

Tema 1. Introducción a la Computación Paralela

Grado en Ingeniería Informática

Ingeniería de Computadores

José María Granado Criado

Índice

- Introducción.
- ¿Por qué necesitamos cada vez más rendimiento?
- Problemas de los programas paralelos.
- Medidas de rendimiento.
- Diseño de programas paralelos.
- OpenMP-MPI.
 - OpenMP.
 - MPI.
 - Hibridación.

Introducción

- Desde 1986 hasta 2002 los microprocesadores fueron evolucionando como un cohete.
 - 50% de incremento en prestaciones por año.
- Desde entonces ha descendido hasta un aumento del 20% por año.
- ¿A qué se debe este descenso?
 - Transistores más pequeños = procesadores más rápidos.
 - Procesadores más rápidos = más consumo energético.
 - Más consumo energético = más calor.
 - Más calor = procesadores menos fiables.

Introducción

- Solución:
 - En lugar de diseñar y construir procesadores más rápidos, ahora se ponen varios procesadores en un sólo circuito integrado.
 - Procesadores multicore.
- Surge un nuevo problema:
 - Añadir muchos procesadores no ayuda si los programadores no son conscientes de que existen...
 - ... o no saben usarlos.
 - Los programas en serie no se pueden beneficiar de estos nuevos paradigmas.
 - Aunque sí ayudan a la multitarea y la concurrencia de aplicaciones.

Introducción

- ¿Qué entendemos por computación paralela?
 - Forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente.
- Clasificación
 - En función de la localización geográfica.
 - Computación paralela: Todas las unidades de procesamiento están situadas en la misma localización.
 - Clusters.
 - Multiprocesadores.
 - Multicores.
 - Computación distribuida: Las unidades de procesamiento están dispuestas entre diferentes localizaciones.
 - Grid.
 - Cloud.

Introducción

- Clasificación

- En función del tipo de aplicación.

- High Performance Computing o HPC (Computación de Alto Rendimiento): Consiste en ejecutar un único programa paralelo con el fin de reducir su tiempo de ejecución.
 - High Throughput Computing o HTC (Computación de Altas prestaciones): Consiste en ejecutar a la vez varias instancias de un único programa.
 - Ejecución con diferentes datos → Reducir el tiempo de ejecución del procesado de un gran volumen de datos independientes.
 - Ejecución con los mismos datos → Reducir el tiempo en la realización de estudios estadísticos en programas con componentes aleatorios.

- En función de la organización de memoria.

- Memoria compartida: Los elementos de proceso comparten la RAM.
 - Memoria distribuida: Los elementos de proceso tienen RAMs diferentes.

Introducción

- Clasificación
 - En función del hardware.
 - Paralelismo a nivel de bit.
 - Paralelismo a nivel de instrucción.
 - Procesadores vectoriales.
 - Segmentación.
 - Multiple issue.
 - Paralelismo a nivel de núcleos.
 - Procesadores multicore: comparten algún nivel de caché (normalmente la caché de nivel 3), la E/S y la RAM.
 - Paralelismo a nivel de procesadores.
 - Multiprocesadores: no comparten las cachés, sí la E/S y la RAM.
 - Paralelismo a nivel de equipos.
 - Multicomputadores: no comparten recursos.

Introducción

- Clasificación
 - En función del flujo de datos (Taxonomía de Flynn).
 - SISD (Single Instruction Stream, Single Data Stream).
 - Sin paralelismo.
 - SIMD (Single Instruction Stream, Multiple Data Stream)
 - Paralelismo de datos.
 - GPUs.
 - Instrucciones multimedia.
 - MISD (Multiple Instruction Stream, Single Data Stream).
 - Paralelismo de instrucciones.
 - Sistemas de tolerancia a fallos.
 - MIMD (Multiple Instruction Stream, Multiple Data Stream)
 - Paralelismo de instrucciones y datos.
 - Multicomputadores, multiprocesadores, multicores...

Introducción

- Clasificación

- Todas estas clasificaciones se pueden combinar:

- Cluster de docencia:

- 4 nodos → Paralelismo a nivel de equipos.
 - Cada nodo tiene 2 procesadores → Paralelismo a nivel de procesador.
 - Cada procesador tiene 4 núcleos con Hyperthreading → Paralelismo a nivel de núcleo.
 - Se pueden ejecutar aplicaciones tanto HTC como HPC.
 - Paralelismo MIMD.
 - Memoria compartida y distribuida.

- Ordenador con GPU:

- 1 procesador con 4 núcleos con Hyperthreading → Paralelismo a nivel de núcleo.
 - 1 GPU Tesla C2075 (448 núcleos).
 - Se pueden ejecutar aplicaciones tanto HTC como HPC.
 - Paralelismo MIMD y SIMD.
 - Memoria compartida (CPU) y distribuida (GPU).

Introducción

- ¿Por qué necesitamos cada vez más rendimiento?
 - La potencia de cómputo se incrementa, pero también los problemas y las necesidades son cada vez mayores.
 - Algunos problemas que nunca imaginamos que podrían resolverse se han podido resolver.
 - La decodificación del genoma humano.
 - Algunos problemas complejos están esperando para ser resueltos.
 - Modelos climáticos globales.
 - Simulación global de 250 años.
 - Parcela de 300x300 km de atmósfera y 100x100 km de océano
 - $\approx 10^{17}$ operaciones matemáticas simples.
 - 2 meses a 100 Gflops.
 - Intel Core i7 3960X 3.30 GHz DDR3-1600 141.09 GFLOPS. \rightarrow 1,42 meses.

Introducción

- ¿Por qué necesitamos cada vez más rendimiento?
 - Algunos problemas complejos están esperando para ser resueltos.
 - Plegamiento de proteínas.
 - Simulación de los cambios estructurales de una proteína durante un milisegundo → 100 días en un supercomputador ANTON.
 - ANTON → Supercomputador de 512 procesadores de propósito específico para plegamiento de proteínas.
 - Nuevos medicamentos.
 - Investigación en nuevas energías.
 - Análisis de datos.
 - El CERN genera 10 TB de datos diarios.
 - ...
 - Todos estos problemas y muchos más son inabarcables con sistemas de procesamiento serie.

Problemas de los programas paralelos

- El sistema de archivos.
 - Sistemas multiprocesador o multicore.
 - El sistema de archivos es compartido.
 - El problema no es el sistema de archivos en sí, sino el acceso simultáneo al mismo.
 - Sistemas tipo cluster.
 - El sistema de archivos está repartido entre los diferentes nodos.
 - Debe permitir que cada proceso acceda a los archivos que necesite.
 - Problema de asignación de archivos.
 - Consiste en repartir los archivos necesarios entre los nodos en función de qué nodo utilice cada archivo.
 - Un archivo sólo es accedido por un nodo.
 - Problema de replicación de archivos.
 - Consiste en copiar los archivos en varios nodos.
 - Problema de consistencia de datos en caso de escritura.

Problemas de los programas paralelos

- El sistema de archivos.
 - Factores que influyen en la optimización del sistema de archivos:
 - La velocidad de comunicación.
 - El balanceo de carga de los nodos.
 - El coste del protocolo de coherencia.
 - En sistemas distribuidos este problema se agrava ya que los costes de comunicación aumentan considerablemente.
 - Ejemplo: ¿Cómo reducir el tiempo necesario para enviar 100TB de información?
 - Usando fibra óptica a 300 Mbits/s → 30'86 días usando el 100% de la fibra, es decir, sin tener en cuenta información extra del protocolo, pérdidas de paquetes, latencias intermedias, transiciones de los datos por zonas lentas, ...

Problemas de los programas paralelos

- Distribución de la carga de trabajo.
 - En ocasiones, algunos procesadores/cores están altamente cargados mientras otros están ociosos.
 - Puede provocar que el tiempo de ejecución esté lejos del óptimo.
 - Repartir el trabajo entre todos los procesadores/cores de forma equitativa es lo que se conoce como distribución de la carga de trabajo.
 - El objetivo es aprovechar toda la potencia de cálculo del sistema.
 - No consiste simplemente en dividir los m datos del problema entre los n procesadores/cores del sistema.
 - En muchos problemas no se tarda lo mismo en procesar un dato que otro.
 - En sistemas heterogéneos, todos los procesadores no tienen por qué trabajar a la misma velocidad.

Problemas de los programas paralelos

- Partición de programas.
 - Consiste en cómo dividir un programa en componentes que sean ejecutados de forma paralela.
 - Un programa paralelo puede llegar a ser más lento que uno secuencial debido a la latencia.
 - Tiempos de sincronización.
 - Tiempos de comunicación.
 - Tiempos en la creación/destrucción de procesos.
 - Tiempos en los cambios de contexto entre procesos.
 - Tiempos de planificación.

Problemas de los programas paralelos

- Partición de programas.
 - A esto se le conoce como el grano de paralelismo.
 - Grano → secuencia de instrucciones que deben ser ejecutadas secuencialmente en un procesador.
 - Grano grueso → Poca división → Poco paralelismo.
 - Puede pasar que tareas potencialmente paralelas se ejecuten secuencialmente.
 - Grano fino → Mucha división → Mucho paralelismo.
 - Puede pasar que aumentemos tanto el nivel de paralelismo que haya muchas tareas con poco tiempo de cómputo → Latencia alta respecto al tiempo de ejecución de cada procesador.
 - Problema min-max.
 - Maximizar el paralelismo minimizando las latencias.
 - Puede hacerse de manera manual (programador) o automática (compilador, sistema operativo, etc.).

Problemas de los programas paralelos

- Acceso a memoria.
 - La jerarquía de memoria es la siguiente:
Memoria Principal – Memoria Caché – Registros
 - Siendo la memoria principal el dispositivo más lento y de mayor capacidad y los registros los dispositivos más rápidos y de menor capacidad.
 - Los procesadores pueden procesar información más rápido de lo que leen o escriben en la memoria principal.
 - La memoria es un cuello de botella.
 - De ahí el uso de las cachés y los registros.
 - Un acceso inadecuado a la memoria puede ralentizar considerablemente la ejecución de un programa paralelo.

Problemas de los programas paralelos

- Acceso a memoria.
 - La memoria caché.
 - La memoria caché es una memoria rápida de poca capacidad que sirve de intermediario entre procesador y memoria principal.
 - Se basa en los principios de localidad del software:
 - Localidad espacial: Si un dato es referenciado en un instante dado, los datos cercanos serán referenciados próximamente → Lectura de bloques.
 - Localidad temporal: Si un dato es referenciado en un instante dado, volverá a referenciarse próximamente → Mantener los datos en caché.
 - Cuando hablamos de memoria compartida, la caché es un problema:
 - Si un procesador modifica un dato en su caché que está siendo compartido con otros procesadores, debe informar al resto de que el dato se ha modificado.
 - Protocolos de coherencia caché.
 - Un mal acceso a los datos puede provocar grandes pérdidas de rendimiento debido a la invalidación de datos en las otras cachés.

Problemas de los programas paralelos

- Acceso a memoria.
 - La memoria caché.
 - El problema es que los programadores no tenemos ningún control sobre lo que se escribe en caché... ¿o sí?
 - Ejemplo secuencial:

Código 1:	Código 2:
<pre>for (i = 0; i < 4; i++) for (j = 0; j < 4; j++) Y[i][j] = A[i][j]*X[i][j];</pre>	<pre>for (j = 0; j < 4; j++) for (i = 0; i < 4; i++) Y[i][j] = A[i][j]*X[i][j];</pre>

- Caché:
 - » 2 líneas, cada línea puede almacenar 4 elementos de Y.
 - » Correspondencia directa.
 - » Bloques de 4 palabras.
- Memoria: Los datos de Y están almacenados consecutivamente.

Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]	Y[1][0]	...	Y[3][2]	Y[3][3]
---------	---------	---------	---------	---------	-----	---------	---------

Problemas de los programas paralelos

- Acceso a memoria.

- La memoria caché.

- Comportamiento de la caché en secuencial para el código 1 → Acceso por filas:

Código 1:

```
for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
        Y[i][j] = A[i][j]*X[i][j];
```

Instante Inicial			

Acceso a Y[0][0] → Fallo			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

Acceso a Y[0][1]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

Acceso a Y[0][2]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

Acceso a Y[0][3]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

Acceso a Y[1][0] → Fallo			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Acceso a Y[1][1]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Acceso a Y[1][2]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Acceso a Y[1][3]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

- En total, se producen 4 fallos de caché.
 - Los fallos se producen al acceder por primera vez al bloque (inevitables).

Problemas de los programas paralelos

- Acceso a memoria.

- La memoria caché.

- Comportamiento de la caché en secuencial para el código 2 → Acceso por columnas:

Código 2:

```
for (j = 0; j < 4; j++)
    for (i = 0; i < 4; i++)
        Y[i][j] = A[i][j]*X[i][j];
```

Instante Inicial			

Acceso a Y[0][0] → Fallo			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

Acceso a Y[1][0] → Fallo			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Acceso a Y[2][0] → Fallo			
Y[2][0]	Y[2][1]	Y[2][2]	Y[2][3]
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Acceso a Y[3][0] → Fallo			
Y[2][0]	Y[2][1]	Y[2][2]	Y[2][3]
Y[3][0]	Y[3][1]	Y[3][2]	Y[3][3]

Acceso a Y[0][1] → Fallo			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]
Y[3][0]	Y[3][1]	Y[3][2]	Y[3][3]

Acceso a Y[1][1] → Fallo			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Acceso a Y[2][1] → Fallo			
Y[2][0]	Y[2][1]	Y[2][2]	Y[2][3]
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Acceso a Y[3][1] → Fallo			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

- En total, se producen 16 fallos de caché.
 - Los fallos se producen por no aprovechar la localidad espacial.

Problemas de los programas paralelos

- Acceso a memoria.
 - La memoria caché.
 - En paralelismo, no sólo influye el orden en que se accede a los datos, también el número de hilos que se utilicen.
 - Ejemplo paralelo:

Código 1:	Código 2:
<pre>#omp parallel for num_threads (4) for (i = 0; i < 4; i++) for (j = 0; j < 4; j++) Y[i][j] = A[i][j]*X[i][j];</pre>	<pre>#omp parallel for num_threads (4) for (i = 0; i < 4; i++) #omp parallel for num_threads (4) for (j = 0; j < 4; j++) Y[i][j] = A[i][j]*X[i][j];</pre>

- Caché: igual que en secuencial.
- Memoria: Igual que en secuencial.
- Sistema de memoria compartida con 4 procesadores.

Problemas de los programas paralelos

- Acceso a memoria.
 - La memoria caché.

Código 1:

```
#omp parallel for num_threads (4)
for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
        Y[i][j] = A[i][j]*X[i][j];
```

- Comportamiento de la caché en paralelo con 4 hilos (hilos i=0-2):

Caché P0			
Acceso a Y[0][0] → Fallo			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

Caché P0			
Acceso a Y[0][1]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

Caché P0			
Acceso a Y[0][2]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

Caché P1			
Acceso a Y[1][0] → Fallo			
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Caché P1			
Acceso a Y[1][1]			
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Caché P1			
Acceso a Y[1][2]			
Y[1][0]	Y[1][1]	Y[1][2]	Y[1][3]

Caché P2			
Acceso a Y[2][0] → Fallo			
Y[2][0]	Y[2][1]	Y[2][2]	Y[2][3]

Caché P2			
Acceso a Y[2][1]			
Y[2][0]	Y[2][1]	Y[2][2]	Y[2][3]

Caché P2			
Acceso a Y[2][2]			
Y[2][0]	Y[2][1]	Y[2][2]	Y[2][3]

- En total, en cada caché se produce 1 fallo de caché.
- Los fallos se producen al acceder por primera vez al bloque (inevitables).

Problemas de los programas paralelos

- Acceso a memoria.
 - La memoria caché.
 - Comportamiento de la caché en paralelo con 16 hilos (hilos i=0 j=0-3):

Código 2:

```
#omp parallel for num_threads (4)
for (i = 0; i < 4; i++)
#omp parallel for num_threads (4)
    for (j = 0; j < 4; j++)
        Y[i][j] = A[i][j]*X[i][j];
```

Caché P0 Acceso a Y[0][0]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

- Fallo de lectura.
- Lee el bloque completo
- Escribe en Y[0][0]
- Invalida Y[0][0] en el resto de cachés (P1, P2 y P3)

Caché P2 Acceso a Y[0][2]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

- Fallo de lectura.
- Lee el bloque completo
- Escribe en Y[0][2]
- Invalida Y[0][2] en el resto de cachés (P0, P1 y P3)

Caché P1 Acceso a Y[0][1]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

- Fallo de lectura.
- Lee el bloque completo
- Escribe en Y[0][1]
- Invalida Y[0][1] en el resto de cachés (P0, P2 y P3)

Caché P3 Acceso a Y[0][3]			
Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]

- Fallo de lectura.
- Lee el bloque completo
- Escribe en Y[0][3]
- Invalida Y[0][3] en el resto de cachés (P0, P1 y P2)

- En total, se producen 16 fallos de caché.
 - » Al compartir bloques las cachés, éstos están continuamente invalidándose aunque en realidad no se modifican los mismos datos.
 - » Cada invalidación implica no sólo un nuevo acceso a memoria, sino que se debe esperar a que el bloque se actualice, es decir, se pierde paralelismo.

Problemas de los programas paralelos

- Acceso a memoria.
 - La memoria caché.
 - Ejemplo:
 - Tamaño de los datos: Matriz cuadrada de 20.000x20.000 enteros.
 - Operación: Multiplicación escalar.
 - Tiempos de ejecución:
 - » Secuencial con acceso por filas: 3'65 segundos.
 - » Secuencial con acceso por columnas: 9'14 segundos (2'5 veces más lento)
 - » Paralelo con 32 hilos con acceso por filas: 0'46 segundos
 - » Paralelo con 32 hilos con acceso por filas: 1'87 segundos (4 veces más lento).

Medidas de rendimiento

- Speedup (S)

- Relación entre el tiempo secuencial y el tiempo paralelo.

$$S = \frac{T_{secuencial}}{T_{paralelo}}$$

- Si tenemos p procesadores

- Speedup ideal $\rightarrow S=p$.
- Speedup real $\rightarrow S < p$.
 - Creación de hilos, cambios de contexto, comunicaciones, sincronizaciones, código estrictamente ejecutado en serie, ...
- Súper speedup $\rightarrow S > p$.
 - Beneficio acumulado de las múltiples cachés, acceso simultáneo a diferentes ficheros en distintos discos duros, ...

Medidas de rendimiento

- Eficiencia (E)
 - Factor de mejora de un código paralelo frente a su versión serie.

$$E = \frac{S}{p} = \frac{\frac{T_{secuencial}}{T_{paralelo}}}{p} = \frac{T_{secuencial}}{p \cdot T_{paralelo}}$$

- Eficiencia ideal $\rightarrow E=1$
- Eficiencia real $\rightarrow E<1$

- Overhead (T_0)
 - Relación entre el coste computacional del código serie respecto al código paralelo.

$$T_0 = p \cdot T_{paralelo} - T_{secuencial}$$

Medidas de rendimiento

- Escalabilidad.
 - Nos da una idea teórica del comportamiento del sistema al incrementar el número de procesadores.
 - Función de isoeficiencia.
 - Se basa en dos consideraciones:
 - La eficiencia de un sistema paralelo decrece cuando se incrementa el número de procesadores.
 - Para un número de procesadores dado, instancias más grandes de un problema obtienen mejores valores de speedup y eficiencia.
 - Si se incrementan a la vez estos dos factores, se podrá mantener la eficiencia.

$$W = \frac{E}{1 - E} \cdot T_0$$

- A menor valor de W , el sistema es más escalable.
 - » W crece exponencialmente respecto a $p \rightarrow$ Posiblemente escalable.
 - » W crece linealmente respecto a $p \rightarrow$ Altamente escalable.

Diseño de programas paralelos

- Grafo de dependencias.
 - Grafo acíclico dirigido donde los nodos representan tareas y una arista conectando una tarea fuente con otra destino representa que para poder ejecutarse la tarea destino debe ejecutarse previamente la tarea fuente.
 - Cada nodo del grafo suele etiquetarse con un valor proporcional al coste computacional de la tarea.
 - Dependiendo de la estrategia de resolución, se pueden obtener diferentes grafos.

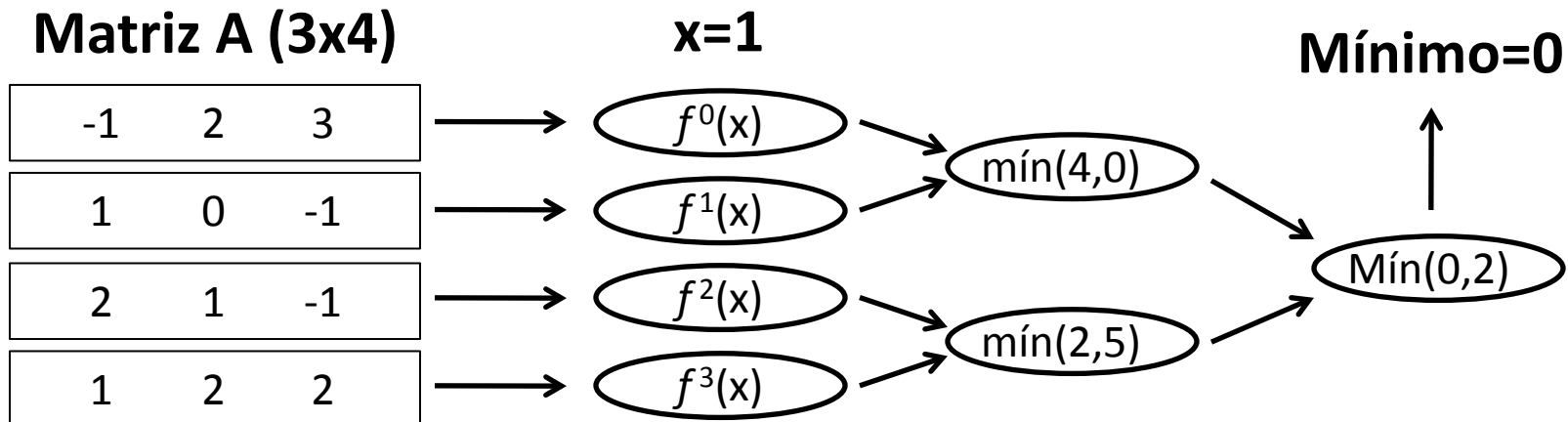
Diseño de programas paralelos

- Ejemplo: Evaluación polinomial.
 - Dada la matriz A de tamaño (NxM)

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,n} \\ \vdots & \ddots & \vdots \\ a_{m,0} & \cdots & a_{m,n} \end{pmatrix}$$

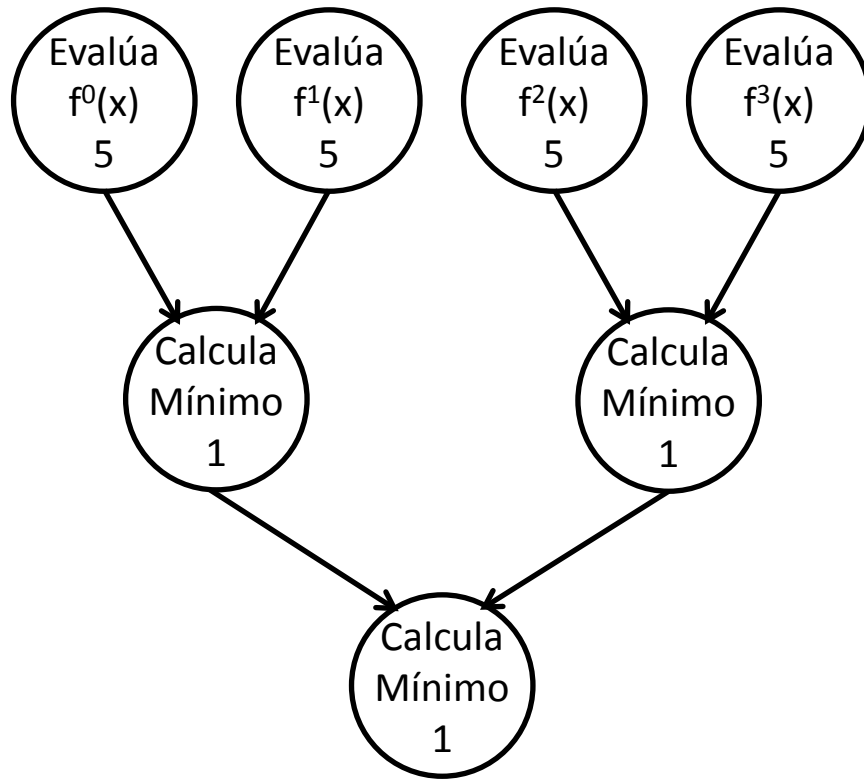
- $f^i(x) = a_{i,0} + a_{i,1}x + \cdots + a_{i,n-1}x^{n-1} + a_{i,n}x^n$, para $i=0, 1, \dots, m-1$

- Se desea obtener la función v, tal que $v = \min_{i=0}^{m-1} \{f(x)\}$

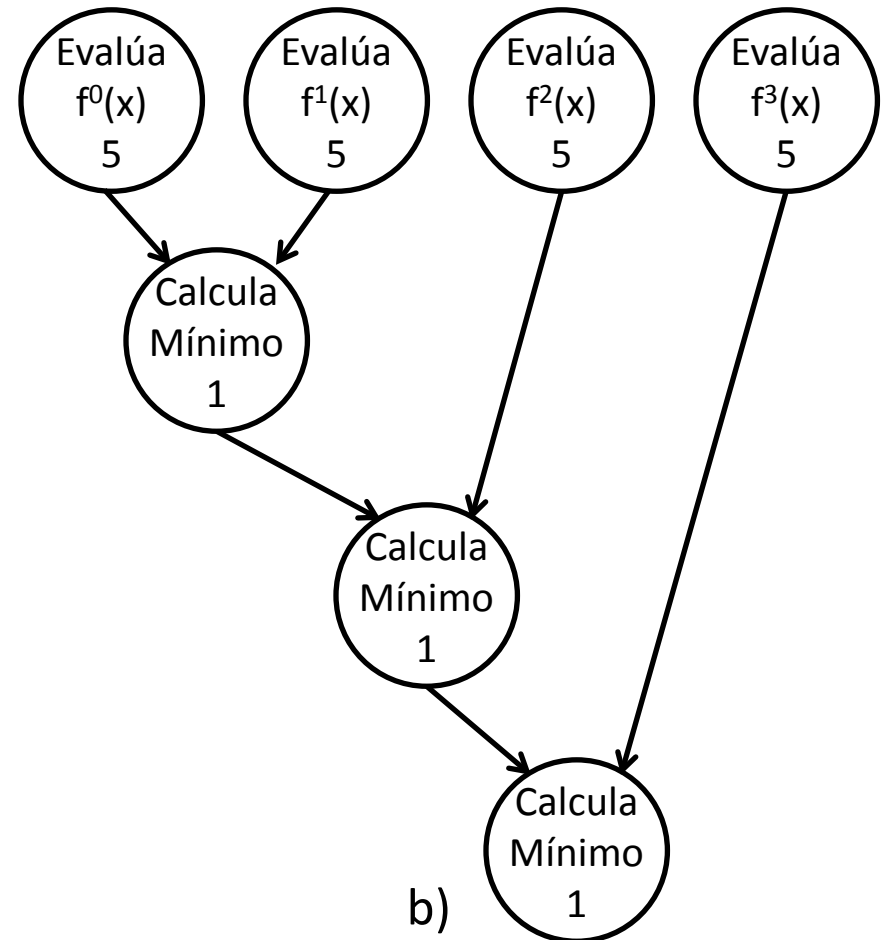


Diseño de programas paralelos

- Grafo de dependencias.
 - Diferentes grafos pueden resolver el mismo problema.



a)



b)

Diseño de programas paralelos

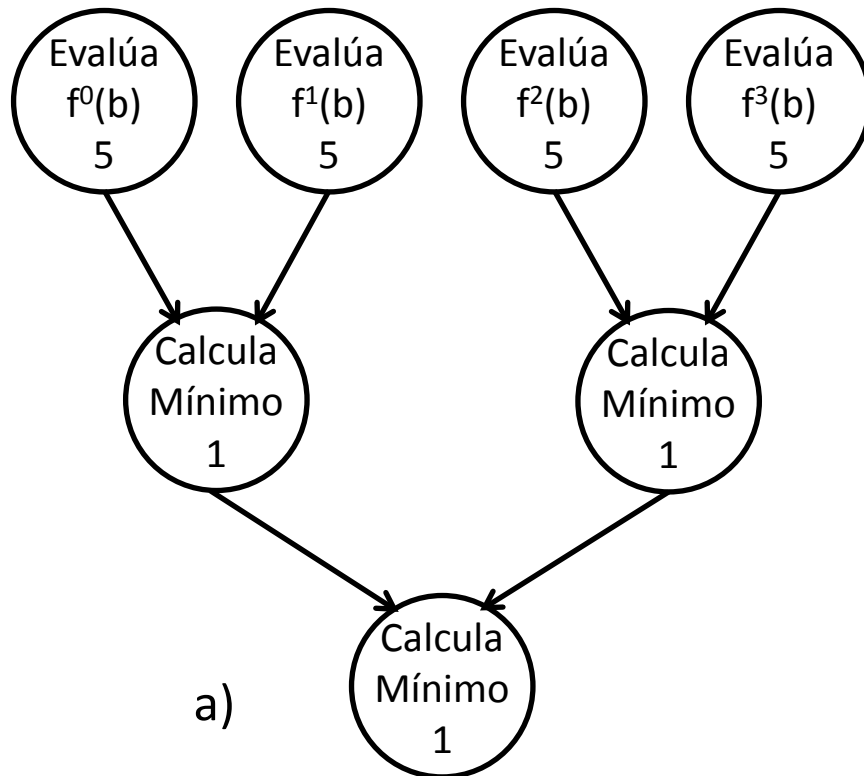
- Grafo de dependencias.
 - Grado de concurrencia.
 - Máximo grado de concurrencia.
 - Mayor numero de tareas cuya ejecución se podrá realizar al mismo tiempo en el grafo de dependencias.
 - Camino crítico.
 - Camino mas largo en el grafo desde un nodo de comienzo hasta un nodo de finalización.
 - La longitud L se obtiene sumando los costes de los nodos que componen el camino crítico.
 - Grado medio de concurrencia (M).
 - Número medio de tareas que se podrán ejecutar en paralelo, considerando todas las fases del algoritmo.
 - Para un grafo con N nodos:

$$M = \frac{\sum_{i=1}^n \text{coste}(\text{nodo}_i)}{L}$$

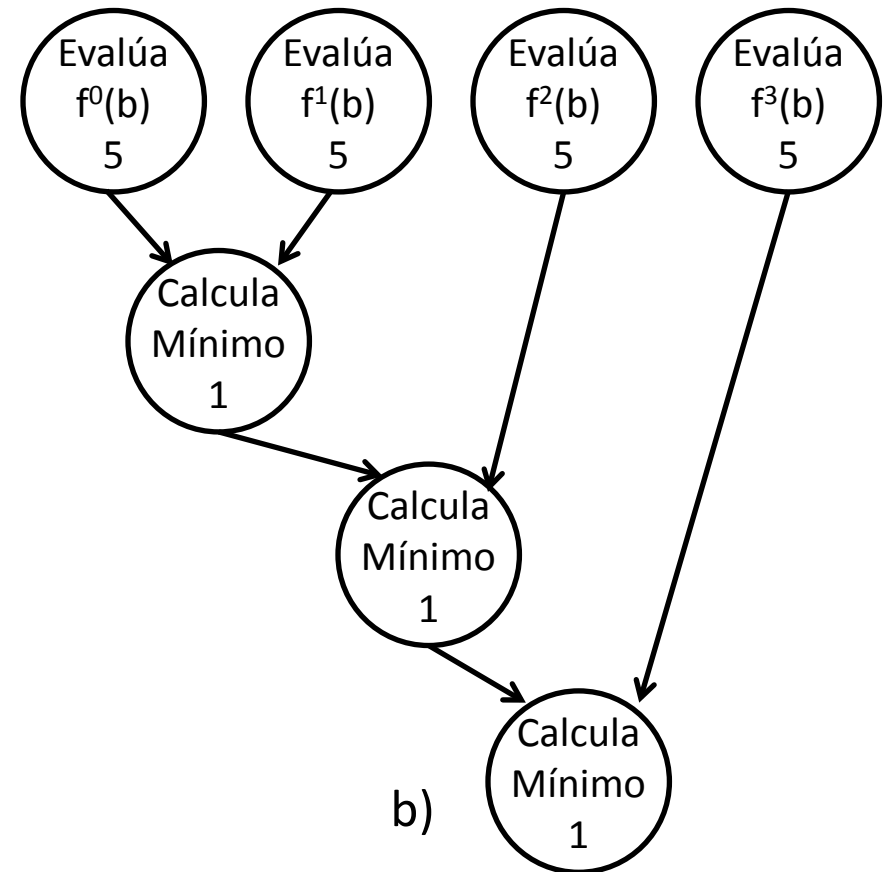
Diseño de programas paralelos

- Grafo de dependencias.

– Grado de concurrencia.



$$M(\text{grafo}(a)) = \frac{23}{7} = 3,28$$



$$M(\text{grafo}(b)) = \frac{23}{8} = 2,875$$

Diseño de programas paralelos

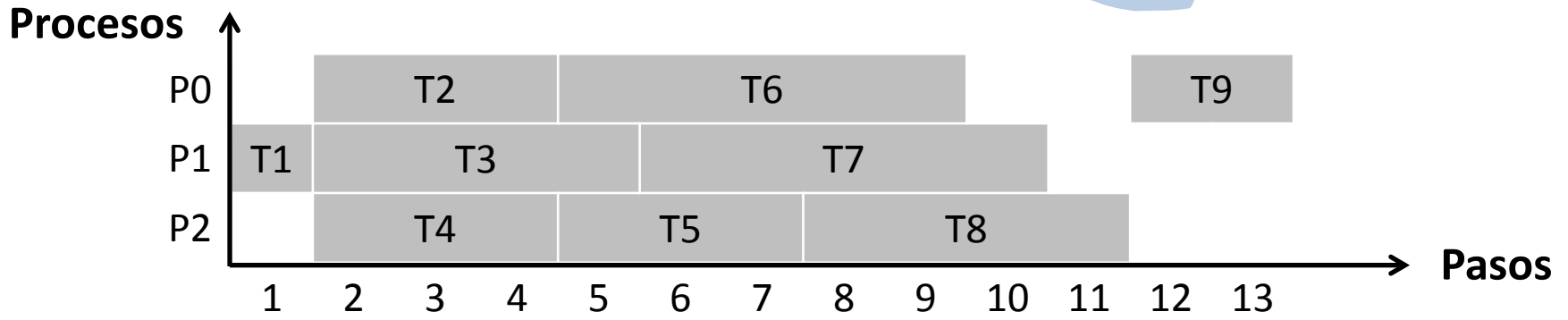
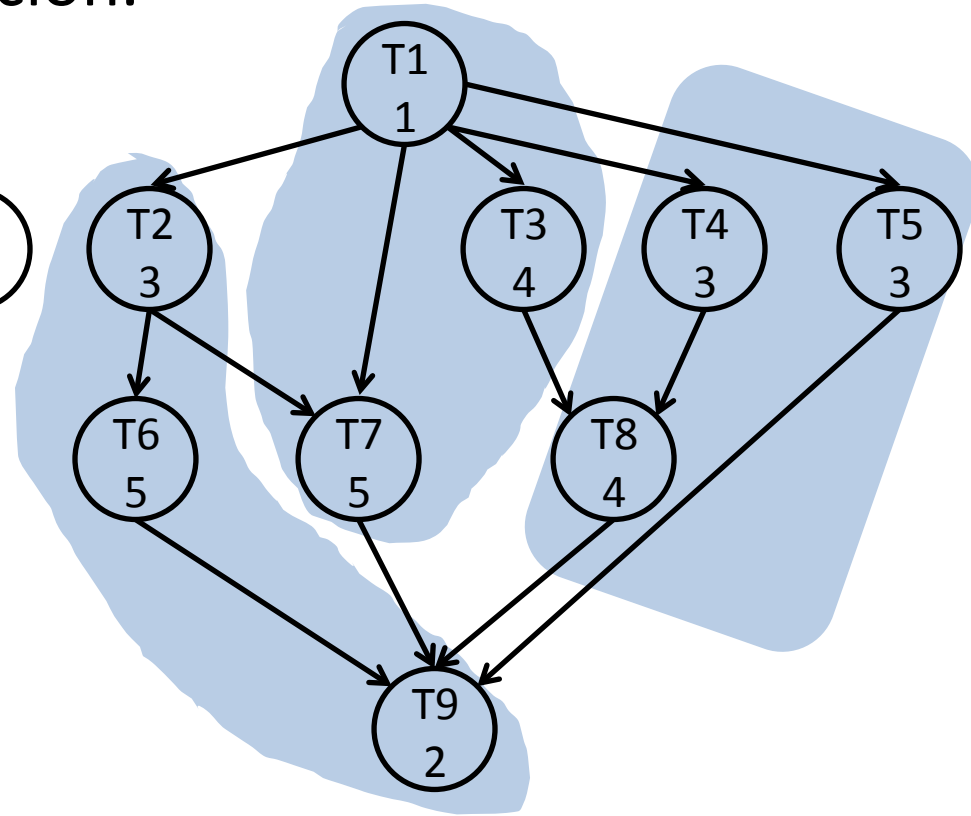
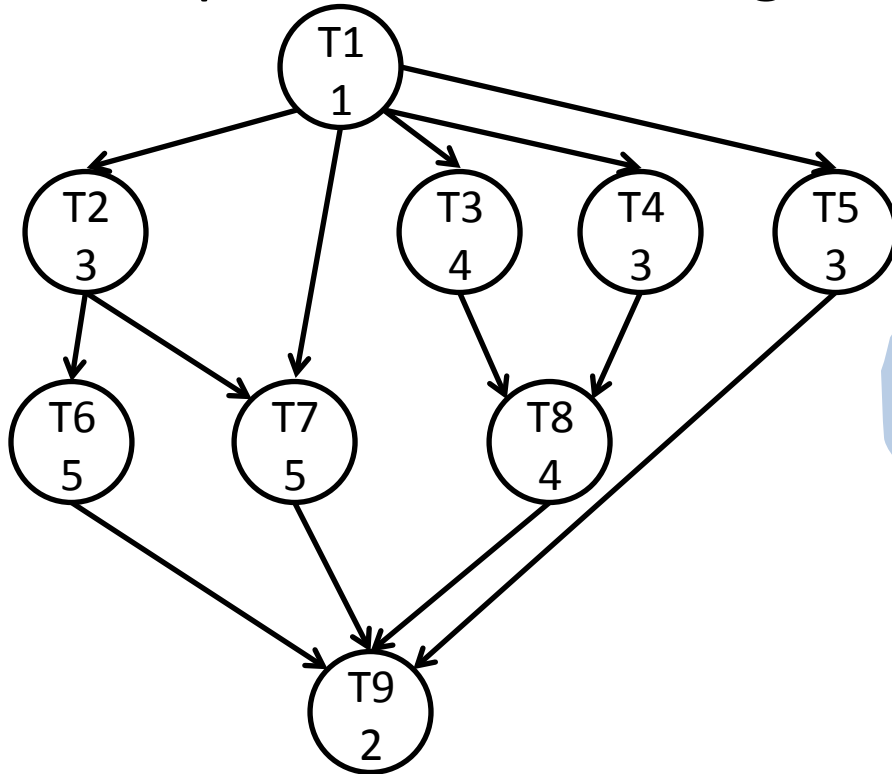
- Paradigmas de los programas paralelos.
 - Maestro/Esclavo.
 - El trabajo se divide entre los diferentes esclavos bajo la supervisión del maestro.
 - Divide y vencerás.
 - El problema se va dividiendo en subproblemas hasta que no se pueda o no se deba seguir dividiendo.
 - Cada subproblema se resuelve de forma independiente y paralela.
 - Suele ser necesario un preprocesado y/o un postprocesado.
 - Árbol binario.
 - Los cálculos sobre N datos se dividen en dos grupos de $N/2$ datos.
 - La división se sigue realizando sobre cada subgrupo.
 - El padre de cada dos subgrupos toma los valores de sus hijos, los procesa y se los devuelve a su padre.

Diseño de programas paralelos

- El problema de la asignación.
 - Problema:
 - Correspondencia tareas-procesos.
 - Agrupamiento previo de algunas tareas en una única tarea.
 - Selección de un orden de ejecución para las tareas.
 - Objetivo:
 - Minimizar tiempo de ejecución total.
 - Estrategias:
 - Tiempo de computación:
 - Maximizar la concurrencia, asignando tareas independientes a diferentes procesos.
 - Tiempo de comunicación:
 - Asignando tareas que interactúen mucho al mismo proceso.
 - Tiempo de ocio
 - Evitar fuentes de ociosidad.

Diseño de programas paralelos

- El problema de la asignación.

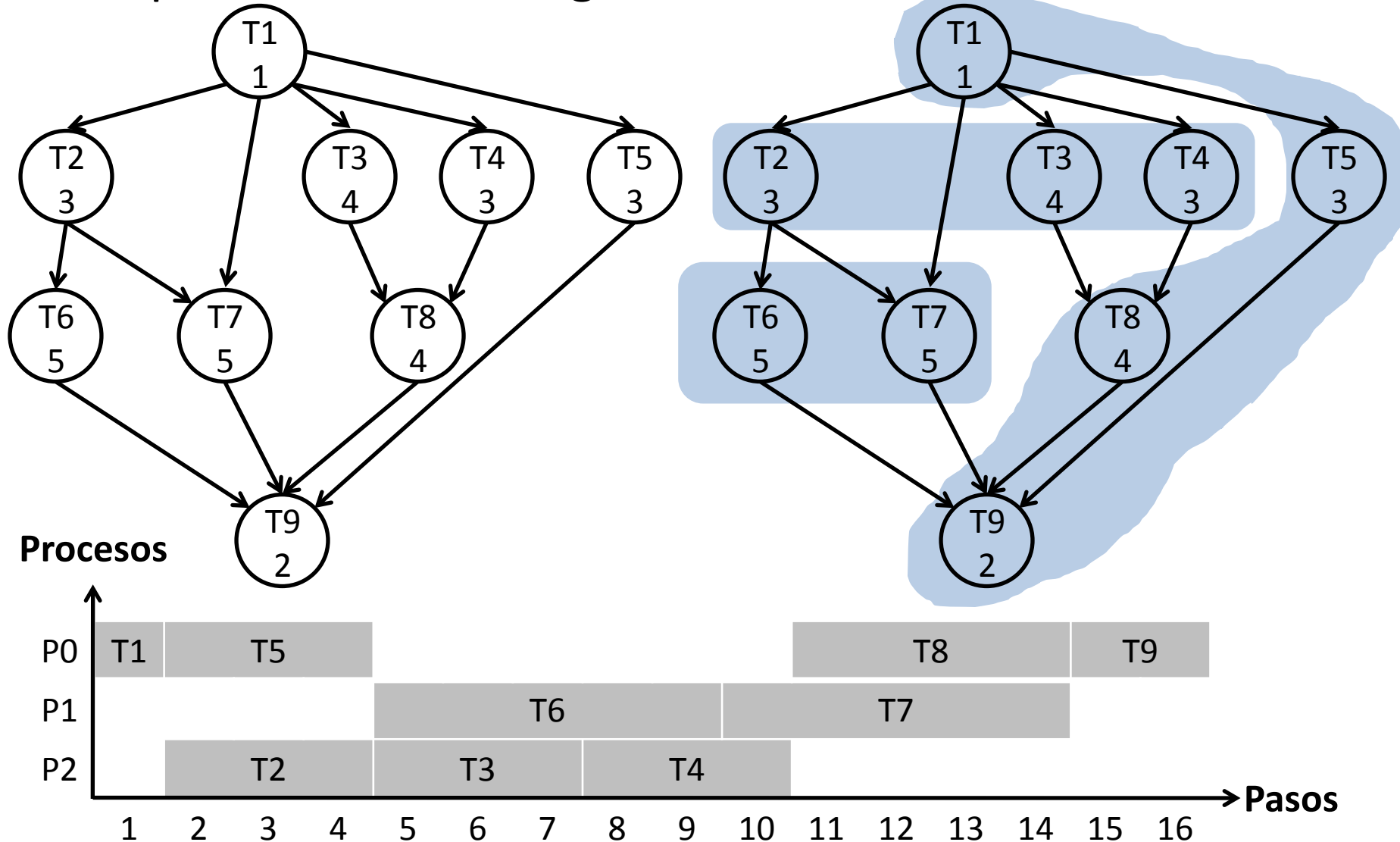


Diseño de programas paralelos

- El problema de la asignación.
 - Fuentes de ociosidad.
 - Desequilibrios de carga.
 - Se busca equilibrar tanto los cálculos como las comunicaciones.
 - Espera entre procesos debida a dependencias entre tareas.
 - Asignar una carga de trabajo similar a cada proceso no es suficiente.
 - Es el grafo de dependencias el que determina que tareas se pueden ejecutar en paralelo.

Diseño de programas paralelos

- El problema de la asignación.



Diseño de programas paralelos

- El problema de la asignación.
 - Asignación estática.
 - Asignación tarea-proceso y orden de ejecución conocidos antes de ejecución.
 - Estimación del numero de tareas, tiempo de ejecución y costes de comunicación.
 - Agrupación de tareas para reducir costes.
 - Asociar tareas con procesos.
 - Problema NP-completo, aunque existen estrategias especializadas y heurísticas.

Diseño de programas paralelos

- El problema de la asignación.
 - Asignación dinámica.
 - Reparto del trabajo computacional entre los procesos.
 - ¿Cuándo se utilizan estas técnicas?
 - Las tareas se generan dinámicamente.
 - Tamaño de las tareas no se conoce a priori.
 - Problemas a resolver:
 - ¿Qué procesos recopilan la información para tomar decisiones de redistribución?
 - ¿Qué condiciones activan la transferencia de carga?
 - ¿Qué procesos deben recibir carga?
 - Desventaja:
 - Añaden sobrecargas en tiempo de ejecución.
 - Principal ventaja:
 - No se necesita conocer el comportamiento antes de la ejecución.

OpenMP-MPI - OpenMP

- Una API para programación en memoria compartida.
- MP = multiprocessing
- Diseñado para sistemas en los que cada thread o hilo puede tener acceso a toda la memoria disponible.
- Directivas (en OpenMP → *#pragma*):
 - Instrucciones especiales de preprocesador.
 - Se añaden a un sistema normalmente para permitir comportamientos que no forman parte de la especificación básica de C.
 - Los compiladores que no soportan “pragmas”, las ignoran.
 - *#pragma omp parallel* → Define una región paralela.
- Equipo.
 - Colección de threads que ejecutan un bloque en paralelo.

OpenMP-MPI - OpenMP

- Cláusulas.
 - Texto que modifica una directiva.
 - Una misma directiva puede tener múltiples cláusulas.
- Establecer el número de threads que se usarán.
 - Para establecer el número de threads que se crearán, tenemos dos opciones:
 - Función *set_num_threads(thread_count)*.
 - Cláusula *num_threads (thread_count)*.
 - *#pragma omp parallel num_threads (thread_count)*.
 - Puede haber limitación en el número de threads que un programa puede lanzar.
 - El estándar OpenMP no garantiza que se ejecuten *thread_count* threads.
 - La mayoría de los sistemas pueden lanzar cientos o miles.

OpenMP-MPI - OpenMP

- Funciones de identificación de threads.
 - *int omp_get_thread_num(void)* → Devuelve el identificador del thread actual.
 - *int omp_get_num_threads(void)* → Devuelve el número total de threads en ejecución.
- Medición del tiempo.
 - *double omp_get_wtime(void)* → Permite medir los tiempos de un thread de forma independiente al resto.
- Sincronización de threads.
 - Directiva *#pragma omp barrier* → Los threads se sincronizan en este punto.
 - Cláusula *nowait* → Los threads no esperarán a la sincronización implícita de un bloque de una directiva.

OpenMP-MPI - OpenMP

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

void Hello(void); /* Thread function */

int main(int argc, char argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    # pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

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);
} /* Hello */
```

OpenMP-MPI - OpenMP

`gcc omp_hello.c -o omp_hello [-Wall] -fopenmp` **Compilación**

`$./omp_hello 4` **Ejecución con 4 threads**

Posibles salidas

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4

OpenMP-MPI - OpenMP

- **Ámbito de las variables.**
 - En OpenMP, el ámbito de una variable se refiere al conjunto de threads que pueden acceder a esa variable en el bloque paralelo.
 - El ámbito se define mediante cláusulas:
 - *shared (n)* → *n* es accesible por todos los threads de un equipo. Por defecto, las variables declaradas antes del bloque paralelo son compartidas.
 - *private (n)* → Se crea una copia privada de *n* en cada thread del bloque paralelo.
 - *firstprivate (n)* → Igual que *private* pero inicializa todas las copias privadas a su valor inicial.
 - *Lastprivate (n)* → Igual que *private* pero el elemento original se actualizará al terminar la tarea.
 - *default (none)* → Obliga al programador a indicar el ámbito de cada variable.
 - *#pragma omp parallel shared (n, m) private (k)*

OpenMP-MPI - OpenMP

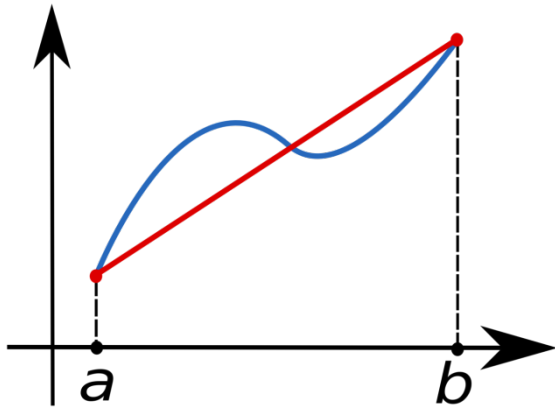
- Regiones críticas.
 - Todos los threads ejecutarán la región crítica en serie.
 - No pueden mezclarse entre ellas.
 - Pueden definirse de tres maneras:
 - *#pragma omp atomic*.
 - Secciones críticas simples del tipo carga-modificación-almacenamiento.
 - Puede usarse con los siguientes operadores: +, *, -, /, &, ^, |, << y >>.
 - *#pragma omp critical [(name)]*.
 - Para bloques de código más complejos que *atomic*.
 - Si se establece un nombre, se pueden crear diferentes regiones críticas.
 - Cerrojos (*lock*).
 - Se pueden establecer diversas regiones críticas con diferentes locks.
 - Sólo un thread puede acceder a un bloque en una región crítica de un *lock* concreto.
 - » A diferencia de *critical*, pueden definirse trabajos opcionales a realizar en caso de que la región crítica esté ocupada.

OpenMP-MPI - OpenMP

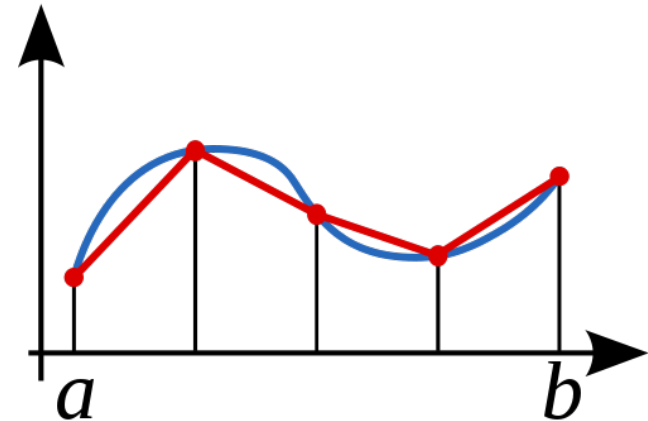
- Regiones críticas.
 - Puede definirse de tres maneras:
 - Cerrojos (*lock*).
 - *void omp_init_lock (omp_lock_t *lock):*
 - » Inicializa el cerrojo y lo deja abierto.
 - *void omp_set_lock (omp_lock_t *lock):*
 - » El thread actual queda bloqueado hasta que se abra el cerrojo.
 - » Si la invoca el thread que lo cerró, se produce un punto muerto.
 - *omp_unset_lock (omp_lock_t *lock):*
 - » Abre el cerrojo.
 - » Debe haber sido cerrado por el mismo thread que trata de abrirlo.
 - *int omp_test_lock(omp_lock_t *lock):*
 - » Si el cerrojo está abierto, lo cierra y devuelve true, si no, devuelve false.
 - » A diferencia *de omp_set_lock*, el thread no se bloquea.
 - *void omp_destroy_lock (omp_lock_t *lock):*
 - » Destruye el cerrojo. Debe estar abierto.

OpenMP-MPI - OpenMP

- Ejemplo. La regla del trapecio.
 - Método matemático de integración numérica.
 - Método para calcular aproximadamente una integral definida.



$$\int_a^b f(x) dx \approx (b-a) \frac{f(a) + f(b)}{2}.$$



$$\int_a^b f(x) dx \sim \frac{b-a}{n} \left(\frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k \frac{b-a}{n}\right) \right)$$

OpenMP-MPI - OpenMP

- Ejemplo. La regla del trapecio.
 - Implementación serie.

$$\int_a^b f(x)dx \sim \frac{b-a}{n} \left(\frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k \frac{b-a}{n}\right) \right)$$

/ Input: a, b, n /

h = (b-a)/n;

approx = (f(a) + f(b))/2.0;

for (i = 1; i <= n-1; i++) {

 x_i = a + i*h;

 approx += f(x_i);

}

approx = h*approx;

OpenMP-MPI - OpenMP

- Ejemplo. La regla del trapecio.

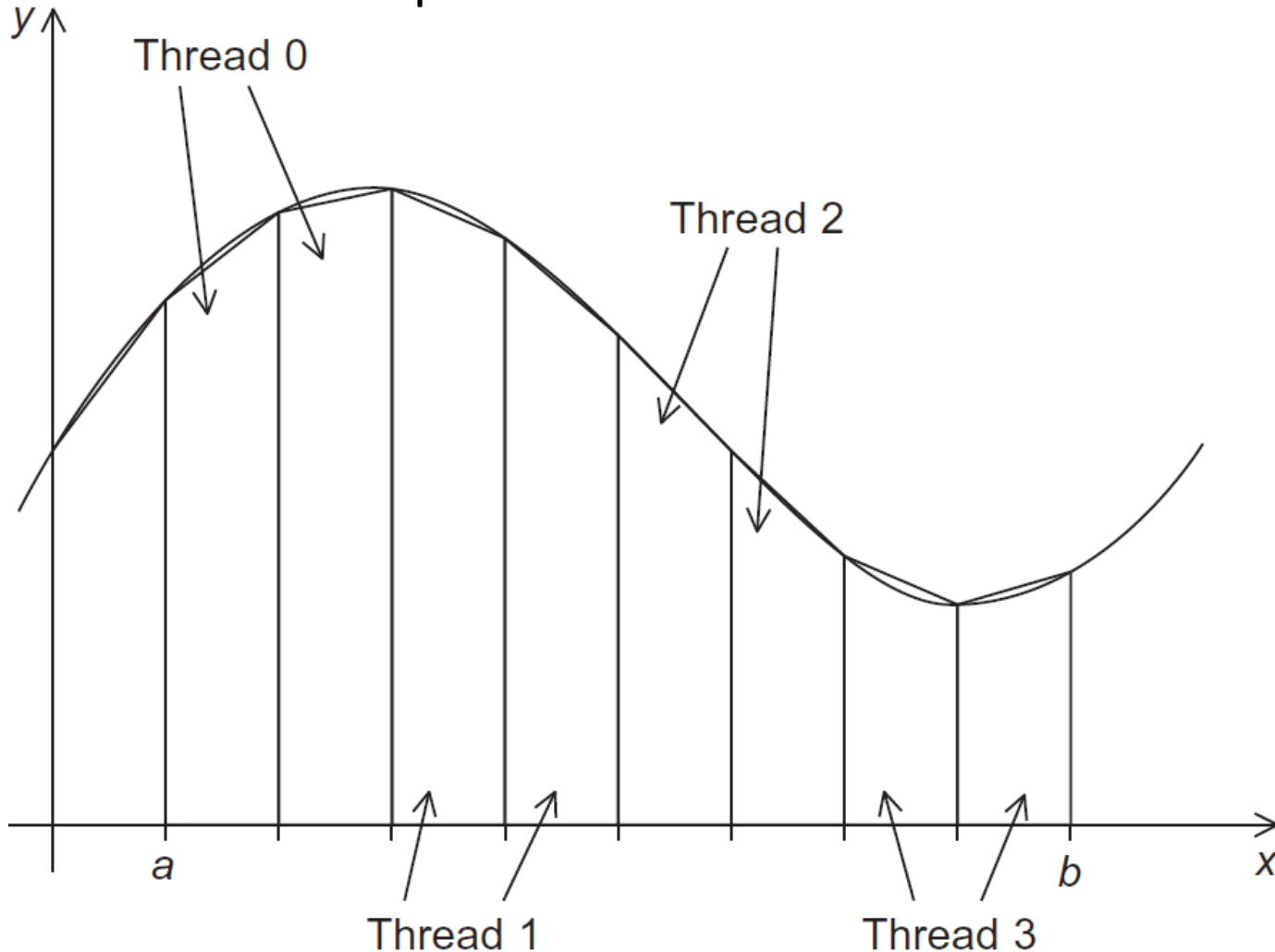
- Primera versión OpenMP.

$$\int_a^b f(x)dx \sim \frac{b-a}{n} \left(\frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k \frac{b-a}{n}\right) \right)$$

- Identificamos dos tipos de tareas:
 - Cálculo de las áreas individuales de los trapezoides.
 - Suma de las áreas de los trapezoides.
- No hay comunicación entre las tareas al principio, pero cada tarea se comunicará al final con las otras.
- Asumimos que puede haber más trapezoides que cores.
- Agregamos tareas asignando bloques contiguos de trapezoides a cada thread (y un solo thread a cada core).

OpenMP-MPI - OpenMP

- Ejemplo. La regla del trapecio.
 - Primera versión OpenMP.



OpenMP-MPI - OpenMP

- Ejemplo. La regla del trapecio.

– Primera versión OpenMP. $\int_a^b f(x)dx \sim \frac{b-a}{n} \left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k\frac{b-a}{n}\right) \right)$

```
void Trap(double a, double b, int n, double *global_result_p) {  
    double h, x, my result, local a, local b;  
    int i, local n;  
    int my_rank = omp_get_thread_num();  
    int thread_count = omp_get_num_threads();  
    h = (b-a)/n;  
    local_n = n/thread_count;  
    local_a = a + my_rank*local_n;  
    local_b = local_a + local_n;  
    my_result = (f(local_a) + f(local_b))/2.0;  
    for (i = 1; i <= local_n-1; i++) {  
        x = local_a + i*h;  
        my_result += f(x);  
    }  
    my_result = my_result*h;  
    *global_result_p += my_result;  
} / Trap /
```

```
global_result=0.0;  
#pragma omp parallel num_threads(num_th);  
    Trap (a, b, n, &global_result);
```

Resultados impredecibles
cuando dos (o más)
threads intentan ejecutar
de forma simultánea

OpenMP-MPI - OpenMP

- Ejemplo. La regla del trapecio.

– Primera versión OpenMP.

$$\int_a^b f(x)dx \sim \frac{b-a}{n} \left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k\frac{b-a}{n}\right) \right)$$

```
void Trap(double a, double b, int n, double *global_result_p) {  
    double h, x, my_result, local_a, local_b;  
    int i, local_n;  
    int my_rank = omp_get_thread_num();  
    int thread_count = omp_get_num_threads();  
    h = (b-a)/n;  
    local_n = n/thread_count;  
    local_a = a + my_rank*h;  
    local_b = local_a + local_n*h;  
    my_result = (f(local_a) + f(local_b))/2.0;  
    for (i = 1; i <= local_n-1; i++) {  
        x = local_a + i*h;  
        my_result += f(x);  
    }  
    my_result = my_result*h;  
    #pragma omp critical  
        *global_result_p += my_result;  
} / Trap /
```

```
global_result=0.0;  
#pragma omp parallel num_threads(num_th);  
    Trap (a, b, n, &global_result);
```

Sólo un thread puede ejecutar este bloque.

OpenMP-MPI - OpenMP

- Otras directivas de exclusión de threads.
 - *#pragma omp single.*
 - El código que define esta directiva sólo será ejecutado por un único thread de todos los lanzados.
 - No tiene que ser obligatoriamente el thread padre.
 - *#pragma omp master.*
 - El código que define esta directiva sólo será ejecutado por un único thread de todos los lanzados.
 - Obligatoriamente será el thread padre.

OpenMP-MPI - OpenMP

- Cláusula de reducción.
 - Un operador de reducción es una operación binaria (como la suma o la multiplicación).
 - Una reducción es un cálculo que se aplica de forma repetitiva a una secuencia de comandos para obtener un único resultado.
 - Todos los resultados intermedios de la operación deben almacenarse en la misma variable: la variable de reducción.
 - Internamente se crea una copia privada para cada hilo.
 - *#pragma omp parallel ... reduction (<operador>: <variable>)*
 - <operador>: +, *, -, &, |, ^, &&, ||
 - A continuación se añade la operación que se va a reducir.

OpenMP-MPI - OpenMP

- Cláusula de reducción.
 - En el código anterior:
 - La función *Trap* devuelve el resultado parcial.
 - Con la reducción hacemos la suma de los resultados parciales.

```
double Trap(double a, double b, int n) {  
    double h, x, my_result, local_a, local_b;  
    int i, local_n;  
    int my_rank = omp_get_thread_num();  
    int thread_count = omp_get_num_threads();  
    h = (b*a)/n;  
    local_n = n/thread_count;  
    local_a = a + my_rank*local_n;  
    local_b = local_a + local_n;  
    my_result = (f(local_a) + f(local_b))/2.0;  
    for (i = 1; i <= local_n-1; i++) {  
        x = local_a + i*h;  
        my_result += f(x);  
    }  
    Return my_result;  
} / Trap /
```

```
global_result=0.0;  
#pragma omp parallel num_threads(thread_count)  
reduction(+: global_result)  
    global_result+=Trap(a, b, n);
```

OpenMP-MPI - OpenMP

- Directiva *for*.
 - Hace un “fork” de un equipo de threads para ejecutar el siguiente bloque estructurado.
 - El bloque que siga a la directiva debe ser un bucle for.
 - Dos opciones:
 - *#pragma omp for* dentro de una directiva *#pragma omp parallel*
 - La región paralela se crea antes de que se realice el reparto del for.
 - Si hay varios *#pragma omp for* en un mismo código, esta opción es más eficiente, ya que únicamente se crea una región paralela.
 - *#pragma omp parallel for*
 - Se crea la región paralela a la vez que se distribuye el trabajo de los hilos.
 - Puede ser poco eficiente al tener que crear, para cada *#pragma omp for*, toda la región paralela (incremento de la latencia).
 - El sistema paraleliza el bucle dividiendo las iteraciones del bucle entre todos los threads.

OpenMP-MPI - OpenMP

- Directiva *for*.
 - En el ejemplo de la regla del trapecio:

Código secuencial

```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```

Código paralelo

```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
#pragma omp parallel for num threads(thread_count) reduction(+: approx)  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```

- Formas permitidas del for:

for	(index < end	index++
			index <= end	++index
			index >= end	index--
			index > end	--index
		index = start ;	index >= end ;	index += incr
				index -= incr
				index = index + incr
				index = incr + index
		index = index - incr		

OpenMP-MPI - OpenMP

- Directiva *for*.
 - Reglas:
 - La variable *index* debe tener tipo *integer* o puntero (no puede ser un *float*).
 - Las expresiones *start*, *end*, e *incr* deben tener un tipo compatible.
 - Por ejemplo, si *index* es un puntero, entonces *incr* debe ser un entero.
 - Las expresiones *start*, *end*, e *incr* no deben cambiar durante la ejecución del bucle.
 - Durante la ejecución del bucle, la variable *index* sólo puede ser modificada por la “expresión de incremento” en el *for*.

OpenMP-MPI - OpenMP

- Directiva *for*.
 - Funcionamiento.

```
N=100;
#pragma omp parallel num_threads(4)
{
    printf("Soy el hilo %d\n", omp_get_thread_num());
    #pragma omp for
    for (i=0; i<N; i++) {
        a[i]=b[i]*c[i]
    }
```

OpenMP-MPI - OpenMP

- Directiva *for*.
 - Funcionamiento.

```
N=100;
```

```
#pragma omp parallel num_threads(4)
```

```
{
```

```
    printf("Soy el hilo %d\n", omp_get_thread_num());
```

```
    #pragma omp for
```

```
    for (i=0; i<N; i++) {
```

```
        a[i]=b[i]*c[i]
```

```
    }
```

Crea una región paralela de 4 hilos con el código contenido entre las llaves.



OpenMP-MPI - OpenMP

- Directiva *for*.
 - Funcionamiento.

```
N=100;  
#pragma omp parallel num_threads(4)  
{  
    printf("Soy el hilo %d\n", omp_get_thread_num());  
    #pragma omp for  
    for (i=0; i<N; i++) {  
        a[i]=b[i]*c[i]  
    }  
}
```

Todos los hilos ejecutan el código completo de la región paralela.

printf("Soy el hilo 0\n");

printf("Soy el hilo 1\n");

printf("Soy el hilo 2\n");

printf("Soy el hilo 3\n");

OpenMP-MPI - OpenMP

- Directiva *for*.
 - Funcionamiento.

```
N=100;  
#pragma omp parallel num_threads(4)  
{  
    printf("Soy el hilo %d\n", omp_get_thread_num());  
    #pragma omp for  
    for (i=0; i<N; i++) {  
        a[i]=b[i]*c[i]  
    }  
}
```

El contenido del *#pragma omp for* se reparte entre todos los hilos de la región paralela

printf("Soy el hilo 0\n");
for (i=0; i<25; i++) {
 a[i]=b[i]*c[i]
}

printf("Soy el hilo 1\n");
for (i=25; i<50; i++) {
 a[i]=b[i]*c[i]
}

printf("Soy el hilo 2\n");
for (i=50; i<75; i++) {
 a[i]=b[i]*c[i]
}

printf("Soy el hilo 3\n");
for (i=75; i<100; i++) {
 a[i]=b[i]*c[i]
}

OpenMP-MPI - OpenMP

- Directiva *for*.
 - Dependencia de datos: Sucesión de Fibonacci.

```
fibo[0] = fibo[1] = 1;  
for (i = 2; i < n; i++)  
    fibo[i] = fibo[i-1] + fibo[i-2];
```



```
fibo[0] = fibo[1] = 1;  
#pragma omp parallel for num_threads(2)  
for (i = 2; i < n; i++)  
    fibo[i] = fibo[i-1] + fibo[i-2];
```

Esto es correcto

1 1

2	3	5	8
---	---	---	---

13	21	34	55
----	----	----	----

Thread 1 Thread 2

Pero a veces
obtenemos esto

1 1

2	3	5	8
---	---	---	---

0	0	0	0
---	---	---	---

Thread 1 Thread 2

OpenMP-MPI - OpenMP

- Directiva *for*.
 - Dependencia de datos: Sucesión de Fibonacci.
 - ¿Qué ocurre?
 - Los compiladores OpenMP no comprueban las dependencias entre las iteraciones en un bucle que está siendo paralelizado con una directiva paralela.
 - Un bucle en el que los resultados de una o más iteraciones dependen de otra iteración, normalmente no puede ser paralelizado correctamente en OpenMP.

OpenMP-MPI - OpenMP

- Directiva *for*.

– Dependencia de datos: Estimando π . $\pi = 4 \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right] = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$

Código secuencial

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2k+1);
    factor = -1*factor;
}
Pi_approx = 4.0*sum;
```

Código paralelo

```
double factor = 1.0;
double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) reduction(+: sum)
    for (k = 0; k < n; k++){
        sum += factor/(2k+1);
        factor = -1*factor;
    }
Pi_approx = 4.0*sum;
```

Depende del valor de la iteración anterior

Solución a la dependencia

```
double factor, double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) reduction(+: sum) private(factor)
    for (k = 0; k < n; k++){
        if (k%2==0) factor=1.0;
        else factor=-1.0;
        sum += factor/(2k+1);
    }
Pi_approx = 4.0*sum;
```

factor tiene ámbito privado para que cada hilo tenga su copia independiente.

OpenMP-MPI - OpenMP

- Planificación – La cláusula *schedule*.
 - *Schedule (tipo[, tamaño])*.
 - Tipo:
 - *static*: las iteraciones pueden asignarse a los threads antes de que el bucle se ejecute.
 - *dynamic* o *guided*: las iteraciones se asignan a los threads mientras el bucle se está ejecutando.
 - *auto*: el compilador y el planificador determinan el orden.
 - *runtime*: el orden se determina en tiempo de ejecución en función de la variable de entorno *OMP_SCHEDULE* (*static*, *dynamic* o *guided*).
 - El tamaño es un entero positivo.

OpenMP-MPI - OpenMP

- Planificación – La cláusula *schedule*.

- *static*.

- Se reparte el trabajo entre los threads en grupos de datos con el tamaño indicado en la cláusula.
 - Ejemplo: Doce iteraciones, 0, 1, . . . , 11, y tres threads.

schedule (static, 1)

Thread 0: 0, 3, 6, 9

Thread 1: 1, 4, 7, 10

Thread 2: 2, 5, 8, 11

schedule (static, 2)

Thread 0: 0, 1, 6, 7

Thread 1: 2, 3, 8, 9

Thread 2: 4, 5, 10, 11

schedule (static, 4)

Thread 0: 0, 1, 2, 3

Thread 1: 4, 5, 6, 7

Thread 2: 8, 9, 10, 11

- Ventaja: Menor latencia al hacer un reparto fijo.
 - Desventaja: Un thread puede quedarse con los trabajos más costosos.
 - Ejemplo (para tamaño 4): Imaginemos que los datos 0, 1, 2 y 3 requieren el doble de tiempo de procesamiento que el resto de datos.
 - Los threads 1 y 2 acabarían a la vez que el thread 0 acaba con el dato 1.
 - El thread 0 tiene que procesar los datos 2 y 3 mientras los otros threads están inactivos.

OpenMP-MPI - OpenMP

- Planificación – La cláusula *schedule*.
 - *dynamic*.
 - Los datos también se trocean en trozos consecutivos.
 - Cada thread ejecuta un trozo, y cuando termina, solicita otro trozo al planificador.
 - Este bucle continúa hasta que las iteraciones se completan.
 - El tamaño, si no se especifica, por defecto es 1.
 - Ventaja:
 - Mientras haya trozos que procesar, no habrá threads sin carga de trabajo.
 - Desventaja:
 - Puede aumentar la latencia.

OpenMP-MPI - OpenMP

- Planificación – La cláusula *schedule*.
 - *guided*.
 - Cada thread ejecuta un trozo, y cuando termina, solicita otro trozo al planificador.
 - El tamaño del trozo se calcula según la siguiente fórmula (se tendrán en cuenta las iteraciones ya asignadas):
$$\text{TamañoTrozo} = \text{IteracionesRestantes} / \text{NúmeroThreads}$$
 - A medida que se van completando trozos, el tamaño de los trozos se irá reduciendo.
 - El tamaño, si no se especifica, por defecto es 1.
 - Si el tamaño se especifica, se decrementa hasta ese tamaño, a excepción del último trozo que puede ser más pequeño que el tamaño.
 - Desventaja:
 - Puede darse un desequilibrio de carga, aunque en menor medida que con static.

OpenMP-MPI - OpenMP

- Planificación – La cláusula *schedule*.
 - guided.
 - Ejemplo.
 - Asignación de las iteraciones de la regla del trapecio 1–10000 utilizando el tipo guiado con 4 threads.

Hilo	Tamaño del trozo	Iteraciones restantes
0	2500	7500
1	1875	5625
2	1406	4219
3	1054	3165
3	791	2374
2	593	1781
1	445	1336
2	334	1002
3	250	752
0	188	564

...

OpenMP-MPI - OpenMP

- Secciones.
 - *#pragma omp sections.*
 - Permite la ejecución de bloques paralelos con códigos diferentes.
 - Se sincronizarán todos los bloques al finalizar.
 - Cada bloque de código irá dentro de un *#pragma omp section.*
- Tareas.
 - *#pragma omp task.*
 - Permite paralelizar recursividad y bucles con un número indeterminado de iteraciones.
 - Es una mejora de las secciones.
 - *#pragma omp taskwait* permite sincronizar todas las tareas creadas por el thread actual.

OpenMP-MPI - OpenMP

- Otras directivas.
 - *#pragma omp ordered* → Indica que el siguiente código de un bucle paralelo debe ejecutarse en serie.
 - *#pragma omp threadprivate(var [, var2...])* → Indica que la lista de variables es privada para un thread. Similar a la cláusula *private*.

OpenMP-MPI - MPI

- Es una especificación de librería de programación paralela por paso de mensajes.
 - Permite la realización de programas paralelos en la que distintos procesos colaboran en un entorno de memoria distribuida y/o compartida.
- Define una sintaxis y semántica diseñadas para la programación de sistemas hardware compuestos por múltiples procesadores en un entorno de memoria distribuida y/o compartida.
- OpenMP trabaja a nivel de hilo o thread, MPI trabaja a nivel de proceso.

OpenMP-MPI - MPI

- Conceptos básicos.
 - Proceso.
 - Secuencia de instrucciones que tiene un espacio de memoria independiente del resto de procesos.
 - Mensaje.
 - Sistema de comunicación entre procesos.
 - Grupo.
 - Conjunto de procesos.
 - Se identifican desde 0 hasta N-1.
 - Comunicador.
 - Objeto que identifica un grupo de procesos.
 - *MPI_COMM_WORLD*
 - Engloba todos los procesos creados a la vez.
 - Si se crean varios grupos, habrá varios comunicadores *MPI_COMM_WORLD*.

OpenMP-MPI - MPI

- Conceptos básicos.
 - Implementación.
 - MPICH v 3.1.3 (<http://www.mpich.org/>)
 - Compilación con MPICH.
 - Debe incluirse en el código `#include <mpi.h>`
 - `mpicc (o mpic++) <codigo_mpi.c (o .cpp)> -o <ejecutable> [-Wall]`
 - Ejecución de un programa MPI.
 - `mpiexec [-n <número de procesos>] [-f hosts] <ejecutable>`
 - Donde *hosts* es un fichero con los diferentes hosts que ejecutarán la implementación.
 - Si no se especifica, únicamente se usará la máquina local.
 - Todos los hosts deben tener el ejecutable.
 - » Si todos los hosts son similares, bastará con copiar el ejecutable.
 - » Si los hosts son diferentes, habrá que compilar el código en cada máquina.

OpenMP-MPI - MPI

- Funciones básicas.

- *int MPI_Init (int *argc, char** argv).*

- Inicializa todo el entorno MPI.
 - *argc* y *argv* son los mismos parámetros que recibe la aplicación.
 - *argc* → número de parámetros, incluido el nombre del ejecutable.
 - *argv* → vector de cadenas de caracteres con todos los parámetros de la aplicación (*argv[0]* es el nombre del ejecutable).
 - No son necesarios desde MPI 1.1.

- *int MPI_Finalize (void).*

- Finaliza el entorno MPI, liberando la memoria y los recursos utilizados.
 - Debemos estar seguros de que todas las funciones MPI hayan terminado.
 - Después de *MPI_Finalize* la única función MPI que puede lanzarse es *MPI_Init*.

OpenMP-MPI - MPI

- Funciones básicas.

- *int MPI_Comm_rank (MPI_Comm comm, int *rank).*
 - Devuelve en *rank* el identificador del proceso dentro del comunicador *comm* especificado.
 - El proceso padre tiene *rank=0*.
- *int MPI_Comm_size (MPI_Comm comm, int *size).*
 - Devuelve en *size* el número de procesos del comunicador *comm*.
- *int MPI_Get_processor_name (char *name, int *strlen).*
 - Devuelve el nombre del procesador del proceso actual.
- *int MPI_Barrier (MPI_Comm comm).*
 - Sincroniza todos los procesos del comunicador *comm*.
 - No suele utilizarse salvo para depuración.
 - La sincronización de procesos suele llevarse a cabo mediante el envío de mensajes síncronos.

OpenMP-MPI - MPI

- Tipos de datos.
 - Lo normal es que en MPI haya comunicación entre los diferentes procesos.
 - Problema.
 - Los procesos pueden estar en máquinas diferentes.
 - Máquinas diferentes pueden usar representaciones de datos diferentes.
 - Esto debe ser transparente al usuario.
 - MPI proporciona descriptores de tipos (MPI_Datatype).

<i>MPI_CHAR</i>	<i>signed char</i>
<i>MPI_SHORT</i>	<i>signed short int</i>
<i>MPI_INT</i>	<i>signed int</i>
<i>MPI_LONG</i>	<i>signed long int</i>
<i>MPI_UNSIGNED_CHAR</i>	<i>unsigned char</i>

<i>MPI_UNSIGNED_SHORT</i>	<i>unsigned short int</i>
<i>MPI_UNSIGNED</i>	<i>unsigned int</i>
<i>MPI_UNSIGNED_LONG</i>	<i>unsigned long int</i>
<i>MPI_DOUBLE</i>	<i>double float</i>

OpenMP-MPI - MPI

- Comunicación punto a punto síncrona.
 - Recepción.
 - *int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status).*
 - Espera la recepción de un mensaje.
 - Los *count* datos de tipo *datatype* se almacenan en *buf* y deben provenir del destinatario *source*, perteneciente al comunicador *comm* y con etiqueta *tag*.
 - *status* es una estructura que almacena información sobre el mensaje.
 - Se utiliza cuando no sabemos quién ha enviado el mensaje ni su etiqueta.
 - Tiene dos campos:
 - » *int MPI_SOURCE* → Indica el emisor del mensaje.
 - » *int MPI_TAG* → Indica la etiqueta del mensaje.

OpenMP-MPI - MPI

- Comunicación punto a punto síncrona.
 - Recepción.
 - *MPI_Recv* esperará siempre hasta que le llegue un mensaje del emisor y con la etiqueta indicados en sus parámetros.
 - Recepción de mensajes sin conocer el destinatario o la etiqueta.
 - Para poder recibir mensajes de cualquier emisor, en el parámetro *source* usaremos la constante *MPI_ANY_SOURCE*.
 - Para poder recibir mensajes con cualquier etiqueta, en el parámetro *tag* usaremos la constante *MPI_ANY_TAG*.
 - Como ya hemos visto, el destinatario y la etiqueta los tenemos en *status*.
 - Para casos en los que conocemos el emisor y la etiqueta, *status* no es necesario, pero debemos incluirlo como parámetro.
 - Si no queremos definir una variable para este caso, podemos usar la constante *MPI_STATUS_IGNORE*.

OpenMP-MPI - MPI

- Comunicación punto a punto síncrona.
 - Envío.
 - Varios tipos de envío de datos síncrono:
 - Envío estándar (*MPI_Send*);
 - Envío síncrono (*MPI_Ssend*);
 - Envío con buffer (*MPI_Bsend*);
 - Envío ready (*MPY_Rsend*);
 - *int MPI_Xsend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm).*
 - X varía para cada función
 - Envía *count* datos de tipo *datatype* almacenados en *buf* al destinatario *dest* con la marca *tag*. *dest* debe estar dentro del comunicador *comm*.
 - En otras palabras, envía al destinatario un array de *count* elementos de tipo *datatype*.

OpenMP-MPI - MPI

- Comunicación punto a punto síncrona.
 - Envío estándar: *MPI_Send*.
 - El emisor puede quedarse bloqueado o no a la espera de la recepción del mismo.
 - Si el tamaño del mensaje a enviar es inferior al tamaño del buffer de comunicación, *MPI_Send* no se bloquea.
 - Si el tamaño es superior, *MPI_Send* se bloquea.
 - Es una combinación entre el envío síncrono y el envío con buffer.
 - Envío síncrono: *MPI_Ssend*.
 - El emisor siempre se queda bloqueado hasta que se inicia la comunicación con el receptor.

OpenMP-MPI - MPI

- Comunicación punto a punto síncrona.
 - Envío con buffer: *MPI_Bsend*.
 - El envío es similar al envío estándar → Los datos a enviar se almacenan en un buffer.
 - Diferencias:
 - El buffer debe ser creado por el programador.
 - Si el mensaje a enviar no cabe en el buffer, se produce un error de ejecución.
 - *MPI_Bsend* nunca se bloquea: o el mensaje cabe en el buffer y se envía, o no cabe y falla la función.
 - El buffer podrá ser utilizado en envíos siguientes siempre que haya espacio suficiente para el nuevo mensaje.
 - Usar *MPI_Bsend* en bucles puede dar problemas.
 - *int MPI_Buffer_attach(void* buffer, int size)* → Crea un buffer de comunicación.
 - *int MPI_Buffer_detach(void* buffer, int* size)* → Libera un buffer de comunicación.

OpenMP-MPI - MPI

- Comunicación punto a punto síncrona.
 - Envío ready: *MPI_Rsend*.
 - El mensaje se envía directamente al destinatario, sin esperar a que esté listo.
 - Se da por hecho que está esperando el mensaje.
 - No se utilizan protocolos de espera ni buffers.
 - Para que el receptor esté listo, debe haber enviado la petición de recepción.
 - En caso de que se envíe el mensaje y el receptor no esté listo, el programa falla.
 - Usar únicamente si se tiene la certeza de que el receptor va a estar en modo recepción.
 - Es una comunicación rápida pero “peligrosa”.

OpenMP-MPI - MPI

- Comunicación punto a punto asíncrona.
 - Comunicación asíncrona → Comunicación no bloqueante.
 - El emisor no espera la recepción del mensaje.
 - El receptor solicita la recepción de un mensaje pero no espera por éste.
 - Ventajas:
 - Mientras el mensaje se transmite, tanto emisor como receptor pueden procesar otros datos.
 - Solapamiento entre comunicación y cómputo.
 - Evita sincronizaciones no necesarias.
 - Evita los interbloqueos
 - El proceso A envía síncronamente al proceso B y el proceso B envía síncronamente al proceso A, quedando ambos a la espera de que el otro reciba los datos.
 - Reduce tiempos ociosos en los procesadores.

OpenMP-MPI - MPI

- Comunicación punto a punto asíncrona.
 - Envío asíncrono.
 - *int MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, **MPI_Request *request**).*
 - Recepción asíncrona.
 - *int MPI_Irecv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, **MPI_Request *request**)*
 - Parámetros:
 - Idénticos a las comunicaciones síncronas.
 - *MPI_Request *request*.
 - Identificador de la comunicación.
 - Se utilizará para "preguntar" si la comunicación ha terminado o no.

OpenMP-MPI - MPI

- Comunicación punto a punto asíncrona.
 - Reglas para comunicación asíncrona.
 - El dato a enviar o recibir no podrá modificarse hasta que la comunicación se lleve a cabo.
 - Que una función de comunicación no bloqueante retorne, no implica ni la emisión ni la recepción de los datos.
 - Hay que implementar mecanismos de comprobación.
 - *int MPI_Wait(MPI_Request *request, MPI_Status *status).*
 - Bloquea al proceso llamador hasta que la comunicación indicada en *request* haya terminado.
 - *int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status).*
 - Comprueba si la comunicación indicada en *request* ha terminado o no.
 - *flag* indica el resultado del test (verdadero o falso).
 - No bloquea al proceso llamador.

OpenMP-MPI - MPI

- Comunicación punto a punto.
 - Recepción por encuesta.
 - Hemos visto cómo recibir mensajes de emisores desconocidos pero, ¿y de tamaños desconocidos?
 - Mediante las encuestas podemos saber si hay mensajes para ser recibidos.
 - Bloqueante → *int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status).*
 - No bloqueante → *int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status).*
 - La encuesta nos dice que hay un mensaje para ser recibido, pero no su longitud.
 - *MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count).*
 - Indica en *count* cuántos datos de tipo *datatype* fueron enviados.
 - Combinado con las encuestas podemos saber cuántos datos se van a recibir.

OpenMP-MPI - MPI

- Comunicación punto a punto.
 - Recepción por encuesta. Ejemplo.

```
int number_amount;
if (world_rank == 0) {
    const int MAX_NUMBERS = 100;
    int numbers[MAX_NUMBERS];
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
    MPI_Status status;
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_INT, &number_amount);
    int* number_buf = (int*)malloc(sizeof(int) * number_amount);
    MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("1 dynamically received %d numbers from 0.\n", number_amount);
    free(number_buf);
}
```

OpenMP-MPI - MPI

- Comunicación colectiva.
 - Permiten el paso de mensajes entre todos los procesos de un comunicador.
 - En MPI 1 y MPI 2 las comunicaciones colectivas son siempre bloqueantes.
 - En MPI 3 se implementan comunicaciones colectivas no bloqueantes.
 - MPI provee las siguientes comunicaciones colectivas:
 - Multidifusión → uno a todos.
 - Reparto → uno a todos.
 - Recolección → todos a uno.
 - Reducción → todos a uno.
 - Reparto entre todos → todos a todos.
 - Recolección entre todos → todos a todos.
 - Reducción entre todos → todos a todos.

OpenMP-MPI - MPI

- Comunicación colectiva.
 - Multidifusión de un mensaje.
 - *int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm).*
 - Envía los *count* datos de tipo *datatype* almacenados en *buf* desde el proceso *root* a todos los demás procesos del comunicador *comm*.
 - La recepción de datos no se realiza con *MPI_Recv*, sino con *MPI_Bcast* indicando en *root* quién es el emisor.

OpenMP-MPI - MPI

- Comunicación colectiva.
 - Reparto.
 - *int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)*
 - Permite repartir de forma automática los datos entre los procesos de un comunicador.
 - Envía *sendcount* datos de tipo *sendtype* de *sendbuf* a cada proceso (cada "trozo" de cada proceso es diferente).
 - Todos los procesos (incluido el *root*) reciben el trozo de tamaño *recvcount* de tipo *recvtype* y lo almacenan en *recvbuf*.

OpenMP-MPI - MPI

- Comunicación colectiva.
 - Recolección.
 - *int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm).*
 - Permite recoger, de forma automática, los fragmentos de datos de los procesos de un comunicador.
 - Todos los procesos (incluido el *root*) envían *sendcount* datos de tipo *sendtype* de *sendbuf* al *root*.
 - El proceso *root* recibe los datos de tamaño *recvcount* de tipo *recvtype* y lo almacenan en *recvbuf*.
 - La recepción se hace por orden estricto de identificador.

OpenMP-MPI - MPI

- Comunicación colectiva.

- Reducción.

- *int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm).*
 - Realiza la operación *op* sobre los *count* datos de *sendbuf* distribuidos en todos los procesos del comunicador *comm* y almacena el resultado en *recvbuf* en el proceso *root*.

- Operaciones de reducción predefinidas:

MPI_SUM	Suma	MPI_PROD	Producto	MPI_LXOR	XOR lógico
MPI_MAX	Máximo	MPI_MIN	Mínimo	MPI_BXOR	XOR a nivel de bit
MPI_LAND	AND lógico	MPI_BAND	AND a nivel de bit	MPI_MAXLOC	Máximo y posición
MPI_LOR	OR lógico	MPI BOR	OR a nivel de bit	MPI_MINLOC	Mínimo y posición

- Operaciones de reducción definidas por el usuario:

- *int MPI_Op_create(MPI_User_function *f, int commute, MPI_Op *op);*
 - Crea un nuevo operador de reducción *op* con la operación *f*, la cual debe cumplir la propiedad asociativa.
 - *commute* indica si *f* cumple la propiedad conmutativa.
 - *int MPI_Op_free(MPI_Op *op) →* Libera el operador.

OpenMP-MPI - MPI

- Comunicación colectiva.

- Reparto entre todos.

- int MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm).*
 - Reparte los datos de *sendbuf* de todos los procesos a todos los procesos agrupados en grupos de tamaño *sendcount*.
 - Básicamente, todos los procesos realizan un *MPI_Scatter*.
 - Ejemplo: *MPI_Alltoall(u, 2, MPI_INT, v, 2, MPI_INT, MPI_WORLD_COMM);*

u								pid	v							
10	11	12	13	14	15	16	17	0	10	11	20	21	30	31	40	41
20	21	22	23	24	25	26	27	1	12	13	22	23	32	33	42	43
30	31	32	33	34	35	36	37	2	14	15	24	25	34	35	44	45
40	41	42	43	44	45	46	47	3	16	17	26	27	36	37	46	47

OpenMP-MPI - MPI

- Comunicación colectiva.
 - Recolección entre todos.
 - *int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm).*
 - Recolecta los datos de *sendbuf* de todos los procesos a todos los procesos agrupados en grupos de tamaño *sendcount*.
 - Básicamente, todos los procesos realizan un *MPI_Gather*.
 - Ejemplo: *MPI_Allgather (u, 2, MPI_INT, v, 2, MPI_INT, MPI_WORLD_COMM);*

u		pid	v							
10	11	0	10	11	20	21	30	31	40	41
20	21	1	10	11	20	21	30	31	40	41
30	31	2	10	11	20	21	30	31	40	41
40	41	3	10	11	20	21	30	31	40	41

OpenMP-MPI - MPI

- Comunicación colectiva.
 - Reducción entre todos.
 - *int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm).*
 - Reduce los valores de todos los procesos y distribuye el resultado entre todos los procesos.
 - Ejemplo: *MPI_Allreduce(u, &v, 8, MPI_INT, MPI_SUM, MPI_COMM_WORLD);*

u								pid	v
10	11	12	13	14	15	16	17	0	912
20	21	22	23	24	25	26	27	1	912
30	31	32	33	34	35	36	37	2	912
40	41	42	43	44	45	46	47	3	912

OpenMP-MPI - MPI

- Tipos de datos derivados.
 - MPI maneja en todas sus funciones de envío/recepción vectores de tipos simples.
 - En general, se asume que los elementos de esos vectores están almacenados consecutivamente en memoria.
 - En ocasiones, sin embargo, es necesario el intercambio de tipos estructurados (*structs*), o de vectores no almacenados consecutivamente en memoria (envío de una columna de un array).
 - MPI incluye la posibilidad de definir tipos más complejos (objetos del tipo *MPI_Datatype*).
 - *int MPI_Type_commit(MPI_Datatype *datatype)* → Crea el nuevo tipo de datos.
 - *int MPI_Type_free(MPI_Datatype *datatype)* → Destruye el nuevo tipo de datos.

OpenMP-MPI - MPI

- Tipos de datos derivados.
 - Definición de tipos homogéneos.
 - Todos los elementos constituyentes son del mismo tipo.
 - *int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)*
 - Crea un nuevo tipo de datos *newtype* formado por *count* elementos de tipo *oldtype*.
 - Los datos estarán consecutivos en memoria.
 - *int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype).*
 - Crea un nuevo tipo de datos *newtype* formado por *count* bloques de *blocklength* elementos de tipo *oldtype*.
 - Los bloques estarán separados en memoria *stride* elementos de tipo *oldtype*.
 - *MPI_Type_contiguous(c, o, n)* es equivalente a *MPI_Type_vector(c, 1, 1, o, n)* o a *MPI_Type_vector(1, c, c, o, n)*.

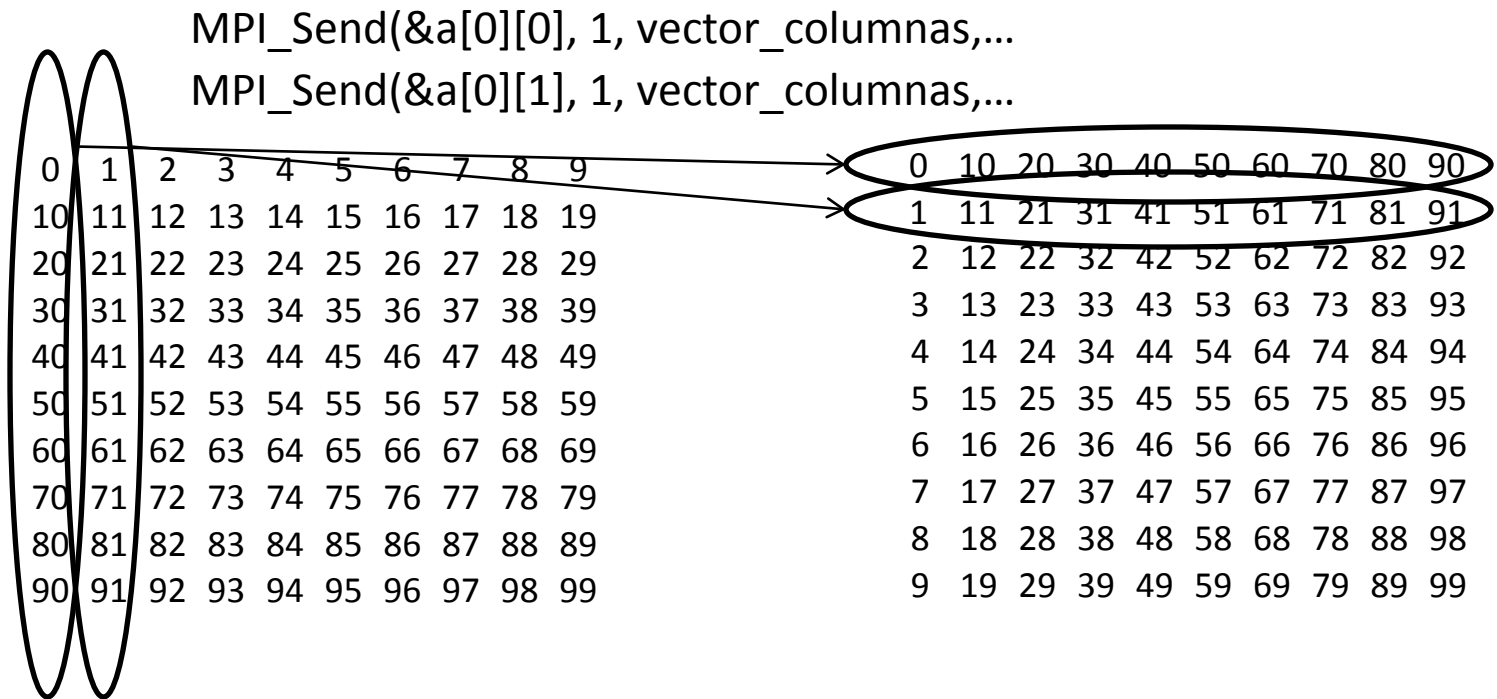
OpenMP-MPI - MPI

- Tipos de datos derivados.
 - Definición de tipos homogéneos. Ejemplo *MPI_Type_vector*.
 - Ejecutaremos el código con 11 procesos.

```
int a[10][10] = {{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, ..., {90, 91, 92, 93, 94, 95, 96, 97, 98, 99}};
if (rank!=0) {
    MPI_Datatype vector_columnas;
    MPI_Type_vector (10, 1, 10, MPI_INT, &vector_columnas);
    MPI_Type_commit(&vector_columnas);
    MPI_Send(&a[0][rank-1], 1, vector_columnas, 0, 0, MPI_COMM_WORLD);
    MPI_Type_free(&vector_columnas);
}
else {
    int b[10][10];
    MPI_Status status;
    for (i=0; i<10; i++) {
        MPI_Recv(&b[i], 10, MPI_INT, i+1, 0, MPI_COMM_WORLD, &status);
    }
}
MPI_Finalize();
```

OpenMP-MPI - MPI

- Tipos de datos derivados.
 - Definición de tipos homogéneos. Ejemplo MPI_Type_vector.
 - MPI_Type_vector (10, 1, 10, MPI_INT, &vector_columns);

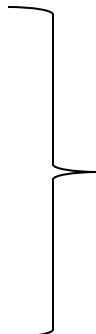


OpenMP-MPI - MPI

- Tipos de datos derivados.
 - Definición de tipos homogéneos. Ejercicio.
 - Dadas las instrucciones del código anterior
 - *MPI_Type_vector (x, y, z, MPI_INT, &vector_columns);*
 - *MPI_Send(&a[i][j], 1, vector_columns, 0, 0, MPI_COMM_WORLD);*
 - ¿Qué valores deben tener *x*, *y*, *z*, *i* y *j* para que los datos se envíen
 - en 5 grupos de dos columnas?
 - en 1 grupo con los pares y otro con los impares?
 - en 1 grupo con la diagonal \ y otro con la diagonal /?

OpenMP-MPI - MPI

- Tipos de datos derivados.
 - Definición de tipos heterogéneos.
 - Los elementos constituyentes son de tipos diferentes.
 - *MPI_Type_struct* (*int count*, *int *array_of_blocklengths*, *MPI_Aint *array_of_displacements*, *MPI_Datatype *array_of_types*, *MPI_Datatype *newtype*).
 - *count* → Número de bloques y tamaño de los arrays *array_of_blocklengths*, *array_of_displacements* y *array_of_types*.
 - Cada bloque *i* tiene *array_of_blocklengths[i]* elementos, está desplazado *array_of_displacements[i]* bytes del anterior, y sus elementos son de tipo *array_of_types[i]*.
 - Ejemplo:
 - B = {2, 1, 3}
 - D = {0, 8, 12}
 - T = {MPI_FLOAT, MPI_INT, MPI_CHAR}
 - MPI_Type_struct(3, B, D, T, newtype)



Desplazamiento	Tipo
0	float
4	float
8	int
12	char
13	char
14	char

OpenMP-MPI - MPI

- Creación dinámica de procesos.
 - Un proceso MPI puede crear nuevos procesos MPI.
 - Esto implica que el número de procesos de un programa MPI puede variar dinámicamente.
 - Se crea así una jerarquía entre padres e hijos.
 - Entre un proceso padre y sus procesos hijos se establece un comunicador que permitirá la comunicación entre ellos.
 - No se puede usar para comunicación entre hijos.
 - La creación dinámica de procesos se fundamenta en tres funciones.
 - *MPI_Comm_spawn* → Crea procesos hijos con el mismo programa.
 - *MPI_Comm_spawn_multiple* → Crea procesos hijos con diferentes programas.
 - *MPI_Comm_get_parent* → Permite a los hijos obtener el comunicador con su padre.

OpenMP-MPI - MPI

- Creación dinámica de procesos.
 - *MPI_Comm_spawn* (*char *programm, char **argv, int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int errors[]*)
 - *programm*: programa que ejecutarán los procesos hijos.
 - *argv*: parámetros de entrada del programa a ejecutar por los hijos.
 - *maxprocs*: número máximo de procesos a crear.
 - *info*: información sobre cómo y dónde ejecutar los procesos. Usaremos *MPI_INFO_NULL*.
 - *root*: proceso que provee los valores de los parámetros anteriores.
 - *comm*: comunicador del grupo de procesos padres.
 - *MPI_COMM_SELF* → Comunicador con un único proceso, el actual.
 - *intercomm*: intercomunicador entre el grupo *comm* y los hijos.
 - *errors*: array de códigos de error, uno por proceso.

OpenMP-MPI - MPI

- Creación dinámica de procesos.
 - *int MPI_Comm_spawn_multiple (int count, char *array_of_commands[], char **array_of_argv[], int array_of_maxprocs[], MPI_Info array_of_info[], int root, MPI_Comm comm, MPI_Comm *intercomm, int errors[])*
 - *Count*: número de programas en *array_of_commands*.
 - *array_of_commands*: lista de programas que ejecutarán los hijos.
 - *array_of_argv*: parámetros de entrada a cada uno de los programas.
 - *array_of_maxprocs*: número máximo de procesos que se crearán con cada programa.
 - *array_of_info*: información sobre cómo y dónde ejecutar los procesos de cada programa. Usaremos *MPI_INFO_NULL*.
 - *Root, comm, intercomm* y *errors*: igual que para *MPI_Comm_spawn*.

OpenMP-MPI - MPI

- Creación dinámica de procesos.
 - *int MPI_Comm_get_parent(MPI Comm *parent);*
 - *Parent*: comunicador entre el padre y todos los hijos.
 - En caso de que haya habido algún error, el nuevo comunicador será *MPI_COMM_NULL* (a parte del posible código de error de la función).
 - *MPI_COMM_WORLD* agrupa a todos los hijos, pero no al padre.
 - Proceso de creación dinámica de hilos:
 - El padre realiza la llamada *MPI_Comm_Spawn* o *MPI_Comm_spawn_multiple* para crear N hijos.
 - Se crea un comunicador entre el padre y los hijos.
 - El padre lo obtiene al ejecutar *MPI_Comm_Spawn* o *MPI_Comm_spawn_multiple*.
 - Los hijos realizan la llamada *MPI_Init*.
 - Crean su propio comunicador *MPI_COMM_WORLD*.
 - Los hijos obtienen el comunicador con el padre mediante la función *MPI_Comm_get_Parent*.

OpenMP-MPI - MPI

- Creación dinámica de procesos. Ejemplo
 - Proceso padre.

```
#include <mpi.h> #include <stdio.h> #include <string.h>
int main(int argc, char *argv[]) {
    int size, myrank, numProcSpw;
    MPI_Comm intercomm; /* comunicador */
    char cadena[100]; int ack, length, i;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("Numero de hijos que desea a crear: "); scanf("%d",&numProcSpw);

    //El ejecutable de los hijos se le pasa al padre como primer parámetro
    MPI_Comm_spawn(argv[1], MPI_ARGV_NULL, numProcSpw, MPI_INFO_NULL, 0,
                   MPI_COMM_SELF, &intercomm, MPI_ERRCODES_IGNORE);
    for(i=0; i<numProcSpw; i++) {
        MPI_Recv(&length, 1, MPI_INT, i, 0, intercomm, MPI_STATUS_IGNORE);
        MPI_Recv(cadena, length, MPI_CHAR, i, 0, intercomm, MPI_STATUS_IGNORE);
        printf("Padre recibiendo cadena %s del hijo %d\n", cadena, i);
        MPI_Send(&ack, 1, MPI_INT, i, 0, intercomm);
    }
    MPI_Finalize();
    return 0;
}
```

OpenMP-MPI - MPI

- Creación dinámica de procesos. Ejemplo
 - Proceso hijo.

```
#include <mpi.h> #include <stdio.h> #include <string.h>
int main(int argc, char *argv[]) {
    int ack, length;
    char cadena[100];
    strcpy(cadena, "HOLA");
    length=strlen(cadena);
    length++;

    MPI_Comm parent;
    MPI_Init(&argc, &argv);
    MPI_Comm_get_parent(&parent);
    if (parent == MPI_COMM_NULL) {
        printf("No existe Padre creado.\n");
        return 0;
    }
    printf("Enviando HOLA al proceso padre\n");
    MPI_Send(&length, 1, MPI_INT, 0, 0, parent);
    MPI_Send(cadena, length, MPI_CHAR, 0, 0, parent);
    MPI_Recv(&ack, 1, MPI_INT, 0, 0, parent, MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
}
```

OpenMP-MPI - Hibridación

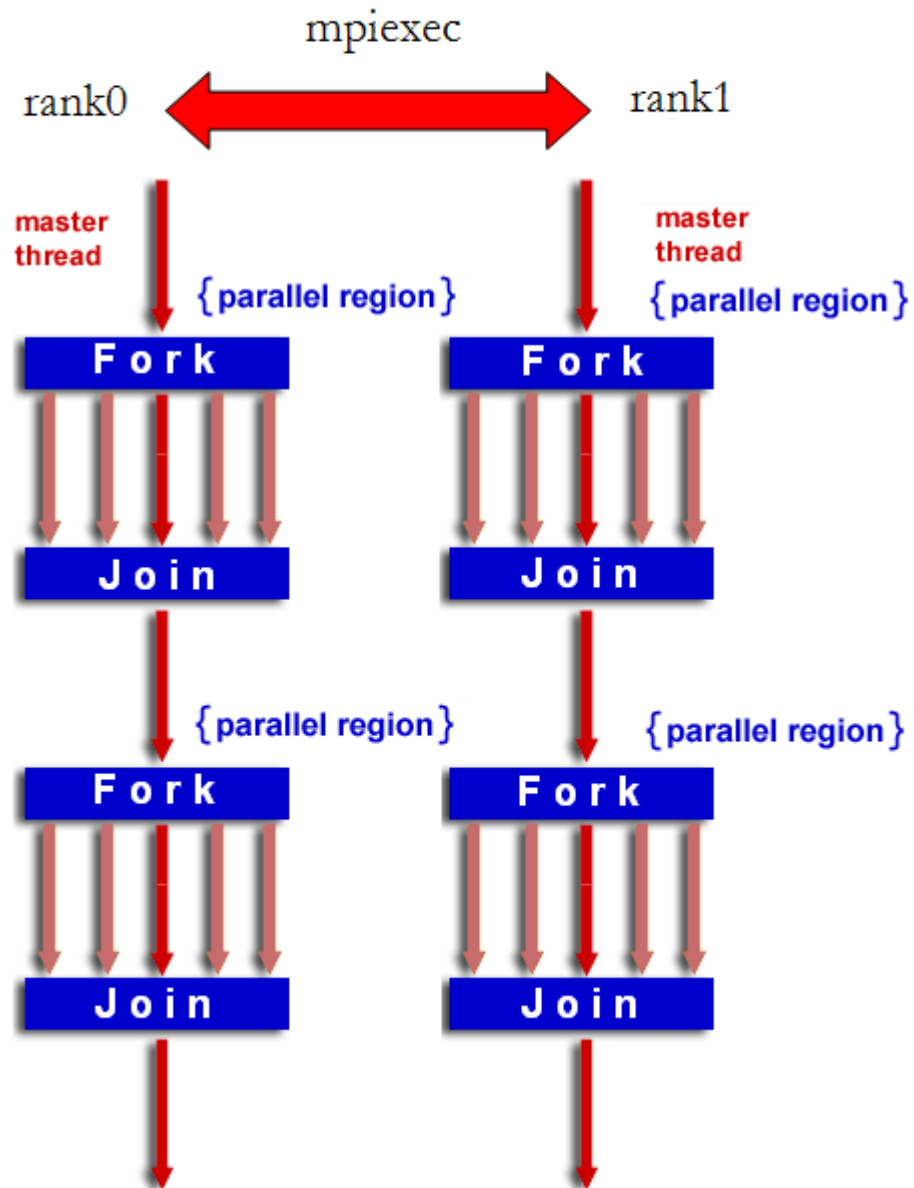
- Definición.
 - Paradigma de programación en el que se combinan los paradigmas de memoria compartida (intra-nodo) y memoria distribuida (inter-nodo).
- Ventajas.
 - Mejoran la escalabilidad del sistema.
 - Incorporan mecanismos de balanceo de carga y no se ven afectados por comunicaciones a nivel intra-nodo.
 - Disminuyen la latencia al reducir el número de mensajes pequeños enviados.
 - Apropiaada para aplicaciones que combinan paralelismo de grano grueso y grano fino.

OpenMP-MPI - Hibridación

- Modelos MPI puros:
 - Inicializar $N \times M$ procesos MPI, donde N es el número de nodos y M el número de cores de cada nodo.
 - Problemas:
 - Balanceo de carga → Sin mecanismos de scheduling.
 - Overhead por comunicaciones y sincronizaciones en memoria compartida.
 - Replicación de datos.
 - Latencias cuando el número de mensajes es elevado.
- Modelos OpenMP puros:
 - Solo a nivel intra-nodo.
 - En el pasado se han realizado propuestas para simular entornos de memoria compartida en multicomputadores:
 - Cluster OpenMP (Intel).

OpenMP-MPI - Hibridación

- Surge una cuestión:
 - ¿Cómo combinarlos para maximizar el rendimiento?
- Pasos a seguir:
 - Se ponen en marcha N procesos MPI que se asignan a distintos nodos en memoria distribuida.
 - Normalmente se lanzará un proceso por nodo.
 - Cada proceso genera, mediante OpenMP, un pool de hilos para realizar tareas en memoria compartida.



OpenMP-MPI - Hibridación

- Esquemas híbridos.
 - Masteronly.
 - El modelo híbrido masteronly usa un único proceso MPI por nodo compuesto por varios hilos OpenMP asignados a cada uno de sus cores.
 - Un único hilo (master) gestionará el paso de mensajes.
 - Por lo tanto, las llamadas a funciones MPI tienen lugar fuera de la región paralela OpenMP (o bien en secciones single/master).

```
if (my_thread_ID == 0) /* Solo el hilo maestro procesa las funciones MPI */  
    MPI_Recv (recepción de datos de otros nodos)  
#pragma omp parallel  
{  
    /* código a paralelizar con OpenMP*/  
    /* Todos los hilos: Ejecutar tareas paralelas */  
}  
if (my_thread_ID == 0) /* Solo el hilo maestro procesa las funciones MPI */  
    MPI_Send (envío de datos a otros nodos)
```

OpenMP-MPI - Hibridación

- Esquemas híbridos.
 - Híbrido con solapamiento.
 - Asigna distintos roles a los hilos OpenMP:
 - Hilos de comunicación: gestionan el paso de mensajes mediante MPI.
 - Hilos de cómputo: realizan tareas de cálculo en paralelo a las comunicaciones.

```
#pragma omp parallel{  
  if (my_thread_ID < ...) { /* Hilos de comunicación: */  
    /* Gestión de paso de mensajes*/  
    MPI_Send (datos a enviar)  
    MPI_Recv (datos a recibir)  
  } else { /* Hilos de cómputo: */  
    /* Ejecutar tareas independientes de la comunicación */  
  }  
  /* Todos los hilos: */  
  /* Ejecutar tareas con los datos recibidos */  
}
```

OpenMP-MPI - Hibridación

- Interacción OpenMP-MPI.
 - Las implementaciones de MPI ofrecen distintos grados de interacción con OpenMP.
 - *MPI_THREAD_SINGLE*: solo se permite un thread de ejecución en el proceso.
 - *MPI_THREAD_FUNNELED*: se permiten múltiples threads OpenMP en el proceso, pero sólo el hilo maestro podrá realizar llamadas a funciones MPI.
 - *MPI_THREAD_SERIALIZED*: se permiten múltiples threads OpenMP en el proceso, y múltiples threads podrán realizar llamadas a funciones MPI, pero solo uno en cada periodo de tiempo (llamadas MPI no concurrentes).
 - *MPI_THREAD_MULTIPLE*: como el anterior, pero sin restricciones (llamadas MPI concurrentes).

OpenMP-MPI - Hibridación

- Interacción OpenMP-MPI.

- Funciones MPI de gestión de hilos:

- *int MPI_Init_thread(int *argc, char **argv[]), int required, int *provided).*
 - Se usa en lugar de MPI_Init para indicar el grado de interacción OpenMP-MPI.
 - Si se usa MPI_Init, el grado de interacción OpenMP-MPI vendrá dado por la implementación.
 - *required*: grado de interacción solicitado.
 - *provided*: grado de interacción obtenido.
 - *int MPI_Is_thread_main (int *flag).*
 - Indica si el hilo actual es el hilo maestro.
 - *flag*: devuelve true si el hilo actual invocó a *MPI_Init* o *MPI_Init_thread*.
 - *int MPI_Query_thread (int *provided).*
 - Sirve para comprobar el grado de interacción.
 - *provided*: grado de interacción OpenMP-MPI establecido. Debe tener el mismo valor que el devuelto por *MPI_Init_thread*.

OpenMP-MPI - Hibridación

- Ejemplo. La regla del trapecio.

$$\int_a^b f(x)dx \sim \frac{b-a}{n} \left(\frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k \frac{b-a}{n}\right) \right)$$

Código secuencial

```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```

Código OpenMP

```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
#pragma omp parallel for num_threads(thread_count) reduction(+: approx)  
    for (i = 1; i <= n-1; i++)  
        approx += f(a + i*h);  
approx = h*approx;
```

OpenMP-MPI - Hibridación

- Ejemplo. La regla del trapecio.

$$\int_a^b f(x)dx \sim \frac{b-a}{n} \left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k \frac{b-a}{n}\right) \right)$$

Código MPI

```
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
h=(b-a)/n;
local_n=n/size;
local_a=a+my_rank*local_n;
local_b=local_a+local_n;

approx=(f(local_a)+f(local_b))/2.0;
for (i=1; i<=local_n; i++)
    approx += f(local_a + i*h);
approx *=h;
```

```
if (my_rank==0) {
    total=approx;
    for(source=1; source<size; source++) {
        MPI_Recv(&approx, 1, MPI_FLOAT, source, tag,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total+=approx;
    }
} else
    MPI_Send(&approx, 1, MPI_FLOAT, source, tag,
             MPI_COMM_WORLD);

MPI_Finalize();
```


OpenMP-MPI - Hibridación

- Ejemplo. La regla del trapecio.

$$\int_a^b f(x)dx \sim \frac{b-a}{n} \left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k \frac{b-a}{n}\right) \right)$$

Código híbrido OpenMP-MPI

```
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
h=(b-a)/n;
local_n=n/size;
local_a=a+my_rank*local_n;
local_b=local_a+local_n;

approx=(f(local_a)+f(local_b))/2.0;
#pragma omp parallel for num_threads(thread_count) \
    reduction(+: approx)
    for (i = 1; i <= n-1; i++)
        approx += f(local_a + i*h);
approx = h*approx;
```

```
if (my_rank==0) {
    total=approx;
    for(source=1; source<size; source++) {
        MPI_Recv(&approx, 1, MPI_FLOAT, source, tag,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total+=approx;
    }
} else
    MPI_Send(&approx, 1, MPI_FLOAT, source, tag,
        MPI_COMM_WORLD);

MPI_Finalize();
```

Bibliografía

- J. Aguilar, E. Leiss: “Introducción a la computación paralela”.
- P. Pacheco: “An introduction to parallel programming”.
- J. M. Mantas Ruiz: “Metodología de diseño de algoritmos paralelos”.
- P. Pacheco: “An introduction to parallel programming”.
- OpenMP Specification 4.0. <http://openmp.org/wp/openmp-specifications/>
- MPI: A Message-Passing Interface Standard Version 3.0. MPI Forum. <http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>
- José M. Alonso: “Programación de aplicaciones paralelas con MPI (Message Passing Interface)”.
- MPICH: High-Performance Portable MPI.
<http://www.mpich.org/>