UNIVERSIDAD AUTÓNOMA METROPOLITANA

ALGORITMOS DE BALANCE DE CARGA CON MANEJO DE INFORMACIÓN PARCIAL

Para obtener el grado de

MAESTRO EN CIENCIAS

(Ciencias y Tecnologías de la Información)

PRESENTA

Ing. Juan Santana Santana

Asesores

Dr. Manuel Aguilar Cornejo

Dr. Miguel Alfonso Castro García

Sinodales

Presidente: Dr. Ricardo Marcelín Jiménez

Secretario: Dr. Miguel Alfonso Castro García

Vocal: Dr. José Oscar Olmedo Aguirre

México - D.F.

Enero de 2010

Resumen

El balance de carga es un elemento esencial del cómputo paralelo, tiene por objetivo mantener el equilibrio del volumen de carga entre los procesadores a tiempo de ejecución, para reducir el tiempo de respuesta de las aplicaciones. Actualmente existen algunas propuestas para realizar balance de carga en aplicaciones paralelas, sin embargo, la mayoría manejan información global, es decir, implica el uso de una gran cantidad de mensajes provocando que la reducción del tiempo de respuesta no siempre sea la esperada, además de presentar problemas de escalabilidad. En esta tesis proponemos dos algoritmos de balance carga con manejo de información parcial cuyas comunicaciones siguen una topología de toroide y de árbol binario. El objetivo de los algoritmos, es reducir de mejor forma el tiempo de respuesta y el problema de escalabilidad. En los experimentos realizados para evaluar el desempeño de los algoritmos se utilizaron dos aplicaciones (estática y dinámica), la primera permite realizar multiplicación de matrices y la segunda resuelve el problema de las N-Reinas. Los resultados obtenidos muestran que cuando las aplicaciones utilizan los algoritmos propuestos, el tiempo de respuesta y el problema de escalabilidad se reducen.

Agradecimientos

A Dios por llenar mi vida de dicha y bendiciones.

A la Universidad Autónoma Metropolitana Iztapalapa y al Consejo Nacional de Ciencia y Tecnología CONACYT por el apoyo brindado para la realización de mis estudios.

Al departamento de Ingeniería Eléctrica y al Laboratorio de Supercómputo y Visualización en Paralelo, por las facilidades otorgadas durante mis estudios y en especial a la comisión de la Maestría por confiar en que podía sacar este proyecto adelante.

A mis Asesores, Dr. Manuel Aguilar Cornejo y Dr. Miguel Alfonso Castro García, a quienes antes que mis asesores considero amigos, por sus enseñanzas, consejos y apoyo durante la realización de este proyecto.

A la Dra. Graciela Román Alonso, por todo el apoyo brindado, tanto en la realización de este proyecto como en la escritura del artículo publicado con los resultados de este trabajo.

A los profesores con quienes tuve la fortuna de tomar clase, por su disposición y ayuda brindadas.

A mis compañeros y amigos de la MCyTI, Ismael, Carlos, Benjamín, Nayeli, José Luis, Luis A., Jorge, Martín, Apolo, con quienes tuve la fortuna de convivir durante buena parte de mi estancia en la Maestría, por su confianza y apoyo.

Por último, pero no menos importante, quiero agradecer a mis Padres, Juan y Ernestina, a quienes agradezco de todo corazón por su amor, cariño y comprensión. En todo momento los llevo conmigo.

A mis Hermanos, Eli, Blanquita y Lalito, por la compañía y el apoyo que me brindan.

Contenido

Lista de Figuras ix

Lista de Tablas 1

1. Introducción 3

1.1. Cómputo paralelo 3

1.2. Justificación 5

1.3. Objetivos 6

1.4. Metodología 7

1.5. Estructura de la tesis 7

2. Balance de carga 9

2.1. Introducción 9

2.2. Algoritmos de Balance de Carga 9

2.3. Políticas de balance de carga 10

2.4. Taxonomía de los algoritmos de balance de carga 13

2.5. Herramientas de desarrollo de aplicaciones paralelas con balance de carga . . 17

2.5.1. The Zoltan Parallel Data Services Toolkit 17

2.5.2. SAMBA 20

2.5.3. DDLB 23

2.5.4. DLML 24

3. DLML **27**

3.1. Introducción 27

3.2. Algoritmo de balance de carga 28

3.3. Arquitectura de DLML 31

3.4. Programación con listas de datos en DLML 32

3.5. Interfaz de programación de DLML 34

4. Algoritmos de balance de carga con manejo de información parcial **39**

4.1. Introducción 39

4.2. Algoritmo toroide 45

4.2.1. Etapa de inicialización 45

4.2.2. Etapa de balance de carga 46

4.2.3. Etapa de búsqueda de una vista global de carga 51

4.2.4. Etapa de terminación 57

4.3. Algoritmo árbol binario 58

4.3.1. Etapa de inicialización 59

4.3.2. Etapa de balance de carga 60

4.3.3. Etapa de búsqueda de una vista global de carga 61

4.3.4. Etapa de terminación 62

5. Plataforma de experimentación y resultados **67**

5.1. Aplicaciones 67

5.1.1. Multiplicación de matrices 67

5.1.2. N-Reinas 69

5.2. Infraestructura 72

5.3. Resultados 73

5.3.1. Tiempos de respuesta en la aplicación de las N-Reinas 73

5.3.2. Distribución de carga en N-Reinas 74

5.3.3. Escalabilidad en N-Reinas 77

5.3.4. Mesajes utilizados en N-Reinas 79
5.3.5. Tiempos de respuesta en Multiplicación de Matrices 80

6. Conclusiones y trabajo a futuro **85**
6.1. Conclusiones 85
6.2. Trabajo a futuro 87

Referencias **97**

Lista de Figuras

1.1. Paralelización de una aplicación 4

2.1. Taxonomía de los algoritmos de balance de carga 14

3.1. Algoritmo de subasta global en DLML 29
3.2. Algoritmo de subasta global en DLML con terminación 30
3.3. Arquitectura DLML 31
3.4. Lista de datos en un ambiente paralelo 33

4.1. Topologías de comunicación con esquema de vecinos y global 40
4.2. Topologías lógicas de comunicación toroide (a) y árbol binario (b) 41
4.3. Secuencia de propagación de mensajes con el protocolo PI 43
4.4. Secuencia de propagación de mensajes con el protocolo PIF 44
4.5. Algoritmo de subasta para la topología toroide 47
4.6. Notificación de disponibilidad de procesamiento 48
4.7. Ejecución del algoritmo PIF en la topología toroide 52
4.8. Ejecución del algoritmo PIF modificado 54
4.9. Árbol formado en tiempo de ejecución del protocolo PIF 55
4.10. Configuración de comunicación de la topología de árbol binario 59
4.11. Recolección de resultados en la topología de árbol binario 63

5.1. Multiplicación de matrices 68
5.2. Manejo de la lista de datos en la multiplicación de matrices 68

5.3. Soluciones y árbol de búsqueda en 4 Reinas 70
5.4. Manejo de la lista de datos en la exploración del árbol de búsqueda 71
5.5. Tiempos de respuesta en aplicación N-Reinas con N=14-19 con cluster heterogéneo 74
5.6. Distribución de carga con algoritmos global, toroide y árbol binario en aplicación N-Reinas con N=16 75
5.7. Distribución de carga con algoritmos global, toroide y árbol binario en aplicación N-Reinas con N=17 76
5.8. Tiempos de respuesta y escalabilidad con aplicación N-Reinas con N=16 . . 78
5.9. Tiempos de respuesta y escalabilidad en aplicación N-Reinas con N=17 . . . 79
5.10. Mensajes utilizados en aplicación N-Reinas con N=16 80
5.11. Mensajes utilizados en aplicación N-Reinas con N=17 81
5.12. Tiempos de respuesta para multiplicación de matrices de tamaño 1000 x 1000 82
5.13. Tiempos de respuesta para multiplicación de matrices de tamaño 1200 x 1200 83

Lista de Tablas

2.1. Clasificación de los algoritmos de balance de carga 18

3.1. Funciones de señalización 34

3.2. Funciones de manipulación 35

3.3. Funciones de recopilación 37

5.1. Soluciones conocidas para el problema de las N-Reinas 69

5.2. Cluster heterogéneo Pacífico 72

5.3. Cluster homogéneo Aitzaloa 73

5.4. Tiempos de respuesta obtenidos en minutos para la aplicación N-Reinas con N=14 y N=19 73

5.5. Desviación estándar (σ) de las soluciones exploradas con N=17 77

Capítulo 1

Introducción

En este capítulo se presentan los conceptos básicos del cómputo paralelo y el balance dinámico de carga, posteriormente se presenta la problemática que motiva este trabajo de investigación, los objetivos, justificación y metodología propuesta para la realización de esta tesis.

1.1. Cómputo paralelo

Muchas de las aplicaciones que actualmente conocemos, han sido desarrolladas para ejecutarse en una sola computadora utilizando un único procesador, de esta forma las aplicaciones se modelan como series de instrucciones que posteriormente serán ejecutadas de forma secuencial. Estas aplicaciones secuenciales hasta hace algunos años, habían sido adecuadas para resolver algunos de los problemas del quehacer diario de los seres humanos en sus distintas áreas de trabajo como ciencia, educación, industria, etc. Sin embargo, han surgido nuevos problemas de mayor complejidad que implican un aumento considerable en la cantidad de cálculos requeridos para conocer su solución, provocando que la solución secuencial pueda tomar días, semanas o años.

Ante esta problemática se podría pensar en dos posibles soluciones. La primera puede ser la de disponer de una computadora con gran capacidad de procesamiento, del tal forma que sea posible ejecutar una aplicación secuencial que resuelve un problema complejo en un tiempo razonable. Sin embargo, siempre existirá alguna aplicación más compleja, la cual no

se podrá resolver en el tiempo requerido.

La segunda solución puede ser la de contar con un conjunto de computadoras que trabajen en colaboración. Este conjunto de computadoras es conocido como *cluster*, así el poder de cómputo de todas permitiría disponer de una gran capacidad de procesamiento como si se tratara de una computadora con una gran cantidad de recursos, de manera que una aplicación que resuelve un problema complejo podría ejecutarse en todas las computadoras permitiendo encontrar la solución en un tiempo adecuado. Esta segunda solución es mejor conocida como cómputo paralelo.

Pero para que una aplicación pueda ejecutarse en paralelo, se deben considerar etapas adicionales a las consideradas en el desarrollo de una aplicación secuencial, estas etapas son: particionamiento, distribución y comunicación. La etapa de particionamiento considera la división de la aplicación en diferentes tareas, de tal manera que estas puedan ejecutarse simultáneamente. Después de realizar el particionamiento será necesario distribuirlas entre los procesadores donde habrán de ejecutarse, esta etapa es la de distribución. Una vez que las tareas se han distribuido entre los procesadores, por lo general se requerirá que puedan comunicarse entre sí, esto para ejecutar operaciones de sincronización o intercambiar información como resultados parciales o el resultado final. La inclusión de estas etapas es lo que llamamos paralelización de la aplicación y se muestran en la Figura 1.1.

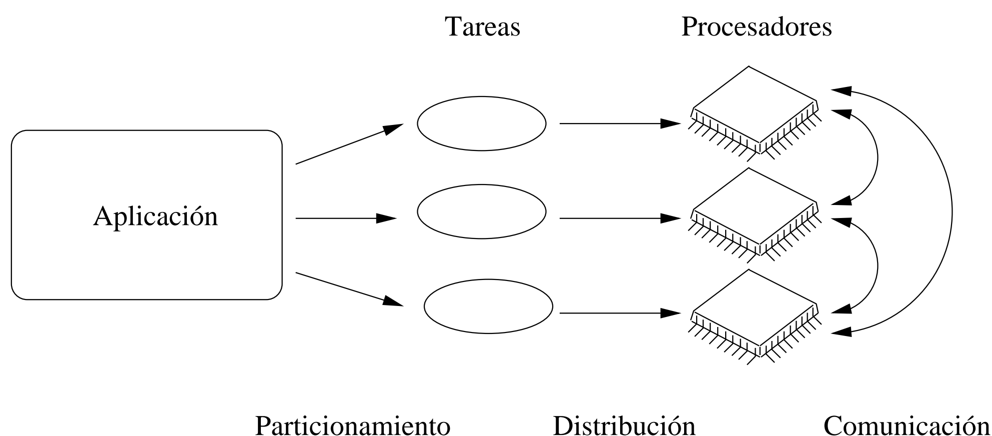


Figura 1.1: Paralelización de una aplicación

Sin embargo, el simple hecho de paralelizar una aplicación no garantiza que el tiempo de respuesta obtenido sea el mínimo, principalmente cuando se presenten los casos en los que el cluster utilizado sea heterogéneo, cluster no dedicado y/o que se trate de una aplicación con generación dinámica de datos.

El primer caso se presenta cuando el cluster está formado por procesadores de distinta capacidad, es decir que hay procesadores con mayor poder de cómputo que otros, esto ocasiona que algunas tareas finalicen su ejecución antes que otras, entonces el tiempo de respuesta de la aplicación dependerá de los procesadores con menor poder de cómputo. Por otro lado, decimos que un cluster es no dedicado cuando la infraestructura es compartida entre varios usuarios. Esto significa que varios usuarios pueden ejecutar tareas adicionales o ajenas en ciertos procesadores, provocando que el tiempo de respuesta esté en función de los nodos ocupados. Finalmente, la generación dinámica de datos puede presentarse en aplicaciones que durante su ejecución generen más datos. Dado que la cantidad de datos que se puede generar no se conoce con anticipación, es difícil mantener un equilibrio en la cantidad de datos asignados a cada procesador, esto por lo general se ve reflejado en que el tiempo de respuesta de la aplicación no es tan bueno.

Todos estos casos o inconvenientes provocan que el desempeño de una aplicación paralela se deteriore. Para evitar que el desempeño de la aplicación se vea afectado surge el balance de carga. El balance de carga consiste en mantener un equilibrio entre la carga procesada por cada procesador, para ello se requiere de algoritmos que permitan redistribuir carga a tiempo de ejecución considerando la cantidad de carga en un procesador y/o en el sistema, capacidad de procesamiento de los procesadores, costo de procesamiento de la carga y costo de comunicación.

1.2. Justificación

Actualmente existen muchos trabajos en los que se presentan estrategias que permiten balancear carga [3], [10], [14], [15], sin embargo, la mayoría de ellos requieren de conocer el estado de carga global del sistema. Esto implica que para balancear carga, se tenga que

recolectar el estado de carga de todos los procesadores que integran el sistema, requiriendo una gran cantidad de comunicaciones y por ende aumento en el tiempo de respuesta (sobre todo cuando el número de procesadores participantes es considerable). En consecuencia este tipo de estrategias presentan problemas de escalabilidad al aumentar el número de procesadores.

Nuestro trabajo propone balancear carga utilizando información parcial, es decir que el balance puede ser ejecutado sin necesidad de conocer el estado de carga de todo el sistema. Para ello se propone implementar algunas topologías de comunicación lógica que permitan organizar a los procesadores en un esquema de vecinos, en donde un procesador pueda balancear carga únicamente con los procesadores que se encuentren en su vecindad. El objetivo de utilizar información parcial es minimizar el número de comunicaciones requeridas en el balance, de esta forma se pretende reducir el problema de escalabilidad y el tiempo de respuesta de las aplicaciones.

Con el fin de verificar que el manejo de información parcial logra reducir el problema de escalabilidad y el tiempo de respuesta, se han diseñado dos algoritmos que implementan dos topologías de comunicación lógica; toroide y árbol binario. Los algoritmos fueron adaptados en la herramienta DLML [5]. DLML (Data List Management Library) es una herramienta que permite desarrollar aplicaciones paralelas mediante el uso de funciones típicas sobre listas. DLML ya incluye un algoritmo de balance de carga con manejo de información global llamado subasta, dicho algoritmo lo substituimos por los nuestros para realizar las pruebas necesarias, y comparamos el desempeño del algoritmo original contra los desarrollados en esta tesis para verificar si efectivamente se reduce el problema de escalabilidad.

1.3. Objetivos

El objetivo general de este trabajo es diseñar algoritmos de balance de carga con manejo de información parcial.

Los objetivos específicos son los siguientes:

- Diseñar algoritmos con manejo de información parcial.

- Adaptar los algoritmos para ser usados bajo la herramienta DLML como algoritmos de balance de carga.
- Probar la escalabilidad de los nuevos algoritmos bajo un escenario con diferentes patrones (tipo de aplicación y ambiente de ejecución).
- Verificar que los tiempos de respuesta de las aplicaciones que utilicen los algoritmos propuestos, disminuyen con respecto al algoritmo original de DLML.
- Comparar la distribución de carga de los algoritmos diseñados con el algoritmo que DLML incluye.

1.4. Metodología

La metodología para la realización de la tesis consistió de las siguientes etapas:

1. Revisión de literatura relacionada con los algoritmos de balance de carga.
2. Estudio del balance de carga en la herramienta DLML.
3. Elaboración y discusión de los algoritmos de balance con manejo de información parcial.
4. Adaptación de los algoritmos propuestos en la herramienta DLML.
5. Selección de aplicaciones e infraestructura de pruebas.
6. Evaluación de los resultados.

1.5. Estructura de la tesis

El resto de la tesis se encuentra organizada de la siguiente manera: en el Capítulo 2 se presentan los conceptos básicos del balance dinámico de carga y los trabajos relacionados. El Capítulo 3 está dedicado a presentar la biblioteca de programación DLML. En el Capítulo 4 presentamos la propuesta de los algoritmos de balance de carga con manejo de información

parcial, mediante la implementación de topologías de comunicación de árbol binario y toroide. En el Capítulo 5 presentamos la plataforma de experimentación así como los resultados obtenidos. Por último, en el Capítulo 6 se presentan las conclusiones y trabajo a futuro.

Capítulo 2

Balance de carga

Este capítulo está dedicado al balance de carga, a los algoritmos que implementan el balance de carga, a las políticas en las que basan su funcionamiento, taxonomía y clasificación.

Además se presentan algunas herramientas que permiten desarrollar aplicaciones paralelas en donde se incorpora uno o más algoritmos de balance de carga.

2.1. Introducción

Como ya mencionamos, el balance de carga es parte fundamental en el cómputo paralelo pues permite obtener tiempos de respuesta menores mediante la redistribución de carga. Para que una aplicación paralela tenga la posibilidad de balancear carga en tiempo de ejecución, debe contar con algún algoritmo que le permitan realizar esta tarea, y que resulte eficiente considerando las condiciones de operación tales como plataforma de ejecución y tipo de aplicación.

2.2. Algoritmos de Balance de Carga

Este tipo de algoritmos permiten redistribuir carga entre procesadores a tiempo de ejecución. El objetivo fundamental de estos algoritmos es mejorar el desempeño de una aplicación paralela. Esto implica la reducción de los tiempos de respuesta, reducción del problema de escalabilidad mediante uso eficientemente de recursos de cómputo y minimización del efecto

de factores de desbalance como el uso de un cluster heterogéneo, cluster compartido y/o generación dinámica de datos.

Para que el objetivo pueda alcanzarse, en el diseño de este tipo de algoritmos se deben tomar en cuenta aspectos como: la cantidad de carga en un procesador y/o en el sistema, poder de cómputo de los procesadores, costo de procesamiento de la carga, costo de comunicación, entre otros. Con este fin, existen políticas que facilitan la identificación de estos aspectos y ayudan a diseñar el algoritmo de balance de tal forma que sea eficiente.

2.3. Políticas de balance de carga

Las políticas de transferencia, selección, localización e información [12], determinan el comportamiento del algoritmo así como su desempeño en aplicaciones y ambientes de ejecución distintos. A continuación se describen cada una de ellas.

Política de transferencia

Esta política determina si la carga debe ser transferida a otro procesador o no. Para ello, principalmente se usan valores de umbral, los cuales son el límite que divide dos regiones de carga. Dependiendo del número de umbrales se tiene el número de regiones de carga (por ejemplo, si la política considera dos umbrales, las regiones de carga son tres: descargado, cargado y sobrecargado), así cuando el estado de carga cruza uno de estos umbrales se puede tomar la decisión de realizar o no una transferencia de carga. Las políticas de umbral se pueden clasificar en políticas de umbral absoluto y umbral relativo.

Las políticas de umbral absoluto consideran únicamente su propio estado local y no consideran el de otros nodos. Las políticas de umbral relativo además de considerar el estado local, también consideran el estado de carga del procesador destino, así un nodo puede considerarse descargado, cargado o sobrecargado si su estado local es menor, igual o mayor al del resto de los nodos respectivamente.

Política de selección

La política de selección determina que tarea (carga) deberá ser transferida. Una vez que la política de trasferencia decide que un procesador se encuentra en estado de sobrecargado, la política de selección elige una tarea para ser transferida, esta política puede ser clasificada en preferente y no preferente.

La política "preferente" permite transferir una tarea que se ha ejecutado parcialmente. Sin embargo, transferir tareas que ya han estado en ejecución conlleva a dos situaciones. La primera se presenta cuando se transfiere la tarea, en este caso también se debe transferir el estado de ejecución en el que se encontraba para que el nodo receptor pueda continuar su ejecución, implicando un costo de comunicación adicional. La otra situación se presenta cuando la tarea elegida está a punto de finalizar, en este se debe considerar si el costo de realizar la transferencia puede ser mayor al de permitir que continúe su ejecución. Por otro lado la política "no preferente" selecciona sólo las tareas que no han comenzado la ejecución, por lo que no se requiere la transferencia del estado de ejecución de la tarea, ni existe riesgo de transferir tareas que estén a punto de finalizar.

Política de localización

La política de localización tiene por objetivo encontrar algún nodo para transferir parte de la carga una vez que se ha decidido realizar la transferencia de carga. Esta política puede ser clasificada en tres categorías; selección aleatoria, selección dinámica y polling de estado.

En la selección aleatoria se elige al azar al nodo al cual se transferirá la carga (sin tomar en cuenta su estado de carga). Esta política es rápida en la elección, sin embargo, puede suceder que un nodo sobrecargado reciba más carga.

La selección dinámica permite seleccionar al nodo destino considerando el estado de carga mantenido en cada nodo, es decir cada nodo mantiene la información del estado de carga de todos los demás procesadores, esto requiere que la información almacenada en cada nodo se necesite actualizar de forma periódica. Aún cuando la elección también es rápida, existe la posibilidad de que el nodo elegido no sea el adecuado, esta situación se puede presentar cuando al momento de la elección la información del estado en el nodo no esté actualizada.

El esquema de polling consiste en consultar el estado de carga de algunos o todos los nodos al momento en que se pretende ejecutar la transferencia, así la elección se toma con base en la información recolectada. La ventaja de este esquema es que la elección del nodo por lo general es la más adecuada pero es lenta con respecto a las anteriores.

Política de información

Esta política determina cuándo la información del estado de carga de otros procesadores se debe recolectar, dónde se encuentra y qué información será considerada como indicador del estado de carga (cantidad de datos o procesos, porcentaje de uso de CPU, etc.). Las estrategias más utilizadas para implementar está política son: sin información, bajo demanda, intercambio periódico y difusión de cambio de estado.

En la primera estrategia no se maneja información del estado de carga, es decir, la elección del procesador al que se ha de realizar la transferencia es aleatoria. Con esta estrategia la elección del procesador puede no ser la más adecuada, pues existe la posibilidad de sobrecargar a algún procesador.

En el enfoque bajo demanda, la información del estado de carga de los demás procesadores se obtiene al momento de iniciar el balance de carga. De esta forma, la decisión con respecto a qué procesador transferir carga se basa en la información obtenida, permitiendo que la elección sea más adecuada.

A diferencia del enfoque anterior, en el intercambio periódico, la información es obtenida de otros procesadores de forma periódica aún y cuando no se inicie el balance de carga, sin embargo, es complicado establecer un intervalo apropiado para recolectar y actualizar la información que asegure la exactitud del estado de carga obtenido y que no represente un costo elevado en comunicación.

La difusión de cambio de estado consiste en propagar el estado de cada procesador siempre que este sufra un cambio, así cuando el balance sea ejecutado la información del estado de carga estará disponible.

Como anteriormente se mencionó, los algoritmos de balance de carga pueden estar compuestos por una o más políticas de balance que determinan su comportamiento. De acuerdo

a las políticas que utilicen estos algoritmos, es posible establecer diferentes tipos de clasificaciones. Aunque a la fecha se han propuesto varias clasificaciones, en este trabajo hemos considerado la taxonomía propuesta por Osman [18]. Con esta taxonomía es posible clasificar un gran número de algoritmos cuyo diseño esté basado en estas políticas.

2.4. Taxonomía de los algoritmos de balance de carga

La taxonomía de Osman (Figura 2.1) tiene como propósito proveer una terminología y un medio para describir y clasificar distintos algoritmos de balance que a la fecha se han propuesto. En ella se considera la existencia de cuatro estrategias principales: inicialización, localización, intercambio y selección. Estas estrategias determinan el grupo al que pertenece un algoritmo de balance de carga, en donde cada familia puede dividirse en dos o más subgrupos. A continuación se describen cada una de estas estrategias.

Iniciación

La estrategia de iniciación se encarga de especificar que mecanismo invoca las actividades de balance de carga. Basada en la política de información, se consideran dos mecanismos: periódico y bajo demanda. Cuando el mecanismo ejecutado se ejecuta de forma periódica se requiere establecer previamente un intervalo de tiempo para la ejecución del mecanismo. La iniciación bajo demanda está basada en el estado de carga local, es decir que se ejecuta cuando hay un cambio en el estado de carga y puede dividirse en: iniciada por el emisor e iniciada por el receptor. Si es iniciada por el emisor, significa que el procesador que transferirá parte de su carga se encuentra sobrecargado, si es iniciada por el receptor entonces el procesador se encuentra descargado.

Las estrategias bajo demanda generalmente tienen mejor respuesta en condiciones de desbalance, mientras que las estrategias periódicas tiene menor complejidad de implementación pero presentan un costo extra cuando la carga se encuentra balanceada.

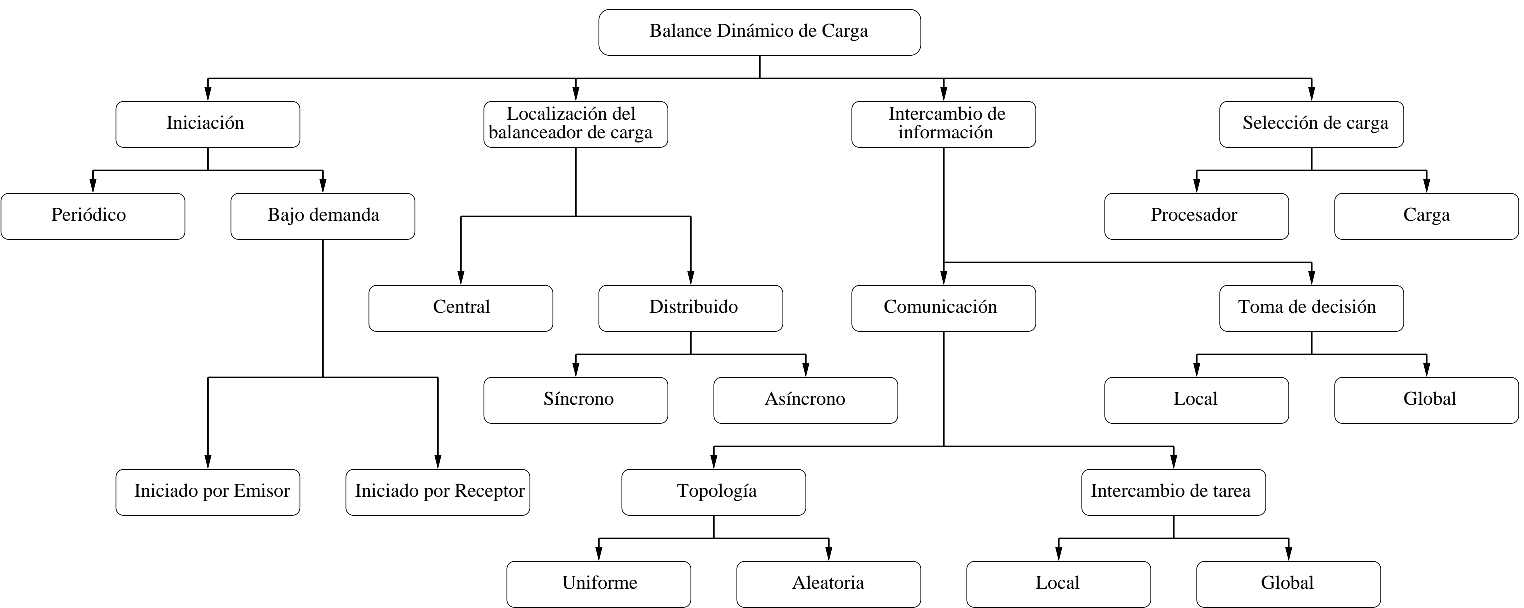


Figura 2.1: Taxonomía de los algoritmos de balance de carga.

Localización del balanceador de carga

Esta estrategia indica en donde el algoritmo de balance es ejecutado. Cuando es ejecutado por un único procesador se dice que es una estrategia centralizada, si es ejecutado por varios procesadores se dice que es distribuida. Los algoritmos distribuidos requieren que todos los procesadores se comuniquen entre sí para compartir información como su estado de carga. Estos algoritmos a su vez pueden ser clasificados en síncronos y asíncronos.

Un algoritmo síncrono se ejecuta simultáneamente en todos los procesadores participantes del sistema, de tal forma que cuando el algoritmo es invocado los procesadores detienen el procesamiento de la aplicación y realizan el balance de carga.

En el caso de un algoritmo de balance asíncrono, este puede ser ejecutado en cualquier momento por cualquier procesador sin importar si ya está siendo ejecutado por otros procesadores.

Tanto la estrategia centralizada como la distribuida pueden presentar inconvenientes. En la estrategia centralizada, un inconveniente es la existencia de un cuello de botella en el sistema, esto se debe a que un solo procesador es el encargado de ejecutar el algoritmo. En la estrategia distribuida al requerir que los procesadores se comuniquen entre sí, el costo en comunicaciones puede ser alto sobre todo cuando el número de procesadores es considerable.

Intercambio de información

Basada en la política de información, esta estrategia se encarga de especificar de dónde se obtiene la información usada para tomar la decisión de balancear carga y cuál es el flujo de la carga a través del sistema. La información usada puede ser: local, obtenida de los procesadores vecinos o de todos los procesadores del sistema. Si la información utilizada es local, el costo de comunicación es muy bajo pero por lo general la decisión tomada no es muy acertada pues sólo se considera el estado de carga local. Si la información utilizada es obtenida de los vecinos, el costo de comunicación se incrementa en función del número de vecinos pero se logra una mejor decisión. En caso de utilizar información obtenida de todos los procesadores, el costo en comunicación es el más elevado, pero la decisión tomada por lo

general es más acertada. Esta estrategia es combinada con la de comunicación, sobre todo cuando la información se obtiene de procesadores vecinos.

La estrategia de comunicación permite especificar la topología de comunicación de los procesadores en el sistema. Mediante la consideración de algunas topologías es posible tener un esquema de comunicación de vecinos sin importar el esquema de comunicación físico. Dependiendo del número de vecinos por procesador, una topología se considera uniforme o aleatoria. Se dice que una topología es uniforme cuando el número de vecinos por procesador es constante, y aleatoria en caso contrario.

Selección de carga

La estrategia permite especificar qué procesadores serán los involucrados en el intercambio de carga y qué parte de la carga es la más adecuada para ser transferida.

Como se mencionó, uno de los propósitos de la taxonomía es proveer una clasificación para los algoritmos de balance. Con este fin, se han considerado algunos algoritmos que sirven para mostrar que la taxonomía propuesta por Osman, efectivamente permite realizar una clasificación de los algoritmos independientemente de las políticas y estrategias utilizadas en su diseño. A continuación se presentan algunos algoritmos estudiados, así como una breve descripción (en la Tabla 2.1 se muestra la clasificación).

- Central, Rendez-vous y Aleatorio: Estos algoritmos fueron propuestos en [11], una de sus principales características es el patrón de comunicación que utilizan, en este caso consideran una comunicación irregular, es decir que un procesador puede comunicarse con cualquier otro sin seguir alguna topología de comunicación en particular. En el caso del algoritmo Central, su diseño está basado en las propuestas de [6] y [7].

- Tiling, X-Tiling, Rake y Vecino: En estos algoritmos [18] el patrón de comunicación considerado es regular y consiste en la formación de grupos de procesadores. El propósito de formar grupos es que un procesador pueda establecer comunicación sólo con los procesadores pertenecientes a su grupo. Para que la comunicación entre grupos se pueda

dar, se permite que un procesador pertenezca a dos grupos, de esta manera sirve como puente entre un grupo y otro.

Como se observa en la Tabla 2.1, es posible tener una clasificación de los algoritmos mediante la taxonomía propuesta, aunque por ahora sólo se consideraron algunos, cabe mencionar que no son los únicos, existe una gran cantidad. De los algoritmos presentados, algunos de ellos han sido utilizados para construir herramientas que permiten desarrollar aplicaciones paralelas con balance de carga. En la siguiente sección se presentan algunas de estas herramientas.

2.5. Herramientas de desarrollo de aplicaciones paralelas con balance de carga

En esta sección se presentan algunas herramientas que permiten desarrollar aplicaciones paralelas con balance de carga. El objetivo de estas herramientas es facilitar el proceso de desarrollo de aplicaciones paralelas a los usuarios, pero además ofrecen la posibilidad de incluir balance de carga en sus aplicaciones, esto a través de la implementación de algunos algoritmos que se encuentran disponibles en sus bibliotecas de funciones. Esto significa que los usuarios pueden dedicar mayor parte de su tiempo en diseñar la estrategia para resolver un problema en particular, que en los detalles propios de la paralelización de la aplicación y de algún algoritmo de balance en particular. Las herramientas que describimos son: Zoltan, SAMBA, DDLB y DLML.

2.5.1. The Zoltan Parallel Data Services Toolkit

Zoltan [8] es una biblioteca de programación a través de la cual se ofrecen servicios para el manejo de datos [16] en aplicaciones paralelas. Los servicios se ofrecen mediante un conjunto de utilidades entre las que se encuentran:

- Herramientas de migración de datos.

Algoritmo	Iniciación	Localización del BC	Intercambio de Información	Comunicación interprocesador	Selección de carga
Central	Periodica	Central	Global - (Global aleatoria)	Empareja procesadores sobrecargados con ociosos	Divide la carga entre el número de procesadores ociosos
Rendez-Vous	Periodica	Central	Global - (Global aleatoria)	Empareja procesadores más cargados con menos cargados	Divide la carga entre el número de parejas de procesadores
Random	Periodica	Asíncrono distribuido	Local - (Global aleatoria)	Aleatoria	Cada nueva tarea se distribuye aleatoriamente
Tilling	Periodica	Síncrono distribuido	Local - (Local uniforme)	Por cada etapa de balance se forman parejas de procesadores llamadas ventanas	La carga es distribuida entre los procesadores de la ventana
X-Tillin	Periodica	Síncrono distribuido	Local - (Global-Uniforme)	Los procesadores balancean carga conectados mediante una topología llamada hipercubo	La carga se distribuye entre los procesadores a través del hipercubo
Rake	Periodica	Síncrono distribuido	Local - (Local uniforme)	Procesadores Adyacentes	La carga sobre el promedio es transferida al procesador adyacente

Tabla 2.1: Clasificación de los algoritmos de balance de carga

- Directorios de datos distribuidos que hacen eficiente la localización de datos fuera del procesador.
- Generación de sub-listas para la distribución de datos.
- Un paquete de comunicación para simplificar la comunicación interprocesador.
- Un paquete de depuración de memoria.
- Un conjunto de herramientas de balance dinámico de carga y particionado paralelo que permiten distribuir datos sobre un conjunto de procesadores.

En relación al último punto, Zoltan considera dos clases de algoritmos para balancear carga: basados en métodos geométricos y basados en topología. La diferencia fundamental entre estas clases de algoritmos radica en la forma en que los datos son redistribuidos en los procesadores (particionado). Cada método de particionado que se utilice en cada clase, da origen a un algoritmo, de ahí que cada clase incluye varios algoritmos. Los algoritmos son los siguientes:

- Basados en métodos geométricos.
  - Bisección recursiva coordinada (RBC).
  - Bisección recursiva inercial (RIB).
  - Particionado basado en refinamiento (REFTREE).
- Basados en topología.
  - Particionado de grafo (GRAPH).
  - Particionado de Hipergrafo (HYPERGRAPH).

El propósito de ofrecer diferentes algoritmos de balance, es permitir a los usuarios elegir el algoritmo más adecuado de acuerdo a las características de su aplicación. Aún y cuando el usuario no sea capaz de decidir con exactitud el algoritmo que su aplicación requiere, el

contar con varios algoritmos abre la posibilidad de realizar comparaciones del desempeño de la aplicación del usuario con cada uno de los algoritmos, y así determinar cuál es el algoritmo más adecuado. Además de poder usar los algoritmos, también se tiene la posibilidad de modificarlos con el objetivo de mejorar su desempeño.

A manera de resumen, Zoltan ofrece varios algoritmos de balance, los cuales pueden ser optimizados para una aplicación en específico, por tal motivo los algoritmos requieren ser modificados para adaptarlos a las necesidades de cada aplicación. Esto implica que antes de que los usuarios puedan desarrollar un algoritmo, primero tenga que familiarizarse con el código fuente de la herramienta y los métodos de partición implementados.

2.5.2. SAMBA

SAMBA (Single Application, Multiple Load Balancing) [2] es una herramienta que permite el desarrollo de aplicaciones paralelas con balance de carga bajo el modelo de programación SPMD. El objetivo de la herramienta es facilitar el desarrollo de aplicaciones bajo este modelo de programación, de tal manera que permita al usuario enfocarse más en el problema a resolver, que en los detalles del balanceador, poniendo especial énfasis en facilitar la elección del algoritmo de balance de carga más adecuado para su aplicación. El diseño de la biblioteca de balance estuvo basado en los siguientes criterios de clasificación de algoritmos de balance: estático, dinámico, bajo demanda, basado en transferencia, vecinos, y orientado a eventos. Samba ofrece los siguientes algoritmos:

- Estático.
- Bajo demanda.
- Distribuido, Global, Colectivo.
- Centralizado, Global, Colectivo.
- Distribuido, Global, Individual.
- Distribuido, Local, Particionado, Colectivo.

- Distribuido, Local, Basado en vecinos, Individual.

A continuación se describe brevemente cada uno de estos algoritmos.

Estático

En este algoritmo un procesador *maestro* es el encargado de distribuir el conjunto inicial de tareas (datos) entre todos los procesadores, incluyéndose el mismo. Con este enfoque a cada procesador se le asigna el mismo número de tareas sin que haya una redistribución posterior, por lo que la carga que se asigna a un procesador únicamente se procesa por éste.

Bajo demanda

Un procesador *maestro* asigna un conjunto inicial de tareas a cada procesador. Cada vez que un procesador termina con el procesamiento de las tareas que le fueron asignadas, solicita al procesador maestro le asigne más tareas.

Distribuido, Global, Colectivo

Un procesador *maestro* distribuye el conjunto inicial de tareas en partes iguales a todos los procesadores, incluyéndose. Cuando un procesador ha procesado las tareas recibidas, este envía un mensaje a los demás procesadores pidiéndoles ejecutar balance de carga. En la etapa de balance, cada procesador envía a los demás procesadores su índice de carga interno, caracterizado por el número de tareas que se encuentran pendientes por procesar. A continuación los procesadores verifican si el índice de carga de al menos un procesador excede cierto umbral. Si esta condición se cumple cada procesador calcula el intercambio de tareas necesario para alcanzar una distribución exacta y ejecutar los intercambios en los cuales está involucrado, en caso contrario simplemente se considera que la carga del sistema es demasiado baja como para realizar balance.

Centralizado, Global, Colectivo

A diferencia del algoritmo anterior, las decisiones son tomadas por un procesador central. Cuando un procesador ha concluido el procesamiento de las tareas recibidas, envía un mensaje a los demás procesadores para pedirles que envíen su estado de carga al procesador central, después de recibir la información de todos los procesadores el procesador central decide si el balance es necesario, de ser así será el encargado de definir los intercambios necesarios e informar a cada procesador de las transferencias de tareas en las cuales está involucrado. Por último cada procesador ejecutará las transferencias indicadas.

Distribuido, Global, Individual

Este algoritmo a diferencia de los dos anteriores, tiene como meta corregir una condición de desbalance en un único procesador. Para ello, siempre que un procesador finaliza la ejecución de las tareas que recibió, envía mensajes a los demás procesadores, quienes en respuesta le envían su índice de carga. Después de recibir la información de todos, el procesador que se encuentre descargado verifica si hay al menos un procesador con carga por encima del umbral, si esto sucede, se envía una solicitud de transferencia al procesador más cargado.

Distribuido, Local, Particionado, Colectivo

Este algoritmo es muy similar al algoritmo *Distribuido, Global, Colectivo*, sólo que ahora se considera la partición de procesadores para formar grupos. De esta forma el balance de carga es ejecutado al interior de cada grupo, la formación de los grupos está definida mediante un archivo de configuración que el usuario debe crear.

Distribuido, Local, Basado en vecinos, Individual

Esta estrategia es similar al algoritmo *Distribuido, Global, Individual*, la diferencia se encuentra en que el intercambio de carga ocurre únicamente entre procesadores del mismo grupo. Los grupos son definidos mediante un esquema de vecinos, que no necesariamente representa la topología de comunicación física y en donde un procesador puede pertenecer

a más de una grupo. Nuevamente la definición de los grupos se realiza mediante un archivo de configuración. A diferencia de los algoritmos anteriores, esta estrategia requiere del uso de un algoritmo de detección de terminación, esto se debe a que los procesadores no puede determinar si continúan o finalizan su ejecución porque no conocen el estado de carga del sistema. Para resolver este inconveniente se utilizó el algoritmo de detección de terminación de Dijkstra [9].

Como se ha podido observar, SAMBA es una herramienta que pone especial énfasis en el balance dinámico de carga, provee una biblioteca con algoritmos que pueden ser utilizados según las necesidades específicas de cada aplicación.

2.5.3. DDLB

DDLB (Distributed Dynamic Load Balancing) [17] es un framework que ofrece dos estrategias de balance de carga en aplicaciones paralelas. Las estrategias de balance que DDLB ofrece son: *Transferencia de carga solicitada por el procesador cargado* y *Transferencia de carga de destino*.

A diferencia de otras herramientas, DDLB considera como carga a los procesos que se ejecutan en un procesador. En este sentido, el balance se puede realizar a través de la migración de procesos completos o mediante la división de los procesos en tareas, mismas que se distribuyen a los demás procesadores. DDLB considera está última técnica para balancear carga.

Para realizar balance de carga, en DDLB cada procesador tiene una lista de procesadores con los cuales se comunica. Además, cada procesador maneja una cola de procesos cuya longitud permite conocer su índice de carga. También se considera que cada procesador maneja dos tablas locales con información del sistema. Una tabla contiene información referente a la localización de nodos descargados, y la otra de nodos sobrecargados.

Cualquier procesador que solicite carga, es considerado un procesador descargado por el procesador que recibe la solicitud. El procesador descargado selecciona un procesador sobrecargado de su tabla (la primer entrada en la tabla) y envía un mensaje solicitando la

transferencia de carga. Inicialmente la tabla se encuentra vacía puesto que no existe información referente al estado de los procesadores, por ello la primera elección es aleatoria.

Con respecto a la actualización de la información de las tablas de nodos cargados y descargados, la arquitectura de DDLB contempla un conjunto de procesos por procesador, responsables de recolectar y actualizar periódicamente esta información

2.5.4. DLML

DLML (Data List Management Library) [5] es una biblioteca de programación mediante listas de datos que facilita el desarrollo de aplicaciones paralelas bajo el modelo de programación por paso de mensajes, y que además incluye un algoritmo de balance de carga. Con DLML los detalles relacionados al manejo de datos y el mecanismo de balance carga son transparentes para los usuarios, esto permite que el desarrollo de una aplicación paralela se facilite y que el tiempo requerido para ejecutar la aplicación se reduzca.

El algoritmo de balance que DLML ofrece es conocido como subasta y utiliza información global. La subasta con información global consiste en que cuando un procesador se queda sin carga, solicita a todos los procesadores información de su índice de carga. El procesador recibe la información y la analiza, escogiendo al procesador con más carga para solicitarle le transfiera parte de su carga.

La arquitectura de DLML contempla dos procesos para cada procesador: proceso aplicación y proceso distribuidor. EL proceso aplicación es el encargado de realizar las operaciones propias del problema que se pretenda resolver. El proceso distribuidor será el encargado de efectuar todo lo relacionado con el balance de carga.

DLML se encuentra implementada en lenguaje C y utiliza la biblioteca de paso de mensajes MPI. Los motivos por los cuales DLML ha sido elegida como plataforma de experimentación son los siguientes:

- Código fuente y documentación disponible.

- Soporte técnico por parte de sus autores.

- Algoritmo de balance de carga con manejo de información global con el cual comparar los resultados de los algoritmos propuestos.

En el siguiente capítulo se presenta una descripción más detallada de la biblioteca DLML, su arquitectura y el algoritmo de balance de carga que incluye.

Capítulo 3

DLML

En este capítulo se presenta de forma más detallada la herramienta DLML y el algoritmo de balance de carga con que DLML cuenta.

3.1. Introducción

DLML (Data List Management Library) es una biblioteca de programación que facilita el desarrollo de aplicaciones paralelas en clusters, y que además incorpora de forma transparente para el programador, un algoritmo de balance de carga. Esta biblioteca es de gran utilidad en aplicaciones en donde los datos se pueden organizar como elementos de una lista. Para acceder a los elementos de una lista, DLML utiliza funciones típicas tales como *get()* e *insert()*. Las ventajas de utilizar una lista para el manejo de los datos son las siguientes:

- Es posible insertar o eliminar elementos en cualquier parte de la lista sin considerar un orden de inserción o eliminación.
- Los elementos de las listas pueden ser procesados de forma simultánea.
- Las listas pueden ser divididas formando sub-listas, mismas que podrán ser repartidas conservando las propiedades de la lista original.

Aunque aparentemente los datos se encuentran organizados en una única lista, internamente DLML trabaja con varias listas que se encuentran distribuidas entre los procesadores

de un cluster. En efecto, esto significa que cada procesador del cluster tiene su propia lista con sus propios datos, a estas listas les llámamos listas locales. Cuando un lista local en un procesador X no tiene más elementos (lista local vacía), DLML obtiene más datos de una lista local remota no vacía localizada en un procesador Y, mediante la ejecución del algoritmo de balance de carga *subasta*. De esta forma los datos del sistema son redistribuidos entre los procesadores hasta que no haya más datos que procesar. En la siguiente sección se describe el algoritmo de balance de carga de DLML.

3.2. Algoritmo de balance de carga

Para mejorar los tiempos de respuesta, DLML incluye un algoritmo de balance de carga cuyo diseño está basado en la política de subasta [22].

El algoritmo consiste en ofrecer poder de cómputo por parte de un procesador al resto de los procesadores. La ejecución de este algoritmo inicia cuando la lista local de un procesador se queda vacía. En este punto el procesador con la lista local vacía, solicita a los demás procesadores su índice de carga (número de elementos en la lista local de cada procesador). Eventualmente el procesador recibe la información solicitada y la analiza con el fin de elegir al procesador cuyo índice de carga sea mayor, debido a que para poder realizar esta elección se requiere conocer el índice de carga de todos los procesadores, se dice que este algoritmo maneja información global. Una vez elegido al procesador más cargado, el siguiente paso es solicitarle parte de su carga.

Una explicación gráfica de la ejecución de este algoritmo se muestra en la Figura 3.1. En la Figura se considera un sistema con cinco procesadores X, Y, S, W y R (círculos). El número que aparece dentro de cada círculo representa el índice de carga del procesador. En la Figura 3.1.a el procesador X termina el procesamiento de los datos en su lista local cuya longitud es 0, bajo está condición inicia el algoritmo enviando un mensaje a todos los procesadores solicitando su índice de carga. Cuando los procesadores responden (Figura 3.1.b), X identifica al procesador más cargado, en este caso el procesador Y con 200 elementos en su lista. Posteriormente se envía un mensaje a Y para solicitar que le transfiera parte de

su carga (Figura 3.1.c). Finalmente Y responde enviando la mitad de su lista, así el proceso X y Y tendrán en sus listas 100 elementos (Figura 3.1.d).

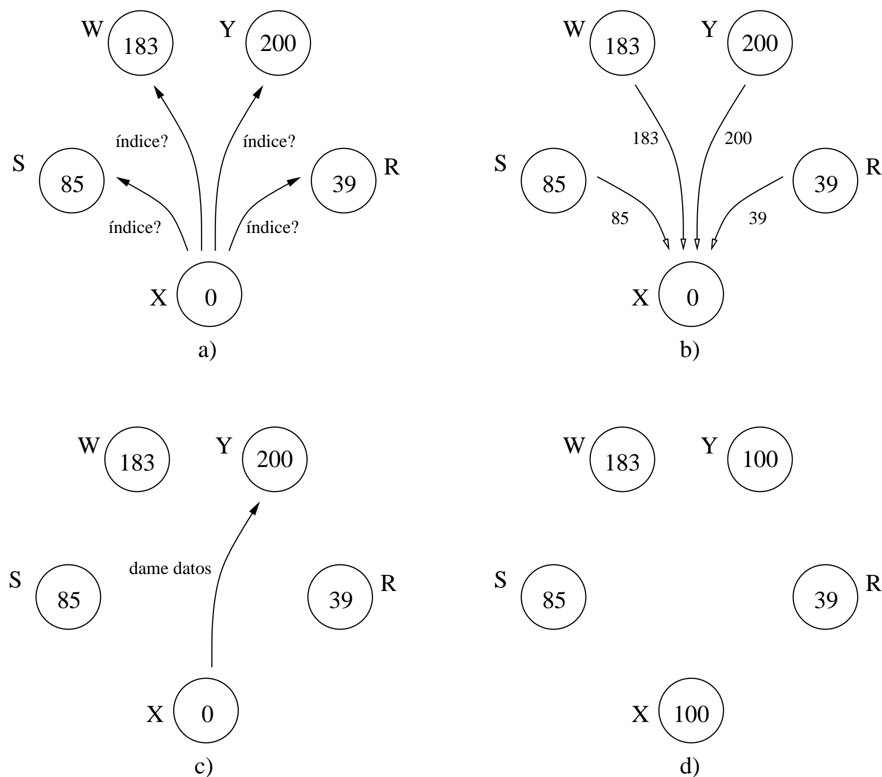


Figura 3.1: Algoritmo de subasta global en DLML

Una condición posible en la ejecución del algoritmo es cuando los índices de carga de los procesadores son *0*. Esto significa que no hay más carga en el sistema y que es posible finalizar la ejecución (Figura 3.2). En la figura los pasos a y b son iguales a los de la Figura 3.1, la diferencia se presenta en que los índices de carga ahora son *0* y ningún procesador es elegible para solicitarle carga, en este caso X envía un mensaje de *terminación* a todos los procesos DLML para indicarles que pueden finalizar su ejecución.

Las políticas de balance en este algoritmo son consideradas de la siguiente manera: la política de localización se basa en la política de información, en donde la información utilizada es obtenida globalmente. En cuanto a la política de transferencia, la distribución de carga es iniciada al momento en que un procesador se queda sin carga, respecto a quién inicia la

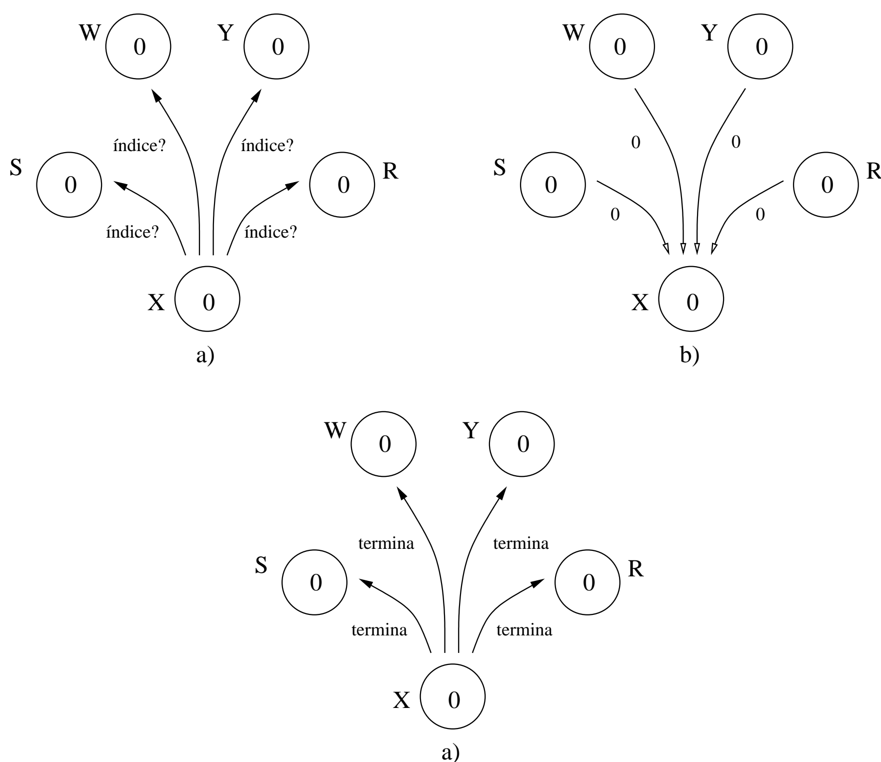


Figura 3.2: Algoritmo de subasta global en DLML con terminación

distribución, son todos los procesadores (control distribuido). Finalmente en la política de selección, lo que se considerada carga son los elementos de las listas (datos), y cuántos datos transferir, depende del número de solicitudes, así puede ser uno o varios elementos de la lista, con la única restricción de que la lista del procesador que transfiere datos no quede vacía, de esta forma se evita que los datos se transfieran de un procesador a otro sin ser procesados.

Para que una aplicación paralela desarrollada con DLML maneje los datos a través de listas y que utilice el algoritmo de balance de carga aquí descrito, se considera la siguiente arquitectura.

3.3. Arquitectura de DLML

La arquitectura de DLML consta de dos procesos por cada procesador, un proceso llamado *Aplicación* y otro proceso llamado *Balanceador* (Figura 3.3). El proceso *Aplicación* es el encargado de ejecutar el código escrito por el programador para la solución de un problema particular, y de las llamadas a las funciones básicas de listas. El proceso *Balanceador* por su parte se encarga de comunicarse con sus contrapartes (procesos balanceadores) localizados en otros procesadores cuando la política de balance requiera ejecutarse.

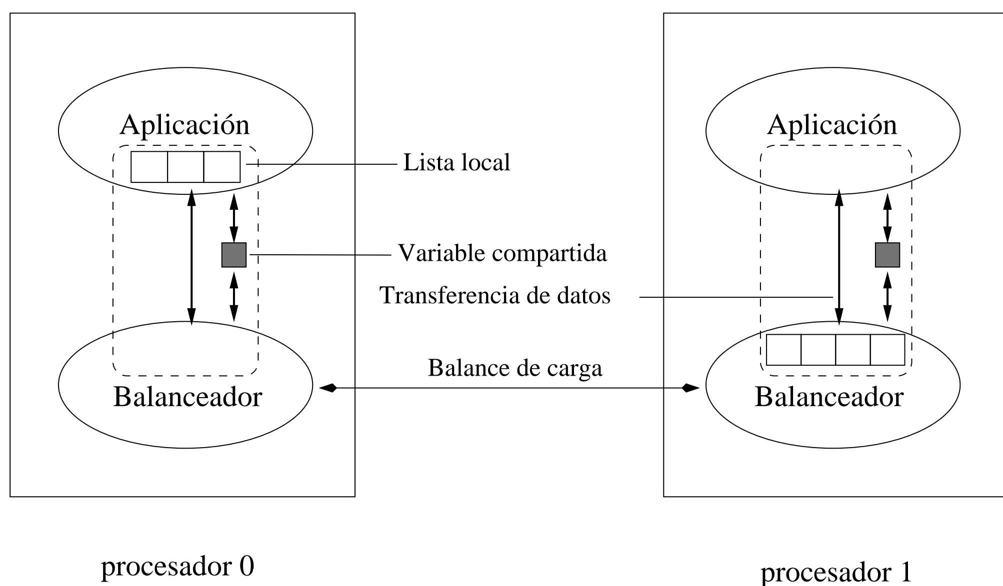


Figura 3.3: Arquitectura DLML

Los procesos *Aplicación* pueden comunicarse con su respectivo proceso *Balanceador* para intercambiar información referente a la sincronización o transferencia de datos.

Con la llamada a la función *Get*, un proceso *Aplicación* puede obtener un elemento de su lista, sin embargo, cuando la lista está vacía el proceso *Aplicación* envía un mensaje al proceso *Balanceador* para indicarle que debe obtener más datos que se encuentran en otros procesadores. A partir de este punto, el proceso *Aplicación* permanece en espera de un mensaje de respuesta del proceso *Balanceador*.

El mensaje de respuesta puede tener dos posibilidades: transferencia de los datos solicitados por el *Balanceador* y solicitud de terminación. La primera posibilidad se presenta después de que el *Balanceador* ha ejecutado el balance y ha podido obtener datos. En este caso, el *Balanceador* envía los datos obtenidos al proceso *Aplicación*. La segunda posibilidad se presenta después de ejecutar el balance y no obtener datos (esto indica que las listas de las otras aplicaciones también están vacías), ante esta situación es necesario definir una estrategia de terminación.

Cuando un proceso *Balanceador* recibe una solicitud de transferencia de carga, modifica el valor de una variable booleana compartida con su proceso *Aplicación* (1 indica que hay peticiones de transferencia de carga), de esta forma el proceso *Aplicación* detiene el procesamiento de los datos y transfiere los datos al *Balanceador*. Posteriormente el proceso *Balanceador* regresa los datos que no se hayan transferido al proceso *Aplicación* y cambia el valor de la variable compartida a 0.

La estrategia de terminación se ejecuta cuando se ha descubierto que las listas remotas no tienen más elementos, bajo esta condición, el *Balanceador* envía un mensaje a su proceso *Aplicación* indicándole que finalice su ejecución. En esta parte es necesario recolectar los resultados parciales que se encuentran distribuidos en cada procesador, con la finalidad de conocer el resultado global. Para ello uno de los procesos *Balanceador* es el encargado de recolectar estos resultados, el resto únicamente le envían su resultado parcial y posteriormente finalizan su ejecución. Cuando el proceso *Balanceador* encargado de recolectar los resultados parciales conoce el resultado global, también finaliza su ejecución.

A continuación se describe la forma en que se realiza la programación de una aplicación con listas de datos en DLML.

3.4. Programación con listas de datos en DLML

La programación de una aplicación paralela con listas de datos en DLML es similar a la programación secuencial. El código que se muestra a continuación es un programa simple que representa el modelo de programación básico de DLML. DLML sigue el modelo de pro-

gramación SPMD (Single Program, Multiple Data) [23], de esta forma el código es copiado y ejecutado en cada procesador que forma parte del cluster.

```
1. initialize(&L);
2. while ( DLML_Get(&L, item) ) {
3.   processing(item, &L);
4. }
5. final_protocol();
```

Al inicio (Línea 1), un proceso *Aplicación* puede insertar algunos elementos en su lista *L*. Posteriormente, es necesario definir un bucle principal, en el cual la función *DLML_Get* es invocada sucesivamente. A través de esta función es posible obtener elementos de la lista de datos *L* para ser procesados. El tipo de retorno de la función es booleano, *1* índica que la función fue exitosa y *0* que no hay más datos en el sistema. Dependiendo del tipo de aplicación y como consecuencia del procesamiento de un elemento (Línea 3), es posible que nuevos elementos tengan que ser agregados a la lista *L*. De esta forma cada procesador puede obtener y agregar elementos como si se tratará de una única lista.

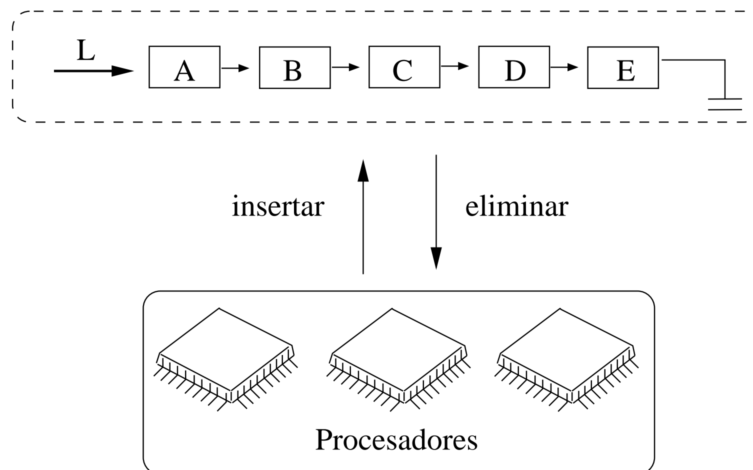


Figura 3.4: Lista de datos en un ambiente paralelo

En la Figura 3.4 varios procesadores pueden ejecutar operaciones de inserción y eliminación como si se tratará de una única lista, sin considerar algún orden y de forma simultánea.

Por último, debido a la necesidad de una estrategia de terminación en la cual se pueda conocer el resultado global, se ha implementado la función *final_protocol*.

Bajo este modelo, el desarrollo de una aplicación paralela con DLML únicamente requiere que el programador identifique los datos a ser procesados en su aplicación, los organice en una lista y maneje con las funciones definidas en la biblioteca DLML.

3.5. Interfaz de programación de DLML

La interfaz actual de DLML está integrada por un conjunto de funciones que han sido clasificadas en funciones de señalización, manipulación y recolección.

Señalización

Las funciones de señalización permiten señalizar o marcar los límites del ámbito de DLML, así como implementar mecanismos de control entre procesos (acceso a sección crítica). La tabla 3.1 muestra a las funciones pertenecientes a esta clasificación, propósito y sintaxis.

Función	Propósito	Sintaxis
DLML_Init	Inicializa listas de datos y las hace vacías.	void DLML_Init(Lista *)
DLML_Finalize	Asegura el no procesamiento de la lista y devuelve el control al sistema operativo.	void DLML_Finalize(void)
DLML_Only	Esta función permite que un solo procesador ejecute las instrucciones contenidas en ésta función y se delimita por llaves {}.	DLML_Only_One {}

Tabla 3.1: Funciones de señalización

Manipulación

Las funciones de manipulación están encargadas de manejar las listas mediante funciones como DLML_Get() y DLML_Insert(), para obtener e insertar elementos respectivamente. La Tabla 3.2 muestra las funciones pertenecientes a ésta clasificación.

Función	Propósito	Sintaxis
DLML_Length	Devuelve un entero, el cual indica la longitud de lista local.	int DLML_Length(void)
DLML_Get	Esta función recibe como parámetro dos apuntadores. El primero apunta a la lista y el segundo apunta al elemento obtenido de la lista, en consecuencia la longitud de la lista se decrementa en uno. DLML se encarga de insertar elementos en la lista local cuando queda vacía, suprimiéndolos de otras listas locales de forma transparente al programador. En caso de no existir más elementos a procesar en todo el sistema, DLML_Get devuelve un valor FALSE.	void DLML_Get(Lista *, node *)
DLML_Query	Realiza lo mismo que DLML_Get pero el elemento no es eliminado de la lista.	char DLML_Query(Lista *, node *)
DLML_Insert	Inserta localmente un elemento del tipo node en la lista apuntada por Lista.	void DLML_Insert(Lista *, node)

Tabla 3.2: Funciones de manipulación

Recopilación

El tipo de recopilación concentra a las funciones que recuperan resultados parciales. Recordar que DLML trabaja sobre listas distribuidas en todos los procesadores, por lo que en muchas ocasiones los resultados parciales se encuentran esparcidos. Las funciones que ayudan en esta recopilación se muestran en la tabla 3.3.

A través de estas funciones como parte de la interfaz de programación de DLML, es posible desarrollar aplicaciones paralelas de forma que los programadores no tengan que preocuparse por detalles propios de la sincronización y transferencia de datos entre procesos. El algoritmo de balance que DLML ofrece implementa la política de *subasta* y utiliza información global.

Como mencionábamos, la subasta consiste en ofrecer poder de cómputo que por decirlo así otros procesadores compran. El uso de información global es porque se requiere conocer el índice de carga de todos los procesadores. Esta última característica del algoritmo implica que un procesador que requiere balancear carga, se comunique con todos los procesadores para obtener esta información. En consecuencia, el número de mensajes intercambiados durante el balance depende del número de procesadores del sistema. Cuando el número de procesadores es considerable, el número de mensajes intercambiados para balancear carga resulta ser muy elevado, esto provoca un incrementando el tiempo necesario para balancear carga y en consecuencia el incremento del tiempo de respuesta de la aplicación.

Para resolver este inconveniente, se proponen dos algoritmos de balance que utilizan información parcial. En el siguiente capítulo se presentan los algoritmos propuestos.

Función	Propósito	Sintaxis
DLML_Exchange	Permite intercambiar resultados entre TODOS los procesadores como parte de la recolección de resultados.	int DLML_Exchange(void *fbuffer, int cantidad, void *dbuffer)
DLML_Reduce_Add	Realiza una reducción o envío de todos los datos parciales a un solo procesador (procesador cero). Los resultados parciales son sumados en el procesador cero.	int DLML_Reduce_Add(void *fbuffer, void *dbuffer)
DLML_Reduce_Average	Esta función realiza una reducción o envío de todos los datos parciales a un solo procesador (procesador cero). Los resultados parciales son promediados.	int DLML_Average(void *fbuffer, void *dbuffer)
DLML_Reduce_Max	Esta función realiza una reducción o envío de todos los datos parciales a un solo procesador (procesador cero). La función devuelve el valor máximo de todos los envíos.	int DLML_Max(void *fbuffer, void *dbuffer)

Tabla 3.3: Funciones de recopilación

Capítulo 4

Algoritmos de balance de carga con manejo de información parcial

4.1. Introducción

En este capítulo presentamos nuestra propuesta de algoritmos para balancear carga con manejo de información parcial. El objetivo de manejar información parcial es reducir el número de mensajes y con ello reducir el tiempo de respuesta en aplicaciones paralelas que utilizan algoritmos que manejan información global.

Como mencionábamos, el manejo de información parcial permite reducir el número de comunicaciones necesarias en la etapa de balance, esto se debe a que sólo se requiere conocer el índice de carga de algunos procesadores y no de todos. En este sentido, los procesadores del sistema son organizados de tal manera que un procesador sólo puede comunicarse con algunos procesadores, para ello se propone el uso de topologías de comunicación.

Debido a que las topologías de comunicación son un factor importante en el balance de carga [21], proponemos utilizar topologías de comunicación lógicas en donde los procesadores puedan comunicarse mediante un esquema de vecinos (Figura 4.1.a), en lugar de un esquema de comunicación global (Figura 4.1.b), de tal forma que cuando un procesador requiera balancear carga sólo lo hará con procesadores que se encuentren en su vecindad, esto significa que sólo utilizará parte de la información del estado de carga del sistema.

Las topologías que se muestran en la Figura 4.2, disminuyen el número de mensajes. Sin

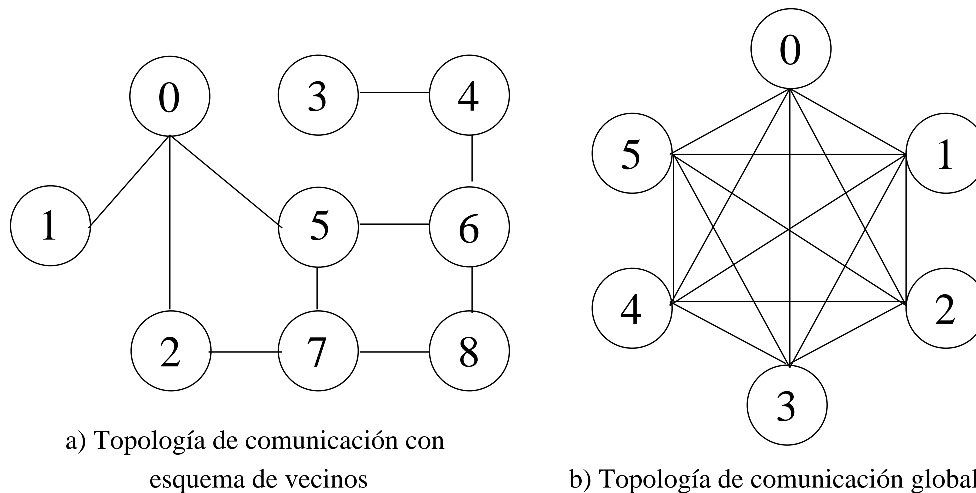


Figura 4.1: Topologías de comunicación con esquema de vecinos y global

embargo, algunas etapas relativamente sencillas cuando se utiliza comunicación global, con el uso de un esquema de vecinos se complican. Por ejemplo, el determinar el momento en el cuál se puede dar por terminada la ejecución de la aplicación. Con los algoritmos que manejan comunicación global la terminación es sencilla, porque todos los procesadores conocen el estado de carga del sistema, pero en el caso de los algoritmos con comunicación con esquema de vecinos esto es más complicado debido a que un procesador sólo conoce el estado de carga de sus vecinos, por ello no puede tomar la decisión de terminar su ejecución basándose en esta información.

Otro caso más se presenta en la recolección de resultados, debido a que cada procesador ha consumido una parte de la carga total del sistema, cada procesador tiene sus propios resultados (resultados parciales), para conocer el resultado global es necesario recolectar todos estos resultados. Cuando los algoritmos manejan información global, basta con especificar que procesador será el encargado de recolectar los resultados para al final tener el resultado global. Cuando se maneja información parcial esta estrategia no es posible debido a que los procesadores sólo se comunican con sus vecinos, por este motivo es necesario especificar cuál será la estrategia para recolectar los resultados parciales y así conocer el resultado global.

En base a lo anterior, el diseño de los algoritmos se realizó considerando las siguiente

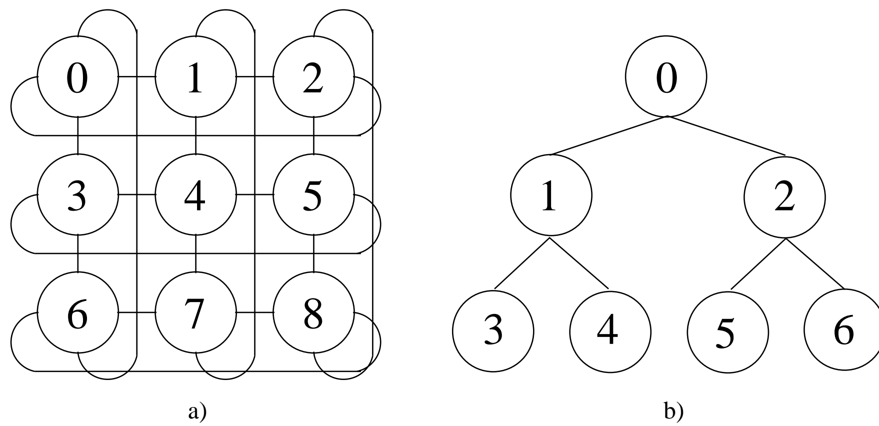


Figura 4.2: Topologías lógicas de comunicación toroide (a) y árbol binario (b)

etapas:

- Inicialización.
- Balance de carga.
- Búsqueda de una vista global de carga.
- Terminación.

Inicialización

En esta etapa se especifica las tareas y la forma en que los procesadores inician su ejecución.

Balance de carga

La etapa de balance de carga permite a un procesador obtener carga de otros procesadores. La estrategia de balance utilizada es la de subasta. A diferencia del algoritmo de DLML, la subasta sólo es ejecutada con los procesadores vecinos.

Búsqueda de una vista global de carga

Saber cuándo un procesador debe dar por terminada su ejecución se complica con el manejo de información parcial. Para resolver este problema se ha utilizado el protocolo PIF (Propagation of Information with Feedback) [19]. El protocolo PIF es una mejora al protocolo PI (Propagation of Information) [19] que permite realizar la propagación de un mensaje en un sistema distribuido a través de una búsqueda en amplitud, el protocolo PI funciona de la siguiente manera:

- Un procesador s es el encargado de propagar un mensaje a los procesadores con los cuales puede comunicarse (vecinos).
- Cada procesador i que reciba el mensaje por primera vez reexpide el mensaje a todos sus procesadores vecinos, incluyendo al procesador s de quien recibió el mensaje por primera vez.
- La terminación se presenta cuando los procesadores han terminado de reexpedir el mensaje a todos sus vecinos.

Una de las propiedades de este protocolo es la generación de un árbol de peso mínimo, en donde la paternidad de un procesador sobre otro dentro del árbol queda determinada por la forma en que el mensaje es propagado, de tal manera que un proceso i sólo puede ser hijo de s, si de este ha recibido el mensaje por primera vez. Si consideramos la topología de comunicación de la Figura 4.1a, una posible secuencia de la propagación del mensajes se muestra en la Figura 4.3.

Al inicio, el procesador *0* propaga un mensaje con etiqueta *1*, el cual indica el orden en el que se envía el mensaje. En seguida los procesos que lo reciben (*1*, *2* y *5*), lo reexpiden con etiqueta *2*. Este proceso continúa hasta que el mensaje es recibido por el procesador *3*, que de acuerdo con la topología de la Figura 4.3 es el más distante del procesador *0*, quien inicio con la propagación.

Aunque es evidente que el mensaje será propagado a todos los procesos sin importar la topología de comunicación, con este protocolo el proceso s desconoce si la propagación del

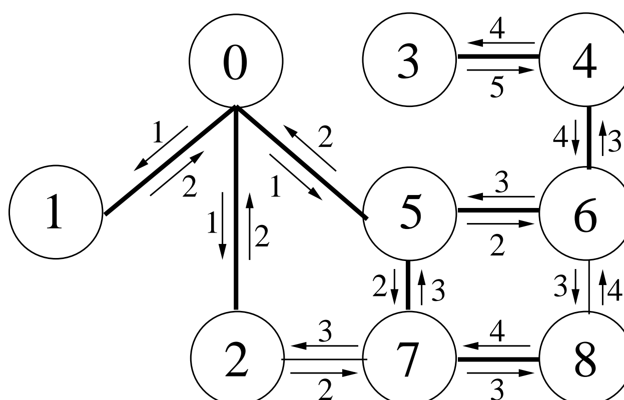


Figura 4.3: Secuencia de propagación de mensajes con el protocolo PI

mensaje ha finalizado, y aún cuando se garantizará confiabilidad en las comunicaciones para asegurar que la propagación finalice, no conocerá el momento en que finaliza. Para resolver este inconveniente el protocolo PIF incorpora una etapa de retroalimentación que en este trabajo es utilizada para conocer si en el sistema aún hay carga que procesar, el protocolo PIF es como sigue:

- Un procesador s es el encargado de enviar un mensaje a los procesadores vecinos.
- Cada procesador i que reciba el mensaje por primera vez, reexpide el mensaje sus vecinos, excepto al procesador s de quien lo recibe. Cada procesador que reside en una hoja del árbol termina por recibir el mensaje propagado de todos sus vecinos, después de recibir el último mensaje envía un mensaje a manera de acuse a su padre.
- Todos aquellos procesos que, dentro del árbol formado tengan hijos, esperan a recibir el acuse tantas veces como hijos tengan, cuando han recibido los acuses esperados devuelven esta misma confirmación a su padre.

Nuevamente si consideramos la topología de comunicación de la Figura 4.1.a, una posible secuencia de la propagación del mensaje con el protocolo PIF se muestra en la Figura 4.4.

Con esta mejora el procesador s que inició el protocolo puede saber cuándo se ha completado la propagación del mensaje. Como se mencionó anteriormente, la etapa de retro-

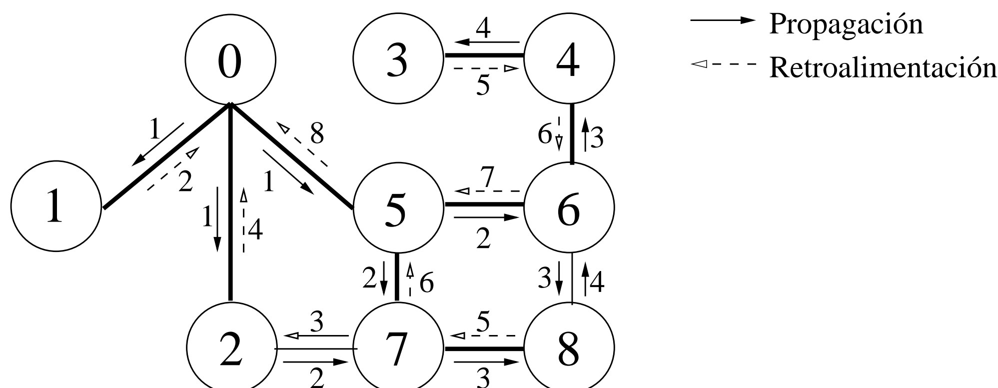


Figura 4.4: Secuencia de propagación de mensajes con el protocolo PIF

alimentación que consiste en enviar mensajes de acuse es aprovechada por cada procesador para enviar su estado de carga. Los procesadores que reciban mensajes de acuse además de enviar su estado de carga a su padre, también envían la de sus hijos. De esta forma al final el procesador s además de conocer cuando ha terminado la propagación, también conoce cuanta carga hay en el sistema, con esto ahora es posible saber si se puede dar por terminada la ejecución de la aplicación.

Terminación

Después de que el procesador s sabe que es posible dar por terminada la ejecución, necesitará dar a conocer a los demás procesadores esta condición, para ello basta con ejecutar una vez más el protocolo PIF. La etapa de propagación sirve para avisar de la finalización y la etapa de retroalimentación para que los procesadores envíen sus resultados. Eventualmente s será quien reciba los resultados de todos los procesadores conociendo así el resultado global.

El diseño de los algoritmos que proponemos está basado en las etapas que se acaban de describir. A continuación se presenta el algoritmo Toroide.

4.2. Algoritmo toroide

El algoritmo consiste en balancear carga entre procesos que se encuentran conectados siguiendo la topología de comunicación lógica conocida como toroide (Figura 4.2.a). Esta topología tiene como característica que cada procesador está conectado con cuatro procesadores vecinos. A continuación se describen las etapas de inicialización, balance de carga, búsqueda de una vista global de carga y terminación. Para cada etapa se describe su funcionamiento y se presenta el pseudocódigo correspondiente. En el pseudocódigo se utilizan variables y mensajes de los cuales también se presenta su descripción. En el caso de los mensajes, estos aparecen en mayúsculas.

4.2.1. Etapa de inicialización

Como se mencionó anteriormente, en esta etapa se especifica qué procesador es el que inicia con toda la carga (en el caso de la topología de la Figura 4.2.a es el procesador con identificador 0). El resto de los procesadores se encuentran sin carga por lo que inician con la etapa de balance de carga. A continuación se muestra el pseudocódigo correspondiente a esta etapa.

```
<1> al estar descargado y balance_iniciado = 0 efectúa
<2> envía REQ_INFO_CARGA a todo k en vecinos
<3> balance_iniciado <- 1
```

En el pseudocódigo se utilizan variables cuyo propósito fundamental es el de mantener el estado de ejecución del algoritmo e información tal como el conjunto de procesadores vecinos, etc. Las variables que se utilizan en el pseudocódigo de esta etapa son las siguientes:

- *vecinos*: almacena los identificadores de los procesadores vecinos con los que un procesador mantiene comunicación.
- *balance_iniciado*: permite saber cuando un procesador ha iniciado la etapa de balance.

Cuando su valor es 0, el procesador no ha iniciado la etapa de balance, *1* cuando la etapa ha iniciado.

Además de utilizar variables, también es necesario el uso de mensajes para que los procesadores se comuniquen. En esta etapa sólo se utiliza el siguiente mensaje:

- *REQ_INFO_CARGA*: Un procesador al pasar a su estado descargado, envía este mensaje a sus vecinos para solicitar que le envíen su índice de carga, este es el inicio de la etapa de balance de carga.

4.2.2. Etapa de balance de carga

Esta etapa es ejecutada por un procesador para obtener carga de otro procesador. Es iniciada cuando un procesador se encuentra descargado y está basada en la misma política que utiliza el algoritmo global de DLML (subasta). La principal diferencia entre la subasta global y en el algoritmo toroide, se encuentra en la cantidad de mensajes requeridos en su ejecución. En el algoritmo global la cantidad de mensajes requeridos para que todos los procesadores ejecuten la subasta es $n(2(n-1))$, donde n es el número de procesadores participantes, de esta forma la complejidad en mensajes es O(n^2). En el caso del algortimo toroide, en donde el número de mensajes depende del número de procesadores vecinos (independientemente del número de procesadores en el sistema con la topología toroide un procesador siempre tiene 4 vecinos), la complejidad es O(n).

Los pasos que sigue la subasta en la topología toroide se muestran en la Figura 4.5. Partiendo del centro, cuando el procesador 4 se queda sin carga, envía un mensaje a sus vecinos (1, 3, 5, 7) para solicitar su índice de carga (Figura 4.5.a), los procesadores responden enviando un mensaje con su índice de carga (Figura 4.5.b). Después de que el procesador ha recibido los índices de carga de sus vecinos, elige al procesador vecino más cargado para solicitar que le transfiera parte de su carga, en este caso el procesador 5 (Figura 4.5.c). Cuando el procesador recibe la solicitud de transferir carga, antes verifica que la cantidad de carga que posee en ese momento sea la suficiente como para no quedarse descargado después

de la transferencia. Si está condición se cumple envía parte de su carga, en este caso el vecino 5 (Figura 4.5.d), en caso contrario, envía un mensaje indicando que se encuentra descargado, por lo que el procesador que se encuentra en espera de carga reinicia la subasta.

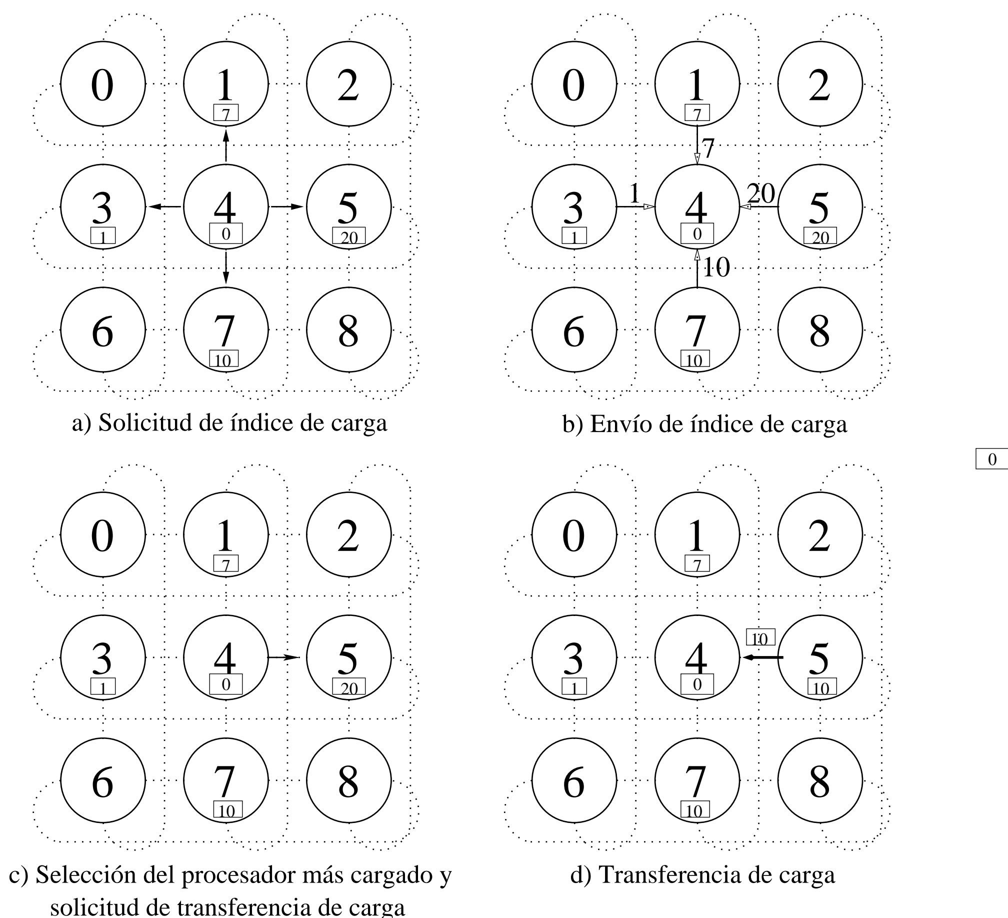
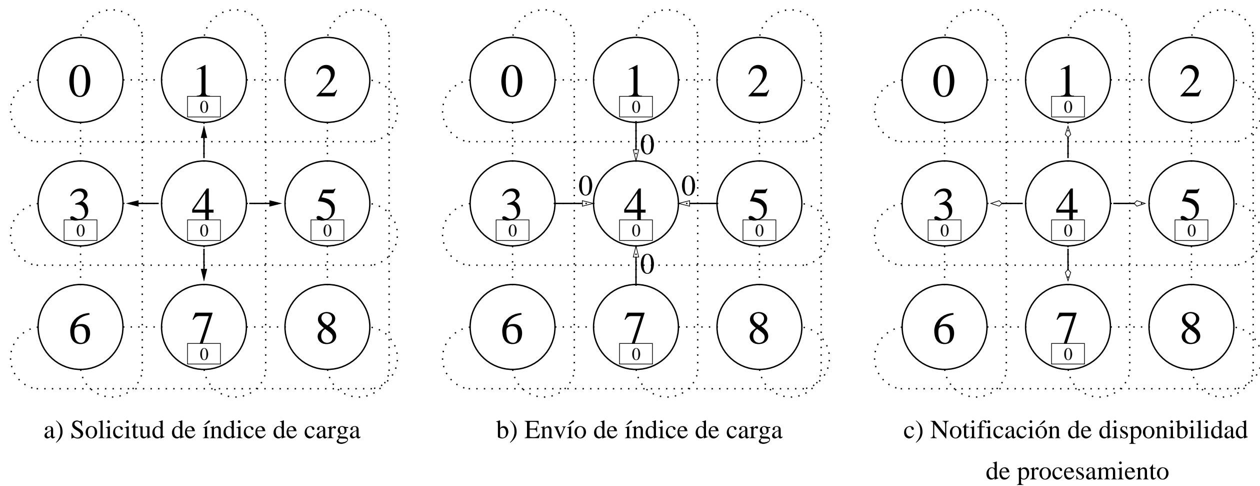


Figura 4.5: Algoritmo de subasta para la topología toroide

Una condición adicional en la ejecución de la subasta anterior se presenta cuando el procesador 4 solicita estado de carga (Figura 4.6.a), y los cuatro procesadores vecinos responden con 0 como índice de carga (Figura 4.6.b). En este caso el procesador en lugar de solicitar transferencia de carga, simplemente notifica a sus vecinos (Figura 4.6.c) que ha ejecutado la subasta sin conseguir carga y que a partir de ese momento se va a bloquear (no obstante que se encuentra disponible para seguir procesando carga). De esta forma un procesador vecino que eventualmente obtenga carga de otro lugar del sistema, sabrá si tiene procesadores veci-

nos que se encuentran disponibles, de ser así podrá enviar un mensaje indicando que pueden reanudar la subasta.



a) Solicitud de índice de carga

b) Envío de índice de carga

c) Notificación de disponibilidad de procesamiento

Figura 4.6: Notificación de disponibilidad de procesamiento

El pseudocódigo de la etapa de balance es el siguiente:

```
<4>  al recibir REQ_INFO_CARGA desde k en vecinos efectúa
<5>     envía RESPUESTA_INFO_CARGA a k en vecinos con el indice_carga

<6>  al recibir RESPUESTA_INFO_CARGA(indice_carga) de K en vecinos efectúa
<7>     respuesta_carga <- respuesta_carga + 1
<8>     guarda indice_carga de k
<9>     si respuesta_carga == 4 /*Si todos han respondido*/
<10>        entonces respuesta_carga <- 0
<11>                determina al k en vecinos más cargado
<12>                si existe algún k en vecinos más cargado
<13>                   entonces envía DAME_CARGA a k en vecinos más cargado
<14>                   otro     si id == s
<15>                               entonces si primera_busqueda == 0
<16>                                  entonces envía BUSCA_CARGA(0)
```

```
                                               a todo k en vecinos
<17>                                           primera_busqueda <- 1
<18>                                  otro envía BUSCA_CARGA(0) a todo k en hijos
<19>                               otro     envía DESCARGADO a todo k en vecinos
<20>    desecha los índices de carga recibidos

<21> al recibir DAME_CARGA desde k en vecinos efectúa
<22>    carga_enviar <- indice_carga / X
<23>    si carga_enviar > 1
<24>       entonces indice_carga <- (indice_carga - carga_enviar)
<25>                envía CARGA(carga_enviar) a k en vecinos
<26>       otro     envía CARGA(0) a k en vecinos
<27>    carga_enviar <- 0

<28> al recibir CARGA(carga) de k en vecinos efectúa
<29>    si carga > 0
<30>       entonces indice_carga <- carga
<31>                balance_iniciado <- 0
<32>                envía BALANCE a todo k en vecinos_descargados
<33>                borra k en vecinos_descargados
<34>       otro     ejecuta <2> y <3> de la etapa de inicialización

<35> al recibir DESCARGADO de k en vecinos efectúa
<36>    guarda k en vecinos_descargados

<37> al recibir BALANCE desde k en vecinos efectúa <2> y <3>
     de la etapa de inicialización
```

Las variables utilizadas son:

- *índice_carga*: carga que posee un procesador.

- *respuesta_carga*: permite conocer cuantas veces ha recibido el mensaje *RESPUESTA_INFO_CARGA* (al recibir este mensaje se incrementa en *1*) para saber si ha recibido los índices de carga de todos los vecinos.

- s: contiene el identificador del proceso encargado de ejecutar las etapas de búsqueda global de carga y terminación, es conocida por todos los procesadores.

- *primera_busqueda*: contiene *0* cuando la búsqueda global de carga aún no se ha ejecutado y permite ejecutar el protocolo PIF en su especificación original (a través de la topología toroide), *1* después de que el procesador finaliza su participación en la primer ejecución de la etapa de búsqueda global de carga y significa que en subsecuentes búsquedas el protocolo PIF se deberá ejecutar sólo a través del árbol de peso mínimo.

- *hijos*: conjunto de vecinos que durante la generación del árbol de peso mínimo hayan confirmado paternidad.

- *carga_enviar*: cociente del índice de carga entre X, indica cuantos datos deberán ser transferidos.

- X: divisor, la cantidad de datos a trasferir depende de su valor, en caso de que su valor fuera *2*, significa que se debe transferir la mitad de la carga de un procesador.

- *carga*: datos transferidos desde otro procesador.

- *vecinos_descargados*: conjunto de vecinos que han notificado disponibilidad de procesamiento.

Los mensajes que se utilizan son:

- *RESPUESTA_INFO_CARGA*: Un procesador fuente envía este mensaje con el índice de carga actual. El procesador que recibe este mensaje incrementa el número de mensajes recibidos de este tipo y almacena la información del procesador que le haya reportado mayor cantidad de carga.

- *DAME_CARGA*: Al elegir al procesador del grupo de *vecinos* más cargado, se le envía un mensaje de este tipo para solicitar la transferencia de carga.
- *CARGA*: Un procesador envía este mensaje para transferir parte de su carga.
- *DESCARGADO*: En caso de que ningún k procesador en *vecinos* se encuentre cargado, después de haber recibido los mensajes *RESPUESTA_INFO_CARGA* esperados, envía este mensaje a todo k en vecinos. Los procesadores que reciban este mensaje, almacenan el identificador del procesador para saber que procesadores vecinos han intentado obtener carga y actualmente se encuentran descargados.
- *BALANCE*: Un procesador envía este mensaje a todos los procesadores que se encuentran en *vecinos_descargados*, al obtener carga de otro procesador. Aquellos procesadores que lo reciban podrán realizar nuevamente etapa de balance.

Debido a que el manejo de información parcial impide a un procesador saber si en algún procesador del sistema aún hay carga que procesar, no puede determinar cuándo es el momento adecuado de pasar a la etapa de terminación. Para resolver este inconveniente se incluyó una etapa en donde a través de un protocolo de búsqueda en amplitud es posible conocer el estado global de carga del sistema, esta etapa es llamada búsqueda de una vista global de carga.

4.2.3. Etapa de búsqueda de una vista global de carga

La responsabilidad de iniciar esta etapa es asignada al primer procesador que tuvo carga.

Si este procesador después de haber ejecutado la subasta no obtiene carga, probablemente se debe a que no hay más carga que procesar en todo el sistema, si es así puede iniciar la etapa de terminación, pero si hay más carga sería un error. Para saber con certeza si hay, o no hay más carga en el sistema, se utiliza el protocolo PIF.

Como se recordará, el protocolo PIF permite realizar una búsqueda en amplitud en todos los procesadores que forman parte del sistema. Este protocolo considera dos etapas: propagación y retroalimentación.

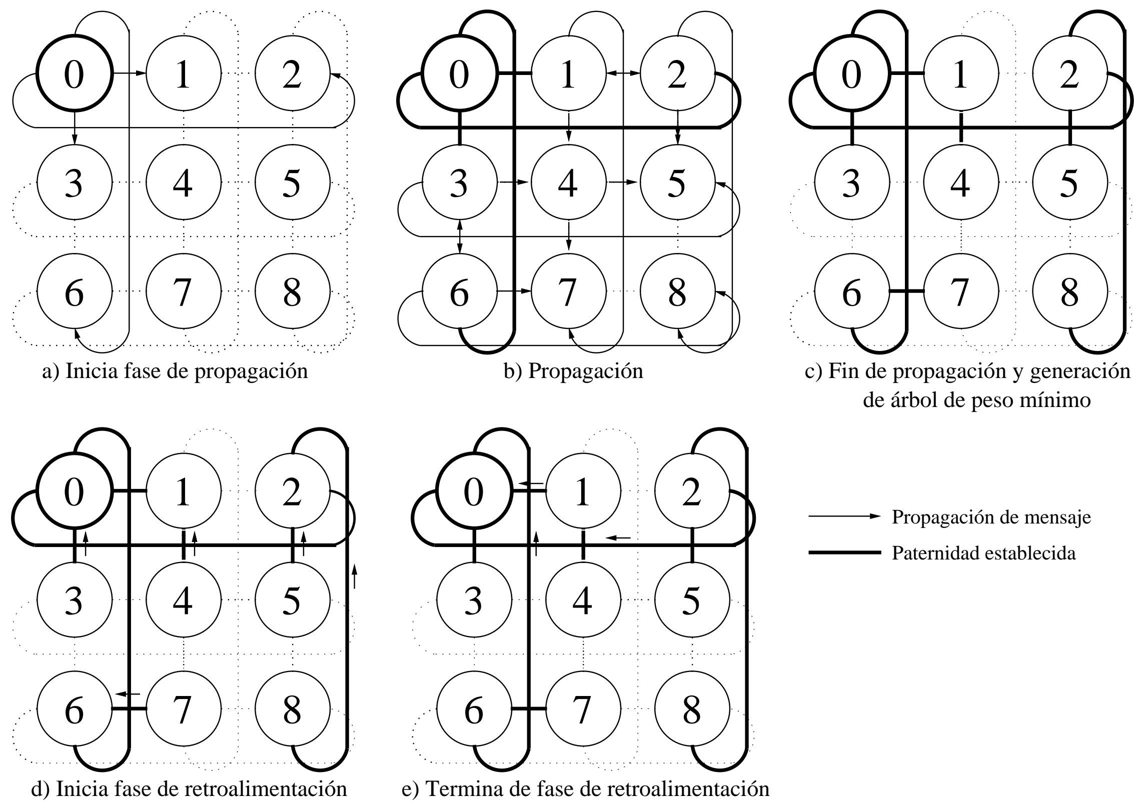


Figura 4.7: Ejecución del algoritmo PIF en la topología toroide

En la topología toroide, la etapa de propagación es iniciada por el procesador 0 quien envía un mensaje a sus vecinos (1, 2, 3 y 6) (Figura 4.7.a). Los procesadores al recibir por primera vez el mensaje, lo reexpiden a sus vecinos, excepto a aquel de quien lo reciben por primera vez, a quien consideran su padre (Figura 4.7.b). Esta etapa se realiza hasta que el mensaje haya sido propagado a todos los procesadores. Una de las propiedades de este protocolo, es la generación de un árbol de peso mínimo [19] (a tiempo de ejecución) en donde el procesador 0 es la raíz del árbol (Figura 4.7.c).

La etapa de retroalimentación inicia cuando un procesador hoja (un procesador es hoja dentro del árbol formado si no tiene hijos), ha recibido el mensaje propagado tantas veces como vecinos tenga (en este caso cuatro), entonces envía el mensaje de confirmación al procesador de quien recibió el mensaje por primera vez (mensaje de confirmación al procesador padre), en el cual envía su estado de carga (Figura 4.7.d). Los procesadores que reciban todos

los mensajes de confirmación de sus hijos, envían su mensaje de confirmación a su padre, enviando su estado de carga más el de su(s) hijo(s). Eventualmente el procesador que inició la etapa de búsqueda (procesador 0) recibe los mensajes de confirmación de sus hijos (1, 2, 3 y 6) en donde se envía el estado de carga de todo el sistema (Figura 4.7.e). Con la información recibida puede determinar si en el sistema aún existe carga, de ser así, este procesador simplemente iniciará nuevamente el protocolo de subasta, en caso contrario iniciará la etapa de terminación.

Como se menciona anteriormente, el protocolo PIF tiene como una de sus propiedades la generación de un árbol de peso mínimo a tiempo de ejecución. Notar que si las subsecuentes ejecuciones del protocolo PIF se realizan siguiendo el árbol formado, el costo en mensajes se reduce al ocupar los enlaces que forman el árbol, en vez de todos los enlaces de la topología.

Para poder utilizar el árbol es necesario que un procesador conozca quién es su padre y sus hijos. Sin embargo, en la especificación del protocolo original un procesador conoce quién es su padre, pero no conoce quiénes son sus hijos.

Para poder resolver este inconveniente se incluye un mensaje adicional al protocolo PIF original. Este mensaje se envía en la etapa de propagación (Figura 4.8.a), justo después de que un procesador recibe por primera vez el mensaje propagado envía un mensaje que llamaremos *confirmación de paternidad* a su procesador padre (Figura 4.8.b), de esta forma todo procesador que reciba este tipo de mensajes sabrá cuántos y quiénes son sus hijos. Con esta información en futuras ejecuciones de la búsqueda global de carga se podrá utilizar el árbol de peso mínimo (Figura 4.9) formado en la primer ejecución.

El pseudocódigo es el siguiente:

```
<38> al recibir BUSCA_CARGA(indice_carga) de k en vecinos efectúa
<39>    carga_encontrada <- carga_encontrada + indice_carga
<40>    mesajes_busca_carga <- mesajes_busca_carga + 1
<41>    si id == s
<42>       entonces si mensajes_busca_carga == # hijos
<43>                 entonces mensajes_busca_carga <- 0
```

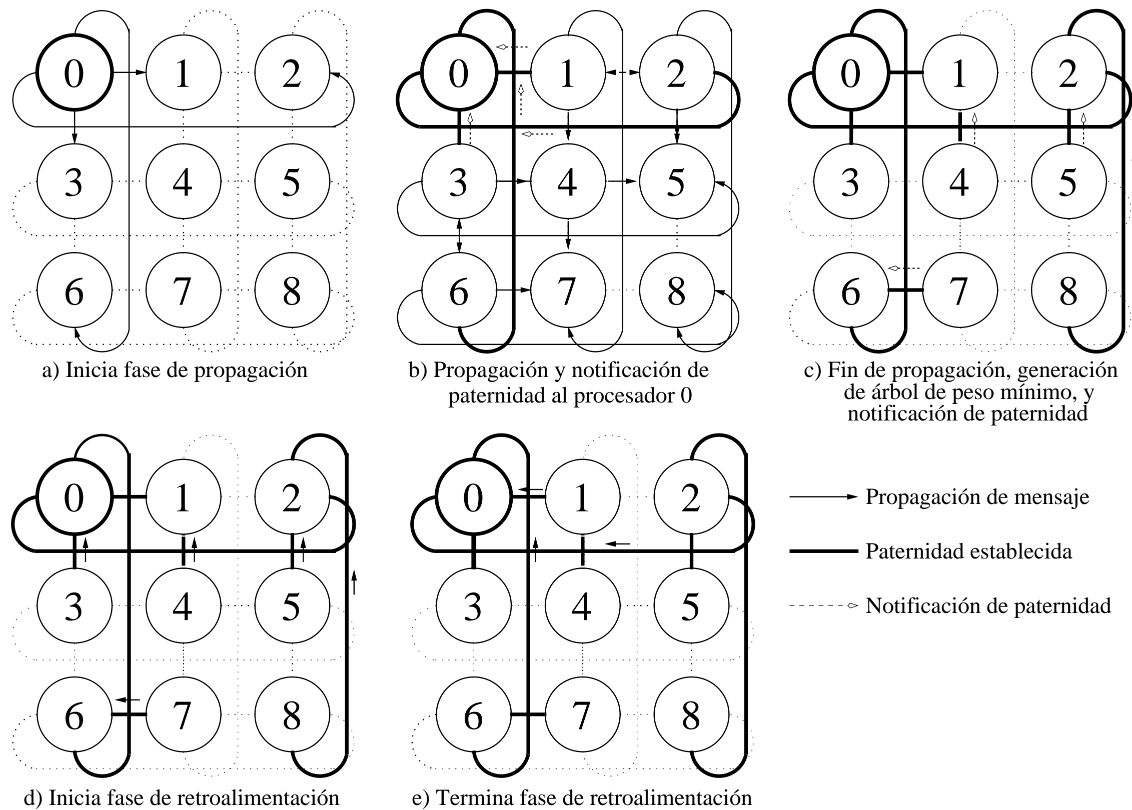


Figura 4.8: Ejecución del algoritmo PIF modificado

```
<44>                          si carga_encontrada == 0
<45>                             entonces envía TERMINA a todo k en hijos
<46>                             otro     ejecuta <2> y <3>
<47>                                      carga_encontrada <- 0
<48>          otro     si primera_busqueda == 0 /*busca carga por primera vez*/
<49>                      entonces si mensajes_busca_carga == 1
<50>                                  entonces envía ERES_MI_PADRE a k en vecinos
<51>                                           padre <- k
<52>                                           reexpide BUSCA_CARGA(indice_carga)
                                                        a vecinos excepto k
<53>                               si mensajes_busca_carga == 4
<54>                                  entonces mensajes_busca_carga <- 0
```

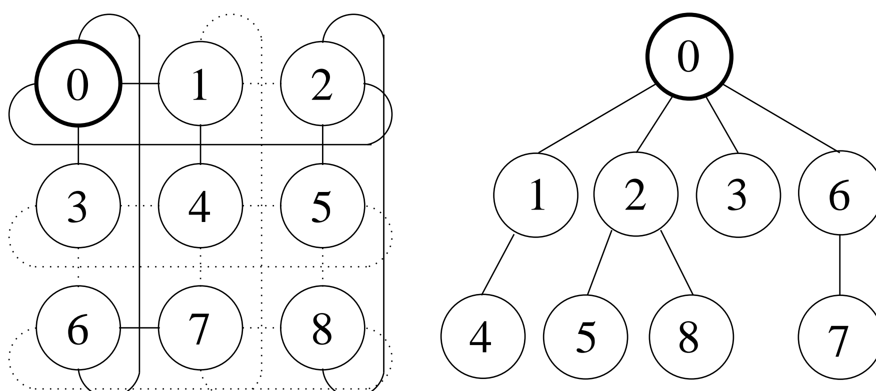


Figura 4.9: Árbol formado en tiempo de ejecución del protocolo PIF

```
<55>                                       primera_busqueda <- 1
<56>                                       envía BUSCA_CARGA(carga_encontrada)
                                                 a padre
<57>                                       carga_encontrada <- 0
<58>                    otro si mensajes_busca_carga == 1 /*busca carga sobre árbol*/
<59>                            entonces si # hijos > 0
<60>            /*desciende*/             entonces reexpide
                                              BUSCA_CARGA(indice_carga) a hijos
<61>            /*Es hoja, asciende*/     otro  mensajes_busca_carga <- 0
<62>                                            envía BUSCA_CARGA(indice_carga)
                                                      a padre
<63>                                            carga_encontrada <- 0
<64>                         si # hijos > 0
<65>                            entonces si mensajes_busca_carga == (# hijos + 1)
<66>                                     entonces mensajes_busca_carga <- 0
<67>                                            envía
           /*asciende*/                         BUSCA_CARGA(carga_encontrada)
                                                a padre
<68>                                            carga_encontrada <- 0
<69>
```

```
<70> al recibir ERES_MI_PADRE de k en vecinos agrega k a hijos
```

Las variables que se presentan en esta etapa son las siguientes:

- *carga_encontrada*: suma de los índices de carga recibidos en la etapa de búsqueda de carga más el índice de carga del procesador que la mantiene.
- *padre*: contiene el identificador del procesador al cual durante la generación del árbol de peso mínimo se notificó paternidad.
- *mensajes_busca_carga*: permite conocer cuántas veces se ha recibido el mensaje *BUSCA_CARGA*, *0* indica que el mensaje recibido deberá ser reexpedido el momento de recibirlo, mayor a *0* el mensaje no se deberá reexpedir. Toma valor de *4* sólo en la primer ejecución de la búsqueda, e indica que se han recibido los mensajes esperados (de sus vecinos) y se debe confirmar a *padre*. En subsecuentes ejecuciones de la búsqueda cuando *mensajes_busca_carga = # hijos*, significa que se debe hacer la confirmación a padre.

En esta etapa se presentan los siguientes mensajes:

- *BUSCA_CARGA*: El procesador *s* envía este mensaje después de haber ejecutado la etapa de balance y no haber encontrado algún *k* procesador en *vecinos* con carga. Con este mensaje inicia la etapa de búsqueda de una vista global de carga.
- *ERES_MI_PADRE*: Es mensaje es enviado al *k* procesador en *vecinos* de quien recibe por primera vez el mensaje *BUSCA_CARGA* en la primer ejecución de la etapa de búsqueda, con ello le notifica que lo reconocerá como padre.

Si después de la búsqueda de una vista global de carga, se determina que no hay más carga que procesar, el mismo procesador que ha iniciado la etapa de búsqueda, será el encargado de iniciar la etapa de terminación que a continuación se describe.

4.2.4. Etapa de terminación

En esta etapa se notifica a los procesadores que pueden iniciar con la recolección de resultados, y posteriormente finalizar su ejecución. Para ello se propaga un mensaje a todos los procesadores a través del árbol de peso mínimo, para notificarles que no hay más carga. El mensaje empieza a propagarse hasta llegar a aquellos procesadores que dentro del árbol sean hojas. Cuando los procesadores hojas reciben el mensaje envían sus resultados parciales a su procesador padre, y posteriormente finalizan su ejecución. Aquellos procesadores que tengan hijos, esperan a recibir los resultados parciales correspondientes, cuando los hayan recibido, envían a sus correspondientes padres sus resultados más los de sus hijos, posteriormente finalizan su ejecución. De esta forma los resultados de cada procesador remontan el árbol hasta llegar al procesador raíz el cual conoce el resultado global. Es importante mencionar que el utilizar el árbol facilita la recolección de resultados, en otro caso sería necesario especificar una estrategia particular para la recolección. A continuación se presenta el pseudocódigo.

```
<71> al recibir TERMINA de padre efectúa
<72>    si # hijos > 0
<73>       entonces reexpide TERMINA a todo k en hijos
<74>       otro envía RESULTADO(resultado_parcial) a padre
<75>            termina

<76> al recibir RESULTADO (resultado) de k en hijos efectúa
<77>    resultados_recibidos <- (resultados_recibidos + 1)
<78>    resultado_acumulado <- resultado_acumulado + resultado
<79>    si resultados_recibidos == # hijos
<80>       entonces si id == s
<81>                   entonces imprime (resultado_acumulado + resultado_parcial)
<82>                            termina
<83>                   otro     resultado_acumulado <-
                                (resultado_acumulado + resultado_parcial)
<84>                            envía RESULTADO(resultado_acumulado) a padre
```

```
<85>                              termina
```

Los variables que se presentan en esta etapa son las siguientes:

- *resultado_parcial*: contiene el resultado de los datos procesados.
- *resultados_recibidos*: permite conocer cuantas veces se ha recibido el mensaje *RESULTADO* con los resultados parciales de los procesadores en *hijos*.
- *resultado_acumulado*: suma de los datos parciales recibidos.

Los mensajes que se presentan en esta etapa son:

- *TERMINA*: El procesador s envía este mensaje después de haber ejecutado el protocolo para buscar carga en el sistema, y tener la certeza de que no hay carga en algún procesador del sistema. El mensaje indica a un procesador que se debe iniciar la recolección de resultados y posteriormente termina la ejecución. Los procesadores que reciban el mensaje lo reexpiden a todo procesador k en el grupo de procesadores *hijos*, de lo contrario envía *RESULTADO*, con este mensaje inicia la etapa de terminación.
- *RESULTADO*: Un procesador envía este mensaje después de haber recibido los datos parciales los procesadores en *hijos* o después de recibir termina y $hijos = \emptyset$. En este mensaje se envían sus resultados parciales más los de sus *hijos* si es el caso.

En la siguiente sección se presenta el algoritmo para la topología de árbol binario.

4.3. Algoritmo árbol binario

Este algoritmo consiste en realizar balance de carga, utilizando información parcial entre procesadores que se encuentran conectados mediante una topología de comunicación de árbol binario. De acuerdo a la topología un procesador puede tener uno, dos o hasta tres procesadores vecinos. Por ejemplo, en el caso de la raíz (Figura 4.10.a), el procesador 0 tiene dos vecinos (1 y 2). En el caso de los procesadores intermedios 1 y 2 (Figura 4.10.b), tienen

tres vecinos 0, 3 ,4 y 0, 5, 6 respectivamente. Finalmente los procesadores hoja sólo tienen un vecino (Figura 4.10.c). Al igual que en el algoritmo toroide, para este algoritmo también se consideran cuatro etapas. A continuación se presenta cada una de ellas.

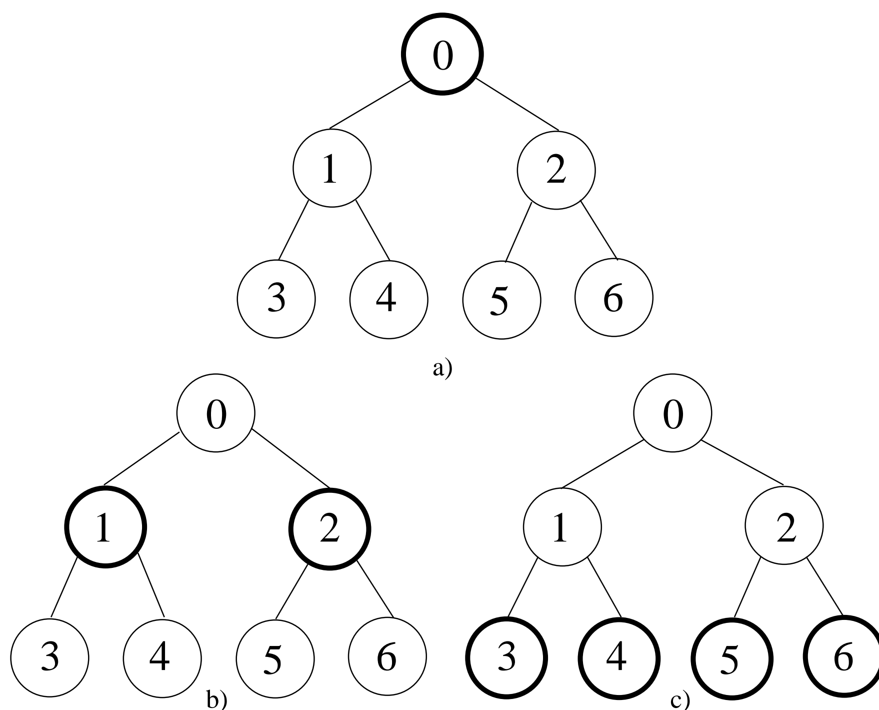


Figura 4.10: Configuración de comunicación de la topología de árbol binario

4.3.1. Etapa de inicialización

En esta etapa se especifica que procesador es el que inicia con toda la carga (en el caso de la topología de la Figura 4.2.b es el procesador con identificador 0). El resto de los procesadores se encuentran sin carga por lo que inician con la etapa de balance de carga. A continuación se muestra el pseudocódigo correspondiente a esta etapa. El pseudocódigo de esta etapa es el siguiente:

```
<1>  al estar descargado y balance_iniciado = 0 efectúa
<2>      envía REQ_INFO_CARGA a todo k en vecinos
<3>      balance_iniciado <- 1
```

4.3.2. Etapa de balance de carga

En este algoritmo el primer procesador con carga será aquel que dentro de la topología sea la *raíz*, a partir de él la carga será distribuida hacía sus ramas. El enfoque implementado para realizar el balance de carga es prácticamente el mismo que se describe en el algoritmo de toroide (subasta), con la diferencia que el número de vecinos involucrados en el balance varía. Con el algoritmo de árbol habrá etapas de balance ejecutadas sólo con un procesador, y otras con tres procesadores dependiendo del procesador que la inicia. El caso particular del procesador raíz es el único que puede balancear carga con dos procesadores vecinos. A continuación se presenta el pseudocódigo correspondiente a esta etapa.

```
<4>  al recibir REQ_INFO_CARGA desde k en vecinos efectúa
<5>     envía RESPUESTA_INFO_CARGA a k en vecinos con el indice_carga

<6>  al recibir RESPUESTA_INFO_CARGA(indice_carga) de todo k en vecinos efectúa
<7>     respuesta_carga <- respuesta_carga + 1
<8>     guarda indice_carga de k
<9>     si respuesta_carga == # vecinos
<10>        entonces respuesta_carga <- 0
<11>                determina al k en vecinos más cargado
<12>                si existe algún k en vecinos más cargado
<13>                   entonces envía DAME_CARGA a k en vecinos más cargado
<14>                   otro     si id == raiz_arbol
<15>                               entonces envía BUSCA_CARGA(0) a todo k en ramas
<16>                               otro     envía DESCARGADO a todo k en vecinos
<17>    desecha los índices de carga recibidos

<18> al recibir DAME_CARGA desde k en vecinos efectúa
<19>    carga_enviar <- indice_carga / X
<20>    si carga_enviar > 1
<21>       entonces indice_carga <- (indice_carga - carga_enviar)
```

```
<22>                envía CARGA(carga_enviar) a k en vecinos
<23>       otro     envía CARGA(0) a k en vecinos
<24>    carga_enviar <- 0

<25> al recibir CARGA(carga) de k en vecinos efectúa
<26>    si carga > 0
<27>       entonces indice_carga <- carga
<28>                balance_iniciado <- 0
<29>                envía BALANCE a todo k en vecinos_descargados
<30>                borra k en vecinos_descargados
<31>       otro     ejecuta <2> y <3>

<32> al recibir DESCARGADO de k en vecinos efectúa
<33>    guarda k en vecinos descargados

<34> al recibir BALANCE desde k en vecinos efectúa <2> y <3>
```

Si el procesador *raíz* después de haber ejecutado el protocolo de subasta no obtiene más carga, inicia la etapa de búsqueda de una vista global de carga, al igual que en el algoritmo toroide.

4.3.3. Etapa de búsqueda de una vista global de carga

En esta etapa nuevamente se utiliza el protocolo PIF para conocer el estado global de carga y así determinar si es conveniente continuar con la etapa de terminación. A continuación se presenta el pseudocódigo de esta etapa.

```
<35> al recibir BUSCA_CARGA(indice_carga) de k en vecinos efectúa
<36>    carga_encontrada <- carga_encontrada + indice_carga
<37>    mesajes_busca_carga <- mesajes_busca_carga + 1
<38>    si id == s
<39>       entonces si mensajes_busca_carga == # vecinos
```

```
<40>                  entonces mensajes_busca_carga <- 0
<41>                           si carga_encontrada == 0
<42>                              entonces envía TERMINA a todo k en ramas
<43>                              otro     ejecuta <2> y <3>
<44>                                       carga_encontrada <- 0
<45>           otro     si mensajes_busca_carga == 1
<46>                       reexpide BUSCA_CARGA(indice_carga) a todo k en ramas
<47>                    si mensajes_busca_carga == # vecinos
<48>                       entonces mensajes_busca_carga <- 0
<49>                                caga_encontrada <- carga_encontrada + indice_carga
<50>                                envía BUSCA_CARGA(carga_encontrada) a raiz
<51>                                                 carga_encontrada <- 0
```

Como mencionábamos, el algoritmo PIF forma un árbol de peso mínimo a tiempo de ejecución que facilita la tarea de obtener una vista global de carga. Con la topología utilizada en este algoritmo no es necesario utilizar este árbol porque la topología ya tiene la forma de un árbol binario. De esta manera la propagación del mensaje inicia en *raíz* hasta los procesadores hoja a través de los enlaces que forman el árbol. Cuando un procesador hoja ha recibido el mensaje, puede iniciar la retroalimentación. En la retroalimentación un mensaje con el índice de carga es enviado desde cada hoja a su procesador padre. Los procesadores intermedios esperan los mensajes de confirmación en los cuales se envían los índices de carga, cuando han recibido los mensajes esperados, envían un mensaje hacía su *padre* con los índices de carga recibidos más el propio. La retroalimentación finaliza cuando el procesador *raíz* recibe los índices de carga de sus hijos con el índice de carga global. Con la información recibida el procesador *raíz* puede decidir si ejecuta nuevamente la etapa de balance o inicia la etapa de terminación.

4.3.4. Etapa de terminación

Esta etapa se presenta cuando se ha determinado que no hay más carga en el sistema y es iniciada por el procesador raíz. Consiste en ejecutar nuevamente el protocolo PIF. A dife-

rencia de la etapa anterior el mensaje propagado será para notificar al resto de procesadores que no hay más carga que procesar, que los resultados parciales se deben recolectar y que posteriormente pueden finalizar su ejecución. En el mensaje enviado en la retroalimentación se incluirán los resultados parciales recolectados de cada procesador (Figura 4.11.a). Eventualmente el procesador raíz que inicio esta etapa recibirá los resultados desde sus ramas y conocerá el resultado global (Figura 4.11.b). A continuación se presenta el algoritmo.

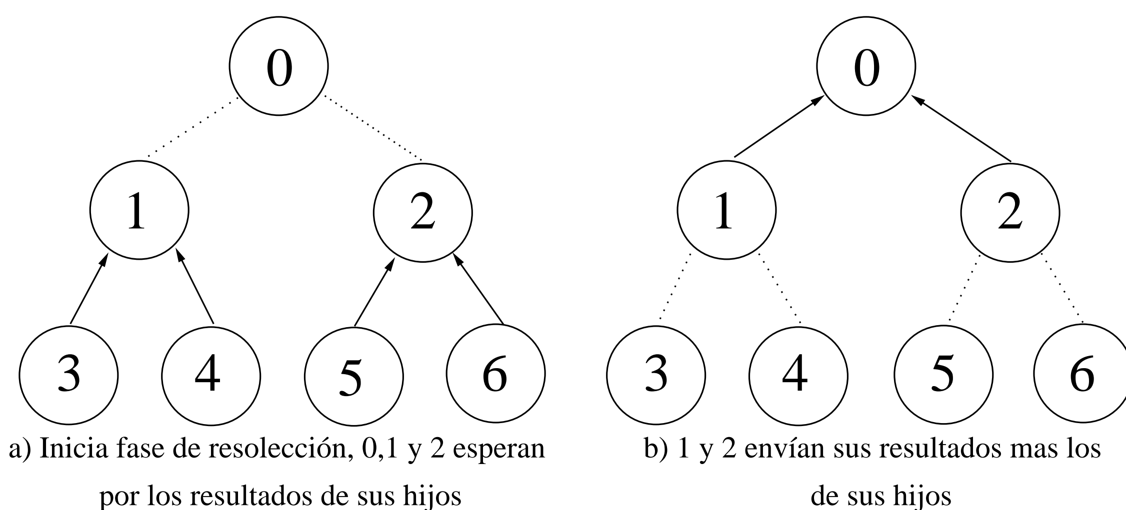


Figura 4.11: Recolección de resultados en la topología de árbol binario

El pseudocódigo de esta etapa es el siguiente:

```
<52> al recibir TERMINA de raíz efectúa
<53>    si # ramas > 0
<54>       entonces reexpide TERMINA a todo k en ramas
<55>       otro envía RESULTADO(resultado_parcial) a raiz
<56>            termina

<57> al recibir RESULTADO (resultado) de k en ramas efectúa
<58>    resultados_recibidos <- (resultados_recibidos + 1)
<59>    resultado_acumulado <- resultado_acumulado + resultado
```

```
<60>    si resultados_recibidos == # ramas
<61>       entonces si id == raiz_arbol
<62>                   entonces imprime (resultado_acumulado + resultado_parcial)
<63>                            termina
<64>                   otro resultado_acumulado <-
                            (resultado_acumulado + resultado_parcial)
<65>                        envía RESULTADO(resultado_acumulado) a raiz
<66>                        termina
```

Al igual que en el algoritmo anterior, cada procesador mantiene un conjunto de variables que le permiten conocer su condición durante la ejecución, las variables son las siguientes:

- *vecinos*, *balance_iniciado*, *índice_carga*, *respuesta_carga*, *carga_enviar*, *X*, *carga*, *vecinos_descargados*, *carga_encontrada*, *mensajes_busca_carga*, *resultado_parcial*, *resultados_recibidos* y *resultado_acumulado*: estas variables se han presentado en el algoritmo toroide y también son utilizadas en este algoritmo en donde cumplen con el mismo propósito.
- *raiz_arbol*: contiene el identificador del procesador raíz en el árbol de la topología de comunicación, el procesador con este identificador es el responsable de iniciar las tareas de búsqueda y terminación.
- *ramas*: conjunto de vecinos que en la topología sean ramas del procesador.
- *raiz*: contiene el identificador del procesador del cual son rama, salvo el procesador con identificador *raiz_arbol* todos los procesadores tienen un procesador raíz.

Los mensajes utilizados en este algoritmo son los siguientes:

- *REQ_INFO_CARGA*.
- *RESPUESTA_INFO_CARGA*.
- *DAME_CARGA*.
- *CARGA*.

- *DESCARGADO.*
- *BALANCE.*
- *BUSCA_CARGA.*
- *TERMINA.*
- *RESULTADO.*

En este algoritmo los mensajes utilizados ya han sido presentados en el algoritmo toroide y cumplen el mismo propósito. Debido a que en este algoritmo el protocolo PIF se ejecuta a través de un árbol, el mensaje de notificación de paternidad *ERES_MI_PADRE* no ha sido requerido, además de que las condiciones para el envío de mensajes cambian como consecuencia de la ejecución del protocolo sobre el árbol y el número de vecinos que puede tener un procesador. El pseudocódigo completo de los dos algoritmos presentados se puede consultar en el Apéndice A.

Los algoritmos presentados se han adaptado en la herramienta DLML, teniendo así una herramienta que permite desarrollar aplicaciones paralelas con la posibilidad de elegir de entre tres algoritmos de balance (global, toroide y árbol binario). En el siguiente capítulo se presenta la plataforma de experimentación (aplicaciones e infraestructura) utilizada y los resultados obtenidos.

Capítulo 5

Plataforma de experimentación y resultados

En esta capítulo se presenta la plataforma de experimentación, en donde escogimos a la biblioteca DLML como plataforma de prueba para integrar y comparar nuestros algoritmos (toroide y árbol) con manejo de información parcial con el algoritmo nativo de DLML que maneja información global. Los índices en esta comparación son principalmente los tiempos de respuesta y la escalabilidad para diferentes tipos de aplicaciones.

5.1. Aplicaciones

En esta sección presentamos un par de aplicaciones para probar nuestros algoritmos de balance de carga: Multiplicación de matrices y N-Reinas.

5.1.1. Multiplicación de matrices

En esta aplicación definimos a C como la matriz resultante de multiplicar la matriz A por la matriz B. Como sabemos, la multiplicación requiere que la matriz A sea de tamaño n x l y la matriz B de tamaño l x m, es decir que el número de columnas de la matriz A sea igual al número de filas de la matriz B. De esta forma cada elemento de la matriz resultante C se calcula por la siguiente fórmula:

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j}$$

Esto significa que cada elemento de la i-ésima fila de A sea multiplicado por un elemento de la j-ésima columna de B, y la suma de todos los productos será el valor del elemento c_{ij} de la matriz resultante como se puede apreciar en la Figura 5.1.

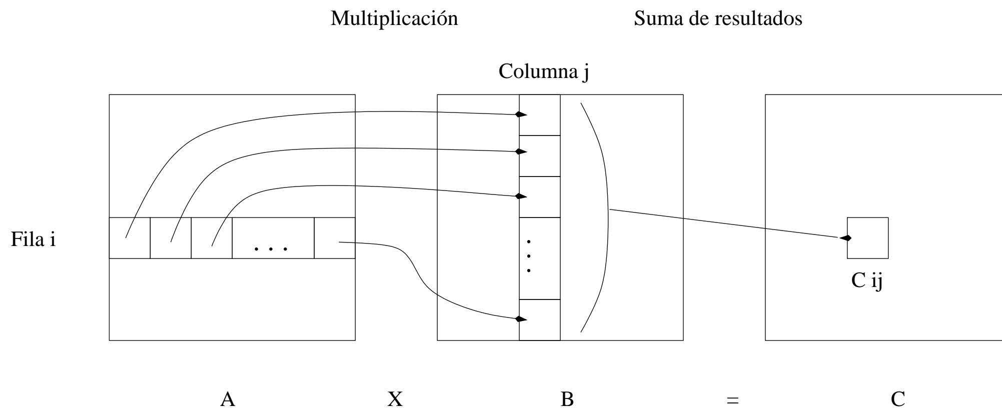


Figura 5.1: Multiplicación de matrices

Para esta aplicación, en DLML cada elemento de la lista contiene una fila de A y una columna B, así cada elemento de la lista contiene los datos necesarios para calcular un elemento de la matriz resultante C.

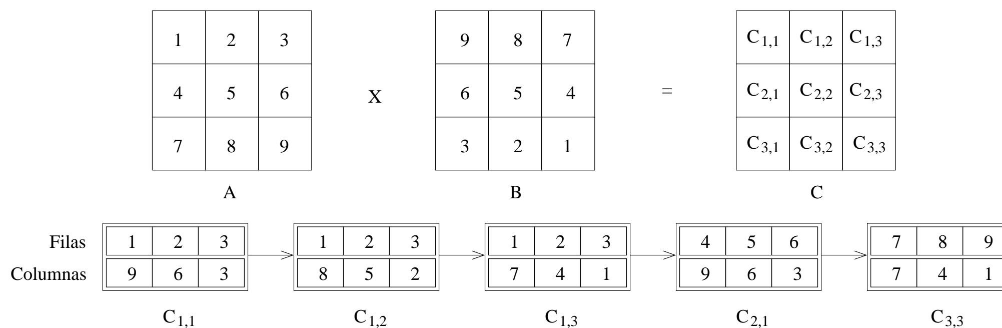


Figura 5.2: Manejo de la lista de datos en la multiplicación de matrices

En la Figura 5.2 se presenta la forma en que los datos de dos matrices de tamaño 3 x 3 son organizados mediante listas.

Al inicio de la ejecución, a la lista se insertan elementos que contienen una fila y una columna de las matrices a multiplicar. El número de elementos a insertar depende del tamaño de las matrices y no se incrementa durante la ejecución, por lo que las operaciones de inserción sólo se realizan al inicio de la ejecución. Una vez que se han insertado todos los elementos, el procesamiento de los datos consiste en obtener un elemento de la lista y realizar las operaciones requeridas para obtener un elemento de la matriz resultante. La ejecución finaliza una vez que la lista queda vacía.

5.1.2. N-Reinas

El problema consiste en colocar N reinas en un tablero de ajedrez de tamaño N x N sin que se ataquen entre sí [1]. En tableros de dimensión 1, 2 y 3 no existe solución, sin embargo, a partir de 4 existen soluciones que se incrementan conforme al tamaño del tablero aumenta. En la Tabla 5.1 se muestran las soluciones para tamaños de tablero 4 hasta 25.

Tablero	Soluciones		Tablero	Soluciones
4	2		15	2,279,184
5	10		16	14,772,512
6	4		17	95,815,104
7	40		18	666,090,624
8	92		19	4,968,057,848
9	352		20	39,029,188,884
10	724		21	314,666,222,712
11	2,680		22	2,691,008,701,644
12	14,200		23	24,233,937,684,440
13	73,712		24	227,514,171,973,736
14	365,596		25	2,207,893,435,808,352

Tabla 5.1: Soluciones conocidas para el problema de las N-Reinas

Para resolver este problema, las posibles soluciones pueden ser modeladas mediante un árbol de búsqueda. En el árbol cada nivel de profundidad representa la columna del tablero y los números que aparecen ahí corresponden al renglón colocado en esa columna. En la Figura 5.3 se muestran las dos únicas soluciones para un tablero con N=4, y su árbol de búsqueda correspondiente. En el primer nivel del árbol (columna 1), es posible colocar una reina por cada renglón, a partir de este punto se generan 4 posibles soluciones a explorar. En el segundo nivel, se colocan reinas sólo en los renglones en que no se ataquen con las reinas del primer nivel, aquellas posibles soluciones que no cumplan con esta condición son descartadas y el resto se consideran nuevas soluciones a explorar. Este procedimiento se realiza hasta encontrar las dos únicas soluciones para el tablero con N=4.

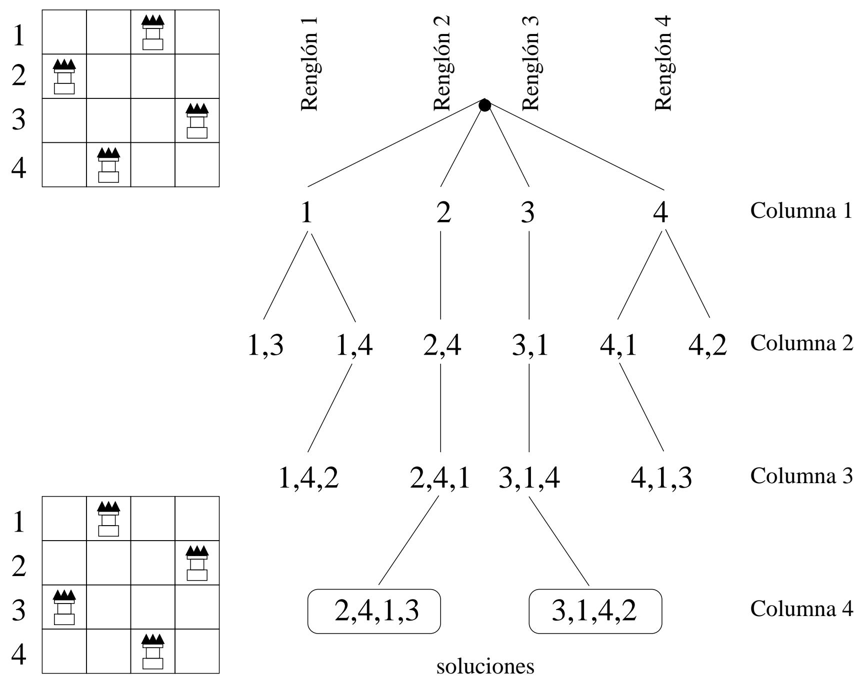


Figura 5.3: Soluciones y árbol de búsqueda en 4 Reinas

En DLML, el árbol puede ser representado a través de una lista de datos que contiene las posibles soluciones a explorar. Para ejemplificar la representación del árbol mediante listas y cómo es que se realiza el manejo de la lista durante la exploración del árbol, se presenta la Figura 5.4.

Considerando nuevamente un tablero con N=4, cada elemento de la lista contiene una posible solución a ser explorada formada por los número de los renglones en los que se coloca una reina (Figura 5.4.a). Como se mencionó anteriormente, al inicio de la ejecución existen 4 posibles soluciones a explorar, por ello en la lista se insertan 4 elementos (Figura 5.4.b). El siguiente paso es tomar al primer elemento de la lista (en este caso con renglón 1), este elemento puede descomponerse en dos posibles soluciones: (1,3) y (1,4), mismas que se insertan en la lista (Figura 5.4.c). Nuevamente se toma el primer elemento de la lista (1,3), sin embargo, al analizar esta posible solución se determina que no se pueden generar más posibles soluciones a partir de ella y se descarta (Figura 5.4.d). Este procedimiento continua hasta que la lista no tiene más elementos encontrando las dos únicas soluciones: (2,4,1,3) y (3,1,4,2) (Figura 5.4.e).

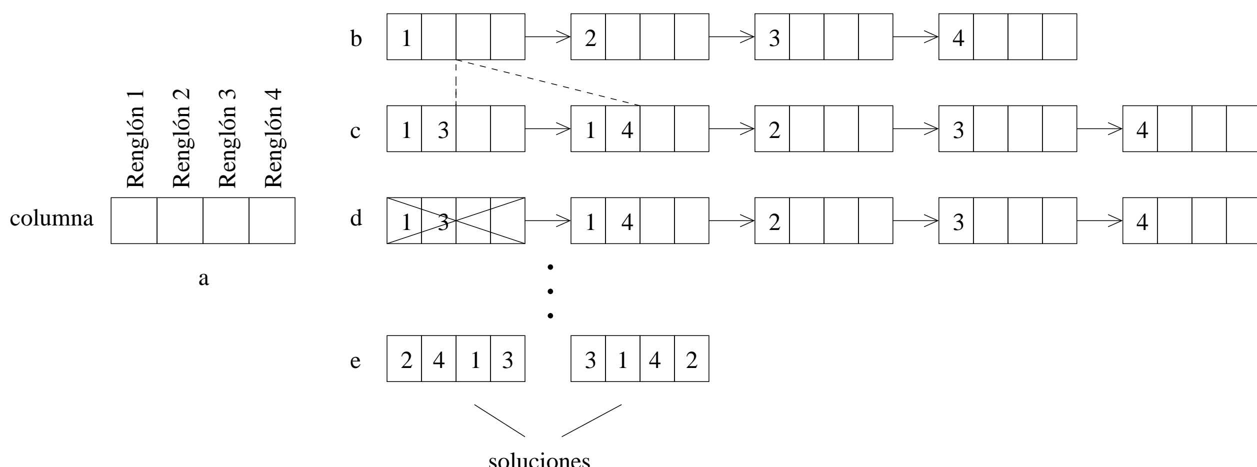


Figura 5.4: Manejo de la lista de datos en la exploración del árbol de búsqueda

Las principal característica de este problema es que las posibles soluciones a explorar son generadas durante la solución del propio problema (generación dinámica de carga), por ello resulta un excelente candidato para ser paralelizado e incorporar un algoritmo de balance de carga.

En la siguiente sección se presenta la infraestructura utilizada para realizar las pruebas a los algoritmos propuestos.

5.2. Infraestructura

La infraestructura utilizada para ejecutar las aplicaciones consta de dos clusters pertenecientes a la Universidad Autónoma Metropolitana Iztapalapa (UAMI). A continuación se detallan las características de cada cluster.

Cluster heterogéneo Pacífico

El cluster Pacífico se encuentra ubicado en el Laboratorio de Sistemas Distribuidos de la UAMI. Se trata de un cluster heterogéneo que cuenta con 40 cores. Su característica heterogénea fue la principal motivación para utilizarlo en las pruebas realizadas, pues permite analizar la manera en que la carga es distribuida entre cores con distintas capacidades de procesamiento. En la Tabla 5.2 se muestran sus características.

Nodos	Proc. por nodo	Sis. Operativo	RAM por nodo	Comunicación
4	2 Dual core 3 GHz	Linux Centos	2 GB	Gigabit Ethernet
6	1 Quad core 2.4 GHz	Linux Centos	4 GB	Gigabit Ethernet

Tabla 5.2: Cluster heterogéneo Pacífico

Cluster homogéneo Aitzaloa

Este cluster que pertenece al Laboratorio de Supercómputo y Visualización en Paralelo de la UAMI, ubicado en la posición 439 del Top500 en Junio de 2009 [20]. Se trata de un cluster con 2160 cores con misma capacidad de procesamiento. Se utilizó principalmente para realizar pruebas que permitieron analizar el comportamiento de los algoritmos con respecto a la escalabilidad y tiempo de respuesta. En la Tabla 5.3 se muestran sus características.

En la siguiente sección se presentan los resultados obtenidos.

Nodos	Proc. por nodo	Sis. Operativo	RAM por nodo	Comunicación
270	8 Intel Xeon Quad-Core 3 Ghz	Linux Centos	16 GB	Gigabit Ethernet / Infiniband

Tabla 5.3: Cluster homogéneo Aitzaloa

5.3. Resultados

En esta sección presentamos los resultados que hemos obtenido después de ejecutar las aplicaciones con los tres algoritmos de balance en ambos clusters, iniciamos con los resultados correspondientes a la aplicación que resuelve el problema de las N-Reinas.

5.3.1. Tiempos de respuesta en la aplicación de las N-Reinas

La Tabla 5.4 muestra los tiempos de respuesta en minutos de la aplicación que resuelve el problema de las N-Reinas. Las ejecuciones se realizaron en el cluster heterogéneo Pacífico usando 40 cores.

Tamaño de tablero	Global	Toroide	Árbol binario
14	0.0622	0.0551	0.041
15	0.2185	0.1794	0.1497
16	1.1362	0.8022	0.6805
17	6.4169	4.7145	4.52
18	36.0361	34.6461	35.0545
19	298.8955	286.9514	293.2508

Tabla 5.4: Tiempos de respuesta obtenidos en minutos para la aplicación N-Reinas con N=14 y N=19

Los resultados muestran que los algoritmos propuestos reducen los tiempos de respuesta

con respecto a los del algoritmo global. En el mejor caso, el algoritmo toroide con N=19 reduce el tiempo en un 4 % comparado con el algoritmo global, que representa una diferencia de aproximadamente 12 minutos.

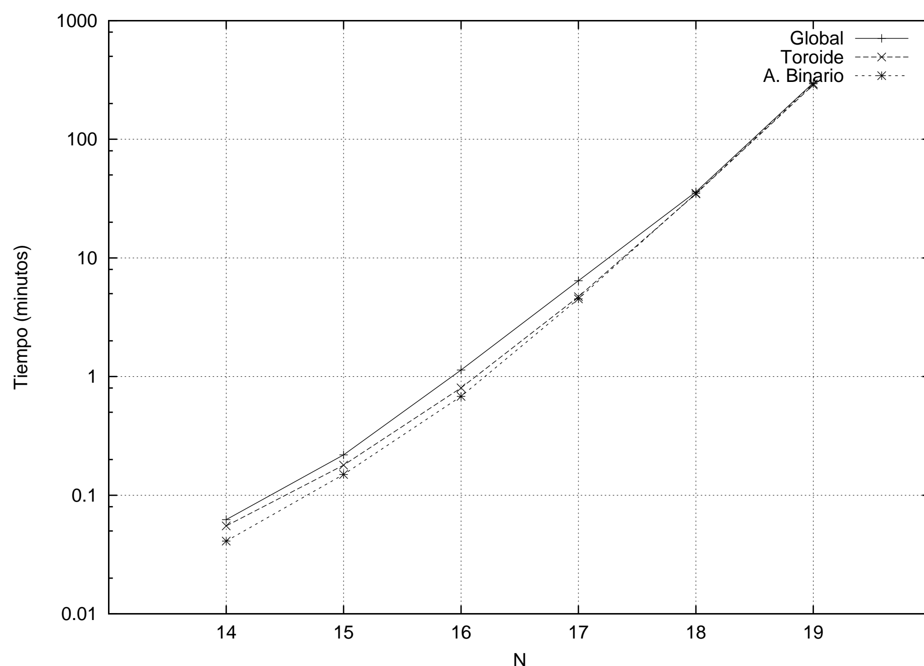


Figura 5.5: Tiempos de respuesta en aplicación N-Reinas con N=14-19 con cluster heterogéneo

En la Figura 5.5 se pueden observar gráficamente los resultados para N variando desde 14 hasta 19. El eje Y corresponde al tiempo de respuesta obtenido en segundos en escala logarítmica y el eje X al tamaño del tablero representado por N.

5.3.2. Distribución de carga en N-Reinas

Otro de los aspectos interesantes a analizar es cómo los algoritmos distribuyen la carga entre los cores que integran el sistema, recordemos que el equilibrio de carga es uno de los objetivos fundamentales de los algoritmos balanceadores.

Para mostrar la forma en que la carga es distribuida se ha elegido la aplicación N-Reinas por su característica de generar carga a tiempo de ejecución, que representa un factor de

desbalance, y un cluster heterogéneo como infraestructura de ejecución, en donde 16 de los 40 cores que lo integran tienen mayor capacidad de procesamiento, que también representa un factor de desbalance.

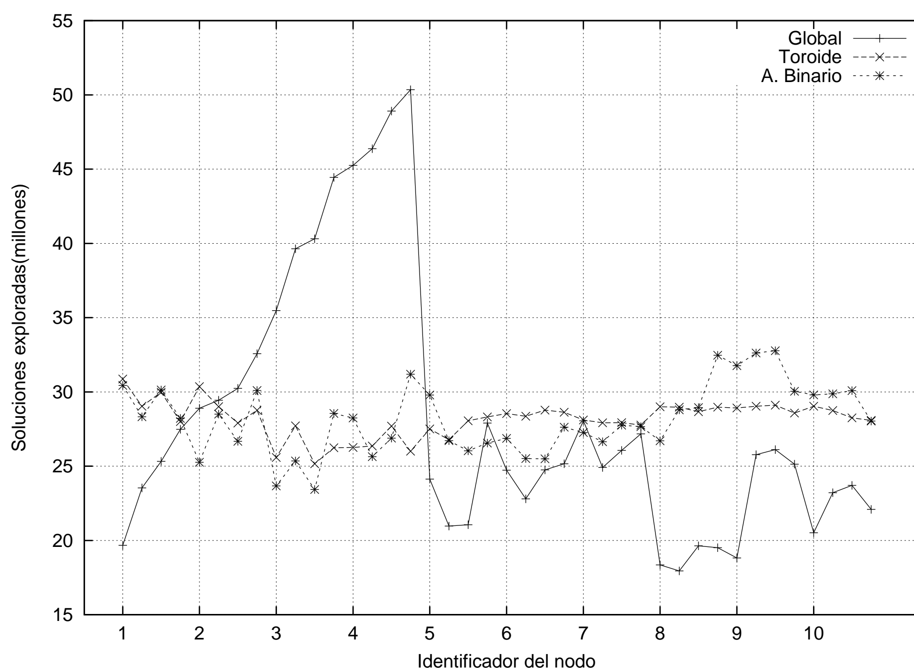


Figura 5.6: Distribución de carga con algoritmos global, toroide y árbol binario en aplicación N-Reinas con N=16

Los resultados de la Figura 5.6 corresponden al número de soluciones exploradas, es decir la cantidad de carga procesada por cada procesador, los resultados fueron obtenidos de las ejecuciones de los 3 algoritmos en la aplicación de las N-Reinas con N=16.

Se puede apreciar que el algoritmo global presenta un gran desequilibrio en cuanto a la cantidad de carga procesada por procesador, se puede notar que existe un mayor desequilibrio en los nodos 1, 2, 3 y 4, que son aquellos que cuentan con cores con mayor capacidad de procesamiento, a partir del nodo 5 se observa que hay una distribución más uniforme pero aún no es la más adecuada. En el caso del algoritmo de árbol binario, tiene una mejor distribución con respecto al algoritmo global, pero también presenta cierto desequilibrio. El algoritmo toroide es el que ha presentado los mejores resultados. Aunque se aprecia un ligero

desequilibrio en la distribución, prácticamente la cantidad de carga procesada es similar en todos los cores, por lo que este algoritmo supera a los dos anteriores, pues los factores de desbalance que representa el uso de un cluster heterogéneo y la generación dinámica de carga no han sido un factor que impida una buena distribución de carga con este algoritmo.

En la figura 5.7 se presentan los resultados obtenidos para la misma aplicación pero con N=17.

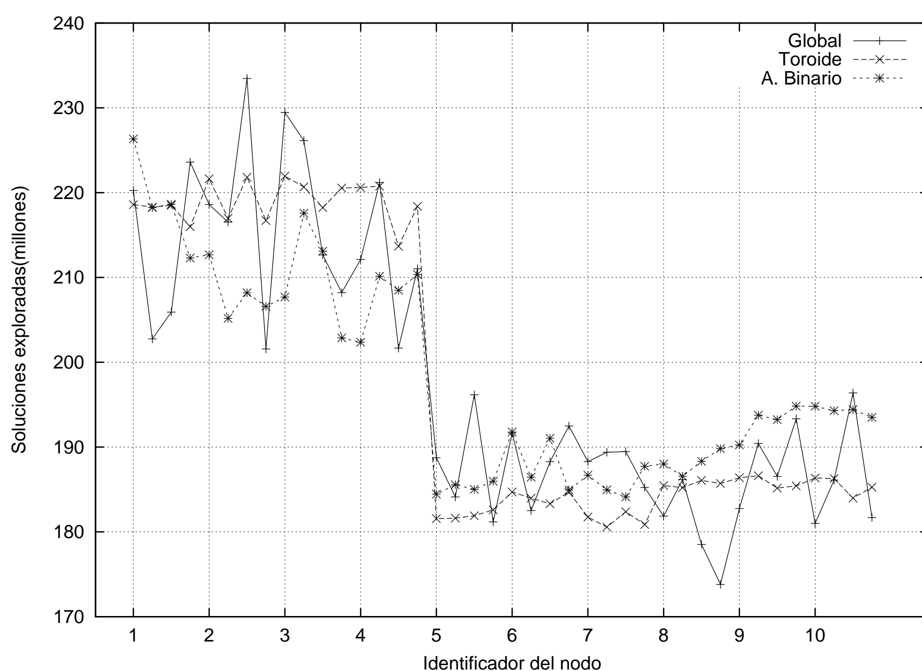


Figura 5.7: Distribución de carga con algoritmos global, toroide y árbol binario en aplicación N-Reinas con N=17

Estos resultados permiten apreciar nuevamente que el algoritmo que presenta una mejor distribución de carga es el toroide, pero también es posible apreciar que los cores con mayor capacidad de procesamiento son los que más carga han procesado, lo que nos muestra que existe una mejora en cuanto al uso eficiente de los recursos de cómputo con el algoritmo toroide.

De estos resultados, se ha calculado la desviación estándar. En la Tabla 5.5 se muestra la desviación estándar para los tres algoritmos en cada región del cluster heterogéneo. Se

observa que el algoritmo toroide obtiene la menor desviación presentando un mejor balance de carga, seguido por el algoritmo de árbol binario y finalmente el algoritmo global. En la siguiente subsección se presentan los resultados referentes a la escalabilidad.

Algoritmo	Nodo 1-4	Nodo 5-10
Global	9,978,768	5,556,479
Toroide	2,350,132	1,952,682
Árbol Binario	6,414,427	3,812,180

Tabla 5.5: Desviación estándar (σ) de las soluciones exploradas con N=17

5.3.3. Escalabilidad en N-Reinas

Además de los resultados mostrados anteriormente de las ejecuciones sobre el cluster heterogéneo, se han realizado ejecuciones en el cluster Aitzaloa que cuenta con un mayor número de cores, lo cual permitió analizar la escalabilidad de los algoritmos. Nuevamente estas pruebas han sido realizadas con la aplicación de las N-Reinas para N=16 y N=17. En cada ejecución realizada se incrementa al doble el número de cores partiendo desde 8, así tenemos ejecuciones hasta con 1024 cores.

En la Figura 5.8 se observan los resultados de la aplicación con 16-Reinas. Es posible observar que para los tres algoritmos, prácticamente los tiempos de respuesta son iguales con 8 y 16 cores, pero cuando el número de cores es 32, el algoritmo toroide presenta menor tiempo de respuesta con respecto al algoritmo de árbol binario y global, manteniendo esta tendencia conforme se incrementa el número de cores.

Uno de los puntos a destacar es que el algoritmo global pierde escalabilidad cuando el número de cores se incrementa de 64 a 128, esto es consecuencia del incremento de mensajes requeridos para balancear carga. A partir de este punto la tendencia ascendente de los tiempos de respuesta se mantiene conforme se incrementa el número de cores. En el caso del algoritmo de árbol binario, presenta mejores resultados con respecto al algoritmo global, pero no así, si

se compara con el algoritmo toroide. El algoritmo toroide empieza a perder eficiencia cuando el número de cores es 1024.

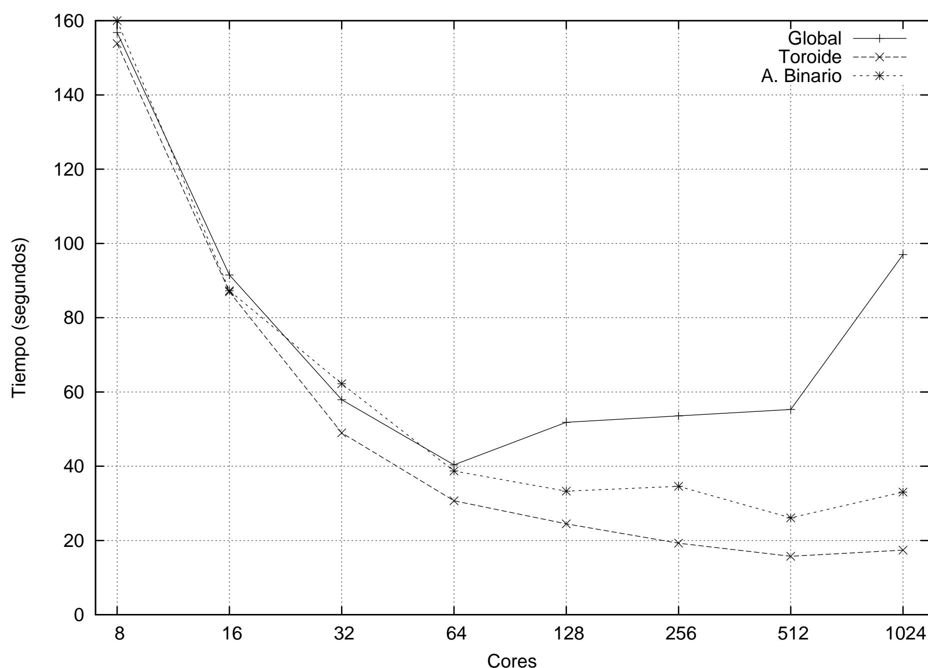


Figura 5.8: Tiempos de respuesta y escalabilidad con aplicación N-Reinas con N=16

Los resultados que se muestran en la Figura 5.9 corresponden a la ejecución de esta misma aplicación pero con un tablero con N=17. Los resultados muestran nuevamente que el algoritmo toroide presenta tiempos de respuesta menores, seguido por el algoritmo de árbol y por último el algoritmo global con los tiempos de respuesta más altos. Cuando el número de cores es 1024, el algoritmo toroide presenta una mejora del 87 % con respecto al tiempo que presenta el algoritmo global, lo cual es una mejora considerable.

En base a estos resultados, hay que destacar que el hecho de aumentar el tamaño del problema implica que la pérdida de escalabilidad de los algoritmos se presente con un mayor número de cores, tal es el caso del algoritmo global, ahora presenta el menor tiempo de respuesta con 128 cores en vez de los 64 para el tablero con N=16.

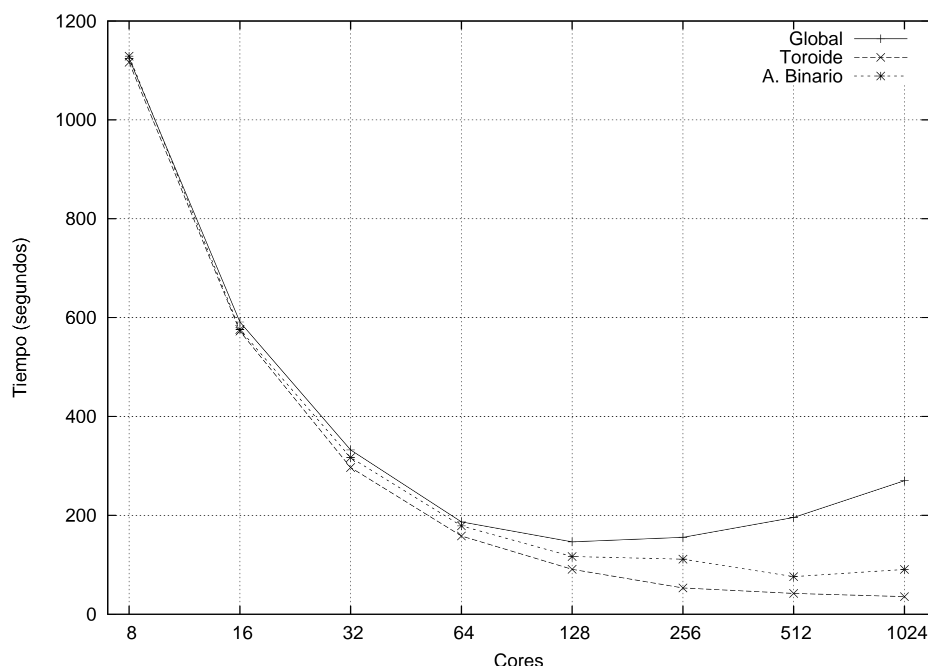


Figura 5.9: Tiempos de respuesta y escalabilidad en aplicación N-Reinas con N=17

5.3.4. Mesajes utilizados en N-Reinas

Como se ha podido observar en la subsección anterior, a partir de 128 cores, la reducción del tiempo de respuesta que se logra con los algoritmos propuestos es más notable. Esto se debe a que a mayor número de cores, el número de mensajes requeridos en el algoritmo global se incrementa en O(n^2), aumentando así el tiempo de respuesta. Para ello, se ha medido el número de mensajes utilizados por la aplicación con cada uno de los algoritmos utilizando 128 cores para N=16 y N=17. Los resultados que encontramos son los siguientes.

La Figura 5.10 muestra el número de mensajes utilizados (mensajes recibidos por core) para N=16. Se puede observar una clara diferencia entre la cantidad de mensajes requeridos por el algoritmo global y los algoritmos propuestos. El promedio de mensajes por core para el algoritmo global es de 17897, mientras de que para el algoritmo toroide es de 1039 y 2102 para el árbol binario, lo cual representa un reducción en el número de mensajes utilizados con respecto al algoritmo global del 94.2 % y 88.26 % respectivamente.

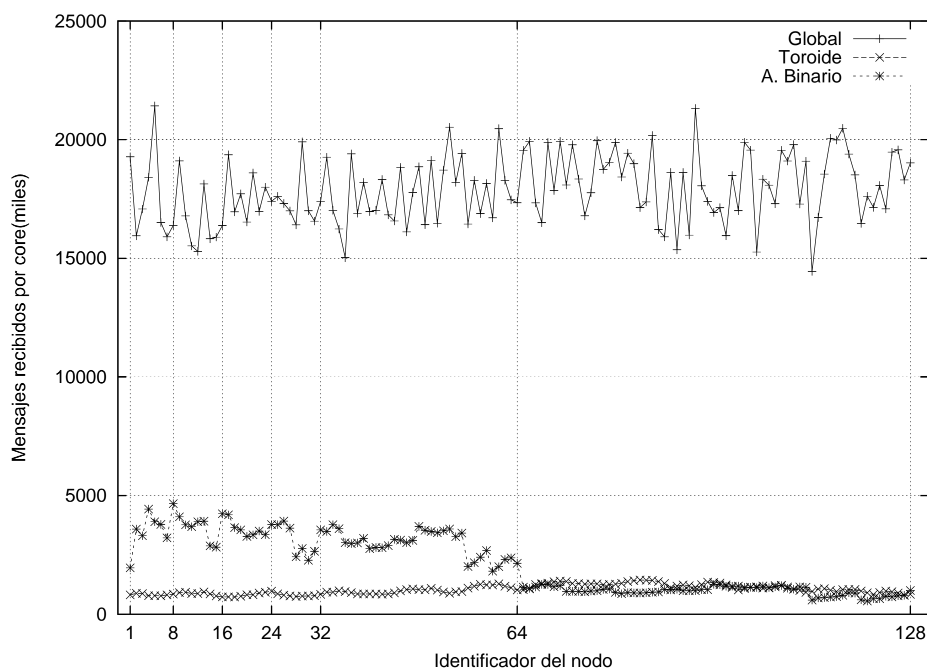


Figura 5.10: Mensajes utilizados en aplicación N-Reinas con N=16

Para N=17, la Figura 5.11 muestra nuevamente que el algoritmo global requiere de un número mayor de mensajes, cuyo promedio por core es de 47698, mientras que el promedio para el toroide es de 1547.27 y de 1869.85 para el árbol binario. Esto representa un reducción en mensajes del 3.24 % y 3.92 % respectivamente.

Además de las ejecuciones realizadas con la aplicación que resuelve el problema dinámico de las N-Reinas, también se realizaron ejecuciones con la aplicación (estática) que permite la multiplicación de matrices, el objetivo es analizar el comportamiento de los algoritmos con aplicaciones en donde no se genera más carga a tiempo de ejecución.

5.3.5. Tiempos de respuesta en Multiplicación de Matrices

Como se mencionaba anteriormente, las características de esta aplicación es que la cantidad de carga a procesar se conoce con anticipación y que no se genera más carga a tiempo de ejecución. El hecho de conocer la cantidad de carga previamente, permite distribuirla entre

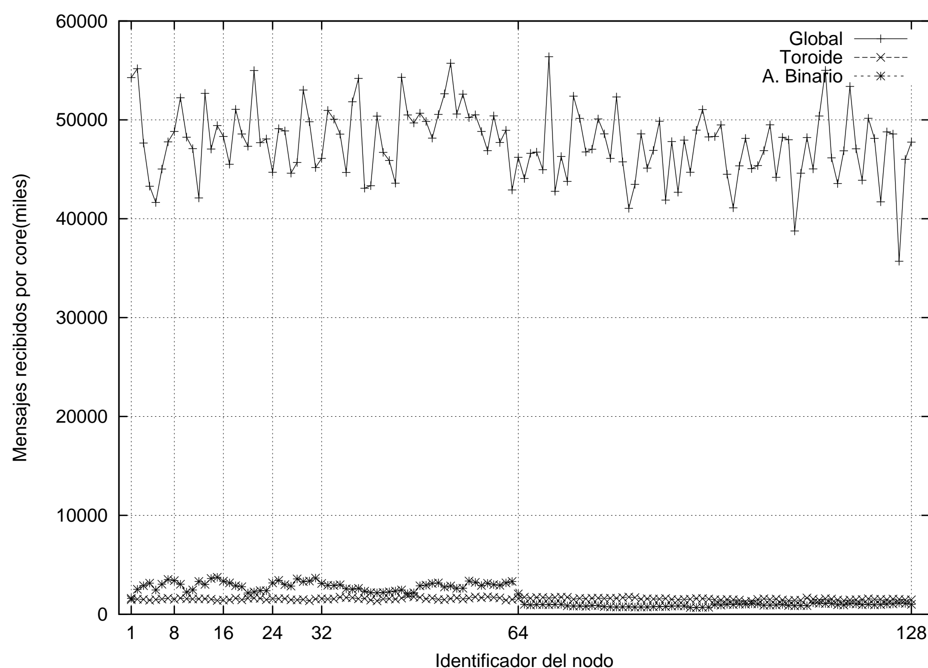


Figura 5.11: Mensajes utilizados en aplicación N-Reinas con N=17

el total de cores al inicio de la ejecución, distinto a lo que sucede con la aplicación N-Reinas en la que la cantidad de carga a procesar no se conoce puesto que ésta se va generando a tiempo de ejecución y por lo tanto no es posible distribuir carga hacia todos los cores al inicio de la ejecución. Los resultados que se muestran a continuación, corresponden a multiplicar dos matrices cuadradas de tamaño 1000 y 1200 usando desde 8 hasta 1024 cores.

En la Figura 5.12 se muestra los resultados correspondientes a la multiplicación de matrices de tamaño 1000.

Los resultados muestran que el algoritmo global tiene mejor desempeño con 8 y hasta 32 cores, a partir de 32 cores empieza a perder escalabilidad. Por otro lado, el algoritmo toroide presenta una mejora significativa a partir de 64 cores, sin embargo, en esta ocasión el algoritmo con mejores resultados es el árbol binario. Al principio el árbol binario tiene el mismo comportamiento que el algoritmo toroide, a partir de 256 cores la diferencia es mayor. Creemos que esta diferencia se debe al menor grado de conectividad de la topología de árbol, pues al incrementarse el número de cores, el tiempo de ejecución de cada etapa del algoritmo

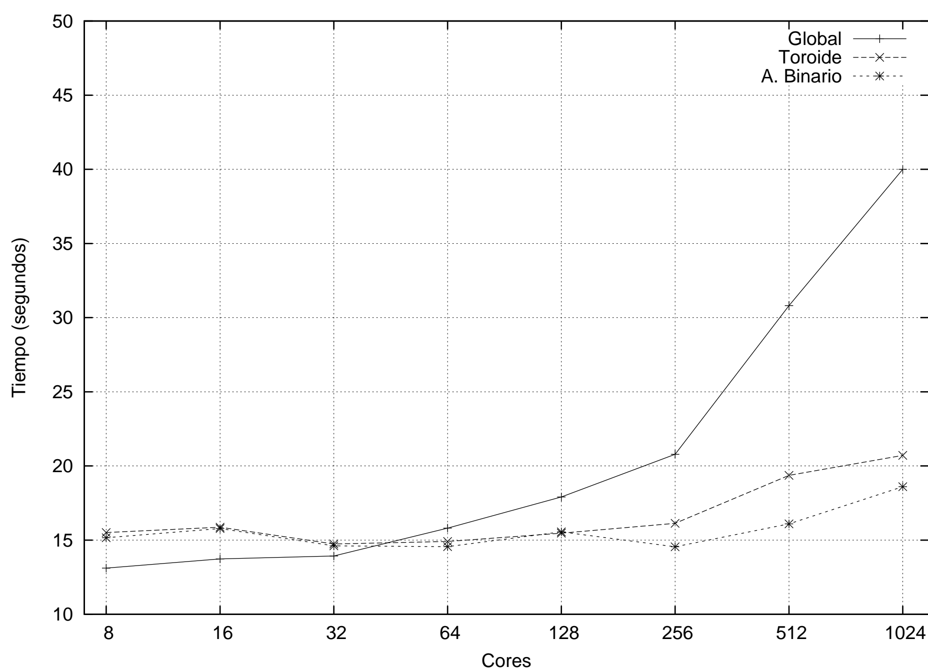


Figura 5.12: Tiempos de respuesta para multiplicación de matrices de tamaño 1000 x 1000

se también se incrementa.

La Figura 5.13 muestra los resultados para la multiplicación de matrices de tamaño 1200, los resultados muestran que aunque se ha aumentado la cantidad de carga a procesar, el algoritmo global pierde escalabilidad a partir de 32 cores. En el caso del algoritmo toroide, con 8 y 32 cores presenta un mayor tiempo de respuesta que se reduce conforme se incrementa el número de cores hasta 256, a partir de este punto empieza a perder escalabilidad. Para el árbol binario, es el que muestra mejores resultados, aunque con 8 y 32 cores tiene mayor tiempo de respuesta que el algoritmo global, este también se reduce al aumentar el número de cores. A diferencia del algoritmo toroide, la pérdida de escalabilidad es mínima.

En el capítulo siguiente se presentan las conclusiones y trabajo a futuro.

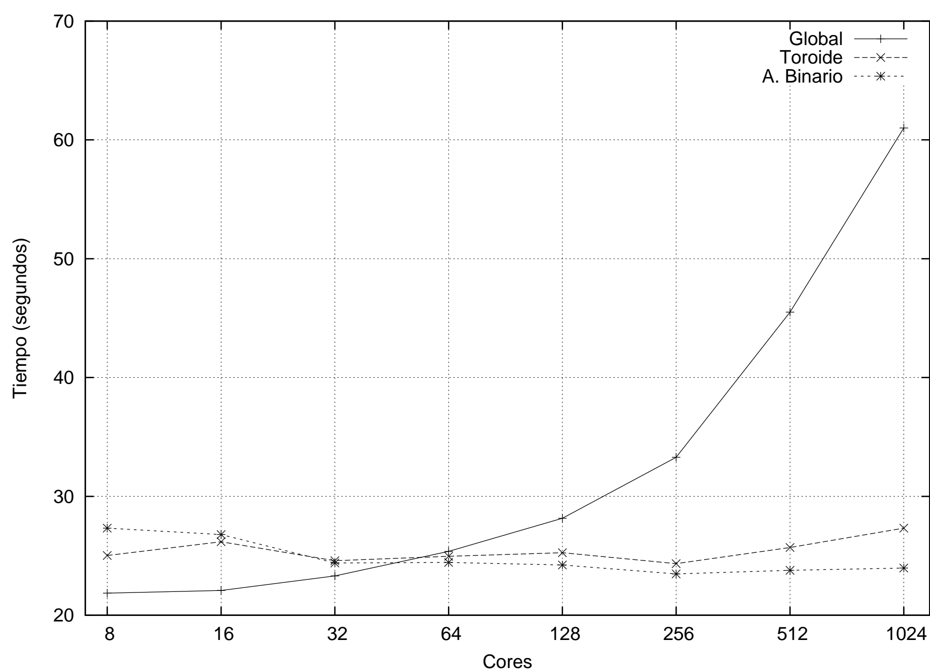


Figura 5.13: Tiempos de respuesta para multiplicación de matrices de tamaño 1200 x 1200

Capítulo 6

Conclusiones y trabajo a futuro

6.1. Conclusiones

En esta tesis se presentaron dos algoritmos de balance de carga con manejo de información parcial. Los algoritmos son una propuesta para balancear carga utilizando sólo una parte de la información del estado de carga del sistema. El uso de información parcial permitió reducir el número de mensajes requeridos para balancear carga y se logró el objetivo de reducir los tiempos de respuesta y el problema de escalabilidad.

Los algoritmos propuestos fueron implementados en la herramienta DLML con el fin de verificar su funcionamiento y comparar su desempeño con el algoritmo con manejo de información global que tiene de origen la herramienta.

Los experimentos fueron realizados utilizando dos aplicaciones: N-Reinas y Multiplicación de matrices. La primera aplicación tiene como característica la generación dinámica de carga y la segunda es considerada estática pues no genera más carga a tiempo de ejecución. La infraestructura utilizada estuvo compuesta por dos clusters: cluster heterogéneo Pacifico y cluster homogéneo Aitzaloa.

Los resultados obtenidos con respecto a la reducción de los tiempos de respuesta, escalabilidad y distribución de carga fueron los siguientes:

Tiempos de respuesta

- En el caso de la aplicación de las N-Reinas (aplicación dinámica), los algoritmos: toroide y árbol binario, presentaron menor tiempo de respuesta con respecto al algoritmo global.

Esta reducción fue más significativa cuando el tamaño del tablero se incrementa y es independiente del cluster utilizado.

- Para la aplicación de multiplicación de matrices (aplicación estática), los resultados mostraron que el algoritmo global tiene mejores tiempos de respuesta cuando el número de cores utilizados está entre 8 y 32. Sin embargo, al incrementar el número de cores sus tiempos de respuesta se incrementan considerablemente, mientras que los tiempos de los algoritmos propuestos se reducen, manteniendo esta tendencia hasta 256 y 512 cores en el algoritmo toroide y árbol binario respectivamente. Superando así los tiempos de respuesta del algoritmo global.

Escalabilidad

- Para la aplicación de las N-Reinas, se puede observar que los algoritmos propuestos logran reducir el problema de escalabilidad del algoritmo global. En los resultados se observa que para un tablero con N=16 y N=17, el desempeño del algoritmo global se degrada a partir de 64 y 128 cores respectivamente, mientras que nuestros algoritmos mantienen un buen desempeño. Sólo hasta 1024 cores se observa que se presenta ligeramente un problema de escalabilidad para las pruebas en donde se considera el tablero con N=16, mientras que para N=17, el desempeño de los algoritmos no se degrada.

- Para la aplicación Multiplicación de Matrices, el problema de escalabilidad es notable para el algoritmo global y se presenta a partir de 16 cores para tamaños de matrices cuadradas 1000 y 1200. En el caso de los algoritmos propuestos, estos presentan mejor desempeño. El problema de escalabilidad se presenta cuando se utilizan 256 cores, aún cuando los algoritmos propuestos también pierden eficiencia a mayor número de cores, esta no es tan significativa, contrario a lo que sucede con el algoritmo global que pierde eficiencia de forma exponencial.

Distribución de carga

- Para comparar la forma en que los algoritmos distribuyen carga, se ocupó la aplicación de N-Reinas. Los resultados mostraron que los algoritmos propuestos tienen una mejor distribución de carga. Cuando la plataforma de ejecución es el cluster heterogéneo Pacifico, se observa que los cores con mayor capacidad de procesamiento son los que han procesado más carga, sobre todo cuando el algoritmo utilizado es el toroide, seguido del árbol binario y finalmente el algoritmo global. Además de que se observa una mayor uniformidad en la cantidad de carga procesada por cada core. Cuando el cluster utilizado es Aitzaloa, la cantidad de carga procesada por core presenta mayor uniformidad con el algoritmo toroide, seguido del árbol binario y finalmente el algoritmo global.

Debido a que la implementación de los algoritmos se realizó en DLML, actualmente esta herramienta cuenta con tres algoritmos de balance de carga que pueden ser elegibles en el desarrollo de aplicaciones paralelas. A continuación se presenta el trabajo a futuro.

6.2. Trabajo a futuro

Como trabajo a futuro se propone implementar nuevas topologías de comunicación tales como hipercubo, árboles distintos al binario, etc. esto con el fin de identificar si existen topologías más eficientes a la del toroide que en términos generales es la que ha presentado mejores resultados.

Por otra parte, el diseño de los algoritmos no contempla tolerancia a fallas, en caso de que algún procesador que integra el sistema presente alguna falla, la ejecución de la aplicación simplemente no podrá continuar. Cualquier falla (sobre todo cuando se trata de aplicaciones cuyo tiempo de procesamiento es considerable) representaría la pérdida de tiempo de procesamiento y uso inútil de recursos de cómputo.

Actualmente el diseño de los algoritmos contempla que un procesador sea el encargado de iniciar las etapas de búsqueda global de carga y terminación. Este enfoque centralizado representa un punto único de falla en el sistema, sería conveniente desarrollar una nueva

estrategia en donde las etapas antes mencionadas no sean iniciadas por un único procesador.

Por ahora el diseño de los algoritmos está pensado para su ejecución sobre clusters, por lo que su buen desempeño en otro tipo de plataforma de ejecución (por ejemplo en Grids) no está garantizado, por lo que sería deseable desarrollar una versión que incorpore la capacidad de ejecución en otra plataforma.

Apéndice A

En este apéndice se muestra el pseudocódigo completo de los dos algoritmos.

Algoritmo Toroide

```
<1>  al estar descargado y balance_iniciado = 0 efectúa
<2>     envía REQ_INFO_CARGA a todo k en vecinos
<3>     balance_iniciado <- 1

<4>  al recibir REQ_INFO_CARGA desde k en vecinos efectúa
<5>     envía RESPUESTA_INFO_CARGA a k en vecinos con el indice_carga

<6>  al recibir RESPUESTA_INFO_CARGA(indice_carga) de K en vecinos efectúa
<7>     respuesta_carga <- respuesta_carga + 1
<8>     guarda indice_carga de k
<9>     si respuesta_carga == 4 /*Si todos han respondido*/
<10>        entonces respuesta_carga <- 0
<11>                determina al k en vecinos más cargado
<12>                si existe algún k en vecinos más cargado
<13>                   entonces envía DAME_CARGA a k en vecinos más cargado
<14>                   otro     si id == s
<15>                               entonces si primera_busqueda == 0
<16>                                  entonces envía BUSCA_CARGA(0)
                                                   a todo k en vecinos
```

```
<17>                                                primera_busqueda <- 1
<18>                                      otro envía BUSCA_CARGA(0) a todo k en hijos
<19>                                   otro     envía DESCARGADO a todo k en vecinos
<20>    desecha los índices de carga recibidos

<21> al recibir DAME_CARGA desde k en vecinos efectúa
<22>    carga_enviar <- indice_carga / X
<23>    si carga_enviar > 1
<24>       entonces indice_carga <- (indice_carga - carga_enviar)
<25>                envía CARGA(carga_enviar) a k en vecinos
<26>       otro     envía CARGA(0) a k en vecinos
<27>    carga_enviar <- 0

<28> al recibir CARGA(carga) de k en vecinos efectúa
<29>    si carga > 0
<30>       entonces indice_carga <- carga
<31>                balance_iniciado <- 0
<32>                envía BALANCE a todo k en vecinos_descargados
<33>                borra k en vecinos_descargados
<34>       otro     ejecuta <2> y <3> de la etapa de inicialización

<35> al recibir DESCARGADO de k en vecinos efectúa
<36>    guarda k en vecinos_descargados

<37> al recibir BALANCE desde k en vecinos efectúa <2> y <3>
     de la etapa de inicialización

<38> al recibir BUSCA_CARGA(indice_carga) de k en vecinos efectúa
<39>    carga_encontrada <- carga_encontrada + indice_carga
<40>    mesajes_busca_carga <- mesajes_busca_carga + 1
```

```
<41>    si id == s
<42>       entonces si mensajes_busca_carga == # hijos
<43>                entonces mensajes_busca_carga <- 0
<44>                         si carga_encontrada == 0
<45>                            entonces envía TERMINA a todo k en hijos
<46>                            otro     ejecuta <2> y <3>
<47>                                     carga_encontrada <- 0
<48>       otro     si primera_busqueda == 0 /*busca carga por primera vez*/
<49>                   entonces si mensajes_busca_carga == 1
<50>                               entonces envía ERES_MI_PADRE a k en vecinos
<51>                                        padre <- k
<52>                                        reexpide BUSCA_CARGA(indice_carga)
                                                     a vecinos excepto k
<53>                            si mensajes_busca_carga == 4
<54>                               entonces mensajes_busca_carga <- 0
<55>                                        primera_busqueda <- 1
<56>                                        envía BUSCA_CARGA(carga_encontrada)
                                                  a padre
<57>                                        carga_encontrada <- 0
<58>                   otro si mensajes_busca_carga == 1 /*busca carga sobre árbol*/
<59>                         entonces si # hijos > 0
<60>           /*desciende*/               entonces reexpide
                                              BUSCA_CARGA(indice_carga) a hijos
<61>           /*Es hoja, asciende*/       otro  mensajes_busca_carga <- 0
<62>                                             envía BUSCA_CARGA(indice_carga)
                                                       a padre
<63>                                             carga_encontrada <- 0
<64>                        si # hijos > 0
<65>                           entonces si mensajes_busca_carga == (# hijos + 1)
<66>                                    entonces mensajes_busca_carga <- 0
```

```
<67>                                              envía
              /*asciende*/                        BUSCA_CARGA(carga_encontrada)
                                                  a padre
<68>                                              carga_encontrada <- 0
<69>

<70> al recibir ERES_MI_PADRE de k en vecinos agrega k a hijos

<71> al recibir TERMINA de padre efectúa
<72>    si # hijos > 0
<73>       entonces reexpide TERMINA a todo k en hijos
<74>       otro envía RESULTADO(resultado_parcial) a padre
<75>            termina

<76> al recibir RESULTADO (resultado) de k en hijos efectúa
<77>    resultados_recibidos <- (resultados_recibidos + 1)
<78>    resultado_acumulado <- resultado_acumulado + resultado
<79>    si resultados_recibidos == # hijos
<80>       entonces si id == s
<81>                   entonces imprime (resultado_acumulado + resultado_parcial)
<82>                            termina
<83>                   otro     resultado_acumulado <-
                               (resultado_acumulado + resultado_parcial)
<84>                            envía RESULTADO(resultado_acumulado) a padre
<85>                            termina
```

Algoritmo Árbol Binario

```
<1>  al estar descargado y balance_iniciado = 0 efectúa
<2>     envía REQ_INFO_CARGA a todo k en vecinos
<3>     balance_iniciado <- 1

<4>  al recibir REQ_INFO_CARGA desde k en vecinos efectúa
<5>     envía RESUPESTA_INFO_CARGA a k en vecinos con el indice_carga

<6>  al recibir RESPUESTA_INFO_CARGA(indice_carga) de todo k en vecinos efectúa
<7>     respuesta_carga <- respuesta_carga + 1
<8>     guarda indice_carga de k
<9>     si respuesta_carga == # vecinos
<10>        entonces respuesta_carga <- 0
<11>                determina al k en vecinos más cargado
<12>                si existe algún k en vecinos más cargado
<13>                   entonces envía DAME_CARGA a k en vecinos más cargado
<14>                   otro     si id == raiz_arbol
<15>                               entonces envía BUSCA_CARGA(0) a todo k en ramas
<16>                               otro     envía DESCARGADO a todo k en vecinos
<17>    desecha los índices de carga recibidos

<18> al recibir DAME_CARGA desde k en vecinos efectúa
<19>    carga_enviar <- indice_carga / X
<20>    si carga_enviar > 1
<21>       entonces indice_carga <- (indice_carga - carga_enviar)
<22>                envía CARGA(carga_enviar) a k en vecinos
<23>       otro     envía CARGA(0) a k en vecinos
<24>    carga_enviar <- 0
```

```
<25> al recibir CARGA(carga) de k en vecinos efectúa
<26>    si carga > 0
<27>       entonces indice_carga <- carga
<28>                balance_iniciado <- 0
<29>                envía BALANCE a todo k en vecinos_descargados
<30>                borra k en vecinos_descargados
<31>       otro     ejecuta <2> y <3>

<32> al recibir DESCARGADO de k en vecinos efectúa
<33>    guarda k en vecinos descargados

<34> al recibir BALANCE desde k en vecinos efectúa <2> y <3>

<35> al recibir BUSCA_CARGA(indice_carga) de k en vecinos efectúa
<36>    carga_encontrada <- carga_encontrada + indice_carga
<37>    mesajes_busca_carga <- mesajes_busca_carga + 1
<38>    si id == s
<39>       entonces si mensajes_busca_carga == # vecinos
<40>                entonces mensajes_busca_carga <- 0
<41>                         si carga_encontrada == 0
<42>                            entonces envía TERMINA a todo k en ramas
<43>                            otro     ejecuta <2> y <3>
<44>                                     carga_encontrada <- 0
<45>       otro     si mensajes_busca_carga == 1
<46>                   reexpide BUSCA_CARGA(indice_carga) a todo k en ramas
<47>                si mensajes_busca_carga == # vecinos
<48>                   entonces mensajes_busca_carga <- 0
<49>                            caga_encontrada <- carga_encontrada + indice_carga
<50>                            envía BUSCA_CARGA(carga_encontrada) a raiz
<51>                                             carga_encontrada <- 0
```

```
<52> al recibir TERMINA de raíz efectúa
<53>    si # ramas > 0
<54>       entonces reexpide TERMINA a todo k en ramas
<55>       otro envía RESULTADO(resultado_parcial) a raiz
<56>            termina

<57> al recibir RESULTADO (resultado) de k en ramas efectúa
<58>    resultados_recibidos <- (resultados_recibidos + 1)
<59>    resultado_acumulado <- resultado_acumulado + resultado
<60>    si resultados_recibidos == # ramas
<61>       entonces si id == raiz_arbol
<62>                   entonces imprime (resultado_acumulado + resultado_parcial)
<63>                            termina
<64>                   otro resultado_acumulado <-
                            (resultado_acumulado + resultado_parcial)
<65>                        envía RESULTADO(resultado_acumulado) a raiz
<66>                        termina
```

Referencias

[1] Aiden Bruen and R. Dixon. *The n-queens problem*, Discrete Mathematics, 12(4): 393-395, 1975

[2] A. Plastino, *Balanceamento de carga de aplicacoes paralelas SPMD*, Doctorate thesis, Departament of Computing, Catholic University of Rio de Janeiro, 2000

[3] BOSQUE José L., ROBLES Oscar D.,PASTOR Luis, RODRIGUEZ Angel, *Parallel CBIR implementations with load balancing algorithms*, Journal of parallel and distributed computing , vol. 66, pp. 1062-1075, 2006

[4] Castro G. M.A. *Balance dinámico de carga en un sistema paralelo con memoria distribuida*, Tesis de Maestría, CINVESTAV-México, Departamento de Ingeniería Eléctrica, México, D.F., 2001

[5] M. A. C. García. *Programación con Listas de Datos para Cómputo Paralelo en Clusters*, Tesis de Doctorado, CINVESTAV-México, Departamento de Ingeniería Eléctrica, México, D.F., 2007

[6] Curt Powley, Chris Ferguson, Richard Korf. *Depth-first heuristics search on a SIMD machine*. Artificial Intelligence, 60, 1993

[7] W. Daniel Hillis. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985

[8] Devine K., Boman E., Heaphy R., Hendrickson B., Vaughan C., Zoltan data management services for parallel dynamic applications, Computing in Science and Engineering 4 (2) páginas 90-97, 2002

[9] E. Dijkstra, W. Seijen, A. Gasteren, Derivatio of a termination detection algorithm for a distributed computation, Information Proccesing Letters 16 217-219, 1983

[10] Elie El Ajaltouni, Azzedine Boukerche, Ming Zhang, *An Efficient Dynamic Load Balancing Scheme for Distributed Simulations on a Grid Infrastructure*,pp.61-68, 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications, 2008

[11] Fonlupt, C., Marquet, P. and Dekeyser, J. *Data-parallel load-balancing strategies*. Parallel Computing 24 1665-1684,1998

[12] Gil-Haeng Lee, *Issues of the State Information for Location and Information Policies in Distributed Load Balancing Algorithm*, Euromicro, pp.1067, 25th Euromicro Conference, vol. 1, 1999

[13] KatherineM. Baumgartnet, Ralph Kling, and Benjamin Wah. *A global load balancing strategy for a distributed system*. In IEEE Conference on Distributed Computing Systems, pages 93-102, Hong-Kong, 1998

[14] Hye-Seon Maeng, Hyoun-Su Lee, Tack-Don Han, Sung-Bong Yang, Shin-Duk Kim, *Dynamic Load Balancing of Iterative Data Parallel Problems on a Workstation Clustering*, pp.563, High-Performance Computing on the Information Superhighway, HPC-Asia '97, 1997

[15] Mohammed Javeed Zaki, Wei Li, Srinivasan Parthasarathy, *Customized dynamic load balancing for a network of workstations*, Journal of Parallel and Distributed Computing, 1997

[16] Devine K., Hendrickson B., Boman E., M. St. John, Vaughan C., Zoltan: A Dynamic Load Balancing Library for Parallel Applications; User´s Guide, Sandia National Laboratories, Albuquerque, NM, http://www.cs.sandia.gov/Zoltan/, 2000

[17] Neeraj Nehra, R.B. Patel, V.K. Bhat, *A Framework for Distributed Dynamic Load Balancing in Heterogeneous Cluster*, Journal of Computer Science 3 (1): 14-24, 2007

[18] A. Osman, H. Ammar, *Dynamic Load Balancing Strategies for Parallel Computers*, International Symposium on Parallel and Distributed Computing ISPDC, 2002

[19] A. Segall, *Distributed network protocols*, IEEE Trans. on Information Theory, 29(1):23-35, Jan 1983

[20] Página web del proyecto Top500 dedicado a posicionar los 500 sistemas de cómputo más potentes en el mundo. Página consultada el 9 de Julio de 2009. http://www.top500.org/lists/2008/11.

[21] Peter Kok Keong Loh, Wen Jing Hsu, Cai Wentong, Nadarajah Sriskanthan, *How Network Topology Affects Dynamic Load Balancing*, IEEE Concurrency, vol. 4, no. 3, pp. 25-35, 1996

[22] John A. Stankovic and Inderjit S. Sidhu. *An adaptive bidding algorithm for processes, clusters, and distributed groups*. In Inter. Conf. on Distributed Computing Systems, pp. 49–59. IEEE, 1984.

[23] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc., first edition, August 1998.

UNIVERSIDAD AUTÓNOMA METROPOLITANA

ALGORITMOS DE BALANCE DE CARGA CON MANEJO DE INFORMACIÓN PARCIAL

Para obtener el grado de

MAESTRO EN CIENCIAS

(Ciencias y Tecnologías de la Información)

PRESENTA

Ing. Juan Santana Santana

Asesores

Dr. Manuel Aguilar Cornejo
Dr. Miguel Alfonso Castro García

Sinodales

Presidente: Dr. Ricardo Marcelín Jiménez

Secretario: Dr. Miguel Alfonso Castro García

Vocal: Dr. José Oscar Olmedo Aguirre

México - D.F.
Enero de 2010