Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

"Procesamiento Paralelo de Listas de Datos en Clusters de Multicores"

Tesis que presenta

Edgar Fabián Hernández Ventura

Para Obtener el Grado de

Maestro en Ciencias

en Computación

Director de Tesis

Dr. Jorge Buenabad Chavez

México, Distrito Federal Febrero , 2012

Resumen

DLML (del inglés *Data List Management Library*) es una librería para el procesamiento paralelo de listas de datos. Los usuarios de DLML modelan sus datos como elementos de una lista y usan funciones DLML para insertar y obtener sus datos para procesarlos, similarmente al modelo productor/consumidor. Internamente, DLML maneja varias listas distribuidas, una por cada nodo del *cluster* donde se ejecuta una aplicación, y cuando una lista se vacía, DLML automáticamente obtiene datos de otros nodos. De esta manera, los aspectos de comunicación y balance de carga necesarios para un buen desempeño son ocultados a los programadores. DLML fue diseñado para *clusters* compuestos de nodos con un sólo procesador, y en base a procesos y paso de mensajes entre los mismos.

Esta tesis presenta MultiCore (MC) DLML, una versión de DLML diseñada para capitalizar mejor en el paralelismo *interno* de clusters compuestos de procesadores multinúcleo (multicore). MC-DLML está diseñado en base a hilos de ejecución, en comunicación de paso de mensajes y de memoria compartida; el paso de mensajes se usa para la comunicación entre procesadores y la memoria compartida para la comunicación entre hilos ejecutándose dentro de un mismo *multicore* mismo nodo. La tesis analiza varias organizaciones de listas de datos, y varios algoritmos de sincronización de memoria compartida para acceder a las listas, con el propósito de balancear la carga de trabajo dentro de cada nodo y entre distintos nodos.

Para evaluar el desempeño de MC-DLML utilizamos un *cluster* con procesadores *quad-core* con *hyper-threading* y varias aplicaciones con distintas granularidades de datos y distintos patrones de acceso a los mismos. Con algunas aplicaciones, MC-DLML corre hasta dos veces más rápido que DLML. Lo anterior fue posible una vez que se resolvieron varios aspectos relacionados con el *overhead* creado por el paso de mensajes debido al uso de múltiples hilos, localidad de datos en las cachés y el consumo de memoria.

Con los distintos algoritmos de sincronización de memoria compartida, el desempeño de MC-DLML varia dependiendo de la granularidad y el patrón de acceso de las aplicaciones, lo que sugiere el diseño de nuevos algoritmos o la integración de adaptabilidad en los algoritmos en tiempo de ejecución; este último aspecto sólo se analiza para trabajo futuro.

Agradecimientos

Quisiera agradecer a las instituciones, principalmente al Consejo Nacional de Ciencia y Tecnología CONACyT, por la beca que me otorgaron, la cual me ayudó a crear esta tesis y a crecer académica y personalmente. También agradecer al Centro de Investigación y de Estudios Avanzados del IPN el cual se convirtió en mi primera casa durante esta maestría. Su apoyo económico para congresos, beca terminal e instalaciones hicieron más fácil mi estancia, pero definitivamente fue la excelencia de su gente la que me impulsó a siempre dar "un poco más, una vez más".

Mi más humilde agradecimiento a mi hermano Felipe, el cual siempre ha sido mi compañero de aventuras y en esta ocasión el principal culpable de que yo iniciara esta empresa. Hoy que he terminado comparto este logro contigo hermano, esperando trazar una vida que nos haga sentir orgullosos a ambos y ansioso de ver que nuevos retos nos depara el camino.

No se me olvidan mis padres Jesús y Marbella que siempre han sido mi faro moral y personal. Gracias a su apoyo y cariño hoy doy un paso más adelante en mi vida. Sin su ejemplo de trabajo y dedicación a la profesión no hubiera perseguido la senda del intelecto y me hubiera perdido en la urgencia del ahora. Muchas gracias.

Quiero agradecer a mis amigos y colegas que han hecho de los momentos más oscuros y estresantes, anécdotas divertidas. En especial quiero agradecer a Alejandro, Luis, Temoc, Mancillas, Saúl y Fernando. Para mi no queda duda que he llegado hasta aquí por ustedes y a pesar de ustedes.

Un especial agradecimiento a Sofy Reza por todo el apoyo que me ha dado. Gracias por las palabras de aliento, tu carácter amable y cariñoso son parte de los cimientos de este departamento.

Por último, pero no menos importante quiero agradecer a Graciela por salvarme. Tu compañía y cariño ha sido mi impulso para la recta final. La causalidad y esta maestría nos han encontrado, gracias por ser parte de mi equipo.

Índice general

Índice de figuras

Glosario

C

cluster Conjunto de procesadores conectados por una interfaz de comunicación rápida que funcionan como una sola computadora.

D

DLML Data List Management Library. Librería para asistir al desarrollo de aplicaciones paralelas utilizando listas de datos.

M

MIMD Multiple Instruction, Multiple Data. Modelo paralelo para la ejecución de códigos que pueden ser distintos y pueden tener entradas diferentes de datos. Cada procesador o núcleo tiene su propia unidad de control para controlar la ejecución de su código.

MPI Message Passing Interface. Biblioteca para el desarrollo de aplicaciones paralelas que permite el control, la comunicación y sincronización de múltiples procesos.

multicore Procesadores con dos o más núcleos dentro de un chip. Estos núcleos comparten memoria y en algunos casos memoria caché.

mutex Mutual exclusion lock. Una variable compartida para restringir el acceso a una sección critica a sólo un proceso si es compartida por dos o más procesos.

S

SIMD Single Instruction, Multiple Data. Modelo paralelo para la ejecución de un solo código con distintos datos de entrada para producir resultados

distintos. La ejecución es controlada por una misma unidad de control que sincroniza a todos los procesadores o núcleos.

spinlock Una variable compartida similar al mutex para restringir el acceso a una sección critica a sólo un proceso si es compartida por dos o más procesos. Internamente, el spinlock hace una espera activa preguntando constantemente el valor de la variable para saber si ha cambiado.

Capítulo 1

Introducción

Los *clusters* son la arquitectura paralela para el cómputo de alto desempeño más usada alrededor del mundo. Su aceptación generalizada es por su bajo costo debido a su configuración basada en componentes de propósito general; inicialmente fueron *desktops* interconectados a través de hardware de red de área local. Estos componentes se han estilizado en su presentación y han mejorado continuamente su desempeño con el tiempo. *Top 500*, el referente de las computadoras más rápidas del mundo, están construidas sobre una arquitectura *cluster* [2]. Los *clusters* colocaron el cómputo paralelo al alcance de todos los interesados a finales de los 90s.

Con la llegada de los procesadores multinúcleo (*multicore*), los cuales integran dos o más núcleos de procesamiento (*CPUs*) dentro de un solo chip, se modifica el desarrollo de software. Las aplicaciones deben de explotar los ahora comunes *multicores* para no desperdiciar recursos, y aprovechar al máximo las ventajas que ofrece el cómputo paralelo, como mayor velocidad de procesamiento y la ejecución initerrumpida de procesos en el fondo que agreguen características al software (por ejemplo incrementar la seguridad). Estas nuevas arquitecturas orillan a la industria y a la academia al desarrollo de nuevas interfaces y tecnologías para esconder la complejidad de la computación paralela a los desarrolladores [3]. Los *clusters* también están evolucionando para integrar nodos *multicores*, creando así dos niveles de paralelismo: i) paralelismo *externo*, entre los procesadores del *cluster* , y ii) paralelismo *interno*,

entre los núcleos de cada *multicore*.

Esto hace a la programación paralela más complicada que la programación secuencial. Una aplicación secuencial consiste en la especificación de un algoritmo que procesa datos de entrada para producir una salida. Además de esta especificación, una aplicación paralela requiere de la especificación de un protocolo de comunicación entre varios procesadores para compartir código y datos, y para coordinar las tareas de los mismos. La mayoría de las aplicaciones paralelas se basan en el modelo computacional *Single Instruction Multiple Data* (SIMD). En SIMD, todos los procesadores (o núcleos dentro de un *multicore*) ejecutan el mismo programa pero procesan datos distintos. Los datos que serán procesados, o la *carga de trabajo*, deben ser particionados y asignados de manera equitativa entre los procesadores, debido a que el tiempo total del procesamiento paralelo depende del tiempo que toma al último procesador en terminar su parte de trabajo. Una aplicación paralela puede tener una distribución desigual de su carga de trabajo entre los distintos núcleos de trabajo, esto puede ocurrir cuando: i) una aplicación genera más trabajo dinámicamente, ii) los recursos de hardware son compartidos por medio de multiprogramación, y iii) el *cluster* consiste de recursos heterogéneos con algunos núcleos/procesadores más lentos que otros. Este balance de carga puede ser codificado por el programador para aumentar el desempeño en cada aplicación, lo cual incrementa aun más la dificultad de la programación.

Para facilitar el desarrollo de aplicaciones paralelas se han propuesto *middlewares* cuyo propósito es ocultar los aspectos de paralelismo y/o balance de carga del programador, entre otros: OpenMP, Skeletons, **DLML** (Data List Management Library), y Mapreduce. OpenMP [4] es un preprocesador que transforma aplicaciones secuenciales en aplicaciones paralelas multi-hilos, que se comunican por medio de memoria compartida. Este enfoque permite capitalizar en el paralelismo intra-nodo dentro de un solo *multicore*; es adecuado principalmente para aplicaciones secuenciales de usuarios finales (por ejemplo, modelos matemáticos) que procesan variables o vectores en ciclos *for* y pueden ser fácilmente paralelizados. Sin embargo para utilizar

más procesadores *multicore*, es necesario especificar comunicación basada en paso de mensajes, para coordinar el paralelismo interno debido a que OpenMP no proporciona esta funcionalidad.

Skeletons [5, 6] y Mapreduce [7] son *middlewares* que ofrecen modelos de programación que abstraen la complejidad, y que ocultan del programador varios aspectos del cómputo paralelo y del balance de carga. Estos *middlewares* fueron inicialmente diseñados para ejecutarse en *clusters* compuestos de nodos con un solo procesador. Con *clusters* de nodos *multicore*, es necesario rediseñarlos para capitalizar mejor en el paralelismo de ambos niveles, externo e interno. Este es el contexto de nuestro trabajo.

DLML [8] es una librería para procesar listas de datos en paralelo de manera transparente al usuario. Los usuarios DLML solo tienen que organizar sus datos en elementos de lista y utilizar funciones de DLML para insertar nuevos elementos y extraer elementos existentes para procesarlos. DLML se encarga del balance y distribución automático de los datos en el *cluster*. Las funciones DLML ocultan la comunicación de sincronización a los usuarios, y el relleno automático de listas tiende a balancear la carga de trabajo de acuerdo a la capacidad de procesamiento de cada procesador.

La primera versión de DLML [8, 9] fue diseñada para *clusters* compuestos de procesadores con un sólo núcleo utilizando MPI [10] para el paso de mensajes. En esa versión, en cada nodo (con un solo núcleo), un proceso ejecuta el código de la aplicación y su proceso *hermano* DLML está a cargo de: i) *hacer* peticiones de datos a nodos remotos cuando la lista *local* se vacia, y ii) *servir* peticiones de datos de procesos remotos. Ambas tareas siguen un protocolo basado en paso de mensajes. Paso de mensajes es también usado entre un proceso de aplicación y su proceso DLML *hermano* (en el mismo procesador) para migrar elementos de datos entre sus espacios de direcciones.

La primera versión de DLML también se puede ejecutar en *clusters* en procesadores *multicore* sin modificación alguna, y capitalizar en paralelismo interno dentro de cada procesador, ejecutando en paralelo en los varios núcleos disponibles dentro

del *multicore*. Solo es necesario ejecutar un proceso de aplicación y un proceso DLML por cada núcleo. Sin embargo, se utiliza el paso de mensajes entre procesos DLML ejecutandose en el mismo procesador.

Esta tesis presenta MultiCore (MC) DLML, un nuevo diseño de DLML que utiliza comunicación de paso de mensajes para paralelismo externo (entre *multicores*), y comunicación de memoria compartida para paralelismo interno de los procesadores *multicore*.

Los objetivos que abarca esta tesis son la creación de una versión más eficiente que capitalice el uso de procesadores *multicores* en un *cluster*. Por lo tanto se deben crear y analizar distintas alternativas de balance interno dentro del procesador y balance externo entre los diferentes procesadores. Una mejora obvia es reducir el costo de la comunicación sustituyendo los mensajes entre los núcleos por memoria compartida. Esto no elimina la necesidad de comunicarse con otros nodos, por lo que se conserva un proceso o hilo en cada núcleo para realizar balance de carga entre nodos.

Esta tesis presenta un análisis de distintas organizaciones de comunicación de memoria compartida para MC-DLML, incluyendo: distintas organizaciones de listas y distintos mecanismos de balance de carga basados en el robo datos entre las mismas. Se analizan también aspectos relacionados con la localidad de acceso en las *cachés* y el consumo de memoria por parte de algunas aplicaciones utilizadas como ejemplo.

1.1. Organización de la Tesis

La tesis continua como sigue:

El Capítulo 2 presenta al cómputo paralelo desde distintas perspectivas: las aplicaciones que lo requieren, las arquitecturas que lo llevan a cabo, y los elementos de programación básicos que lo especifican. Este capítulo sirve para introducir la terminología que es usada en capítulos subsecuentes.

El Capítulo 3 presenta varios *middlewares* para cómputo paralelo: su motivación, su organización y su funcionamiento de manera general. El capítulo presenta con más

detalle la primera versión de DLML, basada en paralelismo entre procesos y paso de mensajes.

El Capítulo 4 presenta un análisis de distintos aspectos de diseños para una nueva versión de DLML basado en procesadores *multicore*. Se presentarán las principales opciones para la organización de las listas de datos y lo que implica con respecto a los algoritmos de balance interno y externo ligados con cada organización. También describe nuestros algoritmos para el balance de carga intra-nodo. Se detallarán cuatro algoritmos: Basic-Locking (BL), Global-Locking (GL), Low-Sync-Locking (LSL) y Thread Locking (TL). Además se explicarán nuestras decisiones para la implementación final de las listas de datos y el método de balanceo de carga externo.

El Capítulo 5 exhibe la plataforma experimental. Se describe el hardware utilizado, las aplicaciones de ejemplo para medir el desempeño de los diferentes algoritmos y el software utilizado para el desarrollo de la implementación de MC-DLML.

El Capítulo 6 muestra los resultados obtenidos con nuestros algoritmos utilizando diferentes aplicaciones de ejemplo con diferentes granularidades para medir su desempeño. Los resultados mostrados son medidos en un solo nodo para apreciar mejor el desempeño de la sincronización interno y en el *cluster* utilizando de 1 a 6 nodos para medir el desempeño externo de cada algoritmo.

El Capítulo 7 muestra nuestras conclusiones obtenidas del trabajo realizado. Se explican nuestros resultados y su relevancia, además se sugiere posible trabajo a futuro.

Capítulo 2

Cómputo paralelo

El cómputo paralelo consiste en la ejecución simultánea de varias instrucciones de una aplicación con el fin de reducir el tiempo de respuesta. Ha sido investigado desde los inicios de la computación a finales de los años 50 del siglo pasado, y se ha usado desde los 70s para resolver en un tiempo adecuado aplicaciones que procesan gran cantidad de datos, como son la predicción del clima y otros fenómenos de la naturaleza. Esta aplicaciones son típicamente conocidas como cómputo científico y de alto desempeño.

El uso del cómputo paralelo estuvo restringido hasta mediados de los 90s a tan sólo a unas pocas instituciones y gobiernos por el excesivo costo, relativamente alto, de las arquitecturas paralelas. No obstante la Internet facilitó compartir las arquitecturas paralelas disponibles a más usuarios. A mediados de los 90s, las redes locales de computadoras personales empezaron a ser utilizadas como arquitecturas paralelas. Esta arquitectura fue eventualmente llamada *cluster*, y se ha refinado desde entonces en calidad y desempeño, en parte gracias a la demanda de sus componentes a nivel mundial. Hoy en día los *clusters* son la arquitectura paralela más extensamente utilizada para el cómputo de alto desempeño.

La llegada de los procesadores multinúcleo (*multicore*) a principios de este siglo, ha establecido al cómputo paralelo como una tecnología básica: indispensable no solo para aplicaciones de cómputo científico sino para todo tipo de aplicaciones, incluyendo aplicaciones empresariales y aplicaciones embebidas [11].

Este capítulo presenta una visión global del cómputo paralelo desde la perspectiva de las aplicaciones que lo requieren, de las distintas arquitecturas que se han propuesto para llevarlo a cabo, y de los elementos de programación utilizados para desarrollar aplicaciones paralelas. La terminología y conceptos que se presentan en este capítulo son fundamentales para poder entender el contenido de los siguientes capítulos.

2.1. Aplicaciones

Las aplicaciones de software pueden clasificarse en dos grandes grupos por el tipo de procesamiento que realizan: aplicaciones numéricas y aplicaciones simbólicas. Las aplicaciones numéricas realizan principalmente sumas, restas, multiplicaciones y divisiones. Las aplicaciones simbólicas realizan principalmente operaciones de comparación y modificación de cadenas de caracteres. Esta sección describe de manera general algunas aplicaciones de cada tipo y algunos aspectos de su paralelización.

2.1.1. Aplicaciones numéricas

Las aplicaciones numéricas corresponden típicamente a un modelo matemático de un proceso de la naturaleza. Su propósito es determinar la variación en el tiempo de ciertas magnitudes. Por ejemplo: en la predicción del clima se desea conocer como cambia la temperatura, la humedad, etc.; en la predicción del movimiento de un planeta o una estrella se desea conocer como varían sus coordenadas. Generalmente, esas magnitudes dependen de otras que también varían en el tiempo: la presión atmosférica y la velocidad del viento en el caso de la predicción del clima, o la posición y la masa de otros planetas en el caso de la predicción del movimiento de un planeta.

Un modelo matemático especifica y resuelve una función que define la dependencia de ciertas variables (magnitudes) sobre otras en el tiempo. La función es primero definida de manera general para cada punto en el *continuo* infinito del espacio y el tiempo. Para poder resolverla con precisión y en tiempo adecuado es necesario discretizarla en ambos, espacio y tiempo. La solución consiste en asignar valores

iniciales a los puntos del espacio discretizado correspondientes a un tiempo t. Entonces se calcula el modelo en intervalos de tiempo, seleccionados a partir de t para cada punto del espacio discretizado. Entre más puntos se calculen en el espacio y el tiempo, mayor será la precisión del modelo. Es por esto que el cómputo paralelo es necesario para el cálculo de aplicaciones de este tipo: mayor precisión con un tiempo de respuesta razonable. Un ejemplo de este tipo de aplicaciones puede ser visto en la Figura 2.1

Es importante notar que el problema que resuelve una aplicación, determina la eficiencia de su paralelización, es decir, si es escalable o no lo es. La predicción del clima es, en general, escalable ya que diferentes procesadores pueden calcular la función correspondiente para diferentes puntos en el espacio casi de manera independiente. La predicción del movimiento de un planeta o una estrella o moléculas *no son* escalables en general. Los cuerpos (planetas, etc.) son representados como una estructura de datos moviéndose en un espacio discretizado, calculando su interacción con otros cuerpos y esto involucra mucha comunicación entre los procesadores.

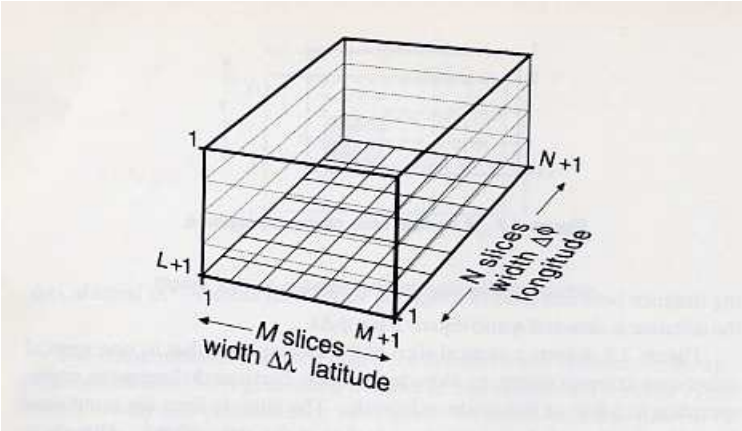

Figura 2.1: Ejemplo de ejecución paralela de una aplicación numérica

2.1.2. Aplicaciones simbólicas

Las aplicaciones simbólicas realizan principalmente operaciones de comparación y modificación de cadenas de caracteres. Ejemplos típicos incluyen: consultas a bases de

datos y encontrar soluciones en un espacio de búsqueda en aplicaciones de inteligencia artificial, entre otras.

En consultas de bases de datos, el cómputo paralelo consiste en particionar las relaciones (tablas de datos) de manera que diferentes procesadores procesan parte de una consulta con parte de las relaciones involucradas. Además, si una consulta consiste de varios operadores relacionales, diferente procesadores procesan un operador distinto.

En inteligencia artificial existen varios métodos que exploran espacios de búsqueda para encontrar una solución o un conjunto de soluciones. Cada solución representa una posición en el espacio de búsqueda, el cual se representa como una cadena de caracteres. Generalmente estos espacios de búsqueda son demasiado grandes para ser explorados completamente por lo que se aplican heurísticas para agregar aleatoriedad a la búsqueda, esperando encontrar la mejor solución de todas en el menor tiempo. Las soluciones encontradas serán alteradas y recombinadas para producir nuevas soluciones, que a su vez serán evaluadas para medir si son mejores que las anteriores. Cada evaluación puede tomar un tiempo considerable, pero tiene la ventaja de ser un proceso altamente paralelizable. Es por eso que el cómputo paralelo es una herramienta importante en la evaluación de distintas soluciones en el área de inteligencia artificial.

2.2. Arquitecturas paralelas

Los primeros procesadores fueron construidos en base al modelo de von Newman y consistían de una unidad de control (CU), una unidad aritmético lógica (ALU) y varios registros. Uno de estos registros, el contador de programa (PC), contiene la dirección en memoria de la siguiente instrucción a ejecutar. La operación básica es como sigue.

1. la CU lee la siguiente instrucción a ejecutar, cuya dirección está en el PC, y la almacena en el registro de instrucción (IR),

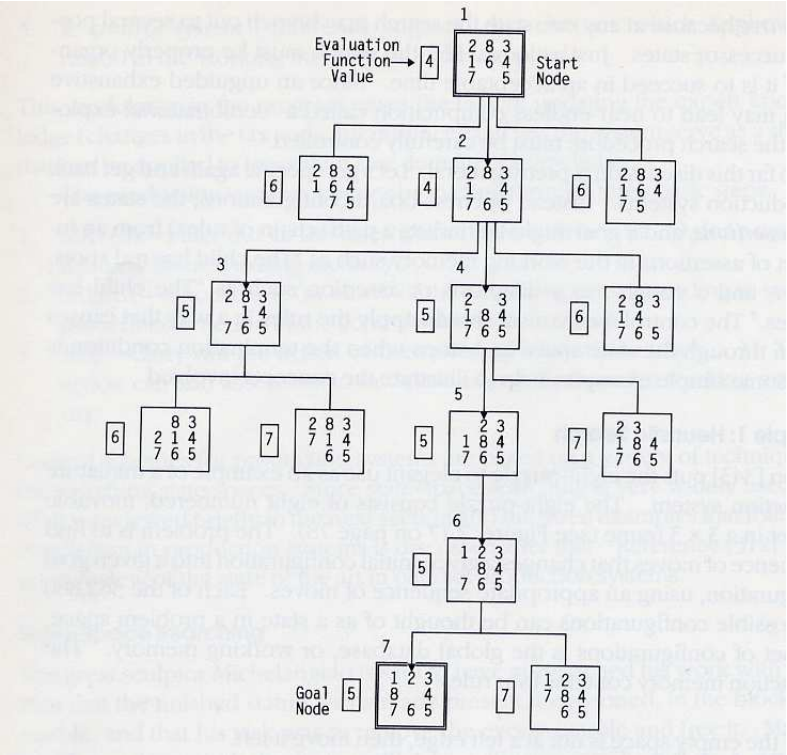

Figura 2.2: Ejemplo de ejecución paralela de una aplicación simbólica

2. la CU interpreta la instrucción en el IR, determina su tamaño en bytes, e incrementa el PC para que apunte a la siguiente instrucción a ejecutar.

3. la CU ordena a la ALU que ejecute la instrucción actual: una suma, resta, multiplicación, etc. La CU vuelve al paso 1.

Las arquitecturas paralelas difieren de una máquina secuencial porque se puede ejecutar más de una instrucción a la vez. Las más usadas han sido las arquitecturas SIMD (Single Instruction Multiple Data) y las arquitecturas MIMD (Multiple Instruction Multiple Data). Como su nombre lo indica, las arquitecturas SIMD ejecutan una sola instrucción sobre múltiples datos distintos a la vez. En efecto, su funcionamiento es equivalente a ejecutar la misma instrucción (sume, resta, multiplicación, etc.) sobre datos distintos en una máquina secuencial. Una máquina secuencial tiene una CU, una ALU y registros. Una máquina SIMD tiene también solo una CU, pero tiene múltiples ALUs y registros. Así, la CU ordena la ejecución de una misma

instrucción a las múltiples ALUs al mismo tiempo sobre distintos datos produciendo distintos resultados.

Las arquitecturas MIMD consisten de varios procesadores secuenciales interconectados por medio de memoria compartida o por una línea de comunicación punto a punto. Cada procesador de una computadora MIMD consiste de una CU, una ALU y registros.

2.2.1. Arquitecturas SIMD

La figura 2.3 muestra la arquitectura SIMD de la Illiac IV de la Universidad de Illinois. En la figura, una PU (processing unit) corresponde a una ALU.

Cada ALU de la Illiac IV tiene 16 KB de memoria. La Illiac IV fue diseñada para resolver aplicaciones de matrices y ecuaciones diferenciales discretizadas en espacio y tiempo.

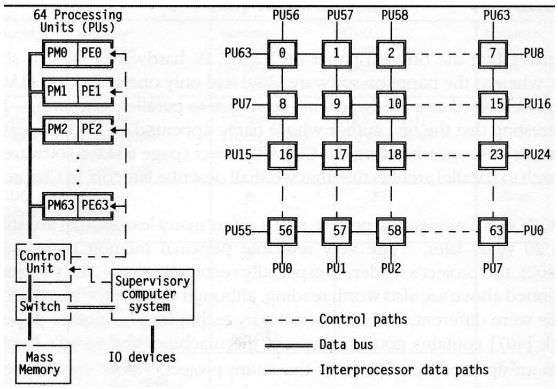

Figura 2.3: Arquitectura de la Illiac IV

Cray-1 (1976) es una *pipelined* SIMD: las fases de una operación sobre cada uno de los datos de un vector se traslapan. la Cray X-MP (1982) tiene 4 procesadores (MIMD-SIMD). la Cray-2 (1985) tiene 4 procesadores y 256 millones de palabras de 64 bits (2 GB). la Cray C-90 (1992) tiene 16 procesadores y 4 GB de memoria. Cray anunció en 1993 máquinas tipo MIMD.

LA CM-1 (1980) era una SIMD para procesamiento simbólico. Tenía 65536 ALUs de 1-bit. Cada uno podía realizar las operaciones: ADD, AND, OR, MOVE, SWAP (sin operaciones de punto flotante). CM-2 (1988) era una SIMD también, pero con 8 GB y un co-procesador de punto flotante por cada 32 ALUs. CM-5 (1991) es una MIMD (parcialmente sincronizada): puede incluir 4 vectores entre cada procesador y su memoria puede crecer hasta 16K nodos.

Otras máquinas SIMD incluyen: CDC Cyber 205, IBM 3090 y ES 9000, NEC SX, IBM GF11, MasPar MP-1, MP-2, etc. los cuales son variantes de los modelos anteriores. detalles de las mismas (y otras arquitecturas) se pueden encontrar en [1]. Una cronología de la evolución de las computadoras paralelas puede se puede ver en la Figura 2.4 donde se muestra el aumento casi exponencial de las las operaciones de punto flotante por segundo alcanzadas por cada una.

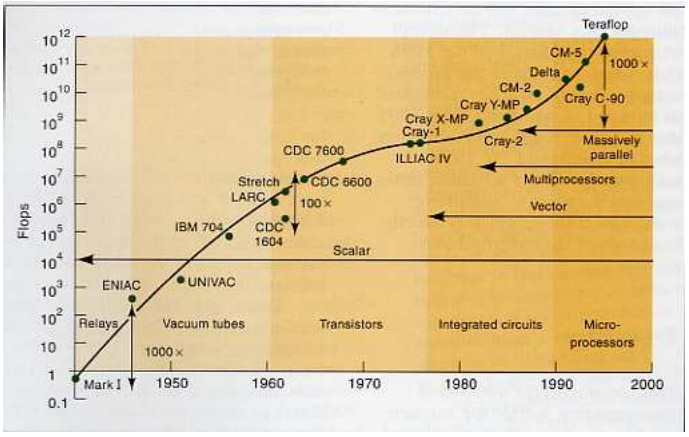

Figura 2.4: Evolución de las computadoras con arquitecturas paralelas [1, p. 33]

2.2.2. Arquitecturas MIMD

Las arquitecturas MIMD fueron motivadas por la llegada del microprocesador a principios de los 80s. Además de su uso en computadoras personales, se pensó utilizar al procesador para construir arquitecturas paralelas económicas por medio de interconectar múltiples procesadores, es decir: arquitectura paralela = procesadores + medio de interconexión.

Existen varios tipos de arquitecturas MIMD. Las multicomputadoras consisten de varios procesadores, cada uno con su memoria privada, e interconectados por medio lineas de comunicación punto a punto. Este tipo de interconexión hace a las multicomputadoras escalables, es decir el número de procesadores que se puede interconectar es relativamente grande, hasta miles. Ejemplos de multicomputadoras, incluyen: iPSC, NCUBE, IBM Victor, Meiko, entre otras.

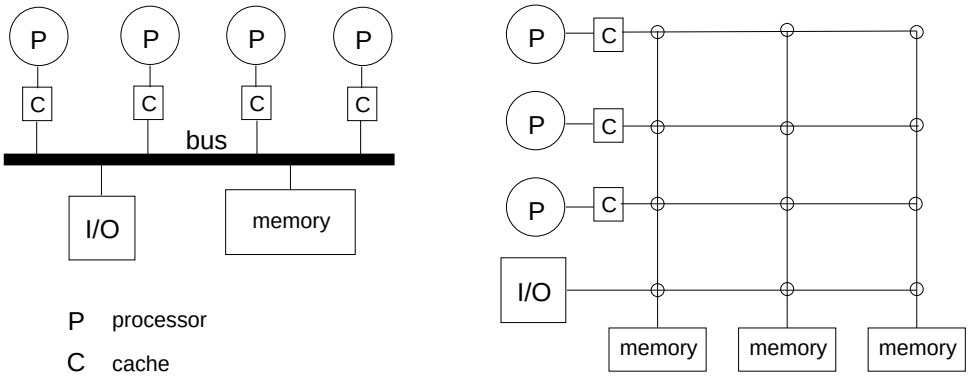

Figura 2.5: Multiprocesador basado en bus (izquierda) y en cross-bar switch.

Los multiprocesadores consisten de varios procesadores interconectados (comunicados) por medio de una memoria común, la cual comparten para acceder código y datos. La memoria compartida es un cuello de botella y no permite que los multiprocesadores escalen más allá de unas pocas decenas de procesadores. La Figura 2.5 muestra dos tipos de interconexión a la memoria típicamente usados en multiprocesadores: bus y cross-bar switch. Ejemplos de multiprocesadores incluyen: IBM 370/168, HEP, Sequent Symmetry, Silicon Graphics, entre otros.

Los multiprocesadores fueron ampliamente utilizados en los 90 y principios del 2000, y han reencarnado en lo que hoy llamamos los procesadores *multicore*. Los *multicores* son equivalentes a los multiprocesadores, solo que todos los procesadores (llamados núcleos) están en un solo chip, gracias a los avances en la integración de circuitos integrados. Nótense las memorias *caché* en la Figura 2.5 entre cada procesador y la memoria. Las cachés juegan un papel fundamental para que los *multicores* y los multiprocesadores tengan un buen desempeño, pues su tiempo de acceso es más

rápido que el de la memoria principal. Los *multicores* actuales tienen hasta 3 niveles de cachés.

En los 90s también se investigaron multiprocesadores *escalables*, arquitecturas MIMD que soportaban un espacio de direcciones compartido sobre memoria distribuida. Algunos ejemplos son: RP3, Cray T3D, DASH, Alewife, DDM, KSR-1, COMA-F y Origin [1]. Con estas computadoras se obtuvieron muy buenos resultados, pero su costo no permitió que su uso fuese generalizado. El principal factor de su costo relativamente alto fue el medio de interconexión, e igualmente paso con las multicomputadoras y los multiprocesadores. El costo del medio de interconexión, por corresponder a un diseño especial, era relativamente alto.

Una vez que llegaron los *clusters*, las demás arquitecturas paralelas fueron utilizadas sólo por grandes compañías. Los *clusters* fueron inicialmente computadoras de escritorio interconectadas a través de una red de área local. Estas redes fueron adoptadas por pequeñas y medianas empresas a nivel mundial para compartir recursos (datos y aplicaciones) y tener acceso a Internet. Debido a su bajo precio, su calidad y velocidad aumentaron dramáticamente en unos pocos años. Los *clusters* son arquitecturas tipo MIMD. Actualmente consisten de nodos *multicore* lo cual incrementa el nivel de paralelismo en ambos niveles, entre los distintos procesadores y dentro de cada núcleo, por lo que la programación para tales sistemas se ha vuelto más complicada.

2.3. Programación paralela

Las arquitecturas SIMD no requieren programación paralela. El compilador del lenguaje de alto nivel, generalmente Fortran, se encarga de generar el código adecuado para utilizar las múltiples ALUs. Aunque esto no es trivial, tampoco es muy difícil debido a que solo hay una unidad de control que está ejecutando una sola instrucción a la vez sobre múltiples datos. El compilador detecta las operaciones sobre arreglos de vectores y carga los datos correspondientes sobre los múltiples ALUs.

En contraste con las máquinas SIMD, las máquinas MIMD sí requieren de programación paralela, debido a que cada procesador tiene una CU y ejecuta instrucciones conforme las va leyendo y al paso indicado por su propio reloj. Aún cuando estén ejecutando las mismas instrucciones, los procesadores se desfasan y deben sincronizarse en algunos puntos que deben ser especificados por el programador. Además, el programador debe particionar la carga de trabajo (datos a procesar) y asignar su parte a cada procesador. *Single Instruction, Multiple Data* (SIMD) es una técnica para especificar aplicaciones paralelas dividiendo la carga de trabajo y especificando los puntos en los cuales se realizará una sincronización de los datos. Se ejecutará la misma aplicación simultáneamente en múltiples procesadores con datos distintos para obtener resultados más rápido.

En otras palabras, la programación paralela consiste en dividir la carga de trabajo y en especificar un protocolo de comunicación entre los procesadores para: i) compartir código y datos, y ii) para sincronizar el trabajo de los procesadores. La comunicación entre procesadores produce un *overhead*, y consecuentemente debe de ser eficiente: a menor tiempo de comunicación, mayor escalabilidad.

2.3.1. Paso de mensajes

En multicomputadoras y *clusters* se utiliza paso de mensajes para ambos tipos de comunicación: compartir código y datos y especificar sincronización. Programar un algoritmo paralelo con paso de mensajes no es simple, pero afortunadamente ahora es portable gracias a la aceptación generalizada de la Message Passing Interface (MPI) [10]. Su adopción generalizada fue en parte debida al uso también generalizado de los *clusters*.

MPI sólo brinda las funciones necesarias para establecer los canales de comunicación y enviar mensajes, sin embargo es responsabilidad del usuario especificar un protocolo coherente de comunicación de acuerdo a las necesidades de sincronización de su aplicación. Un ejemplo típico de este tipo de protocolo es la inicialización de la ejecución. Es necesario especificar el orden en el cual se dividirán los datos

entre los diferentes nodos para que sean procesados. Los nodos que aun no tienen datos deben de esperar que los datos le sean enviados por medio de mensajes para iniciar el procesamiento. Esta espera es generalmente implementada con una barrera MPI_Barrier. Para enviar los datos a otro proceso se utiliza el par de funciones MPI_Send y MPI_Recv. Cuando el proceso termine se establecerá un protocolo de terminación, el cual puede ser implementado con otra barrera.

MPI describe en su especificación cuatro niveles de soporte para ser utilizado en ambientes multihilo. *Single* para el nivel más básico con un solo proceso que realiza todas las llamadas a funciones MPI. *Funneled* para aplicaciones con múltiples hilos, donde solamente un hilo puede hacer llamados MPI. *Serialized* donde todos los hilos pueden hacer llamados a funciones MPI, pero son ejecutados secuencialmente para controlar su concurrencia. Finalmente el nivel de soporte *Multiple* donde todos los hilos pueden hacer llamados a funciones concurrentemente.

2.3.2. Memoria compartida

En multiprocesadores y *multicores*, la memoria compartida evita el especificar comunicación para compartir código y datos. Pero se requiere especificar comunicación para sincronizar a los núcleos por medio de variables compartidas en memoria. Existen bibliotecas para la creación de estás variables como por ejemplo *pthreads* para el lenguaje C.

Semáforos

Es un tipo abstracto de datos para controlar el acceso de múltiples hilos a un recurso compartido. Una manera simple de visualizar un semáforo es como un registro de cuantas unidades de un recurso están disponibles conjuntamente con las operaciones para ajustar el registro o en dado caso, esperar hasta que más unidades estén disponibles. Si el semáforo permite contar una cantidad arbitraria de unidades de un recurso es llamado un semáforo contador. Si su valores está restringidos a 0 y 1 es llamado un semáforo binario.

Candados mutex

Los candados de exclusión mutua (*mutex*) son equivalentes a un semáforo binario. De igual manera se usan para evitar el acceso concurrente a recursos compartidos y secciones críticas. Existen dos tipos de *mutex*: el mutex regular y el *spinlock*. El mutex regular detiene la ejecución del hilo cuando otro hilo ha adquirido el candado y bloqueado la sección crítica; cuando el candado es liberado el hilo continua su ejecución y adquiere el candado. El *spinlock* hace una espera activa del núcleo del procesador verificando constantemente si el candado ha sido liberado. Debido a que los hilos pueden ser detenidos por el planificador del sistema operativo en cualquier momento, si la sección crítica se bloquea por demasiado tiempo y varios hilos están tratando de adquirir el *spinlock*, es posible que el planificador suspenda el hilo y lo active después de que haya sido liberado el candado, esto es lo mismo que ocurre con un mutex regular con la desventaja de que se han desperdiciado ciclos de CPU adicionales.

Barreras

Las barreras son un método de sincronización que detiene un grupo de hilos en un punto de la ejecución, y no los deja continuar hasta que todos los hilos del grupo han llegado a la barrera.

Variables condicionales

Se refiere a un tipo de sincronización parecida a los mutex en la cual un conjunto de hilos son asociados a una condición de paro en la cual deberán de esperar hasta que sea verdadera. Si un hilo llega a la variable condicional y esta es falsa, entrará en espera hasta que otro hilo le envíe una señal de que la condición ha cambiado su valor a verdadero.

Operaciones atómicas

En la programación concurrente, una operación es atómica si aparenta al resto del sistema como si hubiera ocurrido instantáneamente. La atomicidad garantiza el aislamiento de hilos concurrentes en operaciones críticas a un bajo nivel. Las operaciones atómicas generalmente integran un mecanismo contra fallos que verifica si la operación fue exitosa y el estado del sistema ha sido cambiado o si ha fallado y no se ha producido ningún cambio.

Los *multicore* tienen instrucciones especiales con la habilidad de inhibir las interrupciones del procesador temporalmente, asegurando que el hilo que las ejecute no pueda ser detenido. Estas instrucciones ejecutan operaciones básicas como el incremento o substracción de un valor (*Fetch-and-add* y *Fetch-and-sub*), o la comparación y escritura de un valor (*Compare-and-swap*).

2.3.3. Librerías

La programación paralela es un área de investigación interesante por los varios factores involucrados en su eficiencia. Pero es obvio que no es una tarea simple; para ser eficiente requiere de experiencia en la programación de paso de mensajes y de memoria compartida, al igual que conocimiento de la arquitectura paralela subyacente. Por esta razón se han desarrollado librerías de software, cuyo propósito es ocultar los aspectos de la programación paralela y del balance de carga de los usuarios finales, ofreciéndoles una interfaz relativamente más simple e intuitiva para desarrollar sus aplicaciones.

Skeletons

Los Skeletons son funciones preconstruidas de algoritmos paralelos típicos (pipes, divide&conquer, geometric decomposition). Programar con skeletons consiste en ensamblar las funciones preconstruidas necesarias para construir un programa paralelo.

Inicialmente los Skeletons fueron construidos a partir de lenguajes funcionales [12, 13, 14]. Esto complicó la programación ya que era necesario conocer el lenguaje

funcional y hacer los llamados a los Skeletons predefinidos. Actualmente los skeletons constituyen una biblioteca de funciones que pueden ser llamadas desde un lenguaje imperativo como C o Fortran.

El modelo de skeletons ha sido explorado ampliamente [12, 15, 16, 17], y hoy existe una gran diversidad de los mismos lo que complica la selección de el modelo adecuado para resolver un problema en especifico [18].

Existen varios ambientes de programación basados en el modelo de Skeletons. A continuación presentamos 3 de ellos: eSkel (**e**dinburgh**Ske**leton **l**ibrary) [19], SAMBA (Single Application Multiple Load Balancing) [20], y MALLBA (MAlaga, La Laguna, BArcelona) [21, 22].

eSkel define 5 Skeletons: Pipeline, Deal, Farm, Haloswap y Buterfly. De estos cinco sólo los dos primeros han sido implementados y liberados.

SAMBA [20] es un framework basado en skeletons para el desarrollo de aplicaciones SIMD con balance de carga. SAMBA es manejado dentro de un modelo de programación orientado a objetos (clases, métodos, etc). Para hacer un programa con SAMBA es necesario escribir los métodos dentro de las clases que se ofrecen como: Mediador, Balanceador-de-carga, Canal de transferencia, Repositorio y Tareas entre otros. Al ir escribiendo los métodos, el skeleton va tomando forma. Existen dos tipos de skeletons: los generales llamados skeletons-blancos y los finos llamados skeletons-negros. En los blancos, prácticamente se tiene que definir todo incluyendo la definición y el funcionamiento del programa. En los skeletons-negros casi todo esta definido y sólo hay que seleccionar las opciones preestablecidas.

MALLBA [21, 22], es otro ambiente de skeletons, dedicado principalmente a problemas de optimización. MALLBA ayuda a resolver los problemas mediante alguno de sus métodos de optimización como Branch&Bound, Algoritmos Genéticos, Algoritmos Evolutivos, Búsqueda Tabú o Recocido Simulado, entre otros. Su uso es similar a SAMBA, en el sentido que utiliza programación orientada a objetos donde es necesario definir ciertos métodos que describan la aplicación y escoger alguno de los métodos de optimización ofrecidos para resolver la aplicación.

Mapreduce

Mapreduce es un modelo de programación y un ambiente de ejecución de aplicaciones paralelas desarrollado por Google [7]. Fue diseñado para ser ejecutado en *clusters*, y para procesar enormes cantidades de información. Su propósito es ocultar los aspectos de paralelismo relacionados con particionamiento de datos y sincronización entre procesadores, balance de carga y tolerancia a fallas. Los usuarios de mapreduce solo tienen que definir pares de funciones secuenciales *map* y *reduce*.

Google usa mapreduce para : i) generar los datos de su servicio de búsqueda, ii) clasificar datos, iii) mineria de datos, iv) *machine learning*, y otras tareas. Hadoop es una versión libre y abierta de mapreduce desarrollada por Yahoo. Existen servicios web que ejecutan Hadoop en la nube, por ejemplo, con Amazon's Elastic Mapreduce [23].

La Figura 2.6 muestra el ambiente de ejecución mapreduce. La figura muestra un solo par de funciones map-reduce, indicada en la figura como "*Map phase*" y "*Reduce phase*", pero las aplicaciones mapreduce pueden tener muchos pares. El ambiente mapreduce replica las funciones *map* y *reduce* en tantas máquinas como lo indique el programador en un archivo de configuración, e inicia su ejecución simultánea de todas las copias, primero los *mappers* (*workers* en la "*Map phase*" de la figura), y luego los *reducers*.

Los datos de entrada deben ser copiados al sistemas de archivos paralelo y distribuido de Google. Como lo muestra la figura, un archivo en el sistema de archivos de Google está particionado entre varios discos para permitir la lectura paralela simultánea de datos distintos en diferentes discos y mejorar así el desempeño. Además, un archivo en el sistema de archivos de Google está replicado (no mostrado en la figura) para soportar tolerancia a fallas; existen varias copias de cada sector de datos de un archivo en distintos discos.

Así, distintos *mappers* pueden leer distintos datos de entrada simultáneamente. Nótese que cada *mapper* genera resultados intermedios que son almacenados en un archivo local en cada nodo. Una vez que todos los *mappers* terminan, el ambiente

mapreduce recolecta todos los datos intermedios, los ordena, los particiona de acuerdo a una *llave* especificada por el programador, y envía particiones completas a distintos *reducers*.

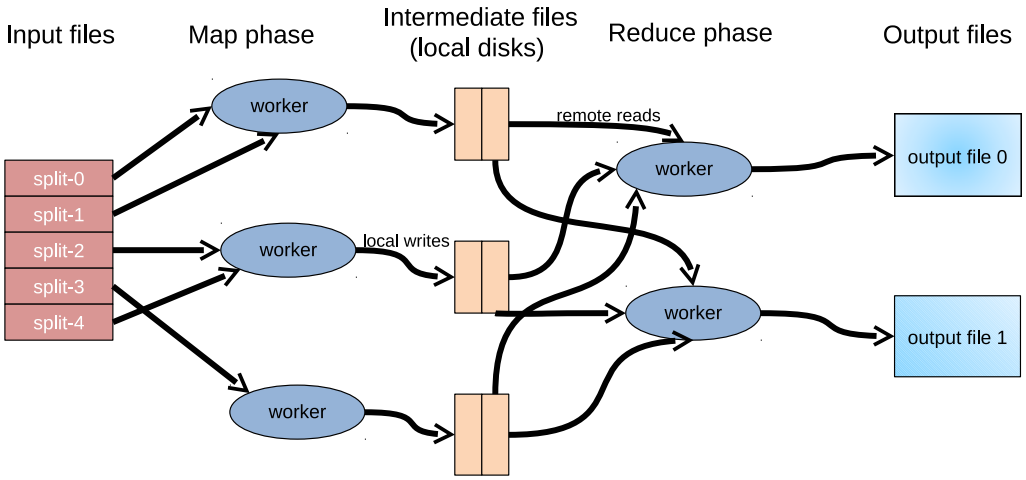

Figura 2.6: Ambiente de ejecución de aplicaciones Mapreduce

La salida de los *reducers* es almacenada de nuevo en el sistema de archivos paralelo distribuido de Google. De esta manera, si la aplicación consiste de 2 pares de funciones map-reduce, los mappers (replicados) del segundo par de funciones pueden acceder distintos datos simultáneamente.

Nótese que el procesamiento de mapreduce puede describirse brevemente como: procesamiento de listas de datos por *pipes* paralelos. Cada dato a procesar por cada *mapper* y cada *reducer* es un registro de un archivo. Cada mapper corresponde al primer paso/operación de un *pipe*, y cada *reducer* al segundo paso de un *pipe*. Un segundo *mapper* y un segundo *reducer* corresponderían al tercer y cuarto paso de un *pipe*. Debido a que los *mappers* y *reducers* son replicados cada uno con un *pipe* asignado, tendremos *pipes* paralelos.

2.3.4. Programación de *clusters* de nodos multinúcleo

En *clusters* de nodos *multicore*, la programación paralela típicamente utiliza comunicación de memoria compartida y de paso de mensajes. La comunicación de memoria

compartida es usada para la sincronización de paralelismo interno, entre los núcleos dentro de cada nodo *multicore*. Comunicación de paso de mensajes es usada para la sincronización de paralelismo externo, entre los distintos *multicores* del *cluster* [24].

Existen sistemas de software de memoria compartida distribuida que pueden ser usados para especificar comunicación de memoria compartida entre los nodos de un *cluster*, por ejemplo, Global Arrays[25], TreadMarks[26]. Sin embargo, estos sistemas construidos en base a paso de mensajes son generales, por lo que la especificación de paso de mensajes puede optimizarse para cada aplicación. También es posible utilizar comunicación de paso de mensajes entre los núcleos dentro de cada nodo. Pero la comunicación basada en memoria compartida es, en principio, más eficiente, ya que el paso de mensajes incurre el *overhead* de preparar, enviar y recibir cada mensaje.

Un elemento importante a considerar cuando se desarrolla software paralelo para *multicores* es la **"simpatía mecánica"** [27]. Se refiere a un entendimiento general del funcionamiento de la arquitectura de los *multicores* para poder programar aplicaciones con mayor desempeño. Como se vió anteriormente, las arquitecturas MIMD añaden una caché local entre cada núcleo y la memoria, esta caché es mucho más rápido de acceder para el procesador que la memoria compartida. Cuando una instrucción requiere acceder (lectura o escritura) a una dirección de memoria, esta se carga también en la caché. Además de la dirección requerida se cargará una línea completa de la memoria (en el caso del procesador Intel i7 920 es de 64 bytes) esperando precargar direcciones que pudieran ser utilizadas en un futuro cercano. Cuando se requiera acceder a una de estas direcciones precargadas, estas se encontrarán de manera más rápida porque ya están en la caché. A esto se le llama *localidad del caché*. De la mano con la localidad de la caché surge un problema: Cuando un núcleo cambia una dirección de memoria que está cargada en el caché de otro núcleo, se disparará un mecanismo de congruencia que invalidará el contenido de ambas cachés para esa dirección. Esto agrega un retardo a la operación de carga y es conocido como *fallo de caché*. Cuando múltiples núcleos intentan escribir a una misma dirección de memoria se provocará competencia en esa dirección y se generarán múltiples fallos

de caché, lo que disminuirá el desempeño de la aplicación. Para tener una mayor "simpatía mecánica" las aplicaciones deben de ser diseñadas procurando la localidad de la caché y evitando en lo posible los fallos de caché.

Capítulo 3

DLML: Data List Management Library

DLML es un *middleware* para procesar listas de datos en paralelo. Los usuarios DLML deben organizar sus datos en una lista, y usar funciones de DLML para insertarlos y obtenerlos de la lista para procesarlos. Típicamente una aplicación primero inserta todos los datos en una lista, y entonces ejecuta un ciclo en el que repetidamente obtiene un dato a la vez para procesarlo, hasta que ya no hay datos disponibles. Algunas aplicaciones pueden también generar datos dinámicamente, insertándolos en la lista dentro de ese ciclo.

```
1  void fill_process_datalist( int my_id ) {
2    DLML_dataitem item;
3    DLML_list *L = DLML_node_list( my_id );
4
5    if ( DLML_Iam_master_node( my_id ) ) {
6      insert_data_items();
7    }
8    while ( DLML_get( L, &item ))
9      process_data_item( &item );
10 }
11
12 int main(int argc, char *argv[]) {
13   DLML_init( argc, argv );
14   fill_process_datalist( DLML_myid() );
15   DLML_finalise();
16   return 0;
17 }
```

Figura 3.1: Estructura general de una aplicación DLML

Se utiliza un modelo de memoria compartida "virtual". Bajo este modelo el programador inserta un nuevo elemento en una lista prácticamente infinita, sin preocuparse del orden de inserción, eventualmente algún procesador del *cluster* procesará el elemento. Cuando se toma un elemento de la lista para ser procesado, no importa de que parte del *cluster* procede. Esto solamente se puede realizar sin los elementos son independientes entre si. La librería administra esta memoria compartida "virtual" procurando que todos los procesadores obtengan elementos para procesar, y balancea la carga de trabajo para utilizar eficientemente todos los recursos.

El uso de DLML es simple conceptualmente. La Figura 3.1 muestra la estructura general de una aplicación DLML. En el procedimiento `main()`, la aplicación primero llama `DLML_init()` para inicializar MPI, listas DLML, y variables globales DLML en cada procesador del *cluster* que ejecutará la aplicación. La aplicación llena entonces la lista de datos a procesar (ver el `fill_process_datalist()` en la figura). Típicamente este paso es realizado por el procesador maestro. Finalmente la aplicación entra en un ciclo para obtener un dato y procesarlo hasta que `DLML_get()` no regrese ningún dato, lo que significa que ya no hay datos disponibles en el todo el *cluster*.

DLML maneja una lista por cada proceso de aplicación en cada procesador. Cuando una lista se vacía, DLML trata de rellenarla robando datos de otra lista de manera transparente al usuario. Soló cuando `DLML_get()` no regresa un dato, la aplicación sabe que no hay más datos y puede terminar. La interfaz DLML oculta de los programadores la comunicación para sincronizar el robo de datos entre listas. Este mismo robo tiende a balancear la carga de trabajo de acuerdo a la capacidad de procesamiento de cada procesador.

La primera versión de DLML [8, 9] fue diseñada para *clusters* compuestos de nodos con un solo procesador o CPU, y basada en paralelismo de múltiples procesos y en paso de mensaje entre los mismos con MPI (Message Passing Interface) [10]. En esta versión, en cada procesador (con un solo núcleo), un proceso de aplicación ejecuta el código de la aplicación, insertando, obteniendo y procesando datos, y un proceso DLML está a cargo de: i) *hacer* peticiones de datos a nodos remotos cuando la lista

local se vacía, y ii) *servir* peticiones de datos de nodos remotos. Ambas tareas siguen un protocolo basado en paso de mensajes. El paso de mensajes es también usado entre un proceso de aplicación y su proceso DLML *hermano* (en el mismo nodo) para mover elementos de datos entre sus espacios de direcciones.

```
1  DLML_process(int *shm) {
2  while ( 1 )
3  MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG,
     ...,&stat);
     // SAP: sibling application process
4  switch (stat.MPI_TAG) {

5     case DATA_SIZE_REQUEST:
6       send local list size
7       break

8     case DATA_REQUEST:
9       enqueue data request
10       ask SAP the local list
11         *shm = 1
12       break

13    case LOCAL_LIST_SIZE:
14      receive local list
15      D = list_size/(requests_nr +1)
16       // +1 counts in this node
17      for each data request:
18       send D tagged REMOTE_DATA_SIZE
19       send D items tagged REMOTE_DATA
20      send SAP remainder of local list:
21       first size and then items
22      *shm = 0
23      break

24    case LOCAL_DATA_REQUEST:
25      send DATA_SIZE_REQUEST to all nodes.
26      break

27    case DATA_SIZE_RESPONSE:
28      store response
29      if responses_count < NODES -1
30        break
31      if all responses == 0
32        send all procss. ITS_OVER; exit(0);
33        choose node with most data
34        send that node DATA_REQUEST
35        break

36    case REMOTE_DATA_SIZE:
37       receive number of items D to receive
38       if D == 0      // try again
39         send DATA_SIZE_REQUEST to
40         break
41       else
42         receive D items tagged REMOTE_DATA
43         send SAP the D items
44         break

45    case ITS_OVER:
46         exit( 0 )
47       }
48 }
```

Figura 3.2: Pseudo-código del proceso DLML (PD).

La Figura 3.2 muestra el proceso DLML (PD) en pseudo-código. PD continuamente llama `MPI_Probe()`, bloqueándose hasta que un mensaje llega de cualquier fuente y con cualquier etiqueta, lineas 2-3 en la figura. (`MPI_Iprobe()` es no bloqueante; pero su uso no fue considerado adecuado en este contexto). El código de las líneas 5-23 corresponde a servir peticiones de datos provenientes de nodos remotos, mientras que el código de las líneas 24-44 corresponde a hacer peticiones de datos a nodos remotos.

La mayor parte del tiempo, la lista (local) de datos en cada nodo está siendo procesada por el proceso de aplicación (PA). Así, cuando PD recibe una petición, líneas 8-12, primero debe pedir la lista local a su PA hermano (ejecutándose en el

mismo nodo). Esto lo hace prendiendo la bandera `*shm`; esta bandera es verificada por cada PA cada vez que llama `DLML_get()`. PD recibe la lista local del PA *hermano* en dos pasos (líneas 13-23): primero un mensaje con el tamaño de la lista etiquetado `LOCAL_LIST_SIZE`, y luego los elementos de lista a través de una serie de mensajes. PD divide la lista entre el número de peticiones recibido, mas 1 para tomar en cuenta a su propio nodo. El resultado de esta división, D, es el número de datos de lista que se enviarán en respuesta a cada petición; primero se envía el tamaño D, seguido de D datos (ver Figura 3.2 lineas 18 y 19). Finalmente, PD apaga la bandera `*shm`.

Cuando en un nodo se vacía una lista de un PA, el PD hermano en ese nodo recibe una `LOCAL_DATA_REQUEST`, y el PD inicia un protocolo de subasta: pidiendo primero el tamaño de las listas en todos los nodos remotos (líneas 24-26 y 4-7), seleccionando el nodo N con la lista más grande, y enviando una petición de datos al nodo N (líneas 27-35). Si el tamaño de todas las listas remotas (`DATA_SIZE_RESPONSE`) es 0, esto indica el fin de la aplicación, PD envía el mensaje de que la ejecución ha terminado (`ITS_OVER`) a todos los PDs (líneas 31-32) y que a su vez lo propagarán a sus PAs y para que no devuelvan elementos en sus llamadas a la función `DLML_get`, terminando la ejecución.

La respuesta a una petición de datos (`DATA_REQUEST`) (línea 34) será `REMOTE_DATA_SIZE` (líneas 36-44), cuyo procesamiento puede ser seguido en la figura.

DLML puede ser visualizado como un grupo de trabajadores en un línea de producción de una fabrica, los trabajadores estarán organizados en pares PA y PD. Los PA son los encargados de consumir y producir el trabajo un elemento a la vez. Los procesos PD son los encargados de administrar el trabajo (organizados en listas de datos). Los PD son los encargados de balancear el trabajo robando de otros PD una parte cuando lo necesiten. Además serán los encargados de señalar a los PA cuando el trabajo se termine. Bajo está analogía el programador solamente diseñará el comportamiento de un trabajador PA. La librería DLML será el encargado de replicar el diseño de PA tantas veces sea necesario y de asignarles automáticamente PDs hermanos para que administren sus listas y balanceen la carga de trabajo.

Capítulo 4

MC-DLML: MutiCore DLML

En este capítulo se presenta un análisis de posibles enfoques para el diseño de una nueva versión de DLML que pueda aprovechar el paralelismo implícito de los nuevos *clusters multicore*. Posteriormente se presentan nuestros diseños para MultiCore Data List Manager Library, MC-DLML basados en múltiples hilos, comunicación a través de memoria compartida y paso de mensajes.

4.1. Aspectos de Diseño de DLML para Clusters Multicore

DLML fue desarrollado para *clusters* con procesadores con un solo núcleo, como se mencionó en el capítulo anterior. Debido a la extendida utilización de los *clusters multicore* se desea reutilizar el modelo de memoria "virtual" en el *cluster* pero tomando ventaja de la verdadera memoria compartida para los núcleos de cada *multicore* del *cluster*. Las aplicaciones desarrolladas con DLML pueden ser ejecutadas en *clusters* con procesadores *multicore* por medio de ejecutar un par de procesos hermanos por cada núcleo del procesador. Por ejemplo, para un procesador con dos núcleos se ejecutarán dos procesos aplicación (PA) y dos procesos DLML (PD) para balancear las carga interna y externamente. En general se necesitan $2N$ procesos para un procesador con N núcleos. Pero esto genera una sobrecarga de mensajes MPI, lo cual no es

óptimo y no toma ventaja de la memoria compartida entre los procesos que se ejecutan en un mismo nodo. En principio la comunicación basada en memoria compartida entre los núcleos de un procesador *multicore* es más eficiente que el paso de mensajes, ya que para enviar un dato por medio de mensajes se necesita copiar el dato de la lista a un mensaje, enviar el mensaje, recibirlo y copiarlo a la nueva lista. Cada uno de estos pasos tiene un retraso que puede ser evitado utilizando la memoria compartida.

En otras palabras, se puede optimizar el balance de carga interno entre núcleos dentro del mismo *multicore* creando aplicaciones híbridas MPI + pthreads [24] utilizando varios hilos de aplicación y un hilo DLML (Hilo Balanceador de ahora en adelante) dentro de un solo proceso. De esta manera los datos en listas son visibles a todos los hilos de aplicación y al hilo balanceador encargado del balance inter- nodo.

Entonces surge un nuevo reto: la sincronización de los hilos para balancear la carga intra-nodo. Existen dos métodos básicos para el balance de carga, la distribución de trabajo (*work sharing*) y el robo de trabajo (*work stealing*). La distribución de trabajo reparte el trabajo disponible a otros hilos que puedan necesitarlo en un futuro, inclusive si no han hecho una petición de datos. Esta manera de balancear el trabajo es conveniente en la etapa inicial de la ejecución, cuando se debe de dividir el trabajo y cuando se está trabajando con aplicaciones estáticas regulares.

El método de robo de trabajo fue propuesto originalmente por Arora *et al* [28] para aplicaciones paralelas irregulares donde se genera nuevo trabajo dinámicamente. Este trabajo fue presentado utilizando *listas de hilos de ejecución* que deben de ser ejecutados. En cambio DLML utiliza *listas de datos*, que utilizan los mismos hilos creados desde un principio para no generar cambios de contexto adicionales a los ordenados por el planificador del sistema operativo.

El algoritmo de robo de trabajo es el siguiente. Cuando uno de los hilos o procesos tiene una lista vacía, se convierte en "ladrón" y deberá escoger a una "víctima" a la cual robará una porción de trabajo de su lista. Una vez que la operación de robo es completada, el ladrón se "reformará" y volverá a su operación normal.

El robo de trabajo ha sido ampliamente estudiado [29, 30, 31, 32]. Se ha demos-

trado que es más eficiente que la distribución de trabajo, tanto en espacio como en la reducción de la sincronización, debido a que cuando todos los hilos tienen trabajo no se hacen intentos de migrar datos entre ellos [31]. Inclusive se ha demostrado su estabilidad y alta adaptabilidad en sistemas heterogéneos multiprogramados como a los que se enfoca esta tesis [29, 32].

Un factor importante para decidir el mecanismo específico de balance de carga entre los hilos, es la organización de las listas de trabajo dentro de cada nodo *multicore*. Existen distintas opciones como: el uso de una lista global por nodo, la división de los datos en sublistas locales, una por cada núcleo y una organización híbrida que combine las dos anteriores, con una lista global y listas locales.

En el resto de esta sección se describirán cada una de las organizaciones de listas antes mencionadas y sus ventajas. Aunque puedan existir otras organizaciones más complejas, éstas fueron seleccionadas para demostrar la amplitud del espacio de diseño y las implicaciones de la selección de una organización para la lista de datos, con repercusiones en el diseño de los mecanismos de carga externo e interno.

En el resto de esta sección se utilizará el término "lista de datos" indistintamente para referirnos a la estructura donde serán almacenados los datos en la memoria compartida sin implicar que tendrán un orden o serán accedidos de manera específica. Esto nos dará libertad de delegar los detalles de la implementación subyacente a la lista de datos a la siguiente sección.

4.1.1. Una lista global

La organización más simple dentro de cada nodo para los datos es ordenarlos en una sola lista. Al tener todos los elementos almacenados en un solo lugar, el control se centraliza y es más sencillo de diseñar.

Cada vez que un hilo de aplicación (**HA**) requiera insertar nuevos datos sólo requerirá de una variable compartida de acceso a la lista (*mutex*, *splinlock*, *compare-and-set*). De la misma manera, para la extracción de los elementos, en una organización con una sola lista no es necesaria la implementación de un mecanismo de balanceo de

carga interna lo que simplifica la implementación. No obstante, el *overhead* causado por la competencia por la memoria en la variable de sincronización puede ser excesivo como se discute en [27].

Acoplar el balance de carga inter-nodo a esta organización es sencillo debido a que con una sola variable de sincronización se bloquea el acceso a la lista de los HA, permitiendo a un hilo balanceador (**HB**) enviar datos a otro nodo y agregar a la lista nuevos datos recibidos. Aunque esto tampoco es óptimo, puesto que al detener todos los HAs para enviar datos a otro nodo se hace más lento el procesamiento de los datos y el desempeño en general. El mayor inconveniente de una sola lista global es que se desea evitar los retrasos en el acceso a los datos, para reducir el tiempo de ejecución, siempre que hayan datos en la lista, porque los HA deberán de poder acceder a ellos lo más rápidamente posible para continuar con su procesamiento, pero esto se retrasa debido a que el HB los bloqueará para cada envío y recepción de datos externos.

4.1.2. Listas locales

Podemos dividir la carga de trabajo de cada nodo en sublistas para cada HA lo que es conveniente en principio porque evita la competencia por la memoria, ya que cada uno insertará y extraerá datos sin la necesidad de sincronizarse con los demás. El reto principal de esta organización será cuando se deba de extraer datos pero la lista local esté vacía y aun existan más datos en el nodo. Tendremos entonces la necesidad de balancear la carga de trabajo.

Para balancear la carga de trabajo cuando una lista esté vacía se roba carga de otra lista. En esta fase de robo se avisará al HA propietario de la lista que se están retirando datos para no afectar la integridad de la lista. Dado que el tamaño de la lista varía constantemente, la selección de la posible víctima dependerá de una "política de robo". Se puede pensar que el seleccionar como víctima a la lista con más trabajo nos permite robar una mayor cantidad de datos y tiende a reducir robos posteriores. Sin embargo es posible que el HA propietario esté ocupando su lista local u otro HA ya esté robando datos de ella por lo que será más rápido escoger otra víctima. Se

pueden seleccionar otras políticas de robo como la selección de una víctima aleatoria o quizá una elección basada en un algoritmo determinista como *round robin*.

Por otro lado, cuando todas las listas estén vacías se avisará al HB que debe enviar una petición de datos a otro nodo, y cuando reciba los datos deberá repartirlos entre las listas locales. Cuando un HB recibe una petición de datos de otro nodo, avisará a los HA que el acceso a su lista está bloqueado para poder extraer de ellas los datos requeridos. Es importante señalar que esta es una política de robo distinta al robo local puesto que se desea dar prioridad al procesamiento de los datos internamente en el nodo, porque al enviarlos a ser procesados a otro nodo requiere de un tiempo extra y no se desea detener más de lo necesario el tiempo de acceso a los datos como se mencionó anteriormente. La decisión de la cantidad de datos a enviar y el número de listas locales que serán bloqueadas puede ser adaptable al tiempo de procesamiento de cada dato o tomar valores estáticos.

Una pequeña variante a esta organización es la utilización de dos listas locales por cada HA. Utilizar una lista de datos para la inserción y una para extracción de datos tiene repercusiones interesantes en el desempeño teórico del sistema en general. Cuando un HA esté ocupando una de las dos, el ladrón utilizará la lista libre para efectuar el robo sin necesidad de detener la operación normal de la víctima. De la misma manera el HB podría extraer datos del nodo sin necesidad de detener a ningún HA.

4.1.3. Organización híbrida

Por último mencionaremos la organización híbrida de las listas que combina las ventajas de las dos organizaciones previamente descritas: la lista global y las listas locales. Una organización híbrida reduce la competencia por la memoria para el acceso a los datos por parte de ambos tipos de hilo. Los HA tendrán acceso a su lista local sin necesidad de sincronización y el HB podrá enviar datos a otros nodos bloqueando solamente la lista global.

El balanceo de carga dentro de esta configuración híbrida es como sigue. Para

balancear la carga interna del *multicore* cuando un HA se queda sin datos en su lista local accede a la lista global y toma una porción de los datos, varios a la vez, y los almacena en la lista local sin necesidad de interrumpir a ningún otro HA que se encuentre con datos. Cuando la lista global está vacía, el HA toma datos de las listas locales. Esto requiere del bloqueo de dichas listas y el movimiento de los datos de un lugar de la memoria compartida a otro. Esta es una operación más lenta que el simple robo entre dos listas locales.

Cuando se requiere enviar datos a otro nodo para balancear la carga global del *cluster* el HB accederá a la lista global para enviar una fracción de los datos. Cuando todas las listas locales estén vacías y la lista global también lo este, se avisará al HB para que envíe una petición de datos externa.

Aunque las operaciones de sincronización sobre variables compartidas tiene un efecto en el desempeño de la aplicación en general, también se debe de tomar en cuenta el costo en tiempo que toma una operación de balance de carga. Una organización híbrida aumenta el número de operaciones en cada balance y la complejidad de estas, debido a que se deben de migrar los datos más de una vez entre las diferentes listas.

4.2. Implementación de MC-DLML

Despues de analizar el espacio de diseño, decidimos usar una estructura de listas ligadas consumidas como pilas organizadas en listas locales de datos, una por cada hilo de aplicación. Esta decisión surge después del análisis y la experimentación con distintas organizaciones. Se intentó en primer lugar con dos listas locales para cada HA y una lista global por cada nodo, sin embargo, esta versión tenía un pobre desempeño debido al incremento del tiempo de las operaciones de robo y la complejidad de éstas. También se intentó con una sola lista global por nodo, de manera similar a la descrita en [27]. Esta lista global de datos fue implementada en un arreglo circular para beneficiarse de la localidad de la caché y obtener una mayor "simpatía mecánica" como se mencionó en la sección 2.3.4. Sin embargo, esta organización de lista funciona

mejor cuando se tienen pocos productores que inserten datos y muchos lectores concurrentes a los mismos elementos. Nuestro caso para MC-DLML es distinto, debido a que tenemos igual número de productores y consumidores de trabajo. Además cada dato consumido debe de ser eliminado de la lista puesto que no será consumido por ningún otro. Esto hace que una implementación con una sola lista global tenga una alta competencia por la memoria y un alto número de fallos de caché, lo que provoca un pobre desempeño.

La organización final implementa listas locales, debido a que reduce la competencia por la memoria en las variables compartidas, ya que en una operación normal, mientras todos los HA tienen datos, pueden acceder a su lista local sin necesidad de sincronización entre ellos.

Otra razón importante para utilizar listas locales es la preservación de la localidad de la caché. Cuando un HA produce y consume datos de su lista local se aumenta la probabilidad de que los datos aun estén en la caché, lo que reduce el tiempo de carga y aumenta el desempeño.

Debido a que los elementos de la lista de datos son insertados en la lista en un orden temporal, la lista puede ser accedida de dos maneras: como una cola FIFO (*First in, First Out*) y como una pila LIFO (*Last in, First Out*). Las aplicaciones paralelas desarrolladas con MC-DLML pueden ser irregulares y generar dinámicamente nuevos elementos, que a su vez generen nuevos elementos. Debido a esto, un acceso de pila acumula menos elementos en la lista de datos porque recorre los elementos primero en profundidad [33]. Por esto se implementa un acceso de pila para nuestras listas locales.

Las listas locales necesitan de un método de balanceo de carga tanto interno dentro del *multicore* como global en todo el *cluster*. Un solo hilo balanceador HB por nodo es suficiente para realizar los movimientos de migración de datos para el balanceo de carga en el *cluster*, muy similarmente al descrito en el capítulo 3. Esta decisión responde a dos factores importantes: mantener el protocolo de comunicación externo simple y utilizar el nivel *funneled* de seguridad de hilos el cual produce menor

overhead de sincronización que los niveles *serialiazed* y *multiple* como se menciona en la sección 2.3.1.

En esta sección se presentan cuatro algoritmos distintos de balanceo de carga interno basados en robo de trabajo, detallando su funcionamiento y ventajas. Se describirá a detalle el primero y más sencillo, el bloqueo básico (BL), los demás serán descritos basándose en este.

4.2.1. Bloqueo básico (BL)

El algoritmo más simple de balanceo de carga intra-nodo es asignar un *spinlock* a cada lista local. Lo denominamos bloqueo básico o BL por sus siglas en inglés (Basic-Locking). Cuando un HA inserta o extrae un elemento de la lista la bloquea con el *spinlock* como se muestra en la Figuras 4.1 y 4.2 respectivamente. Si intenta extraer un elemento pero la lista está vacía, sale del modo de operación normal (Figura 4.2: líneas 2-9) y entra en modo de robo local (líneas 10-45). Primero se incrementará atómicamente un contador global de listas vacías `emptylists` (línea 13), después se bloquea un candado mutex global (línea 21) y escoge como víctima a la lista con más elementos (línea 24). Se bloqueará la lista víctima con el *spinlock* y se tomará la mitad de sus elementos(línea 26-33). Si la operación es exitosa se regresa al modo de operación normal, si no, se mantiene iterando en el modo de robo local mientras existan datos en el procesador (líneas 34-44). Cuando todos los HA entran en modo de robo local y han incrementado la variable global `emptylists` se pasa al modo de robo externo(líneas 47-59), y el HB inicia el protocolo de subasta para robar trabajo de otro *multicore*.

```
1 void DLML_insert(DLML_list *L, DLML_dataitem *item){
2    /*+ normal operation +*/
3    spin_lock(&L->lock)
4    insert data into L
5    spin_unlock(&L->lock)
6    /*+ end of normal operation +*/
7 }
```

Figura 4.1: Inserción en basic-locking (BL)

```
1  int DLML_get(DLML_list *L, DLML_dataitem
       *item) {
2    spin_lock(&L->lock)
3    if (L->size > 0) {
4     /*+ normal operation +*/
5     *item = top element of L
6     spin_unlock(&L->lock);
7     return 1; // found data
8     /*+ end of normal operation +*/
9    }
10   else {
11 // no data in this thread's list, at all
12   /*- local steal operation -*/
13    atomic_increment (dlml_emptylists)
14     __TRY_AGAIN__:
15    if (dlml_is_finished) {
16     spin_unlock(&L->lock)
17     return 0;
18    }
19    spin_unlock(&L->lock)
20    //Block global mutex lock
21    mutex_lock(refilllock)
22    spin_lock(&L->lock)
23    // 'this' is the index of the largest
          list
24    int this = choose the largest list
25    //LL is the array of local lists
26    if (this >= 0 && LL[this]->size > 1) {
27     spin_lock(&LL[this]->lock)
28     move LL[this]->size / 2 items to L
29     atomic_decrement (emptylists)
30     spin_unlock(&L->lock)
31     spin_unlock(&tl[this]->lock);
32     mutex_unlock(refilllock)
33     return DLML_get(L, item) //now it has
           data
34    } else if (emptylists <
          DLML_APP_THREADS_NR) {
35    //there are still data in the node
36     if (L->size > 0) {
37      *item = top element of L
38      spin_unlock(&L->lock)
39      mutex_unlock(refilllock);
40      return 1; // found data
41     }
42     mutex_unlock(refilllock)
43     yield()
44     goto __TRY_AGAIN__ //try to steal
           again
45    }
46  /*- end of local steal operation -*/
47    else {
48  /** external steal operation **/
49     if (!external_steal_request)
50       atomic_increment (
             external_steal_request)
51     spin_unlock(&L->lock)
52     while (!dlml_is_finished &&
           external_steal_request)
53       cond_wait(&
             dlml_finished_or_refillingon_cond
             , refilllock);
54     mutex_unlock(refilllock)
55     if (dlml_is_finished) { // game is
           over
56      return 0
57     }
58  /** end of external steal operation **/
59     return DLML_get(L, item)
60    }
61  }
62 }
```

Figura 4.2: Extracción en basic-locking (BL)

Cuando el HB recibe una petición de datos toma el mutex global, roba de la lista con más datos y los envía al HB ladrón que envió la petición. El HB ladrón reparte los datos entre las listas locales y avisa a los HA para que regresen a su operación normal. Si ninguno de los nodos tiene datos la ejecución termina. El HB se encuentra iterando continuamente hasta que se termine la ejecución, entre hacer peticiones para el robo de datos (Figura 4.3) y responder a las peticiones externas (Figura 4.4). Como se vio en la sección 2.3.2 el costo de los *spinlocks*, aunque es menor que los mutex es aun alto, además de que, si solamente se toman datos de una de las listas locales se realiza una operación de robo externo más rápida, pero menos efectiva para el balance global del *cluster*. Además, cada vez que se realice una operación de robo externo se provoca un desbalance interno en el nodo víctima.

```
1 DLML_make_request() {
2  send all other nodes DATA_SIZE_REQUEST
3  while( 1 )
4    MPI_Probe( ..., &stat ) ;
5    switch (stat.MPI_TAG)
6     case DATA_SIZE_RESPONSE:
7      choose largest list,
8      send DATA_REQUEST to "largest-list
    " node
9      break

10    case REMOTE_DATA_SIZE:
11     if D == 0
        // TRY AGAIN
12     send all other nodes
    DATA_SIZE_REQUEST
13     break
14     else
15      receive data
16      refill lists
17     return

18    case ITS_OVER:
19     dlml_finished = 1
20     return

21    case DATA_SIZE_REQUEST
        //Do nothing

22    case DATA_REQUEST
23     send 0 tagged DATA_SIZE_RESPONSE
24     break
25 }
```

Figura 4.3: Envío de peticiones externas

```
1 DLML_serve_request() {
2  MPI_Iprobe(ANY, ..., &flag, &stat )
3   if ( !flag )
4    return

5   switch (stat.MPI_TAG) {
6    case DATA_SIZE_REQUEST:
7     send size of all lists
8     return
9    case DATA_REQUEST:
10     lock all lists as in DLML_get()
11     serve data requests
12     unlock lists
13     return

14   case ITS_OVER:
15     dlml_finished = 1
16     return
17 }
```

Figura 4.4: Respuesta de peticiones externas

4.2.2. Bloqueo global (GL)

El algoritmo de bloqueo global [34], GL (Global-locking), es más complejo que BL, porque no recurre a la simplicidad de los *spinlocks* por cada lista, sino que utiliza una variable compartida de sincronización. En las Figuras 4.5 y 4.6 (líneas 7-10) se muestra que ni la inserción ni la extracción de un elemento requieren de bloqueo. Al eliminar los *spinlocks* se aumenta la fluidez del modo de operación normal, el cual ocupa la mayor parte del tiempo de ejecución. No obstante, una operación de robo local cuando una lista está vacía requiere bloquear todas las listas. Para hacer un robo:

I El HA ladrón incrementa `emptylists` (línea 16).

II Activa una bandera de paro (`stopthreads`) para que otros HA dejen de usar sus listas(línea 24).

III El HA ladrón selecciona la lista más grande y roba la mitad de sus datos(línea 26). Debido a que el hilo víctima se encuentra detenido, al inicio de la inserción o la extracción (líneas 2-6), es seguro tomar sus datos.

IV El resto del algoritmo es de manera similar a BL.

El bloqueo global obtiene información más reciente respecto al tamaño de las posibles víctimas que BL, debido a que cuando se pregunta el tamaño de la lista, se encuentra bloqueada y su tamaño no cambia. BL en cambio, escoge a su víctima basándose en un tamaño que puede que cambie cuando por fin se bloquea su *spinlock* para robar los datos.

La mayor ventaja de GL, con respecto a BL, es que facilita el bloqueo de varias listas con una sola operación atómica en `stopthreads`. Esto permite hacer operaciones de robos externos utilizando más de una lista a la vez, permitiendo migrar un mayor número de elementos. Esto tiende a reducir el número de robos externos. Otra ventaja es que, al quitar la mitad de los elementos de todas las listas del nodo en la operación de robo, la carga de trabajo local de la víctima no se desbalancea como es el caso de BL.

```
1  void DLML_insert(DLML_list *L, DLML_dataitem *item){
2     if (stopthreads > 0){
3         //wait if a refill is going on
4         copy = refillstamp
5         wait while copy == refillstamp
6     }
7     /*+ normal operation +*/
8     insert data into L
9     /*+ end of normal operation +*/
10 }
```

Figura 4.5: Inserción en global-locking (GL)

4.2.3. Bloqueo de baja sincronización (LSL)

Una de las principales desventajas de GL es la necesidad de que los HAs lleguen a la condición de paro para notificar que no están haciendo uso de su listas locales para ser consideradas como posibles víctimas. Esto deja espacio para la optimización porque

```
1  int DLML_get(DLML_list *L, DLML_dataitem
       *item) {
2     if (stopthreads > 0){
3        //wait if a refill is going on
4        copy = refillstamp
5        wait while copy == refillstamp
6     }
7     if (L->size > 0) {
8        /*+ normal operation +*/
9        *item = top element of L
10       return 1; // found data
11       /*+ end of normal operation +*/
12    }
13   else {
14       // no data in this thread's list,
             at all
15       /*- local steal operation -*/
16       atomic_increment (dlml_emptylists)
17       ___TRY_AGAIN___:
18       if (dlml_is_finished) {
19          return 0
20       }
21
22       //Block global mutex lock
23       mutex_lock(refilllock)
24       stopthreads = 1
25       // 'this' is the index of the
             largest list
26       int this = choose the largest list
             with copy == refillstamp
27       //LL is the array of local lists
28       if (this >= 0 && LL[this]->size >
             1) {
29          move LL[this]->size / 2 items to
                L
30          stopthreads = 0
31          refillstamp++
32          atomic_decrement (emptylists)
33          mutex_unlock(refilllock)
34          return DLML_get(L, item) //now
                it has data
35       } else if (emptylists <
             DLML_APP_THREADS_NR) {
36          stoppedthreads = 0
37          refillstamp++
38          //there are still data in the
                node
39          if (L->size > 0) {
40            *item = top element of L
41             mutex_unlock(refilllock)
42             return 1 // found data
43          }
44          mutex_unlock(refilllock)
45          yield()
46          goto ___TRY_AGAIN___; //try to
                steal again
47       }
48       /*- end of local steal operation -
             */
49       else {
50          /** external steal operation **/
51          if (!external_steal_request)
52             atomic_increment (
                   external_steal_request)
53          stoppedthreads = 0
54          refillstamp++
55          while (!dlml_is_finished &&
                external_steal_request)
56             cond_wait(&
                   dlml_finished_or_refillingon_cond
                   , refilllock)
57          mutex_unlock(refilllock)
58          if (dlml_is_finished) { // game
                is over
59             return 0;
60          }
61          /** end of external steal
                operation **/
62          return DLML_get(L, item)
63       }
64    }
65 }
```

Figura 4.6: Extracción en global-locking (GL)

cuando un HA se encuentra procesado un dato tampoco se encuentra haciendo uso de su lista. Tenemos entonces dos fases distinguibles de la ejecución de un HA: dentro de *DLML* cuando está insertando o extrayendo un elemento de la lista y fuera de *DLML* cuando se encuentra procesando un elemento.

El bloqueo de baja sincronización[35], LSL (Low-Sync-Locking), aumenta el desempeño del balance de carga interno al marcar la entrada y la salida de las fases con una operación atómica. Cuando un HA inicia la inserción de un elemento nuevo a su lista local activa atómicamente la bandera `inside_dlml` y cuando termina la desactiva también atómicamente como se muestra en la Figura 4.7 (líneas 8-12).

Recordemos que las operaciones atómicas son menos costosas que los *spinlocks*.

Cuando un HA intenta extraer un elemento de su lista local, si la bandera global `stopthreads` está encendida, espera a que termine la operación de robo actual y procede a activar atómicamente `inside_dlml` (Figura 4.8 líneas 2-7). Si la lista local tiene datos, toma el primero, desactiva a `inside_dlml` y sale de la función `DLML_get` (líneas 8-14). Si la lista está vacía entra en modo de robo local muy similar al descrito para GL, pero esta vez, además incluye a los HAs que se encuentren detenidos por la bandera de paro `stopthreads`. También incluye como posibles víctimas a los que tengan desactivado `inside_dlml`, porque significa que no se encuentran haciendo uso de su lista local (líneas 28 y 29). Si bien hay un pequeño sacrificio en velocidad al indicar la entrada y salida de DLML con una operación atómica, esto se justifica con la reducción del tiempo de espera del HA ladrón para efectuar una operación de robo. Una desventaja de LSL es que, es probable que no se elija a la lista con mayor número de elementos como víctima del robo local.

Para hacer una operación de robo externo el HB seguirá un proceso similar al robo local, marcando las listas que serán utilizadas para la migración de datos, tomando en cuenta las que se encuentran detenidas por la bandera `stopthreads`, y las que tienen desactivada la bandera `inside_dlml` (líneas 54-59). Despues de realizar una operación de robo externo se genera un desbalance, el cual es mayor que GL porque sólo se seleccionará a un subconjunto de las listas.

```
1  void DLML_insert(DLML_list *L, DLML_dataitem *item) // insert first
2  {
3     if (stopthreads > 0){
4     //wait if a refill is going on
5         copy = refillstamp
6         wait while copy == refillstamp
7     }
8     atomic_increment (L->inside_dlml)
9     /*+ normal operation +*/
10    insert data into L
11    /*+ end of normal operation +*/
12    atomic_decrement (L->inside_dlml)
13 }
```

Figura 4.7: Inserción en low-sync-locking (LSL)

```
1  int DLML_get(DLML_list *L, DLML_dataitem
       *item) {
2    if (stopthreads > 0){
3    //wait if a refill is going on
4        copy = refillstamp
5        wait while copy == refillstamp
6    }
7    atomic_increment (L->inside_dlml)
8    if (L->size > 0) {
9       /*+ normal operation +*/
10      *item = top element of L
11      atomic_decrement (L->inside_dlml)
12      return 1 // found data
13      /*+ end of normal operation +*/
14   }
15  else {
16      // no data in this thread's list,
            at all
17      /*- local steal operation -*/
18      atomic_increment (dlml_emptylists)
19      ___TRY_AGAIN___:
20      if (dlml_is_finished) {
21         atomic_decrement (L->inside_dlml
               )
22         return 0
23      }
24      //Block global mutex lock
25      mutex_lock(refilllock)
26      stopthreads = 1
27      // 'this' is the index of the
            largest list
28      int this = choose the largest list
            if:
29           copy == refillstamp || !L->
                inside_dlml
30      //LL is the array of local lists
31      if (this >= 0 && LL[this]->size >
            1) {
32         move LL[this]->size / 2 items to
               L
33         stopthreads = 0
34         refillstamp++
35         atomic_decrement (emptylists)
36         mutex_unlock(refilllock)
37         atomic_decrement (L->inside_dlml
               )
38         return DLML_get(L, item); //now
               it has data
39      } else if (emptylists <
          DLML_APP_THREADS_NR) {
40         stoppedthreads = 0
41         refillstamp++
42         atomic_decrement (L->inside_dlml
               )
43         //there are still data in the
               node
44         if (L->size > 0) {
45           *item = top element of L
46           mutex_unlock(refilllock)
47           return 1; // found data
48         }
49         mutex_unlock(refilllock)
50         yield();
51         goto ___TRY_AGAIN___ //try to
               steal again
52      }
53      /*- end of local steal operation -
            */
54      else {
55         /** external steal operation **/
56         ...
57         /** end of external steal
               operation **/
58         return DLML_get(L, item)
59      }
60   }
61 }
```

Figura 4.8: Extracción en low-sync-locking (LSL)

4.2.4. Bloqueo de hilos (TL)

Finalmente presentamos el bloqueo de hilos [36], TL (Thread-Locking), un algoritmo de balance más complejo que los anteriores, que integra la ausencia de sincronización bajo una operación normal lograda en GL y a la vez decrementa el tiempo requerido para una operación de robo de datos utilizando un subconjunto de las listas locales como posibles víctimas como en LSL.

El algoritmo funciona como sigue. Cuando un HA intenta acceder a su lista local para insertar o extraer un elemento, verifica que no haya sido marcado como posible víctima. De haber sido marcado, señala un acuse de recibo en la variable local

`refill_state` con una operación atómica y esperará que termine la operación de robo, o el HA ladrón lo descarte como posible víctima y cancele la operación (Figura 4.9, líneas 2-7). Ni la operación de inserción, ni la operación de extracción requieren de sincronización alguna a menos que su lista local esté vacía. Para efectuar un robo local el ladrón:

I Incrementa la variable global `emptylists`, bloquea el mutex global y señala en una variable local que está haciendo un robo (líneas 16 - 20).

II Itera entre las demás listas locales, escogiendo como posibles víctimas a las R listas con más elementos que no estan siendo usadas por otro ladrón. R es un parámetro preestablecido con valores de 1 a $T-1$, donde T es el número de listas locales (líneas 21 - 26).

III Espera para recibir el acuse de recibo de las R listas marcadas indicando que se encuentran bloqueadas, selecciona la lista más grande y descarta a las demás como posibles víctimas para que continúen su ejecución normal (líneas 27 - 30).

IV Roba la mitad de los datos de la lista seleccionada.

V En caso de que las listas bloqueadas tenga un tamaño menor que 2, se cancela el bloqueo y se intenta de nuevo (líneas 39-46).

VI En caso de haber todavía datos en el nodo debido a que `emptylists` es menor que el número de HA, se intenta de nuevo (líneas 47-57).

VII Si todas las listas locales están vacias se realiza una operación de robo externo de manera similar a BL (líneas 59-64).

Para realizar una operación de robo externo se realizan operaciones similares al robo local, bloqueando sólo R de los T hilos para tomar los datos de la lista más grande, lo que generará un desbalance local esperando que esto se compense con la rapidez que toma realizar una operación de robo.

```
1  int DLML_get(DLML_list *L, DLML_dataitem
       *item) {
2    if ( L->refilling == 1 ) { //
         Acknowledge stealing
3      atomic_increment( L->refill_state)
4      while L->refill_state !=0 && !L->
           refill_cancelled;
5      L->refill_cancelled = 0
6      L->refill_state = 0
7      atomic_decrement( L->refilling )
8    if (L->size > 0) {
9      /*+ normal operation +*/
10     *item = top element of L
11     return 1 // found data
12     /*+ end of normal operation +*/
13   }
14   else { // no data in this thread's list
         , at all
15     /*- local steal operation -*/
16     atomic_increment (dlml_emptylists)
17     ___TRY_AGAIN___:
18     //Block global mutex lock
19     mutex_lock(refilllock)
20     L->refilling = 1
21     loop other lists choosing largest
           lists LL
22       whose flag LL[i]->refilling == 0
23     if |LL| > 0
24      signal requests on each list in LL
           :
25        LL[i]->refill_state = 1
26        atomic_increment( LL[i]->
              refilling )
27      loop waiting for first acknowledge
           :
28       if LL[i]->refill_state == 2 && LL
            [i]->size > 1
29       //--- cancel other requests (
              next sentence)
30       atomic_decrement(LL[i]->
              refill_cancelled)
31       //--- unlock other lists (next 2
              sentences)
32       atomic_decrement (emptylists)
33       mutex_unlock(refilllock)
34       refill L from half of LL[i]
35       //--- unlock L and LL[i]
36       L->refilling = 0
37       LL[i]->refill_state=0
38       return DLML_get(L, item); //now
             it has data
39     else if LL[i]->size < 2
40      //--- cancel req. on LL[i]:
41      atomic_ increment (LL[i]->
             refill_cancelled)
42      delete list i from LL
43      if |LL| == 0
44       L->refilling = 0;
45       unlock( refilllock );
46       goto ___TRY_AGAIN___ // again
47   } else if (emptylists <
         DLML_APP_THREADS_NR) {
48     //there are still data in the
           node
49     if (L->size > 0) {
50       *item = top element of L
51       mutex_unlock(refilllock)
52       return 1; // found data
53     }
54     mutex_unlock(refilllock)
55     yield();
56     goto ___TRY_AGAIN___ //try to
           steal again
57   }
58   /*- end of local steal operation -
         */
59   else {
60     /** external steal operation **/
61      ...
62     /** end of external steal
           operation **/
63     return DLML_get(L, item)
64   }
65 }
66 }
```

Figura 4.9: Extracción en thread-locking (TL)

Capítulo 5

Plataforma y Metodología Experimental

Este capítulo describe la plataforma experimental utilizada para evaluar el desempeño de los varios diseños de MC-DLML presentados en el capítulo 4. Se presenta el hardware y software utilizados, las aplicaciones y la manera de desarrollar los experimentos. Se describen las librerias específicas utilizadas para el desarrollo de MC-DLML para el paso de mensajes y la creación de hilos. Además se describe el software de desarrollo, eclipse y PTP (*Parallel Tools Platform*) el cual facilitó el desarrollo de las aplicaciones de ejemplo.

5.1. Plataforma experimental

En esta sección se describen las plataformas de desarrollo y el *cluster* utilizado para ejecutar los experimentos con las aplicaciones ejemplo.

5.1.1. Hardware

Nuestra plataforma de hardware en la cual se ejecutaron los experimentos es:

- Un *cluster* de 32 nodos *multicore*.
- Cada nodo tiene un procesador Intel de 2.67 GHz i7 920 (Bloomfield con arquitectura Nehalem) con 4 núcleos con tecnología de HyperThreading (también

conocido como HT Technology). Esta tecnología consiste en simular dos procesadores lógicos dentro de un único procesador físico. El resultado es una mejora en el rendimiento del procesador, puesto que al simular dos procesadores se pueden aprovechar mejor las unidades de cálculo manteniéndolas ocupadas durante un porcentaje mayor de tiempo. En estos procesadores se pueden ejecutar hasta 8 hilos de procesamiento concurrentemente

- 4 GB de memoria RAM en cada nodo.
- Disco Duro de 500 GB
- Switch Gigabit de conexión entre los nodos.
- Ubicación: Departamento de Computación, Cinvestav-IPN, Ciudad de México

5.1.2. Software

A continuación se describen las librerias utilizadas en el desarrollo de MC-DLML y el entorno de desarrollo *Eclipse* para el desarrollo y depuración de aplicaciones paralelas.

Pthreads

POSIX Threads o pthreads como comúnmente se le conoce, es un estándar POSIX para la creación y manipulación de hilos de procesamiento. Existen distintas implementaciones para múltiples sistemas operativos. Pthreads define un conjunto de tipos, funciones y constantes en C, agrupados en la cabecera pthread.h

Hay alrededor de 100 procedimientos definidos en pthread.h todos con el prefijo "pthread" los cuales pueden ser categorizados en cuatro grupos.

- Manejo de hilos. Para crear, terminar, unir hilos, etc.
- Candados mutuamente excluibles (mutex)
- Variables condicionales

- Sincronización entre hilos con candados y barreras

Para compilar pthreads se necesita el compilador estándar GCC con el parámetro "-pthread". Se utilizó la versión 4.1.2 de pthreads para Linux.

Open MPI

La interfaz de paso de mensajes (MPI) es un conjunto de funciones desarrollado para ser portable y utilizado en una gran variedad de computadoras paralelas. Se define un estándar el cual incluye un conjunto de funciones base en Fortran 77 y C. Sus objetivos principales son el alto desempeño, la escalabilidad y la portabilidad. Hoy día es el modelo más usado en el cómputo de alto desempeño. MPI ha sido implementado para casi cualquier arquitectura existente de memoria distribuida, lo que hace que los programas desarrollados con MPI tengan una alta portabilidad, además de ser más rápidos debido a que cada implementación está optimizada para el hardware donde se ejecuta.

Open MPI es una implementación de MPI que combina tecnologías y recursos de otras implementaciones como LAM/MPI y FT-MPI. Es usada por varias de las supercomputadoras más rápidas del mundo reportadas en el top500 [2] incluyendo Roadrunner la cual fue la computadora más rápida del mundo de Junio del 2008 a Noviembre de 2009 y por K, la supercomputadora más rápida desde Junio de 2011 hasta el momento [2].

Una de las características más importantes de Open MPI es la estabilidad de la concurrencia entre hilos brindando los cuatro niveles de soporte para hilos MPI establecidos en el estándar.

Nuestro *cluster* utiliza la versión 1.4.1 de Open MPI.

Eclipse

Eclipse es un entorno de desarrollo para múltiples lenguajes con un sistema de *plugins* extensible. Está escrito en Java y puede ser usado para desarrollar aplicaciones en Java

y por medio de sus *plugins* desarrollar en otros lenguajes como C, C++, PHP, Scala, Python, etc.

El *plugin* para desarrollo de aplicaciones en C y C++ es CDT. Contiene una interfaz gráfica de usuario para el desarrollo de aplicaciones. Un modelo de estructura de archivos y recursos para la creación de proyectos. Una estructura de depuración de aplicaciones y mecanismos para el desarrollo colaborativo de software y manejo de versiones.

Se utiliza el *plugin* para desarrollo de aplicaciones paralelas PTP (Parallel Tools Platform). Este *plugin* incluye una amplia variedad de arquitecturas y bibliotecas paralelas, un depurador escalable paralelo y una interfaz de usuario que simplifica la interacción del usuario con la plataforma donde se ejecutan las aplicaciones paralelas [37]. Algunas características convenientes de este entorno de desarrollo es la posibilidad de medir el desempeño de las aplicaciones paralelas,el monitoreo y análisis de su comportamiento y la verificación automática de bloqueos innecesarios y *deadlocks*.

Quizá la ventaja más significativa de las anteriormente mencionadas para el desarrollo de las aplicaciones en este trabajo, fue el depurador escalable paralelo (SDM) que permite pausar la ejecución de una aplicación paralela en todo el *cluster*. Este depurador está integrado en PTP y se utiliza de manera similar al depurador default GDB.

Esto permite, no solamente saber el estado en el cual está cada nodo, sino también el valor de las variables en cada hilo y la evaluación dinámica de expresiones. Esto simplifica el desarrollo debido a que es muy complicado encontrar los fallos y errores en una aplicación paralela. En aplicaciones paralelas el uso de la salida estándar para imprimir mensajes como se hace normalmente en aplicaciones secuenciales es confuso y a veces inútil puesto que los mensajes se muestran sin un orden aparente y el estado de las variables cambia constantemente.

Para el desarrollo de esta tesis se utiliza Eclipse versión 3.6 (Indigo) y PTP versión 5.0.

5.2. Aplicaciones

A continuación se describen las aplicaciones de ejemplo utilizadas para medir el desempeño de nuestros algoritmos de sincronización en MC-DLML.

5.2.1. Segmentación de imágenes (SI)

En esta aplicación se reconstruyen imágenes 3D de un cerebro usando Mean-Shift (MSH) [38, 39]. La reconstrucción se logra aplicando MSH sobre un conjunto de cortes o imágenes 2D en cada pixel, lo cual resulta costoso por ser varios cortes y por la alta resolución requerida. Para reducir el tiempo de procesamiento, los cortes se dividen entre los distintos procesadores. A pesar de que el número de cortes es fijo, el costo de procesamiento de cada corte es diferente debido a que el tiempo de ejecución del método MSH varía dependiendo de los cambios en la intensidad de los pixeles. En esta aplicación cada elemento de datos es un corte. Debido a la resolución necesaria, cada elemento se toma un tiempo de aproximadamente 18 segundos en ser procesado, esto es un tiempo elevado que permite ver como interaccionan los mecanismos de sincronización cuando una lista queda inactiva.

5.2.2. Multiplicación de matrices (MM)

Se utiliza una multiplicación simple de matrices $C = A \times B$ donde cada elemento de procesamiento contiene todos los elementos necesarios para calcular un elemento en la matriz resultante C. Una fila completa de A y una columna completa de B y la posición i y j del elemento de la matriz C. A, B y C son matrices $N \times N$ y se experimentaron con matrices de tamaño 400 y 1000. Esta aplicación fue escogida debido a su tamaño fijo y de tiempo de procesamiento predecible por cada elemento. Esta aplicación de ejemplo no genera nuevos elementos para procesamiento lo que permite analizar la rapidez de los mecanismos de robo de datos cuando no hay un desbalance real.

5.2.3. El problema de las N reinas (NAQ)

Consiste en encontrar la posición de N reinas en un tablero de ajedrez de tamaño $N \times N$ sin que se ataquen entre sí [40]. Las soluciones se encuentran explorando todo un árbol de búsqueda, descartando a aquellas que ya no pueden generar una solución. Para este problema se considera un elemento a un arreglo de posiciones de tamaño N con la posición de las reinas colocadas hasta ese momento. Esta aplicación es particularmente interesante porque conforme se va explorando el árbol se van generando nuevos elementos y se van eliminando aquellos elementos que no producen nuevos resultados. Esto genera un desbalance en la carga de las listas la cual permite visualizar como se adaptan los algoritmos de balance tanto interno como externo.

5.3. Organización de los experimentos

Los experimentos fueron diseñados para apreciar el efecto de los distintos algoritmos de balance de carga, con las aplicaciones de distintas granularidades previamente mencionadas. Las aplicaciones son ejecutadas con los mismos parámetros y solamente se varía el algoritmo de balanceo interno. El proceso de balanceo externo es el mismo para todas las aplicaciones, pero debido a que los distintos algoritmos de balance interno permiten tomar distintas cantidades de trabajo del nodo para servir a las peticiones externas, el desempeño global en el *cluster* también debe ser evaluado.

Es por eso que hemos decidido separar la evaluación de nuestros resultados en dos partes: una evaluación en un solo nodo, en donde se puedan observar el desempeño de cada algoritmo de balance y no se tengan al hilo balanceador HB, creando *overhead* con mensajes MPI. Y la evaluación del balance global en el *cluster* de 1 a 6 nodos donde se observa el comportamiento de las distintos algoritmos en múltiples nodos con 8 hilos ejecutándose en cada uno. La razón de porque se ejecutan 8 hilos solamente por nodo es porque es el máximo número de hilos que se pueden ejecutar concurrentemente en un procesador del *cluster* de nuestra plataforma experimental. El siguiente capítulo explora ambas evaluaciones para cada una de las tres aplicaciones.

Capítulo 6

Resultados

En este capítulo se muestran los resultados obtenidos con las tres aplicaciones seleccionadas para probar nuestros algoritmos de balance de carga interno y externo para MC-DLML. La segmentación de imágenes de granularidad grande, la multiplicación de matrices de granularidad media y tamaño estático y el problema de las reinas con una granularidad pequeña pero que genera desbalance en los nodos. Se muestran por cada problema primero los resultados en un nodo para resaltar el balance interno en el *multicore* y después los resultados en el *cluster*, con resultados entre uno y seis nodos para medir su comportamiento. Los resultados se calculan con el tiempo mínimo de tres ejecuciones. Esto se hace debido a que se desea evaluar al algoritmo con mayor velocidad, no el que se comporta más establemente para lo cual se utilizaría un promedio. Cuando se evalúa el balance interno de un procesador *multicore* se reporta de 1 a 10 hilos ejecutados en un nodo y para el balance externo (global en el *cluster*) se ejecutan 8 hilos, esto se debe a que el procesador *Intel i7 920* en cada nodo de nuestro *cluster* puede ejecutar hasta 8 hilos concurrentemente. En general, este es el número máximo de hilos concurrentes para la mayoría de los algoritmos probados antes de que el desempeño empiece a reducirse. Así cuando se grafique el tiempo para N hilos, se inferirá que se ejecutó en $N/8$ nodos.

Los resultados son contrastados con la implementación de DLML para procesadores con un sólo núcleo. Debido a la implementación del protocolo de subasta de DLML

se necesitan al menos dos pares de procesos hermanos aplicación (PA) y DLML (PD) por lo que no se incluyen resultados para la ejecución con un solo par PA-PD. Para DLML el eje X indica el número de PAs ejecutados.

Se excluyen los resultados del balance de carga para el algoritmo de bloqueo de hilos (TL) debido a la dificultad de crear una implementación estable para el robo de carga externo causado por que su implementación esta optimizada para el balance en un *multicore*.

6.1. Segmentación de imágenes

Se presentan los resultados del balance interno y externo para la aplicación de segmentación de imágenes para los algoritmos Basic-Locking (BL), Global-Locking (GL) y Low-Sync-Locking (LSL). Se anexan los resultados del DLML para comparación.

6.1.1. Resultados de balance interno

Se observa en la Figura 6.1 que todos los algoritmos excepto GL son visiblemente más rápidos que DLML, siendo el más rápido LSL. La razón por la cual GL es tan lento para aplicaciones de granularidad gruesa es que necesita que la lista del HA esté bloqueada en la condición de paro para tomarlo como víctima. Debido a que la aplicación tarda un tiempo considerable entre accesos a la lista, ésta sincronización toma mucho tiempo y retrasa el proceso de balance para GL. Esto se intensifica alrededor de los 8 hilos donde se crea un pico visible en el aumento de tiempo de procesamiento. La razón por la cual los tiempos de ejecución decrementan cuando se aumenta el número de hilos es debido a que no todos los HA se pueden ejecutar simultáneamente y algunos se encuentran detenidos por el planificador de sistema. Así, cuando se intenta una operación de robo, el HA ladrón puede tomar datos de esas listas inactivas lo cual reduce el tiempo de espera durante la sincronización. Esto es una mejora, pero no iguala al mejor tiempo de procesamiento para GL que es para 5 hilos. Esto se debe a la alta granularidad de procesamiento de cada elemento de datos.

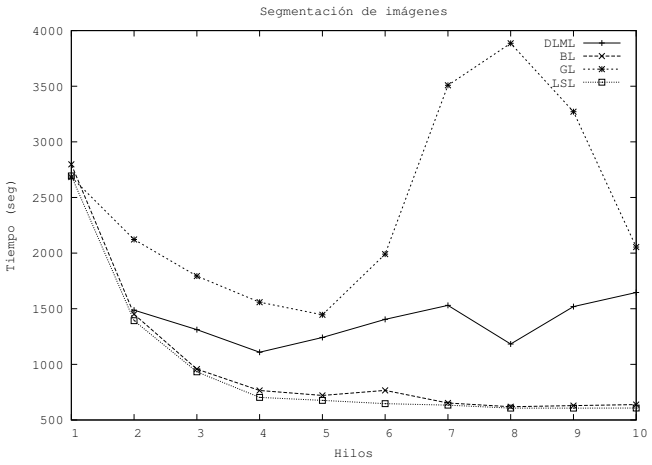

Figura 6.1: Segmentación de imágenes en un nodo (DLML, BL, GL y LSL)

Debido a que si un HA está procesando un dato no está utilizando su lista, es seguro robar datos de él. Esta es la ventaja que LSL busca explorar y el comportamiento esperado.

En la Figura 6.2 se muestra una subsección de la Figura 6.1 de 4 a 8 hilos en escala logarítmica para el eje Y. Se puede observar más claramente que LSL tiene el mejor desempeño, ya que debido a la alta granularidad de procesamiento de cada uno de los elementos de la lista, el costo de las operaciones atómicas sobre la bandera `inside_dlml` es despreciable y supera a BL que utiliza spinlocks.

6.1.2. Resultados de balance externo

La Figura 6.3 muestra la escalabilidad de los algoritmos con balance inter-nodo. Notamos las mismas tendencias que en el balance intra-nodo. El algoritmo más rápido es LSL, seguido de BL. El algoritmo de GL muestra un pobre desempeño aun con respecto a DLML. Como se explico al inicio del capítulo se utilizaron 6 nodos para mostrar el desempeño de los algoritmos en el *cluster* debido a que, son suficientes para extrapolar el comportamiento del balance.

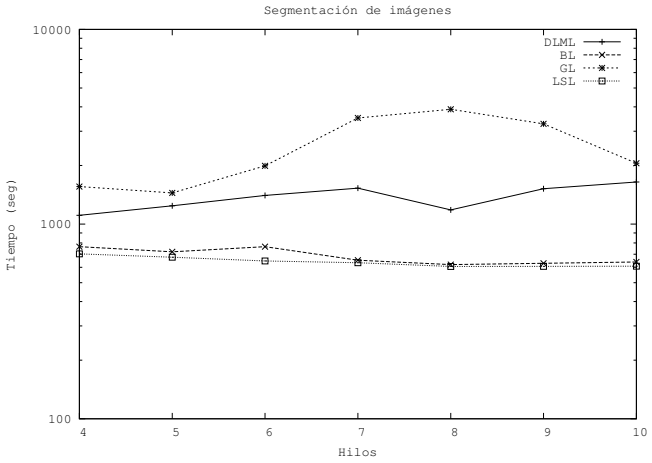

Figura 6.2: Segmentación de imágenes en un nodo (DLML, BL, GL, LSL) de 4 a 10 hilos

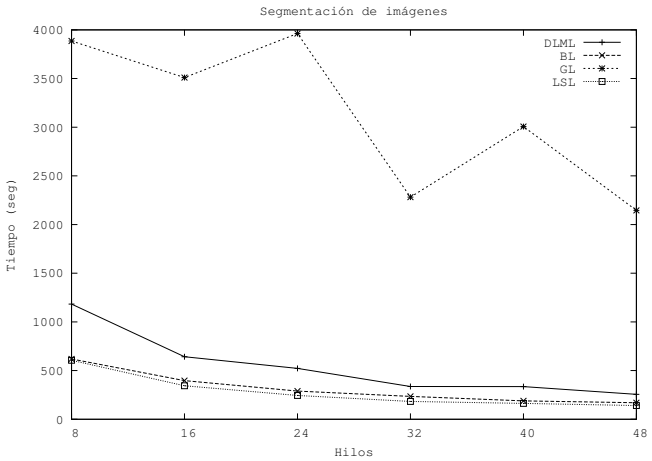

Figura 6.3: Segmentación de imágenes en varios nodos (DLML, BL, GL y LSL)

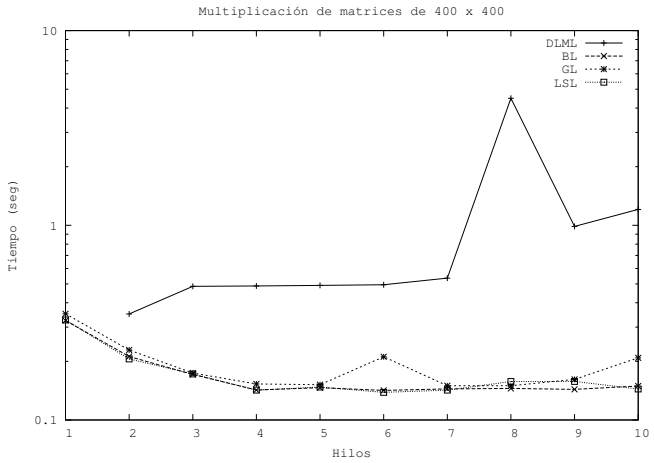

Figura 6.4: Multiplicación de matrices con 400 elementos en un nodo (DLML, BL, GL y LSL)

6.2. Multiplicación de matrices

Se presentan los resultados del balance interno y externo para la aplicación de multiplicación de matrices $N \times N$ con tamaños $N = 400$ y $N = 1000$ para los algoritmos Basic-Locking (BL), Global-Locking (GL) y Low-Sync-Locking (LSL). Se anexan los resultados de DLML para comparación.

6.2.1. Resultados de balance interno

La Figura 6.4 nos muestra los tiempos de ejecución para la multiplicación de matrices $N \times N$ con 400 elementos. Esta aplicación tiene la característica de ser estática, puesto que todos los elementos son insertados al inicio y distribuidos equitativamente, por lo que después de 4 hilos trabajando concurrentemente ya no se reduce notable el tiempo de procesamiento. Observamos que, para aplicaciones que no tienen desbalance, todos los algoritmos se comportan muy similarmente.

Al incrementar el tamaño de la matriz a 1000 (Figura 6.5) cada elemento de la lista tarda más en procesarse y se aumenta el número de elementos a procesar. Se distingue que el costo de sincronización de todos los algoritmos en MC-DLML se vuelve mayor que el que tiene DLML. Esto se debe a que las listas toman casi el

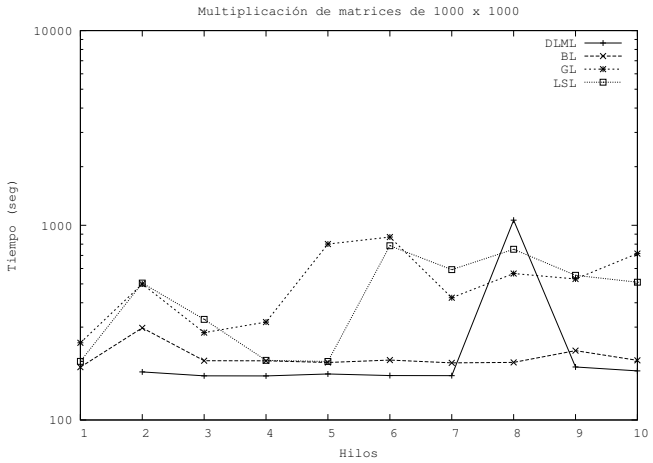

Figura 6.5: Multiplicación de matrices con 1000 elementos en un nodo (DLML, BL, GL y LSL)

mismo tiempo en ser procesadas, y cuando se vacía una de ellas, intenta robar la mitad de los elementos de las listas restantes. Esto obviamente genera un desbalance que vuelve inestable la ejecución y afecta el desempeño. Observamos que el algoritmo con un comportamiento más predecible es BL que, aunque no tiene una reducción del tiempo de procesamiento a medida que incrementan el número de hilos, el tiempo se mantiene. Destaca que DLML muestra un pico en las Figuras 6.4 y 6.5 en la ejecución para 8 procesos concurrentes, esto obedece que 8 es el número de proceso concurrentes que se pueden ejecutar, por lo que, cuando todas las listas están vacías excepto una, los PD inician el proceso de subasta y se genera competencia por los núcleos entre los 8 procesos PD pidiendo datos entre si y los 8 PA solicitando datos a los PD. Cuando el número se incrementa a 9, el PA con datos termina de procesar los elementos en su lista y responde a los demás procesos que ya no hay datos, terminando así el protocolo de subasta más rápidamente que con 8.

6.2.2. Resultados de balance externo

Los resultados obtenidos para varios nodos en las Figuras 6.6 para $N = 400$ y 6.7 para $N = 1000$ son bastante interesantes. Se puede apreciar que cuando la granularidad es

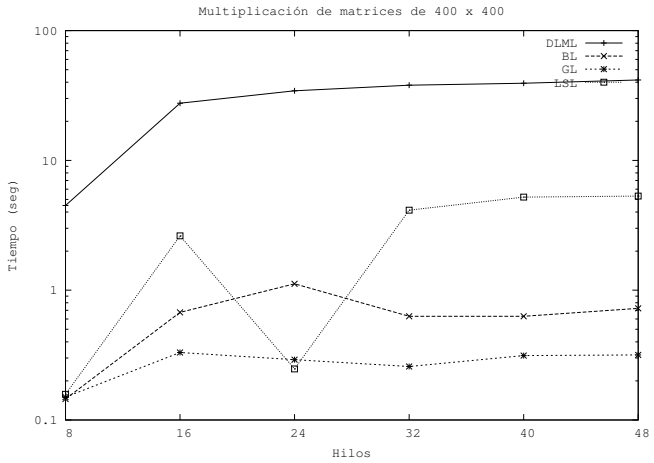

Figura 6.6: Multiplicación de matrices con 400 elementos varios nodos (DLML, BL, GL y LSL)

menor como en 6.6, ningún algoritmo reducirá su tiempo de procesamiento, ni siquiera DLML, puesto que tomará más tiempo enviar los datos a otro nodo que procesarlos internamente. Observamos que todos los algoritmos de MC-DLML son más rápidos en varios ordenes de magnitud que DLML, siendo el más rápido GL puesto que cuando realiza robos externos, deja menor desbalance dentro del nodo víctima.

Al incrementar el tamaño de N a 1000 elementos continúan teniendo un mejor desempeño los algoritmos para MC-DLML, sin embargo ahora el algoritmo de balance más rápido es BL, esto se puede explicar porque ahora el costo del bloqueo con *spinlocks* es menor al costo de sincronización debido a que la mayor parte del tiempo los hilos se encuentran procesando los datos, por lo que obtener el candado para bloquear la lista será rápido. Esto beneficia también a LSL que tiene una reducción de tiempo de procesamiento inicial en 16 y 24 hilos en la Figura 6.7. Recordemos que LSL considera como posibles víctimas a los nodos que se encuentren procesando un dato. Una vez que el desbalance generado por el robo de datos supera a la reducción de tiempo obtenida por la distribución de los datos en paralelo, su tiempo de procesamiento total aumenta, como se ve en la Figura 6.7 esto ocurre en 32 hilos.

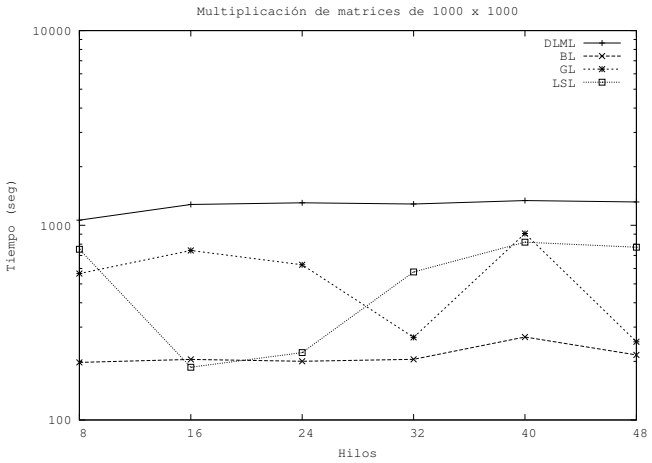

Figura 6.7: Multiplicación de matrices con 1000 elementos en varios nodos (DLML, BL, GL y LSL)

6.3. Problema de las N reinas

Se presentan los resultados del balance interno y externo para la aplicación del problema de las reinas con 14 y 16 reinas para los algoritmos Basic-Locking (BL), Global-Locking (GL) y Low-Sync-Locking (LSL). Se anexan los resultados de DLML para comparación. En esta sección todas las gráficas tienen escala logarítmica en el eje Y.

6.3.1. Resultados de balance interno

Se muestra en la Figura 6.8 los resultados para 14 reinas. Recordemos que esta aplicación inserta nuevos datos irregularmente en las listas por lo cual se genera un desbalance. Es destacable que todos los algoritmos para balance intra-nodo para MC-DLML tienen un mejor desempeño que el DLML original. Este desempeño se aprecia en una reducción de tiempo casi constante la cual se estabiliza en los 8 hilos. Como se mencionó anteriormente, 8 hilos es el número máximo de hilos concurrentes que pueden ejecutarse en un procesador *multicore* de nuestro *cluster*.

Debido a que todos los algoritmos tienen un desempeño similar para 14 reinas, se muestran los resultados para 16 reinas. Al incrementar el número de reinas en el tablero se incrementa factorialmente el número de elementos a procesar. El tiempo de

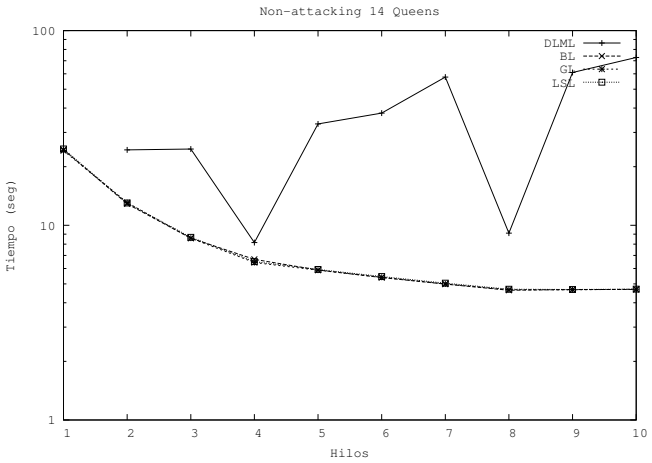

Figura 6.8: 14 Reinas en un nodo (DLML, BL, GL y LSL)

procesamiento de cada elemento varía muy levemente, sin embargo, se generan más elementos de cada elemento procesado de manera irregular en cada HA, lo que genera un mayor imbalance. En la Figuras 6.9 se muestran los tiempos de procesamiento para 16 reinas. Tiene una reducción de tiempo menor, a medida que se aumenta el número de hilos concurrentes, con respecto a la Figura 6.8 para 14 reinas. Podemos deducir entonces que todos los algoritmos de balance intra-nodo para MC-DLML tienen un desempeño muy similar con elementos de granularidad pequeña y no son afectados por el número de datos ni la cantidad de imbalance que se genere.

6.3.2. Resultados de balance externo

Finalmente se presentan los resultados para 14 y 16 reinas (Figuras 6.10 y 6.11). El balance externo muestra una reducción de tiempo importante en todos los algoritmos teniendo un desempeño superior los algoritmos de balance para MC-DLML. Los más rápidos para ambas Figuras son BL y LSL. El algoritmo muestra un desempeño pobre para la sincronización de GL en 2 nodos debido a que envía la mitad de los datos para el procesamiento externo, operación durante la cual ambos nodos se detendrán por completo. Ya que cada elemento tarda muy poco en ser procesado, por cada operación de robo externo, se dejarán de procesar cientos de elementos lo que afectará el

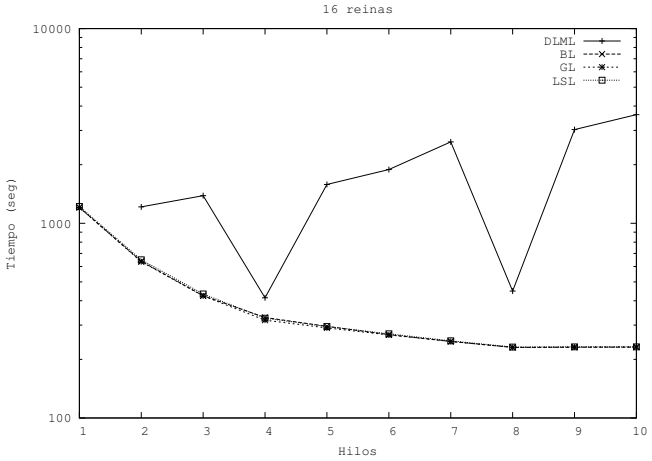

Figura 6.9: 16 Reinas en un nodo (DLML, BL, GL, LSL)

desempeñó final.

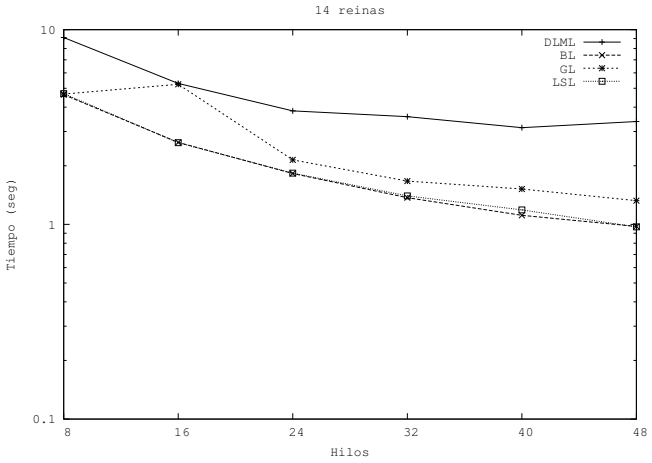

Figura 6.10: 14 Reinas en varios nodos (DLML, BL, GL y LSL)

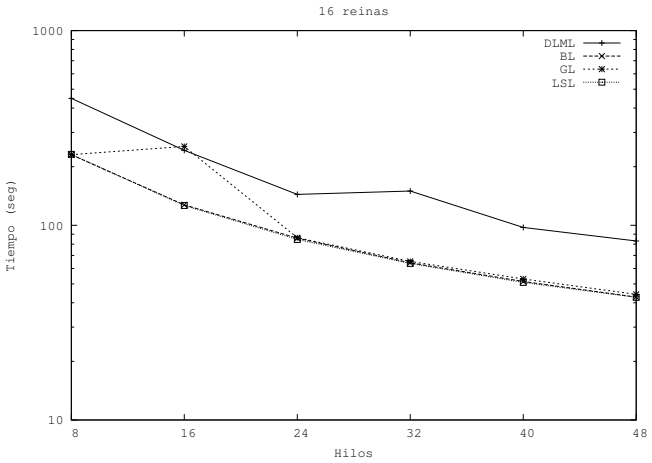

Figura 6.11: 16 Reinas en varios nodos (DLML, BL, GL y LSL)

6.4. Resumen

En este capítulo presentamos los resultados para nuestras implementaciones de los algoritmos propuestos para balance de carga en el capítulo anterior. BL, GL y LSL.

Los resultados fueron evaluados en un solo nodo para apreciar mejor el desempeño interno y en varios nodos para apreciar el balance externo.

El algoritmo más rápido para la segmentación de imágenes fue LSL debido a la alta granularidad de procesamiento de cada elemento lo que permitía tomar elementos de las listas en fases seguras.

El algoritmo más estable para la multiplicación de matrices fue BL y el más rápido para el balance externo fue GL porque genera menor imbalance al realizar una operación de robo.

Para el problema de las reinas, todos mostraron un buen desempeño y una alta adaptabilidad al imbalance generado por la aplicación sin importar el número de elementos a procesar siendo BL y LSL los mejores para este tipo de aplicaciones.

Capítulo 7

Conclusiones

Esta tesis presentó MC-DLML, una nueva versión de diferentes algoritmos de balance para *clusters multicore* de DLML. MC-DLML continúa con el trabajo anterior agregando al protocolo de comunicación entre nodos existente basado en paso mensajes, una capa interna de comunicación en el *multicore* basada en memoria compartida.

Aunque DLML puede ser usado sin modificaciones en *cluster multicore*, ejecutando dos procesos por cada núcleo del procesador, la comunicación es basaba en paso de mensajes. En MC-DLML se utilizan hilos para la capa interna del nodo que permiten la comunicación más directa entre ellos a través del uso de memoria compartida evitando la copia de los datos a mensajes y viceversa para comunicar a procesos aplicación en un mismo nodo. Esto también elimina el uso de procesos DLML *hermanos* para administrar las listas de datos dentro del nodo.

Se analizaron distintas organizaciones para los datos dentro del nodo. Se tenían tres opciones disponibles: una lista global, listas locales por cada hilo aplicación y una organización híbrida con una lista global y listas locales en cada nodo. Un factor que se descubrió que afecta el desempeño, es la localidad de caché y la competencia por la memoria que genera fallos de caché. Se decidió utilizar una lista local por cada núcleo, utilizada como pila por ser la más eficiente en espacio y velocidad de acceso. Esto es, debido a que al insertar y extraer en la misma lista se preservan los datos en caché para el consumo del hilo propietario de la lista. Si la lista es consumida

como pila, se explora los elementos más recientemente insertados, lo que ocasiona una exploración primero en profundidad (*depth first*), lo cual es más eficiente en espacio de memoria. Lo anterior es particularmente importante para aplicaciones de crecimiento irregular que generan desbalance y pueden agotar la memoria disponible. Debido a que cada hilo de aplicación tiene su propia lista de datos, se reduce la competencia por localidades de memoria y reducen los fallos de caché.

Se conservó el mecanismo de subasta para el balance externo usado en DLML, utilizando para la comunicación a un hilo balanceador en cada nodo. Este diseño obedece a tres motivos principales. 1) mantener el protocolo de balanceo de carga externo simple 2) la eliminación de los procesos DLML para administrar las listas de datos y realizar el balanceo como en la versión anterior 3) la utilización de un nivel de soporte MPI para comunicación de hilos *funneled*. El nivel *funneled* permite la utilización de varios hilos en una aplicación pero restringe las llamadas de funciones MPI a sólo uno de ellos, por lo que produce menos *overhead* de sincronización para el envío de mensajes y una comunicación más eficiente.

Para el balance de carga entre las listas locales se desarrollaron cuatro algoritmos distintos. *Basic-Locking* (BL) es el algoritmo más simple basado en bloquear las listas locales cada una con un *spinlock*. *Global-Locking* (GL) utiliza una sola bandera `stopthreads` para bloquear a todas las listas y robar datos. *Low-Sync-Locking* (LSL) que identifica fases seguras en la ejecución para realizar un robo. Finalmente se desarrolló *Thread-Locking* (TL), un algoritmo que toma un subconjunto de las listas, escoge la que tiene más datos y rápidamente desecha a las demás posibles para que continúen su ejecución normal. Esto hace a TL altamente paralelo con una sincronización mínima. Esto, aunque es una ventaja para el desempeño interno, no permitió desarrollar una versión estable para el balance externo y se excluyo de los experimentos realizados.

Para medir el desempeñó de los algoritmos se utilizaron tres aplicaciones con distintas características. La segmentación de imágenes que tiene un tamaño fijo de elementos pero toma un tiempo elevado y variante para procesar a cada uno. La

multiplicación de matrices que tiene un tamaño estático y toma una cantidad media de tiempo para procesar cada uno. Y el problema de las N reinas que procesa cada elemento muy rápidamente pero genera nuevos elementos irregularmente, desbalanceando la carga de trabajo en el *cluster* y dentro de cada procesador *multicore*.

Los resultados obtenidos muestran que todos los algoritmos son altamente resistentes al desbalance de la carga pero muy susceptibles a las aplicaciones sin desbalance, llegando incluso a afectar el desempeño al tratar de balancearla constantemente (diferentes algoritmos mostraban un mejor desempeño en cada uno de los problemas). MC-DLML tiene en general un mejor desempeño que DLML, llegando incluso a ser ejecutado en la mitad del tiempo tomando ventaja del uso de la memoria compartida para la sincronización.

En la aplicación de la multiplicación de matrices el desempeño inclusive empeoró al ejecutarse en más de un nodo para **todos** los algoritmos de balance.

Se propone como trabajo a futuro, la creación de una versión estable del algoritmo de TL para que sea comparado con los demás algoritmos de balance. Además, la integración de mecanismos adaptables de evaluación para decidir que cantidad de datos a enviar a otro nodo, basados en el tiempo de procesamiento de cada elemento y la predicción del tiempo que tomará enviarlos contra el tiempo que tomará procesarlos localmente.

Un punto importante para continuar con el trabajo de esta tesis, es probar los algoritmos de balance en *clusters* con diferentes arquitecturas a la utilizada para realizar pruebas. Se utilizó un *cluster* con procesadores Intel i7 920 (*Bloomfield*) con arquitectura *Nehalem* que tiene un bus de datos directo a la memoria. Intel tiene nuevos procesadores *multicore* con un bus circular de datos (Sandy bridge) y diferentes técnicas de predicción de acceso a la memoria que resultan en un comportamiento distinto de la caché. Intel también se encuentra desarrollando la denominada arquitectura Intel MIC (*Many Integrated Core*) con 48 núcleos dentro de un solo chip. AMD también ha desarrollado procesadores *multicore*, como el Opteron, compatibles con MC-DLML. Es recomendable hacer pruebas de desempeño de MC-DLML en estas

otras arquitecturas para expandir la usabilidad de la librería y analizar el verdadero efecto del comportamiento de la caché.

MC-DLML fue desarrollado basado en las bibliotecas Pthreads y Open MPI. Existen otras librerías con un mayor nivel de abstracción para el desarrollo de aplicaciones paralelas como Intel Cilk Plus ó Intel Threading Building Blocks (TBB). Ambas librerías integran mecanismos automáticos de balance de carga con los cuales se puede comparar MC-DLML para enriquecer nuestro trabajo.

También se deja como trabajo a futuro el desarrollo de un mecanismo de adaptabilidad para utilizar el mecanismo de balanceo de carga interna que mejor se adapte al comportamiento de la carga de trabajo.

Otro punto importante es que no se cuenta con una interfaz lo suficientemente sencilla como para utilizar a MC-DLML como una librería de producción. Esto se podría lograr con la precompilación del código utilizando palabras clave de manera similar a OpenMP. Esto está fuera del alcance de esta tesis y se deja como trabajo a futuro.

Bibliografía

[1] G.S. Almasi and A. Gottlieb. *Highly parallel computing*. The Benjamin/Cummings series in computer science and engineering. Benjamin/Cummings Pub. Co., 1994.

[2] Top500.org. Top500 supercomputing sites. http://www.top500.org/lists/2011/11, 2012 (accesado enero, 2012).

[3] Samuel H. Fuller and Lynette I. Millett. Computing performance: Game over or next level? *IEEE Computer*, 44(1):31 – 38, Enero 2011.

[4] OpenMP.org. Openmp.org. http://openmp.org/wp/, 2012 (accesado enero, 2012).

[5] Murray and Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389 – 406, 2004.

[6] Haruto Tanno and Hideya Iwasaki. Parallel skeletons for variable-length lists in sketo skeleton library. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 666–677. Springer Berlin / Heidelberg, 2009.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Enero 2008.

[8] Jorge Buenabad-Chávez, Miguel Castro-García, and Graciela Román-Alonso. Simple, list-based parallel programming with transparent load balancing. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3911 of *Lecture Notes in Computer Science*, pages 920–927. Springer Berlin / Heidelberg, 2006.

[9] J. Santana-Santana, M.A. Castro-Garcia, M. Aguilar-Cornejo, and G. 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*, pages 64 –68, feb. 2010.

[10] M. Snir. *MPI–the Complete Reference: The MPI-2 extentions*. Scientific and engineering computation. MIT Press, 1998.

[11] IBM Rational. Cutting-edge multicore development techniques for the next wave of electronics products. white paper. http://www-01.ibm.com/software/rational/info/multicore/, 2012 (accesado enero, 2012).

[12] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer Berlin / Heidelberg, 1993.

[13] Herbert Kuchen. A skeleton library. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 85–124. Springer Berlin / Heidelberg, 2002.

[14] D.B. Skillicorn. *Foundations of Parallel Programming*. Cambridge International Series on Parallel Computation. Cambridge University Press, 2005.

[15] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. Research monographs in parallel and distributed computing. Pitman, 1989.

[16] Marco Danelutto, Roberto Di Meglio, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. *Programming Languages for Parallel Processing*, chapter A methodology for the development and the support of massively parallel programs, pages 319–334. IEEE Computer Society Press, Diciembre 1994.

[17] P.J. Parsons and F.A. Rabhi. Specifying problems in a paradigm based parallel programming system, 1995.

[18] Duncan K G Campbell. Towards the classification of algorithmic skeletons. *Computer*, (YCS 276):1–20, 1996.

[19] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In José Cunha and Pedro Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 613–613. Springer Berlin / Heidelberg, 2005.

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

[21] E. Alba, F. Almeida, M. Blesa, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, G. Luque, J. Petit, C. Rodríguez, A. Rojas, and F. Xhafa. Efficient parallel lan/wan algorithms for optimization. the mallba project. *Parallel Computing*, 32(5-6):415 – 440, 2006.

[22] J. González, C. León, and C. Rodríguez. An asynchronous branch and bound skeleton for heterogeneous clusters. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 157–180. Springer Berlin / Heidelberg, 2004.

[23] Amazon Elastic Mapreduce. Amazon elastic mapreduce. http://aws.amazon.com/es/elasticmapreduce/, 2012 (accesado enero, 2012).

[24] Charles Wright. Hybrid programming fun: Making bzip2 parallel with mpich2 & pthreads on the cray xd1. www.asc.edu/seminars/Wright_Paper.pdf, 2006.

[25] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[26] Christiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Febrero 1996.

[27] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. http://code.google.com/p/disruptor/, 2011.

[28] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[29] P. Berenbrink, T. Friedetzky, and L.A. Goldberg. The natural work-stealing algorithm is stable. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 178 – 187, Octubre 2001.

[30] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356 –368, Noviembre 1994.

[31] Michael Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 212–221, New York, NY, USA, 1998. ACM.

[32] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6271 of *Lecture Notes in Computer Science*, pages 217–229. Springer Berlin / Heidelberg, 2010.

[33] M. Korch and T. Rauber. Evaluation of task pools for the implementation of parallel irregular algorithms. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 597 –604, Agosto 2002.

[34] Jorge Buenabad-Chávez, Miguel Castro-García, Graciela Román-Alonso, José Luis Quiróz-Fabián, Daniel M. Yellin, Manuel Aguilar Cornejo, and Edgar Fabián Hernández-Ventura. Reducing communication overhead under parallel list processing in multicore clusters. *8th International Conference on Electrical Engineering, Computing Science and Automatic Control, México.*, 1:780 – 785, 2011.

[35] J. Buenabad-Chávez, M. A. Castro-García, J. L. Quiroz-Fabián, D. M. Yellin, G. Román-Alonso, and E. F. Hernández-Ventura. Low-synchronisation work stealing under parallel data-list processing in multicores. *International Conference on Parallel and Distributed Techniques and Applications, U.S.A*, 1:850–856, 2011.

[36] J. Buenabad-Chávez, M. A. Castro-García, J. L. Quiroz-Fabián, D. M. Yellin, G. Román-Alonso, and E. F. Hernández-Ventura. Thread-locking work stealing under parallel data list processing in multicores. *Parallel and Distributed Computing and Systems, U.S.A*, pages 190–197, 2011.

[37] eclipse.org. Ptp - parallel tools platform. http://eclipse.org/ptp/, 2012 (accesado enero, 2012).

[38] J.R. Jiménez-Alaniz, V. Medina-Bañuelos, and O. Yañez-Suárez. Data-driven brain mri segmentation supported on edge confidence and a priori tissue information. *Medical Imaging, IEEE Transactions on*, 25(1):74 –83, jan. 2006.

[39] G. Roman-Alonso, J.R. Jimenez-Alaniz, J. Buenabad-Chavez, M.A. Castro-Garcia, and A.H. Vargas-Rodriguez. Segmentation of brain image volumes using the data list management library. In *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pages 2085 –2088, Agosto 2007.

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