Concurrencia y Paralelismo

Clase 12

Facultad de Informática

UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Arquitecturas Paralelas:
https://drive.google.com/u/0/uc?id=1MPwHy-Q4v19JBJ14B0XYgMRQcM9UXCMP&export=download

- Diseño de Algoritmos Paralelos:
https://drive.google.com/u/0/uc?id=1ao5Yzh2EbxG4hNgVfw0vGG1oneJdcwYA&export=download

- Métricas de Rendimiento:
https://drive.google.com/uc?id=1b7DRKwCaOx0mH9sEzmWeGroW-VluJcv7&export=download

- Paradigmas de Programación Paralela:
https://drive.google.com/uc?id=15zyZYIL5VWxU-JxxOwjMdC7LhmkchNKv&export=download

Arquitecturas Paralelas

Clasificación de arquitecturas paralelas

Hay diferentes enfoques para clasificar las arquitecturas paralelas:

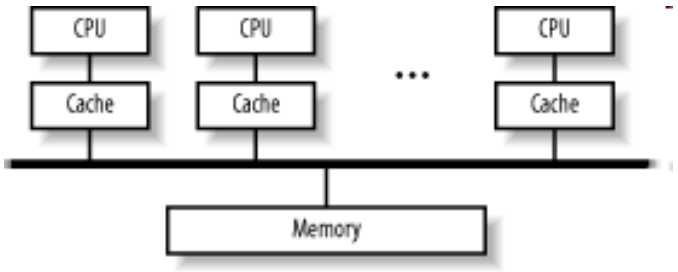
- Por la organización del espacio de direcciones.
- Por la granularidad.
- Por el mecanismo de control.
- Por la red de interconexión.

Clasificación por el Espacio de Direcciones

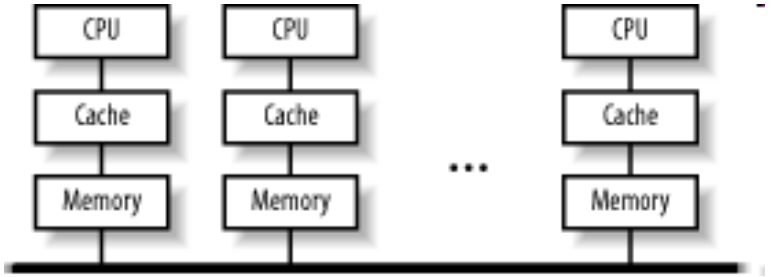
- Las arquitecturas paralelas se clasifican según su espacio de direcciones en:
  - Memoria Compartida.
  - Memoria Distribuida.
- Esta clasificación se relaciona con el modelo de comunicación a utilizar:
  - Accesos a Memoria Compartida (memoria compartida).
  - Intercambio de mensajes (principalmente memoria distribuida).
- En algunos casos también tenemos en la misma plataforma ambos mecanismos.

Clasificación por el Espacio de Direcciones

- Multiprocesadores de memoria compartida.
  - Interacción modificando datos en la memoria compartida.
  - Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos). Problemas de sincronización y consistencia.
  - Esquemas NUMA para mayor número de procesadores distribuidos.
  - Problema de consistencia.



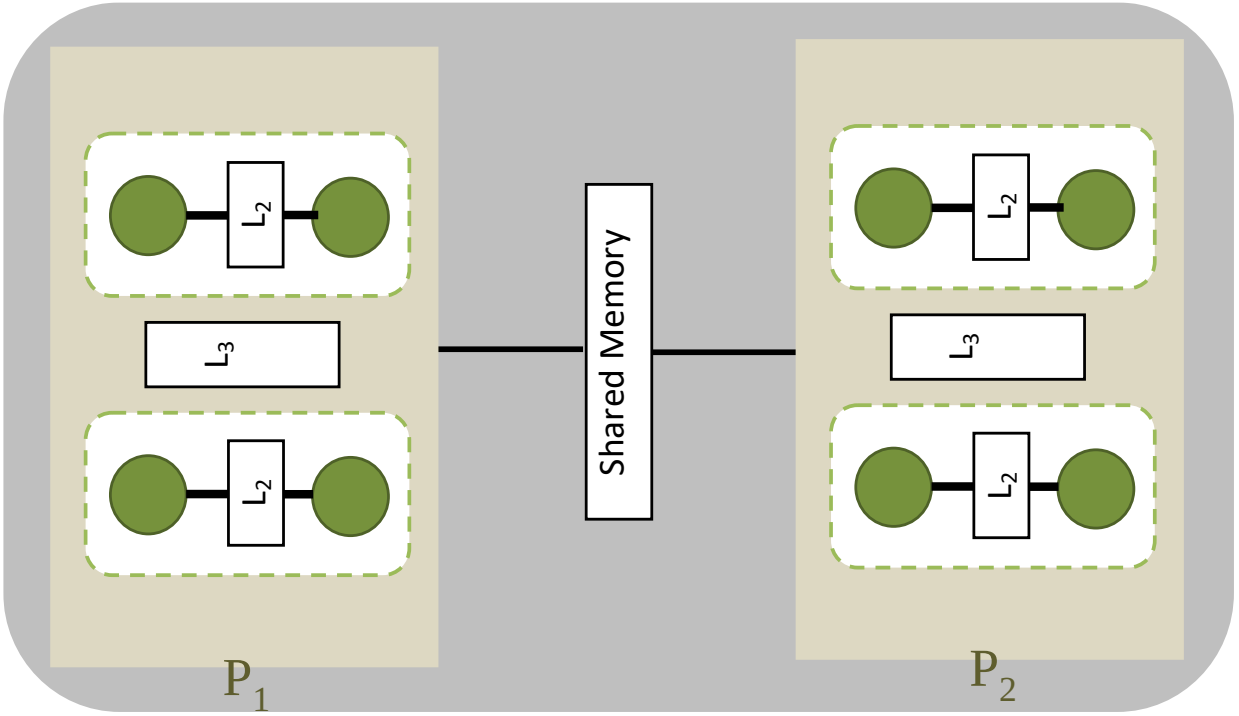
Esquema UMA



Esquema NUMA

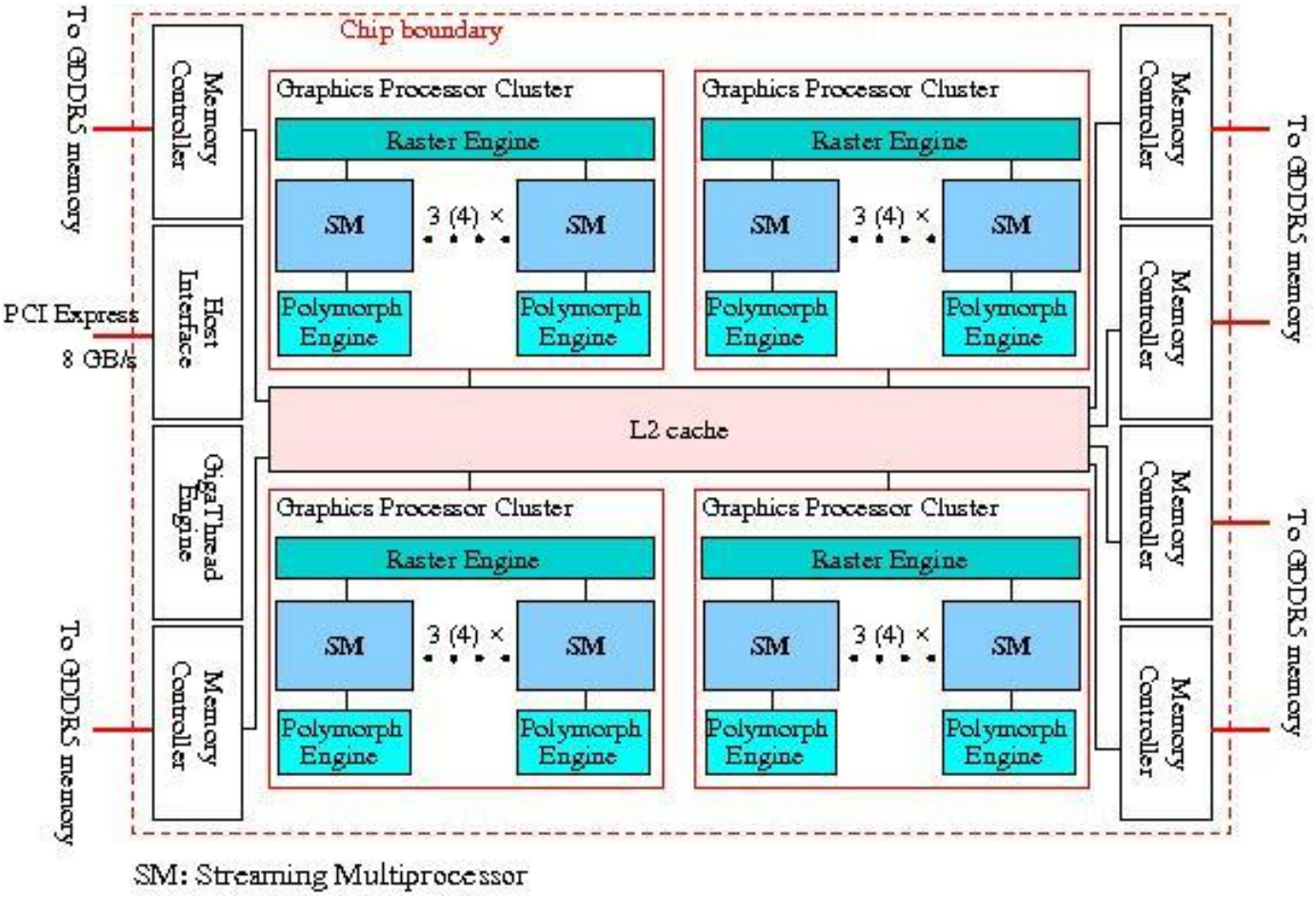
Clasificación por el Espacio de Direcciones

- Ejemplo de multiprocesador de memoria compartida: multicore de 8 núcleos.



Clasificación por el Espacio de Direcciones

- Ejemplo de multiprocesador de memoria compartida: GPU.



Clasificación por el Espacio de Direcciones

- Multiprocesadores con memoria distribuida.
  - Procesadores conectados por una red.
  - Memoria local (no hay problemas de consistencia).
  - Interacción es sólo por pasaje de mensajes.
  - Grado de acoplamiento de los procesadores:
    - Multicomputadores (máquinas fuertemente acopladas). Procesadores y red físicamente cerca. Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores. Alto ancho de banda y velocidad.
    - Memoria compartida distribuida.
    - Clusters.
    - Redes (multiprocesador débilmente acoplado).

Clasificación de arquitecturas paralelas

Hay diferentes enfoques para clasificar las arquitecturas paralelas:

- Por la organización del espacio de direcciones.
- Por la granularidad.
- Por el mecanismo de control.
- Por la red de interconexión.

Clasificación por el Mecanismo de Control

Propuesta por Flynn ("Some computer organizations and their effectiveness", 1972).

Se basa en la manera en que las ***instrucciones*** son ejecutadas sobre los ***datos***.

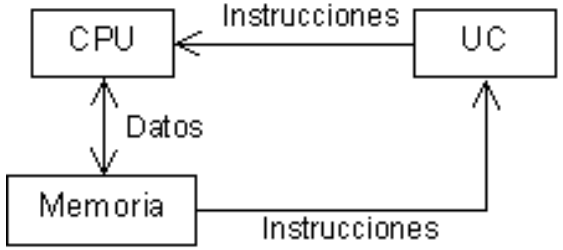
Clasifica las arquitecturas en 4 clases:

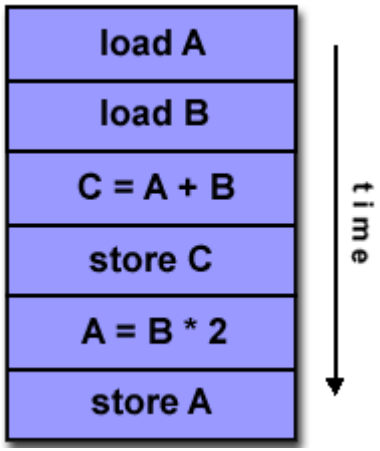
- SISD (Single Instruction Single Data).
- SIMD (Single Instruction Multiple Data).
- MISD (Multiple Instruction Single Data).
- MIMD (Multiple Instruction Multiple Data).

Clasificación por el Mecanismo de Control

SISD: Single Instruction Single Data

- Instrucciones ejecutadas en secuencia, una por ciclo de instrucción.
- La memoria afectada es usada sólo por ésta instrucción.
- Usada por la mayoría de los uní procesadores.
- La CPU ejecuta instrucciones (decodificadas por la UC) sobre los datos. La memoria recibe y almacena datos en las escrituras, y brinda datos en las lecturas.
- Ejecución determinística.

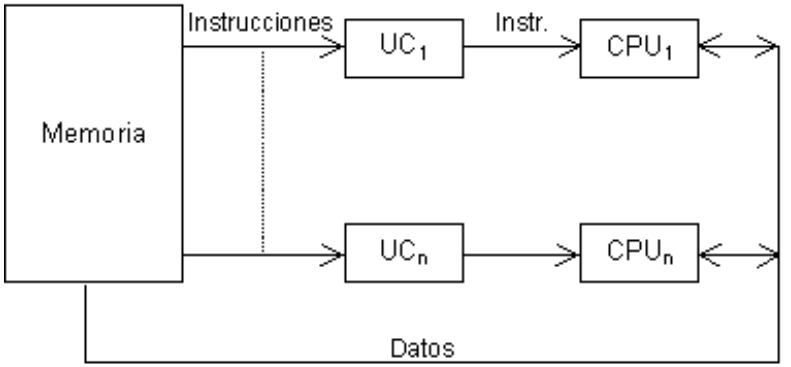




Clasificación por el Mecanismo de Control

MISD: Multiple Instruction Single Data

- Los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes.
- Operación sincrónica (en lockstep).
- No son máquinas de propósito general (“hipotéticas”, Duncan).
- Ejemplos posibles:
  - Múltiples filtros de frecuencia operando sobre una única señal.
  - Múltiples algoritmos de criptografía intentando crackear un único mensaje codificado.



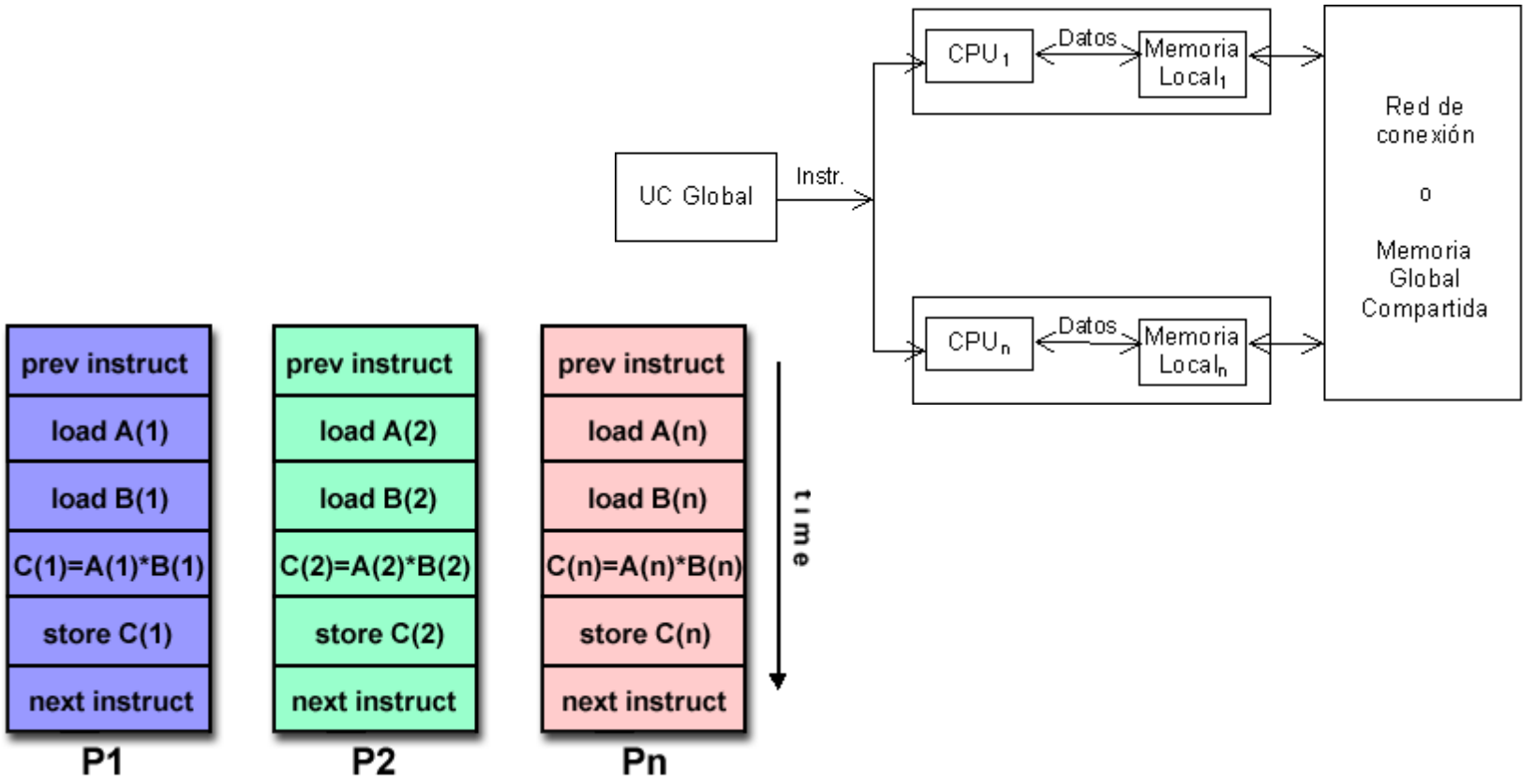
Clasificación por el Mecanismo de Control

SIMD: Single Instruction Multiple Data

- Conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos.
- Los procesadores en general son muy simples.
- El host hace broadcast de la instrucción. Ejecución sincrónica y determinística.
- Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones.
- Adecuados para aplicaciones con alto grado de regularidad, (por ejemplo procesamiento de imágenes).

Clasificación por el Mecanismo de Control

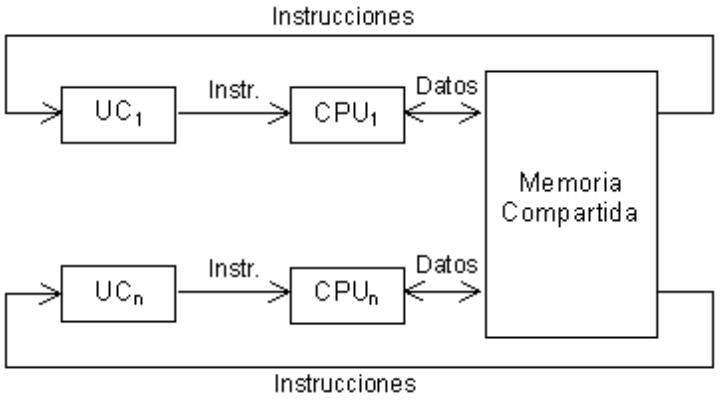
- **Ejemplos de máquina SIMD:** Array Processors. CM-2, Maspar MP-1 y 2, Illiac IV.

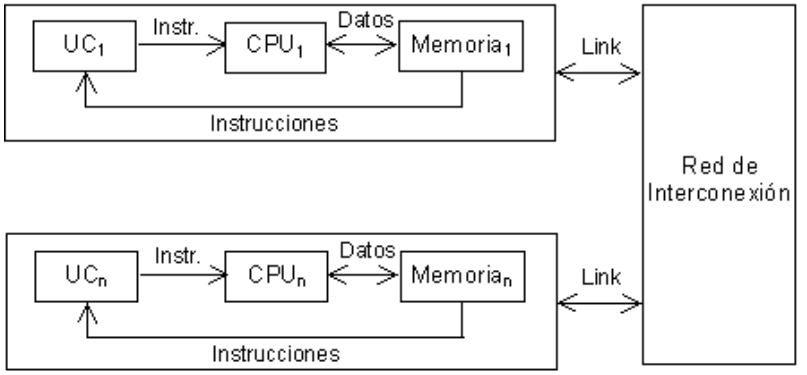


Clasificación por el Mecanismo de Control

MIMD: Multiple Instruction Multiple Data

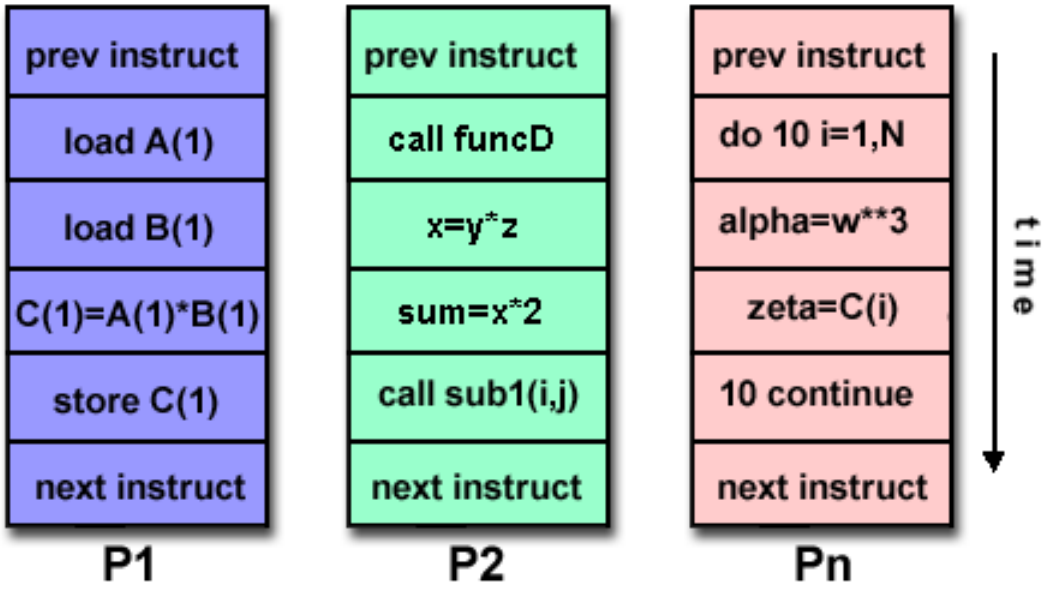
- Cada procesador tiene su propio flujo de instrucciones y de datos ⇒ cada uno ejecuta su propio "programa" a su ritmo.
- Pueden ser con memoria compartida o distribuida.
- Sub-clasificación de MIMD:
  - MPMD (multiple program multiple data): cada procesador ejecuta su propio programa (ejemplo con PVM).
  - SPMD (single program multiple data): hay un único programa fuente y cada procesador ejecuta su copia independientemente (ejemplo con MPI).





Clasificación por el Mecanismo de Control

- **Ejemplos de máquina MIMD:** nCube 2, iPSC, CM-5, Paragon XP/S, máquinas DataFlow, red de transputers, multicores, cluster.



Clasificación de arquitecturas paralelas

Hay diferentes enfoques para clasificar las arquitecturas paralelas:

- Por la organización del espacio de direcciones.
- Por la granularidad.
- Por el mecanismo de control.
- Por la red de interconexión.

Clasificación por la Red de Interconexión

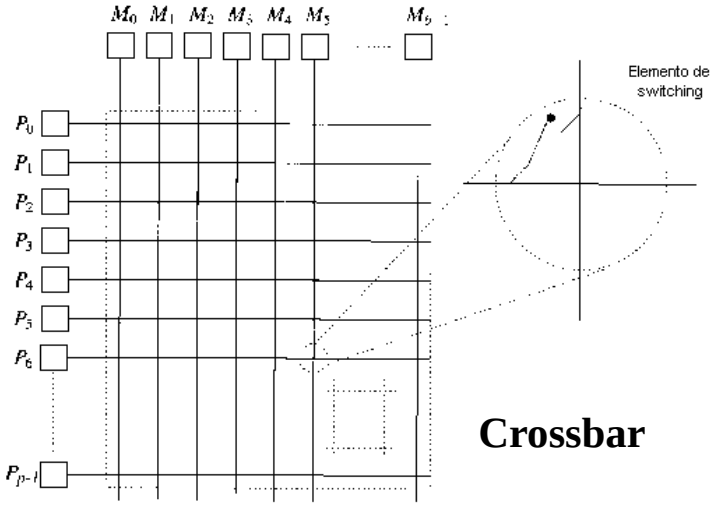
Tanto en memoria compartida como en pasaje de mensajes las máquinas pueden construirse conectando procesadores y memorias usando diversas redes de interconexión:

- Las ***redes estáticas*** constan de links punto a punto. Típicamente se usan para máquinas de pasaje de mensajes.
- Las ***redes dinámicas*** están construidas usando switches y enlaces de comunicación. Normalmente para máquinas de memoria compartida.

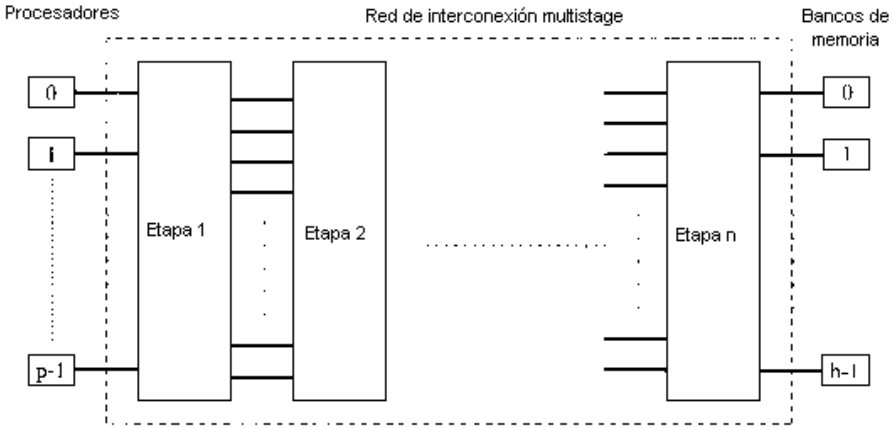
El diseño de la red de interconexión depende de una serie de factores (ancho de banda, tiempo de startup, paths estáticos o dinámicos, operación sincrónica o asincrónica, topología, costo, etc.).

Clasificación por la Red de Interconexión

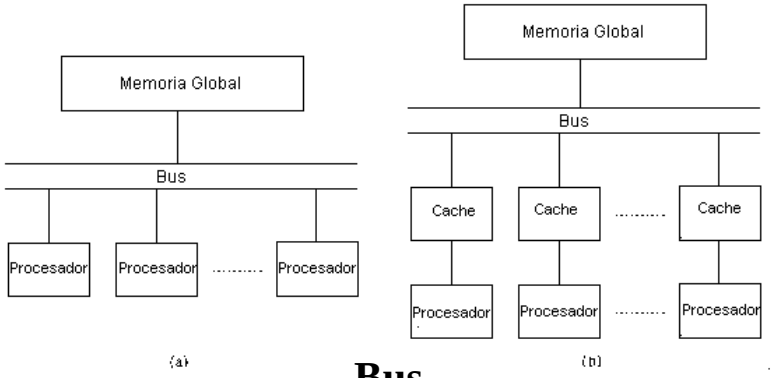
Redes de interconexión dinámicas



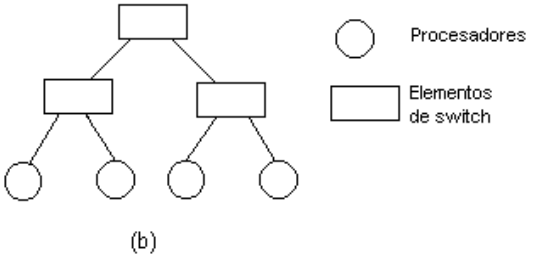
Crossbar



Multistage



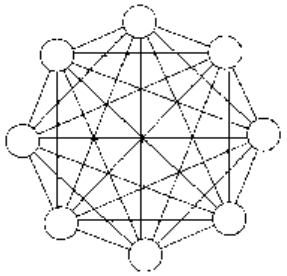
Bus



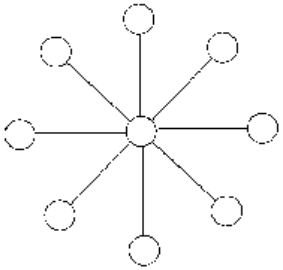
Árbol dinámico

Clasificación por la Red de Interconexión

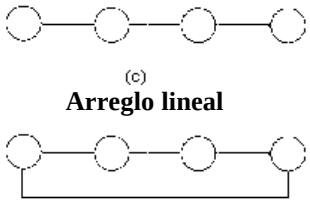
Redes de interconexión estáticas



(a)
Completamente conectada

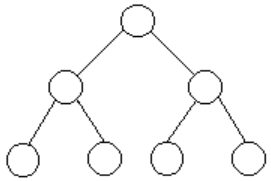


(b)
Estrella

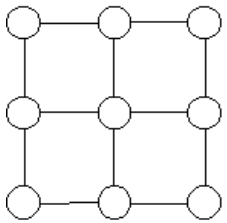


(c)
Arreglo lineal

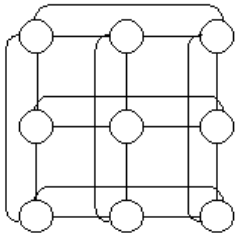
(d)
Anillo



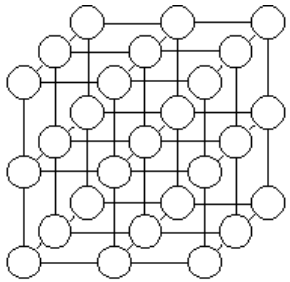
(a)
Árbol estático



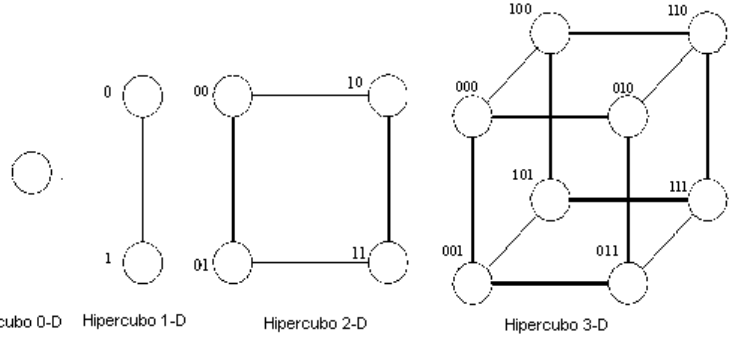
(a)
Mesh



(b)
Toro



(c)
Mesh 3D



Un hipercubo d-dimensional tiene $p=2^d$ procesadores

Perfomance del Sistema de Memoria

Limitaciones en la perfomance del Sistema de Memoria

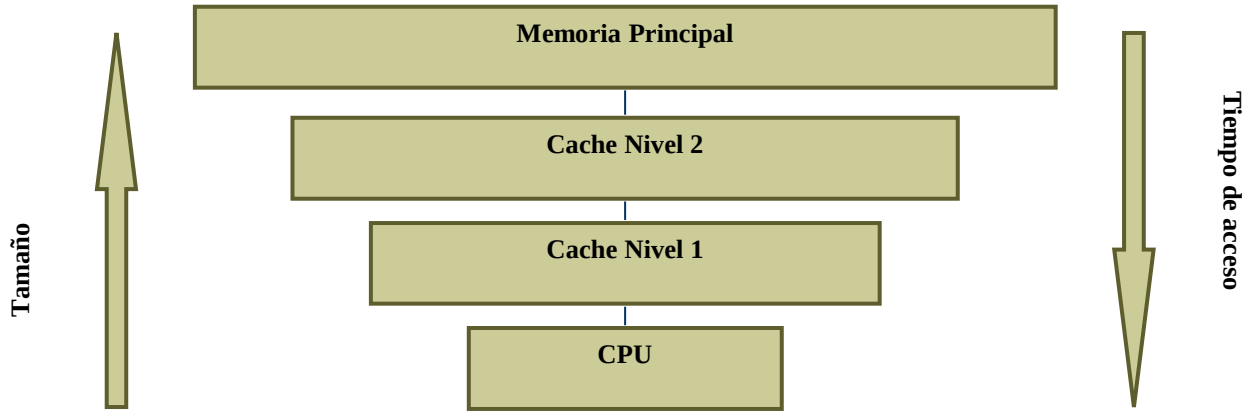
- En muchas aplicaciones la limitación está en el sistema de memoria, no en la velocidad y potencia de cálculo del procesador.
- Los dos parámetros esenciales son la latencia y el ancho de banda del sistema de memoria.
- La latencia es el tiempo que va desde el "*memory request*" hasta que los datos están disponibles.
- El ancho de banda es la velocidad con la cual el sistema de memoria puede ponerlos en el procesador.

Mejora de la latencia efectiva utilizando Caches.

- La memoria cache se caracteriza por tener menor capacidad y menor latencia que la memoria DRAM (además de mayor costo).
- La idea es disminuir la latencia del sistema de memoria, maximizando el número de datos que se acceden de caché.
- Un “cache hit” es cuando buscamos un dato que está en caché. El % de hits es importante porque incide en la latencia global del sistema.

Mejora de la latencia efectiva utilizando Caches.

- **Niveles de memoria.**
  - Jerarquía de memoria. ¿Consistencia?
  - Diferencias de tamaño y tiempo de acceso.
  - Localidad temporal y espacial de los datos.



Optimización del acceso a Memoria. Ejemplo.

- Dado el siguiente código:

```
for (i = 0; i < 1000; i++)
    {  suma[i] = 0.0;
       for (j = 0; j < 1000; j++)  suma[i] += b[j][i];
     }
```

- ¿Cuál es el efecto? → Sumar las columnas de la matriz ***B*** en el vector ***suma***.
- ¿Cuál es el problema si la matriz está en memoria por filas?

- ¿Cuál es el efecto y la diferencia al modificar el código?

```
for (i = 0; i < 1000; i++) column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
     for (i = 0; i < 1000; i++)  column_sum[i] += b[j][i];
```

Resumen de ideas sobre la perfomance del Sistema de Memoria

- Hay que tratar de explotar la localidad temporal y espacial de los datos.
- La relación entre el número de operaciones de procesamiento y el número de accesos a memoria es muy importante para establecer la perfomance efectiva.
- La relación entre el algoritmo y la forma de estructurar los datos en memoria influirá sensiblemente en la perfomance.

Coherencia de Cache en Sistemas Multiprocesador

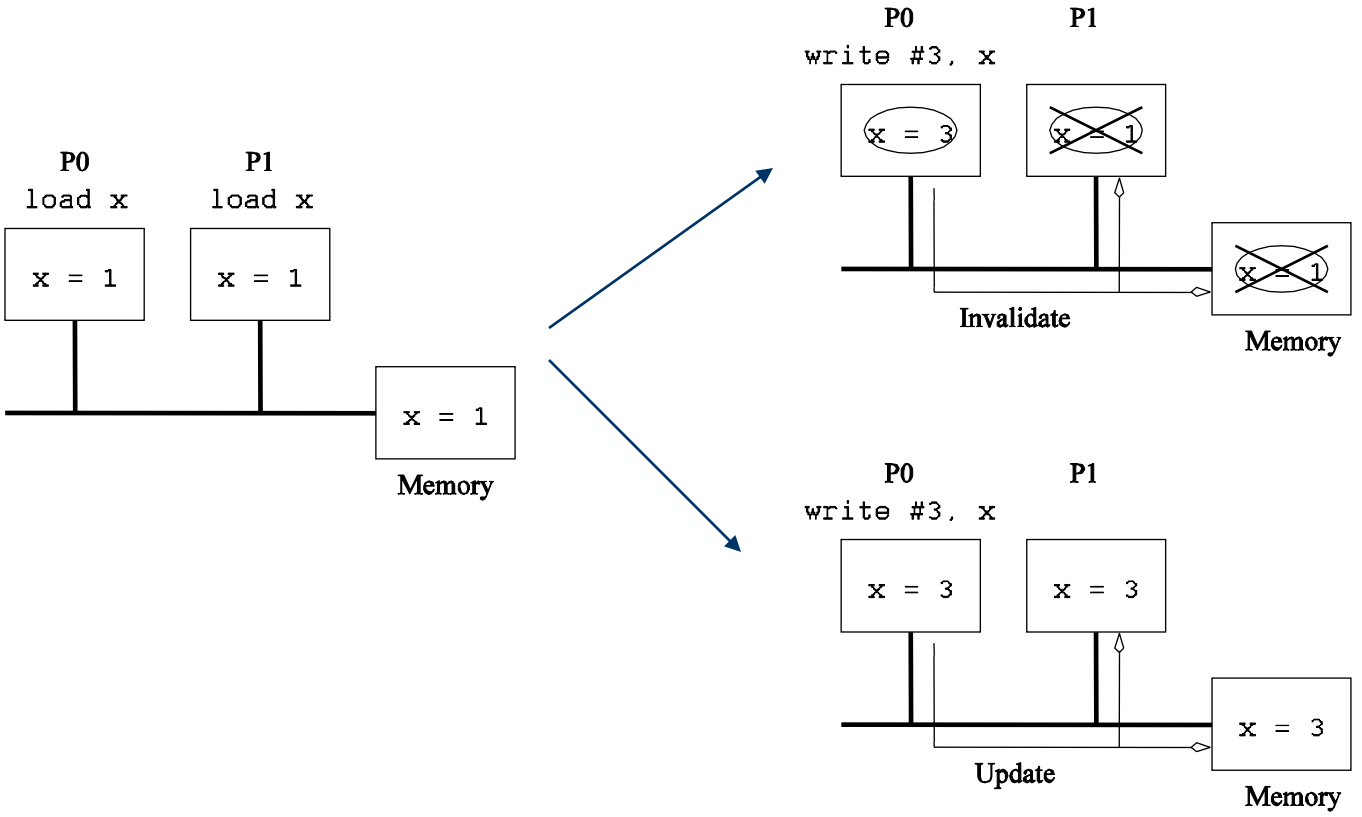
- Los esquemas de interconexión aseguran la comunicación (o transferencia de datos).

- Si tenemos memoria compartida entre procesadores (que pueden tener datos replicados) debemos asegurar la coordinación en el acceso a los mismos.

- Además hay que asegurar la semántica de esta coordinación para que los datos conserven coherencia. Normalmente esto significará alguna serialización en las instrucciones sobre la máquina paralela.

Coherencia de Cache en Sistemas Multiprocesador

Cuando el contenido de una variable cambia, todas sus copias deben ser actualizadas o invalidadas.



Coherencia de Cache: Protocolos Update e Invalidate

- Si un procesador sólo lee un valor una vez y no lo necesita más, un protocolo update puede generar un overhead significativo innecesario. En este caso es mejor el protocolo invalidate.

- Si 2 procesadores trabajan en forma coordinada sobre una variable (alternativamente) será mejor un protocolo de actualización.

- Ambos protocolos sufren de “false sharing” cuando en la misma línea de memoria caché se almacenan datos no compartidos (overhead innecesario).

- La mayoría de las máquinas paralelas utilizan protocolos de invalidación.

- Tradeoff entre overhead de comunicación (update) y ociosidad de procesadores (invalidate).

Diseño de Algoritmos Paralelos

Diseño de Algoritmos Paralelos

- La mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes.
- Puede darse un enfoque metódico para maximizar el rango de opciones consideradas, brindar mecanismos para evaluar las alternativas, y reducir el costo de *backtracking* por malas elecciones.
- Aspectos independientes de la máquina tales como la concurrencia son considerados tempranamente, y los aspectos específicos de la máquina se demoran.

Diseño de Algoritmos Paralelos

¿Por qué es compleja la programación paralela?

- Decidir cuál es la granularidad óptima de las tareas.
- Mapear tareas y datos a los nodos físicos de procesamiento (¿en forma estática o dinámica?)
- Manejar comunicación y sincronización.
- Asegurar corrección. Evitar deadlocks. Evitar desbalances.
- Obtener un cierto grado de Tolerancia a Fallos.
- Manejar la heterogeneidad.
- Lograr escalabilidad en todos los casos (potencia, tamaño de la arquitectura y del problema).
- Consumo energético.

Diseño de Algoritmos Paralelos

- Para diseñar un algoritmo paralelo se deben realizar alguno de los siguientes pasos:
  - Identificar porciones de trabajo (tareas) concurrentes.
  - Mapear tareas a procesos en distintos procesadores.
  - Distribuir datos de entrada, intermedios y de salida.
  - Manejo de acceso a datos compartidos.
  - Sincronizar procesos.

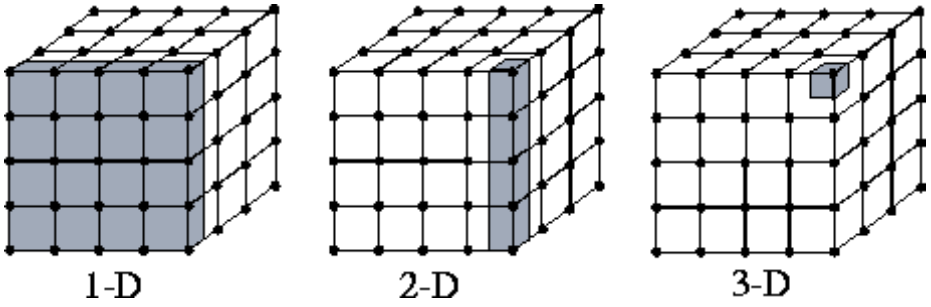
- Pasos Fundamentales: ***Descomposición en Tareas*** y ***Mapeo de Procesos a Procesadores***.

Descomposición en tareas

- Para desarrollar un algoritmo paralelo el primer punto es descomponer el problema en sus componentes funcionales concurrentes (procesos/tareas).
- Se trata de definir un gran número de pequeñas tareas para obtener una descomposición de grano fino, para brindar la mayor flexibilidad a los algoritmos paralelos potenciales.
- En etapas posteriores, la evaluación de los requerimientos de comunicación, arquitectura de destino, o temas de IS pueden llevar a descartar algunas posibilidades detectadas en esta etapa, revisando la partición original y aglomerando tareas para incrementar su tamaño o granularidad.
- Esta descomposición puede realizarse de muchos modos. Un primer concepto es pensar en tareas de igual código (normalmente ***paralelismo de datos o dominio***) pero también podemos tener diferente código (***paralelismo funcional***).

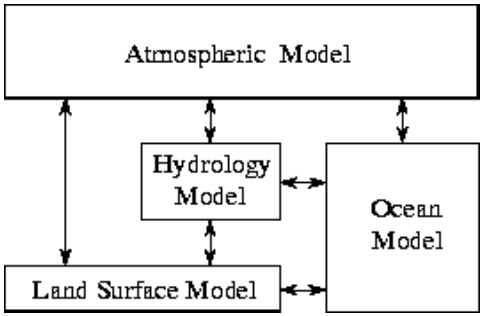
Descomposición en Tareas

- ***Descomposición de datos***: determinar una división de los datos (en muchos casos, de igual tamaño) y luego asociarle el cómputo (típicamente, cada operación con los datos con que opera).
- Esto da un número de tareas, donde cada uno comprende algunos datos y un conjunto de operaciones sobre ellos. Una operación puede requerir datos de varias tareas, y esto llevará a la ***comunicación***.
- Son posibles distintas particiones, basadas en diferentes estructuras de datos. Por ejemplo, diferentes formas de descomponer una estructura 3D de datos. Inicialmente la de grano más fino.



Descomposición en Tareas

- ***Descomposición funcional***: primero descompone el cómputo en tareas disjuntas y luego trata los datos.
- Los requerimientos de datos pueden ser disjuntos (partición completa) o superponerse significativamente (necesidad de comunicación para evitar replicación de datos). En el segundo caso, probablemente convenga descomponer el dominio.
- Inicialmente se busca no replicar cómputo y datos. Esto puede revisarse luego para reducir costos.
- La descomposición funcional tiene un rol importante como técnica de estructuración del programa, para reducir la complejidad del diseño general. Modelos computacionales de sistemas complejos pueden estructurarse como conjuntos de modelos más simples conectados por interfaces.



Aglomeración

- El algoritmo resultante de las etapas anteriores es abstracto en el sentido de que no es especializado para ejecución eficiente en una máquina particular.
- Esta etapa revisa las decisiones tomadas con la visión de obtener un algoritmo que ejecute en forma eficiente en una clase de máquina real.
- En particular, se considera si es útil combinar o aglomerar las tareas para obtener otras de mayor tamaño. También se define si vale la pena replicar datos y/o computación.

Aglomeración

- 3 objetivos, a veces conflictivos, que guían las decisiones de aglomeración y replicación:
  - ***Incremento de la granularidad***: intenta reducir la cantidad de comunicaciones combinando varias tareas relacionadas en una sola.
  - ***Preservación de la flexibilidad***: al juntar tareas puede limitarse la escalabilidad del algoritmo. La capacidad para crear un número variante de tareas es crítica si se busca un programa portable y escalable.
  - ***Reducción de costos de IS***: se intenta evitar cambios extensivos, por ejemplo, reutilizando rutinas existentes.

Características de las Tareas

Una vez que tenemos el problema separado en tareas conceptualmente independientes, tenemos una serie de características de las mismas que impactarán en la perfomance alcanzable por el algoritmo paralelo:

- Generación de las tareas.
- El tamaño de las tareas.
- Conocimiento del tamaño de las tareas.
- El volumen de datos asociado con cada tarea.

Mapeo de tareas a procesadores

- Se especifica dónde ejecuta cada tarea.
- Este problema no existe en uniprocesadores o máquinas de memoria compartida con scheduling de tareas automático.
- **Objetivo:** minimizar tiempo de ejecución. Dos estrategias, que a veces conflictúan: ubicar tareas que pueden ejecutar concurrentemente en ≠ procesadores para mejorar la concurrencia o poner tareas que se comunican con frecuencia en = procesador para incrementar la localidad.
- El problema es NP-completo: no existe un algoritmo de tiempo polinomial tratable computacionalmente para evaluar tradeoffs entre estrategias en el caso general. Existen heurísticas para clases de problema.

Mapeo de tareas a procesadores

- Normalmente tendremos más tareas que procesadores físicos.
- Los algoritmos paralelos (o el scheduler de ejecución) deben proveer un mecanismo de “mapping” entre tareas y procesadores físicos.
- Nuestro lenguaje de especificación de algoritmos paralelos debe poder indicar claramente las tareas que pueden ejecutarse concurrentemente y su precedencia/prioridad para el caso que no haya suficientes procesadores para atenderlas.
- La dependencia de tareas condicionará el balance de carga entre procesadores.
- La interacción entre tareas debe tender a minimizar la comunicación de datos entre procesadores físicos.

Criterio para el mapeo de tareas a procesadores

- Un buen mapping es crítico para el rendimiento de los algoritmos paralelos.

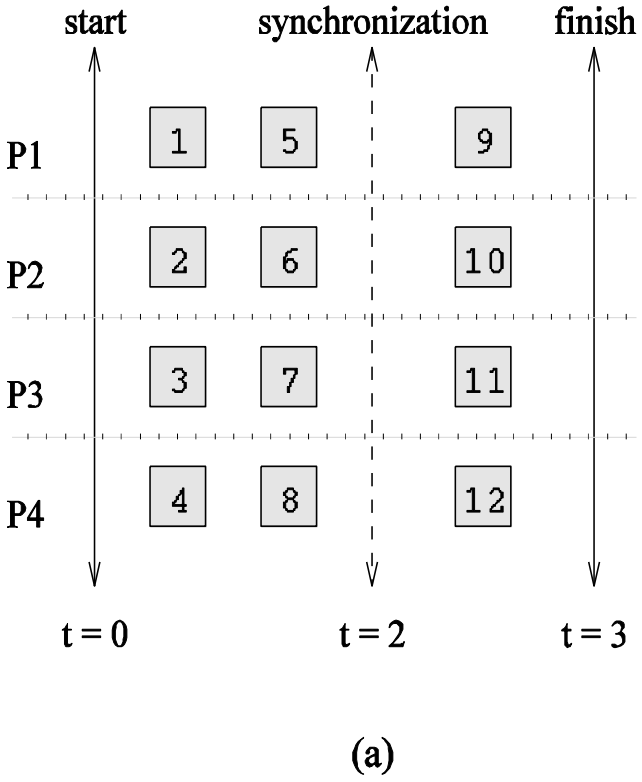
  1. Tratar de mapear tareas independientes a diferentes procesadores.
  2. Asignar prioritariamente los procesadores disponibles a las tareas que estén en el camino crítico.
  3. Asignar tareas con alto nivel de interacción al mismo procesador, de modo de disminuir el tiempo de comunicación físico.

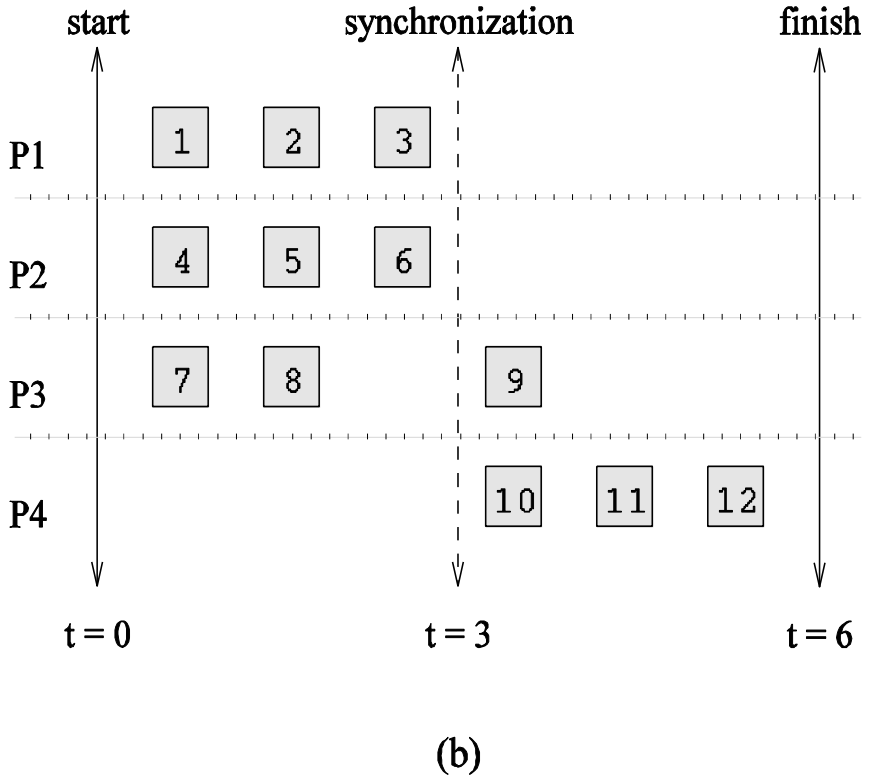
  ⇒ Notar que estos criterios pueden oponerse entre sí … por ejemplo el criterio 3 puede llevarnos a NO paralelizar.

- Debe encontrarse un equilibrio que optimice el rendimiento paralelo ⇒ MAPPING DETERMINA LA EFICIENCIA DEL ALGORITMO.

Técnicas de Mapeo para Mínima espera Ociosa (Idling)

Balancear la carga de trabajo, NO asegura un mapeo con mínima espera:





Técnicas de Mapeo para Mínima espera Ociosa (Idling)

El mapeo puede ser ***estático*** o ***dinámico***.

- ***Estático:*** Las tareas se mapean a procesos a priori.
  - ✓ Es necesario tener una buena estimación del tamaño de cada tarea.
  - ✓ Problema NP completo en el caso general para tareas no uniformes.
  - ✓ Heurísticas que brindan soluciones casi óptimas.
  - ✓ Algoritmos más fáciles de diseñar y programar

- ***Dinámico:*** Las tareas se mapean en ejecución (porque se generan dinámicamente o porque no se conoce su tamaño).
  - ✓ Puede incluir migración de datos que agrega overhead.
  - ✓ Algoritmos más complicados.

Esquemas de Mapeo Estático

- ***Mappings basados en particionamiento de datos***: estructuras de arreglos (bloque, cíclica, bloque-cíclica, bloque-random) y grafos.

- ***Mappings basados en particionamiento de tareas***: Grafos de tareas - Heurísticas.

- ***Mappings híbridos / jerárquico***: en ocasiones conviene combinar las técnicas de mapeo para evitar dejar procesadores ociosos. Por ejemplo: combinar la descomposición en base al grafo de dependencia de tareas con la descomposición de datos en los niveles superiores.

Esquemas de Mapeo Dinámico

- El mapeo dinámico es necesario cuando el estático puede resultar en distribución desbalanceada o cuando el grafo de dependencia de tareas es en sí mismo dinámico.
- Suele referirse como balance de carga dinámico, ya que el Balance de Carga es la principal motivación del mapeo dinámico.
- Requieren alguna forma de mantener una visión global del sistema (a distintos niveles de acuerdo al algoritmo) y algún mecanismo de negociación para migración de procesos y/o datos.
- Es fundamental considerar el costo de implementar el algoritmo de balance de carga dentro de la aplicación.
- Los esquemas dinámicos pueden ser ***centralizados*** (requieren una vista completa del sistema) o ***distribuidos*** (balancean con información obtenida sólo de sus vecinos).

Mapeo dinámico centralizado

- Los procesadores contienen procesos *master* o *worker*.
- Cuando un proceso *worker* termina su trabajo le pide al master correspondiente más trabajo.
- A mayor número de procesos *worker*, mayor tiempo de espera en la comunicación con el master.
- A mayor carga distribuida entre los procesos *worker* inicialmente, mayor probabilidad de desbalance.

Mapeo dinámico distribuido

- Para reducir el “cuello de botella” del Master, podemos habilitar que la carga pueda ser distribuida entre procesos “pares” (sin master).

- Los problemas que surgen son de sincronización: ¿Qué proceso inicia la distribución de carga? ¿Cómo se comunican para decidir qué proceso transfiere a cual? ¿Cuándo se dispara una transferencia?

Métricas de Rendimiento

Métricas del paralelismo

- En el mundo serial la performance con frecuencia es medida teniendo en cuenta los requerimientos de tiempo y memoria de un programa.
- En un algoritmo paralelo para resolver un problema interesa saber cuál es la ganancia en performance.
- Hay otras medidas que deben tenerse en cuenta siempre que favorezcan a sistemas con mejor tiempo de ejecución.
- A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número de procesadores (*sistema paralelo = algoritmo + arquitectura sobre la que se implementa*).

Métricas del paralelismo

- La diversidad torna complejo el análisis de performance...
  - ¿Qué interesa medir?
  - ¿Qué indica que un sistema paralelo es mejor que otro?
  - ¿Qué sucede si agrego procesadores?
- En la medición de performance es usual elegir un problema y testear el tiempo variando el número de procesadores. Aquí subyacen las nociones de speedup y eficiencia, y la *ley de Amdahl*.
- Otro tema de interés es la *escalabilidad*, que da una medida de usar eficientemente un número creciente de procesadores.

Métricas del paralelismo
Speedup (S)

- S es el cociente entre el tiempo de ejecución del algoritmo serial conocido más rápido (T_s) y el tiempo de ejecución paralelo del algoritmo elegido (T_p):

$$S = \frac{T_S}{T_P}$$

- Speedup óptimo depende de la arquitectura (en homogénea P).

$$S_{óptimo} = \sum_{i=0}^{P} \frac{PotenciaCalculo(i)}{PotenciaCalculo(mejor)}$$

- Rango de valores: en general entre *0* y $S_{óptimo}$
- Speedup lineal o perfecto, sublineal y superlineal.

Métricas del paralelismo
Eficiencia (E)

- Cociente entre Speedup y Speedup Óptimo.

$$E = \frac{S}{S_{óptimo}}$$

- Mide la fracción de tiempo en que los procesadores son *útiles* para el cómputo.

- El valor está entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.

Escalabilidad de los Sistemas Paralelos

- Es muy difícil extrapolar la perfomance de un sistema paralelo, a partir de configuraciones con pocos procesadores y conjuntos de datos reducidos.

  - No sirven los estudios con 2, 4, 8 procesadores que proyectan el *Sp* alcanzable con 128 o 256 procesadores o el tiempo de procesamiento cuando tengamos 100 o 1000 veces más datos… ¿Por qué?

  - Básicamente porque los resultados con pequeños conjuntos de datos están afectados por la localidad en el manejo de la memoria, y los resultados con pocos procesadores porque las comunicaciones no computan los costos relacionados con la distancia entre procesadores y la disminución del ancho de banda efectivo.

Métricas del paralelismo
Factores que limitan el Speedup

- Alto porcentaje de código secuencial (*Ley de Amdahl*).
- Alto porcentaje de entrada/salida respecto de la computación.
- Algoritmo no adecuado (necesidad de rediseñar).
- Excesiva contención de memoria (rediseñar código para localidad de datos).
- Tamaño del problema (puede ser chico, o fijo y no crecer con *p*).
- Desbalance de carga (produciendo esperas ociosas en algunos procesadores).
- Overhead paralelo: ciclos adicionales de CPU para crear procesos, sincronizar, etc.

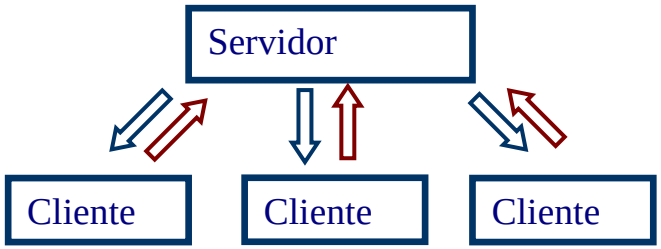
Paradigmas de Programación Paralela

Paradigmas de Programación Paralela

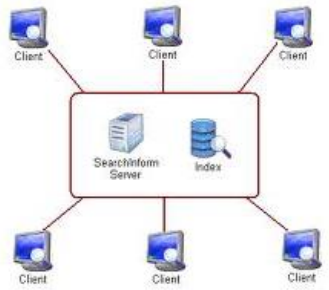
- ***Paradigma de programación***: clase de algoritmos que resuelve distintos problemas, pero tienen la misma estructura de control.
- Para cada paradigma puede escribirse un esqueleto algorítmico que define la estructura de control común.
- Dentro de la programación paralela pueden encontrarse paradigmas que permiten encuadrar los problemas en alguno de ellos.
- En cada paradigma, los patrones de comunicación son muy similares en todos los casos.

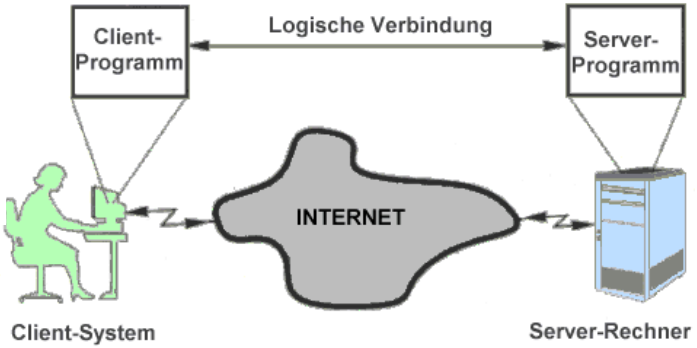
Cliente / Servidor

- Cliente-servidor es el esquema dominante en las aplicaciones de procesamiento distribuido.
- Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Naturalmente unos y otros pueden ejecutarse en procesadores diferentes. Comunicación bidireccional. Atención de a un cliente a la vez, o a varios con multithreading.
- Mecanismos de invocación variados (rendezvous, RPC, monitores).
- El soporte distribuido puede ser simple (LAN) o extendido a la WEB.



Cliente/Servidor





Master/slave o master/worker

- Basado en organizaciones del mundo real.
- El master envía iterativamente datos a los workers y recibe resultados de éstos.
- Posible "cuello de botella" (por ejemplo, por tareas muy chicas o *slaves* muy rápidos) → elección del grano adecuado.
- Dos casos de acuerdo a las dependencias de las iteraciones:
  - ✓ Iteraciones dependientes: el master necesita los resultados de todos los workers para generar un nuevo conjunto de datos.
  - ✓ Entradas de datos independientes: los datos llegan al maestro, que no necesita resultados anteriores para generar un nuevo conjunto de datos

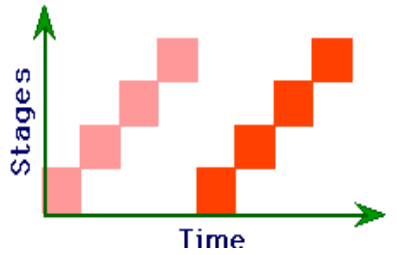
Master/slave o master/worker

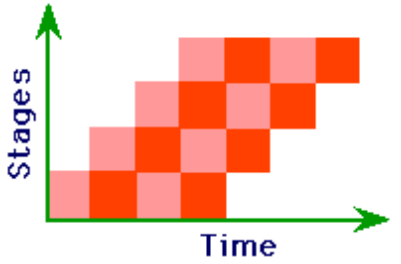
- Dos opciones para la distribución de los datos:
  - ✓ Distribuir todos los disponibles, de acuerdo a alguna política (estático).
  - ✓ Bajo petición o demanda (dinámico).
- Existen variantes, pero básicamente un procesador es responsable de la coordinación y los otros de resolver los problemas asignados.
- Es una variación de SPMD donde hay dos programas en lugar de sólo uno.
- Casos:
  - ✓ Procesadores heterogéneos y con distintas velocidades → problemas con el balance de carga.
  - ✓ Trabajo que debe realizarse en “fases” → sincronización.
  - ✓ Generalización a modelo multi-nivel o jerárquico.

Pipeline y Algoritmos Sistólicos

- El problema se particiona en una secuencia de pasos. El stream de datos pasa entre los procesos, y cada uno realiza una tarea sobre él.
- Ejemplo: filtrado, etiquetado y análisis de escena en imágenes.
- Mapeo natural a un arreglo lineal de procesadores.

Pipe





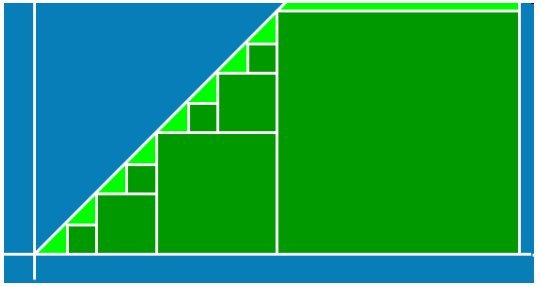
- Extensiones:
  - ✓ Procesadores especializados no iguales.
  - ✓ Más de un procesador para una tarea determinada.
  - ✓ El flujo puede no ser una línea simple (ejemplo: ensamble de autos con varias líneas que son combinadas) → procesamiento sistólico.

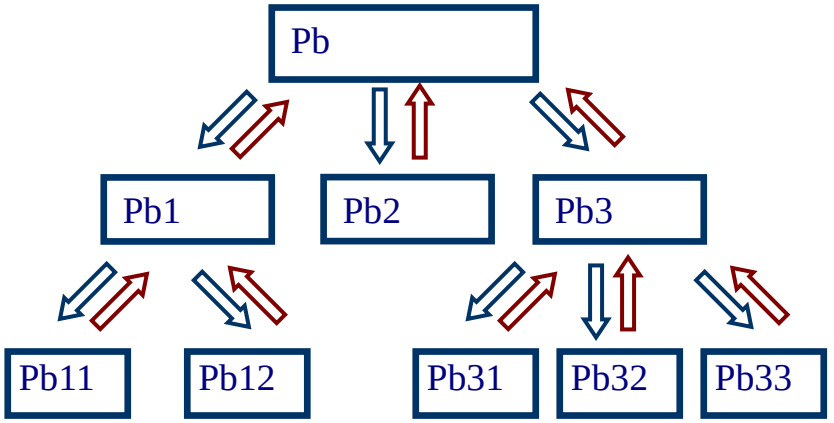
Dividir y Conquistar

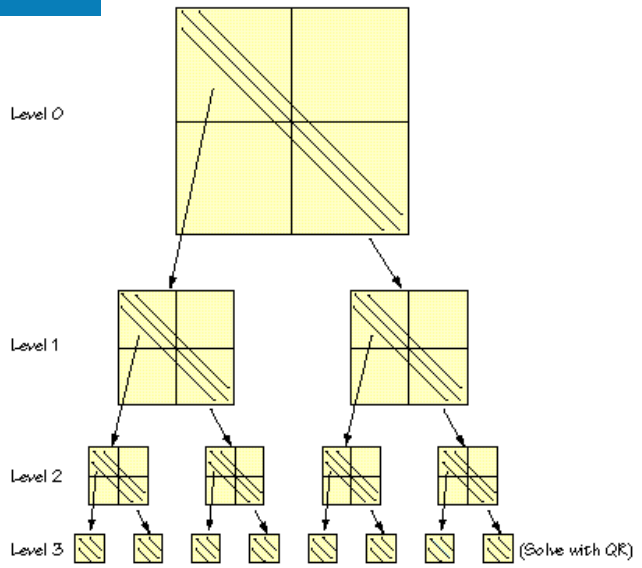
- En general implica ***paralelismo recursivo*** donde el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (*dividir y conquistar*).
- División repetida de problemas y datos en subproblemas más chicos (fase de dividir); resolución independiente de éstos (conquistar), con frecuencia de manera recursiva. Las soluciones son combinadas en la solución global (fase de combinar).
- La subdivisión puede corresponderse con la descomposición entre procesadores. Cada subproblema puede mapearse a un procesador. Cada proceso recibe una fracción de datos: si puede los procesa; sino, crea un n° de “hijos” y les distribuye los datos.

Dividir y Conquistar

- Ejemplos clásicos son el “sorting by merging”, el cálculo de raíces en funciones continuas, problema del viajante.

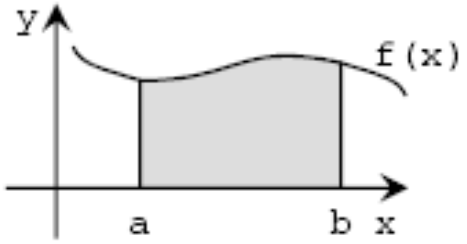


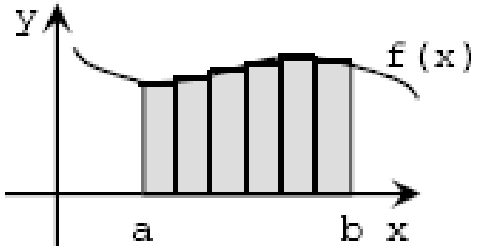




Dividir y Conquistar

Ejemplo el "Problema de la cuadratura": calcular una aproximación de la integral de una función continua *f*(x) en el intervalo de *a* a *b*



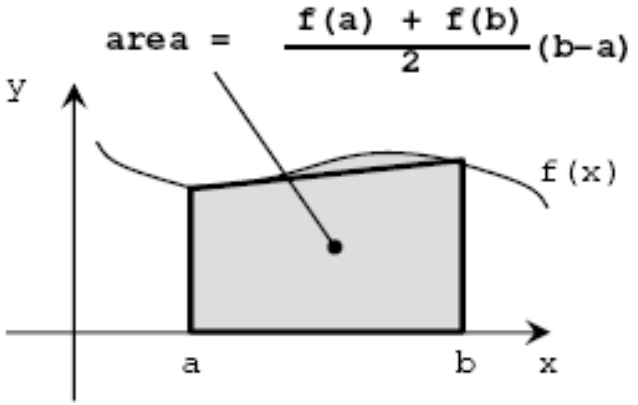


Solución secuencial iterativa (usando el método trapezoidal):

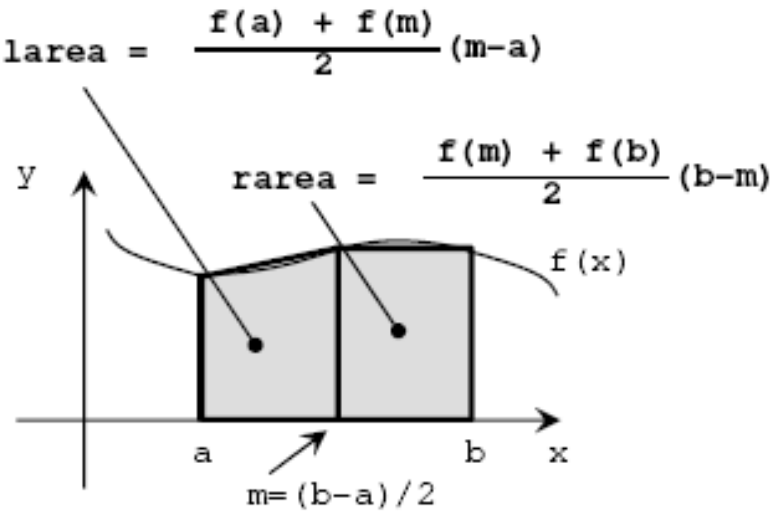
```
double fl = f(a), fr, area = 0.0;
double dx = (b-a)/ni;
for [x = (a+dx) to b by dx]
  {  fr = f(x);
     area = area + (fl+fr) * dx / 2;
     fl = fr;
  }
```

Dividir y Conquistar

Procedimiento recursivo adaptivo



(a) First approximation (area)



(b) Second approximation (larea + rarea)

Si ***abs((larea + rarea) - area) > e***, repetir el cómputo para cada intervalo ***[a,m]*** y ***[m,b]*** de manera similar hasta que la diferencia entre aproximaciones consecutivas esté dentro de un dado ***e.***

Dividir y Conquistar

Procedimiento secuencial

```
double quad(double l,r,fl,fr,area) {
  double m = (l+r)/2;
  double fm = f(m);
  double larea = (fl+fm)*(m-l)/2;
  double rarea = (fm+fr)*(r-m)/2;
  if (abs((larea+rarea)-area) > e) {
     larea = quad(l,m,fl,fm,larea);
     rarea = quad(m,r,fm,fr,rarea);

  }
  return (larea+rarea);
}
```

Procedimiento paralelo

```
double quad(double l,r,fl,fr,area) {
   double m = (l+r)/2;
   double fm = f(m);
   double larea = (fl+fm)*(m-l)/2;
   double rarea = (fm+fr)*(r-m)/2;
   if (abs((larea+rarea)-area) > e) {
      co larea = quad(l,m,fl,fm,larea);
      || rarea = quad(m,r,fm,fr,rarea);
      oc
   }
   return (larea+rarea);
}
```

- Dos llamados recursivos son independientes y pueden ejecutarse en paralelo.
- Uso: area = quad (a, b ,f(a), f(b), (f(a) + f(b)) * (b-a) /2)

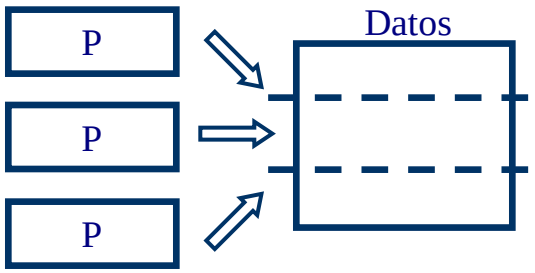
SPMD

- El programador genera un programa único que ejecuta cada nodo sobre una porción del dominio de datos. La diferente evaluación de un predicado en sentencias condicionales permite que cada nodo tome distintos caminos del programa

- Dos fases: 1) elección de la distribución de datos y 2) generación del programa paralelo

1) Determina el lugar que ocuparán los datos en los nodos. La carga es proporcional al número de datos asignado a cada nodo. Dificultades en computaciones irregulares y máquinas heterogéneas.

2) Convierte al programa secuencial en SPMD. En la mayoría de los lenguajes, depende de la distribución de datos.



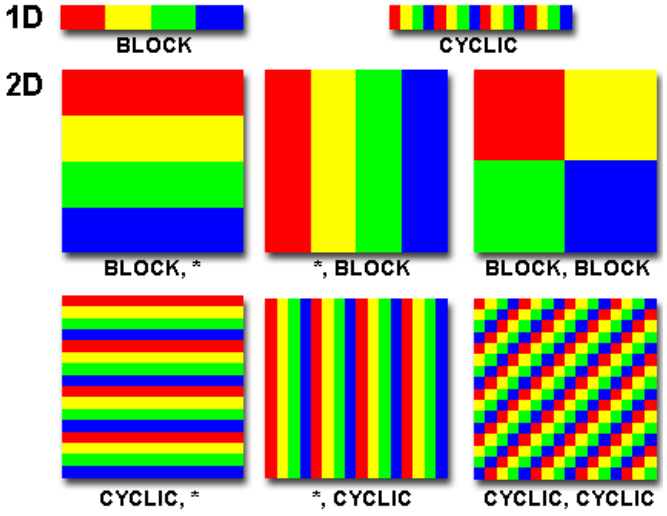
SPMD

SPMD

- Suele implicar ***paralelismo iterativo*** donde un programa consta de un conjunto de procesos los cuales tiene 1 o más *loops*. Cada proceso es un programa iterativo.

- Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}$$

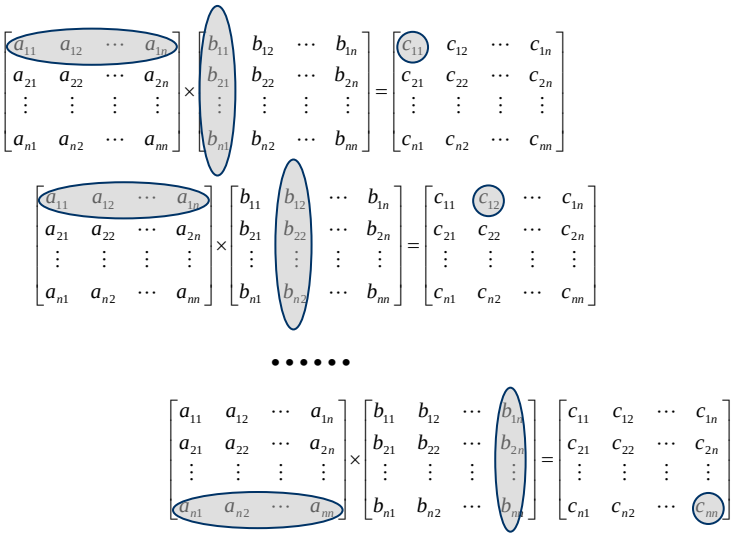


SPMD

Ejemplo de SPMD: multiplicación de matrices.

Solución secuencial:

```
double a[n,n], b[n,n], c[n,n];
for [i = 1 to n]
  { for [j = 1 to n]
      { c[i,j] = 0;
        for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
  }
```

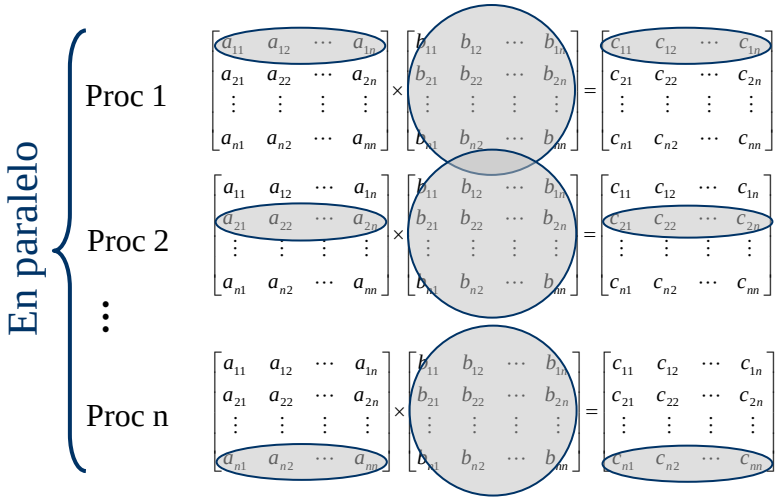


- El loop interno calcula el producto interno de la fila *i* de la matriz *a* por la columna *j* de la matriz *b* y obtiene *c*[*i*,*j*].
- El cómputo de cada producto interno es independiente. Aplicación *embarrasingly parallel* (muchas operaciones en paralelas).
- Diferentes acciones paralelas posibles.

SPMD

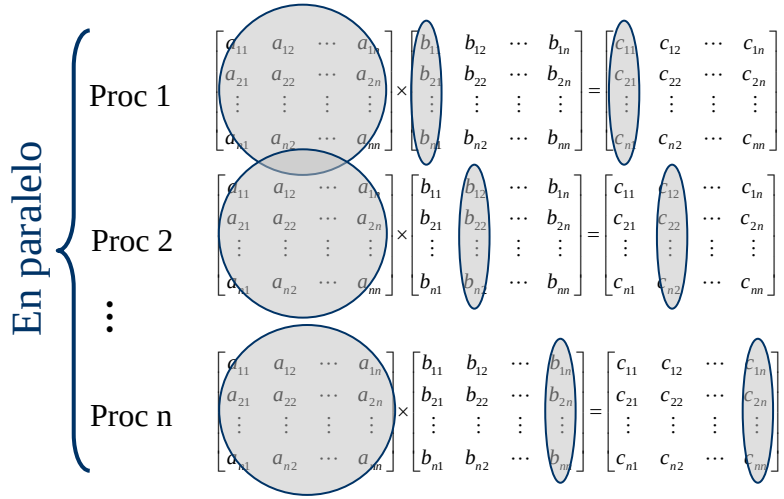
Solución paralela por fila:

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
  { for [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
  }
```



Solución paralela por columna:

```
double a[n,n], b[n,n], c[n,n];
co [j = 1 to n]
  { for [i = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
  }
```



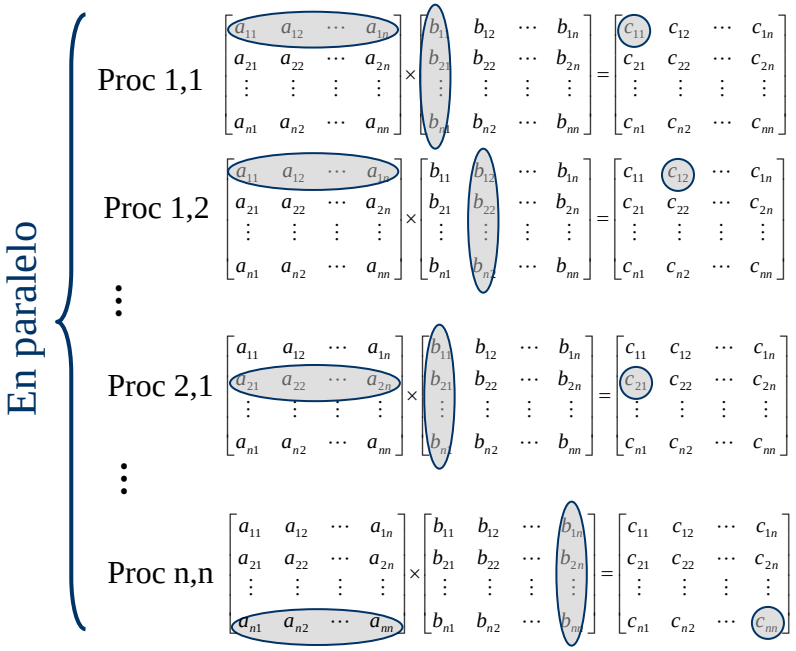
SPMD

Solución paralela por celda (opción 1):

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n , j= 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
```

Solución paralela por celda (opción 2):

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
  { co [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
  }
```



SPMD

Solución paralela por fila con process:

```
process fila [i = 1 to n]
  { for [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
  }
```

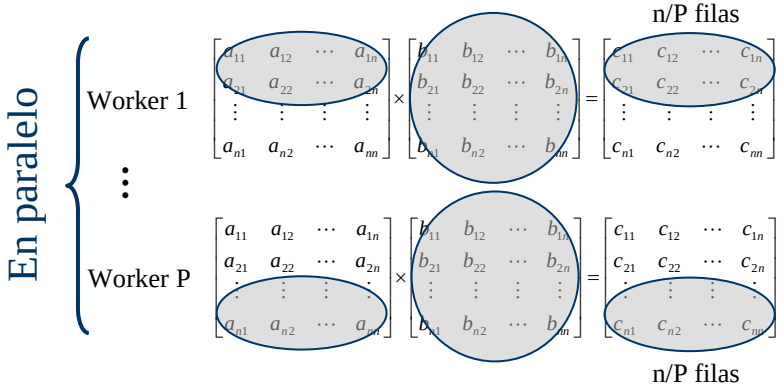
¿Qué sucede si hay menos de *n* procesadores?

- Se puede dividir la matriz resultado en *strips* (subconjuntos de filas o columnas) y usar un proceso ***worker*** por strip.
- El tamaño del strip óptimo es un problema interesante para balancear costo de procesamiento con costo de comunicaciones.

SPMD

Solución paralela por strips:
(*P* procesadores con *P*<*n*)

```
process worker [ w = 1 to P]
  { int primera = (w-1)*(n/P) + 1;
    int ultima = primera + (n/P) - 1;
    for [i = primera to ultima]
      { for [j = 1 to n]
          { c[i,j] = 0;
            for [k = 1 to n]
              c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
          }
      }
  }
```



Ejercicio: a) Si P=8 y n=120. ¿Cuántas asignaciones, sumas y productos hace cada procesador?.
b) Si P1=...=P7 y los tiempos de asignación son 1, de suma 2 y de producto 3; y si P8 es 2 veces más lento. ¿Cuánto tarda el proceso total?. ¿Cuál es el speedup?. ¿Qué puede hacerse para mejorar el speedup?.