Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Departamento de Ingeniería Eléctrica
Sección de Computación

Difusión automática de datos bajo cómputo paralelo en clusters

Tesis que presenta

Santiago Domínguez Domínguez

para obtener el Grado de Doctor en Ciencias
en la Especialidad de Ingeniería Eléctrica
opción Computación

Director de la Tesis
Dr. Jorge Buenabad Chávez

México, D.F. Enero 2006

Agradecimientos

A mis Padres que me enseñaron a dar los primeros pasos en la vida, por enseñarme los buenos valores morales, por darme la oportunidad de realizar mis estudios, por su cariño y su ayuda siempre incondicional.

A mi esposa Nelly por su amor, su paciencia y su apoyo permanente para continuar preparándome académicamente. A quien he dejado durante mucho tiempo la mayor parte de las tareas de nuestro hogar.

A mis hijas Vero, Nelly y Karen las cuales siempre me han dado muchas alegrias y sus sonrisas han sido un motor que me fortaleció para continuar.

Agradezco al Dr. Jorge Buenabad por su paciencia, consejos y enseñanzas que han sido de un gran valor durante todos mis estudios de doctorado.

A mis hermanos Jorge y Fredy que siempre me han apoyado y alentado a continuar mis estudios.

A mi suegro Don José y mi cuñada Bica que me ayudaron económicamente en momentos muy dificiles.

Al CONACYT que me apoyó con su programa de becas de postgrado la cual fue escencial para realizar mis estudios de doctorado.

Al CINVESTAV que me dió los recursos académicos necesarios para obtener una mejor preparación y la oportunidad de terminar satisfactoriamente esta tesis.

A mis amigos Sais e Inox por darme sus palabras de aliento en los momentos dificiles del doctorado. A Héctor y en especial a Inox por darme sus observaciones y consejos para mejorar la escritura de esta tesis.

A Sofi que siempre me ayudó a realizar de manera más fácil los trámites de la documentación durante todo el doctorado.

A Dios y la Virgen que me han permitido vivir para cumplir mi sueño y el de toda mi familia.

A mis Padres
Por su cariño y ayuda incondicional

A mi esposa Nelly
Por su amor, su paciencia y su apoyo

A mis hijas: Verónica, Nelly y Karen
Por ser un motor que me dió la fuerza para continuar

Resumen

Hoy en día los *clusters* junto con las bibliotecas de paso de mensajes y los sistemas de memoria compartida distribuida han puesto al cómputo paralelo al alcance de todos. La programación con paso de mensajes actualmente es portable pero es complicada, excepto para simples patrones de acceso a datos. Además, es substancialmente diferente cuando la cantidad de datos a procesar excede la capacidad de la memoria: utilizando a ésta sólo como un caché. Un sistema de memoria compartida distribuida simplifica la programación, pero los sistemas actuales no pueden integrar arquitecturas heterogéneas, por ejemplo, de 32 y 64 bits.

Esta tesis presenta el diseño de una memoria compartida distribuida implementada toda en software, llamada DDS (del inglés: Data Diffusion Space). Su espacio de direcciones compartido es de 2^{64} bytes, en ambas arquitecturas de 32 y de 64 bits, y es extra al espacio de direcciones de cada proceso en cada nodo corriendo una aplicación. Los datos colocados en DDS son difundidos, o migrados y replicados, en forma automática en la memoria de los procesadores que los utilizan, sin importar si se encuentran en su disco o en la memoria o en el disco de otros procesadores. De esta manera, la programación de aplicaciones bajo DDS es la misma si la cantidad de datos excede o no la capacidad de la memoria. Su interfaz, para acceder *datos compartidos*, es similar a la de un sistema de archivos.

Esta tesis también presenta el diseño de un sistema de archivos paralelo, FSDDS, cuyos archivos de datos son mapeados sobre DDS, y por consiguiente sus datos son difundidos en las memorias de los nodos que los utilizan. La interfaz de FSDDS para acceder los datos es la misma que la de DDS. De esta manera, la programación con DDS y FSDDS es prácticamente la misma para aplicaciones cuyos datos caben en memoria, cuyos datos desbordan la memoria, o cuyos datos se leen/escriben en archivos. Y FS/DDS pueden integrar algunas plataformas heterogéneas.

FS/DDS manejan un caché en cada nodo y los accesos a datos son resueltos, de ser posible, con una de las copias en la caché de otro nodo, disminuyendo así el número de accesos a disco y, potencialmente, mejorando el desempeño de las aplicaciones. Esto se demuestra utilizando un modelo analítico y resultados experimentales. El desempeño de FS/DDS es en general mejor que el de MPI/PVFS; y su uso es más fácil.

Abstract

Today clusters along with messages passing libraries and distributed shared memory systems have made parallel computing ready to hand. Programming with message passing is now portable but is complicated, except for simple access patterns to data. Moreover, it is substantially different when the amount of data to process exceeds the capacity of the available memory: using this only as a cache. A distributed shared memory simplifies the programming task, but the current systems cannot easily integrate heterogenous architectures, for example, 32- and 64-bit architectures.

This thesis presents the design of a distributed shared memory implemented all in software, call DDS (Data Diffusion Space). Its shared addresses space is of size 2^{64} bytes, in both architectures of 32 and 64 bits, and is extra to the addresses space of each process in each node running an application. The data placed in DDS are diffused, or migrated and replicated, automatically in the memory of the accessing processors regardless of their location, their local disk or the memory or disk of other processors. Thus the programming of applications under DDS is the same whether or not the amount of data to process exceeds the memory capacity. Its interface to access *shared data* is similar to that of a file system.

This thesis also presents the design of a parallel file system, FSDDS, whose data files are mapped onto DDS, and therefore their data are diffused to the memories of the accessing nodes. The interface of FSDDS to access data is the same as that of DDS. This way, the programming with DDS and FSDDS are practically the same for applications whose data fit in memory, whose data overflow the memory, or whose data are read from or written to files. And FS/DDS can integrate some heterogenous platforms.

FS/DDS handle a cache in each node and data accesses are solved, if possible, with a copy in the cache of other nodes, thus decreasing the number of accesses to disk and, potentially, improving application performance. This is demonstrated using an analytical model and experimental results. The performance of FS/DDS is in general better than the that of MPI/PVFS; and its use is simpler.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Los *clusters*, o redes de área local (LANs), son cada vez más populares para ejecutar aplicaciones paralelas debido a su bajo costo y cada vez mejor desempeño, proporcionando un beneficio extraordinario en precio-rendimiento. Además, los *clusters* son adecuados para ejecutar ambos tipos de aplicaciones: *in-core*, aplicaciones que procesan datos almacenados todos en memoria; y *out-of-core* (también conocidas como de Entrada/Salida, E/S), las cuales procesan una cantidad de datos más grande que la memoria disponible, y por lo tanto son programadas para leer y escribir los datos en disco, utilizando la memoria como un caché solamente. En un *cluster* cada nodo incluye, en general, un disco duro, y por consiguiente la capacidad de almacenamiento en disco es proporcional al número de los nodos.

La programación de aplicaciones paralelas en *clusters* generalmente sigue una de dos alternativas. Una alternativa consiste en el uso de bibliotecas de paso de mensajes, tales como MPI [67, 70] y PVM [82]. En la actualidad, estas bibliotecas son las más utilizadas en la programación de aplicaciones paralelas en *clusters* principalmente por su portabilidad ya que existe una versión para la mayoría de las arquitecturas.

Sin embargo, la programación de aplicaciones paralelas basada en paso de mensajes es complicada excepto para simples patrones de acceso a datos, debido a que el programador debe especificar la comunicación mediante mensajes entre procesos, para distribuir los datos. El programador necesita tener en mente *dónde* está el dato, debe decidir *cuándo* un proceso necesita comunicarse, con *quién* comunicarse, y *qué* dato transmitir.

Además, con paso de mensajes, la programación de una aplicación *in-core* no es la misma que la programación de su versión *out-of-core*. Se requieren dos versiones, en la versión *in-core*, los procesadores intercambian datos con paso de mensajes. En la versión *out-of-core*, los procesadores leen y escriben los datos directamente a archivos en disco, si se utiliza un sistema de archivos paralelos. Un sistema de archivos paralelo particiona un archivo en bloques de datos y los distribuye en los discos de diferentes nodos, tal que los datos pueden ser accedidos en paralelo bajo múltiples operaciones de lectura/escritura concurrentes a un mismo archivo. Si no se utiliza un sistema de archivos paralelos, los datos se encuentran en archivos locales, y la programación involucra ambos el paso de mensajes y la lectura y escritura a disco.

Por otro lado, si se tienen accesos concurrentes a archivos locales o a archivos paralelos, el programador debe implementar un protocolo de coherencia "múltiples lectores un solo escritor", lo cual complica aún más la programación. El protocolo es implementado con operaciones de lectura y escritura directas a disco haciendo que el manejo de los archivos sea ineficiente en cuanto a desempeño.

Otra alternativa para desarrollar aplicaciones paralelas en *clusters* es el uso de un sistema de memoria compartida distribuida (DSM, por sus siglas en inglés). Una DSM es un área del espacio de memoria lógica de cada procesador en una arquitectura con memoria distribuída. Esta área es compartida por todos los procesos que ejecutan una aplicación paralela. Los datos en esta área son, por lo tanto, también compartidos entre los procesos. Una DSM mueve los datos compartidos automáticamente entre las memorias de los nodos de acuerdo a los patrones de acceso de la aplicación. El programador accede los datos sin utilizar mensajes ni necesita conocer la localización de los datos. Los datos son accedidos por medio de referencias a memoria haciendo transparente la comunicación, lo que facilita la programación de aplicaciones paralelas.

Existen sistemas de DSM basados en hardware, basados en software o una combinación de ambos. Los sistemas basados en hardware son implementaciones eficientes porque la comunicación es realizada por hardware, y el tamaño de la unidad de transferencia normalmente es de 16 a 128 bytes, siendo utilizados para aplicaciones con un paralelismo de grano fino. Sin embargo, estas DSMs son plataformas específicas propietarias con costos elevados. Además su diseño se hace más complicado conforme aumenta el número de procesadores debido a que el hardware no es fácil de escalar. Ejemplos de

estas DSMs son: arquitecturas de acceso no uniforme a memoria con coherecia en la caché (CC-NUMA, del inglés: cache-coherent non-uniform memory access), tales como DASH [59, 60] y Origin [58]; y arquitecturas de difusión de datos (también conocidas como arquitecturas de memoria cache, o COMAs, del inglés: cache only memory architectures), tales como DDM [107] y COMA-F [48]. En las CC-NUMAs, los datos se mueven a la caché de cada procesador que los está usando, ya sea que el dato es *local* (residente en el nodo de memoria principal más cercano a un procesador) o *remoto*. En las COMAs, la organización de la memoria principal es asociativa, y de esta manera los datos se mueven a los nodos de memoria principal, y desde estos dentro de las cachés del procesador.

Los sistemas de DSM combinados con software y con hardware también son implementaciones eficientes. El soporte de hardware es típicamente el utilizado por los sistemas de memoria virtual, y es usado para detectar la ausencia de los bloques en la memoria local de un nodo que los accede. Sin embargo, el hardware utilizado en estas DSMs limita su portabilidad. Algunas implementaciones de DSM tal como TreadMarks [55, 102] soportan el mapeo de archivos sobre el espacio de direcciones compartido, lo que facilita la programación de aplicaciones *out-of-core*. Sin embargo, estas implementaciones están limitadas por la capacidad de direccionamiento de la memoria de cada nodo. Aunque ya están disponibles procesadores de 64 bits, en la mayoría de las redes locales actuales se utilizan procesadores de 32 bits cuya capacidad total de direccionamiento es de 4 Giga-bytes (GBs). Este rango de direccionamiento no es suficiente para algunas aplicaciones *out-of-core* las cuales requieren hasta varios cientos de GBs [19]. Además, los sistemas de DSM actuales no trabajan en un medio ambiente heterogéneo con arquitecturas de 32 y 64 bits, lo que limita su portabilidad.

Las implementaciones desarrolladas sólo en software no requieren de soporte del hardware o del sistema operativo para transmitir y recibir los mensajes, lo que las hace portables. Ejemplos de estos sistemas son CRL [49] y Shasta [87], los cuales no soportan mapeo de archivos. Sin mapeo de archivos, la programación de aplicaciones *in-core* y *out-of-core* es diferente.

En base a lo anterior, el objetivo de esta tesis es el desarrollo de un sistema de difusión de datos que facilite la programación de aplicaciones paralelas *in-core* y *out-of-core*, que puedan ser portables en ambientes heterogéneos y que se ejecuten con un

buen desempeño.

La propuesta de esta tesis es desarrollar un ambiente de programación para cómputo paralelo en *clusters*, en donde el programador no se preocupe por la localización de los datos ni por la capacidad de almacenamiento en memoria y disco de los nodos. Para desarrollar aplicaciones *out-of-core* es necesario integrar al ambiente un sistema de archivos paralelos en donde los datos de los archivos puedan replicarse en la memoria de los nodos que los utilizan y el programador no necesite especificar un protocolo de coherencia en los accesos concurrentes, ni ser necesario especificar un patrón de distribución de los archivos de datos.

Esta tesis presenta el diseño de una memoria compartida distribuida toda en software, llamada espacio de difusión de datos, o DDS (del inglés: Data Diffusion Space) [16]. DDS es un espacio de direcciones adicional al espacio de direcciones virtuales de cada proceso ejecutando una aplicación paralela. DDS es una versión toda en software de COMA-F, extendida para manejar dos (en lugar de un) nivel de almacenamiento: memoria y disco. DDS es sólo para datos compartidos, los cuales el programador debe explícitamente especificar como tales a través de simplemente declararlos dentro de una estructura de datos del lenguaje C. Por medio de la estructura se define el espacio de memoria de DDS que será compartido en todos los nodos que ejecutan la aplicación. En cada nodo se reserva un área de su memoria y de su disco local para almacenar las copias de los bloques de DDS, y posiblemente puedan existir varias copias de un mismo bloque en las memorias de diferentes nodos. Los datos definidos en este espacio automáticamente se difunden, o migran y replican, en la memoria de los nodos que los utilizan de acuerdo a sus patrones de acceso bajo un protocolo de coherencia "múltiples lectores un solo escritor".

El tamaño de DDS puede ser hasta de 2^{64} bytes, en ambas arquitecturas de 32 y 64 bits. Por consiguiente, algunos de los datos compartidos pueden no estar en memoria, sino en el espacio del disco de los nodos de procesamiento. Sin embargo, el programador utiliza la misma interfaz para ganar acceso a un dato compartido (sin especificar ninguna localización del dato). Por lo tanto, la programación de aplicaciones *in-core* y *out-of-core* es la misma.

La interfaz de DDS es similar a la que se utiliza para acceder los datos en archivos. La interfaz es una biblioteca conteniendo llamadas a funciones con operaciones de lectura

y escritura que permiten ganar acceso a un bloque de datos y mapearlo dentro del espacio de los procesos antes de que sea accedido. La implementación actual de DDS sólo provee una interfaz para el lenguaje C, pero con pocos cambios se puede portar a otros lenguajes de programación. Además, esta interfaz permite que DDS presente un modelo de programación que es independiente del sistema. El modelo es simple y fácil de usar porque sólo utiliza las operaciones de lectura y escritura de datos.

Para una lectura, el programador primero llama a `DDS_Read()`; para una escritura el programador primero llama a `DDS_Write()`. Estas funciones utilizan como parámetro un identificador de la región de memoria en DDS (una variable o un arreglo definido en DDS) y un *offset* dentro de la región especificando el bloque deseado, y retornan un apuntador a la base del área del bloque de datos dentro de la región especificada. El programador entonces utiliza el dato de la misma manera a como se usa un dato en su espacio de dirección local. Después de usar el dato el programador debe llamar a `DDS_UnRead()` o `DDS_UnWrite()`, respectivamente, las cuales delegan el control de acceso del bloque y permiten que otro proceso gane su acceso.

En una lectura cada proceso obtiene una copia del área de datos solicitada, y en una escritura sólo un proceso obtiene una copia exclusiva de los datos, mientras los otros procesos esperan que el bloque de datos sea modificado por el proceso que obtuvo la copia exclusiva.

Esta tesis también presenta el diseño de un sistema de archivos distribuido paralelo sobre DDS llamado FSDDS (del inglés: File System atop the Data Diffusion Space [17]). El propósito de FSDDS es mejorar el desempeño y facilitar la programación de aplicaciones *out-of-core* que leen/escriben datos en archivos. La interfaz de FSDDS es similar a la interfaz de DDS utilizando las mismas funciones para las operaciones de lectura y escritura de DDS: `DDS_Read()`, `DDS_UnRead()`, `DDS_Write()` y `DDS_UnWrite()`.

Cuando un archivo es abierto bajo FSDDS, éste es automáticamente mapeado dentro del espacio de dirección compartido de DDS. El primer Tera byte de DDS es reservado para los datos compartidos de los arreglos (es decir, no pertenecen a un archivo de datos); los siguientes Tera bytes son usados uno para cada archivo que es abierto. De esta manera, los archivos y los datos compartidos son accesados en la misma forma, y bajo el mismo protocolo de coherencia.

En DDS y FSDDS los datos de una aplicación tienden a moverse desde las memo-

rias de los nodos, lo cual disminuye el número de operaciones de E/S a disco y por consiguiente mejora el desempeño de las aplicaciones *out-of-core*.

Comparado con el paso de mensajes, DDS es transparente en la programación de aplicaciones *in-core* y *out-of-core* porque permite una misma programación para ambos tipos de aplicaciones, y es más fácil de usar porque no se especifica la localización de los datos. Además, la capacidad de direccionamiento es de hasta 2^{64} bytes.

Comparado con los sistemas DSM, DDS ofrece portabilidad e integra plataformas heterogéneas porque es implementada toda en software; maneja un espacio de 64 bits para arquitecturas de 32 y 64 bits; y presenta una misma programación para aplicaciones *in-core* y *out-of-core*. Sin embargo, la programación bajo DDS es un poco más complicada.

Otras características de DDS son su portabilidad y su flexibilidad ya que el sistema fue diseñado utilizando las primitivas de comunicación del sistema operativo. Por lo que modificaciones al *kernel* no son necesarios, debido a que las implementaciones modernas de UNIX y WINDOWS proveen todas las funciones requeridas para el manejo de las comunicaciones y de la memoria. Para implementar las comunicaciones se utiliza a los *sockets* ya que son soportados en la mayoría de las plataformas y operan sobre el protocolo TCP/IP.

Un modelo analítico ha sido desarrollado para evaluar el desempeño de DDS y FSDDS. El modelo determina el desempeño de aplicaciones paralelas *out-of-core* considerando los siguientes factores: el tamaño del espacio de difusión, la cantidad de datos de la aplicación y el número de procesadores utilizados para ejecutar la aplicación. Con este modelo también es posible realizar una estimación del número de operaciones de E/S a disco generadas por la aplicación considerando los factores anteriores.

Además, con el fin de evaluar experimentalmente el desempeño de aplicaciones paralelas usando DDS y FSDDS, hemos ejecutado varias aplicaciones en un *cluster* utilizando un número diferente de procesadores y diferentes tamaños del problema. Se ejecutó una aplicación a la vez utilizando toda la capacidad de almacenamiento en memoria y disco de cada nodo. Las aplicaciones utilizadas son programas de evaluación típicos usados en muchos sistemas de DSM. Tienen patrones de acceso regulares, predecibles y son de E/S intensiva.

Los resultados obtenidos al ejecutar las aplicaciones utilizando DDS y FSDDS, nos muestran que se disminuye el número de operaciones de E/S a disco y consecuentemente

el tiempo de ejecución de la aplicación. Las operaciones de E/S son disminuidas porque se tiene una caché en cada nodo y los accesos a datos son resueltos, de ser posible, con las copias desde las cachés de otros nodos.

Otro aspecto a considerar es el tiempo de sincronización cuando se tienen operaciones colectivas al compartir datos entre procesos. Al utilizar una biblioteca de paso de mensajes, el tiempo de operación de una sincronización colectiva tiende a crecer con el número de procesos. En DDS no es necesario que los procesos se comuniquen en forma colectiva para acceder los datos almacenados en la memoria o incluso en el disco gracias a su protocolo de coherencia. De esta manera, bajo DDS la granularidad de la sincronización es menor y por tanto se disminuye el tiempo de acceso a los datos.

Aunque los dos aspectos anteriores dependen del patrón de acceso de la aplicación, creemos que es posible asumir que el tiempo de ejecución al emplear DDS será al menos similar o mejor, al tiempo bajo una biblioteca de paso de mensajes como MPI. Pero con DDS la programación es más sencilla. Lo que permite escribir programas en forma más fácil y rápida.

Resumen y organización de la tesis

Este capítulo presentó el alcance de nuestra investigación: el diseño y la implementación de una memoria compartida distribuida toda en software, a la cual llamamos DDS (del ingés: Data Diffusion Space). Su diseño fué motivado para facilitar el desarrollo de aplicaciones paralelas *in-core* y *out-of-core*, ejecutarlas en un medio ambiente heterogéneo con arquitecturas de 32 y 64 bits y mejorar su desempeño.

Otro aspecto de este sistema es que integra un sistema de archivos paralelo junto con la memoria compartida distribuida llamado FSDDS (del inglés: File System atop the Data Diffusion Space). En FSDDS los archivos son vistos como arreglos de memoria y pueden ser accedidos como si estuvieran almacenados en la memoria local de los nodos de un *cluster*.

El resto de la tesis está organizada como sigue:

El Capítulo 2 describe el cómputo paralelo en *clusters*, los modelos de programación utilizados en *clusters*, y las ventajas y desventajas de estos modelos.

El Capítulo 3 ofrece una descripción de los sistemas de archivos paralelos y sus

aspectos de diseño, principalmente a los sistemas de archivos para *clusters*. Lo cual nos sirve para introducir los conceptos utilizados en los siguientes capítulos.

El Capítulo 4 describe el diseño de nuestro espacio de difusión de datos, su arquitectura, su operación y el modelo de programación utilizado para desarrollar aplicaciones paralelas.

El Capítulo 5 muestra el diseño de nuestro sistema de archivos paralelo. La arquitectura y operacion del sistema de archivos son explicados en detalle. Se describe el particionamiento y mapeo de un archivo dentro de DDS y su interfaz de programación.

El Capítulo 6 presenta un modelo de análisis de rendimiento de aplicaciones *out-of-core*. Discutimos los costos de las operaciones de acceso a datos y presentamos nuestra metodología experimental para evaluar el rendimiento de aplicaciones ejecutadas utilizando DDS y FSDDS.

El Capítulo 7 expone nuestras conclusiones, las aportaciones de la investigación realizada. Además se listan las limitaciones del trabajo desarrollado y el trabajo futuro.

Capítulo 2

Cómputo paralelo en clusters

El cómputo paralelo es utilizado para mejorar el tiempo de ejecución de muchas aplicaciones. En este capítulo presentamos una breve reseña del cómputo paralelo y mostramos cómo ha evolucionado hacia los *clusters* como una excelente alternativa para ejecutar aplicaciones paralelas.

2.1. Introducción

El cómputo paralelo es la ejecución simultánea de la misma tarea en varios procesadores con el fin de reducir el tiempo de ejecución de una aplicación [38]. Para disminuir el tiempo de ejecución de la aplicación el código y/o los datos de la aplicación son divididos y distribuidos entre los procesadores. En cada procesador se ejecuta una o más tareas las cuales realizan una parte del cómputo de la aplicación. Los procesadores son interconectados por un bus o una red de comunicación que permite a las tareas compartir sus datos y/o sus resultados.

El cómputo paralelo ha sido ampliamente utilizado para mejorar el desempeño de aplicaciones que demandan gran cantidad de cómputo [40]. Algunas de éstas aplicaciones en general se caracterizan porque realizan gran cantidad de operaciones de cálculo en la solución de problemas de la ciencia e ingeniería [4, 19], como por ejemplo: diseño y procesamiento de materiales virtuales, modelado del clima y simulación de eventos discretos, entre otras. Otras aplicaciones ejecutan gran cantidad de operaciones que comparan y modifican patrones de información, como la consulta de bases de datos, el

reconocimiento visual y la búsqueda de yacimientos petrolíferos.

Para reducir el tiempo de ejecución de estas aplicaciones, en los años 60s y 70s del siglo pasado se diseñaron y construyeron diversas computadoras paralelas que consistían de una sola unidad de control y varios elementos de procesamiento interconectados entre sí. En estas computadoras, cada procesador ejecutaba de manera simultánea la misma instrucción sobre sus datos locales. Estas computadoras se conocen como máquinas SIMD (del inglés: Single Instruction Multiple Data) y son del tipo: procesadores de vectores o procesadores de arreglos.

La principal ventaja de las máquinas SIMD fué su facilidad de programación. El paralelismo es realizado de manera transparente al programador. Por ejemplo, en las aplicaciones numéricas normalmente se ejecutan operaciones aritméticas (como sumas, restas, multiplicaciones y divisiones) repetidamente sobre un conjunto grande de datos y el compilador puede fácilmente generar las instrucciones necesarias para cargar los datos en un vector o arreglo y ejecutar una sola instrucción sobre todos los datos en el vector. La principal desventaja de las máquinas SIMD fué el costo excesivo del diseño y la construcción de su hardware de propósito específico.

En los 80s y principios de los 90s surgieron un gran número de propuestas de arquitecturas paralelas basadas en procesadores de propósito general y por consiguiente de bajo costo, interconectando cientos de estos procesadores En estos sistemas, cada procesador tiene su propia unidad de control y es independiente de los demás procesadores, lo que hace necesario dividir y distribuir la carga de trabajo entre los procesadores. El mismo o un diferente conjunto de intrucciones es cargado en cada procesador, con un conjunto distinto de datos. Sin embargo, para compartir y distribuir los datos y el código entre los procesadores debe especificarse la comunicación y coordinación entre sus tareas.

Estas arquitecturas son conocidas como máquinas MIMD (del inglés: *Multiple Instruction Multiple Data*) y se clasifican en dos tipos: *multiprocesadores* y *multicomputadoras* [4, 45]. En los multiprocesadores todos los procesadores comparten una sola memoria principal por medio de una interconexión o *bus*. Los procesadores se comunican accediendo a un área de memoria compartida; un procesador escribe datos dentro de una localización en esa memoria y otro procesador lee los datos. Con este tipo de comunicación no es necesario especificar un nodo fuente ni un destino, lo que hace fácil la

programación de aplicaciones paralelas. La programación de aplicaciones paralelas en estos sistemas normalmente se realiza con hilos concurrentes (aunque también pueden usarse procesos), los cuales son ejecutados en forma simultánea en cada procesador. El acceso a esta memoria puede ser controlada usando técnicas desarrolladas desde computadoras multi-tareas, como semáforos, candados o monitores. Sin embargo, los multiprocesadores no son escalables arriba de unas decenas de procesadores, debido a que el medio de interconexión común a la memoria conduce a un cuello de botella en configuraciones grandes y su costo es elevado debido a que el hardware se hace más complejo conforme aumenta el número de procesadores.

Las multicomputadoras son arquitecturas de memoria distribuida donde cada procesador tiene su propia memoria y la comunicación entre procesadores es realizada por medio de mensajes. Estas arquitecturas son escalables debido a que no hay contención en el acceso a la memoria por lo que el ancho de banda de la memoria crece con el número de procesadores. Sin embargo, para compartir los datos el programador debe especificar los nodos fuente y destino en cada mensaje. Además, la coordinación entre tareas se hace más difícil conforme crece el número de procesadores [45].

Durante los 90s se diseñaron arquitecturas que incluían los beneficios de los multiprocesadores y las multicomputadoras, principalmente la facilidad en la programación y la escalabiliad. Diseñándose sistemas multiprocesadores escalables o multiprocesadores de memoria compartida distribuida, los cuales soportan un espacio de direcciones compartido sobre una memoria distribuida [2, 8, 14, 58, 59, 65, 77, 107]. En estas arquitecturas, cada procesador es de propósito general, tiene su propia memoria local y se conecta a la memoria de otros nodos a través de un hardware especial. Este hardware de interconexión es el encargado de obtener una copia de los datos desde un nodo remoto y llevarlos a quien los utilice. Sin embargo, la construcción de estas arquitecturas resultó más difícil de lo esperado al aumentar el número de procesadores y pocos modelos fueron construidos sin ganar la aceptación general esperada debido a su alto costo.

Desde mediados de los 80s también se investigó el uso de las redes locales de computadoras personales o de estaciones de trabajo para explotar el paralelismo debido a su bajo costo y gran demanda. Estas redes fueron inicialmente concebidas para compartir recursos y facilitar la comunicación entre un gran número de procesadores. En la actualidad, la amplia disponibilidad de procesadores cada vez más potentes, las interconex-

iones rápidas (como Fast Ethernet, Gigabit Ethernet, Infiniband y Myrinet), el desarrollo de las bibliotecas portables para el paso de mensajes tales como MPI [13, 67, 70] y PVM [82, 96] y los sistemas de memoria compartida distribuida (como TreadMarks [102], Rthreads [34, 33], OpenMP [76], CRL [49] y otros más) han hecho de las redes locales, o *clusters*, la alternativa más económica y, consecuentemente, de aceptación más generalizada para ejecutar aplicaciones paralelas.

Un *cluster* consiste de un conjunto de nodos independientes conectados entre sí [18]. Cada nodo del *cluster* puede ser una computadora con un solo procesador o un sistema multiprocesador, ambos con su propia memoria, con facilidades de E/S y su propio sistema operativo ejecutándose en el nodo. Los nodos pueden existir en un solo gabinete o estar físicamente separados. En ambos casos son conectados por medio de una red de comunicación (tal como una Gigabit Ethernet y Myrinet), similar a una LAN. Junto al avance tecnológico y a la disminución en el precio de los dispositivos que los conforman, los *clusters* o redes de computadoras personales presentan un beneficio extraordinario en costo-rendimiento en la ejecución de aplicaciones distribuidas o paralelas con cómputo intensivo. Así también pueden utilizarse para ejecutar aplicaciones secuenciales. Un ejemplo de tal sistema es un *cluster* Beowulf[12]. La arquitectura típica de un *cluster* es mostrada en la Figura 2.1.

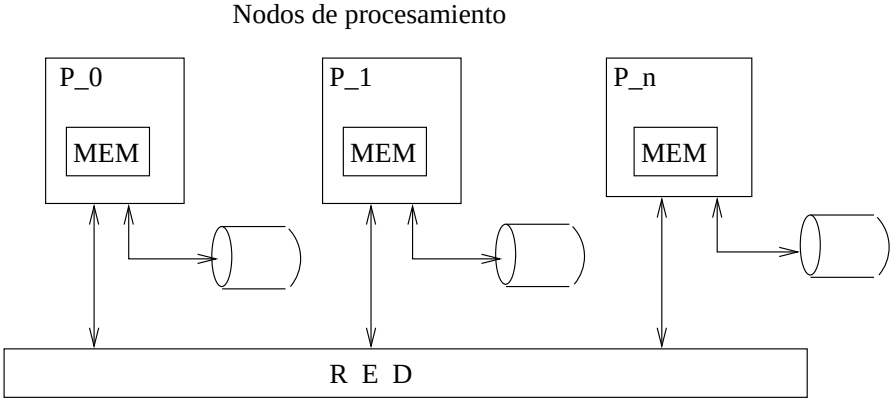

Figura 2.1: Arquitectura de un *cluster*.

En el resto del capítulo se muestran los requerimientos de hardware y software de un *cluster*, el modelo de programación utilizado para desarrollar aplicaciones paralelas en *clusters* y se presentan las ventajas y desventajas de las bibliotecas que son utilizadas

para implementar y ejecutar estas aplicaciones.

2.2. Configuración de un cluster

Actualmente las redes de area locales (llamadas LAN) son necesarias para compartir los recursos y facilitar la comunicación entre las estaciones de trabajo o PCs (del inglés: Personal Computers) dentro de una organización. Una LAN es una red que abarca una área relativamente pequeña. Estas se encuentran comúnmente dentro de un edificio o un conjunto de edificios que estén contiguos. Así mismo, una LAN puede estar conectada con otras LANs a cualquier distancia por medio del Internet y compartir sus recursos entre ellas. Por su gran demanda son cada vez más rápidas y más baratas, lo cual ha permitido que sean una excelente alternativa para ejecutar aplicaciones paralelas. Un *cluster* es similar a una LAN porque consiste de una colección de computadoras o nodos independientes conectados conjuntamente y en nuestro caso ambos son lo mismo.

Las LANs, en general, son fáciles de configurar y permiten realizar operaciones colectivas con un grupo específico de nodos. Una LAN se configura mediante hardware y software. El hardware establece el medio físico de las interconexiones entre las PCs dentro de la LAN y el software permite que los nodos puedan comunicarse de manera individual o colectiva entre ellos.

2.2.1. Hardware

Los nodos de una LAN son computadoras que tienen uno o varios procesadores, tales como PCs o estaciones de trabajo. Una PC es una computadora de propósito general para realizar trabajos de oficina, casa o ingeniería, que cuenta con al menos un procesador, memoria, un disco duro, e interfaces hacia el usuario como un monitor, teclado, etc. Por otro lado, actualmente las estaciones de trabajo son computadoras con características similares a una PC con diferencias principalmente en la capacidad de procesamiento, como son: la velocidad del procesador, el tamaño de la memoria, el tamaño y la velocidad del disco, entre los más importantes. (Aunque en el pasado fueron construidas con un hardware específico.)

Los nodos en una LAN son interconectados por un medio de transmisión para comunicarse. Las redes LAN tradicionales operan con medios de transmisión tales como

cable de par trenzado (del inglés: Unshielded Twisted Pair), cables coaxial (ya casi obsoleto porque presenta muchos problemas), fibra óptica (inmune a la mayoría de interferencias), portadoras de rayo infrarojo o láser, radio y microondas en frecuencias no comerciales. Las velocidades en las redes de área local van desde 10 Megabits por segundo (Mbps) hasta varios Gigabits por segundo (Gbps).

Los nodos normalmente están interconectados por medio de un dispositivo de red como puede ser: un concentrador, un switch, un puente, o un repetidor. A la forma en que están conectados cada uno de los nodos se le llama topología. Estos dispositivos de red permiten realizar diferentes topologías de una LAN para interconectar los nodos entre sí. Una red tiene dos diferentes topologías: una física y una lógica. La topología física es la disposición física actual de la red, la manera en que los nodos están conectados unos con otros. La topología lógica es el método que se usa para comunicarse con los demás nodos, la ruta que toman los datos de la red entre los diferentes nodos de la misma. Las topologías física y lógica pueden ser iguales o diferentes. Las topologías más comunes son: un bus, un anillo, una estrella y un árbol (ver Figura 2.2).

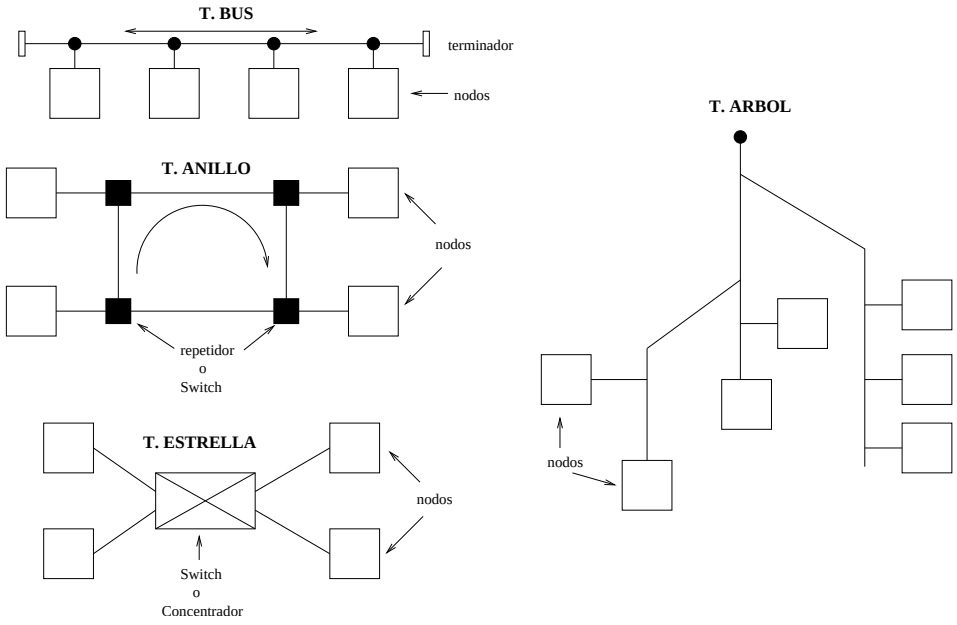

Figura 2.2: Topologías de una LAN.

Una topología bus es una arquitectura LAN lineal en la que los nodos son conectados a través de un mismo canal de comunicación en la cual las transmisiones de un nodo

se propagan a través del medio y son recibidas por todos los otros nodos. Consiste en un cable con un terminador en cada extremo. Todos los nodos de la red están unidos a este cable: el cual recibe el nombre de *Backbone Cable*. El bus es pasivo, no se produce regeneración de las señales en cada nodo. Los nodos en una red tipo "bus"generalmente transmiten la información y esperan que ésta no vaya a chocar con otra información transmitida por otro de los nodos. Si esto ocurre, cada nodo espera una pequeña cantidad de tiempo al azar, después intenta retransmitir la información. El tiempo de transmisión depende del ancho de banda del canal, pero conforme el número de nodos crece la contención del canal aumenta lo cual a su vez aumenta el tiempo de comunicación y puede degradar el rendimiento de las aplicaciones.

Una topología anillo es una arquitectura LAN que consiste de dispositivos conectados uno a otro por un enlace de transmisión unidireccional para formar un solo lazo cerrado. La información fluye a través de los nodos en un solo sentido. Cada nodo examina la información que es enviada a través del anillo. Si la información no está dirigida al nodo que la examina, la pasa al siguiente en el anillo. La desventaja del anillo es que el tiempo de transmisión depende del número de nodos y si se rompe una conexión, la comunicación mo es posible.

Una topología estrella es una arquitectura LAN en la cual los nodos sobre una red son conectados a un elemento central común, un concentrador o un switch, por medio de enlaces dedicados. Los datos emitidos por un nodo en estas redes fluyen a través del elemento central hacia su destino. El concentrador o switch realiza todas las funciones de la red y actúa como amplificador de los datos para que la señal pueda viajar a mayores distancias. Este esquema tiene una ventaja al tener un panel de control (un concentrador o switch) que monitorea el tráfico, evita las colisiones y la interrupción de una conexión no afecta al resto de la red. El tiempo de comunicación depende del ancho de banda del canal de comunicación y del tiempo de respuesta del concentrador. Su desventaja es que si falla el elemento central falla toda la red.

Una topología árbol es una arquitectura LAN que es similar a la topología bus o a una estrella sin un elemento central, en donde cada rama puede tener múltiples nodos. Las ramas son formadas por otras estrellas utilizando concentradores o switches como elementos de interconexión. Algunas de estas estrellas tienen más prioridad que otras y así es posible encausar la información a través de diferentes estrellas.

Estas topologías son ampliamente utilizadas por Ethernet y utilizándolas conjuntamente pueden hacerse diversas configuraciones. En el resto de la tesis usaremos el término *cluster* o LAN de manera indistinta para referirnos a una red de computadoras.

2.2.2. Software

Los nodos de un *cluster* pueden ser heterogéneos al tener diferentes sistemas operativos y un distinto hardware y en general soportan ambientes de multiprogramación. Normalmente ejecutan un sistema operativo multitareas (como Linux o Windows) que permite ejecutar de manera concurrente los procesos de una aplicación paralela.

Los ambientes de programación paralela utilizados en los *clusters* son herramientas portables, eficientes y de fácil uso para el desarrollo de aplicaciones paralelas. Estas herramientas eliminan el problema de la heterogeneidad de los nodos, facilitan la ejecución de los procesos de las aplicaciones y ofrecen un medio de comunicación de datos rápido y confiable entre los nodos del *cluster* y el mundo exterior.

Los datos compartidos entre los nodos en una LAN o *cluster* se pueden transmitir de tres formas: *unicast, multicast y broadcast*. En una transmisión *unicast*, un solo paquete es transmitido desde el nodo fuente al nodo destino sobre la red. Una transmisión *multicast* consiste de un solo paquete de datos que es copiado y transmitido a un subconjunto específico de nodos sobre la red. Una transmisión *broadcast* consiste de un solo paquete de datos que es copiado y transmitido a todos los nodos sobre la red.

Con un uso adecuado de los tipos de comunicación de datos entre los nodos de un *cluster*, los nodos del *cluster* pueden operar como computadoras individuales o pueden trabajar de manera colectiva como una sola máquina integrada y aparecer como un solo sistema a los usuarios y a las aplicaciones.

La portabilidad en un *cluster* se facilita empleando *sockets* en conjunto con el protocolo TCP/IP utilizado en las redes LAN. Un *socket* es una forma de comunicarse con otros programas usando descriptores de archivo estándar de Unix. En este caso, el archivo es una conexión de red. Los *sockets* son soportados por la mayoría de los sistemas operativos para realizar las comunicaciones entre los procesos de diferentes nodos. Normalmente presentan un conjunto de funciones que permiten su configuración y su uso. Además, las comunicaciones con TCP/IP presentan buenos rendimientos en la transmisión de datos, como ha sido mostrado experimentalmente por Nog y Kotz [75].

Existen dos tipos de *sockets*: los *sockets* de UNIX utilizados para la comunicación entre procesos locales de un mismo nodo y los *sockets* de Internet utilizados para la comunicación entre procesos de diferentes nodos. Hay dos tipos de sockets de internet: los *sockets* de flujo y los *sockets* de datagramas. Los *sockets* de flujo (del inglés: Stream sockets) definen flujos de comunicación en dos direcciones, fiables y con conexión. Los *sockets* de datagramas son *sockets* sin conexión no confiables. Cuando un datagrama es enviado no se garantiza que los mensajes lleguen en secuencia ni que lleguen a su destino. Aunque, si llegan a su destino los datos que contiene el paquete puede presentar errores.

2.3. Modelo SPMD: Single Program Multiple Data

La ejecución de un programa de una aplicación en un nodo puede hacerse en forma fácil al dar el nombre del programa en la línea de comandos del sistema operativo. Si ejecutamos la misma aplicación en varios nodos y utilizamos diferentes programas, la implementación, el arranque y ejecución de cada programa se hace más complicado conforme el número de nodos crece. Sin embargo, si ejecutamos una copia del mismo programa en cada nodo la tarea se simplifica.

Por otro lado, si deseamos construir un programa portable y escalable es recomendable escribir un solo programa, el cual decide en ejecución lo que realmente va a hacer. En lugar de estructurar nuestra aplicación como programas separados, escribimos una sola pieza del código fuente el cual en ejecución decide qué acción tomar dependiendo del identificador del procesador en donde se está ejecutando el programa.

El uso de *clusters* bajo cómputo paralelo es generalmente bajo el modelo SPMD (del inglés: Single Program Multiple Data), el cual es el más comúnmente utilizado para desarrollar aplicaciones paralelas en *clusters*. En este modelo el mismo código del programa de la aplicación es ejecutado en todos los procesadores. Los datos de la aplicación son divididos y distribuidos en los procesadores donde fue ejecutado el programa. Así, cada procesador ejecuta básicamente la misma pieza de código pero sobre una parte diferente de los datos. El particionamiento de datos en este modelo también es conocido como paralelismo geométrico (debido a que los datos generalmente son divididos en bloques del mismo tamaño), descomposición en el dominio, o paralelismo en datos. La Figura 2.3 muestra una representación esquemática de este paradigma.

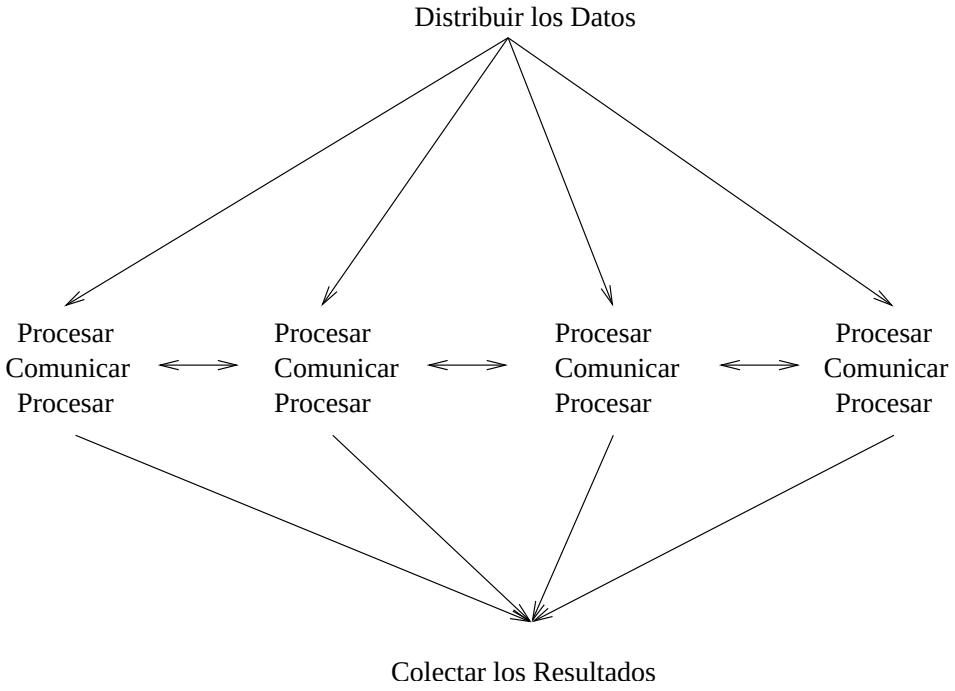

Figura 2.3: Estructura básica de un programa SPMD.

Los procesos de una aplicación paralela normalmente requieren compartir los datos entre ellos, por lo que deben comunicarse y sincronizarse para accederlos. Para realizar la comunicación entre los procesos de un *cluster* se utilizan dos clases de ambientes de programación: el paso de mensajes y la memoria compartida distribuida. Estos ambientes de programación están basados en bibliotecas que se ejecutan en el espacio de dirección del proceso. Así también, al utilizar las bibliotecas no se requiere del entendimiento del sistema operativo o de funciones especiales para el manejo de los recursos como la memoria o el disco. Cada biblioteca provee un conjunto de funciones o primitivas que permiten al programador crear los programas de una aplicación paralela.

2.4. Programación con paso de mensajes

Actualmente, el modelo de programación predominante para el cómputo paralelo en *clusters* con memoria distribuida es el paso de mensajes. Como se recordará, en un *cluster* los nodos no comparten la memoria física, cada nodo tiene su propia memoria

local y ningún nodo puede leer o escribir directamente a la memoria de otro nodo; no hay una memoria direccionable globalmente y accesible por todos los procesos que ejecutan la aplicación.

Al ejecutar una aplicación paralela normalmente es necesario compartir los datos entre los procesos. Para compartir los datos, los procesos pueden comunicarse transmitiendo mensajes a través de la red unos a otros. Los mensajes llevan la información requerida por los procesos y son implementados por el programador considerando varios aspectos que serán detallados en la sección 2.4.1.

2.4.1. Aspectos de diseño

En el modelo de programación con paso de mensajes, para transmitir un mensaje la siguiente información debe ser especificada:

- *qué* procesador está transmitiendo el mensaje.
- la *localización* del dato en el procesador transmisor.
- qué *clase* de dato se está transmitiendo. *Cuántos* datos están ahí.
- *cuál(es)* procesador(es) está(n) recibiendo el mensaje.
- en *dónde* sería dejado el dato sobre el procesador receptor.
- *cuántos* datos está preparado para aceptar el procesador receptor.

En general, como se recordará, el programador debe considear varios aspectos: necesita saber en *dónde* está el dato, debe decidir *cuándo* deben comunicarse los procesos, con *quién* comunicarse y *qué* dato transmitir. La utilización de estas bibliotecas no es una tarea fácil y tiende a complicarse aún más por la presencia de uno o varios de los siguientes elementos: la utilización de estructuras de datos complejas, un número grande de nodos o cuando el tamaño del problema crece.

Al transmitir un mensaje entre dos procesos, el proceso transmisor y el proceso receptor cooperarán en proporcionar la información para que pueda ser enviado y recibido el mensaje de manera confiable. En la Figura 2.4 se tienen dos procesos cada uno con su propio espacio de almacenamiento de datos x y y. El proceso 0 envía un mensaje

al proceso 1 de una LONGITUD dada a través de la red de comunicación del sistema. Ambos procesos identifican al proceso con el que intercambian los datos (proceso origen y destino), determinan el tipo de datos del mensaje y utilizan una etiqueta que permite identificar el mensaje que se transmiten. Aunque en algunos casos también existe comunicación en la que el receptor no especifica un origen particular.

```
char x[LONGITUD];                                    char y[LONGITUD];
...                                                  ...
SEND(destino, &x, LONGITUD, tipodato, etiqueta);     RECV(origen, &y, LONGITUD, tipodato, etiqueta);
...                                                  ...
          (a)Proceso 0                                         (b) Proceso 1
```

Figura 2.4: Comunicación con paso de mensajes.

La forma más simple de envío de un mensaje es una comunicación punto a punto en la cual un mensaje se envía desde un proceso emisor a un proceso receptor (procesos 0 y 1 en Figura 2.4). Solamente estos dos procesos necesitan conocer cualquier cosa sobre el mensaje. La comunicación en sí misma está compuesta de dos operaciones: una transmisión y una recepción.

La operación de transmisión puede ser síncrona o asíncrona. La transmisión síncrona (envío bloqueante) termina solamente cuando el mensaje correspondiente está siendo recibido por un proceso receptor. El emisor y el receptor se sincronizan para intercambiar cada mensaje. La transmisión asíncrona (envío no bloqueante) termina tan pronto como el mensaje correspondiente se haya entregado al sistema de comunicación. El emisor sigue ejecutándose sin esperar a que la comunicación se complete.

Muchos sistemas de paso de mensajes también proporcionan operaciones colectivas que permiten que más de dos procesos se comuniquen entre sí. Las operaciones colectivas para comunicación entre procesos se pueden construir a partir de las comunicaciones punto a punto. Una transferencia colectiva involucra a más de un emisor o a más de un receptor. De esta manera, un mensaje puede ser transmitido desde un solo proceso a varios procesos, o un proceso puede colectar un conjunto de datos al recibir mensajes desde varios procesos para formar uno solo. A partir de las anteriores operaciones con mensajes pueden hacerse combinaciones para generar otras operaciones colectivas. Una característica de las operaciones colectivas es que todos los procesos deben sincronizarse para llevar a cabo la operación, formando una barrera entre los procesos que intervienen en la operación colectiva.

La implementación de un programa paralelo utilizando directamente a los sockets para transmitir los mensajes entre los procesos no es muy adecuada debido a que los sockets son de un nivel de abstracción muy bajo. El programador tiene que realizar un gran esfuerzo para especificar cada parámetro y definir el orden de cada uno en el cuerpo del mensaje, con el fin de que sean accesibles por el emisor y el receptor del mensaje. Para eliminar este inconveniente y facilitar la implementación de programas paralelos se han desarrollado varias interfaces de paso de mensajes que utilizan a los sockets.

Actualmente, las interfaces de paso de mensajes tales como PVM[82] y MPI[67], son las más utilizadas en la programación de aplicaciones paralelas en arquitecturas de memoria distribuida. Estas interfaces hacen posible escribir programas paralelos que son portables a una amplia variedad de plataformas sin sacrificar el rendimiento.

2.4.2. PVM

PVM (del inglés: Paralell Virtual Machine) fue el primer ambiente de paso de mensajes ampliamente aceptado que proporcionó portabilidad e interoperabilidad a través de plataformas heterogéneas [82]. PVM permite que una red de computadoras heterogéneas sea usada como un solo recurso computacional llamada la máquina virtual paralela.

El ambiente de PVM consiste de tres partes: un demonio que reside sobre todas las computadoras en la máquina virtual paralela, una biblioteca de funciones de interfaces de PVM y una consola de PVM para interactivamente iniciar, examinar y modificar la máquina virtual. Antes de ejecutar una aplicación PVM, el usuario necesita iniciar un demonio PVM sobre cada máquina (computadora), de esta manera se creará la máquina virtual paralela. La aplicación PVM necesita ser ligada con la biblioteca de PVM, la cual contiene funciones de comunicación de punto a punto, comunicación colectiva, creación dinámica de tareas, coordinación de tareas y modificación de la máquina virtual, entre otras. Esta aplicación puede ser iniciada desde cualquiera de las computadoras en la máquina virtual en la línea de comandos del *shell* o desde la consola de PVM.

La mayor ventaja de PVM es su portabilidad en ambientes heterogéneos. Un programa de PVM puede ejecutarse sobre cualquier plataforma sobre la cual es soportado, además diferentes tareas del mismo programa pueden ejecutarse sobre cualquier plataforma al mismo tiempo como parte del mismo programa.

La principal desventaja de PVM es que su rendimiento no es tan bueno como otros

sistemas de paso de mensajes como MPI. Esto es debido principalmente porque PVM sacrifica rendimiento por flexibilidad al permitir heterogeneidad en la arquitectura de las computadoras.

2.4.3. MPI

MPI (del inglés: Message Passing Interface) es un estándar para paso de mensajes que ha sido desarrollado por un comité integrado por representantes de laboratorios de investigación, la universidad y la industria. La primer versión MPI-1 [67] fué establecida en 1994 y la segunda versión MPI-2 [70] fué desarrollada en 1997. MPI es un paradigma de paso de mensajes explícitos donde las tareas se comunican entre sí transmitiendo mensajes. Los tres principales objetivos de MPI son la escalabilidad, la portabilidad y el alto rendimiento.

MPI consiste de una biblioteca que provee un conjunto de funciones para especificar el paso de mensajes entre procesos. MPI define el concepto de comunicadores el cual combina contextos de mensajes y grupos de tareas para proporcionar mensajes seguros. Los *Intra-comunicadores* permiten el paso de mensajes seguros dentro de un grupo de tareas y los *inter-comunicadores* permiten el paso de mensajes seguros entre dos grupos de tareas.

MPI provee diferentes primitivas de comunicación de punto a punto, bloqueantes y no bloqueantes y tiene un soporte para *buffers* estructurados y tipos de datos derivados. MPI también provee muchos tipos diferentes de rutinas de comunicación colectiva entre tareas pertenecientes a un grupo. Otras funciones incluyen manejo de topologías de procesos (un medio para ordenar los procesos dentro de topologías), funciones de chequeo y control de estado del medio ambiente. MPI-2 también incluye la creación dinámica de tareas en MPI.

El estándar MPI no tiene una especificación para asignar las tareas a los CPUs. Tampoco el estándard especifica como las distintas implementaciones de MPI pueden comunicarse (como por ejemplo, LAM [56] y MPICH [68]). Esto obstaculiza el uso de MPI en una red consistiendo de máquinas con diferentes implementaciones de MPI.

2.4.4. Ejemplo con MPI

En esta sección mostramos dos ejemplos simples con paso de mensajes bajo MPI. MPI ha sido elegido en lugar de PVM en la implementación de estos ejemplos debido a que es más fácil especificar los datos que se transmiten entre procesos y presenta mejor rendimiento en la ejecución de aplicaciones paralelas.

La Figura 2.5 muestra un ejemplo simple de paso de mensajes bajo MPI. Antes de transmitir un mensaje, los procesos se sincronizan utilizando una barrera con `MPI_Barrier`. El proceso 0 transmite un mensaje y el proceso 1 lo recibe. En este ejemplo, en la función de transmisión `MPI_Send` y la función de recepción `MPI_Recv`, el programador especifica la dirección del mensaje (*buffer*), el tamaño del mensaje (*lenbuf*), el tipo o tamaño de los datos que se transfieren (`MPI_CHAR`), la identificación de los procesos que intervienen en la transmisión (*destino y origen*) y una etiqueta del mensaje (*type*) es especificada para identificar a cada mensaje. El parámetro `MPI_COMM_WORLD` especifica un comunicador el cual es un conjunto de procesos con los que se puede comunicar cada proceso. El paso de mensajes bajo PVM es similar, pero más tedioso. Primero se debe establecer una conexión para la transmisión de un mensaje, empaquetar cada uno de los datos y después transmitirlos. En el lado opuesto debe haber una operación de recepción y después desempaquetar los datos en el mismo orden que fueron empaquetados.

```
/* Este código se ejecuta en el Proceso 0 (origen) */
MPI_Barrier(MPI_COMM_WORLD);
MPI_Send(buffer, lenbuf, MPI_CHAR, destino, type,
                                    MPI_COMM_WORLD);
...
```

(a)Proceso 0

```
/* Este código se ejecuta en el Proceso 1 (destino) */
MPI_Barrier(MPI_COMM_WORLD);
MPI_Recv(buffer, lenbuf, MPI_CHAR, origen, type,
                                    MPI_COMM_WORLD, &status);
...
```

(b) Proceso 1

Figura 2.5: Comunicación con MPI.

En la Figura 2.6 mostramos un ejemplo simple de una suma de matrices utilizando MPI. En este ejemplo, al ejecutar la aplicación cada proceso obtiene su identificación y el número de procesos que ejecutan la aplicación. Con estos valores, cada proceso determina el número de filas que le corresponde calcular.

En el código de la suma de matrices, el proceso 0 transmite las filas de A y B que le corresponden a cada proceso con la función `MPI_Scatter`. En esta función deben

```
:
  unsigned int A[ROWS][COLUMNS];             /* declarando las matrices de datos */
  unsigned int B[ROWS][COLUMNS];
  unsigned int C[ROWS][COLUMNS];
:
main() {
:
  MPI_Init(&argc,&argv);                     /* inicializando MPI */
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  rows = ROWS/nprocs;                        /* Determinar el número de filas por proceso */
  offset = myid * (ROWS/nprocs);

  /* Transmitir los valores de las matrices A y B a los procesos */
  MPI_Scatter(&A, ROWS*COLUMNS, MPI_INT, rows*COLUMNS, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Scatter(&B, ROWS*COLUMNS, MPI_INT, rows*COLUMNS, MPI_INT, 0, MPI_COMM_WORLD);

     for (r=0; r < rows; r++){
       i = r + offset;
       for (j=0; j < COLUMNS; j++){          /* calculando valores de C */
         C[i][j] = A[i][j] + B[i][j];
       }
     }

  /* Colectar los valores de la matriz C en el proceso 0 */
  MPI_Gather(&C, rows*COLUMNS, MPI_INT, rows*COLUMNS, MPI_INT, 0, MPI_COMM_WORLD);

  MPI_Finalize();                            /* finalizando MPI */
:
}
```

Figura 2.6: Ejemplo de suma de matrices con MPI.

especificarse los buffers de transmisión y recepción de los procesos, el número y tipo de datos, el proceso emisor y el comunicador. $ROWS/nprocs$ filas de C son calculadas por cada procesador. Al terminar los cálculos, cada proceso transmite los valores obtenidos de C al proceso 0. El proceso 0 colecta los valores parciales de C utilizando la función `MPI_Gather`.

En los ejemplos anteriores la identificación de los procesos es simple. Sin embargo, si la identificación de los procesos no es trivial o si el conjunto de procesos que pueden comunicarse entre sí varía, la complejidad del desarrollo de una aplicación puede incrementarse significativamente.

2.5. Programación con memoria compartida

La programación de aplicaciones paralelas en *clusters* también puede realizarse empleando el modelo de memoria compartida. Para esto se han desarrollado sistemas que ofrecen un espacio de direcciones compartido sobre máquinas con memoria distribuida. Este espacio también es comúnmente conocido como memoria compartida distribuida (DSM, del inglés distributed shared memory). Una DSM es una extensión al modelo de programación de memoria compartida en sistemas con memoria distribuida [74]. En un sistema con DSM, cada procesador tiene su propia memoria y un espacio de la memoria aparece como compartido a cada uno de los procesadores que ejecutan una aplicación paralela. El espacio de datos compartidos es lineal y accedido con operaciones de lectura y escritura normales. Este espacio de direccionamiento es adicional al espacio de memoria de cada proceso y permite que estructuras de datos complejas sean accedidas por referencia al esconder los mecanismos del paso de mensajes. Un proceso no necesita conocer la localización de los datos que desea acceder; el sistema los encontrará y accederá automáticamente.

Cuando los datos compartidos son accedidos en una DSM tienden a moverse entre las memorias de los procesadores, los cuales mantienen copias de los datos. Las copias permiten que los datos estén cerca de los procesadores que los utilizan lo que tentativamente disminuye su tiempo de acceso y permite accesos concurrentes de varios nodos. Sin embargo, cuando una de las copias es modificada se presenta un problema de consistencia para mantener la coherencia de los datos. Cuando los accesos concurrentes no son

controlados cuidadosamente, los accesos a memoria pueden ser ejecutados en un orden diferente al que espera el programador. Este problema no es trivial si se considera que la actualización de las otras copias debe hacerse utilizando la red, si se considera que la velocidad de la red es menor que la velocidad de procesamiento de las computadoras e incluso menor que la velocidad de acceso de la memoria local. Una solución a menudo es sacrificar un poco la consistencia (utilizando un modelo de consistencia) con el propósito de mejorar el desempeño.

Un modelo de consistencia permite que actualizaciones realizadas sobre la memoria compartida se hagan visibles en cada uno de los nodos en un orden establecido. Basados en el orden de actualización de las copias varios modelos de consistencia han sido propuestos [63], como son: consistencia estricta, secuencial, causal, PRAM, del procesador, débil, de liberación, de liberación perezosa y de entrada. La semántica más intuitiva para la coherencia de la memoria es la consistencia estricta. La consistencia estricta es definida por la siguiente condición: *cualquier lectura a una localidad de memoria X retorna el valor almacenado por la más reciente operación de escritura a X.* La consistencia secuencial fué definida por Lamport [74]. En donde expresa que un sistema es secuencialmente consistente si: *el resultado de cualquier ejecución es el mismo que si las operaciones de todos los procesadores se ejecutaran en algún orden secuencial y las operaciones de cada procesador individual aparecen en esta secuencia, en el orden especificado por su programa.*

De esta manera, para mantener la coherencia de los datos compartidos son necesarios varios mecanismos para controlar o sincronizar los accesos. Estos mecanismos son los encargados de buscar, de solicitar, de mover o enviar los datos a los procesadores que los utilizan y de mantener su coherencia. Los mecanismos deciden si una referencia a datos puede ser satisfecha localmente o desde otro nodo. Si la referencia no puede ser satisfecha localmente transmiten un mensaje a otro procesador solicitando una copia del dato y esperan por una réplica. Estos mecanismos también reciben las solicitudes enviadas desde otros procesadores, ejecutan las acciones necesarias de la coherencia y transmiten su respuesta.

Los tres mecanismos básicos manejados por el lado del procesador y desde el lado de la memoria son los siguientes:

- `Prueba de acierto/falla (del lado del procesador)`: Determinar si una ref-

erencia particular puede ser satisfecha localmente.

- `Solicitud para transmitir (del lado del procesador)`: Reaccionar cuando una referencia no puede ser satisfecha localmente; transmitir un mensaje a otro procesador solicitando una copia del dato y esperar por la respuesta.

- `Del lado de la memoria`: Recibir una solicitud desde otro procesador, ejecutar las acciones de la coherencia que sean necesarias y transmitir una respuesta.

Mucho trabajo se ha realizado en los últimos años para mejorar el rendimiento de los sistemas de DSM [11, 108, 35]. Johnson, Kaashoek y Wallach propusieron una clasificación de los sistemas de DSM basada en los tres mecanismos requeridos para implementar la DSM y ya sea que estos mecanismos sean implementados en hardware o en software.

Utilizando las características de los mecanismos de control y acceso de los datos compartidos, la implementación de la DSM puede ser: *todo-hardware, mayoría-hardware, mayoría-software y todo-software* [49].

2.5.1. Todo en Hardware

En los sistemas DSM todo en hardware, los tres mecanismos son implementados en hardware; ejemplos incluyen a COMA-F [48], DASH [60], DDM [107]. Son implementaciones eficientes para aplicaciones con un paralelismo de grano fino porque el tamaño de la unidad de memoria compartida es pequeña, típicamente de 16 a 128 bytes. Estas DSM tienen un costo elevado y su diseño se hace más complicado conforme aumenta el número de procesadores. Por lo que estos sistemas no son muy escalables.

2.5.2. Mayoría en Hardware

En los sistemas DSM implementados mayormente en hardware, los mecanismos del lado del procesador son implementados en hardware, pero el soporte del lado de la memoria es manejado en software. Esto es, el software del sistema es desarrollado para controlar el acceso a la DSM y el envío de los datos es realizado en hardware; ejemplos incluyen a: Alewife [2] y KSR-1 [15]. Estas arquitecturas también son muy costosas y su complejidad aumenta con el número de procesadores.

2.5.3. Mayoría en Software

Los sistemas DSM implementados mayormente en software son conocidos como sistemas de memoria compartida virtual o VSM (del inglés: Virtual Shared Memory). Son basados en paginación en los cuales la funcionalidad para verificar el acierto/falla es implementada en hardware por medio de los mecanismos de protección de la memoria virtual. A continuación describimos brevemente algunos sistemas VSM.

- **IVY** (Integrated Shared Virtual Memory at Yale) fué el primer sistema que introdujo la noción de memoria compartida distribuida en estaciones de trabajo [62]. IVY realiza paginación entre los módulos de memoria distribuida utilizando la unidad de manejo de la memoria (MMU, por sus siglas en inglés) de una estación de trabajo. IVY emplea la consistencia secuencial usando un protocolo de *múltiples lectores un solo escritor*. IVY está basado en páginas como la unidad de transferencia entre procesos, pero presenta dos desventajas: Primero, los protocolos de consistencia secuencial presentan un bajo rendimiento debido al gran número de mensajes de consistencia y, segundo, los protocolos de un solo escritor tienen el problema del *false sharing*. El *false sharing* es causado por dos o más variables localizadas en la misma página. Si cada variable es usada por un diferente procesador, las páginas se mueven entre los nodos, aún cuando no se tengan datos compartidos entre ambos nodos.

- **Munin** es un sistema de DSM basado en variables compartidas [21]. Munin también utiliza la unidad de manejo de la memoria para implementar la consistencia. Una consistencia de liberación ansiosa (del inglés: *eager release consistency*) es implementada con un protocolo de múltiples escritores. Para evitar el problema del *false sharing*, en Munin se asigna cada variable compartida a una página diferente. Sin embargo, cuando un bloque es liberado en Munin, el proceso que lo liberó transmite mensajes del dato modificado a todos los procesadores quienes lo almacenan en la caché. Este proceso es llamado *poststore*.

- **Midway** es otro sistema de DSM basado en variables [10]. Midway es similar a Munin en muchos aspectos. Su diferencia principal es el modelo de consistencia. Midway utiliza *la consistencia de entrada* (del inglés: *entry consistency*). La consistencia de entrada asocia una variable de sincronización con cada variable

compartida. De manera similar a una consistencia de liberación perezosa (del inglés: *lazy release consistency*), los mensajes de consistencia son propagados cuando las variables de sincronización son adquiridas. Sin embargo, es diferente a la consistencia de liberación (del inglés: *release consistency*), la consistencia de entrada actualiza/invalida sólo las variables compartidas asociadas con las variable de sincronización adquirida.

- **TreadMarks**: Al igual que Midway y Munin, TreadMarks utiliza la MMU para implementar la paginación [1, 102, 28]. TreadMarks se basa en primitivas del sistema operativo para detectar fallas de página (no accede al hardware directamente). TreadMarks alcanza un rendimiento equivalente a una arquitectura de paso de mensajes reemplazando la consistencia secuencial con una consistencia de liberación perezosa. La consistencia de liberación perezosa retrasa la propagación de los mensajes de consistencia hasta que una variable de sincronización es adquirida por un diferente procesador.

  El problema del *false sharing* es mitigado utilizando protocolos con mútiples escritores, los cuales permiten que una página sea escrita concurrentemente por diferentes procesadores usando diferentes partes de la página. Sin embargo, TreadMarks genera una copia de las páginas modificadas para permitirle crear una página diferencial. La página diferencial es creada comparando palabra por palabra de la página original y la modificada, conteniendo sólo los datos modificados. Este proceso es realizado en cada nodo en donde se modifique una página sin realizar intercambio de datos entre los nodos. Cuando varios procesos acceden a la misma página sólo los datos de la página diferencial se transmitirán. Aunque el tamaño del mensaje es menor y potencialmente podría reducir la sobrecarga de la red, TreadMarks mantiene dos copias de la memoria modificada sobre un nodo. Dado que un algoritmo de ordenamiento tiende a modificar muchos de los datos asignados a un nodo, se crea una gran cantidad de páginas diferenciales. Tal acción crea grandes requerimientos de memoria, por lo que TreadMarks tiende a utilizar rápidamente toda la memoria principal disponible. Para problemas con una gran cantidad de datos esto potencialmente podría degradar el rendimiento.

2.5.4. Todo en Software

Sistemas de DSM implementados usando la estrategia todo en software no cuentan con ningún soporte de hardware. Orca [9] es uno de tales DSM, pero es orientado a objetos. Las bibliotecas CRL (del inglés: *C Region Library*) [49] y Rthreads [34] son otros sistemas de DSM con todo en software.

- **CRL** (The C Region Library): En CRL los datos compartidos son definidos por el programador por medio de funciones que crean regiones de memoria compartida con un tamaño específico similar a un `malloc`. El usuario antes de acceder una región de memoria debe mapearla al espacio de dirección local y al terminar de usarla debe informar a CRL la finalización de su uso para su liberación. Cualquier número de regiones pueden ser mapeadas de manera simultánea por un solo nodo, pero el tamaño máximo está limitado a la memoria disponible en cada nodo. La dirección a la cual una región en particular es mapeada dentro de un espacio de dirección local puede no ser el mismo en todos los procesadores.

  Para acceder a los datos de una región de memoria compartida en CRL, se realizan llamadas a funciones especiales que delimitan su uso. Estas funciones realizan dos tipos de operaciones: operaciones de *lectura o de escritura*. Una operación de lectura puede realizarse de manera simultánea por varios procesadores. Una operación de escritura es realizada por un procesador a la vez en forma secuencial. Sin embargo, cada que se escribe a una región debe usarse una función que envíe los datos modificados hacia su nodo *home*.

- **Rthreads** (Remote threads): Remote threads o Rthreads es una extensión a los Pthreads entre diferentes espacios de direcciones. Pthread es una interfaz para escribir aplicaciones con múltiples hilos en una computadora con memoria compartida [80]. Rthreads es un ambiente de DSM que soporta la compartición de variables globales en un *cluster* de computadoras con memoria distribuida. Rthreads requiere llamadas explícitas a funciones para acceder los datos compartidos distribuidos. Sus primitivas de sincronización son sintáctica y semánticamente similares a las de Pthreads. El medio ambiente de Rthreads consiste de un precompilador que transforma automáticamente los programas con Pthreads en Rthreads. *Clusters*

heterogéneos son soportados por Rthreads implementándose sobre un medio ambiente de paso de mensajes como MPI o PVM.

2.6. Resumen

Las LAN o PC-clusters son hoy día una excelente alternativa para ejecutar aplicaciones paralelas porque ofrecen un beneficio extraordinario en costo-rendimiento. En las últimas décadas se investigó el uso de las LAN para explotar el paralelismo debido a su bajo costo y gran demanda. Estas redes permiten compartir recursos y facilitar la comunicación entre un gran número de procesadores. Estos beneficios han propiciado el desarrollo de los *clusters*. Un *cluster* consiste de una colección de computadoras o nodos independientes conectados conjuntamente en una LAN. Los nodos de un *cluster* pueden ser computadoras independientes o un sistema multiprocesador. Los ambientes de programación paralela utilizados en los *clusters* son herramientas portables y eficientes para el desarrollo de aplicaciones paralelas.

El tipo de aplicaciones que son comúnmente ejecutadas en los *clusters* se caracterizan por ejecutar el mismo programa en los procesadores, pero con diferentes datos. Este modelo de programación es conocido como SPMD. Las aplicaciones desarrolladas con el modelo SPMD pueden ser muy eficientes si los datos son uniformemente distribuidos a los procesadores y el sistema es homogéneo. La comunicación entre los procesadores en este modelo es realizada por medio de mensajes o con la memoria compartida distribuida.

La programación con paso de mensajes no es trivial. El programador necesita tener en mente *dónde* está el dato, debe decidir *cuándo* deben comunicarse los procesos, con *quién* comunicarse y *qué* dato transmitir. Esto hace difícil la programación y puede complicarse aún más si se utilizan estructuras de datos complejas y el número de nodos o el tamaño del problema crece.

Otra alternativa es desarrollar aplicaciones con DSM. La DSM permite que sea más fácil programar para el usuario, pero su rendimiento tiende a disminuir debido a que en ocasiones se generan más comunicaciones entre los procesos al difundir, migrar o replicar los datos entre las memorias de quiénes los solicitan. Mucho trabajo se ha realizado en los últimos años para mejorar el rendimiento de los sistemas de DSM disminuyendo estas comunicaciones. Estos sistemas están basados en el tipo de implementación de los

mecanismos de comunicación para compartir los datos: en hardware o en software o combinados. Las aplicaciones paralelas ejecutadas en estos sistemas de DSM son aplicaciones en las cuales los datos son principalmente almacenados en memoria principal.

Capítulo 3

Entrada/Salida paralela en clusters

3.1. Introducción

En la actualidad existen muchas aplicaciones que además de requerir gran capacidad de cómputo procesan una gran cantidad de datos. Algunas aplicaciones son: el análisis del clima, las aplicaciones físicas o químicas, el procesamiento sísmico, el análisis de imágenes, las bases de datos del Internet, entre otras [39, 53]. Estas aplicaciones normalmente manipulan datos del orden de cientos de GBytes en cada ejecución lo que complica su manejo en memoria. Esto es debido a que estos volúmenes de datos no pueden ser almacenados al mismo tiempo en memoria principal por el menor tamaño de esta memoria. En esos casos es necesario un espacio de almacenamiento adicional como el disco.

Una aplicación que lee/escribe los datos desde/hacia el disco es conocida como de Entrada/Salida (E/S) (o *out-of-core*). Una aplicación de E/S es diseñada para manejar explícitamente el movimiento de datos dentro y fuera de la memoria principal. Para estas aplicaciones, la memoria principal sirve como una pequeña caché y el grueso de los datos reside sobre los discos u otro almacenamiento secundario.

No obstante que el manejo de los datos se puede cubrir usando el disco, el disco es un dispositivo mecánico y es lento en comparación con el procesador y la memoria que son dispositivos electrónicos. Esta diferencia ocasiona que el uso del disco degrade el rendimiento de las aplicaciones de E/S [89].

Por otro lado, los microprocesadores incrementan su velocidad entre un 50 % a un

100 % anual, mientras que los discos normalmente incrementan su capacidad de almacenamiento en mayor grado que su velocidad de acceso [78]. Estas diferencias en velocidad (o en desempeño) hacen que el sistema de cómputo (el procesador, la memoria, el disco, la comunicación, entre otros) no funcionen en conjunto de la mejor manera. De esta manera se crea un sistema desbalanceado.

Considerando la velocidad de procesamiento y el tiempo de acceso a datos puede obtenerse el grado de balance de un sistema de cómputo. Para medir el grado de balance de un sistema de cómputo se utiliza la expresión F/b (*Regla de Thumb [78]*), donde F es la velocidad de ejecución de las operaciones de punto flotante por segundo y b la velocidad de transferencia de datos de la E/S en bits por segundo. Si la relación $F/b \approx 1$ el sistema está balanceado.

Para compensar las diferencias tecnológicas entre el procesador y el disco, una alternativa es colocar múltiples discos en paralelo que aumenten el ancho de banda en la transferencia de datos desde/hacia el disco. Sin embargo, esta organización de los discos trajo consigo problemas de integridad y la solución común fue codificar los datos con algún nivel de redundancia. El codificado normalmente consiste en determinar y probar la paridad de los datos almacenados en varios discos con la paridad que está almacenada en uno o varios disco. De tal manera que si un disco falla el dato podría ser reconstruido desde los otros discos [51]. Esta clase de discos en la actualidad son conocidos como RAIDs (del inglés: Redundant Arrays of Inexpensive Disks) [23, 105] y fue ampliamente aceptada en la industria.

Los RAIDs definen como almacenar y proteger el dato, pero no la interfaz de acceso de los datos. La mayoría de los sistemas usan una interfaz secuencial convencional, donde el RAID opera como un solo dispositivo que tiene una mayor capacidad y un mayor ancho de banda. Sin embargo, en los sistemas paralelos, múltiples tareas son ejecutadas a la vez y los dispositivos de E/S llegan a ser un recurso compartido. Consecuentemente tal interfaz es con frecuencia un limitante, porque los datos necesitan ser accedidos en paralelo por múltiples procesadores. Para eliminar este problema, los dispositivos de E/S son conectados de manera independiente e igualmente accesibles por todos los procesadores. Esta organización tiene la ventaja de que una tarea no será perturbada por las operaciones de E/S de cualquier otra tarea, como sería el caso si en alguna forma los dispositivos de E/S fueran ligeramente acoplados (por ejemplo, conectados por medio

de una red) a la primer tarea. Basados en estos argumentos, varios diseños de máquinas paralelas con arquitecturas de E/S paralelas fueron contruídos para mejorar el acceso a datos desde el disco [36, 37]. Algunos ejemplos incluyen la Connection Machine CM-5 de Thinking Machines Corp, la nCUBE, la Intel iPSC, la Intel Paragon, la Meiko CS-2 y la SP2 de IBM, entre otras [101].

Durante los 90s se diseñaron sistemas masivamente paralelos (MPP, del inglés Massive Parallel Processor) con dispositivos de E/S conectados en red [50, 89]. En estos sistemas se utilizaron nodos de E/S dedicados que proveen una E/S de alto rendimiento a las aplicaciones paralelas. Un nodo de E/S generalmente es un nodo de procesamiento con al menos un disco. Los nodos de E/S son utilizados para almacenar los datos, reducir la carga y la congestión de la E/S que se produce cuando los discos son compartidos por varios procesadores. Los nodos de E/S son integrados internamente al sistema MPP o en forma externa mediante uno o varios buses de interconexión. La Figura 3.1 muestra las topologías básicas de las arquitecturas de E/S con nodos de E/S internos y externos, incluyendo el de un *cluster* con nodos de E/S.

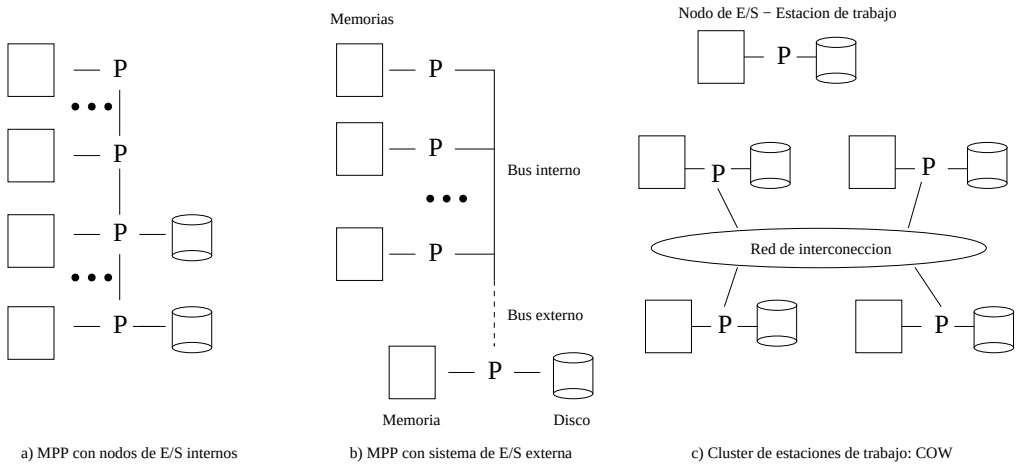

Figura 3.1: Topologías de las arquitecturas de E/S.

Muchos de los MPP modernos (como Intel Paragon, Meiko CS-2, SGI ORIGIN 3900, IBM SP2 e IBM Blue Gene/L presentados en [101]) utilizan nodos de E/S internos (ver Figura 3.1.a) para mejorar el rendimiento de la E/S. En estos sistemas los nodos pueden operar como nodos de cómputo o como nodos de E/S ya que contienen procesador, memoria y disco, por lo que el software, ya sea el sistema operativo y/o el programa

de la aplicación hacen la distinción entre cada uno. Estos sistemas comúnmente son integrados con una red de alto rendimiento (del orden de Gigabits) y ofrecen una gran capacidad de procesamiento. Sin embargo, debido a que los costos de estos sistemas son muy elevados su disponibiliad es restringida.

En la actualidad, la gran disponibilidad de las computadoras personales (PCs) y su bajo costo, junto con las nuevas tecnologías de comunicación (como Giganet, Myrinet e Infiniband) han sido una opción para construir un *cluster* de PCs (o PC-cluster) con muchos discos (ver Figura 3.1.c). Los *clusters* son particularmente apropiados para ejecutar aplicaciones paralelas de E/S, ya que cada nodo puede ser fácilmente configurado con uno o más discos y usados en paralelo para mejorar el tiempo de acceso a datos. Los nodos y discos crecen linealmente y por consiguiente, la capacidad de almacenamiento en disco es proporcional al número de los nodos. También pueden usarse las memorias de los nodos de un *cluster* para construir una caché de mayor tamaño que la de un nodo con la finalidad de mejorar el rendimiento de los discos.

Sin embargo, el desarrollo de estas aplicaciones demanda grandes esfuerzos del programador, quien tiene que definir y/o conocer el particionamiento de los datos tomando en cuenta el número de nodos de E/S, el tamaño de las particiones, el tamaño de los bloques, entre otros aspectos (tarea que no es sencilla aún para un experto).

En lo que resta de este capítulo veremos los aspectos de diseño y ejemplos de sistemas de E/S paralela particularmente en *clusters* con sus ventajas y sus limitaciones.

3.2. Aspectos de diseño de E/S paralela

La E/S paralela consiste en dividir y almacenar o distribuir los archivos de datos de una aplicación en varios discos permitiendo el acceso simultáneo de los datos por los procesadores mejorando su desempeño. Cuando un acceso simultáneo es realizado, varios bloques de datos son leídos a la vez desde varios discos para formar un solo bloque de datos de mayor tamaño. En el caso ideal, el tiempo de acceso del bloque mayor es aproximadamente igual al tiempo de acceso de un bloque menor (considerando que son accedidos al mismo tiempo en cada disco) más la sobrecarga del sistema para llevarlos y unirlos en una misma área de memoria. Cuando el número de discos se incrementa, un archivo puede dividirse en más bloques de menor tamaño reduciendo el tiempo de

acceso de los datos.

En un *cluster*, los archivos se almacenan en los discos de diferentes nodos lo que implica hacer transferencias de datos entre nodos. Los nodos son conectados por un medio de comunicación y los datos viajan a través de este medio, por lo que también es necesario considerar el tiempo que tardan los datos en viajar a través de los medios de comunicación entre los nodos. Por otro lado, si los bloques leídos son almacenados temporalmente en la memoria de los nodos, los accesos posteriores a estos bloques podrán ser satisfechos desde esta memoria reduciendo el número de accesos a disco y aumentando el rendimiento del sistema de E/S.

Considerando los aspectos anteriores varias estrategias de E/S paralela han sido desarrolladas con al menos uno de los siguientes objetivos: 1) utilizar el mayor número de discos en paralelo para incrementar el ancho de banda, 2) disminuir el número de operaciones de lectura y escritura a disco (debido a que son los elementos más lentos de un sistema), 3) minimizar el número de mensajes relacionados con E/S entre nodos para disminuir los costos en la comunicación y 4) aumentar la tasa de acierto local (*hit ratio*) para evitar los accesos a datos remotos.

En general, el diseño de un sistema de E/S paralela de alto rendimiento debe distribuir los datos entre los discos de tal manera que puedan ser leídos/escritos en paralelo desde/hacia tantos discos como sea posible y el acceso a datos pueda hacerse desde cualquier nodo en el *cluster* en forma transparente.

Otros aspectos importantes del diseño de un sistema de E/S paralela son facilitar el desarrollo de programas y mejorar el desempeño. Estos aspectos se logran con el *caching*, el *prefetching*, las operaciones colectivas, el control de concurrencia, apuntadores de archivo compartido y acceso paralelo, entre otros.

Antes de analizar los sistemas de E/S paralela se presenta una descripción detallada de los aspectos de diseño de E/S paralela.

3.2.1. Particionamiento y distribución de datos

En un sistema de E/S paralela los archivos de datos son generalmente divididos en particiones de tamaño fijo, donde cada partición es dividida en bloques. La división del archivo en particiones y las particiones en bloques, permite ver al archivo como un arreglo de dos dimensiones. Las particiones (también conocidas como *stripes*) son porciones

contiguas de un archivo y son definidas al momento de creación del archivo. El número de particiones permanece constante durante toda la vida del archivo. Cada partición puede ser almacenada en uno o en diferentes nodos de E/S (discos). La asignación de los nodos en donde se almacenan las particiones es realizada por el sistema de E/S sin intervención del usuario. No obstante el usuario puede dar sugerencias sobre que partes del archivo son mejores dentro del mismo nodo (o disco) y cuales partes serían ubicadas en diferentes discos para incrementar el ancho de banda de la E/S.

Los bloques o fragmentos de datos son conjuntos de bytes contiguos que son almacenados en un solo nodo de E/S. El tamaño del bloque también es ajustado al momento de creación del archivo y normalmente permanece sin modificarse durante toda la vida del archivo.

La descripción de los archivos de datos se almacena en un archivo de datos llamado *archivo de metadatos*. Un *metadato* contiene los parámetros que establecen el particionamiento de un archivo en varios nodos de E/S. El archivo de metadatos puede almacenarse en un solo nodo, o estar replicado en todos los nodos de E/S. Los metadatos permiten localizar un bloque de datos dentro de un archivo que está almacenado en varios nodos de E/S y es necesario para especificar el particionamiento y distribución de cada archivo en un sistema de E/S paralela.

De esta manera, los archivos de datos son generalmente divididos en bloques de tamaño fijo, los cuales son distribuidos cíclicamente a los discos montados en diferentes nodos (ver Figura 3.2). En este simple esquema de particionamiento, el número de discos (o nodos de Entrada/Salida) es llamado el *factor stripe* y el tamaño de los bloques es el *fragmento del stripe* (Figura 3.2).

El *factor stripe* determina el grado de paralelismo y por lo tanto la máxima razón de transferencia. El tamaño del *stripe* puede ser tan pequeño como unos cuantos bytes o tan grande como varios sectores del disco se tengan disponibles.

3.2.2. Caching

La caché del sistema de archivos, o *buffer* de la caché, es un mecanismo que minimiza el número de accesos a disco de un sistema de archivos. La idea básica consiste en mantener los bloques de los archivos usados por la aplicación en *buffers* de memoria. Cuando una aplicación se ejecuta tiene una localidad temporal (al cargarse en memoria),

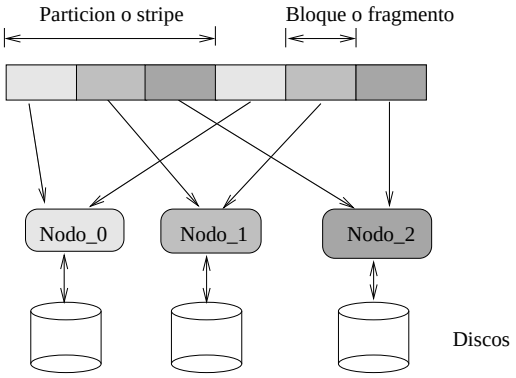

Figura 3.2: Distribución de datos en disco típica en sistemas de archivos paralelos.

el mismo conjunto de bloques se utiliza durante cierta cantidad de tiempo. En este caso, los bloques solicitados sólo tienen que ser leidos desde el disco la primera vez, la próxima vez que sean solicitados serán encontrados en la memoria caché. Las escrituras son realizadas directamente en la caché y los bloque son llevados al disco cuando el espacio en la caché es insuficiente o después de un determinado tiempo que es establecido por el sistema de E/S o por el sistema operativo. Este simple mecanismo incrementa el rendimiento de las operaciones de lectura y escritura de manera significativa.

Este mecanismo puede ser aplicado a muchos niveles; puede encontrarse una caché en el controlador del disco, en el sistema operativo, en las bibliotecas de E/S y en el código del usuario. En un PC-cluster es posible tener aún más niveles en donde puede usarse el *caching*. Si el cliente de una aplicación se ejecuta en un nodo que no tiene un disco, todas las solicitudes irán a un nodo diferente que tenga un sistema de E/S. Esto crea nuevas posibilidades de *caching*. Podemos tener una caché en el servidor, en el cliente o en ambos. Además, si el cliente tiene un disco local, puede utilizar su propio disco como una caché de la información remota reduciendo la comunicación con los servidores.

Sin embargo, si varios nodos clientes (procesos) depositan en la caché local los datos que su aplicación necesita y uno de ellos modifica sus datos, se generará *el problema de la coherencia de la caché*. Para mantener la coherencia de la caché, cada nodo mantiene en su caché local una copia de un bloque dado que también es usado por los procesos de la aplicación. Si uno de los procesos modifica su copia del bloque, el resto de los procesos no tendrá una copia actualizada. Para resolver este problema dos soluciones han sido propuestas [79]. La primera consiste en una semántica relajada al compartir los datos. La

segunda semántica trata con una caché cooperativa. El problema de la primer semántica es que no es fácil de implementar en sistemas de archivos paralelos/distribuidos en un PC-cluster. Los programadores de la aplicación tienen que rediseñar su código y la carga de trabajo para mantener una semántica más estricta es dejada al programador de la aplicación. El programador debe utilizar *tokens* o instrucciones de control de acceso para iniciar y finalizar el uso de un bloque de un archivo. En la segunda semántica, un bloque modificado es visto inmediatamente por todas las aplicaciones, en la cual existe un *caching* cooperativo entre los nodos del sistema.

Una caché cooperativa es una caché que trata de mejorar el rendimiento de de un sistema de archivos paralelo/distribuido, coordinando el contenido de las cachés encontradas en todos los nodos. Esta coordinación permite que una solicitud desde un nodo dado sea servida por la caché local de un nodo diferente. Sin embargo, aunque esta cooperación mejora el rendimiento, un bloque puede estar replicado en varios nodos y es necesario un algoritmo de coherencia de la caché, lo cual incrementa la complejidad del sistema. Este método además requiere de más espacio de la memoria ya que define una caché local en donde se almacenan los bloques necesarios por los procesos locales y una caché global que es utilizada para mantener los bloques necesarios por otros nodos pero que no pueden ser guardados en sus caches.

3.2.3. Prefetching

El *prefetching* es otro de los aspectos considerados para incrementar el rendimiento del sistema de E/S. El *prefetching* consiste en anticipar el acceso de los bloques antes de que sean solicitados por el usuario y traerlos a memoria. Esta técnica aumenta el desempeño cuando un bloque es accedido por primera vez porque el bloque será accedido desde la memoria en lugar del disco. Este mecanismo comparte todas las estructuras de datos usadas por el *caching* más un *bloque predictor*. El predictor es el encargado de decidir cuales bloques son más probable a ser accedidos en un futuro cercano.

Aunque el *prefetching* es una técnica ampliamente usada en sistemas comerciales y prototipos de investigación, la mayoría del trabajo actual ha sido dedicado a sistemas mono-procesador. Esta simple idea es usada para realizar un *prefetching* paralelo, en donde cada nodo anticipa el acceso de sus datos de manera independiente. El paralelismo ofrecido por múltiples discos es usado para anticipar los datos de las solicitudes emitidas

por cualquiera de los nodos.

3.2.4. Operaciones colectivas

Al incrementar el número de discos en la E/S paralela para distribuir un archivo, los bloques se hacen más pequeños con lo cual las solicitudes de E/S llegan a ser muy pequeñas (del orden de cientos de bytes). Conforme se incrementa el número de discos se incrementa el grado del paralelismo en una aplicación de E/S paralela. Cuando los procesos solicitan muy pocos bytes en una operación, el sistema de E/S tiene problemas para alcanzar un alto desempeño porque al tener una solicitud muy pequeña el tiempo de acceso al disco es dominado por la operación de búsqueda en el disco (el *seek time*). Para eliminar este tiempo o disminuir su efecto en una operación de acceso a disco, una E/S colectiva ha sido propuesta.

En una E/S colectiva todos los nodos de cómputo cooperan entre si para ejecutar las operaciones de E/S en el disco a fin de mejorar su eficiencia. Esto permite al sistema construir una sola operación de mayor tamaño uniendo todas las solicitudes pequeñas emitidas por cada cliente.

3.2.5. Control de la concurrencia

Un desafío para un sistema de E/S es soportar acceso concurrente desde varios procesos. Al particionar un archivo sobre mútliples discos conectados a diferentes nodos, el sistema de E/S tiene que manejar dos diferentes mapeos de datos: *el mapeo desde múltiples nodos de cómputo a un archivo compartido* y *el mapeo de un archivo compartido a múltiples nodos de E/S* (o discos).

La Figura 3.3 muestra un ejemplo de este mapeo con cuatro nodos de cómputo y dos nodos de E/S. Las filas de arriba muestran a los cuatro nodos de cómputo, cada uno con dos bloques de datos. La fila de en medio muestra a los bloques dentro de la estructura lógica del archivo. Las filas de abajo muestran como el archivo es particionado sobre los nodos de E/S. En este ejemplo, la distribución de los datos entre los nodos de cómputo empareja al particionamiento del archivo, así que cada nodo de cómputo puede mandar bloques enteros de datos a uno solo de los nodos de E/S. Los datos se mueven directamente entre los nodos de cómputo y los nodos de E/S; todos los datos

nunca residen juntos en un mismo lugar como es mostrado por la fila de en medio.

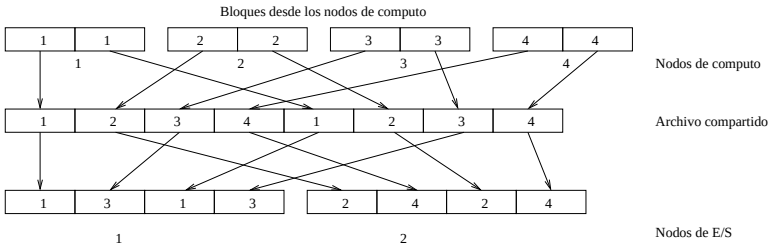

Figura 3.3: Mapeo de bloques de datos de un archivo compartido desde nodos de cómputo a nodos de E/S.

Una vez que se ha hecho el particionamiento y el mapeo de un archivo paralelo, cada proceso inicia abriendo el archivo antes de realizar un acceso al archivo. Cada nodo de E/S maneja un subconjunto de los bloques que forman el archivo en un subarchivo local, así cada archivo tiene un *inodo* (del inglés: inode) sobre cada nodo de E/S. Un *inodo* es una estructura de datos que contiene una lista de los bloques del disco que mantienen los datos en un archivo. El sistema de archivos necesita una forma para buscar cada uno de estos *inodos* cuando abre el archivo. Una posible solución es que cada nodo de E/S mantenga su propio directorio de información y busque en sus propios *inodos*. Otra solución es utilizar un servidor central de nombres, un proceso que recibe las peticiones de búsqueda de los números de los *inodos* que está usando un archivo que son emitidas por los nodos de E/S. Esto evita la necesidad de replicar el directorio de datos sobre cada nodo de E/S.

Algunos sistemas de archivos ajustan el *factor stripe* y la *longitud del stripe* cuando el sistema es configurado; otros permiten al usuario especificar estos parámetros en forma separada para cada archivo. Cuando el archivo es creado, el sistema de archivos selecciona al nodo de E/S que almacenará al primer bloque del archivo. Al variar esta localización, se distribuye el trabajo entre los nodos de E/S; si la localización es fija, todos los archivos pequeños residen en el mismo nodo de E/S. Los bloques restantes son almacenados en los otros nodos de E/S en un orden fijo o en un orden aleatorio, dependiendo de su patrón de particionamiento.

El patrón de particionamiento y el bloque inicial del archivo permiten a cada proceso determinar cual nodo de E/S es responsable de la localización de cada byte de un

archivo. Cuando una lectura o escritura de un archivo es emitida, los nodos de cómputo determinan cual nodo de E/S manejará cada dato que va a ser movido. Los nodos de cómputo transmiten las solicitudes a los nodos de E/S en forma paralela y los nodos de E/S llevan a cabo las solicitudes en paralelo. Este particionamiento es fácil de manejar debido a que los nodos de E/S son los que tienen el control de acceso de los dispositivos de almacenamiento.

El problema surge cuando el sistema tiene que mantener la consistencia secuencial cuando varios procesos sobre diferentes nodos de cómputo tratan de acceder a una misma parte de un archivo. Si tenemos a dos procesos que escriben al mismo rango de localizaciones dentro de un archivo y el rango se extiende a dos bloques sobre diferentes nodos de E/S (ver Figura 3.4), la consistencia secuencial requiere que los dos nodos de E/S escriban sus porciones de datos en el mismo orden, asegurando que las solicitudes de escritura ocurran en una secuencia bien definida.

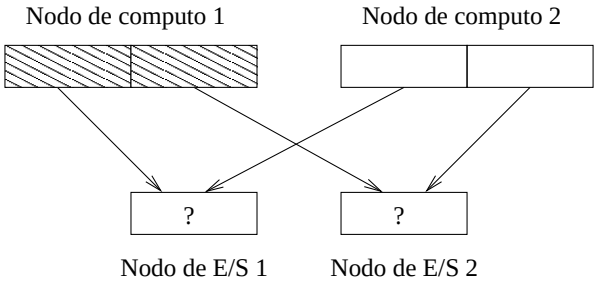

Figura 3.4: Acceso concurrente de dos nodos de cómputo para escribir sus datos en un mismo espacio de un archivo.

3.2.6. Apuntadores de archivo compartidos

Los programas que escriben a un solo archivo concurrentemente desde varios procesos necesitan un soporte especial para los apuntadores de archivo. En la semántica de UNIX, cada proceso que abre un archivo tendrá su propio apuntador de archivo que el sistema de archivos actualiza independientemente de los otros procesos. En un archivo paralelo compartido, todos los procesos normalmente tienen apuntadores de archivos separados. Sin embargo, cada vez que un proceso escribe algún dato podría dañar los datos que otro proceso escribió con anterioridad. Para prevenir este problema es necesario un

mismo apuntador de archivo que es compartido por los procesos, todos los procesos necesitan una vista común de la localización actual del apuntador de archivo. Así cuando cualquier proceso escribe datos, el sistema de archivos actualiza el apuntador sobre los otros procesos. Un problema similar surge cuando varios procesos necesitan leer un solo archivo que contiene una lista de tareas por hacer de manera coordinada: no importa que tarea realiza cada proceso, pero exactamente un proceso debe leer cada tarea.

Varios sistemas de archivos ofrecen apuntadores de archivo compartido, los cuales permiten esta clase de acceso coordinado. Sin embargo, los apuntadores de archivo son difíciles de implementar eficientemente. Los sistemas de archivo deben dar a todos los procesos acceso del apuntador de archivo, pero deben permitir a un solo proceso a la vez para actualizarlo. Una solución común es usar un proceso servidor. El proceso servidor coordina todas las solicitudes de los nodos de cómputo para leer o actualizar el apuntador de archivo. Utilizar un apuntador de archivo compartido reduce (pero no necesariamente se elimina) el acceso paralelo a un archivo, así que los programas lo usarían sólo cuando sea necesario.

3.3. Alternativas de implementación de sistemas de E/S paralelos

Hasta ahora hemos presentado los aspectos básicos en el diseño de E/S paralela. A continuación discutiremos como han manejado el problema de la E/S diferentes grupos de investigación.

En general, dos alternativas son consideradas: bibliotecas de E/S como soporte para lenguajes de alto rendimiento y sistemas de archivos paralelos.

Las bibliotecas de E/S realzan los lenguajes convencionales de alto rendimiento tales como Fortran o C/C++, mientras que los sistemas de archivos paralelos son una solución de bajo nivel (el sistema operativo es fortalecido con características especiales que tratan directamente con la E/S al nivel de archivos).

3.3.1. Bibliotecas de E/S

En los últimos años, muchos grupos de investigación han trabajado sobre diferentes interfaces de E/S. Algunas interfaces son conservadoras y sólo presentan pequeños cambios sobre la interfaz tradicional del sistema Unix. Otras, han tratado de ser innovadoras proponiendo nuevas semánticas. Estas nuevas semánticas son diseñadas para permitir al usuario especificar la naturaleza de sus aplicaciones de una manera más precisa.

El desarrollo de nuevas semánticas ha seguido básicamente dos alternativas. 1) Integrar estas nuevas semánticas dentro del mismo sistema de archivos y 2) Construir una biblioteca para ofrecer las nuevas semánticas al usuario. La primera opción realiza una mejor integración con el sistema de archivos, por consecuencia ofrece un buen rendimiento. La segunda opción es mucho más flexible y portable debido a que no es dependiente de un sistema de archivos específico.

El propósito de las bibliotecas de E/S es proveer al programador con una herramienta que facilite la creación de sus programas y mejore su desempeño utilizando un lenguaje de programación convencional como C, C++ o Fortran.

Las bibliotecas de E/S, tentativamente, también ofrecen al programador un manejo transparente entre el cómputo y el acceso local de la E/S de una solicitud generada desde un nodo remoto. Sin embargo, cuando se requiere realizar de manera simultánea el cómputo y las solicitudes de E/S puede ser muy complicado explotar el paralelismo entre estas operaciones. Esto es debido a que el programa de la aplicación es ligado con las bibliotecas y debe ejecutar ambas tareas con el mismo código porque las bibliotecas no tienen un servidor de E/S. De esta manera, este procesamiento normalmente es realizado en forma de *pipeline*, es decir lo que complica su programación.

El problema anterior puede ser un factor limitante del ancho de banda de la E/S. En consecuencia, el desempeño de las aplicaciones puede disminuir o no ser el mejor al utilizar estas bibliotecas. Las ventajas de usar estas bibliotecas es que son una alternativa rentable debido a su relación de costo-efectividad en muchas clases de aplicaciones paralelas; pueden ser implementadas sobre PC-clusters; y no requieren modificar el sistema operativo.

A continuación describimos brevemente algunas de estas bibliotecas y en la tabla 3.1 mostramos un resumen de sus características:

Panda

Panda (The Panda Project: Data Management for High-Performance Scientific Computation) [25, 94] es una biblioteca de E/S diseñada para desarrollar programas *out-of-core* de tipo SPMD ejecutándose en arquitecturas paralelas de memoria distribuida. Consiste de nodos de cómputo llamados clientes y nodos de E/S llamados servidores. En esta biblioteca, la aplicación se comunica con el cliente local por medio de una interfaz de alto nivel. Mediante esta interfaz el cliente pasa a los servidores una descripción (de alto nivel) de la solicitud de E/S, conteniendo la distribución de los datos en memoria y disco. Entonces, los servidores leen secuencialmente los datos desde los archivos y los esparcen a los clientes tan pronto se leen del disco. En el caso de una escritura, el servidor reúne los datos desde los clientes y los escribe secuencialmente al disco. Esta biblioteca permite seleccionar una organización eficiente de datos entre los servidores y optimiza los accesos a disco desde los clientes [24].

PASSION

PASSION (del inglés: Parallel and Scalable Software for Parallel I/O) [26] es una biblioteca de E/S que enfoca el problema de la E/S desde tres puntos de vista: del lenguaje, el compilador y la ejecución. PASSION implementa técnicas para optimizar los accesos no contiguos. En PASSION, cuando la aplicación genera solicitudes de E/S no contiguas, las solicitudes son combinadas y una sola solicitud de E/S de mayor tamaño es realizada al sistema de archivos. Todos los datos son llevados hacia la memoria en una sola solicitud y los datos necesitados son extraídos desde la memoria. Este método de dos fases habilita la implementación de E/S paralela para analizar y combinar las solicitudes de E/S de diferentes procesos (en un ambiente paralelo). En muchos casos, las solicitudes combinadas pueden ser grandes o contiguas, aunque las solicitudes individuales de cada proceso no sean contiguas. Por lo tanto, las solicitudes combinadas pueden ser servidas eficientemente.

TPIE

TPIE (del inglés: Transparent Parallel I/O Environment) [7, 103] es un sistema diseñado para facilitar el desarrollo de algoritmos de E/S sobre sistemas de discos paralelos.

La biblioteca TPIE consiste de un kernel y un conjunto de algoritmos eficientes de E/S y estructuras de datos implementadas sobre el kernel. TPIE opera en el modelo SPMD y está basado en el lenguaje C++. Para escribir programas con TPIE, los programadores definirán una clase de objetos sobre los cuales operarán sus programas. El sistema consiste de tres componentes: una máquina de transferencia de bloques (BTE), un manejador de memoria (MM) y una interfaz de acceso (AMI). El BTE maneja la transferencia de bloques de un solo procesador. El MM ejecuta el manejo de la memoria a bajo nivel a través de todos los procesadores en el sistema. La AMI trabaja sobre la MM y uno o más BTEs. Cada BTE se ejecuta sobre un solo procesador para proporcionar una interfaz uniforme a los programas de la aplicación.

El BTE trabaja junto con el buffer tradicional de la caché en un sistema de archivos. Diferente al buffer de la caché, el cual debe soportar accesos concurrentes a archivos desde múltiples espacios de dirección, el BTE es diseñado específicamente para soportar un alto procesamiento de datos desde la memoria secundaria a través de un solo espacio de dirección del nivel de usuario.

ViPIOS

ViPIOS [88] es un módulo de soporte de E/S para lenguajes de alto rendimiento, ViPIOS provee un acceso eficiente a archivos para optimizar la distribución de los datos en los discos y permite operaciones paralelas de lectura/escritura.

La arquitectura de ViPIOS es distribuida. En ViPIOS el ancho de banda proporcionado por la E/S es principalmente dependiente de la disponibilidad de los nodos de E/S. Todos los archivos de datos y metadatos son almacenados en forma distribuida y paralela a través de múltiples dispositivos de E/S. El usuario y el compilador pueden dar sugerencias de la distribución de los archivos de datos. Esto permite al sistema ejecutar varios *modos de acceso a datos* en forma eficiente y paralela.

Las operaciones de acceso a disco son separadas de la aplicación y ejecutadas por un sistema de E/S independiente para reducir el congestionamiento. De esta manera, una aplicación transmite las solicitudes de disco a ViPIOS, el cual a su vez ejecuta los accesos a disco.

La arquitectura de ViPIOS es construida sobre un conjunto cooperativo de procesos servidores, los cuales efectúan las solicitudes de los procesos clientes de la aplicación.

Los procesos servidores se ejecutan de manera independiente sobre todos los nodos o sobre un número de nodos de procesamiento dedicados en un sistema MPP. También es posible que un cliente de la aplicación y un servidor compartan el mismo procesador.

MPI-IO

MPI-IO es una de las plataformas más utilizadas en PC-clusters para desarrollar aplicaciones paralelas de E/S. MPI-IO [69] está diseñado como un estándar de la interfaz de E/S paralela basado en la plataforma de paso de mensajes MPI. La idea principal es que la E/S sea modelada como paso de mensajes. Así, escribir a un archivo es como mandar un mensaje, mientras que leer de un archivo corresponde a recibir un mensaje. MPI-IO soporta una interfaz de alto nivel que incluye el particionamiento del archivo entre múltiples procesos, transferencias entre procesos de las estructuras de datos globales de las memorias y los archivos y optimizaciones de la distribución física del archivo en los dispositivos de almacenamiento.

MPI-IO provee tres diferentes funciones de acceso, incluyendo posicionamiento, sincronización y coordinación. El posicionamiento es realizado con *offsets* explícitos, apuntadores de archivo individual y apuntadores de archivo compartidos. Para la sincronización, MPI-IO provee versiones síncronas y asíncronas (bloqueantes y no bloqueantes, respectivamente) y para la coordinación se tienen funciones colectivas y no colectivas.

El acceso a un archivo puede ser independiente o colectivo. MPI-IO utiliza un tipo de datos derivado de MPI para definir la distribución de los datos dentro del archivo y para acceder los archivos compartidos. El tipo de datos para los archivos en MPI-IO es mostrado en la Figura 3.5. Un archivo con MPI-IO puede ser distribuido entre procesos paralelos en forma de bloques disjuntos. Además, MPI-IO puede ser usado en los lenguages de programación C y Fortran.

Existen varias implementaciones de MPI-IO:

- PMPIO - Biblioteca de MPI-IO portable desarrollada por la NASA Ames Research Center.
- ROMIO - Una implementación de MPI-IO portable, de alto rendimiento, que es desarrollada por el Argonne National Laboratory, ha sido incluida dentro de las implementaciones ampliamente distribuidas y portables de MPI: MPICH y LAM; ambas disponibles en los sistemas Linux.

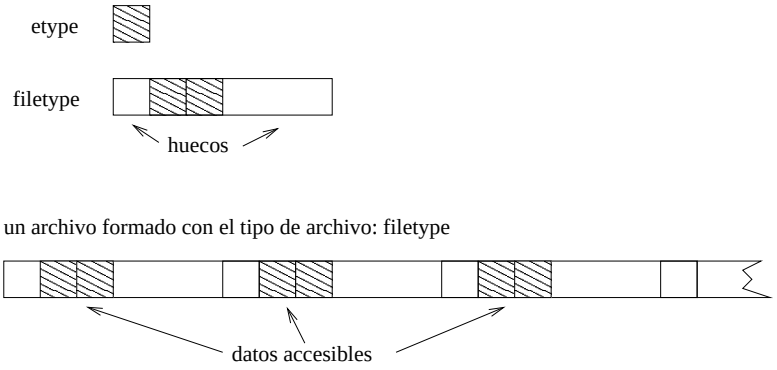

Figura 3.5: Estructura de un archivo en MPI.

- MPI-IO/PIOFS - Desarrollada por IBM Watson Research Center, y
- HPSS - Implementación desarrollada por el Lawrence Livermore National Laboratory.

Sin embargo, aún con todo este conjunto de funciones, el usuario tiene que realizar gran parte del trabajo para mantener la coherencia cuando existen accesos concurrentes a un mismo bloque.

Una lista detallada de las bibliotecas de E/S presentadas en el capítulo con las características que soportan y sus limitaciones es mostrada en la Tabla 3.1 (para mayor detalle de comparación ver [95]).

3.3.2. Sistemas de archivos paralelos

Un sistema de archivos paralelo permite un acceso colectivo y paralelo a un archivo almacenado en varios nodos. Comparado a una biblioteca de E/S, los sistemas de archivos paralelos tienen la ventaja de que se ejecutan en forma independiente a la aplicación. El sistema paralelo incluye a los servidores de E/S (nodos de E/S) y, de esta manera, los nodos de procesamiento pueden concentrarse en el programa de la aplicación y no ser sobrecargados por las solicitudes de E/S.

Sin embargo, los sistemas de archivos paralelos no son fáciles de usar con los lenguajes de alto rendimiento, proporcionan a la aplicación un acceso limitado del disco y no permiten que un programador pueda coordinar los accesos a disco. En la mayoría de los casos, el programador se enfrenta a una caja negra, lo que dificulta y hace muy difícil

Tabla 3.1: Bibliotecas de E/S.

Nombre	Sincrono/asincrono	Caching	prefetching	Operaciones colectivas	Control de concurrencia	Apuntador de archivo compartido	Datos paralelos	Nuevas Ideas
Panda	+	+	−	+	+	−	−	Servidores E/S, compresion de datos
PASSION	+	+		+				TPM, problemas irregulares
TPIE	+						+	Ambiente de algoritmos para E/S paralela transparente
ViPIOS		+		+	−	+	+	Influencia desde tenologias de DB
MPI−IO	+	−	−	+		+	+	E/S para MPI
MPI 2	+	−	−	+		+	+	E/S para MPI
ROMIO	+	−	−	+	+	+	+	Implementacion portable de MPI−IO

Notacion: + ... es soportado
− ... no es soportado

la realización de un mapeo óptimo y no permite un acceso a disco en forma óptima. El mapeo debe darse entre una distribución o vista lógica de los datos del programa de la aplicación y la distribución física en disco de los datos (ver Figura 3.6).

Para mejorar el desempeño de aplicaciones con E/S se han desarrollado diversos sistemas de archivos paralelos [81, 84, 85, 100]. Los sistemas de archivos paralelos pueden ser divididos en tres grupos: sistemas de archivos paralelos comerciales, sistemas de archivos paralelos de proyectos de investigación y sistemas de archivos para *clusters*.

El primer grupo comprende los sistemas de archivos comerciales tales como PFS para la Intel Paragon [66], PIOFS y GPFS para la IBM SP [66], HFS para la HP Exemplar [66] y XFS para la SGI Origin2000 [66]. Estos sistemas de archivo paralelo incluyen las tecnologías de E/S actuales, tales como: E/S paralela, *caching*, prefetching, entre otras, con la finalidad de proveer un alto rendimiento y la funcionalidad deseada para aplicaciones intensivas de E/S. Sin embargo, los dispositivos de almacenamiento son ligeramente acoplados con los nodos de cómputo y sólo están disponibles en plataformas específicas en las cuales el vendedor los ha implementado. Agregar recursos de almacenamiento externos a estos sistemas no es una tarea fácil.

El segundo grupo de sistemas de archivos paralelo de proyectos de investigación

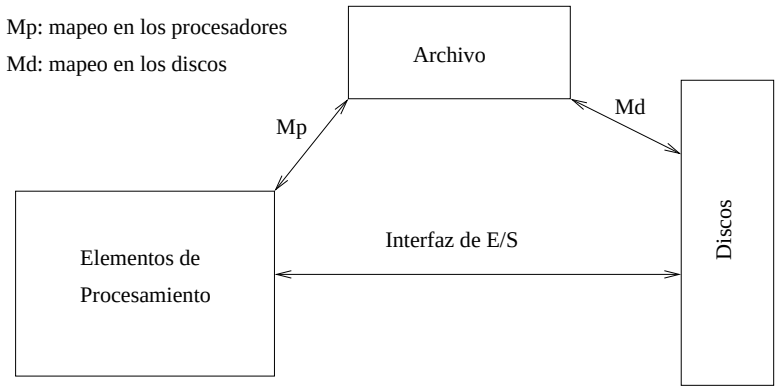

Figura 3.6: Mapeo de datos entre los procesadores y los discos.

incluye entre otros a: Vesta [29, 30, 66], PIOUS [66, 71], PPFS [44], Galley [73] y DPFS [92].

- **Vesta**: Vesta es un sistema de archivos *opensource* desarrollado por IBM (PIOFS es la versión comercial). Vesta abandona el modelo Unix de un archivo como una secuencia lineal de bytes. Vesta particiona físicamente los archivos en múltiples subarchivos separados (llamados celdas), que pueden ser accedidos en paralelo y estos a su vez en bloques (BSUs, del inglés Basic Stripe Unit) (ver Figura 3.7). Vesta también ofrece la posiblidad de particionar lógicamente un archivo. Ambos esquemas de particionamiento (físico y lógico) son restringidos sólo por el conjunto de datos que pueden ser particionados dentro de arreglos rectangulares de dos dimensiones. Sin embargo, las aplicaciones tienen poco control sobre el *caching*.

- **PIOUS** (Parallel Input/Output System): PIOUS es un sistema de archivos paralelos que proporciona almacenamiento permanente a grupos de procesos en una red de cómputo. En PIOUS un archivo está compuesto de varios segmentos o subarchivos separados (ver la Figura 3.8). Cada segmento reside en un solo servidor de E/S (PDS, del inglés PIOUS Data Server). En PIOUS las operaciones de E/S son ejecutadas desde el punto de vista de las transacciones para proveer la consistencia secuencial en el acceso a archivos. Sin embargo, está diseñado para usarse sólo con PVM.

- **PPFS** (Portable Parallel File System): PPFS se enfoca sobre el caching y el

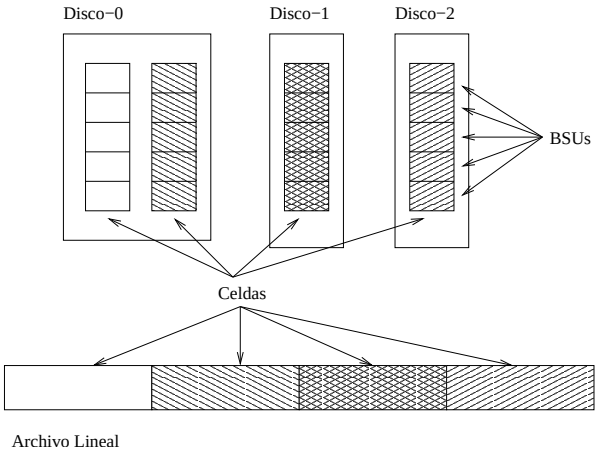

Figura 3.7: Distribución de datos en Vesta con un archivo de dos dimensiones.

prefetching adaptivo. PPFS es un sistema de archivos paralelos portable que permite a las aplicaciones controlar el caching, el pre-fetching, la distribución de los datos y las políticas para compartir archivos en PPFS. Los archivos son divididos dentro de registros de tamaño variable, llamados segmentos (ver Figura 3.9). Cada segmento es manejado por un solo servidor de E/S. PPFS es implementado como una biblioteca al nivel del usuario portable a través de varios sistemas de archivos paralelos.

- **Galley**: Galley es un sistema de archivos paralelos que se enfoca en las optimizaciones de acceso a disco y alternativas de organizaciones de archivos. Los archivos en Galley están compuestos de uno o más subarchivos (ver Figura 3.10). Cada subarchivo reside sobre un solo disco en un servidor de E/S y contiene una o más ramificaciones llamadas *forks*. Cada ramificación es una secuencia lineal de bytes, similar a un archivo tradicional de UNIX. Al crear un archivo, se define el número de subarchivos. En un subarchivo el número de ramificaciones no es fijo y tienen un tamaño diferente. La estructura final del archivo es de tres dimensiones: subarchivos, ramificaciones y datos. En Galley la estructura paralela del archivo es ocultada a la aplicación. Galley presenta una interfaz particular la cual permite definir particiones simples, particiones anidadas y operaciones anidadas por lotes. Con estas particiones el usario puede definir diferentes distribuciones y formas de acceso de un archivo. Sin embargo, el usuario no puede utilizar vistas lógicas de

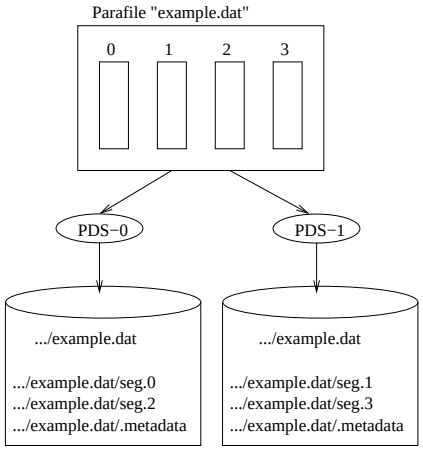

Figura 3.8: Distribución de datos de un archivo en PIOUS.

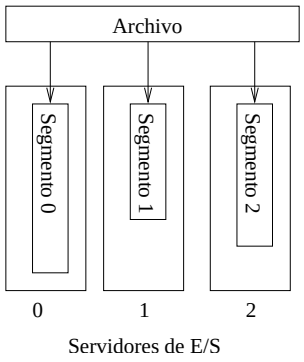

Figura 3.9: Particionamiento en PPFS.

un archivo, tal que el programador debe calcular los índices de los accesos.

Galley está basado en el modelo cliente-servidor, donde los nodos de cómputo son los clientes y los nodos de E/S son los servidores (independientes de los nodos de cómputo). Un cliente es un programa de la aplicación que es ligado con una biblioteca. La biblioteca permite realizar las solicitudes de E/S y pasarlas (como mensajes) directamente hacia los servidores. Los servidores de E/S están compuestos de varias unidades independientes. Cada unidad es implementada como un solo *thread*. Además, cada servidor tiene un *thread* asignado a cada cliente para manejar las solicitudes del cliente. Cuando un servidor recibe una solicitud desde un cliente, el *thread* del cliente apropiado interpreta la solicitud, la pasa al

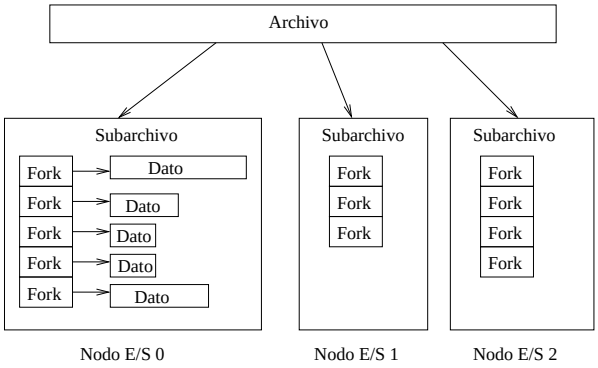

Figura 3.10: Distribución de datos en galley de un archivo de tres dimensiones.

apropiado *thread* trabajador y entonces se maneja la transferencia de datos entre el servidor y el cliente. Esta forma de *múltiples threads* permite atender múltiples solicitudes provenientes desde muchos clientes en forma simultánea. Sin embargo, este diseño puede limitar la escalabilidad del sistema. En cada nodo también se tiene un manejador de la caché que atiende las solicitudes de los bloques de datos, el cual busca en una lista LRU (del inglés: Least Recently Used Page) los bloques residentes en caché. Sin embargo, los manejadores de la caché no realizan una caché cooperativa entre los servidores de E/S para realizar un control de la concurrencia.

- **DPFS** (Distributed Parallel File System): DPFS provee un mecanismo de particionamiento que divide los archivos dentro de piezas pequeñas y las distribuye a través de múltiples dispositivos de almacenamiento para acceso paralelo a datos (ver Figura 3.11). Provee tres niveles de un archivo que pueden hacer el acceso del archivo más eficiente. Cada nivel corresponde a un método de particionamiento del archivo. También provee un método de particionamiento multidimensional que puede resolver problemas eficientes de particionamiento lineal para muchos patrones de acceso utilizados en muchas aplicaciones. DPFS presenta una arquitectura cliente-servidor. Los clientes (nodos de cómputo) trasmiten las solicitudes a los servidores (nodos de E/S) cuando necesitan ejecutar una entrada o salida. El servidor es el responsable de mandar los datos solicitados al cliente o almacenar los datos del cliente en el dispositivo de almacenamiento local. DPFS es constru-

ido sobre el sistema de archivos local sin cambiar el software del sistema de bajo nivel y así toma las ventajas de las técnicas de optimización tales como *caching* y *prefetching* del sistema de archivos local. Múltiples solicitudes pueden ser atendidas concurrentemente creando múltiples procesos o *threads*. Sin embargo, DPFS no utiliza un *caching* cooperativo con las cachés de todos los nodos de E/S.

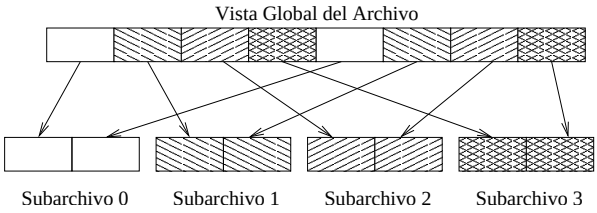

Figura 3.11: Particionamiento en DPFS.

3.3.3. Sistemas de archivos para clusters

El diseño de un sistema de archivos paralelo para *clusters* presenta varios de los mismos problemas de un sistema de archivos con memoria distribuida, debido a que ambos tipos de sistemas de archivos consisten de múltiples nodos que intercambian datos a través de una red. Estos problemas incluyen el control de la consistencia y la selección entre el *buffer* (o la caché) del cliente o del servidor. Además, los sistemas de archivos para *clusters* deben manejar la heterogeneidad en varios niveles, como son: 1) *clusters* individuales difieren uno del otro; 2) el hardware dentro de un *cluster* puede ser heterogéneo; y 3) la configuración de un *cluster* puede cambiar aún mientras el programa esté ejecutándose. El sistema de archivos de un *cluster* maneja los dos primeros niveles de heterogeneidad utilizando interfaces de programación estándard e implementan los servicios de archivo utilizando el sistema de archivos local que se ejecuta en cada nodo. En el tercer nivel de heterogeneidad, cuando la configuración del *cluster* cambia el manejo de archivos es más difícil porque tienen que adaptarse a los cambios del medio ambiente. Algunos sistemas de archivos para *cluster* tratan de adaptarse a los cambios y otros suponen que la configuración del *cluster* permanece estable.

Para diseñar un sistema de archivos para *clusters* particularmente deben considerarse los siguientes aspectos:

- *Utilización de nodos de E/S no dedicados*: al distribuir las funciones de manejo de los archivos en los nodos de cómputo.
- *Almacenamiento intermedio* (buffer en el cliente): utilizar una caché en el cliente y manejar el paso de una señal (o *token*) para el control de la concurrencia.
- *Replicación de datos*: al utilizar técnicas de los RAIDs para evitar que los archivos sean corrompidos.
- *Distribución de datos*: selección de los nodos de E/S para distribuir los datos, los nodos pueden actuar como nodos de cómputo y como nodos de E/S. Particionando los datos en muchos nodos puede requerir un movimiento excesivo de datos a través de la red.

Sistemas de archivo paralelos tales como xFS [3, 106, 5], GPFS [90], Clusterfile [47] y PVFS [20, 57, 64] han sido desarrollados para *clusters*. A continuación damos una breve descripción de estos sistemas de archivos:

xFS

xFS fue desarrollado en Berkeley por su proyecto NOW(Network of Workstations) [31, 6]. xFS divide los archivos en grupos de *stripes* y cada grupo de *stripes* forma una unidad RAID [105] controlada en software. No hay servidores separados; el software para el manejo de los archivos se ejecuta en todos los nodos de cómputo y cada archivo es manejado por un nodo *manager*. Las tareas del *manager* incluyen localizar la ubicación del bloque del archivo sobre los discos y las copias de las caches. El sistema utiliza un *token* o señal para mantener la consistencia de la caché. Sin embargo, debido a que xFS maneja un bloqueo de archivos a nivel de bloques, el *false sharing* es un gran problema.

GPFS

GPFS [90] es un sistema de archivos con discos compartidos para *clusters* de Linux (también disponible en la Supercomputadora Paralela RS/6000 SP). Un sistema GPFS consiste de un *cluster* de nodos conectados a los discos o el sistema de discos sobre una estructura de switcheo (ver Figura 3.12). GPFS soporta un tamaño de archivo de 64

bits permitiendo un tamaño máximo de un archivo de $2^{63} - 1$ bytes. Un archivo grande es particionado en bloques de igual tamaño y bloques consecutivos son colocados en diferentes discos en forma de *round-robin*. GPFS maneja bloques de 256K para minimizar la sobrecarga debida a la búsqueda en el disco (*seek*), pero puede ser configurado entre 16K y 1M byte.

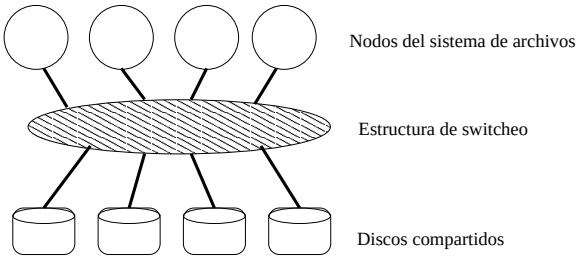

Figura 3.12: Discos compartidos en GPFS.

En GPFS todos los nodos en el *cluster* tienen un mismo acceso a todos los discos. Los accesos a disco lecturas/escrituras paralelas desde múltiples nodos en el *cluster* deben ser sincronizadas, o los datos del usuario y los metadatos del sistema de archivo serán dañados. GPFS utiliza un bloqueo distribuido para sincronizar los accesos a los discos compartidos. Los protocolos empleados en el bloqueo distribuido garantizan la consistencia del sistema de archivo sin importar el número de nodos que simultáneamente leen desde y escriben a el sistema de archivo, mientras que al mismo tiempo se permite el paralelismo necesario para realizar la máxima transferencia de datos en menor tiempo. Sin embargo, en GPFS no se maneja un *caching* cooperativo entre los diferentes nodos del sistema de archivos.

Clusterfile

Clusterfile es un sistema de archivos paralelo que permite un acceso paralelo a archivos en un *cluster* de computadoras. Un archivo en Clusterfile es una secuencia lineal de bytes. El archivo es físicamente particionado dentro de uno o más subarchivos (ver Figura 3.13). El particionamiento es descrito por un desplazamiento del archivo y un patrón de particionamiento. El desplazamiento es una posición absoluta de bytes a partir del inicio del archivo. El patrón de particionamiento consiste de la unión de n

conjuntos de segmentos llamados FALLS (del inglés: FAmiliy of Line Segments). Cada FALLS define un subarchivo. Los conjuntos deben describir regiones del archivo que no se traslapen y deben ser regiones contiguas.

Las operaciones de lectura y escritura son optimizadas por un pre-cálculo del mapeo directo entre los patrones de acceso y los discos. Clusterfile utiliza la misma representación de datos para el diseño del archivo, los patrones de acceso y los mapeos entre cada uno.

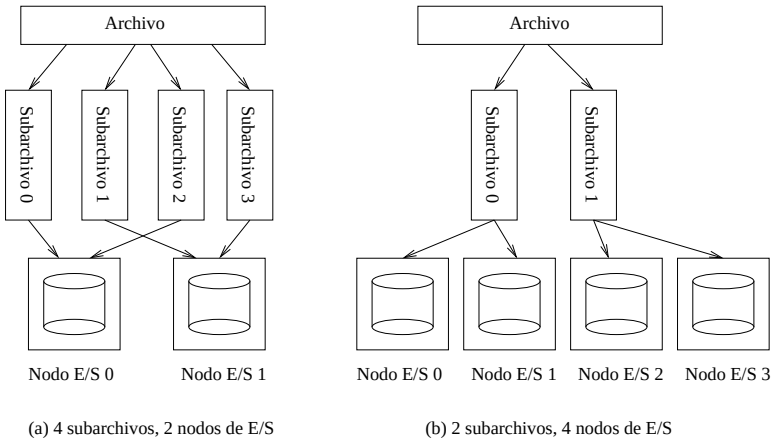

Figura 3.13: Particionamiento en Clusterfile.

PVFS

PVFS (Parallel Virtual File System) es un sistema de archivos paralelo de alto rendimiento para *clusters* de Linux desarrollado por la Universidad Clemson. PVFS es diseñado como un sistema cliente-servidor, utiliza procesos servidores de E/S que pueden ejecutarse sobre múltiples nodos de E/S. Además, el sistema también permite ejecutar al software del servidor de E/S sobre los nodos de cómputo.

El modelo de particionamiento de un archivo es similar a los subarchivos de Vesta. Un archivo en PVFS es particionado en *stripes*. Cada *stripe* consiste de un número fijo de fragmentos (o bloques) los cuales consisten en un número fijo de bytes. Cada fragmento es almacenado en un solo nodo de E/S y son asignados en forma de *round robin* (ver Figura 3.14). Actualmente, la distribución de un archivo en PVFS está dada por tres parámetros: el número base de los nodos de E/S, el número de nodos de E/S y el tamaño

del *stripe*; no utiliza el valor del fragmento. Estos parámetros junto con un ordenamiento de los nodos de E/S especifican completamente la distribución del archivo.También se utiliza un desplazamiento u *offset del stripe*, longitud en bytes desde el inicio del archivo hasta donde inicia el primer *stripe*. El patrón de particionamiento de cada archivo puede ser especificado por las aplicaciones.

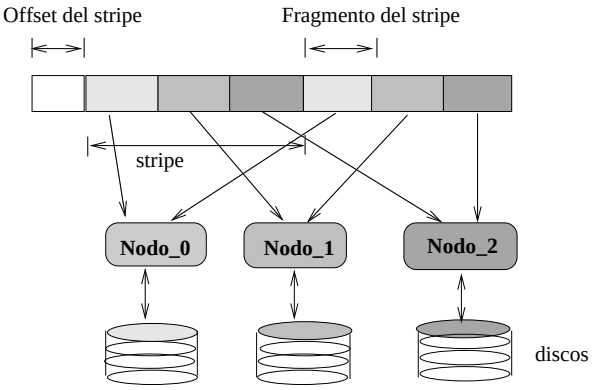

Figura 3.14: Particionamiento en PVFS.

PVFS ofrece varias interfaces que le permiten a los procesos definir sus propios conjuntos no entrelazados de un archivo. PVFS no maneja en forma directa los bloques de datos de archivos; en su lugar, confía en el sistema de archivos nativo local en cada nodo de E/S para manejar esta tarea. Sin embargo, PVFS no maneja un *buffering* (o caché) directamente, puede haber un *buffering* bajo el sistema de archivos; sus efectos son similares al *buffering* del servidor. Además, el usuario debe controlar el acceso concurrente en las escrituras para garantizar la coherencia de los datos, haciendo la programación más compleja. La concurrencia se controla mediante el particionamiento del archivo cuando los procesos acceden a distintas regiones de un bloque de datos (no-entrelazados).

Una lista detallada de sistemas de archivos con las características que soportan es mostrada en la Tabla 3.2 (para mayores detalles ver [95]).

Tabla 3.2: Sistemas de archivos paralelos.

Nombre	Modelo de memoria	Sincrono/asincrono	Caching	prefetching	Operaciones colectivas	Control de concurrencia	Apuntador de archivo compartido	Datos paralelos	Nuevas Ideas
Vesta	D	+	+	−	+	+	−	+	
PIOUS	D	+	−	−	−	+	+	+	E/S para ambientes de computo con PVM
PPFS	D	+	+	−	+	+	−		Con control en tiempo real y politica de control adaptivo
Galley	D	+	+		−	−	−	−	Estructura 3d de un archivo
DPFS	D	−	+	−	+	−	−		Tres tipos de particionamiento
xFS	D								
GPFS	D	+	−	+	+	+		+	
Clusterfile	D								
PVFS	D	+	−	−	+	−	−	+	Sistema de archivo paralelo para clusters de Linux

Notacion: + ... es implementado D ... memoria distribuida
− ... no es implementado S ... memoria compartida

3.4. Ejemplo de E/S paralela con MPI-IO y con PVFS

En esta sección presentamos dos ejemplos de aplicaciones con E/S paralela utilizando MPI-IO y PVFS. Utilizamos estos sistemas de E/S porque son utilizados en MPI y MPI es el ambiente más utilizado en la actualidad para desarrollar aplicaciones paralelas en PC-clusters.

Cuando desarrollamos una aplicación paralela *out-of-core*, primero debemos especificar el particionamiento de los archivos y después especificamos la forma de accederlos en el programa de la aplicación. La primer aplicación nos muestra el particionamiento y el acceso de un archivo de datos utilizando MPI-IO. La segunda aplicación presenta como se controla el acceso de un bloque para mantener su coherencia cuando dos o más procesos acceden al bloque y uno de ellos lo modifica. Para visualizar como se mantiene la coherencia de los bloques del archivo analizamos la bilioteca de E/S MPI-IO y el sistema de archivos PVFS.

3.4.1. MPI-IO

Primer ejemplo: particionamiento y acceso de un archivo con MPI-IO

En el primer ejemplo mostrado en la Figura 3.15 se presenta el particionamiento de datos de un archivo en MPI-IO. Consiste de cuatro pasos: El primer paso consiste en definir el tamaño del elemento básico o *newtype* del archivo. Este elemento es usado para construir los patrones necesitados en los siguientes pasos. En el segundo paso, se especifica el mapeo o vista lógica al construir el patrón de acceso del archivo. El patrón define el subarchivo que va a accederse en cada operación de lectura/escritura. Con este concepto de subarchivo, el usuario especifica que bloques del archivo van a ser usados por cada proceso de la aplicación. Cuando el subarchivo ha sido especificado, cada operación de lectura y escritura puede especificar un patrón diciéndole al sistema dónde colocar los elementos accedidos desde el archivo. El tercer paso se realiza al abrir el archivo, especifica que partes del archivo completo serán usadas para construir el subarchivo abierto. Finalmente, en el cuarto paso, el usuario define como serían almacenados en un *buffer* los elementos accedidos. Este cuarto paso es realizado en cada operación de lectura/escritura.

La Figura 3.15 muestra en código de lenguaje C los puntos principales involucrados en la definición y uso de una vista lógica de los datos utilizando MPI-IO. Para una matriz $N \times N$ y p procesadores, la vista lógica especifica que el procesador 0 utiliza sólo los primeros N/p elementos en cada fila, el procesador 1 utiliza sólo los segundos N/p elementos en cada fila y así sucesivamente. (Esta vista es útil para implementar un algoritmo de multiplicación de matrices, $C = A \times B$, donde cada procesador calcula un valor parcial de cada elemento en la matriz C; antes de leer A, cada procesador habrá leído N/p filas de la matriz B; ver la Sección 6.3.2.)

Una vista lógica es definida usando cuatro arreglos (de tipo *int*, líneas 4-11) : *array_of_gsizes* contiene el tamaño de cada dimensión en el arreglo sobre el cual la vista lógica es definida; *array_of_distribs* expresa si una dimensión es ditribuida y como (NONE, BLOCK, or CYCLIC); *array_of_dargs* especifica el tamaño de la unidad de distribución (BLOCK puede ser el default; CYCLIC requiere el tamaño de la unidad) y *array_of_psizes* especifica el número de elementos a distribuir a cada procesador. Cuando la vista lógica es creada (líneas 13-16), un nuevo tipo (*newtype*) es definido con esta vista. También el *buffer* para almacenar los datos es creado (línea 17). De esta manera,

```
1  MPI_Datatype newtype;                                  /* La nueva vista lógica */
2  int ndims=2;
3
4  array_of_gsizes[0]   = N;                              /* tamaño de cada dimensión*/
5  array_of_gsizes[1]   = N;
6  array_of_distribs[0] = MPI_DISTRIBUTE_BLOCK;           /* divide las filas por bloques */
7  array_of_distribs[1] = MPI_DISTRIBUTE_NONE;            /* no dividir las columnas */
8  array_of_dargs[0]    = MPI_DISTRIBUTE_DFLT_DARG;       /* bloque = rowsize/processors */
9  array_of_dargs[1]    = MPI_DISTRIBUTE_DFLT_DARG;       /* no aplicable */
10 array_of_psizes[0]   = 0;                               /* calcular rowsize/processors */
11 array_of_psizes[1]   = 1;                               /* not calcular */
12 MPI_Dims_create( nprocs, ndims, array_of_psizes);       /* dividir rows/nprocs */
13 MPI_Type_create_darray(nprocs, myrank, ndims,          /* definir la vista */
14                       array_of_gsizes, array_of_distribs, array_of_dargs,
15                       array_of_psizes, MPI_ORDER_C, MATRIX_MPI_TYPE, &newtype);
16 MPI_Type_commit( &newtype );                           /* manejar la vista lógica */
17 MPI_Type_size( newtype, &bufcount );
18
19 MPI_File_open( MPI_COMM_WORLD, ..., &f );
20 MPI_File_set_view( f, 0, MATRIX_MPI_TYPE, newtype, "native", MPI_Info );  /* usar la vista en un archivo */
21 MPI_File_read_all( f, readbuf, N, MATRIX_MPI_TYPE, &status );
22 MPI_File_close( &f );
```

Figura 3.15: Vista lógica para un archivo usando MPI-IO.

la vista es unida a un archivo (línea 20) y posteriormente usada para acceder al archivo (línea 21).

Segundo ejemplo: control de la concurrencia con MPI-IO

En la Figura 3.16 se presenta el segundo ejemplo utilizando MPI-IO. En este ejemplo dos procesos abren un mismo archivo llamado *sample* utilizando la función colectiva `MPI_File_open` (línea 2). Ambos procesos acceden un mismo bloque del archivo, el proceso 0 para escribir nuevos datos del bloque y el proceso 1 para leer el nuevo bloque de datos desde el archivo. Si ambos procesos acceden el archivo al mismo tiempo utilizando las funciones `MPI_File_write_at` por el proceso 0 y `MPI_File_read_at` por el proceso 1, no puede garantizarse que los datos leídos por el proceso 1 sean los correctos (coherentes).

Para garantizar la coherencia del bloque de datos del archivo, la escritura del bloque debe ser directa a disco o al nodo de E/S. (Las bibliotecas de E/S como MPI-IO[70, 66, 100] utilizan funciones que permiten realizar accesos directos al disco). Entonces los dos procesos deben sincronizarse; y si otro proceso desea leer ese bloque, debe hacerlo

desde disco o desde el nodo de E/S. La técnica estándar es una secuencia de *"sync-barrier-sync"*, como es mostrado en el programa de la Figura 3.16, donde el proceso 0 es el escritor y el proceso 1 es el lector.

```
1 /* El siguiente código se ejecuta en el Proceso 0 */
2 MPI_File_open (MPI_COMM_WORLD, 'sample', ..., fh, ...);
3 ...
4 MPI_File_write_at(fh, 0, buf, 100, MPI_BYTE, &status, ...);
5 MPI_File_sync(fh,err);
6 MPI_Barrier(MPI_COMM_WORLD);
7 MPI_File_sync(fh,err);
8 ...
9 /* El dato ahora es disponible al Proceso 1; */
10 /* El Proceso 0 puede continuar usando el archivo */
11 MPI_File_close(fh,err);
```

(a)Proceso 0

```
/* El siguiente código se ejecuta en el Proceso 1 */
MPI_File_open(MPI_COMM_WORLD, 'sample' , ..., fh, ...);
...

MPI_File_sync(fh,err);
MPI_Barrier(MPI_COMM_WORLD);
MPI_File_sync(fh,err);
MPI_File_read_at(fh, 0, buf, 100, MPI_BYTE, &status, ...);
...

MPI_File_close(fh,err);
```

(b) Proceso 1

Figura 3.16: Escritura y lectura desde un archivo por dos procesos (el código está simplificado en cuanto al número de parámetros de las funciones MPI utilizadas).

La primera llamada a `MPI_File_sync` (línea 5) provoca el almacenamiento en disco de cualquier dato que el escritor escribió en su memoria. La segunda llamada a `MPI_File_sync` (línea 7) refresca cualquier copia en la memoria del lector; en la caché del archivo de datos del lector. `MPI_Barrier` (línea 6) asegura que la lectura del proceso lector ocurra después de la escritura del dato por el escritor. Podríamos desear quitar la segunda sincronización en el escritor y la primera en el lector, ya que ninguna parece necesaria. Sin embargo, `MPI_File_sync` es una llamada colectiva y todos los nodos deben participar (de hecho, `MPI_File_sync` puede actuar como una barrera en algunas implementaciones). De lo contrario, se podría crear un estado de bloqueo (o *deadlock*) con la operación de la barrera.

La Figura 3.17 muestra las secuencias de las transferencias de datos entre los nodos de este ejemplo. El caso de E/S mostrado en esta figura es el más simple, en el cual sólo se tienen dos nodos y el archivo está almacenado en el disco de uno de ellos. Pero los archivos podrían estar almacenados en forma distribuida a través de varios nodos. En este caso, los almacenamientos provocados por la función de sincronización (`MPI_File_sync`) serían remotos, hacia el disco del nodo que corresponda a cada bloque. Así se tendrá una escritura remota adicional además de la lectura remota provocada por la segunda llamada de sincronización (`MPI_File_sync`).

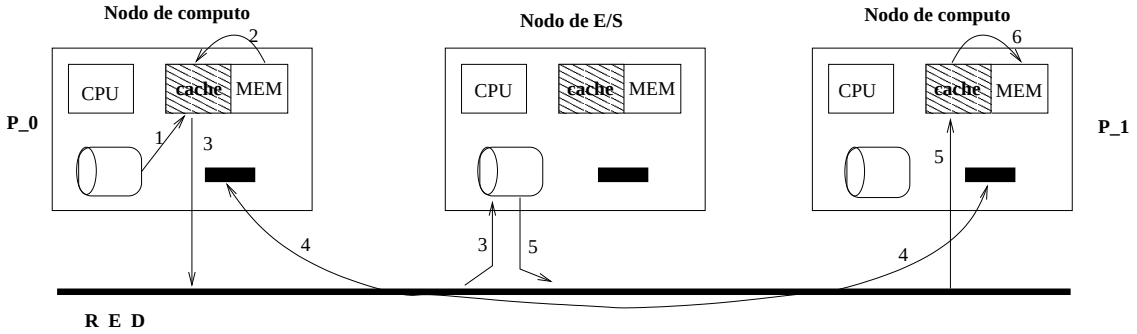

Figura 3.17: Escritura y lectura con MPI-IO.

3.4.2. PVFS

Segundo ejemplo: control de la concurrencia con PVFS

En la Figura 3.18 se muestra el código en lenguaje C de dos procesos que acceden a un archivo de PVFS utilizando el API de PVFS (ejemplo similar al de la Figura 3.16). En este ejemplo primero se define un patrón de distribución del archivo utilizando la estructura `pvfs_filestat`. Los parámetros de esta estructura especifican que el archivo es creado en 2 nodos de E/S con un tamaño del *stripe* de 16 Kbytes, siendo el nodo 0 el nodo base en donde se almacena el archivo. Después, dos procesos abren el archivo llamado *sample* utilizando la función `pvfs_open`. Ambos procesos acceden a un mismo bloque del archivo, el proceso 0 para escribir nuevos datos del bloque y el proceso 1 para leer el nuevo bloque de datos desde el archivo. Si ambos procesos acceden el archivo al mismo tiempo utilizando las funciones `pvfs_write_at` por el proceso 0 y `pvfs_read_at` por el proceso 1, no puede garantizarse que los datos leídos por el proceso 1 sean coherentes.

Para mantener la coherencia, PVFS no tiene funciones de control de la concurrencia para escribir y leer a un mismo bloque a la vez. El usuario debe utilizar barreras u otro tipo de control con la ayuda de bibliotecas como MPI. En el código de la Figura 3.18, el proceso 0 escribe sus resultados al archivo y el proceso 1 los lee. Cuando el archivo ya ha sido abierto, en cada proceso se crea un *buffer* local de 256 Kbytes para optimizar los accesos. Los datos leídos desde el disco son almacenados dentro de este *buffer* y desde ahí son accedidos en posteriores accesos. Para sincronizar los accesos es necesario utilizar una barrera (con `MPI_Barrier`) antes de la operación de lectura del proceso 1 y

```
/* Especificar el patrón de distribución */
struct pvfs_filestat pstat = 0, 2, 16384, 0, 0;
FILE *fp;

fp = pvfs_open ('sample', O_RDWR O_CREAT, 0666,
&pstat, NULL);
setvbuf(fp, NULL, _IOFBF, 256*1024);
/* Ahora tenemos un buffer de 256K y la E/S es
completamente usada con buffering */
...
pvfs_write_at(fp, buf, 100 );
MPI_Barrier(MPI_COMM_WORLD);
...
/* El dato ahora es disponible al Proceso 1; */
/* El Proceso 0 puede continuar usando el archivo */
pvfs_close(fp);
```

(a)Proceso 0

```
/* Especificar el patrón de distribución */
struct pvfs_filestat pstat = 0, 2, 16384, 0, 0;
FILE *fp;

pvfs_open('sample', , 0, &pstat, NULL);
setvbuf(fp, NULL, _IOFBF, 256*1024);
/* Ahora tenemos un buffer de 256K y la E/S es
completamente usada con buffering */

...
MPI_Barrier(MPI_COMM_WORLD);
pvfs_read_at(fp, buf, 100 );
...

pvfs_close(fp);
```

(b) Proceso 1

Figura 3.18: Escritura y lectura desde un archivo por dos procesos en PVFS.

después de la operación de escritura del proceso 0. En PVFS, cada vez que se realiza una operación de lectura, los datos viajan del nodo de E/S al nodo de cómputo y cuando se tiene una operación de escritura los datos viajan en sentido inverso. Este movimiento de datos en PVFS garantiza la coherencia, pero genera más mensajes a través de la red que podrían degradar el rendimiento de la aplicación paralela y la programación puede complicarse cuando más procesos realizan diferentes operaciones a un mismo bloque.

3.5. Resumen

Los PC-clusters son sistemas de cómputo apropiados para ejecutar aplicaciones *out-of-core* debido a que normalmente cada uno de sus nodos cuenta con disco y memoria. En un PC-cluster la capacidad de almacenamiento en disco y memoria es proporcional al número de nodos y agrupando en paralelo los nodos que tienen disco se han desarrollado sistemas con una E/S paralela.

La E/S paralela se desarrolló con el fin de disminuir el tiempo de acceso a los datos que están almacenados en archivos para mejorar el desempeño de aplicaciones *out-of-core* y facilitar su programación. Mediante la E/S paralela, los archivos son particionados y distruibudos en los discos de varios nodos para permitir un acceso paralelo y simultáneo de varios procesos. Para desarrollar la E/S paralela varias caracterísitcas se consideran en su diseño, como son: el *caching*, el *prefetching*, las operaciones colectivas, el control

de la concurrencia, apuntadores de archivo compartido y el acceso paralelo, entre otras. La finalidad de estas características es optimizar el tiempo de acceso de los dispositivos de E/S (o discos), disminuir el número de operaciones de lectura y escritura a disco y disminuir el número de mensajes entre procesos.

Para manejar la E/S paralela, dos alternativas han sido consideradas: las bibliotecas de E/S y los sistemas de archivos paralelos.

Las bibliotecas de E/S realzan los lenguajes de alto rendimiento con características especiales de E/S a disco y facilitan la programación de aplicaciones paralelas con E/S. Estas bibliotecas integran nuevas semánticas al sistema de archivos o son construidas para ofrecer nuevas semánticas al usuario teniendo abajo de éstas las semánticas del sistema de archivos nativo del sistema operativo. Las bibliotecas típicamente son flexibles y portables debido a que no dependen de ningún sistema de archivos en particular y la programación de aplicaciones de E/S se facilita. Sin embargo, cuando se realiza de manera simultánea el procesamiento y las solicitudes de E/S puede ser muy complicado explotar el paralelismo. Esta limitación es debida a que las bibliotecas no cuentan con un servidor de E/S que atienda de manera independiente las solicitudes de E/S.

Los sistemas de archivos paralelos son diseñados para proveer muy alto rendimiento de la E/S cuando son accedidos por muchos procesos a la vez. Estos procesos son distribuidos a través de todos los nodos de un PC-cluster. Los nodos en un PC-cluster pueden operar como un nodo de cómputo, como uno de E/S o como ambos, según sea el caso. Para alcanzar un alto rendimiento, un sistema de archivos paralelo normalmente particiona los archivos a través de los nodos similar a un RAID. Dividir los datos a través de los nodos es una forma de ganar paralelismo por medio de sistemas de E/S secuencial. Debido a que los archivos son divididos en esta manera y los programas paralelos tienden a trabajar sobre regiones específicas de un archivo compartido, las cargas en la red y los discos pueden ser difundidas a los nodos de almacenamiento. Cuando se ejecuta una aplicación paralela que accede los datos de un archivo, en una lectura los datos viajan desde los nodos de E/S hacia los nodos de cómputo que los procesará y en una escritura en sentido contrario. Si más de un proceso accede un dato a la vez, podrá existir un problema de coherencia si alguno de los procesos modifica el dato. Así el resultado de una aplicación puede ser incorrecto si los datos son inconsistentes.

Para mantener la coherencia, los accesos deben ser directos a disco y los procesos

deben sincronizarse cuando se tienen accesos simultáneos de un mismo bloque. El proceso que modifica el dato debe bloquear a los demás procesos para prevenir sus accesos. Así los datos almacenados en la caché local del nodo siempre serán los últimos modificados. Esta técnica de programación se complica cuando el número de nodos crece disminuyendo la escalabilidad del sistema y puede generar un bajo rendimiento en la ejecución de la aplicación. El usuario debe implementar manualmente un protocolo de coherencia "múltiples lectores un solo escritor".

.

Capítulo 4

El Espacio de difusión de datos

En los dos capítulos anteriores hemos presentado algunos de los sistemas de software que pueden utilizarse en el desarrollo y ejecución de aplicaciones paralelas en *clusters*. En este capítulo presentamos nuestro diseño para una memoria compartida distribuida implementada totalmente en software. Su propósito es simplificar el desarrollo de aplicaciones paralelas *in-core* y *out-of-core* utilizando una misma programación para ambos tipos de aplicaciones y adicionalmente mejorar su tiempo de ejecución respecto a otras herramientas de cómputo paralelo.

4.1. Motivación y nuestra propuesta

Para desarrollar una aplicación paralela en PC-clusters, el programador puede utilizar bibliotecas de paso de mensajes o bibliotecas para memoria compartida distribuida. La programación y ejecución de aplicaciones paralelas utilizando bibliotecas de paso de mensajes en *clusters* es fácilmente escalable (si el algoritmo también es escalable) y portable debido a que los programas implementados con estas bibliotecas no dependen de una arquitectura en concreto. Además proporcionan una buena relación costo/desempeño en la ejecución de las aplicaciones. Sin embargo, en el paso de mensajes, el programador normalmente debe especificar la comunicación entre procesos utilizando mensajes. En cada mensaje el programador normalmente debe especificar varios parámetros al compartir los datos entre diferentes procesos, tales como el tamaño y tipo de los datos, así como la identificación de los procesos que intervienen en la transmisión.

En general, el programador necesita considerar varios aspectos: debe saber *dónde* está el dato, *cuándo* deben comunicarse los procesos, con *quién* comunicarse y *qué* dato transmitir. El uso de estas bibliotecas no es una tarea fácil y tiende a complicarse aún más cuando los patrones de acceso a datos son irregulares o dinámicos con estructuras de datos complejas.

Por otro lado, la programación de aplicaciones *out-of-core* con paso de mensajes en *clusters* es diferente de la programación de aplicaciones *in-core* y se hace más compleja. Se requieren dos versiones, en la versión *in-core*, los procesos intercambian datos con paso de mensajes. En la versión *out-of-core*, los procesos leen y escriben los datos directamente a archivos en disco. El acceso a datos desde los archivos incrementa el tiempo de su acceso y en consecuencia incrementa el tiempo de ejecución de estas aplicaciones. Con el fin de mejorar el tiempo de acceso a datos desde archivos, los programadores han utilizado sistemas de archivos paralelos los cuales particionan los archivos en bloques y los distribuyen en los discos de diferentes nodos. Estos sistemas de archivos permiten realizar accesos colectivos y paralelos a los archivos, lo que tiende a disminuir el tiempo de acceso de un bloque de datos. Sin embargo, la distribución de los datos en los archivos es fija durante toda la ejecución de la aplicación y en muchas ocasiones no corresponde con los patrones de acceso de la aplicación. Por lo que, los datos difícilmente están en la memoria de los procesadores que los utilizan y puede producirse un mal desempeño si no se eligen los parámetros más adecuados en la distribución de los datos dentro del sistema paralelo. Idealmente, el usuario no debería preocuparse por la localización de los datos y accederlos como si estuvieran almacenados en la memoria local del nodo.

Una memoria compartida distribuida (DSM, por sus siglas en inglés) simplifica la programación paralela debido a que el programador no necesita conocer la localización de los datos. Los datos en una DSM son compartidos por todos los procesos que ejecutan una aplicación paralela. Los datos compartidos se mueven automáticamente entre los nodos de procesamiento de acuerdo a los patrones de acceso de cada aplicación y no es necesario utilizar mensajes entre procesos. Para mover los datos y controlar su accceso la mayoría de los diseños de DSM requieren del soporte del hardware y/o del software, el cual es fácilmente disponible en la mayoría de las plataformas y sistemas operativos. Si una DSM soporta mapeo de archivos dentro de la memoria compartida, el cómputo *out-of-core* es tan simple de programar como el cómputo *in-core*. Esto será más útil en

arquitecturas de 64 bits, ya que en las arquitecturas de 32-bit sólo 4 GB son disponibles, mientras que en algunas aplicaciones *out-of-core* actualmente manejan rangos del orden de cientos de GBs. Por otro lado, en arquitecturas heterogéneas con procesadores de 32 y 64 bits la programación se complica al utilizar una DSM ya que la mayoría de las DSM dependen del hardware o software utilizado. Por lo que el programador tendría que modificar el código de la DSM para ajustar las diferencias entre estas arquitecturas, como por ejemplo el tamaño de página de la memoria caché.

La DSM que se describe en este capítulo es un *Espacio de Difusión de Datos* (`DDS`, del inglés Data Diffusion Space). El espacio de difusión de datos es un sistema de difusión automática de datos entre los procesadores que ejecutan una aplicación paralela en un *cluster*, el cual es implementado todo en software. DDS es un espacio de direcciones compartido adicional al espacio de direcciones virtuales de cada proceso que ejecutan la aplicación paralela. DDS es independiente del sistema operativo y del hardware, lo cual lo hace portable para arquitecturas de 32 y 64 bits. De esta manera, DDS puede operar en un *cluster* heterogéneo utilizando ambas arquitecturas. Con la difusión automática de los datos en este espacio compartido, el usuario no se preocupa por la localización de los mismos, sólo realiza accesos de los datos como si estuvieran almacenados en la memoria local del nodo, lo cual facilita la programación y mejora su desempeño al disminuir los accesos a disco. Además, la capacidad de almacenamiento de un nodo en particular puede aumentarse al utilizar la memoria y los discos de todos los nodos de un *cluster*, manejando un rango de almacenamento de hasta 2^{64} bytes.

A continuación presentamos el espacio de difusión de datos llamado DDS, mostramos su arquitectura, su operación y su modelo de programación para desarrollar aplicaciones paralelas.

4.2. Arquitectura y operación de DDS

La Figura 4.1 muestra la arquitectura de DDS. DDS está diseñado para simplificar la programación de aplicaciones paralelas bajo el modelo SPMD. Bajo este modelo, un proceso es creado sobre cada nodo de procesamiento para ejecutar una aplicación paralela. Con DDS, el proceso DDSP también es creado y ejecutado sobre cada nodo de procesamiento. El proceso DDSP es el encargado de atender las solicitudes de acceso

a bloques de datos, de mover los bloques entre los nodos (en la memoria y el disco) y de mantener su coherencia. El espacio de difusión de datos es adicional al de cada proceso ejecutando una aplicación paralela en un PC-cluster. Para el almacenamiento de los datos en este espacio de difusión, en cada nodo se define un espacio de dirección local (EDL) o caché que es transparente al programador. El EDL es compartido por el proceso de la aplicación y el proceso DDSP. El tamaño de este espacio puede variar en cada nodo y depende de la capacidad que tenga en memoria (se utilizó la memoria virtual de cada nodo). En conjunto con EDL, cada nodo tiene un archivo temporal para almacenar en el disco los bloques que sean reemplazados de la memoria. Al inicio de DDS, este archivo está vacío y crecerá conforme los bloques sean almacenados en disco hasta la capacidad máxima del disco del nodo. DDS es sólo para datos compartidos, los cuales el programador debe especificar como tales a través de simplemente declararlos dentro de una estructura de datos en lenguaje `C`. Un dato en el espacio de difusión es mapeado dinámicamente dentro del espacio de dirección de EDL de cualquier proceso de la aplicación que esté usando el dato. Los datos colocados sobre DDS son difundidos, o migrados y replicados, automáticamente por DDSP en la memoria de cada procesador que los utiliza. Desde ahora usaremos el término *dato compartido* para referirnos al dato dentro del espacio de difusión.

Los datos se difunden bajo un protocolo de coherencia "múltiples lectores un solo escritor". Cuando un procesador desea leer un bloque, obtiene una copia del mismo. Cuando un procesador desea escribir un bloque, obtiene una copia exclusiva del mismo, para lo cual el protocolo invalida todas las demás copias en el sistema.

Dado que bajo DDS los datos se mueven automáticamente, el usuario no debe especificar su localización para accederlos. No es necesario especificar quién tiene un bloque ni quién lo solicita, sólo se realiza una lectura o una escritura como si dicho bloque estuviera almacenado localmente y sin importar si está en la memoria o en el disco. Así, cuando un bloque es escrito en varios nodos, el bloque migra a cada uno de ellos. Es decir, la ubicación de un bloque no es fija; no es determinada al inicio de la ejecución de una aplicación a través de, por ejemplo, una función de particionamiento.

El protocolo de DDS es similar al utilizado por COMA-F (siglas del inglés, Cache-only Memory Architecture-Flat), una arquitecura de memoria compartida distribuida implementada totalmente en hardware [48]. COMA-F está basado en directorios y utiliza

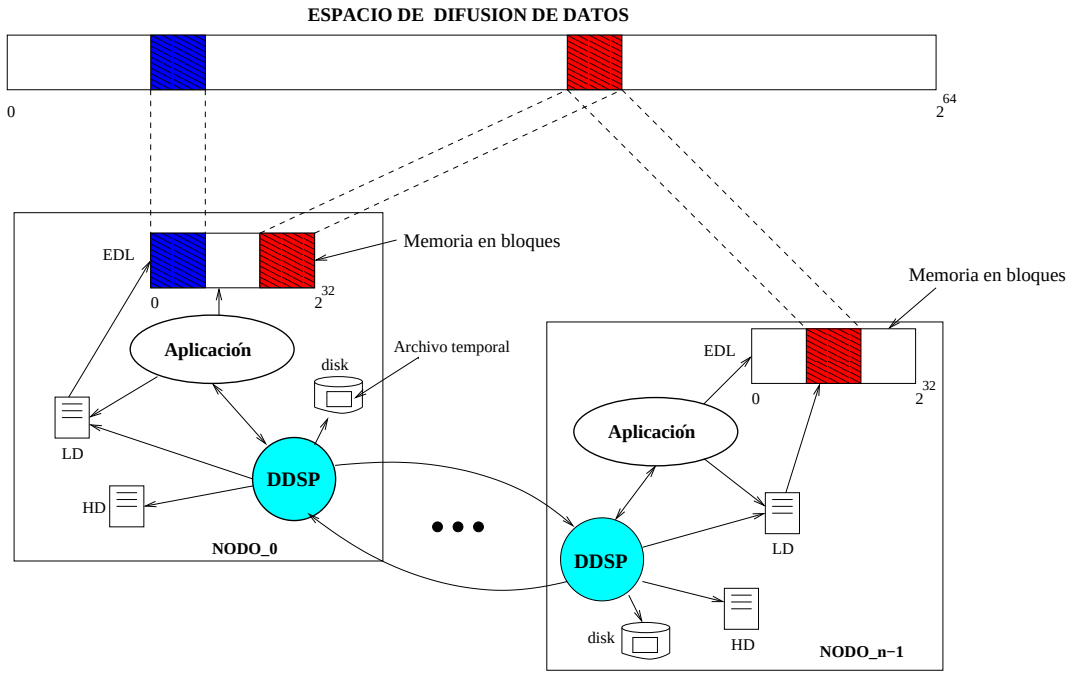

Figura 4.1: Arquitectura de DDS.

una memoria principal asociativa similar a la utilizada por memoria caché. El dato no tiene una posición fija o *home*, se mueve hacia la memoria de los procesadores que lo acceden y permanece ahí hasta que es invalidado por la escritura de un procesador o hasta que es desalojado para dar el lugar a otro dato usado más recientemente. Cuando una lectura o una escritura falla dentro de una memoria de un nodo, una solicitud es transmitida al *directorio home* del elemento del dato. Este directorio mantiene la localización (del nodo) y la información del estado del elemento (maestro, exclusivo, compartido). Si la localización del elemento es la misma que el nodo del directorio *home*, este mismo nodo sirve la solicitud; de otra manera transmite la solicitud al nodo que actualmente tiene el elemento. Un directorio *home* es manejado en cada nodo y algunos bits de la dirección de cada elemento son usados para identificar un directorio *home*.

DDS utiliza dos directorios en cada nodo (ver Figura 4.1): un *directorio local* (LD) y un *directorio home* (HD). Estos directorios tienen el propósito de facilitar la localización de los bloques de datos y mejorar el desempeño de una aplicación. El *directorio local* juega el papel del directorio de una memoria asociativa. La dirección de un elemento del

dato es obtenida desde el directorio local si el correspondiente elemento del dato está en la memoria. Sin embargo, nuestra organización del directorio local no sólo contiene información del dato cuando está almacenado dentro de la memoria de un nodo, sino también cuando está dentro del espacio del disco del nodo. Es decir, en DDS tenemos dos niveles de la caché: memoria y disco (COMA-F sólo maneja un nivel de la memoria caché). De esta manera, este directorio contiene la dirección de la memoria o la localización dentro de un archivo de reemplazo en donde están almacenadas las copias de los bloques que actualmente están en el nodo. El directorio es utilizado por el proceso de la aplicación y el proceso DDSP para buscar un bloque dentro del nodo. El directorio home sólo es accedido por el proceso DDSP y es utilizado para identificar al nodo que tiene una copia maestra de bloque solicitado por un nodo remoto (ver detalles en 4.3.1).

Cuando un proceso de la aplicación solicita un dato compartido, el dato primero es buscado en el directorio local. Si ninguna copia del dato está residente en la memoria local del nodo, el proceso DDSP busca en el directorio *home* quién tiene el dato y solicita el dato a un nodo remoto. Cuando el dato llega al nodo local (desde el nodo remoto), es almacenado por DDSP dentro del espacio de dirección de la aplicación e insertado en el directorio local utilizando una función *Hash* dada por la ecuación 4.1. La dirección donde el dato fué puesto es dada a la aplicación a través de la interfaz de DDS (descrita en la sección 4.6.2). Se utiliza una función *Hash* porque su eficiencia depende del factor de almacenamiento. Este factor es dado por $f = M/n$, donde n es el número de elementos y M es el tamaño de la tabla (en nuestro caso M es el tamaño del directorio). Para valores grandes de n y un valor razonable de f un buen esquema de *hashing* requiere normalmente menos pruebas (del orden de 1.5 a 2) que cualquier otro método de búsqueda. Además, en DDS se utiliza un algoritmo de *rehashing* para disminuir las colisiones cuando se insertan elementos en el directorio.

La transferencia de datos entre los nodos es realizada utilizando las primitivas de comunicación del SO, para lo cual se han utilizado a los *sockets* debido a que son soportados en la mayoría de los sistemas. Modificaciones al *kernel* de Unix no son necesarias porque las implementaciones modernas de Unix proveen todas las funciones requeridas por DDS para el manejo de las comunicaciones y de la memoria.

Hemos implementado a DDS para PC-clusters con Linux utilizando la versión Red-Hat 9.0. Sin embargo, siendo todo en software, DDS requeriría pocos cambios para

hacerlo inmediatamente usable en otras plataformas de memoria distribuida y sistemas operativos. Actualmente, DDS está organizado dentro de una biblioteca de funciones que es ligada a una aplicación paralela utilizando el lenguaje de programación C.

Los componentes del software (antes mencionados) de un nodo al desarrollar una aplicación con DDS se muestran en la Figura 4.2.

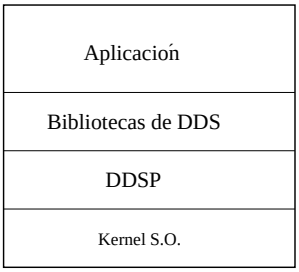

Figura 4.2: Componentes de software de un nodo bajo DDS.

4.3. Directorios

La estructura de un directorio local es un poco diferente a la estructura de un directorio *home*. Esto es porque cumplen funciones diferentes (ver Figura 4.3).

El *directorio local* de un nodo contiene la identificación, el estado y la localización (en la memoria o el disco local) de cada bloque almacenado en ese nodo.

El *directorio home* en un nodo contiene, para un subconjunto de bloques del total de bloques de una aplicación, su identificación, su estado y su localización (es decir, la identificación de los nodos que tienen una copia). El subconjunto de bloques cuya información se almacena en un nodo *home* es determinado por una función de particionamiento al iniciar la ejecución de una aplicación.

La Figura 4.3.a muestra la estructura de una entrada de los directorios locales. Tiene un identificador del número del bloque (*NB*) y el *estado* actual de ese bloque. Los estados registrados por el *directorio local* son:

- Inválido - Este nodo no tiene una copia válida del bloque. Este estado no está registrado en alguna entrada del directorio local; se infiere al no encontrar el bloque.

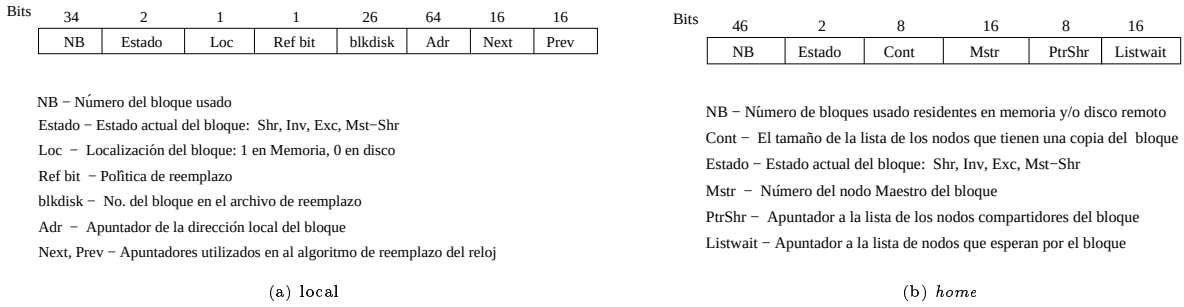

Figura 4.3: Directorios de DDS.

- Compartido - Dos o más nodos de procesamiento en el sistema tienen una copia sólo para lectura (otro nodo es el maestro).
- MaestroCompartido - Este nodo tiene una copia legible en la memoria y alguien más tiene una copia.
- Exclusivo - Sólo este nodo tiene una copia legible y escribible del bloque.

El campo *Loc* del directorio local indica si el bloque se encuentra en la memoria o en el disco. El campo *Refbit* es utilizado por la política de reemplazo, la cual decide cuales bloques son reemplazados, cuando la memoria llega a un límite de su capacidad máxima y así tener espacio para almacenar a un nuevo bloque. Cuando un bloque es reemplazado de la memoria, es almacenado en un archivo temporal, en la posición dada por *blkdisk*. Los campos *Next* y *Prev* son utilizados en el algoritmo de reemplazo para identificar al siguiente y anterior bloque en la lista de reemplazo.

Cuando un bloque no se localiza en la memoria o el disco local del nodo, es necesario localizarlo en el *directorio home*. La figura 4.3.b muestra la estructura del directorio *home*. En el *directorio home* los estados registrados de un bloque son:

- Compartido - Uno o más nodos de procesamiento en el sistema tienen una copia sólo legible.
- Exclusivo - Exactamente un nodo de procesamiento en el sistema tiene una copia legible y escribible.

En el directorio *home* se tiene un contador de accesos (*Cont*) que identifica el número de nodos que están usando el bloque, un identificador del nodo (*Mstr*) que tiene una

copia maestra del bloque y un apuntador (PtrShr) de la lista de los nodos que tienen una copia de ese bloque. Además, el directorio tiene un apuntador (*Listwait*) a la lista de los nodos que esperan por el bloque cuando está ocupado por otro nodo. El *directorio home* se encuentra distribuido en cada uno de los nodos del sistema y su tamaño (el número de elementos) depende de la capacidad de la memoria de cada nodo.

La búsqueda de un bloque en un directorio local o un directorio *home* se realiza a través de la *función Hash* dada por la siguiente ecuación:

$$HashDir = [(H1 + (id * H2))\,\%TT\_HASH] \tag{4.1}$$

Donde:

- $H1 = (nblock\,\%TT\_HASH)$,
- $H2 = 1 + (nblock\,\%(TT\_HASH - 1))$,
- TT_HASH es el tamaño de la tabla Hash,
- id es un índice de la tabla, y
- $nblock$ es el número del bloque.

Para buscar una posición libre en la tabla hash o un número de bloque en la tabla hash, el valor de id está en el siguiente rango: $0 < id < TT\_HASH - 1$.

4.3.1. Manejo de los directorios

El manejo de los directorios *home* y *local* se realiza utilizando una tabla hash para cada uno. En el diseño e implementación de los directorios en DDS se utilizó una tabla hash cerrada porque una tabla hash abierta crece dinámicamente conforme son insertados nuevos elementos en la tabla y esto incrementa su tiempo de respuesta. Aunque una tabla hash cerrada tiene la desventaja de ocupar más espacio en la memoria.

En la Figura 4.4 se observa una tabla hash del directorio local y se tienen tres matrices que están almacenadas parcialmente en memoria y en disco. Cada matriz está formada con cuatro filas o bloques que se insertaron en la tabla hash. Las entradas de la tabla que tienen los bloques que han sido insertados apuntan a la dirección de la memoria local o del disco local en donde está almacenado cada bloque. En la tabla se observa el número

del bloque (*nblk*) y su localización. La localización del bloque se obtiene utilizando el campo *loc* en conjunto con los campos *blkd* y *Adr* del directorio local. Si $Loc = 0$ el dato está en memoria y *Adr* tiene la dirección de la memoria en donde está almacenado el bloque. Si $Loc = 1$ el bloque está almacenado en el disco en un archivo temporal creado en cada nodo y el campo *blkd* tiene la posición inicial del bloque dentro de este archivo.

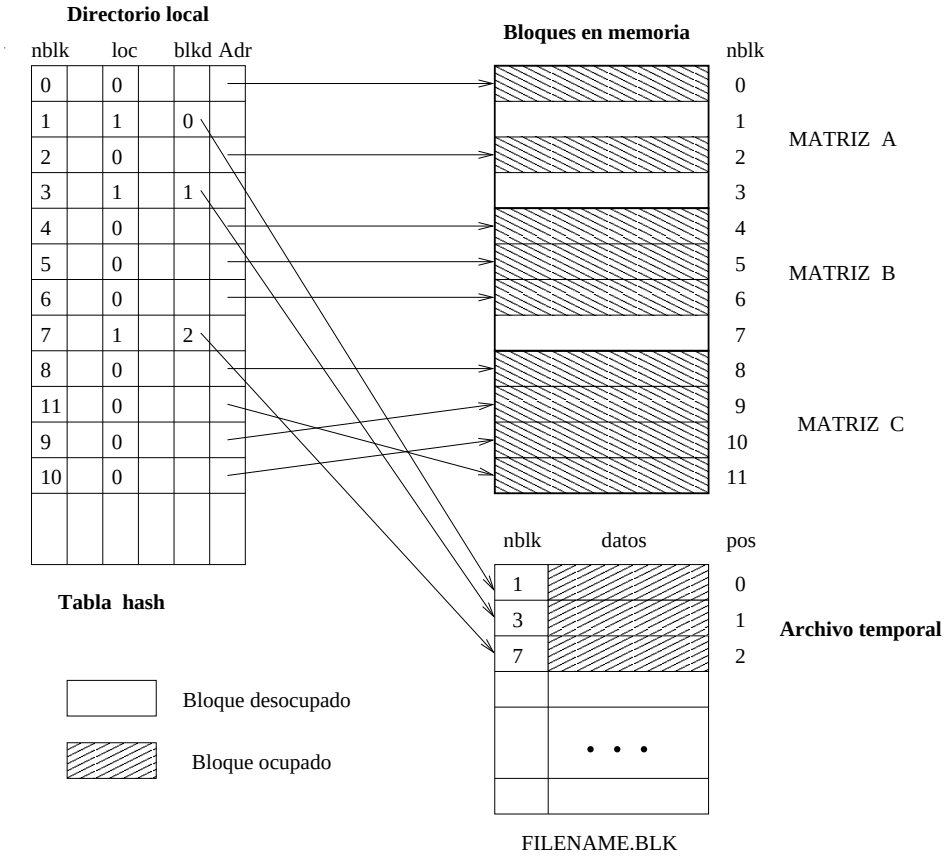

Figura 4.4: Manejo de los directorios en DDS.

En DDS los directorios actualmente son manejados en memoria virtual para mejorar el tiempo de acceso a datos y aumentar la capacidad de direccionamiento. Sin embargo, la capacidad de direccionamiento está limitada por la capacidad de almacenamiento de la memoria virtual para almacenar los directorios.

4.3.2. Selección del nodo Home

La selección del *nodo home* de un bloque es realizada por una *función Hash* dada por la ecuación 4.2, la cual obtiene el módulo del número del *superbloque* con el número de nodos del *cluster*. Un *superbloque* es un bloque de datos consecutivos que son almacenados en un solo nodo y es un múltiplo del tamaño de un bloque de datos. Puede estar formado por uno o varios bloques de datos.

La siguiente ecuación nos permite obtener el nodo home para un bloque de datos:

$$nhome = superblk \,\%\, n\_nodos \tag{4.2}$$

Donde:

- $superblk = off/ssize$,
- $off = nblock * bsize$,
- $superblk$ es el número del *superbloque* dentro del espacio de difusión,
- n_nodos es el número de nodos del *cluster*.
- off es el *offset* del bloque $nblock$ dentro del espacio de difusión,
- $ssize$ es el tamaño del *superbloque*, y
- $bsize$ es el tamaño del bloque,

Actualmente nuestro sistema DDS está diseñado considerando los discos de los nodos lo que permite que el sistema sea escalable. Sin embargo, DDS está limitado por la capacidad de almacenamiento de estos discos.

4.4. Manejo de la memoria en DDS

El mapeo de los datos compartidos dentro del espacio de difusión en el sistema DDS es mostrado en la Figura 4.5. La Figura muestra un espacio de difusión de datos que es visible a 8 nodos. La capacidad de almacenamiento del espacio de difusión está limitado por la suma del espacio de la memoria y del disco. Cada nodo tiene un procesador, una memoria local (EDL) y un disco con un archivo temporal. Los datos mapeados a DDS

se difunden en el espacio de memoria creado en cada nodo llamado EDL. Este espacio es creado con memoria virtual y es definido como un área de memoria compartida dentro del nodo. El tamaño de EDL depende de la capacidad máxima permitida por el sistema operativo del nodo. Esta memoria será compartida entre el proceso de la aplicación y el proceso DDSP ejecutados en cada nodo. DDS y EDL son divididos en bloques de igual tamaño para el almacenamiento de los datos. Cuando un dato compartido es accedido por un proceso de la aplicación, DDS obtiene el número del bloque que le corresponde a este dato dentro de DDS utilizando la siguiente ecuación:

$$n\_bloque = \lceil dir\_dato / t\_bloque \rceil \tag{4.3}$$

Donde, dir_dato es la dirección del dato en el espacio de difusión y t_bloque es el tamaño de cada bloque de la memoria.

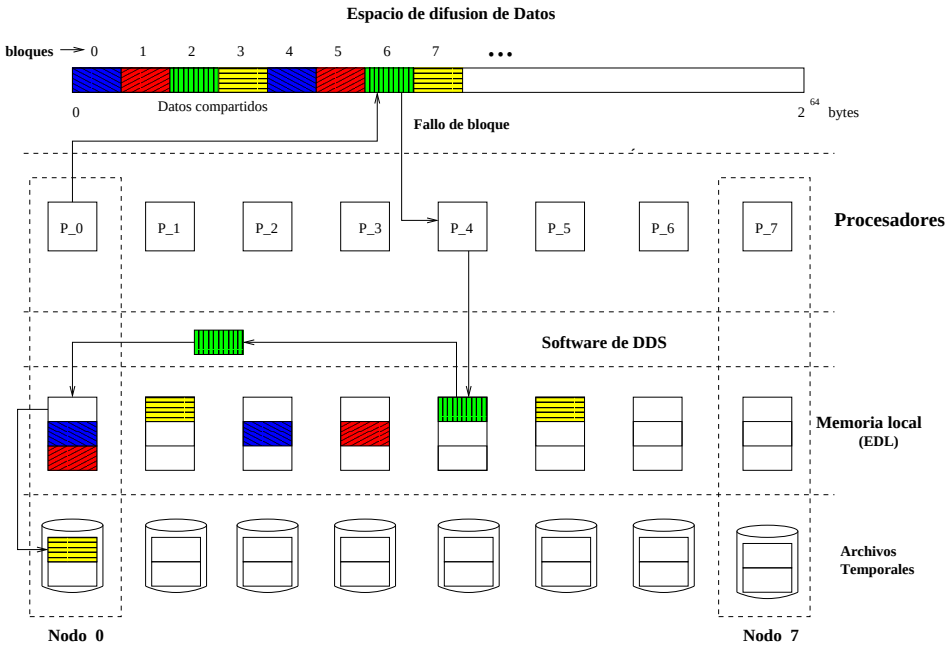

Figura 4.5: Mapeo de la memoria en DDS.

En la Figura 4.5 el nodo 0 busca al bloque 6 que inicialmente no tiene en EDL, entonces realiza una solicitud al nodo 4 y este nodo transmite una copia del bloque al nodo 0. Cuando el nodo 0 recibe el bloque 6, el nodo reemplaza un bloque hacia el disco para liberar espacio en la memoria y poder almacenar el nuevo bloque.

Al iniciar la ejecución de DDS, en cada nodo la dirección inicial de los bloques de EDL es ingresada a una lista de bloques libres llamada *blkmem*. Esta lista contiene las direcciones de los bloques de memoria del nodo local que no están siendo ocupados por los datos. Cuando un proceso de la aplicación requiere usar un bloque de datos en un nodo, la dirección de un bloque de memoria es tomado de la lista *blkmem* e ingresada al directorio local. La posición o entrada correspondiente al número del bloque en el directorio local es obtenida por medio de la *tabla hash*. Al ingresar la dirección de memoria de un bloque en el directorio local ésta es eliminada de la lista *blkmem* y a la vez el número del bloque es ingresado al directorio home.

Para obtener la dirección de un bloque de datos almacenado en EDL, el bloque primero es buscado en el directorio local del nodo utilizando la tabla hash (ver Figura 4.6). Cuando el bloque es encontrado en el directorio local, DDS obtiene la dirección de memoria que está almacenada en la entrada de la tabla hash (el campo *Adr* del directorio local) y junto con la dirección base de EDL del proceso (*dshmadr*) se obtiene la dirección inicial o el *offset* de la memoria EDL en donde está almacenado el bloque.

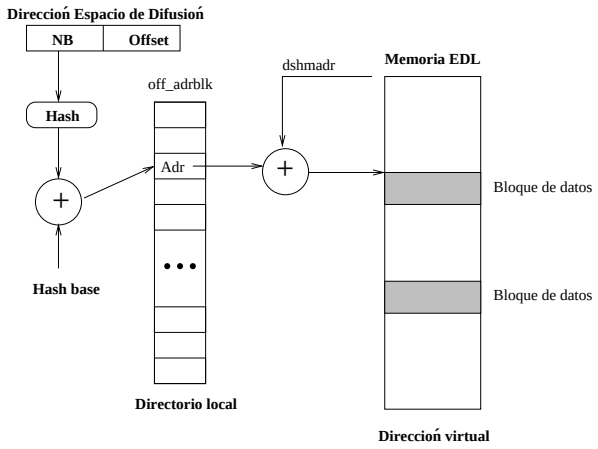

Figura 4.6: Direccionamiento de la memoria en DDS.

De esta manera, el mapeo de una dirección del espacio de difusión a una dirección de EDL dentro del proceso es obtenida utilizando la siguiente ecuación:

$$adr\_edl = dshmadr + off\_adrblk \tag{4.4}$$

En donde, *adr_edl* es la dirección de la memoria en donde está almacenado el bloque dentro del espacio de memoria del proceso en el nodo, *dshmadr* es la dirección base de

EDL y $off\_adrblk = Adr$ es el offset o desplazamiento del bloque dentro de EDL. La Figura 4.7 muestra el mapeo de un bloque de datos dentro de los diferentes espacios de direccionamiento de DDS.

Para el manejo del archivo temporal se utiliza una lista de los bloques del archivo llamada *blkdisk*, en donde los bloques son puestos como libres. Cuando un bloque de datos es almacenado en el archivo, se obtiene el primer bloque del archivo que está en la lista *blkdisk*, se saca de la lista y el bloque de datos se almacena en el archivo en esta posición. El número del bloque que es ocupado en el archivo es insertado en el campo *blkd* del directorio local y el campo *loc* es puesto a 1 (ver Figura 4.4). Cuando un bloque es llevado del disco a la memoria, la posición correspondiente al bloque se elimina del campo *blkd* del directorio local y se inserta nuevamente en la lista de bloques libres *blkdisk* (más detalles ver la sección 4.7).

La dirección obtenida de los bloques es dada al usuario utilizando un apuntador en las funciones de lectura y escritura de datos (ver detalles en la sección 4.6.2). A través de este apuntador el usuario accede los datos compartidos de la misma manera que un arreglo de datos.

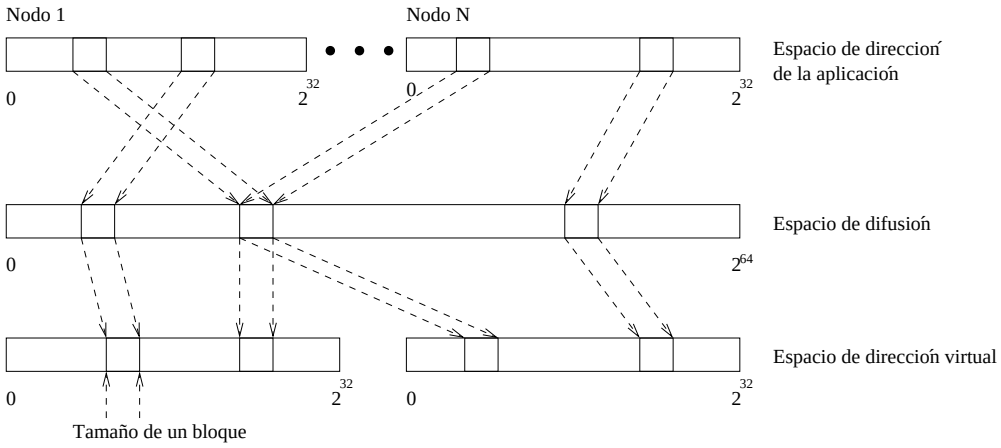

Figura 4.7: Espacios de direccionamiento de la memoria en DDS.

4.5. El Protocolo de coherencia

El protocolo de coherencia sirve para mantener la coherencia de los datos cuando son accedidos por varios procesos. El protocolo es ejecutado por el proceso DDSP en cada nodo y es el encargado de recibir todas las solicitudes de lectura y escritura de cada bloque de datos en el sistema.

La descripción del protocolo de coherencia es facilitado clasificando los nodos de procesamiento dentro de tres categorías: *el nodo local*, *el nodo home* y el *nodo maestro*. El *nodo local* es el nodo que emite una operación de lectura o escritura a un bloque de datos. Un *nodo home* es el nodo que contiene la ubicación de un bloque accedido pero no encontrado en el nodo local. El *nodo maestro* es el nodo que contiene una copia del bloque, en su memoria y/o en su disco, que no puede ser eliminada cuando es necesario liberar espacio en la memoria y/o el disco. Se notará que, en una operación de lectura o escritura, el nodo *home* y/o el nodo maestro pueden ser el mismo nodo local o un *nodo remoto* (cualquier otro nodo que no es el local).

4.5.1. Lectura de un bloque

La Figura 4.8 muestra el pseudocódigo de las funciones del protocolo al recibir una solicitud de lectura de un bloque de datos desde su nodo local. El protocolo busca el bloque de datos en el directorio local. Si lo encuentra en un estado válido (compartido, maestro o exclusivo), devuelve a la aplicación la dirección del bloque de datos para acceder a éstos. El bloque puede no estar en la memoria pero si en el disco, en cuyo caso es primero leído a la memoria.

Si el bloque no se encuentra en el directorio local, se procede a buscarlo en el directorio *home*. Para esto, el protocolo envía un mensaje al nodo *home* con la identificación del bloque deseado (llamada a la función `LecNodoHome` de la Figura 4.8) y espera la respuesta desde el nodo maestro del bloque.

El nodo *home* realiza una búsqueda en su directorio *home* y de esta manera determina la identificación actual del nodo maestro del bloque. Una vez determinado el nodo maestro, el nodo *home* transmite una solicitud de lectura a ese nodo para obtener una copia del bloque y espera un reconocimiento del maestro. Al recibir la solicitud, el nodo maestro buscará en su directorio local el bloque. Si el bloque está en un estado válido,

```
Protocolo-lectura(bloque)
{
Buscar el bloque en el directorio local;
if el bloque está en el directorio local then {
   if el bloque está en la memoria y en cualquier estado then {
      regresar la dirección de memoria del bloque de datos a la aplicación;
   }
   else if el bloque está en el disco then {
      leer el bloque del disco a la memoria;
      regresar la dirección de memoria del bloque de datos a la aplicación;
   }
} else if el bloque no está en el nodo local then {
      solicitar copia del bloque al nodo home (llama a LecNodoHome);
      . . .
      esperar la respuesta del nodo maestro;
      . . .
      recibir una copia del bloque desde el nodo maestro y estos son
      depositados en la memoria;
      mandar un reconocimiento al nodo maestro y
      el bloque pasa a estado Comp en el directorio local;
      regresar la dirección de memoria del bloque de datos a la aplicación;
   }
} (protocolo-lectura)
```

(a)Protocolo de lectura

```
LecNodoHome(bloque)
{
Buscar el bloque en el directorio home;
if el bloque es encontrado then {
   solicitar una copia del bloque al nodo maestro;
   . . .
   esperar el reconocimiento del nodo maestro después
   de que éste transmita los datos al nodo solicitante;
   . . .
   recibir reconocimiento del nodo maestro,
   poner el estado del bloque como compartido y
   agregar al nodo solicitante a la lista de
   compartidores del bloque.
   }
} (lectura-nodo-home)
```

(b) Flujo de la lectura en el nodo home

Figura 4.8: Lectura de un bloque.

le será transmitido al nodo solicitante una copia del mismo y se le envía un mensaje de reconocimiento al nodo *home*. Cuando el nodo *home* obtiene el reconocimiento del maestro, pasa el estado del bloque a compartido en el directorio *home* y agregará al nodo solicitante a una lista de compartidores del bloque, para identificar qué nodos tienen una copia de este bloque de datos.

Cuando DDS en el nodo local recibe la copia del bloque desde el maestro, la deposita en memoria y pone su estado como compartido en el directorio local. Finalmente, regresa a la aplicación la dirección de la memoria donde se ubica el bloque.

4.5.2. Escritura de un bloque

La Figura 4.9 muestra el pseudocódigo de las funciones del protocolo al recibir una solicitud de escritura de un bloque de datos desde su nodo local (nodo solicitante). El protocolo busca el bloque de datos en el directorio local. Si el bloque se encuentra en la memoria o el disco local en estado exclusivo, la aplicación obtiene la dirección del bloque de datos. Si el bloque está en estado compartido, se envía una solicitud de lectura exclusiva al nodo *home* (función `EscNodoHome` en la Figura 4.9) y se espera por

el reconocimiento del *home* de que la copia se ha vuelto exclusiva. El nodo *home* recibe la solicitud de copia exclusiva y busca el bloque en su directorio *home* para identificar al nodo maestro, al cual le reenvía la solicitud. Además, el nodo *home* solicita la invalidación del bloque a cada uno de los nodos que tienen una copia del mismo y espera su reconocimiento. El nodo maestro invalida su copia y transmite un reconocimiento al *home*. Los nodos compartidores mandan un reconocimiento de invalidación al *home* e invalidan sus copias. Cuando el nodo *home* recibe el reconocimiento de todos estos nodos, le transmite un mensaje de reconocimiento al nodo solicitante y se actualiza el directorio *home*. El estado del bloque pasa a exclusivo y se indica cual es la nueva copia maestra. Cuando el nodo solicitante recibe el reconocimiento del *home* cambia el estado del bloque a exclusivo y se regresa a la aplicación la dirección del bloque.

```
Protocolo-escritura(bloque)
{
   Buscar el bloque en el directorio local;
   if el bloque está en el directorio local then {
      if el bloque es exclusivo then {
         if no está en la memoria then {
            leer el bloque del disco a la memoria
         }
         pasar la dirección del bloque a la aplicación;
      } else if el bloque está compartido then {
         mandar una solicitud de escritura exclusiva al nodo home;
         (llama a EscNodoHome)
         . . .
         esperar reconocimiento del nodo home;
         . . .
         el bloque pasa a exclusivo (Excl) en el directorio local;
         pasar la dirección del bloque a la aplicación;
         }
   } else {  /* No está localmente */
      mandar una solicitud de lectura exclusiva al nodo home;
      . . .
      esperar el bloque desde el nodo maestro; y cuando se reciba,
      escribir el bloque en la memoria;
      . . .
      esperar reconocimiento del nodo home; cuando llegue,
      poner el bloque como exclusivo en el directorio local y
      pasar la dirección del bloque a la aplicación.
   }
} (protocolo-escritura)
```

(a) Protocolo de escritura

```
EscNodoHome(bloque)
{
Buscar el bloque en el directorio home;
if el bloque es encontrado then {
   solicitar una copia exlusive al nodo maestro;
   obtener el conjunto de nodos que tienen una copia del
   bloque;
   mandar una invalidación a cada uno de los compartidores
   del bloque;
   . . .
   esperar reconocimientos de los compartidores y del maestro;
   . . .
   mandar reconocimiento al nodo solicitante;
   actualizar el directorio home (bloque como Excl);
   el nodo solicitante pasa a ser el nodo maestro;
   }
} (escritura nodo home)
```

(b)Flujo de la escritura en el nodo home

Figura 4.9: Escritura de un bloque.

Si el bloque no se encuentra en el directorio local del nodo solicitante, se solicita una copia exlusiva al nodo *home*. Si el estado del bloque es compartido, el nodo *home*

solicita al nodo maestro que envíe una copia exclusiva al nodo solicitante; y solicita una invalidación del bloque a cada uno de los nodos que tienen una copia del mismo y espera un reconocimiento de cada uno de ellos. El nodo maestro envía una copia exclusiva al solicitante y después envía un reconocimiento al nodo *home* e invalida su copia. El nodo *home* una vez que recibe el reconocimiento del maestro, pone como maestra a la copia del nodo solicitante e invalida la copia del maestro anterior. Cuando el *home* recibe todos los reconocimientos de invalidación de los compartidores del bloque, manda un reconocimiento al nodo solicitante y cambia el estado del bloque a exclusivo. Cuando el nodo solicitante recibe el reconocimiento del nodo *home* cambia el estado del bloque a exclusivo en el directorio local. Por otro lado, si el estado del bloque en el nodo *home* es exclusivo, el *home* reenvía la solicitud al maestro y espera un reconocimiento del mismo. El maestro envía una copia del bloque al nodo solicitante, su estado es cambiado a inválido y envía un reconocimiento al *home*. Cuando el *home* recibe el reconocimiento del maestro cambia el identificador del maestro del bloque; pone como maestro al solicitante y el estado del bloque sigue siendo exclusivo. Cuando el nodo local recibe el bloque desde el maestro lo deposita en la memoria y el estado del bloque es puesto como exclusivo.

4.5.3. Reemplazo de un bloque

Para reemplazar un bloque de la memoria debemos buscar un bloque candidato a reemplazar. La política de reemplazo utilizada por DDS es el algoritmo de reemplazo del reloj (o algoritmo de segunda oportunidad) con el cual tomamos en cuenta el bit de referencia del directorio local [98]. Este algoritmo ofrece una relación muy buena respecto a la selección del mejor candidato a reemplazarse y el tiempo de búsqueda del candidato. El algoritmo tiene una lista circular con todos los bloques almacenados en memoria y se utiliza una manecilla que apunta al primer bloque en la lista. Si el valor del bit de referencia del bloque que es apuntado por la manecilla es cero reemplazamos el bloque, pero si es 1, le damos una segunda oportunidad, ponemos su bit de referencia a cero y seleccionamos el siguiente bloque. Además son considerados dos bits adicionales, uno para lectura y otro para escritura. Cuando un bloque está siendo accedido tiene uno de estos bits en 1 y para reemplazarlo es necesario que tanto estos bits como el de referencia estén en cero. Cuando un bloque es reemplazado se elimina de la lista y se libera la memoria ocupada.

Un nodo que reemplaza un bloque compartido puede simplemente descartarlo e informar al nodo *home* por medio de un mensaje que lo elimine de la lista de compartidos. Sin embargo, si el estado del bloque es exclusivo o maestro-compartido, entonces el nodo que inició el reemplazo es el nodo maestro actual. Primero debe mandar un mensaje de reemplazo al nodo *home* con el bloque a reemplazar. Cuando el *home* recibe la solicitud, primero checa en su memoria si hay espacio para almacenar el bloque, si lo hay el bloque es almacenado en ésta y el *home* ahora es el maestro. Si no hay espacio en su memoria, el nodo *home* seleccionará a otro nodo remoto en el sistema que tenga la capacidad para almacenar el bloque reemplazado. La selección de otro nodo consiste en dos pasos: primero se obtiene el identificador del *home* (*idhome*) y se determina si el identificador del nodo que solicitó el reemplazo (*idsrc*) es par o impar. Si $idsrc$ es par, se selecciona al nodo cuyo valor sea $idhome + 1$; en caso de ser impar se selecciona al nodo con el valor igual a $idhome - 1$. Por ejemplo el nodo 1 es el que realiza la solicitud de reemplazo y el *home* es el nodo 5, el posible candidato para reemplazar el bloque sería el nodo 4. Si el solicitante fuera el nodo 2, el candidato siguiente sería el nodo 6. Si el nodo seleccionado no tiene espacio suficiente para almacenar el dato, éste le contestará al *home* con un mensaje de rechazo de reemplazo y otro nodo es seleccionado nuevamente. Ahora se probará con el nodo que tenga el valor $idhome + 2$ o $idhome - 2$ según sea el caso. En el ejemplo anterior sería el nodo 3 para $idsrc = 1$ o el nodo 7 para $idsrc = 2$. Si el nodo seleccionado es el nodo que inició la operación, se prueba con el siguiente nodo (Esto puede verse como una lista circular conformada por todos los nodos del sistema.). Este proceso se repetirá hasta que el espacio sea encontrado para acomodar el bloque reemplazado o el nodo seleccionado sea el nodo *home*. Si el nodo seleccionado es el *home* el bloque será almacenado en un archivo de reemplazo temporal de este nodo.

4.6. Modelo de programación

El modelo de programación que proponemos permite definir un espacio de difusión de datos entre varios nodos de un *cluster*. La Figura 4.10 muestra el uso de DDS en la suma paralela de dos matrices: $C = A + B$. El programador debe definir un dato compartido dentro de la estructura `DDS` en el lenguaje C. Antes de usar un dato compartido, el programador debe llamar a la función `DDS_Init` como mostrado en la figura. En cada

nodo, `DDS_Init` mapea el dato compartido al espacio de difusión, inicializa el directorio local y el directorio home, e inicia los procesos DDSP.

En el código de la suma de matrices, $ROW/nprocs$ filas de C son calculadas por cada procesador. Antes de acceder los datos, cada nodo debe ganar acceso hacia el dato, a través de la llamada `DDS_Write` o `DDS_Read`. Estas funciones utilizan como parámetro un identificador de la región de memoria en DDS (una variable o un arreglo definido en la estructura DDS, en este ejemplo: `DDS_A`, `DDS_B` y `DDS_C`) y un *offset* dentro de la región especificando el bloque deseado y retornan un apuntador a la base del área del bloque de datos dentro de la región especificada. Cuando estas funciones retornan, el dato ya está en la memoria del procesador. DDS utiliza un apuntador al dato compartido porque no garantiza que las direcciones en el espacio local de las regiones mapeadas sean las mismas en cada procesador debido a que la dirección local es diferente de la dirección en DDS en la cual la región es mapeada. El programador entonces utiliza el dato de la misma manera a como se usa un dato en su espacio de dirección local. Después de usar el dato el programador debe llamar a `DDS_UnRead()` o `DDS_UnWrite()`, respectivamente, las cuales delegan el control de acceso del bloque y permiten que otro proceso gane su acceso.

Una lectura puede ser ejecutada concurrentemente independientemente del procesador en el cual es ejecutada y cada proceso obtiene una copia del área de datos solicitada. Durante esta operación sólo se permite leer los datos compartidos del área de datos de la región. En una escritura sólo un proceso obtiene una copia exclusiva de los datos, mientras los otros procesos esperan que el bloque de datos sea modificado por el proceso que obtuvo la copia exclusiva. Mediante esta operación, los datos compartidos podrán ser leídos, modificados y almacenados en el área de datos de la región especificada en la escritura. Las operaciones de escritura son serializadas con respecto a todas las otras operaciones sobre la misma región, incluyendo aquellas en otros procesadores.

`DDS_A`, `DDS_B` y `DDS_C` son constantes 0,1 y 2 respectivamente. Estas constantes se refieren al orden en el cual los arreglos A, B y C fueron declarados dentro de la estructura DDS. Ellos son usados en ejecución para indexar el arreglo `dds_vars`. Este arreglo sólo es utilizado por DDS y para cada variable/arreglo contiene los siguientes valores: el tamaño de cada elemento, el número total de elementos y la dirección compartida inicial (DDS). Esta información es utilizada, junto con los otros dos parámetros transmitidos

```
:
struct DDS {                                        /* declarando datos compartidos */
  unsigned int A[ROWS][COLUMNS];
  unsigned int B[ROWS][COLUMNS];
  unsigned int C[ROWS][COLUMNS];
};
:
main() {
:
int *dds_shmemA, *dds_shmemB, *dds_shmemC;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  DDS_Init(sizeof(struct DDS), &myid);                 /* inicializando DDS */

  rows = ROWS/nprocs;
  offset = myid * (ROWS/nprocs);

  for (r=0; r < rows; r++){
    i = r + offset;

    DDS_Write(DDS_C, i*COLUMNS, COLUMNS, &dds_shmC);   /* ganando acceso */
    DDS_Read(DDS_A, i*COLUMNS, COLUMNS, &dds_shmA);    /* a datos compartidos */
    DDS_Read(DDS_B, i*COLUMNS, COLUMNS, &dds_shmB);

    for (j=0; j < COLUMNS; j++){                       /* usando dato compartido */
      dds_shmemC[j] = dds_shmemA[j] + dds_shmemB[j];
    }

    DDS_UnWrite(DDS_C, i*COLUMNS, COLUMNS, &dds_shmC);  /* liberando datos */
    DDS_UnRead(DDS_A, i*COLUMNS, COLUMNS, &dds_shmA);
    DDS_UnRead(DDS_B, i*COLUMNS, COLUMNS, &dds_shmB);
  }

  DDS_Finalize();                                       /* finalizando DDS */
  MPI_Finalize();
:
}
```

Figura 4.10: Ejemplo del modelo de programación de DDS: suma de matrices.

en DDS_Write/DDS_Read, para calcular la dirección DDS del dato siendo accedido. El dato es actualmente accedido a través de los apuntadores dds_shmemA, dds_shmemB y dds_shmemC especificado en las funciones DDS_Write y DDS_Read.

4.6.1. Modelo de consistencia de la memoria

DDS provee un modelo de consistencia de la memoria similar a una consistencia de entrada [10] o de liberación [74] en términos de los almacenamientos y las modificaciones individuales. El almacenamiento de un dato global sólo es permitido dentro de secciones propiamente sincronizadas (utilizando las operaciones DDS_oper) y modificaciones a una región sólo son visibles a los otros procesadores después de la operación apropiada de liberación (una llamada a DDS_UnWrite). Sin embargo, en DDS los objetos de sincronización son implícitamente proporcionados por el programador en la interfaz de DDS: cada región tiene un objeto de sincronización asociado (el número del bloque de datos) el cual es adquirido (*acquired*) y liberado (*released*) utilizando las llamadas DDS_oper y DDS_Unoper, respectivamente.

4.6.2. Interfaz de programación de la aplicación

La interfaz de programación de la aplicación (API) de DDS es relativamente simple y fácil de usar. Incluye llamadas a funciones para la creación de los procesos DDSP (aunque esto es transparente al usuario), para la creación e inicialización del espacio de difusión de los datos compartidos de la aplicación y funciones para el control de acceso a datos entre los procesos. (Para la creación de los procesos de la aplicación actualmente se utiliza a MPI).

A continuación presentamos la sintaxis, la descripción y el valor de retorno de las funciones en DDS.

- *int DDS_Init(void **dshm, long long shmsize, struct dds_arrstat *, int np)*

  La función DDS_Init es ejecutada en cada nodo del PC-cluster y es utilizada para crear el espacio de difusión especificado en la estructura DDS. La función DDS_Init regresa la dirección de la memoria del espacio de difusión local en el apuntador *dshm*, utiliza como parámetro el tamaño de la estructura DDS en la variable *shmsize*, en la estructura *dds_arrstat* el usuario puede especificar una distribución

inicial de los datos compartidos en caso de ser necesario y en el parámetro *np* se debe dar la identificación del nodo local.

Valor de retorno: Esta función es una función colectiva que al término de su ejecución se comunica con el nodo 0 conocido como *nodo manager*. Si algún nodo al ejecutar `DDS_Init` no se comunicara con el nodo *manager* es indicativo de que existió un error al arranque del sistema. Al regresar la función retorna la dirección en memoria del espacio de difusión en la variable `dsm`. Si `DDS_Init` regresa un valor positivo su ejecución fué exitosa y un valor 0 si existió algún error.

- *int DDS_Finalize()*

  La función `DDS_Finalize` se utiliza para liberar el espacio de difusión de DDS cuando los procesos de la aplicación han finalizado de usar los datos compartidos. Esta función no utiliza parámetros.

- *int DDS_Write(int arr, int offset, int ndatos, int type, void *blk)*

  La función `DDS_Write` realiza una solicitud de lectura exclusiva de un bloque de datos de una variable o arreglo compartido en el espacio de difusión. Esta función utiliza como parámetros el nombre del dato o arreglo compartido especificado en *arr* el cual es generado en un proceso de precompilación, el *offset* dentro del arreglo, el número de datos solicitados es dado en *ndatos* y el tamaño de cada dato en bytes es dado por *type*. Cuando esta función retorna el bloque de datos solicitado ya está en el espacio EDL del nodo, es exclusivo al proceso que lo solicitó y permanecerá ahí mientras la función correspondiente `DDS_UnWrite` no sea ejecutada. La dirección del bloque dentro de EDL es dada en el apuntador *blk*, el cual será utilizado para acceder a los datos del bloque solicitado.

  Valor de retorno: `DDS_Write` regresa un valor positivo si la operación fué exitosa, e indica que el bloque ha sido almacenado en la memoria local del nodo asignada al espacio de difusión de DDS. Si un valor negativo es regresado, la solicitud presentó un error en la lectura del bloque y éste no ha sido almacenado en la memoria local del nodo. Cuando el bloque es almacenado en la memoria su estado es puesto como exclusivo y su control es otorgado al proceso que lo solicitó. A partir de este momento el proceso que solicitó el bloque puede accederlo y modificarlo y

ningún otro nodo podrá acceder el bloque hasta que sea liberado por el proceso, ni podrá ser reemplazado o movido para liberar espacio en memoria.

- *int DDS_Read(int arr, int offset, int ndatos, int type, void *blk)*

  La función `DDS_Read` permite realizar una solicitud de lectura de un bloque de datos de una variable o arreglo compartido en el espacio de difusión. Esta función utiliza los mismos parámetros que `DDS_Write`. Cuando la función retorna, el bloque de datos solicitado ya está en el espacio EDL del nodo, tiene un estado compartido y permanecerá ahí mientras la función correspondiente `DDS_UnRead` no sea ejecutada

  Valor de retorno: `DDS_Read` regresa un valor positivo si la operación fué exitosa e indica que el bloque ha sido almacenado en la memoria local del nodo asignada al espacio de difusión de DDS, o un valor negativo si existió algún error en la lectura y el bloque no ha sido almacenado en la memoria local del nodo. Cuando el bloque es almacenado en la memoria su estado es puesto como compartido y es amarrado a esta memoria y permanecerá ahí hasta que sea liberado.

- *int DDS_UnWrite(int arr, int offset, int ndatos, int type, void *blk)*

- *int DDS_UnRead(int arr, int offset, int ndatos, int type, void *blk)*

  Estas dos últimas funciones (DDS_UnWrite y DDS_UnRead) permiten liberar un bloque de datos de la memoria. Cuando el bloque es liberado puede ser reemplazado o movido por DDS hacia otro nodo o hacia el disco si es necesario usar su espacio en memoria para almacenar otro bloque. Los parámetros utilizados en estas funciones son los mismos que se usaron en `DDS_Write` y `DDS_Read` respectivamente.

4.7. Aspectos de implementación

Para reemplazar los bloques de la memoria al disco, se utiliza un archivo de reemplazo (archivo temporal) en cada nodo del sistema (ver Figura 4.5). Este archivo de reemplazo es dividido en bloques de igual tamaño a los de memoria y el tamaño de este archivo puede ser hasta del espacio que se tenga en el disco. Al inicio de DDS y de la aplicación el tamaño de este archivo es de 0 bytes, los bloques son marcados como libres y un

identificador de cada bloque es insertado en una lista de bloques libres. Cuando un bloque de datos es reemplazado de la memoria se almacena en el archivo de reemplazo en la posición dada por el bloque obtenido de la lista de bloques libres. El bloque utilizado en el archivo de reemplazo se saca de la lista de libres y se marca como ocupado. Cuando un bloque de datos es llevado nuevamente a la memoria, el identificador del bloque del archivo de reemplazo se marca como libre y se inserta nuevamente al principio de la lista. La identificación y posición del bloque que está siendo ocupado en el archivo de reemplazo es almacenado en el directorio local (ver Figura 4.4).

4.7.1. Otros aspectos de diseño

Si el usuario lo desea puede definir un patrón de distribución inicial de los datos utilizando la estructura `dds_arrstat`. En esta estructura se especifica el nodo base (*base*), el número de nodos en donde serán iniciados los datos que son definidos dentro de la estructura DDS (*pcount*), el tamaño del *superbloque* (*ssize*) y el tamaño del bloque de datos (*bsize*). El nodo base es el nodo inicial de la lista de nodos a partir del cual el arreglo es creado. El tamaño del superbloque es el número de datos consecutivos que serán almacenados en un nodo. La sintaxis de `dds_arrstat` es la siguiente:

```
struct dds_arrstat arstat={base,pcount,ssize,bsize};
```

En el ejemplo de la Figura 4.10, si hacemos que los datos sean almacenados en todos los nodos del *cluster* iniciando desde el nodo 0, y que el tamaño del *superbloque* y el tamaño del bloque de datos sean iguales al tamaño de un renglón de las matrices. Si consideramos que el tamaño de un entero en DDS es `DDS_UINT`, entonces tendríamos:

```
struct dds_arrstat arstat={0,nprocs,COLUMNS*DDS_UINT,COLUMNS*DDS_UINT};
```

4.8. Resumen

En este capítulo presentamos un *Espacio de Difusión de Datos bajo cómputo paralelo en Clusters* llamado `DDS`, cuyo propósito es facilitar el desarrollo de aplicaciones paralelas y mejorar su desempeño.

En DDS las aplicaciones paralelas son ejecutadas bajo el modelo SPMD. DDS es un espacio adicional al de cada proceso ejecutando una aplicación paralela y su tamaño

puede ser hasta de 2^{64} bytes, ya sea para arquitecturas de 32 o 64 bits. DDS es definido como un espacio de memoria compartida e implementado completamente en software para cómputo paralelo en PC-clusters.

En DDS, un proceso de la aplicación y el proceso DDSP son ejecutados en cada nodo de procesamiento. El proceso DDSP es el encargado de difundir, migrar, o replicar los datos entre las memorias de los nodos que los utilizan. DDS permite difundir los datos entre los procesadores de manera automática conforme los necesiten. Los datos se difunden bajo un protocolo de coherencia *mútiples lectores un solo escritor*. Para una solicitud de lectura, una copia del dato es obtenida; para una solicitud de escritura, una copia exclusiva es obtenida invalidando todas las demás copias, asegurando que todos los procesadores tengan el mismo dato compartido.

Cuando un bloque es solicitado por un proceso de la aplicación, DDSP utiliza dos directorios para localizarlo y revisar su estado. El directorio local contiene la localización del bloque en memoria o disco y el estado actual del bloque. El directorio home permite buscar un bloque en otro nodo, revisar su estado y ver cuántas copias existen entre los nodos.

Para facilitar la programación de aplicaciones paralelas, DDS tiene una interfaz de programación simple y fácil de usar. EL usuario sólo tiene que definir los datos compartidos dentro de una estructura del sistema, mapear estos datos dentro del espacio de DDS utilizando la función de inicialización `DDS_Init` y acceder los datos con las funciones de lectura y escritura: `DDS_Read` y `DDS_Write`. Estas funciones facilitan la programación de aplicaciones paralelas porque el usuario sólo se enfoca en el problema algoritmico al omitir la localización de los datos, la identificación y sincronización de los nodos y definir un espacio de difusión de datos utilizando la memoria de todos los nodos que mejora su desempeño.

En relación con otros sistemas de programación paralela, DDS tiene las siguientes ventajas: ofrece una misma interfaz para programar aplicaciones *in-core* y *out-of-core*, al ser implemetado totalmente en software es portable en aruitecturas heterogéneas de 32 y 64 bits y maneja un espacio de direccionamiento de 64 bits.

Capítulo 5

Sistema de archivos paralelo sobre DDS

5.1. Introducción

En el capítulo anterior vimos el diseño de un espacio de difusión de datos (DDS). DDS es una memoria compartida distribuida toda en software, que permite difundir los datos desde la memoria o el disco entre diferentes procesadores, para ejecutar aplicaciones paralelas en PC-clusters. DDS tiene una capacidad de almacenamiento de 2^{64} bytes y utiliza la memoria o disco de los nodos para almacenar los datos. En DDS este espacio de almacenamiento es ideal para ejecutar aplicaciones *out-of-core*. Estas aplicaciones generan o procesan grandes cantidades de datos, del orden de cientos de GBytes, basadas en la generación y el acceso de sus datos podemos clasificarlas en 3 tipos: las que utilizan como entrada de datos a archivos de gran tamaño; las que generan y procesan grandes cantidades de datos sin necesidad de una entrada o salida de datos en archivos; y las que requieren almacenar sus resultados en archivos grandes. Este tipo de aplicaciones normalmente requiere de un acceso constante a disco durante su ejecución. Sin embargo, el acceso constante a disco puede potencialmente incrementar el tiempo de ejecución de estas aplicaciones y degradar su rendimiento.

Una alternativa para mejorar el tiempo de ejecución de estas aplicaciones es tener un sistema de archivos paralelo capaz de almacenar estos datos, que permita un acceso simultáneo a diferentes discos y capaz de leer o escribir cientos de MBytes por segundo.

En un sistema de archivos paralelo los archivos son divididos en bloques los cuales son distribuidos entre los nodos de E/S. De esta manera, un archivo puede estar almacenado en varios nodos y disminuir el tiempo de acceso al permitir acceder de manera paralela diferentes bloques por diferentes procesadores. Sin embargo, aún cuando se tienen accesos paralelos utilizando varios discos a la vez para mejorar el tiempo de acceso, normalmente se realizan accesos directos al disco local del nodo de E/S que almacena el bloque de datos solicitado. Esta operación generará al menos un mensaje de E/S entre el nodo de cómputo y el nodo de E/S y un acceso al disco del nodo de E/S lo que también puede degradar el desempeño de las aplicaciones.

Otra alternativa para mejorar el tiempo de ejecución de las aplicaciones de E/S es disminuir el número de los accesos a disco. Los accesos disminuyen al acceder los datos desde la memoria de los procesadores que tengan una copia del dato. El número de accesos a disco puede potencialmente ser reducido, si el sistema de archivos tiene un espacio en caché lo suficientemente grande para mantener en memoria los datos de los archivos que una aplicación accede en un determinado tiempo. El espacio de la caché puede aumentarse si se consideran todas las memorias de los nodos de un *cluster*.

Otro aspecto a considerar en un sistema de archivos paralelo es cuando varios nodos de cómputo acceden en forma simultánea a un bloque de datos y al menos un nodo lo modifica, lo cual puede llevar a problemas de sincronización para mantener su coherencia. Para sincronizar los procesos el programador debe implementar un protocolo de coherencia "múltiples lectores un solo escritor" usando funciones de paso de mensajes que complican la programación.

Por otro lado, la distribución de los bloques en los archivos normalmente es fija durante toda la ejecución de la aplicación y en muchas ocasiones no corresponde con los patrones de acceso de la aplicación. Los bloques dificilmente están en la memoria o en el disco de los procesadores que los utilizan, lo que puede producir un mal desempeño si no se eligen los parámetros más adecuados en la distribución de los bloques dentro del sistema de archivos paralelo.

Para diseñar un sistema de archivos paralelo de alto rendimiento deben considerarse los aspectos anteriores con la finalidad de incrementar el ancho de banda, es decir disminuir el número de operaciones de acceso a disco, disminuir el número de mensajes de E/S entre los procesos y maximizar la tasa de acierto del bloque (o *hit rate*) dentro de

la memoria de un nodo para evitar los accesos a datos remotos.

Considerando estos aspectos se diseño e implementó un *Sistema de Archivos Paralelo sobre DDS* llamado FSDDS (del inglés: File System atop Data Diffusion Space [17]). FSDDS es integrado a DDS para desarrollar y correr aplicaciones paralelas *out-of-core*. El propósito de FSDDS es mejorar el desempeño y facilitar la programación de aplicaciones *out-of-core* que leen/escriben datos en archivos y que pueda ser portable en arquitecturas heterogéneas. FSDDS, como la mayoría de los sistemas de archivos paralelo, provee acceso paralelo a archivos particionados en bloques y distribuidos a través de múltiples nodos y/o discos de un PC-cluster (estos nodos no necesitan ser homogéneos). FSDDS es similar a PVFS, salvo que FSDDS utiliza la memoria de todos los nodos como un solo caché. Cuando un archivo es abierto bajo FSDDS, éste es automáticamente mapeado dentro del espacio de dirección compartido de DDS y este archivo es manejado como si fuera un arreglo de memoria, similar a los arreglos de DDS. Es decir, los arreglos definidos en el espacio de memoria de DDS y los bloques de datos de los archivos FSDDS son accedidos de manera idéntica bajo DDS y FSDDS. De esta manera, cuando los bloques de datos del archivo son accedidos por cualquier nodo del *cluster*, éstos son copiados y almacenados (difundidos) en la memoria del nodo que los solicitó.

Cuando un archivo es manejado como un arreglo en memoria, facilita al usuario la programación de aplicaciones *out-of-core*. Con esta técnica de programación el usuario no debe preocuparse por la capacidad en memoria de los nodos del *cluster*, ni por la localización de los bloques dentro del *cluster*. Este manejo en memoria de los bloques de datos de los archivos tiende a mejorar el desempeño de aplicaciones *out-of-core* ya que se disminuyen los accesos a disco. Además, los bloques tienden a moverse hacia quién los necesita disminuyendo las operaciones de E/S entre los nodos del *cluster*.

En la Figura 5.1 mostramos el código de la nueva versión del ejemplo de la Figura 3.16 en el cual un proceso escribe datos a disco y otro proceso los lee del mismo. Ahora no es necesario escribir los datos hacia el disco, DDS los transfiere a quién los solicita en forma transparente y directamente a los módulos de memoria de los nodos involucrados. Se sigue utilizando la barrera con MPI sólo para establecer el orden de acceso entre los procesos.

En las secciones siguientes presentamos el diseño de FSDDS: su arquitectura, el particionamiento y distribución de los datos, el mapeo de los archivos, su acceso y la

```
fh = DDS_Open("sample", ..., &filestat);
DDS_Write(fh, off, lenbuf, &Arr);
/* Accediendo dato por el proceso 0 */
DDS_UnWrite(fh, off, lenbuf, &Arr);
DDS_Barrier(MPI_COMM_WORLD);
...
/* Dato compartido por el proceso 1*/
...
DDS_Close(fh);
```

(a)Proceso 0

```
fh = DDS_Open("sample", ..., &filestat);
...
/* Dato ocupado por el proceso 0 */
...
DDS_Barrier(MPI_COMM_WORLD);
DDS_Read(fh, off, lenbuf, &Arr);
/* Accediendo dato por el proceso 1 */
DDS_UnRead(fh, off, lenbuf, &Arr);
DDS_Close(fh);
```

(b) Proceso 1

Figura 5.1: Acceso a un archivo por dos procesos con FSDDS

interfaz de programación del usuario. Además presentamos la programación de una aplicación mostrando su facilidad de programación.

5.2. Arquitectura

Al igual que otros sistemas de archivos paralelo, tal como PVFS, FSDDS es diseñado como un sistema cliente-servidor con múltiples servidores de E/S llamados nodos de E/S. En la Figura 5.2 se muestra un PC-cluster con los nodos que componen al sistema de archivos FSDDS. FSDDS tiene cinco tipos de nodos: los *nodos de E/S*, los *nodos de cómputo*, los *nodos home*, los *nodos maestros* y el *nodo coordinador*.

Un *servidor de E/S o nodo de E/S* (NES) en FSDDS es un nodo del *cluster* que tiene al menos un disco que almacena los archivos de datos, que atiende las peticiones de E/S locales o las solicitudes de E/S generadas desde otro nodo dentro del *cluster* y es ejecutado de manera independiente al cómputo de la aplicación.

Los *nodos de cómputo*, también conocidos como nodos de procesamiento (NP), son los nodos del *cluster* donde corre la aplicación. Los *nodos home* son los nodos que contienen la ubicación de un bloque, coordinan el acceso a un bloque por varios nodos y permiten el intercambio de datos entre los nodos de cómputo y los nodos de E/S. Los *nodos maestros* son los nodos que tienen una copia maestra del bloque (los nodos home y los nodos maestros son los mismos nodos presentados en la difusión de datos del capítulo 4, por lo que no se darán más detalles de estos nodos). El *nodo coordinador* es el nodo que arranca el sistema FSDDS, sincroniza a los nodos de E/S y permite crear, abrir o cerrar un archivo en varios nodos del *cluster*. El *nodo coordinador* es el nodo en donde inicia la ejecución de una aplicación. Cualquier nodo en el sistema paralelo

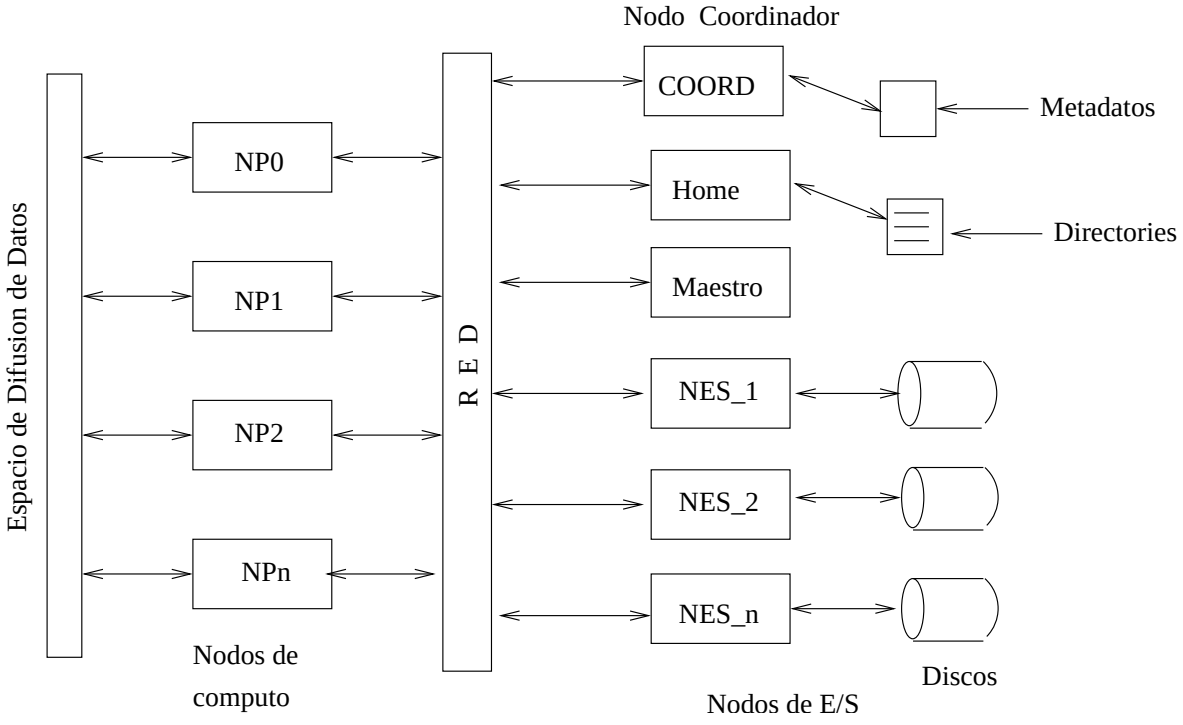

Figura 5.2: Arquitectura del sistema de archivos FSDDS.

puede funcionar como cualquiera de estos tipos de nodos o desempeñar la función de varios tipos a la vez; sólo existe una restricción para que un nodo del sistema pueda ser un nodo de E/S: debe tener un disco local (como se mencionó anteriormente) y el disco debe tener al menos 1 GB de espacio para el almacenamiento de datos (este valor puede ser ajustado por el programador de acuerdo a sus necesidades).

En FSDDS también se utiliza el proceso DDSP de DDS (ver Capítulo 4). Al correr una aplicación con FSDDS cada nodo en el *cluster* corre al proceso DDSP. El proceso DDSP es el que ejecuta las operaciones del protocolo de coherencia (Sección 4.5) y además realiza las operaciones de E/S cuando es ejecutado en un nodo de E/S. De esta manera, el proceso DDSP también atiende las solicitudes de E/S a disco.

Al ejecutar una aplicación *out-of-core* en un *cluster*, ejecutamos un proceso de la aplicación en cada nodo del *cluster*. Para que los procesos de la aplicación se puedan comunicar con el proceso DDSP se utiliza una biblioteca de funciones de E/S (ver Figura 5.3 para mayores detalles de cada nodo). Estas funciones permiten a la aplicación realizar todas las solicitudes de acceso de los bloques de datos, coordinar su acceso, realizar el

particionamiento y distribución de los archivos de datos; así como crear, abrir, borrar y cerrar los archivos en FSDDS.

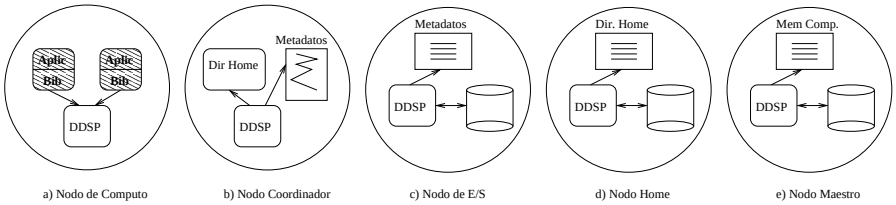

Figura 5.3: Tipos de nodos en FSDDS.

Cuando los procesos de la aplicación desean acceder a un bloque de un archivo, hacen las solicitudes al sistema de archivos FSDDS, utilizando llamadas a las funciones de la biblioteca. Por medio de estas funciones, la aplicación pasa la solicitud de un bloque al proceso DDSP. DDSP se encarga de buscar, traer y almacenar el bloque en la memoria local del nodo (para una mayor descripción de los accesos ver las secciones 5.4 y 5.5).

5.3. Particionamiento y distribución de datos

FSDDS particiona los archivos de datos en bloques y los distribuye a través de múltiples discos sobre múltiples nodos de E/S. Para definir la distribución de los datos en los nodos de E/S, se usan en conjunto varios parámetros de particionamiento, como son: el número del nodo de E/S base, el número de nodos de E/S, el tamaño del *stripe*, el tamaño del bloque de datos y el número de bloques consecutivos asignados a un *stripe*. Un *stripe* en FSDDS es un conjunto de datos contiguos del archivo que es almacenado en un mismo nodo. Estos parámetros se almacenan en un archivo de *metadatos* (ver Tabla 5.1). El archivo de metadatos es replicado en todos los nodos de un *cluster* donde corre una aplicación.

Un *metadato* es la información que describe las características de distribución de un archivo. La información de la distribución contiene tanto la localización del archivo sobre el disco, como la localización de los discos en el *cluster*. Los parámetros de particionamiento definen el patrón de distribución de los archivos, cuya función es determinar las características de distribución física en disco. Los parámetros son usados por FSDDS para localizar los bloques de datos de un archivo cuando una aplicación solicita acce-

der al archivo. De esta manera, los datos son distribuidos físicamente a través de los dispositivos de E/S al momento de crearlos y son movidos a la memoria de los nodos que corren el programa paralelo conforme a sus patrones de acceso. La aplicación ve al archivo como un conjunto lineal de datos y lo accede de manera similar a los archivos en UNIX.

Filename	mult
base	2
pcount	4
ssize	64 K
bsize	16 K

Tabla 5.1: Archivo de metadatos de FSDDS.

5.3.1. Parámetros del particionamiento

En FSDDS un archivo consiste de un conjunto ordenado de *stripes*. Los *stripes* son asignados en forma de *round-robin* a través de los nodos de E/S y a su vez son divididos o particionados en bloques. El tamaño de este bloque debe ser un múltiplo del tamaño de los bloques de la memoria de difusión. En la Figura 5.4 observamos un archivo particionado en 6 *stripes*, cada *stripe* es dividido en 4 bloques y es almacenado en 3 nodos de E/S. Este particionamiento junto con un ordenamiento de los nodos de E/S, especifican completamente la distribución de un archivo en el sistema de archivos FSDDS.

Los nodos de un *cluster* pueden ser seleccionados para operar como nodos de E/S, como nodos de cómputo o como ambos a la vez. También es posible seleccionar a los nodos de E/S para almacenar a un archivo. En la Figura 5.5 observamos a 8 nodos de E/S y un archivo llamado *mult* que es distribuido sólo a través de 4 nodos de E/S, iniciando desde el nodo 2. El archivo es localizado en */home/dds/*. La Figura 5.5 muestra como un archivo llamado *mult* localizado en */home/dds/* es distribuido en FSDDS. Podemos observar que aunque hay 8 nodos de E/S en el ejemplo, el archivo es distribuido sólo a través de 4 nodos de E/S, iniciando desde el nodo 2. Cada nodo de E/S almacena su porción de datos del archivo *mult*. El almacenamiento en los nodos de E/S es realizado en un archivo local sobre el sistema de archivos nativo en el nodo de E/S correspondiente.

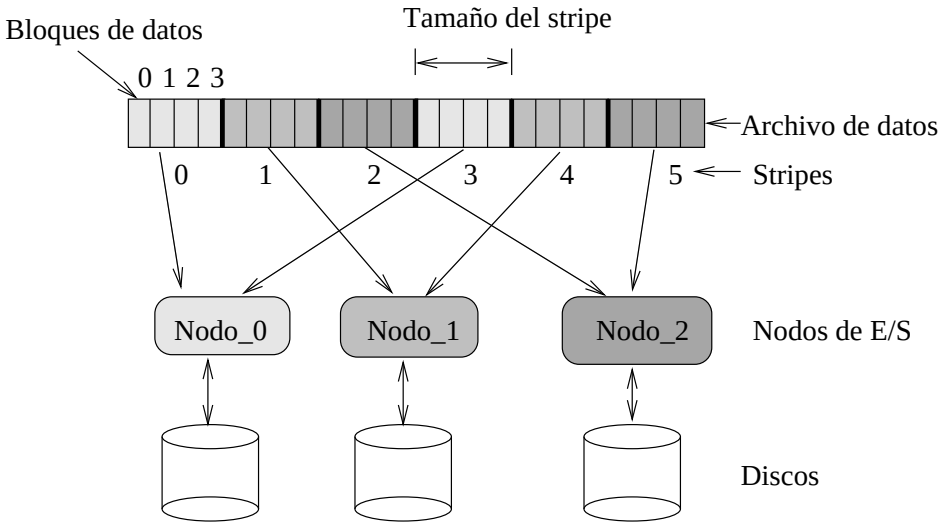

Figura 5.4: Particionamiento de un archivo en FSDDS.

El nombre de cada archivo local se forma concatenando al nombre original del archivo el número del nodo de E/S del archivo local menos el número del nodo base. En nuestro ejemplo, el archivo está almacenado en 4 nodos y el nodo base es el nodo 2 (NES_2). Por tanto, en el nodo 2 el nombre del archivo local será */home/dds/mult0*, en el nodo 3 (NES_3) es */home/dds/mult1* y así sucesivamente hasta *.../mult3*.

Un ejemplo de los campos del metadatos para el archivo "mult"de la Figura 5.5 se muestra en la Tabla 5.1. El campo *base* especifica que el primer nodo de E/S (o nodo base) es el nodo 2, *pcount* especifica que los datos son divididos en cuatro nodos de E/S, *ssize* especifica que el tamaño del *stripe* es 64 Kbytes y *bsize* es el tamaño del bloque de datos (16 Kbytes). La distribución de datos puede ser especificada por el usuario o dejar que FSDDS la asigne de manera automática.

Cuando una aplicación accede un bloque de datos de un archivo en FSDDS, la tabla de metadatos es utilizada para determinar su localización física. Con estos valores, FSDDS determina la identificación del nodo de E/S en donde está almacenado el *stripe* que contiene el bloque. Primero obtiene el *offset* (o la posición inicial) del bloque dentro del archivo para localizar los datos iniciales del bloque usando la siguiente ecuación:

$$offset = nblock * bsize \tag{5.1}$$

Donde: *nblock* es el número del bloque y *bsize* es su tamaño.

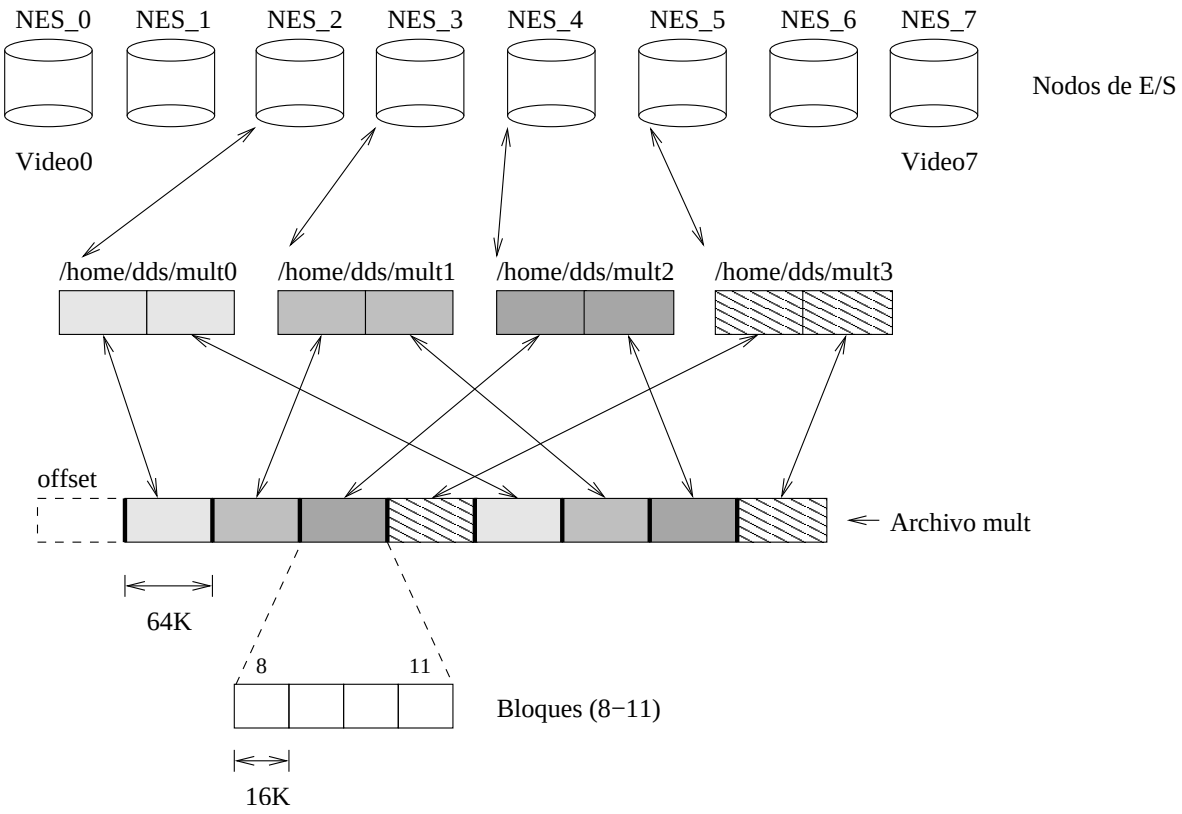

Figura 5.5: Almacenamiento de un archivo en FSDDS.

Después, el número del *stripe* correspondiente a este bloque es determinado con la siguiente ecuación:

$$nstripe = \lfloor \frac{offset}{ssize} \rfloor \tag{5.2}$$

Así, la identificación del nodo de E/S (nes) donde está almacenado un stripe está dado por la siguiente ecuación:

$$nes = (nstripe \,\%pcount) + base \tag{5.3}$$

La expresión $nstripe\,\%pcount$ determina qué nodo de E/S almacena el bloque y se le suma la *base* para obtener el número del nodo dentro del *cluster*. Si el bloque ya ha sido accedido anteriormente y existe una copia del bloque en la memoria de otro nodo, entonces es necesario identificar el *nodo home* del bloque para localizar la copia del bloque. La identificación de un *nodo home* (dado por *nhome*) depende del número

del *stripe* obtenido en la ecuación 5.2 y del número de nodos de E/S utilizados para almacenar el archivo. Para determinar al nodo *home* utilizamos la siguiente ecuación:

$$nhome = nstripe \% nnodos \tag{5.4}$$

5.4. Mapeo de archivos

Los archivos en FSDDS se pueden acceder de manera tradicional con operaciones de E/S similar a la interfaz del sistema de archivos UNIX. Como se recordará, en estas operaciones de E/S el usuario especifica una dirección de la memoria antes de acceder a un bloque del archivo que será leído/escrito en esta memoria. Adicionalmente a estas operaciones de E/S, FSDDS permite que un archivo sea manejado como un arreglo almacenado en el espacio de difusión de DDS, sin especificar una dirección de la memoria en particular (Figura 5.6). El arreglo es dividido en bloques que pueden ser compartidos a la vez por más de un nodo de procesamiento permitiendo que múltiples tareas de una aplicación puedan acceder distintas particiones del archivo de manera paralela o independiente. Para que el manejo de la memoria en DDS sea transparente al usuario, el espacio de difusión de DDS es dividido en bloques de un Terabyte (TB). (Se tomó este valor porque los archivos actuales tienen un tamaño menor, pero puede ser ajustado en caso necesario). Así, el primer TB es reservado por DDS para el manejo de los arreglos compartidos definidos en memoria dentro de la aplicación; los siguientes TBs son utilizados uno para cada archivo. FSDDS automáticamente mapea el primer archivo en el segundo TB del espacio de difusión de DDS, el segundo archivo en el tercer TB y así sucesivamente.

Para mapear un archivo en el espacio de difusión de DDS éste debe ser abierto o creado utilizando FSDDS. Cuando una aplicación abre o crea un archivo usando FSDDS, una solicitud es transmitida al *nodo coordinador* desde un *nodo de cómputo* (ver Figura 5.7a). El *nodo coordinador* determina cuales son los nodos de E/S en los que se abrirá o creará el archivo y les retransmite la solicitud a estos nodos. Estas solicitudes fluyen a través de los procesos DDSP entre los nodos del *cluster*. Cuando los nodos de E/S reciben la solicitud determinan qué archivo está siendo accedido y verifican si existe en su disco local. Si el archivo ya existe, éste es abierto con los atributos dados en la solicitud; en caso contrario, el archivo es creado en la dirección especificada en la

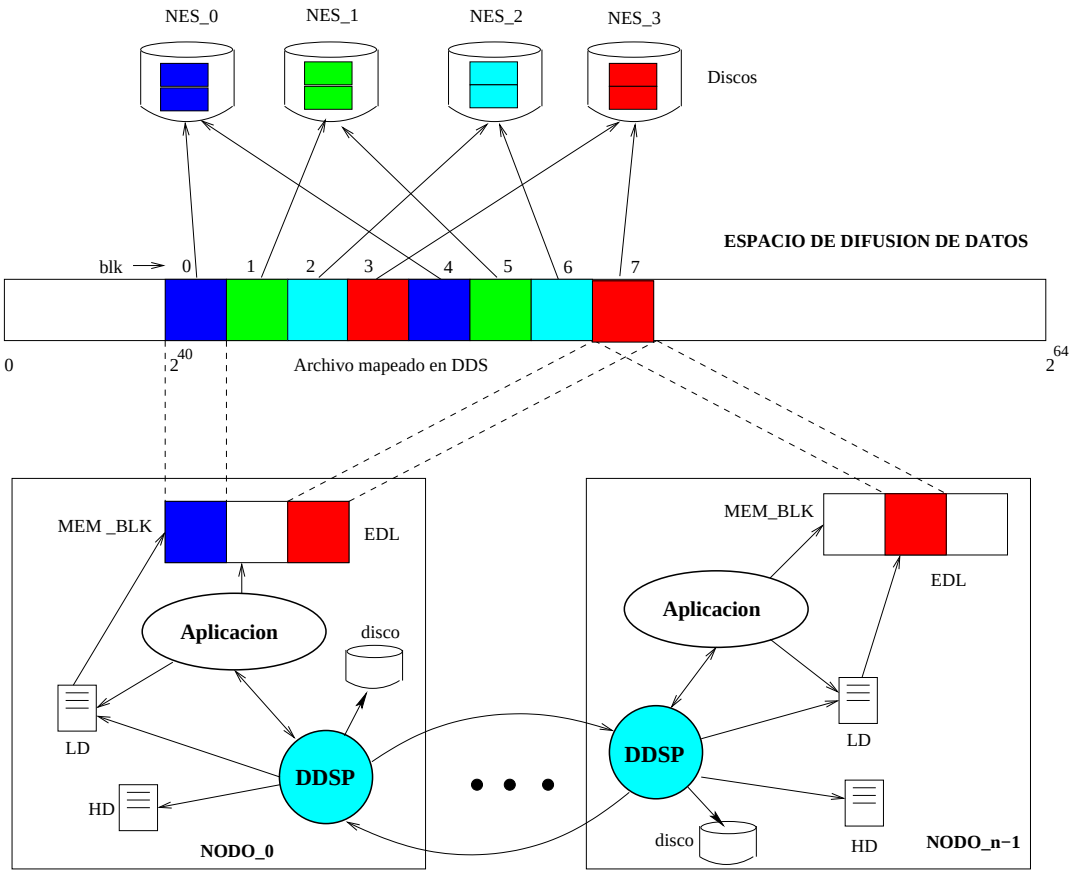

Figura 5.6: Mapeo de un archivo en FSDDS.

solicitud. Finalmente, el archivo es abierto o creado por todos los nodos de E/S.

Cuando el archivo ya está abierto, todos los accesos a los bloques de datos de este archivo se realizarán a través del *nodo home*. Esto es, primero se transmite una solicitud de acceso del bloque al nodo home y éste identificará al nodo maestro que tiene el bloque para retrasmitirle la solicitud. En la Figura 5.7b puede observarse que el nodo de cómputo envía la solicitud al home cuando requiere acceder a un bloque.

Al realizar por primera vez la lectura o escritura de un bloque de datos de un archivo, DDS lo mapea en el espacio de difusión de datos DDS dentro del rango del Terabyte que le corresponde al archivo. Los datos son leídos desde el disco y son almacenados en forma automática en el espacio EDL del nodo. Por ejemplo, en la Figura 5.8 tenemos un *cluster* formado por 8 nodos de cómputo con memoria y disco cada uno. Los nodos 2, 3, 4 y 5 son utilizados adicionalmente como nodos de E/S en donde es almacenado

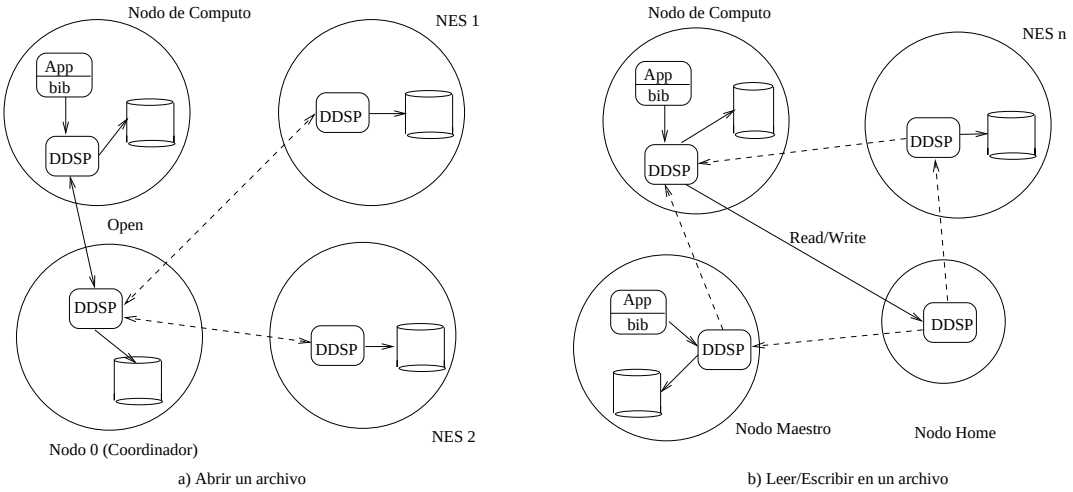

Figura 5.7: Operaciones de acceso a un archivo en FSDDS.

un archivo llamado *mult*. También se observa un espacio de difusión común a todos los nodos en donde se mapea el archivo. En el *cluster* corremos una aplicación en donde un proceso es ejecutado en cada uno de los 8 nodos de cómputo. En este ejemplo observamos la secuencia de las transacciones realizadas cuando el proceso 0 solicita leer el bloque 6 del archivo `mult` (secuencia 1). El tamaño del bloque es igual al tamaño del *stripe*. Si DDS no encuentra el bloque localmente, entonces transmite una solicitud de lectura al nodo *home* (secuencia 2). En este ejemplo el nodo *home* y el nodo de E/S es el mismo para este bloque. El nodo de E/S (NES_4) lee el bloque desde el disco (secuencia 3) y lo lleva a la memoria en el espacio de difusión dentro del nodo (secuencia 4). Cuando el bloque está en la memoria, DDS lleva una copia del bloque hacia el nodo 0 (secuencia 5) y lo almacena en su memoria. En subsecuentes accesos, el bloque es accedido desde la memoria bajo la difusión de datos de DDS. El usuario no requiere mover los datos entre los módulos de memoria de los nodos y los bloques de los archivos son mapeados al espacio de direcciones de la aplicación, haciendo transparente su uso.

Cuando un bloque de datos de un archivo está mapeado en la memoria del espacio de difusión de DDS y es modificado por un nodo de procesamiento, el bloque permanece en la memoria del nodo de procesamiento que lo accedió hasta que es solicitado para lectura o escritura por otro nodo o es reemplazado para liberar espacio de la memoria del nodo. Así, los bloques se moverán entre las memorias de los nodos evitando en lo posible que

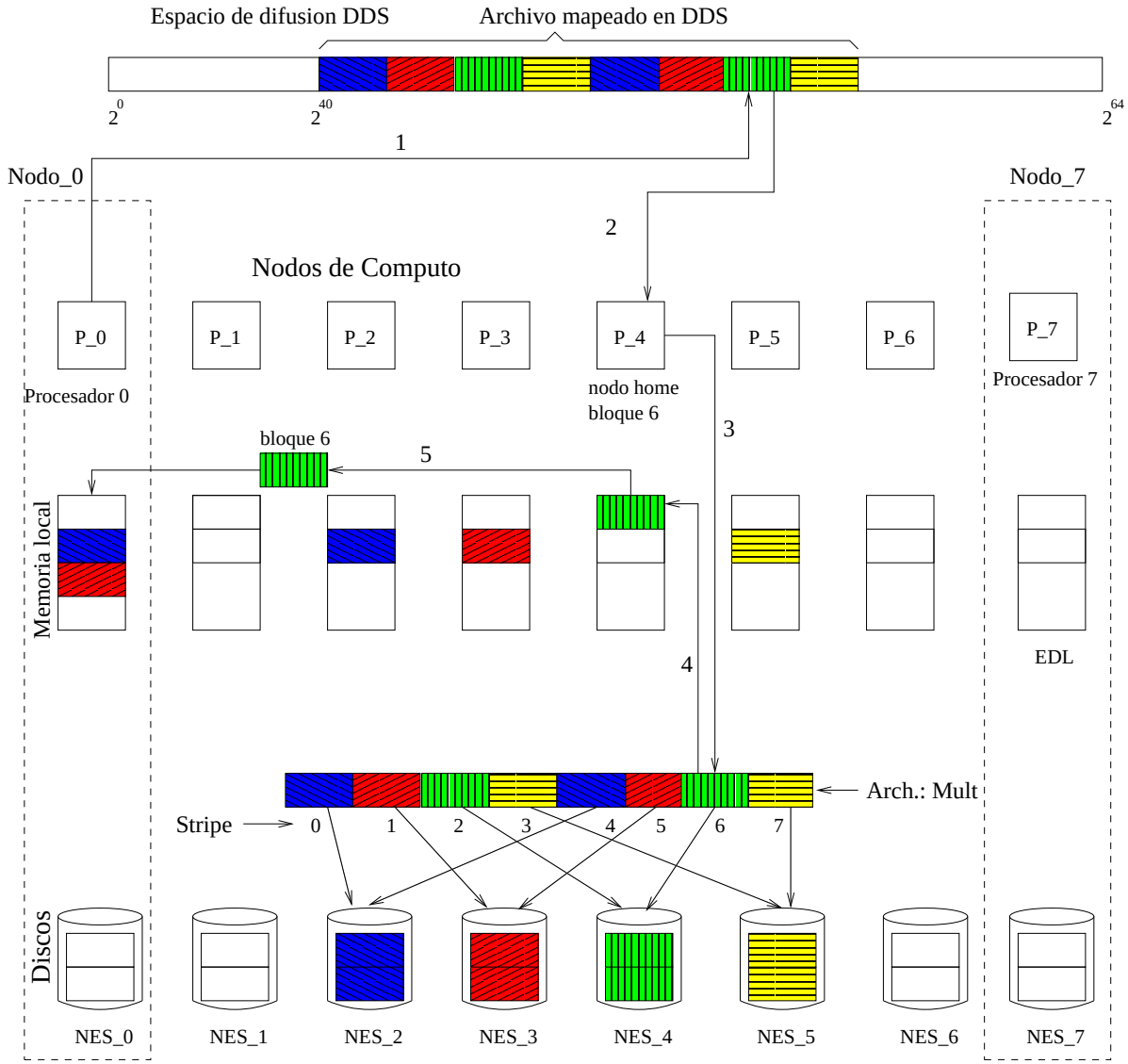

Figura 5.8: Ejemplo del mapeo de un archivo en FSDDS.

sean regresados al disco del nodo de E/S. Los bloques serán llevados y almacenados en el disco del nodo de E/S hasta que son reemplazados para liberar espacio en la memoria o hasta que el archivo sea cerrado en FSDDS por todos los nodos participantes en su acceso.

El manejo de los archivos requirió de algunos cambios del protocolo de difusión de datos, principalmente referentes a la política de reemplazo. En cada nodo, cuando la memoria está llena, el bloque menos recientemente utilizado es seleccionado y una acción es tomada de acuerdo a su estado. Si el estado es compartido, el bloque es descartado. Si su estado es exclusivo y el bloque pertenece a un arreglo (no un archivo), el bloque es intercambiado (*swaped*) dentro de un archivo temporal local. Si es exclusivo y el bloque pertenece a un archivo FSDDS, el bloque es enviado a su nodo de E/S, a menos que el nodo actual sea ese nodo de E/S, en cuyo caso el bloque es *swaped* dentro del archivo local FSDDS. Si el estado del bloque es maestro compartido y pertenece a un arreglo, el bloque es enviado a su nodo home, a menos que el nodo actual sea el nodo home, en cuyo caso el bloque es enviado a otro nodo seleccionado aleatoriamente. Si el estado del bloque es maestro compartido y pertenece a un archivo FSDDS, el bloque es *swaped* dentro de un archivo temporal local o si el nodo actual es su nodo de E/S es *swaped* a su archivo local FSDDS.

Los nodos de E/S también funcionan como nodos *home* para el archivo de datos que almacenan en sus discos, aún si no funcionan como nodos de cómputo. Para determinar el nodo *home* de un bloque, primero se determina si el bloque pertence a un arreglo o a un archivo. Para este propósito se utilizan los TBs asignados a los archivos. De esta manera, la dirección de cada bloque determina si una función *hash* o el archivo de metadatos sean utilizados para obtener el número del nodo *home*.

5.5. Acceso a datos

Cuando un bloque de un archivo es solicitado por un nodo de cómputo, primero se transmite una solicitud al nodo *home*. El *home* a su vez determina qué nodo tiene una copia maestra del bloque para retransmitirle la solicitud. Inicialmente, al crear o abrir un archivo en FSDDS los nodos de E/S también son los nodos maestros de los bloques de datos del archivo. Cuando el nodo de E/S recibe la solicitud de acceso del

bloque, primero verifica si el bloque se encuentra almacenado localmente. Si el bloque se encuentra localmente y está libre, el nodo de E/S transmite una copia del bloque a quién lo solicitó. Si el bloque no está en el nodo, se buscará una copia en otro nodo. Si el bloque está ocupado por otro nodo se esperará a que el bloque sea liberado por ese nodo; mientras tanto, la solicitud del nodo de cómputo se ingresará a una cola de espera.

Los detalles de la lectura y la escritura de un bloque de un archivo en FSDDS son presentados en las siguientes secciones.

5.5.1. Lectura de un bloque de un archivo en FSDDS

Cuando un proceso de la aplicación solicita un bloque de datos de un archivo en FSDDS para lectura, el protocolo de DDS busca el bloque de datos en el directorio local del nodo (la Figura 5.9 muestra la secuencia de las transacciones de una lectura de un bloque). Si el bloque es encontrado en el directorio local en un estado válido (compartido, maestro o exclusivo), DDS devuelve a la aplicación la dirección del bloque de datos para que realice su acceso. El bloque puede no estar en la memoria pero si en disco, en cuyo caso es primero leído desde el disco y almacenado en la memoria. Una vez que el bloque está en memoria, la dirección de ésta es dada a la aplicación.

Si el bloque no se encuentra en el directorio local del nodo de cómputo, este nodo transmite una solicitud de acceso al nodo *home* (secuencia 1) y espera la respuesta desde el nodo maestro del bloque o desde un nodo de E/S. Cuando el nodo *home* recibe una solicitud de lectura de un bloque, realiza una búsqueda en su directorio *home* para identificar al nodo maestro del bloque. Cuando el maestro del bloque es identificado, el *home* le retransmite la solicitud de acceso del bloque (secuencia 2). El nodo maestro puede ser un nodo de E/S o cualquier nodo de cómputo si el bloque ya ha sido accedido y modificado por uno de estos nodos. El maestro, al recibir la solicitud del bloque lo buscará en su directorio local. Si el bloque se encuentra en la memoria se transmite una copia de este bloque al nodo de cómputo que solicitó el bloque, si el bloque no se encuentra en la memoria, primero se procede a leer el bloque desde el disco y después se transmitirá una copia del bloque al nodo de cómputo (secuencia 3). El nodo de cómputo recibe el bloque, lo almacena en memoria y le pasa la dirección al proceso de la aplicación. Este proceso de almacenamiento en memoria y darle la dirección de esta memoria a la

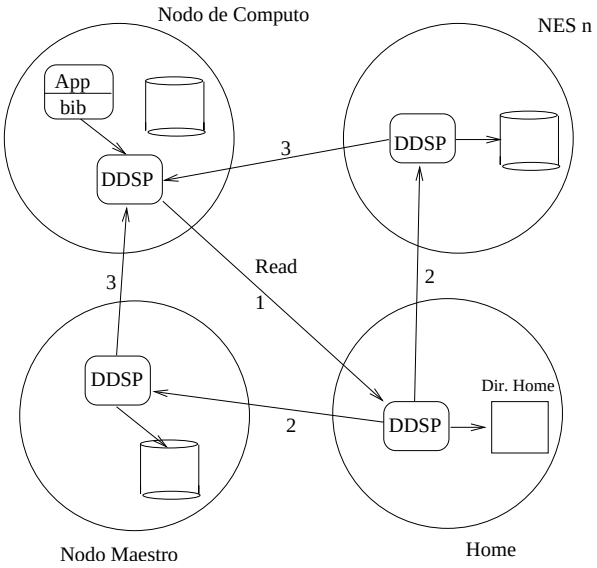

Figura 5.9: Lectura de un bloque en FSDDS, mostrando la secuencia o pasos de las operaciones.

aplicación es el mismo al realizado en la difusión de datos (sección 4.5.1). Así como el manejo de los estados del bloque y la inserción de los compartidores del bloque en la cola respectiva. La diferencia fundamental del acceso de un bloque de un archivo y un bloque definido en memoria es cuando el bloque se accede por primera vez por un nodo de cómputo. En este caso, el bloque del archivo inicialmente está almacenado en el disco del nodo de E/S y el bloque de un arreglo está almacenado en la memoria del nodo maestro.

5.5.2. Escritura de un bloque de un archivo en FSDDS

Cuando un proceso de la aplicación desea escribir en un bloque de datos, transmite una solicitud de lectura exclusiva del bloque al proceso DDSP en el nodo local (la Figura 5.10 muestra la secuencia de las transacciones de una escritura de un bloque). El protocolo de DDS busca el bloque de datos en el directorio local. Si el bloque es encontrado en el directorio local en estado exclusivo y se encuentra almacenado en la memoria, DDS regresa la dirección de la memoria del bloque a la aplicación. En otro caso, si el bloque está almacenado en el disco local en estado exclusivo, el bloque primero

es leído desde el disco y almacenado en memoria y después la dirección de la memoria del bloque de datos es dado a la aplicación. Si el bloque está en estado compartido, se envía una solicitud de lectura exclusiva al nodo *home* (secuencia 1 de la Figura 5.10a) y se espera por el reconocimiento del *home* de que la copia ha sido puesta como exclusiva. Cuando el *home* recibe la solicitud del bloque busca en el directorio *home* la identificación del nodo maestro del bloque y checa quién tiene una copia del bloque. El *home* transmite una solicitud de invalidación de la copia del bloque (secuencia 2) al nodo maestro del bloque y a los nodos que tienen una copia (también llamados nodos compartidores del bloque). Cuando el *home* recibe todos los reconocimientos de invalidación de todas las copias del bloque (secuencia 3), transmite un reconocimiento de invalidación al nodo de cómputo (secuencia 4).

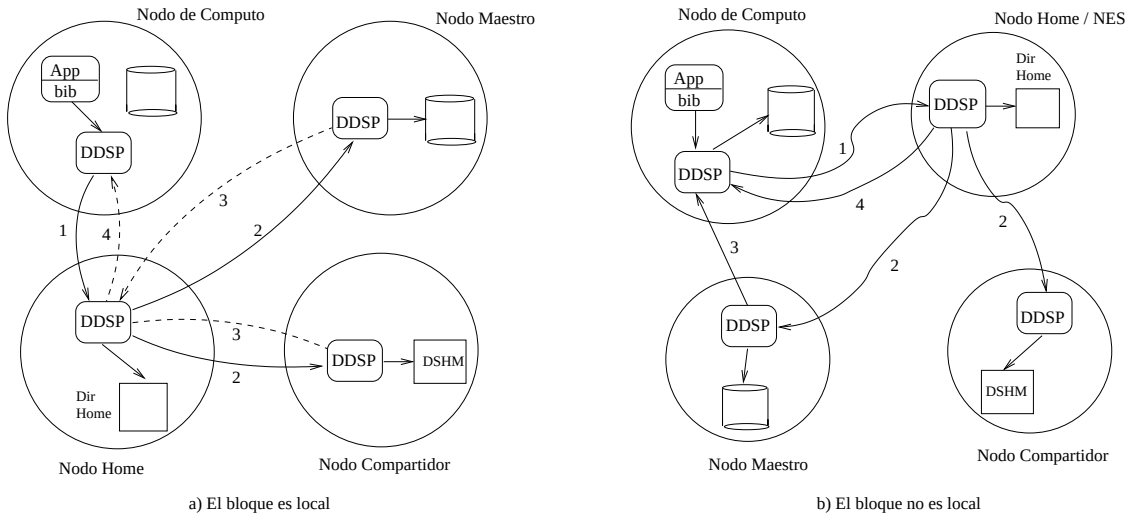

Figura 5.10: Escritura de un bloque en FSDDS.

Si el bloque no se encuentra en el directorio local (Figura 5.10b), DDS envía una solicitud de lectura exclusiva al nodo *home* (secuencia 1) y se espera por la copia maestra del bloque. Cuando el *home* recibe la solicitud del bloque busca en el directorio *home* la identificación del nodo maestro del bloque y checa quién tiene una copia del bloque. El *home* transmite la solicitud de lectura exclusiva al nodo maestro y además transmite una solicitud de invalidación de la copia del bloque a los nodos que tienen una copia (secuencia 2). Cuando el maestro recibe la solicitud del bloque lo busca en su directorio local y cuando lo encuentra le transmite la copia maestra al nodo de cómputo que

solicitó el bloque (secuencia 3) e invalida su copia. Cuando el nodo maestro invalidó su copia le transmite un reconocimiento al *home* indicándole que transmitió el bloque al nodo de cómputo y ha invalidado su copia. Cuando el *home* recibe los reconocimientos de invalidación de todas las copias del bloque y la invalidacón del nodo maestro, transmite un reconocimiento de invalidación al nodo de cómputo (secuencia 4). Cuando el nodo de cómputo recibe el reconocimiento desde el *home* cambia el estado de su copia a exclusivo y se pasa la dirección del bloque a la aplicación para su acceso.

5.6. Interfaz de programación

En esta sección presentamos como *FSDDS* facilita el desarrollo de aplicaciones paralelas de E/S intensiva en *clusters*. Desarrollamos una interfaz de programación de aplicaciones (API, siglas del inglés Application Programming Interface) de E/S intensiva u *out-of-core* para utilizar FSDDS.

Para acceder los archivos de datos los procesos de una aplicación interactúan con FSDDS por medio de la biblioteca del cliente. Esta biblioteca contiene un conjunto de funciones para el manejo de los archivos: `DDS_Open` para crear y/o abrir un archivo FSDDS, `DDS_Close` para cerrar el archivo FSDDS; así como para leer y escribir bloques de datos de los archivos con `DDS_Read` y `DDS_Write`, respectivamente. Cuando los procesos ejecutan operaciones tales como abrir, crear o cerrar archivos, utilizan las funciones de la biblioteca para comunicarse con el proceso DDSP localizado en su nodo local dentro del *cluster*. Cuando un archivo es abierto o creado, se obtiene un descriptor del archivo que utilizamos de manera semejante a un descriptor de archivo de UNIX. Este descriptor es utilizado en las operaciones de lectura y escritura de un bloque del archivo, como también para cerrar el archivo.

El API de FSDDS además tiene funciones similares a las funciones de UNIX/POSIX para realizar lecturas y escrituras de bloques de datos de los archivos en las cuales el usario debe especificar una dirección de memoria en donde será almacenado el bloque solicitado (este espacio es diferente del espacio de DDS). Estas funciones son: `DDS_FRead` y `DDS_FWrite`.

El API de FSDDS también puede ser utilizado con múltiples interfaces de programación de aplicaciones, por ejemplo: la misma API de UNIX/POSIX [46], la API de

PVM y de MPI-IO. En todas estas APIs, la comunicación con los nodos de E/S, con los nodos maestros y con los nodos *home* es manejada de manera transparente dentro de la implementación del API.

Con el API de FSDDS, el usuario no tiene que preocuparse por la ubicación de un bloque. No se necesita especificar un nodo fuente en particular para escribir o leer un bloque de datos, ni se preocupa por la capacidad de almacenamiento en memoria; lo que facilita la programación. En FSDDS el usuario sólo tiene que especificar el espacio de difusión deseado para almacenar los bloques de datos de los archivos y definir los descriptores de archivo para el manejo de los archivos. Este espacio de difusión se crea en el sistema al definir la estructura de datos `DDS` (especificada en el capítulo 4) con variables (tipo arreglo) que representen a los archivos. Estas variables serán compartidas por todos los nodos del *cluster* (ver detalles en la sección 4.6.2). Los bloques de datos de los archivos son llevados desde el disco a este espacio de difusión cuando se acceden por primera vez. En este espacio los bloques son manejados como arreglos de memoria compartida distribuida dentro del *cluster* y se acceden como si fuesen locales en el nodo de cómputo.

5.6.1. Funciones para el manejo de los archivos

FSDDS presenta un conjunto de funciones incluidas en una biblioteca que permiten al usuario, crear, abrir y cerrar archivos que son particionados y distribuidos en los discos de varios nodos de un PC-cluster. Incluye también un conjunto de funciones para posicionarse dentro del archivo y acceder a los datos en forma de bloques. Para usar estas funciones dentro de una aplicación, el usuario debe incluir el archivo `pdds_proto.h` en el programa de la aplicación. En este archivo se encuentran las declaraciones de las definiciones de las funciones de FSDDS.

A continuación presentamos la sintaxis, la descripción y el valor de retorno de las funciones para el manejo de los archivos de FSDDS.

- *int DDS_Open(const char *pathname, int flags, struct dds_filestat *fstat).*

  La función `DDS_Open` crea y/o abre un archivo en FSDDS cuyo nombre del archivo es la cadena de caracteres apuntada por *pathname*. Los demás parámetros de esta función son: las banderas de acceso del archivo (*flags*) y un apuntador a la estruc-

tura que especifica el patrón de particionamiento del archivo (*fstat*). Los valores empleados para el parámetro *flags* son `O_RDONLY, O_WRONLY y O_RDWR` los cuales solicitan abrir el archivo para sólo lectura, sólo escritura o lectura/escritura respectivamente. Estos valores pueden usarse en conjunto con `O_CREAT, O_APPEND, O_NONBLOCK o O_DELAY y O_SYNC`.

Valor de retorno: `DDS_Open` devuelve un descriptor del archivo que es utilizado en las funciones de lectura y escritura de DDS (`DDS_Read` y `DDS_Write`), así como al cerrar el archivo.

- *int DDS_Read(int fd, int64_t offset, int ndatos, int type, void *adrblk).*

  La función `DDS_Read` realiza una solicitud de lectura de un bloque de datos en un archivo FSDDS. El bloque puede estar almacenado en el archivo o en la memoria de algún otro nodo que corra DDS. El archivo es especificado por el parámetro *fd*, el cual es el descriptor del archivo obtenido con `DDS_Open`. La dirección inicial del bloque de datos dentro del archivo es dada por *offset*, el número de datos a leer es *ndatos* y el tamaño de cada dato es especificado por *type*.

  Valor de retorno: `DDS_Read` devuelve un valor positivo cuando la operación es exitosa e indica que el bloque ha sido almacenado en la memoria local del nodo asignada al espacio de difusión de DDS. La dirección de la memoria en donde se almacenó el bloque es puesta en el apuntador *adrblk*. Cuando el bloque es almacenado en la memoria su estado es puesto como compartido y es amarrado a esta memoria y permanecerá ahí hasta que sea liberado. Cuando existe un error en la lectura y el bloque no ha sido almacenado en la memoria local del nodo devuelve un valor negativo.

- *int DDS_UnRead(int fd, int64_t offset, int ndatos, int type, void *adrblk).*

  La función `DDS_UnRead` permite que un bloque de un archivo sea liberado y pueda ser reemplazado o movido por DDS hacia otro nodo o el disco si se necesita espacio en memoria para almacenar otro nodo. El bloque es de tamaño *ndatos*, donde cada dato es de tipo *type* y el parámetro *offset* especifica su posición dentro del archivo. Al ejecutarse esta función, FSDDS no garantiza que el bloque se encuentre en la memoria local del nodo especificada en el apuntador *adrblk*.

Valor de retorno: `DDS_UnRead` devuelve un valor positivo cuando la operación es exitosa o un valor negativo cuando existe un error.

- *int DDS_Write(int fd, int64_t offset, int ndatos, int type, void *adrblk).*

  La función `DDS_Write` realiza una solicitud de lectura exclusiva de un bloque de un archivo FSDDS y es utilizada para escribir o modificar los datos de ese bloque. El archivo es especificado por su descriptor de archivo con el parámetro *fd*. `DDS_Write` utiliza los mismos parámetros que `DDS_Read`. Esto es, la dirección inicial del bloque de datos dentro del archivo es dada por *offset*, el número de datos a leer es *ndatos* y el tamaño de cada dato es especificado por *type*. La diferencia fundamental es que con esta función el bloque es puesto como exlusivo y no hay más copias del bloque en el *cluster* cuando retorna la función. De la misma manera que `DDS_Read` esta función garantiza al usuario que los datos sean almacenados en la memoria local del nodo y no serán movidos de ésta.

  Valor de retorno: `DDS_Write` devuelve un valor positivo cuando la operación es exitosa e indica que el bloque ha sido almacenado en la memoria local del nodo asignada al espacio de difusión de DDS. La dirección de la memoria en donde se almacenó el bloque es puesta en el apuntador *adrblk*. Cuando existe un error en la lectura y el bloque no ha sido almacenado en la memoria local del nodo devuelve un valor negativo. Cuando el bloque es almacenado en la memoria su estado es puesto como exclusivo y es amarrado a esta memoria y su control es otorgado al proceso que lo solicitó. A partir de este momento el proceso que solicitó el bloque puede accederlo y modificarlo y ningún otro nodo podrá acceder el bloque hasta que sea liberado por el proceso, ni podrá ser reemplazado o movido para liberar espacio en memoria.

- *int DDS_UnWrite(int fd, int64_t offset, int ndatos, int type, void *adrblk).*

  Esta función permite que un bloque leído con anterioridad utilizando `DDS_Write` sea liberado y que pueda ser copiado, reemplazado o movido por DDS hacia otro nodo o hacia el disco. El parámetro *fd* especifica el descriptor del archivo, el bloque es de tamaño *ndatos* donde cada dato es de tipo *type* y es localizado dentro del archivo en la posición dada por *offset*. Al ejecutarse esta función, FSDDS no garantiza que el bloque se encuentre en la memoria local del nodo especificada en

el apuntador *adrblk* o se encuentre en un estado exclusivo.

Valor de retorno: `DDS_UnWrite` devuelve un valor positivo cuando la operación es exitosa o un valor negativo cuando existe un error.

- *int DDS_Fread(int filedes, uint64_t offset, char *buf, int32_t bsize).*

  La función `DDS_Fread` lee *bsize* elementos de datos desde el archivo dado por el descriptor *filedes*. Los datos son leídos desde el archivo en la posición del apuntador dado por *offset* y son almacenados en memoria en la dirección dada por *buf*. Si el valor de *offset* es negativo el *offset* del archivo no se mueve.

  Valor de retorno: `DDS_Fread` devuelve un valor positivo cuando la operación es exitosa o un valor negativo cuando existe un error.

- *int DDS_Fwrite(int filedes, uint64_t offset, char *buf, int32_t bsize).*

  Esta función escribe *bsize* elementos de datos en el archivo dado por el descriptor *filedes*. Los datos son leídos desde la dirección de memoria dada por *buf* y son escritos en el archivo en la posición dada por *offset*. Si el valor de *offset* es negativo el *offset* del archivo no se mueve.

  Valor de retorno: `DDS_Fwrite` devuelve un valor positivo cuando la operación es exitosa o un valor negativo cuando existe algún error.

- *int DDS_Lseek(int filedes, off_t offset, int whence).*

  Esta función reposiciona el *offset* dentro del archivo *filedes* de acuerdo a la directiva *whence*: `SEEK_SET, SEEK_CUR, SEEK_END`.

  Valor de retorno: `DDS_Lseek` devuelve un valor positivo cuando la operación es exitosa o un valor negativo cuando existe un error.

- *int DDS_Close(int fd).*

  Al finalizar el trabajo de cómputo con los bloques de datos de un archivo, el archivo puede cerrarse con la función `DDS_Close` para que los datos sean llevados por FSDDS de manera transparente al nodo de E/S correspondiente. Esta función sólo utiliza como parámetro al descriptor del archivo obtenido al crear o abrir el archivo.

Valor de retorno: `DDS_Close` devuelve un valor positivo cuando la operación es exitosa o un valor negativo cuando existe un error.

5.6.2. Estructuras de datos del patrón de particionamiento en FSDDS

En FSDDS, el usuario puede especificar un patrón de distribución de los archivos antes de crearlos o abrirlos o dejar que el sistema utilize una configuración base. Sin embargo, la mejor distribución de los archivos de datos es dependiente de cada aplicación, por lo que es conveniente que el usuario pueda especificar la más adecuada basado en los patrones de acceso de la aplicación. El patrón de distribución es especificado utilizando la estructura `dds_filestat`. Dentro de esta estructura se define el nodo base, el número de nodos de E/S, el tamaño del *stripe* y el tamaño del bloque. La estructura `dds_filestat` es declarada como sigue:

```
struct dds_filestat {
    int32_t     base;         /* número del node base */
    int32_t     pcount;       /* número de nodos de E/S */
    int32_t     ssize;         /* tamaño del stripe */
    int32_t     soff;          /* no usado por ahora */
    int32_t     bsize;         /* tamaño del bloque */
};
```

Por ejemplo si deseamos especificar el patrón de distribución de la Figura 5.5. Utilizamos la siguiente definición.

```
struct dds_filestat filestat={2, 4, 64*1024, -1, 16*1024};
```

5.6.3. Ejemplo de programación con FSDDS

En esta sección presentamos el uso de FSDDS, mostrando su facilidad y la similitud en la programación de aplicaciones *in-core* y *out-of-core*. La similitud en la programación es mostrada al desarrollar la versión paralela *out-of-core* de la suma de dos matrices a partir de su versión *in-core* presentada en la sección 4.6.

Suma de matrices

La versión *out-of-core* de la suma de dos matrices $A + B = C$ utilizando archivos FSDDS se muestra en la Figura 5.11. En esta aplicación, las matrices son distribuidas en los nodos de E/S por filas, asignando a cada proceso N/p filas consecutivas de las matrices, donde N es el número de filas de las matrices y p es el número de procesos.

Para almacenar y distribuir las matrices en archivos se define un patrón de particionamiento de los archivos con la estructura `dds_filestat` (línea 10), en la cual se especifica el nodo de E/S base, el número de nodos de E/S en donde se almacenarán los archivos, el tamaño del *stripe* y el tamaño del bloque de datos.

Al ejecutar la aplicación, cada proceso inicia a MPI (por ahora sólo lo utilizamos para el arranque de los procesos), obtiene el número de procesos ejecutados y su identificación (líneas 17 a 19). `DDS_Init` debe ser llamado antes de manejar los archivos FSDDS; `DDS_Init` establece la comunicación con el proceso DDSP, el cual asigna la memoria caché (DDS) para almacenar los datos compartidos e inicializa los directorios local y *home* (línea 21).

Para usar los datos de los archivos en FSDDS, los archivos primero deben ser creados y/o abiertos mediante la función `DDS_Open` (líneas 22 a 24). `DDS_Open` utiliza como parámetros el nombre del archivo, las banderas de creación del archivo y el patrón de distribución del archivo especificado en la estructura `dds_filestat`. `DDS_Open` retorna el descriptor de archivo que será utilizado para acceder los bloques de datos del archivo. Con esta función los datos del archivo son automáticamente mapeados dentro de DDS.

En esta suma de matrices cada procesador calcula N/p filas de C. Antes de usar los datos, cada nodo debe ganar acceso hacia el dato, a través de la llamada `DDS_Read o DDS_Write`. Cuando estas funciones retornan, los datos ya están en la memoria del procesador y permanecerán ahí hasta que la función correspondiente `DDS_UnRead o DDS_UnWrite` sea utilizada (de la misma manera que bajo DDS). En estas funciones se utiliza como parámetro al descriptor de archivo obtenido al crear o abrir el archivo con la función `DDS_Open`, el *offset* del bloque dentro del archivo y el tamaño del bloque (ver líneas 31 a 33 y 37 a 39).

Los datos se acceden a través de apuntadores que son especificados en las funciones `DDS_Read` y `DDS_Write`. En el ejemplo los apuntadores `shmem_A`, `shmem_B` y `shmem_C` son asociados a las matrices A, B y C respectivamente. Los apuntadores son actualizadas por

```
 1 #include <dds.h>
 2 #define  filename1     "matriz_A"
 3 #define  filename2     "matriz_B"
 4 #define  filename3     "matriz_C"
 5 #define  NPROCS        16                    /* número de procesadores */
 6 #define  ROWS          10000                 /* número de filas en las matrices */
 7 #define  COLUMNS       10000                 /* número de columnas en las matrices */
 8
 9       /* Particionamiento de los archivos: base, pcount, ssize, soff, bsize*/
10 struct dds_filestat filestat={0,NPROCS,(ROWS*COLUMNS*DDS_LONG)/NPROCS,0,COLUMNS*DDS_LONG};
11
12 int  fa, fb, fc;                             /* Definición de los descriptores de archivo */
13 int  *shm_A, *shm_B, *shm_C;
14
15 main() {
16
17   MPI_Init(&argc,&argv);                     /* inicializando MPI */
18   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
19   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
20
21   DDS_Init(NULL, NULL, &myid);               /* inicializando DDS */
22   fa = DDS_Open(filename1, O_RDWR | O_CREAT, &filestat);   /* abrir los archivos */
23   fb = DDS_Open(filename2, O_RDWR | O_CREAT, &filestat);
24   fc = DDS_Open(filename3, O_RDWR | O_CREAT, &filestat);
25
26   rows = ROWS/nprocs;                        /* determinar el número de filas por procesador */
27   offset = myid * (ROWS/nprocs);
28
29   for (r=0; r < rows; r++){
30      i = r + offset;
31      DDS_Write(fc, i*COLUMNS, COLUMNS, &shm_C);       /* ganando acceso */
32      DDS_Read(fa, i*COLUMNS, COLUMNS, &shm_A);        /* a datos compartidos */
33      DDS_Read(fb, i*COLUMNS, COLUMNS, &shm_B);
34      for (j=0; j < COLUMNS; j++){              /* usando dato compartido */
35          shm_C[j] = shm_A[j] + shm_B[j];
36      }
37      DDS_UnWrite(fc, i*COLUMNS, COLUMNS, &shm_C);    /* liberando datos */
38      DDS_UnRead(fa, i*COLUMNS, COLUMNS, &shm_A);
39      DDS_UnRead(fb, i*COLUMNS, COLUMNS, &shm_B);
40   }
41   DDS_Close(fa);                            /* cerrar los archivos */
42   DDS_Close(fb);
43   DDS_Close(fc);
44
45   DDS_Finalize();                           /* terminar DDS y MPI */
46   MPI_Finalize();
47 }
```

Figura 5.11: Ejemplo del modelo de programación de FSDDS: suma de matrices.

el proceso DDSP de acuerdo a la dirección del bloque solicitado en la función `DDS_Read` o `DDS_Write` y a la localización actual en donde el dato es almacenado en la memoria local del nodo.

Al finalizar de usar los bloque de datos, los archivos son cerrados por cada proceso de la aplicación que abrió el archivo (líneas 41 a 43). Los datos modificados y almacenados en la memoria de los nodos se llevan automáticamente por FSDDS hacia el nodo de E/S correspondiente especificado con el patrón de particionamiento y se depositan en su disco local.

5.7. Resumen

La ejecución de aplicaciones *out-of-core* requiere de un sistema de archivos paralelos para obtener un buen desempeño. En estos sistemas de archivos, los archivos son particionados en bloques y distribuidos en los nodos del *cluster*. El acceso de los bloques de estos archivos es realizado en forma directa desde el nodo de E/S lo cual degrada el desempeño de las aplicaciones. Por otro lado, la distribución de los bloques es fija y en muchas ocasiones no corresponde con los patrones de acceso de la aplicación. Para mejorar el desempeño, los accesos deben realizarse desde la memoria de los nodos que tienen una copia del bloque, sin que el usuario se preocupe por su localización y que se muevan automáticamente o se difundan hacia la memoria de quien los solicite.

En este capítulo se muestra la arquitectura y operación de nuestro sistema de archivos paralelo sobre DDS llamado FSDDS. En FSDDS los archivos se particionan en bloques y se distribuyen en los nodos de E/S. En este sistema de archivos los bloques se manejan como arreglos en memoria. Cuando los procesos de una aplicación acceden los bloques desde la memoria tiende a disminuir el número de accesos a disco. Al disminuir el número de accesos a disco se mejora el desempeño en la ejecución de las aplicaciones paralelas *out-of-core*.

FSDDS es diseñado como un sistema cliente-servidor con múltiples servidores de E/S llamados nodos de E/S. Los nodos de E/S son nodos del *cluster* en los que se almacenan los bloques de los archivos en sus discos locales. En estos nodos se ejecuta un proceso DDSP que atiende las solicitudes de E/S y controla el acceso de los bloques. El almacenamiento de los bloques en los nodos de E/S está dado por un patrón de

distribución del archivo.

El patrón de distribución del archivo determina las características de distribución física en disco. Este patrón de distribución está definido por varios parámetros de particionamiento que describen las características del archivo. Los parámetros de particionamiento del archivo son: el nodo base, el número de nodos de E/S, el tamaño del stripe y el tamaño del boque de datos. Este particionamiento permite que un archivo sea almacenado en varios nodos de E/S y se pueda acceder en paralelo por varios procesos dentro del *cluster*.

Cuando un bloque se accede por primera vez, el bloque es llevado del disco a la memoria en el espacio de difusión de DDS. En accesos posteriores, el bloque se accede desde la memoria y DDS lo difunde entre la memoria de los nodos que lo solicitan. El movimiento de los bloques desde la memoria de los nodos permite que los bloques sean mapeados en la memoria del *cluster*. El mapeo en memoria hace que los bloques de datos de los archivos sean vistos como arreglos en memoria. La programación y acceso de estos bloques puede realizarse con una interfaz de programación de aplicaciones de FSDDS (API de FSDDS).

La interfaz de FSDDS es similar a la interfaz de DDS utilizando las mismas funciones para las operaciones de lectura y escritura de DDS: `DDS_Read()`, `DDS_UnRead()`, `DDS_Write()` y `DDS_UnWrite()`.

Cuando un archivo es abierto bajo FSDDS, este es automáticamente mapeado dentro del espacio de dirección compartido de DDS. El primer Terabyte de DDS es reservado para los datos compartidos de los arreglos (es decir, no pertenecen a un archivo de datos); los siguientes Terabytes son usados uno para cada archivo que es abierto. De esta manera, los archivos y los datos compartidos se acceden en la misma forma y bajo el mismo protocolo de coherencia.

Con relación a los otros sistemas de archivos paralelos FSDDS tiene las siguientes ventajas: los archivos son manejados como arreglos en memoria al ser mapeados en el espacio de DDS sin importar la capacidad de almacenamiento de un nodo en particular, los datos pueden ser compartidos por todos los procesos y se mueven entre las memorias de los nodos por lo que no es necesario especificar un patrón de distribución de los archivos de datos y no es necesario implementar un protocolo de coherencia en los accesos concurrentes lo que facilita su programación.

.

Capítulo 6

Análisis y evaluación del desempeño de DDS

6.1. Introducción

Este capítulo presenta un análisis y una evaluación del desempeño de DDS y FSDDS. El análisis es basado en un modelo analítico que toma en cuenta: el tiempo de cómputo paralelo, el tamaño de la memoria caché, el número de las operaciones de E/S, el costo de las operaciones de acceso a datos y las comunicaciones involucradas entre tareas. La evaluación está basada en la ejecución de 3 aplicaciones bajo DDS y FSDDS con diferentes número de procesadores y diferentes tamaños del problema.

Nuestros resultados son comparados con MPI-IO/PVFS porque estas herramientas de programación paralela son las más usadas en el cómputo paralelo en *clusters*, su interfaz es similar a DDS por estar implementada toda en software y por esta razón sus interfaces son muy similares basadas en llamadas a funciones.

6.2. Modelo analítico

Para evaluar la eficiencia de una arquitectura paralela, la aceleración paralela (S) es ampliamente utilizada como una métrica de desempeño [42]. La aceleración de una aplicación paralela se define como la relación del tiempo secuencial de la aplicación (T_s) ejecutada en un solo procesador y el tiempo paralelo (T_p) obtenido al ejecutar la misma

aplicación en p procesadores. Tal que

$$S = \frac{T_s}{T_p} \tag{6.1}$$

En una situación ideal, la aceleración tiende a ser lineal con respecto al número de procesadores p. Sin embargo, en la práctica la aceleración lineal es difícil de obtener debido a varios factores: 1) la falta de paralelismo en la aplicación, 2) la degradación algorítmica y 3) la degradación de la arquitectura. El primer factor hace referencia a la parte no paralelizable del programa. El segundo factor es debido a los *overheads* del software que se tienen en la sincronización y en el particionamiento los cuales tienden a crecer con el número de procesadores. El tercer factor es debido a los *overheads* del hardware de comunicación y de la consistencia los cuales también tienden a incrementarse con el número de procesadores [83].

6.2.1. Tiempo de ejecución

El tiempo de ejecución (T) de una aplicación secuencial sobre un procesador normalmente es expresado como una función del tiempo del ciclo del reloj del procesador (T_{clock}), el contador de instrucciones (N_{inst}) y el número de ciclos por instrucción (CPI); dado por Henessy en [43]:

$$T = N_{inst} \times CPI_{avg} \times T_{clock} \tag{6.2}$$

Las tres cantidades de la ecuación 6.2 dependen de uno o más de los siguientes factores:

- el programa de la aplicación, es decir del algoritmo o del programa fuente ya que establecen el valor de N_{inst} y éste a su vez depende del compilador y de la arquitectura del conjunto de intrucciones,
- la arquitectura del procesador y su conjunto de instrucciones,
- la organización del sistema de memoria.

De lo anterior podemos decir que el tiempo de ejecución secuencial de una aplicación T_s depende de su tiempo de cómputo T_{comp} y del tiempo de los accesos a datos realizados

en la memoria o el disco del nodo T_{Local}. Considerando que los accesos a datos son locales y el cómputo es realizado por un solo procesador, el tiempo secuencial es dado por

$$T_s = T_{comp} + T_{Local} \tag{6.3}$$

Por otro lado, cuando tenemos un sistema paralelo con p procesadores, al ejecutar una aplicación paralela el tiempo T es en general reducido pues la carga de trabajo se comparte entre los procesadores. En este caso, T es conocido como el tiempo de ejecución paralela T_p. En un sistema paralelo con p procesadores, si el tiempo de ejecución secuencial de la aplicación es T_s y la carga de trabajo es dividida equitativamente entre los procesadores, entonces decimos que $T_p < T_s$ para $p > 1$.

Si asumimos que cada nodo tiene la misma configuración procesador-memoria y que el tiempo de comunicación entre los procesadores no influye en el tiempo de ejecución de la aplicación, podemos decir que el tiempo de ejecución paralela es optimizado para p procesadores. Tal que

$$T_p = T_s/p \tag{6.4}$$

Sin embargo, el tiempo de ejecución paralela (en un multiprocesador) podemos decir que consiste principalmente de dos componentes: el cómputo y la comunicación. Estos componentes son debidos al tiempo de cómputo (T_{comp}) realizado por las tareas de la aplicación y el tiempo de comunicación utilizado para compartir datos y sincronizar las tareas (T_{shared}) [83].

En un sistema multiprocesador con memoria compartida distribuida con p procesadores, el tiempo de ejecución de una aplicación paralela puede ser expresado como:

$$T_p = T_{comp}(p) + T_{shared}(p) \tag{6.5}$$

Por lo que la aceleración de un programa paralelo de la aplicación en un sistema con p procesadores puede ser expresada como:

$$S = \frac{T_s}{T_p} = \frac{T_{comp} + T_{Local}}{T_{comp}(p) + T_{shared}(p)} \tag{6.6}$$

Al ejecutar una aplicación paralela los datos compartidos pueden accederse localmente y/o remotamente desde otro nodo. El tipo de acceso (local o remoto) depende de

la organización de la memoria y de la distribución de los datos. Por lo tanto, el tiempo de ejecución compartido (T_{shared}) depende de la cantidad de estos accesos (local o remoto) y del tiempo en cada uno. Entonces, para p procesadores tenemos

$$T_{shared}(p) = \underbrace{N_{sh\_local} \times T_{sh\_local}}_{T_{Local}(p)} + \underbrace{N_{sh\_remoto} \times T_{sh\_remoto}}_{T_{comm}(p)} \tag{6.7}$$

donde

N_{sh_local} es el número de accesos a datos compartidos que son encontrados localmente.

T_{sh_local} es el tiempo para acceder localmente datos compartidos.

N_{sh_remoto} es el número de accesos a datos (o bloques) compartidos que requieren un acceso remoto.

T_{sh_remoto} es el tiempo para acceder un bloque compartido remotamente.

El número de accesos remotos realizados por cada procesador dependerá de la cantidad de datos que le son asignados para procesar (N_p) y del tamaño de la memoria local (M) que está disponible para almacenar los datos. Si la memoria local del nodo no es suficiente para almacenar todos los datos que le corresponden al nodo, el nodo dividirá los datos en bloques de tamaño M y cada bloque requerirá un acceso remoto si los bloques están en otro nodo. El número de estos accesos depende además del tamaño del problema N y del número de procesadores. De esta manera, asumiendo el caso ideal en el que la distribución de datos es equitativa en todos los procesadores y por cada bloque se requiere un acceso, obtenemos

$$N_{sh\_remoto} = \lceil \underbrace{(N/p)}_{N_p} / M \rceil \tag{6.8}$$

Cuando se requiere un acceso remoto, se transmite una solicitud de acceso a los nodos que tienen una copia maestra de los datos. El número de solicitudes a datos remotos R_p por un procesador depende del tamaño m de cada mensaje, del tamaño de un bloque de datos B y del tipo de acceso (por ejemplo, acceso por filas o columnas); si asumimos que para $m = B$ tenemos

$$R_p = (\lceil M/B \rceil) \times N_{sh\_remoto} = \lceil N_p/B \rceil \tag{6.9}$$

Si consideramos que el tipo de acceso es dado por un factor de distribución γ, la expresión anterior quedaría como

$$R_p = \lceil N_p/B \rceil \times \gamma \tag{6.10}$$

Podemos observar en la expresión anterior que para optimizar el número de solictudes es necesario que M sea un múltiplo de B.

Cuando los datos son almacenados remotamente es necesario que los procesos que deseen acceder estos datos se sincronizen y que estos datos sean transmitidos a través de la red de comunicación a quien los solicitó. El tiempo de comunicación T_{comm} de un procesador (bajo una aplicación paralela en un *cluster* con p procesadores) dependerá del número de accesos remotos y del tiempo de estos accesos. Este tiempo es definido como:

$$T_{comm}(p) = N_{sh\_remoto} \times T_{sh\_remoto} \tag{6.11}$$

En general, T_{sh_remoto} depende del tiempo de comunicación debido al tiempo de transferencia de los datos (T_{send}) y a la sincronización entre los procesadores (T_{sync}) que intervienen en la transferencia. El tiempo de comunicación de una aplicación paralela incluye todas las solicitudes de acceso remoto. Sumando el tiempo de estas solicitudes tenemos

$$T_{comm}(p) = \sum_{i=1}^{R_p} T^i_{send} + \sum_{i=1} T^i_{sync} \tag{6.12}$$

La ecuación 6.12 expresa el tiempo de comunicación debido a la transferencia de datos de uno de los procesadores en un *cluster* con p procesadores, pero pueden ocurrir varios pares de *send-sync* simultáneamente entre diferentes procesadores.

El tiempo de transferencia T_{send} de un mensaje de m bytes de longitud, depende del ancho de banda del canal de comunicación β_{comm}, del tiempo de enlace o latencia l y de la contención del canal de comunicación c. Esto es

$$T_{send} = l + m/\beta_{comm} + c \tag{6.13}$$

En DDS para ejecutar una aplicación paralela en un *cluster*, las operaciones de acceso a datos consisten en solicitudes de lectura y escritura de bloques de datos y también de la sincronización. Entonces, T_{comm} puede ser expresado como

$$T_{comm}(p) = \sum_{i=1}^{R_p} T^i_{acc} + \sum_{i=1}^{p} T^i_{sync} \tag{6.14}$$

Una lectura o escritura (a un dato remoto) implica hacer una búsqueda del bloque en el directorio *home* (T_{search}) para localizar la copia maestra y traer una copia desde el nodo maestro (T_{send}). De esta manera, el tiempo de acceso remoto (T_{acc}) depende del tiempo de búsqueda del bloque en el directorio *home* (T_{search}), del tiempo de acceso a memoria (T_{mem}) y del tiempo de transferencia de los datos (T_{send}). Una escritura, a diferencia de la lectura, también puede incluir el tiempo para invalidar (T_{inv}) las copias compartidas del bloque. Para analizar estas operaciones, obtenemos el tiempo de lectura (T_{read}) y el tiempo de escritura (T_{write}) de un bloque en forma independiente.

Para una lectura $T_{acc} = T_{read}$,

$$T_{read} = T_{search} + T_{mem} + T_{send} \tag{6.15}$$

Para un escritura $T_{acc} = T_{write}$,

$$T_{write} = T_{search} + T_{mem} + T_{send} + T_{inv} \tag{6.16}$$

Un acceso remoto ocurre cuando un dato no se encuentra en la memoria local de un nodo. Se inicia al fallar su búsqueda en el directorio local. Entonces se determina quién es el nodo *home* del bloque y se le transmite una solicitud de acceso t_{req}. El nodo *home* checa que el bloque esté libre en el estado adecuado (maestro, compartido o exclusivo). Si el bloque está ocupado la solicitud se ingresa a una cola de espera y permanece ahí hasta que el bloque sea liberado (T_{wait}). Por otro lado, si el bloque está libre y no existe una copia del bloque en el nodo *home*, se retransmite la solicitud al nodo maestro. Si consideramos que el tiempo de transmisión de una solicitud es mucho mayor que al tiempo realizado por el procesador para checar por el estado del bloque. Entonces, tenemos que

$$T_{search} = T_{req} \tag{6.17}$$

El protocolo TCP/IP puede utilizar bloques de hasta 64 Kbytes para comunicación entre nodos. Sin embargo, por restricciones del hardware en las ethernet [27], sólo maneja bloques de 1500 bytes para comunicarse entre nodos a través de la red. En DDS utilizamos bloques de 16 Kbytes por default, pero puede usarse un tamaño máximo de hasta 1 MByte. Este valor es ajustado de acuerdo al patrón de acceso de la aplicación.

Si consideramos los bytes del *header* de TCP/IP y los utilizados para checar la transmisión, en cada intercambio de datos se trasmiten 1448 bytes. Por lo que, el costo para transferir de 1 hasta 1448 bytes es el mismo. Entonces, si el tamaño de datos de una solicitud (sin datos de la aplicación) no es mayor a 100 bytes, podemos asumir que el tiempo de transferencia de una solicitud sea igual a la latencia. De esta manera tenemos el tiempo de una solicitud T_{req} como

$$T_{req} = \begin{cases} l & \text{para el mejor caso: cuando el bloque está en el } home \\ 2 \times l & \text{en otro caso: el bloque está en un nodo diferente al } home \end{cases} \quad (6.18)$$

El acceso a memoria de un bloque remoto podemos decir que está formado de dos componentes (por dos operaciones de acceso a memoria en nodos diferentes). La lectura en el nodo maestro y la escritura en el nodo solicitante. La lectura en el maestro depende básicamente del tiempo de acceso a la memoria y del tamaño del bloque. La escritura en el nodo solicitante depende del espacio de difusión local del nodo solicitante. Si hay espacio suficiente para almacenar el bloque, el tiempo de escritura depende del tiempo de acceso de la memoria (t_{acc}). Si no hay espacio en la memoria, el tiempo de escritura depende del tiempo de reemplazo de un bloque para almacenar el bloque solicitado. En el análiss siguiente, para simplicar el análisis sólo consideramos el tiempo de reemplazo del bloque debido a que es mucho mayor que el tiempo de acceso a memoria, haciendo practicamente $t_{acc} = 0$.

Por lo tanto, decimos que el tiempo de acceso a la memoria t_{mem} es el tiempo que un nodo tarda en ejecutar el reemplazo de un bloque t_{reemp} para almacenar el dato accedido por su tarea local. t_{mem} depende del tamaño de los datos del problema, así como de la memoria utilizada por el nodo y de la cantidad de datos almacenados a este nodo. Si permitimos que M_x represente el espacio máximo de la memoria local que el nodo x puede proporcionar a la aplicación (o la tarea ejecutándose en el nodo) para sus demandas de memoria. Cuando una tarea i ejecutándose en el nodo solicita un espacio de memoria m_i para almacenar un bloque de datos y todos los bloques solicitados por esta tarea pueden ser almacenados en M_x. Entonces, podemos decir que, la latencia de acceso a memoria puede ser desechada.

Sin embargo, si el procesador no tiene suficiente espacio de memoria, una falla de bloque ocurrirá en forma frecuente durante la ejecución de la tarea. Cuando ocurre una falla de bloque los mecanismos del manejo de la memoria de DDS son lanzados para

ejecutar el reemplazos de varios bloques a la vez y liberar suficiente espacio en la memoria para evitar que se genere una falla de bloque en forma inmediata. Cuando ya existe espacio en la memoria el nuevo bloque es almacenado en este espacio de la memoria. Los accesos a memoria de la tarea serán retrazados debido a la latencia de ejecutar estos reemplazos de bloque. Básicamente, el tiempo total requerido para ejecutar un reemplazo de bloque puede ser dividido dentro de dos componentes discretos. El primer componente es el tiempo utilizado para búscar (t_{seek}) los bloques de datos a reemplazar. En DDS utilizamos el algoritmo del reloj con segunda oportunidad o RELOJ2 [98]. En este algoritmo se tiene una lista circular en forma de reloj con todos los bloques que se encuentran almacenados en la memoria del nodo. Una manecilla apunta hacia el bloque más antiguo. Los bloques en la lista son marcados con un bit de referencia R para indicar si han sido accedidos con anterioridad. Cuando un bloque se accede el bit R es puesto a 1. Al ocurrir un fallo de bloque se inspecciona el bloque al cual apunta la manecilla, si su bit $R = 0$ se retira de la memoria, se inserta el nuevo bloque en su lugar en el reloj y la manecilla avanza una posición, si $R = 1$ la mancecilla avanza una posición y el bit se limpia, esto continúa hasta encontrar un bloque con $R = 0$. Cuando el bloque es encontrado intercambiamos este bloque a disco o hacia otro nodo. El segundo componente de interés es el tiempo requerido para intercambiar el bloque de datos solicitado por la tarea con el que se seleccionó a reemplazar (t_{swap}), para moverlo desde el disco o desde la memoria de otro nodo a la memoria física del nodo que solicitó el bloque. Por lo que, el tiempo de acceso a la memoria de un nodo x es expresado como:

$$t^x_{mem} = \begin{cases} 0 & \text{si } M_x \geq \sum_{i \in x} m_i \\ \underbrace{P^x_r (t_{seek} + t_{swap})}_{t_{reemp}} & \text{si } M_x < \sum_{i \in x} m_i \end{cases} \tag{6.19}$$

En la ecuación 6.19, P^x_r es el número de bloques reemplazados por el nodo x para almacenar los bloques de datos requeridos por la tarea local, t_{seek} es el tiempo promedio utilizado por el nodo x en buscar un bloque de datos en el algoritmo del RELOJ2 y liberar el espacio ocupado por este bloque y t_{swap} es el tiempo promedio requerido para intercambiar el bloque nuevo hacia la memoria del nodo x.

Para simplificar nuestro análisis se considera que $t_{reemp} = 0$, debido a que los reemplazos no son realizados de manera contínua al reemplazarse varios bloques en una misma operación. De esta manera se puede hacer que $t^x_{mem} = 0$.

De la ecuación 6.16 podemos deducir que el costo de las invalidaciones, C, para el proceso x es dependiente del número de procesadores P que comparten un bloque, tal que

$$t_{inv}^{x} = \sum_{y=1, y \neq x}^{P} C_{xyk} \tag{6.20}$$

donde

$$C_{xyk} = \begin{cases} t_{rqi} & \text{si el nodo } x \text{ y el nodo } y \text{ comparten el bloque } k \\ 0 & \text{en otro caso.} \end{cases}$$

t_{rqi} = el costo de una solicitud de invalidación de un bloque de datos.

Cuando más de un nodo desea acceder a un bloque para escritura, deben ser sincronizados por DDS para darle el control a un solo nodo a la vez. La sincronización en DDS es realizada utilizando barreras, normalmente para mantener un orden de las operaciones de lectura y escritura de un bloque. Las barreras son realizadas utilizando colas FIFO (del inglés, First Input First Output) para el acceso a cada bloque. El tiempo de sincronización depende del número de barreras ejecutadas antes de un acceso de lectura (b_r) y de las barreras de las escrituras (b_w). Si consideramos a t_b el tiempo de una barrera, entonces tenemos que

$$T_{sync\_DDS} = (b_r + b_w) \times t_b \tag{6.21}$$

Sustituyendo los términos anteriores en la ecuación 6.14 tenemos, para una lectura:

$$T_{comm}(p) = \lceil \frac{N_p}{B_D} \rceil (T_{req} + T_{send}) + (b_r + b_w) \times t_b \tag{6.22}$$

Para una escritura:

$$T_{comm}(p) = \lceil \frac{N_p}{B_D} \rceil (T_{req} + t_{inv} + T_{send}) + (b_r + b_w) \times t_b \tag{6.23}$$

Hasta ahora hemos considerado que los datos son almacenados en memoria al ejecutar una aplicación paralela en un PC-cluster utilizando DDS. Al ejecutar una aplicación paralela de E/S intensiva en un sistema paralelo, los datos de la aplicación normalmente son almacenados en archivos. Los archivos son divididos en bloques y almacenados en

los discos de varios nodos de E/S. El tiempo de E/S (t_{io}), debido al acceso de bloques en disco, afecta también al tiempo de ejecución de un solo procesador T_1:

$$T_1 = T_{comp}(1) + T_{Local}(1) + t_{io}(1) \tag{6.24}$$

Para un sistema paralelo con p procesadores, el tiempo de ejecución paralela de la aplicación paralela es expresado como:

$$T_p = T_{Local}(p) + T_{shared}(p) + t_{io}(p) \tag{6.25}$$

Asi, la aceleración ahora es expresada en términos de la E/S como:

$$S = \frac{T_1}{T_p} = \frac{T_{comp}(1) + T_{Local}(1) + t_{io}(1)}{T_{comp}(p) + T_{shared}(p) + t_{io}(p)} \tag{6.26}$$

De la expresión anterior, el tiempo de ejecución paralela ahora es descrito como

$$T_p = T_{comp}(p) + T_{Local} + T_{comm}(p) + t_{io}(p) \tag{6.27}$$

El tiempo de E/S (t_{io}) de una aplicación depende del número de solicitudes de E/S (R_{rqio}), es decir el número de accesos de lectura y escritura de los bloques que no están en la memoria de los nodos y del número de las solicitudes a disco (R_{dsk}). Por tanto, t_{io} puede expresarse en función de estos parámetros:

$$t_{io} = R_{rqio} \times t_{rqio} + R_{dsk} \times t_{dsk} \tag{6.28}$$

donde t_{rqio} es el tiempo requerido por una solicitud de E/S y t_{dsk} es el tiempo que tarda una solicitud a disco en ser atendida.

Por otro lado, en un sistema de archivos paralelos, un archivo de tamaño F es almacenado en N_d discos o nodos de E/S considerando que cada nodo tiene un solo disco. Si el archivo es particionado en bloques de tamaño B_D y cada nodo tiene N_B bloques, entonces el tamaño del archivo puede ser expresado como sigue:

$$F = N_d \times (B_D \times N_B) \tag{6.29}$$

Si un bloque de datos es solicitado por una tarea, es posible que esté almacenado en la caché del sistema de archivos del nodo. Por lo que la solicitud puede ser satisfecha desde la memoria caché, disminuyendo el número de solicitudes a disco. Así, el número de datos

(F_c) que fueron leídos en un ciclo anterior y están almacenados en la memoria caché de DDS, depende del tamaño de la memoria caché (M_c). Si el número máximo de bloques que pueden estar almacenados en la memoria caché es N_c, entonces $M_c = N_c \times B_D$. Si recordamos que el tamaño de la memoria caché de cada nodo es M_x, entonces para p procesadores M_c también puede ser expresado como $M_c = M_x \times p$. Cuando las memorias cachés de todos los nodos están totalmente ocupadas, decimos que $F_c = M_c$, tal que

$$F_c = N_c \times B_D \tag{6.30}$$

Si consideramos que el tamaño de una solicitud de un bloque de datos sea R_D, el número de solicitudes a disco (R_{dsk}) puede expresarse en función de R_D. Sin embargo, también es necesario tomar en cuenta el patrón de distribución del archivo en disco. Por ejemplo, cuando un bloque es almacenado en varios nodos en forma de *round-robin* o traslapado se generará un acceso paralelo hacia los nodos que tienen el bloque almacenado en disco por cada solicitud de E/S. Si γ es una constante de distribución del archivo. Entonces tenemos:

$$R_{dsk} = \lceil (N/R_D) \rceil \times \gamma \tag{6.31}$$

Si consideramos el tamaño de la caché y hacemos que el tamaño de una solicitud (R_D) sea igual al tamaño de un bloque de datos (B_D), podemos obtener el número de solicitudes por procesador, $R_{dsk}(p)$, como sigue:

$$R_{dsk}(p) = \lceil (N_p - F_c)/B_D \rceil \times \gamma \tag{6.32}$$

donde $N_p = N/p$ y $B_D = R_D$.

Si el tamaño de la memoria caché de DDS en cada nodo fuera nula o los bloques se acceden por primera vez, entonces $R_{dsk}(p)$ puede ser expresado como

$$R_{dsk}(p) = \lceil N_p/B_D \rceil \times \gamma \tag{6.33}$$

Si analizamos las ecuaciones 6.32 y 6.33 puede observarse que $R_{dsk}(p)$ es menor cuando consideramos un espacio en la caché. Por lo que, se concluye que a mayor tamaño de la caché menor es el número de operaciones de E/S.

En DDS, el número de solicitudes u operaciones de E/S a disco (R_{dsk}) potencialmente puede disminuir respecto a las emitidas por el programa paralelo de una aplicación

debido a la utilización de una caché adicional. Esta caché está formada por las memorias de todos los nodos del *cluster* y se le ha identificado como el espacio de difusión de datos (E_D). Su tamaño está integrado por la memoria compartida utilizada en cada nodo del *cluster*. Así, las transferencias de los bloques de datos pueden ser realizadas desde la memoria del nodo que tiene el último dato actualizado. Aunque los bloques son accedidos desde el disco en su primer acceso, en subsecuentes accesos los bloques son tomados desde la memoria. De esta manera, los bloques son mantenidos en la memoria mientras el espacio de ésta sea suficiente para almacenar el dato. En otro caso, el bloque nuevamente es llevado al disco. Si decimos que el tamaño de la caché en DDS es igual a E_D y sustituimos a F_c por E_D en la ecuación 6.32, entonces, el valor de R_{dsk} también será dependiente del espacio de difusión utilizado en DDS. Considerando lo anterior, el tiempo de acceso de la E/S también es dependiente del tamaño del espacio de difusión E_D, esto es debido a que el número de operaciones de E/S puede ser reducido cuando los accesos a datos son realizados desde la memoria. Por lo que, en DDS el número de solicitudes a disco por procesador se puede expresar como:

$$R_{dsk}(p) = \lceil Np/B_D + (Np - E_D)/B_D \rceil \times \gamma \tag{6.34}$$

El primer componente de la ecuación 6.34 (Np/B_D) representa a los datos que son leídos desde disco en un primer acceso y el segundo componente ($(Np-E_D)/B_D$) representa a los subsecuentes accesos a datos que son realizados desde la memoria caché debidos al patrón de acceso de la aplicación.

Como fué mencionado antes, el tamaño del espacio de difusión del sistema está formado por el espacio de la memoria compartida de todos los nodos. Si el tamaño de la memoria compartida por procesador es M_{shr}, entonces el tamaño del espacio de difusión en memoria física para p procesadores es expresado como sigue:

$$E_D = \sum_{i=1}^{p} M_{shr_i} \tag{6.35}$$

Asumiendo que el tamaño de M_{shr} es el mismo en cada nodo, entonces:

$$E_D = p * M_{shr} \tag{6.36}$$

Además, si consideramos que algunos bloques están almacenados en la memoria de los nodos y otros bloques están en los discos. Para obtener el número de solicitudes

que se realizan a disco necesitamos hacer una estimación de las solicitudes que fallan al buscar un bloque en memoria. Por lo cual obtenemos la tasa de acierto (del inglés, *hit rate*) de los bloques que están en memoria y su correspondiente tasa de error (del inglés, *miss ratio*).

Para estimar las fallas o errores, primero considerar un *cluster* con p procesadores que no utilizan una caché. También asumir que el conjunto de datos son uniformemente distribuidos entre los procesadores y no hay localidad de datos. Así, cuando el número de procesadores se incrementa la tasa de acierto se disminuye. Por lo que la tasa de acierto promedio $a(p)$ con p procesadores es expresada en la siguiente ecuación:

$$a(p) = 1/p \tag{6.37}$$

La correspondiente tasa de error $e(p)$ sería el valor inverso de $a(p)$, la cual es expresada como:

$$e(p) = \frac{p-1}{p} \tag{6.38}$$

Sin embargo, en la práctica la mayoría de las aplicaciones poseen algún grado de localidad y los *clusters* normalmente proveen algún medio para explotar esta localidad. Por lo que, la ecuación 6.38 puede ser reescrita usando un coeficiente L:

$$e(p) = L(\frac{p-1}{p}) \tag{6.39}$$

Donde, el coeficiente L es una medida aproximada del grado de localidad obtenida por una aplicación ejecutada en un *cluster* usando FSDDS. L toma valores desde 0 (localidad completa, $e(p) = 0$) hasta 1 (sin localidad). Es decir, para $L = 0$ implica que los datos se encuentran en la memoria y no existe error en su búsqueda dentro de esta memoria y para $L = 1$ indica que los datos no son encontrados en memoria. Si se considera el espacio de difusión E_D y la cantidad de datos solicitados por la aplicación debido a su patrón de acceso N_a, la probabilidad de encontrar un bloque en memoria es dada por $P_r = E_D/N_a$. Entonces L es expresada como:

$$L = 1 - \frac{E_D}{N_a} \tag{6.40}$$

En la ecuación anterior puede observarse que al aumentar el espacio de difusión, el valor de L tiende a 0, lo que implica que el error disminuye aumentando la probablidad de encontrar el bloque en memoria.

De lo anterior, si tomamos en cuenta que E_D es usado para determinar el error e al buscar los bloques de datos y el error se obtiene a partir de todos los bloques solicitados. Entonces, en la ecuación 6.34 se elimina el valor E_D tal que al considerar el valor de e, tenemos:

$$R_{dsk}(p) = \lceil \frac{N_p}{B_D} + (\frac{N_p}{B_D})e \rceil \times \gamma \tag{6.41}$$

Finalmente, el tiempo de acceso a disco t_{dsk} en un nodo con k discos locales, puede expresarse como:

$$t_{dsk} = max_{i=1}^{k}(L_i + S_i) + \frac{B_D}{k * r} \tag{6.42}$$

donde L_i y S_i son la latencia y el tiempo de búsqueda respectivamente en el disco i y r es el tiempo de transferencia de un byte de datos desde el disco.

Si consideramos los parámetros obtenidos en la escritura debido a que genera más operaciones que una lectura y el tiempo de la E/S en la ecuación 6.27, tenemos

$$T_p = T_s/p + \lceil \frac{N_p}{B_D} \rceil (T_{req} + t_{inv} + T_{send}) + (b_r + b_w) \times t_b + t_{io} \tag{6.43}$$

Por lo tanto, de la ecuación 6.26, la aceleración de DDS es expresada como:

$$S = \frac{T_s + t_{io}(1)}{\frac{T_s}{p} + \lceil \frac{N_p}{B_D} \rceil (T_{req} + t_{inv} + T_{send}) + (b_r + b_w)t_b + t_{io}(p)} \tag{6.44}$$

Si analizamos a DDS, los bloques tienden a moverse hacia el *home* con lo cual se considera a $T_{req} = l$, por lo que la aceleración sería:

$$S = \frac{T_s + t_{io}(1)}{\frac{T_s}{p} + \lceil \frac{N_p}{B_D} \rceil (l + t_{inv} + T_{send}) + (b_r + b_w)t_b + R_{dsk}(p) \times t_{dsk}} \tag{6.45}$$

Evaluamos el desempeño de DDS en dos aplicaciones: una multiplicación de matrices y la Transformada Rápida de Fourier. Estas aplicaciones fueron seleccionadas porque son similares en dos aspectos: Primero, son de cómputo intensivo y, segundo, el paralelismo entre tareas es explotado utilizando un modelo SPMD. Un análisis del comportamiento del sistema es ejecutado para DDS sobre estas aplicaciones y el modelo teórico es utilizado para analizar el comportamiento de las aplicaciones al variar el tamaño de la caché, el tamaño del problema y el número de procesadores.

6.2.2. Ejemplo de la multiplicación de matrices

En la multiplicación de matrices (MM), cada proceso calcula un renglón distinto de la matriz de salida C (ver detalles del 2o. algoritmo en la sección 6.3.2 y la distribución de los datos en la Figura 6.3). El proceso principal espera hasta que todos los procesos han finalizado de leer la matriz de salida. El número de operaciones realizadas en la MM es n^3, n es la dimensión de la matriz. Por lo que el tiempo de ejecución secuencial está dado por $t_s = n^3 * t_{oper}$, donde t_{oper} es el tiempo de una operación, tal que $t_p = \frac{n^3}{P}$.

Las variables dependientes de la aplicación y del protocolo de DDS, que son especificadas en la expresión de la aceleración dada por la ecuación 6.45, son analizadas con el BMM para obtener los valores siguientes:

- $N_p = n^2/p$, considerando sólo una matriz.
- $t_{inv} = 0$, porque no existen escrituras a bloques compartidos por más de un nodo.
- $c = 0$, considerando que la red está libre de contención.
- $t_b = 0$, debido a que no hay competencia por bloques por lo que los procesadores no necesitan sincronizarse.
- $B_D = n$, el tamaño de un bloque es igual al número de elementos de un renglón de las matrices.
- $R_p = (n^2 - n^2/p)/n = n - n/p$
- $\gamma = R_p$, representa el patrón de acceso a datos de la aplicación.
- $e(p) = (1 - P_r)(\frac{p-1}{p}) = (1 - \frac{E_D}{N})(\frac{p-1}{p})$
- El número de operaciones de E/S es dado por:

  $R_{dsk}(p) = \lceil (n^2/p + n^2/p)/n + [(n^2/p)/n]e\gamma \rceil$

  Simplificando la expresión anterior resulta en:

$$R_{dsk}(p) = \lceil (n/p)(2 + e\gamma) \rceil \tag{6.46}$$

Si $E_D \geq 3n^2$ se obtiene el mejor valor de $R_{dsk}(p)$ porque todas las matrices son almacenadas en memoria, tal que $e = 0$. Si $E_D < 3n^2$, entonces $e \neq 0$. Por lo que tenemos las siguientes expresiones:

$R_{dsk}(p) = \lceil 2n/p \rceil$, para $e = 0$

$R_{dsk}(p) = \lceil (n/p)(2 + e\gamma) \rceil$, para $e \neq 0$

- $t_{dsk} = L + n/r$
- $t_{io}(1) = 3n^2/n \times (L + n/r) = 3n(L + n/r)$
- $t_{io}(p) = \frac{n}{p}(2 + e\gamma)(L + n/r) + \frac{n}{p}(L + n/r)$

  Analizando $t_{io}(p)$ para valores de $e = 0$ y $e \neq 0$:

  $t_{io}(p) = \frac{3n}{p}(L + n/r)$, para $e = 0$

  $t_{io}(p) = (\frac{3n}{p} + \frac{ne\gamma}{p})(L + n/r)$, para $e \neq 0$

Con los datos anteriores, la aceleración de la MM es dada por la siguiente ecuación:

$$S = \frac{n^3 + 3n(L + n/r)}{\frac{n^3}{p} + (n - n/p)(2l + \frac{n}{\beta_{comm}}) + (\frac{3n}{p} + \frac{ne\gamma}{p})(L + n/r)} \tag{6.47}$$

Simplificando la ecuación anterior e ignorando los términos menos significativos, la aceleración puede expresarse como:

$$S \cong \frac{n^2 + 3(L + n/r)}{\frac{n^2}{p} + (1 - 1/p)(2l + \frac{n}{\beta_{comm}}) + (\frac{3}{p} + \frac{e\gamma}{p})(L + n/r)} > 1 \tag{6.48}$$

El análisis en p de esta desigualdad indica que la siguiente condición tiende a ser verdadera para tener una aceleración mayor que 1.

$$p > \frac{n^2 + (3 + e\gamma)(L + n/r) - (2l + n/\beta_{comm})}{n^2 + 3(L + n/r) - (2l + n/\beta_{comm})} > 1 \tag{6.49}$$

Puede observarse que p es directamente proporcional al error y por consiguiente depende del tamaño del espacio de difusión del *cluster*.

Ahora procedemos a analizar el número de operaciones de E/S a partir de la ecuación 6.46 tomando los siguientes valores: Una matriz con $n = 16384$; $M_C = 256$ y 384 MBytes por nodo; $p = 4, 8, 16$; $B_D = 16384$.

$M_C = 256$ MB		
Procesadores	Lecturas	Escrituras
4	69632	4096
8	34816	2048
16	17408	1024

$M_C = 384$ MB		
Procesadores	Lecturas	Escrituras
4	17408	4096
8	5888	2048
16	2048	1024

Tabla 6.1: Lecturas y escrituras de la MM obtenidas con el modelo de FS/DDS para matrices de 16K * 16K

Los resultados mostrados en la Tabla 6.1 nos muestran un valor aproximado a los obtenidos experimentalmente en la sección 6.3.2. Existe una diferencia entre las lecturas de ambos resultados porque en el modelo no consideramos las operaciones de E/S debidas al algoritmo de reemplazo. Sin embargo, puede observarse que al aumentar el tamaño de la memoria caché el número de operaciones de E/S disminuye.

6.2.3. Ejemplo de la transformada rápida de Fourier

La Transformada Rápida de Fourier (FFT) [52] en 4 dimesiones se aplica a la matriz de una imagen degradada. El algoritmo es detallado en la sección 6.3.3 y la distribución de los datos de la matriz es mostrada en la Figura 6.5. El número de operaciones realizadas en la FFT es n^4, donde n es la dimensión de la matriz. Por lo que el tiempo de ejecución secuencial está dado por $t_s = n^4 * t_{oper}$, donde t_{oper} es el tiempo de operación de la FFT, tal que $t_p = \frac{n^4}{P}$.

Las variables especificadas en la expresión de la aceleración dada por la ecuación 6.45 que dependen de la aplicación y de DDS, son analizadas con el algorimtmo de la FFT para obtener los valores siguientes:

- $N = n^2$
- $n = 256$, el número de matrices de un solo renglón.
- $B_D = 1$, el tamanño de un bloque es igual a la matriz original.
- $t_{inv} = l$, se invalida sólo un bloque.
- $\gamma = R_p$, depende del patrón de acceso a datos.

- $R_p = n^2 - n^2/p$
- $e(p) = (1 - P_r)(\frac{p-1}{p}) = (1 - \frac{E_D}{N})(\frac{p-1}{p})$
- El número de operaciones de E/S es dado por:

$$R_{dsk}(p) = \frac{N}{p}(2 + e\gamma) = \frac{n^2}{p}(2 + e\gamma) \tag{6.50}$$

- $t_{dsk} = L + n/r$
- $t_{io}(1) = 2n^2(L + 1/r)$
- $t_{io}(p) = \frac{n^2}{p}(2 + e\gamma)(L + 1/r)$

Por consiguiente, la aceleración de la FFT es dada por la siguiente ecuación:

$$S = \frac{n^4 + 2n^2(L + 1/r)}{\frac{n^4}{p} + (n^2 - n^2/p)(l + 2l + \frac{1}{\beta_{comm}}) + \frac{n^2}{p}(2 + e\gamma)(L + n/r)} \tag{6.51}$$

Simplificando la ecuación anterior e ignorando los términos menos significativos, la aceleración puede expresarse como:

$$S = \frac{n^2 + 2(L + 1/r)}{\frac{n^2}{p} + (1 - 1/p)(3l + \frac{1}{\beta_{comm}}) + \frac{1}{p}(2 + e\gamma)(L + n/r)} \tag{6.52}$$

El análisis en p de esta desigualdad indica que la siguiente condición tiende a ser verdadera para tener una aceleración mayor que 1.

$$P > \frac{n^2 + (2 + e\gamma)(L + n/r) - (3l + 1/\beta_{comm})}{n^2 + 2(L + n/r) - (3l + 1/\beta_{comm})} > 1 \tag{6.53}$$

En la expresión anterior se observa que p es directamente proporcional al error y por consiguiente también depende del tamaño del espacio de difusión del *cluster*.

Ahora procedemos a analizar el número de operaciones de E/S a partir de la ecuación 6.50. Si tenemos los siguientes valores: 1) Para una imagen de 64×64 pixeles tenemos una matriz de 512 MBytes, $n = 128$ matrices, $B_D = 32$ KBytes. 2) Para una imagen de 128×128 pixeles tenemos una matriz de 6 GBytes, $n = 256$ matrices, $B_D = 128$ KBytes. En ambos casos utilizamos $M_C = 256$ y 384 MBytes por nodo y $p = 4, 8, 16$.

Al igual que en el ejemplo de la MM, los resultados mostrados en la Tabla 6.2 también nos muestran un valor aproximado a los obtenidos experimentalmente en la sección 6.3.3 y se observa que al aumentar la memoria caché se disminuyen las operaciones de E/S.

$M_C = 256$ MB			$M_C = 384$ MB		
Procesadores	Lecturas	Escrituras	Procesadores	Lecturas	Escrituras
4	45248	4096	4	4096	4096
8	20864	2048	8	2048	2048
16	9152	1024	16	1024	1024

Tabla 6.2: Lecturas y escrituras de la FFT obtenidas con el modelo analítico de FS/DDS

6.3. Evaluación experimental

Para realizar un mejor análisis del desempeño de DDS y FSDDS, hemos coleccionado algunos datos experimentales de varios aspectos utilizando un *cluster* de computadoras personales (PC-cluster) y varias aplicaciones paralelas de E/S intensiva. Analizamos y discutimos estos datos en las siguientes secciones y comparamos los resultados obtenidos con el modelo analítico mostrado en la sección 6.2. Describimos los experimentos realizados para validar la implementación de DDS y también describimos las aplicaciones usadas para coleccionar los datos experimentales. Los resultados obtenidos muestran el desempeño de DDS al ejecutar varias aplicaciones *out-of-core*.

6.3.1. Aplicaciones

Para evaluar el desempeño de DDS hemos ejecutado dos aplicaciones en el PC-cluster utilizando diferentes números de procesadores y diferentes parámetros del problema. Dos versiones de las aplicaciones fueron ejecutadas, una versión bajo DDS y otra versión bajo MPI-PVFS [69]. La versión de MPI-IO que fue utilizada es también conocida como ROMIO [100] y fue usada bajo nuestra versión de MPI con cómputos de datos *in-core y out-of-core*. Aunque sólo experimentamos con dos aplicaciones para evaluar a DDC, nuestras aplicaciones muestran diferencias significativas en el particionamiento de datos, los patrones de acceso y los requerimientos de espacio al desbordar la capacidad de la memoria. En la versión *out-of-core*, los datos son particionados por bloques y almacenados en forma de *round robin* en el espacio del disco de los nodos. Cada bloque es de tamaño n/p filas, donde n es el número de filas de cada arreglo y p es el número de procesadores usados en cada ejecución de la aplicación. Bajo DDS, la versión *out-of-core* de la aplicación es programada de manera similar a la versión *in-core*. No hay necesi-

dad de especificar el particionamiento de datos dentro de los discos. Estas aplicaciones son: una multiplicación de matrices (MM) y la Transformada Rápida de Fourier (FFT) (descritas más adelante).

Cabe señalar que, en la programación de aplicaciones *out-of-core* bajo MPI-PVFS, los programadores conocen la partición de los datos dentro del espacio del disco. También, los programadores saben cuándo leer de, y escribir en, disco los datos. Esto es, el programador sabe que todos los datos no caben en memoria al mismo tiempo y por lo tanto utiliza alguna memoria sólo como un *buffer* temporal. El número de solicitudes de E/S de esta manera es implícitamente definido por el programador. Bajo DDS, algunas lecturas y escrituras pueden ser satisfechas desde las copias almacenadas en las memorias de otros nodos; de aquí el número de solicitudes de E/S puede potencialmente ser reducido al aumentar la capacidad de la memoria utilizando otros nodos del *cluster*. De esta marea, pensamos que el *overhead* (sobrecarga) de las operaciones de E/S a disco es una métrica adecuada para evaluar el desempeño de DDS.

Así, nuestro objetivo es mostrar que el tiempo de ejecución de aplicaciones paralelas de E/S intensiva puede disminuir al utilizar DDS, al disminuir las operaciones de E/S.

También mostraremos que los bloques de datos se mantienen coherentes en la memoria cuando se escribe y lee al mismo bloque de datos por diferentes procesos.

El *cluster* está conformado por 16 computadoras personales (PC), con las siguientes características en cada nodo: procesador Intel Celeron a 1.7 GHz, 512 MB en memoria RAM y discos duros con una capacidad de almacenamiento desde 1 hasta 4 GBytes (con diferentes capacidades de almacenamiento en los nodos). Todos los nodos fueron interconectados por un Switch Fast Ethernet 3COM con 48 puertos. El sistema operativo fue Linux RedHat 9.0.

6.3.2. Desempeño de la multiplicación de matrices

La multiplicación de matrices (MM) se realiza, en general, con patrones de acceso predecibles y un particionamiento de datos estático. Implementamos 2 algoritmos de la MM y ejecutamos cada uno sobre 4, 8 y 16 procesadores. Así, para el primer algoritmo tenemos 6 ejecuciones: 2 (versiones: MPI-PVFS y DDS) por 3 (diferentes configuraciones de procesadores). El tamaño de datos de las matrices y el tamaño del bloque de acceso de datos de la MM es el mismo en cada configuración de los procesadores y bajo MPI-PVFS

y DDS.

Como se recordará, en la versión secuencial de la MM, $C = A * B$, cada elemento en la matriz resultante de C es formado multiplicando un renglón de A y una columna de B; se multiplica cada elemento del renglón con el correspondiente elemento de la columna (estos deben ser de la misma longitud) y sumando estos valores. En lenguaje C el algoritmo secuencial es mostrado a continuación:

```
for (i=0; i<NRA; k++) {
   for (j=0; j<NCB; j++) {
      for (k=0; k<NCA; k++) {
         C[i][j] += A[i][k] * B[k][j];
      }
   }
}
```

Donde: NCB es el número de columnas en la matriz B (y C), NRA es el número de renglones en A (y C) y NCA es el número de columnas en A (y renglones en B). La matriz C es inicializada a cero.

MM - Algoritmo 1

En nuestro primer algoritmo paralelo (MM-1), cada procesador realiza parte del cómputo de un renglón de C. Las matrices son distribuidas en los nodos de E/S asignando a cada procesador n/p renglones consecutivos de cada matriz; donde n es el tamaño de cada renglón y cada columna y p es el número de procesadores. Este particionamiento es conveniente porque los elementos en el mismo renglón de la matriz son almacenados consecutivamente en memoria. Por ejemplo para $n = 4$ y $p = 2$, la distribución puede observarse en la Figura 6.1.a (parte superior). El procesador 0 almacena los renglones 0 y 1 y el procesador 1 almacena los renglones 2 y 3 de cada matriz. El tamaño de un bloque de acceso de datos es igual al tamaño de un renglón de las matrices.

Este algoritmo para la multiplicación de matrices es implementado como sigue:

1. En el primer paso, los procesos leen $\lfloor \frac{n}{p} \rfloor$ renglones de B dentro de la memoria, tal que los renglones de todos los procesos conforman la totalidad de la matriz B

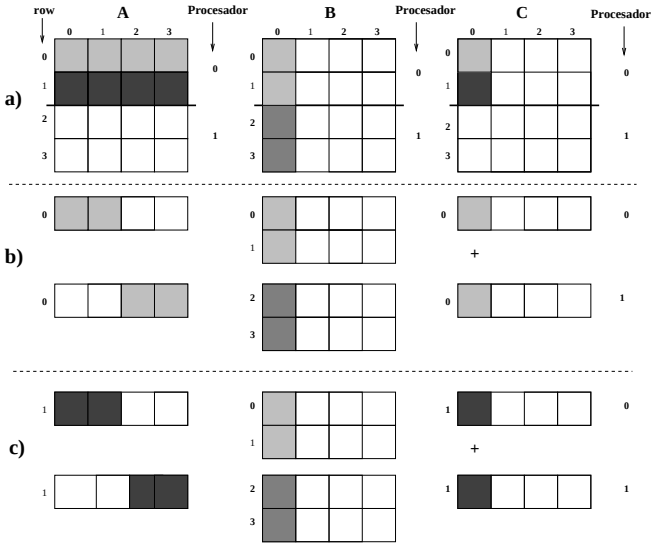

Figura 6.1: Patrones de acceso de E/S de la MM-1.

(ver Figura 6.1.b, parte media, central). Adicionalmente, todos los procesos leen el primer renglón de la matriz A dentro de su memoria.

2. Los datos presentes en la memoria de todos los procesos son suficientes para calcular el primer renglón de la matriz C. Cada proceso realiza una multiplicación parcial de todos los elementos de C, utilizando: $\lfloor \frac{n}{p} \rfloor$ elementos del renglón de A y $\lfloor \frac{n}{p} \rfloor$ elementos de una columna de B. Lo cual requiere una suma global de estos renglones parciales. Cuando la suma global es terminada, el proceso que tenía almacenado en disco el renglón de A, también almacenará en su disco el nuevo renglón de C.

3. A continuación, los procesos leen el siguiente renglón de A (Figura 6.1.c) y calculan el siguiente renglón de C.

4. El procedimiento se itera hasta que todos los renglones de C son calculados.

La matriz C es calculada en su totalidad de esta manera. En cualquier momento, cada procesador tiene en su memoria un renglón de la matriz A y C y n/p renglones de la matriz B.

Resultados de la MM - Algoritmo 1

En todos los experimentos, cada matriz es de $10000 * 10000$ elementos de enteros (de 4 bytes cada uno) generando matrices de 400 MBytes, 1.2 GBytes en total para las 3 matrices y el tamaño total de la memoria en 4, 8 y 16 procesadores es de 2, 4 y 8 GBytes respectivamente. Cada procesador tiene en cualquier momento n/p renglones de B, un renglón de A y uno de C. Al utilizar 4 procesadores, cada procesador tiene 100,080,000 Bytes de datos en la memoria, para 8 procesadores 50,040,000 Bytes y para 16 procesadores 25,020,000 Bytes. Por tanto, en cada configuración de procesadores el tamaño de los datos usados por cada procesador no desborda la capacidad de su memoria y no son requeridos accesos adicionales al disco. Bajo MPI-PVFS con operaciones individuales (no colectivas) de lectura/escritura (PVFS-I) y bajo FSDDS, el tamaño del *stripe* en todas la matrices fue N/p filas. También se ejecutó una versión de MPI-PVFS donde cada procesador usa operaciones colectivas de lectura/escritura (PVFS-C), utilizando un tamaño del *stripe* de N/p elementos de datos para únicamente la matriz A.

La tabla 6.3 muestra el número de solicitudes de lectura y escritura bajo PVFS-I, PVFS-C y FSDDS para MM-1. Ambas versiones de PVFS generan el mismo número de solicitudes de lectura porque sólo se diferencian en la clase de lectura que utilizan: cada operación de lectura es por la misma cantidad de datos. Bajo FSDDS, se observa que el número de operaciones de lecturas es menor debido a que algunas lecturas de las filas de la matriz A fueron satisfechas desde las copias almacenadas en la memoria de otros nodos. Todas las versiones generan el mismo número de solicitudes de escritura porque utilizan la misma unidad de escritura, una fila. Bajo PVFS-I/C, cada fila es escrita por un solo procesador una vez que es calculada. Bajo FSDDS, las filas de la matriz C son mantenidas en memoria tanto como sea posible; por consiguiente algunas de ellas son escritas al disco sólo cuando el archivo es cerrado.

MPI-PVFS-I/C			DDS		
Procesadores	Lecturas	Escrituras	Procesadores	Lecturas	Escrituras
4	12500	2500	4	5000	2500
8	11250	1250	8	2500	1250
16	10625	625	16	1250	625

Tabla 6.3: Lecturas y escrituras de la MM-1 con MPI-PVFS y FSDDS.

La Figura 6.2 muestra el tiempo de ejecución de la MM-1 bajo PVFS-I, PVFS-C y FSDDS. En todas las versiones se muestra casi el mismo tiempo de ejecución, aún cuando bajo FSDDS menos de la mitad de las solicitudes de lectura son generadas. La razón de esto es la memoria caché de los datos. Todos los procesadores acceden cada fila de la matriz A iniciando en la primer fila y una vez que finalizan el cálculo de los resultados parciales de la fila correspondiente en C, la fila es descartada; en contraste, cada procesador mantiene las filas de la matriz B que utiliza hasta finalizar el cómputo. Esto es, cada procesador está accediendo cada fila de cada matriz dentro de su memoria una sola vez, bajo PVFS y FSDDS. También esto implica que en PVFS algunas lecturas fueron satisfechas desde la memoria principal, más probable desde la caché de los nodos de E/S. FSDDS muestra un ligero mejor desempeño, sobre 8 y 16 procesadores, porque las versiones de PVFS utilizaron *MPI_Gather()* para coleccionar todos los resultados parciales para una fila de la matriz C y de esta manera incurren en un costo de sincronización, siendo mayor a mayor número de procesadores. Bajo FSDDS, cada procesador escribe exclusivamente sus resultados parciales tan pronto como ganan acceso a cada fila. En esta versión de FSDDS, los procesos cuando leen los renglones de A y de C, con `DDS_Read` y `DDS_Write` respectivamente, no necesitan sincronizarse globalmente; lo cual genera una sobrecarga menor por sincronización.

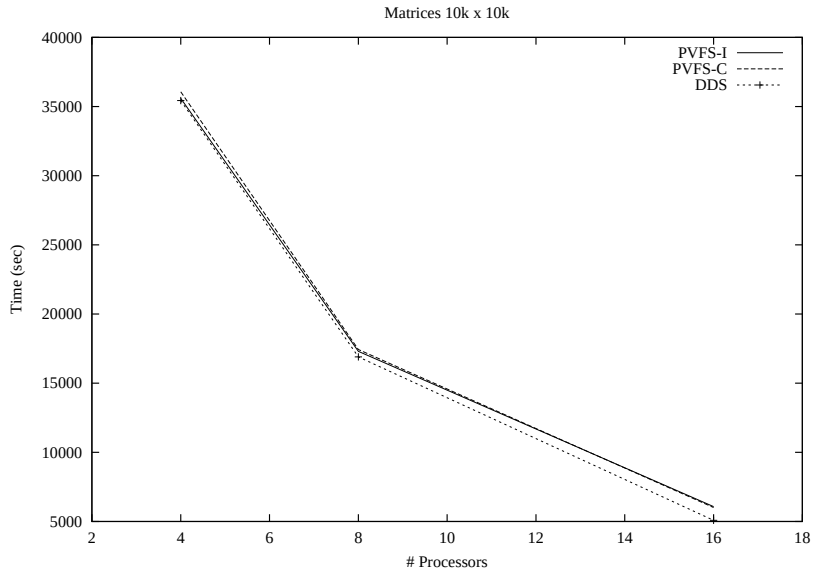

Figura 6.2: Tiempo de ejecución de la MM-1 con MPI-PVFS y DDS.

MM - Algoritmo 2

En el segundo algoritmo de la MM (MM-2) se cambió la distribución de los datos y su forma de acceso. La matrices A y C son almacenadas en forma de filas y la matriz B en forma de columnas (los elementos de una columna están almacenados en localidades de memoria consecutivas). Las matrices son distribuidas en los nodos asignando a cada procesador $\lfloor \frac{n}{p} \rfloor$ columnas consecutivas de la matriz B y $\lfloor \frac{n}{p} \rfloor$ filas de las matrices A y C. Este particionamiento permite que cada procesador calcule el valor total de un elemento de una fila de C. Por ejemplo para $n = 4$ y $p = 2$, la distribución puede observarse en la Figura 6.3.

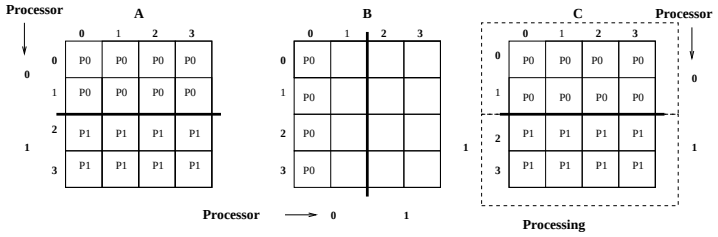

Figura 6.3: Distribución de las matrices - algoritmo 2.

En este algoritmo cada procesador lee $\lfloor \frac{n}{p} \rfloor$ filas de A, pero sólo $(n/p)/b$ filas a la vez, donde $b = 1, 2$ y 4 para $p = 16, 8$ y 4 respectivamente. Para calcular un renglón de C es necesario leer todas las columnas de B, b veces. De esta manera, en el primer paso cada procesador lee la primera columna de B y se multiplica por cada uno de las filas de A almacenadas en la memoria, para obtener los $\lfloor \frac{n}{p} \rfloor$ elementos correspondientes a la primer columna de C. A continuación, se lee la siguiente columna de B y se calcula la siguiente columna de C. Este procedimiento se repite hasta que todas las columnas de C son calculadas.

Resultados de la MM - Algoritmo 2

La tabla 6.4 muestra el promedio ($total/p$) del número de operaciones de E/S bajo MPI-PVFS y FSDDS para nuestro segundo algoritmo de la MM. Se observa que el número de operaciones de E/S es mayor para MPI-PVFS en todas las configuraciones de procesadores, debido a que en FSDDS algunas lecturas son satisfechas desde las copias almacenadas en la memoria de otros nodos. En este algoritmo se utilizaron matrices de

$16K * 16K$ elementos de enteros de 8 bytes cada uno, generando matrices de 2 GBytes, 6 GBytes en total para los arreglos, mientras que el tamaño total de la memoria es de 2 GBytes para 4 procesadores, 4 GBytes para 8 procesadores y 8 Gbyte para 16 procesadores. Así, las matrices se desbordan de memoria en las configuraciones de 4 y 8 procesadores. Cada procesador tiene n/p filas de A y de C y una columna de B en cualquier momento. Si el tamaño de una fila es de 128 KBytes, al utilizar 4 procesadores, cada procesador debe acceder 512 MBytes de datos de la matriz A y 512 MBytes de C, para 8 procesadores cada uno debe acceder 256 MBytes de A y 256 MBytes de C y para 16 procesadores 128 MBytes. Por lo tanto, el tamaño de los datos correspondientes en las configuraciones de 4 y 8 procesadores al utilizar las matrices A y C desbordan la capacidad de la memoria en cada procesador. De esta manera, en estas configuraciones son requeridos accesos adicionales al disco. Para 16 procesadores la capacidad de la memoria es suficiente para almacenar los datos accedidos y no son requeridos accesos adicionales al disco. En MPI-PVFS estos accesos son controlados por el usuario, el cual debe primero determinar el tamaño necesario de la memoria para leer las n/p filas de la matriz A. Si no hay suficiente memoria para almacenar las n/p filas de A, éstas son divididas en bloques de igual tamaño. El tamaño de este bloque es un múltiplo de n para al menos leer una fila de A y es menor que la memoria del nodo. Así, la matriz B será leída más de una vez debido a que la memoria es insuficiente para almacenar las filas de A necesarias para el cómputo de C. En este caso es posible que la memoria no sea aprovechada en su totalidad porque es controlada por el usuario, en cambio en FSDDS el manejo de la memoria es transparente al usuario haciendo un uso más eficiente de ésta. En FSDDS el número de operaciones de E/S es menor porque algunas solicitudes de datos son satisfechas desde las copias en memoria de los nodos remotos.

MPI-PVFS		
Procesadores	Lecturas	Escrituras
4	69632	4096
8	34816	2048
16	17408	1024

FSDDS		
Procesadores	Lecturas	Escrituras
4	20480	4096
8	6144	2048
16	2048	1024

Tabla 6.4: Lecturas y escrituras de la MM-2 con MPI-PVFS y FSDDS para matrices de 16K * 16K

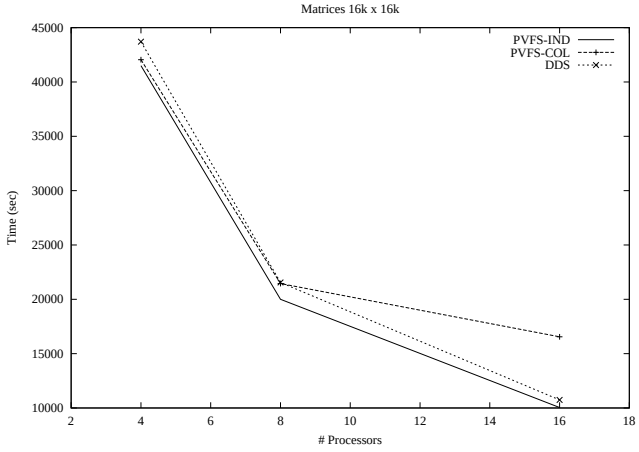

Figura 6.4: Tiempo de ejecución de la MM-2 con MPI-PVFS y FSDDS con una matriz de 16Kx16K enteros de 8 bytes

La Figura 6.4 muestra el tiempo de ejecución de MM-2 bajo MPI-PVFS y FSDDS, con E/S independiente (PVFS-IND) y con E/S colectiva (PVFS-COL). FSDDS y PVFS-IND muestran resultados similares en el tiempo de ejecución en todas las configuraciones de procesadores. Sin embargo, aún cuando en PVFS-IND y PVFS-COL se generaron el mismo número de operaciones de E/S, en PVFS-COL la sincronización debida a las operaciones colectivas genera un mayor tiempo de ejecución conforme el número de procesadores se incrementa, como es mostrado para 16 procesadores.

6.3.3. Desempeño de la transformada rápida de Fourier

La FFT es aplicada para restaurar imágenes degradadas o desenfocadas. Para una imagen de $N \times N$ pixeles es usada una matriz de $N \times N \times 8$ bytes. A partir de esta imagen, una *matriz de imágenes* es creada la cual corresponde a un proceso de autocorrelación. La matriz de imágenes contiene $M \times M$ imágenes y su tamaño es de $2N \times 2N \times (N \times N) \times 8 = (N^4) \times 32$ bytes. La FFT se aplica a la matriz de imágenes como sigue: El procesador 0 aplica la FFT a las primeras M/p filas y a las primeras M/p columnas (de imágenes) a lo largo de las filas en cada matriz imagen (1ra), a lo largo de las columnas en cada matriz imagen (2a), saltando a través de las filas en diferentes matrices imagen (3ra) y saltando a través de columnas de la misma manera (4a); el procesador 1 aplica la

FFT a las segundas M/P filas y a las segundas M/p columnas (de imágenes), ..., y así sucesivamente. La Figura 6.5 muestra la matriz de imágenes para $N = 2$ y su particionamiento para 4 procesadores.

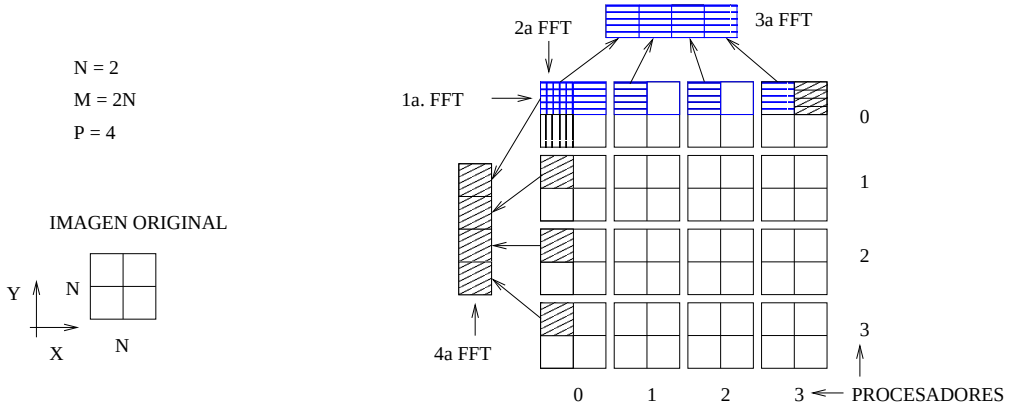

Figura 6.5: Particionamiento de datos y procesamiento de la matriz de imágenes.

La aplicación de la FFT fue ejecutada en 4, 8 y 16 procesadores bajo PVFS (sin operaciones colectivas) y bajo FSDDS. En todas las ejecuciones, el tamaño del *stripe* fue de M/p filas de imágenes y la matriz de imágenes fue de tamaño $((128)^4) \times 32 = 8$ GBytes y pudo ser almacenada totalmente en memoria en todas las configuraciones de procesadores.

La Tabla 6.5 muestra el número de solicitudes de E/S por procesador. En cada configuración de procesadores, el número de solicitudes de E/S bajo FSDDS fue más pequeño que bajo PVFS, porque bajo FSDDS algunas lecturas fueron satisfechas desde las copias en la memoria de otros nodos y porque las escrituras ocurrieron en las copias de la memoria las cuales fueron emitidas al disco sólo cuando la memoria no fue suficiente o hasta que el archivo fue cerrado.

MPI-PVFS		
Procesadores	Lecturas	Escrituras
4	49152	49152
8	24576	24576
16	12288	12288

FSDDS		
Procesadores	Lecturas	Escrituras
4	48729	48732
8	23638	23638
16	9934	9934

Tabla 6.5: Lecturas y escrituras de la FFT con MPI-PVFS y FSDDS

La Figura 6.6 muestra el tiempo de ejecución bajo PVFS (sin operaciones colectivas) y FSDDS, en 4, 8 y 16 procesadores. En 4 procesadores, aún cuando el número de lecturas y escrituras bajo FSDDS fue más pequeño que el número de lecturas y escrituras bajo PVFS, el tiempo de ejecución bajo FSDDS fue mucho mayor porque la memoria disponible fue relativamente pequeña y así la política de reemplazo de FSDDS fue ejercida frecuentemente. En 8 y 16 procesadores, la mayor disponibilidad de memoria significó menor uso de la política de reemplazo, lo que mejoró el desempeño.

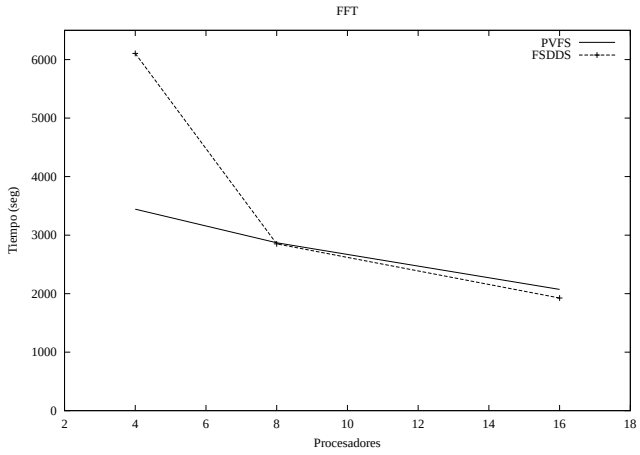

Figura 6.6: Tiempo de ejecución de la FFT con MPI-PVFS y FSDDS

6.3.4. Prueba de la coherencia

En el primer algoritmo de la MM, realizamos una prueba de la coherencia de los bloques de datos al leer y escribir los renglones de la matriz C por cada uno de los procesadores del *cluster*. En este algoritmo cada renglón de C es visto como una variable de competencia por todos los nodos debido a que todos los nodos tratan de modificarlos. Para calcular cada renglón de C se realizan sumas parciales de cada elemento del mismo renglón utilizando las funciones `DDS_Write` y `DDS_UnWrite` en cada procesador para tener una copia exclusiva del bloque; no importa quién accede a los datos primero ni quién los tenga en cualquier momento. La escritura de un mismo bloque es transparente al usuario. En MPI-PVFS, el usuario es el que controla al nodo fuente y destino de los bloques de datos en todo momento y sólo un procesador recibe todos los datos parciales

para obtener el cómputo de cada renglón. La matriz resultante C fue la misma en ambas versiones, bajo MPI-PVFS y FSDDS.

6.4. Discusión

Nuestros resultados confirman que DDS tiende a disminuir el número de operaciones de E/S y consecuentemente el tiempo de ejecución. Esto es debido a que bajo DDS, algunas de las solicitudes de datos pueden ser satisfechas desde la memoria de otros nodos del *cluster*.

Otro aspecto favorable de DDS es la disminución de sincronización entre varios procesos. Bajo MPI, cuando varios procesos participan en sincronización el tiempo de la misma tiende a crecer con el número de procesos. En DDS no es necesario que los procesos se comuniquen en forma colectiva para acceder datos en memoria o incluso en disco.

Aunque estos dos aspectos dependen del patrón de acceso de la aplicación, es posible asumir que el tiempo de ejecución al emplear DDS será al menos similar al tiempo bajo MPI o mejor.

6.5. Resumen

Para evaluar el desempeño de aplicaciones paralelas *out-of-core* bajo FS/DDS ha sido desarrollado un modelo analítico. El modelo permite determinar para qué número de procesadores una aplicación paralela se acelera utilizando FS/DDS. El modelo considera varios factores entre los cuales tenemos: el tamaño del problema, el número de procesadores, el tiempo de comunicación, el tiempo debido a la sincronización de los nodos y el tiempo de acceso a disco.

Con el modelo también puede hacerse una estimación del número de solicitudes a disco considerando el tamaño del problema, el tamaño del espacio de difusión o caché del *cluster* y el número de procesadores. En el modelo también se consideran los patrones de acceso de la aplicación para obtener una mejor aproximación de los resultados del modelo con los resultados experimentales.

Para evaluar experimentalmente el desempeño de DDS y FSDDS se utilizaron tres

aplicaciones paralelas *out-of-core* que utilizan matrices de datos. Las aplicaciones se ejecutaron bajo DDS/FSDDS y MPI-IO/PVFS en un *cluster* con 16 nodos usando diferentes números de procesadores en cada ejecución. Todos los nodos del *cluster* funcionaron como nodos de cómputo y como nodos de E/S. Cada nodo conteniendo un disco con un tamaño diferente y con diferentes velocidades de acceso. Los archivos fueron particionados en bloques y distribuidos en los nodos de E/S en forma cíclica. En cada aplicación se utilizaron diferentes patrones de particionamiento, en donde las matrices fueron distribuidas en forma de renglones y de columnas y el tamaño de un bloque fue igual al tamaño de un renglón de las matrices. La distribución de las matrices se cambió con el fin de utilizar diferentes patrones de acceso de la aplicación.

Los resultados obtenidos de la ejecución de las aplicaciones nos muestran que al utilizar DDS/FSDDS se obtuvo mejor desempeño (3 - 10 %) que al usar MPI-IO/PVFS. Esto es porque el número de solicitudes de E/S obtenido en DDS/FSDDS es menor que el número de solicitudes de E/S obtenido en MPI-IO/PVFS. En DDS/FSDDS algunas solicitudes de E/S son satisfechas desde la memoria de los nodos. Sin embargo, bajo MPI-IO los programadores leen los datos desde y escriben hacia el espacio del disco debido a que saben que los datos no caben en su totalidad en memoria y utilizan un espacio temporal en memoria (*buffer*) para almacenar algunos datos. De esta manera, el número de solicitudes de E/S es implícitamente definido por el programador.

.

Capítulo 7

Conclusiones y trabajo futuro

Esta tesis ha presentado el diseño y la implementación de un sistema de memoria compartida distribuida implementada toda en software, a la cual llamamos DDS (del inglés: Data Diffusion Space). DDS ofrece un nuevo modelo de programación paralela en el que se tiene una misma interfaz para programar aplicaciones *in-core* y *out-of-core*, con la cual la programación es la misma para ambos tipos de aplicaciones, cuyo código es portable en arquitecturas de 32 y 64 bits y potencialmente se ejecuten con un buen desempeño gracias a la cantidad de memoria caché que se utiliza de manera transparente. Además, la misma interfaz es utilizada para aplicaciones que leen/escriben a archivos gracias al mapeo de archivos de FS/DSS. DDS fue diseñado para cómputo paralelo en *clusters*, pero por ser su diseño e implementación todo en software es portable a otras plataformas de memoria distribuida. DDS es un espacio de direcciones adicional al espacio de direcciones virtuales de cada proceso que ejecuta una aplicación paralela bajo el modelo SPMD. El tamaño de este espacio puede ser hasta de 2^{64} bytes, ya sea para arquitecturas de 32 o 64 bits. Los datos colocados en DDS son difundidos, o migrados y replicados, en forma automática en la memoria de los procesadores que los utilizan bajo un protocolo de coherencia "múltiples lectores un solo escritor". Además, si los datos no caben en memoria son almacenados en el espacio del disco cuando no son utilizados por los procesos.

DDS ofrece un ambiente de programación de aplicaciones paralelas que presenta una misma interfaz para programar aplicaciones *in-core* y *out-of-core*. Además, DDS puede ejecutarse en arquitecturas heterogéneas con procesadores de 32 y 64 bits y con un buen

desempeño.

La interfaz de DDS proporciona transparencia en la programación de aplicaciones *in-core* y *out-of-core* porque el programador no se preocupa por la localización de los datos, ni por la capacidad de almacenamiento en memoria del nodo y los utiliza como si estuvieran almacenados en la memoria local del nodo. Además, el programador utiliza un mismo código para ambos tipos de aplicaciones, lo cual lo hace portable entre estas aplicaciones y facilita su programación.

La interfaz de DDS es una biblioteca conteniendo llamadas a funciones de lectura y escritura a un bloque de datos que mapean el bloque dentro del espacio de los procesos y permiten ganar el acceso del bloque antes de que sea accedido. En una lectura los nodos obtienen una copia del bloque de datos y en una escritura sólo un nodo obtiene una copia exclusiva del bloque con lo que es posible garantizar su coherencia.

Para desarrollar aplicaciones paralelas *out-of-core*, se diseñó un sistema de archivos distribuido paralelo sobre DDS llamado FSDDS (del inglés: File System atop the Data Diffusion Space). FSDDS particiona los archivos en bloques y estos bloques son distribuidos en los nodos de E/S. Un *metadato* es utilizado para describir el particionamiento del archivo. Un archivo es creado con funciones similares a las de UNIX. Cuando el archivo es creado no es necesario especificar un patrón de distribución de los datos, FSDDS los distribuye en forma dinámica entre los nodos de E/S del *cluster*.

FSDDS provee una interfaz de funciones para el acceso a datos en archivos. Esta interfaz es similar a la que se utiliza en DDS para realizar las operaciones de lectura y escritura de datos. Por medio de la interfaz, cuando un archivo es abierto bajo FSDDS, éste es mapeado al espacio de direcciones de DDS y sus datos son llevados a la memoria cuando son accedidos por la aplicación.

Un archivo mapeado dentro del espacio de DDS es manejado como un arreglo en memoria, lo que facilita al usuario la programación de aplicaciones *out-of-core*. Con esta técnica de programación el usuario no debe preocuparse por el tamaño de la memoria de los nodos del *cluster*, ni por la localización de los bloques de datos del archivo dentro del *cluster*.

El manejo en memoria de los bloques de datos de los archivos tiende a disminuir los accesos a disco debido a que algunas operaciones de E/S son satisfechas desde las copias almacenadas en la memoria de los nodos, por lo que se mejora el desempeño de

aplicaciones *out-of-core*. Además, los bloques tienden a moverse hacia quién los necesita lo que también disminuye las comunicaciones entre los nodos del *cluster*.

Para realizar las comunicaciones entre los nodos que ejecutan una aplicación, DDS y FSDDS utilizan *sockets* TCP-IP y por lo tanto son portables a las implementaciones modernas de UNIX y WINDOWS.

Para evaluar el desempeño de aplicaciones paralelas *out-of-core* bajo FS/DDS, ha sido desarrollado un modelo analítico. El modelo permite determinar para qué número de procesadores una aplicación paralela se acelera utilizando FS/DDS. El modelo considera varios factores entre los cuales tenemos: el tamaño del problema, el número de procesadores, el tiempo de comunicación, el tiempo debido a la sincronización de los nodos y el tiempo de acceso a disco.

Con el modelo también puede hacerse una estimación del número de solicitudes a disco considerando el tamaño del problema, el tamaño del espacio de difusión o caché del *cluster* y el número de procesadores. En el modelo también se consideran los patrones de acceso de la aplicación para obtener una mejor aproximación de los resultados del modelo con los resultados experimentales.

Para evaluar experimentalmente el desempeño de DDS y FSDDS se utilizaron tres aplicaciones paralelas *out-of-core* que utilizan matrices de datos. Las aplicaciones se ejecutaron bajo DDS/FSDDS y MPI-IO/PVFS en un *cluster* con 16 nodos usando diferentes números de procesadores en cada ejecución.

DDS/FSDDS obtuvo mejor desempeño que MPI-IO/PVFS. Esto es porque el número de solicitudes de E/S obtenido en DDS/FSDDS es menor que el número de solicitudes de E/S obtenido en MPI-IO/PVFS. En DDS/FSDDS algunas solicitudes de E/S son satisfechas desde la memoria de los nodos. Bajo MPI-IO los programadores leen los datos desde y escriben hacia el espacio del disco debido a que saben que los datos no caben en su totalidad en memoria y utilizan un espacio temporal en memoria (*buffer*) para almacenar algunos datos. De esta manera, el número de solicitudes de E/S es implícitamente definido por el programador.

La principal aportación es un nuevo ambiente de programación paralela en el que se tiene una misma interfaz para programar aplicaciones *in-core* y *out-of-core*, con la que la programación es la misma para ambos tipos de aplicaciones, cuyo código es portable en arquitecturas de 32 y 64 bits y potencialmente se ejecuten con un buen desempeño.

Las aportaciones particulares de nuestro trabajo son resumidas como sigue:

- Diseño e implementación de una memoria compartida distribuida implementada totalmente en software y por lo tanto portable.
- Diseño e implementación de un sistema de archivos distribuido paralelo sobre DDS, llamado FSDDS, para desarrollar y ejecutar aplicaciones paralelas *out-of-core* que leen/escriben datos en disco.
- Diseño de la interfaz de DDS y FSDDS. Esta interfaz constituye en si misma un modelo de programación simple y fácil de usar para cómputo paralelo en *clusters* que es independiente del hardware del sistema.
- Diseño de un modelo analítico para la evaluación del desempeño de aplicaciones paralelas utilizando DDS y FSDDS, que permite realizar un análisis apriori de la ejecución de estas aplicaciones.
- Una evaluación preliminar del desempeño de DDS y FSDDS.

Limitaciones y trabajo futuro

Las limitaciones de DDS y el trabajo futuro son:

- Los resultados obtenidos estuvieron limitados debido a que sólo contamos con un *cluster* de 16 nodos con una limitada capacidad de almacenamiento en memoria y disco de cada nodo (menor a 2 GB por nodo). Por lo que es conveniente probar a DDS con más procesadores para ver su escalabilidad y su eficiencia.
- Otra limitación presentada en este trabajo de tesis es el número relativamente pequeño de aplicaciones utilizadas. Hay que probar a DDS con otro tipo de aplicaciones con diferentes patrones de acceso.
- Actualmente la interfaz de DDS es un poco complicada porque el programador tiene que generar un archivo con las estructuras de datos que son definidas en el espacio de difusión de DDS y son usadas para compilar la aplicación, por lo que es necesario un pre-compilador para facilitar la generación del código del programa.

- DDS Y FSDDS actualmente no pueden ser ejecutados en un sistema multiprocesador. En la actualidad muchos *clusters* utilizan nodos con arquitecturas multiprocesador, por lo que es necesario adaptar a DDS y FSDDS para ejecutarse en estas arquitecturas.

- Actualmente DDS utiliza a MPI para ejecutar a los procesos de la aplicación en los nodos de cómputo. Para que DDS sea totalmente independiente es necesario crear un ambiente de carga y ejecución automática de los procesos de una aplicación paralela en un *cluster*.

- Actualmente el sistema genera un reporte con los resultados parciales y finales de varios parámetros de la ejecución de las aplicaciones y genera un reporte en forma de texto de las comunicaciones entre los procesos de la aplicación. Para observar y analizar dinámicamente el comportamiento de la ejecución de las aplicaciones o hacer un análisis posterior a la ejecución de la aplicación es necesario incluir un módulo de depuración gráfico con los datos anteriores.

- En la versión actual de DDS, el usuario obtiene manualmente los patrones de acceso de la aplicación para especificar el patrón de distribución de los datos de la aplicación. Con el fin de determinar en forma automática el patrón de distribución de los datos en disco que ofrezca el mejor desempeño de la aplicación es necesario incluir un módulo de detección de los patrones de acceso de las aplicaciones.

.

Bibliografía

[1] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. *TreadMarks: Shared Memory Computing on Networks of Workstations.* IEEE Computer, Vol. 29, No. 2, pp. 18-28, February 1996.

[2] Anant Agarwal. *The MIT Alewife Machine: Architecture and Performance.* Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp 2-13, 1995.

[3] Tom Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, Randy Wang. *xFS: Serverless Network File Systems.* 15th Symposium on Operating Systems Principles, ACM Transactions on Computer Systems , 1995. URL: http://now.cs.berkeley.edu/Xfs/xfs.html.

[4] George S. Almasi y Allan Gottlieb. *Highly Parallel Computing.* The Benjaming/Cummings Publishing Company, Inc., second edition, 1994.

[5] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, R. Wang. *Serverless Network File Systems.* 15th Symposium on Operating Systems Principles, ACM Transactions on Computer Systems, 1995.

[6] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. *A Case for Networks of Workstations: NOW.* IEEE Micro, Feb, 1995.

[7] L. Arge, O. Procopiuc, J. S. Vitter. *Implementing I/O-Efficient Data Structures Using TPIE.* Proc. 10th European Symposium on Algorithms (ESA '02), Rome, Italy, September 2002.

[8] Remzi H. Arpaci. *Empirical Evaluation of the CRAY-T3D: a Compiler Perspective.* Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp 320-331, 1995.

[9] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. *ORCA: A Languaje for Parallel Programming of Distributed Systems.* IEEE Transactions on Software Engineering, March 1992.

[10] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. *The Midway distributed shared memory system.* Procc. IEEE Computer Conf., pp. 528-537, 1993.

[11] A. Bilas, D. Jiang, J.P. Singh. *Shared virtual memory clusters: bridging the cost-performance gap between SMPs and hardware DSM systems.* Journal of Parallel and Distributed Computing, vol. 63, no. 12, page 1257-1276, December 2003.

[12] R. Brown, *Engineering a Beowulf-style computer cluster.* URL: http://www.phy.duke.edu/brahma/beowulf_online_book/beowulf_book.html, 2002.

[13] Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Catalin Rosu, H., Raymond Strong. *Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations.* SPAA, 1995:64-76

[14] Ray Bryant. *The RP3 Parallel Computing Environment.* Proceedings of USENIX 1988 Supercomputer Workshop, pp 69-91, 1988.

[15] J. Buenabad-Chávez, H.L. Muller, P.W.A. Stallard and D.H.D Warren. *Virtual memory on data diffusion architectures.* Parallel Computing, 29 (2003) 1021–1052.

[16] J. Buenabad-Chávez, S. Domínguez-Domínguez. *The Data Diffusion Space for Parallel Computing in Clusters.* 11th International Euro-Par Conference on Parallel Processing, Euro-Par 2005.

[17] J. Buenabad-Chávez, S. Domínguez-Domínguez. *Comparing Two Parallel File Systems: PVFS and FSDDS.* International Conference on Parallel Computing, ParCo 2005.

[18] Rajkumar Buyya. *High Performance Cluster Computing: Architecture and Systems.* Vol. 1, Prentice Hall, 1999.

[19] Rajkumar Buyya. *High Performance Cluster Computing: Programming and Applications.* Vol. 2, Prentice Hall, 1999.

[20] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. *PVFS: A Parallel File System For Linux Clusters.* Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, pp. 317-327, October 2000.

[21] J.B. Carter. *Design of the Munin Distributed Shared Memory System.* J. Parallel and Distributed Computing, Special Issue on Distributed Shared Memory, 1995.

[22] Yi Chan Zhuang, Ce Kuen Shieh, Tyng Yue Liang, Chih Hui Chou. *Maximizing Speedup through Performance Prediction for Distributed Shared Memory Systems.* IEEE, 2001.

[23] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson. *RAID: High Performance and Reliable Secondary Storage.* ACM Computing Surveys, vol. 26(2), pp. 145-185, 1994.

[24] Y. Chen. *Automatic Parallel I/O Performance Optimizations in Panda.* University of Illinois at Urbana-Champaign. Thesis of Ph. D. 1999.

[25] Y. Chen, M. Winslett, S. Kuo, Y. Cho, M. Subramaniam, and K.E. Seamons. *Performance Modeling for the Panda Array I/O Library.* In Supercomputing '96, ACM Press and IEEE Computer Society Press, November 1996.

[26] Alok Choudhary, Rajesh Bordawekar, Sachin More, K. Sivaram, and Rajeev Thakur. *PASSION runtime library for the Intel Paragon.* In Proceedings of the Intel Supercomputer User's Group Conference, June 1995.

[27] Douglas E. Comer, David L. Stevens. *Internetworking with TCP/IP Volume 3: Client-Server Programming and Applications.* Prentice Hall, 2001.

[28] *Concurrent Programming with Treadmarks.* ParallelTool L.L.C, 1996.

[29] P.F: Corbett and D.G. Feitelson. *Design and Implementation of the Vesta Parallel File System.* Proceedings of the Scalable High-Performance Computing Conference, pag. 63-70, 1994.

[30] P.F: Corbett and D.G. Feitelson, *The Vesta parallel file system.* In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, High Performance Mass Storage and Parallel I/O: Technologies and Applications, chapter 20, pp. 285-308. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[31] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, Ch. Yoshikawa, F. Wong. *Parallel Computing on the Berkeley NOW.* 9th Joint Symposium on Parallel Processing, 1997.

[32] P.J. Denning. *Virtual Memory.* Computing Surveys, vol. 2, No. 3, September 1970.

[33] Bernd Dreier, Markus Zahn, and Theo Ungerer. *Rthreads - a Uniform Interface for Parallel and Distributed Programming.* Second International Conference on Massively Parallel Computing Systems MPCS'96, 530-534, 1996.

[34] Bernd Dreier, Markus Zahn, and Theo Ungerer. *The Rthreads Distributed Shared Memory System.* Third International Conference on Massively Parallel Computing Systems MPCS'98, Colorado Springs, April 1998. URL: http://www.rz.uni-augsburg.de/~zahn/Rthreads/

[35] C. Fantozzi, A. Pietracaprina, G. Pucci. *Implementing Shared Memory on Clustered Machines.* IEEE, 2001.

[36] Dror G. Feitelson, Peter F. Corbett, Yarsun Hsu, and Jean-Pierre Prost. *Parallel I/O Systems and Interfaces for Parallel Computers.* URL: http://www.cs.huji.ac.it/~feit/pub.html, 1995

[37] Dror G. Feitelson, Peter F. Corbett, Sandra Johnson Baylor, and Yarsun Hsu. *Parallel I/O subsystems in massively parallel supercomputers.* En Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, High Performance Mass Storage and Parallel I/O: Technologies and Applications, chapter 25, pp. 389-407. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[38] I. Foster. *Designing and Building Parallel Programs* Addison Wesley, 1996. URL: http://www.mcs.anl.gov/dbpp

[39] N. Galbreath, W. Gropp, D. Levine. *Applications-driven parallel I/O.* In Proceedings Supercomputing '93. pp. 462-471, Nov. 1993.

[40] *Grand Challenging Applications.* URL: http://www.mcs.anl.gov/Projects/grand-challenges/.

[41] P.B. Hansen. *Model Programs for Computational Science: A Programming Methodology for Multicomputers.* Concurrency: Practice and Experience, vol. 5(5), pp. 407-423, 1993.

[42] Philip Heidelberger, Sthepen S. Lavenberg. *Computer Performance Evaluation Methodology.* IEEE Transactions on Computer, Vol. c-33, No. 12, pp. 1195-1220, December 1984.

[43] John L. Hennessy, David A. Patterson. *Computer Architecture: A Quantitative Approach.* The Morgan Kaufmann Series in Computer Architecture and Design.

[44] James V. Huber, Cristopher L. Elford, Daniel A. Reed, Andrew A. Chien, David S. Blumenthal. *PPFS: A High Performance Portable File System.* En Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona 1995. URL: http://www-pablo.cs.uiuc.edu/Projects/PPFS/ppfs.html

[45] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993

[46] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)-part1: System application program interface (API) [C Language]. 1996 edition.

[47] Florin Isaila, Walter F. Tichy. *Clusterfile: A Flexible Physical Layout Parallel File System.* In 3rd IEEE International Conference on Cluster Computing (CLUSTER'01), October 08 - 11, 2001. Newport Beach, CA

[48] Truman Joe. *COMA-F: A Non-hierarchical Cache Only Memory Architecture.* Stanford University Department of Electrical Engineering. Thesis of Ph. D. 1995.

[49] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. *CRL: High-Performance All-Software Distributed Shared Memory.* In Proceedings of the Fifteen Symposium on Operating Systems Principles, December 1995.

[50] R.H. Katz. *High-performance network and channel based storage.* Proceedings of IEEE, Vol. 80, No. 8, pp. 1238-1261, August 1992.

[51] R.H. Katz, G.A. Gibson, and D.A. Patterson. *Disk system architectures for high performance computing.* Proceedings of IEEE, Vol. 77, No. 12, pp. 1842-1858, December 1989.

[52] C. Koelbel, D. Loveman, R. Shriber, G. Steele, Jr., and M.Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[53] David Kotz. *Applications of Parallel I/O.* Technical Report PCS-TR96-297, Department of Computer Science, Dartmouth College, October 14, 1996.

[54] David Kotz. *Disk-Directed I/O for MIMD Multiprocessors.* ACM Transactions on Computer Systems, vol. 15(1), February 1997.

[55] Ce Kuen shieh, Su Cheong Mac and Bor Jyh Shieh. *Reducing File-related Network Traffic in TreadMarks via Parallel File Input/Output.* Journal of Information Science and Engineering, 15, pp. 569-583, 1999.

[56] *LAM/MPI Parallel Computing.* URL: http://www.lam-mpi.org/

[57] R. Latham, N. Miller, R. Ross, and P. Carns. *PVFS2: A Next-Generation Parallel File System for Linux Clusters: An introduction to the second Parallel Virtual File System.* Linux World Magazine URL: at http://www.LinuxWorld.com, January 2004.

[58] James Laudon and Daniel Lenosky. *The SGI Origin: A ccNUMA Highly Scalable Server.* Proceedings of the 24th Annual International Symposium on Computer Architecture, pp 241-251, 1997.

[59] Daniel Lenoski. *The Stanford Dash Multiprocessor.* IEEE Computer, pp. 63-79, March, 1992.

[60] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. *The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor.* In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 148-159, May 1990.

[61] Kay Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors.* Yale University Department of Computer Science. Thesis of Ph. D. 1986.

[62] K. Li. *IVY: A Shared Virtual Memory Systems for Parallel Computing.* In Proceedings of the 1988 International Conference on Parallel Processing (ICP'88). Vol. II, pp. 94-101, August 1988.

[63] K. Li, and P. Hudak. *Memory Coherence in Shared Virtual memory Systems.* ACM Trans. Computer systems, vol. 7, no. 4, nov. 1989

[64] W. B. Ligon III and R. B. Ross, *An Overview of the Parallel Virtual File System.* Proceedings of the 1999 Extreme Linux Workshop, June, 1999.

[65] Tom Lovett and Russell Clapp. *STING: A CC-NUMA Computer System for the Commercial Marketplace.* Proceedings of the 23rd Annual International Symposium on Computer, pp 308-317, 1996.

[66] John M. May. *Parallel I/O for High Performance Computing.* Morgan Kaufmann Pub., 2001

[67] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* Version 1.1, June 1995. URL:http://www.mpi-forum.org/docs/docs.html

[68] *MPICH - A Portable Implementation of MPI.* URL: http://www-unix.mcs.anl.gov/mpi/mpich/

[69] *MPI-IO: A Parallel File I/O Interface for MPI.* The MPI-IO Committee, April 1996.

[70] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface.* July 1997. URL: http://www.mpi-forum.org/docs/docs.html

[71] Steven A. Moyer and V.S. Sunderam. *PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments.* In Proceedings of the Scalable High-Performance Computing Conference, 1994. URL: http://www.math.cs.emory.edu/pious/

[72] Nils Nieuwejaar and David Kotz. *Low-level Interfaces for High-level Parallel I/O.* Workshop for I/O in Parallel and Distributed Systems, IPPS 1995, pag. 47-62, 1995.

[73] Nils Nieuwejaar and David Kotz. *The Galley Parallel File System.* Parallel Computing, 2384, June 1997. URL: http://www.cs.dartmouth.edu/~dfk/nils/galley.html

[74] Bill Nitzberg and Virginia Lo. *Distributed Shared Memory: A survey of Issues and Algorithms.* IEEE Computer, vol. 24 (8), pp. 52-60, 1991.

[75] S. Nog, D. Kotz. *A performance comparison of TCP/IP and MPI on FDDI, Fast Ethernet, and Ethernet.* Dartmouth Technical Report PCS-TR95-273, November 1995. URL: ftp://ftp.cs.dartmouth.edu/TR/TR95-273.ps.Z

[76] OpenMP. URL: http://www.openmp.org/

[77] Jaswinder Pal Singh *et. al. Empirical Comparison of the Kendall Square Research KSR-1 and the Stanford Dash Multiprocessors.* In Proceedings of the 1993 International Conference on SUPERCOMPUTING, pp 214-225, 1993.

[78] David A. Patterson, K.L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[79] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. *Informed Prefetching and Caching.* In Fifteenth ACM Symposium on Operating Systems Principles, ACM Press, pp. 79-95, CO, December 1995.

[80] Pthreads. URL: http://www.humanfactor.com/pthreads/

[81] PVFS. *The Parallel Virtual File System.* URL: http://parlweb.parl.clemson.edu/pvfs/

[82] PVM. *Parallel Virtual Machine.* URL: http://www.epm.ornl.gov/pvm/pvm_home.html

[83] Sanjay Raina. *Emulation of a Virtual Shared Memory Architecture.* Ph. D. of the University of Bristol, September 1993.

[84] Rob B. Ross. *Providing Parallel I/O on Linux Clusters.* Argonne National Laboratory, Mathematics and Computer Science Division. URL: http://www.mcs.anl.gov

[85] Rob B. Ross. *Providing Parallel I/O on Linux Clusters.* Second Annual Linux Storage Management Workshop, Miami, FL, October 2000.

[86] K. Salem and H. Garcia-Molina. *Disk Striping.* IEEE 1986 Conference on Data Engineering, pag. 336-346, 1986.

[87] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. *Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory.* Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems. October, 1996.

[88] Erich Schikuta, Thomas Fuerle, and Helmut Wanek. *ViPIOS: The Vienna Parallel Input/Output Systems.* In Euro-Par'98, Southampton, England, Springer-Verlag, September 1998.

[89] Erich Schikuta and Heinz Stokinger. *Parallel I/O for clusters: Methodologies and systems.* In Rajkumar Buyya, editor, High Peformance Cluster Computing, pp. 439-462. Prentice Hall PTR, 1999.

[90] Frank Schmuck and Roger Haskin. *GPFS: A Shared-Disk File System for Large Computing Clusters.* In Proceedings of the Conference on File and Storage Technologies (FAST'02), pp. 28-30, January 2002.

[91] H. Shan, J.P. Singh, L. Oliver, R. Biswas. *Message passing vs. shared address space on a cluster of SMPs.* Parallel and Distributed Processing Symposium, April 2001.

[92] Xiaohui Shen, Alok Choudhary. *A high-performance distributed parallel file system for data-intensive computations.* Journal of Parallel and Distributed Computing, vol. 64, pp. 1157-1167, April 2004.

[93] C.K. Shieh, S.Ch. Mac, J.Ch. Ueng. *Improving the performance of ditributed shared memory systems via parallel file input/output.* Journal of Systems and Software, vol. 44, No. 1, pp. 3-15, December 1998.

[94] Kent E. Seamons. *Panda: Fast Access to Persistent Arrays Using High Level Interfaces and Server Directed Input/Output.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, May 1996.

[95] H. Stockinger. *Classification of Parallel Input/Output Products.* In Parallel and Distributed Porcessing Techniques and Applications Conference. July, 1998.

[96] Vaidy S. Sunderam, G. A. Geist, Jack Dongarra, R. Manchek. *The PVM Concurrent Computing System: Evolution, Experiences and Trends.* Parallel Computing 20(4):531- 545(1994)

[97] Mihai Surdeanu, Dan Moldovan. *Design and Performance Analysis of a Distributed Java Virtual Machine.* IEEE Transactions on Parallel and Distributed Systems, vol. 13, No. 6, June 2002.

[98] Tanenbaum, Andrew S. *Operating Systems: Design and Implementation (Second Edition).* New Jersey: Prentice-Hall 1997

[99] R. Thakur and A. Choudhary. *An Extended Two-Phase Method for Accesing Sections of Out-of-Core Arrays.* Scientific Programming, vol. 5(4), 1996.

[100] R. Thakur, E. Lusk, and W. Gropp. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

[101] TOP500. *TOP500 Supercomputing Sites.* URL: http://www.top500.org/

[102] TreadMarks. URL: http://www.cs.rice.edu/~willy/TreadMarks/overview.html

[103] D. E. Vengroff. *A Transparent Parallel I/O Environment.* Proceedings of the Third DAGS Symposium on Parallel Computation, Hanover, NH, July 1994, 117-134.

[104] Murali Vilayannur, Robert B. Ross, Philip H. Carns, Rajeev Thakur, Anand Sivasubramaniam, Mahmut Kandemir. *On the Performance of the POSIX I/O Interface to PVFS.* Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004.

[105] R. Y. Wang, T. E. Anderson. *The Design of Large-Scale, Do-It-Yourself RAIDs.* White Paper, 1993. URL: http://now.cs.berkeley.edu/Papers2/

[106] R. Y. Wang, T. E. Anderson. *xFS: A Wide Area Mass Storage File System.* White Paper, 1993. URL: http://now.cs.berkeley.edu/Papers2/

[107] David H. D. Warren and Seif Haridi. *DATA DIFFUSION MACHINE: A Scalable Shared Virtual Memory Multiprocessors.* Proceedings of the International Conference on Fifth Generation Computer Systems, pp 943-952, 1988.

[108] Werstein, P., Pethick, M., and Huang, Z., *A performance comparison of DSM, PVM, and MPI.* Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 03). Chengdu, China (2003) pp. 476-482.

[109] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, San Francisco, CA, 1999.

[110] Tyng Yeu Liang, Yen Tao Liu, Ce Kuen Shieh. *Adding Memory Resource Consideration into Workload Distribution for Software DSM Systems.* Proceedings of the IEEE International Conference on Cluster Computing, 2003.