Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Departamento de Computación

Programación con Listas de Datos para Cómputo Paralelo en Clusters

Tesis que presenta

Miguel Alfonso Castro García

para obtener el Grado de

Doctor en Ciencias

en la Especialidad de Ingeniería Eléctrica

Directores de la Tesis

Dr. Jorge Buenabad Chávez
Dra. Graciela Román Alonso

México, D.F. Mayo de 2007

Resumen

La programación paralela en general requiere que el programador realice el particionamiento del problema y asigne las partes entre los procesadores. Estas dos tareas se pueden complicar dependiendo del problema, y en el caso de usar un cluster como plataforma de ejecución pueden ser inadecuadas para un buen desempeño al existir factores ajenos al problema como: la multiprogramación y la heterogeneidad del hardware. Bajo estos factores el balance de carga (datos) es necesario para mejorar el desempeño.

Para facilitar la programación paralela, esta tesis presenta un modelo de programación cuasi-secuencial en el que el particionamiento, la asignación y el balance de datos entre los procesadores se realiza de manera transparente al programador. Nuestro modelo, basado en listas de datos, sólo requiere que el programador proporcione sus datos organizados en una lista, los inicialice y realice llamados a funciones organizadas en una biblioteca. Los llamados incluyen funciones para la manipulación de las listas, como `insertar y eliminar` elementos, y funciones para soportar el paralelismo.

Nuestro modelo facilita la programación y tiende a mejorar el rendimiento de los siguientes tipos de aplicaciones: 1) Aplicaciones con una cantidad fija de datos, con procesamiento constante o variable, es decir, misma o diferente carga computacional en el procesamiento de los datos; y 2) Aplicaciones con una cantidad variable de datos con procesamiento constante o variable. En nuestro modelo se hace un balance de datos entre los procesadores, basado en un algoritmo de subasta, de manera automática (transparente al programador) dentro de las funciones en nuestra biblioteca de manejo de listas.

Nuestro enfoque lo hemos probado en un cluster con las siguientes aplicaciones: Ordenamiento paralelo (datos fijos, procesamiento fijo); algoritmos evolutivos (datos fijos, procesamiento variable), N-Reinas (datos variables, procesamiento variable) y Procesamiento digital de imágenes (datos fijos/variables, procesamiento variable). Los resultados obtenidos muestran que nuestro sistema mejora el desempeño de manera notable, cuando hay necesidad de balance de carga.

Agradecimientos

Quisiera agradecer a las instituciones, principalmente al Consejo Nacional de Ciencia y Tecnología, CONACyT, por la beca que me otorgaron, la cual ayudó a sostenerme económicamente y así poder dedicarme de manera completa a los estudios de doctorado.

De manera similar agradezco al Centro de Investigación y de Estudios Avanzados del IPN, CINVESTAV (o cinves como lo usamos la mayoría de los estudiantes), cuyas máximas autoridades siempre me apoyaron, ya fuera en lo académico o en apoyo económico para congresos o becas terminales para obtención del grado.

De la misma forma agradezco a la Sección de Computación del Departamento de Ingeniería Eléctrica del CINVESTAV, quién me ofreció el plan de estudios de Doctor en Ciencias. En la Sección de Computación obtuve dos apoyos principalmente, el académico a través de mis profesores y el técnico a través de la infraestructura y servicios que me ofrecieron. En general quiero agradecer a todos los profesores de la sección, pero en especial a aquellos que me impartieron algún curso o seminario que definitivamente me ayudó a ampliar mi formación. Los doctores que quiero agradecer son: (en estricto orden de aparición) el Dr. Óscar Olmedo, Dr. Arturo Díaz, Dr. Sergio Chapa, Dr. Carlos Coello y el Dr. Matías Alvarado.

De manera paralela quiero agradecer a la Universidad Autónoma Metropolitana, UAM, quien a través de la división de Ciencias Básicas e Ingeniería me apoyaron con la beca por estudios de posgrado. Sin esta beca, hubiera tenido que seguir trabajando en la UAM y dedicarme de medio tiempo al doctorado, lo cual retrasaría la finalización de mis estudios de doctorado.

Un lugar especial se los reservo a mis asesores, Graciela Román y Jorge Buenabad, quienes aparte de dirigirme profesionalmente durante todo el doctorado y lograrme transmitir en todo momento sus conocimientos, fueron siempre unos amigos, los cuales en todo momento me apoyaron, motivaron e impulsaron a seguir adelante.

Quiero agradecer también a las secretarias del cinves y de la uam, Sofi, Feli, Flor, Mara y Paula por apoyarme siempre en todos los papeleos necesarios durante mis

estudios.

No podrían faltar los amigos o compañeros de doctorado, los cuales unos a otros siempre nos apoyamos para seguir adelante, además de ir aprendiendo los unos de los otros. Aquí quiero agradecer (por orden de sala) a Ulises, Rafael, Héctor, Lety, Luis, Lorena y Amilcar. Agradezco de manera especial a Santiago, quien es el amigo con el que más he convivido y ha estado siempre ahí tanto en las buenas como en las malas.

Quiero agradecer a mis padres, Esteban y Rolis, quienes siempre me apoyaron y me incentivaron a continuar en esta empresa. A mis padres les quiero dedicar el grado, ya que se que es un enorme gusto y satisfacción para ellos el ver como sus hijos, nos vamos superando.

Quiero también agradecer a mi hermano Esteban, quien ha sido un gran ejemplo a seguir y sobre todo un buen motivador, más aún desde el principio de la Licenciatura hasta ahora que he obtenido el grado de doctor.

Finalmente quiero agradecer a mi esposa Lupita quien ha soportado esas ausencias, quizá poco entendibles por ella, pero que nunca las ha discutido, al contrario ella en muchas ocasiones me ha presionado a trabajar. También quiero agradecer a mi hijo Miguel Axl, por ser una motivación en esta empresa, quiero disculparme también por aquellos momentos en los cuales por motivos del doctorado, no le dediqué la atención necesaria, situación que ahora va a cambiar. Agradezco a mi hija Liliana Guadalupe, simplemente por nacer y llenarnos de amor día tras día.

Contenido

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

Los clusters[1] y los ambientes de programación paralela han hecho que el cómputo paralelo se encuentre al alcance de la mano. Hoy día los clusters o redes de área local ofrecen un buen rendimiento a bajo costo, y numerosos ambientes (herramientas) de programación facilitan el desarrollo de aplicaciones paralelas.

Para desarrollar aplicaciones paralelas en clusters, generalmente se sigue el modelo SPMD (Single Program, Multiple Data). En el modelo SPMD, el mismo programa (código de la aplicación) corre en los diferentes procesadores. En general, el programador debe dividir la carga (datos) entre varios procesadores, y especificar la comunicación entre los mismos para compartir datos y coordinar su operación.

Con el uso de ambientes de programación paralela se facilitan u ocultan al programador algunos de los pasos anteriores. Estos ambientes también determinan la facilidad para diseñar e implementar balance de carga o redistribución dinámica de datos. En clusters, el balance de carga es esencial para un buen desempeño debido a la multiprogramación, la posible heterogeneidad del equipo y el comportamiento dinámico de algunas aplicaciones. En la multiprogramación hay varias aplicaciones externas compartiendo el cluster (no dedicado). Estas aplicaciones externas afectan el procesamiento de los datos, al compartir el procesador, conduciendo a un desbalance de los datos a

[1]Usamos "cluster" para referimos a LANs o redes de área local

procesar. Con la heterogeneidad, algunos nodos del cluster son más rápidos que otros. En este caso, los nodos rápidos terminarán antes su procesamiento en comparación con los nodos lentos. Finalmente, existen aplicaciones dinámicas cuyos datos son generados durante la ejecución, es decir, la descomposición inicial de los datos puede generar subsecuentes datos y descomposiciones durante el tiempo de ejecución. Este comportamiento dinámico hace que unos procesadores tengan más datos que otros y por ende tarden más en procesarlos.

Algunos ambientes de programación facilitan el diseño e implementación de balance de carga, o incluso eliminan su necesidad. Estos ambientes se proponen como bibliotecas de paso de mensajes, sistemas de memoria compartida o compartida-distribuida, skeletons y tipos de datos abstractos compartidos, entre otros.

Las bibliotecas de paso de mensajes ofrecen diferentes funciones para la transmisión de datos. Algunas de estas funciones permiten la comunicación entre procesos así como su sincronización. En general con las bibliotecas de paso de mensajes se pueden generar los programas más eficientes. Sin embargo, excepto para aplicaciones muy simples, la programación con paso de mensajes es complicada debido a que el programador se encarga de todo el protocolo de comunicación y sincronización que incluye la identificación de procesos, así como el envío y recepción de mensajes entre los mismos. En cuanto a la distribución de procesos, esta la hacen de manera cíclica en todos los procesadores. También ofrecen una distribución explícita de procesos, es decir, el programador decide donde ejecutarlos. Sin embargo, las bibliotecas de paso de mensajes no ofrecen un balance automático de datos, éste tiene que ser implementado por parte del programador. Entre las bibliotecas de paso de mensajes más conocidas están PVM [43] y MPI [86, 87].

Los sistemas de memoria compartida-distribuida (Distributed Shared Memory DSM) soportan un espacio de direcciones compartido sobre memoria distribuida [61]. Estos sistemas hacen que todos los accesos a memoria compartida (local o remota) se vean como locales, simplificando así la comunicación entre los procesadores. Sin embargo, debido a los accesos concurrentes a las áreas de memoria compartida (regiones críticas), es necesario que el programador utilice mecanismos de sincronización como candados,

semáforos o monitores, para mantener la coherencia de los datos. En los sistemas DSM no se hace distribución de procesos (se maneja un proceso por procesador), en cuanto a los datos compartidos, estos se migran o replican en todo el sistema distribuido de acuerdo al acceso de los procesadores. Con respecto al balance de datos, este no se hace ya que los datos se migran o replican sin tomar en cuenta si el cluster tiene multiprogramación o si es heterogéneo, lo cual puede afectar el rendimiento. Algunos sistemas DSM son CRL [54], TreadMarks [6] y DDS [35].

Los skeletons son plantillas predefinidas de algoritmos paralelos típicos [29]. Estas plantillas incluyen patrones paralelos como: *pipeline(), distribución_cíclica(), farm()* o *divide&conquer()*; otros skeletons incluyen patrones de balance de datos típicos manejados dentro de clases [72].

Usando skeletons, el programador elige las plantillas necesarias para construir su aplicación. Sin embargo, ésto no siempre es sencillo ya que la aplicación tiene que ajustarse a los skeletons disponibles y es necesario tener en cuenta conocimientos de programación paralela al momento de elegir los skeletons. Adicionalmente, el uso de estas plantillas puede ser complicado debido al gran número de parámetros que manejan, algunos requieren hasta 15-17 parámetros. Otra limitante es que algunos skeletons (por ejemplo eSkel [10]), necesitan plataformas de comunicación especializadas como LA-MPI (Los Álamos-MPI) [7]. Algunos skeletons manejan distribuciones estáticas al inicio del programa, como el skeleton Deal que maneja una distribución cíclica de datos. Otros como SAMBA [72] manejan varias distribuciones como: estática y demanda, entre otras. En la estática se hace una distribución inicial de datos entre los procesadores y nunca mas. En la demanda un procesador maestro distribuye parte de los datos entre todos los procesadores y se queda con el resto. Posteriormente cuando un procesador se queda sin datos le pide al maestro.

Los tipos de datos abstractos (Abstract Data Types ADTs) son una especificación matemática de un conjunto de datos y un conjunto de operaciones sobre los mismos [32]. Algunos ejemplos de ADTs son las pilas, las colas, los árboles y las listas. Estos ADTs tienen las operaciones push, pop, enqueue, dequeue, insert y get entre otras.

Originalmente los ADTs se usaron bajo programación secuencial, pero hoy se usan también bajo programación paralela [84]. En los ADTs paralelos, diferentes elementos del conjunto de datos pueden ser accesados por diferentes procesadores simultáneamente. Mediante este paradigma, se ocultan los pasos de la sincronización y la comunicación, ya que éstos son transparentes en el llamado a las operaciones sobre los datos. Con ADTs, la tarea del programador es organizar sus datos en los ADTs disponibles.

Un ejemplo de ADTs paralelos es SADTs (Shared Abstract Data Types) [45], SADTs encapsula la comunicación entre los procesadores mediante el acceso a sus ADTs, que en su caso particular ofrece colas compartidas, colas compartidas de prioridad y tipos básicos compartidos como enteros o flotantes. El proyecto SADTs también propuso las listas y los árboles compartidos, sin embargo, no se desarrollaron.

Con SADTs se tiene una visión global compartida de sus ADTs, es decir, todos los procesadores de alguna forma ven la misma cola. Para lo anterior SADTs desarrolló diferentes implementaciones en diferentes arquitecturas de hardware. Las implementaciones que desarrollaron son: centralizada, replicada y particionada. Las arquitecturas fueron multiprocesador y clusters.

En la implementación *centralizada* todos los procesadores acceden a una cola única, cuyo acceso es controlado por candados para evitar inconsistencias. Esta implementación tiene el inconveniente de provocar un cuello de botella en la cola única. En la implementación *replicada* se tienen copias de la misma cola distribuidas en todos los procesadores, esta característica permite que los procesadores accedan de manera local a su propia cola. Sin embargo, en los accesos concurrentes por parte de los procesadores a las colas replicadas (copias) es necesario el uso de un protocolo de coherencia (write-update). Cabe mencionar que el protocolo de coherencia se puede volver muy costoso cuando se incrementa el número de procesadores (escalabilidad). En la implementación *particionada* se pierde en parte la visión global compartida, aquí el ADT se encuentra distribuido entre todos los procesadores (no hay copias), es decir, cada procesador tiene una parte de la cola. Con este esquema la coherencia se lleva de fuerte a débil [68], sin embargo, como los autores dicen, se pierde el ORDEN de las colas. Cada vez que

se encola de manera concurrente entre varios procesadores el orden al desencolar no será el mismo (propiedad de linealidad). Si se requiere mantener el ORDEN, es necesario colocar explícitamente barreras de sincronización por parte del programador donde crea necesario [67].

En cuanto a las arquitecturas se utilizaron máquinas multiprocesador como la Cray T3D con arquitectura NUMA. Sobre esta máquina se apoyaron con el uso de la biblioteca SHMEM que es propiedad de Cray [66]. En general bajo esta arquitectura obtienen buenos rendimientos y escalabilidad (100 procesadores aprox.), principalmente por el hardware dedicado en la comunicación entre pares de procesador-memoria. Sin embargo bajo clusters de máquinas INDY SGI con red Ethernet, los costos de comunicación son más altos, en este caso su escalabilidad se reduce a 12-16 procesadores. Esto se debe al costo de la coherencia ya sea fuerte o débil.

En los ADTs paralelos como SADTs no existe propiamente una redistribución de datos, debido a que existen copias de los datos a ser procesados en todos los procesadores. No obstante, existe un balance ya que los procesadores más rápidos procesan más datos del ADT (cola) que los procesadores lentos, logrando una distribución natural y eficiente de los datos. El tipo de aplicaciones que son ideales para SADTs son aquellas donde se requiere un orden, por ejemplo el agente viajero. En el agente viajero las posibles rutas a explorar se van guardando en un cola de prioridad, la prioridad consiste en que las rutas o recorridos completos van al frente de la cola, para poder evaluarse y descartar rutas no factibles.

Esta tesis presenta el diseño de una biblioteca de manejo de listas de datos para cómputo paralelo en clusters. Nuestra biblioteca llamada DLML (Data List Management Library), facilita la programación de algunos tipos de aplicaciones, además de realizar balance de datos. De manera similar a SADTs, DLML encapsula los aspectos de comunicación a través de las operaciones sobre la *lista*.

Las listas no requieren tener un ORDEN (a diferencia de las colas de SADTs) ya que las inserciones y eliminaciones sobre la lista se pueden hacer en cualquier parte de

la lista. Esta característica tiene la ventaja de que las listas se pueden dividir fácilmente en sublistas y seguir cumpliendo con la propiedad de listas (no así las colas). El poder dividir las listas también permite que las sublistas se puedan distribuir o mover de un lado a otro en un sistema paralelo.

Con DLML la programación paralela se transparenta al efectuar una programación casi secuencial en un lenguaje imperativo con llamados a funciones típicas sobre listas. El paralelismo se encapsula dentro de los llamados a las funciones como *get()* e *insert()* donde DLML se encarga del manejo y distribución de las listas en el cluster. Por ejemplo, un procesador normalmente obtiene elementos (datos) de la lista local mediante una operación *get()*. Pero si la lista local esta vacía, DLML busca datos que procesar en las listas remotas del cluster de manera transparente al programador.

La programación con DLML la conducimos a través de una metodología para programación con listas. La metodología incluye varios pasos que van desde la determinación de los datos para las listas, la granularidad o elementos en la lista, la declaración de los tipos de datos en la lista, la inicialización de la lista, el código o procesamiento sobre cada elemento de la lista y la terminación del programa.

En cuanto al balance de datos, DLML usa un algoritmo de balance conocido como subasta o *bidding* [88]. Este algoritmo *bidding* se eligió porque balancea las listas de la aplicación de una manera natural. Adicionalmente de que algoritmo no requiere un ajuste de parámetros por parte del programador. En el algoritmo, cada vez que una lista se queda sin elementos, el proceso DLML busca más elementos en la lista con mayor cantidad de elementos. Mediante este algoritmo, DLML balancea las listas distribuidas, disminuyendo el efecto del desbalance provocado por la multiprogramación, la heterogeneidad y la generación dinámica de datos de las aplicaciones.

En la arquitectura de DLML utilizamos dos procesos por cada procesador del cluster. Un proceso para la aplicación y el otro para DLML. El proceso aplicación se encarga propiamente de la ejecución de la aplicación. En este proceso se hacen los llamados a las funciones típicas sobre listas. Los llamados modifican la lista, la cual es compartida con el proceso DLML. Si una lista esta vacía en un nodo, el proceso DLML correspon-

diente se encarga de buscar elementos en las listas remotas. El proceso DLML también se encarga de recibir peticiones, de aquellos nodos que tienen su lista vacía. Todo el control entre procesos DLML (locales y remotos) se establece mediante un protocolo de comunicación basado en mensajes.

DLML está implementado en lenguaje C, y utiliza MPI [87] como plataforma de comunicación en el paso de mensajes.

Hemos comparado la facilidad de programación y el desempeño de DLML con MPI. La razón de usar MPI es porque es el más portable y tal vez el más usado por el rendimiento que puede ofrecer [57]. SADTs se propuso en el 96, sin embargo el código no se pudo obtener al no estar disponible en internet. En la evaluación de DLML utilizamos diferentes tipos de aplicaciones y las corrimos en diferentes tipos de clusters.

Los tipos de aplicaciones que se probaron fueron aplicaciones estáticas, las cuales manejan un número fijo de datos, y dinámicas que manejan un número variable y desconocido de datos. Las aplicaciones estáticas fueron una aplicación de algoritmo evolutivo [27] y otra de ordenamiento paralelo [79]. Las aplicaciones dinámicas fueron una de tipo Branch & Bound (B&B) [19], otra dividir para vencer de procesamiento digital de imágenes [78] y otra embarazosamente paralela también de procesamiento digital de imágenes [80]. Los clusters utilizados fueron homogéneos y heterogéneos, con multiprogramación y sin multiprogramación.

La programación de las aplicaciones resultó sencilla debido a que, como programador, solo se tuvo que organizar los datos de la aplicación en el ADT lista, esto mediante la definición de los datos en la estructura *lista* predefinida. En el caso del algoritmo evolutivo los elementos de la lista fueron los individuos a evaluar. En el ordenamiento paralelo, cada elemento de la lista es un número a ser ordenado. Para la aplicación dinámica tipo B&B que fue el problema de las N-Reinas, los elementos de la lista fueron las diferentes ramas que representan posibles soluciones. Estas ramas dinámicamente podían crecer (insertar nuevos elementos a la lista) o ser podadas (eliminar elementos de la lista). Para la primer aplicación de procesamiento digital de imágenes los elementos de la lista fueron las imágenes a procesar, las cuales a su vez dependiendo del tipo de

procesamiento se podían volver a subdividir en otras subimágenes, generando nuevos elementos para la lista. En la segunda, los elementos de la lista fueron las imágenes de cortes cerebrales que en su conjunto formaban un objeto tridimensional.

Resumen y mapa de la tesis

Este capítulo ha introducido de manera general la propuesta realizada en esta tesis: Una biblioteca para programación paralela basada en listas de datos. Usando esta biblioteca se facilita la programación paralela de aplicaciones cuyos datos se pueden organizar en una lista. El desempeño de la biblioteca tiende a ser óptimo por el balance de datos que maneja.

El resto de la tesis se organiza de la siguiente forma:

El Capítulo 2 presenta una introducción de los aspectos generales de cómputo paralelo, principalmente en la programación paralela, abarcando los principales modelos de programación. Esto sirve para introducir toda la terminología que es usada en el resto de los capítulos.

El Capítulo 3 presenta el estado del arte de los ambientes y herramientas de programación, describiendo sus características y modelo de programación. Este capítulo sirve para conocer las ventajas y desventajas de los ambientes y herramientas existentes.

En el Capítulo 4 se muestra la necesidad de un balance de carga cuando se ejecutan aplicaciones dinámicas y/o cuando se usan clusters heterogéneos con multiprogramación. En el capítulo también se muestran los aspectos generales del balance de carga.

El Capítulo 5 presenta la motivación de nuestra biblioteca DLML para programación paralela basada en listas. En el capítulo se muestran los aspectos de programación e interfaz al programador.

El Capítulo 6 presenta el diseño de DLML, su arquitectura, los algoritmos de balance de datos y los aspectos de sincronización entre procesos.

El Capítulo 7 muestra la evaluación de DLML.

El Capítulo 8 presenta nuestras conclusiones y las sugerencias para trabajo futuro.

Capítulo 2

Programación Paralela

En este capítulo se presenta una introducción acerca de los aspectos generales de la programación paralela. Estos aspectos incluyen una descripción de las aplicaciones que requieren gran demanda de cómputo y que originan precisamente el uso del cómputo paralelo. Adicionalmente se presentan los principales modelos de programación paralela. Todo lo anterior nos ayuda a introducir la terminología que es usada en el resto de los capítulos.

2.1. Introducción

Durante las últimas décadas, el creciente avance tecnológico y la alta integración en los componentes de los procesadores ha permitido tener computadoras cada vez más rápidas. Sin embargo, aún a la fecha un solo procesador no puede cubrir la demanda de procesamiento de muchas aplicaciones comerciales y científicas. Algunas aplicaciones comerciales con gran demanda de procesamiento son las transacciones bancarias y las búsquedas sobre grandes bases de datos (Google, 8 billones de páginas web). Por otro lado, aplicaciones científicas con gran demanda de cómputo son la predicción del clima, la dinámica de fluidos y el movimiento de cuerpos espaciales (planetas, estrellas) entre otras [75, 99].

Para tener una idea de la magnitud del cómputo requerido para unas aplicaciones,

analicemos el movimiento de cuerpos espaciales. El movimiento de cuerpos espaciales se rige por las fuerzas de atracción que generan unos a otros. En un sistema de tamaño N, cada cuerpo siente la fuerza de atracción de $N-1$ cuerpos, por lo que el número total de fuerzas es $N*(N-1) = N^2 - N \approx N^2$, que se traduce en N^2 cálculos. Una galaxia contiene alrededor de 10^{11} estrellas, por lo que se tendrían que hacer 10^{22} cálculos. Para completar estos cálculos se puede usar alguno de los procesadores más rápidos de la actualidad como el Pentium 4®, el Pentium D® o los procesadores AMD® (los cuales procesan aproximadamente 500 millones de instrucciones por segundo)[5, 17]. Sin embargo, haciendo cuentas, se necesitarían 634195 años de procesamiento ininterrumpido, para lograr una sola iteración.

¿Nuestras opciones?, básicamente dos:

- Dejar que avance la tecnología individual de los procesadores.
- Agruparlos (*union is strength*). [1]

En la primera opción, simplemente hay que dejar pasar los años para poder usar procesadores más veloces (por cierto siempre costosos en su lanzamiento). Por el contrario, tomando la segunda opción no es necesario esperar años, se puede lograr un mejor desempeño agrupando varios procesadores. La idea de agrupar varios procesadores, para obtener más poder de procesamiento y así reducir el tiempo de respuesta, es el principio fundamental del *cómputo paralelo*.

El *cómputo paralelo* se define como la ejecución simultánea de varias tareas en múltiples procesadores con el fin de obtener resultados más rápidos [97]. Las tareas ejecutadas en los distintos procesadores necesitan comunicarse ya sea para sincronizarse[2] o para intercambiar resultados y alcanzar un objetivo en común.

[1] La unión hace la fuerza: Lema que figura en el escudo de armas de la República de Bélgica y expresa la idea del esfuerzo mancomunado, uniendo las individualidades en procura de un logro.

[2] Con sincronización nos referimos a a que se opere al unísono u operar con exacta coincidencia en tiempo y en frecuencia.

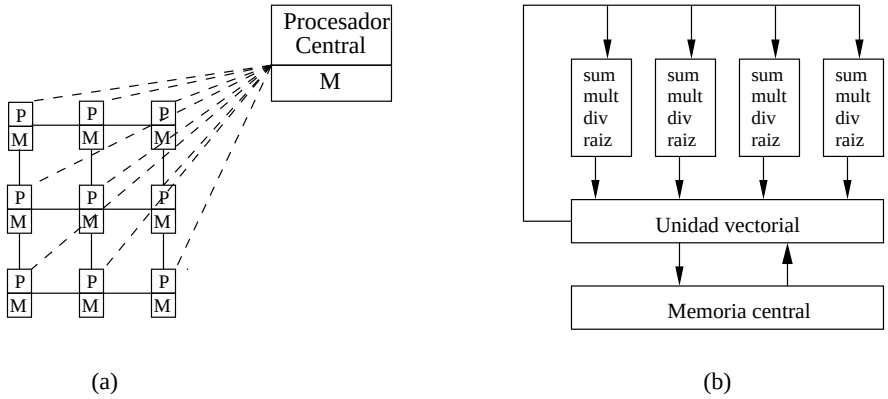

Figura 2.1: Arquitecturas SIMD: (a)arreglo de procesadores; (b)procesadores vectoriales.

En resumen, para poder hacer cómputo paralelo es necesario contar con una arquitectura y con algún modelo o software de programación que nos permita desarrollar aplicaciones multitarea.

2.1.1. Arquitecturas paralelas

Las arquitecturas paralelas, también llamadas *computadoras paralelas*, pueden ejecutar varias tareas (subtareas) de manera simultánea. Las computadoras paralelas han sido clasificadas de diferentes maneras a través de los años. Una de las clasificaciones más conocidas es la de Flynn [40].

En la década de los 70's, Flynn clasifica a las computadoras en cuatro tipos diferentes. Su clasificación se basa en el flujo de *instrucciones* que manejan, y en el número de *datos* que utilizan esas instrucciones: SISD (Single Instruction, Single Data), MISD (Multiple Instruction, Single Data), SIMD (Single Instruction, Multiple Data) y MIMD (Multiple Instruction, Multiple Data).

Los tipos secuenciales SISD y sistólicas MISD se pueden ver con más detalle en Quinn [74]; a continuación describimos los tipos SIMD y MIMD.

En la arquitectura SIMD una misma instrucción es ejecutada sobre un conjunto de

datos distintos. Para ello un solo procesador central se encarga de enviar la instrucción a los procesadores quienes la aplican sobre sus propios datos. La sincronización entre los procesadores no es necesaria, ya que es el procesador central quien la maneja. Existen básicamente dos tipos: arreglos de procesadores y procesadores vectoriales. En el arreglo de procesadores (Figura 2.1(a)) un procesador central se encarga de difundir una a una las instrucciones a un arreglo de procesadores, los cuales reciben la instrucción y la aplican sobre sus propios datos. En los procesadores vectoriales (Figura 2.1(b)) se tienen varias unidades funcionales idénticas. Estas unidades funcionales reciben de la unidad vectorial los datos y la instrucción para procesarse simultáneamente. La unidad vectorial también debe recolectar los resultados y almacenarlos en una memoria central.

Ejemplos de SIMD son la CM-200 del tipo arreglos de procesadores [63] y la CDC Cyber 205 del tipo vectorial [4].

Aunque estas computadoras tienen sincronización implícita y pueden procesar al mismo tiempo una instrucción sobre muchos datos, su uso se ve limitado a cierto tipo de aplicaciones especializadas como la suma de matrices o el procesamiento de imágenes, entre otras.

Las computadoras MIMD manejan más de un flujo de instrucciones a la vez, operando sobre diferentes datos. Esta característica hace que su uso sea más general a diferencia de las SIMD. Otra diferencia con las SIMD es la sincronización o comunicación entre procesadores. En las MIMD la sincronización es necesaria debido a que se tienen varios flujos de instrucciones de manera simultánea, los cuales necesitan comunicarse.

Las MIMD, se clasifican en computadoras de memoria compartida y de memoria distribuida. A las de memoria compartida se les conoce como *multiprocesadores*, mientras que a las de memoria distribuida se les conoce como *multicomputadoras* (supercomputadoras, clusters y grids). Ambas computadoras se presentan a continuación.

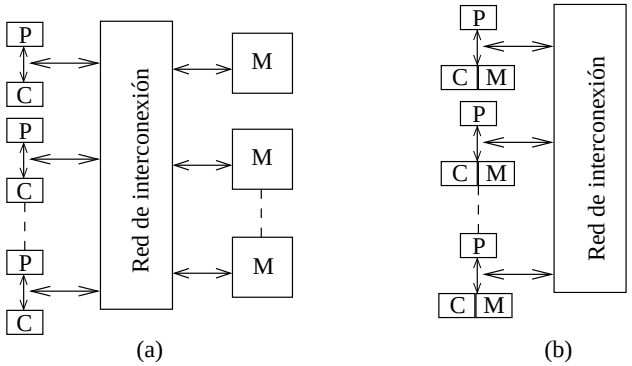

Figura 2.2: Arquitecturas para en un sistema multiprocesador: (a)UMA y (b)NUMA.

Multiprocesadores

Un sistema multiprocesador se forma por varios procesadores de propósito general, los cuales comparten un solo espacio de direcciones. Usando este espacio, los procesadores se comunican mediante la escritura y lectura de datos e instrucciones.

Entre los sistemas multiprocesador existen dos arquitecturas, la UMA del inglés (Uniform Memory Access) y la NUMA (Non-Uniform Memory Access).

En la arquitectura UMA, (Figura 2.2(a)), todos los procesadores (P) tienen relativamente los mismos tiempos de acceso a los módulos de memoria (M). Esta característica permite que los procesadores lean o escriban en cualquier dirección teniendo el mismo rendimiento. Por esta razón también se les conoce como SMP (Symmetric Multi-Processing)

En las UMA, los procesadores y los módulos de memoria se pueden conectar mediante una red de interconexión que puede ser un concentrador o un conmutador [31]. A través del concentrador se pueden agregar fácilmente más procesadores o módulos al sistema. Sin embargo, esto también conduce a un cuello de botella, ya que el concentrador sólo soporta a la vez una conexión entre procesadores y un módulo de memoria. Usando conmutadores, se pueden soportar más conexiones simultáneas, aunque de manera limitada (4 o 5), además de que el concentrador es costoso al ir aumentando el número de conexiones. Es por esta razón que las UMA no son escalables. y manejan un

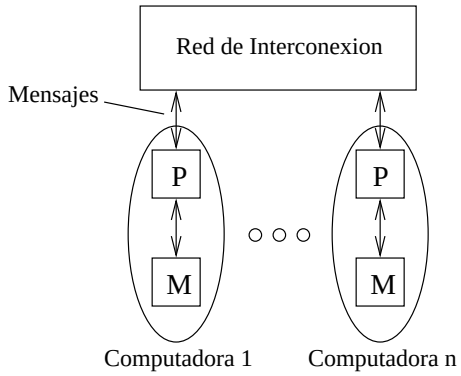

Figura 2.3: Sistema Multicomputador.

orden a lo más de cientos de procesadores. Algunos ejemplos de UMA son la Pentium Quad Pack y la Sun Enterprise Server [31, 95]

Una alternativa que mejora la escalabilidad de las UMA son las arquitecturas NUMA, como la que se muestra en la Figura 2.2(b). En las NUMA los módulos de memoria pueden ser locales o remotos en relación a un procesador, y así también los accesos a los datos. Si el acceso es local, se tiene un canal directo entre procesador y memoria haciéndolo más rápido; mientras que en los accesos remotos, se tiene una red de interconexión (concentrador o conmutador) en donde es más lento el acceso.

Normalmente, esta arquitectura descongestiona la red de interconexión, ya que muchos de los accesos son de manera local, haciendo posible escalar el número de procesadores. Sin embargo, existe el problema de la coherencia del cache (C). En la coherencia del cache, si un procesador cambia un dato en una memoria local, tiene que actualizar su propio cache y las posibles copias que existan. Sistemas como la Cray T3E [31] hacen un manejo automático por hardware de los accesos locales y remotos. Sin embargo, no guardan en cache las lecturas remotas, precisamente para evitar los mecanismos de coherencia. En la SGI-Origin [83], el cache guarda los datos leídos de memorias remotas y por un hardware complejo mantiene la coherencia.

Multicomputadoras

Un sistema multicomputador, Figura 2.3, se forma por varias computadoras o nodos. Cada computadora consiste de un procesador (P) y su propia memoria (M).

Las multicomputadoras son parecidas a las NUMA, en el sentido que cada procesador tiene una memoria local. Sin embargo, a diferencia de las NUMA, las multicomputadoras no comparten el mismo espacio de direcciones. Cada computadora tiene su propio espacio de direcciones independiente de los demás.

Las computadoras se comunican por medio de mensajes. Los datos pueden contener resultados parciales en una aplicación o servir para sincronizar algunos nodos.

Todo el mecanismo de comunicación entre los nodos, es hecho por hardware especializado para esa multicomputadora. Este hardware especializado permite que las comunicaciones sean rápidas; sin embargo, debido a que es sólo útil para cierto tipo de multicomputadoras este hardware es muy costoso.

En cuanto a la escalabilidad, las multicomputadoras si son escalables, ya que basta agregar nuevos nodos para incrementar los procesadores y por consiguiente el rendimiento. En cuanto a los cuellos de botella para comunicar procesador-memoria, estos no se dan, ya que cada nodo tiene integrado su propio procesador y memoria. Algunos ejemplos de multicomputadoras son: la IBM SP2 y la Intel Paragon XP/S [31, 95]. La IBM SP2 se forma de varios sistemas RS6000 (Symmetric Multiprocessors) en varios gabinetes, los cuales se conectan en varias topologías por hardware de comunicación especializado. La Intel Paragon es un solo gabinete con procesadores i860 conectados en forma de malla (toro) mediante procesadores especializados de comunicación.

Una multicomputadora con componentes más comerciales (menos específicos) son los clusters.

PC Clusters

Los PC clusters (o clusters simplemente), al igual que las multicomputadoras, son computadoras o nodos conectados a través de una red. No obstante, las computadoras

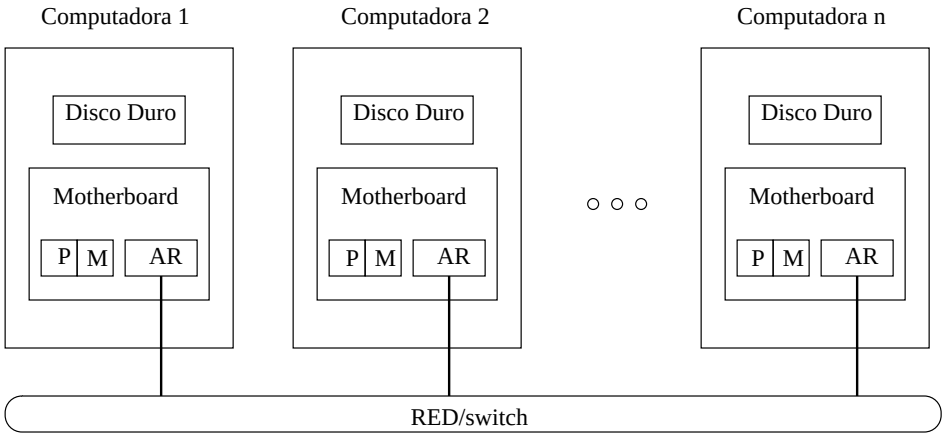

Figura 2.4: Arquitectura de un Cluster.

de un cluster no son necesariamente especializadas como las de las multicomputadoras. En los clusters (Figura 2.4), cada computadora esta formada por una tarjeta madre, procesador (P), memoria (M), adaptador de red (AR), y demás componentes como el disco duro. Las computadoras se conectan mediante un concentrador o un conmutador.

La siguiente lista da algunas de las razones por las que los clusters son preferidos sobre computadoras paralelas especializadas.

- Las computadoras personales se vuelven cada vez más potentes. El rendimiento ha sido incrementado en los últimos años y es doblado cada 18 o 24 meses.
- El ancho de banda, es decir la velocidad de transferencia de los conmutadores se ha incrementado por las nuevas tecnologías y protocolos implementados.
- Se pueden agregar fácilmente más computadoras al cluster. Generalmente basta conectar la computadora al conmutador o concentrador.
- Desarrollo de muchas herramientas de software para clusters, en comparación con las computadoras paralelas especializadas.
- Los clusters son mucho más baratos en comparación con las multicomputadoras o los multiprocesadores, debido a que todos sus componentes son de propósito general.

- Las computadoras del cluster se pueden fácilmente actualizar, agregando memoria o procesadores adicionales.

Los clusters son clasificados en muchas categorías, basados en los siguientes factores [20]:

- Tipo de nodos: clusters de PC's, estaciones de trabajo (workstations) o SMP's (Symmetric Multiprocessors).

- Redes y mecanismos de interconexión: Fast Ethernet, Gigabit Ethernet, Myrinet e Infiniband. Fast Ethernet y Gigabit Ethernet son ampliamente usados con velocidades de 100 y 1000 Megabits por segundo. Myrinet es una conexión de 1.28 Gbps ofrecida por Myricom [13]. Infiniband es uno de los mecanismos de interconexión más rápidos de la actualidad (10 Gbps) [81].

- Sistema operativo: Linux, Solaris de Sun®, NT de Microsoft®, AIX de IBM® y HP-UX de HP® entre otros.

- Configuración de componentes de hardware: homogéneos y heterogéneos. Los homogéneos tienen arquitectura similar y corren el mismo sistema operativo, los heterogéneos no.

- Uso de los nodos: dedicado y no-dedicado. Un cluster es dedicado si solamente un usuario lo utiliza a la vez; y es no-dedicado si lo utiliza más de un usuario.

- Tamaño del cluster: Grupo, departamentales, organización, nacionales e internacionales. Si se unen varios clusters, se puede tener un Grid [73].

- Tipo de aplicaciones: alto desempeño y alta confiabilidad. En una aplicación de alto desempeño el tiempo de respuesta es importante, por el contrario si es de alta confiabilidad no importa el tiempo de respuesta sino que no falle el procesamiento sobre la aplicación.

2.1.2. Software especializado

Hasta ahora se ha definido la infraestructura o el hardware necesario para hacer cómputo paralelo. Como se mencionó anteriormente, los clusters son los que ofrecen una excelente relación costo-rendimiento. Sin embargo, esto no es suficiente para hacer cómputo paralelo, es necesario cierto software (programas) como un sistema operativo que pueda manejar la computadora paralela y alguna metodología o modelo para desarrollar las aplicaciones (programación paralela).

En cuanto a los sistemas operativos, es común que la arquitectura paralela tenga su propio sistema operativo como Irix [83], Solaris [96] o AIX [65], y ofrezcan sus propios programas para desarrollo. Algunos de estos programas son buenos, pero la restricción es que sólo pueden ser usados si se adquiere la costosa computadora paralela que ellos venden. Si estos programas se logran instalar en arquitecturas tipos cluster, estos no son compatibles o son poco eficientes.

La otra opción es usar algún sistema operativo tipo Linux. Los sistemas operativos Linux son de libre distribución y gratuitos. Algunos ejemplos de estos sistemas son: Suse [64], Debian [50] y Fedora [53], entre otros. Por ser de uso general, estos sistemas operativos resultan ser eficientes para máquinas paralelas como lo son los clusters (recientemente muchas de las costosas máquinas paralelas comienzan a dar la opción de usar sistemas operativos Linux). Adicionalmente existen compiladores y paquetería que facilitan el desarrollo de aplicaciones. Sin embargo, hay que tomar muchos factores en cuenta, pues aún teniendo estas herramientas es necesario un modelo de programación que permita desarrollar las aplicaciones.

2.2. Programación paralela

Para programar en paralelo es necesario seguir varios pasos o etapas, como se muestra en la Figura 2.5. Estas etapas son: el particionamiento, la distribución y la comunicación.

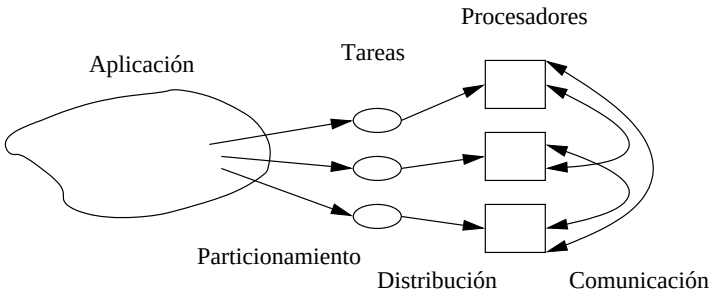

Figura 2.5: Paralelización de una aplicación.

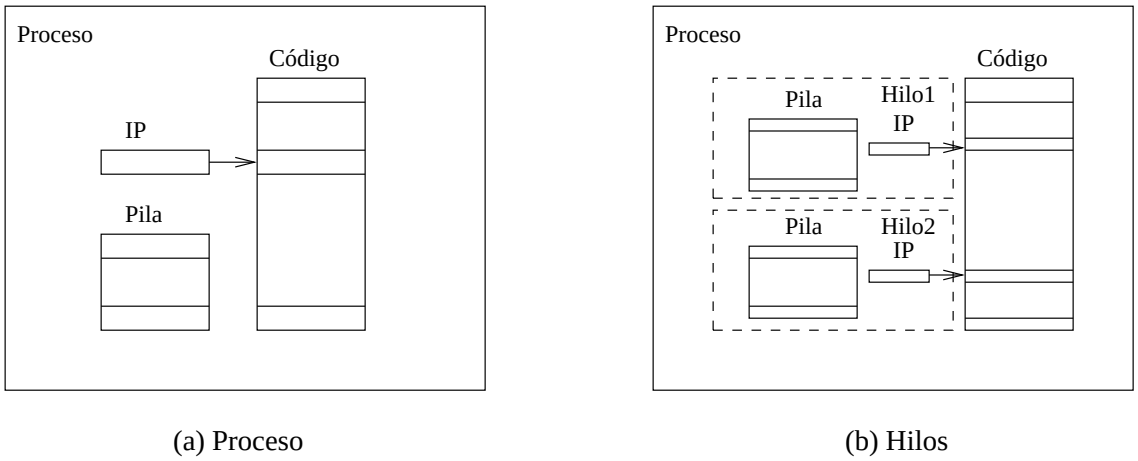

Figura 2.6: (a) Estructura básica de un proceso. (b) Estructura de dos hilos.

Para que una aplicación pueda ser ejecutada en paralelo necesita ser dividida o *particionada* en diferentes tareas. Al tener más de una tarea, estas pueden ser procesadas simultáneamente. Para lograr esto, las tareas se *distribuyen* entre los procesadores para ser ejecutadas. Normalmente, durante su ejecución las tareas necesitan *comunicarse* entre ellas, ya sea para sincronizarse o para intercambiar algún resultado parcial o final.

Las tareas distribuidas entre los procesadores, en realidad pueden implementarse en forma de procesos o hilos. A continuación damos una breve descripción de la implementación de tareas mediante procesos e hilos, después de la cual daremos pie a los modelos de programación paralela.

Procesos

Un proceso es la instancia ejecutable de un programa. Las partes básicas de un proceso pueden verse en la Figura 2.6(a). Una parte es el apuntador de instrucción (IP, *Instruction Pointer*), cuya función es mantener la dirección de la siguiente instrucción a ejecutar. Otra parte es la pila, la cual sirve para no perder la continuidad en el llamado a procedimientos. Cada vez que se crea un proceso, se establece un nuevo código, apuntador de instrucción y pila.

La creación básica de un proceso, se hace mediante una llamada al sistema operativo (como *clone* y *fork* [1]). La llamada al sistema *fork()* crea un nuevo proceso. El nuevo proceso, también llamado proceso *hijo* es una copia del proceso *padre* pero con un identificador (*id*) de proceso único y diferente. En cuanto a las variables, el proceso *hijo* tiene una copia de las variables del proceso *padre* las cuales puede manipular independientemente (en *clone* no son independientes, son las mismas). En caso de éxito, el llamado *fork()* devuelve un 0 al proceso *hijo* y el identificador del *hijo* al proceso *padre*. En caso de falla, devuelve -1 al proceso *padre*, y el proceso *hijo* no es creado. Ambos procesos son unidos o sincronizados mediante las llamadas *wait()*, *waitpid()* y *exit()*.

El siguiente pseudocódigo muestra un sencillo ejemplo de la creación de procesos, más detalles se pueden encontrar en Rochkind[77].

```
pid=fork()
if(pid==0)
   // código a ser ejecutado por el proceso hijo
else
   // código a ser ejecutado por el proceso padre
if (pid==0)
  exit(0);      // el hijo termina
else
  wait(0);     // el padre espera a que termine el hijo
```

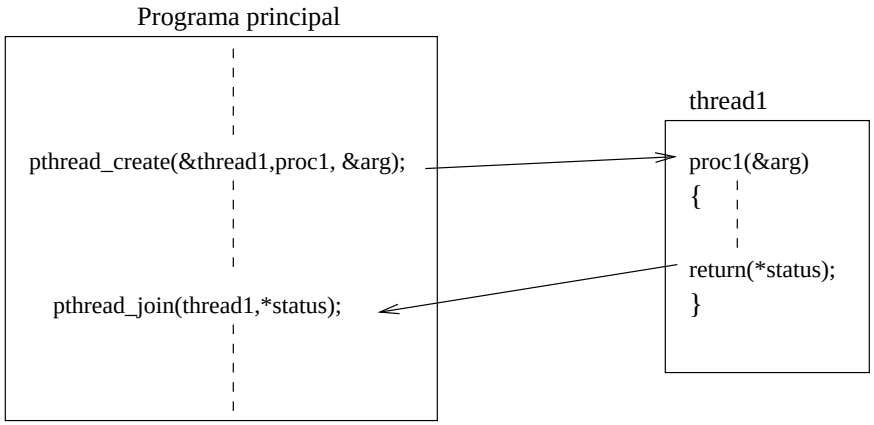

Figura 2.7: Bifurcación y unión de hilos POSIX.

Hilos

Un hilo tiene generalmente la misma función que un proceso, ejecutar instrucciones (una tarea). No obstante, la creación de más de un hilo es diferente a la creación de más de un proceso. En la Figura 2.6(b) podemos ver la estructura de dos hilos. Al igual que un proceso un hilo tiene un apuntador de instrucción y una pila. Sin embargo, para cada creación de un hilo, sólo se crea un nuevo apuntador de instrucción junto con su pila. El nuevo apuntador de instrucción indica otra secuencia de ejecución dentro del mismo código los cuales comparten la misma memoria o área de datos. Cabe mencionar que se puede crear más de un hilo en cada proceso.

Inicialmente el manejo de hilos se hacía dependiendo de la arquitectura en hardware y del sistema operativo. Sistemas operativos como SUN Solaris y Windows NT ofrecían un manejo de hilos a sus usuarios, pero no eran compatibles entre ellos. Afortunadamente, ya existe un estándar IEEE para hilos, los PThreads o hilos POSIX (Portable Operating System Interface).

En la Figura 2.7, se muestra la creación de hilos POSIX mediante el llamado *pthread_create*. Con el llamado se crea el hilo *thread1*, el cual ejecuta el procedimiento *proc1*. A este procedimiento se le pueden pasar parámetros.

Una vez que se ha creado el hilo *thread1*, éste se ejecuta de manera concurrente con el programa principal (es posible la creación de más hilos). Para lograr una sincronización,

el programa principal hace el llamado *pthread_join*. Este llamado espera a que el *thread1* termine su ejecución para poder continuar. Más detalle de la programación con hilos se puede encontrar en Nichols [70].

Una vez que se ha visto como se modelan las tareas por medio de procesos e hilos, presentamos los modelos básicos de programación paralela.

2.2.1. Modelo de programación de paso de mensajes

El modelo de paso de mensajes se puede comparar al correo y al teléfono. Con el correo, las personas se pueden comunicar mediante el envío y recepción de cartas. Para lograr esto es necesario saber hacia dónde va dirigida la carta. Por otro lado, con el teléfono las personas se comunican marcando el número de la persona deseada.

En el modelo de paso de mensajes, un proceso *fuente* (emisor) inicia la comunicación enviando un mensaje (datos) a un proceso *destino* (receptor). Para lograr esto el proceso fuente necesita conocer el identificador (*id*) del proceso destino.

El envío y recepción de mensajes puede ser de dos tipos: síncrono (teléfono) y asíncrono (correo). El envío y recepción de mensajes síncrono necesita que los procesos comunicantes (fuente y destino) se pongan de acuerdo. En este caso, el proceso fuente envía el mensaje al proceso destino y no puede continuar hasta que éste lo recibe. Esta comunicación síncrona es semejante al proceso de hacer una llamada telefónica. En este caso, la persona que llama tiene que esperar hasta que la persona conteste para poder enviar el mensaje (comenzar a hablar).

El envío y recepción asíncrono no necesita que los procesos que se comunican se pongan de acuerdo para hacer la transmisión. Es decir, el proceso fuente envía el mensaje y continúa su procesamiento, sin esperar a que el proceso destino reciba el mensaje. En sí, al igual que el sistema de servicio postal, el mensaje llega a un buffer o área de memoria del proceso destino (buzón) donde permanece hasta que el destinatario lo lee.

En paso de mensajes cada proceso tiene sus propios datos y éstos no son modificados por un mensaje. Sin embargo, con el paso de mensajes es muy importante el orden y

destino de los mensajes, ya que un envío sin recepción, una recepción sin envío o un envío a un destinatario incorrecto, puede provocar que el programa se bloquee o devuelva un mal resultado.

A nivel de proceso, los mensajes pueden ser enviados por medio de *sockets* [77]. Los sockets establecen una comunicación síncrona mediante el protocolo *TCP* y asíncrona mediante el protocolo *UDP*. En el protocolo TCP se establece una ruta de conexión similar a una llamada telefónica, donde el emisor marca un número destino y la comunicación se establece hasta que el receptor contesta la llamada. En UDP, no se establece una ruta de comunicación, por lo que los paquetes pueden tomar diferentes rutas y llegar desordenados.

No obstante que los sockets implementan el paso de mensajes, su manejo es un tanto complicado. Es necesario seguir una serie de pasos tanto en el proceso emisor como en el receptor mediante las llamadas *socket()*, *bind()*, *listen()*, *connect()*, *accept()*, *send(),* y *receive()* entre otras.

El siguiente código muestra un programa cliente que se comunica con un programa servidor (no ilustrado) mediante sockets usando el protocolo de comunicación TCP.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3490                   // el número de puerto
#define MAXDATASIZE 100             // número máximo de datos en bytes

int main(int argc, char *argv[])
{
```

```
    int sockfd, numbytes;
    char buf[MAXDATASIZE];           // donde se almacena el texto recibido
    struct hostent *he;      // estructura que recibe información sobre el servidor
    struct sockaddr_in server;       // información sobre la dirección del servidor

    if ((he=gethostbyname(argv[1])) == NULL) {
       perror("gethostbyname");     // se obtiene la dirección IP del servidor
       exit(1);
    }
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
       perror("socket");              // se  crea un socket Internet-TCP
       exit(1);
    }
    // se llena una estructura para la conexion con el servidor
    server.sin_family = AF_INET;     // familia
    server.sin_port = htons(PORT);   // puerto
    server.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(server.sin_zero), '\0', 8);  // cero para el resto del struct
    if (connect(sockfd, (struct sockaddr *)&server, sizeof(struct sockaddr)) == -1) {
       perror("connect");             //se establece la conexión con el servidor
       exit(1);
    }
    if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
       perror("recv");       // se reciben numbytes datos del servidor, usando
       exit(1);              // el socket declarado y se guarda en el buffer
    }
    buf[numbytes] = '\0';
    printf("Received: %s",buf);  // se imprime el mensaje
    close(sockfd);               // se cierra el socket
    return 0;
}
```

2.2.2. Modelo de programación de memoria compartida

El modelo de memoria compartida se puede comparar con un tablero de anuncios. En el tablero varias personas se comunican mediante la colocación y lectura de anuncios. En la programación con memoria compartida varias tareas se ejecutan concurrentemente en los procesadores disponibles y se comunican entre ellas mediante la escritura y lectura en áreas compartidas de memoria. Usando procesos, éstos se pueden crear y se pueden comunicar a través de áreas de memoria compartida (tablero) previamente declaradas. Con hilos, sólo basta crearlos y comunicarlos a través de sus variables globales. No obstante que la creación y comunicación resulta sencilla, es necesario tomar en cuenta ciertos aspectos de sincronización. La sincronización es necesaria cuando más de un proceso o hilo hace lecturas y escrituras simultáneas sobre un dato compartido. Por ejemplo puede suceder que dos procesos ejecuten el siguiente código:

```
Proceso1              Proceso2
1. leer x             1. leer x
2. incrementar x      2. incrementar x
3. escribir x         3. escribir x
```

Si el valor inicial de x fuera 10, se esperaría que al final de la ejecución de los dos procesos, el valor de x fuera 12. Sin embargo, si los procesos se ejecutan simultáneamente, el valor de x sería al final 11. En estos casos existe una región *crítica* definida por los pasos 1, 2, y 3, la cual no debe ser ejecutada al mismo tiempo por más de un proceso (exclusión mutua).

En este caso, instrumentos como los candados o los semáforos controlan el acceso no simultáneo a la región crítica. Un candado o lock es una variable que permite o no el acceso a una región crítica. Internamente éstos se construyen mediante instrucciones *test&set* (TAS) o *compare&swap* (CAS). TAS es una instrucción atómica usada para escribir en una localidad de memoria. Atómica se refiere a que no puede ser interrumpida. Esta instrucción escribe un valor pero primero efectúa una prueba (*test*) donde verifica si el valor leído es el mismo que otro valor dado. Si es igual se escribe el valor

sino, la prueba falla. Si muchos procesadores pueden acceder a la misma memoria, y si un proceso esta actualmente ejecutando un test-and-set, ningún otro proceso puede comenzar otro TAS. La instrucción CAS funciona de manera similar pero es para 32 bits (TAS es de un bit). TAS y CAS llevados a candados funcionan así, si el valor de un candado es 1 se puede acceder a la región crítica, en caso contrario no se puede acceder y la tarea queda bloqueada hasta que dicho candado valga 1. Si se puede acceder, el valor de candado se cambia a 0 y se regresa a 1 cuando se sale de la región crítica. El código anterior con candados queda de la siguiente forma:

```
Proceso1                Proceso2
0. lock(y)              0. lock(y)
1. leer x               1. leer x
2. incrementar x        2. incrementar x
3. escribir x           3. escribir x
4. unlock(y)            4. unlock(y)
```

Internamente dentro de la primitiva lock existe un ciclo *while* el cual verifica (lee) constantemente el valor del candado, y sólo hasta que es 1 se sale del ciclo *while*. Esta operación tan costosa para el procesador se evita con los semáforos. Un semáforo tiene la misma función operativa que un candado, sin embargo, cada vez que un proceso no puede entrar a la región crítica, éste se bloquea (duerme). El desbloqueo (despertar) viene cuando el proceso que entró a la región crítica sale de ésta y despierta al proceso dormido. Las operaciones básicas con los semáforos son: *P(s)* y *V(s)*. *P* y *V* son las siglas de *Passeren* y *Vrigeven* en holandés que significa "pasar" y "liberar" respectivamente.

Para terminar esta subsección, presentamos un ejemplo de programación con memoria compartida usando hilos. En este ejemplo, se crean dos hilos, los cuales ejecutan de manera concurrente el procedimiento *proc1* y *proc2*. En ambos procedimientos se incrementa en uno la variable compartida *resultado*. Para tener un resultado correcto es necesario el uso de candados en la región crítica.

```
#include <stdio.h>
#include <pthread.h>
```

```
#define MAXHILOS 2

int resultado=0;             //variable global compartida por los dos hilos
pthread_mutex_t Vacia;       // variable candado necesaria para controlar el acceso
                             // a  una región critica
void proc1(void)  {                  // hilo 1
  int x;
 // candado que evita el acceso simultáneo a la región crítica (variable resultado)
  pthread_mutex_lock(&Vacia);
  x=resultado;                       // lectura de la variable compartida
  x=x+1;                             // incremento en uno
  resultado=x;                       // escritura de la variable compartida
  pthread_mutex_unlock(&Vacia);      // liberación del candado
  pthread_exit(0);
}

void proc2(void)  {                  // hilo 2
  int x;

  pthread_mutex_lock(&Vacia);
  x=resultado;                       // lectura de la variable compartida
  x=x+1;                             // incremento en uno
  resultado=x;                       // escritura de la variable compartida
  pthread_mutex_unlock(&Vacia);
  pthread_exit(0);
}

int main()  {
  pthread_t hilo[MAXHILOS];

  pthread_mutex_init(&Vacia,NULL);                    //valor inicial default = 1

  pthread_create(&hilo[0],NULL,(void *)&proc1,NULL);       //crea el hilo proc1
  pthread_create(&hilo[1],NULL,(void *)&proc2,NULL);       //crea el hilo proc2
```

```
  pthread_join(hilo[0],NULL);                //espera a que termine el hilo proc1
  pthread_join(hilo[1],NULL);                //espera a que termine el hilo proc2

  printf("EL valor de x=%d \n",resultado);
  return 0;
}
```

2.2.3. Modelos de programación paralela funcional y lógico

Estos modelos considerados de alto nivel describen a un programa como un conjunto de funciones (a diferencia de un lenguaje imperativo donde un programa se describe como una secuencia de acciones). En estos modelos las funciones reciben entradas y las mapean a salidas. Las entradas en forma de parámetros pueden ser enteros, reales u otras funciones.

Los modelos funcionales son naturalmente adecuados para la programación paralela, porque el programador no especifica el orden de ejecución de las operaciones, sólo define los argumentos de las funciones. Los argumentos pueden ser tareas (funciones) que se pueden ejecutar de manera independiente .

Sin embargo, esta identificación de tareas no siempre es suficiente para obtener un buen rendimiento. Frecuentemente se tienen tareas de grano fino, es decir tareas con poco cómputo y mucha comunicación. Estas tareas requieren canales de comunicación rápidos como en las máquinas de memoria compartida, donde cada tarea ejecuta una función, pero aún ahí la sobrecarga es significativa [60]. Es todo un reto identificar a las tareas paralelas con una granularidad razonable y con pocos requerimientos de comunicación.

Los modelos de programación lógica forman parte de una rama que ha sido desarrollada independientemente del paralelismo. Al igual que la programación funcional los modelos de programación lógica describen el *qué* es evaluado, mas no el *cómo*. Mientras que los lenguajes funcionales expresan un programa como un anidamiento de funciones,

los lenguajes lógicos expresan un programa como un conjunto de cláusulas lógicas.

En la programación lógica existen principalmente dos fuentes de paralelismo AND y OR. AND se refiere a resolver múltiples metas en paralelo. Una meta es exitosa si una submeta y todas las demás submetas ocurren. El paralelismo OR se refiere a investigar muchas cláusulas en paralelo. Una meta sucede si una u otras cláusulas suceden.

Existen tres tipos de clausulas: hechos, reglas y metas. Un ejemplo en Prolog los ilustra.

```
padre(susan,ben).                                    (1)
padre(ben,eve).                                      (2)
ancestro(X,Y) :- padre(X,Y).                         (3)
ancestro(X,Y) :- padre(X,Z), ancestro(Z,Y)           (4)
```

En el código las mayúsculas denotan variables y las otras constantes. El código expresa dos hechos (1,2) y dos reglas (3,4) cuya lectura es la siguiente:

```
1   susan es padre de ben
2   ben es padre de eve
3   cualquier X es un ancestro de Y si X es es un padre de Y
4   cualquier X es un ancestro de Y si hay una Z tal que X es un padre de Z y
    Z es un ancestro de Y
```

El programa comienza especificando una meta, por ejemplo *ancestro(susan,R)* y se evalúa de arriba hacia abajo y de izq a der. La cláusula 3 responde, colocando *susan* en X y R a Y. Ahora el lado derecho de la cláusula es la siguiente meta *padre(susan,R)*, ahora la clausula 1 responde y la respuesta final es $R = ben$.

Con el paralelismo AND las submetas de 4, *padre(X,Z)* y *ancestro(Z,Y)* pueden ser evaluadas en paralelo. Sin embargo el paralelismo no es conveniente ya que la cantidad de cómputo es demasiado pequeña para justificar la creación de un proceso para *padre(X,Z)* y por otro lado la existencia de la variable compartida Z forza al segundo proceso a esperar hasta que el primer proceso asigne la variable.

Como nos damos cuenta, los modelos funcionales y lógicos requieren mucha notación matemática con la cual muchos de los programadores o desarrolladores de aplicaciones no están familiarizados. Otra de las desventajas es que sólo obtienen buenos rendimientos en máquinas con memoria compartida. Finalmente, el rango de aplicaciones se limita principalmente a aplicaciones sencillas con patrones regulares en la comunicación con todos los datos conocidos de antemano.

2.2.4. Modelo de programación por skeletons

Los skeletons son funciones preconstruidas de algoritmos paralelos típicos (pipes, divide&conquer, geometric decomposition). La programación en base a skeletons consiste en usar las funciones preconstruidas necesarias para construir un programa paralelo.

El modelo de skeletons al igual que los lenguajes funcionales se encuentra o se define como un modelo de alto nivel (muchos skeletons se construyen a partir de lenguajes funcionales). Esto quiere decir que el desarrollador de aplicaciones se olvida de alguna forma del particionamiento, de la asignación y de la comunicación. Sin embargo, surgen otras situaciones como las que citamos a continuación.

El modelo de skeletons ha sido explorado ampliamente [29, 34, 33, 76]. Esto ha ocasionado que exista una gran diversidad de skeletons cada uno con diferentes nomenclaturas. Aunque se ha tratado de hacer una clasificación general, ésta no ha prosperado y sigue el problema de elegir a un grupo de skeletons o a varios para poder desarrollar aplicaciones [21].

Otra limitante en la programación con skeletons es que muchos están construidos en base a modelos funcionales. En sí, muchos skeletons no son excluyentes de los modelos funcionales por lo que para programar es necesario saber algún lenguaje funcional. Otra complicación es *¿cuáles y cuántos usar?*. Si los skeletons son del tipo general, es decir, cubren los aspectos básicos de la paralelización, se necesitan escoger muchos skeletons de este tipo para construir la aplicación final. Por otro lado, si son específicos, se necesitan menos skeletons pero este tipo de skeletons no puede cubrir a la mayoría

de las aplicaciones. El peor de los casos es que muchas veces las aplicaciones son tan diferentes y especializadas que termina implementándose un nuevo skeleton para esa aplicación.

Actualmente el modelo de skeletons ha cambiado de ser un lenguaje o modelo para ser una biblioteca de funciones que se puede usar como una extensión a un lenguaje como C.

Un ejemplo de un sólo skeleton, olvidándonos de inicializaciones y anidamientos es el skeleton DEAL que es como un repositorio donde se toman tareas. En este skeleton se necesitan definir 17 parámetros para hacer funcionar el skeleton, lo cual complica la programación.

```
DEAL(DEALWORKERS, IMPL, DealWorker, workercol, STRM, NULL,
        0,0,SPGLOBAL, MPI_INT, NULL, 0, &outmul, SPGLOBAL, MPI_INT,
        0,mycomm());
```

2.2.5. Modelo de programación por ADTs paralelos

Un *tipo de dato abstracto* (ADT Abstract Data Type) se puede definir como un modelo matemático con una serie de operaciones definidas en ese modelo. Las dos propiedades fundamentales que proporcionan los ADT son la generalización y la encapsulación. Los ADT son generalizaciones de los tipos de datos primitivos (enteros, caracteres, ...), al igual que las operaciones para operarlos, que son generalizaciones de operaciones primitivas (suma, resta, etcétera). Un ADT encapsula cierto tipo de datos, en el sentido de que es posible localizar la definición del tipo de datos y todas las operaciones con ese tipo en una sección del programa.

Entre los ADT más conocidos están las pilas, colas, árboles y listas. Las pilas tienen como operación básica el *push* y *pop*; las colas tienen las operaciones *enqueue* y *dequeue*; los árboles tienen *parent, brother, left, right, setleft* y *setright*; finalmente las listas tienen entre sus operaciones básicas *insert* y *get* [2, 52, 58]. Este concepto, aplicado ampliamente en la programación secuencial, ha sido extrapolado a la programación paralela.

En el modelo de programación paralela con ADTs, el usuario programa de la misma forma como lo haría tradicionalmente en la programación secuencial con ADTs. No obstante, las operaciones sobre los ADTs son paralelas, pero esto pasa desapercibido por el programador. Es decir, el programador sólo tiene que escoger algún ADT para representar sus datos y comenzar a utilizar las operaciones disponibles para ese ADT. Al ejecutar las operaciones del ADT éstas se encargan internamente de la distribución y sincronización de los datos en el sistema.

Un ejemplo hipotético puede ser el ingresar el *hostname* de todos los nodos de un cluster en una cola. Con el modelo de mensajes sería necesario hacer explícitamente el envío del nombre de todos los nodos hacia algún nodo que los reciba y los almacene en la cola (conforme van llegando). Con el modelo de ADT paralelos sólo se requiere elegir el ADT cola y la operación encolar, *enqueue(nombre,cola_x)*, la cual es llamada por todos los procesadores. Cabe resaltar que con el modelo ADT el programador no se tiene que preocupar por dónde esta la cola física ni tampoco en cómo se trasmite el dato de un procesador a otro para encolarse.

En el contexto paralelo los ADTs pueden ser compartidos o no-compartidos.

En el ADT *compartido* [45] se tiene una visión global del ADT, es decir, todos los procesadores ven el mismo ADT, a diferencia del ADT *no-compartido* donde los procesadores no tienen una vista global del ADT, sino más bien local.

En los ADTs compartidos, se tienen dos versiones: copia única y varias copias. En el de copia única todos los procesadores acceden (efectúan sus operaciones) sobre un único ADT, ubicado en un procesador. Para controlar el acceso entre los procesadores, se requiere de mecanismos de sincronización como candados. Esta versión de copia única tiene la desventaja de que los accesos a un ADT único son un cuello de botella que puede afectar el rendimiento. En la versión de varias copias, un ADT compartido se encuentra replicado en todos los procesadores. Cada vez que un procesador accede al ADT, lo hace en su copia local. Esta versión evita el cuello de botella que se tiene en la versión de copia única; sin embargo, requiere que las copias del ADT, distribuidas en los diferentes procesadores, se mantengan coherentes, es decir, cualquier acceso al ADT en

cualquier procesador, se debe reflejar en todas las copias en todos los procesadores. Por todo esto, el modelo de ADTs compartidos son más eficientes en máquinas de memoria compartida (multiprocesador) donde el costo implicado por la coherencia no es tan alto. Los ADTs compartidos no son eficientes cuando se utilizan sobre multicomputadoras o clusters con una gran cantidad de procesadores.

Por el otro lado, explorados en un menor grado están los ADTs no-compartidos. En los ADTs no-compartidos se tiene una visión local del ADT. En este caso todos los procesadores acceden a su propio ADT. Esto tiene la ventaja de que los accesos al ADT por parte un procesador son independientes de los demás accesos que se tengan en otros procesadores, evitando así la coherencia. Al principio un procesador puede contener todos los datos a ser procesados, pero estos son distribuidos en todos los procesadores a través de los accesos "locales" que se efectúan (por ejemplo con *get()*, *insert()*). Esto es transparente al programador al ser el procesamiento de los datos independientes y al no ser necesaria mantener ninguna copia coherente. Otra de las ventajas de este esquema distribuido es que el modelo puede ser escalable o puede manejar un número mayor de procesadores en comparación con los ADTs compartidos, donde la escalabilidad se ve afectada por la sincronización. Por otro lado, en el modelo no-compartido es necesario considerar primitivas que permitan hacer una recolección de los datos o resultados parciales que se encuentran distribuidos en la arquitectura paralela. Esta situación no es necesaria en los ADTs compartidos donde todos los datos y resultados pueden ser accesados en todo momento por todos los nodos de la arquitectura paralela.

2.3. Ejecución SPMD

En la programación paralela, normalmente todos los procesadores ejecutan el mismo código (programa) pero con diferentes datos. Precisamente en el modelo de ejecución SPMD (Single Program-Multiple Data)[47, 99], el programador escribe un único programa, el cual al momento de la ejecución es copiado y ejecutado en cada uno de los procesadores.

Esto tiene la ventaja de que no es necesario escribir un código para cada procesador. En el caso que se requiera que uno o varios procesadores efectúen una tarea distinta, se puede usar autoidentificación que defina que hacer o no.

Un ejemplo de un programa SPMD que considera un procesador maestro y todos los demás esclavos es:

```
...
id=mi_id()     //donde id va desde 0 hasta #procesadores-1
if (id=0)  {
    // código del procesador maestro
  }
else  {
    // código ejecutado por todos los procesadores esclavos (#procesadores-1)
  }
...
```

Esta filosofía de ejecución puede ser utilizada en cualquiera de los modelos de programación descritos con anterioridad.

2.4. Resumen

En este capítulo se ha hecho notar la gran demanda de cómputo de muchas aplicaciones, científicas y comerciales. Algunas de estas tardarían mucho tiempo (días, meses, años) en poder completarse. En estos casos el cómputo paralelo es una alternativa eficiente para reducir los tiempos de respuesta.

Sin embargo, el cómputo paralelo requiere más desarrollo tecnológico en comparación con el computo secuencial, principalmente en cuanto al hardware (arquitectura paralela) y al software (modelos de programación).

Las máquinas paralelas pueden ser desde multiprocesador hasta multicomputadora. Sin embargo, estas máquinas resultan ser inaccesibles por su altísimo costo. Una alternativa de menor precio son los clusters, donde la máquina paralela se construye

con varias computadoras de uso común y se conectan a través de un conmutador o concentrador.

En cuanto al software, este va desde algún sistema operativo como puede ser Linux hasta lenguajes y compiladores como C. Aún con todo lo anterior, se necesita seguir alguna técnica o modelo para construir los programas paralelos.

Para construir un programa paralelo es necesario descomponer el problema en varias tareas las cuales se asignan y ejecutan en los procesadores. Durante la ejecución estas tareas se comunican y sincronizan.

Intentando tener una organización sobre estos pasos, se han propuesto varios modelos de programación, entre los principales (básicos) tenemos el de paso de mensajes y el de memoria compartida.

En el modelo de paso de mensajes se tienen varios procesos que se comunican mediante el envío y recepción de mensajes. Usando este envío/recepción se lleva también a cabo la sincronización de procesos. Este modelo resulta ser eficiente en multicomputadoras o clusters debido a que el modelo de paso de mensajes plantea una memoria distribuida, al igual que en las multicomputadoras o clusters.

En el de memoria compartida varios procesos comparten un área de datos que procesar. El control sobre esa área de datos es mediante mecanismos de sincronización como candados, semáforos, variables de condición, etc. La comunicación entre los procesos es a través de los accesos a los datos (lectura-escritura).

Los dos modelos anteriores son básicos y pueden llegar a ser muy eficientes, ya que la programación se realiza paso a paso. Sin embargo, para muchos de los programadores puede resultar ser complicado el efectuar o definir todos los pasos de la programación de manera explícita. Por eso se han propuesto modelos de más alto nivel como los modelos funcionales/lógicos y los skeletons.

Los modelos funcionales y lógicos, utilizan el manejo de funciones o predicados para expresar paralelismo. Estos modelos ocultan totalmente los pasos del paralelismo, en aplicaciones sencillas. Sin embargo, para aplicaciones más complicadas como las de patrones de comunicación irregulares o volumen de datos dinámico la programación

se complica. En sí, para muchos de los programadores resulta difícil el manejo de la notación matemática que involucran estos modelos.

Los skeletons pueden verse como una caja de herramientas paralelas. Los skeletons ofrecen funciones que cumplen con algún patrón conocido en paralelismo (distribución, pipes, dividir para vencer, maestro-esclavo entre otros) o algún algoritmo de balance de datos. También existen skeletons que resuelven problemas de optimización por algún método como recocido simulado o algoritmos genéticos.

Este modelo oculta en cierta forma la notación matemática de los modelos funcionales/lógicos no obstante que muchos de los skeletons se construyen a partir de lenguajes funcionales/lógicos. Una de las complicaciones es la gran diversidad de skeletons que existe, y hasta el momento no se ha logrado una especificación estándar. Además, el tener un conjunto de skeletons no asegura una buena programación ya que ésta depende de la buena elección de los skeletons disponibles y el orden en que se acomoden. Existe una complicada relación entre la generalidad o granularidad de los skeletons. Un conjunto general de skeletons pueden abarcar más aplicaciones, sin embargo, se necesita un mayor volumen de estos. Por el otro lado se pueden tener skeletons mas especializados para aplicaciones específicas, en esos casos la programación se facilita. Sin embargo, existe una amplia variedad de aplicaciones, que muchas veces se termina haciendo un nuevo skeleton por cada aplicación.

El modelo de ADT (Abstract Data Types) consiste en utilizar tipos de datos abstractos como pilas, colas, árboles, listas, con sus correspondientes operaciones (push, pop, enqueue, dequeue, insert, get) los cuales funcionan de manera paralela. Los llamados a las operaciones aparentan ser de manera secuencial, pero en realidad son paralelos. Todo esto de manera transparente al programador. Existen ADTs compartidos y no-compartidos. En los ADTs compartidos se tiene un tipo de dato abstracto global, es decir, todos los procesadores tienen la misma visión del ADT. Físicamente el ADT compartido puede encontrarse en un sólo procesador (un sólo original) o en varios procesadores (varias copias). En el primer caso se tiene el inconveniente de que el acceso al ADT por parte de los procesadores provoca un cuello de botella; en el segundo ca-

so es necesario el mantener la coherencia de las copias a través de los accesos en los procesadores.

En los ADTs no-compartidos, los datos del ADT se encuentran distribuido en todos los procesadores con la característica de que los datos son diferentes y se pueden procesar de manera independiente. Ésto permite que no exista una fuerte sincronización y que se tenga una buena escalabilidad al utilizar una mayor cantidad de procesadores. Para programar, es necesario que el programador organice sus datos en el ADT no-compartido y comience a utilizar sus operaciones. Debido a que el procesamiento se encuentra distribuido, al final es necesario hacer una recolección de los resultados parciales para obtener el resultado final. El modelo ADT no-compartido distribuido es eficiente en arquitecturas como los clusters, multicomputadora y multiprocesador

Finalmente, vimos el modelo de ejecución SPMD, que es una alternativa sencilla de programación, en el que un sólo programa es necesario para la ejecución paralela. El modelo de ejecución SPMD puede ser utilizado en cualquiera de los modelos de programación anteriores.

Capítulo 3

Ambientes de Programación Paralela

En el capítulo anterior, presentamos las ventajas y limitaciones de los principales modelos de programación paralela. En este capítulo, se presentan algunos ambientes de programación que los soportan. Estos ambientes de programación se describen a nivel de uso e implementación.

En la Tabla 3.1 se muestran los ambientes de programación que describimos en este capítulo, así como el modelo de programación al cual pertenecen. Se escogieron estos ambientes por ser de los más representativos y usados en cada modelo.

Ambientes de programación	Modelos de programación
PVM, MPI [43]	Paso de mensajes
TreadMarks [6], Linda [8, 24]	Memoria compartida
Haskell [89]	Funcional
Prolog	Lógico
MALLBA, eSkel [10] y SAMBA [72]	Skeletons
SADTs [45]	ADTs

Tabla 3.1: Ambientes de programación y su modelo de programación.

3.1. PVM y MPI

PVM (Parallel Virtual Machine) y MPI (Message Passing Interface) [43, 86, 87], son ambientes de programación que siguen el modelo de paso de mensajes. Tanto PVM como MPI permiten la creación procesos, los cuales se comunican mediante el envío y recepción de mensajes. Esta comunicación se logra mediante la identificación de los procesos involucrados.

Ambos ambientes son considerados de bajo nivel porque requieren una programación o construcción de programas en todos los pasos (particionamiento, asignación, comunicación, sincronización). No obstante, esta dificultad en la programación permite que se pueda resolver aplicaciones de manera eficiente, en el sentido de que se hacen programas a la medida de las aplicaciones.

Para construir un programa con PVM o MPI es necesario hacer un particionamiento de los datos de la aplicación, crear los procesos necesarios con los datos a procesar y comunicarlos y sincronizarlos mediante el paso de mensajes.

Para facilitar en cierto grado la programación, la interfaz de PVM/MPI se presenta como una biblioteca de funciones adicional a ciertos lenguajes como C y C++ [55], Fortran [42] o Java [22]. Esto tiene la ventaja de que no es necesario aprender un nuevo lenguaje, sino solamente conocer y utilizar la sintaxis y los parámetros de las funciones ofrecidas.

Para conocer un poco más el uso de estos ambientes, consideremos una aplicación sencilla como la suma paralela de n números con p procesadores (Figura 3.1).

Usando el ambiente de programación MPI, la suma de n números se puede resolver mediante estos pasos:

1. Inicialización de datos.

2. Hacer un particionamiento del total de los n números entre el número p de procesadores.

3. Crear y asignar 1 proceso con sus n/p datos en cada procesador.

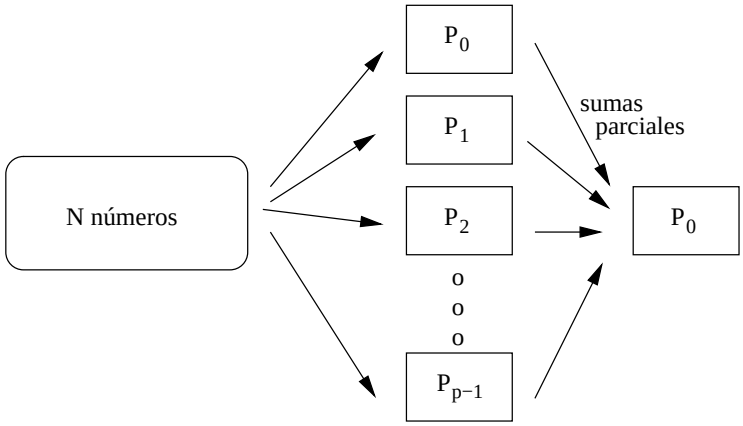

Figura 3.1: Paralelización de la suma de n números.

4. Obtener las sumas parciales en cada procesador y

5. Enviar los resultados parciales a un procesador particular para calcular la suma final.

Con un poco más de detalle, estos pasos se extienden en el siguiente código (MPI-C) con un modelo SPMD.

```
1.  #include <stdio.h>
2.  #include "mpi.h"
3.  #define N 100
4.  #define DATOS 100
5.  #define SUMA_P 101
6.
7.  Inicializa(int n_datos[])  {
8.    int i;
9.    for (i=0;i<N;i++)
10.     n_datos[i]=2;
11. }
12.
13. main(int argc, char *argv[]) {
14.   int n_datos[N], datos_parciales[10], suma_par=0,suma_total=0,i;
15.   int id, p;
```

```
16.   MPI_Status stat;
17.
18.   MPI_Init(&argc, &argv);
19.   MPI_Comm_rank(MPI_COMM_WORLD,&id);
20.   MPI_Comm_size(MPI_COMM_WORLD,&p);
21.
22.   if (id==0) {            // el proceso 0 inicializa los datos y los reparte
23.     Inicializa(n_datos);
24.     for(i=0;i<p;i++)
25.       MPI_Send((n_datos+(i*N/p)),N/p,MPI_INT,i,DATOS,MPI_COMM_WORLD);
26.     }
27.
28.   MPI_Recv(&datos_parciales,N/p,MPI_INT,0,DATOS,MPI_COMM_WORLD,&stat);
29.   for(i=0;i<N/p;i++)      // los procesadores reciben los datos, hacen la suma
30.     suma_par=suma_par+datos_parciales[i];  // parcial y reenvian al procesador 0
31.   MPI_Send(&suma_par,1,MPI_INT,0,SUMA_P,MPI_COMM_WORLD);
32.
33.   if(id==0)  {            // el procesador 0 recibe, hace la suma final e imprime
34.     for(i=0;i<p;i++)  {
35.       MPI_Recv(&suma_par,1,MPI_INT,MPI_ANY_SOURCE,SUMA_P,MPI_COMM_WORLD,&stat);
36.       suma_total=suma_total+suma_par;
37.     }
38.     printf("RESULTADO es %d \n",suma_total);
39.   }
40.   MPI_Finalize();
41. }
```

Con el modelo SPMD se crea directamente 1 proceso por procesador, y todos los procesos ejecutan el código anterior. En las líneas 1-5 se definen cabeceras y constantes. De la línea 7-11 se define la función que inicializa el arreglo de los números a sumar. En la interfaz de MPI es necesaria una inicialización (línea 18-20). La línea 19 en especial obtiene un identificador (id) único en la ejecución para cada proceso. El id va desde $0 < id < p - 1$, donde p representa el número total de procesos (obtenido en la línea 20). Esta identificación sirve en las comunicaciones, tanto en el envío (línea 25 y 31)

como en la recepción (línea 28 y 35).

En las línea 24-26 el proceso *0* hace el *particionamiento* de los datos y los *asigna* (envía) a los diferentes procesos. Después todos los procesos reciben la parte de datos que les corresponde (línea 28) para posteriormente obtener la suma parcial (línea 29-30). En la recepción está incluida la *sincronización* ya que el proceso no continúa hasta que reciba todos los datos.

Una vez obtenida la suma parcial, cada proceso envía sus resultados hacia el proceso cero (línea 31), quien recibe todos los resultados, efectúa la suma final e imprime el resultado (línea 33-39).

Se puede observar que en MPI es necesario hacer de manera explícita todos los pasos de la paralelización (particionamiento, asignación, comunicación y sincronización). Esto tiene dos consecuencias, una buena y una mala. La buena es que el control siempre lo tiene el programador, quien puede diseñar programas a la medida de las aplicaciones, obteniendo el mejor rendimiento. La mala es que muchas aplicaciones son tan complicadas (irregulares) que se puede dificultar su programación.

También es posible que en la programación se generen bloqueos o abrazos mortales (mensajes no enviados, respuestas sin recibir). Esto se da con más frecuencia en aplicaciones irregulares, pero también sucede en aplicaciones sencillas. En este caso, sólo el programador puede ir quitando los bloqueos mediante herramientas de depuración.

En la interfaz de MPI y PVM estos son manejados como bibliotecas adicionales a los lenguajes C, C++ y Fortran. También se manejan como un paquete adicional al lenguaje Java [22, 23]. En cuanto a la comunicación es completamente estándar, ya que utiliza sockets TCP/IP.

3.2. TreadMarks y Linda

TreadMarks es un ambiente de programación paralela basado en el modelo de memoria compartida. En sí, TreadMarks permite que el modelo de memoria compartida se implemente sobre un sistema de memoria distribuida ofreciéndose como un DSM

(Distributed Shared Memory). Con TreadMarks todos los procesos ven un solo espacio de direcciones compartido y no un espacio de direcciones distribuido.

Por el modelo de memoria compartida, no es necesario saber *cuándo* un procesador se necesita comunicar, o con *quién* comunicarse. La programación se facilita ya que la comunicación entre procesos se da directamente mediante la escritura y lectura de datos compartidos.

En cuanto a la interfaz, algo muy importante es la declaración de las variables compartidas en el sistema distribuido. Para el manejo de secciones críticas se usan candados para evitar que más de un proceso entre a una sección crítica y provoque inconsistencias. Para manejar la sincronización en puntos específicos TreadMarks permite el manejo de barreras. La barrera es un mecanismo de sincronización el cual establece un punto donde la ejecución de un proceso se detiene hasta que todos los procesos de la aplicación lleguen a ese punto. El uso de ambos, candados y barreras se ejemplifican a continuación, en la suma paralela de números.

```
1.  include "TreadMarks.h"
2.  int *n_datos;
3.  int *suma_final;
4.  int suma_parcial=0, candado;
5.
6.  main () {
7.    Tmk_startup();
8.    if(Tmk_proc_id==0)  {
9.      n_datos=Tmk_malloc(N*sizeof(int))
10.     Inicializar (n_datos);
11.     suma_final=Tmk_malloc(sizeof(int));
12.   }
13.   candado=0;
14.   Tmk_barrier(0);
15.   cantidad= N / Tmk_procs;
16.   for(i=0;i<cantidad;i++)
17.     suma_parcial=suma_parcial+n_datos[(Tmk_proc_id*cantidad)+i];
18.   Tmk_lock_acquire(candado);
```

```
19.   *suma_final=*suma_final+suma_parcial;
20.   Tmk_lock_release(candado);
21.   Tmk_barrier(1);
22.   if (Tmk_proc_id==0)  IMPRIME(*suma final);
23.   Tmk_exit();
24. }
```

Del pseudocódigo se puede ver que con este ambiente se logra evitar la comunicación entre los procesadores debido al sistema DSM. No así el *particionamiento* y la *sincronización*.

De manera similar a PVM/MPI, TreadMarks se maneja como una biblioteca de funciones para C o Fortran (línea 1). También requiere una inicialización y una terminación (línea 7 y 21 respectivamente). Pero la parte fuerte de TreadMarks en comparación con PVM/MPI es que no hay asignación explícita de datos. Sin embargo, cada proceso debe saber *qué* datos le tocan en base a su identificador de proceso.

En TreadMarks, se sigue el modelo de ejecución SPMD, por lo que la identificación es necesaria cuando un solo proceso requiere hacer una operación especial, como por ejemplo, la inicialización de los datos. Para ello TreadMarks utiliza la palabra reservada *TMK_proc_id*, equivalente a la instrucción *MPI_Comm_rank* de MPI.

En la parte de inicialización un solo proceso (en este caso el 0) declara y reserva las áreas o variables compartidas para todo el sistema (línea 8-12). Para el caso de la suma paralela de los elementos de un arreglo, es necesario declarar un arreglo que contenga los datos (números) que se compartirán entre todos los procesadores y otra variable que contenga la suma final. En la línea 13 se inicializa la variable `candado`, para controlar el acceso a la región crítica que la suma final representa.

A continuación en la línea 14 se coloca una barrera, ésta sirve para que los demás procesos $1 \leq p \leq P-1$, no continúen hasta que se hayan inicializado los datos por el proceso 0. Debe notarse que la barrera es una forma de sincronización costosa, mucho más cuando existen muchos procesadores.

En la línea 15 se hace el particionamiento de los datos y se hace la suma parcial

de cada subarreglo dependiendo del *id* del proceso (líneas 16-17). Después se calcula la suma final, esto se hace sumando el resultado parcial en la variable `suma_final` la cual es compartida por todos los procesadores. Para evitar inconsistencias en la lectura y escritura es necesario colocar candados (línea 18 y 20) para que un solo proceso lea y actualice la suma final en base a su suma parcial. En la línea 21 nuevamente se coloca una barrera para asegurar que se sumen todas las sumas parciales antes de imprimir la suma total. Finalmente, en la línea 22 el proceso con *id* igual a cero imprime el resultado.

Como se ve, TreadMarks es un ambiente de programación paralela que evita la la comunicación explícita entre los procesadores, lo que facilita la programación. Sin embargo, requiere del manejo de identificadores para acciones especiales como la inicialización o la asignación de datos. En cuanto a eficiencia, ésta se puede ver afectada por la barreras globales y los candados (más aún cuando crece el número de procesadores). A este respecto otros ambientes han considerado el evitar el uso de candados de forma explícita mediante un proceso de precompilación [28, 71]. Otro sistema similar a TreadMarks es CRL [54] y otro algo diferente es Linda [8, 24].

Linda se define como un modelo de programación de tuplas. Donde las tuplas son leídas y escritas en un espacio de tuplas. Una tupla, se puede ver como una estructura de datos, por ejemplo:

$"jones", 32, true$

Define una tupla de tres elementos que consiste en un string, un entero y un booleano. Sobre esta tupla se pueden aplicar las operaciones in, out y read.

La operación *in* obtiene (elimina) un elemento del espacio de tuplas, por ejemplo la sentencia:

$in("jones", ?edad, ?estado\_civil)$

asigna las variables edad y estado_civil del registro jones. Es decir, el valor de edad y estado_civil después de la sentencia es 32 y true respectivamente.

La operación *out* de manera similar escribe en el espacio de tuplas y por último la operación *read* sólo lee la tupla pero no la elimina.

Algunas consideraciones son que si la tupla no existe, las operaciones *in* o *read* se bloquean. Por otro lado, pudiera ser que existan tuplas duplicadas, en cuyo caso las operaciones *in* y *read* obtienen una tupla aleatoria.

En base a estas operaciones de memoria asociativa (identificación por algún campo de la tupla) los procesos se pueden comunicar. Sin embargo, continúa el inconveniente de los sistemas compartidos, es este caso, el sistema de tuplas, el cual se encuentra centralizado y es un cuello de botella para los procesos. Pudiera ser que el espacio de tuplas se encontrara físicamente distribuido y los accesos serían locales, pero eso conduce a mantener la coherencia de los datos (tuplas). Por otro lado, debido al funcionamiento de las tuplas, si dos procesos acceden o eliminan una tupla, sólo uno la obtiene y el otro permanece bloqueado, lo que conduce a que frecuentemente se tengan abrazos mortales.

3.3. Haskell y Prolog

Haskell es un lenguaje de programación funcional de alto nivel, en el cual la programación se enfoca en el *qué* mientras que en el bajo nivel la programación se enfoca en el *cómo*.

Básicamente un programa funcional es una expresión principal la cual es evaluada como una función. Esto es muy parecido a las hojas de cálculo (como Excel), donde el valor de una celda está en función del valor de otras. Al programar en Excel, el desarrollador construye funciones en base a otras funciones o datos colocados en otras celdas, sin preocuparse por declarar tipos de datos o tener idea de donde se guardan estos en memoria. Otra situación es que el programador construye su función sin importarle el orden en que se evalúen el contenido de esta, a diferencia de una secuencia de comandos la cual sigue un estricto orden de ejecución (evaluación).

Por ejemplo, en la suma de n números, Haskell necesita las especificaciones de las funciones que darán el resultado. El código siguiente muestran un ejemplo de programación en Haskell para tal problema.

```
1.  suma []       =  0
```

```
2.  suma (x:xs) =  x + suma xs
```

Como es de esperarse, para un lenguaje funcional de alto nivel, el código es corto. Sin embargo, no siempre es comprensible a primera vista. Un nuevo lenguaje implica nuevas reglas, sintaxis, identificadores, etc. Esto se agrava si se cambia el concepto de una programación imperativa a una programación declarativa.

Describiendo el código anterior, la línea 1 indica que si se aplica la función *suma* sobre un arreglo vacío el resultado es cero. En la línea 2 la expresión de suma es sobre varios números, representados por x y el resto por xs, en ese caso la suma a calcular es x + *suma* xs. Si nos damos cuenta la evaluación se reduce a encontrar la *suma* xs, aplicando nuevamente la expresión de la línea 2 de manera recursiva hasta completar la suma de todos los números.

El código descrito con anterioridad se compila con GHC (Glasgow Haskell Compiler) [92] el cual genera código para entornos de hardware secuenciales. GHC se encarga de generar un código "ejecutable" que sigue más o menos la receta del párrafo anterior. Esto tiene la ventaja que el programador evita la generación de código, su función sólo se limita a la especificación. Sin embargo, la eficiencia del código va a depender totalmente de la efectividad del compilador GHC. Otra de las desventajas es el código generado por GHC no es un ejecutable puro, este requiere de un intérprete similar al Java Runtime Environment (JRE). Como es sabido, el uso de intérpretes ocasiona que no se tengan los mejores rendimientos en la ejecución de las aplicaciones.

Para entornos de hardware paralelo, Haskell dispone de GPH (Glasgow Parallel Haskell) [89, 91]. GPH es un ambiente de desarrollo funcional que extiende a Haskell en ambientes paralelos ayudándose internamente de ambientes como PVM para la comunicación. GPH es similar a Haskell en el sentido del modelo de programación funcional y en que utiliza a GHC como su compilador. No obstante, agrega dos nuevas primitivas o combinadores que ayudan en la especificación del paralelismo. Las primitivas son: *par* y *seq* [51]. La primitiva *par* especifica el paralelismo de dos funciones, de igual forma la primitiva *seq* indica la secuencialidad o orden de evaluación de dos funciones.

La sintaxis de la primitiva *par* es de la forma *p 'par' e*, donde *p* y *e* son las funciones a las cuales se les aplica el paralelismo. Internamente la primitiva *par* crea un hilo para evaluar la función *p* y el hilo principal se encarga de evaluar la función *e*, todo esto de manera paralela. De forma similar la primitiva *seq* se aplica sobre dos funciones *p* y *e* de la forma *p 'seq' e*. Esto indica que la evaluación de *e* no se devuelve hasta que se haya evaluado *p*.

Para más claridad de las primitivas *par, seq* y de GPH, se muestra un ejemplo para el cálculo paralelo de la serie de Fibonacci. La serie parte de que el Fibonacci de un número *n* es la suma del Fibonacci *n-1* + Fibonacci *n-2*, con los casos base de que el Fibonacci(0)=1 y el Fibonacci(1)=1.

```
1.  pfib n
2.    |  n <= 1 = 1
3.    |  otherwise = n1 'par' n2 'seq' n1+n2
4.                   where
5.                     n1 = pfib (n-1)
6.                     n2 = pfib (n-2)
```

En la línea 1 se especifica el nombre para la función la cual recibe como entrada un número *n*. Dentro del código se tienen dos partes o funciones principales precedidas por el símbolo pipe |. En la primera parte (línea 2) se tiene la definición base para la serie de Fibonacci. De la línea 3 a 6 se define el caso general de la serie de Fibonacci. En la línea 3 es donde se hace uso de los combinadores *par* y *seq* (el operador 'seq' tiene mayor precedencia que 'par').

En la instrucción se crea un hilo con la evaluación de *n1*. El hilo padre evalúa *n2*, quién no hace la suma (n1 + n2) hasta que *n2* termine. No obstante, si *n2* termina antes que *n1*, no puede continuar porque el hilo hijo no ha terminado (n1). Por otro lado, si *n1* termina antes que *n2* no puede continuar la recursividad paralela hasta que termine *n2*. El programa anterior daría resultados erróneos o quedaría bloqueado si se hubiera utilizado 'par' en lugar de 'seq'. En si los autores de GPH [89], comentan que la buena eficiencia de un programa va a depender del uso de los combinadores 'seq' y

‘par‘, incrustados en todo el programa por parte del programador.

Otro ejemplo de GPH donde el uso de los combinadores se complica aún más con la notación matemática (lógica combinatoria), es el siguiente.

```
parMap  : :  (a->b) -> [a] -> [b]
parMap  f  []  = []
parMap  f  (x:xs)  = fx ‘par‘ fxs  ‘seq‘ (fx:fxs)
                          where
                            fx  =  f  x
                            fxs  = parMap f xs
```

El código anterior describe cómo una función x es evaluada de manera paralela sobre un arreglo de datos. En el código se tiene un mapeo paralelo llamado parMap, que consiste en que la misma función pasada como argumento es aplicada a cada uno de los elementos de un arreglo, todo esto de manera paralela.

3.4. eSkel y SAMBA

Estos ambientes de programación pertenecen al modelo de programación de Skeletons. Sólo para recordar, los Skeletons son funciones predefinidas que siguen algún patrón paralelo.

Inicialmente los Skeletons fueron construidos a partir de los lenguajes funcionales [34, 56, 84]. Esto complicó la programación ya que era necesario conocer el lenguaje funcional y hacer los llamados a los Skeletons predefinidos.

La vertiente de algunos Skeletons es que éstos se usen desde un lenguaje imperativo como Skil [14, 15] en vez de hacerlo desde un lenguaje declarativo. Aquí la programación fue un poco más sencilla, pero aún sigue siendo necesario el aprender un nuevo lenguaje. En el siguiente pseudocódigo se muestra un fragmento de Skil que resuelve la multiplicación de matrices en forma paralela.

```
void matmult (int n) {
array <int> a, b, c ;
```

```
a = array_create (2, {n,n} 4, {0,0}, {-1,-I}, init_fl, DISTR_TORUS2D) ;
b = array_create (2, {n,n}, {0,0}, {-I,-I}, init_f2, DISTR_TORUS2D) ;
array_rotate_parts_horiz ((-)(0), a, a) ;
array_rotate_parts_vert ((-)(0), b, b) ;
for (i = 0 ; i < xPartDim ; i++) {
array_rotate_parts_horiz (minone, a, a) ;
array_rotate_parts_vert (minone, b, b) ; }
array_destroy (a) ; array_destroy (b) ;  }
```

Estos enfoques de ver a los Skeletons con un nuevo lenguaje (ya sea declarativo o imperativo) dificulta la aceptación ya que la mayoría de los programadores de aplicaciones paralelas utilizan C [55] o Fortran [42]. Una tercera opción es ver a los Skeletons no como un lenguaje, sino como una biblioteca de funciones incluida en un lenguaje de programación como C o el mismo Fortran.

Esta última opción es la que sigue eSkel (The **e**dinburgh **Ske**leton **l**ibrary) [10]. La actual especificación de eSkel define 5 Skeletons (Pipeline, Deal, Farm, Haloswap y Buterfly). De estos cinco sólo los dos primeros han sido implementados y liberados.

El Skeleton Pipeline permite un conjunto numerado de actividades (estados), las cuales son encadenadas y se les aplica un flujo de entradas de datos. Los datos son pasados de un estado a otro siguiendo la numeración. En la transferencia de datos el programador elige entre una transferencia automática (los datos de salida son la entrada del siguiente estado) o un modo explícito donde las salidas no son necesariamente entradas de otro estado. En este caso el programador utiliza las llamadas explícitas *Give* y *Take*.

El Skeleton Deal sirve para distribuir tareas de manera cíclica a los trabajadores. Esto conlleva a que el skeleton sea apropiado para tareas que sean homogéneas. El skeleton es útil anidado en un Pipeline como se muestra en la Figura 3.2. En la figura se muestra un pipeline de 3 estados (S0, S1 y S2) con un skeleton Deal anidado en el estado S1.

La semántica de los skeletons es un tanto complicada, ya que es necesario definir un gran número de parámetros como vemos en el siguiente skeleton Pipeline.

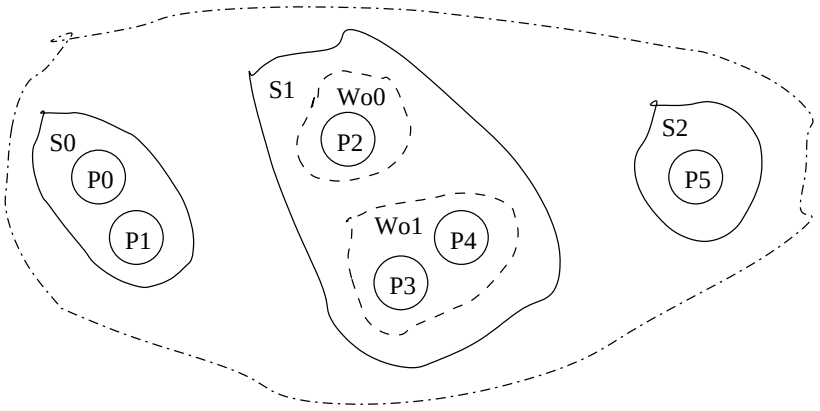

Figura 3.2: Estados y distribución de procesos en eSkel.

```
Pipeline(STAGES, imodes, stages, mystagenum, BUF, spreads, types, (void *)inputs,
    INPUTSZ, INPUTNB, (void *)results, INPUTSZ, &outmul, INPUTNB*INPUTSZ,mycomm());
```

Estos 15 parámetros indican a grandes rasgos el número de estados, la interactividad entre ellos (automática o heredada), el nombre de los estados, el número de procesos por estados (P0-P5 en Figura 3.2), el manejo global o local de los datos en los estados, el tipo de datos, los datos a transferir, el tamaño y la distribución de estos en los estados, el arreglo de salida, el tamaño y la distribución, y por último el comunicador del skeleton. De manera similar el skeleton Deal maneja 17 parámetros [10].

Como vemos el manejo de los dos skeletons disponibles no es tarea simple, por el gran número de parámetros que manejan. No obstante, ocultan en cierta forma el particionamiento, la comunicación y la sincronización. Por otro lado es necesaria la identificación de procesos en la asignación de procesos a los estados del pipeline. Es decir, dependiendo del número de proceso, este se asigna como parte de un estado del pipeline, Figura 3.2.

Otro inconveniente es que es necesario una comunicación explícita entre los estados del pipeline, mediante operaciones Give&Take, aunque ésta puede ser automática para patrones regulares. Algo más es que eSkel necesita versiones de MPI especializadas en el manejo de threads seguros [7].

SAMBA (Single Application Multiple Load Balancing) [72] es un framework (skele-

ton) para el desarrollo de aplicaciones SPMD 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 llenar (escribir) los métodos dentro de las clases que ofrecen como: Mediador, Balanceador-de-carga, Canal de transferencia, Repositorio y Tareas entre otros. Dentro de estas clases es donde toma forma el skeleton al ir llenando los métodos previamente establecidos por SAMBA. 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.

Por ejemplo para una multiplicación de matrices tienen que hacerse los siguientes pasos: escribir dentro de la clase Mediador los métodos *crear_tareas* y *manejo_de_resultados*. En el primero se escribe el código para que un procesador central lea las matrices A y B, para después replicar o enviar toda la matriz B a todos los procesadores. Posteriormente es necesario hacer el código que defina una tarea, la cual contiene un reglón de la matriz A, y es insertada en el repositorio. En el segundo método es necesario definir el código donde el procesador central reciba los resultados obtenidos en la matriz resultante R.

En el método *ejecutar_tarea* de la clase Tareas, se concentra en el código referente al cómputo de un renglón de la matriz A por todas la columnas de la matriz B, todo esto para obtener la matriz resultante.

Adicionalmente es necesario definir la forma en que se maneja el repositorio; en este caso de SAMBA sólo se tiene implementada el acceso tipo pila. Por otro lado es necesario definir o escoger la estrategia de distribución de carga entre los procesadores (7 algoritmos a escoger).

Se puede apreciar que implementar una tarea en SAMBA no es tarea sencilla, más que nada porque adicionalmente al manejo de skeletons, es necesario el saber programación orientado a objetos, lo cual, muchos desarrolladores (como químicos, físicos o biólogos) desconocen. También cabe mencionar que aunque existan muchas cosas hechas, como los algoritmos de distribución de carga, es difícil escoger un algoritmo

adecuado sin un entendimiento total del sistema. Por otro lado, el manejo de los skeletons dentro de un concepto de objetos hace posible el reuso del código para varios dominios de aplicaciones.

MALLBA (MAlaga, La Laguna, BArcelona) [3, 44], 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.

3.5. SADTs

En el modelo de tipos de datos abstractos paralelos encontramos a SADTs (Shared Abstract Data Types) [45]. La meta original de SADTs es facilitar la programación paralela basándose en el concepto de ADTs encapsulando la comunicación entre procesadores a través del uso de las operaciones sobre el ADT. Para ello el ambiente maneja diferentes tipos de datos abstractos como: tipos compartidos simples (enteros, flotantes), acumuladores, colas y colas de prioridad. Los tipos compartidos simples son variables compartidas que todos los procesos pueden escribir/leer. El acumulador es un tipo que combina un valor con una función de actualización [46]. El valor es actualizado combinando el nuevo valor con el valor actual, a través de la función de actualización. Las colas y colas de prioridad (subsección 2.2.5) son estructuras tradicionales que permiten el encolar y desencolar datos.

SADTs simplifica el desarrollo de aplicaciones separando por un lado el código propio de la aplicación y encapsulando en sus llamados al ADT, el trabajo de la comunicación, la sincronización y la distribución de datos. Adicionalmente, ofrece un alto rendimiento soportando diferentes implementaciones de sus ADTs para diferentes arquitecturas.

Las tres diferentes implementaciones que propone son: secuencial, concurrente y

concurrentes distribuidos. Las cuales en general se aplican a todos sus ADTs, de manera general explicamos las implementaciones para las colas.

En la implementación secuencial la cola esta en una sola memoria donde los procesos de los procesadores pueden acceder a ella (ya sea para encolar o desencolar), pero sólo uno a la vez. Este control de acceso de logra mediante candados. En la implementación concurrente la cola se encuentra en una sola memoria, pero ahora son permitidos accesos concurrentes (dos en total). El control se logra manteniendo apuntadores al frente y final de la cola y usando primitivas wait-free (similares a test&set y compare&swap) [49]. En estos dos primeros tipos de acceso el problema fue el cuellos de botella que ocasiona el acceso a la cola. Para evitar los cuellos de botella SADTs propone accesos concurrentes distribuidos donde la cola se distribuye en todo el sistema.

En este tipo de acceso se tienen dos implementaciones: una donde la cola se encuentra *replicada* en todos los procesadores y otra donde la cola se encuentra *repartida* físicamente entre todos los procesadores.

En la implementación replicada, se tiene una copia de la cola por cada procesador. Esta implementación permite que los procesadores accedan de manera local a la cola, sin embargo, tiene el inconveniente que cada acceso requiere de mantener una coherencia fuerte de las copias. La coherencia fuerte se refiere a que cada acceso o escritura a la cola (encolar-desencolar) requiere que se difunda a todas las copias del sistema. Existe una versión de SADTs donde la coherencia es débil, en ella todos los procesadores pueden agregar elementos a su cola local pero al momento de tomar elementos pueden hacer de diferentes partes de la cola, no necesariamente del frente, perdiéndose así el concepto de cola y cola de prioridad. Una alternativa es tener un proceso adicional que cada cierto tiempo verifique las colas y haga los movimientos o intercambios necesarios para mantener el concepto de cola. Esta versión fue ejecutada en clusters de máquinas SGI con sistema operativos y compiladores propietarios. El lenguaje de desarrollo para la versión es *Modula-3* [69] que es un lenguaje desarrollado por DEC (Digital Equipment Corporation) y adicionalmente para la comunicación se utilizó *Network Objects System* que es un sistema para programación distribuida en Modula-3 [12].

En la implementación repartida, la cola necesita de un ambiente de memoria compartida. El cual puede ser brindado por una arquitectura multiprocesador o por una multicomputadora que maneje memoria compartida. En SADTs consideran una multicomputadora Cray T3D la cual ofrece la biblioteca SHMEM [9] para el manejo de memoria compartida, junto con lenguaje C. Bajo el ambiente de memoria compartida se tiene en SADTs una arquitectura donde se tienen dos listas por cada procesador. Una lista es para encolar y otra para desencolar. Este esquema tiene la ventaja de ser muy eficiente y escalable como reportan sus autores, sin embargo, como ellos también reportan puede suceder que la cola no sea consistente, es decir, que los elementos no sean procesados en el orden en que fueron ingresados en la cola.

A manera de resumen las versiones de SADTs tienen el compromiso entre consistencia vs. eficiencia. Si se requiere consistencia del ADT el ambiente es más lento y requiere de arquitecturas multiprocesador con un mayor ancho banda en las comunicaciones. Si se requiere eficiencia se sacrifica en la definición del ADT (no necesariamente FIFO) y de todas formas es necesario una máquina de alta velocidad. En las arquitecturas multicomputadora definitivamente no son eficientes ni consistentes. En el caso que se requiera más consistencia se vuelven aún más lentas.

La parte fuerte y novedosa de SADTs es su interfaz de programación, cuyos requerimientos es elegir alguno de los ADTs disponibles y comenzarlos a usar en el programa. Para ello SADTs se ofrece como una biblioteca de funciones la cual puede ser llamada desde algún lenguaje como C. A continuación se muestra un ejemplo de la suma de N números usando SADTs.

```
1.  #include SADTs.h
2.  #define N 100
3.  Queue_SADT cola_numeros;
4.  Int_SADT suma_total=0;
5.  int numeros[N];

6.  Init_Queue_ADT(cola_numeros);
7.  if (pid==0)  {
```

```
8.    Inicializa (numeros);
9.    for (i=0;i<N;i++)
10.     Queue_Enqueue(cola_numeros,numeros[i]);
11. }
12. barrier();
13. while(true)  {
14.     valor=Queue_Dequeue(cola_numeros);
15.     if (task_empty(valor))    exit;
16.     suma_total=suma_total+valor;
17. }
18. barrier();
19. if (pid==0)
20.   printf("La suma total es %d \n",suma_total);
```

En la línea 1 se declara la biblioteca que contiene los ADTs de de SADTs. En la línea 2 se define N que es la cantidad de números a sumar en paralelo. De la línea 3-5 se hacen las declaraciones o tipos de ADT que se van a utilizar, una cola compartida que contiene los números a sumar y una variable compartida que contendrá la suma total. Adicionalmente se requiere un arreglo que contenga los números para que estos sean encolados en la cola compartida (línea 5).

En la línea 6 comienza el código propiamente con una inicialización de la cola. En las líneas 7-11 el proceso cero inicializa el arreglo de números, es decir, les asigna un valor. Después el mismo proceso cero encola todo el arreglo de números en la cola compartida. Mientras se hace todo lo anterior los demás procesos $1 \leq p \leq P-1$ permanecen bloqueados en la barrera de la línea 12. La barrera detiene el avance de los procesos hasta que todos sin excepción lleguen a ese punto (en este caso el proceso 0). Una vez pasada la barrera los procesos entran a un ciclo while (línea 13-17) en el que todos los procesos desencolan un número de la cola compartida (línea 14). Estos procesos verifican si se obtuvo un valor válido (número) o inválido (cola vacía y fin de programa) en la línea 15. En caso de que sea un valor válido hacen la suma del número obtenido con la variable *suma_total* de ese instante. Hecho esto, vuelven a

intentar desencolar otro número. Cuando ya no hay elementos que procesar en la cola, los procesos salen del ciclo para entrar a una nueva barrera. Esta barrera es necesaria ya que algún proceso puede estar haciendo una última suma (línea 18). Finalmente el procesador cero imprime el resultado final.

Del ejemplo se puede decir que la programación es similar con TreadMarks (Sección 3.2) en el sentido de que ambos usan un esquema compartido. Aunque la diferencia es que con los ADTs de SADTs, en este caso la cola compartida, se evita la asignación explícita de los datos. Otra ventaja de la programación con SADTs es que en su interface no existen los candados, estos se manejan de manera interna, por ejemplo al hacer las sumas concurrentes en la variable *suma_total* del tipo Int_SADT. Algo similar con TreadMarks es el uso explícito de barreras como mecanismo de sincronización.

A manera de resumen SADTs oculta la asignación de datos , así como la comunicación y sincronización explícita entre procesos. En cuanto al particionamiento, o la definición de los datos, esta se hace de manera explícita. El problema principal es organizar los datos en sus ADTs disponibles. Entre sus ADTs se encuentran la cola, cola de prioridad, acumulador y tipos simples compartidos (enteros, flotantes). El diseño de SADTs es para arquitecturas multiprocesador UMA o NUMA. En esas arquitecturas tienen varias implementaciones de sus ADTs replicados-coherentes distribuidos-no coherentes. Estos últimos son más rápidos y tienen una mejor escalabilidad. Para arquitecturas de memoria distribuida sus implementaciones no son coherentes ni tampoco escalables. Las aplicaciones que han reportado son aplicaciones irregulares como el agente viajero, el conjunto de Mandelbroot y CFD (Dinámica de Fluidos Computacional) [45, 66]

3.6. Resumen

En este capítulo hemos descrito algunos de los principales ambientes de programación paralela. Estos ambientes se han presentado en el contexto de los modelos de programación discutidos en el Capítulo 2. Los ambientes han sido escogidos por ser de

los más representativos y usados en cada modelo.

PVM y MPI pueden aún ser considerados como ambientes de programación de bajo nivel. Esto permite desarrollar programas paralelos ad-hoc, con el mejor de los rendimientos. Por otro lado la programación es complicada debido a que todos los pasos de la programación paralela (particionamiento, asignación, comunicación y sincronización) se hacen de manera explícita por el programador.

TreadMarks es un ambiente de programación basado en memoria compartida distribuida. Con TreadMarks, no existe comunicación para compartir datos pero si existe particionamiento y comunicación para sincronización. La comunicación es transparente al momento de acceder a las variables compartidas, no así la sincronización, la cual es explícita y es manejada por candados. Cabe mencionar que dependiendo de la aplicación el rendimiento se puede ver afectado por la coherencia y el control de la concurrencia en el acceso a los datos. Esto último se ve aumentado cuando se manejan muchos procesadores o arquitecturas débilmente acopladas como los clusters donde el mantener la coherencia de los datos puede ser costosa si los canales de comunicación son lentos.

GPH es un ambiente de programación paralelo basado en el lenguaje funcional de alto nivel Haskell. Con GPH, la programación es muy corta y en sí, se evitan todos los pasos de la programación. Sin embargo, la programación no es sencilla ni eficiente. Lo primero es debido al manejo de notación matemática (lógica combinatoria), lo segundo se debe a que toda la generación del código paralelo se deja a los compiladores. Estos compiladores tratando de ser más eficientes han optado por tener operadores o combinadores (‘par‘ y ‘seq‘) con los cuales el programador, de manera explícita, describe más la aplicación ayudando al compilador con la generación del código.

eSkel y SAMBA son un ejemplo de ambiente de programación paralela basado en el modelo de Skeletons. Los skeletons definen un conjunto de funciones predefinidas con un patrón paralelo. El gran problema de los skeletons es que no existe un estándar de estos y existe una enorme variedad de ellos, entre ellos eSkel. En sus skeletons los cuales son manejados como una biblioteca de skeletons, eSkel facilita el particionamiento, la asignación, la comunicación y hace transparente la sincronización. Sin embargo, el

problema de eSkel es que todos esos pasos van incluidos como parte de la definición de parámetros en sus skeletons Pipeline y Deal, lo que origina que la programación sea complicada. SAMBA es del mismo estilo con la diferencia que su definición y funcionamiento se hace dentro de los métodos de algunas clases predefinidas (modelo orientado a objetos).

SADTs es un ambiente de programación de mediano nivel de abstracción. Con SADTs se evita la asignación explícita de datos, así como la comunicación y sincronización explícita entre procesos. Con SADTs el programador coloca sus datos a procesar en algunos de los ADTs (colas, colas de prioridad, acumulador y tipos básicos compartidos) y los comienza a utilizar usando las llamadas típicas del ADT. SADTs se encarga de manera transparente del manejo de los datos en todo el sistema. Esto incluye coherencia, replicación y acceso controlado a los ADTs. Dependiendo de la aplicación estos factores pueden afectar el rendimiento más aun si se trata de arquitecturas débilmente acopladas como los clusters. Otra desventaja es que las aplicaciones a desarrollar se deben representar en forma de los ADTs disponibles, en este caso las colas.

Capítulo 4

Balance de Carga en los Ambientes de Programación Paralela

En el capítulo anterior se describieron algunos de los principales ambientes de programación paralela. Estos ambientes como ya se discutió, se basan en los modelos de programación planteados en el Capítulo 2. En este capítulo hablaremos de la necesidad de integrar técnicas de distribución de carga en los ambientes de programación descritos, y la manera en que su implementación se hace transparente al usuario.

4.1. Introducción

El tiempo de ejecución de un programa paralelo se mide cuando TODOS los procesadores han terminado con su carga, es decir, el tiempo de ejecución TOTAL es igual al tiempo de ejecución del procesador que termina al último. Por esta razón, es de vital importancia que todos los procesadores finalicen más o menos al mismo tiempo y no suceda el caso extremo de que *p-1* procesadores finalicen; pero el procesador restante todavía tiene mucho trabajo[1].

Una alternativa para que los procesadores terminen al mismo tiempo es hacer una

[1]trabajo, tarea, datos y carga se usan de manera indistinta en este capítulo.

planificación estática sobre la asignación de carga en los procesadores. En la planificación estática la decisión de cómo asignar las tareas en los procesadores se toma antes de que la ejecución comience, es decir, a tiempo de compilación. Para ello es necesario conocer los tiempos de ejecución de las tareas, y los recursos de procesamiento. En la planificación estática una tarea siempre se ejecuta en el procesador que fue asignado. Esta planificación tiene la ventaja de que el costo adicional es a tiempo de compilación y no a tiempo de ejecución. Sin embargo, tiene algunas desventajas como el generar una óptima planificación es un problema NP-completo. Esta planificación puede ser mapeada al "particionamiento de conjuntos"[98]que indica.

Sea Z el conjunto de enteros. Dado un conjunto finito A y un tamaño $s(a) \in Z$ para cada $a \in A$, encontrar un subconjunto de A (esto es A') tal que:

$$\sum_{a \in A'} s(a) = \sum_{b \in (A - A')} s(b) \tag{4.1}$$

Adicionalmente, existen ciertos casos donde la planificación estática se complica, por ejemplo, cuando la aplicación es dinámica, cuando el cluster es heterogéneo y cuando existe multiprogramación.

Caso de aplicación dinámica

Una aplicación es dinámica cuando no se conoce la cantidad de datos a procesar, ni el costo que involucra el procesamiento. En sí, los datos se van generando a tiempo de ejecución, es decir, el procesamiento de ciertos datos puede generar el procesamiento de más datos. Si algunos procesadores tienen más datos que procesar que otros, intuitivamente se afectará la eficiencia total de la aplicación. En este caso, una redistribución de datos (dinámica) después de la asignación inicial (estática) puede mejorar la eficiencia.

Otro caso de aplicación dinámica es cuando la cantidad de datos a procesar no cambia, pero el procesamiento de cada dato tiene costo diferente, es decir, el procesar un dato u otro involucra diferente costo de procesamiento. Por ejemplo, en el procesamiento de imágenes, el procesar un pixel o una región de pixels puede tener diferente costo,

comparado con otros pixels o regiones dependiendo de los colores de cada uno. En este caso, una asignación de datos equitativa entre los procesadores afectará la eficiencia, ya que algunos procesadores terminarán antes que otros. De nuevo una redistribución dinámica puede ayudar.

Caso de cluster heterogéneo

Cuando un cluster es heterogéneo, algunos nodos tienen mayor poder o capacidad de procesamiento que otros. Bajo este escenario, si el conjunto total de datos a procesar se distribuye equitativamente entre todos los nodos, los procesadores con mayor capacidad terminarán antes el procesamiento de sus datos que los procesadores de menor capacidad. Una asignación equitativa (y definitiva) afecta la eficiencia, ya que los nodos con más capacidad tienen que esperar o estar ociosos, hasta que los nodos con menor capacidad terminen. Nuevamente una redistribución de datos entre los nodos, puede mejorar la eficiencia.

Caso de multiprogramación

En un cluster con multiprogramación, más de un usuario puede ejecutar sus aplicaciones a la vez (no-dedicado). En tal caso, una asignación equitativa y definitiva de los datos no es eficiente, ya que el procesamiento de datos en nodos compartidos por varios usuarios es más lento en comparación con los nodos no compartidos. Una forma de mejorar la eficiencia sería que, a tiempo de corrida, algunos datos a procesar de los nodos compartidos (sobrecargados), se redistribuyan hacia otros nodos menos cargados.

4.2. Balance de carga

El balance dinámico de carga o simplemente balance de carga, se define como la redistribución de carga entre los procesadores a tiempo de ejecución en función de la capacidad de los procesadores. Esta redistribución es ejecutada transfiriendo tareas desde procesadores altamente cargados a procesadores menos cargados con el objetivo

de mejorar el rendimiento de la aplicación. Entre las ventajas del balance de carga están que no se necesita saber el tiempo de ejecución de las aplicaciones, la flexibilidad de ajustarse a los cambios imprevistos de la aplicación y ajustarse a las contenciones y comunicaciones en clusters. Entre las desventajas está el costo que involucra esta redistribución.

4.2.1. Políticas de balance de carga

Para definir la forma en que se balancea la carga, se han propuesto diferentes políticas[2] que ayudan a definir la distribución. Estas políticas incluyen: información, localización, transferencia y selección [59, 101].

La política de información define lo que se considera "carga", y el *cómo* recolectarla, la localización define hacia *dónde* mover carga dependiendo de la información recolectada. La política de transferencia define el *cuándo* y *quién* hace la distribución y finalmente la política de selección indica *qué* y *cuánta* carga se va a transferir. Los detalles vienen a continuación.

Política de información

La política de información consiste en obtener información sobre el estado de carga del sistema o de los procesadores. Para ello es necesario definir lo que será considerado como carga para posteriormente recolectarla del sistema.

Algunos indicadores de carga pueden ser: el número de procesos en el procesador, el porcentaje de utilización de CPU, la longitud de la cola en espera de CPU [36, 38, 39, 82] o el número de datos a procesar en una aplicación específica [19, 79].

El número de procesos puede reflejar la carga en el procesador. A mayor número de procesos mayor carga. Sin embargo, puede suceder que existan muchos procesos pero con poco trabajo o que no estén haciendo nada (sleeping), de igual forma puede haber

[2]RAE (Real Academia Española). Orientaciones o directrices que rigen la actuación de una persona o entidad en un asunto o campo determinado.

pocos procesos pero estos se encuentran en ejecución (running).

El porcentaje de utilización de CPU y la longitud de la cola de espera reflejan de mejor forma la carga ya que miden directamente el uso actual del procesador.

El número de datos que tiene que procesar una aplicación en específico es otra forma de considerar la carga.

Una vez definido lo que es la carga, es necesario saber cómo se recolecta. Esta recolecta puede ser global o parcial. Es global cuando se colecta el estado de carga de todos los procesadores del sistema. Por otro lado, es parcial cuando se colecta el estado de carga de algunos procesadores.

Las dos formas de recolección tienen ventajas y desventajas. La ventaja de la global es precisamente que tiene una vista global de la carga por lo que pueden hacer una distribución (balance) más eficiente. La desventaja es que puede ser muy costosa por el gran número de colectas que requiere la vista global, más aún cuando son muchos procesadores. La recolección parcial tiene la ventaja que la colecta de información no es tan costosa en comunicaciones, sin embargo, no se obtiene una vista general de todo el sistema, lo que conduce a no tener un balance global, al menos en ese instante.

Política de localización

En el balance de carga, la política de localización determina el *dónde* mover la carga. Esta determinación puede ser de dos formas: ciega o basada en la política de información.

En la política de localización ciega, el procesador destino, que es el procesador al que se le transfiere la carga, se elige sin tomar en cuenta el estado de carga del sistema (no hay política de información). Esta política tiene la ventaja de ser muy rápida en la elección. Sin embargo, la desventaja es la posibilidad de que el procesador destino se encuentre con mucha carga y se le asigne todavía más. Un ejemplo de política de localización ciega es la política de localización cíclica (se distribuye la carga a los procesadores de manera circular).

La política de localización basada en la política de información sí toma en cuenta

en estado de carga de los procesadores y en base a esa información se puede hacer una transferencia óptima. Esta política es principalmente útil cuando la información de carga es tan variable como en las aplicaciones dinámicas, en los clusters heterogéneos y/o los clusters no-dedicados. Sin embargo, la desventaja es el costo que involucra el tener que recolectar dicha información.

Política de transferencia

La política de transferencia define el *cuándo* se tiene que hacer la distribución de carga y el *quién* hace la distribución. Para el *cuándo*, se maneja el concepto de Fronteras de Carga. Una frontera de carga se define como el límite que divide dos regiones de carga. En la Figura 4.1 se muestran tres regiones de carga (descargado, carga normal y sobrecargado), divididas por las fronteras de carga `F1` y `F2`. Puede suceder que la `F1` sea igual a la `F2` en cuyo caso sólo se tienen dos regiones de carga (descargado y sobrecargado). El pasar de una región a otra, en base a alguno de los indicadores de carga previamente descritos en la política de información, determina o marca el *cuándo* debe comenzar una distribución o redistribución. Por ejemplo, si un procesador pasa de un estado descargado a uno sobrecargado se inicia una redistribución buscando a aquellos procesadores que se encuentren en el estado descargado, para enviarles carga. También puede suceder que un procesador en estado sobrecargado pase a un estado descargado, en cuyo caso buscará uno o varios procesadores sobrecargados para solicitarles carga.

La otra parte de la política de transferencia se relaciona con el *quién* toma la decisión de distribución. Aquí se habla de un control centralizado o distribuido. El control es centralizado cuando un solo procesador central se encarga de decidir los momentos de distribución. Por otro lado, el control es distribuido cuando cualquier procesador en el sistema es capaz de decidir cuando iniciar una transferencia.

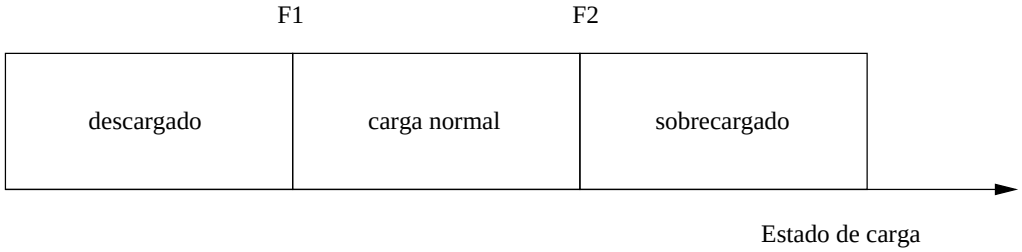

Figura 4.1: Política de transferencia con mecanismo de doble frontera.

Política de selección

La política de selección define el *qué*, *cuál* y *cuánta* carga transferir. El qué se refiere a si lo que va a transferir son procesos o datos. Si se trata de procesos estos pueden ser nuevos o en ejecución. Si son nuevos, antes de comenzar su ejecución estos se transfieren a algún procesador para ejecutarse. Si los procesos ya están en ejecución, estos también se pueden transferir hacia otro procesador (migración). Cabe mencionar que en muchas ocasiones los costos de migración son más altos comparados con la decisión de dejar a que el proceso termine su procesamiento localmente (sin migración). Una vez decidido qué tipos de procesos transferir (nuevos o en ejecución), falta definir cuáles y cuántos de esos procesos. En el caso de datos, la política de selección define de manera similar cuáles y cuántos datos transferir.

4.2.2. Algoritmos de balance de carga

Un algoritmo de balance de carga se construye en base a las políticas de de carga seleccionadas. A continuación mencionamos los que son relevantes para nuestro trabajo. Otros algoritmos se pueden encontrar en Castro01 [26].

Global distribuido

Este algoritmo maneja un política de información con recolección global y una política de transferencia con control distribuido. En este algoritmo todos los procesadores cuantifican su estado de carga cada cierto tiempo. Cada vez que existe un cambio en el

estado de carga, se lo notifican a los demás procesadores. Esto hace que cada procesador tenga una vista global del estado de carga de todo el sistema. Cuando es necesario hacer una redistribución, cada procesador analiza la carga del sistema, y en base ello efectúa la redistribución o asignación. Este algoritmo típicamente se activa por carga alta (iniciado por el emisor), es decir, cuando algún procesador cambia su estado de carga a sobrecargado, se inicia la distribución, verificando la información global que posee y seleccionando al nodo menos cargado para enviarle carga. La información físicamente es un vector de carga (arreglo de datos) el cual contiene los estados de carga de todos los procesadores.

Subasta

Este algoritmo consiste en subastar poder de cómputo. El algoritmo comienza cuando un procesador se queda sin carga y subasta su poder de cómputo a los demás procesadores. Para ello solicita a los demás procesadores su oferta (estado de carga). El procesador quien da más (en este caso carga) es el elegido para venderle el poder de procesamiento o lo que es lo mismo, pedirle carga. El algoritmo se repite cada vez que un procesador se queda sin carga hasta que no existe más carga en el sistema. En cuanto a las política de localización esta se basa en la política de información con colecta global o parcial. En cuanto a la política de transferencia, el *cuándo* se inicia la distribución es cuando se queda sin carga un procesador. En cuanto al *quién* inicia la distribución son todos los procesadores (control distribuido). Finalmente en cuanto a la política de selección el *qué* normalmente son datos y el *cuántos* datos a transferir puede ser desde uno, hasta todos los datos que tenga el procesador elegido.

4.3. Balance de carga en ambientes de programación

En esta sección hablaremos de la integración del balance de carga, en el contexto de algunos de los ambientes de programación del Capítulo 3.

En general, los ambientes de programación tienen como objetivo principal ser una herramienta para el desarrollo de programas paralelos y no para balance de carga. No obstante, algunos contemplan ciertas formas de balance.

PVM/MPI

PVM y MPI consideran por defecto un balance de procesos basada en una política de localización cíclica (sin política de información). Con PVM es posible definir otras políticas de distribución pero estas tienen que ser implementadas por el programador. Cada vez que PVM o MPI comienzan la ejecución de n procesos, los distribuyen de manera unitaria, cíclica y automática en los procesadores disponibles. Es unitaria porque se asigna un proceso por procesador a la vez. Es cíclica porque una vez que todos los procesadores tienen un proceso, se comienza de nuevo la asignación con el primer procesador. Finalmente es automática porque la distribución la hace el ambiente de manera transparente. Entre las limitantes para ambas herramientas es que sólo se distribuyen procesos. Si se quieren distribuir datos, el programador debe diseñar y programar la distribución de los mismos usando las primitivas de envío y recepción mensajes de PVM y MPI.

TreadMarks/Linda

Los ambientes TreadMarks/Linda al igual que PVM y MPI hacen una distribución de procesos con política de localización cíclica. En cuanto al balance de datos, este no se realiza ya que no se toma en cuenta la carga del sistema (por ejemplo con multiprogramación o heterogeneidad del cluster). No obstante el sistema indirectamente

mueve datos. En la lectura, el sistema se encarga de mover los datos compartidos hacia los procesadores que los necesitan, mejorando la eficiencia al tener accesos locales. En la escritura, la situación es distinta ya que puede haber muchas copias. Si alguna copia se modifica esto se debe reflejar de alguna forma en las demás copias (mantener coherentes). Existen básicamente dos protocolos de coherencia: writer-update y writer-invalidate [16]. En writer-update cada vez que una copia se modifica (escribe), la actualización se difunde a todas las copias. En writer-invalidate cada vez que una copia se modifica, el sistema invalida las copias existentes (no las actualiza), en este caso cuando los procesadores a los cuales se les invalida su copia, el sistema les envía una nueva copia cuando estos la requieran. El protocolo writer-invalidate es el que usa TreadMarks.

Haskell

Algunas implementaciones de Haskell como GUM (Graph reduction for a Unified Machine model) [90] manejan una distribución de carga. La distribución de GUM, llamada FISH tiene una política de transferencia de control distribuido y una política de localización ciega (aleatoria). La distribución FISH funciona de la siguiente manera, cada vez que un procesador x se queda sin carga (funciones o datos que evaluar) elige de manera aleatoria otro procesador y, el cual puede tener o no carga. Si tiene, y se la envía y el receptor x responde con una aceptación. En caso contrario, incrementa un contador (edad) y elige aleatoriamente otro procesador. El contador o edad de pesca sirve para detener una búsqueda infructuosa. Si la edad llega a cierto límite la pesca se detiene un tiempo (*delay*) y después vuelve a comenzar la búsqueda. Este algoritmo tiene la ventaja de ser rápido, sin embargo, algo difícil puede ser la selección de la variable edad y de la variable *delay*. Por ejemplo, puede suceder que todos los procesadores estén desocupados y la variable edad sea alta (buscando donde no hay peces). Esta búsqueda quizá no les quita tiempo de CPU a los procesadores pero si puede saturar los canales de comunicación. Un caso extremo con la variable delay es que por alguna razón no se pesco en la primera ocasión pero ciertamente hay peces, en este caso si

la variable delay es alta se tardará más tiempo en pescar procesadores sobrecargados. Otra limitante/desventaja es que la elección aleatoria no asegura la pesca del pez más gordo (procesador con más carga).

MALLBA/eSkel/SAMBA

En el ambiente MALLBA se considera una distribución llamada petición-respuesta [44], la cual corresponde a un control distribuido y una política de información con recolección global. El funcionamiento es el siguiente: cada vez que un procesador se queda sin datos que procesar, éste le hace una petición a todos los procesadores solicitándoles datos, estos siempre responden ya sea con carga `DATOS` (respuesta positiva) o con el mensaje `NO_HAY_CARGA` (respuesta negativa). En base a las respuestas, el procesador que inició la petición guarda la información o estados de carga (ocupado=positiva o desocupado=negativa) de los procesadores, para utilizarla en las peticiones subsecuentes (la siguiente vez sólo le pide a los ocupados). Debido a que todos los procesadores manejan un vector de carga, es decir donde se guarda la información del estado de carga de los nodos, cada vez que se da una respuesta a una petición (positiva o negativa), ésta se difunde a TODOS los procesadores para que actualicen su vector de información. Para evitar abrazos mortales o bloqueos debido al envío/recepción de mensajes, se maneja un número de petición (# de pedido) y un contador de respuestas pendientes. Esto es necesario debido a que el algoritmo es asíncrono, es decir, la petición no espera por una respuesta. En sí, las posibles respuestas a las peticiones se mezclan con el procesamiento de la aplicación, el cual se da en un ciclo. Cada cierto número de ciclos fijados manualmente por una variable o contador, se verifica si hay respuestas pendientes. Si las hay se contestan sino, se inicializa el contador y se continúa con el procesamiento. Entre las ventajas de este enfoque, podemos mencionar que las comunicaciones son asíncronas, lo que reduce tiempos de espera, así como que la información se encuentra de manera local en cada procesador. Sin embargo, cada cierto tiempo es necesario verificar si hay mensajes que responder para que la distribución tome lugar. Este tiempo no siempre es fácil de definir, ya que si es rápido se ocupa más el procesador, pero si es tardado la

distribución se retrasa cuando pudiera ser necesaria. Otra desventaja es el gran número de mensajes requeridos en la actualización global de la información del estado de carga (todos contra todos).

eSkel es muy simple en su distribución de carga [10, 11]. Su distribución es manejada únicamente por el skeleton DEAL y consiste en una política de transferencia cíclica de datos. Cada vez que existen datos que procesar estos se van asignando de manera cíclica a los procesadores. El esquema es sencillo, pero tiene varias limitantes, por ejemplo, el procesamiento de cada dato debe tener el mismo costo para que el skeleton DEAL no cause un desbalance entre los procesadores. Otra limitante es que todos los nodos deber tener las mismas características en poder de procesamiento, ya que la repartición de datos es equitativa. Si los nodos fuera diferentes unos nodos trabajarían más que otros. Finalmente la repartición equitativa no es eficiente cuando se trata de un cluster con multiprogramación como se vio con anterioridad (Sección 4.1).

El ambiente SAMBA [72] considera siete formas diferentes de distribución de carga: 1. Estática, 2. Demanda, 3. Distribuida global colectiva, 4. Centralizada global colectiva, 5. Distribuida global individual, 6. Distribuida local-particionada colectivo y 7. Distribuida local vecinos individual. En la distribución estática (1), un procesador maestro distribuye equitativamente todas las tareas (datos) entre todos los procesadores, posteriormente no hay redistribución. En la distribución por demanda (2), un procesador maestro se queda con una parte de las tareas y divide el resto entre todos los procesadores, los cuales al terminar su trabajo, solicitan al maestro más tareas y éste, por cada solicitud, les envía un número de tareas predefinido por el usuario (umbral). En la distribución distribuida global colectiva (3), el procesador maestro distribuye inicialmente todas las tareas equitativamente (incluyéndose el mismo), cuando algún procesador termina el procesamiento de sus tareas, éste procesador sin carga le envía un mensaje a todos los procesadores para iniciar una posible redistribución de tareas colectiva (entre todos). Los procesadores reciben el mensaje y reenvían a todos los procesadores su estado de carga actual (número de tareas por procesar). Después todos los procesadores analizan la información y si al menos un procesador tiene un número

de tareas que procesar por arriba de cierto umbral, se lleva a cabo una redistribución colectiva, obviamente enviándole tareas al proceso que no tiene y originó la redistribución. En la distribución centralizada global colectiva (4), es similar al 3, excepto en que la decisión de distribuir es tomada por un procesador central, para ello, cuando un procesador termina de procesar sus tareas, éste le envía un mensaje al resto para que a su vez le envíen su estado de carga al procesador central, quien analiza la información y decide o no la distribución colectiva. En la distribución distribuida global individual (5), la distribución de datos ocurre sólo entre dos procesadores, a diferencia de la colectiva que es entre todos. Nuevamente cuando un procesador se queda sin tareas, éste le solicita información a todos los procesadores quienes le responden con su estado de carga actual, posteriormente el procesador sin tareas analiza esta información de igual forma que en el algoritmo 3, es decir, si hay un procesador que rebase cierto umbral (cargado), el procesador sin tareas solicita tareas al procesador cargado. La distribución distribuida local-particionada colectiva (6) es básicamente la distribuida global colectivo (3), a diferencia de que los procesadores son particionandos en grupos disjuntos, definidos por el programador. Cuando un procesador termina sus tareas, manda un mensaje a los procesadores de su grupo solamente, quienes responden al mismo grupo y deciden hacer o no la distribución. La distribución distribuida local-vecinos individual (7) es similar a la distribuida global individual (5), excepto en que los mensajes nuevamente van hacia un grupo, nuevamente el tamaño del grupo es configurado por el programador. Sin embargo, en el 7, no existe distribución entre grupos.

En general podemos decir que SAMBA tiene muchas variantes de distribución de carga, lo que lo hace muy atractivo. Sin embargo, algo complicado en SAMBA es el saber escoger la adecuada distribución de carga para cierta aplicación, así como la selección del umbral adecuado. El umbral como ya se dijo en el párrafo anterior es el valor numérico que determina si se hace o no la distribución.

SADTs

SADTs considera diferentes formas de distribución de carga la cuales varían en función de sus diferentes implementaciones (Sección 3.5). En la implementación de copias replicadas, la distribución de datos prácticamente no existe ya que existen copias de los ADTs replicados en todos los procesadores. Por ejemplo, si un procesador obtiene un elemento de su cola local (la cual está replicada), el cambio se refleja en todos las copias mediante el protocolo de coherencia por actualización. De igual forma si algún procesador inserta un elemento en su cola local, el sistema se encarga de insertar el mismo elemento en todas las copias (coherencia por actualización). De esta forma, decimos que no hay una re/distribución de datos, los datos siempre están de manera local en cualquier procesador; obviamente con los costos que puede implicar el mantener la coherencia de las copias.

En la implementación donde el ADT se encuentra repartido en los diferentes espacios de direcciones sucede lo siguiente. En esta implementación SADTs se encarga de mantener una parte del ADT (cola o cola de prioridad) distribuido en todos los nodos de un cluster. Cada procesador puede encolar y desencolar en su cola local. La inconsistencia aquí es que la cola no es una cola compartida entre todos los procesadores, es decir, si se insertan elementos en cierto orden en el tiempo, estos no son recuperados en el mismo orden (consistencia débil), a menos que las operaciones encolar y desencolar sean locales.

Finalmente SADTs implementa una distribución de carga basada en el algoritmo de Wu-Kung[100]. El algoritmo mapeado a SADTs hace lo siguiente, cada procesador tiene una pila local a la cual inserta y elimina tareas (datos). Cuando no hay tareas busca en un pool global compartido. Si tiene tareas las obtiene de ahí, sino busca en las pilas de otros procesadores. Si no encuentra el algoritmo termina, pero si encuentra las coloca en el pool global y vuelve a obtener elementos de ahí. Este esquema tiene la ventaja que efectivamente hace una redistribución aunque obviamente se pierde todo el concepto de cola ya que no existe ningún orden y menos en las colas de prioridad.

El pool global es implementado con una cola pero sin respetar ninguna de sus reglas. Adicionalmente el esquema sólo fue probado para máquinas NUMA como la T3D y no en clusters.

4.4. Resumen

En el capítulo se mostró que la eficiencia de un programa paralelo se da en función de su tiempo de ejecución. Para menores tiempos de ejecución es necesario que los procesadores/procesos terminen con su trabajo más o menos al mismo tiempo, ya que el tiempo total lo da el procesador que terminó al último.

Para lograr igualdad en la terminación se puede usar planificación estática, que consiste en asignar de manera definitiva tareas a procesadores. Para ello es necesario encontrar la asignación óptima conociendo los costos de procesamiento de las tareas y la capacidad en los recursos (procesador, memoria, red, etc.). Aún con estos datos la planificación óptima es un problema N-P Completo. Aunado a esto, existen algunos casos donde la estimación de los costos se complica. Por ejemplo, en las aplicaciones dinámicas, en los clusters heterogéneos y en los clusters con multiprogramación. En las aplicaciones dinámicas se desconocen el número de datos a procesar y el costo de cada dato puede ser diferente. En los clusters heterogéneos los nodos con mayor capacidad procesan más tareas que los nodos con menor capacidad. En los clusters con multiprogramación los nodos compartidos con otros usuarios procesan menos tareas que los nodos no compartidos. En los tres casos anteriores una redistribución de las tareas puede mejorar la eficiencia.

El balance de carga hace una redistribución de tareas a tiempo de ejecución, en base a la capacidad de los procesadores. Esto tiene la ventaja que no es necesario conocer de antemano los costos de procesamiento de las tareas o la capacidad de los recursos para hacer la distribución. El balance se hace en base al estado del sistema durante la ejecución. Para definir la forma en que se hace la distribución se han propuesto diferentes políticas incluyendo información, localización, transferencia y selección. La

política de información define lo que será considerado como indicadores de carga y el cómo se recolecta. Los indicadores pueden ser el número de procesos, el porcentaje de uso de CPU, la cola en espera de CPU y el número de datos. En cuanto a la recolección ésta puede ser global o parcial, en la global se colecta la información de carga de todos los procesadores y en la parcial se colecta la información de algunos procesadores. La política de localización define dónde las tareas deben ser transferidas, esto puede ser de forma ciega o basado en la política de información. La política de transferencia indica cuándo se hace la distribución y quién la hace. El cuándo se define por fronteras de carga y el quién puede ser un solo procesador (control centralizado) o cualquier procesador en el sistema (control distribuido).

Las políticas de balance en su conjunto definen un algoritmo de balance de carga. Algunos de los más utilizados son: el global distribuido y el de subasta. En el global distribuido todos los procesadores tienen información del estado de carga de los demás procesadores. Cada vez que ocurre un cambio de carga en un procesador se los notifican al resto de los procesadores. Cuando se decide hacer una redistribución cada procesador verifica su carga, si ésta es alta, busca en su vista global del sistema quién es el procesador menos cargado para enviarle carga. En el algoritmo de subasta un procesador subasta su poder de cómputo cuando se termina su trabajo. Para ello inicia la subasta difundiendo un mensaje a los demás procesadores para ver quién ofrece más por su poder de cómputo. El procesador que ofrece (oferta) más es el que tiene más trabajo. Finalmente el procesador que oferta más cede parte de su trabajo al procesador que subasto (que no tenía carga).

Los ambientes de programación que consideramos en los capítulos anteriores contemplan algunas formas de balance de carga. PVM/MPI hacen una distribución cíclica de procesos a procesadores (localización cíclica). Entre sus limitantes está la no distribución de datos, la cual tiene que ser implementada en todo caso por el programador. TreadMarks, de igual forma que PVM y MPI, hace una distribución cíclica de procesos. En cuanto a los datos, TreadMarks se encarga de mover los datos compartidos hacia los procesadores que los requieren. La distribución consiste en copiar los datos compar-

tidos a los procesadores, los cuales posteriormente pueden ser invalidados si se da una escritura sobre alguna copia (protocolo de coherencia writer-invalidate). Haskell en su distribución GUM implementa una distribución llamada FISH, que consiste en que un procesador que se queda sin carga elige aleatoriamente otro procesador para solicitarle carga, si este procesador tiene carga se la envía, sino se incrementa un contador y se busca otro procesador con carga de manera aleatoria. El contador sirve para detener la pesca, la cual, vuelve a comenzar después de cierto (delay). El contador y el delay son ajustados por el programador lo cual va a estar en función de la aplicación y puede ser complicado el ajuste. Otra limitante es que la selección aleatoria no asegura el procesador con más carga (pez más gordo). MALLBA implementa un esquema de distribución de petición-respuesta asíncrono. En el esquema, cada vez que un procesador se queda sin carga solicita ésta a todos los procesadores, quienes responden con datos o con una respuesta negativa. Debido a que la información del estado de carga se mantiene en todos los procesadores, cada vez que se da una respuesta a una petición ésta se difunde a los demás procesadores para que actualicen su información y cuando pidan sólo lo hagan a los procesadores que tenga carga. Debido a que el esquema es asíncrono, es decir, la petición no espera por la respuesta, se manejan variables como número de petición y peticiones sin contestar. Las respuestas a las peticiones se dan por otra variable o contador que el programador escoge. El esquema tiene la desventaja en cuanto a la selección de la variable contador y en cuanto a la difusión global de la información (todos contra todos). eSkel maneja en su skeleton DEAL una distribución cíclica de datos a procesadores, el esquema es simple pero se limita a aplicaciones sencillas con datos cuyo costo de procesamiento es similar. Esta distribución cíclica no es eficiente cuando se trata de aplicaciones dinámicas, clusters heterogéneos y/o con multiprogramación. SAMBA ofrece 7 diferentes esquemas de distribución los cuales van desde un esquema estático que no es más que una distribución de tareas equitativa y definitiva entre procesadores; hasta esquemas más complejos que incluyen manejo de información global o parcial, toma de decisión de distribución centralizada (un solo procesador) y distribuida (todos los procesadores) y redistribuciones entre un par de procesadores y

entre todos los procesadores. SADTs maneja tres diferentes distribuciones dependiendo de las implementaciones o versiones de sus ADTs. En la implementación replicada los datos más que redistribuirse se copian en todos los procesadores, esta copia se mantiene coherente en todos los procesadores mediante el protocolo writer-update. En la implementación repartida SADTs se encarga de alguna forma de mantener parte de la representación del ADT en todos los procesadores. Adicionalmente cada cierto tiempo hace una redistribución entre todos los procesadores para repartir datos con alta prioridad de procesamiento en todos los procesadores. Esto último es para las colas de prioridad. Finalmente implementa la distribución de Wu-Kung que consiste en pilas locales donde los datos se insertan y eliminan dependiendo de la aplicación, cuando no existen más datos en la pila buscan datos en un pool global. Si existen datos en el pool se obtiene de ahí sino se busca en datos que procesar en las demás pilas remotas. Si existen datos se obtienen de las pilas remotas y se depositan en el pool global de donde posteriormente se toman. El algoritmo termina cuando no hay más datos. La desventaja de este esquema es que se pierde el concepto de cola compartida, además de que el esquema implementado requiere de una arquitectura NUMA. En el pool global, se usa el ADT cola, más otras variables compartidas que llevan la cuenta de los datos en el pool y de los accesos, más otras variables que funcionan como candados.

Capítulo 5

Programación con Listas de Datos para Cómputo Paralelo en Clusters

En este capítulo presentamos nuestra propuesta de un ambiente de programación con listas de datos para cómputo paralelo en clusters.

5.1. Introducción

Como hemos visto en los capítulos anteriores, la programación paralela depende de la arquitectura, del modelo de programación a utilizar [85] y de las características de la misma aplicación.

En el Capítulo 2 enfatizamos que la programación en clusters puede hacerse siguiendo un modelo de programación de bajo nivel como el paso de mensajes, ya que normalmente se obtienen los mejores tiempos de respuesta en la ejecución de aplicaciones. Sin embargo, la programación a bajo nivel puede resultar compleja para algunas personas (químicos, biólogos, físicos, entre otros), principalmente porque necesitan tener conocimientos de cómputo paralelo para implementar los pasos de la programación (particionamiento, distribución, comunicación). Por otro lado, aunque el uso de una herramienta o modelo de alto nivel, como los lenguajes funcionales/lógicos ocultan toda o gran parte de la programación paralela, se requiere de mucha notación matemática en

su interfaz y además no siempre son eficientes. De manera similar, los skeletons considerados de alto nivel dificultan la programación al concentrar en su extenso número de parámetros la definición completa de las aplicaciones. Además en su programación es necesaria una autoidenficación para realizar alguna inicialización o distribución específica (si soy $proceso_0$ ejecuto inicialización, si soy $proceso_1$ ejecuto cierta distribución). En el nivel medio de los modelos de programación, los ADTs (Abstract Data Types) paralelos ofrecen una programación sencilla para ciertas aplicaciones. Los datos de estas aplicaciones deben poder representarse y procesarse como uno de los ADTs disponibles. Con los ADTs es necesario definir el tipo de elemento el cual forma parte de la estructura completa del ADT. Algo más es saber elegir entre los ADTs disponibles por tipo y por implementación. Sin embargo, los ADTs compartidos desarrollados hasta ahora son poco eficientes en arquitecturas débilmente acopladas como los clusters, principalmente por el costo computacional que implica mantener la replicación de datos en la arquitectura paralela. Existen otras implementaciones donde no existe replicación (ADT repartido) pero se pierde el concepto de algunos de sus ADTs en cuanto a consistencia (el orden de lo desencolado no es el mismo que el orden de lo encolado). Si se requiere un orden estricto es necesario poner barreras explícitas por parte del programador.

En nuestra propuesta, el ADT que elegimos es la lista. La lista, a diferencia de las colas, permite una inserción y eliminación en cualquier parte de ella y sin orden de inserción o eliminación. Además de que el procesamiento de diferentes elementos puede hacerse de manera concurrente. También, una lista puede ser dividida y repartida y cada sublista mantiene las mismas propiedades de la lista.

Nuestra propuesta consiste en una biblioteca de programación con listas de datos para cómputo paralelo en clusters. Con esta biblioteca, a la cual llamamos DLML (Data List Management Library por sus siglas en inglés), la programación paralela se reduce a una programación cuasi-secuencial, usando un lenguaje secuencial-imperativo C [55] con llamados a una biblioteca de funciones de manejo de listas.

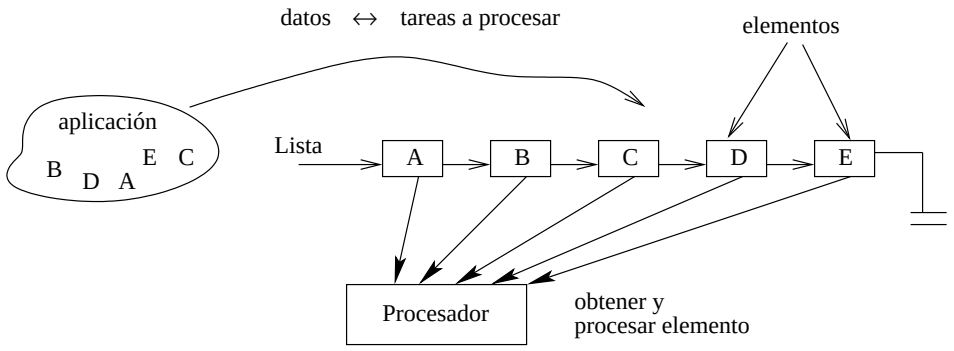

Figura 5.1: Datos de una aplicación vistos como tareas que procesar en una lista.

5.2. Programación con listas de datos

Los datos de una aplicación pueden ser vistos como el conjunto de tareas a procesar, las cuales se pueden organizar como los elementos de una lista. La programación con listas de datos se resume a una inicialización de los elementos de la lista, obtener uno a uno los elementos para procesarlos (mientras existan elementos) y finalmente mostrar resultados, (ver Figura 5.1).

5.2.1. Metodología

Un programador puede desarrollar aplicaciones basadas en listas siguiendo estos pasos:

1. Distinguir los datos de la aplicación y determinar la forma de organizarlos en listas.

2. Definir la granularidad o cantidad de elementos en la lista. Indirectamente la cantidad de elementos va en función del contenido de cada elemento. La granularidad es fina cuando los datos se organizan en el mayor número de elementos.

3. Declarar todos los tipos de datos dentro de la estructura que define un elemento de la lista.

4. Definir el código o procedimiento de inicialización de cada elemento de la lista.

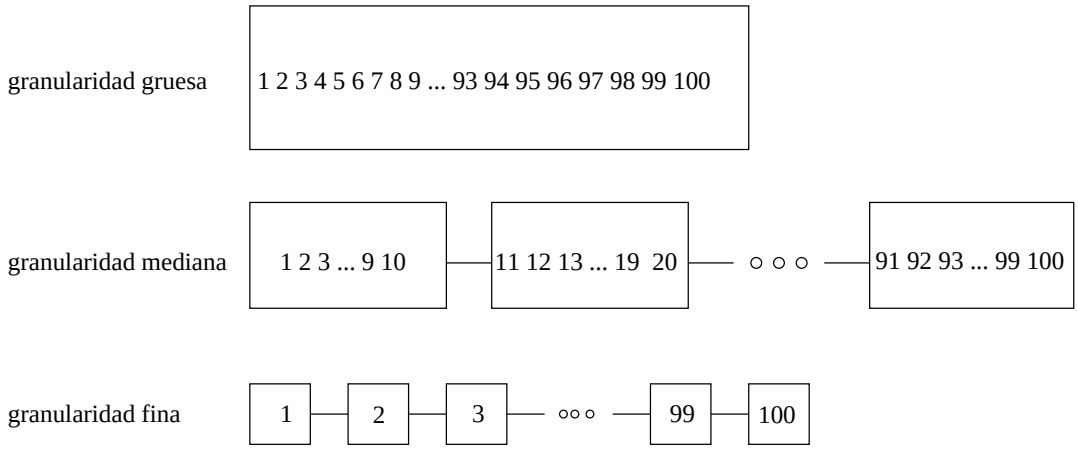

Figura 5.2: Diferentes granularidades para una lista de 100 números.

5. Definir el código/procedimiento para procesar cada elemento de lista. Este código en general estará dentro de un loop (lazo) e incluye un `get()`.

6. Imprimir resultados, cerrar archivos y finalizar procesos, entre otros.

Para explicar mejor la metodología nos auxiliamos de un ejemplo sencillo, la suma de 100 números. En la suma de 100 números los datos que se pueden colocar en las listas son precisamente los 100 números (paso 1). No obstante, existen diferentes formas de organizarlos en la lista. Desde una granularidad gruesa hasta una granularidad fina tal y como se muestra en la Figura 5.2. Para este caso en especial elegimos una granularidad fina (paso 2).

En el paso 3 de la metodología se define la estructura del elemento en base a la granularidad elegida. La estructura para granularidad fina es:

```
typedef struct estructura_elemento elemento;
struct estructura_elemento   {
  int número;   // si fuera mediana sería  >>  int número[10];
  elemento *siguiente;
};
elemento * Lista;
```

El paso 4 de la inicialización sería un código donde se le asigna un valor a los números y se insertan en la lista. Para el caso de granularidad fina, el código sería el siguiente:

```
for (i=0;i<100;i++)   {
  p=new(elemento);       // se reserva un nuevo elemento para la lista
  p.número=random();     // se asigna el campo número con un valor aleatorio
  insertar(&Lista,p)     // se inserta el elemento en la lista
}
```

En el paso 5, se recuperan los elementos, se hace el procesamiento y se guarda el resultado en algún lugar. Todo esto, mientras haya elementos en la lista.

```
while (obtener(&Lista,&p))   {
  suma_parcial=suma_parcial+p.número;  // suma parcial se inicializa a 0
}
```

Finalmente en el paso 6 se imprime el resultado

```
printf (La suma total es: $suma_parcial);
```

Podemos notar que dependiendo de la granularidad elegida (paso 2), se determinan los siguientes pasos de la metodología (paso 3 al 5). A continuación definimos algunos tipos de aplicaciones que se pueden modelar con listas.

5.2.2. Aplicaciones

De entrada, los tipos de aplicaciones que se pueden modelar con listas son aplicaciones regulares e irregulares. En las regulares se conoce de antemano todos los datos a procesar; mientras que en las irregulares los datos se van descubriendo durante la ejecución y/o la granularidad es variable.

- Aplicaciones regulares pueden ser las Numéricas o de Procesamiento Digital de Imágenes (PDI).
    - Numéricas. suma de n números, suma de matrices (apéndice B.1) y multiplicación de matrices.
    - PDI. cualquier aplicación que requiera el procesamiento de pixels o áreas de una imagen (apéndice B.2).

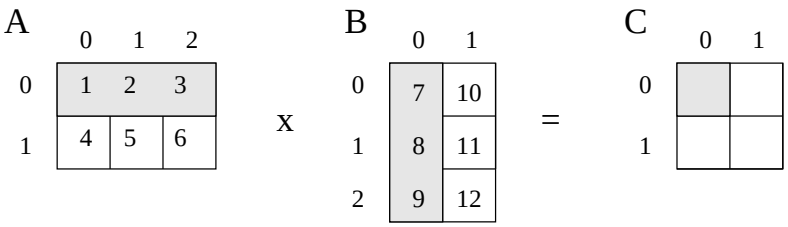

Figura 5.3: Multiplicación de dos matrices A y B.

- Aplicaciones irregulares pueden ser Branch & Bound (B&B) y Dividir para Vencer (D&C).
  - B&B. El agente viajero, el problema de las N-Reinas.
  - D&C. algoritmos de ordenamiento.

A continuación son descritas algunas de estas aplicaciones.

Multiplicación de matrices

La multiplicación de matrices se define como:

$$C[i,j] = \sum_{k=1}^{n} A[i,k] * B[k,j] \tag{5.1}$$

Donde A y B son las matrices a multiplicar y C es la matriz resultante. k es un valor que sirve como contador de columnas en la matriz A y contador de renglones en la matriz B. En una multiplicación si la matriz A es de tamaño $m \times o$ y B es de tamaño $p \times n$, la matriz resultante C es de tamaño $m \times n$, donde o y p deben ser iguales.

Una granularidad fina para la multiplicación se puede ver en la Figura 5.3. Para encontrar el resultado en C en la posición (0,0), de la fórmula de la multiplicación de matrices es necesario el renglón 0 y la columna 0. Esto da como un elemento de la lista a {(0,0),{1,2,3},{7,8,9}}. Otro elemento de la lista es el que obtiene la multiplicaciones en la coordenada (0,1) donde se necesita el renglón cero y la columna uno. En este caso el elemento de la lista es {(0,1),{1,2,3},{10,11,12}}. En total se tienen 4 elementos en la lista. Obviamente puede haber otra granularidad por ejemplo mediana,

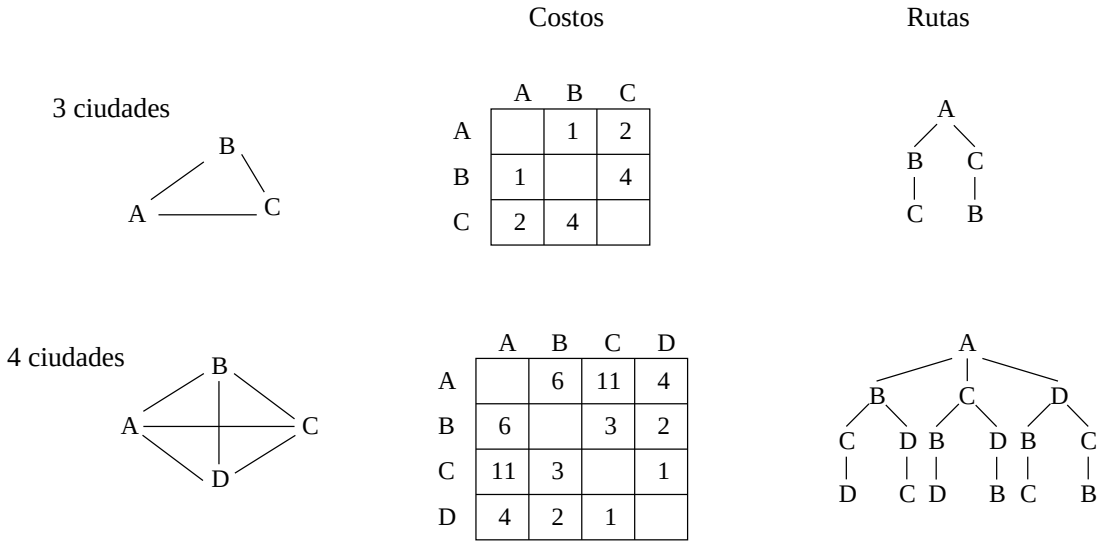

Figura 5.4: Agente viajero para 3 y 4 ciudades, con costos y rutas.

donde con cada elemento se encuentra un renglón completo del resultado. El contenido en ese caso de cada elemento de la lista es el número de renglón, los datos del número de renglón de la matriz A y toda la matriz B. Los elementos de la lista serian: {0,{1,2,3},{{7,10},{8,11},{9,12}}} y {1,{4,5,6},{{7,10},{8,11},{9,12}}}. Una vez elegida la granularidad, se define la estructura adecuada que soporte al elemento de la lista y se insertan en ella. Posteriormente se retiran uno a uno los elementos de la lista, se procesan y se guarda el resultado respectivo en la matriz C.

Agente viajero

El problema del agente viajero consiste en encontrar el mejor recorrido visitando n ciudades. Un recorrido se forma al ir de una ciudad a otra (sin regresar a la misma) e ir sumando los costos que genera ese sub-recorrido. El mejor recorrido será el que obtenga el menor costo total. En la Figura 5.4, se muestran dos ejemplos para 3 y 4 ciudades. Se muestran también los costos para ir de una ciudad a otra mediante una matriz. Para encontrar la mejor ruta una opción es encontrar TODAS las rutas posibles y escoger la menor. En la misma figura se muestran en un árbol todas las rutas posibles para 3 y 4 ciudades. Haciendo números el total de rutas para recorrer n ciudades es $(n-1)!$,

donde no es difícil ver que para valores grandes de n la búsqueda exhaustiva resulta prohibitiva.

Branch & Bound es un método que evita hacer una búsqueda exhaustiva, eliminando aquellas posibles soluciones que no sean prometedoras. En el agente viajero una solución no prometedora es aquella cuyo costo actual es mayor que el costo del mejor recorrido encontrado hasta ese momento. Volviendo a la Figura 5.4, consideremos el problema para 4 ciudades cuyos recorridos son:

```
1. A B C D    10
2. A B D C     9
3. A C B D    16
4. A C D B    14
5. A D B C     9
6. A D C B     8
```

Con Branch&Bound la secuencia sería la siguiente: Inicialmente no se tiene ningún recorrido por lo que $mejor = \infty$. De la Figura 5.5 primero se recorre la rama ABCD cuyo recorrido es 10. Este valor reduce al mejor recorrido conocido por lo que ahora $mejor = 10$. Posteriormente se recorre la rama ABDC y se obtiene un nuevo recorrido que reduce el mejor conocido, ahora $mejor = 9$. A continuación se intenta recorrer AC, pero el costo es superior al mejor conocido, por lo que ya no se continua, es decir, se poda esa posible solución. Después se recorre la rama ADBC pero el recorrido es igual al mejor conocido y no se hace nada. Finalmente se recorre ADCB y el recorrido es 8 que reduce la mejor solución y la búsqueda termina.

La generación anterior se muestra con listas en la misma Figura 5.5 (lado derecho). En cada lista se guarda el recorrido actual y en la última posición se guarda el costo actual. Inicialmente se está en la ciudad A y el costo es 0. Posteriormente el elemento se toma y este se descompone en las posibles soluciones AB, AC y AD, las cuales se insertan en la lista con su respectivo costo. A continuación se toma el primer elemento de la lista AB el cual se descompone en las rutas ABD y ABD las cuales se insertan en la lista. Después se toma el primer elemento el cual se descompone en ABCD y se termina un recorrido siendo este menor al mejor conocido. Posteriormente se toma el valor ABD

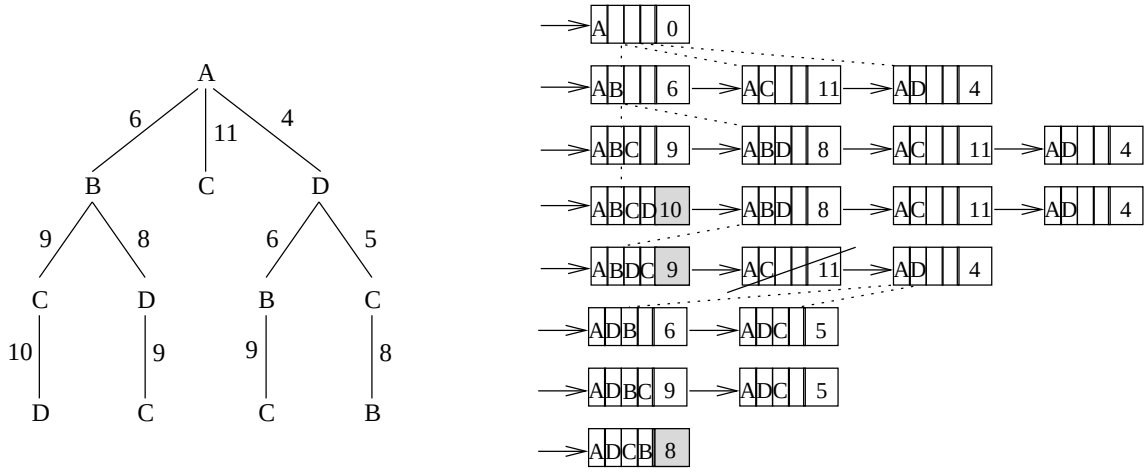

Figura 5.5: Recorrido del agente viajero con Branch&Bound en representación de Árbol y de Lista.

el cual genera ABDC y se encuentra otra mejor solución, ahora con el valor de 8. Se toma el siguiente elemento AC pero éste es mayor que la mejor conocida y elimina, es decir, ya no expande ni se inserta nada en la lista. El procedimiento continua hasta encontrar la solución ADCB que es la mejor con el valor de 8.

```
1.  int best=999999999;
2.
3.  p=new(elemento);
4.  p.tour[0]=A;   p.costo=0;
5.  inserta(&Lista,p);
6.
7.  while (obtener (&Lista,&p))  {
8.    if(p.costo < best)
9.      if(final_recorrido())
10.       best=p.costo;  guarda_recorrido();
11.     else  {
12.       expander(p.tour,izq,der)
13.       insertar(&Lista,izq);
14.       insertar(&Lista,der);
15.     }
16. }
```

El código anterior se explica a continuación. En la línea 1 se declara la variable *best* que contiene el menor costo conocido. Inicialmente tiene un valor muy grande. En la línea 3 se reserva memoria para un nuevo elemento de la lista. La lista tiene dos campos: un arreglo de tamaño n donde se guarda el tour actual y el otro campo con el costo. En la línea 4 se inicializan estos campos con el valor A y 0 respectivamente. Posteriormente se inserta en la lista este primer elemento. A continuación de la línea 7 a la línea 16 se entra en un ciclo mientras existan elementos. El algoritmo termina cuando ya no hay más elementos. Una vez obtenido el elemento en el ciclo while (línea 7) se verifica si su costo actual es menor que el costo global (best), si no es así se poda (se elimina y se va por el siguiente elemento); pero si es menor puede ser que sea el final del recorrido en cuyo caso se actualiza la variable *best* y se guarda el recorrido(línea 9 y 10). En caso contrario, el elemento obtenido se divide en nuevas posibles soluciones las cuales se insertan en la lista (línea 12-14).

Como notamos el algoritmo del viajero es una aplicación irregular donde los datos a procesar se van conociendo a tiempo de ejecución. Dicho de otras palabras en las aplicaciones regulares TODOS los elementos a procesador se insertan en la lista desde el principio y se van consumiendo hasta terminar; mientras que en las irregulares se comienza con un solo elemento en la lista y durante su ejecución la lista crece y decrece hasta finalmente terminar.

N-Reinas

Otro ejemplo de aplicación irregular es el problema de las N-Reinas [18]. El problema consiste en colocar en N reinas en una tablero de NxN sin que éstas se ataquen entre sí. Para ello, es posible modelar las posibles soluciones en un árbol de búsqueda, donde se van insertando las posibles soluciones y eliminando las que no pueden ser. Esto es similar al problema del agente viajero con la diferencia que aquí se encuentran TODAS las soluciones; en el agente viajero se encuentra sólo la mejor solución (el camino con menor costo). En la Figura 5.6 se muestran las dos soluciones para N=4, {2,4,1,3} y {3,1,4,2}. Para ello se generó el espacio de búsqueda que se muestra en la misma

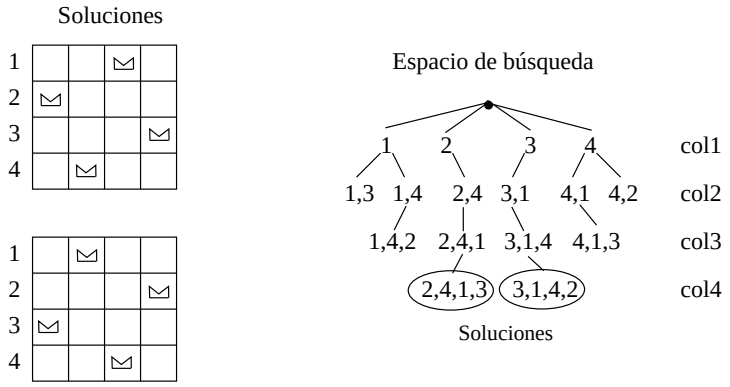

Figura 5.6: Soluciones y espacio de búsqueda para el problema de las N-reinas con N=4.

figura. En el espacio de búsqueda cada nivel de profundidad define la columna y los números que aparecen en el árbol corresponden al renglón colocado en esa columna. En el primer nivel (columna1 col1) se muestra que se puede colocar una reina en la col1-ren1. Pero también otra posible solución a explorar es colocar una reina en la col1-ren2. Así sucesivamente hasta col1-ren4. En el segundo nivel (columna 2), se nota que la solución ren1 genera la posible solución ren(1,3) y la ren(1,4). La solución ren(1,2) no se genera porque es inválida. Este procedimiento continúa hasta encontrar las dos únicas soluciones para N=4, ren(2,4,1,3) y ren(3,1,4,2).

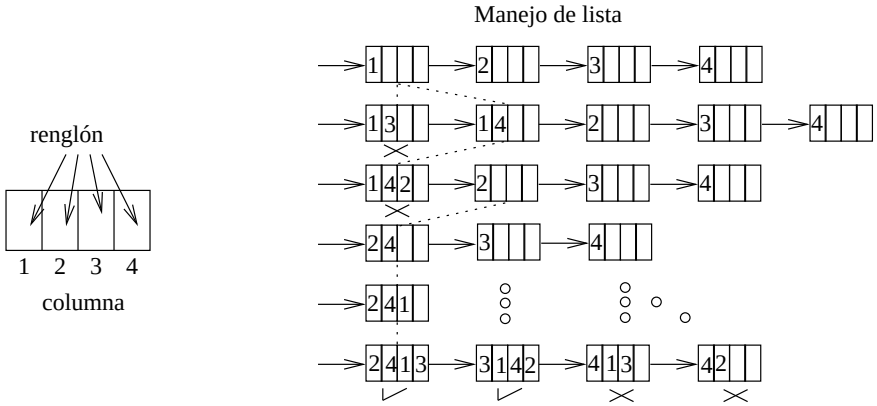

Figura 5.7: Inserción y eliminación de posibles soluciones en el problema de las N-reinas.

Este espacio de búsqueda se puede representar mediante una lista de datos. En la lista, los datos contenidos por cada elemento pueden representar una posible solución

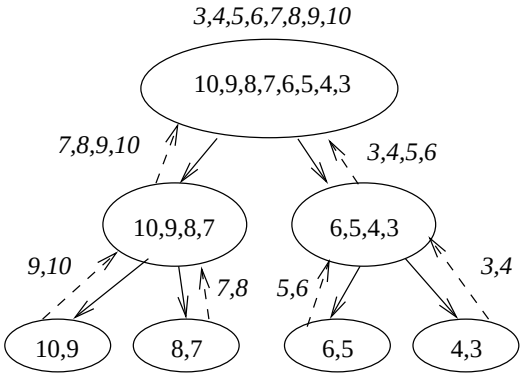

Figura 5.8: Ordenamiento MergeSort mediante un método DPV.

a explorar. Para explicar este comportamiento mostramos la Figura 5.7. En la Figura los elementos de la lista representan una posible solución. Inicialmente son posibles soluciones para la columna 1 los renglones 1,2,3 y 4. para ello se insertan 4 items. Después se toma el primer elemento de la lista con ren1, el cual se puede descomponer en dos posibles soluciones ren(1,3) y ren(1,4). Éstas dos posibles soluciones se insertan en la lista y continua el procesamiento obteniendo el primer elemento, ahora se trata de ren(1,3). Al analizar esta posible solución se nota que ya no se pueden generar o continuar posibles soluciones a partir de ésta, ren(1,3,2) o ren(1,3,4) son soluciones inválidas, por eso aparece una equis (x) abajo de la referida solución. Después se toma el siguiente elemento de la lista ren(1,4), el cual genera la posible solución ren(1,4,2) la cual se inserta nuevamente en la lista. El proceso continua hasta tener una lista vacía encontrándose todas las soluciones.

Sort

Los sorts consisten en ordenar una secuencia de números. Un algoritmo de ordenamiento es el MergeSort es un ejemplo del paradigma dividir para vencer (DPV)[1].

El algoritmo divide el conjunto total de números hasta cierto límite. Posteriormen-

[1]DPV consiste en la descomposición de un problema de tamaño n en problemas más pequeños, para que a partir de la solución de dichos problemas se construya con facilidad la solución al problema completo.

te se ordena ese subconjunto y se mezcla con otro subconjunto ordenado. Un ejemplo muy sencillo se muestra en la Figura 5.8. Se observa que el problema completo {10,9,8,7,6,5,4,3} se divide primero en dos subproblemas {10,9,8,7},{6,5,4,3}. Posteriormente en una segunda vuelta se vuelven a dividir hasta llegar a cuatro subproblemas {10,9},{8,7},{6,5},{4,3}. A partir de ahí se resuelve el subproblema que es un ordenamiento trivial. Una vez ordenado se mezcla con otro conjunto ya ordenado lo que genera un conjunto más grande. El procedimiento se repite hasta resolver el problema completo.

El problema llevado a una programación con listas es el siguiente: En el paso 1, los datos del problema son los números desordenados. En el paso 2 se puede elegir una granularidad fina con dos números por elemento de la lista. Una vez definida la granularidad se define la estructura adecuada (paso 3) y se insertan los cuatro elementos en la lista (paso 4). El siguiente código define con más detalle los pasos restantes (paso 5 y 6).

```
1.  int *lista_ordenada=NULL;
2.  int lista_parcial[2] =NULL;
3.
4.  INSERTAR_CUATRO_ELEMENTOS_EN_LISTA;
5.
6.  while(obtener(&Lista,&p))    {
7.    ordenar(p.números,lista_parcial);   //donde números es un arreglo de tamaño 2
8.    mezclar(lista_ordenada,lista_parcial);
9.  }
10. imprimir(lista_ordenada);
```

Del código se observa que una vez insertados los elementos en la lista (línea 4), se entra en un ciclo (línea 6-8) donde se toma el primero elemento, se ordena y se guarda el resultado en el arreglo *lista_parcial*. Posteriormente *lista_parcial* se mezcla con la *lista_ordenada* hasta ese momento. El procedimiento se repite hasta terminar los elementos de la lista. Finalmente en la línea 10 se imprime el resultado.

5.3. Programación basada en listas con DLML

En la sección anterior mostramos cómo la programación con listas se puede adaptar a varias aplicaciones. A continuación veremos cómo la programación con listas también se puede extender a un *sistema paralelo*, buscando como objetivo la reducción de los tiempos de respuesta.

5.3.1. Ambiente DLML

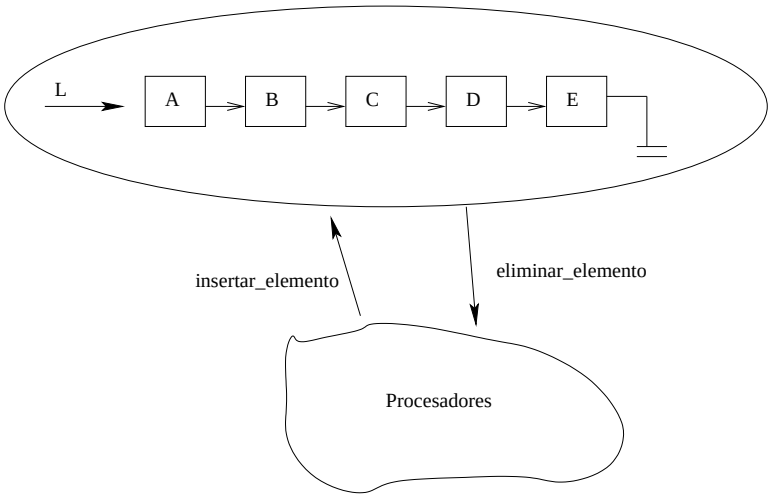

Figura 5.9: Lista en un sistema paralelo.

Hemos visto que las listas no tienen orden, sustentado en que la inserción y eliminación de elementos se da en cualquier parte. Esta "falta" de orden permite que las listas se puedan dividir y moverse fácilmente. Teniendo esto en cuenta, la representación de la lista se puede llevar a un sistema paralelo como se muestra en la Figura 5.9.

En la figura un conjunto de procesadores puede insertar y eliminar elementos de una lista sin alterar ni violar ninguna de sus propiedades. Este razonamiento nos lleva al ambiente de programación DLML (Data List Management Library).

DLML es un ambiente para el desarrollo de aplicaciones paralelas y está basado en la programación con listas. Al igual que la clásica programación con listas (CPL), con DLML el programador sólo tiene que identificar los datos a ser procesados en su

aplicación, organizarlos en una lista y manejarlos con las funciones del ADT lista. No obstante, las funciones de DLML son paralelas y se encargan de manera transparente de distribuir los datos (elementos de la lista) entre los procesadores de acuerdo a la capacidad de éstos. [30, 36, 37, 62].

DLML esta organizado bajo el modelo de ejecución SPMD [99] lo cual simplifica la programación al definir un solo programa. En el programa el programador utiliza normalmente un ciclo principal dentro del cual hace los llamados a las funciones del ADT lista (insertar, eliminar) para procesar los datos. Al final, el programa termina cuando ya no existen más datos que procesar. Todo lo anterior es similar a la clásica programación con listas que presentamos en la Sección 5.2. No obstante, existe una diferencia. Con CPL se tiene una lista única; con DLML se tienen muchas listas, y estas se encuentran distribuidas en todo el cluster. En sí, el número de listas es igual al número de procesadores. Al tener las listas distribuidas, los resultados parciales también se encuentran distribuidos, por lo que al final es necesario colectar los datos. Esta colecta se cubre con las funciones de DLML.

5.3.2. Esqueleto de un programa DLML

Para hacer un programa usando DLML lo primero es tener una aplicación como las que se describen en la subsección 5.2.2. La aplicación debe poder representar sus datos en listas, y poder ser procesados de manera independiente de cualquier otro dato, es decir, en cualquier procesador y en cualquier orden.

Una vez identificados los datos, se procede a definir este dato como parte de un elemento o item que conforma la *Lista*. La definición se hace dentro el archivo *info.h* que forma parte del ambiente DLML.

```
typedef struct DLML_item Info;
struct DLML_item
{
        // agregar AQUI el tipo de dato definido
};
```

Por ejemplo si lo datos de la lista son número enteros, se debe agregar entre las llaves la línea

```
int numero;
```

Otros ejemplos de definición pueden ser:

```
float numero;
def_estructura valor;  // donde def_estructura tiene más de un campo
```

Después de hacer la definición en el archivo info.h, lo que sigue es hacer el programa. Un programa en DLML (como cualquier programa en C), necesita de cabeceras, en este caso:

```
#include "dlml.h"
```

Enseguida vienen las declaraciones de una variable tipo *Lista* y otro tipo *Info*. La primera funciona como cabecera principal a lo que es la lista y la segunda como ya se definió en el archivo *info.h* es para conformar el contenido de cada elemento de la lista. Ambas se declaran de forma global por facilidad, no obstante, pueden ser declaradas de manera local y ser pasadas como parámetro a las funciones que lo necesiten. La declaración es la siguiente:

```
Lista L;
Info elem;
```

A continuación viene lo que es el cuerpo principal del programa (*main()*), cuyo interior se debe encontrar delimitado por las instrucciones:

```
main()
{
  DLML_Init();
  ...
  ...
  DLML_Finalize();
}
```

Entre estos delimitadores lo que va normalmente son las inserciones y eliminaciones a lista. En las eliminaciones es recomendable incluirlas en un ciclo *while*, hasta terminar de procesar todos los elementos de la lista. Puede suceder que no se eliminen los elementos en ciclo while, pero el programa termina y elimina los elementos restantes con la llamada a *DLML_Finalize()*. A continuación vemos un ejemplo típico de este código.

```
1.  Inicializa(&elem);         // introduce datos en el elemento de la lista
2.  DLML_Insert(&L,elem);      // inserta el elemento en la lista L
3.  while(DLML_Get(&L,&elem))  //si existe un elemento, lo obtiene y lo procesa
4. {
    // procesamiento sobre el elemento obtenido de la lista
5. }
```

En el paso 1 se se inicializan los datos del elemento (se les asigna algún valor). Una vez inicializados se inserta el elemento en la lista L (paso 2). En el paso 3, si existe un elemento de la lista L, lo obtiene y lo procesa (paso 4).

Por último, normalmente es necesaria una recopilación de los resultados parciales. Si este es el caso, esto se hace mediante alguna instrucción del tipo recopilación, definidas posteriormente en la Tabla 5.1.

Una vez terminado el programa, lo que sigue es la compilación y ejecución de la aplicación que se describirán en la subsección de instalación de DLML A.4.

5.4. Formalización de DLML en BNF

La formalización en BNF (Backus Normal Form) del esqueleto anterior de DLML se puede representar de la siguiente forma:

```
<programa_dlml>   ::= <encabezado><variables><cuerpo>
<encabezado>      ::= #include \dlml.h"
<variables>       ::= <variable_lista>
                      <variable_elemento>
<variable_lista>  ::= Lista <identificador_lista>
```

```
<variable_elemento>::= Info <identificador_elemento>
<cuerpo>           ::= main (int argc, char **argv)
                       {
                        <cuerpo_dlml>
                       }
<cuerpo_dlml>       ::= <inicializacion_dlml>
                        <procesamiento_dlml>
                        <recopilacion_dlml>
                        <finalizacion_dlml>
<inicializacion_dlml>::=  DLML_Init(&<identificador_lista>);
<procesamiento_dlml> ::= (DLML_Only_one {)?
                        <inicializacion_de_elementos_de_lista>
                        <insercion_inicial>
                        }?
                        <ciclo_dlml>
<inicializacion_de_elementos_de_lista>::= // sintaxis de C de asignación de
                                          valores a los campos de un elemento
<insercion_inicial>::=(DLML_Insert(&<identificador_lista>,<identificador_elemento>))+
<ciclo_dlml>      ::=while(DLML_Get(&<identificador_lista>,&<identificador_elemento>)
                     {
                      <insercion_local> |
                      <procesamento_local_del_elemento_obtenido> |
                      <intercambio_resultado_local>
                     }
<procesamiento_local_obtenido>::= // sintaxis de C para procesamiento específico de
                                  cada elemento dependiendo de la aplicación
<recopilacion>      ::= <ALL for ALL> | <reduccion>
<ALL for ALL>       ::= DLML_Exchange (<var_entrada>,<var_salida>)
<reduccion>         ::= <funcion_reduccion>(<var_entrada>.<var_salida>)
<funcion_reduccion> ::= DLML_Reduce_Add |
                        DLML_Reduce_Average |
                        DLML_Reduce_Max
<finalizacion_dlml>     ::= <impresión_resultado><terminacion>
<impresión_resultado>   ::= //impresión resultados
<terminacion>           ::= DLML_Finalize():
```

5.5. Ejemplos de programación con DLML

En esta sección presentamos algunos ejemplos del desarrollo de aplicaciones regulares e irregulares con DLML.

5.5.1. Aplicaciones regulares

Sólo para recordar, en las aplicaciones regulares se conocen de antemano todos los datos a procesar, un ejemplo de ello, es la suma de N números, pero ahora en paralelo con DLML.

suma de n números

En el ejemplo, los elementos de la lista son los números a ser sumados. El código se presenta a continuación.

```
1   #include <stdio.h>
2   #include "DLML.h"
3   #define N 100   // define algun valor para N
4
5   Lista L;
6   Info  elem;
7   main() {
8    int suma_parcial=0, dato;
9    int suma_total=0;
10   DLML_Init(&L);
11   DLML_Only_one {         //generación de los datos
12    for(i=0;i<N;i++){
13     elem.dato=aleatorio()
14     DLML_Insert(&L,&elem);
15    }
16   } // fin de DLML_Only_one
17   while(DLML_Get(&L,&elem) {
18     suma_parcial+=dato;
19  }  // fin del while
```

```
20    DLML_Reduce_Add(suma_parcial,suma_total);
21    DLML_Only_one   {
22     printf("La suma total es %d\n",suma_total);
23    }  // fin del DLML_Only_one
24    DLML_Finalize();
25   }
```

Se recordará que DLML utiliza el modelo de ejecución SPMD. Por lo tanto el código mostrado se carga en cada nodo que corre la aplicación. Sin embargo, no todo el código se ejecuta en cada procesador. Por ejemplo, el código dentro de la primitiva `DLML_Only_one` (línea 11-15) lo ejecuta un solo procesador, que en esta caso se encarga de inicializar los datos (asignarles un valor) e insertarlos en la lista. Pasado la inicialización se tiene una lista con 100 elementos. Cabe mencionar que esta inicialización podrían hacerla todos los procesadores en vez de uno. En este caso las líneas 11-16 se pueden sustituir por el siguiente código.

```
for(i=0;i<(N/Procesadores),i++)  {  // la variable especial Procesadores, define el
 elem.dato=aleatorio();                // número de procesadores usados del cluster
 DLML_Insert(&L,&elem);
}
```

En este caso cada procesador de manera paralela genera su propia lista, cada una con 10 elementos.

Posteriormente en la línea 17, todos los procesadores intentan obtener un elemento de la lista mediante la función $DLML\_Get()$. Este elemento puede ser obtenido de la lista local o de una lista remota. Si la inicialización la hizo un solo procesador, los demás procesadores buscan en la lista remota, todo esto de manera transparente hacia el programador. Una vez obtenido se realiza la suma parcial y se vuelve a repetir el proceso. Mientras un procesador no obtenga ningún dato, la ejecución de éste se encuentra bloqueada. Una vez que no hay datos que procesar en todo el sistema, la función $DLML\_Get()$ devuelve un $FALSE$ a los procesadores bloqueados y salen del ciclo.

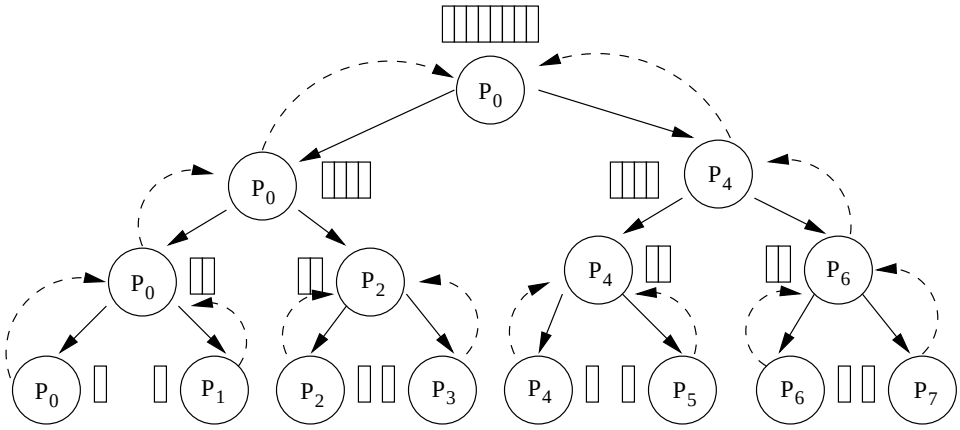

Figura 5.10: Ordenamiento paralelo MergeSort mediante DPV.

En la línea 20 se utiliza una función colectiva que recopila el valor de una variable (en este caso `suma_parcial`) de todos los procesadores y SUMA estos valores. El resultado lo deja en el procesador cero. Posteriormente en la línea 21 un solo procesador (el procesador cero), imprime el resultado final. Finalmente el llamado de la línea 25 devuelve el control al sistema operativo.

5.5.2. Aplicaciones irregulares

MergeSort paralelo

E MergeSort [41] es un algoritmo de ordenamiento de números que usa un método dividir para vencer. Con este método inicialmente un procesador tiene todos los datos a ordenar, como se muestra en la Figura 5.10, donde el procesador P_0 tiene los datos o números a ordenar. En el siguiente paso divide los datos y envía la mitad de estos a otro procesador (P_4). Ambos procesadores vuelven a dividir y enviar hasta que los datos quedan distribuidos entre todos los procesadores disponibles, (P_0 - P_7). Una vez que cada procesador tiene sus datos, los ordena por algún método, para después enviar los números ordenados a su proceso padre. Por ejemplo, en la Figura 5.10 P_1 envía su números ordenados a P_0, quien a su vez se encarga de mezclar ambos arreglos (el propio y el recibido por P_1). De manera similar los hacen los procesadores P_3, P_5 y P_7. Una vez

que se ha completado ese nivel, los procesos envían nuevamente el conjunto de números ordenado a su procesador padre, y así sucesivamente hasta terminar.

Cabe mencionar que aunque la definición del algoritmo parece sencilla, la implementación paralela se vuelve complicada, como se muestra en el siguiente código desarrollado en MPI y siguiendo una distribución explícita de datos (DE).

```
1.  dpv_parallel_merge(int my_id, int nproc)   {
2.    int tama, tamaño, i, cont=0;
3.    int *s, *s1, *s2, *hijos;
4.    if (my_id == 0)   {
5.      tamaño = MAX;
6.      inicializa_arreglo(s,tamaño);
7.    }
8.    if (my_id!=0)
9.      Recv(s,PARENT);
10.   if(es_par(my_id))   {  // sólo los pares
11.     tama=tamaño;
12.     while(tama > PARTICION MINIMA DE DATOS)   {
13.       divide(&s,tama,&s1,&s2);
14.       tama=tama/2;
15.       Send(s2,tama,hijos[cont]);
16.       s=s1;
17.       cont++
18.     }
19.     ORDENAMIENTO_PARCIAL(s,tama);
20.     for(i=cont-1;i>=0;i--)   {
21.       Recv(&arreglo_parcial,hijos[i])
22.       MEZCLO(s,arreglo_parcial)
23.     }
24.   }    // fin de si_es_par
25.   else   // si id no_es_par
26.     ORDENAMIENTO_PARCIAL(s,tamaño);
27.    if(my_id!=0)  Send (s,tamaño,PARENT);
28.    if(my_id==0) IMPRIME_ARREGLO_ORDENADO;
```

```
29.  }
```

Del código se puede notar la identificación de procesos, como en la línea 4 donde el proceso 0 inicializa la lista. Después procede a hacer la distribución. Para ello todos los procesos a excepción del proceso 0 están en espera de recibir datos (línea 8-9). Los procesos pares son los únicos que dividen sus datos y los envían a los procesos definidos en un arreglo (línea 10-18). Una vez que ha terminado la distribución todos los procesos impares y pares hacen el ordenamiento parcial (línea 19 y 26 respectivamente). El arreglo ya ordenado es enviado a los procesos padres (línea 26) quienes lo reciben y lo mezclan (línea 20-23). Finalmente el proceso cero se encarga de imprimir el resultado.

Esta distribución explícita de datos, se evita usando DLML como se muestra en el siguiente código.

```
1. Parallel_Merge_Sort(List * L, int processid)
2. { sorted_list = Empty;
3.   DLML_Only_one
4.     DLML_Insert(arreglo_datos,&L);
5.   while (DLML_Get(L))
6.       Merge( &sorted_list, &L );
7.   Coordinator_Gathers_and_Merges_Partial_Results( procid );
8.   return( sorted_list );
9. }
```

Con DLML un proceso se encarga de hacer la inicialización (línea 3 y 4). A partir de ahí todos los procesos intentan obtener elementos y mezclarlos en una lista ordenada parcial (línea 5 y 6). Este proceso sigue hasta terminar de mezclar los elementos. Cabe mencionar que DLML se encarga de hacer la distribución de forma transparente y adecuada a la capacidad de los nodos. Finalmente como los arreglos ordenados se encuentran distribuidos, un coordinador se encarga de juntar los arreglos y mezclarlos para formar el arreglo completo ordenado.

N-reinas

El problema de las N-Reinas se resuelve con el algoritmo siguiente.

```
1  function Non-attacking-queens(List *L) {
2    queen = 1; chessboard[1<=i<=N] = 0; nr_solutions = 0;
3    DLML_Only_One {
4      q = Create_Element(queen,chessboard); // la función crea una primer solución
5      DLML_Insert(&L, q);
6    }
7    while( DLML_Get(&L, &B) ) {
8      for(column=1;column <= N; column++ ) { // genera elementos
9        if (! Attacked(B.queen,column,B.chessboard) )  //si la solución parcial es
10         if (B.queen < N) {                   // no atacada y no se han terminado
11           B.chessboard[B.queen] = column;    // los renglones, se crea el
12           q = Create_Element(B.queen+1,B.chessboard);  // elemento de la lista
13           DLML_Insert(&L,q);                           // y se inserta
14         } else
15           nr_solutions= nr_solutions +1;// la solución es completa y se incrementa
16     } // N-queens have been placed      // el contador
17   }
18   final_result= DLML_Reduce_Add( nr_solutions );
19   return final_result;
20 }
```

Recordemos que DLML sigue el modelo de ejecución SPMD, es decir, el código anterior lo ejecutan todos los procesos. En las líneas 3-6 un solo proceso crea la primera posible solución la cual es insertada como primer elemento en la lista. Esta posible solución (y todas las demás) se componen de un arreglo y una variable que indica la siguiente reina a colocar. En el arreglo se guarda en cada posición (renglón) un número de columna. Esta combinación de columnas es lo que forma las posibles soluciones. En la línea 7 los procesos o procesadores intentan obtener elementos (posibles soluciones) a explorar, pero la primera vez sólo un proceso `x` obtiene el único elemento que existe. Los demás procesos al no poder obtener elementos de la lista, permanecen bloqueados.

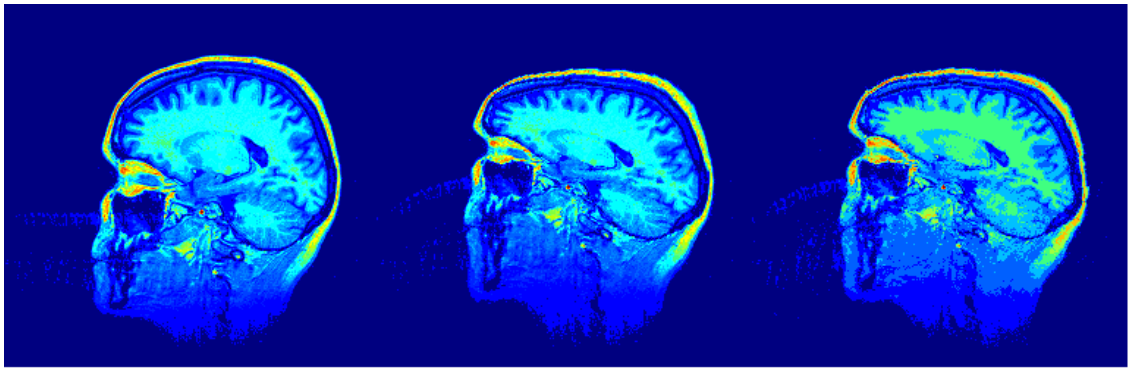

Figura 5.11: Alineación de imágenes por NRIR.

El proceso `x` que obtuvo el elemento, hace el procesamiento y verifica si puede generar otras posibles soluciones parciales, si es así, el proceso las crea (línea 12) y las inserta en la lista (línea 13). A partir de ahí los demás procesadores pueden obtener elementos de la lista (posibles soluciones) e ir insertando o eliminando soluciones. Al final cada uno de los procesos tienen parte de las soluciones encontradas, por lo que para encontrar el total, éstas se suman globalmente en la línea 18.

NRIR

Otras aplicación irregular que se puede programar con DLML es el procesamiento llamado "Registro de imágenes no rígido" [25, 78] donde se tiene un par de imágenes *fuente* y *destino* a las cuales se les hace cierto procesamiento. El algoritmo o el procesamiento consiste en transformar la imagen *fuente* para que sea alineada (similar) a la imagen *destino*, mediante la intensidad de los pixels.

En la Figura 5.11 se muestran la imagen *fuente*, *destino* y *alineada*. Para llegar a la imagen alineada se implementa un algoritmo dividir para vencer descrito a continuación.

Inicialmente la imagen completa (fuente) se procesa y el resultado es comparado con la imagen destino. La comparación consiste en la similitud de las imágenes a nivel de pixels. Si la comparación es mínima a un cierto porcentaje el procesamiento termina,

sino la imagen se divide en 4 partes y comienza el mismo procesamiento para cada cuarto de la imagen. Cabe mencionar que cada cuarto se puede volver a dividir si la comparación en esa parte de la imagen no esta por abajo de un mínimo y así sucesivamente. Del algoritmo se nota que dependiendo de la subdivisión, algunos cuadrantes o partes de la imagen se procesarán más rápido que otros. El código para esta aplicación se puede encontrar en [78].

Otra aplicación que se puede programar con DLML es la segmentación (cortes) de imágenes de cerebro. La segmentación sirve para seguir y cuantificar la evolución de muchas lesiones como la esclerosis múltiple, el mal de Parkinson o el Alzheimer. Para llevar a cabo la segmentación se obtienen varios cortes del cerebro mediante tomografía para después realizar un procesamiento sobre los cortes . Normalmente el procesamiento se realiza de manera secuencial procesando uno a uno los cortes. Un ejemplo de dos cortes se muestra en la Figura 5.12. Nuevamente el código o algoritmo de programación para esta aplicación se encuentra en [80].

Hasta ahora se han mostrado algunas de las funciones y macros que conforman la interfaz del modelo, estas fueron `DLML_Init`, `DLML_Only_one`, `DLML_Insert`, `DLML_Get`, `DLML_Reduce_Add` y `DLML_Finalize`. En la siguiente sección mencionamos todas las funciones que conforman a DLML.

5.6. Interfaz de DLML

Las funciones que integran la interfaz actual de DLML se dividen en tres tipos: señalización, manipulación y recopilación. Estos tres tipos y las funciones que los componen, se resumen en la Tabla 5.1.

- Señalización. El objetivo de estas funciones es señalizar o marcar los límites del ámbito de DLML. Las funciones de señalización son:

  DLML_Init - Inicializa la lista

  Sintaxis - *void DLML_Init(Lista *)*

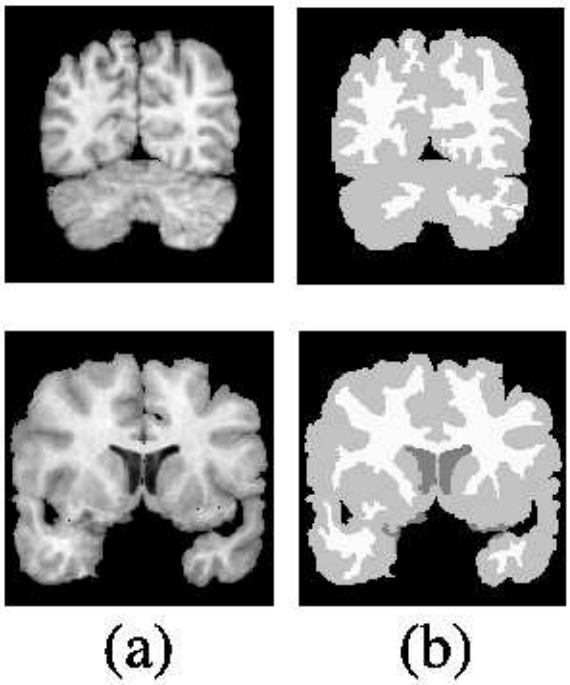

Figura 5.12: Segmentación de dos imágenes de cerebro. (a) Imágenes fuente. (b) Imágenes segmentadas.

Tipo	Señalización	Manipulación	Recopilación
Funciones	DLML_Init() DLML_Finalize() DLML_Only_one	DLML_Length() DLML_Get() DLML_Query() DLML_Insert()	DLML_Exchange() DLML_Reduce_Add() DLML_Reduce_Average() DLML_Reduce_Max()

Tabla 5.1: Tipos de funciones en DLML.

Descripción - Esta función se encarga de inicializar una lista y hacerla vacía (longitud=0). Después de esto la lista puede ser manipulada.

DLML_Finalize - Finaliza DLML

Sintaxis - *void DLML_Finalize(void)*

Descripción - Asegura el no procesamiento de la lista y devuelve el control al sistema operativo.

DLML_Only_One - Sólo un proceso

Sintaxis - *DLML_Only_One* { }

Descripción - Esta macro señala que un solo procesador ejecutará las instrucciones contenidas en ésta. La macro se delimita por llaves { }.

- Manipulación. En el tipo de *manipulación* están las funciones que se encargan del manejo de las listas.

DLML_Length - Longitud de la lista

Sintaxis - *int DLML_Length(void)*

Descripción - Devuelve un entero, el cual indica la longitud actual de lista local.

DLML_Get - Obtiene elemento

Sintaxis - *char DLML_Get(Lista *, node *)*

Descripción - El elemento es obtenido de la lista apuntada por `Lista` y es dejado en el apuntador del tipo `node`. Éste elemento es obtenido de la lista local, decrementando su tamaño en uno. DLML se encarga de insertar elementos en la lista local cuando queda vacía, suprimiéndolos de otras listas locales de forma transparente al programador. En caso de que no existan más elementos que procesar en *todo el sistema*, `DLML_Get` devuelve un valor $FALSE$.

DLML_Query - Consulta elemento

Sintaxis - *char DLML_Query(Lista *, node *)*

Descripción - Es similar a *DLML_Get*. La única diferencia es que el elemento no es eliminado de la lista. Esta función puede servir cuando se requiere calcular el promedio de los datos de la lista local.

DLML_Insert - Inserta elemento

Sintaxis - *void DLML_Insert(Lista *, node)*

Descripción - Inserta localmente un elemento del tipo `node` en la lista apuntada por `Lista`. Esta inserción se da normalmente al inicio del programa, pero puede haber aplicaciones donde la inserción se haga a tiempo de ejecución, como vimos en la subsección 5.5.2

- Recopilación. El tipo de *recopilación* concentra a las funciones que recuperan resultados parciales o previos. Recordar que DLML trabaja sobre listas distribuidas en todos los procesadores, por lo que en muchas ocasiones los resultados parciales se encuentran esparcidos. Las funciones que ayudan en esta recopilación son:

  DLML_Exchange - Intercambia resultados

Sintaxis - *int DLML_Exchange(void *fbuffer, int cantidad, void *dbuffer)*

Parámetros de entrada

fbuffer - dirección de inicio de la variable fuente

cantidad - número de datos a ser transferidos

Parámetro de salida

dbuffer - dirección de inicio de la variable destino

Descripción - Esta función realiza un intercambio de datos, entre TODOS los procesadores. Para ello la función se auxilia de un arreglo donde se guardan TODOS los datos enviados. Para tener un orden, cada posición del arreglo contiene el dato enviado por dicho procesador. Por ejemplo, el arreglo 5,2,3 indica que P_0 envió el dato 5, P_1 el dato 2 y P_2 el dato 3. Al final de la función `DLML_Exchange` los tres procesadores (P_0-P_2) tienen el mismo arreglo. La coherencia del arreglo se garantiza ya que no se prosigue con la siguiente instrucción hasta que la función `DLML_Exchange` se ha completado en todos los procesadores.

DLML_Reduce_Add - Reducción con suma

Sintaxis - *int DLML_Reduce_Add(void *fbuffer, void *dbuffer)*

Parámetro de entrada

fbuffer - dirección de inicio de la variable fuente

Parámetro de salida

dbuffer - dirección de inicio de la variable destino

Descripción - Esta función realiza una reducción o envío de todos los datos parciales a un solo procesador (procesador cero). Adicionalmente al envío, los resultados parciales son sumados y el resultado es dejado en la variable `dbuffer` del procesador cero.

DLML_Reduce_Average - Reducción con promedio

Sintaxis - *int DLML_Average(void *fbuffer, void *dbuffer)*

Parámetro de entrada

fbuffer - dirección de inicio de la variable fuente

Parámetro de salida

dbuffer - dirección de inicio de la variable destino

Descripción - Esta función realiza una reducción o envío de todos los datos parciales a un solo procesador (procesador cero). Adicionalmente al envío, los resultados parciales son promediados.

DLML_Reduce_Max - Reducción con máximo

Sintaxis - *int DLML_Max(void *fbuffer, void *dbuffer)*

Parámetro de entrada

fbuffer - dirección de inicio de la variable fuente

Parámetro de salida

dbuffer - dirección de inicio de la variable destino

Descripción - Esta función realiza una reducción o envío de todos los datos parciales a un solo procesador (procesador cero). Adicionalmente, la función devuelve el valor máximo de todos los envíos.

5.7. Comparación de DLML con otros ambientes de programación

En la evaluación cualitativa, la programación con DLML comparado con PVM/MPI [43, 57] evita la asignación explícita de datos, la sincronización explícita y los identificadores de procesos, lo cual facilita la programación. Otra de las ventajas de DLML con respecto a MPI es que prácticamente no existe la posibilidad de tener inter-bloqueos, ya

que la programación se reduce a inserciones y eliminaciones de una lista con funciones predefinidas.

En comparación con TreadMarks [6], DLML evita el uso de identificadores de procesos en la programación. Otra de las ventajas es que con DLML no es necesaria una sincronización explícita mediante barreras globales lo cual afecta el rendimiento. Otra de las ventajas es que se evita el uso de candados explícitos. DLML también maneja candados pero son internos o transparentes al programador. Con respecto a Linda [85], la arquitectura distribuida de DLML evita un acceso centralizado de los datos (tuplas) como se tiene en Linda. Algo similar de DLML con Linda son sus operaciones de acceso a datos (en DLML Get e Insert, y en Linda Read y Out), no obstante que ambas son atómicas las operaciones en DLML son más rápidas porque son distribuidas y en Linda son centralizadas. Esto último provoca en Linda un fuerte cuello de botella. Otra ventaja es que el modelo DLML evita los interbloqueos que se pueden tener con Linda.

Con respecto a Haskell [92], una de las ventajas de DLML es que no necesita el uso de combinadores como `par` y `seq` para hacer eficiente la programación. Con DLML la programación es cuasi-secuencial, sin la necesidad de usar explícitamente combinadores paralelos y secuenciales. Esta programación cuasi-secuencial con DLML también tiene la ventaja de ser más sencilla en comparación con la fuerte notación matemática que requiere Haskell, esto porque la programación con Haskell es declarativa, mientras que con DLML es imperativa.

En comparación con los skeletons como eSkel [11], DLML tiene la ventaja que sus funciones son muy sencillas y con pocos parámetros a diferencia de eSkel donde sus skeletons o funciones manejan una cantidad importante de parámetros. Otra de las ventajas de DLML con respecto a eSkel y en general con toda la programación realizada con skeletons es que la eficiencia no depende la buena selección de los skeletons disponibles. Con respecto a SAMBA [72], DLML puede resultar más sencillo de programar ya que la programación es secuencial-estructurada a diferencia de la programación de objetos de SAMBA. Esta programación con objetos hace necesario el llenado de los métodos dentro de las varias clases que maneja SAMBA, complicando un tanto la programa-

ción. Quizá una de la ventajas de SAMBA frente a DLML es que implementa varias políticas de distribución, aunque por otro lado esto también puede ser una desventaja ya que el programador debe escoger cual política de distribución es la que le conviene, dependiendo de los aspectos de su aplicación, si es cluster homogéneo-heterogéneo, dedicado o no dedicado. Con la versión final de DLML sólo se tiene la política de subasta (transparente al programador), la cual como se verá en la siguiente capítulo resulta ser muy eficiente.

Finalmente, comparado con SADTs [66], DLML es muy similar en la programación, quizá sea un poco más fácil la programación con SADTs ya que ésta no requiere un recolección final de resultados parciales. No obstante, DLML tiene la ventaja que su modelo requiere que los datos de las aplicaciones se representen en lista y no en colas, lo cual es menos restrictivo. Otra de las ventaja de DLML frente a SADTs, es que las listas de DLML son distribuidas y con procesamiento independiente, mientras que en SADTs las colas son replicadas y compartidas lo cual requiere de algún mecanismo de coherencia, lo cual afecta el rendimiento.

En cuanto a las limitaciones, la programación con DLML es cuasi-secuencial, lo cual significa que el programador debe estar consciente que aunque escriba un solo código, este realmente se ejecuta en todos los procesadores. Esto conlleva a que durante la ejecución, existan resultados parciales distribuidos en todos los procesadores. Esta distribución hace necesaria una etapa o proceso de recopilación de resultados (no obstante que DLML ofrece funciones para ello). Otra limitación de la programación con DLML, es que los datos de las aplicaciones a desarrollar deben ser independientes, esto quiere decir, que los datos puedan ser procesados en cualquier procesador y en cualquier orden.

5.8. Resumen

En el capítulo se enfatizó la necesidad de tener ambientes o herramientas que faciliten la programación paralela en clusters, sin dejar de tomar en cuenta la eficiencia. Para ello proponemos una programación basada en listas la cual es sinónimo de programación

paralela. Esto lo afirmamos porque las listas no requieren un orden de sus elementos, lo que les permite dividirse y distribuirse fácilmente. Precisamente lo anterior es lo que define los pasos de la programación paralela. Para ello propusimos una metodología que define la programación basada en listas. Esta metodología se aplicó en diferentes aplicaciones regulares e irregulares. Posteriormente se presenta nuestro ambiente de programación DLML el cual prácticamente es la implementación y el front-end de la metodología descrita con anterioridad. Se presenta un esqueleto de lo que es un programa bajo DLML mostrando la similitud con la programación basada en listas. El ambiente se redondea presentando ejemplos de aplicaciones desarrolladas bajo DLML. También se presentó la interfaz de DLML la cual cuenta con funciones del tipo: Señalización, Manipulación y Recopilación. Finalmente, se muestra una evaluación cualitativa de DLML comparado con otros ambientes de programación.

Capítulo 6

Diseño de DLML

En este capítulo se describe el diseño de DLML. En el apéndice A se describe su implementación, instalación y configuración.

6.1. Arquitectura de DLML

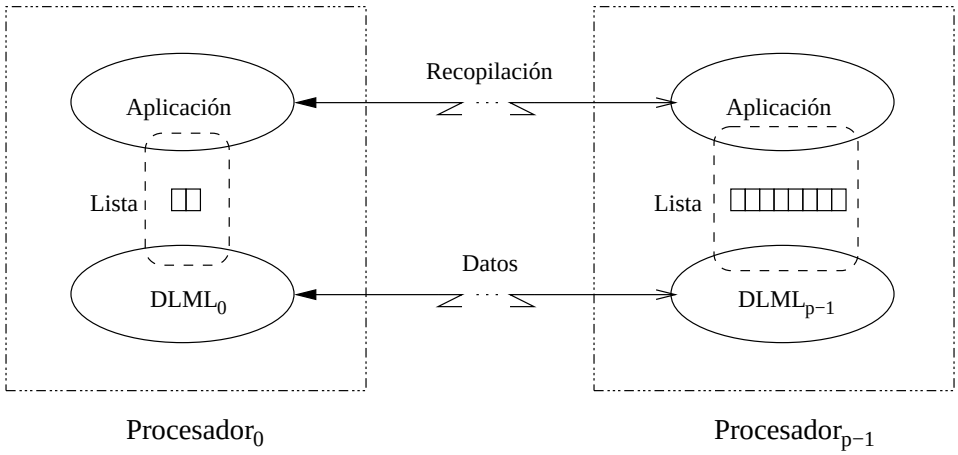

Figura 6.1: Arquitectura DLML

La Figura 6.1 presenta la arquitectura de DLML, la cual consta de dos procesos por procesador: *Aplicación* y *DLML*. A esta configuración se llega de la siguiente manera.

Una vez escrito el programa el cual ya se compiló y se ligó con la biblioteca dlml.h,

se ejecuta la instrucción `DLML_run todo`. Esta instrucción del ambiente DLML crea una copia del proceso *Aplicación* y del proceso *DLML* en cada procesador, tal y como se muestra en la Figura 6.1.

Cada proceso *Aplicación* ejecuta el código escrito por el programador (modelo SPMD). Una de las primeras instrucciones dentro de ese código es la instrucción `DLML_Init` que inicializa la ejecución del ambiente DLML y la lista. La inicialización del ambiente DLML permite que se puedan usar las funciones disponibles como la inserción de elementos a la lista. Cada vez que el programador utiliza la función `DLML_Insert` el proceso *Aplicación* agrega elementos en su lista local. La lista en cada nodo es compartida por el proceso *Aplicación* y *DLML* en ese nodo. También puede suceder que en el programa desarrollado se tenga la primitiva `DLML_Get`, en este caso el proceso *Aplicación* se encarga de obtener/eliminar los elementos de la lista local. Cuando el proceso *Aplicación* no encuentra más datos (lista vacía), el proceso *DLML* empieza a correr.

Una de la funciones del proceso *DLML* en cada nodo es conseguir datos de otros nodos cuando la lista local se encuentra vacía. Para ello el proceso *DLML* establece una comunicación con los demás procesos *DLML* con el objetivo de obtener datos. En la Figura 6.1, en algún momento en el $Procesador_0$ (izquierda), el proceso *Aplicación* consumirá los dos elementos de su lista. En ese momento el proceso *Aplicación* se bloquea y el proceso $DLML_0$ se activa. Al hacer la búsqueda de datos por medio de los procesos *DLML* remotos supondremos que elige al proceso $DLML_{p-1}$ del $Procesador_{p-1}$ para obtener datos. En este caso el proceso $DLML_{p-1}$ accede a su lista local para quitar elementos. Una vez que el proceso $DLML_{p-1}$ tiene los elementos se los envía al proceso $DLML_0$ que tiene su lista vacía. Posteriormente el proceso $DLML_0$ recibe los datos y los inserta en su lista local, previamente vacía. Finalmente el proceso $DLML_0$ se bloquea después de activar al proceso *Aplicación*. Una vez activo el proceso *Aplicación* vuelve a obtener datos de su lista local para su procesamiento. La comunicación que se establece entre los procesos *DLML* es mediante paso de mensajes. El paso de mensajes sirve para la búsqueda de datos así como para su envío y recepción. Como es de

esperarse en algún momento no habrá más elementos que procesar, es decir, las lista de todos los procesadores estarán vacías. Cuando este sea el caso, el proceso DLML detecta que no hay más elementos y en ese instante desbloquea al proceso *Aplicación*, el cual se encontraba bloqueado o en espera de datos con la instrucción `DLML_Get`.

Cuando el procesamiento de los elementos de la lista ha terminado, generalmente es necesario hacer una recopilación de los resultados parciales. Esta recopilación e intercambio de mensajes se realiza directamente entre los procesos *Aplicación* quienes son los que tienen estos resultados parciales. El intercambio puede ser de todos a todos (`DLML_Exchange`) o de todos a uno (`DLML_Reduce_`).

6.2. Balance de carga

Algo muy importante en la programación con clusters es el balance de carga. Recordando del Capítulo 4, existen casos donde realmente es necesario. Estos casos son: cuando la aplicación es dinámica, cuando el cluster es heterogéneo y cuando el cluster es multiprogramado. En la aplicación dinámica los datos se generan de manera impredecible durante la ejecución. Cuando el cluster es heterogéneo los nodos con menor capacidad provocan un aumento en el tiempo de respuesta. Finalmente cuando hay multiprogramación, los nodos utilizados por otros usuarios externos procesan menos datos que los nodos dedicados o exclusivos.

Para ayudar a mejorar el desempeño bajo las condiciones anteriores, DLML utiliza el algoritmo de subasta presentado en el Capítulo 4. Este algoritmo de subasta consiste en que un procesador sin carga (datos) busca el elemento con más datos para pedirle. En el ámbito de DLML la carga es el número de elementos en la lista de cada procesador. A continuación mostramos la versión del algoritmo de subasta, implementado con recolección global de información.

6.2.1. Subasta con información global

La subasta con información global consiste en que un procesador que se queda sin carga (elementos en su lista) le solicita a TODOS los procesadores información de su carga (longitud de su lista). El procesador recibe la información y la analiza, escogiendo al procesador con más carga. El siguiente caso es pedirle carga. Sin embargo puede suceder que en el transcurso de ese tiempo (desde que se envío la información hasta que se pide la carga) ya no haya elementos. A continuación presentamos ambos casos: en uno todavía existen datos que repartir (con datos disponibles), mientras que en el otro los datos ya no existen (sin datos disponibles).

Lo que presentamos a continuación es el protocolo de comunicación entre los procesos DLML, los cuales hemos etiquetado como $DLML_0$ hasta $DLML_{p-1}$. Recordemos que hay un proceso *DLML* por cada procesador.

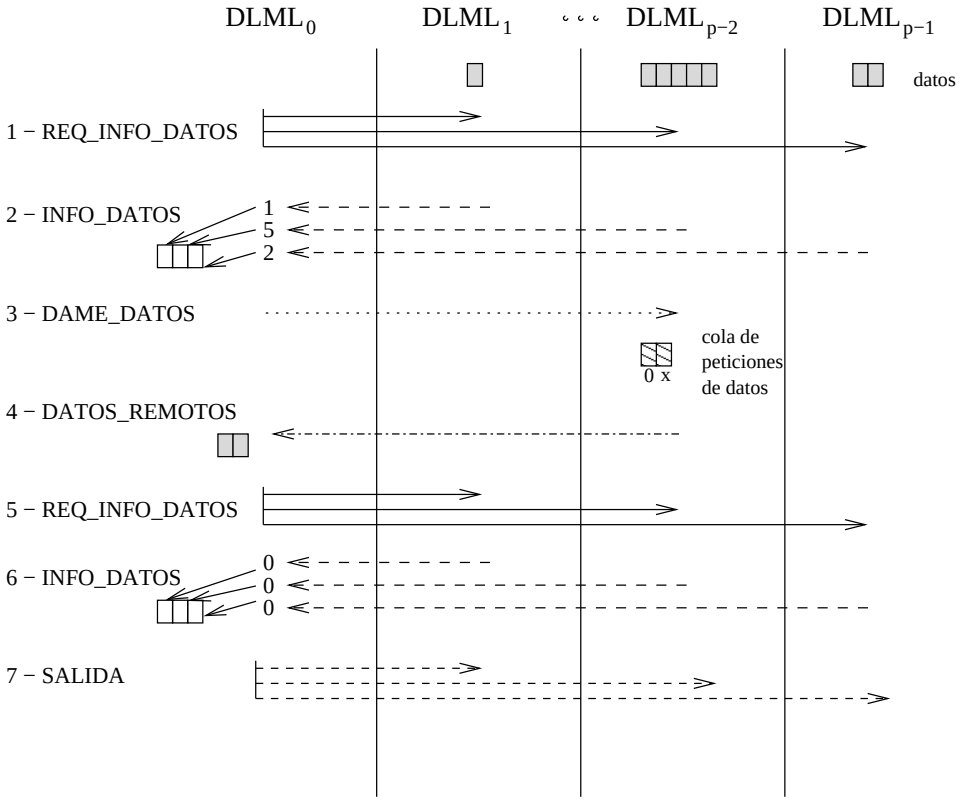

Figura 6.2: Balance de datos con información global (con disponibilidad de datos).

En el caso *con datos disponibles* (Figura 6.2), el flujo de datos comienza cuando un proceso *DLML* no tiene datos (lista vacía), en este caso $DLML_0$. Este proceso lanza un mensaje a todos los procesos DLML del cluster (paso 1 de la figura), solicitando información acerca de la longitud de sus listas. Los procesos *DLML* que reciben esta solicitud contestan con otro mensaje indicando la longitud de su lista local actual. En la Figura 6.2 se nota que los valores son 1, 5 y 2. Estos valores enviados con la etiqueta INFO_DATOS en el paso 2, son guardados en un arreglo donde cada posición corresponde al número de datos en el procesador correspondiente. Cuando han respondido todos los procesos *DLML*, se busca en el arreglo cual es el proceso *DLML* con el mayor número de elementos (datos) para pedirle datos (en este caso el proceso $DLML_{p-2}$). Dicha petición se hace en el paso 3, enviando un mensaje de tipo DAME_DATOS. Este tipo de petición y las demás peticiones que lleguen al proceso *DLML* son guardadas por éste en una cola, para posteriormente atender las peticiones de datos en el orden recibido. Por ejemplo, en la Figura 6.2 se tiene una cola de peticiones con el valor 0 y un valor x que representa la petición de datos de algún procesador x. El tener más de un valor en la cola de peticiones significa que de manera simultánea o casi al mismo tiempo dos procesos *DLML* se quedaron sin datos e iniciaron la subasta, encontrando el mismo proceso *DLML* al cual ambos le solicitan datos. Cuando el proceso *DLML* ganador de la subasta se percata que tiene peticiones de datos en su cola, distribuye el número de elementos en su lista (datos a procesar) entre el número total de peticiones más uno (el mismo). Esta repartición hace que los datos se balanceen equitativamente entre los procesos *DLML*. Los datos enviados hacia el proceso *DLML* se mandan con la etiqueta DATOS_REMOTOS en el paso 4. El algoritmo de subasta con manejo de información global termina cuando un proceso *DLML* requiere información de la longitud de las listas (5 - REQ_INFO_DATOS) y recibe todos sus mensajes de tipo INFO_NODOS con el valor de cero (paso 6), esto indica que nadie tiene elementos que procesar y el algoritmo termina, mandando en el paso 7 un mensaje de SALIDA a todos los procesos *DLML*.

El caso del algoritmo con información global *sin datos disponibles*, Figura 6.3, es similar en los primeros 3 pasos (1-REQ_INFO_DATOS, 2-INFO_DATOS, 3-DAME_DATO)

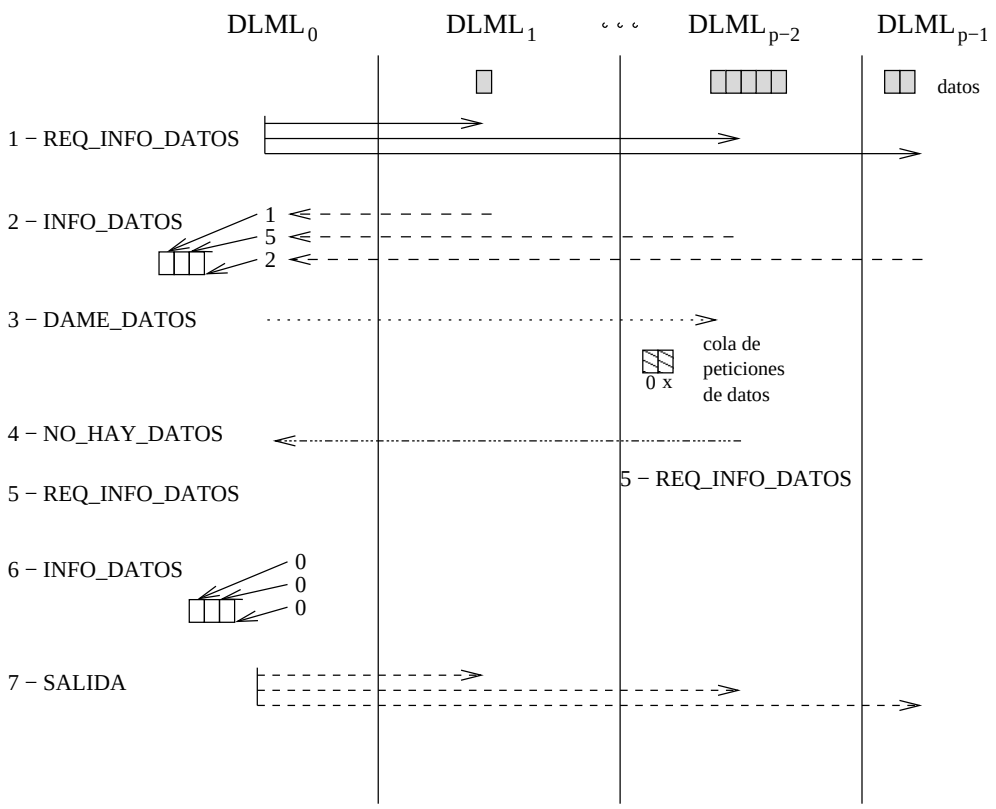

Figura 6.3: Balance de datos con información global (sin disponibilidad de datos)

al algoritmo *con datos disponibles*. No obstante, después del paso 3, es posible que no existan datos que repartir o que no alcancen datos todos los procesos DLML. En esta situación a los procesos que no obtienen datos, se le envía un mensaje con la etiqueta NO_HAY_DATOS (paso 5). Los procesos *DLML* que reciben este mensaje comienzan de nuevo el proceso de buscar datos, recomenzando a solicitar información a todos los procesos *DLML*. Por su parte, el proceso DLML que envió el mensaje NO_HAY_DATOS en el paso 5, hace lo mismo, volver a buscar datos (REQ_INFO_DATOS). Este caso de algoritmo termina de manera similar al caso *con datos disponibles*. La terminación se da cuando un proceso DLML requiere información (paso 6) y todos los procesos DLML responden con un mensaje INFO_DATOS conteniendo el valor cero (paso 7). A continuación el proceso que requirió la información envía el mensaje de SALIDA a los demás procesos DLML (paso 8).

En ambos casos del algoritmo de subasta global (con datos disponibles y no disponibles) se evita que por un periodo largo de tiempo un proceso o procesador tenga demasiados datos (glotón) o por el contrario no tengan ningún dato que procesar (hambriento). Esta situación se evita debido a que un procesador hambriento siempre encuentra un proceso glotón. Cabe mencionar que aunque el proceso glotón este muy ocupado procesando datos, una vez que le han solicitado datos, éste los reenvía pasado el procesamiento de su dato o elemento de lista actual. Cabe mencionar que el proceso hambriento puede no ser atendido en su primer solicitud de datos, en cuyo caso el proceso que no pudo cubrir dicha petición le envía un mensaje al proceso hambriento para que busque comida (datos) en otros procesos.

Esta implementación del algoritmo de subasta con información global tiene muy buena eficiencia. Sin embargo, una limitante es involucrar a todos los procesadores en la obtención de datos, ya que al aumentar el número de procesadores se incrementa también el número de comunicaciones necesarias para obtener la información global del sistema.

6.2.2. Otras alternativas

Al principio de nuestra investigación exploramos otras alternativas para el balance de datos que en su momento consideramos adecuadas. Esas alternativas incluían un algoritmo global distribuido que usaba tres procesos. En el algoritmo global distribuido cada cierto tiempo los procesadores evalúan su carga, si hay un cambio se lo notifican a todos los demás procesadores. Cuando es necesario un balance todos los procesadores verifican localmente su estado de carga, si es sobrecargado buscan en su información el procesador menos cargado para enviarle datos. Sin embargo, el algoritmo resultó ser complicado en el ajuste de sus parámetros por lo que decidimos cambiarlo al algoritmo de subasta.

Este algoritmo global-distribuido fue implementado en base a los procesos contabilizador, administrador y distribuidor, como se muestra en la Figura 6.4. La función de

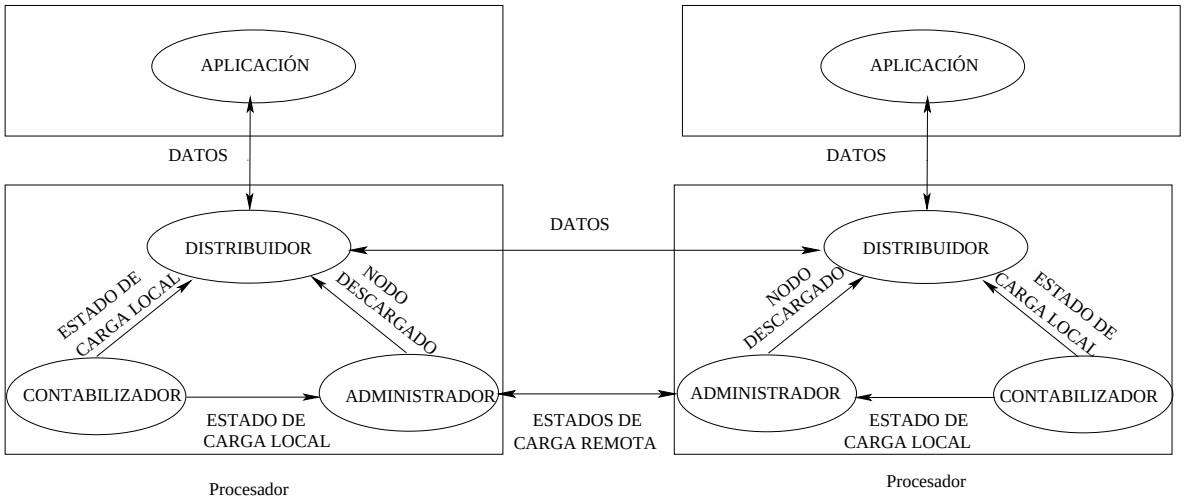

Figura 6.4: Algoritmo de distribución global con módulo DLML de tres procesos.

cada proceso es la siguiente:

- El proceso Contabilizador se encarga de medir el estado de carga actual del procesador. En este caso el estado de carga esta en función del porcentaje de utilización de CPU el cual se mide cada cierto tiempo. Este tiempo es ajustado por el programador. Cada vez que el proceso Contabilizador detecta un cambio en el estado de carga (descargado $\Leftrightarrow$ sobrecargado) le envía un mensaje al proceso Distribuidor y al proceso Administrador indicándoles el nuevo estado de carga. La frontera de carga que divide las dos regiones (descargado y sobrecargado) es otro parámetro ajustable por el programador.

- El proceso Administrador se encarga de mantener una vista global de la carga en el sistema. Esta vista global la mantiene en un vector de carga donde cada posición contiene el estado de carga del procesador correspondiente. Este vector de carga es actualizado por el proceso Contabilizador cuando hay un cambio en el estado de carga, por ejemplo de sobrecargado$\rightleftarrows$descargado. Posteriormente difunde este nuevo estado de carga a todos los procesadores, enviando un mensaje con el nuevo estado de carga a los procesos Administrador remotos. Precisamente otra función del proceso Administrador es recibir los estados de carga remota, actualizar el

vector de carga con esa información y analizar si hubo un cambio en el nodo o procesador descargado. Si es que lo hubo, le envía esa información al proceso Distribuidor.

- El proceso Distribuidor se encarga de transferir datos. Esta transferencia la hace en base a la carga local y al nodo descargado. Si la carga local es sobrecargada y el nodo descargado es diferente de él mismo, se hace la transferencia de datos hacia el nodo descargado. Si el nodo descargado es el mismo nodo local, o la carga local es descargada entonces no se hace ninguna transferencia. El número de elementos o datos a transferir es un parámetro ajustable por el programador. El proceso distribuidor también se encarga de recibir carga remota y de ser el puente para las transferencias de datos con el proceso aplicación.

6.3. Aspectos de sincronización

6.3.1. Sincronización Aplicación - DLML

Entre los procesos *aplicación* y *DLML* suceden dos tipos de sincronización. Una sincronización para acceder a la lista y otra sincronización para lanzar la subasta.

La primera sincronización es necesaria ya que ambos procesos pueden acceder a la lista simultáneamente, ya que la lista esta compartida entre ambos procesos. Por un lado el proceso aplicación inserta y elimina elementos de las lista (local) en cualquier momento. Por otro lado el proceso DLML también inserta y elimina elementos de la lista, no obstante la inserción es un caso especial visto adelante (segunda sincronización). La eliminación de elementos de la lista por parte del proceso DLML se da cuando otro proceso DLML externo requiere datos. Esta eliminación por parte del proceso DLML se puede dar en cualquier momento y puede coincidir con un acceso por parte del proceso aplicación, por lo que la sincronización usando un candado es necesaria.

La segunda sincronización es al lanzar la subasta. Es este caso el proceso Aplicación y el proceso DLML actúan de manera conjunta. El proceso Aplicación al quedarse sin

datos que procesar (lista vacía) notifica al proceso DLML y después se queda dormido. El proceso DLML inicia entonces la subasta de acuerdo a los algoritmos de distribución descritos con anterioridad. Si el proceso DLML obtiene datos, los inserta en la lista y despierta al proceso Aplicación. Cabe mencionar que la inserción anterior no requiere de candados ya que el proceso Aplicación se encuentra dormido. También puede suceder que no halla más datos y por lo tanto nada que insertar en la lista, en ese caso de igual forma se despierta al proceso Aplicación quien devuelve en su función DLML_Get el valor de FALSE y termina el programa.

6.3.2. Terminación

El sistema DLML finaliza cuando ya no existen elementos que procesar. Esto sucede cuando un proceso *DLML* sin datos solicita información, ya sea mediante la etiqueta REQ_INFO_DATA y recibe como respuesta (INFO_DATOS) el valor cero de todos los procesos *DLML*. Cuando se cumple esta situación, este proceso se encarga de hacer la terminación, enviando el código de SALIDA a todos los procesos *DLML*, quien a su vez se encargan de terminar sus propios procesos de *Aplicación*.

Si por alguna razón no se diera la situación anterior, por ejemplo que hubiera un DLML_Insert, pero no hubiera su DLML_Get respectivo, en ese caso la función de terminación DLML_Finalize elimina los elementos que quedaron en la lista, finaliza los procesos Aplicación y DLML y le devuelve el control al sistema operativo.

Un caso interesante en la terminación se da cuando los procesos *DLML* se han quedados sin datos y súbitamente algún proceso aplicación genera más datos. Aparentemente en este caso sucedería que los procesos DLML con lista vacía terminarían y dejarían solo al procesador que súbitamente generó más datos, impidiendo de esta forma el proceso de balance de datos en el sistema. Esta situación se evita con las siguientes dos determinaciones: 1) un proceso DLML sólo envía el estado de carga cero (lista vacía) cuando se encuentra buscando elementos y no cuando su lista local se encuentra vacía; 2) un proceso DLML al que se le solicitan datos (después de haber sido elegido como el

proceso con el mayor número de elementos) nunca se queda sin elementos que procesar. De esta forma el estado de carga (número de elementos) del proceso que cede datos siempre es diferente de cero (lista vacía). Recalcando, la terminación únicamente se da cuando todos los procesos DLML están en busca de datos. Cuando se ha llegado a esta condición un primer proceso DLML detecta que el resto de los procesos DLML le han confirmado que sus listas están vacías e inicia entonces el proceso de terminación.

6.4. Resumen

En este capítulo se presentó el diseño de DLML, el cual se conforma de dos procesos por procesador. Un proceso es la aplicación y otro proceso es DLML. El proceso aplicación es una instancia del programa ejecutado bajo el paradigma SPMD. Por otro lado el proceso DLML es el encargado del balance de los datos. Ambos procesos acceden a un lista compartida ya sea para insertar o eliminar elementos en el caso de la aplicación, o para redistribuir elementos en el caso del proceso DLML. Esto último se hace entre procesos DLML mediante el envío y recepción de mensajes. Otra comunicación se da entre procesos aplicación los cuales pueden hacer intercambios de resultados parciales o finales.

En el capítulo también se muestra la necesidad de una distribución de carga para casos de aplicaciones dinámicas, clusters heterogéneos y no-dedicados. Para ello se muestran el algoritmo de balance implementado bajo DLML, el cual consiste en una subasta con información global. También se muestran aspectos de sincronización entre la aplicación y DLML, así como la terminación de los procesos.

Capítulo 7

Evaluación

En este capítulo mostramos una evaluación del desempeño de nuestro ambiente de programación DLML.

7.1. Metas experimentales

De manera general el desempeño de un sistema paralelo va en función de sus tiempos de respuesta. Si en una aplicación los tiempos de ejecución se reducen, se puede decir que el sistema paralelo tiene un buen desempeño. Adicionalmente, sería bueno considerar aspectos que pudieran afectar ese desempeño. En razón de esto, suena interesante ver el comportamiento de DLML en cuanto a desempeño bajo situaciones de desbalance. Por ejemplo, para aplicaciones dinámicas donde los datos se van generando a tiempo de ejecución, una meta experimental es comparar DLML con una versión paralelas sin balance. Esto permitiría evaluar el desempeño del algoritmo de balance de DLML. De manera similar se pueden considerar aspectos como la heterogeneidad de un cluster, donde los nodos con mayor capacidad procesan más elementos (datos) que los nodos de menor capacidad. Nuevamente un experimento interesante sería ver el comportamiento de DLML con una versión paralela sin balance. Otro factor a evaluar sería un cluster con multiprogramación donde por definición existen otros usuarios en ciertos nodos. Este factor podría ser controlado de una mejor forma si de manera artificial se genera

sobrecarga en ciertos nodos. La sobrecarga podrían ser aplicaciones que demandan fuertemente el uso del CPU.

Por otro lado algo importante a evaluar es propiamente el balance de los datos (cómo se distribuyen). En el balance de carga que definimos, la distribución esta en función de la capacidad de los nodos, es decir, los nodos con más capacidad deben de evaluar o procesar más elementos de la lista, no obstante que exista cualquiera de las condiciones de desbalance que mencionamos en el párrafo anterior. Bajo esta premisa un experimento interesante sería observar la distribución de los datos procesados por los procesadores.

Otra de los experimentos interesante es que la biblioteca se presenta para cómputo paralelo en clusters, por lo que probar con diferentes arquitecturas de cluster variando capacidad, número de procesadores y arquitecturas de nodo monoprocesador o multiprocesador suena interesante.

Resumiendo lo anterior, se propone la siguiente plataforma de experimentación:

7.2. Plataforma de experimentación

7.2.1. Clusters utilizados

Los clusters que utilizamos en la evaluación de DLML son los siguientes:

- Cluster A
  - Arquitectura de cada nodo - Monoprocesador
  - # de nodos - 8
  - Tipo de procesador - Intel Pentium III a 450MHz
  - Memoria - 64 MB
  - Tipo de conmutador - Fast-Ethernet (100 Mbits/seg)
  - Sistema operativo - Red-Hat 7.3

- Ubicación - Universidad Autónoma Metropolitana, Iztapalapa, MÉXICO

- Cluster B
  - Arquitectura de cada nodo - Monoprocesador
  - # de nodos - 17
  - Tipo de procesador - Intel Pentium IV a 2.8GHz
  - Memoria - 512 MB
  - Tipo de conmutador - Fast-Ethernet (100 Mbits/seg)
  - Sistema Operativo - Fedora Core 4
  - Ubicación - Universidad Autónoma Metropolitana, Iztapalapa, MÉXICO

- Cluster C
  - Arquitectura de cada nodo - Tetraprocesador
  - # de nodos - 30 (120 procesadores).
  - Tipo de procesador - AMD Opteron a 2.4GHz
  - Memoria - 8 GB por nodo
  - Tipo de conmutador - Infiniband (10 Gbits/seg)
  - Sistema operativo - Suse 9.1 (x86-64)
  - Ubicación - Technische Universität München, DEUTSCHLAND

7.2.2. Configuración de DLML

Un aspecto importante en el desempeño de DLML es su política de balance de datos. Las políticas que probamos son:

- Global distribuida. Cada cierto tiempo los procesadores cuantifican su carga, si existen un cambio, se lo notifican a todos los procesadores. La notificación se guarda en un vector que contiene los estados de todo el sistema, en donde cada

posición del vector indica el estado de carga en dicho procesador. Cuando es necesario un balance de carga (elementos de la lista), los procesadores con mayor carga ceden parte de los elementos de su lista a los procesadores con menor carga.

- Subasta global. cada vez que un un procesador se queda con su lista vacía, busca en todas las listas del cluster y elige la que tiene mayor longitud para obtener elementos.

7.2.3. Aplicaciones

Las aplicaciones que utilizamos en nuestra evaluación de DLML fueron:

Clasificación basada en algoritmos evolutivos

El problema consiste en utilizar algoritmos evolutivos para encontrar una red neuronal que haga la tarea de clasificador [27]. Cada red neuronal se representa como un individuo que evoluciona (mejor individuo = mejor red neuronal = mejor clasificador). Con DLML cada elemento de sus listas es un individuo. Inicialmente, la población total (número total de individuos) es fija y se divide equitativamente entre todos los procesadores formando subpoblaciones del mismo tamaño. Cada procesador evalúa su subpoblación (elementos de sus lista) y encuentra su mejor individuo el cual se difunde para encontrar el mejor global. El problema es la sincronización entre todos los procesadores para encontrar el mejor individuo, esto se agrava debido a que el procesamiento de cada individuo tiene costo diferente. Se corrió la aplicación con una población de 48 individuos 10 veces, en un cluster de 6 procesadores dedicados y no-dedicados, con balance y sin balance de individuos. Con la política de configuración global distribuida

Ordenamiento paralelo mergesort

La aplicación ordena números de manera paralela mediante el paradigma dividir para vencer (DPV) [41]. Paso a paso se divide y asigna el conjunto total de números entre todos los procesadores. Posteriormente cada procesador ordena su parte y comienza

la reconstrucción de la solución final (enviando y mezclando números). Sin embargo, aspectos como la heterogeneidad de los clusters y la multiprogramación afectan el desempeño. Con DLML el paradigma DPV se lleva a una lista con un elemento (todos los números) a una lista con muchos elementos (pocos números) que se distribuyen paralelamente. En este caso el número de datos que se procesa es fijo, al igual que el costo por procesar cada elemento. Sin embargo, este costo varía dependiendo de los aspectos descritos anteriormente (heterogeneidad y multiprogramación). Se corrió la aplicación con 2^{19} números 10 veces, en un cluster de 8 procesadores con y sin multiprogramación para una versión con DLML y otra con distribución explícita (DE).

Segmentación de imágenes

Esta aplicación reconstruye imágenes de cerebro en 3D usando Mean-Shift (MSH) [80]. La reconstrucción se logra aplicando MSH sobre varias imágenes 2D (cortes) a nivel de pixel (costoso por el gran número de cortes y la alta resolución requerida). Para reducir tiempos los cortes se dividen entre los procesadores. No obstante que el número de cortes es fijo (número de imágenes), el costo de procesamiento de cada corte es diferente ya que el costo de MSH aumenta o disminuye de acuerdo a los cambios en la intensidad de los pixels. Bajo el ambiente DLML cada elemento de la lista es un corte. Se corrió la aplicación con 165 cortes con resolución de 217 $\times$ 181 pixels en un cluster homogéneo de 4 a 120 procesadores sin multiprogramación. Se probaron 3 versiones: estática (distribución de cortes equitativa y definitiva), maestro-esclavo (un procesador maestro tiene todos los cortes los cuales le son solicitados uno a uno por los procesadores esclavos) y DLML.

No Ataque de Reinas (NAQ)

NAQ consiste en encontrar todas las posibilidades de colocación de N reinas en un tablero de N $\times$ N sin que se ataquen entre sí [18]. La soluciones (colocación) se encuentran explorando todo el árbol de búsqueda de las posibles soluciones descartando

aquellas que ya no puedan ser solución. Bajo DLML cada elemento de la lista contiene una posible solución a explorar conformada por un el número de reina a ser colocada y un vector de tamaño N con la posición de la reinas colocadas hasta ese momento. Los aspectos que afectan el desempeño es que el tamaño de las listas crece y decrece conforme se van generando nuevas soluciones o eliminando respectivamente (número de datos variable). El costo por procesar cada elemento (generar nuevas soluciones) es variable y esta en función de la profundidad del elemento en el árbol de búsqueda (costo inversamente proporcional a profundidad). Se corrió la aplicación con $12 \leq N \leq 17$ reinas en un en versión secuencial (Pentium IV a 3.4GHz) y versión DLML (cluster homogéneo de 17 procesadores). Adicionalmente se corrió la aplicación con $8 \leq N \leq 17$ en una versión secuencial (Pentium IV a 3.4 GHz), en una versión paralela-plana (distribución de elementos sólo al inicio) y versión DLML (ambas en un cluster de 17 procesadores homogéneo y heterogéneo). Finalmente se corrió la aplicación con N=17 para una versión paralela-plana y versión DLML (ambas en un cluster de 17 procesadores homogéneo y heterogéneo).

Registro de Imágenes No Rígido (NRIR)

NRIR es una aplicación que genera un mapa entre un par de imágenes usando funciones no lineales [25, 78]. A partir de las imágenes fuente y destino se hace un procesamiento (se alínea) sobre los pixels para la imagen fuente sea alineada (similar) con la imagen destino. La aplicación utiliza un paradigma DPV que compara inicialmente las imágenes completas. Si la alineación es satisfactoria el procesamiento se termina, sino ambas imágenes se subdividen en cuatro y el procesamiento se repite para cada cuadrante. Bajo DLML cada cuadrante que se va generando es un elemento de la lista, cabe mencionar que cada cuadrante se puede subdividir más que otros, por lo que el costo de procesamiento de los elementos es diferente y el número de datos (elementos) es variable dependiendo de las imágenes. Se corrió la aplicación con imágenes fuente y destino de 256×256 pixels con 256 niveles de gris 15 veces en un cluster dedicado de 4 procesadores.

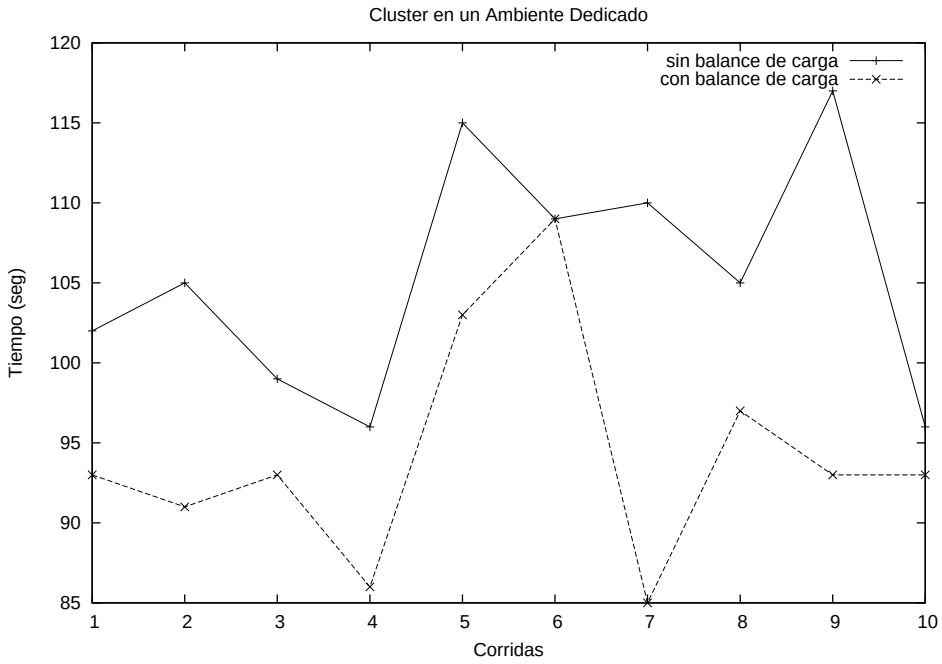

Figura 7.1: Clasificación en base a algoritmos evolutivos en ambiente dedicado, con y sin balance de individuos.

7.3. Resultados

7.3.1. Clasificación basada en algoritmos evolutivos

Las Figuras 7.1 y 7.2 muestran los tiempos de respuesta de la clasificación basada en algoritmos evolutivos bajo un ambiente dedicado y no-dedicado respectivamente. El cluster utilizado fue A con nodos homogéneos; la configuración de balance fue global distribuido. Las configuraciones en la figura corresponden a DLML sin balance y con balance de individuos. La fuente de desbalance de carga la genera la evaluación de cada individuos por el no-determinismo del evolutivo y el ambiente no-dedicado.

De la Figura 7.1 podemos ver los tiempos de respuesta calculados en segundos para 10 diferentes corridas de la clasificación basada en algoritmos evolutivos. De la figura se observa que la versión con balance de carga mejora los tiempos de ejecución en 9 de los 10 casos y el restante es un empate (el empate ocurrido en punto 6 fue una casualidad debido al no determinismo del algoritmo evolutivo). Esta mejora o reducción en los

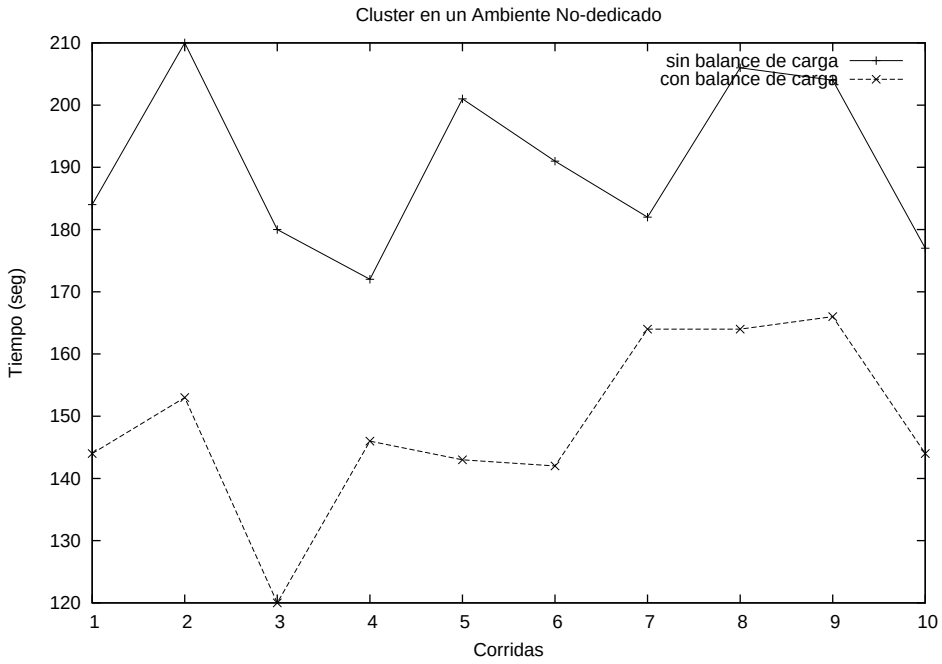

Figura 7.2: Clasificación en base a algoritmos evolutivos en ambiente no-dedicado, con y sin balance de individuos.

tiempos, se debe a que el balance de individuos ayuda cuando el costo de procesamiento de cada individuo es diferente.

De la figura 7.2 igualmente vemos los tiempos de respuesta calculados en segundos para 10 corridas. De la figura se observa que la versión con balance de carga mejora en todos los casos a la versión sin balance de carga. En este caso la diferencia está marcada debido a que se trata de un ambiente no-dedicado aunado al costo de procesamiento diferente de cada individuo. Sin embargo esta versión de balance de carga (misma que la gráfica anterior) la abandonamos debido a que su eficiencia estaba en función de un ajuste manual de parámetros del algoritmo por parte del programador [27].

7.3.2. Mergesort

La Figura 7.3 muestra el tiempo de respuesta de MergeSort bajo un ambiente dedicado y no dedicado. El cluster utilizado fue A; la configuración de balance fue global distribuido. Las configuraciones en la figura corresponden a la versión con DLML y la

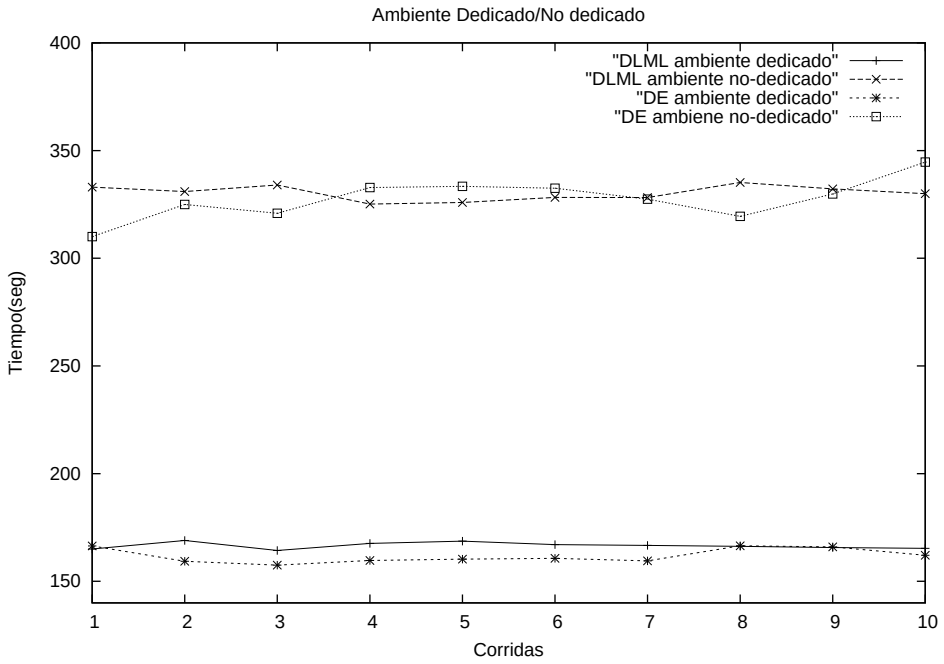

Figura 7.3: Mergesort para versión DLML y DE en ambientes dedicados y no dedicados.

versión con distribución explícita (DE). La fuente de desbalance la genera principalmente la multiprogramación (ambiente no-dedicado).

De la figura podemos ver los tiempos de respuesta calculados en segundos para diez diferentes corridas de Mergesort, en ambiente dedicado y no dedicado. Los tiempos de respuesta son comparables, es decir, los tiempos de respuesta para DLML y para la distribución explícita (DE) en un ambiente dedicado son similares en promedio ($\approx$ 160 seg), al igual que los tiempos para DLML y DE en un ambiente no-dedicado ($\approx$ 330 seg). Observando con más detalle se nota que bajo ambientes dedicados, la versión DE mejora a DLML, en varias corridas, esto se debe a que DE esta hecha ad-hoc a la aplicación y no hay factores de desbalance. Esta situación cambia en ambientes no-dedicados donde DLML es mejor en 4 de las 10 corridas. Esta mejora de DLML se debe a que ahora existen condiciones de desbalance (ambiente no-dedicado).

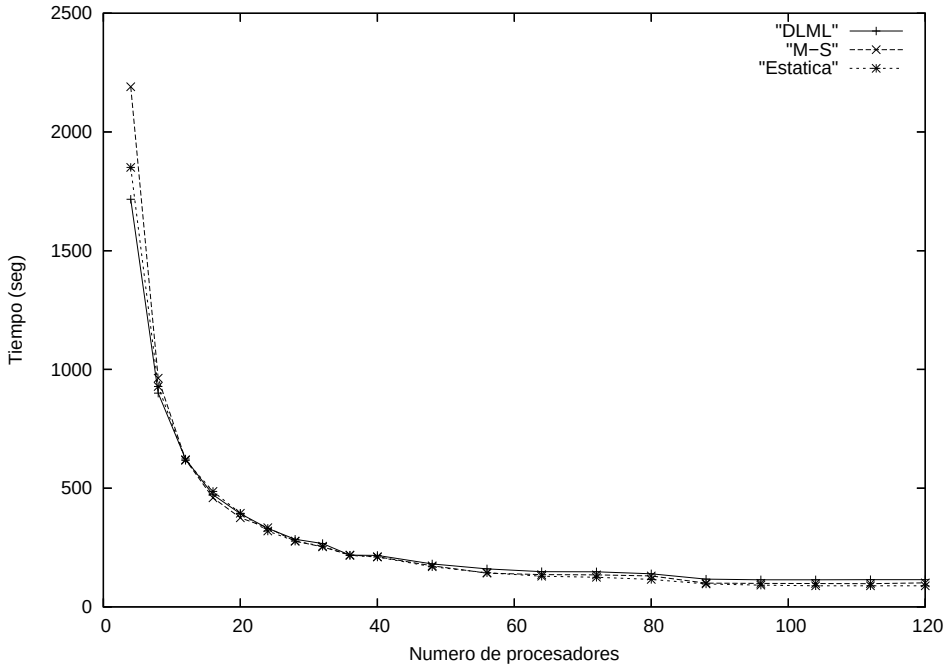

Figura 7.4: Segmentación para versiones DLML, Master-Slave y Estática en 4, 8, 12, ..., 32 y 40, 48, 56, ..., 120 procesadores

7.3.3. Segmentación de imágenes

La Figura 7.4 muestra el tiempo de respuesta de la Segmentación bajo un ambiente dedicado. El cluster utilizado fue C; la configuración de balance fue subasta global. Las configuraciones en la figura corresponden a la versión DLML, Master-Slave (M-S) y Estática corriendo desde 4 hasta 120 procesadores. La fuente de desbalance es el particionamiento dinámico de la imagen.

De la gráfica observamos que para un número de procesadores menor a 32, DLML tiene menores tiempos de respuesta que la versión estática y es similar a los tiempos obtenidos por la versión M-S. Para plataformas entre 32 y 120 procesadores, la versión estática y M-S obtienen ligeramente menores tiempos de respuesta que DLML. Esto es debido al tamaño del problema ya que al aumentar el número de procesadores ya no es posible disminuir el tiempo. Esto ocasiona que DLML de manera general genere más sobrecarga que las otras dos versiones (al recolectar globalmente los estados de carga)

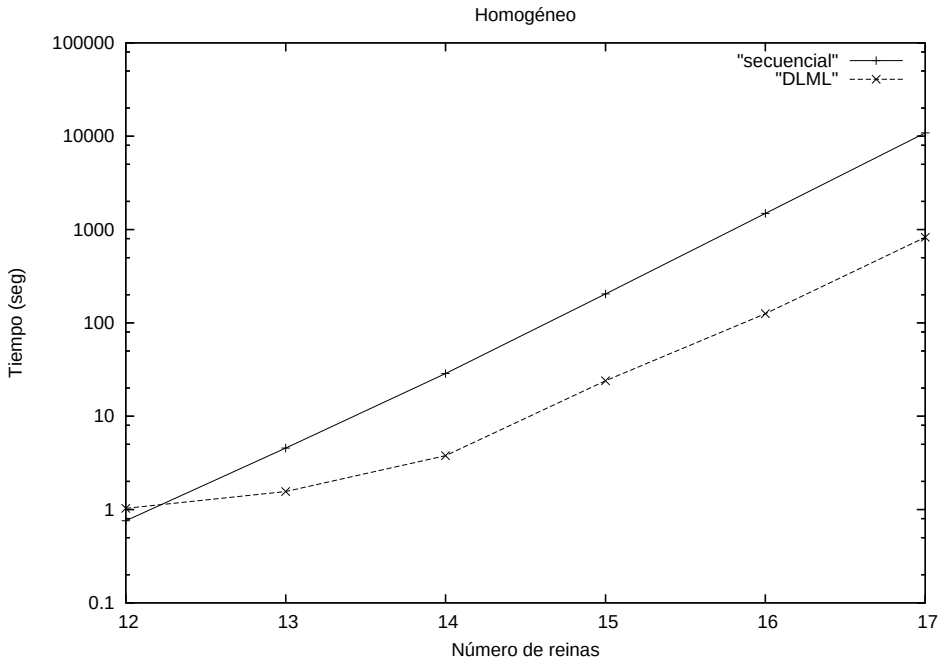

Figura 7.5: Tiempos de ejecución para N-reinas en secuencial y DLML

y por lo tanto incremente un poco sus tiempos de respuesta.

7.3.4. N-Reinas

La Figura 7.5 muestra el tiempo de respuesta (escala log_{10}) de las N-Reinas bajo un ambiente dedicado. Se utilizó una máquina secuencial (Pentium IV a 3.4GHz) y el cluster B; la configuración de balance fue subasta global. Las configuraciones en la figura corresponden a la versión secuencial y DLML (cluster homogéneo de 17 procesadores). La fuente de desbalance es el árbol de búsqueda de la aplicación.

De la figura se observa que DLML fue mejor (en gran medida) para $N \geq 13$, para valores menores fue mejor la versión secuencial, principalmente por el pequeño número de soluciones y el costo del protocolo de mensajes de DLML. Para N=12 hay 14200 soluciones, mientras que para N=13 y N=17 hay 73212 y 95815104 soluciones respectivamente. En el caso de N=17 los tiempos de respuesta de redujeron de 180.6 minutos a 13.7 minutos.

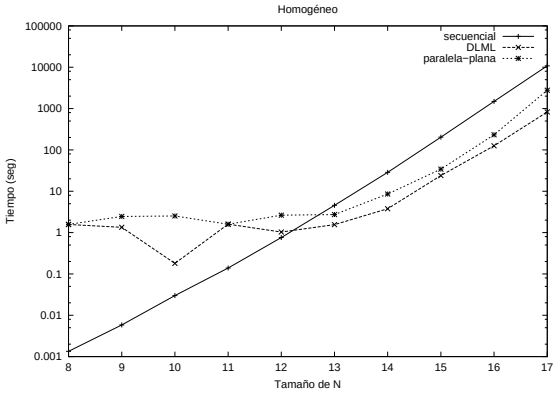

Figura 7.6: Tiempos de respuesta en un cluster homogéneo

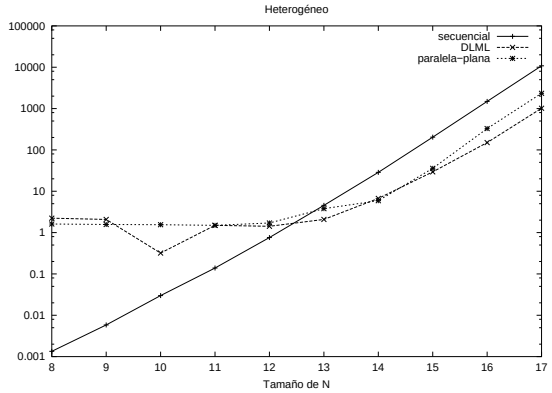

Figura 7.7: Tiempos de respuesta en un cluster heterogéneo

También hicimos otras evaluaciones para cuantificar los efectos del balance en cuanto a tiempo y distribución del procesamiento de datos. Con este propósito probamos una versión paralela plana (repartición equitativa y definitiva) y la versión DLML. Con ambas versiones medimos tiempos de respuesta y distribución de procesamiento.

Las Figuras 7.6 y 7.7 muestran el tiempo de respuesta (escala log_{10} de las N-Reinas en un ambiente dedicado. En ambas versiones se utilizó el cluster B pero con nodos homogéneos y heterogéneos respectivamente. Las configuraciones de ambas figuras corresponden a la versión DLML (con subasta global) y la versión paralela-plana (distribución inicial equitativa). La fuente de desbalance es el árbol de búsqueda y el cluster heterogéneo.

De las figuras se observa que para clusters homogéneos y heterogéneos DLML es mejor que las versión paralela-plana y secuencial para $N \geq 13$. Para $N \leq 12$ DLML tiene mayores tiempos comparado con las otras versiones. Esto se debe como ya se dijo con anterioridad al costo adicional a la aplicación que genera DLML el cual no es comparable con el tiempo de respuesta (≈ 1 segundo) para ese tamaño de problema.

Las Figuras 7.8 y 7.9 muestran la distribución de elementos procesados (millones) por cada procesador en un ambiente dedicado. En ambas versiones se utilizó el cluster B pero con nodos homogéneos y heterogéneos respectivamente. Las configuraciones

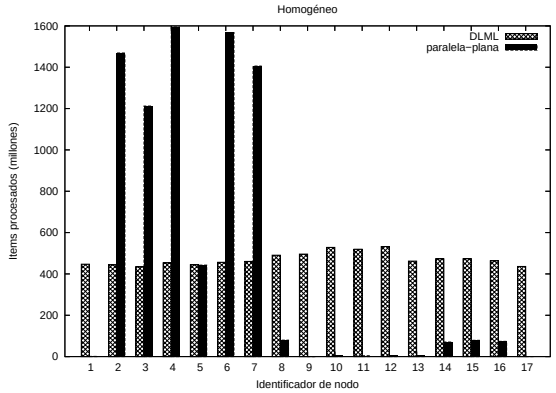

Figura 7.8: Distribución en cluster homogéneo del problema de las N-Reinas para N=17

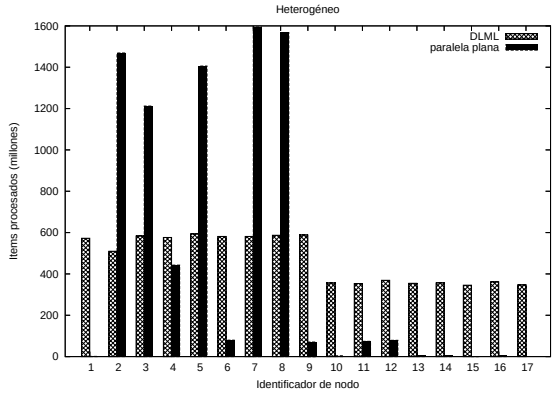

Figura 7.9: Distribución en cluster heterogéneo del problema de las N-Reinas para N=17

de ambas figuras corresponden a la versión DLML (con subasta global) y la versión paralela-plana (distribución inicial equitativa). La fuente de desbalance es el árbol de búsqueda y el cluster heterogéneo.

De la figura 7.8 notamos una distribución completamente irregular para la versión DLML sin el balance activado, esto se debió a la naturaleza dinámica de la aplicación donde una distribución inicial equitativa no garantiza que se mantendrá así hasta el final. Por el contrario, en la versión DLML con el balance activado, se nota una distribución uniforme en los 17 nodos homogéneos del cluster. De la figura 7.9 (cluster heterogéneo), notamos nuevamente un distribución irregular en el procesamiento de los datos, para la versión DLML sin balance (por las razones antes descritas), pero en la versión DLML con balance se nota una uniformidad en los nodos con capacidades similares. En los nodos con mayor capacidad (nodo 1-9) se procesó un mayor número de elementos de la lista, mientras que en los nodos con menor capacidad se procesaron menos elementos. Esto último hace notar que la distribución está en función del poder de procesamiento de los nodos.

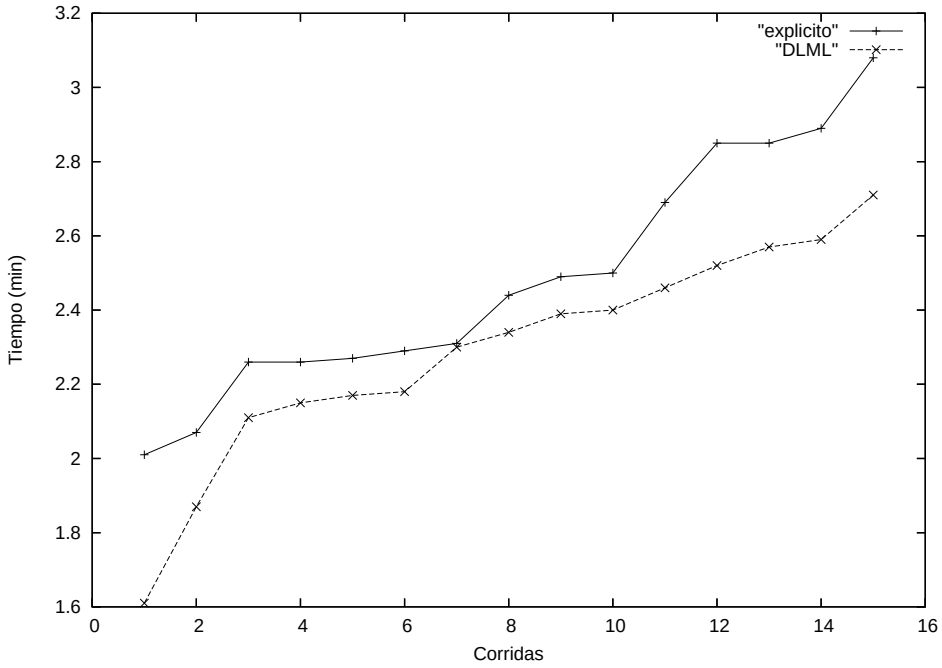

Figura 7.10: Tiempos de respuesta de la versión estática y DLML para NRIR

7.3.5. NRIR

En la Figura 7.10 muestra el tiempo de respuesta de NRIR (Non-Rigid Image Registration) bajo un ambiente dedicado. El cluster utilizado fue B; la configuración de balance fue subasta global. Las configuraciones de la figura corresponden a la distribución explícita y DLML. La fuente de desbalance es la misma aplicación.

De la figura podemos ver los tiempo de respuesta calculados en minutos para 15 corridas diferentes de NRIR. La gráfica muestra que en todos los casos la versión con DLML tiene menores tiempos de respuesta en comparación con la versión estática. La razón evidente es que una distribución estática (versión estática) en la programación paralela no beneficia mucho a aplicaciones de naturaleza dinámica como es el caso. Recordar que en NRIR el procesamiento de algunos cuadrantes de la imagen normalmente es diferente al procesamiento de otros cuadrantes.

7.4. Discusión

En general se observa de todas las gráficas anteriores que DLML tiene mayores ventajas sobre versiones explícitas programadas con MPI o con versiones paralelas sin balance, precisamente cuando existen condiciones de desbalance. Estas condiciones son principalmente la multiprogramación (ambiente no-dedicado), la heterogeneidad del cluster y las aplicaciones dinámicas (el procesamiento de datos genera más datos, además de que el procesamiento de cada dato puede ser diferente). Por el contrario si no existen estas condiciones de desbalance, DLML no resulta ser el mejor pero sigue siendo competitivo (similar) a las demás versiones. Cabe aclarar que algo importante es el aspecto cualitativo, donde la programación con DLML se facilita en comparación con la programación explícita.

7.5. Resumen

En este capítulo se mostró la evaluación que hicimos de DLML, usando 3 diferentes tipos de clusters y dos diferentes versiones de algoritmos de distribución. La evaluación se hizo en ambientes dedicados y no dedicados.

Para la evaluación descrita con anterioridad se utilizaron aplicaciones con cantidad fija de datos con procesamiento constante y variable y aplicaciones con una cantidad variable de datos con procesamiento constante y variable de datos.

Las aplicaciones fueron: clasificación en base a algoritmos evolutivos, mergesort, segmentación de imágenes, n-reinas y NRIR.

En la aplicación de clasificación se obtuvieron buenos resultados usando el balance de carga comparado sin usar balance de carga. El algoritmo de balance fue global distribuido.

En la aplicación de ordenamiento paralelo se probaron dos versiones, una versión implícita ad-hoc programada con MPI y otra versión con DLML. Ambas versiones fueron probadas en un cluster dedicado y no-dedicado. En la evaluación del conjunto de

pruebas se obtuvieron tiempos similares para la versión implícita y DLML, tanto en clusters dedicados y no dedicados. En esta aplicación nuevamente se utilizó el algoritmo global distribuido, sin embargo, esta versión se abandonó por el ajuste manual de parámetros del algoritmo por parte del programador.

Se evaluó una aplicación de segmentación de imágenes o reconstrucción en 3D. Para esta aplicación se implementaron tres diferentes versiones, estática, maestro-esclavo y DLML. En la versión estática el número total de cortes se divide entre el número total de procesadores. En maestro-esclavo un procesador central contiene todos los cortes y estos le son solicitados uno a uno por los esclavos. Con DLML inicialmente un procesador tiene todos los cortes pero estos se van distribuyendo con el algoritmo de subasta. Nuevamente con estas versiones se obtuvieron los tiempos de respuesta usando desde 4 hasta 120 procesadores. La evaluación mostró que DLML es mejor en comparación con la versión estática y maestro-esclavo para un número de procesadores menor 32. A partir de ese número y hasta 120 procesadores los tiempos de DLML son ligeramente superiores a las otras dos versiones. De estos resultados hacemos notar dos cosas: la primera es que el procesamiento de la aplicación no es tan costoso y para un número mayor a 32 procesadores ya no hay decremento en los tiempos para la tres versiones. La segunda es que en este caso DLML tiene tiempos de respuesta ligeramente mayores que las otras dos versiones debido al envío y recepción de mensajes generado por el manejo de la información global del algoritmo de subasta.

En la aplicación dinámica de las N-Reinas se comparó una versión secuencial, DLML y una versión paralela plana (repartición equitativa y definitiva). Para estas tres versiones se obtuvieron los tiempos de respuesta y para las dos última versiones (DLML y paralela plana) se obtuvieron la distribución de los datos procesados. Todo esto fue probado en un cluster homogéneo y no homogéneo. La evaluación muestra que con DLML los tiempos de respuesta se pueden llegar a reducir hasta en un 92 % en comparación con la versión secuencial. En cuanto a la distribución de los datos se nota la eficiencia de la versión DLML donde el procesamiento de los datos se hizo en función de las capacidades de cada nodo, el cual incluye la velocidad de procesador y la cantidad

de memoria principalmente.

Otras aplicaciones que se evaluaron fueron dos aplicaciones de procesamiento digital de imágenes. La primera es una aplicación de "registro de imágenes no rígido", esta consiste en encontrar un vector de transformación entre una imagen fuente y una destino, mediante un algoritmo dividir para vencer. El algoritmo consiste en que el vector de transformación encontrado se aplica a la imagen fuente completa y se compara la imagen obtenida con la imagen destino, si la comparación está por abajo de cierto valor, el procesamiento termina; sino la imagen se divide en cuatro partes y comienza el procesamiento con uno de los cuadrantes. Nuevamente si se cumple una similitud el procesamiento termina, sino se vuelve a dividir entre cuatro y así hasta completar la imagen. Esta aplicación dinámica se programó en una versión paralela explícita con 4 procesadores y con DLML. La evaluación, dirigida a tiempos de respuesta, mostró que DLML reduce los tiempos de respuesta obtenidos por la versión explícita. La razón es que DLML hace un balance de datos, el cual es necesario en la aplicación ya que unos cuadrantes de las imágenes requieren más procesamiento que otros.

Finalmente se presentó una discusión donde DLML tiene mejores resultados en ambientes con desbalance, esto comparado con versiones explícitas o paralelas sin balance. En ambientes sin desbalance DLML tiene un menor rendimiento en comparación con estas dos versiones, no obstante, cabe mencionar que la programación con lista de DLML es más sencilla que en la versión explícita.

Capítulo 8

Conclusiones y Trabajo Futuro

Esta tesis ha presentado DLML (Data List Management Library), una biblioteca de programación con listas de datos para cómputo paralelo en clusters.

La programación con DLML, como con otros tipos de datos abstractos permite que varios aspectos del paralelismo se encapsulen dentro de las operaciones o funciones de sus ADT . Principalmente los aspectos de comunicación y el manejo de datos. De esta forma el usuario programa de manera tradicional usando los llamados del ADT colocados en una biblioteca.

Con DLML usamos las listas. Como sabemos la lista es una secuencia de elementos sin orden, la cual permite insertar y eliminar en cualquier parte de la lista. El no tener orden permite también que una lista se pueda dividir para formar pequeñas sublistas conservando siempre su propiedad de listas. Además al tener varias sublistas, éstas se pueden distribuir fácilmente entre los procesadores y así poder procesarse de manera paralela. En DLML los datos de una aplicación se organizan como elementos que conforman una lista. Considerando que la lista se puede dividir y distribuir entre los procesadores, lo mismo sucede con los datos. Este comportamiento se abstrae a nivel de uso de las funciones típicas de listas como *get()* e *insert()*.

Con las funciones DLML, el usuario manipula los elementos de su lista de manera tradicional, no obstante, de manera transparente los datos son distribuidos en todo el

sistema paralelo. Con el llamado `get()` un procesador obtiene elementos de su lista local para procesarlos, pero si no existen, DLML busca elementos en otras listas remotas, haciendo una redistribución de los elementos de la lista.

Para facilitar la programación con DLML, propusimos una metodología para el desarrollo de aplicaciones con listas. La metodología consiste en una serie de pasos que incluyen tres bloques principales: inicialización, procesamiento y finalización. La inicialización contiene la determinación de los datos que se manejan en listas, la selección de la granularidad de la lista y la asignación de datos a cada elemento de la lista. El bloque de procesamiento define propiamente lo que se hace con cada elemento de la lista (procesamiento característico de la aplicación) una vez que se ha obtenido de la lista. Finalmente el bloque de terminación incluye recopilar datos parciales si los hay, imprimir resultados, cerrar archivos y finalizar procesos, entre otros.

En cuanto al diseño de DLML propusimos una arquitectura con 2 procesos por procesador. Un proceso ejecuta propiamente el código definido para la *Aplicación*, y otro proceso llamado *DLML* se encarga de distribuir los datos de acuerdo a la capacidad de procesamiento de los nodos, es decir, balancea la carga del sistema.

El balance de carga que implementamos es un algoritmo de subasta o *bidding* global. En la subasta un procesador que se queda sin carga (sin datos que procesar) subasta su poder de procesamiento. Para ello lanza un mensaje al resto de los procesadores quienes responden con su mejor oferta (longitud actual de su lista). La mejor oferta la da el procesador cuya longitud de lista es más grande que las demás (en caso de empates se elige a aquel procesador cuyo identificador sea menor que los demás). Posteriormente, el procesador que ganó la subasta, cede parte de su lista al procesador sin carga. Esta implementación permite disminuir el desbalance provocado por: aplicaciones con generación dinámica de datos (los datos a procesar se van generando a tiempo de ejecución), clusters con multiprogramación o no-dedicados y/o clusters heterogéneos. En los clusters con multiprogramación algunos nodos son usados por otros usuarios externos lo que provoca un desbalance. En un cluster heterogéneo la diferencia en la capacidad del procesamiento de los nodos es causa de un desbalance.

Para evaluar el desempeño de DLML propusimos una serie de experimentos los cuales consideran circunstancias de desbalance. Las circunstancias que incluimos son aplicaciones con generación dinámica de datos, cluster con multiprogramación y cluster heterogéneo. Esto fue probado en tres diferentes configuraciones de cluster, las cuales van desde una configuración de 8 procesadores en arquitectura monoprocesador, hasta una configuración de 120 procesadores en arquitectura tetra-procesador. Las aplicaciones que se evaluaron fueron: la clasificación en base a algoritmos evolutivos, mergesort paralelo, segmentación de imágenes, n-reinas y el registro no rígido de imágenes.

En general, DLML reduce/mejora el tiempo de respuesta bajo tales condiciones de desbalance. En la clasificación con algoritmos evolutivos comparamos DLML en su versión con el balance y sin balance en un ambiente dedicado y no-dedicado. En este escenario la versión con balance resulta mejor por la redistribución de individuos que hace entre los procesadores (en la aplicación el procesamiento de cada individuo es diferente) y por el desbalance del ambiente no-dedicado. En la versión dedicada los tiempos se reducen un 10 % y en la no-dedicada un 22 % [27]). En la aplicación de MergeSort paralelo comparamos una versión paralela explícita y DLML para ambientes dedicados y no dedicados. En este escenario los tiempos de ambas versiones son comparables en los ambientes dedicados y también en no-dedicados. Esto se debe a que el balance de DLML se contrarresta con la versión explícita adhoc. En la segmentación de imágenes comparamos DLML con una versión estática y maestro-esclavo en un cluster con nodos multiprocesador. Es este escenario DLML reduce los tiempos de la versión estática y de maestro-esclavo debido a que el costo de cada corte de imagen es diferente y va en función de la intensidad de sus pixels. En la aplicación de N-reinas comparamos a DLML con una versión paralela plana (sin balance) en un cluster homogéneo y heterogéneo. En esta comparación evaluamos los tiempos de respuesta y la distribución de los datos a procesar. En este escenario DLML reduce los tiempos (en cluster homogéneo y heterogéneo) debido a que la aplicación es dinámica. En cuanto al procesamiento de los datos, DLML los distribuye de acuerdo a la capacidad de los procesadores. Finalmente en la aplicación NRIR (registro de imágenes no-rígido) comparamos DLML con una

versión paralela explícita. En este escenario DLML gana a la versión explícita debido a que el procesamiento de cada cuadrante de la imagen varía en función de la intensidad de sus pixels y del grado de comparación entre una imagen alineada y una imagen fuente.

En suma, las principales contribuciones de nuestro trabajo son:

- El diseño de un ambiente para programación con listas de datos para cómputo paralelo en clusters.
- Una implementación de tal diseño en una biblioteca de funciones paralelas para manejo de listas en C.
- Una metodología para la programación basada en listas.
- El diseño del algoritmo de subasta global para nuestro ambiente.
- La implementación del algoritmo de subasta global que permite un balance implícito de los datos entre los procesadores.
- Una evaluación preliminar bajo diferentes escenarios: DLML en clusters monoprocesador, dedicados y no-dedicados; y DLML en clusters multiprocesador homogéneos.

Limitaciones y Trabajo futuro

DLML tiene varias limitaciones. Una es en cuanto escalabilidad, ya que actualmente su algoritmo de subasta global no es escalable. En otras palabras, el algoritmo de subasta global tiene un mayor costo cuando se incrementa el número de procesadores, debido a los mensajes que utiliza en la búsqueda del mejor postor (procesador más cargado). Una alternativa es hacer una búsqueda parcial o por grupos. En relación a esto se tiene ya una propuesta de un algoritmo de información parcial.

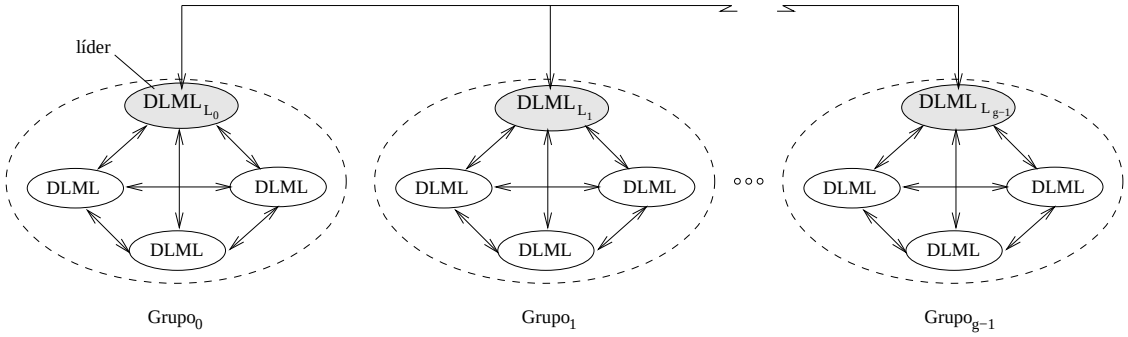

Figura 8.1: Distribución de datos con información parcial

El algoritmo con información parcial es similar al de información global en el sentido de que los procesos *DLML* son los encargados de la distribución de datos (elementos de la lista). Con información parcial, se maneja el concepto de *grupos* y *líderes de grupo*. Esto se ejemplifica en la Figura 8.1 donde se distinguen desde el $Grupo_0$ hasta el $Grupo_{g-1}$. En cada grupo existe un solo proceso *DLML* que es líder de grupo (desde L_0 hasta L_{g-1}) y cada grupo tiene el mismo número de procesos *DLML* que los demás (no obstante cada grupo podría tener diferente número de procesos DLML, por ejemplo 4,4,3). En esta implementación el líder de grupo es fijo durante la ejecución y se determina en base a su identificador. En el ejemplo de la Figura 8.1 se tienen 12 procesos DLML, identificados del 0 al 11. En este caso los líderes de grupo son el proceso 0, 4 y 8.

Haciendo esta descripción, el funcionamiento es el siguiente: cada grupo de procesos *DLML* sigue un algoritmo de distribución de datos con información global de manera independiente (Figura 6.2). Es decir, cada vez que un proceso *DLML* nota que su lista local esta vacía, inicia el algoritmo de distribución haciendo una búsqueda global de datos, pero en el contexto de su grupo (intragrupo). De ésta forma se evita hacer una búsqueda general (global) en todos los grupos, reduciendo el costo de las comunicaciones.

Bajo este contexto es posible que un grupo se quede sin elementos que procesar, es decir, que las listas de todos los procesos *DLML* en ese grupo se queden vacías. Cuando esto sucede el algoritmo hace una redistribución pero a nivel intergrupo, es decir, entre

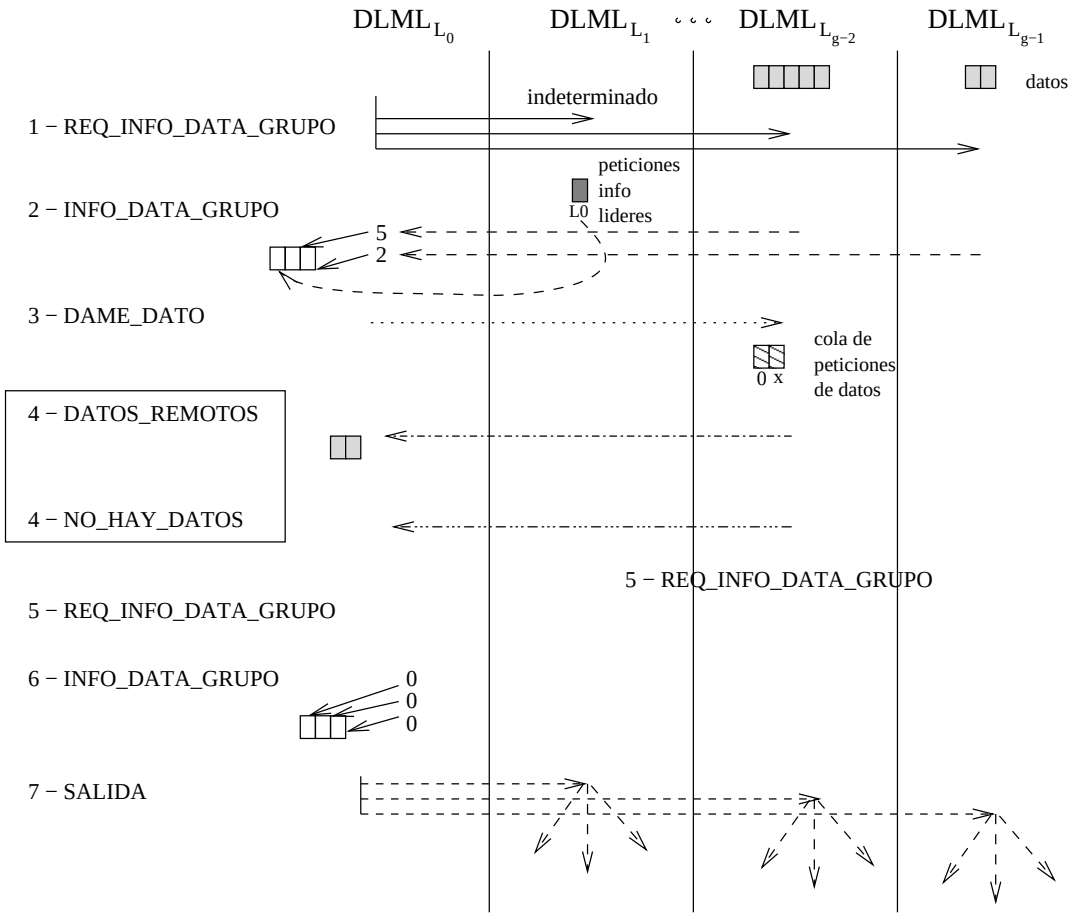

Figura 8.2: Distribución de datos entre líderes de grupo

grupos. Es ahí donde entran los procesos *DLML* líderes de grupo, quienes se encargan propiamente de la redistribución grupal mientras que los demás procesos del grupo (no líderes) pasan a un estado de *standby* (a espera de ser utilizados).

En la Figura 8.2 mostramos como es la distribución de datos entre los líderes de grupo ($0 \leq L \leq g-1$). De la Figura, cuando un grupo se queda sin datos, (en este caso el $Grupo_0$), el proceso DLML líder de grupo ($DLML_{L_0}$) inicia una subasta intergrupo solicitando información de carga (paso 1). En este caso la carga es la longitud de la lista de los líderes de grupo quienes responde con su carga, esto se nota en el paso 2 donde $DLML_{L_{g-2}}$ y $DLML_{L_{g-1}}$ responde con 5 y 2 respectivamente. Sin embargo $DLML_{L_1}$ no responde por estar en un estado indeterminado. Este estado se da cuando no se tienen

datos pero se están buscando o no se los han dado. Una vez que sale del estado envía la información. Cuando el proceso $DLML_{L_0}$ tiene toda la información busca al líder con el mayor número de elementos para pedirle datos (paso 3) en este caso $DLML_{Lg-2}$, quién envía los datos (DATOS_REMOTOS) o manda un mensaje indicando que ya no tiene datos (NO_HAY_DATOS). Si se da el último caso ambos líderes comienzan de nuevo la subasta intragrupo. El algoritmo termina cuando todos los líderes responden con un cero a cierto líder ($DLML_{L_0}$ en el paso 6), indicando que no tienen datos. Este líder se encarga de la terminación mandando un mensaje de SALIDA (paso 7), a todos los líderes quienes a su vez hacen lo mismo dentro de su grupo

Con este algoritmo parcial, DLML puede extenderse para ser usado en grids (clusters de clusters) donde cada grupo sería un cluster y cuando no exista más carga que procesar en ese cluster, se busca en otros grupos (cluster remoto). Esta implementación con información parcial tiene como limitante el estado indeterminado que hace que los demás líderes esperen hasta que el proceso indeterminado salga de su estado. Otra limitante de la implementación es que sólo se tiene un nivel 1 de grupos y no una jerarquización (grupos de grupos (nivel 2) o grupos de grupos de grupos (nivel 3)). En este caso el líder de líderes nuevamente sería el líder con menor identificador de proceso. En resumen la propuesta puede ser rediseñada y optimizada.

En los experimentos que realizamos en la tesis siempre se manejo una granularidad fija pero se pudieron haber hecho las mismas pruebas con una granularidad variable. Este estudio podría mostrar cierto patrón que relacione el desempeño con la granularidad elegida. Por un lado es conveniente tener la granularidad más fina para un mejor paralelismo, pero por otro lado puede ser tan fino que sea más conveniente evaluar un elemento con más datos que con pocos datos. Otra sugerencia sobre DLML va en el sentido de reducir el número de envíos cuando se transfieren elementos de la lista (granularidad de balance). Actualmente cuando se hace un balance, se envía uno a uno los elementos de la lista. Esto se podría mejorar si en un solo mensaje se envían más de un elemento de la lista.

Otra limitación es que la programación es cuasi-secuencial mas no secuencial. Esto

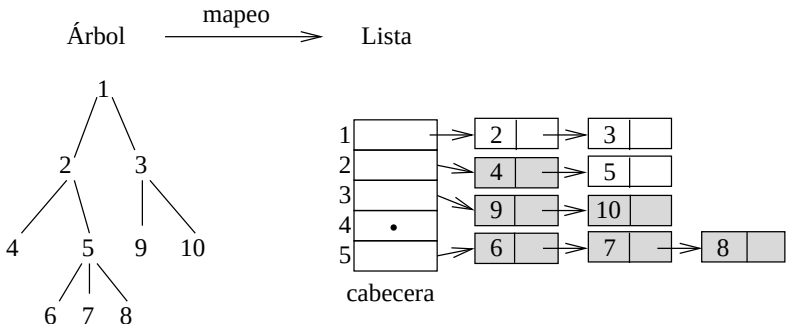

Figura 8.3: Mapeo de una estructura de árbol a estructura de lista

repercute en que el programador debe estar consciente que su programa realmente se ejecuta en paralelo, y que al final tiene recolectar los resultados parciales, si los hay, que se encuentren distribuidos en todos los procesadores (no obstante que DLML ofrece funciones para facilitar esa tarea). Una alternativa para hacer transparente la recolección es usar precompilación. La precompilación haría un análisis de una aplicación-α (alfa), donde la aplicación-α es la versión secuencial que no contempla recolección de datos; en sí, el programador lo da por hecho al suponer que se ejecuta en un solo procesador. La precompilación haría una búsqueda de todas aquellas variables que guardan resultados, por ejemplo en código de la aplicación-α buscarían instrucciones como $x = x + y$. En este caso todos los procesadores tendrían una parte de la suma final de x por lo que faltaría hacer una suma final entre todos los procesadores. Para estas instrucciones el precompilador construiría el código que realice el intercambio o podría usar la función DLML adecuada del tipo recopilación que hiciera dicha tarea. Después de aplicar el proceso de precompilación se tendría una versión β (beta) de la aplicación la cual incluiría los llamados adecuados para la recolección de datos y esta versión sería la que se corre bajo DLML.

Otras de las limitaciones y quizás las más importantes, es que DLML necesita que los datos de la aplicación sean representados en listas y que los datos se puedan procesar de manera independiente en cualquier procesador. Hasta ahora hemos contemplado aplicaciones cuyos datos se pueden representar fácilmente en listas y donde el procesamiento de estos es independiente (Subsección 5.2.2). Sin embargo, se pueden tener

aplicaciones cuyos datos se representan en otras estructuras como árboles y adicionalmente tienen una dependencia entre ellos, Figura 8.3 (izq). La dependencia consiste en que no se puede evaluar un nodo del árbol, mientras no se hayan evaluado sus hijos. De la figura sólo se puede evaluar el nodo 5, si se han evaluado sus hijos 6,7 y 8. En este sentido un trabajo futuro para llevar este tipo de aplicaciones hacia su uso con DLML podría ser un mapeo árbol → lista como el que se muestra en la misma Figura 8.3 (der). Bajo esta representación DLML podría hacer la distribución pero la dependencia limita procesamiento, por ejemplo el nodo 5 (final de la cabecera de lista 2), no se puede evaluar hasta que no hayan sido evaluados todos los elementos de la lista 5. Una alternativa para resolver este problema (aunque quizá no sea eficiente) es hacer un análisis de dependencias previo a la ejecución. El análisis se buscaría a todos aquellos elementos que no tienen dependencias. Los elementos que no tienen dependencias en el mapeo a lista son aquellos que no son parte de la cabecera (en la representación de árbol son las hojas). En la Figura 8.3 (der) se muestran sombreados los elementos sin dependencia. Estos elementos sin dependencias pueden ser distribuidos y procesados en cualquier procesador, pero los resultados de los elementos 6,7 y 8 podrían quedar en diferentes procesadores y en este caso el elemento 5 necesita de los resultados de los 3 elementos para poder continuar. Una alternativa o solución es que los nodos hermanos como 6,7 y 8 se mantengan unidos, es decir, que el procesamiento de los tres sólo lo haga un procesador. DLML podría tener una nueva función que en un lugar de dividir elementos, concatene varios de ellos. El siguiente paso sería un reenvío del procesamiento parcial de los 3 elementos hacia donde se necesita, en este caso el final de la lista 2. En resumen, debo admitir que la implementación se ve complicada, no obstante, es una línea a seguir.

Otra limitante es que DLML no es tolerante a fallas. Actualmente si un nodo o procesador se cae, DLML falla. A este respecto un posible camino sería que DLML maneje listas y nodos de respaldo. Para el respaldo de listas, cada proceso DLML guardaría el último estado de su lista en un servidor (recordar que en DLML se tiene una lista por procesador). Cabe mencionar que es necesario determinar una frecuencia

de respaldo, la cual en primera instancia estaría en función de la probabilidad de falla. Bajo este esquema cuando un nodo se cae, un demonio o proceso de recuperación se encargaría de tomar la copia respectiva de la lista guardada en el servidor y asignarla en uno de los nodos de respaldo. Este demonio se debe de encargar de la reetiquetación de nodos, principalmente al momento de la subasta, ya que el nodo caído no respondería (lo cual puede ser el inicio del proceso de recuperación). Si un nodo no responde después de cierto tiempo (por estar caído) se puede omitir su respuesta y hacer la subasta con los procesadores que contestaron. También puede ser que el demonio indique la nueva dirección del nuevo nodo con la lista recuperada.

En resumen, nuestro trabajo ofrece un ambiente de programación basado en listas que permite el cómputo paralelo en clusters. La programación se facilita al encapsular u ocultar la comunicación entre procesadores a través del llamado a funciones típicas sobre listas. Para ello proponemos una metodología que orienta el desarrollo de aplicaciones bajo listas. Adicionalmente, el ambiente incluye un algoritmo de balance que distribuye eficientemente entre los procesadores los datos de las listas. Nuestro ambiente lo hemos probado en diferentes aplicaciones obteniendo un buen desempeño y más aún cuando existen condiciones de desbalance.

Apéndice A

Implementación de DLML

DLML esta implementado en lenguaje C. Entre las razones por las que elegimos C, es por ser un estándar de programación en los sistemas Unix y Linux y por la eficiencia de su compilador.

A.1. MPI

Para la comunicación entre los procesos *DLML* utilizamos una de las bibliotecas de comunicación más difundidas en el área del cómputo paralelo y distribuido, *MPI (Message Passing Interface)* [48, 86, 87].

MPI, lo utilizamos principalmente para tres operaciones:

- Ejecutar varias instancias de la aplicación (programación SPMD). Cuando el programador utiliza DLML, lo primero que hace es un programa en C el cual se va a ejecutar en todos los procesadores, para ello MPI facilita la ejecución del código en todos los procesadores de manera paralela.

- Distribuir las listas entre los diferentes procesadores. Estas listas se pueden generar en un solo procesador para posteriormente ser distribuidas, o se pueden generar individualmente en cada uno de los procesadores. La distribución se lleva a cabo cuando los procesadores sin datos (elementos en su lista) buscan datos de otros

procesadores. En este caso, MPI facilita el envío y recepción de mensajes entre los procesos DLML para la distribución de datos.

- Recopilar resultados parciales. Las aplicaciones que se ejecutan con DLML, generalmente generan resultados parciales que se encuentran distribuidos en todos los procesadores, es decir, cada procesador, normalmente contiene una parte para determinar el resultado final. En este caso, nuevamente MPI facilita la recopilación de los resultados parciales en un solo procesador mediante el envío y recepción de mensajes.

Cabe mencionar que algunas de las funciones de DLML son envolturas (wrappers) de las funciones de MPI, principalmente en aquellas en la funciones de recopilación de resultados.

A.2. Requerimientos para la instalación de DLML

Para usar DLML es necesario tener ciertos requisitos de hardware y software, a continuación describimos cada uno de ellos.

Hardware

Lo principal es un cluster, el cual puede ser con nodos homogéneos o heterogéneos. Como ya dijimos, DLML optimiza la distribución sin importar el tipo de nodo. En cuanto a la conectividad de los nodos, el requerimiento es un simple concentrador.

El cluster se puede utilizar de manera dedicada o no-dedicada. En la primera opción se tiene un uso exclusivo del cluster al momento de ejecutar las aplicaciones. En la otra opción el cluster puede estar compartido por más de un usuario a la vez. Se resalta que en ambas opciones DLML también optimiza la ejecución de las aplicaciones en el cluster.

Software

Para poder ejecutar DLML es necesario el siguiente software:

- Sistema operativo. DLML puede correr en cualquier sistema operativo tipo Linux. Estos sistemas pueden ser Red-Hat, Fedora, Debian o Suse [64] entre otros. Este sistema operativo debe estar instalado en el cluster y tener configurado un sistemas de archivos de red como NFS (Network File Systems). El sistema de archivos en red es necesario para facilitar la ejecución distribuida de las aplicaciones.

- MPI-C. Como ya mencionamos DLML esta implementado con MPI. Por ello es necesario obtener e instalar algunas de las versiones gratuitas y de libre distribución de MPI-C como mpich [57], lam-mpi [93] o mvapich [94].

A.3. Instalación de DLML

Para programar con DLML es necesario hacer la instalación que se describe a continuación:

1. Crear un directorio de trabajo, por ejemplo:
   mkdir dlml_work

2. Descargar en el directorio anterior el archivo *DLML.tar.gz*. Este archivo puede ser descargado de la dirección
   http://computacion.cs.cinvestav.mx/~mcastro/DLML
   http://palancar.izt.uam.mx/~mcastro/DLML

3. Descomprimir y desempaquetar el archivo ejecutando la instrucción
   tar zxf DLML.tar.gz

4. Comenzar a programar.

A.4. Compilación y Ejecución

Para poder compilar la aplicación, esta previamente se salva con un *nombre* y la extensión *.C*. Después se invoca la siguiente línea de comando.

```
mpicc -o aplicacion aplicacion.c Opercolas.c -lm
```

En la línea anterior se invoca al compilador de MPI llamado *mpicc*. Este compilador revisa todas las instrucciones de MPI manejadas dentro de la funciones de DLML que van en la aplicación. Este mismo compilador revisa toda la sintaxis del lenguaje C que tiene la aplicación y sus llamados a DLML incluidos como la biblioteca dlml.h. Después viene el nombre del archivo de salida y el archivo fuente. El siguiente archivo *Opercolas.c* el cual es ligado al archivo de salida, contiene algunas operaciones necesarias para el control de las peticiones de datos en los procesadores. Finalmente *-lm* es una biblioteca matemática la cual puede ser o no necesaria dependiendo de las funciones matemáticas utilizadas en la aplicación.

Adicionalmente a la compilación de la aplicación, es necesario compilar a DLML. Esto se hace mediante la instrucción

```
mpicc -o DLML DLML.c Opercolas.c
```

Nuevamente la compilación se realiza mediante el compilador de MPI, dando como parámetros el archivo de salida, el de entrada y el archivo para la operación de colas descrito previamente.

Una vez completada la compilación, se puede proceder a la ejecución de la aplicación. Para ello es necesario dar de alta las máquinas (nodos) que contribuirán en la ejecución. En el caso de que la versión de MPI sea lam/mpi [93] la instrucción es:

```
lamboot -v maquinas
```

Donde máquinas es un archivo que contiene los nombres de los nodos a ser usados. En el caso de que los nodos sean de más de un procesador, este se indica con la palabra reservada *cpu*

```
maquina1
maquina2 cpu=2
maquina3 cpu=4
```

Después de dar de alta las máquinas que conforman la plataforma, se ello se ejecuta la `aplicación` junto con `DLML`. Para ellos se ejecuta la instrucción

```
mpirun -np 1 dlml_todo
```

Es este caso, la instrucción mpirun de MPI ejecuta una instancia del programa dlml_todo. Este programa tiene instrucciones que verifican el número de nodos y lo principal la instrucción MPI_Spawn_Multiple que se encarga de la asignación de los programas ejecutables `Aplicación` y `DLML` en todos los nodos del cluster (previamente dados de alta).

Apéndice B

Otras aplicaciones con programación con listas

B.1. Suma de matrices

La suma de matrices se representa por la ecuación:

$$C[i,j] = A[i,j] + B[i,j] \tag{B.1}$$

Donde A y B son las matrices a sumar y C es la matriz resultante. En la suma de dos matrices A y B (Figura B.1) se conocen de antemano los datos a procesar (las matrices mismas). La suma de dos matrices A y B es la suma de los elementos correspondientes

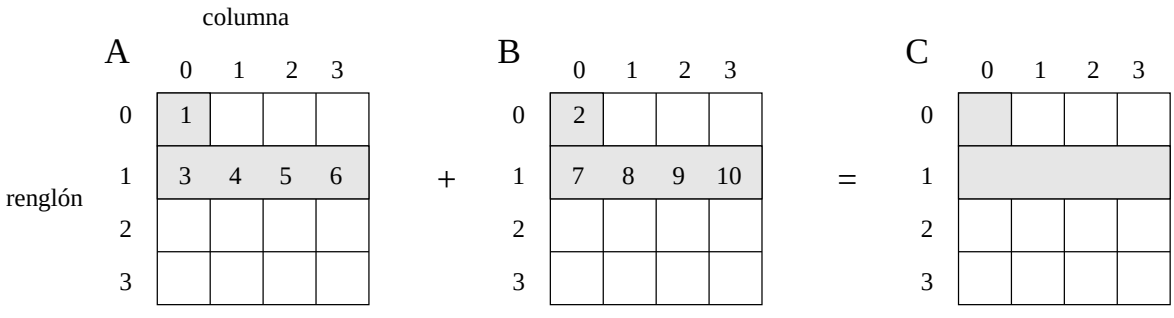

Figura B.1: Granularidad fina y mediana en la suma de matrices

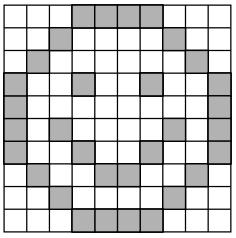

Figura B.2: Imagen para procesamiento digital

de las dos matrices. Para la programación con listas se puede tener una granularidad fina o mediana. En la fina, cada elemento de la lista contiene 3 campos: 1) las coordenadas (renglón,columna) de un elemento, 2) el valor de ese elemento en A y 3) el valor de ese elemento en B. En este caso el contenido de un elemento de la lista sería {(0,0),1,2}. Con esta granularidad cada coordenada define un elemento de la lista con 3 campos, haciendo un total de 16 elementos en la lista. Si se elige una granularidad mediana, por ejemplo renglones, cada elemento de la lista contiene el renglón y los valores de ese renglón en las matrices A y B. Por ejemplo un elemento de la lista sería {1,{3,4,5,6},{7,8,9,10}}, haciendo un total de 4 elementos en la lista. El procedimiento posterior es seguir los pasos 3 al 6 de la metodología.

B.2. Procesamiento digital de imágenes

Para el procesamiento digital, se puede tener una imagen tan simple como la carita feliz en la Figura B.2. Esta es una imagen de 10 x 10 pieles (100 pieles en total). Si el procesamiento consiste en evaluar cada uno de los pieles, el procesamiento es similar a la suma de 100 números que explicamos con anterioridad (Subsección 5.2.1). De igual manera que en esa aplicación se puede tener varias granularidades desde un pixel (100 elementos), un renglón de la imagen (10 elementos), o la imagen completa (1 elemento). Esta granularidad se mostró de manera similar en la suma de 100 números, Figura 5.2.

Una vez elegida la granularidad se procede a definir la estructura adecuada que define a un elemento de la lista, posteriormente se hacen las inserciones y finalmente

un ciclo donde se procesen uno a uno los pieles o los renglones hasta terminar.

Bibliografía

[1] *Linux Programmer's Manual*.

[2] Alfred V. Aho, J. D. Ullman, and J. E. Hopcroft. *Data Structures and Algorithms*. Pearson Education, first edition, January 1983.

[3] 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, June 2006.

[4] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin Cummings, second edition, 1994.

[5] AMD. Advanced micro devices. Website URL: http://www.amd.com/la-es/, 2007.

[6] 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, February 1996.

[7] Rob T. Aulwes, David J. Daniel, Nehal `N.` Desai, Richard L. Graham, L. Dean Risinger, Mark A. Taylor, Timothy S. Woodall, and Mitchel W. Sukalski. Architecture of la-mpi, a network-fault-tolerant mpi. In *18th International Parallel and Distributed Processing Symposium (IPDPS04)*, page 15b, April 2004.

[8] Henri E. Bal, Jennifer F. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[9] Ray Barriuso and Allan Knies. Shmem user's guide for c. Technical report, Cray Research Inc., Eagan, MN., June 1994.

[10] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In *EuroPar 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 761–770. Springer-Verlag, June 2005.

[11] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Using eskel to implement the multiple baseline stereo application. *Parallel Computing: Current & Future Issues of High-End Computing,*, 33:673–680, October 2006.

[12] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software Practice and Experience*, 25(S4):87–130, 1995.

[13] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, JakovÑ. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[14] George Horatiu Botorog and Herbert Kuchen. Efficient parallel programming with algorithmic skeletons. In *Europar 96*, volume 1123 of *Lecture Notes in Computer Science*, pages 718–731. Springer Verlag, 1996.

[15] George Horatiu Botorog and Herbert Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252, August 1996.

[16] Azzedine Boukerche, Alba Cristina M. A. Melo, Jeferson G. Koch, and Cicero R. Galdino. Multiple coherence and coordinated checkpointing protocols for dsm

systems. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 531–538. IEEE Computer Society, June 2005.

[17] Barry B. Brey. *INTEL Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium ProProcessor, Pentium II, III, 4*. Prentice-Hall, 7th edition, March 2005.

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

[19] Jorge Buenabad-Chávez, Miguel A. Castro-García, and Graciela Román-Alonso. Simple, list-based parallel programming with transparent load balancing. In *Parallel Processing and Applied Mathematics*, volume 3911 of *Lecture Notes in Computer Science*, pages 920–927. Springer-Verlag, September 2005.

[20] Rajkumar Buyya, editor. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice Hall, first edition, May 1999.

[21] Duncan K. G. Campbell. Towards the classification of algorithmic skeletons. Technical report, Department of Computer Science, University of York, December 1996.

[22] Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko, and Sang Lim. mpijava 1.2: Api specification. Technical report, Northeast Parallel Architectures Centre, Syracuse University, 2002.

[23] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. Mpj: Mpi-like message passing for java. *Concurrency: Practice and Experience*, 12(11):1019–1038, September 2000.

[24] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[25] Norma Pilar Castellanos, Pedro Luis del Angel, and Veronica Medina. Nonrigid medical image registration technique as a composition of local warpings. *Pattern Recognition*, 37(11):2141–2154, 2004.

[26] Miguel Alfonso Castro-García. Balance dinámico de carga en un sistema paralelo con memoria distribuida. Master's thesis, Centro de Investigación y de Estudios Avanzados del IPN, Mayo 2001.

[27] Miguel Alfonso Castro-García, Graciela Román-Alonso, Jorge Buenabad-Chávez, Alma Edith Martínez-Licona, and John Goddard-Close. Integration of load balancing into a parallel evolutionary algorithm. In *Advanced Distributed Systems*, volume 3061 of *Lecture Notes in Computer Science*, pages 219–230. Springer-Verlag, January 2004.

[28] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, October 2000.

[29] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[30] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1):22–31, 1999.

[31] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A hardware/software approach*. Morgan Kaufmann, first edition, August 1998.

[32] Nell B. Dale and Henry M. Walker. *Abstract Data Types: Specifications, Implementations and Applications*. Houghton Mifflin Company College Division, first edition, March 1996.

[33] 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, December 1994.

[34] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W.Ñ. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE '93, Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer Verlag, 1993.

[35] Santiago Domínguez Domínguez. *Difusión Automática de Datos bajo Cómputo Paralelo en Clusters*. PhD thesis, Centro de Investigación y de Estudios Avanzados del IPN, Sección Computación, Enero 2006.

[36] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12(5):662–675, 1986.

[37] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6(1):53–68, March 1986.

[38] Domenico Ferrari and Songnian Zhou. A load index for dynamic load balancing. In *Proceedings of 1986 ACM Fall joint computer conference*, pages 684–690. IEEE Computer Society Press, 1986.

[39] Domenico Ferrari and Songnian Zhou. An empirical investigation of load indices for load balancing applications. In *Proc. Performance '87, the 12th Int'l Symp. on Computer Performance Modeling, Measurement and Evaluation*, pages 515–528, 1987.

[40] Michael Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[41] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison Wesley, first edition, January 1995.

[42] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, Kremmer, C. Tseng, and M. Wu. Fortran d language specification. Technical report, Dept of Computer Science, Rice University, 1990.

[43] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A Users'Guide and Tutorial for Network Parallel Computing*. MIT Press, first edition, December 1994.

[44] Juan R. González, Coromoto Léon, and Casiano Rodríguez. An asynchronous branch and bound skeleton for heterogeneous clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *LNCS*, pages 191–198. Springer-Verlag Berlin Heidelberg, September 2004.

[45] D. M. Goodeve, S. A. Dobson, J. M. Nash, J. R. Davy, P. M. Dew, M. Kara, and C. P. Wadsworth. Towards a model for shared data abstration with performance. *Journal of Parallel and Distributed Computing*, 49(1):156–167, February 1998.

[46] Don Goodeve, John Davy, and Chris Wadsworth. Shared accumulators. In *World Transputer Congress*, 1995.

[47] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, second edition, January 2003.

[48] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Scientific and Engineering Computation. The MIT Press, second edition, November 1999.

[49] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[50] Benjamin Mako Hill, David B. Harris, and Jaldhar Vyas. *Debian GNU/Linux 3.1 Bible*. Wiley, bk&cd-rom edition, August 2005.

[51] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, May 1986.

[52] Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed. *Fundamentals of Data Structures in C*. W. H. Freeman Company, first edition, September 1992.

[53] Paul Hudson and Andrew Hudson. *Red Hat Fedora 5 Unleashed*. SAMS, first edition, May 2006.

[54] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. Crl: High-performance all-software distributed shared memory. *ACM SIGOPS Operating Systems Review , Proceedings of the fifteenth ACM symposium on Operating systems principles SOSP '95*, 29(5):213–228, December 1995.

[55] Brian W. Kernighan and Dennis M Ritchie. *The C Programming Language*. Pearson Education, second edition, March 1989.

[56] Herbert Kuchen, Rinus J. Plasmeijer, and Holger Stoltze. Efficient distributed memory implementation of a data parallel functional language. In *PARLE '94, Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*, pages 464–477. Springer Verlag, 1994.

[57] Argonne-National Laboratory. Mpich2 home page. Web page http://www-unix.mcs.anl.gov/mpi/mpich/, April 2007.

[58] Yedidyah Langsam, Moshe J. Augenstein, and Aaron M. Tenenbaum. *Data Structures Using C and C++*. Prentice-Hall, second edition, December 1995.

[59] Gil-Haeng Lee. Issues of the state information for location and information policies in distributed load balancing algorithm. In *EUROMICRO Conference, Informatics: Theory and Practice for the New Millennium*, volume 1, pages 67–70. IEEE, 1999.

[60] Cluadia Leopold. *Parallel and Distributed Computing: A survey of models, paradigms, and approaches*. Wiley-Interscience, first edition, November 2000.

[61] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Dept. of Computer Science, Yale University, 1986.

[62] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38, 1987.

[63] Kapil K. Mathur and S. Lennart Johnsson. All-to-all comunications on the connection machine cm-200. Technical report, Center for Research in Computing Technology, Harvard University, January 1993.

[64] Mike McCallister. *SUSE Linux 10.0 Unleashed*. SAMS, book&dvdrm edition, November 2005.

[65] Randal K. Michael. *AIX 5L Administration*. McGraw-Hill, first edition, August 2002.

[66] Jonathan Nash, Peter Dew, and Martin Berzins. Using sadts to support irregular computational problems. In *I-SPAN'99 Int. Symposium on Parallel Architectures, Algorithms and Networks*, pages 338–343. IEEE Computer Society, 1999.

[67] Jonathan M. Nash, Peter M. Dew, and John R. Davy. A parallelisation approach for supporting scalable and portable computing. Technical Report Report 97.20, University of Leeds, 1997.

[68] Jonathan M. Nash, Peter M. Dew, John R. Davy, and Martin E. Dyer. Implementation issues relating to the wpram model for scalable computing. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, volume 1124-II of *Lecture Notes in Computer Science*, pages 319–326. Springer Verlag, 1996.

[69] Greg Nelson. *Systems Programming with Modula-3*. Prentice-Hall, May 1991.

[70] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly, first edition, September 1996.

[71] Julio C. Peralta, Jorge Buenabad-Chávez, and Santiago Domínguez-Domínguez. Compiler support for an all-software distributed shared memory. April 2006.

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

[73] Pawel Plaszczak and Richard Wellner. *Grid Computing: The Savvy Manager's Guide*. Morgan Kaufman, August 2005.

[74] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, second edition, September 1993.

[75] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, first edition, June 2003.

[76] Fethi A. Rabhi. *Abstract machine models for highly parallel computers*, chapter Exploiting parallelism in functional languages: a "paradigm-oriented" approach, pages 118–139. Oxford University Press, 1995.

[77] Marc Rochkind. *Advanced Unix Programming*. Prentice-Hall, second edition, April 2004.

[78] Graciela Román-Alonso, Norma Pilar Castellanos-Abrego, Jorge Buenabad-Chávez, and Miguel A. Castro-García. Improving a parallel version of a non-rigid image registration algorithm. In *Advanced Distributed Systems*, Lecture Notes in Computer Science. Springer-Verlag, January 2006.

[79] Graciela Román-Alonso, Miguel A. Castro-García, and Jorge Buenabad-Chávez. Easing message-passing parallel programming through a data balancing service. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*,

volume 3241 of *Lecture Notes in Computer Science*, pages 295–302. Springer-Verlag, September 2004.

[80] Graciela Román-Alonso, Juan Ramón Jiménez-Alanis, Jorge Buenabad-Chávez, Miguel Alfonso Castro-García, and Abraham Helios Vargas-Rodríguez. Segmentation of brain image volumes using the data list management library. September 2006.

[81] Tom Shanley. *InfiniBand Network Architecture*. Addison-Wesley, first edition, October 2002.

[82] Behrozz A. Shirazi, Ali R. Hurson, and Krishna M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[83] Silicon-Graphics. Sgi® origin 3000. Web page http://www.sgi.com/pdfs/3399.pdf, August 2003.

[84] David Skillicorn. *Foundations of Parallel Programming*. Cambridge International Series on Parallel Computation:6. Cambrigde University Press, 1994.

[85] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[86] Marc Snir and William Gropp. *MPI: The Complete Reference (2-volume set)*. MIT Press, 2nd edition, September 1998.

[87] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, The MPI Core*, volume 1. MIT Press, second edition, August 1998.

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

[89] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 1(8):23–60, January 1998.

[90] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, pages 79–88. ACM, May 1996.

[91] Glasgow University. Glasgow parallel haskell. Web page http://www.cee.hw.ac.uk/ dsg/gph/, May 2004.

[92] Glasgow University. The glasgow haskell compiler. Web page http://www.haskell.org/ghc/, June 2006.

[93] Indiana University. Lam/mpi parallel computing. Web page http://www.lam-mpi.org/, March 2006.

[94] Ohio-State University. Mvapich mpi over infiniband project. Web page http://nowlab.cse.ohio-state.edu/projects/mpi-iba/, May 2006.

[95] Aad van der Steen and Jack Dongarra. Overview of recent supercomputers. Technical report, Univ. of Mannheim, Univ of Tennessee and NERSC/LBNL, 1996-2004.

[96] Paul Watters. *Solaris 10 : The Complete Reference*. McGraw-Hill, first edition, January 2005.

[97] the free encyclopedia Wikipedia. Parallel computing. Website URL: http://en.wikipedia.org/wiki/Parallel_computing, 2006.

[98] the free encyclopedia Wikipedia. Partition problem. Website URL: http://en.wikipedia.org/wiki/Partition_problem, 2006.

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

[100] I-Chen Wu and H. T. Kung. Communication complexity for parallel divide-and-conquer. In *Proceedings of the 32nd annual symposium on Foundations of computer science*. IEEE Computer Society Press, 1991.

[101] Songnian Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering.*, 14(9):1327–1341, September 1988.

Glosario

DLML Data List Management Library, biblioteca de funciones para el manejo de listas. Las cuales interna y transparentemente al programador, se manejan de forma paralela.

Embarazosamente adjetivo asignado a aquellas aplicaciones paralelas cuyos datos pueden ser procesados de manera independiente.

Granularidad referente al número de elementos en una lista. Si la lista tiene pocos elementos se habla de una granularidad gruesa; por el contrario si tiene muchos elementos se tiene una granularidad fina.

Independiente relacionado con aquellos datos los cuales pueden ser procesados en cualquier procesador, en cualquier orden y sin tomar en cuenta el procesamiento del resto de los datos.

Particionamiento proceso en el que los datos de una aplicación se separan o dividen con el objetivo de que se procesen en diferentes procesadores.

SADTs Shared Abstract Data Types, ambiente de programación paralela basado en el modelo de tipos de datos abstractos. Este ambiente encapula la comunicación y la sincronización dentro de los llamados a sus ADTs, principalmente en las colas.

Sincronización operación donde al menos dos procesadores operen al unísono o con exacta coincidencia en tiempo y frecuencia.