Introducción a la computación de altas prestaciones

Ivan Rodero Castro
Francesc Guim Bernat

PID_00209165

UOC
Universitat Oberta de Catalunya

Índice

Introducción..... 5

Objetivos..... 6

1. Motivaciones de la computación de altas prestaciones..... 7
1.1. Utilidad de la simulación numérica 7
1.2. Tipos básicos de aplicaciones orientadas a altas prestaciones 8
1.2.1. Aplicaciones *HTC*..... 9
1.2.2. Aplicaciones *HPC*..... 9

2. Paralelismo y arquitecturas paralelas..... 12
2.1. Necesidad de la computación paralela 12
2.2. Arquitecturas de computación paralela 14
2.2.1. Una instrucción, múltiples datos (*SIMD*) 14
2.2.2. Múltiples instrucciones, múltiples datos (*MIMD*) 18
2.2.3. Coherencia de memoria caché 22
2.3. Sistemas distribuidos 26

3. Programación de aplicaciones paralelas..... 27
3.1. Modelos de programación de memoria compartida 28
3.1.1. Programación con flujos 28
3.1.2. OpenMP 29
3.1.3. *CUDA* y OpenCL 30
3.2. Modelos de programación de memoria distribuida 33
3.2.1. *MPI*..... 33
3.2.2. *PGAS*..... 35
3.3. Modelos de programación híbridos 36

4. Rendimiento de aplicaciones paralelas..... 39
4.1. *Speedup* y eficiencia 39
4.2. Ley de Amdahl 41
4.3. Escalabilidad 42
4.4. Análisis de rendimiento de aplicaciones paralelas 43
4.4.1. Medidas de tiempos 44
4.4.2. Interfaces de instrumentación y monitorización 45
4.4.3. Herramientas de análisis de rendimiento 46
4.5. Análisis de rendimiento de sistemas de altas prestaciones 50
4.5.1. Pruebas de rendimiento 51
4.5.2. Lista Top500 52

5. Retos de la computación de altas prestaciones..... 57

5.1. Concurrencia extrema 58
5.2. Energía 59
5.3. Tolerancia a fallos 59
5.4. Heterogeneidad 60
5.5. Entrada/salida y memoria 60

Bibliografía..... 63

Introducción

En este primer módulo didáctico estudiaremos las motivaciones y características de la computación de altas prestaciones y repasaremos sus fundamentos, de arquitectura y de programación, para poner en contexto y vertebrar los contenidos de los siguientes módulos didácticos.

Estudiaremos las arquitecturas de los sistemas paralelos y otros conceptos de relevancia, como por ejemplo las redes de interconexión. También estudiaremos conceptos fundamentales relacionados con el rendimiento de sistemas de altas prestaciones, así como métricas y el papel que desempeñan las herramientas de análisis de rendimiento.

Una vez presentados los sistemas paralelos, nos centraremos en su programación. Veremos los conceptos fundamentales de los modelos de programación, tanto para memoria compartida (por ejemplo, OpenMP) como para memoria distribuida (por ejemplo, MPI).

Finalmente, estudiaremos los retos actuales más importantes de la computación de altas prestaciones, como los relacionados con la gestión de paralelismo masivo o las limitaciones en cuanto a la disponibilidad de energía.

Objetivos

Los materiales didácticos de este módulo contienen las herramientas necesarias para lograr los objetivos siguientes:

1. Entender las motivaciones de la computación de altas prestaciones y del paralelismo.

2. Conocer los fundamentos del paralelismo y las arquitecturas paralelas, tanto los relacionados con sistemas de memoria compartida como de memoria distribuida.

3. Conocer las características de los sistemas paralelos, como por ejemplo, la jerarquía de memoria, redes de interconexión, etc.

4. Entender las diferencias entre sistemas paralelos y distribuidos.

5. Conocer los modelos de programación para sistemas de altas prestaciones, tanto de memoria compartida como de memoria distribuida.

6. Conocer los fundamentos relacionados con el rendimiento de sistemas de altas prestaciones y del análisis de rendimiento.

7. Estar al corriente de los retos actuales de la computación de altas prestaciones.

1. Motivaciones de la computación de altas prestaciones

En general, la computación de altas prestaciones ha estado motivada por la continua y creciente demanda de prestaciones y velocidad de computación necesarias para resolver problemas computacionales cada vez más complejos en un tiempo razonable. Esta gran demanda de computación es requerida por áreas como, por ejemplo, modelos numéricos y simulación de problemas de ciencia e ingeniería.

1.1. Utilidad de la simulación numérica

La simulación numérica se considera el tercer pilar de la ciencia, además de la experimentación u observación y la teoría. El paradigma tradicional de la ciencia e ingeniería está basado en la teoría o el diseño en papel para después realizar experimentos o crear el sistema. Este paradigma tiene numerosas limitaciones frente al problema que hay que resolver, como:

- El problema es muy difícil, como por ejemplo, construir túneles de viento enormes.
- El problema es muy caro económicamente, como por ejemplo, fabricar un avión solo para experimentar.
- El problema es demasiado lento, como por ejemplo, esperar el efecto del cambio climático o la evolución de galaxias.
- El problema es muy peligroso, como por ejemplo, pruebas de armas, diseño de fármacos, experimentación con el medio ambiente, etc.

El paradigma de la ciencia computacional está basado en **utilizar sistemas de altas prestaciones** para simular el fenómeno deseado. Por lo tanto, la ciencia computacional se basa en las leyes de la física y los métodos numéricos eficientes.

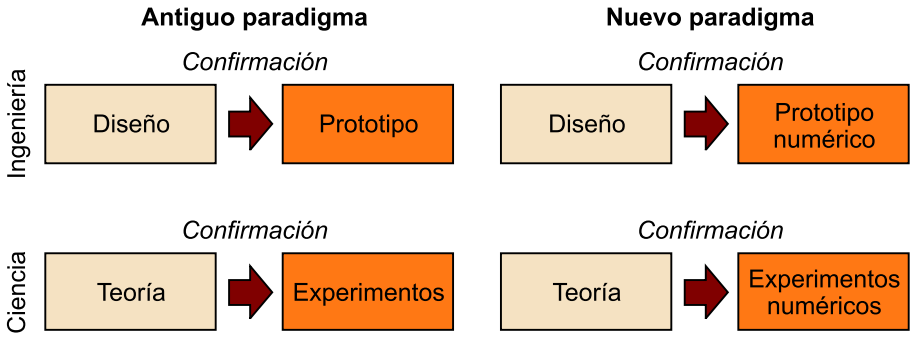
Algunas de las aplicaciones que tradicionalmente han sido un reto para la comunidad y que necesitan computación de altas prestaciones, puesto que no pueden ser ejecutadas con otro tipo de computador (por ejemplo, computadores personales) son:

- Aplicaciones científicas, como por ejemplo, el cambio climático, los modelos astrofísicos, el análisis genómico, el plegamiento de proteínas (diseño de fármacos), etc.

- Aplicaciones de ingeniería, como por ejemplo, la simulación de tests de choque, el diseño de semiconductores, los modelos estructurales de terremotos, etc.
- Aplicaciones de negocio, como por ejemplo, los modelos financieros y económicos, el proceso de transacciones, los servicios web, los motores de busca, etc.
- Aplicaciones de defensa y militares, como por ejemplo, las simulaciones de pruebas con armas nucleares, la criptografía, etc.

En aplicaciones de ingeniería, la computación de altas prestaciones permite disminuir la utilización de prototipos en el proceso de diseño y optimización de procesos. Así pues, permite disminuir costes, aumentar la productividad –al disminuir el tiempo de desarrollo– y aumentar la seguridad. En aplicaciones científicas, la computación de altas prestaciones permite la simulación de sistemas a gran escala, sistemas a muy pequeña escala y el análisis de la validez de un modelo matemático. La figura siguiente ilustra el cambio de paradigma en aplicaciones de ingeniería y ciencia.

Figura 1. Cambio de paradigma en aplicaciones de ingeniería y ciencia



1.2. Tipos básicos de aplicaciones orientadas a altas prestaciones

Las aplicaciones que requieren altas prestaciones se pueden dividir en dos grandes grupos: **aplicaciones de alta productividad o** *HTC*[1] y **aplicaciones de altas prestaciones o** *HPC*[2].

(1) *Del* inglés *high throughput computing.*

(2) *Del* inglés *high performance computing.*

También lo podemos ver desde el punto de vista de la manera de tratar el problema. La computación paralela permite el desarrollo de aplicaciones que aprovechen la utilización de múltiples procesadores de forma colaborativa con el objetivo de resolver un problema común. De hecho, el objetivo fundamental que persiguen estas técnicas es conseguir reducir el tiempo de ejecución de una aplicación mediante la utilización de múltiples procesadores. Adicionalmente, también es posible querer resolver problemas mayores mediante el aprovechamiento de las diferentes memorias de los procesadores involucrados en la ejecución. Podemos hablar básicamente de dos conceptos fundamentales:

1) Particionamiento de tareas: consiste en dividir una tarea grande en un número de diferentes subtareas que puedan ser complementadas por diferentes unidades de proceso.

2) Comunicación entre tareas: a pesar de que cada proceso realice una subtarea, generalmente será necesario que estos se comuniquen para poder cooperar en la obtención de la solución global del problema.

1.2.1. Aplicaciones *HTC*

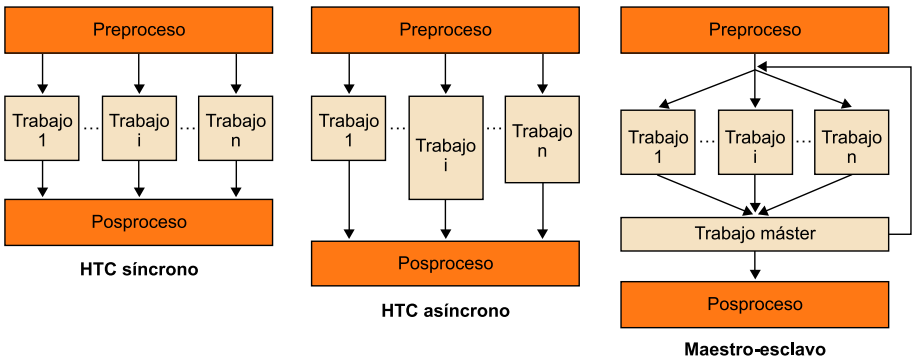
El objetivo de las aplicaciones *HTC* es aumentar el número de ejecuciones por unidad de tiempo. Su rendimiento se mide en número de trabajos ejecutados por unidad de tiempo (por ejemplo, trabajos por segundo).

Aplicaciones *HTC*

Algunas de las áreas que suelen requerir este tipo de aplicaciones son la bioinformática o las finanzas.

En la figura siguiente se muestran tres tipos de modelos de aplicaciones *HTC* suponiendo que la tarea que hay que ejecutar se pueda descomponer en diferentes trabajos (normalmente denominados *jobs*). En el primero (*HTC* síncrono), los trabajos están sincronizados y acaban a la vez; en el segundo (*HTC* asíncrono), los trabajos acaban en diferentes instantes, y en el último (maestro-esclavo), hay un trabajo especializado (maestro) que se encarga de sincronizar el resto de los trabajos (esclavos).

Figura 2. Modelos de aplicaciones *HTC*



1.2.2. Aplicaciones *HPC*

El objetivo de las aplicaciones *HPC* es reducir el tiempo de ejecución de una única aplicación paralela. Su rendimiento se mide en número de operaciones en punto flotante por segundo[3] (Flop/s) y normalmente se suele medir en millones, billones, trillones, etc., tal y como muestra la tabla 1.

[3]En inglés, *floating point operations per second*.

Tabla 1. Unidades de medida de rendimiento

Nombre	Flops
Megaflops	10^6

Nombre	Flops
Gigaflops	10^9
Teraflops	10^{12}
Petaflops	10^{15}
Exaflops	10^{18}
Zettaflops	10^{21}
Yottaflops	10^{24}

Algunas áreas que suelen requerir este tipo de aplicaciones son:

a) Estudio de fenómenos a escala microscópica (dinámica de partículas). La resolución está limitada por la potencia de cálculo del computador. Cuantos más grados de libertad (puntos), mejor se refleja la realidad.

b) Estudio de fenómenos a escala macroscópica (sistemas descritos por ecuaciones diferenciales fundamentales). La precisión está limitada por la potencia de cálculo del computador. Cuantos más puntos, más se acerca la solución discreta a la continua.

Actualmente la computación de altas prestaciones es sinónimo de paralelismo, por lo tanto, del aumento del número de procesadores. En general, se quiere aumentar el número de procesadores para:

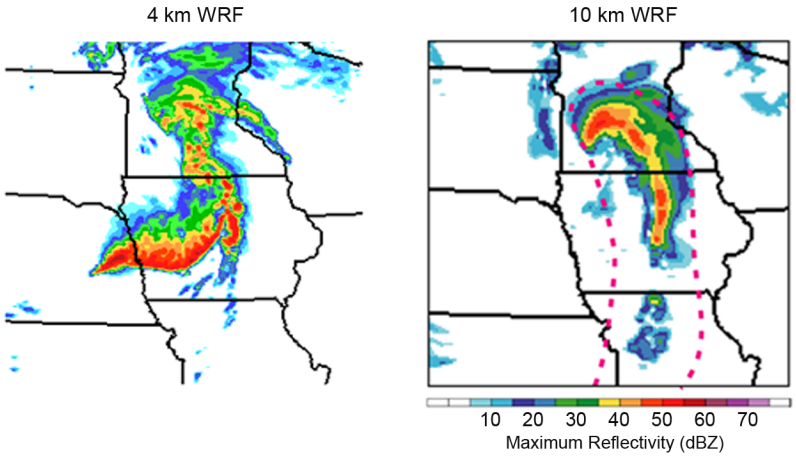
- Resolver problemas en un tiempo de ejecución inferior, utilizando más procesadores.
- Resolver problemas con más precisión, utilizando más memoria.
- Resolver problemas más reales, utilizando modelos matemáticos más complejos.

La predicción meteorológica

Un ejemplo de esto es la predicción meteorológica, que requiere gran capacidad de cálculo y de memoria. La predicción meteorológica es una función de la longitud, latitud, altura y tiempo. Para cada uno de estos puntos se deben calcular varios parámetros, como por ejemplo, la temperatura, presión, humedad y velocidad del viento. Hay casos en los que la predicción meteorológica se necesita en "tiempo real" (en cuestión de horas, no de días), como en el caso de la predicción de la formación, la trayectoria y el posible impacto de un huracán. Hay que tener en cuenta que llevar a cabo las simulaciones con una precisión insuficiente puede provocar perder detalles importantes y llegar a conclusiones poco acertadas que pueden tener consecuencias devastadoras. La figura 3 ilustra los resultados obtenidos con el modelo *WRF*[4] con dos niveles de precisión diferentes:

[(4)]Weather Research and Forecasting.

Figura 3. Predicciones meteorológicas con *WRF* de diferentes precisiones

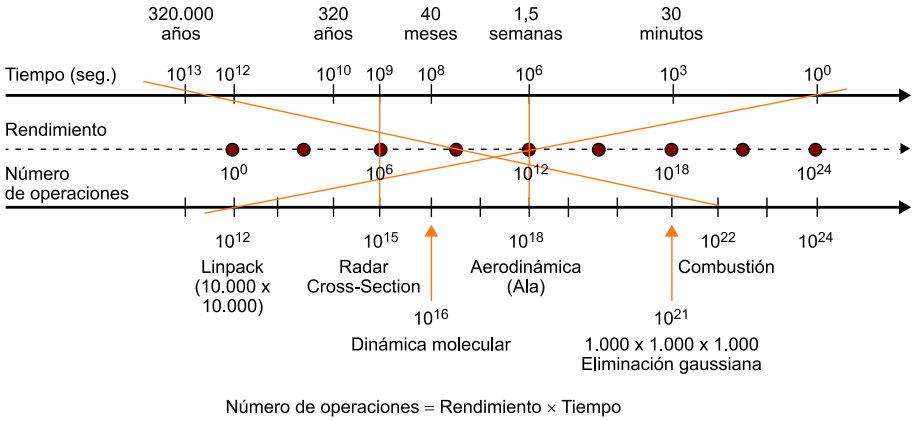


Fuente: The National Center for Atmospheric Research

En la simulación del gráfico de la izquierda la cuadrícula es de 4 kilómetros cuadrados, mientras que en el de la derecha es de 10 kilómetros cuadrados. Se puede apreciar claramente que hay ciertos fenómenos que no se pueden observar con claridad en la figura de la derecha. Hay que tener en cuenta que una predicción cuidadosa de estos fenómenos, aparte del impacto directo en la población, también afecta a la economía (por ejemplo, se estima que en Estados Unidos el 40% de pequeñas/medianas empresas cierran en los siguientes 36 meses si se ven forzadas a cerrar 3 o más días después de un huracán). Así pues, disponer de computación de altas prestaciones puede llegar a salvar vidas y/o la economía de regiones.

La figura 4 muestra otros ejemplos clásicos que requieren gran potencia de cálculo y, por lo tanto, computación de altas prestaciones, y los cuantifica en cuanto a cantidad de cálculo requerido para llevarlos a cabo.

Figura 4. Ejemplos de la necesidad de altas prestaciones



Número de operaciones = Rendimiento × Tiempo

2. Paralelismo y arquitecturas paralelas

2.1. Necesidad de la computación paralela

Durante las últimas décadas, el rendimiento de los microprocesadores se ha incrementado de media en un 50% por año. Este incremento en rendimiento y prestaciones significa que los usuarios y los desarrolladores de programas podían simplemente esperar la siguiente generación de microprocesadores para obtener una mejora sustancial en el rendimiento de sus programas. En cambio, desde el 2002 la mejora de rendimiento de los procesadores de forma individual se ha frenado en un 20% por año. Esta diferencia es muy grande, puesto que si tenemos en cuenta un incremento de rendimiento del 50% por año, en cuestión de 10 años el factor sería de aproximadamente 60 veces, mientras que con un incremento del 20%, el factor solo es de 6 veces.

Uno de los métodos más relevantes para mejorar el rendimiento de los computadores ha sido el aumento de la velocidad del reloj del procesador debido al aumento de la densidad de los procesadores. A medida que el tamaño de los transistores disminuye, se puede aumentar su velocidad y, en consecuencia, la velocidad global del circuito integrado aumenta.

Esto está motivado por la ley de Moore, una ley empírica que dice que "la densidad de circuitos en un chip se dobla cada 18 meses" (Gordon Moore, Chairman, Intel, 1965). La tabla 2 muestra la evolución de la tecnología en los procesadores Intel. Aun así, la ley de Moore tiene límites evidentes.

Los límites de la ley de Moore

Por ejemplo, si tenemos en cuenta la velocidad de la luz como velocidad de referencia para el movimiento de los datos en un chip, para desarrollar un computador que tenga un rendimiento de 1 Tflop con 1 TB de datos (1 THz), la distancia (r) para acceder al dato en memoria debería ser inferior a $c/10^{12}$. Esto quiere decir que el tamaño del chip tendría que ser de 0,3 x 0,3 mm. Para contener 1 TB de información, una palabra de memoria debería ocupar 3 amstrongs x 3 amstrongs, que es el tamaño ¡de un átomo!

Tabla 2. Evolución de la tecnología de microprocesadores (familia Intel no completa)

Procesador	Año	Número de transistores	Tecnología
4004	1971	2.250	10 µm
8008	1972	3.500	10 µm
8080	1974	6.000	6 µm
8086	1978	29.000	3 µm
286	1982	134.000	1,5 µm

Procesador	Año	Número de transistores	Tecnología
386	1985	275.000	1 µm
486 DX	1989	1.200.000	0,8 µm
Pentium	1993	3.100.000	0,8 µm
Pentium II	1997	7.500.000	0,35 µm
Pentium III	1999	28.000.000	180 nm
Pentium IV	2002	55.000.000	130 nm
Core 2 Duo	2006	291.000.000	65 nm
Core i7 (Quad)	2008	731.000.000	45 nm
Core i7 (Sandy Bridge)	2011	2.300.000.000	32 nm

También hay que tener en cuenta que a medida que la velocidad de los transistores aumenta, el consumo eléctrico también lo hace. La mayoría de este consumo se transforma en calor, y cuando un circuito integrado se calienta mucho no es fiable. Incluso podemos encontrar limitaciones en cuanto a paralelismo a nivel de de instrucción[5], puesto que haciendo el *pipeline* más y más grande se puede acabar obteniendo un peor rendimiento.

(5)En inglés, *instruction level parallelism*.

Así pues, actualmente no es viable continuar incrementando la velocidad de los circuitos integrados. En cambio, todavía podemos continuar incrementando la densidad de los transistores, pero hay que destacar que solo por un cierto tiempo.

La manera que nos queda para sacar partido al aumento en la densidad de los transistores es mediante el paralelismo. En vez de construir procesadores monolíticos cada vez más rápidos y complejos, la solución que la industria adoptó fue el desarrollo de procesadores con múltiples núcleos, centrándose en el rendimiento de ejecución de aplicaciones paralelas en lugar de programas secuenciales. De esta manera, en los últimos años se ha producido un cambio muy significativo en la industria de la computación paralela.

Actualmente, casi todos los ordenadores de consumo incorporan procesadores multinúcleo[6] y el concepto de núcleo[7] se ha convertido en sinónimo de CPU. Desde la incorporación de los procesadores multinúcleo en dispositivos cotidianos, desde procesadores duales para dispositivos móviles hasta procesadores con más de una docena de núcleos para servidores y estaciones de trabajo, la computación paralela ha dejado de ser exclusiva de supercomputadores y sistemas de altas prestaciones. Así, estos dispositivos proporcionan funcionalidades más sofisticadas que sus predecesores mediante computación paralela.

(6)En inglés, *multicore*.

(7)En inglés, *core*.

Este cambio también tiene consecuencias drásticas de cara a los programadores, puesto que las nuevas generaciones de circuitos integrados que incorporan más procesadores no mejoran automáticamente el rendimiento de las aplicaciones secuenciales, como sucedía hasta entonces. Tal y como veremos más adelante, serán necesarias nuevas técnicas y modelos de programación para sacar provecho a las nuevas generaciones de procesadores.

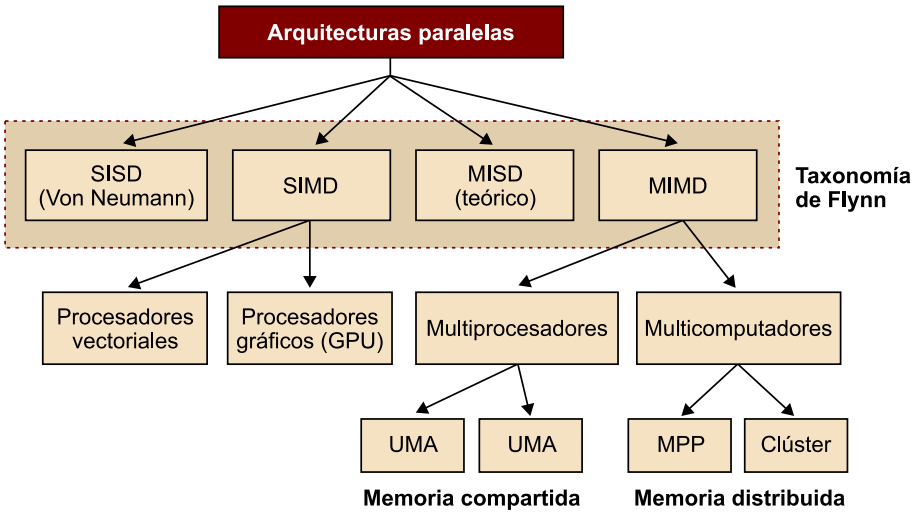
2.2. Arquitecturas de computación paralela

En computación paralela normalmente se suele utilizar la taxonomía de Flynn para clasificar las arquitecturas de computadores. Esta clasifica un sistema según el número de flujos de datos y de instrucciones que pueden gestionar simultáneamente. A pesar de que veremos esta taxonomía con más detalle en otro módulo didáctico, los dos tipos fundamentales se exponen a continuación. La figura siguiente muestra la clasificación de las arquitecturas paralelas, incluyendo la taxonomía de Flynn.

Ved también

La taxonomía de Flynn se trata con detenimiento en el módulo "Procesadores y modelos de programación" de esta asignatura.

Figura 5. Clasificación de las arquitecturas paralelas



2.2.1. Una instrucción, múltiples datos (*SIMD*)

En los sistemas *SIMD*[8], una misma instrucción se aplica sobre varios datos, así que se podría ver como tener una única unidad de control y múltiples ALU.

(8)Del inglés *single instruction, multiple data stream.*

Una instrucción se envía desde la unidad de control hacia todas las ALU y cada una de las ALU o bien aplica la instrucción a su conjunto de datos o bien queda sin trabajo si no tiene datos asociados. Los sistemas *SIMD* son ideales para ciertas tareas, como por ejemplo para paralelizar bucles sencillos que operan sobre vectores de datos de gran tamaño. Este tipo de paralelismo se denomina paralelismo de datos. El problema principal de los sistemas *SIMD* es que no siempre funciona bien para todos los tipos de problemas. Estos sistemas han evolucionado de una manera un poco peculiar durante su historia. A comien-

zos de los años noventa la mayoría de los supercomputadores paralelos estaban basados en sistemas *SIMD*. En cambio, a finales de la misma década los sistemas *SIMD* eran básicamente procesadores vectoriales. Más recientemente, los procesadores gráficos o GPU y los procesadores de gran consumo usan aspectos de la arquitectura *SIMD*.

Procesadores vectoriales

A pesar de que lo que constituye un procesador vectorial ha ido cambiando con el paso de los años, su característica clave es que **operan sobre vectores de datos**, mientras que los procesadores convencionales operan sobre elementos de datos individuales o escalares.

Estos tipos de procesadores son rápidos y fáciles de utilizar para bastantes tipos de aplicaciones. Los compiladores que vectorizan el código son muy buenos identificando código que se puede vectorizar y también bucles que no se pueden vectorizar. Los sistemas vectoriales tienen un ancho de banda en memoria muy elevado y todos los datos que se cargan se utilizan en contra de los sistemas basados en memoria caché, que no hacen uso de todos los elementos de una línea de memoria caché. La parte negativa de los sistemas vectoriales es que no pueden trabajar con estructuras de datos irregulares y, por lo tanto, tienen limitaciones importantes respecto a su escalabilidad, puesto que no pueden tratar problemas muy grandes. Los procesadores vectoriales actuales tienen las siguientes características:

- Registros vectoriales: son registros que son capaces de guardar vectores de operandos y operar simultáneamente sobre sus contenidos. El tamaño del vector lo fija el sistema y puede oscilar desde 4 hasta, por ejemplo, 128 elementos de 64 bits.

- Unidades funcional vectorizadas: hay que tener en cuenta que la misma operación se aplica a cada elemento del vector o, en operaciones como por ejemplo la suma, la misma operación se aplica a cada par de elementos de los dos vectores de una sola vez. Por lo tanto, las operaciones son *SIMD*.

- Instrucciones sobre vectores: son instrucciones que operan sobre vectores en vez de sobre escalares. Esto provoca que para realizar las operaciones, en lugar de hacerlo individualmente para cada elemento del vector (por ejemplo, *load*, *add* y *store* para incrementar el valor de cada elemento del vector) se pueda hacer por bloques.

- Memoria intercalada[9]: el sistema de memoria consiste en múltiples "bancos" de memoria, a los que se puede acceder más o menos independientemente. Después de acceder a un banco, habrá una demora antes de poder volver a acceder a él, pero a los otros bancos se puede acceder mucho más

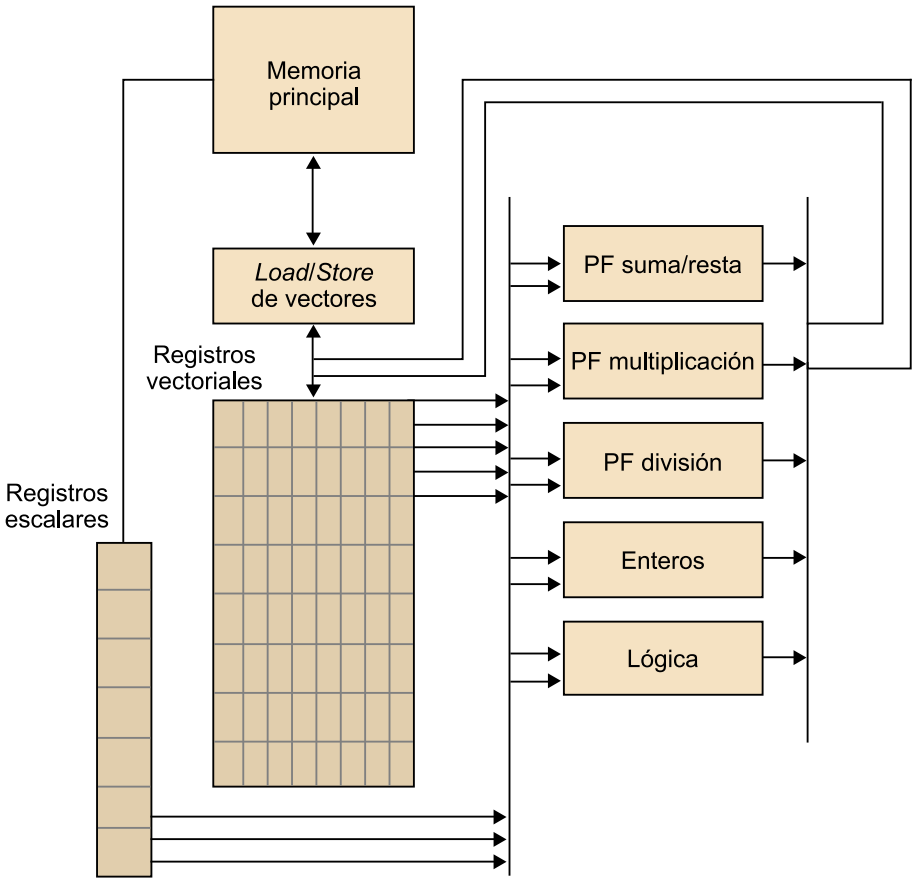
(9)En inglés, *interleaved memory*.

pronto. Si los elementos de un vector están distribuidos en múltiples bancos, habrá un acceso muy rápido para leer o escribir elementos sucesivos.

- Acceso a memoria por intervalos: con este tipo de acceso el programa accede a elementos de un vector localizado a intervalos. Por ejemplo, accediendo al segundo elemento, al sexto, al décimo, etc. sería acceso a memoria en un intervalo de 4.

La figura siguiente muestra un procesador vectorial simple, donde se puede apreciar que los bloques más básicos pueden actuar sobre un conjunto de registros vectoriales.

Figura 6. Arquitectura vectorial simple



Procesadores gráficos (GPU)

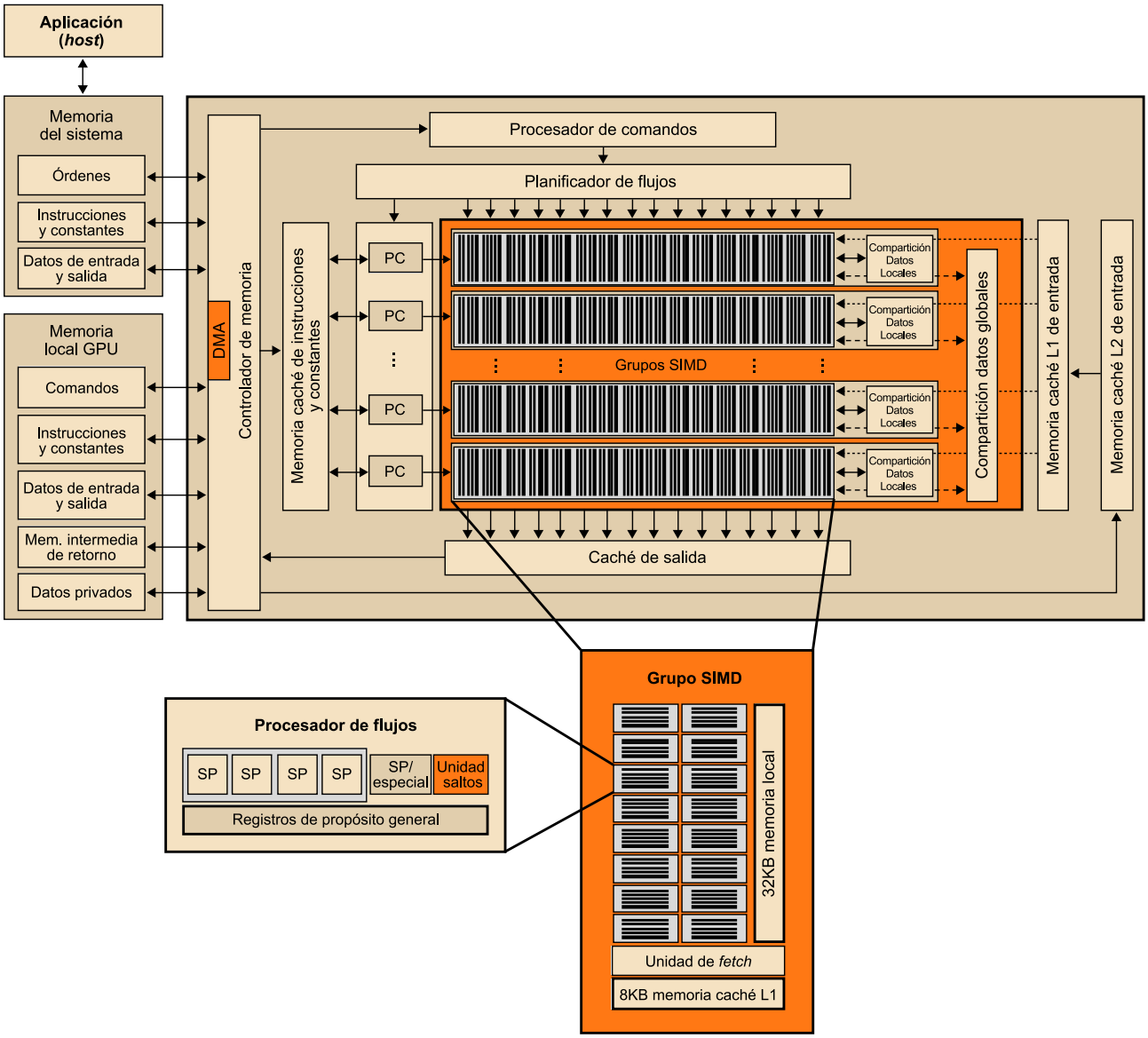
Los procesadores gráficos tradicionalmente **funcionan mediante un *pipeline* de procesamiento** formado por etapas muy especializadas en las funciones que desarrollan, y que se ejecutan en un orden preestablecido. Cada una de las etapas del *pipeline* recibe la salida de la etapa anterior y proporciona su salida a la etapa siguiente.

Gracias a la implementación mediante una estructura de *pipeline*, el procesador gráfico puede ejecutar varias operaciones en paralelo. Como este *pipeline* es específico para la gestión de gráficos, normalmente se denomina *pipeline* gráfico o pipeline de renderización. La operación de renderización consiste en proyectar una representación en 3 dimensiones en una imagen en 2 dimensiones (que es el objetivo de un procesador gráfico). Aun así, hay etapas del *pipeline* gráfico que se pueden programar, e incluso en GPU actuales orientadas a propósito general hay gran flexibilidad de programación mediante los llamados *kernels*. Los *kernels* son normalmente códigos bastante cortos en los que el paralelismo está implícito, puesto que, cuando se instancian los *kernels*, estos se encargan de procesar la parte de los datos que les correspondan. De hecho, las GPU también pueden utilizar paralelismo *SIMD* para mejorar el rendimiento, y las generaciones actuales de GPU lo hacen poniendo un número elevado de ALU (podéis ver la figura 7) en cada núcleo. Los procesadores gráficos se están haciendo muy populares en la computación de altas prestaciones y varios lenguajes se han desarrollado para facilitar al usuario su programación.

Número elevado de ALU

En la arquitectura Evergreen, de AMD, se ponen 1.600 ALU.

Figura 7. Diagrama de bloques de la arquitectura Evergreen de AMD



2.2.2. Múltiples instrucciones, múltiples datos (*MIMD*)

Los sistemas *MIMD*[10] normalmente consisten en un conjunto de unidades de procesamiento o **núcleoscompletamente independientes** que tienen su propia unidad de control y su propia ALU.

(10)Del inglés *multiple instruction, multiple data stream* (*MIMD*).

A diferencia de los sistemas *SIMD*, los *MIMD* normalmente son asíncronos, es decir, que pueden operar por su parte. En muchos sistemas *MIMD*, además, no hay un reloj global y puede ser que no haya relación entre los tiempos de los sistemas de dos procesadores diferentes. De hecho, a no ser que el programador imponga cierta sincronización, incluso aunque los procesadores estén ejecutando la misma secuencia de instrucciones, en un momento de tiempo concreto pueden estar ejecutando partes diferentes.

Hay dos tipos principales de sistemas *MIMD*: sistemas de memoria compartida y sistemas de memoria distribuida, tal y como ilustran las figuras 8 y 9. En un sistema de memoria compartida un conjunto de procesadores autónomos se conectan al sistema de memoria mediante una red de interconexión y cada procesador puede acceder a cualquier parte de la memoria. Los procesadores normalmente se comunican implícitamente mediante estructuras de datos compartidos en la memoria. En un sistema de memoria distribuida, cada procesador está ligado a su propia memoria privada y los conjuntos procesador-memoria se comunican mediante una red de interconexión. Por lo tanto, en un sistema de memoria distribuida los procesadores normalmente se comunican explícitamente enviando mensajes o utilizando funciones especiales que proporcionan acceso a la memoria de otro procesador.

Acceder a la memoria de otro procesador

Un ejemplo de este último tipo es RDMA, una característica de interfaces de red que permiten a un computador acceder directamente a la memoria de otro computador sin pasar por el sistema operativo. Una implementación que se utiliza en computación de altas prestaciones es sobre Infiniband.

Actividad

Os proponemos que busquéis información sobre Infiniband y de su implementación de RDMA.

Figura 8. Sistema de memoria compartida

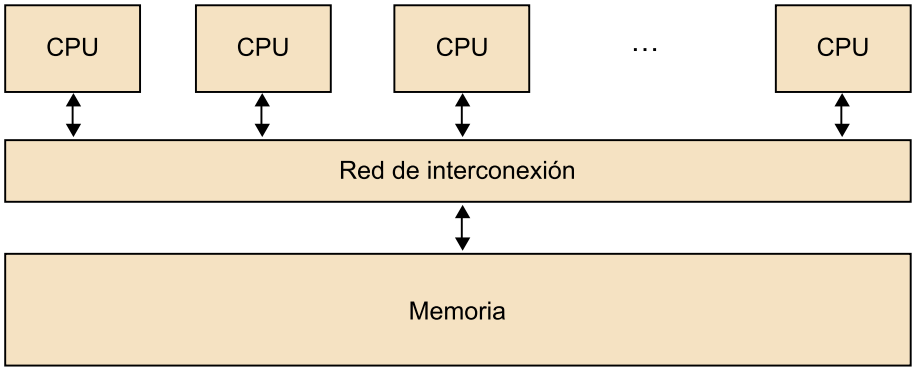
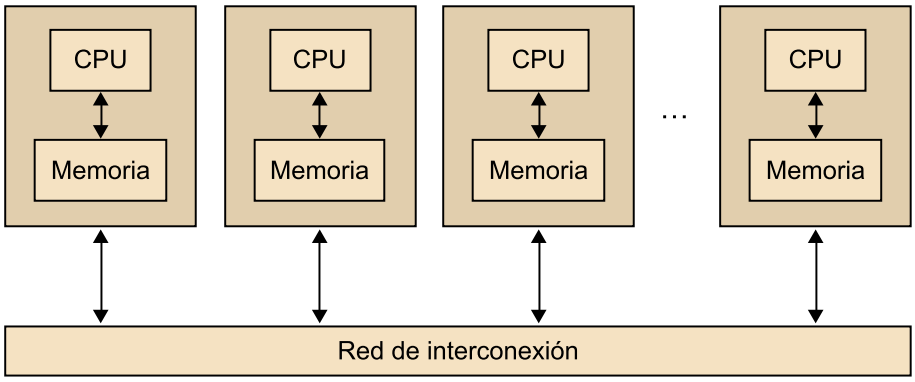


Figura 9. Sistema de memoria distribuida



Sistemas de memoria compartida

Los sistemas de memoria compartida más utilizados utilizan uno o más procesadores multinúcleo, que utilizan una o más de una CPU en un mismo chip. Típicamente, los núcleos tienen una memoria caché privada de nivel 1, mientras que otras memorias caché pueden estar o no compartidas entre los distintos núcleos.

En sistemas de memoria compartida con múltiples procesadores multinúcleo, la red de interconexión puede o bien conectar todos los procesadores directamente con la memoria principal o bien cada procesador puede tener acceso directo a un bloque de la memoria principal, y los procesadores pueden acceder a los bloques de memoria de los otros procesadores mediante hardware especializado incorporado en los procesadores, tal y como muestran las figuras 10 y 11.

Figura 10. Sistema multinúcleo *UMA*

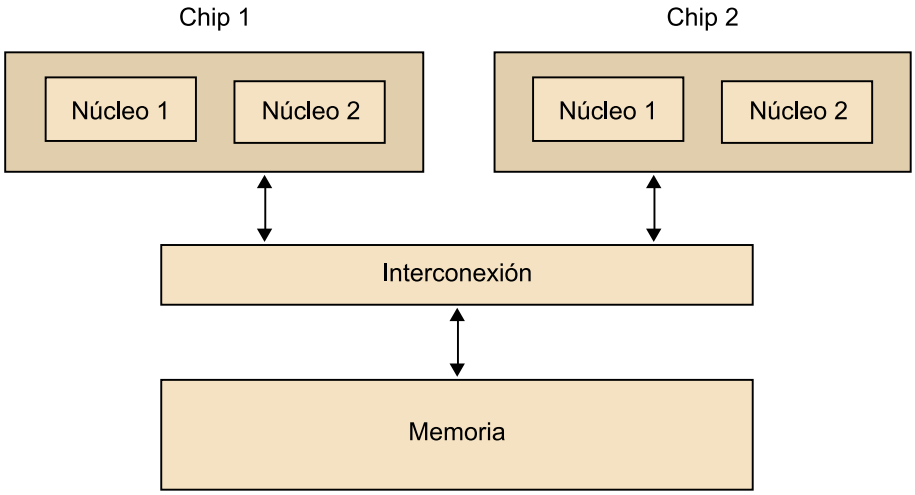
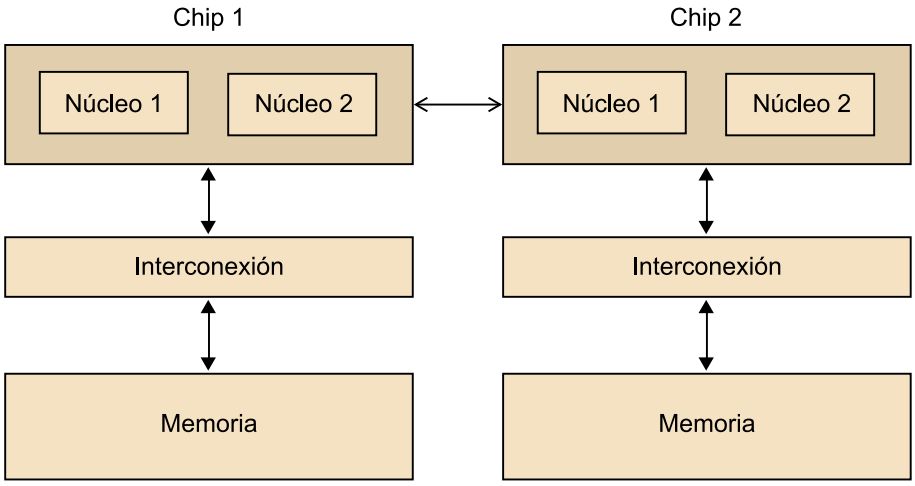


Figura 11. Sistema multinúcleo *NUMA*



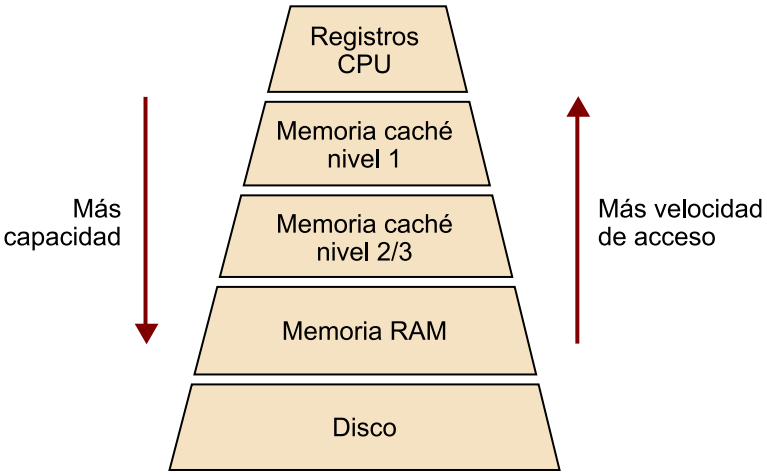
En el primer tipo de sistema, el tiempo de acceso a todas las posiciones de memoria es el mismo para todos los núcleos, mientras que en el segundo tipo el tiempo de acceso a la memoria, a la que el núcleo está directamente conectado, será más rápido que el de acceder a la memoria de otro chip. Por lo tanto, el primer tipo de sistema se denomina *UMA*[11] y el segundo se denomina *NUMA*[12]. Los sistemas *UMA* normalmente son mucho más fáciles de programar, puesto que el programador no se ha de preocupar del tiempo de acceso a los datos y, por lo tanto, de la localidad de los datos. Esta ventaja, no obstante, queda en cierto modo compensada por el acceso más rápido a la memoria a la que el núcleo está conectado en los sistemas *NUMA* y también por el hecho de que normalmente estos sistemas suelen disponer de más cantidad de memoria que los *UMA*.

(11) *Del* inglés *uniform memory access*.

(12) Del inglés *non-uniform memory access*.

Por lo tanto, podemos decir que la jerarquía de memoria y su gestión eficiente es clave para la computación de altas prestaciones. La figura siguiente muestra los niveles clásicos de la jerarquía de memoria de un computador enfatizando el hecho de que cuanto más rápida es una memoria, más pequeña y costosa suele ser. La existencia de diferentes niveles de memoria implica que el tiempo dedicado a realizar una operación de acceso a memoria de un nivel aumenta con la distancia al nivel más rápido, que es el acceso a los registros del procesador. Una mala gestión de la memoria provoca muchos fallos de página, que acaban realizando un uso excesivo del sistema de memoria virtual, lo cual implica realizar costosos accesos al disco. Por lo tanto, el diseño de aplicaciones eficientes requiere un completo conocimiento de la estructura de la memoria del computador y saber explotarla.

Figura 12. Jerarquía de memoria de un computador



La optimización del proceso de cálculo secuencial implica, entre otras cosas, una selección adecuada de las estructuras de datos que se van a utilizar para representar la información relativa al problema, atendiendo a criterios de eficiencia computacional. Por ejemplo, la utilización de técnicas de almacenamiento de matrices dispersas pertenece a esta categoría, lo que permite almacenar únicamente los elementos no nulos de una matriz. Además, este nivel incluye la selección de los métodos eficientes para la resolución del problema. Una fase importante de las aplicaciones, especialmente para aquellas orientadas a procesos de simulación, corresponde a la grabación de datos en disco. Dado que el acceso a un disco duro presenta tiempos de acceso muy superiores a los correspondientes a memoria RAM, resulta indispensable realizar una gestión eficiente del proceso de E/S[13] para que tenga el mínimo impacto sobre el proceso de simulación.

[13] Se refiere a un proceso de entrada/salida.

Finalmente, y por encima del resto de las capas, aparece la utilización de computación paralela. En este sentido, una buena estrategia de partición de tareas que garantice una distribución equitativa de la carga entre los procesadores involucrados, así como la minimización del número de comunicaciones entre estos, permite reducir los tiempos de ejecución. Además, con la participación de las principales estructuras de datos de la aplicación entre los diferentes procesadores se puede conseguir la resolución de problemas más grandes.

Sistemas de memoria distribuida

Los sistemas de memoria distribuida más extendidos y populares son los denominados clústeres. Estos están compuestos por un conjunto de sistemas de consumo[14], como por ejemplo PC, conectados a una red de interconexión también de consumo, como por ejemplo Ethernet. De hecho, los nodos de estos clústeres, que son unidades de computación individuales interconectadas mediante una red, son normalmente sistemas de memoria compartida con uno o

[14] En inglés, *commodity*.

más procesadores multinúcleo. Para diferenciar estos sistemas de los que son puramente de memoria distribuida, en ocasiones se denominan sistemas híbridos.

Actualmente se entiende que un clúster está compuesto de nodos de memoria compartida. Además, también se entiende que los grandes sistemas de computación paralelos son clústeres por definición. La principal diferencia entre grandes centros de procesamiento o de datos[15] que utiliza la industria (por ejemplo, Google, Facebook, etc.) y los supercomputadores es la red de interconexión. La clave en estos sistemas está en poder ofrecer una latencia muy reducida en la red de interconexión para que el paso de mensajes sea muy rápido, puesto que las aplicaciones que requieren supercomputación son básicamente fuertemente acopladas[16], es decir, que los diferentes procesos que se ejecutan en diferentes nodos distribuidos tienen dependencias de datos con los otros procesos y se deben comunicar muy a menudo mediante paso de mensajes.

(15)En inglés, *datacenters*.

(16)En inglés, *tightly coupled*.

2.2.3. Coherencia de memoria caché

Hay que recordar que las memorias caché las gestionan sistemas hardware, por lo tanto, los programadores no tienen control directo sobre ellas. Esto tiene consecuencias importantes para los sistemas de memoria compartida. Para entender esto, suponemos que tenemos un sistema de memoria compartida con dos núcleos, cada uno de los cuales tiene su propia memoria caché de datos. No habría ningún problema mientras que los dos núcleos solo lean datos compartidos.

Por ejemplo, supongamos dos núcleos compartiendo memoria y que x es una variable compartida que se ha inicializado en 2, $y0$ es privada y propiedad del núcleo 0, y $y1$ y $z1$ son privadas y propiedad del núcleo 1. Ahora supongamos el siguiente escenario que muestra la tabla 3.

Tabla 3. Escenario ilustrativo de coherencia de memoria caché

Tiempo	Núcleo 0	Núcleo 1
0	$y0 = x;$	$y1 = 3 * x;$
1	$x = 7;$	Código que no involucra x
2	Código que no involucra x	$z1 = 4 * x;$

Entonces, la posición de memoria de $y0$ finalmente tomará el valor 2 y la posición de memoria de $y1$ tomará el valor 6. En cambio, no está claro qué valor que tomará $z1$. Inicialmente podría parecer que como el núcleo 0 actualiza x a 7 antes de la asignación a $z1$, $z1$ tendrá el valor 4 * 7 = 28. En cambio, en el instante de tiempo 0, x está en la memoria caché del núcleo 1. Por lo tanto, a no ser que por alguna razón x sea desalojada de la memoria caché del núcleo 0 y después realojada en la memoria caché del núcleo 1, en realidad el valor original $x = 2$ podría ser utilizado y $z1$ tomará el valor 4 * 2 = 8.

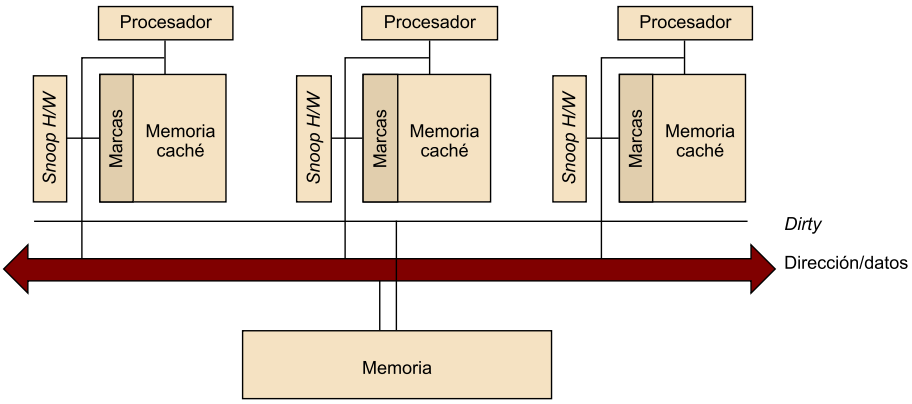
Claramente esto es un problema importante y el programador no tiene por qué tener control directo de cuándo las memorias caché son actualizadas y, por lo tanto, su programa no podrá suponer en el caso del ejemplo anterior

qué valor tendrá *z*1. Aquí hay varios problemas, pero lo que se quiere destacar es el hecho de que las memorias caché de sistemas de un único procesador no tienen mecanismos para asegurar que, cuando las memorias caché de múltiples procesadores guardan la misma variable, una modificación hecha por uno de los procesadores a la variable en memoria caché sea vista por el resto de los procesadores. Esto quiere decir que los valores en memoria caché en otros procesadores sean también actualizados y, por lo tanto, hablamos del problema de coherencia de memoria caché.

Técnica de coherencia de memoria caché *snooping*

Hay dos alternativas básicas para asegurar coherencia en la memoria caché: la técnica de *snooping* y la basada en directorio. La idea de la técnica *snooping* viene de los sistemas basados en bus: cuando los núcleos comparten un bus, cualquier señal transmitida por el bus puede ser consultada por todos los núcleos conectados al bus. Por lo tanto, cuando el núcleo 0 actualiza la copia de *x* que hay en su memoria caché, si también se distribuye esta información por el bus y el núcleo 1 está escuchándolo, este podrá ver que *x* se ha actualizado y podrá marcar su propia copia de *x* como inválida. Así es básicamente cómo funciona la coherencia por *snooping*, a pesar de que en la implementación real cuando se distribuye la información de algún cambio se hace informando a los otros núcleos de que la línea de memoria caché que contiene *x* se ha actualizado y no de que *x* se ha actualizado. La figura siguiente muestra un esquema del sistema de coherencia por *snooping*.

Figura 13. Sistema sencillo de coherencia de memoria por *snooping*



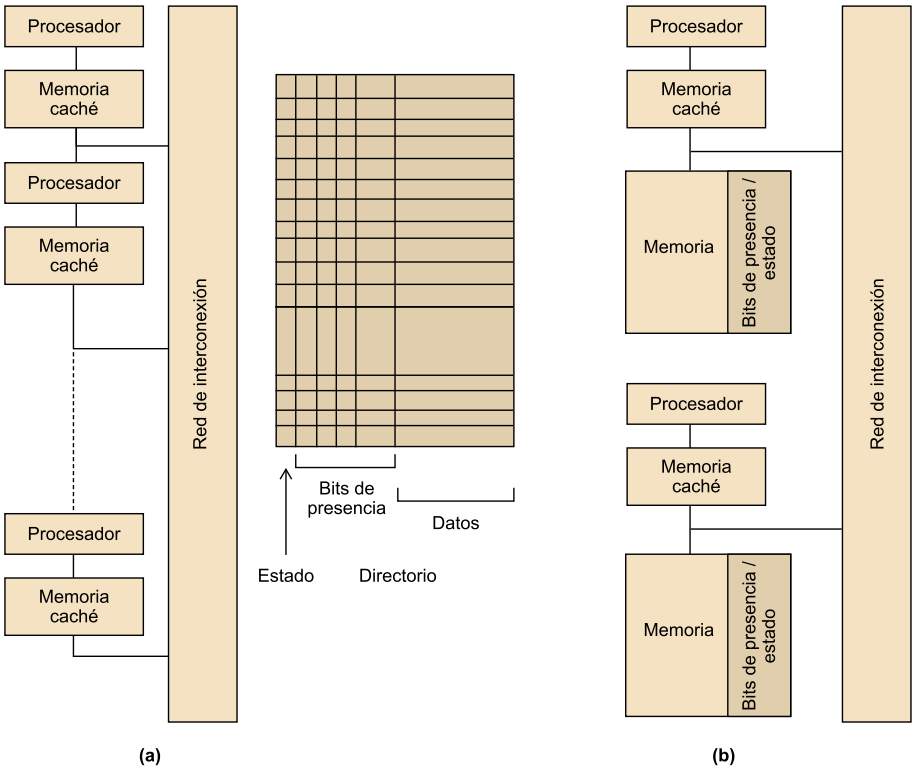
Técnica de coherencia de memoria basada en directorio

En redes de interconexión grandes utilizar la técnica de *snooping* no es viable, puesto que hay que distribuir datos por la red cada vez que hay un cambio y esto claramente no es escalable, ya que el rendimiento se degradaría mucho.

Los protocolos de memoria caché basados en directorio intentan resolver este problema mediante una estructura de datos que se denomina directorio. El directorio almacena el estado de cada línea de memoria caché. Típicamente, esta estructura de datos es distribuida, así que cada par de procesador/memoria es responsable de almacenar la parte de la estructura correspondiente al estado

de las líneas de memoria caché en su memoria local. Cuando una variable se actualiza, se consulta el directorio y los controladores de memoria caché de los núcleos que tienen la línea de memoria caché donde aparece esta variable en su memoria caché son invalidados.

Figura 14. Sistemas de coherencia de memoria basados en directorio: (a) con un directorio centralizado y (b) con directorio distribuido



Con esta técnica claramente se necesita más espacio de memoria para el directorio, pero cuando una variable en memoria caché se actualiza, solo los núcleos que almacenan esta variable necesitan ser contactados y, por lo tanto, se reducen las comunicaciones por las redes y en consecuencia el rendimiento mejora respecto a la técnica de *snooping*.

Falsa compartición

Es importante recordar que las memorias caché de los procesadores están implementadas en hardware, por lo tanto, operan sobre líneas de memoria caché y no sobre variables individuales. Esto puede tener consecuencias catastróficas en relación con el rendimiento. Como ejemplo, supongamos que queremos llamar repetidamente a una función `f(i,j)` y añadir el valor devuelto dentro de un vector tal y como muestra el código siguiente:

```
int i, j, m, n;
double y[m];
/* Asignar y = 0 */
...
for(i=0; i<m; i++)
   for(j=0; j<n; j++)
```

```
        y[i] += f(i,j);
```

Podemos paralelizarlo dividiendo las iteraciones del bucle exterior entre los diferentes núcleos. Si tenemos `num_cores` núcleos, podemos asignar las primeras `m/num_cores` iteraciones al primer núcleo, las siguientes `m/num_cores` iteraciones al segundo núcleo, y así sucesivamente, tal y como se muestra en el siguiente código:

```
/* Variables privadas*/
int i, j, num_iter;
/* Variables privadas inicializadas por un núcleo */
int m, n, num_cores;
double y[m];
num_iter = m/num_cores;
/* El núcleo 0 hace lo siguiente */
for(i=0; i<num_iter; i++)
   for(j=0; j<n; j++)
      y[i] += f(i,j);
/* El núcleo 1 hace lo siguiente */
for(i=num_iter+1; i<2*num_iter; i++)
   for(j=0; j<n; j++)
      y[i] += f(i,j)
...
double y[m];
/* Assignar y = 0 */
...
```

Supongamos que tenemos un sistema de memoria compartida que tiene dos núcleos, `m = 8`, el tipo *double* es de 8 bytes, las líneas de memoria caché son de 64 bytes e y[0] está almacenado al principio de una línea de memoria caché. Una línea de memoria caché puede almacenar ocho *doubles* e `y` ocupa una línea de memoria caché completa. Si los núcleos 0 y 1 ejecutan simultáneamente sus códigos, como todo el vector `y` está almacenado en una única línea de memoria caché, cada vez que uno de los núcleos ejecuta la línea `y[i] += f(i,j)`, la línea será invalidada y la próxima vez que el otro núcleo intente ejecutar esta línea deberá tomar la línea otra vez de memoria principal. Por lo tanto, si se realizan muchas `(n)` iteraciones podemos suponer que un gran porcentaje de veces que se ejecuta `y[i] += f(i,j)` se accederá a memoria principal a pesar de que tanto el núcleo 0 como el núcleo 1 nunca acceden a los elementos de `y` que le corresponden al otro núcleo. Esto es la llamada falsa compartición[17] puesto que el sistema se comporta como si los elementos de `y` estuvieran siendo compartidos por los núcleos.

[17] En inglés, *false sharing*.

Hay que remarcar que la falsa compartición no causa resultados incorrectos, sino que puede arruinar el rendimiento de un programa porque puede ser que acceda a la memoria principal muchas más veces de lo necesario. Podemos mitigar este efecto utilizando almacenamiento temporal que sea local al flujo

o proceso y después copiando el almacenamiento temporal en el almacenamiento compartido. También se pueden aplicar altas técnicas, como por ejemplo *padding*.

Actividad

Os proponemos que busquéis información sobre la técnica de *padding*.

2.3. Sistemas distribuidos

A pesar de que el modelo de computación de altas prestaciones es tradicionalmente el que hemos visto en las arquitecturas paralelas, en la última década se han desarrollado sistemas distribuidos que han aparecido como una fuente viable para ciertas aplicaciones de computación de altas prestaciones, como son los sistemas *grid*.

Los **sistemas *grid*** proporcionan la infraestructura necesaria para transformar redes de computadores distribuidos geográficamente a través de Internet en un sistema unificado de memoria distribuida.

En general, estos sistemas son muy heterogéneos, puesto que individualmente los nodos que los componen pueden ser de diferentes tipos de hardware e incluso de diferentes tipos de software. Aun así, mediante una capa intermedia[18] de software, la infraestructura *grid* proporciona las interfaces y abstracciones necesarias para trabajar con computadores distribuidos y de diferentes instituciones de manera transparente como si fuera un sistema convencional.

[18] En inglés, *middleware*.

También se han desarrollado otros tipos de sistemas altamente distribuidos, que inicialmente no estaban diseñados para la computación de altas prestaciones, como el *peer-to-peer* o el *cloud computing*.

Ved también

Los sistemas *peer-to-peer* y *cloudcomputing*, justamente con la computación en *grid*, se tratarán en el cuarto módulo de esta asignatura y también se tratará su relación con la computación de altas prestaciones.

3. Programación de aplicaciones paralelas

La mayoría de los programas que se han desarrollado para sistemas convencionales con un único núcleo no pueden explotar los múltiples núcleos de los procesadores actuales. Podemos ejecutar diferentes instancias de un programa, por ejemplo dejando que el sistema operativo las planifique en los diferentes procesadores, pero normalmente esta solución no es suficiente, especialmente en la computación de altas prestaciones, donde se requiere paralelismo masivo. Las principales soluciones son o bien reescribir los programas o utilizar herramientas que paralelicen automáticamente los programas. Desgraciadamente, todavía no se han podido encontrar soluciones aceptables para esta segunda opción.

La manera de escribir programas paralelos depende del modo como se quiera dividir el trabajo. Hay dos tipos principales: paralelismo a nivel de tareas y paralelismo a nivel de datos.

En el **paralelismo a nivel de tareas** el trabajo se reparte en varias tareas que se distribuyen por los diferentes *cores*. En el **paralelismo a nivel de datos** los datos se dividen para resolver el problema entre los diferentes núcleos, que realizan operaciones similares sobre los datos que les corresponden.

Paralelismo a nivel de datos

Un ejemplo de paralelismo a nivel de datos es la computación gráfica o basada en GPU.

Actualmente los programas paralelos más potentes se escriben utilizando construcciones de paralelismo explícito utilizando extensiones de lenguajes, como por ejemplo C/C++. Estos programas incluyen instrucciones que son específicas para gestionar el paralelismo –por ejemplo: el núcleo 0 ejecuta la tarea 0, el núcleo 1 ejecuta la tarea 1, etc.–, sincronizar todos los núcleos, etc. y, por lo tanto, los programas devienen muchas veces muy complejos. Hay otras opciones para escribir programas paralelos, como por ejemplo utilizar lenguajes de más alto nivel, pero estos suelen sacrificar rendimiento para facilitar el desarrollo y mejorar la productividad.

Nos centraremos en los programas que son explícitamente paralelos. Los principales modelos de programación son: paso de mensajes o MPI[19], flujos (por ejemplo, flujos POSIX o Pthreads), y OpenMP. MPI y Pthreads son librerías con definiciones de tipos, funciones y macros que se pueden utilizar por ejemplo en programas escritos en C. OpenMP consiste en una librería y también ciertas modificaciones del compilador, por ejemplo de C. Adicionalmente también trataremos otros modelos de programación, como los de computación gráfica u otros que proporcionan abstracciones más modernas.

(19)Del inglés *message passing interface*.

Estos modelos de programación dan respuesta a los dos tipos principales de sistemas paralelos: sistemas de memoria compartida y sistemas de memoria distribuida. En un sistema de memoria compartida los núcleos pueden compartir el acceso a la memoria, por lo que hay que coordinar el acceso de los diferentes núcleos a memoria teniendo que examinar y actualizar el espacio compartido de la memoria. En un sistema de memoria distribuida cada núcleo tiene su propio espacio de memoria privado y los núcleos se deben comunicar explícitamente haciendo paso de mensajes por la red de interconexión. Pthreads y OpenMP se diseñaron para programar sistemas de memoria compartida y proporcionan mecanismos para acceder a posiciones de memoria compartida. En cambio, MPI se diseñó para programar sistemas de memoria distribuida y proporciona mecanismos para paso de mensajes.

3.1. Modelos de programación de memoria compartida

3.1.1. Programación con flujos

A pesar de que se pueden considerar diferentes tipos de flujos para programar programas paralelos, como por ejemplo flujos propios de UNIX o de Java, nos centraremos en los Pthreads, que son una librería que cumple los estándares POSIX y que nos permiten trabajar con diferentes flujos al mismo tiempo (concurrentemente). Pthreads son más efectivos en un sistema multiprocesador o en un sistema multinúcleo, donde el flujo de ejecución se puede planificar en otro procesador o núcleo, con lo que se gana velocidad debido al paralelismo.

Los Pthreads tienen menos sobrecoste que el *fork* o creación de un nuevo proceso porque el sistema no inicializa un nuevo espacio de memoria virtual y entorno para el proceso. A pesar de que son más efectivos en sistemas con múltiples procesadores o núcleos, también se puede obtener beneficio en un sistema uniprocesador, puesto que permite explotar la latencia de la entrada/salida y otras funciones del sistema que pueden parar la ejecución del proceso en ejecución. Los Pthreads se utilizan en sistemas de memoria compartida y todos los flujos de un proceso comparten el mismo espacio de direcciones. Para crear un flujo hay que definir una función y sus argumentos, que serán utilizados por el flujo. El objetivo principal de utilizar Pthreads es conseguir más velocidad (mejor rendimiento).

El siguiente código muestra un ejemplo donde se puede apreciar cómo crear dos flujos (cada uno con diferentes parámetros) y esperar su finalización.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
main() {
   pthread_t thread1, thread2;
```

```
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int iret1, iret2;
    /* Crea flujos independientes, que ejecutarán la función */
    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
    /* Espera a que todos los flujos acaben antes de continuar con el programa principal */
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}
void *print_message_function( void *ptr ) {
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Uno de los elementos clave en la gestión de flujos Pthread es su sincronización y exclusión mutua, que habitualmente se lleva a cabo mediante semáforos.

3.1.2. OpenMP

A pesar de que tanto Pthreads como OpenMP son interfaces de programación para memoria compartida, tienen muchas diferencias importantes. Con Pthreads se requiere que el programador especifique explícitamente el comportamiento de cada flujo. Por otro lado, OpenMP permite que el programador solo tenga que especificar que un bloque de código se ejecute en paralelo, y determinar las tareas y qué flujo las ejecutará se deja en manos del compilador y del sistema de ejecución[20]. Esto sugiere otra diferencia importante con Pthreads: mientras que Pthreads es una librería de funciones que se puede añadir al montar un programa en C, OpenMP necesita soporte a nivel de compilador y, por lo tanto, no necesariamente todos los compiladores de C han de compilar programas OpenMP como programas paralelos. Por lo tanto, podemos decir que OpenMP permite mejorar la productividad mediante una interfaz de más alto nivel que la de Pthreads, pero por otro lado no nos permite controlar ciertos detalles relacionados con el comportamiento de los flujos.

[20] En inglés, *runtime*.

OpenMP fue concebido por un grupo de programadores y científicos informáticos que pensaban que escribir programas de altas prestaciones a gran escala utilizando interfaces como la de Pthreads era demasiado difícil. De hecho, OpenMP se diseñó explícitamente para permitir a los programadores paralelizar programas secuenciales existentes progresivamente y no tener que volver a escribirlos desde cero.

OpenMP proporciona una interfaz de memoria compartida basada en las llamadas "directivas". Por ejemplo en C/C++ esto quiere decir que hay ciertas instrucciones especiales para el preprocesador denominadas *pragmes*. Normalmente se añaden *pragmes* en un programa para especificar comportamientos que no son parte de la propia especificación del lenguaje de programación. Esto significa que los compiladores que no entienden estos *pragmes* los pueden ignorar, lo que permite que un programa pueda ejecutarse en un sistema que no los soporta.

Los *pragmes* en C/C++ empiezan por `#pragma`, al igual que otras directivas de procesamiento. A continuación se muestra un programa OpenMP muy sencillo que implementa el típico "hello, world". En este, podemos apreciar la utilización de una directiva (o *pragma*) de OpenMP y también algunas funciones típicas asociadas a OpenMP.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Hello(void);
int main(int argc, char* argv[]) {
   /* Obtener el número de flujos del intérprete de órdenes */
   int thread_count = strtol(argv[1], NULL, 10);
#  pragma omp parallel num_threads(thread_count)
   Hello();
   return 0;
}
void Hello(void( {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
```

3.1.3. *CUDA* y OpenCL

CUDA[21] es una especificación inicialmente de propiedad desarrollada por Nvidia como plataforma para sus productos de computación gráfica (GPU). *CUDA* incluye las especificaciones de la arquitectura y un modelo de programación asociado.

(21) *Del* inglés *compute unified device architecture*.

CUDA se desarrolló para aumentar la productividad en el desarrollo de aplicaciones de propósito general para computación gráfica. Desde el punto de vista del programador, el sistema está compuesto por un procesador principal[22], que es una CPU tradicional, como por ejemplo un procesador de arquitectura Intel, y por uno o más dispositivos[23], que son GPU.

(22) En inglés, *host*.

(23) En inglés, *devices*.

CUDA pertenece al modelo *SIMD*; por lo tanto, está pensado para explotar el paralelismo a nivel de datos. Esto quiere decir que un conjunto de operaciones aritméticas se pueden ejecutar sobre un conjunto de datos de manera simultánea. Afortunadamente, muchas aplicaciones tienen partes con un nivel muy elevado de paralelismo a nivel de datos.

Un programa en *CUDA* consiste en una o más fases que pueden ser ejecutadas o bien en el procesador principal (CPU) o bien en el dispositivo GPU. Las fases en las que hay muy poco o nada de paralelismo a nivel de datos se implementan en el código que se ejecutará en el procesador principal, y las fases con un nivel de paralelismo a nivel de datos elevado se implementan en el código que se ejecutará al dispositivo.

Los elementos fundamentales de *CUDA* son:

a) Modelo de memoria. Es importante remarcar que la memoria del procesador principal y del dispositivo son espacios de memoria completamente separados. Esto refleja la realidad de que los dispositivos son típicamente tarjetas que tienen su propia memoria DRAM. Para ejecutar un *kernel* en el dispositivo GPU normalmente hay que seguir los pasos siguientes:

- Reservar memoria en el dispositivo.
- Transferir los datos necesarios desde el procesador principal al espacio de memoria asignado al dispositivo.
- Invocar la ejecución del *kernel* en cuestión.
- Transferir los datos con los resultados desde el dispositivo hacia el procesador principal.
- Liberar la memoria del dispositivo (si ya no es necesaria), una vez finalizada la ejecución del *kernel*.

b) Organización de flujos. En CUDA, un *kernel* se ejecuta mediante un conjunto de flujos (por ejemplo, un vector o una matriz de flujos). Como todos los flujos ejecutan el mismo *kernel* (modelo SIMT) se necesita un mecanismo que permita diferenciarlos, y así poder asignar la parte correspondiente de los datos a cada flujo de ejecución. CUDA incorpora palabras clave para hacer referencia al índice de un flujo.

c) *Kernels*. El código que se ejecuta en el dispositivo (*kernel*) es la función que ejecutan los diferentes flujos durante la fase paralela, cada uno en el rango de datos que le corresponde. Cabe señalar que CUDA sigue el modelo SPMD[24] y, por lo tanto, todos los flujos ejecutan el mismo código.

[24] Del inglés *single program, multiple data*.

A continuación se muestra la función o *kernel* de la suma de matrices y su llamada.

```
__global__ matrix_add_gpu (float *A, float *B, float *C, int N)
{
```

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i<N && j<N){
       C[index] = A[index] + B[index];
    }
}
int main(){
    dim3 dimBlock(blocksize, blocksize);
    dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
    matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

OpenCL es una interfaz estándar, abierta, libre y multiplataforma para la programación paralela. La principal motivación para el desarrollo de OpenCL fue la necesidad de simplificar la tarea de programación portable y eficiente de la creciente cantidad de plataformas heterogéneas, como por ejemplo, CPU multinúcleo, GPU o incluso sistemas empotrados. OpenCL fue concebida por Apple, a pesar de que la acabó desarrollando el grupo Khronos, que es el mismo que impulsó OpenGL y es su responsable.

OpenCL consiste en tres partes: la especificación de un lenguaje multiplataforma, una interfaz a escala de entorno de computación y una interfaz para coordinar la computación paralela entre procesadores heterogéneos. OpenCL utiliza un subconjunto de C99 con extensiones para el paralelismo y utiliza el estándar de representación numérica IEEE 754 para garantizar la interoperabilidad entre plataformas.

Hay muchas similitudes entre OpenCL y *CUDA*, a pesar de que OpenCL tiene un modelo de gestión de recursos más complejo, puesto que soporta múltiples plataformas y portabilidad entre diferentes fabricantes. OpenCL soporta modelos de paralelismo tanto a nivel de datos como nivel de tareas.

Del mismo modo que en *CUDA*, un programa en OpenCL está formado por dos partes: los *kernels* que se ejecutan en uno o más dispositivos y un programa en el procesador principal que invoca y controla la ejecución de los *kernels*. Cuando se efectúa la invocación de un *kernel*, el código se ejecuta en tareas elementales[25] que corresponden a los flujos de *CUDA*. Las tareas elementales y los datos asociados a cada tarea elemental se definen a partir del rango de un espacio de índices de dimensión N[26]. Las tareas elementales forman grupos de tareas[27], que corresponden a los bloques de *CUDA*. Las tareas elementales tienen un identificador global que es único. Además, los grupos de tareas elementales se identifican dentro del rango de dimensión N y, para cada grupo, cada una de las tareas elementales tiene un identificador local, que irá desde 0

(25)En inglés, *work items*.

(26)En inglés, *ND ranges*.

(27)En inglés, *work groups*.

hasta la medida del grupo-1. Por lo tanto, la combinación del identificador del grupo y del identificador local dentro del grupo también identifica de manera única una tarea elemental.

OpenCL también tiene su propio modelo de memoria y de gestión de *kernels* y de dispositivos. A continuación se muestra la función o *kernel* de la suma de matrices en OpenCL.

```
__kernel void matrix_add_opencl ( __global const float *A,
                  __global const float *B,
                  __global float *C,
                  int N) {
   int i = get_global_id(0);
   int j = get_global_id(1);
   int index = i + j*N;
   if (i<N && j<N){
      C[index] = A[index] + B[index];
   }
}
```

3.2. Modelos de programación de memoria distribuida

3.2.1. *MPI*

MPI es una librería de paso de mensajes. Se puede utilizar básicamente en programas C o Fortran, desde los cuales se hacen llamadas a funciones de *MPI* para la gestión de procesos y para comunicar los procesos entre sí.

MPI no es la única librería disponible de paso de mensajes, pero puede considerarse el estándar actual para el paradigma de memoria distribuida. Anteriormente a *MPI* hubo otros modelos, como por ejemplo *PVM*[28], pero *MPI* se acabó imponiendo.

[28] Del inglés *parallel virtual machine*.

Esta especificación fue desarrollada por el *MPI* Forum, una agrupación de universidades y empresas que especificaron las funciones que debería tener una librería de paso de mensajes. A partir de la especificación los diferentes fabricantes de multicomputadores incluyeron implementaciones de *MPI* específicas para sus equipos y aparecieron varias implementaciones libres, de que las más extendidas son MPICH y OpenMPI (que es una distribución de código libre de MPI2).

Resumiendo, *MPI* ha conseguido una serie de hitos, entre los que podemos encontrar los siguientes:

Altas prestaciones

IBM, por ejemplo, tiene su propia librería de *MPI* para sus sistemas BlueGene.

- **Estandarización.** Las implementaciones de la especificación han hecho que se convirtiera en el estándar de paso de mensajes y que no sea necesario desarrollar programas diferentes para máquinas diferentes.

- **Portabilidad.** Los programas *MPI* funcionan sobre multiprocesadores de memoria compartida, multicomputadores de memoria distribuida, clústeres de computadores, sistemas heterogéneos, etc. siempre que haya una versión de *MPI* para ellos.

- **Altas prestaciones.** Los fabricantes han desarrollado implementaciones eficientes para sus equipos.

- **Amplia funcionalidad.** *MPI* incluye gran cantidad de funciones para llevar a cabo de manera sencilla las operaciones que suelen aparecer en programas de paso de mensajes.

Cuando se pone en marcha un programa *MPI* se crean varios procesos, ejecutando el mismo código, cada uno con sus propias variables (modelo SPMD). A diferencia de OpenMP, no hay un proceso maestro que controla al resto (como en OpenMP).

A continuación se muestra un programa que implementa el "hello, world" en *MPI*.

```
#include <stdio.h>
#include <mpi.h>
int main ( int argc, char *argv[])
{
   int rank, size;
   MPI_Init (&argc, &argv); /* starts MPI */
   MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* Obtiene el identificador del proceso */
   MPI_Comm_size (MPI_COMM_WORLD, &size); /* Obtiene el número de procesos */
   printf( "Hello world from process %d of %d\n", rank, size );
   MPI_Finalize();
   return 0;
}
```

A pesar de que se trata de un código muy simple, se pueden observar algunos componentes esenciales de *MPI*:

- Hay que incluir la librería de *MPI* (`mpi.h`).

- Todos los procesos ejecutan el mismo código desde el principio, con lo que todos tienen variables `myrank` y `size`. A diferencia de OpenMP, estas

variables son diferentes y pueden estar en memorias diferentes en procesadores distintos.

- Los procesos trabajan de manera independiente hasta que se inicializa *MPI* con la función `MPI_Init`. A partir de este punto, los procesos pueden colaborar intercambiando datos, sincronizándose, etc.

- La función `MPI_Finalize` se llama cuando ya no es necesario que los procesos colaboren entre sí. Esta función libera todos los recursos reservados por *MPI*.

- Las funciones *MPI* tienen la forma `MPI_Nombre(parámetros)`. De la misma manera que en OpenMP, es necesario que los procesos sepan su identificador (entre 0 y el número de procesos menos 1) y el número de procesos que se han puesto en marcha. Las funciones `MPI_Comm_rank` y `MPI_Comm_size` se utilizan para conseguir esto.

- Vemos que estas funciones tienen un parámetro `MPI_COMM_WORLD` que es una constante *MPI* y que identifica el comunicador al que están asociados todos los procesos. Un comunicador es un identificador de un grupo de procesos, y las funciones *MPI* han de identificar en qué comunicador se están realizando las operaciones que se llaman.

Tal y como veremos en otro módulo de esta asignatura, *MPI* ofrece una amplia interfaz donde podemos encontrar operaciones tanto punto a punto (por ejemplo, para enviar una fecha de un proceso a otro) como colectivas (por ejemplo, para sincronizar todos los procesos en un punto concreto o reducir un conjunto de datos que están distribuidos entre los diferentes procesos del programa *MPI*).

Ved también

La interfaz de *MPI* se trata en el tercer módulo de esta asignatura.

3.2.2. *PGAS*

Muchos programadores encuentran los modelos de programación de memoria compartida mucho más atractivos y prácticos que los de paso de mensajes. Para dar respuesta a esto, diferentes grupos están desarrollando lenguajes de programación paralela, que permiten utilizar algunas técnicas de memoria compartida para programar sistemas que realmente son de memoria distribuida. Esto no es tan simple como parece, puesto que si, por ejemplo, escribimos un compilador que gestione un conjunto de memorias distribuidas como si fueran una compartida, nuestros programas tendrían un rendimiento extraordinariamente bajo y difícil de predecir, ya que no está claro cuándo se accede a la memoria local o distribuida. Acceder a la memoria de otro nodo puede ser centenares o incluso miles de veces más lento que acceder a la memoria local.

Los lenguajes *PGAS*[29] proporcionan algunos de los mecanismos de los programas de memoria compartida, pero también proporcionan herramientas al programador para poder controlar la localidad de los datos dentro del espacio

(29)Del inglés *partitioned global address space*.

(abstracto) de memoria compartida. Las variables privadas se alojan en la memoria local del núcleo en el que el proceso se está ejecutando, y el programador puede controlar la distribución de los datos en estructuras de datos compartidos. De este modo, por ejemplo, el programador puede saber qué elementos de un vector compartido están en la memoria local de cada proceso.

Hay varias implementaciones de lenguajes *PGAS*, las más importantes de las cuales son *UPC*[30], Co-Array Fortran y Titanium, que extienden C, Fortran y Java, respectivamente.

[30] Del inglés *unified parallel C*.

3.3. Modelos de programación híbridos

Los modelos de programación vistos anteriormente, como OpenMP y MPI, dan soluciones concretas a sistemas de memoria compartida y distribuida, respectivamente. En cambio, los actuales sistemas de altas prestaciones combinan sistemas de memoria distribuida (básicamente clústeres) con sistemas de memoria compartida (procesadores o incluso multiprocesadores multinúcleo).

Así pues, también se utilizan modelos de programación híbridos, como por ejemplo MPI+OpenMP, para aplicaciones que tienen más de un nivel de paralelismo (multinivel). A pesar de que hay diferentes maneras de combinar dos tipos de modelos de programación, en estos casos se suele utilizar *MPI* para los bucles más exteriores y OpenMP para los bucles internos, aprovechando así la localidad de los datos.

A continuación se muestra un programa híbrido MPI+OpenMP muy sencillo que implementa una versión del "hello, world" visto anteriormente. En este ejemplo no hay realmente paso de mensaje pero ilustra el modo de consultar más de un nivel de paralelismo.

```
#include <stdio.h>
#include "mpi.h"
#include <omp.h>
int main(int argc, char *argv[]) {
   int numprocs, rank, namelen;
   char processor_name[MPI_MAX_PROCESSOR_NAME];
   int iam = 0, np = 1;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(processor_name, &namelen);
   #pragma omp parallel default(shared) private(iam, np)
   {
      np = omp_get_num_threads();
      iam = omp_get_thread_num();
      printf("Hello from thread %d out of %d from process %d out of %d on %s\n",
         iam, np, rank, numprocs, processor_name);
```

```
    }
    MPI_Finalize();
}
```

Otra motivación de los modelos de programación híbridos es que los diferentes modelos de programación tienen sus ventajas e inconvenientes y la combinación de más de un modelo de programación puede aprovechar sus ventajas y dejar de lado sus inconvenientes. Podemos comparar diferentes aspectos de *MPI* y OpenMp, los dos paradigmas más habituales:

- **Grano de paralelismo.** OpenMP es más apropiado para paralelismo de grano más fino, es decir, cuando hay poca computación entre sincronizaciones entre procesos.

- **Arquitectura de ejecución.** OpenMP funciona bien en sistemas de memoria compartida, pero en sistemas distribuidos es preferible MPI.

- **Dificultad de programación.** En general, los programas OpenMP son más parecidos a los secuenciales que los *MPI* y es más fácil desarrollar programas OpenMP.

- **Herramientas de depuración.** Se dispone de más herramientas de desarrollo, depuración, autoparalelización, etc. para memoria compartida que para paso de mensajes.

- **Creación de procesos.** En OpenMP los flujos se crean dinámicamente, mientras que en *MPI* los procesos se crean estáticamente, a pesar de que algunas versiones de *MPI* permiten la gestión dinámica.

- **Uso de la memoria.** El acceso a la memoria es más sencillo con OpenMP por el hecho de utilizar memoria compartida. Aun así la localización de los datos en la memoria y el acceso simultáneo puede reducir el rendimiento. La utilización de memorias locales con *MPI* puede producir en accesos más rápidos.

Así pues, algunas de las ventajas de la programación híbrida son:

- Puede ayudar a mejorar la escalabilidad y a mejorar el balanceo de las tareas.

- Se puede utilizar en aplicaciones que combinan paralelismo de grano fino y grueso o que mezclan paralelismo funcional y de datos.

- Puede utilizarse para reducir el coste de desarrollo, por ejemplo, si se tienen funciones que utilizan OpenMP y se utilizan dentro de programas MPI.

Aun así, también hay que tener en cuenta que utilizar dos paradigmas de manera híbrida puede complicar su programación.

Finalmente, los modelos híbridos también son una solución para determinadas funcionalidades que no se suelen dar con un único modelo de programación. Un ejemplo de ello es el modelo de programación *MPI+CUDA*. Mientras que *CUDA* permite aprovechar el gran potencial de cómputo de los procesadores gráficos, *MPI* permite distribuir las tareas en varios nodos y así poder usar simultáneamente más aceleradores gráficos de los que puede tener un único nodo físicamente.

Ved también

En el tercer módulo se tratarán los diferentes modelos de programación brevemente introducidos en este subapartado.

4. Rendimiento de aplicaciones paralelas

Cuando escribimos programas paralelos normalmente podremos apreciar una mejora en el rendimiento, puesto que en muchos casos se puede sacar provecho de disponer de más núcleos. En cambio, escribir un programa paralelo es un arte, pues hay muchas formas de hacerlo, ya que no hay una norma general. Así pues, hay que saber qué es lo que podemos esperar de nuestra implementación y necesitaremos poder evaluarla para saber si se puede mejorar su rendimiento y cómo hacerlo. A continuación veremos las principales métricas utilizadas para determinar el rendimiento de un programa paralelo, veremos los principales estándares en relación con la computación de altas prestaciones y comprobaremos que existen herramientas muy convenientes que permiten analizar el rendimiento de nuestras aplicaciones de tal modo que podemos mejorar nuestra productividad al desarrollar programas paralelos.

4.1. *Speedup* y eficiencia

Normalmente el mejor modo de escribir nuestros programas paralelos es dividiendo el trabajo entre los diferentes núcleos a partes iguales, a la vez que no se introduce ningún trabajo adicional para los núcleos. Si realmente podemos conseguir ejecutar el programa en p núcleos mediante un flujo o proceso en cada uno de los núcleos, entonces nuestro programa paralelo irá p veces más rápido que el programa secuencial. Si denominamos el tiempo de ejecución secuencial T_{seq} y el tiempo de ejecución de nuestro programa paralelo T_{par}, entonces el límite que podemos esperar es $T_{par} = T_{seq}/p$. Cuando esto sucede decimos que el programa paralelo tiene *speedup* lineal.

En la práctica es muy difícil poder obtener *speedup* lineal porque solo por el hecho de utilizar múltiples procesos/flujos se añade un sobrecoste[31].

(31) En inglés, *overhead*.

Por ejemplo, en programas con memoria compartida hay que añadir mecanismos de exclusión mutua, como por ejemplo, semáforos, y en programas de memoria distribuida en algún momento habrá que transmitir datos por la red para implementar el paso de mensajes. Además, hay que tener en cuenta que los costes adicionales aumentarán a medida que se incremente el número de procesos o flujos.

Así pues, si definimos el *speedup* de un programa paralelo de la siguiente forma:

$$S = \frac{T_{seq}}{T_{par}}$$

entonces el *speedup* lineal tiene $S = p$, que no es nada habitual. Además, a medida que p aumente, hemos de esperar que S sea una fracción cada vez más pequeña del *speedup* lineal p, que es el ideal. Otra manera de verlo es que S/p será probablemente cada vez más pequeño, puesto que p aumenta. La tabla

4 muestra un ejemplo de cómo S y S/p cambian a medida que p aumenta. Este valor S/p se denomina *eficiencia del programa paralelo*. Si sustituimos la fórmula por S, vemos que la eficiencia es:

$$E = \frac{S}{p} = \frac{\left(\frac{T_{seq}}{T_{par}}\right)}{p} \frac{T_{seq}}{p \cdot T_{par}}$$

Tabla 4. *Speedups* y eficiencias de un programa paralelo

p	1	2	4	8	16
S	1,0	1,9	3,6	6,5	10,8
$E = S/p$	1,0	0,95	0,90	0,81	0,68

Es evidente que T_{par}, S y E dependen del número de procesos o flujos (p). También hay que tener presente que T_{par}, S, E y T_{seq} dependen del tamaño del problema.

Por ejemplo, si ejecutamos la aplicación con la que se obtuvo la tabla 4 pero con la mitad y el doble del tamaño del problema, podríamos obtener los *speedups* y las eficiencias que se muestran en las figuras 15 y 16, respectivamente.

Figura 15. *Speedups* del programa paralelo con diferente tamaño del problema

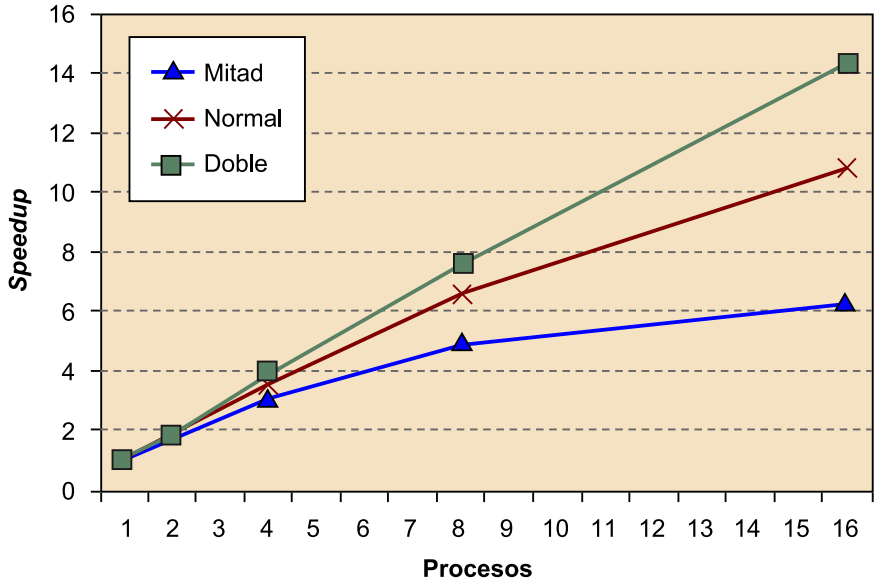
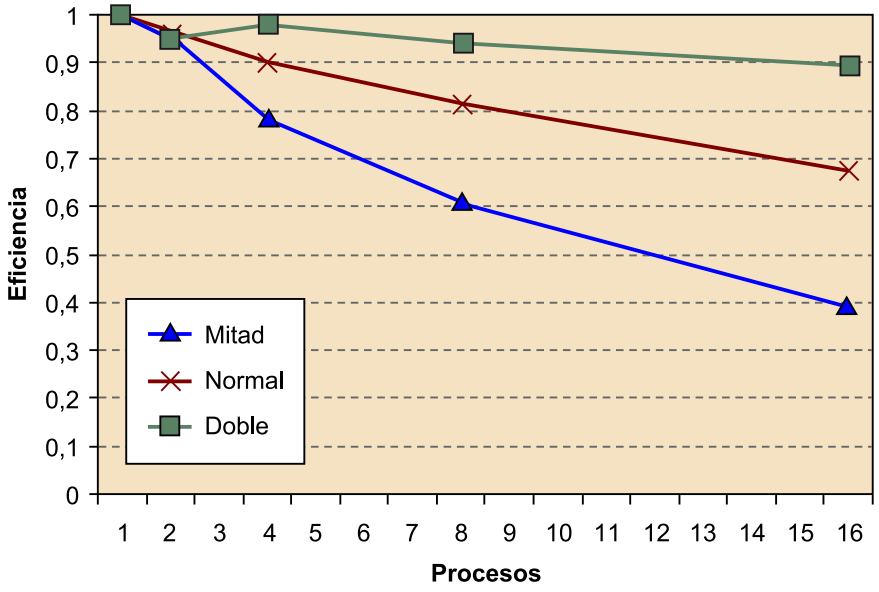


Figura 16. Eficiencias del programa paralelo con diferente tamaño del problema



Tal como vemos en este ejemplo, cuando el tamaño del problema es más grande, los *speedups* y las eficiencias aumentan, mientras que disminuyen cuando el tamaño del problema es más pequeño. Este comportamiento es muy habitual.

Muchos programas paralelos se desarrollan dividiendo el trabajo de la versión secuencial entre los diferentes procesos/flujos y añadiendo el sobrecoste necesario para manejar el paralelismo, como por ejemplo por la exclusión mutua o por las comunicaciones. Así pues, si denominamos este sobrecoste del paralelismo $T_{overhead}$, tenemos que:

$$T_{par} = \frac{T_{seq}}{p} + T_{overhead}$$

También hay que tener en cuenta que habitualmente, a medida que el tamaño del problema aumenta, $T_{overhead}$ aumenta más lentamente que T_{seq}. Si esto sucede, tanto el *speedup* como la eficiencia aumentarán. En otras palabras, si hay más trabajo para realizar por parte de los diferentes procesos/flujos, la proporción de tiempo destinada a coordinar las tareas será inferior a si el tamaño del problema es más pequeño.

4.2. Ley de Amdahl

La ley de Amdahl (Gene Amdahl, 1967) dice que a no ser que un programa secuencial pudiera ser completamente paralelizado, el *speedup* que se obtendría será muy limitado, independientemente del número de núcleos disponibles.

Supongamos que hemos podido paralelizar el 90% del código del programa secuencial y que, además, lo hemos paralelizado perfectamente. Según la ley de Amdahl, independientemente del número de núcleos p que utilicemos, el *speedup* de esta parte del programa será p. Si la parte secuencial de la ejecución es T_{seq} = 20 segons, el tiempo de ejecución de la parte paralela será 0,9 × T_{seq}/p = 18/p y el tiempo de ejecución de la parte no paralela será 0,1 × T_{seq} = 2. El tiempo de ejecución de la parte paralela será:

$$T_{par} = 0{,}9 \times \frac{T_{seq}}{p} + 0{,}1 \times T_{seq} = \frac{18}{p} + 2$$

y el *speedup* será:

$$S = \frac{T_{seq}}{0{,}9 \times \frac{T_{seq}}{p} + 0{,}1 \times T_{seq}} = \frac{20}{\frac{18}{p} + 2}$$

Si el número de núcleos p se hiciera más y más grande (p → ∞), entonces 0,9 × T_{seq}/p = 18/p tendería a 0 y, por lo tanto, el tiempo de ejecución total no podría ser más pequeño que 0,1 × T_{seq} = 2. Por lo tanto, el *speedup* debería ser más pequeño que:

$$S \leq \frac{T_{seq}}{0{,}1 \times T_{seq}} = \frac{20}{2} = 10$$

Esto quiere decir que aunque hagamos una paralelización perfecta del 90% del programa, nunca podríamos obtener un *speedup* mejor que 10, aunque dispusiéramos de millones de núcleos.

De modo general, si una fracción r del programa secuencial no se puede paralelizar, entonces la ley de Amdahl dice que no podremos obtener un *speedup* mejor que 1/r. La aplicación de la ley de Amdahl puede parecer muy pesimista, pero también existe la ley de Gustafson, que considera que la parte no paralela de un programa decrece cuando el tamaño del problema aumenta. Por lo tanto, según la ley de Gustafson, cualquier problema suficientemente grande puede ser paralelizado eficientemente y el *speedup* pasa a ser:

$$S = p - r + (p - 1)$$

4.3. Escalabilidad

De modo general, podemos decir que una tecnología es **escalable** si es capaz de manejar un problema de tamaño creciente. Más formalmente, decimos que un programa es escalable si la eficiencia se mantiene constante, independientemente de si el tamaño del problema aumenta.

Hay dos nociones comunes de la escalabilidad. La primera es el escalado fuerte, en el que si se incrementa el número de procesos/flujos, se puede conservar la eficiencia sin la necesidad de incrementar el tamaño del problema. El segundo es el escalado débil, en el que si queremos conservar la eficiencia fija, hay que incrementar el tamaño del problema del mismo modo que se aumenta el número de procesos/flujos.

4.4. Análisis de rendimiento de aplicaciones paralelas

En el momento de programar nuestras aplicaciones paralelas podemos seguir algunos estándares generales o seguir procedimientos de buenas prácticas, pero es difícil saber si nuestra aplicación podrá cumplir los requisitos de rendimiento que deseamos. En el caso de querer mejorar el rendimiento de nuestra aplicación o simplemente si obtenemos un rendimiento que consideramos pobre por la propia experiencia, es muy difícil poder identificar la fuente del problema y encontrar solución sin analizarla detenidamente.

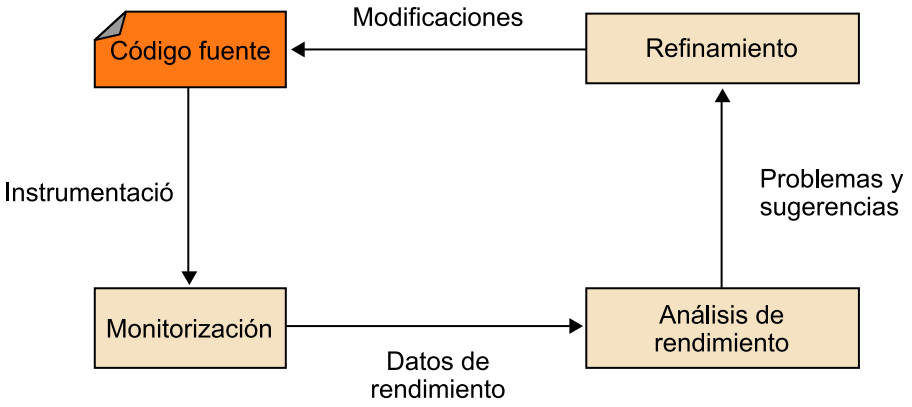
En general, el análisis y la optimización de rendimiento de aplicaciones paralelas es un proceso cíclico, tal y como muestra la figura siguiente.

- La fase de instrumentación y monitorización consiste en añadir a la aplicación una cierta información de instrumentación que permite recopilar información respecto al comportamiento de la aplicación (por ejemplo, número de fallos en la memoria caché de nivel 2).

- La fase de análisis consiste en inspeccionar la información recopilada en la fase anterior para encontrar problemas de rendimiento, deducir las posibles causas y determinar soluciones.

- La fase de refinamiento consiste en aplicar los cambios oportunos al código de la aplicación para poder corregir los posibles problemas de rendimiento identificados.

Figura 17. Proceso cíclico de mejora del rendimiento



Para analizar el rendimiento de aplicaciones paralelas se pueden utilizar diferentes mecanismos, unos más simples y otros más complejos, más generales y más precisos, tal y como veremos en los siguientes subapartados. Hay que remarcar que la utilización de estos mecanismos no es siempre igual para todos los casos y el análisis de rendimiento de aplicaciones paralelas normalmente deviene un arte y se aprende mediante la experimentación.

4.4.1. Medidas de tiempos

Esta es la manera más simple y tradicional de instrumentación y monitorización de aplicaciones, pero en ciertas ocasiones puede ser muy eficaz y rápida. De hecho, hay varias razones para tomar medidas de tiempos, como por ejemplo para determinar si el programa que se está desarrollando se comporta como lo debería hacer o, una vez el programa ya está desarrollado, poder determinar hasta qué punto es bueno su rendimiento.

Hay que tener en cuenta que no siempre será importante conocer el tiempo de ejecución de toda la aplicación, sino que será más importante medir el tiempo de ejecución de ciertas partes del programa. Debido a esto, normalmente no se podrá utilizar la llamada `time` típica del intérprete de órdenes de Unix, la cual devuelve el tiempo de ejecución del programa desde el principio hasta el final.

También hay que tener presente que no siempre nos interesará saber el tiempo de procesador, que es el que devuelve la función estándar de C `clock`. De hecho, se trata del tiempo total que el programa pasa ejecutando código del programa. Esto puede ser un problema, puesto que no incluye el tiempo que el programa está parado.

Programa en espera

Un ejemplo de esto es cuando un programa de memoria distribuida está esperando a que le llegue un mensaje desde otro nodo.

Por lo tanto, para tomar medidas de programas paralelos, como normal general habrá que tener en cuenta el tiempo total de ejecución[32]. A continuación se muestra un código sencillo para medir el tiempo de ejecución de un programa.

[32] También conocido como *wall clock*.

```
Double start, finish;
...
start = get_current_time();
/* Código que queremos medir */
...
finish = get_current_time();
printf("Tiempo transcurrido = %e segundos\n", finish-start);
```

En este ejemplo la función `get_current_time()` es una hipotética función que devuelve los segundos que han transcurrido desde un instante de tiempo prefijado del pasado. La función concreta que deberemos utilizar dependerá de la interfaz de programación.

Ejemplo

Por ejemplo, MPI dispone de la función `MPI_Wtime` y OpenMP dispone de la función `omp_get_wtime`. Ambas devuelven el tiempo total de ejecución y no el tiempo de procesador.

Hay una cuestión muy importante que se debe tener en cuenta a la hora de tomar medidas de tiempos: la resolución de las funciones de medida de tiempo. Siempre será necesario que la resolución, que es el intervalo de tiempo entre dos medidas, sea menor a la medida que queremos tomar, puesto que si no, obtendremos una medida de 0.

La resolución

Hay muchas funciones de medida de tiempo que tienen una resolución de microsegundos (10^{-3}) y muchas veces necesitaremos resoluciones de nanosegundos (10^{-9}) para poder identificar acontecimientos de interés para el análisis.

Finalmente, cabe destacar que cuando se toman medidas de tiempos, se pueden obtener diferentes valores en diferentes ejecuciones. Por tanto, habrá que tener en cuenta esta posible variabilidad de las medidas y cuando se realicen estudios a partir de medidas de tiempos, utilizar valores estadísticos, como por ejemplo el valor medio obtenido de varias ejecuciones. También hay que tener en cuenta que nos podemos encontrar medidas anómalas debido a factores puntuales, como por ejemplo un mal funcionamiento de la red de interconexión.

4.4.2. Interfaces de instrumentación y monitorización

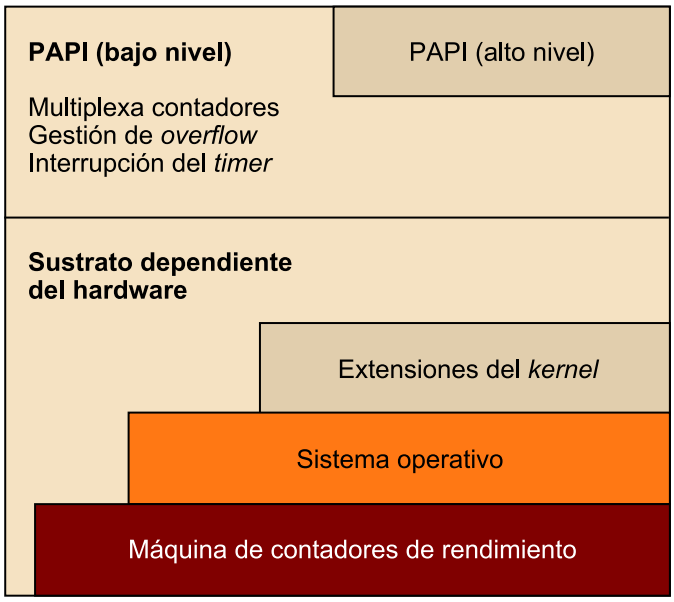
Las herramientas de monitorización normalmente constan de dos partes: una librería o conjunto de librerías que permiten insertar código de instrumentación para medir y almacenar datos, y una serie de módulos que ofrecen la posibilidad de mostrar los datos generados durante la monitorización de la aplicación paralela. Hay que tener en cuenta que la instrumentación puede afectar a las características de rendimiento de la aplicación paralela, puesto que pueden añadir un cierto sobrecoste relacionado con la instrumentación.

A pesar de que hay muchas, una de las herramientas de instrumentación más utilizadas es *PAPI*[33]. *PAPI* es una interfaz para poder acceder a los contadores de hardware relacionados con el rendimiento, que están disponibles en la mayoría de los procesadores actuales. Estos contadores son un conjunto de registros que van contando las veces que determinados acontecimientos suceden en el procesador.

(33)Del inglés *performance application programming interface.*

El *kernel* del sistema operativo puede necesitar algunas modificaciones para permitir acceder a los contadores de hardware mediante el uso de *PAPI*. Dependiendo del procesador y del sistema operativo, puede llegar a ser necesario aplicar un *patch* (por ejemplo, `PertCtr`) y tener que recompilar el *kernel*. La figura siguiente muestra la arquitectura básica de PAPI.

Figura 18. Arquitectura de *PAPI*



Cuando añadimos *PAPI* a una parte de nuestro código, se le pueden pasar a la aplicación como parámetros los nombres de los contadores de hardware de los que se quiere obtener información. Hay que tener presente que hay bastantes contadores de hardware a los que se puede acceder mediante *PAPI*.

Uso de *PAPI*

A modo de ejemplo, para obtener el número de MIPS o MFlops que la aplicación está obteniendo se necesitan los siguientes contadores de hardware: PAPI_FP_OPS (número de operaciones en coma flotante), PAPI_TOT_INS (número total de instrucciones) y PAPI_TOT_CYC (número total de ciclos). Otros contadores de máquina dan, por ejemplo, información relacionada con la memoria, como el número de fallos en memoria caché de cada uno de sus niveles. A continuación se muestra un ejemplo de la utilización de *PAPI*:

```
#include &#8220;papi.h&#8221;
#define NUM_EVENTS 2
int Events[NUM_EVENTS] = {PAPI_FP_OPS, PAPI_TOT_CYC};
INT EventSet;
Long long valías[NUM_EVENTS];
/* Inicialización de la librería */
retval = PAPI_library_init (PAPI_VER_CURRENT);
/* Alocatar espacio para el nuevo elemento */
retval = PAPI_create_eventSet (&EventSet);
/*Añade Flops y otros ciclos que hay que monitorizar */
retval = PAPI_add_eventset (&EventSet, Events, NUM_EVENT);
/* Empiezan a funcionar los contadores */
retval = PAPI_start(EventSet);
hacer_trabajo(); /* Lo que suponemos que monitorizamos */
/* Para los contadores y restaura los valores por defecto */
retval = PAPI_stop(EventSet, valías);
```

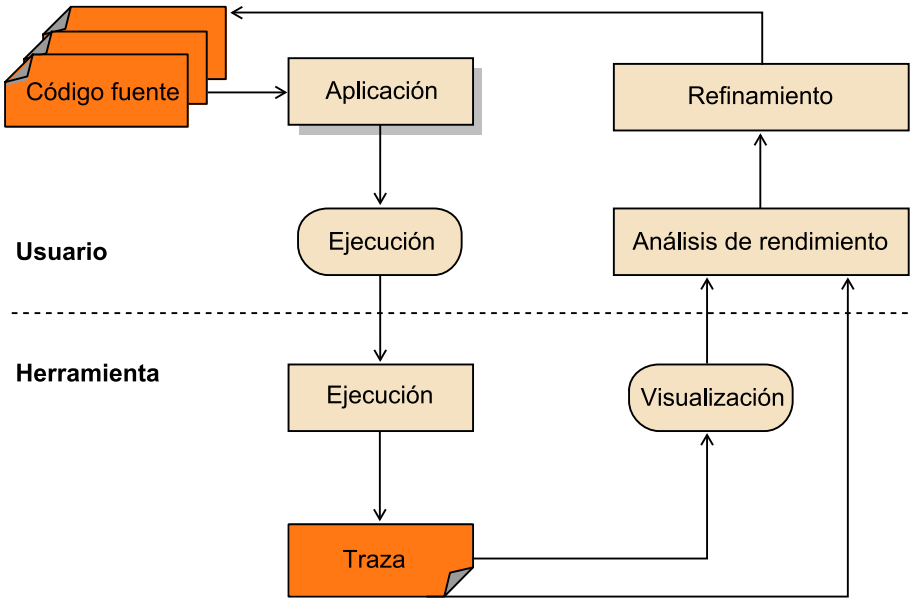
4.4.3. Herramientas de análisis de rendimiento

El objetivo de las herramientas de análisis de rendimiento es analizar de manera más o menos automática la información generada durante la fase de monitorización. Mediante estas herramientas podemos identificar posibles cuellos de botella y otros problemas propios de las aplicaciones paralelas, aunque

no proporcionan la solución a estos problemas de modo automático, sino que requiere la intervención humana para extraer conclusiones y proponer soluciones o mejoras.

La forma clásica de análisis de rendimiento está basada en la visualización de la ejecución de la aplicación paralela mediante una traza, una vez que esta ha acabado de ejecutarse. Este proceso se muestra en la figura siguiente y se denomina *post-mortem*.

Figura 19. Metodología clásica de análisis de rendimiento



Normalmente las herramientas que se utilizan en este proceso muestran información específica sobre el comportamiento de la aplicación mediante diferentes vistas gráficas y numéricas. Por ello, primeramente se requiere el uso de la herramienta que realice la monitorización para obtener datos de rendimiento de la ejecución del programa paralelo. La inserción de la instrumentación se puede realizar de manera estática a través de la herramienta o manualmente por el usuario. El proceso de monitorización se puede realizar siguiendo varias técnicas:

- Basadas en tiempo de ejecución, mediante la cual se detecta qué parte de la aplicación paralela se ejecuta más tiempo.
- Basadas en contadores que indican el número de veces de un determinado acontecimiento en la aplicación (por ejemplo, el caso de *PAPI*).
- Basadas en muestreo, que generan medidas periódicamente sobre el estado de la aplicación.
- Basadas en trazas de acontecimientos, que proporcionan información asociada a acontecimientos concretos definidos en la aplicación.

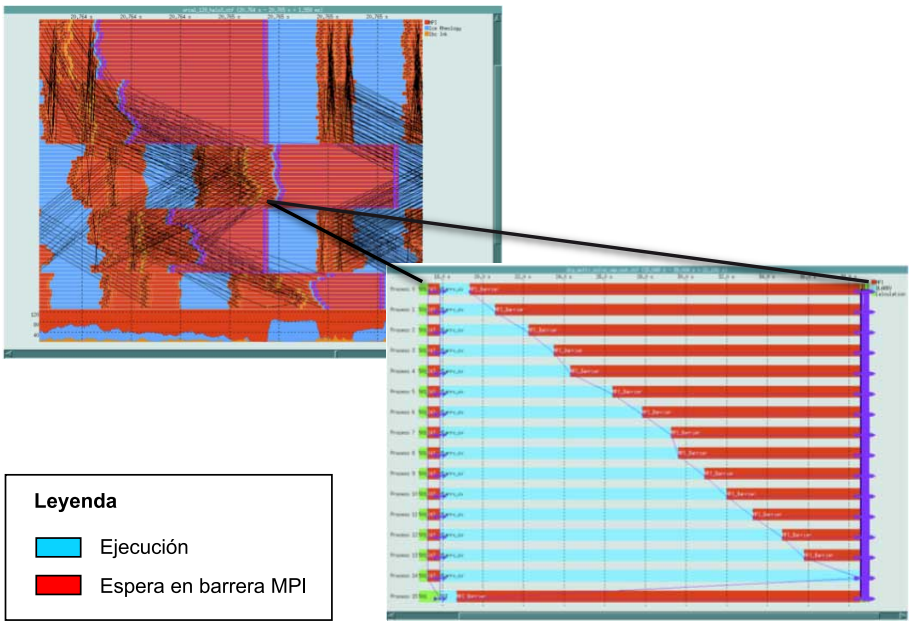
Como los datos se almacenan en un fichero de traza de la aplicación, las herramientas de visualización generan gráficos sobre patrón que sigue la aplicación, como por ejemplo diagramas de Gantt, diagramas circulares, de barras,

etc. La información mostrada se debe corresponder con aspectos relacionados con paso de mensajes, comunicaciones colectivas, ejecución de rutinas de la aplicación, entre otros. Finalmente, el usuario debe analizar estas representaciones gráficas, buscando problemas de rendimiento, determinando las causas de estos problemas y cambiando el código fuente manualmente si es necesario. De este modo, el proceso global se repite volviendo a compilar y ejecutar la aplicación, hasta que obtengamos el rendimiento que deseamos.

El análisis de rendimiento clásico requiere un elevado grado de experiencia en programación paralela para llevarlo a término de modo eficiente, así que resulta una tarea especialmente difícil para usuarios no expertos. La complejidad de esta tarea se debe principalmente a la interpretación y el tamaño del fichero de traza, que es proporcional al tamaño y al tiempo de ejecución de la aplicación. Además, esta aproximación no es fiable cuando las aplicaciones o los entornos de ejecución tienen un comportamiento dinámico. Muchas aplicaciones tienen un comportamiento diferente según los datos de entrada o incluso pueden variar durante la misma ejecución. Además, muchas herramientas de visualización no escalan bien, por lo que cuando el número de procesos implicados en la aplicación es muy elevado, los gráficos generados no son legibles correctamente.

La figura siguiente muestra una traza donde se puede observar un alto nivel de desbalanceo en los tiempos de espera de varios procesos MPI en una barrera. En esta situación sabemos que los núcleos permanecerán sin trabajo durante los tiempos de espera en la barrera (color rojo del gráfico), pero no es tan evidente encontrar una solución.

Figura 20. Ejemplo de traza. Muestra un alto nivel de desbalanceo en una barrera MPI



A continuación veremos un par de ejemplos de herramientas de análisis de rendimiento de aplicaciones paralelas, Vampir y Taubench. Cabe señalar que hay una gran diversidad de herramientas de análisis de rendimiento, como por ejemplo comerciales, como VTune de Intel, *PPW*[34] o Paraver, y otras de visualización, como Jumpshot.

(34)Del inglés *parallel performance wizard*.

Vampir es una herramienta de análisis de rendimiento que permite la visualización gráfica y el análisis de los datos del estado de un programa, mensajes punto a punto, operaciones colectivas y contadores de rendimiento de hardware, junto con resúmenes estadísticos. Está diseñada como una herramienta de fácil utilización, que permite a los desarrolladores visualizar rápidamente el comportamiento de su aplicación en un determinado nivel de detalle.

Vampir empezó a desarrollarse en el centro de matemática aplicada del Centro de Investigación de Jülich, en Alemania, y en el Centro de Computación de Altas Prestaciones de la Universidad Técnica de Dresde, también en Alemania, y está disponible como producto comercial desde 1996. Esta herramienta se ha utilizado ampliamente por la comunidad de la computación de altas prestaciones durante muchos años. El formato de traza que gestiona (actualmente *Open Trace* u OTF) es compatible con otras muchas herramientas.

Esta herramienta permite visualizar las actividades y comunicaciones de la aplicación a lo largo del tiempo en gráficos temporales, que el usuario puede explorar haciendo *zooms* y desplazándose por la ejecución para detectar la causa de los problemas de rendimiento, además de permitir la correcta paralelización y balanceo de carga. También proporciona gráficos estadísticos por carácter cuantitativos. Vampir está disponible para prácticamente todas las plataformas de 32 y 64 bits, como PC, clústeres Linux, IBM, etc. También hay disponible un software de instrumentación y medida conocido como VampirTrace.

VampirTrace proporciona una infraestructura que permite el desarrollo con instrumentación y facilidades para recoger medidas en aplicaciones *HPC*. Cubre el análisis de aplicaciones desarrolladas en *MPI* y OpenMP. La instrumentación modifica la aplicación para detectar y almacenar acontecimientos de interés generados durante la ejecución, por ejemplo, una operación de comunicación *MPI* o una determinada llamada a función. Esto se puede hacer a nivel de código fuente, durante la compilación o en tiempo de enlace mediante varias técnicas. La librería de VampirTrace se encarga de la recogida de datos en todos los procesos. Estos datos incluyen acontecimientos definidos por el usuario, acontecimientos *MPI* y OpenMP, así como información sobre temporización o localización. Además, también permite obtener información por medio de contadores de hardware mediante *PAPI*. La instrumentación automática del código fuente se puede llevar a cabo con varios compiladores, entre los que se encuentra el de GNU. La instrumentación binaria se realiza con Dy-

nist. Finalmente, los datos de rendimiento recogidos se almacenan en fichero de formato OTF. En la figura 20 se mostraba un ejemplo de traza visualizada con Vampir.

TAU[35] es un sistema de análisis de rendimiento paralelo que integra un entorno y un conjunto de herramientas automáticas para la instrumentación, la medida, el análisis y la visualización del rendimiento de aplicaciones ejecutadas en sistemas paralelos de gran escala. Una de sus principales características es el gran número de plataformas de hardware y software que soporta. *TAU* se puede ejecutar en la mayoría de los entornos de altas prestaciones y permite varios lenguajes, como por ejemplo C/C++, Java, Python, Fortan, OpenMP, MPI y Charm. Su arquitectura se organiza en tres capas: instrumentación, medida y análisis.

(35)Del inglés *tuning and analysis utilities*.

TAU utiliza un modelo de instrumentación flexible basado en instrumentación dinámica, que le permite al usuario insertar instrumentación de rendimiento llamando a la interfaz de medidas de *TAU*. El concepto clave de la capa de instrumentación es que en esta capa es donde se definen los acontecimientos de rendimiento. El mecanismo de instrumentación de *TAU* permite diferentes tipos de acontecimientos que definen el rendimiento, incluyendo acontecimientos definidos por localizaciones de código, acontecimientos de interfaz de librería, acontecimientos del sistema y acontecimientos definidos por el propio usuario. Así pues, la salida de la instrumentación es información sobre los acontecimientos de un experimento de rendimiento, que será utilizada por otras herramientas. La capa de instrumentación se comunica con la capa de medida mediante la interfaz de medida de *TAU*. Los módulos de *TAU* se dividen en componentes para hacer la depuración del código y obtener trazas, que se pueden visualizar con herramientas especializadas, como ParaProf o Vampir.

4.5. Análisis de rendimiento de sistemas de altas prestaciones

Del mismo modo que podemos analizar el rendimiento de nuestras aplicaciones paralelas en un determinado sistema, también hemos de poder analizar los sistemas de altas prestaciones en relación con las aplicaciones que queremos ejecutar. Esto nos será útil de cara a decidir o diseñar las características del sistema que necesitamos para que nuestras aplicaciones nos puedan dar el rendimiento deseado.

Tal y como hemos visto anteriormente, hay unidades de medida de rendimiento, como por ejemplo los Flops, que resultan sencillas de obtener desde un punto de vista teórico, puesto que viene determinado por la arquitectura. En cambio, este rendimiento teórico no se puede lograr en la práctica, puesto que existen los sobrecostos relacionados con el paralelismo y con la escalabilidad en sí de las aplicaciones paralelas que hemos visto anteriormente.

Identificar la red de interconexión

Un ejemplo de este tipo de estudio es identificar qué red de interconexión necesitaremos en cuanto a ancho de banda y latencia para lograr un cierto rendimiento (por ejemplo, medido en Mflops) de una aplicación de memoria distribuida con paso de mensajes. Para llevar a cabo este tipo de estudio, normalmente se utilizan herramientas de análisis de rendimiento que pueden obtener el patrón de comportamiento de la aplicación utilizando una configuración determinada y sistemas de predicción, que nos pueden decir cuál sería el rendimiento después de modificar la configuración del sistema. Aun así, este proceso no es sencillo y es específico para cada aplicación paralela.

Por tanto, para poder comparar sistemas y establecer una referencia en cuanto al rendimiento que pueden ofrecer los sistemas de altas prestaciones, se suelen utilizar pruebas de rendimiento[36], tal y como veremos a continuación.

[36]En inglés, *benchmarks*.

4.5.1. Pruebas de rendimiento

Como hemos comentado, las pruebas de rendimiento o *benchmarks* nos permiten medir el rendimiento de un sistema o de un componente en concreto (CPU, memoria, disco, etc.). Mediante los *benchmarks* podemos establecer referencias y tener criterios para comparar diferentes sistemas o componentes. En concreto, los *benchmarks* son un conjunto de programas representativos de la carga del sistema que se quiere analizar.

El *benchmark* de referencia para sistemas de altas prestaciones es el Linpack, que fue desarrollado por Jack Dongarra en 1976 en el Argone National Laboratory. El Linpack es un conjunto de subrutinas que analizan y resuelven ecuaciones lineales de alta densidad, las cuales se suelen utilizar en las aplicaciones de altas prestaciones. Los sistemas de ecuaciones lineales que se tiene en cuenta utilizan diferentes formas de matrices: generales, simétricas, triangulares, etc.

En general, el *benchmark* realiza la resolución de un sistema de ecuaciones generado aleatoriamente, expresado como una matriz de coeficientes que se representan con número de punto flotante, normalmente de precisión de 64 bits. El paso crucial de esta solución es la descomposición LU con pívot parcial de la matriz de coeficientes. El resultado del *benchmark* es un valor de rendimiento expresado en MFlops, la unidad tradicionalmente utilizada para hacer comparaciones entre diferentes sistemas.

Originalmente, Linpack se implementó en Fortran para máquinas uniprocesador, vectoriales y de memoria compartida. A medida que los sistemas de memoria distribuida se popularizaron, se hizo necesario desarrollar el *benchmark* HPL[37]. HPL es una implementación de Linpack desarrollada en lenguaje C que se puede ejecutar en cualquier equipo que disponga de una implementación de MPI.

[37]Del inglés *high performance Linpack*.

HPL es el *benchmark* utilizado para medir el rendimiento de los computadores más rápidos del mundo, que se recopilan cada seis meses en la lista Top500.

Otro *benchmark* representativo es el *HPC* Challenge Benchmark. De hecho, se trata de un conjunto de *benchmarks* que prueban varias características de sistemas de altas prestaciones. En concreto está compuesto por los siguientes siete tests:

- *HPL*: es el *high performance Linpack* que hemos comentado anteriormente.
- DGEMM: está centrado en la memoria y mide el rendimiento en coma flotante de la ejecución de la multiplicación de matrices de números reales de doble precisión.
- STREAM: es un *benchmark* basado en un programa sintético que mide el ancho de banda a memoria sostenido (en GB/s).
- *PTRANS*[38]: está centrado en las comunicaciones entre pares de procesadores que se comunican entre ellos simultáneamente. Este *benchmark* está orientado a la evaluación de la capacidad de la red.
- Acceso aleatorio: está centrado en medir el rendimiento de accesos aleatorios a memoria (también denominado GUPS).
- FFT: es un *benchmark* que mide el rendimiento en punto flotante de la ejecución de una transformada discreta de Fourier compleja unidimensional de doble precisión.
- Ancho de banda y latencia de comunicaciones: es un conjunto de pruebas que miden la latencia y el ancho de banda de varios patrones de comunicaciones simultáneamente. Está basado en el *benchmark b_eff*[39].

(38) Del inglés *parallel matrix transpose*.

(39) Del inglés *effecive bandwidth benchmark*.

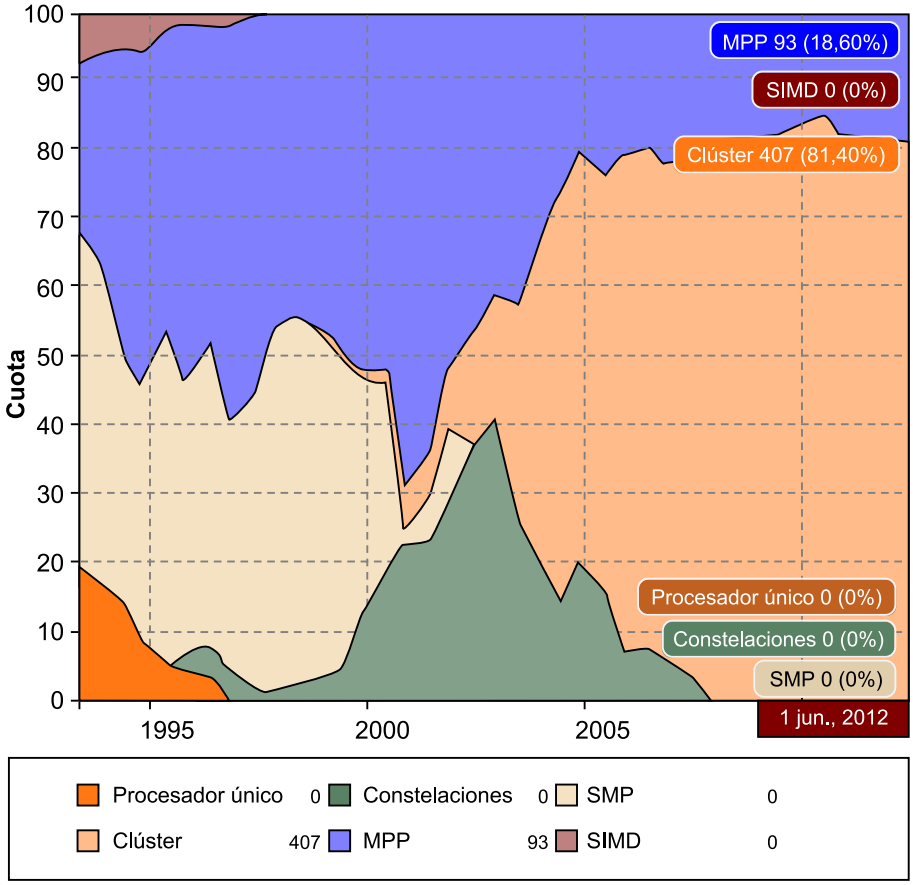
4.5.2. Lista Top500

Cada seis meses, los 500 supercomputadores más rápidos del mundo se evalúan ejecutando el *benchmark* Linpack que hemos comentado anteriormente, sobre un conjunto de datos muy grande (entre otras razones, para evitar limitaciones de escalabilidad debido a un tamaño de problema demasiado pequeño). La lista, llamada Top500, varía de año en año y se puede ver como una especie de competición.

Además de hacer públicos los supercomputadores más rápidos en cada momento y sus características, el Top500 también publica estadísticas de otra información de interés para entender la evolución de la computación de altas prestaciones.

Es interesante observar en la figura siguiente la evolución en el tiempo de la arquitectura de los supercomputadores del Top500. En 1993, 250 sistemas eran básicamente de memoria compartida y todos ellos desaparecieron de la lista en junio del 2002. Los sistemas *SIMD* desaparecieron en 1997. Los sistemas MPP fueron muy populares hacia el año 2000 pero han tenido una evolución a la baja a pesar de que todavía se pueden encontrar en la lista. En cambio, los sistemas basados en clúster aparecieron en 1999 y actualmente son los más populares del Top500, representando la clase de arquitectura dominante. La principal diferencia entre los clústeres y los sistemas MPP es que los clústeres utilizan componentes de mercado, mientras que los sistemas MPP tienen los propios diseños específicos de nodos, módulos, red de interconexión, etc.

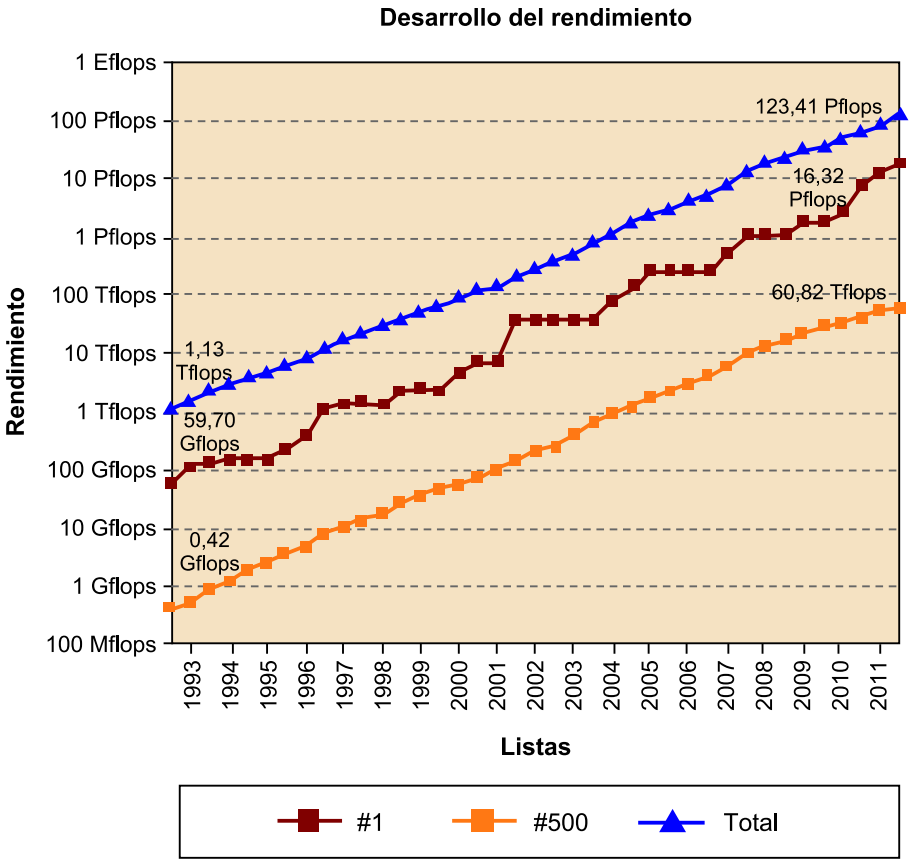
Figura 21. Evolución del tipo de arquitecturas de los sistemas del Top500 desde 1993
Fuente: http://www.top500.org



La figura siguiente muestra la evolución del rendimiento de los supercomputadores del Top500 desde 1993 hasta junio del 2012. El eje de las *Y* está escalado en términos de rendimiento sostenido medido en GFlops, TFlops y PFlops. La serie del medio corresponde al rendimiento del supercomputador más rápido del mundo durante las casi dos décadas, que se incrementa de 58,7 GFlops a 16,32 PFlops. La serie inferior corresponde a la velocidad del último sistema del Top500 (en la posición número 500) y la superior corresponde a la suma del rendimiento de los 500 supercomputadores de un mismo periodo de tiem-

po. En junio del 2012 la suma de los 500 supercomputadores suma 123,41 PFlops. Es interesante observar que la suma total de rendimiento aumenta de manera prácticamente lineal.

Figura 22. Evolución del rendimiento de los supercomputadores del Top500 desde 1993
Fuente: http://www.top500.org



En la tabla siguiente se muestran las características de los diez supercomputadores más rápidos en la lista del Top500 de junio del 2012 junto con su rendimiento. La velocidad sostenida, Rmax (en PFlops), se mide a partir de la ejecución del *benchmark* Linpack con el máximo tamaño de matriz. Rpeak es la velocidad máxima sostenida cuando todos los elementos del sistema están siendo utilizados completamente. La potencia eléctrica es en KW, lo que quiere decir que el supercomputador de la lista que requiere más potencia eléctrica pasa los 12 MW. Hay que destacar que el orden de los sistemas en cuanto a su rendimiento no coincide con el de número de núcleos o potencia.

Tabla 5. Lista de los 10 supercomputadores más rápidos de la lista del Top500 de junio del 2012

#	Institución	Computador/Año Fabricante	Núcleos	Rmax/ Rpeak	Potencia
1	DOE/NNSA/LLNL Estados Unidos	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom / 2011 IBM	1572864	16324/ 20132	7890

#	Institución	Computador/Año Fabricante	Núcleos	Rmax/ Rpeak	Potencia
2	RIKEN Advanced Institute for Computational Science (AICS) Japó	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	705024	10510/ 11280	12659
3	DOE/SC/Argonne National Laboratory Estados Unidos	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	786432	8162/ 10066	3945
4	Leibniz Rechenzentrum Alemania	SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR / 2012 IBM	147456	2897/ 3185	3422
5	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 / 2010 NUDT	186368	2566/ 4701	4040
6	DOE/SC/Oak Ridge National Laboratory Estados Unidos	Jaguar - Cray XK6, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA 2090 / 2009 Cray Inc.	298592	1941/ 2627	5142
7	CINECA Italia	Fermi - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	163840	1725/ 2097	821
8	Forschungszentrum Juelich (FZJ) Alemania	JuQUEEN - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	131072	1380/ 1677	657
9	CEA/TGCC-GENCI Francia	Curie thin nodes - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR / 2012 Bull	77184	1359/ 1667	2251
10	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 / 2010 Dawning	120640	1271/ 2984	2580

También hay que destacar otras iniciativas relacionadas, como el Graph500 o Green500. Mientras que el primero ordena los sistemas de altas prestaciones orientado a aplicaciones intensivas de datos, puesto que cada vez están

Ved también

La eficiencia energética se trata en el último módulo de esta asignatura.

tomando más protagonismo en la computación de altas prestaciones, el segundo se centra en la eficiencia energética, tal y como veremos en el último módulo de la asignatura.

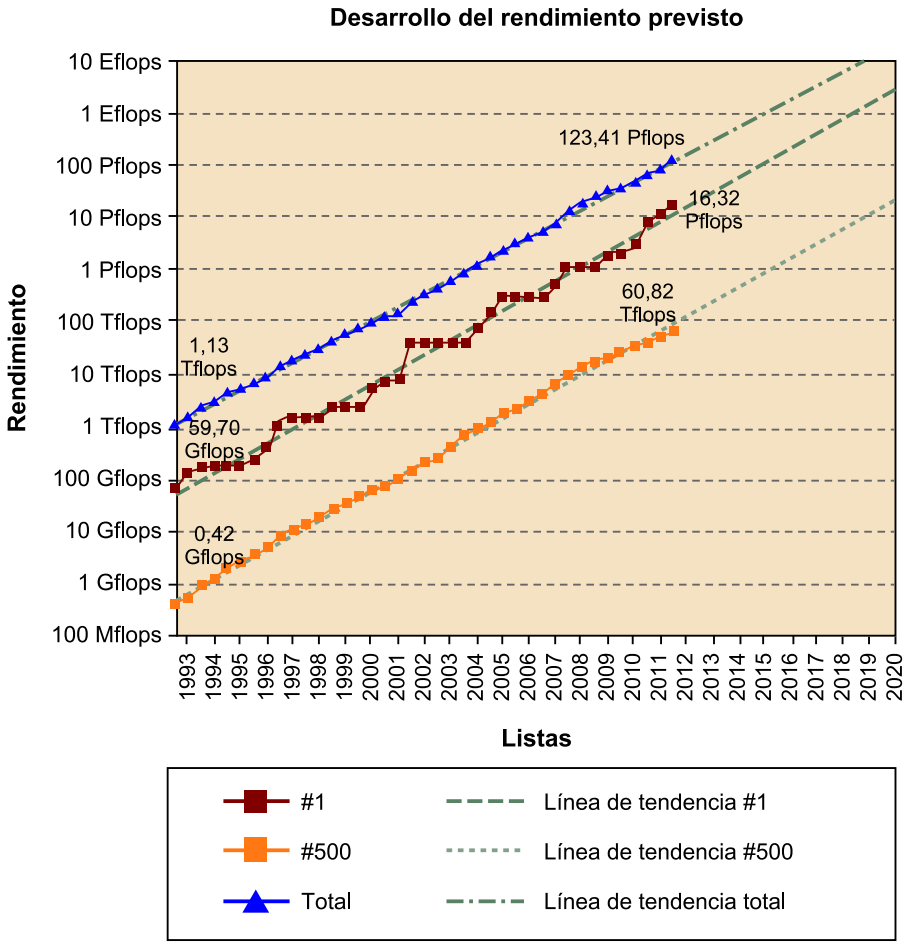
5. Retos de la computación de altas prestaciones

Con los avances tecnológicos el rendimiento de los computadores de altas prestaciones mejora año tras año, como hemos visto anteriormente en la figura 32. Aun así, muchos de estos avances tecnológicos (por ejemplo, el aumento de densidad de transistores en los microprocesadores) están llegando a sus límites por varios motivos, como pueden ser los límites físicos o energéticos, y es necesario un cambio de paradigma para poder construir un computador que tenga un rendimiento sostenido de un exaflop.

El objetivo actual de la comunidad científica es poder desarrollar sistemas de la escala del exaflop aproximadamente en el 2018. Este objetivo coincide con la proyección del rendimiento de los supercomputadores de cara al 2020, tal y como muestra la figura siguiente, pero –como veremos a continuación– hay nuevos retos que habrá que tomar muy seriamente para lograr el exaflop. De hecho, dentro de las múltiples iniciativas en esta dirección hay que destacar el proyecto "The International Exascale Software", en el que científicos de todo el mundo han definido una hoja de ruta para conseguir esta escala de rendimiento de manera sostenible. El proyecto cuenta con el apoyo de instituciones, gobiernos y empresas de todo el mundo, especialmente de la NSF[40] y del Departamento de Energía de Estados Unidos, y agencias en Europa y Asia.

[40]Del inglés National Science Foundation.

Figura 23. Proyección del rendimiento de los supercomputadores del Top500 hasta el año 2020



Fuente: http://www.top500.org

5.1. Concurrencia extrema

Para conseguir un rendimiento de exaflop habrá que aumentar el nivel de concurrencia en hasta 1.000 veces más por cada trabajo individual. Esto quiere decir que los trabajos estarán compuestos posiblemente de centenares de miles o millones de elementos que hay que sincronizar y gestionar adecuadamente y de modo eficiente.

De esta manera, habrá que disponer de nuevos modelos de programación que den respuesta a la necesidad de tanto nivel de paralelismo y que, además, faciliten que grupos de aplicaciones puedan gestionar la concurrencia correspondiente de manera más natural. Esta capacidad seguramente deberá incluir una escalabilidad fuerte, puesto que el aumento del volumen de memoria principal no coincidirá con el de los procesadores. Esto también querrá decir que habrá que reducir al mínimo el sobrecoste relacionado con las capas de software que serán necesarias.

5.2. Energía

La eficiencia energética es una de las prioridades actuales de los profesionales que trabajan en el diseño de los futuros sistemas de exaflop, que, si se desarrollara con tecnología actual, consumiría varios centenares de MW de potencia eléctrica, lo que no es viable ni sostenible desde varias perspectivas. Así pues, se requiere una reducción de la energía de varias órdenes de magnitud y esto debe ir acompañado de optimizaciones a diferentes niveles, junto con nuevos algoritmos y diseños de aplicaciones.

Con la eficiencia energética como prioridad, se espera que hacia el 2017 los computadores llegarán a 200 petaflops, con un límite de consumo energético de 10 MW. De cara a como mucho el 2020 el reto es llegar a 1 exaflop con un consumo de 20 MW. Esto supondría una eficiencia energética 20 veces superior a la de las máquinas que consumen menos en la actualidad (por ejemplo, BlueGene/Q; podéis ver la tabla 5).

Primero hay que tener en cuenta que no toda la energía se destina a alimentar los núcleos. En los sistemas actuales, los procesadores necesitan un gran volumen de energía, a menudo un 40% o incluso más. La energía restante se utiliza para las memorias, la red de interconexión y el sistema de almacenamiento. Estas proporciones están cambiando y la memoria principal cada vez consume una proporción más grande de energía.

También hay que tener en cuenta que la energía que se requiere no solo es para cómputo, sino que también es para transportar por la red y analizar los datos que se pueden ir generando durante simulaciones a escalas extremas. Como una parte importante de la energía requerida para un sistema a escala de exaflop se deberá destinar a mover los datos entre procesadores y memoria y de forma más global, habrá que tener la infraestructura de software que lo pueda gestionar eficazmente.

Ved también

Recordad que estas cuestiones de la eficiencia energética se tratarán en el último módulo de esta asignatura.

5.3. Tolerancia a fallos

Los futuros sistemas de exaflop estarán construidos con dispositivos VLSI que no serán tan fiables como los que se utilizan actualmente. Todo el software, y por lo tanto todas las aplicaciones, deberán tener en cuenta la tolerancia a fallos como un factor más en su diseño. Así, cuando la escala del sistema aumente hasta el exaflop tendremos que soportar el reconocimiento y la adaptación a errores de manera continua y proporcionar los mecanismos necesarios para que las aplicaciones hagan lo mismo. De hecho, el paralelismo masivo y el elevado número de componentes electrónicos de estos sistemas hará que los fallos sean inevitables y también que haya una cierta incertidumbre que se deberá tener en cuenta como elemento en el diseño de estas futuras aplicaciones.

5.4. Heterogeneidad

Los sistemas heterogéneos ofrecen la oportunidad de explotar el rendimiento de dispositivos, como GPU para computación de propósito general. Un ejemplo de ello es el computador Nebulae, el cual utiliza CPU Intel Xeon y aceleradores Nvidia. De la misma manera, las aplicaciones científicas también se están volviendo más heterogéneas.

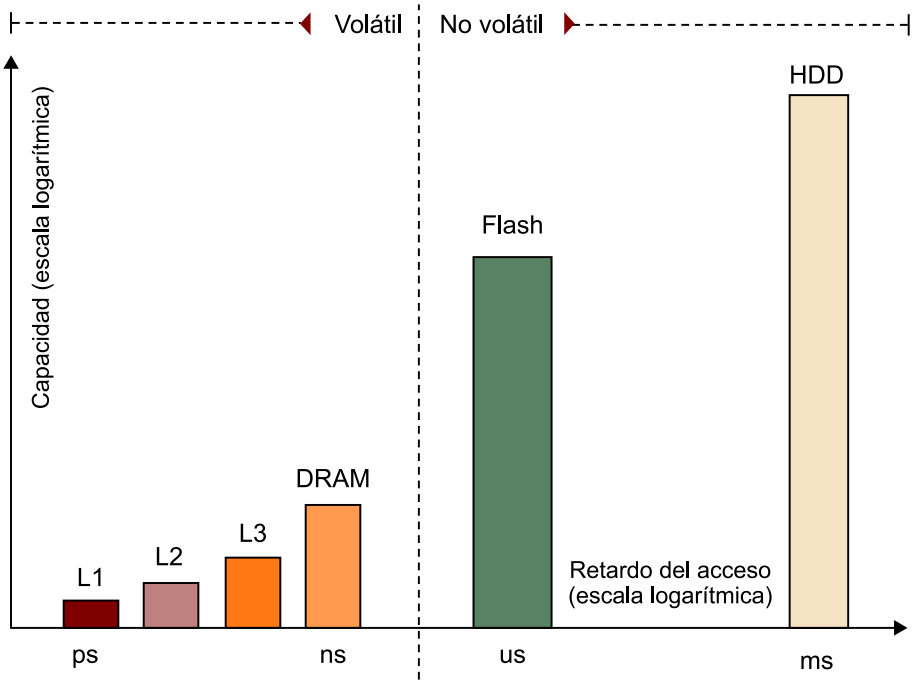
5.5. Entrada/salida y memoria

No disponer de suficiente capacidad de entrada/salida hoy en día es un cuello de botella. Además, habrá que gestionar y analizar grandes volúmenes de datos que se generarán muy rápidamente, lo que aumentará vertiginosamente durante la próxima década.

La jerarquía de memoria deberá cambiar, puesto que habrá nuevas tecnologías de memoria que habrá que tener en cuenta. Este cambio en la jerarquía de memoria acabará afectando tanto a los modelos de programación como a las optimizaciones. La figura siguiente muestra la diferencia de acceso a los distintos niveles de la jerarquía de memoria como motivación para la incorporación de nuevas tecnologías, como por ejemplo, la memoria no volátil o NVRAM[41].

(41)Del inglés *non-volatile random-access memory*.

Figura 24. Diferencias en tiempos de acceso a los distintos niveles de la jerarquía de memoria
Fuente: Fusion-IO



Hay que tener en cuenta que los dos ejes de la figura anterior están en escala logarítmica. Así pues, podemos ver claramente que hay una diferencia muy elevada tanto de rendimiento como de tiempo de acceso entre la memoria DRAM y el disco duro. En la tabla 6 se presenta una analogía bastante ilustrativa.

Tabla 6. Analogía de los tiempos de acceso a los diferentes niveles de la jerarquía de memoria

Nivel	Comida	Tiempo de acceso relativo
Caché nivel 1	Comer de la boca	Fracciones de segundo
Caché nivel 2	Coger comida del plato	1 segundo
Caché nivel 3	Coger comida de la mesa	Unos pocos segundos
DRAM	Coger comida de la cocina	Unos pocos minutos
FLASH	Coger la comida de una tienda cercana	Unas pocas horas
Disco duro	Coger la comida de Marte	3-5 años

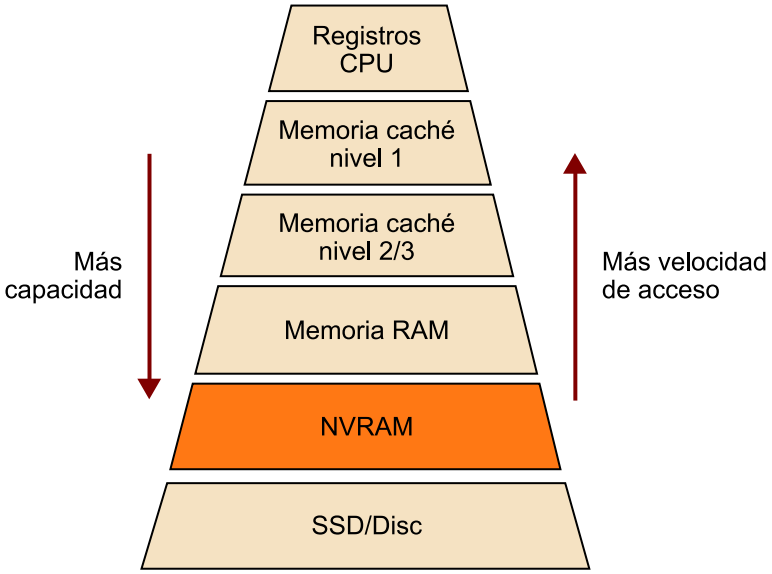
Teniendo en cuenta lo expuesto, una de las soluciones que se están adoptando y que pasarán a tener un papel destacado en el desarrollo de sistemas de exaflop es extender la jerarquía de memoria con un nuevo nivel que ocupará la memoria no volátil, como muestra la figura siguiente, que extiende los conceptos que salieron en la figura 12. También se espera la utilización de otras tecnologías que proporcionan rendimientos intermedios, como la utilización de tecnología *SSD*[42].

[42]Del inglés *solid state drive.*

Ved también

Encontraréis la figura 12 en el subapartado 2.2.2, al tratar los sistemas de memoria compartida.

Figura 25. Jerarquía de memoria extendida con memoria RAM no volátil



Bibliografía

Buyya, R. (1999). *High Performance Cluster Computing: Architectures and Systems* (vol. 1). Englewood Cliffs, NJ: Prentice Hall.

Grama, A.; Karypis, G.; Kumar, V.; Gupta, A. (2003). *Introduction to Parallel Computing*. Reading, Massachusetts: Addison-Wesley.

Hwang, K.; Fox, G. C.; Dongarra, J. J. (2012). *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Burlington, Massachusetts: Morgan Kaufmann.

Jain, R. (1991). *The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience.

Kirk, D. B.; Hwu, W. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, Massachusetts: Morgan Kaufmann.

Lin, C.; Snyder, L. (2008). *Principles of Parallel Programming*. Reading, Massachusetts: Addison Wesley.

Munshi, A.; Gaster, B.; Mattson, T. G.; Fung, J.; Ginsburg, D. (2011). *OpenCL Programming Guide*. Reading, Massachusetts: Addison-Wesley Professional.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Burlington, Massachusetts: Morgan Kaufmann.