CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL I.P.N.

UNIDAD ZACATENCO
DEPARTAMENTO DE INGENIERIA ELECTRICA
SECCION COMPUTACION

Balance Dinámico de Carga en un Sistema Paralelo con Memoria Distribuida

Tesis que presenta

Miguel Alfonso Castro García

Para obtener el Grado de

Maestro en Ciencias

En la especialidad de

Ingeniería Eléctrica

Dra. Graciela Román Alonso
Director

Dr. Adriano de Luca Pennacchia
Codirector

México, D.F., Mayo del 2001

INDICE

RESUMEN

Esta tesis es una contribución dentro del área de los sistemas distribuidos enfocándose en la problemática de la asignación dinámica de carga. Los algoritmos propuestos para resolver este problema consideran un conjunto de procesadores en los cuales tienen que distribuir la carga (procesos), de tal forma que el rendimiento del sistema y el tiempo de ejecución de las aplicaciones mejoren. Uno de los objetivos de estos algoritmos puede ser la asignación equitativa de trabajo, tomando en cuenta criterios de balance de carga.

Para apoyar el estudio y la evaluación del rendimiento de los algoritmos de asignación dinámica de carga, en esta tesis se propone una plataforma de experimentación. Esta plataforma fue desarrollada utilizando la herramienta PVM la cual permitió implantar los elementos de control y de información que rigen el funcionamiento de cinco algoritmos de asignación de carga: algoritmo centralizado, cíclico, distribuido, vecinos directos y algoritmo vectorial.

Las aportaciones principales de la plataforma propuesta, son las siguientes:

- Facilita el estudio del comportamiento de un algoritmo particular de asignación dinámica de procesos, permitiendo el ajuste de sus parámetros.
- Permite obtener una comparación de rendimiento entre varios algoritmos de asignación dinámica.
- Posibilita implantar y evaluar nuevas propuestas de algoritmos de asignación dinámica.
- Permite elegir un algoritmo de asignación que reduzca el tiempo de ejecución de una aplicación en particular.

Los resultados obtenidos se basaron principalmente en la ejecución de un conjunto de programas de prueba de tipo sintético (su objetivo es generar carga sobre el sistema). Los experimentos se efectuaron sobre un cúmulo con 9 procesadores Pentium III-LINUX, interconectados bajo una topología tipo bus.

INTRODUCCION

No es raro escuchar la frase "Dos cabezas piensan mejor que una", a partir de la frase anterior, es fácil pensar que 2 computadoras tendrán una mejor eficiencia o rendimiento que una sola. Por lo que un trabajo ejecutado en dos computadoras se debería ejecutar en la mitad del tiempo.

Pero, ¿qué se puede decir acerca de 3, 10, 100 ó 1000 computadoras?, ¿Tendrán mejor rendimiento que dos?. Esta tesis situada dentro del área de los Sistemas Paralelos y Distribuidos, presenta un estudio acerca de la asignación dinámica de carga.

El problema de distribuir equitativamente procesos y la actividad de comunicación a través de una red de computadoras a fin de que ningún dispositivo esté sobrecargado, puede entenderse como Balance de Carga. El balance de carga generalmente es considerado dentro de un algoritmo de asignación. Cuando la asignación de procesos se realiza en tiempo de ejecución se habla de asignación dinámica de carga, esta es especialmente importante para los sistemas donde es difícil predecir el número de peticiones que serán enviadas a un servidor[1], así como la cantidad de trabajo de varios usuarios.

Para resolver el problema de la distribución se han propuesto ciertos algoritmos de asignación dinámica [8], los cuales en tiempo de ejecución tratan de decidir si cierto proceso o procesos serán ejecutados de forma local o serán transferidos hacia otros procesadores del sistema[21,8,4,7].

Un ejemplo práctico de balance dinámico, es en los Web Servers (Figura 0), en donde si un servidor comienza a saturarse, las peticiones son redireccionadas a otro servidor con más capacidad.

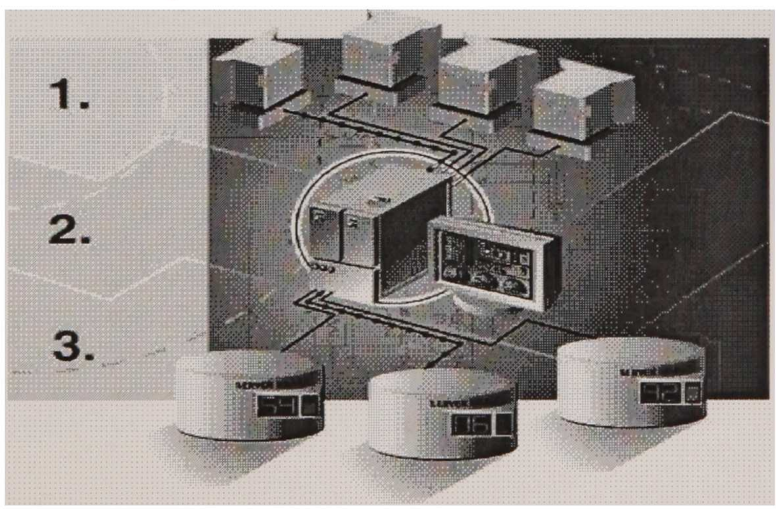


Figura 0. Redireccionamiento de Peticiones en Web Servers

A la fecha varios algoritmos se han propuesto para la asignación de procesos. Gran parte de ellos, han sido más bien de tipo estático, los cuales analizan el uso de los recursos (memoria, cpu) en tiempo de compilación y deciden una asignación de los procesos sobre los procesadores, antes de comenzar la ejecución[15,18,20,17]. Otros estudios de tipo dinámico, se han encaminado a la variación y ajustes de parámetros en las aplicaciones, mediante un análisis de costos de ejecución local de procesos contra costos de comunicación entre procesos[9,6,2].

En esta tesis se presenta la realización de una plataforma para el estudio y comparación de algoritmos de asignación dinámica de procesos, utilizando la herramienta PVM[3]. Después de implantar 5 algoritmos con diferentes características en cuanto la asignación de carga y el manejo de la información, se obtuvo una comparación de sus rendimientos, mediante la ejecución de un conjunto de programas de prueba sintéticos. Los resultados que guiaron nuestra comparación obtenidos para cada programa, básicamente es el tiempo de ejecución, aunque también, es posible obtener el factor de aceleración, el número de procesadores usados y el número de procesos ejecutados por cada procesador.

La Organización de esta Memoria es la siguiente:

En el Capítulo I, se presenta la problemática de la Asignación y Balance Dinámico de Carga. En el Capítulo II, se describen los 5 algoritmos de Asignación Dinámica (Centralizado, Distribuido, Cíclico, Vectorial y Vecinos Directos) estudiados. Los programas sintéticos de prueba son descritos en el Capítulo 3, presentando después los resultados obtenidos. Por último, se presentan las conclusiones y los anexos referentes a la herramienta PVM y algunos programas de prueba.

CAPITULO 1
ASIGNACIÓN Y BALANCE DINÁMICO DE CARGA

Estudios en la actualidad [22,23], han reflejado que en sistemas a los cuales se les ha aplicado una técnica de balance de carga sobre sus procesos, el tiempo total de ejecución del sistema es reducido a la mitad del tiempo que tomaría ejecutarlo sin el balance.

Por mencionar un ejemplo, la simulación de una partícula de una célula en tres dimensiones en donde el tiempo de cómputo para realizar esta simulación es de alrededor de 4 meses sobre una máquina Cray^{tm} TD3 con 256 procesadores, aplicando ciertos algoritmos de balance de carga, el tiempo es reducido a la mitad[4].

La asignación de carga puede ser de dos tipos:

- Asignación Estática
- Asignación Dinámica

La asignación estática de carga se aplica cuando en un sistema se puede estimar la utilización de los recursos (uso de procesador y de comunicaciones), conociendo cuántos procesos se ejecutarán, las relaciones de comunicación y de precedencia. La asignación de los procesos de la aplicación sobre los procesadores, se decide directamente antes de comenzar la ejecución y son ejecutados de inicio a fin en el procesador correspondiente, independientemente de la variación de la carga en el sistema.

Este tipo de asignación estática no siempre es el adecuado, ya que si los procesos se ejecutan en un sistema multiusuario el estado de carga del sistema puede variar durante la ejecución. En este caso no se puede hacer nada hasta que todos los procesos terminen.

Por el contrario, en la asignación y balance dinámico de carga la asignación de procesos sobre los procesadores se decide a tiempo de ejecución, considerando la carga actual del sistema como la utilización de los procesadores o la cantidad de mensajes en las colas de entrada/salida.

Para la realización de cualquier técnica de asignación y balance dinámico, son necesarios dos elementos (Figura 1):

1. Información relacionada al estado de carga del sistema (Elemento de Información).
2. Alguien que conociendo la información de la carga global o parcial asigne más carga a los procesadores menos cargados y por el contrario, evite más carga de procesamiento a los que estén más ocupados (Elemento de control).

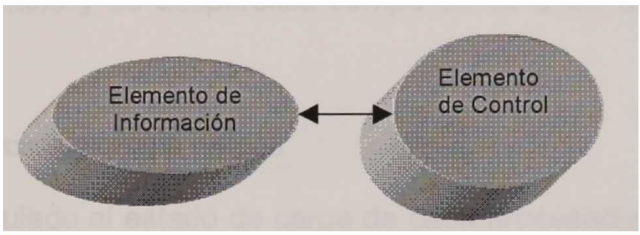


Figura 1. Módulos básicos en la Asignación Dinámica.

En la sección siguiente se presentan más a detalle dichos elementos.

1.1 Elemento de Información

Dos preguntas importantes se plantean respecto a este elemento: ¿cómo se cuantifica el estado de carga de los procesadores? y ¿cómo se estima el estado de carga del sistema?. Para responder a esas preguntas, un proceso (ejecutado sobre cada procesador) se pone en marcha con la tarea de intercambiar información con otros procesadores en el sistema. Para obtener la información sobre el estado de carga del sistema, son necesarias dos cosas, primero cuantificar la carga de cada uno de los procesadores para obtener sus estados de carga locales. Posteriormente, en base a dichos estados, recolectar una información global o parcial del sistema, para que se pueda decidir el lugar más adecuado de asignación de los procesos.

1.1.1 Cuantificación de la carga

Existen diversas formas de poder cuantificar la carga en un procesador o nodo, una aproximación puede ser considerando el número total de procesos sobre un procesador (los cuales pueden estar en ejecución o listos para ser ejecutados), otra mediante el porcentaje de utilización del CPU o tomando la longitud de la cola de E/S.

La primera forma referente a contabilizar el número total de procesos, es la que se utilizó en la construcción de la plataforma propuesta.

Para mantener actualizado un elemento de información, se hace uso del concepto de frecuencia de cuantificación, la cual es un parámetro que indica cada cuántos segundos se debe cuantificar la carga de un procesador para verificar si su estado de carga ha cambiado. Aquí se notan 2 casos, cuando la frecuencia de cuantificación es grande y cuándo es corta.

Si la frecuencia de cuantificación es grande, se tiene el problema de no tener información reciente acerca del estado de carga del procesador, por el contrario, si esta frecuencia es muy corta, pudiera darse el caso de que no existiera ningún cambio y se desperdicie tiempo de CPU en estar verificando la carga actual.

1.1.2 Información global o parcial

Una vez calculado el estado de carga de cada procesador, la gran parte de los algoritmos de asignación dinámica consideran un mecanismo de recolección de dichos estados. Con esto, los procesadores pueden obtener una información del estado de carga del sistema (Figura 2).

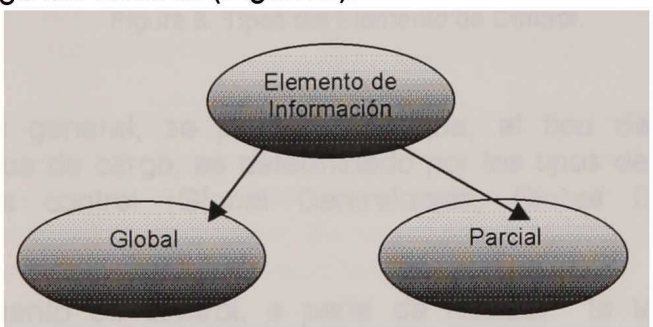


Figura 2. Tipos del Elemento de Información.

Cuando la información contiene los estados de carga de todos los procesadores en el sistema, se dice que el elemento de información es de tipo **Global**. En otro caso, cuando la información comprende los estados de carga de sólo algunos procesadores, por ejemplo, vecinos, se dice que el elemento de información es de tipo **Parcial.**

1.2 Elemento de Control

El elemento de control de un algoritmo de asignación dinámica, es la parte que se encarga de responder las siguientes preguntas: ¿cuándo es necesaria la transferencia de un proceso hacia otro procesador? y ¿hacia qué procesador transferir dicho proceso?. Generalmente el elemento de control usa la información

recolectada por el elemento de información (global o parcial) para detectar los nodos descargados y así decidir el sitio donde se asignará un proceso.

El elemento de control puede ser de dos tipos (Figura 3). Cuando un único procesador en el sistema decide cuándo y a dónde transferir los procesos, se dice que el elemento de control es de tipo **Centralizado**. Generalmente, en ese caso el procesador central considera un elemento de información de tipo Global y todos los demás procesadores tienen que consultar al procesador central para saber el lugar de ejecución de cada proceso. En otro caso, si más de un procesador es capaz de tomar sus propias decisiones respecto a la transferencia, se dice que el elemento de control es de tipo **Distribuido**, pudiendo considerar un elemento de información de tipo Global o Parcial.

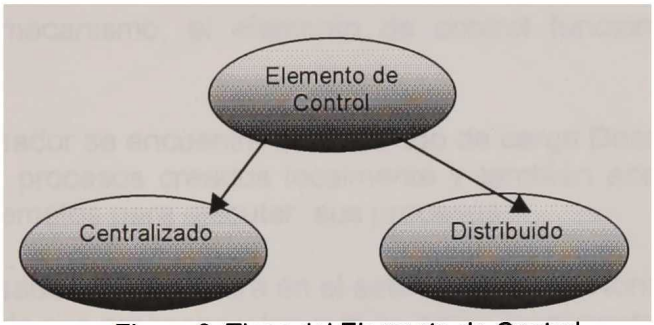


Figura 3. Tipos del Elemento de Control.

De manera general, se puede decir que, el tipo de un algoritmo de asignación dinámica de carga, es determinado por los tipos de sus elementos de información y de control (Global Centralizado, Global Distribuido, Parcial Distribuido).

En el elemento de control, a parte de realizar la tarea de decidir la transferencia de carga de los procesadores sobrecargados hacia los menos cargados, puede buscar también balancear la carga sobre los procesadores del sistema. En este caso, todos los procesadores trabajarían en igual proporción, buscando la mejora en el rendimiento del sistema. Una solución para balancear la carga es el uso del mecanismo de doble frontera:

En este mecanismo se considera el concepto de tres estados de carga. En la Figura 4, un procesador se encuentra inicialmente en un estado *Descargado*, pero si su carga aumenta rebasando cierto límite (Frontera1), su carga pasa al siguiente estado al cual llamaremos de *carga normal*. De igual forma, si el número de procesos sigue incrementándose y el estado de carga rebasa la Frontera2 se entra al estado de carga *Sobrecargado,* esto se observa en la Figura 4.

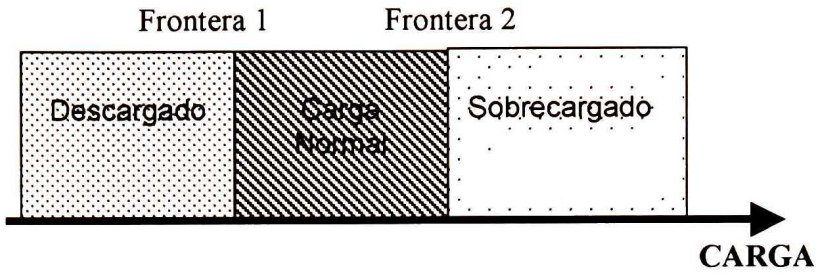


Figura 4. Fronteras de Carga de un Procesador.

Con este mecanismo, el elemento de control funciona de la siguiente manera:

- Si un procesador se encuentra en el estado de carga Descargado:
  Puede ejecutar procesos creados localmente y también aceptar peticiones de procesadores remotos para ejecutar sus procesos.

- Si un procesador se encuentra en el estado de carga Normal:
  Sigue ejecutando sus procesos y los que se creen localmente, pero no aceptará peticiones de procesadores remotos para ejecutar procesos.

- Si un procesador se encuentra en el estado de carga Sobrecargado:
  Si se requiere crear un nuevo proceso, se busca a otro procesador en el sistema que se encuentre en estado Descargado y se le envía una petición para que la ejecute, por otra parte; no se aceptan peticiones de ejecución de procesos por parte de procesadores remotos.

En la sección siguiente, se describen los elementos de información y de control de algunos algoritmos usados frecuentemente para la asignación dinámica de carga.

1.3 Ejemplos de Algoritmos de Asignación Dinámica de Carga

En esta parte se describe el comportamiento de los elementos de información y de control de algunos de los algoritmos de asignación dinámica para repartir la carga en los sistemas paralelos.

1.3.1 Algoritmo de tipo global centralizado

El comportamiento del este algoritmo se puede representar mediante el esquema de nodos (procesadores) y comunicaciones de la Figura 5. En esta figura se modela una Máquina Virtual Paralela compuesta por 4 nodos.

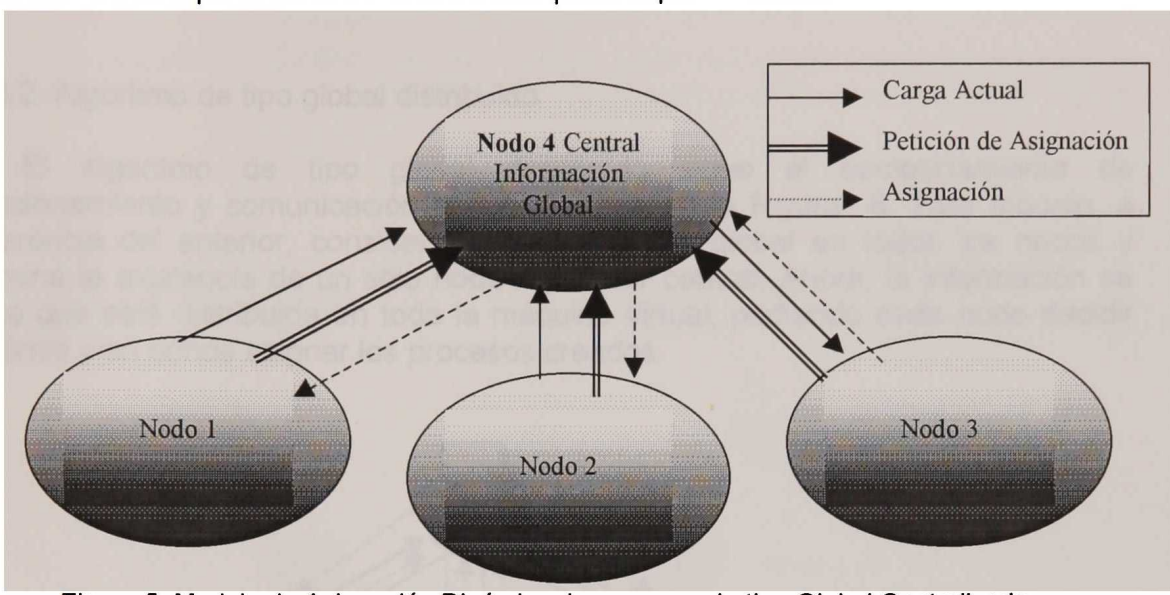


Figura 5. Modelo de Asignación Dinámica de procesos de tipo Global Centralizado.

En este algoritmo existe un nodo (el nodo4), quien es considerado como un nodo especial central, el cual se encarga de tener disponible la información relacionada con el estado de carga actual de todos los nodos en el sistema (información global). Para tal efecto, recibe continuamente los nuevos estados de carga de los nodos cuando ocurren cambios. Contando con dicha información, el nodo4 puede "visualizar" quién o quiénes son los nodos menos cargados.

El nodo central es el único que se encarga de decidir el lugar en donde van a ser asignados los nuevos procesos o tareas. Este nodo consulta la información global que posee sobre el estado de carga del sistema, para indicar a quien lo requiera, cuál es el nodo menos saturado de la máquina virtual y así poder asignar en forma dinámica el proceso en dicho nodo.

El comportamiento de los otros nodos (nodo1, nodo2 y nodo3), es semejante. Cuando se detecta un cambio con respecto al estado de carga local, se tiene que hacer actualizar dicha información, enviando mediante un mensaje, el nuevo estado de carga al nodo central 4. Ha de notarse que la actualización es sólo para el estado de carga del nodo que cambió y no en los demás.

Cuando algún nodo necesita hacer una creación de cierto proceso, interroga mediante un mensaje al nodo central, para obtener el nodo más

apropiado de ejecución. Una vez obteniendo dicho dato, se realiza la asignación del proceso sobre el nodo indicado.

Este es un esquema para 4 Nodos en la máquina virtual, pero dicho esquema permanece en esencia para 4 nodos o más.

1.3.2. Algoritmo de tipo global distribuido

El algoritmo de tipo global distribuido sigue el comportamiento de funcionamiento y comunicación esquematizado en la Figura 6. Este modelo, a diferencia del anterior, considera una información global en todos los nodos y elimina la existencia de un solo nodo repartidor central. Ahora, la información se dice que está distribuida en toda la máquina virtual, pudiendo cada nodo decidir cuándo y en dónde asignar los procesos creados.

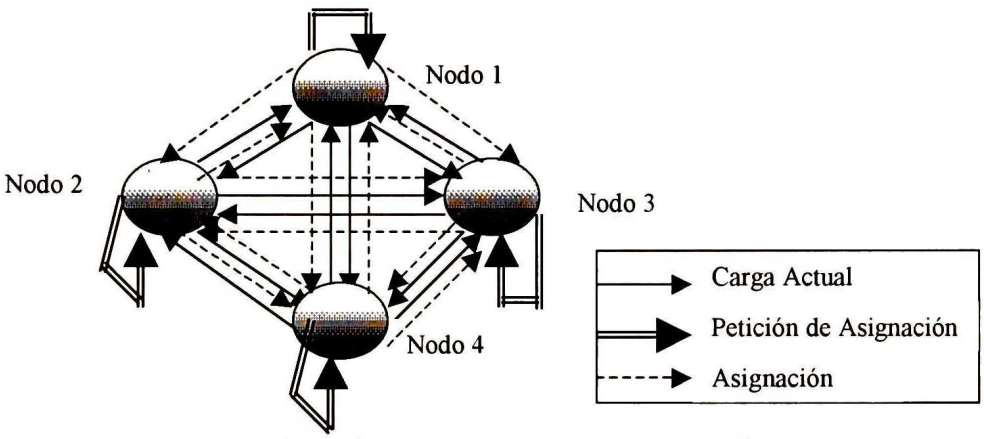


Figura 6. Modelo de asignación dinámica de procesos con información distribuida

El comportamiento de cada nodo respecto a la asignación de procesos, es semejante: cada nodo posee la información concerniente al estado de carga de todos los nodos en el sistema. Cuando un nodo detecta un cambio en su estado de carga local, envía un mensaje con dicho estado a los demás nodos de la máquina virtual, para que hagan la actualización correspondiente. Si esta actualización no se hiciera, los módulos de Información Global de los nodos contendrían información incoherente.

El modelo en comparación con el centralizado, tiene la ventaja de que la información está distribuida en toda la máquina virtual, por lo que permite distribuir el control. Sin embargo, la desventaja cuando esa información tiene que ser actualizada es que se tiene que enviar un número de mensajes igual al numero

total de nodos en la máquina virtual, considerando que cualquier cambio del estado de carga de un nodo tiene que ser actualizado en las demás máquinas.

1.3.3. Cíclico

El algoritmo Cíclico de tipo ciego (sin elemento de información) y centralizado, tiene un comportamiento representado con el esquema de la Figura 7

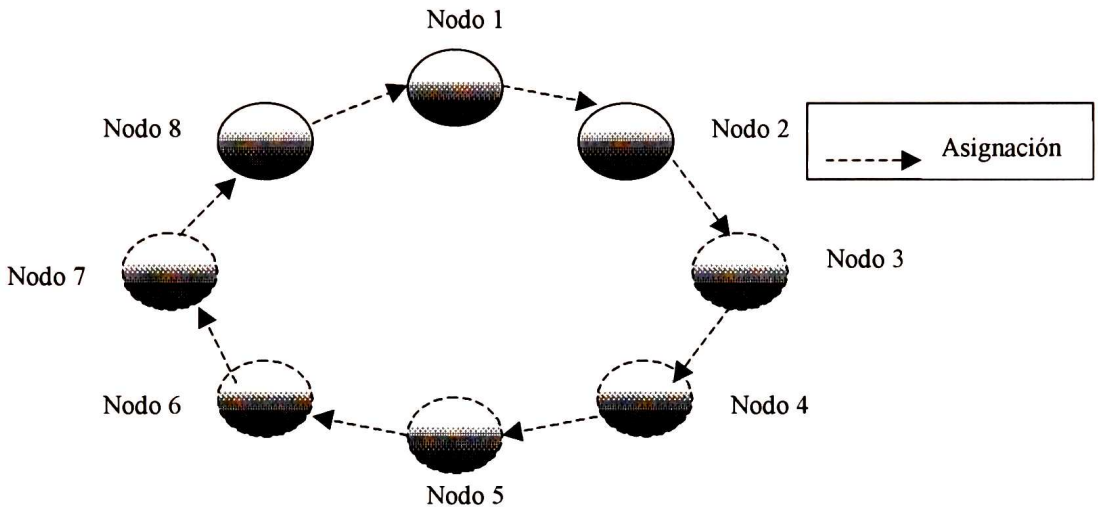


Figura 7. Modelo de asignación dinámica de procesos utilizando el algoritmo cíclico.

Como puede observarse, en este modelo no hay necesidad de poseer información respecto al estado de carga del sistema. Como en el caso centralizado, existe un nodo especial el cual determina en base a un orden cíclico, el nodo que ejecutará a un nuevo proceso. Ahora cada vez que un nodo requiere crear un proceso, consulta al nodo especial para saber el procesador donde se ejecutará.

Como se considera que la carga se repartirá equitativamente, no se hace necesaria la recolecta de información concerniente al estado de carga actual de los procesadores.

1.3.4. Vecinos directos

El algoritmo de asignación dinámica mediante el manejo de vecinos directos, se basa en una topología que puede ser una malla o red lógica (Figura 9)

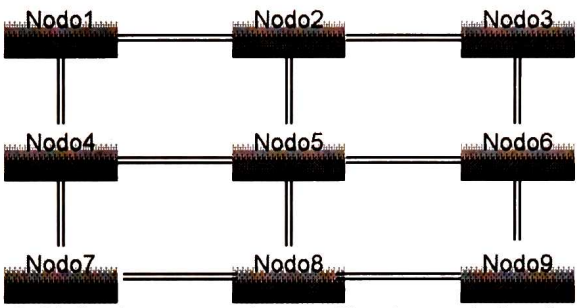


Figura 9. Malla de procesadores utilizada en el algoritmo de vecinos directos

Este algoritmo de asignación dinámica, es parecido al algoritmo de asignación dinámica de tipo Global Distribuido (Figura 6), en cuanto a que cada nodo puede decidir el lugar de asignación de los procesos creados basándose en cierta información.

La diferencia radica ahora en que la información, referente al estado de carga de los procesadores, poseída por cada nodo no es global. En este algoritmo todos los nodos poseen una información parcial del sistema, sólo el estado de carga de los vecinos directos (en la malla lógica), la cual será usada para tratar de repartir y equilibrar la carga global.

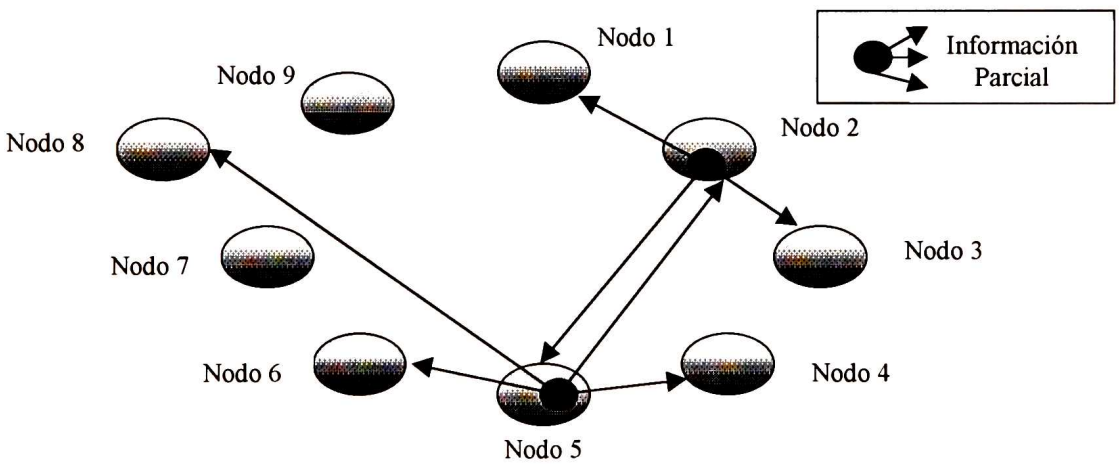


Figura 10. Asignación dinámica de procesos mediante el algoritmo de procesadores vecinos

El funcionamiento del algoritmo es el siguiente:

Cuando en un nodo del sistema, por ejemplo el nodo 2, se registra un cambio en su estado de carga local, se envía un mensaje a sus nodos vecinos (los directamente conectados): 1, 3 y 5 anunciando el cambio. Estos, en cuanto reciban dicho mensaje, actualizan su información parcial, considerando el nuevo estado de carga del nodo 2. Un comportamiento equivalente pasaría si cambia el estado de carga del nodo 5, sus vecinos 2, 4, 6 y 8 deben actualizar su información parcial.

La ventaja de usar un elemento de información de tipo parcial, es que ahora la actualización de información no se hace hacia todos los nodos del sistema (como se ilustra en la Figura 6), sino que solo se hace hacia los vecinos directos del nodo que cambió de estado (Figura 10).

1.3.5. Vectorial

El algoritmo de asignación dinámica Vectorial [5] se basa en el siguiente esquema (Figura 11):

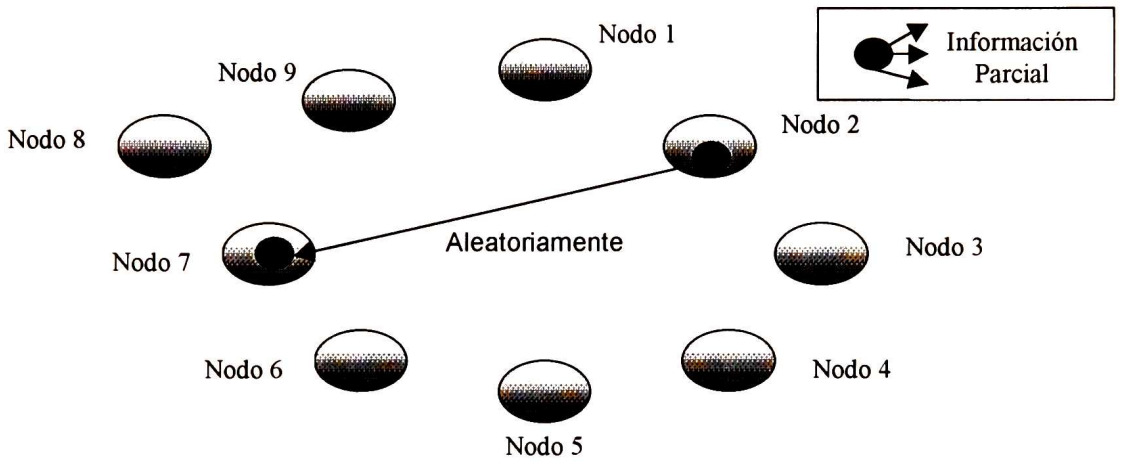


Figura 11. Asignación dinámica de procesos mediante el algoritmo Vectorial

El trabajo desarrollado en [5], asigna a cada nodo i un vector L conteniendo el estado de carga de m nodos: L[0],L[1],L[2],...L[m-1], m<n, donde n es el número total de nodos en el sistema. El primer elemento de L es el estado de carga local del nodo i y el resto son los estados de carga de otros m-1 nodos arbitrarios. Para cada nodo i, cuando ocurre un cambio en su estado de carga local, el vector es actualizado de la siguiente manera siguiendo un método aleatorio:

1. Se actualiza en L la carga propia del nodo i (L[0])

2. Se elige al azar un nodo j y se le envía la primer mitad del vector L.

3. Cuando el nodo j recibe la mitad del vector, modifica su propio vector L intercalando en las posiciones impares (L[1], L[3], L[5],..) de su vector los valores recibidos.

En base a esta información parcial (vector), cada procesador del sistema es capaz de decidir el sitio donde asignará un proceso.

A manera de ejemplo, si se tienen 2 nodos (i y j) con los vectores respectivos Li y Lj siguientes:

L i :

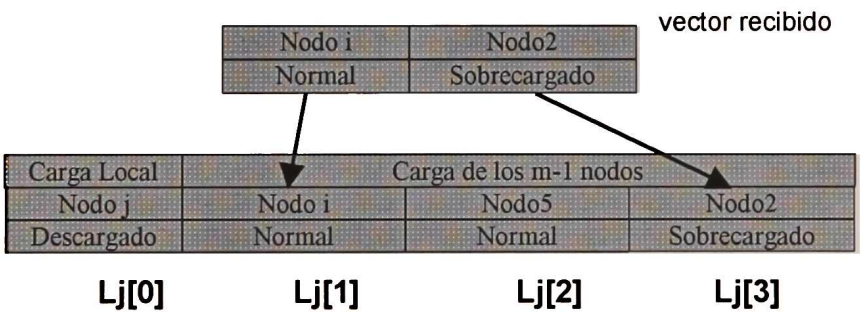
Carga Local	Carga de los m-1 nodos		
Nodo i	Nodo 2	Nodo 7	Nodo 8
Normal	Sobrecargado	Descargado	Descargado

L j (antes de actualización):

Carga Local	Carga de los m-1 nodos		
Nodo j	Nodo 1	Nodo 5	Nodo 6
Descargado	Normal	Normal	Sobrecargado

Si el nodo i requiere hacer una actualización de su estado de carga de Normal a Descargado, modifica el primer elemento de su vector **Li**[0] y elige aleatoriamente a un nodo j para enviarle la primer mitad de **Li**. Al recibir dicha información, el nodo j la intercala en su propio vector **Lj** como se muestra a continuación.

L j (después de actualización):



Después de haber presentado las características de algunos algoritmos básicos para la asignación dinámica de procesos y para favorecer el estudio y aplicación de estas técnicas, se desarrolló una plataforma que será expuesta con mayor detalle en el siguiente capítulo.

CAPÍTULO 2

PLATAFORMA PROPUESTA PARA LA IMPLEMENTACIÓN Y ESTUDIO DE ALGORITMOS DE REPARTICIÓN DINÁMICA

En este capítulo presentamos la plataforma desarrollada para estudiar e implementar algoritmos de diferentes tipos para la asignación dinámica de procesos. Con la ayuda de la herramienta PVM [3] (Parallel Virtual Machine), pudimos trabajar con varios procesadores interconectados mediante una red como si fuera una sola máquina virtual multiprocesadora (paralela). Las funciones proporcionadas por PVM permiten crear procesos automáticamente sobre la máquina virtual y permiten, mediante el intercambio de mensajes, la comunicación y sincronización de estos. Las características de PVM son descritas en el *Anexo A.* Para una descripción más detallada, ver el manual de PVM [3].

2.1 Módulos Básicos

Para poder implementar un algoritmo de asignación dinámica, cada nodo del sistema ejecuta un conjunto de procesos, los cuales interactúan entre sí (a nivel local). Esos procesos pueden verse como módulos que tienen su propia tarea y cuyo funcionamiento es complementario (Figura 11).

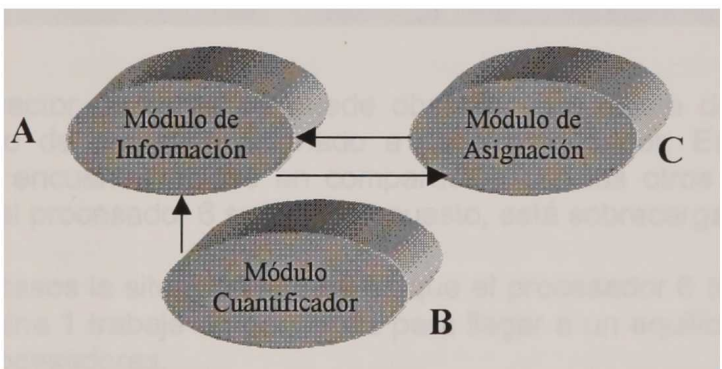


Figura 11. Modelo de implementación de algoritmos de asignación dinámica

Los módulos A, B y C de un nodo pueden también colaborar con los módulos A, B y C de otros nodos en el sistema. Dependiendo del algoritmo que se implante, los módulos tienen un comportamiento determinado dependiendo de los tipos de sus elementos de control y de información, para esto a continuación se presenta la tarea y funcionamiento general de cada uno de estos módulos.

2.1.1 El módulo de información A

Este módulo se encarga de almacenar los datos referentes a la carga actual de los diferentes procesadores que componen la Máquina Virtual[3]. Dependiendo del tipo del elemento de Información del algoritmo, los datos consideran una información parcial o global del estado de carga. Estos datos son actualizados por:

- El módulo cuantificador **B** (local), cuando ocurre un cambio en cuanto a la carga del propio procesador.
- Otros módulos de información **A** remotos (en otros nodos), cuando ocurre un cambio en cuanto a la carga de otros procesadores.

Los datos son almacenados en un arreglo al cual llamaremos vector de carga. Un ejemplo de vector de carga con 6 procesadores se muestra a continuación

	Procesador 1	Procesador 2	Procesador 3	Procesador 4	Procesador 5	Procesador6
Carga o Número de Procesos.	descargado	carga normal	carga normal	carga normal	carga normal	sobrecargado

En este vector de carga se puede observar que existe desequilibrio en cuanto al número de procesos asignado a cada procesador. Es claro que el procesador 1 se encuentra ocioso en comparación con los otros procesadores, mientras que en el procesador 6 sucede lo opuesto, está sobrecargado.

En estos casos la situación idónea es que el procesador 6 trabaje un poco menos y la máquina 1 trabaje un poco más para llegar a un equilibrio en la carga para todos los procesadores.

	Procesador 1	Procesador 2	Procesador 3	Procesador 4	Procesador 5	Procesador6
Carga o Número de Procesos.	carga normal	carga normal	carga normal	carga normal	carga normal	carga normal

Como se ve, es de suma importancia este módulo de información, ya que es medio fiable para llegar a un estado de equilibrio, además de que en todo

momento conoce la información en cuanto al estado de carga actual de los procesadores con los que puede compartir trabajo.

Este módulo de información generalmente colabora con el módulo de asignación **C** para ayudarle a decidir el lugar donde se puede asignar un nuevo proceso con el objetivo de no deteriorar del rendimiento del sistema.

Un poco más adelante se hablará de la forma en que puede estar distribuido o centralizado este vector de información, observando sus ventajas y desventajas del manejo de la información.

2.1.2 El módulo cuantificador B

La principal tarea de este módulo es la de cuantificar con cierta frecuencia, la carga del nodo en el cual se encuentra. Cuando se registra algún cambio en cuanto a la carga del nodo, el módulo cuantificador envía un mensaje al módulo de información **A** indicando el cambio. El cambio puede ser un incremento o decremento de la carga.

En la plataforma propuesta, este módulo puede cuantificar la carga de 2 formas: De manera unitaria o bien mediante un mecanismo de doble frontera.

En la manera unitaria se detecta cualquier cambio con respecto al número de procesos.

> **Ejemplo:**
>
> Si se registra un cambio en el número de procesos ejecutados en un cierto procesador, de 5 a 6 por ejemplo, este cambio es considerado y tiene que ser actualizado en el módulo de información. Lo mismo ocurre cuando la cantidad de procesos se decrementa pasando de 6 a 5, nuevamente se debe realizar una actualización sobre el Módulo **A.**

Esta forma de actualizar de la carga aunque aparentemente es muy precisa, no resulta adecuada cuando se manejan grandes cantidades de procesos (100-1000), porque provocan saturación en el canal de comunicación entre el módulo cuantificador y el módulo de información ya sea del mismo nodo o de otro.

La otra manera implantada para cuantificar la carga, es usando un mecanismo de doble frontera, como se explicó en la sección 1.2 del capítulo anterior. Aquí la carga de un procesador se estima mediante tres posibles

estados: Descargado, Normal y Sobrecargado, considerando dos fronteras de carga.

Ejemplo:

Si se tienen las fronteras F1= 40 procesos y F2 = 60 procesos y en un nodo el número de procesos ejecutados cambia de 5 a 6, no se genera una notificación hacia el módulo de información, pero se incrementa un contador. Si la cantidad de procesos activos en dicho nodo se vuelve a incrementar, es decir, de 6 a 7, se incrementa nuevamente el contador y así sucesivamente. Cuando dicho contador alcanza un valor máximo de 40, en ese momento se detecta un cambio de estado de carga (de Descargado a Normal) y se le notifica al Módulo **A**, para que actualice su información.

Lo importante aquí es saber cuáles son los valores de las dos fronteras que hay que alcanzar, ya que depende de esto el grado de repartición de la carga que se obtendrá. Si las Fronteras son muy pequeñas, entonces habrá una tendencia a repartir lo más posible el trabajo entre todos los nodos del sistema.

En el caso particular en que la *frontera1* = *frontera2* = *F* un nodo puede ejecutar procesos locales y remotos mientras su estado de carga no cruce la frontera *F*. Cuando la frontera *F* es cruzada, el nodo no aceptará más carga e intentará transferirla sobre nodos descargados.

2.1.3 El módulo de asignación C

Este módulo es el encargado de implantar el elemento de control de un algoritmo de asignación dinámica. Cuando un nodo necesita crear un nuevo proceso, el módulo de asignación **C** puede consultar al módulo de Información **A** para saber el nodo o procesador que en ese momento se encuentra menos saturado de trabajo.

Este módulo **C** de asignación o repartición realiza como un primer paso una consulta al módulo de información mediante el envío de un mensaje, el módulo **A** de información revisa su vector de carga y manda una respuesta al módulo de repartición con el número o nombre del nodo que tiene registrado como el menos cargado. El módulo de asignación es en sí el que administra las asignaciones de procesos nuevos hacia los procesadores mas descargados.

2.2 Descripción de la Plataforma

En la plataforma desarrollada fueron programados los 5 algoritmos de asignación dinámica descritos en la sección 1.3: el Centralizado, el Completamente Distribuido, el Cíclico, el Vectorial y el de Vecinos Directos. Mediante un parámetro especificado antes de la compilación, el usuario puede seleccionar con cuál de esos algoritmos desea trabajar al ejecutar su aplicación.

Los 3 módulos necesarios para el manejo de los algoritmos dinámicos (Información, Asignación y Cuantificación) fueron implementados como procesos, además de un proceso de inicialización.

Luego entonces, en la plataforma propuesta se tienen 4 programas escritos en ANSI 'C' que utilizan las librerías de PVM. Estos programas son manejados como procesos que se ejecutan sobre cada nodo de un cúmulo compuesto por 9 máquinas Pentium III con Sistema Operativo Linux 7.1 y en una topología de red tipo BUS (Fig. 12).

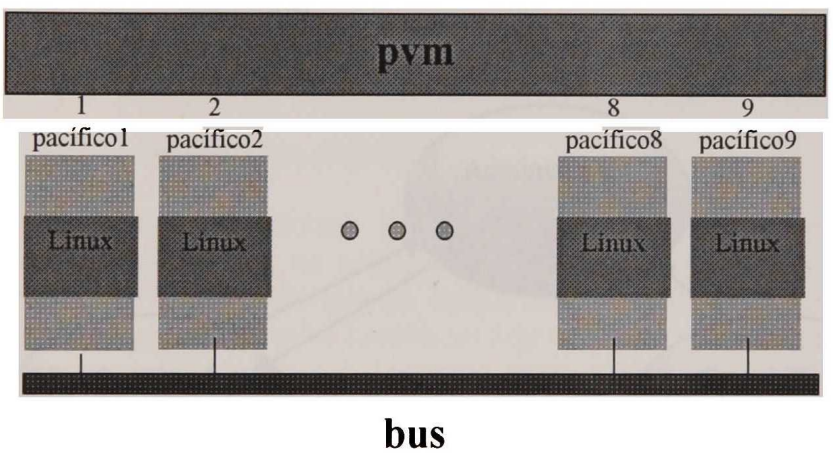


Figura 12. Características del Cluster de Experimentación.

A continuación se describen los 4 programas realizados:

2.2.1 Inicialización

Este proceso es ejecutado una sola vez y su función como el nombre lo dice, es el de inicializar nuestra plataforma, en este proceso se da el manejo de algunos parámetros como el tipo de algoritmo de asignación que se va a usar (Centralizado, Distribuido, Vecinos, etc.).

Otros parámetros que se manejan, son los valores de las 2 fronteras de carga (F1 y F2), revisadas en la sección 1.2, así como la frecuencia de cuantificación de la carga (ver sección 1.1).

2.2.2 Administrador de información

Con este programa se implanta el módulo de información descrito anteriormente. Sus interacciones con los otros programas se describen en la Figura 13.

Este administrador tiene entre sus funciones:

- Recibir la información del proceso contabilizador de carga para actualizar la información referente al estado de carga de cierto procesador (local o remoto).
- Recibir peticiones de información desde un asignador de procesos. La tarea al recibir una petición de información es la de distinguir el nodo menos cargado (búsqueda lineal en el vector de información) y luego enviar como respuesta ese nodo.

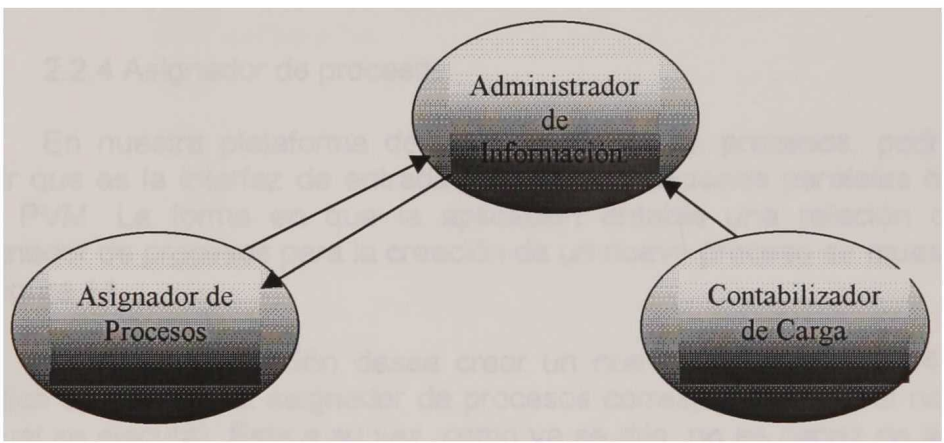


Figura 13. Funciones del administrador de información

2.2.3 Contabilizador de carga

Este proceso tiene la función de contar el número de procesos de la aplicación con cierta frecuencia (cada segundo por ejemplo). Los procesos contabilizados forman parte de la misma plataforma y del sistema. La forma de contar es verificando el número de procesos que se encuentran en el directorio */proc,* (esto se realiza redireccionando el listado de todos los identificadores del directorio hacia un archivo para después contar los

procesos en el archivo creado). En esta plataforma se considera el uso del mecanismo de doble frontera (se manejan tres estados de carga) para tratar de lograr un balance global de la carga.

En sí, el proceso contabilizador de carga se encarga de verificar en qué región de carga se encuentra un procesador. Recordemos, un procesador puede estar en:

- Región 1 Descargado.
- Región 2 Carga Normal.
- Región 3 Sobrecargado.

Si el número de procesos aumenta o disminuye, pero no se cambia de región, no se hace nada; pero si se cambia de región, se notifica al proceso que administra la información mediante el envío de un mensaje en el cual se indica la nueva región de carga y el procesador que cambió (Fig. 13). Más adelante en la sección 2.3 se explica con más detalle el funcionamiento de este proceso, dependiendo del algoritmo de repartición que se está utilizando.

2.2.4 Asignador de procesos

En nuestra plataforma de este asignador de procesos, podríamos decir que es la interfaz de entrada para las aplicaciones paralelas hechas con PVM. La forma en que la aplicación entabla una relación con el asignador de procesos para la creación de un nuevo proceso se muestra en la Figura 14.

Si alguna aplicación desea crear un nuevo proceso (Fig. 14-1), le notifica este deseo al asignador de procesos correspondiente (del nodo en el cual se ejecuta). Este a su vez, como ya se dijo, no es capaz de asignar directamente debido a que requiere la información necesaria de la carga del sistema (global o parcial), información que la va a obtener del administrador de información (Fig. 14-2). Una vez recibida esta información (Fig. 14-3) se procede a asignar este proceso al procesador menos cargado para que lo ejecute (Fig. 14-4).

La instrucción de PVM que nos permite realizar la creación de un nuevo proceso asignándolo a un determinado nodo es la siguiente:

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

donde:

task Cadena de caracteres conteniendo el nombre del archivo del proceso ejecutable a ser iniciado.

argv Puntero a un arreglo de argumentos para el programa ejecutable.

flag Puede tener las siguientes opciones:

- *0 pvm elige la máquina donde comenzar la tarea.*
- *1 se especifica una máquina con el parámetro where.*
- *2 se especifica una arquitectura en el parámetro where.*

where Cadena de caracteres que especifica donde comenzar él proceso pvm, dependiendo del valor de flag.

ntask entero que especifica el número de copias del programa ejecutable a ser iniciado.

tids Devuelve el identificador del proceso iniciado.

numt Devuelve el número de procesos iniciados, un valor devuelto de 0 indica error.

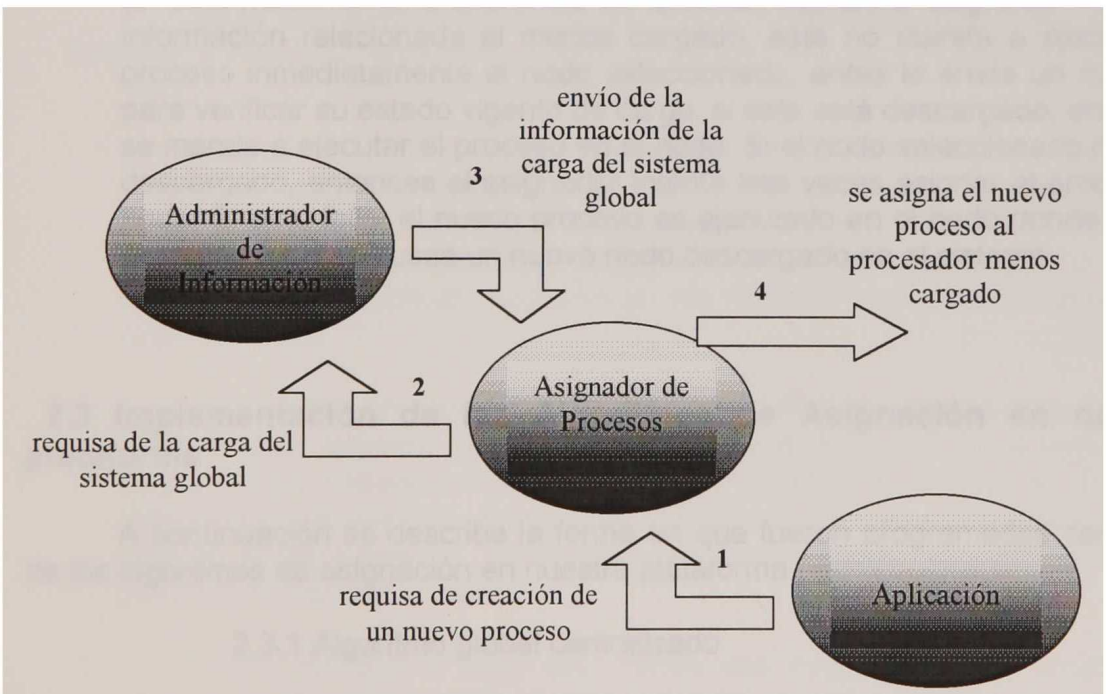


Figura 14. Etapas para la Creación de un Nuevo Proceso.
(Las flechas indican la dirección de los mensajes).

En la plataforma de experimentación se toma como un parámetro más el rechazo de ejecución de procesos o el no rechazo. Se puede notar de la Figura 14 que el asignador de procesos consulta información del administrador de información. Este identifica, en la información poseída, un procesador que se encuentra con un estado de carga menor y le envía esa información al asignador que hizo la consulta. Una vez recibida la información, el asignador puede comportarse de una de dos maneras: realizando la asignación directamente, o bien, utilizando un mecanismo de rechazo como se explica en los dos puntos siguientes.

2.2.4.1 Mecanismo sin rechazo

Cuando el asignador recibe la información concerniente a un nodo menos cargado, la toma como actual y manda a ejecutar al nuevo proceso en el nodo indicado. En un mecanismo sin rechazo, el procesador el cual se eligió para ejecutar el nuevo proceso siempre lo acepta y lo ejecuta, independientemente de que si su estado de carga hubiera cambiado antes de que pudiera actualizarse la información del administrador de información que lo eligió.

2.2.4.2 Mecanismo con rechazo

En este mecanismo, a diferencia del anterior, cuando el asignador recibe la información relacionada al menos cargado, este no manda a ejecutar el proceso inmediatamente al nodo seleccionado, antes le envía un mensaje para verificar su estado vigente de carga, si este está descargado, entonces se manda a ejecutar el proceso en el nodo. Si el nodo seleccionado no está descargado, entonces el asignador intenta tres veces asignar el proceso al nodo. Si esto falla, el nuevo proceso es ejecutado en el nodo donde recide el asignador o se busca un nuevo nodo descargado en el sistema.

2.3 Implementación de los Algoritmos de Asignación en nuestra plataforma

A continuación se describe la forma en que fueron programados cada uno de los algoritmos de asignación en nuestra plataforma

2.3.1 Algoritmo global centralizado

El primer paso a seguir para echar a andar este algoritmo es la inicialización, en donde se hace ejecutar el proceso que administra la

información global en algún nodo del cluster. En esta misma inicialización se hace ejecutar un proceso contabilizador de carga en cada uno de los nueve procesadores del cluster; se necesitan nueve porque la carga durante la ejecución de la aplicación paralela puede variar en todos los procesadores. Por último, se crea en cada nodo el asignador de procesos el cual va a atender los requerimientos de nuevas creaciones de procesos por parte de la aplicación.

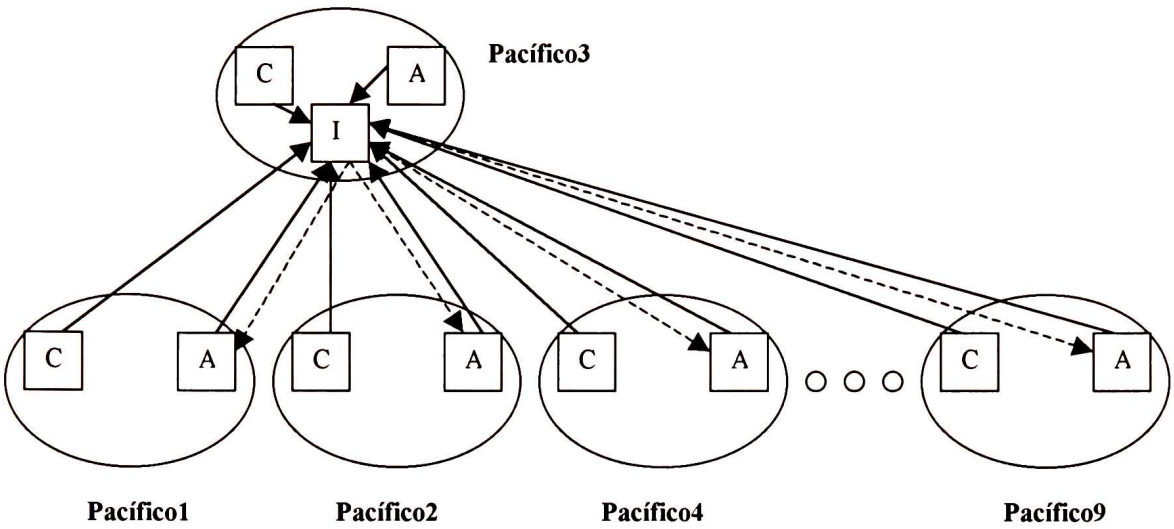


Figura 15. Implementación del algoritmo de repartición de tipo global centralizado en nuestra plataforma. A representa al asignador de procesos, C al contabilizador de carga e I al administrador de información.

De la figura 15 se puede notar que, cuando se registra un cambio de estado de carga en cualquiera de los nodos (pacífico1...pacífico9), se le debe notificar al administrador de información en (pacífico3) del cambio de la carga en dicho nodo mediante un mensaje. En PVM el destino de los mensajes es controlado mediante un identificador numérico de tarea (TID task identifier).

Cualquier aplicación que se esté ejecutando en algún procesador del cúmulo que requiera crear un nuevo proceso, se lo comunica a su asignador de procesos local (A) como se muestra en la Figura 15. Este último a su vez le envía un mensaje al administrador de la información para que le proporcione el procesador menos cargado en ese momento (información de tipo global).

Una vez que el asignador recibe esa información, se encarga de asignar a ese procesador la ejecución del nuevo proceso, para esto la instrucción *(pvm_spawn)*[10] nos permite crear nuevos procesos en forma dinámica.

2.3.2 Algoritmo global distribuido

En el algoritmo completamente distribuido en la parte de inicializacion se crean nueve procesos (uno por nodo) que administran la información. Despues se hace ejecutar un proceso contabilizador de carga en cada uno de los nueve procesadores del cúmulo. Por último, se hace ejecutar en cada nodo un asignador de procesos, los cuales como ya se dijo, atenderán los requerimientos de nueva creación de procesos por parte de la aplicación (Figura 16).

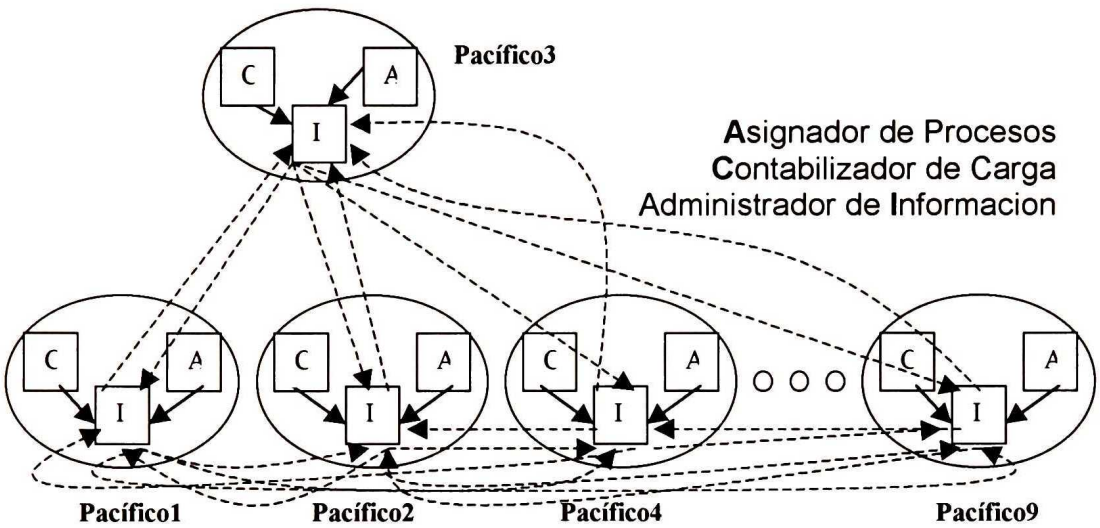


Figura 16. Implementación del algoritmo de repartición de tipo global distribuido en nuestra plataforma.

En la implementación de este algoritmo, cada nodo debe poseer una información global del estado de carga del sistema. Cuando se detecta un cambio en el estado de carga por parte del contabilizador de carga en un nodo, por ejemplo, en el nodo Pacífico4 (Figura 16), este se lo notifica a su administrador de información local para que se actualice. Debido al manejo de información global y control distribuido, esta información debe también ser actualizada en todos los administradores de información del cúmulo. El administrador localizado en Pacífico4 es entonces el encargado de informar a todos los demás administradores del cambio, percatado en su parte del cúmulo (Figura 17).

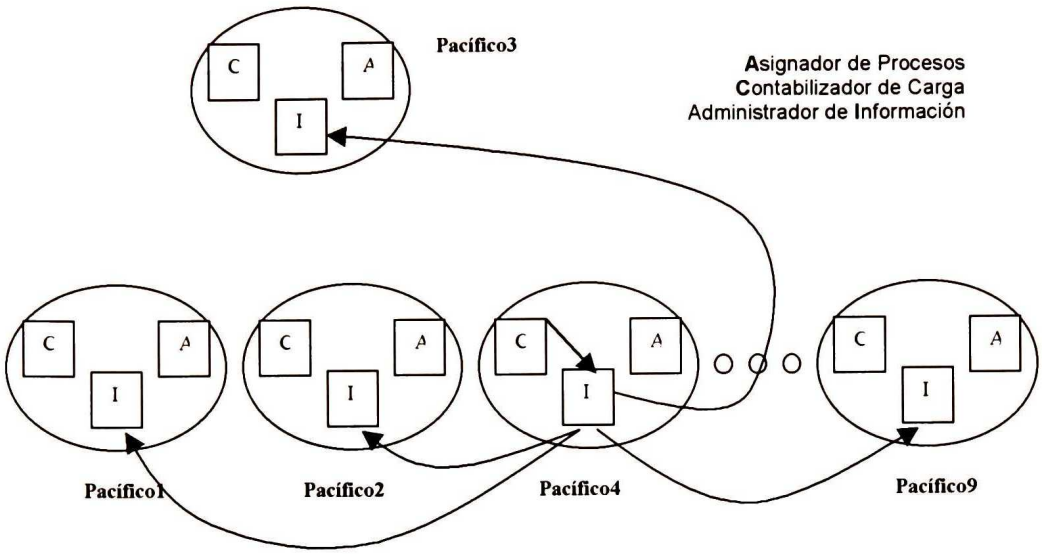


Figura 17. Mensajes enviados para la actualización de la carga a todos los nodos del sistema

Para dicha actualización de información, todos los administradores de información conocen los TIDS (identificadores de tarea) de todos los administradores de información en el cúmulo. Con esta información el sistema realiza un multicast para la actualización de la información en todo el sistema usando la función *(pvm_mcast)* [16].

En cuanto al funcionamiento del proceso asignador en cada nodo, es en parte similar al algoritmo Centralizado explicado en la subsección 2.3.1, ya que el asignador de procesos va a ser el medio por el cual la aplicación va a requerir de la creación de un nuevo proceso. Para realizar esto, el asignador se va a comunicar con el administrador de información local de la siguiente manera:

Si el asignador de procesos de Pacífico4 requiere crear un nuevo proceso, este le pide al administrador de información local el identificador del procesador menos cargado en el sistema. Dicho administrador local posee ya una información global que contiene el estado de carga de todo el cúmulo.

Una vez que el asignador conoce el identificador del procesador menos cargado, hace la asignación para que el nuevo proceso se ejecute ahí.

2.3.3 Algoritmo cíclico

Para este algoritmo cíclico, se tiene una parte de inicialización donde no se se consideran ni procesos que se encarguen de la administración de la información, ni de procesos contabilizadores de carga, sólo se hace ejecutar en un solo nodo un asignador de procesos (por ejemplo en Pacífico3), este se encargará como ya se dijo de atender los requerimientos para la nueva creación de procesos por parte de los asignadores de las aplicaciones (Fig. 18).

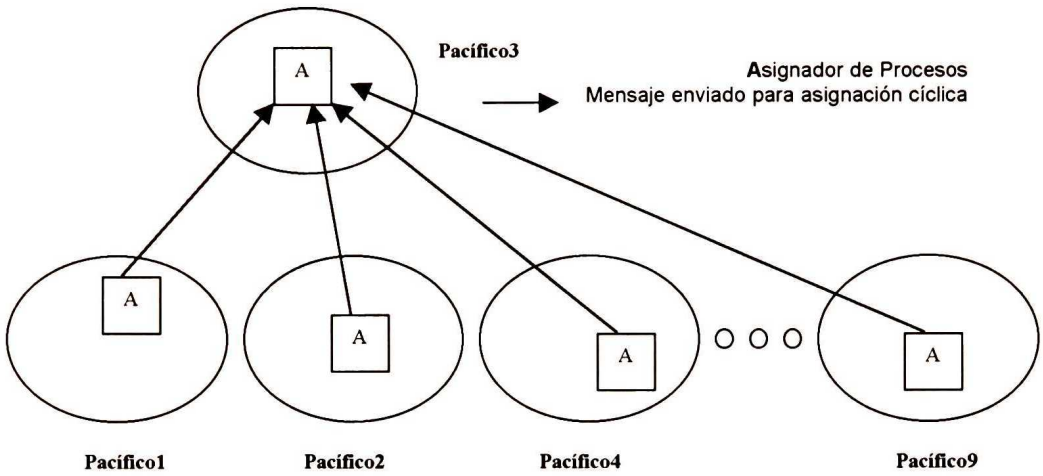


Figura 18. Implementación del algoritmo de repartición de tipo cíclico en nuestra plataforma.

Bajo este esquema cuando cualquier aplicación requiere de la creación de un nuevo proceso, este envía un mensaje al único asignador de procesos. Este tiene una variable la cual le indica a donde será ejecutado el nuevo proceso, dicha variable se incrementa en uno para permitir que el siguiente nuevo proceso sea ejecutado en el siguiente nodo.

2.3.4 Algoritmo de vecinos directos

Al igual que en los algoritmos anteriores, el algoritmo de vecinos directos tiene una inicialización, en la cual se dan de alta nueve administradores de la información en cada uno de los elementos del cúmulo. En esta parte se fijan tambien las fronteras de carga y las frecuencias de cuantificación. También se inicializan nueve contabilizadores de carga y nueve asignadores de procesos en todo el cúmulo.

En este algoritmo, a diferencia de los anteriores, el proceso que administra la información va a ser el encargado de manejar solo información de tipo parcial, es decir, solo tendrá en su vector de carga (ver las subsecciones 1.3.4,1.3.5) información local y de sus vecinos directos.

En nuestra plataforma definimos una malla lógica y el manejo de vecinos directos se consideró de la siguiente forma (Figura 19):

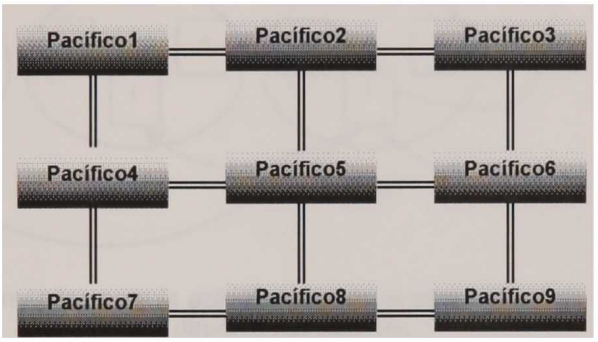


Figura 19. Malla lógica para la implantación del algoritmo de asignación de vecinos directos.

De la Figura 19 se puede notar que se está manejando una malla lógica para determinar los vecinos en el cúmulo. Esto es debido a que se tiene una configuración de red tipo bus (Fig. 12). En base a la Figura 19, se nota que el procesador Pacífico1 tiene como vecinos a Pacífico2 y Pacífico4. Pacífico4 tiene como vecinos directos a Pacífico1, Pacífico7 y Pacífico5. La determinación de los vecinos se realiza en el proceso que administra la información.

La función del administrador de información, es similar que en los algoritmos anteriores (ver subsecciones 2.3.1,2.3.2) a diferencia de que, cuando algún proceso contabilizador de carga detecta un cambio de estado de carga, se lo notifica a su administrador de información, quien a su vez notifica dicho cambio de carga a sus vecinos (Figura 20). Esta notificación de tipo parcial es diferente a la manejada en el algoritmo completamente distribuido (explicado en la subsección 2.3.2), donde la notificación era a todos los elementos del cúmulo y no sólo a los vecinos directos.

De igual forma, cuando alguna aplicación requiere de la creación de un nuevo proceso, se hace una notificación al asignador de procesos local. Este a su vez requiere de la información del Administrador local de información, el cual identifica y devuelve el procesador menos cargado tomando en cuenta sólo a él mismo y a sus vecinos.

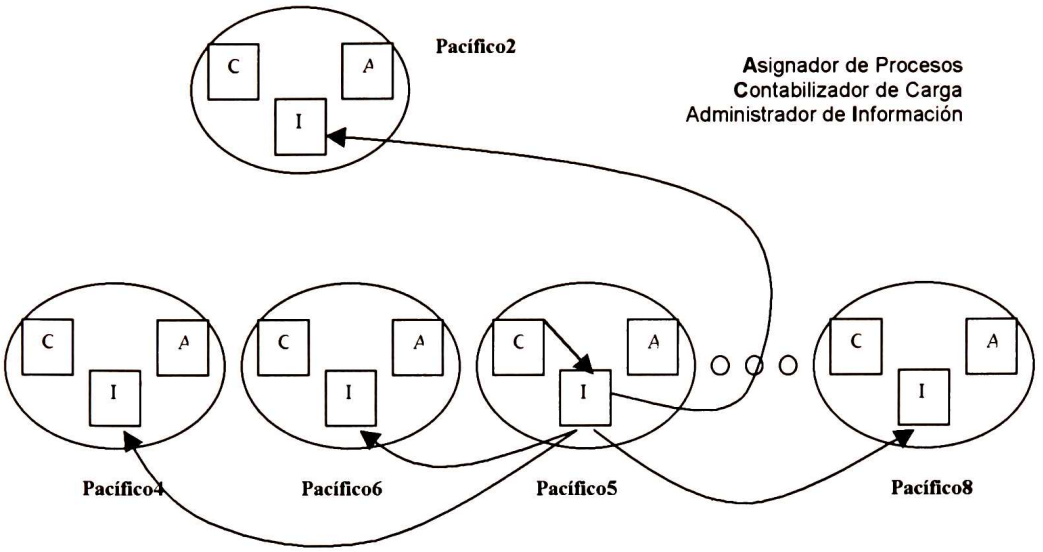


Figura 20. Mensajes enviados para la actualización de la carga del nodo 5 a todos mis vecinos directos del sistema.

2.3.5 Algoritmo vectorial

El algoritmo de asignación de tipo vectorial fue realizado de la siguiente forma:

Se da un manejo muy similar al algoritmo de vecinos directos, es decir, se tiene un proceso de inicialización en donde se crean nueve procesos que administran una información parcial en cada nodo del cluster, también se crean nueve que contabilizan la carga y nueve que asignan procesos.

En cada uno de los nueve procesos que administran la información, se considera la existencia de un vector de información de tamaño M que contiene el estado de carga de M nodos, elegidos estos de manera aleatoria. Como en los algoritmos anteriores, el proceso administrador de la información, es el responsable de recibir las notificaciones de cambio del estado de carga local o remoto, enviadas por el contabilizador.

Si el proceso contabilizador detecta un cambio en el estado de carga, se lo notifica al administrador de información, el cual a su vez procede a hacer lo siguiente:

1.- Guarda en la primer localidad de su vector el nuevo estado de carga adquirido

2.- Elige aleatoriamente un nodo del cluster y le envía la primera mitad de su vector de información a dicho nodo (1.3.5).

3.- Cuando un nodo elegido al azar recibe esa mitad del vector (en donde viene el cambio registrado en la carga del procesador emisor), la mezcla intercalándola con su vector local actual. Dicho manejo se representa en la Figura 21.

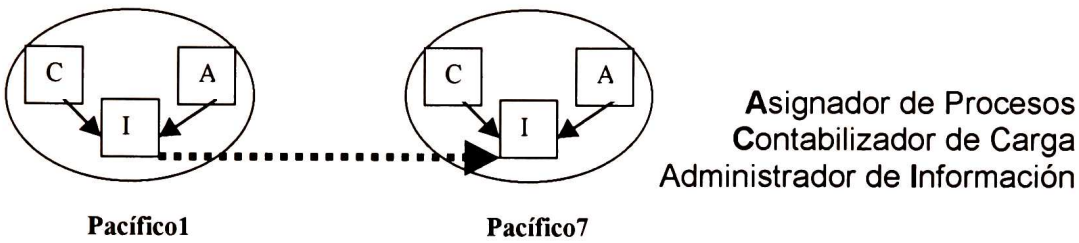


Figura 21. Envío de la mitad del vector de información a otro procesador

De la figura 21, se puede notar cuando se detecta un cambio en el estado de carga de Pacífico1 por parte del Contabilizador de Carga (C), este se lo notifica a su Administrador de Información (I), el cual actualiza su propio vector de carga y envía después la mitad de ese vector conteniendo dicha actualización a otro Administrador de Información (I) de un procesador elegido aleatoriamente (Pacífico7), este a su vez recibe la mitad de ese vector y lo intercala con su vector propio (1.3.5).

Vector de carga para Pacífico1

Nodo	Pacífico1	Pacífico5	Pacífico6	Pacífico2
Carga	Carga normal	Sobrecargado	Descargado	Descargado

Vector de carga de Pacífico7 antes de la actualización

Nodo	Pacífico7	Pacífico9	Pacífico3	Pacífico6
Carga	Carga normal	Sobrecargado	Sobrecargado	Carga normal

Nuevo vector de carga para Pacífico7, después del envío de la primera mitad del vector de carga de pacífico1 e intercalando con las posiciones impares

Nodo	Pacífico7	Pacífico 1	Pacífico 3	Pacífico 5
Carga	Carga normal	Carga normal	Sobrecargado	Sobrecargado

En el momento de la creación de un nuevo proceso, el proceso asignador recibirá de parte del proceso administrador de información, el identificador del procesador con menor carga a nivel vectorial, es decir, en base a los procesadores que aleatoriamente se eligieron.

El comportamiento de los algoritmos pudo ser estudiado y comparado gracias a su realización en la plataforma desarrollada. En el capítulo siguiente, describimos las aplicaciones paralelas usadas para evaluar el rendimiento de tales algoritmos.

CAPITULO 3
PROGRAMAS EJECUTADOS.

En la actualidad se han desarrollado diversos evaluadores que en cierta forma cuantifican el rendimiento de los programas que obtienen los resultados de un problema real. De esta forma es posible encontrar los mejores parámetros de ejecución para tal aplicación en particular [11, 12, 6, 20, 18]. Cuando se quiere estudiar y medir el rendimiento de un sistema paralelo en cuanto a la explotación de los recursos y en cuanto al tiempo de ejecución, el uso de programas que sintetizan el comportamiento de los programas reales simplifica el estudio [13].

Un **programa sintético** no da resultados exactos a un problema real, pero sí genera una carga aproximada en el sistema. Varios programas que resuelven problemas reales pueden abstraerse mediante cierto programa sintético y pueden ser analizados en cuanto a la carga que generan en el sistema. Cabe mencionar que la carga se puede referir tanto a los recursos de comunicación como al uso de los procesadores. Algunos ejemplos de uso de este tipo de programas se pueden encontrar en [15, 9, 14, 19].

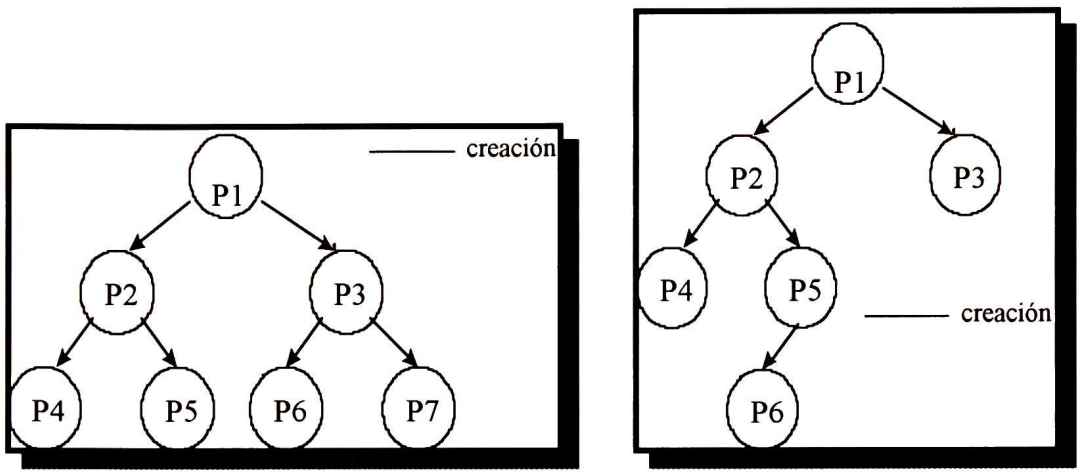
En este capítulo, se describen los programas sintéticos escritos con PVM que nos ayudaron en la evaluación de los algoritmos de repartición dinámica. Los programas sintéticos utilizados se componen de un conjunto de procesos creados dinámicamente (en tiempo de ejecución) que hacen trabajar al cúmulo realizando una cantidad dada de cálculos (sumas por ejemplo) sin que el resultado de las operaciones tenga una interpretación o significado. Con la misma filosofía, los procesos generan mensajes en el sistema (mensajes sin interpretación alguna), para utilizar los canales de comunicación al transferir una cierta cantidad de bytes.

Con la ejecución de estos programas de prueba, se han generado un conjunto de resultados que se presentarán en el capítulo siguiente. Los programas se reagruparon en dos clases: los programas en donde los procesos envían un mensaje y los programas en donde los procesos intercambian varios mensajes. Para ambos casos se consideraron programas cuyos grafos de creación de procesos tienen las características que se presentan en la sección siguiente.

3.1 Grafos de Creación de los Programas de Prueba

Dentro del conjunto de programas de prueba, consideramos a su vez dos características referentes a los grafos de creación de procesos:

- Programas con grafo de creación en forma de árbol completo (Figura 19-a) en donde cada nodo (menos las hojas) tiene el mismo número K de hijos y
- Programas con grafo de creación en forma de árbol irregular (por ejemplo como el de la Figura 19-b) en donde cada nodo (menos las hojas), puede tener un número diferente de hijos.



a) Árbol Completo b) Árbol Irregular

Figura 19. Estructura de los programas de prueba ejecutados.

Para nuestro estudio, en la estructura de los programas de prueba en forma de grafo de creación de árbol completo, trabajamos con los tres casos especiales siguientes:

El primer caso se refiere a un grafo de creación en forma de árbol, con una altura H > 2 y un número de hijos K > 1. En la Figura 19-a tenemos un ejemplo de este tipo de grafo con H = 3 y K= 2.

El segundo caso especial considera un grafo de creación de procesos que es un árbol de poca profundidad y de gran amplitud, como se muestra en la Figura 20 (H=2 y K= N). Este tipo de grafo es interesante, ya que lo ideal sería tener un número de procesadores igual al número de procesos creados, para explotar el máximo de paralelismo. Los algoritmos de asignación de carga deberán entonces, asignar la carga de la manera más equilibrada posible para obtener buenos tiempos de ejecución.

- **Caso 1: 1 Padre – N hijos**

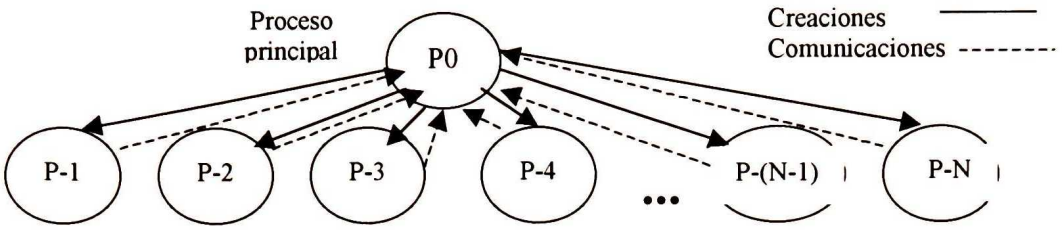


Figura 20. Árbol completo con un padre con N hijos

El tercer caso consiste en generar un grafo donde la profundidad es grande y la amplitud es limitada a un proceso. En la Figura 21, se puede ver un ejemplo de este tipo de grafo donde la profundidad H= N+1 y se tiene un solo hijo, K=1, creado por proceso.

Este último caso también es interesante, ya que si existe mucha comunicación entre un proceso padre y su hijo (por ejemplo P1 con P2 ó P2 con P3), sería conveniente asignar a ambos procesos a un mismo procesador para reducir el costo en comunicación.

- **1 Hijo – Altura N**

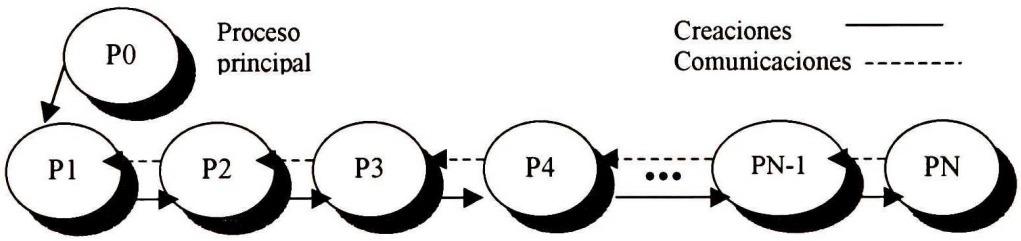


Figura 21. Árbol completo con un hijo y altura N

Un programa cuyo grafo de creación de procesos está en forma de árbol completo, puede verse en el *Anexo B*, con H=7 y K =2.

En los grafos de creación en forma de árbol completo que acabamos de presentar, el comportamiento de los programas es predecible respecto al número de creaciones realizadas por cada proceso. Para hacer variar de manera indeterminada dicho número de creaciones, se implantó un programa cuyo grafo de creación es irregular (Figura 22). Este corresponde al grafo generado en el espacio de búsqueda de la solución al problema de poner N reinas en un tablero

de tamaño NxN (sin que se coman entre ellas). El código fuente de este programa, para N=7 puede verse en el *Anexo C*.

- **Irregular (Reinas)**

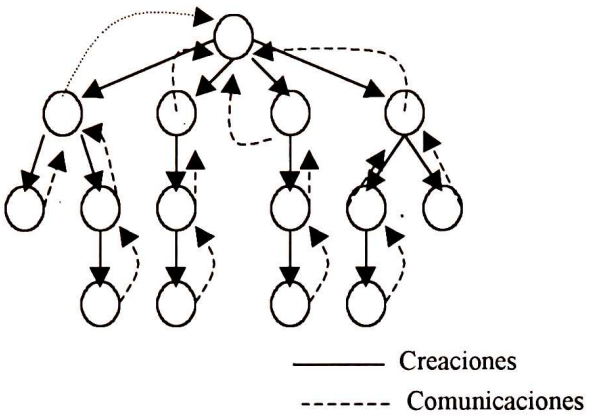


Figura 22. Grafo irregular correspondiente al espacio de soluciones en el problema de acomodar N=4 reinas en un tablero de NxN

En la Tabla 1, se describen los parámetros de algunos de los programas de prueba utilizados que poseen grafos de creación en forma de árbol lleno, presentados en las Figuras 19-a, 20 y 21.

	Núm. Hijos	Altura	Núm. Total de Procesos
1	5	5	781
2	2	7	127
3	500	2	501
4	50	2	51
5	1	500	500
6	1	20	20

Tabla 1. Grafos de creación en forma de árbol lleno

En la siguiente Tabla 2 se detallan los parámetros de dos de los programas utilizados con grafos de creación en forma de árbol irregular (Figura 22), los cuales consideran el problema del acomodo de las reinas.

	Núm. Máx. de Hijos	Altura	Núm. Total de Procesos
7	7	7	512
8	6	6	149

Tabla 2. Grafos de creación en forma de árbol irregular

Los grafos de creación de estos ocho programas, fueron tomados en cuenta para formar los dos grupos de programas de prueba en donde los procesos intercambian un solo mensaje y en donde hacen varios intercambios de mensajes. En la sección siguiente se presentan dichos grupos.

3.2 Primer Grupo de Programas de Prueba: Procesos Intercambiando un solo Mensaje

Este primer grupo está compuesto de programas en los cuales la paralelización de los cálculos es importante. Para cada programa, los procesos creados hacen una cierta cantidad de cálculos antes de enviar un mensaje de terminación al proceso que los creó. En la figura 23 se muestra un ejemplo de grafo de creaciones (árbol lleno de altura 3) y de comunicaciones (un mensaje de terminación del hijo al padre) de un programa perteneciente a este primer grupo:

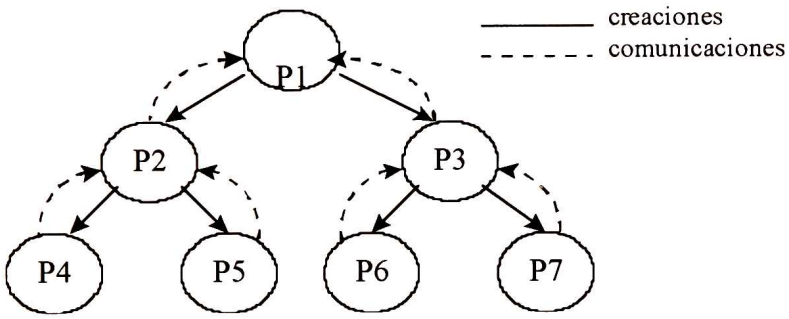


Figura 23. Ejemplo de un grafo de creaciones y comunicaciones del primer grupo de programas

Un parámetro adicional que se aplica a los programas de prueba de la Tabla 1 y la Tabla 2, es la cantidad de cálculos realizados por los procesos padres mientras esperan el mensaje de terminación por parte de sus hijos. Este manejo se detalla a continuación.

En los programas usados, la cantidad de cálculos efectuada por cada proceso, puede determinarse mediante alguna de las tres maneras siguientes:

a) IGUAL (I): Todos hacen la misma cantidad de cálculos.

b) DECRECIENTE (D): Los procesos que están en el nivel X del árbol de creaciones, hacen más cálculos respecto a los que están en el nivel X+1. Para realizar esto, se estableció una cantidad de cálculos que llamamos "decremento". Esta cantidad es sustraída de los cálculos realizados por un proceso, para determinar los cálculos que realizarán sus hijos.

c) SIEMPRE TRABAJAN (S): Los procesos realizan repetidamente una cantidad fija de cálculos, hasta que reciben el mensaje de terminación de los procesos creados (sus hijos), es decir, trabajan siempre. En esta implantación se hace uso de recepciones no bloqueantes debido a que si fueran recepciones bloqueantes no podría hacer cálculos hasta que llegara un mensaje.

La siguiente Tabla muestra la forma en que se consideraron los cálculos para los programas ejecutados.

	I(iguales)		D(decreciente)		S(siempre trabajan)	
Cálculos	1	1000	3	10000	5	1000
	2	500	4	500	6	500

Tabla 3. Tipos y Números de Cálculos.

En resumen, este primer grupo de programas de prueba está formado de (8 x 6) 48 programas con diferentes características en cantidad de cálculos y en grafo de creación de procesos.

Detallando los 48 programas, se notan tres subgrupos, cada uno compuesto de 16 programas que se describen a continuación

Programas	Núm. Hijos	Altura	Cálculos
1	500	2	Decrecientes 10000
2	50	2	
3	5	5	
4	2	7	
5	1	500	
6	1	20	
7	7	7	
8	6	6	
9	500	2	Decrecientes 500
10	50	2	
11	5	5	
12	2	7	
13	1	500	
14	1	20	
15	7	7	
16	6	6	

Tabla 4. Grafos de creación de procesos en forma de árboles llenos e irregulares con cálculos decrecientes

Programas	Núm. Hijos	Altura	Cálculos
17	500	2	
18	50	2	
19	5	5	
20	2	7	
21	1	500	Iguales
22	1	20	10000
23	7	7	
24	6	6	
25	500	2	
26	50	2	
27	5	5	
28	2	7	
29	1	500	Iguales
30	1	20	500
31	7	7	
32	6	6	

Tabla 5. Grafos de creación de procesos en forma de árboles llenos e irregulares con cálculos iguales.

Programas	Núm. Hijos	Altura	Cálculos
33	500	2	
34	50	2	
35	5	5	
36	2	7	
37	1	500	Siempre trabajan
38	1	20	1000
39	7	7	
40	6	6	
41	500	2	
42	50	2	
43	5	5	
44	2	7	
45	1	500	Siempre trabajan
46	1	20	500
47	7	7	
48	6	6	

Tabla 6. Grafos de creación de procesos en forma de árboles llenos e irregulares y en donde los procesos siempre trabajan

3.3 Segundo Grupo de Programas de Prueba: Procesos Intercambiando Varios Mensajes

Si los procesos de un programa paralelo intercambian datos frecuentemente y si están situados en nodos muy alejados en el sistema, el tiempo de ejecución del programa puede ser retardado. Esto es una consecuencia de la distancia y de la sobrecarga de los canales de comunicación utilizados para hacer llegar el mensaje del nodo fuente al nodo destino. Con los programas que

pertenecen a este segundo grupo generamos carga en los canales de comunicación de dos maneras: haciendo variar el volumen de datos transmitidos y haciendo variar el número de mensajes intercambiados entre dos procesos.

Los programas de este grupo también poseen un grafo de creaciones en forma de árbol completo y de árbol irregular. La diferencia entre los dos grupos es el número de comunicaciones (intercambio de mensajes) efectuados entre procesos. Las comunicaciones en los programas pueden hacerse, ya sea entre un proceso padre y sus hijos (como se muestra en la Figura 24), o bien, entre procesos hermanos (Figura 25). Una cantidad de cálculos es realizada por los procesos entre cada envío y recepción de mensajes.

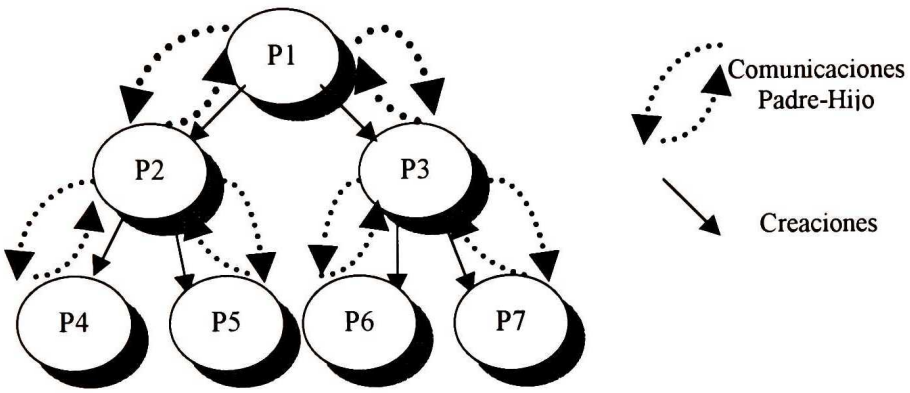


Figura 24. Ejemplo de un grafo de creaciones y comunicaciones Padre Hijo

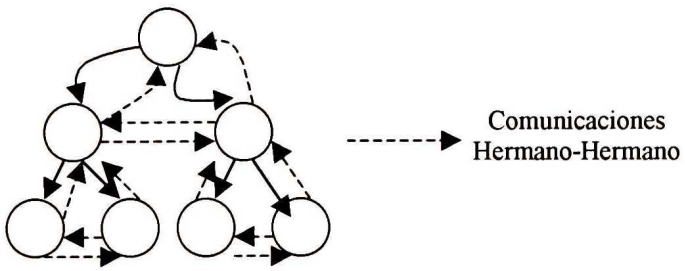


Figura 25. Ejemplo de un grafo de creaciones y comunicaciones Hermano Hermano

En la Tabla 7, se muestran las características de los programas ejecutados. Los parámetros manejados son el número de intercambio de mensajes entre dos procesos, el volumen de los datos intercambiados y la cantidad de cálculos realizados entre cada envío y recepción de mensajes.

Tipo		Núm. de Comunic.	Volumen del Mensaje	Cantidad de Cálculos
1	Padre-Hijo	1	5000	500
2		1	5000	3000
3		1	6000	500
4		1	6000	3000
5		3	5000	500
6		3	5000	3000
7		3	6000	500
8		3	6000	3000
9	Hermanos	1	5000	500
10		1	5000	3000
11		1	6000	500
12		1	6000	3000
13		3	5000	500
14		3	5000	3000
15		3	6000	500
16		3	6000	3000

Tabla 7. Características aplicables al grupo 2, donde los procesos hacen varios intercambios de datos

En resumen, el segundo grupo de programas de prueba está formado por grafos de creación de procesos en forma de árbol completo, haciendo un total de (6 x 16) 96 programas con diferentes características en cantidad de comunicaciones.

Detallando nuevamente los 96 programas, se tienen 6 subgrupos, cada uno con una forma de creación de procesos y con las 16 características de comunicación de la Tabla 7.

Tipo		Núm. de Comunic.	Volumen del Mensaje	Cantidad de Cálculos	Creación
Prog 49	Padre-Hijo	1	5000	500	4 hijos altura 7
Prog 50		1	5000	3000	
Prog 51		1	6000	500	
Prog 52		1	6000	3000	
Prog 53		3	5000	500	
Prog 54		3	5000	3000	
Prog 55		3	6000	500	
Prog 56		3	6000	3000	
Prog 57	Hermanos	1	5000	500	
Prog 58		1	5000	3000	
Prog 59		1	6000	500	
Prog 60		1	6000	3000	
Prog 61		3	5000	500	
Prog 62		3	5000	3000	
Prog 63		3	6000	500	
Prog 64		3	6000	3000	

Tabla 8. Subgrupo 1 de programas de prueba ejecutados con 4 hijos y altura 7, con las características de la tabla 7.

Tipo		Núm. de Comunic.	Volumen del Mensaje	Cantidad de Cálculos	Creación
Prog 65	Padre-Hijo	1	5000	500	2 hijos altura 7
Prog 66		1	5000	3000	
Prog 67		1	6000	500	
Prog 68		1	6000	3000	
Prog 69		3	5000	500	
Prog 70		3	5000	3000	
Prog 71		3	6000	500	
Prog 72		3	6000	3000	
Prog 73	Hermanos	1	5000	500	
Prog 74		1	5000	3000	
Prog 75		1	6000	500	
Prog 76		1	6000	3000	
Prog 77		3	5000	500	
Prog 78		3	5000	3000	
Prog 79		3	6000	500	
Prog 80		3	6000	3000	

Tabla 9. Subgrupo 2 de programas de prueba ejecutados con 2 hijos y altura 7 con las características de la tabla 7.

Tipo		Núm. de Comunic.	Volumen del Mensaje	Cantidad de Cálculos	Creación
Prog 81	Padre-Hijo	1	5000	500	100 hijos altura 2
Prog 82		1	5000	3000	
Prog 83		1	6000	500	
Prog 84		1	6000	3000	
Prog 85		3	5000	500	
Prog 86		3	5000	3000	
Prog 87		3	6000	500	
Prog 88		3	6000	3000	
Prog 89	Hermanos	1	5000	500	
Prog 90		1	5000	3000	
Prog 91		1	6000	500	
Prog 92		1	6000	3000	
Prog 93		3	5000	500	
Prog 94		3	5000	3000	
Prog 95		3	6000	500	
Prog 96		3	6000	3000	

Tabla 10. Subgrupo 3 de programas de prueba ejecutados con 100 hijos y altura 2 con las características de la tabla 7.

Tipo		Núm. de Comunic.	Volumen del Mensaje	Cantidad de Cálculos	Creación
Prog 97	Padre-Hijo	1	5000	500	50 hijos altura 2
Prog 98		1	5000	3000	
Prog 99		1	6000	500	
Prog 100		1	6000	3000	
Prog 101		3	5000	500	
Prog 102		3	5000	3000	
Prog 103		3	6000	500	
Prog 104		3	6000	3000	
Prog 105	Hermanos	1	5000	500	
Prog 106		1	5000	3000	
Prog 107		1	6000	500	
Prog 108		1	6000	3000	
Prog 109		3	5000	500	
Prog 110		3	5000	3000	
Prog 111		3	6000	500	
Prog 112		3	6000	3000	

Tabla 11. Subgrupo 4 de programas de prueba ejecutados con 50 hijos y altura 2

Tipo		Núm. de Comunic.	Volumen del Mensaje	Cantidad de Cálculos	Creación
Prog 113	Padre-Hijo	1	5000	500	1 hijo altura 100
Prog 114		1	5000	3000	
Prog 115		1	6000	500	
Prog 116		1	6000	3000	
Prog 117		3	5000	500	
Prog 118		3	5000	3000	
Prog 119		3	6000	500	
Prog 120		3	6000	3000	
Prog 121	Hermanos	1	5000	500	
Prog 122		1	5000	3000	
Prog 123		1	6000	500	
Prog 124		1	6000	3000	
Prog 125		3	5000	500	
Prog 126		3	5000	3000	
Prog 127		3	6000	500	
Prog 128		3	6000	3000	

Tabla 12. Subgrupo 5 de programas de prueba ejecutados con 1 hijo y altura 100

Tipo		Núm. de Comunic.	Volumen del Mensaje	Cantidad de Cálculos	Creación
Prog 129	Padre-Hijo	1	5000	500	1 hijo altura 20
Prog 130		1	5000	3000	
Prog 131		1	6000	500	
Prog 132		1	6000	3000	
Prog 133		3	5000	500	
Prog 134		3	5000	3000	
Prog 135		3	6000	500	
Prog 136		3	6000	3000	
Prog 137	Hermanos	1	5000	500	
Prog 138		1	5000	3000	
Prog 139		1	6000	500	
Prog 140		1	6000	3000	
Prog 141		3	5000	500	
Prog 142		3	5000	3000	
Prog 143		3	6000	500	
Prog 144		3	6000	3000	

Tabla 13. Subgrupo 6 de programas de prueba ejecutados con 1 hijo y altura 20

Como se había dicho, este juego de programas de prueba nos permitió obtener una comparación del rendimiento de los algoritmos de repartición dinámica implantados en nuestra plataforma, así como un ajuste de parámetros para un mejor desempeño. En el siguiente capítulo mostraremos los resultados obtenidos.

CAPITULO 4
RESULTADOS

En este capítulo se presentan los resultados obtenidos con la plataforma desarrollada, ejecutando los programas de prueba descritos en el capítulo 3 y usando diferentes técnicas de asignación dinámica. Los resultados son separados y mostrados en 3 grupos. El primer grupo se refiere a la ejecución de programas donde los procesos intercambia un solo mensaje, el segundo presenta los resultados de ejecución, donde los procesos intercambian varios mensajes. Por último, el tercer grupo muestra la manera en que la plataforma permite ajustar parámetros de un algoritmo particular de asignación dinámica de carga, para obtener un mejor rendimiento del sistema

En todas las gráficas se compara a los programas de prueba en sus tiempos de ejecución, obtenidos para los 5 diferentes algoritmos de asignación dinámica de procesos presentados en la sección 1.4.. los algoritmos centralizado, distribuido, vecinos, vectorial y cíclico.

4.1 Ejecución de Programas donde los Procesos Intercambian un Mensaje

El objetivo de esta primera parte de experimentos es el de poder comparar los rendimientos de los diferentes algoritmos de asignación implantados al ejecutar aplicaciones paralelas en donde lo más importante es la paralelización de cálculos en lugar de las comunicaciones.

En esta sección se presentan los resultados (Tiempos de Ejecución) obtenidos al ejecutar ocho programas de prueba de la Tabla 1 y la Tabla 2 (capítulo 3, sección 3.1). En dichos programas se tienen grafos de creación de procesos en forma de árbol completo y grafos de creación de procesos en forma de árbol irregular.

Estos ocho programas de prueba fueron combinados haciendo variaciones en el tipo y número de cálculos, dando un total de (8x6) 48 programas de prueba, cuyos parámetros fueron mostrados en las tablas 4, 5 y 6. Los resultados se muestran en las gráficas de las Figuras 30, 31 y 32 y se detallan a continuación

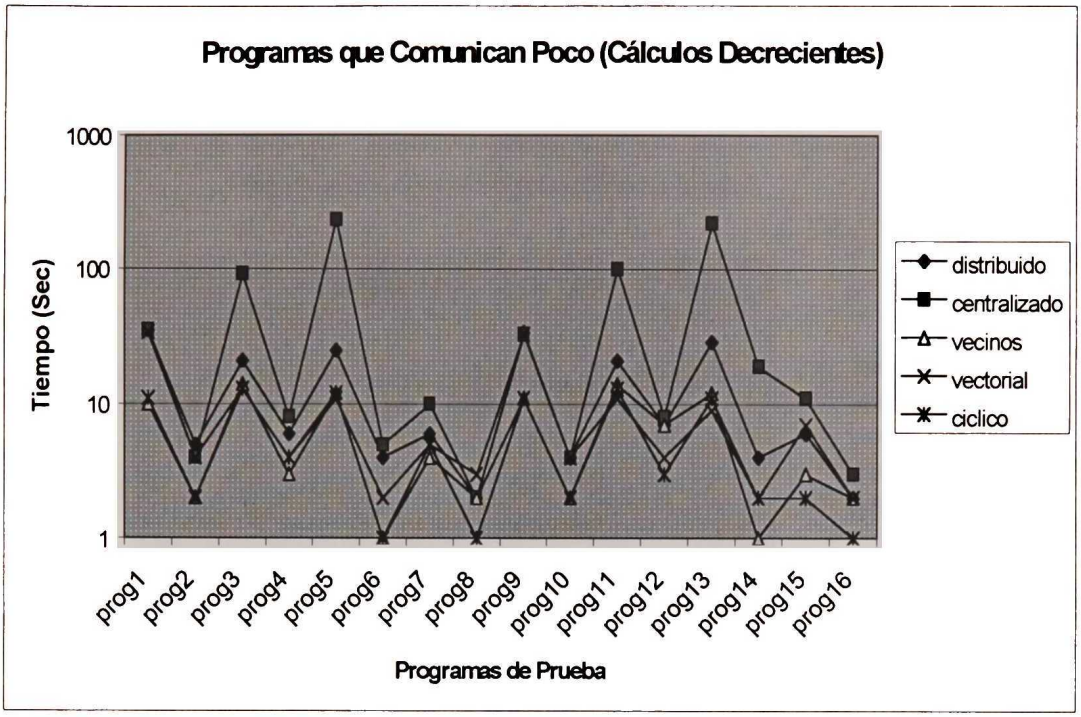


Figura 30. Tiempos de Ejecución en donde los programas comunican poco y efectúan cálculos decrecientes.

En la figura 30 se muestra el resultado obtenido al ejecutar el grupo de 16 programas de prueba de la Tabla 4 (capítulo 3 sección 3.1), este grupo se distingue porque cada proceso efectúa cálculos decrecientes dependiendo de la altura en que se encuentra el árbol de creación

En los primeros ocho programas, los procesos realizan cálculos decrecientes a partir de 10000. Del programa 9 al 16, los procesos realizan también cálculos decrecientes comenzando con 500 cálculos. En esta gráfica se observa que los mejores tiempos de ejecución obtenidos, resultaron de los algoritmos de asignación, vecinos y cíclico. Por otra parte, los peores tiempos de ejecución fueron para el algoritmo de asignación centralizado. Se considera que el algoritmo de asignación de vecinos obtiene buenos tiempos de ejecución debido al manejo de la información parcial, al igual que el algoritmo de asignación cíclico, donde no se consideran envíos de mensajes, solo se asigna al siguiente nodo. Por el contrario, los peores tiempos de ejecución en el algoritmo de asignación centralizado muy probablemente son debido a la enorme cantidad de mensajes que se tiene hacia un solo nodo que es el que contiene toda la información referente a los estados de carga.

En la figura 31, se muestran a otros 16 programas, pero ahora del grupo donde los procesos efectúan cálculos iguales, independientemente de la altura en que se encuentren (Tabla 5), con una variación de 10000 cálculos para los primeros 8 y de 500 para los siguientes.

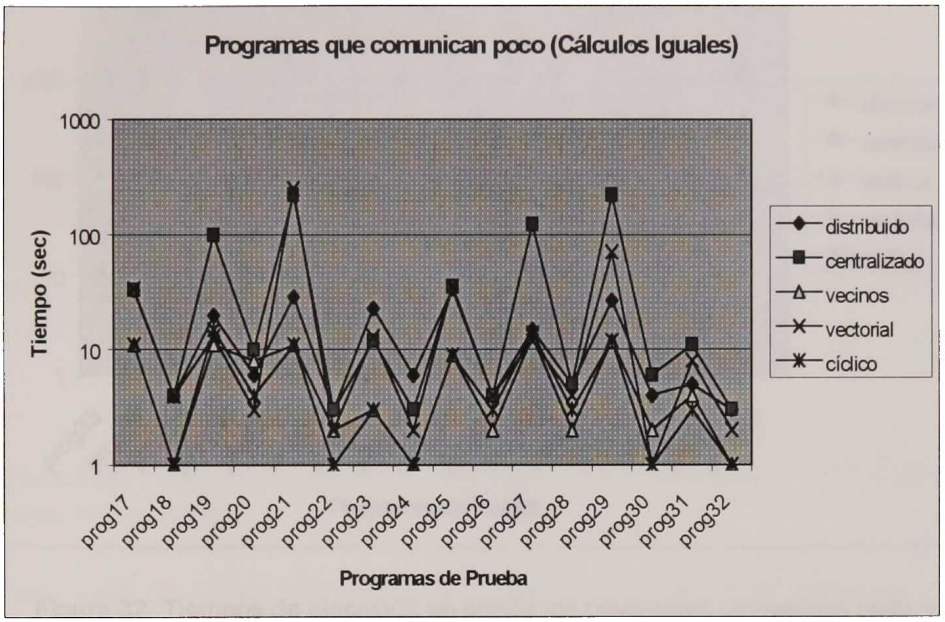


Figura 31. Tiempos de Ejecución en donde los programas comunican poco y efectúan cálculos iguales.

Nuevamente se puede notar que los mejores tiempos se obtienen de los algoritmos cíclico y de vecinos directos, a excepción del programa 20, en donde el mejor tiempo lo obtiene el algoritmo de asignación vectorial.

Por último, los programas 33 al 48 (cap. 3, tabla 6) de la Figura 32, corresponden al tipo de "siempre trabajan", es decir, realizan cálculos mientras no reciben mensaje de terminación por parte de sus hijos.

En este tipo de programas el algoritmo de asignación cíclico ya no resulta tan eficiente en cuanto a los tiempos de ejecución en todos los casos.

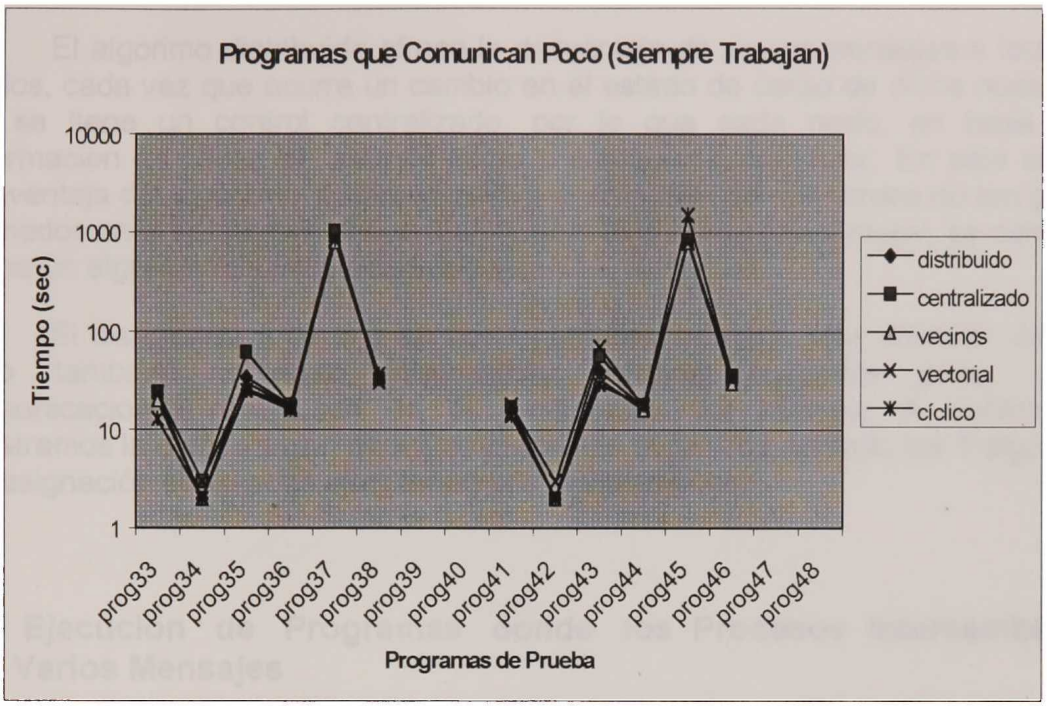


Figura 32. Tiempos de ejecución en donde los programas comunican poco y siempre trabajan

Dando una interpretación a las tres gráficas (30-32), se observa que los mejores tiempos son dados para el algoritmo de balance de vecinos directos, y en algunos casos, tambien lo es el algoritmo cíclico, a excepción del programa 45, en donde el algoritmo vectorial fue mejor. En este programa se tiene el caso de 1 hijo y una altura de 500, además de que en este tipo siempre trabajan los procesos.

Se considera que el algoritmo de balance de vecinos directos, ofrece los mejores tiempos, debido al concepto del manejo parcial de la informacion, y esto es posiblemente a que en un ambiente de información parcial, el número de mensajes es menor, además de que pudiera ser más tardado esperar por una respuesta global del procesador menos cargado en el sistema, siendo que es más rápido ejecutarlo el mismo, que mandarlo a ejecutar en el procesador vecino con menor carga.

Después los mejores tiempos los ofrecen los algorimos de balance distribuido y vectorial. Y por último, el algoritmo con control centralizado.

Se considera que el algoritmo centralizado tiene la desventaja de un manejo global de la informacion, además de que es el único capaz de asignar carga. En este caso el numero de mensajes o de peticiones hacia un solo procesador central pudieran saturarlo y por ende incrementar los tiempos de ejecución.

El algorimo distribuido ofrece la desventaja de enviar mensajes a todos los nodos, cada vez que ocurre un cambio en el estado de cargo de dicho nodo, pero no se tiene un control centralizado, por lo que cada nodo, en base a su informacion es capaz de asignar carga a cualquier procesador. En este caso la desventaja del algoritmo distribuido se ve disminuida por el numero no tan grande de nodos en el cúmulo de nueve nodos, pero aun asi no fue el mejor, se considera como un algoritmo de asignacion regular.

Si los procesos de una aplicación paralela no nada más efectúan cálculos sino también necesitan intercambiar información entre ellos, dichas comunicaciones repercuten en el rendimiento del sistema. A continuación mostramos la comparación de los rendimientos obtenidos, usando los 5 algoritmos de asignación estudiados ejecutando tales aplicaciones.

4.2 Ejecución de Programas donde los Procesos Intercambian Varios Mensajes

Para este grupo se ejecutaron un total de (6x16) 96 programas de prueba, los cuales se probaron con los mismos algoritmos de asignación dinámica y se obtuvieron los siguientes resultados.

En primer lugar se consideraron los programas descritos en la Tabla 7 del Capítulo 3. En este tipo de programas se efectúan 2 tipos de comunicaciones en los programas de prueba, estas son comunicaciones Padre-Hijo y entre Hermanos. En los programas también se realizan variaciones en cuanto al Número de Comunicaciones y al Volumen del Mensaje.

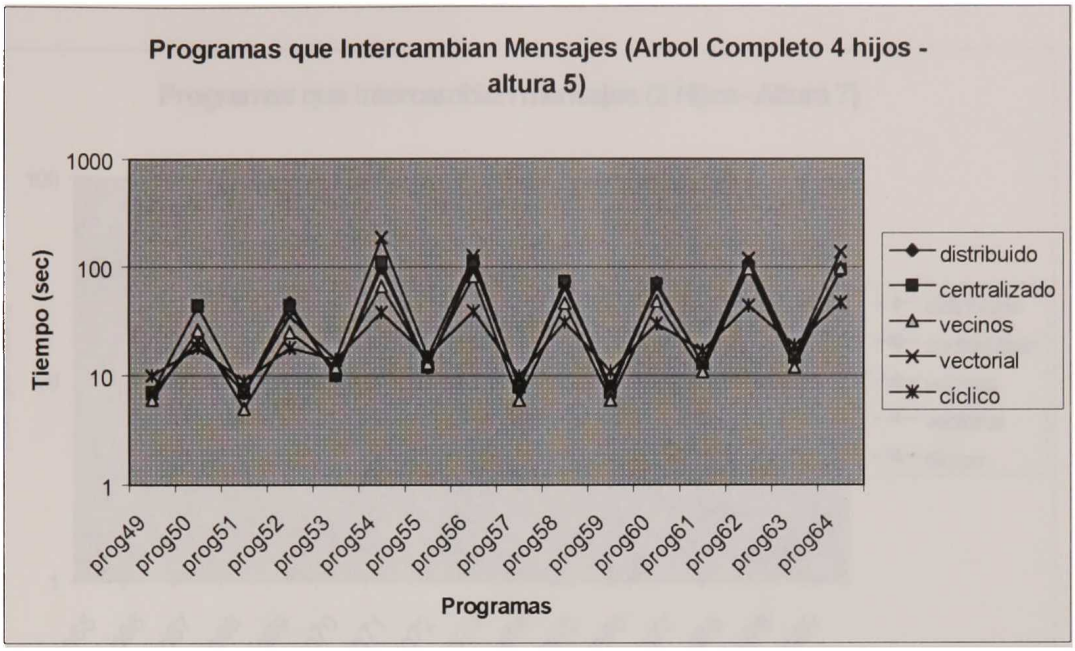


Figura 33. Tiempos de ejecución donde los programas intercambian mensajes, árbol completo de 4 hijos y altura 5.

En la figura 33, se presentan los resultados de la combinación de los 16 programas de prueba. Del programa 49 al 56 (tabla 8), son programas que intercambian mensajes entre PADRE e HIJOS, los siguientes 8 prog57-prog64 (tabla 8) intercambian mensajes con sus HERMANOS, como ya se mencionó en el capítulo anterior, las diferencias de cada bloque de 8 son las características de la Tabla 7, donde se hacen variaciones en cuanto al número de mensajes, volumen del mensaje y la cantidad de cálculos.

Analizando la figura 33 se nota que los mejores tiempos son dados por el Algoritmo de Vecinos Directos y por el Algoritmo de asignación cíclico. El algoritmo cíclico es conveniente cuando se efectúa una cantidad de cálculos mayor; mientras que cuando el número de cálculos es menor, el algoritmo asignación de vecinos directos es el más adecuado.

A continuación se presenta dentro del mismo grupo, los resultados de ejecución de los programas del 65 al 80 (tabla 9), los cuales presentan un grafo de creación en forma de árbol completo con 2 hijos y altura 7 (Figura 34).

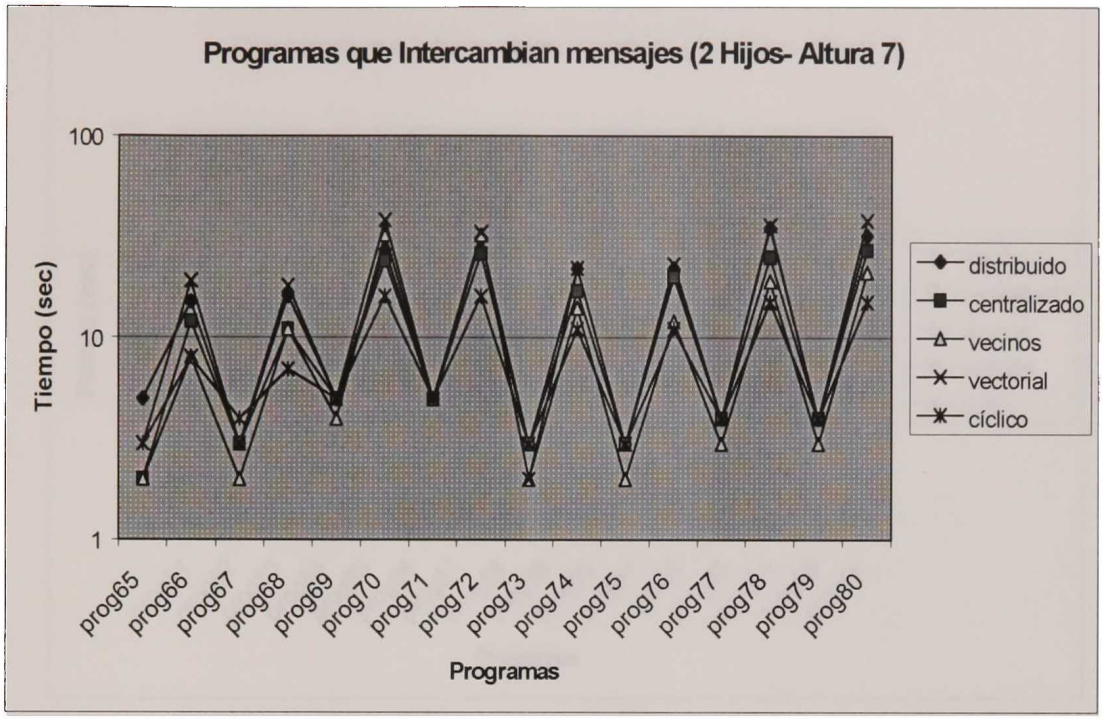


Figura 34. Tiempos de ejecución donde los programas intercambian mensajes árbol completo de 2 hijos y altura 7

Nuevamente se puede notar la tendencia de que si los tiempos son menores a 7 segundos (para este caso en especial), el algoritmo de vecinos directos es el más adecuado, pero para casos que requieren más tiempo, el algoritmo cíclico resulta ser más efectivo.

Las siguientes 2 gráficas (Figuras 35 y 36), muestran los resultados obtenidos del grupo de grafo de creación de procesos en forma de abanico (un solo padre, muchos hijos), dichos procesos efectúan varias comunicaciones.

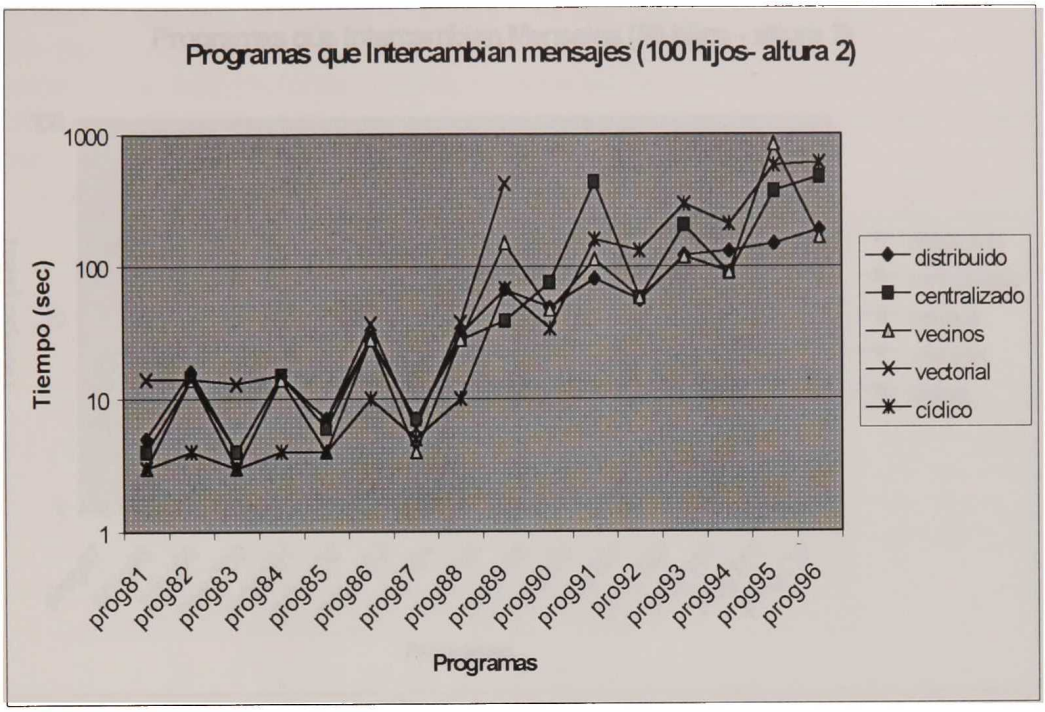


Figura 35. Tiempos de ejecución donde los programas intercambian mensajes Abanico de procesos de 100 hijos y altura 2.

La interpretación para los resultados de la figura 35, se pueden mostrar en 2 partes. En la primera parte los programas del 81 al 88 (tabla 10) efectúan comunicaciones Padre-Hijo, debido a las características del grafo de procesos, estas comunicaciones son directas. Para este tipo de programas resultó conveniente utilizar el algoritmo de asignación cíclico. Pero esto no sucede en la segunda parte prog. 89-96 (tabla 10), donde el número de comunicaciones se incrementa debido a que se trata de comunicaciones con todos los hermanos. En este caso, el algoritmo cíclico resulta inconveniente y otros algoritmos son más estables como el distribuido o el de vecinos directos. Se supone que esto es debido a que las comunicaciones costarán más tiempo de ejecución si se ejecutan estas entre todo el cluster (9 nodos), a que si se efectúan todas las comunicaciones entre unos cuantos nodos.

Un comportamiento ligeramente diferente se presenta en la siguiente figura de resultados (Figura 36), en esta se tiene el mismo tipo de creación de procesos en forma de abanico, pero a diferencia con la Figura 35, ahora se tiene una cantidad de hijos menor, la cual es la mitad del caso anterior.

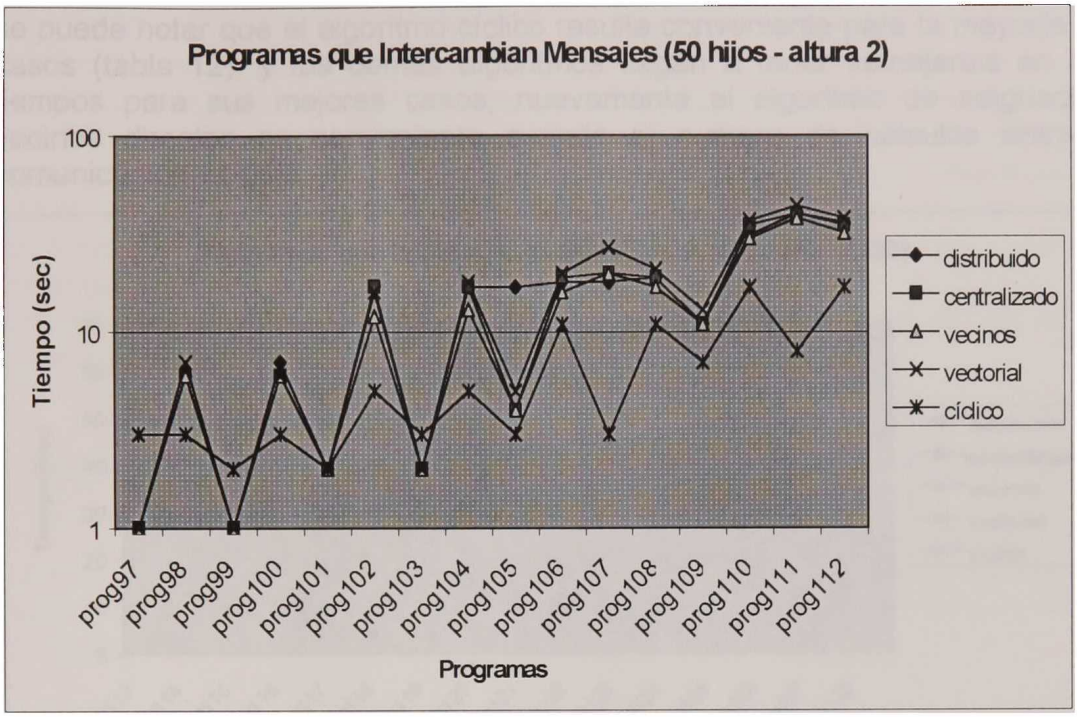


Figura 36. Tiempos de ejecución donde los programas intercambian mensajes Abanico de procesos de 50 hijos y altura 2.

En la figura 36, en los primeros 8 programas se tiene la tendencia de que los programas que efectúan pocos cálculos, resultan ser más eficientes en cualquiera de los algoritmos en lugar del algoritmo cíclico prog. 97 y 99 (tabla 11) pero la situación se revierte cuando se trata de efectuar una cantidad de cálculos mayor entre cada comunicación prog. 98 y 100 (tabla 11). Todo lo anterior es para el caso donde la comunicación es Padre-Hijos. Pero en el caso de comunicaciones Hermano-Hermano prog. 105-112 (tabla 11) el algoritmo cíclico es el más eficiente de todos, dicha situación fue diferente en la figura 35.

Por último se presentan los resultados del grupo de grafo de creación de procesos en forma de lista lineal, en este grupo se tienen los casos de 1 hijo con altura 100 (figura 37) y 1 hijo con altura 20 (Figura 38). A continuación se muestran las gráficas que describen dicho comportamiento.

Utilizando una escala no logarítmica para mejor apreciación, en la figura 37 se puede notar que el algoritmo cíclico resulta conveniente para la mayoría de los casos (tabla 12), y los demás algoritmos llegan a tener semejanza en cuanto tiempos para sus mejores casos, nuevamente el algoritmo de asignación de vecinos directos es conveniente cuando el numero de cálculos entre cada comunicación es bajo.

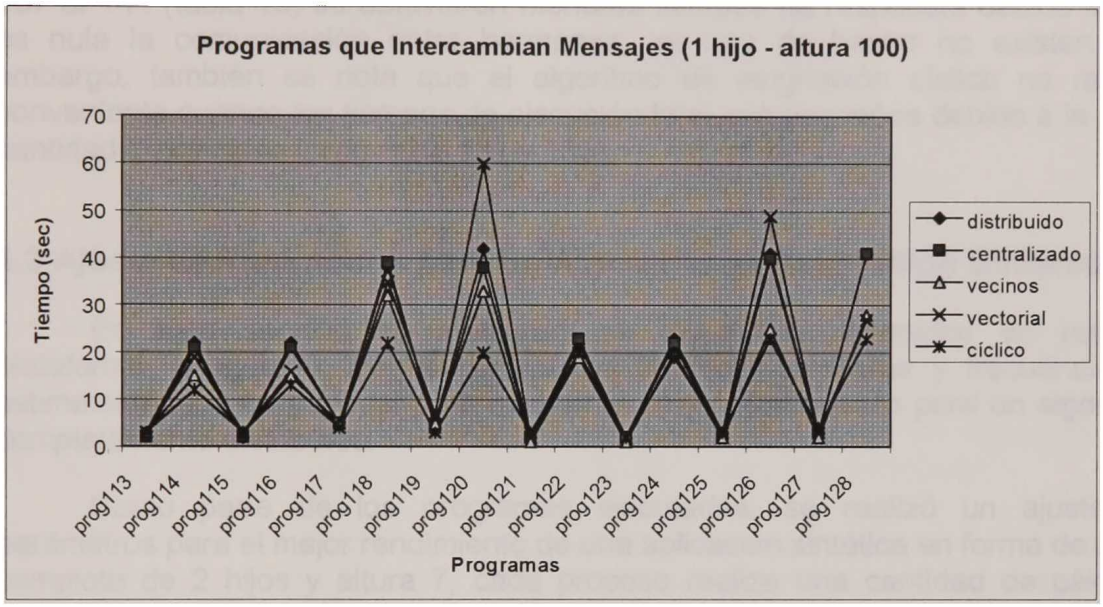


Figura 37. Tiempos de ejecución donde los programas intercambian mensajes Lista lineal de procesos de 1 hijos y altura 100.

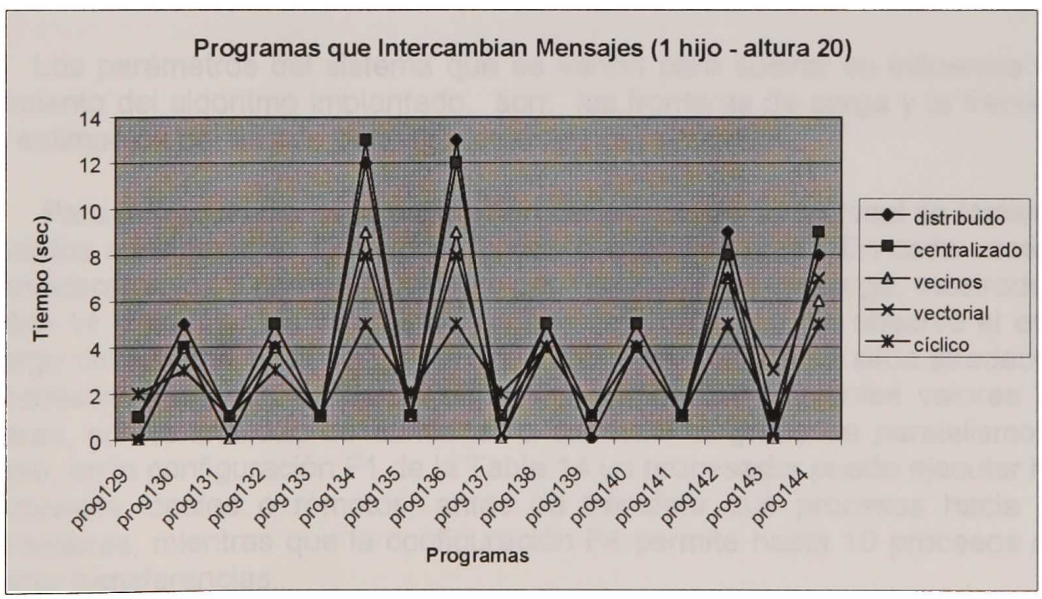


Figura 38. Tiempos de ejecución donde los programas intercambian mensajes Lista lineal de procesos de 1 hijos y altura 20.

En la figura 38, se muestran los tiempos para el grafo de creación con 1 hijo y altura 20, de aquí se pueden resaltar los programas 134 y 136 (tabla 13), donde los tiempos se incrementan para todos los algoritmos a excepción del cíclico, se considera que este comportamiento es debido a las comunicaciones Padre-Hijo que tienen que efectuar los procesos. Debido a la estructura de creación de procesos (lista lineal), en el grupo de comunicaciones Hermano-Hermano prog. 137 al 144 (tabla 13) se obtuvieron menores tiempos de respuesta debido a que es nula la comunicación entre hermanos, ya que de hecho no existen. Sin embargo, también se nota que el algoritmo de asignación cíclica no resulta conveniente cuando los tiempos de ejecución total son pequeños debido a la poca cantidad de cálculos (prog. 129, 131 y 135).

4.3 Ajuste de Parámetros para un Algoritmo de Asignación Dinámica

En esta sección se muestran los resultados obtenidos en nuestra plataforma, al ajustar ciertos parámetros fronteras de carga y frecuencia de estimación de la carga local para obtener el mejor rendimiento para un algoritmo completamente distribuido.

Como parte de los programas ejecutados, se realizó un ajuste de parámetros para el mejor rendimiento de una aplicación sintética en forma de árbol completo de 2 hijos y altura 7, cada proceso realiza una cantidad de cálculos específica y al final cada proceso envía un mensaje de terminación a su padre. La cantidad de cálculo efectuada por cada proceso, depende del nivel del árbol en que se encuentra.

Los parámetros del sistema que se varían para ilustrar su influencia en el rendimiento del algoritmo implantado, son: las fronteras de carga y la frecuencia de la estimación del estado de carga local de un procesador.

Para la frecuencia de la estimación del estado de carga local se toman tres escenarios considerando 1, 2 y 3 segundos respectivamente. En cada escenario se consideraron los 6 diferentes valores para las fronteras de carga, mostrados en la Tabla 14. Para fijar los valores de las fronteras de carga, se observó el estado de carga del sistema en ausencia de usuarios, este número se sitúa alrededor de 80 procesos. Partiendo de esta estimación, asignamos diferentes valores a las fronteras, con la finalidad de aumentar o disminuir el grado de paralelismo. Por ejemplo, en la configuración F1 de la Tabla 14 un procesador puede ejecutar hasta 20 procesos locales o remotos, antes de transferir sus procesos hacia otros procesadores, mientras que la configuración F4 permite hasta 10 procesos antes de hacer transferencias.

Configuración	Frontera 1	Frontera 2
F1	92	100
F2	92	95
F3	92	92
F4	85	90
F5	85	86
F6	80	81

Tabla 14. Configuraciones para las fronteras de carga

En primera instancia se ejecutó la aplicación sobre un solo procesador, obteniendo un tiempo de ejecución igual a 108 seg. Posteriormente se ejecutó la aplicación en forma paralela utilizando el algoritmo descrito en la sección 3. La ejecución paralela se repitió para cada par posible de valores de **frontera-frecuencia** de estimación del estado de carga local, a partir de las configuraciones de la Tabla 14. Los resultados obtenidos fueron el tiempo de ejecución y el número de procesadores involucrados en la ejecución de la aplicación.

Se calculó, para cada caso, el factor de aceleración (speed-up), dividiendo el tiempo obtenido en la ejecución sobre un solo procesador entre el tiempo obtenido en la ejecución paralela. Los resultados obtenidos se encuentran en la Figura 39. Ahí podemos observar que los parámetros para los que se obtuvo un mejor factor de aceleración, corresponden a la configuración F6 (tabla 14), con una frecuencia de estimación de carga local de 1 seg.

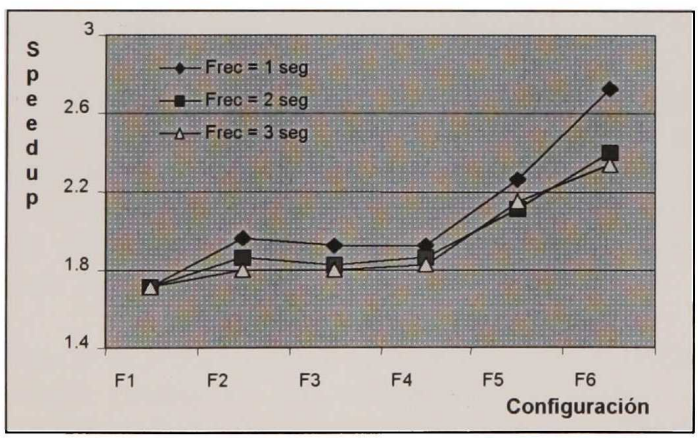


Figura 39. Factor de aceleración

En la Figura 40 se muestra el número de procesadores utilizados en cada ejecución. Notamos que la configuración F6-1seg. (tabla 14), utilizó todos los

procesadores del sistema, lo cual explica su factor de aceleración. Otra combinación de valores que obtuvo un grado de paralelismo alto fue la F6-3seg. La razón por la cual el factor de aceleración para esta combinación de valores fue menor, es que los vectores de información de los procesadores se actualizaron con menos frecuencia. Esto implica un menor costo de comunicación por parte del algoritmo de asignación pero ocasiona que los procesadores utilicen información obsoleta al decidir una transferencia.

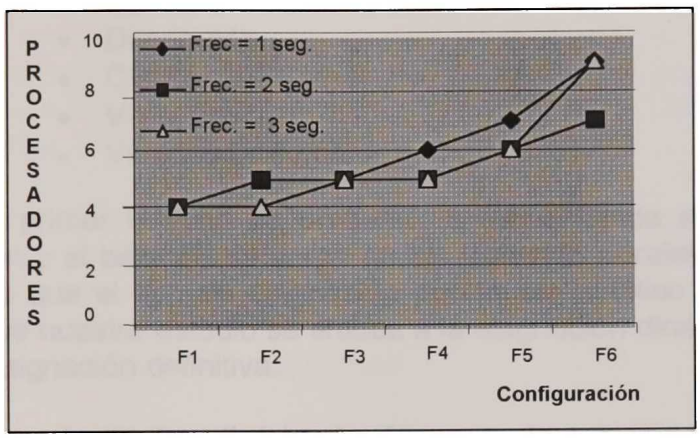


Fig. 40. Número de procesadores utilizados

CAPITULO 5
CONCLUSIONES

En esta tesis, se presentó un estudio de la importancia de los algoritmos de asignación dinámica de carga. El estudio consistió en un análisis de cinco algoritmos:

- Global Centralizado
- Distribuido
- Cíclico
- Vectorial
- Vecinos Directos.

En primer término se presentó la teoría ligada al problema de la asignación y el balance de carga en los sistemas paralelos. En esta parte se resaltó que el tipo de asignación puede ser estático o dinámico, y se acentó que nuestro estudio se enfoca a la asignación dinámica de procesos usando asignación definitiva.

Después se describió la plataforma implementada en este trabajo de tesis, así como la manera en que se implementan los 5 algoritmos estudiados.

Los programas que se ejecutaron en la plataforma de experimentación fueron programas de tipo sintéticos usando PVM. Esto debido a que un programa sintético puede modelar un grupo de aplicaciones con ciertas características como la forma de creación de procesos y el envío y recepción de mensajes.

Por último, se presentaron algunos de los resultados obtenidos mediante la comparación de los tiempos de ejecución de los programas de prueba usando los cinco algoritmos de asignación. También se mostró mediante un ejemplo la parte pragmática de la plataforma que permite ajustar los parámetros de asignación para mejorar el rendimiento de un algoritmo en particular.

Llego a la proposición y realización de una plataforma de ejecución de programas paralelos, los cuales son ejecutados en un ambiente de balance de carga a través de distintos algoritmos de asignación dinámica; pudiendo obtener una comparación de sus rendimientos. Otro punto importante es que se permitió ratificar la importancia del buen ajuste de los parámetros de la distribución de carga tales como:

- Rechazo o no rechazo de procesos
- Fronteras de carga
- Frecuencia de cuantificación

La aportación de este trabajo, es la plataforma con la cual otros trabajos pueden tener un punto de comparación para la ejecución de ciertos programas, además de que se marca el camino para estudios posteriores de nuevas técnicas de asignación dinámica de carga.

La experiencia ganada en este trabajo es la comprensión del comportamiento y de la utilidad de los algoritmos de asignación dinámica con los cuales los tiempos de ejecución para ciertas aplicaciones pueden ser reducidos considerablemente.

Como perspectivas de esta aportación, observamos que es interesante estudiar y proponer nuevos algoritmos de tipo información parcial y control distribuido para asignar la carga en un sistema, en donde se tenga una topología diferente a la de bus (que fue la utilizada) y donde se ejecuten aplicaciones no sintéticas.

ANEXOS

ANEXO A

PVM

Hablar de PVM (Parallel Virtual Machine)[3], es hablar de la herramienta que se utilizó para la implantación de los algoritmos de balance dinámico. PVM se puede definir como un software que permite a una red de máquinas heterogéneas, ser usadas como una sola y gran computadora paralela con memoria no compartida. Para de este modo resolver grandes problemas computacionales utilizando muchas y potentes computadoras.

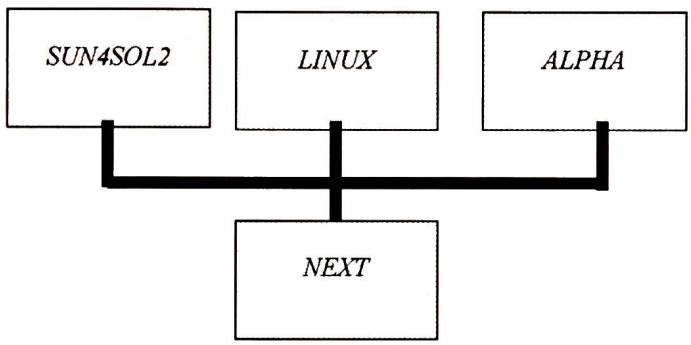


Figura 39. Ejemplo de una maquina virtual.

En la figura 39 cada caja identifica una computadora o procesador y el nombre dentro de la caja es el sistema operativo que usa. Todas las máquinas están conectadas en red.

PVM fue desarrollado a mediados de 1989 en el Oak Ridge National Laboratory (ORNL). Ahora es un proyecto de investigación conjunto que involucra a Vaidy Sunderam (Emory University), Al Geist (ORNL), Robert Mancheck (University of Tennessee), Adam Beguelin(Carnegie Melon y Pittsburgh Supercomputer Center) y Jack Dongarra(ORNL) entre otros. Cabe mencionar que, este software es gratuito y distribuido libremente en todo el mundo.

Una de las características de PVM, es que poseen funciones que automáticamente inician tareas en la máquina virtual, las cuales a su vez se comunican y se sincronizan entre ellas.

Las aplicaciones que utilizan las librerías de PVM, pueden ser escritas, en Fortran77 o en C, en especial, este último fue el compilador que se utilizó. PVM soporta la heterogeneidad de arquitecturas en su máquina virtual, esto es muy adecuado para cierto tipo de aplicaciones, las cuales pueden explotar de mejor forma la arquitectura más adecuada para su solución.

A manera de ejemplo, se citan las siguientes arquitecturas manejadas por PVM

PVM_ARCH (Nombre de la Arquitectura en PVM)	Máquina
ALPHA	DEC Alpha
BFLY	BBN Butterfly TC2000
CM5	Thinking Machines CM5
HP300	HP-9000 model 300
LINUX	80386-80686 LINUX Box
NEXT	NeXT
SGI	Silicon Graphics
SUN4SOL2	Sun 4, SPARCstation

Características de PVM

- Número Identificador de Tareas.
  Todos los procesos en PVM son representados por un identificador entero (tid). Este identificador es único en toda la máquina virtual, ellos son proporcionados por el demonio pvmd y no son asignados por el usuario.

- Control de Procesos.
  Existen en la librería de PVM rutinas que agregan y borran máquinas de la máquina virtual, así como rutinas que inician tareas y las eliminan en tiempo total de ejecución. También existen instrucciones de PVM que brindan información acerca de la configuración y tareas activas.

- Tolerancia a Fallas.
  La tolerancia a fallas en PVM, sólo se limita a eliminar un host de la máquina virtual, en caso de que esta falle. Es responsabilidad del programador considerar este caso y tomar las medidas respectivas para hacer la aplicación tolerante a fallas.

- Comunicación
  PVM propociona rutinas para el envío de mensajes empaquetados. De esta forma cualquier tarea en PVM puede enviar mensajes a cualquier tarea que se encuentre ejecutando en la máquina virtual. No hay límite y tamaño para tales mensajes. La comunicación en PVM es asíncrona bloqueante al enviar y al recibir, además de que se puede recibir mensajes de una manera no bloqueante

Implementación de PVM

Para la implementación de PVM se consideraron 3 aspectos.

- La capacidad de escalabilidad a cientos de máquinas y miles de tareas.
- La portabilidad del sistema a cualquier versión de UNIX y también a máquinas que no corrieran UNIX (MPP, Máquinas con Paso de Mensajes con muchos procesadores).
- Tolerancia a fallos para resistir fallas en el host y en la red.

La portabilidad de PVM se logra evitando el uso de un sistema operativo, o ciertas características de un lenguaje de programación. En la implementación también se decide no utilizar código multihilo en entrada y salida y en procesamiento de tareas.

En la implementación se utilizan sockets para la comunicación entre procesos internos, además de que cada host en la máquina virtual puede comunicarse o conectarse a cada uno de los demás hosts mediante el uso de protocolos IP (Transmission Control Protocol [TCP] y User Datagram Protocol [UDP]).

Con respecto a los identificadores de tarea tids (task identificator), PVM utiliza un entero de 32 bits para direccionar a las tareas y grupos de tareas dentro de la máquina virtual. Este tid es único en el ámbito de la máquina virtual.

ANEXO B

Programa de un Arbol completo con H=7 y K=2

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

#define CENTRALIZADO 0
#define TIPO_PVM     1
#define NUM_HIJOS    2
#define ALTURA_MAX   7  /*la misma que en el main*/

#define TCOM_PH          100
#define TFIN       101

#define NUM_COMUNIC  3
#define VOL_MENSAJE  5000

#define VOL_CALCULOS 3000
#define CALC_DECR    20   /*verificar que
CALC_DECR*(ALTURA_MAX-1)<VOL_CALCULOS */

#define IGUAL        500   /*tipo de cálculo/
#define DECRECIENTE  501

#define TIPO_CALC    IGUAL

/*
Uso: Envía un mensaje al repartidor de procesos, dicho repartidor es el encargado de crear él
      nuevo proceso.
Entrada: Recibe el identificador propio de la tarea y la altura actual para ser enviados al repartidor
          de procesos.
Salida: Devuelve el identificador de la nueva tarea creada.
*/
int crea_proceso(int tidpropio,int alturaa,char **argvv)
{
 int  x,cc;
 int  mitid;
 char host_name[50];
 char  cual;
 int  cual_num;
 int  donde,tida;
 int  *tid_repartidor;
 int  info[2];
 int tag,tidp,tid_hijo[1];

 info[0]=alturaa;
 info[1]=tidpropio;

 if ((tid_repartidor=(int *)malloc(20))==NULL)
```

```
    { printf("error");
      fflush(stdout);
    }

 gethostname(host_name,25);
 cual=host_name[8];

 cual_num=cual-48;

 printf("LA POSICION DEL REPARTIDOR EN APL4 ES %d \n",cual_num);
 fflush(stdout);

 *tid_repartidor=atoi(argvv[cual_num+1]);

 printf("el repartidor es %x tid propio %x  altura %x \n",*tid_repartidor,info[1],info[0]);
 fflush(stdout);

 pvm_initsend(PvmDataDefault);
 pvm_pkint(info,2,1);
 pvm_send(*tid_repartidor,1);  // mensaje al repartidor con altura y tid del padre

 cc=pvm_recv(*tid_repartidor,-1);
 pvm_bufinfo(cc,(int *)0,&tag,&tidp);
 pvm_upkint(tid_hijo,1,1);
 printf("Soy %x, envio al repartidor %x y recibo hijo %x\n",tidpropio,*tid_repartidor,tid_hijo[0]);
 fflush(stdout);
 return tid_hijo[0];

}


/*
Uso: Util para aplicaciones que requieren ejecutar cálculos entre cada envío y recepción de
       mensajes
Entrada: La altura del proceso y dependiendo si son cálculos decrecientes (a mayor altura
          menor y cantidad de cálculos) o iguales, es la cantidad de cálculos que se van a realizar.
Salida: Ninguna
*/
void haz_calculos(int nivel)
{
  int k,z,num_instruc ;

  if(TIPO_CALC == DECRECIENTE)
    num_instruc = VOL_CALCULOS -((ALTURA_MAX-nivel)*CALC_DECR );
  else num_instruc = VOL_CALCULOS ;
  printf("Hago %d calculos\n",num_instruc);
  fflush(stdout);
  for(k=0;k<num_instruc;k++)
                    for(z=0;z<num_instruc;z++);
  return;
}
```

```
main (int argc,char *argv[])
{
  int  *valor;
  int cc,tid,ptid,tid_hijos[NUM_HIJOS];
  char buf[100];
  int   suma;
  int   altura;
  int   padre,n_h,n_c,datos[VOL_MENSAJE];
  int etiqueta,tid_apl;

  printf("el Padre de apl4 es t%x \n",padre=atoi(argv[11]));
  fflush(stdout);

  if ((valor=(int *)malloc(20))==NULL)
     { printf("error");
       fflush(stdout);
     }

  *valor = atoi(argv[1]);

  if (*valor==1)          // se trata de una hoja
     {
      if (*valor != ALTURA_MAX)  // no es la raíz
       {
         for (n_c = 0; n_c < NUM_COMUNIC; n_c ++)
             {
               pvm_initsend(PvmDataDefault);
               pvm_pkint(datos,VOL_MENSAJE,1);
               pvm_send(padre,TCOM_PH);

               haz_calculos(*valor);

               cc = pvm_recv(padre,TCOM_PH);
               pvm_upkint(datos,VOL_MENSAJE,1);
             }

         strcpy(buf,argv[1]);
         pvm_initsend(PvmDataDefault);
         pvm_pkstr(buf);
         pvm_send(padre,TFIN);
       }
      else
       {
         haz_calculos(*valor);
         strcpy(buf,argv[1]);
         pvm_initsend(PvmDataDefault);
         pvm_pkstr(buf);
         pvm_send(padre,TFIN);
       }
      free(valor);
     }

   else
     {
```

```
    tid=pvm_mytid();
    altura=*valor;
    altura--;

    for(n_h = 0; n_h < NUM_HIJOS; n_h++)
         tid_hijos[n_h] = crea_proceso(tid,altura,argv);

    for (n_c = 0; n_c < NUM_COMUNIC; n_c ++)
     {
       if (*valor!= ALTURA_MAX) /*no es el proceso raíz del árbol */
         {
           pvm_initsend(PvmDataDefault);
           pvm_pkint(datos,VOL_MENSAJE,1);
           pvm_send(padre,TCOM_PH);
         }

       haz_calculos(*valor);
       for(n_h = 0; n_h < NUM_HIJOS; n_h++)
         {
           cc = pvm_recv(-1,TCOM_PH);
           pvm_upkint(datos,VOL_MENSAJE,1);
         }
        if (*valor!= ALTURA_MAX) /*no es el proceso raíz del árbol */
         {
            cc = pvm_recv(padre,TCOM_PH);
            pvm_upkint(datos,VOL_MENSAJE,1);
         }
        haz_calculos(*valor);

        for(n_h = 0; n_h < NUM_HIJOS; n_h++)
          {
            if (tid != tid_hijos[n_h])
             {
              pvm_initsend(PvmDataDefault);
              pvm_pkint(datos,VOL_MENSAJE,1);
              pvm_send(tid_hijos[n_h],TCOM_PH);
             }
          }
     }
     for(n_h = 0; n_h < NUM_HIJOS; n_h++)
      {
        cc=pvm_recv(-1,TFIN);
        pvm_upkstr(buf);
      }

     pvm_initsend(PvmDataDefault); // envía mensaje al padre
     pvm_pkstr(buf);
     pvm_send(padre,TFIN);
     free(valor);
   } //fin del else
pvm_exit();

exit(0);
}
```

ANEXO C

Programa de un Arbol Irregular con N=7

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

#define CENTRALIZADO 0
#define TIPO_PVM     1
#define NUM_MAXHIJOS   7
#define TCOM_P          100
#define TCOM_H     102
#define TFIN       101
#define TSOLUCION   110
#define TRESPUESTA  111
#define NUM_COMUNIC  0
#define VOL_MENSAJE  200
#define VOL_CALCULOS 500
#define CALC_DECR    20
#define IGUAL       500
#define DECRECIENTE  501
#define SI          502
#define NO          503
#define TIPO_CALC    IGUAL
#define CICLICO      SI



/*
Uso: Envía un mensaje al repartidor de procesos, dicho repartidor es el encargado de crear el
      nuevo proceso.
Entrada: Recibe el identificador propio de la tarea y la altura actual para ser enviados al repartidor
           de procesos.
Salida: Devuelve el identificador de la nueva tarea creada.
*/

int crea_proceso(int tidpropio,int alturaa,char **argvv,int altura2)
{
int  x,cc;
int  mitid;
char host_name[50];
char  cual;
int  cual_num;
int  donde,tida;
int  *tid_repartidor;
int  info[3];
int tag,tidp,tid_hijo[1];
info[0]=alturaa;
```

```
info[1]=tidpropio;
info[2]=altura2;

if ((tid_repartidor=(int *)malloc(20))==NULL)
   { printf("error");
     fflush(stdout);
   }

gethostname(host_name,25);
cual=host_name[8];
cual_num=cual-48;


if (CICLICO == SI)
   cual_num=9;

printf("LA POSICION DEL asigna_proceso EN APL4 ES %d \n",cual_num);
fflush(stdout);

*tid_repartidor=atoi(argvv[cual_num+1]);
printf("mi asigna proceso es %x tid propio %x  altura %x \n",*tid_repartidor,info[1],info[0]);
fflush(stdout);

pvm_initsend(PvmDataDefault);
pvm_pkint(info,3,1);
pvm_send(*tid_repartidor,1);  // mensaje al repartidor con altura y tid del padre

cc=pvm_recv(*tid_repartidor,-1);
pvm_bufinfo(cc,(int *)0,&tag,&tidp);
pvm_upkint(tid_hijo,1,1);
printf("Soy %x, envio al repartidor %x y recibo hijo %x\n",tidpropio,*tid_repartidor,tid_hijo[0]);
fflush(stdout);
return tid_hijo[0];
}



/*
Uso: Util para aplicaciones que requieren ejecutar cálculos entre cada envío y recepción de
       mensajes
Entrada: La altura del proceso y dependiendo si son cálculos decrecientes (a mayor altura
           menor y cantidad de cálculos) o iguales, es la cantidad de cálculos que se van a realizar.
Salida: Ninguna
*/

void haz_calculos(int nivel, int ALTURA_MAX)
{  int k,z,num_instruc ;
   if(TIPO_CALC == DECRECIENTE)
      num_instruc = VOL_CALCULOS -((ALTURA_MAX-nivel)*CALC_DECR );
      else num_instruc = VOL_CALCULOS ;
   printf("Hago %d calculos\n",num_instruc);
   fflush(stdout);
   for(k=0;k<num_instruc;k++)
        for(z=0;z<num_instruc;z++);
```

```
   return;
}

int   posición_valida(int *solución, int nivel, int j)
{  int flag = 1, i;
   int resta_nivel,diag_izq,diag_der;
   for(i=0;i<nivel;i++)
    {
       resta_nivel = nivel - i;
       diag_izq = solución[i] - resta_nivel;
       diag_der = solución[i] + resta_nivel;
        if ( (j == solución[i]) || (j == diag_izq) || (j == diag_der))
              flag = 0;
    }
   return flag;
}


main (int argc,char *argv[])
{
   int  *valor;
   int cc,tid,ptid,tid_hijos[NUM_MAXHIJOS];
   char buf[100],resultado[NUM_MAXHIJOS*2];
   int   suma;
   int   altura,j;
   int   padre,n_h,n_c,datos[VOL_MENSAJE];
   int etiqueta,tid_apl;
   int altura2,ALTURA_MAX;
   int solución[NUM_MAXHIJOS],i,resp[1];
   int num_creaciones=0,num_soluciones=0,nivel,nivel_sigte;

   printf("el Padre de apl4 es t%x \n",padre=atoi(argv[11]));
   fflush(stdout);
   if ((valor=(int *)malloc(20))==NULL)
       { printf("error");
         fflush(stdout);
       }
   *valor = atoi(argv[1]);
   altura2=atoi(argv[12]);
   ALTURA_MAX=altura2;
   tid = pvm_mytid();
   nivel = ALTURA_MAX - *valor;
   nivel_sigte=nivel+1;
   altura = *valor -1;
   printf("Valor= %d,MAX= %d ,nivel=%d\n",*valor,ALTURA_MAX,nivel);
   fflush(stdout);
   if (*valor == ALTURA_MAX)  // es la raíz
       {  for (i=0;i< NUM_MAXHIJOS; i++)
          solución[i] = 0 ;
       }
    else
       {  cc=pvm_recv(padre,TSOLUCION);
          pvm_upkint(solucion,NUM_MAXHIJOS,1);
       }
```

```
    for(i=0;i<ALTURA_MAX;i++)
       {
          if( posicion_valida(solucion,nivel,i+1))
            {  printf("SI VALIDA=%d\n",i+1);
               fflush(stdout);
               solución[nivel] = i+1;
               if(nivel_sigte < ALTURA_MAX)
                     {   printf("NIVELSIGTE SI %d\n",nivel_sigte);
                         fflush(stdout);
                         tid_hijos[num_creaciones] = crea_proceso(tid,altura,argv,altura2);
                         pvm_initsend(PvmDataDefault);
                         pvm_pkint(solucion,NUM_MAXHIJOS,1);
                         pvm_send(tid_hijos[num_creaciones++],TSOLUCION);
                          printf("Num_creacion es=%d\n",num_creaciones);
                         fflush(stdout);
                     }
                else
                     {   resultado[0]='\0';
                         for(j=0; j< ALTURA_MAX; j++)
                              sprintf(resultado,"%s%d,",resultado,solucion[j]);
                         printf("SOLU: %s\n",resultado);
                         num_soluciones++;
                     }
             }
       } //fin del for

    for(i=0;i<num_creaciones;i++)
       {
          cc=pvm_recv(-1,TRESPUESTA);
          pvm_upkint(resp,1,1);
          num_soluciones += resp[0];
       }
    if( *valor != ALTURA_MAX)
        {  resp[0] = num_soluciones;
          pvm_initsend(PvmDataDefault);
          pvm_pkint(resp,1,1);
          pvm_send(padre,TRESPUESTA);
         }
    else
         { printf("NUM SOLUC:%d \n",num_soluciones);
          fflush(stdout);
          pvm_initsend(PvmDataDefault);
          pvm_pkstr(buf);
          pvm_send(padre,num_soluciones);
         }
    free(valor);
    pvm_exit();
    exit(0);
}
```

BIBLIOGRAFIA

[1] PC Webopedia http://webopedia.internet.com/TERM/l/load_balancing.html .

[2] Briat (J.), Kannat (S.E.) y Morel (E.).- Plateforme d'évaluation de stratégies de regulation dynamique de charge pour le systeme logique parallele PLoSys. En: RenPar'7, ed por Dekeyser (Libert et Manneback).- PIP-FPMs Mons, Belgique, May 1995.

[3] Al Geist, Adam Beguelin, Jack Dongarra., PVM 3 User's Guide and Reference Manual. http://www.csm.ornl.gov/pvm/

[4] Jerrell Watts A Practical Approach to Dinamic Load Balancing: IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 3, March 1998

[5] Barak (A.) y Shiloh (A.).- A distributed load-balancing policy for a multicomputer. Software Practice and Experience, vol. 15, no. 9, september 1985, pp.901-913.

[6] Christaller (M.).- Athapascan-0: definition et mode d'emploi.- Reporte Apache no. 11, Grenoble, IMAG- equipe APACHE, Juin 94.

[7] Dowaji (S.) y Roucairol (C.).- Load balancing strategy and priority of tasks in distributed environments. In : Fourtheenth Annual IEEE Conference- International Phoenix Conference on Computers and Communications, pp. 15-22. - Scottsdale, Arizona(USA).

[8] Elleuch (A.), Kanawati (R.), Muntean (T.) y Talbi (E-G.).- Dynamic load balancing mechanisms for a parallel operating system kernel. En : LNCS.- Springer Verlag. Linz, Austria.

[9] Hemery (F.).- Etude de la répartition dynamique d'activités sur architectures décentralisées.- France, These de PhD,, Université des Sciences et Technologies de Lille, Juin 1994.

[10] http://nacphy.physics.orst.edu/PVM/ref/routines/pvm_spawn.html

[11] Tango Lite: A multiprocessor simulation environment.- Reporte técnico, Computer systems laboratory, Standford university, 1993.

[12] Covington (R.G.), Dwarkadas (S.), Jump (J.R.), Sinclair (J.B.) y Madala (S.).- The efficient simulation of parallel computer systems. En International Journal in Computer Simulation, pp. 31-58.

[13] Poplawski (D.A.).- Synthetic models of distributed memory parallel programs.- Reporte Técnico ORNL/Tm-11634, Oak Ridge, Tenesse 37831-USA, Oak Ridge National Laboratory- Martin Marietta, 1990.

[14] Kitajima (J.P.).- Modeles quantitatifs d'algorithmes paralleles.- France, These de PhD, Institut National Politechnique de Grenoble, Octobre 1994.

[15] Rakotoarisoa (H.) y Mussi(P).- PARSEVAL: PARallelisation sur Réseaux de Transputers de Simulations pour l'EVALuation de performances.- Reporte Técnico no. 131, Sophia Antipolis, France, Unité de recherche INRIA-SOPHIA ANTIPOLIS, Septembre 1991.

[16] http://nacphy.physics.orst.edu/PVM/ref/routines/pvm_mcast.html

[17] Schnekenburger (T.), Friedrich (M.), Weininger (A.) y Schoen (T.).- Parsim: A tool for the analysis of parallel and distributed programs. En: International Conference on Parallel Processing, pp. 689-700.- Lyon.

[18] Stromboni (J.P.) y Kwiatkowski(L.).- Conception d'un modele pour l'analyse du parallélisme. En: OPOPAC. Actes des Journées Internationales sur les Problemes Fondamentaux de l'Informatique Parallele et Distribuée, ed. por Lavallée I. (Parker Y.), pp.171-185.

[19] Román-Alonso (G.).- "Contribution à l'étude des algorithmes d'allocation dynamique sur des machines MIMD à mémoire distribuée" Tesis doctoral. Université de Technologie de Compiegne, France.

[20] Yan (J.) Sarukkai (S.) y Mehra (P.).- Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. Software- Practice and Experience, vol. 25, no.4, April 1995, pp.429-461.

[21] Zhou (S.).- A trace-driven simulation study of dynamic load balancing. IEEE Transactions on Software Engineering, vol. 14, no. 9, September 1988, pp. 1327-1341.

[22] Corrada (A.) Leonardi (L.).- Diffusive Load-Balancing Policies for Dynamic Applications. IEEE Concurrency, January-March 1999

[23] Jacob (J.) Lee (S.).- Task Spreading and Shrinking on Multiprocessor Systems and Networks of Workstations. IEEE Transactions on Parallel and Distributed Systems, Vol.10,No.10, October 1999.

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará el Ing. Miguel Alfonso Castro García, declaramos que hemos revisado la tesis titulada:

"Balance Dinámico de carga en un Sistema Paralelo con Memoria Distribuida"

y consideramos que cumple con los requisitos para obtener el grado de Maestro en Ciencias, con especialidad en Ingeniería Eléctrica.

Atentamente,

Dr. Adriano de Luca Pennacchia

Dra. Graciela Román Alonso

Dr. Luis Gerardo de la Fraga

Dr. Sergio V. Chapa Vergara