

# COMPUTACIÓN DE ALTAS PRESTACIONES

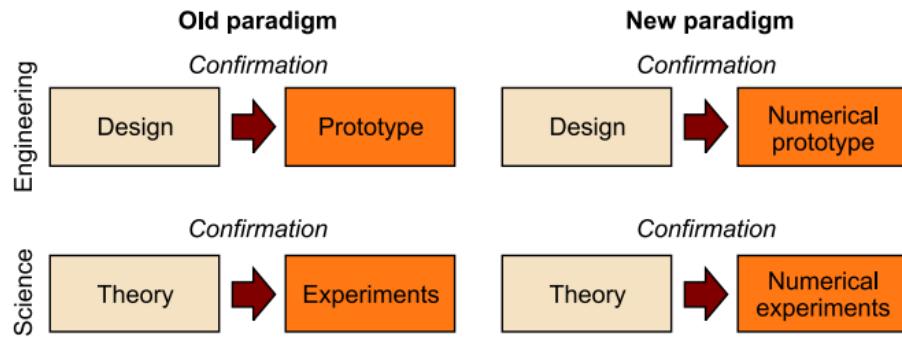
Iker Caballero Bragagnini

## Tabla de contenido

LA COMPUTACIÓN DE ALTAS PRESTACIONES O HPC .....	2
EL PARALELISMO Y LAS APLICACIONES EN PARALELO: ARQUITECTURAS EN PARALELO .....	11
EL PARALELISMO Y LAS APLICACIONES EN PARALELO: EL RENDIMIENTO .....	ERROR! BOOKMARK NOT DEFINED.
LA COMPUTACIÓN DISTRIBUIDA: FUNDAMENTOS .....	ERROR! BOOKMARK NOT DEFINED.
LA COMPUTACIÓN DISTRIBUIDA: COMPUTACIÓN <i>GRID</i> Y <i>CLOUD</i> .....	ERROR! BOOKMARK NOT DEFINED.
LA PROGRAMACIÓN EN PARALELO: MODELOS DE MEMORIA COMPARTIDA .....	33
LA PROGRAMACIÓN EN PARALELO: MODELOS GRÁFICOS .....	92
LA PROGRAMACIÓN EN PARALELO: MODELOS DE MEMORIA DISTRIBUIDA .....	105
LA PROGRAMACIÓN EN PARALELO: ESQUEMAS PARALELOS ALGORÍTMICOS.....	121
LAS ARQUITECTURAS DE SISTEMAS HPC: DATOS Y TAXONOMÍA .....	33
LAS ARQUITECTURAS DE SISTEMAS HPC: SIMD Y MIMD .....	39
LAS ARQUITECTURAS DE SISTEMAS HPC: ARQUITECTURAS DE MÚLTIPLES NÚCLEOS .....	ERROR! BOOKMARK NOT DEFINED.
LAS ARQUITECTURAS DE SISTEMAS HPC: REDES DE INTERCONEXIÓN Y SISTEMAS DE ARCHIVOS .....	53
LAS ARQUITECTURAS DE SISTEMAS HPC: GESTIÓN DE TAREAS Y RECURSOS.....	66
LAS ARQUITECTURAS DE SISTEMAS HPC: REDES DE INTERCONEXIÓN Y SISTEMAS DE ARCHIVOS	ERROR! BOOKMARK NOT DEFINED.
LA COMPUTACIÓN VERDE: FUNDAMENTOS .....	132
LA COMPUTACIÓN VERDE: EFICIENCIA ENERGÉTICA Y SOSTENIBILIDAD .....	ERROR! BOOKMARK NOT DEFINED.

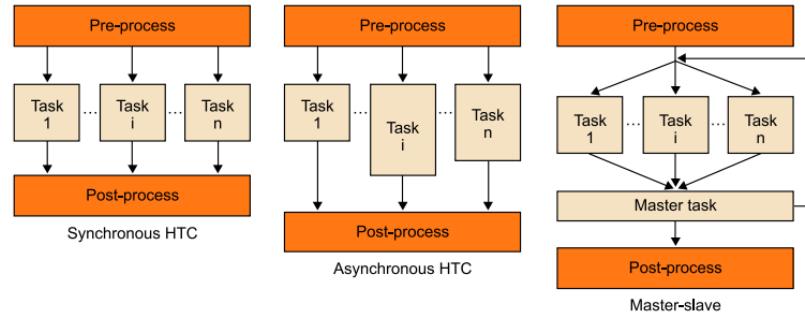
## La computación de altas prestaciones o HPC

- En general, la computación de altas prestaciones ha sido una respuesta a la demanda creciente para los niveles de mayores niveles de rendimiento computacional y rapidez necesaria para resolver problemas incrementalmente complejos dentro de un intervalo de tiempo razonable
  - La simulación numérica se considera como el tercer pilar de la ciencia, con las otras dos siendo experimentación/observación y teoría. El paradigma tradicional para la ciencia y la ingeniería se basa en la teoría o diseños en papel, de modo que los experimentos se pueden realizar en el sistema creado a partir de la teoría o del diseño
    - No obstante, este paradigma sufre de algunas limitaciones para resolver ciertos tipos de problemas, tales como problemas difíciles, caros, lentos o muy arriesgados
    - El paradigma de la ciencia computacional se basa en usar sistemas de alto rendimiento para simular el fenómeno deseado. Por lo tanto, la ciencia computacional se basa en las leyes de la física y los métodos numéricos eficientes
  - Hay algunos tipos de aplicaciones que tradicionalmente han sido un reto para la comunidad académica, y que también requieren computación de altas prestaciones porque no se pueden realizar en otros tipos de computadores
    - Algunas de estas aplicaciones son en ciencia (cambio climático, astrofísica, etc.), en ingeniería (simulaciones de comprobación, diseño de semiconductores, etc.), en negocios (modelos financieros, procesos de transacción, etc.) y en aplicaciones militares (criptografía, comprobación nuclear, etc.)
  - En aplicaciones de ingeniería, la computación de altas prestaciones decrece la necesidad de usar prototipos durante el proceso de diseño y optimización de procesos. Esto a su vez reduce costes, incrementa la productividad (por la reducción del tiempo de desarrollo), y el incremento de seguridad



- En aplicaciones científicas, la computación de altas prestaciones permite simular sistemas a gran escala y a pequeña escala, y la validez de la modelo matemática también se puede analizar
- Las aplicaciones que requieren alto rendimiento se pueden dividir en dos grandes grupos: las aplicaciones de computación de alto *throughput* o *high-throughput computing* (HTC) y las aplicaciones de computación de altas prestaciones o *high performance computing* (HPC)
  - Se pueden mirar a estas aplicaciones desde la perspectiva de cómo gestionan el problema. La computación en paralelo se basa en el desarrollo de aplicaciones que toman ventaja del uso colaborativo de múltiples procesadores con tal de resolver el problema
    - En verdad, el objetivo fundamental que persiguen estas tecnologías es reducir el tiempo de ejecución de una aplicación para resolver un problema
  - Los esfuerzos también se pueden hacer para resolver grandes problemas usando las múltiples memorias de los procesadores involucrados en la ejecución. Los dos conceptos más fundamentales son la partición de tareas y la comunicación entre tareas
    - La partición entre tareas consiste en dividir una gran tarea en varias subtareas diferentes que se pueden llevar a cabo utilizando múltiples unidades de procesamiento
    - Aunque cada proceso realiza una única subtarea, generalmente será necesario que estos procesos se comuniquen entre sí para cooperar en la resolución del problema general
  - El propósito de las aplicaciones de HTC es incrementar el nivel de ejecuciones por unidad de tiempo. Su rendimiento se mide como el número de tareas ejecutadas por número de tiempo (por ejemplo, tareas por segundo)

- Existen tres tipos de modelos para aplicaciones HTC, en donde se asume que la tarea que se está intentando realizar se puede dividir en múltiples tareas (a veces llamadas trabajos o *jobs*): las tareas síncronas, las tareas asíncronas, y las tareas *master-slave*

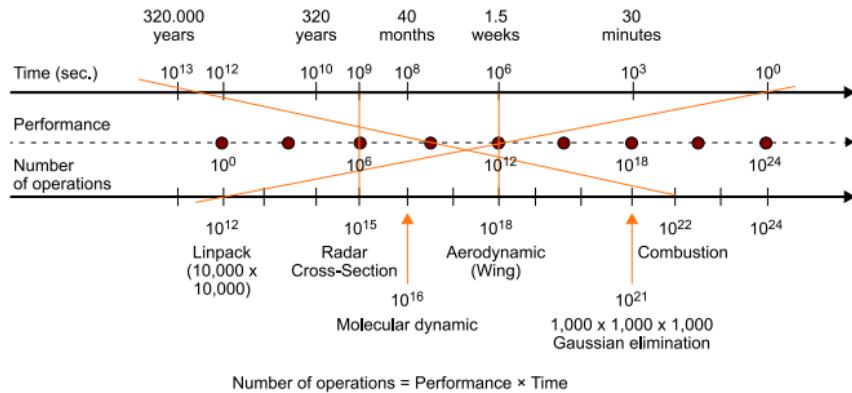


- En las aplicaciones HTC síncronas, los *jobs* se sincronizan y se terminan al mismo tiempo, mientras que, en las aplicaciones asíncronas, los *jobs* pueden terminar en momentos diferentes. En las aplicaciones HTC *master-slave*, existe un *job* especializado (*master*) que es responsable de sincronizar el resto de trabajos (*slaves*)
- El propósito de las aplicaciones HPC es reducir el tiempo de ejecución de una única aplicación en paralelo. Su rendimiento es se mide en número de operaciones de punto flotante por segundo o flops (*floating-point operations per second*), normalmente medido en millones, billones y trillones de flops

Name	Flops
Megaflops	$10^6$
Gigaflips	$10^9$
Teraflips	$10^{12}$
Petaflips	$10^{15}$
Exaflips	$10^{18}$
Zettaflips	$10^{21}$
Name	Flops
	$10^{24}$

- Las dos áreas que suelen requerir este tipo de aplicaciones son la del estudio de fenómenos en escala microscópica, en donde la resolución se limita a la potencia de la computadora (cuantos más grados de libertad, mejor se representa la verdad), y el estudio de fenómenos a gran escala, en donde la precisión se determina por la computadora (cuantos más puntos, la solución discreta se parecerá cada vez más a la continua)

- Actualmente, la computación de altas prestaciones es sinónimo con el paralelismo, por lo que se necesita un número creciente de procesadores. Estos se deben incrementar para resolver problemas con menor tiempo de ejecución usando más procesadores, resolver problemas con más precisión al usar más memoria, y resolver problemas más realistas usando modelos matemáticos más complejos



- Un repaso de la historia del rendimiento de los sistemas computacionales permite entender la necesidad de su mejora y como aquellas técnicas de concurrencia y paralelismo son una de las grandes soluciones para reducir los tiempos de ejecución
  - Durante décadas recientes, el rendimiento de los microprocesadores ha incrementado a un ritmo del 50% cada año, lo cual significó que los usuarios y desarrolladores solo tenían que esperar a la siguiente mejora de microprocesadores con tal de obtener una mejora sustancial del rendimiento de sus programas
  - No obstante, desde 2002, las mejoras en el rendimiento de procesadores individuales se han ralentizado un 20% cada año. Esta es una diferencia significativa dado que, teniendo en cuenta un incremento del rendimiento del 50% anual, en 10 años el factor será aproximadamente 60, mientras que con un incremento del 20%, el factor sería 6
  - Uno de los métodos más importantes para incrementar el rendimiento computacional es incrementar la velocidad del reloj del procesador, lo cual ha sido posible por incrementar la densidad del procesador. Cuando el tamaño del transistor decrece, su velocidad se puede incrementar, y la velocidad del circuito entero incrementa
  - Esto se explica por la ley de Moore, una observación empírica que explica que la densidad de los circuitos en un chip se dobla cada 18 meses. No obstante, esta ley tiene límites obvios

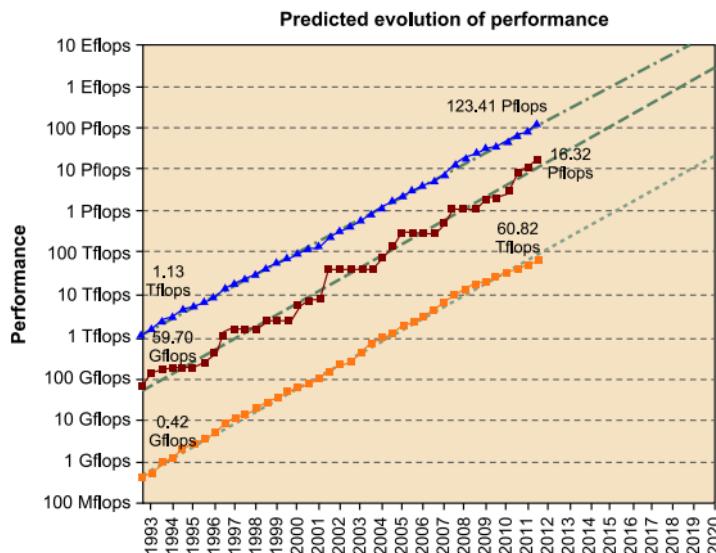
<b>Processor</b>	<b>Year</b>	<b>Number of transistors</b>	<b>Technology</b>
4004	1971	2,250	10 µm
8008	1972	3,500	10 µm
8080	1974	6,000	6 µm
8086	1978	29,000	3 µm
286	1982	134,000	1.5 µm
386	1985	275,000	1 µm

<b>Processor</b>	<b>Year</b>	<b>Number of transistors</b>	<b>Technology</b>
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

- Cuando la velocidad del transistor incrementa, su consumo eléctrico también incrementa, y en su mayoría se transforma en calor (lo cual hace que, cuando un circuito se sobrecalienta, sea poco fiable)
  - A la vez, también se tienen limitaciones en relación al paralelismo a nivel de las instrucciones, debido a que hacer el *pipeline* más extenso puede hacer que eventualmente se baje el rendimiento
  - Esto significa que actualmente no es viable continuar incrementando la velocidad de circuitos integrados. También es importante enfatizar que, aunque se puede seguir incrementando la densidad de los transistores, esto solo seguirá siendo posible por un periodo de tiempo limitado
- Esto deja al paralelismo como la única manera restante de usar la ventaja de la densidad de los transistores: en vez de hacer procesadores monolíticos que son crecientemente más veloces y complejos, la solución que la industria ha adoptado es el desarrollo de procesadores de varios núcleos, enfocados al rendimiento de la ejecución para aplicaciones en paralelo y no tanto para aplicaciones secuenciales
  - Esto significa que, en los años recientes, un cambio muy significativo ha tomado lugar en la industria de la computación en paralelo

- Actualmente, casi todas las computadoras para consumidores incorporan procesadores con varios núcleos, y el concepto de núcleo se ha vuelto sinónimo del de CPU. Como los procesadores con múltiples núcleos se han vuelto parte del día a día, la computación en paralelo no es algo exclusivamente para computadoras especiales y sistemas de alto rendimiento
- Los nuevos dispositivos son capaces de usar computación en paralelo para proporcionar características más sofisticadas que la de sus predecesores
- Este cambio también comporta consecuencias dramáticas para los programadores, dado que la nueva generación de circuitos integrados que incorporan más procesadores no incrementa el rendimiento de manera automática en aplicaciones secuenciales (como era el caso anteriormente). Se tienen que usar nuevas técnicas de programación para usar ventajosamente esta nueva generación de procesadores
- Los avances tecnológicos han permitido incrementar el rendimiento de las computadoras año tras año. No obstante, muchos de estos avances tecnológicos están llegando a su límite por varios motivos, ya sean físicos o energéticos, haciendo necesario un cambio en el paradigma para crear mejores ordenadores
  - Aunque el objetivo de la comunidad científica era desarrollar sistemas en la escala del exaflop, han surgido nuevos retos que se tienen que tener en cuenta con tal de alcanzar este tiempo de ejecución deseado



- Dentro de las múltiples iniciativas para moverse en esta dirección, el proyecto *International Exascale Software* es la más importante,

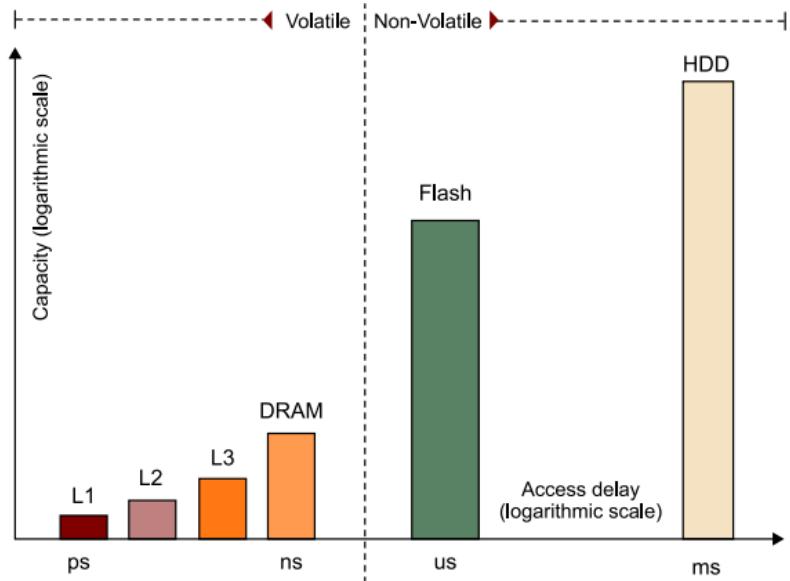
dado que define una guía para conseguir esta escala de rendimiento de manera sostenible

- Este proyecto es apoyado por varias compañías y órganos gubernamentales alrededor del mundo
- Con tal de conseguir rendimientos de exaflops, el nivel de concurrencia o simultaneidad se tendría que incrementar 1000 veces para cada tarea individual. Esto significa que las tareas tendrían que estar compuestas de cientos de miles o millones de elementos, todos los cuales se deben sincronizar y gestionar de una manera eficiente
  - Esto va a requerir nuevos modelos de programación que respondan a la necesidad de tal nivel de paralelismo. Los grupos de aplicaciones también tendrán que ser capaces de gestionar la concurrencia o simultaneidad correspondiente de la manera más natural
  - La capacidad seguramente tiene que incluir una fuerte escalabilidad, dado que el incremento en la cantidad de memoria principal no coincidirá con los incrementos en los procesadores. Todo esto también significa que los gastos generales relacionados con las capas de software necesarias deberán reducirse al mínimo posible
- La eficiencia energética es una de las prioridades actuales de los profesionales en el diseño de sistemas futuros que rindan a exaflops. Esto se debe a que, si estos sistemas se desarrollan usando la tecnología actual, se consumiría muchísima energía, lo cual no es sostenible desde varias perspectivas
  - No obstante, esto significa que se requiere una reducción en la energía necesaria en múltiples órdenes de magnitud, y esto debe estar acompañado por la optimización en varios niveles junto con nuevos algoritmos y diseños de aplicaciones
  - Se tiene que recordar que no toda la energía se dedica a los núcleos: en los sistemas actuales, los procesadores requieren un gran volumen de energía (40% o más), pero el resto de la energía se usa para las memorias, la red interconectada y el sistema de almacenamiento. Sin embargo, las proporciones están cambiando, y la memoria principal está consumiendo una parte importante de la energía
  - También se tiene que tener en cuenta que la energía requerida no se usa para la computación, pero también para el transporte de la red y el análisis de los datos generados durante simulaciones

en gran escala. Dado que una porción significativa de la energía requerida para un sistema que rinda en exaflops se tiene que dedicar a mover los datos (entre procesadores y memorias y también globalmente), la infraestructura de *software* necesaria para manejar el movimiento también se tiene que tener en cuenta

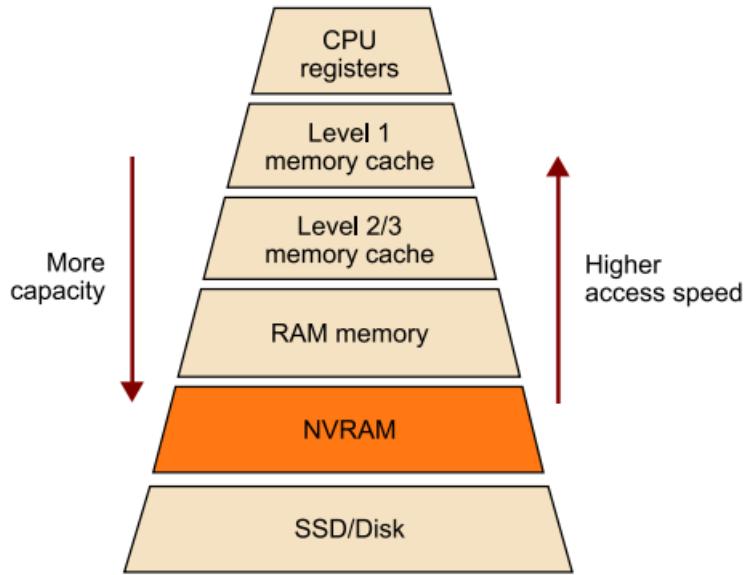
- Los sistemas que rinden en exaflops deben construirse usando dispositivos de integración de escala muy alta o *very large scale integration* (VLSI) que no serán tan fiables como los actuales. Todo el *software* tendrá que considerar una tolerancia de error como un factor más en su diseño
  - Esto significa que cuando la escala del sistema incrementa en un exaflop, se tiene que reconocer y adaptar a una serie continua de errores, mientras que también proporciona los mecanismos necesarios para permitir a las aplicaciones adaptarse
  - El paralelismo masivo y el número alto de componentes electrónicos en estos sistemas significa que los errores son inevitables y cuando se diseñan estas aplicaciones futuras, también habrá un grado de incertidumbre que se tendrá que tener en cuenta
- Los sistemas heterogéneos ofrecen la oportunidad de aprovechar el rendimiento de otros dispositivos (tales como GPUs) para computación general
  - Un ejemplo de esto es la computadora Nebulae, que usa una Intel Xeon CPU y aceleradores NVIDIA
  - De manera similar, las aplicaciones científicas también se están volviendo más y más heterogéneas
- Actualmente, una falta de capacidad de *input-output* es un cuello de botella. Además, la gestión y análisis también se tendrán que realizar en grandes volúmenes de datos que se generan muy rápido, con esta cantidad incrementando más y más cada vez
  - La jerarquía de memoria tendrá que cambiar con tal de tener en cuenta las nuevas tecnologías de memoria. Este cambio en la jerarquía de la memoria acabará afectando a los modelos de programaciones y a las optimizaciones
  - El siguiente gráfico muestra como una diferencia en el acceso en los diferentes niveles de jerarquía de memoria es una motivación para incorporar nuevas tecnologías, tales como el acceso de

memoria aleatorio no volátil o *non-volatile random-access memory* (NVRAM). Los ejes tienen una escala logarítmica, de modo que hay una diferencia sustancial entre la memoria DRAM y el disco duro (HDD) en términos de rendimiento y tiempo de acceso



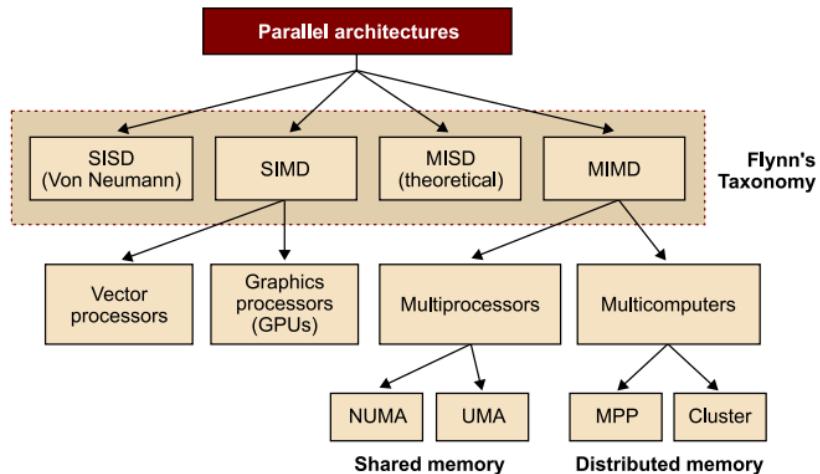
Level	Food	Relative access time
Level-1 cache	Put food in mouth	Fractions of a second
Level-2 cache	Take food from a plate	1 second
Level-3 cache	Take food from the table	A few seconds
DRAM	Get food from the kitchen	A few minutes
FLASH	Get food from the local store	A few hours
Hard disk	Get food from Mars	3-5 years

- Tomando todo esto en cuenta, una de las soluciones que se está adoptando es expandir la jerarquía de memoria con un nuevo nivel ocupado por la memoria no volátil



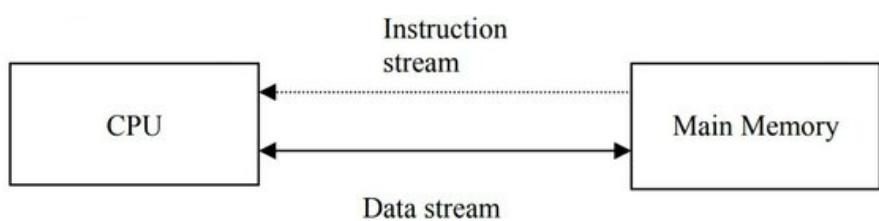
## El paralelismo y las aplicaciones en paralelo: arquitecturas en paralelo

- En la computación en paralelo, la taxonomía de Flynn se usa para clasificar las arquitecturas computacionales. Un sistema se clasifica basado en el número de flujos de datos y flujos de instrucciones que puede manejar simultáneamente, y los dos tipos fundamentales de arquitecturas son la SIMD y la MIMD



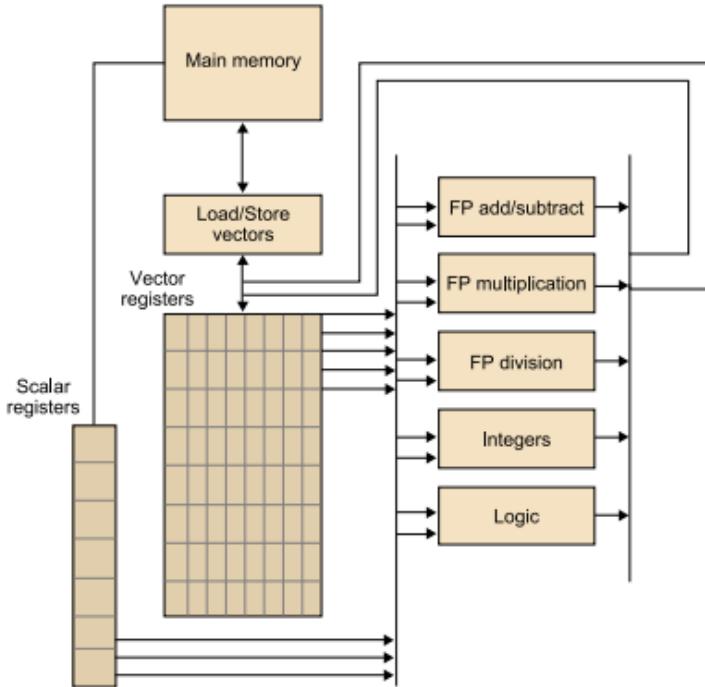
- Antes de poder explicar los esquemas de paralelismo de la taxonomía de Flynn, es necesario entender algunos conceptos clave sobre la CPU
  - El contador del programa o *program counter* (PC) es un registro especial que toma nota o sigue la dirección de memoria de la siguiente instrucción en un programa. Este componente es importante porque permite que la CPU obtenga instrucciones de la memoria de manera secuencial (la CPU sabe qué instrucción obtener después)

- Un ciclo de reloj o *clock cycle* (también llamado ciclo de máquina) es la unidad de tiempo básica de una CPU, la cual representa una operación completa de la CPU incluyendo la obtención, la decodificación, la ejecución y el almacenamiento de los datos. Este ciclo funciona a través de una señal de un reloj interno dentro de la CPU, la cual sincroniza las operaciones de diferentes componentes para que trabajen de manera coordinada
- La taxonomía de Flynn utiliza el concepto de flujo de datos y flujo de instrucciones en vez de utilizar conceptos más granulares debido a que los programas suelen contener una secuencia de datos e instrucciones sobre los cuales se opera



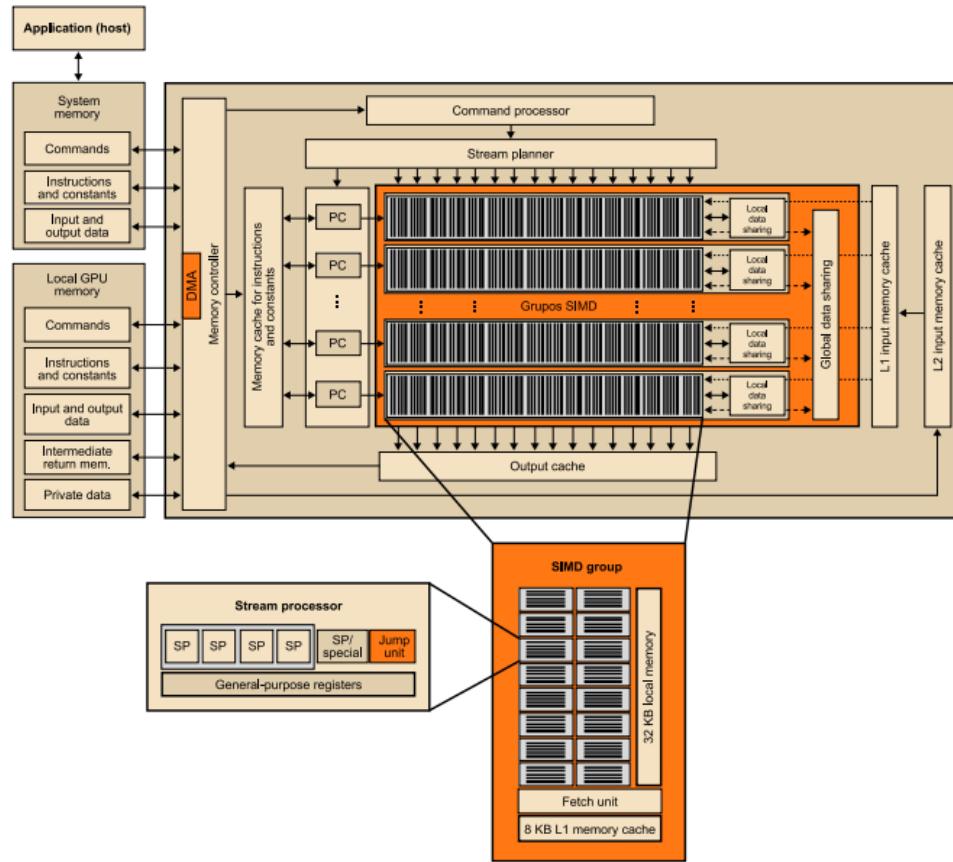
- En el flujo completo de la ejecución de instrucciones de un programa sobre unos datos, se establece un flujo de instrucciones de la memoria principal a la CPU, la cual recibe una secuencia de instrucciones que tiene que utilizar
- Similarmente, hay un flujo bidireccional de operandos entre el procesador y la memoria (se obtienen nuevos datos al ejecutar las instrucciones a partir de viejos datos obtenidos de la memoria) que se conoce como flujo de datos
- En un sistema de una instrucción única y múltiples datos o *single instruction multiple data* (SIMD), una sola instrucción se aplica a varios conjuntos de datos, lo cual es lo mismo que tener una sola unidad de control y múltiples unidades de lógica aritmética o *arithmetic logic units* (ALUs)
  - Una unidad de control, en la arquitectura de Von Neumann, es una parte de la CPU que se encarga de organizar la implementación consistente de algoritmos de decodificación de instrucciones que provienen de la memoria del dispositivo, responder a situaciones de emergencia y realizar funciones de dirección general de todos los nodos de computación
  - Una instrucción se envía desde la unidad de control a todos los ALUs, y cada ALU aplica la instrucción a su conjunto de datos o se mantiene sin trabajo si no hay datos asociados

- Estos sistemas SIMD son ideales para ciertas tareas, tales como paralelizar bucles simples que operan en vectores de datos grandes. Este tipo de paralelismo se conoce como paralelismo de datos
- El problema principal con este tipo de sistemas es que no siempre funcionan bien para todos los tipos de problemas. Por lo tanto, aunque la mayoría de supercomputadores se basaron en sistemas SIMD, al final estos sistemas pasaron a ser procesadores vectoriales
- Recientemente los procesadores de alta consumición o *high-consumption processors* y las unidades de procesamiento gráfico o *graphics processor units* (GPUs) han estado utilizando aspectos de la arquitectura SIMD
- Aunque el significado de lo que constituye un procesador vectorial ha cambiado durante los años, la característica principal es que estos operan sobre vectores de datos, mientras que los procesadores convencionales operan sobre elementos o escalares individuales
  - Este tipo de procesadores son rápidos y fáciles de usar para muchos tipos de aplicaciones. Los compiladores que vectorizan el código son muy buenos en identificar código que se puede vectorizar y bucles que no pueden ser vectorizados
  - Los sistemas vectoriales tienen una gran banda ancha de memoria y todos los datos cargados se usan. Esto es contrario a los sistemas basados en cachés de memoria, que no usan todos los elementos de la línea de caché de memoria
  - El problema de los sistemas vectoriales es que no pueden trabajar con estructuras de datos irregulares, y por lo tanto tienen limitaciones significativas con respecto a la escalabilidad (no pueden gestionar problemas muy grandes)
- Los procesadores vectoriales tienen las siguientes características, mostradas en el esquema:



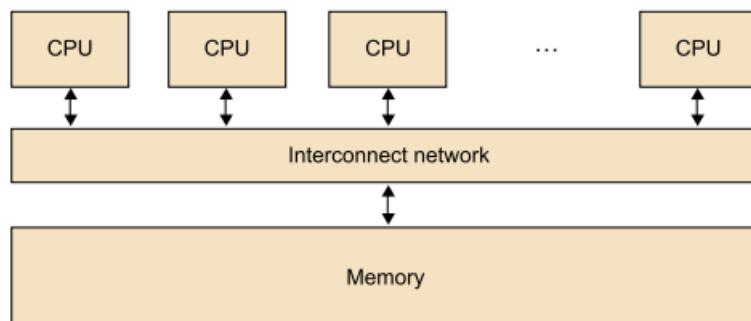
- Tienen registros vectoriales, que son registros capaces de guardar vectores operantes y operar simultáneamente sobre sus contenidos. El sistema que establece el tamaño del vector, que puede variar de 4 a 128 elementos de 64 bits, por ejemplo
- En los procesadores vectoriales se tienen unidades funcionales vectoriales, por lo que la misma operación se aplica a cada elemento de los vectores, o en operaciones tales como la suma de vectores, la misma operación se aplica a cada par de elementos de los dos vectores solo una vez. Estas operaciones son, por tanto, SIMD en cuanto al paralelismo usado se refiere
- Las instrucciones sobre vectores son instrucciones que operan en vectores en vez de en escalares. Esto significa que, en vez de realizar las operaciones individualmente para cada elemento del vector (por ejemplo, cargar, añadir, guardar, etc.), esto se puede hacer por bloques (a la vez, y no secuencialmente)
- En los procesadores vectoriales se tiene acceso a la memoria por intervalos, de modo que, con este tipo de acceso, el programa accede a elementos del vector localizados en intervalos. Por ejemplo, acceder al segundo elemento, el sexto y el décimo reflejaría un acceso a la memoria en un intervalo de 4
  - Los procesadores gráficos o GPUs tradicionalmente funcionaban usando un *pipeline* de procesamiento, que se construye de varias fases que son muy especializadas en término de las funciones que realizan, y que se

ejecutan en un orden preestablecido. Cada fase del *pipeline* recibe el resultado de la fase previa y proporciona un resultado a la siguiente

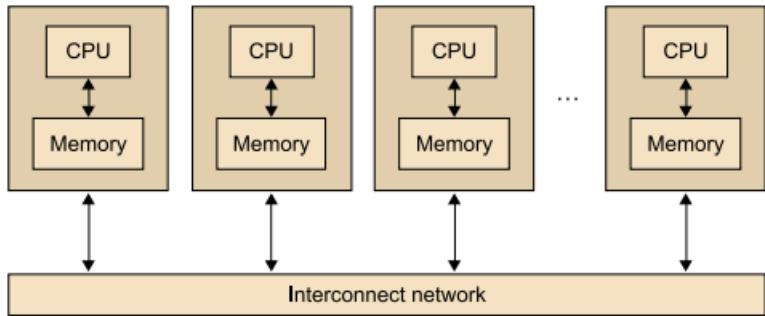


- La implementación a través de una estructura de *pipeline* permite al procesador gráfico ejecutar múltiples operaciones en paralelo. Como este *pipeline* es específico para la gestión de gráficos, normalmente se denomina *pipeline* gráfico o de renderización
- La renderización consiste en proyectar una representación tridimensional en una imagen bidimensional (que es el propósito del procesador gráfico). Hay fases del *pipeline* gráfico que se pueden programar y, aunque las GPUs actuales estén orientados a un uso general, hay una flexibilidad de programación alta proporcionada por lo que se conoce como *kernels*
- Los *kernels* son normalmente secciones cortas de código en donde el paralelismo es implícito dado que, cuando los *kernels* se ejecutan, son responsables del procesamiento de la porción de datos asignada a estos. En verdad, las GPUs pueden usar paralelismo SIMD para mejorar el rendimiento, y las generaciones actuales de GPUs están usando esta arquitectura a través de poner un gran número de ALUs en cada núcleo

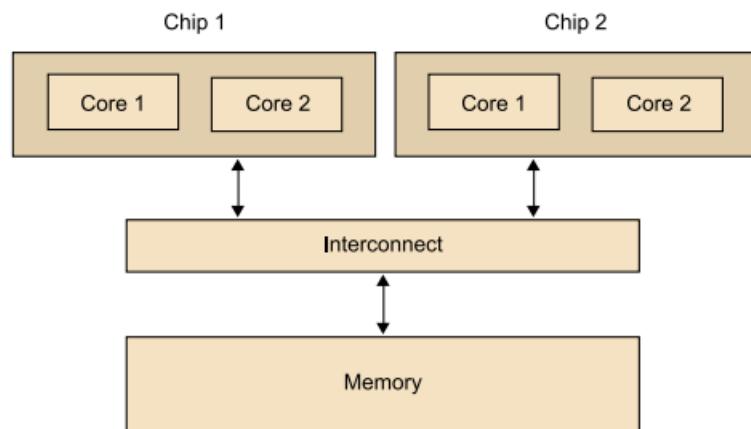
- Los procesadores gráficos están siendo muy populares para la programación de alto rendimiento, y varios lenguajes de programación se han desarrollado para hacer más fácil que los usuarios programen estos procesadores
- Un sistema de múltiples instrucciones y múltiples datos o *multiple instructions multiple data* (MIMD) consiste de un conjunto de unidades de procesamiento o núcleos completamente independientes, que tienen su propia unidad de control y su propia ALU
  - En contraste a los sistemas SIMD, los sistemas MIMD normalmente son asíncronos u operan por ellos mismos. Muchos sistemas MIMD tampoco tienen un reloj global, y puede no haber relación entre los tiempos de sistema de dos procesadores diferentes
  - Debido a que el programador no impone ninguna sincronización específica, puede ser posible que, aunque los procesadores estén ejecutando la misma secuencia de instrucciones, en cualquier punto del tiempo, pueden estar ejecutando partes diferentes de este
- Hay dos tipos principales de sistemas MIMD, los sistemas de memoria compartida y los de memoria distribuidos
  - En un sistema de memoria compartida, un conjunto de procesadores autónomos está conectado a la memoria del sistema a través de una red interconectada y cada procesador tiene acceso a cualquier parte de la memoria

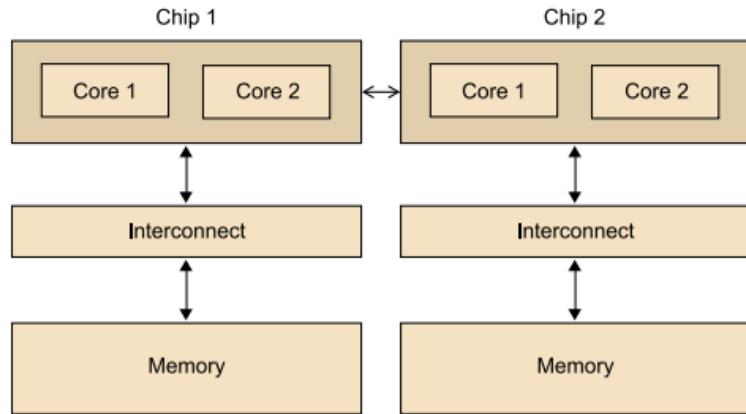


- En un sistema de memoria distribuida cada procesador está vinculado a su propia memoria privada y los conjuntos de procesador-memoria se comunican a través de una red interconectada. Esto significa que, en un sistema de memoria distribuido, los procesadores típicamente se comunican entre ellos explícitamente enviando mensajes o usando funciones especiales que proporcionan acceso a la memoria de otro procesador

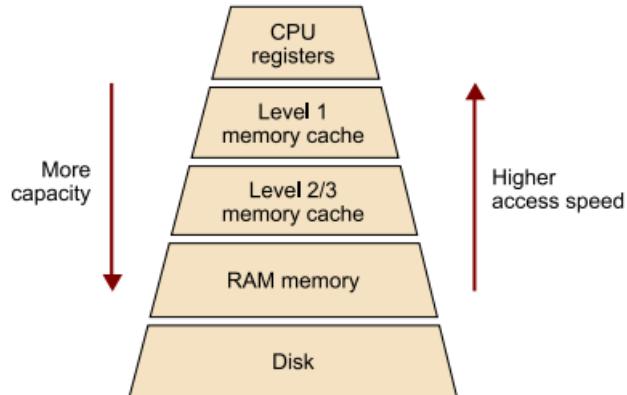


- Los sistemas de memoria compartida más usados usarán uno o más procesadores de múltiples núcleos, que usan uno o más CPUs en un solo chip. Los núcleos normalmente tienen una memoria caché privada de nivel 1, mientras que los otros cachés de memoria pueden o no ser compartidos entre los diferentes núcleos
  - En un sistema de memoria compartida con múltiples procesadores de múltiples núcleos, la red interconectada puede conectar a todos los procesadores directamente con la memoria principal (acceso uniforme a memoria o *uniform memory access*), o cada procesador puede tener acceso directo a un bloque de la memoria principal (acceso no uniforme a memoria o *non-uniform memory access*). Los procesadores pueden también acceder a los bloques de memoria de otros procesadores usando *hardware* especializado incorporado en los procesadores





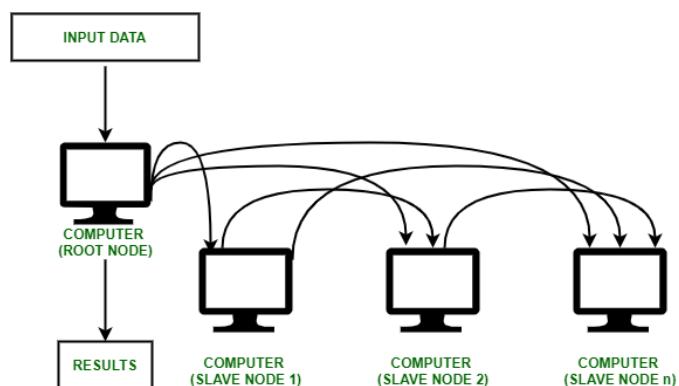
- En el primer tipo de sistema, el tiempo requerido para acceder a cualquier tipo de localizaciones de memoria es el mismo para todos los núcleos, mientras que, en el segundo tipo, el tiempo requerido por un núcleo para acceder a la memoria conectada directamente será menor que el tiempo necesario para acceder a la memoria de otro chip
- Los sistemas UMA son más fáciles de programar, dado que el programador no se tiene que preocupar del tiempo requerido para acceder a los datos y, por tanto, no se tiene que preocupar sobre la localización de los datos. Esta ventaja se balancea en cierto grado con el hecho que los sistemas NUMA proporcionan un acceso a la memoria más rápido (a la memoria conectada directamente), y estos sistemas tienden a tener una mayor cantidad de memoria que los sistemas UMA
- Por lo tanto, se puede decir que la jerarquía de la memoria (y su gestión eficiente) es un aspecto muy importante de la computación de alto rendimiento. Cuanto más rápido es el acceso a una memoria, menor y más cara tiende a ser



- La existencia de diferentes niveles de memoria implica que el tiempo dedicado a realizar una operación de acceso a memoria

en un nivel dado incrementa con la distancia al nivel más rápido, que es el acceso a los registros del procesador

- Una gestión pobre de la memoria causa problemas, llevando al uso excesivo de la memoria virtual del sistema y a un acceso costoso al disco. Esto significa que diseñar aplicaciones eficientes requiere conocimiento completo de la estructura de la memoria de una computadora y cómo explotar esta
  - Optimizar el proceso de cálculo secuencial involucra, entre otras cosas, la selección apropiada de las estructuras de datos usadas para representar la información relacionada al problema. Por ejemplo, esto puede incluir el uso de matrices escasas o *sparse matrices* para guardar en memoria solo los elementos no nulos y también seleccionar los métodos más eficientes para resolver el problema
  - Registrar datos al disco es una fase importante de las aplicaciones, especialmente para aquellas orientadas a procesos de simulación. Como acceder al disco duro requiere tiempos de acceso más grandes a aquellos correspondientes a la memoria RAM, es esencial hacer una gestión eficiente del proceso de *input-output* con tal de minimizar el impacto en el proceso de simulación
  - Finalmente, utilizar computación en paralelo está por encima de todas las capas, donde los tiempos de ejecución se pueden reducir para aplicar una buena estrategia para dividir las tareas. Esto asegura una distribución balanceada de la carga entre los procesadores involucrados, y también minimiza el número de comunicaciones que se realizan entre ellos
- Los sistemas de memoria distribuida más usados y populares son aquellos conocidos como clústeres. Estos se constituyen de un conjunto de sistemas de *commodities*, tales como servidores, conectados a una red interconectada como Ethernet



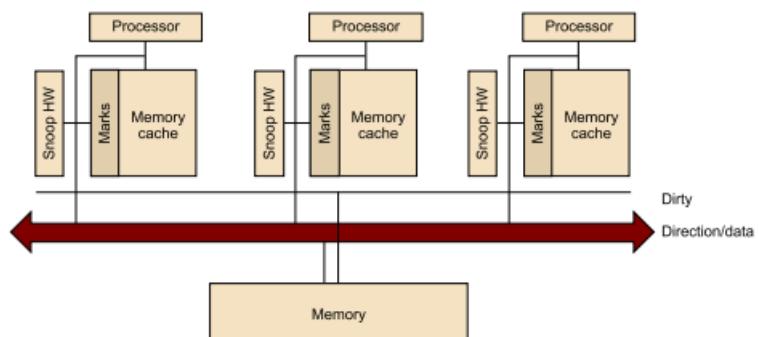
- Los nodos de estos clústeres, que son unidades computacionales individuales interconectadas a través de una red, son usualmente sistemas de memoria compartida con uno o más procesadores de múltiples núcleos. Estos ocasionalmente se refieren como sistemas híbridos, para diferenciarlos de aquellos de memoria puramente distribuida
- Un clúster está hecho de nodos de memoria compartida, y se entiende que los grandes sistemas de computación paralela son clústeres por definición
- La principal diferencia entre supercomputadores y grandes centros de procesamiento o *datacentres* usados en la industria es la red interconectada: dado que las aplicaciones que requieren supercomputación están estrechamente acopladas, un aspecto clave de estos sistemas es su capacidad de ofrecer una latencia muy pequeña en su red de interconexión, por lo que los mensajes pueden viajar muy rápidamente. En otras palabras, los procesos que se están ejecutando en diversos núcleos distribuidos tienen dependencias de datos con todos los procesos y deben comunicarse frecuentemente usando mensajes
- Se tiene que recordar que los cachés de memoria se gestionan por los sistemas de *hardware*, de modo que los programadores no tienen control directo sobre ellos. Esto tiene consecuencias significativas para sistemas de memoria compartida
  - Planteando una situación en la que se tiene un sistema de memoria compartida con solo dos núcleos que tienen su propio caché de memoria para los datos, no habría problema mientras que ambos núcleos solo lean datos compartidos
    - Si se tiene una variable compartida  $x$  que se ha fijado a 2, y  $y_0$  es privada y perteneciente al núcleo 0 e  $y_1$  y  $z_1$  son privadas también y pertenecen al núcleo uno, entonces uno puede verse envuelto en la siguiente problemática:

Time	Core 0	Core 1
0	$y_0 = x;$	$y_1 = 3 * x;$
1	$x = 7;$	Code that does not involve $x$
2	Code that does not involve $x$	$z_1 = 4 * x;$

- La localización en la memoria  $y_0$  acabaría con un valor de 2 y la localización en la memoria  $y_1$  acabaría con un valor de 6, pero no está claro qué valor debería tener  $z_1$ . Puede parecer que, debido

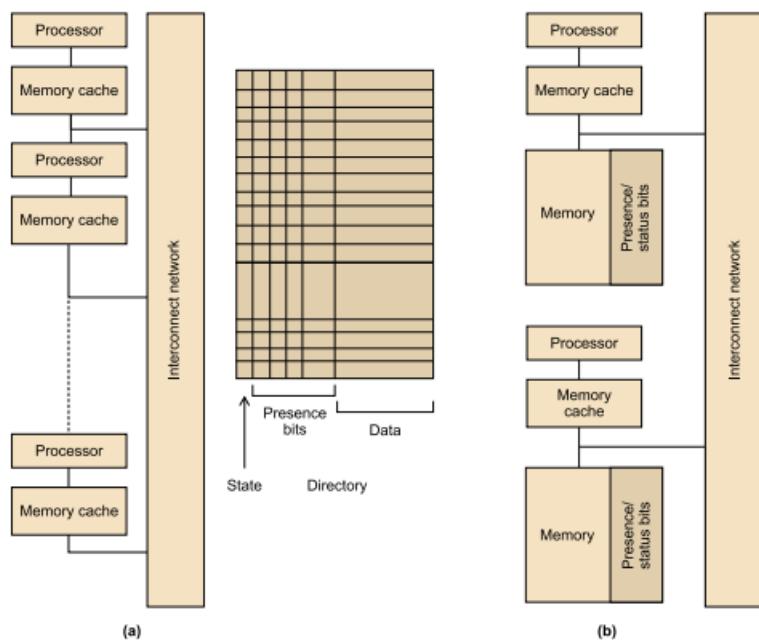
a que el núcleo cero actualiza el valor de  $x$  a 7 antes,  $z1 = 28$ , pero a no ser que  $x$  se saque de la memoria caché del núcleo 0 y se asigne otra vez al caché de memoria del núcleo 1, el valor original  $x = 2$  se mantiene (estaba en el caché de memoria del núcleo 1) y  $z1 = 8$

- Esto, obviamente, es un problema significativo, dado que como el programador no tiene control directo sobre cuando los cachés de memoria se actualizan, el programa puede no asumir el valor de  $z1$ 
  - En sistemas con un solo procesador, no hay mecanismos para asegurar que cuando los cachés de memoria guardan la misma variable, una modificación hecha por uno de los procesadores a esta en el caché de memoria será vista por el resto de procesadores
  - Los valores en los cachés de memoria de otros procesadores también necesitan ser actualizados, y esto es por lo que se habla de un problema de coherencia de caché de memoria
- Hay dos alternativas para asegurar coherencia en el caché de memoria: la técnica de *snooping* y la técnica del directorio
  - La idea del *snooping* se fundamenta en los sistemas de buses (canales que transfieren datos entre componentes): cuando los núcleos comparten un bus, cualquier señal transmitida por el bus se puede ver por todos los núcleos conectados al bus. Por lo tanto, cuando el núcleo 0 actualiza la copia de  $x$  en su caché de memoria, si esta información también se distribuye por el bus y el núcleo 1 está escuchando, el núcleo también puede ver que se ha actualizado  $x$  y guardar una marca propia de  $x$  como invalido



- Esto es esencialmente como la consistencia por *snooping* funciona, aunque en su implementación real la información de cualquier cambio se distribuye notificando a los otros núcleos de que la línea en el caché de memoria que contiene  $x$  ha cambiado, pero no se actualiza  $x$

- El uso de la técnica de *snooping* no es asequible para grandes redes interconectadas, dado que requiere que los datos se distribuyan por la red cada vez que hay un cambio, lo cual no es escalable por la degradación del rendimiento que provocaría
- En cambio, los protocolos basados en directorio han intentado resolver el problema usando una estructura de datos llamada directorio, la cual se distribuye de manera que cada par de procesador-memoria es responsable de guardar la parte de la estructura correspondiente al estatus de las líneas de cachés de memoria en su memoria local. Cuando una variable se actualiza, se consulta al directorio y los controladores de caché de memoria de los núcleos que tienen una línea donde la variable aparece se invalidan



- Esta técnica obviamente requiere de más espacio de memoria por el directorio, pero, cuando una variable en el caché de la memoria se actualiza, solo los núcleos que guardan esta variable necesitan ser contactados. Esto significa que la comunicación en las redes se puede reducir y así aumentar el rendimiento, comparada con la técnica de *snooping*
- Es importante recordar de que los cachés de memoria se implementan en el *hardware*, y por lo tanto operan en las líneas de caché de memoria y no en variables individuales. No obstante, esto puede tener consecuencias catastróficas en relación al rendimiento

- Planteando una situación en la que se quiere repetir una función  $f(i,j)$  y añadir el valor devuelto en un vector, se podría usar el siguiente código:

```

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

```

- Se podría paralelizar esto dividiendo las iteraciones del bucle externo en varios núcleos. Teniendo  $num\_cores$  núcleos, se pueden asignar las primeras  $m/num\_cores$  iteraciones en el primer núcleo y las siguientes  $m/num\_cores$  en el segundo, y así

```

/* Private variables */
int i, j, num_iter;
/* Private variables initialized by a core */
int m, n, num_cores;
double y[m];
num_iter = m/num_cores;
/* Core 0 does the following */
for(i=0; i<num_iter; i++)
    for(j=0; j<n; j++)
        y[i] += f(i,j);
/* Core 1 does the following */
for(i=num_iter+1; i<2*num_iter; i++)
    for(j=0; j<n; j++)
        y[i] += f(i,j)
...
double y[m];
/* Assign y = 0 */
...

```

- Ahora se supone que se tiene un sistema de memoria compartida con dos núcleos,  $m = 8$ , el tipo de dato  $double$  es 8 bits, las líneas de la memoria de caché son de 64 bits y  $y[0]$  se guarda al principio de la línea de caché de memoria. Una línea de caché de memoria puede guardar 8 dobles e  $y$  ocupa toda una línea de memoria de caché
- Si los núcleos 0 y 1 ejecutan el código de manera simultánea, y como  $y$  se guarda en una sola línea de memoria de caché, cada vez que uno de los núcleos ejecute  $y[i] += f(i,j)$ , la línea será invalidada y la siguiente vez que la memoria intente ejecutar esta línea, tendrá que tomar la línea otra vez de la memoria principal. Por lo tanto, si se hacen  $n$  iteraciones, se puede asumir que un gran porcentaje de veces que  $y[i] += f(i,j)$  se ejecute, se accederá a la memoria principal, aunque ambos núcleos nunca accedan a los elementos de  $y$  correspondientes al otro núcleo
- Esto se conoce como **compartición falsa**, dado que el sistema se comporta como si los elementos de  $y$  se estuvieran

compartiendo entre los núcleos. Se debe destacar que esto no causa resultados erróneos, sino que baja el rendimiento por el acceso repetido a la memoria principal

- Esto se puede mitigar usando almacenamiento temporal (local en el *stream* del proceso) y copiando ese almacenamiento temporal en el almacenamiento compartido. También es posible aplicar varias otras técnicas, como el *padding*
- El modelo usado para la computación de altas prestaciones tradicionalmente ha sido el que se ha visto de arquitecturas paralelas, pero, en la última década, los sistemas distribuidos que se han desarrollado parecen una fuente viable para ciertas aplicaciones de alto rendimiento
  - Este tipo de aplicaciones de alto rendimiento son tales como los sistemas *grid*. Los sistemas *grid* proporcionan la infraestructura necesaria para transformar redes de computadoras geográficamente dispersas en un sistema unificado de memoria compartida a través de Internet
    - En general, estos sistemas son muy heterogéneos, dado que los nodos individuales en los que están construidos pueden ser diferentes tipos de *hardware* o ejecutar diferentes tipos de *software*
    - Debido a esto, un nivel intermedio de *software* llamado *middleware* se usa para permitir que la infraestructura del sistema *grid* proporcione diferentes interfaces y abstracciones necesarias cuando se trabaja con computadores distribuidos de diferentes instituciones de manera transparente, como si fueran un sistema convencional
  - Otros tipos de sistemas altamente distribuidos que se han desarrollado son los sistemas *peer-to-peer* o P2P y la computación en la nube o *cloud computing*, aunque estos no se diseñaron inicialmente para la computación de alto rendimiento
- La mayoría del *software* que se ha desarrollado para sistemas convencionales con un solo núcleo no se puede usar para múltiples núcleos de los procesadores más nuevos. Aunque se podría dejar al sistema operativo ejecutar en varios procesadores, esto no es suficiente y menos cuando se requiere paralelismo masivo, de modo que las soluciones primarias suelen ser reescribir los programas o usar herramientas para automáticamente parallelizar estos programas
  - La manera en la que los programas se escriben depende de la manera en la que uno quiera dividir las tareas. Hay dos maneras principales de hacer esto: a través de paralelismo a nivel de tarea o de paralelismo a nivel de datos

- Con paralelismo a nivel de tarea, el trabajo se divide en diferentes tareas y se distribuye entre los varios núcleos. Y con paralelismo a nivel de datos, el problema se resuelve dividiendo los datos entre los múltiples núcleos, que realizan operaciones similares en los datos que les pertenece
- Actualmente, los programas paralelos más potentes se escriben utilizando construcciones explícitas de paralelismo y extensiones de lenguajes como C/C++
  - Estos programas incluyen instrucciones 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. Esto significa que los programas suelen acabar siendo muy complejos
  - Existen otras opciones para escribir programas paralelos, como el uso de lenguajes de alto nivel. Sin embargo, éstos tienden a sacrificar el rendimiento para facilitar el desarrollo y mejorar la productividad
- Se hará el enfoque explícitamente en programas paralelos. Los principales modelos de programación son: la interfaz de paso de mensajes o MPI, los streams (por ejemplo, POSIX streams o Pthreads) y OpenMP
  - MPI y Pthreads son bibliotecas con definiciones de tipos, funciones y macros que pueden utilizarse, por ejemplo, en programas escritos en C
  - OpenMP consiste en una biblioteca junto con ciertas modificaciones para el compilador, por ejemplo, para C. También pueden manejar otros modelos de programación, como los utilizados para computación gráfica o los que proporcionan abstracciones más modernas
- Estos modelos de programación reflejan los dos tipos principales de sistemas paralelos: los sistemas de memoria compartida y los sistemas de memoria distribuida
  - En un sistema de memoria compartida, los núcleos pueden compartir el acceso a la memoria, lo que significa que el acceso a la memoria por parte de los distintos núcleos debe coordinarse y el espacio de memoria compartida debe examinarse y actualizarse. En un sistema de memoria distribuida, cada núcleo dispone de su propio espacio de memoria privado, y los núcleos

deben comunicarse explícitamente enviando mensajes a través de la red de interconexión

- Pthreads y OpenMP se diseñaron para programar sistemas de memoria compartida y proporcionan mecanismos para acceder a ubicaciones de memoria compartida. En cambio, MPI se diseñó para programar sistemas de memoria distribuida y proporciona mecanismos para intercambiar mensajes
- Ahora es posible analizar y estudiar modelos de programación de memoria compartida, tales como la programación en *streams*, OpenMP, CUDA y OpenCL
  - Aunque diferentes tipos de *streams* se pueden considerar para programar programas en paralelo, tales como *streams* de UNIX o Java, uno se puede enfocar más en Pthreads
    - Esta es una librería que se ajusta a los estándares POSIX y que permite trabajar con varios *streams* a la vez, de modo que se pueden usar varios *threads* para usar múltiples *streams* en paralelo
    - Los Pthreads son más efectivos en un multiprocesador o sistema de multinúcleos, donde el *stream* o *thread* de ejecución se puede planear en ejecutarse en diferentes procesadores o núcleos, produciendo así una ganancia de velocidad por el paralelismo
    - Pthreads define un conjunto de tipos, funciones y constantes en lenguaje C para poder programar usando diversos *threads*
  - Los Pthreads implican una sobrecarga mucho menor que la creación de un nuevo proceso (por ejemplo, los llamados *fork* o *exec*), ya que el sistema no inicializa un nuevo espacio de memoria virtual ni un nuevo entorno para el proceso

```
#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;
    /* Create independent flows that execute the function */
    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
```

```

    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
    /* Wait for all flows to finish before continuing with the main program */
    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);
}

```

- Aunque estos *streams* son muy efectivos en sistemas con múltiples procesadores o núcleos, los beneficios también se pueden obtener en un sistema de procesador único, como la latencia de *input/output* se puede explotar junto con las otras características de sistemas que pueden parar la ejecución de un proceso en ejecución
- Los Pthreads se utilizan en sistemas de memoria compartida y todos los *streams* de un proceso comparten el mismo espacio de direcciones. Para crear un *stream* o *thread*, se debe definir una función y sus argumentos que serán utilizados por él
- El principal objetivo del uso de Pthreads es conseguir mayor velocidad (mejor rendimiento) de las aplicaciones
- Uno de los elementos clave para gestionar los *streams* de Pthreads es su sincronización y exclusión mutua, que normalmente se realiza mediante semáforos
- Aunque tanto Pthreads y OpenMP son interfaces de programación para memoria compartida, hay muchas diferencias significativas entre ellos: Pthreads requiere que el programador especifique el comportamiento de cada *stream* explícitamente, mientras que OpenMP solo requiere que se especifique que bloque del código se ejecuta en paralelo
  - Determinar qué tareas y *threads* lo ejecutarán es trabajo del compilador y del sistema de ejecución, también conocido como *runtime*. Esto también muestra una diferencia importante: mientras que Pthreads es una biblioteca de funciones que se pueden añadir al ensamblar un programa en C, OpenMP requiere soporte a nivel de compilador, y por lo tanto no todos los compiladores de C compilarán programas OpenMP como programas paralelos
  - Por lo tanto, se puede decir que OpenMP permite mejorar la productividad utilizando una interfaz de mayor nivel que la de

Pthreads, aunque, por otro lado, no permite controlar ciertos detalles relacionados con el comportamiento de los *streams*

- OpenMP fue concebido por un grupo de programadores e informáticos que pensaban que era demasiado difícil escribir programas a gran escala y de alto rendimiento utilizando interfaces como Pthreads
  - De hecho, OpenMP se diseñó explícitamente para permitir a los programadores parallelizar programas secuenciales existentes de forma progresiva, en lugar de tener que reescribirlos desde el principio
- OpenMP proporciona una interfaz de memoria compartida basada en "directivas". Por ejemplo, en C/C++ esto significa que hay ciertas instrucciones especiales para el preprocesador conocidas como "pragmas"
  - Estos pragmas suelen añadirse a un programa para especificar comportamientos que no forman parte de las especificaciones propias del lenguaje de programación
  - Esto significa que los compiladores que no entienden estos pragmas pueden simplemente ignorarlos, y esto es lo que permite que un programa que utiliza pragmas pueda ejecutarse en un sistema que no los soporta
- Los pragmas en C/C++ comienzan con #pragma, al igual que otras directivas de procesamiento. El siguiente ejemplo muestra un programa OpenMP muy sencillo que produce la frase "hola, mundo":

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Hello(void);
int main(int argc, char* argv[]) {
    /* Get the number of flows of the command-line interpreters */
    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);
}
```

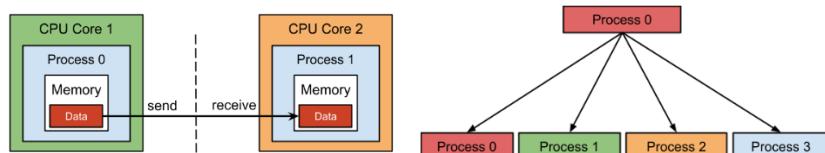
- En este ejemplo, podemos observar el uso de una directiva OpenMP (o pragma) y también algunas otras características típicas asociadas a OpenMP

- OpenMP utiliza un modelo de *fork-join* para la ejecución en paralelo: todos los programas OpenMP comienzan con un solo *thread* maestro que se ejecuta secuencialmente hasta que se encuentra una región en paralelo (pragma), y crea un equipo de *threads* paralelos (paso *fork*). Cuando el equipo completa la región paralela, se sincronizan y se terminan, dejando solo el *thread* maestro que ejecuta secuencialmente (paso *join*)
- La especificación propietaria CUDA (de *compute unified device architecture*) desarrollada por NVIDIA principalmente se inició como una plataforma para sus productos de computación gráfica (GPU). CUDA incluye especificaciones para la arquitectura y un modelo de programación asociado
  - Desde la perspectiva del programador, el sistema consta de un procesador principal o *host*, que es una CPU tradicional (por ejemplo, un procesador con arquitectura Intel) y uno o más dispositivos que son GPUs
  - La arquitectura CUDA pertenece al modelo SIMD, y por tanto se diseñó para aprovechar el paralelismo a nivel de datos, lo que significa que un conjunto de operaciones matemáticas se puede ejecutar en un conjunto de datos simultáneamente. Afortunadamente, muchas aplicaciones tienen partes con un nivel de paralelismo muy alto a nivel de datos
- Un programa en CUDA consiste de una o más fases, que se puede ejecutar por un procesador principal (CPU) o un dispositivo (GPU)
  - Las fases con poco o sin paralelismo a nivel de los datos son implementadas en el código que se va a ejecutar en el procesador principal, mientras que las fases con un mayor nivel de paralelismo a nivel de datos se implementan en el código que se va a ejecutar en el dispositivo
  - La programación en CUDA requiere de escribir un código *host*, el cual se ejecuta en la CPU y gestiona la transferencia de datos entre el CPU y el GPU, y un código del dispositivo, que se ejecuta en paralelo en los núcleos del GPU y es el código a ejecutar en paralelo. El código *host* lanza los *kernels* de los códigos de los dispositivos para que se ejecuten en el GPU
- Los elementos fundamentales de CUDA son el modelo de memoria, la organización de los *streams* y las funciones *kernels*

- Sobre el modelo de memoria, es importante mencionar que el procesador principal y la memoria del dispositivo son espacios de memoria completamente separados (aunque algunas arquitecturas y modelos modernos incluyen memoria compartida entre el *host* y el dispositivo). Esto refleja el hecho de que los dispositivos son usualmente tarjetas que tienen su propia memoria DRAM
- Para ejecutar un *kernel* en el dispositivo GPU se tiene que reservar la memoria en el dispositivo, transferir los datos necesarios del procesador principal al espacio de memoria asignado del dispositivo, ejecutar el *kernel*, transferir los datos con los resultados al procesador principal y soltar la memoria en el dispositivo cuando la ejecución del *kernel* finaliza (si ya no se necesita)
- En CUDA, un *kernel* se ejecuta a través a través de un conjunto de *streams* (por ejemplo, un vector o matriz de *streams*). Como todos los *streams* se ejecutan en el mismo *kernel* (como en un modelo SIMD), se necesita un mecanismo para permitir que se diferencien, de modo que la porción correspondiente de los datos se pueda asignar a cada *stream* de ejecución (CUDA incorpora palabras clave para referirse al índice para un *stream*)
- El código que se está ejecutando en el dispositivo (*kernel*) es la función que ejecuta varios *streams* durante la fase paralela cada uno dentro de su correspondiente rango de datos. Se debe tener en cuenta que CUDA sigue un modelo de un solo programa y múltiples datos (SPMD), y que todos sus *streams* ejecutan el mismo código

```
__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, *open-source* y multiplataforma para la programación en paralelo, cuya motivación principal de desarrollo fue la necesidad de simplificar la tarea de programar de manera portátil y eficiente para un creciente número de plataformas heterogéneas (como CPUs multinúcleo o GPUs)
  - OpenCL fue concebido por Apple, aunque el grupo Kronos completó su desarrollo (el mismo grupo que promovió OpenGL y que fue responsable de él)
  - OpenCL consiste de tres partes: la especificación del lenguaje multiplataforma, una interfaz a escala del entorno computacional y una interfaz para coordinar la computación en paralelo entre procesadores heterogéneos. OpenCL utiliza un subconjunto de C99 con extensiones para el paralelismo y usa un estándar de representación numérica IEEE 754 para asegurar la interpretabilidad entre plataformas
- También es posible analizar y estudiar modelos de programación de memoria distribuida, tales como la MPI y el espacio partido de dirección global (PGAS)
  - La MPI (de *message passing interface*) es una librería de intercambio de mensajes, y se puede usar esencialmente en programas de C y Fortran desde los que llamar a funciones MPI para la gestión de procesos y permitir que se comuniquen entre ellos
    - El *message-passing* es una técnica para ejecutar un programa en una computadora. El programa que llama envía un mensaje al proceso y confía en el proceso y su infraestructura subyacente para seleccionar y ejecutar el código apropiado



- Aunque MPI no es la única librería de intercambio de mensajes, es el estándar actual para el paradigma de memoria distribuida. Antes de MPI había modelos como PVM (máquina virtual paralela), aunque MPI ha acabado siendo la dominante
- Esta especificación fue desarrollada por el fórum MPI, que especificaron las características que librerías de este tipo deberían tener. Basadas en estas especificaciones, muchas empresas y desarrolladores han creado implementaciones específicas

- MPI ha conseguido una serie de metas importantes, tales como las siguientes:
  - La estandarización, dado que las implementaciones de las especificaciones se han vuelto estándar y ya no es necesario desarrollar diferentes programas para diferentes máquinas
  - La portabilidad, porque los programas MPI pueden operar en multiprocesadores de memoria compartida, multicomputadoras de memoria distribuida, clústeres de computadoras, y otros sistemas computacionales, siempre que haya una versión de MPI compatible con estos
  - El alto rendimiento, dado que los desarrolladores han desarrollado implementaciones eficientes para los equipos
  - El amplio rango de funciones, porque MPI incluye un gran número de funciones que se usan para llevar a cabo operaciones que tienden a aparecer en programas de intercambio de mensajes de una manera simple
- Cuando un programa de MPI se enciende o empieza, se crean muchos procesos que ejecutan el mismo código, pero cada uno tiene sus propias variables (modelo SPMD). A diferencia de OpenMP, no hay un proceso o *thread* maestro que controla a todos los otros

```
#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);/* Get the process identifier */
    MPI_Comm_size (MPI_COMM_WORLD, &size);/* Get the number of processes */
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

- La librería MPI se debe incluir arriba'
- Todos los *threads* ejecutan el mismo código desde el comienzo, de modo que todos tienen las mismas variables. A diferencia de OpenMP, estas variables son diferentes y pueden estar en diferentes memorias de múltiples procesadores

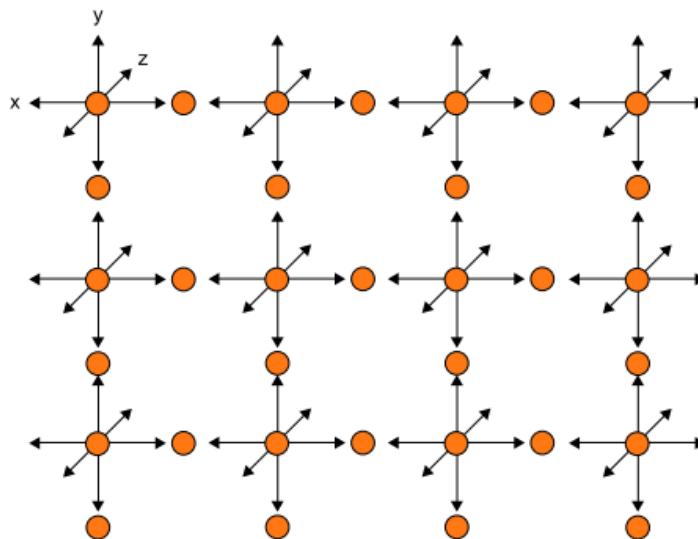
- Los *threads* trabajan independientemente hasta que MPI se inicializa usando la función *MPI\_Init*. Después de este punto, los procesos pueden colaborar intercambiando datos, sincronizándose, y otros
- La función *MPI\_Finalize* se llama cuando los *threads* o procesos no necesitan colaborar más entre ellos. Esta función deja ir a todos los recursos reservados para MPI
- Las funciones MPI tienen el formato *MPI\_Name(parameters)*. De la misma manera que OpenMP, los procesos o *threads* deben conocer su identificador (entre 0 y el número de subprocesos -1) y el número de procesos que se han comenzado (usando las funciones *MPI\_Comm\_rank* y *MPI\_Comm\_size*)
- Estas funciones tienen un parámetro *MPI\_COMM\_WORLD*, el cual es una constante de MPI que identifica el comunicador con los que se asocian todos los procesos. Un comunicador es un identificador para un grupo de procesos, y las funciones de MPI deben identificar en qué comunicador se dan las operaciones que se están llamando
- Los modelos de programación presentados anteriormente proporcionan soluciones específicas para sistemas de memoria distribuida y sistemas de memoria compartida. No obstante, muchos sistemas de altas prestaciones combinan sistemas de memoria distribuida (básicamente clústeres) y de memoria compartida (procesadores con múltiples núcleos)

## Las arquitecturas de sistemas HPC: datos y taxonomía

- Las arquitecturas de computación de altas prestaciones son las estructuras de los sistemas de computación para sistemas de altas prestaciones. Para poder entender su funcionamiento, primero es necesario entender la descomposición de datos y la descomposición funcional
  - La mayoría de las arquitecturas de alto rendimiento que se encuentran actualmente en el mercado permiten ejecutar más de un *thread* de ejecución de forma paralela
    - Algunas de ellas también permiten acceder a unidades de procesamiento específicas capaces de aplicar el mismo conjunto de operaciones a varios conjuntos de datos simultáneamente
  - Para obtener el máximo rendimiento de todas las prestaciones que ofrecen estas arquitecturas, tanto las aplicaciones como los modelos de programación utilizados deben adaptarse a sus características

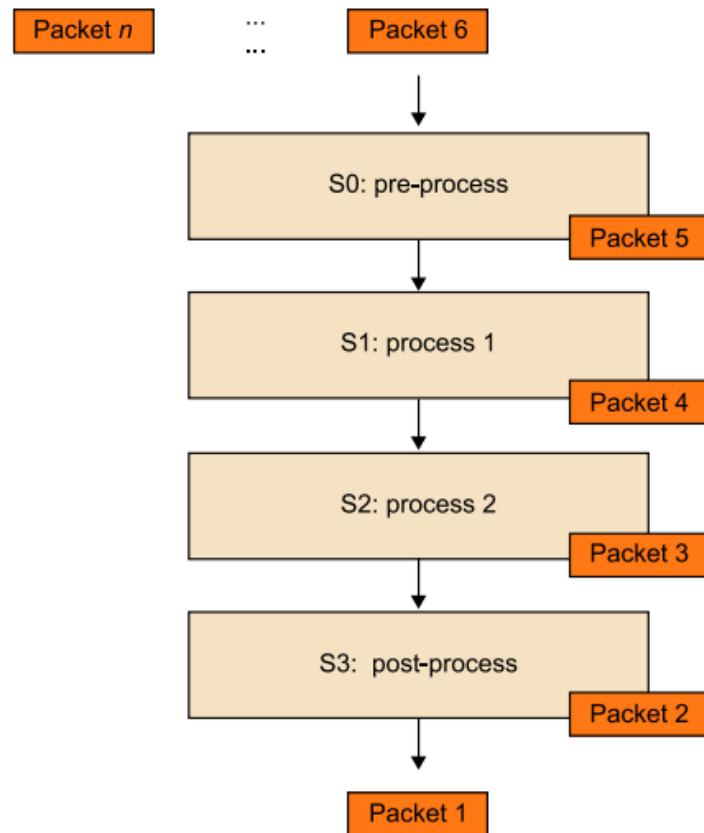
- Por un lado, los modelos de programación deben proporcionar a las aplicaciones formas de aprovechar explícitamente las fuentes de paralelismo. Por otro, los algoritmos que implementan las aplicaciones también deben adaptarse a ellas
- Para adaptar una aplicación a un modelo de programación orientado a arquitecturas paralelas, primero hay que comprender las características del modelo y la forma en que procesa los datos
- Asumiendo que los datos se pueden procesar independientemente, cuando se diseña una implementación paralela, cada procesamiento de estos puede ser descompuesto independientemente. A este proceso se le llama descomposición de datos

$$(x,y) = F(x - 1, x + 1, z - 1, z + 1, y - 1, y + 1)$$



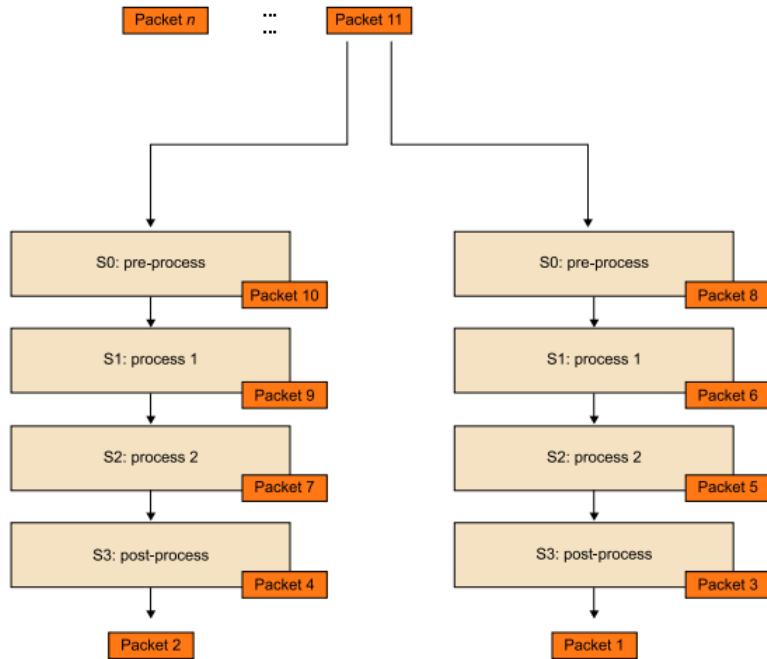
- El término de descomposición de datos también se puede aplicar a decisiones sobre cómo procesar los datos de un problema específico
- Esta descomposición es típicamente la tarea más compleja cuando se paralleliza una aplicación, dado que se tiene que tomar una decisión no solo sobre cómo los múltiples *threads* procesarán los datos, sino cómo se realizará el almacenamiento en los diferentes niveles de caché de memoria
- El segundo problema principal encontrado cuando se paralleliza una aplicación consiste en dividir el problema que se está intentando resolver en varias funciones en las cuales el problema se puede descomponer. A este proceso se le conoce como descomposición funcional

- La descomposición funcional del problema se refiere al proceso de definir qué funciones son aquellas definiendo el problema involucrado, cómo están relacionadas y cómo circularán los datos
- Se tiene que notar que, después de la descomposición funcional, puede ser que estas funciones se ejecuten en paralelo o no
- Cada elemento procesado por el algoritmo en cuestión pasa por cada una de las fases en las que se hace la descomposición. El siguiente esquema muestra un ejemplo de descomposición funcional:

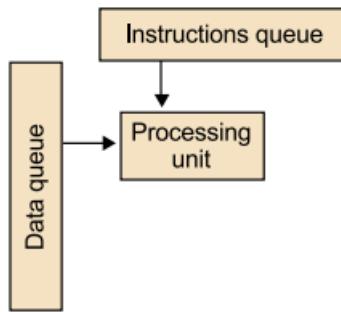


- Aunque se ha tratado la descomposición de datos y funcional como dos cosas separadas, en realidad, estos dos problemas están directamente relacionados
  - Cuando se analizan maneras de parallelizar una aplicación, tanto la descomposición funcional como la descomposición de datos deben estudiarse y, en general, el objetivo está en encontrar la combinación óptima de los dos
  - Por un lado, la descomposición funcional define los diferentes niveles de un algoritmo y cómo se pueden ejecutar independientemente, mientras que la descomposición de datos determina qué paquetes se pueden ejecutar

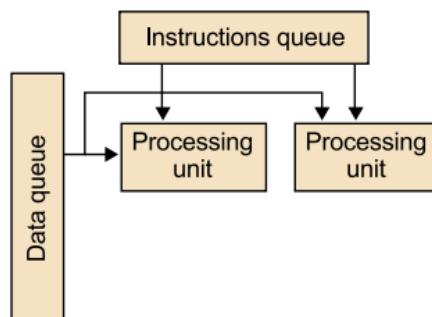
independientemente. En el siguiente esquema se muestran los dos tipos de descomposición:



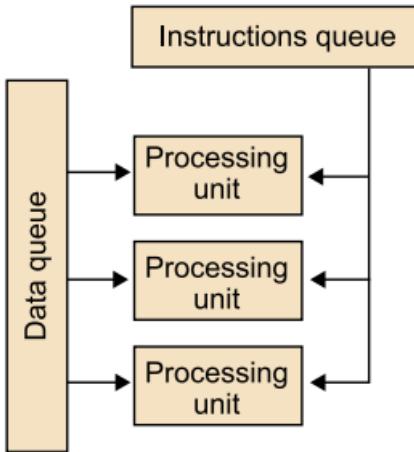
- Existen muchos tipos de arquitecturas computacionales, que van de procesadores con un solo *thread* de ejecución y un *pipeline* ordenado, a procesadores con cientos de núcleos y unidades computacionales que usan paralelismo
  - Flynn, en 1972, propuso un sistema de clasificación para arquitecturas computacionales en cuatro categorías. Este sistema clasifica los computadores según cómo manejan los flujos de datos e instrucciones
    - Aunque esta taxonomía se diseñó hace 50 años, la mayoría de procesadores actuales aún se pueden clasificar de esta manera. Además, muchos trabajos académicos en el campo siguen usando esta nomenclatura
    - Los cuatro tipos de arquitecturas que Flynn describe son la de *single instruction, single data* (SISD); la de *single instruction, multiple data* (SIMD); la de *multiple instructions, single data* (MISD); y la de *multiple instructions, multiple data* (MIMD)
  - Las arquitecturas SISD son arquitecturas tradicionales consistiendo de una sola unidad de procesamiento



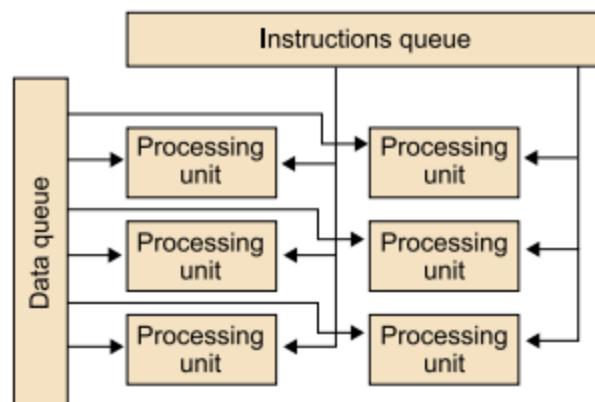
- Estas arquitecturas no se aprovechan del paralelismo a nivel de *threads* o de los datos: a cada elemento del conjunto de datos se le aplica una instrucción en cada momento
- Las arquitecturas SIMD son arquitecturas en las que una sola instrucción se ejecuta en paralelo por múltiples unidades de procesamiento, mientras que usa múltiples *streams* de datos



- Cada procesador tiene su propia memoria con datos, y, por tanto, aunque haya diferentes datos en general, la unidad de memoria y control de instrucciones del nodo o sistema se comparte
- El ejemplo más conocido de este tipo de arquitecturas son las unidades de procesamiento vectoriales. Las arquitecturas de este tipo son actualmente usadas en la mayoría de procesadores de altas capacidades, dado que permiten cálculos en grandes volúmenes de datos de manera paralela
- Esto hace posible incrementar el número de *flops* de los procesadores sustancialmente. Además, este paradigma computacional también se usan arquitecturas dedicadas al procesamiento de imágenes o gráficos
- Las arquitecturas MISD, esto permite que haya múltiples flujos de instrucciones que se ejecutan en el mismo bloque de datos



- En este caso hay múltiples unidades de procesamiento operando en los mismos datos, pero se aplican varias instrucciones a estos mismos datos. Las arquitecturas de este tipo tienen usos muy específicos, y no se encuentran dentro de las arquitecturas usadas para procesadores de altas prestaciones o de uso general
- No obstante, ha habido implementaciones de multiprocesadores de propósito especial que se ajustan a esta definición
- Las arquitecturas MIMD se construyen desde unidades de procesamiento múltiple que trabajan en múltiples flujos de datos de manera paralela



- La arquitectura se construye de múltiples unidades de procesamiento, que tienen múltiples flujos de instrucciones trabajando con múltiples flujos de datos
- Este es el paradigma de computación más nuevo, donde las arquitecturas permiten que muchos flujos de datos se procesen usando múltiples flujos de computación. Las arquitecturas MIMD facilitan el acceso a múltiples *threads* computacionales y flujos de datos

- Los flujos computacionales pueden estar completamente aislados de otro, o pueden compartir recursos y contextos: en el primer caso, se habla de diferentes procesos, donde cada proceso contiene un contexto de ejecución que es completamente aislado de otros, y en el segundo caso, se habla de *threads* de ejecuciones con múltiples *threads* compartiendo un único contexto (comparten los datos y las instrucciones)
- Es importante notar que algunas arquitecturas actualmente en uso siguen más de un solo paradigma de los descritos
  - Por ejemplo, hay varias arquitecturas de múltiples núcleos que contienen unidades vectoriales, que se usan para realizar cálculos vectoriales de una manera paralela. Esto significa que un solo procesador puede tener partes SIMD y MIMD
  - Estas arquitecturas se pueden encontrar en el campo de la computación de altas prestaciones son esencialmente arquitecturas SIMD y MIMD

### \*Las arquitecturas de sistemas HPC: SIMD y MIMD

- Las arquitecturas SIMD se caracterizan por la habilidad de procesar múltiples flujos de datos usando un solo flujo de instrucciones. Las arquitecturas vectoriales son el tipo de arquitecturas SIMD más conocidas, de modo que se desarrollan a continuación
  - La mayoría de procesadores de alto rendimiento tienen unidades basadas en vectores específicas. Las arquitecturas de este tipo tienen ciertas propiedades que las hacen muy atractivas cuando se realizan cálculos científicos y se procesan grandes volúmenes de datos

```

Parameters:
IntegerVector[1024] A;
IntegerVector[1024] B;
IntegerVector[1024] C;

Code:
for(i = 0; i < 1024; i++)
    C[i] = A[i]+B[i];

Parameters:
IntegerVector[1024] A;
IntegerVector[1024] B;
IntegerVector[1024] C;

Code:
VectorRegister1512 a,b,c;

For(i=0; i<512; i+=16)
{
    load a,A[i]); load(b,B[i]);
    c = vector_sum(a,b);
    store(C[i],c);
}

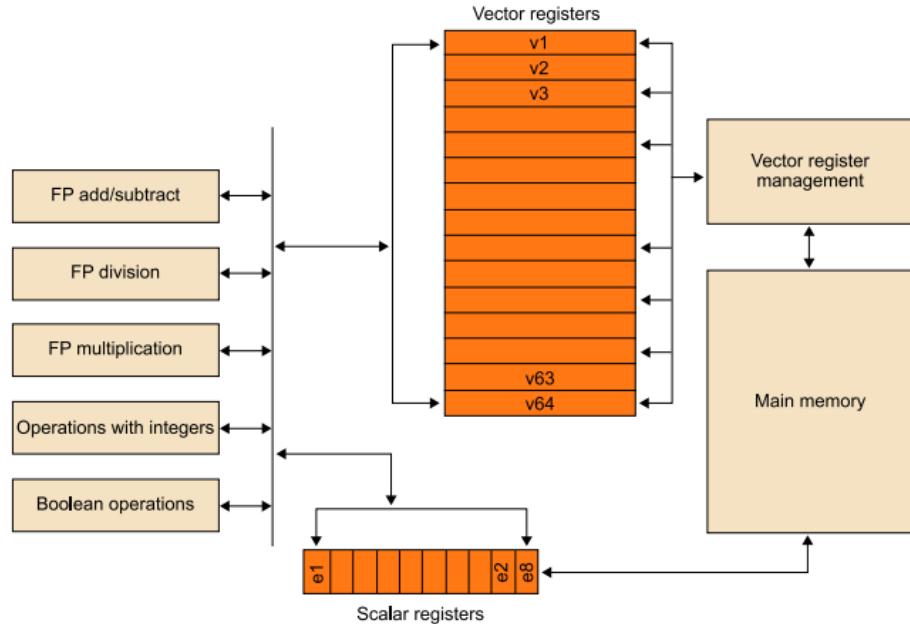
```

- A diferencia de las arquitecturas SISD, las arquitecturas vectoriales operan a nivel de vectores de datos. Esto significa que una sola instrucción se puede usar para sumar dos registros que pueden guardar  $k$  elementos diferentes
- Aunque los códigos anteriores son un ejemplo simplificado, da una idea de la mejora de rendimiento que las aplicaciones pueden tener usando unidades vectoriales. En el primer código, se tendrían que realizar 1024 operaciones, de modo que se tendría que leer el valor entero de la localización que se está procesando cada vez, mientras que el segundo código aprovecha que se tiene una unidad vectorial de 512 *bytes* para trabajar con iteraciones de 16 elementos cada una (elementos enteros de 32 *bytes*), ahorrando iteraciones
- Como se puede ver en el segundo código, los incrementos son de 16 debido a que  $512/32 = 16$ , por lo que en cada iteración se suman 16 elementos a la vez. Esto da una idea de la potencia al procesar estructuras matriciales y vectoriales con respecto a utilizar unidades escalares
- No siempre es posible trabajar con bloques de 512 *bytes*, dado que depende de las estructuras de datos que se usan y de la complejidad del código
  - Esto no es solo un mecanismo de *software*. Es decir, el sistema y las librerías proporcionan un conjunto de interfaces que permiten la operación usando bloques de 512 *bytes*, aunque estas llamadas acaban

siendo traducidas al lenguaje nativo del procesador que es capaz de operar con estos bloques

- Cada procesador vectorial proporciona una arquitectura específica de conjuntos de instrucciones (ISA), de modo que puede operar usando datos de tipo vectorial. Esto significa que el tipo de operaciones vectoriales que se pueden realizar en una arquitectura específica dependerán del ISA proporcionado
- En el ejemplo anterior, si los procesadores vectoriales ISA no proporcionan la suma de dos registros vectoriales, el compilador probablemente traslada esta suma de dos registros en 216 sumas escalares. Por otra parte, si esta suma se proporciona, el compilador traducirá la suma de estos registros en una sola instrucción que calculará la suma de ambos elementos
- Los procesadores vectoriales o unidades pueden mejorar significativamente el rendimiento de aplicaciones que se pueden vectorizar. No obstante, las ventajas de realizar operaciones vectoriales no acaban con la capacidad de realizar  $m$  sumas o restas en paralelo: la capacidad de las operaciones vectoriales de operar sobre un conjunto de datos en un bloque da a los procesadores vectoriales algunas características interesantes
  - Trabajar con bloques de datos permite al compilador o al programador especificar al procesador que no hay dependencias entre diferentes elementos que hacen a los vectores, de modo que el procesador no tiene que realizar verificaciones conforme a los riesgos estructurales o riesgos de datos entre los diferentes elementos de los vectores. Con un procesador escalar normal, el procesador tendría que verificar esto para todas las operaciones de cada elemento, mientras que con uno vectorial solo se hace esto para los registros vectoriales
  - Para realizar cálculos en paralelo de varios elementos de los registros vectoriales, el procesador puede usar unidades funcionales replicadas para cada elemento de los registros de manera paralela
  - En general, los procesadores proporcionan un conjunto extenso de instrucciones vectoriales
  - Esto hace que este tipo de arquitecturas sean atractivas para aplicaciones científicas y de ingeniería, sobre todo cuando se emplean grandes volúmenes de datos o se necesita un gran nivel de paralelismo de datos

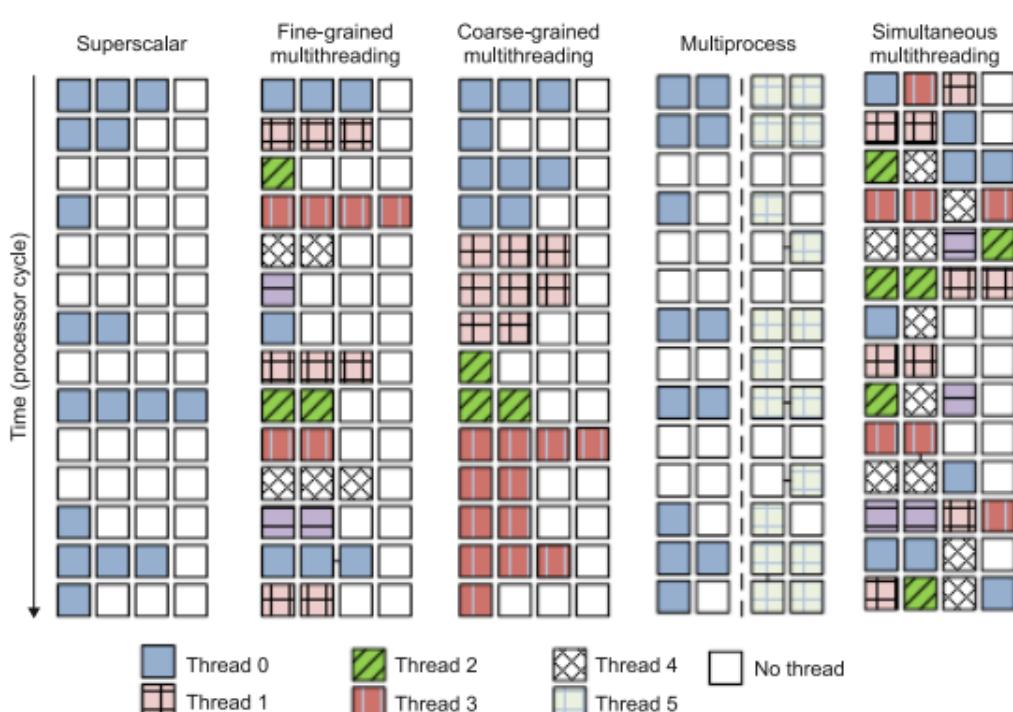
- El siguiente esquema es un ejemplo de un posible procesador vectorial que sirve para discutir los diversos elementos involucrados



- Tal y como se puede ver, el procesador vectorial está hecho de dos tipos de bancos de registros: un bloque que contiene los registros, y otro bloque que está hecho de unidades funcionales. También hay otros dos, pero que son de naturaleza diferente
- En el primer bloque que contiene los registros hay vectores que pueden guardar 512 *bytes* de información, de modo que cada uno almacena elementos de punto flotante o enteros: si se quieren guardar enteros, se pueden guardar hasta 16 elementos (32 *bytes* por elemento), y si son de punto flotante, se pueden guardar hasta 8 (64 *bytes*). Por otra parte, el procesador también tiene un conjunto de registros escalares disponibles, dado que algunas instrucciones vectoriales pueden usar escalares como parte de la operación, o pueden generar resultados escalares
- El segundo bloque consiste de unidades funcionales que están subdivididos en tres grandes bloques: el primero está por las unidades funcionales, dedicadas a realizar cálculos con vectores que contienen elementos de punto flotante o que se van a tratar como tal; el segundo incluye la unidad funcional dedicada a realizar cálculos con vectores que contienen datos enteros; y el tercero es responsable de realizar operaciones booleanas con registros vectoriales
- El tercer y el cuarto bloque incluyen los elementos orientados a gestionar el acceso a la memoria (ya sea caché L1/L2 o la memoria principal) y transfiriendo datos de ahí a los registros vectoriales

- En el caso de los registros vectoriales, que trabajan con bloques de 512 *bytes*, hay una unidad específica disponible que es responsable de gestionar el acceso a la memoria. Los registros escalares funcionan de la misma manera que funcionarían en un procesador escalar normal, y, por tanto, no se necesitan elementos específicos para realizar este papel
- Las arquitecturas de *multiple instruction multiple data streams* (MIMD) nacen de la motivación de incrementar el rendimiento de las aplicaciones a través del paralelismo
  - Durante las primeras décadas del desarrollo de microprocesadores (1970-1980), el objetivo principal era permitir que las aplicaciones explotaran el SISD o el paralelismo a nivel de instrucciones tanto como fuera posible
    - Durante estas décadas se realizó el esfuerzo para conseguir un mejor rendimiento para aplicaciones usando técnicas que permitían que las instrucciones fueran ejecutadas *out-of-order*, que aplicaran técnicas de predicción más precisas, o que tuvieran jerarquías de memoria de capacidad más alta
    - No obstante, el rendimiento de las aplicaciones no escaló proporcionalmente con la cantidad de recursos añadidos. Por ejemplo, si se pasa de una arquitectura que permite iniciar dos instrucciones por cada ciclo a una que permite cuatro por ciclo, esto no necesariamente duplica el rendimiento, sino que normalmente solo lo aumenta un poco
  - Esto motivó la aparición de las arquitecturas MIMD, que permiten múltiples *threads* de ejecución que se ejecutan simultáneamente en un solo procesador (paralelismo a nivel de *threads*). Estas arquitecturas tienen las siguientes propiedades:
    - Múltiples *threads* de ejecución comparten las unidades funcionales del procesador
    - El procesador debe tener estructuras independientes para cada uno de los *threads* que está ejecutando: un registro de renombrado, un contador de programa, etc.
    - Si los *threads* pertenecen a diferentes procesos, el procesador debe proporcionar mecanismos que permitan trabajar con diferentes tablas de páginas (estructuras de datos usadas por la memoria virtual para guardar el mapeado entre direcciones lógicas y físicas)

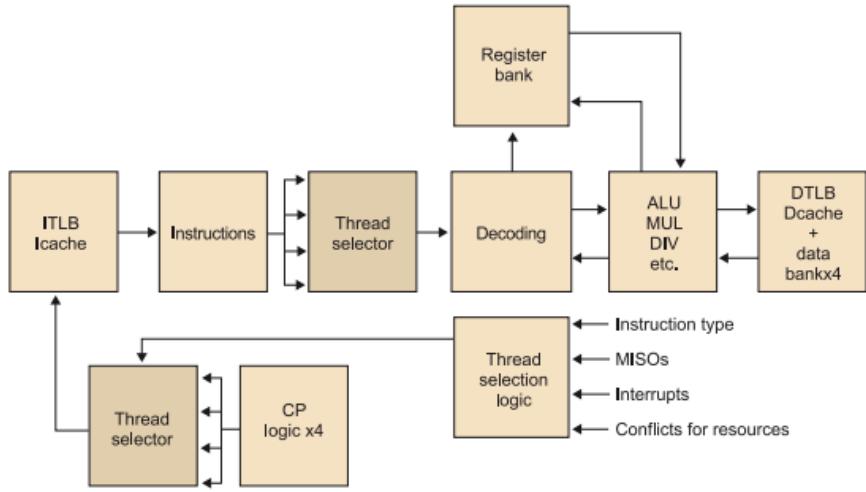
- Las arquitecturas actuales se aprovechan de este paradigma, pero la motivación sigue siendo la misma: la mayoría de aplicaciones actuales tienen un gran nivel de paralelismo y su rendimiento incremento cuanto más paralelismo se añade a nivel de procesador
  - El objetivo es mejorar la productividad de los sistemas computacionales que puedan ejecutar inherentemente aplicaciones paralelas
  - Bajo estas circunstancias, ganar rendimiento usando paralelismo a nivel de *thread* o *thread-level parallelism* (TLP) puede ser mucho más efectivo en términos de coste/rendimiento que usar paralelismo a nivel de instrucciones o *instruction-level parallelism* (ILP). En la mayoría de casos, cuanto más paralelismo hay, mayor es el rendimiento que se puede conseguir
  - En cualquier evento, la mayoría de técnicas que se han usado con sistemas paralelos a nivel de instrucciones también se han usado con sistemas TLP
- Existen varios tipos de arquitecturas TLP que se han propuesto las últimas décadas, algunas de las cuales son las siguientes:



- Las arquitecturas de multiprocesador o *multiprocessor architectures* (MP) son la extensión más simple de un modelo de ILP
  - En este caso, se replica la arquitectura de ILP  $n$  veces

- El modelo más básico para estas arquitecturas es el multiprocesador simétrico. No obstante, su mayor desventaja es que, aunque se tienen varios *threads* de ejecución, cada uno continua teniendo las limitaciones del ILP
- No obstante, hay variaciones en donde cada procesador tiene múltiples *threads* al mismo tiempo
- Las arquitecturas con múltiples *threads* o *multithreaded architectures* (conocidas como *super-threading*) son la siguiente variación del TLP que se basa en expandir el *pipeline* del modelo del procesador para también considerar el concepto del *thread* de ejecución
  - En este caso, el planificador (que escoge las instrucciones que comenzarán durante el ciclo) es capaz de seleccionar cuales de los *threads* de ejecución iniciará la siguiente instrucción durante el siguiente ciclo
- Las arquitecturas con múltiples *threads* simultáneos o *simultaneous multithreading architectures* (SMT) son una variación de las arquitecturas con múltiples *threads* que permiten a la lógica planificadora seleccionar instrucciones de cualquier *thread* durante cada ciclo de reloj
  - Esta característica hace que el uso de recursos sea más alto y más eficiente. Sin embargo, su mayor fallo es la complejidad de la lógica necesaria para realizar esta gestión
  - La habilidad de comenzar múltiples instrucciones en múltiples *threads* es muy costosa. Esto significa que estas arquitecturas típicamente usan un número relativamente bajo de *threads*
- Las arquitecturas con múltiples núcleos o *multicore architectures*, conocidas como *chip-multiprocessor architectures* (CMP) o *system-on-chip* (SOC) son muy similares a las primeras arquitecturas descritas (conceptualmente) pero a una escala mucho más pequeña
  - En el caso de sistemas MP, hay  $n$  procesadores independientes que pueden compartir memoria. Sin embargo, no comparten recursos entre ellos (como el último nivel de memoria de caché)
  - El SOC es una evolución conceptual del MP, pero transferido al nivel del procesador. En un SOC, un solo procesador está hecho de  $m$  núcleos, sobre los cuales los *threads* se pueden ejecutar y estos pueden estar relacionados o compartir recursos

- A partir de ver las diferentes arquitecturas MIMD, ahora se puede desarrollar cada una de manera más detallada. Primero se revisan las arquitecturas de *super-threading* y SMT
  - El TLP mejora el rendimiento porque los recursos se están compartiendo entre múltiples *threads* de ejecución. No obstante, hay dos maneras en las que se puede llevar a cabo esta compartición
    - La primera manera consiste en compartir los recursos en el espacio y en el tiempo: durante un ciclo específico y en un nivel de procesador específico, las instrucciones de múltiples *threads* pueden estar compartiendo los mismos niveles del procesador
    - La segunda manera consiste en compartir los recursos en tiempo: durante un ciclo específico y en un nivel de procesador específico, las únicas instrucciones que se hayan provienen de un solo *thread*. Esto se denomina *superthreading*, y hay dos maneras de compartir recursos en el tiempo entre varios *threads*: a través de *fine-level sharing* o de *coarse-level sharing*
  - En *fine-level sharing*, el procesador cambia *threads* en cada instrucción que está ejecutando. Esto permite que los *threads* se ejecuten de una manera intercalada
    - El cambio de *thread* normalmente se da siguiente una distribución *round robin* (asigna elementos a un grupo de dos o más en orden): una instrucción del *thread n* se ejecuta, y entonces se ejecuta una de  $n + 1$ , otra de  $n + 2$ , etc. En los casos en donde un *thread* esté bloqueado, se esquiva y se sigue con el siguiente
      - Con tal de llevar a cabo los cambios de *threads* el procesador debe poder hacer un cambio de *thread* por cada ciclo
      - La ventaja de esta opción es que ayuda a equilibrar bloques largos y cortos para los *threads* que se están ejecutando, dado que el *thread* cambia en cada ciclo. La mayor desventaja es que reduce el rendimiento de los *threads* individuales, dado que los *threads* que están listos para ejecutarse se retrasan por las instrucciones para los otros *threads*
      - Esto tiene implicaciones en términos de la complejidad del diseño y el consumo de energía, dado que el procesador tiene que ser capaz de hacer estos cambios de *thread*
  - El *pipeline* del *fine-level sharing* se puede esquematizar conceptualmente de la siguiente manera:



- El *pipeline* de este tipo de arquitectura contiene un interruptor que determina qué *thread* se ejecutará durante el siguiente ciclo. Este selector cambia el *thread* en cada ciclo, aunque solo puede escoger del conjunto de *threads* disponibles en cada momento del tiempo
- Cuando hay un evento de latencia larga (un error de la memoria de caché que obliga a una llamada a la memoria, por ejemplo), el *thread* que ha causado el error se borra del *round robin*. Una vez este evento acaba, el *thread* se vuelve a añadir para la rotación de acceso a las unidades funcionales
- El hecho de que el *pipeline* se comparta con múltiples *threads* durante cada ciclo significa que cada *thread* se ejecutará más lento, mientras que el uso del procesador es muy alto. Otro efecto interesante es que los errores de memoria de caché tienen un impacto más pequeño: si uno de los *threads* causa un error, los otros pueden continuar usando los recursos compartidos y seguir teniendo progreso
- Tal y como se ve en el esquema, todos los elementos del *pipeline* se usan por solo un *thread* durante un solo ciclo. No obstante, algunas de las estructuras se replican para el número de *threads* que tiene el núcleo (por ejemplo, la lógica de gestión para el contador del programa o PC)
- En este caso, el núcleo debe ser capaz de determinar en qué parte del *stream* está el *thread*, y también significa que esta estructura se requiere para cada *thread*. Por otro lado, el resto de la lógica, tal como la lógica de decodificación, es única y solo se usa por un *thread* durante el ciclo

- En el *coarse-level sharing*, el cambio de *thread* tiene lugar cuando el *thread* de ejecución tiene un bloqueo de larga duración
  - La ventaja de esta opción es que el rendimiento de un *thread* individual no decrece, porque el cambio de *thread* solo ocurre si hay un evento que bloquee el *thread* (evento de latencia larga)
  - La desventaja es que no puede mejorar el rendimiento cuando los bloques son cortos. Como el núcleo solo genera instrucciones para un *thread* durante un ciclo, cuando el *thread* se bloquea durante un ciclo específico (por una dependencia de las instrucciones, por ejemplo), todos los niveles siguientes del procesador se bloquean o se hielan, y solo se usan otra vez si el *thread* vuelve a poder procesar instrucciones
  - Otra desventaja significativa es que los nuevos *threads* que comienzan en el procesador deben pasar a través de todos los niveles antes de que el procesador comience a tomar instrucciones de ese *thread*. El modelo previamente discutido tenía menos efecto en la productividad debido a que los *threads* cambiaban en cada ciclo
  - Por ejemplo, con un procesador con 12 ciclos de niveles y 4 *threads* de ejecución, en el mejor caso, un nuevo *thread* debe esperar 12 ciclos antes de recibir la primera instrucción. Pero durante esos 12 ciclos, los otros 3 *threads* han podido recibir hasta 12 instrucciones
- Las arquitecturas con SMT son variaciones de las arquitecturas con múltiples *threads* que permiten que las instrucciones sean escogidas de cualquiera de los *threads* que el procesador está ejecutando durante el ciclo
  - Esto significa que múltiples *threads* son simultáneamente compartiendo una variedad de recursos del procesador. Esta compartición es tanto horizontal (por ejemplo, fases sucesivas dentro del *pipeline*) y vertical (por ejemplo, recursos dentro del mismo nivel del procesador)
  - El resultado de estas técnicas es un alto uso de los recursos del procesador y una eficiencia mucho más grande. Se tiene que recordar que este no era el caso con las arquitecturas paralelas anteriores, donde dos *threads* no pueden compartir los mismos recursos de uno de los niveles del procesador en el mismo momento en el tiempo

- Sin embargo, la ganancia en eficiencia provoca que el procesador deba contener una lógica muy extensiva y compleja. De manera predefinida, los *threads* de diferentes procesos no necesariamente se ven entre ellos y, por razones relacionadas a la seguridad y funcionalidad, la ejecución de una instrucción *A* para un *thread X* no puede modificar el comportamiento funcional del *thread B*
- En un sistema operativo, cada proceso tiene su propio contexto. Este contexto es independiente de los contextos de otros procesos, a no ser que se usen técnicas explícitas para la compartición de recursos
- Las arquitecturas que son completamente SMT y son capaces de trabajar con muchos *threads* son extremadamente costosas en términos de complejidad, y las estructuras necesarias para soportar este tipo de compartición crecen proporcionalmente con el número de *threads* disponible. Aunque está claro que el incremento de paralelismo provoca mejor rendimiento, también causa necesidad de más espacio y más consumo de energía, de modo que se quiere balancear todo esto
- Ahora que se han explicado las arquitecturas SMT, se puede desarrollar el diseño de este tipo de arquitecturas y el diseño de algunas de las arquitecturas más relevantes en la literatura
  - Igual que se ha dicho antes, las arquitecturas SMT permiten la ejecución de múltiples *threads* para compartir una variedad de unidades funcionales. Para permitir este tipo de compartición, el estado de cada uno de los *threads* se tiene que guardar independientemente
    - Por ejemplo, los registros usados por los *threads* se deben duplicar, igual que el contador del programa y también debe haber una tabla de páginas separada para cada *thread*
    - Si estas estructuras no se duplican, la ejecución funcional de cada uno de los *threads* interferirá con la de otro *thread*. Esto puede crear serios problemas de seguridad: por ejemplo, al compartir una tabla de páginas, eso quiere decir que un *thread* podría compartir el mapeado de direcciones virtuales a físicas (un *thread* puede acceder a la memoria de otro *thread* sin ningún tipo de restricción)
    - No obstante, hay otros recursos que no necesitan ser replicados, como el acceso a las unidades funcionales para acceder a la memoria (porque los mecanismos de la memoria virtual soportan la programación múltiple)

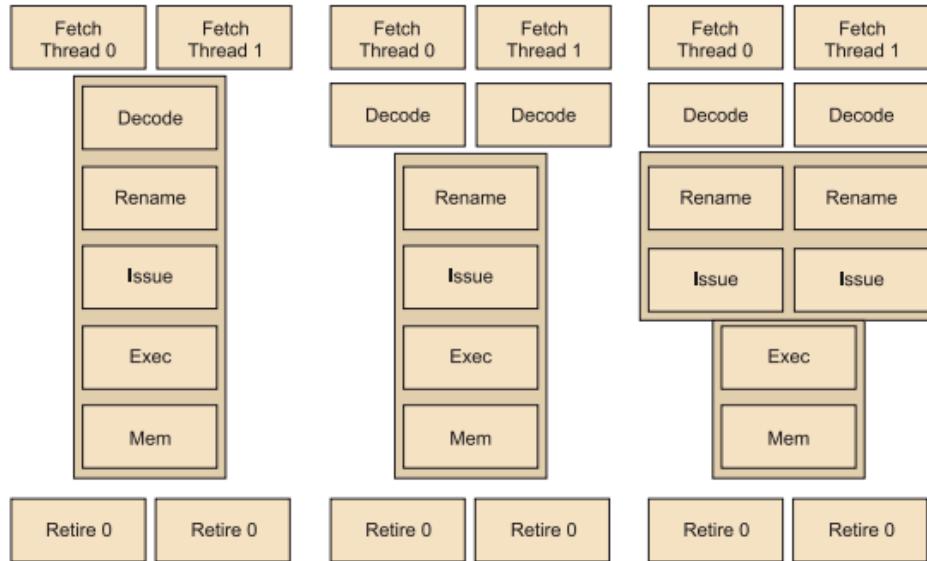
- El número de instrucciones que un procesador SMT puede generar por ciclo está limitado a los desequilibrios en los recursos necesitados para ejecutar los *threads*, además de la disponibilidad de estos recursos
  - Sin embargo, hay otros factores que limitan esto, tales como el número de *threads* activos, las posibles limitaciones al medir las colas disponibles, la habilidad de generar suficientes instrucciones para los *threads* disponibles, o limitaciones en el tipo de instrucciones que se pueden generar desde cualquier *thread* y para todos los *threads*
- Las técnicas de SMT asumen que el procesador va a proporcionar un conjunto de mecanismos que permitirán usar paralelismo a nivel de *thread*. En particular, estas arquitecturas tienen un gran conjunto de registros virtuales que se pueden usar para guardar registros de cada uno de los *threads* independientemente
  - Gracias al cambiar el nombre de los registros, las instrucciones de diferentes *threads* pueden mezclarse durante las diferentes fases del procesador, sin causar confusión sobre las fuentes y destinos de los varios *threads* disponibles. Es importante notar que este es el tipo de técnica que se podría usar en procesadores SISD *out-of-order* (ejecuta instrucciones tan pronto como los recursos y datos estén disponibles, sin necesariamente seguir el orden del programa), pero en ese caso, el procesador tendría que tener una sola tabla de renombramiento
  - Al final, el proceso de renombramiento para los registros es exactamente el mismo que el proceso realizado por un procesador *out-of-order*. Debido a esto, un SMT puede considerarse como una extensión de este tipo de procesador (añadiendo la lógica necesaria para soportar los contextos de varios *threads*)
  - Es posible construir una arquitectura que sea *out-of-order* y SMT al mismo tiempo
- Para el siguiente ejemplo, si no se cambiara el nombre de los registros r2, r3 y r4 para cada uno de los *threads*, la ejecución funcional de las distintas instrucciones provocaría errores

```

cycle(n)thread 1-->"LOAD #43,r3"
cycle(n+1)thread 2-->"ADD r2,r3,r4"
cycle(n+2)thread 1-->"ADD r4,r3,r2"
cycle(n+3)thread 1-->"LOAD (r2),r4"

```

- En los casos 1 y 4, los valores leídos por los dos *threads* serían incorrectos. En el primer caso, el *thread 2* estaría usando el valor del registro r3, que fue modificado por el *thread 1* durante el ciclo anterior
  - De modo similar, lo mismo ocurriría con el ciclo 3, donde el *thread 1* añadiría un valor r3 modificado por otro *thread*
- El proceso de finalizar una instrucción, conocido como *committing*, no es tan simple como con un procesador que no sea SMT (que solo considera un *thread*): en este caso, se quiere que la finalización de la instrucción sea independiente para cada *thread* (para que se pueda avanzar de manera independiente)
  - Esto se puede realizar usando estructuras que permitan a este proceso realizarse para cada uno de los *threads* (por ejemplo, con un *buffer* de reordenación por *thread*)
- En términos generales, los SMTs siguen la misma arquitectura que se usa para procesadores superescalares. Esto incluye tanto los diseños generales para las diferentes fases (fase de búsqueda de instrucciones, decodificación de registros, lectura de registros, etc.) y las técnicas o algoritmos usados
  - No obstante, con tal de soportar los múltiples contextos que el procesador debe gestionar, muchas de estas estructuras necesitan ser replicadas. Algunas de estas son las mínimas requeridas para prevenir interferencias entre *threads* y asegurar que las aplicaciones se ejecutan correctamente (tal como tener contadores de programas separados o tablas de páginas separadas)
  - Además, variaciones a las arquitecturas pueden no ser estrictamente necesarias, pero pueden proporcionar un mejor rendimiento dependiendo del contexto
- Algunas de las opciones que se pueden tener en cuenta en relación a la arquitectura general de un procesador SMT se representan en el siguiente esquema. Este esquema muestra las diferentes posibilidades en las que múltiples *threads* pueden compartir (o no) las diferentes fases del procesador (asumiendo fases típicas para un procesador superescalar *out-of-order*)

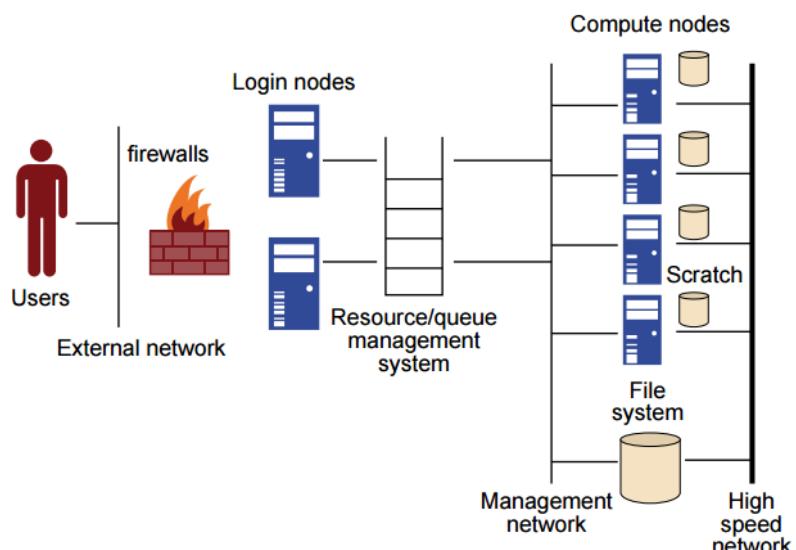


- La primera opción asume que solo la búsqueda de instrucciones se divide en *threads*, mientras que el resto de fases se comparten
- Cuanto más se mueve uno a la derecha, se puede ver como las fases se separan más en *threads*, lo cual significa que el procesador ejecuta la fase dividida en *threads* y que cada *thread* tiene estructuras separadas que se usan para hacerlo. No obstante, es razonable pensar que todos los *threads* comparten una lógica, dado que el mecanismo es común en todos ellos
- En cualquier caso, la ejecución y las fases de acceso a la memoria se comparten por todos los *threads*. En un contexto académico, se podría pensar que estas se replican para cada *thread*, pero esto es costoso en la práctica en términos de espacio y consumo de energía
- Además, es lógico pensar que el uso de estos recursos sería mucho mayor en casos donde todos los *threads* comparten estas fases. Esto significa que, cuando algunos *threads* se bloquean, los otros usarán los recursos y viceversa, o que cuando un *thread* está accediendo a la memoria, los otros usan las unidades aritméticas (que tienen que estar compartidas por eficiencia energética)
- Las fases de decodificación y renombramiento identifican las fuentes y los destinos de las operaciones, y también calculan las dependencias entre las instrucciones que se están ejecutando. En este caso, el hecho de que las instrucciones para los diferentes *threads* sean independientes significa que la lógica correspondiente se puede separar también

- No obstante, en muchos casos tener estructuras de renombramiento compartidas permite que el procesador sea más eficiente
- Por ejemplo, si se supone que hay un total de 50 registros de renombramiento para los 2 *threads* y que las estructuras están separadas, cada *thread* puede acceder a un máximo de 25 registros. En el caso de que uno de los *threads* pueda usar solo 10 pero el otro necesite 40, el sistema estará infrautilizado y el rendimiento del segundo *thread* disminuirá sustancialmente

## Las arquitecturas de sistemas HPC: redes de interconexión y sistemas de archivos

- Tal como se ha visto, la mayoría de sistemas de altas prestaciones o supercomputadoras son sistemas de tipo clúster, aunque antes se usaban más sistemas de memoria y multiprocesadores compartidos
  - La necesidad de un mayor nivel de paralelismo y la aparición de las redes de interconexión de alto rendimiento han significado que los sistemas de altas prestaciones actuales son una colección de computadoras independientes conectadas entre ellas, trabajando conjuntamente para resolver un problema
  - Por lo tanto, los computadores y la red de comunicación son elementos esenciales en un sistema de altas prestaciones. No obstante, hay otros elementos clave, los cuales se muestran en el siguiente esquema:



- Los sistemas de computadoras consisten principalmente en dos tipos de computadoras o nodos: los nodos de registro o *login nodes*, y los nodos de computación o *compute nodes*

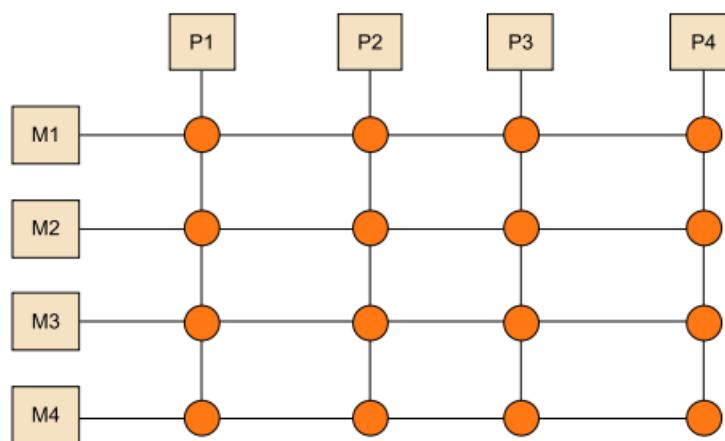
- Los nodos de registro o *login nodes* son nodos accesibles por los usuarios, normalmente a través de conexiones seguras del tipo *ssh*, y estos son comunes para supercomputadoras. Este tipo de nodo facilita las funciones básicas, tales como crear y editar archivos, copiar código fuente, mandar y gestionar *jobs* en la cola, etc.
- En algunos casos, los nodos de registro son del mismo tipo que los nodos de computación, pero pueden ser de diferentes tipos y tener incluso una arquitectura diferente. En este último caso, el usuario tiene que especificar las opciones de compilación de modo que el programa pueda ejecutar correctamente el *job* acorde a los nodos de computación
- Como parte de la infraestructura de seguridad del sistema, los *firewalls* normalmente se ponen de modo que permitan aislar el sistema de las redes externas
- Los nodos de computación o *compute nodes* son nodos que ejecutan aplicaciones en paralelo usando una red de alta velocidad. Aunque puedan ser heterogéneos, normalmente suelen ser homogéneos o tienen particiones diferentes que son homogéneas (por ejemplo, puede haber una colección de nodos con un CPU, otros que incluyen GPUs y otros para visualizar y guardar características específicas como más capacidad de memoria)
- Uno de los elementos esenciales de sistemas de altas prestaciones que se diferencian de otros sistemas, tales centros de datos o *data centres*, es la red de interconexión
  - Normalmente, uno encuentra redes de alta velocidad, lo que conlleva una banda ancha amplia, pero con muchas latencias reducidas, y otra red para gestión que no se involucra en la ejecución de aplicaciones paralelas
  - Los sistemas de altas prestaciones suelen usar un sistema de archivos paralelos compartida con nodos del sistema. Este tipo de sistemas permite un mayor rendimiento con un gran número de nodos o el uso de volúmenes grandes de datos
  - De manera complementaria, los nodos pueden tener un espacio temporal de almacenamiento, por ejemplo, en un formato de disco o SSD para mejorar la localidad de los datos cuando no se tienen que compartir. De este modo se reduce el uso de la red de la interconexión y el sistema de archivos paralelos, pero teniendo

en cuenta de que solo se puede hacer al final de la ejecución, cuando los datos se mueven al soporte de almacenamiento permanente

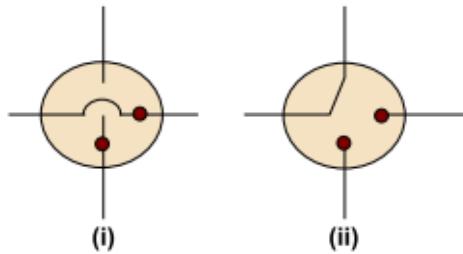
- Finalmente, uno puede también encontrar copias de respaldo o grandes dispositivos de almacenamiento, proporcionando una gran capacidad, pero con mucha más latencia
- Con tal de usar los recursos de manera más eficiente y compartirlos entre múltiples usuarios, los sistemas de altas prestaciones usan sistemas de cola como una interfaz entre el usuario y los nodos computacionales
  - Además de mantener los *jobs* de los usuarios del sistema, estos pueden también gestionarse usando políticas de planificación
- En el nivel de *software*, se tiene que notar que, actualmente, los sistemas de altas prestaciones usan sistemas basados en Unix, especialmente Linux. Aparte de eso, se puede incluir *middleware* con los siguientes objetivos:
  - Proporcionar transparencia al usuario de modo que no se tenga que preocupar sobre detalles de bajo nivel
  - Mejorar la escalabilidad y la disponibilidad del sistema para utilizar aplicaciones de los usuarios
  - Usar el concepto de SSI (o *single system image*) para permitir que el usuario vea el clúster de una manera unificada como si fuera un solo ordenador o recurso
- Desde el punto de vista de la ejecución y el entorno de desarrollo, el *software* usual, tal como los compiladores y los paquetes matemáticos, normalmente se gestionan usando módulos
  - Esto permite una modificación del entorno de manera dinámica (usar una cierta versión de un compilador, por ejemplo)
  - Algunas acciones útiles cuando se usan módulos son la consulta de módulos disponibles (*module avail*), la carga de módulos (*module load <app\_name>*) o el *unloading* de estos (*module unload <app\_name>*)
- Las redes de interconexión tienen un rol decisivo en el rendimiento de sistemas de memoria distribuida y de memoria compartida: aún si los procesadores y memorias tuvieran una capacidad de rendimiento ilimitado, la red de interconexión deterioraría el rendimiento de programas paralelos de manera

significativa. Aunque algunas redes de interconexión comparten algunos aspectos, existen varias diferencias dependiendo del sistema en el que se use

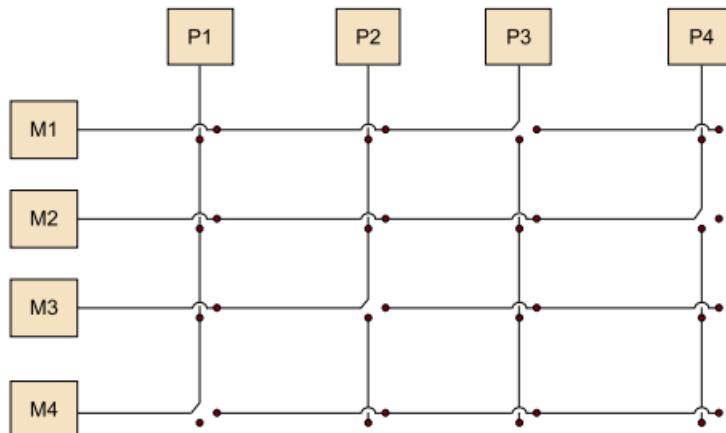
- A día de hoy, las dos redes de interconexiones más usadas en sistemas de memoria compartida son los buses y los *crossbars*
  - Un bus no es más que una colección de cables de comunicación en paralelo con un *hardware* que controla el acceso al bus. La característica más importante es que los cables de comunicación están compartidos por todos los dispositivos conectados
  - Los buses tienen la ventaja de tener un tamaño pequeño y ser muy flexibles, de modo que varios dispositivos se pueden conectar al bus con un coste adicional muy pequeño. No obstante, como los cables de comunicación se comparten, cuando el número de dispositivos conectados incrementa, la posibilidad de contención para el uso del bus también incrementa, y eso causa un riesgo de reducir el rendimiento
  - Por lo tanto, cuando se conectan un gran número de procesadores a un bus, se puede suponer que se tendrá que esperar frecuentemente para acceder a la memoria principal. En consecuencia, cuanto más aumenta el tamaño de la compartición de memoria, los buses se reemplazan por redes de interconexión comutadas o *switched interconnection networks*
- Las redes de interconexión comutadas o *switched interconnection networks* comutan (o *switches*) para controlar la ruta de los datos entre los dispositivos conectados. El siguiente gráfico muestra una *crossbar*, donde las líneas representan los vínculos de comunicación bidireccional, los cuadrados son núcleos o módulos de memoria y los círculos representan comutadores



- Los conmutadores individuales pueden tener una o dos configuraciones, tal como se muestra en el siguiente esquema con las dos configuraciones posibles:

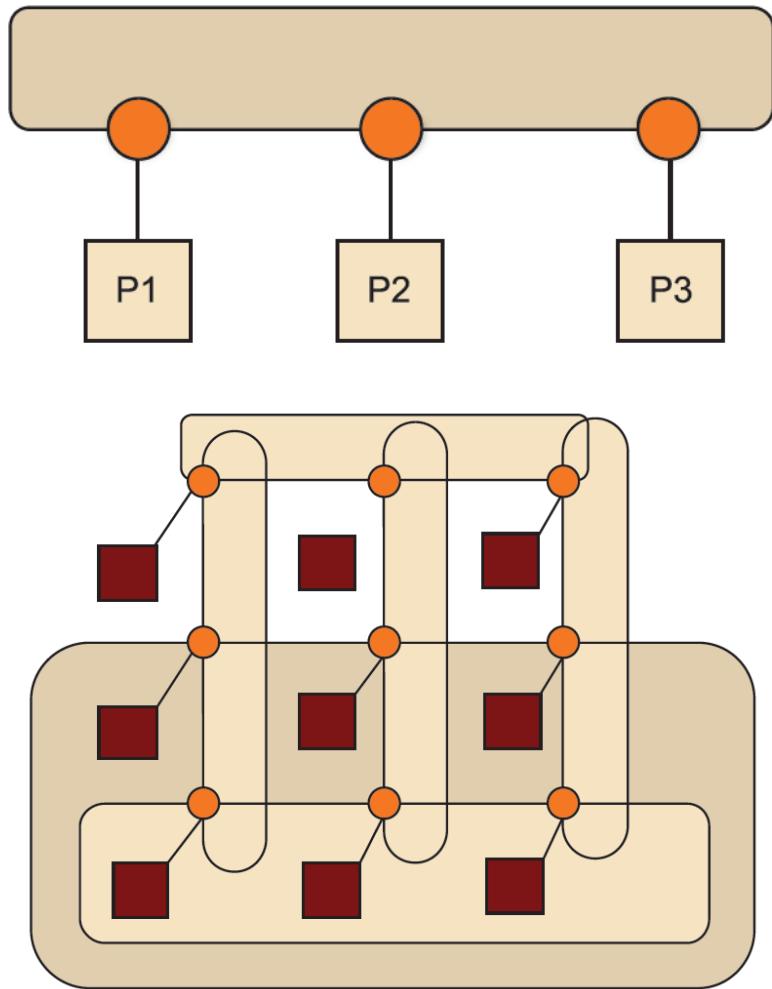


- Con estos conmutadores y al menos tantos módulos de memoria como procesadores, solo existirán conflictos entre núcleos que intenten acceder a la memoria si los dos núcleos intentan acceder al mismo módulo simultáneamente
- Para entender mejor el funcionamiento, el siguiente esquema muestra la configuración de los conmutadores si  $P1$  escribe a  $M4$ ,  $P2$  lee de  $M3$ ,  $P3$  lee de  $M1$  y  $P4$  escribe a  $M2$



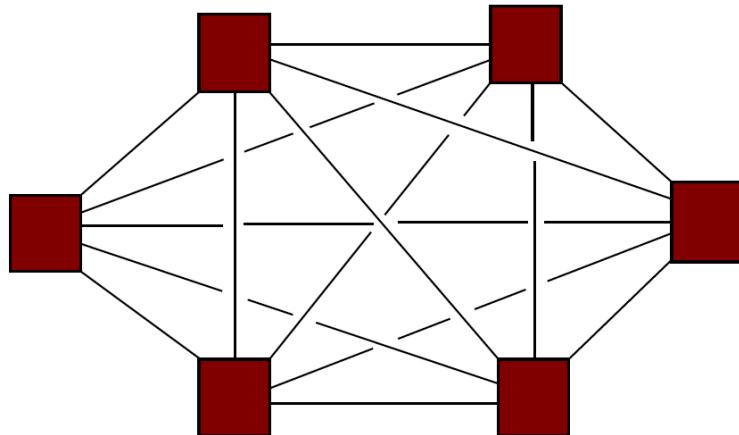
- Los *crossbars* permiten comunicación simultánea entre diferentes dispositivos y, por lo tanto, son más rápidos que los buses. De otro lado, el coste de los conmutadores y los vínculos es relativamente alto, por lo que sistemas de buses son generalmente más baratos para un sistema pequeño
  - Las redes de interconexión para sistemas de memoria distribuida normalmente se dividen en dos grupos: las redes de interconexiones directas y las de interconexiones indirectas
    - En una red de interconexión directa, cada uno de los conmutadores está conectado a un par memoria-procesador y los conmutadores están conectados entre ellos (los vínculos son bidireccionales). Ejemplos de este tipo de redes son los anillos y

las rejas toroidales, estas redes se esquematizan igual que antes, con círculos representando conmutadores, las líneas vínculos bidireccionales, y los cuadrados procesadores:

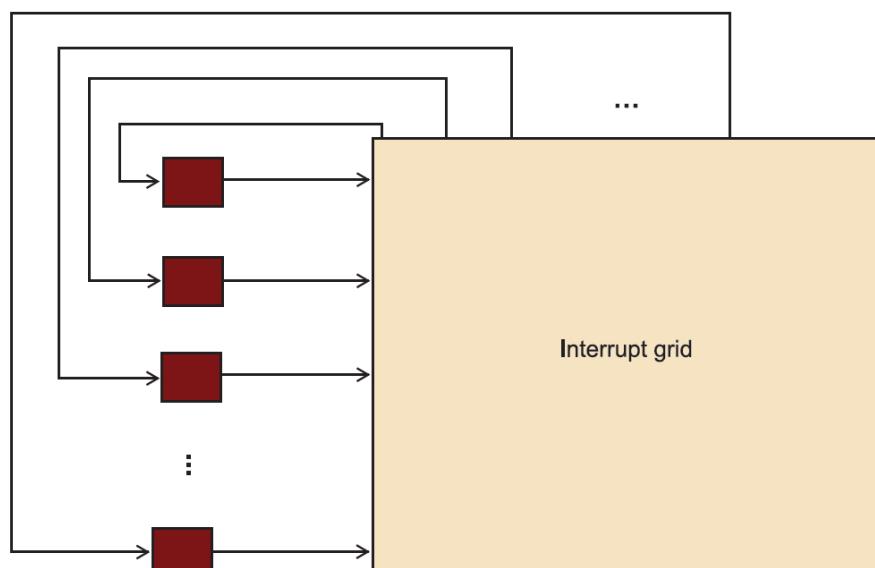


- Un anillo es mejor que un simple bus, dado que permite múltiples conexiones simultáneas. No obstante, existen situaciones en las que algunos procesadores tendrán que esperar a que otros acaben sus comunicaciones
- Una reja toroidal es más costosa que un anillo porque los conmutadores son más complejos (deben tener 5 vínculos, o posible combinación de puertos, para entenderse mejor, en vez de 3 como en un anillo) y, si hay  $p$  procesadores, el número total de vínculos (cables o segmentos de líneas negras en los esquemas) sería de  $3p$  en una reja toroidal, mientras que en un anillo solo habrían  $2p$
- La banda ancha de un vínculo (cable) es la velocidad o tasa a la que puede transmitir datos, y normalmente se expresa en megabits por segundo. La

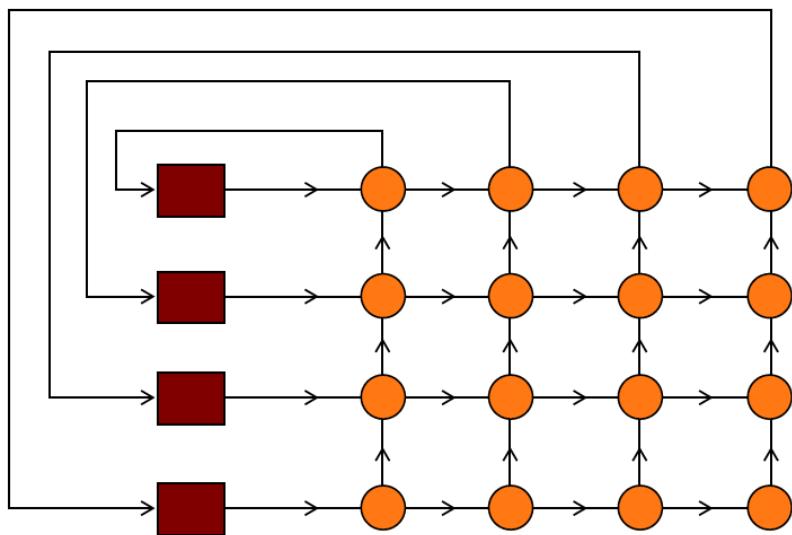
red interconectada ideal está completamente conectada y todos los conmutadores están conectados entre ellos



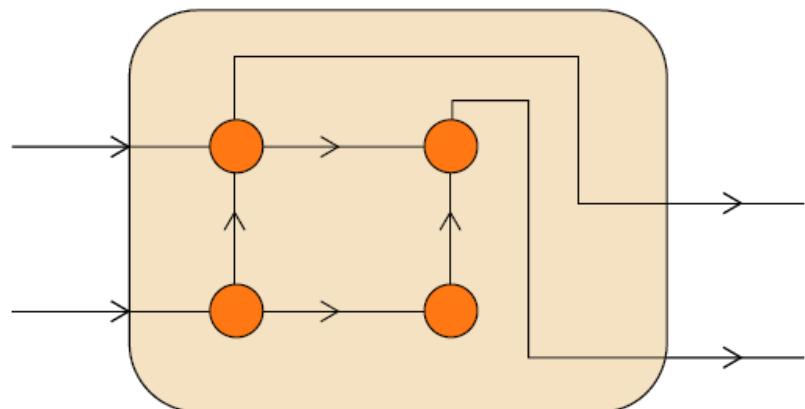
- El problema de este tipo de red es que los sistemas no se pueden construir con más de unos pocos nodos, dado que requieren un total de  $p + p/2$  vínculos y cada conmutador tiene que ser capaz de conectarse a  $p$  vínculos
- Por lo tanto, esta red es más una red interconectada teórica en el mejor de los casos, pero no una red práctica. Esta se usa como referencia para evaluar y analizar otro tipo de redes interconectadas
- Las interconexiones indirectas son la alternativa a las interconexiones directas, en donde los conmutadores no se pueden conectar directamente a un procesador. Muchas veces, estos se ven como vínculos unidireccionales y una colección de procesadores, donde cada uno tiene vínculo (o cable) de entrada y otro de salida, como en el siguiente esquema:

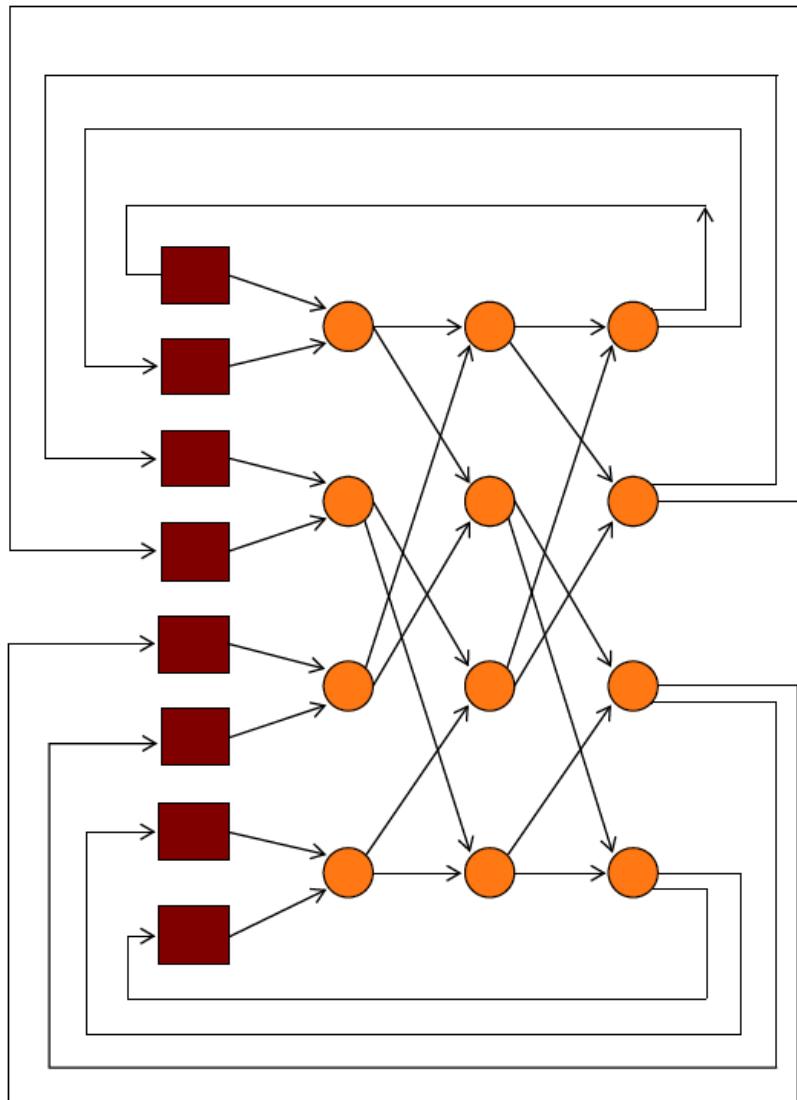


- Dos ejemplos importantes y relativamente simples de este tipo de redes son las redes *crossbar* y las redes omega
- La red de interconexión indirecta *crossbar* tiene vínculos unidireccionales, a diferencia de su contraparte directamente conectada. Mientras que dos procesadores no intenten comunicarse a la vez con el mismo procesador, todos los procesadores se pueden conectar a otros procesadores simultáneamente



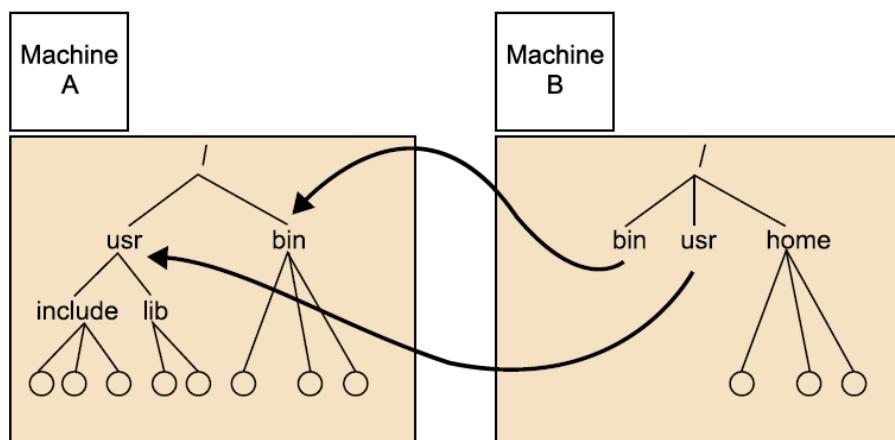
- En una red omega, los conmutadores son *crossbars*  $2 \times 2$ : a diferencia de la red indirecta *crossbar*, hay comunicaciones que no pueden darse simultáneamente. No obstante, es menos costosa que una red *crossbar*, dado que esta red usa una cantidad de  $(1/2)p \log_2(p)$  de los conmutadores de los *crossbars*  $2 \times 2$  y, por tanto, usan un total de  $2p \log_2(p)$  conmutadores (una red indirecta *crossbar* usa  $2 \times 2$ )





- Un aspecto importante para analizar de los sistemas de altas prestaciones son las características y las cuestiones básicas relacionadas a sistemas de archivos. Además, también es adecuado discutir algunos tipos de sistemas de archivos
  - En general, un sistema de archivos distribuidos permite un acceso transparente y eficiente a archivos en sistemas remotos para los procesos
    - Además, proporcionan una interfaz de programación que esconde los detalles de la localización y cómo el almacenamiento se realiza realmente
    - Algunas ventajas de estos sistemas son el acceso de archivos propios desde otros dispositivos, el acceso de diferentes usuarios a los mismos archivos, la facilidad de gestión (solo hay uno o un grupo de servidores) y la confianza mejora, debido a que se puede añadir tecnología RAID (o *redundant array of independent disks*)

- No obstante, estos sistemas también presentan algunos retos como la escalabilidad, la tolerancia de los errores, la consistencia, la seguridad y la eficiencia
- El sistema de red de archivos o *network file system* (NFS) es un sistema de compartición de archivos entre máquinas en la red de modo que parece como si se estuviera trabajando con el disco duro local. Un sistema Linux puede trabajar como un servidor NFS o cliente (o ambos), lo cual da la habilidad de exportar sistemas de archivos, así como montar sistemas de archivos que otras máquinas pueden exportar



In the example, machine A exports directories /usr and /bin and mounts them on machine B.

- Con tal de usar NFS, Linux utiliza una combinación de soporte a nivel de *kernel* y *daemon* en ejecución continua para permitir el intercambio de archivos NFS. Por lo tanto, el soporte NFS debe estar activo en el núcleo del sistema operativo
- Por otro lado, NFS utiliza *remote producer calls* o RPC para canalizar las demandas entre clientes y servidores. El uso de RPC indica que un servicio de mapeo de puertos o *portmap service* (*daemon* encargado de crear conexiones RPC con el servidor y permitir o denegar el acceso) tiene que estar activo y disponible
- Los procesos RPC que son necesarios son los siguientes, aunque intervienen algunos cuantos procesos adicionales:
  - El *rpc.mountd*, que es un proceso que recibe la demanda de montar un directorio desde el cliente NFS y verificar que coincide con el sistema de archivos exportado actualmente
  - El *rpc.nfsd*, que es un proceso implementando componentes del espacio del usuario del servicio NFS. Funciona con el *kernel* de Linux para satisfacer demandas de cliente dinámicas NFS, tales como proporcionar procesos de servidor adicionales de uso para clientes NFS

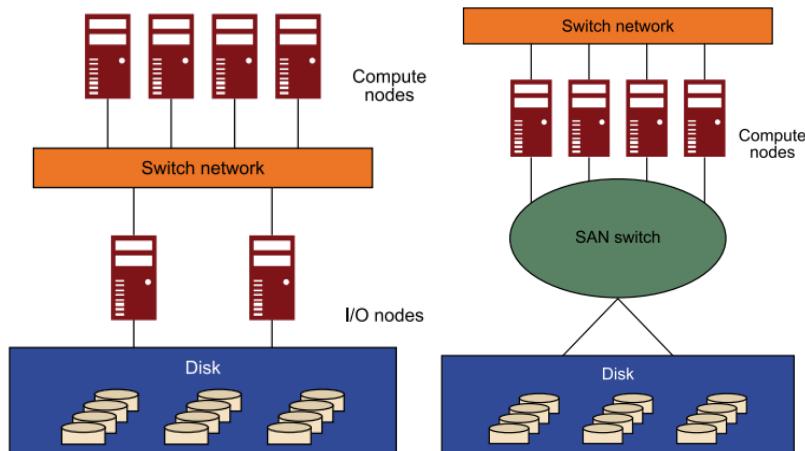
- La configuración del servidor NFS mediante el archivo etc/exports permite especificar cuestiones muy importantes relacionadas con el intercambio de archivos. Entre ellas, la lista de máquinas autorizadas en la compartición se puede encontrar, junto a otras opciones tales como las siguientes:
  - La opción *ro* (de *read-only*), de modo que se muestran máquinas *read-only* que no pueden modificar el sistema de archivos
  - La opción *rw* (de *read-write*), de modo que se muestran máquinas *read-write* que si pueden modificar el sistema de archivos
  - La opción *async*, que permite al servidor escribir datos en el disco cuando crea que es apropiado
  - La opción *sync*, que hace que toda escritura en el disco se deba realizar antes de devolver el control al cliente
  - La opción *wdelay*, que hace que el servidor NFS retrase la escritura del disco cuando sospecha que otra demanda del cliente es inminente
  - La opción *no\_wdelay*, que hace que se desactive la opción previa (pero solo funciona si se usa la opción *sync*)
  - La opción *root\_squash*, que da ciertos privilegios a usuarios conectados remotamente (*root*), tales como *local root*
  - La opción *root\_squash*, que desactiva la opción anterior
  - La opción *all\_squash*, que hace que todos los usuarios se conviertan otra vez
- Los sistemas de archivos paralelos son intentan resolver la necesidad de entrar/salir de ciertas aplicaciones que gestionan volúmenes masivos de datos y, por tanto, requieren una gran cantidad de operaciones de entrada/salida en dispositivos de almacenamiento
  - Aunque se ha visto un gran incremento en la capacidad de memoria de los dispositivos, lo mismo no ha ocurrido con su rendimiento, en cuanto a latencia y banda ancha se refiere. Esto ha causado que un desequilibrio sustancial entre la capacidad de procesamiento y la entrada/salida, lo cual ha tenido repercusiones negativas en aplicaciones que son intensivas en operaciones de entrada/salida

- Los sistemas de archivos paralelos se basan en el mismo principio que la computación para mejorar el rendimiento, lo que significa que proporcionan entrada/salida paralela. Esto da como resultado la distribución de datos entre múltiples dispositivos de almacenamiento y permite el acceso a los datos en paralelo
- Hay diferentes tipos de conexiones de dispositivos de almacenamiento: el *direct-attached storage* (DAS), el *network-attached storage* (NAS) y las *storage area networks* (SAN)
  - El DAS es la solución clásica, en donde se asocia un disco duro a cada nodo
  - El NAS es una solución en donde un nodo gestiona una colección de discos duros
  - El SAN es una solución que utiliza una red dedicada al almacenamiento. Este almacenamiento no está vinculado a ningún nodo (es como un “disco” en la red), hay conectividad total entre nodos y dispositivos (cualquiera puede acceder a otro) y hay conectividad directa entre dispositivos (lo cual acelera la replicación, las copias de seguridad, etc.)
- En cuanto a la organización de los sistemas de archivos en paralelo, estos sistemas usan técnicas de *stripping*, que consisten en guardar datos de archivos distribuidos entre discos de sistemas de modo similar al de RAID-0, pero para *software* y varios nodos. Estos sistemas se comparten en una distribución funcional de dos niveles: el nivel inferior y el superior
  - El nivel inferior es el servicio de almacenamiento distribuido (por ejemplo, SAN) y el nivel superior es el sistema de archivos en cada nodo computacional. Para acceder a los discos como si fueran locales, cada nodo computacional debe gestionar la meta-información de los datos
  - El siguiente esquema ilustra la organización conceptual en capas de sistemas de archivos en paralelo:
- Algunos ejemplos de estos sistemas son el sistema GPFS, el Lustre, el PVFS y el sistema de archivos de Google. Los dos primeros se desarrollan abajo con tal de ejemplificar las características de estos sistemas
  - El sistema GPFS se desarrolló por IBM como un sistema de archivos paralelo, permitiendo a los usuarios acceder a los datos que están dispersos entre múltiples nodos, además de interactuar a través de interfaces UNIX estándar

- GPFS mejora el rendimiento del sistema, garantiza la consistencia de datos, tiene una alta capacidad de recuperación y disponibilidad de los datos, proporciona el sistema con alta flexibilidad y simplifica la gestión
- Las mejoras en el rendimiento se deben a factores como los siguientes:
  - Permite que múltiples procesos o aplicaciones accedan a los datos de manera simultánea desde todos los nodos usando un sistema de llamadas estándar
  - Incrementa la banda ancha a cada uno de los nodos que intervienen en el sistema GPFS
  - Balancea la carga uniformemente entre todos los nodos en el sistema GPFS. Un disco puede no tener más actividad que otro
  - Soporta datos de grandes dimensiones y permite leer y escribir simultáneamente (concurrentemente) desde cualquier nodo en el sistema GPFS
- GPFS utiliza un sistema de administración complejo que garantiza la consistencia de datos al mismo tiempo que permite rutas múltiples e independientes para archivos con el mismo nombre desde cualquier lugar en el clúster
  - Cuando la carga de ciertos nodos se vuelve más alta, GPFS puede encontrar una ruta alternativa para el sistema de archivos de datos
  - Para recuperar y facilitar la disponibilidad de datos, GPFS crea entradas de *log* separadas para cada uno de los nodos del sistema, permitiendo que el *hardware* se organice dentro de un número de grupos de fallida. Además, la función de replicación de GPFS permite especificar el número de copias de archivos que se quieren mantener
  - GPFS también permite añadir recursos, sean discos o nodos, dinámicamente, sin necesidad de parar ni reiniciar el sistema
- En un sistema GPFS, cada uno de los nodos se organiza según componentes como los comandos administrativos, las extensiones de *kernel*, un *multithread daemon* o una capa portátil de código *open source*
  - La extensión de *kernel* proporciona una interfaz entre el sistema operativo y el sistema GPFS, que facilita la manipulación de datos

en un entorno GPFS. Por ejemplo, para copiar un archivo, el comando usual es `cp file.ext`

- Los *daemons* GPFS se ejecutan en todos los nodos de *input/output* y en un *buffer* administrador por GPFS. Todos estos nodos están protegidos por un administrador de *tokens*, que asegura que los múltiples nodos cumplan con la atomicidad y proporcionen al sistema de archivos su consistencia



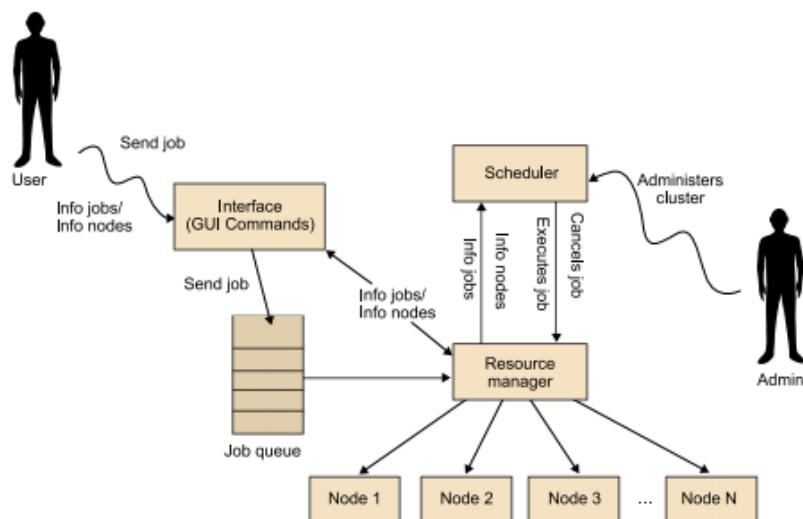
- Los *daemons* son procesos *multithread* con algunos de los *threads* dedicados a propósitos específicos. Esto asegura de que el servicio nunca se interrumpa por otro *thread* que esté ocupado en una rutina de la red
- Las funciones específicas del *daemon* son la asignación de espacio de disco a nuevos archivos, la administración del directorio, la asignación del bloque para la protección de datos y la integridad de los metadatos, iniciar el servidor del disco con un *thread* del *daemon*, y administrar la seguridad junto con el administrador del sistema de archivos

## Las arquitecturas de sistemas HPC: gestión de tareas y recursos

- Como se discutió en los primeros capítulos, los sistemas de alto rendimiento actuales suelen ser clústeres de ordenadores interconectados con redes de alta velocidad. En estos sistemas normalmente hay varios usuarios que quieren acceder a los recursos para ejecutar programas en paralelo, y para organizar el acceso y la gestión de usuarios de manera eficiente, los sistemas suelen tener sistemas de gestión de recursos basados en colas
  - En sistemas de altas prestaciones no tiene sentido proporcionar a los usuarios acceso a recursos de manera interactiva, porque puede crear

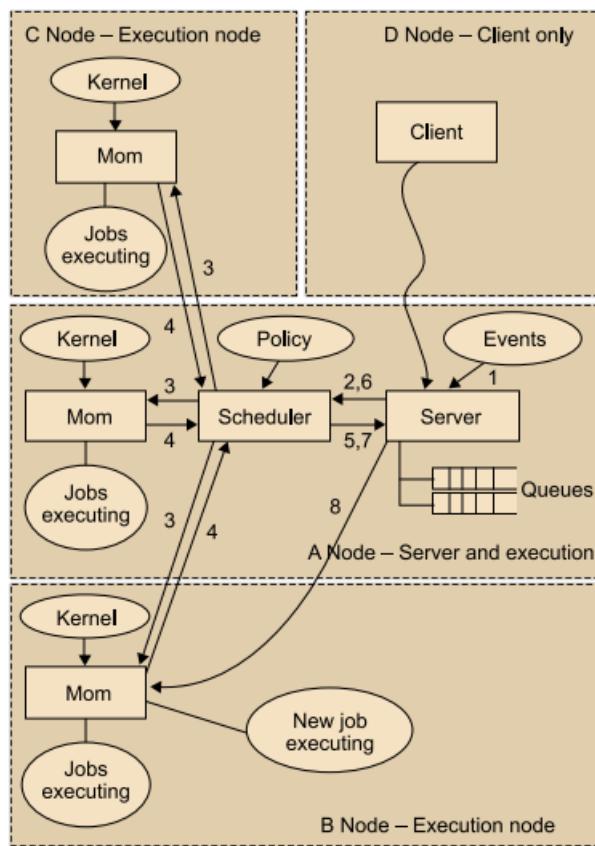
caos: ocurrirían conflictos en el uso de recursos y habría poca eficiencia en su uso y compartición

- Además, los usuarios no deberían preocuparse por problemas relacionados con la gestión de la computadora, tales como el balanceo de la carga de datos u otros
- Por lo tanto, los sistemas de altas prestaciones se basan en aplicaciones que se pueden guardar en una cola y ejecutarse después en segundo plano. Estos sistemas tradicionalmente se conocen como *batch systems*
- Los componentes esenciales para la gestión computacional en un clúster aparecen en el siguiente esquema:



- En sistemas de altas prestaciones uno normalmente se encuentra con un puñado de nodos que son responsables de la computación (computadoras), pero a los que no se pueden acceder directamente por razones de seguridad y de gestión, y hay una colección de nodos (que son iguales o diferentes del resto) que se usan como nodos de acceso o *login nodes*
- Estos nodos de acceso normalmente permiten el desarrollo de aplicaciones, teniendo sistemas completos con herramientas de compilación, *debugging*, y otros, que permiten a los usuarios enviar sus tareas o *jobs* al sistema de cola para ejecutarse
- El organizador de *jobs* y los gestores de recursos son sistemas de administración, de planificación y de ejecución de *jobs* que se envían al clúster. Por lo tanto, ahora es necesario explicar los sistemas PBS

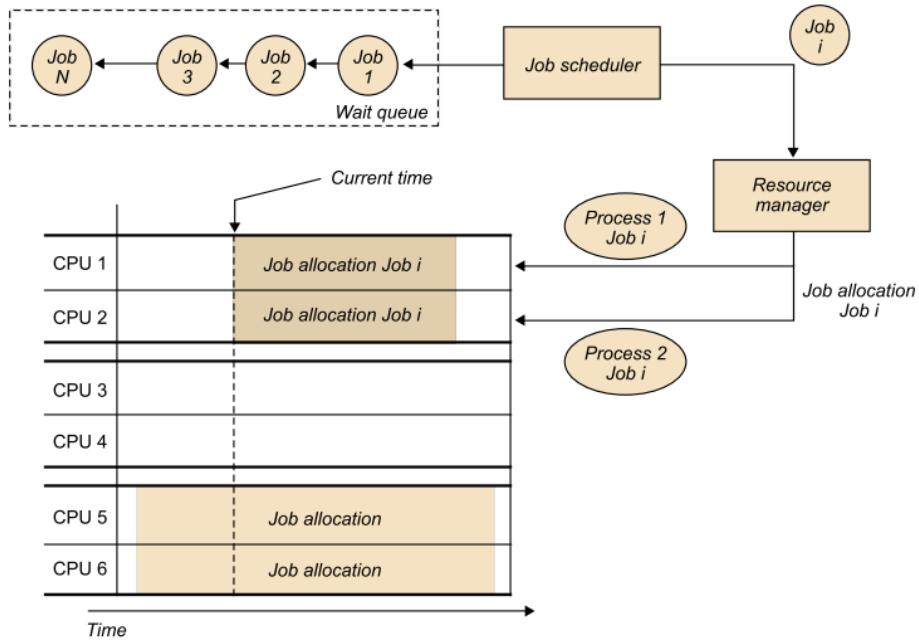
- Un sistema de *batch* portátil o *portable batch system* (PBS) es un sistema de colas diseñado originalmente por la NASA y que se ha vuelto uno de los sistemas de cola más utilizados en computación de altas prestaciones
  - Los PBS proporcionan una serie de herramientas para gestionar *batch jobs* usando una unidad de programación de tareas. Además, esto permite la planificación de *jobs* en diferentes computadoras y define e implementa el uso de políticas para estos recursos
- El esquema del sistema de cola es el siguiente, en el cual se pueden distinguir los componentes que lo componen y las interacciones entre ellos, en donde se pueden distinguir tres componentes clave:



- El primer componente clave es el servidor o *server*, el cual es un *daemon* (proceso en segundo plano que gestiona peticiones para servicios y que no está activo cuando no se requiere) responsable de recibir *jobs* para ser ejecutados y esperar comandos del usuario
- El segundo es el ejecutor de *jobs* o *mom*, el cual es un *daemon* que se ejecuta en todos y cada uno de los nodos y que envía y recibe los *jobs* a ejecutarse

- El tercero es el organizador o *scheduler*, el cual es un organizador que decide que *jobs* ejecutará dependiendo de la política establecida
- Estos componentes interactúan usando los pasos descritos a continuación de manera cíclica:
  - Un evento le dice al servidor que tiene que comenzar el ciclo de planificación, y el servidor envía esta petición de planificación al organizador
  - El organizador pide información sobre los recursos del *mom* y este devuelve la información requerida al organizador
  - Entonces, el organizador pide información sobre el *job* al servidor y este le devuelve esta información sobre el estatus del *job*.
  - El organizador toma después la decisión y ejecuta el *job* o no dependiendo de la política establecida
  - Finalmente, el organizador envía la petición de ejecución al servidor y este último le envía el *job* al *mom* para que se ejecute
- Los usuarios interactúan con el sistema de cola usando una serie de instrucciones. Para ejecutar el programa, el usuario tiene que definir una tarea usando el *script*, donde se debe indicar el programa a ejecutar, sus argumentos, y algunos de los siguientes elementos:
  - La especificación si es un *batch* o un *job* interactivo. El hecho de que sea interactivo no significa que se ejecute inmediatamente, sino que cuando se ejecuta, el usuario puede interactuar con el problema, en contraste a los *batch jobs* que son ejecutados en segundo plano sin interacción directa del usuario
  - La definición de una lista de recursos necesarios, de la prioridad de *jobs* para la ejecución, del tiempo de ejecución (que constituye información importante para planear *jobs* efectivamente) y las dependencias
  - La especificación de si enviar un e-mail al usuario cuando la ejecución comienza, acaba o se aborta, y de si hacer *checkpointing* (que se vuelve muy importante para grandes ejecuciones, en donde hay una necesidad de monitorizar durante la ejecución)
  - La sincronización de un *job* con otros *Jobs*

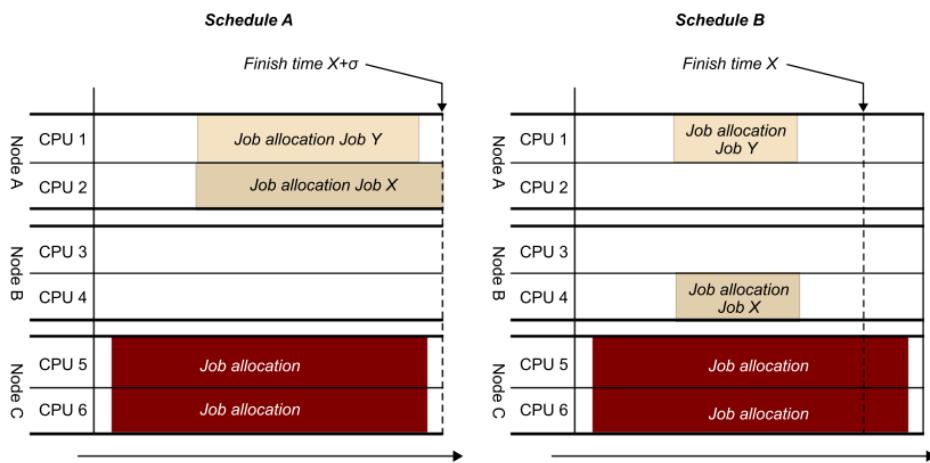
- Además de garantizar que el *job* se pueda ejecutar, la lista de recursos se puede especificar en el archivo de definición del *job* permite un mejor uso del sistema, dado que el organizador puede tomar decisiones más cuidadosamente. Los recursos más comunes son los siguientes:
    - El *cput*, que es el tiempo máximo de la CPU usado por todos los procesos de trabajo
    - El *pcput*, que es el tiempo máximo de la CPU usado por cada uno de los procesos de trabajo
    - El *mem*, que es la cantidad máxima de la memoria física usada por un *job*
    - El *pmem*, que es la cantidad máxima de memoria física usada por cada uno de los procesos de los *Jobs*
    - El *walltime*, que es el tiempo de ejecución (en reloj, no en CPU)
    - El tamaño máximo de cualquier archivo que pueda crear el *job*
    - El *host*, que es el nombre de los nodos computacionales en los que se ejecuta el *job*
    - Los nodos o *nodes*, los cuales son el número y/o tipos de nodos que se tienen que reservar para el uso exclusivo de un *job*
- El PBS proporciona una interfaz para el intérprete y la interfaz gráfica. No obstante, el enfoque más común en computación de altas prestaciones es el uso de la interfaz a través del intérprete de comandos para enviar, monitorizar, modificar y eliminar *Jobs*
- El problema de planear tareas o *jobs* en un sistema de altas prestaciones se puede definir como, dada una lista de *jobs* que están esperando en cola, sus características (el número de procesadores que se necesitan, los archivos ejecutables, los archivos de entrada, etc.) y el estatus del sistema, decidiendo qué *jobs* deberían ejecutarse y en cuáles procesadores
  - Este problema se puede dividir en dos pasos en donde cada uno sigue una política definida: la política de organización y la política de selección de recursos



- Durante el desarrollo teórico, se usa el término *job* para referirse a la ejecución de una aplicación usando variables concretas
- En el primer paso de la organización, el organizador tiene que decidir qué *jobs* van a comenzar de los que están esperando en las colas (*job 1, job 2, ..., job N*), teniendo en cuenta los recursos disponibles (CPU 1, CPU 2, ..., CPU N)
- El algoritmo que el organizador utiliza para hacer la selección se llama política de organización de *jobs*, una vez que el *job* más apropiado para ejecutarse se ha seleccionado, el organizador pregunta al gestor de recursos asignar el *job* seleccionado (*job i*) al procesador más adecuado
- Entonces, el gestor de recursos selecciona los procesadores más adecuados y asigna procesos de *jobs* a cada uno de ellos. El gestor de recursos implementa una política específicamente para la selección de recursos
  - Muchas técnicas y políticas de planificación y de selección de recursos se han propuesto durante los años para optimizar recursos y reducir los tiempos de respuesta
    - La mayoría de políticas de organización de *jobs* se basan en el concepto de *backfilling*. Estas políticas, proporcionan un buen balance entre la eficiencia del sistema (número de procesadores usados) y el tiempo de respuesta
    - No obstante, el mayor problema de los algoritmos de este tipo es que el usuario tiene que proporcionar una estimación del tiempo

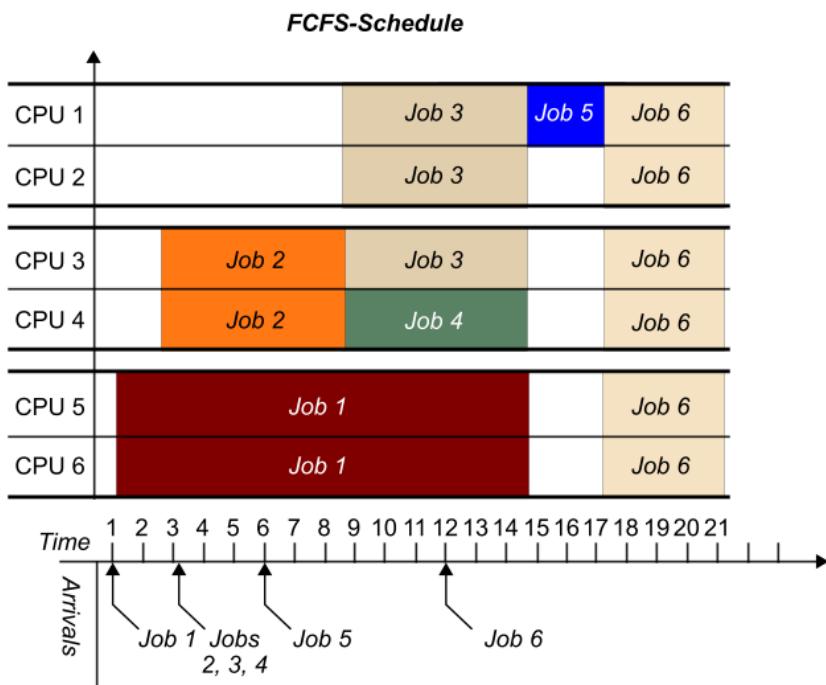
de ejecución de las tareas o *jobs* enviados, y estas estimaciones no suelen ser precisas. Además, en muchas situaciones el usuario no tiene suficiente conocimiento o información para las variables

- Normalmente, el sistema de recursos asigna los *jobs* de dos maneras diferentes: se aplica una colección de filtros predefinidos a las características de los *jobs* para encontrar procesadores adecuados, o se asignan procesos a nodos enteros cuando el usuario especifica que los *jobs* necesitan ejecutarse en nodos no compartidos
  - La primera manera no tiene en cuenta el sistema y, por tanto, una actividad intensiva en memoria puede asignarse a un nodo cuya banda ancha (máxima cantidad de datos que se pueden transferir entre CPU y la memoria del sistema) esté casi saturada, causando un deterioro significativo de todos los *jobs* en el nodo
  - La segunda manera normalmente involucra un uso sustancialmente reducido de los recursos, debido a que los *jobs* no usan los procesadores y los recursos del nodo en el que se han asignado. Por lo tanto, en algunas arquitecturas, cuando los *jobs* comparten recursos (como memoria) pueden causar una sobrecarga, teniendo un impacto negativo en el rendimiento del sistema (colateral)
- Este último problema se puede ilustrar a través del siguiente esquema, en donde diferentes organizaciones de *jobs* se muestran en los casos donde el *job X* y el *job Y* se envían al sistema



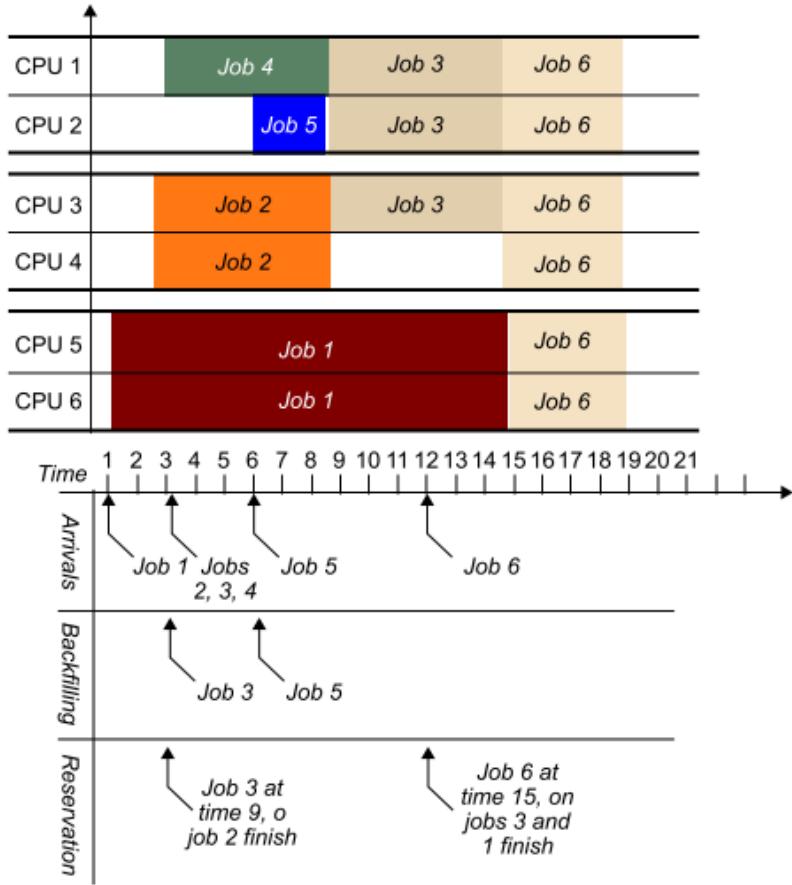
- Con las políticas actuales de planificación (no se tiene en cuenta el sistema), el caso más probable de organización sería el de A, de modo que los *jobs* se asignan consecutivamente en cada el mismo nodo. No obstante, si ambos *jobs* son intensivos en memoria, entonces habrá una reducción del rendimiento en la ejecución por la contención de los recursos

- En cambio, si el organizador tiene en cuenta el sistema y los requerimientos de los *jobs*, entonces el caso más probable sería el B, en donde ningún *job* se penaliza por contención de recursos en términos de rendimiento de la ejecución
- El *backfilling* es la optimización de las políticas de planificación FCFS (*first-come, first-served*), las cuales son una de las políticas más sencillas de analizar
  - En FCFS, el organizador pone en cola todos los *jobs* se envían en orden de llegada y cada uno de esos *jobs* se asocia con un número de procesadores requeridos



- Cuando se completa la ejecución de un *job* es completa, si existen suficientes recursos para comenzar el siguiente *job* en la cola, el organizador lo toma y comienza la ejecución. De otro modo, el organizador el organizador debe esperar hasta que haya recursos disponibles y ninguna otra tarea se puede ejecutar
- El mayor problema con este tipo de política es que el sistema sufre de fragmentación y que el uso de los recursos podría ser muy bajo
- La planificación *backfilling* permite ejecutar *jobs* que están en cola después de otros *jobs* que ya estaban en la cola, sin retrasar el tiempo de ejecución estimado de los *jobs* que están a la espera

FCFS with EASY - backfilling optimization - schedule



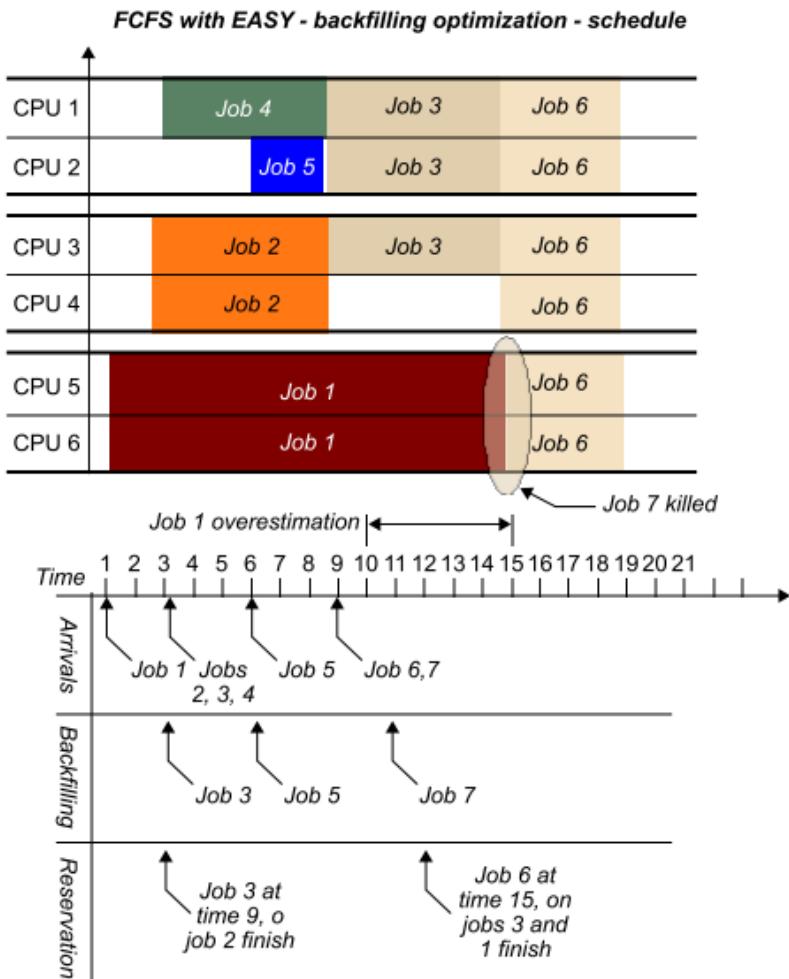
- El esquema muestra una posible política FCFS con EASY-backfilling (una de las diferentes variantes de *backfilling*), de modo que se aplica este último en algunos *jobs* como el 4 y el 5 porque el *job* 3 y el 6 no pueden comenzar al haber procesadores insuficientes
- En un momento  $t$  solo hay dos posibilidades: ejecutar en algunos espacios disponibles un *job* que acaba de llegar o ejecutar un *job* que esté en espera. Para saber qué *jobs* de la cola de reserva asignar, se tiene que ver aquella combinación que minimice los momentos de ejecución (que minimice el tiempo de ejecución) de los *jobs*

CPU	TIME	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
CPU 1	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J10	J10	J10	J10	J10	J10	
CPU 2	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J10	J10	J10	J10	J10	J10	
CPU 3	J1	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J10	J10	J10	J10	J10	J10
CPU 4	J1	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J10	J10	J10	J10	J10	J10
CPU 5	J1	J1	J1	J1	J1	J1	J1	J1	J1	J9	J9	J9	J9	J6	J6	J6	J11	J11	J11	J11	J11	J16	J16	J16	J16	J16	J16
CPU 6	J1	J1	J1	J1	J1	J1	J1	J1	J9	J9	J9	J9	J6	J6	J6	J11	J11	J11	J11	J11	J16	J16	J16	J16	J16	J16	
CPU 7	J1	J1	J1	J1	J1	J1	J1	J1	J1	J9	J9	J9	J9	J6	J6	J6	J12	J12	J12	J12	J12	J16	J16	J16	J16	J16	J16
CPU 8	J1	J1	J1	J1	J1	J1	J1	J1	J1	J9	J9	J9	J9	J6	J6	J6	J12	J12	J12	J12	J12	J16	J16	J16	J16	J16	J16
CPU 9	J2	J2	J2	J3	J3	J3	J3	J3	J3	J3	J3	J3	J3	J6	J6	J6	J14	J14	J14	J14	J14	J16	J16	J16	J16	J16	J16
CPU 10	J2	J2							J5	J5	J7	J7	J7	J7	J13	J13	J13	J13	J15	J15	J15	J15	J15	J15	J15	J10	J10
Util	80%	100%	100%	90%	90%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	80%	80%	80%

- Debido a la optimización, el sistema puede ser mejorado de una manera significativa. Muchos procesadores que se mantendrán

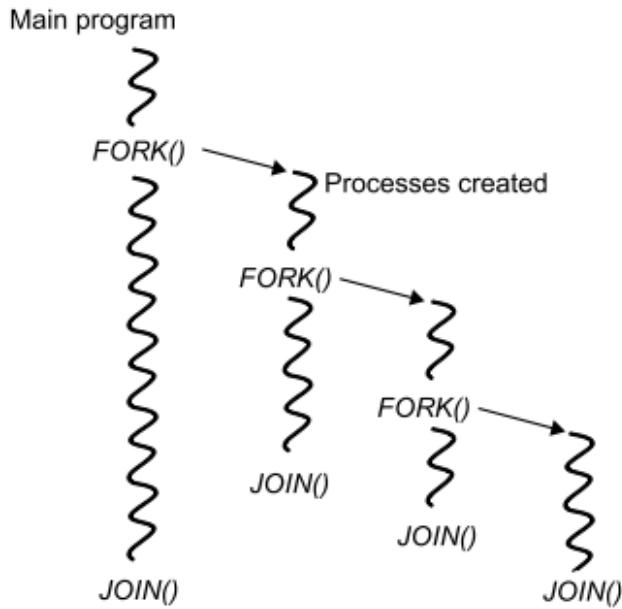
inactivos usando solo una política de FCFS pueden ejecutarse si se usa *backfilling*

- El mayor problema de este tipo de algoritmo es que el usuario debe proporcionar un tiempo de ejecución estimado de los *jobs* cuando se envía al sistema de cola. En situaciones en donde este tiempo se queda corto, el organizador “mata” el *job* cuando detecta que se ha excedido el tiempo
  - Otra situación que es bastante común es el escenario en que el tiempo de ejecución estimado es sustancialmente más grande que el tiempo de ejecución real, lo cual también causa problemas
  - Esta situación se describe en el siguiente esquema. En este, el *job* 1 se ha sobreestimado, por lo que cuando el organizador detecta esta situación, usa el tiempo de ejecución estimado por el usuario y aplica *backfilling* al *job* 7 pero lo acaba matando por el exceso de tiempo (si no se mata, se retrasaría el momento de ejecución del *job* 6)



## La programación en paralelo: modelos de memoria compartida

- Anteriormente se han visto las diferentes maneras de poder programar dependiendo del funcionamiento de la memoria de un sistema. En este caso, para sistemas de memoria compartida, una solución efectiva y muy usada suele ser programar usando procesos y *streams*
  - Los sistemas de multiprocesadores de memoria compartida o sistemas de múltiples núcleos son aquellos en que cualquiera de los procesadores o núcleos pueden acceder al espacio de memoria. Por lo tanto, se encuentra un solo espacio de direcciones de memoria
    - Cada localización de memoria tiene una sola dirección dentro de un rango de direcciones posibles
  - Las alternativas principales disponibles para programar en sistemas de memoria compartida son las siguientes:
    - Usar el procesamiento convencional usado por el sistema operativo
    - Usar *streams*, usando, por ejemplo, Pthreads
    - Usar un nuevo lenguaje de programación o una librería con un lenguaje de programación
    - Modificar la sintaxis de un lenguaje de programación secuencial para crear un lenguaje de programación en paralelo
    - Usar un lenguaje secuencial de programación y suplementarlo con directivas de compilación para especificar el paralelismo, como con OpenMP
  - Uno de los posibles mecanismos para implementar paralelismo de memoria compartida es el uso de procesos convencionales mientras que se aplica un modelo de *fork-join*



- Este es modelo de un solo programa con múltiples datos (SPMD), donde, en general, el código correspondiente al proceso maestro es diferente del código ejecutado por los procesos creados, normalmente conocidos como procesos esclavos (fase *fork*). Además, el proceso maestro tiene que esperar a que los procesos esclavos finalicen (fase *join*)
- La estructura de un código implementado con un modelo de *fork-join* es la siguiente:

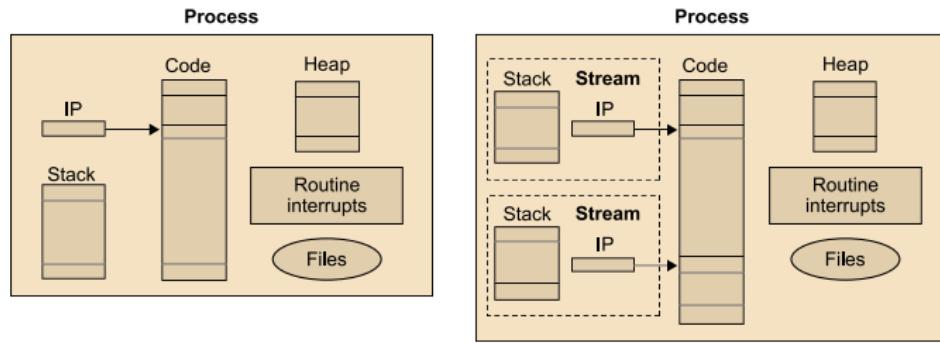
```

pid = fork();
if (pid==0) {
    // code to execute a slave process
}
else{
    // code to execute the master process
}
if (pid==0) exit;
else wait(0);
...

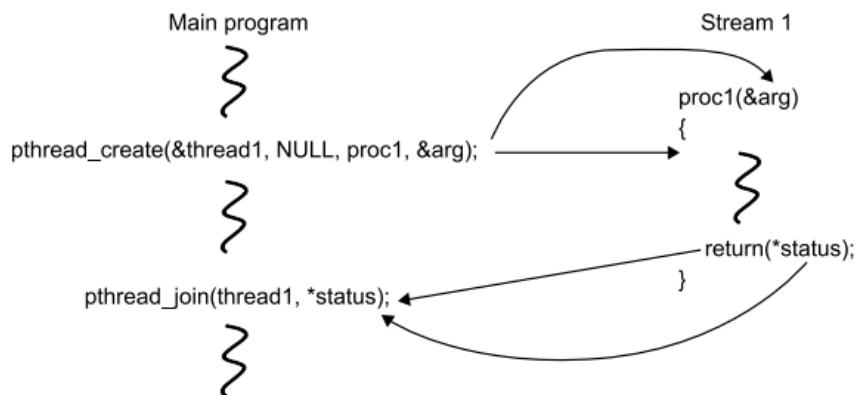
```

- El problema principal que hace este tipo de solución (la implementación directa) poco efectiva es el tiempo de ejecución excesivo (*overheads*) asociado con crear y gestionar estos procesos. En cambio, el modelo introducido anteriormente se aplica en la mayoría de los casos de programación de memoria compartida

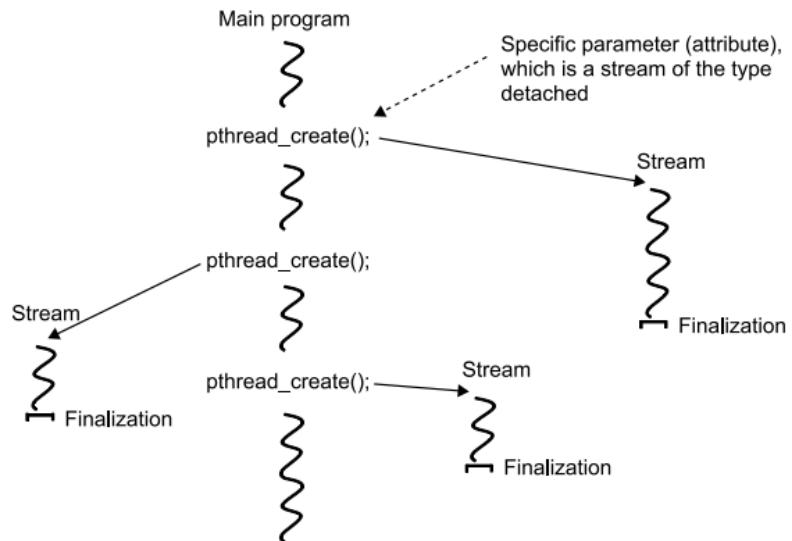
- Una alternativa obvia a usar estos procesos tradicionales es el uso de *streams*. Aunque los procesos son muy intensivos en código (representan códigos independientes con diferentes variables y estructuras de memoria), los *streams* comparten un espacio de memoria global entre las diferentes rutinas



- Es posible utilizar el modelo *fork-join* usando *streams*, tal como se ha visto con Pthreads



- La librería Pthread también permite que se usen *streams* que no requieren que el programa principal espere a la finalización de los procesos esclavos creados (usando el paso *join* directamente a través de su función correspondiente). Los *streams* de este tipo se conocen como *streams sustraídos*, y cuando se finalizan, se destruyen y sus recursos se liberan



- Cuando se usan *streams*, se tiene que recordar que el espacio de memoria es compartido, y puede haber problemas en relación a la consistencia y al rendimiento
  - Una rutina es segura si la ejecución de múltiples copias de una rutina en diferentes *streams* produce los resultados correctos. Esto hace que cuando se acceden a datos compartidos, siempre se tenga que verificar que las rutinas son seguras
  - Habiendo dos *streams* que incrementan el valor de la variable compartida  $x$ , primero se tiene que leer la variable, y después se tiene que calcular  $x + 1$ , escribiéndose finalmente en  $x$ . Como los dos *streams* leen el valor original de  $x$  al mismo tiempo, el valor final de  $x$  solo reflejará un único incremento

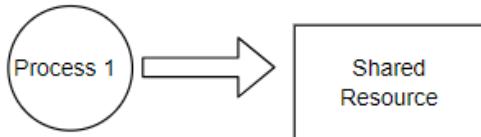
Instruction	Stream 1	Stream 2
$x = x+1;$	read $x$	read $x$
	calculate $x+1$	calculate $x+1$
	write to $x$	write to $x$

↓

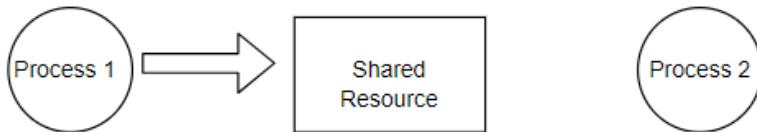
Time

- Un mecanismo usado para asegurarse de que solo un *stream* accede a un recurso específico al mismo tiempo es definir las secciones de código que se pueden ejecutar solo una vez al mismo tiempo. Las secciones de código de este tipo se denominan secciones críticas, y el mecanismo asociado con el uso de estas se conoce como exclusión mutua

1. Process 1 is using the shared resource.



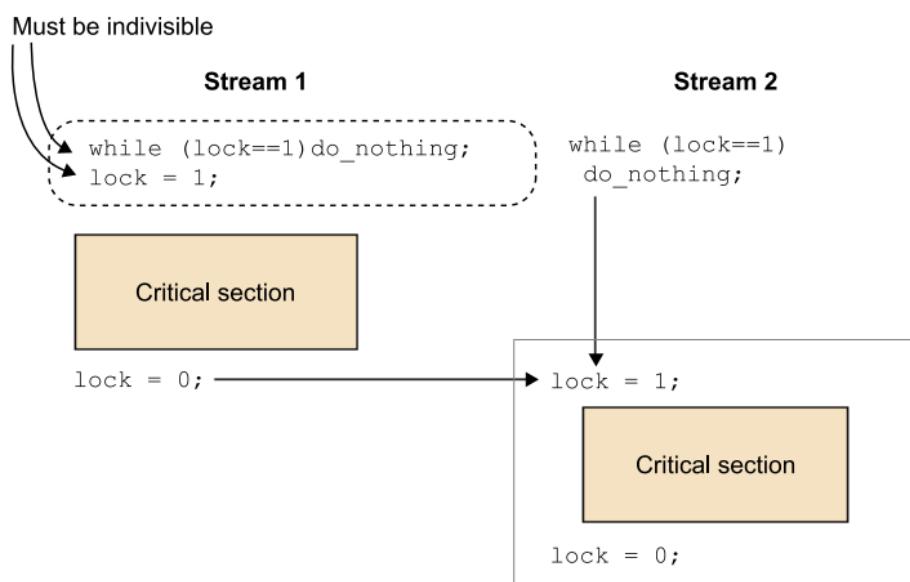
2. Process 2 is now in need of the shared resources.



3. Process 2 enters busy waiting till it has access to the shared resource.



- El mecanismo más simple se basa en candados, que son variables de un bit en las cuales, cuando su valor es 1, esto indica de que el *stream* ha entrado en la sección crítica, mientras que si es 0, entonces indica que aún no se ha entrado
- Una posible implementación de exclusión mutua se puede observar en lo que se conoce como *busy waiting* o *spin loops*, que consiste en hacer esperar a un *stream* y comprobar constantemente una condición con la variable candado para poder proceder a la ejecución de la sección crítica. En esta implementación se siguen los siguientes pasos:



- Hay que esperar a que el candado indique el que no hay ningún *stream* en esa sección crítica

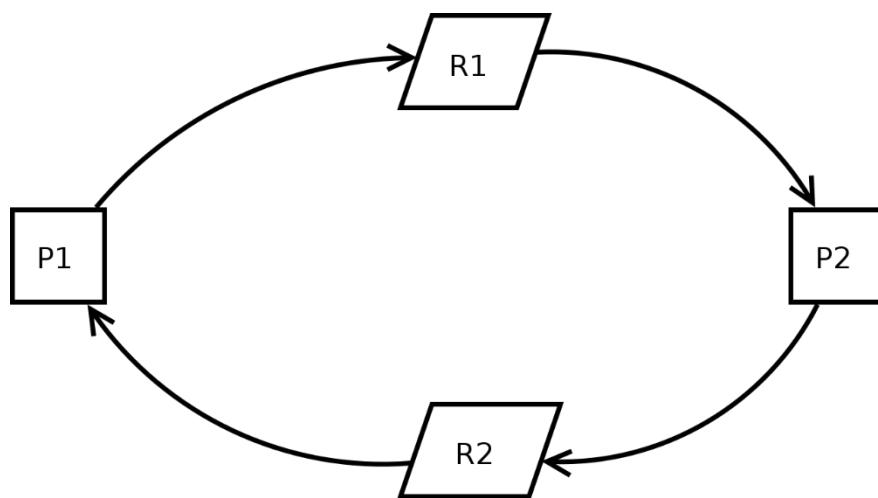
- Se bloquea la sección crítica cambiando el valor de la variable candado a 1
- Se ejecuta el código de la sección crítica y se bloquea esta cambiando el valor de la variable candado a 0 una vez se ha ejecutado
- La librería Pthreads implementa los candados directamente, usando variables candado de exclusión mutua conocidas como variables mutex. Las secciones críticas se definen de la siguiente manera:

```

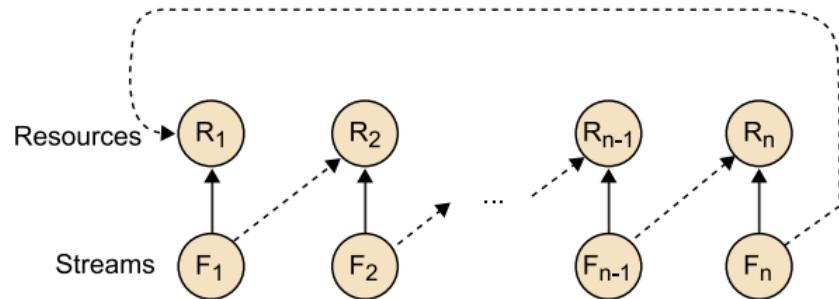
pthread_mutex_lock(&mutex1);
Sección crítica
pthread_mutex_unlock(&mutex1);

```

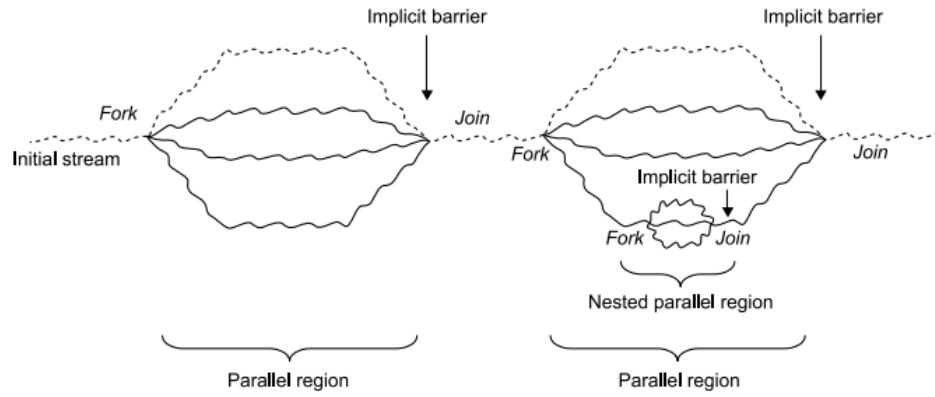
- Cuando un *stream* llega a *pthread\_mutex\_lock*, este permanecerá bloqueado hasta que el candado indique que se puede abrir otra vez. Si hay múltiples *streams* bloqueados en esta parte, el sistema determinará qué flujo puede acceder a la sección crítica cuando el candado se desbloquee
- Hay que recordar que el *stream* que bloquea el candado es el único que puede desbloquearlo otra vez
- Uno de los problemas más importantes con la exclusión mutua se conoce como *deadlock*, que es la situación en que dos *streams* ocurren cuando cada uno de los dos necesita el recurso que el otro *stream* está bloqueando. En otras palabras, ocurre cuando los *streams* esperan una acción de otros *streams* para continuar ejecutándose, pero como están igual, entonces se produce una espera eterna



- Este fenómeno se puede producir de manera circular, en cuyo caso se conoce como espera circular o *circular wait*



- Para prevenir *deadlocks*, Pthreads ofrece la operación `pthread_mutex_trylock()`, que permite hacer una *query* para revisar si una variable candado está bloqueada o no en el *stream*. Esta operación bloquea el candado y devuelve 0 si el candado estaba previamente abierto o *EBUSY* si estaba previamente bloqueado
- Aunque es posible usar Pthreads para desarrollar programas paralelos de memoria compartida, es necesario gestionar los *streams* y se tendrá que lidiar con los problemas anteriores y con problemas de sincronización. Se necesitan abstracciones de alto nivel que faciliten la programación y una ejecución eficiente de aplicaciones paralelas, lo cual es aportado por OpenMP
  - OpenMP consiste de una interfaz de programación que usan directivas o *pragmas* para expandir lenguajes de programación como C y Fortran. OpenMP es el estándar actual para la programación de sistemas de memoria compartida, tales como sistemas multinúcleo y computadoras de alto rendimiento basadas en multiprocesadores con una memoria virtual compartida
    - Se usa típicamente en sistemas con un número no demasiado alto de procesadores, aunque hay implementaciones de memoria compartida con cientos de núcleos
    - También hay versiones de OpenMP para otros tipos de sistemas, como clústeres y aceleradores. No obstante, en estos casos, el rendimiento de los programas puede ser menor que usando un entorno de programación específico aplicable a estos sistemas
  - El modelo de ejecución de OpenMP es el de *fork-join*, el cual es ilustrado en el siguiente esquema:



- El programa primero trabaja con un *stream* de instrucciones único hasta que llega a una región paralela (cuando llega al constructor `#pragma omp parallel`), que inicializa un número de *streams* esclavos (la parte de *fork* del modelo)
- Este *stream* trabaja inicialmente como el *stream* maestro para este conjunto de esclavos, y se numeran del 0 al número total de *streams* menos 1 (porque el 0 es el maestro). El maestro y los esclavos trabajan en paralelo, en el bloque que aparece después del constructor
- Al final de la región en paralelo existe una sincronización de todos los *streams*, y los esclavos se eliminan, quedándose así otra vez solo el *stream* maestro. Después de este punto, el maestro continúa trabajando de manera secuencial hasta que llega a otra región en paralelo y repite el proceso anteriormente descrito
- Un código prototípico de OpenMP requiere de algunos elementos básicos. Por ejemplo, se tiene que incluir la librería `omp.h` (donde se definen las funciones OpenMP), y usa directivas `#pragma omp` para decir al compilador cómo debería distribuirse el trabajo, o también se pueden usar cláusulas para poder realizar operaciones específicas

```

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int i, n;
    float a[100], b[100], sum;

    /* Some initializations - sequential region */

```

```

n = 100;
for (i=0; i < n; i++)
    a[i] = b[i] = i * 1.0;

sum = 0.0;

#pragma omp parallel for reduction(+:sum) /* parallel region */
for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);

printf("Sum = %f\n",sum); /* Sequential region */
}

```

- Las directivas de OpenMP siguen los estándares para la compilación de directivas en C/C++: son sensibles a mayúsculas y minúsculas, y solo un nombre de una directiva puede especificarse para cada directiva. La estructura general de una directiva es la siguiente:

```
#pragma omp directive_name [clauses, ...]
```

- En este caso, `#pragma omp` es una petición de OpenMP a directivas C/C++, `directive_name` es un nombre de válido de la directiva, y debe aparecer después de `pragma` y antes de cualquier cláusula, y `[clauses, etc.]` son cláusulas opcionales que pueden ir en cualquier orden y repetirse si es necesario (a no ser que haya restricciones)
- Cada directiva se aplica al menos a una oración subsiguiente, que puede ser un bloque estructurado de código. Para directivas largas, el carácter \ puede ser usado al final de la línea para separarla en diferentes trozos
- La directiva usada para crear los flujos esclavos es `parallel`, usado de la siguiente manera:

```
#pragma omp parallel [clauses]
block
```

- Un grupo de `streams` se crea y el `stream` que los comienza toma el rol de `stream` maestro. El número de `streams` que se deben crear se pueden obtener desde la variable del entorno `OMP_NUM_THREADS` o a través de una llamada a la librería
- Hay una barrera al final de la región que causa que el `stream` maestro espere a que los esclavos se finalicen antes de continuar con la ejecución secuencial

- Si hay otro constructor *parallel* dentro de una región, cada eslavo crea otro grupo de *streams* esclavos y se convierte en maestro de estos. Esto se conoce como paralelismo anidado o *nested parallelism*, y aunque programar de esta manera es posible, no siempre se puede realizar en algunas implementaciones
  - Las cláusulas para la compartición de variables que la directiva *parallel* incluye son sobre todo *private* , *firstprivate* , *lastprivate*, *default*, *shared*, *copyin* y *reduction*
- Las cláusulas anteriores indican como las directivas tienen que llevar a cabo la compartición de variables, con todas ellas estando en la memoria compartida. Cada directiva tiene una serie de cláusulas que se pueden aplicar a ellas:
  - La cláusula *private(list)* indica que las variables apareciendo en la lista son privadas para los *streams* (cada *stream* diferente tiene sus propias variables con el mismo nombre). Las variables no se inicializan antes de entrar en la región paralela (no se puede poner un valor inicial antes de entrar), y su valor no se guarda cuando se sale de la región (no se guarda al acabar la ejecución paralela)
  - La cláusula *firstprivate(list)* indica que las variables son privadas para los *streams* y que se inicializan cuando se entra en la región paralela usando el valor correspondiente a la variable en el *stream* maestro definida antes de la región en paralelo
  - La cláusula *lastprivate(list)* indica que las variables son privadas para los *streams* y que cuando se deja la región en paralelo, guardan su valor de la última iteración (si se está en un *loop* paralelo) o sección (del *thread* que se ejecute al final)
  - La cláusula de *shared(list)* indica que las variables se comparten por todos los *streams*. En la mayoría de casos, esta es la configuración estándar, de modo que realmente no se necesita usar esta cláusula
  - La cláusula *default(shared|none)* indica cómo estas variables deberían configurarse de manera predefinida. Si se especifica *none* , entonces se tiene que usar una cláusula *shared* para explícitamente especificar que se quiere compartir las variables
  - La cláusula *reduction(operator: list)* hace que las variables para la lista se obtengan por la aplicación de un operador (que debe ser asociativo), obteniendo así un solo resultado

- La cláusula *copyin(list)* se usa para asignar el valor de una variable en el *stream* maestro a una variable de tipo *threadprivate*
- Una vez que el conjunto de *streams* esclavos se ha generado usando *parallel*, estos *streams* y el maestro pueden trabajar en paralelo para resolver un problema
- Hay dos maneras de dividir el trabajo entre *streams*: usando una directiva *for* (paralelismo de datos) o una *sections* (paralelismo de funciones). Primero se revisa la directiva *#pragma omp for*

```
#pragma omp for [clauses]
for loop
```

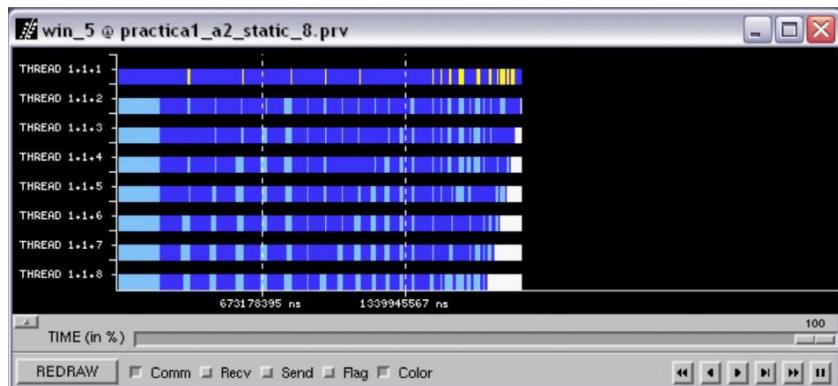
- Las iteraciones se ejecutan en paralelo por los *streams* que existen previamente (se tienen que crear anteriormente con *parallel*). El *loop* tiene que tener la forma canónica: la inicialización tiene que tener una tarea, la parte de incremento debe tener una suma o una resta, la parte de evaluación debe comparar una variable entera y su signo con otro valor, y los valores que aparecen en cada una de estas tres partes debe ser entero

$$\text{for}(\text{index}=\text{start}; \text{index} \left\{ \begin{array}{l} < \\ \leq \\ \geq \\ > \end{array} \right\} \text{final}; \left\{ \begin{array}{l} \text{index}++ \\ \text{++index} \\ \text{Index--} \\ \text{--index} \\ \text{index+=inc} \\ \text{index-=inc} \\ \text{index=index+inc} \\ \text{index=inc+index} \\ \text{index=index-inc} \end{array} \right\})$$

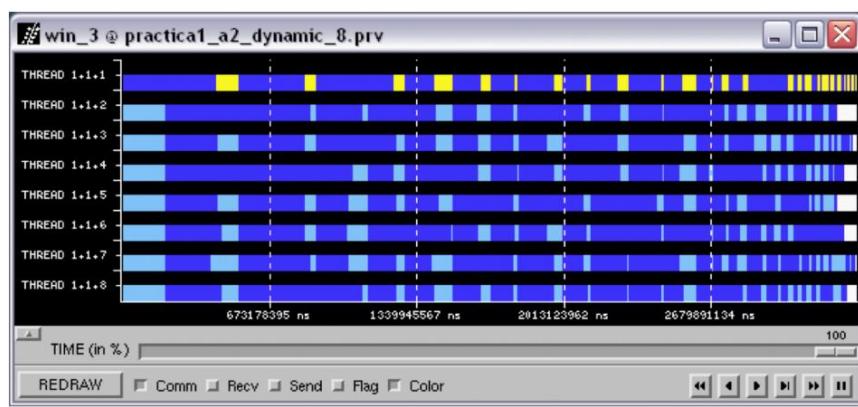
- A no ser que se usa la cláusula *nowait*, hay una barrera implícita al final del *loop* para esperar a que acaben las iteraciones paralelas
- Las cláusulas de compartición de variables admitidas son *private*, *firstprivate*, *lastprivate* y *reduction*
- Si se representa una directiva abreviada para incluir *parallel* y *for*, entonces se tienen las mismas cláusulas y todo, pero no se puede usar *nowait* porque acabaría el proceso de *parallel*

```
#pragma omp parallel for [clauses]
for loop
```

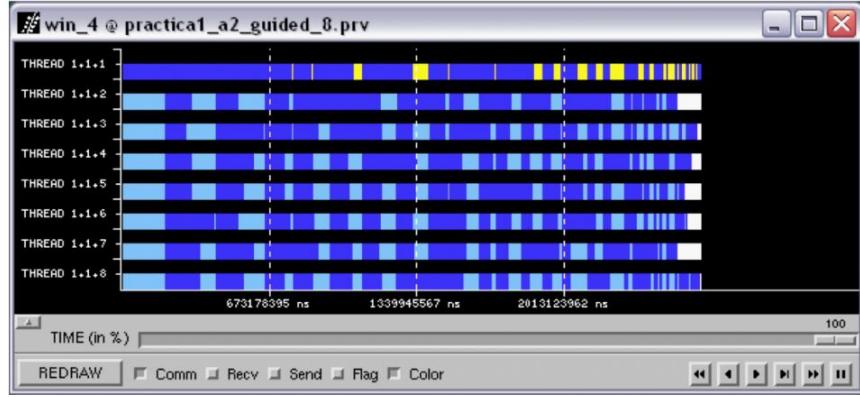
- La cláusula de *schedule* puede aparecer en esta directiva para indicar como las iteraciones de *for* deben dividirse entre *streams*, de modo que da una política de planificación. Las políticas de planificación disponibles en OpenMp son las siguientes:
  - La política de *schedule(static, size)* hace que las iteraciones se dividan acorde al tamaño de un bloque involucrado (habrá tantos bloques como grupos de *size* iteraciones se puedan hacer). Si se tiene un número de *streams* menor al número de bloques, estos se bloques se asignan de manera cíclica a los *streams*, y si no se indica *size*, entonces *size = nº iter./nº streams*



- La política de *schedule(dynamic, size)* hace que se agrupen las iteraciones acordes a *size* y que se asignen dinámicamente cuando se finalizan las tareas. Cuando el volumen de computación de cada iteración no se sabe previamente, es preferible usar una asignación dinámica porque de esta manera se reducen desequilibrios



- La política de *schedule(guided, size)* hace que las iteraciones se asignen dinámicamente para los *streams*, pero con un tamaño de bloque que se reduce hasta que se llega al valor especificado de *size* (1 por defecto). El tamaño inicial del bloque y la manera en la que decrece dependerá de la implementación



- La política *schedule(runtime)* hace que se decida la política de planificación en el tiempo de ejecución o *runtime* usando la variable de entorno *OMP\_SCHEDULE*
- La directiva *sections* toma la siguiente forma:

```
#pragma omp sections [clauses]
{
    [#pragma omp section]
    block
    [#pragma omp section
    block
    ...
}
}
```

- Cada sección se ejecuta usando un *stream*, y la manera en la que las secciones se distribuyen entre los *streams* depende de la implementación específica de OpenMP
- Hay una barrera final implícita a no ser que se especifique explícitamente *nowait*, y las cláusulas admisibles son *private*, *firstprivate*, *lastprivate* y *reduction*
- Si se representa una directiva abreviada para incluir *parallel* y *sections*, entonces se tienen las mismas cláusulas y todo, pero no se puede usar *nowait* porque acabaría el proceso de *parallel*

```
#pragma omp parallel sections [clauses]
```

- Dentro de una región paralela, con tal de prevenir situaciones de *deadlock*, puede ser necesario sincronizar los *streams* en el acceso para

variables específicas o áreas de código. OpenMP ofrece varias características con este propósito, como las siguientes:

- La característica *single*, en donde el código que se ve afectado por la directiva será ejecutado en un solo *stream*. Los *streams* que no estén trabajando durante la ejecución de la directiva esperarán hasta el final, y las cláusulas que se admiten son la de *private*, *firstprivate* y *nowait*
- Para la característica *single*, las bifurcaciones desde o hacia un bloque *single* no se permiten. Este bloque es útil para secciones de código que no puedan ser seguras para la ejecución en paralelo (por ejemplo, para *input/output*)
- La característica *master*, en donde el código se ejecuta solo por el *stream* maestro. El resto de *streams* no se ejecutan en esta sección del código
- La característica *critical*, en donde se protege a una sección de código de todo que solo un *stream* puede acceder a esta sección al mismo tiempo. Las bifurcaciones desde o hacia un bloque *critical* no están permitidas
- Se le puede asociar un nombre a esta sección, por lo que significa que puede haber secciones críticas protegiendo diferentes áreas del programa, haciendo que varios *streams* a la vez pueden acceder a secciones con diferentes nombres. Las regiones con el mismo nombre se tratan como si fueran la misma región, t

```
#pragma omp critical [name]
```

- La característica *atomic*, en donde se asegura que la localización de la memoria se modifique sin múltiples *streams* intentando escribir sobre esa localización simultáneamente. Esta directiva aplica a la frase a la que sigue, y en modo exclusivo, solo se asegura la actualización de la variable, pero no la evaluación de la expresión
- La característica *barrier*, en donde se sincronizan todos los *streams*. Cuando el *stream* alcanza la barrera, espera a que los otros lleguen, y una vez todos han llegado, entonces la ejecución continua
- La característica *threadprivate*, en donde se convierten variables globales en variables locales perteneciendo al *stream* a través de múltiples regiones en paralelo

- La característica *ordered*, la cual asegura que el código se ejecute en el mismo orden en que las iteraciones se ejecutan en su forma secuencial. Esto puede aparecer solo una vez en el contexto de un *for* o *parallel for* y las bifurcaciones desde o hacia este bloque no están permitidas
  - En una sección *ordered* solo puede haber un flujo ejecutándose simultáneamente, y una sola iteración de un bucle no puede ejecutar la misma directiva *ordered* más de una vez, y no hay necesidad de que ejecute más de una directiva *ordered*
  - La característica *flush*, la cual asegura de que los valores de las variables se actualicen en todos los *streams* en donde sean visibles. Si no hay lista de variables, todas serán actualizadas
- Las funciones que son más usadas en OpenMP son aquellas relacionadas al número de *streams*, como *omp\_set\_num\_threads* , *omp\_get\_num\_threads* y *omp\_get\_thread\_num*. Las otras funciones importantes son las siguientes:
  - La función *omp\_get\_max\_threads*, la cual obtiene el número máximo de *streams* posibles
  - La función *omp\_get\_num\_procs*, la cual obtiene el número de procesadores que se pueden asignar al programa
  - La función *omp\_in\_parallel*, la cual obtiene un valor diferente a cero si se está ejecutando una región en paralelo (1, por ejemplo)
  - La función *omp\_set\_dynamic*, que permite encender o apagar la asignación dinámica del número de *streams* en las regiones paralelas
  - La función *omp\_get\_dynamic*, que permite ver si se tiene permiso para un ajuste dinámico, devolviendo un valor diferente a cero si el ajuste dinámico del número de *streams* está permitido
  - La función *omp\_set\_nested* , que permite desautorizar el paralelismo anidado
  - La función *omp\_get\_nested* , que permite obtener un valor diferente a cero si se permite el paralelismo anidado
- También hay funciones para gestionar los *locks*, tales como las siguientes:
  - La función *void omp\_init\_lock(omp\_lock\_t \* lock)* , que permite inicializar un *lock* como “no puesto”

- La función `void omp_init_destroy(omp_lock_t * lock)`, que permite destruir un *lock*
- La función `void omp_set_lock(omp_lock_t * lock)` , que permite fijar un *lock*
- La función `void omp_unset_lock(omp_lock_t * lock)` , que permite desfijar un *lock*
- La función `void omp_test_lock(omp_lock_t * lock)` , que permite comprobar si un *lock* está fijado para evitar *locks* indeseados
- Con tal de compilar un programa OpenMP, se necesita un compilador que pueda interpretar las directivas que aparecen en el código, y desde la versión 4.1, OpenMP puede compilarse con el compilador GCC y por otros compiladores propietarios como ICC de Intel
  - La opción de compilación de GCC es `-fopenmp` o `-openmp`, mientras que para ICC es `-openmp`

```
gcc -fopenmp -o program program.c
```

- La ejecución del archivo del programa binario se da como con cualquier otro programa, excepto que se debe determinar el número de *streams* que se van a involucrar en la ejecución en paralelo. Se puede usar una variable de entorno como `OMP_NUM_THREADS`, que dice el número de *streams* que se deben usar
  - Si esta variable no se inicializa, tendrá un valor predeterminado cuando se ejecute el programa OpenMP. Este valor normalmente coincide con el número de núcleos en el nodo con el que se trabaja
  - Los valores para la variable se pueden establecer antes de la ejecución, por ejemplo, desde la interpretación de comandos. En esta situación, se establece el valor de la variable de la siguiente manera:

```
export OMP_NUM_THREADS=6
```

- Se puede establecer el número de *streams* que se pueden usar en la región paralela independientemente del número de núcleos que se tenga disponible, dando así la posibilidad de que se usen más *streams* que procesadores hay en el sistema. En términos

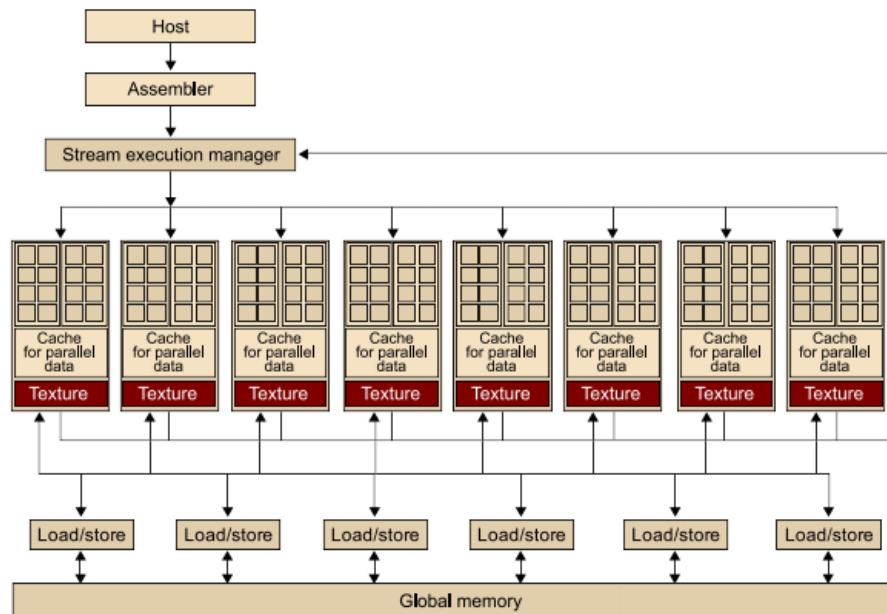
generales, el programa OpenMP que se esté ejecutando utilizando más *streams* que procesadores disponibles tendrá un rendimiento menor de lo que se puede conseguir usando un *stream* por procesador (incluso un tiempo mayor que con ejecución secuencial)

- Aunque no se tengan suficientes procesadores para ejecutar todos los *streams*, puede ser posible obtener un mejor rendimiento al mejorar las particiones y las asignaciones de datos (usando, por ejemplo, páginas de memoria de caché para acelerar la ejecución)
- Existen tres variables de entorno además de *OMP\_NUM\_THREADS*, las cuales son las siguientes:
  - La variable *OMP\_SCHEDULE*, la cual indica el tipo de planificación para *for* y *parallel for*
  - La variable *OMP\_DYNAMIC*, la cual autoriza o desautoriza el ajuste dinámico del número de *streams*
  - La variable *OMP\_NESTED*, que autoriza o desautoriza paralelismo anidado (siendo desautorizado como valor predeterminado)

## \*La programación en paralelo: modelos gráficos

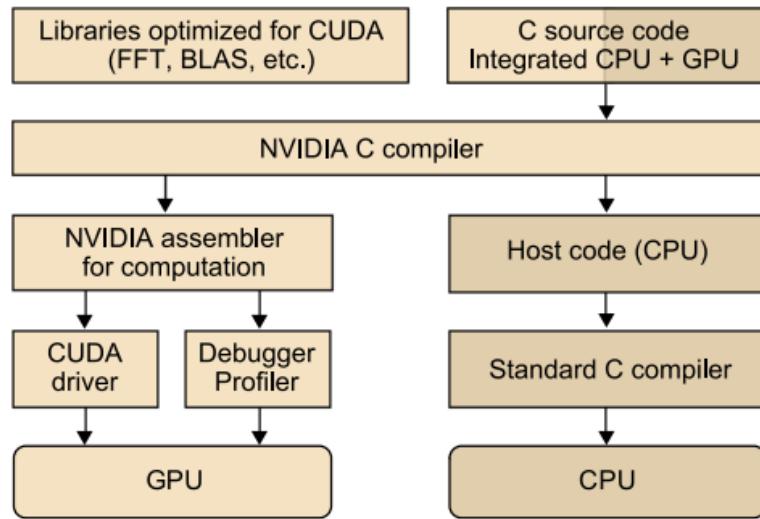
- En años recientes, se ha encontrado que, para llevar a cabo ciertas tareas, la capacidad de cálculo de las unidades de procesamiento gráfico (GPUs) es más grande que las capacidades encontradas en muchos procesadores multinúcleo avanzados. Esta situación ha hecho que este tipo de dispositivo se haga muy popular no solo para generar gráficos, sino también para el cálculo de algoritmos generales, lo cual dio pie al *general purpose GPU computing* o GPGPU
  - Debido a las características arquitecturales y funcionales, las GPUs pueden proporcionar un gran rendimiento solo para ciertas aplicaciones
    - En general, se puede decir que las aplicaciones que pueden tomar ventaja de las capacidades de una GPU son aquellas que trabajan con grandes vectores de datos y aquellas que tienen tipos de paralelismo SIMD
  - Cuando se trabaja con una GPU, una de las mayores dificultades para el programador es transformar programas diseñados para CPUs tradicionales en programas que se pueda construir y ejecutar eficientemente en una GPU

- Por esta razón, varios modelos de programación se han desarrollado para que el programador pueda tener un nivel de abstracción más cerca a aquel que se obtendría cuando se programa para una CPU, simplificando el proceso
  - Los modelos más famosos son CUDA (de NVIDIA) y OpenCL (que es *open source*)
- CUDA proviene de *compute unified device architecture*, y es una especificación desarrollada por NVIDIA para sus productos GPU. CUDA incluye especificaciones para la arquitectura y un modelo de programación asociado
  - La arquitectura de una GPU compatible con CUDA (genérica) se organiza en multiprocesadores *streaming* o *streaming multiprocessors* (SMs), que tienen un gran número de *streams* de ejecución, y estos organizan en bloques, de modo que un bloque tiene un número de SMs (dependiendo de la implementación para cada dispositivo)

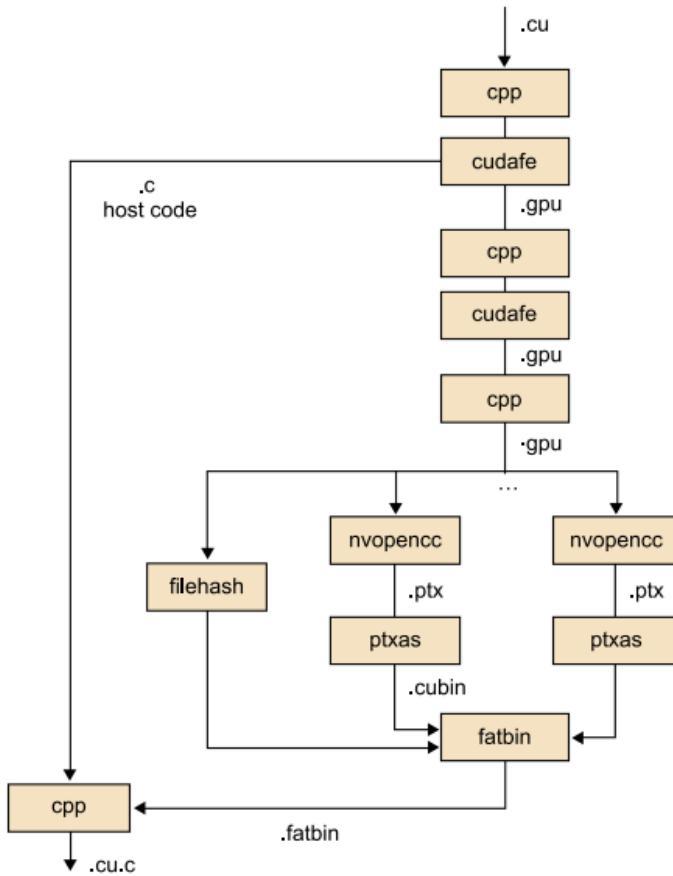


- Además, cada SM tiene un cierto número de procesadores de *streaming* o *streaming processors* (SPs), que comparten la lógica de control y da instrucciones a la memoria de caché
  - La GPU tiene una memoria DRAM a una tasa de GDDR (o *graphics double data rate*), que se conoce como la memoria global del dispositivo. Esta memoria es diferente de la memoria DRAM de la computadora porque se usa esencialmente para gráficos (llamado *framebuffer*), donde guarda imágenes de video y la información de las texturas

- No obstante, cuando se usa para cálculos de propósito general, funciona como una memoria externa con una gran banda ancha, pero también con una latencia que es un poco más alta que la típica de la memoria del sistema. En aplicaciones paralelas masivas, la mayor banda ancha compensa por la mayor latencia
  - La GPU NVIDIA G80 tiene 768 *streams* por SM, por lo que tiene 12000 *streams* en un solo chip, mientras la GPU NVIDIA GT2000 tiene 1024 *streams* por SM, teniendo 30000 *streams* en total. Por lo tanto, se puede ver una tendencia creciente del nivel de paralelismo de las GPUs, por lo que es importante explorar este alto nivel de paralelismo para aplicaciones de propósito general
- CUDA se desarrolló para incrementar la productividad al diseñar aplicaciones de propósito general para GPUs. Desde el punto de vista del programador, el sistema consiste en un *host* (una CPU tradicional con arquitectura Intel) y uno o más dispositivos GPU
  - CUDA pertenece al modelo SIMD, por lo que está diseñado para explotar el paralelismo a nivel de datos, lo que significa que las operaciones aritméticas se pueden ejecutar en un conjunto de datos simultáneamente. Afortunadamente, muchas aplicaciones tienen partes con un alto nivel de paralelismo en un entorno de datos
  - Un programa en CUDA consiste de una o más fases que se pueden ejecutar tanto en el *host* como en el dispositivo GPU. Las fases con poco o ningún paralelismo en un entorno de datos se implementan en el código que se ejecutará por el procesador principal, mientras que las fases con un nivel alto de paralelismo en el entorno de datos se implementarán en el código que se ejecutará por la GPU
- Durante el proceso de compilación, el compilador de NVIDIA (el binario principal es *nvcc*) es responsable de proporcionar las partes del código correspondiente a ambos el *host* y el dispositivo



- El código correspondiente al *host* es simplemente código ANSI C, que se compila usando el compilador de C estándar como para la CPU. El código correspondiente al dispositivo también es ANSI C, pero con extensiones para incluir palabras clave para definir funciones usadas para procesar los datos en paralelo (estas funciones se conocen como *kernels*)
- El código para el dispositivo se compila otra vez usando *nvcc*, y entonces ya estará listo para ser ejecutado por la GPU. En situaciones en donde no hay un dispositivo disponible, o donde el *kernel* es más apropiado para la CPU, uno también puede ejecutar los *kernels* en una CPU convencional usando emuladores proporcionados por la plataforma CUDA

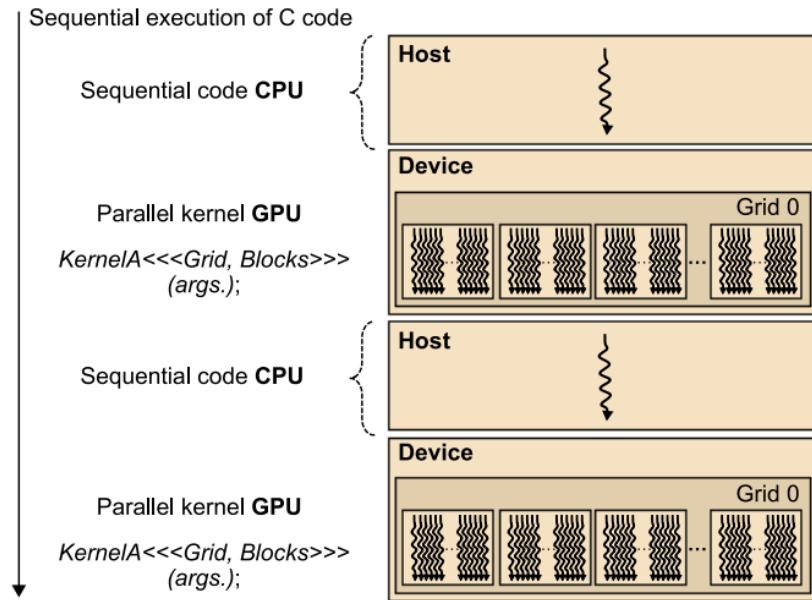


- La siguiente tabla describe como el compilador *nvcc* interpreta los diferentes tipos de archivos de insumo:

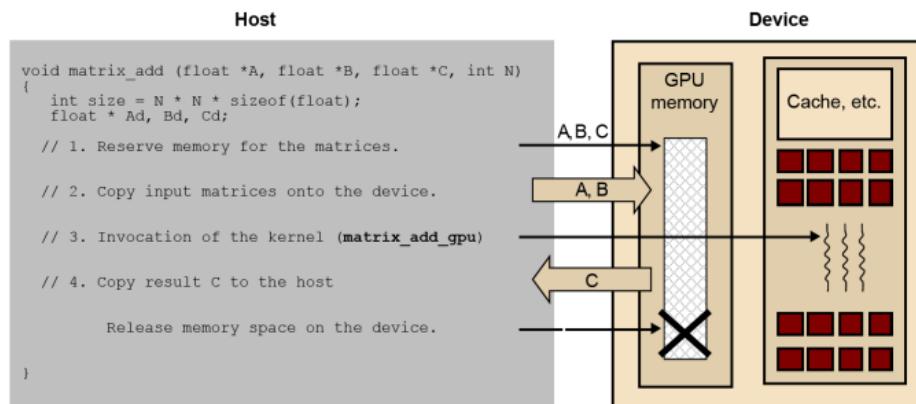
.cu	CUDA source code, which contains both the host's code and the functions for the device.
.cup	Pre-processed CUDA source code, which contains both the host's code and the functions for the device.
.c	Source code file in C
.cc, .cxx, .cpp	Source code file in C++
.gpu	Intermediate GPU file
.ptx	Intermediate ptx assembler file
.o, .obj	Object file
.a, .lib	Library file
.res	Resource file
.so	Shared object file

- Con tal de explotar el paralelismo a nivel de datos, los *kernels* deben generar un gran número de *streams* de ejecución (decenas o cientos de miles de *streams*). Se tiene que tener en cuenta que los *streams* de CUDA usan un código más ligero que aquellos de CPU, y debido al *hardware* solo se necesitan unos cuantos ciclos de reloj cuando se generan y se

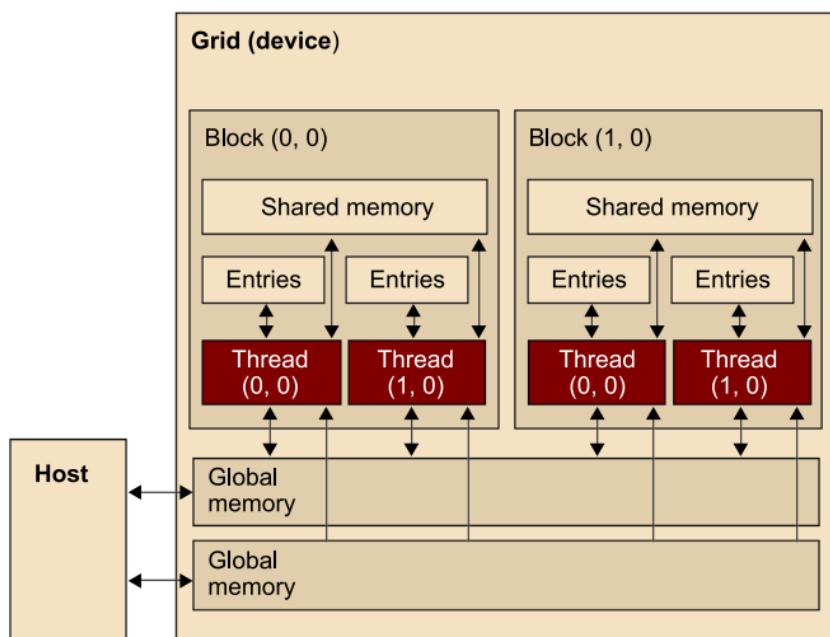
planean estos *streams* (en contraste a *streams* convencionales que necesitan cientos de ciclos)



- La ejecución de un programa CUDA suele comenzar por la ejecución del *host* (CPU), y cuando el *kernel* se llama, entonces la ejecución se mueve al dispositivo GPU, con un gran número de *streams* generados. Cuando un *kernel* se invoca, el conjunto entero de *streams* generados se conoce como *grid*
- Un *kernel* acaba cuando la ejecución se finaliza para todos los *streams* del *grid* correspondiente. Una vez que el *kernel* finaliza, el programa sigue ejecutándose en el *host* hasta que otro *kernel* se invoca
- Es importante mencionar que la memoria del *host* y la memoria del dispositivo son dos espacios de memoria completamente separados, lo cual refleja el hecho de que los dispositivos normalmente tienen su propia memoria DRAM. Los siguientes pasos deben ser usados para ejecutar el *kernel* en un dispositivo GPU:



- Primero se reserva la memoria en el dispositivo GPU
- Después se transfieren los datos necesarios del *host* al espacio de memoria asignado para el dispositivo
- Se llama a la ejecución del *kernel* que se necesita
- Finalmente, se transfieren los datos con los resultados del dispositivo al *host*, y después se libera la memoria (si no se necesita) del dispositivo una vez el *kernel* ha finalizado
- El modelo de memoria CUDA presentado al programador en términos de asignación, transferencia y uso de diferentes tipos de dispositivos de memoria es el siguiente:



- El *host* puede realizar una transferencia de datos bidireccional a la memoria global y constante por *grid*
- Desde la perspectiva del dispositivo, los diferentes tipos de memorias pueden accederse a través del acceso de escritura/lectura a la memoria global por *grid*, del acceso de lectura a la memoria constante por *grid*, del acceso de escritura/lectura a las entradas por *thread*, del acceso de escritura/lectura a la memoria local por *thread* y del acceso de escritura/lectura a la memoria compartida por bloque
- La interfaz del programa que asigna y libera la memoria global del dispositivo consiste de dos funciones básicas principales: *cudaMalloc()* y *cudaFree()*

- La primera se puede llamar desde el *host* para asignar un espacio de memoria local a un objeto. La función *cudaMalloc()* tiene dos variables: la primera es la dirección del puntero o *pointer* hacia el objeto (una vez se ha asignado el espacio de memoria), la cual es del tipo *void\*\** (no depende de ningún tipo de dato), y la segunda libera el espacio de memoria designado para el objeto

```

float *Matrix;
int size = WIDTH * LENGTH * sizeof(float);
cudaMalloc((void **) &Matrix, size);
...
cudaFree(Matrix);

```

- Una vez que el programa ha asignado memoria global para los objetos de los datos del programa, se pueden transferir datos como necesarios para el cálculo del *host* del dispositivo. Esto se realiza mediante la función *cudaMemcpy()*, eso permite que haya transferencia de datos (asíncrona) entre memorias
- Esta función tiene cuatro variables: la primera es un puntero a la destinación objetivo en donde se copiaran los datos, la segunda variable apunta a los datos para ser copiados, la tercera variable especifica el número de *bytes* para ser copiados, y la cuarta indica el tipo de memoria que se involucra al copiar
- Los tipos de memoria son *cudaMemcpyHostToHost* (de la memoria del *host* a la misma memoria del *host*), *cudaMemcpyHostToDevice* (de la memoria del *host* a la memoria del dispositivo), *cudaMemcpyDeviceToHost* (de la memoria del dispositivo a la memoria del *host*) y *cudaMemcpyDeviceToDevice* (de la memoria del dispositivo a su misma memoria del dispositivo)
- Esta función se puede usar para copiar los datos de una memoria a ella misma, pero no entre diferentes dispositivos

```

void matrix_add (float *A, float *B, float *C, int N)
{
    int size = N * N * sizeof(float);

    float * Ad, Bd, Cd;

    // 1. Reserve memory for matrices
    cudaMalloc((void **) &Ad, size);
    cudaMalloc((void **) &Bd, size);
    cudaMalloc((void **) &Cd, size);

    // 2. Copy entry matrices to the device
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    ...

    // 4. Copy the result C to the host
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    ...
}

```

- Además de los mecanismos de asignación de memoria y transferencia de datos, CUDA también soporta diferentes tipos de variables. Las diferentes variables usando los diferentes tipos de memoria se usan dentro de varios focos con distintos ciclos de vida

Declaration variables	Memory types	Scope	Life cycle
Default (different from vectors)	Entry	Thread	Kernel
Default vectors	Local	Thread	Kernel
<code>__device__ __shared__ int SharedVar;</code>	Shared	Block	Kernel
<code>__device__ int GlobalVar;</code>	Global	Grid	Application
<code>__device__ __constant__ int ConstVar;</code>	Constant	Grid	Application

- El código ejecutado en el dispositivo *kernel* es la función que los diferentes *threads* ejecutan durante la fase paralela, donde cada uno lo ejecuta con su rango de datos correspondiente. Hay que recordar que CUDA sigue un modelo SPMD, por lo que todos los *threads* ejecutan el mismo código

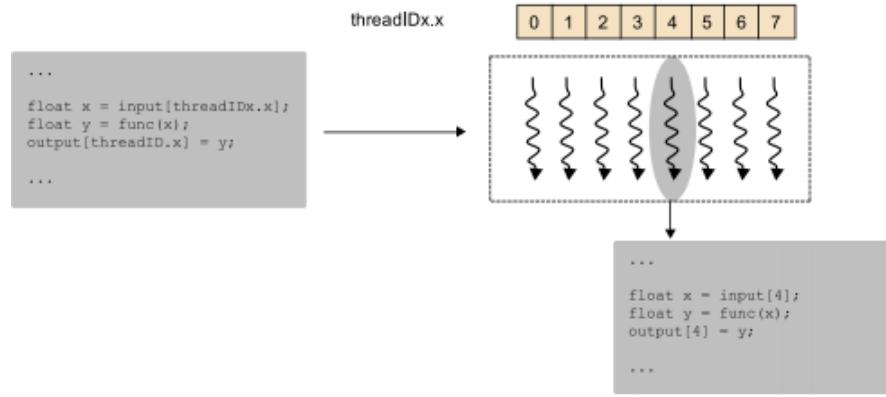
```

__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];
    }
}

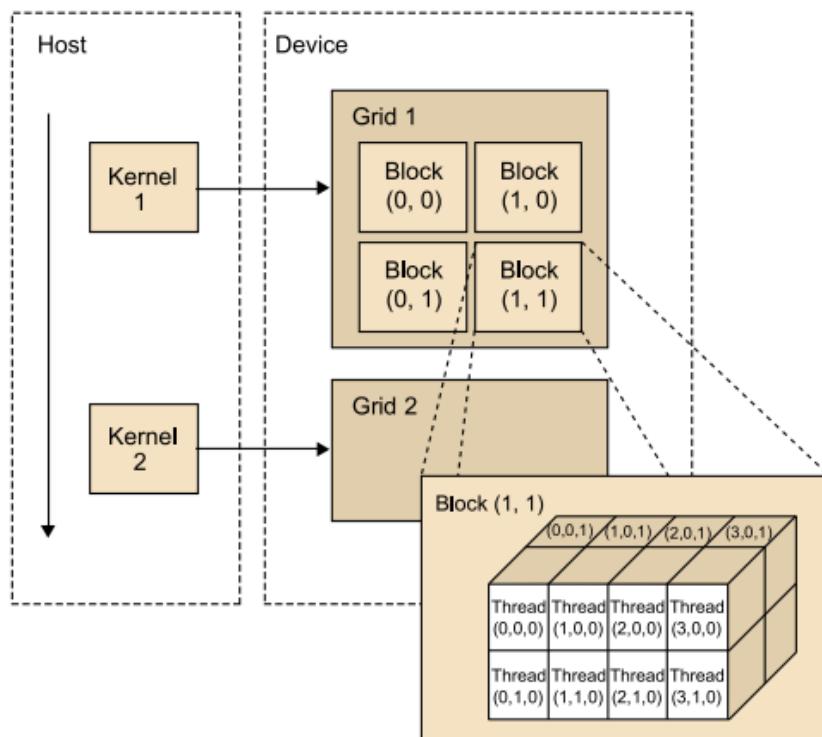
int main(){
    dim3 dimBlock(blocksize, blocksize);
    dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
    matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
}

```

- El uso de la palabra clave `_global_` antes de la declaración del nombre de la función del *kernel* indica que esta función es un *kernel* y que se debe llamar desde el *host* para generar un *grid* de *threads* que ejecutarán el *kernel* en el dispositivo. Además de `_global_`, hay dos otras palabras clave antes de la declaración de una función: `_device_` y `_host_`
- La palabra clave `_device_` indica que la función declarada es una función de dispositivo CUDA. Una función de dispositivo solo se ejecuta en un dispositivo CUDA y solo se puede llamar desde el *kernel* o desde otra función de dispositivo
- La palabra clave `_host_` indica que la función es una función *host*, de modo que una simple función de C que ejecuta dentro del *host* y, consecuentemente, puede ser llamado desde cualquier función de *host*. Por defecto, todas las funciones en un programa CUDA son funciones *host* si no se especifican palabras clave en la definición de función
- Las palabras clave `_host_` y `_device_` se pueden usar simultáneamente en la declaración de una función. Esto hace que el compilador genere dos versiones de la misma función, en donde una se ejecuta en el *host* y la otra en el dispositivo, y solo se pueden llamar desde sus respectivas localizaciones
- En CUDA, el *kernel* se ejecuta usando un grupo de *threads* (un vector o matriz de *threads*), y debido a que los *threads* ejecutan el mismo *kernel*, se sigue el modelo SIMT (de *single instruction multiple threads*) y requiere que el mecanismo los diferencie y asigne la parte de datos correspondientes a cada *thread*. Por ello, CUDA incorpora palabras clave para referirse al índice de cada *thread*



- El *kernel* se refiere al identificador del *thread* y, durante la ejecución, este se reemplaza por su valor correspondiente en cada uno de los *threads*. Consecuentemente, las variables *threadIdx.x* y *threadIdx.y* tienen diferentes valores para cada uno de los *threads* de ejecución
- Las coordenadas reflejan una organización multidimensional de los *threads* de ejecución
- Normalmente, el *grid* que se usa en CUDA está hecho de varios *threads* (miles o millones), y estos se organizan en una jerarquía de dos niveles: *grids* y bloques



- Cada bloque en un *grid* tiene una única coordenada dentro de un espacio bidimensional usando *blockIdx.x* y *blockIdx.y*. Estos

se organizan dentro de un espacio tridimensional con un máximo de 512 *threads* de ejecución

- Las coordenadas de los *threads* de ejecución se identifican, por tanto, por tres componentes *threadIdx.x*, *threadIdx.y* y *threadIdx.z*, aunque no todas las aplicaciones usan las tres dimensiones
- Cuando el *host* llama un *kernel*, se genera un *grid* y las dimensiones de este y los bloques de *threads* se suelen especificar en la configuración de ajustes. El primer ajuste especifica las dimensiones del *grid* en términos de bloques y el segundo especifica las dimensiones de los bloques en términos de *threads*, con ambas dimensiones del tipo *dim3* (una estructura con 3 campos de tipo *unsigned int*)
- Como los *grids* son bidimensionales, el tercer campo de la configuración se suele ignorar

```
// Blocks and grid dimensions configuration
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);

// Call to kernel (sum of matrices)
matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
```

- CUDA también ofrece un mecanismo de sincronización de *threads* para el mismo bloque usando una función de barrera *\_syncthreads()*
  - Cuando esta función se llama, el *thread* que la ejecuta se bloquea hasta que los *threads* en el bloque hayan llegado al mismo punto. Esto sirve para asegurarse de que todos los *threads* en el bloque han completado una fase antes de pasar a la siguiente
- OpenCL es una interfaz abierta, gratuita u multiplataforma para la computación en paralelo, nacida con la motivación de simplificar la tarea de hacer programas portátiles y eficientes para un gran número de plataformas heterogéneas como CPUs, GPUs o hasta sistemas incrustados
  - OpenCL consta de tres partes: una especificación de lenguaje multiplataforma, una interfaz dentro del alcance de un entorno de programación y una interfaz para coordinar la computación paralela entre procesadores heterogéneos
    - OpenCL usa un subgrupo C99 con extensiones para paralelismo y la representación estándar IEEE 754 para garantizar la interpretabilidad entre plataformas

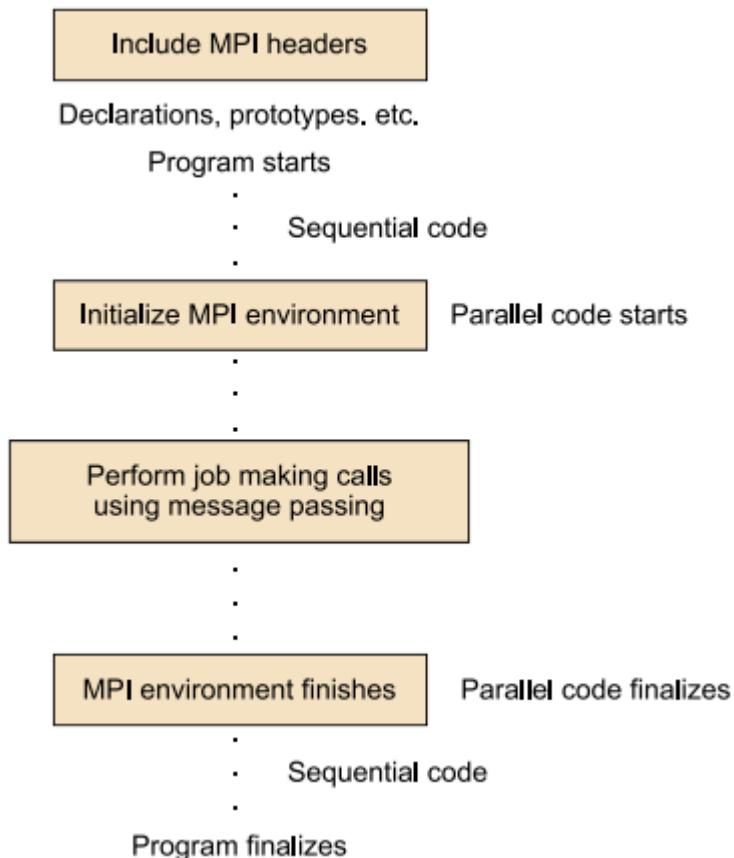
- Hay muchas similitudes entre OpenCL y CUDA, aunque OpenCL tiene un modelo de gestión de recursos complejo, dado que soporta plataformas múltiples y portabilidad entre diferentes manufacturas. OpenCL soporta modelos de paralelismo dentro de las áreas de datos y de tareas
- De la misma manera que CUDA, un programa OpenCL consiste en dos partes: *kernels* que se ejecutan en uno o más dispositivos y un programa *host* que invoca y controla la ejecución de los *kernels*
  - Cuando se realiza una invocación de *kernel*, el código se ejecuta en elementos de tareas que corresponden a *threads* de CUDA. Los elementos de trabajo o *work items* y los datos asociados a cada uno se definen en el punto en un espacio de índices *n*-dimensional (NDRange), y los elementos de trabajo del grupo de trabajo o *work groups* corresponden a los grupos de CUDA
  - Los elementos tienen un identificador global que es único, y los grupos de trabajo están identificado dentro del rango *n*-dimensional para cada grupo, y cada uno de los elementos tienen un identificador local que va desde cero al tamaño del grupo. Consecuentemente, la combinación del identificador del grupo con el identificador local dentro del grupo identifica de manera única al elemento de trabajo
  - Algunas equivalencias entre OpenCL y CUDA son las siguientes:

OpenCL	CUDA
kernel	kernel
Host program	Host program
NDRange (N-dimensional range)	Grid

OpenCL	CUDA
Work item	Thread
Work group	Block
<code>get_global_id(0);</code>	<code>blockIdx.x * blockDim.x + threadIdx.x</code>
<code>get_local_id(0);</code>	<code>threadIdx.x</code>
<code>get_global_size(0);</code>	<code>gridDim.x*blockDim.x</code>
<code>get_local_size(0);</code>	<code>blockDim.x</code>

## La programación en paralelo: modelos de memoria distribuida

- Tal como se ha explicado anteriormente, MPI es una librería de *message-passing* que puede ser usada por programas de C y Fortran, desde los cuales se puede llamar a funciones MPI para la gestión y la comunicación de procesos
  - La estructura de un programa MPI es la mostrada en el siguiente esquema:



- Inicialmente, un solo proceso ejecuta el código secuencialmente hasta que el entorno MPI se inicializa (momento en que el código paralelo comienza, indicado por una función)
- En la sección de código paralelo, la tarea concreta que será realizada y los mensajes serán usados para comunicar los diferentes procesos MPI
- Finalmente, una vez el trabajo que se tenía que realizar en paralelo se ha acabado, el fin del código en paralelo se indica (con una función) y el código restante comienza a ejecutarse secuencialmente otra vez hasta que el programa se ha ejecutado completamente

- Contrario a OpenMP, el número de procesos no puede ser establecido durante el tiempo de ejecución (el paso en el que se ejecutan las instrucciones). En cambio, se necesita especificar cuándo debería ejecutarse el programa
- El modelo de un solo programa y múltiples datos (SPMD) es una de las maneras más usuales de utilizar MPI

```

main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // See the process range

    if (myrank == 0)
        master(); // code that the master process executes
    else
        slave(); // code that the slave processes execute

    MPI_Finalize();
}

```

- En este caso, es necesario diferenciar el código que el proceso maestro utiliza (típicamente, el proceso con un rango o identificador igual a cero) desde el cual los procesos esclavos se ejecutan
- Un comunicador permite definir una serie de procesos que incluyen aquellos que realizan las comunicaciones
  - Cada proceso se puede encontrar en varios comunicadores a la vez y tendrán un identificador en cada uno de ellos (conocidos como rangos), los cuales van de 0 hasta el número de procesos en el comunicador menos 1. Cuando MPI comienza, todos los procesos están en *MPI\_COMM\_WORLD*, una constante que identifica un comunicador que incluye a todos los procesos
  - Aparte de *MPI\_COMM\_WORLD*, uno puede definir comunicadores con un número de procesos más pequeños, por ejemplo, para comunicar datos en cada una de las filas de los procesos en una rejilla o *grid*, permitiendo la transmisión en las diferentes filas de procesos sin que las comunicaciones de uno afecten a las demás. Aún así, también se pueden usar para comunicados de punto a punto
- Hay dos tipos de comunicadores: los intracomunicadores y los intercomunicadores. Los intracomunicadores se usan para enviar mensajes entre procesos dentro del comunicador, mientras que los

intercomunicadores existen para enviar mensajes entre procesos en diferentes comunicadores

- Las comunicaciones entre procesos en diferentes comunicadores tienen sentido cuando se diseñan librerías: los comunicadores se crean y un usuario de una librería desea comunicar un proceso de su programa a otra librería MPI
- Un comunicador crea un grupo que consiste de una colección ordenada de procesos a los que se les asocia identificadores y un contexto, que es un identificador asociando el sistema al grupo
- En MPI hay dos tipos básicos de comunicadores: los comunicadores de punto a punto o *point-to-point*, los cuales involucran dos procesos que intercambian mensajes solamente entre ellos, y los comunicadores colectivos o *collective*, los cuales involucran una combinación de procesos realizando una tarea específica colectivamente. Primero se empieza viendo los comunicadores de punto a punto
  - En ejemplos anteriores, los procesos no interactúan entre ellos. Es normal que los procesos no trabajen independientemente, sino que intercambien información usando *message-passing*
    - Para enviar mensajes entre dos procesos (un proceso de origen y un proceso destino) se usan los comunicadores de punto a punto
    - En el fragmento de código MPI mostrado, un proceso con rango 0 envía un entero  $x$  a un proceso con rango 1. El programa muestra la manera normal de trabajar con *message-passing*

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    int x;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

- El mismo código se ejecuta en todos los procesos, pero se ejecutan diferentes procesos en diferentes partes del código. En el ejemplo, las funciones *MPI\_Send* y *MPI\_Recv* se usan para enviar y recibir mensajes respectivamente

```

int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)

```

- La opción *buf* contiene el comienzo de la región de memoria desde la cual los datos se toman para ser enviados o donde se almacenarán los datos recibidos (puede servir como un arreglo o una fuente de donde se envían datos o como un *buffer* en donde se almacenan los datos que se han enviado)
- La opción *count* indica el número de elementos de datos que serán enviados o el espacio disponible para recibir el mensaje (no el número de elementos de datos del mensaje recibido, dado que esto se determina por el remitente o *sender*)
- La opción *datatype* es el tipo de datos que se transfieren, y deben ser un tipo de datos MPI. Todos los mensajes deben tener un tipo de dato, los cuales pueden ser los siguientes:

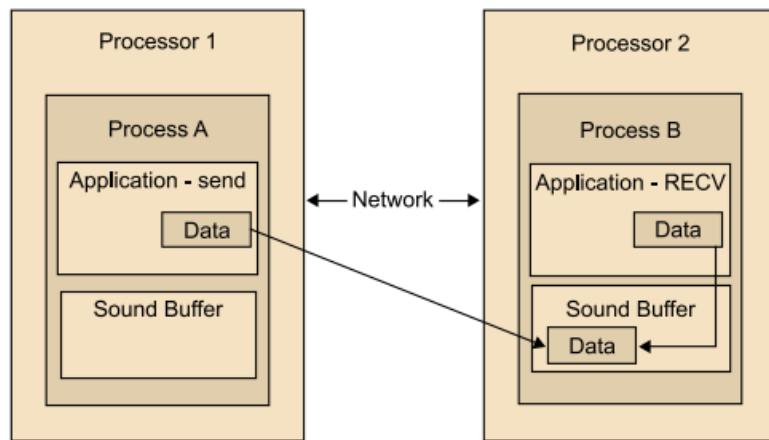
C data types	
MPI_CHAR	Signed char
MPI_WCHAR	Wchart_t wide character
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed
MPI_LONG_LONG_INT MPI_LONG_LONG	Signed long long int
MPI_SIGNED_CHAR	Signed char
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNISGNED	Unsigned int
MPI_SIGNED_CHAR	Signed char

<code>MPI_UNSIGNED_CHAR</code>	<code>Unsigned char</code>
<code>MPI_FLOAT</code>	<code>Float</code>
<code>MPI_DOUBLE</code>	<code>Double</code>
<code>MPI_LONG_DOUBLE</code>	<code>Long double</code>
<code>MPI_C_COMPLEX</code> <code>MPI_C_FLOAT_COMPLEX</code>	<code>Float_Complex</code>
<code>MPI_C_DOUBLE_COMPLEX</code>	<code>Double_Complex</code>
<code>MPI_C_LONG_DOUBLE_COMPLEX</code>	<code>Long double_Complex</code>
<code>MPI_C_BOOL</code>	<code>_Bool</code>
<code>MPI_C_LONG_DOUBLE_COMPLEX</code>	<code>Long double_Complex</code>
<code>MPI_INT8_T</code> <code>MPI_INT16_T</code> <code>MPI_INT32_T</code> <code>MPI_INT64_T</code>	<code>Int_8_t</code> <code>Int_16_t</code> <code>Int_32_t</code> <code>Int_64_t</code>
<code>MPI_UINT8_T</code> <code>MPI_UINT16_T</code> <code>MPI_UINT32_T</code> <code>MPI_UINT64_T</code>	<code>Uint_8_t</code> <code>Uint_16_t</code> <code>Uint_32_t</code> <code>Uint_64_t</code>
<code>MPI_BYTE</code>	<code>8 binary digits</code>
<code>MPI_PACKED</code>	<code>Data packed or unpacked with MPI_PACK() / MPI_UNPACK</code>

- Las opciones *dest* y *source* son, respectivamente, los identificadores del proceso al que se envía el mensaje y del proceso desde el cuál se recibe el mensaje. La constante *MPI\_ANY\_SOURCE* se puede usar para indicar la posibilidad de recibir el mensaje de cualquier proceso
- La opción *tag* se usa para diferenciar entre mensajes, y su valor debe coincidir en los procesos de envío y recibimiento. La constante *MPI\_ANY\_TAG* se puede usar para indicar que el mensaje es compatible con mensajes con cualquier identificador
- La opción *comm* es el comunicador dentro del cual se realiza la comunicación. Es de tipo *MPI\_Comm*, y una constante para indicar que se usa el comunicador que contiene todos los procesos es *MPI\_COMM\_WORLD*
- La opción *status* se refiere a la variable del tipo *MPI\_Status*. No se usa en el programa, pero contiene información del mensaje

recibido y puede ser consultado para identificar cualquier característica del mensaje (su longitud, su proceso fuente, etc.)

- La comunicación que se ha visto usando *MPI\_Send* y *MPI\_Recv* es del tipo bloqueo o *blocking type*, dado que se lleva a cabo una sincronización entre el proceso que envía el mensaje y el que lo recibe
  - Con *MPI\_Send* y con *MPI\_Recv*, el proceso que envía el mensaje continúa ejecutándose una vez que la transferencia ha acabado
  - Si el proceso que recibe el mensaje pide la recepción de este antes de que llegue, se bloquea el
- MPI proporciona otras posibilidades más que las transferencias del tipo bloqueo, por ejemplo, *MPI\_SSend*, *MPI\_Bsend* y *MPI\_Rsend*. Estas funciones difieren entre ellas en la manera en que gestionan la transferencia y de cómo se puede gestionar el *buffer* de comunicación o cómo se especifica que el proceso rompa datos directamente de la memoria



- Un *buffer* de comunicación es un dispositivo de almacenamiento que compensa la diferencia en la tasa de flujo de los datos o el tiempo de ocurrencia de eventos cuando se transmiten datos de un dispositivo a otro
- La función *MPI\_Sendrecv* combina en una misma llamada la transferencia y la recepción entre dos procesos, pero continúa siendo una función de bloqueo. El envío y la recepción ocurren a la vez, de modo que ambos procesos pueden recibir y mandar datos sin bloqueos
- MPI también proporciona comunicación *non-blocking* en las que el proceso receptor pide el mensaje y continúa ejecutándose aún si el mensaje no ha llegado. Las funciones que permiten esto son las funciones *MPI\_Isend* y *MPI\_Irecv*

```

int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status)

```

- Si el proceso receptor no ha recibido aún el mensaje, puede que haya un punto en el que tenga que esperar para que continue su ejecución. También se tiene la función *MPI\_Wait()* para esperar a la llegada de un mensaje y *MPI\_Test()* para verificar que la operación se ha completado

```

MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    int x;
    MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
    compute(); // Makes a calculation while sending
    MPI_Wait(req1, status);
}
else if (myrank == 1) {
    int x;
    MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}

```

- Además de los comunicadores punto a punto, MPI ofrece una serie de funciones para llevar a cabo comunicación en la que todos los procesos del comunicador intervienen
  - Cuando sea posible, es mejor realizar comunicaciones usando estas comunicaciones colectivas, dado que hacer esto facilita la programación y evita posibles errores
    - Si las funciones se optimizan para el sistema con el que se está trabajando, se pueden desarrollar programas más eficientes usando comunicaciones colectivas y no comunicaciones de punto a punto
  - La comunicación colectiva de barrera o *barrier* establece una barrera en donde todos los procesos esperan llegar a esta barrera para continuar realizando la operación una vez que hayan llegado (se espera a que todos hayan llegado)

```

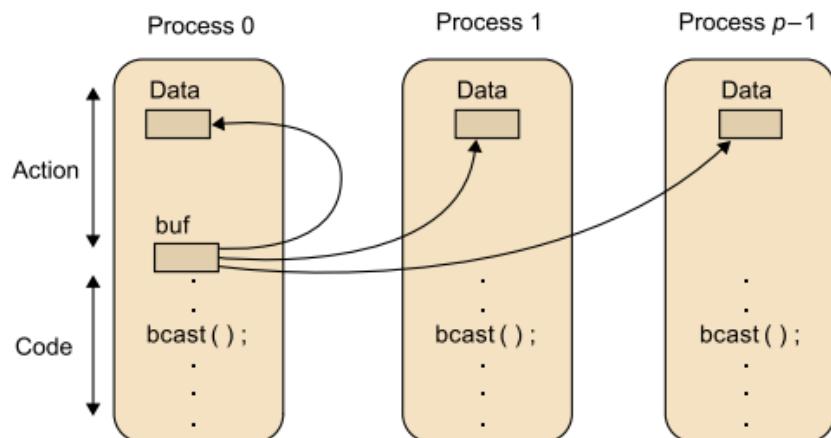
int MPI_Barrier(MPI_Comm comm)

```

- Se usa un comunicador para establecer un grupo de procesos que se sincronizan, de modo que se bloquean todos los procesos hasta que todos llamen a esta rutina, forzando una sincronización en esta parte del código
- Todas las funciones de comunicaciones devuelven un código de error
- La comunicación colectiva de retransmisión o *broadcast* realiza una operación de retransmisión (comunicación de un proceso a todos los otros) en la que datos de un tipo se envían del proceso raíz o *root* al resto de procesos en el comunicador

```
int MPI_Bcast(void *buffer, int count,
              MPI_Datatype datatype, int root, MPI_Comm comm)
```

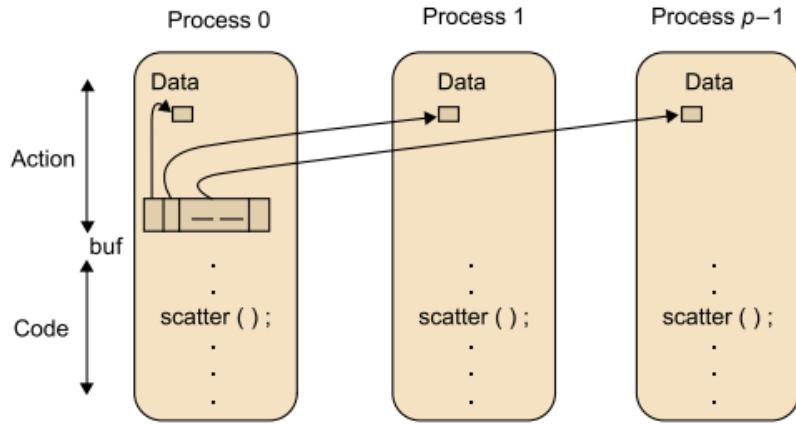
- Todos los procesos que intervienen deben llamar a la función indicando el proceso que sirve como raíz. En la raíz, los datos que se están enviando se toman de la zona indicada por el *buffer*, y aquellos que se reciben se guardan en la memoria reservada en el *buffer*



- Con esta función, se envía una misma pieza de datos (la cual va al *buffer* y se toma de los datos de un proceso raíz) a todos los procesos
- La comunicación de dispersión o *scatter* se usa para enviar diferentes mensajes de un proceso al resto de procesos

```
int MPI_Scatter (void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, void *recvbuf,
                  int recvcount, MPI_Datatype recvtype,
                  int root, MPI_Comm comm)
```

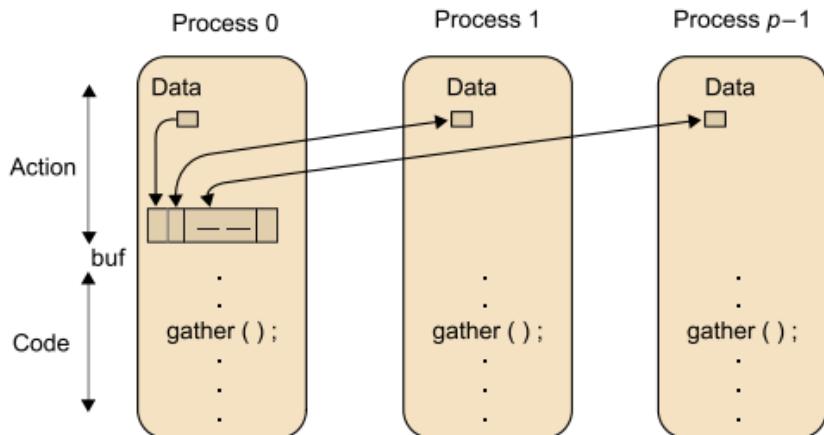
- En el proceso raíz, el mensaje se divide en segmentos de tamaño igual a *sendcount* y el enésimo segmento envía el proceso *n*. Si uno quiere enviar bloques de diferentes tamaños a los procesos diferentes, o si los bloques a enviar no se siguen en la memoria, se puede usar la función *MPI\_Scatterv*



- A diferencia de con la función *MPI\_Bcast*, esta función divide una pieza de datos en pequeños segmentos y los envía a los diferentes procesos
  - La comunicación de recolección o *gather* se usa para enviar diferentes mensajes del resto de procesos a un proceso

```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

- Todos los procesos (incluyendo la raíz) envían datos al proceso raíz desde *sendbuf*, y el proceso raíz los almacena en *recvbuf* en el orden de los procesos. Si uno quiere que cada proceso envíe bloques de diferentes tamaños, se usa *MPI\_Gatherv*



- De este modo, ahora se recolectan los diferentes segmentos (que pueden ser del mismo o diferente tamaño) de los diferentes procesos al proceso raíz (obteniendo así el bloque original)
- Para enviar bloques de datos de todos los procesos a todos los procesos, se utiliza *MPI\_Allgather*. En este caso, el bloque enviado por el enésimo proceso se guarda en todos los procesos como el enésimo bloque en *recvbuf* (para enviar bloques de diferentes tamaños, se utiliza *MPI\_Allgatherv*)

```
int MPI_Allgather (void *sendbuf, int sendcount,
                   MPI_Datatype sendtype, void *recvbuf,
                   int recvcount, MPI_Datatype recvtype,
                   MPI_Comm comm)
```

- Para enviar bloques de diferentes datos a diferentes procesos, se utiliza la función *MPI\_Alltoall*, donde cada proceso *i* envía un bloque *j* a un proceso *j*, que lo almacena en *recvbuf* como un bloque *i* (para enviar bloques de diferentes tamaños, se utiliza *MPI\_Alltoallv*)

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm)
```

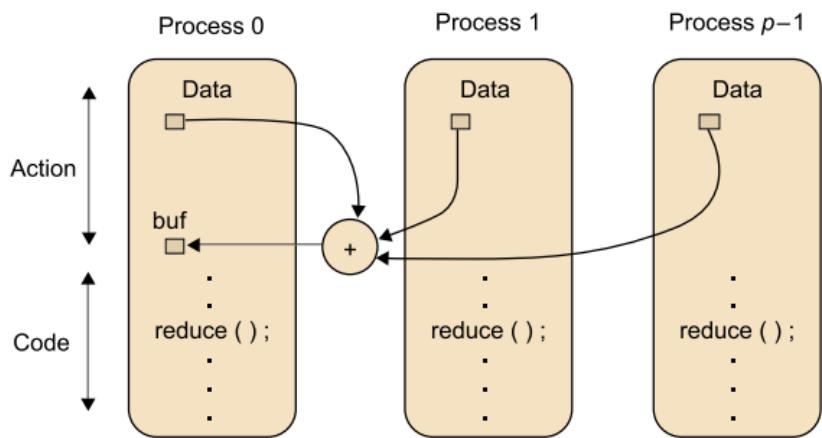
- La comunicación de reducción realiza una reducción de todo a uno, de modo que los valores distribuidos se recogen en los diferentes procesos y una sola operación se aplica a todos ellos. Esto significa que los datos que necesitan una “reducción” se toman de la zona indicada por *sendbuf*, y los resultados se guardan en el *recvbuf* indicado

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm)
```

- Si el número de elementos de datos (*count*) es más grande que 1, entonces la operación se realiza en los *buffers* separadamente y elemento por elemento. El resultado se guarda en el proceso *root* y la operación aplicada a los datos viene dada por el parámetro *op*, que puede tener los siguientes valores:

Operation	Meaning	Permitted C types
MPI_MAX MPI_MIN MPI_SUM MPI_PROD	maximum minimum sum product	Integers and floating point Integers and floating point Integers and floating point Integers and floating point
MPI_LAND MPI_LORD MPI_LXORD	AND logic OR logic XORD logic	Integers Integers Integers
MPI_BAND MPI_BOR MPI_BXOR	AND bit-wise OR bit-wise XOR bit-wise	Integers and bytes Integers and bytes Integers and bytes
MPI_MAXLOC MPI_MINLOC	Maximum and location Minimum and location	Type Pairs Type Pairs

- El siguiente esquema muestra cómo funciona la función *MPI\_Reduce*:



- En el ejemplo del código inferior, se puede ver como se usa la operación para sumar valores de un intervalo basado en las sumas parciales de cada uno de los procesos MPI

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
```

```

int MyProc, tag=1, size;
char msg='A', msg_recept ;
MPI_Status *status ;
int root ;
int left, right, interval ;
int number, start, end, sum, GrandTotal;
int mystart, myend;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);
MPI_Comm_size(MPI_COMM_WORLD, &size);
root = 0;
if (MyProc == root) /* The root process reads the limits of the interval */
{
    printf("Give the left and right limits of the interval\n");
    scanf("%d %d", &left, &right);
    printf("Proc root reporting : the limits are : %d %d\n", left, right);
}
MPI_Bcast(&left, 1, MPI_INT, root, MPI_COMM_WORLD); /*Bcast limits to all */
MPI_Bcast(&right, 1, MPI_INT, root, MPI_COMM_WORLD);
if (((right - left + 1) % size) != 0)
    interval = (right - left + 1) / size + 1; /*Sets local limits to sum*/
else
    interval = (right - left + 1) / size;
mystart = left + MyProc*interval ;
myend = mystart + interval ;
/* establishes interval limits correctly */
if (myend > right) myend = right + 1;
sum = root; /* Adds locally in each MPI process */
if (mystart <= right)
    for (number = mystart; number < myend; number++) sum = sum + number ;
/* Performs the reduction in the root process */
MPI_Reduce(&sum, &GrandTotal, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD) ;
MPI_Barrier(MPI_COMM_WORLD);
/* The root process returns the results */
if(MyProc == root)
    printf("Proc root reporting : Grand total = %d \n", GrandTotal);
MPI_Finalize();
}

```

- Cuando todos los procesos necesitan recibir el resultado de la operación, se puede utilizar la función *MPI\_Allreduce*

```

int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

```

- Ahora que se han visto diferentes funciones de MPI, se puede estudiar como se compila y se ejecuta un código MPI
  - La manera normal de compilar un código MPI es a través de *mpicc*, que es un comando que llama al compilador de C establecida y lo monta con una librería MPI. El código a compilar se indica y normalmente se pasan diversas opciones

```
mpicc program.c -o program
```

- Una vez el código se ha generado, el *mod* de ejecución también depende de la compilación. Normalmente se usa *mpirun*, pasándole el código para la ejecución y una serie de argumentos que indican los procesos a comenzar y la distribución de los procesos entre los diferentes procesadores

```
mpirun -np 4 program
```

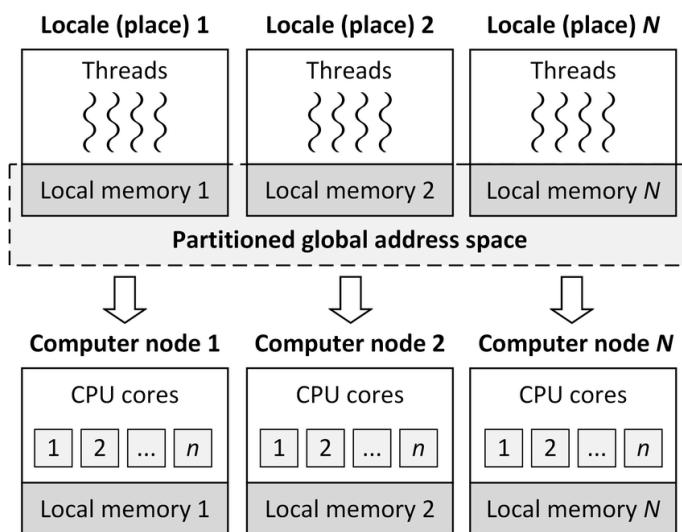
- En el ejemplo anterior, se comienzan 4 procesos (opción *-np 4*) que ejecutan el mismo código. Los procesadores a los que se asignan los procesos no se especifican, de modo que se utiliza la manera predefinida de asignar los procesos en los procesadores por el sistema: puede ser que todos los procesos se asignen a un mismo nodo (que puede tener varios procesadores o núcleos o solo uno) o que se asignen a los diferentes nodos disponibles en el sistema
- Hay varias maneras de indicar cómo se deben asignar los procesos. Una manera es a través de *-machinefile filename.txt*, que es una opción e indica los procesos a través de un archivo ".txt" que indica los servidores que se pueden usar para ejecutar los procesos MPI. Este texto puede ser una lista con el nombre de los servidores

```
mpirun -np 4 -machinefile nodes.txt program
```

```
node1  
node2
```

- En el ejemplo, si se está ejecutando en el nodo 0, los cuatro procesos se asignan en el siguiente orden: el proceso 0 se va a al nodo en que el programa se está ejecutando (el nodo 0), el proceso 1 se va al nodo 1, el 2 al nodo 2, y cuando la lista pasada está completa, comienza otra vez desde el principio, de modo que el proceso 3 va al nodo 1, y así
- El modo de asignar depende de la implementación y puede ser que el nodo en el que uno está no se le asigne ninguno de los procesos o de que entre en una asignación cíclica una vez que el archivo de máquinas se complete. Por ejemplo, con la opción *-nolocal* se puede evitar usar el servidor raíz (esto es útil cuando se está usando un nodo *login* que no se debería usar para computación, como con un clúster)

- La programación de sistemas distribuidos normalmente se ha igualado al paradigma de pasar mensajes, pero se pueden implementar abstracciones de mayor nivel por debajo y por encima de un sistema de memoria distribuida. Durante la última década, varios grupos de investigación han desarrollado estas abstracciones, resultando en una familia de lenguajes conocidos como *partitioned global address space* (PGAS)
- El *global address space* se refiere al hecho de que el lenguaje proporciona una única región de memoria por encima de las memorias virtuales de las máquinas distribuidas. Este espacio de memoria se compone de varios bloques de memoria que son compartidos entre diferentes nodos



- Obviamente, los lenguajes PGAS no proporcionan memoria compartida, dado que no se puede esperar que el *hardware* se encargue de la coherencia. Aun así, el *global address space* permite definir estructuras de datos globales, lo cual mejora el modelo de memoria distribuida en la que el programador llama a una colección de estructuras de datos independientes que actúan como una estructura de datos distribuidas difícil de mantener
- Con los lenguajes PGAS, los programadores se pueden enfocar en el comportamiento de un solo proceso y distinguir los datos locales de los datos no locales o remotos. No obstante, estos lenguajes permiten simplificar la programación porque eliminan los detalles de *message-passing* y mejoran la productividad
- Los compiladores generan llamadas a la comunicación cuando hay referencias a las direcciones de memoria que no son locales
- Las implementaciones más importantes de los lenguajes PGAS son UPC (extiende C), *co-array* Fortran (extiende Fortran) y Titanium (extiende Java)

- El *unified parallel C* (UPC) se desarrolló cerca del 2000, y este lenguaje proporciona al programador con una visión general de la región de memoria de modo que cuando los elementos se definen (por ejemplo, matrices) como compartidas, se distribuyen entre las memorias de varios procesos, de manera que cuando se distribuyen los elementos compartidos en un estilo cíclico o con bloques cíclicos
  - Este tipo de distribución permite que la carga esté mejor distribuida, pero en general, también significa que la localidad de los datos se reduce. Sin embargo, se puede mejorar la localidad de los datos asignando fragmentos de datos compartidos directamente entre procesadores, de modo que muchos elementos de datos contiguos siguen en el mismo proceso
  - El hecho de que UPC extiende el lenguaje C permite al usuario usar *pointers*, los cuales pueden ser privados (locales al proceso) o compartidos entre procesos. Dado que los *pointers* pueden ser privados o compartidos y que, además, pueden apuntar a elementos de datos privados o compartidos, hay cuatro tipos de *pointers*

#### Pointer property

Reference property	<i>Private</i>	<i>Shared</i>
	<i>Private</i>	<i>Private-private</i> , p1
<i>Shared</i>	<i>Shared-private</i> , p3	<i>Shared-shared</i> , p4

```

int *p1; /* private pointer pointing to local data */
shared int *p2; /* private pointer pointing to shared space */
int *shared p3; /* shared pointer pointing to local data */;
shared int *shared p4; /* shared pointer pointing to shared space */

```

- El elemento *upc\_forall* es una abstracción que distribuye las iteraciones en bucles de C normales (bucles *for*) entre los procesos usando una cláusula de afinidad (que es la cuarta cláusula de *upc\_forall*). La cláusula de afinidad del bucle indica donde se deberían ejecutar las interacciones dentro del bucle

```

shared int v1[N], v2[N], v1v2sum[N];

void main()
{
    int i;
    shared int *p1, *p2;
    p1=v1;
    p2=v2;
    upc_forall(i=0; i<N; i++; p1++; p2++; i)
    {
        v1v2sum[i]=*p1-*p2;
    }
}

```

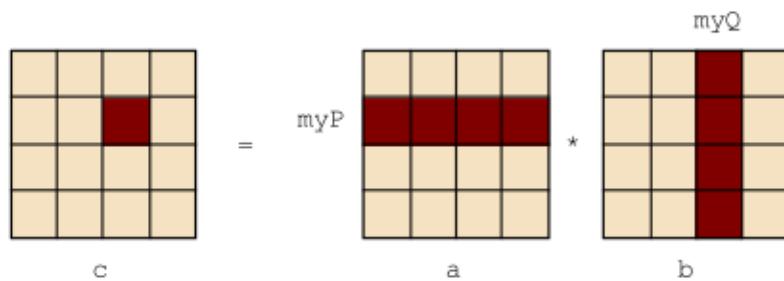
- En el ejemplo anterior, el proceso que se ejecuta en la iteración  $n$  es aquel asociado con una  $n$ . El comando *upc\_forall* es la operación global en que la mayoría del código se ejecuta en un proceso localmente

- Co-Array Fortran

```

real, dimension(n,n) [p,*]::a,b,c
...
do k=1,n
    do q=1,p
        c(i,j) [myP,myQ]=c(i,j) [myP,myQ]+a(i,k) [myP,q]*b(k,j) [q,myQ]
    enddo
enddo

```



- Titanium

```

public static void matMul(    double [2d] a,
                             double [2d] b,
                             double [2d] c)
{
    foreach (ij in c.domain())
    {
        double [1d] aRowi=a.slice(1, ij[1]);
        double [1d] bColj=b.slice(2, ij[2]);
        foreach (k in aRowi.domain())
        {
            c[ij]+=aRowi[k]*bColj[k];
        }
    }
}

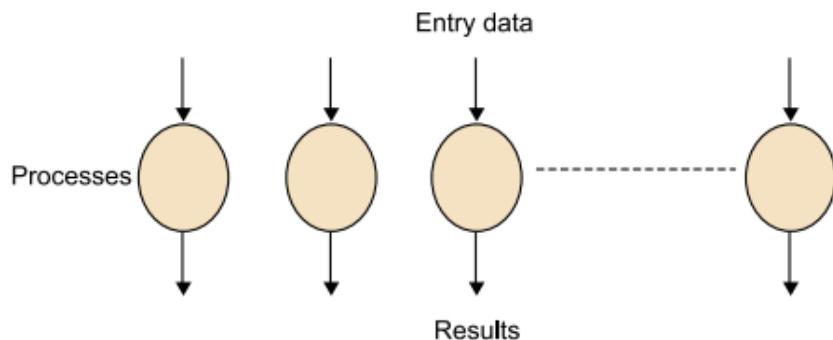
int single stepCount=0;
int single endCount=100;
for (; stepCount<endCount; stepCount++)
{
    Read remote particles
    Compute forces that are mine
    Ti.barrier();
    It writes my particles using new forces
    Ti.barrier();
}

```

## La programación en paralelo: esquemas paralelos algorítmicos

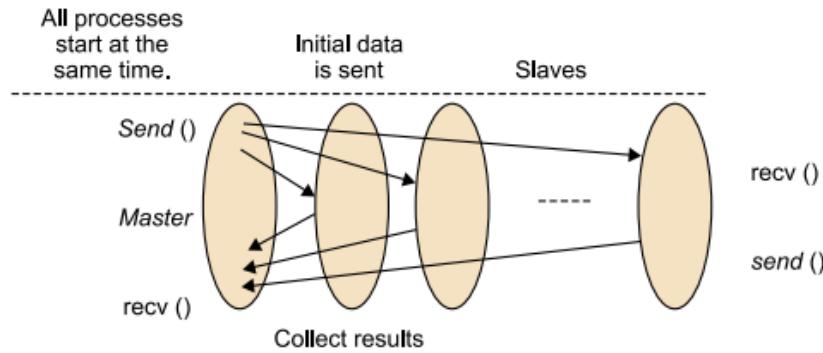
- La técnica de paralelismo de datos se puede usar cuando se trabaja con una gran cantidad de datos de manera equivalente o similar. Típicamente, requiere algoritmos numéricos en los que los datos se encuentran en vectores y se trabajan usando el mismo proceso
  - Esta técnica es apropiada para memoria compartida, y normalmente el paralelismo se obtiene dividiendo el trabajo de ciclos o *loops* entre diferentes *threads* (por ejemplo, en OpenMP se haría con *#pragma omp for*)
    - Los algoritmos en los que aparecen *loops* en donde no hay dependencias de datos, o en las que las dependencias se puedan evitar, son adecuados para el paralelismo de datos

- Si los datos se distribuyen entre procesos trabajando en memoria distribuida, se tendría que usar la técnica de partición de datos y no de paralelismo de datos
- Este tipo de paralelismo se encuentra en algoritmos de multiplicación de matrices en OpenMP, de modo que, si la multiplicación se paraleliza, se tiene que indicar simplemente con `#pragma omp parallel for` en el bucle más externo, de modo que cada *thread* calcule una serie de filas de la matriz resultante
  - La aproximación de un número entero también utiliza este paradigma
- La suma secuencial de  $n$  números se realiza en un solo bucle *for*, el cual se puede paralelizar usando `#pragma omp parallel for` con la cláusula de *reduction* para indicar la manera en que los *threads* acceden la variable en la que la suma se guarda
  - La paralelización se realiza de una manera simple y, por tanto, se habla de paralelismo implícito, dado que el programador no es responsable por la distribución del trabajo o del acceso a las variables compartidas
- En algunos casos, igual se quiere implementar un *job* para cada uno de los *threads* (por ejemplo, para reducir el coste de gestión de los *threads*), y hacer al programador responsable de la distribución y acceso a los datos, de modo que la programación se complica. En este caso, se habla de paralelismo explícito
  - Este tipo de problema, compuesto de un conjunto de tareas independientes, también se conoce como *embarrassingly parallel*. En este tipo de aplicaciones hay poca o nula comunicación entre procesos y cada proceso realiza una tarea sin necesidad de interacción con otros procesos

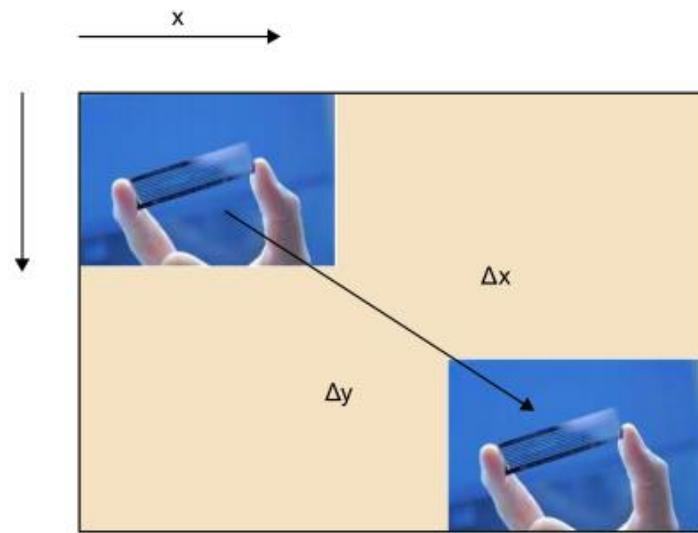


- Si se piensa en una interacción estática con MPI de este modelo basado en el patrón de maestro-esclavo, se puede ver como el

proceso maestro crea un conjunto de procesos esclavos que se sincronizan usando *message-passing* al comienzo, completan su tarea independiente y, finalmente, se sincronizan otra vez al final, cuando el proceso maestro recoge los resultados de los otros procesos



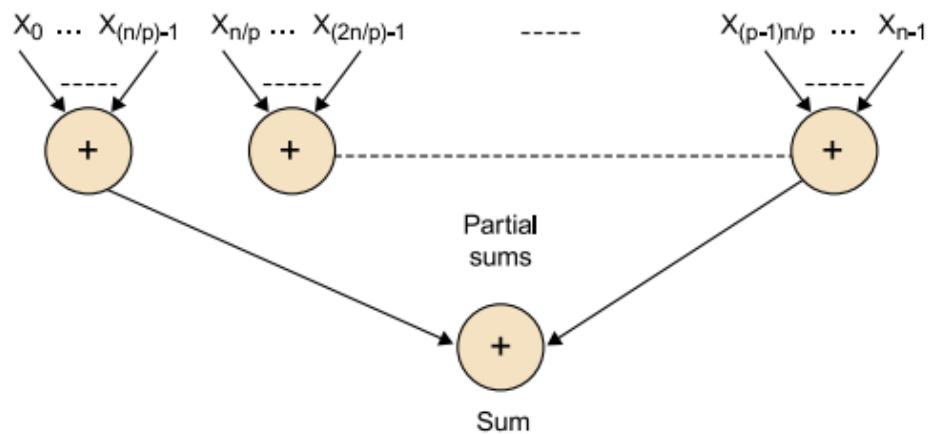
- Aplicaciones como procesar una imagen son típicas de este esquema de paralelismo. Un ejemplo claro de esto es desplazar una imagen en la que cada píxel sufre un incremento determinado



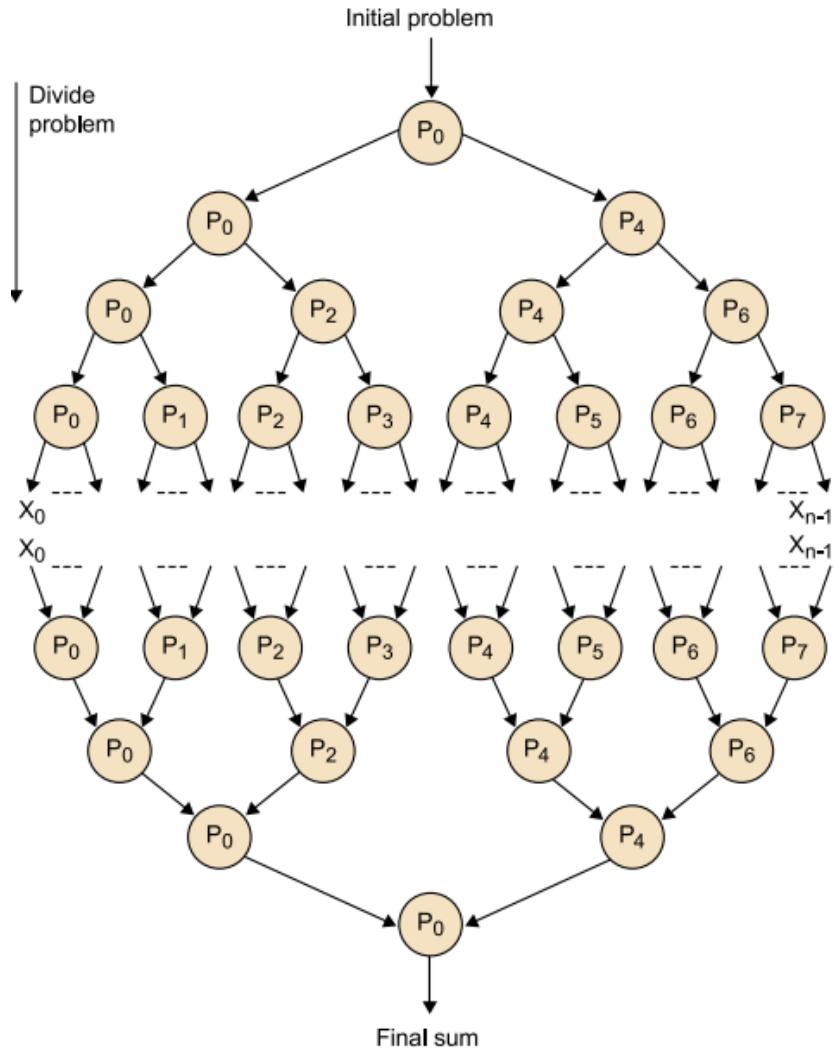
- La técnica de partición de datos sirve para una aplicación que trabaja con grandes volúmenes de datos, normalmente en vectores y matrices, y se obtiene paralelismo al dividir el trabajo en diferentes zonas de datos entre distintos procesos
  - A diferencia del paralelismo de datos visto anteriormente, en este caso, los datos se distribuyen entre procesos y esta distribución determina la distribución de trabajo
    - Lo más usual es dividir el espacio de datos en regiones en donde hay un intercambio de datos entre regiones adyacentes del

algoritmo, de modo que las regiones adyacentes se deben asignar a procesadores, queriendo prevenir que las comunicaciones se vuelvan muy costosas

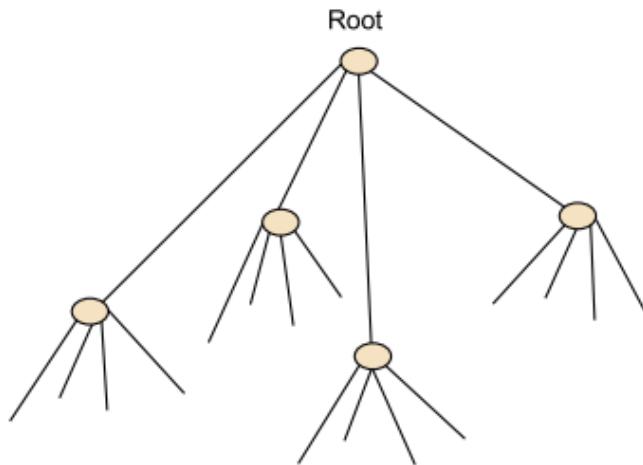
- Algunas comunicaciones comparten datos y están distribuidas en la memoria del sistema, de modo que el coste de comunicación resultante suele ser alto. En este sentido, para obtener un mejor rendimiento es necesario tener un volumen más alto de computación que de comunicación
- Un tipo específico de partición de datos es la técnica de “divide y vencerás”. Este tipo de problema se caracteriza por el hecho de que el problema se divide en subproblemas similares al problema principal, y se suele dividir en aún más subproblemas de forma recursiva



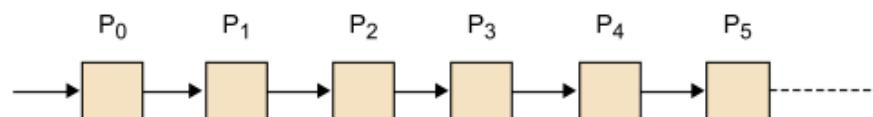
- El esquema anterior muestra un ejemplo de suma de una secuencia de números usando la técnica de “divide y vencerás”, en donde cada proceso realiza una suma parcial que después se sumará al final para obtener la suma final
- Los esquemas de árboles paralelos nacen de que muchos problemas se pueden representar en la forma de un árbol o de un grafo, y resolver estos consiste en ir a lo largo del árbol haciendo computaciones
  - Una técnica para desarrollar programas paralelos es, dado un problema, obtener un grafo de dependencias que represente la computación requerida (los nodos de un grado) y las dependencias de datos (los arcos del grafo)



- A partir del gráfico, se deciden las asignaciones de procesos y comunicaciones de tareas a partir de los arcos que comunican los nodos asignados a diferentes procesos
- Un ejemplo de esto es el esquema anterior de una suma del prefijo, en donde la primera fase divide el problema en una mitad, y sigue generando hijos hasta que, en la segunda parte, los resultados se recogen mientras que se ejecuta el árbol
  - Los nodos de los árboles pueden tener más de dos ramificaciones o hijos



- El esquema o patrón de *pipeline* se usa para resolver un problema partiéndolo en una serie de tareas sucesivas, de modo que los datos fluyen en una dirección a través de la estructura de procesos
  - Uno puede ver un tipo de partición funcional en la que el problema se divide en diferentes funciones que se tienen que llevar a cabo sucesivamente



- Este esquema tiene una estructura de *message-passing* lógica, de modo que los datos deben de estar comunicados entre las tareas consecutivas
- Puede ser útil cuando no hay un solo conjunto de datos a procesar, sino una serie de conjuntos de datos que se tienen que calcular uno después de otro, o en problemas en que hay un paralelismo funcional, con varios tipos de operaciones en el conjunto de datos y que se deben de realizar uno después de otro
- Por ejemplo, se puede suponer que se quieren sumar elementos de un vector usando un algoritmo con un bucle, y se puede implementar este bucle de manera secuencial:

```

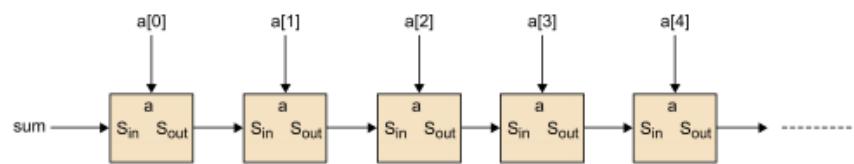
for (i = 0; i < n; i++)
    sum = sum + a[i];
  
```

```

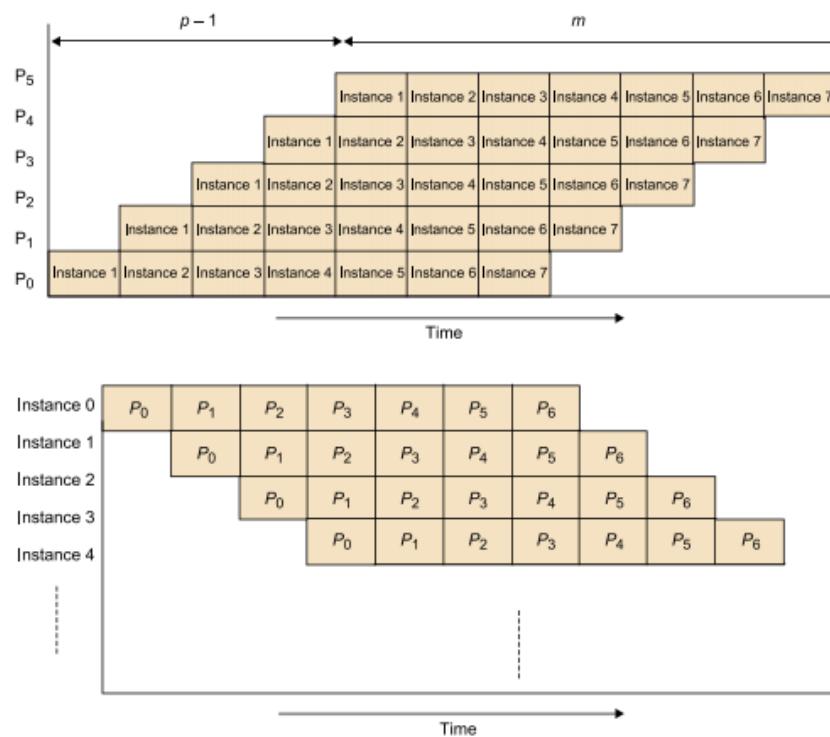
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
...

```

- De esta manera, se acaba con un *pipeline* como en el siguiente esquema:



- Si se supone que, en general, el problema se puede dividir en una serie de tareas secuenciales, el esquema del *pipeline* proporciona un tiempo de ejecución menor en los siguientes tipos de computación:
  - Siempre que sea posible ejecutar más de una instancia del problema entero al mismo tiempo y se pueda ejecutar como en los diagramas:

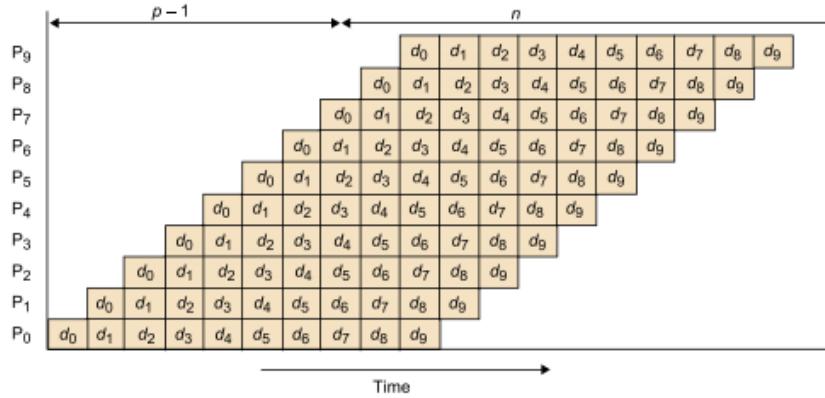


- Cuando una serie de elementos de datos se tiene que procesar y cada uno requiere múltiples operaciones, como en el siguiente diagrama:

(a) Pipeline structure

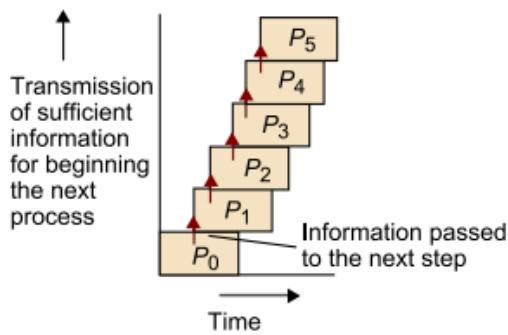


(b) Time diagram

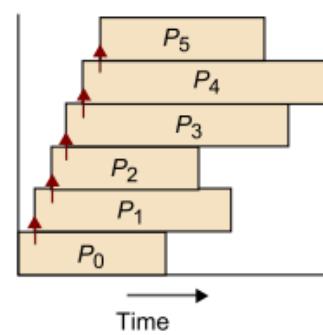


- Cuando la información necesaria para comenzar la ejecución del siguiente proceso puede pasarse al proceso siguiente antes de que el proceso actual complete sus operaciones. Un ejemplo de este tipo es la búsqueda de cierta información en una base de datos

(a) Processes with the same execution time



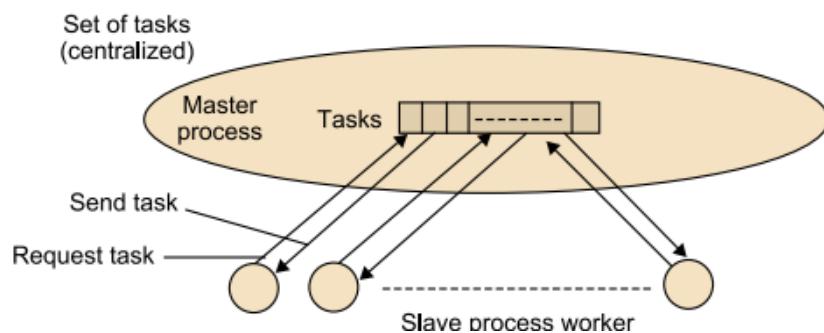
(b) Processes with different execution times



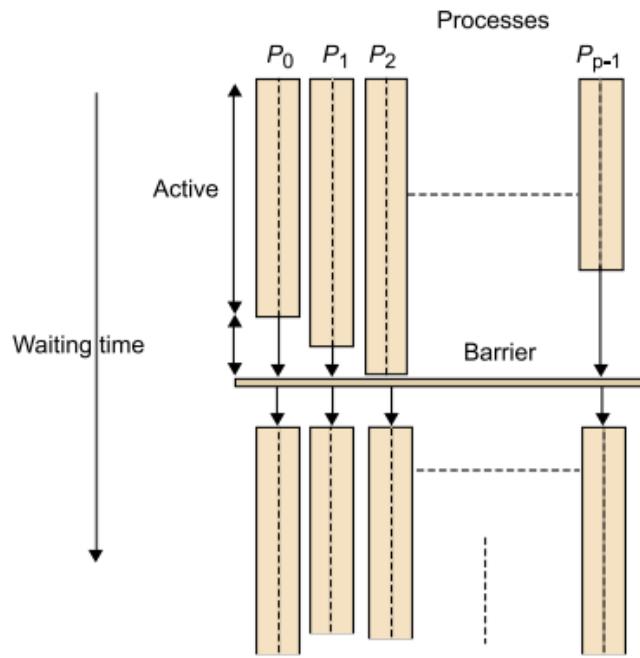
- En el paradigma de maestro-esclavo, se puede tener un flujo o proceso diferenciado (llamado maestro) que pone en movimiento a los otros procesos (llamados esclavos), les asigna una tarea y recoge las soluciones parciales generadas
  - En algunos casos se habla de una granja de procesos, que se refiere a cuando un grupo de procesos trabaja junto pero independientemente en la resolución de un problema

- Este paradigma tiene similitudes con el esquema de maestro-esclavo si se considera que los procesos constituyentes de la granja son los esclavos (o los esclavos y el maestro si es que el maestro también interviene en la computación)
- Otro paradigma relacionado a estos dos es aquel de trabajadores replicados. En este caso, los trabajadores (los *threads* o procesos) actúan de manera autónoma y resolver las tareas que pueden dar paso a nuevas tareas que pueden resolverse por el mismo trabajador o por otro diferente
  - Se gestiona una bolsa de tareas: cada trabajador toma un trabajo perteneciente a esta bolsa y, una vez el trabajo de la tarea asignada se completa, toma una nueva tarea de la bolsa para trabajar con ella, y así hasta que ya todas estén resueltas
  - Se considera el problema de la terminación, porque puede no haber ninguna tarea en la bolsa cuando el trabajador pida una. No obstante, esto no significa que no haya más tareas a resolver, dado que puede haber otro proceso que esté computando genere nuevas tareas
  - La condición de terminación será que no haya tareas restantes en la bolsa y que no haya ningún trabajador trabajando. Hay diferentes tipos de maneras de enfocar el problema de la terminación
  - Esta es la manera típica en la que OpenMP trabaja. Inicialmente un solo *thread* trabaja y, cuando llega a un constructor paralelo, pone un conjunto de *threads* esclavos en movimiento y se convierte en maestro
- En programación *message-passing*, el patrón también se usa rutinariamente: en MPI, cuando los procesos se inicializan con una orden como *mpirun -np n program*, *n* procesos iguales se inicializan
  - No obstante, la entrada y la salida normalmente se realizan usando un proceso que actúa como un maestro, generando un problema y distribuyendo los datos. Después de que este proceso de computación comience, en la que el maestro puede intervenir o no, el maestro recibe los resultados de los esclavos
- La asignación de las tareas de los esclavos se puede realizar de manera estática o dinámica:
  - En la asignación estática, el maestro decide qué tareas se asignan a los esclavos y los envía

- En la asignación dinámica, el maestro genera trabajos y los guarda en una bolsa de tareas que este mismo gestiona, y los esclavos piden tareas de la bolsa en cuanto están libres para realizar nuevas tareas. De este modo, la carga se balancea de manera dinámica, pero podría ocurrir una sobrecarga al gestionar la bolsa, lo cual puede crear un cuello de botella
- Además, en algunos problemas, resolver una tarea genera nuevas tareas que deben incluirse en la bolsa, haciendo que haya más comunicaciones con el proceso maestro



- Se habla de un esquema de paralelismo sincronizado cuando un problema se resuelve por iteraciones sucesivas en la que todos los procesos se sincronizan
  - Las características generales de este tipo de esquema son las siguientes:
    - Cada proceso lleva a cabo la misma tarea en una porción diferente de los datos
    - Parte de los datos de una iteración se usan en la siguiente, de modo que al final de cada iteración hay una sincronización que puede ser local o global
    - La tarea se completa cuando cualquier criterio de convergencia se cumple, para el cual se requiere una sincronización global
  - Muchos problemas se pueden resolver utilizando este tipo de patrón. El mecanismo principal es el uso de las barreras donde los procesos se deben sincronizar



- Estos procesos pueden continuar ejecutándose una vez todos hayan llegado a este punto
- También se puede utilizar el lenguaje de programación Python para construir sistemas de alto rendimiento
  - Python es un lenguaje de programación de alto nivel y de propósito general que es muy popular que soporta muchos paradigmas de programación, incluyendo programación orientada a objetos, programación imperativa y funcional, y programación procedimental
    - Tiene un sistema de tipo dinámico y gestión automática de memoria, además de una librería estándar amplia. Existen varios interpretadores de Python para varios sistemas imperativos
    - Este lenguaje se está volviendo popular para entornos de alto rendimiento porque permite trabajar más rápido e integrar sistemas de manera más efectiva. Python es fácil de aprender y se consiguen aumentos de productividad por esa razón, pero las soluciones basadas en modelos de computación de alto rendimiento como MPI proporcionan un rendimiento superior
    - Python tiene extensiones como Cython, que permite el uso de funciones propias de C. Esta extensión es un compilador estático que hace que escribir extensiones de C para Python sea sencillo como Python, haciendo posible a los compiladores generar código de C muy eficiente basado en código de Cython

## La computación verde

- La computación verde es un término que normalmente se refiere al uso eficiente de recursos computacionales, pero es un término amplio. No obstante, la motivación principal está en el uso de los recursos computacionales para minimizar el impacto medioambiental, maximizar la viabilidad económica y sostenibilidad y cumplir con las obligaciones sociales
  - Aunque hay mucho interés en soluciones que permitan el uso de fuentes de energía más verdes y sostenibles, el desarrollo se centra en la eficiencia energética
    - Se puede hablar de eficiencia en varios contextos, desde la producción, a la distribución o del consumo
  - Los conceptos de energía y potencia eléctrica son de vital importancia para las métricas
    - En física, la energía eléctrica se define como una forma de energía que resulta de la existencia de una diferencia en potencial entre dos puntos, que permite el establecimiento de una corriente eléctrica entre los puntos cuando contactan a través de un conductor eléctrico. La energía eléctrica se puede transformar en otros tipos de energía, tal y como la térmica en el contexto de circuitos electrónicos
    - La potencia eléctrica es la relación del pasaje de un flujo de energía por unidad de tiempo, por lo que es la cantidad de energía que liberada o absorbida por un elemento en un momento particular. En consecuencia, la energía es una función de la integración de la potencia a lo largo del tiempo, por lo que reducir la potencia no necesariamente reduce la energía
    - En el sistema internacional, las unidades de medida de estos conceptos son el Joule (J) para la energía y el Watt (W) para la potencia
    - Se puede definir la eficiencia energética muy generalmente como la obtención de un resultado que minimice el consumo de energía o, complementariamente, todas las acciones que tienden a reducir su consumo
  - Las métricas son esenciales para medir cuantitativamente y evaluar el consumo energético, siendo las bases de la toma de decisiones. Hay dos tipos de métricas: métricas para computadoras individuales y para sistemas paralelos

- La métrica más básica de eficiencia energética proviene de la familia del diseño de circuitos y de la fórmula  $ED^n$ , donde  $E$  es la energía consumida durante la ejecución de la aplicación,  $D$  es el tiempo necesario para completarla y  $n$  es un parámetro no negativo que caracteriza el balance entre  $E$  y  $D$ . De este modo, se combina energía y tiempo
  - La métrica  $ED2P$ , que es  $ED^n$  cuando  $n = 2$ , es la más usada, sobre todo cuando se utilizan técnicas de voltaje dinámico y escalamiento de frecuencia. Con esta métrica, la influencia de la escala de la frecuencia se puede cancelar porque  $E$  es proporcional al cuadrado de la frecuencia, mientras que  $D^2$  es inversamente proporcional al cuadrado de la frecuencia
  - $ED2P$  considera la eficiencia y el consumo de energía, pero no tiene en cuenta las necesidades de diferentes sistemas
  - Para generalizar la métrica  $ED2P$ , esta se puede formular ponderándose por  $\delta$  con  $|\delta| \leq 1$  y determinado por la preferencia de los usuarios. Esta métrica intenta favorecer el rendimiento cuando  $\delta > 0$  o la energía si  $\delta < 0$

$$\text{Weighted } ED2P = E^{(1-\delta)} \times D^{2(1+\delta)}$$

- Cuando el valor de  $n$  en  $ED^n$  es muy grande, se produce un sesgo a favor de sistemas paralelos masivos. La variante opuesta de  $ED^n$ ,  $1/ED^n$ , representa  $\text{Efficiency}^n/\text{Power}$  o  $\text{Flops}^n/W$ 
    - Cuando una supercomputadora tiene  $s$  procesadores y cada uno de estos proporciona  $F$  flops de  $P$  watts, la métrica  $\text{Flops}^n/W$  se puede reformular de la siguiente manera:
- $$\frac{\text{Flops}^n}{W} = \frac{(sF)^n}{sP} = s^{n-1} \frac{F^n}{P}$$
- En la lista Green500, la métrica se usa con  $n = 1$ , dado que para  $n > 1$  el valor crece exponencialmente con el número de procesadores y por tanto es poco fiable
  - Otras métricas usadas para los sistemas en paralelo son las siguientes:
    - El coste total de propiedad o *total cost of ownership* (TCO) representa el coste total de tener un sistema computacional durante su vida útil, incluyendo costes de adquisición, manutención, consumo de energía y eliminación. Además, hay costes de capital relacionados con la ocupación de espacio que ponderan mucho en el cálculo de la TCO, y factores como la

eficiencia energética, la densidad, el líquido de refrigeración y otros contribuyen a un bajo TCO

- La efectividad de uso de batería o *power usage effectiveness* (PUE) se mide de una cantidad de energía eléctrica que entra en un clúster o un centro de datos y, en general, se usa para calcular, de modo que se absorbe por los sistemas de información tecnológica. El PUE teóricamente perfecto está en 1, pero el PUE medio de un centro de datos es de 2.3

$$PUE = \frac{\text{Total power of installation}}{\text{Power of computers IT}} =$$

$$= \frac{\text{Cooling} + \text{Loss of Power} + \text{Lighting} + \text{TI}}{\text{IT}}$$

- La efectividad del reuso de energía o *energy reuse effectiveness* (ERE) se refiere a la explotación del calor generado por los sistemas computacionales. Esta técnica se denomina energía termal y su reuso implicaría PUE<1, que no tiene sentido matemático

$$ERE = \frac{\text{Total energy} - \text{Reused Energy}}{\text{Energy consumed by TI}}$$

- La efectividad del uso del carbono o *carbon usage effectiveness* (CUE) mide el CO2 total que se emite causado por un centro de datos y se divide por la energía de la carga del sistema (energía consumida por los servidores). La fórmula se puede descomponer en un factor de emisión de carbono (CEF) del sistema y el PUE, y el primero depende de la mezcla de producción de energía que alimenta el centro de datos

$$CUE = \frac{\text{CO2 emitted (KgCO2eq)}}{\text{Energy unit (Kwh)}} \times \frac{\text{Total energy of system}}{\text{Energy consumed by TI}} =$$

$$= CEF \times PUE$$

- La productividad de IT por watt incrustado (IT-PEW) mide la eficiencia de energía en relación a la productividad. La productividad es la producción del servicio sistemático de watts incrustados y son los watios de los sistemas de información tecnológica

$$IT - PEW = \frac{\text{Productivity}}{\text{Embedded Watt}}$$

- La eficiencia energética de un centro de datos y el índice de productividad de eficiencia energética y productividad (DC-EEP), que mide la eficiencia de todo un clúster o centro de datos

$$DC - EEP = \frac{Productivity}{Total\ cluster\ power}$$