Capítulo 2

Arquitecturas y programación paralela

Gracias al claro incremento del coste computacional de las aplicaciones actuales, existe una gran demanda de arquitecturas cada día más rápidas y con mejores prestaciones. En esta evolución de los computadores, las arquitecturas paralelas son un paso inevitable para la consecución de mayores velocidades de procesamiento.

Como consecuencia, los modelos y técnicas de programación también han evolucionado para cubrir el desarrollo de algoritmos para estas nuevas plataformas paralelas. Sin embargo, es un hecho conocido, que la evolución de las herramientas de desarrollo de algoritmos paralelos sigue un menor ritmo que el de estas arquitecturas. Esta situación impide en gran medida el uso generalizado de las máquinas paralelas existentes. Por ello, actualmente se están incrementando los esfuerzos para idear y mejorar distintos paradigmas de programación paralela.

Así pues, en este capítulo dedicamos dos secciones a introducir las nociones básicas implicadas en los dos parrafos anteriores: en la sección 2.1 daremos un repaso rápido al soporte *hardware* paralelo disponible actualmente, centrándonos en la plataforma particular que hemos utilizado para validar nuestras soluciones: el Cray T3D/T3E; seguidamente, el punto de vista *software*, es decir el de las utilidades de ayuda a la programación disponibles para las plataformas paralelas, será tratado en la sección 2.2.

2.1 Las arquitecturas paralelas

Por motivos tecnológicos, el número de transistores utilizados en un procesador está creciendo más rápidamente (40% por año) que la frecuencia de reloj (10% por año). Ese mayor número de transistores ha redundado en avances de tipo arquitectural, que han dominado, frente al aumento de la frecuencia, en el incremento global de prestaciones del procesador. Estos avances han conducido a procesadores con **paralelismo a nivel de instrucción** (ILP) que reducen (en media) el número de ciclos de reloj necesarios para ejecutar una instrucción o que incluso ejecutan varias instrucciones en paralelo.

Esto ocurre por ejemplo en las arquitecturas **superescalares** o en las más complejas **VLIW** (Very Long Instruction Word) donde existen gran cantidad de unidades funcionales independientes. Es tarea del compilador el reestructurar los programas de forma que se maximice el número de unidades funcionales que operan en paralelo.

Sin embargo, el crecimiento de las prestaciones de un procesador debido al paralelismo a nivel de instrucción también es limitado (por ejemplo, actualmente no es rentable el coste asociado a una arquitectura que lanza a ejecutar ocho instrucciones o más). Así que, si la tendencia es seguir aumentando el número de transistores, tendremos que cambiar la estrategia ILP por otra que permita un paralelismo a un nivel mayor (o más grueso). Esto justifica la relevancia de los **multiprocesadores**, o supercomputadores con un conjunto de procesadores que cooperan para la rápida resolución de grandes problemas computacionales [89]. Cuando el número de procesadores es elevado utilizaremos la denominación de sistemas MPP *Massively Parallel Processor* o procesadores masivamente paralelos.

En el siguiente apartado introducimos algunas nociones, parámetros característicos y clasificación de los multiprocesadores actuales. Nos centraremos más tarde en la plataforma paralela sobre la que hemos montado y chequeado nuestros algoritmos.

2.1.1 Generalidades y clasificación

La evolución de las arquitecturas paralelas ha derivado en el desarrollo de distintos modelos, como los arrays sistólicos, máquinas de flujo de datos, arquitecturas de memoria compartida, computadores de pase de mensajes, etc. Actualmente se está produciendo un movimiento de convergencia de estas arquitecturas que, en base a la experiencia pasada, extrae y auna las características beneficiosas de cada estrategia descartando las demás.

Por lo pronto, las tendencias actuales han extendido la noción clásica de arquitectura de computadores de forma que se contemplen los conceptos de **comunicación** y **cooperación**. Es decir, en el diseño de un sistema multiprocesador, además de la arquitectura del procesador o nodo del sistema, está implicada una arquitectura de comunicaciones. Esta última se organiza en una capa de más alto nivel, de primitivas de comunicación (HW/SW) visibles por el programador, y en la capa de bajo nivel que contempla la implementación eficiente de esas primitivas ya sea mediante memoria compartida o distribuida.

Los parámetros de diseño importantes de esta arquitectura de comunicaciones son:

- **Latencia**: Tiempo necesario para iniciar una comunicación.
- **Ancho de Banda**: Número de datos que se pueden comunicar por segundo.
- **Sincronización**: Modo en que se sincronizan los productores y consumidores de datos.
- **Granularidad de los Nodos**: Criterios para asignar silicio (o número de transistores) a unidades de procesamiento o a memoria.

- **Aplicabilidad**: Propósito general o especial.
- **Denominación**: Define como referenciamos a los datos compartidos entre los procesos, y la semántica de las primitivas que soportan los lenguajes de programación. Existen dos alternativas básicas:
  - Modelo de **Memoria centralizada o global** donde todos los procesadores pueden acceder de forma directa a todas las posiciones de memoria del sistema. Un dato escrito en memoria es accesible por cualquier procesador de forma que los nodos pueden intercambiar información (cooperación). Es decir la comunicación se integra a nivel de memoria.
  - Modelo de **Memoria privada** o **Pase de mensajes** donde los procesadores tienen acceso a su espacio de direcciones local y las operaciones de cooperación (comunicación y sincronización) se realizan mediante pase de mensajes. Es decir, éstas operaciones se integran a nivel de E/S y son necesarias llamadas explícitas al S.O. para realizarlas (rutinas *send/receive*).

Estos modelos de memoria global y pase de mensajes pueden ser considerados como la capa de alto nivel visible por el programador, pero no tiene por qué establecer restricciones HW fundamentales. De hecho, ambos representan modelos de programación que pueden ser simulados tanto en maquinas físicamente con **memoria compartida** (los procesadores acceden a cualquier módulo de memoria mediante una red de interconexión), o **memoria distribuida** (los procesadores acceden a su memoria local e intercambian mensajes con otros nodos mediante una red de interconexión).

Independientemente de si la red de interconexión está entre procesadores y módulos de memoria o entre nodos compuestos de procesador y memoria local, las necesidades de eficiencia de la red de interconexión son similares: es necesario reducir la latencia y aumentar el ancho de banda. Si no es posible reducir la latencia mediante caches, controladores de comunicaciones, etc, podemos reducir el coste asociado agrupando comunicaciones o solapando computaciones con comunicaciones. La topología de la red de interconexión afecta directamente a la latencia, pero consideraciones de coste resultan en diseños finales con topologías más baratas aunque tengan menor conectividad. Las topologías más populares actualmente en orden de mayor a menor conectividad son: hipercubo, toro y malla; bidi- o tridimensional estas dos últimas [89].

Ejemplos de máquinas comerciales recientes con memoria compartida son el SGI Power Challenge y los Y-MP y C90 de Cray. En el otro bando, máquinas con memoria distribuida son: SP2 de IBM con RS6000, CM5 Connection Machine de Thinking Machines con procesadores SPARC, CS2 Computing Surface 2 de Meiko, Paragon de Intel con i860, NCube, Maspar MP-2, y otras.

Sin embargo, la convergencia arquitectural, de la que hablábamos al comienzo, está dando lugar a arquitecturas de **memoria compartida-distribuida**, donde físicamente la memoria está distribuida entre los procesadores, pero existe un soporte HW para permitir un espacio de direcciones globales e incluso gestión de la coherencia entre caches locales a cada nodo. Esta filosofía auna la sencillez de la programación en un modelo de memoria compartida junto con la escalabilidad que proporcionan los sistemas de memoria distribuida.

Las máquinas con memoria compartida-distribuida se pueden organizar en dos grandes bloques[1]:

- Máquinas **NUMA** (*NonUniform Memory Access*) o con espacio de direcciones físico estático entre las que encontramos arquitecturas **sin coherencia cache**, como el Cray T3D con procesadores DEC-Alpha 21064, **con coherencia parcial de cache**, como el Cray T3E con DEC-Alpha 21164, o **con coherencia cache** (CC-NUMA o *Cache Coherent NonUniform Memory Access*) como el Origin 2000 de SGI con procesadores R10000.

- Máquinas **COMA** (*Cache Only Memory Architecture*) o con espacio de direcciones físico dinámico, donde las memorias distribuidas son convertidas en caches. En este caso particular de máquinas NUMA no hay jerarquía de memoria en cada procesador. Esto ocurre en la KSR-1 de Kendall Square Research y en la arquitectura I-ACOMA propuesta en la universidad de Illinois [144].

Por último, otro modelo que queremos tratar es el de las **arquitecturas de paralelismo de datos**. Históricamente, estas máquinas se construyen mediante una matriz de gran cantidad de procesadores simples con una memoria local limitada. Esta matriz se conecta a un procesador de control de propósito general que distribuye las instrucciones que serán ejecutadas síncronamente. La arquitectura se denomina **SIMD** de *Single Instruction Multiple Data*. Los procesadores trabajan con sus datos locales y cooperan mediante una red de comunicaciones eficiente y con capacidad de sincronización global rápida. Ejemplos de esta arquitectura encontramos en el CM-2 de Thinking Machines y el Maspar MP-2. Su aplicabilidad reducida a cálculos científicos regulares provoca que el modelo converja (de nuevo la idea de convergencia arquitectural) o evolucione en el modelo **SPMD** (Single Program Multiple Data). Bajo el modelo SPMD, un mismo programa es cargado en los distintos nodos del multiprocesador, mientras que los datos se distribuyen entre ellos, de forma que los procesadores operarán mayoritariamente con la fracción local de datos que les ha correspondido. Sin embargo el modelo de programación asociado a las arquitecturas de paralelismo de datos, sigue vigente debido a su sencillez de programación, y será tratado en la sección 2.2.

Cuando en la arquitectura del sistema multiprocesador, no existe una sincronización a nivel de instrucciones, los procesadores tienen una capacidad de proceso elevada, capaces de ejecutar programas de forma independiente, pero están conectados mediante una red de comunicaciones, a la arquitectura se la conoce con el nombre **MIMD** (*Multiple Instruction Multiple Data*).

2.1.2 Los multiprocesadores Cray T3D y T3E

Dedicamos una subsección a los computadores masivamente paralelos Cray T3D y T3E ya que han sido utilizados para comprobar la validez de los algoritmos descritos en esta memoria.

[1]Aunque esta taxonomía aparece inicialmente para clasificar máquinas de memoria compartida.

Cray T3D

El **Cray T3D** es un sistema MPP MIMD con arquitectura NUMA que puede llegar a proporcionar 300 GFLOPS[2] de pico en su configuración máxima de 2048 procesadores [35]. El sistema completo consta de un array de procesadores conectado a un "host" donde se ejecuta el sistema operativo. El array de procesadores está compuesto de nodos de procesamiento, donde cada uno de ellos contiene dos procesadores DEC Alpha 21064 a una frecuencia de 150 MHz. Este procesador soporta operaciones enteras de 64 bits y flotantes en formato IEEE-754 de 64 bits, permitiendo unas prestaciones de pico de 150 MFlops de 64 bits. En cuanto a la jerarquía de memoria, el Alpha 21064 incluye una cache de datos de 8Kbytes y otra de instrucciones del mismo tamaño. Las caches contienen 256 líneas de 32 bytes cada una. Cada procesador tiene disponible una memoria local de 64Mbytes.

Los nodos están conectados entre sí por una topología de toro tridimensional, donde cada uno de los seis enlaces soporta velocidades de transferencia de hasta 300 Mbyte/s. Aunque, como vemos, la memoria está físicamente distribuida, existe un soporte *hardware* para proporcionar un espacio único de direcciones aunque no está garantizada la coherencia cache en el sistema multiprocesador [45]. La circuitería de soporte necesaria para extender el control y las funciones de direccionamiento del procesador permite:

- Interpretación de direcciones. En el Cray T3D los pines de direcciones del procesador no se conectan directamente al bus de direcciones de la memoria física, sino que la circuitería de soporte interpreta la dirección y encamina los datos entre el procesador y la memoria local o remota.

- Escrituras no bloqueantes[3], lecturas cacheables[4] o no, y prebúsqueda de datos de memorias no locales.

- Soporte para el paso de mensajes entre procesadores. Cuando se recibe un mensaje, la circuitería de soporte lo coloca en una cola de mensajes e interrumpe al procesador para que pueda leerlo de la cola. La cola de mensajes se sitúa en memoria local y puede almacenar hasta 256 Kbytes de información.

- Barreras de sincronización. Se soportan dos tipos de operaciones de sincronización: barreras y eurekas. Existen dos registros de barrera de 8 bits cada uno, donde cada bit está asociado a circuitos de sincronización de barrera independientes y con idéntico funcionamiento. El circuito de sincronización de barrera está formado por un árbol de puertas AND que devuelve un 1 a todos los procesadores si todos tienen el bit correspondiente a 1. Las sincronizaciones de eureka permite avisar a los demás procesadores cuando uno de ellos ha completado cierta información. El circuito de sincronización de eurekas está asociado a una operación OR global.

[2]Un GFLOPS *Giga FLoating-point OPearations per Second* es equivalente a mil millones de operaciones en punto flotante por segundo.

[3]Este tipo de escritura permite al procesador seguir procesando instrucciones mientras que la circuitería de soporte completa la operación de escritura, ya sea local o remota.

[4]Una operación de lectura cacheable copia el dato leído en cache.

En cuanto a las operaciones de entrada/salida, el array de procesadores incluye cuatro puertos de E/S: dos de ellos proporcionan conexión directa con dispositivos de E/S, mientras que los otros dos conectan el array con el "host" que puede ser un Cray Y_MP o un C90, y que proporciona posibilidades adicionales de E/S (conexión a red, almacenamiento masivo, etc...). Estos puertos de E/S permiten un ancho de banda teórico bidireccional de 400 Mbyte/s.

Cray T3E

El Cray T3E es el siguiente miembro de la familia de arquitecturas MPP de Cray Research Inc. Mantiene las características MIMD y NUMA de su predecesor, pero ahora es escalable hasta 2176 procesadores de forma que puede llegar a proporcionar un TeraFLOP de pico [46].

El T3E no necesita un "host" al que conectarse sino que se dice que es *self hosted*. La matriz de procesadores está compuesta por nodos de procesamiento, pero esta vez, cada nodo sólo tiene un procesador: el DEC Alpha 21164 a 300MHz capaz de proporcionar 600 MFLOPS de pico ya que su característica superescalar le permite procesar hasta dos operaciones en punto flotante y dos enteras por cada ciclo de reloj.

En cuanto a la jerarquía de memoria, el procesador incluye:

- Caches internas de datos e instrucciones de 8Kbytes (256 líneas) cada una y con asignación directa. La cache de datos tiene una política de actualización de memoria principal de escritura directa (*write-through*). La cache de instrucciones puede almacenar hasta 2048 instrucciones.

- Segundo nivel de Cache interna para datos e instrucciones, asociativa por conjuntos de 3 líneas por conjunto y con capacidad de 96Kbytes. Utiliza la política de escritura retardada (copy-back) y las lecturas y escrituras son cacheables.

- También dentro del chip existe un buffer de escrituras y un fichero de direcciones fallidas (Missed Address File –MAF–) que combina fallos de cache de distintas palabras de una misma línea para que sean cargadas en una única operación de lectura.

En el nodo de procesamiento se encuentra la memoria local configurable entre 64Mbytes y 2Gbytes, junto con la circuitería de soporte [134]. Esta circuitería de soporte consiste en:

- 32 circuitos de barrera y eureka para 32 contextos distintos.

- Registros E, usados para ocultar la latencia al acceder a posiciones de memoria no locales. Existen 512 registros E de usuario y 128 de sistema.

- *Hardware* auxiliar para translación de direcciones, control de errores, coherencia cache, etc.

En relación al último punto hay que matizar que la coherencia cache se mantiene sólo parcialmente. Es decir, si un procesador tiene un dato modificado en su cache, un acceso por parte de otro procesador a ese dato de la memoria principal provoca una actualización previa de dicha memoria. Sin embargo, si la misma dirección global está en caches de procesadores distintos, las modificaciones de una no se reflejan en la otra.

La interconexión de los nodos también utiliza una topología de toro tridimensional, pero el ancho de banda de la red aumenta a 480Mbytes/s. Esta red está controlada por un *hardware* específico de enrutado. En cuanto a la entrada/salida, existen controladores de E/S que permiten una conexión en anillo, llamada GigaRing, para permitir la comunicación entre cualquier procesador del T3E y un dispositivo externo.

2.2 Herramientas para el desarrollo

En la sección anterior hemos repasado varias de las arquitecturas adoptadas en máquinas paralelas. Sin embargo, ese aspecto *hardware* debe ser combinado con las utilidades *software* necesarias en función de la técnica de programación que vaya a ser utilizada. Básicamente, podemos decir que existen cuatro alternativas, de aceptación general, para construir un programa paralelo: paralelización manual usando **pase de mensajes**, paralelización semi-automática basada en el paradigma de **paralelismo de datos**, **paralelización automática** y paralelización manual en **memoria compartida**. Las cuatro han sido abordadas en la memoria, por lo que serán introducidas bajo este epígrafe.

Ya sea considerando una u otra aproximación, debemos contemplar el mismo aspecto básico: en máquinas paralelas el problema a resolver se divide entre los procesadores disponibles, con la intención de conseguir el mismo resultado que obtendríamos con uno sólo, pero en un tiempo inversamente proporcional al número de procesadores. Existen muchas formas de implementar esta descomposición del problema. Por ejemplo, en función de la granularidad, tendremos paralelismo a:

- **nivel de instrucción** (grano fino) cubierto por los procesadores superescalares, VLIW, etc, como se comentó al comienzo de la sección 2.1.
- **nivel de bucle** (grano medio) donde manteniendo la estructura del código secuencial, los bucles sin dependencias tienen repartido su espacio de iteraciones entre los procesadores disponibles. El paralelismo a este nivel domina en problemas de computación científica. Una razón importante para ello es que, si aumenta el tamaño del problema, suele ser suficiente con incrementar el número de procesadores para conseguir la solución en el mismo tiempo.
- **nivel de tarea** (grano grueso) aplicable cuando tenemos un gran número de tareas semi-independientes que pueden ser ejecutadas simultáneamente en varios procesadores. La simplicidad de este modelo, permite a sistemas operativos como Solaris, Windows NT, u OS/2 [5] explotar este tipo de paralelismo operando sobre

[5]Generalmente sistemas operativos *multithread* que manejan procesos simples que se comunican mediante memoria compartida: 'threads'.

una máquina con pocos procesadores y memoria compartida.

Pero quizás el aspecto más importante resida en elegir una descomposición del problema que provea una distribución equitativa de la carga (el trabajo) entre los procesadores de forma que éstos estén ocupados durante el mismo tiempo (**balanceo de la carga**), y por otro lado, minimice el número de comunicaciones interprocesador necesarios para la resolución global del problema (**overhead de comunicaciones**). Esta última cuestión estará condicionada en gran medida por el nivel de **localidad** que consigamos alcanzar. Explotar la localidad consiste en maximizar el número de computaciones que un procesador realiza con un conjunto de datos que tiene asignado, manteniendo en un mínimo el número de interacciones inter-procesador. Este aspecto cobra especial interés en maquinas con memoria y/o caches locales al procesador[6].

Las distintas técnicas de programación proporcionan distintos medios para controlar estos aspectos (balanceo, *overhead*, localidad, ...) con distintos costes de desarrollo. El compromiso aparece entre estos dos parámetros, ya que el paradigma de pase de mensajes proporciona un mayor control pese a requerir un gran esfuerzo por parte del programador de la aplicación paralela. Por contra, el paradigma de paralelización automática delega la tarea de generar el código eficiente en un compilador. Actualmente, esta aproximación consigue menores prestaciones, pero a cambio, presenta un coste de desarrollo mínimo. El paradigma de paralelismo de datos representa una solución de compromiso intermedia, donde tanto el programador como un compilador ponen algo de su parte para resolver el problema de la generación del código paralelo, típicamente a nivel de bucle, eficiente y en poco tiempo. La programación en memoria compartida es en cierto modo una simplificación del paradigma de paralelismo de datos. Presenta un esquema sencillo con espacio único de direcciones, pero con problemas para conseguir buenos resultados ya que por lo general el programador se suele despreocupar de explotar la localidad. Aparte aparecen otros problemas, como exclusión mutua, coherencia cache y *false sharing*, contención de memoria, etc, que discutimos en la subsección 2.2.4.

Para cuantificar las prestaciones del algoritmo paralelo se utilizan diferentes medidas. La **aceleración** representa el tiempo de ejecución del algoritmo secuencial dividido entre el tiempo de ejecución del algoritmo paralelo. Generalmente, el valor máximo de esta medida coincide con el número de procesadores implicados en la resolución paralela, aunque excepcionalmente puede tomar un valor mayor[7]. La **eficiencia** se define como el cociente entre la aceleración y el número de procesadores que han colaborado en resolver el problema en paralelo. Salvo en los casos excepcionales mencionados, su valor óptimo es uno. Se dice que un algoritmo es **escalable** si al aumentar el tamaño de problema linealmente con el número de procesadores la eficiencia se mantiene constante. Una medida de la escalabilidad es la **isoeficiencia** [89] que relaciona el tamaño del problema con el número de procesadores para mantener una eficiencia constante.

Una vez aclarados los parámetros que nos permiten cuantificar la bondad de nues-

[6] Lo cual es típico en los procesadores actuales que al menos llevan un nivel de cache en el propio chip (on-chip).

[7] Este efecto, conocido con el nombre de super-aceleración o super-linearidad, se justifica en muchas situaciones debido a que al aumentar el número de procesadores aumenta el tamaño total de la cache usada, ya que cada procesador suele disponer de una cache local.

tros algoritmos, dedicamos un apartado a cada una de las técnicas contempladas en esta memoria. Adicionalmente, dirigimos una última subsección a introducir algunas librerías matemáticas que permiten optimizar códigos numéricos en computadores con jerarquía de memoria.

2.2.1 Interfaces de pases de mensajes: PVM, MPI y SHMEM

El paradigma del pase de mensajes, donde un número de procesadores se comunican mediante el envío de mensajes entre ellos, ha sido extensivamente utilizado para programar sistemas multiprocesador. Una de las razones para ello, es que prácticamente, la totalidad de las plataformas paralelas pueden soportar el pase de mensajes. Los programas escritos bajo este modelo pueden ejecutarse en multiprocesadores, típicamente MIMD, con memoria compartida o distribuida (principalmente esta última), redes de estaciones de trabajo, o combinación de ambas.

La gran flexibilidad y el alto grado de libertad en el control del programa pueden ser considerados como un punto a favor del paradigma de pase de mensajes. Sin embargo, esa misma característica presenta otro filo: el programador tiene la responsabilidad absoluta de mantener todos los procesadores ocupados, reducir el coste asociado a las comunicaciones, y evidentemente, debe asumir el elevado tiempo de desarrollo y depuración (por ejemplo, evitando interbloqueos, solapando comunicaciones y computaciones, etc), que conlleva este paradigma.

Aunque existen algunas variaciones, los conceptos básicos sobre procesos comunicantes a través de mensajes están bien asentados. En los últimos 10 años, se han conseguido sustanciales progresos en el desarrollo de aplicaciones bajo este paradigma. Más recientemente, varios sistemas han demostrado que la programación paralela mediante pase de mensajes puede ser implementada de forma eficiente, e incluso puede que más importante, de forma portable, ya que es difícil encontrar una plataforma paralela que no soporte alguno o varios de los interfaces de pase de mensajes. Los más populares parecen ser, **PVM** (Parallel Virtual Machine) [70] y **MPI** (Message Passing Interface) [105] el cual se está convirtiendo en el estándar en este tipo de interfaces. De hecho, los avances y mejoras de MPI se han materializado en el conocido como MPI-2[106], el cual supera con creces las prestaciones y posibilidades del PVM.

Las características comunes de los dos interfaces son:

- *Software* de dominio público, diseñado, especificado e implementado por grupos independientes de las empresas de plataformas paralelas.
- Definen funciones portables de alto nivel en C o Fortran 77 para el intercambio de información entre grupos de procesadores (`send` y `recv`), realizándose las conversiones necesarias en caso de operar sobre máquinas heterogéneas.

Sin embargo existen diferencias importantes que constatan la superioridad de MPI:

- Las implementaciones de MPI suelen ser más eficiente que las de PVM (que fue inicialmente pensado para trabajar sobre *clusters* de estaciones de trabajo).

- MPI soporta el concepto de topología entre los procesos comunicantes, disminuyendo la penalización por comunicaciones cuando una topología virtual se adapta a la topología de la red de interconexión física del multiprocesador.

- El soporte de grupos, comunicadores, tipos de datos definidos por el usuario, mayor variedad de comunicaciones síncronas y asíncronas, comunicaciones globales (*gather*, *scatter*, comunicaciones todos a todos, reducciones), y manejo eficiente de las situaciones de error, etc, proporciona a MPI una versatilidad, potencia y facilidad de uso superior a la de PVM.

Estos dos interfaces para el paso de mensajes están disponibles tanto en el Cray T3D como en el T3E. En este caso, tanto las rutinas de PVM como las de MPI, están construidas mediante llamadas a procedimientos de una librería de más bajo nivel. Esta librería, con el nombre **SHMEM** [25], está diseñada sobre el principio básico de sacrificar la facilidad de uso en aras de una mayor velocidad y flexibilidad. Por tanto, se pueden conseguir mayor ancho de banda y menor latencia utilizando directamente las rutinas de esta librería, pagando esas ventajas con un tiempo adicional de desarrollo de los códigos paralelos.

En esta librería, disponible para C y Fortran 77, sólo se han implementado un número pequeño de operaciones, pero útiles para la gran mayoría de programadores de aplicaciones. Las rutinas básicas de la librería SHMEM se clasifican en:

- **Individuales**, es decir, son llamadas por un único procesador, independientemente de cuantos procesadores estén involucrados en la operación. Estas rutinas utilizan las capacidades de memoria compartida del Cray para soportar transferencia de datos por medio de un protocolo unilateral (*single-party protocol*). Las dos rutinas principales son `shmem_put` y `shmem_get` que escriben y leen, respectivamente, de memorias remotas, sin ningún tipo de intervención por parte del procesador remoto. Esto tiene las siguientes consecuencias:

  1. Es necesaria alguna sincronización vía barrera (`shmem_barrier()`) o *handshake* (`shmem_wait()`) para evitar inconsistencias (por ejemplo, en el caso del `put`, que el receptor lea el dato antes de que esté escrito, o que no "se entere" de cuando está el dato disponible).
  2. En el Cray T3D no existe el *hardware* de soporte adecuado para mantener la coherencia cache, por lo que es necesario que se ocupe de ello el programador. Para ello puede utilizar rutinas como `shmem_set_cache_inv()` y `shmem_clear_cache_inv()` (habilitan y deshabilitan la invalidación automática de las líneas de cache necesarias), o `shmem_udcflush()` (invalida todas las líneas de la cache de datos).
  3. Las direcciones de lectura y escritura pasadas como parámetro a las rutinas `put` y `get` referencian posiciones locales, y por tanto el procesador que hace la llamada debe conocer la dirección donde va a escribir el dato (put) o de donde lo va a leer (get).

- **Colectivas** o globales, cuando todos los procesadores involucrados en la operación deben llamar síncronamente a la rutina. Estas rutinas permiten realizar operaciones de reducción, barreras, radiaciones y colecciones, y se ejecutan en un conjunto específico de procesadores que puede ser seleccionado mediante los parámetros de la rutina.

Una comparación del rendimiento de estos tres interfaces para el pase de mensajes se incluye en la tabla 2.1, caracterizando cada uno de ellos usando los parámetros de Hockney y Carmona [88]:

- r_∞, la velocidad de transferencia máxima en MBytes/s (equivale al ancho de banda).
- $n_{\frac{1}{2}}$, la longitud del mensaje necesaria para alcanzar la mitad de la velocidad especificada por r_∞.

Interfaz	Protocolo	r_∞	$n_{\frac{1}{2}}$	Latencia (μs)
PVM	shmem	25.7	3357	138
MPI (EPCC)	shmem	50.4	2662	36.4
`shmem_get`	shmem	58	362	6
`shmem_put`	shmem	120.2	780	6

Tabla 2.1: Comparación entre los sistemas de pase de mensajes.

Es claro que las rutinas SHMEM consiguen un mayor ancho de banda y una menor latencia. Y en particular la rutina `shmem_put` tiene el doble del ancho de banda que `shmem_get` debido a que en esta última rutina, por cada palabra remota que se lee, es necesario mandar su dirección y luego recibir la palabra correspondiente.

Es importante matizar que algunos investigadores distingue el paradigma de programación, que a veces llaman `put-get`, del de pase de mensajes. Otros, sin embargo lo consideran un caso particular, denominándolo paradigma de `mensajes unilaterales`. Aunque no queremos entrar en discusiones acerca de la terminología correcta, si es admitido por todos que esta aproximación evita el *rendez-vous* de dos procesadores, cuando en principio es suficiente con implicar a uno de ellos. Esto, en cierto modo, simplifica el desarrollo de códigos paralelos, ya que el programador no tiene por qué aparear cada `send` con su correspondiente `recv`, fuente de interbloqueos y problemas en la planificación de comunicaciones.

En el capítulo 3 los códigos paralelos presentados se apoyan en el modelo de pase de mensajes, donde los tres interfaces, PVM, MPI y SHMEM han sido utilizados.

2.2.2 Lenguajes de paralelismo de datos: CRAFT y HPF

La idea que subyace bajo el paradigma de paralelismo de datos, claramente heredada de las máquinas SIMD, es soportar completamente las operaciones sobre arrays en paralelo. En general, la distribución de los datos y las computaciones es una tarea que implementará el compilador, guiado por las indicaciones que le proporciona el programador mediante ciertas directivas o anotaciones.

Aunque se pierde control sobre el código y la eficiencia del programa paralelo suele ser inferior a la que se consigue mediante pase de mensajes, realmente el inconveniente más importante es que los compiladores de paralelismo de datos actuales aún no permiten codificar cualquier tipo de problema. En particular los problemas irregulares son considerados como la *bête-noire* de este paradigma, y una de las cuestiones más urgentes a resolver.

Sin embargo, para problemas regulares, donde los códigos están bien estructurados, estas herramientas pueden generar, en tiempo de compilación, eficientes programas paralelos utilizando simples distribuciones de datos y computaciones. En este caso las ventajas de este paradigma son claras: facilidad y simplicidad de las implementaciones, portabilidad de código existente o futuro y mantenimiento de la compatibilidad con otro estándar, el Fortran90 [1, 64].

Actualmente los lenguajes de paralelismo de datos más populares son el CM-Fortran [142], Fortran D [67], Vienna-Fortran [160], y sobre todo el considerado como estándar *High-Performance Fortran* (HPF) [40, 85, 86, 104]. Todos estos compiladores tienen las siguientes características en común:

- Son compiladores fuente-fuente que a partir del programa secuencial con anotaciones (directivas) generan un único código que se puede ejecutar en varios procesadores. Es decir, se apoya en el modelo SPMD (*Single Program Multiple Data*) donde un único programa determina las operaciones a realizar. Sin embargo no existe la sincronización a nivel de instrucción que encontrábamos en las arquitecturas SIMD.

- Espacio de direcciones global, ya que el programador ve una única memoria, dejándose al compilador la tarea de distribuir los datos, acceder a memoria, gestionar las comunicaciones, etc. Las directivas ayudan al compilador a decidir esta distribución y a gestionar adecuadamente las comunicaciones necesarias (ya sea mediante pase de mensajes o memoria compartida).

- Las directivas se indican mediante una palabra clave después de un comentario (por ejemplo, !HPF$), de forma que el código se puede seguir compilando en secuencial. Las directivas del compilador sólo afectarán a las prestaciones del código ejecutable, pero no a su significado.

- Las operaciones sobre estructuras de datos uni- o multi-dimensionales se realizarán en paralelo entre varios procesadores. Son necesarias algunas extensiones del lenguaje (aparte de las directivas) como las que proporciona el Fortran90: la sentencia FORALL para bucles sin dependencias, funciones intrínsecas de reducción incluídas en librerías, etc.

La implementación de HPF soportada por las plataformas MPP de Cray se llama HPF-Craft [46]. Esta implementación es un híbrido entre HPF_LOCAL (entorno extrínseco definido por HPF1.1 y 1.2), HPF2 [86] y CRAFT (antiguo entorno de paralelismo de datos proporcionado por Cray para el T3D), diseñado por Cray e implementado por PGI (The Portland Group). La implementación actual del compilador consiste en un preprocesador fuente-fuente que genera código Fortran90 (o Fortran77) con llamadas a rutinas de la librería SHMEM.

En el capítulo 4 se describe una aproximación para la representación en HPF-Craft de nuestros algoritmos de factorización dispersos que no dio buenos resultados, así que posteriormente introducimos unas mínimas extensiones del lenguaje HPF que permiten una eficiente implementación del problema.

2.2.3 Paralelización automática: Polaris y PFA

Un compilador no sólo simplifica la tarea de escribir y leer programas, sino que además proporciona portabilidad entre distintas plataformas. Sin embargo estas ventajas no deben suponer una significativa disminución en la eficiencia del código ejecutable. La coexistencia de estos tres aspectos dio lugar a que desapareciese progresivamente la programación en ensamblador, en una evolución hacia el uso intensivo de los lenguajes de alto nivel. Actualmente, la programación a mano de códigos paralelos mediante un modelo de pase de mensajes es el equivalente de lo que en aquellos tiempos era la programación en ensamblador. Sin embargo, los usuarios, esperan que algún día ellos puedan seguir escribiendo su programas en algún lenguaje de alto nivel y que un compilador se encarge de generar el ejecutable para cualquier plataforma independientemente de su arquitectura.

Sin lugar a equívocos, un compilador para una arquitectura paralela presenta una tarea mucho más compleja que para una máquina secuencial. Los primeros, deben afrontar aspectos como la cuantificación y explotación eficiente del paralelismo, sincronización, manejo de la localidad de datos espacial y temporal, así como otros más dependientes de la arquitectura en particular. La gran cantidad de información que será necesario procesar para abordar eficientemente estos aspectos, induce a la interacción entre el compilador, un subsistema en tiempo de ejecución, el sistema operativo e incluso puede que el usuario.

Distintamente a como sucedió con la vectorización automática de programas, totalmente entendida y superada [98], la paralelización automática no llega a madurar aunque se encuentra en una etapa de rápido desarrollo. Los dos ejemplos más significativos de herramientas de paralelización los encontramos avalados por sendos proyectos en la Universidad de Illinois: Parafrase-2 [75, 113] y Polaris [31, 32]. Ambos, reestructuran código secuencial en Fortran (y Parafrase-2 también en C), para convertirlos en sus versiones paralelas, proporcionando un interfaz adecuado para que el usuario pueda interaccionar en las distintas fases del proceso de transformación. Mientras Parafrase-2 no ha resultado totalmente operativo con códigos reales, Polaris proporciona códigos paralelos competitivamente eficientes no sólo para plataformas de memoria compartida, sino también compartida-distribuida como el Cray T3D [108]. Otro compilador representativo es SUIF [84] (*Stanford University Intermediate Format*). Este compilador tranforma códigos secuenciales en F77 y C en sus versiones paralelas SPMD para memoria compartida. El código generado contiene llamadas a una librería en tiempo de ejecución disponible actualmente para las plataformas de SGI y el multiprocesador DASH.

Un compilador paralelo comercial con el cual se compara Polaris en las referencias anteriores es PFA [137, 136] de KAI. En sus versiones para Fortran y C permite generar código paralelos para sus plataformas multiprocesador (Power Challenge, Origin 2000, etc), aunque suele necesitar un posterior ajuste por parte del usuario, para el cual se dispone de un potente interfaz gráfico.

En el capítulo 5 discutimos en mayor profundidad estas utilidades y abordamos el problema de paralelización automática de códigos dispersos aportándose nuevas técnicas. Nos centraremos principalmente en el compilador Polaris al que hemos tenido acceso. Esta elección se justifica en los excelentes resultados que Polaris ha proporcionado con un grán número de códigos reales, no sólo para plataformas de memoria compartida, sino también a través de rutinas de comunicación unilateral (`put` y `get`). El reestructurador de Polaris incluye varias técnicas tanto estáticas como dinámicas (en tiempo de ejecución), a saber: análisis de dependencias (en tiempo de compilación y/o de ejecución), análisis simbólico, detección de variables de inducción y de reducción, análisis interprocedural, privatización de arrays, etc.

2.2.4 Modelo de programación de memoria compartida

Un importante sector de los programadores de aplicaciones prefieren escribir sus algoritmos bajo el paradigma de programación de memoria compartida. La razón de esta opcción estriba principalmente en la facilidad de programación. Esta mayor sencillez en la programación es también una de las razones para que muchos de los paralelizadores automáticos generen código según este paradigma de programación (PFA, Polaris y SUIF, entre otros).

A cambio, por lo general, las aplicaciones desarrolladas según este esquema suelen reportar peores prestaciones que las basadas por ejemplo en pase de mensajes. Algunos arguyen que la razón pricicipal de estos resultados está en el menor control del programador sobre la localidad de los datos. Esto conduce a pérdidas de rendimiento debidas a contención de memoria, acceso a memorias no locales, coste temporal debido al soporte de coherencia cache como el *false sharing*[8], etc.

Básicamente, este modelo de programación se basa en la inclusión de ciertas directivas en el programa paralelo, en lugar de extender la sintaxis del lenguaje (normalmente Fortran 77 o 90). La directiva más importante es el `DOACROSS` o `DOALL` seguida de determinadas cláusulas. Su uso fuerza al compilador a generar un código paralelo que ejecuta las iteraciones del bucle al que acompaña sobre distintos procesadores. Por ejemplo, en las plataformas de SGI la directiva es `C$ DOACROSS` y las cláusulas pueden ser `IF`, para controlar el efecto de la directiva en tiempo de ejecución, `LOCAL`, para indicar las variables locales a cada procesador, `SHARE`, para indicar las variables compartidas, y `REDUCTION` para indicar las variables que se reducen, entre otras [135].

Otras directivas suelen ajustarse a las propuestas por el Foro de Computación

[8]Problema que aparece cuando dos procesadores acceden en paralelo a celdas de memoria correspondientes a una misma línea cache y alguno de ellos la modifica.

Paralela (PCF), cuyo modelo es la base para el estándard propuesto ANSI–X3H5. Estas directivas permiten identificar secciones de código paralelas (aparte de los bucles), secciones críticas, barreras de sincronización, etc. Sin embargo el paralelismo de bucle es mucho más utilizado y generalmente suficiente.

La condición esencial requerida para paralelizar un bucle correctamente es que cada iteración sea independiente de las demás. Formalmente se dice que no deben existir dependencias entre iteraciones o dependencias generadas por bucle (*loop-carried dependences*). Si el bucle cumple esta condición las iteraciónes se pueden ejecutar en cualquier orden o al mismo tiempo y el resultado sigue siendo el mismo. Existen tres tipos de dependencias: verdaderas o de flujo (también llamada de lectura despues de escritura); anti-dependencias (también de escritura después de lectura); y de salida (o de escritura después de escritura). Las del primer tipo, cuando aparecen, son inevitables ya que representan el flujo de los datos a través de los distintos pasos en la ejecución del programa. Sin embargo las anti-dependencias y las de salida son dependencias "falsas" causadas únicamente por el reuso de posiciones de memoria. Estos dos tipos de dependencias relacionadas con la memoria pueden ser eliminadas utilizando diferentes posiciones de memoria para distintos accesos. Eliminar estas dependencias permite explotar un mayor grado de paralelismo y aumentar así las prestaciones del código paralelo. La privatización es una técnica que permite romper estas falsas dependencias en máquinas de memoria compartida. Cada procesador realiza una copia local de la variable en conflicto y accede únicamente a su copia local.

Este modelo de programación será discutido más extensivamente a lo largo del capítulo 5 ya que el paralelizador automático Polaris genera código paralelo siguiendo este paradigma.

2.2.5 Librerías: BLAS, LAPACK y HSL

Hasta ahora hemos revisado utilidades y modelos de programación que permiten generar códigos paralelos más o menos eficientes con mayor o menor sencillez. Pero en cualquier caso, al final, ya sea uno o varios procesadores ejecutarán algún código donde es muy importante evitar referencias innecesarias a memoria. Los procesadores modernos basan gran parte de su eficiencia en una adecuada gestion de la jerarquía de memoria, que debe ser tenida en cuenta también por el programador para evitar que las prestaciones del algoritmo se vean limitadas por el tráfico de información con memoria. Esto es así debido a que el movimiento de datos en memoria puede ser, en los procesadores actuales, más costoso que las operaciones aritméticas. Reducir este coste es una motivación suficiente como para reorganizar los algoritmos existentes de forma que se minimicen los movimientos de datos.

Con esa intención aparece a principios de los 70's una librería orientada a la solución eficiente de problemas del álgebra lineal, llamada BLAS (de *Basic Linear Algebra Subprograms*) [99]. BLAS es un interfaz que especifica el nombre y la secuencia de argumentos de subrutinas FORTRAN que realizan operaciones básicas y frecuentes del algebra lineal. Existen tres niveles o conjuntos de rutinas BLAS. En el nivel uno las rutinas tienen al menos un vector como parámetro, por ejemplo la rutina SAXPY realiza la operación $y = y + \alpha x$ donde x e y son vectores y α es un escalar. Esta rutina tiene una relación operaciones de memoria – operaciones punto flotante de 3/2. Sin embargo el nivel dos de BLAS incluye operaciones matriz-vector como SGEMV que realiza la operación $y = \alpha Ax + \beta y$, donde A es una matriz. En este caso la relación anterior es de 1/2, es decir se realizan el doble de operaciones en punto flotante que de movimiento de datos. Por último el nivel 3 trata las operaciones matriz-matriz, como SGEMM que realiza la operación $C = \beta Cx + \alpha AB$, donde A, B y C son matrices. Estas operaciones reportan la mejor relación entre operaciones de memoria y FLOPS igual a $2/n$ donde n es la dimensión de las matrices [54]. Por ejemplo, las prestaciones en MFLOPS alcanzadas por estas rutinas en el Cray T3D, que alcanza 150 MFLOPS de pico, son las siguientes[5]:

Nivel BLAS:	1 (DAXPY)	2 (DGEMV)	3 (DGEMM)
MFLOPS:	29.3	54.7	107.9

Dado que estas rutinas proporcionan eficiencia, portabilidad y legibilidad de los códigos han sido muy utilizadas incluso para crear librerías matemáticas más completas. Por ejemplo la librería LAPACK (de *Linear Algebra Package*) [11] proporciona rutinas para resolver sistemas de ecuaciones lineales, mínimos cuadrados, autovalores, etc. Esta librería se creó a partir de otras dos: EISPACK [139] y LINPACK [53] pero prestando un mayor interes en explotar la localidad en cache. Aparte de estas rutinas secuenciales se están desarrollando las correspondientes versiones paralelas para plataformas de memoria distribuida y paradigma de pase de mensajes mediante PVM o MPI. Estas rutinas paralelas se agrupan dentro del paquete *ScaLAPACK* [30] apoyado en PBLAS (*Parallel BLAS*) y las rutinas BLACS (*Basic Linear Algebra Communications Subprogram*) para simplificar la gestión de las comunicaciones.

Sin embargo LAPACK está orientada a matrices densas o en banda, pero no proporciona rutinas para matrices dispersas. De esto se encargan otro paquetes como HSL (*Harwell Subroutine Library*) [141] o SPARSPACK [72] ambas proporcionan algoritmos dispersos secuenciales para resolver sistemas lineales, mínimos cuadrados, autovalores, etc. Para operaciones más sencillas se está completando la versión dispersa de las rutinas BLAS: SBLAS [59]. Dadas su eficiencia y portabilidad, las operaciones disponibles en las librerías BLAS y SBLAS han sido extensivamente usadas en nuestros algoritmos.

En el caso particular de algoritmos dispersos para la resolución de sistemas, otros paquetes con mantenimiento *software* son Y12M[163], métodos iterativos en YSMP, desarrollado en la Univ. de Yale [62], en las rutinas IMSL o en las ESSL de IBM. Evidentmente, existen más códigos secuenciales en desarrollo o como herramientas de investigación (como el paquete SMMS desarrollado en Wisconsin [6]). También existe un conjunto de librería para la resolución paralela de sistemas dispersos por métodos iterativos llamado *BlockSolve95* [92] o las rutinas CAPSS [122] incluidas en ScaLAPACK para la factorización Cholesky paralela de matrices simétricas definidas positivas. Pero hasta donde conocemos, no existe ningún algoritmo paralelo comercial para la factorización LU de matrices dispersas genéricas. En el capítulo 3 resumimos los trabajos y el estado del arte en la paralelización del problema de la factorización de matrices dispersas.