UNIVERSIDAD AUTÓNOMA METROPOLITANA

Procesamiento de grandes volúmenes de datos con los modelos de programación MapReduce y DLML

Idónea Comunicación de Resultados
para obtener el grado de
MAESTRO EN CIENCIAS
(Ciencias y Tecnologías de la Información)
Presenta:

Luis Alberto Pérez Suárez

Asesor

Dr. Miguel Alfonso Castro García

Jurado Calificador

Presidente: Dr. Santiago Domínguez Domínguez, CINVESTAV-IPN
Secretario: Dra. Graciela Román Alonso, UAM-Iztapalapa
Vocal: Dr. Miguel Alfonso Castro García, UAM-Iztapalapa

México-D.F.
Abril de 2013

Resumen

El procesamiento de grandes cantidades de datos otorga diversos beneficios, desde apoyar a la mercadotecnia para el diseño y la creación de nuevos productos, al campo financiero para la toma de decisiones e inversiones, así como al campo científico para el análisis de distintos fenómenos, entre otros. Para esto, existen diversos ambientes que permiten realizar el procesamiento de datos, entre ellos se encuentran los modelos de programación MapReduce y DLML (Data List Management Library). MapReduce se ha convertido en un modelo de programación muy popular entre la comunidad científica, debido a que permite procesar grandes volúmenes de datos con computadoras de uso común, además de ser tolerante a fallos. Por otro lado, DLML ha ganado interés debido al procesamiento que ofrece para diferentes tipos de aplicaciones, a la facilidad de su modelo de programación y a las distintas versiones que tiene para correr en diferentes arquitecturas (desde GPU's hasta Grids). Desafortunadamente, estas versiones se encuentran limitadas por el tamaño de las aplicaciones a procesar, las cuales no pueden rebasar la capacidad de la memoria RAM del sistema.

En esta tesis proponemos una extensión a DLML, a la cual llamamos DLML-IO. Esta extensión permite procesar grandes cantidades de datos que superan la cantidad total de memoria RAM con la que se cuenta, ya que emplea el manejo de archivos para esta tarea. Para evaluar esta extensión se hace una comparación entre el ambiente MapReduce y DLML-IO. En los experimentos realizados se utilizaron 4 aplicaciones, una donde MapReduce es popular por los resultados que obtiene, otra donde DLML presenta buenos resultados, y dos más, las cuales ya han sido implementadas anteriormente con MapReduce y DLML.

Los resultados experimentales muestran que con DLML-IO se consigue procesar cantidades de datos, de al menos el doble de la memoria RAM de la infraestructura con la que se cuenta, además de obtener mejores tiempos de ejecución que los de MapReduce en las 4 aplicaciones utilizadas.

Agradecimientos

Agradezco y dedico esta tesis a Dios y a mi familia por el apoyo que me han brindado ante cualquier situación; a mis padres Francisco Javier Pérez Gutiérrez y María Suárez Rangel, y a mis hermanos Francisco Javier Pérez Suárez y Cristian Iván Pérez Suárez.

Al Dr. Miguel Alfonso Castro García por haber aceptado ser mi asesor de tesis y por el tiempo que invirtió en orientarme y dirigirme para culminar este proyecto.

Al Dr. Santiago Domínguez Domínguez y a la Dra. Graciela Román Alonso, por haber aceptado ser mis sinodales.

Al Dr. Manuel Aguilar Cornejo por sus consejos y apoyo.

A mis amigos y compañeros de la maestría, especialmente a José Luis Quiroz Fabián, Adriana Espinosa Pérez y Luis Alarcón, quienes me apoyaron en todos los aspectos.

A la Universidad Autónoma Metropolitana por ser mí casa de estudios durante esta bonita experiencia.

Finalmente agradezco al Consejo Nacional de Ciencia y Tecnología (CONACyT) por la beca recibida.

Contenido

Lista de Figuras

Lista de Tablas

Capítulo 1
Introducción

La inminente necesidad del ser humano por realizar mayores cantidades de cálculos en menores tiempos, dio pie al surgimiento de las computadoras. Esta necesidad se ha hido incrementando, ya que aunque la mayoría de las aplicaciones en la industria, educación, etc., fueron diseñadas inicialmente para ser ejecutadas de manera independiente en computadoras con un solo procesador, fue inevitable la aparición de tareas que requerían tiempos de procesamiento prohibitivos, o que sobrepasaban el límite de almacenamiento. Ésto ha influenciado en gran medida el desarrollo de nuevas tecnologías para aumentar la capacidad de procesamiento y almacenamiento en las computadoras.

Una primera solución a las aplicaciones que requieren altas capacidades de memoria y procesamiento, es la adquisición de tecnología de punta, sin embargo, esta solución no siempre es factible debido a que algunas aplicaciones pueden demandar capacidades de cómputo que son poco probables encontrarlas en una sola computadora. Por otra parte, una segunda solución al problema es utilizar una súper computadora, donde generalmente se encuentran grandes capacidades de almacenamiento y procesamiento, como en la Sequoia de IBM, en la cual se encuentran 1,572,864 cores y 1,572,864 GB de RAM total [1]. Las supercomputadoras están compuestas por grandes conjuntos de computadoras interconectadas mediante una red de alta velocidad, y a través de la coordinación entre éstas se puede distribuir la carga de trabajo, y así reducir los tiempos de ejecución. Aunque optar por ésta idea suele ser muy buena, el uso de estos grandes equipos tiene un alto costo económico. Una tercera solución es contratar algunas entidades de cómputo con un proveedor de Cloud Computing como Amazon y pagar sólo por el tiempo de procesamiento utilizado, y en su caso por el espacio de almacenamiento solicitado [2]. El Cloud Computing se puede definir como un conjunto de servicios de red habilitados, proporcionando escalabilidad, calidad de servicio garantizada,

normalmente personalizada, infraestructura de cómputo de bajo costo sobre demanda, la cual se podría acceder en una forma simple y generalizada [3]. Aunque la tendencia por el Cloud Computing se ha incrementado en los últimos años, muchas instituciones siguen utilizando una cuarta solución, la cual consiste en interconectar un conjunto de computadoras de propósito general (a menudo recursos con los que ya se cuenta) mediante una red, que les permita coordinarse para ejecutar una tarea en común. A ésta última solución se le llama cluster, suele ser muy buena, y puede llegar a obtener resultados similares a los que se obtienen con una súper computadora.

1.1. Cómputo paralelo

La mayoría de las aplicaciones pueden ser divididas en trabajos, los cuáles pueden ser procesados de forma independiente por distintos procesadores, ya sea que se encuentren localizados en una misma computadora o dentro de un conjunto de computadoras con capacidad de comunicación entre éstas (como los clusters). A éste enfoque se le conoce como computó paralelo.

Una solución fiable a las aplicaciones que requieren altas capacidades de memoria y procesamiento, es el uso del cómputo paralelo en un cluster, no obstante, se muestran los principales factores que pueden afectar el tiempo de ejecución:

- ***Clusters heterogéneos***

  Un cluster heterogéneo está compuesto por computadoras con diferentes capacidades de almacenamiento y procesamiento, así como también diferentes sistemas operativos. Generalmente tener diferentes características de hardware y software ocasiona que algunos procesadores trabajen más rápido que otros, y como consecuencia, que tengan un tiempo de ocio más alto que los demás, desperdiciando así tiempo de cálculo que podría ser utilizado para reducir la carga global del sistema y reducir los tiempos de ejecución.

- ***Clusters no dedicados***

  Un cluster no dedicado comparte la infraestructura entre diversos usuarios, permitiendo así que los usuarios pueden ejecutar procesos en cualquier momento. Una consecuen-

cia de ésto, es que, si en algún momento se ejecuta una aplicación que requiere la disponibilidad de todos los cores en el cluster, y éste ya se encuentra ejecutando procesos de algún otro usuario, los tiempos de ejecución de las pruebas pueden aumentar o variar (dependiendo de la cantidad de procesos ajenos ejecutandose en el cluster), proporcionando resultados no confiables.

- ***Generación dinámica de datos***

  Es importante considerar la existencia de aplicaciones que generan datos de forma dinámica, ya que a diferencia de las aplicaciones donde la cantidad total de datos a procesar se conoce desde un principio, en este tipo de aplicaciones no es así. Por lo tanto, es difícil dividir la cantidad total de datos generados entre el número de cores existentes en el cluster, lo que provoca una desigualdad de carga de trabajo para algunos cores, y el aumento en los tiempos de ejecución para este tipo de aplicaciones.

En conclusión, se pueden encontrar diversos problemas por los que el desempeño de una aplicación puede verse afectado, sin embargo, existe una buena alternativa para mejorar el desempeño ante estos problemas, la cual se denomina balance de carga. El balance de carga tiene como objetivo equilibrar la carga encontrada en el sistema, nivelándola en cada uno de los cores del cluster.

1.2. Balance de carga

El balance de carga se ha convertido en una parte esencial del cómputo paralelo, ya que permite distribuir equitativamente la carga total de una aplicación (el total de los datos generados durante la ejecución de una aplicación) entre los cores que integran un cluster, a tiempo de ejecución. De esta manera, el balance de carga ayuda a reducir los tiempos de ejecución en las aplicaciones dinámicas y estáticas, e incrementa el rendimiento de los recursos en clusters heterogéneos y no dedicados.

Entre los modelos para la programación paralela se encuentran: el modelo de programación por memoria compartida, el modelo de programación por skeletons, el modelo de programación por paso de mensajes, entre otros (en [12] se realizó un análisis de estos modelos de programación). En esta tesis se utiliza el modelo de programación paralela por paso

de mensajes, en el cual los procesos pueden comunicarse con otros dentro del mismo cluster a través del envío y recepción de mensajes. Entre los principales ambientes que facilitan la programación paralela por paso de mensajes se encuentran OpenMPI [14] y LAM/MPI [15], sin embargo, en esta tesis se optó por utilizar LAM/MPI, debido a que una de las dos plataformas utilizadas en este proyecto fue desarrollada en éste ambiente, y aunque se ha utilizado también OpenMPI, los mejores resultados en las aplicaciones han sido obtenidos con LAM/MPI. La segunda plataforma utilizada en este proyecto hace uso de RPCs (Remote Procedure Calls) para la comunicación entre procesos [16].

El desarrollo de una aplicación paralela ya sea con LAM/MPI, OpenMPI, o RPCs, no incluye de forma automática el balance de carga a tiempo de ejecución entre los cores de un cluster. Cada uno de estos modelos requiere que el programador realice la implementación de los algoritmos para el balance de carga, lo cual puede llegar a ser sumamente complejo. No obstante, se pueden encontrar varias herramientas con un nivel más alto, que realizan el balance de carga de forma trasparente al usuario, permitiéndole concentrarse en la resolución de un problema y despreocuparse por los detalles de la algoritmia empleada para este trabajo. Entre las diversas herramientas que permiten facilitar el desarrollo de aplicaciones paralelas con balance de carga se encuentran: The Zoltan Parallel Data Services Toolkit [4], SAMBA [5], DDLB [6], ADLB [7], MapReduce [8] y DLML [12]. En [13] se encuentra un análisis de [4], [5], [6] y [12], no obstante, a continuación se muestra una breve descripción de las herramientas ADLB, MapReduce y DLML.

- **ADLB**

  ADLB (Asynchronous Dynamic Load Balancing) [7] es una biblioteca y un modelo de programación que permite desarrollar aplicaciones paralelas con un balance de carga automático y transparente al usuario. En ADLB el usuario debe dividir el trabajo total a procesar en unidades de trabajo, de manera que cada unidad sea ejecutada independientemente por un sólo proceso.

  Como se muestra en la Figura 1.1, en la arquitectura de ADLB se ve como un anillo, en donde se encuentran dos tipos de procesos, los *Servidores ADLB* y los *Procesos Aplicación*. Los *Servidores ADLB* se encargan de llevar a cabo el balance de carga para suministrar unidades de trabajo a los *Procesos Aplicación*, ya sea localmente por ellos

mismos, o remotamente de otros *Servidores ADLB*. Por otra parte, los *Procesos Aplicación* se encargan de llevar a cabo todas las operaciones relacionadas con la solución del problema.

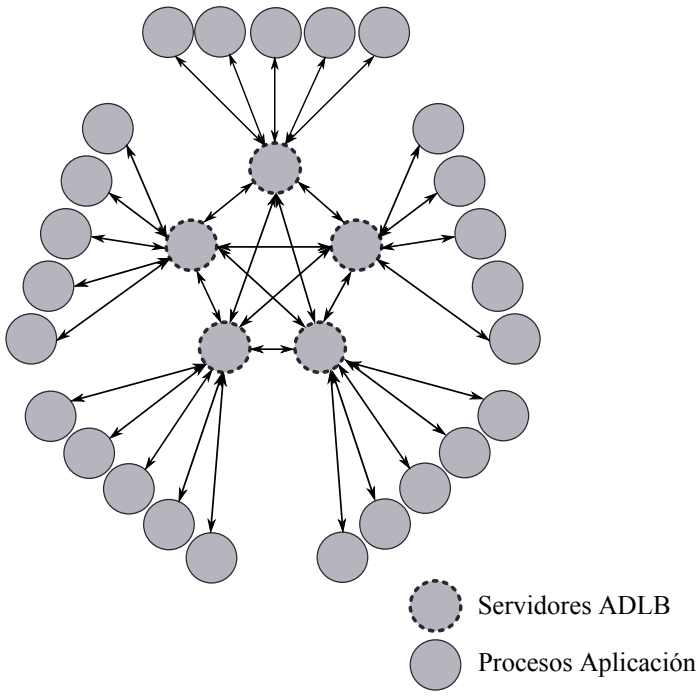

Figura 1.1: Arquitectura de ADLB.

ADLB proporciona un API (Application Programmer Interface) para el desarrollo de aplicaciones. Entre las principales funciones del API se encuentran: *ADLB_put*, *ADLB_Reserve* y *ADLB_Get_Reserved*. *ADLB_put* permite insertar unidades de trabajo en los *Servidores ADLB* locales. Por otro lado, cuando la aplicación desea obtener unidades de trabajo, primero especifica a través de *ADLB_Reserve*, una lista de tipos de trabajo a procesar, y posteriormente, si existe alguna unidad de trabajo que coincida con los tipos de la lista, ADLB retorna un identificador global, el cual es utilizado por *ADLB_Reserve* para obtener la unidad de trabajo.

- **MapReduce**

MapReduce es un modelo de programación y un framework patentado por Google [8] para el procesamiento de grandes volúmenes de datos. Una implementación open-source muy popular para MapReduce es Hadoop[23]. En MapReduce el usuario debe

especificar una función *map* que reciba como entrada un par del tipo llave/valor, para producir un conjunto de pares intermedios del tipo llave/valor, y una función *reduce* que reciba como entrada un conjunto de pares intermedios de tipo llave/valor, para generar un posible conjunto de pares más pequeño.

Como muestra la Figura 1.2, la ejecución general de un programa MapReduce involucra un conjunto de procesos Map destinados a realizar las operaciones *map*, un conjunto de procesos Reduce destinados a realizar las operaciones *reduce*, y una serie de pasos. Dentro de los pasos se encuentran: la lectura de los datos de entrada, tres fases, y la escritura de resultados. A continuación de describe brevemente cada una de las fases:

1. **Fase Map**. Cada proceso Map procesa pares del tipo llave/valor, por ejemplo k1/v1, y genera un conjunto de pares intermedios, por ejemplo, {k'1/v'1, k'2/v'3, k'1/v'2, k'3/v'5, k'2/v'4}.
2. **Fase Shuffle**. Esta fase toma el conjunto de pares intermedios y mezcla todos los valores asociados a una misma llave. Así que considerando el conjunto de pares intermedios del ejemplo anterior, esta fase se producirían los pares k'1/lista(v'1, v'2), k'2/lista(v'3, v'4), y k'3/lista(v'5).
3. **Fase Reduce**. Cada proceso Reduce procesa pares producidos por la fase Shuffle (generalmente procesan los valores contenidos en la lista de cada par).

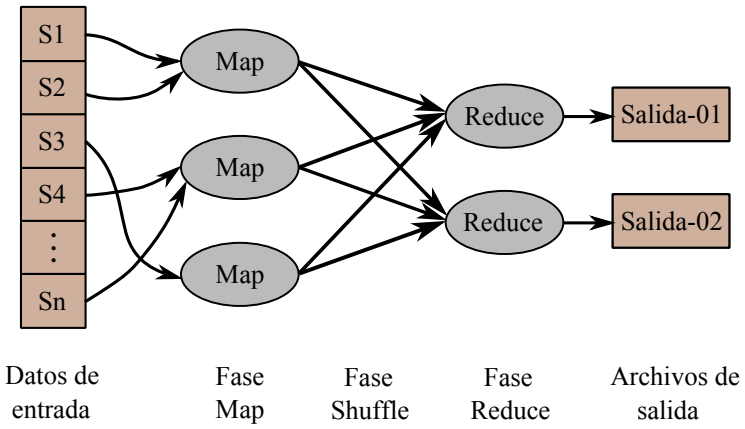

Figura 1.2: Ejecución general de un programa en MapReduce.

Es sustancial destacar que MapReduce es una plataforma tolerante a fallos, y se encarga de todo lo relacionado con la planificación de los procesos.

- **DLML**

  DLML (Data List Management Library) 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 [12]. En DLML los datos de una aplicación deben organizarse como elementos o items de lista, con el objetivo de que cada uno pueda procesarse de manera independiente a los demás.

  DLML ofrece por defecto un algoritmo de balance de carga que es conocido como subasta, y utiliza información global. Éste algoritmo consiste en que cuando un core se queda sin carga de trabajo, comienza a solicitar a todos los cores del sistema información acerca de su índice de carga, así que cuando recibe ésta información, la analiza y solicita al core con mayor carga de trabajo que le transfiera parte de su carga.

  Como se muestra en la Figura 1.3, la arquitectura de DLML contempla dos procesos por core: un proceso aplicación y un proceso DLML. El proceso aplicación es el encargado de realizar las operaciones propias del problema a resolver. Por otro lado se encuentra el proceso DLML, el cual se encarga de ejecutar el balance de carga.

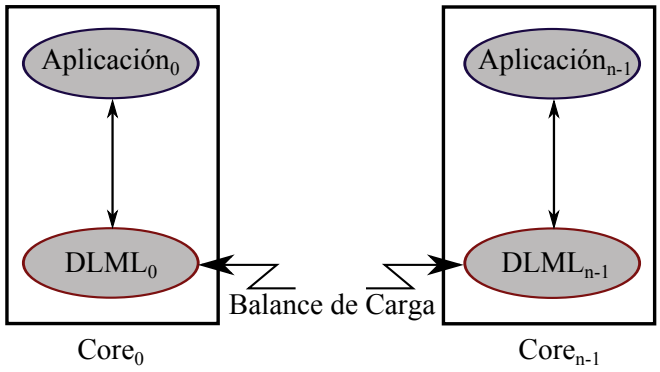

Figura 1.3: Arquitectura de DLML.

  La biblioteca DLML se encuentra desarrollada bajo el lenguaje C y hace uso de la biblioteca de paso de mensajes MPI. Asimismo, proporciona un conjunto de funciones para desarrollar aplicaciones con este modelo de programación.

Cada uno de los modelos de programación descritos anteriormente permiten facilitar el desarrollo de aplicaciones paralelas con balance de carga, sin embargo, hoy en día, debido

al incremento de la información digital, abunda la existencia de aplicaciones que procesan grandes volúmenes de datos (del orden de gigabytes o terabytes). Según la IDC (International Data Corporation) [25], tan solo la cantidad de información digital en 2010 estuvo cerca de los 1,203 exabytes, y más aún se espera que para el 2015 esta cantidad alcance los 7,910 exabytes, por lo tanto, la capacidad de procesar grandes cantidades de información se ha convertido en un requerimiento fundamental para muchos modelos de programación.

Esta tesis se centra en la comparación de MapReduce y DLML. Hemos excluido a ADLB principalmente por ser similar a DLML. Cabe mencionar que tanto ADLB como DLML se encuentran limitados por la capacidad de la memoria RAM de los nodos empleados en el sistema, por lo tanto, ambos modelos de programación permiten procesar grandes cantidades de datos, siempre y cuando esta labor no sobrepase la capacidad total de memoria RAM de los nodos con los que se cuenta. No obstante, en el caso de que una aplicación sobrepase la capacidad total de memoria disponible, colapsará, ya sea que se encuentre implementada con ADLB o DLML. Finalmente, aunque DLML presenta la limitación en memoria RAM, se tiene una propuesta que le permite procesar cantidades mayores a las capacidades de memoria RAM con las que se cuenta.

Otros motivos por los cuales MapReduce y DLML han sido elegidos como plataformas de experimentación son los siguientes:

- Código fuente y documentación disponible para ambos modelos de programación.
- Diversas fuentes de ayuda para MapReduce, y soporte técnico por parte de los autores para DLML.
- Ambas herramientas incluyen balance de carga, con el cual comparar resultados.

1.3. DLML con lecturas y escrituras a archivos

Como se mencionó anteriormente, DLML se encuentra limitado por la capacidad de memoria RAM, lo cual hace muy difícil procesar grandes cantidades de datos que la sobrepasen, por lo tanto es necesario hacer una extensión a esta biblioteca para añadir dicha característica.

El objetivo principal de este trabajo es la extensión de la biblioteca DLML para hacer manejo de archivos, el cual permitirá procesar grandes volúmenes de datos. Originalmente

en DLML se realizaba una recolección de resultados parciales, sin embargo, en este caso, dependiendo de las aplicaciones, los resultados parciales serán almacenados en archivos de salida independientes.

El resto de la tesis se organiza de la siguiente forma: en el siguiente capítulo se presenta MapReduce. En el Capítulo 3 se presenta DLML. En el Capítulo 4 se presenta la extensión para DLML (DLML-IO). En el Capítulo 5 se presenta un análisis de los resultados obtenidos de los modelos de programación MapReduce y DLML-IO. Finalmente en el Capítulo 6, se presentan las conclusiones y el trabajo futuro.

Capítulo 2
MapReduce

2.1. Introducción

MapReduce [8] es un modelo de programación y una plataforma patentada por Google para el procesamiento de grandes conjuntos de datos. El modelo de programación MapReduce presentado en 2004 está inspirado en las funciones *map* y *reduce* utilizadas en la programación funcional. Un lenguaje para la programación funcional es Lisp [26], el cual toma como entrada una función y una secuencia de elementos, posteriormente aplica la función a cada uno de los elementos, y finalmente combina todas las salidas en una operación. La función que se le aplica a cada uno de los elementos es conocida como la función *map*, y la función que combina las salidas es conocida como función *reduce*. La ventaja del modelo de programación MapReduce sobre Lisp u otros lenguajes de programación funcional es que permite el procesamiento distribuido de las funciones *map* y *reduce*. Todas las operaciones *map* pueden ejecutarse en paralelo, siempre y cuando sean independientes unas de las otras, y de igual forma las operaciones *reduce* también pueden ejecutarse en paralelo.

MapReduce permite desarrollar aplicaciones que procesen grandes conjuntos de datos. En MapReduce los usuarios especifican una función *map* que procesa pares de tipo llave/valor, para generar un conjunto de pares intermedios del tipo llave/valor, por cada par de entrada. Otra función que debe ser especificada por el usuario es la función *reduce*, la cual procesa conjuntos de pares intermedios para generar posiblemente conjuntos más pequeños. Los programas escritos en este estilo funcional son automáticamente paralelizados y ejecutados sobre grandes clusters de máquinas de propósito general [8]. Cabe mencionar que esta plataforma se encarga de la planificación de los trabajos, de su monitoreo y re-ejecución de aquellos que resulten fallidos.

Existen implementaciones de MapReduce en C++, Java, Python, entre otros lenguajes de programación. Una implementación open source muy pupular en Java utilizada en esta tesis es proporcionada por Hadoop [23], la cual posee el mismo enfoque del MapReduce de Google, y por consecuencia, la mayor parte de sus características.

2.2. Modelo de programación

Este modelo de programación está caracterizado por las funciones *map* y *reduce* definidas por el usuario. Para mostrar como funciona este modelo de programación se considera como ejemplo una aplicación muy popular en MapReduce, la cual se denomina "Conteo de Palabras" (o WordCount), y consiste en contar el número de ocurrencias de cada palabra que se encuentra en un archivo, o en un conjunto de archivos. En la Figura 2.1 se muestran los pseudocódigos de las funciones *map* y *reduce* empleadas para esta aplicación.

```
1 map( key, value ) {
2   inter_values_list=break(value);
3   for each element in inter_values_list{
4      EmitIntermediatePair(element, 1);
5   }
6 }//end-map
```

a)

```
1 reduce( key2, list_of_values ) {
2   occurrence = 0;
3   for each element in list_of_values{
4      occurrence = occurrence + element;
5   }
6   EmitIntermediatePair(key2, occurrence);
7 }//end-reduce
```

b)

Figura 2.1: (a) Pseudocódigo de la función map; (b) Pseudocódigo de la función reduce.

Como describe el pseudocódigo de la Figura 2.1a, la función *map* toma como entrada un par del tipo llave/valor (línea 1a), donde la llave es el desplazamiento de archivo donde comienza el renglón a procesar, y el valor es la cadena completa de caracteres encontrada en el renglón. La función *map* se encarga de dividir la cadena de caracteres en palabras (línea 2a), y genera un par intermedio por cada palabra en el renglón, donde cada par establece la palabra obtenida como la llave, y un número de ocurrencia "1" como el valor, generado así un conjunto de pares intermedios del tipo llave/valor (líneas 3a-5a) por cada par de entrada.

Una vez que se aplica la función map a cada uno de los pares de entrada, la plataforma MapReduce se encarga de mezclar todos los valores asociados a una misma llave, por lo que la función *reduce* (Figura 2.1b) toma como entrada un conjunto de pares intermedios de tipo llave/valor, donde la llave de cada par es una palabra, y el valor es una lista de valores

asociados a la palabra (línea 1b). La función reduce se encarga de sumar todos los elementos en la lista de valores de cada par (líneas 3b-5b), y una vez que termina la suma emite un par del tipo llave/valor, donde la llave del par es la misma palabra del par de entrada, y el valor es el número de ocurrencias de la palabra (línea 6b). Cabe mencionar que el conjunto de pares intermedios generado por la operación *reduce* es en este caso, almacenado en un solo archivo y puede ser más pequeño que el conjunto de pares intermedios de entrada.

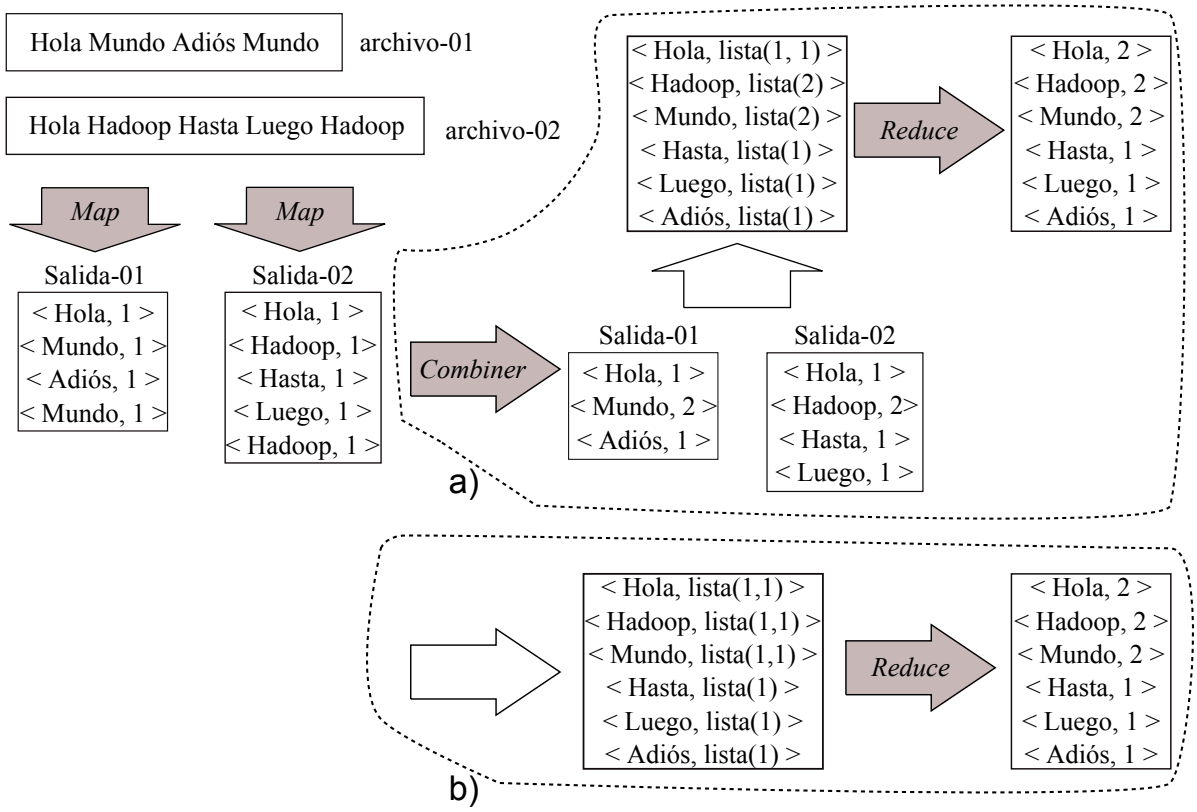

Figura 2.2: Conteo de cada palabra en un conjunto de documentos utilizando el modelo de programación MapReduce.

La Figura 2.2 muestra el flujo de la ejecución que toma el "Conteo de Palabras", con dos procesos que realizan la operación Map, un proceso que realiza la operación Reduce, y alternativamente con el uso de una operación Combiner. Como muestra la Figura 2.2a, una vez que terminan de producirse las salidas de las operaciones *map*, se les puede aplicar una operación *combiner*, la cual les aplica la función *reduce* de forma local, haciendo que probablemente se reduzca el conjunto de pares intermedios, ya que los pares con la misma

llave se agrupan en uno solo, y se incrementa sólo el valor del número de ocurrencias del par. Emplear una operación *combiner* no afecta el resultado de la aplicación, y una vez que se aplica esta operación, se le transfiere el conjunto de pares intermedios resultante al proceso destinado a realizar la operación *reduce* (la flecha blanca resalta la transferencia), continuando así con el proceso descrito anteriormente.

La Figura 2.2b muestra el flujo de la aplicación sin el uso de la operación *combiner*. Como puede percibirse, la cantidad de información transferida aumenta ligeramente (denotado por la flecha blanca), sin embargo, el beneficio que puede aportar la operación *combiner* puede depender de la naturaleza de cada aplicación, por lo tanto, si ésta operación no pudiera otorgar algún beneficio, el flujo tomaría la ejecución del inciso b).

2.3. Arquitectura

MapReduce emplea una arquitectura maestro/esclavo para el cómputo distribuido [17] y para el almacenamiento distribuido. El sistema de almacenamiento distribuido en el caso particular de Hadoop es HDFS (Hadoop File System), y antes de mostrar la arquitectura de la ejecución general de un programa en MapReduce se presentan las siguientes definiciones:

- Procesos Map: Son los procesos destinados a ejecutar las operaciones *map*, y son también conocidos como mappers.

- Procesos Reduce: Son los procesos destinados a ejecutar las operaciones *reduce*, y son también conocidos como reducers.

- Tareas: Hadoop divide el trabajo en tareas, las cuales se clasifican en tareas *map* y tareas *reduce*. Una tarea *map* es un conjunto de pares del tipo llave/valor que tiene que ser procesado por un mapper. Por otro lado, una tarea *reduce* es un conjunto de pares intermedios del tipo llave/valor que tiene que ser procesado por un reducer.

- JobTracker: Es uno de los procesos que se consideran como "maestros", existe solo uno en el cluster, y como se menciona en [17], funciona como enlace entre la aplicación y Hadoop, ya que se encarga de la asignación de las tareas, de monitorearlas, y de re-ejecutar aquellas que resulten fallidas.

- TaskTracker: Es uno de los procesos que se considera como "esclavos", se encuentra uno por cada nodo del cluster, y se encarga de ejecutar de forma local las tareas del tipo *map* o *reduce*, que le son asignadas por el JobTracker (ver Figura 2.3). El TaskTracker se comunica continuamente con el JobTracker para proporcionale información acerca de las tareas que se le asignaron, entre otras cosas.

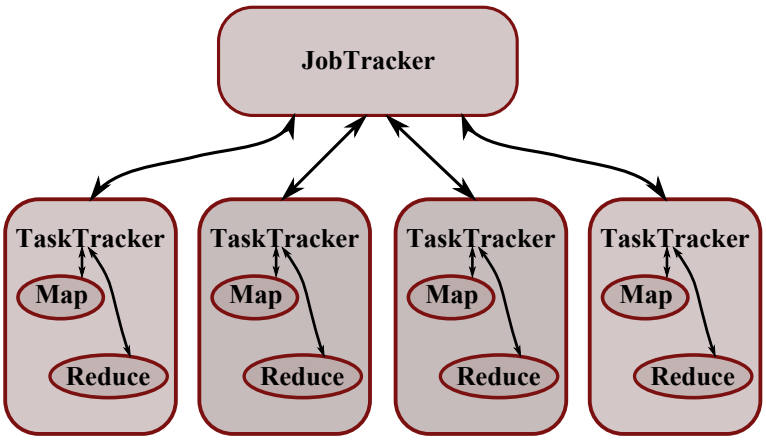

Figura 2.3: Interacción entre el JobTracker y los TaskTracker.

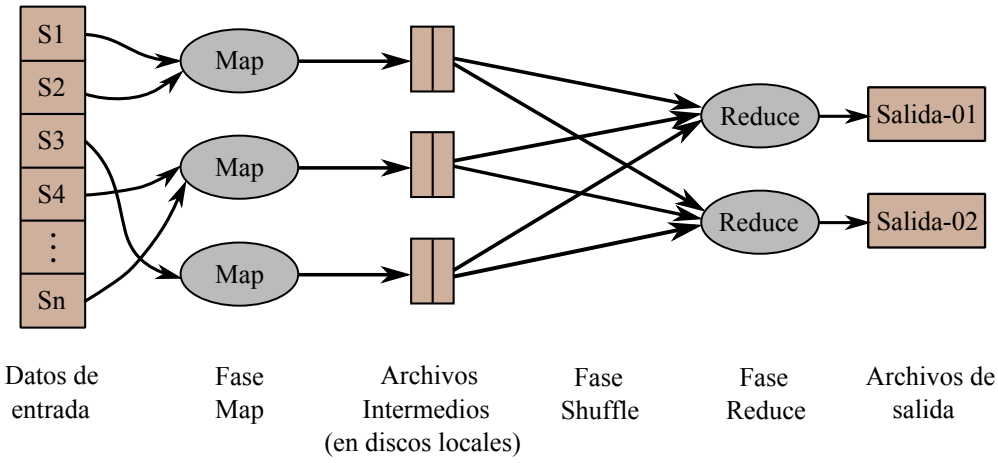

Figura 2.4: Flujo de una ejecución general de un programa en MapReduce.

A continuación se describe el flujo de una ejecución general de un programa en MapReduce. Para ésto se retoma la Figura 1.2, y se incluye una parte adicional en la imagen con el objetivo de detallar más la explicación (ver Figura 2.4):

Lectura de los archivos de entrada

Los archivos de entrada son automáticamente divididos en segmentos por la plataforma MapReduce, para posteriormente distribuirlos entre los mappers que se encuentren con un estado ocioso.

Fase Map

En esta fase los mappers se encargan de aplicar la función *map* definida por el usuario, a las tareas de este mismo tipo, las cuales se encuentran constituidas por pares del tipo llave/valor. La salidas de cada uno de los procesos Map son escritas en el disco duro local del nodo en que residen, sin embargo, el proceso es un poco más complejo. Cada proceso Map tiene un buffer de memoria circular en el que escribe los pares de salida. El buffer es por defecto de 100MB, puede ser ajustado, y emplea además un umbral (al 80 % del buffer), con el objetivo que cuando éste se sobrepase, un hilo en segundo plano comience a verter el contenido del buffer al disco duro local. Las salidas generadas por el procesamiento de una tarea *map* continúan siendo escritas en el buffer mientras el vertido se realiza, pero si el buffer se queda sin memoria disponible, el proceso Map se bloquea hasta que el vertido se complete. Cada vez que el buffer alcanza el umbral, un nuevo archivo de vertido es creado, por lo que después de que el proceso Map ha escrito su último registro de salida pueden haber varios archivos de vertido. Finalmente, antes de que finalice el procesamiento de la tarea *map*, los archivos de vertido son mezclados en un solo archivo de salida, el cual se encuentra ordenado con respecto a las llaves que tienen que procesar los diferentes procesos *reduce*.

Fase Shuffle

En esta fase se transfieren las salidas de los procesos *map* a los procesos *reduce* para su posterior procesamiento. Como se mencionó anteriormente, las salidas generadas por el procesamiento de una tarea *map* son almacenadas en un archivo de salida particionado y ordenado, no obstante, las particiones del archivo de salida son transferidas a los correspondientes procesos Reduce a través de HTTP (Hypertext Transfer Protocol).

El archivo de salida particionado y ordenado situado en el TaskTracker que ejecutó la tarea *map* es necesario para los TaskTrackers que tienen que ejecutar la tarea *reduce*, ya que tienen que obtener la partición del archivo que les fue asignada. En general, una tarea *reduce*

necesita la salidas generadas por la fase *map* para su partición de varias salidas *map* [16]. Es importante señalar que el procesamiento de las tareas *map* puede finalizar en diferentes tiempos, por lo que las salidas generadas por el procesamiento de las tareas *map* son copiadas a las correspondientes tareas *reduce* conforme se van completando.

Fase Reduce

En esta fase los reducers se encargan de aplicar la función *reduce* definida por el usuario, a las tareas de este mismo tipo, las cuales se encuentran constituidas por pares del tipo llave/valor, donde el valor es una lista de valores asociados a la llave del par.

Como se describió en la fase shuffle, antes de que la fase *reduce* pueda comenzar se requiere que las salidas generadas por la fase *map* hayan sido copiadas a las correspondientes tareas *reduce*. Sin embargo, es importante resaltar que cuando una tarea *reduce* obtiene sus respectivas salidas *map*, antes de que pueda ser procesada, se mueve dentro de una "fase de ordenamiento", la cual mezcla las salidas *map* para posteriormente poder iniciar con la fase *reduce*.

Escritura de los resultados

La salida de la fase *reduce* es escrita directamente sobre el sistema de archivos HDFS, en archivos de salida independientes, y cabe señalar que el número de archivos de salida depende del número de reducers, por lo tanto se va a obtener el mismo número de salidas como reducers se ejecuten en la aplicación. A continuación se describen más detalles del sistema de archivos HDFS.

2.4. HDFS

HDFS es un sistema de archivos distribuido diseñado para almacenar grandes cantidades de datos (del orden de gigabytes, terabytes, e incluso petabytes), generalmente sobre grandes clusters conformados por computadoras de uso común. HDFS es usado en grandes empresas como Yahoo!, Facebook, y Twitter [27]. HDFS emplea también una arquitectura maestro/esclavo para el almacenamiento distribuido, e incluye los siguientes componentes:

- NameNode: Es considerado como uno de los procesos "maestros", se encuentra sólo uno

en el cluster, y es el encargado de establecer comunicación con los procesos DataNode para asignarles tareas de E/S de bajo nivel. El NameNode se encarga también de llevar un registro de como se distribuyen los bloques que constituyen los archivos (almacenados en el sistema de archivos) entre los DataNodes, es decir, se encarga de mantener los metadatos de los archivos almacenados en HDFS.

- DataNode: Es considerado como uno de los procesos "esclavos", se encuentra uno por cada nodo en el cluster, y es el encargado de realizar lecturas o escrituras de bloques en HDFS. Los procesos DataNode se comunican constantemente con el NameNode para proporcionarle información relacionada con los bloques que almacenan de forma local, información sobre sus cambios locales, así como también para recibir instrucciones para crear, eliminar, o mover bloques del disco local. Los procesos DataNode pueden comunicarse entre ellos para replicar bloques cuando se requiere redundancia. La figura 2.5 muestra la interacción entre el NameNode y los DataNodes, al almacenar 2 archivos en HDFS.

- Secondary NameNode (SNN): Es un proceso asistente para el monitoreo del cluster con HDFS, se encuentra solo uno en el cluster, y se comunica con el NameNode para registrar información de los metadatos periódicamente. La información registrada ayuda a minimizar la perdida de datos cuando falla el NameNode.

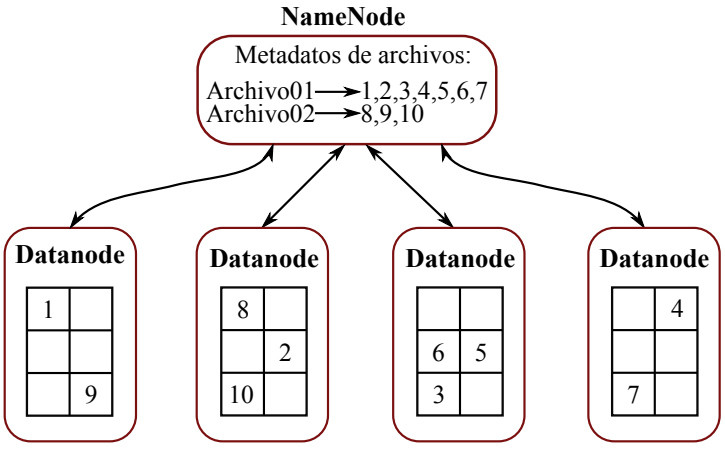

Figura 2.5: Iteracción entre el NameNode y los DataNodes al almacenar dos archivos en HDFS.

Cuando se crea un archivo en HDFS es dividido en bloques, los cuales se distribuyen entre los DataNodes. La figura 2.5 muestra la descomposición en bloques de dos archivos almacenados en HDFS con un valor de replica igual a uno. Con este valor de replica los respectivos bloques de cada archivo no se repiten.

HDFS usa por defecto bloques de 64MB, sin embargo este tamaño puede configurarse. HDFS usa una política de distribución aleatoria para mapear bloques de un archivo en diferentes DataNodes, de manera que cuando se crea un bloque, el NameNode se encarga de seleccionar aleatoriamente un DataNode para almacenarlo [28].

Hadoop intenta sacar ventaja de la distribución de los bloques en el sistema de archivos para la asignación de las tareas *map* que operan sobre los bloques de un archivo, ya que intenta asignar cada tarea *map* a un bloque almacenado en el mismo nodo, generando así un ahorro en el ancho de banda de la red. Sin embargo, en caso de que la asignación no pueda llevarse a cabo, la tarea *map* se asigna a un nodo remoto.

Finalemente, un cliente accede al sistema de archivos en nombre del usuario mediante la comunicación con el NameNode [16]. Un cliente primero se comunica con el NameNode para saber cuales son los DataNodes que debe contactar para acceder a los primeros bloques de un archivo, o en su caso, para realizar escrituras a bloques. Es importante resaltar que los clientes guardan la información otorgada por el NameNode para comunicarse directamente con los DataNodes.

Debido a la popularidad de MapReduce en el campo científico, así como en las instituciones publicas y privadas, surgieron diversas implementaciones basadas en esta plataforma que se enfocan en mejorar su desempeño. En la siguiente sección se presentan algunas.

2.5. Trabajo relacionado

Entre las numerosas implementaciones basadas en MapReduce se encuentran: Pipelined-MapReduce [9], CGL-MapReduce [10], y MapReduce con paso de mensajes [11]. A continuación se describen las principales características de cada uno de estas implementaciones.

Pipelined-MapReduce

Pipelined-MapReduce (PMR) es un modelo de programación paralela MapReduce mejorado [9]. PMR se enfoca en mejorar la transferencia de los pares intermedios (generados

por los procesos *map*) en la plataforma MapReduce de Hadoop, a través del uso de un diseño sencillo de pipeline. Esta modificación permite reducir los tiempos de ejecución de las aplicaciónes, y la tasa de utilización del sistema, conservando el modelo de programación, la interfaz de programación y la tolerancia a fallos.

En PMR los procesos *map* envían directamente sus pares intermedios de salida a los procesos *reduce*. Para llevar a cabo ésto, cada tarea *reduce* contacta a cada tarea *map*, y abre un socket TCP, el cual será usado para canalizar la salida de la función *map*. En cuanto un proceso *map* produce un registro de salida, determina a cual proceso *reduce* debe enviarse, e inmediatamente lo envía a través del socket apropiado. Por otro lado, cuando las tareas *reduce* reciben sus respectivos datos de entrada, y se les notifica que todas las tareas *map* han finalizado se encargan de mezclar todos los datos recibidos. Finalmente se realiza el procesamiento de las tareas *reduce* y los datos de salida son almacenados en HDFS, al igual que en la distribución MapReduce de Hadoop.

La transferencia de datos intermedios es una parte de la plataforma MapReduce donde se pueden encontrar diversos trabajos enfocados a mejorarla. Una implementación basada en MapReduce que emplea el streaming para todas las comunicaciónes es CGL-MapReduce.

CGL-MapReduce

CGL-MapReduce es un novedoso entorno de ejecución MapReduce que usa streaming para todas las comunicaciones, las cuales eliminan el overhead asociado con la comunicación por un sistema de archivos [10]. El streaming en esta versión permite enviar directamente los datos producidos por los procesos *map* a los procesos *reduce*. Una ventaja de CGL-MapReduce sobre MapReduce es que permite realizar aplicaciones donde se ejecuta un sola fase *map* y una sola fase *reduce*, así como también programas donde se pueden ejecutar estas fases iterativamente durante una sola ejecución (es decir, múltiples veces).

CGL-MapReduce utiliza el sistema de archivos NFS (Network File System) [31], y delega la tolerancia a fallos de las comunicaciones a la red de difusión de contenido basada en streaming que es también desarrollada por los autores.

Tanto la implementación de MapReduce proporcionada por hadoop, Pipelined-MapReduce, así como CGL-MapReduce, se encuentran desarrolladas en Java, sin embargo existen más implementaciones basadas en MapReduce que no precisamente se desarrollaron bajo este

lenguaje de programación. A continuación se describe una implementación de MapReduce, construida con MPI (Message Passing Interface).

MapReduce con paso de mensajes

MapReduce con paso de mensajes es una implementación basada en la plataforma MapReduce que hace uso de la biblioteca MPI para transferir los datos intermedios. El objetivo de esta implementación al utilizar este tipo de comunicación es explotar la memoria del ancho de banda, y así obtener una comunicación más eficiente entre procesos *map* y *reduce* [11]. El desempeño de esta versión sobrepasa el de MapReduce de hadoop cuando la cantidad a procesar es pequeña.

A comparación de la plataforma MapReduce proporcionada por hadoop, donde el flujo de una ejecución general de un programa sigue 3 fases, se hace uso de HDFS para la lectura de los archivos de entrada y escritura de los datos de salida, y se tiene tolerancia a fallos, en esta implementación se encuentran 6 fases, se utiliza NFS como sistema de archivos para la lectura de los archivos de entrada y la escritura de los datos de salida, y no se tiene tolerancia a fallos.

En las secciones 2.2 y 2.3 se describió el modelo de programación MapReduce (en Hadoop) y la arquitectura de una ejecución general respectivamente. En la sección 2.5, se presentaron CGL-MapReduce, Pipelined-MapReduce y MapReduce con paso de mensajes. Estas tres últimas implementaciones están inspiradas en el modelo de programación MapReduce, aunque no incluyen en la totalidad todas las características de la arquitectura original.

Otro modelo de programación mencionado anteriormente es DLML. DLML presenta buenos resultados en diversas aplicaciones, no obstante se encuentra limitado por la cantidad de memoria RAM de la cual se hace uso.

Capítulo 3
DLML

3.1. Introducción

DLML (Data List Management Library) es una biblioteca de programación que facilita el desarrollo de aplicaciones paralelas a través del uso de listas de datos, las cuales permiten procesar los datos de una aplicación de forma independiente. Durante la ejecución de una aplicación, DLML se encarga del particionamiento de la lista de datos, de su distribución entre los cores de un cluster y del balance de carga entre éstos. DLML proporciona funciones para la inserción y eliminación de elementos en la lista de datos de la aplicación. Estas funciones se denominan *DLML_Insert* y *DLML_Get* respectivamente.

Originalmente DLML incluye un algoritmo de balance de carga que utiliza una política de subasta global, la cual toma en cuenta todos los cores del cluster para el balance de carga. Cabe mencionar, que actualmente se encuentran 3 versiones más de DLML. La segunda versión se denomina DLML con manejo de información parcial, y a diferencia de la primera versión, cada core se encarga de realizar el balance de carga entre un subconjunto de los cores del cluster. La tercera versión, llamada MC-DLML, está enfocada en aprovechar el paralelismo de las nuevas máquinas multicore en clusters mediante el uso de memoria compartida, hilos de procesamiento y paso de mensajes. Finalmente, la cuarta versión de DLML, nombrada DLML-Grid, se encarga de trasladar DLML a un entorno Grid (el cual se encuentra conformado por diversos clusters), para la utilización de los recursos existentes, realizando balance de carga entre los clusters del Grid.

En este capítulo se describe como funciona el modelo de programación de DLML y su arquitectura, así como también se presenta un resumen de las versiones mencionadas anteriormente.

3.2. Modelo de programación

Los datos a procesar por una aplicación pueden ser vistos como el conjunto de tareas a procesar, las cuales pueden organizarse como elementos de una lista. La programación con DLML se resume a la inserción de los elementos en una lista, obtener uno a uno los elementos para procesarlos (mientras existan elementos) y finalmente mostrar resultados. A continuación se muestra una serie de pasos para desarrollar aplicaciones con DLML[12]:

1. Distinguir los datos de la aplicación y determinar la forma de organizarlos en listas.
2. Definir el código o procedimiento de inicialización de cada elemento de la lista.
3. Definir el código o procedimiento para procesar cada elemento de la lista. Este código en general estará dentro de un loop (bucle) e incluye un get().
4. Imprimir resultados, cerrar archivos y finalizar procesos, entre otros.

Para ejemplificar el uso de los pasos anteriores, se describe una aplicación sencilla en la que se realiza la suma de 100 números aleatorios. Los datos a procesar en la aplicación son los 100 números aleatorios, y una forma de organizarlos en una lista es introducir cada uno de ellos como un elemento de la lista (paso 1). Como se muestra en la Figura 3.1a, la estructura de cada elemento en la lista contiene un número y un enlace para el siguiente elemento en la lista. Continuando con el procedimiento se tiene el código de inicialización (ver Figura 3.1b), en el cual inicialmente se reserva memoria para cada elemento a insertar en la lista, después se le asigna un número aleatorio y finalmente es insertado mediante la función *DLML_Insert* (paso 2). Cuando termina la inicialización, se procede a obtener cada elemento con la función *DLML_Get*, y posteriormente procesarlo (paso 3). En este caso, el procesamiento corresponde a ir sumando una variable parcial con cada elemento obtenido (ver Figura 3.1c).

Concluyendo con la serie de pasos (paso 4), se imprime el resultado de la suma (ver Figura 3.1d), no obstante, ésto puede llegar a ser opcional dependiendo de las características de cada aplicación. La Figura 3.2 ilustra el flujo de este procedimiento. Este flujo es percibido en el desarrollo de la mayoría de las aplicaciones con DLML. Como se mencionó anteriormente, DLML se encarga automáticamente de los detalles del balance de carga de manera transparente al usuario.

```
typedef struct DLML_item item;
struct DLML_item{
   int  numero;
   item *siguiente;
};
```

a)

```
for(i=0; i<100; i++){
   elemento=new(item);
   elemento.numero=random();
   DLML_Insert(&Lista,elemento);
}
```

b)

```
while(DLML_Get(&Lista,&elemento)){
   suma_parcial=suma_parcial + elemento.numero;
}
```

c)

```
printf(Suma total = suma_parcial);
```

d)

Figura 3.1: a) Código de la estructura de un elemento en la lista; b) Código de inicialización; c) Código para la obtención y procesamiento de cada elemento en la lista; d) Código para la impresión de resultados.

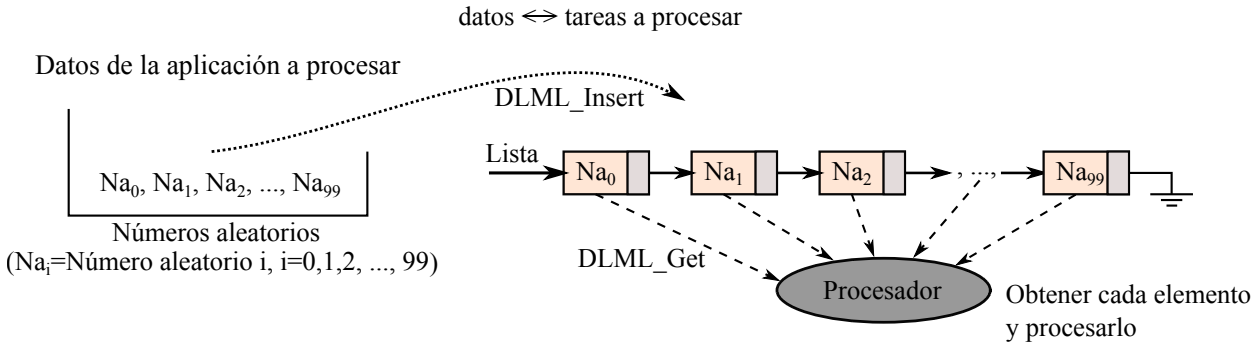

Figura 3.2: Flujo del procedimiento de la suma de 100 números aleatorios en DLML.

DLML particiona automáticamente la lista de datos y la distribuye entre los cores del clúster, de manera que cada uno de los cores obtiene un segmento de lista para procesar. Cuando un core consume los elementos de su lista local o se encuentra sin carga de trabajo, ejecuta un algoritmo de balance de carga con el objetivo de localizar al core con la mayor carga de trabajo, es decir, con la longitud de lista más grande dentro del conjunto de cores del cluster. Cuando se localiza al core con la mayor carga, se le solicita que proporcione un segmento de su lista, por lo que éste la particiona y le proporciona una sublista al core que ejecutó el algoritmo de balance de carga. De esta manera se logra un balance de carga entre los cores de un cluster, proporcionando así, datos a procesar a aquellos cores sin carga de

trabajo.

La Figura 3.3 muestra un ejemplo del balance de carga para la aplicación anterior donde se encuentran cuatro cores para el procesamiento de los datos. Como se muestra en el inciso "a", la inserción de los elementos se lleva a cabo por el core 0, por lo que la carga de trabajo se encuentra inicialmente en éste. Los demás cores en el sistema se encuentran inicialmente sin carga de trabajo, por lo que proceden a ejecutar un algoritmo de balance de carga. Cuando el core 0 recibe las peticiones de los demás, divide su lista de datos entre el número de peticiones más el mismo (entre 4), y le transfiere un segmento de lista a cada core. Finalmente, como resultado del balance de carga cada core recibiría 25 items a procesar (ver Figura 3.3b).

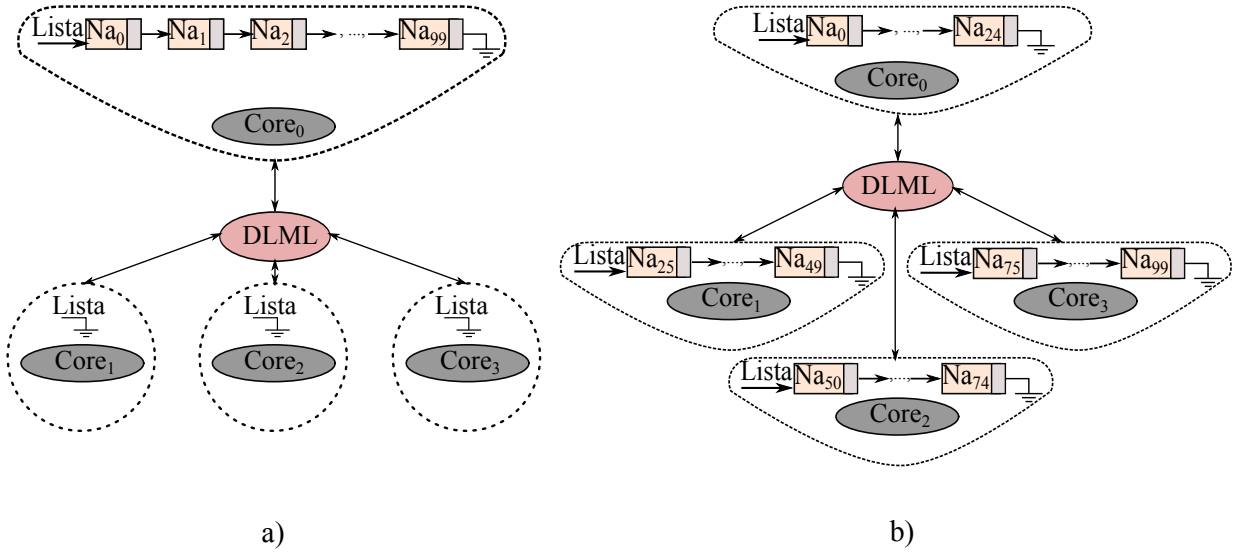

Figura 3.3: Distribución de carga entre los cores de un cluster para la suma de 100 números aleatorios.

3.3. Arquitectura

La arquitectura de DLML está formada por dos procesos, el proceso *Aplicación* y el proceso *DLML*, los cuales se ejecutan simultáneamente en cada core del cluster (ver Figura 3.4). El proceso *Aplicación* es el encargado de ejecutar el código de la aplicación para el procesamiento de los datos, e invoca las operaciones básicas de lista, ya sea para insertar un nuevo elemento en lista, para obtener un item de lista, o para obtener una nueva lista de elementos para procesarla. Por otra parte, el proceso *DLML* es el responsable de entablar la

comunicación con los demás procesos *DLML* cuando se ejecuta el protocolo de balance de carga.

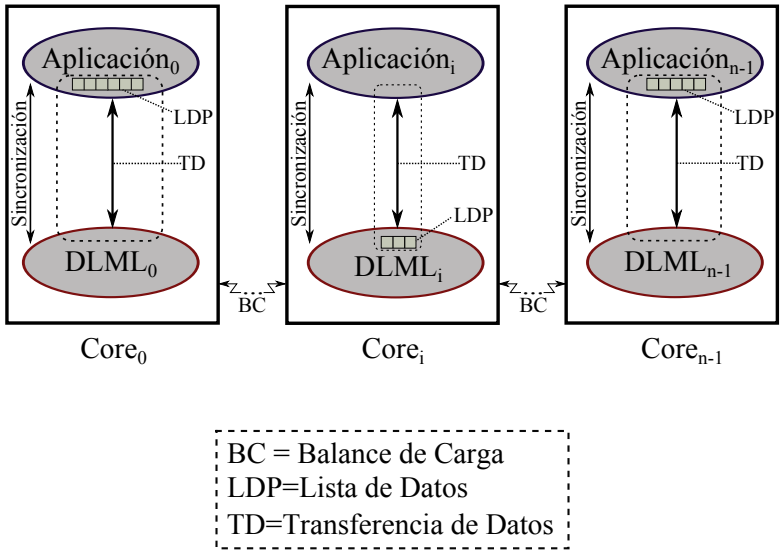

Figura 3.4: Arquitectura de DLML.

La *Aplicación* se comunica con el *DLML* con el fin de obtener información relacionada con la transferencia de datos. Cuando la lista en alguno de los cores se vacía, la *Aplicación* envía un mensaje a su *DLML* local con el fin de obtener más datos remotos, y luego permanece bloqueada hasta que recibe un mensaje del *DLML*. El *DLML* local se comunica con los *DLMLs* remotos mediante la ejecución de un algoritmo de balance de carga (que hace uso del envío y recepción de mensajes) para solicitarle la transferencia de datos al core con la mayor carga de trabajo. El mensaje de respuesta del *DLML* local para la *Aplicación* puede indicar la transferencia de un segmento de lista a procesar (en caso de que se localice al core con mayor carga), o la terminación de la ejecución (cuando no se encuentran más datos a procesar en el sistema).

El algoritmo de balance de carga utilizado por DLML, originalmente emplea una política de balance de carga que define todo lo relacionado con la distribución de la carga de trabajo entre los cores de un cluster (los elementos a transferir, el momento en que debe realizarse la transferencia, quién realizará la transferencia, quién recibirá la transferencia, etc). Esta política es conocida como *"Política de subasta con manejo de información global"*, y actualmente, entre las diferentes versiones de DLML, se encuentra una que aplica una política de

balance de carga diferente.

3.4. Política de balance de carga en DLML

Para efectuar el balance de carga se requieren políticas que definan como se llevará a cabo la distribución de carga entre los cores del cluster. Originalmente DLML utiliza una política de subasta con manejo de infomación global, la cual se describe a continuación.

Subasta con manejo de información global

La subasta con manejo de información global involucra a todos los cores en el cluster para la ejecución del balance de carga. Su objetivo es subastar el poder de cómputo de un core sin carga de trabajo al resto de los cores. Este algoritmo se ejecuta cuando la lista en algún core se vacía. El core con la lista local vacía solicitá a los otros cores su índice de carga (el número de elementos sin procesar en su lista), y una vez que obtiene esta información eligé al core con mayor carga de trabajo. Una vez elegido al core con mayor carga, procede a solicitarlé parte de su carga.

La Figura 3.5 muestra el proceso del balance de carga con la política de subasta global en un cluster con 6 cores. Las letras A, B, C, D, E y F representan los cores. El número que aparece dentro de cada elipse representa el índice de carga del core, y para este ejemplo, inicialmente el core F se encuentra sin carga de trabajo, por lo que les solicita su índice de carga a cada uno de los cores (ver Figura 3.5a). Cuando el core F recibe el índice de carga de los demás cores (ver Figura 3.5b), identifica al core con mayor carga (el core E). Una vez ésto, el core F le envía un mensaje al E para solicitarle que le transfiera parte de su carga (ver Figura 3.5c), y finalmente, el core E le responde a F enviándole la mitad de su carga, por lo que después, ambos, E y F obtienen 390 elementos en sus listas (ver Figura 3.5d).

En el caso donde el core sin carga F, identifica que los índices de carga de los cores son 0 (que no hay más elementos para procesar), les envía un mensaje de terminación para indicarles que pueden finalizar su ejecución.

Aplicar esta política de balance de carga puede ocasionar problemas de escalabilidad, tales como cuellos de botella, ya que los cores sin carga (que puede ser desde un core hasta todos) envían y reciben mensajes de todos los cores en el cluster, lo que puede generar un alto

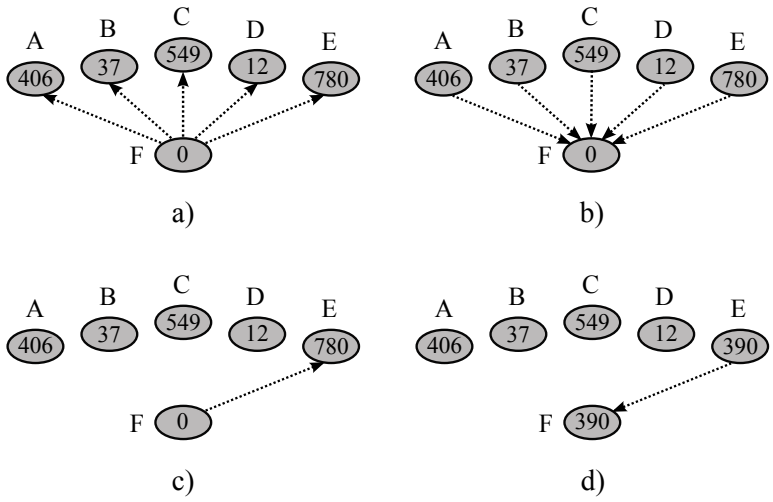

Figura 3.5: Algoritmo de subasta global en DLML.

costo en las comunicaciones cuando se cuenta con un cluster grande o cuando se aumenta el número de cores en el sistema.

Actualmente existen varias versiones de DLML enfocadas en mejorar el desempeño de la versión original. La siguiente sección presenta cada una de éstas y describe sus principales características.

3.5. Trabajo relacionado

Hasta el momento, además de la primera versión de DLML se encuentran 3 versiones más: DLML con manejo de información parcial, MC-DLML, y DLML-Grid. Cada una de estas diferentes versiones se describe a continuación.

DLML con manejo de información parcial

Esta implementación de DLML hace uso de una política de subasta con manejo de información parcial, en la cual cada core utiliza un subconjunto de los cores en el cluster para ejecutar el balance de carga. Esta política ayuda a distribuir las comunicaciones entre cores y permite escalar el sistema aumentando el número de cores en el cluster [13]. Cabe mencionar que esta versión también utiliza la arquitectura de la Figura 3.4.

En esta versión, cuando un core consume todos los elementos de su lista se comunica con un subconjunto de cores en el cluster para subastar su capacidad de procesamiento. Los cores

del subconjunto son los vecinos del core sin carga de trabajo, y cada core vecino tiene a su vez un subconjunto de vecinos con los cuales se comunica cuando se queda sin carga. Ésto ayuda a propagar la carga del sistema entre los cores del cluster y a mantener el sistema balanceado.

La comunicación de cada core con sus respectivos vecinos se basa en la topología lógica toroide. La Figura 3.6 muestra un toroide con 9 cores, y como puede observarse, cada core en esta topología tiene 4 vecinos. Cabe mencionar que cada core solo conoce el estado de carga de sus vecinos, por lo que no puede percibir cuando se ha terminado la carga de trabajo en los demás cores del cluster. Ésto causa que no se pueda determinar cuándo el sistema debe dar por terminada la ejecución de una aplicación, sin embargo, en esta versión de DLML se implementa el algoritmo PIF (Propagation of Information with Feedback) [29] como solución a este problema, ya que el algoritmo PIF permite establecer un panorama global de la carga de trabajo en el sistema, logrando de esta manera determinar el momento en que debe finalizarse la ejecución de una aplicación.

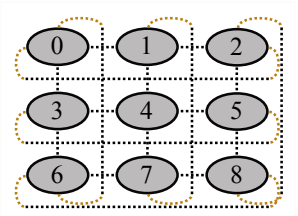

Figura 3.6: Topología toroide para un cluster con 9 cores.

En el protocolo PIF un core es el encargado de iniciar la propagación de la petición del índice de carga a sus cores vecinos. Posteriormente, cada core i-ésimo que recibe el mensaje por primera vez se encarga de reenviarlo a sus cores vecinos, excepto al core del que lo recibió.

El protocolo PIF crea un árbol de forma dinámica entre los cores del cluster. Los cores que se consideran como padres son aquellos que envían la petición a otro u otros cores vecinos. Por otro lado, cada core que recibe una petición es hijo de aquel core que se la envío. Cada core hijo responde a su padre cuando todos sus cores hijos le han enviado su índice de carga, por lo tanto, el mensaje de respuesta para su padre contiene el índice de carga de sus hijos y de él mismo.

La Figura 3.7a muestra una topología lógica de cores en la que se ejecuta el protocolo

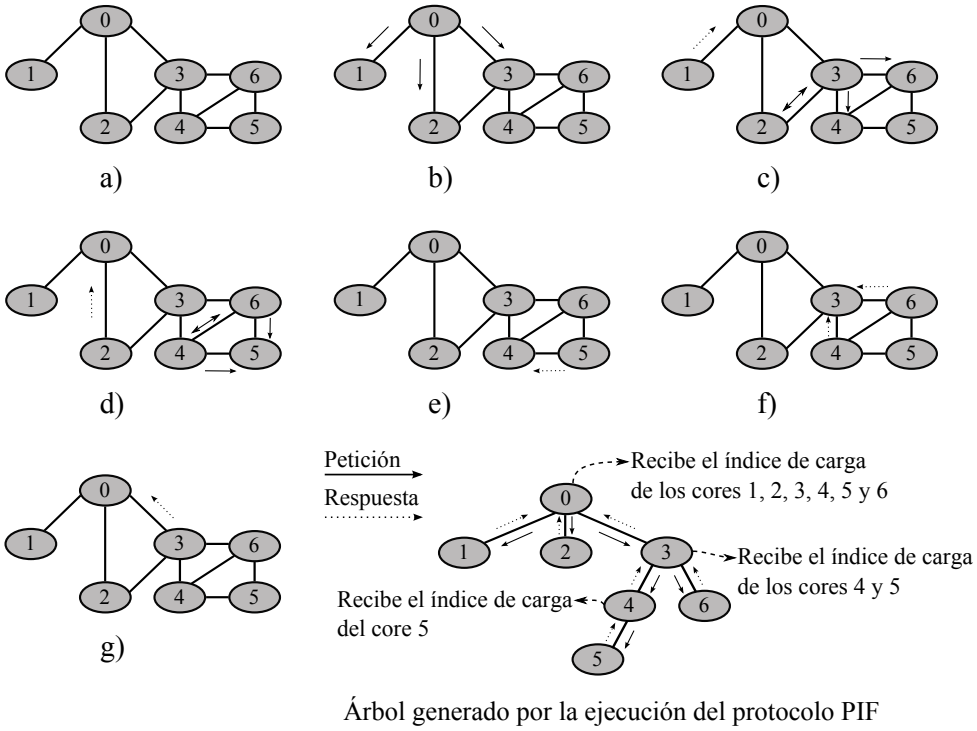

Figura 3.7: Ejecución del protocolo PIF en una topología y su árbol resultante.

PIF. En este ejemplo el core 0 se encarga de iniciar la propagación de la petición del índice de carga a sus vecinos, en este caso los cores 1, 2 y 3 (ver Figura 3.7b). El core 1 al no tener hijos le envía al core 0 su índice de carga, mientras que los cores 2 y 3 se encargan de propagar la petición a todos sus vecinos excepto al que se las envío, es decir, a su padre el core 1. Como puede verse en la Figura 3.7c los cores 2 y 3 son vecinos, y aunque se envían mensajes el uno al otro, los ignoran ya que recibieron la petición del core 0 antes de este intercambio de mensajes se realizará. Una vez ésto, el core 2 al no tener hijos, envía su índice de carga su padre, mientras que los cores 4 y 6 envían la petición a sus vecinos (ver Figura 3.7d). Los cores 4 y 6 se énvian la petición uno al otro como en el caso anterior, y tienen en común el mismo vecino, el core 5. En este caso se supone que la petición del core 4 llega antes que la del core 6, por lo que el core 5 ignora la petición del core 6 y establece al core 4 como padre.

El core 5 al no tener a quién más enviar la petición, le envía su índice de carga a su padre (ver Figura 3.7e). De igual manera los cores 4 y 6 envían su índice de carga a su padre (ver Figura 3.7f). Cabe mencionar que el mensaje que recibe el core 3 por parte del core 4 incluye el índice de carga de los cores 4 y 5. Por lo tanto, cuando el core 3 recibe los mensajes de los

cores 4 y 6, recibe los índices de carga de los cores 4, 5 y 6. Finalmente el core 0 recibe un mensaje del core 3 (ver Figura 3.7g), y de esta manera el core 0 obtiene los índices de carga de los cores 1, 2, 3, 4, 5 y 6.

La Figura 3.7 muestra también el árbol generado por la ejecuciíon del protocolo PIF en la topología de la Figura 3.7a. El árbol se genera de forma dinámica, ya que la velocidad en que se envían y reciben los mensajes en la topología influye en gran medida en la definición de padres e hijos. El árbol es utilizado para las siguientes evaluaciónes de carga en el sistema, por lo que cada core almacena infomación acerca de quién es su padre y quienes son sus hijos.

En conclusión, DLML con manejo de información parcial ayuda a reducir la cantidad de comunicaciones generadas durante la ejecución de una aplicación, permitiendo así la escalabilidad del sistema. Cabe señalar que tanto DLML con manejo de información global, así como DLML con manejo de información parcial, pueden trabajar en clusters multicore, ya que sólo deben ejecutar los procesos *Aplicación* y *DLML* de manera simultánea en cada core. No obstante, aún dentro de un mismo core, la comunicación de estos procesos emplea paso de mensajes, ocasionando así sobrecarga en la cantidad de mensajes MPI. Una versión de DLML que disminuye el overhead de comunicaciones generado por el paso de mensajes entre los procesos residentes en un mismo nodo es MC-DLML (Multicore DLML).

MC-DLML

Originalmente DLML fue diseñado para ejecutarse en clusters conformados por nodos con un solo procesador, sin embargo, hoy en día esta presente la existencia de nodos con procesadores multicore. En estos nodos con procesadores multicore se puede ejecutar el DLML original, sin embargo la comunicación por paso de mensajes no es la más natural ya que se provoca un aumento en el overhead de comunicaciones. MC-DLML (Multicore DLML) [19] permite reducir el overhead de comunicaciones mediante la reducción del paso de mensajes entre los cores dentro de un mismo nodo, ya que utiliza memoria compartida para la comunicación entre los cores de un nodo, y paso de mensajes para la comunicación entre los nodos del cluster.

Como se muestra en la figura 3.8, a diferencia de las versiónes de DLML descritas anteriormente, MC-DLML utiliza un hilo *Aplicación* por cada core del cluster, y sólo uno de los hilos en cada uno de los nodos, nombrado “maestro” (la *Aplicación* con índice 0 en cada nodo) es

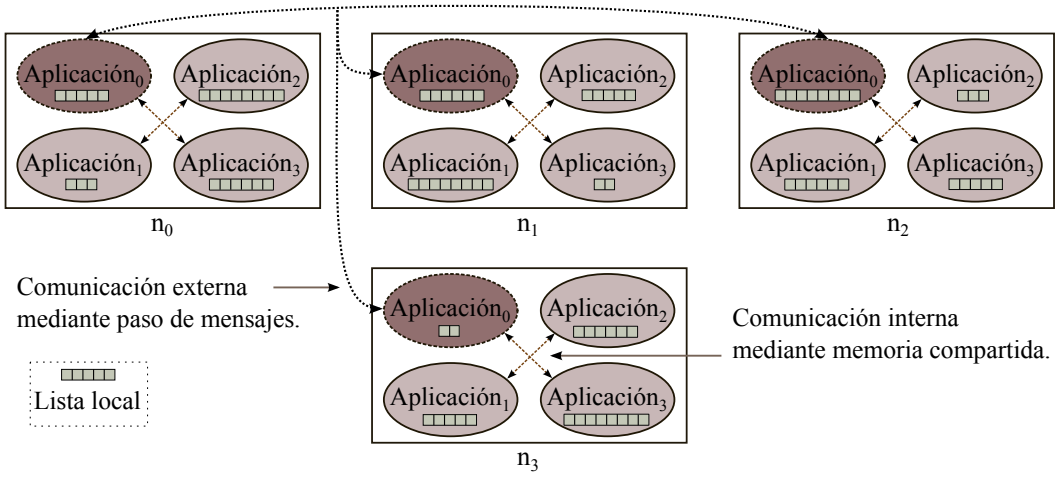

Figura 3.8: Arquitectura de MC-DLML en un cluster con 4 nodos, y 4 cores por nodo.

utilizado para hacer/atender peticiones hacia/desde nodos remotos. La comunicación entre los hilos *Aplicación* en un mismo nodo se lleva a cabo mediante memoria compartida, y la comunicación entre los hilos "maestros" (localizados en nodos diferentes) se realiza mediante paso de mensajes. Por otro lado, al igual que las versiones anteriores, se encuentra una lista de datos en cada hilo *Aplicación*, en la cual se insertan y obtienen elementos del frente.

Cada hilo *Aplicación* obtiene elementos de su lista local mediante la funcion *DLML_get*. Cuando la lista de alguno se vaciá, la función *DLML_get* permite obtener items de las listas de otros hilos *Aplicación*. Para esta tarea, la función *DLML_get* hace que los demás hilos dentro del mismo nodo paren de usar sus listas. Posteriormente se determina cuál hilo contiene la lista más grande y se transfiere la mitad de su lista al hilo *Aplicación* sin carga de trabajo.

Cuando un hilo "maestro" identifica la ausencia de carga de trabajo en todos los hilos *Aplicación* del nodo en que se encuentra, contacta a los demás hilos "maestros" en el cluster para solicitarles los índices de carga de sus respectivos hilos *Aplicación*. Los hilos "maestros" le envían los índices de carga, y una vez que los recibe, se encarga de determinar cuál es el índice con mayor carga y en que nodo se localiza. Una vez que determina ésto envía una solicitud de datos al respectivo hilo "maestro" remoto. El hilo "maestro" remoto recibe la solicitud, y procede a bloquear las listas de los hilos *Aplicación* en el nodo, para transferirle los datos al hilo "maestro" que envío la petición y posteriormente desbloquear las listas para continuar con el procesamiento.

En el momento en el que un hilo "maestro" sin carga de trabajo recibe datos remotos, se

encarga de distribuirlos entre todas las *Aplicaciónes* del nodo en el que reside (incluyéndose), llevando a cabo el procedimiento descrito inicialmente. Por otro lado, si se descubre que no hay más datos a procesar en el sistema, el hilo "maestro" les envía un mensaje de terminación a los hilos *Aplicación* y se finaliza la ejecución.

Esta versión de DLML toma ventaja de la memoria compartida entre los hilos *Aplicación* que son ejecutados en el mismo nodo (ver figura 3.8). Con las versiones anteriores de DLML, el enviar un dato entre procesos *Aplicación*, aún dentro de un mismo nodo, implicaba copiar el dato a un mensaje, enviar el mensaje, recibirlo, y finalmente copiarlo a una nueva lista, situación que es agilizada en esta versión por el uso de la memoria compartida. Finalmente cabe resaltar que en algunas aplicaciones implementadas, esta versión se ejecuta hasta el doble de rápido que la versión original de DLML, cuando la comunicación por paso de mensajes no es excesiva [19].

Las versiónes de DLML descritas hasta el momento presentan buenos resultados ante diversas aplicaciones, sin embargo, no contemplan el caso de aplicaciones que requieren capacidades tan grandes de procesamiento que son casi prohibitivas ejecutarlas en un solo cluster, o en el peor de los casos, imposibles ejecutarlas con la infraestructura que se cuenta. Una tercera versión DLML que permite acceder a multiplés clusters, y así incrementar la capacidad de procesamiento es DLML-Grid.

DLML-Grid

DLML-Grid es una arquitectura jerárquica de software para DLML, la cual está diseñada para hacer un mejor uso de los recursos en los ambientes Grid. El propósito de la arquitectura es localizar la comunicación dentro de los clusters tanto como sea posible, y así reducir la sobrecarga de comunicaciones [20]. En esta arquitectura, el balance de carga *intra-cluster* (que se lleva a cabo dentro de cada cluster) tiene mayor prioridad que el balance de carga *inter-cluster* (que se realiza entre los diferentes clusters), y en el momento en el que la carga de trabajo en un cluster se agota, se hace uso del balance de carga inter-cluster.

El diseño de esta versión de DLML fue inspirado en un modelo jerárquico descrito en [21] aunque no lo adopta totalmente. Tal modelo jerárquico considera a un Grid como un sistema de clusters interconectados, que son administrados por 3 tipos de procesos diferentes en tres niveles distintos (ver figura 3.9). Cada nivel se describe a continuación:

- **Nivel 0**. En este nivel se encuentra un proceso llamado GM (Grid Manager), el cual lleva a cabo las siguientes acciones:

  - Recolecta información sobre la carga de trabajo de cada uno de los clusters en el Grid.
  - Toma decisiones de balance de carga a nivel *inter-cluster*, por ejemplo, determinar qué cluster debe transferir carga de trabajo a un cluster descargado.
  - Interactúa con los cores conocidos como CMs (Clusters Managers) que se encuentran en el nivel 1 para organizar la transferencia de carga entre ellos.
  - Recoge los resultados parciales finales de los CMs.

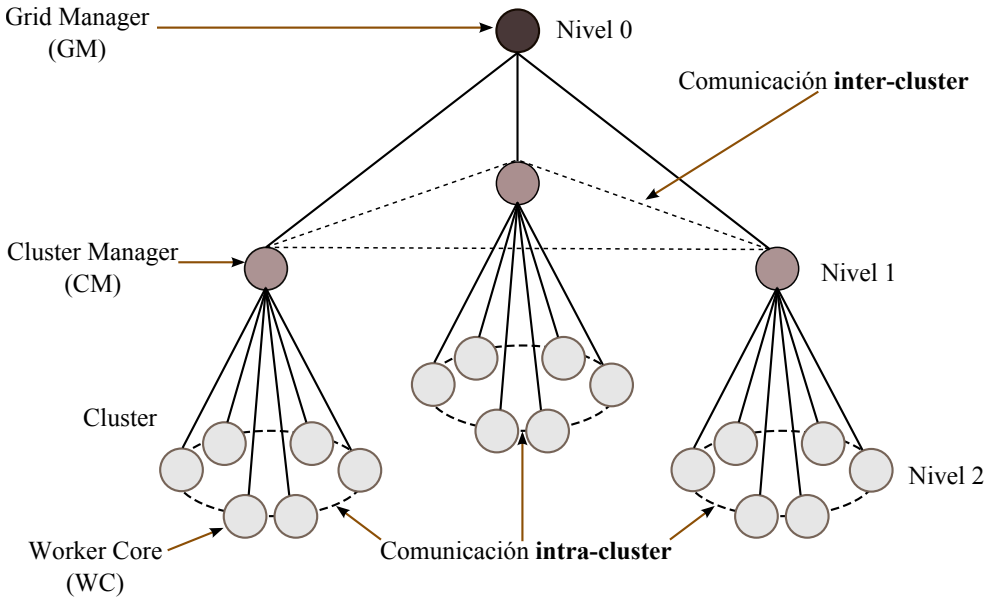

Figura 3.9: Modelo jerárquico para un ambiente Grid.

- **Nivel 1**. En este nivel se encuentran los CMs (Clusters Managers), los cuales se encargan de llevar a cabo las siguientes acciones:

  - Recolectar información acerca de la carga de los cores WCs (Worker Cores) que residen en el mismo cluster (éstos se encuentran en el nivel 2).
  - Comunicarse con el GM para enviar la carga total en su cluster, para preguntar por más datos, o para transferir la carga requerida.

  - Comunicarse con los demás CM para enviar o recibir carga, siguiendo la desición tomada por el GM.
  - Comunicarse con los WC para recolectar y enviar carga de trabajo en caso de una transferencia.
  - Recolectar los resultados parciales de los WCs y enviarselos al GM.

- **Nivel 2**. En este nivel se encuentran todos los cores designados al procesamiento de los datos también conocidos como WCs (Worker Cores), los cuales se encargan de realizar las siguientes acciones:

  - Enviar su información de carga a su respectivo CM cuando es requerido.
  - Ejecutar el código de la aplicación.
  - Participar en el balance de carga a nivel *intra-cluster*.
  - Enviar sus resultados parciales a su CM al final de la ejecución.

Como se mencionó anteriormente, esta versión de DLML, aunque se basa en el modelo descrito en [21], no lo adopta completamente. La principal diferencia se encuentra en la estrategia de balance de carga, ya que con el modelo original, el balance de carga en cada cluster es administrado por el CM de manera centralizada, situación que cambia en DLML-Grid, debido a que cada WC puede enviar acciones de balance de carca con los WCs dentro del mismo cluster.

La figura 3.10 muestra la arquitectura de DLML-Grid para un Grid conformado por cuatro clusters. Como puede observarse, cada WC se comunica con su respectivo CM y con los WCs que residen en el mismo nodo. Los CMs se comunican entre ellos y con el GM a través de Internet. Los CMs se comunican para transferirse carga, según las indicaciones del GM. El algoritmo que emplea DLML-Grid para llevar a cabo el balance de carga utiliza la política de subasta global descrita anteriormente, por lo tanto se hace uso de la información de carga de todos los clusters para realizar el balance de carga *inter-cluster*.

Como se mencionó anteriormente, DLML-Grid hace uso de Internet como medio de comunicación, y se emplea también una VPN (Virtual Private Network), ya que permite la comunicación de dos o más redes LAN a través de Internet. El beneficio de una VPN es que

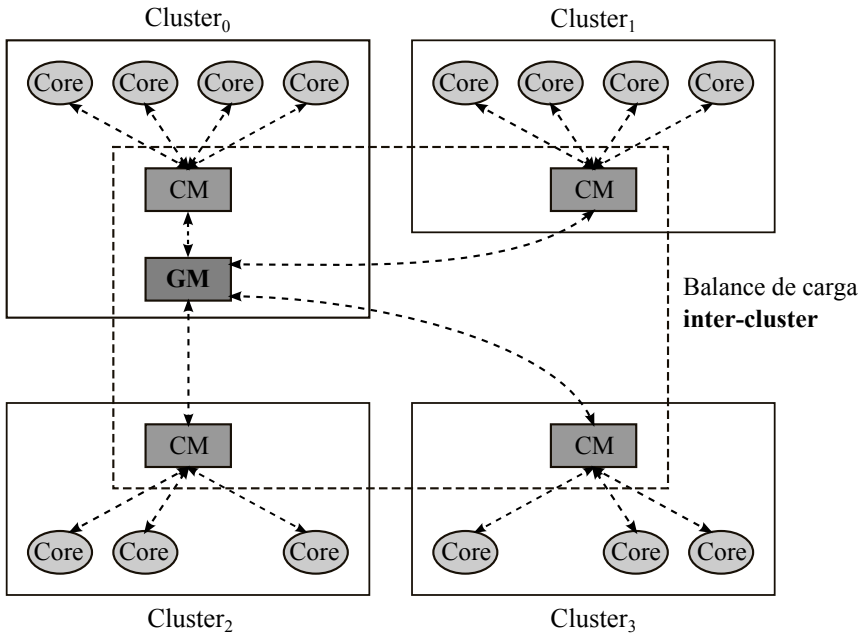

Figura 3.10: Arquitectura de DLML-Grid.

crea una red virtual, en la cual cada elemento en ésta se comporta como cualquier elemento que reside en una misma red LAN.

En la secciones 3.4 y 3.5 se presentó el funcionamiento de DLML con el manejo de información global y con el manejo de información parcial respectivamente. En la sección 3.5 se describió MC-DLML, y finalmente en la sección 3.5 se presentó DLML-Grid. Cabe mencionar que todas estas versiones de DLML tienen una limitacioń en común, y radica en la cantidad de memoria RAM empleada para procesar los datos de una aplicación. La versión original de DLML (con una política de subasta con manejo de información global), independientemente de presentar problemas de escalabilidad con la transferencia simultanea de mensajes, deja de lado la existencia de aplicaciones en las que la cantidad total de datos a procesar es superior a la capacidad de memoria RAM con la que se cuenta. Lo mismo sucede con la versión de DLML que utiliza una política de subasta con manejo de información parcial, y con la implementación MC-DLML. Si bien estas implementaciones permiten reducir la sobrecarga de comunicaciones en el sistema, e inclusive reducir tiempos de ejecución, no están enfocadas en tratar con aplicaciones estáticas con cantidades de datos que sobrepasan la cantidad de memoria de la infraestructura utilizada. Finalmente, la versión de DLML-Grid permite tener acceso a multiplés clusters y así poder ejecutar aplicaciones imposibles de ejecutar en un sólo

cluster, no obstante, existen aplicaciones que pueden sobrepasar incluso las capacidades de cada uno de los clusters en el grid (principalmente la memoria RAM), aún con el balance de carga que se incluye en esta implementación.

Una solución a la limitación de las versiones existentes de DLML, es realizar una modificación a la biblioteca DLML para que permita realizar el procesamiento de grandes cantidades de datos que superan la capacidades de memoria con la que se cuenta, aún si se tiene acceso a un sólo cluster.

Capítulo 4
DLML-IO

En este capítulo presentamos la propuesta de la extensión a la biblioteca DLML para procesar grandes cantidades de datos mediante el uso de archivos.

4.1. Introducción

Como se describió en el capítulo anterior, DLML permite distribuir la carga de trabajo de una aplicación entre los cores de un cluster, y más aún, ejecutar el balance de carga a tiempo de ejecución, sin embargo, la ejecución de una aplicación cuyos datos sobrepasen la cantidad de memoria RAM con la que se cuenta, provocaría el colapso del sistema.

El procesamiento de grandes volúlmenes de información requiere principalmente grandes capacidades de memoria y de almacenamiento, ya que las aplicaciones de este tipo llegan a procesar datos del orden de Gigabytes (GB), Terabytes (TB), e incluso Petabytes (PB). Cabe mencionar que actualmente es muy común encontrar clusters en donde los discos duros de los nodos que los conforman alcanzan los 500 GB o 1 TB, lo que beneficia en gran medida a este tipo de aplicaciones, ya que generalmente la información a procesar se encuentra almacenada en archivos de cientos de Gigabytes.

Si bien la capacidad actual de los discos duros ayuda a realizar el procesamiento de grandes volúmenes de información, existe otro problema que puede impedir esta labor en DLML, y es la capacidad de memoria RAM de la cual se hace uso. Aunque las capacidades de memoria han aumentado y continúan haciéndolo, todavía no es común encontrar computadoras que alcancen capacidades del orden de cientos de Gigabytes, y mucho menos con capacidades superiores. Incluso empleando el uso de un cluster se tiene este problema, ya que aunque la capacidad de memoria RAM puede aumentar (debido a la suma de las capacidades de memoria de los nodos que lo conforman), la existencia de aplicaciones cuyos datos a procesar

sobrepasen la capacidad de memoria del cluster es inminente. Una solución a este problema es aumentar la cantidad de recursos en el cluster cada vez que sea necesario, lo cual puede resultar tan costoso como emplear el uso de una supercomputadora, no obstante, otra alternativa es idear la manera de procesar grandes cantidades de datos con los recursos con los que ya se cuenta.

En esta tesis se presenta DLML-IO, una extensión a la bibloteca DLML que permite procesar grandes cantidades de información, conservando el modelo de programación de DLML descrito anteriormente. La siguiente sección describe el diseño empleado para el desarrollo de DLML-IO.

4.2. Diseño de DLML-IO

DLML-IO retoma la arquitectura del DLML original descrita en la sección 3.3, e incluye el manejo de archivos, ya que como se mencionó en la sección anterior, generalmente las grandes cantidades de datos que deben ser procesadas por una aplicación, se almacenan en archivos del orden de Gigabytes, además de que la información generada por el procesamiento de los datos de entrada, suele ser tan grande que es imposible mantenerla en memoria RAM, por lo que nuevamente se hace uso de los archivos para el almacenamiento de esta información.

Como parte del manejo de archivos, DLML-IO incluye la lectura distribuida de los archivos de entrada que tienen que ser procesados. Ésto permite dividir la lectura de los archivos e inicialización de items, entre los procesos *Aplicación* que se encuentran en el sistema. La Figura 4.1 muestra un ejemplo de la lectura distribuida de un archivo a procesar. Como puede verse, cada proceso *Aplicación* realiza la lectura de un segmento del archivo de entrada. Cabe mencionar que los procesos *Aplicación* realizan la inicialización de items conforme efectúan la lectura de su respectivo segmento. Por esta razón, una vez que los procesos *Aplicación* terminan la lectura de datos, obtienen items en su lista local que están relacionados con los segmentos que procesaron.

Durante la lectura de archivos e inicialización de items, los procesos *Aplicación* corren el riesgo de consumir la memoria disponible, ya que los segmentos de los archivos de entrada que tienen que procesar, pueden llegar a ser tan grandes que pueden sobrepasar la memoria disponible, lo que colapsaría el sistema en algún punto de la inicialización de items. Para evitar

esta situación con DLML-IO, se implementó una solución en la cual, cada proceso *Aplicación* verifica si hay memoria disponible antes de insertar un item en su lista local, de otra manera, detiene la inserción de items, y comienza a procesar los items que ya se encuentran en la lista. Conforme se libera espacio en la memoria RAM, los procesos *Aplicación* reinician la lectura e inicialización de los datos de entrada restantes, hasta que nuevamente se detecte la ausencia de memoria disponible, o hasta que se complete la lectura de los archivos de entrada.

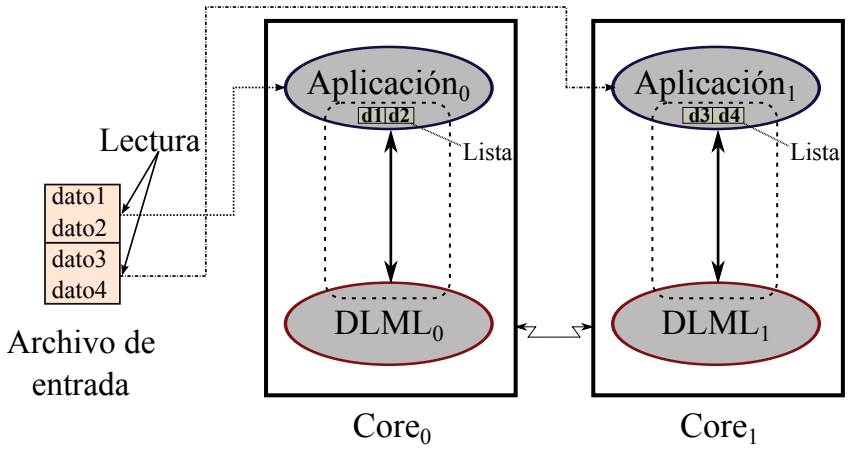

Figura 4.1: Lectura distribuida con DLML-IO.

En la solución propuesta, los procesos *Aplicación* solicitan al sistema operativo la información relacionada con la memoria disponible. No obstante, las pruebas realizadas permitieron identificar que la memoria liberada por los procesos *Aplicación*, no es reportada inmediatamente por el sistema operativo, o sólo reporta una parte. Ésto sobresalió en el momento en el que los procesos *Aplicación* detenían la lectura de datos de entrada e inicialización de items, y comenzaban a procesar los items que ya se encontraban en sus listas locales. Este detalle provocó que los procesos *Aplicación* detectaran menos memoria disponible e incrementaran el tiempo de espera para la siguiente de lectura de los datos de entrada, lo que a su vez afecto los tiempos de procesamiento.

Es importante resaltar que aunque el sistema operativo no reporte inmediatamente las cantidades de memoria que son liberadas, no significa que no las haya liberado, ya que por razones de eficiencia el sistema operativo mantiene información en memoria para aumentar la velocidad de acceso a datos, sin embargo, la memoria ocupada por esta información puede ser reasignada por el sistema operativo, o reportada como memoria libre más adelante.

Con el objetivo de eliminar el detalle de memoria localizado en la solución propuesta, se planteó una modificación que consiste en realizar un conteo independiente de la memoria disponible. En esta modificación, los procesos *Aplicación* son los encargados de llevar el conteo de la memoria disponible del nodo en que residen. Para llevar a cabo ésto, al inicio de la ejecución de una aplicación, cada proceso *Aplicación* realiza una lectura de la memoria disponible del nodo en el que se encuentra, para posteriormente dividirla entre los procesos *Aplicación* residentes en el nodo (incluyéndose). De esta manera cada proceso *Aplicación* obtiene una cantidad de memoria disponible para utilizar (ver Figura 4.2).

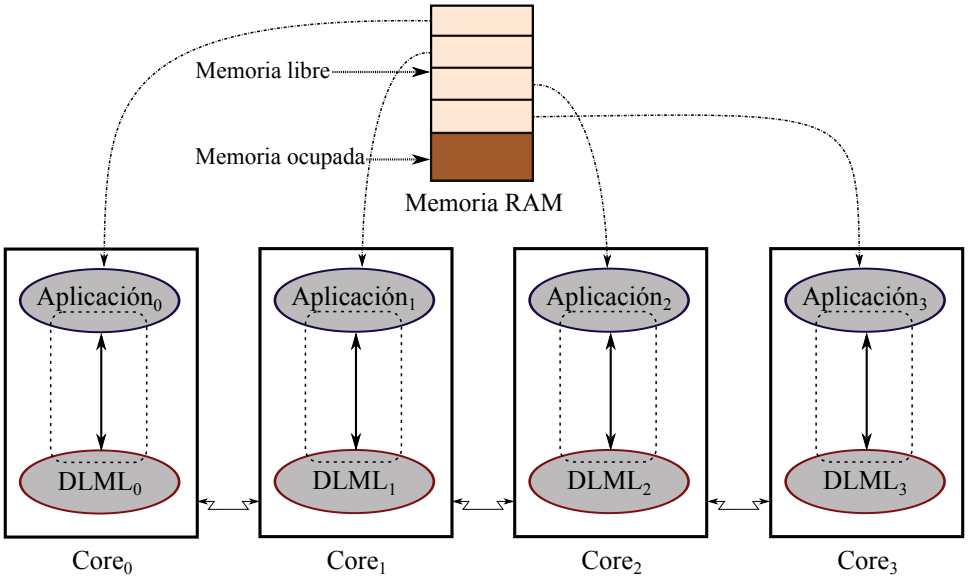

Figura 4.2: División de la cantidad de memoria disponible entre los procesos *Aplicación* residentes en un nodo con 4 cores.

El detalle con esta solución propuesta, es que además de los procesos *Aplicación*, los procesos *DLML* también consumen memoria por más mínima que sea (independientemente de la que utilizan para realizar la transferencia de datos), de manera que dividir la cantidad de memoria disponible, sólo entre los procesos *Aplicación*, puede provocar el colapso del sistema. Ésto se debe a que el consumo de memoria de los procesos *DLML*, no es contabilizado por los procesos *Aplicación*, lo que ocasiona que éstos puedan intentar usar cantidades de memoria mayores a las que realmente se tienen. Para evitar esta situación, se reservó una cantidad de la memoria disponible para el uso de los procesos *DLML*. Esta reserva se consiguió modificando el número en que los procesos *Aplicación* dividen la cantidad de memoria disponible del

nodo en que se encuentran. Este número se redefinió como el número de procesos *Aplicación* residentes en el nodo, más una unidad, donde la unidad representa la memoria de reserva para los procesos *DLML* residentes en el nodo.

Si bien los procesos *Aplicación* se encargan de administrar una cantidad de la memoria disponible (del nodo en el que se encuentran), es importante que dentro de la administración consideren que no deben consumir completamente su cantidad de memoria, ya que ésto puede ocasionar que se termine la memoria disponible de los nodos en que residen, lo que nuevamente colapsaría el sistema. No obstante para eliminar este problema, se implementó un umbral de memoria que les permite saber a los procesos *Aplicación*, hasta que punto de su cantidad de memoria disponible deben utilizar (ver Figura 4.3).

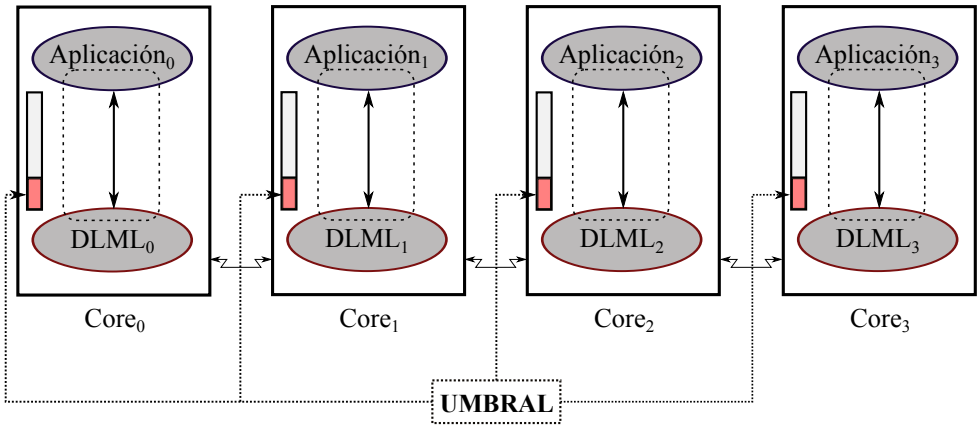

Figura 4.3: Umbral de memoria para los procesos Aplicación de un nodo con 4 cores.

Una vez que los procesos *Aplicación* obtienen su cantidad de memoria a utilizar, establecen el umbral de memoria. Cabe mencionar que el umbral de memoria es definido mediante una macro en el código de la aplicación a ejecutar. Por defecto, el umbral es definido al 30 % de la memoria disponible, sin embargo, puede modificarse para aumentarse o reducirse, dependiendo de la aplicación.

La Figura 4.4 muestra un ejemplo de cómo se realiza la distribución de la cantidad de memoria disponible, de un nodo con cuatro cores. Como puede verse, en cada core del nodo se encuentra un proceso *Aplicación* y un proceso *DLML*, además, hay 2 GB de memoria disponible en el nodo, y un umbral de memoria con el valor por defecto (al 30 %). Al inicio de la ejecución cada proceso *Aplicación* hace la lectura de la memoria disponible y la divide entre 5 (2GB/5). Este número corresponde a los 4 procesos *Aplicación* en el sistema, más la

unidad utilizada como reserva para los procesos *DLML*. Una vez ésto, cada proceso *Aplicación* dispondrá de una cantidad de 400 MB de memoria, y establecerá el umbral a los 120 MB.

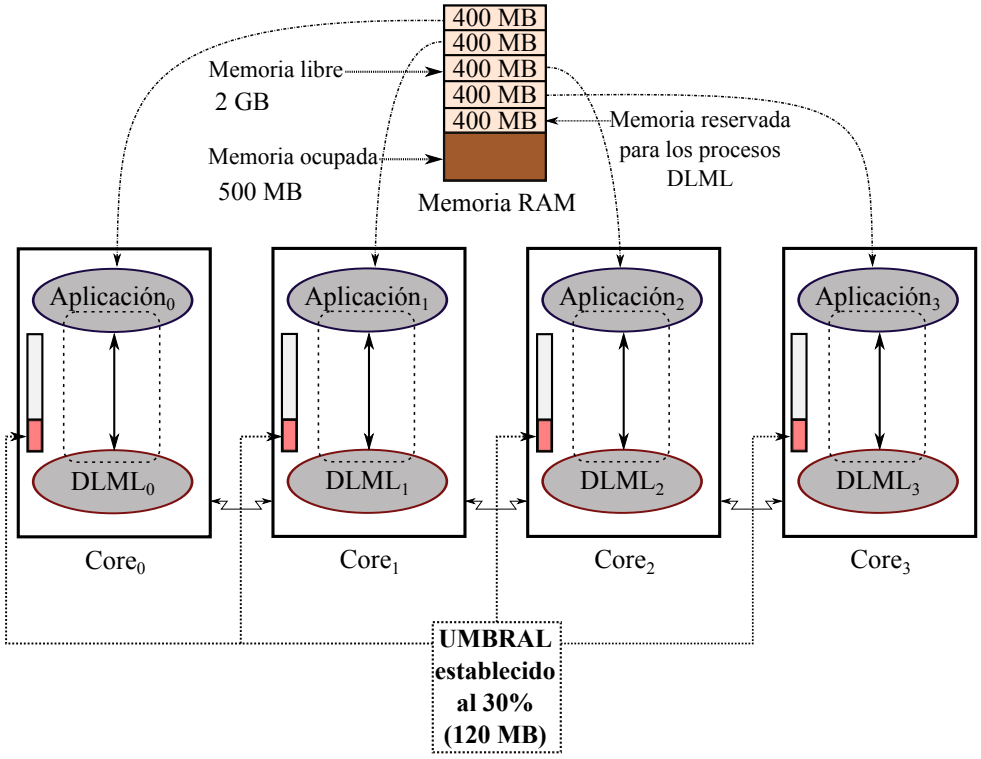

Figura 4.4: Umbral de memoria para los procesos Aplicación de un nodo con 4 cores.

Como se mencionó anteriormente, DLML-IO permite realizar la lectura distribuida de los archivos de entrada, además, permite también procesar gradualmente los grandes archivos que superan la cantidad de memoria disponible, sin embargo, para llevar a cabo ésto, incluimos un conjunto de funciones en DLML-IO, las cuales se describen en la siguiente sección. Cabe mencionar que estas funciones son utilizadas una vez que los procesos *Aplicación* han determinado su cantidad de memoria a utilizar, y han establecido su respectivo umbral de memoria.

4.3. Funciones de DLML-IO

En esta sección se describe cada una de las funciones de DLML-IO implementadas hasta el momento. Las funciones incluidas son: DLML_fileList, DLML_distributeOffsets, DLML_Load, y DLML_Write.

4.3.1. Función DLML_fileList

A menudo las aplicaciónes que procesan archivos contenidos en un directorio, necesitan llevar un registro de los archivos a procesar. La función *DLML_fileList* permite crear una lista con los nombres de los archivos localizados en un directorio, y recibe como parámetros, la ruta del directorio, una lista, y un contador de memoria actual. Conforme la función obtiene los nombres de los archivos a procesar, inserta los nombres como elementos en la lista. Cada vez que se inserta el nombre de un archivo en la lista, la función descuenta la memoria utilizada del contador de memoria disponible. Por otro lado, la memoria utilizada por los elementos de la lista es recuperada por el contador de memoria, conforme se van procesando los archivos. Cabe mencionar que esta función es generalmente ejecutada sólo por un proceso *Aplicación* en el sistema, el cual lleva el control sobre los archivos a procesar.

4.3.2. Función DLML_distributeOffsets

Al igual que la función *DLML_fileList*, esta función es ejecutada por un sólo proceso *Aplicación* en el sistema. El objetivo de esta función es realizar una segmentación lógica de los archivos de entrada, para posteriormente distribuir la información de los segmentos a procesar, entre los diferentes procesos *Aplicación*. Como se mencionó en la sección 3.3, cada proceso *Aplicación* tiene una lista local de datos a procesar, sin embargo, en DLML-IO, cada proceso *Aplicación* incluye una segunda lista de datos, la cual es utilizada para almacenar la información de los segmentos a procesar.

La función *DLML_distributeOffsets* recibe como entrada el nombre de uno o dos archivos a procesar, un contador de memoria actual y la segunda lista local (ver Figura 4.5). Típicamente la necesidad de procesar 2 archivos a la vez se debe a que la aplicación en cuestión requiere leer datos simultáneamente de ambos archivos. Por otro lado, cuando no se tiene esta dependencia, los archivos de entrada son procesados uno a uno, es decir, se toma un archivo de entrada para su procesamiento, y cuando éste termina se procede a tomar otro de los archivos restantes.

Una vez que la función *DLML_distributeOffsets* recibe los parámetros de entrada, procede a realizar la partición lógica de los archivos a procesar. La partición lógica depende del número de procesos *Aplicación*, de manera que los archivos de entrada son segmentados lógicamente entre el número de procesos *Aplicación* en el sistema. Para llevar a cabo la seg-

mentación, la función *DLML_distributeOffsets* cuenta el número de renglones de los archivos, y posteriormente divide el número de renglones de cada archivo (en caso de recibir 2 archivos a procesar) entre el número de procesos *Aplicación* en el sistema.

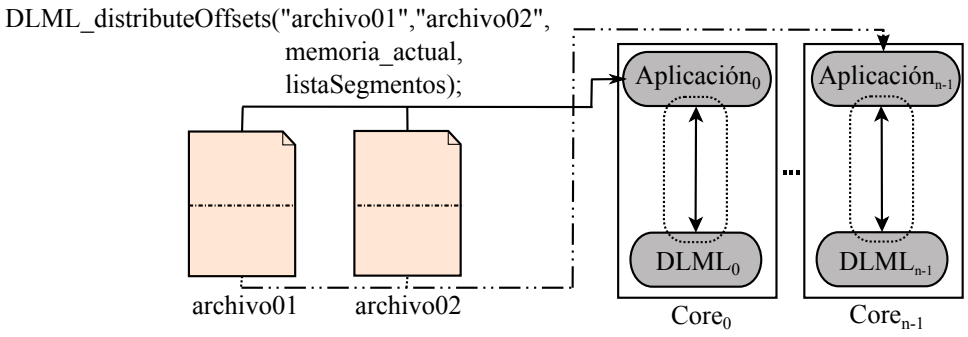

Figura 4.5: Funcionamiento de la función DLML_DistributeOffsets.

Cuando la función *DLML_distributeOffsets* recibe dos archivos de entrada, considera que los dos archivos contienen el mismo número de renglones, ya que generalmente el tipo de aplicaciones que requieren ésto, tienen que leer datos simultáneamente de cada archivo. Por otra parte se tienen 3 casos a considerar cuando la función *DLML_distributeOffsets* recibe uno o dos archivos de entrada. Suponiendo que la función recibe un archivo, el primer caso surge cuando la división del número de renglones entre el número de procesos *Aplicación* es exacta (ver Figura 4.6a), el segundo caso surge cuando la división no es exacta y el número de procesos *Aplicación* es menor que el número de renglones (ver Figura 4.6b), y finalmente el tercer caso surge cuando el número de procesos *Aplicación* es mayor que el número de renglones (ver Figura 4.6c).

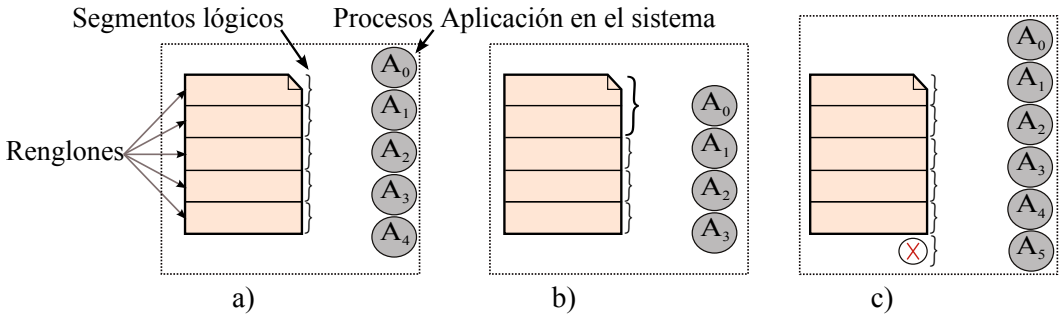

Figura 4.6: Diferentes formas de segmentación de un archivo con DLML_distributeOffsets.

En cualquier caso, la función *DLML_distributeOffsets* crea tantos segmentos como pro-

cesos *Aplicación* hay en el sistema (ver Figura 4.6), sin embargo, el número de renglones dentro de los segmentos es lo que varia en cada caso. Con el primer caso, la función *DLML_distributeOffsets* incluye la misma cantidad de renglones en cada uno de los segmentos (ver Figura 4.6). Con el segundo caso, la función *DLML_distributeOffsets* toma la parte entera de la división para determinar la cantidad de renglones de cada segmento, sin embargo, como puede verse en la Figura 4.6b, la función incluye dentro del primer segmento, tanto la cantidad de renglones calculada, como los renglones que sobran en la división, por lo que en este ejemplo el primer segmento contiene 2 renglones. Finalmente, con el tercer caso, la función *DLML_distributeOffsets* incluye un renglón a cada segmento, sin embargo, una vez que se terminan los renglones del archivo, la función crea información para los siguientes segmentos, la cual indica la ausencia de renglones.

Es importante señalar que una vez que la función *DLML_distributeOffsets* determina el número de renglones para los segmentos, se encarga de obtener el desplazamiento de donde inicia cada uno de los segmentos. La función *DLML_distributeOffsets* utiliza la segunda lista local para almacenar la información asociada a los segmentos. Básicamente los items o elementos de esta segunda lista local, contienen el nombre del archivo a procesar, el desplazamiento de inicio del segmento y su número de renglones.

Cuando la función *DLML_distributeOffsets* recibe dos archivos de entrada, cada elemento de la segunda lista local contiene información sobre dos segmentos provenientes de cada uno de los archivos. Esencialmente cada elemento contiene los nombres de los archivos, el desplazamiento de inicio de cada segmento, y su respectivo número de renglones. Cabe mencionar que los segmentos tienen el mismo número de renglones, y más aún, tienen una dependencia entre sí, es decir, si el segmento del primer archivo tuviera los renglones del 0 al 9, el segmento del segundo archivo tendría la misma secuencia de renglones.

En el caso en el que los segmentos no contienen renglones, los elementos de la segunda lista sólo contienen ciertos valores negativos para indicar la ausencia de renglones.

Conforme la función *DLML_distributeOffsets* inserta datos en la segunda lista, descuenta la memoria utilizada del contador de memoria actual, no obstante, la memoria es recuperada por el contador en cuanto se eliminan los datos de la lista. Una vez que esta función inserta los datos de los segmentos a procesar, le permite al proceso *Aplicación* que la invoco, distribuirlos entre los procesos *Aplicación* en el sistema (incluyéndose). De esta manera los procesos

Aplicación obtienen la información de uno o dos segmentos a procesar (ver Figura 4.5).

4.3.3. Función DLML_Load

Esta función es ejecutada por los procesos *Aplicación*, una vez que reciben los datos de los segmentos a procesar y comprueban la existencia de renglones en los segmentos.

La función *DLML_Load* permite hacer la lectura de datos en los segmentos de entrada, y la posterior creación de items asociados a los datos que se van leyendo, no obstante, para segmentos que contienen grandes cantidades de datos, la creación de items puede llegar a consumir toda la memoria disponible, por lo que el objetivo de esta función, es regular la cantidad de items que son creados a partir de los datos en los segmentos.

La función *DLML_Load* recibe como entrada: la información asociada a los segmentos que se tienen que procesar, una lista de datos local, un contador de memoria actual, el umbral de memoria, y un operador que habilita uno de los 3 modos de funcionamiento. Como puede verse en la Figura 4.7, esta función tiene tres modos de operación o funcionamiento, "+", "*" y "w". Como puede observarse, cada uno de estos modos permite realizar una lectura de datos y creación de items de forma diferente.

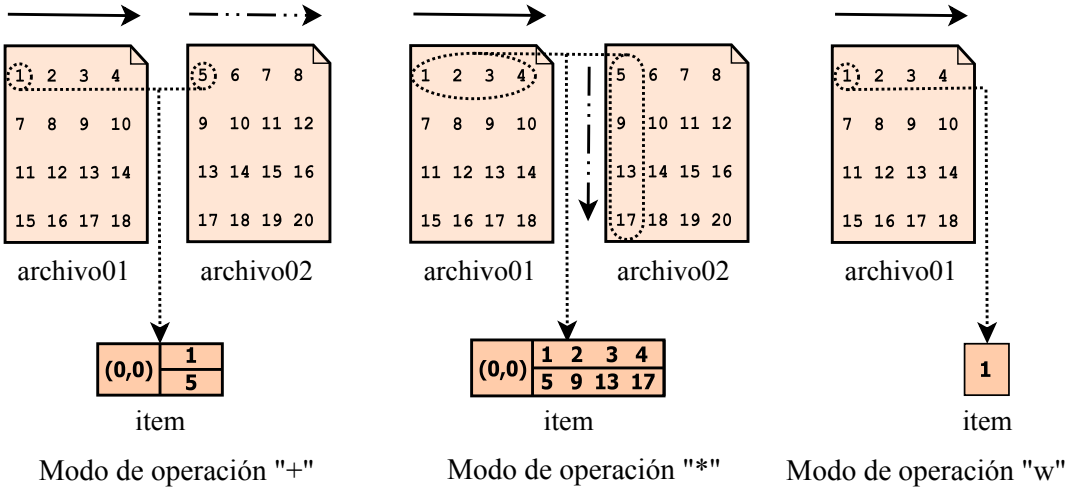

Figura 4.7: Modos de operación de DLML_Load.

Cuando la función *DLML_Load* opera con el modo "+", recibe la información relacionada con 2 segmentos a procesar (cada uno en un archivo diferente), los cuales contienen la misma cantidad de renglones. Posteriormente, este modo permite realizar una lectura por renglones

en ambos segmentos, no obstante, la lectura se realiza de forma simultánea, de manera que se van leyendo al mismo tiempo cada uno de los elementos en ambos reglónes, por ejemplo, como muestra la Figura 4.7 (con este modo de operación), si tuviera que leerse el primer renglón de los archivos 01 y 02, se leerían al mismo tiempo los datos 1 y 5, posteriormente los datos 2 y 6, y así hasta llegar a los datos 4 y 8.

En el caso en el que la función *DLML_Load* opera con el modo "*", recibe la información relacionada con un segmento a procesar y el nombre de un archivo. Esta función trabaja también sobre dos archivos de entrada, por lo que el segmento a procesar contiene renglones del primer archivo, y el segundo parámetro es el nombre del segundo archivo a procesar (útil para aplicaciones como la multiplicación de matrices). Una vez que se recibe ésto, este modo permite hacer una lectura por renglones para el primer archivo, y una lectura por columnas para el segundo (ver Figura 4.7).

El funcionamiento del modo "w" es parecido al del modo "+", ya que permite hacer una lectura de datos por renglones, sin embargo, a diferencia del modo "+", el modo "w" recibe un segmento a procesar. Posteriormente, este modo permite leer de forma secuencial cada uno de los elementos de los renglones a procesar, por ejemplo, retomando la Figura 4.7 (con este modo de operación), si tuvieran que leerse los dos primeros renglones del archivo 01, se leería el dato 1, posteriormente el 2, y así hasta llegar al elemento 10.

Como se mencionó inicialmente, la función *DLML_Load* hace la creación de items de lista conforme realiza la lectura de datos en los segmentos de entrada, sin embargo, la creación de items depende de los modos de funcionamiento. En los primeros 2 modos los archivos son vistos como matrices, lo que les permite incluir información adicional en la creación de items. Cuando se utiliza el modo "+", además de que cada item contiene los datos que fueron leídos, contienen la posición de la que se leyeron, por ejemplo, retomando la Figura 4.7 (con este modo de operación), si tuviera que leerse el primer renglón de los archivos 01 y 02, el primer item tendría los datos 1 y 5, y la coordenada (0,0), posteriormente el segundo item tendría los datos 2 y 6, y la coordenada (0,1), y así hasta llegar al item con los datos 4 y 8, y la coordenada (0,3).

El modo "*" es utilizado en la aplicación de la multiplicación de matrices, donde se obtiene cada componente de la matriz resultante multiplicando renglón por columna. Por ésta razón, la creación de cada item con este modo de operación contiene los datos de un

renglón del primer archivo, los datos de una columna del segundo archivo, y la posición que se desea obtener, por ejemplo, retomando la Figura 4.7 (con este modo de operación), si tuviera que calcularse el primer renglón de la matriz resultante, el primer item tendría los datos del primer renglón, de la primera columna, y la coordenada (0,0), posteriormente el segundo item tendría los datos del segundo renglón, de la segunda columna, y la coordenada (0,1), y así hasta llegar al item con los datos del cuarto renglón, de la cuarta columna, y la coordenada (0,3).

A diferencia del los dos modos de operación anteriores, la creación de cada item con el modo "w" incluye sólo el dato leído, por ejemplo, retomando el archivo 01 de la Figura 4.7 (con este modo de operación), si tuviera que leerse el primer renglón, el primer item tendría sólo el dato 1, el segundo item el dato 2, y así hasta llegar al cuarto item con el dato 4.

Independientemente del modo de operación, la función *DLML_Load* regula la creación de items generada por la lectura de los segmentos de entrada. Para esta labor, la función *DLML_Load* se encarga de consultar la memoria disponible y el umbral de memoria cada vez que va a crear un item de lista, por lo que si se procesan segmentos cuya creación de items sobrepasa la cantidad de memoria disponible, esta función se encarga de detener la lectura de datos y creación de items, para permitir el procesamiento de los items que ya se encuentra en memoria. No obstante, una vez que se libera memoria, la función debe ser llamada nuevamente por el proceso *Aplicación* que la ejecutó al inicio, para posteriormente continuar con la lectura de los datos restantes y la respectiva creación de items. Ésto se lleva a cabo cada vez que se detecta que la memoria disponible sobrepasa el umbral de memoria, o hasta que se completa la lectura de datos de los segmentos.

4.3.4. Función DLML_Write

La función *DLML_Write* permite realizar la escritura de resultados parciales en archivos de salida. Ésta función recibe como parámetros el puntero del archivo en donde se va a escribir, la coordenada del resultado calculado (opcional) y el resultado.

En la secciones 4.2 y 4.3 se describió el diseño de DLML-IO y las funciones que incluye, respectivamente. Como puede percibirse, el uso de DLML-IO permite procesar archivos cuyos tamaños sobrepasen las cantidades de memoria disponibles. Es importante mencionar que DLML-IO requiere de un sistema de archivos, ya que los procesos *Aplicación* que utilizan las

funciónes de esta implementación, pueden residir en diferentes nodos y además requerir el acceso a archivos en común para realizar lecturas de datos. La función *DLML_Write* puede también requerir el sistema de archivos para la escritura de resultados parciales, sin embargo, en este trabajo la escritura de resultados se almacena en los nodos locales.

Capítulo 5
Plataforma de experimentación y resultados

En este capítulo se presenta la plataforma de experimentación empleada en la comparación de MapReduce y DLML. Decidimos comparar con MapReduce debido a que es una de las herramientas más populares en el manejo de grandes volúmenes de datos. Esta comparación se basa en la programación de distintas aplicaciones con ambas herramientas, resaltando el modelo de programación, los resultados, las ventajas y desventajas de cada herramienta ante cada aplicación.

5.1. Aplicaciones

En esta sección se presenta la descripción de 4 aplicaciones empleadas para la comparación entre los dos modelos de programación MapReduce y DLML. Cabe señalar que para esta comparación se ha elegido una aplicación para cada modelo de programación en donde cada modelo es reconocido por obtener muy buenos resultados, y 2 aplicaciones más que han sido ya antes programadas en cualquiera de los dos. Las aplicaciones empleadas en este trabajo son: el conteo de la ocurrencia de cada palabra en un conjunto de documentos (donde MapReduce es famoso por los resultados que presenta), el problema de las N-Reinas (donde DLML presenta buenos resultados), la suma de matrices, y la multiplicación de matrices.

5.1.1. Conteo de Palabras

Esta aplicación se encarga de realizar el conteo de ocurrencias de cada palabra que se encuentra en un conjunto de archivos. Como se describió en la sección 2.2, en MapReduce el programador sólo tiene que especificar una función *map* y una función *reduce*. Para esta

aplicación, la función *map* toma como entrada un par del tipo llave/valor, donde la llave es el desplazamiento de archivo donde comienza el renglón a procesar, y el valor es la cadena completa de caracteres encontrada en el renglón. La función *map* se encarga de dividir la cadena de caracteres en palabras, y genera un par intermedio por cada palabra en el renglón, donde cada par establece la palabra obtenida como la llave, y un número de ocurrencia "1" como el valor, generando así un conjunto de pares intermedios por cada par de entrada.

Una vez que se aplica la función *map* a cada uno de los pares de entrada, la plataforma MapReduce se encarga de mezclar todos los valores asociados a una misma llave. Posteriormente la función *reduce* se encarga de sumar todos los elementos en la lista de valores de cada par, y una vez que termina la suma emite un par final, donde la llave del par es una palabra, y el valor es el número de ocurrencias de la palabra. Cabe mencionar que los pares de salida generados por la operación *reduce* son almacenados en el sistema de archivos.

En DLML-IO esta aplicación requiere mayor trabajo por parte del programador. La Figura 5.1 muestra el ejemplo del flujo que toma esta aplicación al procesar un archivo con DLML-IO. En este ejemplo se considera un sistema con 2 procesos *Aplicación* y 2 procesos *DLML*, sin embargo, para efectos de simplicidad sólo se muestran los procesos *Aplicación*. Inicialmente el proceso $Aplicación_0$ es el encargado de llamar a la función *DLML_fileList* para enlistar los archivos contenidos en el directorio a procesar, sin embargo, como sólo se tiene un archivo en este ejemplo, se omite la llamada a esta función y se aplica la función *DLML_distributeOffsets* al archivo (paso 1). Una vez ésto, los procesos *Aplicación* obtienen la información relacionada con un renglón a procesar, e inmediatamente ejecutan la función *DLML_Load* con el modo de operación "w" (paso 2). Este modo de operación permite crear un item por cada palabra leída en el renglón, de manera que cuando se termina se hacer la lectura de las palabras en el rengón a procesar, cada proceso *Aplicación* obtiene una lista de palabras (paso 3).

Cuando termina la creación de items de lista, los procesos *Aplicación* se encargan de obtener cada item en su lista local y aplicarle una función hash a la palabra del item. La función hash regresa el identificador de un archivo (A) en el que debe almacenarse la palabra (paso 4), cabe mencionar que cada proceso *Aplicación* tiene un número de archivos equivalente al número de procesos *Aplicación* en el sistema. Una vez que las palabras son almacenadas, los archivos son transferidos a los procesos *Aplicación* (paso 5) tomando en cuenta el último dígito de cada archivo, por ejemplo, los archivos A00 y A10 son transferidos

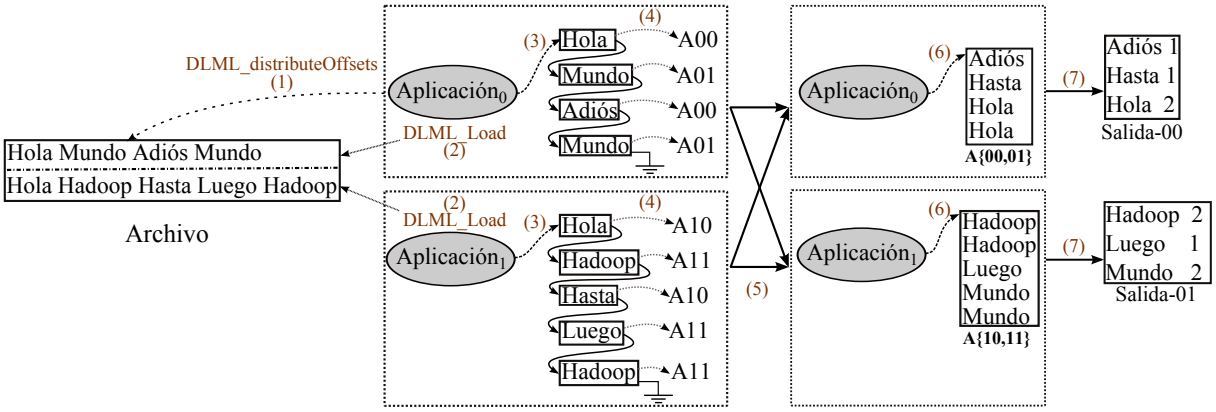

Figura 5.1: Flujo de la ejecución del conteo de palabras para DLML-IO.

al proceso *Aplicación*$_0$, y los archivos A10 y A11 son transferidos al proceso *Aplicación*$_1$. En cuanto procesos *Aplicación* reciben sus respectivos archivos, realizan un ordenamiento de las palabras contenidas en los archivos y posteriormente realizan el conteo de ocurrencias de cada palabra (paso 7), por lo que al final de la ejecución se obtiene el mismo tipo de salida que es generada en MapReduce, y se generan tantos archivos de salida como el número de procesos *Aplicación* en el sistema. A diferencia de MapReduce, en DLML-IO los archivos de salida son almacenados por los procesos *Aplicación*, en los discos locales de los nodos en que residen.

Cabe mencionar que los archivos en donde se almacenan inicialmente las palabras de los items, son transferidos del disco local (de los nodos donde residen) al sistema de archivos, y posteriormente a los discos locales de los correspondientes procesos *Aplicación* (paso 5). Esta trasferencia es realizada mediante el uso del comando "mv" de UNIX. La posterior ordenación de las palabras contenidas en los archivos es realizada mediante el comando "sort" (paso 6), y el conteo de ocurrencias es llevado a cabo con los comandos "uniq -c" y "awk" (paso 7). Básicamente el comando "uniq -c" se encarga del conteo de ocurrencias de cada palabra, y el comando "awk" de invertir el orden de la salida producida por el comando "uniq -c", ésto con el objetivo de producir el mismo tipo de salida que MapReduce, donde se tiene cada palabra y su número de ocurrencias. Es importante resaltar que los mejores mejores resultados fueron obtenidos con el uso de estos comandos.

5.1.2. N-Reinas

Este problema consiste en colocar N reinas en un tablero de ajedrez de tamaño N x N sin que se ataquen entre sí. Este problema puede modelarse mediante un árbol de búsqueda donde se van insertando las posibles soluciones, y se van descartando las que no lo son. Cada nivel de profundidad en el árbol representa una columna del tablero y los números que aparecen dentro del árbol corresponden al número del renglón donde se coloca una reina. La Figura 5.2 muestra las dos únicas soluciones para un tablero con N=4, y su árbol de búsqueda. En el primer nivel del árbol (columna 1), se coloca una reina por cada renglón, por lo que se generan 4 posibles soluciones a explorar. En el segundo nivel, se colocan reinas sólo en renglones donde no ataquen las reinas del primer nivel, y aquellas posibles soluciones que no cumplan con esta condición son eliminadas. Conforme se sigue el procedimiento pueden crearse nuevas soluciones a explorar, ya sea hasta llegar a una solución final o a anular una posible solución.

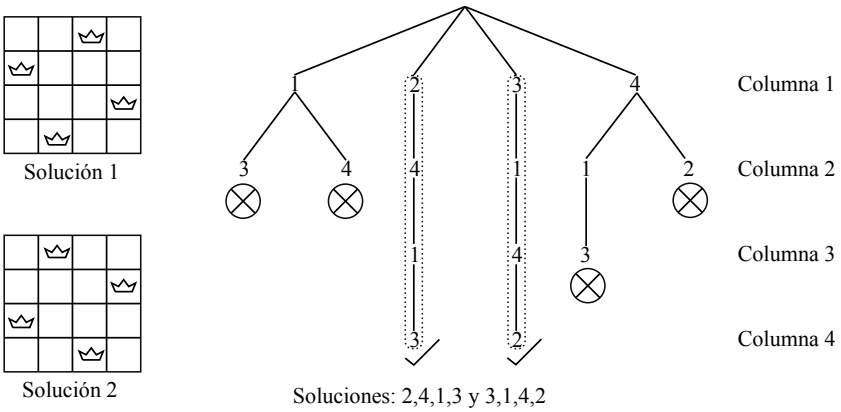

Figura 5.2: N-Reinas con un tablero de 4x4.

En MapReduce se pueden aplicar varias fases Map antes de una fase Reduce, y una vez que se aplica la fase Reduce nuevamente es posible aplicar varias fases Map. Ésta característica permite realizar programas más elaborados, uno de ellos es la aplicación de N-Reinas.

En MapReduce los niveles del árbol de búsqueda son calculados mediante fases Map, de manera que se calcula un nivel por cada fase Map. Ésto es posible debido a que la llave de cada uno los pares que reciben los procesos Map (en cada fase), permite representar una posible posible solución a explorar. Por otro lado, una vez que se llega al último nivel, cada proceso Map emite un par especial por cada solución encontrada, donde la llave del par

es una etiqueta (por ejemplo la palabra "#sol") y el valor es el número "1". Una vez que la plataforma mezcla los pares asociados a la misma llave, produce sólo un par, el cual es procesado durante la fase Reduce por un proceso de este tipo. Finalmente cuando el proceso Reduce termina la suma total de las soluciones encontradas (suma todos los "1" en la lista de valores del par), escribe el resultado en un archivo de salida.

La Figura 5.3 muestra el ejemplo del procesamiento de un tablero de 4x4 con 4 reinas. Como puede observarse, el archivo a procesar tiene 4 renglones y en cada renglón dos números separados por un espacio. El primer número indica la posible solución a explorar en la columna 1 (o el primer nivel), y el segundo número indica el número de reinas, el cual ayuda a los procesos Map a continuar con la exploración de posibles soluciones en los siguientes niveles.

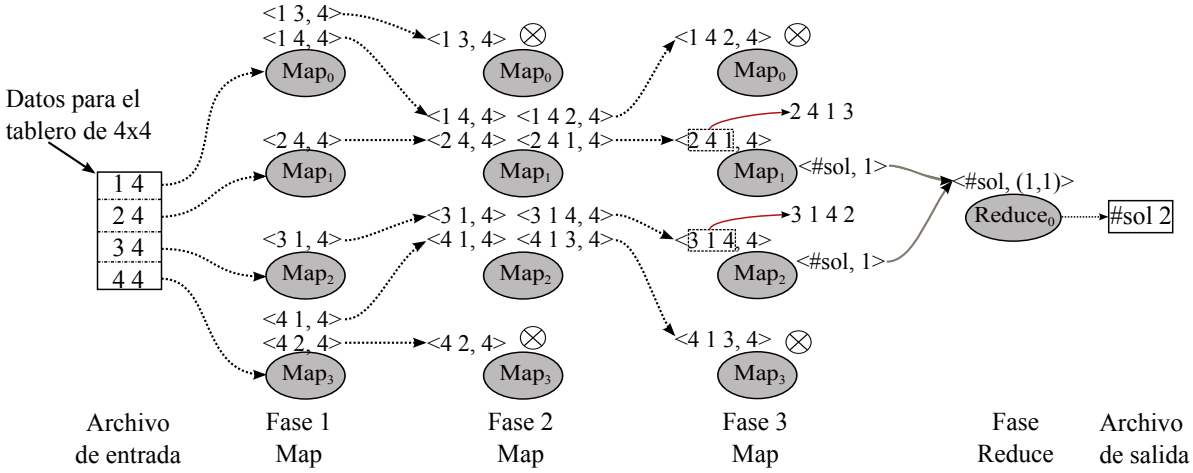

Figura 5.3: Flujo de la ejecución de la aplicación de reinas con un tablero de 4x4.

La plataforma se encarga de segmentar el archivo de entrada en renglones, y de distribuir los renglones entre los procesos Map. En la primera fase Map, los procesos reciben un par que contiene como llave, el desplazamiento de inicio del renglón a procesar, y como valor, el contenido del renglón. Cada proceso Map se encarga de leer el primer dato del renglón y de establecerlo en la columna 1 como primera solución a explorar. Posteriormente cada proceso Map se encarga de explorar las posibles soluciones de la siguiente columna (o nivel). Cabe mencionar que por cada posible solución encontrada, los procesos Map producen un par de salida, donde la llave del par contiene las columnas 1 y 2 que representan una posible solución a explorar, por ejemplo, del dato "1" del primer rengón se generan dos pares, uno con llave "1 3" y otro con llave "1 4" (y ambos con el valor "4"). Una vez que los procesos

Map terminan de calcular las siguientes posibles soluciones, la plataforma toma los pares de salida y los distribuye entre los procesos Map de la segunda fase.

En cada una de las siguientes fases, los procesos Map toman las llaves de los pares de entrada y se encargan de explorar las posibles soluciones del siguiente nivel. De la misma manera, por cada nueva posible solución, los procesos Map generan un par, sin embargo, cuando los procesos Map descubren que un par de entrada no es una solución lo descartan. Una vez que los procesos Map exploran el siguiente nivel (o columna), nuevamente la plataforma se encarga de distribuir los pares generados como posibles soluciones, entre los procesos Map para la siguiente fase. Cuando se llega a la última fase Map, los procesos de este tipo se encargan se tomar la llave de cada par a procesar y explorar el último nivel, sin embargo, a diferencia de las otras fases, donde se generaba un par por cada posible solución encontrada, en esta fase los procesos Map producen un par especial por cada solución encontrada, donde la llave del par es la palabra "#sol", y el valor es el número "1".

Cuando finaliza la última fase Map, la plataforma se encarga de mezclar todos los pares asociados a la misma llave, por lo que en este caso, se genera sólo un par para la fase Reduce, donde la llave del par es la palabra "#sol", y el valor del par es una lista de unos (cada uno representa una solución encontrada). Finalmente el proceso Reduce se encarga de sumar todos los valores "1" en la lista de par, lo que le permite calcular el número total de soluciones. Una vez que el proceso Reduce obtiene el número total de soluciones escribe la llave del par y el número de soluciones en un archivo de salida.

En general para esta aplicación, se requieren "n-1" fases Map para un número "n" de reinas, y una fase Reduce (con un proceso de este tipo). La cantidad de procesos Map depende de la infraestructura utilizada (ya que puede perjudicar los tiempos de respuesta).

En DLML el árbol de búsqueda puede ser representado mediante una lista de datos donde cada elemento de la lista representa una posible solución a explorar. En la figura 5.4 retoma un tablero de 4x4, y como se puede observar, inicialmente las posibles soluciones a explorar en la columna 1 son los renglones 1, 2, 3, y 4, por lo que se encuentra cada uno de ellos en un item de lista distinto. Posteriormente se obtiene el primer elemento de la lista (el item con el renglón 1 en la columna 1), el cual se descompone en dos posibles soluciones, (1,3) y (1,4), por lo que son también insertadas en la lista. Siguiendo el procedimiento se obtiene el item con (1,3) el cual al ser evaluado se descubre que no es una solución y queda descartado.

El procedimiento continua hasta que se vacía la lista, encontrando de esta manera todas las soluciones.

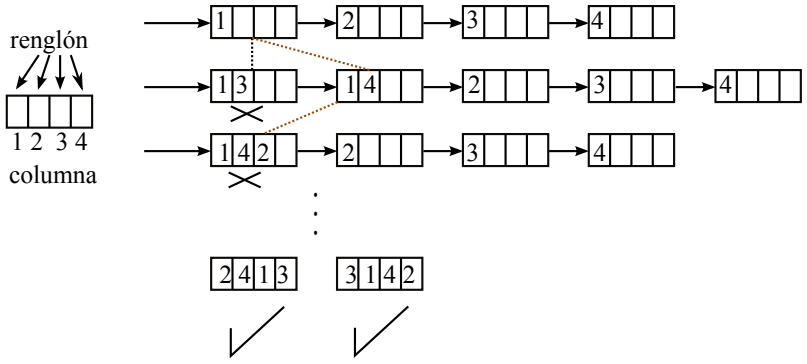

Figura 5.4: Inserción y eliminación de posibles soluciones en el árbol de búsqueda.

5.1.3. Suma de Matrices

En esta aplicación, para ambos modelos de programación, se supone que las matrices a procesar son cuadradas (es decir que tienen el mismo número de elementos por filas y columnas). La Figura 5.5a muestra un ejemplo de dos matrices de 3x3 a ser sumadas. En la suma de matrices cada coordenada de la matriz resultante se obtiene sumando la misma coordenada en las dos matrices, por ejemplo, si se deseará obtener la coordenada (0,0) de una matriz resultante C, se tendría que sumar la coordenada (0,0) de la matriz A, con la misma coordenada de la matriz B.

Para ambos modelos de programación se considera que las matrices se encuentran almacenadas en archivos (cada una en un archivo diferente). En MapReduce es necesario agregar información adicional a las matrices (ver Figura 5.5b), la cual indica el número de renglón para cada renglón en la matriz. La información adicional es utilizada por la función *map* para la creación de llaves de los pares intermedios, ya que en esta aplicación, cada una de las llaves se conforma de 2 componentes o elementos.

Como muestra la Figura 5.5c, cuando la función *map* recibe un renglón a procesar, lee el primer dato del rengón y lo establece como la primera componente para las llaves de los pares a generar, posteriormente esta función activa un contador para determinar la posición de los datos restantes en el rengón. Por cada uno de los datos restantes, la función *map* produce un par intermedio, donde el primer elemento de la llave es la componente calculada al inicio,

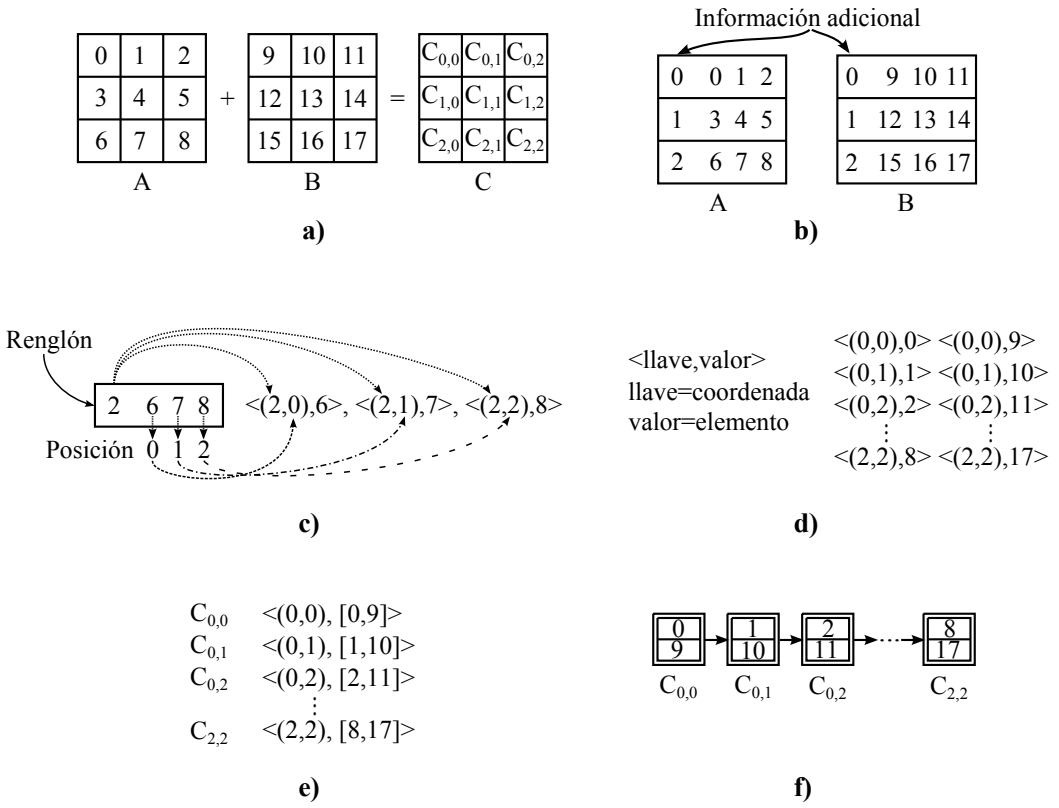

Figura 5.5: Suma de Matrices.

y el segundo elemento es la posición del dato leído. Cabe mencionar que el dato leído es establecido como el valor del par intermedio. Básicamente el objetivo de la función *map* es poner la misma llave a los valores de la matriz A y B ubicados en la misma coordenada (ver Figura 5.5d).

Finalmente la función *reduce* se encarga de procesar pares del tipo llave/valor, donde la llave representa una coordenada de la matriz resultante a calcular, y el valor es una lista de valores que contiene los datos para obtener la coordenada (ver Figura5.5e). En este caso, el procesamiento de los pares de entrada consiste en sumar los elementos o datos contenidos en la lista de valores.

Con DLML lo primero a realizar para esta suma de matrices es insertar los elementos de las matrices (A y B) como items de una lista. En la Figura 5.5f se observa la construcción de un item formado por la coordenada (0,0) de ambas matrices, así por cada posición hasta llegar al item formado por la coordenada (2,2). Al insertar todos los elementos, el procesamiento de los datos consiste en obtener un elemento de la lista a la vez y realizar la operación

correspondiente para obtener un elemento de la matriz resultante.

Como se mencionó en la sección 4.3, la extensión DLML-IO, a través de las funciones ya descritas, se encarga de la creación automática de items para esta aplicación, y permite procesar gradualmente la carga en caso de que sea lo suficientemente grande para sobrepasar la memoria disponible con la que se cuenta. A diferencia del DLML original, en esta extensión se hace uso de archivos, los cuales contienen cada matriz a procesar.

5.1.4. Multiplicación de Matrices

Como en el caso anterior, para esta aplicación se supone que las matrices a procesar son cuadradas. La figura 5.6a muestra un ejemplo de dos matrices de 2x2 a ser multiplicadas. En la multiplicación de matrices cada coordenada de la matriz resultante se obtiene multiplicando un renglón de la matriz A por una columna de la matriz B, por ejemplo, si se deseará obtener la coordenada (0,0) de una matriz resultante C, se tendría que multiplicar el valor de la coordenada (0,0) de la matriz A con el valor de la coordenada (0,0) de la matriz B, el valor la coordenada (0,1) de la matriz A con el valor de la coordenada (1,0) de la matriz B, y finalmente sumar los resultados de las multiplicaciones (ver Figura 5.6b).

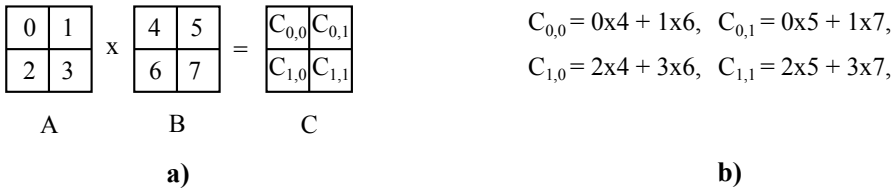

Figura 5.6: Multiplicación de Matrices de tamaño 2x2.

Al igual que en la aplicación anterior, para ambos modelos de programación se considera que las matrices se encuentran almacenadas en archivos (cada una en un archivo diferente). Nuevamente en MapReduce es necesario agregar información adicional a las matrices, la cual incluye 3 datos a cada renglón de las matrices (ver Figura 5.7a). El primer dato indica si el renglón a procesar pertenece a la matriz A o a la matriz B (el valor 1 denota que pertenece a la matriz A, y el valor 2 indica que pertenece a la matriz B), el segundo dato es el número del renglón en la matriz, y el tercer dato es el número de datos que tiene el renglón sin contar la información adicional. Cabe mencionar que la información adicional es requerida para la creación de llaves la cual se describe más adelante.

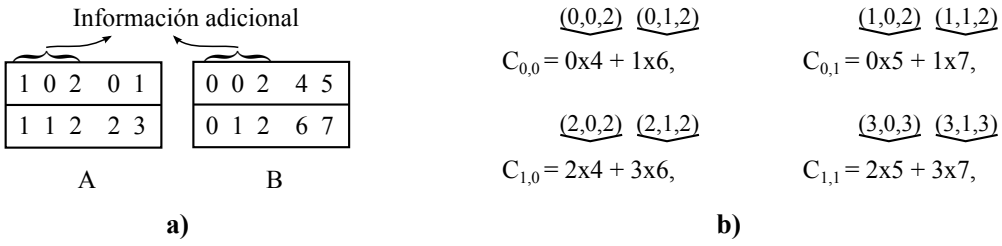

Figura 5.7: Multiplicación de Matrices en MapReduce.

Para obtener cada coordenada de la matriz resultante en la multiplicación de matrices se requieren hacer multiplicaciones y sumas. En MapReduce esta aplicación requiere de dos programas MapReduce, donde el primer programa realiza todas la multiplicaciones necesarias, y el segundo se encarga de realizar todas las sumas correspondientes.

En el primer programa, la función *map* se encarga de asignarle la misma llave a los datos que deben multiplicarse entre si (ver Figura 5.7b), no obstante, cada llave se compone de 3 datos. El primer componente de cada llave es un identificador de coordenadas, por ejemplo, el valor 0 indica que la multiplicación es requerida para calcular la coordenada (0,0), el 1 indica que la multiplicación es requerida para calcular la coordenada (0,1), y así hasta llegar al valor 3, que indica que la multiplicación es requerida para calcular la coordenada (1,1). La segunda componente de la llave es un contador de multiplicaciones, en este caso por cada coordenada a calcular se requieren dos multiplicaciones por lo que esta componente va del valor 0 al valor 1. Finalmente la tercera componente indica el número de elementos que contiene el renglón, en este caso 2.

La creación de llaves de la función *map* depende de si los renglones a procesar pertenecen a la matriz A o a la matriz B. Cabe mencionar que para este caso en donde las matrices son cuadradas, cada uno de los elementos en las matrices se multiplica por “n” números diferentes, donde “n” corresponde a una de las dimensiones de las matrices (ya sea renglones o columnas), por ejemplo, el valor 0 de la matriz A se multiplica por los valores 4 y 5 de la matriz B, el valor 1 de la matriz A se multiplica por los valores 6 y 7 de la matriz B, y así hasta llegar al valor 7 de la matriz B que se multiplica por los valores 1 y 3 de la matriz A (ver Figura 5.7b). Por ésta razón, independientemente de que la función *map* reciba renglones de la matriz A o B, se encarga de generar “n” pares intermedios por cada elemento de las matrices, en este caso por tratarse de matrices de 2x2 genera 2 pares por cada elemento.

Las Figuras 5.8a y 5.8b muestran la creación de llaves con un renglón que pertenece a la matriz A y con un renglón que pertenece a la matriz B, respectivamente. Cuando la función *map* recibe un renglón, realiza la lectura del primer dato para saber a que matriz pertenece el renglón, posteriormente registra el segundo dato como el número de renglón y el tercero como el número de datos en el renglón.

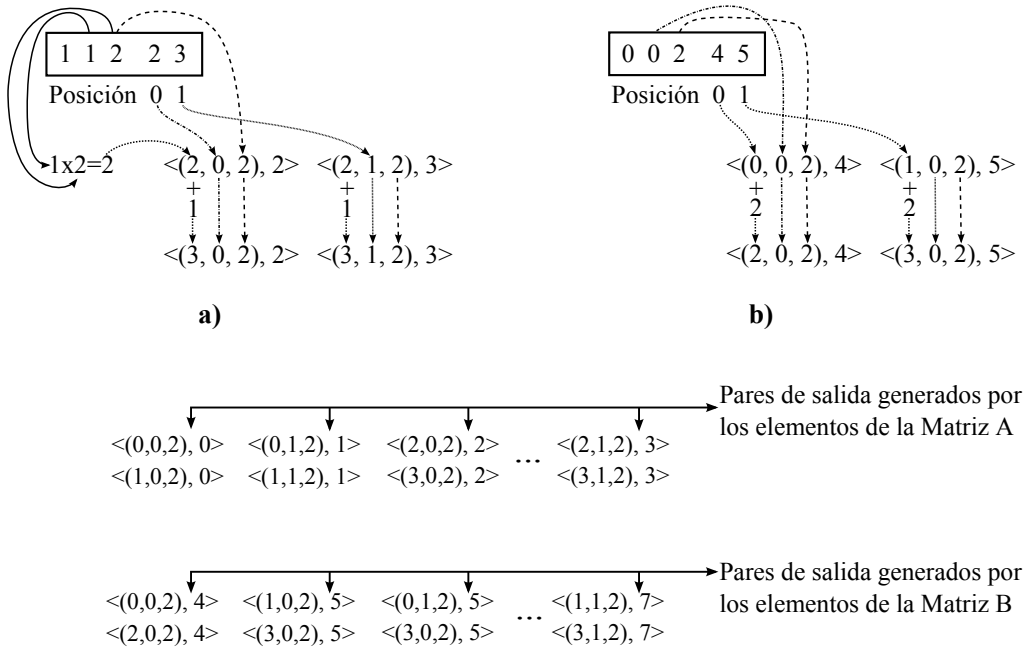

Figura 5.8: Procesamiento de los reglónes con la función map.

Cuando la función *map* lee la información adicional y determina que el renglón proviene de la matriz A, realiza la lectura de los siguientes datos en el renglón y produce dos pares de salida por cada uno (ver Figura 5.8a). El primer par de salida de cada dato es obtenido por la función *map* una vez que calcula su respectiva llave. Para ello, la función *map* multiplica el segundo dato del renglón con el tercero, y el resultado lo establece como la primer componente de la llave, posteriormente la función se encarga de asignar la posición del dato leído a la segunda componente de la llave, y finalmente establece el número de datos del renglón como tercera componente. Una vez ésto, la función *map* produce un par de salida intermedio con la llave descrita y el dato leído. Posteriormente, la función *map* procede a obtener el segundo par del dato leído, no obstante, para obtener este par, la función *map* modifica la primera

componente de la llave del par anterior. La modificación consiste en incrementar en una unidad la primer componente de la llave del primer par, y una vez ésto la función *map* genera el segundo par del dato leído.

En el caso en el que la función *map* lee la información adicional y determina que el renglón proviene de la matriz B, nuevamente realiza la lectura de los siguientes datos en el renglón y produce dos pares de salida por cada uno (ver Figura 5.8b). El primer par de salida de cada dato es obtenido por la función *map* una vez que calcula su respectiva llave. Para ello, la función *map* asigna la posición del dato leído a la primera componente de la llave, posteriormente la función se encarga de establecer el segundo dato del renglón a la segunda componente de la llave, y finalmente establece el número de datos del renglón como tercera componente. Una vez ésto, la función *map* produce un par de salida con la llave descrita y el dato leído. Posteriormente, la función *map* procede a obtener el segundo par del dato leído, no obstante, para obtener este par, la función *map* modifica la primera componente de la llave del par anterior. La modificación consiste en incrementar en dos unidades la primer componente de la llave del primer par, y una vez ésto la función *map* genera el segundo par del dato leído.

La Figura 5.8c muestra los pares intermedios de salida generados por la función *map* al procesar cada uno de los elementos de las matrices. Cabe mencionar que la plataforma se encarga de tomar los pares de salida y fusionar los pares asociados a la misma llave, de manera que como muestra la Figura 5.9a, los pares de entrada que recibe la función *reduce* tienen una llave y una lista de valores asociados a la llave del par. Finalmente la función *reduce* se encarga de procesar los elementos de la lista de valores de cada par, donde el procesamiento consiste en multiplicar entre si los elementos en la lista (ver Figura 5.9a).

El segundo programa MapReduce se encarga realizar las respectivas sumas en el procedimiento de la multiplicación de matrices. La función *map* de este programa recibe como entrada, los pares de salida de la función *reduce* del primer programa (ver Figura 5.9b). La función *map* genera un par de salida por cada par de entrada, y para ello, por cada par de entrada, esta función modifica la llave mediante la formula que se muestra en la Figura 5.9b, y emite un par con la nueva llave y con el mismo valor del par de entrada. La nueva llave representa la coordenada de la matriz resultante a calcular, y el valor es uno de los datos necesarios para calcular la coordenada del par. Una vez ésto, la plataforma fusiona los pares

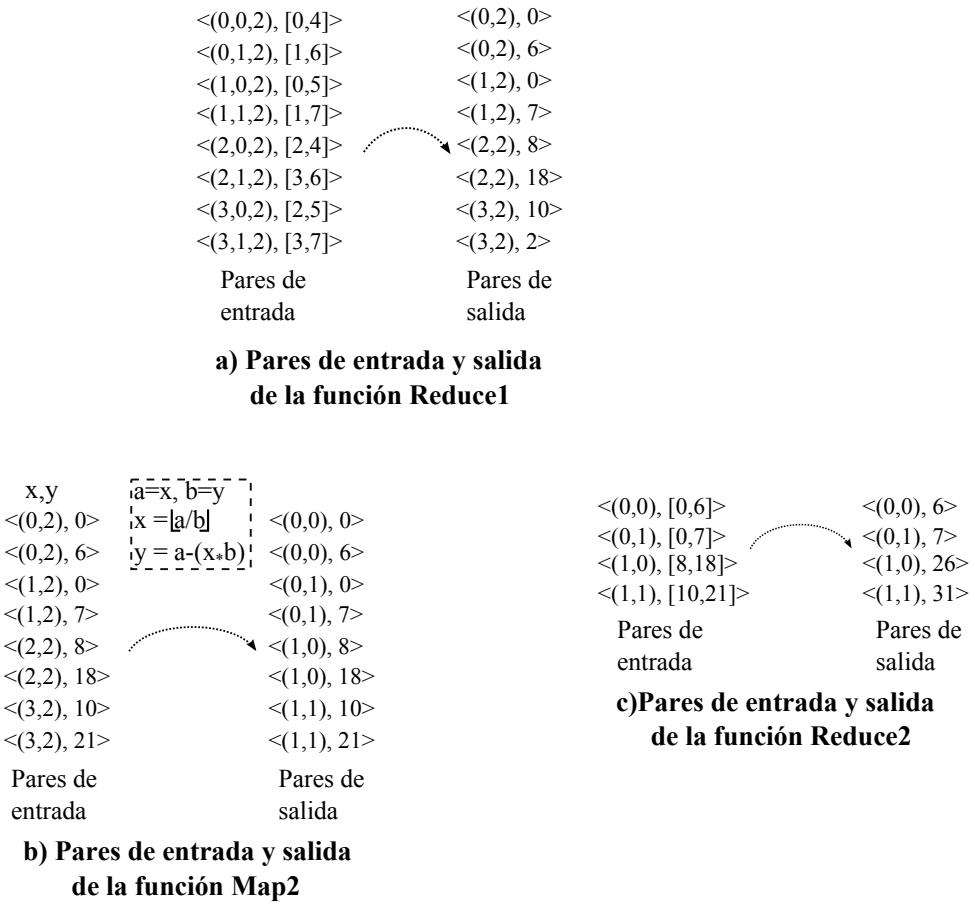

Figura 5.9: Pares de entrada y salida de las funciónes reduce_1, map_2 y reduce_2.

asociados a la misma llave, por lo que la función reduce recibe como entrada pares que tienen una llave y una lista de valores, donde la llave es la coordenada a calcular, y en la lista de valores se encuentran los datos a sumar para obtener el valor de la coordenada. Finalmente, la función *reduce* se encarga de sumar los datos en la lista de valores de cada par de entrada, obteniendo así una coordenada de la matriz resultante por cada par (ver Figura 5.9c).

En DLML lo primero a realizar para llevar a cabo esta multiplicación de matrices (ver Figura 5.10a), es organizar los elementos de las matrices como items de una lista, lo cual se logra guardando los valores de cada renglón y columna en un item, de manera que por ejemplo, los valores para la coordenada (0,0) son almacenados en un item (ver figura 5.10b, y así sucesivamente hasta llegar a la coordenada (2,2). Al organizar los valores de las matrices de esta forma, se procede a obtener un elemento a la vez de la lista para procesarlo y obtener

una de las coordenadas de la matriz resultante.

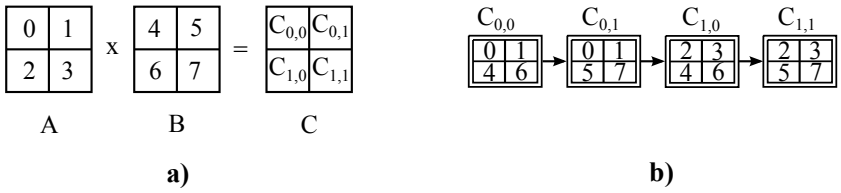

Figura 5.10: Multiplicación de matrices en DLML.

De igual manera que en la aplicación de la suma de matrices, la extensión DLML-IO, a través de las funciones descritas en la sección 4.3, permite hacer la creación automática de items de lista para esta aplicación, además de que cuando la creación de items de lista provoca que la cantidad de memoria disponible alcance el umbral de memoria, esta extensión permite procesar los datos ya almacenados en memoria, para posteriormente continuar con la creación de items asociados a los datos de las matrices. A diferencia del DLML original, ésta extension hace uso de archivos, los cuales en este caso contienen las matrices a procesar.

En la siguiente sección se describe la infraestructura utilizada en las pruebas para la comparación propuesta.

5.2. Infraestructura

La infraestructura empleada para la ejecución de las aplicaciones en esta comparación consta de un cluster que es propiedad de la Universidad Autónoma Metropolitana Iztapalapa (UAMI). El nombre del cluster es Pacífico y se localiza en el Laboratorio de Sistemas Distribuidos de la UAMI. Este cluster cuenta con 5 nodos, donde cuatro de éstos tienen 2 procesadores Dual core a 3 GHz, 2GB RAM y sistema operativo CentOS 5.4. El nodo restante tiene un procesador Quad core a 2.6 GHz, 4 GB RAM y sistema operativo CentOS 5.7. Las comunicaciones entre los nodos se llevaron a cabo a través de un switch Gigabit Ethernet. Para MapReduce utilizamos la plataforma libre Hadoop v. 0.20.2, y para DLML-IO utilizamos LAM/MPI v. 7.1.4.

En el cluster se utilizaron 3 sistemas de archivos (aunque no de forma simultanea), los cuales son HDFS, NFS (Network File System) [30] y PVFS2 (Parallel Virtual File System, Version 2) [33]. Ésto fue debido a que HDFS es por defecto utilizado por MapReduce, y tanto NFS como PVFS2 fueron utilizados para DLML, con el objetivo de obtener resultados que

permitan identificar las ventajas y desventajas que conlleva el hacer uso de estos distintos tipos de sistemas de archivos. En NFS las lecturas y escrituras a archivos son realizadas de forma centralizada en un sólo servidor. La última versión de este sistema de archivos es la versión 4.1, sin embargo, en este trabajo se utilizó la versión 3 [32]. Por otro lado, en PVFS2 las lecturas y escrituras a archivos son realizadas de forma distribuida entre los nodos del cluster. La versión de PVFS2 utilizada en este trabajo es la versión 2.8.2.

En este trabajo buscamos las mejores condiciones para cada modelo de programación. En MapReduce el mejor desempeño fue obtenido con 2 procesos Map y 2 procesos Reduce por nodo, ya que el desempeño empeoro al aumentar la cantidad de procesos debido a que la cantidad de memoria de cada nodo es insuficiente para cubrir la demanda de más procesos. En DLML-IO las mejores condiciones fueron obtenidas con 4 procesos *Aplicación* y 4 procesos *DLML* por nodo.

Es importante resaltar cuando un archivo es almacenado en HDFS es dividido por defecto en bloques de 64MB, y cada bloque es almacenado entre los diferentes nodos del cluster en una forma de round-robin. En PVFS2 un archivo es dividido en unidades de "stripes" con un tamaño por defecto de 64KB, los cuales son almacenados también entre los nodos del cluster en una forma de round-robin. Tanto el tamaño de bloques de HDFS, así como el de "stripes" de PVFS2, son parámetros que pueden modificarse.

Para nuestras pruebas con MapReduce utilizamos el tamaño de bloques por defecto, sin embargo, en DLML-IO modificamos el tamaño de los "stripes" de PVFS2, así como también la configuración de la instalación en cluster, con el objetivo de obtener mejores resultados. Básicamente se establecieron 2 tipos de configuraciones en el cluster, una en la que se almacenan los "stripes" entre los 5 nodos en el cluster, y otra en la que son almacenados en un sólo nodo. Cabe mencionar que también se realizaron pruebas modificando el tamaño de los "stripes" en ambas configuraciónes.

5.3. Resultados

En esta sección presentamos los resultados obtenidos en las pruebas de las aplicaciones empleadas para nuestra comparación. Iniciamos con la aplicación del Conteo de Palabras, en la cual MapReduce es popular por los resultados que obtiene.

5.3.1. Tiempos de ejecución del Conteo de Palabras

La Tabla 5.1 muestra los tiempos de ejecución en segundos de la aplicación del Conteo de Palabras para ambos modelos de programación. Para nuestras pruebas utilizamos algunos archivos de Freebase [34], y como puede observarse, llegamos a procesar hasta un volumen 16 GB de datos, lo que equivale al doble de la suma de memoria RAM de los 4 nodos utilizados en esta aplicación, ya que cada uno cuenta con 2 GB en RAM.

Como se mencionó en la sección 5.2, PVFS2 permite modificar los tamaños de los "stripes" y la configuración del sistema de archivos en el cluster. Cabe mencionar que se realizarón más pruebas de DLML-IO con PVFS2 donde se modificó la configuración del cluster, sin embargo no se obtuvieron mejoras significantes, por lo cual sólo se incluyen los resultados de la configuración en la que se instala el sistema de archivos sobre los 5 nodos del cluster, y se conservan los tamaños de "stripe" por defecto (64KB).

Volumen de datos (GB)	MapReduce	DLML-IO_pvfs2	DLML-IO_nfs
2	1097	1436	1045
4	2495	3522	2439
6	4453	5305	3912
8	6202	7768	5768
10	8044	9768	7686
12	10123	11803	8746
14	11901	13715	10388
16	14413	15757	12861

Tabla 5.1: Tiempos de ejecución en segundos obtenidos de DLML-IO y MapReduce en el Conteo de Palabras.

Los resultados muestran que DLML-IO con NFS, redujo los tiempos de respuesta con respecto a los de MapReduce e inclusive los de DLML-IO con PVFS2. El tiempo de ejecución para el procesamiento de 16 GB, mostró que DLML-IO con NFS obtuvo una reducción en los tiempos de respuesta de un 10.8 % en comparación con los de MapReduce.

En la Figura 5.11 se pueden observar grŕaficamente los resultados obtenidos esta aplicación. El eje X representa la cantidad de datos procesados, los cuales van de 2 a 16 GB. El eje Y corresponde al tiempo de respuesta obtenido en segundos.

DLML-IO con PVFS2 presentó un incremento en los tiempos de ejecución debido a la

sobrecarga de peticiones de acceso a datos generadas, ésto debido a que los datos se encuentran distribuidos entre los nodos del cluster. MapReduce presenta una situación similar, sin embargo, pensamos que la infraestructura y la escala de las pruebas puede ser aún pequeña para que HDFS y PVFS2 muestren mejores resultados.

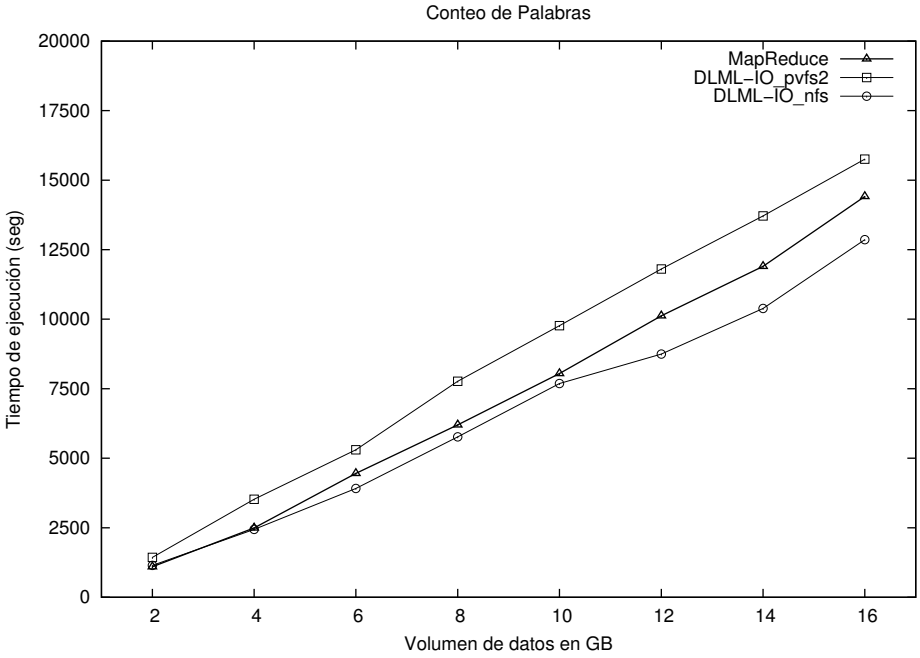

Figura 5.11: Tiempos de ejecución en segundos de DLML-IO y MapReduce en el Conteo de Palabras.

5.3.2. Distribución de carga del Conteo de Palabras

En esta aplicación el trabajo realizado puede dividirse en una fase Map y en una fase Reduce. La fase Map se encarga de distribuir las palabras contenidas en los archivos, entre los diferentes nodos del sistema, y una vez ésto, la fase Reduce se encarga de realizar el conteo de ocurrencias de cada palabra encontrada en los respectivos nodos. Estas dos fases se encuentran tanto en MapReduce como en DLML-IO como se describió en la sección 5.1.1, no obstante, las Figuras 5.12 y 5.13 muestran los datos procesados por las fases Map y Reduce respectivamente.

La Figura 5.12 muestra la cantidad de datos procesados por cada uno de los 4 nodos esclavos en el cluster en la fase Map. Como puede observarse, existe una desigualdad de

carga en ambos modelos de programación, ya que el nodo con identificador 0 procesa una mayor cantidad de datos en ambos casos. DLML-IO con PVFS2 y MapReduce, presentan balances donde el nodo 0 procesa una mayor cantidad de datos, el nodo 2 procesa la menor cantidad de datos, y la diferencia de la carga de trabajo entre los nodos 1 y 3 está ligeramente desplazada hacia el nodo 3. Una razón por la cual el nodo 2 procesa la menor cantidad de datos, es debido a que la mayor parte de los datos que tiene que procesar pueden encontrarse en otros nodos, lo que provocaría que este nodo pase la mayor parte del tiempo de ejecución haciendo peticiones de acceso a datos para posteriormente procesarlos. Por otro lado, el nodo 0 pudo haber procesado una mayor cantidad de datos debido a que su acceso a datos fue más rápido, lo que conllevaría a una mayor velocidad en el procesamiento de datos.

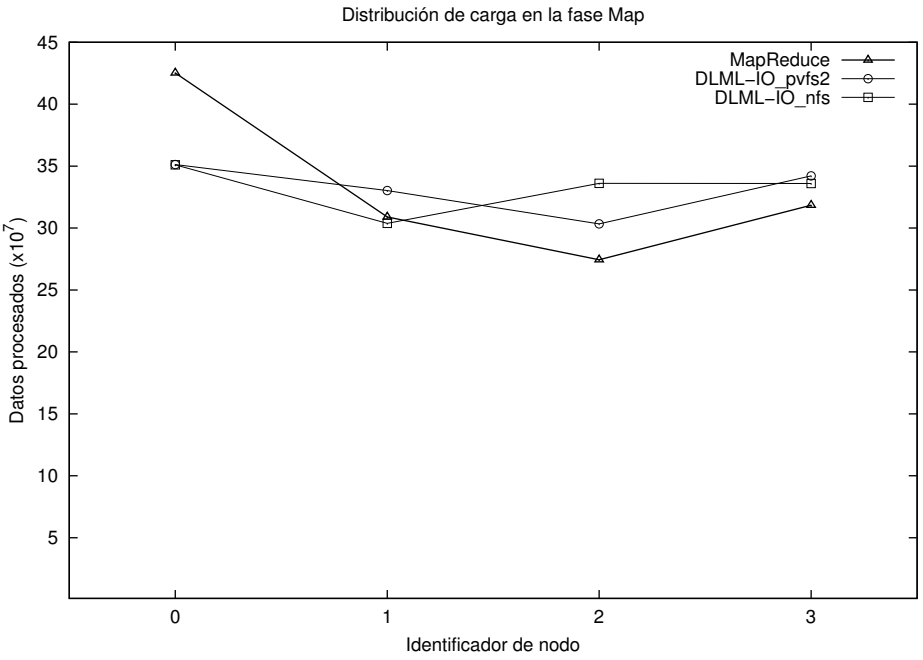

Figura 5.12: Distribución de carga de DLML-IO y MapReduce en la fase Map.

DLML-IO con NFS presenta balances en los que la diferencia de carga de trabajo de los nodos 0, 2 y 3, es ligeramente menor en comparación a los balances de MapReduce y DLML-IO con PVFS2, debido a que el acceso a datos de los nodos en NFS es más rápido que el de HDFS y PVFS2. Ésto permite a los nodos procesar datos a una velocidad similar, no obstante, el nodo 1 pudo haber procesado la menor cantidad de datos debido a que como se mencionó en la descripción de esta aplicación, los segmentos de los archivos de entrada

contienen renglones, sin embargo puede ocurrir que el número de palabras que contiene un renglón sea mucho mayor a la cantidad de palabras en otros renglones, lo que provocará que el procesamiento de un sólo renglón tome un mayor tiempo que el de varios, lo que pudo haber afectado la cantidad de datos procesados del nodo 1.

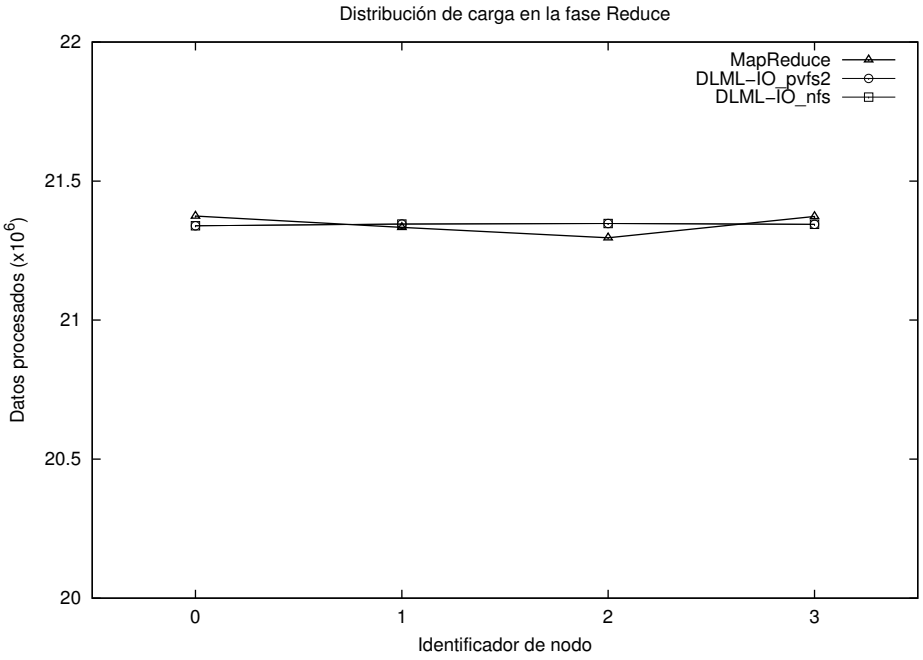

Figura 5.13: Distribución de carga de DLML-IO y MapReduce en la fase Reduce.

La Figura 5.13 muestra la cantidad de datos procesados por cada uno de los 4 nodos esclavos en el cluster en la fase Reduce. Como puede observarse, tanto DLML-IO con NFS, así como DLML-IO con PVFS2, obtienen los mismos balances debido a que la distribución de palabras entre los nodos para esta fase, se lleva a cabo mediante la misma función hash en esta aplicación. En MapReduce se puede observar una ligero aumento de carga de trabajo en el nodo 0 y una decaída en el nodo 2. Ésto puede deberse al mecanismo de tolerancia a fallos de MapReduce, ya que establece un tiempo promedio para cada tarea a procesar y una vez que se rebasa el tiempo la plataforma crea una replica en otro nodo.

5.3.3. Tiempos de ejecución de N-Reinas

La Tabla 5.2 muestra los tiempos de ejecución en segundos de la aplicación de N-Reinas para ambos modelos de programación. Para nuestras pruebas utilizamos de 13 a 16 reinas.

Cabe mencionar que los tableros para cada número de reinas son almacenados en archivos con el formato descrito en la sección 5.1.2.

En los resultados presentamos los tiempos de MapReduce, los de DLML-IO con NFS y los de DLML-IO con PVFS2, donde PVFS2 es instalado sobre los 5 nodos del cluster con los valores de los "stripes" por defecto. Cabe mencionar que en esta aplicación, los datos a procesar son generados de forma dinámica a tiempo de ejecución, de manera que el sistema de archivos es mínimamente usado, ya que solo se utiliza cuando se lee el tablero de entrada y cuando se escribe el número total de soluciones.

Número de Reinas	MapReduce	DLML-IO_pvfs2	DLML-IO_nfs
13	45	0.54	0.35
14	168	3	3
15	892	12	12
16	8019	80	79

Tabla 5.2: Tiempos de ejecución en segundos obtenidos de DLML-IO y MapReduce en N-Reinas.

Los resultados muestran que tanto los tiempos de DLML-IO con NFS, así como los de DLML-IO con PVFS2 son muy similares, debido a que el uso del sistema de archivos es mínimo. Por otro lado puede observase que ambas variantes de DLML-IO obtuvieron tiempos de respuesta menores a los de MapReduce. El tiempo de ejecución para el procesamiento de 16 reinas, mostró que DLML-IO con NFS obtuvo una reducción en los tiempos de respuesta de un 99 % en comparación con los de MapReduce.

En la Figura 5.14 se pueden observar grŕaficamente los resultados obtenidos esta aplicación. El eje X representa el número de reinas procesadas, las cuales van de 13 a 16. El eje Y corresponde al tiempo de respuesta obtenido en segundos.

MapReduce presentó un incremento en los tiempos de ejecución debido a que como se mencionó anteriormente, en esta aplicación los datos a procesar son generados de forma dinámica, lo que provoca que algunos datos tarden más en procesarse que otros. Ésto ocasiona que la plataforma active la réplica de tareas al detectar que el procesamiento de datos sobrepasa el tiempo promedio que establece durante la ejecución de la aplicación. Otra razón por la cual MapReduce eleva rápidamente los tiempos de ejecución, es que los datos que se

van generando son tan grandes que son escritos en archivos por la plataforma (en general, para la mayoría de las aplicaciones sucede ésto cuando la cantidad de datos a procesar es muy grande), situación que es diferente en DLML-IO para esta aplicación, ya que los datos que van generándose gradualmente y se encuentran en memoria RAM.

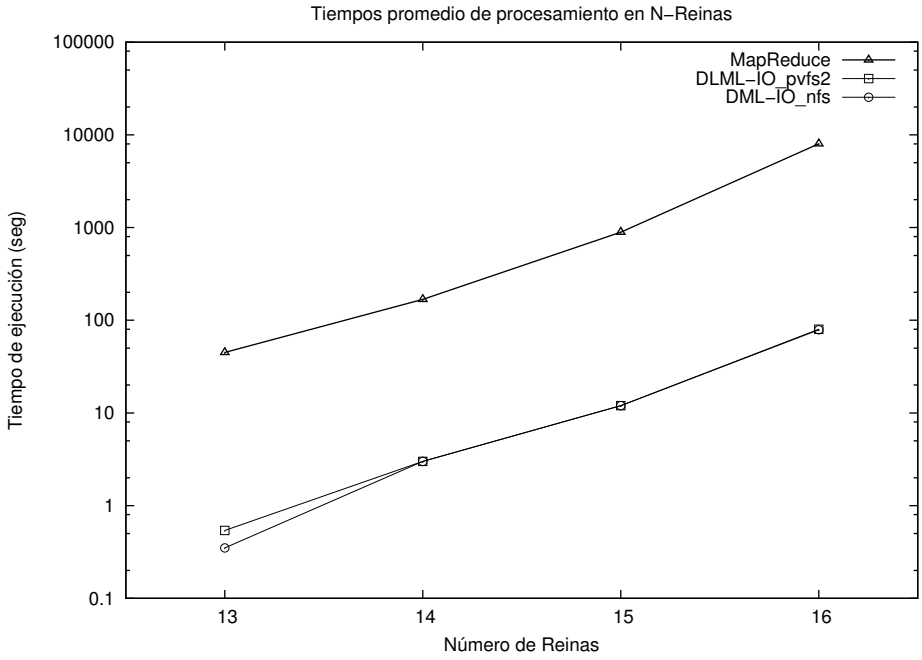

Figura 5.14: Tiempos de ejecución en segundos de N-Reinas con MapReduce y DLML-IO.

5.3.4. Distribución de carga de N-Reinas

La Figura 5.15 muestra la distribución de carga para la aplicación N-Reinas (con 16 reinas), en cada uno de los 4 nodos esclavos del cluster, para ambos modelos de programación. Como puede observarse, DLML-IO con NFS y con PVFS2 tienen similarmente la misma distribución de carga, ya que en este tipo de aplicaciones (dinámicas) DLML presenta un buen desempeño por la forma en la que realiza el balance de carga. En el caso de MapReduce, los nodos 0 y 1 tienen ligeramente una mayor carga de trabajo, lo cual puede deberse a que como ya se mencionó anteriormente, en esta aplicación los datos a procesar pueden o no generar más datos, sin embargo, cuando los datos generan más datos, el tiempo promedio que establece la plataforma para el procesamiento de una tarea es violado, lo que ocasiona que la plataforma haga réplicas de las tareas en otros nodos del cluster (en este caso en los

nodos 0 y 1).

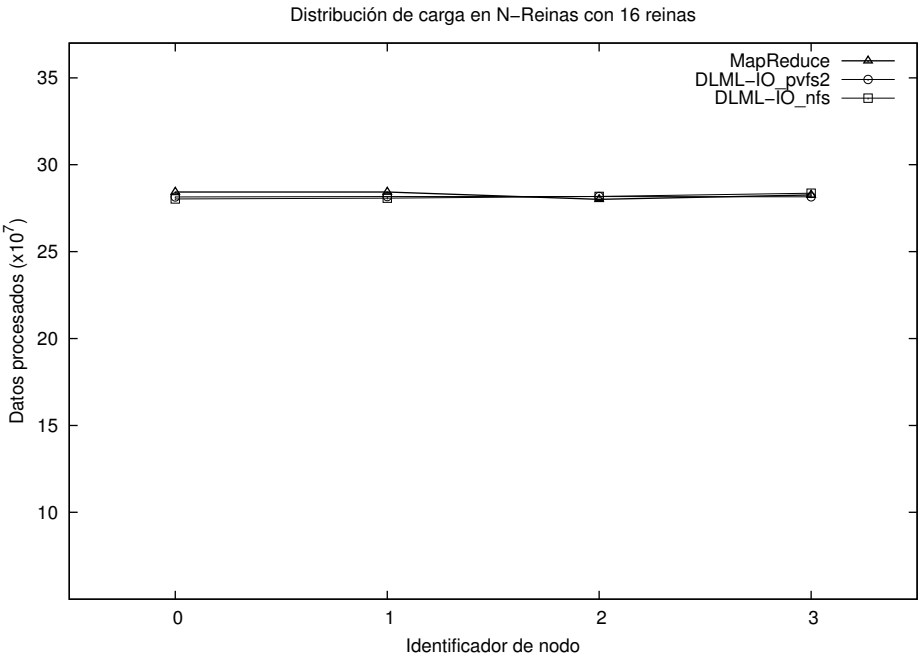

Figura 5.15: Distribución de carga de DLML-IO y MapReduce en N-Reinas con 16 reinas.

5.3.5. Tiempos de ejecución de la Suma de Matrices

La Tabla 5.3 muestra los tiempos de ejecución en segundos de la aplicación de la Suma de Matrices para ambos modelos de programación. Para nuestras pruebas utilizamos matrices contenidas en archivos (un archivo por cada matriz), con tamaños de 5000x5000 hasta tamaños de 40000x40000. Cabe mencionar que los archivos de las matrices A y B de 40000x40000, tienen un tamaño aproximado de 16 y 18 GB respectivamente.

A diferencia de los tiempos presentados en la aplicación anterior, en esta Tabla incluimos algunos de los resultados que obtuvimos al modificar los tamaños de los "stripes" de PVFS2 en la configuación del cluster (donde el sistema de archivos está instalado sobre los 5 nodos del cluster). Por lo tanto, en los resultados de DLML-IO se muestra una columna con el nombre SDU (Stripe Definido por el Usuario), otra con el nombre de pvfs2_SDU, y dos más con el nombre de pvfs2 y nfs respectivamente. La columna SDU contiene los tamaños de los "stripes" con los que se obtuvieron los mejores resultados, la columna de pvfs2_SDU muestra los tiempos de respuesta obtenidos con el SDU, la columna pvfs2 muestra los tiempos

obtenidos con los "stripes" por defecto (de 64 KB), y la columna de nfs presenta los resultados de DLML-IO con este sistema de archivos.

Tamaño de las Matrices	MapReduce	DLML-IO			
		SDU	pvfs2_SDU	pvfs2	nfs
5000x5000	195	2MB	53	73	13
10000x10000	1420	2MB	222	276	83
15000x15000	3765	512KB	467	649	523
20000x20000	7831	256KB	1231	1682	1032
25000x25000	-	64KB	3564	3534	1676
30000x30000	-	128KB	4922	5552	2608
35000x35000	-	256KB	6135	8173	4003
40000x40000	-	128KB	9130	10915	5192

Tabla 5.3: Tiempos de ejecución en segundos obtenidos de DLML-IO y MapReduce en la Suma de Matrices.

La Tabla 5.3 no muestra los tiempos de MapReduce para las matrices de 25000x25000 en adelante, ya que los tiempos de ejecución para este modelo fueron elevándose rápidamente. Por otro lado, los resultados muestran que DLML-IO con NFS, redujo los tiempos de respuesta con respecto a los de MapReduce e inclusive los de DLML-IO con PVFS2 y con PVFS2_SDU. Tomando en cuentra el último resultado obtenido en MapReduce (en las matrices de 20000x20000), podemos percibir que DLML-IO con NFS logró una reducción en los tiempos de respuesta de un 86.8 % en comparación con los de MapReduce.

En la Figura 5.16 se pueden observar grŕaficamente los resultados obtenidos esta aplicación. El eje X representa los tamaños de las matrices (cuadradas) procesadas, los cuales van de los 5000 a los 40000. El eje Y corresponde al tiempo de respuesta obtenido en segundos en escala logarítmica.

Las pruebas de DLML-IO con PVFS2 y con PVFS2_SDU presentaron un incremento en los tiempos de ejecución debido a la sobrecarga de peticiones de acceso a datos generadas. MapReduce presenta una situación similar, lo cual pudo ocurrir debido a la sobrecarga de acceso a datos, ya sea para las lecturas o escrituras a archivos requeridas por la plataforma durante la la ejecución de esta aplicación (independientemente de la escritura de resultados). Por otro lado, las pruebas de DLML-IO con PVFS2_SDU muestran que no se obtiene una mejora significativa en los resultados al variar el tamaño de los "stripes" en PVFS2, ya que el

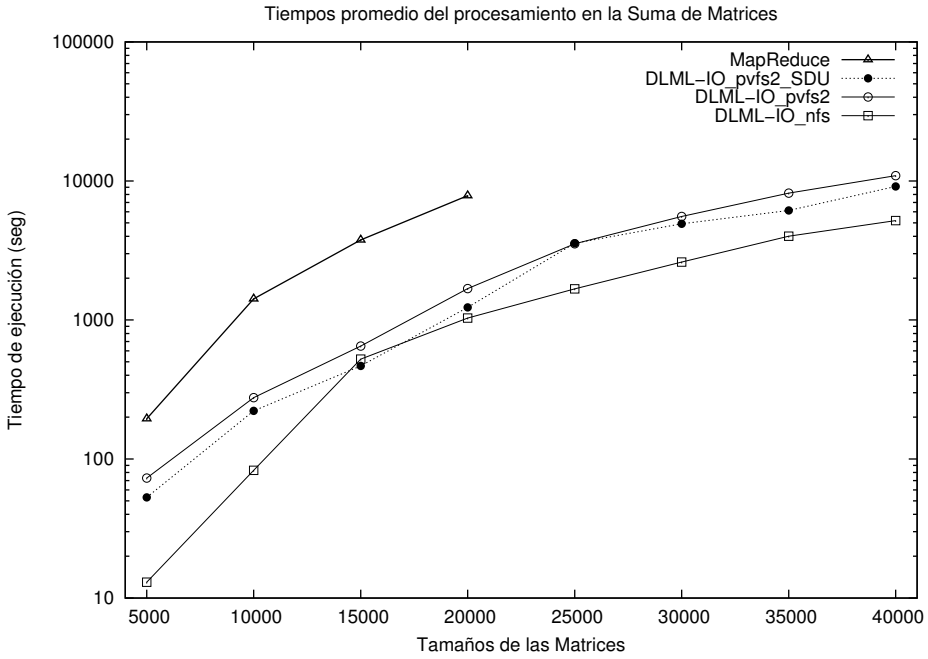

Figura 5.16: Tiempos de ejecución en segundos de la Suma de Matrices en DLML-IO y MapReduce.

mejor resultado en PVFS2_SDU se obtuvo en las matrices con tamaño de 15000x15000. Para los demás tamaños de matrices, DLML-IO con NFS prevaleció con los mejores resultados.

5.3.6. Distribución de carga de la Suma de Matrices

La Figura 5.17 muestra la distribución de carga para la suma de matrices de 20000x20000, en cada uno de los 4 nodos esclavos del cluster, para ambos modelos de programación. En el caso de MapReduce, estos datos son los pares procesados en cada nodo durante la fase Reduce, ya que en esta fase es donde se llevan a cabo tanto los cálculos de cada coordenada, así como la escritura de resultados. En el caso de DLML-IO, estos datos son los items procesados por los procesos *Aplicación* residentes en el nodo, ya que son los que se encargan de realizar los cálculos de cada coordenada, y de la escritura de resultados.

Como puede observarse, la distribución de carga en MapReduce es la mejor de todas. Ésto se debe a que el mecanismo de tolerancia a fallos de la plataforma, interfiere de forma mínima en la ejecución de la aplicación, ya que las matrices contienen el mismo número de elementos, lo que consigue que difícilmente se sobrepase el tiempo promedio (establecido por

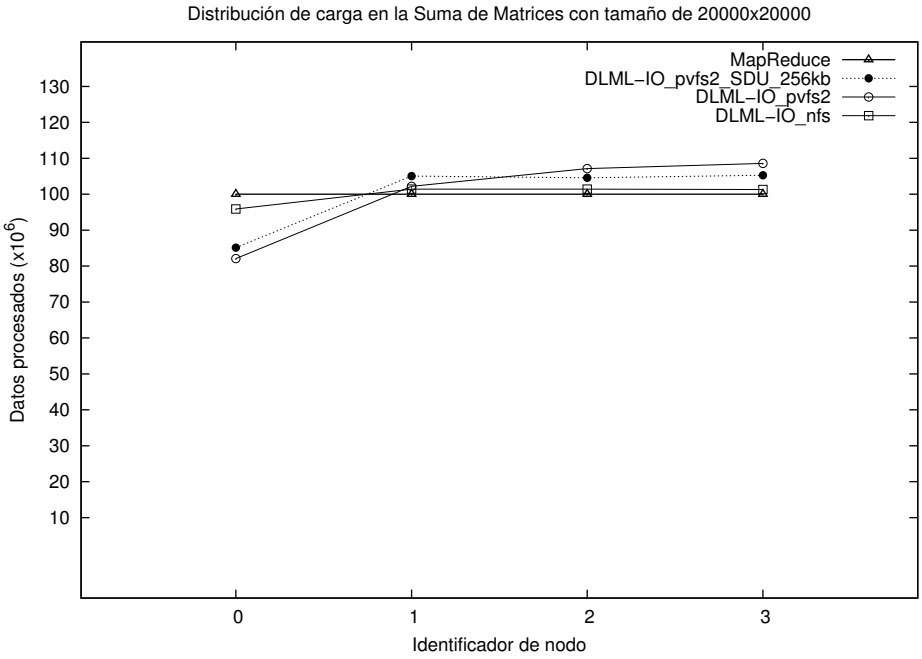

Figura 5.17: Distribución de carga de DLML-IO y MapReduce en las sumas de matrices de tamaño 20000x20000.

la plataforma) en que deben procesarse las particiones de los archivos de entrada. Al tener una interferencia mínima por parte del mecanismo de tolerancia a fallos, la plataforma puede realizar una mejor distribución de carga trabajo, situación que se observa en los balances de carga de la Figura 5.17.

En DLML-IO se presentan resultados con PVFS2 y NFS, sin embargo, para PVFS2 se presentan dos tipos resultados. El primer resultado con PVFS2 representa la cantidad de datos procesados obtenidos con PVFS2 y SDU, donde el tamaño de los "stripes" para esta aplicación fue de 256 KB (tamaño en donde se obtuvo menores tiempos de ejecución). El segundo resultado representa la cantidad de datos procesados obtenidos con el tamaño de los "stripes" de 64 KB (el valor por defecto).

Tanto DLML-IO con NFS, así como DLML-IO con PVFS2 y con PVFS2_SDU, presentan algo en común, y es la decaída de carga de trabajo en el nodo 0. Ésto se debe a que inicialmente el proceso $Aplicación_0$ (que reside en el nodo 0), se encarga de realizar la segmentación lógica de los archivos de entrada, y de la posterior distribución de la información asociada a los segmentos, de manera que una vez que realiza ésto, es el último en tomar la información de los segmentos que tiene que procesar. Ésto y el que este proceso sea el último en procesar la

carga de trabajo en cada redistribución de datos, puede provocar que durante la ejecución del balance de carga por parte de los los procesos *DLML*, de los demás procesos *Aplicación*, se identifique que el proceso $Aplicación_0$ es el que tiene mayor carga de trabajo, lo que le restaría carga de trabajo al nodo 0.

Finalmente puede identificarse que los balances de carga de DLML-IO con NFS son mejores que los de DLML-IO con PVFS2 y con PVFS2_SDU, ya que los resultados de PVFS2 con ambas alternativas se ven afectados por la sobrecarga de acceso a datos, principalmente PVFS2 con los tamaños de "stripes" por defecto.

5.3.7. Tiempos de ejecución de la Multiplicación de Matrices

La Tabla 5.4 muestra los tiempos de ejecución en segundos de la aplicación de Multiplicación de Matrices para ambos modelos de programación. Para nuestras pruebas utilizamos matrices contenidas en archivos (un archivo por cada matriz), con tamaños de 100x100 hasta tamaños de 1100x1100.

A diferencia de los tiempos presentados en la aplicación anterior, presentamos algunos de los resultados que obtuvimos al modificar los tamaños de los "stripes" de PVFS2 y la configuación del cluster. La modificación del sistema de archivos en el cluster consistió en almacenar todos los "stripes" en un único servidor, de manera que las lecturas y escrituras se llevarán a cabo de forma centralizada.

Como muestra la Tabla 5.4, dentro de DLML-IO se encuentran 4 columnas, una columna con el nombre SDU, otra con el nombre de pvfs2_CC_SDU, y dos más con el nombre de pvfs2 y nfs respectivamente. La columna SDU contiene los tamaños de los "stripes" con los que se obtuvieron los mejores resultados en la configuración centralizada de PVFS2, la columna de pvfs2_CC_SDU muestra los tiempos de respuesta obtenidos con el SDU en la configuración centralizada (CC), la columna pvfs2 muestra los tiempos obtenidos con los "stripes" por defecto (de 64 KB) en la configuración donde el sistema de archivos se instala en los 5 nodos del cluster, y finalmente la columna de nfs presenta los resultados de DLML-IO con este sistema de archivos.

La Tabla 5.4 no muestra los tiempos de MapReduce para las matrices de 900x900 en adelante, ya que los tiempos de ejecución para este modelo fueron elevándose rápidamente, así mismo tampoco se muestran los tiempos de DLML-IO con PVFS2 para las matrices de

Tamaño de las Matrices	MapReduce	DLML-IO			
		SDU	pvfs2_CC_SDU	pvfs2	nfs
100x100	56	128KB	0.82	0.75	0.7
200x200	140	512KB	10.94	44	11
300x300	354	1MB	55	1056	56
400x400	1675	2MB	112.94	3578	116
500x500	4035	2.5MB	291.83	8468	298
600x600	6506	5MB	641	16369	700
700x700	11971	7MB	1239	27980	1279
800x800	20651	8.5MB	1731	-	1850
900x900	-	10MB	2875	-	2996
1000x1000	-	15MB	4822	-	5252
1100x1100	-	17.5MB	7399	-	7834

Tabla 5.4: Tiempos de ejecución en segundos obtenidos de DLML-IO y MapReduce en la Multiplicación de Matrices.

800x800 debido a la misma razón. Por otro lado, los resultados muestran que la diferencia de tiempos de respuesta entre DLML-IO con NFS y DLML-IO con PVFS2_SDU_CC, es ligeramente mayor en DLML-IO con NFS. DLML-IO con PVFS2_SDU_CC redujo considerablemente los tiempos de respuesta con respecto a los de MapReduce, e incluso los de DLML-IO con PVFS2. Tomando en cuentra el último resultado obtenido en DLML-IO con PVFS2 (en las matrices de 700x700), podemos percibir que DLML-IO con PVFS2_SDU_CC logró una reducción en los tiempos de respuesta de un 89.6 % en comparación con los de MapReduce, y de un 95.6 % con respecto a los de DLML-IO con PVFS2. En cuanto a DLML-IO con NFS, se puede observar que DLML-IO con PVFS2_SDU_CC tiene ligeramente menores tiempos de respuesta, y en este tamaño de matrices, la mejora de DLML-IO con PVFS2_SDU_CC ante DLML-IO con NFS es de un 3.1 %.

En la Figura 5.18 se pueden observar grŕaficamente los resultados obtenidos esta aplicación. El eje X representa los tamaños de las matrices (cuadradas) procesadas, los cuales van de los 100 a los 1100. El eje Y corresponde al tiempo de respuesta obtenido en segundos en escala logarítmica.

Las pruebas de DLML-IO con PVFS2 y con MapReduce presentaron un incremento en los tiempos de ejecución debido a la sobrecarga de peticiones de acceso a datos. DLML-IO con PVFS2_SDU_CC presenta menores tiempos de respuesta en todas las pruebas (en incluso

menores a los de DLML-IO con NFS), ya el concentrar los "stripes" en un solo nodo y al aumentar su tamaño, se realizan lecturas contiguas de datos, lo que ayuda a disminuir el tiempo de procesamiento. Cabe mencionar que esta misma configuración fue probada en las pruebas anteriores, sin embargo, no se obtuvieron resultados como éstos.

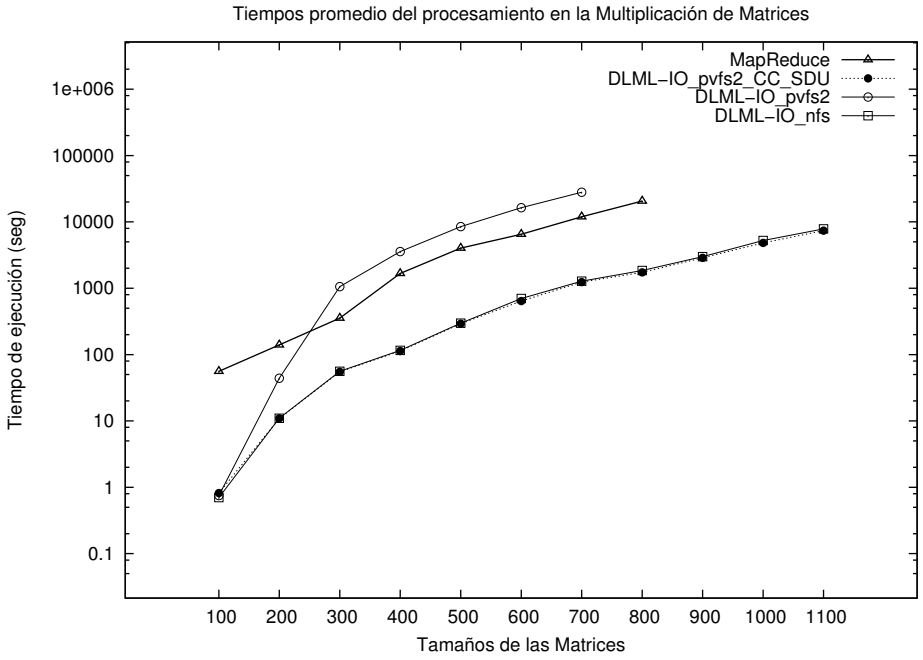

Figura 5.18: Tiempos de ejecución en segundos obtenidos de DLML-IO y MapReduce en la Multiplicación de Matrices.

5.3.8. Distribución de carga de la Multiplicación de Matrices

La Figura 5.19 muestra la distribución de carga para la multiplicación de matrices de 700x700, en cada uno de los 4 nodos esclavos del cluster, para ambos modelos de programación. En MapReduce se requieren dos programas para llevar a cabo esta aplicación, sin embargo, los datos que se muestran son los pares procesados en cada nodo durante la fase Reduce del segundo programa MapReduce, ya que en esta fase se calcula finalmente cada coordenada, y se realiza la posterior escritura de resultados. En el caso de DLML-IO, estos datos son los items procesados por los procesos *Aplicación* residentes en el nodo, ya que son los que se encargan de realizar los cálculos para cada coordenada, y de la escritura de resultados.

Nuevamente la distribución de carga en MapReduce es la mejor de todas. Como se mencionó en la aplicación anterior, ésto se debe a que el mecanismo de tolerancia a fallos de la plataforma interfiere de forma mínima en la ejecución de la aplicación, ya que los renglones del archivo que es generado por el primer programa MapReduce para esta aplicación, contienen la misma cantidad de datos, lo que nuevamente consigue que difícilmente se sobrepase el tiempo promedio (establecido por la plataforma) en que deben procesarse las particiones de los archivos de entrada. La interferencia mínima del mecanismo de tolerancia a fallos permite que la plataforma realice una mejor distribución de carga trabajo, situación que se observa en los balances de carga de la Figura 5.19.

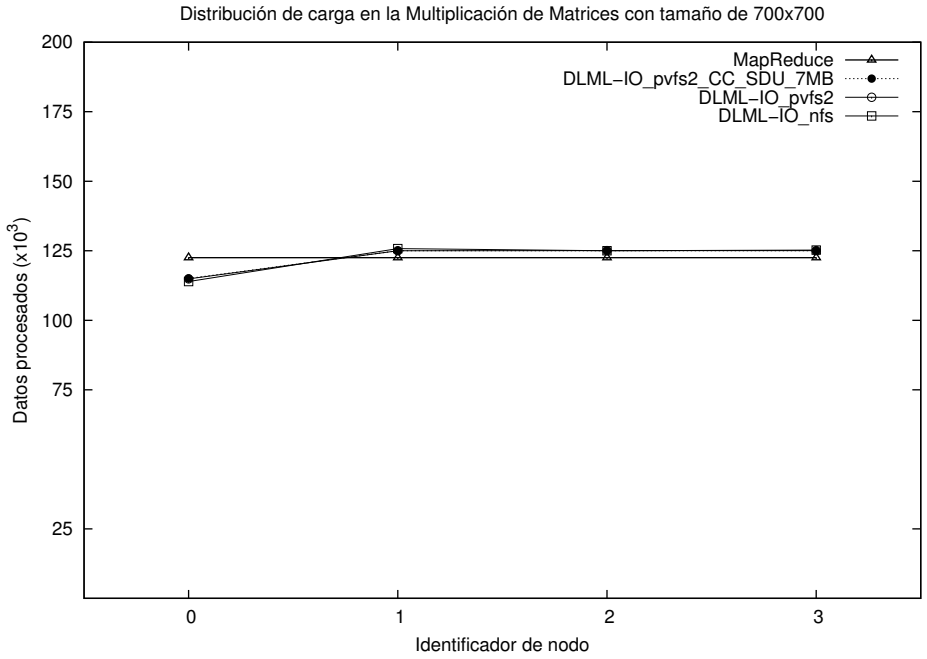

Figura 5.19: Distribución de carga de DLML-IO y MapReduce en la Multiplicación de Matrices con tamaño de 700x700.

Al igual que en la aplicación anterior, en DLML-IO se presentan tiempos de ejecución con PVFS2 y NFS, donde para PVFS2 se presentan dos tipos resultados. El primer resultado con PVFS2 representa la cantidad de datos procesados obtenidos con PVFS2_CC_SDU, donde el tamaño de los "stripes" para esta aplicación fue de 7MB (tamaño en donde se obtuvo menores tiempos de ejecución). El segundo resultado representa la cantidad de datos procesados obtenidos con el tamaño de los "stripes" de 64 KB (el valor por defecto) en la configuración donde el sistema de archivos se instala sobre los 5 nodos del cluster.

Nuevamente tanto DLML-IO con NFS, así como DLML-IO con PVFS2 y con PVFS2_SDU, presentan algo en común, y es la decaída de carga de trabajo en el nodo 0. Como se mencionó en la aplicación anterior, ésto se debe a que inicialmente el proceso $Aplicación_0$ (que reside en el nodo 0), se encarga de realizar la segmentación lógica de los archivos de entrada, y de la posterior distribución de la información asociada a los segmentos. Ésto y el que este proceso sea el último en procesar la carga de trabajo en cada redistribución de datos, puede provocar que durante la ejecución, el nodo 0 procese una menor carga de trabajo, ya que las peticiones para la redistribución de carga pueden llegar constantemente al proceso $Aplicación_0$, el cual reside en este nodo.

Finalmente puede identificarse que los balances de carga de DLML-IO en todas las versiones son muy similares, lo que puede ser ocasionado por el tamaño de las matrices, ya que en este caso, son relativamente pequeñas, de manera que el acceso a datos es más rápido que en la aplicación anterior, donde las matrices alcanzan tamaños de hasta 40000x40000.

En este capítulo se describieron las aplicaciones empleadas en nuestra comparación, así como también se presentaron los tiempos de ejecución y balances de carga obtenidos en cada una, no obstante, el siguiente capítulo presenta las conclusiones y el trabajo futuro.

Capítulo 6
Conclusiones y trabajo futuro

6.1. Conclusiones

En esta tesis se compararon los modelos de programación de DLML y MapReduce, no obstante, para esta comparación se propuso una extensión a la biblioteca DLML para el manejo de archivos, también llamada DLML-IO. Esta extensión permitió procesar grandes cantidades de datos a través del uso archivos, logrando cumplir el objetivo de procesar cantidades de datos que sobrepasan las capacidades de memoria RAM con las que se cuenta.

El diseño de la extensión para la biblioteca DLML hace uso de un umbral de memoria que le permite delimitar la creación de datos asociados a la información contenida en los archivos que se va leyendo. Esta extensión le permite a DLML procesar cantidades que eran imposibles de procesar con la versión original.

Para las pruebas se utilizaron 4 aplicaciones, una donde MapReduce es popular por los resultados que obtiene, otra donde DLML es popular por sus resultados, y dos más que ya han sido desarrolladas anteriormente en ambos modelos de programación. Las aplicaciones utilizadas son el Conteo de Palabras, N-Reinas, Suma de Matrices y Multiplicación de Matrices respectivamente.

Los resultados obtenidos con respecto a los tiempos de respuesta y la distribución de carga en cada aplicación fueron los siguentes:

Tiempos de respuesta

- En la aplicación del Conteo de Palabras, DLML-IO con NFS mostró los mejores tiempos de ejecución, debido a que la escala de las pruebas es relativamente pequeña para que los sistemas de archivos HDFS y PVFS2 muestren un mejor desempeño (aún con la

sobrecarga de peticiones de acceso a datos).

- En la aplicación de N-Reinas, DLML-IO con NFS y PVFS2, mostró los mejores tiempos de ejecución, ya que esta aplicación es una en las que DLML presenta buenos resultados debido a la forma en que realiza el balance de carga. MapReduce mostró un rápido crecimiento en los tiempos de respuesta debido a que la plataforma requiere hacer lecturas y escrituras a archivos durante la ejecución de la aplicación (generalmente en cualquier aplicación que procese grandes cantidades de datos), situación que es diferente en DLML-IO, ya que en esta aplicación los datos a procesar se generan gradualmente y se encuentran en memoria RAM.

- En la aplicación de Suma de Matrices, DLML-IO con NFS mostró los mejores tiempos de ejecución, debido a que nuevamente la sobrecarga de acceso a datos generada con los sistemas de archivos PVFS2 (con los diferentes tamaños de "stripe") y HDFS, provoca que se eleven los tiempos de respuesta, ya que la transferencia de datos que se encuentran localizados en distintos nodos conlleva ciertas cantidades de tiempo.

- En la aplicación de Multiplicación de Matrices, DLML-IO con PVFS2 mostró los mejores tiempos de ejecución en el caso en el que se utilizó una configuración centralizada y se fue variando el tamaño de los "stripes" en cada prueba. Básicamente DLML-IO con PVFS2 obtuvo los mejores resultados debido a que las matrices que se procesaron son relativamente pequeñas, y con la modificación establecida se consiguió realizar una mayor lectura contigua de datos, lo que permitió aumentar la velocidad de acceso a datos y con ello reducir los tiempos de ejecución. En el caso de MapReduce, y PVFS2 con la configuración distribuida y con los tamaños de los "stripes" por defecto, los tiempos de respuesta se vieron afectados por la sobrecarga de acceso a datos generada por la lectura de datos localizados en distintos nodos.

Distribución de carga

- En el Conteo de Palabras, los balances de carga tanto para la fase Map, así como para la fase Reduce, fueron mejores en DLML-IO con NFS. MapReduce se vio afectado por el mecanismo de tolerancia a fallos, ya que cuando los segmentos del archivo de entrada sobrepasaron el tiempo promedio de procesamiento (debido a que los segmentos tienen

diferentes cantidades de palabras), la plataforma se encargó de realizar réplicas de las tareas en otros procesos, los cuales en este caso se encontraban mayormente en otros nodos.

- En la aplicación de N-Reinas, tanto DLML-IO con NFS, así como DLML-IO con PVFS2, obtuvieron buenos balances de carga. MapReduce en particular tuvo un ligero incremento en 2 de los 4 nodos esclavos, debido a que esta aplicación es dinámica, lo que provocó que el mecanismo de tolerancia a fallos de la plataforma realizará réplicas de tareas en otros procesadores, en este caso, localizados en otros nodos.

- En la Suma y Multiplicación de Matrices MapReduce obtuvo los mejores balances de carga. Todos los resultados de DLML-IO se vieron afectados por una característica en común, la cual consistió en la decaída de la carga de trabajo en uno de los 4 nodos esclavos del cluster. Ésto se debió a que el proceso *Aplicación* que realiza la segmentación lógica de los archivos de entrada, y la posterior distribución de la información asociada a los segmentos, es el último (en cada redistribución de datos) en comenzar a procesar sus respectivos segmentos, lo que ocasionó que en el momento en que los demás procesos *Aplicación* consumieran su carga, se le identificara como el proceso con la mayor carga de trabajo, lo que redujo la cantidad de datos procesados del nodo en el que residió.

  En la Suma de Matrices los balances de DLML-IO con NFS con respecto a los demás resultados de DLML-IO, obtuvieron la mejor distribución de datos, ya que los demás resultados se vieron perjudicados por la sobrecarga de acceso a datos. Por otra parte, en la Multiplicación de Matrices los balances de DLML-IO en cualquiera de sus sistemas de archivos fueron similares, ya que el tamaño de las matrices a procesar fue relativamente pequeño.

Dentro de esta tesis pueden realizarse diversas mejoras para una mejor comparación entre los modelos de programación MapReduce y DLML, no obstante, a continuación se presenta el trabajo futuro.

6.2. Trabajo Futuro

Como trabajo futuro se propone implementar un nuevo diseño de DLML-IO que proporcione una interfaz fácil de usar, con el objetivo de que realice el manejo de archivos de forma transparente al usuario, de forma similar como lo hace MapReduce. Así mismo también se propone evaluar y adoptar un sistema de archivos distribuido que le suministre a DLML las mismas prestaciones que HDFS le proporciona a MapReduce.

Por otra parte, tanto la versión original de DLML, así como la extensión para el manejo de archivos DLML-IO, no incluyen tolerancia a fallos en caso de que algún procesador en el sistema presente algún fallo, de manera que la ejecución de una aplicación puede verse interrumpida si ésto sucede. Cualquier tipo de falla, ya sea por los procesadores del cluster, o por los discos en los que se monta el sistema de archivos, representa la pérdida de tiempo de cálculo y de información.

Finalmente, dentro del trabajo futuro también se propone llevar a cabo más pruebas con mayores volúmenes de datos a procesar (que superen los 100 GB o incluso los TB) y con una mayor infraestructura. Ésto con el objetivo de encontrar los diferentes problemas de escalabilidad con DLML-IO, ya que el impacto con una escala mayor de pruebas permitirá identificar posibles cuellos de botella, como podría ser el balance de carga con manejo de información global que es utilizado en esta extensión para DLML, o la segmentación lógica de los archivos de entrada que realiza DLML-IO.

Apéndice A
Pseudocódigos para MapReduce

En este apéndice se muestran los pseudocódigos de las funciones Map y Reduce para las 4 aplicaciones desarrolladas en el modelo de programación MapReduce.

Pseudocódigo de las funciones Map y Reduce para el Conteo de Palabras

```
\*****************************************************************************\
 La funcion map recibe un par de entrada que tiene 2 parametros,
     key: es un desplazamiento de archivo en bytes.
     values: es una cadena de caracteres.
 Esta funcion produce un par de salida por cada palabra en la cadena de caracteres.
\*****************************************************************************\

1.       map(long key, string values){
2.           para cada palabra p en values:
3.                EmiteParIntermedio(p, 1);
4.           fin-para
5.       }//fin-map
-------------------------------------------------------------------------------

\*****************************************************************************\
 La funcion reduce recibe un par de entrada que tiene 2 parametros,
     key: es una palabra.
     list_of_values: es una lista de valores que tiene cada ocurrencia
                  de la palabra (key) del par.
 Esta funcion produce un par de salida por cada par de entrada.
\*****************************************************************************\

1.       reduce(string key, list lista_de_valores){
2.           int suma_ocurrencias = 0;
3.           para cada ocurrencia oc en lista_de_valores
4.                suma_ocurrencias += oc;
5.           fin-para
6.           EmiteParFinal(key, suma_ocurrencias);
7.       }//fin-reduce
-------------------------------------------------------------------------------
```

Pseudocódigo de las funciones Map y Reduce para N-Reinas

En esta aplicación se requieren 2 tipos de funciones map, una función map1 que lea inicialmente los datos del tablero desde un archivo, y una segunda función map2 que reciba los pares de entrada generados en las distintas fases Map.

```
\*****************************************************************************\
 La funcion map recibe un par de entrada que tiene 2 parametros,
     key: es un desplazamiento de archivo en bytes.
     values: es una cadena de caracteres.
 Esta funcion produce pares de salida por cada par de entrada.
\*****************************************************************************\

1.       map1(long key, string values){
2.           i=0; j=0;
3.           primeraColumna=""; numeroReinas=""; llave""; tableroIesimo="";
4.           numeroRenglones=0; numeroColumnas=0; renglonesColocados=1;
5.
6.           primeraColumna = leePrimerDato(values);
7.           numeroReinas = leeSegundoDato(values);
8.           numeroRenglones = numeroColumnas = numeroReinas;
9.
10.          tablero = creaArregloDeTama\~no(numeroReinas);
11.          tablero[0]=convierteCadena_a_Entero(primeraColumna);
12.          renglonesColocados++;
13.
14.          si renglonesColocados-1 = numeroRenglones
15.              llave = creaCadena("#soluciones");
16.              EmiteParIntermedio(llave, 1);
17.          en otro caso
18.              para i=1 hasta que i<=numeroColumnas hacer
19.                  si renglonesColocados < numeroRenglones
20.                      si !noAtacan(renglonesColocados, i, tablero, numeroReinas)
21.                          tablero[renglonesColocados-1]=i;
22.                          para j=0 hasta que j<numeroRenglones hacer
23.                              tableroIesimo= convierteEntero_a_Cadena(tablero[j]);
24.                              llave=concatenaCadena(tableroIesimo+" ");
25.                              j++;
26.                          fin-para
27.                          EmiteParIntermedio(llave, numeroReinas);
28.                      fin-si !noAtacan
29.                  en otro caso
30.                       si !noAtacan(renglonesColocados, i, tablero, numeroReinas)
31.                           llave = creaCadena("#soluciones");
32.                           EmiteParIntermedio(llave, 1);
33.                       fin-si !noAtacan
34.                  fin-en otro caso
35.                  llave="";
36.                  i++;
37.              fin-para
```

```
38.          fin-en otro caso
39.     }//fin-map
-------------------------------------------------------------------------------


\*****************************************************************************\
 La funcion noAtacan recibe 4 parametros de entrada,
             renglonesColocados: es el numero de renglones con reina.
             columna: es la columna donde se va a colocar la posible siguiente reina.
             tablero: es el tablero de NxN que contiene algunas o todas las N-reinas.
             TAM: es el numero total de reinas
Esta funcion regresa true en caso de que la reina a colocar ataque a alguna reina ya
antes colocada, y regresa false en caso contrario.
\*****************************************************************************\

1.     noAtacan(int renglonesColocados, int columna, int[] tablero, int TAM){
2.           i=0; j=0;
3.           sinProblema=true;
4.           renglonesColocados=renglonesColocados-1;
5.           i=renglonesColocados-1;
6.
7.           //verifica columna
8.           mientras (i>=0) && (tablero[i]!=columna)
9.               i--;
10.          fin-mientras
11.
12.          si i < 0 //no es comida en columna
13.              i=renglonesColocados-1;
14.              j=columna-1;
15.
16.              //verifica diagonal izquierda
17.              mientras (i>=0) && (j>=0) && (tablero[i]!=j)
18.                  i--; j--;
19.              fin-mientras
20.
21.              si (i<0) || (j < 0)
22.                  i=renglonesColocados-1;
23.                  j=columna+1;
24.
25.                  //verifica diagonal derecha
26.                  mientras (i>=0) && (j<=TAM) && (tablero[i]!=j)
27.                      i++; j--;
28.                  fin-mientras
29.                  si (i<0) || (j > TAM)
30.                      sinProblema = false; //se puede colocar la reina
31.                  fin-si
32.              fin-si
33.          fin-si i<0
34.          regresa sinProblema;
35.     }fin-noAtacan
-------------------------------------------------------------------------------
```

```
\*****************************************************************************\
 Esta funcion recibe un par de entrada que tiene 2 parametros,
             key: es una cadena.
             values: es un entero.
 Esta funcion produce pares de salida por cada par de entrada.
\*****************************************************************************\

1.          map2(long key, int value){
2.
3.            i=0; j=0;
4.            primeraColumna=""; numeroReinas=""; llave""; tableroIesimo="";
5.            numeroRenglones=0; numeroColumnas=0; renglonesColocados=0;
6.
7.            numeroReinas = value;
8.            numeroRenglones = numeroColumnas = numeroReinas;

9.            primeraColumna = leePrimerDato(values);
10.           tablero = creaArregloDeTama\~no(numeroReinas);
11.
12.           para cada dato d en key
13.               tablero[renglonesColocados]=convierteCadena_a_Entero(d);
14.               si tablero[renglonesColocados]!=0;
15.                   renglonesColocados++;
16.               fin-si
17.           fin-para
18.
19.           renglonesColocados++;
20.           para i=1 hasta que i<=numeroColumnas hacer
21.                si renglonesColocados < numeroRenglones
22.                   si !noAtacan(renglonesColocados, i, tablero, numeroReinas)
23.                       tablero[renglonesColocados-1]=i;
22.                       para j=0 hasta que j<numeroRenglones hacer
23.                           tableroIesimo= convierteEntero_a_Cadena(tablero[j]);
24.                           llave=concatenaCadena(tableroIesimo+" ");
25.                           j++;
26.                       fin-para
27.                       EmiteParIntermedio(llave, numeroReinas);
28.                   fin-si !noAtacan
29.                en otro caso
30.                   si !noAtacan(renglonesColocados, i, tablero, numeroReinas)
31.                        llave = creaCadena("#soluciones");
32.                        EmiteParIntermedio(llave, 1);
33.                   fin-si !noAtacan
34.               llave="";
35.               i++;
36.           fin-para
37.
38.       }//fin-map2
-------------------------------------------------------------------------------
```

```
\*****************************************************************************\

 La funcion Reduce recibe un par de entrada que tiene 2 parametros,
             key: es una cadena (en este caso la etiqueta #soluciones).
             list_of_values: es una lista de valores que tiene un valor 1
                             por cada soluci\'on encontrada en el tablero.
 Esta funcion produce un par de salida.
\*****************************************************************************\

1.          reduce(string key, list lista_de_valores){
2.            soluciones = 0;
3.            para cada solucion s en lista_de_valores
4.                soluciones += s;
5.            fin-para
6.            EmiteParFinal(key, soluciones);
7.         }//fin-reduce

-----------------------------------------------------------------------------
```

Pseudocódigo de las funciones Map y Reduce para la Suma de Matrices

```
\*****************************************************************************\
 La funcion map recibe un par de entrada que tiene 2 parametros,
     key: es un desplazamiento de archivo en bytes.
     values: es una cadena de caracteres.
 Esta funcion produce un par de salida por cada dato en la cadena de caracteres.
\*****************************************************************************\

1.        map(long key, string values){
2.            i=0; x=0; y=0; elemento=0;
3.            posicion_x=""; posicion_y=""; llave="";
4.
5.            para cada dato d en values:
6.                si i>0
7.                    elemento = convierteCadena_a_Entero(d);
8.                    posicion_y = convierteEntero_a_Cadena(y);
9.                    llave = creaCadena(posicion_x+" "+posicion_y);
10.                   EmiteParIntermedio(llave, elemento);
11.                   y++;
12.               en otro caso
13.                   posicion_x = d;
14.               i++;
15.           fin-para
16.
17.       }//fin-map
-----------------------------------------------------------------------------
```

```
\*****************************************************************************\
 La funcion reduce recibe un par de entrada que tiene 2 parametros,
     key: es una cadena (en este caso la coordenada a calcular).
     list_of_values: es una lista de valores que tiene los datos para
                 calcular la llave (la coordenada) del par.
 Esta funcion produce un par de salida por cada par de entrada.
\*****************************************************************************\

1.       reduce(string key, list lista_de_valores){
2.           suma_ocurrencias = 0;
3.           para cada ocurrencia oc en lista_de_valores
4.                suma_ocurrencias += oc;
5.           fin-para
6.           EmiteParFinal(key, suma_ocurrencias);
7.       }//fin-reduce
-------------------------------------------------------------------------------
```

Pseudocódigo de las funciones Map y Reduce para la Multiplicación de Matrices

```
\*****************************************************************************\
 La funcion map1 recibe un par de entrada que tiene 2 parametros,
     key: es un desplazamiento de archivo en bytes.
     values: es una cadena de caracteres.
 Esta funcion produce tantos pares de salida por cada dato en la cadena de caracteres,
 como el tamano de una dimension de las matrices a multiplicar (recordar que son
 matrices cuadradas).
\*****************************************************************************\


1. map1(long key, string values){
2.
3.  i=0; lectura=0; columna=0; renglon=0; renglonIesimo=0;
4.  llave=""; componente1Key=""; componente2Key=""; componente3Key="";
5.  elementosTotales=0; componente1=0; contadorPares=0;
6.  inicioComponente1_LecturaH=0;
7.
8.  para cada dato d en values:
9.
10.   si i > 2
11.       si lectura = 1
12.           componente1 = inicioComponente1_LecturaH;
13.               mientras contadorPares != 0
14.                   componente1Llave = convierteEntero_a_Cadena(componente1);
15.                   componente2Llave = convierteEntero_a_Cadena(columna);
16.                   componente3Llave = convierteEntero_a_Cadena(elementosTotales);
17.                   llave=creaCadena = ( componente1Key+" "+componente2Key+" "+
18.                                        componente3Key);
```

```
19.                   EmiteParIntermedio(llave, d);
20.                   componente1++;
21.                   contadorPares--;
22.               fin-mientras
23.
24.               columna++;
25.       fin-si lectura=1
26.
27.       si lectura != 1
28.               componente1=renglonIesimo;
29.
30.               mientras contadorPares != 0
31.                   componente1Key = convierteEntero_a_Cadena(componente1);
32.                   componente2Key = convierteEntero_a_Cadena(renglon);
33.                   componente3Key = convierteEntero_a_Cadena(elementosTotales);
34.                   llave=creaCadena = ( componente1Key+" "+componente2Key+" "+
35.                                        componente3Key);
36.                   EmiteParIntermedio(llave, d);
37.                   componente1+=elementosTotales;
38.                   contadorPares--;
39.               fin-mientras
40.
41.               renglonIesimo+=1;
42.       fin-si lectura!=1
43.
44.       contadorPares = elementosTotales;
45.   fin-si i>2
46.
47.   si i = 0
48.       lectura = convierteCadena_a_Entero(d);
49.   fin-si i=0
50.
51.   si i = 1
52.       renglon = convierteCadena_a_Entero(d);
53.   fin-si i=1
54.
55.   si i = 2
56.       elementosTotales = convierteCadena_a_Entero(d);
57.       inicioComponente1_LecturaH = renglon * elementosTotales;
58.       contadorPares = elementosTotales;
59.   fin-si i=2
60.
61.  i++;
62.
63.  fin-para
64.
65. }//fin-map1
```

```
\*****************************************************************************\
 La funcion reduce1 recibe un par de entrada que tiene 2 parametros,
     key: es una cadena (en este caso tiene 3 datos).
     list_of_values: es una lista de valores que tiene los datos para
                 a multiplicar para despues calcular una coordenada.
 Esta funcion produce un par de salida por cada par de entrada, donde el par de salida
 tiene una llave que representa la coordenada a calcular, y el valor es una de los
 elementos necesarios en el calculo de la suma final para obtenerla.
\*****************************************************************************\


1.       reduce1(string key, list lista_de_valores){
2.           posicion_x=""; posicion_y=""; llave="";
3.           i=0; multiplicacion=1;
4.
5.           para cada dato d en key
6.               si i = 0
7.                   posicion_x = d;
8.               si i = 2
9.                   posicion_y = d;
10.          fin-para
11.
12.          para cada valor v en lista_de_valores
13.              multiplicacion = multiplicacion * v;
14.          fin-para
15.
16.          llave=creaCadena( posicion_x+" "+posicion_y);
17.          EmiteParFinal(llave, multiplicacion);
18.
19.      }//fin-reduce1
-------------------------------------------------------------------------------


\*****************************************************************************\
 La funcion map2 recibe un par de entrada que tiene 2 parametros,
     key: es un desplazamiento de archivo en bytes.
     values: es una cadena de caracteres que contiene la coordenada
             a calcular, y uno de los datos requeridos en la suma final
             para calcular la coordenada en la multiplicacion de matrices.
 Esta funcion produce un par de salida por cada par de entrada.
\*****************************************************************************\


1.       map2(long key, string values){
2.
3.           i=0; x=0; y=0; a=0; b=0; elemento=0;
4.           posicion_x=""; posicion_y=""; dato""; llave="";
5.
6.           posicion_x = leePrimerDato(values);
7.           posicion_y = leeSegundoDato(values);
8.           dato = leeTercerDato(values);
```

```
9.
10.          x=convierteCadena_a_Entero(posicion_x);
11.          y=convierteCadena_a_Entero(posicion_y);
12.          elemento=convierteCadena_a_Entero(dato);
13.
14.          a=x; b=y;
15.          x=a/b;
16.          y=a/(x*b);
17.
18.          x=funcionPiso(x);
19.          y=funcionPiso(y);
20.
21.          posicion_x = convierteEntero_a_Cadena(x);
22.          posicion_y = convierteEntero_a_Cadena(y);
23.
24.          llave = creaCadena(posicion_x+" "+posicion_y);
25.          EmiteParIntermedio(llave, elemento);
26.
27.      }//fin-map2
-------------------------------------------------------------------------------
```

```
\*****************************************************************************\
 La funcion reduce2 recibe un par de entrada que tiene 2 parametros,
     key: es una cadena (en este caso la coordenada a calcular.
     list_of_values: es una lista de valores que tiene los datos para
                 calcular la llave (la coordenada) del par.
 Esta funcion produce un par de salida por cada par de entrada.
\*****************************************************************************\
```

```
1.       reduce2(string key, list lista_de_valores){
2.           valor_coordenada = 0;
3.
4.           para cada dato d en lista_de_valores
5.               valor_coordenada += d;
6.           fin-para
7.           EmiteParFinal(key, valor_coordenada);
8.
9.       }//fin-reduce2
-------------------------------------------------------------------------------
```

Apéndice B
Pseudocódigos para DLML-IO

En este apéndice se muestran los pseudocódigos de las funciones de DLML-IO, y los pseudocódigos para las 4 aplicaciones desarrolladas en el modelo de programación DLML.

Pseudocódigo de la función DLML_fileList de DLML-IO

```
\*****************************************************************************\
 La funcion DLML_fileList recibe 3 parametros,
    directorio: es la ruta de un directorio con archivos a procesar
    A: es una lista de archivos.
    memoria_actual: es la cantidad de memoria actual en bytes.
 Esta funcion produce una lista de los archivos en un directorio y regresa un 0 en caso
 de que todo salga bien, y 1 en caso contrario.
\*****************************************************************************\

1. int DLML_fileList(char* directorio, list* A, long* memoria_actual){
2.
3.       archivo=""; script;"./enlista_archivos.sh";
4.       rutaArchivo="";
5.       nodoArchivo* nodo;
6.       anterior="";
7.
8.       salida=0;
9.       FILE* fichero;
10.
11.      rutaArchivo=script;
12.      copiaCadena(rutaArchivo,directorio);
13.
14.      //se encarga de hacer un ls -1 al directorio y producir
15.      //el archivo lista_de_archivos.txt
16.      llamadaSistema(rutaArchivo);
17.
18.      fichero=abreArchivo("lista_de_archivos.txt");
19.      si fichero esta abierto
20.          imprime "lista de archivos abierta";
21.      sino
22.          imprime "Error con el fichero";
```

```
23.          regresa 1;
24.      sin-si
25.
26.      haz{
27.          archivo=leeNombreArchivo(fichero);
28.          nodo=crea_nodoArchivo();
29.          insertaNodo(A,nodo);
30.          memoria_actual-=sizeof(nodo);
31.
32.          //lee el salto de linea
33.          leeSiguienteCaracter(fichero);
34.
35.          //se guarda el comienzo de cada palabra
36.          anterior=guardaPosicion(fichero);
37.
38.          //verifica si se encontro el fin de linea
39.          //de lo contrario se regresa a la posicion anterior
40.          si leeSiguienteCaracter(fichero)=finDeArchivo
41.              salida=1;
42.          en otro caso
43.              regresaPoscionArchivo(fichero,anterior);
44.
45.      }mientras(salida!=1);
46.
47.         fichero=cierraArchivo("lista_de_archivos.txt");
48.         si fichero esta cerrado
49.             imprime "lista de archivos cerrada";
50.             regresa 0;
51.         sino
52.             imprime "Error con el fichero";
53.             regresa 1;
54.         sin-si
55.
56.     }//fin-DLML_fileList
```

Pseudocódigo de las función DLML_distributeOffsets de DLML-IO

```
\*****************************************************************************\
 La funcion DLML_distributeOffsets: recibe 4 parametros,
    archivo1: es la cadena de datos a escribir.
    archivo2: es el apuntador al archivo donde va a almacenarse el dato.
    memoria_actual: es la cantidad de memoria actual en bytes.
    lista_segmentos: es la lista en la que se almacena cada nodo que contiene
                     informacion de los segmentos a procesar.

 Esta funcion produce una lista, cuyos nodos almacenan informacion de segmentos de los
 archivos de entrada a procesar. Regresa un 1 en caso de que todo salga bien, y 0
 en caso contrario.
\*****************************************************************************\
```

```
1. int DLML_distributeOffsets(char *archivo1,
                              char* archivo2, long *memoria_actual, lista* D){
2.   i=0; j=0; sin_segmento=0; procesosAplicacion=0;
3.   lineas=0; salida=0;
4.   FILE *datos_a, *datos_b;
5.   k=0; retorno=0; numeroRenglones=0; renglonesA0=0;
6.   caracterA=""; caracterB="";
7.   nodoSegments *nodo;
8.
9.   //determina el numero de procesos aplicacion en el sistema
10.  //DLBE_Procesadores() ya esta incluida en la plataforma
11.  procesosAplicacion = DLBE_Procesadores();
12.
13.  //abre los archivos de entrada
14.  datos_a=abreArchivo(archivo1);
15.  si archivo2 != NULL;
16.      datos_b=abreArchivo(archivo2);
17.      si !(datos_a && datos_b)
18.          imprime("ERROR con los ficheros");
19.          regresa 0;
20.      fin-si
21.  en otro caso si !(datos_a)
22.      imprime("ERROR con el archivo1");
23.      regresa 0;
24.  fin-en otro caso
25.
26.  //determina el numero de lineas en uno de los archivos
27.  haz
28.      //en cada lectura se posiciona al siguiente caracter
29.      caracterA=leeCaracter(datos_a);
30.      si caracterA='\n';
31.          lineas++;
32.      en otro caso si caracterA=EOF
33.          lineas++;
34.          salida=1;
35.      fin-en otro caso
36.  mientras salida!=1;
37.
38.  //se divide el numero de lineas entre los procesos aplicacion
39.  numeroRenglones=lineas/procesosAplicacion;
40.  renglonesA0=numeroRenglones;
41.
42.  //si hay mas aplicaciones que renglones
43.  //se proporciona un renglon a cada aplicacion mientras se pueda
44.  //a los restantes se les envia un mensaje de no-carga -> -1
45.
46.  si numeroRenglones=0;
47.      numeroRenglones=1;
48.      renglonesA0=numeroRenglones;
49.      sin_segmento=procesosAplicacion-lineas;
```

```
50.      para i=0 hasta i<sin_trabajo hacer
51.          nodo=creaNodoConInformacionNegativa;
52.          insertaNodo(D, nodo);
53.          (*memoria_actual)=(*memoria_actual)-sizeof(nodo);
54.          i++;
55.      fin-para
56.
57.  en otro caso si (numeroRenglones * procesosAplicacion)!= lineas
58.      renglonesA0 = numeroRenglones + lineas-(numeroRenglones * procesosAplicacion)
59.  fin-en otro caso
60.
61.  //se analiza todo contenido de los dos archivos
62.  //se guardan los desplazamientos del inicio de los segmentos correspondientes
63.
64.  k=0;
65.  regresaPoscionArchivo(datos_a,0);
66.  para i=0 hasta i<lineas hacer
67.      si i<renglonesA0
68.          haz
69.              //por cada caracter leido se posiciona al siguiente caracter
70.              //de forma automatica
71.              caracterA=leeCaracter(datos_a);
72.          mientras caracterA!='\n' && caracterA!=EOF
73.
74.          si archivo2 != NULL;
75.              haz
76.                  caracterB=leeCaracter(datos_b);
77.              mientras caracterB!='\n' && caracterB!=EOF
78.          fin-si
79.
80.          k++;
81.          si k%renglonesA0=0
82.              k=0;
83.          fin-si
84.
85.      en otro caso
86.          si k=0
87.              //se guardan los desplazamientos para las aplicaciones
88.              //excepto para la aplicacion 0
89.              si archivo2 != NULL;
90.                  nodo=creaNodo(archivo1,desplazamiento(datos_a), archivo2,
91.                                desplazamiento(datos_b), numeroRenglones);
92.              en otro caso
93.                  nodo=creaNodo(archivo1, desplazamiento(datos_a), numeroRenglones);
94.              fin-en otro caso
95.
96.               insertaNodo(D,nodo);
97.               (*memoria_actual)=(*memoria_actual)-sizeof(nodo);
98.          fin-si k=0
99.
100.         haz
```

```
101.             //por cada caracter leido se posiciona al siguiente caracter
102.             //de forma automatica
103.             caracterA=leeCaracter(datos_a);
104.         mientras caracterA!='\n' && caracterA!=EOF
105.
106.         si archivo2 != NULL;
107.             haz
108.                 caracterB=leeCaracter(datos_b);
109.             mientras caracterB!='\n' && caracterB!=EOF
110.         fin-si
111.
112.         k++;
113.         si k%renglonesA0=0
114.             k=0;
115.         fin-si
116.
117.     fin-en otro caso
118.
119.     i++;
120. fin-para
121.
122. //se crea el nodo de desplazamiento para la aplicacion 0
123. si archivo2 != NULL;
124.     nodo=creaNodo(archivo1, 0, archivo2, 0, numeroRenglones);
125. en otro caso
126.     nodo=creaNodo(archivo1, 0, numeroRenglones);
127. fin-en otro caso
128.
129. insertaNodo(D,nodo);
130. (*memoria_actual)=(*memoria_actual)-sizeof(nodo);
131.
132. cierraArchivo(archivo1);
133. si archivo2 != NULL;
134.     cierraArchivo(archivo2);
135. fin-si
136.
137. regresa 1;
138.
139.}//fin DLML_distributeOffsets
```

Pseudocódigo de la función DLML_Load de DLML-IO

```
\*****************************************************************************\
 La funcion DLML_Load recibe 4 parametros,
    datosSeg: es una estructura que contiene los elementos: long desp_a, long *desp_b,
              long reng, long col, FILE *datos_a, FILE *datos_b, listaItems *L, int carga_inf,
              long lineas, long aux_i, long aux_j, long aux_k, long posx, int *nodo_desp.
    memoria_actual: es la cantidad de memoria actual en bytes.
    umbral: es el umbral para la memoria actual en bytes.
    modo_operacion: indica uno de los 3 modos de funcionamiento de esta funcion.
```

```
 Esta funcion se encarga de llamar a la funcion asociada el modo de operacion, para la
 posterior inicializacion de los items asociados a los datos de los
 segmentos a procesar. Esta funcion regresa 1 si se inicializo algun item, y regresa 0
 si el segmento de entrada no contiene renglones, o si se ha terminado la lectura y
 posterior creacion de items asociados con los segmentos.
\*****************************************************************************\


1.  int DLML_Load(infoSegmento *datosSeg, long *memoria_actual, long umbral,
              char modo_operacion){
2.
3.       valor=0;
4.
5.       //se carga el segmento correspondiente del archivo
6.       //si la informacion del segmento no es negativa
7.       si (datosSeg->desp_a > -1) && (datosSeg->carga_inf = 0)
8.
9.           lee(modo_operacion)
10.              caso "+"
11.                   valor=cargaItemsSumas(datosSeg, memoria_actual, umbral);
12.                   regresa valor
13.              fin-caso "+"
14.              caso "*"
15.                  valor=cargaItemsMultiplicaciones(datosSeg, memoria_actual, umbral);
16.                  regresa valor;
17.              fin-caso "*"
18.              caso "w"
19.                  valor=cargaItemsPalabras(datosSeg, memoria_actual, umbral);
20.                  regresa valor;
21.              fin-caso "w"
22.          fin-lee
23.
24.      fin-si_datos_cargados
25.      en otro caso si (datosSeg->desp_a = -1) && (datosSeg->carga_inf = 0);
26.          (datosSeg->carga_inf)=1;
27.          (*datosSeg->nodo_desp)=0;
28.          regresa 0;
29.      fin-en otro caso
30.
31.      //se verifica si ya se cargo todo el segmento del archivo
32.      si (datosSeg->carga_inf)=1
33.          (*datosSeg->nodo_desp)=0;
34.          regresa 0;
35.      fin-si
36.
37. }//fin-DLML_Carga
-------------------------------------------------------------------------------------
```

```
\*****************************************************************************\
 La funcion cargaItemsPalabras recibe 3 parametros,
    datosSeg: es una estructura que contiene los elementos: long desp_a, long *desp_b,
              long reng, long col, FILE *datos_a, FILE *datos_b, listaItems *L, int carga_inf,
              long lineas, long aux_i, long aux_j, long aux_k, long posx, int *nodo_desp.
    memoria_actual: es la cantidad de memoria actual en bytes.
    umbral: es el umbral para la memoria actual en bytes.
 Esta funcion se encarga de la inicializacion de los items asociados a los datos de los
 segmentos a procesar (para el caso del Conteo de Palabras). Esta funcion regresa 1 si
 se inicializa algun item, y regresa 0 si el segmento no contiene renglones, o  si se
 ha terminado la lectura y posterior creacion de items asociados con los segmentos.
\*****************************************************************************\

1.  int cargaItemsPalabras(infoSegmento *datosSeg, long *memoria_actual, long umbral){
2.
3.       i=0; actualiza=0; cmp=0; rompe=0; chk=0; datos_cargados=0; finLinea=0;
4.       retorno=0;
5.       nodo *item;
6.       caracter=""; car2=""; palabra[2000];
7.
8.       //se posiciona el puntero al segmento a procesar
9.       //al inicio del segmento o al inicio de los renglones restantes
10.      establecePoscionEnArchivo( (datosSeg->datos_a), (datosSeg->desp_a) );
11.
12.      para i=0 hasta que i<(datosSeg->reng) hacer
13.          haz
14.
15.              //se verifica si hay que actualizar los indices
16.              si (datosSeg->aux_i)!=0 && actualiza=0
17.                  i=(datosSeg->aux_i);
18.                  actualiza=1;
19.              fin-si
20.
21.              //Cuando se viola el umbral se detiene la creacion de nodos
22.              cmp=(*memoria_actual)-sizeof(nodo);
23.              si cmp <= umbral
24.                  (datosSeg->aux_i) = i;
25.                  (datosSeg->desp_a) = desplazamiento(datosSeg->datos_a);
26.                  rompe=1;
27.                  break;//rompe el ciclo inmediato
28.                  en otro caso
29.                  rompe=0;
30.              fin-en otro caso
31.
32.              //Verifica si la palabra a leer es un salto de linea
33.              retorno = desplazamiento(datosSeg->datos_a);
34.              car2=leeCaracter(datosSeg->datos_a);
35.              si car2!='\n' && car2!='\r' && car2!=' '){
36.                  establecePoscionEnArchivo( (datosSeg->datos_a), retorno );
37.                  chk=leeyGuardaPalabra( (datosSeg->datos_a), "%s", palabra );
38.
```

```
39.                 si chk!=-1
40.                     item=creaNodo(palabra,-2,1);
41.                     DLML_insert( (datosSeg->L), item);
42.                     (*memoria_actual)= (*memoria_actual)-sizeof(nodo);
43.                     datos_cargados=1;
44.                 fin-si
45.
46.                 caracter=leeCaracter(datosSeg->datos_a);
47.                 si chk=-1 || caracter='\n'
48.                     finLinea=1;
49.                     (datosSeg->lineas)++;
50.                 fin-si
51.
52.             en otro caso si car2='\n' || car=EOF
53.                 finLinea=1;
54.                 (datosSeg->lineas)++;
55.             fin-en otro caso
56.
57.         mientras finLinea!=1;
58.         finLinea=0;
59.
60.         si rompe=1
61.             break;
62.         fin-si
63.     i++;
64.     fin-para
65.
66.     si i=(datosSeg->num_reng) && i=(datosSeg->lineas)
67.         (datosSeg->carga_inf)++;
68.         (datosSeg->aux_i)=i;
69.         (*datosSeg->nodo_desp)=0;
70.         cierraArchivo(datosSeg->datos_a);
71.     fin-si
72.
73.     regresa datos_cargados;
74.
75.  }//fin-cargaItemsPalabras
```

```
\*****************************************************************************\
 La funcion cargaItemsSumas recibe 3 parametros,
    datosSeg: es una estructura que contiene los elementos: long desp_a, long *desp_b,
              long reng, long col, FILE *datos_a, FILE *datos_b, listaItems *L, int carga_inf,
              long lineas, long aux_i, long aux_j, long aux_k, long posx, int *nodo_desp.
    memoria_actual: es la cantidad de memoria actual en bytes.
    umbral: es el umbral para la memoria actual en bytes.
 Esta funcion se encarga de la inicializacion de los items asociados a los datos de los
 segmentos a procesar (para el caso de la Suma de Matrices). Esta funcion regresa 1 si
 se inicializa algun item, y regresa 0 si el segmento no contiene renglones, o  si se
 ha terminado la lectura y posterior creacion de items asociados con los segmentos.
\*****************************************************************************\
```

```
1.   int cargaItemsSumas(infoSegmento *datosSeg, long *memoria_actual, long umbral){
2.
3.       i=0; j=0; actualiza=0; cmp=0; rompe=0; datos_cargados=0;
4.       a=0; b=0; pos_y=0;
5.       nodo *item;
6.
7.       //se posiciona el puntero al segmento a procesar
8.       //al inicio del segmento o al inicio de los renglones restantes
9.       establecePoscionEnArchivo( (datosSeg->datos_a), (datosSeg->desp_a) );
10.      establecePoscionEnArchivo( (datosSeg->datos_b), (*datosSeg->desp_b));
11.
12.      para i=0 hasta que i<(datosSeg->reng) hacer
13.          para j=0 hasta que j<(datosSeg->col) hacer
14.          haz
15.
16.              //se verifica si hay que actualizar los indices
17.              si ( (datosSeg->aux_i)!=0 || (datosSeg->aux_i)!=0 ) && actualiza=0
18.                  i=(datosSeg->aux_i);
19.                  j=(datosSeg->aux_j);
20.                  actualiza=1;
21.              fin-si
22.
23.              //Cuando se viola el umbral se detiene la creacion de nodos
24.              cmp=(*memoria_actual)-sizeof(nodo);
25.              si cmp <= umbral
26.                  (datosSeg->aux_i) = i;
27.                  (datosSeg->aux_j) = j;
28.                  (datosSeg->desp_a) = desplazamiento(datosSeg->datos_a);
29.                  (*datosSeg->desp_b) = desplazamiento(datosSeg->datos_b);
30.                  rompe=1;
31.                  break;//rompe el ciclo inmediato
32.                  en otro caso
33.                      rompe=0;
34.                  fin-en otro caso
35.
36.              leeyGuardaDato( (datosSeg->datos_a), "%d", &a);
37.              leeyGuardaDato( (datosSeg->datos_b), "%d", &b);
38.
39.              //se establece el numero del renglon a procesar
40.              si j=0
41.                  (datosSeg->pos_x)=a;
42.              fin-si
43.
44.              //se crean los nodos de cada uno de los componentes del renglon
45.              //con las posiciones correspondientes
46.              si j>0
47.                  pos_y=j-2;
48.                  pos_y++;
49.                  item=creaNodo( (datosSeg->pos_x), pos_y, a, b);
```

```
50.                 DLML_insert( (datosSeg->L),item);
51.                 (*memoria_actual)= (*memoria_actual)-sizeof(nodo);
52.                 datos_cargados=1;
53.             fin si
54.         j++;
55.         fin-para j
56.
57.         //rompe ciclo para-i
58.         si rompe=1
59.             break;
60.         fin-si
61.
62.         //regresa a contar desde la primera columna
63.         si j=( (datosSeg->col)-1 )
64.             j=0;
65.         fin-si
66.
67.     i++;
68.     fin-para i
69.
70.     si i=(datosSeg->reng) && j=(datosSeg->col)
71.         (datosSeg->carga_inf)++;
72.         (datosSeg->aux_i)=i;
73.         (datosSeg->aux_j)=j;
74.         (*datosSeg->nodo_desp)=0;
75.         cierraArchivo(datosSeg->datos_a);
76.         cierraArchivo(datosSeg->datos_b);
77.     fin-si
78.
79.     regresa datos_cargados;
80.
81.  }//fin-cargaItemsSumas
```

```
\*****************************************************************************\
 La funcion cargaItemsMultiplicaciones recibe 3 parametros,
    datosSeg: es una estructura que contiene los elementos: long desp_a, long *desp_b,
              long reng, long col, FILE *datos_a, FILE *datos_b, listaItems *L, int carga_inf,
              long lineas, long aux_i, long aux_j, long aux_k, long posx, int *nodo_desp.
    memoria_actual: es la cantidad de memoria actual en bytes.
    umbral: es el umbral para la memoria actual en bytes.
 Esta funcion se encarga de la inicializacion de los items asociados a los datos de los
 segmentos a procesar (para el caso de la Multiplicacion de Matrices). Esta funcion
 regresa 1 si se inicializa algun item, y regresa 0 si el segmento no contiene renglones,
 o  si se ha terminado la lectura y posterior creacion de items asociados con los
 segmentos.
\*****************************************************************************\
```

```
1.  int cargaItemsMultiplicaciones(infoSegmento *datosSeg,
                                   long *memoria_actual, long umbral){
2.
3.       i=0; j=0; k=0; actualiza=0; cmp=0; rompe=0; datos_cargados=0;
4.       a=0; b=0; pos_y=0; despi_a; despi_b; l=-1;
5.       nodo *item;
6.       Mat renglon, columna; //Mat es fijado por el numero de columnas en la matriz
7.
8.       //se asignan los desplazamientos de inicio
9.       despi_a=(datosSeg->desp_a);
10.      despi_b=(datosSeg->desp_b[0]);
11.
12.      para i=0 hasta que i<(datosSeg->reng) hacer
13.          para j=0 hasta que j<(datosSeg->col) hacer
14.
15.              //comienza la creacion de items
16.              si j>2
17.                  establecePoscionEnArchivo( (datosSeg->datos_a), despi_a );
18.                  para k=0 hasta que k<(datosSeg->col) hacer
19.
20.                      //se verifica si hay que actualizar los indices
21.                      si ( (datosSeg->aux_i)!=0 || (datosSeg->aux_i)!=0 ||
                             (datosSeg->aux_k)!=0 ) && actualiza=0
22.                          i=(datosSeg->aux_i);
23.                          j=(datosSeg->aux_j);
24.                          k=(datosSeg->aux_k);
25.                          actualiza=1;
26.                      fin-si
27.
28.                      //Cuando se viola el umbral se detiene la creacion de nodos
29.                      cmp=(*memoria_actual)-sizeof(nodo);
30.                      si cmp <= umbral
31.                          (datosSeg->aux_i) = i;
32.                          (datosSeg->aux_j) = j;
33.                          (datosSeg->desp_a) = despi_a);
34.                           rompe=1;
35.                           break;//rompe el ciclo inmediato
36.                      en otro caso
37.                          rompe=0;
38.                      fin-en otro caso
39.
40.                      leeyGuardaDato( (datosSeg->datos_a), "%d", &a);
41.
42.                      //se establece el numero del renglon a procesar
43.                      si k=1
44.                          (datosSeg->pos_x)=a-1;
45.                      fin-si
46.
47.                      si k>2
48.                          //se posciona al primer elemento de la matriz_b
49.                          establecePoscionEnArchivo( (datosSeg->datos_b),
```

```
                                                (datosSeg->desp_b[0]) );
50.
51.                         //ciclo para obtener el elemento de la matriz b
52.                         haz
53.                             leeyGuardaDato( (datosSeg->datos_b), "%d", &b);
54.                             l++;
55.                         mientras l<j-3
56.                         l=-1;
57.                         renglon[k-3]=a;
58.                         columna[k-3]=b;
59.                     fin-si
60.
61.                     si k+1=(datosSeg->col)
62.
63.                         //se crean e insertan los nodos de los componentes
64.                         //del renglon, con las posiciones correspondientes
65.                         pos_y=j-4;
66.                         pos_y++;
67.                         item=creaNodo( (datosSeg->pos_x), pos_y, renglon, columna);
68.                         DLML_insert( (datosSeg->L),item);
69.                         (*memoria_actual)= (*memoria_actual)-sizeof(nodo);
70.                         datos_cargados=1;
71.                     fin si
72.
73.                     k++;
74.                 fin-para k
75.
76.                 //rompe ciclo para-j
77.                 si rompe=1
78.                     break;
79.                 fin-si
80.
81.                 //se guarda el inicio del sisguiente renglon
82.                 si j+1=(datosSeg->col);
83.                     despi_a=desplazamiento(datosSeg->datos_a);
84.                 fin-si
85.
86.             fin-si-j>2
87.
88.             j++;
89.         fin-para-j
90.
91.         //rompe ciclo para-i
92.         si rompe=1
93.             break;
94.         fin-si
95.
96.         //actualiza el desplazamiento inicial del reglon
97.         despi_a=desplazamiento(datosSeg->datos_a);
98.
99.         i++;
```

```
100.     fin-para i
101.
102.     si i=(datosSeg->reng) && j=(datosSeg->col) && k=(datosSeg->col)
103.         (datosSeg->carga_inf)++;
104.         (datosSeg->aux_i)=i;
105.         (datosSeg->aux_j)=j;
106.         (datosSeg->aux_k)=k;
107.         (*datosSeg->nodo_desp)=0;
108.         cierraArchivo(datosSeg->datos_a);
109.         cierraArchivo(datosSeg->datos_b);
110.     fin-si
111.
112.     regresa datos_cargados;
113.
114. }//fin-cargaItemsMultiplicaciones
```

Pseudocódigo de la función DLML_Write de DLML-IO

```
\*****************************************************************************\
 La funcion DLML_Write recibe 3 parametros,
    *archivo: es el apuntador al archivo donde va a almacenarse el dato.
    *coordenada: es la coordenada del resultado.
    *cadenaDatos: es la cadena de datos a escribir.
 Esta funcion escribe la coordenada (en caso de haberla) y la cadena de datos en un
 archivo.
\*****************************************************************************\


1.  int DLML_Write(File* archivo, char *coordenada, char* cadenaDatos,){
2.    char *resultado;
3.
4.    si coordenada!=NULL;
5.        resultado=concatena(coordenada+" "+cadenaDatos);
6.        escribe(archivo,resultado);
7.    en otro caso
8.   escribe(archivo, cadenaDatos);
9.  }//fin-DLML_Write
```

Pseudocódigo de la aplicación del Conteo de Palabras

```
1.  #include <stdio.h>
2.  #include "dlml.h"
3.
4.  //macro para determinar la memoria libre
5.  #define MEMORIA_LIBRE (sysconf(_SC_PAGESIZE) * sysconf(_SC_AVPHYS_PAGES))
6.
7.  //macro para establecer el umbral de memoria
```

```
8.  #define UMBRAL 0.30
9.
10. //macro para obtener el numero de procesadores en el nodo
11. #define PROCESADORES_POR_NODO sysconf(_SC_NPROCESSORS_CONF)
12.
13.
14. main(char *directorio){
15.
16.  listaItems L;
17.  listaSegmentos D;
18.  listaArchivos A;
19.
20.  tiempo_inicial=0;
21.  tiempo_final=0;
22.  memoria_actual=0;
23.  umbral=0;
24.  procAplicacionTot=0;
25.  procAplicacionLoc=0;
26.
27.  archivo[2000];
28.  nodoArchivo archivoI;
29.
30.  //DLBE_Procesadores() ya esta incluida en la biblioteca
31.  procAplicacionTot=DLBE_Procesadores();
32.  procAplicacionLoc=PROCESADORES_POR_NODO;
33.
34.  //calcula el umbral al 30% de la memoria actual
35.  memoria_actual=MEMORIA_LIBRE/procAplicacionLocales+1;
36.  umbral=memoria_actual*UMBRAL;
37.
38.  si idProcAplicacion=0
39.      tiempo_inicial=tiempo();
40.      DLML_fileList(directorio, &A, &memoria_actual);
41.
42.      si A->longitud>0
43.          mientras A->longitud>0 haz
44.              archivoI=obtenArchivo(&A);
45.              copia(archivo,archivoI.nombre);
46.              memoria_actual+=sizeof(nodoArchivo);
47.              DLML_distributeOffsets(archivo, NULL, &memoria_actual, &D);
48.          fin-mientras
49.
50.          //DISRIBUCION DE SEGMENTOS
51.          envia_lista_segmentos(&D,memoria_actual);
52.
53.      fin-si
54.      Procesa(&L,&D,procAplicacionTot,idProcAplicacion,&memoria_actual,umbral);
55.
56.      tiempo_final=tiempo();
57.      imprime "Tiempo de ejecucion: [%f] segundos", tiempo_final-tiempo_inicial;
58.
```

```
59.  en otro caso
60.      Procesa(&L,&D,procAplicacionTot,idProcAplicacion,&memoria_actual,umbral);
61.
62.  fin-en otro caso
63.
64.}
-------------------------------------------------------------------------------


\*****************************************************************************\
 La funcion Procesa recibe 6 parametros,
    *L: la lista de items.
    *D: la lista de items que contienen la informacion de los Segmentos a procesar.
    nproc: el numero de procesos aplicacion en el sistema.
    idproc: el identificador del proceso aplicacion que invoco esta funcion.
    *memoria_actual: el contador de memoria actual (en bytes).
    umbral: es el umbral para la memoria actual en bytes.
 Esta funcion se encarga del procesamiento de items en la aplicacion.
\*****************************************************************************\


1.  Procesa(listaItems *L,listaSegmentos *D,int nproc,int idproc,long *memoria_actual,
                                                                    long umbral){
2.
3.   i=0; j=0; salida=1; datosProcesados=0; total=0; nodo_desp=0; checa_nodo=1;
4.   resultadosParciales=0;
5.   FILE *archivo[nproc];
6.   char fileNombre[2000]
7.   nodoSegments segmento;
8.   infoSegmento datosSeg;
9.   nodo item;
10.  lineas=0;
11.  espacio="";
12.
13.  //descontar las variables y estructuras declaradas
14.  (*memoria_actual)=(*memoria_actual)-sizeof(datosSeg)-sizeof(7*int)-
                      sizeof(fileNombre)-sizeof(espacio);
15.
16.  //se abren archivos locales para cada proceso aplicacion
17.  para i=0 hasta i<nproc
18.      copiaCadena(fileNombre, "/tmp/palabras/%d.%d.txt",idproc,i);
19.      archivo[i]=abreArchivo(fileNombre);
20.      i++;
21.  fin-para
22.
23.  recibe_lista_segmentos(D,memoria_actual);
24.
25.  mientras salida!=0
26.
27.      //obtener item con informacion de un segmento
28.      //en caso se que no se este cargando ninguno
```

```
29.      si D->longitud>0 && nodo_desp=0
30.          segmento=obtieneSegmento(D);
31.          (*memoria_actual)+=sizeof(nodoSegments);
32.
33.          datosSeg.reng=segmento.reng;
34.          si segmento.nombre!="";
35.              datosSeg.datos_a=abreArchivo(segmento.nombre1);
36.          fin-si
37.  nodo_desp=1;
38.          datosSeg.desp_a=segmento.desp_a;
39.          datosSeg.lineas=0;
40.          datosSeg.aux_i=0;
41.          datosSeg.carga_inf=0;
42.          datosSeg.checa_nodo=0;
43.          datosSeg.nodo_desp=&nodo_desp;
44.          datosSeg.L=L;
45.
46.      fin-si
47.
48.
49.      mientras DLML_Load(&datosSeg, memoria_actual, umbral, "w" )
50.
51.          mientras( obtieneItem(L,D,&item,&salida,datosSeg.nodo_desp) )
52.
53.              (*memoria_actual)+=sizeof(nodo);
54.              datosProcesados++;
55.
56.              //se aplica una funcion hash para almacenar la palabra
57.              //en uno de los archivos
58.              DLML_Write( archivo[hash(item.nombre)], "", item.nombre);
59.
60.          fin-mientras-obtieneItem
61.
62.      fin-mientras-DLML_Load
63.
64.  fin-mientras-salida
65.
66.  imprime("idProcesoAplicacion= %d procesados=%d\n",idproc,procesados);
67.
68.  //se cierran los archivos locales del proceso aplicacion
69.  para i=0 hasta i<nproc
70.      cierraArchivo(archivo[i]);
71.      i++;
72.  fin-para
73.
74.  //se copian los archivos a los respectivos procesos aplicacion
75.  para i=0 hasta i<nproc
76.      copiaCadena(fileNombre, "mv /tmp/palabras/%d.%d.txt /rutaSistemaArchivos",
                    idproc,i);
77.      llamadaSistema(fileNombre);
78.      i++;
```

```
79.  fin-para
80.
81.  //Una vez que los procesos aplicacion envian sus archivos al sistema de archivos
82.  //se comunican entre si para notificar esta accion
83.  sincronizacion();
84.
85.  para i=0 hasta i<nproc
86.      copiaCadena(fileNombre,"mv /rutaSistemaArchivos/%d.%d.txt /tmp/palabras/",
                    i,idproc);
87.      llamadaSistema(fileNombre);
88.      i++;
89.  fin-para
90.
91.  //Una vez que los procesos aplicacion transfieren sus respectivos archivos del
92.  //sistema de archivos a su disco local, se comunican para notificar esta accion
93.  sincronizacion();
94.
95.  imprime "idProcesoAplicacion= %d, Transferencia completada\n",idproc);
96.
97.  //se realiza el conteo de ocurrencias de cada palabra en los archivos del proceso
98.  //aplicacion. El script hace un sort a los archivos de palabras del proceso aplicacion,
99.  //un uniq -c e invierte el orden.
100. //sort 0.idproc.txt 1.idproc.txt ... 15.idproc.txt |uniq -c| awk '{print $2, "\t"$1}'
101. // >  /tmp/palabras/wordsAplic$1.txt
102.
103. imprime "idProcesoAplicacion= %d, calculando el numero de ocurrencias\n",idproc);
104. copiaCadena(fileNombre,"./ocurrencias.sh %d",idproc);
105. llamadaSistema(fileNombre);
106.
107. //se cuenta la cantidad de palabras en cada archivo
108. //en cada aplicacion
109. imprime "idProcesoAplicacion= %d, contando el numero de lineas en wordsAplic%d.txt\n",
              idproc,idproc);
110. copiaCadena(fileNombre,"/tmp/palabras/wordsAplic%d.txt",idproc);
111. datos_b = abreArchivo(fileNombre);
112.
113. si datos_b esta cerrado
114.     imprime "Error con el fichero";
115.     regresa 1;
116. sin-si
117.
118. lineas=0;
119. j=0;
120. haz
121.     espacio=leeCaracter(datos_b);
122.     si espacop='\n'
123.         lineas++;
124.     en otro caso si espacio=EOF
125.         j=1;
126.     fin-en otro caso
127. mientras j!=1;
```

```
128.
129. si datos_b esta cerrado
130.     imprime "...wordsAplic%d.txt, Numero de renglones=%d...",idproc,lineas;
131. en otro caso
132.     imprime "Error: fichero NO CERRADO";
133.
134. si idProcAplicacion=0
135.     para i=1 hasta i<nproc hacer
136.         resultadosParciales=0;
137.         recibe resultadosParciales del proceso aplicacion con id=i;
138.         total+=resultadosParciales;
139.         i++;
140.     fin-para
141.
142.     para i=1 hasta i<nproc hacer
143.         resultadosParciales=0;
144.         recibe resultadosParciales del proceso aplicacion con id=i;
145.         lineas+=resultadosParciales;
146.         i++;
147.     fin-para
148.
149.     imprime "Datos procesados:%ld, palabras diferentes=%d", total,lineas);
150.
151. en otro caso
152.     envia datosProcesados al proceso aplicacion con id=0;
153.     envia lineas al proceso aplicacion con id=0;
154.
155. fin-en otro caso
156.
157.}
```

```
\*****************************************************************************\
 La funcion obtieneItem recibe 5 parametros,
    *L: la lista de items.
    *D: la lista de items que contienen la informacion de los Segmentos a procesar.
    *item: un apuntador para el item de lista a obtener.
    *salida: la variable salida para modificar su valor en caso de ausencia total
            de items.
    nodo_desp: permite identificar si se esta realizando la lectura y creacion de
               items asociados a un segmento.
 Esta funcion se encarga de ejecutar la funcion DLML_Get(). Cabe mencionar que
 por cada item obtenido con DLML_Get, esta funcion regresa un 1, en caso de que no
 hayan mas items de lista, ni locales, ni remotos, ni segmentos a procesar, esta
 funcion cambia el valor de salida a 0 para terminar la ejecucion.
\*****************************************************************************\


1.  int obtieneItem(listaItems *L, listaSegmentos *D, nodo *item, int *salida,
                                                                  int nodo_desp){
```

```
2.
3.   valor=0;
4.
5.   si(L->longitud>0)
6.       DLML_Get(L,item);
7.       regresa 1;
8.   en otro caso si L->longitud=0 && D->longitud=0 && nodo_desp=0
9.       valor=DLML_Get(L,item);
10.      si valor=0;
11.          *salida=0;
12.          regresa 0;
13.      en otro caso
14.          regresa 1;
15.      fin-en otro caso
16.
17.  fin-en otro caso si
18.
19.  en otro caso
20.      regresa 0;
21.
22.  fin en otro caso
23.
24.}
```

Pseudocódigo de la aplicación de N-Reinas

```
#include <stdio.h>
#include "dlml.h"

1. main(){
2.
3.   listaItems L;
4.
5.   tiempo_inicial=0;
6.   tiempo_final=0;
7.   memoria_actual=0;
8.   umbral=0;
9.   procAplicacionTotales=0;
10.  procAplicacionLocales=0;
11.
12.  //DLBE_Procesadores() ya esta incluida en la biblioteca
13.  procAplicacionTotales=DLBE_Procesadores();
14.  procAplicacionLocales=PROCESADORES_POR_NODO;
15.
16.
17.  si idProcAplicacion=0
18.      tiempo_inicial=tiempo();
19.      leeTablero("tablero.txt",&L);
20.      Procesa(&L, procAplicacionTotales, idProcAplicacion);
```

```
21.
22.      tiempo_final=tiempo();
23.      imprime "Tiempo que tardo la ejecucion [%f] segundos", tiempo_final-tiempo_inicial;
24.
25.  en otro caso
26.      Procesa(&L, procAplicacionTotales, idProcAplicacion);
27.
28.  fin-en otro caso
29.
30.}
-------------------------------------------------------------------------------------
```

```
\*************************************************************************************\
 La funcion leeTablero recibe 2 parametros,
    *Tablero: el arreglo donde se almacena la solucion a explorar.
    *L: la lista de items que contienen las posibles soluciones a explorar.
 Esta funcion se encarga de construir el primer item en la aplicacion.
\*************************************************************************************\
```

```
1. leeTablero(char *Tablero, listaItems *L){
2.
3.   i=0; j=0;
4.   a=0;
5.   nodo *item;
6.
7.   FILE *datos_a;
8.   datos_a=abreArchivo(Tablero);
9.
10.  //TABLERO es definido por el usuario
11.  int tablero[TABLERO];
12.
13.  para i=0 hasta i<TABLERO hacer
14.      para j=0 hasta j<2 hacer
15.          a=convierteCadena_a_Entero( leeDato(datos_a) );
16.          si j==0
17.              tablero[0]=a;
18.              item=creaNodo(2,tablero);
19.              DLML_Insert(L,item);
20.          fin-si
21.          j++;
22.      fin-para-j
23.      i++;
24.  fin-para-i
25.
26. cierraArchivo(datos_a);
27.}
-------------------------------------------------------------------------------------
```

```
\*****************************************************************************\
 La funcion Procesa recibe 3 parametros,
    *L: la lista de items.
    nproc: el numero de procesos aplicacion en el sistema.
    idproc: el identificador del proceso aplicacion que invoco esta funcion.
 Esta funcion se encarga del procesamiento de items en la aplicacion.
\*****************************************************************************\


1. Procesa(listaItems *L, int nproc, int idproc){
2.
3.   i=0; col=0; datosProcesados=0; total=0; eliminados=0;
4.   num_sol=0;
5.   resultadosParciales=0;
6.   nodo item, *nuevo;
7.   char resPaciales[20];
8.   resultado="";
9.
10.  mientras( DLML_Get(L,&item) )
11.
12.      datosProcesados++;
13.
14.      //TABLERO es definido por el usuario al inicio de la ejecucion
15.      para col=1 hasta col<=TABLERO hacer
16.          si item.renglon<TABLERO
17.              si !es_comida(item.renglon,col,item.tablero)
18.                  item.tablero[item.renglon-1]=col;
19.                  nuevo=creaNodo(item.renglon+1,item.tablero);
20.                  DLML_Insert(L,&nuevo);
21.              fin-si
22.
23.              en otro caso
24.                  eliminados++;
25.
26.          en otro caso si !es_comida(item.renglon,col,item.tablero)
27.              num_sol+=1;
28.          i++;
29.      fin-para
30.
31.  fin-mientras-obtieneItem
32.
33.  si idProcAplicacion=0
34.
35.      FILE *archivo;
36.      char resPar[40];
37.
38.      para i=1 hasta i<nproc hacer
39.          resultadosParciales=0;
40.          recibe resultadosParciales del proceso aplicacion con id=i;
41.          total+=resultadosParciales;
42.          i++;
```

```
43.      fin-para
44.
45.      imprime "#Soluciones=%d", total);
46.
47.      //se abren archivo para escribir el resultado
48.      copiaCadena(resParciales,"Soluciones.txt");
49.      archivo=abreArchivo(resParciales);
50.      resultado=convierteEntero_a_Cadena(total);
51.      DLML_Write(archivo,"#Soluciones= ",resultado);
52.
53.      //se cierra el archivo de resultados del proceso aplicacion
54.      cierraArchivo(archivo);
55.
56.  en otro caso
57.      envia datosProcesados al proceso aplicacion con id=0;
58.
59.  fin-en otro caso
60.
61.}
```

```
\*****************************************************************************\
 La funcion es_comida recibe 3 parametros,
    reng: es el numero de renglones establecidos con reina.
    col: es la columna donde se va a colocar la posible siguiente reina.
    Tablero: es el tablero de NxN que contiene algunas o todas las N-Reinas.
 Esta funcion regresa V en caso de que la reina a colocar ataque alguna reina
 ya antes colocada, y regresa F en caso contrario.
\*****************************************************************************\

1. int es_comida(int reng, int col, int *Tablero){
2.
3.   i=0; j=0;
4.   comida=V;
5.
6.   reng=reng-1;
7.   i=reng-1;
8.
9.   //verifica columna
10.  mientras (i>=0) && (Tablero[i]!=col)
11.      i--;
12.  fin mientras
13.
14.  si i<0
15.      i=reng-1;
16.      j=col-1;
17.
18.      //verifica diagonal izquierda
19.      mientras (i>=0) && (j>=0) && (Tablero[i]!=j)
```

```
20.          i--; j--;
21.      fin-mientras
22.
23.      si (i<0) || (j < 0)
24.          i=reng-1;
25.          j=col+1;
26.
27.          //verifica diagonal derecha
28.          mientras (i>=0) && (j<=MAXSOL) && (Tablero[i]!=j)
29.              i--; j++;
30.          fin-mientras
31.
32.          si (i<0) || (j > MAXSOL)
33.              comida = F;
34.          fin-si
35.
36.     fin-si
37.
38.  fin-si
39.
40.  regresa comida;
41.}
```

Pseudocódigo de la aplicación de la Suma de Matrices

```
1.  #include <stdio.h>
2.  #include "dlml.h"
3.
4.  //macro para determinar la memoria libre
5.  #define MEMORIA_LIBRE (sysconf(_SC_PAGESIZE) * sysconf(_SC_AVPHYS_PAGES))
6.
7.  //macro para establecer el umbral de memoria
8.  #define UMBRAL 0.30
9.
10. //macro para obtener el numero de procesadores en el nodo
11. #define PROCESADORES_POR_NODO sysconf(_SC_NPROCESSORS_CONF)
12.
13.
14. main(){
15.
16.    listaItems L;
17.    listaSegmentos D;
18.
19.    tiempo_inicial=0;
20.    tiempo_final=0;
21.    memoria_actual=0;
22.    umbral=0;
23.    procAplicacionTotales=0;
24.    procAplicacionLocales=0;
```

```
25.
26.    //DLBE_Procesadores() ya esta incluida en la biblioteca
27.    procAplicacionTotales=DLBE_Procesadores();
28.    procAplicacionLocales=PROCESADORES_POR_NODO;
29.
30.    //calcula el umbral al 30% de la memoria actual
31.    memoria_actual=MEMORIA_LIBRE/procesosAplicacion+1;
32.    umbral=memoria_actual*UMBRAL;
33.
34.    si idProcAplicacion=0
35.      tiempo_inicial=tiempo();
36.      DLML_distributeOffsets("Matriz_a.txt", "Matriz_b.txt", &memoria_actual, &D);
37.
38.      //DISRIBUCION DE SEGMENTOS
39.      envia_lista_segmentos(&D,memoria_actual);
40.
41.      Procesa(&L,&D,procAplicacionTotales,idProcAplicacion,&memoria_actual,umbral);
42.
43.      tiempo_final=tiempo();
44.      imprime "Tiempo de ejecucion: [%f]segundos", tiempo_final-tiempo_inicial;
45.
46.   en otro caso
47.      Procesa(&L,&D,procAplicacionTotales,idProcAplicacion,&memoria_actual,umbral);
48.
49.   fin-en otro caso
50.
51.}
```

```
\*****************************************************************************\
 La funcion Procesa recibe 6 parametros,
    *L: la lista de items.
    *D: la lista de items que contienen la informacion de los Segmentos a procesar.
    nproc: el numero de procesos aplicacion en el sistema.
    idproc: el identificador del proceso aplicacion que invoco esta funcion.
    *memoria_actual: el contador de memoria actual (en bytes).
    umbral: es el umbral para la memoria actual en bytes.
 Esta funcion se encarga del procesamiento de items en la aplicacion.
\*****************************************************************************\


1.   Procesa(listaItems *L, listaSegmentos *D, int nproc, int idproc,
                                              long *memoria_actual, long umbral){
2.
3.   i=0; j=0; salida=1; datosProcesados=0; total=0; nodo_desp=0;
4.   resultadosParciales=0;
5.   FILE *resultados;
6.   nodoSegments segmento;
7.   infoSegmento datosSeg;
8.   nodo item;
```

```
9.   char resParciales[40];
10.  char coordenada [40];
11.  suma=0;
12.
13.  //descontar las variables y estructuras declaradas
14.  (*memoria_actual)=(*memoria_actual)-sizeof(datosSeg)-sizeof(9*int)-
                        sizeof(resParciales*2);
15.
16.   //se abren archivos locales para cada proceso aplicacion
17.   copiaCadena(resParciales,"/tmp/resultados_%d.txt",idproc);
18.   resultados=abreArchivo(copiaCadena);
19.
20.   recibe_lista_segmentos(D,memoria_actual);
21.
22.   mientras salida!=0
23.
24.      //obtener item con informacion de un segmento
25.      //en caso se que no se este cargando ninguno
26.      si D->longitud>0 && nodo_desp=0
27.          segmento=obtieneSegmento(D);
28.          (*memoria_actual)+=sizeof(nodoSegments);
29.  nodo_desp=1;
30.          datosSeg.datos_a=abreArchivo(segmento.archivo1);
31.          datosSeg.datos_b=abreArchivo(segmento.archivo2);
32.          datosSeg.reng=segmento.reng;
33.          datosSeg.col=segmento.col;
34.          datosSeg.desp_a=segmento.desp_a;
35.          datosSeg.desp_b=segmento.desp_b;
36.
37.          datosSeg.aux_i=0;
38.          datosSeg.aux_j=0;
39.          datosSeg.posx=0;
40.          datosSeg.carga_inf=0;
41.
42.          datosSeg.nodo_desp=&nodo_desp;
43.          datosSeg.L=L;
44.
45.      fin-si
46.
47.
48.      mientras DLML_Load(&datosSeg, memoria_actual, umbral, "+" )
49.
50.          mientras( obtieneItem(L,D,&item,&salida,datosSeg.nodo_desp) )
51.
52.              (*memoria_actual)+=sizeof(nodo);
53.              datosProcesados++;
54.
55.              //se realiza la suma y se almacena su resultado
56.              suma = item.a + item.b;
57.              copiaCadena(coordenada, "%d %d", item.pos_x, item.pos_y);
58.              DLML_Write( resultados, coordenada, suma );
```

```
59.
60.          fin-mientras-obtieneItem
61.
62.      fin-mientras-DLML_Load
63.
64.  fin-mientras-salida
65.
66.  imprime("idProcesoAplicacion= %d procesados=%d\n",idproc,procesados);
67.
68.  //se cierra el archivo de resultados del proceso aplicacion
69.  cierraArchivo(resultados);
70.
71.  si idProcAplicacion=0
72.      para i=1 hasta i<nproc hacer
73.          resultadosParciales=0;
74.          recibe resultadosParciales del proceso aplicacion con id=i;
75.          total+=resultadosParciales;
76.          i++;
77.      fin-para
78.
79.
80.      imprime "Datos procesados:%ld", total);
81.
82.  en otro caso
83.      envia datosProcesados al proceso aplicacion con id=0;
84.
85.  fin-en otro caso
86.
87.}
```

Pseudocódigo de la aplicación de la Multiplicación de Matrices

```
1.  #include <stdio.h>
2.  #include "dlml.h"
3.
4.  //macro para determinar la memoria libre
5.  #define MEMORIA_LIBRE (sysconf(_SC_PAGESIZE) * sysconf(_SC_AVPHYS_PAGES))
6.
7.  //macro para establecer el umbral de memoria
8.  #define UMBRAL 0.30
9.
10. //macro para obtener el numero de procesadores en el nodo
11. #define PROCESADORES_POR_NODO sysconf(_SC_NPROCESSORS_CONF)
12.
13.
14. main(){
15.
16.    listaItems L;
17.    listaSegmentos D;
```

```
18.
19.    tiempo_inicial=0;
20.    tiempo_final=0;
21.    memoria_actual=0;
22.    umbral=0;
23.    procAplicacionTotales=0;
24.    procAplicacionLocales=0;
25.
26.    //DLBE_Procesadores() ya esta incluida en la biblioteca
27.    procAplicacionTotales=DLBE_Procesadores();
28.    procAplicacionLocales=PROCESADORES_POR_NODO;
29.
30.    //calcula el umbral al 30% de la memoria actual
31.    memoria_actual=MEMORIA_LIBRE/procesosAplicacion+1;
32.    umbral=memoria_actual*UMBRAL;
33.
34.    si idProcAplicacion=0
35.      tiempo_inicial=tiempo();
36.      DLML_distributeOffsets("Matriz_a.txt", NULL, &memoria_actual, &D);
37.
38.      //DISRIBUCION DE SEGMENTOS
39.      envia_lista_segmentos(&D,memoria_actual);
40.
41.      Procesa(&L,&D,procAplicacionTotales,idProcAplicacion,&memoria_actual,umbral);
42.
43.      tiempo_final=tiempo();
44.      imprime "Tiempo de ejecucion: [%f]segundos", tiempo_final-tiempo_inicial;
45.
46.   en otro caso
47.      Procesa(&L,&D,procAplicacionTotales,idProcAplicacion,&memoria_actual,umbral);
48.
49.   fin-en otro caso
50.
51.}
```

```
\*****************************************************************************\
 La funcion Procesa recibe 6 parametros,
    *L: la lista de items.
    *D: la lista de items que contienen la informacion de los Segmentos a procesar.
    nproc: el numero de procesos aplicacion en el sistema.
    idproc: el identificador del proceso aplicacion que invoco esta funcion.
    *memoria_actual: el contador de memoria actual (en bytes).
    umbral: es el umbral para la memoria actual en bytes.
 Esta funcion se encarga del procesamiento de items en la aplicacion.
\*****************************************************************************\


1.   Procesa(listaItems *L, listaSegmentos *D, int nproc, int idproc,
                                                  long *memoria_actual, long umbral){
```

```
2.
3.   i=0; j=0; salida=1; datosProcesados=0; total=0; nodo_desp=0;
4.   resultadosParciales=0;
5.   FILE *resultados;
6.   long *desp_b;
7.   nodoSegments segmento;
8.   infoSegmento datosSeg;
9.   nodo item;
10.  char resParciales[40];
11.  char coordenada [40];
12.  valor=0; almInfo=0;
13.
14.  //descontar las variables y estructuras declaradas
15.  (*memoria_actual)=(*memoria_actual)-sizeof(datosSeg)-sizeof(9*int)-sizeof(resParciales*2);
16.
17.  //se abren archivos locales para cada proceso aplicacion
18.  copiaCadena(resParciales,"/tmp/resultados_%d.txt",idproc);
19.  resultados=abreArchivo(copiaCadena);
20.
21.  recibe_lista_segmentos(D,memoria_actual);
22.
23.  mientras salida!=0
24.
25.      //obtener item con informacion de un segmento
26.      //en caso se que no se este cargando ninguno
27.      si D->longitud>0 && nodo_desp=0
28.          segmento=obtieneSegmento(D);
29.          (*memoria_actual)+=sizeof(nodoSegments);
30.
31.          si almInfo=0
32.              //almacena informacion del archivo 2
33.              desp_b=pideMemoriaDeTamano(segmento.reng);
34.              (*memoria_actual)=(*memoria_actual)-sizeof(long*segmento.reng);
35.              guardaOffsetInicioDeCadaRenglon(desp_b,segmento.archivo2);
36.              almInfo=1;
37.          fin- si
38.  nodo_desp=1;
39.          datosSeg.datos_a=abreArchivo(segmento.archivo1);
40.          datosSeg.datos_b=abreArchivo(segmento.archivo2);
41.          datosSeg.reng=segmento.reng;
42.          datosSeg.col=segmento.reng+3;
43.          datosSeg.desp_a=segmento.desp_a;
44.          datosSeg.desp_b=desp_b;
45.
46.          datosSeg.aux_i=0;
47.          datosSeg.aux_j=0;
48.          datosSeg.aux_k=0;
49.          datosSeg.posx=0;
50.          datosSeg.carga_inf=0;
51.
52.          datosSeg.nodo_desp=&nodo_desp;
```

```
53.          datosSeg.L=L;
54.
55.      fin-si
56.
57.
58.      mientras DLML_Load(&datosSeg, memoria_actual, umbral, "*" )
59.
60.          mientras( obtieneItem(L,D,&item,&salida,datosSeg.nodo_desp) )
61.
62.              (*memoria_actual)+=sizeof(nodo);
63.              datosProcesados++;
64.
65.              //se calcula el valor de la coordenada y se almacena el resultado
66.              valor=0;
67.              para i=0 hasta i<numeroElementos(item.renglon[]) hacer
68.                  valor += item.renglon[i] * item.columna[i];
69.                  copiaCadena(coordenada, "%d %d", item.pos_x, item.pos_y);
70.                  DLML_Write( resultados, coordenada, valor );
71.                  i++;
72.              fin-para
73.
74.          fin-mientras-obtieneItem
75.
76.      fin-mientras-DLML_Load
77.
78.  fin-mientras-salida
79.
80.  imprime("idProcesoAplicacion= %d procesados=%d\n",idproc,procesados);
81.
82.  //se cierra el archivo de resultados del proceso aplicacion
83.  cierraArchivo(resultados);
84.
85.  si idProcAplicacion=0
86.      para i=1 hasta i<nproc hacer
87.          resultadosParciales=0;
88.          recibe resultadosParciales del proceso aplicacion con id=i;
89.          total+=resultadosParciales;
90.          i++;
91.      fin-para
92.
93.
94.      imprime "Datos procesados:%ld", total);
95.
96.  en otro caso
97.      envia datosProcesados al proceso aplicacion con id=0;
98.
99.  fin-en otro caso
100.
101.}
```

```
\*****************************************************************************************\
 La funcion guardaOffsetInicioDeCadaRenglon recibe 2 parametros,
    *desp_b: es un apuntador a un arreglo.
    *Matriz_b: es el archivo de la matriz b.
 Esta funcion se encarga de insertar el desplazamiento de inicio de cada renglon en
 el arreglo desp_b.
\*****************************************************************************************\

1. guardaOffsetInicioDeCadaRenglon(long *desp_b, char *Matriz_b){
2.
3.  i=1;
4.  caracter="";
5.  FILE *datos_b;
6.
7.  datos_b=abreArchivo(Matriz_b);
8.  desp_b[0]=0;
9.
10. haz
11.     caracter=leeCaracter(datos_b);
12.     si caracter='\n'
13.         desp_b[i]= desplazamiento(datos_b);
14.         i++;
15.     fin-si
16. mientras caracter!=EOF;
17.
18. cierraArchivo(datos_b);
19.}
-----------------------------------------------------------------------------------------
```

Referencias

[1] Top500 supercomputing sites. url: http://www.top500.org/lists/2012/06, Junio de 2012.

[2] Amazon Elastic Compute Cloud. url: http://aws.amazon.com/ec2, Julio de 2012.

[3] Wang, L., G. von Laszewski, M. Kunze, and J. Tao, Cloud computing: a perspective study, In *Proceedings of the Grid Computing Environments (GCE) workshop*, pages 137-146, 2008.

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

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

[6] 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): pages 14-24, 2007.

[7] E. L. Lusk, S. C. Pieper, and R. M. Butler. More scalability, less pain: A simple programming model and its implementation for extreme computing, *SciDAC Review*, 17: pages 30-37, 2009.

[8] Jeffrey Dean and Sanjay Ghemawat. *Mapreduce: simplified data processing on large clusters*, In OSDI04: Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation USENIX Association, 2004, pp 137-149.

[9] Li Wang, Zhiwei Ni, Yiwen Zhang, Zhang Jun Wu, Liyang Tang. Pipelined-MapReduced: An Improved MapReduce Parallel Programming Model. In *Proceedings of the 2011 4th*

International Conference on Intelligent Computation Technology and Automation, pages 871–874, 2011.

[10] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. MapReduce for Data Intensive Scientific Analyses. In *ESCIENCE '08 Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277-284, 2008.

[11] Yu-Fan Ho, Sih-Wei Chen, Chang-Yi Chen, Yung-Ching Hsu and Pangfeng Liu. A MapReduce Programming Framework Using Message Passing. In *Computer Symposium (ICS), 2010 International*, pages 883–888, 2010.

[12] Miguel Alfonso Castro García, *Programación con Listas de Datos para Cómputo Paralelo en Clusters*, Ph.D. Ciencias, Departamento de Computación, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.

[13] 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.

[14] Open MPI: Open Source High Performance Computing. url http://www.open-mpi.org, Junio de 2012.

[15] LAM/MPI Parallel Computing. url: http://www.lam-mpi.org, Junio de 2012.

[16] Tom White, *The Definitive Guide. OReilly*, 2009, 528 páginas.

[17] Chuck Lam, *Hadoop in Action. Manning*, 2010, 336 páginas.

[18] Juan Santana-Santana, Miguel A. Castro-García, Manuel Aguilar-Cornejo, and Graciela Roman-Alonso, *Load balancing algorithms with partial information management for the dlml library*, In Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on 2010, pp 64-68.

[19] Jorge Buenabad-Chávez, Miguel A. Castro-Garca, José L. Quiroz-Fabián, Edgar F, Hernández-Ventura, Graciela Román-Alonso, Daniel M. Yellin and Manuel Aguilar-Cornejo. Reducing Communication Overhead under Parallel List Processing in Mul-

ticore Clusters. In *The 2011 8th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE2011)*. pp 780-785.

[20] Apolo H. Hernández, Graciela Román-Alonso, Miguel A. Castro-García, Manuel Aguilar-Cornejo, Santiago Domínguez-Domínguez, Jorge Buenabad-Chávez. A Software Architecture for Parallel List Processing on Grids, In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics (ICPPAM)*, pages 720-729, 2011.

[21] Yagoubi, B., Medebber, M.: A load balancing model for grid environment. In *22nd International Symposium on Computer and Information Sciences, ISCIS 2007*, pp. 268-274, 2007.

[22] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. MapReduce for Data Intensive Scientific Analyses. In *ESCIENCE '08 Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277-284, 2008.

[23] Hadoop, *http://hadoop.apache.org*, Diciembre de 2011.

[24] HDFS. The Hadoop Distributed File System: Architecture and Design, *http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf*, Noviembre de 2012.

[25] IDC, *http://idc.com*, Octubre de 2012.

[26] John McCarthy, LISP 1.5 Programmer's Manual, The MIT Press, 1962.

[27] Twitter Engineering, *http://engineering.twitter.com*, Noviembre de 2012.

[28] Wittawat Tantisiriroj , Seung Woo Son , Swapnil Patil , Samuel J. Lang , Garth Gibson , Robert B. Ross, On the duality of data-intensive file system design: reconciling HDFS and PVFS. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 12-18, 2011.

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

[30] R. Sandberg, D. Goldberg, S . Kleiman, D. Walsh, and B. Lyon, Design and Implementation of the Sun Network Filesystem, *Innovations in Internetworking, Artech House*, Inc., Norwwod, MA, 1988.

[31] Network File System and Linux, *http://www.ibm.com/developerworks/linux/library/l-network-filesystems/index.html*, Noviembre de 2012.

[32] NFS Version 3 Protocol Specification, *http://tools.ietf.org/html/rfc1813*, Noviembre de 2012.

[33] Parallel Virtual File System, Version 2, *http://www.pvfs.org*, Noviembre de 2012.

[34] Freebase, *http://download.freebase.com/wex/2012-07-14*, Noviembre de 2012.