Tema 4 PAR - Arquitecturas Paralelas

Paralelismo en los uniprocesadores

Solo hay una única máquina (modelo Von Neumann) ejecutando instrucciones, hay diferentes variantes del uniprocesador:

- Uniprocesadores sèrie: el procesador ejecuta instrucciones en orden.

- Uniprocesadores segmentados: Se ejecutan varias instrucciones a la vez en el pipeline del procesador. La segmentación del procesador hace que se "escondan" la latencia de memoria, fallo de caché. Hay pérdida de ciclos por problemas estructurales o de dependencias de datos[1].

Sobre el uniprocesador podemos "conseguir" paralelismo de varias formas:

- Paralelismo a nivel de instrucción (procesadores superescalares) : Añadimos hardware extra en el pipeline y podemos ejecutar más instrucciones a la vez (en el mismo ciclo) DEL MISMO CONTEXTO DE EJECUCIÓN[2].

- Paralelismo a nivel de thread: "Rellenar" los ciclos vacíos (o perdidos por fallo de cache, latencia de memoria) del pipeline con instrucciones de diferentes contextos de ejecución.

- Paralelismo a nivel de datos (instrucciones multimedia-SIMD[3]) : Se trata de calcular, una única instrucción sobre un gran conjunto de datos (por ejemplo los datos de un bucle for se pueden hacer todos en paralelo con una única instrucción SIMD).

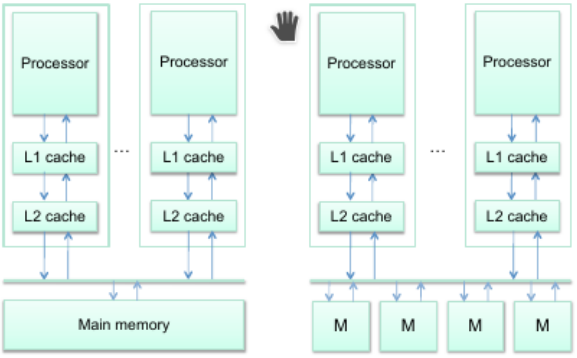
En resumen, podemos conseguir paralelismo en un programa con:

- Principios de localidad espacial y temporal.
- Vectorizando operaciones con instrucciones SIMD.
- Alineando a bloque cache / memoria para un acceso más rápido
- Reordenación de instrucciones para explotar el paralelismo a nivel de instrucción.
- Típicos de todos los programas, a través de la jerarquía de memoria.
- Programación consciente de la arquitectura (PCA).

[1] Temario de arquitectura de computadores II
[2] Del mismo thread
[3] Single Instruction Multiple Data

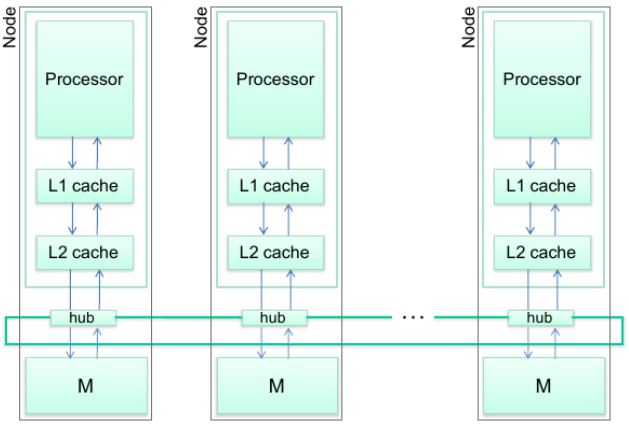
Arquitecturas paralelas

Hay muchas maneras de construir/distribuir un multiprocesador, todas tienen ventajas y desventajas. Se pueden combinar arquitecturas.



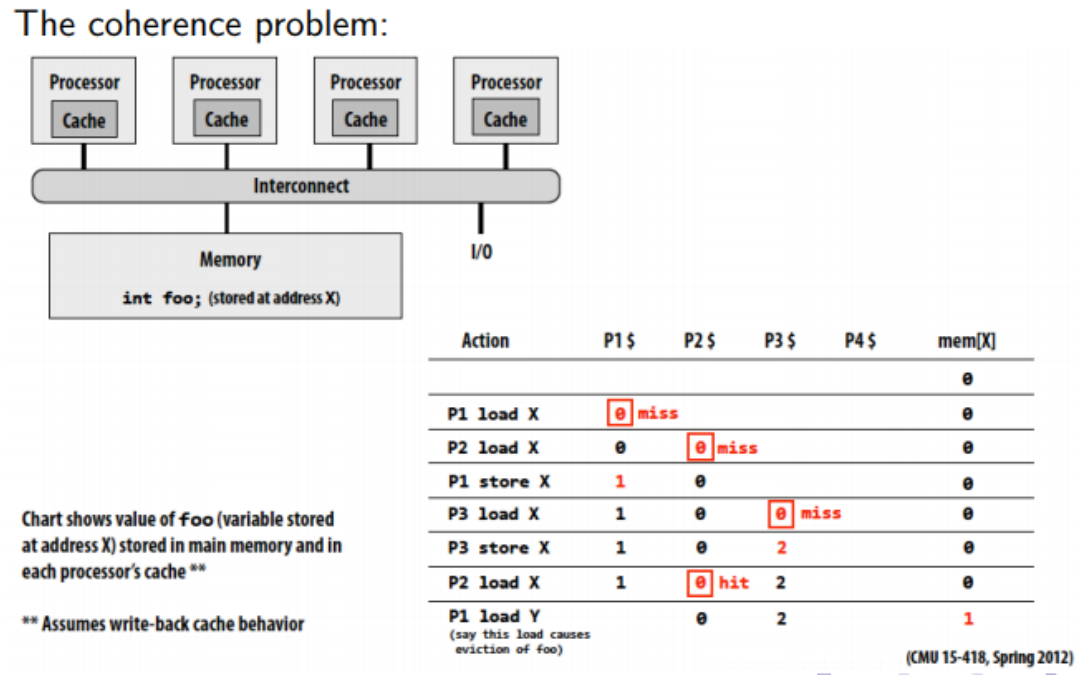
- Memoria centralizada (*shared-memory*): *Symmetric Multiprocessor Architecture* SMP
    - Un único espacio de direcciones y una única instancia de S.O.
    - El acceso a memoria de todos los procesadores es uniforme (U.M.A.).
    - El acceso a memoria se hace con instrucciones load/store.
    - **Existen problemas de coherencia de datos en memoria**

- Memoria distribuida(*Distributed-Shared memory*). Diferentes implementaciones:
    - *NUMA architecture*
        - Tiempo no uniforme de acceso (se utilizan instrucciones load/store).
        - Único espacio de direcciones compartido, pero los módulos de memoria están distribuidos.
    - *Multi - node/cluster architecture* (clusters).
        - Diferentes módulos independientes (y instancias de S.O.)
        - Los accesos a memoria se hacen por una interfaz de paso de mensajes(MPI).



Coherencia de datos

En el ejemplo, acabamos teniendo 3 valores diferentes de una misma variable.



En PAR, hay dos formas (protocolos) de gestionar estas situaciones de coherencia de datos:

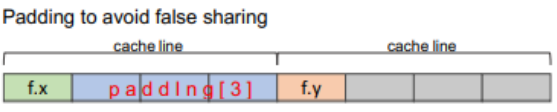
1) **Write-update:** Cada escritura de un procesador hace que se actualicen todas las copias.

2) **Write-invalidate:** Cada escritura fuerza a invalidar las copias y el nuevo valor es proporcionado cuando es pedido por otra caché o la política de reemplazo expulsa un bloque.

Existen dos implementaciones hardware que dan soporte a los dos protocolos de coherencia (*broadcast* y *directory-based (centralized)*).
Para gestionar los datos de una caché, se utiliza la unidad de línea de caché; es decir que los protocolos y mecanismos de coherencia actúan por **cada línea de memoria caché del sistema y no por las variables. Este hecho hace que el tráfico en el bus pueda causar overhead**

True versus false sharing

Si en una misma línea de caché, tenemos mezcla de datos que no compartidos y datos que si estamos manteniendo una coherencia sobre datos que no se están utilizando. A este fenómeno se le llama ***false sharing***. Si todos los datos que se comparten sí que se necesitan compartir, esto se llama ***true sharing.*** En el ejemplo, podemos ver como dos variables de un *struct* se pueden separar añadiendo *padding*.



Broadcast-based (snooping) - MSI

Todas las caches ven todas las operaciones de acceso a memoria de todas las CPU's en un orden establecido. El clásico protocolo usado es el MSI o alguna variante suya:

Write-invalidate snooping protocol - MSI

En este protocolo, tendremos que cada CPU puede realizar lecturas y escrituras (eventos PrRd y PrWr) contra la jerarquía de memoria y el sistema de *broadcast* genera diferentes peticiones por el bus que ven todas las caches:

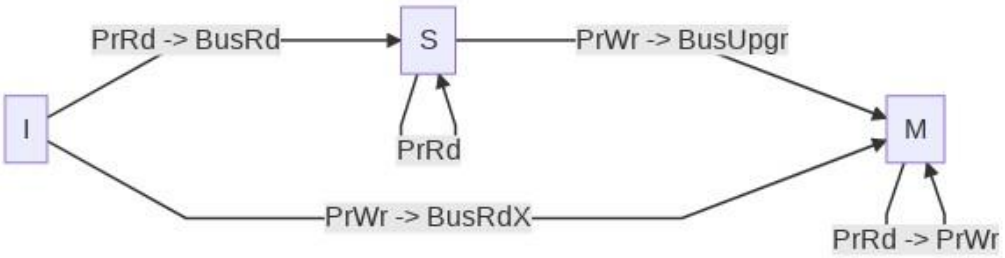
- BusRd: Petición de una línea para lectura.
- BusRdX: Petición de una línea con intención de modificarla.
- BusUpgr : Petición de escritura contra una línea de caché, causa la invalidación de las copias.
- Flush : Alguien nos pide el dato de caché o la política de reemplazo saca una linea en estado **M.**

Tenemos 3 posibles estados para una línea de caché:

1. **I**nvalido: La línea de caché no es válida o no existe en la caché.
2. **M**odificado: Copia "sucia" de la línea de caché.
3. **S**hared: Copia "limpia" y compartida de una línea de cache.

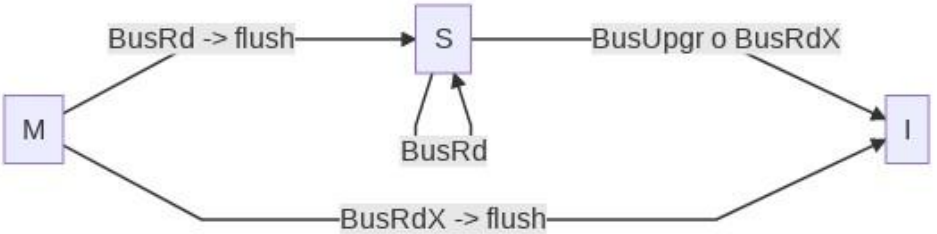
Tenemos dos diagramas de transición de estados:

1. En función de qué operación hace nuestra CPU y cómo afecta a nuestras líneas de caché.



En la imagen, -> se lee como "implica"

2. En función de qué operación realiza otra CPU y cómo afecta a nuestras líneas de caché. Cada caché observa los eventos del bus.



En la imagen, -> se lee como "implica". Si llega cualquier evento a I, nos quedamos y solo saldremos de él si necesitamos leer algo.

Ejemplo de ejecución - MSI

Ejemplo de cómo varían los estados en una ejecución de instrucciones, con política de escritura retardada en acierto.

Evento	**Cache P1**	**Cache P2**	**Cache P3**	**Status P1**	**Estado P2**	**Estado P3**	**MEM**	**Comentario**
Init, T = 0	-	-	-	I	I	I	0	Si la linea no esta en caché, estado I
P1 ld X	0 (miss)	-	-	S	I	I	0	Estado I → Estado S a causa de una lectura (PrRd)
P2 ld X	0	0(miss)	-	S	S	I	0	IDEM anterior
P1 st 1, X	1	0	-	M	I	I	0	Estado S → Estado M Una PrWr del procesador invalida todas las demás copias.
P3 ld X	1	0	1 (miss)	S	I	S	0	P3 hace una lectura (de I → S) y el dato se lo proporcionará la caché de P1 (flush). El dato de P1 pasa a estar compartido (estado S)
P3 st 2, X	1	0	2	I	I	M	0	P3 tiene una copia válida, estado S → estado M e invalida las demás.
P2 load X	1	2	2	I	S	S	0	P2 tiene una copia invalida, P3 hace flush y proporciona la línea de cache a P2.
P1 load Y	mem[Y]	2	2	I	S	S	0	Flush no upd.
P2 load Y	mem[Y]	mem[Y]	2	I	I	S	2	Flush+ update

Resumen y escalabilidad

- El protocolo de coherencia actúa sobre las líneas de caché.
- Si estamos en los estados **I** o **S,** el dato lo proporciona la memoria principal.
- Solamente puede haber una única copia sucia en toda la caché.
- Si hay una copia sucia (estado **M**) en alguna caché y hay una petición sobre esta línea, esta caché proporcionará el dato.
- Hay variaciones/optimizaciones de este protocolo añadiendo más estados para controlar ciertos casos.
- Este tipo de protocolos de *broadcast* no escalan, la red se puede colapsar en algún momento por la cantidad de mensajes necesarios; podemos evitar esto teniendo un sistema centralizado (directorio).

Directory based - protocol MSU

Se centraliza el protocolo, y hay una región de memoria donde se mantiene el estado de cada línea de memoria.

- Pese a que el directorio esta en memoria principal, cada línea del directorio puede estar en un nodo
- La coherencia se mantiene con mensajes entre nodos.
- Podemos tener juntos protocolos de *Snooping(broadcast)* y *Directory(Centralized)*.
- Cada entrada del directorio, tiene una lista de nodos compartidos(un bit por cada nodo) y dos bits de estado:
    - Estado U : No hay copias, Uncached.
    - Estado S : Shared.
    - Estado M : Modificado.
- Existen diferentes tipos de nodos, en función de quien mantiene la coherencia:
    - *Home node* : Nodo donde la línea esta asignada y donde esta la *slice* del directorio.
    - *Local node* : Procesador del nodo que accede a una línea.
    - *Remote node* : Nodo que accede a una línea y la modifica (y se convierte en *owner* o se convierte en *reader*).

Al *home node* pueden llegar peticiones de un *local node*:

- RdReq : Solicita una copia de una línea para lectura.
- WrReq : Solicita una copia para escritura.
- UpgrReq : Solicita modificar un dato de una línea existente. Un mensaje de control de respuesta es necesario (ACK)

Un *home node* puede generar dos peticiones a otros *remote node*:

- Fetch : Solicita a un *owner* una copia (Dreply)
- Invalidate : Solicita a un *reader* que invalide una copia suya (se necesita un ACK también).

Inicialización de variables en memoria distribuida

Hay que tener en cuenta donde inicializamos las variables, para poder distribuir la carga

```
for (int i=0; i<128; i++) {
    a[i] = random();
    b[i] = random();
}
```

Vectors a and b are allocated in a single node of of the NUMA system, as follows

M_0
0..127

	P_0 @ M_0	P_1 @ M_1	P_2 @ M_2	P_3 @ M_3
for1	0..31	32..63	64..95	96..127
for2	0..31	32..63	64..95	96..127

x..y No coherence traffic

x..y Coherence traffic

```
#pragma omp parallel num_threads(4)
{
  int myid = omp_get_thread_num();
  int BS = 128 / omp_get_num_threads();
  for (int i=myid*BS; i<(myid+1)*BS; i++) {
    a[i] = random();
    b[i] = random();
  }
}
```

Vectors a and b are distributed across the memories of the NUMA system, as follows

M_0	M_1	M_2	M_3
0..31	32..63	64..95	96..127

	P_0 @ M_0	P_1 @ M_1	P_2 @ M_2	P_3 @ M_3
for1	0..31	32..63	64..95	96..127
for2	0..31	32..63	64..95	96..127

x..y No coherence traffic

x..y Coherence traffic

Anexo 1 - Repaso de memoria caché

La jerarquía de memoria explota las propiedades de la localidad espacial y temporal típicas de los programas. Hay varias propiedades que afectan al rendimiento de la caché:

1. Organización de la jerarquía de memoria.
    a. Una única caché.
    b. Múltiples niveles de caché.
    c. Caché de datos y de instrucciones.

2. Política de asignación en la caché : ¿Dónde colocamos una dirección de memoria?
    a. Directa.
    b. Asociativa.
    c. Asociativa por conjuntos.
3. Política de reemplazo ¿Que hacer si no hay sitio y tenemos que traer un bloque?
    a. LRU
    b. FIFO
    c. etc...
4. Escritura en acierto : ¿Qué hacer si una escritura acierta en caché?
    a. Write-through : Actualizar MEM + CACHE.
    b. Write-back : Actualizar CACHE y memoria cuando la línea salga de la caché.
5. Escritura en fallo : ¿Qué hacer si escribimos y fallamos ?
    a. Write-allocate: Nos traemos la línea de caché.
    b. Write-no-allocate: No nos traemos la línea.