SISTEMAS PARALELOS

Clase 2 – Sistema de memoria

UNIVERSIDAD
NACIONAL
DE LA PLATA

Agenda de la clase anterior

- Fundamentos de procesamiento paralelo
- Plataformas de procesamiento paralelo
- Modelos de programación paralela
- Evolución de los procesadores
- Clusters

Agenda de esta clase

- *Memory Wall:* Limitaciones en el rendimiento del sistema de memoria
- Arreglos multidimensionales y su organización en memoria
- Coherencia de caché en arquitecturas multiprocesador
- Costos de comunicación

MEMORY WALL: LIMITACIONES EN EL RENDIMIENTO DEL SISTEMA DE MEMORIA

Memory Wall: Limitaciones en el rendimiento del sistema de memoria

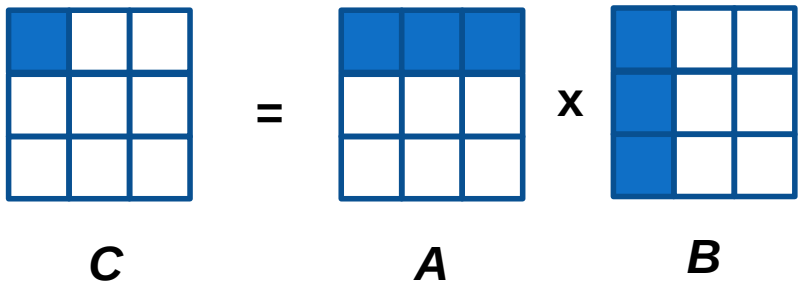
- En muchas aplicaciones, la limitación se encuentra en el sistema de memoria y no en la velocidad del procesador
- Los dos parámetros fundamentales del sistema de memoria a tener en cuenta son la **latencia** y el **ancho de banda**
- La latencia es el tiempo que transcurre desde que se solicita el dato hasta que el mismo está disponible
- El ancho de banda es la velocidad con la cual el sistema puede *alimentar* al procesador
  - Analogía con manguera y agua

Latencia de la memoria: un ejemplo

- Consideremos un procesador capaz de ejecutar 1 instrucción de punto flotante por nanosegundo (ns) conectado a una memoria con una latencia de 100 ns (sin cachés)
  - El rendimiento pico teórico de este sistema es de 1 segundo = 1.000.000.000 nanosegundos → 1.000.000.000 flop's por segundo = 1GFlops

- Supongamos que debemos multiplicar 2 matrices A y B de *n*x*n* elementos cada una:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$$



Latencia de la memoria: un ejemplo

- Consideremos un procesador capaz de ejecutar 1 instrucción de punto flotante por nanosegundo (ns) conectado a una memoria con una latencia de 100 ns (sin cachés)
  - El rendimiento pico teórico de este sistema es de 1 segundo = 1.000.000.000 nanosegundos → 1.000.000.000 flop's por segundo = 1GFlops

- Supongamos que debemos multiplicar 2 matrices A y B de nxn elementos cada una
  - Para esto se requieren n^3 operaciones de multiplicación/suma → $2n^3$ ns de cómputo
  - Para cada operación de multiplicación/suma se requiere acceder a memoria para buscar ambos operandos → $200n^3$ ns de acceso a memoria
  - Para guardar los resultados en memoria se requiere un acceso por cada elemento → $100n^2$ ns

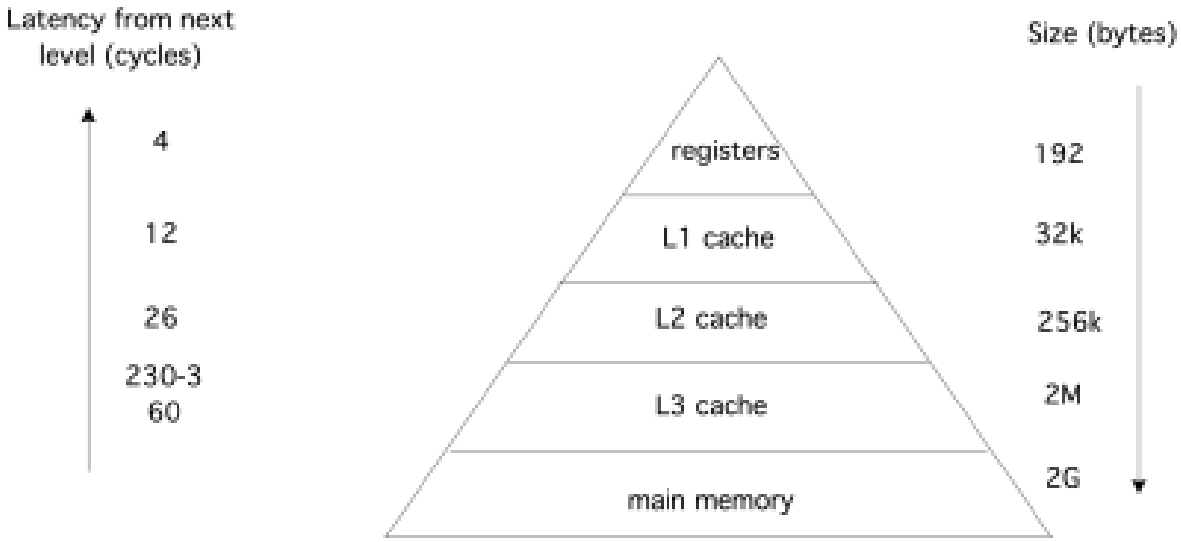
Latencia de la memoria: un ejemplo

- Si *n*=32 entonces:
  - Se requieren $2x32^3$ operaciones de multiplicación/suma → $2x2^{15}$ ns de cómputo.
  - Para cada operación de multiplicación/suma se requiere acceder a memoria para buscar ambos operandos → $200x2^{15}$ ns de acceso a memoria
  - Para guardar los resultados en memoria se requiere un acceso por cada elemento → $100x2^{10}$ ns
  - En total: $102,64x2^{16}$ ns para resolver 2^{16} operaciones → 9,75 Mflops

El rendimiento baja de 1 Gflop (1000 Mflops) a 9,75 Mflops

Reduciendo la latencia usando cachés

- Las caché:
  - Son memorias de alta velocidad y baja capacidad que (usualmente) están integradas al chip
  - Actúan como memoria intermedia entre los registros de la CPU y la memoria principal



Reduciendo la latencia usando cachés

- El objetivo es disminuir la latencia del sistema de memoria maximizando el número de datos que se acceden desde de la caché.

- Cuando la CPU necesita un dato, primero se revisa si el mismo reside en la caché
  - Si el dato está en la caché, se produce un *cache hit* y el pedido se satisface rápidamente, con baja latencia.
  - Si el dato no está en la caché, se produce un *cache miss* (o fallo de caché) y el dato debe ser buscado en memoria RAM. En ese caso, el dato es copiado en la caché antes de llegar a los registros.

- La tasa de *hits* es importante porque incide directamente en la latencia global del sistema

Impacto del uso de caché: un ejemplo

- Consideremos el mismo sistema del ejemplo anterior pero ahora con una memoria caché de 32Kb con latencia de 4 ns y ancho de banda igual a un elemento de la matriz
- Se debe resolver la misma multiplicación de matrices con n=32. Como ambas matrices (A y B) entran en caché, se accede a memoria únicamente una vez por cada elemento:
  - Para esto se requieren n^3 operaciones de multiplicación/suma → $2n^3$ ns de cómputo
  - Para cada operación de multiplicación/suma se requiere acceder a memoria para buscar ambos operandos → Para cada elemento se accede una vez a memoria (100 ns) y el resto a caché (4x(n-1) ns) → en total se requieren (96+4xn) $2n^2$ ns de acceso a memoria
  - Para guardar los resultados en memoria se requiere un acceso por cada elemento → $100n^2$ ns

Impacto del uso de caché: un ejemplo

- Si *n*=32 entonces:
  - Se requieren $2x32^3$ operaciones de multiplicación/suma → $2x2^{15}$ ns de cómputo.
  - Para cada operación de multiplicación/suma se requiere acceder a memoria o caché para buscar ambos operandos → $448x2^{10}$ ns de acceso a operandos
  - Para guardar los resultados en memoria se requiere un acceso por cada elemento → $100x2^{10}$ ns
  - En total: $9,5625x2^{16}$ ns para resolver 2^{16} operaciones → 104,6 Mflops

El rendimiento sube de 9,75 a 104,6 Mflops

  - La mejora obtenida por la inclusión de la caché se basa en la suposición de que habrá una repetición de referencias a determinados datos en una ventana de tiempo pequeña (*localidad temporal*)

Impacto del ancho de banda

- El ancho de banda es la velocidad con la que los datos pueden ser transferidos desde la memoria al procesador y está determinado por el ancho de banda del bus de memoria como de las unidades de memoria

- Una técnica común para mejorar el ancho de banda del sistema consiste en incrementar el tamaño de los bloques de memoria que se transfieren por ciclo de reloj

- El sistema de memoria requiere *l* unidades de tiempo (latencia) para obtener *b* unidades de datos (b es el tamaño del bloque medido en bits, bytes o words)

Impacto del ancho de banda: un ejemplo

- Consideremos el mismo sistema del ejemplo anterior (memoria caché de 32Kb con latencia de 4 ns) pero con ancho de banda igual a 4 elementos de la matriz
- Se debe resolver la misma multiplicación de matrices con $n=32$. Como ambas matrices (A y B) entran en caché, se accede a memoria únicamente una vez por cada 4 elementos:
  - Para esto se requieren n^3 operaciones de multiplicación/suma → $2n^3$ ns de cómputo
  - Para cada operación de multiplicación/suma se requiere acceder a memoria para buscar ambos operandos → Para cada 4 elementos se accede una vez a memoria (100 ns) y el resto a caché (4x(n-1) ns del primero más 3x4xn ns del resto) → en total se requieren (24+4xn) $2n^2$ ns de acceso a memoria
  - Para guardar los resultados en memoria se requiere un acceso por cada 4 elementos → $25n^2$ ns

Impacto del ancho de banda: un ejemplo

- Si *n*=32 entonces:
  - Se requieren $2x32^3$ operaciones de multiplicación/suma → $2x2^{15}$ ns de cómputo.
  - Para cada operación de multiplicación/suma se requiere acceder a memoria o caché para buscar ambos operandos → $304x2^{10}$ ns de acceso a operandos
  - Para guardar los resultados en memoria se requiere un acceso por cada 4 elementos → $25x2^{10}$ ns
  - En total: $6,140625x2^{16}$ ns para resolver 2^{16} operaciones → 162,8 Mflops

El rendimiento sube de 104,6 a 162,8 Mflops

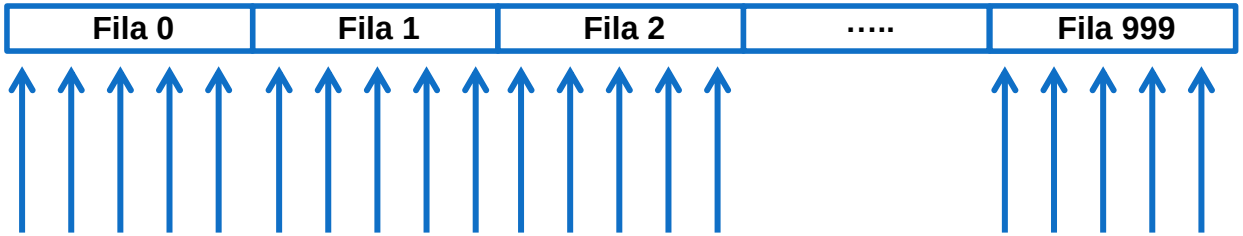
  - La mejora de rendimiento es posible dado que las sucesivas instrucciones usan datos que se encuentran consecutivos en la memoria (*localidad espacial*)

Impacto del acceso no lineal

- El siguiente código computa la suma total por columnas de B:

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

- ¿Cuál es el problema si la matriz B está en memoria por filas?

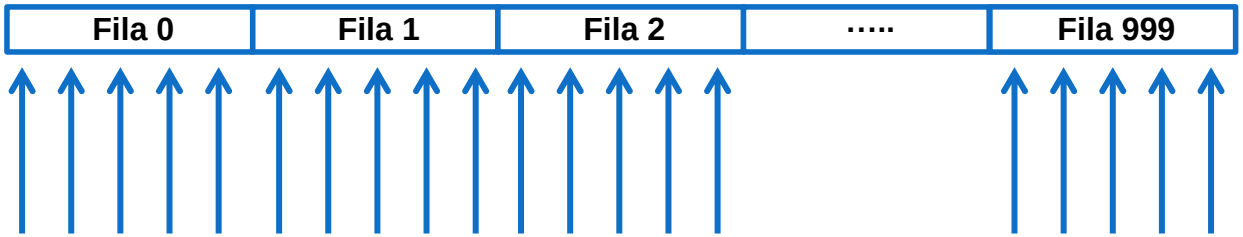


- *No se aprovecha localidad de datos*

Impacto del acceso no lineal

- ¿Qué sucede al restructurar el código de la siguiente forma?:

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



- ¿Qué sucede si la matriz es muy grande y no entra en la caché? ¿Cómo hacer para aprovechar la localidad?

Resumen de ideas para mejorar el rendimiento del sistema de memoria

- Explotar localidad espacial y temporal de datos es crítico para amortizar la latencia e incrementar el ancho de banda efectivo
- La relación entre el número de instrucciones y el número de accesos a memoria es un buen indicador temprano del rendimiento efectivo del sistema
- La organización de los datos en la memoria y la forma en que se estructura el código pueden impactar significativamente en el rendimiento final del sistema

ARREGLOS MULTIDIMENSIONALES Y SU ORGANIZACIÓN EN MEMORIA

Arreglos multidimensionales y su organización en memoria

- La estructura de datos más utilizada en HPC son los arreglos multidimensionales
  - Listas, árboles o grafos también son posibilidades aunque se emplean excepcionalmente.
  - Como veremos más adelante, esto está directamente ligado al beneficio que provee disponer de los datos en forma de arreglo.

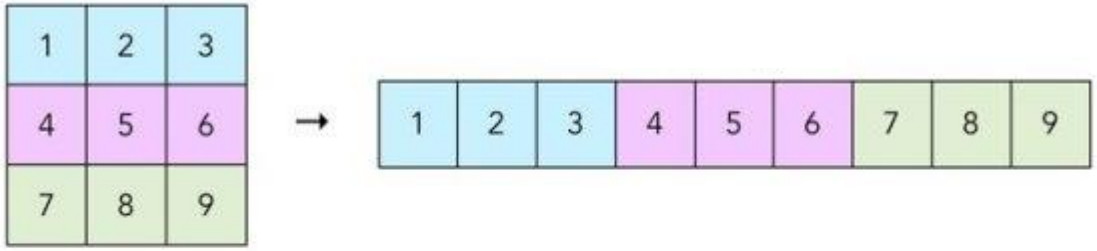
- Existen 2 maneras en que los datos de un arreglo son almacenados en memoria:
  - Por filas
  - Por columnas

Arreglos multidimensionales y su organización en memoria

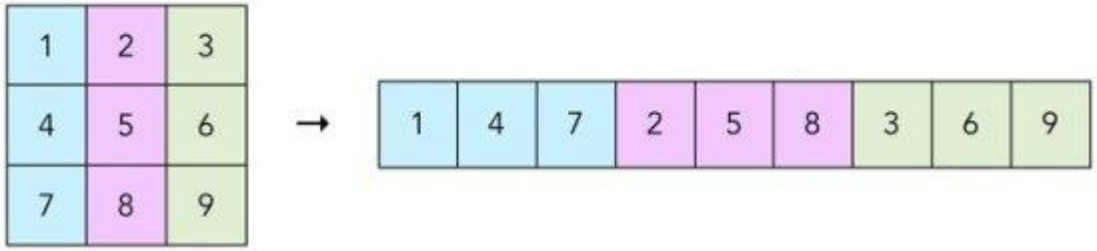
Vista lógica

Vista física
(en memoria)

Por filas
C, C++, Pascal,
Python, entre otros



Por columnas
Fortran, Octave, R,
Julia, entre otros



Arreglos multidimensionales y su organización en memoria

- ¿Cómo declarar un arreglo en el lenguaje C?
  - Supongamos una matriz de 100x100 elementos *float*
  - Veamos las alternativas…

Definición de arreglos en C

- **Alternativa 1 (arreglo estático):**

```
#define N 100

int main (int argc, char * argv[])
    …
    float matriz[N][N];
    …
}
```

¿Cómo accedo a la posición [i , j] siendo *i* el número de fila y *j* el de la columna?

- ¿Tiene un tamaño máximo por la forma en que está declarado?
- ¿Puedo cambiar su tamaño en ejecución?
- ¿Puedo elegir cómo se organizan sus datos (por filas o por columnas)?
- ¿Están todos sus datos contiguos en la memoria?

Definición de arreglos en C

- **Alternativa 2 (arreglo de longitud variable):**

```
int main (int argc, char * argv[])
     …
     int n = 100;
     float matriz[n][n];
     …
}
```

¿Cómo accedo a la posición [i , j] siendo *i* el número de fila y *j* el de la columna?

- ¿Tiene un tamaño máximo por la forma en que está declarado?
- ¿Puedo cambiar su tamaño en ejecución?
- ¿Puedo elegir cómo se organizan sus datos (por filas o por columnas)?
- ¿Están todos sus datos contiguos en la memoria?


Definición de arreglos en C

- **Alternativa 3 (arreglo dinámico como vector de punteros a filas/columnas):**

```
#define N 100

int main (int argc, char * argv[])
     …
     float ** matriz = malloc(N*sizeof(float*));
     for (i=0; i < N; i++)
          matriz[i] = malloc(N*sizeof(float));
     …
}
```

¿Cómo accedo a la posición [i , j] siendo *i* el número de fila y *j* el de la columna?

- ¿Tiene un tamaño máximo por la forma en que está declarado?
- ¿Puedo cambiar su tamaño en ejecución?
- ¿Puedo elegir cómo se organizan sus datos (por filas o por columnas)?
- ¿Están todos sus datos contiguos en la memoria?

Definición de arreglos en C

- **Alternativa 4 (arreglo dinámico como vector de elementos):**

```
#define N 100

int main (int argc, char * argv[])
{
    …
    float * matriz = malloc(N*N*sizeof(float));
    …
}
```

¿Cómo accedo a la posición [i , j] siendo *i* el número de fila y *j* el de la columna?

- ¿Tiene un tamaño máximo por la forma en que está declarado?
- ¿Puedo cambiar su tamaño en ejecución?
- ¿Puedo elegir cómo se organizan sus datos (por filas o por columnas)?
- ¿Están todos sus datos contiguos en la memoria?

Definición de arreglos en C

- **Alternativa 4 (arreglo dinámico como vector de elementos):**

```
#define N 100

int main (int argc, char * argv[]) {
     …
     float * matriz = malloc(N*N*sizeof(float));
     …
}
```

- Acceder a la posición [i,j] cuando *matriz*:
  - está ordenada por filas: `matriz [i*N+j]`
  - está ordenada por columnas: `matriz [j*N+i]`

Definición de arreglos en C

Arreglo dinámico como vector de elementos

- En forma general, al almacenar una matriz como un vector de elementos, el cálculo del índice para acceder a sus elementos depende de cómo estén organizados.

- Cuando una matriz está organizada por filas, se debe multiplicar el número de fila por la cantidad de elementos de cada fila (cantidad de columnas) y sumarle el número de columna.
  - En el ejemplo anterior: `matriz [i*N+j]`

- Cuando una matriz está organizada por columnas, se debe multiplicar el número de columna por la cantidad de elementos de cada columna (cantidad de filas) y sumarle el número de fila.
  - En el ejemplo anterior: `matriz [j*N+i]`

Definición de arreglos en C

Arreglo dinámico como vector de elementos

- Ventajas:
  - Favorece al aprovechamiento de la localidad de datos
  - Hace posible el uso de instrucciones SIMD
  - Facilita el intercambio de arreglos entre programas escritos en diferentes lenguajes

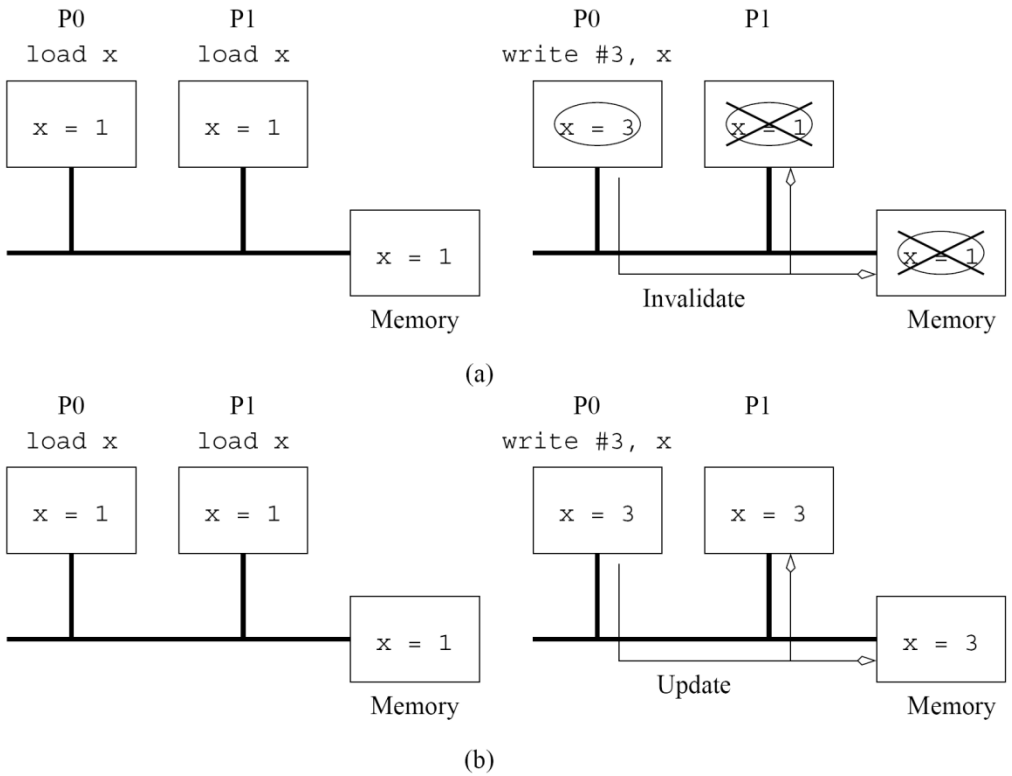
COHERENCIA DE CACHE EN ARQUITECTURAS MULTIPROCESADOR

Coherencia de caché en arquitecturas multiprocesador

- Las redes de interconexión proveen mecanismos para comunicar datos
- En máquinas de memoria compartida pueden existir múltiples copias del mismo dato → se requiere hardware específico para mantener la consistencia entre estas copias
- El mecanismo de coherencia debe asegurar que todas las operaciones realizadas sobre las múltiples copias son *serializables* → tiene que existir algún orden de ejecución secuencial que se corresponde con la planificación paralela

Coherencia de caché en arquitecturas multiprocesador

Protocolos de coherencia de caché: (a) invalidación (b) actualización

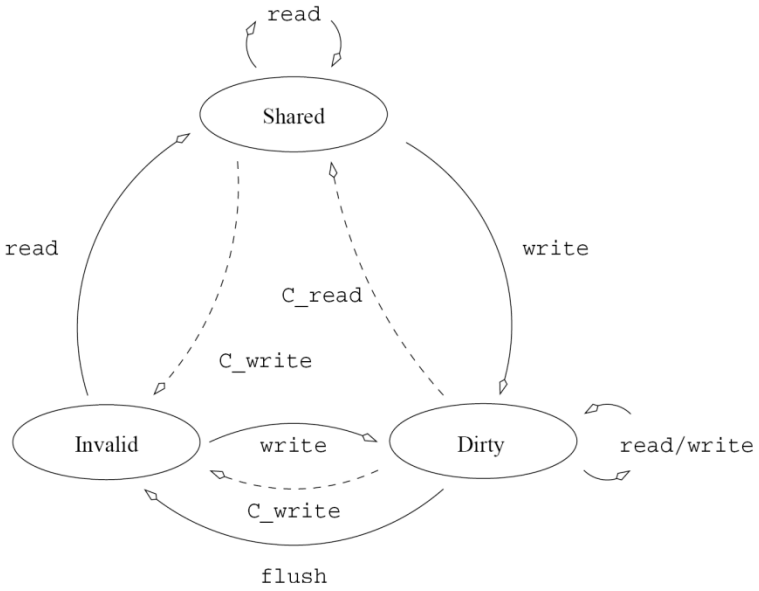


Protocolos de coherencia de caché

- Si un procesador lee un dato una vez y no vuelve a usarlo, un protocolo de actualización puede generar un overhead innecesario
  - En este caso es mejor un protocolo de invalidación
- Si dos procesadores trabajan sobre la misma variable en forma alternada, un protocolo de actualización será mejor opción
  - Se evita/reduce ocio por espera del dato actualizado
- Relación costo-beneficio: los protocolos de actualización pueden producir overhead por comunicaciones innecesarias mientras que los de invalidación pueden producir ocio ante la espera de actualización de un dato
- En la actualidad, la mayoría de los protocolos se basan en invalidación

Protocolos de coherencia de caché basados en invalidación

- Esquema simple donde cada copia se asocia con uno de 3 estados: *compartida (shared)*, *inválida (invalid)* o *sucia* (*dirty)*
- En el estado *compartido*, hay múltiples copias válidas del dato (en diferentes memorias). Ante una escritura, pasa a estado *sucia* donde se produjo mientras que el resto se marca como *inválida.*
- En el estado *sucia*, la copia es válida y se trabaja con esta.
- En el estado *inválida*, la copia no es válida. Ante una lectura, se actualiza a partir de la copia válida (la que está en estado *sucia*)



Protocolos de coherencia de caché basados en invalidación: ejemplo

Time ↓

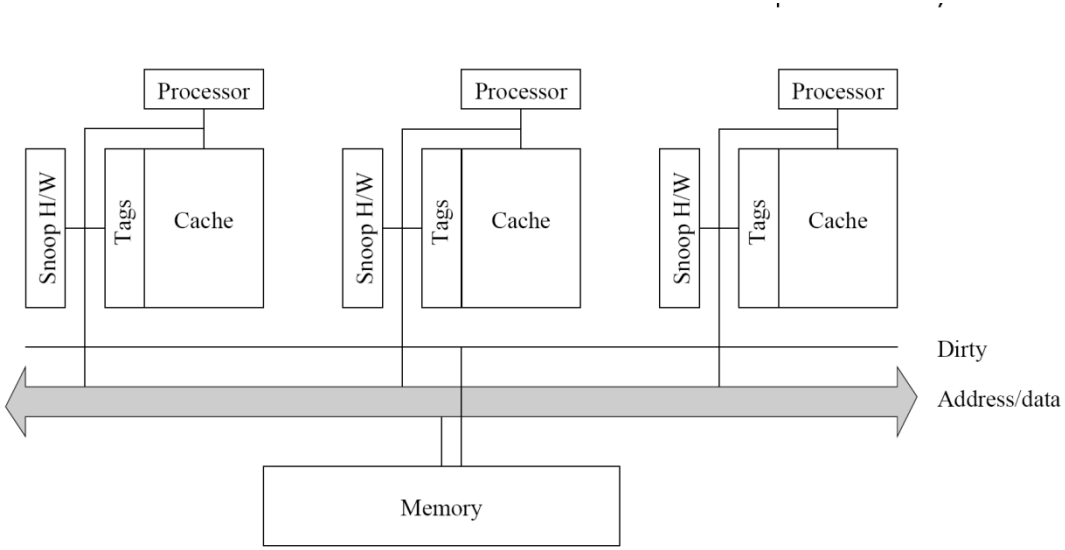
Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
				x = 5, D y = 12, D
read x		x = 5, S		x = 5, S
	read y		y = 12, S	y = 12, S
x = x + 1		x = 6, D		x = 5, I
	y = y + 1		y = 13, D	y = 12, I
read y		y = 13, S	y = 13, S	y = 13, S
	read x	x = 6, S	x = 6, S	x = 6, S
x = x + y		x = 19, D	x = 6, I	x = 6, I
	y = x + y	y = 13, I	y = 19, D	y = 13, I
x = x + 1		x = 20, D		x = 6, I
	y = y + 1		y = 20, D	y = 13, I

Implementación de protocolos de coherencia de caché

- Existe una variedad de mecanismos de hardware para implementar protocolos de coherencia de caché:
  - Sistemas *snoopy*
  - Sistemas basados en directorios
  - Combinaciones de los dos anteriores

Implementación de protocolos de coherencia de caché: Sistemas de caché Snoopy

- Asociado usualmente a los sistemas multiprocesador interconectados con alguna red *broadcast*, como bus o anillo.
- La caché de cada procesador mantiene un conjunto de tags asociados a sus bloques, los cuales determinan su estado.
- Todos los procesadores monitorizan (*snoop*) el bus, lo que permite realizar las transiciones de estado de sus bloques:
  - Cuando el hw snoopy detecta una lectura sobre un bloque de caché marcado como *sucio*, entonces toma el control del bus y cumple el pedido.
  - Cuando el hw snoopy detecta una escritura sobre un bloque de datos del cual tiene copia, entonces la marca como *inválida*.



Implementación de protocolos de coherencia de caché: Sistemas de caché Snoopy

- Ampliamente estudio y usado en sistemas comerciales, por ser un esquema simple, de bajo costo y de buen rendimiento para operaciones locales.
- Si cada procesador opera sobre datos disjuntos, entonces los mismos pueden ser *cacheados:*
  - Ante operaciones de escritura, el dato es marcado como *sucio*. Al no haber operaciones de otros procesadores, las siguientes peticiones se satisfacen localmente.
  - Ante operaciones de lectura, el dato es marcado como *compartido* (aun cuando sean varios procesadores). Las peticiones siguientes se satisfacen localmente en todos los casos.
  - En ambos casos, el protocolo no agrega overhead adicional.

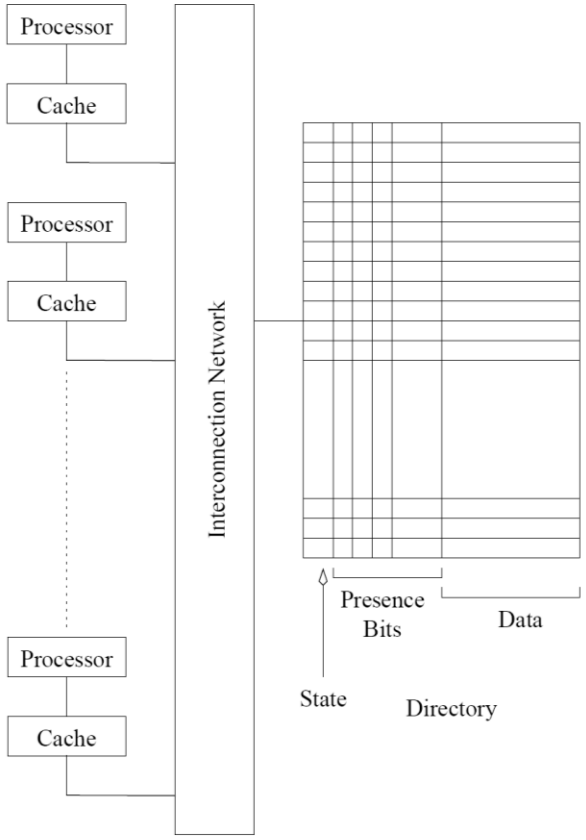
Implementación de protocolos de coherencia de caché: Sistemas de caché Snoopy

- Si diferentes procesadores realizan lecturas y escrituras sobre el mismo dato, se genera tráfico en el bus para poder mantener la coherencia de los datos.
  - Tener en cuenta que al basarse en redes *broadcast*, el mensaje de coherencia les llegará todos los procesadores, aun cuando no tengan el dato en cuestión.
  - Como el bus a su vez tiene un ancho de banda limitado, se convierte en un **cuello de botella**

- Una solución obvia a este problema consiste en sólo propagar las operaciones de coherencia a aquellos procesadores que tienen el dato involucrado, lo cual requiere mantener un registro de qué datos tiene cada procesador → *Sistemas basados en directorios*

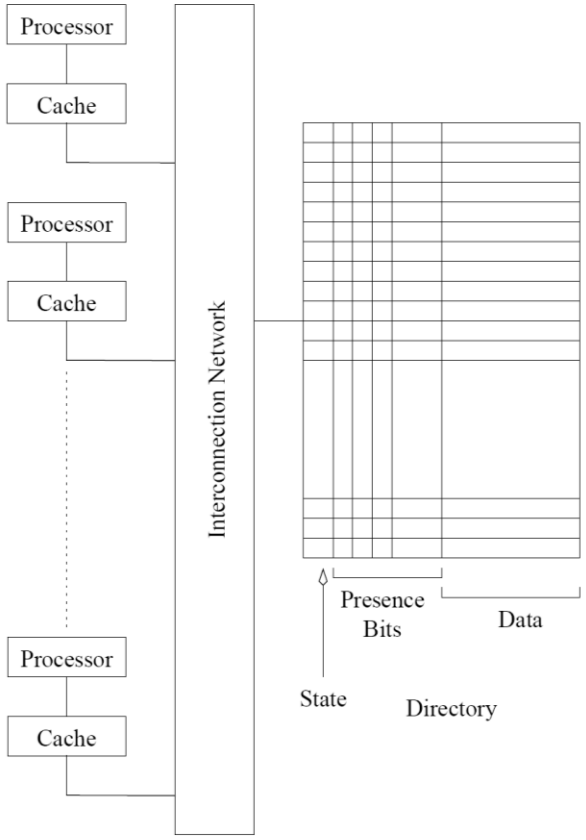
Implementación de protocolos de coherencia de caché: Sistemas basados en directorios

- La memoria principal incorpora un *directorio* que mantiene información de estado (bits de presencia + estado) sobre los bloques de caché y los procesadores donde están *cacheados.*
- La información contenida en el directorio permite que sólo aquellos procesadores que tienen un determinado dato queden involucrados en las operaciones de coherencia.
- Al igual que con Snoopy, si los procesadores operan sobre datos disjuntos, las peticiones pueden cumplirse localmente (no agrega overhead).



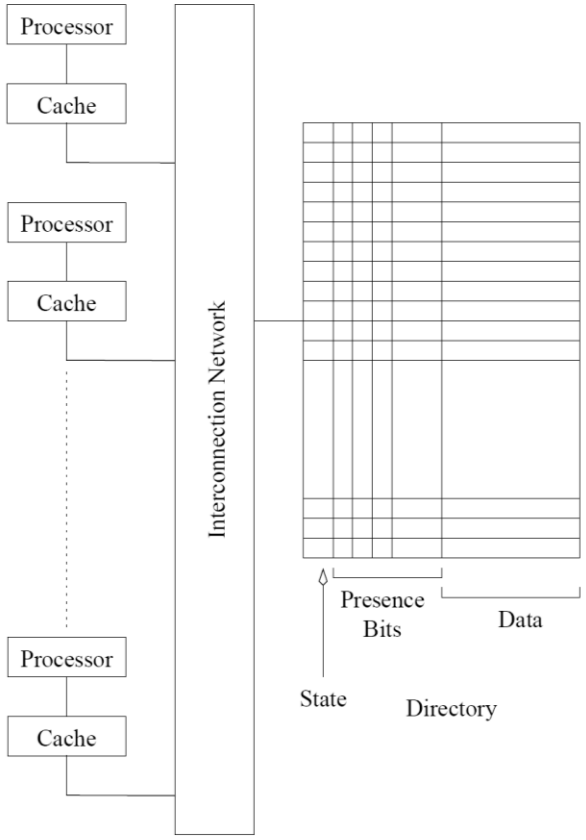
Implementación de protocolos de coherencia de caché: Sistemas basados en directorios

- Cuando múltiples procesadores leen y escriben los mismos datos, se generan operaciones de coherencia → provoca overhead adicional por la necesidad de mantener actualizado el directorio.
  - Como el directorio está en memoria, si un programa paralelo requiere a un gran número de operaciones coherencia, se genera overhead por la competencia en el acceso al recurso (la memoria sólo puede satisfacer un número limitado de operaciones por unidad de tiempo).
  - Además, la cantidad de memoria requerida por el directorio podría convertirse en un cuello de botella a medida que el número de procesadores crece



Implementación de protocolos de coherencia de caché: Sistemas basados en directorios

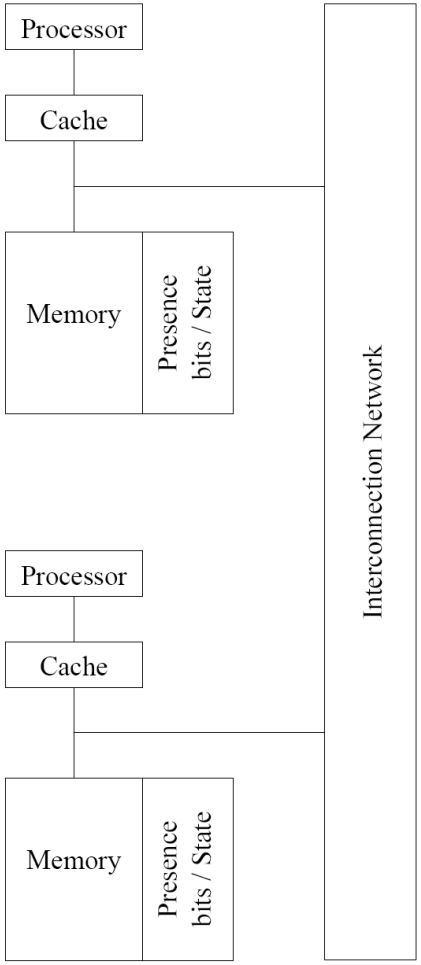
- Como el directorio representa un punto centralizado de acceso (overhead por competencia), una solución posible es particionarlo → *Sistemas basados en directorios distribuidos*



Implementación de protocolos de coherencia de caché: Sistemas basados en directorios distribuidos

- Se da en arquitecturas escalables, donde la memoria se encuentra físicamente distribuida.

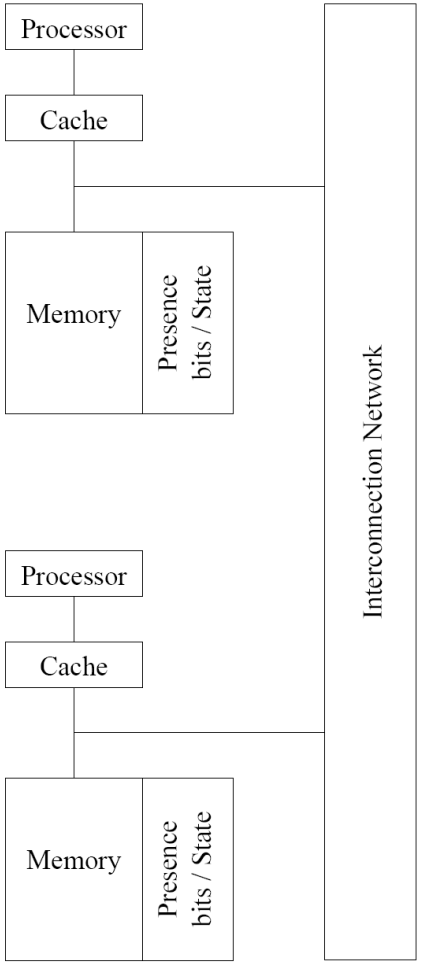
- Cada procesador es responsable de mantener la coherencia de sus propios bloques (mantiene su propio directorio).
  - Cuando un procesador desea leer un bloque por primera vez, debe pedíselo al *dueño*, quien redirige el pedido de acuerdo a la información del directorio.
  - Cuando un procesador escribe un bloque de memoria, envía una invalidación al *dueño,* quien luego la propaga a todos aquellos que tienen una copia.



Implementación de protocolos de coherencia de caché: Sistemas basados en directorios distribuidos

- Como el directorio está distribuido, la competencia en el acceso al mismo se alivia → sistema más escalable

- La latencia y el ancho de banda de la red de interconexión se convierten ahora en los cuellos de botella del rendimiento de estos sistemas



COSTOS DE COMUNICACIÓN

Costos de comunicación

- Uno de los mayores overheads en los programas paralelos proviene de la comunicación entre unidades de procesamiento
- El costo de la comunicación depende de múltiples factores y no sólo del medio físico: modelo de programación, topología de red, manejo y ruteo de datos, protocolos de software asociados.
- Costos diferentes según la forma de comunicación:
  - Modelo simplificado para pasaje de mensajes: $t_{comm} = t_s + m\ t_w$, donde t_s es el tiempo requerido para preparar el mensaje, m es el tamaño del mensaje medido en palabras (*words*) y t_w es el tiempo requerido para transmitir una palabra
  - Memoria compartida: resulta difícil modelar costos por múltiples factores que escapan al control del programador (los tiempos de acceso dependen de la ubicación del dato, el overhead de los protocolos de coherencia son difíciles de estimar, la localidad espacial es difícil de modelar, competencia generada por accesos compartidos depende de la planificación en ejecución, entre otros).

Bibliografía usada para esta clase

- Capítulo 2, “Introduction to Parallel Computing”. Grama, Gupta, Karypis, Kumar. Addison Wesley (2003).

- Capítulo 1, “Introduction to High Performance Computing for Scientists and Engineers”. Georg Hager, Gerard Wellein. CRC Press (2011).

- “Introduction to parallel computing” Blaise Barney, Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/parallel_comp