UNIVERSIDAD AUTÓNOMA METROPOLITANA

DLML PARA UN AMBIENTE GRID

Para obtener el grado de

MAESTRO EN CIENCIAS
(Ciencias y Tecnologías de la Información)

PRESENTA:

Apolo H. Hernández Santos

Asesores

Dra. Graciela Román Alonso
Dr. Miguel Alfonso Castro García

Sinodales

Presidente: Dr. Ricardo Marcelín Jiménez [UAM-I]

Secretario: Dr. Santiago Domínguez Domínguez [CINVESTAV]

Vocal: Dr. Miguel Alfonso Castro García [UAM-I]

México D.F.
Febrero 2011

Resumen

Como varios sabemos los clusters son una alternativa para obtener grandes capacidades de cómputo. Sin embargo, para usar de forma eficiente los recursos de un cluster se necesita el balance de carga, por lo cual existen herramientas diseñadas para este propósito. Una de estas herramientas es DLML (Data List Management Library), la cuál es una biblioteca diseñada para facilitar la programación de aplicaciones paralelas que organizan sus datos mediante listas. DLML proporciona funciones que permiten insertar y retirar datos de la lista para ser procesados de manera independiente y transparente para el programador, además DLML distribuye la lista (carga) entre los cores de un cluster, permitiendo el procesamiento en paralelo de los datos.

Actualmente existen dos versiones de DLML, una de ellas usa la subasta global y otra la subasta parcial para balancear carga. Sin embargo, estas versiones de DLML funcionan en un único cluster. Esta es una limitante importante, debido a que existen aplicaciones que demandan más poder de cómputo del que puede estar presente en un sólo cluster. Una alternativa a esta limitante es la tecnología Grid, la cuál posee recursos que en teoría podrían considerarse como ilimitados. Por esta razón, esta tesis se centra en el desarrollo de una extensión a DLML para ser usada en un ambiente Grid.

En esta tesis proponemos una nueva arquitectura para DLML, la cual incluye un conjunto de administradores, encargados de coordinar la comunicación y el buen funcionamiento de la nueva arquitectura. Estos administradores incluyen algoritmos para la distribución y balance de carga, entre los diferentes clusters que conforman el ambiente Grid.

La nueva arquitectura fué implementada usando LAM/MPI y ejecutada sobre una VPN

formada por un cluster en la UAM-Iztapalapa y otro en el CINVESTAV Zacatenco (ambos localizados en México DF). En las pruebas realizadas sobre la VPN, se comparó el tiempo de procesamiento de las versiones originales de DLML contra dos nuevas versiones creadas a partir de la nueva arquitectura. Estas dos ultimas versiones muestran mejoras en el tiempo de procesamiento, debido a que la nueva arquitectura reduce el costo de las comunicaciones al limitar la comunicación entre cores de diferente cluster.

Keywords: Grid, Balance de Carga en Grids, Balance de Carga en Clusters, VPN, Aplicaciones MPI sobre OpenVPN, DLML, Cómputo Paralelo, Programación Paralela.

Agradecimientos

Al Consejo Nacional de Ciencia y Tecnología, CONACyT, por la beca otorgada para estudios de posgrado.

A la Universidad Autónoma Metropolitana - Unidad Iztapalapa y al Dr. Santiago Domínguez Domínguez del CINVESTAV-IPN, por facilitar el uso del Cluster Xena perteneciente a esta institución y por ayudar al desarrollo de este proyecto. A todos los Académicos que han sido parte importante en mi formación, en particular al Dr. Miguel Alfonso Castro García co-asesor de esta tesis y la Dra. Graciela Román Alonso por su apoyo, no sólo en esta tesis, sino también, en el proyecto terminal de la Lic. en Computación.

Finalmente agradezco a mis padres por el apoyo mostrado a lo largo de esta maestría.

Contenido

Lista de Figuras **9**

Lista de Tablas **11**

1. Introducción **13**

1.1. Programación paralela en clusters 14

1.2. Balance de carga 16

1.3. DLML para múltiples clusters 17

2. DLML **19**

2.1. Introducción 19

2.2. Arquitectura 19

2.3. Políticas de balance de carga 21

2.3.1. Subasta con manejo de información global 22

2.3.2. Subasta con manejo de información parcial 23

3. Generalidades de una Grid **31**

3.1. Qué es una Grid 31

3.2. Características de una Grid 32

3.3. Servicios que proporciona una Grid 34

3.4. Usos principales de una Grid 35

3.5. Balance de carga en Grids 36

3.5.1. Modelo para el balance de carga en un ambiente Grid 38

4. Propuesta de DLML para un ambiente Grid 41

4.1. Introducción 41

4.2. Arquitectura DLML para un ambiente Grid 42

4.3. Balance de carga Inter-cluster 48

4.4. Comunicación inter-cluster 60

5. Resultados 69

5.1. Plataforma de experimentación 69

5.2. Resultados en el tiempo de ejecución de N-Reinas 72

5.3. Resultados en la distribución de carga de N-Reinas 80

5.4. Resultados en el tiempo de ejecución de Multiplicación de Matrices 86

5.5. Resultados en la distribución de carga de Multiplicación de Matrices 92

6. Conclusiones y trabajo a futuro 95

A. Instalación y configuración de OpenVPN 101

A.1. Instalación 101

A.2. Configuración del Servidor VPN 103

A.3. Configuración del Cliente VPN 106

B. Configuración del Toroide 109

Referencias 113

Lista de Figuras

1.1. Distribución de una lista de datos usando DLML 17

2.1. Arquitectura de DLML 20

2.2. Balance de carga con información global 23

2.3. Cores usando la topología toroide para organizar su comunicación 24

2.4. Búsqueda de carga usando el algoritmo PIF 25

2.5. Evaluación de la carga global del sistema usando PIF 26

2.6. Árbol generado por el algoritmo PIF 27

2.7. Balance de carga con información parcial 28

3.1. Fases para el balance de carga 38

3.2. Modelo jerárquico para el balance de carga en un ambiente Grid 39

4.1. Modelo para el balance de carga de DLML en un ambiente Grid 43

4.2. Nueva arquitectura DLML para el balance de carga en un ambiente Grid . . 45

4.3. Integración del AG, AC, *Aplicación* y proceso *DLML* en la nueva arquitectura 46

4.4. Comunicación entre el *Administrador Cluster* y los cores trabajadores 47

4.5. Carga al inicio del procesamiento 50

4.6. Subasta iniciada por el cluster receptor 58

4.7. Redistribución de la carga, después de una subasta 59

4.8. Túnel VPN entre dos redes LAN 61

4.9. Cluster virtual para ejecutar aplicaciones paralelas 62

4.10. VPN formada por dos clusters, usando OpenVPN 63
4.11. Configuraciones para un toroide sobre una VPN de 84 cores 65
4.12. Conexiones lógicas entre cores de diferente cluster para un toroide de 12 x 7 65
4.13. Conexiones lógicas entre cores de diferente cluster para un toroide de 7 x 12 66
4.14. Conexiones por Internet para la versión de DLML con subasta global 67
4.15. Conexiones por Internet para la nueva arquitectura 68

5.1. Interconexión de dos clusters usando una VPN 70
5.2. Tiempo de ejecución para N-Reinas con Glo-VPN y Glo-Grid 72
5.3. Tiempo de ejecución para N-Reinas con Toro-VPN y Toro-Grid 75
5.4. Tiempo de ejecución para N-Reinas usando las 4 versiones 77
5.5. Tiempo de ejecución de N-Reinas con subasta global en 1 y 2 clusters 78
5.6. Tiempo de ejecución de N-Reinas con subasta parcial en 1 y 2 clusters 79
5.7. Distribución de carga para N-Reinas con Glo-VPN y Glo-Grid 81
5.8. Porcentaje de distribución de carga en los dos clusters 82
5.9. Distribución de carga para N-Reinas con Toro-VPN y Toro-Grid 83
5.10. Porcentaje de distribución de carga en los dos clusters 84
5.11. Distribución de carga para N-Reinas usando las 4 versiones 85
5.12. Tiempo de ejecución para Multiplicación de Matrices con Glo-VPN y Glo-Grid 87
5.13. Tiempo de ejecución para Multiplicación de Matrices con Toro-VPN y Toro-Grid 88
5.14. Tiempo de ejecución para Multiplicación de Matrices con las 4 versiones . . 89
5.15. Tiempo de ejecución de Multiplicación de Matrices en 1 y 2 clusters 91
5.16. Distribución de carga para Multiplicación de Matrices con las 4 versiones . . 92

B.1. Configuraciones del toroide para un cluster de 16 cores 109
B.2. Etapas en la distribución de carga al usar el algoritmo toroide 110
B.3. Distribución de 1000 datos en un cluster de 16 cores 111

Lista de Tablas

5.1. Número de conexiones entre cores de diferente cluster con Glo-VPN 73
5.2. Mensajes a través de Internet generados por Glo-VPN y Glo-Grid 74
5.3. Conexiones entre cores de diferente cluster con Toro-VPN 74
5.4. Mensajes a través de Internet generados por Toro-VPN y Toro-Grid 76
5.5. Mensajes a través de Internet generados por las 4 versiones de DLML 76
5.6. Tiempo de ejecución de la subasta Global en 1 y 2 clusters ejecutando N-reinas con un tablero 16, 17 y 18 78
5.7. Desviación estándar de la carga procesada en Glo-VPN y Glo-Grid 80
5.8. Desviación estándar de la carga procesada en Toro-VPN y Toro-Grid 84
5.9. Mensajes a través de Internet generados por Glo-VPN y Glo-Grid 88
5.10. Mensajes a través de Internet generados por Toro-VPN y Toro-Grid 89
5.11. Tiempo de ejecución de Multiplicación de Matrices en 1 y 2 clusters 90
5.12. Proporción en la capacidad de procesamiento en uno y dos clusters 91
5.13. Desviación estándar de la carga procesada por las 4 versiones 94

Capítulo 1

Introducción

En la mayor parte de las ciencias se presentan fenómenos que necesitan ser estudiados, científicos e investigadores se apoyan en las ciencias de la computación para procesar o simular dichos fenómenos. Para simular o dar solución a ciertos problemas se requieren capacidades de cómputo que no están presentes en una computadora normal. En la actualidad las computadoras personales poseen procesadores hasta con 8 núcleos o cores (en este trabajo hablaremos de núcleos o cores de forma indistinta) y pueden tener instalado uno o más procesadores, con una capacidad en RAM que oscila entre los 2 y 8 Giga-Bytes (GB). Sin embargo, tales capacidades de cómputo no son suficientes para problemas en donde la cantidad de datos, variables y posibles soluciones crecen exponencialmente. Ante tal escenario nos vemos en la necesidad de buscar un mayor poder de cómputo.

Una solución es optar por usar una supercomputadora (generalmente tienen un alto costo económico) con gran cantidad de cores y GB en RAM como la Tianhe-1A, la cual posee 186,368 cores y 229,376 GB en RAM [1]. Otra solución, es usar un conjunto de computadoras de propósito general interconectadas entre ellas (tienen un costo menor comparado con las supercomputadoras). La segunda solución es la adoptada por la mayor parte de las instituciones e investigadores que se dedican al análisis de tales fenómenos, debido principalmente al bajo costo que representa, a diferencia de una supercomputadora.

A pesar de tener un costo menor, el conjunto de computadoras de propósito general se puede estructurar de tal forma, que obtenga resultados similares a los que se obtendrían con una supercomputadora. A este conjunto de computadoras o nodos se le conoce como

cluster (un cluster es una agrupación de computadoras o también llamados nodos mediante la cual, podemos usar el poder de cómputo de todos los cores de estas computadoras, de la misma forma que se haría cuando se tiene un computadora con múltiples cores). Esto es, si tenemos un cluster con 128 cores, divididos en diferentes nodos, podremos disponer del poder de cómputo de los 128 cores para resolver un problema.

1.1. Programación paralela en clusters

Para resolver un problema especifico, usando un cluster, necesitamos usar la programación paralela, por medio de la cual, se puede dividir el problema en subtareas, y asignar cada una de ellas a los cores del cluster, para que sean procesadas de forma simultánea (en paralelo). La programación paralela indica que al finalizar el procesamiento, se debe hacer una recolección de resultados parciales, debido a que en cada uno de los cores se procesa una de las subtareas, y ya que estas pertenecen a un problema más general sus resultados parciales son necesarios para la solución del problema.

Existen varios modelos para la programación paralela, entre ellos está el modelo de programación de paso de mensajes, programación por memoria compartida, programación por skeletons, programación por ADTs paralelos, entre otros (en [2] se incluye un análisis de estos modelos de programación). Para esta tesis usaremos el modelo de programación paralela por paso de mensajes: en este modelo cada uno de los cores pueden comunicarse con otros cores dentro del mismo cluster mediante envío y recepción de mensajes.

Para facilitar la programación del modelo de paso de mensajes existen varios ambientes, entre los principales están, OpenMPI [3] y LAM/MPI [4]. De estas dos opciones, se opto por usar LAM/MPI, ya que ofrece la posibilidad de asignar procesos a los diferentes cores de forma independiente, además, la biblioteca usada como plataforma para este proyecto está construida en LAM/MPI.

Mediante la programación paralela podemos dividir el problema en subtareas, sin embargo, al usar un clusters nos enfrentamos a dos factores que podrían afectar el tiempo de procesamiento:

- ***Clusters heterogéneos***

  Un cluster heterogéneo está formado por computadoras con diferentes características en hardware y software. Este tipo de cluster puede representar un problema, debido a que existen cores con mayor capacidad de cómputo que otros. Esto conlleva, a que en el conjunto de cores del cluster unos podrán procesar más datos que otros, por lo cual, los que posean mayor capacidad de cómputo terminarán de procesar primero, que los cores de menor capacidad de cómputo. Con lo anterior, los cores más rápidos tendrán más tiempo de ocio (ya que deben esperar a que los cores más lentos terminen de procesar sus datos) desperdiciando poder de cómputo, que podría ser usado para mejorar el tiempo de procesamiento global.

- ***Clusters no dedicados***

  Otro factor que influye en la velocidad con la que un core procesa los datos, es la cantidad de carga de trabajo asignada, esto quiere decir que si en un cluster, un core tiene asignadas tareas ajenas a la aplicación de nuestro interés, terminará de procesar los datos de nuestra aplicación en un tiempo mayor al de un core que no tiene tareas ajenas. A este tipo de cluster se le denomina cluster no dedicado.

Al margen de los factores expuestos, en la programación paralela existen aplicaciones estáticas y dinámicas. En las aplicaciones estáticas, se conoce desde el inicio del procesamiento la cantidad de datos que se deben procesar (a los datos por procesar, se les conoce como carga). Conociendo la carga total de la aplicación, se puede dividir entre el número total de cores del cluster, para garantizar que cada core procese la misma cantidad de carga. No obstante, el problema derivado de los clusters heterogéneos y no dedicados sigue estando presente, por esta razón no se puede garantizar que todos los cores del cluster terminarán de procesar su carga al mismo tiempo.

Por otra parte, tenemos a las aplicaciones dinámicas, que en general al procesar su carga, generan aún más carga. En las aplicaciones dinámicas no se conoce cuanta carga se generará ni en qué momento, por lo cual, es difícil dividir la carga equitativamente entre los cores del

cluster. Esta generación dinámica de carga, provoca una desigualdad en la cantidad de carga que cada core debe procesar, debido a que algunos cores pueden generar una gran cantidad de carga a tiempo de ejecución, mientras que los demás sólo podrían generar una pequeña cantidad, a diferencia de los primeros. Aunado a lo anterior, tenemos el problema de los clusters heterogéneos y no dedicados, al igual que las aplicaciones estáticas.

En resumen, existen varios factores y tipos de aplicaciones, que pueden afectar el rendimiento o la ejecución de las aplicaciones en un cluster. Una alternativa que mejora lo anterior es el balance de carga.

1.2. Balance de carga

El balance de carga se define como la distribución equitativa de la carga total entre los cores de un cluster a tiempo de ejecución. Con el balance de carga, se puede mejorar el tiempo de procesamiento de las aplicaciones dinámicas y el aprovechamiento de los recursos presentes en clusters heterogéneos y no dedicados. Para llevar a cabo el balance de carga existen diversas herramientas, algunas de ellas son: The Zoltan Parallel Data Services Toolkit [5], SAMBA [6], DDLB [7] y DLML [2]. Estas herramientas facilitan la programación de aplicaciones paralelas, ya que incorporan las funciones para balance de carga (un análisis de estas herramientas lo encontramos en [2]).

En esta tesis nos centraremos en la biblioteca DLML(Data List Management Library), que 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 y recolección de resultados parciales de todos los cores del cluster [8].

DLML se puede usar en aplicaciones que organizan su carga en listas, en donde cada elemento de la lista es independiente (no necesita de ningún otro elemento de la lista para ser procesado), y puede ser procesado por cualquier core del cluster. DLML divide y distribuye (mediante el paso de mensajes) los elementos de la lista entre los cores del cluster, de manera transparente para el usuario (Figura 1.1).

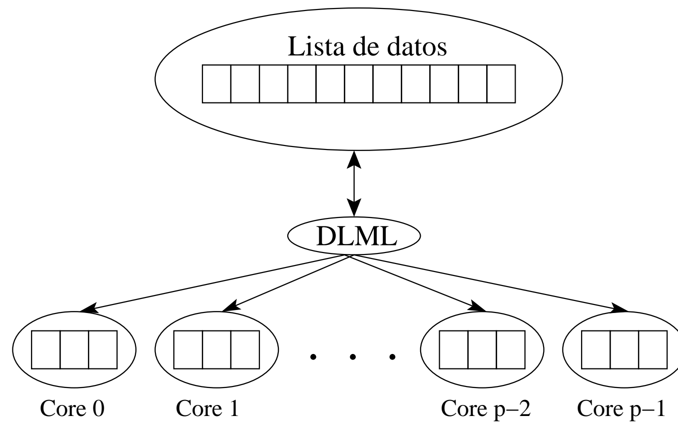

Figura 1.1: Distribución de una lista de datos usando DLML

DLML presenta buenos resultados para balancear la carga a tiempo de ejecución, sin embargo, sólo puede usar los recursos de un único cluster a la vez. Es decir, no es capaz de usar recursos que podrían estar presentes en otros clusters. Esta es una limitante importante ya que existen aplicaciones que demandan un poder de cómputo superior al que puede estar presente en un único cluster.

1.3. DLML para múltiples clusters

Debido a que DLML sólo puede usar recursos de un cluster, se debe modificar para que pueda acceder a más clusters. Una posible solución, es migrar DLML a un ambiente Grid [9], en el cual es posible acceder a múltiples clusters, con ello tenemos la posibilidad acceder a una mayor cantidad de recursos; obteniendo un número de cores igual a la suma de los cores en todos los clusters participantes.

El objetivo principal de este trabajo es diseñar una nueva versión de DLML, la cual podrá usar recursos de varios clusters, y balancear carga a tiempo de ejecución entre dichos clusters. Las versiones actuales de DLML cuentan con funciones para balancear carga entre los cores de un cluster, balance *intra*-cluster, con la nueva versión se pretende balancear carga entre varios clusters, balance *inter*-cluster.

El resto de la tesis se organiza de la siguiente forma: en el siguiente Capítulo se presenta

el estado del arte de DLML. En el Capítulo 3 se presentan las generalidades de una Grid y el balance de carga *inter*-cluster. La nueva arquitectura para DLML y el análisis de los algoritmos para dicha arquitectura se presenta en el Capitulo 4. En el Capítulo 5 se hace un análisis de los resultados obtenidos al comparar la nueva versión de DLML con las anteriores. Finalmente en el Capítulo 6, se presentan las conclusiones y el trabajo futuro.

Capítulo 2

DLML

2.1. Introducción

DLML (Data List Management Library) es una biblioteca que facilita la construcción de aplicaciones paralelas, que usan listas para representar y procesar sus datos de forma independiente. DLML divide y distribuye de manera transparente la lista de datos entre los cores de un cluster a tiempo de ejecución, es decir, hace balance de carga entre los cores del cluster. Para llevar a cabo el balance de carga, DLML ofrece funciones para la inserción y eliminación de elementos en la lista perteneciente a la aplicación, además, proporciona funciones para la recuperación de resultados parciales.

Actualmente existen dos versiones de DLML, una balancea la carga entre los cores de un cluster tomando en cuenta a todos los cores, DLML con manejo de información global. La otra versión sólo balancea la carga entre algunos cores, DLML con manejo de información parcial. En este capítulo se estudia el funcionamiento de estas dos versiones y su arquitectura.

2.2. Arquitectura

La arquitectura de DLML está formada por dos procesos, *Aplicación* y *DLML*, los cuales corren en cada uno de los cores del cluster, Figura 2.1. Los procesos *Aplicación*, se encargan de procesar la lista de datos, y los procesos *DLML*, se encargan de balancear la carga a tiempo de ejecución. En la figura se puede apreciar que el proceso *Aplicación* posee la lista

de datos, mientras que el proceso *DLML* tiene acceso a esta lista, por medio de intercambio de mensajes. Esto es necesario, debido a que el proceso *Aplicación* toma datos de la lista para procesarlos y en cuanto la lista se queda vacía (el proceso *Aplicación* ha terminado de procesar los datos de su lista local), el proceso *DLML* se encarga de buscar más datos, para que la *Aplicación* los procese.

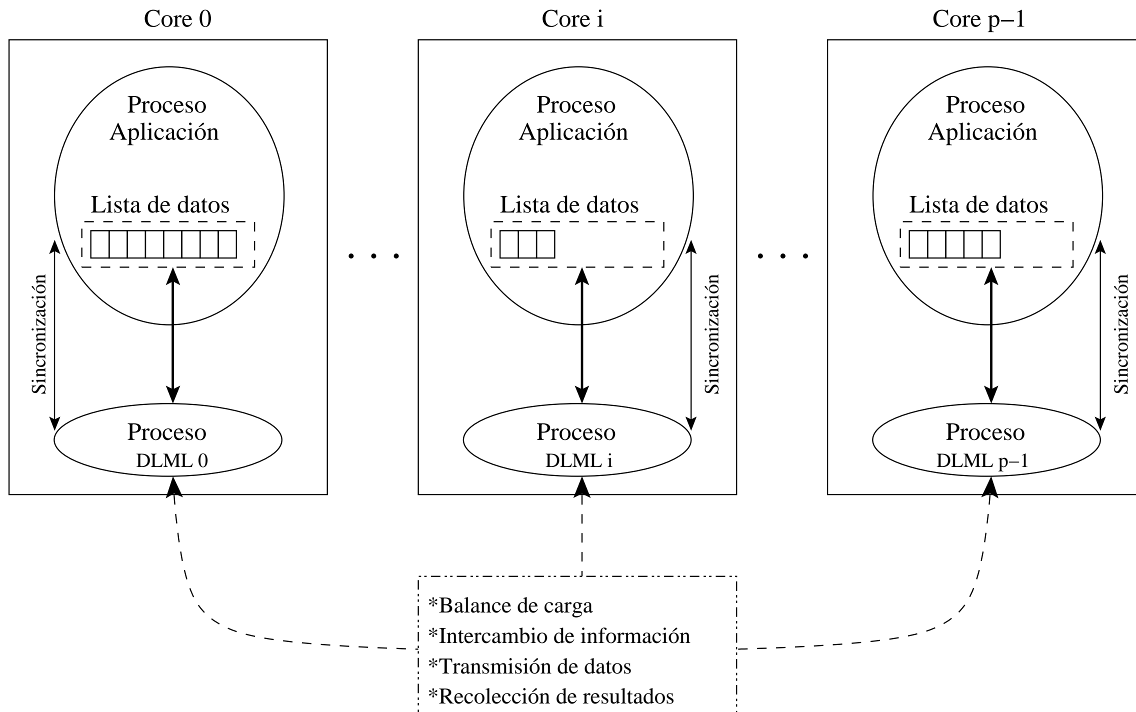

Figura 2.1: Arquitectura de DLML

La búsqueda de datos o carga, se hace mediante el intercambio de información y datos (mediante envío y recepción de mensajes), entre los *DLML's* del cluster. En cuanto un *DLML* sin carga, obtiene los datos de algún *DLML* remoto, los inserta en la lista local de la *Aplicación* para que empiece a procesar nuevamente.

Las dos versiones existentes de DLML, usan esta misma arquitectura para balancear carga entre los cores del cluster. La diferencia entre las dos versiones radica, como se verá más adelante, en la aplicación de las políticas de balance de carga; *información, transferencia, selección y localización.* Estas políticas, entre otras, determinan el momento de la transferencia, el qué se transferirá, quién realizará la transferencia y quién recibirá la carga.

2.3. Políticas de balance de carga

Para llevar a cabo el balance se necesitan políticas que definan como se va a distribuir la carga entre todos los cores del cluster, las políticas de balance que usa DLML en la versión de DLML con manejo de información global y la versión de DLML con manejo de información parcial se enlistan a continuación:

- **Política de información:** Define lo que se considera como carga y como recolectarla. La carga puede ser el número de procesos que está atendiendo el core, el porcentaje de utilización del CPU, la longitud de la cola de espera del CPU y el número de datos que faltan por procesar en una aplicación especifica. También se debe tomar en cuenta como recolectar la información, esto quiere decir, que se debe elegir si hacer una recolección global (obtener información de todos los cores) o parcial (obtener información sólo de un conjunto de cores).

  Para el caso de DLML, esta es la única política que varía en sus dos versiones, se considera carga a los datos de la lista, que faltan por procesar en el proceso *Aplicación*, y se usa información global y parcial para determinar a donde enviar la carga. Para DLML con manejo de información global, se toma en cuenta la información de carga de todos los cores del cluster, y en base a esta se redistribuye la carga, mientras que para DLML con manejo de información parcial, sólo se toma en cuenta la información de carga de algunos cores, ya que en esta versión los *DLML* poseen un número limitado de vecinos para intercambiar información.

- **Política de transferencia:** Define el momento en que se debe hacer la distribución de carga y el quién debe hacer la distribución. Para DLML el momento de hacer una transferencia es cuando un core termina de procesar su lista de datos (se queda sin carga), y en cuanto al quién, es el mismo core quien termina su carga, el que determina que core será el que le envíe parte de su carga.

- **Política de selección:** Define que se va a transferir (procesos o datos), cuáles y cuantos

(datos o procesos) se van a transferir. En DLML se transfieren datos de la lista local de un core, en cuanto a cuáles y cuántos, DLML divide el número de datos de su lista, que le faltan por procesar, entre el número de solicitudes que le han llegado más uno (para incluirse él mismo).

- **Política de localización:** Define a dónde mover la carga dependiendo de la información recolectada. Para DLML, la carga se mueve del core al que se le ha hecho una petición de carga al core descargado, esto se cumple, si el core al que se le ha hecho la petición, posee carga al momento de atender la petición.

Con las cuatro políticas mencionadas se delimita el funcionamiento de DLML con manejo de información global y DLML con manejo de información parcial.

2.3.1. Subasta con manejo de información global

Como es mostrado en la Sección 2.2, la arquitectura de DLML consta de dos procesos por core, *Aplicación* que se encarga de procesar los datos de la lista y *DLML* encargado de balancear la carga en el sistema. Al inicio del procesamiento, la *Aplicación* 0 (en el core 0) es la única que posee carga, las demás *Aplicaciones* solicitan carga a sus procesos *DLML*. Los procesos DLML usan el algoritmo subasta global para subastar el poder de cómputo del core descargado.

Una explicación gráfica del funcionamiento de la subasta global la encontramos en la Figura 2.2, en la cual tenemos un cluster con 6 cores M, N, P, Q, R, T. Inicialmente el core M se queda sin carga, por tanto, pide información de carga a todos los cores del sistema (Figura 2.2.a). Como se muestra en la Figura 2.2.b, los cores responden a la petición de M enviando la información de su carga, es decir, la longitud de sus listas. M almacena en una lista la información de carga de todos los cores, y en base a esta lista, M escoge al core que tiene más carga para proporcionarle su poder de cómputo. De esta manera selecciona a R. Una vez que M determina que R tiene más carga, le envía un mensaje para solicitarle parte de su carga (Figura 2.2.c). Ante la solicitud de M, R divide su carga entre 2 (la petición de M y la carga para el mismo), y envía la mitad su carga a M (Figura 2.2.d).

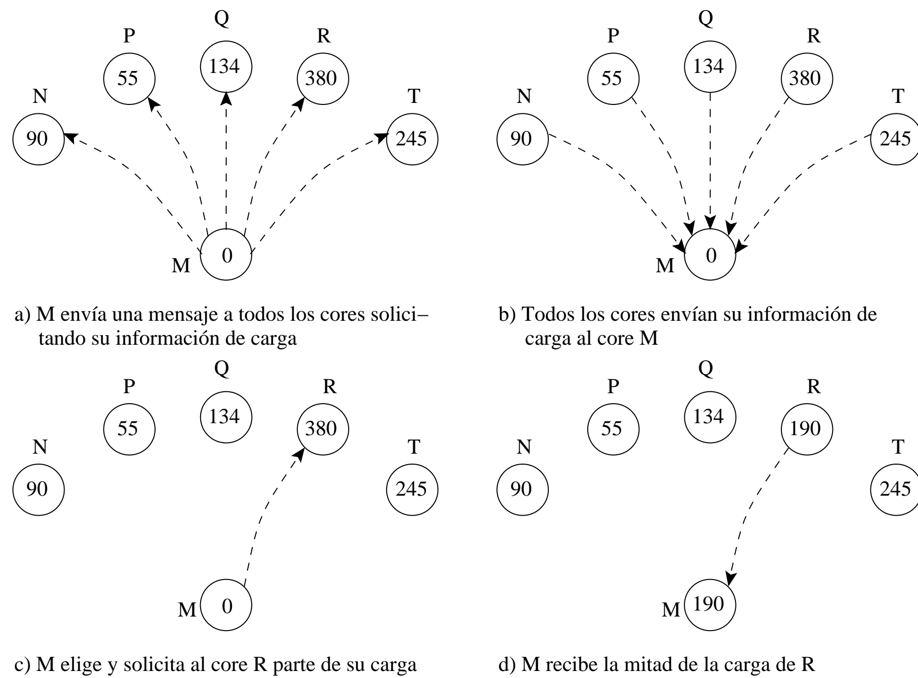

Figura 2.2: Balance de carga con información global

Como se nota en el algoritmo de DLML con manejo de información global, los cores descargados envían y reciben mensajes de todos los cores del sistema, cuando solicitan información de carga. El enviar mensajes a todos los cores del sistema, cada vez que se desea subastar el poder de cómputo, representa una limitante en la escalabilidad del sistema, debido a que uno o la totalidad de los cores pueden enviar o recibir mensajes en un momento dado, esto se traduce en un alto costo en comunicaciones al crecer el número de cores participantes.

Ante esta limitante se planteó la nueva versión de DLML, en la cual, un core no conoce el estado de carga de todos los cores del sistema, sólo mantiene información de carga de algunos cores vecinos.

2.3.2. Subasta con manejo de información parcial

En esta versión se usa la política de información parcial, cada core sólo tiene comunicación con un conjunto de cores vecinos, esto reduce el costo de comunicaciones y también el problema de la escalabilidad. En cuanto a la arquitectura, se tienen los mismos elementos

que en la versión de DLML con manejo de información global, descrita en la Sección 2.2.

En esta versión, un core (sin carga) solamente solicita información de carga a sus cores vecinos para subastar su poder de cómputo. El que un core sólo mantenga comunicación con sus vecinos no impide que el sistema completo se balancee. Esto se debe a que todos los cores tienen sus propios cores vecinos, y estos a su vez son vecinos de otros. De esta forma, se obtiene una propagación de carga hacia todos los cores del cluster.

En esta versión la comunicación entre los cores del cluster, se hace mediante el uso de la topología lógica toroide. Con esta topología se puede garantizar que un core sólo tenga comunicación con 4 cores vecinos[8], como se puede ver en la Figura 2.3, en donde tenemos 16 cores, usando la topología toroide para organizar su comunicación.

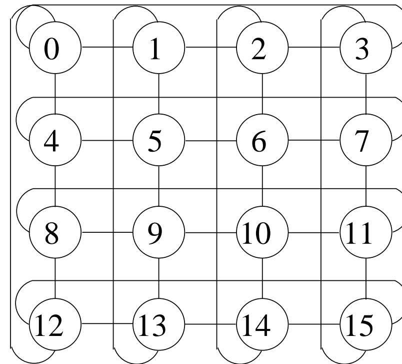

Figura 2.3: Cores usando la topología toroide para organizar su comunicación

Con esta versión de DLML, se debe tomar en cuenta que cada core sólo conoce el estado de carga de 4 de los cores del cluster. Esto significa, que un core no es capaz de identificar, por sí solo, cuando todos los cores del cluster se han quedado sin carga. Por esta razón, se debe tener un mecanismo que permita identificar cuando el sistema, en su totalidad, se ha quedado sin carga y está listo para iniciar la recolección de resultados parciales.

Este mecanismo lo proporciona el algoritmo PIF (Propagation of Information with Feedback) [10], el cual se incluye en la versión de DLML con manejo de información parcial. Con PIF se puede obtener una visión global de la carga, y por tanto, conocer cuándo se puede dar por terminada la ejecución.

PIF inicia en un core especial llamado *iniciador* (core 0 en DLML), el cuál es el encargado

de iniciar la propagación de la petición de información de carga enviando un mensaje para solicitar la información de carga de todos sus vecinos.

Un core cualquiera i que reciba el mensaje de solicitud por primera vez envía un mensaje de solicitud a todos sus cores vecinos, excepto al core de quien recibió la petición.

Los pasos anteriores forman un árbol entre los cores que envían peticiones y los que las reciben. Los que envían las peticiones, son padres de los cores a quienes les envían la petición, y todo core que recibe una petición es hijo de quien la recibió. De esta forma, todo proceso hijo responde a su proceso padre, cuando todos sus procesos hijo le han enviado su estado de carga. La respuesta incluye la información de carga propia y la de sus hijos.

Por ejemplo, con la topología mostrada en la Figura 2.4.a, se forma el árbol de la Figura 2.6 al ejecutar PIF. El core iniciador lanza la petición de información de carga a sus cores vecinos 1 y 2, Figura 2.4.b. En la Figura 2.4.c, el core 1 responde enviando su estado de carga al iniciador, mientras que el core 2 (como es la primera vez que recibe una petición) envía la petición a sus cores vecinos 3, 5 y 4, exceptuando al iniciador, porque es de quién la recibió.

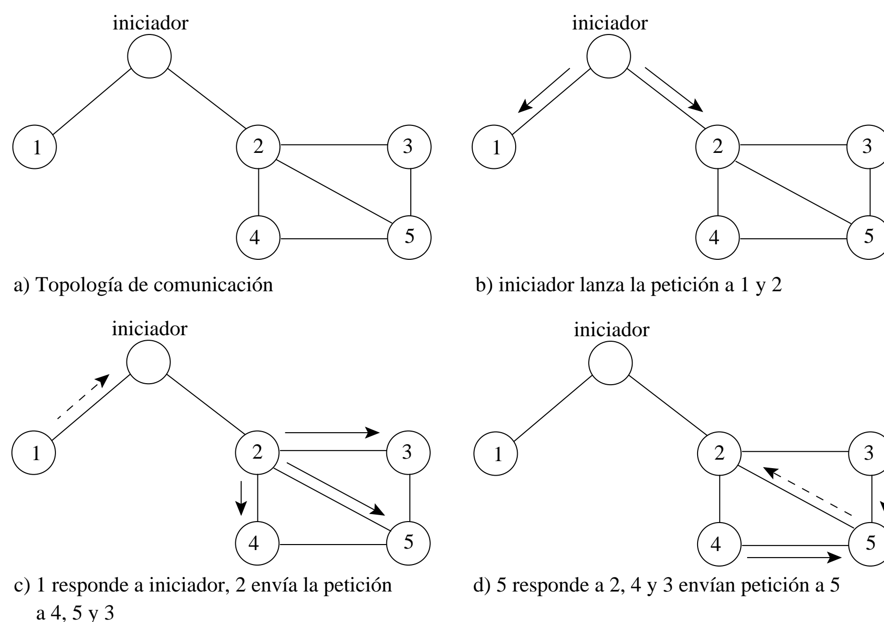

Figura 2.4: Búsqueda de carga usando el algoritmo PIF

El core 3, al recibir la petición, envía la petición a 5, pero no a 2, porque es de quién

la recibió. Sin embargo, el core 5 no atenderá la petición de 3, ya que PIF sólo atiende a la primera petición que llegó, en este caso, la del core 2. Por esta razón, el core 3 no posee ningún hijo y responde con su estado de carga a su core padre (el core 2).

De la misma forma, al recibir la petición de 2, el core 5 envía una petición a 3 y 4, pero como estos cores ya habían recibido la petición del core 2, no atienden esta petición. Por lo cual, se deja al core 5 sin hijos, y envía su estado de carga a su core padre (el core 2), Figura 2.4.d. Para el caso del core 4, solamente envía la petición a 5, pero como 5 ya había recibido la petición de 2, no la atiende y deja al core 4 sin hijos. Como el core 4 no tiene hijos, envía su estado de carga al core 2, quién es su padre.

Finalmente, el core 2 envía su estado de carga junto con el de sus hijos 3, 4 y 5 al iniciador, Figura 2.5.e. Con esto, el iniciador obtiene la información de carga global del sistema, como se muestra en la Figura 2.5.f.

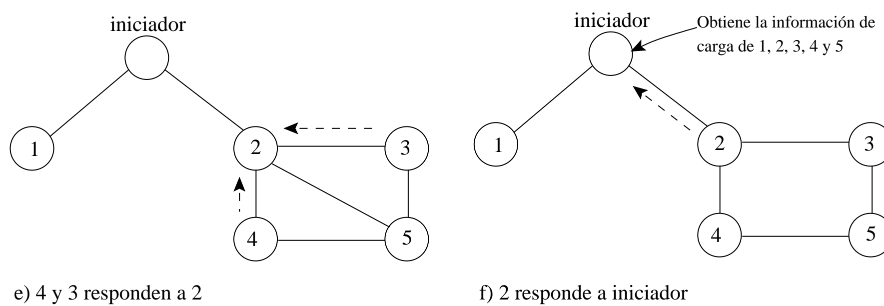

Figura 2.5: Evaluación de la carga global del sistema usando PIF

Como resultado del envió y recepción de mensajes (entre cores vecinos) se obtiene el árbol de la Figura 2.6, aunque este árbol no es el único que se puede generar al ejecutar PIF, ya que todo depende de la velocidad con la que un core haga la petición a otro core vecino. Esto se debe a que el core que recibe la petición, sólo propaga la primera (generando hijos). En caso contrario se limita a esperar la respuesta de sus hijos.

La información para generar el árbol es almacenada la primera vez que se ejecuta el algoritmo PIF, es decir, cada core almacena que core es su padre y que cores son sus hijos. De esta forma, en las siguientes evaluaciones (de la carga global del sistema), los cores usarán el árbol que se formó en la primera ejecución de PIF.

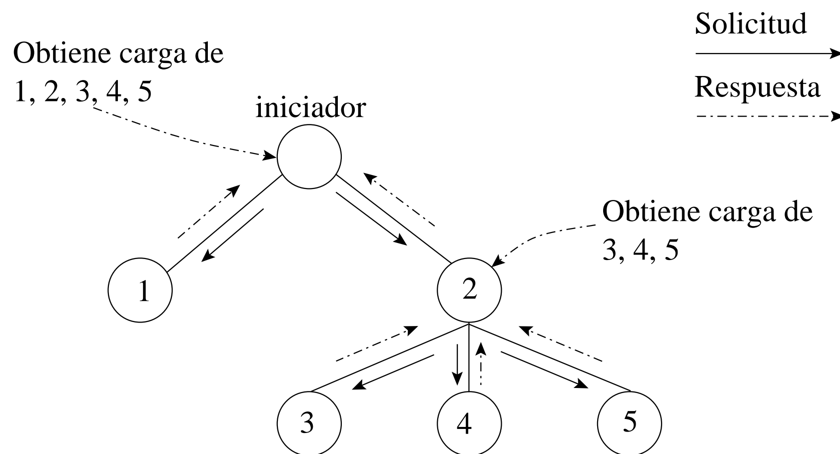

Figura 2.6: Árbol generado por el algoritmo PIF

Como podemos observar, el algoritmo PIF es una parte esencial para el funcionamiento de DLML con manejo de información parcial, ya que sin este algoritmo no se podría determinar cuándo finalizar el procesamiento de la aplicación. PIF permite realizar uno de los cuatro estados que componen esta versión de DLML: los estados de iniciación, distribución de carga, búsqueda de carga del sistema (usando PIF) y terminación. Estos estados se describen a continuación[8]:

- ***Estado de inicialización***

  En la inicialización se ejecutan los procesos *Aplicación* y *DLML* en todos los cores del sistema y se tienen algunos datos en el core 0.

- ***Estado de distribución de carga***

  La distribución de carga inicia cuando un core se encuentra descargado, ya sea porque el sistema se está iniciando o porque ha terminado de procesar su lista. Por ejemplo, si tenemos un cluster de 16 cores como el de la Figura 2.7, en donde el core 5 se queda sin carga. Como el core 5 esta descargado inicia el algoritmo de balance solicitando información de carga a todos sus cores vecinos, Figura 2.7.a.

  Cuando los cores vecinos reciben la petición de carga, responden enviando la cantidad de carga que tienen en sus listas, Figura 2.7.b. Con la información recibida el core 5

elige al core 6 (ya que es el que mayor carga posee), y le envía una petición de carga, Figura 2.7.c. El core 6 responde enviando una parte de su carga como se muestra en la Figura 2.7.d.

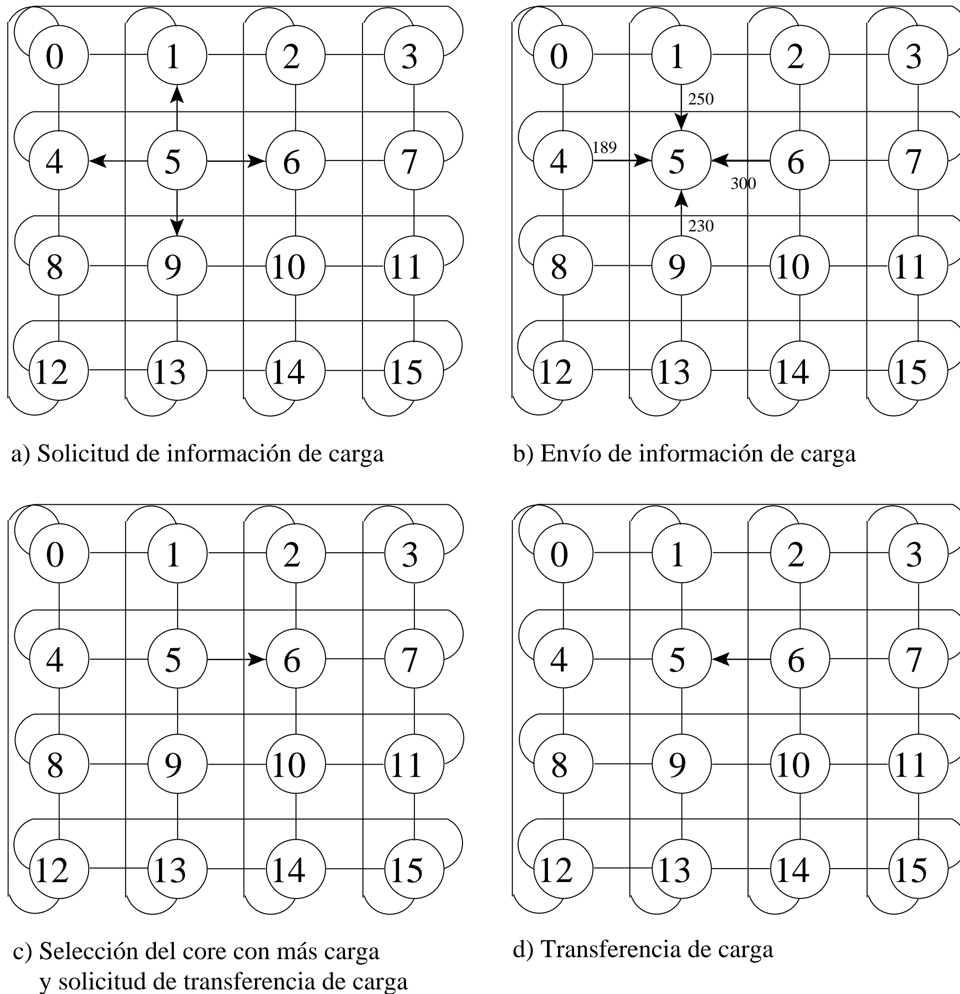

Figura 2.7: Balance de carga con información parcial

La cantidad de carga que 6 envía a 5 está determinada por la cantidad de solicitudes que 6 tiene en su cola de peticiones. Esto se debe a que el core 6 puede recibir más peticiones de carga (de sus otros vecinos descargados). Estas peticiones son almacenadas en una cola de peticiones, y en el momento de la distribución, los datos de la lista (del core 6) se dividirán entre el número de solicitudes más uno (carga para el mismo). Sin embargo, si la longitud de la lista es menor que el número de solicitantes, el core 6 envía un dato

de la lista a cada core solicitante, y en el momento que agote su lista envía la etiqueta SIN DATOS a los cores restantes.

Un caso interesante sucede cuando un core descargado como 5 pide información de carga a sus cores vecinos y estos también se encuentran descargados. El core 5 no puede iniciar la terminación del sistema, ya que sólo conoce información de carga de sus vecinos, y no posee una visión global del sistema. Por lo cual envía una notificación de disponibilidad a todos sus vecinos y se queda inactivo, hasta recibir un mensaje de respuesta de estos cores.

Los cores vecinos, que reciben el mensaje de disponibilidad, guardan los ID's de los cores en una lista de disponibilidad. Cuando alguno de los cores recibe carga, éste envía a los cores en su lista de disponibilidad un mensaje para que cada uno de ellos inicie su algoritmo de búsqueda de carga.

- ***Estado de búsqueda de carga global del sistema***

  Cuando un core solicita carga y sus vecinos también están descargados, no significa que el sistema completo esté descargado, puede haber un core lejano trabajando. Por lo tanto, se debe tener un mecanismo mediante el cual un core pueda tener una visión global del sistema.

  Este mecanismo es el algoritmo PIF, mencionado anteriormente, con el cual se recolecta información de carga de todos los cores del sistema. Teniendo una visión global del estado de carga de todo el sistema se puede saber el momento en que se debe iniciar la recolección de resultados parciales. No obstante, si todos los cores del sistema inician la búsqueda global del sistema con el algoritmo PIF, el costo en comunicaciones sería alto debido a que muchos cores podrían detectar que sus vecinos no tienen carga. Ante esta situación, en DLML sólo el core 0 es el encargado de determinar cuándo el sistema completo está descargado.

- ***Estado de terminación***

  Este estado inicia cuando el core 0 ha verificado que ya no existe carga en el sistema,

mediante el algoritmo PIF. El core 0 envía el mensaje de terminación a todos los cores del sistema, a través del árbol generado por el algoritmo PIF, en el estado de búsqueda de carga global del sistema. El mensaje se propaga desde el core 0, que es la raíz del árbol generado, hasta los cores hoja. En el momento en que los cores hoja reciben el mensaje de terminación, finalizan su ejecución enviando sus resultados parciales a sus cores padre, y estos a su vez, envían sus resultados parciales junto con los de sus hijos, hasta llegar al core 0.

En la Sección 2.3.1 y 2.3.2 se presentó el funcionamiento de DLML con manejo de información global y con manejo de información parcial respectivamente, y se describieron los algoritmos para el balance de carga. Sin embargo, en estas dos versiones, no se considera el caso en el cual una aplicación y el tamaño del problema demandan más poder de cómputo que el que está presente en el cluster. En este caso, ninguna de las dos versiones de DLML ofrecen alguna solución, debido a que DLML fue diseñada para funcionar y tomar recursos de un sólo cluster.

Una posible solución es modificar DLML para que pueda acceder a recursos que están presentes en otros clusters, esto nos daría la posibilidad de incrementar el poder de cómputo. Esta posibilidad la ofrece la tecnología Grid, mediante la cual se puede acceder a una gran cantidad de recursos interconectados a través de Internet. Con la Grid se puede solucionar la limitante de las dos versiones de DLML, por esta razón en el siguiente capítulo se abordará el concepto de Grid, sus características y algunos de los enfoques en cuanto a balance de carga en Grids.

Capítulo 3

Generalidades de una Grid

En este capítulo describiremos lo que es la tecnología Grid, sus características y sus principales usos, de entre los cuales, el supercómputo distribuido es el que más nos interesa para esta tesis. Esto se debe a que mediante esta tecnología buscamos acceder a un mayor número de recursos, con el objetivo de correr DLML en un ambiente *multi*-Cluster.

3.1. Qué es una Grid

Grid es una nueva infraestructura que fue descrita por primera vez por Ian Foster y Carl Kesselman en [9], en donde describen la necesidad de compartir recursos a nivel global, haciendo una analogía con las compañías que proporcionan energía eléctrica. Foster y Kesselman comparan la forma en la que los usuarios de la corriente eléctrica pueden acceder a ésta, a través de una terminal que llega hasta sus casas. Todo lo anterior sin tener conocimiento de toda la infraestructura de la planta eléctrica, además de ser, un servicio estándar y de bajo costo.

En base a esta analogía, en 1998 Ian Foster y Carl Kesselman definen una Grid como una infraestructura de hardware y software que proporciona acceso confiable, constante, penetrante, y barato a recursos de cómputo con la última tecnología[9]. Posteriormente en 2001 Ian Foster redefine a la Grid, como el compartimiento coordinado de recursos y la solución de problemas en organizaciones virtuales dinámicas (VO's), multi-institucionales [11]. Aunado a esta definición se han sumado otras que definen a Grid como un tipo de sistema

paralelo y distribuido que permite dinámicamente la compartición, selección, y agregación de recursos "autónomos", distribuidos geográficamente en tiempo de ejecución, dependiendo de su disponibilidad, capacidad, rendimiento, costo, y los requerimientos de calidad del servicio que demande el usuario [12]. Las aplicaciones de una Grid son muy variadas, por lo cual está emergiendo como un nuevo paradigma para la solución de problemas en la ciencia, ingeniería, industria y comercio [13].

3.2. Características de una Grid

Para que un sistema pueda ser identificado como Grid, debe cumplir con tres puntos principales [13], estos se en listan a continuación:

- Debe contar con un compartimiento de recursos coordinado, sin control centralizado, y que los usuarios residan en dominios administrativos diferentes.
- El sistema debe usar interfaces y protocolos abiertos, estándares y de propósito general. Si esto no fuera así, los componentes del sistema no podrán comunicarse y probablemente se trate de un sistema de aplicación especifica, en lugar de una Grid.
- Distribución de calidades de servicio no triviales, esto quiere decir, que la Grid permite utilizar los recursos que la constituyen de manera coordinada, para ofrecer diferentes calidades de servicio. Por ejemplo, se pueden ofrecer diferentes tiempos de reacción, diferente rendimiento del procesamiento, diferente disponibilidad y seguridad [14].

De esta manera, podemos decir que la Grid está basada en compartir recursos, como computadoras, dispositivos de almacenamiento, sensores y un conjunto de redes heterogéneas. Compartir recursos está condicionado, por políticas de seguridad (quién puede acceder a los recursos), políticas de acceso a recursos (a qué recursos se puede acceder y por cuánto tiempo), políticas de negociación y el pago o costo por acceder a estos recursos.

Se han identificado diez características principales que una Grid debe tener[15]:

1. Escalabilidad: una Grid debe estar habilitada para tratar con sólo algunos recursos o con millones de ellos.

2. Distribución Geográfica: los recursos que componen la Grid pueden estar localizados en diferentes lugares.

3. Heterogeneidad: los recursos de la Grid pueden tener variaciones en el hardware y en el software.

4. Compartimiento de recursos: los recursos de la Grid pertenecen a diferentes organizaciones que permiten el acceso a sus recursos.

5. Múltiples administraciones: ya que los recursos de la Grid pertenecen a diferentes organizaciones, estos tienen diferentes políticas de seguridad y administración.

6. Coordinación de recursos: los recursos de la Grid se deben coordinar para proveer capacidades de cómputo.

7. Acceso transparente: una Grid debería verse como una sola computadora virtual.

8. Acceso fiable: el acceso a la Grid, debe asegurar al usuario que recibirá un rendimiento y una calidad de servicio de alto nivel.

9. Acceso consistente: para el acceso a la Grid deben usar servicios, protocolos e interfaces estándar.

10. Acceso permanente: en la Grid los recursos aparecen y desaparecen dinámicamente, por lo tanto, la Grid debe extraer el máximo rendimiento de los recursos disponibles.

3.3. Servicios que proporciona una Grid

Los servicios que proporciona un sistema Grid se resumen en los siguientes puntos:

- ***Seguridad***

  Seguridad para los recursos que forman parte de la Grid y para los usuarios que usan esos recursos. La seguridad de la Grid está basada en la criptografía, uso de "firewalls" y la autentificación de usuarios u otros sistemas.

- ***Monitoreo de recursos***

  En un sistema tan complejo como una Grid, cada uno de los componentes de software y de hardware deben funcionar correctamente para que la Grid brinde un servicio eficiente. Debido a esto, es necesario que un sistema Grid cuente con un servicio de monitoreo robusto, para los recursos que conforman el sistema, que como ya se ha mencionado, cuentan con recursos pertenecientes a dominios administrativos multi-institucionales.

- ***Comunicación y colaboración***

  La Grid toma tecnologías existentes, como XML y Web Services, las modifica y las adecúa para lograr la comunicación entre conjuntos de sistemas que interactúan unos con otros, dentro y fuera de la Grid.

- ***Scheduling y administración de recursos***

  La Grid involucra la administración de recursos heterogéneos, distribuidos geográficamente y disponibles dinámicamente (los recursos pueden estar o no estar disponibles, de un momento a otro). Sin embargo, la efectividad de la Grid depende de la eficiencia y efectividad de los calendarizadores [13]. El "Scheduling" en la Grid involucra 4 etapas: localización de recursos, selección de recursos (en base a su estado, desocupado u ocupado, y a su latencia), generación de la calendarización y la ejecución del trabajo.

Cuatro de los sistemas administradores de recursos más usados en Grids son Condor, SGE, PBS and LSF [13].

- ***Administración de flujos de trabajo***

  La administración de flujos de trabajo es un servicio mediante el cual la Grid organiza el flujo de trabajo, esto es necesario para que una aplicación pueda acceder a los recursos de la Grid (para cumplir con objetivo en particular). El flujo de trabajo, lo define la WfMC (Workflow Management Coalition) [16] como la automatización de todo o parte del proceso de negocio, durante el cual, documentos, información o tareas son pasadas de un participante a otro siguiendo un conjunto de reglas procesales [13], las cuales dictan como se debe recolectar la información dentro del proceso.

3.4. Usos principales de una Grid

Las principales categorías en las que se puede agrupar el uso de las Grids, son las siguientes[15]:

- ***Supercómputo distribuido:*** permite a las aplicaciones usar la Grid, para reducir el tiempo de procesamiento, usando un gran número de recursos.

- ***Cómputo de alto rendimiento específico:*** permite a las aplicaciones usar la Grid, para aprovechar los ciclos de CPU que no se usan, para realizar diversas tareas ligeramente acopladas o independientes.

- ***Cómputo en demanda:*** permite a las aplicaciones usar la Grid, para acceder a recursos que no pueden ser costeados o que no se encuentran localmente. Mediante el cómputo en demanda, se puede acceder a recursos como aplicaciones, datos y poder de computo, entre otros. Para acceder a estos recursos sólo se debe pagar por su uso, sin tener la necesidad de adquirirlos.

- ***Cómputo intensivo de datos:*** permite a las aplicaciones usar la Grid, para sintetizar nueva información a partir de repositorios de datos distribuidos, bibliotecas digitales y

bases de datos. Por ejemplo, la creación de una base de datos, usando otras bases de datos remotas.

- ***Cómputo colaborativo:*** permite a las aplicaciones usar la Grid, para conectar aplicaciones y humanos dentro de un grupo de trabajo colaborativo. El sistema permite la interacción en tiempo de real entre humanos y aplicaciones, usando espacios virtuales. Un ejemplo de aplicaciones que podrían usar el cómputo colaborativo proporcionado por la Grid son las aplicaciones multi-conferencia.

- ***Cómputo multimedia:*** la Grid proporciona la infraestructura para aplicaciones multimedia en tiempo real. Esto requiere dar soporte en la calidad del servicio a través de múltiples máquinas, por ejemplo en las aplicaciones para videoconferencia.

Recapitulando, dentro de los principales usos de las Grids, el de supercómputo distribuido tiene como principal función, el reducir el tiempo de procesamiento usando un gran número de recursos. Esta función de las Grid, es la que se está buscando para superar la problemática de sobrecarga de trabajo que se presenta en un cluster, cuando se ejecutan aplicaciones que demandan mayores capacidades de cómputo.

Como ya se mencionó, se necesitan acceder a más recursos, con el objetivo de minimizar el tiempo de procesamiento de un sólo cluster, teniendo acceso a más recursos que podrían estar presentes en otros clusters. Sin embargo, al acceder a más de un cluster se debe atender el balance de carga entre los diferentes clusters, por lo cual estudiaremos el balance de carga en Grids.

3.5. Balance de carga en Grids

Para el balance de carga debemos ver a la Grid como un conjunto de clusters, en donde cada cluster posee un conjunto de cores trabajadores, que se comunican a través de una red LAN [17], y cada uno de estos clusters se comunica usando Internet. Tomando en cuenta esta infraestructura, el balance de carga se hará entre los diferentes clusters que conforman

la Grid (balance de carga *inter*-cluster). De la misma forma que en un cluster es necesario el balance de carga, en la Grid el balance de carga es importante debido a que está compuesta de recursos heterogéneos, autónomos y dinámicos [17], y al igual que en el cluster se busca aprovechar al máximo los recursos presentes en la Grid y minimizar el tiempo de ejecución global.

Como en el caso de los clusters, en un ambiente Grid también debemos enfrentarnos al problema de las aplicaciones dinámicas y a la heterogeneidad de los clusters. Aunado a esto, se debe tomar en cuenta, que cada cluster puede pertenecer a un dominio de administración diferente (por lo cual sus políticas de seguridad y uso, seguramente serán diferentes), y esto podría afectar en el rendimiento o la coordinación *inter*-cluster.

Dentro de las propuestas estudiadas para el balance de carga [17], [18], [19], [20], encontramos una estructura similar en cuanto a la arquitectura y funcionalidad de los elementos en la Grid. Se tiene un conjunto de procesos encargados de administrar y vigilar el estado de carga de los cores y de los clusters, y otro conjunto de procesos encargados de procesar la carga, llamados procesos trabajadores. Los procesos administradores se encargan de identificar el desbalance, recolectar información y ordenar la redistribución de carga; mientras que los procesos trabajadores procesan la carga y ejecutan las decisiones de los procesos administradores. Los procesos administradores se ejecutan en cores diferentes a los cores, donde se ejecutan los procesos trabajadores, con el fin de evitar que las tareas de procesamiento afecten a las tareas de administración y viceversa (por esta razón hablaremos de cores trabajadores y cores administradores).

En cuanto a los algoritmos para el balance de carga se tienen generalmente tres fases, Figura 3.1. Estas fases se ejecutan cíclicamente hasta finalizar el procesamiento global [21]: recolección de información, toma de decisión y migración de datos.

En la fase de recolección de información, se toma información de la carga de trabajo de los procesos trabajadores y el estado del ambiente de cómputo, para detectar un posible desbalance. En la fase de toma de decisión, se calcula la mejor distribución para la carga, y en la última fase, se migra la carga ejecutando la decisión tomada en la fase anterior.

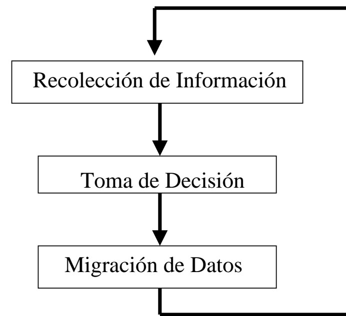

Figura 3.1: Fases para el balance de carga

3.5.1. Modelo para el balance de carga en un ambiente Grid

Para el balance de carga encontramos los enfoques de multi-agente DLB [19], agentes inteligentes [20] y el del modelo jerárquico independiente de la infraestructura Grid [17]. A grandes rasgos, el enfoque del multi-agente DLB (Dynamic Load Balancing) está basado en dos grupos de agentes, un grupo está encargado de informar el estado de carga de una máquina y el tiempo de respuesta de un host a otro, y el segundo grupo de agentes toma la información del primer grupo para efectuar el balance de carga. De la misma forma, en el enfoque de agentes inteligentes, se tiene un conjunto de agentes encargados de evaluar el estado de carga de los recursos. Sin embargo, en este enfoque se usa un mecanismo de predicción, para evaluar el rendimiento de una aplicación, y en base a las predicciones y a la información de carga se pueden tomar las decisiones de balance.

Por otra parte está el modelo jerárquico (en el cual nos centraremos para esta tesis), en donde se describe un modelo para ejecutar una aplicación en un ambiente *multi*-cluster, independiente de su infraestructura, Figura 3.2.

En este modelo tenemos dos tipos de procesos administradores, un *Administrador Grid* y un *Administrador Cluster*. También se cuenta, con un conjunto de procesos trabajadores, encargados de procesar los datos y vigilar el estado de carga de cada uno de los nodos de la Grid. Como ya se había mencionado, los procesos administradores y los trabajadores se ejecutan en cores diferentes, por lo cual, para el balance de carga se hace distinción entre los cores encargados de procesar (ejecutan a los procesos trabajadores) y los cores encargados de

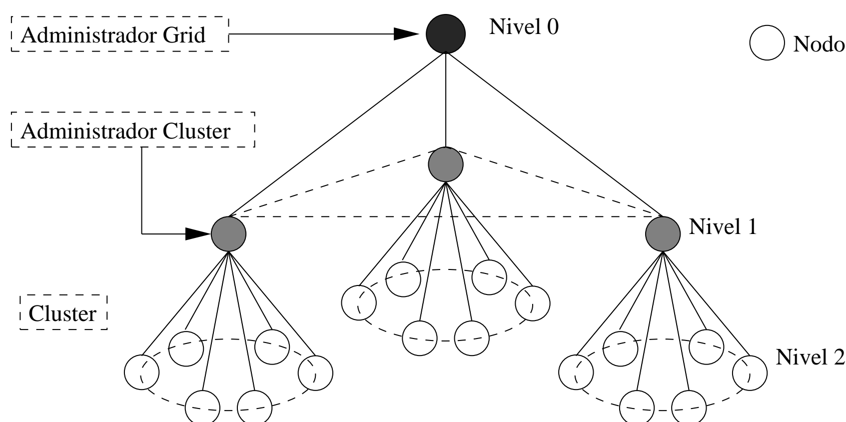

Figura 3.2: Modelo jerárquico para el balance de carga en un ambiente Grid

administrar (ejecutan a los procesos administradores). Por esta razón, en este capítulo y en los siguientes hablaremos de procesos trabajadores y cores trabajadores de forma indistinta.

Los procesos o cores están organizados de forma jerárquica en una estructura arborescente de tres niveles, como se puede apreciar en la Figura 3.2. Cada uno de los niveles realiza tareas de administración, procesamiento y monitoreo, según corresponda:

- ***Nivel 0:*** En este nivel se tiene al administrador de la Grid (se ejecuta en un core dedicado, solamente a la administración de la Grid), el cuál almacena información de la carga de trabajo de los clusters, esta información es proporcionada por los *Administradores Cluster* del nivel 1. A partir de esta información decide cuándo empezar el balance de carga *intra*-Grid o balance *inter*-cluster. La decisión se envía a los *Administradores Cluster* para que la ejecuten.

- ***Nivel 1:*** En este nivel tenemos a los administradores de los clusters (se ejecutan en cores dedicados, solamente a la administración del cluster), cada uno de ellos se encarga de mantener información de carga de los nodos bajo su dominio. Calcula la carga asociada al cluster y envía la información al *Administrador Grid* del nivel 0, y decide cuándo se debe hacer un balance de carga entre los nodos de un cluster, balance *intra*-cluster.

- ***Nivel 2:*** Cada nodo trabajador perteneciente a un cluster se encuentra en este nivel, su

labor principal es obtener la información de carga del nodo, enviarla al administrador del cluster y ejecutar las decisiones de balance, provenientes del *Administrador Cluster*.

En este modelo se hacen dos tipos de balance de carga, uno entre cores (balance *intra*-cluster) y otro a nivel Grid (balance *intra*-Grid o *inter*-cluster). Está diseñado para favorecer el balance de carga *intra*-cluster frente al balance *inter*-cluster, debido a que en el balance de carga, un nodo sólo puede buscar carga entre los nodos del mismo cluster (ya que el modelo impide que se comunique con nodos de otro cluster), solamente cuando el cluster ha terminado de procesar la carga que se le asignó puede pedir carga a un cluster remoto. Con esto se limitan las comunicaciones a través de Internet y se favorecen las comunicaciones locales, que son menos costosas.

Como hemos visto a lo largo de este capítulo, la tecnología Grid nos da acceso a un gran número de recursos y el modelo jerárquico para el balance de carga, proporciona la estructura para diseñar una aplicación que pueda balancear carga entre diferentes clusters. Por esta razón, se usarán estos dos enfoques para hacer las modificaciones a DLML, con el objetivo de poder balancear carga entre diferentes clusters.

Sin embargo, como hemos observado una Grid es un sistema complejo y difícil de administrar, que proporciona un gran número de servicios como seguridad, monitoreo de recursos, comunicación y colaboración, administración de recursos y administración de flujos de trabajo. De estos servicios, sólo la comunicación y colaboración entre los diferentes clusters, está dentro de los alcances para esta tesis, por lo cual, se buscó una alternativa para la comunicación *inter*-cluster (para no tener que usar un sistema tan complejo). La alternativa de comunicación *inter*-cluster es la Virtual Private Network (VPN) [22], que como se verá en el siguiente capítulo, proporciona un mecanismo mediante el cual podemos comunicar varios cluster de una forma relativamente sencilla.

En el siguiente capítulo veremos el funcionamiento de la VPN, el diseño de la nueva arquitectura para DLML y los algoritmos de balance de carga propuestos para la nueva versión de DLML.

Capítulo 4

Propuesta de DLML para un ambiente Grid

En este capítulo presentamos la propuesta de la nueva arquitectura y los algoritmos para que DLML pueda balancear carga en un ambiente Grid. Veremos cómo se implementa la arquitectura usando una biblioteca para paso de mensajes, y como se crea el ambiente Grid por medio de una VPN (Virtual Private Network).

4.1. Introducción

Para la propuesta de esta tesis retomaremos el modelo jerárquico para el balance de carga descrito en la Sección 3.5, considerando que este modelo es uno de los más usados para el balance de carga en un ambiente Grid. A diferencia de una Grid como Globus o Glite [23], un ambiente Grid no cumple completamente con los características descritas en [14] para identificar a una Grid, debido a que no incorpora múltiples calidades de servicio y tampoco protocolos e interfaces de propósito general. No obstante, si cumple con el requisito de tener una coordinación de recursos que no está sujeta a un control centralizado, esto quiere decir que para este sistema, se integran y coordinan recursos y usuarios que pertenecen a diferentes controles de domino, en donde cada uno tiene sus propias políticas de seguridad y administración. Este modelo es adecuado para los alcances de este proyecto, debido a que se pretende extender el sistema actual DLML que funciona en un sólo cluster, a un ambiente

multi-cluster, con el objetivo de acceder a un mayor número de recursos, que pueden estar presentes en diferentes clusters.

Para crear el ambiente Grid se propone usar una VPN, por medio de la cual, se interconectaran clusters separados geográficamente en la ciudad de México. De esta forma, los recursos de los clusters pertenecientes a la VPN pertenecerán a un cluster virtual, ya que la VPN permite que los recursos de los cluster, se comporten como si pertenecieran a un cluster que engloba a todos los recursos de dichos clusters. En este cluster virtual se podrán ejecutar aplicaciones (como aplicaciones MPI), de la misma forma que en un cluster normal. Sin embargo, sus recursos se comunicarán por el enlace creado por la VPN a través de Internet. Sobre la VPN se desarrollará la nueva arquitectura de DLML usando LAM/MPI para su implementación.

4.2. Arquitectura DLML para un ambiente Grid

Para el diseño de la nueva arquitectura, proponemos integrar el modelo jerárquico para el balance de carga (descrito en la Sección 3.5.1), al sistema actual de DLML, con el propósito de proporcionar balance de carga a nivel Grid. En este modelo se ve a la Grid como un sistema de clusters, en donde cada cluster posee un conjunto de cores trabajadores y un conjunto de cores administradores. Los cores trabajadores ejecutan a los procesos encargados de procesar carga, y los cores administradores, ejecutan a los procesos dedicados a la administración del balance de carga entre clusters. En el conjunto anterior, se encuentra el AG (*Administrador Grid*) y un AC (*Administrador Cluster*) por cada cluster, el AG se encarga de tomar las decisiones de balance de carga *inter*-cluster y los AC se encargan de ejecutar las decisiones de balance del AG y proporcionar el estado de carga de su cluster.

Al igual que en el enfoque estudiado, los cores trabajadores y administradores están organizados en una estructura arborescente (Figura 4.1). En esta estructura, los cores trabajadores pertenecen a un dominio local (a una red LAN), y cada cluster mantiene comunicación con otros clusters a través de Internet. La estructura arborescente tiene como principal objetivo establecer una jerarquía entre el balance de carga a nivel Grid y el balance de carga a nivel

cluster. En esta jerarquía, se le da prioridad al balance de carga a nivel cluster (*intra*-cluster), y se toma en cuenta el balance de carga *inter*-cluster, sólo cuando ya no es posible obtener carga dentro del mismo cluster. Esto se debe a la diferencia en el costo de las comunicaciones locales (en una red LAN) y las comunicaciones remotas a través de Internet (más costosas que las comunicaciones locales).

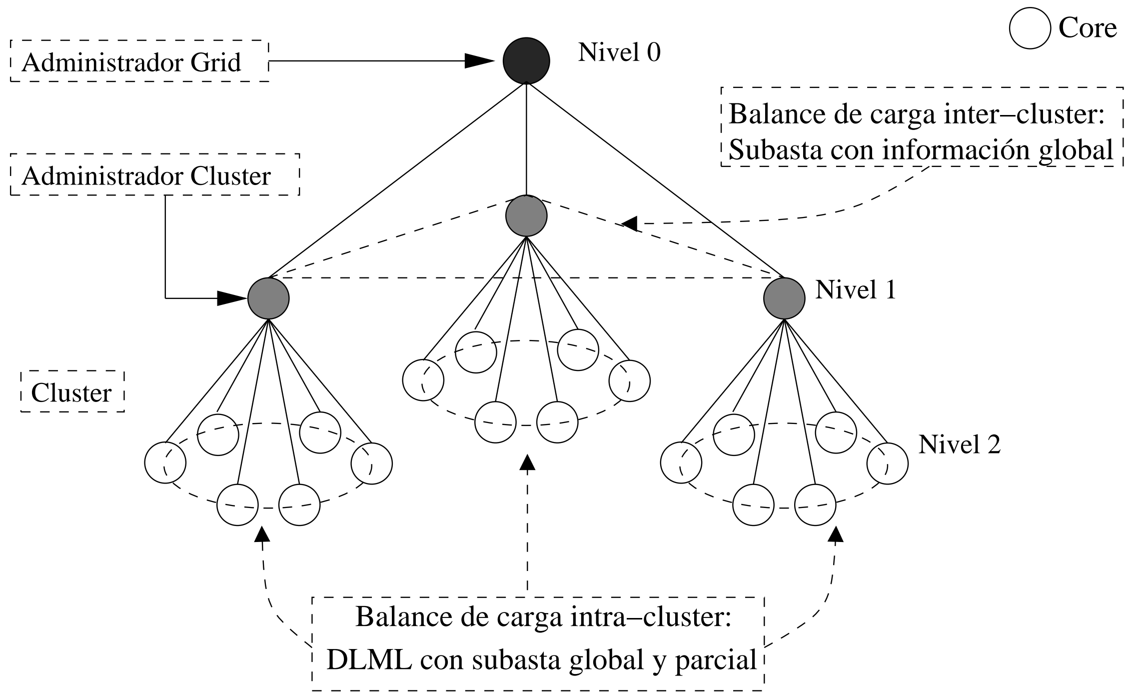

Figura 4.1: Modelo para el balance de carga de DLML en un ambiente Grid

Como se recordará, en el modelo jerárquico para el balance de carga (estudiado en la Sección 3.5.1), el balance de carga en cada cluster está dirigido por el AC de forma centralizada. Sin embargo, para la nueva arquitectura de DLML, el balance de carga *intra*-cluster está a cargo de los algoritmos de balance de carga, con subasta global o parcial descritos en el Capítulo 2. De este modo, en la nueva versión de DLML los cores trabajadores (nivel 2 de la Figura 4.1) no sólo procesan carga, también están encargados de balancearla localmente (balance de carga *intra*-cluster) usando los procesos *DLML* de las versiones anteriores. Mientras que para el balance de carga *inter*-cluster se hace una subasta global, esto significa, que se toma en cuenta el estado de carga de todos los clusters del sistema.

Como se puede observar en la Figura 4.1, este modelo cuenta con tres niveles, los cuales se describen a continuación:

- ***Nivel 0:*** En este nivel está el *Administrador Grid*, el cual tienes las siguientes funciones:
  - Recolecta información de carga de los *Administradores Cluster*.
  - Toma las decisiones de balance *inter*-cluster, es decir, evalúa cuál de los clusters puede donar carga a algún cluster descargado.
  - Se comunica con los *Administradores Cluster* para ordenar la donación de carga a algún cluster descargado.
  - Recolecta los resultados parciales provenientes de los *Administradores Cluster* en el nivel 1.
- ***Nivel 1:*** En este nivel tenemos a los *Administradores Cluster*, los cuales tienen las siguientes funciones:
  - Recolectan información de carga de los cores trabajadores bajo su dominio.
  - Evalúan la carga total del cluster y la envían al *Administrador Grid*.
  - Se comunican con otro *Administrador Cluster*, para recibir o enviar carga, atendiendo la decisión tomada en el nivel 0.
  - Se comunican con los cores trabajadores bajo su dominio, para recolectar o enviarles carga, en caso de un desbalance.
  - Recolectan los resultados parciales de los cores trabajadores y los envían al *Administrador Grid*.
- ***Nivel 2:*** A este nivel pertenecen todos los cores designados para el procesamiento o cores trabajadores, sus principales funciones son:
  - Envían su información de carga al *Administrador Cluster* cuando lo solicita.
  - Procesar y balancear carga a nivel cluster, usando el algoritmo de subasta con manejo de información global o parcial.

- Se coordinan con el *Administrador Cluster* para enviar o recibir carga en caso de un desbalance.
- Envían sus resultados parciales al *Administrador Cluster*.

Como podemos observar, para poder realizar el balance de carga *inter*-cluster (siguiendo este modelo) es necesario contar con un *Administrador Grid*, un *Administrador Cluster* y un conjunto de procesos o cores trabajadores. Los procesos trabajadores ya están presentes en las versiones anteriores de DLML, sólo es necesario integrar al *Administrador Grid* y al *Administrador Cluster* para obtener la nueva arquitectura de DLML, como se muestra en la Figura 4.2.

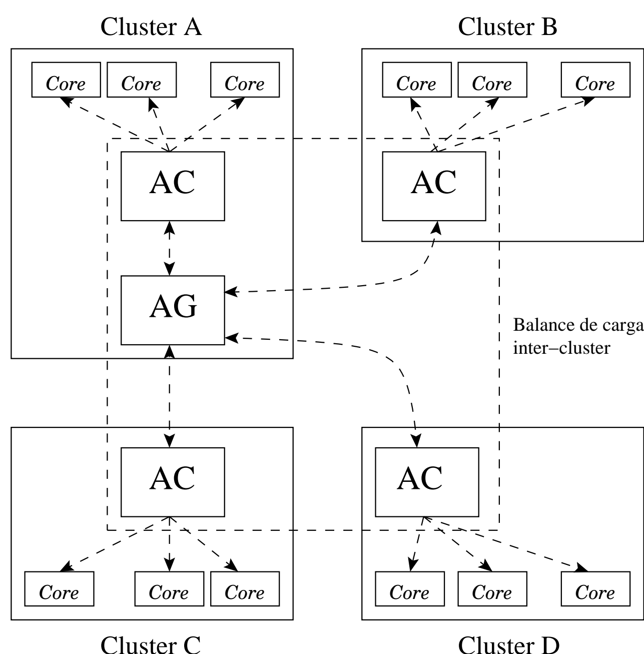

Figura 4.2: Nueva arquitectura DLML para el balance de carga en un ambiente Grid

En esta Figura tenemos un ejemplo de un sistema con cuatro clusters, en el cluster A tenemos al *Administrador Grid* y al *Administrador Cluster* del cluster A, mientras que en los tres clusters restantes sólo tenemos a un *Administrador Cluster*. También podemos ver que los *Administradores Cluster* se comunican y coordinan a través de Internet con el *Administrador Grid*, excepto el *Administrador Cluster* del cluster A, ya que está presente en el mismo

cluster que el *Administrador Grid*. En conjunto se tienen 5 cores administradores encargados del balance de carga *inter*-cluster, y un conjunto de cores destinados al procesamiento.

En esta nueva arquitectura se retoman los elementos de la arquitectura anterior de DLML, en donde se tiene por cada core, un proceso *Aplicación* y un proceso *DLML*, encargados del procesamiento y del balance de carga *intra*-cluster respectivamente. En la Figura 4.3 podemos ver cómo se integran todos los procesos en la nueva arquitectura de DLML, con el propósito de ofrecer balance de carga a dos niveles, el *intra*-cluster y el *inter*-cluster.

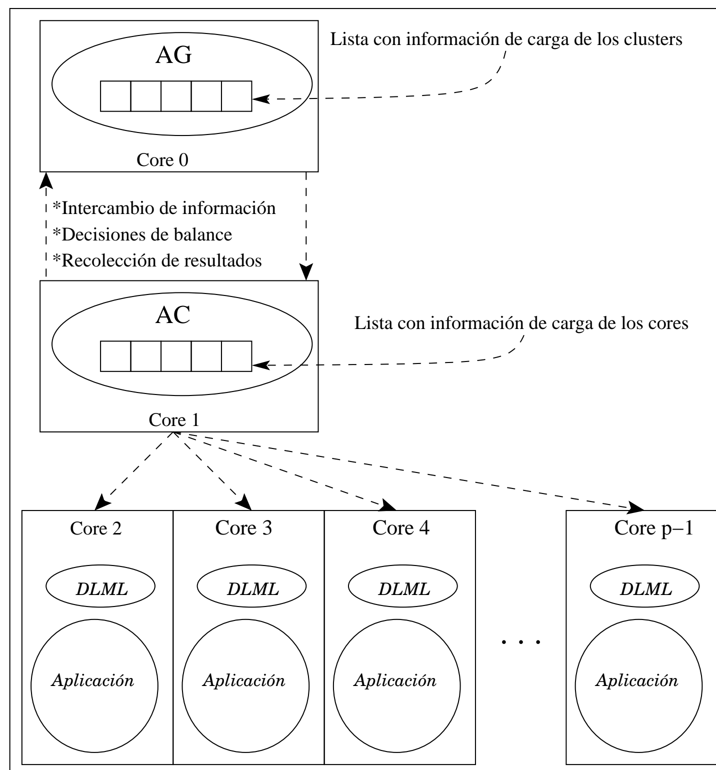

Figura 4.3: Integración del AG, AC, *Aplicación* y proceso *DLML* en la nueva arquitectura

En esta Figura se puede apreciar la jerarquía de comunicación entre los cores trabajadores y los cores administradores. Los cores trabajadores sólo mantienen comunicación con el *Administrador Cluster* de su propio cluster, esto significa que los cores trabajadores de un cluster, no se pueden comunicar con cores trabajadores de otro cluster.

De esta forma, la única vía para acceder a la carga de otro cluster es a través de los *Administradores Cluster*, los cuales mantienen comunicación directa con el *Administrador Grid*

y entre ellos. Sin embargo, la comunicación entre *Administradores Cluster* está restringida al intercambio de carga y sólo por la orden del *Administrador Grid*. En esta Figura también podemos apreciar, que a través de los *Administradores Cluster*, el *Administrador Grid* obtiene información y hace la recolección de resultados parciales al finalizar el procesamiento.

Por otra parte, se debe tomar en cuenta que en la nueva arquitectura es necesario que los procesos *DLML* tengan nuevas funciones. Esto se debe a que cada core trabajador, necesita mantener comunicación con su *Administrador Cluster*, ya sea para intercambiar información de carga o para el envió y recepción de carga, Figura 4.4.

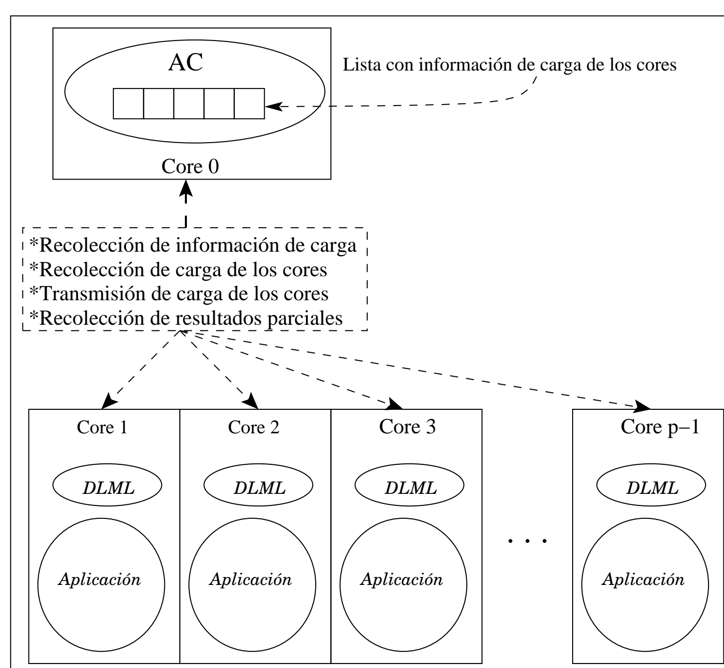

Figura 4.4: Comunicación entre el *Administrador Cluster* y los cores trabajadores

Como se mencionó en el Capítulo 2, en las versiones anteriores de DLML la detección de un cluster descargado es importante para la recolección de resultados y para finalizar el procesamiento. En la nueva versión, la detección de un cluster descargado es importante para iniciar la búsqueda de carga en otros clusters.

En la versión de DLML con subasta global, cualquier proceso *DLML* puede detectar que no hay carga en el cluster, mientras que en la versión de DLML con subasta parcial, sólo el

DLML 0 es el que puede detectar si el cluster está descargado. En la nueva versión de DLML, se mantiene este mismo esquema, sin embargo, cuando se detecte la descarga del cluster, en lugar de iniciar la terminación (como en las versiones anteriores) se hará una petición de carga al *Administrador Cluster*, para que éste a su vez solicite carga al *Administrador Grid* e inicie un balance de carga *inter*-cluster.

4.3. Balance de carga Inter-cluster

Ahora que se posee una arquitectura capáz de intercambiar información mediante envió y recepción de mensajes, se definen las políticas mediante las cuales se hace el balance de carga. Se tiene al igual que a nivel core, una subasta iniciada por el receptor, sin embargo, en este algoritmo, el balance de carga *inter*-cluster se hace en coordinación con el *Administrador Grid*, el cuál aplicará las siguientes políticas de balance:

- **Política de información:** Como se mencionó en el Capítulo 2, DLML es una biblioteca diseñada para balancear carga, en aplicaciones que usan listas (con elementos independientes) para representar sus datos. En las versiones anteriores de DLML, la política de información para el balance de carga entre cores del cluster es el número de elementos (de la lista) que le faltan por procesar al proceso *Aplicación*. Por lo cual para esta versión DLML, se toma como carga, la suma de elementos de las listas que se encuentran procesando los procesos *Aplicación*, en cada uno de los cores trabajadores.

  En cuanto al cómo se recolecta la información, en esta versión se hace de forma global, esto quiere decir que el *Administrador Grid* recolecta información de carga (a través de los *Administradores Cluster*) de cada uno de los cluster para tomar una decisión de balance.

- **Política de transferencia:** La transferencia de carga se inicia cuando uno de los *Administradores Cluster* pide carga al *Administrador Grid*. Este último identifica cual de los cluster tiene más carga, para ordenarle una donación de carga al *Administrador Cluster* solicitante. Por tanto, el que se encarga de distribuir la carga es el cluster que

reporta más carga al inicio de la subasta. Una vez que se confirma la transferencia, el *Administrador Grid* termina su papel en el balance y puede seguir atendiendo peticiones de otros clusters, incluso mientras se hace la transferencia de carga del cluster emisor al cluster receptor.

Aunque se debe tomar en cuenta que el cluster seleccionado, podría no ser el que tiene más carga al momento de la transferencia, ya que mientras se hace la recolección de información de carga y la evaluación del cluster con mayor carga, los cores trabajadores (de los diferentes clusters) continúan el procesamiento de su carga. Esto significa que la información de carga reportada al inicio de la subasta, seguramente será diferente al momento de la transferencia.

- **Política de selección:** La carga que se va a transferir es un porcentaje de la carga total del cluster, este porcentaje dependerá de la capacidad de procesamiento del cluster receptor y del cluster emisor. Para este trabajo, la capacidad de procesamiento de un cluster se calcula sumando la capacidad de procesamiento de sus cores trabajadores.

  El porcentaje de carga a transferir sigue las siguientes reglas:

  - Si el cluster receptor tiene menor capacidad de procesamiento que el cluster emisor, el cluster emisor transmitirá un porcentaje menor al 50 % de su carga.
  - Si el cluster receptor y el cluster emisor cuentan con la misma capacidad de procesamiento, el cluster emisor hará una transferencia del 50 % de su carga.
  - Si el cluster receptor cuenta con mayor capacidad de procesamiento, el cluster emisor transmitirá más del 50 % de su carga.

- **Política de localización:** La carga se distribuirá al cluster descargado (receptor), desde el cluster que haya sido elegido por el *Administrador Grid* como cluster emisor.

Basándose en las políticas mencionadas y la información de carga de cada uno de clusters, el *Administrador Grid* toma las decisiones de balance de carga. Sin embargo, también evalúa si la carga reportada por los clusters es lo suficientemente grande para considerar una migración

de datos. Esta evaluación se hace con la información de carga de los clusters, si la carga reportada como mayor no cumple con el mínimo preestablecido para aceptar la migración de datos, ésta no se realizara. Debido a que este caso se presenta cuando existe poca carga en el sistema, y una migración de carga (en estas condiciones) sólo retrasaría el procesamiento, en vez de agilizarlo. La forma en que se establece el mínimo, y los algoritmos donde se aplican las políticas de balance se describen más adelante.

Subasta iniciada por el cluster receptor

Un aspecto a considerar en el balance de carga *inter*-cluster, es que al inicio del procesamiento sólo uno de los clusters posee carga (cluster A de Figura 4.5), los demás clusters, deben pedir carga al *Administrador Grid*, a través de su *Administrador Cluster* (cluster B y C de la Figura 4.5). Para dar pie a la distribución de carga y su posterior procesamiento en todos los clusters del sistema.

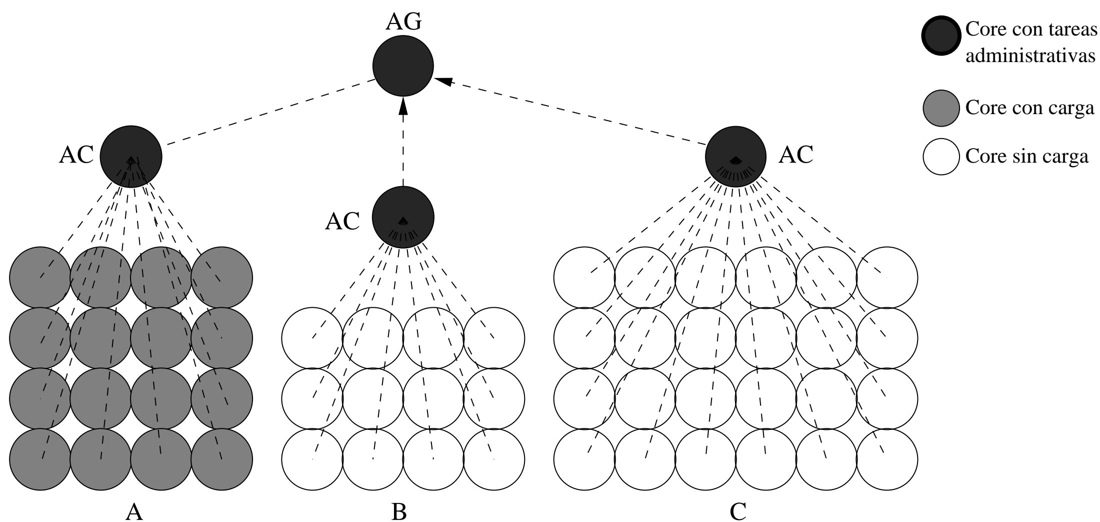

Figura 4.5: Carga al inicio del procesamiento

Como podemos observar el balance de carga *inter*-cluster es necesario desde el inicio y hasta el final del procesamiento. Al inicio y durante el procesamiento, el balance se hace tomando en cuenta a los clusters que se quedan sin carga. Si uno de los cluster termina de procesar la carga que se le asignó, subasta su poder de cómputo a los demás clusters del

sistema, de la misma forma, que en la versión de DLML con subasta global. Esto quiere decir que en esta versión, el balance de carga *inter*-cluster se hace recolectando información de carga de todos los clusters del sistema, y a partir de esta información, se toma una decisión de balance para redistribuir la carga. Las etapas para la distribución de carga son: búsqueda de carga local, búsqueda de carga externa, selección del cluster emisor y redistribución.

- **Búsqueda de carga local**

  En esta etapa, los cores trabajadores identifican que el cluster al que pertenecen, esta descargado. Ante esta situación, uno de los cores trabajadores envía una petición de carga (a través de su proceso *DLML*) a su *Administrador Cluster*. El core encargado de hacer la petición al *Administrador Cluster*, dependerá del algoritmo que se esté usando para el balance de carga *intra*-cluster. Si se usa la subasta global, cualquier core puede hacer la petición de carga al *Administrador Cluster*. Sin embargo, si se trata de la subasta parcial, sólo el primer core trabajador podrá hacer la petición. Esto se debe a que en esta versión, sólo el proceso *DLML 0* es capaz de identificar la descarga del cluster.

  Cuando el *Administrador Cluster* recibe la petición de carga de uno de sus cores trabajadores, hace una petición de carga al *Administrador Grid*, quién a su vez, inicia la búsqueda de carga externa.

- **Búsqueda de carga externa**

  Esta etapa está dividida en dos partes, la primera inicia cuando el *Administrador Grid* recibe una petición de carga de algún *Administrador Cluster* descargado. Para responder a esta petición, el *Administrador Grid* pide información de carga a todos los clusters del sistema (excepto al cluster que solicitó carga), y espera por las respuestas de los clusters. Si durante la espera, algún otro cluster pide carga, el *Administrador Grid* lo encola en una cola de peticiones para atenderlo más tarde. El algoritmo que usa el *Administrador Grid* para la recolección de información de carga es el siguiente:

```
proc Recolect_Info_Carga_Clusters(N, V C, V IC)
    (N : # clusters, V C: Clusters disponibles, V IC: Información de carga)
    para i = 0 hasta N – 1 haz
        si i ≠ S entonces
            (a S no se le hace la petición porque es el que pide carga)
            peticion_info_carga_clusters(V C[i])
        termina
    termina
    para i = 0 hasta N – 2 haz
        espera_mensaje_cualquier_cluster()
        si tipo_mensaje = info_carga entonces
            (el mensaje contiene información de carga)
            V IC[i][0] = id_cluster_envia_mensaje
            V IC[i][1] = info_carga
        otro
            (si llega una peticion de carga se encola)
            encola_id(id_cluster_envia_mensaje)
            V IC[i][0] = id_cluster_envia_mensaje
            V IC[i][1] = 0
        termina
    termina
termina
```

En el algoritmo anterior, podemos observar que el *Administrador Grid* usa dos vectores, VC para guardar los ID's de los *Administradores Cluster* y VIC para almacenar la información de carga de cada *Administrador Cluster*. Con la información del primer vector pide información de carga a todos los clusters, excepto a S que es una variable que almacena el ID del cluster solicitante. Posteriormente espera N-1 respuestas (ya que el cluster que solicito no tiene por qué responder), si una de esas respuestas, es una

petición de carga de otro cluster, el *Administrador Grid* encola el ID del cluster, y le asigna automáticamente una carga igual a 0, en el vector de VIC.

La segunda parte de esta etapa, inicia cuando los *Administradores Cluster* reciben la petición de información de carga del *Administrador Grid*. En ese momento, los *Administradores Cluster* inician la recolección de información de carga usando el siguiente algoritmo:

```
proc Recolect_Info_Carga_Cores_T (id ini, id f in)
    ( id_ini: id del primer core trabajador,  id_fin: id del ultimo core trabajador )
    para i = id_ini hasta id_fin haz
        peticion_info_carga[i]
    termina
    para i = id_ini hasta id_fin haz
        espera_mensaje_cualquier_core()
        info_carga = info_carga + info_carga_core_trabajador
    termina
    envia_AGrid(info_carga)
termina
```

Con este algoritmo el *Administrador Cluster* pide información de carga a cada uno de los cores trabajadores, para hacer esto envía una petición a los procesos *DLML* usando sus ID's (variables id_ini y id_fin). Los *DLML* responde enviando su información de carga al *Administrador Cluster*, quién suma todas las respuestas en la variable info_carga, para obtener la carga total del cluster, y la envía al *Administrador Grid*. En esta etapa, si un cluster no tiene carga y recibe la petición de información de carga del *Administrador Grid*, responde a la petición con una carga igual a 0. Este caso sucede, cuando un cluster hace una petición de carga después del cluster al que se está atendiendo.

Cuando el *Administrador Grid* obtiene todas las respuestas inicia la evaluación del cluster emisor, en la siguiente etapa.

- **Evaluación del cluster emisor**

Para tomar una decisión de balance (con la información recolectada en la etapa anterior) el *Administrador Grid* evalúa cuál de los clusters tiene mayor cantidad de carga. El cluster con mayor carga será designado como cluster *emisor*, y el cluster solicitante como *receptor*. La determinación del cluster *emisor* junto con la evaluación de la transferencia, es lo que constituye la decisión de balance de carga.

Al terminar la evaluación, el *Administrador Grid* ordenará al cluster con mayor carga, hacer una transferencia de carga hacia el cluster *receptor*. El algoritmo para evaluar al cluster emisor es el siguiente:

```
proc Evaluacion_Cluster_Emisor(N, V IC)
    ( N : # clusters, V IC: Vector con Información de carga )
    carga_mayor = V IC[0][1]
    cluster_emisor = V IC[0][0]
    para i = 1 hasta N – 1 haz
        si carga_mayor < V IC[i][1] entonces
            carga_mayor = V IC[i][1]
            cluster_emisor = V IC[i][0]
        termina
    termina
termina
```

En caso de que la carga (más grande) sea mayor a 0, se debe evaluar al cluster emisor, para verificar que la carga reportada cumpla con el requisito para la transferencia. Si no se cumple con el requisito, la transferencia no se realiza, y el ID del cluster solicitante se ingresa a la cola de peticiones del *Administrador Grid*, para ser atendida más adelante.

El algoritmo que usa el *Administrador Grid*, para evaluar la transferencia de carga se presenta a continuación:

```
proc  Evaluacion_Transferencia(carga mayor, V IC, cluster_emisor)
      ( N :  # clusters,  V IC:  Vector con Información de carga  )
      si  carga _mayor > 0  entonces
         si  carga proviene de al menos el 50 % de los cores trabajadores entonces
            (Se ordena al cluster_emisor hacer una transferencia a  S)
         otro
            encola_id(S)
         termina
      otro
         (Se avisa a todos los clusters disponibles que no hay carga)
      termina
termina
```

Con este algoritmo se verifica el valor de "carga_mayor". Si es igual a 0, la carga en el sistema se habría terminado. Sin carga en el sistema, el *Administrador Grid* pide los resultados parciales a todos los *Administradores Cluster*.

En caso contrario (carga_mayor >0), se verifica que la carga reportada provenga de al menos el 50 % (este valor se escogió en base al comportamiento de la aplicación, que puede ser estática o dinámica) de los cores trabajadores (del cluster emisor). Si 50 % o más del 50 % de los cores reportaron carga, significa que hay suficiente carga para ser repartida entre dos clusters (emisor y receptor). Si no fuese así, es probable que el cluster emisor esté cerca de terminar el procesamiento de su carga (sin ninguna ayuda), y por consiguiente, una distribución hacia el cluster receptor, en lugar de reducir el tiempo de procesamiento, sólo lo retrasaría e incrementaría el tiempo de procesamiento global. Este retraso es consecuencia de las comunicaciones que involucra una redistribución de carga y de la interrupción (en el procesamiento) que se hace al cluster emisor, al momento de pedirle la carga.

Un ejemplo del caso anterior, se presenta cuando la carga del sistema se ha reducido de tal forma, que es insuficiente para ser distribuida entre todos los clusters, en este caso

se presentan peticiones y distribuciones de un cluster a otro constantemente, acarreado un retraso en el tiempo de ejecución global, al aumentar el costo en las comunicaciones. Sin embargo, si la carga proviene de al menos el 50 % de los cores trabajadores, la distribución de la carga ayuda a disminuir el tiempo de procesamiento global.

- **Redistribución de carga**

  La etapa de redistribución de carga inicia cuando el *Administrador Grid* ordena a uno de los clusters (emisor) hacer una transferencia de carga a otro cluster (receptor), a través de sus *Administradores Cluster*.

  Para hacer la transferencia el *Administrador Cluster* emisor hace una petición de carga a sus cores trabajadores, los cores responden enviando un porcentaje de sus listas de carga al *Administrador Cluster*, quién a su vez recopila todas las listas en una lista general para enviarla al *Administrador Cluster* receptor a través de Internet. El algoritmo que usa el cluster emisor es el siguiente:

proc *Transferencia_Carga_Cluster_R(S, id_ini, id_fin)*

    (*S:* id del Administrador Cluster receptor, *id_ini:*id del primer core trabajador,
    id f in: id del ultimo core trabajador)

    para *i* = *id_ini* **hasta** *id_fin* **haz**

        (Se pide una parte de la carga al core trabajador i)

        peticion_carga(i)

    termina

    para *i* = *id_ini* **hasta** *id_fin* **haz**

        (Se reciben las listas)

        lista_parcial = recibe_lista_cualquier_core()

        lista_carga_total = lista_carga_total + lista_parcial

        longitud_lista_total = longitud_lista_total + longitud_lista_parcial

    termina

①

```
①
    envia_AGrid(longitud_lista_total)
    si longitud_lista_total > 0 entonces
        envia_lista_cluster_receptor(lista_carga_total)
    termina
termina
```

El porcentaje de carga que se envía, dependerá de la diferencia en la capacidad de procesamiento, entre los dos clusters. Cuando los *DLMLs* de los núcleos trabajadores reciben la petición de carga, acceden a la lista de datos de la *Aplicación*, dividen la lista en dos partes, una parte se envía al *Administrador Cluster* y la otra se queda con la *Aplicación*.

Por su parte el *Administrador Cluster* receptor, recibe la carga enviada por el emisor y la distribuye a sus cores trabajadores con el siguiente algoritmo:

```
proc Recepcion_Carga_Cluster_E(id_ini, id_fin)
    (id_ini: id del primer core trabajador,   id_fin: id del ultimo core trabajador )
    lista_externa = recepcion_lista_externa()
    distribuye_lista_externa(lista_externa,id_ini, id_fin)
termina
```

Con los algoritmos anteriores se finaliza la subasta del cluster descargado, esta subasta se hará cada vez que exista un cluster descargado en el sistema, ya sea al inicio o a lo largo del procesamiento.

Una explicación gráfica del funcionamiento de los algoritmos presentados, la encontramos en la Figura 4.6, en donde tenemos un sistema de tres clusters, A, B y C. La subasta *inter*-cluster inicia cuando el cluster A se queda sin carga (recibe una petición de carga de alguno de sus cores trabajadores). En ese momento, el *Administrador Cluster* (AC) del cluster A hace una petición de carga al *Administrador Grid* (AG) (Figura 4.6.a). Ante esta petición el AG pide la información de carga a todos los AC, excepto al cluster A quien pidió carga (Figura

4.6.b). Los AC de los clusters B y C piden información de carga a sus cores (Figura 4.6.c). Los cores envían su información de carga a sus AC (Figura 4.6.d). Los AC reciben y suman la información recibida de sus cores y la envían al AG (Figura 4.6.e). Con la información recibida, el AG evalúa cuál de los clusters posee más carga, en este caso es el cluster C, por lo cual, el AG indica al AC del cluster C enviar parte de su carga al AC del cluster A (Figura 4.6.f).

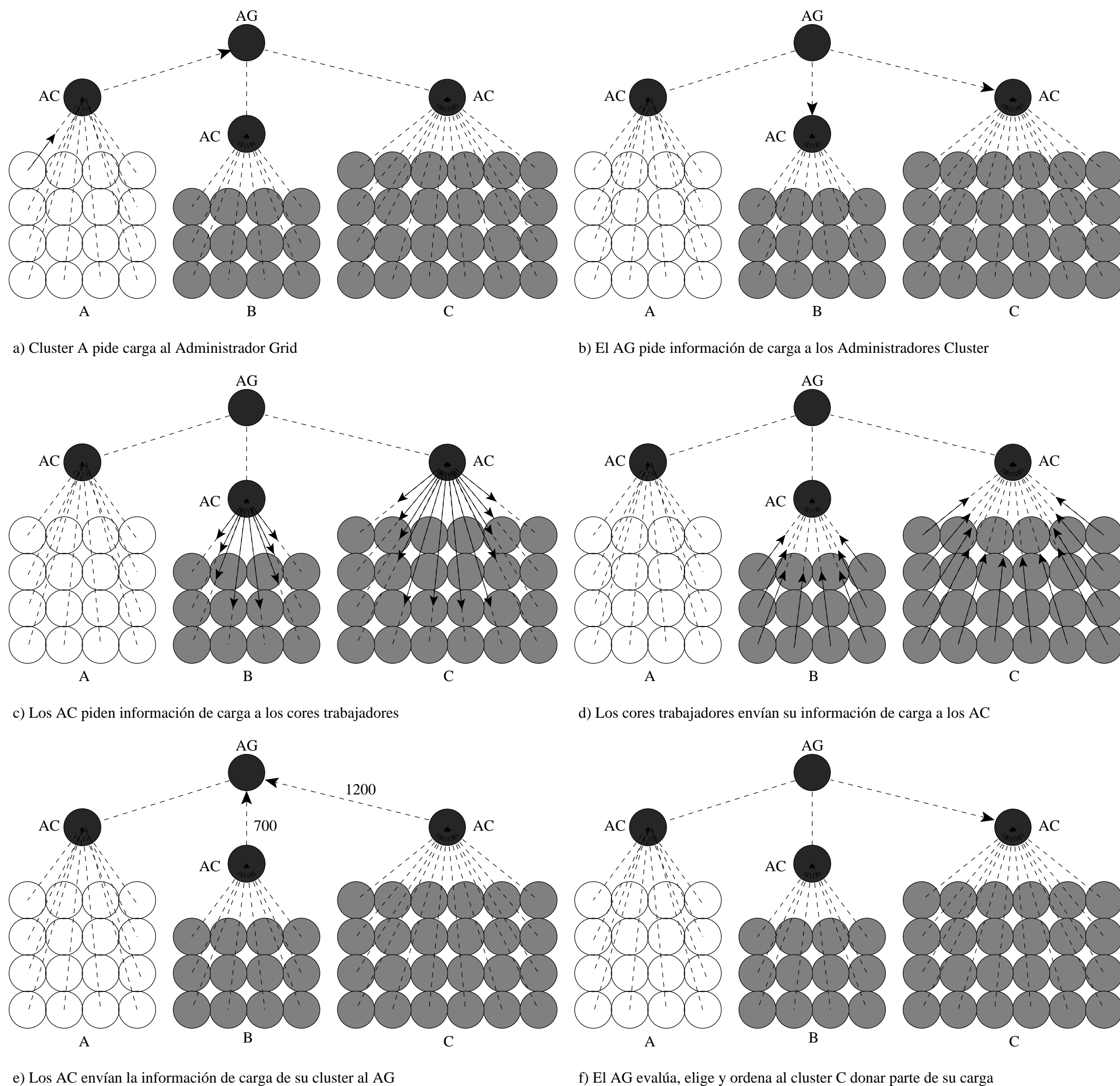

Figura 4.6: Subasta iniciada por el cluster receptor

Cuando el *Administrador Cluster* del cluster C recibe la orden de donar parte de su carga

al cluster A, transmite la orden de transferencia a sus cores trabajadores (Figura 4.7.g). Los cores responden enviando una parte de su lista de carga al AC, (Figura 4.7.h). El AC une todas las listas de los cores formando una lista general, esta lista contiene la carga total a transferir. Finalmente el AC del cluster C, envía la lista al AC del cluster A (Figura 4.7.i).

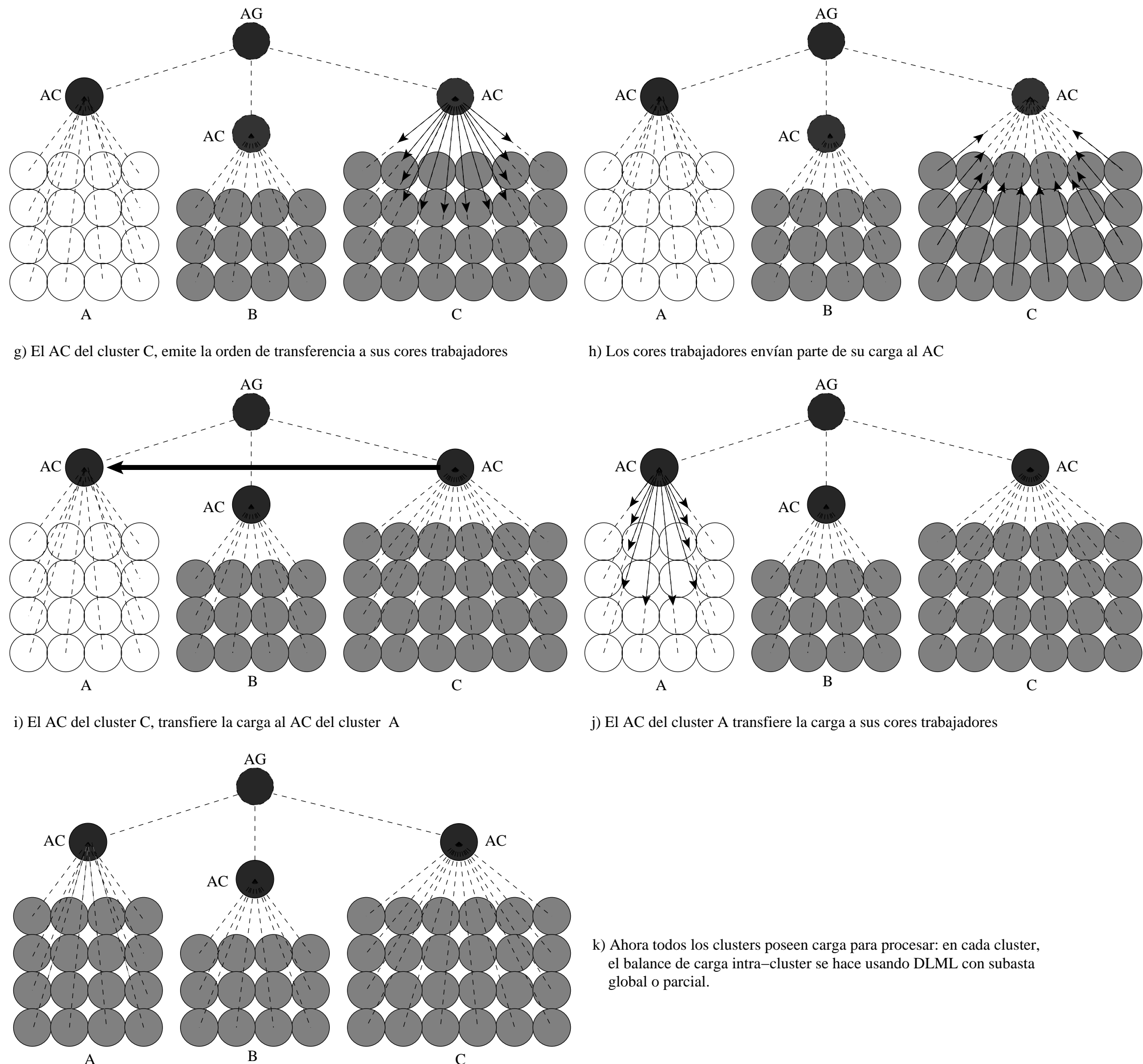

Figura 4.7: Redistribución de la carga, después de una subasta

Cuando el AC del cluster A recibe toda la lista la distribuye a sus cores trabajadores, esta distribución depende de la versión de DLML que se está usando para el balance *intra*-cluster. Para DLML con subasta global, la carga es dividida y distribuida entre todos los cores

trabajadores, mientras que para DLML con subasta parcial, la carga es enviada al DLML 0 (quien se encarga de distribuir la carga a los otros *DLML's*), esto se debe a las reglas de balance que usa esta versión de DLML (Figura 4.7.j). Al final en la redistribución de carga, el cluster que inició la subasta ha obtenido carga para continuar con el procesamiento de sus datos (Figura 4.7.k).

Con el balance iniciado por el cluster receptor, se completa el balance de carga a nivel Grid para las nuevas versiones de DLML, a las que llamaremos Glo-Grid (que usa balance iniciado por el cluster receptor y la subasta global para el balance *intra*-cluster) y Toro-Grid (que usa balance iniciado por el cluster receptor y la subasta parcial para el balance *intra*-cluster). Sin embargo, debemos tomar en cuenta que para realizar el balance *inter*-cluster, la comunicación de un *Administrador Cluster* a otro es directa, es decir, no se usan técnicas de tipo store and forward, para enviar y recibir información o carga de un cluster a otro. Esto es posible gracias a que se usa una VPN para establecer la comunicación *inter*-cluster.

4.4. Comunicación inter-cluster

Como ya se mencionó, en la nueva arquitectura se hace balance de carga entre clusters, para esto se debe contar con un medio de comunicación como Internet o un enlace dedicado. En nuestro caso el medio de comunicación, entre los clusters será Internet. Para ello se construye una VPN (Virtual Private Network) [24], que es una red virtual por medio de la cual se pueden comunicar dos o más redes LAN a través de Internet. Esta red es virtual debido a que no existe una conexión directa entre las redes. Sin embargo, mediante el uso de una VPN cada elemento que pertenezca a ella, se comporta como lo harían elementos pertenecientes a una misma red LAN. Una VPN es privada, ya que cuenta con un mecanismo de encriptación, esto quiere decir, que el tráfico de datos que viaja a través de la VPN es encriptado por el emisor y desencriptado por el receptor después de la transmisión. En la Figura 4.8 podemos apreciar un túnel VPN entre dos redes LAN (A y B), a través de este túnel (que usa Internet para enviar y recibir datos) los recursos de ambas redes se pueden comunicar, como si pertenecieran a la misma LAN.

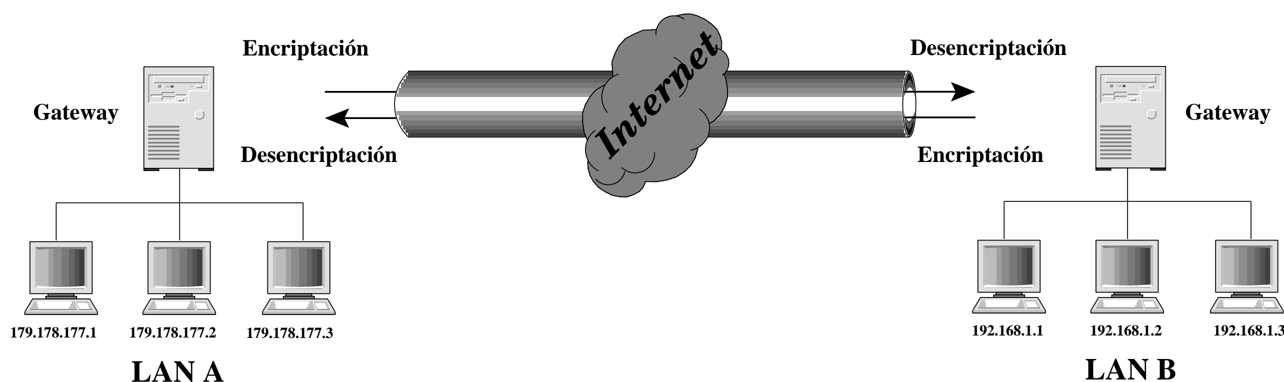

Figura 4.8: Túnel VPN entre dos redes LAN

La encriptación resguarda los datos transmitidos de una red a otra, al igual que los muros de un túnel protegen a un tren al pasar a través de una montaña. Esto explica porque a la Red Privada Virtual a menudo se le conoce simplemente como túnel o túnel VPN, y a la tecnología se le llama tunneling [25]. Cuando un elemento de la red A quiere enviar información a otro elemento de la red B, la información viaja por el túnel VPN formado entre las dos redes (Figura 4.8). Las dos redes pueden ubicarse en diferentes continentes, ya que el único requerimiento para poder crear la VPN entre las dos redes, es tener, con una conexión a Internet.

Tomando en cuenta que un cluster usa una red LAN para comunicar sus cores o nodos, se puede usar la VPN para comunicar clusters separados geográficamente a través de Internet sin ningún problema, más que el que se presentaría, con las políticas de seguridad y administración que existan en cada institución, a la que pertenezcan los clusters. Por ejemplo, si crea una VPN entre dos cluster podemos acceder a los recursos de los dos clusters, como si los recursos de ambos estuvieran presentes en un sólo cluster: se tendría acceso a un cluster virtual que poseería los recursos de los dos clusters (Figura 4.9). En este cluster virtual, se pueden ejecutar aplicaciones como en un cluster normal, por tanto, podemos ejecutar aplicaciones construidas en MPI, como DLML o cualquier otra aplicación paralela [22][26]. Para crear la VPN existen varios software, entre ellos esta OpenVPN [27], que es multiplataforma, es decir, con OpenVPN se puede construir una VPN que incluya elementos con diferentes arquitecturas y sistemas operativos, como en el caso de MAC OS, UNIX, Windows y Linux.

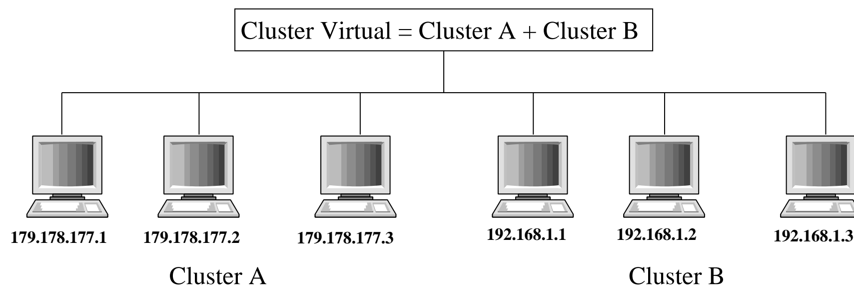

Figura 4.9: Cluster virtual para ejecutar aplicaciones paralelas

Para crear el ambiente Grid de nuestra propuesta usaremos OpenVPN, el cual es relativamente fácil de configurar. Para poder crear una VPN entre varios clusters se debe contar con los siguientes requisitos:

- Los clusters deben tener acceso a Internet a través de su servidor.
- Cada cluster debe tener una red LAN diferente.
- Cada cluster debe contar con una dirección IP estática, para mayor facilidad en su configuración.
- Los servidores deben permitir el acceso a un puerto de común acuerdo, para el caso de OpenVPN el puerto default es el 1194.
- Una cuenta de acceso, con el mismo nombre de usuario en todos los clusters, esta cuenta no necesita privilegios de root.

Con los requisitos anteriores se puede crear una VPN, en donde uno de los clusters será el servidor de la VPN y los demás serán Clientes VPN. Por ejemplo, si tenemos dos clusters como en la Figura 4.10, cluster A y cluster B, el cluster A se puede configurar como el Servidor VPN y el cluster B como el Cliente VPN. Al conectar los dos clusters por el puerto 1194 se generara un cluster Virtual, en donde podemos ejecutar aplicaciones paralelas sobre todos los recursos del cluster A y del cluster B.

De esta forma, OpenVPN crea un Gateway virtual por medio del cual, hace una redirección de las peticiones de la red del cluster A a las del cluster B y viceversa. Por ejemplo, si

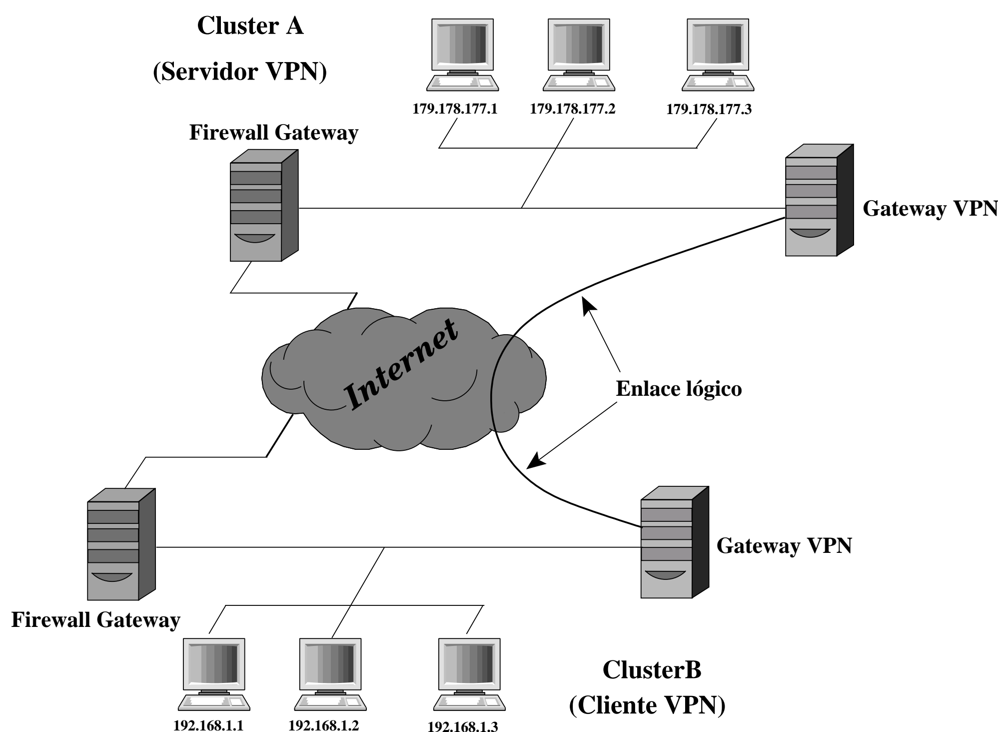

Figura 4.10: VPN formada por dos clusters, usando OpenVPN

el nodo con dirección 192.168.1.1 (al que llamaremos B1) quiere hacer una petición al nodo con dirección 179.178.177.1 (A1), el servidor del cluster B usa el Gateway virtual y envía la petición por el enlace lógico, como se muestra en la Figura 4.10. La petición llega al servidor del cluster A, y el servidor usa el Gateway virtual para enviar la petición al nodo A1. La petición es encriptada por el servidor del cluster B y desencriptada en el servidor del cluster A, como se mostró en la Figura 4.8. Para ver la configuración del Servidor VPN y el Cliente VPN ir al apéndice A.

Sobre la VPN (formada por los dos clusters), se pueden ejecutar aplicaciones paralelas del mismo modo que en un cluster normal. Una aplicación como DLML se ejecuta perfectamente sobre una VPN, como la que se muestra en la Figura 4.10. Sin embargo, al ejecutar aplicaciones paralelas sobre una VPN se debe tomar en cuenta, que en aplicaciones como DLML se usan mensajes para intercambiar información, y los mensajes que se hagan entre cores que pertenezcan a clusters diferentes, serán más costosos que los mensajes que se hagan

entre cores del mismo cluster. Esto se debe a que en estos mensajes (mensajes entre cores de diferentes clusters), interviene la banda ancha con la que cuentan cada uno de los clusters. Lo anterior significa, que si la banda ancha de uno de los clusters es lenta o está saturada (como suele ocurrir a ciertas horas del día), los mensajes entre los cores serán mucho más lentos, y se afectará el rendimiento de la aplicación.

Este problema lo podemos apreciar en la versión de DLML con subasta parcial, en donde se usa la topología lógica toroide para organizar la comunicación de los cores trabajadores. En esta versión de DLML, cada core sólo se puede comunicar con 4 cores vecinos, y el toroide puede adoptar diferentes configuraciones, en su número de filas y de columnas (toroide de MxN). En un cluster normal (donde todos los cores pertenecen a una red local y usan un switch de alta velocidad) la configuración en las dimensiones del toroide, no modifica el costo de las comunicaciones. Debido a que el costo en la comunicación de core a otro, dentro de un ámbito local (en un cluster) se le considera constante. Sin embargo, al ejecutar DLML con subasta parcial sobre una VPN, la configuración del toroide si afecta el tiempo de ejecución de la aplicación. Esto se debe a que las dimensiones del toroide aumentan o disminuyen el número de procesadores que intercambian información a través de Internet. Por ejemplo, supongamos que tenemos un cluster A con 64 cores trabajadores y otro cluster B con 20 cores trabajadores, si se formara una VPN entre los dos clusters se tendrían 84 cores para procesar la carga en forma simultánea.

Al ejecutar DLML con subasta parcial sobre los 84 cores de la VPN, las configuraciones más recomendables para la distribución de carga en el toroide son: M=12 y N=7 o M=7 y N=12 (ver apéndice B). Como se puede apreciar en la Figura 4.11, estas dos configuraciones forman diferentes toroides y por consiguiente diferente número de conexiones a través de Internet.

El número de mensajes a través de Internet varía según las conexiones lógicas establecidas entre los cores de diferentes clusters. Como se muestra en la Figura 4.12, en donde tenemos 16 conexiones lógicas entre cores de dos clusters diferentes, en un toroide que usa una configuración de 12x7. Estas 16 conexiones, se dan entre los cores del extremo inferior del cluster B

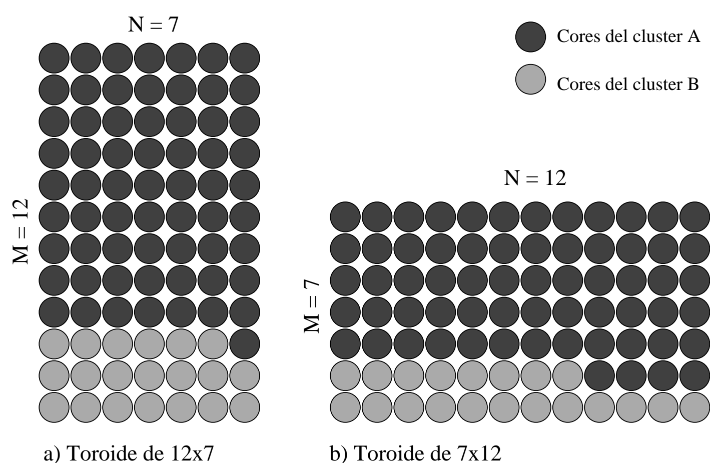

Figura 4.11: Configuraciones para un toroide sobre una VPN de 84 cores

y los cores del extremo superior del cluster A, y también entre los cores del extremo inferior del cluster A y los cores del extremo superior del cluster B. Por medio de estas 16 conexiones se envían y reciben mensajes a través de Internet, los mensajes principalmente son de tipo: información de carga, peticiones de carga y transferencia de carga.

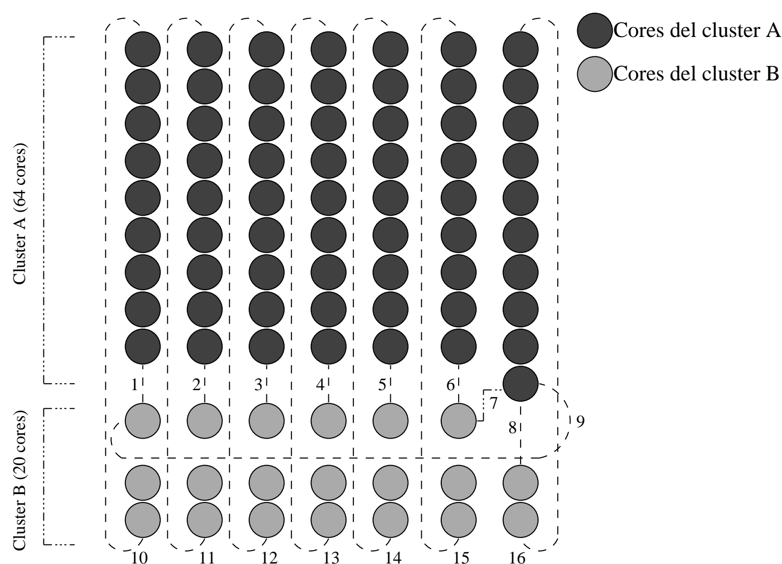

Figura 4.12: Conexiones lógicas entre cores de diferente cluster para un toroide de 12 x 7

Es de notarse que el costo de estas comunicaciones (por Internet), es mayor al de las comunicaciones locales (las que se dan entre cores de un mismo cluster), y el costo total

dependerá básicamente, de la cantidad de mensajes, de la velocidad y el estado de la banda ancha de los dos clusters (que puede estar saturada por el uso de otras aplicaciones o usuarios).

Por otra parte, si cambiamos la configuración del toroide a 7x12, como el de la Figura 4.13, se observa que el número de conexiones a través de Internet aumenta a 26. El aumento se debe, a que el número de cores del cluster A que tienen comunicación lógica con cores del cluster B se incrementa.

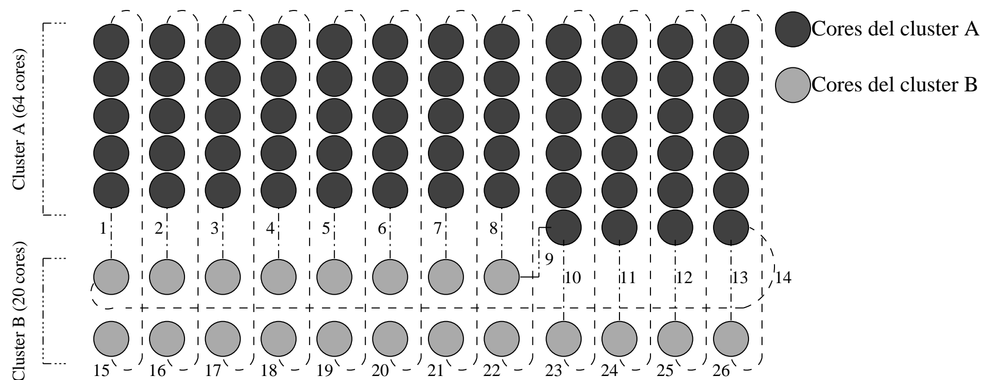

Figura 4.13: Conexiones lógicas entre cores de diferente cluster para un toroide de 7 x 12

El cambio en la configuración del toroide nos indica que mientras mayor sea el número de columnas (en el toroide), mayor número de comunicaciones entre cores de diferente cluster tendremos. Esto quiere decir, que si podemos elegir la configuración del toroide debemos escoger la que menor número de columnas tenga (de entre las configuraciones recomendadas).

Para el caso de DLML con subasta global tenemos un problema similar, sin embargo en esta versión se tiene un mayor número de conexiones a través de Internet, debido a que cada core descargado pide información de carga a todos los cores del sistema, algunos de ellos pertenecen a su mismo cluster y otros a un cluster diferente. Esto significa, que si tenemos un cluster A con 4 cores y un cluster B con 6 cores, en el momento en que un core del cluster A pide información de carga, se comunica con 3 cores de su cluster y con 6 cores del cluster B. De estas 9 comunicaciones, 6 serán con cores de diferente cluster, y como los 4 cores (del cluster A) pueden realizar las mismas conexiones, tenemos 24 conexiones por Internet (Figura 4.14). Lo mismo sucede con los cores del cluster B, en este cluster cada vez que un core se

quede sin carga, se comunicará con 5 cores de su propio cluster y con 4 cores del cluster A, dando un total de 24 conexiones por Internet para los cores del cluster B.

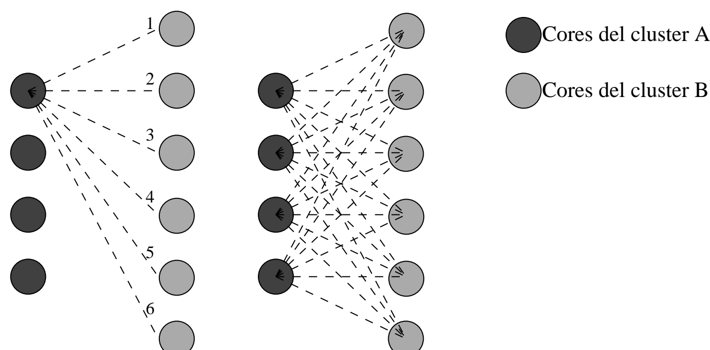

Figura 4.14: Conexiones por Internet para la versión de DLML con subasta global

Si se aumentara el número de cores como en el ejemplo de DLML con subasta parcial, en donde se tiene un cluster A con 64 cores y un cluster B con 20 cores, tendríamos 1280 conexiones por Internet, al ejecutar DLML con subasta global sobre los cores de la VPN (los cores de los dos clusters). No se debe olvidar que las 1280 conexiones entre cores de diferente cluster no son el número de mensajes que se harán a través de estas conexiones, este número varía según el comportamiento dinámico de la aplicación y afectan el costo de las comunicaciones.

Como podemos ver, las versiones anteriores de DLML (subasta global y parcial) presentan una cantidad variable de conexiones entre cores trabajadores de diferentes clusters, a diferencia de la versión con la nueva arquitectura en donde se limitan estas conexiones. El número de conexiones se limita (en la nueva arquitectura) al crear subgrupos de trabajo por cada cluster y no permitir que dos cores de diferente cluster se comuniquen entre sí, como ya se había mostrado en la Sección 4.3. Con esta restricción, se reduce el costo en las comunicaciones a través de Internet, ya que solamente los *Administradores Cluster* y el *Administrador Grid* intercambian mensajes por Internet.

Con la nueva arquitectura, se tienen dos tipos de conexiones entre cores de diferente cluster; por un lado tenemos las conexiones para el intercambio de información y peticiones, y por otro las conexiones para transferencia de carga. El número de conexiones para intercambio

de información y peticiones (entre el *Administrador Grid* y los *Administradores Cluster*) es igual al número de cluster menos 1 (sin importar el número de cores trabajadores que tenga cada cluster). Como se muestra en la Figura 4.15.a con un sistema de 4 clusters.

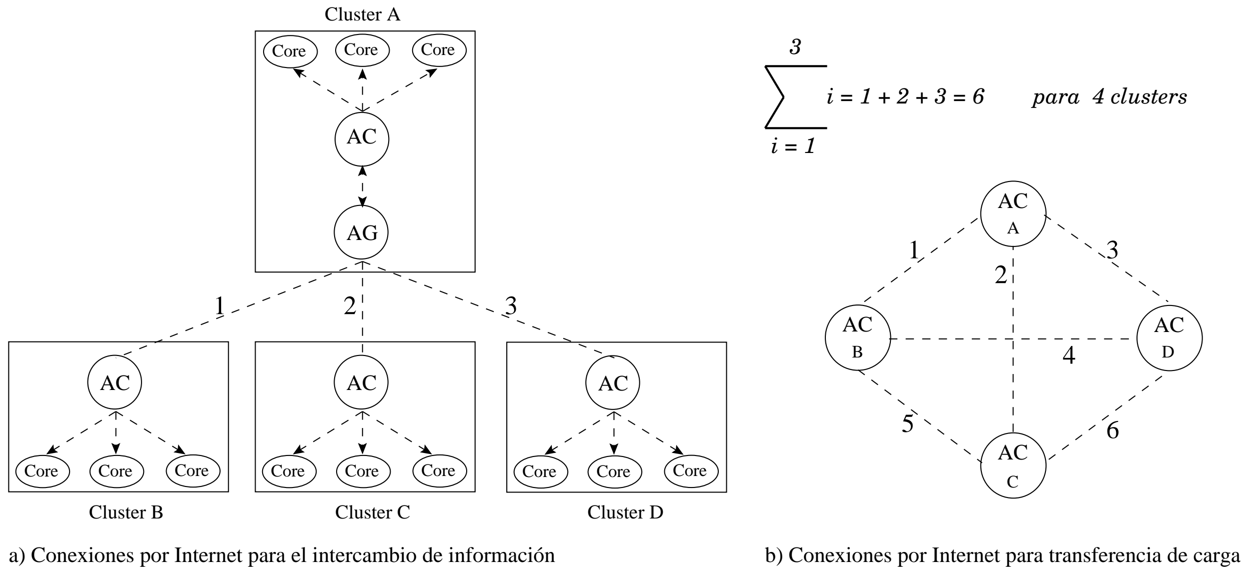

Figura 4.15: Conexiones por Internet para la nueva arquitectura

El número de conexiones para la transferencia de carga (entre los *Administradores Cluster*), se obtiene con la siguiente formula:

$$\# \quad conexiones \quad para \quad transferencia = \sum_{i=1}^{N-1} i \qquad para \quad N = \#clusters$$

Como se muestra en el ejemplo de la Figura 4.15.b (en donde se tienen 4 clusters), en este caso se tienen 6 conexiones para la transferencia de carga y 3 para el intercambio de información y peticiones, haciendo un total de 9 conexiones a través de Internet. Para un sistema de 2 clusters, donde un cluster A tiene 64 cores y otro cluster B tiene 20 cores (como el ejemplo de la Figura 4.13 y la Figura 4.14), tendríamos: 1 conexión para transferencia y otra para el intercambio de información.

En el siguiente capítulo presentamos la plataforma de experimentación y los resultados obtenidos en cuanto a tiempo de ejecución y balance de carga.

Capítulo 5

Resultados

En este capitulo presentamos los resultados obtenidos al comparar las versiones originales de DLML (con subasta global y subasta parcial ejecutadas sobre la VPN) contra sus dos nuevas versiones Glo-Grid y Toro-Grid, que se crearon a partir de la nueva arquitectura, en donde se favorece a las comunicaciones locales frente a las comunicaciones hechas a través de Internet.

5.1. Plataforma de experimentación

Para las pruebas se usaron dos cluster, uno ubicado en la UAM-I (delegación Iztapalapa) y otro en el CINVESTAV (delegación Gustavo A. Madero) de la Ciudad de México. Cada uno de los clusters tiene sus propias políticas de administración y de seguridad, además, cuentan con características heterogéneas en Software y Hardware. Los clusters utilizados (en donde se tiene un total de 150 cores, distribuidos en 37 nodos) para las pruebas son los siguientes:

1. Cluster Xcaret: 22 cores Intel®Core™2 Quad CPU 2.40GHz, con 4 GB en RAM por nodo, UAM-Iztapalapa. Cuenta con un switch Fast Ethernet.

2. Cluster Xena: 128 cores Intel®Core™i7 CPU 2.67GHz, con 4 GB en RAM por nodo, CINVESTAV Zacatenco. Cuenta con un switch Gigabit Ethernet.

En los dos clusters se instaló y configuró una VPN (con OpenVPN), con la configuración de la VPN se creó una interconexión promedio de 5 MB/Seg entre los dos clusters, que como

mencionamos están separados geográficamente. La conexión establecida para formar la VPN, la podemos observar en la Figura 5.1.

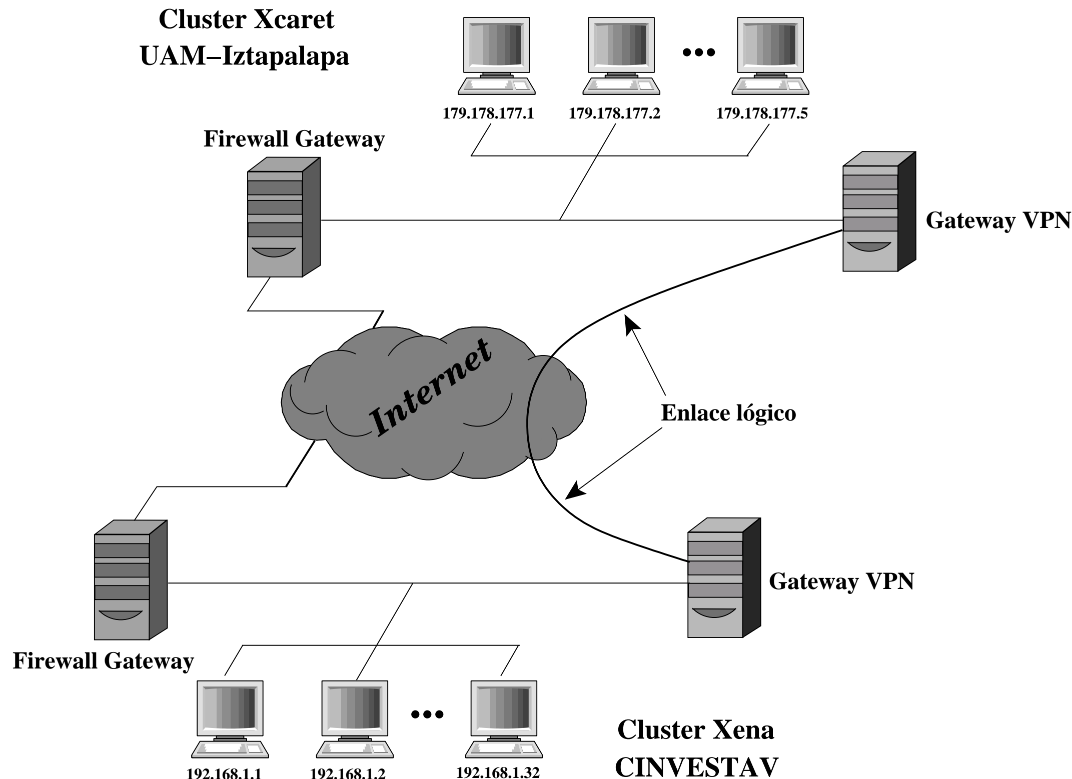

Figura 5.1: Interconexión de dos clusters usando una VPN

Versiones de DLML y aplicaciones para las pruebas

Sobre esta VPN se ejecutaron las pruebas, usando todas versiones de DLML, con la aplicación de N-Reinas y Multiplicación de Matrices. N-Reinas es una aplicación que resuelve el problema de colocar n reinas en un tablero de nxn, de tal forma que ninguna de ellas se ataque, el problema de las n reinas es analizado en [2]. Esta aplicación es una buena opción para evaluar la nueva arquitectura, debido principalmente, a que es una aplicación que genera una gran cantidad carga de forma dinámica, por lo cual se debe hacer un balance de carga a tiempo de ejecución. Por otra parte, la aplicación Multiplicación de Matrices (que hace los cálculos correspondiente a la multiplicación de matrices cuadradas y un procesamiento

de datos) es una buena opción para observar el comportamiento de la nueva arquitectura al ejecutar una aplicación estática.

Para las pruebas se compararon las dos versiones originales de DLML, contra las 2 nuevas versiones implementadas en este trabajo, en las cuales se usa la nueva arquitectura. Las cuatro versiones se listan a continuación:

1. Glo-VPN : Versión de DLML con subasta global.

2. Toro-VPN: Versión de DLML con subasta parcial.

3. Glo-Grid: Nueva versión de DLML, que usa la subasta global para llevar a cabo su balance *intra*-cluster y otra subasta global entre los *Administradores Clusters* (iniciada por el cluster receptor), para llevar a cabo el balance de carga *inter*-cluster.

4. Toro-Grid: Nueva versión de DLML, que usa la subasta parcial basada en la topología Toro, llevar a cabo su balance *intra*-cluster y la subasta global entre los *Administradores Clusters*, para llevar a cabo el balance de carga *inter*-cluster.

Las cuatro versiones se ejecutaron sobre 32, 64, 84, 116 y 148 cores para procesar la carga, con el objetivo de medir el tiempo de ejecución de N-Reinas (al usar un tablero de tamaño 16, 17 y 18) y de Multiplicación de Matrices (con matrices de 400x400 y 600x600). Cabe mencionar que para las nuevas versiones de DLML se usaron 2 cores más, ya que estos están dedicados a ejecutar a los *Administradores Cluster* y al *Administrador Grid*. Por otro lado, para las pruebas de distribución de carga, en este trabajo únicamente se presentan los resultados al usar 148 cores para procesar la carga de las dos aplicaciones, considerando todos los recursos disponibles en los dos clusters. Así mismo, el tamaño del tablero para N-Reinas fue de 17x17 y el orden de las matrices multiplicadas fue de 600x600, teniendo una generación de datos que requería el poder de cómputo de los dos clusters.

5.2. Resultados en el tiempo de ejecución de N-Reinas

Los resultados que se presentan a continuación, son resultado de comparar la versión de DLML con subasta global (Glo-VPN), contra la nueva versión Glo-Grid, al balancear carga en la aplicación N-Reinas. En la primera parte se ejecutó la aplicación N-Reinas (con un tablero de 16, 17 y 18) sobre 32 cores trabajadores, usando la versión Glo-VPN para realizar el balance. Posteriormente se realizaron las mismas pruebas pero ahora con 64, 84, 116 y 148 cores trabajadores. Cada una de las pruebas mencionadas, se repitieron 5 veces para obtener el promedio de los tiempos de ejecución de Glo-VPN. De la misma forma, se ejecutó Glo-Grid (en donde se usa el modelo jerárquico que favorece el balance local), los resultados obtenidos se presentan en la Figura 5.2.

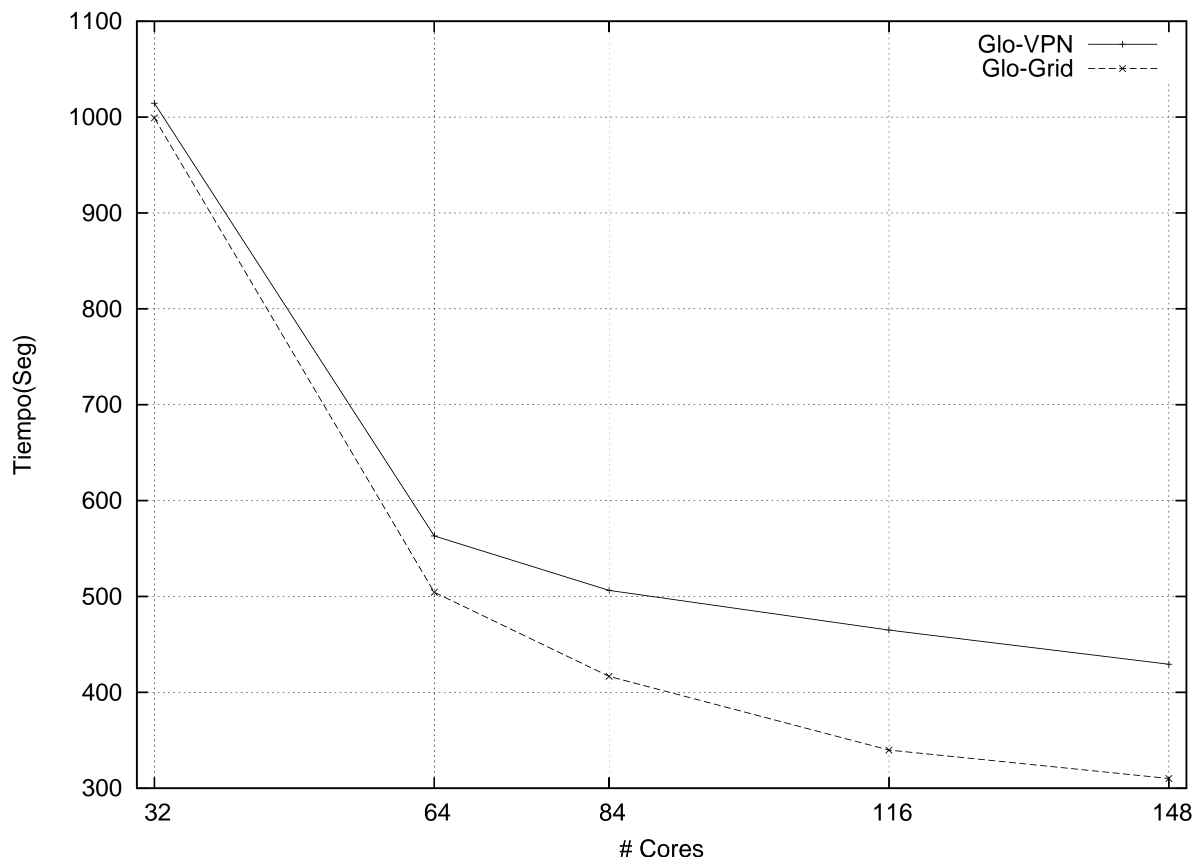

Figura 5.2: Tiempo de ejecución para N-Reinas con Glo-VPN y Glo-Grid

Como se puede observar, la versión Glo-Grid es más veloz que la versión Glo-VPN, esto se debe básicamente a que en la versión Glo-Grid, el número de mensajes a través de Internet, es mucho menor que en la versión original, Glo-VPN. Como se muestra en la sección 4.4, en la versión Glo-VPN existe un mayor número de conexiones entre cores de diferente cluster, y esto aumenta el número de mensajes a través de Internet. Mientras que en la versión Glo-Grid, el número de conexiones entre cores de diferente cluster se reduce a 2 (uno para peticiones y otro para transferencias), ya que sólo se usaron 2 clusters, y en uno de ellos están el *Administrador Grid* y el *Administrador Cluster*. Por lo tanto, solamente se tiene una conexión por Internet con el *Administrador Cluster* perteneciente al segundo cluster.

El número de conexiones entre cores de diferente cluster (conexiones por Internet) lo podemos ver en la Tabla 5.1, donde tenemos el número de cores trabajadores que se usaron en la VPN y como están agrupados en los dos clusters, al ejecutar Glo-VPN.

# Cores VPN	*# Cores Xcaret*	*# Cores Xena*	*Conex. Internet*
32	16	16	256
64	20	44	880
84	20	64	1280
116	20	96	1920
148	20	128	2560

Tabla 5.1: Número de conexiones entre cores de diferente cluster con Glo-VPN

En la tabla anterior, al usar 148 cores se tienen 2560 conexiones entre cores de diferente cluster, mientras que en la versión Glo-Grid en todas las configuraciones de la VPN, 32, 64, 84, 116 y 148 sólo se tiene una conexión a Internet. En base al número de las conexiones, se incrementa o reduce la cantidad de mensajes por Internet. Para ilustrar esto, en la Tabla 5.2 podemos ver la cantidad de estos mensajes, al ejecutar la aplicación N-Reinas con un tablero de tamaño 17, usando DLML en su versión Glo-VPN y Glo-Grid sobre 148 cores. En esta tabla se puede ver la diferencia que existe entre ambas versiones, en Glo-VPN se hacen 1,762,570 peticiones por Internet, mientras que en la versión Glo-Grid sólo se hacen 35

peticiones y en la transferencia de carga tenemos 9,444 para la versión Glo-VPN y 1,364 para Glo-Grid. Esta diferencia se debe, a que en Glo-Grid solamente 2 procesos intercambian carga e información, a través de Internet, mientras que en Glo-VPN los 148 procesos intercambian carga e información usando Internet. En consecuencia, obtenemos una reducción del 39 % en el tiempo de ejecución, a favor de Glo-Grid.

Versión de DLML	*# Peticiones*	*# Transferencias de carga*
Glo-VPN	1,762,570	9,444
Glo-Grid	35	1,364

Tabla 5.2: Mensajes a través de Internet generados por Glo-VPN y Glo-Grid

Para el caso de las versiones Toro-Grid y Toro-VPN, se presentan los resultados del tiempo de procesamiento en la Figura 5.3. En estas versiones, la diferencia de tiempos no es tan grande como en la versión global, esto se debe a que en la versión Toro-VPN se está usando el algoritmo de subasta parcial, en el cual, cada core de la VPN solamente se puede comunicar con 4 cores vecinos. Al sólo poderse comunicar con 4 cores vecinos se limita el número de conexiones a través de Internet (Tabla 5.3), a diferencia de la versión global en donde cada core se puede comunicar con cualquier core de la VPN (como se mostró en la Sección 4.4).

# Cores VPN	*# Cores Xcaret*	*# Cores Xena*	*Config. toroide*	*Conex. Internet*
32	16	16	8x4	8
64	20	44	8x8	18
84	20	64	12x7	16
116	20	96	29x4	8
148	20	128	37x4	8

Tabla 5.3: Conexiones entre cores de diferente cluster con Toro-VPN

De la misma forma que en la versión Glo-VPN, presentamos en la Tabla 5.3 el número de

conexiones por Internet que existe para cada una de las diferentes configuraciones del toroide, en la versión Toro-VPN. Como se recordará, en esta versión el toroide puede adoptar varias configuraciones, de entre las cuales, para estas pruebas se escogieron las que se muestran en la columna *Config. toroide* de la tabla.

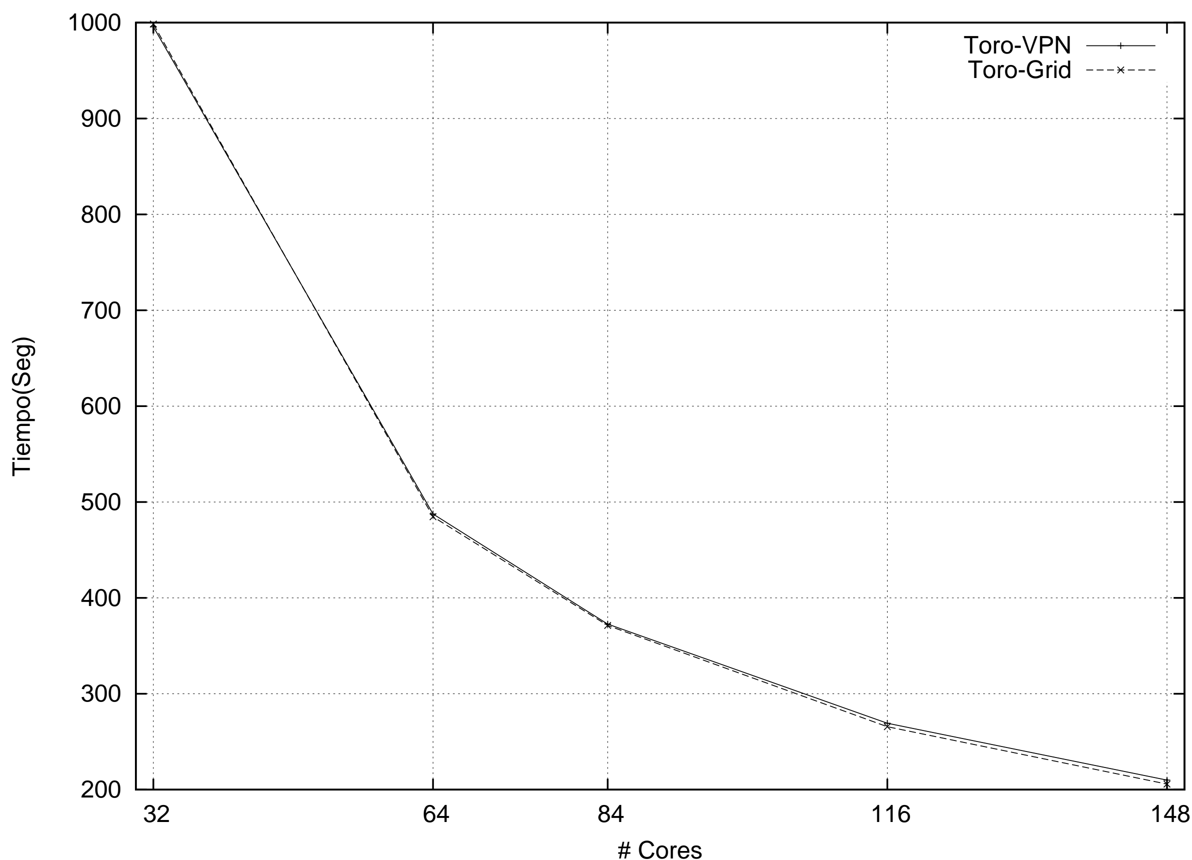

Figura 5.3: Tiempo de ejecución para N-Reinas con Toro-VPN y Toro-Grid

Como se puede observar, el número (de conexiones a través de Internet) es considerablemente menor que en la versión Glo-VPN, teniendo un máximo de 18 conexiones por Internet para Toro-VPN (frente a las 2560 de la versión Glo-VPN). Mientras que en la versión Toro-Grid el número de conexiones es nuevamente de 2, como en el caso de Glo-Grid. En la Tabla 5.4, podemos ver el número de mensajes que se generan en las dos versiones (Toro-VPN y Toro-Grid) al ejecutar N-Reinas con un tablero de 17, usando 148 cores trabajadores.

Versión de DLML	*# Peticiones*	*# Transferencias de carga*
Toro-VPN	20,180	11,069
Toro-Grid	23	1,395

Tabla 5.4: Mensajes a través de Internet generados por Toro-VPN y Toro-Grid

En esta tabla se observa que la diferencia entre el número de peticiones, a pesar de ser grande 20180 y 23, no se compara a la diferencia mostrada en las versiones Glo-VPN y Glo-Grid (donde se tenía 1,762,570 y 35 respectivamente). Estos resultados muestran que las comunicaciones más costosas, son las destinadas a buscar carga en el sistema, a diferencia de las que se usan para transferir carga de un core a otro (para el caso de las versiones Glo-VPN y Toro-VPN) y de un cluster a otro en las versiones Glo-Grid y Toro-Grid. Comparando las peticiones y transferencias de carga en las 4 versiones, podemos entender mejor esta afirmación, Tabla 5.5.

Versión de DLML	*# Peticiones*	*# Transferencias de carga*
Glo-VPN	1,762,570	9,444
Glo-Grid	35	1,364
Toro-VPN	20,180	11,069
Toro-Grid	23	1,395

Tabla 5.5: Mensajes a través de Internet generados por las 4 versiones de DLML

Se puede observar que en ambos pares de pruebas, (Glo-VPN, Glo-Grid) y (Toro-VPN, Toro-Grid), la diferencia entre las transferencias de carga es similar, por un lado tenemos 9,444 y 1,364 para la Glo-VPN y Glo-Grid respectivamente, y para las versiones Toro-VPN y Toro-Grid tenemos 11,069 y 1,395. Sin embargo, en los dos pares de pruebas (Glo-VPN, Glo-Grid) y (Toro-VPN, Toro-Grid), se tiene una diferencia importante en cuanto a los tiempos de ejecución. Para (Glo-VPN, Glo-Grid) la reducción (en el tiempo de procesamiento) más grande que se obtuvo en las pruebas es de 27.78 % (proporcionado por Glo-Grid), mientras

que para (Toro-VPN, Toro-Grid) sólo se tiene una reducción del 1.53 % (proporcionado por Toro-Grid).

Al comparar el tiempo de ejecución de las 4 versiones de DLML (Figura 5.4) y la cantidad de mensajes por Internet, podemos concluir que la diferencia entre ambos pares de pruebas, está en el número de peticiones por Internet, la eficiencia de los algoritmos usados localmente en cada uno de los clusters (en las versiones Glo-Grid y Toro-Grid) y como se verá más adelante, por la distribución de carga asociada a cada algoritmo de subasta.

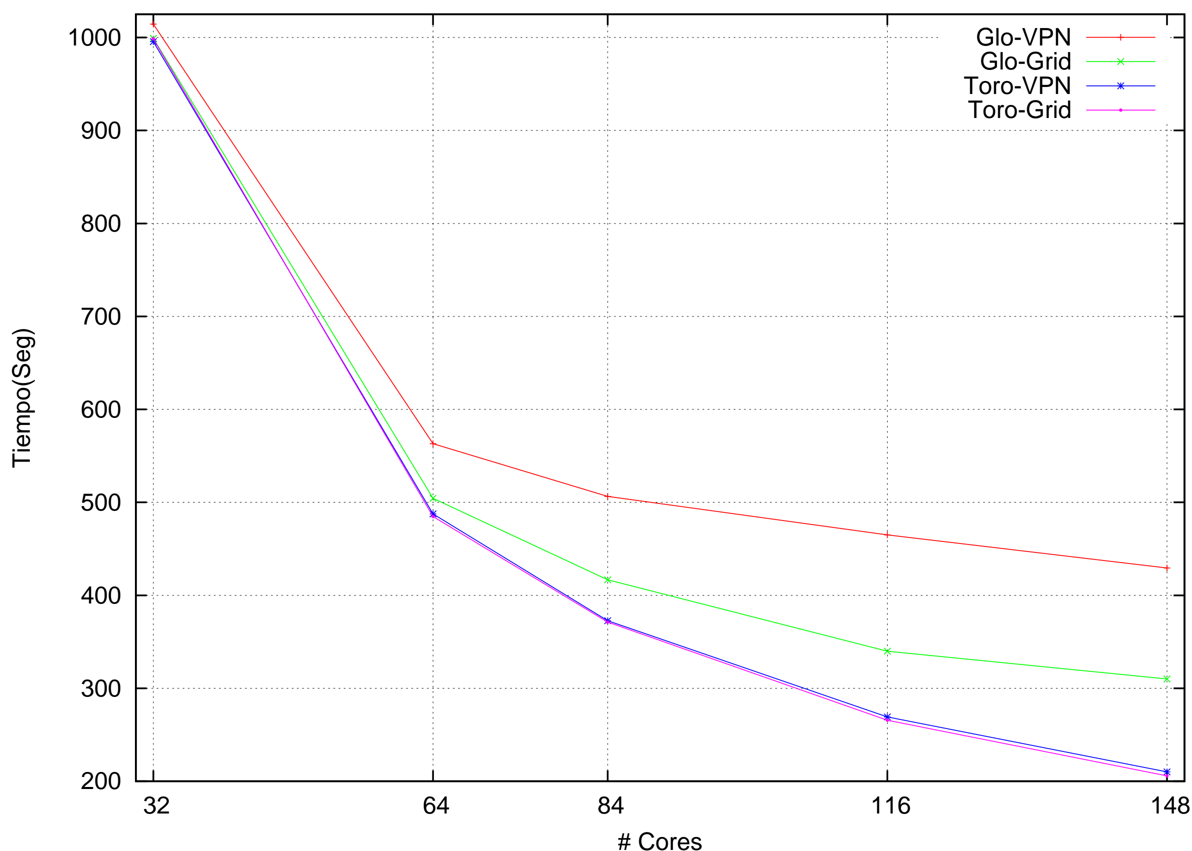

Figura 5.4: Tiempo de ejecución para N-Reinas usando las 4 versiones

Con los resultados obtenidos se hizo una comparación (de los tiempos de procesamiento) ejecutando DLML (en la versión global y parcial) sobre un solo cluster. Con el propósito de verificar que efectivamente hay una disminución en el tiempo de procesamiento, al ejecutar DLML sobre dos clusters. Los resultados de la subasta global los podemos ver en la Tabla

5.6, en donde se observa que el problema de la escalabilidad de la subasta global se hace presente sobre los dos clusters a partir de los 84 cores VPN.

#Cores 1Cluster	*Global(Seg)*	*#Cores 2Cluster*	*Glo-VPN(Seg)*	*Glo-Grid(Seg)*
16=42.72Ghz	1818.83	32=81.12Ghz	1014.43	999.00
44=117.48Ghz	684.16	64=165.48Ghz	563.04	504.32
64=170.88Ghz	497.91	84=218.88Ghz	506.38	416.61
96=256.32Ghz	346.74	116=304.32Ghz	464.96	339.78
128=341.76Ghz	313.52	148=389.76Ghz	429.37	310.05

Tabla 5.6: Tiempo de ejecución de la subasta Global en 1 y 2 clusters ejecutando N-reinas con un tablero 16, 17 y 18

Lo anterior nos indica, que a pesar de contar con 20 cores menos la subasta global (sobre un cluster con 64 cores) vence en tiempo a Glo-VPN (sobre 2 clusters con 84 cores). Si se grafican los datos de la Tabla 5.6 en la misma gráfica (a pesar de que en Global se usen menos cores), se pueden observar los tiempos de ejecución de N-Reinas con Global, Glo-VPN y Glo-Grid, al usar uno y dos cores (Figura 5.5).

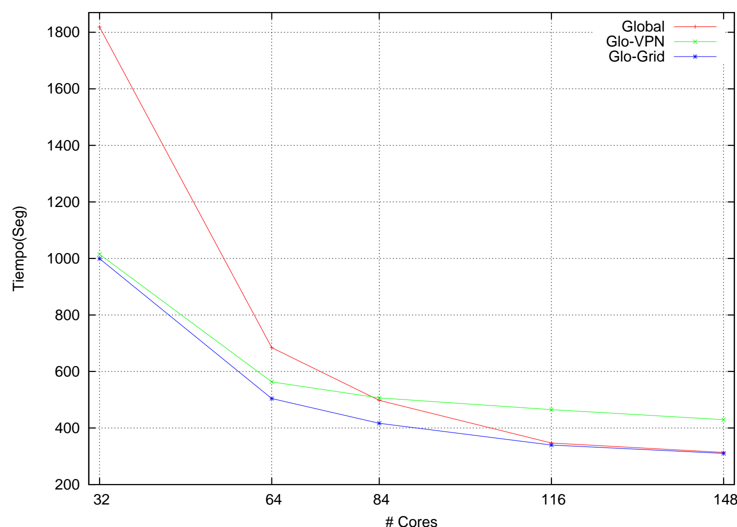

Figura 5.5: Tiempo de ejecución de N-Reinas con subasta global en 1 y 2 clusters

En estos resultados podemos ver que en Glo-VPN se presenta el problema de la escalabilidad, en donde el costo de las comunicaciones es alto, debido a que cada core se comunica con todos los cores del sistema en cada búsqueda de carga (por lo cual mientras más cores, mayor costo en comunicaciones), aunado a esto, en Glo-VPN muchas de estas comunicaciones son a través de Internet. En Global también se presenta el problema de la escalabilidad, sin embargo, no se hacen comunicaciones a través de Internet, lo que representa una ventaja frente a Glo-VPN. Por otra parte, tenemos a Glo-Grid (2 clusters) que a pesar de vencer en tiempo a la versión Global (en un Cluster), muestra pocas mejoras conforme aumenta la diferencia (en capacidad de procesamiento) entre los clusters. Esto indica, que se llegará a un punto (como se verá en la Multiplicación de Matrices) en que esta diferencia represente un problema, y en lugar de obtener beneficios se obtengan retrasos. Este comportamiento también se presenta en la subasta parcial, el cual podemos observar al comparar el tiempo de procesamiento de uno y dos clusters, del mismo modo que en la subasta global (Figura 5.6). Aunque en estas versiones, Toroide (ejecutado en un sólo cluster) no es capaz de vencer a Toro-VPN o a Toro-Grid que se ejecutan en dos clusters, debido a que en la subasta parcial el problema de la escalabilidad es menor que en la global [8], y por consiguiente, con Toro-VPN y Toro-Grid se obtienen mejores resultados al ejecutar la aplicación sobre dos clusters.

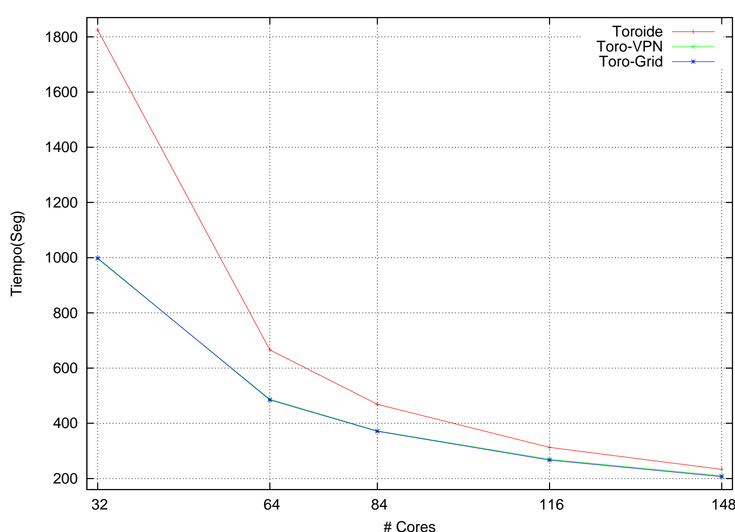

Figura 5.6: Tiempo de ejecución de N-Reinas con subasta parcial en 1 y 2 clusters

5.3. Resultados en la distribución de carga de N-Reinas

El objetivo principal de la nueva versión DLML es el balance de carga a tiempo de ejecución, a nivel cluster y a nivel Grid (balance *inter*-cluster). Como se mencionó, la principal función del balance de carga es distribuir de forma equitativa la carga total del sistema entre todos los cores de la VPN, esto quiere decir, que los cores deben procesar la misma cantidad de carga. Esto es prácticamente imposible en una aplicación que genera carga a tiempo de ejecución, sin embargo, con DLML podremos aspirar a una distribución equitativa en todos o la mayoría de los cores de la VPN, ya que unos cores podrían ser más poderosos que otros.

Como se mencionó en la Sección 5.1, para las pruebas realizadas se usaron dos clusters ubicados en la UAM-Iztapalapa y en el CINVESTAV, el de la UAM es de 20 cores a 2.40GHz para procesar carga y el del CINVESTAV es de 128 cores a 2.67GHz. Como se puede apreciar la capacidad de cómputo del cluster del CINVESTAV es más grande, no sólo en el número, si no en la capacidad de procesamiento de sus cores.

De la misma forma que en las pruebas de tiempo de procesamiento, en las pruebas de distribución de carga se tomaron promedios de distribución de carga en los 148 cores de la VPN, usando la aplicación de las N-Reinas con un tablero de tamaño 17 (como se muestra en la Figura 5.7), para la versión Glo-VPN y Glo-Grid.

En la gráfica de la Figura 5.7 lo primero a analizar es la diferencia que se puede observar entre los dos clusters, los primeros 20 cores pertenecen al cluster Xcaret de la UAM-Iztapalapa y los 128 restantes a Xena del CINVESTAV. En los cores de Xcaret se puede apreciar una mejor distribución en la versión Glo-Grid que en la versión Glo-VPN. Esto lo podemos constatar con la desviación estándar de estas dos versiones (Tabla 5.7).

Versión	*Desviación estándar*
Glo-VPN	19,106,986.06
Glo-Grid	14,329,966.75

Tabla 5.7: Desviación estándar de la carga procesada en Glo-VPN y Glo-Grid

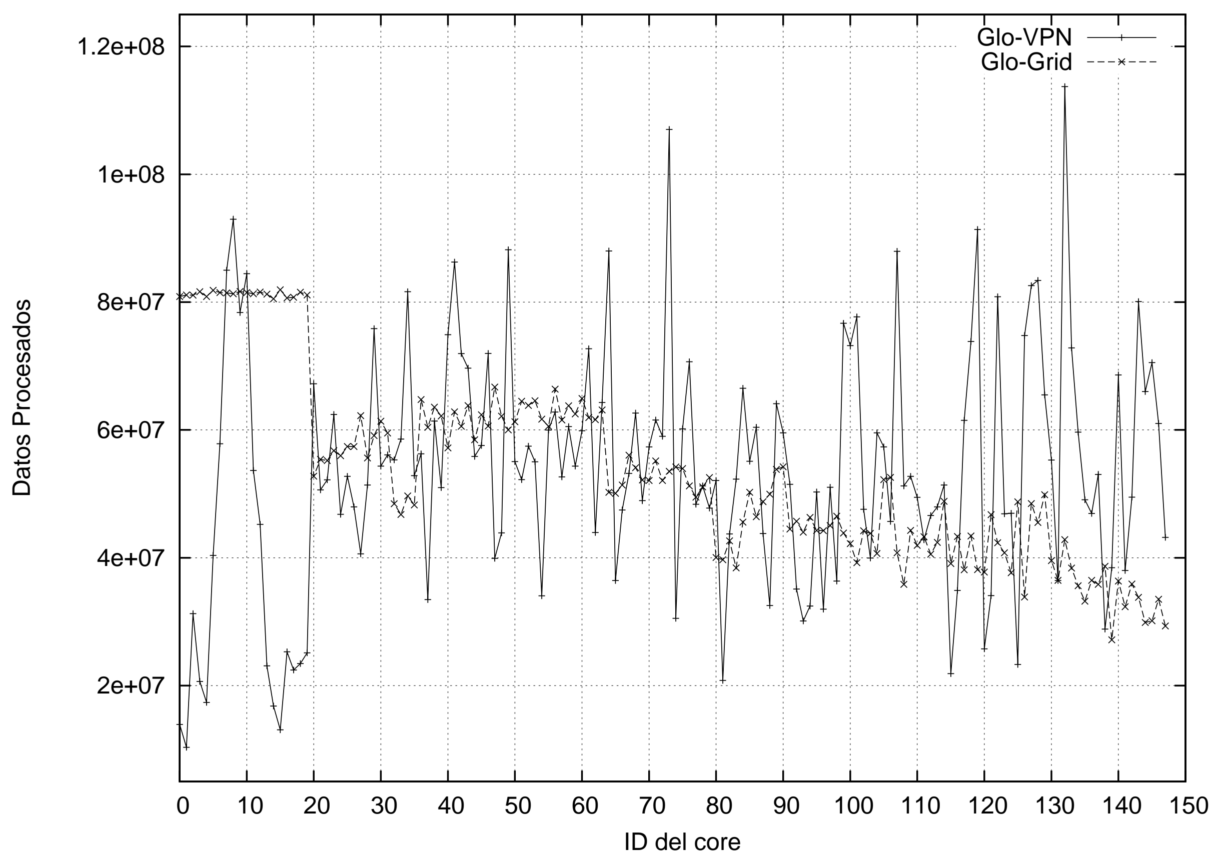

Figura 5.7: Distribución de carga para N-Reinas con Glo-VPN y Glo-Grid

Como podemos ver Glo-Grid distribuye mejor la carga al generar un grupo de trabajo por cada cluster e impedir que cores de diferente cluster se comuniquen. Además en esta versión el balance de carga *intra*-cluster se ve favorecido, ya que en Glo-Grid se usa el algoritmo de subasta global para el balance de carga *intra*-cluster, el cual muestra mayor efectividad que el algoritmo con subasta parcial hasta los 32 cores [8].

La distribución en el cluster de Xena es desigual aún para la versión Glo-Grid, ya que se tienen 128 cores, y como mencionamos la subasta global muestra buenos resultados hasta con 32 cores. A pesar de esto sigue siendo mejor que en la versión Glo-VPN, que tiene una desviación estándar que supera a la de Glo-Grid por más de 5,000,000 de datos (consecuencia directa del balance de carga en los 148 cores de la VPN).

Una vez que se ha analizado la distribución de carga de los cores, ahora vamos a analizar

la distribución de carga de un cluster a otro, en donde entenderemos por qué en la gráfica de Glo-Grid (en la Figura 5.7), se tiene una variación notable en la cantidad de carga que procesan los cores de Xcaret (los primeros 20 cores), con respecto a la cantidad de carga que procesan los cores de Xena (los últimos 128 cores). En la Figura 5.8, tenemos el porcentaje de carga que procesa cada cluster.

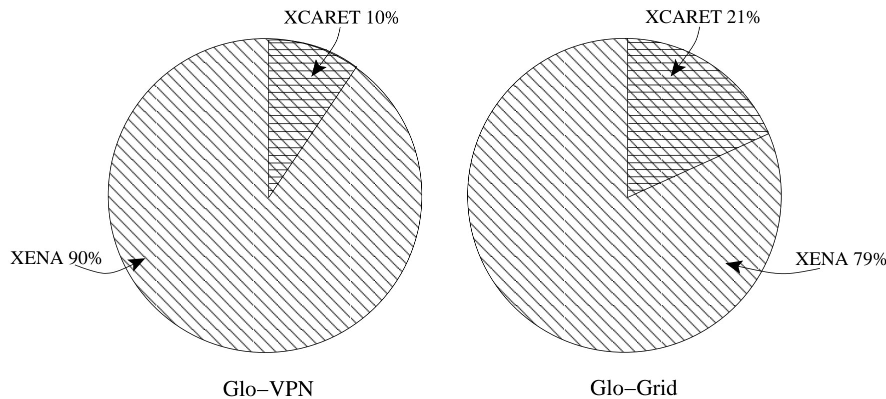

Figura 5.8: Porcentaje de distribución de carga en los dos clusters

En la Figura 5.8 se puede apreciar la diferencia que existe en las capacidades de cómputo de los clusters, Xcaret sólo cuenta con 20 cores a 2.4 GHz, mientras que Xena posee 128 cores a 2.67 GHz, por lo cual Xena hace la mayor parte del procesamiento (como era de esperarse). En los 20 cores de Xcaret se procesa el 21 % de los datos, (de un total de 7,921,206,828 datos procesados), este 21 % es igual a 1,625,426,848 datos procesados por el cluster Xcaret. Tomando en cuenta que Xcaret posee 20 cores, cada core procesa 81,271,342 $\pm$ 14,329,966 datos, mientras que para los 128 cores de Xena se tiene una carga de 6,295,779,980 (el 79 % del total), cada core procesar 49,185,781 $\pm$ 19,106,986 datos. Estos valores se ajustan a los que se presenta en la gráfica de Glo-Grid en la Figura 5.7.

En cuanto a las versiones Toro-VPN y Toro-Grid tenemos una distribución de carga más equitativa (Figura 5.9), excepto en los cores 80, 81, 82 y 83 que pertenecen a un nodo del cluster Xena, el cuál presenta menor capacidad de procesamiento que los demás nodos de Xena. En los resultados obtenidos debemos tomar en cuenta que en las dos versiones mencionadas se usa el algoritmo con subasta parcial, en Toro-Grid para el balance *intra-*

cluster y en Toro-VPN para el balance en todos los cores de la VPN. Como ya se mencionó la subasta parcial demuestra mejores resultados para un número mayor de 32 cores, al disminuir el problema de la escalabilidad limitando las comunicaciones a 4 vecinos por core, por esta razón, estas versiones muestran mejores resultados en la distribución de carga (entre los cores de la VPN), que la mejor versión hasta ahora, Glo-Grid.

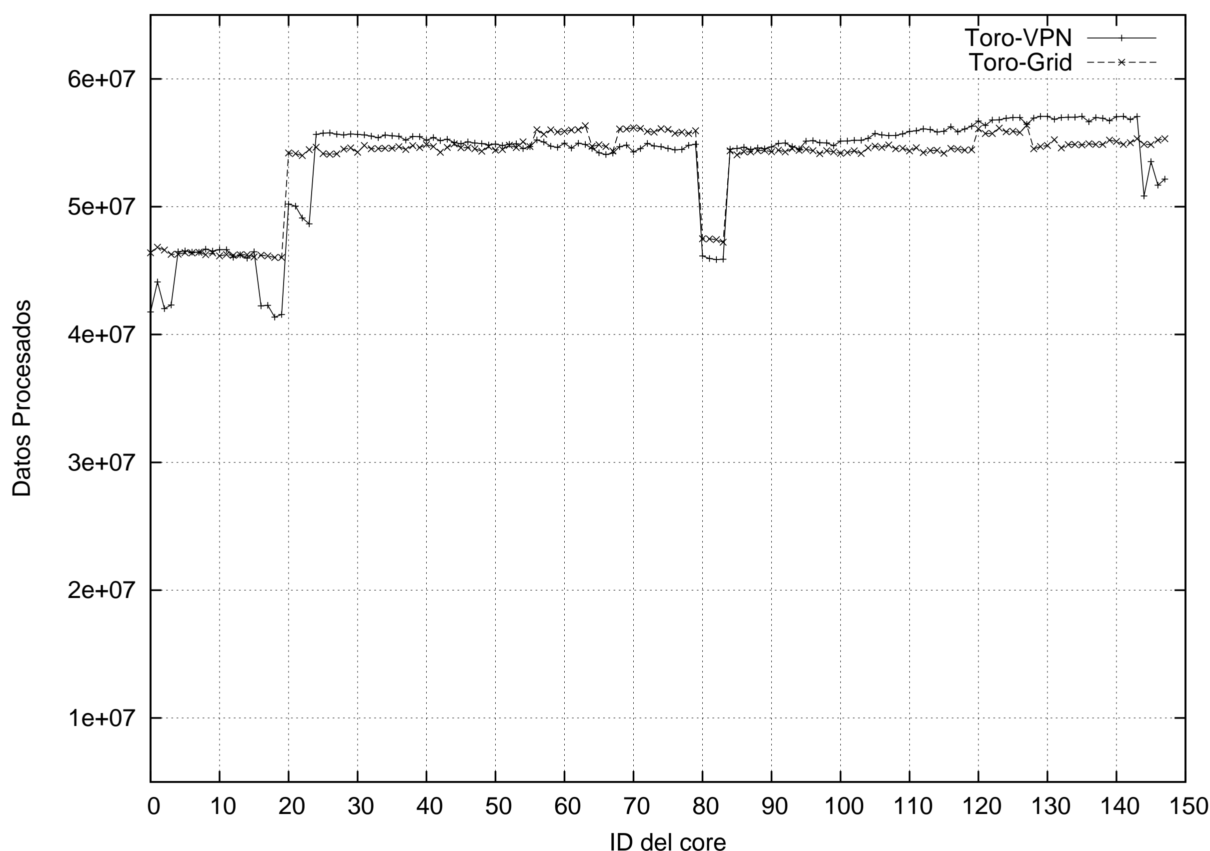

Figura 5.9: Distribución de carga para N-Reinas con Toro-VPN y Toro-Grid

A pesar de las mejoras presentadas por Toro-VPN, Toro-Grid muestra una distribución de carga más equitativa sobres los cores de la VPN, como se puede ver en la Tabla 5.8 en donde se muestra la desviación estándar de estas dos versiones (en la cual tienen una diferencia de 927,191 datos a favor de Toro-Grid). Esto se debe fundamentalmente a que se hacen grupos de trabajo para cada cluster, al igual que en la versión Glo-Grid.

De la misma manera que en las versiones de Glo-VPN y Glo-Grid tenemos una diferencia

Versión	*Desviación estándar*
Toro-VPN	4,087,513
Toro-Grid	3,160,321

Tabla 5.8: Desviación estándar de la carga procesada en Toro-VPN y Toro-Grid

entre la cantidad de carga que procesan los cores de Xcaret y los cores de Xena en estas dos versiones (Toro-VPN y Toro-Grid). Esta diferencia se explica por la cantidad de carga que procesa cada cluster como se puede ver en la Figura 5.10. En Toro-VPN tenemos un 11 % para Xcaret y 89 % para Xena, mientras que para Toro-Grid tenemos un 12 % para Xcaret y un 88 % para Xena (de un total de 7,921,206,828 datos procesados en ambas versiones).

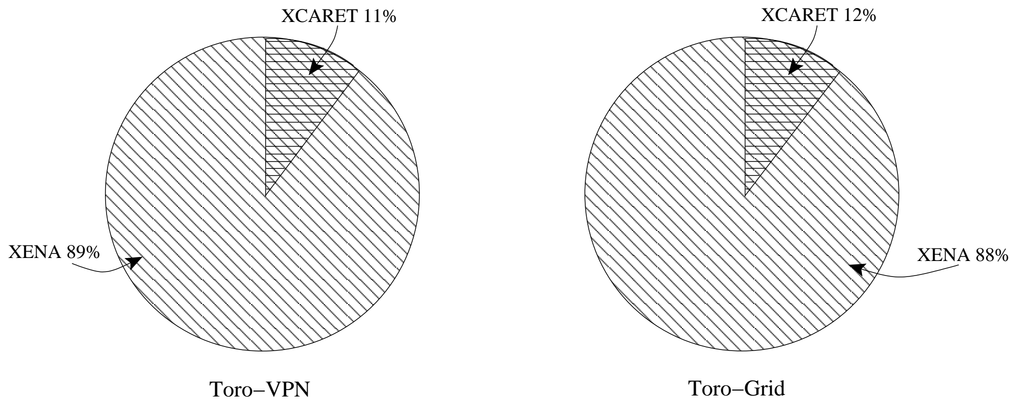

Figura 5.10: Porcentaje de distribución de carga en los dos clusters

Esto indica que los 20 cores de Xcaret en Toro-VPN deben procesar 44,737,713 ± 4,087,513 datos cada uno, y en Toro-Grid cada core de Xcaret procesa 46,289,496 ± 3,160,321 datos, justamente lo que se muestra en la Figura 5.9. Por otra parte, los 128 cores de Xena muestran que en la versión Toro-VPN cada core debe procesar 54,894,160 ± 4,087,513 datos, y en Toro-Grid cada core de Xena procesa 54,651,694 ± 3,160,321, una vez más como lo indican los datos de la Figura 5.9.

En base a las pruebas realizadas en las 4 versiones, podemos afirmar que las dos nuevas versiones de DLML Glo-Grid y Toro-Grid hacen una mejor distribución de carga entre los

cores de la VPN, debido principalmente a que en estas versiones se usa el modelo jerárquico para el balance de carga, en donde se crean grupos de trabajo por cada cluster, y con esto se favorece el balance de carga *intra*-cluster. Esta característica permite una mejor distribución de carga entre los cores de cada cluster, traduciéndose en una disminución en el tiempo de ejecución global de la aplicación. En la Figura 5.11 podemos ver la distribución de carga de las 4 versiones presentadas.

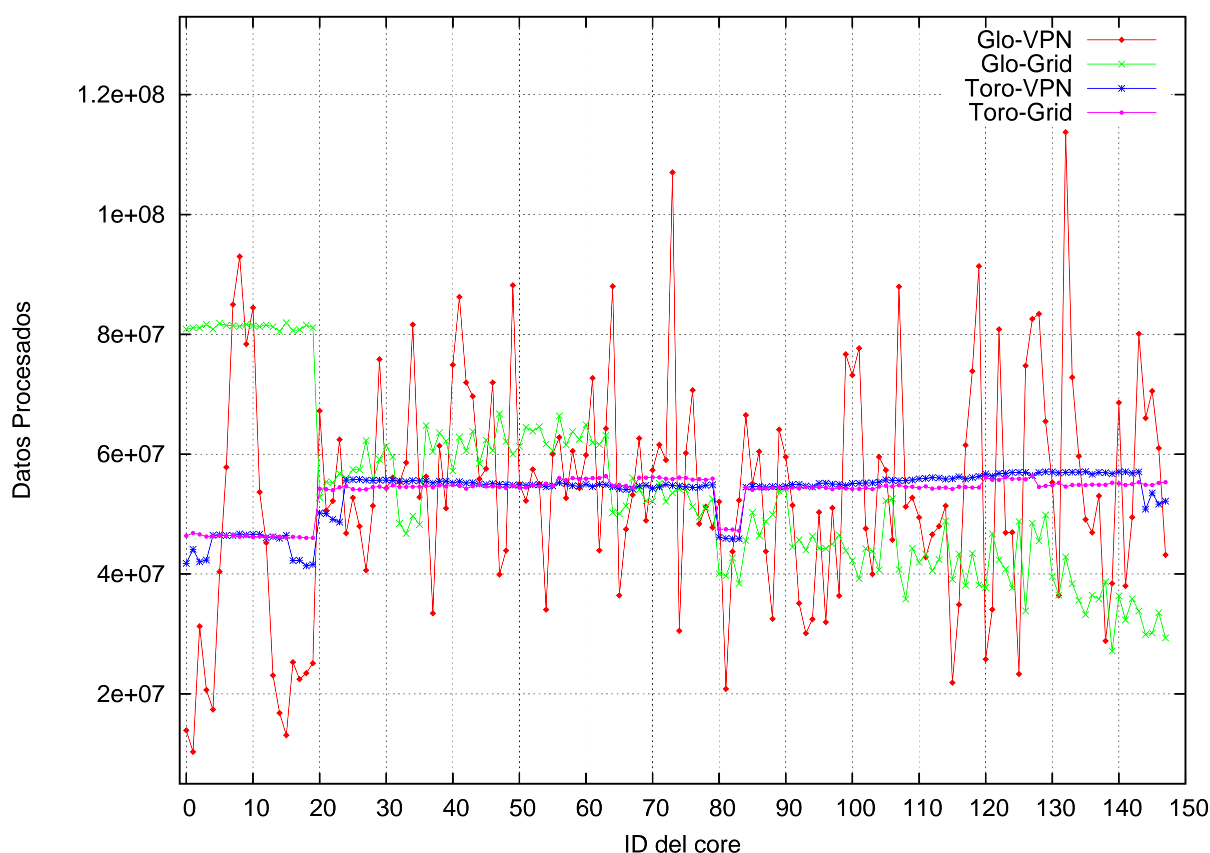

Figura 5.11: Distribución de carga para N-Reinas usando las 4 versiones

En la siguiente Sección, se presentan los resultados obtenidos al ejecutar la aplicación Multiplicación de Matrices, sobre la misma cantidad de cores trabajadores que se usaron en las pruebas de N-Reinas.

5.4. Resultados en el tiempo de ejecución de Multiplicación de Matrices

En estos resultados se debe tomar en cuenta que la Multiplicación de Matrices es una aplicación estática, y en una aplicación estática se conoce desde el inicio del procesamiento la carga total que se debe procesar (a diferencia de N-Reinas, que al ser una aplicación dinámica genera nuevos datos que procesar a tiempo de ejecución). Ya que se conoce la cantidad de carga total, se divide y distribuye entre todos los cores trabajadores desde el inicio del procesamiento, esto garantiza que cada core recibirá la misma cantidad de carga inicialmente.

En la Multiplicación de Matrices también se debe tomar en cuenta que las transferencias de carga son más pesadas que las que se hacen en N-Reinas, esto quiere decir, que la cantidad de datos (del orden de Mega-byte) que se transfieren de un cluster a otro en N-Reinas no supera los 4 MB, a diferencia de Multiplicación de Matrices en donde se tienen transferencias iniciales que van de los 200 a los 700 MB aproximadamente. Esto se debe a que un elemento de la lista en N-Reinas es de 96 bytes, mientras que un elemento de la lista de Multiplicación de Matrices es de 2424 bytes. Lo anterior se traduce en un retraso en el tiempo de procesamiento global para las 4 versiones de DLML, ya que se deben hacer transferencias de carga de un cluster a otro a través de Internet. Por esta razón, en ocasiones especificas (aunque se aumente la cantidad de cores en la VPN) la velocidad de procesamiento al ejecutar la aplicación sobre los cores de la VPN, será mayor, a la velocidad de procesamiento obtenida en un sólo cluster.

Las pruebas para la aplicación de Multiplicación de Matrices se hicieron usando las 4 versiones de DLML (Glo-VPN, Glo-Grid, Toro-VPN y Toro-Grid) sobre 32, 64, 84, 116 y 148 cores trabajadores. Del mismo modo que en N-Reinas, se obtuvieron promedios para evaluar el tiempo de procesamiento de las 4 versiones. El procesamiento en esta aplicación consiste en multiplicar matrices cuadradas de 400x400 y de 600x600, y en realizar un procesamiento de datos por cada valor de la matriz resultante. Los resultados correspondientes a la versión Glo-VPN y Glo-Grid los podemos ver en la Figura 5.12.

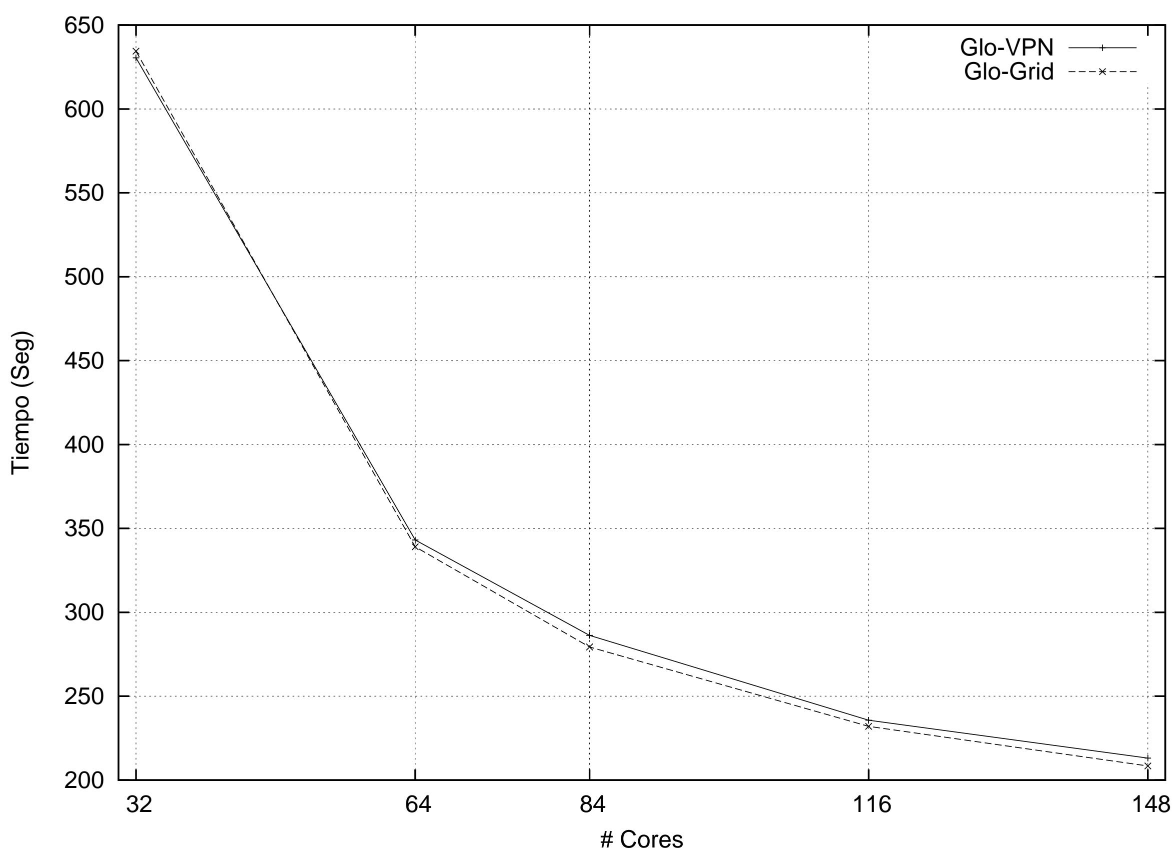

Figura 5.12: Tiempo de ejecución para Multiplicación de Matrices con Glo-VPN y Glo-Grid

Como podemos apreciar Glo-VPN es más veloz que Glo-Grid con 32 cores, esto se debe a que el número de peticiones para conseguir carga en un ambiente de 32 cores, no representa un problema e incluso la versión global puede vencer en tiempo de ejecución a la versión parcial con el algoritmo del toroide [8] al usar 32 cores. Esto no se presentó en N-Reinas, ya que en N-Reinas el número de peticiones era mucho mayor debido a que se procesó una mayor cantidad de datos (que los que procesa la Multiplicación de Matrices) y por consiguiente el número de peticiones para conseguir carga es mucho mayor. Para ilustrar esto, en la Tabla 5.9 se presentan los mensajes a través de Internet al ejecutar Multiplicación de Matrices, para procesar 360000 datos (al multiplicar matrices de 600x600) sobre 148 cores trabajadores usando Glo-VPN y Glo-Grid.

Con los datos de la tabla, podemos ver que la cantidad de peticiones en Glo-VPN tiene

Versión de DLML	*# Peticiones*	*# Transferencias de carga*
Glo-VPN	120,901	328,013
Glo-Grid	7	299,136

Tabla 5.9: Mensajes a través de Internet generados por Glo-VPN y Glo-Grid

una diferencia importante con respecto a Glo-Grid, sin embargo no es tan grande como en N-Reinas, aunque se debe tomar en cuenta que en N-Reinas se procesan 7,921,206,828 datos (para un tablero de tamaño 17), mientras que en Multiplicación de Matrices sólo se procesaron 360000 datos. Esto también se presenta en las versiones Toro-VPN y Toro-Grid (Figura 5.13), como se puede apreciar en los datos de la Tabla 5.10.

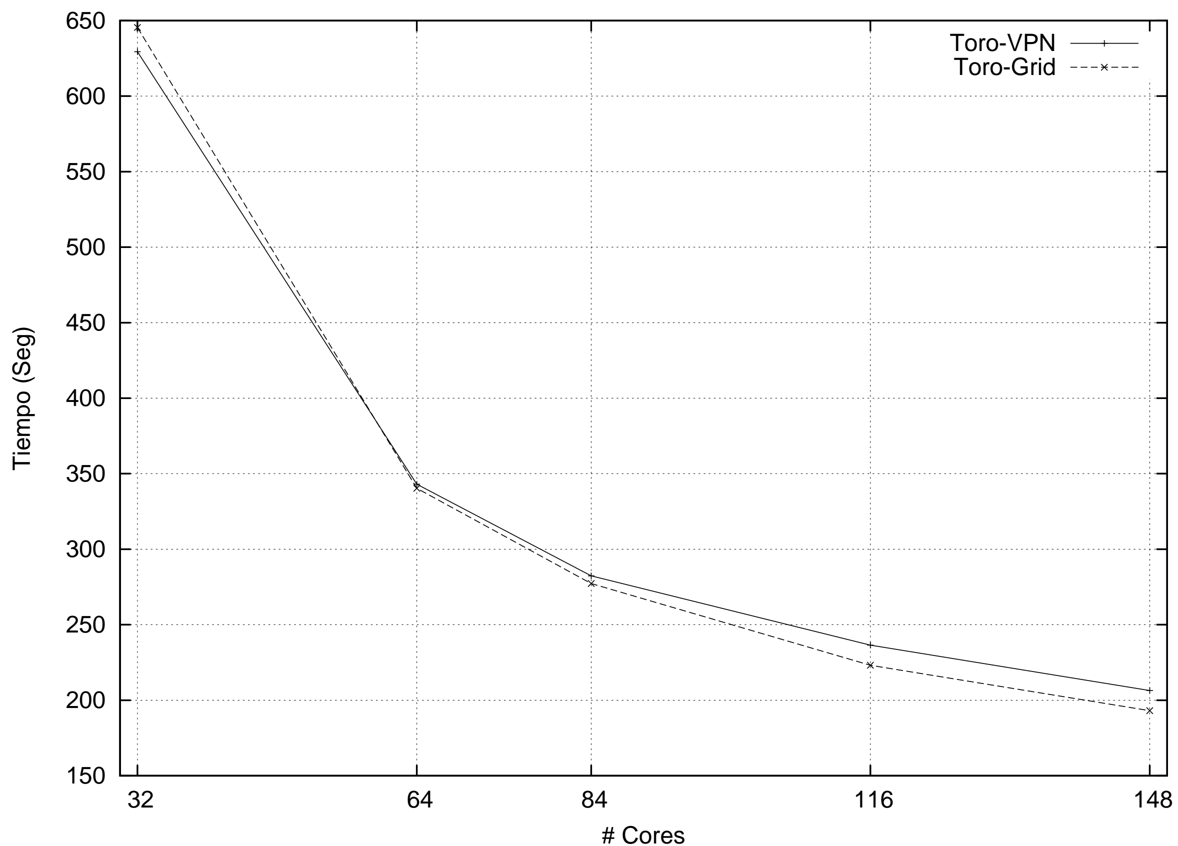

Figura 5.13: Tiempo de ejecución para Multiplicación de Matrices con Toro-VPN y Toro-Grid

Versión de DLML	*# Peticiones*	*# Transferencias de carga*
Toro-VPN	11,436	319,610
Toro-Grid	9	297,388

Tabla 5.10: Mensajes a través de Internet generados por Toro-VPN y Toro-Grid

En el caso de Toro-VPN y Toro-Grid tenemos aún menos cantidad de mensajes a través de Internet, aunado ala eficiencia de los algoritmos (en donde el algoritmo global muestra mejores resultados que el parcial con 32 cores), nos ayuda a entender porque la diferencia entre Toro-VPN y Toro-Grid es mayor que en Glo-VPN y Glo-Grid al usar 32 cores, como se puede apreciar en la Figura 5.14, en donde se presentan los tiempos de las 4 versiones.

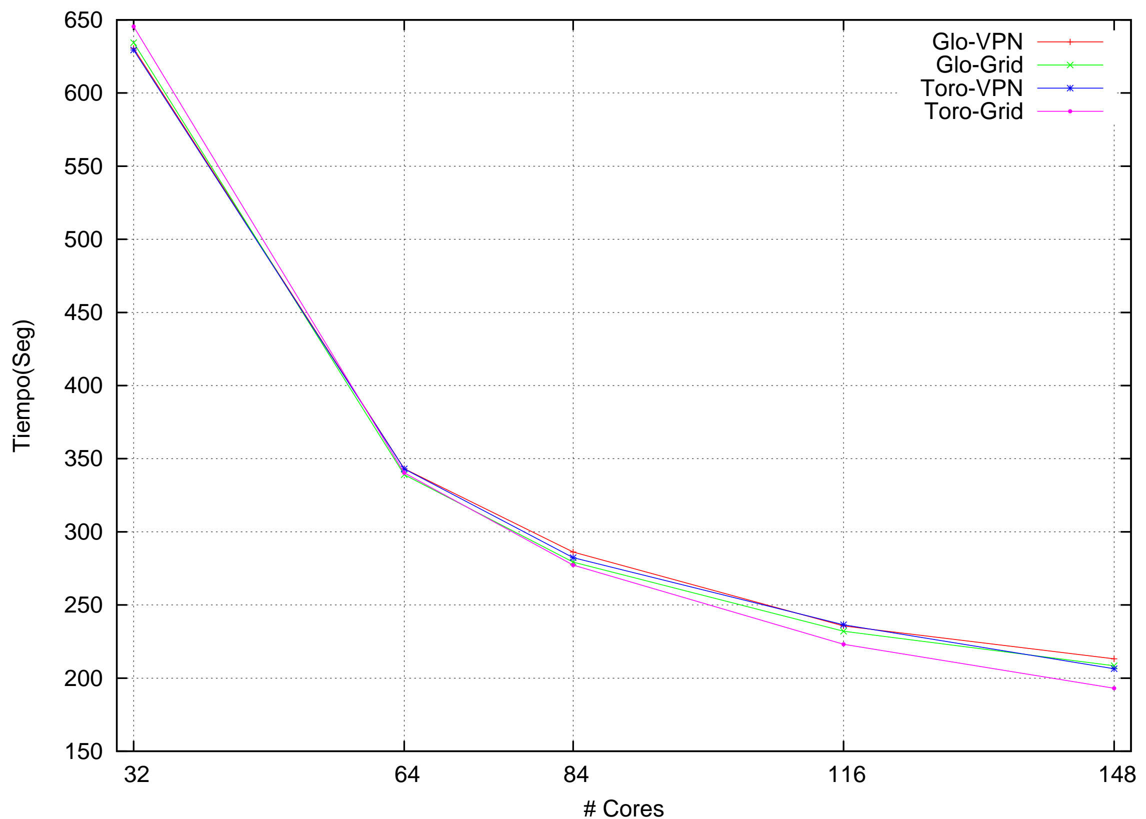

Figura 5.14: Tiempo de ejecución para Multiplicación de Matrices con las 4 versiones

En esta figura podemos ver que la versión más lenta es Toro-Grid con 32 cores, sin embargo

es la más rápida al aumentar el número de cores. No obstante de las mejoras presentadas por Glo-Grid y Toro-Grid, el alto costo de las transferencias de carga de un cluster a otro, en Multiplicación de Matrices, hace que estas dos versiones muestren mejoras mientras la proporción de la capacidad de cómputo externo (a un único cluster) sea mayor al 28 %, en otro caso al ejecutar la aplicación en más de un cluster la velocidad de procesamiento global aumentará, en lugar de disminuir. Esto lo podemos ver en la Tabla 5.11, en donde mostramos la velocidad de procesamiento al procesar 360000 datos, en Multiplicación de Matrices sobre uno y dos clusters.

#Cores 1Cluster	*Global*	*Toroide*	*#Cores 2Cluster*	*Global-Grid*	*Tor-Grid*
16=42.72Ghz	1714.64	1714.62	32=81.12Ghz	979.14	1001.44
44=117.48Ghz	628.84	626.93	64=165.48Ghz	517.11	515.79
64=170.88Ghz	441.16	441.54	84=218.88Ghz	418.93	417.86
96=256.32Ghz	297.39	299.77	116=304.32Ghz	342.06	334.031
128=341.76Ghz	226.75	225.96	148=389.76Ghz	300.6601	287.15

Tabla 5.11: Tiempo de ejecución de Multiplicación de Matrices en 1 y 2 clusters

Si tomamos como 100 % la capacidad de procesamiento del cluster 1 (columna 1 de la Tabla 5.11) y obtenemos la proporción de la capacidad de procesamiento de los dos clusters (columna 4 de la Tabla 5.11), podemos observar que en Multiplicación de Matrices la versión Glo-Grid muestra menores tiempos de ejecución cuando el cluster externo representa más del 28 % de la capacidad de procesamiento del cluster 1. Esto se puede observar en la Tabla 5.12 (con 16, 44 y 64 cores en un cluster, y 32, 64 y 84 cores en 2 clusters).

En la Figura 5.15 comparamos los resultados que se obtuvieron al usar un cluster con los que se obtuvieron al usar dos clusters. En estas pruebas en particular podemos ver que es recomendable usar más de un cluster, sólo mientras la capacidad de cómputo externa sea mayor al 28 %, como con 32, 64 y 84 cores. Con 116 y 148 cores, el tiempo de procesamiento de los dos clusters es más alto (a pesar de contar con más cores), que el que se obtiene al procesar la misma cantidad de carga, con 96 y 128 cores en un sólo cluster.

#Cores 1Cluster	*Proporción (%)*	*#Cores 2Cluster*	*Proporción (%)*
16=42.72Ghz	100	32=81.12Ghz	189.88
44=117.48Ghz	100	64=165.48Ghz	140.86
64=170.88Ghz	100	84=218.88Ghz	128.09
96=256.32Ghz	100	116=304.32Ghz	118.73
128=341.76Ghz	100	148=389.76Ghz	114.04

Tabla 5.12: Proporción en la capacidad de procesamiento en uno y dos clusters

Algo similar ocurre en N-Reinas (en donde se hacen comparaciones con la misma capacidad de procesamiento), sin embargo en N-Reinas, las versiones Global y Toroide al ser ejecutadas sobre un sólo cluster, nunca sobrepasan la velocidad de procesamiento de Glo-Grid y Toro-Grid con dos clusters. No obstante, muestran el mismo comportamiento al aumentar la diferencia en las capacidades de procesamiento de los clusters. En Multiplicación de Matrices esto es más evidente debido al costo de las transferencias iniciales en esta aplicación.

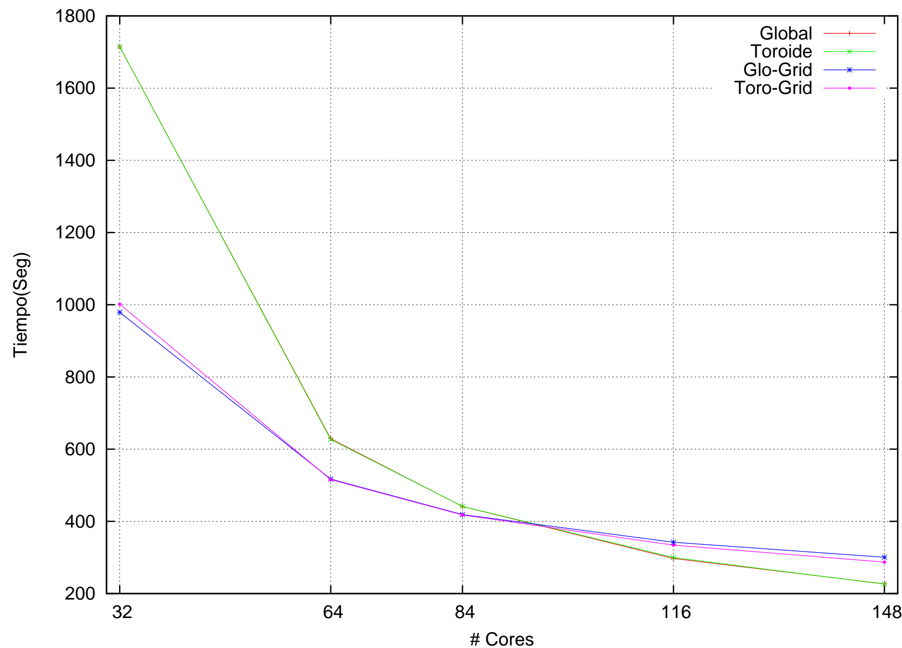

Figura 5.15: Tiempo de ejecución de Multiplicación de Matrices en 1 y 2 clusters

5.5. Resultados en la distribución de carga de Multiplicación de Matrices

Las pruebas de distribución de carga se hicieron con 148 cores al ejecutar Multiplicación de Matrices con matrices de 600x600. Los resultados obtenidos en las versiones Glo-VPN, Glo-Grid, Toro-VPN y Toro-Grid se muestran la Figura 5.16.

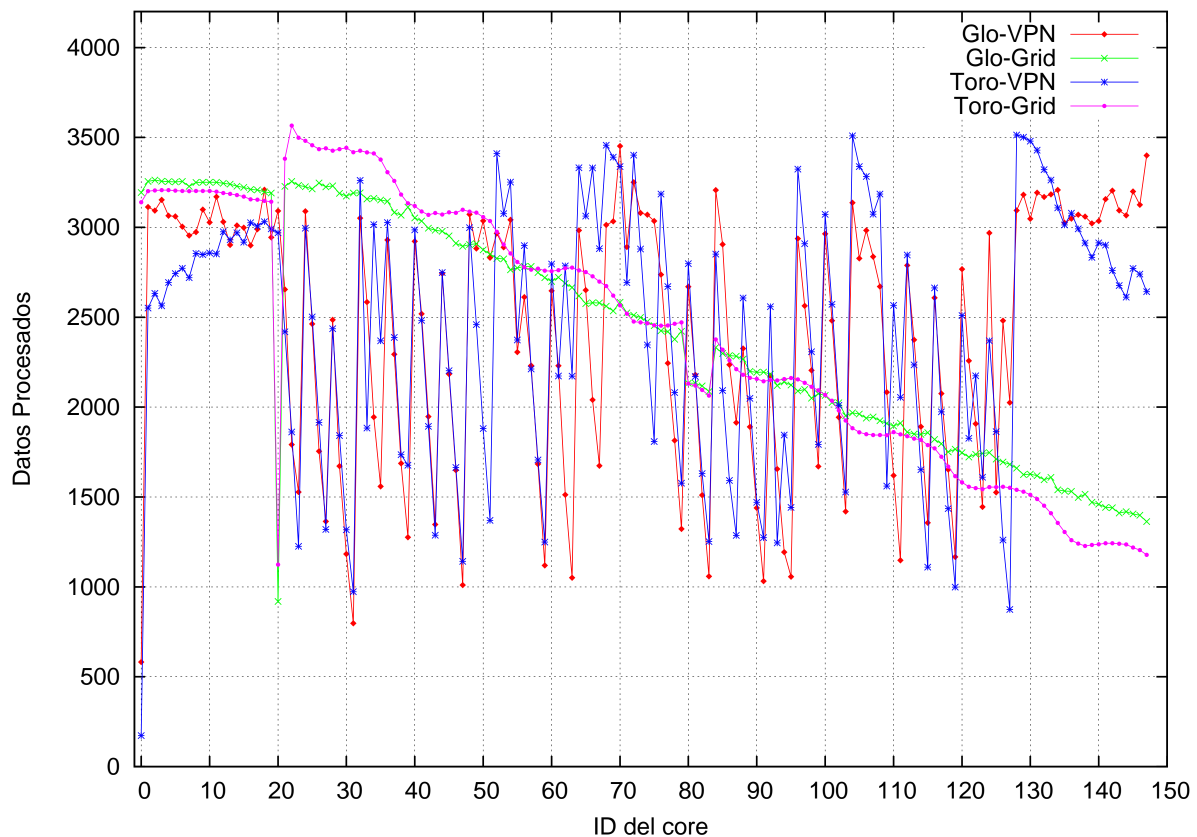

Figura 5.16: Distribución de carga para Multiplicación de Matrices con las 4 versiones

Como en el caso de N-Reinas los primeros 20 cores pertenecen al cluster Xcaret, en donde se genera la carga total y se distribuye hacia los cores del segundo cluster Xena (los siguientes 128 cores en la VPN). Como se puede ver Glo-VPN y Toro-VPN presentan un comportamiento similar en la distribución de carga perteneciente al cluster Xcaret, ya que

en las dos versiones el primer core procesa muy poca carga (a diferencia de los otros cores de Xcaret), y a partir del segundo core la carga aumenta significativamente hasta el core 19. Esto es debido a que desde el primer core de Xcaret, se distribuye carga hacia todos los cores de la VPN (en la transferencia inicial). Como la transferencia inicial es de más de 500 MB (al procesar una matriz de 600x600 sobre 148 cores), el primer core inicia su procesamiento en un tiempo diferente al de sus cores vecinos (al finalizar la transferencia), ya que debe distribuir carga a los cores que están en el segundo cluster (a través de Internet). Por esta razón, al momento de finalizar la transferencia, muchos de los cores del primer cluster están por finalizar su carga inicial o ya la han finalizado y están en busca de carga, muchos de ellos la obtienen del primer core y de los cores más lentos, de esta forma el primer core dona mucha de su carga a otros cores.

En las versiones Glo-Grid y Toro-Grid la distribución de carga en el cluster Xcaret (primeros 20 cores) es similar y equitativa, sin embargo, en las dos versiones podemos ver una disminución en la cantidad de carga, conforme se alejan los cores del cluster Xcaret. Esto quiere decir, que los primeros cores de Xena procesan más carga que los últimos. Lo anterior es una consecuencia de la transferencia inicial de Xcaret a Xena, ya que Xcaret divide y envía una cantidad de carga proporcional al número de cores en Xena, al inicio del procesamiento. En estos resultados también podemos ver que en los cores 80, 81, 82 y 83 se presenta una menor cantidad de carga procesada (de la misma forma que en N-Reinas), esto se debe a que estos cores pertenecen a un nodo de Xena con menor capacidad de procesamiento.

La transferencia de Xcaret a Xena no afecta el procesamiento de Xcaret, ya que en estas versiones se cuenta con el *Administrador Cluster*, que se encarga de enviar la carga mientras sus cores trabajan. Esta transferencia se hace entre los dos *Administradores Clusters* de Xcaret y Xena respectivamente, sin embargo conforme el *Administrador Cluster* de Xena recibe la carga éste la va enviando a sus cores trabajadores de forma ordenada uno tras otro, esto quiere decir que los últimos cores de Xena, reciben su carga al final de la transferencia y los primeros cores la reciben al principio. De la misma forma que en el cluster Xcaret, los cores que reciben primero la carga son los que procesan más carga, ya que inician el procesamiento

antes que los demás. En las gráficas correspondientes a Glo-Grid y Toro-Grid de la Figura 5.16, podemos ver como decrece la cantidad de carga que procesan los cores que están más alejados de Xcaret.

Aunque Toro-Grid, en promedio muestra mejores resultados en tiempo de procesamiento, Toro-Grid no es el que mejor que distribuye la carga en los 148 cores trabajadores. En la Tabla 5.13 tenemos la desviación estándar de las 4 versiones de DLML, y la versión que mejor distribuye la carga es la versión Glo-Grid con una desviación estándar de 634. Esto también se ve reflejado en la gráfica perteneciente a Glo-Grid de la Figura 5.16, en donde podemos ver menos fluctuaciones (en la cantidad de carga) inclusive que la gráfica de Toro-Grid.

Versión	*Desviación estándar*
Glo-VPN	713
Glo-Grid	634
Toro-VPN	714
Toro-Grid	728

Tabla 5.13: Desviación estándar de la carga procesada por las 4 versiones

Habiendo mostrado los resultados obtenidos y su análisis, en el siguiente Capítulo se presentan las conclusiones y el trabajo a futuro.

Capítulo 6

Conclusiones y trabajo a futuro

En este trabajo de tesis, se diseñó e implementó una nueva arquitectura para la librería DLML sobre un ambiente Grid. El objetivo de esta propuesta es utilizar recursos de más de un cluster, mejorando el funcionamiento de las versiones originales de DLML que solamente funcionan en un único cluster.

La arquitectura propuesta sigue un modelo jerárquico de distribución de carga, en el cual se favorece al balance de carga *intra*-cluster sobre el balance de carga *inter*-cluster. Es decir, la solicitud de transferencia de carga de un cluster X a otro Y no podrá efectuarse, mientras exista carga en el entorno local de X. En esta arquitectura, para el balance de carga *inter*-cluster se usa información de carga de todos los clusters (balance global), mientras que para el balance *intra*-cluster, se usa DLML con información global o DLML con información parcial. Con la nueva arquitectura, se logró disminuir el tiempo de procesamiento al usar más de un cluster para procesar y balancear carga en aplicaciones que demandan más recursos, que los que están presentes en un sólo cluster.

Para crear el ambiente Grid se usó una VPN (Virtual Private Network), la cuál permite interconectar un conjunto de clusters a través de Internet, y usar los recursos de los dos clusters, de la misma forma que en un cluster normal. Usando la arquitectura jerárquica, se reduce el número de comunicaciones (costo en las comunicaciones) entre diferentes clusters (ya que las comunicaciones entre clusters se hace a través de Internet), y en consecuencia también se reduce el tiempo de procesamiento global. Esto es importante, ya que al tener una VPN sobre varios clusters, las versiones originales de DLML (DLML con información global

y DLML con información parcial) pueden ejecutarse sobre los recursos de todos los clusters (recursos VPN). Sin embargo, como en estas versiones no se hace una distinción entre las comunicaciones locales (comunicaciones entre cores del mismo cluster) y las comunicaciones que se hacen con recursos de otro cluster (a través de Internet), se provoca un aumento en el costo de las comunicaciones y en el tiempo de procesamiento global.

Para las pruebas realizadas se usaron las dos versiones originales de DLML (a las cuales denominamos Glo-VPN y Toro-VPN) y dos nuevas versiones de DLML, una denominada Glo-Grid, en donde se hace balance de carga sobre un ambiente Grid usando DLML con información global para el balance *intra*-cluster y otra denominada Toro-Grid, en donde también se hace balance de carga sobre un ambiente Grid, pero usando DLML con información parcial para el balance *intra*-cluster. Usando estas cuatro versiones de DLML, se ejecutó la aplicación dinámica N-Reinas y la aplicación estática Multiplicación de Matrices sobre una VPN formada por 2 clusters, uno en la UAM-Iztapalapa y otro en el CINVESTAV.

Los resultados para el tiempo de procesamiento y la distribución de carga son los siguientes:

- **Tiempo de procesamiento**

  Para la aplicación dinámica de las N-Reinas, las dos nuevas versiones de DLML Glo-Grid y Toro-Grid mostraron mejorías en el tiempo de procesamiento, con respecto a las versiones orinales Glo-VPN y Toro-VPN al ejecutarlas sobre los recursos de la VPN. Esta mejoría se debió a la reducción de las comunicaciones a través de Internet en las 2 nuevas versiones.

  En Multiplicación de Matrices, se obtuvieron resultados similares en el tiempo de procesamiento, Toro-Grid fue la versión más rápida de las 4 versiones, a pesar de que en esta aplicación la carga transferida de un cluster a otro fue mucho más grande que en N-Reinas (lo que implica un retraso debido a que las transferencias se hacen a través de Internet). Por este motivo, en Multiplicación de Matrices se observó, en este caso particular, que es aconsejable usar más de un cluster para procesar carga, siempre que la proporción del poder de cómputo externo sea mayor al 28 % del cómputo del cluster

local. En caso contrario, podría ser que un sólo cluster termine de procesar la carga más rápido que con la ayuda de otros clusters. A diferencia de N-Reinas, en donde no se observó este comportamiento.

En estas pruebas también se observó que en Multiplicación de Matrices la diferencia de velocidades entre Glo-VPN y Glo-Grid no fue tan evidente como en el caso de N-Reinas, esto se debe a que la principal ventaja de la nueva arquitectura es limitar el número de mensajes a través de Internet. Por consiguiente, si el número de peticiones a través de Internet en las versiones Glo-VPN o Toro-VPN no es significativo (como en el caso de Multiplicación de Matrices), la nueva arquitectura no podrá mostrar grandes ventajas.

- **Distribución de carga**

  En la distribución de carga, la aplicación dinámica de las N-Reinas mostró que usando la versión Toro-Grid, la cantidad de carga asignada a cada core (en cada uno de los clusters) es más equitativa que en la versión Toro-VPN, y en las otras dos versiones. También se observó que la cantidad de carga que procesa un cluster está en relación al poder de cómputo que este posee, con respecto al poder de cómputo total (el poder de cómputo de la VPN).

  La versión Glo-Grid también mostró buenos resultados con respecto a la versión Glo-VPN, en la cual, se obtuvo la peor distribución de carga de las cuatro versiones. Esto se debe a que en esta versión se usa DLML con subasta global, en donde el costo de comunicación es muy alto, ya que cada core se debe comunicar con todos los cores de la VPN al hacer un balance de carga, y esto incrementa el número de mensajes a través de Internet.

  En Multiplicación de Matrices se obtuvieron resultados diferentes. La versión que mejores resultados mostró (aunque no hubo mucha diferencia como en N-Reinas) fue Glo-Grid, seguida de Glo-VPN, Toro-VPN y Toro-Grid. Esto se debe, a que al ser una aplicación estática no se necesita en general de muchas búsquedas y transferencias, ya que se distribuye la misma cantidad de carga a todos los cores, los cuales tienen ca-

pacidades de procesamiento similares y por consiguiente terminan de procesar su carga en tiempos similares. Por esta razón, la subasta parcial (del toroide) no tuvo la oportunidad de mostrar las ventajas que tiene al presentarse un gran número búsquedas y transferencias frente a la subasta global, como en el caso de N-Reinas.

Con el desarrollo e implementación de la nueva arquitectura, DLML puede balancear carga en uno o varios clusters (ambiente Grid) interconectados por una VPN. Esto brinda la posibilidad de ofrecer mayor poder de cómputo a problemas que demandan más recursos de los que están presentes en un único cluster.

Como se pudo observar a lo largo de este trabajo, actualmente DLML proporciona un balance de carga a nivel Grid, en donde un cluster descargado ofrece su poder de cómputo a otros clusters (balance iniciado por el cluster receptor). Sin embargo, sería conveniente desarrollar una versión en donde un cluster que se encuentra sobrecargado, pida ayuda a otros clusters que tengan menos carga (balance iniciado por el cluster emisor). Ya que con estos dos tipos de balance, probablemente se obtendrían mejores resultados en cuanto a tiempo de procesamiento y balance de carga.

Otro aspecto importante a considerar en el balance de carga *inter*-cluster, es la determinación de la capacidad de un cluster para procesar una cierta cantidad de carga, al momento de determinar el porcentaje de carga que se enviará del cluster emisor al cluster receptor. Para evaluar dicha capacidad, en este trabajo únicamente se toma en cuenta el poder de procesamiento del cluster en base al poder de cómputo de todos sus nodos. A partir del poder de procesamiento de los clusters se determina el porcentaje de carga que debe transferir el cluster emisor. Sin embargo, el poder de procesamiento no es lo único que determina la capacidad de un cluster para procesar una carga, se deben considerar aspectos como la velocidad de la red interna del cluster, la memoria cache de los procesadores y las capacidades en RAM y disco duro para aplicaciones que así lo requieran. Aunado a esto, en esta tesis sólo se toma en cuanta la cantidad de carga que reporta un cluster para designarlo como emisor. Sin embargo, en un ambiente Grid también se debe considerar el costo en la comunicación al hacer una transferencia, por lo que es necesario tomar en cuenta la latencia del cluster

receptor hacia los demás clusters (al momento de designar a uno de ellos como emisor). Por lo anterior, seria recomendable crear un modelo analítico que considere los aspectos mencionados para evaluar la transferencia de carga de un cluster a otro, con el objetivo de hacer más eficiente el balance de carga y en consecuencia mejorar el tiempo de procesamiento global.

Por otra parte, ahora que se hace procesamiento en más de un cluster, es importante no perder de vista la posibilidad de falla en alguno de los cluster o su conexión a Internet. Con una falla de este tipo se perderían una gran cantidad de datos, por lo cual, se necesita un mecanismo que permita recuperar los datos perdidos y los resultados parciales que se vayan obteniendo en cada uno de los clusters, para poder finalizar el procesamiento global sin pérdida de datos o resultados parciales. Este mecanismo podría ser un dispositivo intermedio (que también podría estar distribuido en varios cluster con replicación), en donde se almacenen los resultados parciales y los datos que se van enviando a los clusters antes del procesamiento, con el fin de recuperar los resultados parciales que había obtenido el cluster (hasta antes de la falla) y de recuperar los datos que debió haber procesado un cluster que ha fallado. Al contar con un respaldo de estos datos, éstos se pueden asignar a otro cluster en el momento en que haga una petición de carga al *Administrador Grid*.

También es importante recordar, que actualmente el ambiente Grid sobre el que se ejecuta DLML está formado por una VPN, la cual proporciona una forma relativamente sencilla para interconectar clusters. Sin embargo, esta no cuenta con sistema para el descubrimiento de recursos, como en el caso del sistema Grid Globus. Por lo cual, sería conveniente probar y hacer modificaciones a las nuevas versiones de DLML, para ejecutarlas en un sistema Grid como Globus.

Apéndice A

Instalación y configuración de OpenVPN

A.1. Instalación

Para formar una VPN entre 2 clusters se debe instalar y configurar OpenVPN en los servidores de los 2 clusters, siguiendo las instrucciones que se muestran a continuación (si se desea más información sobre la configuración o instalación se puede consultar el libro *Beginning OpenVPN 2.0.9.* [25]).

Instalar openvpn2.1.1 en un SO basado en RedHat

Primero se debe descargar openvpn2.1.1 de su página principal con el comando:
wget http://openvpn.net/release/openvpn-2.1.1.tar.gz

El archivo descargado contiene e código fuente de OpenVPN, por lo cual se debe crear el archivo RPM para instalarlo en un SO basado en RedHat. Para crear un RPM se debe instalar *rpm-build* con el comando: *yum install rpm-install*
Para crear el RPM y evitar problemas de algunas dependencias se usa el comando:
rpmbuild –define=‘with_pkcs11=“”’-ta openvpn-2.1.1.tar.gz
Al intentar crear el RPM de OpenVPN se puede presentar un mensaje como el siguiente:
error: Falló la construcción de dependencias:
openssl-devel >= 0.9.6 se necesita para openvpn-2.1.1-1. x86_64

lzo-devel >= 1.07 se necesita para openvpn-2.1.1-1. x86_64

pam-devel se necesita para openvpn-2.1.1-1. x86_64

En este caso, se debe instalar primero el repositorio DAG, dependiendo de la versión de Linux se debe descargar este repositorio. En el caso de CentOS-5.4-x86_64, se necesitó descargar el repositorio siguiente:

wget http://dag.wieers.com/rpm/packages/rpmforge-release/rpmforge-release-0.3.6-1.el5.rf.x86_64.rpm

Después de descargarlo se instala con el siguiente comando:

rpm -ivh rpmforge-release-0.3.6-1.el5.rf.x86_64.rpm

Nota: Después de la instalación, el archivo *rpmforge.repo* debe estar presente en el directorio */etc/yum.repos.d*

Ahora que se ha instalado el repositorio, se instalan los paquetes requeridos con los comandos:

yum install openssl-devel

yum install lzo-devel

yum install pam-devel

Después de instalar los paquetes intentamos crear el rpm de OpenVPN con el comando:

rpmbuild –define='with_pkcs11=“”'-ta openvpn-2.1.1.tar.gz

Si todo ha salido bien deberíamos tener el rpm en el siguiente directorio:

/usr/src/redhat/RPMS/x86_64/openvpn-2.1.1-1.x86_64.rpm

Ya que se tiene el RPM, se instala o se actualiza OpenVPN con el comando:

rpm -Uvh /usr/src/redhat/RPMS/x86_64/openvpn-2.1.1-1.x86_64.rpm (para actualizar)

rpm -ivh /usr/src/redhat/RPMS/x86_64/openvpn-2.1.1-1.x86_64.rpm (para instalar)

Instalación sobre un SO basado en Debian

En caso de tener un sistema basado en Debian la instalación de OpenVPN es muy simple, ya que los repositorios de estos sistemas tienen la versión más actual de OpenVPN. Para instalar OpenVPN se usa el siguiente comando: *aptitude install openvpn*

A.2. Configuración del Servidor VPN

Después de instalar OpenVPN en el servidor del cluster, se configura OpenVPN como servidor en el cluster elegido como servidor VPN. Para esto se seguirán una serie de pasos descritos a continuación:

Paso 1

Entramos al directorio: */etc/openvpn* y copiamos en este directorio la siguiente carpeta */usr/share/doc/openvpn-2.1.1/easy-rsa* con el siguiente comando:

cp -r /usr/share/doc/openvpn-2.1.1/easy-rsa

Para crear las claves para los clientes y para el servidor entramos al directorio:

/usr/share/doc/openvpn-2.1.1/easy-rsa/2.0

En este directorio se encuentra el archivo *vars*, el cual nos servirá para la configuración, en este archivo se deben modificar las últimas 5 líneas con la información de institución a donde pertenece el cluster. Ya que el cluster Xcaret pertenece a la UAM y ha sido elegido como servidor, se usa la siguiente información para modificar el archivo *vars*.

export KEY_COUNTRY= “MX”

export KEY_PROVINCE= “DF”

export KEY_CITY= “DistritoFederal”

export KEY_ORG= “UAM-I”

export KEY_EMAIL= “apolo.h.s@gmail.com ”

Después de la modificación se actualizan las variables del archivo *vars* con los siguientes comandos:

. vars

./clean-all

Paso 2

Ahora se inicia la creación de los archivos de certificación necesarios para los clientes, para hacer esto usamos el comando:
./build-ca

Con este comando OpenVPN hace una serie de preguntas sobre el servidor y la institución, muchas de ellas tendrán una sugerencia (obtenida de los datos en el archivo *vars* que hemos modificado), para aceptar las sugerencias sólo se presiona Enter.
Ahora pasamos a crear la llave para el servidor con el siguiente comando:
./build-key-server server

Al igual que el comando anterior, OpenVPN hace una serie de preguntas, cuyas respuestas están en el archivo *vars*. Una vez que se ha creado la llave para el servidor, pasamos a crear las llaves para cada uno de los clientes (que se conectaran con el servidor para pertenecer a la VPN) con el siguiente comando:
./build-key client1

Después de responder a la serie de preguntas de OpenVPN, se crea una llave para un cliente llamando "client1". Si queremos más clientes, usamos el mismo comando con un nombre diferente para el cliente.
Para finalizar con la creación de archivos para el servidor usamos el comando:
./build-dh
Ahora copiaremos los siguientes archivos al directorio */etc/openvpn* del servidor:
* *ca.crt*
* *ca.key*
* *server.key*
* *server.crt*
* *dh1024.pem*

Estos archivos están presentes en */usr/share/doc/openvpn-2.1.1/easy-rsa/2.0/keys*. En el mismo directorio están los archivos para el cliente (se deben mandar al cliente de una forma segura):

* *ca.crt*

* *client1.crt*

* *client1.key*

Paso 3

Ahora nos movemos al directorio */etc/openvpn* y ahí creamos el archivo de configuración "server.conf" para el servidor con la siguiente información:

---

port 1194 /* puerto por defecto del servidor OpenVPN */

proto tcp /* protocolo para el intercambio de información*/

dev tun /* tipo de dispositivo de red virtual a través del cual se accede al túnel*/

#— Archivos para el cifrado —

ca ca.crt

cert server.crt

key server.key

dh dh1024.pem

#———————————

server 10.1.0.0 255.255.255.0 /* rango de direcciones a asignar a los clientes que se conecten*/

ifconfig-pool-persist ipp.txt

#directorio para la red de los clientes

client-config-dir ccd

#redes de los clientes

#client1

route 192.168.1.0 255.255.255.0

push route 179.178.177.0 255.255.255.0/*ruta a insertar en la tabla de ruteo del cliente*/

keepalive 10 120

---

persist-key

persist-tun

status openvpn-status.log

verb 3

Para que la configuración esté completa, se debe crear un archivo por cada cliente en el directorio ccd (dentro de /etc/openvpn). Este archivo se crea con el mismo nombre del cliente (para nuestro caso se llama “client1”), en el cual se anota información de la red y la máscara de red que usa el cliente, de este modo:
iroute 192.168.1.0 255.255.255.0

A.3. Configuración del Cliente VPN

La configuración del cliente es relativamente simple, ya que desde la configuración de servidor hemos mandado los archivos necesarios al directorio */etc/openvpn* del cliente:
* *ca.crt*
* *client1.crt*
* *client1.key*

Con estos archivos, sólo se necesita el archivo de configuración “client.conf”, que debe tener la siguiente información:

client

dev tun /* tipo de dispositivo de red virtual a través del cual se accede al túnel*/

proto tcp /* protocolo para el intercambio de información*/

remote (IP del servidor) 1194 /*IP del servidor y el puerto que usa para Open-VPN*/

resolv-retry infinite

nobind

persist-key

persist-tun

#—— Archivos para el cifrado ——

ca ca.crt

cert client1.crt

key client1.key

#——————————————————-

verb 3

Ya que se tiene al cliente y al servidor configurados, se inicia OpenVPN en cada uno de ellos. Para probar el túnel se pueden lanzar pings desde los nodos del cliente a los del servidor y viceversa.

Una vez que se ha probado el túnel, debemos configurar los nodos para que se pueda acceder a ellos mediante ssh, sin necesidad de password. Para poder hacer esto, copiamos el contenido del archivo authorized_keys del cluster servidor y lo pegamos al final del archivo authorized_keys del cluster cliente, del mismo modo copiamos el contendio de authorized_keys del cliente y lo pegamos al final del archivo authorized_keys del servidor. El archivo authorized_keys está en el directorio *.ssh* del HOME del usuario que usará la VPN. Con esta configuración se tendría una VPN formada por todos los nodos del cluster servidor y por los nodos del cluster cliente, preparada para ejecutar cualquier aplicación paralela.

Apéndice B

Configuración del Toroide

El toroide que usa la versión de DLML con subasta parcial puede adoptar diversas configuraciones para un determinado número de cores. Por ejemplo, si tenemos un cluster con 16 cores, las posibles configuraciones (de un toroide MxN para la distribución de carga) son: 2x8, 4x4 y 8x2. De estas tres configuraciones tomaremos 8x2 y 4x4 (ya que con la configuración 2x8 el toroide obtiene la misma forma que con la configuración 8x2) para ilustrar como se hace la distribución de carga en el toroide, y cuál de estas dos configuraciones es la mejor, Figura B.1.

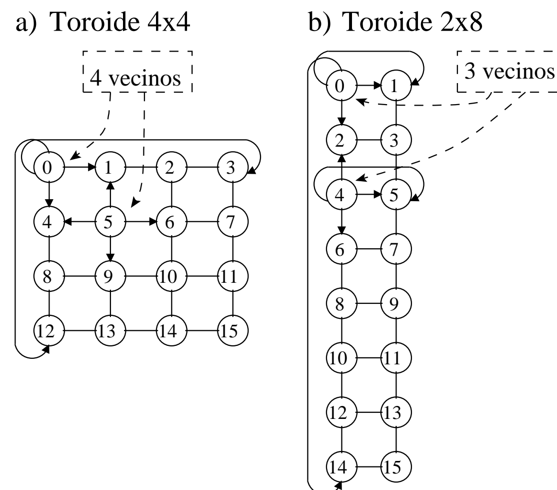

Figura B.1: Configuraciones del toroide para un cluster de 16 cores

Como ya se había mencionado, para hacer la distribución de carga en el toroide, cada core cuenta con 4 cores vecinos, sin importar la configuración del toroide. Sin embargo, con

diferentes configuraciones se puede tener un número diferente de vecinos reales. En la Figura B.1 tenemos dos configuraciones diferentes para un cluster de 16 cores, por una parte tenemos un toroide de 4x4, en donde cada core tiene 4 vecinos diferentes (Figura B.1.a). En cambio, en el toroide de 8x2 (Figura B.1.b) cada core sólo tiene 3 vecinos diferentes (ya que el vecino de la izquierda es el mismo que el vecino de la derecha).

Cuando los cores tienen más vecinos con los cuales repartir su carga, la distribución es más equitativa en todos los cores del cluster. Esto lo podemos ver en la Figura B.2, en donde se presenta la forma en que los 16 cores de un cluster son alcanzados al distribuir la carga en los dos toroides, presentados en la Figura B.1.

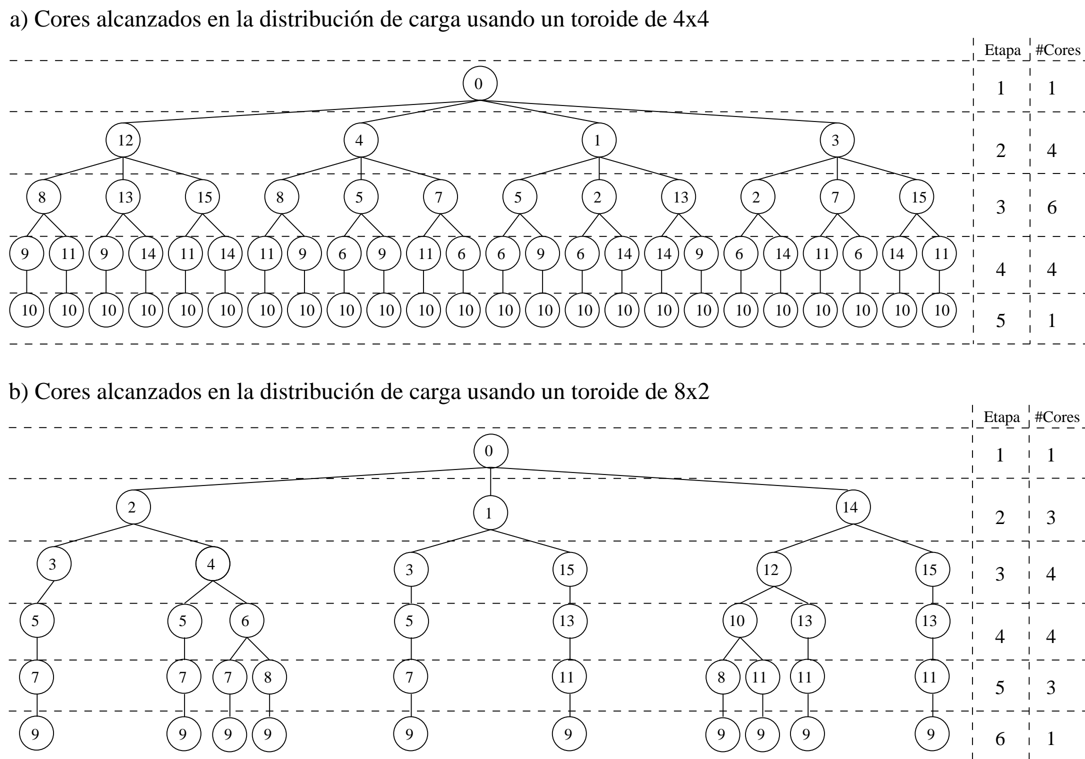

Figura B.2: Etapas en la distribución de carga al usar el algoritmo toroide

En la Figura B.2.a tenemos al toroide de 4x4, en donde la carga se distribuye desde el core 0 hasta el core 10, quién es el core más alejado de la raíz. El core 0 (está en la primera etapa) distribuye la carga a sus 4 cores vecinos (12, 4, 1 y 3), en la segunda etapa los 4 cores (12, 4, 1 y 3) distribuyen la carga hacia sus cores vecinos. Sin embargo, en ninguna etapa

se puede acceder a un core que ya fue alcanzado en una etapa anterior, por esta razón, en la tercera etapa se tiene a 3 cores vecinos por cada core de la segunda etapa (ya que en la segunda etapa se encuentra el cuarto vecino). La distribución continúa hasta alcanzar a los 16 cores del cluster, como se puede ver en la columna #Cores de la Figura B.2.a (en donde se tiene el número de cores diferentes que se alcanzan en cada etapa).

En la Figura B.2.b tenemos al toroide de 8x2, en este toroide no se tienen 4 vecinos diferentes, sólo 3 por cada core, y en consecuencia la forma en que se alcanza a los cores es completamente diferente a la del toroide de 4x4. También se debe notar que al no tener 4 vecinos reales, la carga se distribuye en 6 etapas (una más que en el otro toroide), debido a que hay menos conexiones para llegar de un core a otro. Por esta razón, la forma en que se distribuye la carga en los dos toroides es completamente diferente. Como se muestra en la Figura B.3

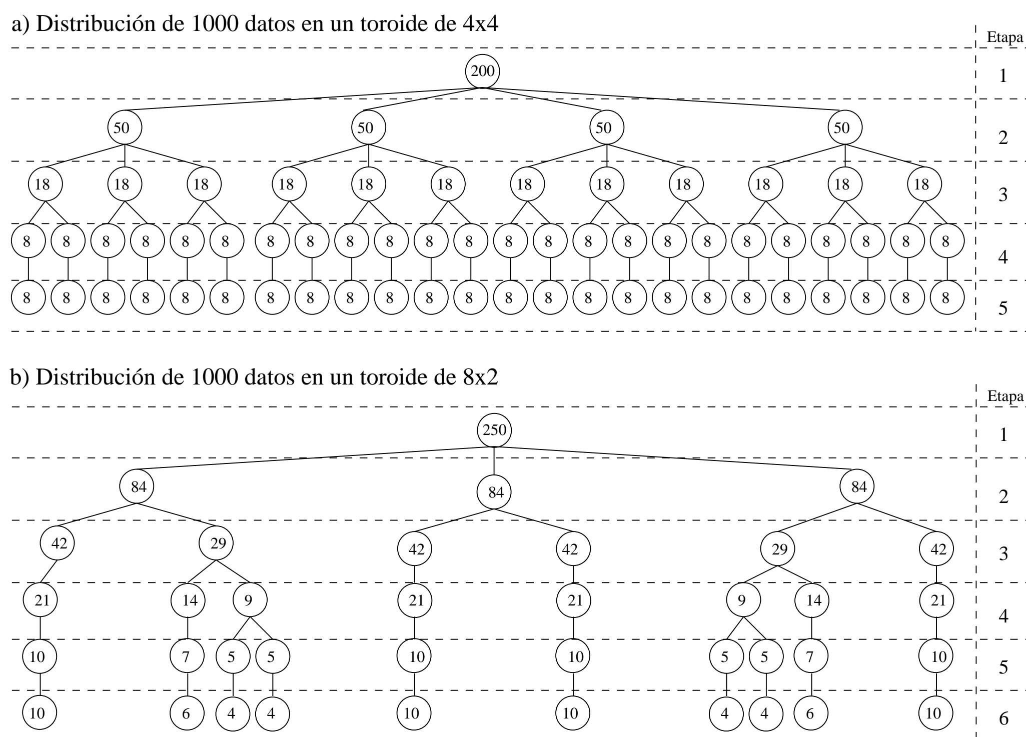

Figura B.3: Distribución de 1000 datos en un cluster de 16 cores

En la Figura B.3.a podemos observar cómo se distribuyen 1000 datos sobre los 16 cores,

a lo largo de las 5 etapas. Con el toroide de 4x4 la distribución de carga es equitativa en cada etapa, a pesar de que no lo sea de una etapa a otra. Por otra parte, tenemos al toroide de 8x2 en la Figura B.3.b, en donde tenemos una distribución menos equitativa (no es equitativa ni siquiera en cada etapa) en comparación con el toroide de 4x4.

Comparando los dos toroides, podemos ver que la distribución de carga es mejor en el toroide de 4x4, y además se alcanzan los 16 cores en un menor número de etapas. Por lo anterior, es recomendable usar configuraciones para el toroide que se acerquen a un toroide cuadrado, esto quiere decir, que para la distribución de carga las configuraciones de MxN donde la diferencia entre M y N es pequeña, son mejores que las configuraciones donde la diferencia entre M y N es más grande.

Referencias

[1] Top500 supercomputing sites. url: *http://www.top500.org/list/2010/11/100*, December 2010.

[2] Miguel Alfonso Castro García. *Programación con Listas de Datos para Cómputo Paralelo en Clusters*. PhD thesis, CINVESTAV-México, Departamento de Ingeniería Eléctrica, México, D.F., 2007.

[3] Open mpi: Open source high performance computing. url: *http://www.open-mpi.org*, Febraury 2010.

[4] Lam/mpi parallel computing. url: *http://www.lam-mpi.org*, Febraury 2010.

[5] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[6] A. Plastino, C. C. Ribeiro, and N. Rodriguez. Developing spmd applications with load balancing. *Parallel Computing*, 29(6):743 – 766, 2003.

[7] Neeraj Nehra, R.B. Patel, and V.K. Bhat. A framework for distributed dynamic load balancing in heterogeneous cluster. *Journal of Computer Science*, 3(1):14–24, 2007.

[8] Juan Santana Santana. Algoritmos de balance de carga con manejo de información parcial. Master's thesis, UAM-Iztapalapa, Departamento de Ingeniería Eléctrica, México, D.F., 2009.

[9] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco USA, 1st edition, 1998.

[10] A. Segall. Distributed network protocols. *Information Theory, IEEE Transactions on*, 29(1):23 – 35, 1983.

[11] Ian Foster, Carl Kesselman, and Tuecke S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 2001.

[12] Grid computing info centre: Frequently asked questions (faq). url: *http://www.gridcomputing.com/gridfaq.html*, October 2009.

[13] Maozhen Li and Mark Baker. *The Grid: Core Technologies*. John Wiley & Sons, 2005.

[14] Ian Foster. What is the grid? a three point checklist. *GRIDtoday*, 1(6), July 2002.

[15] Miguel L. Bote-Lorenzo, Yannis A. Dimitriadis, and Eduardo Gómez-Sánchez. Grid characteristics and uses: a grid definition. In *Across Grids 2003, LNCS 2970*, pages 291–298, 2003.

[16] Wfmc (workflow management coalition). url: *http://www.wfmc.org*.

[17] Belabbas Yagoubi and Meriem Medebber. A load balancing model for grid environment. *IEEE*, 2007.

[18] Elie El Ajaltouni, Azzedine Boukerche, and Ming Zhang. An efficient dynamic load balancing scheme for distributed simulations on a grid infrastructure. *IEEE*, 2008.

[19] R.U. Payli, E. Yilmaz, A. Ecer, H.U. Akay, and S. Chien. Dlb-a dynamic load balancing tool for grid computing. In *In Proceedings of Parallel CFD'04 Conference, Canary Island*, May 2004.

[20] Junwei Cao, Daniel P. Spooner, Stephen A. Jarvis, and Graham R. Nudd. Grid load balancing using intelligent agents. *Future Generation Computer Systems*, pages 135–149, 2005.

[21] Yawei Li and Zhiling Lan. A survey of load balancing in grid computing. In LNCS, editor, *Computational and Information Science, First International Symposium, CIS 2004*, pages 280–285, 2004.

[22] Jens Mache, Damon Tyman, Andre Pinter, and Chris Allick. Performance implications of using vpn technology for cluster integration and grid computing. *IEEE*, 2006.

[23] Glite part of the egee project. url: *http://glite.web.cern.ch/glite/*, Febraury 2010.

[24] Markus Feilner. *OpenVPN Building and Integrating Virtual Private Networks*. Packt Publishing, April 2006.

[25] NewAuthor1 and Norbert Graf. *Beginning OpenVPN 2.0.9. Build and Integrate Virtual Private Networks Using OpenVPN*. Packt Publishing, December 2009.

[26] Ala Rezmerita, Tangui Morlier, Vincent Neri, and Franck Cappello. Private virtual cluster:infrastructure and protocol for instant grids. In LNCS, editor, *In Proceedings of the International Euro-Par Conference on Parallel Processing (Euro-Par 2006)*, pages 393–404, Dresden, Germany, 2006.

[27] Openvpn. url: *http://openvpn.net/*.

Casa abierta al tiempo

UNIVERSIDAD AUTÓNOMA METROPOLITANA

DLML PARA UN AMBIENTE GRID

Para obtener el grado de

MAESTRO EN CIENCIAS

(Ciencias y Tecnologías de la Información)

PRESENTA:

Apolo H. Hernández Santos

Asesores

Dra. Graciela Román Alonso

Dr. Miguel Alfonso Castro García

Sinodales

Presidente: Dr. Ricardo Marcelín Jiménez [UAM-I]

Secretario: Dr. Santiago Domínguez Domínguez [CINVESTAV]

Vocal: Dr. Miguel Alfonso Castro García [UAM-I]

México D.F.

Febrero 2011