UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES
ARAGÓN

ANÁLISIS DE SISTEMAS DE ARCHIVOS EN PARALELO PARA APLICACIONES CIENTÍFICAS

T E S I S

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

PRESENTA:

FERNANDO ROBLES MORALES

ASESOR:

ENRIQUE CRUZ MARTÍNEZ

MÉXICO, 2007

II

AGRADECIMIENTOS

Este trabajo de tesis no es de una sola persona, es de muchas personas que directamente o indirectamente han participado en este.

Al que nos recogió cuando fuimos rechazados, al que nos ha instruido no solo en llevar una vida material digna, sino también una vida espiritual limpia, gracias a ti Maestro Samuel Joaquín Flores, este trabajo es un fruto más por el esfuerzo que has hecho por nosotros.

En especial le agradezco a Dios por tener a Ruth, mi mama trabajadora, que por los valores y la formación que me ha dado, y sobre todo su valentía y amor no estaría aquí.

A mi tía Eddy, mi otra mama trabajadora, Dios te siga dando salud y bienestar, gracias por tu cariño, tu confianza, tu amor, no se que haría sin tus regaños.

A mi querida hermana Betsabe, que te había pedido desde mi niñez, es una alegría que esa petición se haya vuelto una realidad y quiero que sepas que si tienes fe y amor puedes lograr lo que quieras, no eres más ni eres menos que nadie, haz una brecha en el camino que te ha tocado vivir y llega a donde debes llegar.

A mis queridos hermanos de la comunidad a la que pertenezco, gracias por su apoyo y sus consejos que han influido en mi forma de pensar y actuar.

A la niña Erika Elizabeth, sigo pensando que eres para mi y no se definirlo, lo único que se me ocurre es quererte mucho tal como la cosa que eres.

A la niña Alicia, eres la chica más linda que conozco, tu nobleza y sobre todo tu valentía es lo que admiro de tí, nunca dejes de luchar, te quiero mucho.

A la niña Ana Lilia, eres mi maestra, me diste una perspectiva que no tenía de las personas que me rodean, tu temperamento y tu locura es lo que aprecio más de ti, y a pesar de que eres una ingrata te quiero mucho.

A la niña Helem, eres mi consentida, me impresiona tu cariño, tu calor humano y el don que tienes para ayudar a los demás, nunca dejes tu sueño y lucha por ello.

Al ingeniero Toño, gracias por compartir un cacho de tu vida, tus regaños y consejos no han sido en vano conio, tu fortaleza y compañerismo es lo que aprecio más de ti.

Al maistro Cesar Cardenas, te agradezco por compartir tus dichas y desdichas, la determinación que tienes es impresionante, ahora eres papá primerizo y espero que vengan en camino muchos más frutos para enarbolar a tu familia, te aprecio mucho.

No lo he olvidado, Cesar y Olga, gracias por su cariño y amistad en los momentos difíciles y en las alegrías que hemos pasado, sus comentarios en la elaboración de este trabajo de tesis fueron de gran ayuda, que Dios les siga dando más.

También a Elias, tu ayuda fue primordial en la elaboración de este trabajo, me induciste al mundo maravilloso de LaTeX, gracias por tu amistad.

A Lizbeth, Nora Isabel, gracias por su apoyo en las buenas y en las malas, las quiero mucho.

A mi asesor Enrique Cruz Martínez, gracias por tu confianza, apoyo incondicional y tu amistad, creiste en mi, y he aprendido muchas cosas, no solo técnicas si no también aportas un cacho en mi formación profesional y de vida.

A Eduardo Murrieta, gracias por tu apoyo, confianza y amistad, de ti he aprendido mucho, sin ti no se hubiera logrado este trabajo.

A Jose Luis Gordillo Ruíz, gracias por la confianza que has puesto en mi, el apoyo en este proyecto y tu amistad, no se lograria este trabajo sin tu ayuda.

A los integrantes del Departamento de Supercómputo de la Dirección General en Servicios en Cómputo Académico en la UNAM, los que estan y los que ya se han ido, Yoli, Reyna, Silvia, Marisol, Hector Yuen, David, Eduardo Cabrera, Adrian, Leobardo, Arión, Humberto y Jenny se que estás en un lugar mejor, al que esperamos llegar algún día, gracias por su apoyo.

A los integrantes del Laboratorio de Cómputo para la Enseñanza de las Ciencias en la UACM, donde realize las pruebas finales de mi trabajo de tesis, Verónica, Agustín, Alejandro y Daniel gracias por su confianza, apoyo y su amistad.

A mis revisores de tesis, mis profesores en alguna ocasión durante la carrera, gracias por la formación que me han dado, su apoyo y la amistad que me han dado.

A la UNAM, gracias a la institución que me ha formado desde el CCH hasta ahora, he tenido dichas y desdichas en tu cobijo, pero me has enseñado esa fuerza que hace hablar al espíritu y a salir adelante.

A los que no he mencionado, no porque me haya olvidado de ustedes, son tantos que mejor haría un capítulo completo para cada uno, gracias por su amistad, apoyo, cariño, compañerismo, sin ustedes este trabajo no fuera posible.

Índice general

Índice de figuras

Índice de cuadros

INTRODUCCIÓN

Desde tiempos antiguos, las civilizaciones han tenido la necesidad de interpretar comportamientos y fenómenos naturales, con base en la cuantificación y clasificación. Para llevar a cabo una clasificación basada en la cuantificación, se requieren herramientas de medición en varias características de un proceso natural.

Con un conjunto de mediciones sobre un comportamiento o fenómeno natural, se crea la necesidad de predecir el comportamiento de un proceso natural con base en modelos. Para desarrollar los modelos, conjuntaron varios conocimientos adquiridos en las mediciones y emplearon un lenguaje común para describirlos y ser entendibles para los demás.

En consecuencia, el desarrollo de los modelos amplió los conocimientos adquiridos y estos se dividieron en ramas del conocimiento especializado en un comportamiento de la naturaleza.

A las ramas del conocimiento especializado se les denominó ciencias y continuaron en su desarrollo para crear nuevas ramas de especialización del conocimiento. No solamente fue el desarrollo de las ciencias, también fue la aplicación de procesos naturales en las actividades humanas para satisfacer necesidades. Con los modelos obtenidos, se realiza el diseño de herramientas y mecanismos para construir máquinas que realizan un proceso natural.

Las primeras máquinas fueron imprecisas en su funcionamiento. Por ello, se procedió a especializar las herramientas y las máquinas para obtener el mejor funcionamiento posible. Para llevar a cabo un buen funcionamiento, se requiere de modelos más precisos con base en cálculos con un grado mayor de precisión. De esta manera, se crea una rama de las ciencias aplicadas en la construcción de herramientas de cálculo.

Con el avance de las ciencias aplicadas o tecnología, se crearon diversas máquinas de cálculo, desde las mecánicas, hasta lo que ahora conocemos como computadora. El desarrollo de las computadoras se debe en parte a

la necesidad de realizar cálculos más complejos para representar un modelo acerca de un proceso natural. Hoy en día, no solamente se requiere de un modelo para un proceso natural, sino también de su simulación, por ello los recursos de cálculo con el paso del tiempo llegan a ser insuficientes, por el aumento de la complejidad de los problemas que requieren un mayor número de resultados por obtener.

De esta manera aparece el cómputo de alto rendimiento, que es una de las áreas que aporta un gran desarrollo en las ciencias e ingeniería. En el cómputo de alto rendimiento se emplean máquinas que puedan satisfacer las necesidades de cálculo para las aplicaciones científicas, que superan a lo que ofrece una computadora de escritorio.

En este ambiente las aplicaciones requieren cantidades de datos cada vez mayores, que implica aumentos de capacidad en los sistemas de almacenamiento para satisfacer la demanda de grandes volúmenes de datos.

Este trabajo de tesis se enfoca en la utilización de los sistemas de almacenamiento, porque es uno de los principales obstáculos que limitan el rendimiento de una aplicación científica. Esto se debe a que los datos almacenados en el sistema, tardan un lapso de tiempo en ser leídos o escritos.

Por ello, se ha creado un área de investigación sobre los sistemas de archivos utilizados en las computadoras de alto rendimiento, cuyo objetivo es desarrollar y evaluar sistemas de archivos en varios de sistemas de almacenamiento y determinar cuál es mejor en términos de lectura y escritura de datos para determinar el rendimiento del sistema.

Las aplicaciones científicas que requieren una gran demanda de cálculos y datos son diseñadas e implementadas para su procesamiento en paralelo, y la mayoría utilizan MPI (que por sus siglas en inglés es Message Passing Interface).

En el caso particular del trabajo de tesis, se describe la implementación y desarrollo de una biblioteca paralela reducida, con base en el estándar de programación paralela por envío de mensajes MPI, en el subconjunto de funciones dedicadas al proceso de lectura/escritura en archivos.

La ventaja de MPI es su portabilidad que lo ha llevado a ser el estándar en la mayoría de las computadoras de alto rendimiento. Por ello se escogió para implementar las funciones necesarias para operar con el sistema de archivos distribuido AFS (que por sus siglas en inglés es Andrew File System), y ser comparado con el sistema de archivos paralelo PVFS (que por sus siglas en inglés es Parallel Virtual File System).

La meta es crear una biblioteca paralela para el sistema de archivos distri-

buido AFS que tenga un buen rendimiento casi igual al obtenido en PVFS, en donde PVFS es el sistema de archivos paralelo donde las aplicaciones de MPI con actividad intensiva de lectura/escritura obtienen un buen rendimiento.

En el desarrollo del tema, en el capítulo 1 se describirá el panorama del cómputo de alto rendimiento; en el capítulo 2 tratará lo referente a los sistemas de archivos; en el capítulo 3 se explicará las bibliotecas paralelas utlizadas en lectura/escritura de archivos, el capítulo 4 se describe el desarrollo e implementación de la biblioteca paralela reducida; el capítulo 5 se referirá el benchmark utilizado para medir la tasa de transferencia de los sistemas de archivos; el capítulo 6 se tratará la aplicación utilizada para medir el rendimiento de los sistemas de archivos y el capítulo 7 enfocará las conclusiones del trabajo realizado.

Capítulo 1

CÓMPUTO DE ALTO RENDIMIENTO

El cómputo de Alto Rendimiento tiene un gran crecimiento debido a la demanda cada vez mayor de recursos computacionales para la soluciones que ofrecen las aplicaciones científicas e industriales. Este es el punto de partida en este trabajo de tesis; en este capítulo se describe el panorama real que afronta el cómputo del Alto Rendimiento, así como las arquitecturas de computadoras utilizadas para este fin y el desarrollo del área en la actualidad.

1.1. ¿Qué es el cómputo de Alto Rendimiento?

El Cómputo de Alto Rendimiento es el conjunto de conocimientos computacionales enfocados a resolver las demandas de capacidad computacional [véase 14]. Dichos conocimientos ayudan al uso eficiente de las computadoras, optimizando los problemas que requieren de una gran capacidad computacional o alto rendimento e igualmente sirviendo de herramienta para el alcance de metas específicas.

Las siguientes preguntas surgen de la búsqueda de estas metas:

- ¿Cómo se debe formular un problema de tal manera que facilite su tratamiento computacional?
- ¿Cómo se puede representar el dominio de un problema mediante estructuras formales para un procesamiento computacional?

- ¿Cuáles son las arquitecturas de computadoras que proporcionan un buen rendimiento en la búsqueda de la solución del problema específico?
- ¿Cuáles algoritmos proporcionan la mejor aproximación y menor complejidad para la solución del problema específico?
- ¿Cuáles herramientas de software existentes ofrecen las mejores expectativas en la solución de problemas?

Previo a la invención de las computadoras, la investigación científica y tecnológica ha requerido herramientas de cálculo, por lo que las investigaciones científicas y tecnológicas han considerado el uso de modelado, la simulación, y el control de los sistemas del mundo real. Así los científicos e ingenieros optaron por aproximaciones sucesivas experimentales basándose en los métodos numéricos.

De esta manera, se desarrollaron los métodos numéricos, enfocados hacia la búsqueda de soluciones para problemas específicos. Con la llegada de las computadoras digitales dichos métodos han alcanzado un gran desarrollo en la comprensión de problemas técnicos y de interés científico.

Actualmente, se ha incrementado la complejidad de los problemas debido al avance en cuestiones computacionales. Existe una clasificación conocida como problemas del Gran Reto, que se refieren a problemas con un amplio impacto económico, científico y social. Su solución se pueden alcanzar aplicando precisamente cómputo de alto rendimiento. A grandes rasgos, alguna de sus características principales son:

- Manejo de un gran número de variables a resolver, elevando los requerimientos de memoria y velocidad de procesamiento;
- Procesamiento considerable de grandes conjuntos de datos.
- Manejo en las escalas de unidades en diversas aplicaciones, que requieren escalas demasiado pequeñas o demasiado grandes difíciles de comparar en forma experimental.

Algunos de los problemas del Gran Reto son:

- Estudio de la estructura y dinámica de las moleculas biológicas.
- Pronóstico del clima y predicción de los cambios climáticos globales.

- Investigación sobre la creación y evolución de galaxias.
- Diseño Aereoespacial.
- Farmacología.
- Industria Automotriz.
- Modelación de superconductores.
- Diseño de nuevos materiales.
- Sistemas biomédicos.

Existe una búsqueda continua por mayor capacidad computacional con el fin de satisfacer los requerimientos de los problemas del Gran Reto. En años anteriores, el desarrollo de hardware y software ha permitido que dos o más computadoras en red unan sus capacidades computacionales para resolver problemas. En las siguientes secciones se identifican las tecnologías actuales de las computadoras de alto rendimiento, así como del desarrollo de la capacidad computacional que se ha tenido desde las últimas seis décadas a la fecha.

1.2. Computadoras de Alto Rendimiento

1.2.1. ¿Qué es una computadora de alto rendimiento?

Una Computadora de Alto Rendimiento (CAR), es una máquina con una arquitectura avanzada que tiene las siguientes características:

- Paralelismo en diferentes niveles (hardware y software).
- Dos o más niveles de memoria en función de sus tiempos de acceso, denominado memoria jerárquica.
- Interconexión de alta velocidad entre los procesadores ó máquinas.
- Subsistemas especializados de E/S.
- Uso de software especializado en tales arquitecturas (por ejemplo; sistema operativo, herramientas de análisis, compiladores, etcétera).

Las computadoras de alto rendimiento han adquirido un papel importante en el desarrollo científico y tecnológico del siglo XX, ya que esto ha contribuido al desarrollo de redes de interconexión, procesadores, diseño de memorias, y sistemas de almacenamiento.

El desarrollo de la tecnología es de tal magnitud que un grupo de computadoras de escritorio en una red mediante software pueden comunicarse entre sí para realizar una tarea común. A esto se le conoce como cluster o cúmulo de pc's alcanzando en algunos casos rendimientos comparables a los de una máquina paralela o vectorial. Estas máquinas se clasifican como Computadoras de Alto Rendimiento (CAR).

1.2.2. Clasificación de CAR

En 1972, Flynn proporcionó una clasificación de las CAR en función de su flujo de instrucciones y datos. Algunas computadoras actuales no pueden entrar en la clasificación de Flynn y se añaden otros tipos de tipologias de acuerdo a la manera de organizar la memoria. A continuación se muestran las tipologias más generales que abarcan a las computadoras de alto rendimento [véase 2].

Máquinas SISD

Las computadoras SISD (Single Instruction, Single Data), cuyo esquema se muestra en la Figura 1.1, incluyen a las computadoras convencionales que contienen un procesador ejecutando de manera serial un flujo de instrucciones. Aunque hay sistemas que tienen 2 o más procesadores, ejecutan flujos de instrucciones que no representan partes de un solo proceso. Estos sistemas, conocidos como mainframes son un acoplamiento de varias computadoras SISD que actúan en diferentes espacios de datos.

Máquinas SIMD

Las computadoras SIMD (Single Instruction, Multiple Data), cuyo esquema se muestra en la Figura 1.2 consideran dos tipos: sistemas multiprocesadores y las computadoras vectoriales. En los sistemas multiprocesador, un gran número de procesadores ejecutan la misma instrucción en un conjunto de datos. De esta manera, la instrucción manipula varios elementos de un conjunto de datos en paralelo. En el caso de las computadoras vectoriales,

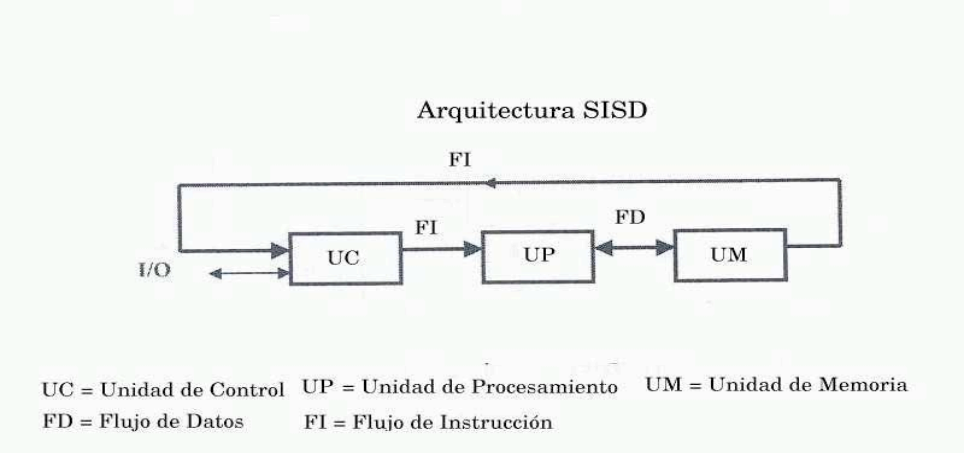

Figura 1.1: Arquitectura SISD

un procesador vectorial aplica una instrucción sobre un arreglo de datos similares, en lugar de tomar un solo elemento del arreglo de datos, tal como lo hacen los procesadores escalares. Cuando los procesadores vectoriales manipulan los datos, el resultado de la instrucción se obtiene en 2 o 3 ciclos de reloj. Así, los procesadores vectoriales ejecutan sus datos en paralelo, siempre y cuando la instrucción sea una operación vectorial.

Máquinas MISD

Las computadoras MISD (Multiple Instruction, Single Data), tiene como ejemplo ilustrativo el superpipeline donde el flujo de datos entra en diferentes elementos procesadores como se muestra en la Figura 1.3.

En cada ciclo de reloj, el primer elemento procesador lee un dato de memoria aplicándole una instrucción. El resto de los procesadores toman el flujo de datos saliente de su vecino y ejecutan una instrucción. Antes del siguiente ciclo de reloj cada procesador prepara el flujo de datos saliente. Este sistema podría usarse para procesamiento en tiempo real, pero dicho hardware no se ha construido con fines comerciales.

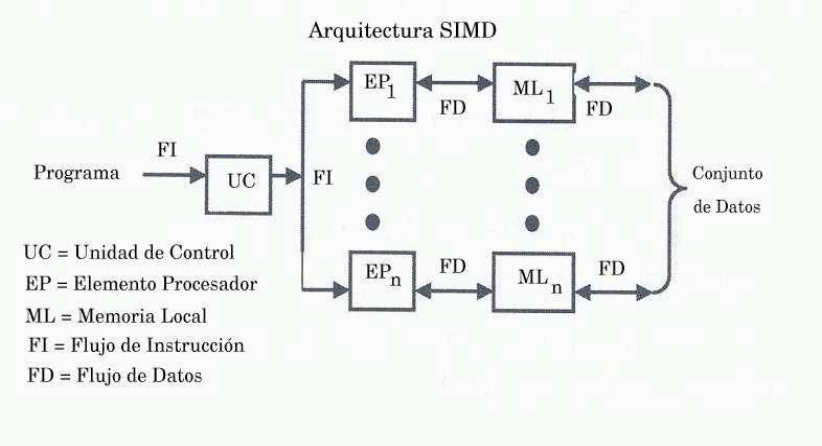

Figura 1.2: Arquitectura SIMD

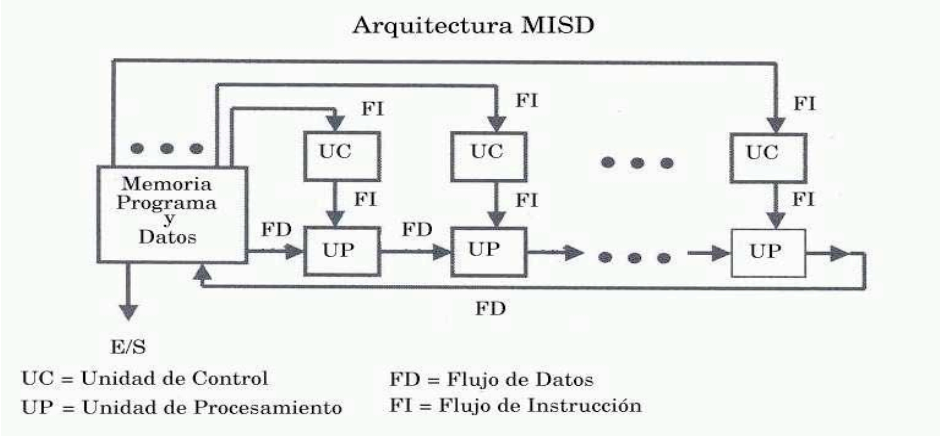

Figura 1.3: Arquitectura MISD

Máquinas MIMD

Las computadoras MIMD (Multiple Instruction, Multiple Data), cuyo esquema se muestra en la Figura 1.4, ejecutan varios flujos de instrucciones sobre diferentes flujos de datos en paralelo. La diferencia con los sistemas multiprocesador SISD (mainframes) es la relación que existe entre las instrucciones y los datos. Cada procesador toma las instrucciones y datos que representan partes de un solo proceso, a causa de esto, las máquinas MIMD tienen varios subprocesos en paralelo, que pueden estar ordenados de acuerdo al tiempo de solución aproximado para ser ejecutados.

Existe una gran variedad de máquinas MIMD que no entran adecuadamente en la clasificación de Flynn [véase 2]. De ahí que se añaden otras tipologías de acuerdo a la forma de organizar la memoria.

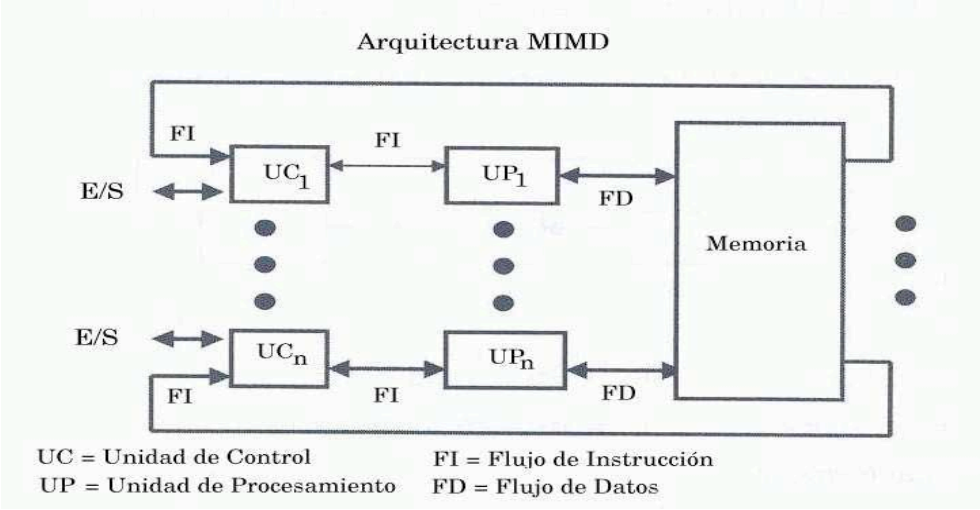

Figura 1.4: Arquitectura MIMD

Sistemas de Memoria Compartida

Los Sistemas de Memoria Compartida, tienen múltiples procesadores que comparten el mismo espacio de direcciones de la memoria principal, esto es,

los procesadores están conectados a la misma memoria principal bajo las mismas condiciones. El usuario no está pendiente de la ubicación de los datos en memoria. Las máquinas SIMD o MIMD descritas anteriormente, pueden ser sistemas de memoria compartida; una máquina SIMD es una computadora vectorial, por contraparte, una máquina MIMD cuenta con una gran variedad en modelos, los cuales se abordarán con detalle en las siguientes secciones. Para concluir esta parte, resta señalar que se utilizará la notación SM-SIMD y SM-MIMD para las máquinas SIMD y MIMD de memoria compartida respectivamente.

Sistemas de Memoria Distribuida

Dentro de este rubro, los procesadores cuentan con su memoria local específica. Cada procesador está conectado en red y puede intercambiar datos entre sus correspondientes memorias cuando se requiera, en contraste con las máquinas de memoria compartida, el usuario tiene que mover los datos o distribuirlos cuando se necesite. Los sistemas de memoria distribuida pueden ser SIMD o MIMD; la primera clase de sistemas SIMD operan en sincronía para todas las memorias distribuidas asociadas a los procesadores, por su parte, los sistemas de memoria distribuida MIMD están basados en sus conexiones de red. Para denotar los sistemas de memoria distribuida se usará la notación DM-SIMD o DM-MIMD.

Una tendencia que ha adquirido importancia es el cómputo distribuido, partiendo del concepto de máquinas DM-MIMD. En lugar de varios procesadores alojados en un sólo lugar físico como una Work Station, Mainframe; en este caso las computadoras se conectan a una red, ya sea gigabit ethernet, canal de fibra, etc. y por medio de software, pueden procesar en red partes de la misma tarea.

Máquinas SIMD con sistema de Memoria Compartida

Esta clase de máquinas son equivalentes a las de un simple procesador vectorial (Figura 1.5). En un primer periodo, esta clase de procesador leía los operandos desde la memoria y almacenaba los resultados inmediatamente en memoria. Actualmente, utilizan registros vectoriales, lo cual ayuda a no afectar la velocidad de las operaciones, ya que proporcionan mayor flexibilidad al cargar los operandos y manipularlos con resultados intermedios.

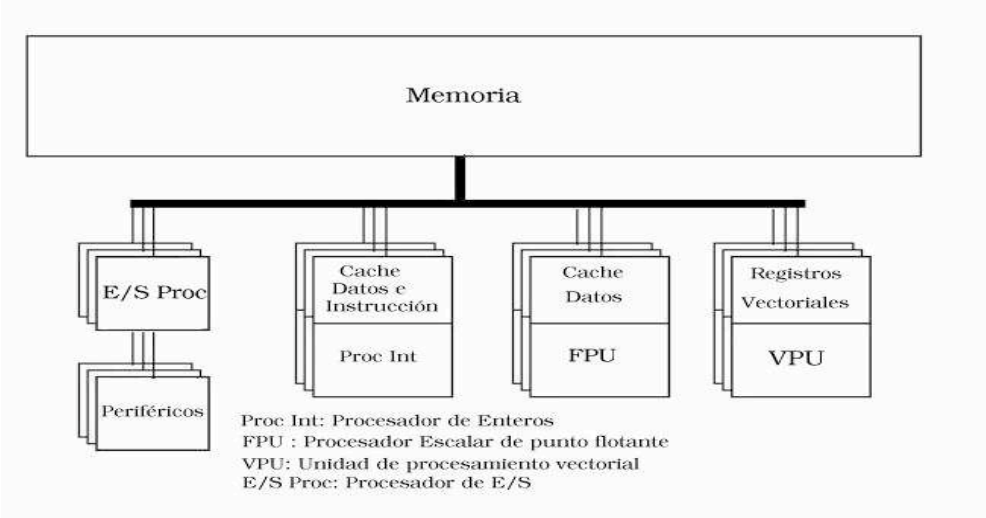

Figura 1.5: Esquema de una computadora vectorial

Como se puede ver en el esquema de la Figura 1.5, cada unidad artimeticológica vectorial consiste en un número de unidades funcionales vectoriales o "tuberías" donde una o más fluyen en la unidad vectorial. Las tuberías son diseñadas para realizar funciones de acceso a memoria, asegurando la entrega oportuna de operandos, de modo que se utilizan las tuberías aritméticas para almacenar los resultados en memoria.

Máquinas SIMD con sistema de Memoria Distribuida

Las máquinas de este tipo son conocidas como arreglos de procesadores. Un arreglo de procesadores funciona con todos los procesadores ejecutando la misma instrucción a la vez (pero en diferentes elementos de datos) sin que exista sincronización entre los mismos. Cuenta con un procesador de control que toma las instrucciones que serán ejecutadas por cada uno de los procesadores del arreglo y un procesador (front-end) el cual se conecta en el bus de datos antes de llegar al procesador de control.

Su topología de interconexión en este tipo de máquinas es una malla bidimensional (Figura 1.6). Al final de cada línea de la malla, se conecta una

línea al lado opuesto, tomando la forma de un toroide, que no es el único esquema de interconexión. Puede conectarse también en forma tridimensional, diagonalmente o en estructuras más complejas.

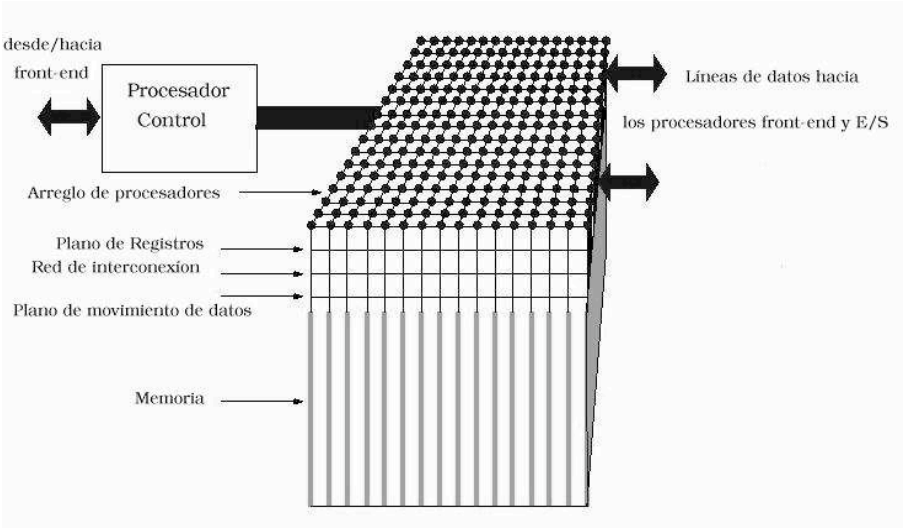

Figura 1.6: Diagrama SIMD con sistema de memoria distribuida

Máquinas MIMD con sistema de Memoria Compartida

La característica principal de los sistemas de memoria compartida es la conexión de múltiples procesadores con la memoria. Al conectar más procesadores a la memoria, lo ideal sería que el ancho de banda se incremente en función del número de procesadores; sin embargo, esto en la realidad no ocurre, pues cada procesador debe comunicarse directamente con todos sus procesadores vecinos, sin utilizar a la memoria como un estado intermedio de comunicación.

Desafortunadamente, la interconexión implica un costo que crece en orden $O(n^2)$ mientras que el número de procesadores crece en orden $O(n)$. Por ello, se han intentado varias alternativas; una red *crossbar* por ejemplo, (Figura 1.7) crece en orden $O(n^2)$ conexiones, mientras que una red Ω crece en orden $O(n\ log\ n)$ de conexiones, y el bus solamente tiene una conexión.

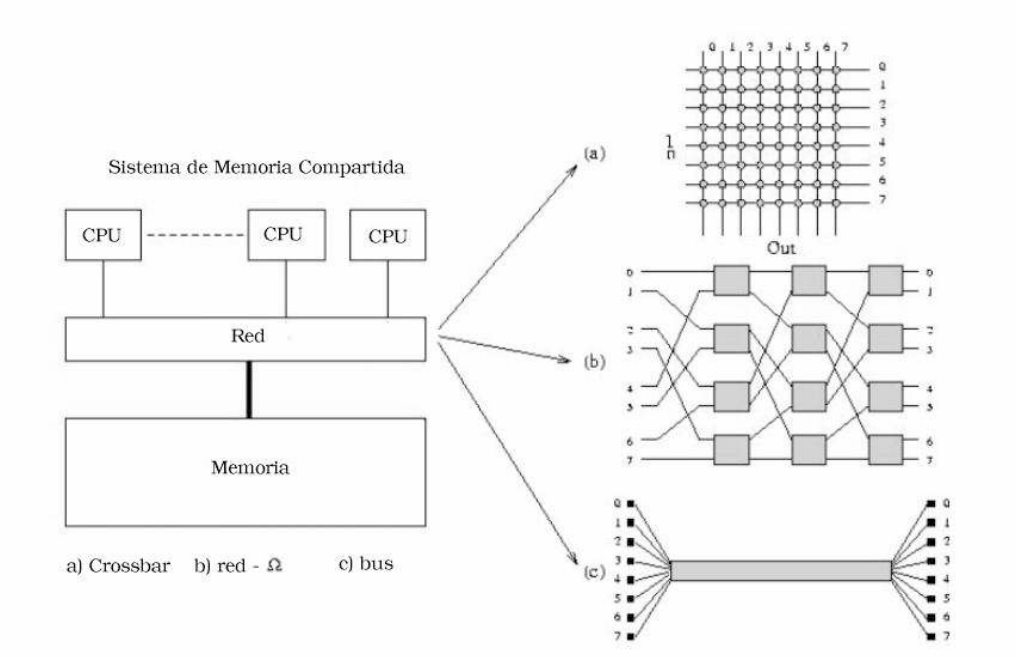

Figura 1.7: Máquinas MIMD con memoria compartida

Este costo se refleja en el uso de cada trayectoria para las diferentes interconexiones. En una red *crossbar* cada trayectoria de datos es directa y no es compartida por otros elementos. En el caso de una red Ω son (log n) estados, y varios elementos compiten por una trayectoria. Para un bus, todos los elementos comparten el mismo canal, de esta manera, n elementos compiten una vez por este en cualquier momento.

Máquinas MIMD con sistema de Memoria Distribuida

Esta familia de computadoras de alto redimiento presentan una mayor evolución. Una de las ventajas principales de estos sistemas es que, con respecto al problema del ancho de banda con los sistemas de memoria compartida, logra ser resuelto gracias a que dicho ancho de banda se escala automáticamente con el número de procesadores. De esta manera, la velocidad de acceso a la memoria es un factor crítico para los sistemas con memoria compartida.

Para que un sistema de memoria compartida obtenga un alto rendimiento comparable al de una máquina DM-MIMD, la velocidad de procesamiento debe ser tan alta como la velocidad de acceso a memoria, lo cual no tiene repercusiones en esta clase de máquinas, ya que muchos procesadores pueden comunicarse sin tener el problema del ancho de banda presentado en los sistemas de memoria compartida.

Con todo, las máquinas DM-MIMD tienen algunas desventajas. La comunicación entre procesadores es mucho más lenta que en un sistema SM-MIMD, de ahí que la sobrecarga de sincronización en el caso de las tareas de comunicación son más altas que en los sistemas de memoria compartida.

La topología y la velocidad de las rutas de datos en las máquinas DM-MIMD son indispensables para el sistema. Como en el caso de las máquinas SM-MIMD, la estructura de conexión balancea los costos. Existen varias estructuras de conexión, pero solo algunas son utilizadas en la práctica. Una de ellas es la topología de hipercubo (Figura 1.8 (a)).

Muchos de los sistemas DM-MIMD emplean *crossbars*. Por ejemplo, en una red de árbol (Figura 1.8 (b)) para un número pequeño de nodos (hasta 64 nodos) se utiliza un *crossbar* de 1 estado, mientras que para conectar un gran número de procesadores se utilizan los *crossbar* multiestado. Un *crossbar* multiestado sirve para conectar un *crossbar* de un nivel con otro, en lugar de conectar directamente los nodos más distantes de la topología. Así, se pueden conectar cientos de nodos a través de varios estados.

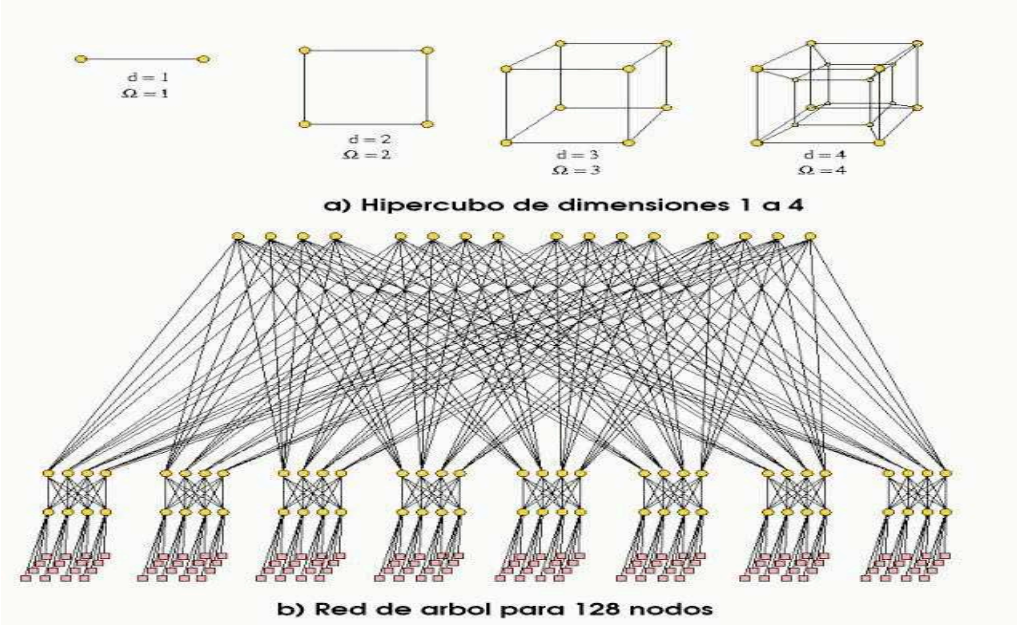

Figura 1.8: Máquinas MIMD con memoria distribuida

Máquinas CC-NUMA

La tendencia actual es construir sistemas pequeños de hasta 16 procesadores, donde integrados en un bloque, llamado sistema multiprocesador simétrico (SMP). Estos procesadores en el bloque son interconectados virtualmente (Figura 1.9) vía un crossbar de 1 estado en el SMP. Cada SMP se interconecta por una red de menor costo.

Lo anterior es equivalente a los ensambles de grandes sistemas de máquinas DM-MIMD, con la diferencia que todos sus procesadores pueden acceder a todas las direcciones en memoria simultáneamente Por ello, a este tipo de máquinas se les puede considerar SM-MIMD. Empero, debido a la memoria físicamente distribuida, no hay garantía de que las operaciones de acceso se satisfagan en el tiempo requerido.

En este sentido las máquinas se denominan sistemas CC-NUMA, (Coherencia de Cache- Acceso a la Memoria No Uniforme).

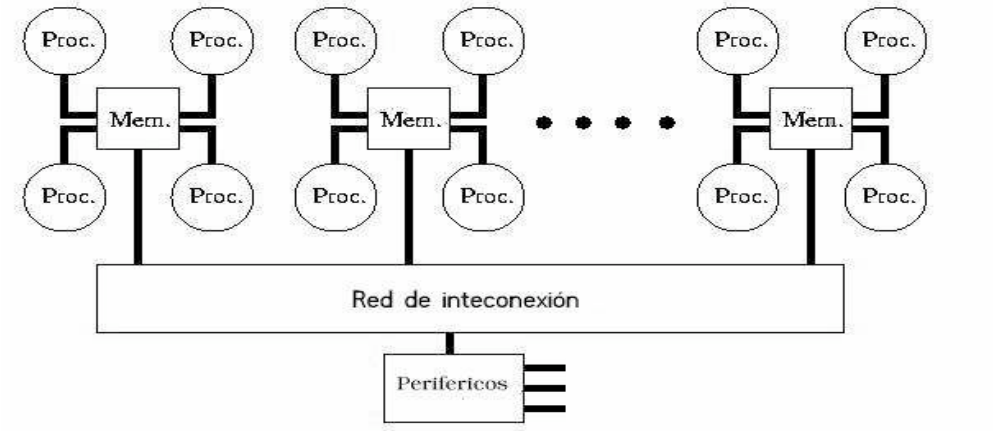

Figura 1.9: Diagrama a bloques de una conexión crossbar

Clusters

Los clusters son sistemas que evolucionaron a partir del primer sistema de este tipo, el cual se denominó Beowulf y fue desarrollado en 1994. Consisten

en un conjunto de estaciones de trabajo o pc conectadas en una red local. La ventaja principal de un cluster es el bajo costo tanto en hardware y software, además de la flexibilidad en el control del sistema para los programadores y usuarios.

Actualmente existe un mercado atractivo para clusters, las grandes compañías dedicadas al cómputo de alto rendimiento (HPC) ofrecen esta clase de sistemas para aquellos grupos de investigación u otro ámbito que no pueden construirlos. Ahora bien, existe una diferencia principal entre los clusters y los sistemas integrados: que los clusters pueden utilizarse en situaciones donde no hay alternativas para adquirir sistemas integrados debido a su gran costo.

A pesar de sus beneficios, los clusters cuentan con varias desventajas: la velocidad de la interconexión, el rendimiento de la carga de trabajo de las diversas aplicaciones y el tamaño de la memoria principal.

Para solucionar esto, se han desarrollado herramientas que ayudan a solventar dichas desventajas. Las diferencias entre los clusters y los sistemas integrados DM-MIMD o SM-MIMD disminuyen debido a la velocidad de los procesadores, así como a la creación de herramientas de software para clusters por parte de los vendedores de sistemas DM-MIMD.

1.2.3. Futuro

Dos cosas permanecen constantes en el cómputo de alto rendimiento:

- La necesidad constante por más poder computacional (mayor velocidad de procesamiento y ancho de banda de comunicación),
- La interfaz más simple pero más completa para utilizar todos los recursos disponibles (mayor facilidad de interacción entre humano-máquina).

Por ello, en años anteriores, se ha puesto atencion en las Grids computacionales. El término Grid se utilizó en los años noventa para definir una infraestructura avanzada en cómputo distribuido y satisfacer las necesidades de capacidad computacional en las ciencias e ingeniería. El problema real del concepto Grid, es la coordinación de compartir recursos y problemas entre las organizaciones virtuales. Una organización virtual proporciona servicios para aplicaciones, almacenamiento, y permite un aprovechamiento en cómputo para la solución de problemas en conjunto [veáse 15].

Capítulo 2

SISTEMAS DE ARCHIVOS

Todas las computadoras necesitan almacenar y recuperar datos, así pues, mientras un proceso está en ejecución debe ser capaz de recuperar y almacenar datos dentro de su espacio de direcciones. El tamaño de los datos se limita al espacio de direcciones posibles para cada proceso, por ello se deben guardar los datos dentro del espacio de direcciones cuando el proceso finalice, de lo contrario la información se perdería. Los datos usualmente se guardan por semanas, meses o por periodos más largos.

En un sistema multiprocesador, varios procesos acceden a subconjuntos de datos simultáneamente. Por ejemplo, si un proceso tiene un libro almacenado dentro de su espacio de direcciones, dicho proceso es el único que accede a los datos. Una manera de resolver este problema es independizar la información de los procesos. Como consecuencia de lo anterior, existen 3 requerimientos para almacenar datos [véase 4]:

- Debe ser posible almacenar una gran cantidad de datos.
- Los datos deben permanecer después de la terminación del proceso que los utilizó o creó.
- Múltiples procesos deben ser capaces de acceder a los datos de manera concurrente.

Para cumplir con estos requerimientos, es necesario almacenar los datos dentro de discos o medios externos de almacenamiento en unidades llamadas archivos. Los procesos pueden leer y escribir nuevos datos en los archivos si se requiere. Los datos almacenados en los archivos deben ser persistentes,

es decir, no ser afectados por la creación y terminación de los procesos. Un archivo debe desaparecer cuando su dueño explícitamente lo remueva.

Los archivos son administrados por una parte del sistema operativo denominado sistema de archivos, que se verá en los siguientes puntos.

2.1. Estructuras de los Sistemas de Archivos

En la organización del sistema de archivos, para los sistemas operativos unix (SunOs, Unixware, Irix, Linux, MacOs, AIX, entre los más importantes) cada disco se divide en una o más particiones. Cada partición contiene un sistema de archivos. Un sistema de archivos se describe por su superbloque, que está localizado al inicio de cada una de las particiones del disco.

El superbloque contiene datos críticos, por lo que se guarda para futuras fallas en los discos. El superbloque se genera en el momento que un sistema de archivos se crea. Los datos del superbloque no cambian, contienen los parámetros básicos del sistema de archivos e incluyen el número de bloques de datos del sistema de archivos, así como un contador para el máximo número de archivos y un puntero hacia una lista ligada, que hace referencia a los bloques libres del sistema de archivos [véase 22].

2.1.1. I-nodo

Los archivos tienen asignado un descriptor denominado i-nodo [veáse 18], cuya estructura se describe a continuación:

- Informacion estática: el i-nodo contiene el tamaño del archivo, permisos, dueños, tiempos de acceso y un contador de ligas,
- Lista de entradas de directorios cache: una lista de todas las entradas de directorio que apuntan al i-nodo,
- Operaciones de i-nodo: se definen las operaciones que actúan en cada i-nodo incluyendo la búsqueda, creación, ligas simbólicas, lectura de ligas, renombramiento, creación de directorio, eliminación de directorio, eliminación de ligas, creación de i-nodos, entre otros.

2.1.2. Entradas de directorio

Un directorio sirve para realizar las búsquedas y manipulación de i-nodos eficientemente. Al buscar un nombre de archivo en particular, primero se encuentra la entrada del directorio y se efectúa una búsqueda para localizar el i-nodo asociado. Si la entrada no se encuentra en el directorio, se recurre a una búsqueda en el sistema de archivos [véase 18].

2.1.3. Sistema de Archivos Virtual (VFS)

El sistema de archivos virtual es una arquitectura para colocar múltiples implementaciones de sistemas de archivos bajo un sistema operativo [véase 20]. Los objetivos de la arquitectura son:

- Independizar la implementación lógica del sistema de archivos de la implementación del sistema de archivos por parte del sistema operativo, de esta manera proporciona una interfaz bien definida entre las dos partes.
- La interfaz debe utilizarse, en el caso de un sistema de archivos de red, por el servidor de archivo para satisfacer las peticiones del cliente.
- El conjunto de operaciones de la interfaz no debe bloquear las operaciones entre archivos. Si se requiere un bloqueo debe realizarse desde la implementación del sistema de archivos en el sistema operativo.

Dicha implementación lógica del sistema de archivos se logra a través de VFS, que hace referencia a la capa de i-nodos que es parte de la implementación por el sistema operativo, a través de descriptores lógicos denominados v-nodos.

2.1.4. Disco Cache (Cache Disk)

Existen varias razones por las que la multiprogramación se limita a traslapar la actividad de procesamiento junto con la actividad de E/S, entre las más importantes se encuentran:

- Los tiempos de acceso a disco es 100 veces en promedio menor respecto a la velocidad de procesamiento.

- La tasa de transferencia de E/S asociada con la aplicación permanece proporcional a la velocidad de procesamiento.

En consecuencia, se crean cuellos de botella en la actividad de E/S, por lo que se implementa un mecanismo como es el disco cache [véase 3], ya que funciona como una parte de la memoria (buffer) y es utilizado para almacenar porciones de contenido del espacio de direcciones en disco. Este buffer tiene las siguientes funciones:

- Capturar una fracción de las operaciones de E/S.
- No ocupar tiempo de procesamiento y un gran espacio en memoria.

Dichas funciones proporcionan un mejor rendimiento en la actividad de E/S, al predecir o eliminar los cuellos de botella.

2.1.5. Bloqueo de Archivos (File locking)

En ocasiones los procesos necesitan sincronizar las modificaciones de un archivo, utilizando un bloqueo por separado en el mismo. Un proceso puede intentar dicho bloqueo, si su creación es viable el proceso puede proceder con su modificación. En el caso de que hubiese una falla entonces el proceso debe esperar e intentar nuevamente [véase 22].

Empero, este mecanismo tiene su desventaja: los procesos consumen tiempo del CPU para esperar e intentar bloqueos. Existen 2 clases de esquemas de bloqueo: bloqueos duros y bloqueos de advertencia, donde la diferencia consiste en la prioridad de la aplicación. Un bloqueo duro siempre se aplica cuando un programa intenta acceder a un archivo; mientras que un bloqueo de advertencia solo se da cuando es requerido por la aplicación. Las facilidades del bloqueo en archivos se presentan cuando los procesos realizan bloqueos compartidos y exclusivos. Solo un proceso puede tener un bloqueo exclusivo en un archivo cuando múltiples bloqueos compartidos puedan estar presentes, de ahí que un proceso no puede tener bloqueos exclusivos y compartidos simultáneamente en un archivo.

2.1.6. Distribución de los datos en el disco

La distribución de los datos depende del sistema de archivos que se utilice. Es importante mencionar, que existen varios sistemas de archivos y esos se describirán en la siguientes secciones.

Sistema de Archivos Rápido (FFS)

El Sistema de Archivos Rápido (Fast File System) (Figura 2.1) consiste en dividir al disco en áreas denominadas grupos de cilindros. En cada cilindro del grupo se cuenta con una estructura de índices (i-nodos) que contienen las direcciones en disco de los bloques de datos directos, indirectos, dobles indirectos y triplemente indirectos.

De este modo, en cada cilindro del grupo se cuentan los bloques de datos disponibles y se genera informacion descriptiva del porcentaje de uso en cada grupo [veáse 22].

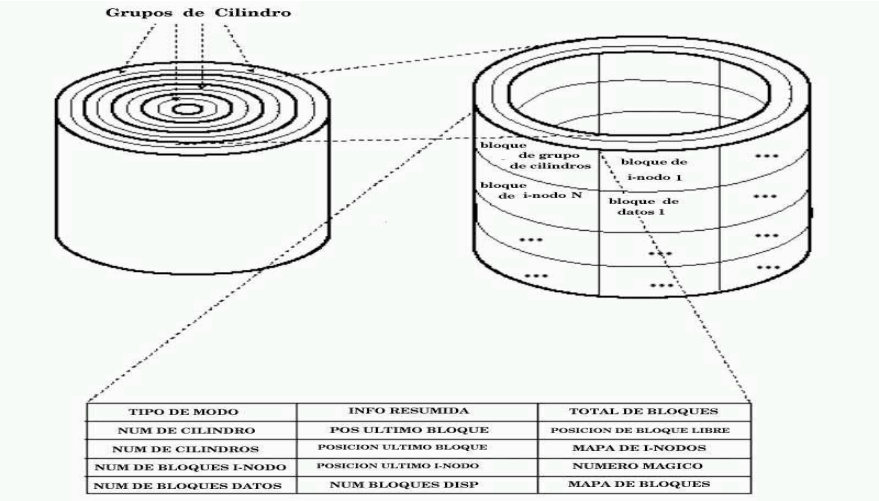

Figura 2.1: Esquema del sistema de archivos rápido (FFS)

Sistema de Archivos por Registros Estructurados (LFS)

El sistema de archivos por registros estructurados (Log File System) (Figura 2.2) tiene como objetivo mejorar el rendimiento del sistema de archivos FFS al almacenar todos los datos en forma de registros.

La disposición de datos en LFS y FFS es la misma, pero LFS realiza sus operaciones de E/S en forma secuencial como si fueran registros de una

base de datos e incorpora la estructura de i-nodos en cada registro para una recuperación eficiente de los datos [véase 21].

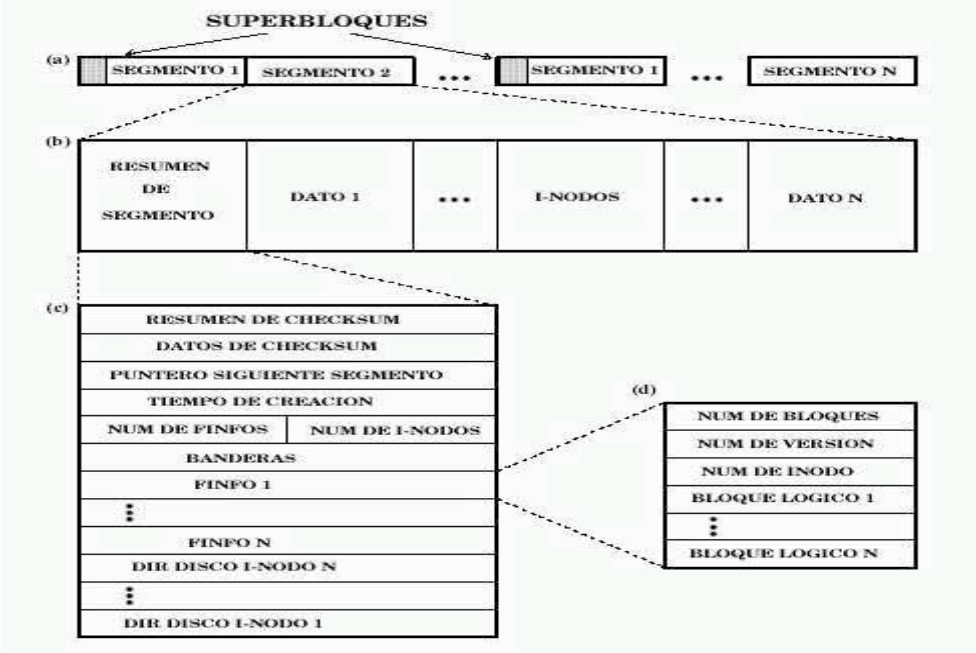

Figura 2.2: Esquema del sistema de archivos por registros estrucurados (LFS)

Sistema de Archivos por Bitácoras (JFS)

El Sistema de Archivos por Bitácoras (Journaling File System) consiste en la conservación de las versiones incompletas anteriores y recientes de los bloques de datos para escribir las versiones más recientes en una localidad separada del disco, hasta que se realice la última modificación, posteriormente se escriben las versiones más recientes en sus localidades en disco [véase 18, 38].

Las versiones más recientes de los metadatos se registran en un área reservada del disco denominada bitácora (journal). Esta bitácora consiste en un registro de los contenidos de bloques de metadatos mientras se realizan las transacciones.

2.2. Sistemas de Archivos en Red

Los sistemas de archivos en red, surgieron con los sistemas de tiempo compartido. Esto debido a la necesidad de compartir el uso de periféricos de costo elevado, como los grandes discos, impresoras, sistemas de cinta y archivos.

La forma de compartir recursos en una red de computadoras no es la misma que en un sistema de tiempo compartido, en este ambiente cada estación de trabajo tiene su dirección de red específica como identificación para acceder a los archivos compartidos. Además se requiere de acciones explícitas por parte del usuario para compartir archivos. Antes de utilizar un archivo, se ejecuta un programa para transferirlo desde el nodo de la red en el que se encuentra hasta su localidad.

2.2.1. Sistemas de Archivos Remotos

Un sistema de archivos remoto es el conjunto de funciones remotas (RPC's) que permiten acceder a los archivos desde una máquina cliente a una máquina servidor. Estas funciones remotas son independientes de la arquitectura, sistema operativo, red y protocolos de transporte.

Debido a la diversidad de los sistemas, el protocolo de transporte no concuerda exactamente con cada sistema existente. Por ello se proporcionan las bases para su portabilidad e interoperabilidad.

Como ejemplo está NFS (Network File System), que alcanza la independencia de la arquitectura del sistema operativo a través de una separación estricta del protocolo y su implementación [véase 6].

La meta principal de NFS es proporcionar rendimiento especialmente a través de la escritura. El rendimiento se logra por medio de las siguientes características:

- Escrituras asíncronas confiables.
- Atributos en todas las réplicas.
- Una débil consistencia de cache para permitir al cliente una mayor eficiencia en las transferencias entre caches.

2.2.2. Sistema de Archivos Distribuidos

Un sistema de archivos distribuido se implementa para la cooperación de un conjunto de computadoras servidores para operar como un solo sistema. Las computadoras restantes en la red, que utilizan el sistema de archivos distribuido, se les denomina clientes. Una razón para utilizar un sistema de archivos distribuido es economizar el uso de los dispositivos de almacenamiento. El uso de computadoras pequeñas como servidores de archivo permite un rango amplio de capacidad de almacenamiento. Es posible expandir un sistema de archivos distribuido, desde una simple computadora con un disco hasta varios servidores con varios discos [véase 35].

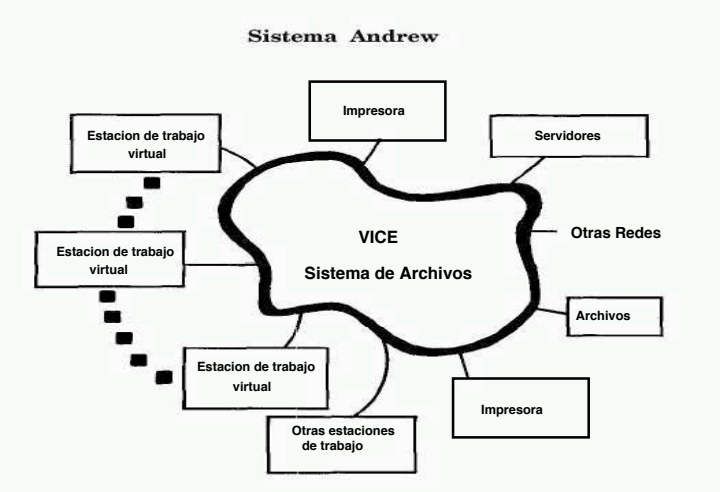

Figura 2.3: Sistema Andrew

Como ejemplo está AFS(Andrew File System) (Figura 2.3) donde los archivos completos se guardan en una estación de trabajo. Cuando un programa de aplicación realiza una llamada para abrir un archivo, el sistema operativo de la estación de trabajo analiza la petición para determinar si el archivo es local o compartido [véase 17]. Existe un proceso llamado Venus, este maneja el disco cache local y la comunicación con el servidor remoto, si un archivo es local o compartido Venus revisa el cache para verificar si existe una copia. Posteriormente la petición de abrir archivo se trata como una petición para abrir una copia en el cache. Si el archivo no está presente en el cache o la

copia no es la actual, se toma la versión más reciente de la copia desde el servidor de archivos apropiado.

2.2.3. Sistemas de Archivos Paralelos

Un sistema de archivos paralelo se define como un sistema de archivos en red, donde se utilizan múltiples servidores de archivos simultáneamente. La meta del sistema es proporcionar una gran disponiblidad y canal de transferencia que no puede ser alcanzado con un solo servidor. El cliente reparte el archivo a través de los servidores en diferentes piezas de datos que se almacenan en varios servidores.

Esto difiere de los sistemas de archivos remotos y distribuidos, donde el ancho de banda de E/S se limita al rendimiento de un solo servidor de archivos junto con su ancho de banda en memoria y su velocidad de procesamiento; la interfaz de red, los buses de E/S y la velocidad de los discos. La meta de tales sistemas de archivos radica en la posibilidad de utilizar conjuntos de computadoras conectadas por redes de alta velocidad para ejecutar aplicaciones masivamente paralelas. Estos sistemas alcanzan rendimientos de E/S semejantes a los de una supercomputadora.

Como ejemplo está PVFS (Parallel Virtual File System) dedicado a aplicaciones paralelas que requieren E/S intensiva. PVFS es un sistema de archivos paralelo para clusters Linux. Se ha tomado como herramienta para explorar el diseño, la implementación y el uso de aplicaciones que requieren E/S en paralelo [véase 32].

Para lograr una E/S en paralelo se necesita un sistema de archivos con una gran velocidad para las operaciones de E/S, una visión coherente de los datos y una interfaz para el manejo de los mismos.

De acuerdo a la Figura 2.4 PVFS tiene 3 componentes:

- Servidor de Metadatos (MGR): se encarga de controlar los metadatos en los archivos. Los metadatos contienen el nombre del archivo, la localización en el directorio, dueño, permisos y como están distribuidos en los nodos dedicados a E/S en el cluster. El servidor de metadatos comúnmente se sitúa en el nodo maestro del cluster.

- Servidor de E/S (IOD): se encarga del control de E/S en el dispositivo de almacenamiento local. Además manipula los archivos para su creación, eliminación o modificación. Finalmente, es compatible con los sistemas de archivos locales: ext2, ext3 y reiser fs.

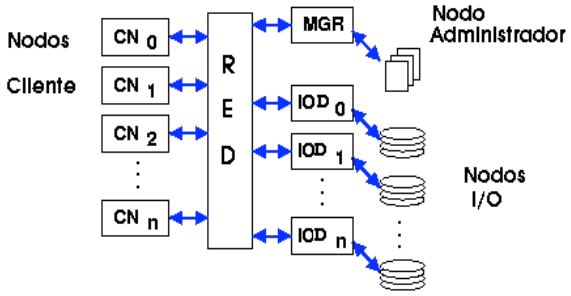

Figura 2.4: Esquema del sistema de archivos PVFS

- Libpvfs: se utiliza como medio de comunicación entre el proceso cliente (CN) y el servidor pvfs (MGR). Además permite el acceso al sistema de archivos en cada nodo cliente y de esta manera las aplicaciones acceden a los archivos sin modificación alguna.

2.3. Dispositivos de Almacenamiento

Los primeros sistemas de procesamiento utilizaron la cinta magnética como medio de almacenamiento. Las mejoras en el costo, capacidad y el rendimiento de los dispositivos de almacenamiento ha ocasionado el crecimiento de los sistemas y su capacidad de aplicación [véase 34].

En las siguientes secciones se proporciona una descripción de los dispositivos de almacenamiento utilizados en la actualidad.

2.3.1. Sistema RAID

Los usuarios de las computadoras se han beneficiado con el contínuo aumento en la velocidad de las mismas. Por ejemplo, a partir de la aparición de los chips se mejoró el rendimiento en un 40 % por año con respecto a sus versiones anteriores. Sin embargo, un procesador rápido no hace que un sistema sea rápido si consideramos a la transferencia de datos entre un procesador y memoria utilizando la regla: *"Cada instrucción por segundo del procesador, necesita un byte de la memoria principal"*.

Sin duda, resulta claro que la capacidad y la velocidad son factores determinantes para encontrar el rendimiento del sistema. Por ejemplo, la velocidad de la memoria principal se ha mantenido por 2 razones:

- Con la invención del cache, un buffer se puede manejar automáticamente para contener una fracción substancial de las referencias en memoria.
- La tecnología SRAM utilizada para construir los caches, aumenta su velocidad en el rango de 40 % a 100 % anualmente.

En contraste con las tecnologías en la memoria principal, el rendimiento de los discos aumenta de manera moderada. Con la aparición de las computadoras personales, ha mejorado la rapidez de los discos debido a la demanda creciente del mercado, sin embargo debido a su vulnerabilidad se tienen que realizar respaldos de la información en caso de falla.

Para superar los problemas de rapidez y confiabilidad, se utilizan los discos para que contengan información redundante para recuperar la información original cuando se dañe un disco y además obtener un buen rendimiento en los patrones de acceso a disco.

El acrónimo RAID significa "arreglo de discos removibles redundantes". Se tiene una clasificación de 5 organizaciones diferentes de arreglos de discos, que van de los discos espejo y progresa a través de una variedad de alternativas con diferentes niveles de rendimiento y confiabilidad. A cada organización se le denomina como nivel RAID. Estos niveles RAID pueden implementarse tanto en software como en hardware. Cada configuración contiene los siguientes parámetros:

- D = número total de discos con datos (no incluyen a los discos de verificación).
- G = número total de discos de datos en un grupo (no incluye los discos de verificación).
- C = número total de discos de verificación del grupo.
- n_G = número de grupos = D/G.

Nivel 1 RAID (Discos espejo)

Este nivel considera a los discos con su duplicado ($G = 1$,$C = 1$). Por cada escritura al disco de datos también se escribe al disco de verificación. Una versión optimizada sería considerar a los discos por parejas y doblar el número de controladores para tolerancia a fallas y tomar lecturas que ocurran en paralelo.

Las transferencias de un sistema RAID (Figura 2.5) comunmente son accesos individuales o agrupados. Cuando los accesos son individuales estos se realizan directamente en cada disco del grupo. Si los accesos son agrupados estos se realizan en todos los discos del grupo simultáneamente. De esta forma el ancho de banda es constante y se distribuye de manera uniforme, y por medio de paralelismo se reduce la cola de retardo.

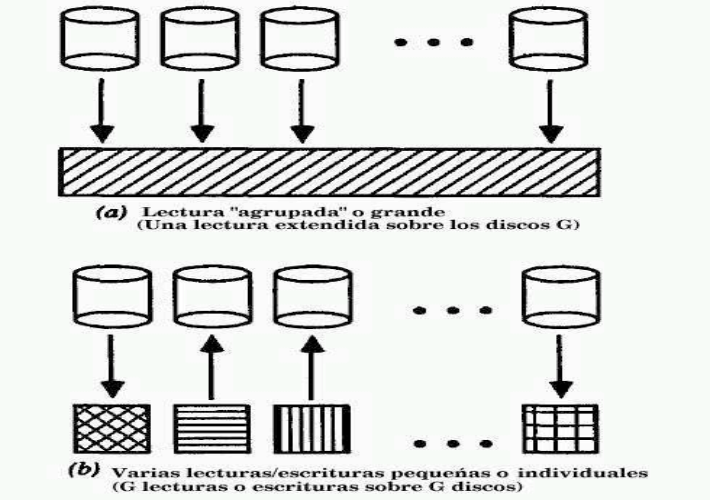

Figura 2.5: Tranferencias de un sistema RAID .

En este nivel se considera a un disco de datos por grupo. Para realizar grandes transferencias se recomienda que exista el mismo número de discos por grupo en cada nivel RAID. Al duplicar el número de discos se puede doblar la carga del sistema e utilizar solo el 50 % de la capacidad de almacenamiento en disco.

Nivel 2 RAID (Código de Hamming para corrección de errores)

Este nivel tiene como antecedente las organizaciones de memoria principal que aportan una manera de reducir el costo de la confiabilidad. En un sistema con varios dispositivos DRAM, se accede a los datos en grupos de 16 a 64 chips a la vez, por ello se agregaron chips con datos redundantes para corregir errores y detectarlos en cada grupo.

Debido a que algunos accesos son a grupos de discos, en vez de leer al grupo completo para verificar si la información es correcta, se realiza lo que en un sistema DRAM interpolando los datos de los discos de un grupo por medio del código de Hamming y escribirlos en varios discos de verificación.

Para los accesos agrupados, el nivel 2 tiene el mismo rendimiento que el nivel 1 debido a que en este nivel se utilizan pocos discos de verificación. Para accesos individuales, el rendimiento es bajo debido a que todos los discos del grupo deben accederse para una paqueña transferencia limitando el número de accesos simultáneos a n_G. Este nivel es recomendable para aplicaciones de supercómputo, pero inapropiado para aplicaciones basadas en transacciones. Al incrementar el número de discos de datos del grupo, se incrementan las diferencias entre los dos tipos de aplicaciones.

Nivel 3 RAID (Sólo un disco de verificación por grupo)

Este nivel considera necesario a un sólo disco con información redundante para detectar un error en los discos de datos. Esto debido a que las señales especiales proporcionadas en la interfaz de los discos o de la información extra de validación al final de cada sector, se utilizan para detectar y corregir errores.

El rendimiento de este nivel es el mismo que el nivel 2, aunque el rendimiento por disco se incrementa, debido a que solo existe un disco de verificación (C=1). Una ventaja del nivel 2 sobre el nivel 3, radica en que la información de verificación asociada con cada sector del disco para corrección de errores no es necesaria, incrementando la capacidad en disco al menos en un 10 %.

Nivel 4 (Lecturas y escrituras independientes)

Este nivel, al dividir una transferencia entre todos los discos de un grupo, se obtiene la ventaja de que el tiempo de transferencia de una operación agrupada se reduce, debido a que el ancho de banda de transferencia del

arreglo de discos está distribuido. Sin embargo, se cuenta con las siguientes desventajas:

- Leer o escribir hacia un disco en un grupo requiere que se lea o escriba a todos los discos en un grupo. Los niveles 2 y 3 RAID, pueden realizar solo una operación de E/S por grupo.
- Si los discos no están sincronizados, no se aprecian los retardos de rotación, ni los tiempos promedio de las funciones de acceso, por lo que los retardos pueden tender al peor de los casos.

Este nivel 4 (Figura 2.6) se distingue por mejorar el rendimiento de las transferencias pequeñas a través del paralelismo, debido a la capacidad de realizar más de una operación de E/S por grupo simultáneamente.

Debido a que solo hay un disco de verificación, este tiende a convertirse en un cuello de botella, cuya solución se presenta en el último nivel.

NIVEL 5 RAID (Ningún disco de verificación)

En el nivel anterior, se alcanza el paralelismo por medio de las lecturas. Sin embargo, las lecturas están limitadas a una por grupo, donde cada escritura debe ser leída y escrita hacia el disco de verificación. El último nivel RAID distribuye los datos y la información de verificación a través de todos los discos del grupo. Esta situación se muestra en la Figura 2.7 donde la organización de la información de verificación del nivel 4 se compara con la información en el nivel 5.

El efecto en el rendimiento es grande, porque este nivel puede soportar múltiples escrituras individuales por grupo. Esto hace que el último nivel, esté cerca de dos entornos de aplicación: el de la lectura-modificación-escritura en bloques pequeños enfocados a la velocidad por disco como en el nivel 1, y por otro lado, el rendimiento de las grandes transferencias por disco y alto porcentaje de capacidad de almacenamiento utilizable como en los niveles 3 y 4. Dividiendo los datos entre todos los discos mejora el rendimiento de las lecturas pequeñas y existe más espacio en disco por cada grupo que contenga discos de datos.

El sistema RAID ofrece una opción para el crecimiento exponencial de la velocidad de acceso a memoria y de transferencia hacia el procesador. Con las ventajas de rendimiento, confiabilidad, consumo de energía y crecimiento modular se da pauta para nuevas tecnologías por descubrir.

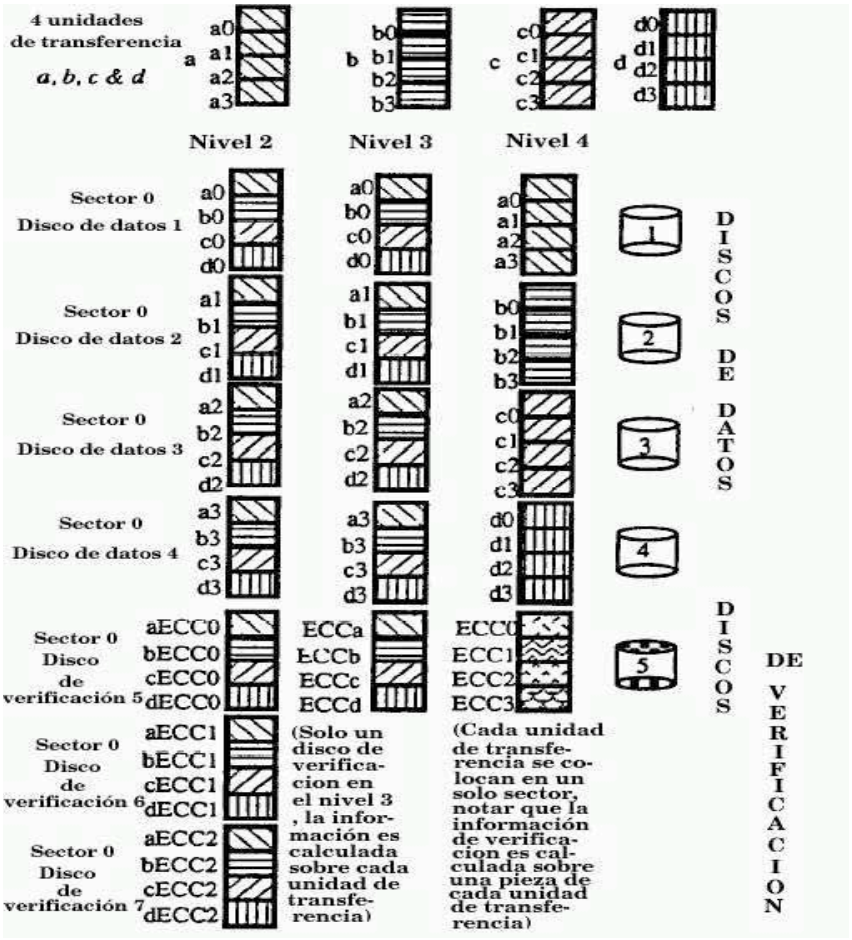

Figura 2.6: Se muestra la localización de los datos, y la información de verificación entre los niveles 2,3 y 4 cuando $G = 4$, debe notarse el aumento mínimo de información para la verificación por sector cuando se agrega un controlador de disco para detectar y corregir los errores de algún sector.

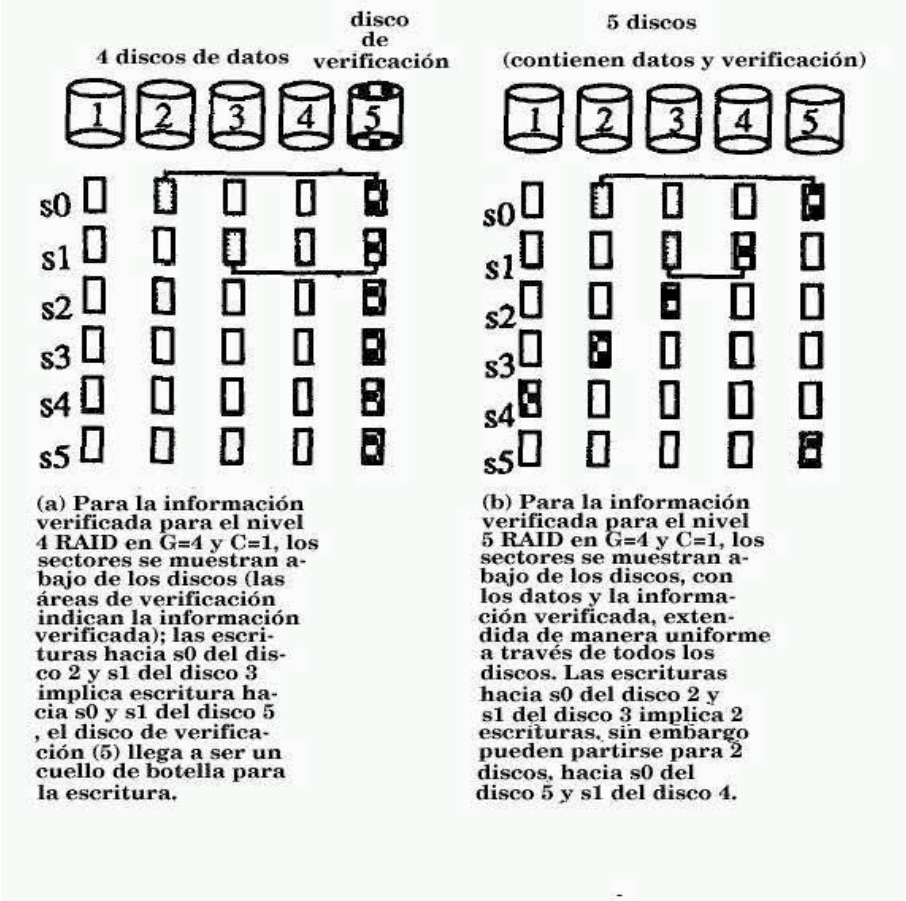

Figura 2.7: Información de verificación del nivel 4 en comparación del nivel 5.

2.3.2. Sistemas NAS y SAN

Las tecnologías SAN (Storage Area Network) y NAS (Network Attached Storage) reemplazan a los servidores tradicionales de archivos. El objetivo de dichos sistemas es ofrecer un rango amplio de servicios, un incremento en la flexibilidad y un almacenamiento sencillo a un bajo costo. Ambas tecnologías SAN y NAS proporcionan un rendimiento para diversos tipos de aplicaciones [véase 7].

Características de SAN

La infraestructura de red para un sistema SAN, se diseña para proporcionar un ambiente flexible, de alto rendimiento con alta escalabilidad. El sistema SAN alcanza este objetivo por medio de varias conexiones directas entre los servidores de archivos y los dispositivos de almacenamiento (Figura 2.8), por ejemplo, los sistemas RAID y los sistemas de archivos en masa MSS.

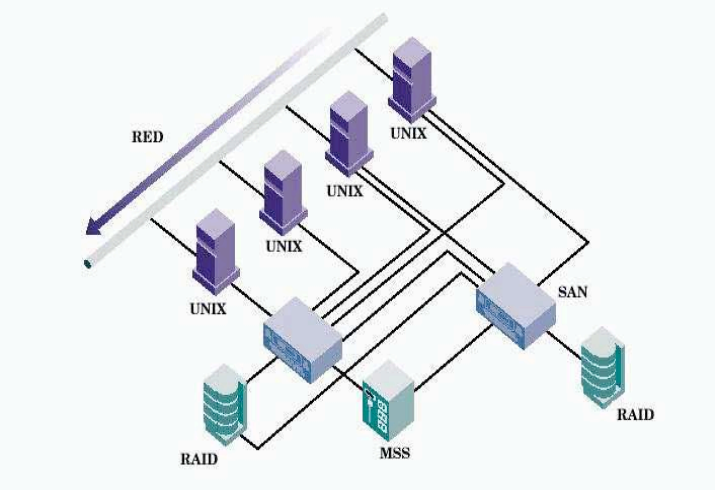

Figura 2.8: Configuración del sistema SAN

Características NAS

Los sistemas NAS se configuran típicamente como una aplicación de servidor de archivo (Figura 2.9), que son accedidas por estaciones de trabajo y servidores a través de un protocolo como TCP/IP y aplicaciones como NFS para acceder a los archivos.

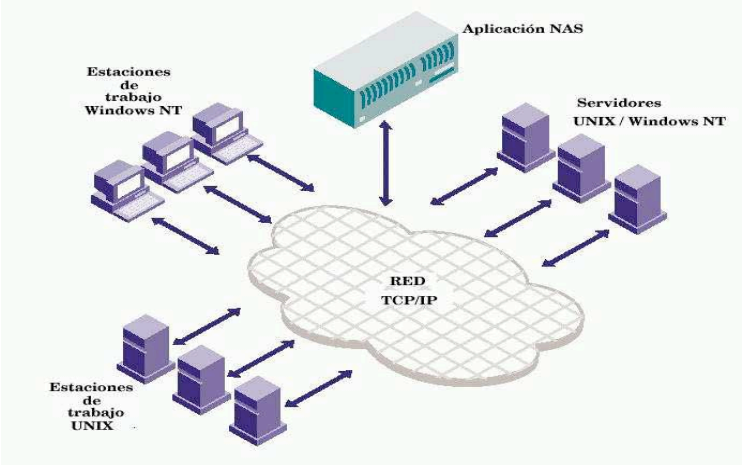

Figura 2.9: Configuración del sistema NAS

Beneficios SAN y NAS

Las grandes aplicaciones se benefician al emplear un sistema SAN, por ejemplo, las grandes simulaciones y aplicaciones de misión crítica, con la garantía de un almacenamiento, respaldo y alta disponibilidad eficiente. En contraste, el soporte de NAS comparte archivos con múltiples pares de clientes. Debido a que el acceso hacia los archivos es de bajo volumen y menos sensible al tiempo de respuesta, el rendimiento y la distancia es de menor importancia en un sistema NAS.

Capítulo 3

BIBLIOTECAS PARALELAS PARA LAS OPERACIONES EN ARCHIVOS

La demanda por recursos computacionales, orilló al uso de varios procesadores y así se crearon las computadoras paralelas. El uso de las computadoras paralelas impulsaron el desarrollo del hardware y el sofware. En cuanto al software, se aplicó la teoría de los procesos distribuidos para el uso de varios procesadores y más adelante apareció el paralelismo para comunicar varios procesadores entre sí. A lo largo del tiempo, se desarrollaron bibliotecas paralelas por parte de las compañías dedicadas a la construcción de supercomputadoras. Para lograr portabilidad entre varías computadoras de Alto Rendimiento, se crearon los estándares de bibliotecas paralelas. En este capítulo, se describe la teoría básica de los procesos paralelos y distribuidos, así como la biblioteca paralela estándar bajo el esquema de envío de mensajes. En esta biblioteca nos enfocaremos a las funciones paralelas especializadas en escritura/lectura en archivos y se describen las diferentes implementaciones para varios sistemas de archivos descritos en el capítulo anterior.

3.1. Introducción

El término paralelismo designa a una técnica aplicable con varios procesadores para resolver un problema [véase 8] Es común que los problemas del ámbito científico y de la ingeniería requieran capacidades computacionales

que rebasan las proporcionadas por un procesador único; es por ello que se han desarrollado las aplicaciones paralelas.

A pesar de que el desarrollo de la tecnología de las computadoras paralelas se incrementa constantemente, el desarrollo de aplicaciones aumenta de manera moderada por dos razones principalmente: en primer lugar, dependen de las herramientas de software especializado y por otro lado, los desarrolladores de software paralelo se enfrentan a problemas diferentes a los de la programación secuencial: no determinismo, comunicación, sincronización, partición de datos, balanceo de carga, tolerancia a fallas, sistema de memoria, bloqueos (*deadlocks*) y condiciones de competencia por mencionar algunos.

El cómputo paralelo tiene más de 20 años de desarrollo y resulta poco sorprendente que a futuro inmediato se convierta en parte central para las aplicaciones prácticas de cómputo [veáse 9]. La importancia del paralelismo se debe principalmente a las siguientes razones:

- El mundo real es inherentemente paralelo, la naturaleza realiza sus cálculos de manera paralela o al menos en una manera que no impida el paralelismo.
- El paralelismo dispone de un rendimiento mayor comparado con el caso de un solo procesador.
- Es cercano el rendimiento (Ley de Moore) para un solo procesador [véase 1]. En los siguientes años los procesadores ópticos podrían dar un gran salto en el rendimiento y ser la tecnología de vanguardia para las computadoras.
- Cada generación de procesadores aumenta en un 10 % su rendimiento en comparación con las versiones anteriores En consecuencia, varios sistemas uniprocesador interconectados y con procesadores de generaciones previas, son capaces de obtener rendimientos aceptables (Cuadro 3.1).

3.2. Procesos Distribuidos

El concepto de procesos distribuidos, se utilizó inicialmente en la programación concurrente y fue propuesto para aplicaciones de tiempo real. Dichas

Cuadro 3.1: *Rendimiento de los procesadores* $(1964 - 2005)$

Ley de Moore

año		Megaflops
	Computadoras	
1964	CDC6600	0.48
1969	CDC7600	3.3
1976	Cray-1	110
1982	Fujitsu VP-200	500
1984	Cray-XMP	$1,000$
1985	Cray-2	$2,000$
1990	NEC-SX3	$23,000$
1993	Fujitsu VPP5000	$170,000$
1997	ASCI Red	$1,068,000$
1998	ASCI Blue Mountain	$1,608,000$
2000	ASCI White	$7,304,000$
2002	Earth Simulator	$35,860,000$
2004	SGI Altix	$51,873,000$
2005	Blue Gene/L	$136,800,000$

aplicaciones alcanzaban los límites de la programación existente, debido a que tenían que supervisar los diversos procesos simultáneos de manera confiable y en intervalos cortos de tiempo [véase 25].

Debido a la necesidad de aplicaciones de tiempo real que fueron capaces de ejecutar una variedad de tareas concurrentes y que respondieran rápidamente a respuestas no deterministas, se crearon diferentes lenguajes de programación con las siguientes características:

- Tienen un número fijo de procesos concurrentes que inician simultáneamente y permanecen mientras la computadora esté en funcionamiento. Un proceso puede acceder solamente solamente a sus propias variables, no hay variables en común para otros procesos.
- Permitir que los procesos ejecuten procedimientos en común.
- Permitir la sincronización de procesos por medio de sentencias no deterministas denominadas regiones críticas.

A raíz de las características mencionadas anteriormente, se presentó una deficiencia en cuanto al rendimiento y la tolerancia a fallas. En consecuencia, se optó por varias maneras de realizar la programación, lo que condujo a la programación paralela.

3.3. Programación Paralela

El paralelismo es el resultado de los esfuerzos continuos enfocados a lograr mejores rendimientos. Se han hecho intentos para disfrazar esta característica por medio de hardware o software.

En primer lugar, a nivel de software, el entorno de un proceso en ejecución de una computadora paralela es semejante al proceso en ejecución de un sistema con un sólo procesador. Una de las acciones más importantes de una aplicación paralela, es la comunicación con procesos remotos o localidades de memoria que consumen tiempo debido a que hay más de un proceso en ejecución por cada procesador de una máquina paralela.

En un ambiente de multiprogramación, el cambio de un proceso en ejecución a otro genera un gran costo debido al cambio del espacio de direcciones por cada proceso [veáse 16], por ello se han optado por el uso de hilos. Un proceso puede tener múltiples hilos, estos tienen su propia pila, contador de

programa y comparten el resto de la memoria con el resto de los hilos pertenecientes al proceso. En consecuencia, el espacio de direcciones no necesita cambiarse y el costo del cambio de un hilo en ejecución a otro es bajo.

3.3.1. Comunicación entre procesos

Los procesos e hilos se comunican de diferentes maneras, estas se restringen de acuerdo a la aquitectura paralela que se utilice [véase 9]. A continuación se muestran las 3 formas más comunes de comunicación:

- *Envío de Mensajes.* Un proceso envía el mensaje por medio de paquetes con una cabecera indicando el procesador receptor y el procesador destino; y los datos que enviará, dicho mensaje se canaliza a través de la red de interconexión. Una vez que el mensaje sale del origen, el proceso que envió el mensaje puede continuar su ejecución. A este tipo de envío se le denomina sin bloqueo.

- *Transferencias a través de memoria compartida.* En las arquitecturas de memoria compartida, la comunicación entre procesos consiste en asignar o tomar valores de ciertas localidades de memoria, a esta comunicación se le denomina directa.

- *Acceso directo a memoria.* Con las primeras arquitecturas de memoria distribuida, el procesador se interrumpía cada vez que se recibía una petición de la red. Esto ocasionaba una baja en el rendimiento de las aplicaciones. Por ello se desarrollaron las máquinas SMP, que cuentan con uno o más procesadores de aplicaciones y un procesador de mensajes para aligerar el tráfico de la red. De este modo un mensaje se trata como un acceso a memoria de otro procesador.

3.3.2. Estrategias para el desarrollo de aplicaciones

Las estrategias para el desarrollo de aplicaciones paralelas pueden ser de tres tipos diferentes. La descripción de las estrategias se presenta de acuerdo al nivel de abstracción que tiene el programador con respecto a la implementación de una aplicación paralela:

a) *Paralelización Automática.* Consiste en relevar al programador de las tareas de paralelización. Por ejemplo, un compilador puede aceptar un

código obsoleto (*dusty-deck*) y producir un código objeto paralelo y eficiente sin (o con muy poco) trabajo adicional por parte del programador.

b) *Portabilidad del codigo paralelo.* Consiste en el uso de bibliotecas paralelas. La idea básica es encapsular algún código paralelo que sea común a varias aplicaciones en una biblioteca paralela, para que pueda implementarse eficientemente y reutilizarse en otros códigos.

c) *Desarrollo de una aplicación paralela.* Consiste en escribir una aplicación paralela desde el principio, lo que otorga mayor grado de libertad al programador para que pueda escoger el lenguaje y el paradigma de programación. Sin embargo, su nivel de portabilidad es limitado.

3.3.3. Granularidad

La granularidad consiste en identificar los pedazos de código que son candidatos potenciales a paralelizar [véase 31]. Las aplicaciones paralelas emplean la granularidad de código con la finalidad de incrementar la eficiencia del procesador, de esta manera, el paralelismo se puede detectar en varios niveles:

- Granos muy finos (hardware pipeline).
- Granos finos (repartir datos o bloques de datos a los procesadores).
- Granos medios (nivel de control esencialmente en ciclos).
- Granos grandes (subrutinas y procedimientos).

De los cuatro niveles de paralelismo, los dos primeros se realizan por hardware o utilizando compiladores enfocados en la paralelización. Los programadores manejan los dos últimos niveles de paralelismo.

3.4. Diseño de Algoritmos Paralelos

En el diseño de algoritmos paralelos, se necesita de un diseño metodológico que permita al programador trabajar con aspectos independientes de la arquitectura y, en el último paso del diseño, enfocarse a los aspectos de la

arquitectura donde se implemente la aplicación [véase 31]. A continuación se muestran los cuatro pasos sugeridos para el proceso de diseño:

a) *Particionamiento.* Consiste en la descomposición de los datos y las funciones relacionadas con el problema en varios subproblemas.

b) *Comunicación.* Consiste en determinar el flujo de datos y la coordinación de los subproblemas que se crean en el particionamiento. La naturaleza del problema y el método de descomposición determinan el patrón de comunicación entre los subproblemas de la aplicación paralela.

c) *Aglomeración.* Consiste en evaluar los subproblemas y su patrón de comunicación en términos del rendimiento y su costo de implementación. Por ejemplo, se determina si los subproblemas se agrupen en subproblemas más grandes y de manera similar que las comunicaciones se manejen conjuntamente. Esto mejora el rendimiento y/o reduce los costos de desarrollo.

d) *Mapeo.* Consiste en asignar una subproblema a cada procesador para maximizar el uso de los recursos del sistema (tiempo de procesador) y minimizar los costos de comunicación. La relación puede realizarse en forma estática (tiempo de compilación) o dinámica (tiempo de ejecución) utilizando algún método de balanceo de carga.

3.4.1. Paradigmas de Programación Paralela

Los paradigmas son metodologías de alto nivel que ayudan en el diseño de algoritmos eficientes. En un algoritmo paralelo, varias subtareas se ejecutan simultáneamente y sus referencias de acceso a datos corresponden a un patrón de comunicación. A continuación se detallan los paradigmas más utilizados en la programación paralela [véase 28, 31].

1. *Redes de Procesos.* Este paradigma consta de dos entidades: un proceso maestro y múltiples procesos esclavos. El proceso maestro descompone el problema en varios subproblemas y los distribuye entre los procesos esclavos; además reune los resultados parciales en cierto orden para producir el resultado final. Los procesos esclavos ejecutan sus tareas en un ciclo, toman un mensaje del proceso maestro, procesan el mensaje y manda el resultado al proceso maestro.

2. *Un Programa con Múltiples Datos (SPMD)* Este paradigma es el más utilizado, cada proceso ejecuta la misma pieza de código con una porción diferente de los datos. Esto implica la división de los datos de la aplicación entre los procesadores disponibles. A este tipo de paralelismo se le conoce paralelismo geométrico, descomposición del dominio o paralelismo de datos.

3. *Línea de ensamble de datos (Pipeline de Datos)* Este paradigma utiliza granularidad fina. Con base en la descomposición funcional, las tareas del algoritmo que se ejecutan en forma simultánea se identifican y en cada procesador se ejecuta una parte del algoritmo. El pipeline es uno de los paradigmas más simples y más utilizados en la descomposición funcional. El pipeline consiste de varias etapas representadas, cada una de ellas por un proceso y ejecutan una función.

4. *Divide y vencerás* Este paradigma es bien conocido en el desarrollo de algoritmos secuenciales. El paralelismo requiere la división del problema original y combinar los resultados parciales de los subproblemas. De esta manera, los subproblemas se resuelven simultáneamente, mientras que para la partición del problema y la combinación de los resultados requieren de un patrón de comunicación entre procesos. Sin embargo debido a que los problemas son independientes, no se requiere una comunicación constante entre los procesos.

5. *Paralelismo Especulativo* Este paradigma se utiliza cuando es difícil obtener un algoritmo a través de los paradigmas antes mencionados. Algunos problemas dependen de datos complejos y se reduce la posibilidad de diseñar un algoritmo paralelo. En estos casos, una solución apropiada es resolver el problema en partes pequeñas al aplicar una ejecución especulativa u optimista para facilitar el paralelismo. Un uso común de este paradigma es en los casos donde se emplean diferentes algoritmos para resolver el mismo problema.

6. *Modelos Híbridos* Las fronteras entre los paradigmas suelen ser difusos y en algunos problemas existe la necesidad de mezclar elementos de varios paradigmas. Los métodos hibridos se utilizan a menudo en las aplicaciones masivamente paralelas, donde se necesita mezclar simultáneamente los datos y las tareas en partes diferentes de la aplicación.

3.5. Bibliotecas Paralelas

Las bibliotecas paralelas representan un modelo de cómputo paralelo que va desde un bajo nivel hasta un alto nivel de abstracción. Esta se define como una máquina abstracta que proporciona funciones a nivel de programación que requiere de una implementación a bajo nivel en cada una de las arquitecturas donde se utilice.

Esto se hace para separar el desarrollo del software de la ejecución en paralelo, lo que proporciona un cierto nivel de abstracción y estabilidad. El nivel de abstracción se logra debido a que la implementación de las funciones con base en los niveles subyacentes, simplifica la estructura del software y reduce la dificultad de su construcción.

3.5.1. MPI (Biblioteca de envío de mensajes)

El envío de mensajes es utilizado ampliamente en ciertas clases de máquinas paralelas, especialmente en las de memoria distribuida.Aunque con modificaciones se presenta el concepto básico de comunicación de procesos por envío de mensajes. En los últimos diez años se ha progresado significativamente en la creación de aplicaciones bajo este paradigma. En el diseño, MPI[1] cuenta con el uso de características eficientes de sistemas de envío de mensajes seleccionados e introducidos como parte del estándar. En la estandarización de MPI, participan alrededor de 60 personas de 40 organizaciones de Estados Unidos y Europa.

Las principales ventajas de establecer un estándar es lograr portabilidad y facilidad de utilización. La definición de un estándar de envío de mensajes, consiste en proporcionar un conjunto de rutinas precisas para implementarse eficientemente o en algunos casos proporcionar soporte en hardware al incrementarse la escalabilidad.

3.5.2. MPI-IO

El patrón de acceso para E/S de varias aplicaciones paralelas consiste en acceder a un gran número de piezas pequeñas de datos no contiguas. Cuando una aplicación necesita hacer peticiones pequeñas de E/S, el rendimiento disminuye drásticamente. Para evitar este problema, la biblioteca MPI-IO

[1]Message Passing Interface

permite a los usuarios acceder a los datos no contiguos en una sola llamada de E/S, esto es diferente a las operaciones utilizadas en UNIX.

Para resolver las limitaciones del rendimiento y portabilidad de las interfaces de E/S en paralelo, en el foro de MPI (hecho por los vendedores de computadoras, investigadores y científicos) acordaron una interfaz para las operaciones de E/S en paralelo como parte del estándar MPI-2. La interfaz se conoce como MPI-IO [veáse 39], es una interfaz diseñada con varias características enfocadas al rendimiento y portabilidad de E/S en paralelo.

La manera de hacer portable a MPI-IO, es implementar las funciones de MPI-IO con base en las funciones básicas de E/S en UNIX. Como las funciones de E/S en UNIX son portables, MPI-IO es portable en varias máquinas con sus respectivos sistemas de archivos. Sin embargo, existen limitaciones de funcionalidad y rendimiento, como se muestra a continuación:

- Las funciones básicas de E/S no son suficientes para implementar MPI-IO en cualquier sistema de archivos.
- Aunque las funciones básicas de E/S están en varios sistemas de archivos, es común que existan funciones "recomendadas" en términos de rendimiento para cada sistema de archivos.

Para superar este problema, MPI-IO utiliza ADIO [véase a 36] (Interfaz Abstracta de Dispositivo), diseñada para la implementación de API's de E/S paralela en múltiples sistemas de archivos. Esta interfaz se desarrolló antes de que MPI-IO se convirtiera en un estándar, se utilizó y experimentó con varias API's de E/S en paralelo hasta llegar al estándar MPI-IO.

La implementación de MPI-IO es conocida también como ROMIO y trabaja en los sistemas de archivos como PIOFS de IBM, PFS de Intel, HFS, XFS de SGI, SFS de NEC, NFS, UFS y PVFS. ROMIO se diseñó para utilizarse en cualquier implementación MPI, por ejemplo en las implementaciones MPICH, HPMPI, LAM-MPICH y SGIMPI.

En el diseño de MPI-IO, en lugar de la definición de los métodos de acceso para expresar los patrones comunes para acceder a un archivo compartido en transmisión (broadcast), reducción (reduce), distribución (scatter) y unión (gather), se escoge otra alternativa, donde la partición de datos se expresa por medio de tipos de datos derivados.

Como ejemplo, *filetype* es un tipo de representación de archivo (Figura 3.1) es la base para distribuir un archivo entre los procesos y define una

guía para acceder al archivo, este puede estar construido de múltiples instancias de un tipo de dato elemental(*etype*). También una vista de archivo (*view*) define el conjunto actual de datos visibles y accesibles desde un archivo abierto como un conjunto ordenado de *etypes* (Figura 3.2). Cada proceso tiene su propia vista del archivo, definida por 3 valores: un desplazamiento, un *etype* y un tipo de representación de archivo. De esta forma, un grupo de procesos puede utilizar vistas complementarias para alcanzar una distribución global de los datos como un patrón de distribución/ensamble.

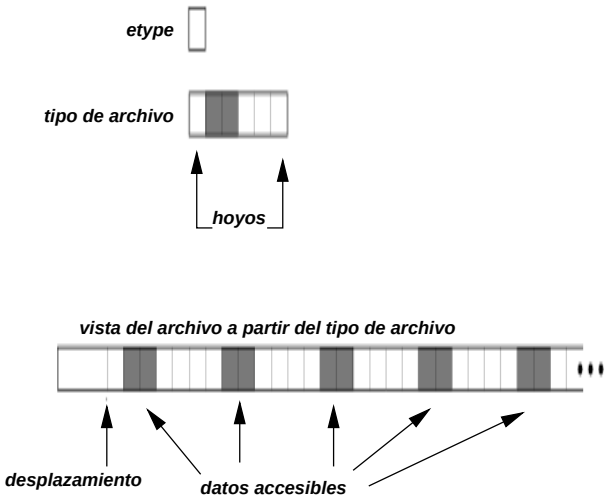

Figura 3.1: Tipos de archivo y etype

3.5.3. Biblioteca de entrada/salida Panda

Varios proyectos de investigación intentan facilitar el problema para los programadores de aplicaciones que requieren operaciones de E/S en paralelo por medio de sistemas de archivos paralelos o bibliotecas de E/S paralela.

La biblioteca Panda de E/S en paralelo, se desarrolló en la Universidad de Illinois [véase 19], está escrita en lenguaje C++ para un nivel usuario y proporciona una interfaz simple por medio de un arreglo paralelo multidimensional de E/S.

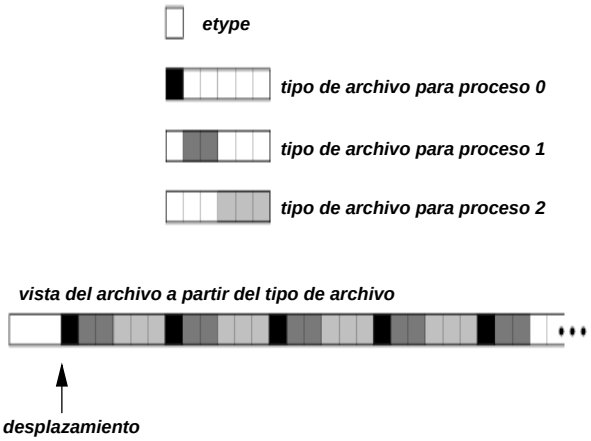

Figura 3.2: Vistas de archivo

Los arreglos multidimensionales son distribuidos en múltiples procesadores, donde cada procesador tiene su propio conjunto de arreglos pequeños e independientes. Para garantizar la portabilidad, Panda utiliza a MPI para comunicar los procesos y las llamadas estándar POSIX que se utilizan en la plataforma subyacente del sistema de archivos. Panda puede escribir los datos en un formato nativo binario o en un formato de datos jerárquico (HDF), que proporciona portabilidad en el archivo.

La biblioteca Panda divide los procesos participantes en dos grupos (Figura 3.3): los procesos para cálculo y los procesos para operaciones de E/S. Los procesos de cálculo están situados en los procesadores dedicados para las aplicaciones y los procesos para operaciones de E/S se sitúan en los procesadores dedicados a E/S en los sistemas de archivos.

Cuando se realizan las operaciones de E/S, cada procesador de cálculo distribuye una petición de E/S hacia su cliente local Panda de forma colectiva. Un cliente seleccionado como maestro manda una descripción de alto nivel relacionada con la petición de E/S denominada esquema, a un servidor seleccionado como maestro para el proceso de E/S.

El proceso de E/S se realiza como se muestra en la Figura 3.4: una operación en un servidor de E/S cero (también el servidor de E/S uno, puede realizar procesos similares en paralelo) recibe el mensaje con la información

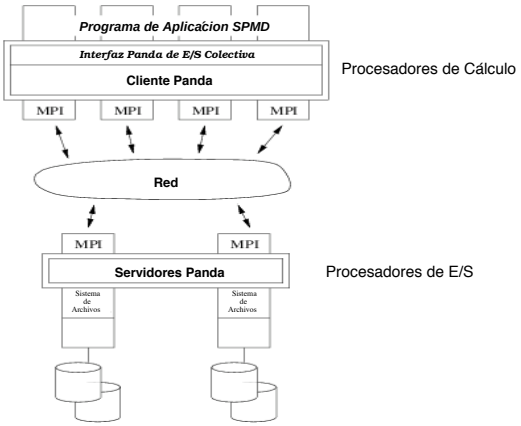

Figura 3.3: Estructura de la Biblioteca Panda

del arreglo de los procesos de cálculo, y en el servidor maestro determina la base y el tamaño del pedazo que se va a escribir. Entonces, el servidor determina qué proceso de cálculo tiene datos disponibles para el pedazo de E/S correspondiente. De acuerdo a la distribución del arreglo entre los procesos de cálculo, el servidor envía un mensaje solicitando datos a los procesos de cálculo y estos responden enviando los datos requeridos al servidor de E/S. Posterior a la recepción de datos de los procesos, el servidor reorganiza los datos por medio de los tipos de datos derivados de MPI para crear pedazos de E/S de datos contiguos y escribir los datos al disco local.

3.5.4. Sistema de Archivos Paralelo Galley

Mientras que la velocidad de los componentes de las computadoras paralelas aumentan constantemente, el subsistema de E/S no sigue esta tendencia. Esta limitación en el hardware retrasa el desarrollo de los sistemas de archivos multiprocesador. Desde el punto de vista del software, la información es limitada de cómo las aplicaciones utilizarán este tipo de sistemas de archivos en multiprocesador y cómo los programadores pueden utilizar a futuro estos sistemas de archivos. Como resultado de examinar cómo las aplicaciones científicas paralelas deben utilizar el sistema de archivos, se crea el sistema de archivos paralelo Galley [veáse 23].

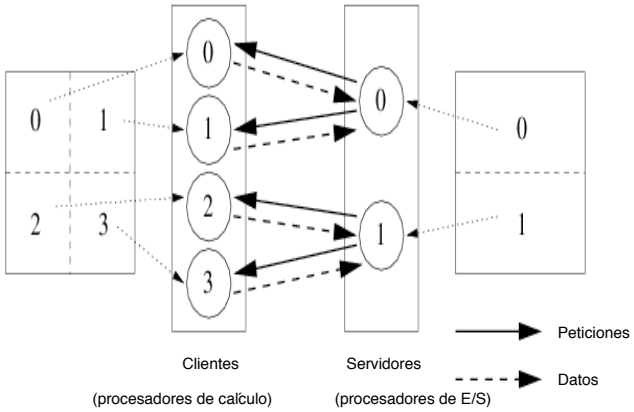

Figura 3.4: Operación de la Biblioteca Panda

Las aplicaciones paralelas en Galley, tienen la capacidad de indicar explícitamente a qué disco desean acceder en cada petición. Para lograr esta función, los archivos se componen de uno o más subarchivos que son direccionados por medio de la aplicación. Esta puede escoger cuántos subarchivos tendrá un archivo cuando este es creado.

Cada subarchivo en Galley se estructura como una colección de una o más ramificaciones independientes (Figura 3.5). Cada ramificación es una secuencia lineal de bytes direccionables, semejante al de un archivo en Unix. En caso contrario al número fijo de subarchivos de un archivo, el número de ramificaciones de un subarchivo no es fijo. De esta manera, las bibliotecas y las aplicaciones pueden agregar ramificaciones o removerlas en cualquier instante.

El sistema de archivos paralelo Galley, se estructura como un conjunto de clientes y servidores (Figura 3.6). Este modelo está basado en una típica arquitectura multiprocesador que dedica algunos procesadores al cálculo y el resto a operaciones de E/S, los procesos de cálculo funcionan como clientes y los procesadores de E/S actúan como servidores.

Un cliente en Galley es una aplicación ligada a la biblioteca de Galley que corre en un procesador de cálculo. De esta manera, por medio de la biblioteca, el cliente traslada o recibe los datos de los procesos de E/S.

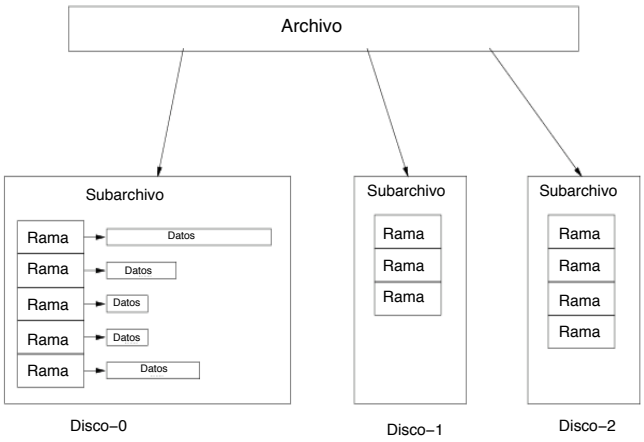

Figura 3.5: Estructura de un archivo en la interfaz Galley

Los servidores de E/S en Galley están compuestos de varias unidades funcionales, donde cada una está implementada como un hilo de ejecución. De este modo, cada proceso de E/S es un hilo designado para atender las peticiones de E/S. Por ello, se logra facilitar el servicio de peticiones de E/S para atender a varios clientes simultáneamente.

Cada proceso de E/S tiene un cache manejado por el administrador de cache, este decide qué bloques deben ser guardados en el cache y localiza los bloques requeridos para los hilos asignados a cada proceso de cálculo. Además se cuenta con un administrador de discos, que mantiene una lista de bloques que el administrador de cache requiere para leer o escribir, cuando las peticiones nuevas llegan al administrador de cache, estas se colocan en la lista de acuerdo a la manera de administrar la cola de peticiones.

La interfaz estándar de Unix proporciona sólo las funciones primitivas para acceder a los datos, estas funciones no son suficientes para reunir las necesidades de las aplicaciones paralelas, porque las aplicaciones paralelas científicas frecuentemente realizan pequeñas operaciones a un archivo con patrones de acceso en serie (strided).

Estos patrones de acceso permiten al sistema de archivos combinar varias peticiones pequeñas en una sola petición, lo que mejora el rendimiento en dos aspectos. En primer lugar, reduce el número de peticiones que bajan los

costos de latencia, en particular con aquellas aplicaciones que realizan cientos o millones de operaciones pequeñas. En segundo lugar, la información proporcionada por el sistema de archivos, ayuda a realizar una implementación más eficiente de la cola de peticiones para mantener el control de los accesos a disco y utilizar los discos cache eficientemente.

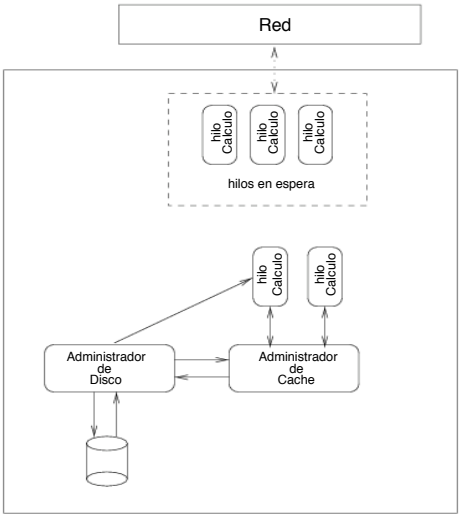

Figura 3.6: Estructura interna de un proceso de entrada/salida Galley

3.5.5. Parallel Virtual File System

Como se mencionó en el capítulo dos, los programas escritos para utilizar las operaciones de E/S en Unix pueden trabajar en PVFS sin realizar cambios [veáse 32]. Esta transparencia en el acceso a los datos se debe tanto al movimientos de datos a través del kernel con la biblioteca de PVFS para comunicarse con el demomio cliente pvfsd (Figura 3.7).

La biblioteca PVFS tiene la particularidad de escoger diversas distribuciones físicas o lógicas para un archivo. La distribución física permite escoger al programador el número de nodos de E/S donde se almacenan los archivos de datos y el tamaño en bytes que se almacenará en cada uno de los nodos (tamaño de la distribución).

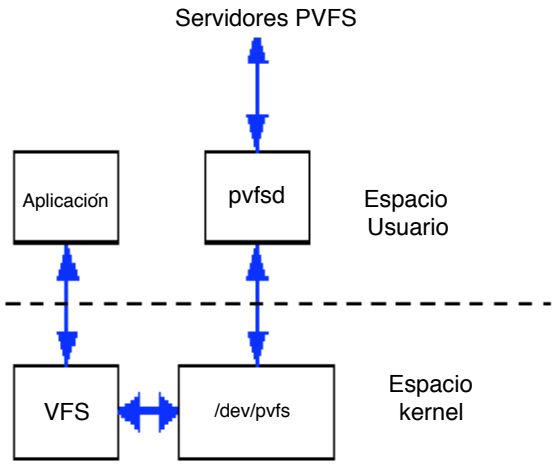

Figura 3.7: Comunicacion de PVFS con el kernel

La biblioteca PVFS permite al programador de aplicaciones describir las regiones de interés en un archivo y posteriormente acceder a estas regiones de una manera eficiente. Este acceso es más eficiente porque el sistema PVFS permite que distintas regiones puedan ser descritas como una partición lógica para acceder a estas en una sola unidad. Otra alternativa es realizar múltiples accesos, lo que disminuye el rendimiento debido al número de operaciones y al movimiento reducido de datos por operación.

La partición o distribución lógica facilita la programación paralela al simplificar el acceso a los datos de un conjunto compartido por varias tareas correspondientes a una aplicación. Cada tarea puede tener su propia partición lógica y todas las operaciones de E/S será únicamente para dicha partición lógica del conjunto de datos por cada tarea.

Capítulo 4

BIBLIOTECA PARALELA REDUCIDA: DISEÑO E IMPLEMENTACIÓN

En el capítulo anterior se mencionaron algunos ejemplos de bibliotecas de E/S paralela que son más utilizadas para diferentes sistemas de archivos paralelos y remotos. En este capítulo se enfoca en presentar una propuesta. El diseño y la implementación que aqui se describen corresponden a una biblioteca de E/S reducida. Existe una carencia de bibliotecas de E/S paralela para los sistemas de archivos distribuidos, por ello se diseñó e implementó una biblioteca de E/S paralela para el sistema de archivos distribuido AFS.

4.1. Diseño

4.1.1. Interfaz Abstracta de Dispositivo (ADIO)

Para el diseño de la biblioteca de E/S paralela se utilizó el concepto de Interfaz Abstracta de Dispositivo (ADIO) que es una estrategia para la implementación de bibliotecas de E/S paralela para cualquier sistema de archivos [veáse 29].

La implementación de ADIO consiste de un conjunto de funciones diseñadas con base en el estudio de varias bibliotecas de E/S paralela y sistemas de archivos con el objetivo de obtener una mayor eficiencia en las operaciones de E/S, y en cuanto a la portabilidad se utiliza a MPI que es un estándar

para la mayoría de las máquinas paralelas, el esquema de ADIO se muestra en la Figura 4.1.

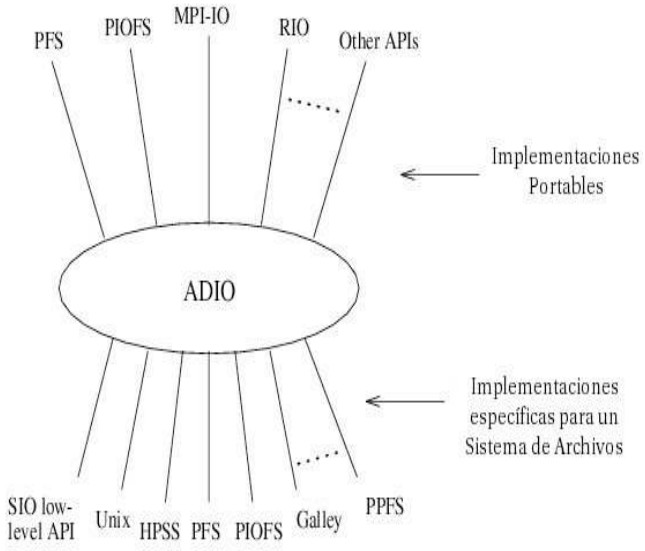

Figura 4.1: Esquema de ADIO

En la parte superior del esquema de ADIO puede implementarse cualquier biblioteca paralela de E/S, en contraste la capa inferior ADIO (ADIOI) debe implementarse para cada sistema de archivos.

En nuestro caso de estudio, se implementó una biblioteca paralela en ADIOI para el sistema de archivos distribuido AFS que se describe a continuación.

4.1.2. Andrew File System (AFS)

El sistema de archivos AFS [véase 17], considera una estructura funcional con base en un esquema de comunicación cliente-servidor. Por ejemplo, un grupo de máquinas servidor proporciona el espacio en disco para un sistema de almacenamiento central controlado por alguna organización. Las peticiones de acceso a los archivos y directorios son realizados desde máquinas que ejecutan el programa cliente AFS. Si el cliente está autorizado por la organización para realizar una operación, entonces el servidor procede a ejecutarla.

El espacio en disco utilizado por AFS no almacena directamente los archivos al sistema de archivos local de la máquina servidor. En contraste, los contenidos de los subdirectorios son colocados en contenedores denominados volúmenes.

Volúmenes

Un volumen [veáse 10] varía en tamaño de acuerdo a las operaciones entre los archivos: escritura, lectura-escritura y borrado. Cada uno de estos tiene una quota o nivel máximo de almacenamiento permitido, un sistema de archivos local puede tener más de un volumen. Los volúmenes tienen dos identificadores, el primero es un número de 32 bits denominado *volume ID* y el segundo es una cadena de caracteres denominada *volume name*.

Los subdirectorios contenidos en volúmenes individuales son unidos al sistema de archivos distribuido, y en las máquinas clientes se forma un árbol de subdirectorios único. A estos puntos de unión se les denomina puntos de montaje.

Los puntos de montaje AFS difieren de los puntos de montaje NFS. En NFS se monta de forma dinámica la partición remota, pero no se garantiza que el espacio de nombres sea uniforme en varias máquinas.

El volumen raíz de AFS contiene el espacio global de nombres. En cada máquina cliente perteneciente a una organización se monta el volumen raíz en el directorio */afs*. De esta manera, desde ese punto de montaje se accede al árbol de subdirectorios global. Los volúmenes son la unidad administrativa de los datos contenidos en AFS, proporcionan las bases para la replicación, reacomodo y respaldo de los datos.

Seguridad

Se utiliza el sistema de autenticación [veáse 37] Kerberos para identificar a los usuarios que acceden al árbol de subdirectorios global. Este sistema proporciona una autenticación mutua para asegurar que los servidores AFS son los que interactúan con los clientes, y que los clientes AFS no sean impostores.

La información de la autenticación se utiliza en objetos denominados *tickets*. Los clientes registran sus password en el sistema de autenticación y se crean los *tickets*. Un *ticket* contiene la versión cifrada del nombre del usuario e información adicional. El uso de los *tickets* se presenta cuando se

requiere de uno para realizar operaciones en los archivos. En estos casos se descifra el *ticket*, se conoce el sistema de autenticación que creo el *ticket* e identifica al usuario que desea realizar las operaciones en el archivo.

Para determinar si un usuario puede acceder a un conjunto de archivos, AFS utiliza sus listas de control de acceso (ACL) donde se especifica el conjunto de usuarios que tienen permitido el acceso a los directorios y a modificar los archivos, estas listas se organizan en pares *(principal,derechos)* donde *principal* puede ser el nombre de usuario o un grupo de usuarios y *derechos* son los permisos que tienen sobre los archivos que se mencionan a continuación:

- **Read(r)**: permiso de lectura.
- **Lookup(l)**: permiso de acceso a directorios padre.
- **Write(w)**: permiso de crear, escribir y sobrescribir archivos.
- **Insert(i)**: permiso de insertar archivos en un directorio, pero no sobreescribir los existentes.
- **Delete(d)**: permiso para borrar archivos en un directorio.
- **Lock(k)**: permiso para crear bloqueos de advertencia en un directorio.
- **Administer(a)**: permiso para cambiar las ACL de un directorio.

Célula

La célula es el conjunto de servidores y clientes que son controlados por una organización independiente. Los administradores de las células toman las decisiones acerca de la configuración de los servidores, los esquemas de respaldo y las estrategias de replicación independiente de otras células.

Cada máquina cliente pertenece solo a una célula [veáse 11], en consecuencia esta información determina los recursos y servicios a los que se tiene acceso. Mediante las células se permite la colaboración de varias organizaciones para conjuntar un espacio de nombres individuales. Por convención un nombre de archivo inicia con */afs* como parte del espacio de nombres global y se puede acceder por cualquier máquina AFS.

Desde un punto de montaje, se puede contener la información de las celulas, permitiendo que los volúmenes de células vecinas sean montados en

el sistema de archivos global. En consecuencia, el directorio */afs* contiene un punto de montaje para el volumen *root.cell* que representa una célula de la comunidad AFS. Por ejemplo, la raíz del árbol de subdirectorios controlado por la célula *xyz* se representa por el directorio */afs/xyz*.

Servidor de Archivos

El servidor de archivos AFS se compone de un conjunto de procesos que porporcionan diferentes servicios que se mencionan a continuación:

File Server : El servidor de archivos [veáse 12] proporciona espacio en disco para un conjunto de archivos junto con los volúmenes, y permite que los archivos sea accesibles para los usuarios autorizados en las máquinas cliente.

Volume Location Server : El servidor de localización de volúmenes [veáse 13] mantiene y exporta la base de datos de localización de volúmenes (VLDB). Esta base almacena las direcciones del servidor o del conjunto de servidores donde un volumen o instancias del mismo residen. Otra de sus funciones es responder las peticiones acerca de la localización de volumen, ID del volumen y creación, borrado y modificación de los registros en la VLDB. La VLDB puede repartirse en dos o más máquinas servidor para su disponibilidad. De esta manera el *Volume Location Server* se ejecuta en cada máquina servidor con una copia de la VLDB.

Volume Server : El servidor de volumenes [veáse 13] realiza tareas administrativas y verifica el rendimiento del conjunto de volúmenes AFS que residen en la máquina donde se ejecuta el servidor. Otra de sus funciones son:

- Creación y borrado de volúmenes.
- Renombrar volúmenes.
- Descarga y restauración de volúmenes.
- Alterar la lista de replicación de sitios para los volúmenes de solo lectura.
- Creación y propagación de un volumen de solo lectura.
- Creación y actualización de volúmenes de respaldo.

- Listado de todos los volúmenes en una partición y examinar el estado del volumen.

Authenticacion Server : El servidor de autenticación mantiene y exporta la base de datos de autenticación (ADB) [veáse 37]. Esta base registra los passwords cifrados de los usuarios de la célula. Implementa el protocolo de autenticación mutua y crea los *tickets* de identificación para las sesiones de usuario. La ADB puede repartirse en dos o más máquinas servidor para su disponibilidad. De esta manera el *Authentication Server* se ejecuta en cada máquina servidor con una copia de ADB.

Protection Server : El servidor de protección mantiene y exporta la base de datos de protección. Esta base registra los nombres de usuario y grupos junto con su identificador AFS. Entre sus funciones esta la manipulación de los registros de usuario y grupos.

BOS Server : El servidor BOS [veáse 11] es una herramienta administrativa que corre en cada máquina servidor de archivo perteneciente a una célula. Este servidor monitorea el estado de los procesos de los servicios AFS. Entre otras funciones inicia el conjunto de los procesos servidor AFS despues del inicio del sistema operativo, responde a peticiones sobre el estado y reinicia cuando algún proceso falla. Acepta comandos para iniciar, suspender, resumir el estado del proceso e instalar nuevos procesos.

Update Server/Client : Los procesos de actualización de servidor y cliente se utilizan para distribuir los archivos importantes para un servidor y sus binarios. Por ejemplo, cuando se distribuye un servidor de archivos para un conjunto de máquinas en una célula. Entonces una de las máquinas se crea un punto de distribución y se corre el proceso *Update Server*. Cada uno de los servidores corre una instancia del proceso de actualización del cliente, que periódicamente activa al proceso de actualización de servidores. De esta manera el servidor de archivos nuevo es detectado y se actualiza. En consecuencia, los nuevos servidores solo necesitan instalarse y la distribución para detectarlos se realiza automáticamente.

Cliente

Una parte de AFS que se ejecuta en las máquinas cliente se denomina *Cache Manager*. Este código se ejecuta en modo kernel y se utiliza como puente de comunicación entre el usuario y los servidores de archivo. La función principal del *Cache Manager* [veáse 12] es crear la ilusión de que el árbol de subdirectorios globa está en la máquina cliente. Como lo dice su nombre, el *cache manager* mantiene un cache de archivos referenciados del sistema de archivos global en el disco de la máquina cliente. Todas las operaciones realizadas por los programas de aplicación en los archivos son realizados en las imágenes almacenadas en el cache.

Las referencias hacia las operaciones equivalentes en AFS se realizan por *VFS* y las interfaces *vnode*. El *cache manager* almacena y toma los archivos desde y hacia el servidor de archivos AFS para realizar las operaciones en los archivos.

Tiene otras funciones como almacenar la información de autenticación del usuario, almacenar sus *tickets* y utilizarlos durante las interacciones con el servidor de archivos.

4.1.3. Integración de ADIO con AFS

En un contexto de un cluster, cada nodo se convierte en una máquina cliente AFS. Cada uno de los clientes requiere mantener el cache de los archivos accedidos de los servidores AFS. En un ambiente serial, se realizan las operaciones de lectura/escritura con las copias de los pedazos de los archivos situadas en los caches de archivos de cada nodo. En contraste, en un ambiente paralelo no solo se requiere mantener la consistencia de cache con los servidores AFS, tambien es requerida la sincronización de los caches de archivos en todos los nodos cuando varios procesos acceden a un solo archivo y realizan sus operaciones de lectura/ escritura.

Callbacks

Para sincronizar las diferentes versiones de las porciones de un archivo en AFS, se utilizan los *callbacks* [veáse 12]. Cada cliente AFS debe verificar que las copias de los pedazos de un archivo en el cache, deben coincidir con las secciones del archivo almacenado en la máquina servidor, antes de que un proceso realice las operaciones en estos pedazos.

Para verificar a los pedazos de archivos, se emplea la noción de un *callback*, que consiste en un algoritmo de consistencia de cache. Cuando una máquina servidor entrega uno o más pedazos de un archivo a un cliente, tambien incluye un *callback*, es decir, una "promesa" de que el cliente notificara al *file server* si se realizaron modificaciones a los datos en un archivo.

De esta manera, mientras el cliente esté utilizando el archivo, tiene el *callback* del archivo y sincroniza la version almacenada en el *file server*, esto permite que varios procesos realizen operaciones en el mismo archivo sin interactuar con el *file server*. Antes de que un *file server* almacene la versión más reciente de un archivo en disco, primero termina con los *callbacks* de este. Existen tres tipos de *callbacks*:

EXCLUSIVE : Indica que un solo cache manager tiene el control de las versiones de un archivo.

SHARED : Indica el estado de la versión de los pedazos de un archivo respecto a un *cache manager*. Si todos los pedazos de archivos son equivalentes a la versión más actual se garantiza la consistencia del cache. Este *callback* permite que varios *cache managers* tengan *callbaks* del mismo archivo junto con los pedazos del archivo y realizar una sincronización de versiones.

DROPPED : Indica que el callback fue cancelado por el *file server*. En consecuencia, el cache manager marca el estado del archivo como desconocido, de esta manera cuando un proceso tenga que acceder al archivo, el *cache manager* tiene que tomar el pedazo de archivo desde el *file server*.

Como se mencionó en la sección anterior, los procesos realizan sus operaciones de *lectura/escritura* en las copias de los pedazos de archivos almacenados en el cache de archivos de cada nodo del cluster. Esto representa una baja en el rendimiento para una operación de escritura en paralelo, porque hay un retardo en el tiempo al sincronizar las versiones de los *cache managers*, y en adición un tiempo de retardo debido a las pérdidas de datos por la incoherencia de caches.

Para disminuir la baja de rendimiento, se aprovecha el uso de los *callbacks* para establecer la coherencia entre varios *cache managers*. Al escribir o reescribir pedazos de archivos almacenados en el cache de archivos no se notifica al *file server* de dicha modificación, para que el *cache manager* le notifique

al *file server* de una modificación, el proceso de la aplicación debe de llamar a las funciones *fsync()* o *fclose()*.

Data sieving

En la capa ADIOI, para reducir el retardo de las operaciones de *lectura/escritura*, es crucial realizar pocas peticiones al sistema de archivos. Una de las técnicas implementadas en ADIOI es el *data sieving* [veáse a 30]. El *data sieving* asume que el usuario realiza una petición de lectura para varias piezas de datos contiguos. En vez de leer una pieza no contigua de forma separada, se lee un gran pedazo de datos desde el byte inicial hasta el byte final y almacenarlo en un buffer temporal. De esta forma, del buffer temporal se extraen los pedazos requeridos y se colocan en el buffer del usuario en forma contigua Figura 4.2.

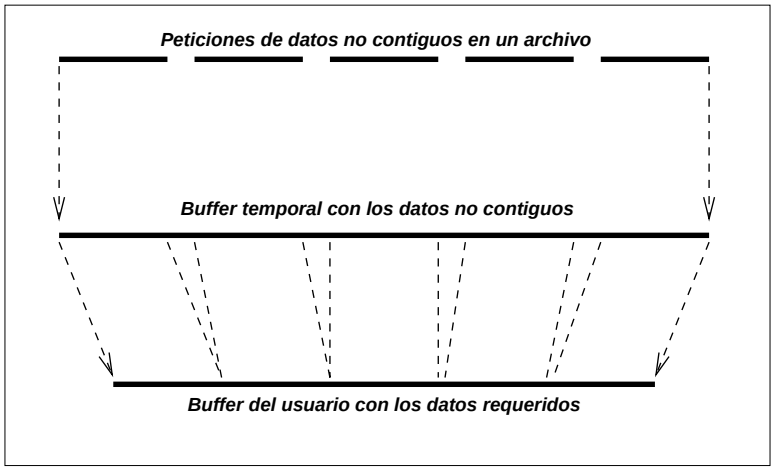

Figura 4.2: Esquema de DataSieving

Acceso a Bloques fuera de orden en dos Fases

La operación en paralelo de la *lectura/escritura* se realiza a través de funciones colectivas. En el caso de un acceso individual se emplea el *data sieving*. En contraste, cuando son dos o más procesos se emplea el concepto de acceso a arreglos de bloques fuera de orden en dos fases. Este método

de acceso en su primera fase, consiste en realizar varios *data sieving* por proceso de los datos requeridos en el archivo de acuerdo al dominio de archivo asignado. En la segunda fase se redistribuyen los datos de acuerdo a los datos requeridos por cada proceso. Un dominio de archivo es la porción de un archivo asignado a cada proceso Figura 4.3.

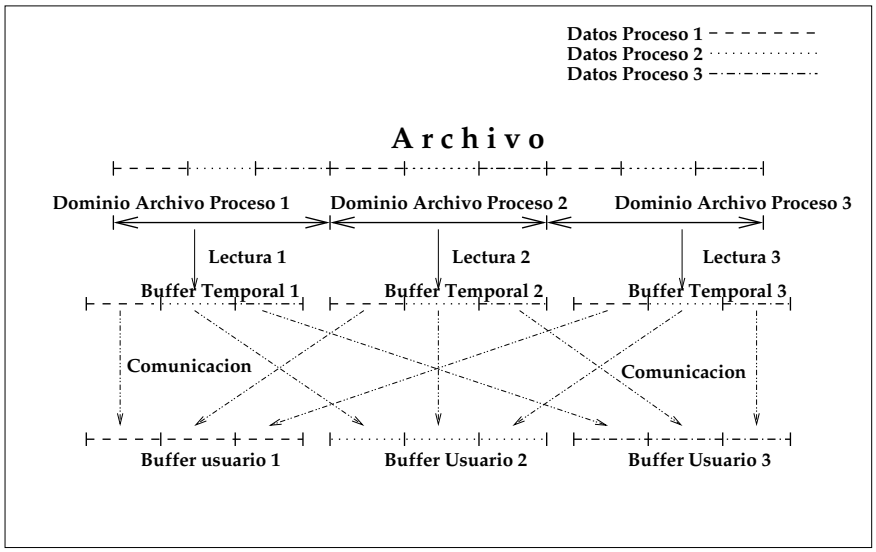

Figura 4.3: Esquema de Acceso Fuera de Orden en dos Fases

Como se menciona anteriormente cada proceso trabaja con las copias de los pedazos de archivo y estas se almacenan en el *cache manager*. En una operación de lectura colectiva no se presentan retardos en cuanto a notificaciones hacia el *file server* porque no hay modificaciones en los pedazos de archivos. En contraste, para una operación de escritura colectiva, se presenta un retardo debido a que al modificarse la copia almacenada en el *cache manager* no se notifica al *file server* hasta que el *cache manager* despues de un intervalo de tiempo, manda automáticamente el *callback* correspondiente.

En caso de que uno o más procesos requieran los datos modificados, puede haber un retardo por la pérdida de datos actualizados y del retraso de la notificación del *cache manager* hacia el *file server* cuando se presentan modificaciones en los datos. Debido a esto, en este contexto se utiliza la función *fsync()* para cada escritura realizada en la capa ADIOI, de esta manera se

aumenta el rendimiento de la escritura y se notifica a tiempo al *file server* de la modificación de los datos. A continuación se describe la implementación del diseño.

4.2. Implementación

En la capa ADIOI, se tiene cada una de las implementaciones para cada sistema de archivos, en nuestro caso de estudio, para la implementacion de la biblioteca paralela para el sistema de archivos AFS se procede a crear una estructura de apuntadores a funciones con base a una estructura genérica, que se muestra a continuación:

```
struct ADIOI_Fns_struct {
/* abrir un archivo */
    void (*ADIOI_xxx_Open) (ADIO_File fd, int *error_code);
/* lectura de datos contiguos */
    void (*ADIOI_xxx_ReadContig) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Status *status, int *error_code);
/* Escritura de datos contiguos */
    void (*ADIOI_xxx_WriteContig) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Status *status, int *error_code);
/* Lectura colectiva de datos no contiguos */
    void (*ADIOI_xxx_ReadStridedColl) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Status *status, int *error_code);
/*Escritura colectiva de datos no contiguos */
    void (*ADIOI_xxx_WriteStridedColl) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Status *status, int *error_code);
/* posicion de apuntador de un archivo individual */
    ADIO_Offset (*ADIOI_xxx_SeekIndividual) (ADIO_File fd,
    ADIO_Offset offset, int whence, int *error_code);
/* control de bloqueo sobre un archivo */
    void (*ADIOI_xxx_Fcntl) (ADIO_File fd, int flag,
    ADIO_Fcntl_t *fcntl_struct, int *error_code);
/* informacion acerca del archivo */
    void (*ADIOI_xxx_SetInfo) (ADIO_File fd, MPI_Info users_info,
```

```
    int *error_code);
/* Lectura de datos no contiguos */
    void (*ADIOI_xxx_ReadStrided) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Status *status, int *error_code);
/* Escritura de datos no contiguos */
    void (*ADIOI_xxx_WriteStrided) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Status *status, int *error_code);
/* Cerrar un archivo */
    void (*ADIOI_xxx_Close) (ADIO_File fd, int *error_code);
/* Lectura de datos contiguos independiente */
    void (*ADIOI_xxx_IreadContig) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Request *request, int *error_code);
/* Escritura de datos contiguos independiente */
    void (*ADIOI_xxx_IwriteContig) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Request *request, int *error_code);
/* Confirmacion de termino de lectura independiente */
    int (*ADIOI_xxx_ReadDone) (ADIO_Request *request,
    ADIO_Status *status, int *error_code);
/* Confirmacion de termino de escritura independiente */
    int (*ADIOI_xxx_WriteDone) (ADIO_Request *request,
    ADIO_Status *status, int *error_code);
/*  Lectura completa de datos */
    void (*ADIOI_xxx_ReadComplete) (ADIO_Request *request,
    ADIO_Status *status, int *error_code);
/*  Escritura completa de datos */
    void (*ADIOI_xxx_WriteComplete) (ADIO_Request *request,
    ADIO_Status *status, int *error_code);
/*  Lectura de datos no contiguos independiente */
    void (*ADIOI_xxx_IreadStrided) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Request *request, int *error_code);
/* Escritura de datos no contiguos independiente */
    void (*ADIOI_xxx_IwriteStrided) (ADIO_File fd, void *buf,
    int count, MPI_Datatype datatype, int file_ptr_type,
    ADIO_Offset offset, ADIO_Request *request, int *error_code);
/* Flujo de datos hacia un archivo */
```

```
    void (*ADIOI_xxx_Flush) (ADIO_File fd, int *error_code);
/* Truncar tamaño de un archivo */
    void (*ADIOI_xxx_Resize) (ADIO_File fd, ADIO_Offset size,
    int *error_code);
/* Borrar un archivo */
    void (*ADIOI_xxx_Delete) (char *filename, int *error_code);
    };
```

La estructura ADIOI_Fns_struct contiene los apuntadores hacia las funciones principales de ADIOI que a su vez son utilizadas por ADIO, y MPI-IO utiliza las funciones de ADIO para su implementación. En el caso de AFS, se tiene que crear esta estructura de funciones:

```
struct ADIOI_Fns_struct ADIO_AFS_operations = {
ADIOI_AFS_Open, /* Open **/
ADIOI_AFS_ReadContig, /* ReadContig **/
ADIOI_AFS_WriteContig, /* WriteContig **/
ADIOI_AFS_ReadStridedColl, /* ReadStridedColl **/
ADIOI_AFS_WriteStridedColl, /* WriteStridedColl **/
ADIOI_AFS_SeekIndividual, /* SeekIndividual **/
ADIOI_AFS_Fcntl, /* Fcntl **/
ADIOI_AFS_SetInfo, /* SetInfo **/
ADIOI_AFS_ReadStrided, /* ReadStrided **/
ADIOI_AFS_WriteStrided, /* WriteStrided **/
ADIOI_AFS_Close, /* Close **/
ADIOI_AFS_IreadContig, /* IreadContig **/
ADIOI_AFS_IwriteContig, /* IwriteContig **/
ADIOI_AFS_ReadDone, /* ReadDone **/
ADIOI_AFS_WriteDone, /* WriteDone **/
ADIOI_AFS_ReadComplete, /* ReadComplete **/
ADIOI_AFS_WriteComplete, /* WriteComplete **/
ADIOI_AFS_IreadStrided, /* IreadStrided **/
ADIOI_AFS_IwriteStrided, /* IwriteStrided **/
ADIOI_AFS_Flush, /* Flush **/
ADIOI_AFS_Resize, /* Resize **/
ADIOI_AFS_Delete, /* Delete **/
};
```

A partir de la declaración de la estructura de las funciones para el sistema de archivos AFS, se procede a implementar cada una de las funciones que se describen en el Apendice A.

Cuando se inicia una operación en paralelo con un archivo, MPI-IO le pide a ADIO que determine en que sistema de archivos se encuentra el archivo, para esto se utiliza la función de ADIOI:

```
static void ADIO_FileSysType_fncall(char *filename,
int *fstype, int *error_code);
```

Donde se determina el tipo de sistema de archivos, que es identificado por un número, por ejemplo:

```
/* file systems */
#define ADIO_NFS      150
#define ADIO_PIOFS    151    /* IBM */
#define ADIO_UFS      152    /* Unix file system */
#define ADIO_PFS      153    /* Intel */
#define ADIO_XFS      154    /* SGI */
#define ADIO_HFS      155    /* HP/Convex */
#define ADIO_SFS      156    /* NEC */
#define ADIO_PVFS     157    /* PVFS for Linux Clusters */
#define ADIO_NTFS     158    /* NTFS for Windows NT */
#define ADIO_TESTFS   159    /* fake file system for testing */
#define ADIO_AFS      160    /* sistema de archivos distribuido AFS */
```

En esta implementación colocamos a ADIO_AFS con el número 160. La variable fstype de acuerdo a su valor entra como parámetro a otra función en ADIOI:

```
void ADIOI_SetFunctions(ADIO_File fd);
```

Donde ADIO_FILE fd tiene la información acerca del archivo, y del tipo de sistema de archivo al que pertenece. De esta forma, dentro de la función mencionada si el sistema de archivos es AFS se procede de la siguiente forma:

```
    case ADIO_AFS:
#ifdef AFS
        *(fd->fns) = ADIO_AFS_operations;
#else
        FPRINTF(stderr, "ADIOI_SetFunctions: ROMIO has not been
        configured to use AFS file system\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
#endif
```

Donde la estructura ADIO_AFS_operations contiene las funciones de *lectura/escritura* en paralelo para ser utilizadas en AFS.

Capítulo 5

BENCHMARKS CASOS DE ESTUDIO SOBRE LOS DIFERENTES SISTEMAS DE ARCHIVOS

Las operaciones de E/S en archivos es el mayor cuello de botella de una aplicación paralela. Aunque los sistemas de almacenamiento estén diseñados para obtener un alto rendimiento, las aplicaciones sólo alcanzan un bajo porcentaje del ancho de banda de E/S pico.

Esto se debe a que los sistemas de almacenamiento son optimizados para realizar grandes accesos en el orden de Megabytes, mientras que algunas aplicaciones paralelas realizan varios accesos pequeños en el orden de kilobytes o menos. Los accesos pequeños ocurren por 2 razones:

- En diversas aplicaciones paralelas cada proceso necesita acceder a un gran número de pedazos pequeños que no son contiguos en el archivo.
- Varios sistemas de archivos paralelos y distribuidos permiten al usuario solamente acceder a un pedazo de datos a la vez.

5.1. Patrones de Acceso

En consecuencia de lo anterior, es crucial para el rendimiento de una máquina paralela la transferencia óptima de datos entre la memoria y el sistema

de archivos. Muchas aplicaciones se benefician de un sistema de archivos que permite la transferencia de datos a través de funciones estandar de E/S, en nuestro caso de estudio estas son las funciones implementadas en MPI-IO.

Cualquier aplicación paralela tiene un patrón de acceso en particular de acuerdo a sus necesidades. Este patrón puede ser realizado en diferentes formas, dependendiendo de las funciones empleadas en la aplicación. Con MPI-IO [veáse 26] se han caracterizado los patrones de acceso más empleados en cuatro niveles que se describen a continuación:

Nivel 0 Varios accesos de lectura/escritura individuales con datos contiguos de un archivo.

Nivel 1 Varios accesos de lectura/escritura colectivos con datos contiguos de un archivo.

Nivel 2 Solo un acceso de lectura/escritura individual con datos no contiguos de un archivo.

Nivel 3 Solo un acceso de lectura/escritura colectivo con datos no contiguos de un archivo.

Los cuatro niveles de patrones representan un aumento de datos en cada acceso, y se presenta la oportunidad para obtener un mejor rendimiento, por ejemplo, si se tienen varios procesos con un patrón de acceso Nivel 0, estos pueden ser reemplazados por un acceso de Nivel 3 para obtener un mejor rendimiento en la tranferencia desde/hacia el Sistema de Archivos.

5.2. Benchmarks

Un benchmark es una aplicación que mide el rendimiento[1] de un componente funcional de una arquitectura de computadora, en nuestro caso de estudio se enfoca al sistema de archivos en una máquina paralela.

Debido a la variedad de los sistemas de almacenamiento, sistemas de archivos y patrones de acceso en las aplicaciones paralelas, se necesita una herramienta de medición para determinar el rendimiento de un patrón de acceso en función del sistema de archivos utilizado y de las características

[1]Medida de desempeño y precisión en función del tiempo de una máquina al realizar un proceso específico.

de la máquina paralela. Los resultados obtenidos pueden guiar a un usuario para escoger un patrón de acceso óptimo, la configuración de un sistema de almacenamiento o la configuración de un sistema de archivos.

En las siguientes secciones se describe el benchmark *b_eff_io* , utilizado para medir el rendimiento de la biblioteca paralela reducida para el sistema de archivos AFS.

5.2.1. b_eff_io

El benchmark paralelo que mide el ancho de banda efectivo de lectura/escritura (*b_eff_io*)[veáse 33], mide diferentes patrones de acceso, reporta resultados detallados y calcula el ancho de banda promedio que caracteriza al sistema de archivos.

La medición del ancho de banda de lectura/escritura parte de una hipótesis, si el sistema de alto rendimiento es balanceado, entonces el tiempo esperado para leer o re-escribir la memoria total del sistema debe ser en aproximadamente 10 minutos, con base en estadísticas de varias pruebas realizadas en diferentes computadoras paralelas con diversos sistemas de archivos.

Los parametros del benchmark se dividen en 6 categorías que se mencionan a continuación:

1. Parámetros de la Aplicación. Consiste en la organización de los datos a utilizar.

2. Utilización del Sistema. Consiste en el número de procesadores disponibles por el sistema y los hilos (threads) utilizados por cada proceso.

3. Interfaz de programación de E/S utilizada. Consiste en determinar la interfaz de programacion de E/S a utilizar, como se mencionó en el capítulo 3 hay varias bibliotecas paralelas en E/S, en nuestro caso de estudio se utiliza a MPI-IO.

4. Relaciones de la interfaz de programación paralela con la interfaz de programación de E/S. Consiste en determinar las relaciones que existen entre la interfaz de programación paralela con la interfaz de programación de E/S, para determinar los patrones de acceso a medir.

5. Sistema de Archivos utilizado. Consiste en verificar la estructura funcional del sistema de archivos paralelo o distribuido como: el número de

servidores dedicados a E/S, los servidores de metadados, servidores de archivos, entre los más importantes y las características del dispositivo de almacenamiento.

6. Cálculo del ancho de banda con base en los parámetros anteriores.

Al definir los parámetros antes mencionados de forma óptima, *b_eff_io* se vuelve dependiente de un sistema en particular. Por ello, para que sea portable se toman los puntos más característicos de los parámetros para utilizarse en varios sistemas.

El benchmark *b_eff_io* mide al Sistema de E/S con base a patrones de acceso, pero los patrones de acceso dependen de la biblioteca optimizada de E/S utilizada para alcanzar un ancho de banda aceptable. En consecuencia, otro de los resultados es la medición de eficiencia de las bibliotecas paralelas en el sistema, en nuestro caso de estudio sirve para medir la eficiencia de la biblioteca paralela reducida descrita en el capítulo 4.

Mediciones de b_eff_io

El benchmark b_eff_io mide las siguientes características:

- El tamaño del conjunto de particiones de datos utilizadas
- El ancho de banda de los métodos de acceso: escritura, escritura/lectura, lectura.
- Los patrones de acceso, divididos en 4 tipos:

  0 Acceso colectivo en pedazos, consiste en dispersar pedazos de datos de la memoria hacia disco y viceversa (Figura 5.1).

  1 Acceso colectivo en pedazos, aplicar una operación de lectura o escritura a disco por pedazo de datos (Figura 5.2).

  2 Acceso no colectivo a un archivo por proceso MPI, consiste en leer o escribir en archivos separados (Figura 5.3.

  3 El mismo patrón de acceso de (2) pero los archivos individuales se ensamblan por segmentos en un solo archivo (Figura 5.4).

  4 El mismo patrón de acceso de (3) pero el ensamble del archivo se hace de forma colectiva (Figura 5.4).

Los archivos utilizados, no se reutilizan para medir otros patrones de acceso.

- Los pedazos de datos bien formados, que se identifican con el tamaño que es una potencia de 2, o pedazos de datos mal formados que su tamaño es de un pedazo bien formado más 8 bytes, generados en la ejecución del benchmark al emplear los patrones de acceso mencionados.

- Diferentes tamaños de pedazos de datos, de $1KB$, $32Kb$, $1MB$, $2M$ hasta $\frac{1}{128}$ del tamaño de la memoria de un nodo donde se ejecuta un proceso del benchmark *b_eff_io*.

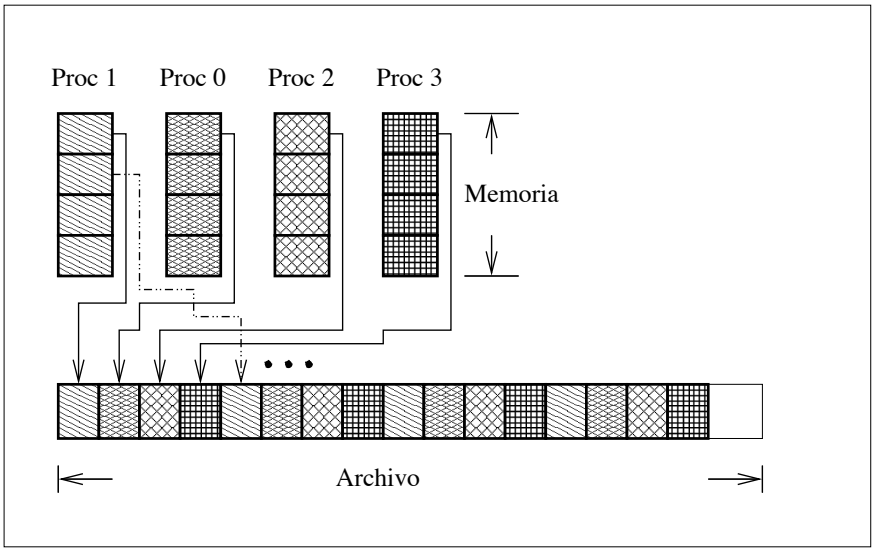

Figura 5.1: Patron tipo 0

De esta forma, el ancho de banda para un patrón de acceso se define como el número de bytes transferidos entre el tiempo desde que se abre el archivo hasta que se cierra. Y el ancho de banda de un método de acceso se define como el promedio de todos los anchos de banda por tipo de patrón de acceso, donde el ancho de banda del patrón de acceso 0 se toma como un valor doble. A continuación se presentan los resultados del benchmark *b_eff_io*.

5.3. Resultados

5.3.1. Equipo Utilizado

El cluster de pruebas contiene las siguientes piezas de hardware:

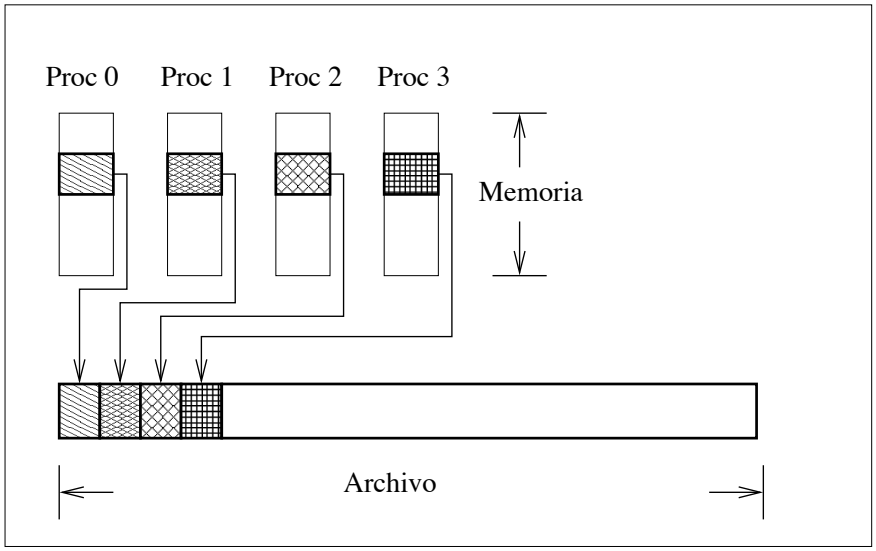

Figura 5.2: Patron tipo 1

Nodo Maestro :

- Pentium 4 HT a 3.2 Ghz
- Disco Duro SATA de 160GB 7200 rpm
- 1GB memoria DDR 2 533 Mhz

Nodo Servidor de Archivos AFS y de Cálculo :

- Pentium D a 3.2 Ghz
- Disco Duro SATA II de 160GB 7200 rpm
- 512MB memoria DDR 2 533Mhz

Nodo Servidor de Metadatos PVFS y de Cálculo :

- Pentium 4 a 2 Ghz
- Disco Duro ATA 133 de 80GB 7200 rpm
- 1.5GB memoria RamBus 800Mhz

4 Nodos Servidores de E/S PVFS y de Cálculo :

- Pentium 4 a 2.4 Ghz
- Disco Duro ATA 133 de 80GB 7200 rpm
- 512MB memoria DDR 400Mhz

Nodo de Cálculo sin funciones adicionales :

- Pentium 4 a 1.9 Ghz
- Disco Duro ATA 133 de 80GB 7200 rpm
- 2GB memoria RamBus 800Mhz

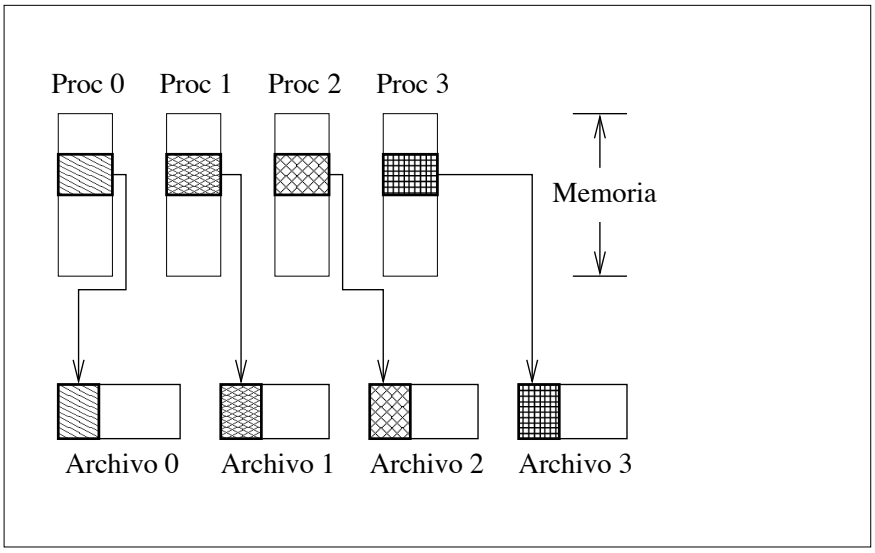

Figura 5.3: Patron tipo 2

En este caso, los nodos servidores de E/S PVFS, el nodo servidor AFS y el nodo servidor de metadados de PVFS también funcionan como nodos de cálculo. Todos los nodos tienen el sistema operativo Linux distribución Slackware. La versión del kernel de linux es 2.4.29 en todos los nodos excepto en el nodo servidor AFS que es la versión 2.6.17.8.

La versión de mpich utilizada es la 1.2.5 donde se desarrolló la biblioteca paralela reducida para el sistema de archivos AFS descrita en el capítulo 4.

El sistema de archivos distribuido AFS cuenta con un solo servidor AFS con una capacidad de 130GB, todos los demás nodos son clientes AFS.

El sistema de archivos paralelo PVFS tiene una capacidad de 147GB repartida en cada uno de los nodos de E/S con 36GB, como se mencionó en el capitulo 2, los nodos pueden actuar como nodos cliente y nodos de E/S a la vez.

La red de interconexión entre los nodos del cluster es mediante un switch ethernet 10/100 Mbps.

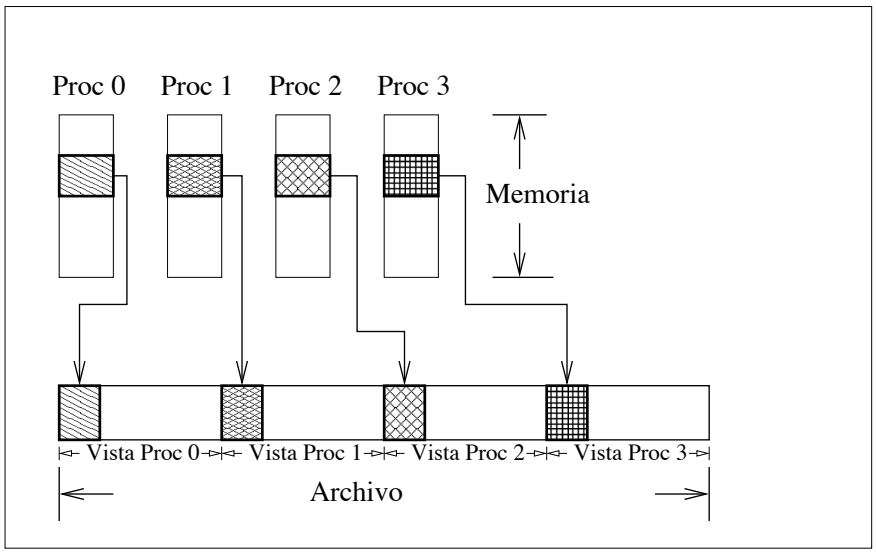

Figura 5.4: Patron tipo 3/4

5.3.2. Comparación de Sistemas de Archivos con *b_ eff_ io*

El benchmark *b_ eff_ io* se probó en los sistemas de archivos PVFS y AFS. A continuación se muestran las gráficas del rendimiento por cada sistema de archivos.

Resultados en PVFS

Los resultados obtenidos con el benchmark *b_ eff_ io* en el sistema de archivos PVFS se muestra en la Figura 5.5.

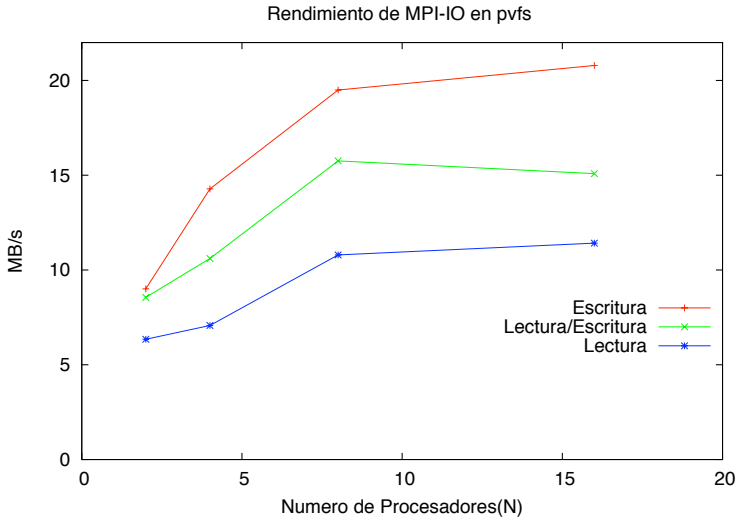

Figura 5.5: Prueba en pvfs

Como se muestra en la Figura 5.5, la tasa de tranferencia de la actividad de escritura para 2 procesos es de $9.004 MB/s$, y para 16 procesos o más es de $20.790 MB/s$ aproximadamente. Esto es debido a que se presenta un cuello de botella porque los procesos son más que los procesadores disponibles, que los nodos disponibles y en el caso de PVFS más procesos que los nodos dedicados a E/S.

En el caso de la operación de lectura, la tasa de tranferencia para 2 procesos es de $6.344 MB/s$, y para 16 o más procesos es de $11.419 MB/s$ aproximadamente. Esto es debido a que se presenta un cuello de botella porque

como en la operación de escritura, hay mas procesos que nodos disponibles, procesadores y nodos dedicados a E/S, pero presenta un retardo mayor al de la escritura.

En el caso de la operación de lectura/escritura, la tasa de transferencia para 2 procesos es de $8.554MB/s$, y para 16 procesos o más es de $15.085MB/s$ aproximadamente. Esto es debido al cuello de botella presentado en la operación de escritura y lectura, afectando el retardo en la lectura mencionado en el párrafo anterior.

Como se menciona en el capítulo 2, PVFS es orientado a un alto rendimiento en las operaciones de escritura, tal como se demuestra con los datos obtenidos del benchmark b_eff_io.

Resultados en AFS

Los resultados obtenidos con el benchmark b_eff_io en el sistema de archivos AFS se muestra en la Figura 5.6.

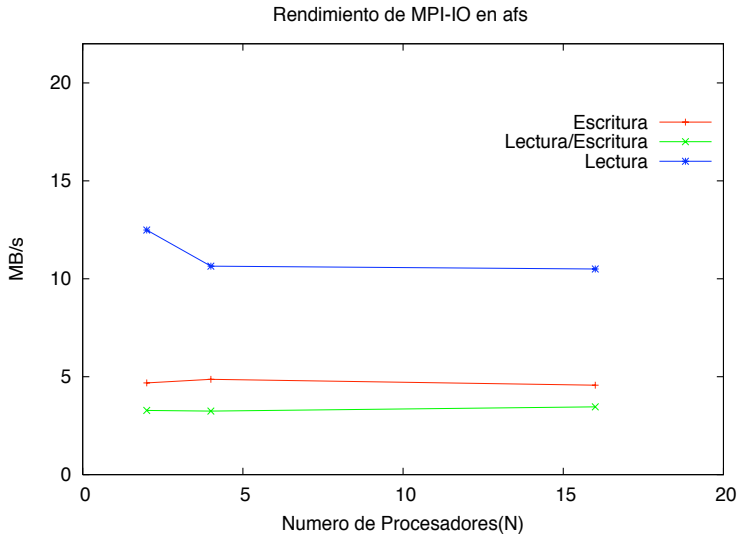

Figura 5.6: Prueba en afs

Como se muestra en la Figura 5.6, la tasa de transferencia de la actividad de escritura para 2 procesos es de $4.694MB/s$, y para 16 procesos o más es

de $4.566MB/s$ aproximadamente. Esto se debe a que se presenta un cuello de botella a limitaciones del hardware, el nodo servidor de archivos de AFS como se menciona en las secciones anteriores, no cuenta con un arreglo de discos y solo tiene un solo acceso a la red interna del cluster. Las peticiones de lectura o escritura de los nodos clientes AFS tienen que retrasarse debido a que la demanda generada por los procesos de *b_ eff_ io* satura a la red interna del cluster. Esto se debe a que hay más procesos que nodos clientes AFS disponibles y el servidor de archivos AFS genera un retraso en las respuestas de las peticiones a los procesos.

En el caso de las operaciones de lectura, la tasa de transferencia para 2 procesos es de $12.490MB/s$, y para 16 procesadores o más es de $10.501MB/s$ aproximadamente. La principal diferencia entre la lectura y escritura, es que en la escritura se hace una operación adicional de actualización del cache manager en cada unos de los clientes AFS, lo que decae en su rendimiento de escritura.

En el caso de las operaciones de lectura/escritura, la tasa de transferencia para 2 procesos es de $3.279MB/s$, y para 16 procesadores o más es de $3.462MB/s$ aproximadamente. Esto se debe a la combinación de los cuellos de botella generados por la lectura que es debida a limitaciones en el hardware y el cuello de botella generado por la escritura es debido a la operación de actualización de datos y la limitación del hardware.

Operaciones de Escritura

A continuación se muestra en la Figura 5.7 una comparativa de las operación de escritura entre AFS y PVFS. En la figura mencionada, se muestra la diferencia de rendimiento en la escritura del sistema de archivos PVFS que es 3.5 veces más que en AFS para 16 procesadores. Mientras que para 2 procesadores el rendimiento de PVFS es 0.9 veces más que en AFS.

Operaciones de Lectura

A continuación se muesta en la Figura 5.8 una comparativa de la operación de lectura entre AFS y PVFS.

En la figura antes mencionada, se muestra la diferencia de rendimiento en la lectura del sistema de archivos PVFS que es 0.08 veces más que en AFS.Mientras que para 2 procesadores el rendimiento de AFS es 1.9 veces más que PVFS. En este caso para las operaciones de lectura tiene un mejor

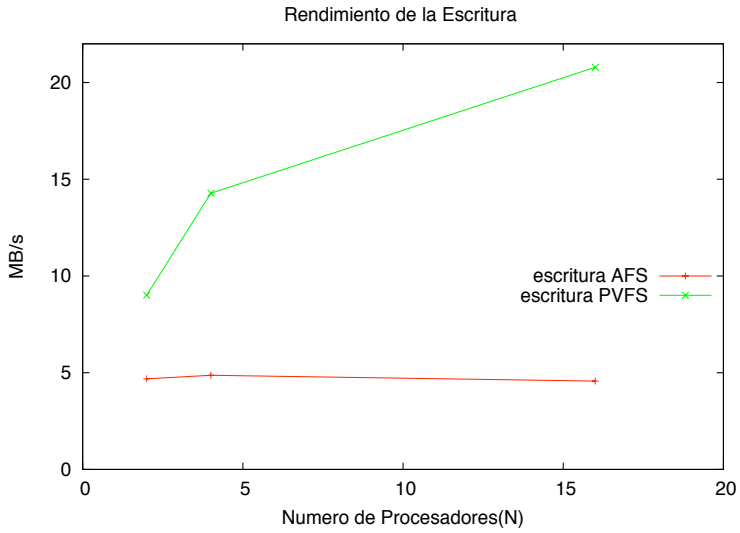

Figura 5.7: Operaciones Escritura

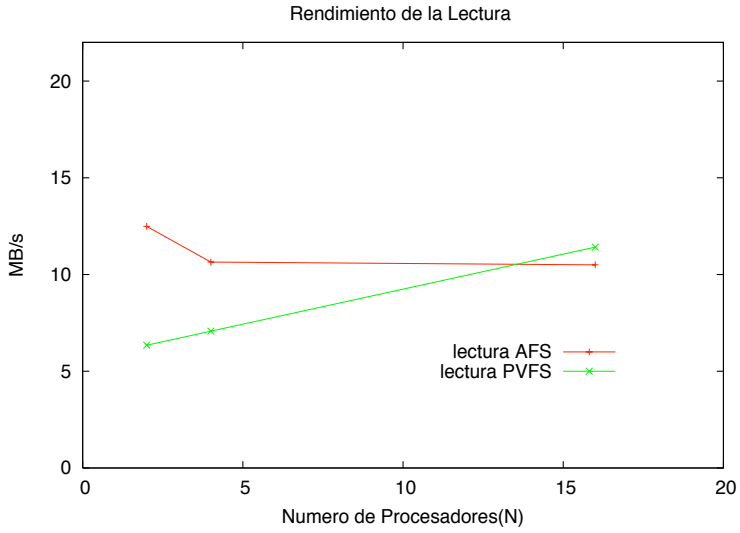

Figura 5.8: Operaciones Lectura

rendimiento AFS.

Operaciones de Lectura/Escritura

A continuación se muestra en la Figura 5.9 una comparativa de la operación lectura/escritura entre AFS y PVFS.

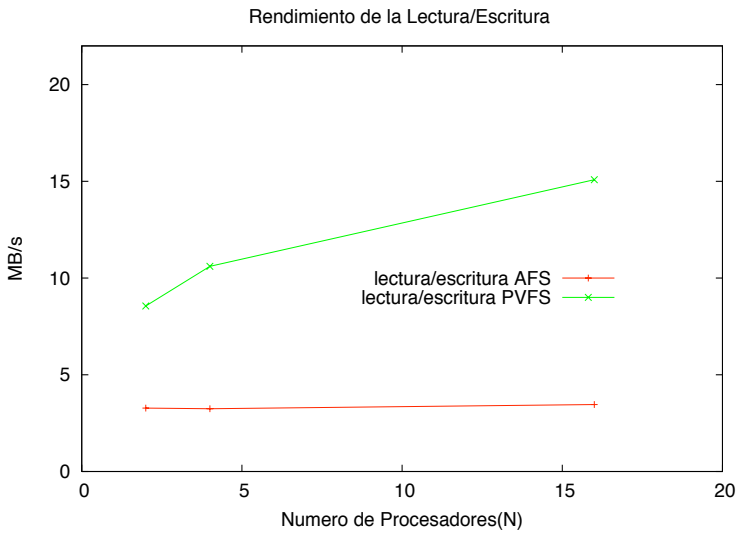

Figura 5.9: Operaciones Lectura/Escritura

En la figura antes mencionada, se muestra la diferencia de rendimiento en la lectura/escritura del sistema de archivos PVFS que es 3.3 veces más que en AFS. Mientras que para 2 procesadores el rendimiento de PVFS es 2.6 veces más que en AFS.

Como se observa en las figuras anteriores el sistema de archivos AFS tiene un mejor rendimiento en cuanto a la lectura que en la escritura. Esto es debido al cache manager que tiene que reportar al servidor AFS los cambios al archivo y notificar a todos los clientes que se realizó un cambio en el pedazo de archivo que se esta utilizando por otros procesos de la aplicación paralela, lo que ocasiona un retardo adicional en las operaciones de escritura. Esto puede solucionarse por medio de hardware, ya que como se había mencionado antes, el servidor de archivos tiene limitaciones en cuando a hardware y genera cuellos de botella adicionales al generado por el servidor de archivos AFS.

Si el servidor de archivos de AFS tuviera un arreglo de discos y una interconexión de red a una velocidad más alta que la red interna del cluster, el cuello de botella se generaría solamente a la capacidad de atender peticiones por parte del servidor AFS.

Capítulo 6

APLICACIONES PARA EL CÓMPUTO PARALELO EN ARCHIVOS

En un ambiente de cómputo paralelo, las aplicaciones científicas distribuyen sus datos entre múltiples procesos ejecutandose en varios procesadores. En algunas aplicaciones conocidas como "embarazosamente paralelas", los procesos se sincronizan debido al frecuente intercambio de datos durante su ejecución. En las aplicaciones de simulación a gran escala y dependientes del tiempo, se requieren los datos periódicos del estado actual que ocurre en el procesamiento del fenómeno físico para su análisis y visualización [veáse 27].

En consecuencia, estas aplicaciones tienen una actividad de lectura y escritura intensiva. En el ambiente de aplicaciones a gran escala, de forma periódica se guardan los datos y el estado del proceso en un archivo conocido como *checkpoint* como prevención a una falla del sistema. De esta manera, puede reiniciarse la aplicación desde un *checkpoint* antes de que ocurriera la falla en el sistema.

Debido al bajo rendimiento de los sistemas de almacenamiento y el software poco eficiente, el rendimiento de las operaciones de E/S es el mayor cuello de botella en las aplicaciones paralelas. En consecuencia se proponen las operaciones de E/S colectiva, donde todos los procesos cooperan para realizar operaciones de E/S a gran escala, como alternativa para evitar los cuellos de botella en varias aplicaciones paralelas, y varias bibliotecas de E/S paralela vistas en el capítulo 3, contienen funciones colectivas de E/S.

6.1. Actividad de E/S en Aplicaciones Paralelas

Para el sistema, la actividad de E/S de una aplicación paralela implica las transferencias de datos entre un nodo y:

a) Sistema de Almacenamiento

b) Red de Interconexión

c) El resto de los nodos

d) Dispositivos especiales, por ejemplo un sistema de visualización.

Para un programador, la actividad de E/S implica:

a) El flujo de datos de entrada para la aplicación.

b) Movimiento de datos intermedios generados durante la ejecución de la aplicación.

c) El fujo de salida de datos que representan los resultados de la aplicación.

Una aplicación con actividad intensa de E/S requiere datos de entrada para la inicialización del cálculo, o entradas de datos por intervalos durante el procesamiento.

Los datos intermedios pueden originarse por 3 razones: por el checkpoint o reinicio de la aplicación, por arreglos de datos más grandes que la memoria principal, o resultados después de un cálculo para ser utilizados en otra parte de la aplicación. El diseño de las aplicaciones paralelas con actividad intensiva de E/S consiste en determinar cuándo cada proceso procesará sus propios datos o cuándo debe de compartirlos con los demás procesos, lo que implica transferencias de datos con resultados intermedios. En el caso de un checkpoint de una aplicación, es determinar los resultados intermedios que pertencen a cada proceso e identificarlos para que reinicie sin modificación el estado de los procesos.

Así como la entrada de los datos, la salida de datos puede ocurrir al concluir la ejecución de la aplicación o en intervalos a través del cálculo. La principal característica de los datos de salida es que puedan ser analizados por medio de un proceso serial, que se ejecute en una estación de trabajo o computadora personal.

6.2. Rendimiento de E/S

La forma más sencilla para alcanzar un buen rendimiento de E/S, consiste en que cada procesador lea o escriba datos en su propio disco local, en el caso de que cada uno de los procesadores sea de la misma capacidad para realizar operaciones de E/S. En contraste, los resultados de varias aplicaciones paralelas se utilizan para su visualización, como son las simulaciones dependientes del tiempo, que manejan un formato de salida en un orden de renglones/columnas requerido por el proceso de visualización.

En consecuencia, los datos en el disco requieren una organización diferente en la memoria principal , y varias bibliotecas de E/S paralela la soportan, como es el caso de MPI-IO con el método de *data sieving* explicado en el capítulo 4. De esta manera se mejora el rendimiento de las aplicaciones paralelas. Pero existen otros factores que disminuyen el rendimiento de E/S en una aplicación paralela, como son los servidores de E/S, los servidores de metadatos o los servidores de archivo.

Cuando se ejecuta una aplicacion paralela, el planificador de procesos podría de forma ideal colocar los procesos de la aplicación en los procesadores de los servidores de E/S y de esta forma incrementar el rendimiento de E/S. En contraste, en un cluster heterogéneo el rendimiento de E/S tiende a ser variable porque algunos servidores de E/S son más lentos de otros servidores.

6.3. Aplicación BT

Como ejemplo de una aplicación paralela que utiliza las operaciones de E/S paralela es el problema de bloques tridiagonales (BT). Esta aplicación forma parte del conjunto de los benchmaks paralelos desarrollados por parte de la NASA [veáse 24].

La aplicación BT emplea una descomposición de dominio llamada multipartición [veáse 5]. Esta consiste en que cada proceso realiza cálculos sobre múltiples subconjuntos cartesianos que pertenecen al dominio, donde el número de subconjuntos esta en función de la raíz cuadrada del número de procesos que participan. En la implementación, cada proceso escribe directamente en un archivo de salida los elementos procesados. En consecuencia, se requiere de un gran rendimiento del sistema para procesar los datos fragmentados. Por ello, el problema y la forma de su implementación sirve como un candidato para verificar el rendimiento de una aplicación paralela utilizando

un buffer colectivo.

La aplicación realiza los cálculos en pasos con respecto al tiempo. Cada cinco pasos un elemento de solución consiste de cinco numeros de precisión doble por cada punto de la malla, que será escrito a uno o más archivos. Después de que todos los pasos en el tiempo son calculados, todos las soluciones que pertenecen a un solo paso deben escribirse en el mismo archivo, y colocarse en orden de componente de vectores en las direcciones x, y, z respectivamente.

El reordenamiento de las soluciones en el archivo, es considerado en el tiempo de la corrida de la aplicación paralela. Después del reordenamiento de datos, se procede a la verificación de los datos y se determina si el cálculo es correcto o no. A partir de los cálculos y el tiempo de pared ocupado por la aplicación, se determinan los *flops(operaciones de punto flotante por segundo)* efectivos de la aplicacion paralela. Las operaciones de E/S de esta aplicación pueden realizarse en varias formas, de esta manera podemos ejecutar la aplicación con diferentes patrones de acceso y determinar su rendimiento, estas opciones de ejecutar la aplicación se mencionan a continuación:

1. **full:** *MPI-IO con operaciones colectivas*. Esto significa que los datos dispersos en memoria por los procesos, son reunidos en un conjunto para todos los procesos y antes de escribirse al archivo son reordenados para incrementar la granularidad.

2. **simple:** *MPI-IO sin operaciones colectivas*. Esto significa que no son reordenados los datos, por lo que se realizan varios accesos para escribir los datos al archivo.

3. **fortran:** Lo mismo que el 2, pero se utilizan las funciones de fortran 77 para manejo de archivos, en vez de MPI-IO.

4. **epio:** Cada proceso escribe sus datos en un archivo separado.

6.3.1. Comparación entre sistemas de archivos

En la figura 6.1 el programa BTIO con el patrón de acceso **full** en el sistema de archivos afs muestra un rendimiento desde 1 proceso de $340 Kflops$, hasta su rendimiento máximo que fue con 4 procesos de $1306 Kflops$ en el nivel A.

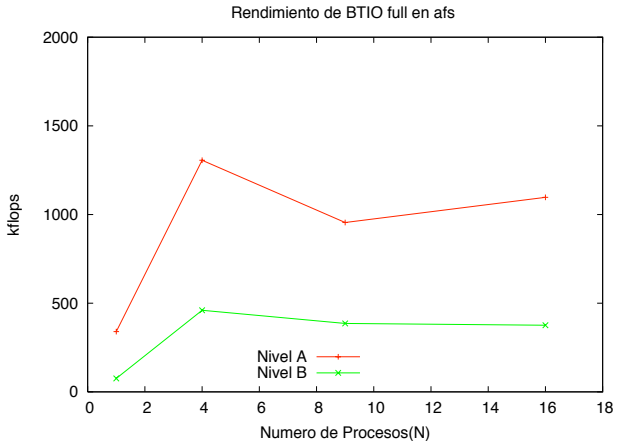

Figura 6.1: Rendimiento de BTIO full en afs

En la figura 6.2 el programa BTIO con el patrón de acceso **full** en el sistema de archivos pvfs muestra un rendimiento desde 1 proceso de $316Kflops$, hasta su rendimiento máximo que fue con 4 procesos de $2224Kflops$ en el nivel A.

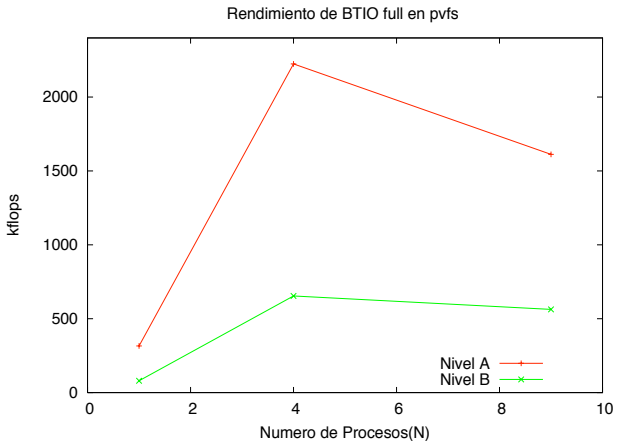

Figura 6.2: Rendimiento de BTIO full en pvfs

En la figura 6.3 el programa BTIO con el patrón de acceso **epio** en el sistema de archivos afs muestra un rendimiento desde 1 proceso de $319Kflops$, hasta su rendimiento máximo que fue con 4 procesos de $2036Kflops$ en el nivel A.

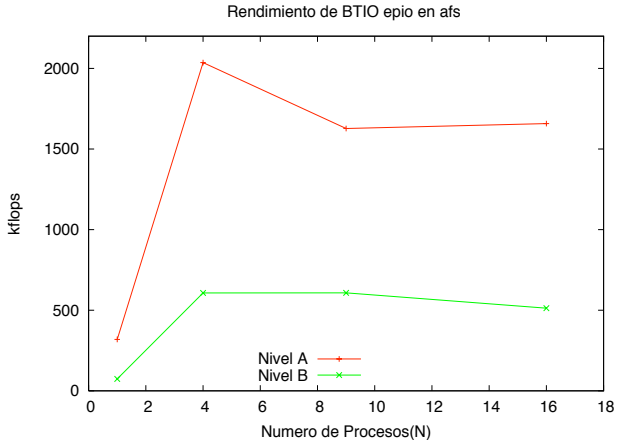

Figura 6.3: Rendimiento de BTIO epio en afs

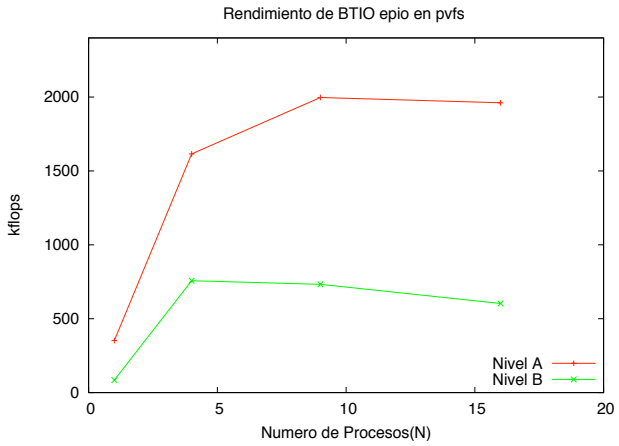

Figura 6.4: Rendimiento de BTIO epio en pvfs

En la figura 6.4 el programa BTIO con el patron de acceso **epio** en el sistema de archivos pvfs muestra un rendimiento desde 1 proceso de $352Kflops$, hasta su rendimiento máximo que fue con 9 procesos de $1997Kflops$ en el nivel A.

Rendimiento Btio full

En la figura 6.5 se muestra la diferencia de rendimiento en las operaciones de punto flotante para la aplicación BTIO con el patrón de acceso *full* nivel A donde se resuelve un sistema de ecuaciones diferenciales parciales de $64x64x64$ en los sistema de archivos PVFS y AFS.En el sistema de archivos PVFS, BTIO alcanza un máximo rendimiento con 4 procesos de $2224kflops$ que es 42 % mayor que el alcanzado por AFS.

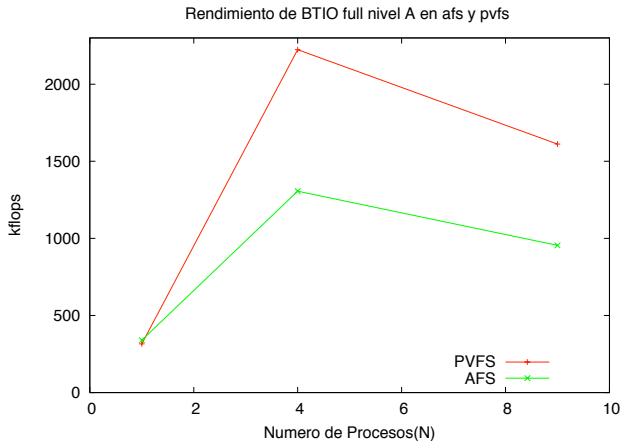

Figura 6.5: Rendimiento de BTIO full nivel A

En la figura 6.6 se muestra la diferencia de rendimiento en las operaciones de punto flotante para la aplicación BTIO con el patrón de acceso *full* nivel B donde se resuelve un sistema de ecuaciones diferenciales parciales de $102x102x102$ en los sistemas de archivos PVFS y AFS.En el sistema de archivos PVFS alcanza un máximo rendimiento con 4 procesos de $654kflops$ que es 30 % mayor que el alcanzado por AFS.

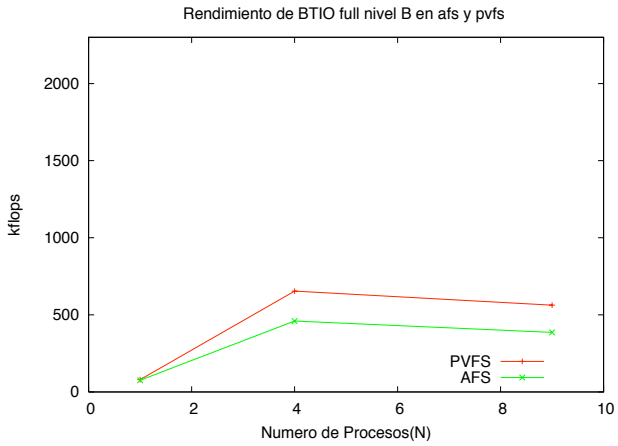

Figura 6.6: Rendimiento de BTIO full nivel B

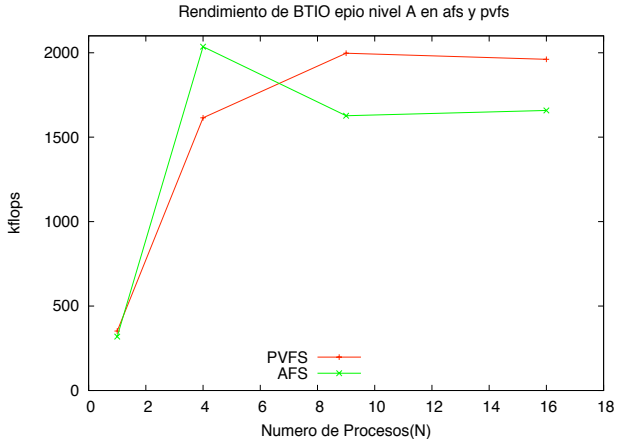

Figura 6.7: Rendimiento de BTIO epio nivel A

Rendimiento Btio epio

En la figura 6.7 se muestra la diferencia de rendimiento en las operaciones de punto flotante para la aplicación BTIO con el patrón de acceso *epio* nivel A donde se resuelve un sistema de ecuaciones diferenciales parciales de $64x64x64$ en los sistemas de archivos PVFS y AFS. En el sistema de archivos AFS alcanza un máximo rendimiento con 4 procesos de $2036kflops$ que es 21 % mayor que el alcanzado por PVFS.

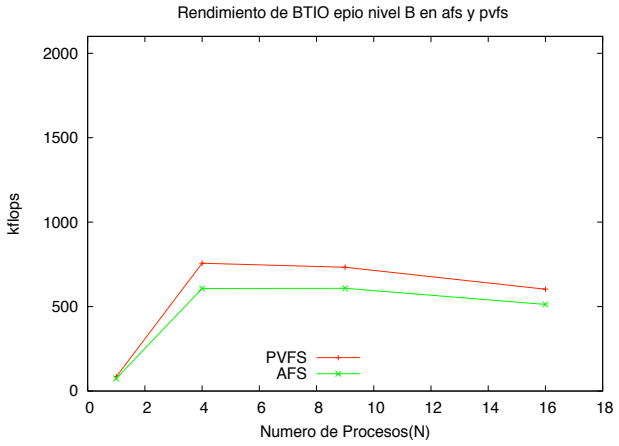

Figura 6.8: Rendimiento de BTIO epio nivel B

En la figura 6.8 se muestra la diferencia de rendimiento en las operaciones de punto flotante para la aplicación BTIO con el patrón de acceso *epìo* nivel B donde se resuelve un sistema de ecuaciones diferenciales parciales de $102x102x102$ en los sistemas de archivos PVFS y AFS. En el sistema de archivos PVFS alcanza un rendimiento máximo con 4 procesos de $757kflops$ que es 20 % mayor que el alcanzado por PVFS.

Como se observa en las figuras anteriores, el sistema de archivos PVFS muestra un mejor rendimiento en las operaciones de punto flotante que el sistema de archivos AFS, execpto en las aplicaciones donde se utiliza un solo procesador. Debido a la dependencia de datos que se presenta al resolver el sistema de ecuaciones diferenciales, el cache manager de AFS tiene que actualizar los datos continuamente para ser compartidos por los procesos participantes, lo que empeora su rendimiento. Como se menciona en el ca-

pítulo anterior, el problema puede ser solucionado por hardware, porque el servidor de archivos AFS tiene limitaciones en su hardware y ocasiona los cuellos de botella que empeoran el rendimiento de las aplicaciones.

Capítulo 7

CONCLUSIONES

Se ha cumplido el objetivo de encontrar formas de analizar el rendimiento de los sistemas de archivos en paralelo para las aplicaciones científicas. Para llevar a cabo el análisis de un sistema de archivos, se analizan las características y la distribución de datos sobre un sistema de almacenamiento. En este caso, se estudiaron las formas de distribución de los datos que realiza un sistema de archivos sobre un disco duro. A partir de cierta forma de distribución de datos realizada por el sistema de archivos, se observaron las configuraciones en cuanto a hardware para obtener un mejor rendimiento de un sistema de archivos, estas configuraciones son hechas mediante los niveles RAID vistos en el capítulo 2.

Las aplicaciones científicas son realizadas en sistemas de alto rendimiento descritos en el capítulo 1, por lo que se estudiaron las diferentes configuraciones de máquinas de alto rendimiento y después relacionar los sistemas de almacenamiento con las máquinas de alto rendimiento. Una parte esencial de las aplicaciones científicas es el vínculo para que los datos situados en el sistema de archivos estén disponibles para los procesos en una computadora de alto rendimiento, en este caso la mayor parte de las aplicaciones de cómputo de alto rendimiento requieren paralelización, porque el problema que resuelven se convierte en un problema intratable en una computadora de un solo procesador. En consecuencia, se desarrollan herramientas de software como son las bibliotecas paralelas que ayudan al desarrollo de aplicaciones científicas de alto rendimiento, donde los procesos se comunican para compartir datos.

Los datos a procesar por una aplicación paralela pueden ser enviados por mensajes a través de la red, pero también hay aplicaciones que se comunican

a través de compartir archivos y realizar diversas operaciones de *lectura/escritura* donde la principal característica es un volumen de datos mucho más grande que lo ofrecido por un buffer de comunicación de mensajes, como es en el caso de MPI.

Como caso de estudio se utilizó el sistema de archivos distribuido AFS y el sistema de archivos paralelo PVFS, Para el sistema de archivos AFS se desarrolló la biblioteca paralela reducida, con el fin de verificar el rendimiento del sistema de archivos para las aplicaciones científicas en un sistema de archivos distribuido y compararlo con el rendimiento obtenido en un sistema de archivos paralelo como PVFS.

Al realizar la comparación se aplicó en benchmark *b_eff_io* para determinar las tasas de transferencias de datos en los sistemas de archivos PVFS y AFS. Para realizar un análisis de rendimiento de cada sistema de archivos, se utilizó la aplicación BTIO y se determinó el rendimiento en cada sistema de archivos.

Con los datos obtenidos y el procedimiento realizado se concluye que el sistema de archivos paralelo PVFS es una buena opción para almacenar datos para ser procesados por aplicaciones científicas paralelas que requieran operaciones de E/S intensiva. Sin embargo, uno de los puntos débiles del sistema de archivos PVFS es que no es tolerante a fallas, si uno de los nodos de E/S falla se pierden los datos procesados.

En consecuencia, se utilizó al sistema de archivos AFS que es tolerante a fallas, porque contiene herramientas de recuperación de datos y replicación de datos, por ello se desarrollo la biblioteca paralela reducida en la capa inferior de ADIO descrita en el capítulo 4.

Los datos arrojados en cuanto a la transferencia de datos por el benchmark *b_eff_io* no son satisfactorios porque se requería que fuera al menos un 50 % menor en las operaciones de escritura. El comportamiento se debe a que el cache manager debe de ser actualizado para que la última versión de los datos esté disponible para los procesos de la aplicación paralela.

En el rendimiento de la aplicación *BTIO*, para el patrón de acceso *full* que se refiere a operaciones colectivas de E/S en el sistema de archivos AFS no son aceptables, porque se presentan algunos errores en la verificación de la solución del problema. Pero en el patrón de acceso *epio* que se refiere a operaciones separadas de E/S por proceso son aceptables, no presentan errores en la verificación de datos y alcanzan un rendimiento aceptable comparado al obtenido por PVFS.

En el caso del sistema de archivos AFS que como se menciona antes en el

capítulo 5, se compone de un solo servidor de archivos, mientras que el sistema de archivos PVFS consiste en 4 servidores de E/S y un servidor de metadatos, si sumamos la tasas de transferencias de cada uno de los servidores de E/S supera por mucho a la interfaz de red que tiene el servidor de archivos AFS, en consecuencia la demanda de datos supera la capacidad de transferencia del servidor AFS, de esta manera genera un cuello de botella al realizar las operaciones de E/S y perdidas de datos a procesar como fue el caso en la aplicación BTIO.

Estos datos sirven de base para mejorar la configuración de un sistema de archivos y la biblioteca paralela reducida con el fin de que AFS se convierta en una opción aceptable y segura para las aplicaciones científicas. A continuación se describe un caso práctico ocurrido en la supercomputadora SGI Altix situada en la Dirección General de Servicios en Cómputo Académico perteneciente a la UNAM.

7.1. Caso Práctico

El caso práctico ocurre en la supercomputadora SGI Altix que consiste en:

a) 24 procesadores Itanium 2 a 1.7Ghz.

b) 24GB de memoria ram, fisicamente distribuida, lógicamente compartida.

c) Interconexión de red NumaLink

d) Topología de interconexión Fat-tree

e) Sistema de almacenamiento de 1 TB distribuido en un arreglo de discos en un nivel RAID 5.

El sistema de archivos de la supercomputadora era lento, como se muestra en la figura 7.1.

Se utilizó el benchmark *b_eff_io* para determinar los resultados mostrados en la figura 7.1, el rendimiento en las operaciones de lectura va desde $145MB/s$ utilizando 2 procesadores hasta $75MB/s$ utilizando los 24 procesadores que conforman la supercomputadora.

En este caso se atribuye a un cuello que botella que afecta en casi 50 % al rendimiento de la operaciones de lectura. Se planean dos hipótesis respecto

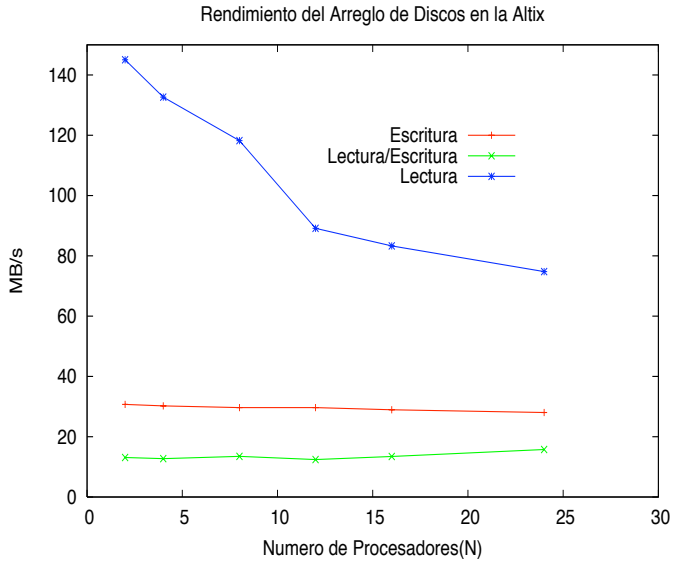

Figura 7.1: Resultados Altix

al cuello de botella: la primera a que sólo hay una conexion desde los procesadores hacia el sistema de almacenamiento, la segunda, que el Sistema de Almacenamiento es más lento que la velocidad de conexión a los procesadores.

En las operaciones de escritura, el rendimiento se mantiene en $30MB/s$ promedio utilizando 2 procesadores hasta los 24 procesadores. En este caso existe un cuello de botella debido a que la velocidad de escritura permanece constante aún incrementando el número de procesos.

Esto se debe a que el sistema de almacenamiento cuenta con un sistema RAID nivel 5, donde el rendimiento de la lectura y escritura dependen del tamaño de stripe utilizado. Un stripe se refiere al tamaño mínimo en bytes que se distribuyen por igual en cada uno de los discos del arreglo. Si el tamaño del stripe es pequeño, se obtiene un buen rendimiento en la lectura y un bajo rendimiento en la escritura, como se muestra en la Figura 7.1. Esto se debe al cálculo para obtener los bits de paridad por cada stripe escrito al disco, en el caso de lectura no se realiza ningun cálculo a menos de que un disco este dañado.

Se sugirió que para mejorar el rendimiento en cuanto a la escritura, se debe aumentar el tamaño del stripe utilizado en el nivel RAID 5. Y para minimizar el cuello de botella, utilizar otro tipo de red de conexión del sistema de almacenamiento hacia los procesadores.

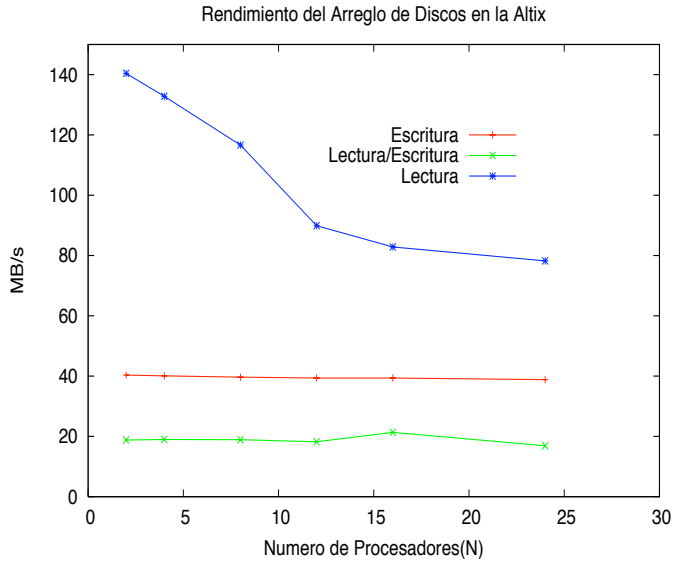

Figura 7.2: Resultados Altix primera modificación

El resultado se observa en la figura 7.2, se observa el rendimiento de las operaciones de lectura cuya tasa de transferencia empieza en $140MB/s$ para dos procesadores, hasta $78MB/s$ para 24 procesadores. En las operaciones de escritura, su tasa de transferencia empieza en $40MB/s$ para dos procesadores, hasta $38MB/s$ para 24 procesadores. Y en las operaciones de lectura/escritura, la tasa de transferencia empieza en $18MB/s$ para dos procesadores, hasta $16MB/s$ para 24 procesadores.

En este caso, el sistema de archivos mejoró en un 10 % en las operaciones de escritura y lectura/escritura. En las operaciones de lectura permaneció constante, se procedió a realizar otra modificación que se muestra en la figura 7.3.

Se observa el rendimiento de las operaciones de lectura cuya tasa de transferencia empieza en $141MB/s$ para dos procesadores, hasta $79MB/s$ para 24

procesadores. En las operaciones de escritura, su tasa de transferencia empieza en $82MB/s$ para dos procesadores, hasta $77MB/s$ para 24 procesadores. Y en las operaciones de lectura/escritura, la tasa de transferencia empieza en $31MB/s$ para dos procesadores, hasta $45MB/s$ para 24 procesadores.

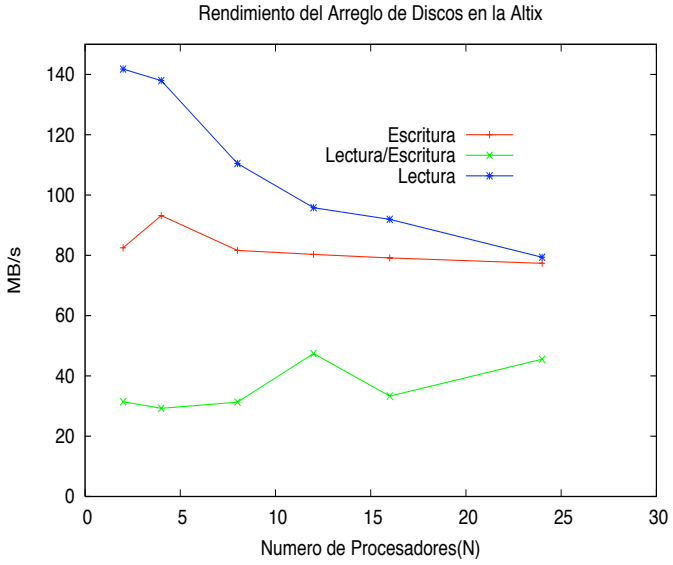

Figura 7.3: Resultados Altix ultima modificación

En la última configuración del arreglo de discos se observa que el aumento del rendimiento en las operaciones de lectura es mínimo, en las operaciones de escritura se observa el aumento del rendimiento en un 2 veces más y para las operaciones de lectura/escritura es de 2.81 veces más con respecto a los datos de la modificación anterior mostrada en la Figura 7.2.

Al modificar el tamaño del stripe en el arreglo de discos en el nivel RAID 5, aumenta o disminuye el rendimiento de las operaciones de escritura. En el caso específico del arreglo de discos puede modificarse aún más para elevar el rendimiento de la escritura, pero afectará al rendimiento en las operaciones de lectura.

En consecuencia, el arreglo de discos cumple con las condiciones necesarias para la ejecución de aplicaciones científicas. La condición es que la tasa de transferencia de escritura debe ser al menos el 50 % de la tasa de transferencia

por operaciones de lectura. En el arreglo de discos la tasa de transferencia de escritura para 2 procesadores empieza con el 58 % de la tasa de lectura, hasta el 97 % de la tasa de lectura para 24 procesadores.

En este caso se observa un cuello de botella en las operaciones de lectura a medida que aumenta el número de procesadores, mientras que el rendimiento de las operaciones de escritura es casi constante. Esto es debido a la organización del nivel RAID 5 donde se saturan las peticiones de los procesos hacia el sistema de archivos.

7.2. Trabajos Futuros

El trabajo realizado servirá como base para desarrollar una biblioteca paralela de E/S más optimizada para el sistema de archivos AFS, así como el inicio del desarrollo de un conjunto de benchmarks paralelos que midan el rendimiento de los sistemas de archivos para computadoras de alto rendimiento.

Cada día se necesita una mayor capacidad en los sistemas de almacenamiento, porque las necesidades de los problemas del gran reto lo requieren así, la generación de datos procesados por las aplicaciones científicas con actvidad de E/S intensiva aumenta de forma considerable y se deben de buscar patrones de acceso más optimizados para que se obtenga el mejor rendimiento posible.

Otra de las tecnologías a utilizar son las Grid's computacionales como descritas en el capítulo 1, donde la demanda de datos es del orden de Terabytes, para lo que se necesita una biblioteca paralela especializada en Grid's computacionales para operaciones de E/S en sistemas de archivos distribuidos a lo largo de la Grid.

En cuanto a los sistemas de almacenamiento, se renuevan de forma constante, en consecuencia se necesita del desarrollo e inovación de los sistemas de archivos, para encontrar formas de distribuir los datos en los archivos de forma más eficiente y adecuada para los nuevos disposivos de almacenamiento.

APÉNDICES

Apéndice A

BIBLIOTECA PARALELA REDUCIDA

A.1. Estructura del ADIO inferior

A.1.1. Estructura ADIOI_FileD

La implementación en la capa de ADIO (ADIOI) hace referencia al sistema de archivos y al archivo en uso para realizar las operaciones de E/S. Para definir los atributos de un archivo en ADIOI se utiliza la estructura ADIOI_FileD:

```
struct ADIOI_FileD {
```

que define a los atributos de un archivo por las variables miembro:

- ◊ para verificar si hay errores en el archivo;

  ```
  int cookie;
  ```

- ◊ se define al descriptor de archivo;

  ```
  FDTYPE fd_sys;
  ```

- ◊ en caso de que sea el sistema de archivos XFS;

  ```
  #ifdef XFS
  ```

- ◊ variable para habilitar modo directo de E/S, fd_sys servirá como buffer de E/S;

```
int fd_direct;
```

◊ variable bandera, se activa cuando sea lectura;

```
int direct_read;
```

◊ variable bandera, se activa cuando sea escritua;

```
int direct_write;
```

◊ número de datos almacenados;

```
unsigned d_mem;
```

◊ tamaño mínimo del buffer xfer;

```
unsigned d_miniosz;
```

◊ tamaño máximo del buffer xfer;

```
unsigned d_maxiosz;
```

◊ fin de variables para el sistema de archivos XFS;

```
#endif
```

◊ variable puntero a un archivo individual en MPI-IO (en bytes);

```
ADIO_Offset fp_ind;
```

◊ para definir la posición actual del puntero al archivo (en bytes);

```
ADIO_Offset fp_sys_posn;
```

◊ se define la estructura de funciones de E/S a utilizar, se describirá mas adelante;

```
ADIOI_Fns *fns;
```

◊ comunicador de MPI indicando que proceso abrió la comunicación;

```
MPI_Comm comm;
```

◊ nombre del archivo;

```
char *filename;
```

◊ número del tipo de sistema de archivos que se está utilizando, esta numeración es definida en el archivo adio.h;

```
int file_system;
```

◊ modo de acceso al archivo;

```
int access_mode;
```

◊ valor del offset del archivo para MPI-IO;

```
ADIO_Offset disp;
```

◊ tipo de etype a utilizar para MPI-IO;

```
MPI_Datatype etype;
```

◊ tipo de vista de archivo a utilizar para MPI-IO;

```
MPI_Datatype filetype;
```

◊ tamaño del etype en bytes;

```
int etype_size;
```

◊ estructura que contiene valores independientes del sistema de archivos;

```
ADIOI_Hints *hints;
```

◊ valores propios de MPI que no son visibles por la aplicación;

```
MPI_Info info;
```

◊ contador de operaciones colectivas con bloqueo;

```
int split_coll_count;
```

◊ nombre del archivo que comparte su puntero;

```
char *shared_fp_fname;
```

◊ descriptor compartido del archivo;

```
struct ADIOI_FileD *shared_fp_fd;
```

◊ contador de operaciones asíncronas sin bloqueo;

```
int async_count;
```

◊ valor del permiso de acceso al archivo;

```
int perm;
```

◊ valor que define el operacion, atómica o no atómica;

```
int atomicity;
```

◊ variable para los modos en PFS;

```
int iomode;
```

◊ variable para el manejo de errores;

```
MPI_Errhandler err_handler;
};
```

◊ fin de declaración de la estructura ADIOI_FileD.

A.1.2. Estructura ADIOI_Fns_struct

Dependiendo del tipo de sistema de archivos que se está utilizando, por medio de la estructura ADIOI_Fns_struct se hace referencia a las funciones definidas en ADIOI.

```
struct ADIOI_Fns_struct {
```

De esta manera para lograr que múltiples sistemas de archivos puedan ser utilizados por ADIO, la estructrura se define en sus miembros por medio de apuntadores a funciones. A partir de estos apuntadores se hace referencia a las funciones de E/S definidas para un tipo de sistemas de archivos específico:

◊ apuntador a la función ADIOI_xxx_Open;

```
void (*ADIOI_xxx_Open)
        (ADIO_File fd, int *error_code);
```

◊ apuntador a la función ADIOI_xxx_ReadContig;

```
void (*ADIOI_xxx_ReadContig)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_WriteContig;

```
void (*ADIOI_xxx_WriteContig)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_ReadStridedColl;

```
void (*ADIOI_xxx_ReadStridedColl)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_WriteStridedColl;

```
void (*ADIOI_xxx_WriteStridedColl)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_SeekIndividual;

```
ADIO_Offset (*ADIOI_xxx_SeekIndividual)
        (ADIO_File fd, ADIO_Offset offset,
        int whence, int *error_code);
```

◊ apuntador a la función ADIOI_xxx_Fcntl;

```
void (*ADIOI_xxx_Fcntl)
        (ADIO_File fd, int flag,
        ADIO_Fcntl_t *fcntl_struct,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_SetInfo;

```
void (*ADIOI_xxx_SetInfo)
        (ADIO_File fd, MPI_Info users_info,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_ReadStrided;

```
void (*ADIOI_xxx_ReadStrided)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_WriteStrided;

```
void (*ADIOI_xxx_WriteStrided)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_Close;

```
void (*ADIOI_xxx_Close)
        (ADIO_File fd, int *error_code);
```

◊ apuntador a la función ADIOI_xxx_IreadContig;

```
void (*ADIOI_xxx_IreadContig)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Request *request,
        int *error_code);
```

◊ apuntador a la función ADIO_xxx_IwriteContig;

```
void (*ADIOI_xxx_IwriteContig)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Request *request,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_ReadDone;

```
int (*ADIOI_xxx_ReadDone)
        (ADIO_Request *request, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_WriteDone;

```
int (*ADIOI_xxx_WriteDone)
        (ADIO_Request *request, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_ReadComplete;

```
void (*ADIOI_xxx_ReadComplete)
        (ADIO_Request *request, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_WriteComplete;

```
void (*ADIOI_xxx_WriteComplete)
        (ADIO_Request *request, ADIO_Status *status,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_IreadStrided;

```
void (*ADIOI_xxx_IreadStrided)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Request *request,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_IwriteStrided;

```
void (*ADIOI_xxx_IwriteStrided)
        (ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Request *request,
        int *error_code);
```

◊ apuntador a la función ADIOI_xxx_Flush;

```
void (*ADIOI_xxx_Flush)
        (ADIO_File fd, int *error_code);
```

◊ apuntador a la función ADIOI_xxx_Resize;

```
void (*ADIOI_xxx_Resize)
        (ADIO_File fd, ADIO_Offset size, int *error_code);
```

◊ apuntador a la función ADIOI_xxx_Delete;

```
void (*ADIOI_xxx_Delete)
        (char *filename, int *error_code);
};
```

◊ fin de declaración de la estructura ADIOI_Fns_struct.

Por medio de la estructura de apuntadores a función se pueden emplear las diferentes funciones específicas para un sistema de archivos específico. En el caso particular de AFS se define de la siguiente manera:

```
#ifdef AFS
extern struct ADIOI_Fns_struct ADIO_AFS_operations;
#endif
```

donde todos los apuntadores a función definidos en la estructura anterior hacen referencia a las funciones definidas para el sistema de archivos AFS. A continuación se muestran las funciones prototipo más importantes para el sistema de archivos AFS.

A.2. Funciones ADIO-AFS

A.2.1. Función Open

```
/* funcion ADIO AFS OPEN */
/* implementada en el archivo ad_afs_open.c */
void ADIOI_AFS_Open(ADIO_File fd, int *error_code);
```

En la capa alta de ADIO, la función antes mencionada se llama por medio de la función:

```
/* funcion ADIO_Open */
/* implementada en el archivo ad_open.c */
ADIO_File ADIO_Open(MPI_Comm orig_comm,
        MPI_Comm comm, char *filename, int file_system,
        int access_mode, ADIO_Offset disp, MPI_Datatype etype,
        MPI_Datatype filetype, int iomode,
        MPI_Info info, int perm, int *error_code);
```

donde:

- orig_comm especifica el proceso que llamó a la función,
- comm especifica el grupo de procesos participantes,
- filename nombre del archivo,
- file_system tipo de sistema de archivo,
- access_mode tipo de modo de acceso al archivo, que puede ser:
  - ADIO_CREATE si no existe el archivo, se crea,
  - ADIO_RDONLY modo solo lectura,
  - ADIO_RDWR modo lectura y escritura,
  - ADIO_DELETE_ON_CLOSE borrar archivo al cerrarlo (archivos temporales),
  - ADIO_EXCLUSIVE sólo el proceso que llama a la función puede acceder al archivo,
  - ADIO_ATOMIC el sistema de archivos debe garantizar la integridad de las operaciones de E/S.

- disp, etype, filetype se utilizan para el posicionamiento del puntero al archivo, estos están definidos en MPI-IO,
- iomode define los modos de E/S para el sistema de archivos PFS,
- info utilizado por ADIO para que la implementación mejore en su rendimiento,
- perm se definen los permisos de acceso al archivo,
- error_code si ocurre una falla en cualquier operación se retorna el valor contenido en error_code.

A.2.2. Funcion Close

```
/* funcion ADIO AFS CLOSE */
/* implementada en el archivo ad_afs_close.c */
void ADIOI_AFS_Close(ADIO_File fd, int *error_code);
```

La operación close es colectiva, por lo que cualquier proceso perteneciente a un grupo que abra un archivo debe cerrarlo. Dicha función en la capa superior de ADIO se define como:

```
/* funcion ADIO_Close */
/* implementada en el archivo ad_close.c */
void ADIO_Close(ADIO_File fd, int *error_code);
```

donde:

- fd es el descriptor del archivo,
- error_code si ocurre un error se manda un valor contenido en esta variable.

A.2.3. Funciones de E/S Contiguas

En la capa inferior ADIOI se definen las funciones que realizan las operaciones de E/S con datos contiguos en disco y en memoria. A continuación se mencionan estas funciones:

Función ReadContig

```
/* funcion ADIOI AFS ReadContig */
/* implementado en el archivo ad_afs_read.c */
void ADIOI_AFS_ReadContig(ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype,
        int file_ptr_type, ADIO_Offset offset,
        ADIO_Status *status, int *error_code);
```

Inicialmente se cuenta con la función ADIO_ReadContig que es independiente de los procesos, cualquier proceso puede llamar a esta función incluso de forma simultánea; y es de bloqueo debido a que las variables utlizadas en dicha función, en especial los buffers, no pueden ser reutilizados hasta que la función complete su operación.

Esta función ADIO_ReadContig se define en ADIO como:

```
/* funcion ADIO_ReadContig */
/* definida en el archivo adioi.h */
void ADIO_ReadContig(ADIO_File fd,
        void *buf, int count, MPI_Datatype datatype,
        int file_ptr_type, ADIO_Offset offset,
        ADIO_Status *status, int *error_code);
```

donde:

- fd es el descriptor de archivo,
- *buf es la dirección de memoria del buffer donde se almacenan los bytes leidos,
- len N bytes a ser leidos,
- file_ptr_type indica cuando la función debe de ser por offset explícito o por un puntero individual a un archivo,
- offset si file_ptr_type indica el uso de un offset explícito, se toma el valor del offset, en cualquier otro caso se ignora,
- status retoma la información acerca de la operación, ya sea de lectura o escritura.

Función IreadContig

```
/* funcion ADIOI AFS IreadContig */
/* implementado en el archivo ad_afs_iread.c */
void ADIOI_AFS_IreadContig(ADIO_File fd, void *buf,
        int count,
        MPI_Datatype datatype,
        int file_ptr_type,
        ADIO_Offset offset,
        ADIO_Request *request, int *error_code);
```

La función ADIO_IreadContig es la versión de "no bloqueo" de la función ADIO_ReadContig. De esta manera la función puede regresar antes de que complete su operación. Sin embargo, las variables utilizadas en esta función, no pueden ser reutilizadas hasta que no complete la operación de la función. En consecuencia, esta función retorna un valor request que se utiliza para verificar que la operación ya terminó.

En la capa de ADIO se define la función de la siguiente forma:

```
/* funcion ADIO_IreadContig */
/*definida en el archivo adioi.h */
void ADIO_IreadContig(ADIO_File fd, void *buf, int count,
                MPI_Datatype datatype, int file_ptr_type,
                ADIO_Offset offset, ADIO_Request *request,
                int *error_code);
```

donde:

- *request valor de retorno que indica el final de la operación.

Función WriteContig

```
/* funcion ADIOI AFS WriteContig */
/* implementada en el archivo ad_afs_write.c */
void ADIOI_AFS_WriteContig(ADIO_File fd, void *buf, int count,
                        MPI_Datatype datatype, int file_ptr_type,
                        ADIO_Offset offset, ADIO_Status *status,
                        int *error_code);
```

Las características de la función ADIOI_AFS_WriteContig es semejante a la función ADIOI_AFS_ReadContig. Las variables utilizadas en esta función no pueden ser reutilizadas hasta que la función regrese y complete su operación.

En la capa de ADIO esta definida de la siguiente forma:

```
/* funcion ADIO_WriteContig */
/* definida en el archivo adio.h */
void ADIO_WriteContig(ADIO_File fd, void *buf, int count,
                      MPI_Datatype datatype, int file_ptr_type,
                      ADIO_Offset offset, int *bytes_written,
                      int *error_code);
```

donde:

- fd es el descriptor de archivo,
- *buf es la dirección en memoria del buffer para almacenar los bytes que se escribirán,
- count es el número de datos a escribir,
- datatype tipo de dato utilizado para la escritura,
- file_ptr_type indica cuando la función debe de ser por offset explícito o por un puntero individual a un archivo,
- offset si file_ptr_type indica el uso de un offset explícito se toma el valor del offset, en cualquier otro caso se ignora,
- *bytes_written variable que retorna el número de bytes escritos,
- *error_code si ocurre un error se devuelve un valor contenido en esta variable.

Función IwriteContig

```
/* función ADIOI AFS IwriteContig */
/* implementada en el archivo ad_afs_iwrite.c */
void ADIOI_AFS_IwriteContig(ADIO_File fd, void *buf,
        int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Request *request,
        int *error_code);
```

Semejante a la función ADIOI_AFS_IreadContig mencionada anteriormente, la función ADIOI_AFS_IwriteContig es la versión de no bloqueo de la función ADIOI_AFS_WriteContig. En consecuencia, las variables utilizadas

en la función, no pueden ser reutilizadas hasta que la función complete su operación, pero puede regresar para que el proceso siga con su ejecución. De este modo, la función devuelve un valor *request para verificar si la operación ha concluido.

En la capa de ADIO se define la función de la siguiente forma:

```
/* función ADIO_IwriteContig */
/* definida en el archivo adio.h */
void ADIO_IwriteContig(ADIO_File fd, void *buf, int count,
        MPI_Datatype datatype, int file_ptr_type,
        ADIO_Offset offset, ADIO_Request *request,
        int *error_code);
```

donde:

- *request valor de retorno que indica si la operación ya finalizó.

A.2.4. Funciones de E/S No contiguas

En la capa inferior ADIOI se definen las funciones que realizan las operaciones de E/S especificando un acceso no contiguo en una sola llamada. Los patrones de acceso no contiguos pueden ser representados de varias formas, por ello se utilizan los tipos derivados de MPI porque son más generales y estandarizados. A continuación se mencionan estas funciones:

Función ReadStrided

```
/* funcion ADIOI_AFS_ReadStrided */
/* implementada en el archivo ad_afs_read.c */
void ADIOI_AFS_ReadStrided(ADIO_File fd, void *buf,
                int count, MPI_Datatype datatype,
                int file_ptr_type, ADIO_Offset offset,
                 ADIO_Status *status, int *error_code);
```

En principio, la función ADIOI_AFS_ReadStrided es la versión independiente y de bloqueo de la lectura de datos no contiguos. La función se define en la capa superior de ADIO como:

```
/* funcion ADIO_ReadStrided */
/* definida en el archivo adio.h */
void ADIO_ReadStrided(ADIO_File fd, void *buf, int count,
                MPI_Datatype datatype, int file_ptr_type,
```

```
                ADIO_Offset offset, ADIO_Status *status,
                int *error_code);
```

donde:

- ADIO_File es el descriptor de archivo;
- *buf es la dirección en memoria del buffer de lectura, donde se almacenarán los datos,
- count es el número de bytes a almacenar,
- datatype es el tipo de dato utilizado en la escritura,
- file_ptr_type indica cuando la función debe ser por offset explícito o por medio de un puntero individual a un archivo,
- offset si file_ptr_type indica el uso de un offset explícito, se toma el valor del offset o en cualquier otro caso se ignora,
- status si un error ocurre se devuelve un valor contenido en esta variable.

Función IreadStrided

```
/* funcion ADIOI_AFS_IreadStrided */
/* implementada en el archivo ad_afs_iread.c */
void ADIOI_AFS_IreadStrided(ADIO_File fd, void *buf,
                int count, MPI_Datatype datatype,
                int file_ptr_type, ADIO_Offset offset,
                ADIO_Request *request, int *error_code);
```

La función ADIOI_AFS_IreadStrided es la versión de "no bloqueo" de la función ADIOI_AFS_ReadStrided. La función puede retornar antes que la operación de lectura se termine, pero las variables utilizadas en esta función no pueden ser utilizadas hasta que termine la operación. En la capa de ADIO la función se define como:

```
/* funcion ADIO_IreadStrided */
/* definida en el archivo adio.h */
void ADIO_IreadStrided(ADIO_File fd, void *buf, int count,
                MPI_Datatype datatype, int file_ptr_type,
                ADIO_Offset offset, ADIO_Request *request,
                int *error_code);
```

donde:

- *request valor de retorno que indica si la operación finalizó.

A.2.5. Funciones Colectivas E/S

Para establecer operaciones de lectura colectivas, ADIO en su capa inferior proporciona un conjunto de funciones de E/S, donde la función colectiva es utilizada por todos los procesos pertenecientes a un grupo que abre un archivo en común. A continuación se muestran las funciones de E/S colectiva:

Función ReadStridedColl

```
/* funcion ADIO_AFS_ReadStridedColl */
/* implementada en el archivo ad_afs_rdcoll.c */
void ADIOI_AFS_ReadStridedColl(ADIO_File fd, void *buf,
                int count, MPI_Datatype datatype,
                int file_ptr_type, ADIO_Offset offset,
                ADIO_Status *status, int *error_code);
```

La función ADIO_AFS_ReadStridedColl es la versión colectiva de la función ADIO_AFS_ReadStrided donde la función regresa hasta que la operación de lectura termina. En la capa superior de ADIO esta se define como:

```
/* funcion ADIO_ReadStridedColl */
/* definida en el archivo adio.h */
void ADIO_ReadStridedColl(ADIO_File fd, void *buf,
                int count, MPI_Datatype datatype,
                int file_ptr_type, ADIO_Offset offset,
                ADIO_Status *status, int *error_code);
```

Función WriteStridedColl

```
/* funcion ADIOI_AFS_WriteStridedColl */
/* implementada en el archivo ad_afs_wrcoll.c */
void ADIOI_AFS_WriteStridedColl(ADIO_File fd, void *buf,
                int count, MPI_Datatype datatype,
                int file_ptr_type, ADIO_Offset offset,
                ADIO_Status *status, int *error_code);
```

La función ADIOI_AFS_WriteStridedColl es la versión colectiva de la función ADIO_WriteStrided donde la función regresa hasta que la operación de escritura termine. En la capa superior de ADIO se define como:

```
/* funcion ADIO_WriteStridedColl */
/* definida en el archivo adio.h */
void ADIO_WriteStridedColl(ADIO_File fd, void *buf,
                int count, MPI_Datatype datatype,
                int file_ptr_type, ADIO_Offset offset,
                ADIO_Status *status, int *error_code);
```

REFERENCIAS

[1] La Ley de Moore. Página Web, 2005. http://www.intel.com/cd/corporate/techtrends/emea/spa/209840.htm. 3.1

[2] Aad J. Van der Steen and Jack Dongarra. Overview of Recent Supercomputers. Technical report, University of Knoxville, 2003. URL http://www.phys.uu.nl/~steen/web03/overview.html. 1.2.2, 1.2.2

[3] Alan Jay Smith. Disk Cache - Miss Ratio Analysis and Design Considerations. *ACM Transaccions on Computer Systems*, 3(3):161–203, 1985. 2.1.4

[4] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2 edition, 2001. 2

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks Summary and Preliminary Results. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-459-7. URL http://doi.acm.org/10.1145/125826.125925. 6.3

[6] Brian Pawlowsky Chet Juszczak Peter Staubach Carl Smith Diane Lebel David Hitz. NFS Version 3 Desing and Implementation. In *Proceedings of the Summer USENIX Technical Conference*, pages 137–152. USENIX, 1994. 2.2.1

[7] Brocade. *Comparing Storage Area Networks and Network Attached Storage*, 2001.

http://www.brocade.com/san/pdf/whitepapers/SANvsNASWPFINAL3_01_01.pdf. 2.3.2

[8] Cherri M. Pancake. Is Parallelism for You? *IEEE Comput. Sci. Eng.*, 3 (2):18–37, 1996. 3.1

[9] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Comput. Surv.*, 30(2):123–169, 1998. 3.1, 3.3.1

[10] Edward R Zayas. AFS-3 Programmer's Reference: Architectural Overview. Technical report, 1991. 4.1.2

[11] Edward R Zayas. AFS-3 Programmer's Reference: BOS Server Interface. Technical report, 1991. 4.1.2, 4.1.2

[12] Edward R Zayas. AFS-3 Programmer's Reference: File Server/Cache Manager Interface. Technical report, 1991. 4.1.2, 4.1.2, 4.1.3

[13] Edward R Zayas. AFS-3 Programmer's Reference: Volume Server/Volume Location Server Interface. Technical report, 1991. 4.1.2

[14] Enrique Cruz and Eduardo Cabrera. Introducción al Supercómputo. Technical report, DGSCA, Universidad Nacional Autonoma de México, 2002. III Semana de Supercómputo, México D.F. 1.1

[15] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Lectures Notes in Computer Science*, 2150:1+, 2001. URL citeseer.ist.psu.edu/foster01anatomy.html. 1.2.3

[16] W. W. Gropp and E. L. Lusk. A Taxonomy of Programming Models for Symmetric Multiprocessors and SMP Clusters. In *PMMP '95: Proceedings of the Conference on Programming Models for Massively Parallel Computers*, page 2, Washington, DC, USA, 1995. IEEE Computer Society. 3.3

[17] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: A Distributed Personal Computing Environment. *Commun. ACM*, 29(3): 184–201, 1986. 2.2.2, 4.1.2

[18] Jim Mostek, Bill Earl, Steven Levine, Steve Lord, Russell Cattelan, Ken McDonell, Ted Kline, Brian Gaffey, and Rajagopal Ananthanarayanan. Porting the SGI XFS File System to Linux. In *USENIX Annual Technical Conference, FREENIX Track*, 2000. 2.1.1, 2.1.2, 2.1.6

[19] Jonghyun Lee. *Supporting I/O for Remote Visualization of High-Performance Scientific Simulations*. PhD thesis, University of Illinois, Urbana,Illinois, 2003. 3.5.3

[20] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Summer*, pages 238–247, 1986. 2.1.3

[21] Margo I. Seltzer, Keith Bostic, Marshall K. McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *USENIX Winter*, pages 307–326, 1993. 2.1.6

[22] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Trans. Comput. Syst.*, 2 (3):181–197, 1984. 2.1, 2.1.5, 2.1.6

[23] Nils Nieuwejaar and David Kotz. The Galley Parallel File System. In *ICS '96: Proceedings of the 10th International Conference on Supercomputing*, pages 374–381, New York, NY, USA, 1996. ACM Press. 3.5.4

[24] Parkson Wong and Rob F. Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, January 2003. URL `http://www.nas.nasa.gov/News/Techreports/2003/PDF/nas-03-002.pdf`. 6.3

[25] Per Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Commun. ACM*, 21(11):934–941, 1978. 3.2

[26] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO Parallel I/O Interface. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 477–487.

IEEE Computer Society Press and Wiley, New York, NY, 2001. URL citeseer.ist.psu.edu/corbett95overview.html. 5.1

[27] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*, page 59, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-816-9. URL http://doi.acm.org/10.1145/224170.224396. 6

[28] F. Rabhi. A Parallel Programming Methodology based on Paradigms. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 239–251, Amsterdam, 1995. IOS Press. 3.4.1

[29] Rajeev Thakur, William Gropp, and Ewing Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, 1996. 4.1.1

[30] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1999. URL citeseer.ist.psu.edu/article/thakur98data.html. 4.1.3

[31] Rajkumar Buyya and Luis Moura e Silva. Parallel Programming Models and Paradigms. Prentice Hall PTR, NJ, USA, 1999. 3.3.3, 3.4, 3.4.1

[32] Robert B. Ross and Robert Thakur. The Parallel Virtual File System, 2005. URL www.parl.clemson.edu/pvfs/. 2.2.3, 3.5.5

[33] Rolf Rabenseifner and Alice E. Koniges. Effective File-I/O Bandwidth Benchmark. *Lecture Notes in Computer Science*, 1900:1273+, 2001. URL citeseer.ist.psu.edu/article/rabenseifner00effective.html. 5.2.1

[34] L. D. Stevens. The Evolution of Magnetic Storage. *IBM J. RES DEVELOP*, 25(5), September 1981. 2.3

[35] H. Sturgis, J. Mitchell, and J. Israel. Issues in the Design and Use of a Distributed File System. *SIGOPS Oper. Syst. Rev.*, 14(3):55–69, 1980. 2.2.2

[36] Thakur R., Gropp W., and Lusk E. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceeding Frontiers '96:Sixth Symposium on the Frontiers of Massively Parallel Computing*, pages 180–187, 1996. 3.5.2

[37] Transarc Corporation. AFS Programmer's Reference: Authentication Server Interface. Technical report, 1993. 4.1.2, 4.1.2

[38] Stephen C. Tweedie. Journaling the Linux ext2fs Filesystem, 1998. In The 4th Annual Linux Expo Technical Conference. 2.1.6

[39] William Gropp, Steven Huss Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, 1998. 3.5.2