Parallelism (PAR)

Unit 4: Introduction to parallel architectures (or in other words, where the data sharing overheads come from?)

Josep Ramon Herrero (T10), Gladys Utrera (T20), Jordi Tubella (T40), Eduard Ayguadé and Daniel Jiménez

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2024/25 (Fall semester)

Learning material for this lesson

- Atenea: Unit 4.1 Introduction to parallel architectures I
  - Video lesson 5: UMA architectures
  - Quizzes after different parts in video lesson 5
- Atenea: Unit 4.2 Introduction to parallel architectures II
  - Video lesson 6: NUMA architectures
  - Quizzes after different parts in video lesson 6
- These slides to dive deeper into UMA and NUMA architectures
- Collection of Exercises: problems in Chapter 4

Outline

Uniprocessor parallelism

Symmetric multi–processor architectures
- Video lesson 5
- Hardware support for coherence (I)
- Who is causing coherence traffic? true and false sharing

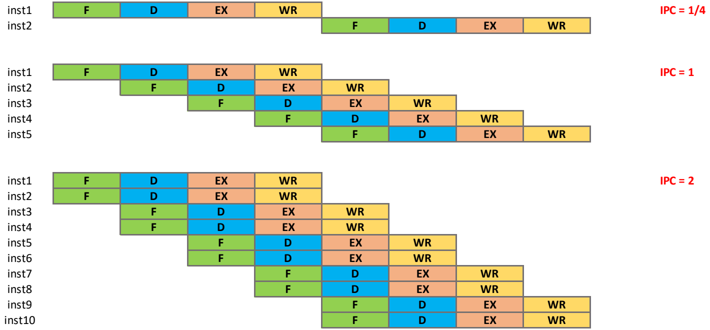
Multicore architectures

Non-Uniform Memory Architectures
- Video lesson 6
- Hardware support for coherence (II)
- Who is causing coherence traffic? data initialization

Hardware support for synchronization

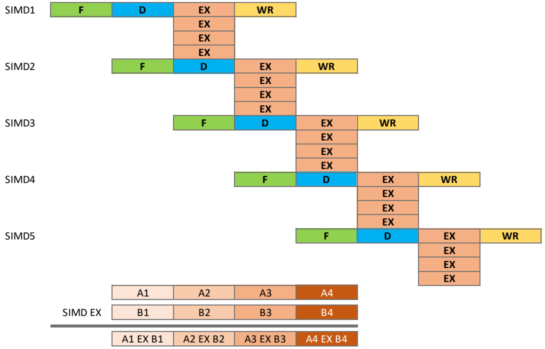
Pipelined and superscalar architecture

- Execution of single instruction divided in multiple stages
- Overlap the execution of different stages of consecutive instructions (pipelined) and consecutive instructions (superscalar)



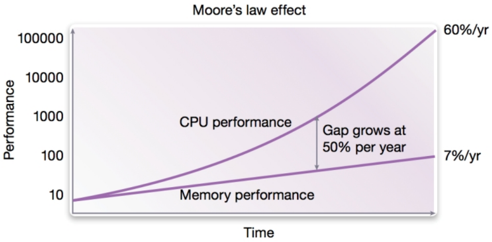
SIMD architecture

- DLP (data-level parallelism): Single-Instruction executed on Multiple-Data (SIMD): vector functional unit

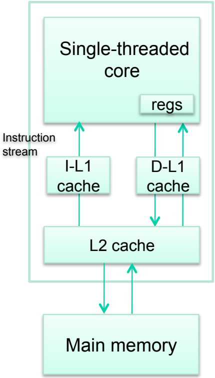


Memory hierarchy

- Addressing the yearly increasing gap between CPU cycle and memory access times



- Size vs. access time



- Non-blocking design

Memory hierarchy

- The principle of locality: if an item is referenced ...
  - Temporal locality: ... it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial locality: ... items whose addresses are close by tend to be referenced soon (e.g., straight line code, array access)
- Line (or block)
  - A number of consecutive words in memory (e.g. 32 bytes, equivalent to 4 words x 8 bytes)
  - Unit of information that is transferred between two levels in the hierarchy
- On an access to a level in the hierarchy
  - Hit: data appears in one of the lines in that level
  - Miss: data needs to be retrieved from a line in the next level

Elements of cache design

- Organization
  - Single vs. multilevel cache, Unified vs. split (instruction/data)
  - Cache size and line size
  - Addressing: logical vs. physical
- Placement algorithm
  - Direct, associative, set associative
- Replacement algorithm
  - Random, Least Recently Used (LRU), First-in First-out (FIFO), Least Frequently Used (LFU)
- Write (on hit) Policy
  - Write-through, Write-back
- Write (on miss) Policy
  - Write-allocate, write-no-allocate

Who exploits this uniprocessor parallelism and memory organization?

In theory, the compiler understands all of this ... but in practice the compiler may need your help, for example:

- Software pipelining to statically schedule ILP
- Vectorization to efficiently exploit SIMD vector units
- Data contiguous in memory and aligned to cache lines
- Blocking (or tiling) to define a problem that fits in register/L1-cache/L2-cache (temporal locality)

Reasons and techniques explored in detail in PCA course (Architecture-Conscious Programming)

Outline

Uniprocessor parallelism

Symmetric multi–processor architectures

- Video lesson 5
- Hardware support for coherence (I)
- Who is causing coherence traffic? true and false sharing

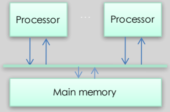
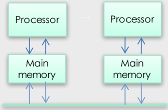
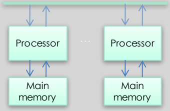
Multicore architectures

Non-Uniform Memory Architectures

- Video lesson 6
- Hardware support for coherence (II)
- Who is causing coherence traffic? data initialization

Hardware support for synchronization

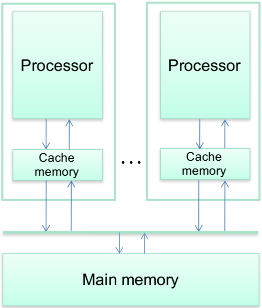
Classification of multi–processor architectures

Memory architecture	Address space(s)	Connection	Model for data sharing	Names
(Centralized) Shared-memory architecture	Single shared address space, uniform access time		Load/store instructions from processors	• SMP (Symmetric Multi-Processor) architecture • UMA (Uniform Memory Access) architecture
Distributed-memory architecture	Single shared address space, non-uniform access time		Load/store instructions from processors	• DSM (Distributed-Shared Memory architecture • NUMA (Non-Uniform Memory Access) architecture
	Multiple separate address spaces		Explicit messages through network interface card	• Message-passing multiprocessor • Cluster Architecture • Multicomputer

Outline

Concepts in video lesson 5

- Centralised shared-memory architectures, also called UMA (Uniform Memory Access time) or SMP (Symmetric) multiprocessors
- Cache coherence problem in centralised shared-memory architectures
  - Programmer vs. hardware views
  - Two snooping-based solutions to cache incoherence: write–update vs. write invalidate



Concepts in video lesson 5 (cont.)

The coherence problem:

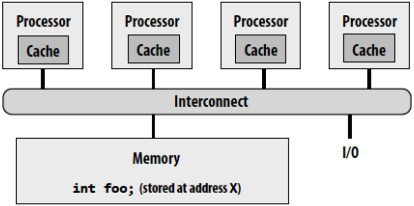


Chart shows value of foo (variable stored at address X) stored in main memory and in each processor's cache **

** Assumes write-back cache behavior

Action	P1 \$	P2 \$	P3 \$	P4 \$	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (say this load causes eviction of foo)		0	2		1

(CMU 15-418, Spring 2012)

Concepts in video lesson 5 (cont.)

Coherence protocols:

- Write-update: writing processor broadcasts **the line** with the new value and forces all others to update their copies
- Write-invalidate: writing processor forces all others to invalidate their copies; **the line** with the new value is provided to others when requested or when flushed from cache

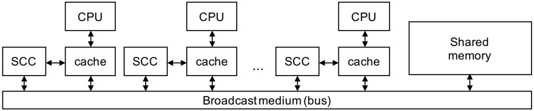
Coherence mechanisms:

- Broadcast-based (snooping): bus serves as broadcast mechanism to maintain coherency among copies of the same memory line in caches
- Directory-based: the sharing status of each line in memory is kept centralised in just one location (directory)

Outline

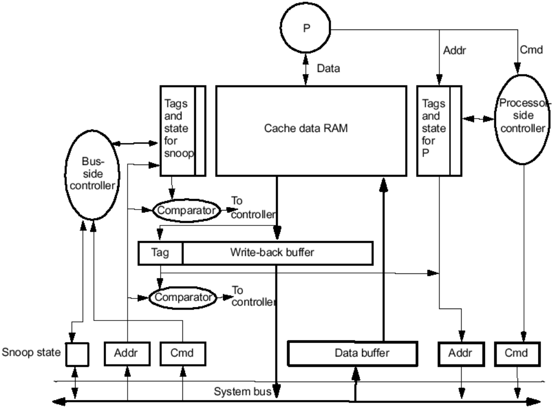
Broadcast-based (snooping) coherence mechanism

- Cache coherence is maintained at **cache line granularity**, NOT at the individual words inside the cache line
- Every cache that has a copy of a line from physical memory keeps its sharing status (status distributed)
- Broadcast medium (e.g. a bus) used to make all transactions visible to all caches and define **ordering**
- Caches monitor (snoop on) the medium and take action on relevant events (SCC: snoopy cache controllers)



Dual-ported caches to support coherence (optional)

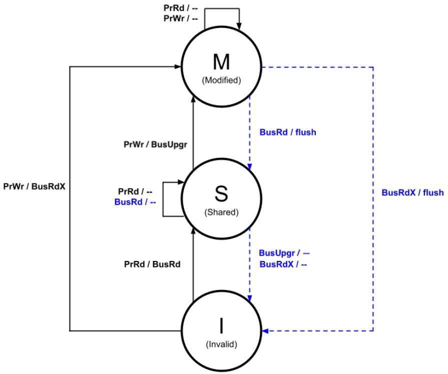
Listen to commands both from processor and from broadcast medium (e.g. bus)



Simple write-invalidate snooping protocol (MSI)

- A line in a cache memory can be in three different states:
  - Modified (M): dirty copy of the line
  - Shared (S): clean copy of the line
  - Invalid (I): invalidated copy of the line (not valid), or it does not exist in cache
- CPU events
  - PrRd (Processor read)
  - PrWr (Processor write)
- Bus events (caused by cache controllers)
  - BusRd: asks for copy with no intent to modify
  - BusRdX: asks for copy with intent to modify
  - BusUpgr: asks for permission to modify existing line, causes invalidation of other copies
  - Flush: puts line on bus, either because requested or voluntarily when dirty line in cache is replaced (WriteBack)

Simple write-invalidate snooping protocol (MSI)



- Who provides the line when requested via BusRd or BusRdX?
  - If line in S or I in other caches then main memory provides it
  - If line in M in another cache then this cache provides it (Flush)

MSI optimizations. Thread–private lines

- MSI requires two bus transactions for the common case of read followed by write, both from the same processor (no sharing at all)
  - Transaction 1: BusRd to move from I to S state
  - Transaction 2: BusUpgr to move from S to M state
- **MESI** protocol adds E (Exclusive) clean state:
  - Cache line in E if only one clean copy of the line (Snoop State bit)
  - If write access by the same processor, the upgrade from E to M does not require a bus transaction (BusUpgr)
  - If line in E and another cache requests it then cache line state changes from E to S

Optional: MSI optimizations. Cache–to–cache transfers

- Does main memory need to be updated when flushing? **MOSI** protocol adds O (Owned) state:
  - When flushing, state in cache for the line transitions from M to O (dirty since main memory is not updated)
  - Cache with line in O state is responsible for providing data when requested (not main memory)
  - Other caches maintain shared line in S state
  - Main memory updated when line in O is replaced from cache

Optional: MSI optimizations. Cache–to–cache transfers

- Does main memory need to supply data if already shared in another cache? **MSIF** protocol adds F (Forward) state:
  - Which cache should provide the line if several copies?
  - Cache with line in F state is responsible for providing data when requested (not main memory)
  - Last cache asking for line transitions to F state (temporal locality), others transition/keep it S
- Combined use possible (MESIF/MOESI/MOESIF)

Outline

True vs. false sharing

- True sharing
  - Data sharing is unavoidable in parallel computing. Coherence mechanisms are there to allow this data sharing; synchronization allows to share appropriately.
- False sharing
  - Cache line may also introduce artefacts: more that 1 (distinct) data object, or also multiple elements of same object, may reside in the same cache line
  - False sharing occurs when different processors make references (read and write) to those different objects or elements within the same cache line, thereby inducing "unnecessary" coherence operations.

True sharing example

Assume each task is executing an instance of the following dot_product function:

```
int result = 0;
void dot_product(int *A, int *B, int n) {
   for (int i=0; i< n; i++)
       #pragma omp atomic
       result += A[i] * B[i];
}
```

The line containing variable result is subject to coherence actions at each iteration of i. It could be easily transformed into

```
void dot_product(int *A, int *B, int n) {
   int tmp = 0;
   for (int i=0; i< n; i++)
       tmp += A[i] * B[i];
    #pragma omp atomic
    result += tmp;
}
```

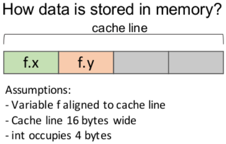
to reduce cache coherence traffic. **Note:** atomic is used to guarantee exclusive access to variable result

False sharing example

```
struct foo {
  int x, y; //x and y will reside in same cache line
} f; // aligned to cache line

void main() {
  int s=0;
  #pragma omp parallel
  #pragma omp single
  {
      #pragma omp task shared(s)
      for (int i=0;i<1000000;i++)
        s+=f.x

      #pragma omp task
      for (int i=0;i<1000000;i++)
        f.y++
  }
}
```



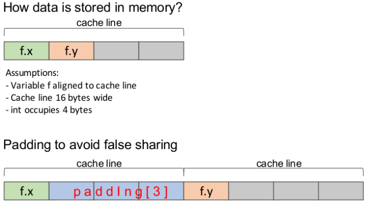
False sharing of the line containing fields x and y. How could we force fields x and y to be in different cache lines?

False sharing example (cont.)

```
struct foo {
  int x;
  int padding[3];
  int  y; //x and y will NOT reside in same cache line
} f; // aligned to cache line

void main() {
  int s=0;
  #pragma omp parallel
  #pragma omp single
  {
      #pragma omp task shared(s)
      for (int i=0;i<1000000;i++)
        s+=f.x

      #pragma omp task
      for (int i=0;i<1000000;i++)
        f.y++
  }
}
```



Add a dummy field in between to separate fields x and y in different cache lines: `int padding[PAD_SIZE];` How much padding?

Outline

Uniprocessor parallelism

Symmetric multi–processor architectures

- Video lesson 5
- Hardware support for coherence (I)
- Who is causing coherence traffic? true and false sharing

Multicore architectures

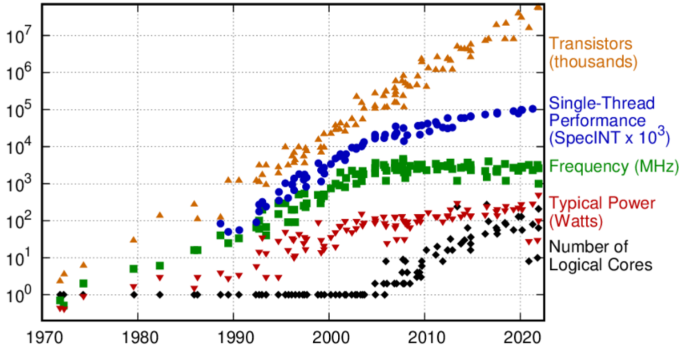
Non-Uniform Memory Architectures

- Video lesson 6
- Hardware support for coherence (II)
- Who is causing coherence traffic? data initialization

Hardware support for synchronization

Transistors, frequency, power, performance and ... cores!

An inflexion point in 2004 ... the power wall

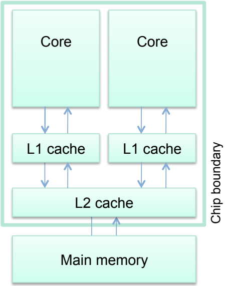


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

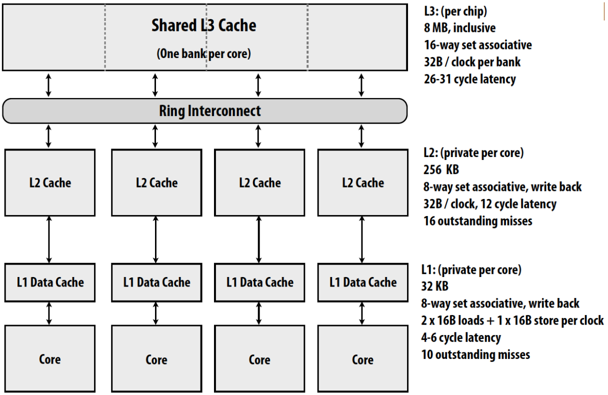
Multicores

- The increasing number of transistors on a chip is used to accommodate multiple processors (cores) on a single chip
- Usually private caches (up to a certain cache level) and one last-level cache (LLC)
- Coherence maintained at the LLC level
- Chip or socket boundary, access to main memory

- Multicore = Chip Multi-Processor (CMP)

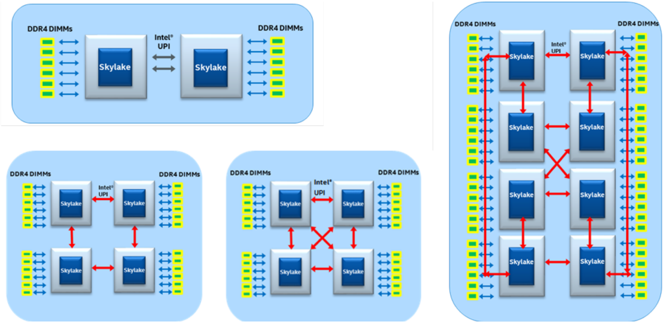


Example: multicore socket based on Intel Nehalem i7



Example: scalable multi-socket systems

Each socket is a multicore processor with a number of cores inside (e.g. SKL up to 24) and connected to memory (DDR DIMMs)



UPI/QPI ports to interconnect sockets and provide cache-coherent shared memory (but not uniform access time anymore!)

Outline

Uniprocessor parallelism

Symmetric multi–processor architectures

- Video lesson 5
- Hardware support for coherence (I)
- Who is causing coherence traffic? true and false sharing

Multicore architectures

Non-Uniform Memory Architectures

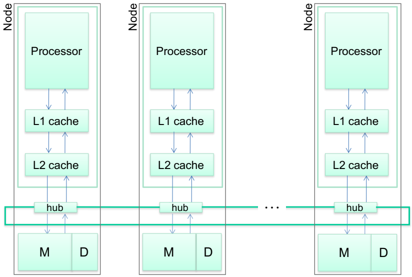
- Video lesson 6
- Hardware support for coherence (II)
- Who is causing coherence traffic? data initialization

Hardware support for synchronization

Outline

Concepts in video lesson 6

- NUMA architecture
  - Main memory distributed across multiple nodes
  - Non-uniform memory access
- Directory to keep track of the status of memory lines
  - Directory divided into "slices", one slice per node
  - Each slice serves the memory lines in the node, one entry per memory line
- Directory entry: clean/dirty line, list of sharer nodes



Outline

Uniprocessor parallelism

Symmetric multi–processor architectures
- Video lesson 5
- Hardware support for coherence (I)
- Who is causing coherence traffic? true and false sharing

Multicore architectures

Non-Uniform Memory Architectures
- Video lesson 6
- Hardware support for coherence (II)
- Who is causing coherence traffic? data initialization

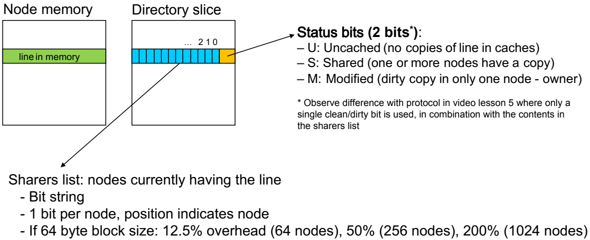
Hardware support for synchronization

Scaling of the broadcast mechanism

- Snooping schemes broadcast coherence messages to determine the state of a line in the other caches
  - Processor initiating access sends command to ALL other processors (having or not copy of the line)
  - Could be extended to support coherence in small NUMA systems, but does not scale to large number of nodes (excessive coherence traffic)
- Alternative: avoid broadcast by storing information about the status of each line in main memory, in the so called directory divided in slices, one slice per node
  - Each slice of the directory tracks the location of copies in caches of its memory lines
  - Coherence is maintained by point-to-point messages between the nodes

MSU directory-based cache coherency

- One slice of the directory associated to each node memory: one entry per line of memory
  - **Status bits**: they track the state of cache lines in its memory
  - **Sharers list**: tracks the list of remote nodes having a copy of a line. For small-scale systems, implemented as a bit string



- Directory slice is the "centralised" structure that "orders" the accesses to the lines in the associated node

Directory–based cache coherency (cont.)

- Who is involved in maintaining coherence of a memory line?
  - **Home** node: node where the line is allocated. It has the directory slice with the information to maintain its coherence.
  - **Local** node: node with the processor accessing the line
  - **Remote** nodes: **Owner** node containing **dirty** copy or **Reader** nodes containing **clean** copies of the line
- But ... how the **home** node for a memory line is decided?
  - **OS managed**, for example using a policy named *first touch*
    - The node that first "touches" a **page** will be the **home** node for all the lines in that page
    - For example, if memory pages are $P = 4$ KBytes and memory lines are $L = 128$ Bytes, then a page will contain $P/L = 32$ consecutive memory lines
    - Unless indicated differently we will assume that the number of memory lines in a page is 1 (i.e. $P = L$)

Simplified coherency protocol

Possible commands arriving to home node from local node:

- **RdReq**: asks for copy of line with no intent to modify
- **WrReq**: asks for copy of line with intent to modify
- **UpgrReq**: asks for permission to modify an existing line, invalidating all other copies

As a result of **RdReq** and **WrReq** the home node sends clean copy of line (**Dreply** command to local node). For **UpgrReq** it sends an acknowledgment (**Ack** command) to give permission.

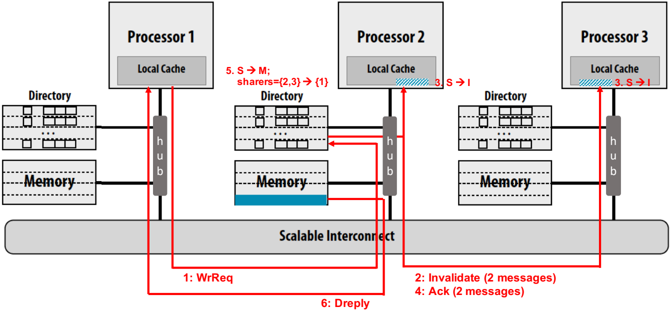
If needed the home node may generate other commands to remote nodes:

- **Fetch**: asks remote (owner) node for a copy of line (**Dreply**)
- **Invalidate**: asks remote (reader) node to invalidate its copy, remote sends confirmation to home (**Ack**)

Directory-based cache coherency: example

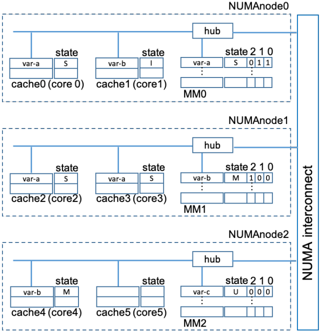
Write miss to clean line with two sharers

- Local node where the miss request originates: processor 1
- Home node where the memory line resides: processor 2
- Copies of line in caches of remote processors 2 and 3



Snooping- and directory-based protocols together!

If nodes have snoopy–based coherence, then the hub becomes an additional agent that interacts with the home (directory) nodes for the cache lines copied in the node



Coherence commands

- **Core:** $PrRd_i$ and $PrWr_i$, being I the core number doing the action
- **Snoopy:** $BusRd_j$, $BusRdX_j$ $BusUpgr_j$ and $Flush_j$, being j the snoopy/cache number doing the action
- **Hub/directoty:** $RdReq_{i \to j}$, $WrReq_{i \to j}$, $UpgrReq_{i \to j}$, $Dreply_{i \to j}$, $Fetch_{i \to j}$, $Invalidate_{i \to j}$, $Ack_{i \to j}$ and $WriteBack_{i \to j}$, from NUMAnode i to NUMAnode j

Line state in cache

- M (Modified), S (Shared), I (Invalid)

Line state in main memory

- M (Modified), S (Shared), U (Uncached)

Outline

- Uniprocessor parallelism
- Symmetric multi–processor architectures
  - Video lesson 5
  - Hardware support for coherence (I)
  - Who is causing coherence traffic? true and false sharing
- Multicore architectures
- Non-Uniform Memory Architectures
  - Video lesson 6
  - Hardware support for coherence (II)
  - Who is causing coherence traffic? data initialization
- Hardware support for synchronization

Data sharing and initialization

- True and false sharing have now a much higher penalty
- What may be wrong with data initialization? Be aware of "first touch"

```
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

```
#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

Data sharing and initialization

```
#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++)
    b[i] = foo1(a[i]);
  for (int i=myid*BS; i<(myid+1)*BS; i++)
    a[i] = foo2(b[i]);
}
```

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

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

Outline

Uniprocessor parallelism

Symmetric multi–processor architectures

- Video lesson 5
- Hardware support for coherence (I)
- Who is causing coherence traffic? true and false sharing

Multicore architectures

Non-Uniform Memory Architectures

- Video lesson 6
- Hardware support for coherence (II)
- Who is causing coherence traffic? data initialization

Hardware support for synchronization

How are synchronizations implemented?

```
#pragma omp atomic
var += non_protected_func();
```

```
#pragma omp critical
{
    // exclusive access
}
```

```
omp_set_lock(&lock_var);
// exclusive access
omp_unset_lock(&lock_var);
```

- In fact, entry to and exit from `critical` is the same as `omp_set_lock` and `omp_unset_lock`, respectively, but using an implicit (hidden) `omp_lock_t` variable
- `atomic` could also be implemented with `omp_lock_t`, but usually there is much better support at the architecture level (see later ...)
- How to implement lock-based synchronisation mechanisms?

Example: a simple, but incorrect, lock

- What's wrong with ...?
  (assume flag=0 means lock is free; taken otherwise)

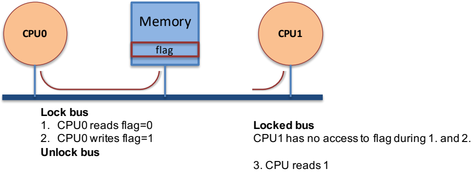
```
                CPU0                                  CPU1

                // omp_lock_t flag                    // omp_lock_t flag
init_lock:      st flag, #0           init_lock:      st flag, 0
                ...                                   ...
set_lock:       ld r1, flag           set_lock:       ld r1, flag
                bnez r1, set_lock                     bnez r1, set_lock
                st flag, #1                           st flag, #1
                ...                                  ...
                // safe access                        // safe access
                ...                                  ...
unset_lock:     st flag, #0           unset_lock:     st flag, #0
                ...                                   ...
```

- Problem: data race because sequence load–test–store is not atomic!

Support for synchronization at the architecture level

- Need hardware support to guarantee atomic (indivisible) instruction to fetch and update memory



- `test-and-set`: read value in location and set to 1
  Example: test-and-set based lock implementation

```
set_lock:    t&s r2, flag
             bnez r2, set_lock     // already locked?
             ...
unset_lock: st flag, #0            // free lock
```

Support for synchronization at the architecture level

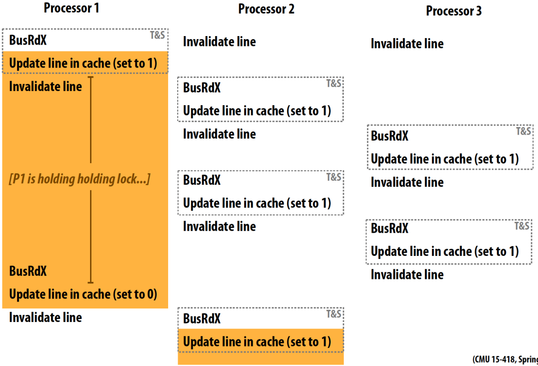
- Atomic exchange: interchange of a value in a register with a value in memory
  Example: atomic exchange based lock implementation

```
            mov r2, #1
set_lock:   exch r2, flag        // atomic exchange
            bnez r2, set_lock    // already locked?
            ...
unset_lock: st flag, #0          // free lock
```

- `fetch-and-op`: read value in location and replace with result after simple arithmetic operation (usually add, increment, sub or decrement). **Valid to implement** `#pragma omp atomic`

test-and-set lock coherence traffic

```
set_lock:   t&s r2, flag          // test and acquire lock if free
            bnez r2, set_lock     // do it again if already locked
            ...
unset_lock: st flag, #0           // free the lock
```



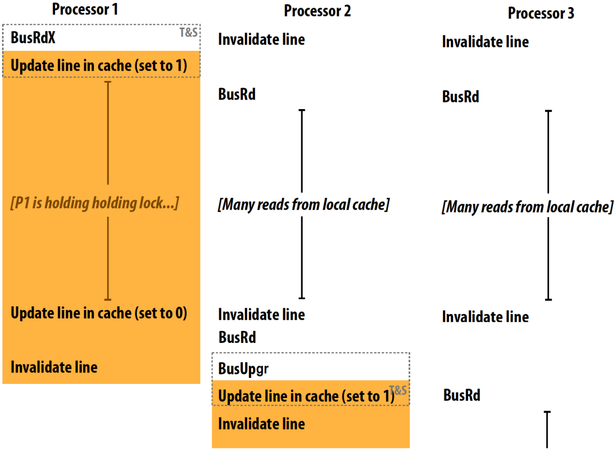
(CMU 15-418, Spring 2012)

Reducing synchronization cost: `test-test-and-set`

- `test-test-and-set` technique reduces the necessary memory bandwidth and coherence protocol operations required by a pure `test-and-set` based synchronization:
  - Wait using a regular load instruction (lock will be cached)
  - When lock is released, try to acquire using `test-and-set`

```
set_lock:   ld r2, flag             // test with regular load
                                    // lock is cached meanwhile it is not updated
            bnez r2, set_lock       // test if the lock is free
            t&s r2, flag            // test and acquire lock if STILL free
            bnez r2, set_lock
            ...
unset_lock: st flag, #0             // free the lock
```

test-test-and-set lock coherence traffic



Support for synchronization at the architecture level

- Atomicity difficult or inefficient in large systems. Alternative: **Load-linked Store-conditional** `ll-sc`
  - `ll` returns the current value of a memory location
  - `sc` stores a new value in that memory location if no updates have occurred to it since the `ll`; otherwise, the store fails
  - `sc` returns success (1) or failure (0)
- Examples implementing atomic exchange (left) and fetch-and-increment (right):

```
// exchange r4 with location.
try: mov r3, r4
     ll r2, location
     sc r3, location
     beqz r3, try
     mov r4, r2
```

```
// add 1 to location
try: ll r2, location
     add r3, r2, #1
     sc r3, location
     beqz r3, try
```

Reducing synchronization cost: `test-test-and-set`

- `test-test-and-set` technique can also be implemented with `ll-sc`
  - First, wait using load linked instruction `ll` (lock will be cached)
  - Second, use store conditional `sc` operation to test if someone else did it first

```
set_lock:   ll r2, flag           // first test with load linked
                                 // lock is cached meanwhile it is not updated
            bnez r2, set_lock     // test if the lock is free
            mov r2, #1
            sc r2, flag           // try to store 1
            beqz r2, set_lock     // repeat if someone else did it before me
            ...
unset_lock: st flag, #0           // free the lock
```

Other synchronization primitives

- How to implement a `barrier` synchronization primitive?
  - Threads arriving wait until all have reached the barrier
  - Structure with fields {lock, counter, flag}

```
barrier:
      acquire_lock(&barr.lock);
      if (barr.counter == 0)
         barr.flag = 0              // reset flag if first
      mycount =  barr.counter++;
      release_lock(&barr.lock);

      if (mycount == P) {           // last to arrive?
         barr.counter = 0           // reset counter for next barrier
         barr.flag = 1              // release waiting processors
      } else
         while (barr.flag == 0)     // busy wait for release
      ...
```

- Does it work when consecutive barriers appear? Try to solve it

Parallelism (PAR)

Unit 4: Introduction to parallel architectures
(or in other words, where the data sharing overheads come from?)

Josep Ramon Herrero (T10), Gladys Utrera (T20),
Jordi Tubella (T40), Eduard Ayguadé and Daniel Jiménez

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2024/25 (Fall semester)