el balanceo de la carga de trabajo es visto como un problema local de variación de la carga y no como un problema global (Campos y Scherson, 1998).

Los algoritmos tradicionales intentan resolver el problema distribuyendo equitativamente la carga total del sistema entre todos los procesadores del sistema. Por el contrario, en el algoritmo "Rate of Change", el que un procesador tenga 3 tareas y otro tenga 10 tareas no es razón suficiente para iniciar una operación de balanceo de la carga y mientras la razón de carga no indique lo contrario se puede decir que el sistema está balanceado. Desde este punto de vista, se define el balanceo de la carga de trabajo como:

- La actividad de migrar unidades de carga de un elemento de procesamiento (EP) a otro, de tal forma que todos los EP's tengan al menos una unidad de carga en todo momento (Campos y Scherson, 1998).

Lo anterior suena lógico, ya que si un EP está ocupado ejecutando tareas, entonces su diferencia de carga con respecto a otros EP's no es tan importante porque no habrá ganancia en el rendimiento si se interrumpe la ejecución de los procesos y se transfieren a otros EP's para reanudar su ejecución. En otras palabras, la decisión de iniciar una transferencia de carga dependerá de cómo cambia la carga de un EP con el tiempo y no del número absoluto de unidades de carga que un EP tiene con respecto a otros EP's. Es así que este nuevo algoritmo utiliza la razón de cambio de la carga de trabajo en cada EP para iniciar cualquier actividad de balanceo.

IV.2 Descripción del algoritmo RoC-LB

Este algoritmo tiene las siguientes características:

Dinámico.- Porque no toma en cuenta información "a priori" sobre las tareas o el sistema. Todas las decisiones de iniciación y de balanceo son hechas en tiempo de ejecución.

Distribuido.- Esto es porque todas las decisiones de balanceo de la carga son hechas localmente y asíncronamente por cada procesador. No hay un control maestro o centralizado que se encargue de controlar la actividad de balanceo.

A petición.- Porque sólo los EP's que necesiten ajustar su carga pueden iniciar una actividad de migración de tareas. En este caso lo harán los procesadores que están a punto de quedarse sin carga.

Con interrupción de procesos.- Este algoritmo soporta que procesos que están siendo ejecutados puedan ser suspendidos, migrados y reanudados en otro procesador.

Implícito.- Porque toda la actividad de balanceo es hecha por el sistema y no por el usuario.

La meta principal de este algoritmo es minimizar el tiempo de ocio de un procesador sin agregar un tiempo extra considerable al tiempo total de ejecución de los procesos.

A continuación describiremos varios conceptos utilizados en el algoritmo. Para ser consistentes con el algoritmo original utilizaremos los identificadores cortos que se utilizaron en inglés.

Intervalo de Muestra (SI).- Es un intervalo de tiempo en el cual se calcula la carga de trabajo de un EP. La longitud de este intervalo no es constante, cambia por el número de solicitudes de carga recibidas o por el tráfico de la red. Este intervalo es medido como un múltiplo de la duración de una rebanada de tiempo, es un parámetro adaptable y es calculado independientemente por cada EP ya que no hay un concepto de "Reloj Global".

Diferencia de la Carga (DF).- Es el cambio que ha sufrido la carga de trabajo con respecto al intervalo anterior. Esta información es usada para predecir el número de tareas que terminarán en intervalos siguientes. Al comienzo de cada intervalo de muestra cada EP calcula su diferencia en la carga.

Retardo de Red (ND).- Es un parámetro adaptable y está definido como el tiempo que pasa entre la iniciación de una solicitud de carga y la recepción de la respuesta, ya sea positiva o negativa.

Límite Superior (HT).- Es un límite que se establece para determinar el estado de un procesador, cuando la carga de un procesador rebasa este límite se dice que está sobrecargado y es considerado un procesador que puede compartir carga (*source*).

Límite Inferior (LT).- Este límite sirve para determinar el estado de un procesador, si el nivel de la carga se encuentra por debajo de este límite entonces es considerado como un procesador que va a solicitar carga (*sink*). Así, si el nivel de carga se encuentra entre HT y LT se dice que el procesador se encuentra en estado neutral.

Límite Crítico (CT).- Este límite se establece para saber cuando un procesador necesita hacer una petición de carga a otro procesador. Si la carga de trabajo de un procesador es menor que CT, inmediatamente se inicia una petición de carga sin importar cual es la diferencia de carga con respecto al intervalo anterior.

Carga Predicha (PL).- Es el número de tareas que debería tener un procesador después de un tiempo igual a ND. Si PL es mayor o igual que cero, entonces se "predice" que el procesador no se quedará ocioso o sin carga durante un periodo igual a ND y por lo tanto no se iniciará una solicitud de carga. Pero si PL es menor que cero, inmediatamente se solicitará carga a otro procesador.

En la figura 2 se muestra un posible escenario de cómo cambia la carga de trabajo de un solo procesador con el tiempo y se presentan los conceptos descritos anteriormente.

Con todo lo anterior, el algoritmo RoC-LB puede ser visto como un problema de decisión de 4 etapas que son las siguientes:

- ¿Cuándo iniciar la migración de tareas? (Cuándo).
- ¿A qué procesador enviar la solicitud de migración? (Dónde).
- ¿Cuántas tareas se van a migrar? (Cuántas).
- ¿Qué tareas se van a migrar? (Cuáles).

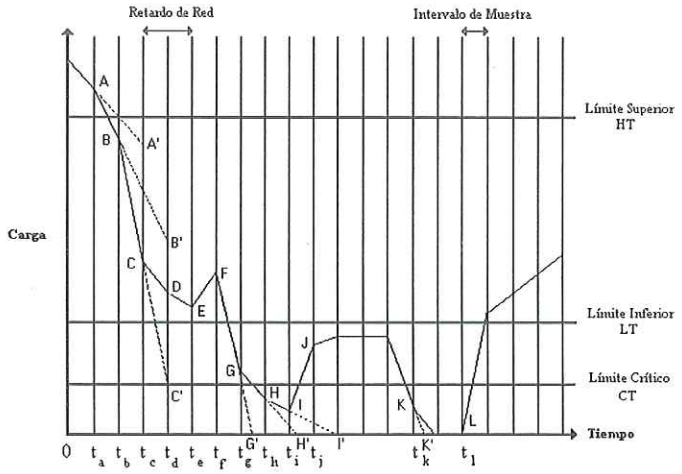

Figura 2: Variación de la carga de un elemento de procesamiento y posibles casos de inicio de una solicitud de carga

Algo que es muy importante es que estas 4 etapas ocurren asíncrona e independientemente en cada procesador. A continuación describiremos cómo se llevan a cabo estas etapas.

IV.2.1 ¿Cuándo?

Existen dos razones por las cuales un EP iniciará una petición de carga y sólo los procesadores que necesiten carga podrán iniciar una operación de balanceo.

La primera se da en la siguiente situación. Al principio de un SI, cada EP tiene que calcular el cambio de su carga con respecto al intervalo anterior. Este cambio en la carga es usado para predecir cuántas tareas finalizarán en los siguientes SI's. De esta manera si se calcula que el EP se quedará ocioso o sin carga en un tiempo menor que ND, entonces el EP iniciará una solicitud de carga.

La otra situación que implica una operación de balanceo es la siguiente. En cualquier momento, si la carga del EP está por debajo de CT, entonces inmediatamente se iniciará una solicitud de carga olvidando la predicción y los cálculos que se

han hecho basados en la carga actual.

Existen 2 excepciones en esta etapa, es decir, aunque se presente alguna de las situaciones anteriores no se hará una petición de carga. Estas excepciones son:

1. Cuando un EP ya tiene una solicitud de carga previa, entonces no se mandará ninguna otra solicitud hasta que la carga previamente solicitada llegue completamente.

2. Cuando el nivel de carga de un EP esté arriba de su HT y se predice que el procesador se quedará ocioso en el siguiente ND.

IV.2.2 ¿Dónde?

Antes de describir a dónde son enviadas las peticiones de carga incluiremos dos tablas. Cada EP tiene 2 tablas locales que contienen información del sistema. La primera es la tabla *sink* en la cual están listados todos los EP's que han solicitado carga. La segunda es la tabla *source* que contiene a todos los EPs que pueden compartir carga, es decir, que están sobrecargados.

La selección del EP al que se enviará la solicitud de carga es simple. El EP que hizo la solicitud selecciona un EP de su tabla *source*. El elemento seleccionado será el primero de la tabla. Una vez que ha sido seleccionado se le envía una petición de carga.

Al iniciar el sistema, la tabla *source* estará vacía ya que aún no hay información del sistema, por lo tanto la selección del EP al que se hará la petición de carga se hace de manera aleatoria. Esto significa que cuando todavía no hay información del sistema, cualquier EP puede ser considerado un EP *source*. Pero ésta es una situación muy especial, usualmente se da al iniciar el sistema.

Otra cosa que hay que resaltar es que el identificador del EP seleccionado es comparado con las entradas o elementos de la tabla "sink" para prevenir que se envíe una solicitud de carga a un EP "sink".

IV.2.3 ¿Cuántas?

Al comienzo de cada SI los EPs calculan su DL. Usando su DL se puede calcular cuántas tareas se debería tener después de un tiempo igual a ND y suponiendo que DL permanece constante durante todo el ND. Esta cantidad es la carga predicha (PL) como se dijo antes. Si PL es mayor o igual a cero, entonces el EP sabe que no se quedará ocioso durante el actual ND y por lo tanto no inicia una solicitud de carga. Por el contrario, si PL es menor que cero, el EP solicita inmediatamente carga al EP "source" previamente seleccionado. El número de tareas solicitadas está dada por **abs(PL)**.

Mientras tanto, el EP *source* sólo transferirá las tareas solicitadas si su carga es mayor que HT y en este caso sólo transferirá el número de tareas que le sobren o que lo mantengan en un estado normal, es decir, si *current_load* es el número de tareas que tiene un EP con carga, entonces se mandan [*current_load* − HT] tareas.

IV.2.4 ¿Cuáles?

Finalmente, para responder a esta pregunta diremos que se migran las tareas más viejas, es decir, las tareas que tienen más tiempo ejecutándose. Esto debido a que estas tareas tienen una probabilidad más alta de vivir tiempo suficiente y compensar su costo de migración. La "edad" en este caso se refiere al tiempo de CPU que han tenido las tareas y no el tiempo que tienen de haber sido creadas.

IV.3 Actualización de las Tablas

Como ya dijimos cada EP tiene 2 tablas que contienen información del sistema, el

EP tiene en estas tablas su "punto de vista" de la distribución de la carga, esto es, la carga o el estado en que se encuentran los otros EP's del sistema. También ya se mencionó que se tiene que mediar entre precisión de la información contenida en las tablas y el tiempo extra que se requiere para difundir tal información.

Este algoritmo usa una regla distribuida para difundir el estado del sistema, es decir, la carga de cada uno de los EP's. Lo anterior se logra usando un mecanismo de actualización de bajo costo con respecto al tiempo y es llevado a cabo durante el envío de un mensaje de solicitud de carga hecho por un procesador sin carga o por un procesador que reenvía dicha solicitud. A continuación describiremos la estructura de dicho mensaje.

IV.3.1 Estructura del Mensaje

Cuando un procesador solicita carga a otro procesador que tiene carga, se hace por medio de un mensaje que es creado cuando se va a realizar la petición. En este caso se sigue el mismo criterio de utilizar los identificadores cortos para tener consistencia con el algoritmo original. El mensaje está compuesto por los siguientes campos:

SPI.- (Sink Processor Identifier) Cada EP en el sistema debe tener un único identificador, dicho identificador es almacenado en este campo una vez que se va a enviar la petición. Esto es necesario por que podría darse el caso de que el mensaje sea reenviado a otro EP y éste último debe ser capaz de identificar quien fue el EP que inició la petición de carga y así poder enviarle las tareas solicitadas.

NLUR.- (Number of Load Units Requested) En este campo se especifica el número de tareas que se han solicitado y se actualiza cada vez que la solicitud va siendo satisfecha por los procesadores que la reciben.

LSP_PE.- (Load State of Preceding Processing Element) Este campo contiene el estado del EP que envía o reenvía la solicitud. Una vez recibido el mensaje, el EP

puede conocer el estado del EP que envió el mensaje. Este campo es usado para actualizar las tablas locales de los EPs cada vez que un mensaje es reenviado.

NF.- (Number of Forwards) Aquí se va contando el número de veces que el mensaje ha sido reenviado. Es usado para evitar un ciclo o reenvío infinito. El mensaje y la solicitud de carga son eleminados o desechados cuando se ha completado el número de tareas solicitadas o cuando NF alcanza un tope máximo predefinido.

IV.3.2 Protocolo de Actualización

El EP que realiza una solicitud de carga tiene que hacer lo siguiente:

- Cuando inicia la solicitud.

  1. Crea un mensaje con SPI igual a su identificador. Inicializa NF con cero y NLUR con el número de tareas solicitadas.
  2. Selecciona el EP *source* al que se enviará la solicitud. Remueve la entrada asociada con el EP seleccionado de la tabla *source*..
  3. Cambia el estado del EP a esperando (*WAITING*) y pon esta información en LSP_PE en el mensaje para prevenir que se haga otra solicitud de carga antes de recibir una respuesta.

  La razón para remover la entrada asociada del EP con carga seleccionado es porque si este EP es capaz de satisfacer la solicitud (parcial o completamente) entonces será agregado otra vez a la tabla de procesadores con carga (*source*) cuando la respuesta llegue. Si por el contrario el EP seleccionado no es capaz de satisfacer la solicitud, entonces ya no es un EP con carga o *source* y no tiene porque estar en la tabla de procesadores con carga.

- Cuando recibe la solicitud.

  1. Agrega a la tabla *source* el identificador del EP que envió la carga.
  2. Si la respuesta satisfizo completamente la solicitud original, entonces cambia el estado del mensaje a listo (*READY*).

Por otro lado, cuando un EP recibe un mensaje con una solicitud de carga, éste tiene que realizar lo siguiente:

1. Actualiza sus tablas agregando SPI a su tabla *sink* y remueve este identificador de la tabla *source* si éste se encuentra ahí. Si el mensaje tiene que ser reenviado, actualiza las tablas con la información que contienen LSP_PE.
2. Si es un EP *source*, entonces verifica la carga exacta.
   - Si se puede satisfacer completamente la solicitud de carga, entonces envía la carga a SPI y elimina el mensaje del sistema.
   - Si se puede satisfacer parcialmente, entonces resta el número de tareas enviadas a NLUR.
3. Si no es un EP *source* o no se puede satisfacer la carga completamente, entonces incrementa NF. Si NF es igual a un máximo predefinido, entonces elimina el mensaje y responde a SPI diciendo que su solicitud de carga ha sido eliminada. De otra forma, actualiza LSP_PE con el estado actual del EP y envía el mensaje a otro EP con carga, previamente seleccionado de la tabla correspondiente.

IV.4 Un posible escenario de balanceo de la carga.

La figura 3 corresponde a la situación donde un EP (EP1 en este caso) inicia una solicitud de carga. En el paso 1, el EP1 crea un mensaje de solicitud de carga y lo

envía a EP3, ya que el se encuentra en la primera localidad de su tabla *source*, después EP1 remueve a EP3 de la tabla *source*.

Cuando EP3 recibe el mensaje, verifica su carga y concluye que solamente puede satisfacer la solicitud parcialmente. Agrega a EP1 a su tabla *sink*, decrementa el campo NLUR en la cantidad de tareas que va a transferir a EP1, incrementa el contador NF y reenvía el mensaje a EP15, que es el EP que se encuentra en la primera localidad de la tabla de nodos con carga de EP3.

Después de que EP3 respondió a EP1, es asignada una nueva entrada para EP3 en la tabla de nodos con carga (*source*) de EP1 (paso 2).

Más adelante EP15 recibe el mensaje que fue reenviado (paso 3), agrega a EP3 a su tabla de nodos con carga (*source*), remueve a EP1 de la misma tabla y lo agrega en la tabla de nodos sin carga (*sink*). Una vez que EP15 pudo satisfacer la carga que faltaba en la solicitud inicial, elimina el mensaje del sistema (paso 4).

Cuando EP1 recibe la respuesta de EP15, agrega a éste último en su tabla de nodos con carga (*source*) y el mecanismo de actualización está completo y el sistema se encuentra balanceado.

En el capítulo siguiente se describe a MOSIX y se presentan los detalles del algoritmo de balanceo de la carga que utiliza.

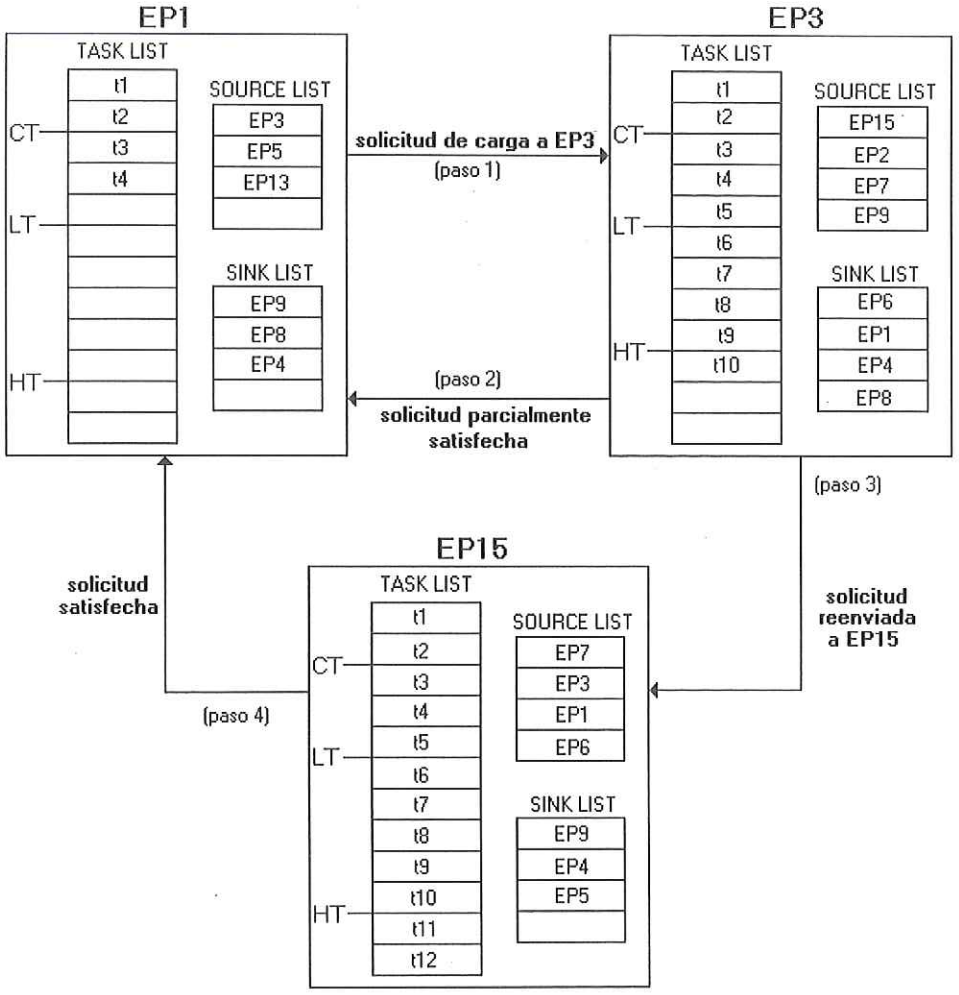

Figura 3: Tres elementos de procesamiento, EP1 inicia una solicitud de carga, EP3 y EP15 satisfacen dicha solicitud

Capítulo V

MOSIX

V.1 Introducción

Para entender lo que es y lo que se puede hacer en MOSIX, primero compararemos un sistema multiprocesador de memoria compartida (SMP) y un cluster de computadoras.

En un SMP varios procesadores comparten la memoria del sistema, lo que significa que todos tienen el mismo espacio de direcciones disponible. Lo anterior permite una comunicación muy rápida entre procesos por medio de la memoria compartida. Los SMPs pueden manejar o administrar un número muy grande de procesos de manera muy eficiente, además de asignar recursos a estos procesos eficientemente. En cualquier momento un proceso puede ser creado, eliminado o cambiar su estado y el sistema se adaptará instantáneamente a estos cambios imprevistos de los procesos. El usuario no está involucrado en la asignación de recursos, en el balanceo de la carga o en la administración de los recursos y mucho menos tiene conocimiento de cómo se realizan dichas actividades.

A diferencia de los SMPs, los clusters están formados por una colección de nodos que pueden ser computadoras personales, servidores, estaciones de trabajo e incluso SMPs. Normalmente estos nodos tienen velocidades de procesamiento distintas, tamaños dis-

tintos de memoria y diferentes tipos de procesadores. La mayoría de los clusters son utilizados por varios usuarios lo que implica que son sistemas multiusuario de tiempo compartido. En estos sistemas el usuario es responsable de asignar sus procesos en los nodos del sistema y de administrar los recursos del mismo.

Actualmente existe software como PVM y MPI que facilitan el trabajo al usuario, es decir, ayudan a la asignación de procesos en los procesadores y en la administración de los recursos del sistema. Estos sistemas proporcionan un ambiente de programación que requiere que el usuario tenga conocimiento del hardware, o por lo menos de la topología, y que las aplicaciones sean hechas tomando en cuenta dicho hardware y topología. Además, requieren una sintáxis específica y particular que limita al desarrollo de aplicaciones que puedan ser fácilmente portables. En estos ambientes se tienen herramientas de asignación de procesos fijas o estáticas, estas herramientas sólo usan consideraciones de carga y no toman en cuenta otros detalles como lo son la memoria disponible, el tiempo extra en la entrada y salida de datos, la unión de nuevos nodos al sistema, etc. Otro detalle importante es que estos sistemas se ejecutan a nivel de usuario, como cualquier aplicación ordinaria. Debido a esto, son incapaces de responder a cambios imprevistos de la carga o en la disponibilidad de otros recursos y no pueden redistribuir la carga de trabajo dinámicamente.

Por el contrario, desde el punto de vista de un usuario los SMPs garantizan un uso eficiente y balanceado de los recursos entre los procesos que se están ejecutando en cada momento, además son fáciles de usar porque el sistema operativo se encarga de administrar los recursos de forma adaptable y totalmente transparente.

Los clusters actuales carecen de estas últimas capacidades, además dependen de la asignación y administración de recursos que el usuario pueda hacer. Esto es un inconveniente y puede guiar a un pérdida significativa de rendimiento debido a una mala asignación de recursos y a una mala distribución de la carga de trabajo.

MOSIX surge como un Sistema Operativo Distribuido con un conjunto de algoritmos que permiten compartir y asignar los recursos en un cluster utilizando migración dinámica de procesos, que es totalmente transparente para el usuario.

V.2 Descripción de MOSIX

MOSIX es un acrónimo de "**M**ulticomputer **O**perating **S**ystem for UN**IX**, que en otras palabras significa que es un sistema operativo distribuido del tipo UNIX. MOSIX está formado por un grupo de algoritmos adaptables y dinámicos que se encargan de la administración de los recursos de un sistema distribuido. MOSIX, también permite que varias computadoras, con uno o varios procesadores, puedan trabajar juntas dando la imagen de un solo sistema, es decir, permite que un cluster de computadoras sea visto por un usuario como si estuviera trabajando frente a un solo sistema.

El desarrollo de MOSIX comenzó a finales de los 70's y principios de los 80's por un grupo de investigadores de la Universidad Hebrea de Jerusalem que tomaron la versión 7 de UNIX y lo modificaron para crear un sistema operativo distribuido llamado MOS, el cual corría sobre un cluster de computadoras PDP-11. A través de los años, el mismo grupo de investigadores ha trabajado sobre diversos sistemas operativos, sobre distintos tipos de computadoras y actualmente han desarrollado lo que ahora se conoce como MOSIX, el cual está basado en LINUX y que corre sobre un cluster de PC's.

Los algoritmos de MOSIX están diseñados para responder dinámicamente ante cambios o variaciones imprevistas de los recursos con los que cuenta cada uno de los nodos en el sistema. Esto último es logrado por medio de la migración de procesos de un nodo a otro, interrumpiendo de manera transparente la ejecución del proceso, con el fin de balancear la carga y evitar pérdida en el rendimiento debido a un excesivo intercambio de páginas de memoria[1] porque el proceso se está ejecutando en un nodo

[1] También conocido como "swapping"

que no tiene suficiente memoria física.

La meta principal de MOSIX es aumentar el rendimiento total de un cluster de computadoras y crear un ambiente multiusuario de tiempo compartido, en donde se ejecuten eficientemente aplicaciones secuenciales y paralelas.

El ambiente de ejecución estándar para MOSIX es un cluster de computadoras en donde todos los recursos del sistema están disponibles para cada uno de los nodos.

Para poder realizar las actividades anteriores, MOSIX está compuesto de dos partes:

- Un mecanismo de interrupción y migración de procesos.
- Un conjunto de algoritmos adaptables para compartir recursos.

Ambas partes son implementadas a nivel del núcleo, usando un módulo cargable, de tal forma que la interfaz del núcleo no se modifica. De esta manera estas actividades son totalmente transparentes al nivel de aplicación.

V.2.1 Mecanismo de Interrupción y Migración de Procesos

Con el Mecanismo de Interrupción y Migración de Procesos de MOSIX (PPM - *Preemptive Process Migration*) se puede migrar cualquier proceso, en cualquier momento y a cualquier nodo que se encuentre disponible.

Usualmente, la migración está basada en información proporcionada por uno de los algoritmos para compartir recursos, pero los usuarios en determinado momento podrían hacer caso omiso de estas decisiones automáticas del sistema y migrar sus procesos manualmente. La migración manual puede ser iniciada síncronamente por el proceso o por una solicitud explícita del usuario o de un proceso de otro usuario. La migración manual puede ser útil para implementar una política particular o para probar diferentes algoritmos de calendarización.

Cada proceso tiene un "Único Nodo Origen" (UHN - *Unique Home-Node*) en donde fue creado. Normalmente este nodo es donde el usuario está conectado.

El modelo de imagen de sistema en MOSIX es un cluster, en el cual cada proceso se ejecuta en su propio UHN y todos los procesos de un usuario comparten el mismo ambiente de ejecución en su UHN.

Los procesos que migran a otros nodos usan los recursos locales del nodo remoto cuando es posible, pero interactúan con el ambiente del usuario a través de su UHN. Por ejemplo, supongamos que un usuario crea varios procesos, algunos de los cuales son migrados del UHN a otros nodos del sistema. Si el usuario ejecuta el comando "*ps*", como salida obtendrá un reporte del estado de todos los procesos aunque estos se estén ejecutando en nodos remotos. Si uno de los procesos lee la hora actual, esto es, invoca a "*gettimeofday( )*", éste obtendrá la hora de su UHN.

Cuando los recursos de los nodos remotos tengan que ser utilizados por procesos locales y la cantidad de estos recursos esté por debajo de cierto límite, los procesos migrados tendrán que ser regresados a su UHN.

La granularidad de la distribución de trabajo en MOSIX es el proceso. Los usuarios pueden ejecutar aplicaciones paralelas simplemente creando sus procesos en un solo nodo, después el sistema asignará estos procesos a los nodos que estén disponibles. Si durante la ejecución se liberan recursos por otros procesos, entonces el sistema se encarga de reasignar estos nuevos recursos a los procesos restantes.

MOSIX no tiene un sistema de control central o maestro-esclavo entre los nodos, sino que cada nodo puede trabajar como un sistema autónomo y tomar sus propias decisiones de control independientemente de otros nodos.

Este diseño permite una configuración dinámica, donde se pueden agregar o quitar nodos en el cluster y los trastornos causados serán mínimos.

V.2.2 Algoritmos para Compartir Recursos

Como se dijo antes, MOSIX tiene un conjunto de algoritmos que permiten a un cluster ser visto como un solo sistema al compartir los recursos de cada nodo. Según (Barak et al, 1999) los principales algoritmos que utiliza MOSIX para compartir recursos son:

- Algoritmo de balanceo de la carga de trabajo.
- Algoritmo de acomodo de memoria.

El algoritmo de balanceo de la carga es dinámico e intenta constantemente reducir la diferencia de la carga entre pares de nodos, migrando procesos del nodo que tiene más carga al que tiene menos carga. Este trabajo es descentralizado, es decir, todos los nodos ejecutan los mismos algoritmos y la reducción de las diferencias de carga es hecha independientemente por pares de nodos. Para el algoritmo de balanceo de la carga es importante el número de procesadores que hay en cada nodo y la velocidad de cada uno de ellos. Este algoritmo responde a cambios en las cargas de los nodos o a las características de ejecución de los procesos.

El algoritmo de acomodo de memoria está diseñado para colocar el máximo número de procesos en la "Memoria del Cluster" y evitar lo más posible el intercambio de páginas entre el disco y la memoria real. Este algoritmo es iniciado cada vez que un nodo empieza a realizar un número muy grande de intercambios de página. En este caso el algoritmo hace caso omiso del algoritmo de balanceo e intenta migrar el proceso que está realizando el intercambio a otro nodo que tenga la memoria suficiente para evitar los intercambios de página, aunque esta acción pueda guiar a un estado no balanceado del sistema.

V.3 Migración de Procesos

Como ya se dijo, MOSIX soporta migración de procesos, la cual se realiza transparentemente para el usuario. Después de la migración, un proceso continúa interactuando con su ambiente sin importar donde se esté ejecutando. Para la implantación de la migración de procesos, se dividió al proceso en 2 partes o contextos (Barak et al, 1999):

Contexto de Usuario.- También llamado *remote*, es la parte que puede ser migrada. Contiene el código del programa, pila, datos, mapas de memoria y registros de los procesos. Esta es la parte del proceso que se ejecuta en modo usuario.

Contexto de Sistema.- También llamado *deputy*, es la parte dependiente del UHN. Contiene una descripción de los recursos que están siendo usados por el proceso. Esta es la parte del proceso que se ejecuta en modo núcleo.

El contexto de usuario puede ser migrado un número indefinido de veces, mientras que el contexto de sistema nunca se migra.

La interfaz entre el contexto de usuario y el contexto de sistema está bien definida. Por tanto, es posible interceptar cada interacción entre estos 2 contextos y enviar esta interacción a través de la red. Esto está implementado en la capa de unión que tiene un canal de comunicación especial para la interacción de los dos contextos. En la Figura 4 se muestran dos procesos que tienen un mismo UHN. En la misma figura podemos ver que el proceso de la izquierda es un proceso normal que se está ejecutando en su UHN, mientras que el proceso de la derecha ha sido migrado y su parte *remote* está ejecutándose en otro nodo del cluster.

El tiempo que un proceso tarda en migrar tiene una componente fija y una componente lineal. La componente fija corresponde al tiempo necesario para establecer o crear una nueva estructura de proceso en el nuevo sitio de ejecución. La componente

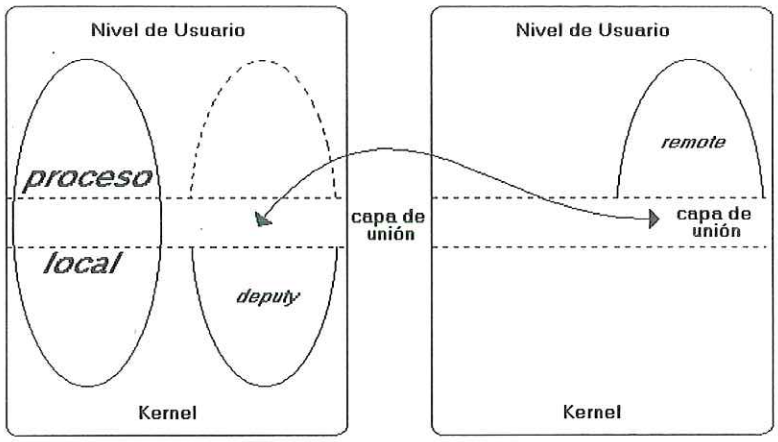

Figura 4: División del proceso en MOSIX, la parte remote es la que puede ser migrada y la parte deputy siempre permanece donde fue creado

lineal es proporcional al número de páginas de memoria que serán transferidas. Para minimizar el tiempo de migración no se transfieren todas las páginas, sino que sólo se transfieren las tablas de páginas y las páginas que están siendo utilizadas.

Cuando se ejecuta un proceso remotamente, la transparencia en la ubicación es lograda enviando sólo las llamadas al sistema, que son dependientes del UHN, hacia la parte que no migra (*deputy*) que se encuentra en el UHN del proceso. Las llamadas al sistema son una forma síncrona de comunicación entre los dos contextos. Todas las llamadas al sistema hechas por el proceso son interceptadas por la capa de unión en el nodo remoto. Si la llamada al sistema es independiente de su UHN, entonces es ejecutada en el nodo remoto. Por el contrario, si la llamada al sistema es dependiente de su UHN, entonces es enviada a la parte que no migra, el cual a su vez, ejecuta la llamada al sistema en el UHN. Finalmente la parte que no migra regresa los resultados a la parte migrable para que continúe con la ejecución del código de usuario en el nodo remoto.

Una desventaja de la parte que no migra es que hay un tiempo extra al ejecutar

llamadas al sistema que son dependientes del UHN, así como al realizar operaciones de acceso a archivos y realizar operaciones de red. Este tiempo extra es inevitable por que hay recursos como son archivos o conexiones virutales entre procesos (*sockets*), los cuales no pueden ser migrados junto con la parte migrable de un proceso y tener acceso a ellos desde otro nodo implica que la comunicación será a través de la red.

V.3.1 Limitantes de la Migración

Ciertas funciones del núcleo de linux no son compatibles con la división de contextos. Algunos ejemplos obvios son las manipulaciones directas de dispositivos de entrada y salida, acceso directo a instrucciones privilegiadas de entrada y salida, acceso directo a la memoria de algunos dispositivos, etc. Algunos otros ejemplos también incluyen planificación de procesos de tiempo real. Esto último no es posible porque no tendría sentido migrar un proceso de este tipo, debido al tiempo extra que implicará la migración y la comunicación con su parte no migrable lo cual entra en contradicción con su objetivo principal que es ofrecer resultados en "tiempo real". Otra razón es que no es posible garantizar una planificación justa y rápida para estos procesos en otros nodos.

De esta forma, un proceso que haga uso de los recursos anteriores será confinado automáticamente a su UHN. Si el proceso ya ha sido migrado a un nodo remoto y tiene que hacer uso de los recursos anteriores, lo primero que se hace es regresarlo a su UHN y después continuar con su ejecución.

V.4 Balanceo de la Carga de Trabajo en MOSIX

Esta sección describe las metas, los algoritmos y procedimientos de la política de BCT que sigue MOSIX. La meta principal de esta política es balancear la carga de todos los procesadores que están activos en el cluster. Esto es logrado mediante con-

tinuos intentos de reducir las diferencias de carga entre pares de procesadores con migraciones dinámicas de procesos. La migración se hace de procesadores que tienen más carga hacia procesadores que tienen menos carga. Es así que los procesadores sobrecargados son los encargados de iniciar la operación de balanceo. Además esta política es simétrica y distribuida en el sentido de que todos los procesadores ejecutan los mismos algoritmos y que las reducciones de las diferencias de carga son hechas independientemente por cualquier par de procesadores. Otra meta de esta política es responder ante los cambios en las cargas de los procesadores, ante el cambio en el número de procesadores y tomar en cuenta las características de los procesos para tomar decisiones de balanceo.

Esta política usa varios parámetros que evitan migración excesiva y un ir y venir de procesos, es decir, que un proceso migre entre dos procesadores varias veces consecutivas. Uno de estos parámetros esta dado por la *Regla de Residencia* para cada proceso. De acuerdo a esta regla, cada proceso debe establecer un tiempo mínimo de ejecución en el procesador donde se esté ejecutando actualmente. Esta regla se aplica a procesos de reciente creación y a procesos recientemente inmigrados de otro procesador. Una desventaja de esta regla es que los procesos de corta duración no serán migrados. Esta regla tiene dos excepciones donde el tiempo de residencia no es tomado en cuenta. La primera es cuando un proceso crea muchos procesos nuevos en un periodo de tiempo relativamente pequeño. En este caso, hay que tratar de migrar este proceso entre los procesadores para que todos los nuevos procesos no sean creados en uno solo y se haga un uso excesivo de algún recurso de dicho procesador. La segunda excepción es cuando un proceso es creado y el procesador actual no tiene suficiente memoria libre disponible para el proceso nuevo, entonces el sistema intentará crear el proceso en otro procesador que tenga suficiente memoria.

La regla de la residencia toma la duración de dicho periodo como un parámetro

importante para el balanceo de la carga. Con este periodo se trata de evitar migraciones inútiles, es decir, sólo se migrará un proceso si al hacer esto se obtendrá una ganancia en el rendimiento o dicho de otra forma, tendremos un decremento en el tiempo de finalización del proceso. Con esta regla sólo se permite la migración de un proceso después de que se ejecute por un periodo de tiempo igual al tiempo que tardaría en migrar a otro procesador. Hay que notar que esta duración depende de la velocidad de los dos procesadores involucrados, la saturación de la red, la carga de los procesadores y otros parámetros locales. Con lo anterior se garantiza que el tiempo de finalización de un proceso sea a lo más dos veces el tiempo de ejecución y así, el peor caso es cuando un proceso termina su ejecución inmediatamente después de haber sido migrado.

Otro parámetro que también es importante para el balanceo de la carga en MOSIX es la *viscosidad*, que impone un tiempo de retardo entre migraciones sucesivas hacia o desde un procesador. La duración de este retardo es llamada *Grado de Viscosidad* y es una función de la carga del procesador actual, la cantidad de memoria disponible y la disponibilidad de otros recursos. Cuando la carga se incrementa mucho, el grado de viscosidad es pequeño, es decir, el retardo para migrar procesos es pequeño permitiendo la emigración de procesos y al mismo tiempo es prohibida la inmigración de otros procesos. Este parámetro también se decrementa si la cantidad de memoria disponible decrementa, o cuando uno de los recursos del sistema no está disponible.

Además de los factores anteriores, la decisión de cuándo migrar un proceso depende del historial de ejecución del proceso y de la carga total. El historial de un proceso nos dice el uso de procesador, la cantidad de E/S requerida o la comunicación con otros procesos.

En MOSIX, las decisiones tomadas en la política de balanceo se llevan a cabo en tres algoritmos principales que son:

- Cálculo de la Carga.

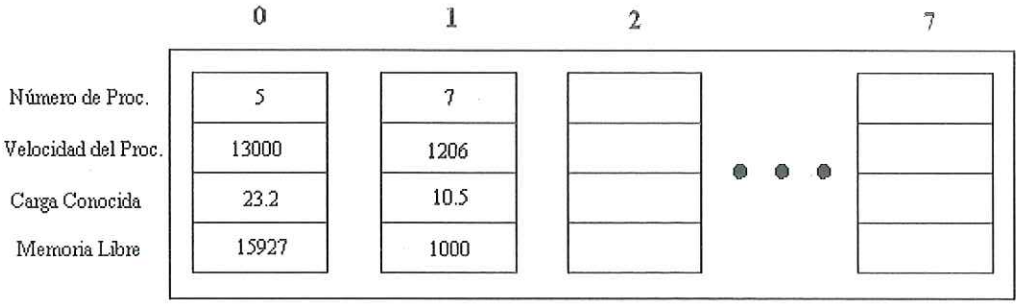

Figura 5: *Loadinfo* es la estructura básica en el balanceo de la carga en MOSIX, contiene información del nodo local y de otros nodos del cluster

- Difusión de la Información.
- Consideraciones de Migración.

Estos tres algoritmos se unen jerárquicamente ya que un algoritmo sirve como base para un siguiente algoritmo en la política de BCT. El algoritmo de Cálculo de la Carga es la base de la política ya que los valores de carga son medidos en este algoritmo y son difundidos entre los procesadores por medio del siguiente algoritmo que es el de Difusión de la Información. Finalmente esta información es usada y tomada en cuenta por el algoritmo que se encarga de la migración, en el punto más alto de la política, que también se encarga de seleccionar los procesos que van a ser migrados y de seleccionar el procesador destino tomando en cuenta también algunas consideraciones y características de los procesos.

En MOSIX existe una estructura la cual es manipulada por las rutinas de balanceo, dicha estructura llamada *loadinfo*, se muestra en la Figura 5. Esta estructura es un arreglo de registros con 4 campos para almacenar la siguiente información:

1. El número de procesador.
2. La velocidad del procesador.

3. La carga del procesador.

4. La cantidad de memoria libre del procesador.

Cada entrada del arreglo anterior representa un procesador en el sistema. El tamaño del arreglo es determinado mediante consideraciones probabilísticas. En la práctica se utiliza un arreglo de 8 entradas para clusters de hasta 512 procesadores (Barak et al, 1993). La primera entrada es utilizada para la información del procesador local. El resto de las entradas representan un subconjunto del resto de los procesadores. Este subconjunto está formado con los procesadores que han tenido comunicación con el procesador local recientemente.

En las siguientes secciones se describirá con más detalles los 3 principales algoritmos usados para balancear la carga en MOSIX.

V.5 Algoritmo para el Cálculo de la Carga

El cálculo de las cargas en cada procesador es un requisito para un buen balanceo de la carga de trabajo en un cluster de computadoras. Los algoritmos descritos en esta sección son ejecutados por cada procesador de manera independiente. Este es el primer paso de la política de balanceo de MOSIX y la información obtenida sirve para los otros algoritmos de dicha política.

Una simple medida de la carga de un procesador en un instante dado podría ser el número de procesos que están listos para ejecución y que están esperando a que se desocupe el procesador, es decir, la carga del procesador puede ser tomada como la longitud de la cola donde están los procesos listos para ejecución. Además este valor debe estar correlacionado con el uso del procesador en el preciso instante de la medición. Este tipo de medición mostrará rápidas variaciones en la carga cuando se hacen varias

mediciones en un intervalo relativamente corto. Estas variaciones podrían proporcionar falsos índices de carga, trayendo como consecuencia migraciones de procesos que pueden ser improductivas.

Por ejemplo, el procesador **A** podría registrar una baja en la carga debido a que en el momento que se hizo la medición, todos los procesos están esperando por alguna operación de E/S. Poco tiempo después, uno o más procesos que se encuentran en el procesador **B**, son migrados al procesador **A**. Al llegar al procesador **A**, los procesos migrados se dan cuenta de que mientras se realizaba la migración, las operaciones de E/S terminaron y los procesos que estaban esperando pasan a la cola de procesos listos en espera de ejecución. Lo anterior nos lleva a una sobrecarga y a un desbalanceo pues ahora, el procesador **A** tiene más carga que el procesador **B**. El ejemplo anterior implica que los valores de carga no deben estar basados en una simple medida. En MOSIX se utilizan dos medidas de la carga las cuales tratan de solucionar situaciones como la anterior, estas dos medidas se describen a continuación.

V.5.1 Algoritmo de la Carga Local

En MOSIX la carga es medida por cada procesador cada unidad atómica de tiempo t, esta unidad de tiempo es llamada *quantum*. En la práctica, un *quantum* depende del ciclo de reloj de la computadora ya que la medición es hecha cada interrupción de reloj. Así, los valores de carga medidos son acumulados y promediados cada periodo de tiempo T, el cual es un múltiplo entero de t, es decir, $T = \mu \times t$, donde μ es una constante. En la práctica se toman los siguientes valores T=*1* segundo y $\mu = 30$ (Barak et al, 1993). Sea W_i el número de procesos que están listos para ejecución en algún procesador durante el intervalo $(t_{i-1}, t_i), i = 1, 2, \ldots, \mu$. Entonces, una aproximación inicial de la carga local, L_t, en un procesador cada unidad de tiempo T está dada por la siguiente formula (Barak et al, 1993):

$$L_t = \sum_{i=1}^{\mu} W_i \tag{1}$$

En un cluster donde hay procesadores de diferentes velocidades, L_T debe ser dividida por la velocidad relativa del procesador. Esta velocidad relativa está definida como el cociente entre la velocidad del procesador local (CPU_{sp}) y la velocidad del procesador más rápido (CPU_{mx}). Esta normalización nos permite comparar cargas de procesadores con distintas velocidades. De lo anterior tenemos que la carga local de un procesador cada periodo T es:

$$LLoad_T = \frac{L_T \times CPU_{mx}}{CPU_{sp}} \tag{2}$$

Por ejemplo, si el sistema tiene 2 procesadores, uno con la máxima velocidad y el otro con la mitad de la velocidad máxima, entonces la carga del más lento será el doble de la carga del primero cuando los dos tengan el mismo número de procesos.

Para calcular la velocidad relativa de los procesadores se utiliza un benchmark, el cual se ejecuta entre dos procesadores en modo de "un solo usuario"[2]. El resultado será un número n el cual indica que la velocidad del procesador medido es $n/10000$ veces la velocidad de un procesador Pentium II a 400 MHz. Esta medida se tiene que tomar para cada uno de los nodos en el cluster.

V.5.2 Algoritmo de la Carga Exportada

La carga local es una muy buena aproximación a la carga real del procesador, pero no es suficiente para el balanceo de la carga del cluster. Supongamos que el procesador **A** tiene menos carga que el procesador **B**. El procesador **B** podría decidir migrar uno o más procesos hacia el procesador **A** de tal forma que se incrementa la carga del procesador **A**. Si la carga del procesador **A** es incrementada de tal forma que es mayor

[2]La información completa se encuentra en el archivo README que viene con MOSIX

que la carga del procesador **B**, entonces el procesador **A** podría decidir migrar algunos procesos al procesador **B**. Esta situación podría iniciar una oscilación de procesos, la cual no contribuye en un aumento de rendimiento sino que, por el contrario, decrementa el rendimiento.

El ejemplo anterior muestra que la carga local de un procesador no puede ser comparada con la carga local de otro procesador. La solución a este problema es usar dos medidas de carga para cada procesador como se muestra en (Barak et al, 1993). La primera es la carga local que se describió en la sección anterior y la segunda medida es la carga exportada. Esta última medida es la información de carga que un procesador da a conocer a los demás procesadores.

La expresión para calcular la carga exportada, cada periodo T, está dada por (Barak et al, 1993):

$$ELoad_T = LLoad_T + C_{Export} \tag{3}$$

donde C_{Export} es una constante dada por:

$$C_{Export} = \frac{CPU_{mx} \times (\mu + \epsilon)}{CPU_{sp}} \tag{4}$$

y ϵ es una constante pequeña. Hay que notar que el valor de C_{Export} es ligeramente más grande que la carga que representa un solo proceso.

También, cada procesador monitorea su utilización, CPU_{ut}, la cual está definida como el porcentaje de tiempo que el procesador está disponible para ejecutar tareas (Barak et al, 1993). Bajo condiciones normales, es decir, el procesador no está haciendo intercambio de páginas entre la memoria y el disco, $CPU_{ut} = \mu$, lo cual representa el 100% de utilización. Cuando el procesador está haciendo intercambio de páginas, el porcentaje de utilización es decrementado por la cantidad de tiempo que el procesador

no puede ejecutar procesos listos. Digamos que ω de μ posibles *quantums* que dura T, el procesador no puede ejecutar procesos listos porque todos estos procesos, están esperando que se les asigne memoria. Entonces en (Barak et al, 1993) la utilización del procesador está definida como:

$$CPU_{ut} = \mu - \omega \tag{5}$$

En este caso, la carga exportada es multiplicada por un factor de μ/CPU_{ut} mientras la carga local permanece sin cambios. También hay que notar que al aumentar el valor de la carga exportada, se evita la inmigración de procesos al procesador actual. Esto da la oportunidad de que el procesador pueda hacer intercambio de páginas sin ser interrumpido por nuevos procesos que lleguen de otros procesadores.

El cálculo de la carga local y el cálculo de la carga exportada se agregaron en una rutina del núcleo de Linux. La rutina es invocada después de una interrupción de reloj, en dicha rutina se van acumulando las 2 cargas tomando en cuenta la utilización del procesador como se mencionó anteriormente.

V.6 Algoritmo de Distribución de la Información

El algoritmo de distribución de la información es el responsable de difundir continuamente la información de la carga entre los procesadores, de tal forma que cada procesador tenga suficiente información del resto de los procesadores. Este algoritmo es descentralizado, ya que no hay un procesador que tenga conocimiento global de las cargas de todos los procesadores en cualquier momento. En lugar de esto, cada procesador mantiene información reciente de un subconjunto de procesadores. El algoritmo es probabilístico ya que la selección de un procesador para notificarle la carga del procesador actual, se hace de manera aleatoria y dicha selección se hace entre todos los procesadores participantes.

Otra propiedad del algoritmo es que soporta configuraciones dinámicas, es decir, se pueden agregar o remover procesadores sin afectar el funcionamiento del algoritmo.

Para la difusión de la información es necesario el vector de carga que se mostró en la Figura 5, en este vector se va almacenando la información de carga recibida de los demás nodos. Ahora, sea T la unidad de tiempo que se definió en la sección V.5.1, asumimos que los procesadores están numerados $1, 2, 3, \ldots, n$, donde n denota el número de procesadores, de los cuales no necesariamente todos están activos. Sea $Load[j]$ la $j - esima$ entrada en el vector de carga, con $0 \leq j \leq (l-1)$, donde l es el tamaño del vector de carga en cada procesador.

Cada tiempo T, después de haber calculado la carga, cada nodo selecciona 2 o 3 nodos de entre todos los participantes en el cluster a los cuales les envíe mim mensaje. Para asegurar que la información es distribuida entre todos los procesadores, se hace distinción entre tres tipos de procesadores:

1. Un procesador i seleccionado aleatoriamente.

2. Un procesador j seleccionado aleatoriamente, pero que se sabe de antemano que está activo.

3. Un procesador k seleccionado aleatoriamente en el mismo nodo. Esto en el caso de nodos SMP.

La selección del primer tipo de procesadores tiene como fin incorporar a procesadores que recientemente se han activado. Los nuevos procesadores envían sus propios mensajes de carga para avisar que se han unido al sistema. Con la elección de procesadores del segundo tipo se tiene como fin dar prioridad a los procesadores que ya se sabía que estaban activos. Esto es muy útil e importante en sistemas donde los procesadores activos son pocos. La elección del tercer tipo tiene como fin dar prioridad a las migraciones dentro de un sistema multiprocesador, de esta forma se aprovecha la

alta velocidad de las conexiones que existen entre procesadores en el mismo nodo.

V.7 Consideraciones para la Migración

En esta sección describiremos los algoritmos relacionados con la migración de procesos, los cuales incluyen un algoritmo para seleccionar un proceso adecuado para migrar y un procesador hacia donde migrar el proceso previamente seleccionado. La meta de estos algoritmos es determinar si con la migración del proceso seleccionado hacia el procesador destino se contribuirá para balancear la carga. Otra meta de estos algoritmos es disminuir el tiempo de ejecución, si esto es posible, al migrar el proceso seleccionado a un mejor ambiente de ejecución, es decir, a otro procesador con más recursos que el actual. Estos algoritmos son totalmente distribuidos ya que son ejecutados por cada procesador independientemente unos de otros y sin un control central. Pero aparte de que no hay un control central, cada proceso tiene sus propias consideraciones para migrar.

Para tomar la decisión de migrar un proceso, MOSIX hace un historial de la ejecución de cada proceso. Estos historiales son hechos durante todo el tiempo de vida de cada proceso o desde su última migración y son actualizados constantemente. El historial muestra los recursos solicitados por el proceso en varios procesadores y su consumo de CPU. Cuando sea seleccionado para migrar, se asumirá que el proceso mantendrá el mismo comportamiento que ha tenido en el procesador actual. Idealmente, el procesador destino fue seleccionado por ser el procesador con el costo mínimo de ejecución para dicho proceso.

Varios factores influyen sobre la migración exitosa de un proceso candidato hacia un procesador destino. Dichos factores son los siguientes:

Tiempo de Respuesta.- La motivación principal para migración es un mejor tiempo de respuesta que el proceso migrante pueda recibir en el procesador destino. El tiempo de respuesta es inversamente proporcional a la carga de dicho procesador. Así, es más deseable migrar un proceso a un procesador con poca carga que a un procesador con mucha carga.

Tiempo extra en la Comunicación.- MOSIX no distribuye archivos entre procesadores. Si un proceso tiene que realizar operaciones remotas de E/S sobre archivos o sobre algún dispositivo que reside en un procesador remoto, estas operaciones tienen que viajar a través de la red de interconexión. Estas operaciones incrementan los costos de comunicación ya que se requiere que los procesadores involucrados ejecuten protocolos de comunicación. Este tiempo extra puede ser eliminado migrando el proceso hacia el procesador donde se encuentran los dispositivos o archivos. Es por eso que la selección de un procesador destino está influenciado por la disminución de este tiempo extra.

Tiempo de Migración.- Este factor, por sí mismo, es muy importante en la decisión de migración, ya que el tiempo de migración es una función lineal del tamaño del proceso.

Número de Procesos Hijos.- Si un proceso creó varios procesos hijos y cada hijo consume pequeñas cantidades de recursos del sistema, entonces no habrá justificación para su migración. Pero la suma total del consumo de recursos de todos los hijos podría ser razón suficiente para migrar al proceso padre.

Capacidad en la Tabla de Procesos.- Si la tabla de procesos está a punto de llenarse y un proceso desea crear un proceso hijo, la migración será casi obligatoria. En tal caso, casi todos los procesadores podrían ser seleccionados como destino.

Cantidad de Memoria Libre.- Si no hay suficiente memoria para permitir la creación de un nuevo proceso, entonces se hará el intento en otro procesador que tenga suficiente memoria.

Así que cuando un proceso detecta una bandera de migración, la cual le indica la necesidad de migración, se tienen que integrar todas las consideraciones de migración y decidir cuándo hay que migrar hacia cualquier procesador y además verificar que su carga permita la migración.

V.7.1 Selección de un Proceso Candidato a Migrar

Para la selección del proceso candidato es necesario revisar y evaluar a todos los procesos que se encuentran en la tabla de procesos del procesador actual. Para cada proceso de usuario, el algoritmo compara los parámetros relacionados con la migración (tiempo de ejecución o periodo de residencia) para encontrar el mejor proceso candidato a migrar. El algoritmo también utiliza varios límites predeterminados para optimizar la selección. Este algoritmo es responsable de seleccionar y marcar a un proceso como candidato a migrar, pero no es responsable de iniciar la migración.

V.7.2 Selección de un Procesador Destino

Una vez que se tiene un proceso candidato, es necesario encontrar el mejor lugar para su ejecución. En este algoritmo se calculan los costos $C[j]$ y después se selecciona el $min(C[j])$ que corresponde al mejor procesador. Los costos se calculan de la siguiente manera. Sea i el número de identificación del proceso actual, $1 \leq i \leq n$. Sea $Load[j]$ la $j - esima$ entrada del vector de carga del procesador i, donde $0 \leq j \leq (l-1)$. $C[j]$ denota el costo de ejecutar un proceso candidato en el procesador cuya información se encuentra almacenada en $Load[j]$.

En resumen, el BCT sigue los siguientes pasos: cada vez que llega información de otro procesador se verifica si hay un procesador, en el vector de carga, que tenga menos carga que el procesador actual; si existe, entonces se selecciona un proceso, que a su vez es seleccionado tomando en cuenta varios factores de migración que puedan influir en el rendimiento. Finalmente se selecciona un procesador destino para la ejecución del proceso candidato.

Capítulo VI

Descripción de la Simulación

VI.1 Introducción

En este capítulo describiremos el programa que se hizo para simular la ejecución de una carga de trabajo en un cluster de computadoras. Durante la ejecución se distribuyen los procesos entre los nodos del cluster utilizando los algoritmos de balanceo de la carga *Rate of Change* y el que utiliza MOSIX. Estos dos algoritmos fueron descritos en los capítulos anteriores.

VI.2 Simulación de los Algoritmos de Balanceo

Para mostrar como está construida la simulación de los algoritmos de balanceo iremos de lo general a lo particular. Primero veremos a la simulación como una "caja negra" que tendrá algunas entradas y algunas salidas, las cuales serán descritas en las siguientes secciones. Lo anterior se muestra en la Figura 6. Conforme avance la descripción iremos abriendo la "caja negra" y poco a poco profundizaremos en los detalles de la simulación.

VI.2.1 Entradas para la Simulación

Como lo que queremos simular es la ejecución de un conjunto de procesos en un cluster de computadoras, entonces necesitamos conocer las características de los pro-

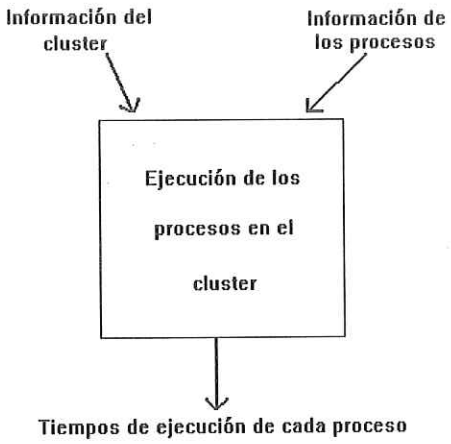

Figura 6: Descripción de la simulación vista como una caja negra, con dos tipos de entradas y una salida

cesos y del cluster. Por lo tanto podemos decir que en la simulación se necesitan dos tipos de entradas.

El primer tipo de entrada está relacionado con la configuración del cluster, específicamente con:

- El número de nodos en el cluster.
- Las características de cada nodo, nosotros sólo utilizamos:
  - Número o identificador de cada nodo para poder hacer referencia a un nodo en particular.
  - Número de procesadores en el nodo, esto debido a que es posible tener sistemas multiprocesadores.
  - El tipo de procesador que tiene el nodo para poder tener una referencia entre procesadores de distinta velocidad.
  - Velocidad relativa del procesador. Esta velocidad se toma en base a un procesador Pentium II a 400 MHz cuya velocidad es de 10000 unidades.
  - Cantidad de memoria en el nodo, ya que tenemos una memoria finita y sólo

podemos cargar un número finito de procesos.

La información de entrada se puso en archivos de tal forma que el programa pueda tomar la información correspondiente y continuar con la simulación. Con esto es posible modificar fácilmente el escenario de balanceo, es decir, la carga de trabajo o el cluster, sin necesidad de recompilar el programa en cada cambio.

Para la información del cluster utilizamos dos archivos de texto. El primer archivo, llamado NODOS, contiene un número que corresponde precisamente al número de nodos en el cluster, el cual es un número entero positivo que va desde 1 hasta un número razonable de nodos. El segundo archivo, llamado COPS, contiene la descripción de cada uno de los nodos del cluster. El archivo está formado con n líneas, una por cada nodo. A continuación mostramos un ejemplo de cómo está almacenada la información en el archivo COPS suponiendo que el cluster tiene 3 nodos con las siguientes características:

```
#    NUMEP  TIPO          SPEED    RAM
1    1      PentiumII     10000    48
2    1      PentiumIII    13000    64
3    1      PTM-MMX/166   2103     32
```

El segundo tipo de entrada contiene información sobre el conjunto de procesos que vamos ejecutar en el cluster. Normalmente a este conjunto de procesos se le conoce como "Carga de Trabajo". La información de cualquier carga de trabajo, necesaria para nuestra simulación, debe ser la siguiente:

- El número de procesos que componen a la carga de trabajo.
- Características de cada proceso. Para nosotros es suficiente:
  - Un identificador de proceso o PID.
  - El tamaño del proceso en Kbytes.

- La prioridad, que puede ser útil para alguna política de planificación de procesos particular.
- Tiempo de ejecución individual. Esta información es útil solamente para fines de la simulación y no para la operación de balanceo, pues en el balanceo de la carga se desconocen los tiempos de ejecución de los procesos que se estén ejecutando en un momento determinado.
- Un número de procesos hijos que pudiera crear el proceso actual.

Para la información de la carga de trabajo también se utilizaron dos archivos de texto. En el primer archivo, llamado PROCESSES, se tiene un número entero positivo que indica el tamaño de la carga, es decir, el número de procesos a ejecutar. En el segundo archivo, llamado INFOPROCESSES se tienen m líneas, una para cada proceso, que muestran las características de la carga de trabajo. A continuación se presenta un ejemplo de cómo debe estar ordenada la información de la carga de trabajo:

```
PID    MEM(k)    PRIORI    Ti      SON
1      1         20        1       0
2      1         20        1       0
3      1         20        1       0
```

Las líneas anteriores representan una carga de trabajo compuesta por tres procesos idénticos, es decir, tienen el mismo tamaño (MEM), prioridad (PRIORI), tiempo de ejecución (T_i) y número de procesos hijos (SON), lo único diferente entre ellos es el identificador de procesos (PID).

VI.2.2 Salidas de la Simulación

Como lo que estamos simulando es la ejecución de procesos y no nos interesa el resultado que pudieran arrojar dichas ejecuciones, sino que lo único que nos interesa es

el tiempo que tardan ejecutándose, entonces la única salida que tendrá la simulación será precisamente el tiempo de ejecución de cada proceso. Esto también se puede ver en la Figura 6.

La simulación terminará cuando el último de los procesos también termine su ejecución y ponga sus resultados en la salida. Los resultados de la simulación son desplegados en pantalla y también se almacenan en un archivo de texto, lo cual permitirá analizar los datos finales después de las pruebas. El nombre de este archivo de salida es: RESULTADOS y contiene m líneas, una para cada proceso, con la siguiente información:

- El identificador del proceso al cual corresponde la información (pid).
- El tiempo de finalización del proceso (t_{if}).
- El tiempo de ejecución del proceso. Este dato es el mismo que se leyó como entrada (T_i).
- El tiempo extra (t_{ext}) debido a:
  - Las migraciones que hizo durante la ejecución (t_{mig}).
  - El tiempo que pasó listo esperando que se le asignara el procesador (t_{list}).
  - Tiempo extra debido a la operación de balanceo (t_{bal}).
  - Un tiempo extra dependiendo de la velocidad del procesador (t_{spdNS}).

Al final de estas m líneas se muestra la métrica de rendimiento que se usó para comparar los algoritmos de balanceo, dicha métrica es un rendimiento normalizado, NP[1], que también fue usada en (Scherson y Campos, 1998). Este rendimiento normalizado determina la efectividad del algoritmo de balanceo, es decir, si $NP \rightarrow 0$, entonces el algoritmo no es efectivo, pero si $NP \rightarrow 1$, significa que el algoritmo si es efectivo. La forma de calcular NP está dada por la siguiente ecuación:

[1]De el inglés "Normalized Performance"

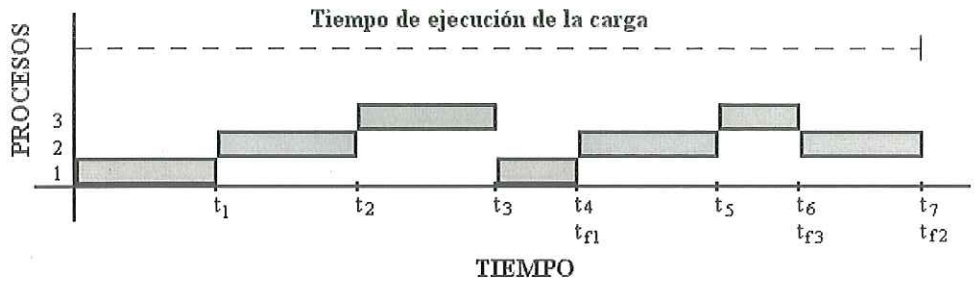

Figura 7: El tiempo de ejecución de la carga de trabajo está formado por el tiempo de finalización de la última tarea

$$NP = \frac{(T_{nobal} - T_{bal})}{(T_{nobal} - T_{opt})} \tag{6}$$

donde, T_{nobal} es el tiempo necesario para ejecutar la carga de trabajo en un cluster sin balanceo de la carga. Para nuestro caso es el tiempo necesario para ejecutar la carga de trabajo en un solo nodo. T_{bal} es el tiempo necesario para ejecutar la carga de trabajo en un cluster con balanceo, este tiempo es el resultado de nuestra simulación. T_{opt} es el tiempo necesario para ejecutar la carga en un solo nodo dividido entre el número de nodos en el cluster.

El tiempo de ejecución de la carga en cualquiera de los casos es tomado como el tiempo que transcurre desde que inició la ejecución del primer proceso hasta que termina de ejecutarse el último de los procesos que forman la carga de trabajo. Si tomamos en cuenta que estamos hablando de sistemas de tiempo compartido en donde a los procesos se les va asignando rebanadas de tiempo en el procesador para su ejecución, entonces la ejecución de tres procesos podría seguir el comportamiento que se muestra en la Figura 7.

En las líneas que se presentan a continuación mostramos un ejemplo de cómo se vería el archivo de salida, RESULTADOS, si la carga de trabajo que queremos ejecutar estuviera formada por tres procesos:

pid	tif	=	ti	+	text	con:	tmig	tlist	tbal	tspdNs
1	1		1		0		0	0	0	0
2	2		1		1		0	1	0	0
3	3		1		2		0	2	0	0

NP = 0.590894

Lo anterior muestra que el proceso, cuyo "*pid*" es 3, finalizó también en 3 unidades de tiempo (t_{if}). En este caso el proceso 3 fue el último en terminar su ejecución por lo que representa el tiempo de ejecución de la carga de trabajo.

En la simulación, el tiempo de finalización de cada proceso está dado por la siguiente ecuación:

$$t_{if} = t_i + t_{ext} \tag{7}$$

donde t_i es el tiempo de ejecución que fue proporcionado como entrada, un poco más adelante se explicará su utilidad en la simulación, y t_{ext} es el tiempo extra generado por la suma de los factores antes mencionados, así:

$$t_{ext} = t_{mig} + t_{list} + t_{bal} + t_{spdNS} \tag{8}$$

Siguiendo con el ejemplo del proceso número 3 del archivo RESULTADOS, tenemos que el tiempo de finalización de este proceso es sólamente la suma de su tiempo de ejecución ($t_i = 1$) más el tiempo que pasó en estado de listo ($t_{list} = 2$), ya que los otros tiempos son 0.

VI.2.3 Proceso Interno de la Simulación

Una vez que se ha leído la información del cluster y de la carga de trabajo, de alguna forma tenemos que simular que los procesos se ejecutan en los nodos del cluster. Por lo

pronto asumiremos que los nodos ya están listos para ejecutar procesos y lo siguiente por hacer es crear o cargar los procesos en el cluster.

En las simulaciones hechas por (Campos y Scherson, 1998) se contemplaron dos posibles escenarios con 16 nodos en el cluster. En el primer escenario, crean la carga de trabajo en todos los nodos del cluster distribuyendo los procesos uniformemente. Podemos decir que éste es el mejor de los casos para iniciar la simulación pues la carga ya está distribuida y el trabajo del algoritmo va a depender de nuevos procesos que sean creados o de procesos que concluyan su ejecución llevando, posiblemente, al sistema a un estado desbalanceado. El segundo caso es donde la carga de trabajo es creada en la mitad de los nodos. Este escenario puede verse como un caso intermedio en donde el sistema está "medio balanceado".

En este trabajo quisimos realizar la simulación en un escenario donde toda la carga de trabajo se crea en un solo nodo. Este escenario puede verse como el peor de los casos ya que el sistema está completamente desbalanceado.

Así, primero creamos los procesos en un nodo. Después el SOD "virtual" de cada nodo se encargará de administrar y despachar los procesos que se encuentren en el nodo. Mediante algún algoritmo de balanceo se migrarán procesos hacia los demás nodos tratando de nivelar la carga de trabajo y ejecutarla en el menor tiempo o en el mayor número de nodos posibles.

La situación anterior se muestra en la Figura 8 donde cada rectángulo representa un nodo y cada círculo representa un proceso en ejecución.

También hay que notar que nosotros no tomamos en cuenta distintas topologías para la conexión de los nodos del cluster como lo hicieron en (Campos y Scherson, 1998), sino que sólo simulamos que la topología de conexión entre los nodos era lineal.

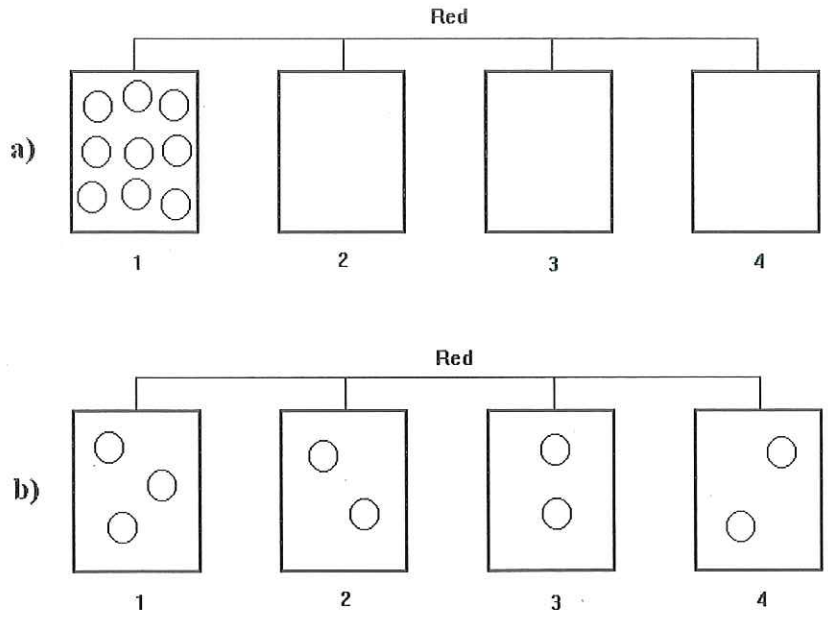

Figura 8: a) Al inicio de la ejecución todos los procesos están en un solo nodo, b) Se distribuyen los procesos para balancear la carga de trabajo

Ejecución de un Proceso en un Nodo

A continuación explicaremos como se simula la ejecución de los procesos en los nodos del cluster.

En realidad los nodos son simples estructuras de datos que contienen información de las características de los nodos, esta información fue proporcionada en la entrada por el archivo COPS. Dependiendo de las características de los nodos, los procesos son ejecutados más rápido, en el caso de un nodo veloz, o más lento, en el caso de un nodo lento.

Cada procesador tiene asociado un arreglo que corresponde a su "Tabla de procesos". Dicho arreglo contiene los identificadores (*pid*) de los procesos que se estén ejecutando en ese momento. Por lo tanto la creación de procesos es equivalente a colocar los *pids* de dichos procesos en el arreglo que simula la tabla de procesos del nodo como se muestra en la Figura 9.

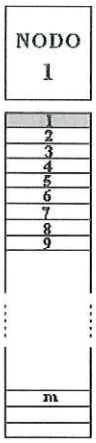

Figura 9: Cada nodo en la simualción tiene una tabla de procesos

Una vez cargados los procesos, la ejecución consiste en leer el tiempo de ejecución (t_i) del proceso que se encuentra en la localidad 0 de la tabla de procesos y decrementarlo *dec* unidades de tiempo durante una "rebanada de tiempo", es decir, SLIDE veces. Se van a asignar rebanadas de tiempo a un proceso mientras su $t_i \geq 1$, de lo contrario significa que el proceso ha terminado su ejecución.

Con lo anterior simulamos el subsistema de planificación de procesos de un sistema operativo, en donde un proceso es ejecutado en una rebanada de tiempo. Si termina su ejecución antes de que concluya su rebanada de tiempo, entonces el proceso es removido de la tabla y se asigna el procesador al proceso que se encuentra en la localidad 1 de la tabla de procesos, es decir, se hace un corrimiento de tal forma que ahora el proceso que estaba en la localidad 1 pasa a la localidad 0. Pero si el proceso estuvo en ejecución durante una rebanada de tiempo y no terminó su ejecución, entonces es movido al final de la tabla de procesos para permitir que el siguiente proceso sea ejecutado. De lo anterior podemos ver que la política de planificación de procesos que se está simulando es "Round Robin".

Una rebanada (SLIDE) es un número que permanece constante durante la simulación. El decremento (*dec*) que se le hace a t_i depende de la velocidad del procesador

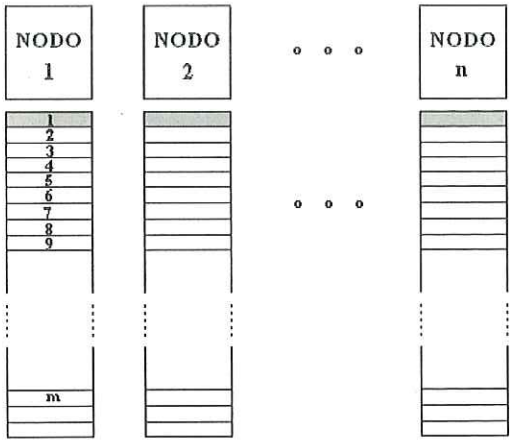

Figura 10: Los procesos pueden ser creados en un nodo o en cualquier número de nodos del cluster

actual, de tal forma que si la ejecución se realiza en un procesador lento entonces $dec \approx 1$, pero si la ejecución es en un procesador rápido, entonces $dec \approx$ SLIDE.

Por ejemplo, si la rebanada de tiempo dura 13 unidades (SLIDE=13), la velocidad del procesador actual es de 1000 unidades y al proceso que está en ejecución aún le faltan 13 unidades de tiempo para terminar su ejecución ($t_i = 13$), entonces t_i se va a decrementar una unidad de tiempo durante SLIDE veces (13, 12, ..., 1, 0). De tal forma que el proceso terminará su ejecución en 13 unidades de tiempo. Ahora, si tenemos las condiciones anteriores, excepto por la velocidad del procesador que es 13000 unidades, entonces t_i se va a decrementar 13 unidades de tiempo una sóla vez (13, 0).

En resumen, la carga puede ser iniciada en un solo nodo (Figura 10) y mediante un mecanismo de balanceo los procesos son repartidos a los demás nodos para aprovechar el número de procesadores como se muestra en la Figura 11. Cada uno de los nodos del cluster ejecuta los procesos que se encuentren en su tabla de procesos de manera totalmente independiente, siguiendo una política de *Round Robin* y de acuerdo a las características de dicho nodo.

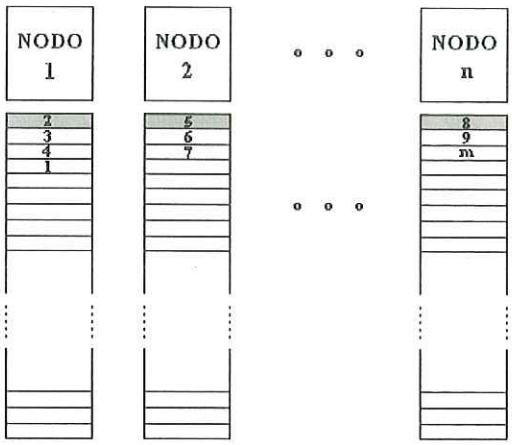

Figura 11: Durante la simulación los procesos son migrados con el objetivo de balancear la carga de trabajo

VI.3 Funciones Importantes de la Simulación

En esta sección describiremos las funciones más importantes de la simulación y de esta forma terminaremos la descripción de la misma. Las funciones que se muestran, no son exactamente las originales pues hemos incluido sólo las instrucciones que reflejan el trabajo medular de la función y en algunos casos hemos incluido pseudocódigo.

VI.3.1 Variables y Estructuras Utilizadas

Primero vamos a comentar las variables y estructuras más importantes que se utilizan en la simulación. Estas variables son:

- *numprocess*,
- *numnodos*,
- *cluster*,
- *workload*,
- *tabla*

Las dos variables que nos dicen el tamaño de la carga y el número de nodos en el cluster son *numprocess* y *numnodos*, respectivamente. El valor que toman estas variables es leído, al inicio de la simulación, de los archivos NODOS y PROCESSES.

La estructura principal, en donde se almacena la información de los nodos, se llama *cluster* y es un arreglo de tamaño de a lo más NUMMAXNODOS. Este tamaño es constante y nosotros lo fijamos indicando el número máximo de nodos que queremos simular. Por su parte, la estructura *workload* es un arreglo que contiene la información de todos los procesos que se ejecutarán en el cluster. Finalmente, *tabla* es un arreglo que contiene las tablas de procesos de cada uno de los nodos del cluster, es decir, $tabla[i]$ es la i-ésima tabla de procesos, la cual corresponde al i-ésimo nodo del cluster que estamos simulando.

La estructura de un proceso en nuestra simulación tiene los siguientes campos:

```
struct infoprocess
{
    int pid;
    int mem;
    int priori;
    int ti;
    int son;

    int tif;
    int text;
    int tremain;
    int tcpu;
    int nmig;
    int tmig;
    double costomig;
    int onNet;
    int numswitch;
    int tlist;
    int bal;
    int SpdNs;
}
```

Los primeros 5 campos contienen la información del proceso que fue leída al inicio de la simulación. Los siguientes campos son usados principalmente para calcular el tiempo de ejecución de cada proceso independientemente de los demás. El campo *tif* contiene el tiempo de finalización del proceso, *text* es el campo en donde se van acumulando los demás tiempos; *tremain* se va decrementando durante la ejecución del proceso e indica el tiempo que le falta al proceso para terminar; *tcpu* es el tiempo que el proceso ha pasado en el CPU, *tnmig* es el número de veces que un proceso ha migrado; *costomig* es el costo para migrar un proceso y este costo se calcula multiplicando el tamaño del proceso por la latencia; *tmig* es el tiempo que pasó migrando y está dado multiplicando el número de veces que migró por el costo de migrar ese proceso; *onNet* el valor de este campo se va decrementando en una unidad desde que el proceso ha sido enviado a otro nodo, al llegar a 0 significa que el proceso ha pasado un tiempo sobre la red; *numswitch* contiene el número de veces que un proceso fue suspendido o quitado del procesador para colocar otro proceso; *tlist* es el tiempo que el proceso pasó en la cola de procesos listos; el campo *bal* contiene el tiempo que se agregó debido a las operaciones de balanceo y finalmente *SpdNs* contiene el tiempo debido a que la ejecución se realizó en un procesador lento.

Por otro lado la estructura de un nodo es la siguiente:

```
struct infonodo
{
    int nodo;
    int numep;
    char tipo[CAD15];
    int speed;
    int mem;

    int carga;
    int estado;
}
```

Como se puede ver, los primeros 5 campos de la estructura de un nodo son para almacenar la información del nodo que fue leída al inicio de la simulación. Los siguientes 2 campos se utilizan durante la simulación, *carga* es el campo en el cual se va guardando la carga que va teniendo el nodo en cada momento de la simulación, en realidad este campo se va actualizando con el número de procesos que están en la tabla de procesos listos; *estado* es un campo que se utiliza en el algoritmo de balanceo de MOSIX, ya que es posible que nuevos nodos se unan o dejen el cluster, pero en realidad no se utilizó pues ese escenario no se probó.

VI.3.2 main( )

Como se dijo en la sección VI.1 y se muestra en la Figura 6, en nuestro programa tenemos dos entradas para la simulación y la salida, es decir, el tiempo de ejecución de la carga de trabajo es desplegada en pantalla y guardada en un archivo. Los pasos anteriores corresponden precisamente a la función principal de nuestro programa. Así que la función principal es la siguiente:

```
main( )
{
    LeeInfoCluster( );
    LeeInfoProgramas( );
    Simula( );
    DespliegaResultados( );
}
```

Las funciones *LeeInfoCluster*() y *LeeInfoProgramas*() son las encargadas de leer los archivos que contienen la información necesaria para la simulación. En la función *DespliegaResultados*() se despliegan los resultados y se guardan en el archivo RESULTADOS. La función *Simula*() será descrita con más detalle en la siguiente sección.

VI.3.3 Simula( )

A continuación se muestra la fución *Simula*():

```
Simula( )
{

    IniVar( );
    CargarProgramas_en(numnodosini);
    repite
    {
       para cada nodo del cluster
       {
         Scheduling(nodo);
       }
    } hay carga en el cluster;
    CalculaTiemposFinales( );

}
```

En esta función primero se inician todas las variables que se van a usar en la simulación, después se cargan todos los procesos en el nodo inicial. Aquí hay que mencionar que la función *CargarProgramas_en(numnodosini)* recibe como argumento un número, el cual corresponde a la cantidad de nodos en los cuales será iniciada la carga para ejecución. Esto último significa que es posible distribuir la carga de trabajo al inicio de la ejecución en 8 o 16 nodos como se hizo en (Campos y Scherson, 1998). Una vez cargados los procesos el programa entra en dos ciclos. El primero ciclo se terminará cuando la carga también se halla terminado, es decir mientras exista un solo proceso en el cluster este ciclo se estará realizando. El segundo es un ciclo que se repite *numnodos* veces, en este ciclo se llama a la función *Scheduling*(*nodo*) que simula ser el subsistema de administración de procesos, esta función es la que se va a explicar en la siguiente sección. Al final de la función *Simula*() se llama a la función *CalculaTiemposFinales*() que simplemente, recopila la información generada durante

la simulación y que está almacenada en la estructura de cada uno de los procesos, para calcular el tiempo de ejecución de la carga de trabajo.

VI.3.4 Scheduling( )

La función *Scheduling*() simula ser el subsistema de administración de procesos que se encarga de asignar intervalos de tiempo a cada proceso siguiendo una política preestablecida, también verifica el fin de la ejecución de algún proceso, verifica la llegada o creación de nuevos procesos y está involucrada directamente con el BCT.

```
Scheduling(currentnodo)
{
    si (cluster[currentnodo].carga > NUMMAXPROCS)
    {
      mensaje(tabla de procesos llena!!!);
      cluster[currentnodo].carga = NUMMAXPROCS;
    }

si (cluster[currentnodo].carga > 0)
    {
       por una rabanada de tiempo o tabla[currentnodo][0]].tremain > 0
       {
         workload[tabla[currentnodo][0]].tremain -= dec;
         workload[tabla[currentnodo][0]].tcpu += dec;
         si(workload[tabla[currentnodo][0]].son > 0)
         {
            crea un proceso hijo en la tabla del nodo actual;
            workload[tabla[currentnodo][0]].son--;
            cluster[currentnodo].carga++;
         }
         para cada proceso en el nodo actual
           incrementa una unidad por estar en la tabla de listos;
       }
       si (tabla[currentnodo][0]].tremain > 0)
       {
         manda al final de la cola al proceso actual;
         corrimiento en la tabla de procesos;
```

```
        }
        else
          cluster[currentnodo].carga--;
    }

#ifdef RoC
    RoCLB();
#endif

#ifdef MOSIX
    MOSIXLB();
#endif

    RecibeProcesos(currentnodo);
}
```

Si vemos el pseudocódigo anterior, la función *Scheduling* tiene un argumento, *currentnodo*. Este argumento es una variable que indica el nodo en el cual se está haciendo la planificación de procesos, también sirve para conocer la carga, manipular la tabla de procesos y balancear la carga, todo, sobre el nodo actual.

La función *Scheduling* es la más importante debido a todas las actividades que se realizan dentro de ella y porque es el punto en donde se hace referencia a los algoritmos de balanceo de la carga de MOSIX y *Rate of Change*.

En determinadas situaciones, es posible que la tabla de procesos del nodo actual se sature y en el caso de que la cantidad de procesos sobrepase el número permitido, se mandará un mensaje y se reducirá la carga al número máximo de procesos como lo haría un sistema real en el momento en que un proceso quiere crear nuevos procesos y la tabla de procesos ya está llena.

Por otro lado, si no han llegado procesos y dado que la mayoría de los nodos al inicio no tienen carga, se verifica la carga actual. Si la carga es 0, entonces automáticamente se intenta balancear la carga o en otras palabras se solicita carga a otros nodos que puedan compartir la suya. De esta forma se evita hacer el trabajo de planificación en

un nodo que no tiene carga.

Si la carga es mayor que 0, entonces se ejecuta el proceso que se encuentra en la localidad 0 de la tabla de procesos del nodo actual. La ejecución dura una rebanada de tiempo, o menos si el proceso terminó su ejecución. Esta ejecución se simula al decrementar el número indicado en el campo *tremain* de la estructura que contiene la información de un proceso, dicha estructura se mostró en la sección VI.3. El campo *tremain* al inicio de la simulación tiene el valor de t_i, que es el tiempo de ejecución proporcionado en la entrada. El decremento *dec*, es el que hace la diferencia entre un procesador rápido y un procesador lento como se explicó en la sección VI.2. Al mismo tiempo que se decrementa *tremain* se incrementa *tcpu* que es otro de los campos de la estructura del proceso, este último campo es usado en "RoC-LB" al seleccionar a los procesos con mayor tiempo de CPU.

Durante su "estancia en el procesador", un proceso puede crear algún proceso hijo si así lo indica el campo *son* de la estructura del proceso. El proceso creado es idéntico al proceso padre, es decir, se copia la información del padre en una nueva entrada de la tabla de procesos del nodo actual. Lo anterior simula el resultado de la llamada al sistema *fork* que crea un proceso nuevo, el cual es una copia del proceso padre. Si un proceso hijo es creado, entonces se tiene que incrementar en una unidad la carga del nodo actual.

Que un proceso esté en ejecución, quiere decir que los demás procesos que se encuentran en la cola de listos están esperando su turno para ejecutarse, esto obviamente implica un retardo o tiempo extra en el tiempo de finalización de cada proceso. Esto último se tiene contemplado en la simulación al incrementar una unidad de tiempo a todos los procesos que están en la cola de listos cada vez que se decrementa *tremain* del proceso que está en ejecución.

Una vez que se terminó la rebanada de tiempo de un proceso, se verifica *tremain*

para saber si el proceso terminó su ejecución o aún necesita regresar al procesador a finalizar su ejecución. Si $tremain \leq 0$, entonces el proceso terminó su ejecución y hay que eliminarlo de la tabla de procesos y decrementar la carga. Pero si $tremain > 0$, entonces el proceso aún no termina su ejecución, por lo que es mandado al final de la tabla de procesos y se hace un corrimiento en la misma tabla para que el proceso que estaba en la localidad 1 pase a la localidad 0 y pueda ejecutarse.

En este preciso momento, en el cual posiblemente ya se recibieron procesos, ya se crearon nuevos procesos, ya se eliminaron procesos o se planificaron otros, es cuando se utiliza alguna de las funciones que simulan el subsistema de balanceo de la carga. Esto no quiere decir que durante la ejecución se pueda seleccionar el algoritmo con el que se va a balancear la carga, sino que antes de compilar el probrama se elige el algoritmo para simular la ejecución y el balanceo. Estas funciones se describirán brevemente en las siguientes secciones.

Al final de la función *Scheduling* se llama a la función *RecibeProcesos*(*current-nodo*), la cual verifica si hay nuevos procesos que han inmigrado de otros nodos, si la respuesta es afirmativa entonces los procesos recién llegados son ubicados al final de la tabla de procesos del nodo actual. Que algún proceso llegue de otro nodo implica que dicho proceso viajó por la red y pasó algún tiempo fuera de cualquier tabla de procesos, en otras palabras, el proceso no está ejecutándose. El viaje y el retraso que esto implica, también lo simulamos utilizando un contador que se va decrementando en una unidad cada rebanada de tiempo. Cuando el contador es cero entonces se asume que el proceso ha llegado al nodo destino y ahora pasa a ser parte de la carga de dicho nodo y es ubicado en la correspondiente tabla de procesos.

VI.3.5 RoCLB( )

En el pseudocódigo siguiente se presenta la función *RoCLB*(), en dicha función se muestra la adaptación que se hizo del algoritmo *Rate of Change* para nuestra simulación.

```
RoCLB( )
{
    si (ha transcurrido un SI)
      LoadBalancingDecision( ); /* cuando? y donde? */
    si (hay carga nueva)
      actualiza tablas sink y source;
    si (hay solicitudes de carga)
      RequestFilling( );        /* cuantas? y cuales? */
}
```

Cada vez que se cumple un intervalo de muestra (SI) se llama a la función *LoadBalancingDecision*(), la cual se encarga de tomar la decisión de iniciar una solicitud de carga y seleccionar al nodo que se le hará dicha solicitud, que corresponden a las etapas ¿Cuándo? y ¿Dónde? descritas en las secciones IV.2.1 y IV.2.2, respectivamente. Una vez hecha una solicitud de carga no se puede iniciar antes de que la solicitud inicial halla sido reenviada al 8 veces. Si se llega a este punto quiere decir que no hay carga suficiente en el cluster para distribuirla entre todos los nodos y al fijar un número máximo de reenvíos se evita que un mensaje ande viajando de nodo en nodo indefinidamente o que en algún momento llegue al nodo que inicio dicha solicitud.

Después de un tiempo llega la carga solicitada, si es que la hubo, y se tienen que actualizar las tablas *sink* y *source* para que el balanceo pueda continuar de manera correcta.

Debido a que este algoritmo es distribuido, es posible que mientras unos nodos están solicitando carga, otros nodos puedan atender solicitudes de carga. Esto se hace en la función *RequestFilling*(), en donde un nodo que tiene suficiente carga, es decir, el

número de unidades de carga es mayor que su HT, intentará satisfacer alguna o algunas solicitudes de carga. Esta actividad corresponde a las etapas ¿Cuántas? y ¿Cuáles? de las secciones IV.2.3 y IV.2.4, respectivamente.

En el capítulo IV se presentan el funcionamiento interno del algoritmo *Rate of Change* que corresponde a la función *RoCLB*() y en (Scherson y Campos, 1998) se muestran algunas funciones del algoritmo.

VI.3.6 MOSIXLB( )

La última función que se describirá en esta sección es la que simula el subsistema de balanceo de la carga de MOSIX. El pseudocódigo siguiente corresponde a dicha función.

```
MOSIXLB( )
{
    CalculaCargas( );
    DistribuyeInfo( );
    LoadBalance( );
}
```

Como se mencionó en le capítulo V, el balanceo de la carga en MOSIX se realiza en tres etapas jerárquicas, es por eso que en la simulación lo planteamos de la misma forma. En el pseudocódigo de la función *MOSIXLB*() podemos ver que la primera etapa consiste en calcular las cargas, local y exportada. Esta etapa se hace en la función *CalculaCargas*() y la información calculada es utilizada en la siguiente etapa, en la cual se va a distribuir la información de la carga de los nodos. Esta segunda etapa se realiza en la función *DistribuyeInfo*(). Finalmente, cuando un procesador recibe información de carga, intentará balancear la carga tal y como se describió en la sección V.7, esto se hace en la función *LoadBalance*().

Capítulo VII

Ejecución, Simulación y Resultados

VII.1 Introducción

En este capítulo se describe todo el trabajo práctico que se hizo después de la documentación e investigación, específicamente describimos el cluster que se construyó, el hardware y el software que lo componen, un experimento que se hizo al ejecutar varios programas, por medio de otro programa (*script*) ejecutado por el interprete de comandos en MOSIX, también describimos la simulación de los algoritmos de balanceo y finalmente todos los resultados obtenidos.

VII.2 Cluster Of Personal computerS (COPS)

En el Laboratorio de Cómputo Paralelo (PARALAB), el cual está ubicado en el Departamento de Ciencias de la Computación del Centro de Investigación Científica de Educación Superior de Ensenada (CICESE) se construyó un cluster de PC's, del tipo BEOWULF, el cual fue llamado **C**luster **O**f **P**ersonal computer**S** o **COPS**. Este cluster está formado con 3 computadoras y el servidor de PARALAB que también hace la función de controlador de entradas y flujos de datos (*gateway*) entre el cluster y otras redes. Las computadoras del cluster están interconectadas por medio de un *HUB* de 16 puertos, el cual transmite información a 10 Mbps. Lo anterior quiere decir que este

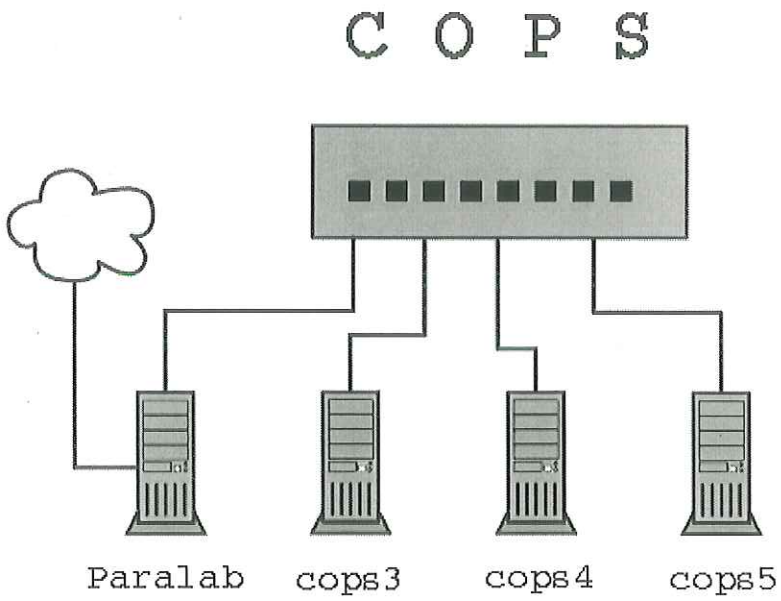

Figura 12: Configuración inicial de COPS en el Laboratorio de Cómputo Paralelo

cluster puede ser expandido hasta tener 16 computadoras conectadas trabajando como un solo sistema.

En la figura 12 se muestra la configuración de COPS. En esta figura se muestran 4 nodos , los cuales corresponden a PARALAB o COPS1, COPS3, COPS4 y COPS5. Las características de cada uno de los nodos son las siguientes:

PARALAB o COPS1.- Esta computadora es un sistema dual o SMP, es decir, tiene 2 procesadores Pentium Pro a 200 MHz, tiene 64 MB en RAM y dos tarjetas de red 3COM a 10 Mbps. Una tarjeta es para conectarse a COPS y la otra es para conectarse hacia el exterior.

COPS3.- Tiene un Procesador Cyrix 6x86 a 66 MHz, tiene 32 MB en RAM y tarjeta de red SMC a 10 Mbps.

COPS4.- Cuenta con un procesador Pentium MMX a 166 MHz, tiene 32 MB en RAM y una tarjeta de red SMC a 10 Mbps.

COPS5.- El procesador de este nodo es Pentium MMX a 166 MHz, tiene 48 MB en RAM y una tarjeta de red 3COM a 10 Mbps.

Ahora describiremos el software que está instalado en COPS. Cada nodo, excepto COPS1, tienen dos sistemas operativos, Linux y MOSIX. COPS1 no tiene MOSIX debido a que es el servidor de páginas WEB de PARALAB y como éste está conectado a la red externa, entonces era posible que llegaran solicitudes del exterior o alguno de los usuarios se conectara remotamente y ejecutara algún proceso, lo cual podría entorpecer los experimentos de balanceo y afectar los resultados. Tener dos sistemas operativos en los otros nodos de COPS significa que es posible iniciar cada nodo con el sistema operativo deseado y utilizar los recursos del cluster sólo con algún mecanismo de paso de mensajes. Aunque no es conveniente que unos sean iniciados con MOSIX y otros con LINUX ya que esto limitaría el balanceo automático de la carga de trabajo exclusivamente a los nodos que tienen MOSIX.

La distribución de Linux instalada en COPS3, COPS4 y COPS5 es Mandrake 7.0 y para COPS1 es Red Hat 6.2. Estas dos distribuciones tienen la versión 2.2.14 del de LINUX la cual es necesaria para la versión de MOSIX que se instaló. Es así que la distribución de Mandrake 7.0 sirvió como base para instalar MOSIX versión 0.97.1 en los nodos indicados anteriormente.

Finalmente, para poder realizar algún tipo de procesamiento paralelo en COPS, se instaló PVM en su versión 3.4.

VII.3 Ejecución de Metabenchmark en COPS

Una vez instalado el ambiente de trabajo, es decir, COPS con el software indicado en la sección anterior, era necesario probar el funcionamiento de MOSIX. Para esto es hizo un *script* al cual le llamamos *Metabenchmark* y que describiremos a continuación.

VII.3.1 Metabenchmark

Para probar el funcionamiento de MOSIX, en particular del balanceo de la carga de trabajo que se hace en MOSIX, era necesario crear varios procesos, ejecutarlos al mismo tiempo y que el sistema operativo, en este caso MOSIX, se encargara de balancear la carga entre los nodos del cluster distribuyendo los procesos entre los nodos de COPS. Pensamos que era necesario tener una carga de trabajo por lo menos de $2n$ procesos en ejecución y que tarden el tiempo suficiente para que MOSIX pueda realizar la operación de balanceo, en este caso n es el número de nodos y como sólo tenemos 3 nodos, entonces es suficiente tener 6 procesos en ejecución.

Para la ejecución simultanea de varios procesos usamos un archivo *script* en el cual se hizo el llamado o invocación, uno por uno, de cada programa de tal forma que la ejecución de los procesos se realizara concurrentemente y con la entrada y salida redireccionadad *background*, es decir, después de la creación de un proceso el control es regresado al interprete de comandos *shell* para poner a un nuevo proceso en ejecución. En el *script* también se mide el tiempo que tarda cada uno de los programas en ejecución, este tiempo será parte de los resultados que estaremos comparando.

Ya que necesitamos varios programas para ejecutar en el cluster, se propuso que estos programas fueran algunos de los *benchmarks* mencionados en la sección II.4.1, lo cual nos lleva a tener un *benchmark* hecho con *benchmarks*. Esta es la razón por la cual llamamos a nuestro *script*: *Metabenchmark*.

Nuestra carga de trabajo está compuesta por 12 programas, en los cuales algunos procesos crean varios procesos hijos. En este caso no nos importa el número de procesos exactos, más bien lo que importa es ejecutar la carga de trabajo en un nodo, en dos, en tres y observar el cambio en el tiempo de ejecución de la carga de trabajo, o en otras palabras lo que nos interesa es el incremento en el rendimiento al aumentar más nodos.

En el *script*, todos los programas fueron ejecutados o creados en un solo nodo,

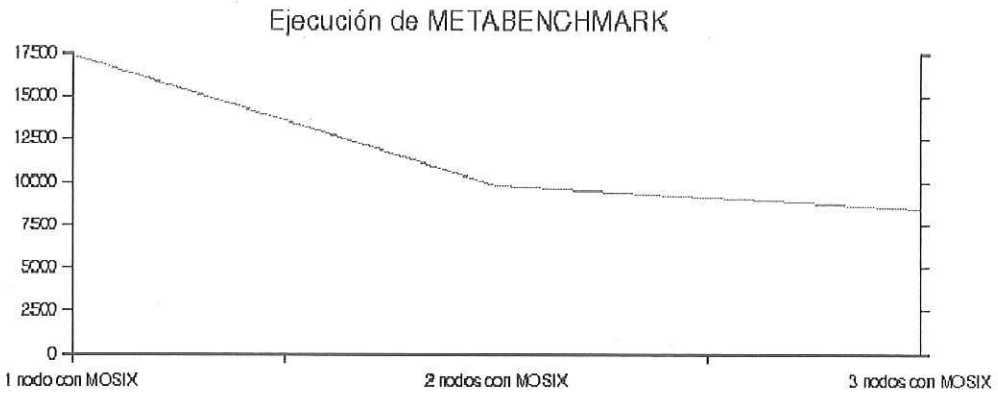

Figura 13: Tiempo de ejecución de Metabenchmark en COPS, cuando se tiene 1 nodo, dos nodos y tres nodos.

en este caso se ejecutan los procesos en el nodo donde nos conectamos al sistema. Después de la creación de los procesos, el sistema los distribuye transparentemente entre los nodos existentes y activos. En MOSIX es posible "bloquear" algunos nodos, es decir, se puede impedir la inmigración y ejecución de cualquier proceso. De esta forma la ejecución en un nodo se hizo bloqueando los otros dos, la ejecución en dos nodos se hizo bloqueando el tercero y finalmente la ejecución en tres nodos se hizo sin bloquear ningún nodo.

La gráfica 13 muestra los resultados de ejecutar *Metabenchmark* en uno, en dos y en tres nodos. La ejecución de la carga de trabajo en un nodo se terminó en 17377 segundos. La misma carga de trabajo, pero en dos nodos, terminó de ejecutarse en 9771 segundos, el tiempo de ejecución de la carga en dos nodos se redujo en comparación con la ejecución en un nodo. Finalmente, al ejecutar la carga de trabajo en 3 nodos tenemos que el tiempo de ejecución de la carga es 8412. Se volvió a decrementar el tiempo de ejecución de la carga en comparación con las dos ejecuciones anteriores.

Los resultados obtenidos en el experimento anterior nos permiten plantear una extrapolación del tiempo de ejecución de la carga de trabajo al ser ejecutada en un cluster con más nodos de los existentes.

Así, si n es el número de procesos que se están ejecutando concurrentemente, m representa el número de nodos totales en el cluster, t_i es el tiempo que se tarda en terminar su ejecución el proceso p_i cuando se ejecuta sólo en el nodo más rápido del cluster ($1 \leq i \leq n$). Ahora, si t_{ik} es el tiempo de finalización del proceso p_i cuando se ejecutan al mismo tiempo los m procesos en k nodos del cluster ($1 \leq k \leq m$), entonces podemos asegurar que $t_i \leq t_{ik}$ y entre menor sea k, esto es, entre menor sea el número de nodos, mayor será la diferencia entre t_i y t_{ik}.

Por otro lado, si $k = m$, entonces $t_{ik} \approx t_i$, pero t_{ik} nunca será menor que t_i y aún más si pudiéramos agregar más nodos de los m existentes, podemos asegurar que no se obtendrá ningún beneficio o ganancia al tener más procesadores que procesos pues tendríamos al menos un proceso en cada procesador y sabemos que t_i es lo más rápido que se puede ejecutar el proceso p_i en el cluster.

Por ejemplo, supongamos que tenemos una carga de trabajo con 5 procesos, que tenemos 5 nodos y que $t_3 = 30$ es el tiempo de ejecución individual del proceso 3. Si ejecutamos los 5 procesos en un solo nodo, todos los procesos estarán compartiendo un solo procesador, lo que implica que el tiempo de ejecución de cada proceso se incrementará. Supongamos que el proceso 3 es el último en terminar su ejecución, por lo que su tiempo de ejecución será también el tiempo de ejecución de la carga como se planteó en la sección VI.2.2, entonces supongamos que $t_3 = 120$.

Si ejecutamos los mismos 5 procesos en 2 nodos, tendremos que $t_3 = 80$ y si repetimos el experimento hasta ejecutar los 5 procesos en los 5 nodos tendremos que $t_3 \approx 30$. Lo anterior suena obvio, pues ahora tenemos en ejecución un proceso en cada nodo. Decimos que $t_3 \approx 30$ ya que se tiene que tomar en cuenta el tiempo extra que se requiere para balancear y migrar los procesos.

En la figura 14 se presenta una gráfica en donde se pueden observar los tiempos de ejecución que debería tener el proceso 3 del ejemplo anterior. Los números que se

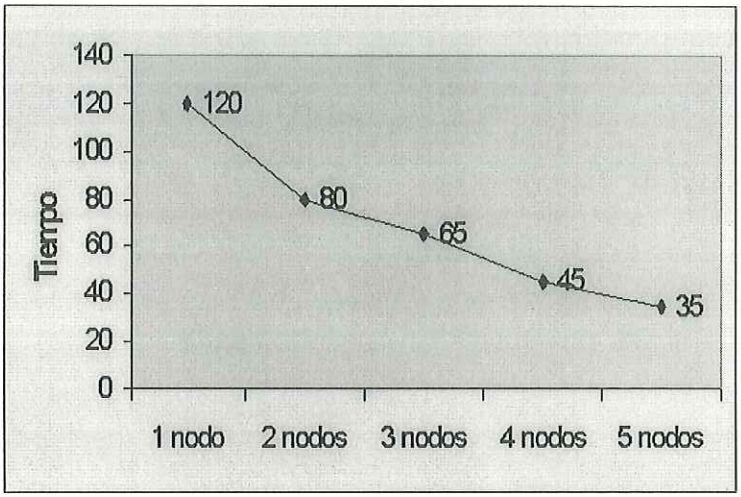

Figura 14: Decremento del tiempo de ejecución de un proceso cuando se ejecuta al mismo tiempo que otros procesos en varios nodos

presentan en la gráfica no son reales, son ficticios, pero tratan de mostrar lo planteado anteriormente. Desafortunadamente, como se ve en la figura 13, sólo pudimos comprobar esta teoría hasta tres nodos, que eran los que formaban nuestro cluster.

VII.4 Simulación de los Algoritmos

En las siguientes secciones describiremos la simulación de la ejecución de una carga de trabajo en un cluster de PC's utilizando dos mecanismos de balanceo de la carga de trabajo como lo son RoC-LB y el subsistema de balanceo de MOSIX.

VII.4.1 Descripción del Escenario

Como se dijo en la sección VI.2, en la simulación se tienen 2 tipos de entradas. El primer tipo de entrada está relacionado con el cluster y el segundo tipo está relacionado con la carga de trabajo.

Como parte de nuestra simulación tenemos un archivo llamado COPS el cual contiene las características de 16 nodos, los cuales representan al cluster sobre el cual se ejecuta cierta carga de trabajo. A continuación mostramos el contenido de nuestro archivo

"COPS:

#	NUMEP	TIPO	SPEED	RAM
1	1	PentiumIII	13000	32
2	1	PentiumIII	13000	32
3	1	PentiumIII	13000	32
4	1	PentiumIII	13000	32
5	1	PentiumIII	13000	32
6	1	PentiumIII	13000	32
7	1	PentiumIII	13000	32
8	1	PentiumIII	13000	32
9	1	PentiumIII	13000	32
10	1	PentiumIII	13000	32
11	1	PentiumIII	13000	32
12	1	PentiumIII	13000	32
13	1	PentiumIII	13000	32
14	1	PentiumIII	13000	32
15	1	PentiumIII	13000	32
16	1	PentiumIII	13000	32

Lo anterior muestra que nuestro cluster es homogéneo, es decir, todos los nodos tienen las mismas características, lo único que tienen diferente y por razones obvias, es el número que los identifica. Aquí hay que aclarar que este "cluster" fue el que utilizamos para todos nuestros experimentos, tanto para RoC-LB como para MOSIX, y que no probamos con configuraciones heterogéneas.

El archivo NODOS contiene un número entero entre 1 y 16 e indica el número de nodos que tiene nuestro cluster. Es así que con los dos archivos mencionados anteriormente podemos representar clusters de hasta 16 nodos. En las pruebas de *speed up* bastaba con ir modificando el archivo NODOS para indicar el número de nodos que teníamos disponibles para la ejecución de la carga de trabajo.

En cuanto a la carga de trabajo, los archivos PROCESSES e INFOPROCESS contienen la información de cada uno de los procesos que la forman. El archivo PROCESSES contiene el número de procesos que queremos simular y el archivo INFOPROCESS contiene las características de cada uno de los procesos. Para ejecutar distintas cargas en dis-

Tabla I: Parámetros importantes para simular RoC-LB

Nombre	Valor
Tamaño tabla SINK	5
Tamaño tabla SOURCE	5
HT	25
LT	10
CT	5
SI	2
ND	1 / 2

Tabla II: Parámetros importantes para simular MOSIX

Nombre	Valor
Tamaño tabla de nodos activos	512
Tamaño del vector de carga	8
T	2 / 3 ciclos
STD_SPD	10000

tintos tipos de clusters, basta con modificar los archivos correspondientes y ejecutar el mismo programa. Además del número de nodos y de procesos, también se puede seleccionar el número de nodos en los cuales podemos iniciar la carga de trabajo, esto se hace por medio de la constante NODOSINI.

En cuanto a los tamaños de las tablas y parámetros importantes en RoC-LB presentamos esa información en la tabla I. Los valores de esta tabla son los mismos que los que utilizaron en (Campos y Scherson, 1998), excepto por ND que nosotros mantuvimos constante. De igual forma en la tabla II se presentan los parámetros importantes para MOSIX, los cuales fueron tomados de (Barak et al, 1993) y del código fuente de MOSIX.

VII.4.2 Experimentos con RoC-LB

En la tabla III se muestran los experimentos que se realizaron utilizando RoC-LB.

Tabla III: Experimentos realizados con RoC-LB

Experimento	1	2	3	4-1	4-2	4-3	5-1	5-2	5-3
No. procesos	2000	2000	2000	1000	1000	1000	1000	1000	1000
ti	1	1	1-9	1	1	1	2	2	2
Tamaño	1k	1k	1-10 k's	1k	1k	1k	1k	1k	1k
Nodos	16	16	16	16	16	16	16	16	16
Nodos con carga	1	1	1	1	8	16	1	8	16
ND	1	2	2	1	1	1	1	1	1
tmig	no	si	si	no	no	no	no	no	no
costomig	no	si	si	no	no	no	no	no	no
SpdNs	no	no	no	no	no	no	no	no	no
bal	no	si	si	no	no	no	no	no	no

En la misma tabla se describen de manera general las condiciones en las cuales se llevó a cabo cada uno de los experimentos, es decir, el número de procesos, el tiempo de ejecución de los procesos, el tamaño de los procesos, el número de nodos del cluster, el número de nodos en los cuales se inició la carga, el tiempo que tardaba en llegar un mensaje de un nodo a otro (ND) o si es que se agregó tiempo extra debido a migración (tmig y costomig), balanceo (bal) y velocidad del procesador donde se ejecutó cada uno de los procesos (SpdNs).

Los experimentos de la tabla III representan a dos tipos, experimentos de control, que permiten de cierta forma verificar el comportamiento correcto de las simulaciones, y los experimentos que no son de control, en los cuales podemos representar o plantear situaciones más complejas y así obtener otro tipo de información. En este caso los experimentos 1, 4-1, 4-2, 4-3, 5-1, 5-2 y 5-3 fueron experimentos de control.

En el experimento 1 se plantea la siguiente situación, se tiene una carga de trabajo de 2000 procesos. El tiempo de ejecución individual es de una unidad y el tamaño de cada proceso también es unitario. Se tienen 16 nodos disponibles en el cluster para aceptar y ejecutar procesos. La carga se inició en un solo nodo. El tiempo que tarda en llegar un mensaje de un nodo a otro es una unidad de tiempo, este tiempo es constante

debido a que tenemos procesos del mismo tamaño y no tiene porque cambiar el tiempo de transmisión. No se agregó tiempo extra debido a la migración de procesos, ni debido a diferencias entre las velocidades de los procesadores de los nodos, pues como dijimos antes, sólo experimentamos con configuraciones homogéneas para el cluster y tampoco agregamos tiempo extra debido a la operación de balanceo. En este experimento se quitaron los tiempos extras porque lo que nos interesaba era verificar que el balanceo y la distribución de la carga se estuviera llevando a cabo correctamente y no nos interesaba el tiempo que tardaba en balancear la carga.

En los experimentos 4-1, 4-2 y 4-3 se plantea una situación similar a la anterior, excepto que el tamaño de la carga es de 1000 procesos y en 4-1 la carga se inicia en 1 nodo, en 4-2 la carga se inicia en 8 nodos y finalmente en 4-3 la carga se inicia en 16 nodos. Estos experimentos representan las 3 situaciones que se plantearon en la sección VI.2.3.

Los experimentos 5-1, 5-2 y 5-3 difieren de los experimentos 4-1, 4-2 y 4-3 en que ahora el tiempo de ejecución de la carga no es unitario sino que es de 2 unidades. Uno podría pensar que estamos hablando de la misma situación pero en realidad en los experimentos donde el tiempo de ejecución es unitario cada proceso termina su ejecución en la primera rebanada de tiempo que le es asignada. Por otro lado, la situación donde el tiempo de ejecución de los procesos es de 2 unidades, corresponde a la situación en donde un proceso no termina su ejecución durante la primera rebanada de tiempo que se le asignó y por lo tanto debe esperar en la tabla de procesos por una nueva oportunidad para terminar dicha ejecución.

En el experimento 2 se plantea la situación en donde 2000 procesos, con tamaño y tiempo de ejecución unitario, son ejecutados en un solo nodo, pero ahora el tiempo que un mensaje pasa en la red no es unitario sino que tarda 2 unidades de tiempo. En este experimento ya se agrega tiempo extra debido a migración y operación de balanceo de

Tabla IV: Resultados de los Experimentos realizados con RoC-LB

Experimento	1	2	3	4-1	4-2	4-3	5-1	5-2	5-3
TnoLB	2000	2000	8463	1000	1000	1000	2000	2000	2000
Topt	125	125	528.38	62.5	62.5	62.5	125	125	125
Tbal	154	198	610.62	95	85	63	165	149	125
NP	0.985	0.961	0.990	0.965	0.976	0.999	0.979	0.987	1.00

la carga al tiempo de ejecución final.

Finalmente el experimento 3 representa una situación en donde tenemos una carga de trabajo de 2000 procesos con tamaños distintos, que van de 1 a 10, y tiempos de ejecución distintos, que van de 1 a 9 unidades, el tiempo que pasa un mensaje en la red es de dos unidades y también se agrega tiempo extra debido a la operación de migración y balanceo.

En la siguiente sección se presentan los resultados que se obtuvieron de los experimentos planteados en la presente sección.

VII.4.3 Resultados de la Simulación con RoC-LB

Después de establecer los valores correspondientes a los parámetros de entrada que se muestran en la tabla III y de ejecutar el programa, obtuvimos los resultados que se muestran en la tabla IV correspondiendo a cada uno de los experimentos antes mencionados. Los valores de esta tabla son los que se describieron en la sección VI.2.2.

En todos los experimentos TnoLB fue calculado cargando todos los procesos en el cluster y quitando las funciones de balanceo y migración, por lo que la carga no tenía más que terminar su ejecución en el nodo o en los nodos donde fue iniciada. Así mismo, Topt fue calculado al dividir TnoLB entre el número de nodos en el cluster, que en todos los casos fueron 16. Tbal es el resultado directo de nuestro programa, este resultado fue calculado como se muestra en las ecuaciones 7 y 8. Para calcular NP se utilizaron los valores de TnoLB, Topt y Tbal calculados previamente de acuerdo a

la ecuación 6.

Para el experimento 1, TnoLB fue igual a 2000 unidades, esto equivale a ejecutar secuencialmente 2000 procesos de una unidad de tiempo en un solo procesador. El resultado que arrojó nuestro programa, Tbal, fue de 154, que a su vez implicó que NP fuera 0.985.

En el experimento 2, NP se redujo a 0.961 al aumentar el tiempo que tarda un mensaje en la red de 1 unidad a 2 unidades, esto nos muestra que al aumentar Tbal, es decir, entre más se acerque Tbal a TnoLB, NP se va a reducir.

Para el experimento 3, donde los tiempos y tamaños de los procesos eran distintos, encontramos que el tiempo que se tarda en ejecutar toda la carga sin balanceo fue 8454, el tiempo óptimo en ejecutar dicha carga fue 528.38, el resultado de nuestro programa fue 610.62 y NP fue igual a 0.990

Los resultados de los experimentos 4-1, 4-2 y 4-3 de cierta forma representan el peor caso, un caso intermedio y el mejor de los casos al inicio de la operación de balanceo. De la tabla IV podemos ver que entre más balanceada esté la carga al inicio de la ejecución, mejores serán los resultados de NP.

Aunque en los experimentos 4-1, 4-2, 4-3, 5-1, 5-2 y 5-3 no se agregan los tiempos extra debido a migración, operación de balanceo y velocidad del procesador, estos reflejan el mismo comportamiento que los que si tienen esos tiempos extras. Esto es muy importante pues es una forma de verificar que el algoritmo de balanceo "Rate of Change" no introduce tiempo extra excesivo al intentar balancear un sistema que ya está balanceado.

A continuación mostramos y describimos unas gráficas las cuales muestran un historial de cómo se fue distribuyendo la carga entre los nodos al ir avanzando la ejecución en los experimentos de la tabla III y el rendimiento que se obtiene al ir modificando el número de nodos en el cluster.

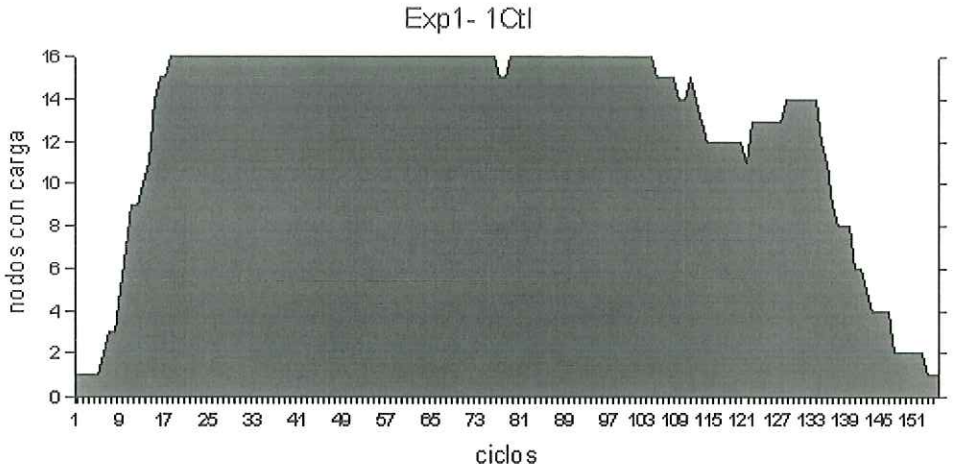

Figura 15: Experimento 1, ejecución de 2000 procesos idénticos iniciados en 1 nodo utilizando "RoC-LB" y con ND=1

En la figura 15 se muestra cómo se fue distribuyendo la carga entre los nodos. La gráfica muestra el número de nodos con carga contra el número de ciclos en los que se monitoreó dicha ejecución. Si vemos, al principio solo un nodo tiene carga. En el ciclo 5, aproximadamente, 2 nodos tienen carga y así transcurre la ejecución hasta que al rededor del ciclo 17 todos los nodos del cluster tienen al menos una unidad de carga. En esta misma gráfica se ve la idea central del algoritmo "Rate of Change" que es mantener ocupados al mayor número de nodos mientras exista carga y en cuanto un nodo se de cuenta que se va a quedar sin carga, tiene que iniciar una solicitud de carga, de tal forma que antes de que se cumpla el plazo en que se quedará sin carga, este mismo nodo ya esté recibiendo la carga previamente solicitada. Al rededor del ciclo 105 la carga se va decrementando y con ella el número de nodos con al menos una unidad y como se ve en el ciclo 121, el algoritmo trata de que el mayor número de nodos tengan carga y en el ciclo 127 al menos 11 nodos tienen una unidad. Finalmente, a partir del ciclo 133 el número de nodos con carga se va reduciendo hasta que los últimos procesos se ejecutan en un solo nodo.

Otro tipo de experimento que también se hizo fue ir modificando el archivo NODOS

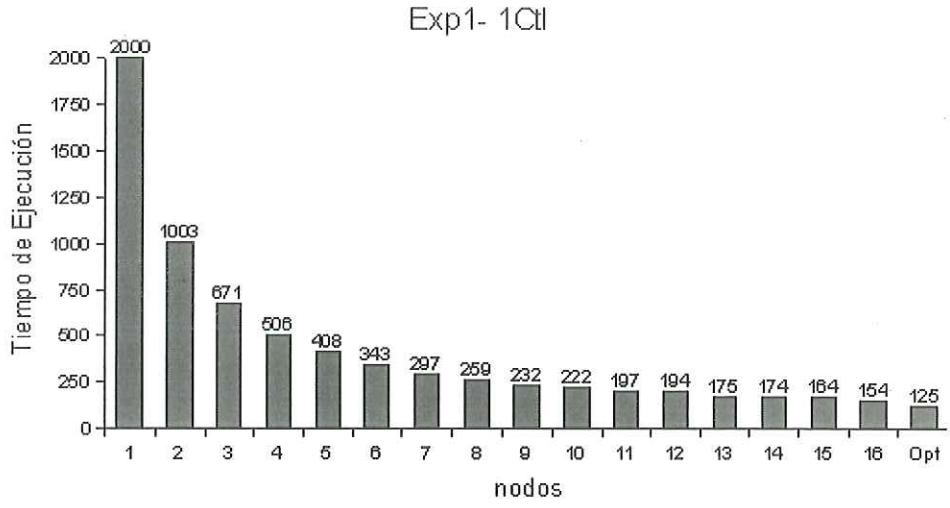

Figura 16: Tiempo de ejecución de la carga de trabajo al ejecutarse en 1 nodo hasta 16 nodos utilizando "RoC-LB"

desde 1 hasta 16 para ver cómo cambiaba el tiempo de ejecución de la carga de trabajo. Los resultados de la ejecución de la carga del experimento 1 se muestran en la figura 16.

El *speed up* es una medida que nos dice cómo se acelera el rendimiento o qué tan rápido se decrementa el tiempo de ejecución de un programa paralelo al ejecutarlo en un número diferente de procesadores. Este *speed up* depende del porcentaje de parte paralela y de parte secuencial que un problema tenga. Nosotros no tenemos un programa paralelo explícitamente creado, pero podemos ver a la carga de trabajo como un programa con casi 100% de parte paralela, siempre y cuando los procesos no dependan unos de otros o no se comuniquen entre ellos. En nuestra simulación no tenemos procesos que se comuniquen o que dependan de otros procesos, por lo que podríamos suponer que el *speed up* seguiría un comportamiento lineal. De esta forma, hicimos un cálculo del *speed up* con los datos mostrados en la figura 16 para observar cómo se comportaba y los resultados de estos cálculos los mostramos en la figura 17. Si bien nuestro *speed up* al principio se aproxima a un *speed up* lineal, llega un momento en que ya no crece al mismo ritmo que el número de nodos. Un experimento que no

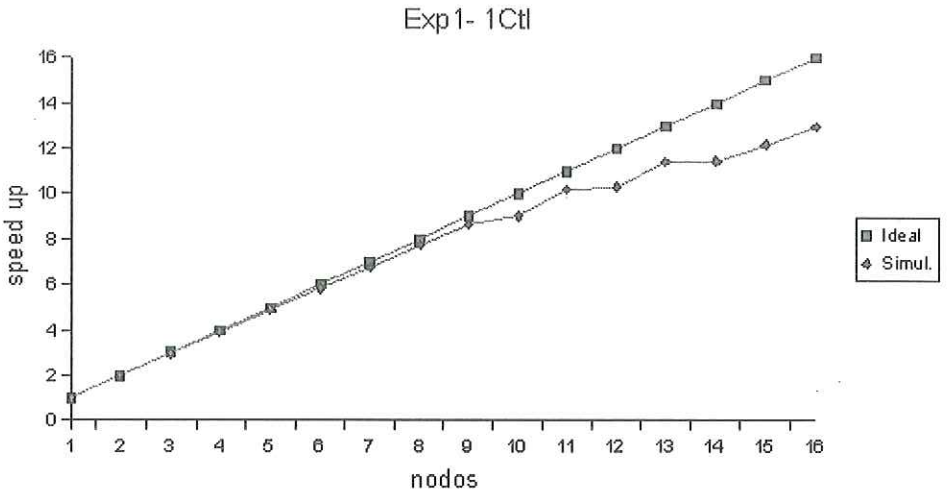

Figura 17: Comparación del speed up ideal y del speed up del experimento 1 con "RoC-LB"

se hizo, pero que sería interesante hacer, es aumentar el número de nodos para ver en qué momento se estabiliza el *speed up* y deja de crecer igual que el número de nodos.

Para cada uno de los experimentos se hicieron los mismos análisis y gráficas que para el experimento 1. En todos el *speed up* siguió el mismo comportamiento que el que se muestra en la gráfica 17, excepto en los experimentos en donde se estableció que el tiempo para los mensajes era de dos unidades, el *speed up* crecía más lento. Lo mismo ocurrió con las ejecuciones de las cargas de todos los experimentos en una cantidad distinta de nodos, es decir, estas gráficas eran muy similares a la gráfica 16.

Las gráficas que no siguieron el mismo comportamiento fueron las que muestran el historial de la distribución de la carga, a continuación mostraremos y describiremos dichas gráficas.

En la figura 18 se muestra la distribución de la carga del experimento 2, en donde el tiempo de viaje de un mensaje era de dos unidades. Pues ésta es precisamente la razón por la cual transcurren más ciclos para que todos los nodos tengan al menos una unidad de carga. Si vemos la figura 15, en el ciclo 17 todos los nodos ya tienen carga y

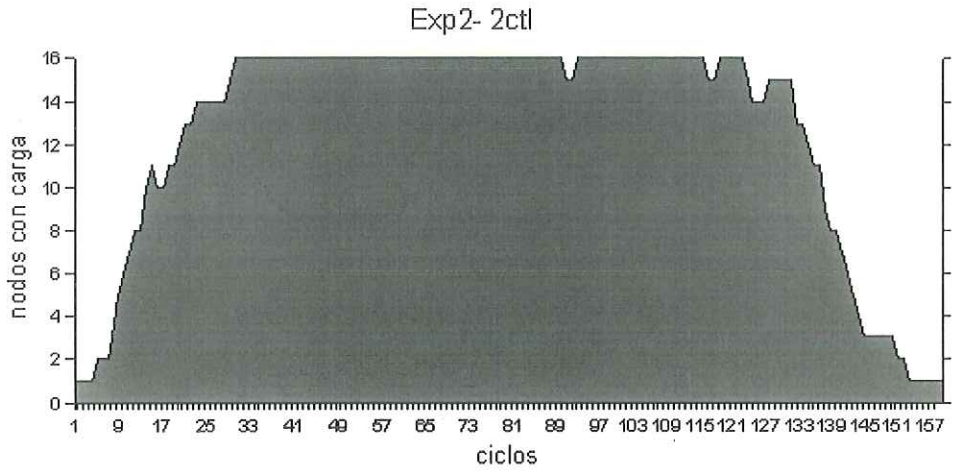

Figura 18: Experimento 2, ejecución de 2000 procesos idénticos iniciados en 1 nodo utilizando "RoC-LB" y con ND=2

en la figura 18 es hasta el ciclo 30 cuando todos los nodos tienen al menos una unidad de carga. Aunque el inicio de la distribución de la carga del experimento 1 es distinto al del experimento 2, el final es muy parecido pues al rededor del ciclo 130 es cuando el número de nodos con carga empieza a diminuir hasta que cerca del ciclo 151 sólo un nodo es el que tiene carga.

En el experimento 3 se tenían 2000 procesos con distintas características, es por eso que la ejecución se realizó en más tiempo que en los experimentos anteriores. La figura 19 muestra la situación anterior, pero lo más importante en esta gráfica vuelve a ser la cantidad de ciclos en los cuales todos los nodos tienen al menos una unidad de carga.

Ahora describiremos las 3 gráficas que corresponden a los experimentos 4-1, 4-2 y 4-3 en los cuales se ejecutaba la misma carga formada por 1000 procesos. En estos casos la carga era iniciada, primero en un nodo, luego en 8 nodos y finalmente en 16 nodos, respectivamente a cada experimento. La figura 20 corresponde al primer caso. Si vemos es muy similar a las gráficas que hemos presentado y donde la carga es iniciada

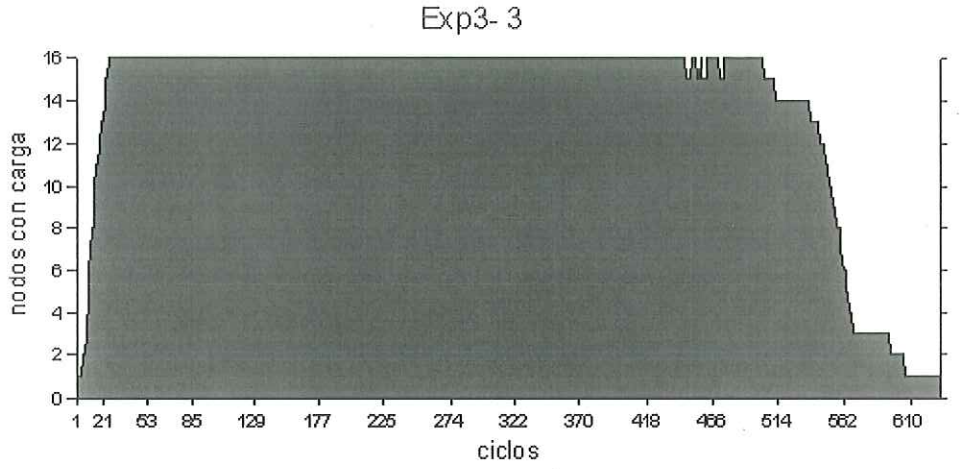

Figura 19: Experimento 3, ejecución de 2000 procesos con tiempos de ejecución y tamaños distintos iniciados en 1 nodo utilizando "RoC-LB"

en un solo nodo. La figura 21 nos muestra la situación en donde la carga es iniciada en 8 nodos, esto hace que en el ciclo 8 todos los nodos tengan al menos una unidad de carga y además el tiempo de ejecución de la carga sea menor que en el experimento 4-1. En la figura 22 se muestra la situación donde los 1000 procesos son distribuidos uniformemente entre los 16 nodos del cluster, es por eso que la ejecución inicia con 16 nodos con carga y termina con 16 nodos con carga. El tiempo de ejecución de la carga es similar o equivalente a ejecutar 63 procesos unitarios en un solo nodo.

Finalmente, en las figuras 23, 24 y 25 se presenta como se distribuyó la carga en los experimentos 5-1, 5-2 y 5-3, respectivamente. Estas figuras siguen el mismo comportamiento que los experimentos 4-1, 4-2 y 4-3, tomando en cuenta que son experimentos muy parecidos, es decir, muestran que entre más balanceada esté la carga al inicio del experimento, mejores serán los resultados.

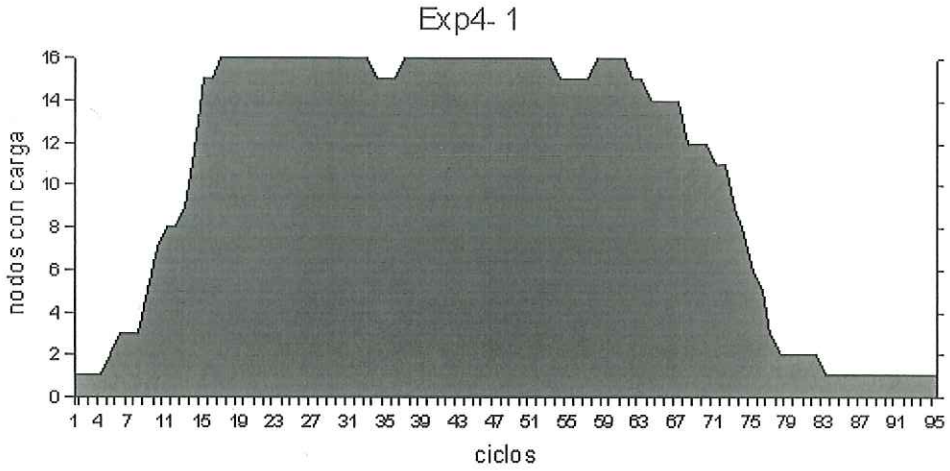

Figura 20: Experimento 4-1, ejecución de 1000 procesos idénticos iniciados en 1 nodo utilizando "RoC-LB" y con ti=1

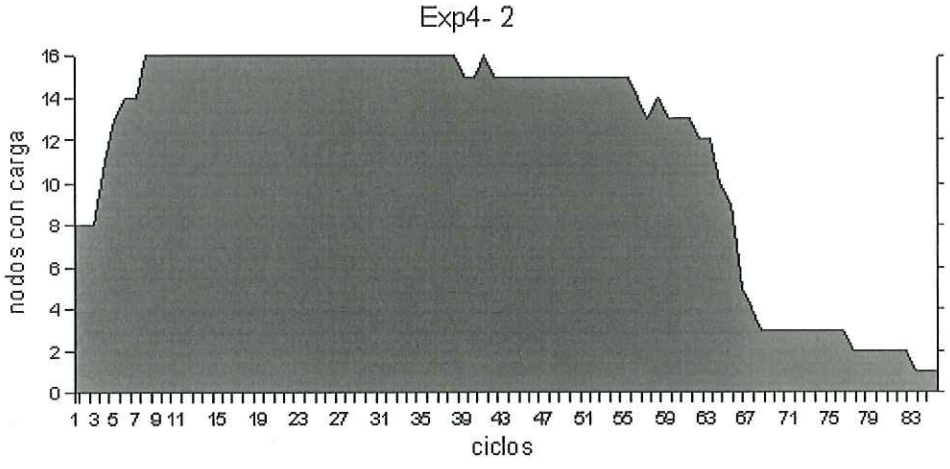

Figura 21: Experimento 4-2, ejecución de 1000 procesos idénticos iniciados en 8 nodos utilizando "RoC-LB" y con ti=1

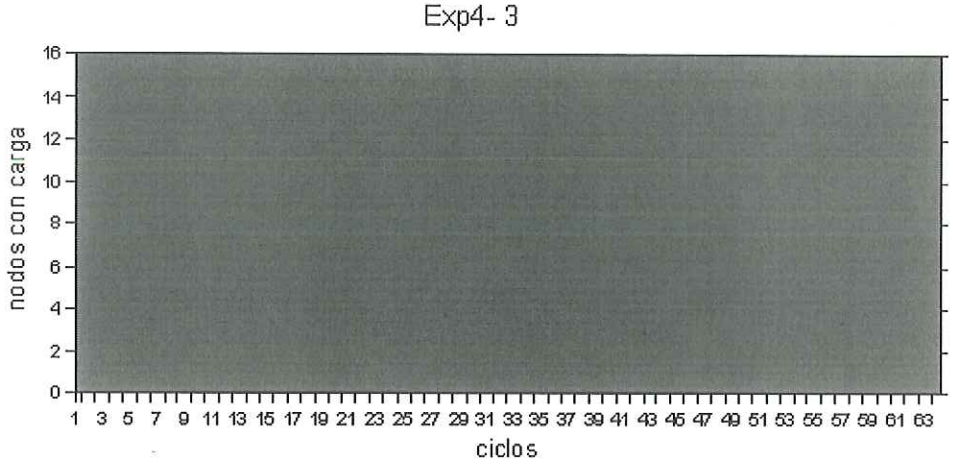

Figura 22: Experimento 4-3, ejecución de 1000 procesos idénticos iniciados en 16 nodos utilizando "RoC-LB" y con ti=1

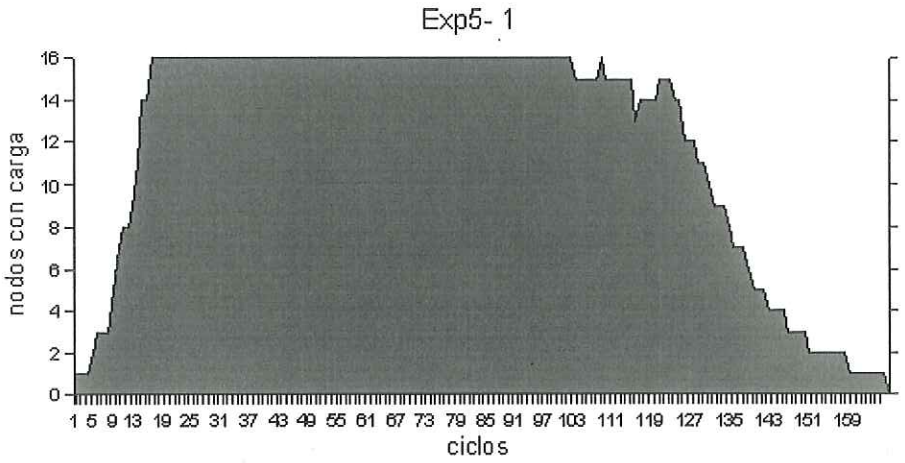

Figura 23: Experimento 5-1, ejecución de 1000 procesos idénticos iniciados en 1 nodo utilizando "RoC-LB" y con ti=2

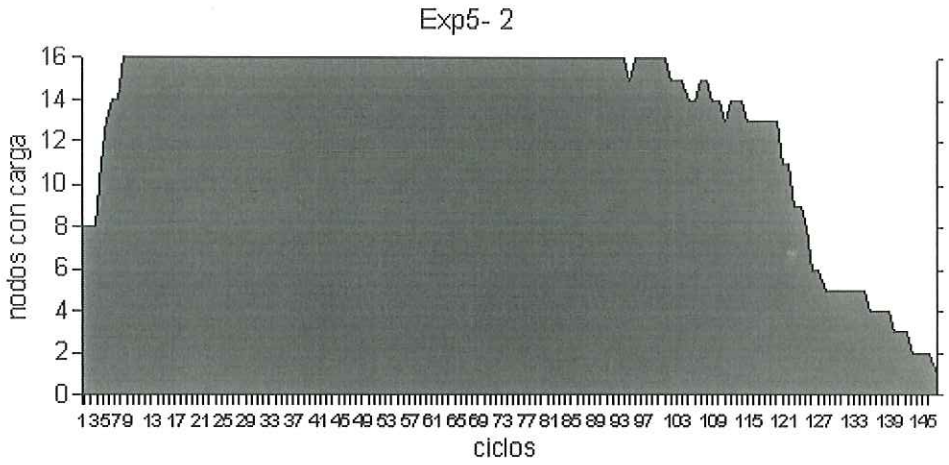

Figura 24: Experimento 5-2, ejecución de 1000 procesos idénticos iniciados en 8 nodos utilizando "RoC-LB" y con ti=2

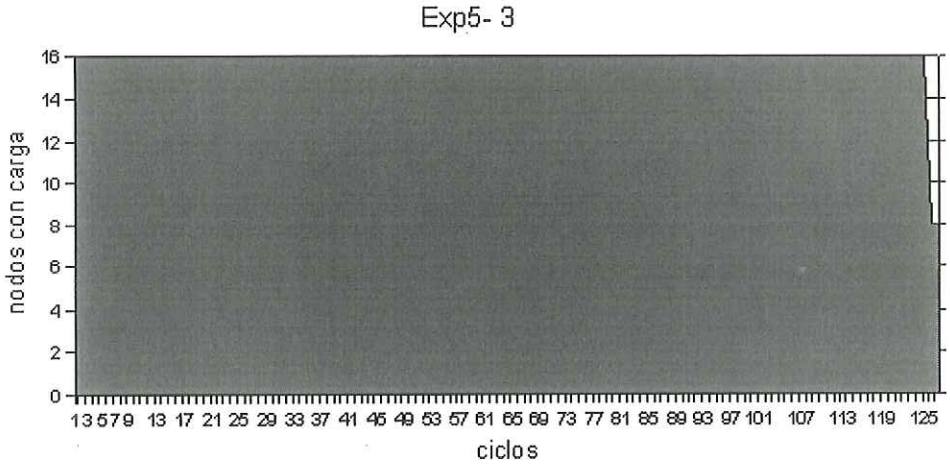

Figura 25: Experimento 5-3, ejecución de 1000 procesos idénticos iniciados en 16 nodos utilizando "RoC-LB" y con ti=2

VII.4.4 Experimentos con MOSIX

Para poder comparar los dos algoritmos realizamos los mismos experimentos que con RoC-LB, por esa razón utilizaremos la tabla III para seguir la secuencia de los experimentos.

VII.4.5 Resultados de la Simulación con MOSIX

Los resultados de los experimentos con MOSIX se muestran en la tabla V. Si comparamos uno a uno los resultados de esta tabla con los resultados de la tabla IV veremos que RoC-LB funcionó mejor que el algoritmo de MOSIX en 8 de los 9 experimentos realizados, específicamente, el experimento 4-2 de MOSIX fue el que tuvo un mejor valor para Tbal y para NP, que en el experimento con RoC-LB.

Nos dimos cuenta que tres factores importantes son los que influyen sobre estos resultados. Primero, la naturaleza de la carga, segundo ciertos factores del algoritmo de MOSIX que fueron omitidos en la simulación y tercero la cantidad de procesos migrados durante cada operación de balanceo. Factores como la "Regla de Residencia" que hace que procesos de, probabilísticamente calculada, corta duración no puedan emigrar a otros nodos pues tardarían más tiempo migrando que en terminar su ejecución. Es así, que en nuestros experimentos, los procesos de una unidad de tiempo eran migrados uno por uno hacia los demás nodos, al llegar al nodo destino se ejecutan una unidad de tiempo y terminan, con lo cual es necesario otra operación de balanceo para llevar carga al mismo nodo. Recordemos que en RoC-LB es posible migrar desde un proceso hasta el número de procesos entre el número de nodos, con lo cual hay más seguridad de que con una operación de balanceo un nodo pueda tener carga suficiente por un buen tiempo y que después de un tiempo, el mismo nodo solicite más carga porque está a punto de quedarse ocioso.

En las gráficas correspondientes a los resultados de MOSIX también se refleja la

Tabla V: Resultados de los Experimentos realizados con MOSIX

Experimento	1	2	3	4-1	4-2	4-3	5-1	5-2	5-3
TnoLB	2000	2000	8463	1000	1000	1000	2000	2000	2000
Topt	125	125	528.38	62.5	62.5	62.5	125	125	125
Tbal	670	671	949	337	68	63	447	131	131
NP	0.709	0.708	0.946	0.706	0.993	0.998	0.828	0.996	0.996

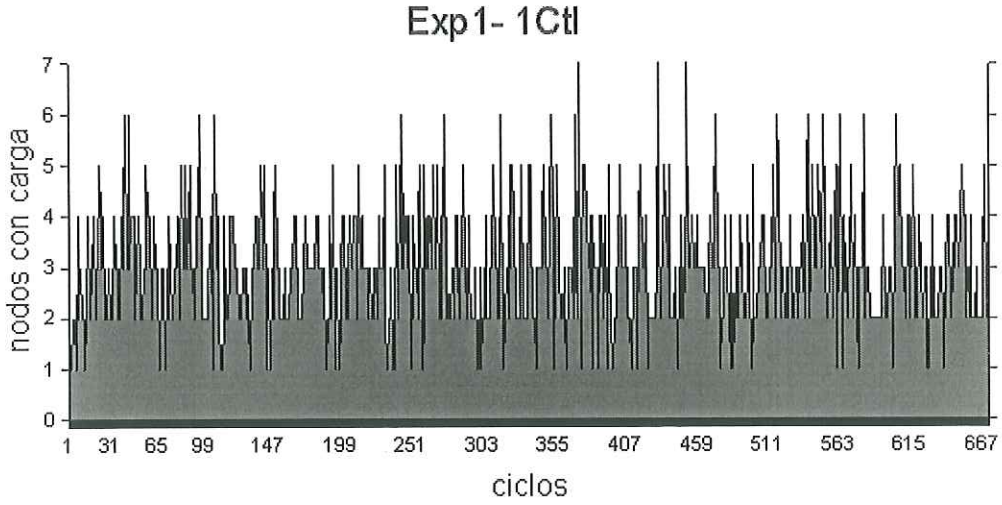

Figura 26: Experimento 1, ejecución de 2000 procesos idénticos iniciados en 1 nodo utilizando MOSIX y con ND=1

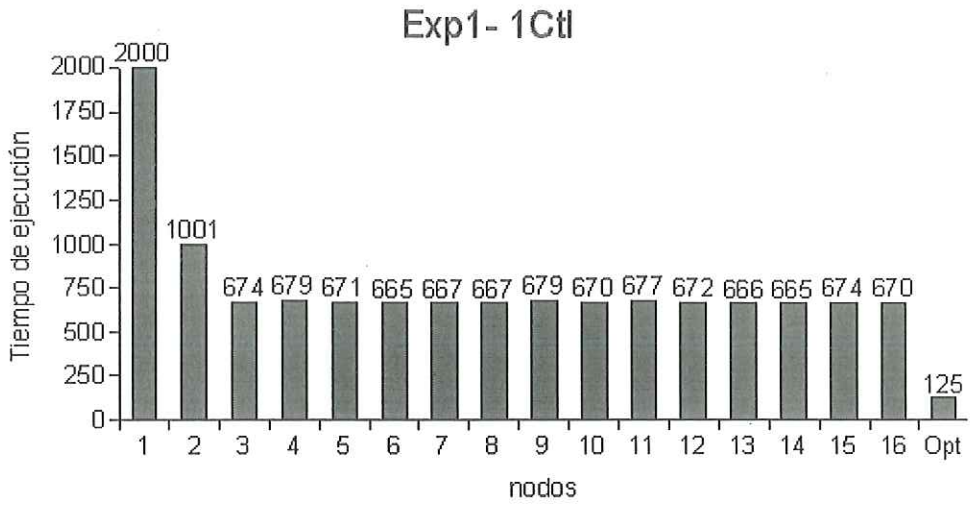

Figura 27: Tiempo de ejecución de la carga de trabajo al ejecutarse en 1 nodo hasta 16 nodos utilizando MOSIX

misma situación. En la figura 26 se muestra un historial de cómo se distribuyó la carga de trabajo entre los nodos durante la ejecución. Si observamos, a lo más 7 nodos son los que llegan a tener al menos una unidad de carga al mismo tiempo, lo cual no es muy bueno pues no se están utilizando, de manera eficiente, todos los nodos que se tienen disponibles en el cluster, además el número de nodos con carga cambia brúscamente de un momento a otro. Lo anterior también hace que el tiempo de ejecución de la carga se más grande, en este mismo experimento RoC-LB terminó en 154 unidades y MOSIX terminó en 670 unidades. En las figuras 27 y 28 se puede observar que en realidad la ejecución de la carga en el experimento 1 utiliza eficientemente a lo más tres nodos. En la figura 27 el tiempo de ejecución disminuye cuando aumentamos de 1 a 2 y a 3 nodos, pero cuando aumentamos 4,5 o hasta 16 nodos, el tiempo de ejecución se mantiene casi constante. En la figura 28 el *speed up* nos muestra lo mismo al detenerse el crecimiento de la curva cerca de 3 unidadess.

En la figura 29 podemos observar el historial de la distribución de la carga para el experimento 2, que es similar al experimento 1 excepto que se incrementó el tiempo que

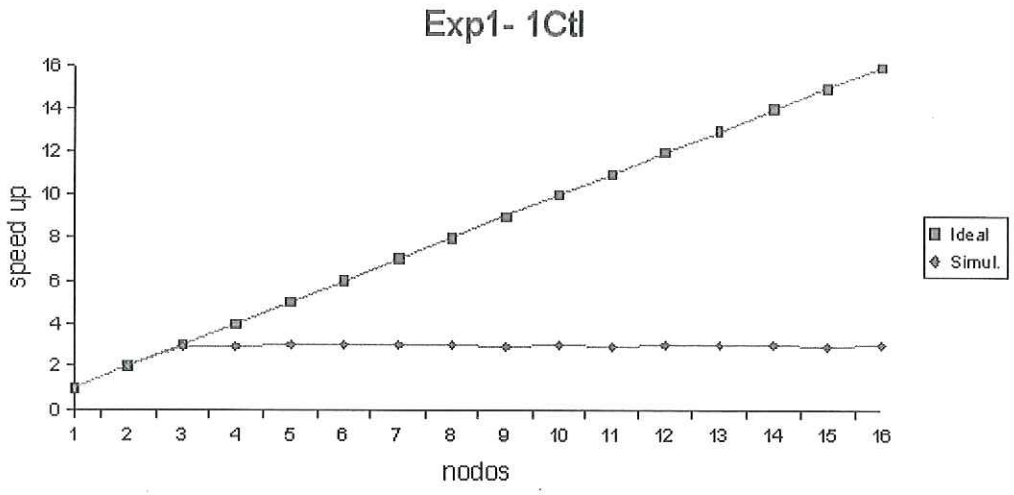

Figura 28: Comparación del speed up ideal y del speed up del experimento 1 con MOSIX

tarda en llegar un mensaje de un nodo a otro. El comportamiento de la distribución también es similar debido a los factores que mencionamos antes.

En el experimento 3, al incrementar el tiempo de ejecución de cada uno de los procesos se logra distribuir mejor la carga de trabajo, pues ahora los procesos al ser migrados y permanecer en ejecución en el nodo destino, dan tiempo al nodo que tiene la carga para poder asignar procesos a otros nodos que no tienen carga. En la figura 30 se ve como en cierto momento se utilizan los 16 nodos, aunque no de manera eficiente.

En las figuras 31 y 32 se muestra que en realidad sólo se está haciendo uso de 9 nodos en el experimento 3-3. En la figura 32 el *speed up* detiene su creciemiento al llegar al nodo 9 y como dijimos esto no es eficiente.

Para los experimentos 4-1, 4-2 y 4-3 en donde analizamos como funcionaba el algoritmo al iniciar la carga en 1, 8 y 16 nodos, los resultados fueron más parecidos a los obtenidos con RoC-LB, es más, en el experimento 4-2 se obtuvieron mejores resultados que con RoC-LB. En la figura 33 se muestra el caso en donde la carga fue iniciada en un

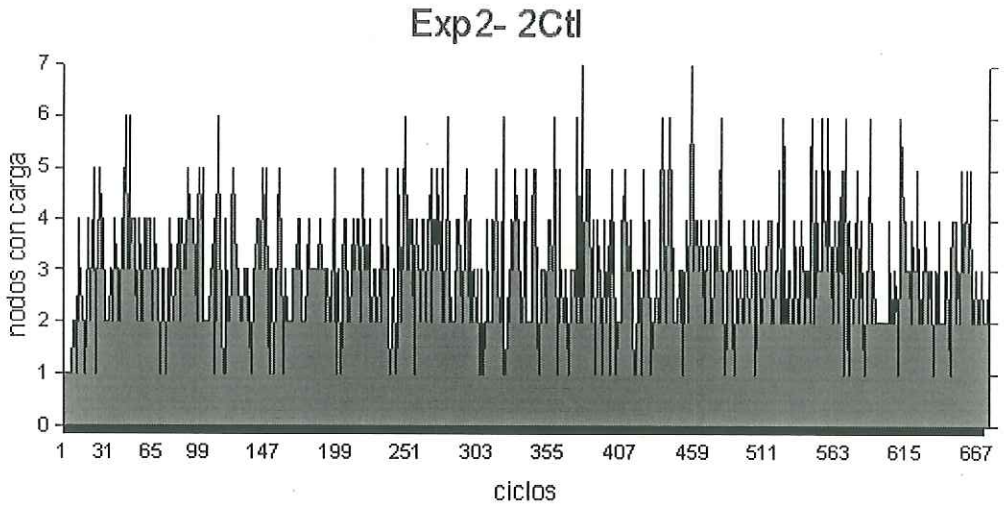

Figura 29: Experimento 2, ejecución de 2000 procesos idénticos iniciados en 1 nodo utilizando MOSIX y con ND=2

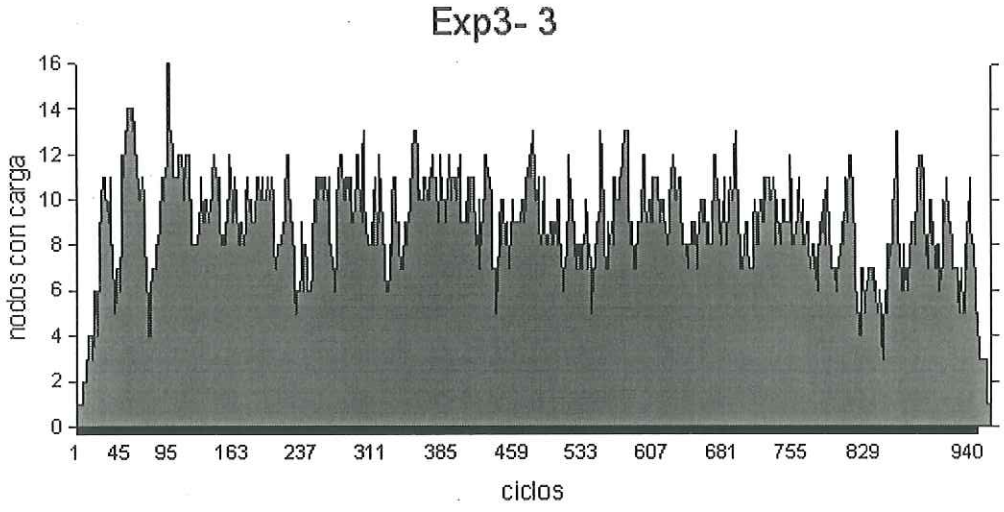

Figura 30: Experimento 3, ejecución de 2000 procesos con tiempos de ejecución y tamaños distintos iniciados en 1 nodo utilizando MOSIX

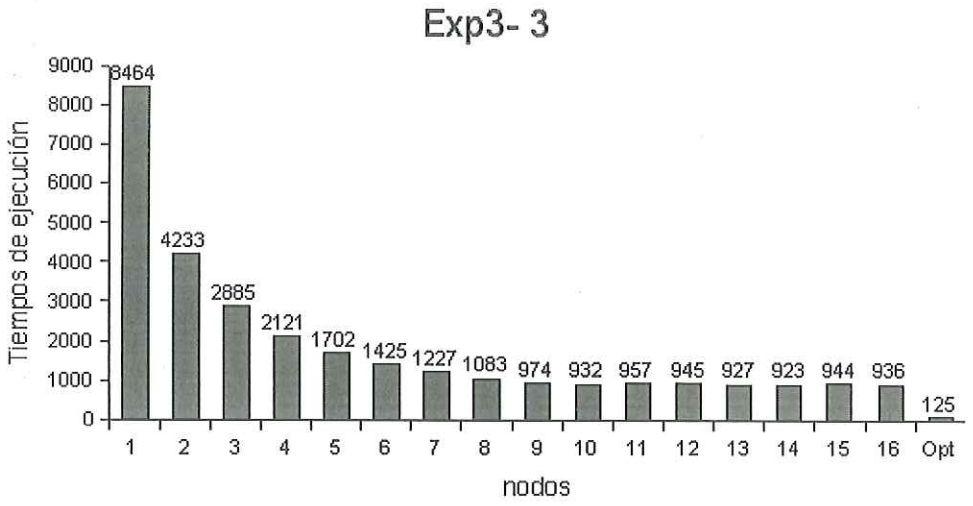

Figura 31: Experimento 3, ejecución de 2000 procesos con tiempos de ejecución y tamaños distintos iniciados en 1 nodo utilizando MOSIX

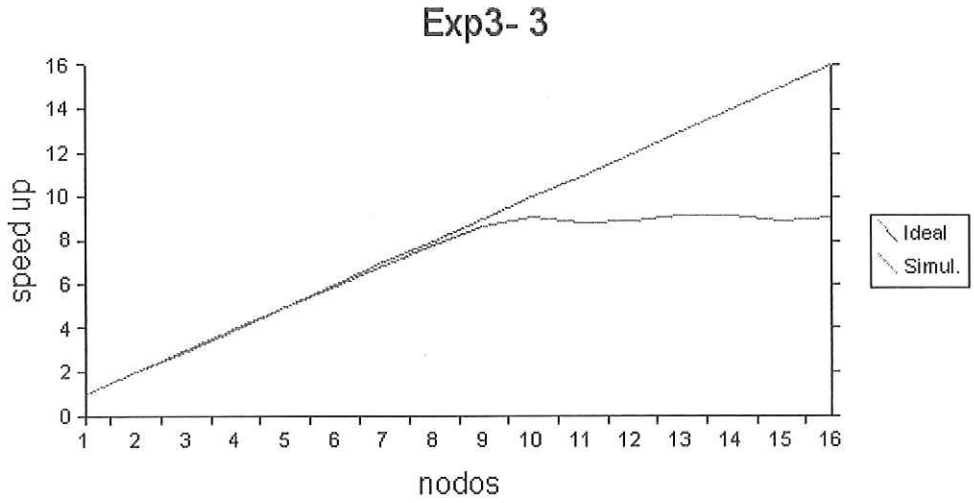

Figura 32: Comparación del speed up ideal y del speed up del experimento 3 con MOSIX

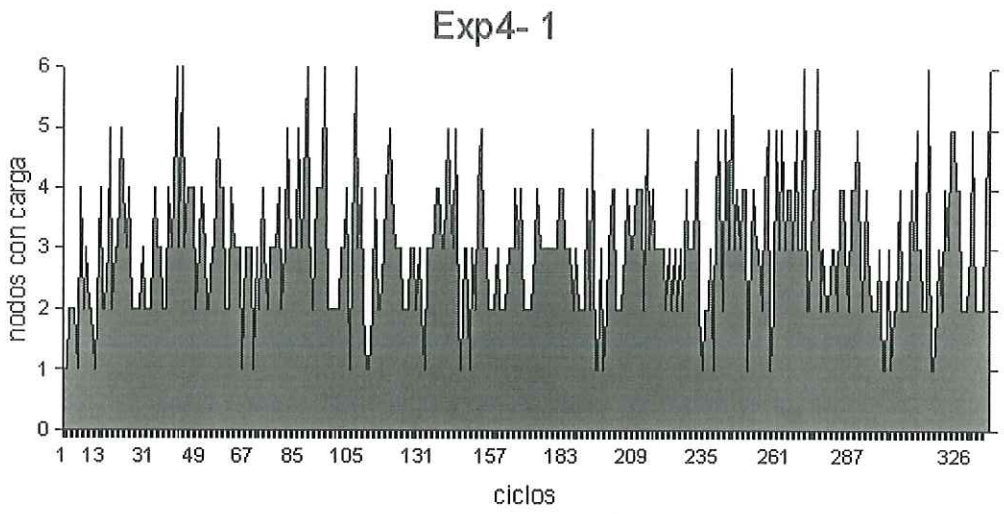

Figura 33: Experimento 4-1, ejecución de 1000 procesos idénticos iniciados en 1 nodo utilizando MOSIX y con ti=1

solo nodo y la distribución es mala como en los experimentos 1 y 2 antes mencionados.

En la figura 34 se muestra el historial de la distribución de la carga al ser iniciada en 8 nodos, éste es el caso en que mejores resultados tuvo el algoritmo de balanceo de MOSIX. Si vemos al rededor del ciclo 12 todos los nodos tienen al menos una unidad de carga y así transcurre la ejecución hasta el ciclo 60 en donde la carga empieza a disminuir y con ella el número de nodos con carga.

En la figura 35 se observa un comportamiento similar al obtenido en RoC-LB cuando ejecutamos la carga de trabajo en 16 nodos. Este resultado significa que el tiempo extra debido a la operación de balanceo es mayor en MOSIX que en RoC-LB.

Para los experimentos 5-1, 5-2 y 5-3 se muestran las gráficas 36, 37 y 38 en donde los resultados son muy similares a los obtenidos en los experimentos 4-1, 4-2 y 4-3 respectivamente.

Finalmente, al analizar los resultados nos dimos cuenta que entre más grande sea

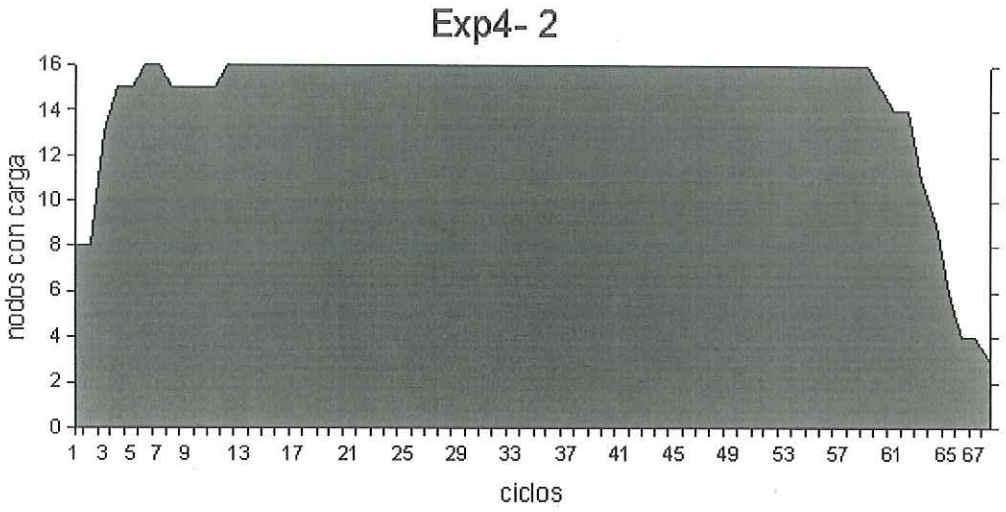

Figura 34: Experimento 4-2, ejecución de 1000 procesos idénticos iniciados en 8 nodos utilizando MOSIX y con ti=1

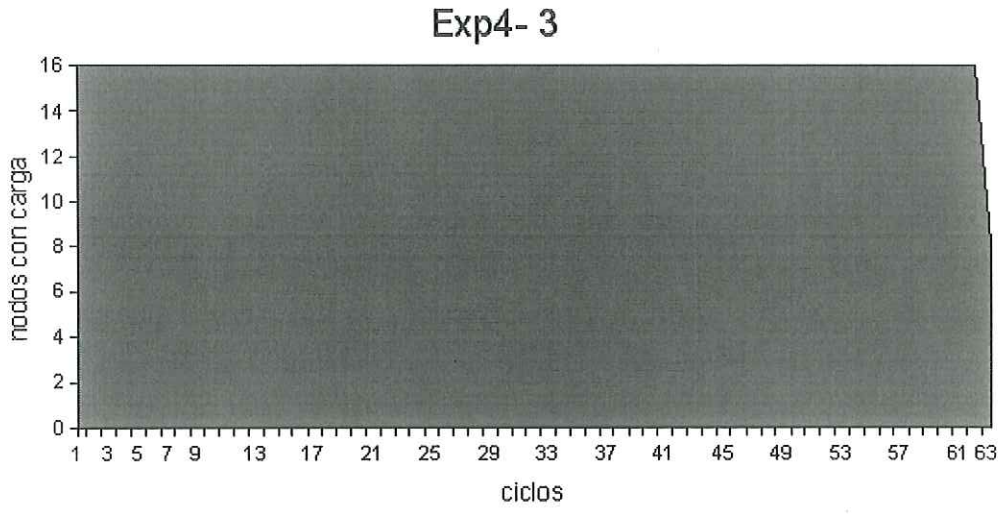

Figura 35: Experimento 4-3, ejecución de 1000 procesos idénticos iniciados en 16 nodos utilizando MOSIX y con ti=1

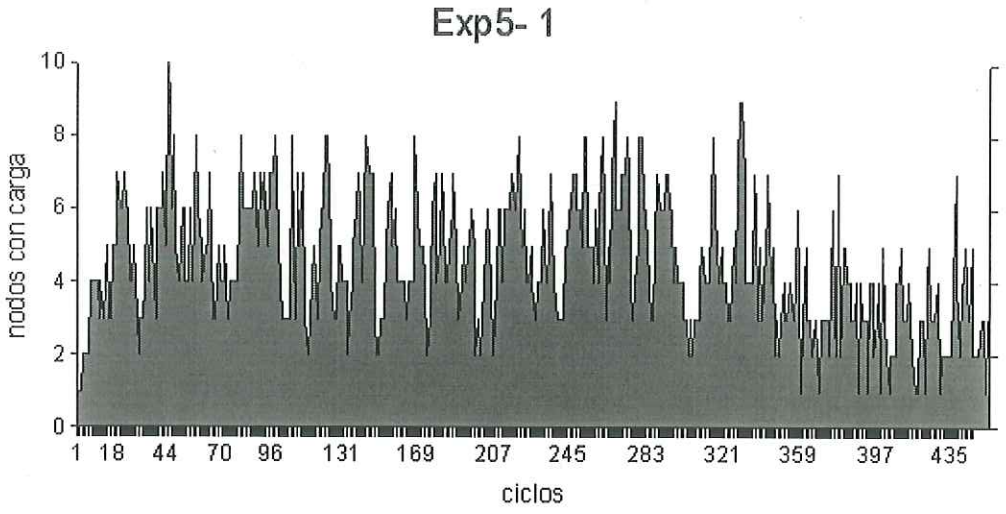

Figura 36: Experimento 5-1, ejecución de 1000 procesos idénticos iniciados en 1 nodo utilizando MOSIX y con ti=2

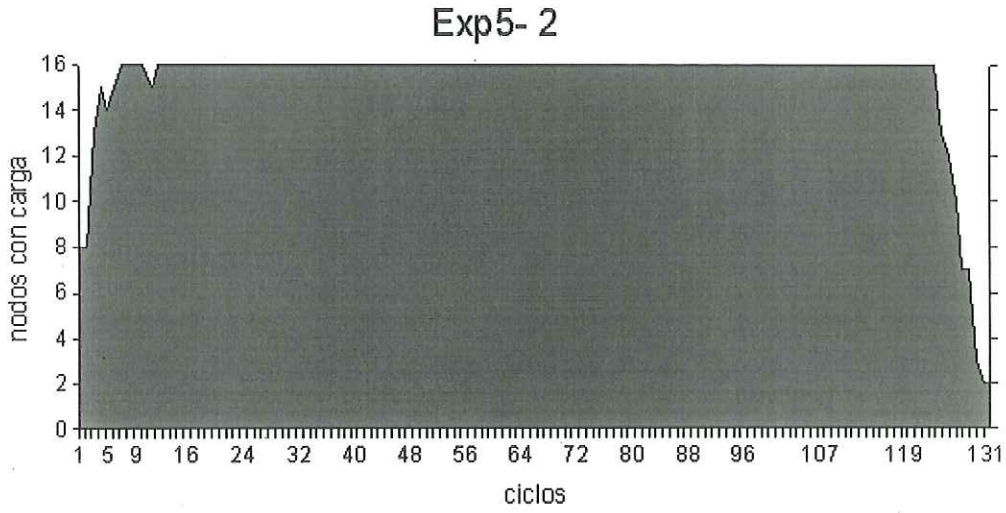

Figura 37: Experimento 5-2, ejecución de 1000 procesos idénticos iniciados en 8 nodos utilizando MOSIX y con ti=2

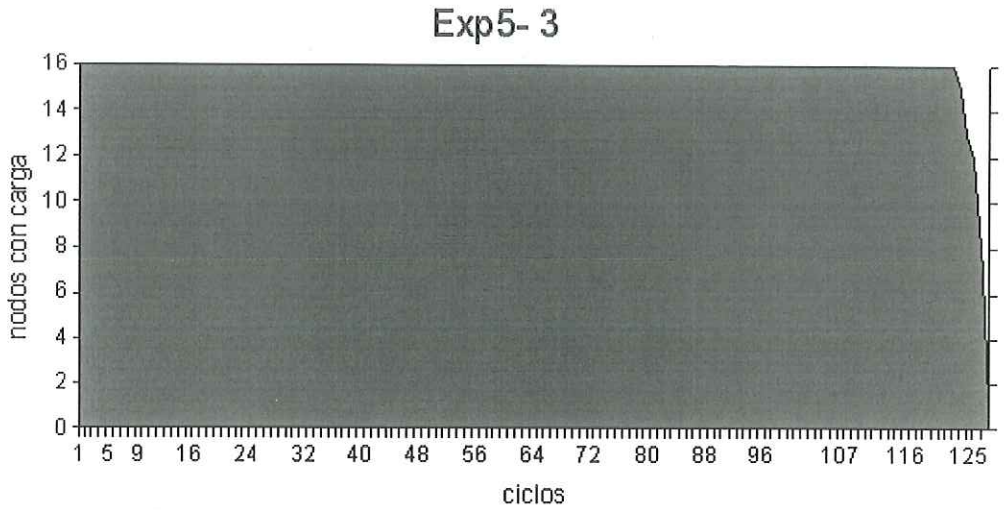

Figura 38: Experimento 5-3, ejecución de 1000 procesos idénticos iniciados en 16 nodos utilizando MOSIX y con ti=2

el tiempo de ejecución individual de los procesos en la carga de trabajo, mejor era el funcionamiento del algoritmo de balanceo de MOSIX por lo que decidimos hacer un experimento para verificarlo. El experimento 6 y sus resultados se muestran en la tabla VI.

También nos dimos cuenta en nuestras simulaciones que ejecutar una carga de trabajo de 2000 procesos con un tiempo de ejecución de 1 unidad era igual que ejecutar 1000 procesos con 2 unidades de tiempo en un solo nodo. Por lo que decidimos que el experimento 6 fuera equivalente al experimento 1 en donde tenemos 2000 procesos con una unidad de tiempo de ejecución. También establecimos las mismas condiciones en cuanto a tiempos de migración, de envío de mensajes, etc. En el nuevo experimento la carga de trabajo es de 40 procesos que tienen como tiempo de ejecución individual 50 unidades. Al ejecutar esta carga en un solo nodo verificamos que el tiempo de ejecución fue de 2000 unidades. Al ejecutarlo en 16 nodos obtuvimos los resultados mostrados al final de la tabla VI, en donde se comprueba nuestra hipótesis de que para procesos con tiempo de ejecución pequeña, nuestra simulación del algoritmo de balanceo de la

Tabla VI: Resultados del Experimento 6 con MOSIX

Experimento	6
No. procesos	40
ti	50
Tamaño	1k
Nodos	16
Nodos con carga	1
ND	1
tmig	no
costomig	no
SpdNs	no
bal	no
TnoLB	2000
Topt	125
Tbal	189
NP	0.965

carga de MOSIX es poco eficiente. Para el experimento 1 obtuvimos como resultado de la ejecución de la carga 670 unidades y NP=0.709, pero para el experimento 6 los resultados fueron 189 unidades de tiempo de ejecución y NP=0.965.

También graficamos un historial de la distribución de la carga para el experimento 6 y en la figura 39 vemos lo que hemos mencionado. Aunque no se hace un uso eficiente de los 16 nodos podemos ver que es mucho mejor que en el experimento 1.

VII.5 Propuesta para la construcción de un SOD

Para concluir este capítulo presentamos las ideas y comentarios que resultaron de la investigación, estudio y práctica de los algoritmos de balanceo de la carga, así como de sistemas operativos distribuidos basados en Linux (Rusling, 1999).

Como se mencionó en la sección V.1 es necesario que un cluster se pueda usar como un sistema multiprocesador, en donde el encargado del balanceo de la carga de trabajo, de la migración de procesos y de la asignación de recursos, entre otras cosas, sea el sistema operativo. Esto permitirá desarrollar mejores aplicaciones y proporcionar a los

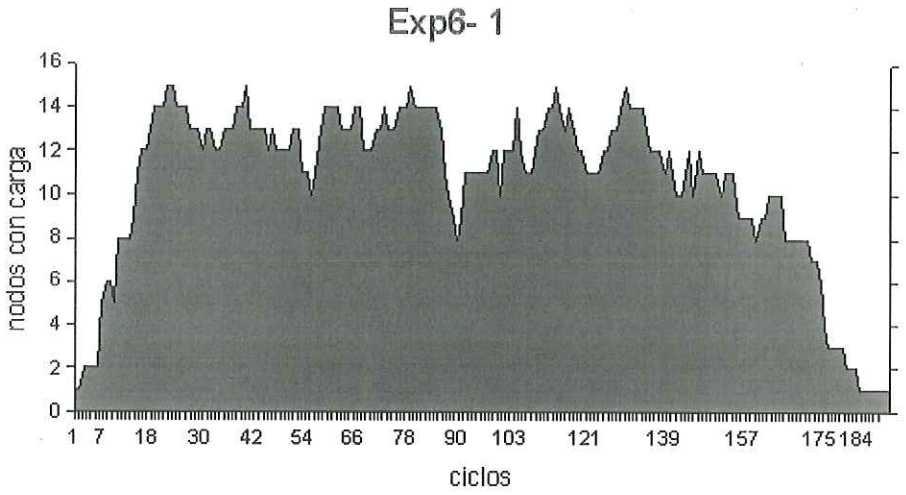

Figura 39: Experimento 6, ejecución de de 40 procesos idénticos iniciados en 1 nodo, utilizando MOSIX y con ti=50

usuarios la "Imagen de un Solo Sistema" (Petri et al, 1998).

También se habló de PVM y MPI como un intento para lograr la transparencia en la migración y ejecución de procesos, pero que tenían la desventaja de que eran sistemas que se ejecutan a nivel de usuario, que se necesita tener conocimientos de la sintáxis de las funciones para el paso de mensajes entre procesos y tener conocimiento de la cantidad de nodos disponibles. Por si esto fuera poco, hay que agregar uno por uno los nodos a la máquina virtual y conectarlos en una topología específica si la aplicación así lo requiere. La distribución de los procesos en los procesadores se hace de manera estática, lo que implica un uso poco eficiente de los recursos del sistema.

Un mejor intento para lograr la "Imagen de un Solo Sistema" sería agregar nuevas funcionalidades al núcleo de un sistema operativo, en particular puede ser un sistema del tipo Linux (Bowman, 1998; Ward, 1999), que es libre y que se puede obtener el código fuente para hacerle las modificaciones necesarias. Lo anterior significa, que es posible modificar el núcleo de Linux para que en ese nivel se lleven a cabo las operaciones de balanceo y migración de procesos (Card et al, 1997). En la figura 40 se

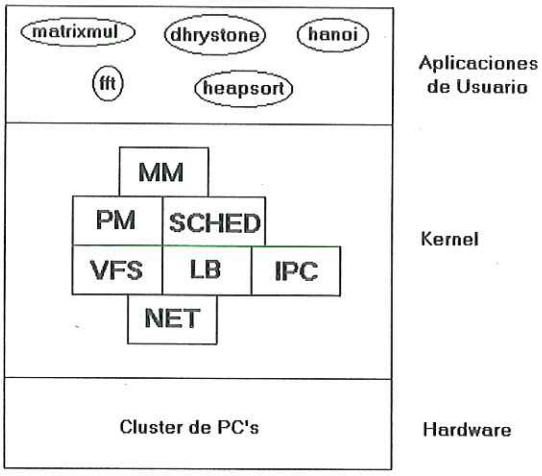

Figura 40: La imagen de un solo sistema de un cluster de computadoras y con un sistema operativo distribuido

presenta la situación anterior. En esta figura se muestran dos subsistemas nuevos en el núcleo, estos nuevos subsistemas son:

Balanceo de la Carga (LB).- Este subsistema sería el encargado de balancear la carga de trabajo utilizando un algoritmo específico. En MOSIX utilizan el algoritmo descrito en la sección V.4 o también podría utilizarse RoC-LB.

Migración de Procesos (PM) .- Este es un subsistema forzosamente necesario cuando se habla de balanceo de la carga, pues debe exisitir un mecanismo que permita interrumpir y enviar procesos a otro nodo para que continúen su ejecución.

En la figura 41 se muestra la relación que debería existir entre los subsistemas existentes (Bowman et al, 1998) y los subsistemas propuestos. Dichas relaciones son:

- El subsistema NET está relacionado con el subsistema PM porque cuando se quiere migrar un proceso o cuando se recibe un proceso de otro nodo, es obvio que se tiene que solicitar un servicio de red para llevar a cabo dicha actividad.

- El subsistema MM está relacionado con el subsistema PM ya que cuando un proceso tiene que ser migrado tiene que liberar memoria o en el caso de que un proceso ha arribado al nodo, entonces se le tiene que reservar y asignar memoria.

- También, el subsistema PM está relacionado con el subsistema SCHED porque cuando un proceso que se está ejecutando tiene que ser migrado, primero tiene que ser suspendido por el planificador de procesos y después enviado a otro nodo.

- Por su parte, el subsistema LB está relacionado con el subsistema NET ya que para enviar solicitudes de carga a otros nodos o para difundir el estado actual del nodo, se tiene que hacer por medio de la red de intercomunicación del cluster, además de que también es posible recibir mensajes de otros nodos solicitando carga.

- Cuando un proceso es recibido y se va a ejecutar en un nodo, el subsistema LB tiene que garantizar que el proceso recién llegado pueda tener acceso al procesador y esto se hace por medio del subsistema SCHED.

- Finalmente, los subsistemas LB y PM están relacionados porque cuando es necesaria una operación de balanceo de la carga inmediatamente se requiere migrar o traer procesos de otro nodo y de esto se encarga el subsistema PM.

Con todo lo anterior, podemos decir que para construir un Sistema Operativo Distribuido es necesario construir cada uno de los subsistemas antes mencionados, lo cual no es un trabajo sencillo y si es extenso. Pero si en lugar de construir todos los subsistemas retomamos la idea de MOSIX, que es ampliar las funcionalidades de un sistema operativo existente, podríamos tener una segunda opción, sólo que en este caso el sistema tomado como base sería MOSIX que ya es un SOD. En la siguiente sección detallaremos más ésta idea.

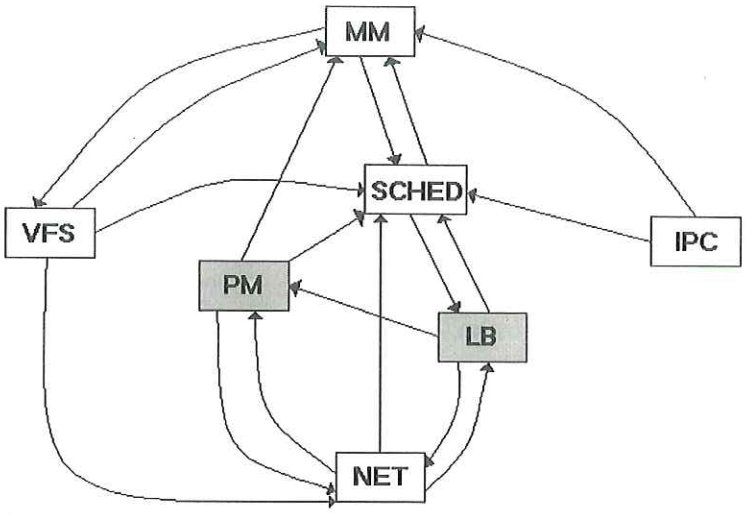

Figura 41: Relaciones entre los subsistemas originales y los subsistemas propuestos

VII.6 MOSIX-RoC

Como se dijo en el capítulo V, MOSIX es una extensión del núcleo de Linux en el cual está contemplada la migración de procesos y el BCT. En otras palabras, con MOSIX tenemos parte de lo que necesitamos. El trabajo ahora consiste en intercambiar el algoritmo de balanceo que tiene MOSIX por el algoritmo RoC-LB, esto se muestra en la figura 42. En esta figura mostramos al núcleo de MOSIX como un sistema formado por varios subsistemas y el subsistema LB es cambiado por el subsistema que llamamos RoC-LB.

Lo anterior no quiere decir que el algoritmo de balanceo usado en MOSIX no funcione o sea deficiente, sino más bien, lo que planteamos es que sería interesante y provechoso llevar a la realidad el algoritmo RoC-LB para comprobar su eficiencia y que tomar a MOSIX como base sería una forma de hacer esto.

Además de la funcionalidad que MOSIX ofrece para nuestros propósitos, se puede observar que el intercambio de algoritmos tiene sentido ya que los dos algoritmos tienen algunas características semejantes, esto es, los dos son distribuidos, dinámicos, con

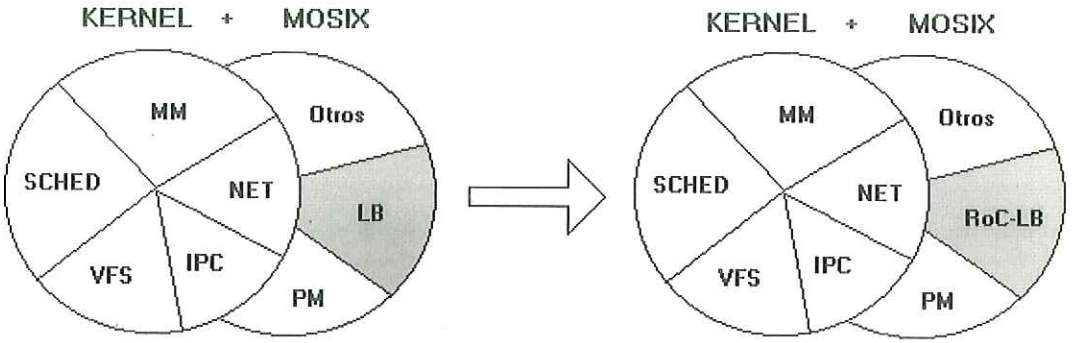

Figura 42: Intercambio de subsistemas de balanceo en MOSIX

Tabla VII: Comparación de los algoritmos de Balanceo de la Carga de Trabajo.

Punto de comparación	**BCT en MOSIX**	**RoC-LB**
Tipo	Tradicional	Innovador
Iniciador	Procesador con carga	Procesador sin carga
Dinámico	Si	Si
Distribuido	Si	Si
A petición	No	Si
Interrupción de procesos	Si	Si
Migración de procesos	Si	Si
Cálculo de la carga	Cada tiempo T	Cada 'SI'
Difusión de la Información	Después de medir la carga	En una solicitud o mensaje
Num. de procesos migrados	1	Carga - HT
Procesos a migrar	El más indicado	Mayor tiempo de CPU
Consi. para migración	Si	No

interrupción de procesos y otras características más que se mencionaron en las secciones V.2.2 y IV.2. También podemos decir que los dos algoritmos de balanceo tienen el mismo objetivo que es balancear la carga de trabajo en un sistema distribuido aunque desde enfoques distintos y opuestos, pues el algoritmo de balanceo de MOSIX sigue los métodos tradicionales y RoC-LB rompe con dichos métodos.

En la tabla VII se muestran algunas características de los dos algoritmos de balanceo, así como las actividades necesarias para tener balanceada la carga de trabajo en un cluster de computadoras. Con estas actividades y características podemos comparar

los dos algoritmos de balanceo desde otro punto de vista.

El tipo de algoritmo es una de las principales diferencias entre los dos algoritmos, pues RoC-LB no sigue los esquemas tradicionales de balanceo y el algoritmo de MOSIX sí las sigue.

En cuanto a cuál nodo es el que inicia las operaciones de balanceo durante la ejecución de una carga de trabajo, podemos decir que en RoC-LB un nodo que predice que se quedará sin carga o que, por alguna razón, ya no tiene carga es el encargado de iniciar una solicitud de carga, mientras que en MOSIX el encargado de iniciar una operación de balanceo es un nodo que encuentra que existe un nodo con menos carga que él.

Verificando las secciones V.2.2 y IV.2 podemos encontrar porque los algoritmos cumplen o no cumplen con las características de ser dinámico, distribuido, a petición, con interrupción de procesos y con migración de procesos.

El cálculo de la carga de trabajo es una actividad que se realiza periódicamente en MOSIX pues se realiza cada tiempo T, es decir cada segundo. Por el lado de RoC-LB, esta actividad se realiza cada intervalo de muestra (SI), no puede decirse que es periódica la medición porque la duración de este intervalo cambia dependiendo de ciertas condiciones durante la ejecución.

La difusión de la carga es una actividad en la cual RoC-LB tiene ventaja sobre el algoritmo de MOSIX. En el primer algoritmo se utilizan los mensajes de solicitud y envío de carga para difundir la información y por lo tanto el tiempo extra que se agrega al tiempo final de la ejecución de la carga es mínimo, mientras que en MOSIX después de que se ha medido la carga en un nodo, éste tiene que darla a conocer a algún subconjunto de nodos existentes, lo que implica que periódicamente se tiene que realizar una distribución y actualización de la carga, lo cual se tiene que ver reflejado en el tiempo final de ejecución.

La cantidad de procesos migrados en cada operación también es un factor muy importante. En MOSIX se migra un proceso a la vez y en RoC-LB se pueden migrar varios procesos al mismo tiempo. Esto último también puede verse como una ventaja para RoC-LB porque si tenemos dos nodos A y B, el nodo A tiene 100 procesos más que el nodo B y los algoritmos de balanceo determinan que hay que migrar 50 procesos al nodo B para tener balanceada la carga. Con RoC-LB es posible detectar y resolver esta situación en un solo paso, mientras que en MOSIX se hará la migración proceso por proceso no sin antes haber medido la carga, haber difundido información, haber seleccionado al proceso adecuado y realizado la operación de migración. Esto último muestra que es más barato, en cuanto a tiempo, establecer una sóla comunicación para migrar varios procesos entre dos nodos, que establecer una comunicación varias veces para migrar un solo proceso.

En cuanto a los procesos seleccionados para migrar, en MOSIX se toman en cuenta factores como la comunicación excesiva de un proceso con otro proceso que se encuentra en otro nodo, el tamaño del proceso, la cantidad de memoria libre en el nodo actual, si el proceso está creando muchos procesos hijos, etc. Mientras que en RoC-LB simplemente se seleccionan a los procesos que tienen mayor tiempo de CPU. Esto último es una ventaja que tiene el algoritmo de balanceo de MOSIX sobre RoC-LB pues con esto se asegura que los procesos que realmente necesitan migrarse lo hagan y no tengan que andar viajando por la red, pero esto mismo tiene la desventaja de que es costoso, en cuanto a tiempo, seleccionar cuidadosamente el proceso que se va a migrar porque esto se hace cada operación de balanceo.

Otra ventaja que tiene MOSIX sobre RoC-LB es que es un sistema real, en el cual se han encontrado y tomado en cuenta factores reales, como cantidad de memoria disponible en un nodo, el espacio libre en la tabla de procesos, la pérdida o unión inesperada de algún nodo en el cluster, etc., que son factores que en la teoría se pueden

tomar como triviales u obvios, pero que en realidad influyen sobre el resultado final. Aunque esto último no quiere decir que esos factores no se puedan tomar en cuenta en RoC-LB, sino que por el momento no fueron considerados y que tomarlos en cuenta sería un avance o mejora a dicho algoritmo.

Después de haber estudiado, probado y comparado los dos algoritmos de balanceo podemos plantear otra alternativa de balanceo de la carga, es decir, en lugar de intercambiar el algoritmo de balanceo de MOSIX por RoC-LB como se planteó antes, se puede construir un nuevo algoritmo de balanceo el cual sea una combinación de los dos algoritmos mencionados.

Esta nueva propuesta implica una comparación y evaluación minuiciosa de cada una de las actividades involucradas en el balanceo de la carga para poder decidir cuál es mejor. Pero para comparar el funcionamiento y desempeño de cada actividad necesitamos que los dos algoritmos estén implantados, ejecutar varios programas y analizar resultados. Por el lado de MOSIX ya está hecha la implantación pero por el lado de RoC-LB no lo está. Esto último nos lleva al problema planteado anteriormente, es decir, se necesita una implantación del algoritmo RoC-LB.

Conclusiones y Trabajo Futuro

Conclusiones

Como se planteó al principio de este documento, este trabajo es parte fundamental un proyecto en el cual se quiere construir un SOD. Con este trabajo se dieron los primeros pasos en cuanto a sentar las bases y crear la infraestructura que permita ir avanzando paso a paso hasta alcanzar los objetivos generales del proyecto.

En los objetivos de nuestro trabajo se perseguían tres aspectos: técnicos (construcción de un cluster), de investigación (estudio de algoritmos de BCT) y prácticos (simular dos algoritmos). Nuestras conclusiones están agrupadas de acuerdo a los tres aspectos anteriores y plantean el nivel de satisfacción de nuestros objetivos.

1. Se construyó un cluster con PC's (COPS) el cual se mostró en la sección VII.2 y de la misma forma que se plantea en (Sterling et al, 1999). El costo de este sistema se resume al costo del HUB que se compró debido a ser el único elemento faltante. De esta forma construimos un sistema distribuido en el cual se pueden ejecutar aplicaciones paralelas con PVM y MPI. Pero como es un sistema multiusuario en el cual cada uno de los usuarios puede estar ejecutando varios procesos a la vez, al usar MOSIX es posible balancear la carga, automática y transparentemente para el usuario y así incrementar el rendimiento del cluster.

   Aunque el número de nodos que tenemos en COPS no ayudó mucho a dar claridad a los resultados de la ejecución de *Metabenchmark* si fue suficiente para darnos

una idea de como cambia el rendimiento con el número de nodos en el sistema.

Un punto importante es que construimos un sistema de rendimiento regular a un costo pequeño y que ese rendimiento puede ir creciendo conforme aumentemos el número de nodos en el cluster o si agregamos computadoras más poderosas que las que teníamos. Lo anterior se resume a un concepto y característica que es muy bueno tener en sistemas paralelos y distribuidos: "Escalabilidad".

2. Al estudiar los dos algoritmos de balanceo nos dimos cuenta que en MOSIX la operación de balanceo de la carga depende tanto del cambio de la carga de los nodos como de las características de los procesos. En *RoC-LB* sólo se toma en cuenta el primer punto. Lo anterior no quiere decir que no se puedan tomar en cuenta las características de la carga para cuestiones de balanceo, sino que no fueron tomadas en cuenta en *RoC-LB* y que tomarlas en cuenta sería un punto que ayudaría a mejorar el funcionamiento de *RoC-LB*.

   Como la construcción de un cluster de computadoras personales permite tener computadoras con distintos tipos de procesadores y cantidades de memoria RAM, es posible que estas diferencias influyan sobre los resultados finales de la ejecución de una carga de trabajo. En MOSIX se toman en cuenta las diferencias entre las velocidades de los procesadores lo cual hace que se les asigne menos trabajo a los procesadores más lentos. En *RoC-LB*, la razón de cambio de la carga puede evitar que le lleguen muchos procesos a un procesador lento. Lo que no puede detectar *RoC-LB* son situaciones en donde un proceso grande comienza a hacer un intercambio de páginas excesivo porque se encuentra en un nodo con poca memoria. Debido a lo anterior no es posible migrar dicho proceso a otro procesador que tenga más memoria libre disponible, lo cual resuelve el problema del intercambio de páginas excesivo.

Otro punto determinante en los resultados obtenidos fue la difusión de la información y la migración de los procesos en los dos algoritmos. En MOSIX se realizan muchas operaciones en cuanto a difusión dado que en cada tiempo T, cada procesador puede mandar 2 o 3 mensajes hacia los demás y una vez recibido el mensaje se genera otro mensaje para verificar que el primero llegó a su destino. Esto mismo lo pueden hacer todos los procesadores al mismo tiempo y por lo tanto el número de mensajes en la red, en determinado momento, se vuelve un factor a considerar. Además, el número de procesos migrados de un nodo a otro es uno por cada mensaje enviado del segundo al primer nodo y como vimos en los experimentos esto es poco eficiente para procesos de duración corta. Por otro lado, la difusión de la información en *RoC-LB* se hace utilizando el mismo mensaje en donde se migran procesos, además de que es posible migrar más de un proceso hacia otro nodo en la misma operación. Con estas dos situaciones se reduce tanto el número de mensajes para difusión de la información como el número de mensajes para migrar procesos.

3. Simular el algoritmo de balanceo de MOSIX fue complicado debido a que el funcionamiento del algoritmo depende de muchos factores reales como la "Regla de residencia" para cada proceso, la cantidad de comunicación de un proceso con procesos que están en otro nodo, el uso de memoria compartida entre procesos, el estado del proceso, el tiempo que pasa el procesador haciendo intercambio de páginas entre la memoria y el disco, etc. Estos factores no son considerados en *RoC-LB* y por lo tanto no fueron considerados en la simulación y tomarlos en cuenta para la simulación habría hecho que ésta fuera más complicada.

   Comparando los números obtenidos en nuestros experimentos, podemos decir que *RoC-LB* fue mejor en la mayoría de los escenarios planteados, excepto en el experimento 4-2 en que MOSIX fue mejor.

Con el experimento 6-1 vimos que el algoritmo de balanceo de MOSIX es poco eficiente para procesos de corta duración, debido a que la migración tomaría má tiempo que la ejecución.

Al final no podemos decir que *RoC-LB* es mejor o peor que el algoritmo de balanceo de la carga de MOSIX, debido a que encontramos que uno realiza ciertas actividades mejor que el otro y nuestros experimentos no fueron diseñados para hacer un análisis comparativo de todas las actividades. Para llevar a cabo estas comparaciones es necesario realizar experimentos específicos que permitan medir y comparar la eficiencia de cada actividad.

Trabajo Futuro

Como actividades o trabajo a futuro se plantean los siguientes puntos, los cuales tendrían el objetivo de mejorar la simulación hecha y continuar con los objetivos generales del trabajo.

- Estudiar el núcleo de Linux y hacer modificaciones que nos permitan en un futuro poder adaptar o cambiar subsistemas del mismo núcleo.
- Estudiar y plantear teóricamente la fusión de los algoritmos de balanceo de MOSIX y *RoC-LB*.
- Dado que la simulación realizada fue para comparar dos algoritmos y no monitorear la ejecución, consideramos que sería interesante tener un sistema gráfico que permita observar y monitorear la distribución de la carga durante la ejecución.
- Agregar al trabajo actual un sistema gráfico que permita monitorear el balanceo

y distribución de la carga de trabajo, permitiría comparar distintos algoritmos, analizarlos y plantear nuevas alternativas de balanceo.

- Paralelizar aplicaciones que requieran poder de cómputo y que sean ejecutadas en COPS.

Abreviaturas Usadas

CT- Límite Crítico (Critical Threshold).

DL- Diferencia de la Carga (Difference in Load).

EP- Elemento de Procesamiento.

GFLOPS- Miles de Millones de Instrucciones de Punto Flotante Por Segundo.

GIPS- Miles de Millones de Instrucciones Por Segundo.

HT- Límite Superior (High Threshold).

ISS- Imagen de un Solo Sistema.

LAN- Red de Área Local (Local Area Network).

LSP_PE- Estado de Carga de Elemento de Procesamiento Anterior (Load State of Preceding Processing Element).

LT- Límite Inferior (Low Threshold).

MIPS- Millones de Instrucciones Por Segundo.

MOSIX- Sistema Operativo de Multicomputadoras del Tipo Unix (Multicomputer Operating System for UNIX).

MPI- Interfaz de Paso de Mensajes (Message Passing Interface).

ND- Retardo de Red (Network Delay).

NF- Número de Reenvíos (Number of Forwards).

NFS- Sistema de Archivos de Red (Network File System).

NLUR- Número de Unidades de Carga Solicitadas (Number of Load Units Requested).

NP- Rendimiento Normalizado (Normalized Performance).

PVM- Máquina Virtual Paralela (Parallel Virtual Machine).

PL- Carga Predicha (Predicted Load).

PPM- Migración de Procesos Interrumpibles (Preemptive Process Migration).

RoC-LB- Balanceo de la Carga de Trabajo basado en la Razón de Cambio (Rate of Change Load Balancing).

SI- Intervalo de Muestra (Sampling Interval).

SOD- Sistema Operativo Distribuido.

SPI- Identificador del Procesador que solicita carga (Sink Processor Identifier).

SMP- Multiprocesador Simétrico (Symmetric MultiProcessing).

TCP/IP- Protocolo de Control de Transmisión sobre el Protocolo Internet (Transmission Control Protocol over Internet Protocol).

TERAFLOPS- Millones de Millones de Instrucciones de Punto Flotante Por Segundo.

UHN- Único Nodo Origen (Unique Home-Node).

Literatura Citada

Baker Mark A., Fox Geoffrey C. y Yau Hon W. 1996. *A Review of Commercial and Research Cluster Management Software.* Northeast Parallel Architecture Center. 111 College Place, Syracuse University New York 13244-4100 USA.

Barak Amnon, Guday Shai y Wheeler Richard G. 1993. *The MOSIX Distributed Operating System.* Springer-Verlag.

Barak Amnon, La'adan Oren y Shiloh Amnon. 1999. *Scalable Cluster Computing with MOSIX for LINUX.* `http://www.mosix.cs.huji.ac.il`. Jerusalem 91904, Israel.

Bowman Ivan. 1998. *Conceptual Architecture of the Linux Kernel.* `http://www.grad.math.uwaterloo.ca/~itbowman/CS756G/a1/`. `ibowman@sybase.com`.

Bowman Ivan, Siddiqi Saheem y Meyer Tanuan. 1998. *Concrete Architecture of the Linux Kernel.* `http://plg.uwaterloo.ca/~itbowman/CS746G/a2`. Department of Computer Science, University of Waterloo. Waterloo, Ontario N2L 3G1.

Campos Luis Miguel y Scherson Isaac. 1998. *Rate of Change Load Balancing in Distributed and Parallel Systems.* Dept. of Information and Computer Science. University of California-Irvine. Irvine, CA 92697, USA.

Card Remy, Dumas Eric y Mevel Franck. 1997. *The Linux Kernel Book.* John Wiley & Sons Ltd. Baffins Lane, Chichester, West Sussex PO19 1UD, England.

Dongarra Jack. 1999. *Performance of Various Computers Using Standard Linear Equations Software.* Computer Science Department. University of Tennessee. Knoxville, TN 37996-1301 and Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831. `http://www.netlib.org/benchmark/performance.ps`.

Geist Al, Beguelin Adam, Dongarra Jack, Jiang Weicheng, Robert Manchek y Vaidy Sunderam. 1994. *PVM: Parallel Virtual Machine.* The MIT Press.

Krishnaswamy Umesh. 1995. *Computer Evaluation Using Performance Vectors.* PhD. Thesis, University of California Irvine.

Melo Alba Cristina. 1998. *Parallel Systems. A Hardware and Software Overview.* University of Brasilia, Brasil.

Miller Barton, Cargille Jonathan, Bruce Irvin, Kunchithapadam Krishna, Callaghan Mark D., Hollingsworth Jeffrey K., Karavanic Karen L. y Newhall Tia. 1996. *The Paradyn Parallel Performance Measurement Tools.* Computer Sciences Department, University of Wisconsin-Madison. 210 W. Dayton Street. Madison, WI 53706.

MPI Forum. 1995. *MPI: A Message Passing Interface Standard.* `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

OECD Informatics Studies. 1974. *The Evaluation of the Performance of Computer Systems.* Organization for Economic Cooperation and Development. Paris.

Parallel C, User Guide. 1989. C V2.1 Release Note, 3L Ltd.

Petri Stefan, Bolz Matthias y Langendorfer Horst. 1998. *Transparent Migration and Rollback for Unmodified Applications in Workstation Clusters.* Institute of Computer Engineering, Medical University at Lubeck and Institute for Operating Systems and Computer Networks, Braunschweig University of Technology.

Rajkumar, Krishna Mohan y Bindu Gopal. 1999. *PARMON: A Comprehensive Cluster Monitoring System.* School of Computing Science, Queensland University of Technology. 825a, PLAS Lab., S-Block. Gardens Point Campus, Brisban, Australia.

Rusling David A. 1999. *The Linux Kernel.* Version 0.8-3. `http://www.linuxdoc.org/LDP/tlk/tlk.html`, `david.rusling@arm.com`.

Russ Samuel H., Banicescu Ioana, Bhafoor Sheikh, Janapareddi Bharathi, Robinson Jonathan y Lu Rong. 1997. *Hectiling: An Integration of Fine and Coarse-Grained Load-Balancing Strategies.* NSF Engineering Research Center for Computational Field Simulation, Mississippi State University.

Scherson Isaac D. y Campos Luis M. 1998. *A Distributed Dynamic Load Balancing Strategy Based on Rate of Change.* Dept. of Information and Computer Science. University of California-Irvine. Irvine, CA 92697, USA.

Schnor Bettina, Petri Stefan y Langendorfer Horst. 1996. *Load Management for Load Balancing on Heterogeneous Platforms: A Comparison of Traditional and Neural Network Based Approaches.* Institut fur Betriebsssysteme und Rechnerverbund, TU Braunschweig. Bultenweg 74/75. D-38106 Braunschweig, Germany.

Sterling Thomas L., Salmon John, Becker Donald J. y Savarese Daniel F. 1999. *How to Build a Beowulf. A Guide to the Implementation and Application of PC Clusters.* The MIT Press.

Tanenbaum Andrew S. 1993. *Sistemas Operativos Modernos.* Prentice Hall Hispanoamericana.

Ward Brian. 1999. *The Linux Kernel HOWTO.* `http://burks.bton.ac.uk/burks/linux/howto/kernelho.htm`. `bri@cs.uchicago.edu`.

XPVM. 1994. *XPVM: A Graphical Console and Monitor for PVM.* Oak Ridge National Laboratory. P.O. Box 2008. Bldg 6012, MS 6367. Oak Ridge TN 37831-6367. http://www.netlib.org/utk/icl/xpvm/xpvm.html.

Xu Chengzhong y Lau Francis C. M. 1997. *Load Balancing in Parallel Computers. Theory and Practice.* Kluwer Academic Publishers.