UNIVERSIDAD DE MÁLAGA

Departamento de Arquitectura de Computadores

TESIS DOCTORAL

Optimización de la Entrada Salida mediante librerías y lenguajes paralelos

Rafael Larrosa Jiménez

Noviembre de 2015

Dirigida por:
María Ángeles González Navarro,
Rafael Asenjo Plaza

AUTOR: Rafael Larrosa Jiménez
http://orcid.org/0000-0002-4166-3041

EDITA: Publicaciones y Divulgación Científica.
Universidad de Málaga

Dra. Dña. Mª Ángeles González Navarro.
Profesora Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

Dr. D. Rafael Asenjo Plaza.
Profesor Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada "Optimización de la Entrada Salida mediante librerías y lenguajes paralelos", ha sido realizada por D. Rafael Larrosa Jiménez bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y constituye la Tesis que presenta para optar al grado de Doctor en Ingeniería Informática.

Málaga, Noviembre de 2015

Dra. Dña. Mª Ángeles González Navarro.
Codirectora de la tesis.

Dr. D. Rafael Asenjo Plaza.
Codirector de la tesis.

A mis hijos Rafa y Javi, y a Macarena,
para quienes esta tesis ha durado desde siempre.

Agradecimientos

En primer lugar, me gustaría dar las gracias a mis directores: María Ángeles González Navarro y Rafael Asenjo Plaza, sin sus ánimos y dedicación esta tesis no habría sido posible. A continuación al Departamento de Arquitectura de Computadores y las personas que lo componen, Carmen, quien consigue que para todos sea sencillo lo complicado, ayudando siempre, todos los profesores y personal del departamento, Paco, Juanjo y MªCarmen, siempre haciendo que los problemas tecnológicos desaparezcan. Aunque muchísimos merecen estar aquí, voy a resumirlo en algunos, Guillermo, siempre dispuesto a concretar hasta los más mínimos detalles de todo, desde la fotografía a la música, Nicolás, compañero de fatigas, Mario, Gerardo, Pablo, Eladio, Sergio, María Antonia, Felipe, Siham, Julián, Francisco, Oswaldo, Ujaldon, Andrés, Manuel Sánchez, Juan, Emilio, Oscar, Sonia, Javier, Chema, Inmaculada, Eligius y especialmente a Julio Villalba, por toda su ayuda durante el Proyecto Fin de Carrera.

Agradecer a Darío todas las decisiones discutidas y consensuadas para conseguir que todo funcione de forma óptima, espero que sigamos optimizando sistemas por mucho tiempo. A Gonzalo todas las discusiones semánticas y sintácticas, como sobre cuándo usar números romanos y cuándo no, y a Rocío todo el conocimiento y buen hacer bioinformático, que incluso aparece reflejado en esta tesis. También a Pepi, Diego, Rocío Romero, Ana, Shanti, Rosario, Hicham, Isabel, Pedro y Marina por todas las animadas conversaciones.

A la mesa redonda de relatividad y mecánica cuántica, Rafael Miranda, Francisco Villatoro, José Galindo y Juan Ignacio Ramos, por todas las discusiones y enseñanzas durante tanto tiempo.

A mis amigos de siempre, Caro, Mancebo, Luengo, Caballero, Alberto, José Luis, y tantos otros que mencionar, por los buenos ratos.

A todos mis profesores, desde Infantil hasta la Universidad, por haber sabido siempre inculcar las ganas de aprender.

Quiero agradecer también al equipo de Chapel su buen trabajo, especialmente a Brad

Chamberlain por dirigirlo y estar siempre al tanto, y a Vassily Litvinov por su ayuda y discusiones en todo aquello relacionado con las distribuciones. También a Alberto Sanz, por su ayuda y por hacer más sencilla la programación.

Además quisiera agradecer el acceso a los recursos de computación usados en esta tesis, a la empresa Cray por el acceso al ordenador Crow, al SCBI (SuperComputación y BioInformática) de la Universidad de Málaga por el acceso a Picasso, y al *Oak Ridge Leadership Facility* del *Oak Ridge National Laboratory*, que está mantenido por la Oficina de la Ciencia del Departamento de Energía de los EEUU bajo el contrato número DE-AC05-00OR22725, el acceso a Jaguar, Titan y EOS.

Agradecer también a David Knaak su ayuda para conocer mejor la arquitectura de los Cray, y las optimizaciones en sus implementaciones de la E/S, contestando las preguntas que se le hicieron, que de otra manera no podríamos haber obtenido.

A Maribel y Pepe, por toda su ayuda y cariño. A Silvia y José Luis, por no desesperar ante mis réplicas recursivas, y a mis sobrinos, Luís y Julio, que me han mostrado otra perspectiva de eso que llamamos ”informática”.

Gracias a toda mi familia, y especialmente a mi padre y mi madre, Rafael y Antonia, que me han apoyado y soportado siempre, y que aún ahora casi no se creen que por fin esta tesis llegue a su fin. A mi hermano Antonio, por toda su ayuda durante tanto tiempo y todos los buenos ratos que hemos pasado y que continuaremos pasando, y a Eva, por aportar felicidad.

Y muy especialmente a Rafa y a Javi, que no se han enterado muy bien de a qué dedicaba tanto tiempo, vuestra presencia me llena de felicidad. Gracias por sorprenderme cada día, os quiero.

Y gracias de todo corazón a Macarena, desde que te conocí aquel 8 de enero has llenado de felicidad, cariño y amor mi vida. Te quiero.

Resumen

Uno de los grandes retos de la HPC (*High Performance Computing*) consiste en optimizar el subsistema de Entrada/Salida, (E/S), o I/O (*Input/Output*). Ken Batcher resume este hecho en la siguiente frase: "Un supercomputador es un dispositivo que convierte los problemas limitados por la potencia de cálculo en problemas limitados por la E/S"[1]. En otras palabras, el cuello de botella ya no reside tanto en el procesamiento de los datos como en la disponibilidad de los mismos. Además, este problema se exacerbará con la llegada del *Exascale* y la popularización de las aplicaciones *Big Data*.

En este contexto, esta tesis contribuye a mejorar el rendimiento y la facilidad de uso del subsistema de E/S de los sistemas de supercomputación. Principalmente se proponen dos contribuciones al respecto: i) una interfaz de E/S desarrollada para el lenguaje Chapel que mejora la productividad del programador a la hora de codificar las operaciones de E/S; y ii) una implementación optimizada del almacenamiento de datos de secuencias genéticas.

Con más detalle, la primera contribución estudia y analiza distintas optimizaciones de la E/S en Chapel, al tiempo que provee a los usuarios de una interfaz simple para el acceso paralelo y distribuido a los datos contenidos en ficheros. Por tanto, contribuimos tanto a aumentar la productividad de los desarrolladores, como a que la implementación sea lo más óptima posible.

La segunda contribución también se enmarca dentro de los problemas de E/S, pero en este caso se centra en mejorar el almacenamiento de los datos de secuencias genéticas, incluyendo su compresión, y en permitir un uso eficiente de esos datos por parte de las aplicaciones existentes, permitiendo una recuperación eficiente tanto de forma secuencial como aleatoria. Adicionalmente, proponemos una implementación paralela basada en Chapel.

[1] "*A supercomputer is a device for turning compute-bound problems into I/O-bound problems*."

Índice de Contenido

Índice de figuras

Índice de Tablas

Prefacio

There ain't no such thing as a free lunch

Históricamente hemos estado acostumbrados a que el software vaya cada vez más rápido, aprovechando la mayor velocidad de las CPUs que año tras año han incorporado mejoras tecnológicas y arquitecturales. Los programas simplemente se ejecutaban en menos tiempo sin que hiciera falta ningún esfuerzo para conseguir ese mejor rendimiento.

Pero la "comida gratis" se acabó. En el año 2005 se publicó el artículo "*The Free Lunch Is Over: A fundamental turn toward concurrency in software*" [2] [116], donde se argumenta que esa etapa terminó, dando paso a otra, en la que en vez de aumentar la velocidad de las CPUs los fabricantes se concentran en la concurrencia, es decir, en incrementar el número de cores disponibles. Esto a su vez implica re-implementar las aplicaciones así como desarrollar nuevo software del sistema (compiladores y *runtime*) para explotar los nuevos niveles de concurrencia en las modernas arquitecturas.

Más aún. Con el advenimiento de la era de los sistemas *Exascale* al final de esta década, aparecen nuevos desafíos que han de ser resueltos para poder escalar desde los actuales sistemas *Petascale*. Entre otras cosas, el aumento del rendimiento se espera que se consiga aumentando drásticamente el número de cores por nodo. Esto dará lugar a sistemas que expongan del orden de 1×10^9 *threads* concurrentes. Para poder explotar este elevadísimo nivel de concurrencia se hace necesario el desarrollo de nuevos modelos de programación. Y no sólo eso, además habrá que resolver el hecho de que la memoria y el ancho de banda por core se verán drásticamente reducidos. Puesto que el coste del movimiento de datos, tanto desde el punto de vista del consumo de energía como del rendimiento, no mejorará al ritmo de los *flops*, se exigirá que los algoritmos traten de minimizar el movimiento de datos. Ello requerirá del soporte de nuevas funcionalidades

[2] "La comida gratis se ha acabado: un giro fundamental hacia la concurrencia en el software"

de E/S a todos los niveles: desde el SoC a la memoria, desde la memoria al nodo que sirve la E/S y desde el nodo de E/S al disco. Por todas estas razones, los nuevos modelos de programación que se diseñen no sólo han de ser capaces de gestionar eficientemente billones de elementos de computación concurrentes, sino que además deben de permitir un uso eficiente de la E/S.

El problema de la E/S viene de intentar conjugar tres factores, la velocidad, la capacidad y la fiabilidad. Los dispositivos de almacenamiento (cintas, discos duros rotacionales, discos SSD, y próximamente NVRAMs) tienen unos parámetros dados por la tecnología disponible en cada momento. Para lograr mayores capacidades y velocidades se suelen agrupar varios dispositivos de almacenamiento, formando dispositivos de bloques, lo que proporciona un acceso transparente y unificado a multitud de dispositivos individuales, sumando las capacidades, y hasta cierto punto las velocidades, de todos ellos. Pero este agrupamiento tiene límites tanto físicos (los racks se llenan), como eléctricos (el consumo se vuelve prohibitivo), como a nivel del número de dispositivos que se pueden conectar en un bus sin saturarlo. Para resolver los límites físicos del hardware en velocidad, capacidad y fiabilidad se recurre a soluciones software que aumentan la complejidad de los sistemas. Esto resulta en un incremento significativo en la dificultad de programar productivamente la E/S, lo que a su vez provoca que muchos usuarios terminen optando por el uso de las operaciones POSIX estándar.

Desde hace ya décadas, los sistemas de ficheros se comparten desde múltiples nodos de computación, llegando en algunos casos a situaciones en que el mismo sistema de ficheros sea accedido simultáneamente desde decenas de miles de nodos. Dichas arquitecturas de almacenamiento requieren de redes de comunicaciones que han ido creciendo en complejidad al tiempo que los sistemas han ido escalando. Estos sistemas de ficheros paralelos constituyen una capa intermedia que ocultan las operaciones de acceso concurrente a los dispositivos de E/S, pero en muchos casos requieren que el programador, si busca rendimiento, sea responsable de afinar algunos parámetros no triviales de la arquitectura del sistema de ficheros o incluso que se haga cargo de aspectos como asegurar la consistencia de accesos concurrentes a ficheros compartidos por varios nodos. En este trabajo no abogamos por optimizar manualmente los programas para explotar todo el potencial de los modernos sistemas paralelos de E/S, sino por ofrecer lenguajes, herramientas y librerías que simplifiquen la obtención del máximo rendimiento sin aumentar la complejidad de la programación.

Con todo esto, esta tesis está guiada por dos objetivos principales. Por un lado presentar una interfaz de operaciones de E/S de alta productividad y alto rendimiento, con el objetivo de facilitar la programación y asegurar el rendimiento a la hora de implementar operaciones de E/S paralelas. Por otro lado, y en el contexto del almacenamiento de secuencias genéticas, proponemos una librería de E/S, actualmente en explotación, que implementa un nuevo formato de fichero de almacenamiento de secuencias genómicas.

En los dos casos nos apoyaremos en el lenguaje paralelo Chapel, inicialmente creado en Cray Inc., y que está ganando aceptación a la hora de codificar en poco tiempo aplicaciones paralelas. En esta tesis en particular, contribuimos a la incorporación de operaciones paralelas de E/S en Chapel. Estudiamos y evaluamos diversas estrategias para implementar la E/S paralela, analizando en cada caso las distintas fuentes de paralelismo en el sistema, e incluso cómo interactúan entre sí, presentando resultados obtenidos en un supercomputador de Cray. Como caso de uso, validamos la idoneidad de Chapel para la gestión paralela de la E/S en problemas de almacenamiento de secuenciamiento genómico, para los que hemos desarrollado un nuevo formato más eficiente, lo que constituye la segunda contribución de la tesis. La novedad de este formato es que incluye la compresión de datos y permite una recuperación eficiente de las secuencias tanto de forma secuencial como aleatoria.

La organización de esta memoria es la siguiente. En el capítulo 1 presentamos el estado del arte en los sistemas de E/S, revisamos las distintas soluciones existentes desde el punto de vista hardware, resumimos las características y funcionalidades de los actuales sistemas de ficheros paralelos, presentamos los lenguajes de alta productividad, entre los que se encuentra Chapel así como las limitaciones de los mismos a la hora de realizar operaciones de E/S paralelas. En el capítulo 2 proponemos una interfaz paralela de E/S en Chapel, damos detalles de su implementación y la validamos en una plataforma de supercomputación. En el capítulo 3 proponemos un nuevo formato de almacenamiento de datos obtenidos por secuenciación genética más eficiente que los publicados hasta la fecha, y analizamos su implementación paralela en Chapel. Concluimos en el capítulo 4 sintetizando las principales aportaciones de esta tesis y discutiendo posibles líneas de trabajo futuro.

1 Estudio del estado del arte en los sistemas de E/S

A finales del siglo XX hubo una evolución en los grandes sistemas de computación que los llevó de usar arquitecturas propietarias a usar componentes estándar de la industria, lo que disminuyó el precio de los sistemas, y provocó la perdida de la supremacía de las grandes compañías y países, al permitir que cualquiera pudiera construirlos. Esto hizo que en los distintos países que competían en supercomputación surgieran proyectos para mantener el liderazgo, siendo destacable el proyecto HPCS (*High Productivity Computing System*) [32] del DARPA (*Defense Advanced Research Projects Agency*) que se marcaba como objetivo la creación de sistemas de computación *Petascale* de alta productividad, que fueran económicamente viables. Para conseguirlo DARPA creó un plan de diez años de duración, dividido en distintas etapas, y en el que implicó a empresas y universidades con la idea de tener sistemas que no sólo fueran una evolución de los ya existentes, si no que proporcionaran importantes avances en la productividad, tanto de los sistemas como de los usuarios. Gracias a los avances que se obtuvieron se pudo llegar a los *Petascale systems* en el plazo previsto.

Nos encontramos ahora en el siguiente paso de la evolución, que pretende construir sistemas aún más grandes y más eficientes, los denominados *Exascale systems*. La Comisión Europea, en un esfuerzo por mantener el liderazgo en HPC *High Performance Computing*, puso en marcha a finales de 2013, la denominada *European Technology Platform for High Performance Computing* [41], cuya agenda define las líneas estratégicas del programa FETHPC-H2020 (*Towards Exascale High Performance Computing*). El objetivo de este programa es alcanzar en el año 2020 sistemas de computación capaces de pasar de los sistemas de 10^{15} flops/seg (*Petaflops*) a 10^{18} flops/seg (*Exaflops*), con un consumo energético de 20 MW (lo que significa reducir por 100 el consumo de los sistemas actuales). La tabla 1.1 resume los principales requerimientos que se espera que

satisfaga un sistema *Exascale* y cuáles son los valores que un sistema *Petascale* consigue actualmente.

Atributo	*Petascale*	*Exascale*
Peak flops	10×10^{15} flops	1×10^{18} flops
Potencia	15-50 MW	20 MW
Memoria	0.5 Petabytes	50 Petabytes
GB RAM (core)	1-2	0.1-0.5
Rendimiento (nodo)	1.25×10^{11} flops	2×10^{12} flops
Ancho banda (nodo)	2.5×10^{10} bytes/s	1×10^{12} flops
Concurrencia (nodo)	12 cores	1000 cores
Número de nodos	2×10^{4} cores	1×10^{6} cores
Concurrencia total	2.25×10^{5} threads	1×10^{9} threads

Tabla 1.1: Proyección sobre los principales atributos de un sistema *Exascale*.

Como vemos, no se tratará simplemente de multiplicar por 1000 el rendimiento del sistema Pestascale. Entre otras cosas, el aumento del rendimiento se espera que se consiga aumentando drásticamente el número de cores por nodo. Esto dará lugar a sistemas que expongan del orden de 1×10^9 threads concurrentes, para lo que se hace necesario el desarrollo de nuevos modelos de programación capaces de gestionar estos elevadísimos niveles de concurrencia, puesto que la tecnología de compilación no será capaz de ocultar todas las posibles fuentes de concurrencia de las aplicaciones. Más aún, habrá que resolver el hecho de que la memoria y el ancho de banda por core se verán drásticamente reducidos. Puesto que el coste del movimiento de datos, tanto desde el punto de vista del consumo de energía como del rendimiento, no mejorará al ritmo de los *flops*, se exigirá que los algoritmos traten de minimizar el movimiento de datos. Ello requerirá del soporte de nuevas funcionalidades de E/S a todos los niveles: desde el SoC a la memoria, desde la memoria al nodo que sirve la E/S y desde el nodo de E/S al disco. Por todas estas razones, los nuevos modelos de programación que se diseñen no sólo han de ser capaces de gestionar eficientemente billones de elementos de computación concurrentes, sino que además deben de permitir un uso eficiente de la E/S.

Por lo que vemos, las operaciones de E/S representan uno de los grandes retos aún no resueltos en HPC, puesto que muchas de las aplicaciones usadas en las grandes plataformas de computación suelen usar grandes volúmenes de datos de forma *intensiva*. Por ejemplo, las denominadas aplicaciones *data-intensive* o de *Big-data*. Además, es usual que esas aplicaciones requieran hacer *checkpoints* [90], [94] de los resultados intermedios para almacenarlos temporalmente en el sistema de almacenamiento, y de esta manera hacer frente a los problemas de fiabilidad que puedan aparecer en el sistema.

Los sistemas modernos de E/S en el ámbito HPC están compuestos de varias capas. En el nivel más bajo, están primero los sistemas hardware de almacenamiento. En una

capa superior tenemos los sistemas de ficheros paralelos, seguidos a un nivel más alto de las librerías de acceso a datos, y finalmente encontramos los tipos de datos y los lenguajes de programación de alta productividad. En las siguientes secciones trataremos el estado del arte en estos distintos niveles del *stack* de E/S.

1.1. Introducción a la aceleración de la E/S

Una opción hardware para acelerar la E/S consiste en instalar un sistema de almacenamiento basado en discos rápidos, bien discos de hasta 15000 rpm o bien usar discos SSD. También se puede mejorar el sistema de array de discos para que proporcione mayor ancho de banda. Otro factor hardware que influye es la red que se use para acceder al sistema de almacenamiento, que se puede cambiar por una red más rápida y/o una red dedicada al tráfico de E/S. Otra opción hardware consiste en aumentar el número de discos y/o el número de servidores de disco.

En cualquier caso todas las mejoras anteriores implican el invertir más capital en el sistema y no es fácilmente justificable si el software no está preparado. Por tanto la primera recomendación es optimizar el software que realiza la operaciones de E/S, aunque también como programador de aplicaciones se pueden aplicar algunas estrategias básicas de aceleración de la E/S. Por ejemplo, cuando se diseña un algoritmo en el que hay que realizar alguna operación de E/S se puede ser consciente de que el sistema operativo lee por bloques, y además suele realizar una precarga (*prefetch*), por lo que en realidad se pueden leer varios bloques de datos en cada acceso. Si el programador es consciente de este detalle puede explotarlo cuando invoca la operación de E/S. Así, si desde la aplicación aseguramos que los accesos son consecutivos, minimizaremos el número de llamadas al sistema de ficheros. Además, si usamos buffers en la aplicación para traer bloques del sistema de ficheros, y desde ahí consumimos los datos en nuestra aplicación, reduciremos significativamente los tiempos.

Otro ejemplo para acelerar el acceso a los datos puede ser el comprimirlos. Esto ofrece varias ventajas: las transferencias serán más rápidas y los datos ocuparán menos espacio, tanto en el sistema de almacenamiento cómo en las distintas cachés del sistema. El único inconveniente es que se tarde más en transferir y descomprimir que sólo en transferir los datos sin comprimir. Esto puede ocurrir si la red de acceso al sistema de almacenamiento es lo suficientemente rápida.

1.1.1. Hardware de almacenamiento

Los criterios que se usan para comparar las distintas soluciones de almacenamiento [74] son: i) la densidad; ii) la eficiencia energética; iii) los tiempos de escritura y lectura; iv) la resistencia; v) el tiempo de retención.

La densidad [51] es el factor más importante, ya que está directamente relacionada con el coste. Se puede asumir que el coste es directamente proporcional a la densidad siempre que el coste del procesamiento y empaquetamiento de las obleas no varíe drásticamente [74]. El siguiente parámetro es el consumo. En dispositivos portátiles esto es obvio por la necesidad de optimizar la energía almacenada en la batería. Históricamente no se había considerado como crítico el consumo en los CPDs (Centro de Proceso de Datos), ya que se iba buscando el rendimiento. Sin embargo, en los sistemas de supercomputación futuros se estima que tendrán del orden de 100 Petabytes de memoria principal, que si se implementaran con la tecnología actual DDR3 DRAM consumirían 52 MW de electricidad, lo que, por ejemplo, supera con creces los 20 MW que se espera consuman los futuros sistemas Exascale [91], según previsiones del programa FETHPC del H2020. El tiempo de acceso de un mismo dispositivo puede ser muy distinto entre lectura y escritura, y además hay que tener en cuenta la posible variabilidad que puede existir en los accesos del mismo tipo a un dispositivo. Por ejemplo al escribir en memorias SSD (Solid State Disk), el dispositivo puede estar en un ciclo de recolección de basura y retrasar en varios ordenes de magnitud la escritura [114]. La resistencia hace referencia al número de veces que se puede sobreescribir una posición dada de la memoria, y la retención es el tiempo que permanecerá estable la información, sin que exista una degradación de los bits.

Podemos comparar la memoria primaria y la secundaria usando para ello las características que acabamos de enumerar, encontrando como factor más importante que la densidad de la memoria secundaria es varios ordenes de magnitud superior a la de la memoria primaria (y por tanto el coste por bit mucho menor). Desde el punto de vista de la eficiencia energética no serían comparables, ya que un sistema secundario se puede apagar y no se pierde la información, pasando a estar en un estado en el que no consumiría energía, aunque esto no es lo más común. La velocidad es el gran factor de ventaja del almacenamiento primario, ya que es varios ordenes de magnitud más rápida que el secundario. En cuanto a la resistencia, depende de la tecnología empleada en cada caso, puesto que puede ser muy variable. Por ejemplo, en el almacenamiento secundario nos encontramos con discos duros rotacionales clásicos, que suelen ser relativamente resistentes, pero también con discos SSD con tecnologías con muy distinta resistencia. Así, SLC (*Single Level Cell*), MLC (*Multi Level Cell*) o TLC (*Third Level Cell*), varían la resistencia entre 100000 ciclos de escritura los mejores SLC a unos 3000 ciclos los peores TLC [25]. Y sobre el tiempo de retención, no llega a ser de un segundo en el almacena-

miento primario (DRAM) [4], por lo que no se puede usar para tener un almacenamiento persistente.

En [63] podemos encontrar un buen resumen de los distintos medios de almacenamiento que se han usado en toda la historia, con el tiempo en que han estado en uso, incluyendo los almacenamientos primarios, secundarios y terciarios.

1.1.2. La capa de almacenamiento persistente, NVRAM

En un futuro cercano, será cada vez más común tener una capa de almacenamiento persistente entre los niveles de memoria primaria y secundaria, compuesto de NVRAM (*Non-Volatile RAM*). La idea es tener una cantidad de memoria que almacene los datos de forma permanente, que se pueda direccionar por bytes (o por líneas de caché, siendo estrictos). En el apéndice A.1.3 podemos encontrar un ejemplo de un sistema que llevará este nivel de almacenamiento.

La forma de aprovechar las nuevas características que proporciona este nivel es un problema aún no resuelto. Una de las opciones más comunes consiste en proponer su uso como un buffer de ráfagas (*burst buffer*) del sistema de fichero distribuido que se use, usualmente Lustre. En [83] se hace un estudio teórico de como funcionaría esa capa, y las mejoras en el rendimiento que se obtendrían, ya que generalmente las aplicaciones escriben por ráfagas, lo que provoca una saturación tanto de los discos como de las redes durante esas etapas de escritura. Con los *bursts buffers* se almacenan esas ráfagas de forma temporal en los buffers NVRAM dispuestos para ellos, y al realizar la transferencia en un intervalo de tiempo mayor se logra no saturar los discos y las redes durante la transferencia.

Otra opción que se plantea es usar la memoria NVRAM creando un sistema de ficheros convencional sobre esta memoria [84]. Sin embargo presenta el problema de que tiene que mantener la coherencia tras reiniciar el ordenador, ya que los datos almacenados lo seguirán estando. Esto puede crear problemas de consistencia y también de seguridad [103]. otra opción es usar esta memoria como un repositorio de los datos del *checkpointing* que vaya realizando la aplicación.

1.1.3. El sistema de almacenamiento secundario

El objetivo del almacenamiento secundario es almacenar los datos que vamos a querer tener disponibles en un futuro, haciendo que estén accesibles desde los sitios donde se necesiten. Esto requiere el uso de distintas arquitecturas de acceso, que pueden ser: i) DAS (*Direct Attached Storage*); ii) NAS (*Network Attached Storage*) iii) SAN (*Storage*

Area Network).

La conexión directa (DAS), como su nombre indica, está caracterizada por conectar, a través de un bus, el almacenamiento al *host*, sin usar ningún elemento de red (*hub*, *router*, *switch*, etc.). Pero si tenemos el sistema de almacenamiento conectado a un sólo nodo sólo ese nodo podrá acceder al almacenamiento. Para solucionar ese problema aparecen las NAS [55], que permiten a otros nodos el acceso a los datos usando protocolos cómo NFS [104] o samba (CIFS) [95] Para ello hay que conectar el almacenamiento, mediante una DAS o una SAN, a un servidor, y este proveerá a los clientes de una interfaz de acceso a los datos a través de la red, permitiendo a los clientes un acceso sencillo a los ficheros y a los datos que contienen. En resumen, lo que provee una NAS es un sistema de ficheros.

Con el tiempo se crearon en las organizaciones multitud de islas de servidores con almacenamiento local, por lo que surgió la necesidad de consolidar el almacenamiento. Para conseguir esto se desarrollaron las SAN [117], permitiendo la conexión entre un sistema de almacenamiento y varios servidores, ya que una SAN provee dispositivos de bloques, y no sistemas de ficheros. Se suele usar el protocolo SCSI sobre determinados medios físicos (InfiniBand (IB), ethernet, FC,...). El hecho de que un sistema de almacenamiento proporcione dispositivos de bloque a los distintos servidores supone varias ventajas: i) se optimizan los recursos hardware; ii) se unifica el almacenamiento secundario para todos los servidores; iii) se reduce el espacio sobrante; iv) reduce el coste; y v) facilita la administración. Posteriormente, si hay necesidad de compartir datos con clientes, se puede montar una NAS, creando un sistema híbrido. Pero si muchos clientes necesitan acceder a grandes cantidades de datos el servidor NAS se convierte en el cuello de botella. La solución obvia es conectar los clientes directamente a la SAN, pero eso tiene el problema de que la SAN sólo provee acceso a nivel de bloque, no existe ninguna capa que provea una interfaz de sistema de ficheros en una SAN.

Para tener un acceso coherente a esos sistemas de bloque aparecieron los sistemas de fichero de disco compartido (*shared-disk file systems*), que se encargan de proporcionar un acceso coherente a los datos, usando servidores de metadatos, y un acceso directo desde los clientes a los bloques que contienen los datos de los ficheros. De esa forma se evita que los datos tengan que pasar por un servidor antes de llegar a su destino, yendo directamente de los dispositivos de E/S al nodo que los solicitó. Eso significa que los clientes necesitan un acceso directo, a nivel de bloque, a la SAN. Esto se puede conseguir uniéndolos a través de alguna red de interconexión, generalmente usando *fibre channel*. Esta solución no es factible en el ámbito de HPC (*High Performance Computing*) donde podemos tener miles de nodos necesitando acceder a los datos y sin posibilidad de acceso directo a la SAN, ya que a pesar de tener sus propias redes de interconexión de alta velocidad, estas no suelen ser compatibles con las redes que usan las SAN para dar acceso a los bloques. La solución actual al problema de acceso a los datos en entornos

HPC se basa en los sistemas de ficheros distribuidos y paralelos (*distributed parallel file systems*). Veremos varios ejemplos en la sección 1.2, e implementaciones concretas a partir de la sección 1.4.2. En esta tesis se ha usado el sistema de ficheros Lustre [12] por ser el más usado en sistemas HPC [109], como explicamos en la sección 1.2.4.

En todas las soluciones mencionadas el acceso a los ficheros se realiza a través de una interfaz POSIX [127], lo que permite que las aplicaciones sean independientes del hardware, del sistema de ficheros y del sistema operativo usados, consiguiendo una mayor productividad del programador al permitir el uso de los programas en una mayor variedad de entornos.

1.1.4. El sistema de almacenamiento terciario

Aunque en este trabajo no se ha usado ningún sistema de almacenamiento terciario, por completitud se realiza aquí una introducción a su uso y funcionamiento, ya que se usa en muchos sistemas HPC para el almacenamiento masivo de datos.

En el almacenamiento terciario podemos mencionar los medios (cintas, discos blu-ray,...), los dispositivos de acceso a los medios (unidades de cinta, unidades blu-ray,...), y finalmente suele existir un mecanismo de intercambio automático. Así podemos encontrar que múltiples medios están depositados en casillas dentro de un armario, junto a uno o más dispositivos de acceso, y uno o más brazos robóticos se encargan del movimiento de los medios dentro del armario, entre las distintas posiciones que puede tomar. Estos armarios suelen tener una o más casillas que ejercen de interfaz con el mundo exterior, de forma que se pueden extraer e incorporar cintas sin necesidad de interrumpir su operación. Encontramos que la característica que define el almacenamiento terciario es que el acceso a los medios se realiza usando un brazo robótico, que se encargará de mover el medio de almacenamiento de datos, cinta o disco, a una unidad que se encargará de acceder a los datos que contiene, lo que aumenta la latencia de acceso.

Una vez insertado el disco o la cinta en la unidad de acceso, hay que moverlo hasta que el cabezal de acceso a los datos está sobre la posición en la que están almacenados los datos, accederlos, ponerlo en un estado estable (en el caso de las cintas rebobinarlas) sacar el medio de la unidad y almacenarlo en un *slot* libre de la librería robótica. Todos esos pasos influyen en el rendimiento (ancho de banda y la latencia) y hacen complicado el realizar una previsión de los tiempos de acceso.

Aunque no vaya a desaparecer el almacenamiento terciario, lo que si está cambiando es el propósito con el que se emplea. Conforme los discos mejoran y se hacen más baratos es más lógico emplearlos para hacer *backups*, pero las cintas aún tienen sus ventajas, como el poder mover con facilidad el backup fuera de las instalaciones y el menor coste,

dependiendo del volumen de los datos que se necesite almacenar, y de los requisitos de recuperación de esos datos. En el futuro se estima que las cintas se usen como segunda o tercera línea de actuación en los backups. Los discos ópticos tampoco se pueden desechar como opción de almacenamiento, ya que si el volumen de datos a acceder no es muy grande son mucho más rápidos que las cintas, al permitir accesos aleatorios. Así podemos encontrar que Facebook está usando discos bluray para almacenar imágenes que se acceden poco, creando lo que llaman un *cold storage* (almacenamiento frío) [130], ya que el 97 % del contenido recibe sólo un 29 % de las peticiones.

1.2. Sistemas de ficheros paralelos

Los sistemas de ficheros paralelos convencionales suelen usar varios servidores con arrays de discos locales para servir los ficheros, pudiendo un fichero almacenarse de una de estas tres formas: i) en un sólo servidor, ii) replicado en varios de forma íntegra, o iii) repartido entre varios, de forma que cada uno de los servidores contiene un trozo. Se podría pensar que si cada fichero se almacena en un sólo servidor, el sistema está desaprovechando recursos, ya que no estaría consiguiendo aprovechar todo el ancho de banda existente. Pero hay que tener en cuenta que un sistema HPC en explotación puede estar orientado a permitir la ejecución concurrente de muchas aplicaciones, por lo que se producen multitud de operaciones de E/S simultáneamente a distintos ficheros. En estos escenarios se suele suponer un reparto homogéneo de peticiones entre los distintos servidores de E/S , por lo que si los ficheros son pequeños, no distribuirlos es buena una opción. Otra opción, en caso de ficheros de gran tamaño, pasa por distribuirlos uniformemente entre los servidores de E/S. Sin embargo, si tenemos muchos servidores sería ineficiente dividir los ficheros entre todos, ya que se podría provocar una sobrecarga de peticiones hacia todos los servidores, por lo que en la práctica el número de servidores que se utiliza para distribuir un fichero suele ser relativamente bajo. En general, los sistemas de ficheros paralelos se aprovechan de que existen multitud de aplicaciones en ejecución simultáneamente para, además de intentar optimizar el acceso a cada uno de los ficheros, optimizar el *throughput* total del sistema.

Vamos a ver en las siguientes secciones la arquitectura y características principales de los sistemas de ficheros paralelos que constituyen el estado del arte en los grandes sistemas HPC en al actualidad: GPFS (*General Parallel File System*), Ceph, EOS y Lustre, este último el más popular en la mayoría de los grandes sistemas HPC. Todos ellos son distribuidos y permiten el acceso a los datos desde distintos servidores en paralelo. Entre la lista de sistemas de ficheros paralelos a analizar no hemos incluido algunos sistemas de ficheros muy populares actualmente, como el GFS (*Google File System*) o el HDFS (*Hadoop Distributed File System*) debido a que son sistemas de ficheros hechos a medi-

da para su uso en entornos concretos. En particular, están diseñados para ser usados en aplicaciones que se basan en el paradigma *map-reduce*, y por tanto no son de propósito general. Uno de los objetivos de estos sistemas de ficheros es acercar los cálculos a los datos en vez de los datos a los cálculos, lo que puede ser una estrategia prometedora en el ámbito de aplicaciones *data-intensive*.

1.2.1. GPFS

GPFS [107] es un sistema de ficheros paralelo y distribuido para clusters desarrollado por IBM. Intenta emular el comportamiento de un sistema de ficheros POSIX, haciendo creer a cada nodo del cluster que tiene un sistema de ficheros local. El nombre ha cambiado recientemente de GPFS a *Spectrum Scale*.

Un sistema de ficheros GPFS está formado por un conjunto de arrays de discos que contienen los datos y los metadatos de los sistemas de ficheros. Cada sistema de ficheros se puede acceder desde todos los nodos del cluster usando el interfaz estándar POSIX. GPFS se encarga de mantener la coherencia y consistencia de múltiples accesos desde distintos nodos, usando bloqueos a nivel de byte, administración distribuida de bloqueos y cuadernos de bitácora (*journaling*). Gracias a todo ello no hace falta modificar las aplicaciones POSIX para poder ejecutarlas en un sistema que use GPFS. Además del interfaz POSIX, GPFS añade un conjunto de interfaces para aumentar la funcionalidad que ofrece a las aplicaciones, con el objetivo de que desde la aplicación se puedan indicar pistas sobre el patrón de acceso a los ficheros. Esa información se usará para optimizar el acceso a los discos, las precargas o el uso de las caches [100]. Además GPFS también es capaz de detectar algunos patrones de acceso en tiempo de ejecución. Por ejemplo, puede detectar un patrón de acceso secuencial, secuencial inverso y aleatorio.

Cuando se crea un sistema de ficheros GPFS se le asignan conjuntos de dispositivos de bloques, llamados NSD (*Network Shared Disks*) en la terminología GPFS. Cada NSD puede ser accedido por uno o más nodos del cluster GPFS, de forma que se puede tener redundancia y en caso de que caiga un servidor otro se encargue de ese NSD. El problema de la consistencia debido el acceso simultaneo desde distintos nodos a los datos y los metadatos se soluciona usando un sistema de *tokens* administrados de forma distribuida que actúan como cerrojos (*locks*), y se encargan de coordinar los accesos a los NSD. La responsabilidad de la administración de los *tokens* se asigna dinámicamente a uno o más nodos del cluster GPFS, esto permite una mayor escalabilidad cuando se están usando muchos ficheros en momentos de mucha carga de trabajo.

Existen tres formas básicas de configurar un cluster GPFS [45]: i) usando discos compartidos entre todos los nodos; ii) usando servidores de E/S; iii) mezclando las dos anteriores.

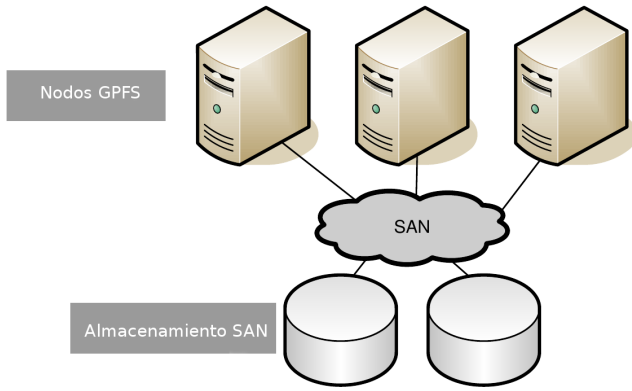


Figura 1.1: Configuración básica de GPFS usando discos compartidos a través de una SAN.

La configuración más básica para tener un cluster GPFS es usando discos compartidos, donde el almacenamiento está conectado a través de una SAN a todas las máquinas del cluster, como se puede ver en la figura 1.1. Esto significa que los dispositivos de bloques se pueden acceder directamente desde todos los nodos usando un protocolo como SCSI o similar, a través de una red como fibre channel, infiniband, u otras, y usando elementos de interconexión como switches. Esta configuración, usando discos compartidos, se suele usar para dar servicio de NFS a través de cNFS (clustered NFS), de forma que los nodos GPFS son los que sirven en paralelo el protocolo NFS. El problema de esta configuración es que conforme los requerimientos de capacidad y procesamiento crecen, las tecnologías de conexión pueden no ser las apropiadas para un cluster con muchos nodos, por lo que en sistemas muy grandes se suele usar una de las otras dos configuraciones.

La configuración más usual en sistemas HPC consiste en usar servidores GPFS de E/S que comparten los discos a través de una SAN, y que sirven los sistemas de ficheros al resto de nodos como se ve en la figura 1.2. El acceso al sistema de ficheros desde los nodos se realiza usando un protocolo llamado NSD (*Network Shared Disk*), que provee una interfaz a nivel de bloques sobre las redes que haya disponibles, por ejemplo usando TCP/IP con ethernet o *verbs* con infiniband. En cualquier caso, para las aplicaciones que usan el interfaz GPFS no hay ninguna diferencia, salvo la de rendimiento, entre acceder los datos directamente por la SAN, o usando servidores de E/S (llamados *NSD servers* en la terminología de GPFS). La decisión de cuántos nodos configurar como servidores de E/S está basada en los requerimientos de rendimiento, la arquitectura de la red y las prestaciones del sistema de almacenamiento.

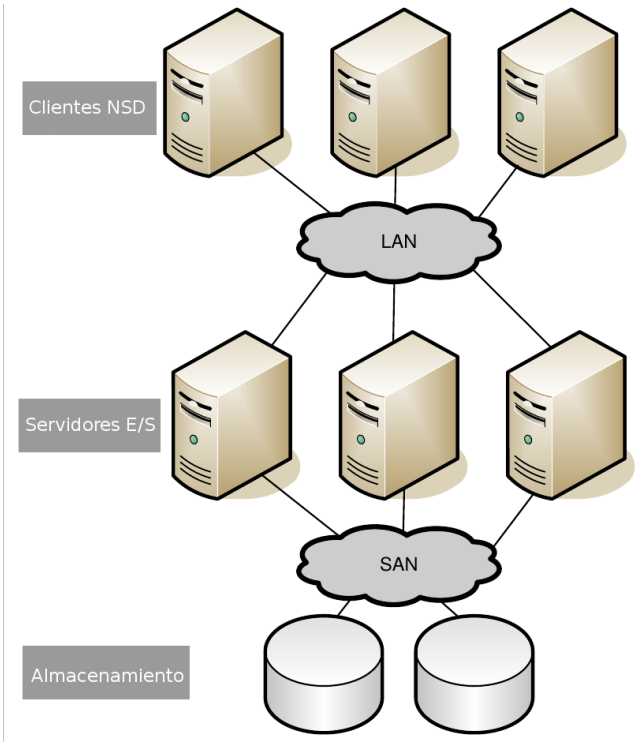


Figura 1.2: Configuración GPFS usando servidores de E/S, que sirven los datos a los clientes, y a su vez acceden a los bloques contenidos en discos compartidos a través de una SAN.

Otra configuración posible es combinar las dos anteriores, algunos nodos se usarán como servidores de E/S y otros accederán directamente a los datos. Esto es sencillo de configurar, ya que a los nodos se les puede indicar múltiples caminos para el acceso a los sistemas de ficheros, usando por defecto siempre el camino más rápido. Si un nodo tiene un HBA (*Host Bus Adapter*) para acceder a la SAN, lo usará, y si no, o si se corta la conexión por el HBA, entonces accederá a través de los servidores de E/S, usando otra de las redes disponibles, como ethernet. Usando esta configuración de acceso mixto, en el que algunos nodos acceden de forma directa y otros no, se pueden configurar de forma directa los grandes consumidores o productores de datos, por ejemplo los nodos de backups, y el resto configurarlos para que accedan a través de los servidores de NSD.

La decisión de usar una conexión directa a la SAN, usar la solución con servidores de E/S, o la solución mixta, es una decisión tanto de rendimiento como económica. El conectar todos los nodos del sistema mediante una SAN exige poner una red de interco-

nexión con una tecnología apropiada para obtener un buen rendimiento, pero dependiendo de para que se vaya a usar, eso no será rentable, ya que el cuello de botella no será la red de interconexión. Si además se consiguen paralelizar los accesos desde distintos nodos la necesidad de tener una red de muy alta velocidad entre los nodos será menor aún. Así, cuando hay muchos clientes, será más rentable el usar servidores de E/S para el acceso al sistema de ficheros GPFS.

1.2.2. Ceph

Ceph [136] es un sistema de ficheros paralelo que sigue a POSIX de forma relajada. Hay dos características en las que se desvía del estándar: en la medida del espacio ocupado por ficheros distribuidos y en la atomicidad de las escrituras desde distintos nodos. El aspecto diferenciador con otros sistemas de ficheros paralelos, desde el punto de vista de la arquitectura, es que en Ceph se separan los datos y metadatos, sustituyendo el interfaz tradicional del sistema de bloques con uno en el que los clientes acceden a los objetos por su nombre, dejando que los dispositivos realicen la decisión de cómo distribuir los bloques de datos a bajo nivel. Así, los clientes comunican las operaciones de metadatos (como *open, rename,..*) al MDS (*MetaData Server*) mientras las operaciones con datos (escrituras y lecturas) se realizan directamente en los OSD (*Object Storage Device*).

El punto fuerte de Ceph es la eliminación de las tablas de asignación de ficheros (*file allocation table, fat*), reemplazándolas con una función CRUSH (*Controlled Replication Under Scalable Hashing*) que se encarga de distribuir los datos entre los OSD de una forma semialeatoria. Para realizar esa distribución la función *CRUSH* [137] se comporta de forma parecida a un *hash*, pero un hash normal no sería efectivo debido a la naturaleza dinámica del almacenamiento, en el que se van añadiendo y desapareciendo recursos conforme las necesidades cambian, o se van rompiendo. La idea central de Ceph es distribuir los datos para realizar un uso eficiente tanto del almacenamiento como de los anchos de banda. Esto se consigue haciendo que los datos que se van incorporando al sistema se vayan distribuyendo de forma aleatoria entre todos los nodos, cuando se incorporan nuevos dispositivos o nodos, se migran subconjuntos aleatorios a los nuevos dispositivos y cuando desaparecen dispositivos se redistribuyen de forma uniforme los datos que contenían. El objetivo de todo esto es evitar los desbalanceos y las asimetrías de cargas (que los datos más accedidos no estén bien repartidos entre todos los dispositivos) entre los dispositivos existentes. Para conseguir estos objetivos, Ceph primero mapea los objetos en grupos de posicionamiento (*Placement Groups* o *PGs*) usando una función de hash simple junto a una máscara de bits que controla el número de *PGs*. Los *PGs* se asignan a los OSD usando la función CRUSH, que mapea cada *PGs* a una lista ordenada de OSD en los que se almacenarán las réplicas, usando una función de distribución semialeatoria. Para localizar un objeto, la función CRUSH sólo requie-

re el *PGs* y el mapa de clusters OSD *cluster map*, que es una descripción compacta y jerárquica de los dispositivos que forman el cluster de almacenamiento. Esto tiene dos ventajas principales, una es que está completamente distribuido, cualquier elemento del sistema puede calcular de forma independiente la localización de cualquier objeto, y la otra ventaja es que el mapa es muy raro que se actualice, con lo que se consigue eliminar cualquier intercambio de metadatos relacionados con esta distribución. Un problema es mantener la función CRUSH aunque cambien los mapas. La solución es etiquetar las distintas versiones de los mapas de un sistema, e indicar cuál es el mapa a usar en cada caso.

Una de las características más importantes es la replicación de los datos, ya que se intenta abaratar los costes no usando RAIDs de discos, u otros complicados y caros sistemas de redundancia. La solución pasa por escribir cada dato en más de un sitio, y cuando el primario falla se intenta acceder a los replicados. El OSD primario solicita realizar copias en otros OSDs, y éstas no se dan por realizadas hasta que todas las réplicas han sido procesadas [136].

Desde el punto de vista de la existencia de problemas en un OSD, se tienen dos variables en Ceph [135] que determinan la vitalidad del OSD, la accesibilidad y la asignación de datos. Un OSD que no esté accesible se marca como *down*, y sus responsabilidades pasan al siguiente OSD del grupo de posicionamiento (*PG*) al que pertenece. Si no se recupera en poco tiempo se marca como *out*, se queda fuera de la distribución de datos, y otro OSD se une a cada uno de los grupos de posicionamiento *PG* a los que pertenecía el OSD caído, y replica su contenido. Los clientes que tenían operaciones pendientes con ese OSD simplemente las reenvían al nuevo OSD primario.

1.2.3. EOS

El CERN (*Conseil Européen pour la Recherche Nucléaire*) es uno de los mayores generadores de datos del planeta, por lo que tiene unas necesidades de almacenamiento especiales, que les ha llevado a buscar soluciones a medida [39]. Experimentos como *Atlas* o *Alice* generan Petabytes de datos que tienen que ser accedidos y procesados por miles de físicos de todo el planeta. Los datos están almacenados en dos infraestructuras, CASTOR y EOS [39]. CASTOR se encarga del almacenamiento en cinta, y EOS del almacenamiento en discos, siendo ambos considerados sistemas de administración de contenidos (*Content Management System*, CMS).

EOS [2] es un sistema de ficheros paralelo desarrollado y mantenido principalmente por el CERN desde 2010. Sigue una semántica POSIX simplificada, con lo que no es POSIX, sólo soporta lo que les interesa, que es almacenar y procesar los datos que generan los instrumentos del CERN [1], que suelen generar ficheros de gran tamaño. Para

el acceso a esos datos se usan diferentes interfaces, como XrootD [34], https, etc. Desde 2013 XrootD se usa como la forma principal de acceso a los datos de *Atlas* [53].

La arquitectura de EOS está basada en tres elementos [2]: i) Un administrador (*MGM*) que se ejecuta en un nodo, pudiendo tener otro de *fail-over* ; ii) una cola de mensajes *Message Queue, MQ*, que coordina los mensajes asíncronos del sistema, y que se suele ejecutar en el *MGM*; iii) un componente de almacenamiento de ficheros *File Storage Component, FSC*, que almacena los datos y los transfiere desde y hacia los clientes, y se ejecuta en los servidores de ficheros. Así, un cliente le pide al *MGM* una operación sobre un fichero, y éste le redirige al *FST* adecuado que se encargará de la operación.

1.2.4. Lustre

La primera versión de Lustre [12] se presentó en diciembre del año 2003, y a partir de ese momento ha ido evolucionando, apoyado por empresas, centros de investigación y universidades. Lustre ha evolucionado desde entonces, con una mejora continua del rendimiento, de la funcionalidad, y de la fiabilidad. Desde el principio, Lustre es un sistema de ficheros paralelo y de código fuente abierto (*open source*). Lustre es usado en aproximadamente la mitad de los 100 ordenadores más rápidos de forma continua desde 2005, y por la mayoría de los sistemas HPC del planeta [109].

Lustre es escalable, y un único sistema Lustre puede dar servicio a varios clusters independientes con decenas de miles de clientes y decenas de Petabytes de almacenamiento distribuido en miles de servidores. Ese es el caso, por ejemplo, de uno de los sistemas de E/S que hemos usado en este trabajo, Spider, y que describimos en la sección 1.4.2. Es por todo lo anterior por lo que se ha decidido usar Lustre como el sistema de ficheros con el que realizaremos la evaluación experimental de nuestra librería de E/S paralela para Chapel.

Los componentes de un sistema Lustre son: i) un servidor de administración (*Management Server, MGS*), que contiene la información de la configuración del sistema; ii) un servidor de metadatos (*MetaData Server, MDS*) que administra los nombres, directorios y metadatos en general del sistema de ficheros; iii) servidores de almacenamiento de objetos (*Object Storage Servers, OSS*) que proveen los servicios de E/S de los ficheros. Los metadatos se almacenan en un (*MetaData Target, MDT*), mientras los datos se almacenan en (*Object Storage Targets, OSTs*). Cada OSS puede servir varios OSTs. Tanto los MDTs como los OSTs son interfaces a dispositivos de bloques, generalmente LUNs (*Logical Unit Number*) SCSI, que los MDS y OSS accederán mediante una SAN, tal y como se explicó en la sección 1.1.3.

En la figura 1.3 se puede ver un sistema Lustre que combina varias de las soluciones

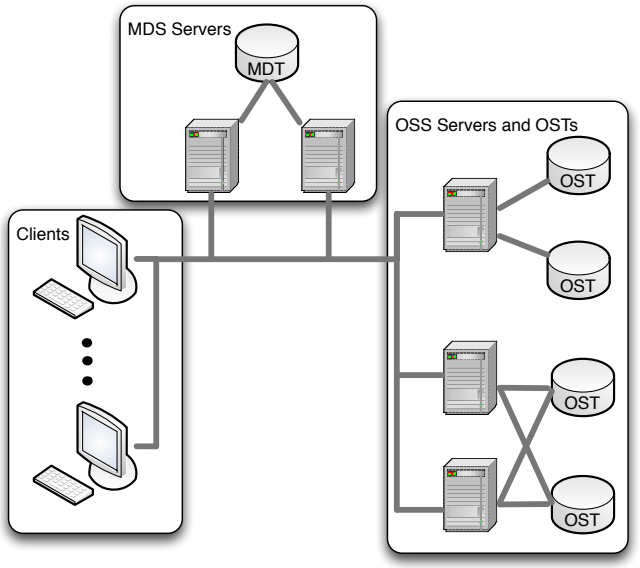


Figura 1.3: Un ejemplo de sistema Lustre, con un MDT, cuatro OSTs, dos MDS y tres OSSs. Existen canales redundantes entre los dos últimos OSTs y OSSs para proporcionar tolerancia a fallos hardware o software.

que se suelen usar. Por un lado el MGS se suele instalar en los nodos MDS, ya que consume muy pocos recursos, tanto de espacio como de ancho de banda. Un sistema Lustre sólo puede tener un MDT y un MDS, por lo que se pueden usar varios nodos para tener un respaldo en caso de que falle el que se está usando. Es por eso por lo que aparecen dos nodos MDS que comparten un MDT en la figura 1.3. Es habitual que un OSS administre varios OSTs. Aunque un OST sólo puede estar administrado por un OSS, es normal que tengan conexiones a varios OSSs por si uno falla que otro se pueda hacer cargo de él. Las redes de interconexión pueden ser ethernet, infiniband u otras.

Podemos encontrar dos fuentes de paralelismo en el sistema de ficheros Lustre. Por un lado tenemos el uso de múltiples OSTs en paralelo gracias a accesos concurrentes de distintos procesos. Por otro lado, el acceso a un único fichero se realiza en paralelo ya que normalmente está particionado en trozos (llamados *stripes*), repartidos entre múltiples OSTs. Esto provoca efectos anti-intuitivos en el acceso a los datos, ya que, por ejemplo, lo normal es que sea más rápido usar más OSTs que usar menos. Sin embargo, en [29] se puede leer que conforme se usan más OSTs el sistema es mucho más lento. Por ejemplo, con 104 agregadores, usando 16 OSTs obtienen un ancho de banda de 1200 MB/s, mientras usando 2 OSTs es de unos 3000 MB/s. Es decir, puede ser más óptimo

el realizar unos pocos accesos contiguos a grandes trozos de ficheros en paralelo. Sin embargo, los autores de [29] defienden que en Lustre, aparecen otros casos en lo que es más óptimo el realizar muchas operaciones de E/S pequeñas y no contiguas, que unas pocas muy grandes y contiguas. El motivo que encuentran es que la escritura de bloques contiguos muy grandes provoca una alta sobrecarga debido a la necesidad de tener que multiplexar de forma concurrente varios canales de comunicaciones con los dispositivos de almacenamiento.

El particionado de un fichero entre distintos OSTs es configurable a nivel de directorio. Todos los ficheros que se crean en un directorio heredan las propiedades del mismo. Las tres variables que definen el comportamiento de la división en *stripes* (bloques) de los ficheros son:

- *stripe_size* indica el tamaño de cada *stripe* en un OST. Por defecto es de un Mebibyte (2^{20} bytes).

- *stripe_count* es el número de OSTs a usar para un fichero.

- El identificador del primer OST donde empezar a crear el fichero. Por defecto es -1 que implica que es el MDS el que elige el primer OST. La política de selección se basa en el espacio disponible y el balanceo de la carga. No se recomienda cambiar el valor por defecto ya que una asignación manual suele conducir a un sistema menos óptimo y desbalanceado.

Existen dos políticas de asignación de *stripes* a OSTs. La primera es *round-robin* (cíclica) de forma que *stripes* consecutivos del fichero se almacenan en OSTs consecutivos. Esta es la política que se usa cuando todos los OSTs están ocupados mas o menos en la misma medida. Si embargo si la máxima diferencia entre los porcentajes de espacio libre de los OSTs supera un umbral (por defecto, el 17 %) se utiliza una política de asignación diferente. Esta política alternativa intenta ocupar primero los OSTs menos ocupados, pero usando cierta aleatoriedad para no seleccionar el mismo OST para *stripes* consecutivos. Tanto el valor umbral como el factor de aleatorización son configurables. Aunque el número máximo de OSTs era de 160 hasta la versión 1.8 de Lustre, actualmente puede llegar a 2000.

En Lustre, los ficheros se identifican mediante un número de 128 bits llamado FID (*File ID*). EL FID permite encontrar en el MDT un mapa de bits en el que se indica en qué OSTs está almacenado el fichero. Una de las mejoras en las últimas versiones de Lustre ha sido el incorporar el FID al nombre del fichero, acelerando la velocidad de los accesos, sobre todo en los listados de ficheros con el comando `ls`.

1.2.4.1. Problemas de rendimiento debidos a la gestión de consistencia.

El sistema de ficheros Lustre incorpora un protocolo de consistencia que arbitra los accesos a los objetos que contienen los datos y los metadatos. La lectura en paralelo de un mismo *stripe* no puede dar lugar a incoherencias, así que distintas lecturas se pueden realizar en paralelo sin que exista ninguna contención entre ellas. El sistema de consistencia es útil en presencia de escrituras y mantiene la coherencia aún cuando se realicen varias escrituras concurrentes sobre el mismo *stripe*. Este sistema también previene la lectura de datos incoherentes de una posible caché en el cliente. Para ello, cada OST actúa como un servidor del sistema de bloqueo para los objetos que controla [12]. El protocolo de bloqueo requiere que se obtenga un cerrojo (*lock* de ahora en adelante) antes de que cualquier dato pueda ser modificado o escrito en la caché del cliente. Veamos a continuación cómo los problemas de contención en los *locks* pueden degradar severamente el rendimiento.

El proceso que se encarga de los bloqueos en Lustre es el LDLM (*Lustre Distributed Lock Manager*) [132][73], que está basado en el sistema de bloqueos del VAX/VMS [113]. De hecho, en el VMS existían los AST (*Asynchronous System Trap*), que son heredados en Lustre y renombrados como *callbacks*. Como veremos a continuación, estos *callbacks* pueden ser suministrados por los clientes para optimizar el funcionamiento del sistema.

En Lustre existen cuatro tipos de *locks* y seis modos distintos de bloqueos. Nos centraremos aquí en resumir las características que nos interesan de los bloqueos para evitar incoherencias en las escrituras. Cuando un cliente solicita un bloqueo de escritura sobre un segmento (*extent*) en un OST, se le concede un cerrojo sobre todos los *stripes* de ese OST que van desde el inicio del segmento seleccionado hasta el final del fichero. Si concurrentemente otro cliente intenta escribir en un *stripe* posterior, entrará en conflicto con el bloqueo que ya existe. Este último cliente puede usar un *callback* para pedir al primer cliente que retire el *lock* del *stripe* que interesa. Si la petición es aceptada el segundo cliente tendrá un *lock* en ese OST desde la posición solicitada hasta el final, y podrá escribir en su *stripe*. Pero si el primer cliente no libera el *lock* (porque por ejemplo quiere acceder a otro *stripe* del fichero afectado por ese bloqueo), el segundo cliente se quedará bloqueado. Por tanto el resultado efectivo del intento de acceso concurrente desde varios nodos a *stripes* almacenados en un mismo OST es que el acceso será serializado. Además estaremos incurriendo en el coste de adquisición de *locks* y la consiguiente contención en el sistema de gestión de consistencia en Lustre. Aunque esta decisión de diseño puede resultar chocante, otras alternativas incrementan demasiado los costes, como se discutirá posteriormente. Además, con la implementación actual Lustre ha demostrado ser suficientemente eficiente en su contexto habitual: sistemas de supercomputación donde se ejecutan al mismo tiempo una gran cantidad de programas

paralelos, todos ellos realizando eventualmente operaciones de acceso al sistema de ficheros.

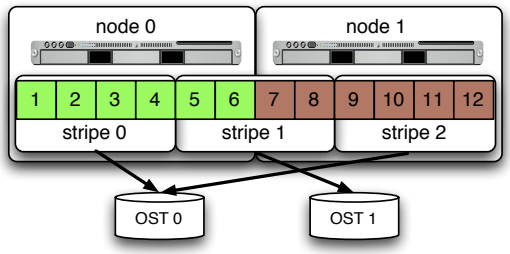


Figura 1.4: Escritura secuencial en Lustre al escribir desde el nodo 0 y 1 en el *stripe* 1.

En la figura 1.4 mostramos un ejemplo de escritura paralela para ilustrar el efecto de los mecanismos explicados en el párrafo anterior. En la figura representamos dos nodos que almacenan un array de 12 elementos distribuido por bloques (6 elementos en el nodo 0 y los últimos 6 en el nodo 1). El tamaño de *stripe* (*stripe_size*) es de 4 elementos y el número de OSTs (*stripe_count*) es dos. Debido a las distribución cíclica de *stripes* sobre OSTs, los *stripes* 0 y 2 se almacenan en el OST 0 y el *stripe* 1 en el OST 1. El problema en escritura es similar al de *false sharing* que aparece en los sistemas de caché cuando distintos cores escriben diferentes elementos de un bloque de caché. En el caso de nuestro ejemplo, si los nodos 0 y 1 escriben concurrentemente en el fichero, el *lock* asociado al *stripe* 1 secuencializará las escrituras en el *stripe* compartido: los elementos 5 y 6 del array no se podrán escribir al mismo tiempo que los elementos 7 y 8. Lo ideal sería que los accesos de escritura estén alineados al tamaño del *stripe*, pero eso no siempre es posible y existe una potencial degradación en el rendimiento [81]. Esta degradación no sólo se produce por el bloqueo producido en el acceso a los *locks* de los *stripes*, sino también por la contención que se produce en el acceso a los servicios de bloqueos de los OST [29].

Una fuente adicional de degradación en el rendimiento es la contención en el acceso a los OSTs. Por ejemplo, en la figura 1.4 también se puede ver que los *stripes* 0 y 2 compiten por el mismo OST 0. Cuando el nodo 0 esté escribiendo en el *stripe* 0 tendrá también bloqueados todos los *stripes* de ese fichero en el OST 0 desde esa posición hasta el final. Si eventualmente el nodo 1 quiere escribir en el *stripe* 2, el acceso estará bloqueado hasta que el nodo 0 no libere el *lock* tras escribir el *stripe* 0. Es decir, los *stripes* 0 y 2 se escribirán secuencialmente.

Al problema de los bloqueos se le pueden dar varias soluciones. Una posibilidad consiste en reducir la granularidad de los *locks* al tamaño de un único *stripe*. De esta

forma se evitaría que el acceso a un *stripe* bloquee el acceso a *stripes* posteriores en el mismo OST. Sin embargo los desarrolladores de Lustre han desestimado esta alternativa ya que aumentaría de forma prohibitiva el número de *locks* a almacenar en el sistema[1]. Otra solución que se lleva implementando desde hace muchos años, consiste en realizar los bloqueos de los *stripes* por adelantado, de forma que cada cliente bloquea con un solo *lock* un número variable de *stripes*. Esta alternativa aún está en desarrollo y sigue presentando problemas de estabilidad en entornos reales [47].

Por otro lado, Lustre soporta bloqueos en grupo, lo que permite que varios clientes compartan un mismo *lock*. Sin embargo, esta estrategia tiene el problema de que la visibilidad de los cambios realizados por esos clientes no es inmediata. Es decir, un cliente necesitaría hacer un *fsync* y liberar el bloqueo ante cada cambio del que se quiera tener visibilidad inmediata. Por ejemplo, si un fichero contiene metadatos relativos al formato del mismo o un CRC (código de redundancia cíclica) que hay que ir actualizando cuando cambia el contenido, la gestión de la visibilidad entre los clientes eliminaría todas las ventajas de los bloqueos en grupo.

Uno de los grandes problemas de rendimiento en todos los sistemas de ficheros distribuidos que implementan POSIX son los bloqueos que hay que usar para evitar accesos inconsistentes a datos cuando se están realizando lecturas y escrituras desde distintos nodos. La forma más sencilla de evitar el uso de *locks* es que cada proceso/nodo escriba en un fichero distinto. De esa forma no existirán bloqueos, aunque encontraremos problemas por la alta carga de acceso a metadatos en trabajos que usen muchos nodos, y por lo tanto muchos ficheros. Recordemos que en grandes sistemas HPC como los descritos en el apéndice A.1.2, un mismo trabajo se puede ejecutar de forma paralela en cientos de miles de cores. Otro problema, quizás más importante, de que varios nodos escriban cada uno en un fichero independiente los datos que almacena localmente, es que los ficheros resultantes incorporan de forma implícita la distribución de datos que tenían la estructuras de datos que se escriben. Si posteriormente la información que almacenan esos ficheros se quiere volver a leer por un programa que se ejecuta en un número diferente de nodos al que se usó en la escritura, o se requieren los datos con una distribución diferente, esto conllevaría una redistribución de los datos y el incremento en el tiempo de lectura y redistribución puede anular las ventajas de la escritura en varios ficheros separados.

También se ha estudiado la posibilidad de usar un enfoque mixto: i) en una primera fase cada nodo escribe en un fichero diferente la porción de datos que le ha tocado según la distribución de datos; ii) en una segunda fase, otro programa diferente se ejecuta en *background* (segundo plano) unificando todos los ficheros resultantes de la fase anterior

[1] Es ampliamente conocido que implementar *locks* de grano fino consume muchos más recursos y es más propenso a errores de sincronización e interbloqueos que implementar *locks* de grano grueso.

en un único fichero. Sin embargo, en sistemas HPC, donde la carga de trabajo es siempre alta, la ejecución de un trabajo adicional en segundo plano no compensa. Este segundo trabajo tendría que volver a abrir los ficheros "parciales", reordenar adecuadamente los datos y hacer una escritura secuencial en un único fichero. Por tanto, el tráfico neto en la red de E/S así como el coste computacional serían mucho mayores que si directamente se hubiera implementado una escritura en paralelo en un único fichero. En [141] se explora esta alternativa y se explota el uso de los mecanismos internos de Lustre para unificar distintos ficheros. Los resultados muestran que efectivamente la escritura en paralelo en distintos ficheros es mucho más rápida (ya que evita completamente el problema de los locks de Lustre). Sin embargo, la unificación de los archivos en uno solo, para su posterior lectura, termina anulando la mejora en rendimiento de la escritura paralela.

1.2.4.2. Arquitecturas de interconexión en Lustre.

Existen implementaciones de Lustre para distintos protocolos de red de bajo nivel, como *verbs* en infiniband, SeaStar, Gemini, Aries, etc, por lo que se puede conseguir un rendimiento óptimo en esos sistemas. Sin embargo, en un cluster que tenga más de una infraestructura de comunicaciones, o en el que el sistema de ficheros Lustre use una red distinta a la que usan otros nodos del cluster, será necesario usar routers entre las distintas redes. Para ello se usa el LNET (*Lustre NETworking*), que se encarga de hacer de *proxies* entre las distintas redes, redirigiendo los comandos que van generando los nodos a los servidores de Lustre apropiados. Cuando un cliente va a realizar una E/S de un fichero que está en Lustre, lo primero es identificar donde está el LNET del MDS que quiere acceder, y después decidir a qué LNET corresponde el OSS que contiene el dato solicitado. Esta operación dependerá de dónde están los nodos que actúan de LNETs, de dónde está el OST que se necesita acceder y de las topologías de las redes de comunicaciones subyacentes. Generalmente se usa un algoritmo conocido como enrutado de grano fino (*fine-grained routing*) [31][43], que consiste en tener un mapa predefinido en cada cliente y servidor. Dada una comunicación, el algoritmo indica qué LNET usar para llegar de uno a otro. En principio se usarán los caminos que tengan una prioridad mayor, que usualmente es equivalente a elegir los que requieren menos saltos (*hops*) por los switches de la red. Sin embargo, si el camino mínimo tiene algún problema se pueden seleccionar otros de mayor coste en aras de la tolerancia a fallos. Esto se explica con ejemplos en las secciones 1.4.3, 1.4.4 y A.1.2.

1.2.4.3. Trabajos relacionados con la evaluación y optimización de Lustre

En "*Evaluation of a Performance Model of Lustre File System*" [143] se intenta modelar un sistema de ficheros Lustre con el objetivo de encontrar los factores que influyen

en el rendimiento para optimizarlo. Con los modelos propuestos, los autores encuentran una diferencia entre los resultados reales y los que predice el modelo de entre un 17 % a un 28 %. Este hecho refleja la gran complejidad de estos sistemas así como la dificultad de modelar con precisión el sistema de ficheros Lustre.

En [110] se describen las decisiones de diseño y las pruebas que se realizaron en uno de los sistemas de ficheros que hemos usado, Spider, en el ORNL (*Oak Ridge National Laboratory*). Spider es un sistema de ficheros Lustre instalado para dar servicio a todos los sistemas del ORNL. En el estudio mencionado detectaron que el disponer de varios sistemas de ficheros, uno por sistema HPC, era ineficiente. Las fuentes de dicha ineficiencia son varias: las necesidades de mover datos de un sistema a otro donde son analizados, el coste de adquisición, mantenimiento y administración de varios servidores de E/S, y la posible redundancia de datos, de los mismos o distintos usuarios, en distintos sistemas de ficheros. Estos motivos desembocaron en la decisión de diseñar un sistema de ficheros autónomo, independientemente del resto de sistemas HPC, pero que pudiera ser compartido por todos ellos. Las lecciones aprendidas tras el diseño del sistema Lustre más grande hasta el momento se encuentran en [111] y en la sección 1.4 resumimos algunos detalles de la arquitectura HW y SW de Spider.

En [71] encontramos un análisis de las mejoras implementadas en MPI-IO por Cray para sus sistemas de archivos basados en Lustre. Proponen el uso de buffers colectivos y agregadores con distintas implementaciones configurables por el usuario mediante variables de entorno. Por ejemplo, la variable `MPICH_MPIIO_CB_ALIGN` permite seleccionar tres estrategias de acceso concurrente al sistema de archivos. La estrategia 2, `MPICH_MPIIO_CB_ALIGN=2`, se basa en usar agregadores. Los agregadores son los nodos de computación encargados de agregar los datos desde otros nodos para luego transferirlos al sistema de archivos. En la estrategia 2, los agregadores colectan los datos de forma que cada agregador se responsabiliza de todos los *stripes* que se han de almacenar un único OST. El resultado final es que cada OST es escrito desde un único agregador y que cada *stripe* se escribe también desde un único agregador. De esta forma, la estrategia 2 esquiva el problema de los bloqueos pero a cambio tiene un alto coste de redistribución de datos entre los nodos y los agregadores, especialmente si los trozos a agregar son muy pequeños. En las pruebas de evaluación en [71] se puede ver que en la escritura de un fichero de 96 GiB de 32 nodos, usando POSIX solo se alcanzan 420 MiB/seg mientras que con la estrategia 2 de MPI IO se obtienen 1629 MiB/seg. En la sección 2.3 discutimos con más profundidad esta alternativa y nuestra propuesta basada en Chapel en lugar de en MPI IO.

Además existen multitud de artículos donde se describen optimizaciones de Lustre, como por ejemplo, realizar E/S colectiva uniendo ficheros [141], o bien se han estudiado distintos algoritmos de enrutamiento en [31][30][43], y también se describen optimizaciones para su uso en MPI-IO en [28][29]. También se ha estudiado el comportamiento

de Lustre de forma exhaustiva, como por ejemplo en [143][109][98][97][112][111][99].

Sin embargo, ninguno de los trabajos anteriores estudia los problemas de mejora de la productividad que afrontamos en esta tesis y que serán abordados en los próximos capítulos.

1.3. Lenguajes de alta productividad

Como hemos comentado anteriormente, el paso a sistemas *Exascale* requerirá del desarrollo de nuevos modelos de programación capaces de gestionar billones de elementos de procesamiento concurrentes así como de manejar eficientemente la E/S. El desafío mayor aparecerá en las grandes aplicaciones de tipo *data-intensive* (*Big-Data*).

Una opción para programar *productivamente* los sistemas de *Exascale* pasa por crear nuevos lenguajes de programación, y de hecho es uno de los objetivos tanto del proyecto HPCS como del programa FETHPC-H2020. Sin embargo, es una actividad altamente arriesgada, ya que se puede invertir mucho esfuerzo en su creación sin que creen un impacto en la comunidad científica. Podemos encontrar ejemplos de proyectos anteriores de gran envergadura, en los que se ha intentado crear nuevos lenguajes de programación que mejoraran los ya existentes. Por ejemplo, entre las décadas de 1970 y 1980 se intentó substituir al COBOL y al Fortran, los lenguajes más populares en aquellos momentos, por un nuevo lenguaje, el Ada, pero a pesar de todos los recursos que se invirtieron en el proyecto, no llegó a ser lo popular que se esperaba, limitándose hoy en día su uso al ámbito del software empotrado. Otro proyecto similar al HPCS, el de *5th-generation*, se creó en Japón en la década de los 80, donde se dieron un plazo de diez años para encontrar un nuevo modelo de programación más productivo. Este esfuerzo dio como resultado el CPL (*Concurrent Logic Programming*), un dialecto de Prolog orientado al paralelismo y a obtener un alto rendimiento, pero que no ha transcendido más allá del ámbito educativo. El motivo de esos fracasos se puede encontrar en que la comunidad de programadores ha valorado más la portabilidad, el rendimiento y la evolución incremental de los lenguajes ya en uso, antes que la elegancia, la expresividad o incluso su facilidad de uso de uno nuevo, al menos en la programación de sistemas HPC.

El diseñar una aplicación que se ejecuta en un sistema de memoria distribuida presenta el gran inconveniente de que, puesto que el espacio de direcciones está repartido entre los nodos del sistema, cada nodo tiene que gestionar el acceso a posiciones de memoria remotas (es decir, alojadas en otro nodo) a través de rutinas de comunicaciones de bajo nivel. Inicialmente la programación en estos sistemas se basaba en el uso de librerías como PVM (*Parallel Virtual Machine*) [54] o MPI (*Message Passing Interface*) [52], que exponen al programador un API para la gestión de comunicaciones remotas

punto a punto, comunicaciones globales colectivas, operaciones de sincronización, etc. Pero los algoritmos implementados con estas librerías son complicados de programar, depurar y mantener.

Para simplificar la gestión de los accesos remotos, se propuso un nuevo modelo de programación paralelo denominado PGAS (*Partitioned global Address Space*) [140], que ofrece al programador la visión de un espacio de direcciones global y lógicamente compartido entre los procesos, dejando en manos del compilador y *runtime* la gestión de los accesos remotos, puesto que los datos están físicamente distribuidos entre los nodos. En este modelo se inspiran lenguajes como UPC (*Unified Parallel C*), [24], X10 [18], Fortress [40], o Chapel [15]. Este paradigma simplifica enormemente la tarea de programación desde el punto de vista del usuario. El modelo de programación PGAS intenta conjugar el rendimiento que aporta el acceso a datos alojados localmente en cada nodo, con la simplicidad y programabilidad de un modelo de memoria compartida, puesto que ofrece un espacio de direccionamiento global, que es directamente accesible por cualquier proceso.

1.3.1. Chapel como ejemplo de lenguaje PGAS

Chapel [15] es un lenguaje paralelo que forma parte del programa HPCS (*High Productivity Computing Systems*) del DARPA, para mejorar la productividad del usuario en sistemas masivamente paralelos. Chapel soporta el modelo de programación PGAS (*Partitioned Global Address Space*). Para ello provee una visión global de las estructuras de datos, así como una visión global del control.

Una ventaja de Chapel respecto al resto de lenguajes HPCS, es que es un lenguaje *multiresolución*. Con esto nos referimos al hecho de que ofrece un buen número de funcionalidades de alto y bajo nivel que permiten trabajar con distintos niveles de abstracción y control, en un entorno unificado. Por ejemplo las funcionalidades de bajo nivel permiten mayor control del hardware y que el programador pueda acceder a detalles de implementación del runtime. Por el contrario, las funcionalidades de alto nivel abstraen el hardware y permiten definir operaciones de manera global, sin preocuparse por introducir comunicaciones para los accesos remotos, sincronizaciones para los accesos compartidos, etc. De hecho, las funcionalidades de alto nivel en Chapel están basadas en las de bajo nivel, para asegurar que todas son interoperables.

Un ejemplo de multiresolución es el soporte que ofrece Chapel para expresar paralelismo de tarea o paralelismo de datos. En general, el paralelismo de tarea ofrece funcionalidades de bajo nivel como crear tareas, despacharlas, sincronizarlas, etc. Por el contrario, el paralelismo de datos ofrece funcionalidades de alto nivel como la definición de distribuciones de datos y la gestión automática del reparto del trabajo y de las comu-

nicaciones. Por ejemplo, una funcionalidad de alto nivel que ayuda a los programadores a razonar sobre localidad es el concepto de *locale*, que en Chapel es un tipo predefinido. Un *locale* es una representación abstracta, para una arquitectura concreta, de los datos que están físicamente compartidos dentro de un nodo (son locales a él). El acceso a un dato por parte de una tarea es *local* si la tarea y el dato están mapeados en el mismo *locale*, y *remote* en cualquier otro caso. En una arquitectura paralela convencional un *locale* hará referencia a un nodo del sistema donde el espacio de direcciones es compartido por todos los *cores/threads* del nodo. El programador puede controlar, usando cláusulas específicas de bajo nivel, el que una tarea o *thread* se ejecute en un determinado *locale*. En los capítulos 2 y 3 se usará frecuentemente el término *locale* en referencia a los nodos de un sistema HPC. Para más detalles consultar [15].

```
#include <omp.h>
#include <iostream>
#include <cmath>

using namespace std;

int main() {

  int   i, n, chunk;
  float a[1000], b[1000],result;

  // Some initializations, done sequentially
  n = 1000;
  chunk = 100;
  result = 0.0;
  for (i=0; i < n; i++)
    {
      a[i] = 1.0/(i+1);
      b[i] = 1.0/(i+1);
    }

#pragma omp parallel for default(shared) private(i) schedule(static,chunk
    ) reduction(+:result)

   for (i=0; i < n; i++)
     result = result + (a[i] * b[i]);

 std::cout << "Final result=" << result << "so pi="  << sqrt(6*result) <<
     std::endl;

}
```

Figura 1.5: Código OpenMP para calcular pi en paralelo [13]

En la figura 1.5 podemos ver un código paralelo para calcular el número pi realizado en OpenMP. En la figura 1.6 tenemos el mismo algoritmo programado en MPI, y finalmente en la figura 1.7 podemos ver el mismo algoritmo implementado en Chapel. En este último caso vemos que el algoritmo sólo ocupa una línea, la 2, que, al igual que en los lenguajes anteriores, se encarga de ejecutar N iteraciones de forma paralela, repartiéndolas automáticamente entre todos los cores que haya disponibles. Después se reducen los resultados sumando todos los valores generados en los distintos *threads*, y finalmente se multiplica la reducción por 6 y se hace la raíz cuadrada, obteniendo tras esa operación una aproximación de pi. En el ejemplo de Chapel, las variables que van almacenando los resultados intermedios, y que serán usados en la reducción, se definen de forma implícita, es decir, no aparecen. Además el código Chapel, permite cambiar el valor de N sin necesidad de cambiar el código. Para ello, en la línea 1 de la figura 1.7 se define N como de tipo *config*, lo que permite que se pueda pasar ese valor como un argumento en la línea de comandos al invocar el ejecutable.

1.3.2. E/S en lenguajes PGAS

En general los lenguajes de programación PGAS presentan una interfaz directa para hacer E/S, con un acceso directo a la interfaz POSIX, pudiendo abrir (*open*), buscar (*seek*), leer (*read*), escribir (*write*) y cerrar (*close*) los ficheros, pero es muy complicado el usar esas funciones de forma distribuida y eficiente, sin crear contención o condiciones de carrera en los accesos, por lo que en general estos lenguajes no aprovechan las posibilidades ofrecidas por los sistemas de ficheros paralelos. Por ejemplo, tanto X10 [18] como Fortress [40] tienen sólo implementado la interfaz POSIX [127] para realizar E/S, y no tienen aún ni en proyecto la implementación de un nuevo interfaz.

En el lenguaje UPC sí crearon un documento [37] cuya última versión es del año 2006, donde se definen una serie de propuestas de adiciones y extensiones al lenguaje UPC, con la idea de que una vez que estén implementadas y sean estables se incorporen a las especificaciones del lenguaje. Estas funciones se añadieron a la versión 1.2 de las especificaciones del lenguaje UPC [22], siendo implementadas en algunos compiladores [128], pero no aparecen en la última versión, la 1.3 [23] y no se garantiza su soporte a largo plazo. Entre las características básicas de esas propuestas están que las nuevas funciones serán colectivas, lo que significa que serán llamadas a la vez desde todos los *threads* del programa en ejecución. Además se apuesta por un modelo de *consistencia débil*, por lo que se ha de cerrar el fichero o realizar una operación de sincronización explícita en todos los *threads* para garantizar que cualquier *thread* pueda acceder correctamente a los datos almacenados por otros. Una restricción de estas propuestas es que no se podrán usar a la vez las funciones POSIX y las definidas por UPC-IO.

```
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <mpi.h>
#include <unistd.h>
#include <string.h>

using namespace std;

int main(int argc, char **argv)
  {
    int i,n=1000, nthreads;
    double p_sum,sum;
    char *CPU_name;
    int loop_min, loop_max, tid;

    MPI_Init(&argc,&argv);
    // get the thread ID
    MPI_Comm_rank(MPI_COMM_WORLD, &tid);
    // get the number of threads
    MPI_Comm_size(MPI_COMM_WORLD, &nthreads);

    n=1000;
    sum = 0.0;
    p_sum = 0.0;
    CPU_name = (char *)calloc(80,sizeof(char));
    gethostname(CPU_name,80);
    printf("thread %d running on machine = %s\n",tid,CPU_name);

    loop_min = 1 +  (int)((long)(tid + 0) *  (long)(n)/(long)nthreads);
    loop_max =      (int)((long)(tid + 1) *  (long)(n)/(long)nthreads);
    printf("thread %d loop_min=%i loop_max=%i\n",tid,loop_min, loop_max);

    for(i=loop_min;i<loop_max;i++)
         p_sum += 1.0/(i*i);
    printf("thread %d partial sum=%f\n", tid,p_sum);

    MPI_Reduce(&p_sum,&sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

    if (tid == 0)
       printf("sum = %f so pi = %f\n",sum,sqrt(6*sum));

    MPI_Finalize();
  }
```

Figura 1.6: Código MPI para calcular pi en paralelo [13].

```
1 config const N = 1000;
2 var pi = sqrt(6.0 * (+ reduce [ i in 1..N ] 1.0/(i*i) ));
3 write ("pi =",pi);
```

Figura 1.7: Cálculo paralelo de pi en el lenguaje de alta productividad Chapel.

En general, la mayoría de los lenguajes PGAS cuando quieren gestionar eficientemente las operaciones de E/S recurren a una librería que actúe de intermediaria entre la aplicación y el sistema de ficheros. La librería más usada en estos casos es MPI IO.

1.3.2.1. MPI IO

MPI IO es la librería más usada para realizar accesos concurrentes sobre sistemas de ficheros paralelos. Su desarrollo comenzó en 1994 en IBM, la NASA la adoptó en 1996 y en 1997 se incorporó al estándar MPI-2. En general se ha realizado poca investigación en MPI-IO debido a que es complejo obtener un buen rendimiento [36][29][143]. Además el interfaz en sí no se puede cambiar sin romper la compatibilidad, por lo que es complicado plantear mejoras del interfaz.

Podemos ver en la figura 1.8 un ejemplo de código MPI-IO para escribir de forma paralela en un fichero. Antes, hay que inicializar la librería MPI, incluyendo llamadas a `MPI_Init`, `MPI_Comm_rank`, `MPI_Comm_size`, y varias a `MPI_Bcast` para difundir los datos que son comunes a todos los procesos MPI. En la línea 3 vemos una llamada a `MPI_Dims_create`, que creará un sistema cartesiano de procesadores para describir la disposición virtual de los nodos que implementarán la E/S. Tras eso, en la línea 6, se llama a la función `MPI_Type_create_darray` para crear un nuevo tipo de fichero que es devuelto en `file_type`, y que contiene la descripción de cómo se proyectarán los datos de las distintas tareas que se van a encargar de la E/S.

Tras crear el nuevo tipo de fichero, en la línea 7 se comunica a MPI para que pueda ser usado. Tras eso abrimos el fichero en la línea 10, con lo que creamos una nueva vista del fichero, para que cada proceso sepa cómo mapear sus datos. Esta función debe ser llamada desde todos los procesos, para que cada uno obtenga su propio mapeo de los datos. Tras definir la vista, en la línea 12 se llama a la función que realmente realiza el acceso al sistema de ficheros en paralelo desde todos los nodos implicados, `MPI_File_write_all` (o a `MPI_File_read_all` en caso de que sea una lectura) para que se realice la escritura (o lectura). Tras los accesos todos los procesos pueden cerrar el fichero, como se realiza en la línea 13.

La librería MPI IO tiene optimizaciones particulares para varios sistemas de ficheros.

```
    ...
1    MPI_Comm_size(MPI_COMM_WORLD, \&pool_size);
2    ndims= number_of_dimensions_of_the_array;
3    MPI_Dims_create(pool_size, ndims, array_of_psizes);
    ...
4    MPI_Comm_rank(MPI_COMM_WORLD, \&my_rank);
5    order = MPI_ORDER_C;
6    MPI_Type_create_darray(pool_size, my_rank, ndims, array_of_gsizes,
         array_of_distribs, array_of_dargs, array_of_psizes, order,
         MPI_INT, \&file_type);
7    MPI_Type_commit(\&file_type);
    ...
8    MPI_Type_extent(file_type, \&file_type_extent);
9    MPI_Type_size(file_type, \&file_type_size);
    ...
10   file_open_error = MPI_File_open(MPI_COMM_WORLD, file_name,
        MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, \&fh);
    ...
11   MPI_File_set_view(fh, 0, MPI_INT, file_type,"native", MPI_INFO_NULL)
    ;
    ...
12   file_write_error=MPI_File_write_all(fh, write_buffer,
          write_buffer_size, MPI_INT, \&status);
    ...
13   MPI_File_close(\&fh);
    ...
```

Figura 1.8: Código MPI IO para escribir en paralelo un array distribuido.

En la práctica, dependiendo del sistema de ficheros subyacente y de la distribución de los datos, las capas más bajas de MPI IO pueden decidir que o bien todos los nodos realicen operaciones en paralelo de E/S o bien que sólo un subconjunto de los nodos realicen las operaciones de E/S. Esos nodos son los denominados *agregadores* [101]. También en algunas implementaciones, se da la posibilidad de elegir el algoritmo para implementar la E/S, realizando la elección a través de una variable de entorno. Por ejemplo, en la implementación de MPI IO en sistemas de Cray se usa la variable `MPICH_MPIIO_CB_ALIGN` para elegir entre distintas estrategias de E/S.

1.4. Sistemas usados durante el desarrollo de esta tesis

Los sistemas con los que hemos trabajado en esta tesis tienen en común que sus sistemas de almacenamiento se basan en sistemas de ficheros distribuidos. Inicialmente usamos Crow, un sistema de desarrollo y pruebas interno de la empresa Cray. Posterior-

mente usamos Jaguar, el que en ese momento era el número 1 de la lista top500 [88], instalado y operado por el OLCF (*Oak Ridge Leadership Computing Facility*) del ORLN (*Oak Ridge National Laboratory*) y que tenía un sistema de ficheros con nombre propio, Spider [110].

En la descripción que realizamos a continuación comenzamos por Crow, el ordenador de pruebas de Cray. A continuación describimos el entorno del OLCF en Jaguar, donde primero detallamos el sistema de E/S (Spider), después la red de interconexión (SION) y las plataformas de cálculo (EOS) del OLCF de las que presentamos resultados en este trabajo. Finalmente describimos en este capítulo la generación actual de Picasso, el superordenador de la Universidad de Málaga, ubicado en el SCBI, en el que también se han obtenido resultados.

Otros equipos que también hemos usado en este trabajo, pero para los que no presentamos resultados, se describen en el apéndice A.1. En particular, todos esos equipos se basan tanto en el sistema de E/S Spider como en la red de interconexión SION. En ese apéndice se incluye una descripción del proyecto Summit, que estará en marcha en el OLCF en el año 2018 y cuyo avance más significativo será en su arquitectura de E/S.

1.4.1. Crow

Al comienzo de este trabajo teníamos la necesidad, a la hora de probar los algoritmos, de realizar las pruebas en un ordenador que dispusiera de un sistema de ficheros distribuido, además de soportar Chapel, el lenguaje en el que nos hemos basado. Desde la empresa Cray nos dejaron acceso a uno de los ordenadores que usan internamente para realizar pruebas y desarrollar código: Crow. Es un Cray XT5, sistema representativo de clusters de tamaño pequeño y medio en HPC, optimizado para cargas intensivas de cálculo. Crow está compuesto por 20 nodos con 2 CPUs AMD Shanghai de 4 cores, a 2.8 GHz y 32 Gigabytes de RAM. Cada nodo tiene 8 cores, sumando en total 160 cores. En el apartado de software encontramos que usan el CLE (*Cray Linux Environment*), que es un SLES 11 (*SuSE Linux Enterprise Server*) con modificaciones realizadas por Cray para que se adapte perfectamente a su hardware, un sistema de colas PBS (*Portable Batch System*) de la empresa Altair, y como sistema de ficheros el Lustre-Cray/1.8.2. El sistema de ficheros en ese caso estaba formado por un MDT (*Metadata Target*) y tres OSTs (*Object Storage Target*).

Tras contactar con David Knaak, el autor de [71], éste nos informó que sólo había dos servidores para Lustre, por lo que el MDT y un OST compartían un servidor y los otros dos OSTs compartían el otro servidor, lo que nos ayudó a entender los resultados que habíamos obtenido en las pruebas preliminares, en las que observamos que resultaba ventajoso usar un único agregador por OST, y que cada agregador construyera bloques

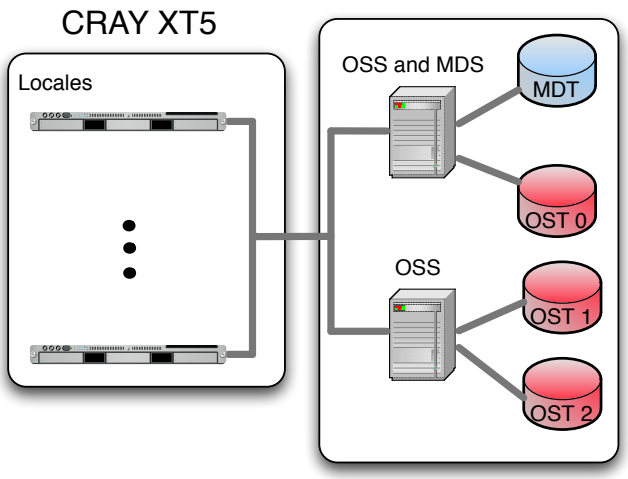


Figura 1.9: Arquitectura E/S de Crow.

de tamaño del *stripe* de Lustre para a partir de ellos realizar las transferencias de E/S con el OST asignado. La arquitectura de E/S de Crow la podemos ver en la figura 1.9.

Los Cray modelo XT5 usan una topología de torus 3D para su red de interconexión entre los nodos, implementada con chips Seastar II+, que incluyen un chip PowerPC 440, un motor DMA y varios enlaces *hypertransport* que interconectan tanto los buses *hypertransport* internos del nodo como los chips Seastar II+ de los nodos cercanos, ofreciendo una velocidad de 9.6 Gigabytes por segundo en cada enlace. La topología de torus 3D tiene varias ventajas: i) no hace falta tener switches en la red de interconexión, con lo que el consumo de energía es más bajo; ii) con un torus se usan exactamente los mismos cables para todas las conexiones, del mismo tipo y la misma longitud, lo que hace que sean simétricas y por tanto más simples de implementar y mantener; iii) en un torus se tienen caminos alternativos si un nodo falla. Se puede ver un esquema de sus interconexiones en la figura 1.10. Además incluye corrección de errores para interceptar y corregir los errores que se produzcan en las comunicaciones entre nodos, y una implementación de la librería *portals* en *firmware* para optimizar los pasos de mensajes entre los nodos, por ejemplo al usar MPI.

1.4.2. Spider I y II

Originalmente cada sistema del OLCF disponía de un sistema de almacenamiento local, lo que originaba varios problemas, entre ellos la necesidad de copiar los ficheros

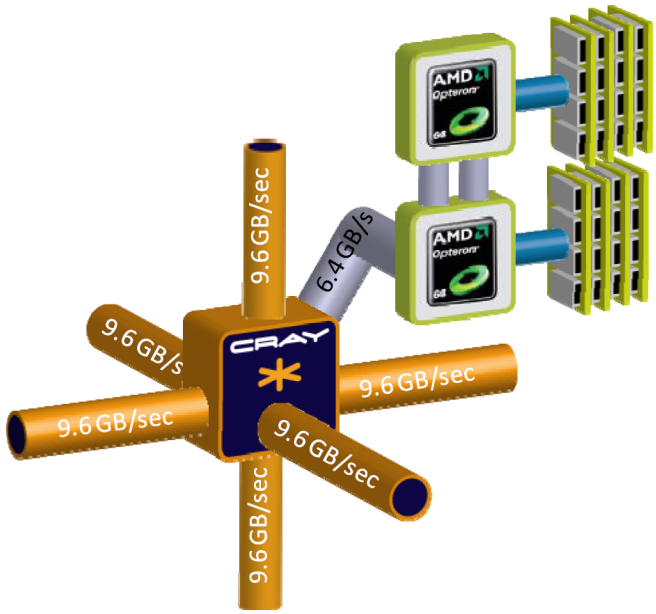


Figura 1.10: Enlaces del chip SeaStar2+, incluyendo su conexión a un nodo.

para ser usados en distintos sistemas, y de crear, construir y mantener distintos sistemas de ficheros de alto rendimiento, con el coste que ello implica [110].

Es por ello que se decidió dividir los sistemas de ficheros en tres partes, una para los datos más importantes de los usuarios, que se accede por NFS y no está visible desde los nodos de cálculo; otra para realizar los cálculos, llamada Spider, que sea de alta velocidad y baja latencia, y que se considera un almacenamiento temporal (*scratch*); y otra permanente de alta capacidad, con una alta latencia de acceso, un HSM (*Hierarchical Storage Management*) con almacenamiento terciario. Así, aunque exista un sistema de ficheros convencional (montado por NFS) para los datos más importantes de los usuarios, éste no está accesible desde los nodos de cálculo, sino sólo desde los servidores de login, nodos de transferencia de ficheros, y en general desde todos los ordenadores menos los de cálculo.

Para los nodos de cálculo se usa el sistema de E/S Spider, que provee un sistema único para las necesidades de almacenamiento de todos los sistemas existentes en el OLCF [110], evitando las copias de datos y la necesidad de disponer de distintos sistemas de ficheros de alta capacidad y velocidad. Otros de los requisitos a la hora de diseñar Spider era el permitir actualizaciones del sistema de ficheros con independencia de las

plataformas computacionales y proveer tolerancia antes fallos, tanto internos al sistema de ficheros como de las distintas plataformas que lo usan. Se eligió Lustre como sistema de ficheros para Spider, por ser el único que podía proveer, en la práctica, todos los requisitos que se le pedían.

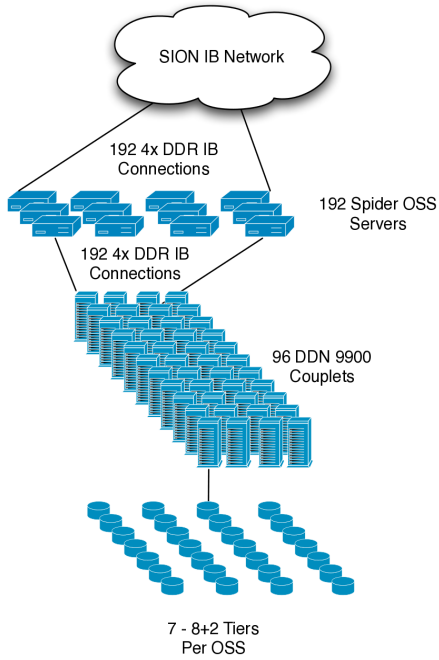


Figura 1.11: Arquitectura hardware del sistema de almacenamiento Spider I [110].

Como se puede ver en la figura 1.11, el primer Spider estaba formado por 48 controladores DDN S2A9900 (96 couplets) con 28 filas de 10 discos cada una configurados en RAID-6, teniendo 1344 OSTs (28*48), y 13440 (28*48*10) discos de 1 terabyte cada uno en total, con un tamaño neto de unos 10 Petabytes usando RAID-6. Para acceder a esos discos se usaron 192 servidores (OSSs en terminología Lustre), con lo que cada servidor sirve 7 RAIDs por defecto, estando configurados en parejas redundantes, ambos conectados a los mismos OSTs, con lo que si uno falla el otro puede tomar su lugar. Cada OSS tiene dos interfaces infiniband 4xDDR, una hacia el almacenamiento y otra hacia los clientes. Los metadatos se almacenan en 2 LSI Engino 7900, y se sirven a través de 3 servidores de 16 cores cada uno.

Para evitar los problemas de pérdida de datos, bloqueos, etc, así como para evitar

el límite de un sólo MDS que tiene el Lustre, en el OLCF decidieron crear dos sistemas de ficheros, repartiendo el espacio total entre los dos a partes iguales. Los trabajos pueden indicar al sistema de colas qué sistema de ficheros van a usar. Además como cada sistema de ficheros Lustre sólo puede tener un MDS se hace muy conveniente el tener diferentes sistemas de ficheros para poder repartir el trabajo, y evitar contención en los MDS dentro de lo posible. Por esos motivos se crearon originalmente dos sistemas de ficheros, `widow1` y `widow2`, que posteriormente pasaron a ser cuatro (`widow1`, `widow2`, `widow3`, `widow4`) y que actualmente, tras crear spider II, vuelven a ser dos con la mitad de la capacidad cada uno (`atlas1` y `atlas2`). En esta tesis se hicieron pruebas en el sistema con `widow` y con `atlas`. Los resultados presentados son sobre `atlas`, por ser más recientes.

Por defecto el número de OSTs es 4, y el tamaño de stripe es de 1 MiB, por lo que cada fichero se divide usando cuatro OSTs, aunque cada usuario puede cambiar los parámetros por defecto si lo considera conveniente. En este trabajo hemos realizado pruebas con entre 4 y 64 OSTs, y tamaño de stripe entre 1 y 10 MiB. El poder seleccionar el número de OSTs creó problemas como el límite de 160 OSTs que tenían las versiones de Lustre anteriores al 2.2, lo que imponía un límite en el tamaño de fichero de 320 Terabytes, (2 Terabytes por objeto $\times$ 160 objetos, uno en cada OST posible).

Tras poner en marcha Spider se comenzó a trabajar en la nueva versión, Spider II [98], que está actualmente en uso [112]. Spider II está dispuesto físicamente en 4 filas con 10 racks cada una, de forma que se puede configurar de distintas formas. Por ejemplo, las distintas filas pueden ser totalmente independientes entre si, permitiendo, por ejemplo, su actualización, tanto de software como de hardware, mientras el resto de filas continúan en uso. Originalmente se configuraron cuatro sistemas de ficheros individuales, y finalmente se dejaron dos sistemas de ficheros. En total Spider II tiene un total de 36 DDN SFA12K40, 288 OSSs, 36 switches infiniband FDR y 20 switches ethernet. Cada uno de los SFA12K40 contiene 2 controladoras y 10 almacenes de 60 slots cada uno, ocupados con 560 discos de 2 Terabytes, en total hay 20160 discos (560*36). Cada uno de los 2016 OSTs tiene 10 discos, y cada uno de los 288 OSSs tiene 7 OSTs, con lo que cada OSS administra 70 discos. Como en Spider II hay dos sistemas de ficheros cada uno tiene la mitad de los recursos, es decir, 10080 discos, 1008 OSTs y 144 OSSs. Cada uno de los OSSs tiene una conexión infiniband FDR hacia la red SION de interconexión y dos hacia las dos controladoras de cada DDN SFA12K40. En la figura 1.12 se muestra la arquitectura de alto nivel de Spider II descrita.

Con lo anterior se presenta un problema, y es que los nodos de cálculo en general no tienen acceso a la red infiniband a la que están conectados los servidores Lustre (los OSSs), sino que cada plataforma usa su propia red de interconexión interna. Por ello se hace necesario el usar nodos dentro de cada plataforma que actúen como routers entre el sistema de ficheros Lustre y la red local, y que presenten Lustre como si estuviera

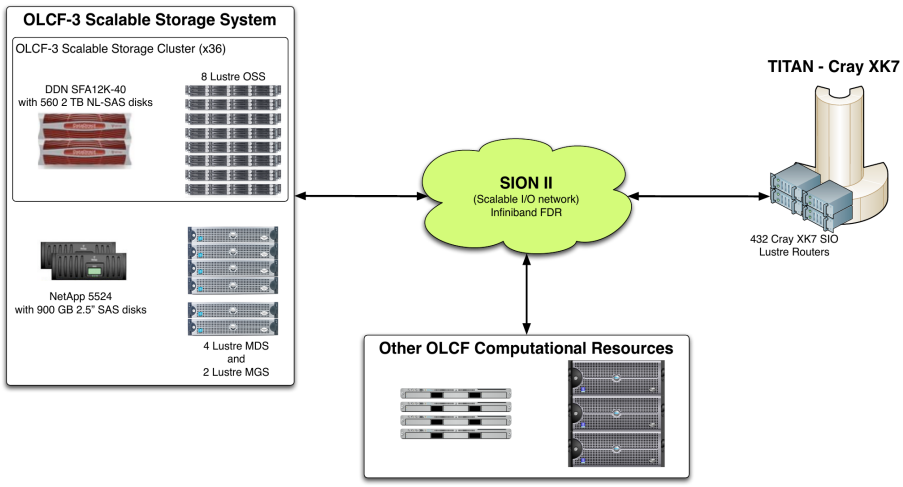


Figura 1.12: Arquitectura de Spider II [112].

conectado a la red local. Además se hace necesario el disponer de una red intermedia entre el sistema de almacenamiento Spider y los clientes, como Jaguar, Titan, EOS, y demás plataformas computacionales presentes en el OLCF del ORNL.

Para el almacenamiento a medio y largo plazo se usa un sistema HPSS, que es un desarrollo conjunto entre IBM y el ORNL, y que está formado por 3 librerías de cintas SL8500, conteniendo 10000 cintas y 24 unidades de cintas cada librería. Más información sobre el sistema de E/S Spider en [110][111][112],[97]. Además en [99] podemos ver un buen resumen de las lecciones aprendidas en todos los años de evaluación, implantación y mantenimiento del sistema de almacenamiento Spider. Una lección interesante es que es importante analizar el rendimiento individual de los discos, ya que los discos mas lentos limitarán el rendimiento del resto. Así, usando un índice de variabilidad del 5 % del rendimiento, durante el despliegue del sistema se cambiaron 1500 discos de 20160, y tras el despliegue se cambiaron 500 discos más. Finalmente prefirieron cambiar el requisito de la variabilidad máxima de un 5 % de rendimiento entre los discos por un 7.5 %, que era más realista. Otra de las lecciones aprendidas es que es conveniente exponer detalles de la infraestructura con una interfaz sencilla para que los usuarios avanzados puedan obtener un mejor rendimiento.

Pasamos a continuación a explicar la red de interconexión entre los nodos de cálculo y los sistemas de ficheros Lustre, llamada SION (*Scalable I/O Network*) y posteriormente en cada plataforma explicamos el sistema de enrutado que se ha usado. El

lector interesado puede encontrar más información sobre la red de interconexión SION en [110], [31], [43].

1.4.3. SION (Scalable I/O Network)

Los nodos de cálculo no disponen de red infiniband, sino que dependiendo del equipo tienen acceso a determinadas redes propietarias de muy alta velocidad y baja latencia, pero que, en principio, sólo sirven para realizar comunicaciones dentro de cada sistema, por lo que no sirven para acceder, por ejemplo, al sistema de almacenamiento. Es posible conectar un sistema de almacenamiento a esa red, pero entonces será local a ese nodo, no pudiendo ser accedido directamente desde fuera de esa red.

Es por ello que se vio la necesidad de crear una red de interconexión entre los sistemas del OLCF que sirviera también para comunicar los nodos de cálculo con el sistema de almacenamiento, la llamada SION. Originalmente estaba compuesta por 4 switches Cisco 7024D IB 4xDDR de 288 puertos cada uno, funcionando uno de los switches como agregador de enlaces: 2 de los switches daban conectividad entre Jaguar y el sistema de almacenamiento Spider y el cuarto daba conectividad al resto de plataformas del OLCF entre ellas, y con Spider a través del switch de agregación. En la figura 1.13 podemos ver la red SION. En la parte de abajo, donde pone Spider, están los 192 OSSs que dan acceso a los sistemas de ficheros que integran Spider, y que están conectados a través de 48 switches hoja a los switches de 288 puertos que proveen de conectividad a Spider, de forma directa a Jaguar y a través de un switch de agregación al resto de plataformas del OLCF. En Jaguar los dos segmentos que aparecen en la figura son el Jaguar antiguo, el XT4, y su sucesor, el XT5. En Jaguar XT5 hay 192 nodos que se usan como nodos de E/S (Service I/O o SIO), que están configurados como routers Lustre, al tener acceso tanto a la red interna SeaStar2+ como a la red infiniband DDR, por tanto todas las peticiones efectuadas por los nodos de cálculo de Jaguar a los sistemas de ficheros de Spider pasarán por esos nodos.

Un problema con la red SION original es que sólo Jaguar tenía routers para acceder a Spider, mientras otras plataformas accedían directamente, lo que provocaba que no sólo el tráfico de almacenamiento, si no también tráfico MPI viajara por la red SION, ya que todos los nodos de todos los sistemas, salvo Jaguar, estaban conectados directamente a la red SION. En la actualización de SION una de las mejoras más importantes ha sido la incorporación de routers LNET (*Lustre NETworking*) en todas las plataformas, con la intención de independizar el tráfico de almacenamiento del resto [97].

En la figura 1.14 podemos ver como están interconectados Spider II a través de SION II con las plataformas, apareciendo como plataforma más relevante Titan, el sustituto de Jaguar. Ahí se puede ver como los SFA12K (rectángulos verdes) están cada uno

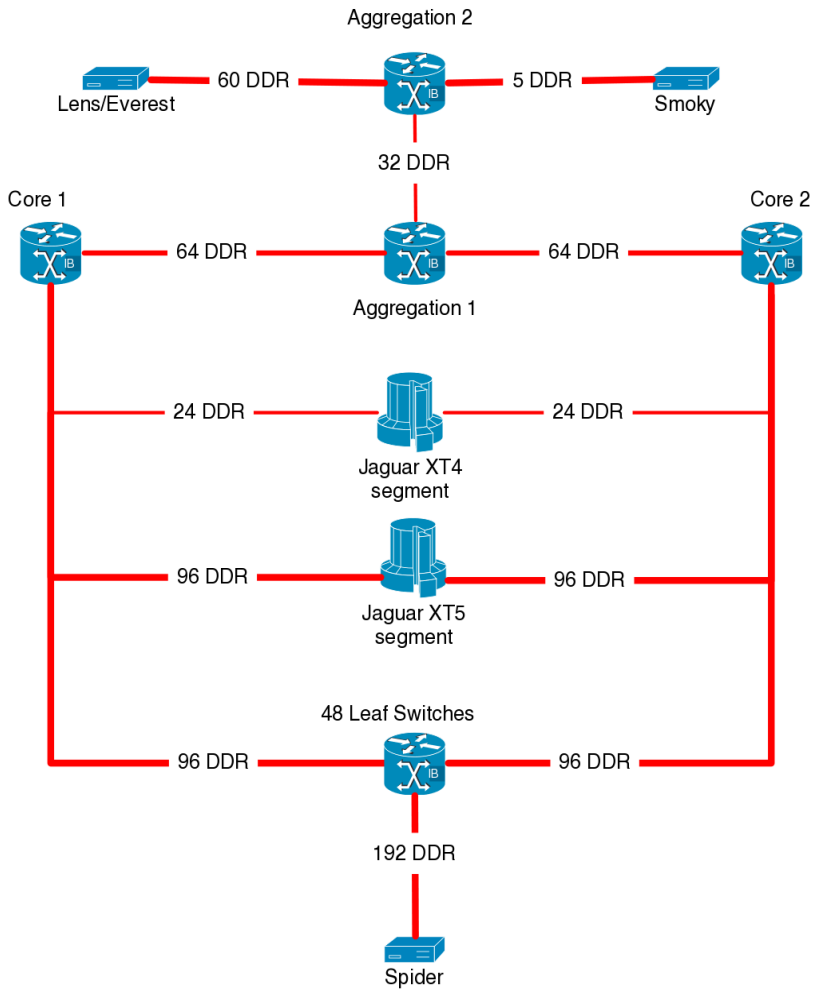


Figura 1.13: Arquitectura SION original [111].

conectados a 2 OSSs (rectángulos morados) para tener redundancia, por defecto sólo usan uno de esos enlaces, y éstos a su vez están conectados a SION (rectángulo cyan), que a su vez comunica Lustre con los LNETs que están en cada sistema concreto, siendo los LNETs los rectángulos naranjas.

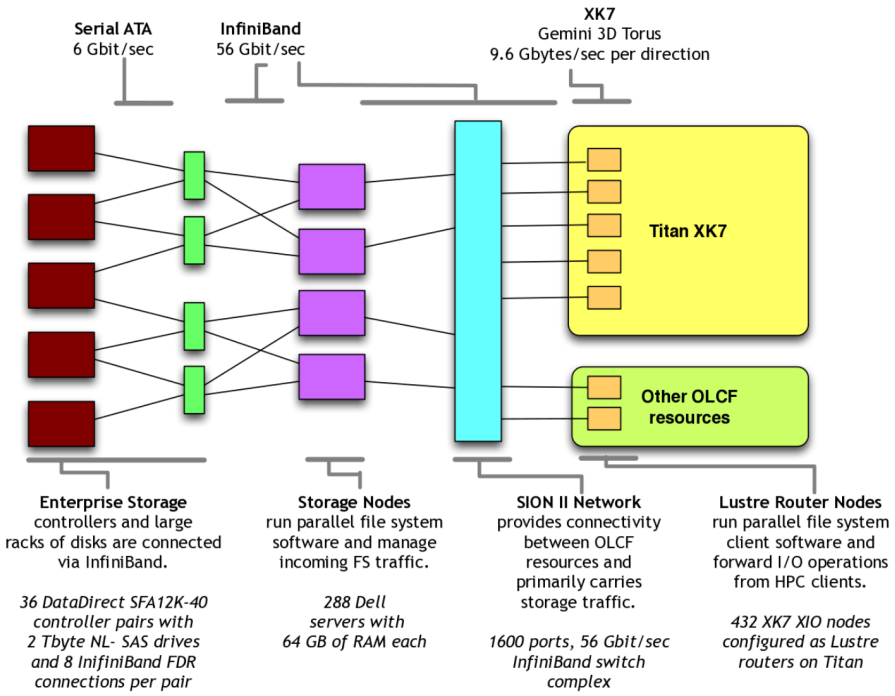


Figura 1.14: Arquitectura Spider II y SION [112].

1.4.4. EOS

El sistema usado para producir los resultados que presentamos en los capítulos siguientes, es EOS, un Cray XC30 con 744 nodos con 16 cores por nodo, con un total de 11904 cores, y que usa una red de interconexión Aries [44], con un diseño propietario de Cray y una topología de interconexión Dragonfly [69]. Hemos preferido usar este sistema para presentar los resultados finales, porque estos resultados son mucho más estables aquí que en Jaguar o Titan [64], básicamente debido a la topología de la red de interconexión.

Los modelos XC30 de Cray están compuestos en su nivel más bajo por blades, cada uno de los blades tiene 4 nodos de cálculo y un chip Aries; cada nodo de cálculo tiene 2 sockets, como se puede ver en el esquema presentado en la figura 1.15, mientras en la figura 1.16 se puede ver la implementación física, que sirve para ver la integración que se ha logrado.

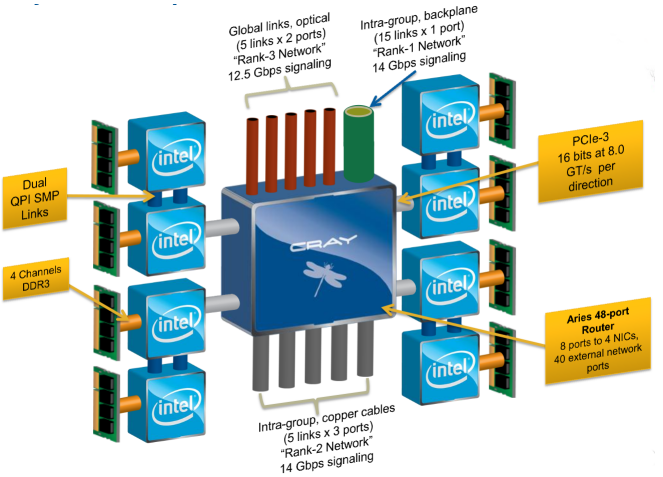


Figura 1.15: Esquema de un blade de un Cray XC30. Las comunicaciones de rango 1 comunican los blades de un chasis, las de rango 2 los blades de un grupo, y las de rango 3 entre distintos grupos.

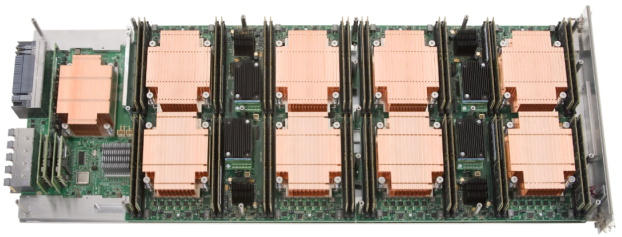


Figura 1.16: Un blade físico de un XC30, cada disipador cubre un chip multicore, el de la izquierda está sobre el chip Aries de comunicaciones.

Al siguiente nivel del XC30 se le llama chasis. Caben 16 blades, o 64 nodos de cálculo en un chasis. La red de rango 1 se encarga de comunicar los blades del chasis, comunicando los blades usando una red todos a todos a través de la placa que los interconecta, por lo que no hacen falta cables para comunicarlos.

En la figura 1.17 se muestra un grupo, que es el nivel superior al del chasis, estando formado por 6 chasis que ocupan 2 racks, teniendo en total 96 chips Aries (384 nodos de cálculo o 768 sockets). La red que comunica los chasis dentro de un grupo, de rango 2 en la terminología Aries, está formada por cables pasivos de cobre y comunica cada blade con todos los que están en el mismo slot en el resto de chasis. Por ejemplo, el chip Aries

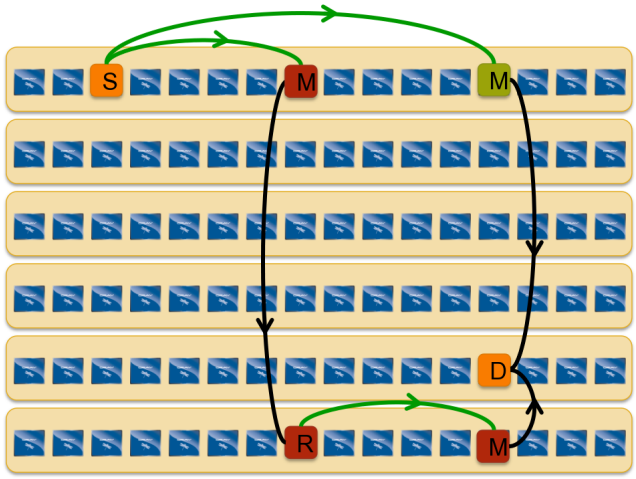


Figura 1.17: Ejemplo de canales de comunicación entre dos blades dentro de un grupo en un Cray XC30. Cada caja azul es un blade, cada rectángulo es un chasis, y los seis rectángulos, la imagen completa, forman un grupo. Las líneas verdes indican comunicaciones dentro del chasis, y las negras entre elementos del grupo.

que está en la posición 0 dentro de un chasis se conecta con los otros cinco chips Aries que están en la misma posición en los otros chasis, el que está en la posición 1 con todos los demás de la posición 1 de los otros cinco chasis del grupo y así sucesivamente. De esa forma la ruta mínima entre dos nodos de un grupo (que no estén en el mismo chasis) va a tener siempre dos saltos, y la ruta máxima, cuando la ruta mínima esté saturada, es de cuatro saltos, como se puede ver en la figura 1.17.

Finalmente tenemos las comunicaciones entre grupos, que se realizan a través de la red de rango 3. Cada grupo tiene 240 puertos de fibra óptica para conectarse con otros grupos [87], por lo que el máximo de grupos es de 241, pero se pueden hacer enlaces (*bonding*) entre las comunicaciones para tener más ancho de banda. En cualquier caso deberán de ser simétricas, es decir, tener las mismas conexiones con todos los grupos, y ser de todos a todos.

Una de las ventajas de la conexión Aries es que si detecta que un enlace está saturado puede decidir usar otro, aunque sea más costoso, para aumentar los anchos de banda entre dos nodos dados. Además el coste y la complejidad del cableado es mucho menor, ya que los tipos de conexiones más abundantes son muy baratos, mientras que el número de conexiones "caras" es mucho menor. Además, en general Aries proporciona latencias mucho menores, ya que se suelen necesitar menos saltos para conectar dos nodos

cualesquiera.

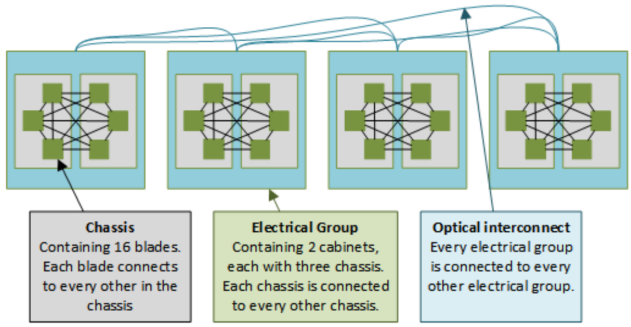


Figura 1.18: Esquema de un sistema XC30, los cuadrados verdes son chasis, cada uno conteniendo 16 blades, los cuadrados azules son 2 racks, conteniendo 6 chasis en total.

El sistema de ficheros está conectado a través de una red infiniband FDR, llamada SION II, a los nodos LNET que enrutan los mensajes de E/S entre los clientes y los servidores de Lustre [30]. Hay en total 9 servidores LNET en EOS, cada uno con 4 conexiones infiniband, lo que da un total de 36 conexiones, una a cada uno de los switches de Spider II. Cada nodo de cálculo tiene una lista con un router primario y otro de respaldo para el caso de que falle el primario, que sirven para comunicarse con los OSSs y los MDSs.

Otra ventaja de la red Aries respecto a la E/S es que, teóricamente, da igual donde se pongan los enrutadores LNET, siempre que estén repartidos entre los grupos, ya que el número de saltos para llegar a ellos es muy reducido [87]. Los enrutadores LNET usan un algoritmo de proyección donde cada uno tiene uno o más OSS asignados y funcionan como *proxies* [43].

1.4.5. Picasso

Para las evaluaciones de rendimiento de la librería FQbin se ha usado Picasso, un cluster ubicado en la Universidad de Málaga, formado por un conjunto heterogéneo de equipos de cálculo, que se pueden dividir en cuatro grupos de ordenadores. Por un lado 32 HP SL230G8 con 2 CPUs E5-2670, con un total 16 cores y 64 Gigabytes de RAM por nodo (en total 512 cores y 2 Terabytes de RAM). Por otro lado, 7 ordenadores HP DL980G7 con 8 procesadores E7-4870 con 80 cores y 2000 Gigabytes de RAM por nodo (en total 560 cores y 14 Terabytes de RAM). Además contiene 16 SL250G8 con 2 CPUs E5-2670 y dos tarjetas NVidia Tesla M2075, en total 256 cores, 1 terabyte de

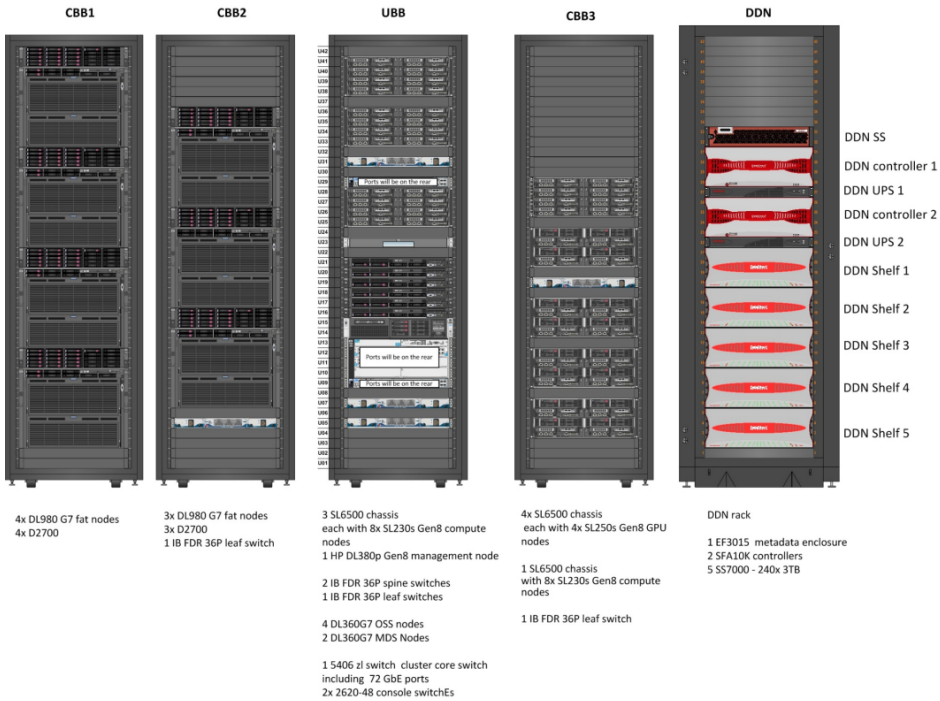


Figura 1.19: Ordenador Picasso de la UMA, elementos conectados por infiniband.

RAM y 32 tarjetas NVidia Tesla M2075. Todos estos equipos están comunicados por una red de interconexión infiniband FDR de 54 Gigabits por segundo con una topología fat-tree a través de 5 switches, 3 dispuestos como hojas, al que se conectan los servidores de cálculo y el sistema de almacenamiento, y los otros 2 ejerciendo de troncales para interconectar los otros 3. En la figura 1.19 se pueden ver los elementos anteriormente descritos, más el sistema de almacenamiento y los servidores del sistema de ficheros.

Además Picasso dispone de 41 nodos HP DL165G7, cada uno compuesto por 2 procesadores AMD Opteron 6176 (con 12 cores cada uno, 24 cores por nodo) y 96 Gigabytes de RAM. En total, estos equipos contienen 984 cores y 4 Terabytes de RAM conectados por ethernet Gigabit entre ellos y al resto de equipos.

El sistema de almacenamiento es un Lustre compuesto por una cabina DDN con 2 controladoras SFA10000 redundantes y 240 discos de 3 Terabytes cada uno repartidos entre cinco cajones de discos, con una capacidad bruta de 720 Terabytes, que, al estar configurados en RAID-6, tienen una capacidad neta de unos 500 Terabytes. Además hay una cabina DDN EF3015 con 10 discos de 600 Gigabytes a 15000 rpm para los

metadatos.

Los sistemas de ficheros Lustre son servidos desde 2 MDSs y 4 OSSs, con 24 OST cada uno de ellos con 10 discos (en RAID-6 8+2), y cada OSS sirve 6 OSTs. Hay 2 MDS para servir los metadatos, y un MDT para cada uno de los sistemas de ficheros.

2 Estudio de una interfaz de E/S en Chapel

En este capítulo presentamos la interfaz e implementación de una librería para realizar E/S paralela desde el lenguaje Chapel. Para ello vamos a introducir las distribuciones en Chapel y la librería GASNet que se usa para realizar las comunicaciones usando de forma óptima las distintas redes disponibles actualmente, entre otras las que ya hemos visto en la sección 1.4.

Además, se presenta una ampliación de las rutinas de comunicación de Chapel para mejorar, de manera transparente al programador, el rendimiento al realizar operaciones de movimiento de datos en arquitecturas con memoria distribuida.

2.1. Distribuciones en Chapel

Una de las características más importantes de Chapel es su soporte de distribuciones de datos. Además de las distribuciones de datos estándar (*Block, Cyclic, Block-Cyclic*) que están disponibles en las librerías del compilador, Chapel también permite al usuario definir distribuciones adicionales. Esto permite a programadores avanzados la creación de implementaciones de distribuciones de datos propias. Con estas distribuciones se permite trabajar con arrays de forma natural, a pesar de que estos pueden estar distribuidos en distintos nodos. Además Chapel permite especificar la forma en la que se distribuirá el array entre los nodos, la estrategia de recorrido paralelo, su disposición específica dentro de la memoria de los nodos, así como otros detalles. Para facilitar la tarea de trabajar con arrays distribuidos, Chapel incorpora el concepto novedoso de mapa de dominio (*domain map*) [16], que es una receta mediante la que se instruye al compilador

la forma de mapear la vista global de los datos en la memoria del nodo. Más concretamente, los mapas de dominio especifican cómo los índices y los elementos del array se mapean en los *locales* (nodos), cómo se almacenan en memoria y cómo se implementan operaciones como recorridos, accesos y particionamientos.

A los mapas de dominio se les puede poner nombre, ser manipulados y pasados a funciones. En particular, Chapel provee la palabra clave `dmapped` que permite mapear los índices del dominio a la arquitectura que designemos como objetivo usando una distribución específica. Un mapa de dominio se podrá llamar *layout* si solamente se va a realizar sobre un *locale*, o distribución si se va a realizar potencialmente sobre múltiples *locales*. Un *layout* determina cómo se almacena e itera sobre dominios y arrays, mientras una distribución además debe decidir cómo mapear los índices del dominio a los *locales*. Un resultado interesante de esa organización es que los arrays declarados sobre los dominios que han sido distribuidos usando la misma instancia de mapa de dominio están alineados, y por tanto la localidad puede ser aprovechada mejor.

En otras palabras, una distribución es una receta que Chapel usa para mapear los datos (y sus cálculos asociados) a los *locales* donde el programa se va a ejecutar [17]. Chapel incluye una librería de mapas de dominio estándar para soportar las distribuciones más comunes, a saber *Block, Cyclic, Block-Cyclic*. Además de las distribuciones que trae Chapel, ofrece soporte para distribuciones definidas por el usuario vía su DSI (*Domain map Standard Interface*) [16]. Esas distribuciones definidas por el usuario se desarrollan directamente en Chapel y pueden usar todas sus características (clases, iteradores,...), lo que simplifica la tarea del programador.

```
1 use BlockDist;
2 config const n=500;
3 var Space = {1..n,1..n};
4 var Dom = Space dmapped Block(boundingBox=Space);
5 var A: [Dom] real;
6 ...
```

Figura 2.1: Declaración en Chapel de una matriz distribuida en bloques.

En la figura 2.1 se puede ver una definición de un conjunto de datos distribuidos. La distribución se hace en tiempo de ejecución, y depende del número de *locales* que se vayan a usar. La distribución de datos es transparente, por lo que la forma de acceder y procesar los datos será óptima cuando se use correctamente. Para realizar la distribución, en la línea 1 indicamos que queremos cargar las definiciones de la distribución por bloques. En la línea 2 definimos una variable de configuración, $n = 500$, lo que significa que la podremos definir en la línea de comandos al invocar el programa. En la línea 3 definimos un dominio bidimensional de $n \times n$ que usaremos en la línea 4 para

crear un mapa de dominio. De esta forma, la *bounding box* definida por *Space* se reparte en porciones similares entre los *locales*. Finalmente, en la línea 5, se define un array, que almacenará datos de tipo *real* usando el dominio *Dom*, por lo que estará distribuido sobre los *locales* siguiendo una distribución en bloques. Por simplicidad, en la línea 4 coincide el dominio de la *bounding box* con los índices a almacenar, pero podrían ser distintos. En ese caso, sólo algunos índices de la *bounding box* contendrían elementos, como veremos en un ejemplo posterior en la figura 2.2.

Para comprender mejor la forma en la que se distribuyen los datos en Chapel, y por tanto lo que son los mapas de dominio, los dominios y los arrays, vamos a ver primero una visión más detallada de los conceptos explicados en los párrafos anteriores, y a continuación unos ejemplos.

2.1.1. Las distribuciones de datos en detalle

Hemos introducido los conceptos de array, dominio y mapa de dominio, que son los conceptos clave para tener datos distribuidos y paralelismo en las operaciones realizadas sobre esos datos. Vamos ahora a ilustrar el entorno de trabajo de Chapel para implementar mapas de dominio, usando la distribución Block estándar como ejemplo. El módulo de Chapel `BlockDist` define tres clases que indican: i) el mapa de dominio; ii) un dominio, mapeado usando los mapas de dominio; y iii) un array sobre ese dominio. En la figura 2.2 ilustramos un ejemplo de distribución de una array bidimensional sobre una malla de 2x2 *locales* implementado mediante 5 líneas de código Chapel. El código (arriba a la derecha) declara el mapa de dominio `myDist`, el dominio `myDom` y el array `M`. En tiempo de ejecución, estos tres componentes se representan usando descriptores globales, que son instancias de los descriptores de las respectivas clases, `Block`, `BlockDom` y `BlockArr`. Además existen descriptores *locales*, que son instancias de las clases `LocBLock`, `LocBlockDom` y `LocBlockArr`. Los descriptores *locales* son almacenados en cada `locale` y contienen el estado correspondiente al subconjunto del espacio de datos que esta almacenado en ese `locale`.

El objeto de distribución de bloque `myDist` almacena el estado global de la distribución: el campo `boundingBox` determina cómo se realiza la descomposición entre los locales; `targetLocDom` es el dominio que describe la red de *locales* que almacenará los datos distribuidos; el array `targetLocales[]` identifica los *locales* reales en uso; y el array `locDist[]` apunta a los descriptores `LocBlock` locales. Esos objetos se almacenan en cada *locale*, y definen las fronteras de las distribuciones *locales* en sus campos `myChunk`. De forma similar, para el dominio en *stride* bidimensional distribuido, `myDom`, el objeto `myDom` mantiene el conjunto de índices en el campo `whole=[3..10 by 3, 2..10 by 2]` y apunta a los cuatro descriptores `LocBlockDom` almacenados en los

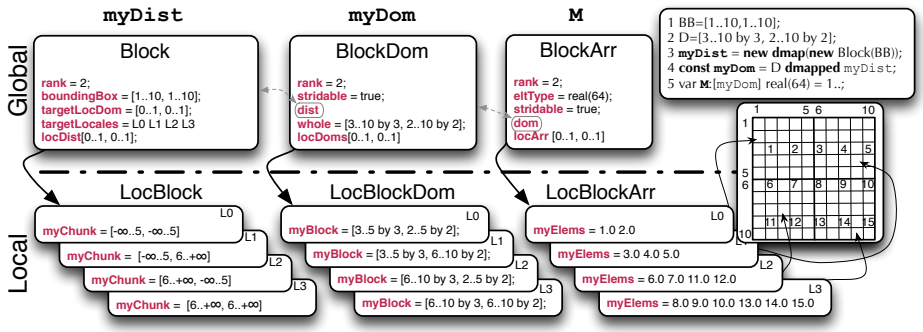


Figura 2.2: Ejemplo de distribución de bloques sobre cuatro *locales*.

locales. Cada uno de ellos a su vez, almacena el conjunto de índices local, `myBlock`, calculados por la intersección entre el dominio global `myDom.whole` con el correspondiente `myChunk` local.

Finalmente, el array `M` se declara sobre `myDom` y se inicializa con la secuencia `1..` (es decir, 1, 2, 3, 4, ...). El descriptor del array global es una instancia `BlockArr` que identifica el tipo de datos que se almacenarán realmente (en el ejemplo son números reales de 64 bits) y apunta a su descriptor de dominio global. Los elementos reales del array `M` se almacenan localmente en cada uno de los arrays *locales* `myElems`. Los arrays `myElems` están mapeados usando el *layout* estándar, `DefaultRectangular`, que será llamado array *DR* de ahora en adelante. Entre los arrays de Chapel, los arrays *DR* son los que representan la memoria física de forma más directa, es decir, un array *DR* se mapea en una única región contigua de memoria.

También podemos ver en la figura 2.2 los cuatro *locales*, llamados L0, L1, L2 y L3. El número de *locales* se puede especificar en el momento de lanzar el programa a ejecución usando el parámetro `-nl` (*number of locales*). Por ejemplo, el ejecutable resultante de compilar un programa Chapel se puede lanzar en 4 *locales* mediante el siguiente comando: `programa -nl 4`. Podemos ver las cinco sentencias Chapel que están en el programa del usuario en la esquina superior derecha de la figura 2.2, y debajo representamos gráficamente los datos que se almacenan en cada *locale* concreto. El código y la distribución resultante es la única parte de la figura 2.2 que es visible al programador. Es decir, los objetos globales de las clases `Block`, `BlockDom` y `BlockArr` así como las instancias de las clases *locales* `LocBLock`, `LocBlockDom` y `LocBlockArr` que se muestran a la izquierda en la figura, son transparentes al usuario y sólo se describen brevemente aquí con objeto de ilustrar las estructuras de datos que hay que manejar a la hora de optimizar las comunicaciones y la E/S en Chapel.

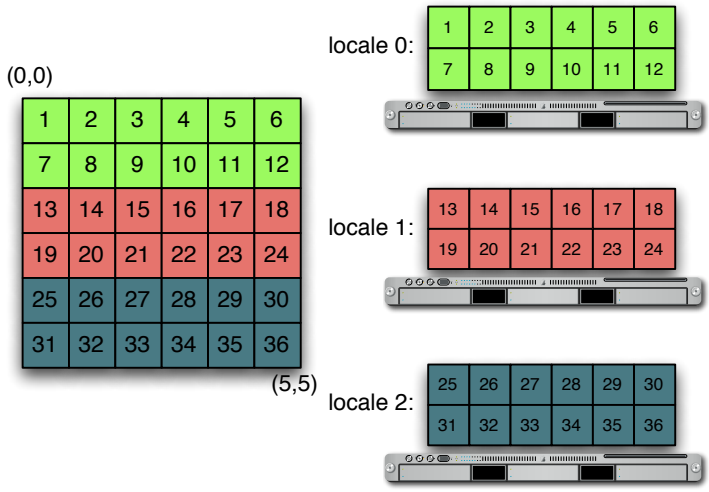


Figura 2.3: Distribución en bloques unidimensional de una matriz en tres *locales*.

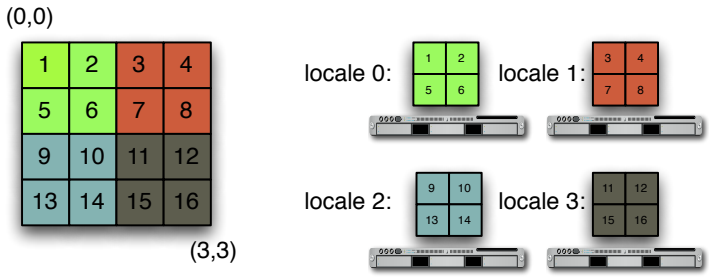


Figura 2.4: Distribución en bloques bidimensional de una matriz en cuatro *locales*.

Por defecto, una distribución *Block* de Chapel reserva subregiones de datos en cada uno de los *locales* (o nodos). En el caso de definir un array de datos de una dimensión, cada *locale* almacenará un bloque consecutivo del array. Sin embargo, al declarar una matriz (un array bidimensional), el particionamiento de los datos se realizará dependiendo del número de *locales* que se usen en tiempo de ejecución, y que se indica en la línea de comando con el argumento que se le pasa a `nl`. El comportamiento por defecto cuando el número de *locales* es primo, es el de configurar una malla unidimensional de *locales*, lo que resulta en una distribución por bloques de las filas de la matriz, tal y como

se muestra en la figura 2.3 para 3 *locales*. Por otro lado, cuando el número de *locales* no es primo, la configuración por defecto es la de una malla rectangular de $nl_1 \times nl_2$ (con $nl_1 \geq nl_2$ y $nl = nl_1 \times nl_2$). En la figura 2.4 mostramos un ejemplo para 4 `locales` que da lugar a una malla de 2×2 `locales` y a la correspondiente distribución bloques bidimensional. Aunque la configuración por defecto de la malla de `locales` es la que hemos descrito en este párrafo, es posible especificar manualmente dicha configuración con sólo un par de líneas de código Chapel. Por otro lado, hemos centrado los ejemplos anteriores en la distribución `Block`, pero el uso de las distribuciones `Cyclic` y `BlockCyclic` es similar.

2.1.2. Uso de las distribuciones de datos

```
1  use BlockDist;

3  config const n=500;
4  var DA   = {1..n,1..n};
5  var DB   = {1..2*n,1..2*n};
6  var Dom1 = DA dmapped Block(DA);
7  var Dom2 = DB dmapped Block(DB);

9  var A: [Dom1] real;
10 var B: [Dom2] real;

12 forall i in DA do A[i]=0;
13 forall i in DB do B[i]=0;

15 var D1 = {1..n by 4, 1..n by 3};
16 var D2 = {1..n, 1..n by 4};


19 A[D1] = B[D1];
20 A[D2] = B[D2];

22 forall i in D1 do
23   A[i]=B[i];

25 forall i in D2 do
26   A[i]=B[i];
```

Figura 2.5: Ejemplo de asignación parcial entre arrays en Chapel, primero con la operación = y después usando bucles forall.

En la figura 2.5 podemos ver un ejemplo de uso de arrays distribuidos por bloques. En las líneas 4 y 5 se definen dos dominios (o conjuntos de índices) `DA` y `DB` con distinto

tamaño. Usando esos índices, se definen dos dominios distribuidos (`Dom1` y `Dom2`) en las líneas 6 y 7, siendo los índices del dominio y de la distribución los mismos (`DA` y `DB`). A continuación, en las líneas 9 y 10 se definen los arrays `A` y `B`, que estarán distribuidos de acuerdo a los dominios sobre los que se definen (`Dom1` y `Dom2`). En la siguiente sección de código trabajaremos con estos arrays y recalcamos que el hecho de que estén distribuidos es transparente al programador. Primero se inicializan a cero los elementos de los arrays mediante bucles `forall` en las líneas 12 y 13. Estos bucles recorren los elementos del conjunto de índices correspondiente usando todo el paralelismo posible, que dependerá del número de *locales* y cores que se tengan disponibles. Posteriormente, en las líneas 15 y 16 se crean dos dominios nuevos, `D1` y `D2`, que contienen un subconjunto de los dominios `DA` y `DB`). Por ejemplo, el dominio `D1` sólo representa las filas con índice múltiplo de 4 mas 1 y las columnas múltiplo de 3 mas 1, entre 1 y n, debido a los *stride* `by 4` y `by 3` respectivamente. Las líneas 19 y 20 copian del array `B` al `A` sólo los elementos indicados por los dominios `D1` primero y `D2` después. El efecto de esas dos líneas es el mismo que el de ejecutar los dos bucles forall que están entre las líneas 22 y 26, que iteran sobre los elementos indicados por `D1` y `D2`.

Insisitimos en que estas operaciones de asignación se realizarán en paralelo dado que los arrays están distribuidos por bloques de acuerdo a las líneas 6 a 10 del código. Eso implica que los datos de origen y destino estén en *locales* diferentes, se implementarán internamente las comunicaciones necesarias entre dichos locales. De esta forma, aunque el resultado final de usar la sintaxis de la línea 20 es equivalente al de las líneas 22 y 23, el tiempo necesario para completar las operaciones no tiene por qué ser el mismo. Sabemos que el coste de realizar muchas transferencias de pocos datos es superior al de agregar todos esos datos en una sola transferencia. Por ejemplo, transferir un único bloque de un Megabyte de datos es mucho más eficiente que realizar un millón de transferencias de un byte cada una. Pues bien, aunque originalmente tanto la copia realizada con la operación "=" (en la línea 19), como la implementada con bucles forall (líneas 22 y 23), copiaban elemento a elemento, en este trabajo hemos implementado la agregación automática de datos [105], lo que reduce la sobrecarga que genera el realizar la copia elemento a elemento. Por tanto, en las últimas distribuciones del compilador Chapel, es más eficiente la asignación de arrays mediante asignación que iterando sobre los arrays con bucles forall.

En cuanto a la implementación de las comunicaciones entre *locales*, hay que tener en cuenta que distintas redes y protocolos proveen distintos mecanismos para realizar la copia de los datos entre nodos. Por ejemplo, se podrían transferir los datos a través de *sockets*, pero para ello habría que establecer una comunicación usando, por ejemplo, los protocolos TCP/IP, inicializar la comunicación, realizar la transferencia y después terminar la comunicación, con todo lo que ello conlleva a todos los niveles, creación de cabeceras TCP/IP, *overheads*, etc.

Para computación de altas prestaciones, es más habitual disponer de RDMA (*Remote Direct Memory Access*) en los nuevos dispositivos de comunicaciones. Este mecanismo permite realizar un acceso desde un nodo a la memoria de otro, sin que las CPUs ni los sistemas operativos tengan que intervenir en la copia, y así acelerar las comunicaciones, y sobre todo bajar la latencia. Sin embargo, las transferencias RDMA dependen del hardware de comunicaciones que se esté usando y de las capas de software que estén instaladas en el sistema. Por ejemplo, la red Gemini [3], instalada en todos los sistemas de última generación de Cray, el XT5, XT6, XE6, etc, provee dos tipos de RDMA. Por un lado, FMA (*Fast Memory Access*) proporciona operaciones remotas atómicas, así como operaciones de `get` y `put`, que se pueden llamar directamente desde modo usuario (sin intervención del OS). FMA consigue una menor latencia y la posibilidad de realizar varias comunicaciones simultáneamente, pero a costa de usar la CPU de los nodos. Por otro lado, para mensajes de más tamaño la red Gemini proporciona otro RDMA basado en un motor de transferencias de bytes hardware (llamado *Byte Transfer Engine*). Éste se accede a través de un modulo del kernel, y presenta como ventaja que las comunicaciones se realizan de forma asíncrona con las CPUs, las cuales quedan liberadas para otras tareas, y que suele tener un mayor ancho de banda en casi todos los casos. A cambio, sólo puede haber una comunicación activa por nodo, lo que obliga a encolar otras transferencias pendientes. De esta forma, a la hora de usar RDMA sobre una red Gemini habría que tomar la decisión de cual de las dos opciones que existen es recomendable usar para cada transferencia. Esto añadiría más complejidad a todas las decisiones de diseño que hay que adoptar al desarrollar una aplicación distribuida. Además, el programar una aplicación para, por ejemplo, usar la red Gemini, haría que la aplicación no funcionara en el resto de redes, y la portabilidad es uno de los factores más importantes a la hora de desarrollar una aplicación.

Encontramos por tanto dos factores encontrados, la portabilidad y la eficiencia. Para usar de la forma más eficiente posible la red de interconexión subyacente hace falta usar los comandos específicos que proporcione esa red, bien sea Infiniband, Gemini, Myricom, etc. La mejor solución es encontrar una librería que encapsule las posibilidades de las distintas redes, ofreciendo una interfaz común a todas ellas. MPI tiene implementaciones específicas para las distintas redes, pero es una librería de alto nivel que históricamente sólo ofrece funciones *two-sided* a los programadores. Las rutinas de comunicación *two-sided* presentan las siguientes desventajas: i) el envío y recepción de mensajes debe estar emparejado para iniciar la transferencia y la sobrecarga de emparejar el envío con la recepción puede ser superior al envío en sí, sobre todo para mensajes pequeños y medianos; ii) MPI garantiza el orden de los mensajes, mientras que muchas de las redes actuales no lo garantizan (esto puede crear una sobrecarga en el software para ofrecer esa garantía [122]); y iii) el requerimiento semántico de necesitar una participación activa por parte del código de la aplicación, tanto en el lado emisor como en el

receptor, hace que la latencia suela ser superior a los valores teóricos, ya que la aplicación debe estar atenta en ambos extremos de la comunicación a lo que sucede en la red, y en caso contrario el rendimiento de las comunicaciones caerá. En MPI 2.0 se añadió un interfaz de comunicaciones de un sólo lado (*one-sided*), pero incorpora limitaciones semánticas que hace complicado su uso en la práctica, como limitar los patrones de acceso que se pueden usar, y que obliga a recurrir a otra capa más alta del software [9]. La única librería que provee un interfaz de bajo nivel unificado, centrada en conseguir muchas pequeñas comunicaciones de baja latencia, y disponible para una gran cantidad de configuraciones hardware, es GASNet [10], y por ese motivo se eligió para proveer la capa de comunicaciones en Chapel. Trataremos con algo más de detalle la librería GASNet en la siguiente subsección.

2.2. Agregación de comunicaciones

Como se ha mencionado anteriormente, la implementación original de las asignaciones en Chapel era poco eficiente. El motivo principal era que las comunicaciones necesarias entre *locales* se realizan elemento a elemento, lo cual tiene un coste muy elevado cuando el volumen de datos es grande. En esta sección proponemos una implementación optimizada de la asignación de arrays Chapel que agrega automáticamente los datos de forma que se reduce el número de paquetes a enviar por la red de comunicaciones. Para esta implementación es importante primero entender con más detalle las rutinas de comunicación con *stride* que proporciona la librería GASNet, aspecto que tratamos a continuación.

2.2.1. La librería GASNet

Uno de los problemas a los que se enfrentan los programadores, los lenguajes y las librerías de programación distribuida son las múltiples capas de comunicaciones existentes hoy en día, cada una con sus protocolos y sus peculiaridades. Esto es así tanto desde el punto de vista del hardware (topología, velocidades, etc) como del software (APIs de programación distintas). Así, simplemente en los ordenadores descritos en la sección 1.4 podemos encontrar las redes de comunicaciones SeaStar 2+, Gemini, Aries, e InfiniBand (IB) con topología *fat-tree*, y cada una de ellas tiene su propia API para realizar comunicaciones. Para dar una visión más unificada del hardware de comunicaciones y permitir la portabilidad de los programas, lo normal es usar librerías que oculten dicho hardware, provean una interfaz completa, alto rendimiento, y al tiempo sean independiente del lenguaje y de los dispositivos. Un ejemplo de este tipo de librerías son MPI y GASNet. MPI es una librería de más alto nivel, pensada para ser usada directamente por el pro-

gramador de aplicaciones y que hasta la versión 2 no tenía comunicaciones *one-sided*. Aunque la versión 2 de MPI implementa comunicaciones *one-sided*, sigue siendo poco eficiente para las tareas que requieren muchas comunicaciones pequeñas [9].

Por otro lado, la librería GASNet [10], está pensada para implementar el sistema de comunicaciones en tiempo de ejecución de lenguajes PGAS (*Partitioned Global Address Space languages*) [140]. Es decir, GASNet está más orientado a ser la capa de comunicaciones a la que recurre un compilador, el cual generará automáticamente las llamadas a las rutinas de comunicaciones. GASNet es por tanto la librería de comunicaciones elegida en muchos lenguajes PGAS como son UPC, CoArray Fortran, Titanium o el mismo Chapel que abordamos en este trabajo.

La API que provee GASNet incluye funciones concretas para realizar transferencias de memoria a memoria para datos contiguos y para datos con saltos (en *stride*), existiendo versiones tanto bloqueantes como no bloqueantes. Como ejemplo, la función `gasnet_get(void *dest, gasnet_node_t node, void *src, size_t nbytes)` trae `nbytes` contiguos de la dirección `src` del nodo `node` y los pone a partir de la dirección `dest` del espacio de memoria local. Esta función es bloqueante, es decir, espera a que la transferencia esté completa para volver de la llamada.

En las operaciones no bloqueantes GASNet requiere una iniciación de la operación (generalmente un `put` o un `get`) y posteriormente una operación de sincronización para garantizar que la transferencia ha terminado.

Una operación muy importante para este trabajo que nos ocupa es la copia/transferencia de arrays con datos no contiguos o (*strided*), ya que, como veremos a continuación, nos permite agregar datos y reducir el número de comunicaciones. GASNet proporciona funciones para transferir varias subregiones no contiguas de memoria en una única llamada, lo que evita la sobrecarga de ir copiando subregión a subregión dentro de un bucle, y el consiguiente coste de abrir y cerrar muchas conexiones entre nodos.

```
void _gasnet_gets_bulk(void *dstaddr, const size_t dststrides[],
    gasnet_node_t srcnode, void *srcaddr, const size_t srcstrides[],
    const size_t count[], size_t stridelevels GASNETE_THREAD_FARG);

void _gasnet_puts_bulk(gasnet_node_t dstnode, void *dstaddr, const size_t
     dststrides[], void *srcaddr, const size_t srcstrides[], const
    size_t count[], size_t stridelevels GASNETE_THREAD_FARG)
```

Figura 2.6: Función de GASNet para realizar una transferencia *bulk-strided*.

Para las operaciones de copia con *stride*, GASNet proporciona las funciones

`gasnet_puts_bulk()` y `gasnet_gets_bulk()`[1]. Además existen las funciones equivalentes no bloqueantes añadiendo el sufijo `_nb` (*non-blocking*) al final. En la figura 2.6 se puede ver la interfaz que presentan las funciones *strided*. Esas funciones tienen los siguientes argumentos, tal y como se explica en [8]:

- `stridelevels` es el número de saltos distintos que se usarán en la transferencia.
- `count` es un array que indica el tamaño de las subregiones en cada dimensión (con un número de entradas igual a `stridelevels+1`, donde `count[0]` es el número de bytes de datos contiguos en la dimensión principal)
- `srcaddr/dstaddr` son las direcciones de los arrays fuente y destino en la memoria de los nodos fuente y destino
- `srcstrides/dststrides` son los arrays que contienen los saltos (*strides*) en cada dimensión en fuente y destino, respectivamente.
- `srcnode` o `dstnode`, dependiendo si la función es un get o un put, respectivamente, indica el *locale* origen de los datos o el *locale* destino de los datos.

Los saltos indicados en `srcstrides` y `dststrides` deben ser monótonamente crecientes y no especificar localizaciones que se solapen, ya que en otro caso la transferencia no se realizará correctamente.

En la figura 2.7 podemos ver un ejemplo de transferencia de un subdominio tridimensional de un array también tridimensional. El array origen está definido en el ejemplo como `A[13][12][11]` (11 columnas, 12 filas y 13 planos) y del destino como `B[16][15][14]` (14 columnas, 15 filas y 16 planos). Sin embargo, se transfiere un subdominio de $4 \times 3 \times 2$ elementos mediante la asignación `B[10..11,3..5,4..7] =A[6..7,6..8,6..9]`. Teniendo en cuenta que al igual que en el lenguaje C, Chapel almacena los arrays por filas [2], la parte inferior de la figura 2.7 muestra la vista linearizada del array tridimensional. Si suponemos que la matriz `A` está en un *locale* y la matriz `B` en otro diferente, podemos usar la función `gasnet_puts_bulk()` del local que contiene `A` al que contiene `B`. Como vemos en la figura, para el primer plano, habrá que mover 3 bloques de 4 elementos cada uno, separados por un *stride* que permite saltar de una fila a la siguiente (11 elementos en `A` y 14 elementos en `B`. Para el segundo plano, primero hay que desplazarse al plano siguiente (saltar 11×12 elementos en `A` y 14×15 elementos en `B`) y repetir la operación del plano anterior. Esos *strides* están indicados en los arrays

[1] La "s" que sigue a `get` y `put` indica que son las versiones *strided* de las funciones `get` y `put`.

[2] Organización *row-major*, es decir, elementos consecutivos de las filas se almacenan de forma consecutiva en memoria.

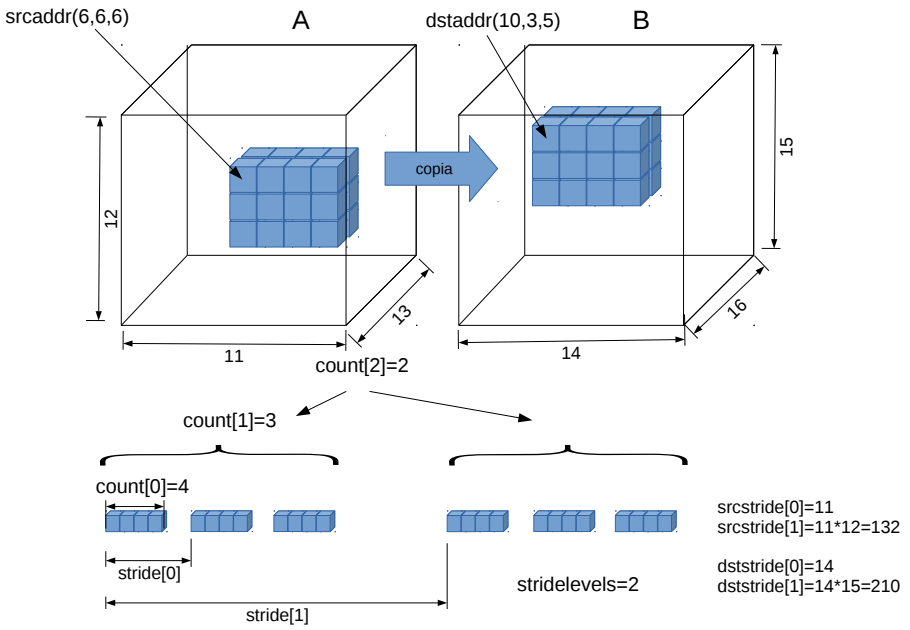


Figura 2.7: Ejemplo de copia entre dos arrays tridimensionales usando dos niveles de salto (`stridelevels`=2), `count`={4,3,2}, `srcstride`={11,11×12} y `dststride` ={14,14×15}.

`srcstride`= {11,11×12} y `dststride`= {14,14×15}. El array `count`= {4,3,2} indica en su primera componente que los bloques consecutivos tienen 4 elementos, que hay 3 bloques en cada plano y por último que vamos a usar 2 planos.

Existen implementaciones concretas de las funciones `gasnet_puts_bulk()` y `gasnet_gets_bulk()` para muchas arquitecturas de forma que su funcionamiento sea lo más óptimo posible. Por ejemplo, para las redes Gemini de Cray se utiliza una técnica de envío de doble-buffer en la que en el nodo local se van empaquetando los bloques no consecutivos en un buffer de envío al tiempo que se está enviando el buffer que se preparó con anterioridad. De esta forma se reduce el número de comunicaciones y se aprovecha mejor el HW de RDMA. En las redes Gemini, la implementación optimizada aún se considera beta, por lo que hay que usar variables de entorno para activar las optimizaciones de copia en *stride*. Específicamente hay que definir las variables `GASNET_VIS_AMPIPE=1`, que activa la optimización de doble-buffer explicada anteriormente, y `GASNET_VIS_REMOTECONTIG=1`, que habilita el empaquetamiento para datos que están dispersos localmente, pero que en el nodo remoto serán almacenados de forma continua.

2.2.2. Ampliación del lenguaje Chapel para usar funciones avanzadas de GASNet

Las funciones de copia en stride se usarán intensivamente en la librería de E/S que proponemos para Chapel. El objetivo es realizar la agregación de los datos en los distintos *locales* y reducir así el número de mensajes que se transfieren por la red. A cambio, el tamaño de los mensajes será mayor pero debido a que el inverso de la latencia es siempre menor que el ancho de banda, las comunicaciones agregadas son ventajosas cuando el volumen de datos es suficiente.

Para usar las funciones de comunicación agregada en Chapel es necesario implementar en el run-time de Chapel las rutinas que hacen de interfaz entre los arrays de Chapel y los arrays de C que se usan en GASNet[3]. Las funciones que proponemos son `chpl_comm_gets` y `chpl_comm_puts` [105], que han sido diseñadas para tener el mismo interfaz (los mismos argumentos) que las funciones de GASNet. La funcionalidad principal de estas funciones es convertir los arrays de Chapel a arrays de C y posteriormente llamar a la función GASNet correspondiente.

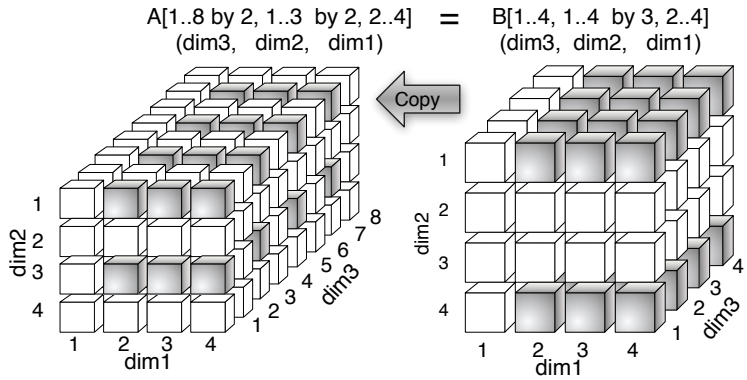


Figura 2.8: Ejemplo más detallado de transferencia entre dos arrays tridimensionales con dos niveles de salto (`stridelevels`=2), `count`={3,2,4}, `srcstride`={12,16} y `dststride`={8,32}.

En la figura 2.8 podemos ver otro ejemplo más detallado de una transferencia agregada en la que ahora se usarán las funciones `chpl_comm_gets` o `chpl_comm_puts` (dependiendo de si el comando se ejecuta en el destino o en el origen, respectivamente). Los argumentos a pasar a la función son los mismos que en las funciones equivalentes de

[3]La librería GASNet está implementada en C

GASNet y que fueron explicados en la sección anterior. En este ejemplo se realiza una transferencia agregada desde el array B, definido como `B[4][4][4]`, los datos del subdominio `B[1..4, 1..4 by 3, 2..4]`. El array destino es `A[8][4][4]`, suponemos que está en otro *locale*, y se escribirán sólo las posiciones `A[1..8 by 2, 1..3 by 2,2..4]`. En la figura, los datos que hay que leer de `B` y copiar en `A` están identificados como cubos con fondo gris oscuro. La versión en memoria de esos arrays contiene los elementos a transferir en 8 bloques de 3 elementos consecutivos, pero separados por *strides* diferentes en los arrays `B` y `A`. Con todo eso, los argumentos de llamada a las funciones `chpl_comm_gets` o `chpl_comm_puts` deben ser: `stridelevels`=2), `count` ={3,2,4}, `srcstride`={12,16} y `dststride`={8,32}. En este ejemplo, `dstaddr` debe apuntar al elemento `A[1,1,2]` y `srcaddr` a `B[1,1,2]`.

El código en la figura 2.9 muestra la primitiva `chpl_comm_gets` que ahora usa el compilador de Chapel. La primitiva `chpl_comm_puts` es análoga. Como describiremos en la siguiente sección, hemos modificado el compilador de Chapel para que inserte estas primitivas de forma automática cada vez que hay una asignación entre arrays distribuidos. También se deben insertar llamadas a funciones que calculen los argumentos que hay que pasar a dichas funciones. Las llamadas a `array_get` que vemos en los argumentos de la primitiva devuelven un manejador o puntero al array Chapel correspondiente.

```
__primitive("chpl_comm_gets",
  __primitive("array_get",destr, buffer._value.getUnshiftedDataIndex(0)),
  __primitive("array_get",dststr,dststrides._value.getDataIndex(0)),
  srcnode,
  __primitive("array_get",src,
    privArr.locArr[locFrom.id].myElems._value.getUnshiftedDataIndex(0)),
  __primitive("array_get",srcstr,srcstrides._value.getDataIndex(0)),
  __primitive("array_get",cnt,count._value.getDataIndex(0)),
  stridelevels);
```

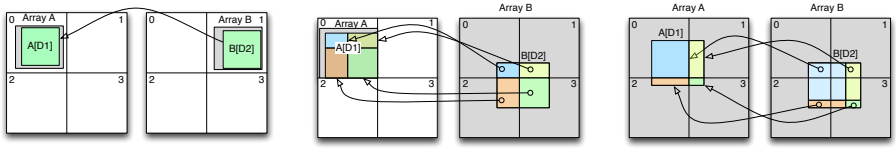
Figura 2.9: Rutina del run-time de Chapel que genera comunicaciones agregadas.

2.2.3. Implementación de la agregación de datos

El siguiente paso consiste en reducir en número de transferencias de datos para asignaciones de arrays siempre que estos usen la distribución Block o Cyclic. Aprovechando las características OOP (*Object Oriented Programming*) de Chapel, se sobrecargó el operador = con una función especializada que proporciona la implementación optimizada.

Como los arrays distribuidos por bloques se implementan usando arrays *DR* (*Default*

Rectangular) que son arrays que están almacenados de forma local en un sólo nodo, hemos implementado primero la comunicación masiva (*bulk*) para realizar asignaciones entre dos arrays *DR*. Sobre eso hemos implementado la asignación entre un array *DR* y uno *BD* (*Block Distributed*), y a continuación la asignación entre dos arrays *BD*. Ilustramos esa organización usando cuatro *locales* en la figura 2.10.



a) DR=DR b) DR=BD c) BD=BD

Figura 2.10: A[D1]=B[D2] cuando A y/o B son arrays *DR* (*Default Rectangular*) o *BD* (*Block Distributed*).

La figura 2.10 a) muestra la asignación `A[D1]=B[D2]` donde `A` y `B` son arrays *DR*, `A` reside en el *locale* 0 y `B` en el *locale* 1. Aquí el programa, en tiempo de ejecución, tiene que calcular los argumentos y ejecutar una sola llamada a `chpl_comm_gets` (si la sentencia se ejecuta en el *locale* 0), o `chpl_comm_puts` (si la sentencia se ejecuta en el *locale* 1), tal y cómo se describe en la sección 2.2.2.

Si `B` es un array del tipo *BD* (distribuido por bloques) sus elementos estarán almacenados en varios arrays de tipo *DR*. Estos arrays son los campos `myElems` de los descriptores locales `LocBlockArr`, que están contenidos en `B`.

Si `A` es un array *DR*, entonces la operación `A[D1]=B[D2]` se compone de varias operaciones del tipo `DR=DR`. Por ejemplo, en la figura 2.10 b), `B[D2]` está repartido entre cuatro *locales*, mientras `A` está almacenada completamente en el *locale* 0. En ese caso se divide `A[D1]` en cuatro regiones, de $A[D1_0]$ a $A[D1_3]$, de forma que $A[D1_i]$ es el destino de los elementos de `B[D2]` que están almacenados en el *locale* i, con $i \in 0..3$. Para cada `i`, `src` representa el trozo de `B.locArr[i].myElems` que corresponde a `B[D2]` en el locale `i`. La función *bulk* `gets` o `puts` que realiza la asignación en el locale 0 está implementada internamente con un `memcpy`.

Finalmente, si `A` y `B` son ambos arrays que siguen una distribución por bloques, entonces dividiremos el problema en varios del tipo `DR=BD`, ya que el array de destino `BD` es simplemente un conjunto de arrays `DR`. En la figura 2.10 c) el array `A` está almacenado en cuatro arrays DR locales. Por tanto la asignación `A[D1]=B[D2]` se convierte en:

```
1 forall i in 0..3 do
2   on A.locArr[i] do //en el local i ejecuta lo siguiente:
3    A.locArr[i].myElems[dest] = B[src] //DR=BD
```

donde `src` y `dest` son las subregiones del dominio fuente y destino que indican que porciones del array van a ser asignados en cada locale. En este ejemplo `A.locArr[0].myElems` necesita cuatro regiones de `B`, que están almacenadas en cuatro locales. `A.locArr[1].myElems` sólo necesita información de los *locales* 1 y 3. La asignación a `A.locArr[3].myElems` es realizada cómo una operación `memcpy` en este ejemplo.

Optimizamos los movimientos de datos al realizar asignaciones entre distribuciones bloque y cíclica, y entre cíclica y bloque de una forma similar, convirtiéndolas en llamadas a `chpl_comm_gets/puts`. Todas esas redistribuciones de datos son casos especiales de la copia de una región rectangular de n-dimensiones a otra con una forma distinta. Por ejemplo, para realizar una redistribución de los datos de un bloque a una distribución cíclica, cada locale fuente determinará el trozo de datos que enviará a cada *locale* de destino. Para cada trozo y para cada *locale* de destino, el locale fuente llamará `chpl_comm_puts` con los argumentos apropiados. En general, esto conduce a una comunicación de todos a todos. El caso de la asignación entre cíclica y bloques es similar, con la diferencia de que la asignación se pedirá desde los *locales* de destino llamando a la función `chpl_comm_gets`.

2.2.4. Cálculo de las subregiones

Como hemos visto en la sección anterior, durante la asignación de las subregiones del array, hay que calcular distintas regiones dentro de cada trozo, como las subregiones $D1_i$ y `src`. Para hacerlo en el caso de la asignación `A[DA]=B[DB]`, primero necesitamos una correspondencia desde el dominio del destino, `DA`, al dominio de la fuente, `DB`.

Esto se puede describir de manera formal de la siguiente forma. Sea:

`DA=[`la_1`..`ha_1 `by` sa_1`, ...,`la_n`..`ha_n `by` sa_n`]`

`DB=[`lb_1`..`hb_1 `by` sb_1`, ...,`lb_n`..`hb_n `by` sb_n`]`

donde `n` es el número de dimensiones (*rank*) de los dominios `DA` y `DB`. Un vector con todos los *strides*, sa_i, se puede obtener por `A._value.dom.whole.stride` y de forma similar para `B`. Sean `[`a_1`,`a_2`,...,`a_n`]` y `[`b_1`, `b_2`,...,`b_n`]` las coordenadas de los dominios `A` and `B` respectivamente. Definimos la función biyectiva `m:DB`$\rightarrow$`DA` de forma que `(`a_1`,`a_2`,...,`a_n`)= m(`b_1`,`b_2`,...,`b_n`)` donde:

$$a_i = la_i + sa_i \times (b_i - lb_i)/sb_i,\ i \in 1..n$$

La función inversa correspondiente `m`$^{-1}$`:DA`$\rightarrow$`DB`, resulta en:

$$(b_1, b_2, ..., b_n) = m^{-1}(a_1, a_2, ..., a_n)$$

```
config const n=500;
var D1 = [1..n,1..n];
var D2 = [1..2*n,1..2*n];

var A: [D1] dmapped Block(D1) real;
var B: [D2] dmapped Block(D2) real;

var DA = [101..200 by 2, 51..200 by 3];
var DB = [201..700 by 10, 301..600 by 6];

A[DA] = B[DB];
```

Figura 2.11: Código Chapel para ilustrar las funciones de correspondencia.

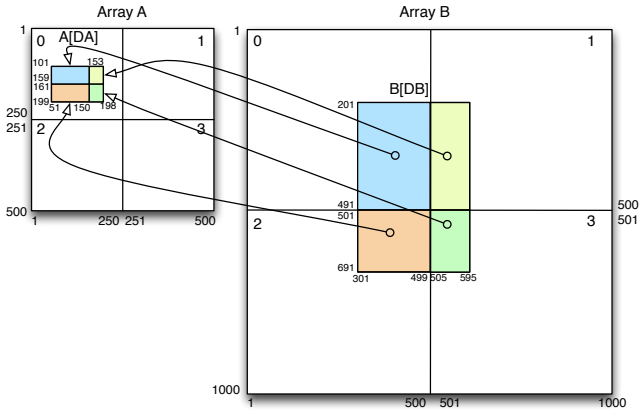


Figura 2.12: Ilustrando la asignación A[DA]=B[DB] de la figura 2.11 cuando se usan 4 locales.

donde:

$$b_i = lb_i + sb_i \times (a_i - la_i)/sa_i,\ i \in 1..n$$

Para ilustrar cómo se usan esas funciones de correspondencia, m y m^{-1}, vamos a basarnos en una versión simplificada bidimensional del ejemplo que venimos estudiando, que podemos ver en la figura 2.11. Si ese código se ejecuta en cuatro *locales*, la figura 2.12 ilustra la distribución y las subregiones que hay que mover, donde podemos ver que `A[DA]` está en el locale 0 mientras `B[DB]` está distribuida entre los cuatro locales.

Cuando se ejecuta la asignación `A[DA]=B[DB]`, se crean dos nuevos alias de los arrays originales, `A'=A[DA]` y `B'=B[DB]`. Ahora, para el caso de DR=BD, como ya hemos dicho, tenemos que ejecutar el siguiente bucle:

```
1 forall i in 0..3 do
2   on A' do // se ejecuta en el \textit{locale} que contiene A'
3     A'[D_i] = B'.locArr[i].myElems //DR=DR
```

De forma que el problema a resolver ahora es el cálculo de las subregiones D_i, $i \in \{0..3\}$. Esto se realiza usando la función de correspondencia m. Por ejemplo, para i=0, el subdominio de B almacenado en el *locale* 0 viene dado por B'.dom.locDoms[0].myBlock = [201..491 by 10, 301..499 by 6]. Entonces podemos obtener D_0 usando la función m para calcular las coordenadas de finalización de la región de destino:

[$lxa_1..hxa_1$ by $2, lxa_2..hxa_2$ by 3]:

$$\begin{aligned} lxa_1 &= 101 + 2 \times (201 - 201)/10 = 101 \\ hxa_1 &= 101 + 2 \times (491 - 201)/10 = 159 \\ lxa_2 &= 51 + 3 \times (301 - 301)/6 = 51 \\ hxa_2 &= 51 + 3 \times (499 - 301)/6 = 150 \end{aligned}$$

De forma que obtenemos D_0=[101..159 by 2, 51..150 by 3]. Y similarmente:

- D_1= [101..159 by 2, 153..198 by 3]
  = M(201..491 by 10, 505..595 by 6)
- D_2= [161..199 by 2, 51..150 by 3]
  = M(501..691 by 10, 301..499 by 6)
- D_3= [161..199 by 2, 153..198 by 3]
  = M(501..691 by 10, 505..595 by 6)

donde M convierte los límites alto y bajo de su rango de argumentos usando la función m.

Como ya hemos dicho, si tanto `A` como `B` son arrays distribuidos en bloques, podemos dividir el problema en varios del tipo `DR=BD` como hemos mostrado en la figura 2.10, pero ahora necesitamos la función m^{-1}. Así, si la función de asignación es `A[DA]=B[DB]`, se crean dos nuevos alias de la misma forma que anteriormente: `A'=A[DA]` y `B'=B[DB]`, y hay que ejecutar el siguiente código:

```
1 forall i in 0..3 do
2   on A'.locArr[i] do // se ejecuta en el \textit{locale} i
3     A'.locArr[i].myElems = B'[D_i] //DR=BD
```

de forma que tenemos que encontrar las subregiones D_i, $i \in \{0..3\}$. Pero ahora tenemos que usar la función m^{-1} para poder calcular los subdominios D_i de B que corresponden a cada dominio local A'.dom.locDoms[i].myBlock. Para el ejemplo anterior, pero con `DA=[101..400 by 2, 51..350 by 3]` y `DB=[201..500 by 2, 151..450 by 3]` y cuatro *locales*, tenemos que `A'.dom.locDoms[0].myBlock= [101..250 by 2, 51..250 by 3]` y D_0`= [201..350 by 2, 150..350 by 3]`. De forma similar para D_i y para el resto de casos.

Para profundizar en otros detalles de implementación el lector puede acceder al código fuente, disponible en: http://chapel.cray.com.

Esta implementación de la agregación es útil en multitud de casos. Tiene la ventaja añadida de no requerir de modificaciones en el código fuente para usarla. Por ejemplo, la asignación `A=B` usará la agregación si es necesario. Mostramos a continuación un ejemplo de uso de la agregación de comunicaciones en el algoritmo PARACR. Además, la usamos también en la librería de E/S paralela para el lenguaje Chapel que describiremos en la sección 2.3.

2.2.5. Evaluación del rendimiento usando el algoritmo PARACR

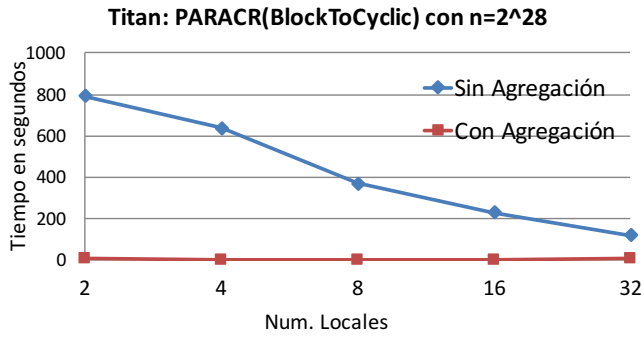


Figura 2.13: Tiempo en segundos del algoritmo PARACR en Titan, sin/con agregación de datos.

La Reducción Cíclica Paralela (*Parallel Cyclic Reduction*), llamada PARACR en [61], es un algoritmo bien conocido para la resolución de sistemas tridiagonales de ecuaciones. Funciona en dos fases: sustitución y solución. Para un problema de tamaño n, la fase de sustitución está compuesta de $O(log\ n)$ pasos y en cada paso se realizan $O(n)$ operaciones de tipo *butterfly*. Este algoritmo requiere de una distribución bloques en las primeras etapas de la computación, pero para explotar mejor la localidad es necesario

cambiar a la distribución cíclica para las últimas etapas del algoritmo. Este cambio de distribución "Block to Cyclic" es inmediato en Chapel ya que basta con realizar la asignación `A=B` con `A` distribuido cíclicamente y `B` distribuido por bloques. Sin embargo, en la implementación original de Chapel con comunicaciones no agregadas el tiempo de redistribución consumía una gran parte del tiempo total de ejecución.

En la figura 2.13 mostramos el tiempo de ejecución en segundos de la redistribución de bloques a cíclica en varios nodos del supercomputador Titan (descrito en el apéndice A.1.2). La red de comunicaciones usada es Gemini y el número de *locales* va desde 2 hasta 32. En este código hay cuatro arrays, cada uno de ellos de tamaño $n = 2^{28}$, que tienen que ser redistribuidos. La agregación aporta un aumento del rendimiento significativo, sobre todo con un bajo número de locales. Cuando el número de *locales* aumenta el tiempo de redistribución crece muy ligeramente, debido a que el número de mensajes *bulk* por locale es $O(num.locales)$. Sin embargo, en el caso de no usar agregación el tiempo de redistribución baja al aumentar el número de locales. Esto es así porque el número total de mensajes punto-a-punto cuando no hay agregación es igual a n (un mensaje por cada elemento del array), pero el número de mensajes por *locale* es $O(n/\#locales)$, por lo que al aumentar el número de *locales* disminuye el número de mensajes por locale. Con 2 *locales*, usar agregación proporciona una aceleración de 264x y con 32 *locales* la aceleración sigue siendo de más de un orden de magnitud (16x).

En [105] se muestran resultados similares para un algoritmo FFT radix-4. Las ventajas de la agregación se han evaluado adicionalmente con otros algoritmos como la transposición de matrices, y algoritmos tipo "*stencil*" en los que hay que mover bloques de datos con frecuencia. En todos los casos, la agregación de mensajes ha resultado ser una optimización de gran valor para reducir el tiempo total de ejecución.

2.3. Librería de E/S en Chapel

Como ya hemos visto anteriormente, uno de los retos de la HPC (*High Performance Computing*) es que el cuello de botella de las aplicaciones está pasando de ser el procesamiento de los datos a ser la disponibilidad de los datos. Muchas aplicaciones HPC acceden de forma intensiva a datos, bien leyéndolos y/o escribiéndolos. Esto hace necesaria la optimización de la E/S para obtener mejores rendimientos. Por razones de portabilidad y de representación de los datos, los sistemas HPC se despliegan a menudo con una amplia variedad de pilas software para manejar la E/S. En la capa de arriba, las aplicaciones científicas realizan la E/S a través de librerías *middleware* como Parallel NetCDF [67], HDF [89] o MPI IO [120]. En cualquier caso, todas las librerías anteriores siguen dejando visibles al programador detalles de bajo nivel, que no son apropiados para una computación de alta productividad.

Por otro lado, en el fondo de la pila de software usada para acceder a los datos, los sistemas de ficheros paralelos sirven directamente las peticiones de E/S dividiendo los bloques de los ficheros entre múltiples dispositivos de almacenamiento. Obtener un buen rendimiento colectivo de E/S cuando se tienen muchos procesos sobre esas capas de software es una tarea compleja, que requiere no sólo conocer los patrones de acceso a los datos por parte de los procesos, sino además comprender cómo funciona la pila de software completa, especialmente el comportamiento del sistema de ficheros subyacente. Creemos que esta es una tarea muy pesada de la que es mejor abstraer al programador. En otras palabras, un lenguaje paralelo debería proveer las construcciones de alto nivel adecuadas para realizar las operaciones de E/S, mientras el código en tiempo de ejecución (runtime) sería el encargado de optimizar el rendimiento y de tratar con el soporte de *middleware* para el correspondiente sistema de ficheros. Para ello, el runtime se puede apoyar en pistas (*hints*) sobre la localidad dadas por los usuarios (por ejemplo, la distribución de datos en los nodos).

En esta tesis se ha realizado la primera implementación de operaciones E/S paralelas en el contexto del lenguaje de alta productividad Chapel. Actualmente Chapel soporta operaciones básicas de E/S. Nuestras nuevas funciones de E/S paralelas se han implementado usando características estándar de Chapel, tomando las funciones básicas de Chapel como capa intermedia y Lustre como el sistema de ficheros objetivo. Conforme evolucionó el trabajo el interfaz que presentamos se hizo más simple, pero igual de potente, con lo que la productividad también mejoró. Además se estudiaron distintos algoritmos para su implementación, las cuales se compararon con un código equivalente en MPI IO, usando la librería ROMIO, resultando el código de Chapel no sólo mucho más simple, si no además un poco más rápido.

A la hora de construir una interfaz de E/S hay que separar el diseño de la interfaz de

la implementación de la E/S paralela. El diseño de la interfaz afecta a la productividad del programador y la implementación al rendimiento que vamos a obtener del sistema informático concreto que usemos.

En el diseño de la interfaz nos basaremos en las distribuciones de datos soportadas por el lenguaje Chapel. Más concretamente, nuestra propuesta se apoya en una distribución `BlockDist` aumentada con nuevas clases de E/S, las cuales se usan gracias a la sobrecarga del operador igual ("=").

2.3.1. Diseño de la interfaz

Como hemos visto en 1.1.3, los sistemas de ficheros suelen usar el estándar POSIX [127] como interfaz de acceso. POSIX provee un conjunto de funciones básicas con las que se puede acceder de forma unívoca y total a los datos de los ficheros, pero no proporciona una interfaz explícita para el acceso paralelo. Aunque han existido propuestas de extensión del interfaz POSIX, estas no se han materializado [124].

Por tanto los usuarios (programadores) históricamente han usado las funciones POSIX aunque estuvieran desarrollando una aplicación paralela. Generalmente la solución adoptada ha consistido en ignorar el paralelismo de E/S a nivel de aplicación y limitarse a leer/escribir desde un único *thread*. En estos casos, este `thread` se hace responsable de repartir/agregar los datos hacia/desde el resto de *threads*. Cuando estamos realizando lecturas, eso tiene como inconveniente que hay que mover los datos desde el almacenamiento a un *locale* y después desde ese *locale* distribuirlo a los sitios donde realmente se necesita. En el caso de la escritura el problema es simétrico, con el problema añadido del mantenimiento de la coherencia, lo que provoca que el rendimiento de la E/S no sea óptimo.

Hay que tener en cuenta que a la hora de implementar una librería no basta con ofrecer una interfaz productiva desde el punto de vista del programador, sino que es importante que también se usen de forma óptima los recursos existentes. Esto se complica si existe una gran variedad de sistemas operativos, sistemas de ficheros y redes de interconexión. Sin embargo, aunque originalmente los sistemas eran muy distintos unos de otros [33][88], y existía un ecosistema bastante variado de sistemas operativos, sistemas de ficheros, redes de interconexión, etc, con el tiempo han ido desapareciendo la mayoría [88]. El motivo de esta convergencia en unos pocos sistemas operativos y sistemas de ficheros es la introducción de componentes de consumo (*commodity HW* y *commodity SW*) en la producción de los clusters, con la consiguiente reducción de los costes y homogeneización de los sistemas.

Originalmente diseñamos una interfaz clásica para realizar la E/S en Chapel, tal y

```
1 var Space =  {0..n, 0..n};
2 var Dom = Space dmapped Block(Space);
3 var A: [Dom] real;
4 ...
5 var outfile = new file(fname,FileAccessMode.write);

6 outfile.open();
7 outfile.write(A);
8 outfile.close();
9 ...
```

Figura 2.14: Diseño original del interfaz para realizar E/S en Chapel.

como se puede ver en la figura 2.14, y como se explica en [78]. En las tres primeras líneas del código de esa figura se declara un array `A` distribuido por bloques. En la línea 5 se crea un objeto `outfile` de tipo `file` que se configura con el nombre del fichero y el modo de acceso (escritura en este caso). Cuando se realiza una operación de lectura o escritura sobre ese array distribuido, como la mostrada en la línea 7, se ejecutará código de escritura en ficheros en los distintos *locales* en los que la matriz `A` esté almacenada. Los algoritmos que se usan para realizar el acceso paralelo al sistema de almacenamiento secundario se detallan en la sección 2.3.2.

```
1 const Space = {0..n,0..n};
2 var Dom = Space dmapped Block(Space);
3 var A: [Dom] int;
4 ...
5 // Declara un fichero Chapel
6 var File = new ArrayIO() ;
7 File.filename="file.data";
8 File.accessMode=read-write;

10 A=File;  //Lectura del fichero
11 ...
12 File[1..n]=A[1,1..n]  //Escritura en el fichero
```

Figura 2.15: Nuevo interfaz para ficheros distribuidos en Chapel.

Posteriormente se propuso una interfaz alternativa como la mostrada en la figura 2.15. Esta posibilidad propone evitar los tradicionales métodos `open`, `read`, `write` y `close` y a cambio sobrecargar el operador "=". Se usa un objeto de la clase ArrayIO (línea 6) para representar el archivo, del cual se puede configurar el nombre de fichero y el modo de acceso con las variables de clase `filename` y `accessMode`, como se ve en

las líneas 7 y 8. También se podría haber usado el constructor de la clase para configurar el objeto (por ejemplo: `var File = new ArrayIO("file.data",read)`). A partir de ese momento, una asignación con `File` en la parte derecha (línea 10) lee del fichero y escribe en el array que aparece en la parte izquierda. Como se ve en la línea 12, es posible especificar dominios, en este caso para escribir sólo en una región del fichero.

Otra posibilidad, que consideramos más elegante, consistiría en crear una distribución específica, `ioFile`, para representar arrays mapeados en el sistema de ficheros. En la figura 2.16 mostramos en la línea 5 la declaración de un array `File` que está mapeado usando una distribución nueva que hemos llamado `ioFile`. En este caso, la instrucción Chapel `dmapped` no mapea un array sobre un conjunto de *locales* sino sobre un archivo almacenado en el sistema de ficheros, especificando un tamaño máximo, `{1..n*n}`, y un nombre de fichero, `file.data`. En las líneas 7 y 9, se muestran dos posibles operaciones de lectura y escritura en fichero, respectivamente.

```
1 const Space = {0..n,0..n};
2 var Dom = Space dmapped Block(Space);
3 var A: [Dom] int;

5 var File = {1..n*n} dmapped ioFile({1..n*n}, "file.data");
6 ...
7 A[1..2,1..5]=File[1..10]; //Lectura de File
8 ...
9 File=A; //Escritura en File
```

Figura 2.16: Otro posible interfaz para ficheros distribuidos en Chapel.

Todos los interfaces anteriores son bloqueantes, es decir, no vuelven hasta que la operación de E/S ha terminado. Gracias al soporte que proporciona Chapel para explotar paralelismo, es muy sencillo ejecutar las operaciones de E/S de forma no bloqueante. En particular, Chapel proporciona la sentencia `begin` que crea una tarea asíncrona que ejecuta el bloque de código que sigue a la sentencia. Por ejemplo, `begin File=A;` en el código de la figura 2.16, lanzaría la escritura del array `A` en el fichero `File` de forma asíncrona y continuaría la ejecución del código siguiente concurrentemente. Evidentemente, será responsabilidad del programador no violar las dependencias de datos durante la ejecución de tareas concurrentes, por ejemplo, modificando el array `A` antes de que se haya completado la escritura. Para ello, Chapel proporciona distintos mecanismos, como variables de sincronización y atómicas, así como la sentencia `sync` que actúa como un *join* de todas las tareas lanzadas concurrentemente en un bloque de código.

2.3.1.1. Comparación con MPI IO

En la figura 1.8 (ver página 28) se muestra una versión resumida de las instrucciones necesarias para realizar la escritura de un array en un fichero usando MPI, o más precisamente, la librería MPI IO. Claramente, la complejidad de realizar E/S paralela usando MPI IO es muy alta cuando se compara con nuestra propuesta en Chapel, presentada en la figura 2.15. En Chapel, sólo necesitamos escribir apenas una línea de código, `File=A`, de forma que el compilador y el run-time de Chapel se encarguen de una implementación paralela de la operación de E/S. Sin embargo, en el código MPI IO no sólo requiere de unas 20 llamadas a funciones del API de MPI IO, que también, sino que además estas llamadas a funciones tienen una gran cantidad de argumentos que por un lado hay que conocer y que por otro dan lugar a una mayor probabilidad de cometer errores de programación.

Aunque una de las métricas para estimar el número de errores de programación que contendrá un código es el número de líneas que ocupa el código [68] (métrica LOC en Inglés, *Lines Of Code*), esta métrica no es realmente precisa y no sirve para predecir la densidad de los fallos [48] ya que no tiene en cuenta la complejidad de cada línea de código. Está ampliamente aceptado que los programas complejos contienen más fallos y son más difíciles de mantener que los programas simples [93]. Para estimar dicha complejidad, recurriremos a dos métricas más precisas que LOC. Nos referimos a CC (*cyclomatic complexity*, complejidad ciclomática) y PE (*programming effort*, esfuerzo de programación). Con respecto al CC, los autores en [85] definen esta métrica como el número de predicados o puntos de decisión de un programa más uno. El número ciclomático se corresponde con el número máximo de caminos independientes que existen en un programa estructurado. Valores altos de este parámetro significan un código más complejo. Por último, el parámetro PE, está definido en [60] como una función del número de operadores únicos, total de operandos y total de operadores encontrados en el código. Los operandos se corresponden a las constantes y a los identificadores, mientras que los símbolos o combinaciones de símbolos que afectan a los valores de los operandos constituyen los operadores. Esta métrica puede ser representativa del esfuerzo de programación requerido para implementar el algoritmo. Así que un valor alto de PE significa que es más difícil para un programador codificar el algoritmo.

Tras calcular el CC y PE de la versión MPI IO y de la versión Chapel (ambas escriben un array distribuido por bloques en un único fichero), el resultado es que la versión MPI IO tiene 22 veces más Complejidad Ciclomática y supone 14 veces más esfuerzo de programación.

Una vez validada la programabilidad del interfaz que proponemos, a continuación pasamos a discutir los aspectos de implementación, las decisiones de diseño con respecto a los posibles algoritmos paralelos de E/S, así como los problemas que presenta la E/S

paralela y distribuida.

2.3.2. Implementación de la interfaz

Si no se ha hecho con anterioridad, se recomienda leer primero la sección 1.2.4 donde se explica la arquitectura del sistema de ficheros Lustre, que es el sistema de E/S que pretendemos optimizar en este trabajo.

Para acceder en paralelo a un array distribuido la solución más obvia consiste en que cada *locale* acceda a la región del fichero que le corresponde almacenar localmente. Aunque para accesos de lectura, esta suele ser la solución más eficiente, en el caso de realizar escrituras a ficheros, esta estrategia suele reportar rendimientos sub-óptimos. Esto es así porque en operaciones de escrituras podemos encontrarnos con situaciones de contención y bloqueos que se producen en el sistema de ficheros. Nos referimos al caso de los accesos concurrentes al mismo *stripe*, que ya hemos explicado en la sección 1.2.4. Recordamos aquí que el problema consiste en que no se deben realizar accesos concurrentes a un *stripe* desde dos nodos distintos. Esto es así porque el mecanismo de acceso en exclusión mutua al mismo *stripe* se realiza mediante *locks* y por tanto provoca una serialización en el acceso y un *overhead* de acceso a través de los *locks*. Una posible solución que evitaría que dos *locales* accediera al mismo *stripe* consiste en configurarlos con un tamaño tal que evite que dos *locales* compartan un mismo *stripe*. Sin embargo esta no es una solución factible ni general ya que Lustre limita el tamaño de *stripe* a múltiplos de 64 KiB. Además, recurrir a un tamaño de *stripe* pequeño, lo cual redundaría en una menor probabilidad de que dos *locales* accedan concurrentemente a un *stripe*, perjudica el rendimiento si el tamaño está por debajo de un Megabyte.

Otra posibilidad que hemos explorado para evitar el acceso concurrente a un mismo *stripe*, y la consiguiente serialización en la escritura de datos, consiste en reducir la probabilidad de conflicto tratando de que los accesos a un mismo *stripe* no ocurran al mismo tiempo. Por ejemplo, cuando escribimos una matriz distribuida unidimensionalmente por bloques, cada *locale* podría escribir su bloque de filas de forma independiente. Si consideramos dos *locales* consecutivos, L_i y L_{i+1}, es muy posible que a los elementos del final de la última fila que tiene que escribir L_i les corresponda el mismo *stripe* que a los elementos del principio de la primera fila que tiene que escribir L_{i+1}.

Por ejemplo, en la figura 2.17 tenemos una matriz de 6×6 distribuida unidimensionalmente por bloques entre 3 *locales*. Si el tamaño de *stripe* es de 8 elementos, el *stripe* 2 estará compartido entre los *locales* 0 y 1, provocando que su acceso en escritura deba de coordinarse, de forma implícita o explícita, entre los nodos para que no haya problemas de coherencia. Sin embargo, si el número de filas en cada bloque es suficientemente grande y si el *locale* L_i empieza a escribir por el principio de su bloque al tiempo que

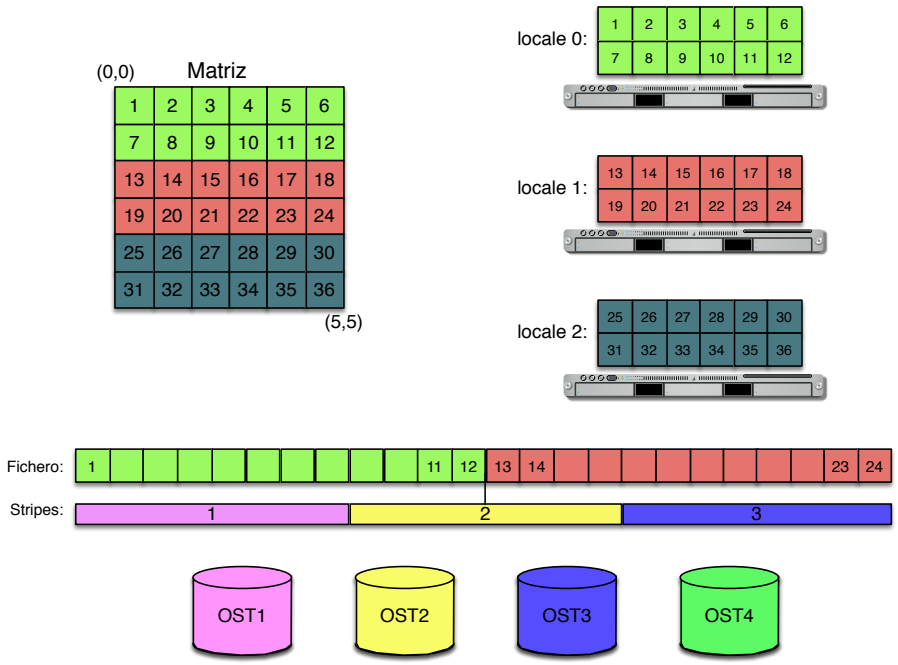


Figura 2.17: Ejemplo de escritura de una matriz distribuida unidimensionalmente en 3 *locales*. El *stripe* 2 se accede desde los *locales* 0 y el 1.

el *locale* L_{i+1} hace lo propio, la probabilidad de que los dos *locales* colisionen en el acceso al mismo *stripe* es muy pequeña. Es decir, L_{i+1} accederá al *stripe* compartido al principio de su operación de escritura, mientras que L_i lo hará al final, de forma que es poco probable que los dos accesos coincidan en el tiempo. Trasladando esa situación al ejemplo 2.17, tendríamos que el *locale* 0 estaría escribiendo en el *stripe* 1 mientras el *locale* 1 escribe sus 4 elementos del final de *stripe* 2. Con suerte, cuando el *locale* 1 quiera escribir sus cuatro elementos del principio del *stripe* 2 el *locale* 2 ya estará escribiendo en otro sitio y los datos del *stripe* 2 estarán consolidados. El problema es que esta aproximación no es general. Por ejemplo, no es válida cuando el número de filas por bloque no es suficientemente grande o cuando las distribuciones son n-dimensionales o de tipo cíclico. En estos dos últimos casos el número de *stripes* compartidos entre *locales* es mucho mayor.

En lugar de que todos los *locales* escriban al mismo tiempo al sistema de ficheros, la alternativa más simple consiste en optar por una escritura en la que secuencialmente

cada *locale* escribe su porción de los datos. Esto no significa que no estemos explotando el paralelismo en la escritura, si no que sólo aprovechamos el paralelismo que provee el sistema de ficheros, que viene dado por el número de OSTs en los que están repartidos los datos que escribe cada *locale*.

Si queremos explotar más paralelismo, otra posibilidad es implementar una escritura colectiva en dos fases (o *collective buffering* en inglés). La idea es recurrir a una solución intermedia entre que todos los *locales* escriban al mismo tiempo o que sólo uno de ellos pueda escribir en un momento dado. Para ello se selecciona un subconjunto de *locales* que serán los responsables de recolectar los datos y transferirlos al sistema de ficheros. A estos *locales* con capacidad de realizar operaciones de E/S se les llama agregadores y todos ellos pueden realizar sus funciones de acceso al sistema de ficheros en paralelo. El número de agregadores es configurable y típicamente inferior al número total de *locales*, lo que permite jugar con el compromiso entre el grado de paralelismo explotado y la contención en el acceso al sistema de ficheros. Es importante organizar la política de agregación de datos que llevan a cabo los agregadores de forma que dos agregadores nunca tengan que acceder a un mismo *stripe* al mismo tiempo. Esta aproximación se estudia en [71], donde se usan 32 *locales*, 16 *OSTs*, 16 agregadores, y un fichero de 96 GiB en un Cray XT5. En estas condiciones, si los 32 *locales* escriben sus propios elementos sólo consiguen un ancho de banda de 420 MiB/seg, que se puede optimizar a 1629MiB/seg cuando sólo escriben los 16 agregadores y se recurre a un algoritmo con agregación de los datos. En otra prueba usando 1000 *locales* la velocidad pasa de 13 MiB/seg (escritura de todos los *locales* en paralelo) a 256 MiB/seg usando sólo los 16 agregadores de datos. Claramente, recurrir a agregadores conduce a una reducción de los bloqueos y de la contención en el acceso al sistema de ficheros distribuido.

La fase en la que los agregadores recolectan los datos de los demás *locales* implica el realizar copias de subconjuntos de los datos entre distintos nodos. En esta fase aprovecharemos las funciones de agregación de datos presentadas en el capítulo 2.2.2, de forma que las transferencias de datos discontinuos entre *locales* se implementen con una única operación. El run-time de Chapel y la librería GASNet se encargarán automáticamente del empaquetamiento de las transferencias para minimizar las comunicaciones que se realizan. Por tanto aprovecharemos mejor las características de la arquitectura de red subyacente.

2.3.3. Posibles fuentes de paralelismo

Vamos a sistematizar la manera de abordar el problema de la escritura de un array distribuido en el sistema de almacenamiento secundario. Pretendemos realizar un estudio metódico de las distintas fuentes de paralelismo que se pueden explotar. Estas fuentes

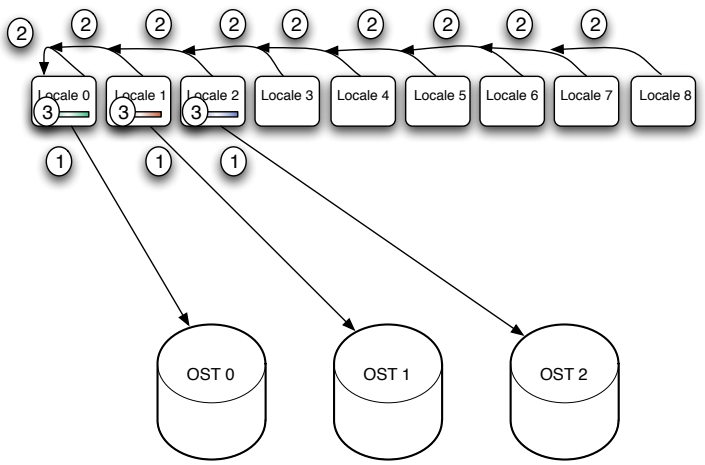


Figura 2.18: Arquitectura global de acceso al sistema de almacenamiento usando 3 agregadores. Se pueden ver tres puntos donde aplicar el paralelismo en el acceso.

de paralelismo son ortogonales y se pueden explotar de forma combinada, así que tendremos que estudiar el rendimiento de cada combinación y las interrelaciones que se producen cuando explotamos varias fuentes de paralelismo al mismo tiempo.

Fuentes de paralelismo en operaciones de escritura		
1	Paralelismo en la E/S	Varios *agregadores* escriben en paralelo, cada uno en un OST diferente. Paraleliza las comunicaciones con los OSTs.
2	Paralelismo en la agregación	Varios *locale* escriben en paralelo en los buffers de los agregadores. Existe un agregador por cada OST. Paraleliza la operación de agregación.
3	Paralelismo entre E/S y agregación	Explota doble buffering en los agregadores. Cada agregador simultanéa las operaciones de agregación y E/S.
4	Paralelismo en la escritura en OST	Varios agregadores escriben en paralelo en un OST. El número de agregadores es un múltiplo del número de OSTs.

Tabla 2.1: Resumen de las distintas fuentes de paralelismo que explotaremos de forma combinada.

En la tabla 2.1 enumeramos las 4 posibles fuentes de paralelismo que se pueden explotar de forma independiente o combinada. Para describir cada una de ellas nos apoyaremos en las figuras 2.18 y 2.19. En estas figuras esquematizamos la arquitectura de E/S para HPC que queremos explotar. Se muestran 9 *locales* que contienen los datos a almacenar en fichero y 3 OSTs encargados de almacenar los *stripes* del fichero. En la figura 2.18 se representan 3 agregadores que coinciden con los *locales* 0, 1 y 2, mientras

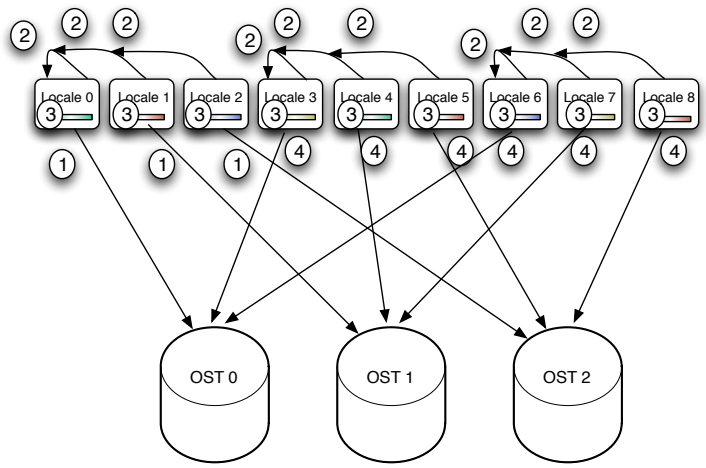


Figura 2.19: Fuente adicional de paralelismo, indicada con un 4, que explota la escritura simultánea de varios agregadores en el mismo OST.

que en la figura 2.19 los 9 *locales* tienen el papel de agregador. Se dibujan con flechas los flujos de datos que intervienen en el sistema, que son dos: i) por un lado hay que mover los datos desde los *locales* hacia los agregadores; y ii) posteriormente el contenido de los buffers de los agregadores se envía hacia los servidores del sistema de ficheros.

En la figura 2.18 nos centramos en las tres primeras fuentes de paralelismo, indicadas por los números 1, 2 y 3. El número 1 representa el paralelismo en las comunicaciones entre los agregadores y los OSTs del sistema de ficheros, es decir, los tres agregadores escriben al mismo tiempo, cada uno en un OST diferente. En esta configuración sólo hay un agregador por cada OST, que es responsable de agregar los *stripes* que se deben almacenar en ese OST.

El número 2 identifica el paralelismo en los movimientos de datos desde los *locales* hacia los agregadores. En la figura se muestra como todos los *locales* agregan en el *locale* 0 todos los *stripes* que deben ser almacenados en el OST 0. Para evitar una excesiva complicación en la figura 2.18 se han obviado los flujos de datos que van desde todos los *locales* hasta los *locales* 1 y 2. Es decir, al tiempo que el *locale* 0 agrega todos los *stripes* que tienen como destino el OST 0, los *locales* 1 y 2 hacen lo propio con los *stripes* que tienen como destino los OSTs 1 y 2, respectivamente.

El número 3 especifica el paralelismo que podemos explotar mediante una técnica de doble buffering que permita simultanear las operaciones de agregación y las de E/S.

Para ello se procede en fases en las que dos buffers se usan de forma alternativa: i) uno de ellos se escribe con los datos que los demás *locales* van agregando en él; ii) del otro buffer se leen los datos que se agregaron en una fase anterior, de forma que se puedan escribir a disco. Cuando ocurra que el buffer de lectura está vacío y el de escritura ya está lleno, se cambian las tornas y se vuelve a proceder como en la fase anterior.

En la figura 2.19 se ilustra una cuarta fuente de paralelismo, representada con el número 4. En esta figura, tenemos 9 agregadores y vemos cómo varios de ellos pueden escribir en paralelo en un mismo OST. Por ejemplo, el OST 0 recibe datos al mismo tiempo de los *locales* 0, 3 y 6. El hecho de tener más agregadores que OSTs permite reducir el tráfico de datos entre los *locales* ya que cada agregador es responsable de menos datos. A cambio, los OSTs están sometidos a una mayor presión en el proceso de escritura. De nuevo en la figura 2.19, bajo el número 2 sólo se muestran los flujos de datos del agregador que agrega los *stripes* que van al OST 0. Los flujos de los agregadores de los datos que pertenecen a los servidores OSTs 1 y 2 serían equivalentes, pero yendo hacia los respectivos agregadores.

Hacemos notar que los gráficos anteriores simplifican la realidad en gran medida. En particular, los grandes sistemas HPC, como Titan, descrito en la sección A.1.2 y EOS, descrito en la sección 1.4.4, presentan una arquitectura de E/S mucho más compleja. Por ejemplo, entre los *locales* y servidores del sistema de ficheros (OSS), existen distintas redes y por tanto es necesario usar routers (llamados LNET, explicados en la sección 1.2.4.2) entre las distintas redes.

Las cuatro fuentes de paralelismo descritas tienen características diferentes e independientes, y hasta cierto punto podemos decir que son ortogonales. Por ejemplo, podemos explotar al mismo tiempo las fuentes de paralelismo 1 y 2 (ir agregando en paralelo y al mismo tiempo escribir los datos disponibles en paralelo en los OSTs) o 1, 2 y 3 (simultaneando la agregación paralela con la escritura paralela). En realidad, la fuente de paralelismo que hemos identificado como 4 incluye de forma implícita al paralelismo en la E/S (fuente de paralelismo número 1). La diferencia es que explotando la fuente 4 el número de agregadores es mayor y nos permite jugar con el compromiso entre la presión en la red de comunicaciones entre *locales* y la presión en la red de comunicaciones hacia la E/S. Cuando el número de agregadores es pequeño, como en la fuente 1, la presión es mayor en la red de comunicaciones. Por el contrario, cuando tenemos más agregadores, como en la fuente 4, ejercemos más presión sobre la red de E/S ya que hay más agregadores escribiendo al mismo tiempo.

Antes de pasar a evaluar las prestaciones que obtenemos de las distintas combinaciones, presentamos a continuación algunos detalles de implementación del código Chapel que permite activar y desactivar independientemente cada una de las fuentes de paralelismo.

2.3.4. Implementación en Chapel

```
1  // Si Paralelismo4=false --> número de agregadores = número de OSTs
2  if ( ! Paralelismo4 ) numAgreg=numOSTs;
3  else                  numAgreg=(numLocales/numOSTs)*numOSTs;

5  // Si Paralelismo1=true --> BSA = forall, y si no BSA = for
6  BSA agregador in 0..#numAgreg do
7     on Locales(agregador) do {

9        if ( ! Paralelismo4 ) {
10          localesToWriteFrom=0;
11          localesToWriteTo=numLocales;
12       } else {
13          localesToWriteFrom=(agregador/numOSTs)*numOSTs;
14          localesToWriteTo=localesToWriteFrom+numOSTs-1;
15       }
16       // Si Paralelismo2=true --> BSL = forall, y si no BSL = for
17       BSL loc in localesToWriteFrom..localesToWriteTo do {

19          if (Paralelismo3)
20             cobegin {
21                GatherToBuffer(bufferA, bufferB, loc,...);
22                WriteBufferToDisk(bufferA, bufferB, file,...);
23             }
24          else {   // Paralelismo3=false
25             GatherToBuffer(bufferA, null, loc,...);
26             WriteBufferToDisk(bufferA, null, file,...);
27          }
28       } //BSL - Bucle sobre los locales
29    } //BSA - Bucle sobre los agregadores
```

Figura 2.20: Pseudocódigo Chapel que implementa el código que ejecutan los agregadores.

En la figura 2.20 mostramos el pseudocódigo del código Chapel que ejecutan los agregadores. Ese algoritmo es configurable de forma que se puedan activar las distintas fuentes de paralelismo descritas anteriormente. En el código esas fuentes de paralelismo están identificadas mediante las variables de configuración[4] `Paralelismo1` a `Paralelismo4`. Las variables `numOSTs` y `numLocales` contienen el número de OSTs y de *locales* con los que se está ejecutando el programa. `BSA` y `BSL` son cons-

[4]En Chapel, las variables de configuración se pueden inicializar en tiempo de llamada al programa pasando los valores deseados como argumentos. Por ejemplo, el comando `./programa --Paralelismo1 =true` lanza la ejecución del `programa` con la variable de configuración `Paralelismo1` igual a `true`.

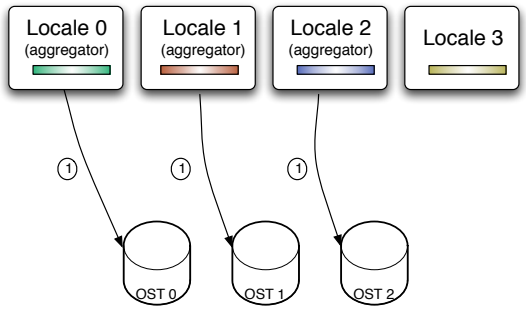


Figura 2.21: Paralelismo 1: Paralelismo en la E/S cuando todos los agregadores escriben a la vez en los OSTs.

tantes que el preprocesador substituye por `forall` o `for` dependiendo de las variables `Paralelismo1` y `Paralelismo2`, respectivamente. De esta forma, el bucle que recorre los agregadores (`BSA`=Bucle sobre los agregadores)) y el que recorre los *locales* (`BSL`=Bucle sobre los *locales*) se pueden ejecutar en serie o en paralelo dependiendo de los argumentos que se pasen a la llamada del programa. Pasamos a detallar la implementación de esos bucles y como se explotan las 4 fuentes de paralelismo en las siguientes subsecciones.

2.3.4.1. Paralelismo 1: en la E/S

En la figura 2.21, identificamos con el número 1 la fase en la que se realiza la transferencia de los agregadores a los OSTs. Todas las transferencias están etiquetadas con un 1 ya que cuando `Paralelismo1=true` las escrituras en los OSTs se realizan en paralelo.

En la figura 2.20, la línea 6 tiene dos posibles implementaciones:

```
for agregador in 0..#numAgreg
```

si `Paralelismo1=false`, o

```
forall agregador in 0..#numAgreg
```

si `Paralelismo1=true`. En la implementación secuencial, usando `for`, la variable `agregador` toma secuencialmente los valores 0, 1, ..., hasta `numAgreg-1`. Esto es así porque en Chapel el operador `#n` en una expresión de rango del tipo `0..#n` representa `n` números consecutivos empezando en 0 (por tanto, hasta el `n-1`). En la alternativa con `forall`, la variable `agregador` toma los mismos valores, pero el bucle se ejecuta en paralelo.

La sentencia de la línea 7 es muy importante ya que especifica en qué *locale* se debe

ejecutar el código que sigue. Con la expresión `on Locales(agregador)do{...}` se obliga a que el bloque de código que sigue al `do` se ejecute en el *locale* con identificador `agregador`. En otras palabras, las líneas 9 a 28 se ejecutan en los *locales* 0 a `numAgreg -1`, de forma secuencial o paralela dependiendo de la variable `Paralelismo1`. En definitiva, `Paralelismo1` activa o desactiva si los agregadores trabajan o no en paralelo.

La agregación es básicamente una redistribución de los datos de forma que cada agregador debe colectar todos los *stripes* que se deben almacenar en el OST correspondiente. La distribución destino es realmente una distribución cíclica por bloques entre los OSTs, y el tamaño de bloque de esa distribución es el tamaño del *stripe*. De esta manera, un agregador se convierte en la única fuente de datos para su OST correspondiente, y aunque la escritura en los OSTs se realice en paralelo, evitamos la pérdida de rendimiento debida a los *locks* (ver sección 1.2.4) que deben ser adquiridos para acceder a *stripes* compartidos por varios *locales*. Esto es porque cada *stripe* se escribe completamente desde un único agregador (no hay *stripes* compartidos entre varios agregadores). El inconveniente es que los agregadores deben invertir un tiempo en colectar del resto de *locales* todos los *stripes* que deben ser escritos en sus respectivos OSTs.

Aunque no está contemplado en el código de la figura 2.20, nuestra implementación también funciona cuando el número de *locales* es inferior al número de OSTs. En ese caso no sólo todos los *locales* son agregadores, sino que estos tienen que servir a varios OSTs. Por ejemplo, si tenemos sólo 2 *locales* y 4 OSTs, el *locale* 0 colectaría los *stripes* que se deben almacenar en los OSTs 0 y 1, mientras que el *locale* 1 agregaría los *stripes* con destino en los OSTs 2 y 3.

2.3.4.2. Paralelismo 2: en la agregación

En la figura 2.22 se muestra de forma esquemática los pasos del proceso de agregación paralela. Por simplicidad, esta figura asume que únicamente el *locale* 0 actúa de agregador y por tanto mantiene los buffers donde agregar los datos de todos los *locales* (incluido él mismo) para luego realizar la escritura en el OST 0. La fase indicada con el número 1 en la figura expresa que todos los *locales* están escribiendo en su correspondiente región del buffer en paralelo. Para el *locale* 0, esa operación implicaría un `memcopy`, pero en la implementación real, los datos del *locale* 0 se leen directamente de su zona de almacenamiento de datos (una matriz en el ejemplo). Los *stripes* de los demás *locales* son agregados mediante llamadas, en última instancia, a `chpl_comm__gets`, presentadas en la sección 2.2.2. La segunda fase está indicada en la figura con un 2 y representa la escritura en disco del buffer. Hacemos notar, que así como el `Paralelismo1` habilita que haya varios agregadores agregando en paralelo, `Paralelismo2` habilita que el trabajo de agregación que hace cada agregador sea también paralelo.

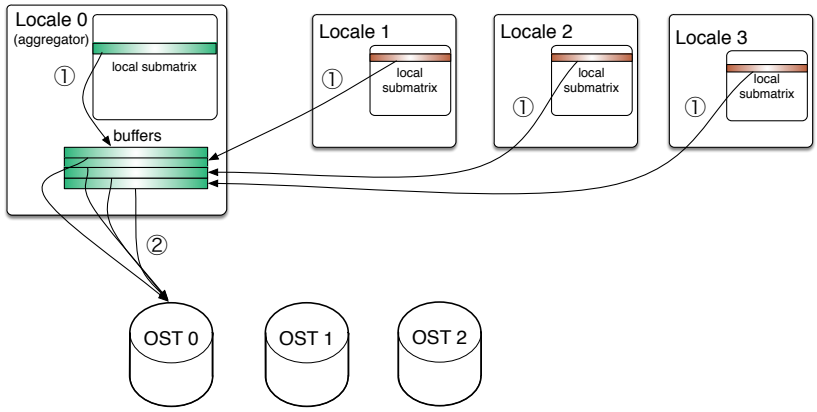


Figura 2.22: Paralelismo 2: Paralelismo en la agregación de los datos haciendo que se reciban datos de todos los *locales* en paralelo.

Volviendo al código de la figura 2.20, el bucle que agrega los datos del resto de los *locales*, `BSL`, aparece en la línea 17. De nuevo, ese bucle tiene dos posibles implementaciones:

```
for loc in localesToWriteFrom..localesToWriteTo do
```

si `Paralelismo2=false`, o

```
forall loc in localesToWriteFrom..localesToWriteTo do
```

si `Paralelismo2=true`. Es decir, si no activamos el paralelismo 2, el agregador irá solicitando los datos a los *locales* uno tras otro, de forma secuencial. En caso contrario, se crearán tantos *threads* como *locales* involucrados en la agregación de forma que aceleraremos el proceso de confección del buffer que hay que escribir al OST. El inconveniente es la gran presión a la que someteremos a la red de comunicaciones ya que, en el caso peor, esta operación de agregación implica una fase de comunicaciones de todos los *locales* con todos los agregadores.

El rango de iteraciones de este bucle `BSL` identifica los *locales*, `loc`, desde los que se deben agregar datos para un agregador determinado. Los *locales* involucrados dependen de si `Paralelismo4` está activado o no, así que posponemos la explicación de este punto a la subsección correspondiente.

2.3.4.3. Paralelismo 3: entre E/S y agregación

En la figura 2.23 mostramos el funcionamiento de la técnica de doble buffer que permite solapar la E/S con las comunicaciones necesarias para la agregación de datos. En la

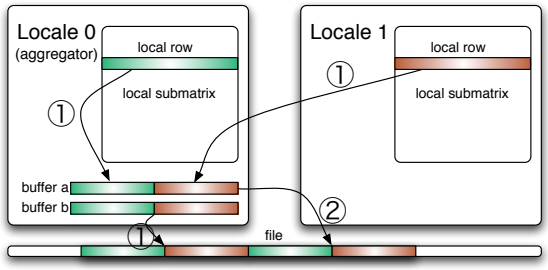


Figura 2.23: Paralelismo 3: Paralelismo entre la E/S y la agregación gracias al uso de doble buffer.

figura suponemos un único agregador, el *locale* 0, que además de su matriz local, reserva espacio en memoria para dos buffers, el `a` y el `b`. La técnica de doble buffer se implementa en una serie de pasos. En cada paso, se usa uno de los buffers para ir agregando datos de otros *locales*, al tiempo que el otro buffer (que ya tiene los datos agregados del paso anterior) se escribe en el OST. En el momento que capturamos en este ejemplo, el buffer `b` tiene elementos de una agregación que tuvo lugar en un paso anterior. Por tanto el número 1 en la figura indica que al mismo tiempo se está escribiendo el buffer `b` en el disco y se están agregando datos en el buffer `a` desde los *locales* 0 y 1. El número 2 en la figura indica que en el paso siguiente se escribirá el buffer `a` en disco y, aunque no mostrado explícitamente, el buffer `b` se estará utilizando para agregar datos. Aunque la figura 2.23 representa un movimiento de datos del *locale* 0 al buffer a, la implementación real evita este movimiento innecesario dentro del *locale* 0.

De vuelta al código de la figura 2.20, el código entre las líneas 19 y 27 contiene las instrucciones relativas al procedimiento que acabamos de explicar. La variable `Paralelismo3` controla si las funciones `GatherToBuffer` y `WriteBufferToDisk` se ejecutan dentro de un `cobegin` o fuera de él. En Chapel, la instrucción `cobegin` crea un *thread* para ejecutar de forma paralela cada línea del cuerpo del `cobegin`. Por tanto, si `Paralelismo3` está activado, `GatherToBuffer` y `WriteBufferToDisk` se ejecutarán al mismo tiempo. En ese caso, entre otros parámetros, estas funciones reciben dos buffers (`bufferA` y `bufferB`) como argumentos, para implementar la técnica de doble buffer. Aunque no se muestra explícitamente en el pseudocódigo de la figura 2.20, se usan variables de sincronización para la comunicación entre los dos *threads* creados en el `cobegin`. Esto permite la coordinación entre los dos threads, evitando por ejemplo que el *thread* que ejecuta `GatherToBuffer` intente reescribir un buffer que aún no ha escrito a disco el thread que ejecuta `WriteBufferToDisk`.

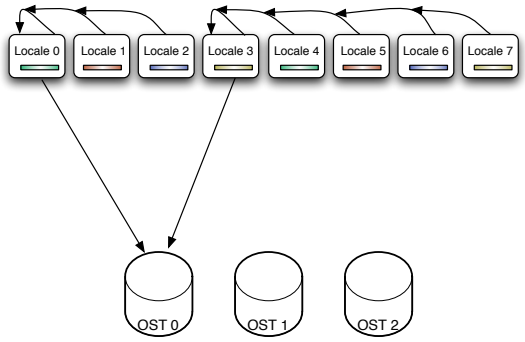


Figura 2.24: Paralelismo 4: Paralelismo en la escritura en cada OST cuando hay más agregadores que OSTs. Los agregadores 0 y 3 se reparten el trabajo de agregar todos los *stripes* que han de almacenarse en el OST 0.

2.3.4.4. Paralelismo 4: en la escritura en un OST

Cuando el número de *locales* es mucho mayor que el número de OSTs, también es mucho mayor que el número de agregadores, porque hasta ahora hemos configurado un agregador para responsabilizarse de todos los *stripes* que van a un OST dado. En ese caso, un pequeño número de *locales* (los agregadores) necesitan acceder al resto para recolectar todos los *stripes* que ese agregador tiene que enviar al OST que representa. Por tanto, cada agregador se convierte en un cuello de botella, concentrando todos los datos que van dirigidos al correspondiente OST. Para evitar esta situación, la activación de la variable `Paralelismo4` permite repartir el trabajo de agregación entre un mayor número de *locales*. Sin pérdida de generalidad, la implementación que hemos validado configura tantos agregadores como el múltiplo del número de OSTs que más se acerque al número total de *locales*.

Por ejemplo, en la figura 2.24 tenemos 8 *locales* y 3 OSTs. Si `Paralelismo4` no está activado, tendremos también 3 agregadores, uno para colectar los *stripes* que van destinados a cada OST (ver figura 2.21). Sin embargo, en nuestro ejemplo, con `Paralelismo4` activado, vamos a configurar 6 agregadores. En la figura 2.24 mostramos explícitamente los agregadores 0 y 3, los cuales se reparten el trabajo de agregar todos los *stripes* que van al OST 0: El agregador 0 agrega de los *locales* 0 a 2 y el agregador 3, de los *locales* 3 a 7. De igual forma los agregadores 1 y 4 se reparten el trabajo de agregar los *stripes* que van al OST 1, y por último, los agregadores 2 y 5, concentran los *stripes* que van al OST 2. Por tanto, al activar `Paralelismo4` reducimos la presión en la red de comunicaciones y también podemos explotar cierta localidad si los agre-

gadores sólo recolectan *stripes* de *locales* vecinos en la red de locales. A cambio, esa presión se traslada a la red del sistema de ficheros ya que hay más agregadores intentado escribir en los OSTs al mismo tiempo.

Recordemos que si `Paralelismo1` no está activado, el recorrido sobre los agregadores será secuencial. Es decir, primero el locale 0 agregará (en paralelo o no dependiendo de `Paralelismo2`) los *stripes* de los *locales* 0 a 2 que han de escribirse en OST 0, a continuación, el *locale* 1 agregará los *stripes* para el OST 1 de los *locales* 0 a 2, y así secuencialmente. Sin embargo, si activamos `Paralelismo1` todos los agregadores escribirán en paralelo en los tres OSTs. Recordemos también que en la sección 1.2.4 se explicó que los *locks* de lustre serializan la escritura de *stripes* de un mismo fichero que van a un mismo OST. Por tanto, la propuesta que implementamos mediante la variable `Paralelismo4` no aumenta el paralelismo de la escritura en cada OST, sino el número de agregadores. Por un lado, este mayor número de agregadores podrían trabajar en paralelo si se activa `Paralelismo1`, y por otro lado tienen que agregar datos en paralelo de menos *locales* si se activa `Paralelismo2`. Es decir, aunque en este trabajo hemos considerado al `Paralelismo4` como una fuente de paralelismo independiente, es en realidad una forma de modular la granularidad con la que se explota el `Paralelismo1` y el `Paralelismo2`.

Para terminar la explicación del código de la figura 2.20, el `if` de la línea 2 permite configurar el número de agregadores igual al número de OSTs (si `Paralelismo4` no está activado) o igual al múltiplo del número de OSTs más cercano al número de *locales* (si `Paralelismo4` está activado)[5]. Además, las líneas 9 a 15 configuran el rango de locales de los que cada agregador debe agregar *stripes*. De esta forma, si `Paralelismo4` no está activado, las variables `localesToWriteFrom` y `localesToWriteTo` delimitarán el rango de todos los *locales* (todos son susceptibles de tener *stripes* que deben ser agregados en todos los agregadores). Sin embargo, si `Paralelismo4` está activado, ese rango será menor y cada agregador recorrerá en el bucle `BSL` un subconjunto del número de *locales*[6].

2.3.4.5. Combinación de estrategias

Como hemos visto en el código de la figura 2.20, las cuatro fuentes de paralelismo se pueden activar o desactivar de forma independiente. Usaremos una tupla binaria con cuatro dígitos para indicar la combinación de fuentes de paralelismo que se está evaluando. El orden de los cuatro dígitos es: (`Paralelismo1`, `Paralelismo2`, `Paralelismo3`, `Paralelismo4`). Dado que la tupla es binaria, cada dígito sólo puede valer 1 para in-

[5] En este código las divisiones son enteras. Por ejemplo 1/2=0.

[6] El pseudocódigo de la figura no contempla el cálculo correcto de ese rango de *locales* cuando el número de *locales* no es múltiplo del número de OSTs.

dicar fuente de paralelismo activada o 0 para indicar fuente de paralelismo desactivada. Por ejemplo, la tupla $(1, 0, 1, 1)$ representa la configuración del algoritmo de escritura en fichero de forma que explotemos el paralelismo en E/S (`Paralelismo1`), el solapamiento entre E/S y agregación (`Paralelismo3`) y la concurrencia en la escritura en cada OST (`Paralelismo4`). El 0 en la segunda posición de la tupla indica que el paralelismo en agregación (`Paralelismo2`) estará desactivado.

Aunque el número total de combinaciones es $2^4 = 16$, en la siguiente sección, donde se evalúan el rendimiento obtenido por nuestro algoritmo en la escritura, no presentamos los resultados de algunas de estas combinaciones por no aportar información relevante.

2.4. Evaluación del nuevo interfaz de E/S

Las siguientes subsecciones describen la evaluación experimental de nuestro algoritmo de E/S paralela escrito en Chapel. En primer lugar, la subsección 2.4.1 precisa las características del *benchmark* que se usa en la evaluación. A continuación, la subsección 2.4.2 resume las características del HW sobre el que se ha ejecutado este benchmark y discute el problema de variabilidad de los tiempos medidos que son inevitables en dicha plataforma. La sección 2.4.3 aborda el estudio del efecto de los distintos parámetros que pueden configurarse en el sistema de E/S. Para saber en qué medida nos acercamos al máximo rendimiento que es posible obtener para este sistema, la subsección 2.4.4 calcula el ancho de banda en escritura que puede conseguirse en una situación ideal. La subsección 2.4.5 evalúa las distintas combinaciones de fuentes de paralelismo y analiza los resultados obtenidos. Finalmente, la subsección 2.4.6 compara nuestra propuesta con un código equivalente que usa MPI-IO de forma que podamos valorar hasta qué punto la implementación de alto nivel en Chapel es competitiva con una implementación optimizada.

2.4.1. Descripción del *benchmark* usado.

Existen multitud de *benchmarks* para medir el rendimiento de la E/S [76] y de la E/S paralela [14]. Algunos de ellos están basados en aplicaciones y otros son sintéticos [108]. Sin embargo estos *benchmarks* están mayoritariamente escritos en C y dado que Chapel es aún un lenguaje emergente hemos tenido que construir nuestro propio algoritmo de *benchmarking*. Nos hemos decantado por implementar un código sintético que se centre en realizar la escritura en un fichero. Es decir, nuestro benchmark no implementa ninguna aplicación o kernel que posteriormente tenga que escribir los datos a disco. Por el contrario, para desacoplar mejor las prestaciones de la E/S, el benchmark sólo escribe, en

un único fichero, un array 1D o 2D, distribuido con la distribución Chapel `BlockDist` y rellenado con una secuencia de números enteros de 64 bits. Los motivos por los que atacamos el problema de escribir en un único fichero, en lugar de que cada *locale* escriba en un fichero diferente, ya se discutieron en la sección 1.2.4. Tras la escritura en fichero, un proceso posterior lee el contenido y comprueba que coincide con la secuencia escrita.

La métrica que usamos para comparar las distintas implementaciones será el ancho de banda en escritura, medido en MBytes/segundo. Se recomendará por tanto que se usen las soluciones que mayor ancho de banda en escritura reporten para el mayor rango de tamaños de fichero, número de *locales*, número de OSTs, etc.

2.4.2. Descripción de la arquitectura HW.

Antes de pasar a detallar las características de la plataforma de evaluación, queremos resaltar que la toma de tiempos de ejecución y el subsiguiente cómputo del ancho de banda en escritura, no está exenta de cierta variabilidad. Estos tiempos se han tomado en un sistema HPC real en producción, esto es, la plataforma no se usa con exclusividad y por tanto otros usuarios se encuentran ejecutando sus aplicaciones HPC de forma concurrente con nuestro *benchmark*. Esto se traduce en que algunos factores que influyen en el rendimiento de la escritura en fichero escapan a nuestro control. A saber:

1. Concurrentemente con nuestra aplicación, hay otros procesos corriendo en la plataforma y eventualmente realizando operaciones de escritura en disco, con una carga que no podemos conocer y variable en el tiempo.

2. Estos otros procesos, también ejercen presión en la red de comunicaciones, de una magnitud que no podemos conocer y variable en el tiempo.

3. En cada ejecución del benchmark, el proceso de carga del ejecutable en la plataforma HPC selecciona los *locales* disponibles, los cuales estarán más o menos cerca de los nodos LNET que hay que atravesar para llegar a los OSTs.

4. Por cada fichero creado por el benchmark, los OSTs asignados para su almacenamiento dependerán del espacio libre y del grado de utilización de los mismos.

De estos dos últimos factores dependen los caminos, en la red de comunicaciones entre *locales* y en la red de E/S, que tendrán que recorrer los mensajes desde los *locales* hasta los servidores del sistema de ficheros.

Tras las primeras mediciones de nuestro benchmark en distintas plataformas HPC con sistema de ficheros basados en Lustre, vimos que en arquitecturas que usan topología

de torus 3D, como Jaguar y Titan, las medidas eran mucho más variables que en otras topologías. El motivo es sobre todo el factor 3 (el mapeo del ejecutable en determinados *locales* de la plataforma). Esta asignación de locales tiene un gran impacto en el ancho de banda en escritura ya que la ejecución en *locales* muy alejados de los nodos LNET provoca una elevada penalización.

Para poder analizar unos resultados experimentales lo más desacopladamente posible de factores fuera de nuestro control se tomaron dos decisiones. La primera consistió en realizar los tests únicamente desde EOS ya que: i) tiene una topología Dragonfly en la que el número de saltos hasta llegar a un LNET es más independiente de los *locales* en los que se mapean los agregadores; y ii) es un sistema más pequeño, con menos usuarios y por tanto con menos carga. La segunda decisión intenta paliar el hecho de que, aún en EOS, los resultados siguen exhibiendo cierta variabilidad. Para contrarrestarla, hemos intentado que las ejecuciones se realizaran en situaciones de carga más o menos equivalentes y las medidas de ancho de banda se han repetido como mínimo 6 veces y como máximo 15 veces, desechando los valores atípicos (*outliers*), por considerarlos consecuencia de una combinación de factores fuera de nuestro control, y de la normalidad. En las siguientes subsecciones reportamos la media de las ejecuciones. También destacamos la dificultad de realizar un gran número de ejecuciones, especialmente cuando se requieren un gran número de *locales*. Estos sistemas HPC funcionan mediante un mecanismo de colas que priorizan los grandes trabajos, y que suelen estar bastante llenas con trabajos de otros usuarios. Algunos de esos trabajos ocupan una gran cantidad de *locales* durante muchos días o semanas. Por tanto, conseguir resultados de una gran cantidad de ejecuciones, y especialmente si estas requieren un gran número de *locales*, puede suponer una espera de varias semanas.

Por tanto, mientras no se diga lo contrario, los siguientes experimentos se ha realizado usando el sistema de almacenamiento Spider II descrito en la sección 1.4.2, la red de interconexión SION II descrita en la sección 1.4.3 y el supercomputador EOS descrito con detalle en la sección 1.4.4. El siguiente párrafo resume las características principales de esos sistemas.

Spider II [98] contiene los sistemas de ficheros Atlas1 y Atlas2, que son idénticos, y está compuesto de 2016 OSTs siendo usados la mitad, 1008 OSTs, en Atlas1 y la otra mitad en Atlas2, con una capacidad total de 28 PBytes, 14 PBytes en cada Atlas. Los 2016 OSTs se sirven desde 288 OSS, cada OSS sirviendo 7 OSTs. Además hay cuatro MDS (*MetaData Servers*) en total. EOS es un cluster Cray XC30 con 744 nodos de 16 cores cada uno. La red de interconexión entre los nodos es Aries, una red propietaria de Cray con topología Dragonfly.

El sistema de ficheros está conectado a través de una red infiniband FDR, llamada SION II, a los nodos LNET de EOS, que se encargan de enrutar los mensajes de E/S

entre los clientes y los servidores Lustre [30]. EOS tiene 9 nodos LNET, que se usan para conectar con los 288 OSSs. Esos enrutadores LNET usan un algoritmo de proyección donde cada LNET tiene uno o mas OSS asignados para los que actúan como *proxies*, tal y como se explica en [43].

Aun cuando los experimentos se han realizado sobre EOS, creemos que las recomendaciones y lecciones aprendidas son también extrapolables a sistemas más grandes, como Jaguar y Titan. Sin embargo, se ha dejado para un trabajo futuro el realizar un análisis estadístico de en qué medida estos grandes supercomputadores se pueden beneficiar de las propuestas que implementamos en este trabajo.

2.4.3. Parámetros que afectan al rendimiento

El rendimiento de nuestro *benchmark* depende de ciertas variables del sistema que han de ser configuradas previamente. Estas variables son:

- El tamaño del array que se va a escribir en el fichero.
- El tamaño de *stripe* (variable *stripe_size* de la configuración Lustre).
- El número de OSTs entre los que se reparte el fichero (variable *stripe_count* de la configuración Lustre).
- El número de *locales* con el que se ejecuta el *benchmark*.
- El tamaño de los buffers usados para agregar datos.

A continuación discutimos los valores que hemos evaluado para esas variables y el impacto que tienen en el rendimiento del código.

En cuanto al tamaño del array, hemos probado con volúmenes de datos que van desde pocos MiB hasta 500 GiB. En estos experimentos se ha comprobado que usar un tamaño de array local (por cada locale) inferior al Gibibyte puede dar lugar a extraer conclusiones engañosas. Esto es así ya que existen buffers y caches en los distintos nodos que hay que recorrer hasta llegar al dispositivo de almacenamiento (nodos LNET y OSSs) que pueden ocultar la escritura real de los datos. Es decir, si los datos a escribir caben en esas caches, desde el punto de vista del *benchmark*, las operaciones de escritura, aun siendo bloqueantes, puedan terminar antes de que los datos estén realmente escritos en el sistema de almacenamiento. Eso provoca que los tiempos de escritura que se obtienen desde el benchmark no sean los reales, ya que no consideran el tiempo hasta la finalización de la escritura en el disco físico. En otras palabras, estas cachés intermedias provocan la ilusión de que tenemos un mayor ancho de banda del que realmente se puede conseguir.

Además, como es habitual en sistemas HPC, si el volumen de datos es pequeño, no se van a compensar los *overheads* de explotar el paralelismo en escritura (arrancar procesos agregadores, gestión de buffers, establecer comunicaciones, etc). Por otro lado, hemos comprobado que aumentar el tamaño de los arrays por encima de 2GiB no conduce a mejores anchos de banda, pero sí a que el benchmark tarde más y por tanto sea poco productivo recolectar resultados de muchas ejecuciones.

Con todo ello, para los experimentos de la sección 2.4.5, finalmente se optó por declarar el array de forma que al distribuirse por bloques, a cada *locale* le corresponda escribir 2GiB. Esto quiere decir que el tamaño final del fichero escrito es igual a 2GiB por el número de locales. Esta estrategia de evaluación en la que el volumen de datos aumenta con el número de *locales* es la habitual en varios trabajos relacionados [76][108][11]. Si el tamaño del array es constante para un amplio rango de *locales*, por ejemplo de 1 a 512, ocurre los siguiente. Si el array es muy grande no cabe en la memoria de un único *locale*. Pero si el array es lo suficientemente pequeño para que entre en la memoria de un único *locale*, cuando se usan 512, a cada uno le corresponde un array local que es demasiado pequeño para evitar los problemas descritos en el párrafo anterior.

En cuanto al tamaño de *stripe*, en el manual de uso de Lustre del ORNL[7] se recomienda usar 1 MiB. En experimentos preliminares hemos evaluado los tamaños 1 MiB, 4 MiB y 10 MiB. Para ello se crearon distintos directorios que se configuraron con los mencionados tamaños de *stripe*. Posteriormente se ejecutaron numerosas instancias de nuestro *benchmark* de forma secuencial, para evitar interferencias entre dos ejecuciones del mismo código al mismo tiempo. Los resultados mostraron que el impacto de esos tamaños de *stripe* en el ancho de banda en escritura no es significativo para nuestro benchmark. Finalmente, se decidió dejar el valor recomendado de 1 MiB para los experimentos de la sección 2.4.5.

Un factor bastante relevante de cara a mejorar el ancho de banda en escritura es el número de OSTs. Además, si `Paralelismo4` no está activado, el número de agregadores coincide con el del número de OSTs, de forma que este parámetro afecta al número de agregadores trabajando en paralelo (si `Paralelismo1`) está activado. En los estudios preliminares del sistema de E/S en EOS encontramos que aumentar el número de OSTs de 4 a 8 y de 8 a 16 mejoraba notáblemente el rendimiento de la escritura. Sin embargo aumentar el número de OSTs más allá de 16 sólo reporta una mejora marginal del ancho de banda en escritura ya que para llegar a los OSTs hay que pasar por los nodos LNET. Dado que en EOS sólo hay 9 nodos LNET, estos terminan convirtiendose en el cuello de botella cuando usamos más de 16 OSTs. Por tanto, evaluaremos nuestro *benchmark* usando 16 OSTs en la sección 2.4.5.

[7] https://www.olcf.ornl.gov/kb_articles/lustre-basics/

En cuanto al número de *locales*, en las siguientes subsecciones se muestra el ancho de banda en escritura cuando este número cambia entre 4 y 64, y entre 16 y 512 *locales*. Algunas combinaciones de fuentes de paralelismo se exploraron sólo entre 4 y 64 *locales*, o incluso menos, ya que se comprobó que no tenía sentido continuar evaluando esas combinaciones por reportar pobres resultados, como se discutirá posteriormente.

Por último, también es necesario decidir cuál debe ser el tamaño de los buffers donde se realizan las agregaciones. Se evaluaron los siguientes tamaños de buffer: 4, 10, 16, 25, 32, 40, 48 y 100MiB. Aunque en principio parecería intuitivo que a mayor tamaño del buffer mayor ancho de banda en escritura, esto no ha resultado ser así. Recordemos primero que los buffers se reusan en los agregadores durante todo el proceso de escritura del archivo: los buffers se llenan con *stripes* desde el resto de los *locales* y luego se vacían al OST, para volver a empezar la fase de agregación. Cuando `Paralelismo3` está activado, estas dos fases (agregación y E/S) se ejecutan en paralelo pero de forma sincronizada. Esto es, no se puede intercambiar el buffer de agregación y el de E/S hasta que las dos operaciones no estén completadas. En la práctica, es imposible balancear ambas fases y la más rápida tendrá que esperar a que la más lenta termine.

Pues bien, en nuestros experimentos hemos comprobado que la fase de agregación es más lenta que la de E/S si los tamaños de buffer son mayores de 32MiB. Es decir, para tamaños de buffer muy grandes, el tiempo de comunicaciones entre todos los *locales* y todos los agregadores para que a cada uno le lleguen los *stripes* que almacenan los OSTs respectivos es mayor que el tiempo de envío del buffer de los agregadores a los OSTs. Es cierto que la red de comunicaciones entre *locales* es más rápida que la de E/S. Pero también lo es que el patrón de comunicaciones de agregación es casi "todos-a-todos" ya que se está haciendo una redistribución de los *stripes*, lo que termina siendo más costoso. Por tanto, para mantener el flujo de escritura en los OSTs de forma ininterrumpida y obtener por tanto el mayor ancho de banda en escritura, el tamaño del buffer debe ser, en esta arquitectura, de tamaño inferior a 32MiB. El resto de los experimentos que mostramos en la sección 2.4.5 se han realizado con buffers de 10MiB.

2.4.4. Estimación del rendimiento ideal

En [98] se anuncia que el rendimiento máximo de Spider II es de 1 TByte/seg. Una versión simplista del rendimiento nos podría llevar a dividir el ancho de banda total entre el número de OSTs para obtener el rendimiento por OST, de forma que 1000TB/s entre 2016 OSTs es aproximadamente 500 MBytes por segundo. Explotando 16 OSTs en paralelo se debería tener un valor máximo de 8000 Mbytes por segundo. Sin embargo esta cota obtenida es poco realista, ya que hay más factores que influyen en el rendimiento y los 16 OSTs pueden estar siendo usados concurrentemente por otros procesos.

Para obtener una estimación más realista de la cota máxima del rendimiento ideal en escritura, hemos medido el ancho de banda de la operación de escritura más básica en las condiciones más favorables que hemos encontrado. Más precisamente, hemos usado el comando `dd`, un comando habitual en sistemas operativos UNIX para realizar operaciones de E/S. El procedimiento consistió en enviar un comando `dd` a cada *locale* a través del sistema de colas de forma que cada uno de esos *locales* escriba en un archivo diferente al mismo tiempo. Los argumentos del comando fueron los siguientes:

```
dd if=/dev/zero of=delete.‘hostname‘.bor bs=1M count=80
```

es decir, la fuente de datos son 0's, el tamaño de bloque, `bs`, es 1MiB y se escriben 80 bloques en un fichero de nombre `delete.EOS-XX.bor` (la `XX` del nombre se substituye por el identificador del *locale* desde el que se está escribiendo). El resultado es que cada *locale* crea un fichero de 80MiB.

La prueba se ha ejecutado usando dos tamaños de *stripe* diferentes (1MiB y 4MiB), 4 OSTs por fichero, y con distinto número de *locales* (1, 16, 32, 48 y 64). Los anchos de banda obtenidos cuando sólo escribe un *locale* estaban en el rango de entre 600-800 MBytes/s. Cuando se ejecutan 16 comandos `dd` en paralelo desde 16 *locales* el ancho de banda sube a unos 3,000 MBytes/s (esto es el resultado de dividir el volumen agregado de todos los ficheros –16×80MiB– dividido entre el tiempo que se consume en crear los 16 ficheros). Si se ejecuta desde 32 nodos, llegamos a unos 4800 MBytes/s. Sin embargo, ejecutar desde más nodos ya no mejora el ancho de banda así que entendemos que en ese punto se alcanza el cuello de botella en la red que comunica los nodos de cálculo con los servidores del sistema de ficheros.

En [141] y en [29] se comprueba que escribir en paralelo en ficheros independientes es más rápido que escribir en paralelo en un único fichero, debido a los *locks* que tienen que gestionar la coherencia en la escritura. Por tanto, entendemos que los 4800MBytes/s representan una cota superior a la que deberíamos intentar acercarnos en nuestra implementación paralela.

2.4.5. Evaluación de los algoritmos de escritura

Hemos realizado varios conjuntos de experimentos para evaluar el rendimiento de los algoritmos de escritura que se proponen en la sección 2.3.2. Dado que existen 16 posibles combinaciones, para ser sistemáticos presentamos los resultados de ancho de banda en dos grupos:

- Combinaciones (0,x,x,x), en la figura 2.25
- Combinaciones (1,x,x,x), en la figura 2.26

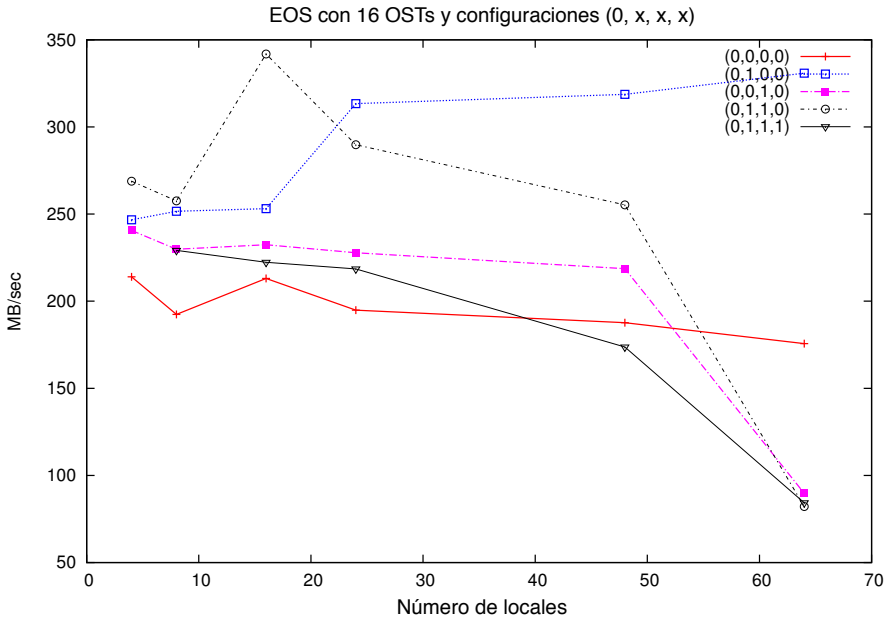


Figura 2.25: Ancho de banda para las mejores combinaciones (0,x,x,x).

En la figura 2.25 presentamos las medidas obtenidas con 16 OSTs, entre 4 y 64 *locales* y usando los algoritmos definidos por las tuplas (0,0,0,0), (0,1,0,0), (0,0,1,0), (0,1,1,0) y (0,1,1,1). En todas ellas `Paralelismo1` está desactivado. Otras tres combinaciones dentro de este grupo son (0,0,0,1), (0,1,0,1) y (0,0,1,1), pero no se muestran en la figura ya que obtienen resultados incluso peores. Como vemos y en general, los anchos de banda obtenidos en estas condiciones están entre los 100MB/s y los 300MB/s. Esto es, más de un orden de magnitud por debajo de la cota máxima estimada en la sección anterior (4800MB/s). Como veremos a continuación, estos resultados son también mucho peores que los que obtenemos con las combinaciones en las que `Paralelismo1` está activado.

De entre las combinaciones que presentamos en la figura 2.25, la (0,0,0,0) no explota ninguna fuente de paralelismo y es una de las que peor rendimiento ofrecen. Al aumentar el número de *locales*, aumenta el tamaño del fichero y casi en la misma medida el tiempo total de escritura. Esto conduce a que el ancho de banda permanezca en torno a los 200MBs. Por el otro extremo, una de las mejores combinaciones es la (0,1,0,0) (sólo activamos el paralelismo en la agregación), especialmente cuando el número de *locales*

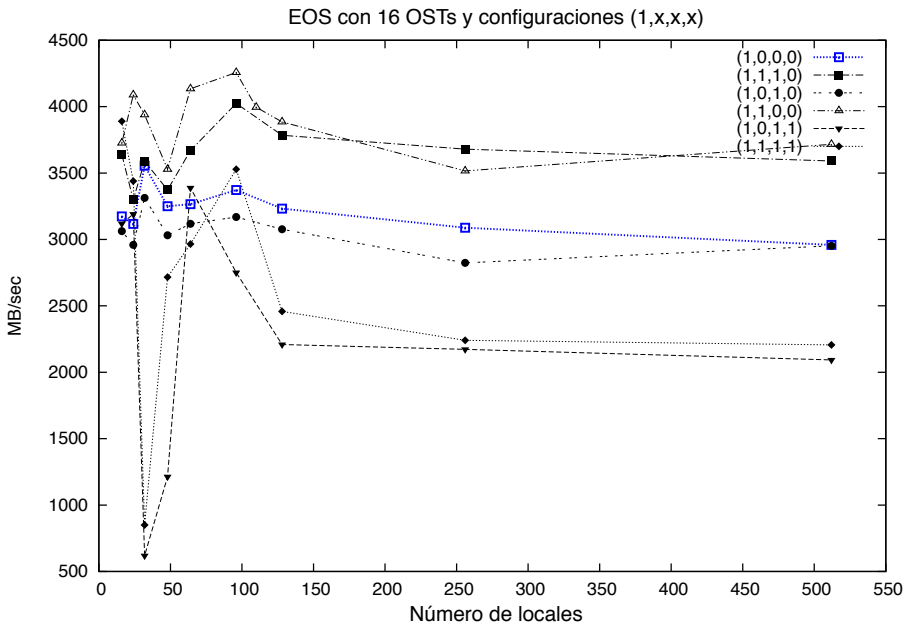


Figura 2.26: Ancho de banda para las mejores combinaciones (1,x,x,x).

es superior a 16, pero aún así el ancho de banda obtenido es bastante pobre. El problema subyacente de estas combinaciones es que ninguna de ellas deja trabajar a los agregadores en paralelo. Por tanto, los OSTs que son alimentados desde cada agregador también están trabajando de forma secuencial. Es decir, es necesario activar el `Paralelismo1` para explotar mejor el hecho de que un fichero se almacena de forma distribuida entre varios OSTs.

En la figura 2.26 se muestran resultados con 16 OSTs, entre 16 y 512 *locales* y con `Paralelismo1` activado. A partir de 128 *locales* se pueden distinguir claramente tres grupos distintos de acuerdo al rendimiento, el más alto, unos 3800MB/s, en el caso de los algoritmos (1,1,x,0), uno intermedio, sobre los 3000MBs, con (1,0,x,0), y el más bajo rendimiento, alrededor de 2200MB/s, con (1,x,x,1). Claramente, en comparación con las configuraciones (0,x,x,x) de la grafica anterior, activar el paralelismo entre agregadores dispara el rendimiento mas o menos un orden de magnitud. Esto confirma la ventaja de escribir en paralelo en los OSTs y nos acerca a los 4800MB/s que hemos medido al escribir en paralelo en ficheros independientes. Es decir, existen combinaciones de paralelismo que evitan la pérdida de rendimiento debida a los *locks* de Lustre y casi igualan

las prestaciones de la escritura en ficheros independientes. El rendimiento de 4800MB/s no se llega a alcanzar, ya que en esta última alternativa ideal no tiene lugar la redistribución entre *locales* y agregadores, penalización que si aparece cuando escribimos en un único fichero.

Sin embargo, algunas de las combinaciones siguen provocando bloqueos en los OSTs. Nos referimos a las combinaciones que tienen `Paralelismo4` activado: las del tipo (1,x,x,1). En contra de nuestras expectativas, esta posibilidad en lugar de estar entre las mejores está entre las que peor redimiento reportan. De hecho, las combinaciones (1,0,0,1) y (1,1,0,1) no se muestran en la figura 2.26 ya que reportaron peores resultados y además estas combinaciones están en cierta forma cubiertas por las combinaciones (1,0,1,1) y (1,1,1,1) que sí están presentes en la figura entre 16 y 512 locales.

Los problemas de rendimiento con las combinaciones (1,x,x,1) radican en que hay varios agregadores escribiendo en el mismo OST (cosa que no ocurre en las combinaciones (1,x,x,0)). Recordemos, pese a que `Paralelismo1` y `Paralelismo4` estén activados, las escrituras de *stripes* del mismo fichero en el mismo OST están serializadas por *locks* de Lustre (ver sección 1.2.4.1). Sin embargo, esperábamos que esta fuente de paralelismo sí ayudase a reducir la carga en la red de comunicaciones entre *locales*, ya que al haber más agregadores por OST, estos se reparten la carga de redistribución de los *stripes* que se han de almacenar en esos OSTs. Por ejemplo, con 512 *locales*, 16 OSTs y `Paralelismo4` desactivado, sólo hay 16 agregadores colectando *stripes* potencialmente desde los otros 512 *locales*. Por el contrario, con `Paralelismo4` activado, los 512 *locales* toman el papel de agregador, 32 de ellos agregan para un mismo OST, lo que significa que cada uno de estos últimos 32 agrega de 16 *locales* diferentes. Lo mismo ocurre para 256, 128, 64, 32 y 16 locales, ya que en todos esos casos cada agregador necesitaría agregar *stripes* de otros 16 locales. Por tanto, esperábamos que tuviese algún impacto en el rendimiento el reducir la carga de los agregadores de tener que comunicarse con 512 *locales* a tener que comunicarse sólo con 16 locales.

Además, las combinaciones (1,x,x,1) estudiadas presentan una caída de rendimiento pronunciada entre 32 y 63 locales. Con `Paralelismo4` activado, todas esas situaciones tienen en común que vamos a tener dos o 3 agregadores por OST. Para confirmar que este era el hecho determinante en la caida de rendimiento, se ejecutaron con 8 OSTs las combinaciones (1,x,x,1), cuyos resultados entre 8 y 56 *locales* se muestran en la figura 2.27. En este caso es entre 16 y 23 *locales* cuando nuestro algoritmo de escritura configura 2 agregadores por OST y cuando se observan las caidas importantes en el rendimiento de escritura. Para la combinación (1,0,0,1) se midió el ancho de banda desde 8 hasta 36 *locales* con paso 1 *locale* (es decir de 1 en 1). En la figura se aprecia que justo en los valores {16, 17, 18,..., 23} *locales* el ancho de banda es el menor y en torno a 250MBs. Estos resultados nos hacen concluir que tener 2 o 3 agregadores contendiendo en el acceso a un OST provoca más pérdida de rendimiento que una configuración en

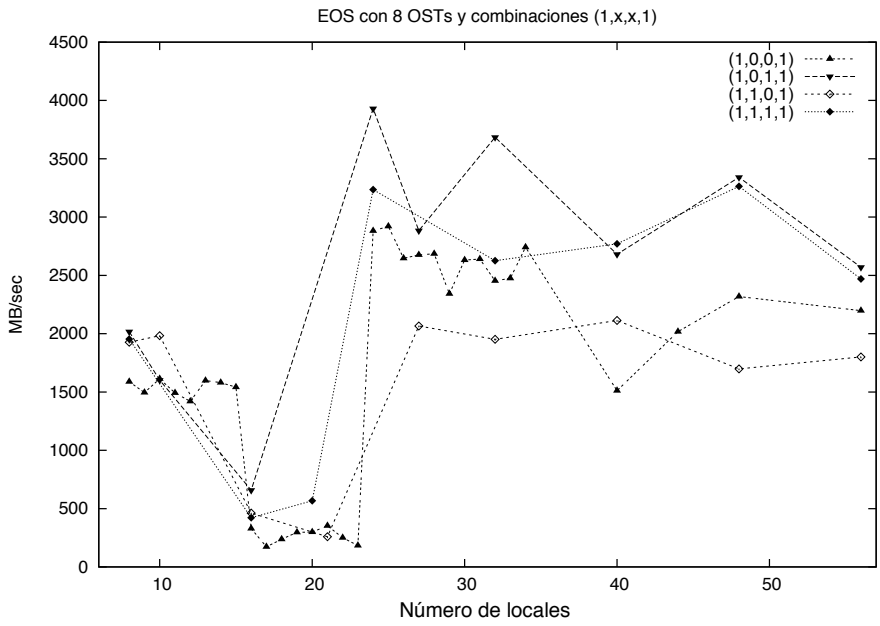


Figura 2.27: Ancho de banda para las combinaciones (1,x,x,1) con 8 OSTs.

la que el número de agregadores contendiendo es mayor. Sin embargo no hemos podido obtener más información que explique los motivos reales de ese hecho.

Si desactivamos `Paralelismo2`, es decir no explotamos el paralelismo en agregación, vemos en la figura 2.26 que obtenemos un rendimiento intermedio en torno a 3000MB/s con las configuraciones (1,0,x,0). El mejor rendimiento, entorno a 3800MB/s, se obtiene cuando además del `Paralelismo1` también activamos el `Paralelismo2`, es decir para las combinaciones (1,1,x,0). Como vemos en la figura, especialmente a partir de 128 *locales*, activar o desactivar el `Paralelismo3` tiene muy poco impacto tanto en las combinaciones (1,0,x,0) como en las (1,1,x,0). Recordemos que este paralelismo se encarga de solapar la fase de agregación con la de E/S. El motivo por el que el `Paralelismo3` no afecta es porque aunque esté desactivado, los buffers y cachés intermedios en los nodos LNET y OSSs están virtualmente permitiendo que este solape entre E/S y agregación también esté teniendo lugar.

Con todo esto, la recomendación es explotar tanto la fuente de paralelismo en E/S (los agregadores escribiendo al mismo tiempo en distintos OSTs) como la fuente de paralelismo en la agregación (cada agregador recolecta *stripes* de los distintos *locales* en

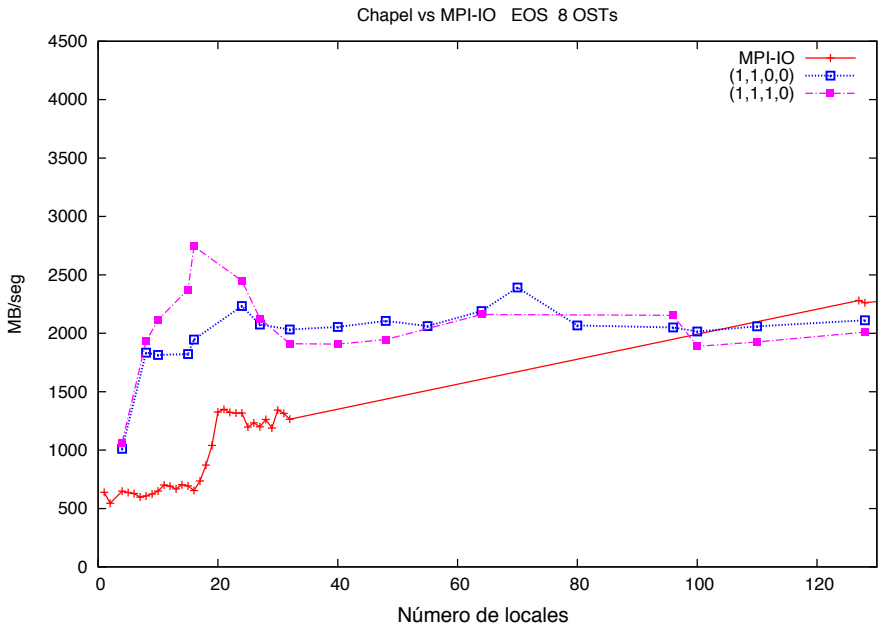


Figura 2.28: Comparación entre los algoritmos (1,1,x,0) y MPI IO con 8 OSTs y entre 1 y 128 locales.

paralelo). En esa situación evitamos los locks de Lustre y nos acercamos a los 4800MB/s que se midieron en la sección 2.4.4 en circunstancias ideales.

2.4.6. Comparación con MPI IO

La librería MPI IO también incorpora un mecanismo de agregación similar al que nosotros hemos implementado en Chapel [101]. En particular, la versión de MPI IO más avanzada que hemos podido ejecutar se basa en un algoritmo de agregación del tipo (1,0,1,0).

En la figura 2.28 se compara el rendimiento de la implementación MPI IO, y las mejores combinaciones que hemos encontrado en la sección anterior, (1,1,x,0). El código MPI IO usado fue la versión optimizada por Cray [142][29] que está basada en ROMIO [119][121] [118], una implementación de alto rendimiento de MPI desarrollada y mantenida en el Argonne National Laboratory (ANL), añadiendo código de la librería Lustre ADIO de Sun Microsystems [71].

El código MPI usa una matriz bidimensional distribuida por bloques de tamaño variable dependiendo del número de *locales*, de forma que a cada *locale* le correspondan 2GB de datos igual que en la implementación Chapel. El sistema donde se realizó la comparación es EOS y tanto en la versión Chapel como la versión MPI, escriben en un único archivo con tamaño de *stripe* de 1 MiB, distribuido en 8 OSTs.

En la figura podemos comprobar cómo la implementación en Chapel reporta mayor ancho de banda que la implementación MPI cuando el número de *locales* es inferior a 128. El motivo es que la implementación MPI no explota el paralismo en la agregación (`Paralelismo2`) que como hemos visto en la sección anterior suele conducir a mejoras significativas en el ancho de banda. Sin embargo, para 128 *locales*, el rendimiento entra la implementación Chapel y la versión MPI es muy similar (entorno a 2400MB/s). Teniendo en cuenta que Chapel es un modelo de programación de más alto nivel y más productivo que MPI, creemos que los resultados obtenidos permiten confiar en que la E/S en Chapel puede ser competitiva en rendimiento.

2.5. Trabajos relacionados

Algunos trabajos previos han estudiado el rendimiento de la E/S en sistemas masivamente paralelos [77][46][142], proporcionando una guía sobre los factores que le afectan. Otros autores [76] han trabajado en la evaluación del rendimiento de sistemas masivamente paralelos, identificando los grandes retos presentes en esos sistemas. Más cercano a nuestra investigación encontramos [142] donde se estudia el rendimiento de una amplia variedad de interfaces de E/S paralelos en plataformas de gran tamaño basadas en Lustre, incluyendo POSIX IO, MPI IO y HDF5. Su estudio incide especialmente en cómo la distribución de los datos sobre los elementos de procesamiento puede afectar a las estrategias de E/S. En cualquier caso apuntan a que las rutinas de E/S deben tener en cuenta tanto el tamaño de *stripe* como el factor de *stripes*, y que el número de procesos realizando accesos a un OST debe estar limitado para evitar la contención.

En general, si los usuarios quieren alcanzar el máximo rendimiento tienen que definir cuidadosamente los parámetros usados en las llamadas de E/S, en particular tienen que ser conscientes de la distribución de datos usada en la aplicación en cuestión, así como de la organización y gestión de los datos en el sistema de ficheros. En este trabajo nos distinguimos en que se provee al usuario con funciones de alto nivel de E/S. mientras que es el runtime, que conoce tanto la distribución de los datos entre los *locales* así como la distribución del sistema de ficheros subyacente, quien se encargue de la organización eficiente de los accesos.

En un trabajo más reciente [29] se analiza el bajo rendimiento que se obtiene con

MPI IO (ROMIO) en entornos donde se usa el sistema de ficheros Lustre, y se propone una nueva librería de nivel de usuario, llamada Y-Lib, para mejorar los resultados. Allí descubren que las implementaciones actuales optimizadas de ROMIO producen patrones de acceso en las comunicaciones de todos a todos (all-to-all), lo que provoca contención en las operaciones de acceso a los OSTs y OSSs. Cuando aparece esta situación en nuestros experimentos, encontramos resultados similares, como se ha explicado en la sección 2.4.5. En [86] llegan a las mismas conclusiones que en [29], pero ninguno de esos trabajos, ni de los otros vistos, aborda el problema que afecta a la velocidad en MPI-IO cuando se producen accesos concurrentes a un mismo fichero en Lustre, y es el relacionado con el funcionamiento de los *locks*. Dicho funcionamiento se explica en [47], y en las secciones 1.2.4 y 2.4.5. En nuestro trabajo, proponemos una estrategia de agregación consciente de este problema que es la finalmente consigue mayor rendimiento.

2.6. Conclusiones

En este capítulo hemos presentado una nueva interfaz de acceso paralelo a la E/S y una implementación de esa interfaz optimizada para ser usada en Lustre, el sistema de ficheros paralelo más usado en grandes sistemas HPC de la actualidad.

El nuevo interfaz de E/S, implementado en el lenguaje de programación de alta productividad Chapel, se basa en una nueva clase o en una nueva distribución que permite mapear un array en el sistema de ficheros Lustre. Las escrituras/lecturas en/de regiones de ese array se convertirán mediante el compilador y el runtime de Chapel en escrituras/lecturas en/de fichero. Esto aumenta la productividad del programador, que se puede despreocupar de la implementación de la distribución de datos y de la comunicación eficiente de los mismos entre los *locales* (esta tarea queda delegada en el compilador y *runtime*). El programador tampoco tiene que preocuparse de la gestión de bloques en el sistema de ficheros paralelo, ni de los accesos a los servidores de datos. Otra ventaja de nuestra implementación es que es independiente del número de nodos que se utilice para crear y escribir el fichero, y posteriormente también es independiente del número de nodos que se use para leerlo.

La conclusión principal es que, cuando se quiere acceder a grandes volúmenes de datos se puede aprovechar eficientemente el sistema de ficheros paralelo si la implementación de la E/S consigue: i) escribir en paralelo en los OSTs desde los *locales* que hacen la función de agregadores; ii) agregar en paralelo los datos a escribir en cada OST; y iii) minimizar la posibilidad de contención, desde distintos *locales*, en los accesos a cada OST. Por otro lado, el solapamiento entre E/S y agregación ya suele estar implementado de forma implícita mediante los buffers y caches de los nodos intermedios entre los clientes y los discos de almacenamiento. Por tanto, en nuestros resultados no aporta un

beneficio sustancial una implementación explícita de esta fuente de paralelismo. Otros párametros del algoritmo de E/S que hemos evaluado y que tienen gran impacto en el ancho de banda de escritura deben ser estudiados y configurados adecuadamente: i) no tiene sentido aumentar el número de OSTs cuando ha dejado de ser el cuello de botella (en EOS, los LNET son el cuello de botella para más de 16 OSTs); y ii) elegir con cuidado el tamaño del buffer que realiza la agregación y que será el que contiene los bloques que se envían en cada operación de E/S al OST.

3 Almacenamiento optimizado de datos de secuenciación genética

En este capítulo presentamos un nuevo formato de almacenamiento de datos genéticos obtenidos mediante técnicas de secuenciación de próxima generación (NGS, *Next Generation Sequencing*) y así como la implementación de una librería paralela para la gestión de dicho formato.

Primero identificaremos los principales desafíos que desde el punto de vista del almacenamiento presentan este tipo de datos genéticos. A continuación describimos el formato que proponemos, FQBin, comparándolo con las soluciones actuales. Finalmente analizaremos las medidas de rendimiento de nuestra implementación considerando diversas configuraciones en el sistema, mostrando la escalabilidad de la solución y el impacto de las redes, así como las limitaciones de la solución.

3.1. El almacenamiento de secuencias genéticas

Antes de comenzar a analizar los distintos formatos existentes de almacenamiento de datos obtenidos usando secuenciación genética de alto rendimiento, tenemos que conocer en qué consisten esos datos y cuál es su origen.

En la bioinformática primero nos encontramos con la muestra tomada, bien de una persona, animal o planta. Esa muestra se somete a distintos procesos con los que se busca preparar el ADN de ese organismo concreto, para después ser insertada en un

secuenciador con el que, mediante distintas técnicas, se obtienen las bases que componen el ADN. Existen 4 bases principales que componen el ADN, la adenina (A), la timina (T), la citosina (C), y la guanina (G), de ahí que el resultado de una secuenciación de ADN sea una o más secuencias formada por esas cuatro letras (ATCG). Las últimas generaciones de secuenciadores usan distintas tecnologías para generar las bases, de forma que la salida directa del secuenciador puede ser una imagen, u otro tipo de datos, que necesitan ser procesados para obtener las bases. Todo este procesamiento, tanto a nivel químico como informático, no es totalmente fiable, por lo que se suele añadir un valor que indica la calidad de la lectura de cada una de las bases, teniendo cada letra asociado un valor de calidad que define la fiabilidad que se le otorga a que esa base sea correcta. En la sección 3.1.3 se explican con más detalle los formatos y la casuística de los valores de calidad.

Además, todavía no existe una tecnología que permita la secuenciación de una cadena de ADN de manera ininterrumpida, por lo que lo que se hace es replicar el ADN y a continuación partir el ADN replicado en trozos de forma aleatoria mediante un proceso químico, de forma que cada una de las bases se tendrá más de una vez, en varias secuencias distintas. Al número de veces que aparece cada base en las secuencias generadas se conoce como cobertura: cuantas más veces esté repetida cada base mejor es la calidad de los resultados finales. Hay que tener en cuenta que no sólo las posiciones por las que se parte la cadena de ADN son aleatorias, si no que además el orden en el que se obtienen los resultados también es aleatorio, por lo que, en resumen, tras secuenciar se obtienen millones de secuencias de pequeños trozos, lo más habitual es de unas 100 a 150 bases cada secuencia, cortadas aleatoriamente, con repeticiones, errores y desordenadas.

Así, el resultado que se obtiene de un secuenciador son varios millones de pequeñas cadenas de bases, y esto es, junto a la calidad, lo que habría que almacenar.

3.1.1. Formatos de ficheros para el almacenamiento de secuencias genéticas

Como hemos explicado, nos encontramos con la necesidad de almacenar y acceder a millones de secuencias, que pueden ser de longitud variable, de entre 20 y 3000 bytes. A priori sólo deberían de contener 4 letras, ATCG, pero en la práctica, dependiendo de la tecnología usada para secuenciar, puede contener algunas letras más.

El formato estándar *de facto* para el almacenamiento de este tipo de datos, se denomina FASTA. Es un formato extremadamente simple y sencillo de usar. Se definió por primera vez en [82], y se puede resumir en que contiene nombres de secuencias y secuencias. Una línea que comienza con un $>$ contiene el nombre de la secuencia y a continuación contiene la secuencia en una o más líneas. Esta definición es muy laxa, ya

que deja multitud de parámetros sin especificar, como el ancho de las líneas, el número máximo de líneas que puede ocupar una secuencia, el tamaño máximo del nombre, etc. En su origen este formato era usado por la suite de programas bioinformáticos `fasta`, del que toma el nombre, y se popularizó por su simplicidad y facilidad de uso.

Pero en el formato FASTA no existe la posibilidad de integrar otros campos, como las calidades, en el mismo fichero. Para especificar las calidades se suele usar otro fichero donde, siguiendo el mismo formato que el fichero FASTA, se van poniendo las calidades de las distintas secuencias, indicando el nombre de la secuencia y a continuación la calidad de las bases. La calidad indica, como hemos comentado, la fiabilidad de la lectura de cada una de las secuencias: una calidad baja indicará que la lectura no es fiable, y muy posiblemente será errónea.

Como el formato FASTA no era realmente un estándar, muchos actores del mundo de la bioinformática (investigadores, compañías, ...) fueron creando sus propios formatos, generalmente incluyendo en el mismo fichero los datos del nombre de la secuencia, la secuencia genética, y la calidad. Todos estos formatos se pasaron a denominar FASTQ (*FASTA* con *Quality*). Esto ocasionó el problema de tener distintos formatos con el mismo nombre, por lo que se intentó clarificar la situación en [21], donde se define el formato FASTQ como un estándar.

Existen multitud de conversores para convertir entre los formatos anteriores, por lo que no se suele considerar un problema, aunque desde el punto de vista del rendimiento si lo sea, ya que una conversión de formato implica una lectura del fichero original y la escritura en otro fichero con el formato deseado. Hoy en día la mayoría de los secuenciadores generan directamente un fichero FASTQ.

3.1.2. Volumen de datos

En 3.1 hemos visto lo que queremos almacenar, y en 3.1.1 hemos visto los formatos que se usan, pero ¿de qué volumen de datos estamos hablando?

El genoma humano ocupa en total unos 3200 millones de pares de bases. Por tanto, en principio, para almacenar el genoma de una persona tendríamos que almacenar unos 3200 millones de caracteres. Si lo codificamos con las letras ATCG, suponiendo que cada letra se almacena en un byte serían 3200 millones de bytes, o unos 3.2 Gigabytes.

Los avances han permitido abaratar los costes de la generación de las bases, pero como la salida consiste en pequeñas secuencias de generalmente unas 100 bases, y éstas están repetidas y desordenadas, se hace necesario que entre las secuencias generadas exista bastante solapamiento, para poder ensamblar las distintas secuencias correctamente. Como hemos mencionado anteriormente, al número de repeticiones por base se

le llama cobertura. Por ejemplo, si en una secuenciación tenemos una cobertura de 30 eso indica que, más o menos, cada una de las bases aparece en 30 secuencias distintas, facilitando su ensamblaje o mapeo correcto. Así, para obtener el genoma de una persona nos podemos encontrar con millones de secuencias cortas que ocupan en total decenas de Gigabytes. Como no todo el genoma se considera que contiene información útil, es habitual secuenciar sólo el exoma, con lo que se consigue abaratar los costes y reducir los datos generados. De hecho, con varios Gigabytes de datos es suficiente para contener el exoma de una persona.

Además de la tecnología, hay dos factores que influyen en el número de bases que podemos obtener a partir de una muestra, como son el dinero que estemos dispuestos a gastar y el tiempo de que se dispone para obtenerlo. Cuando se secuenció por primera vez el genoma humano se usaron las mejores tecnologías y herramientas disponibles y el coste fue de 2700 millones de dolares (en 1991) y se tardó 13 años, de 1990 a 2003. Se terminó dos años antes de lo esperado, y con un coste 300 millones menor, gracias a los avances que se produjeron durante el desarrollo del proyecto.

En 2003 se hizo oficial la obtención del genoma humano, aunque en la práctica es un trabajo en continua evolución, ya que hay áreas del genoma que no es posible secuenciar con las técnicas actuales, debido sobre todo a la repetición de las secuencias. Cada cierto tiempo se publica una nueva versión, actualmente (en 2015) van por la versión 20, GRCh38 [57], y ya han sacado varios parches para esa versión, con pequeños cambios. En 2013 ese mismo desarrollo, la secuenciación del genoma humano, costaba entre 5000 y 6000 dólares y se requerían uno o dos días. Conforme progresan las técnicas, el proceso se va volviendo más y más barato y rápido.

Eso hace posible que una persona concreta secuencie su ADN, por un coste y en un tiempo impensable hace un tiempo, lo que proporciona ventajas para investigar la influencia de los distintos genes en diversas enfermedades, permitiendo en un futuro lo que se ha dado en llamar medicina personalizada.

Actualmente puede ser más útil el secuenciar sólo algunos genes en vez del genoma completo, dependiendo del objetivo que se busque. Así empresas como 23andme (http://www.23andme.com) ofrecen la secuenciación del ADN de una persona a un bajo precio, pero realmente no secuencian el genoma completo, si no que buscan variaciones en determinados lugares del ADN, concretamente analizan más de 700.000 marcadores. De esa forma es mucho más barato, rápido y el volumen de datos es mucho menor que el de una secuenciación completa, habiendo elegido una solución de compromiso entre todos los factores implicados. En caso de querer realizar una secuenciación completa hay que seleccionar además la cobertura que necesitamos, por lo que el volumen de datos varía bastante, entre 9 GBytes y 120 GBytes. Esa cantidad depende de lo completa que se quiera hacer la secuenciación. Además, a veces es necesario hacer la secuenciación

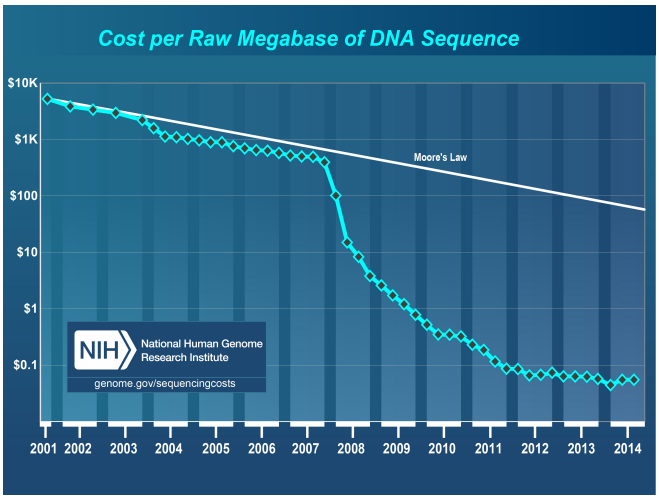


Figura 3.1: Coste de secuenciar un millón de bases (una megabase) [139].

bien de distintos órganos, o bien del mismo órgano en distintos momentos, si es que se están buscando mutaciones sobre el ADN original de la persona. Así tenemos que el problema con el que nos podemos encontrar es el coste de realizar una secuenciación de una persona, con una cobertura adecuada.

En los últimos años las tecnologías disponibles para la secuenciación han mejorado a un ritmo incluso superior a la famosa ley de Moore. El coste de generación de bases ha ido cayendo a un ritmo muy acelerado: en [139] se puede encontrar la tabla que hemos proyectado en la figura 3.1 y que representa el coste de secuenciar una megabase (un millón de bases) usando una escala logarítmica y comparándola con lo que aparecería si se usara la ley de Moore.

Con todo ello, si queremos secuenciar completamente a 1000 personas tendremos que almacenar más de 100000 Gigabytes, o 100 Terabytes de datos, por lo que el coste del almacenamiento es muy alto. Debemos tener en cuenta que el aumento del volumen de datos generados por los secuenciadores es mayor que el aumento de la capacidad de los sistemas de almacenamiento, o dicho de otra forma, el coste de la secuenciación está disminuyendo a un ritmo mayor que el ritmo al que disminuye el coste del almacenamiento [115] [72].

Conforme los costes de secuenciación van disminuyendo, los costes del procesamiento y almacenamiento de los datos generados en la secuenciación van aumentando, al disponerse de más datos que procesar y por tanto más posibilidades de generar infor-

mación a partir de esos datos [106], por lo que los costes totales no bajan tanto como es de esperar cuando se tienen en cuenta todos los factores implicados.

Por otro lado, la cantidad de datos a acceder es tan grande que se comienza a encontrar un cuello de botella en la E/S en los sistemas HPC, provocando una pérdida de rendimiento, al estar los elementos de proceso esperando a los datos, debido al alto volumen de datos que usan los algoritmos genéticos. En [80] se describe un planificador de tareas que tiene en cuenta la E/S a la hora de planificar dichas tareas, para evitar que al ejecutar muchas tareas en paralelo se produzcan esperas debido a la E/S. En ese trabajo se utilizan los sistemas de fichero NFS y Lustre, y la clave de la estrategia que se plantea tiene en cuenta qué OSTs (servidor de datos) usa cada tarea para que no coincidan, evitar la contención y así poder tener un mejor rendimiento global.

Hay que tener también en cuenta que el problema de tener una cantidad muy grande de datos se produce si se almacenan los datos en bruto. Si se almacenan tras ser procesados ocuparán mucho menos, ya que las repeticiones de datos desaparecerán, y se almacenará sólo lo que se haya determinado que es el genoma real del organismo al que se le realizó la secuenciación. El problema de esto es que distintos algoritmos de secuenciación dan distintos resultados, por lo que la decisión de borrar los datos brutos originales no es sencilla, ya que una mejora futura de los algoritmos permitiría obtener resultados más precisos. Por esta razón, hoy en día se suelen almacenar los datos en bruto para que que en un futuro se pueda obtener más información a partir de ellos.

3.1.3. Compresión de los datos

Se han usado también distintos métodos para comprimir los datos obtenidos a partir de los secuenciadores genéticos. Se puede encontrar un resumen en [131]. Como se ha mencionado, estos datos suelen estar organizados en nombres, secuencias y calidades. A veces se usa un campo extra para apuntar alguna información adicional de las secuencias.

Comprimiendo las calidades

Comenzando por las calidades, hay que tener en cuenta que tienen ciertas propiedades que dependen del mecanismo usado para realizar la secuenciación, lo que permite aprovecharnos de eso para comprimir, e incluso para tener compresión con pérdidas sin que afecte al procesamiento de los datos.

Hay varias formas estándar de almacenar las calidades. Usualmente se almacenan como números enteros separados por espacios, o como caracteres ASCII que representan los números. El rango de valores que pueden tomar las calidades viene dado por la tecnología que se haya usado para realizar la secuenciación, y suele ser bastante pequeño, por ejemplo con la tecnología de Illumina varía entre 0 y 40 [21] y en el resto

de tecnologías se suele usar la codificación usada por el programa PHRED [42] y variar entre 0 y 99 [70]. Un método bastante habitual para mejorar la compresión de la calidad es cuantificar el valor de las calidades en varios niveles. Por ejemplo, si las calidades toman valores entre 1 y 100, podríamos usar sólo tres niveles de cuantificación: mala (1), regular (50) y buena (100), de forma que los valores a comprimir sean más simples, y se pueda alcanzar una compresión mayor [96]. Esta compresión es con pérdidas, ya que tras comprimir no se podrán recuperar los valores originales, pero el resultado suele ser suficiente para los algoritmos que se van a aplicar, y tiene el beneficio de facilitar la compresión, ahorrando mucho espacio [96].

Comprimiendo las secuencias genéticas

Los métodos para comprimir los datos de los genomas se pueden clasificar en tres tipos dependiendo de en qué se base la compresión [131]. El más básico se basa en la manipulación de bits, a continuación tenemos los basados en diccionarios y los más sofisticados son los basados en referencias. Vamos a ver las ventajas e inconvenientes de cada uno de ellos.

Los algoritmos basados en manipulación de bits se aprovechan de que las bases están formadas por cuatro letras (ATCG), por tanto no es necesario usar ocho bits para almacenarlos, que es lo que ocupan cuando se usa la codificación ASCII para las letras. En este caso, con dos bits es suficiente: $A \rightarrow 00$, $T \rightarrow 01$, $C \rightarrow 10$, $G \rightarrow 11$. Simplemente con ese sistema las secuencias ocupan cuatro veces menos. En el caso de la calidad podemos adoptar una solución equivalente, si la calidad va de 1 a 100, el valor se puede almacenar en un sólo byte con una codificación binaria en vez de usar tres caracteres. Podemos encontrar ejemplos de este tipo de compresión en [123]. Tenemos un problema si en el origen aparecen más de las cuatro letras básicas, o mayúsculas y minúsculas. Además el fichero resultante es binario, y por tanto no se puede ver como texto.

Los métodos basados en diccionarios se basan en reemplazar secuencias por referencias a un diccionario, que generalmente se suele ir construyendo sobre la marcha. Los ejemplos más típicos son los basados en algoritmos como el LZW (Lempel-Ziv-Welch) [138]. También se puede usar una codificación Huffman, o cadenas de Markov. En estos casos se busca crear una codificación mínima. Eso implica almacenar también el diccionario además de los datos [75].

Finalmente los algoritmos basados en referencias son los que tienen un almacenamiento más eficiente, siempre que ya exista un genoma de referencia para el organismo del que se han obtenido los datos, ya que el organismo secuenciado se debería de parecer mucho al de referencia, y por tanto casi todas las secuencias estarán ya en ese genoma de referencia. En este caso sólo hay que almacenar punteros a los sitios donde aparece cada secuencia. Habrá características que diferencian unos organismos de otros, y esas partes serán mucho más difíciles de comprimir. Otra desventaja es que la velocidad de

compresión no se puede predecir.

3.2. FQbin: una librería para el almacenamiento y acceso a datos de secuenciación genética

Para reducir el impacto sobre el sistema de almacenamiento, que supone la enorme cantidad de información que se genera con las tecnologías de próxima generación de secuenciación (NGS, *Next Generation Sequencing*), proponemos FQbin [59]. Se trata de un formato nuevo y versátil para la compresión, almacenamiento y recuperación de datos obtenidos por secuenciación genética de alto rendimiento de ADN. Es un formato compatible con FASTA y FASTQ y como veremos, mejora el rendimiento de las propuestas existentes. Se basa en la librería zlib y ofrece una compresión de hasta 10x. El fichero comprimido puede leerse y descomprimirse en un tiempo menor al que se necesita con el formato FASTQ, que se puede considerar el formato de facto en este campo. Además, nuestra propuesta de formato proporciona acceso aleatorio a todas las secuencias que almacena [65]. Por otro lado, hemos comprobado que en sistemas distribuidos la ventaja del nuevo formato propuesto es mayor cuanto más lenta sea la red.

Se han realizado dos implementaciones de FQbin, una en C y otra en Chapel, y en este capítulo comparamos el rendimiento y la productividad de cada una de las soluciones.

3.3. El formato de contenedor FQbin

El formato FQbin unifica los campos FASTA, Quality y extras de cada secuencia comprimiéndolos y añadiendo una cabecera para facilitar su acceso aleatorio. En disco se almacenará la cabecera seguida por los datos de esa secuencia, todo comprimido. Se puede ver un esquema en la figura 3.2.

El formato FQbin comienza con una cabecera de longitud variable, los cuatro primeros bytes (`FILE HEADER LENGTH`) indican la longitud de la cabecera del fichero, a continuación viene una cadena alfanumérica (`STRING`) con el identificador del formato, y dos campos, `VERSION` y `SUBVERSION`, con la versión y subversión, controlando la existencia de futuras versiones sin provocar problemas con los ficheros y librerías ya existentes. A continuación viene un conjunto de bloques de datos comprimidos (*compressed blocks*), conteniendo cada bloque un número configurable de secuencias, por defecto 10000, comprimidas como un sólo flujo zlib. Cada uno de los campos de secuencias (`SEQUENCE RECORD`) dentro del bloque comprimido comprende una cabecera de

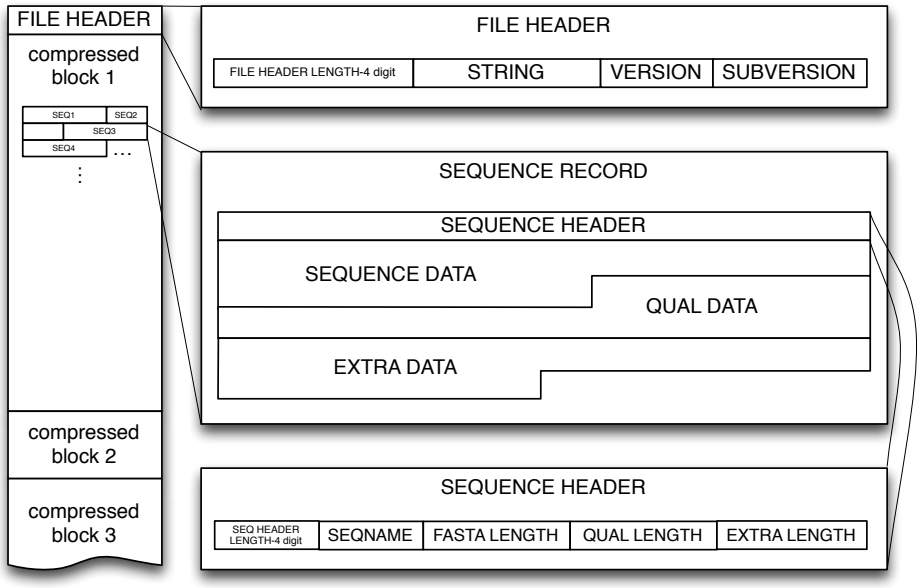


Figura 3.2: Formato FQbin.

secuencia (`SEQUENCE HEADER`) y a continuación el resto de los datos (`SEQUENCE DATA, QUAL DATA, EXTRA DATA`). La cabecera de cada una de las secuencias, `SEQUENCE HEADER`, que desglosamos en la parte inferior de la figura 3.2, comienza con cuatro bytes que contienen la longitud de la cabecera (`SEQUENCE HEADER LENGTH`), que es variable. A continuación vienen cuatro campos de texto que contienen: i) el nombre de la secuencia (`SEQNAME`), que es el campo ID que sirve de identificación para realizar el acceso aleatorio; ii) la longitud de la secuencia (`FASTA LENGTH`); iii) la longitud de la calidad (`QUAL LENGTH`); y iv) la longitud de los extras (`EXTRA LENGTH`), si es que hay. Una vez que el bloque se llena, es decir, se llega al número de secuencias predeterminado, se cierra el flujo y se crea uno nuevo. De esta forma cuando hay que extraer un dato sólo hay que descomprimir el bloque en el que se encuentra y no todo el fichero, ahorrando tiempo y accesos al disco. Como se explicará posteriormente en la sección 3.8.4 esta separación en bloques sirve además como protección contra la corrupción de los datos.

Cada vez que se crea un bloque nuevo se comienza con un diccionario vacío en zlib. Como los datos a comprimir son secuencias genéticas probamos a realizar una precarga del diccionario con datos de secuencias genéricas, con la intención de no partir de un diccionario vacío cada vez que se comienza a comprimir, pero observamos que

no se producía ninguna mejora en la compresión. El motivo es que los datos son semi aleatorios, y por tanto la precarga del diccionario no le supone una ventaja respecto a partir del diccionario vacío, ya que se consigue adaptar en muy poco tiempo a las cadenas que se va encontrando, y en caso de tener el diccionario lleno, tiene que ir olvidando los códigos que contiene para aprender los nuevos, lo que hace que no se obtenga ninguna mejora precargando el diccionario.

La compatibilidad con software actual y versiones anteriores está garantizada gracias a que los contenidos de un fichero FQbin pueden ser enviados vía *streaming* a programas que aceptan datos FASTA/QUAL o FASTQ, evitando la necesidad de realizar una conversión, lo que suele implicar realizar una copia de los datos en otro fichero.

3.4. Acceso aleatorio

El contenedor FQbin y sus herramientas asociadas permiten un acceso aleatorio rápido a secuencias usando su ID, que es su nombre. Esto se consigue gracias a dos ficheros externos, uno que almacena un índice de los datos y otro fichero con un *hash* al índice. Estos ficheros se pueden regenerar en cualquier momento.

El fichero de índices almacena la posición de cada una de las secuencias en el fichero principal. Para que el acceso sea más rápido se incluye el nombre de la secuencia, la posición del bloque donde está almacenado y la posición dentro del bloque donde comienza la cabecera de la secuencia, de esta forma se puede realizar un `seek` al bloque donde está la secuencia, y a continuación habría que moverse descomprimiendo, realizando un `gzseek`, hasta la posición donde está la secuencia dentro del bloque. Esta estrategia permite un acceso aleatorio rápido a una secuencia, aunque no es realmente instantáneo, ya que sería necesario descomprimir el fichero al menos hasta el punto donde se encuentra la secuencia, y, si hay muchos millones de bloques almacenados en el mismo fichero, llegar hasta el último descomprimiendo puede tardar unos segundos.

La siguiente mejora que se planteó fue crear una tabla *hash* que acelere el acceso a la tabla de índices. Para ello se ordena el fichero de índices, se comprime por bloques y en la tabla *hash* se almacena el nombre de la primera y última secuencia que almacena cada bloque y su posición dentro del fichero de índices. Este fichero de *hash* ocupa un espacio mínimo respecto al total, sobre un 0,0001 %, por ejemplo, unos 12 Kilobytes para unos 9.4 Gigabytes de datos. Cuando es necesario acceder a una secuencia concreta, primero se carga el fichero de *hash*, se mira en qué bloque del fichero de índices se puede encontrar la secuencia buscada, se accede a ese bloque y se busca esa secuencia. Si no se encuentra es que no está en el fichero, se da un error de “secuencia no encontrada”. Si se encuentra se lee la posición del bloque dentro del fichero FQbin y la posición de

esa secuencia dentro del bloque, finalmente se accede con un `seek` a ese bloque, y dentro de este con un `gzseek` a la secuencia en cuestión. De esta forma para acceder a una secuencia sólo se accederán una porción mínima de los datos. Además la secuencia se obtendrá en el *stout* en formato FASTA o FASTQ, evitando el tener que pasar por conversores para ser usadas por otros programas.

3.5. Simplificación de los valores de la calidad

Como los valores de la calidad suelen estar bastante repetidos en las secuencias que son útiles [129], se ha añadido un paso de compresión adicional a los valores de calidad (QV, *Quality Values*). Además es posible realizar una compresión con pérdidas de los valores de calidad debido a que a la hora de usar esos datos no se suelen tener en cuenta todos los valores posibles [26], por lo que es posible hacer una cuantificación de los valores sin que influya en los resultados de los procesos a los que se sometan esas secuencias.

La compresión de los valores de calidad incluye los siguientes pasos, y sólo en los opcionales se pierde información:

1. En caso de venir la calidad en formato numérico, como es habitual cuando el formato original es un fichero FASTA+Quality, se procede a convertir los valores numéricos en caracteres, tal y como se usan en el formato FASTQ.

2. Discretización (opcional): el rango de valores de calidad (usualmente de 1 a 40 en secuenciadores NGS) se divide en intervalos de longitud personalizable, sustituyendo los valores que se encuentran dentro de cada intervalo por un sólo valor representativo del intervalo. La fórmula de discretización es:

   $QV_{discretizado}[i] = trunc(QV[i]/long\_personalizable) \times long\_personalizable.$

3. Filtrado (opcional): debido a que usar el valor de calidad en el postprocesado de secuencias NGS es usualmente impráctico [20, 129], sólo los valores bajos de la calidad tienen interés para cortar las bases de baja calidad. Suele considerarse buena calidad cuando el valor de calidad es mayor o igual a 20. En la implementación de FQbin el punto de corte es personalizable. Por tanto para las bases que se consideran de buena calidad su valor de calidad es reemplazado por el valor de corte, y para el resto de las bases el valor se pone a cero.

4. Simplificación de los valores de calidad repetidos: cuando todos los valores de calidad de las bases de una secuencia tienen el mismo valor se almacenan como un sólo valor y esto se indica en la cabecera, en el campo de longitud de la calidad.

Es decir, al leer en la cabecera un 1 en la longitud de la calidad de la secuencia se sabe que se ha usado esta simplificación, y que todos los valores de calidad son iguales. En el momento de la descompresión, al leer un sólo valor de calidad, este se repite tantas veces como indique la longitud de la secuencia.

3.6. Herramientas para manipular el formato FQbin

Se han creado diversas herramientas para la manipulación del formato FQbin. Para crear un nuevo fichero FQbin se usa `mk_fqbin` indicándole los ficheros de entrada y de salida. Para recrear el índice, en caso de que se pierda el fichero de índices, se usa `idx_fqbin`. Para crear el fichero de *hash* se usa `mk_hash`, para acceder a una secuencia de forma aleatoria se usa `read_fqbin`.

Por ejemplo, para leer todas las secuencias de un fichero FQbin y enviarlas al programa blast para ser comparadas contra una base de datos, en el ejemplo blast_database, se puede usar el comando:

```
iterate_fqbin -F file.fqbin | blastn -db blast_database
```

En caso de que otro software no permita el uso de tuberías, se puede evitar el volcar el fichero FQbin en otro mediante el uso de tuberías con nombre (*named pipes*), esto se puede realizar, por ejemplo, como se muestra a continuación:

```
    # crea una tubería con nombre (named pipe)
mk_fifo nam_pipe1
    # bowtie2 usa la tubería
bowtie2 index nam_pipe1 &
    # envía datos a la tubería
iterate_fqbin file.fqbin > nam_pipe1
    # borra la tubería
rm nam_pipe1
```

3.7. Tests

En el caso de los test de la implementación en C de nuestra librería, se usaron tres clases distintas de secuencias: un conjunto de datos generados con un Illumina, el SRR314795 (21,908,723 reads, 3.9 GB y formato FASTQ), otro con un Roche 454/FLX+,

el SRR073389 (811,509 reads, 0.9 GB y formato FASTQ) y un fichero FASTA conteniendo las secuencias sin valores de calidad de los primeros 10 cromosomas humanos (1.6 GB y formato FASTA: AC_000133.1, AC_000134.1, AC_000135.1, AC_000136.1, AC_000137.1, AC_000138.1, AC_000139.1, AC_000140.1, AC_000141.1, AC_000142.1).

FQbin se ejecutó comprimiendo los datos originales sin modificar, y se comparó con otros algoritmos de compresión general como ZIP, GZIP (un envoltorio de ZLIB), BSC (un compresor de alto rendimiento sin pérdidas [49]) y DSRC (el mejor compresor publicado para datos con el formato FASTQ [27]). Posteriormente en otro conjunto de experimentos se añadió a FQbin el filtrado de los valores de calidad que hemos explicado en la sección 3.5, llamando a estos resultados FQbin+*QV filters*, como veremos en la siguiente sección.

Los test de las secciones 3.8.1 y 3.8.2 se han realizado usando un iMac quad-core a 2.8 GHz con 8 GB de RAM, usando un almacenamiento compartido por samba a través de una red ethernet de 1 Gbit/s o de 100 Mbit/s. Las medidas de tiempo se tomaron con el comando de Unix `time`. Para el resto de las secciones de este capítulo se usa el supercomputador Picasso, descrito en la sección 1.4.5.

3.8. Resultados y discusión

Se demostrará que FQbin es un desarrollo robusto que provee un formato apropiado para el almacenamiento de datos generados con NGS, con un acceso casi instantáneo a cualquier secuencia.

3.8.1. Capacidad de compresión

Cuando se compara el factor de compresión de FQbin con el que se consigue con otros compresores tanto genéricos como específicos, como se muestra en la figura 3.3A, se observa que las mayores diferencias se consiguen en los ficheros de más tamaño (Illumina 3.9 GBytes). En particular, los compresores que más comprimen son el BSC, FQbin y el DSRC en todos los casos. En la figura 3.3B representamos por otro lado el tiempo de compresión para los mismos compresores. Podemos observar que BSC a pesar de ofrecer buenos factores de compresión, incrementa notablemente el tiempo con el tamaño del fichero, lo que indica que es sensible al tamaño de los datos de entrada. DSRC es un compresor competitivo porque consigue mayores factores de compresión en menos tiempo que FQbin, sin embargo no es capaz de manejar ficheros con formato FASTA (de ahí que no haya resultados con el fichero Fasta en la figura). Si consideramos la optimización FQbin+ *QV filters*, podemos observar que en ficheros de gran tamaño

como Illumina es el que consigue mayor factor de compresión, lo que confirma la naturaleza repetitiva de los datos de calidad, y las ventajas de realizar un filtrado [129]. Sin embargo, DSRC sigue siendo más rápido cuando se compara con esta implementación de FQbin. Por otro lado, gZip que usa zlib, la misma librería de compresión que FQbin, no es mejor que éste comprimiendo. El motivo es que FQbin olvida todo lo aprendido al iniciar un nuevo bloque, y esto, debido a la alta aleatoriedad de los datos genéticos, ayuda a comprimir mejor. En cuanto a velocidad, gZip es ligeramente más rápido que FQbin debido a que este último tiene la sobrecarga de generar y comprimir los índices a los datos.

En la bibliografía encontramos SpeedGene, que dice tener un factor de compresión que va desde 16 a varios cientos [102], pero no se puede comparar con FQbin, ya que SpeedGene comprime basándose en un genoma de referencia para la realización de estudios de asociación de genoma completo, mientras FQbin está enfocado a almacenar los datos originales completos.

3.8.2. Lectura de los ficheros comprimidos

Podemos ver en la figura 3.3 que el tamaño de un fichero FQbin es entre 6 y 10 veces menor que el de un FASTQ generado con un Illumina, pero su contenido, como se puede ver en la figura 3.2, es más complejo, lo que podría llevar a pensar en que leer todos los datos puede ser más lento con el formato FQbin que con el FASTQ original. Usando los mismos ficheros con formato FASTQ que usamos anteriormente (454/FLX+ e Illumina), en la figura 3.4 representamos la ratio entre el tiempo que se tarda en leer todos los datos usando el formato FASTQ y el tiempo que tardamos el leer esos datos a partir del fichero comprimido con FQbin (aceleración), tanto en redes de 1 Gbit/s como 100MBit/s.

FQbin lee el fichero 454/FLX+ completo en 26s en una red de 1 Gbit/s, siendo 2 veces más rápido que el acceso directo al fichero FASTQ original, que tardó 63 segundos, y mucho más rápido que el acceso con DSRC, que también se midió y tardó un total de 230 segundos. Hay que tener en cuenta que los ficheros comprimidos con otros programas como DSRC o zip, gZip y BSC deben ser convertidos primero en ficheros FASTQ, y a continuación estos ficheros FASTQ ya pueden ser procesados, por lo que con estos formatos siempre será más lento el acceso que con un fichero FASTQ directamente. Con tamaño de ficheros mayores, como es el caso de Illumina, FQbin consigue mayores aceleraciones cuando se compara con FASTQ. Destaca el caso en el que 4 procesos concurrentes están leyendo el mismo fichero (ver la barras 4 X), y en particular cuando ficheros grandes son leídos usando redes lentas de 100 Mbit/s. En ese caso, leer los datos de Illumina disminuyó el tiempo de lectura desde los 3329 segundos del fichero original

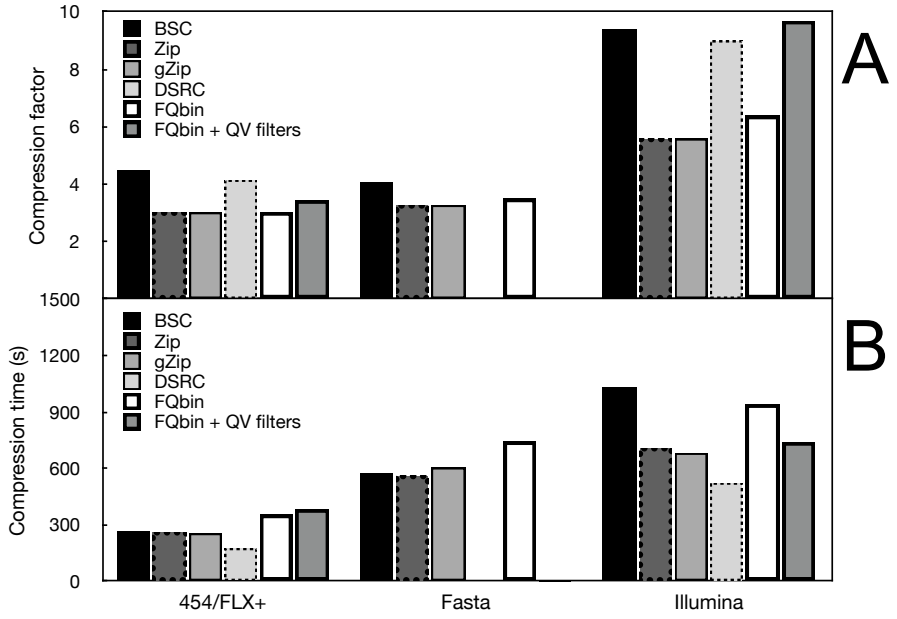


Figura 3.3: Comparación entre distintos algoritmos de compresión. A: Factor de compresión respecto al fichero sin comprimir calculado como el tamaño del fichero original dividido entre el tamaño del fichero comprimido (mayor es mejor). B: Tiempo que se tarda en realizar la compresión (menor es mejor).

en formato FASTQ a 1137 segundos en formato FQbin, consiguiendo una aceleración de 2.9x. En resumen, FQbin no sólo ahorra espacio de disco, sino que además ahorra tiempo en el acceso a los datos, acelerando la carga de las secuencias.

Los compresores con los que se compara FQbin en la figura 3.3 requerían la descompresión del fichero completo para acceder a una secuencia cualquiera, pero DSRC y FQbin permiten acceder a cualquier secuencia sin necesitar la descompresión del fichero completo. La eficiencia en el acceso aleatorio se puede ver comparando el tiempo de acceder a la primera y última secuencia del fichero, tal y como se muestra en la Tabla 3.1. Como era de esperar, en FASTQ el acceso secuencial al último elemento tarda más cuanto más grande es el fichero. Gracias al fichero *hash* que se añade a FQbin para acelerar los accesos, se obtiene un acceso prácticamente instantáneo a cualquier secuencia contenida en FQbin, mejor incluso que con el formato DSRC. Por ejemplo, para el acceso al último elemento, FQbin obtiene una aceleración de 86x con el fichero 454/FLX+ y de

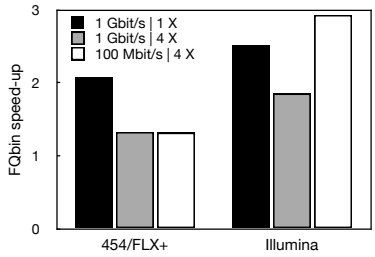


Figura 3.4: Aceleración obtenida con FQbin (mayor es mejor) al leer un fichero completo respecto a leerlo en formato FASTQ cuando se leen ficheros 454/FLX+ e Illumina con la caché del sistema de ficheros deshabilitada y usando redes de 1 Gbit/s y 100 Mbit/s. 1 X indica que sólo un proceso está leyendo el fichero; 4 X significa que cuatro procesos están accediendo el fichero a la vez.

415x para el fichero Illumina, cuando se compara con el formato FASTQ.

Tabla 3.1: Tiempo de acceso para la primera y última secuencia del fichero

Formato testeado	454/FLX+	Illumina
FASTQ 1st	0.19	0.14
FASTQ last	37.08	176.30
DSRC 1st	1.36	0.74
DSRC last	1.06	0.79
FQbin 1st	0.219	0.242
FQbin last	0.429	0.423

Como conclusión, FQbin es un formato apropiado para almacenar y acceder rápidamente a secuencias, ya que el proceso de lectura de una secuencia, como vemos en la Tabla 3.1 o del fichero completo (como vimos en la discusión de la figura 3.4) requiere menos tiempo que con cualquier otro compresor de los analizados, e incluso que leer directamente el formato FASTQ.

3.8.3. Factores que afectan el rendimiento de la lectura en FQbin

A la hora de leer un dato en el formato FQbin, el acceso aleatorio es prácticamente instantáneo, como acabamos de ver, ya que tras haber generado el fichero de *hash* y ordenado el fichero de índice, sólo se requiere el acceso al fichero de *hash*, que es muy pequeño. En caso de no generar el fichero de *hash* el acceso es algo más lento, ya que

va desde menos de una décima de segundo si se lee la primera secuencia, hasta el peor caso, cuando es necesario descomprimir el fichero de índices completo.

Para el caso de la lectura completa la mejora proporcionada por el formato FQbin respecto a leer el fichero original dependerá del rendimiento que dé el sistema de ficheros: a mayor rendimiento del sistema de ficheros menor ganancia en velocidad se va a obtener de leer el fichero comprimido. De la misma forma cuanto más lenta es la red de interconexión mayor es la ganancia de leerlo comprimido respecto a leer el fichero original. En la tabla 3.2 podemos ver una comparativa de accesos desde el sistema de ficheros (acceso a disco) o usando la caché de disco, tanto para el formato FQbin (en las dos primeras columnas) como para los datos originales en formato FASTA con calidades (última columna). En este experimento los ficheros originales ocupaban 2.4 Gigabytes el fichero de secuencias y 7 Gigabytes el fichero de calidad. Se ha evaluado tanto el acceso al disco a través de una red infiniband FDR (54 Gigabits por segundo) y Gigabit ethernet. Así, el tiempo que se tardaría en leer el fichero en formato FQbin sería el indicado en la primera columna, 1.5 segundos para infiniband y 15 segundos para ethernet. Mientras que la lectura del fichero original FASTA tardará lo indicado en la última columna, 11 segundos en infiniband y 81.6 usando ethernet. La columna central incluye la lectura y descompresión del fichero FQbin. La diferencia de tiempos que aparece en la tabla entre infiniband y ethernet cuando los datos están en caché, es debida al uso de distintos sistemas en ambos casos: aunque el sistema de almacenamiento es el mismo, las CPUs son distintas. Los detalles se pueden encontrar en la sección 1.4.5.

	lectura FQbin		lectura		lectura del fichero FASTA	
	infiniband	ethernet	infiniband	ethernet	infiniband	ethernet
directo a disco	1,5	15	60	72	11	81,6
en caché	0,22	1	59	72	2	5

Tabla 3.2: Tiempo en segundos de la lectura secuencial del fichero completo, usando redes infiniband FDR (54 Gbits/s) y ethernet (1 Gbit/s) cuando se lee de disco, o estando los datos en caché. 1ª columna: formato FQbin, 2ª columna: incluyendo la descompresión FQbin; 3º columna: formato FASTA.

Como vemos, el tiempo de descompresión es el factor limitante en el acceso si se quiere recuperar el fichero completo. Así encontramos que usando una red infiniband es más rápido el acceso a los datos directamente desde el fichero descomprimido que si hay que descomprimirlos, ya que el tiempo ahorrado en el acceso al sistema de ficheros es menor que el que se necesita para realizar la descompresión. Se podría implementar un descompresor paralelo, pero por el tipo de procesamiento que se suele realizar sobre las secuencias genéticas, cuando se procesa un fichero completo sus accesos son fácilmente incorporados a un *pipeline*, solapándose con los cálculos, ya que al solicitar un nuevo dato se descomprime un bloque completo, por lo que al pedir el dato siguiente

ya lo encontrará descomprimido. De esta manera la descompresión se habrá realizado concurrentemente con el trabajo de la aplicación que solicita los datos.

3.8.4. Robustez del formato FQbin

Los ordenadores personales cada día contienen más RAM, y la gran mayoría de ellos usan RAM sin códigos de detección de errores, por lo que el problema de la corrupción de la RAM no es irrelevante [19]. Además, como el procesamiento de los datos NGS requiere ficheros de varios Gigabytes, la posibilidad de que haya corrupción en los datos aumenta. Por ejemplo, un momento en el que se pueden producir errores es cuando se copian ficheros, ya que los datos se leen desde el disco, se almacenan en RAM y posteriormente se escriben a disco de nuevo: si hay un fallo en la RAM no se detectará si la RAM no tiene ningún sistema de detección de errores. El problema es que cuando aparece un error de corrupción de este tipo (por ejemplo, al copiar un fichero) el usuario no es avisado de que ha habido un problema. Como FQbin está basado en la librería zlib, y esta librería usa códigos CRC (*Checking Redundancy Code*) internamente para la verificación de la integridad, FQbin avisa al usuario cuando detecta corrupción en los datos, permitiendo que éste descarte los datos y solicite de nuevo la copia del fichero original.

Una corrupción de un sólo bit de un fichero de texto no impide su lectura, pero aún así un cambio mínimo en una secuencia va a provocar consecuencias indeseables. Esa misma corrupción en un fichero binario (por ejemplo, en un fichero comprimido) puede hacer que no se pueda recuperar el contenido del fichero completo. Este problema se minimiza en FQbin al almacenar los datos comprimidos en varios bloques independientes (como se puede ver en la figura 3.2). De esta forma, la corrupción de un bloque no afecta al resto y la mayoría de las secuencias se podrían recuperar.

3.9. Motivación para una nueva implementación en Chapel

Tras realizar la implementación en C del formato FQbin, quedaron varios aspectos que mejorar. Recordemos que esta implementación que hemos analizado es secuencial. Por lo tanto, uno de los aspectos que se puede mejorar es la reducción del tiempo de que se invierte en las operaciones de esta librería mediante la paralelización. Más concretamente, alrededor del 98 % del tiempo que tarda la librería FQbin en el almacenamiento de los datos, se consume en la operación de compresión, por lo que es la primera función a optimizar. En nuestro caso, volveremos a usar Chapel como herramienta de paraleli-

zación, por las ventajas descritas en el capítulo anterior. En esta sección abordaremos por tanto la optimización en Chapel, de la operación de compresión de un fichero que contiene secuencias genéticas y sus calidades, al formato FQbin.

3.9.1. Escritura de los datos

En la implementación C, que es secuencial, se lee en streaming los datos del fichero original, y conforme se leen se van comprimiendo. La función de compresión, al usar la librería zlib, ya escribe directamente en el fichero destino, ya que la función `int gzwrite (gzFile file, voidpc buf, unsigned len);` que es la que se invoca, tiene el descriptor del fichero destino en la estructura `gzFile`. Además en `buf` están los datos que se quieren comprimir, y `len` indica el número de bytes que se van a comprimir. La función devuelve el número de bytes descomprimidos que se han escrito tras ser comprimidos en el fichero de salida, o 0 si hay un error. En la versión C secuencial se van leyendo los datos, se procesan si lo necesitan (sobre todo los datos de calidad), y después se van escribiendo en el fichero de salida, con lo que la cantidad de memoria usada es muy reducida, ya que en un momento dado sólo hace falta tener en memoria el trozo que se está comprimiendo.

En la implementación Chapel paralela los ficheros de entrada se parten en varios trozos, encargándose cada *thread* de uno de los trozos. Esto tiene los siguientes inconvenientes: i) los ficheros no se pueden partir por cualquier sitio, ya que eso daría problemas a la hora de recuperar la información, así que sólo se debe de partir al inicio de una nueva secuencia; ii) no se puede saber a priori las posiciones que ocuparán los datos generados por los distintos *threads* en el fichero final, ya que hasta que no se sepa el tamaño de todos los trozos anteriores a uno dado no se sabrá la posición de éste.

Partición del fichero de datos.

Para solucionar el primer problema dividimos el tamaño total del fichero que contiene las secuencias entre el número de *threads* que se van a usar, y después buscamos, a partir de los desplazamientos obtenidos en la operación anterior, el comienzo de la siguiente secuencia más cercana en el fichero. Cuando la encontramos usaremos su desplazamiento como *offset* final del *thread* siguiente. Así, si el fichero ocupa 4000 bytes y hay 5 *threads*, entonces el punto de comienzo de cada thread sería la posición 0, 800, 1600, 2400, 3200, A continuación cada thread busca, a partir de esas posiciones, el comienzo de una nueva secuencia, que identificaría como el offset final del thread siguiente.

Otro problema que aparece es cuando hay un fichero independiente con las calidades, ya que en ese caso cada secuencia tiene sus bases en el fichero FASTA y la calidad en el fichero QUAL. Recordemos que en FQbin al comprimir se adjunta a cada secuencia su

calidad. Cuando se ejecuta FQbin en secuencial, simplemente se abren los ficheros de bases y de calidad y se van procesando secuencialmente. Sin embargo, al paralelizar el problema es que las posiciones de ambos ficheros no coinciden y no es posible calcular la posición del fichero de calidad a partir de las posiciones del fichero de secuencias ni al revés. La solución más obvia sería buscar la primera secuencia de cada *thread* en el fichero de calidad comenzando desde el principio, pero eso tiene el problema de requerir la lectura del fichero casi completo. La solución que hemos encontrado a este problema es usar una heurística para encontrar cada una de las secuencias en el fichero de calidad sin necesidad de leerlo entero. Como el fichero de calidad almacena las mismas secuencias en el mismo orden que el de bases, y como la longitud que ocupa cada secuencia y sus calidades es equivalente, podemos suponer la posición de de inicio de cada *thread* en el fichero de calidad ligeramente desplazada hacia posiciones inferiores que la que se obtiene en el fichero de bases. En caso de no encontrarse el comienzo de la calidad para la siguiente secuencia, tendrá que buscarse desde el principio.

Posiciones en el fichero final.

El otro problema que aparece al paralelizar es que cuando se escribe de forma secuencial el fichero final, conforme se van comprimiendo los datos se van calculando las posiciones de cada bloque, pero al realizar la compresión en paralelo no se sabe realmente en qué posiciones estará cada bloque. Además no se podrá crear el fichero de índices hasta que se termine de escribir, ya que no se sabe donde estará cada secuencia en el fichero final.

Para resolver el problema de la escritura cada thread escribe en un fichero el trozo que comprime. Para ello usamos la función `shm_open` del POSIX [35], que se basa en el sistema de ficheros en memoria de linux, en el que los ficheros creados residen en memoria, y el acceso tanto al crear como leer dichos ficheros, es muy rápido. Por otro lado, no queda más remedio que esperar al final cuando se compone el fichero final a partir de cada fichero con los bloques comprimidos, para generar el fichero de índices. Aunque en un principio eso requería una relectura de las secuencias tras escribirlas, o ir procesando los datos conforme se van escribiendo para ir generando el fichero de índices a la vez que el principal, la solución final adoptada es crear unas estructuras de datos en las que se van apuntando las posiciones relativas al inicio del trozo que procesa cada *thread*. Cuando ya se conoce la posición en la que se va a escribir ese trozo, sólo hay que añadirle esa posición a las posiciones relativas de los bloques que cada *thread* ha procesado.

En las siguientes secciones estudiamos el rendimiento de distintas implementaciones de FQbin en Chapel. El sistema usado para las pruebas está detallado en 1.4.5 y a partir de este momento será el sistema de referencia para nuestros experimentos. El motivo de usar ese equipo es que tiene una arquitectura bien balanceada entre sistema de almace-

namiento y capacidad de cómputo. Además se usa intensivamente para el procesamiento de datos bioinformáticos.

3.9.2. Implementación de FQbin en un *locale*

En esta sección nos centramos en el estudio de la eficiencia de la implementación Chapel en un nodo del sistema (o *locale* en terminología Chapel). En la figura 3.5 mostramos la declaración de las principales estructuras de datos que necesitamos en Chapel para implementar FQbin en un *locale*.

```
var chunkDom = {0..#parallel_threads};
var chunkArr: [chunkDom] chunk;

var indexForChunkDom = {0..#parallel_threads};
var indexDom = {0..#seqs_per_stream};
var indexArr: [indexForChunkDom][indexDom] index_el;
```

Figura 3.5: Declaración en Chapel de las variables usadas en la librería FQbin en un sólo *locale*.

Presentamos en la figura 3.6 la escalabilidad obtenida por nuestra implementación en Chapel del compresor FQbin cuando se procesa un fichero con formato FASTA con calidades en nuestro sistema de referencia. En la figura representamos el tiempo en segundos así como la aceleración con respecto a la implementación original en C de FQbin, cuando el número de cores varía de 1 a 16, que en nuestro caso es el número máximo de cores de un *locale*. El tamaño de los ficheros originales es de 2.466.636.348 bytes el de secuencias, y 7.044.904.240 bytes el de calidad (2.4 y 7 GB aproximadamente). La implementación original en C de FQbin tarda 15m26s y consigue un factor de compresión del 18.94 %. También observamos que la implementación Chapel para 1 core tiene una pérdida de eficiencia del 18 % con respecto a la versión C (tarda 18m35s), pero mantiene una escalabilidad aceptable para el número de cores estudiado. En este caso, la E/S aún tiene poca influencia en la operación de compresión, siendo el overhead de manejo de estructuras de datos introducido por Chapel el responsable de que la eficiencia paralela para 16 cores baje al 48 %.

3.9.3. Implementación de FQbin distribuida en varios *locales*

Una vez realizada la implementación en un *locale* del compresor FQbin, y de comprobar como escala, hemos llevado a cabo los cambios en el código fuente para conseguir

una implementación distribuida. Para ello hemos de cambiar primero las estructuras de datos definidas en la figura 3.5, convirtiéndolas en distribuciones de Chapel (que se explicaron en la sección 1.3.1). En la figura 3.7 vemos la declaración de variables y los arrays distribuidos para esta nueva versión.

A continuación sólo hay que cambiar algunos bucles para que la iteración de cada elemento se ejecute en el *locale* en el que esté contenido. Esto se realiza como se muestra en la figura 3.8. Hay que tener en cuenta que `chunkArr` es un array que está distribuido entre los *locales* gracias a la distribución definida en la figura 3.7.

En la figura 3.9 podemos ver los tiempos que tarda la librería FQbin en comprimir de forma distribuida el fichero con formato FASTA y calidades que se usó como entrada en los experimentos del apartado previo. En el eje y se indica el tiempo de compresión y escritura de los ficheros FQbin, y en el eje x se indica el número de cores. Se condujeron 8 experimentos diferentes: en cada uno se varió el número de nodos (*locales*) usados, entre uno y ocho. En particular, en cada experimento se incrementaba el número de *threads* desde 1 hasta alcanzar el número máximo de cores en ese escenario. Recordemos que en esta arquitectura hay 16 cores por *locale*, por lo tanto el número máximo de cores en cada escenario viene dado por número de *locales* $\times$ 16. En la gráfica se representa hasta 80 cores. Inicialmente, los *threads* se reparten de forma uniforme entre los nodos

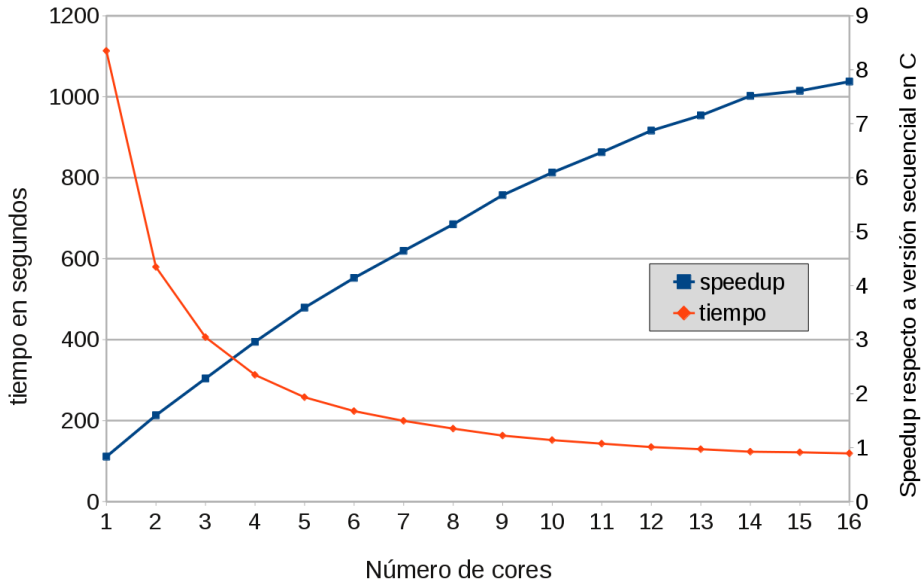


Figura 3.6: Tiempos y aceleración de la implementación Chapel de FQbin para un *locale*.

```
var Space={0..#parallel_threads};

var chunkDom = Space dmapped new dmap(new Block(boundingBox=Space));
var chunkArr: [chunkDom] chunk;

var indexForChunkDom = Space dmapped Block(boundingBox=Space);
var indexDom = {0..#seqs_per_stream};
var indexArr: [indexForChunkDom][indexDom] index_el;
```

Figura 3.7: Declaración en Chapel de las variables usadas en la librería FQbin distribuida.

```
// Bucle original
forall i in 0..#parallel_threads do
        ...

// Bucle anterior cambiado para que sea distribuido
forall L in Locales {
        on L {
                const indices = chunkArr.localSubdomain();
                forall i in indices
                        ...
```

Figura 3.8: Cambios a efectuar en los bucles para que pasen a ejecutarse en todos los *locales* de manera distribuida.

solicitados. Por ejemplo, 16 *threads* pueden corresponder a 1 sólo nodo con 16 cores, a 2 nodos con 8 cores cada uno, a 4 nodos con 4 cores, etc. Por ello se han usado distintos colores para distinguir el número de nodos.

Un primer resultado que llama la atención en la figura 3.9 es que en la implementación Chapel distribuida de FQbin tiene un comportamiento similar a la implementación en un sólo *locale* (ver figura 3.6), pero sólo en configuraciones de hasta 8 *threads* por nodo. Como podemos ver en la figura, en configuraciones de más de 8 *threads* por nodo los tiempos se degradan en cualquier escenario. La razón de esta degradación está relacionada con la planificación de los *threads* que dentro de cada nodo realiza GASNet. Hemos observado que GASNet asigna (*pinning*) cada *thread* a un core concreto dentro del nodo. Mientras el número de *threads* es menor que 8 asigna correctamente un *thread* por core. Sin embargo, a partir de 8 comienza a asignar más de 1 thread por core (a pesar de que aún queden cores libres) lo que provoca la *oversubscription* del core y que empiecen a aparecer problemas de contención en ese core y por lo tanto el desbalanceo

del trabajo entre esos *threads* y el resto, lo que degrada significativamente los tiempos.

Efecto del envío y recepción de mensajes con GASNet

Aunque en principio las transferencias RDMA no usan la CPU ni el sistema operativo de ninguno de los nodos intervinientes, en la práctica hay que preparar las transferencias, y sobre todo hacer un *polling* sobre la cola de terminación (*completion queue*), siendo la intención la de minimizar la latencia cuando termina una operación de RDMA. Se podría realizar con una interrupción, pero eso incrementaría la latencia, ya que habría que preparar la interrupción, realizar un cambio de contexto, afectando a las cachés y habría que procesar la interrupción, necesitando otro cambio de contexto. Por lo tanto, el realizar *polling* para disminuir la latencia de las comunicaciones y de la E/S está documentado en diversos *white papers* de buenas prácticas, como por ejemplo [126], [125], y [62].

Al implementar GASNet decidieron usar *polling* en el propio espacio de usuario, evitando el tener que realizar un cambio de contexto al pasar a modo kernel. Pero este *polling* continuo de la cola de terminación requiere de un *thread* específico que ocupa totalmente un core, lo que puede afectar al resto de *threads* de la aplicación si alguno de ellos es mapeado en el core que realiza el *polling*. Este detalle de la implementación interna de GASNet es el que explica los puntos atípicos que vemos en la figura 3.9.

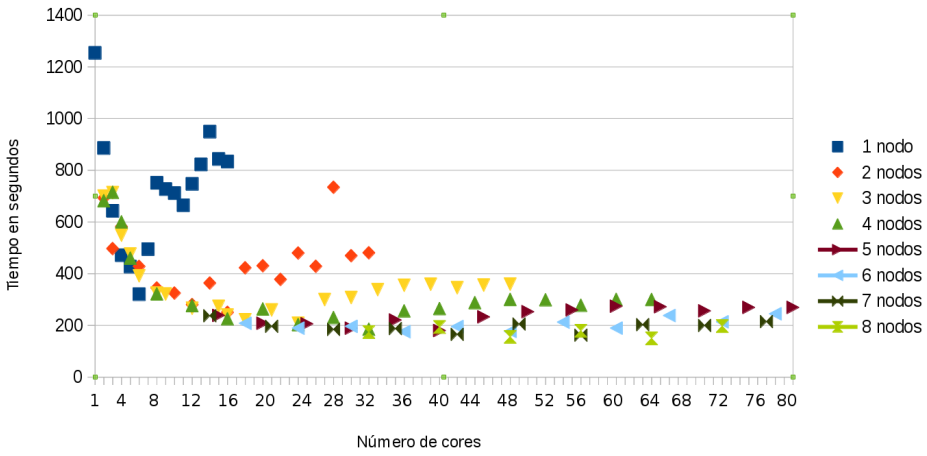


Figura 3.9: Tiempos de la implementación distribuida en Chapel de FQbin. Se representan por número de nodos, entre uno y ocho. En estas pruebas se ha usado GASNet.

3.10. Conclusiones

En ese capítulo se ha presentado un formato de datos nuevo, FQbin, para el almacenamiento y acceso eficiente a datos obtenidos por secuenciación genética de última generación. Se ha diseñado una librería para su fácil manipulación, ofreciendo las siguientes ventajas:

1. Provee el almacenamiento de secuencias y/o valores de calidad y/o extras en el mismo fichero, expandiendo su uso más allá de las secuencias de nucleótidos. De hecho, la base de datos EuroPineDB [50] usa la librería FQbin para recuperar *contigs* de grandes ficheros ACE.
2. Provee de funciones de recuperación directa soportando la E/S de datos FASTA, QUAL y FASTQ.
3. Permite que la información comprimida sea fácilmente transferida por pipeline desde/hacia otros programas, asegurando la compatibilidad con el software actual sin necesidad de recodificar los datos.
4. El decremento en el tamaño de los ficheros se puede comparar favorablemente con el que se obtiene en otros algoritmos que representan el estado del arte.
5. Lee la misma cantidad de datos en menos tiempo que otros y provee un acceso casi instantáneo a cualquier secuencia por su nombre, independientemente de su posición.

Hemos comprobado que gracias a FQbin el acceso a secuencias concretas de forma aleatoria se acelera gracias al uso de índices y de tablas *hash*, como también se aceleran la copia y manipulación en general de los ficheros completos. Pero mientras esperábamos que el acceso a todos los datos contenidos en un fichero FQbin fuera más rápido que al fichero original, esto no es siempre cierto, ya que depende de la relación entre dos factores, la velocidad de la CPU, que es quién deberá descomprimir los datos, y la velocidad del acceso a los datos en el almacenamiento secundario.

Tras la definición del formato, la implementación de la librería en C y el estudio de su rendimiento, hemos usado el lenguaje de programación Chapel para convertir la librería secuencial original en una paralela, consiguiendo una buena escalabilidad cuando se ejecuta dentro de un nodo (o *locale*). Cuando hemos adaptado esta versión para permitir la ejecución distribuida en varios nodos hemos descubierto ciertas ineficiencias debidas a una incorrecta planificación interna de *threads* en la librería GASNet (que es la que se encarga de las comunicaciones remotas de los mensajes entre nodos) así como al efecto del *thread* de *polling* que gestiona el envío y recepción de mensajes en esa librería.

Para finalizar queremos comentar que nuestra solución [59] aparece referenciada en dos patentes, en el año 2014 la patente número 8.847.799 [66], y en 2015 la patente 8.976.049 [65], con título *"Methods and systems for storing sequence read data"*. Además se ha descargado la gema de ruby que contiene la librería FQBin [58] más de 2000 veces a día de hoy, de las cuales más de 400 descargas pertenecen a la última versión.

4 Conclusiones

En este trabajo hemos presentado dos contribuciones principales. Por un lado, proponemos una librería de funciones de E/S de arrays distribuidos en el lenguaje de programación de alta productividad Chapel así como el estudio del rendimiento de dicha librería cuando ésta se configura para explotar distintos grados de paralelismo en el sistema. Por otro lado, diseñamos una librería para el almacenamiento y acceso a secuencias genéticas que reduce el coste del almacenamiento y aumenta el rendimiento y la fiabilidad en los accesos.

4.1. Librería para el almacenamiento distribuido de arrays

El primer criterio de diseño a la hora de crear una librería de E/S paralela ha sido el de garantizar una interfaz de alta productividad. Tras analizar el estado del arte, hemos encontrado que el principal problema a la hora de proporcionar una interfaz amigable al programador viene derivado de la distribución de los datos, tanto entre los distintos nodos como dentro de cada nodo. Gracias a la facilidad de definir arrays distribuidos en Chapel, hemos conseguido crear una interfaz sencilla para realizar la E/S, ampliando la interfaz de las distribuciones de Chapel con operaciones de E/S. De esta forma, la información relativa a la distribución de los datos es accesible desde la librería de E/S al tiempo que se ocultan los detalles de implementación (paso de mensajes, gestión de buffers, etc) al programador, lo cual impacta muy positivamente en un incremento de la productividad de nuestra solución.

Otro criterio de diseño ha consistido en mantener una alta flexibilidad en la librería y la portabilidad de los datos almacenados en fichero. En concreto, nuestra solución per-

mite realizar la escritura de un array distribuido en un único archivo, el cual puede ser leído posteriormente a otro array que siga otra distribución diferente. En otras palabras, nuestra librería desacopla el formato del fichero de la distribución usada en los arrays que almacena. En aras de la portabilidad hemos descartado la alternativa de generar varios ficheros para almacenar los resultados de salida de una aplicación. Esta posibilidad aceleraría la escritura en paralelo, pero complica sobremanera la posterior lectura si se usa una distribución de datos diferente a la usada en escritura, lo que podría suceder simplemente cambiando el número de nodos usados por la aplicación.

Por último, aunque no menos importante, el criterio de diseño relacionado con las prestaciones ha requerido la mayor inversión de tiempo en investigación y desarrollo. Para implementar eficientemente el interfaz anterior hemos buscado en las distintas fases de una operación de E/S todos los niveles en los que es posible explotar paralelismo. Hemos encontrado varios niveles que han resultado ser ortogonales y que por tanto se han podido combinar. Las distintas posibilidades se han validado experimentalmente para seleccionar la mejor solución. Tras realizar un análisis exhaustivo hemos encontrado que el rendimiento óptimo se obtiene al paralelizar tanto las escrituras en disco como la agregación de datos. El solapar la escritura de los datos en disco y la agregación de los datos desde los clientes aumenta de forma no significativa el rendimiento. Sin embargo, intentar explotar más paralelismo incrementando el número de agregadores por OST termina impactando negativamente en el rendimiento.

También en relación con las prestaciones de la E/S, hemos comprobado que los *overheads* necesarios para realizar las comunicaciones son una fuente cada vez más importante de pérdida de rendimiento. Para paliar estos *overheads* hemos propuesto una optimización de las operaciones de copia de arrays en Chapel. Esta mejora consiste en agregar automáticamente los datos para explotar mejor el ancho de banda de las comunicaciones lo que ha repercutido muy positivamente en la gestión de los *buffers* usados en las operaciones de E/S.

4.1.1. Contribuciones

Esta línea ha dado lugar a las siguientes publicaciones:

Rafael Larrosa Jiménez, Rafael Asenjo Plaza, Angeles González Navarro, and Emilio López Zapata. Implementing a chapel library for parallel io. In *Actas de las XXI Jornadas de Paralelismo*, pages 323–330, 2010.

Rafael Larrosa, Rafael Asenjo, Angeles G. Navarro, and Bradford L. Chamberlain. A first implementation of parallel io in chapel for block data distribution. In *PARCO*, volume 22 of *Advances in Parallel Computing*, pages 447–454. IOS Press, 2011.

Alberto Sanz, Rafael Asenjo, Juan Lopez, Rafael Larrosa, Angeles Navarro, Vassily Litvinov, Sung-Eun Choi, and Bradford L Chamberlain. Global data re-allocation via communication aggregation in chapel. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 235–242. IEEE, 2012.

Además, varias de las optimizaciones del runtime de Chapel surgidas durante el desarrollo de esta tesis se incorporaron al repositorio de dominio público disponible en GitHub (https://github.com/chapel-lang/). En particular, estas mejoras están disponibles desde las versiones 1.8 y 1.9 de Chapel.

4.2. Librería para el almacenamiento optimizado de datos de secuenciación genética

Tras estudiar los formatos existentes de almacenamiento de secuencias genéticas así como los casos de uso, hemos propuesto un formato que aúna alta capacidad de compresión y rápido acceso aleatorio. Este nuevo formato llamado FQbin es funcionalmente competitivo con el estándar de facto, el formato FASTQ, ya que incluye todos los campos de datos opcionales de éste (como el campo de calidad o el campo con información extra), pero lo aventaja en productividad y en ahorro de espacio en disco.

Uno de los puntos más importantes de la librería FQbin desde el punto de vista de la productividad es que no hace falta modificar las aplicaciones ya existentes para usar el nuevo formato. Generalmente cuando almacenamos los datos en un formato pero la aplicación que los necesita está preparada para leer otro formato diferente, la solución habitual consiste en realizar una conversión de formatos. Esto suele implicar una descompresión del archivo y el consiguiente consumo de recursos, tanto de espacio en almacenamiento secundario como de ancho de banda de E/S y de cálculo para la descompresión. Nuestra implementación evita la creación del fichero descomprimido en disco mediante una estrategia basada en un *stream* de datos procesados por bloques y mantenidos temporalmente en memoria. El formato FQbin facilita ese proceso al tiempo que permite un rápido acceso aleatorio a las secuencias genéticas. Además los ficheros con formato FQbin ocupan entre 5 y 10 veces menos espacio que los datos originales.

El hecho de que las aplicaciones virtualmente procesen los datos en su versión comprimida (no es necesario descomprimir todo el fichero para que sea procesado) repercute positivamente en varios aspectos ya que se reduce el impacto de la aplicación sobre la jerarquía de memoria, la red de comunicaciones y los propios dispositivos de almacenamiento.

Además de la implementación secuencial se ha realizado la implementación paralela en Chapel de la escritura de ficheros con formato FQbin. La escalabilidad observada es prácticamente lineal cuando se usa memoria compartida. Sin embargo, una decisión de diseño de la capa de comunicaciones GASNet en la que se basa Chapel y que se escapa a nuestro control da lugar a pérdidas de escalabilidad en arquitecturas de memoria distribuida.

4.2.1. Contribuciones

Esta línea ha dado lugar a la siguiente publicación:

Darío Guerrero-Fernández, Rafael Larrosa, and M. Gonzalo Claros. FQbin, a compatible and optimized format for storing and managing sequence data. In *International Work-Conference on Bioinformatics and Biomedical Engineering, IWBBIO 2013, Granada, Spain, March 18-20, 2013. Proceedings*, pages 337–344. 2013.

Además, la librería FQBin [59] ha sido citada en dos patentes en los EEUU, en 2014 en la patente 8,847,799 [66] y en 2015 en la patente 8,976,049 [65], además la gema de ruby de la librería FQBin, disponible en RubyGems (https://rubygems.org/gems/scbi_fqbin/), ha tenido más de 2000 descargas hasta ahora [58].

4.3. Trabajos futuros

Durante el transcurso del presente trabajo han aparecido oportunidades de mejorar las aportaciones propuestas. A continuación presentamos las que consideramos más interesantes:

- La tecnología basada en *burst buffers* [83] ya se integra en algunos sistemas de almacenamiento pioneros y se prevé su uso también en HPC, como por ejemplo en la plataforma Summit (ver sección A.1.3). Este tipo de buffers asimilan las ráfagas que se producen en la escritura, que posteriormente van pasando al almacenamiento físico. De esta forma el usuario percibe una velocidad de escritura superior. Pretendemos estudiar el impacto de esta tecnología en nuestra librería de E/S, y en qué medida puede ayudar a mejorar el rendimiento.
- En relación con la librería FQbin queremos investigar como afectan los accesos de lectura al mismo fichero desde varios nodos de cálculo. Analizaremos los distintos puntos donde se puede realizar la descompresión y como se alteran las comunicaciones realizadas dependiendo del lugar donde se produce la descompresión

(servidores de disco, del sistema de fichero, capa *burst buffer* o nodos de cálculo). Es decir, buscaremos el balance óptimo entre la carga de las CPUs, del sistema de almacenamiento y de las comunicaciones.

A Otros equipos HPC

A.1. Sistemas usados de los que no se presentan resultados

En la sección 1.4 hemos descrito los equipos que hemos usado para obtener los resultados que hemos presentado. Pero además durante el desarrollo de esta tesis se han usado otros equipos, en los que se han obtenido resultados que finalmente no se han presentado, en el presente apéndice describimos esos equipos y los motivos por los que no se han presentado los resultados que obtuvimos en ellos.

A continuación describimos primero a Jaguar. Cuando comenzamos a usarlo ya teníamos mediciones obtenidas en Crow, cuya descripción se puede encontrar en la sección 1.4.1, y conocíamos el sistema Crow con detalle, por lo que lo usamos para escribir [78], continuando el trabajo en Jaguar. Los resultados en Jaguar, y en Titan cuando le sustituyó, eran demasiado bajos y variables debido a varios factores, que explicamos en detalle a continuación, pero que se pueden resumir en que con una topología torus 3D de red es muy complicado el intentar establecer relaciones causa-efecto en la E/S. A eso hay que añadir la intensa carga tanto de cálculo como de E/S que soportan los equipos, lo que estresa las redes internas de comunicación. Además el ancho de banda decrece y la latencia aumenta de forma muy importante conforme los nodos que se comunican están más lejanos.

Aunque entre las características de la implementación Gemini de un Torus 3D tenemos ventajas tan importantes como la posibilidad de usar distintos caminos para comunicar dos nodos, multiplicando el ancho de banda entre ambos, al enviarse los paquetes por varios caminos [3], tiene el inconveniente de que conforme los nodos a comunicar están

más lejanos los paquetes durante su camino se ven afectados por más comunicaciones, por lo que el ancho de banda efectivo disminuye.

En [5] se explica que no se suele tener en cuenta la contención de las redes de interconexión entre nodos en los grandes sistemas HPC, como ejemplo de la poca importancia que se le da, no sólo en investigación si no en los sistemas reales, mencionan que el planificador de trabajos que llevan los Cray XT (Como Jaguar) no tiene en cuenta la topología del sistema a la hora de planificar los trabajos, y además no es nada sencillo el conseguir esa información, e incluso la implementación de las funciones `MPI_Cart` no tiene en cuenta la topología del sistema, por lo que no cumplen su función tan bien como podrían.

En [38] se reconoce el problema anterior y se inicia el trabajo necesario para hacer que el planificador de los Cray XE6 (como Titan), tenga en cuenta la topología del sistema y cual es la relación entre los nodos a la hora de asignarlos a un trabajo. Pero aunque eso puede mejorar las comunicaciones entre los distintos nodos asignados a un trabajo, en el mismo artículo [38] reconocen que la parte de E/S es bastante más complicada de solucionar, debido a que las operaciones de E/S pueden usar cualquiera de los nodos LNET que están repartidos por el sistema, por lo que el tráfico de E/S de otros trabajos va a ser una fuente de interferencia a las comunicaciones que realizan los trabajos entre sus nodos. Así la E/S afectará al rendimiento del sistema independientemente de lo óptimo que el planificador consiga posicionar los nodos asignados a un trabajo.

La única solución que proponen en [38] al problema que crea la E/S es que cada trabajo sólo cree ficheros en los LNETs que estén dentro del trozo que les asigne el planificador, pero esto, si se hiciera, sólo ayudaría en la creación de nuevos ficheros, ya que para acceder a los ficheros ya existentes habría que acceder por el LNET correspondiente, que podría estar en cualquier punto del torus 3D. Además esa solución tiene el inconveniente añadido de que al acceder sólo a una fracción de los nodos LNET del sistema, y por tanto sólo a una fracción de los OST, el rendimiento de la E/S se vería significativamente reducida, al no poder seleccionar el OST óptimo desde el punto de vista del sistema de ficheros, si no entre los que le toquen por la situación física que le asignó el planificador dentro del torus 3D.

En [56] analizan la degradación del rendimiento de la red de comunicación en topologías de torus 3D, usando como plataforma de pruebas un Cray XE6, y encuentran que la latencia se puede multiplicar hasta por 8 cuando hay contención en los enlaces, respecto a estar siendo usados sólo por la aplicación en cuestión. Además encuentran problemas sin una explicación clara, similares a algunos de los que hemos encontrado en el presente trabajo, como por ejemplo, una disminución en el rendimiento de las comunicaciones al usar más de dos procesos, uno por procesador, es decir, que el acceso a comunicaciones desde dos o más cores del mismo procesador hace que el rendimien-

to disminuya de unos 1100 MBytes/seg con uno o dos procesos a 700 MBytes/seg con cuatro y continúa disminuyendo conforme se van añadiendo más procesos. Además se encuentran con el problema de que la implementación UPC aumenta la cantidad de memoria requerida conforme aumenta el número de nodos a usar, por lo que no pueden realizar las pruebas con tantos nodos como quisieran.

Además mientras realizábamos las medidas cambiaron el algoritmo de asignación de los LNET [43], pero sin avisar de cuando exactamente se produjo ese cambio, por lo que las medidas dejaron de ser comparables. El motivo de no avisar a los usuarios es que se trata de una mejora interna, de muy bajo nivel, que no afecta al funcionamiento de los programas, salvo el de ofrecer una mejora en el rendimiento. Se podría comparar con un cambio en el firmware de las controladoras RAIDs, que también podría afectar a la E/S y del que no se suele dar información a los usuarios.

Es por todo lo anterior por lo que no se pueden establecer conclusiones en la E/S en los sistemas Jaguar y Titan, y en general en todos los que tengan una topología de torus, sea 3D o no, y tenga muchos nodos de cálculo. A pesar de ello creemos que es importante conocer con detalle su funcionamiento, cosa que realizamos antes de llegar a las conclusiones anteriores, y que no está suficientemente bien descrito en la literatura disponible, por lo que pasamos a describirlos a continuación.

A.1.1. Jaguar

Jaguar fue el nombre que compartieron varios superordenadores del ORNL, comenzando por un Cray XT3, y que fue sustituido por Titan entre 2012 y 2013. El Jaguar usado en este trabajo fue el XT5, constituido por 18688 nodos de cálculo, más los nodos de servicio y login, estando compuesto cada uno de los nodos de dos AMD Opteron 2435 (Istambul), con 6 cores cada uno, y 16 Gbytes de RAM, teniendo por tanto un total de 224.256 cores y casi 300 Terabytes de RAM. [6]

La red de interconexión es la estándar del XT5, una topología torus en 3D compuesta por chips SeaStar II+, que tienen 7 conexiones cada uno, como se puede ver en la figura 1.10. Una de las conexiones va a las CPUs del nodo y el resto forma el torus tridimensional, uniendo el nodo a los seis nodos vecinos más cercanos, X+, X-, Y+, Y-, Z+ y Z-, optimizando las comunicaciones entre nodos cercanos, aunque haciendo que las posibilidades de que haya una contención en las comunicaciones crezca conforme más lejanos sean los nodos que se comunican [64].

Para los cálculos se usa el sistema de ficheros Spider, ya explicado en la sección 1.4.2, que está compartido por todos los nodos, y que es un almacenamiento temporal (*scratch*), en el que los ficheros se borran automáticamente cuando han pasado entre 14 y 90 días

Figura A.1: Superordenador Jaguar. Courtesy of Oak Ridge National Laboratory, U.S. Dept. of Energy.

sin accederse, dependiendo del directorio exacto que se use.

A.1.2. Titan

Figura A.2: Superordenador Titan. Courtesy of Oak Ridge National Laboratory, U.S. Dept. of Energy.

Titan [7] es el ordenador más potente que tiene el OLCF, y uno de los más rápidos

del planeta [88], es un Cray XK6 con 18688 nodos que incluyen 299008 cores AMD Opteron 6274 (Interlagos), 688 Terabytes de RAM y 14592 gpus Kepler, y una nueva red de interconexión, Gemini [3], que al igual que el XT5 tiene una topología de torus 3D, pero con dimensiones 25x16x24 en vez de 25x32x24 que tenía Jaguar. Las dimensiones del torus son 25 (el número de racks en una fila), por 16 (el número de columnas por 2) por 24 (el número de placas por rack). Los ejes X y Z tienen el doble de velocidad que el Y, ya que cada ASIC Gemini 12 enlaces, 10 enlaces hace el exterior a 9.6 Gigabytes por segundo cada uno y un enlace hypertransport a 25.6 Gigabytes por segundo hacia cada nodo (dos enlaces HT en total). Los 10 enlaces hacia el exterior se reparten entre 2 en el eje X sentido +, 2 eje X sentido -, 1 Y+, 1 Y-, 2 Z+ y 2 Z-, como se puede ver en la figura A.3. Al tener más enlaces en unos sentidos que en otros las comunicaciones no son simétricas.

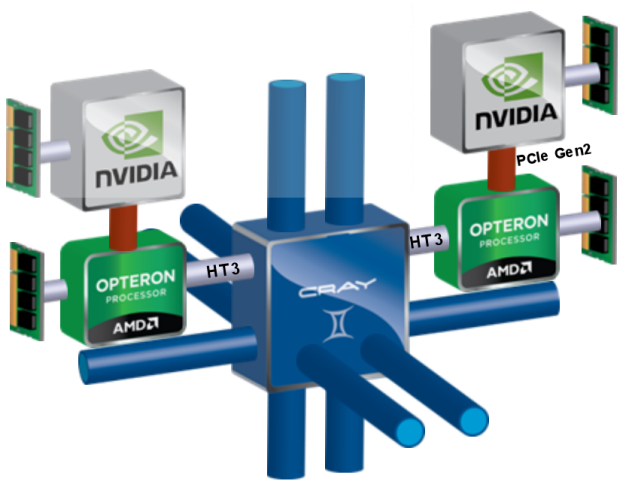


Figura A.3: Arquitectura del chip Gemini.

La ventaja es que el número de cables que se usa en Gemini es el mismo que se usaba en Seastar, y del mismo tipo, por lo que se aprovecharon al actualizar el sistema. En la figura A.4 tenemos una comparativa entre las redes SeaStar y Gemini donde se puede apreciar que el número de enlaces es el mismo en ambos casos, al compartir cada chip Gemini dos nodos de cálculo, mientras con Seastar cada nodo tenía su propio chip.

Los nodos de Titan se conectan entre ellos a través de la red Gemini, pero el sistema de ficheros está conectado a la red SION. Para resolver el problema de la comunicación entre distintas redes hay que usar algunos de los nodos como routers entre esas dos redes, por tanto los nodos que actúen como routers deberán tener acceso a ambas redes.

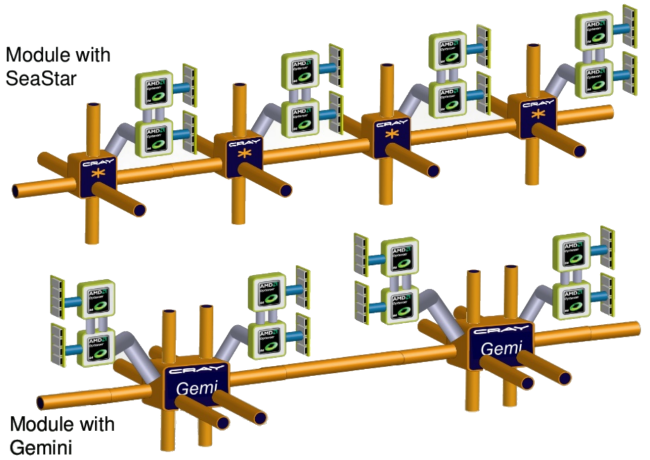


Figura A.4: Comparación entre las redes Seastar2 y Gemini.

La intención es intentar minimizar el movimiento de datos por las redes, minimizando el número de saltos que tengan que dar los datos tanto por los switches infiniband, como sobre todo por la red Gemini. Es por ello que el emplazamiento de los routers LNET es fundamental para minimizar el movimiento de los datos, para ello hay que posicionar con cuidado los 440 nodos que actuaran como routers entre las redes, de los que 432 se encargarán de los datos y 8 de los metadatos. Los routers LNET se organizan en grupos de cuatro nodos contiguos, de forma que cuatro nodos enrutadores de LNET forman un módulo de enrutamiento, con lo que tenemos en total 108 módulos de enrutamiento, que a su vez se dividen en 9 grupos de 12 módulos cada uno, de forma que cada grupo de 12 módulos corresponde a un switch infiniband que están en cada una de las columnas de Spider II. A su vez cada uno de esos 9 grupos se divide en 4 subgrupos, de 3 módulos cada uno, que dan servicio a dos filas de Titan.

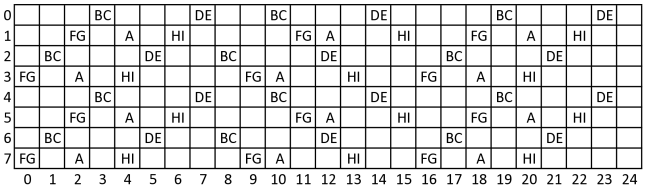


Figura A.5: Distribución de los nodos LNET en Titan [43]

En la figura A.5 se puede ver cómo están distribuidos los nodos LNET en Titan, cada

cuadro es un rack y las letras indican a los switches infiniband que conectan los LNET que están físicamente en esos racks. En Spider II hay 36 switches, 9 en cada una de las cuatro filas que ocupa Spider II, cada una de las letras indica a cual de los 9 switches se conecta ese nodo LNET, y hay cuatro nodos que pertenecen al grupo indicado por la letra. Cada uno de los nodos LNET se conecta al switch indicado en cada una de las cuatro filas de Spider II. Visto desde otro punto de vista, cada uno de los switches tiene 12 puertos usados por Titan, hay 36 switches en Spider II, que dan conectividad al total de 432 nodos LNET que hay en Titan.

Existen problemas añadidos, como por ejemplo que cada ASIC Gemini es compartido por dos nodos, por tanto se creaba contención al principio al estar los OSS de un mismo sistema de ficheros compartiendo el enlace Gemini [43]. La solución fue ir mezclando los LNET de distintos sistemas de ficheros, de esta forma los dos nodos que comparten el enlace Gemini acceden a distintos sistemas de ficheros (en distintas columnas de Spider II), y por tanto se mezclan mejor los accesos, están más barajados.

Además hay otro problema que consiste en la reducción geométrica de los anchos de banda de acceso a un nodo LNET, el motivo es que conforme un nodo está más lejano, al tener que pasar los paquetes por varias conexiones Gemini, el ancho de banda se va viendo disminuido conforme se aleja del nodo al tener que compartir los enlaces con más nodos.

Al final la distribución mostrada en la figura A.5 se consiguió a partir de distintas simulaciones. Además existen distintos algoritmos de enrutamiento LNET, siendo el planteamiento el intentar minimizar la contención de la red Gemini y SION. Los nodos de servicio y de cálculo de Titan se dividen en 12 regiones basándose en su localización topológica. Cuando se arranca un cliente, este usa un algoritmo [43] para descubrir un camino a cada uno de los grupos, del A al I, y se le asigna un router primario y dos secundarios para cada uno de los grupos.

Pero a la hora de asignar nodos a un trabajo por parte del sistema de colas no se tiene en cuenta a que distancia están los enrutadores LNET, ya que no se conoce qué nodos van a realizar E/S, ni los OSSs con los que tendrán que comunicarse.

A.1.3. Summit

Se está desarrollando el que será el sustituto de Titan, Summit, que según la planificación deberá estar disponible en 2018, y ofrecer unas 5 veces más potencia de cálculo que Titan. Para ello se cambiará tanto la arquitectura de proceso como la de almacenamiento. Se pasará de tener 18688 nodos a tener unos 3400, de la familia x86 a Power9 , y de 32 GB por nodo a más de 512 GB de memoria, mezclando memorias con tecnología

DDR4 y HBM (High Bandwidth Memory) [79].

Para el procesamiento la idea es seguir usando técnicas de computación heterogéneas [92], con GPUs más potentes, y aumentar la velocidad de acceso de las GPUs a la memoria central, usando NVLinks, así como tener un acceso unificado a la memoria, de esta forma el programador no se tendrá que preocupar de si los datos están en la memoria de la CPU o de la GPU. Además tendrá una red de interconexión más rápida entre los nodos, probablemente una IB Dual Rail EDR de unos 23 gigabytes por segundo.

Pero el avance más importante se producirá en la arquitectura del sistema de almacenamiento, donde se ha visto que los problemas vienen sobre todo de que las escrituras se producen a ráfagas, que saturan el sistema cuando se producen. Para evitar esto Summit va a implementar un sistema de buffer de ráfagas (*burst buffer*) [134] que consistirá en una capa intermedia de almacenamiento posicionada entre la aplicación y el sistema de ficheros paralelo, que absorberá las ráfagas a una velocidad cientos de veces superior a la del sistema de ficheros, de esta forma las aplicaciones podrán continuar con sus cálculos en menos tiempo, sin necesidad de esperar a que sus datos se vuelquen en el sistema de ficheros. Posteriormente se irán volcando esas ráfagas en segundo plano en el sistema de ficheros, sin saturarlo, con lo que podrá seguir cursando peticiones. Ese buffer de ráfagas estará compuesto por memorias NVRAM en los nodos que actuarán de enrutadores del sistema de ficheros [133].

Esto provocará un cambio substancial en la forma de optimizar la E/S, sobre todo facilitará la programación, haciendo menos preciso el optimizar los códigos de E/S.

Aunque Summit no se ha usado en el presente trabajo, al no estar disponible, lo enumeramos debido a la importancia que su arquitectura tendrá en la E/S en el futuro, ya que, aunque ha sido el primer sistema en ser anunciado con una arquitectura que incluye NVRAM y *burst buffers*, ya existen anuncios de sistemas de almacenamiento para HPC que incluyen esos elementos, y por tanto es de esperar que cada día sean usados en más sistemas, lo que es especialmente relevante desde el punto de vista del desarrollo de una librería de E/S paralela y distribuida, y en las conclusiones hacemos referencia a este tipo de arquitectura, a tener en cuenta para futuros trabajos.

Bibliografía

[1] Geoffray Adde, Belinda Chan, Dirk Duellmann, Xavier Espinal, Alessandro Fiorot, Jan Iven, Lukasz Janyst, Massimo Lamanna, Luca Mascetti, Joaquim M Pereira Rocha, et al. Latest evolution of eos filesystem. In *Journal of Physics: Conference Series*, volume 608, page 012009. IOP Publishing, 2015. (Cited on page 13)

[2] Geoffray Adde, Belinda Chan, Dirk Duellmann, Xavier Espinal, Alessandro Fiorot, Jan Iven, Lukasz Janyst, Massimo Lamanna, Luca Mascetti, Joaquim M Pereira Rocha, Andreas J Peters, and Elvin A Sindrilaru. Latest evolution of eos filesystem. *Journal of Physics: Conference Series*, 608(1):012009, 2015. (Cited on pages 13 and 14)

[3] Robert Alverson, Duncan Roweth, and Larry Kaplan. The gemini system interconnect. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 83–87. IEEE, 2010. (Cited on pages 50, 129 and 133)

[4] A. Bacchini, M. Rovatti, G. Furano, and M. Ottavi. Characterization of data retention faults in dram devices. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014 IEEE International Symposium on*, pages 9–14, Oct 2014. (Cited on page 5)

[5] Abhinav Bhatelé and Laxmikant V Kalé. Quantifying network contention on large parallel machines. *Parallel Processing Letters*, 19(04):553–572, 2009. (Cited on page 130)

[6] Arthur S Bland, Ricky A Kendall, Douglas B Kothe, James H Rogers, and Galen M Shipman. Jaguar: The world's most powerful computer. *Memory (TB)*, 300(62):362, 2009. (Cited on page 131)

[7] Arthur S Bland, Jack C Wells, Otis E Messer, Oscar R Hernandez, and James H Rogers. Titan: Early experience with the cray xk6 at oak ridge national laboratory. *Cray User Group*, 2012. (Cited on page 132)

[8] Dan Bonachea. Proposal for extending the upc memory copy library functions and supporting extensions to gasnet, version 2.0. *Lawrence Berkeley National Laboratory*, 2007. (Cited on page 53)

[9] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99, 2004. (Cited on pages 51 and 52)

[10] Dan Bonachea and Jaein Jeong. Gasnet: A portable high-performance communication layer for global address-space languages. *CS258 Parallel Computer Architecture Project, Spring*, 2002. (Cited on pages 51 and 52)

[11] Julian Borrill, Leonid Oliker, John Shalf, Hongzhang Shan, and Andrew Uselton. HPC global file system performance analysis using a scientific-application derived benchmark. *Parallel Comput.*, 35(6):358–373, 2009. (Cited on page 85)

[12] P.J. Braam. The Lustre storage architecture. https://wiki.hpdd.intel.com/display/PUB/HPDD+Wiki+Front+Page. (Cited on pages 7, 14 and 17)

[13] Engineering Department Cambridge University. Parallel programming, 2015. http://www-h.eng.cam.ac.uk/help/tpl/languages/parallelprogramming.html. (Cited on pages XI, XI, 24 and 26)

[14] Liqiang Cao, Hongbing Luo, and Baoyin Zhang. Jetter: a multi-pattern parallel I/O benchmark. In *CLUSTER*, pages 459–463, 2008. (Cited on page 81)

[15] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *Intl. Journal of High Performance Computing Applications*, 3(21):291–312, August 2007. (Cited on pages 23 and 24)

[16] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, David Iten, and Vassily Litvinov. Authoring user-defined domain maps in chapel. In *CUG conference*, June 2011. (Cited on pages 43 and 44)

[17] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. User-defined distributions and layouts in Chapel: philosophy and framework. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association. (Cited on page 44)

[18] Philip Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform clustered computing. In *OOPSLA*, 2005. (Cited on pages 23 and 25)

[19] Bastien Chevreux. Call for help: bughunting, 2011. (Cited on page 114)

[20] Manuel Gonzalo Claros, Rocío Bautista, Darío Guerrero-Fernández, Hicham Benzerki, Pedro Seoane, and Noé Fernández-Pozo. Why assembling plant genome sequences is so challenging. *Biology*, 1:439–459, 2012. (Cited on page 107)

[21] Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic Acids Research*, 38(6):1767–1771, 2010. (Cited on pages 99 and 102)

[22] The UPC Consortium. UPC language specifications v1.2, 2005. (Cited on page 25)

[23] The UPC Consortium. UPC language specifications v1.3, 2013. (Cited on page 25)

[24] UPC Consortium. UPC language specifications, v1.2. Technical report, Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005. (Cited on page 23)

[25] Michael Cornwell. Anatomy of a solid-state drive. *Commun. ACM*, 55(12):59–63, 2012. (Cited on page 4)

[26] Rodrigo Cánovas, Alistair Moffat, and Andrew Turpin. Lossy compression of quality scores in genomic data. *Bioinformatics*, 30(15):2130–2136, 2014. (Cited on page 107)

[27] Sebastian Deorowicz and Szymon Grabowski. Compression of dna sequence reads in fastq format. *Bioinformatics*, 27(6):860–2, Mar 2011. (Cited on page 109)

[28] Phillip Dickens and Jeremy Logan. Towards a high performance implementation of MPI-IO on the Lustre file system. (Cited on page 21)

[29] Phillip M. Dickens and Jeremy Logan. A high performance implementation of MPI-IO for a Lustre file system environment. *Conc. and Computation: Practice and Experience*, 22(11):1433–1449, 2010. (Cited on pages 15, 16, 18, 21, 27, 87, 92, 93 and 94)

[30] D.A. Dillow, G.M. Shipman, S. Oral, Zhe Zhang, and Youngjae Kim. Enhancing i/o throughput via efficient routing and placement for large-scale parallel file systems. In *Performance Computing and Communications Conference (IPCCC), 2011 IEEE 30th International*, pages 1–9, Nov 2011. (Cited on pages 21, 40 and 84)

[31] David A Dillow, Galen M Shipman, Sarp Oral, and Zhe Zhang. I/o congestion avoidance via routing and object placement. In *Proceedings of Cray User Group Conference (CUG 2011)*, 2011. (Cited on pages 20, 21 and 35)

[32] Jack Dongarra, Robert Graybill, William Harrod, Robert Lucas, Ewing Lusk, Piotr Luszczek, Janice McMahon, Allan Snavely, Jeffrey Vetter, Katherine Yelick, et al. Darpa's hpcs program: History, models, tools, languages. *Advances in Computers*, 72:1–100, 2008. (Cited on page 1)

[33] Jack J Dongarra, Hans W Meuer, and Erich Strohmaier. Top500 supercomputer sites, 1994. (Cited on page 64)

[34] Alvise Dorigo, Peter Elmer, Fabrizio Furano, and Andrew Hanushevsky. Xrootd-a highly scalable architecture for data access. *WSEAS Transactions on Computers*, 1(4.3), 2005. (Cited on page 14)

[35] Cort Dougan and Matt Sherer. RTLinux POSIX API for IO on real-time FIFOs and shared memory. *Finite State Machine Labs*, 2003. (Cited on page 116)

[36] Wikipedia editors. Message passing interface 5.4 i/o, 2015. (Cited on page 27)

[37] Tarek El-Ghazawi, François Cantonnet, Proshanta Saha, Yiyi Yao, Rajeev Thakur, Rob Ross, and Dan Bonachea. UPC-IO: A Parallel I/O API for UPC, 2006. (Cited on page 25)

[38] J Enos, G Bauer, R Brunner, S Islam, R Fiedler, M Steed, D Jackson, and Adaptive Computing. Topology-aware job scheduling strategies for torus networks. In *Cray User Group 2014, Lugano, Switzerland*. CUG, Jan 2014. (Cited on page 130)

[39] X Espinal, G Adde, B Chan, J Iven, G Lo Presti, M Lamanna, L Mascetti, A Pace, A Peters, S Ponce, et al. Disk storage at cern: Handling lhc data and beyond. In *Journal of Physics: Conference Series*, volume 513, page 042017. IOP Publishing, 2014. (Cited on page 13)

[40] E. Allen et al. The Fortress language specification. Technical report, Sun Microsystems, Inc., March 2008. (Cited on pages 23 and 25)

[41] Jean-François Lavignon et al. Etp4hpc strategic research agenda achieving hpc leadership in europe. Technical report, European Technology Platform for High Performance Computing, 2013. (Cited on page 1)

[42] Brent Ewing and Phil Green. Base-calling of automated sequencer traces using phred. ii. error probabilities. *Genome Research*, 8(3):186–194, 1998. (Cited on page 103)

[43] Matthew A Ezell, David Dillow, H Sarp Oral, Feiyi Wang, Devesh Tiwari, Don E Maxwell, Dustin B Leverman, and Jason J Hill. I/o router placement and fine-grained routing on titan to support spider ii. In *Cray User Group 2014, Lugano, Switzerland*, Jan 2014. (Cited on pages XIII, 20, 21, 35, 40, 84, 131, 134 and 135)

[44] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, James Reinhard, et al. Cray cascade: a scalable hpc system based on a dragonfly network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 103. IEEE Computer Society Press, 2012. (Cited on page 37)

[45] Scott Fadden. An introduction to gpfs version 3.5. *IBM Systems and Technology Group*, 2012. (Cited on page 9)

[46] M. Fahey, J. Larkin, and J. Adams. I/O performance on a massively parallel Cray XT3/XT4. In *22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS'08)*, Miami, Florida, 2008. (Cited on page 93)

[47] Patrick Farrell. Shared file performance improvements: Ldlm lock ahead, 2015. (Cited on pages 19 and 94)

[48] Norman E Fenton and Martin Neil. Software metrics: successes, failures and new directions . *Journal of Systems and Software*, 47(2–3):149–157, 1999. (Cited on page 67)

[49] P. Fenwick. *Proceedings of the 19th Australasian Computer Science Conference*, chapter Block sorting text compression, pages 1–10. University of Melbourne, 1996. (Cited on page 109)

[50] Noé Fernández-Pozo, Javier Canales, Darío Guerrero-Fernández, David P Villalobos, Sara M Díaz-Moreno, Rocío Bautista, Arantxa Flores-Monterroso, M Ángeles Guevara, Pedro Perdiguero, Carmen Collada, M Teresa Cervera, Alvaro Soto, Ricardo Ordás, Francisco R Cantón, Concepción Avila, Francisco M Cánovas, and M Gonzalo Claros. Europinedb: a high-coverage web database for maritime pine transcriptome. *BMC Genomics*, 12:366, 2011. (Cited on page 121)

[51] RE Fontana Jr, GM Decad, and SR Hetzler. Volumetric density trends (tb/in. 3) for storage components: Tape, hard disk drives, nand, and blu-ray. *Journal of Applied Physics*, 117(17):17E301, 2015. (Cited on page 4)

[52] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), September 2009. (Cited on page 22)

[53] Robert Gardner, Simone Campana, Guenter Duckeck, Johannes Elmsheuser, Andrew Hanushevsky, Friedrich G Hönig, Jan Iven, Federica Legger, Ilija Vukotic, Wei Yang, et al. Data federation strategies for atlas using xrootd. In *Journal of Physics: Conference Series*, volume 513, page 042049. IOP Publishing, 2014. (Cited on page 14)

[54] Al Geist. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT press, 1994. (Cited on page 22)

[55] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, November 2000. (Cited on page 6)

[56] Jorge González-Domínguez, María J. Martín, Guillermo L. Taboada, Roberto R. Expósito, and Juan Touriño. The servet 3.0 benchmark suite: Characterization of network performance degradation. *Computers & Electrical Engineering*, 39(8):2483 – 2493, 2013. (Cited on page 130)

[57] The Genome Reference Consortium (GRC). Human genome overview. (Cited on page 100)

[58] Dario Guerrero. Read/write fastq or fasta files in FQbin format. Technical report, SCBI, 2013. (Cited on pages 122 and 126)

[59] Darío Guerrero-Fernández, Rafael Larrosa, and M. Gonzalo Claros. FQbin a compatible and optimized format for storing and managing sequence data. In Ignacio Rojas and Francisco M. Ortuño Guzman, editors, *International Work-Conference on Bioinformatics and Biomedical Engineering, IWBBIO 2013, Granada, Spain, March 18-20, 2013. Proceedings*, pages 337–344. Copicentro Editorial, 2013. (Cited on pages 104, 122 and 126)

[60] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977. (Cited on page 67)

[61] R. W. Hockney and C. R. Jessope. *Parallel Comp.: Archit., Programming and Algorithms*. Ed. Adam Hildger, 1988. (Cited on page 61)

[62] HP. Configuring and tuning HP ProLiant servers for low-latency applications. Technical report, HP, 2015. (Cited on page 120)

[63] Leslie Johnston and Ross Harvey. Digital storage media and files. *The Preservation Management Handbook: A 21st-Century Guide for Libraries, Archives, and Museums*, page 293, 2014. (Cited on page 5)

[64] Terry R Jones and Bradley W Settlemyer. Fan-in communications on a cray gemini interconnect. In *Cray User Group 2014, Lugano, Switzerland*. CUG, Jan 2014. (Cited on pages 37 and 131)

[65] C. Kennedy and N. Chennagiri. Methods and systems for storing sequence read data, March 10 2015. US Patent 8,976,049. (Cited on pages 104, 122 and 126)

[66] C.J. Kennedy and N. Chennagiri. Methods and systems for storing sequence read data, September 30 2014. US Patent 8,847,799. (Cited on pages 122 and 126)

[67] P. Kevin, B. Ron, and W. Joshua. Cplant: Runtime system support for multiprocessor and heterogeneous compute nodes. In *IEEE Intl. Conference on Cluster Computing*. IEEE Computer Society, 2002. (Cited on page 63)

[68] T.M. Khoshgoftaar and J.C. Munson. Predicting software development errors using software complexity metrics. *Selected Areas in Communications, IEEE Journal on*, 8(2):253–261, Feb 1990. (Cited on page 67)

[69] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 77–88. IEEE Computer Society, 2008. (Cited on page 37)

[70] Kwangbaek Kim, Minhwan Kim, and Youngwoon Woo. A dna sequence alignment algorithm using quality information and a fuzzy inference method. *Progress in Natural Science*, 18(5):595 – 602, 2008. (Cited on page 103)

[71] David Knaak and Dick Oswald. Optimizing MPI-IO for applications on Cray XT systems, 2009. (Cited on pages 21, 29, 70 and 92)

[72] Yuichi Kodama, Martin Shumway, and Rasko Leinonen. The sequence read archive: explosive growth of sequencing data. *Nucleic Acids Research*, 40(D1):D54–D56, 2012. (Cited on page 101)

[73] Q. Koziol. *High Performance Parallel I/O*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2014. (Cited on page 17)

[74] Mark H Kryder and Chang Soo Kim. After hard drives—what comes next? *Magnetics, IEEE Transactions on*, 45(10):3406–3413, 2009. (Cited on page 4)

[75] Shanika Kuruppu, Bryan Beresford-Smith, Thomas Conway, and Justin Zobel. Iterative dictionary construction for compression of large dna data sets. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 9(1):137–149, January 2012. (Cited on page 103)

[76] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM. (Cited on pages 81, 85 and 93)

[77] J. Laros, L. Ward, R. Klundt, S. Kelly, J. Tomkins, and B. Kellogg. Red storm IO performance analysis. In *Cluster'07*, Austin, Texas, 2007. (Cited on page 93)

[78] Rafael Larrosa, Rafael Asenjo, Angeles G. Navarro, and Bradford L. Chamberlain. A first implementation of parallel io in chapel for block data distribution. In Koen De Bosschere, Erik H. D'Hollander, Gerhard R. Joubert, David A. Padua, Frans J. Peters, and Mark Sawyer, editors, *PARCO*, volume 22 of *Advances in Parallel Computing*, pages 447–454. IOS Press, 2011. (Cited on pages 65 and 129)

[79] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, et al. 25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 432–433. IEEE, 2014. (Cited on page 136)

[80] Junehawk Lee, Hyojin Kang, Seokjong Yu, Chul Kim, and Sang-Jun Yea. Whole cancer genome analysis using an i/o aware job scheduler on high performance computing resource. In *Bioinformatics and Biomedicine (BIBM), 2014 IEEE International Conference on*, pages 10–11, Nov 2014. (Cited on page 102)

[81] W.-K. Liao, A. Ching, K. Coloma, Alok Choudhary, and L. Ward. An implementation and evaluation of client-side file caching for MPI-IO. In *IPDPS 2007*, pages 1 –10, march 2007. (Cited on page 18)

[82] DJ Lipman and WR Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985. (Cited on page 98)

[83] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012. (Cited on pages 5 and 126)

[84] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. Nvm duet: Unified working memory and persistent store architecture. *SIGARCH Comput. Archit. News*, 42(1):455–470, February 2014. (Cited on page 5)

[85] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308 – 320, dec. 1976. (Cited on page 67)

[86] Robert McLay, Doug James, Si Liu, John Cazes, and William Barth. A user-friendly approach for tuning parallel file operations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 229–236. IEEE Press, 2014. (Cited on page 94)

[87] Colin McMurtrie, Nicola Bianchi, and Sadaf Alam. Cray xc30 installation – a system level overview. *Cray User Group*, 2013. (Cited on pages 39 and 40)

[88] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. Top500 supercomputing sites, 2011. (Cited on pages 29, 64 and 133)

[89] D.B. Michelson and A. Henja. A high level interface to the HDF5 file format. Technical report, Swedish Meteorological and Hydrological Institute (SMHI), 2002. (Cited on page 63)

[90] Ethan L Miller, Randy H Katz, and YH Katz. Analyzing the i/o behavior of supercomputer applications. In *Digest of papers, 11th IEEE Symposium on Mass Storage Systems*, pages 51–9. Citeseer, 1991. (Cited on page 2)

[91] S. Mittal and J.S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2015. (Cited on page 4)

[92] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys*, 2015. (Cited on page 136)

[93] John C. Munson and Taghi M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992. (Cited on page 67)

[94] A.a Nisar, W.-K.b Liao, and A.c Choudhary. Delegation-based i/o mechanism for high performance computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):271–279, 2012. (Cited on page 2)

[95] J Norton et al. Snia cifs technical reference. march 2002. *See http://www.snia.org/tech_ activities/CIFS/CIFS-TR-1p00_FINAL.pdf*, 2002. (Cited on page 6)

[96] Idoia Ochoa, Himanshu Asnani, Dinesh Bharadia, Mainak Chowdhury, Tsachy Weissman, and Golan Yona. Qualcomp: a new lossy compressor for quality scores based on rate distortion theory. *BMC Bioinformatics*, 14(1):187, 2013. (Cited on page 103)

[97] Sarp Oral, David A Dillow, Douglas Fuller, Jason Hill, Dustin Leverman, Sudharshan S Vazhkudai, Feiyi Wang, Youngjae Kim, James Rogers, James Simmons, et al. Olcf's 1 tb/s, next-generation lustre file system. In *Proceedings of Cray User Group Conference (CUG 2013)*, 2013. (Cited on pages 22, 34 and 35)

[98] Sarp Oral, David A. Dillow, Douglas Fuller, Jason Hill, Dustin Leverman, Sudharshan S. Vazhkudai, Feiyi Wang, Youngjae Kim, James Rogers, James Simmons, and Ross Miller. Olcf's 1 tb/s, next-generation lustre file system. Technical report, Oak Ridge Leadership Computing Facility, Oak Ridge National Laboratory, 2013. (Cited on pages 22, 33, 83 and 86)

[99] Sarp Oral, James Simmons, Jason Hill, Dustin Leverman, Feiyi Wang, Matt Ezell, Ross Miller, Douglas Fuller, Raghul Gunasekaran, Youngjae Kim, et al. Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 217–228. IEEE Press, 2014. (Cited on pages 22 and 34)

[100] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 58–58. IEEE, 2001. (Cited on page 9)

[101] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Alice E. Koniges, and Alison White. Towards a high-performance implementation of MPI-IO on top of GPFS. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1253–1262, London, UK, 2000. Springer-Verlag. (Cited on pages 28 and 92)

[102] Dandi Qiao, Wai-Ki Yip, and Christoph Lange. Handling the data management needs of high-throughput sequencing data: Speedgene, a compression algorithm for the efficient storage of genetic data. *BMC Bioinformatics*, 13(1):100, 2012. (Cited on page 110)

[103] Benjamin Ransford and Brandon Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 5:1–5:3, New York, NY, USA, 2014. ACM. (Cited on page 5)

[104] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985. (Cited on page 6)

[105] Alberto Sanz, Rafael Asenjo, Juan Lopez, Rafael Larrosa, Angeles Navarro, Vassily Litvinov, Sung-Eun Choi, and Bradford L Chamberlain. Global data reallocation via communication aggregation in chapel. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 235–242. IEEE, 2012. (Cited on pages 49, 55 and 62)

[106] Andrea Sboner, Xinmeng Jasmine Mu, Dov Greenbaum, Raymond K Auerbach, Mark B Gerstein, et al. The real cost of sequencing: higher than you think! *Genome biology*, 12(8):125, 2011. (Cited on page 102)

[107] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244, 2002. (Cited on page 9)

[108] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. (Cited on pages 81 and 85)

[109] T. Sherman. Cray® sonexion® scale-out lustre system for big data and hpc. *Scientific Computing*, 30(4):10–13, 2013. (Cited on pages 7, 14 and 22)

[110] G Shipman, D Dillow, Sarp Oral, and Feiyi Wang. The spider center wide file system: From concept to reality. In *Proceedings, Cray User Group (CUG) Conference, Atlanta, GA*, 2009. (Cited on pages XI, 21, 29, 31, 32, 34 and 35)

[111] Galen Shipman, David Dillow, Sarp Oral, Feiyi Wang, Douglas Fuller, Jason Hill, and Zhe Zhang. Lessons learned in deploying the world's largest scale lustre file system. In *The 52nd Cray User Group Conference*. Citeseer, 2010. (Cited on pages XI, 21, 22, 34 and 36)

[112] Galen M Shipman, David A Dillow, Douglas Fuller, Raghul Gunasekaran, Jason Hill, Youngjae Kim, Sarp Oral, Doug Reitz, James Simmons, and Feiyi Wang. A next-generation parallel file system environment for the olcf. In *Proceedings of the Cray User Group Conference, Stuttgart, Germany*, 2012. (Cited on pages XI, XI, 22, 33, 34 and 37)

[113] William E Snaman and David W Thiel. The vax/vms distributed lock manager. *Digital Technical Journal*, 5(2):44, 1987. (Cited on page 17)

[114] Ayumi Soga, Chao Sun, and Ken Takeuchi. Nand flash aware data management system for high-speed ssds by garbage collection overhead suppression. In *Memory Workshop (IMW), 2014 IEEE 6th International*, pages 1–4. IEEE, 2014. (Cited on page 4)

[115] Lincoln Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010. (Cited on page 101)

[116] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005. (Cited on page XVII)

[117] Jon Tate, Fabiano Lucchese, Richard Moore, and SAN TotalStorage. *Introduction to storage area networks*. IBM Corporation, International Technical Support Organization, 2005. (Cited on page 6)

[118] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *In Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1998. (Cited on page 92)

[119] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, 1999. (Cited on page 92)

[120] Rajeev Thakur and Ewing Lusk. An abstract-device interface for implementing portable Parallel-I/O interfaces. In *Symp. on the Frontiers of Massively Parallel Computation*, pages 180–187, 1996. (Cited on page 63)

[121] Rajeev Thakur and Rajeev Thakur Ewing Lusk. Users guide for ROMIO: A high-performance, portable MPI-IO implementation, 2002. (Cited on page 92)

[122] Vijay Velusamy, Changzheng Rao, Srigurunath Chakravarthi, Jothi P Neelamegam, Weiyi Chen, Sanjay Verma, and Anthony Skjellum. Programming the infiniband network architecture for high performance message passing systems. In *ISCA PDCS*, pages 391–398. Citeseer, 2003. (Cited on page 50)

[123] Gregory Vey. Differential direct coding: a compression algorithm for nucleotide sequence data. *Database*, 2009, 2009. (Cited on page 103)

[124] Murali Vilayannur, Samuel Lang, Robert Ross, Ruth Klundt, Lee Ward, et al. Extending the POSIX I/O interface: A parallel file system perspective. *Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-302*, 2008. (Cited on page 64)

[125] VMware. RDMA performance in virtual machines using QDR InfiniBand on VMware vSphere 5. Technical report, VMware, 2011. (Cited on page 120)

[126] VMware. Best practices for performance tuning of latency - sensitive workloads in vsphere vms. Technical report, VMware, 2013. (Cited on page 120)

[127] Stephen R. Walli. The POSIX family of standards. *StandardView*, 3(1):11–17, March 1995. (Cited on pages 7, 25 and 64)

[128] Richard Walsh. Parallel programming with Unified Parallel C (UPC), 2011. (Cited on page 25)

[129] Raymond Wan, Vo Ngoc Anh, and Kiyoshi Asai. Transformations for the compression of fastq quality scores of next-generation sequencing data. *Bioinformatics*, 28(5):628–35, Mar 2012. (Cited on pages 107 and 110)

[130] Shenggang Wan, Qiang Cao, and Changsheng Xie. Optical storage: an emerging option in long-term digital preservation. *Frontiers of Optoelectronics*, 7(4):486–492, 2014. (Cited on page 8)

[131] Sebastian Wandelt, Marc Bux, and Ulf Leser. Trends in genome compression. *Current Bioinformatics*, 9(3):315 – 326, 2014. (Cited on pages 102 and 103)

[132] Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. Understanding lustre filesystem internals. *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep*, 2009. (Cited on page 17)

[133] Teng Wang, Sarp Oral, Michael Pritchard, Kevin Vasko, and Weikuan Yu. Development of a burst buffer system for data-intensive applications. *CoRR*, abs/1505.01765, 2015. (Cited on page 136)

[134] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. Burstmem: A high-performance burst buffer system for scientific applications. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 71–79. IEEE, 2014. (Cited on page 136)

[135] SA Weil, SA Brandt, EL Miller, DDE Long, and C Maltzahn. Design and implementation of ceph: A scalable distributed file system. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2006. (Cited on page 13)

[136] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. (Cited on pages 12 and 13)

[137] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM, 2006. (Cited on page 12)

[138] T.A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984. (Cited on page 103)

[139] KA Wetterstrand. Dna sequencing costs: Data from the nhgri genome sequencing program (gsp). http://www.genome.gov/sequencingcosts/. (Cited on pages XII and 101)

[140] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007. (Cited on pages 23 and 52)

[141] Weikuan Yu, J. Vetter, R.S. Canon, and S. Jiang. Exploiting lustre file joining for effective collective io. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 267–274, May 2007. (Cited on pages 20, 21 and 87)

[142] Weikuan Yu, J.S. Vetter, and H.S Oral. Performance characterization and optimization of parallel I/O on the Cray XT, 2008. (Cited on pages 92 and 93)

[143] Tiezhu Zhao, V. March, Shoubin Dong, and S. See. Evaluation of a performance model of lustre file system. In *ChinaGrid Conference (ChinaGrid), 2010 Fifth Annual*, pages 191–196, July 2010. (Cited on pages 20, 22 and 27)