University of Southern California

Viterbi School of Engineering

EE352

Computer Organization and Architecture

Parallelism and Memory Systems

References:

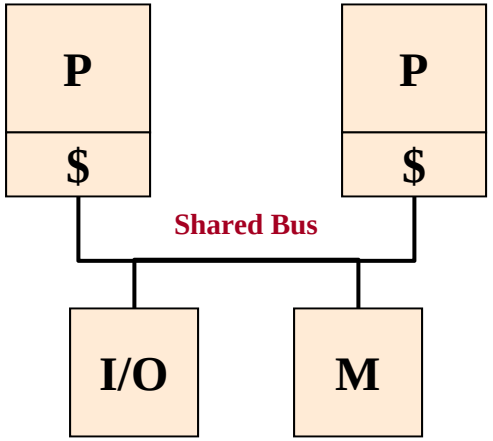
1) **Textbook**
2) **Mark Redekopp's slide series**

Shahin Nazarian

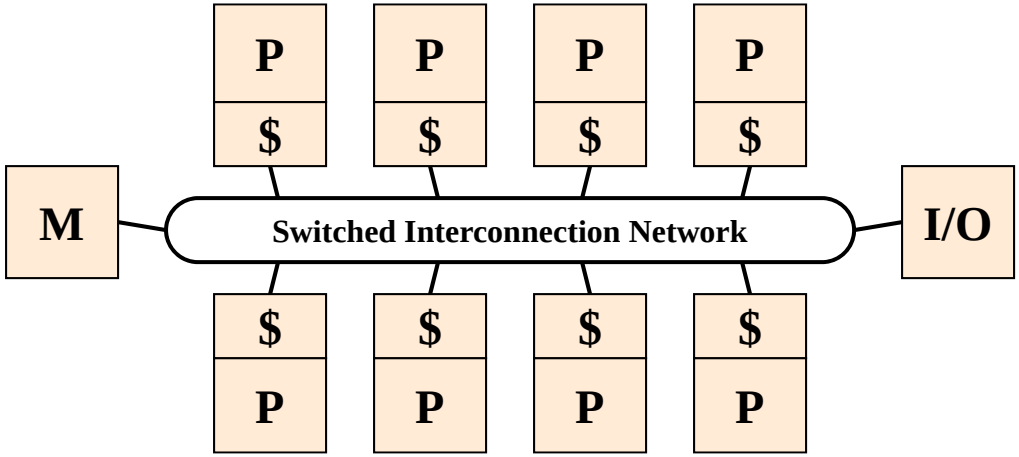
Spring 2010

Chip Multiprocessor Organization

- Processors each have their own L1 caches (always access data from these caches)
- Shared L2 / main memory
- Each processor runs a separate thread = PC/code + registers + stack
- Interconnection network for communicating data between caches
  - Shared Bus allows only one transfer at a time
  - Switched interconnection network allows multiple simultaneous transfers



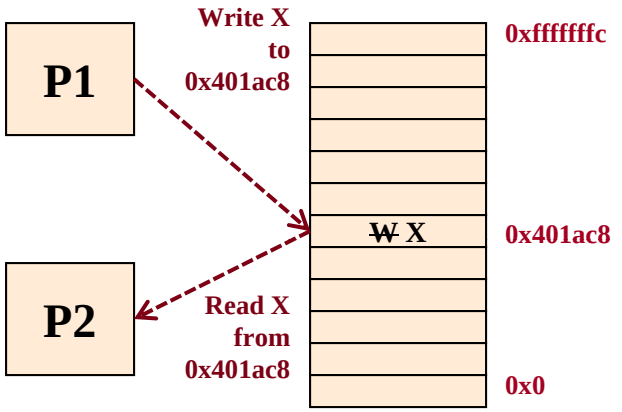
Simple CMP (Chip Multiprocessor) connected via shared bus (Core 2 Duo)



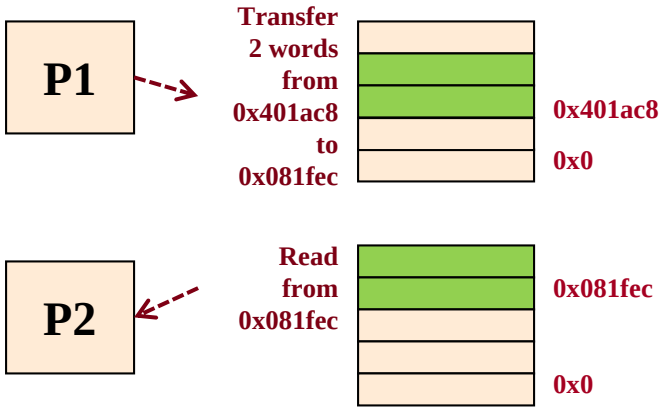
More cores usually requires a more advanced interconnection network (Cell Processor)

Communication Paradigms

- **Shared Memory** (common for multicores)
  - Each processor shares the same memory via a shared address space
  - Communication is transparently (implicitly) handled by the HW. SW program just has to perform normal loads and stores to shared memory locations
  - Coherence of data becomes an issue
- **Message Passing**
  - Each processor has its own memory/address space
  - Communication must be explicitly defined by the SW program/processor (either by DMA [Direct Mem. Access] or some other mechanism)



Shared Memory



Message Passing

Message Passing

- Communication is via explicit message passing
- If the message sender needs confirmation that the message has arrived, the receiving processor can send an acknowledgement message back to the sender

- **Clusters** are the most widespread example of message-passing parallel computers. They are generally collections of commodity computers that are connected to each other over their I/O interconnect via standard network switches and cables
- Each computer runs a distinct copy of the operating system. Virtually every Internet service relies on clusters of commodity servers and switches

Cluster Issues

Cost of administrating

- The cost of a cluster with n machines is almost equal to the cost of administrating n independent machines, whereas the cost of administrating a shared memory multiprocessor with n processors is about the same as administrating a single machine!

I/O Interconnect

- The cluster processors are typically connected using I/O interconnect of each computer, whereas the cores in a multiprocessor are usually connected on the memory interconnect of the computer. The memory interconnect has higher bandwidth and lower latency (i.e., higher communication performance)

Memory division

- A cluster of n computers has n independent memories and n copies of operationg system, but shared multiprocessor allows a single program to use almost all the memory in the computer and it only needs a single copy of the OS

Problems with Parallelism

- Algorithmic and Performance
  - Think in terms of parallel tasks
  - New algorithms
  - Accounting for communication overhead/synchronization
- Implementation and Correctness
  - Problem of Atomicity
  - Problem of Cache Coherence
  - Problem of Consistency
    - Not covered in this class

Problem of Atomicity

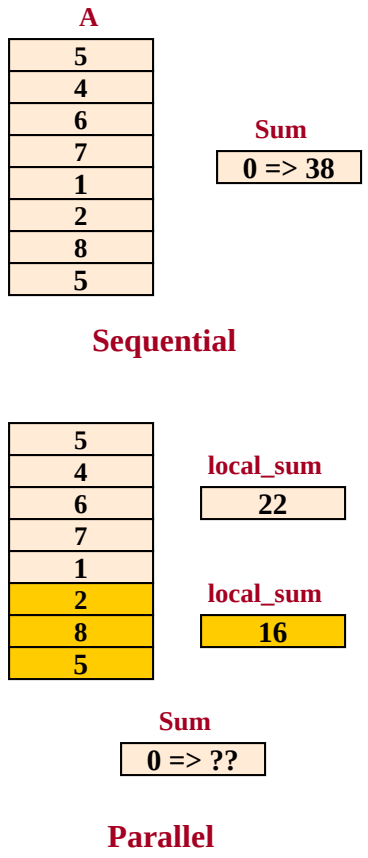
- Sum an array, A, of numbers {5,4,6,7,1,2,8,5}
- Sequential method

```
for(i=0; i < 7; i++) { sum = sum + A[i]; }
```

- Parallel method (2 threads with ID=0 or 1)

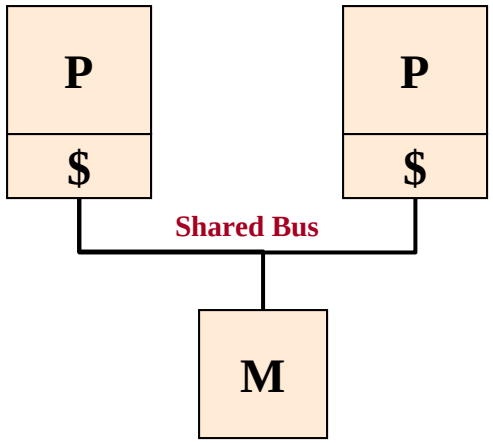
```
for(i=ID*4; i < (ID+1)*4; i++) {
  local_sum = local_sum + A[i]; }
sum = sum + local_sum;
```

- Problem
  - Updating a shared variable (e.g. sum)
  - Both threads read sum=0, perform sum=sum+local_sum, and write their respective values back to sum
  - Sum ends up with only a partial sum
  - Any read/modify/write of a shared variable is susceptible
- Solution: Atomic updates accomplished via some form of locking



Atomic Operations

- Read/Modify/Write (RMW) sequences are usually done with separate instructions
- Possible Sequence:
  - P1 Reads sum (lw)
  - P1 Modifies sum (add)
  - P2 Reads sum (lw)
  - P1 Writes sum (sw)
  - P2 uses old value…
- Partial Solution: Have a separate flag/"lock" variable (0=Lock is free/unlocked, 1 = Locked)
- Lock variable is susceptible to same problem as sum (read/modify/write)
- Hardware has to support some kind of instruction to implement atomic operations usually by not releasing bus between read and write



Thread 1:	Thread 2:
Lock L	Lock L
Update sum	Update sum
Unlock L	Unlock L

LockL:
Read(L)
If (L==0) L=1;

Locking/Atomic Instructions

- TSL (Test and Set Lock)
  - tsl reg, addr_of_lock_var
  - Atomically stores const. '1' in lock_var value & returns lock_var in reg
    - Atomicity is ensured by HW not releasing the bus during the RMW (Read/Modify/Write) cycle
- LL and SC (MIPS & others)
  - Lock-free atomic RMW
  - LL = Load Linked
    - Normal lw operation but tells HW to track any external accesses to address
  - SC = Store Conditional
    - Like sw but only stores if no other writes since LL & returns 0 in reg. if failed, 1 if successful

```
LOCK:     TSL   $4,lock_addr
          BNE   $4,$zero,LOCK
          return;

UNLOCK:   sw    $zero,lock_addr
```

```
          LA    $8,lock_addr
LOCK:     ADDI  $9,$0,1
          LL    $4,0($8)
          SC    $9,0($8)
          BEQ   $9,$zero,LOCK
          BNE   $4,zero,LOCK
```

```
          LA    $t1,sum
UPDATE:   LL    $5,0($t1)
          ADD   $5,$5,local_sum
          SC    $5,0($t1)
          BEQ   $5,$zero,UPDATE
```

Store Conditional

- LL and SC are used in a sequence. If the contents of the memory location specified by the LL are changed before the SC to the same address takes place, the SC fails

- Note that SC is defined to both store the value of a register and to change the value of that register to a 1 if it succeeds and to a 0 if it fails

Atomic Exchange Example

- Since LL returns the initial value and the SC returns 1 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by $s1:

```
Again:  add $t0, $0, $s4      ;copy exchange value
        ll  $t1, 0($s1)
        sc  $t0, 0($s1)
        beq $t0, $0, Again    ; branch if store fails
        add $s4, $0, $t1      ;put load value in $s4
```

- AT the end of the sequence, the contents of $s4 and the memory location specified by $s1 have been atomically exchanged
- Any time the processor intervenes and modifies the value in memory btn LL and SC instructions, SC returns 0 in $t0, causing the code sequence to try again

Solving Problem of Atomicity

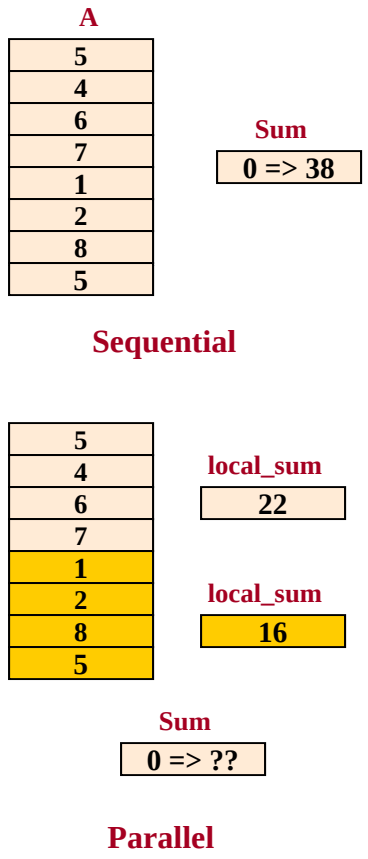
- Sum an array, A, of numbers {5,4,6,7,1,2,8,5}
- Sequential method

```
for(i=0; i < 7; i++) { sum = sum + A[i]; }
```

- Parallel method (2 threads with ID=0 or 1)

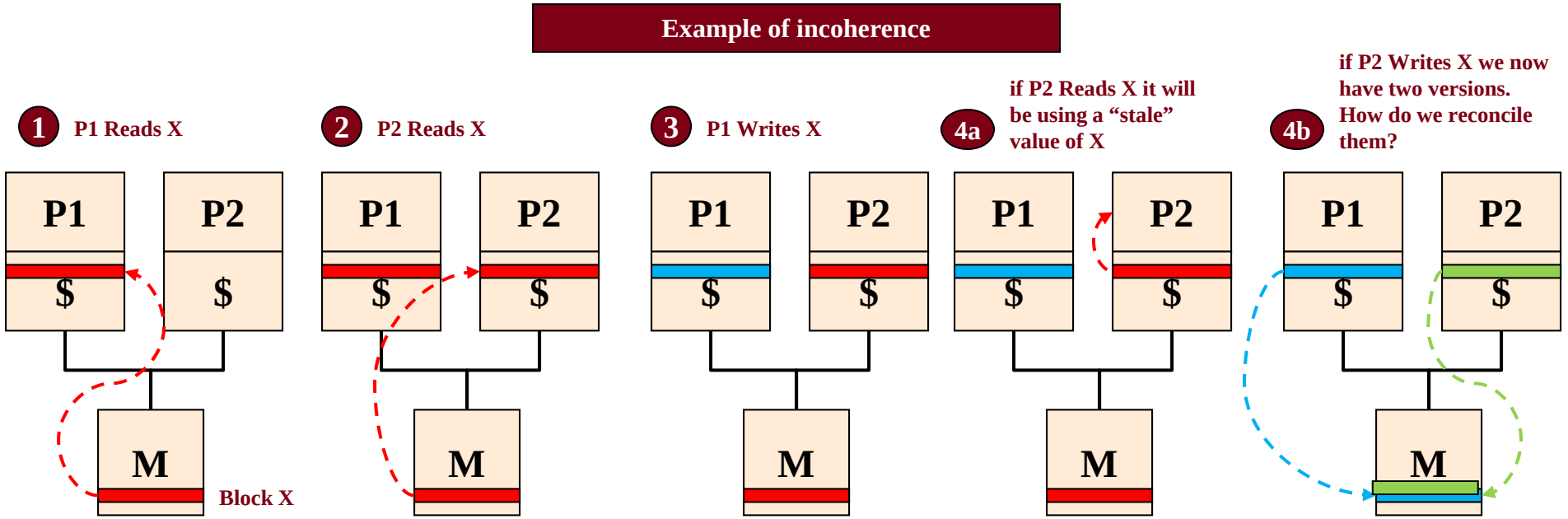
```
lock L;
for(i=ID*4; i < (ID+1)*4; i++) {
  local_sum = local_sum + A[i]; }

getlock(L);
sum = sum + local_sum;
unlock(L);
```



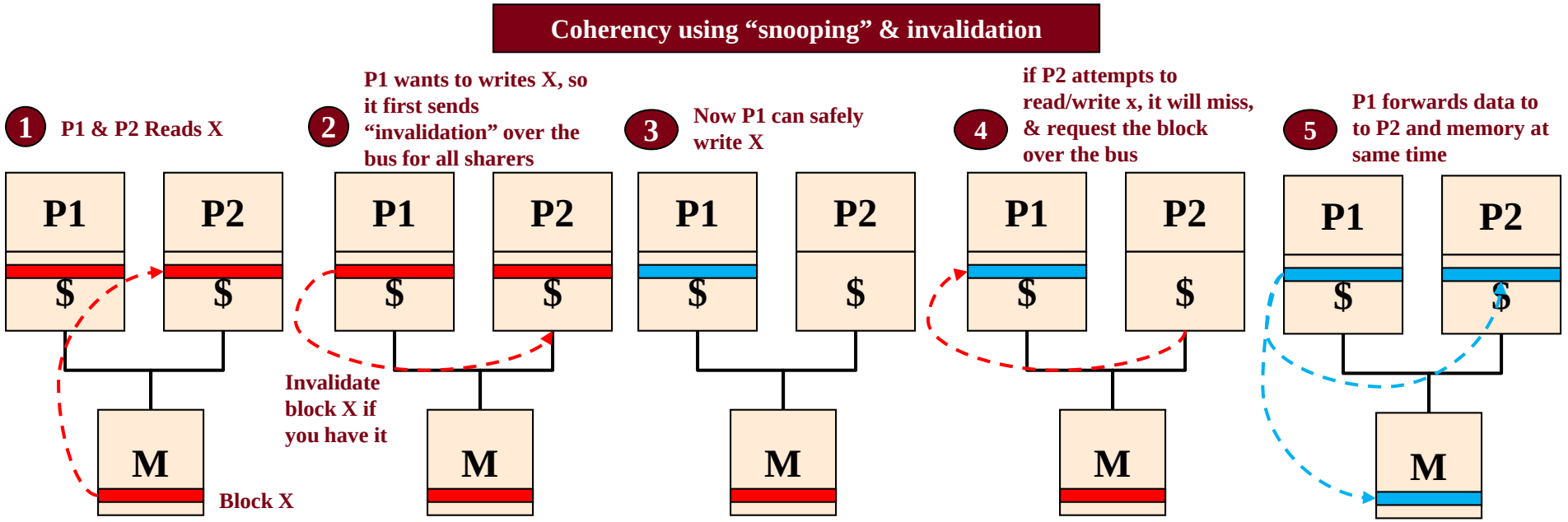
Cache Coherency

- Most multi-core processors are shared memory systems where each processor has its own cache
- Problem: Multiple cached copies of same memory block
  - Each processor can get their own copy, change it, and perform calculations on their own different values…INCOHERENT!



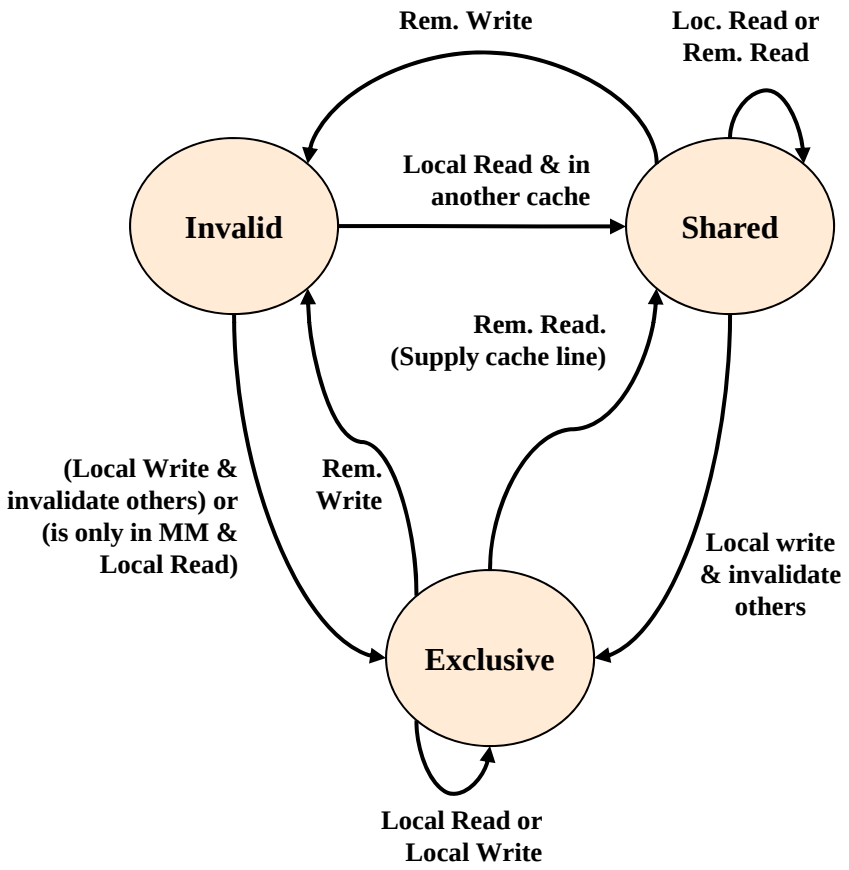
Solving Cache Coherency

- If no writes, multiple copies are fine
- Two options: When a block is modified
  - Go out and update everyone else's copy
  - Invalidate all other sharers and make them come back to you to get a fresh copy
- "Snooping" caches using invalidation policy is most common
  - Caches monitor activity on the bus looking for invalidation messages
  - If another cache needs a block you have the latest version of, forward it to mem & others



Implementation of Cache Coherency

- To be able to implement this invalidation strategy, some amount of "state" (info) needs to be kept for each block
- 3 States
  - Invalid (block must be re-fetched on access)
  - Shared (read-only; others have a copy)
  - Exclusive (read-write; no sharing so safe to write)



Coherency Example

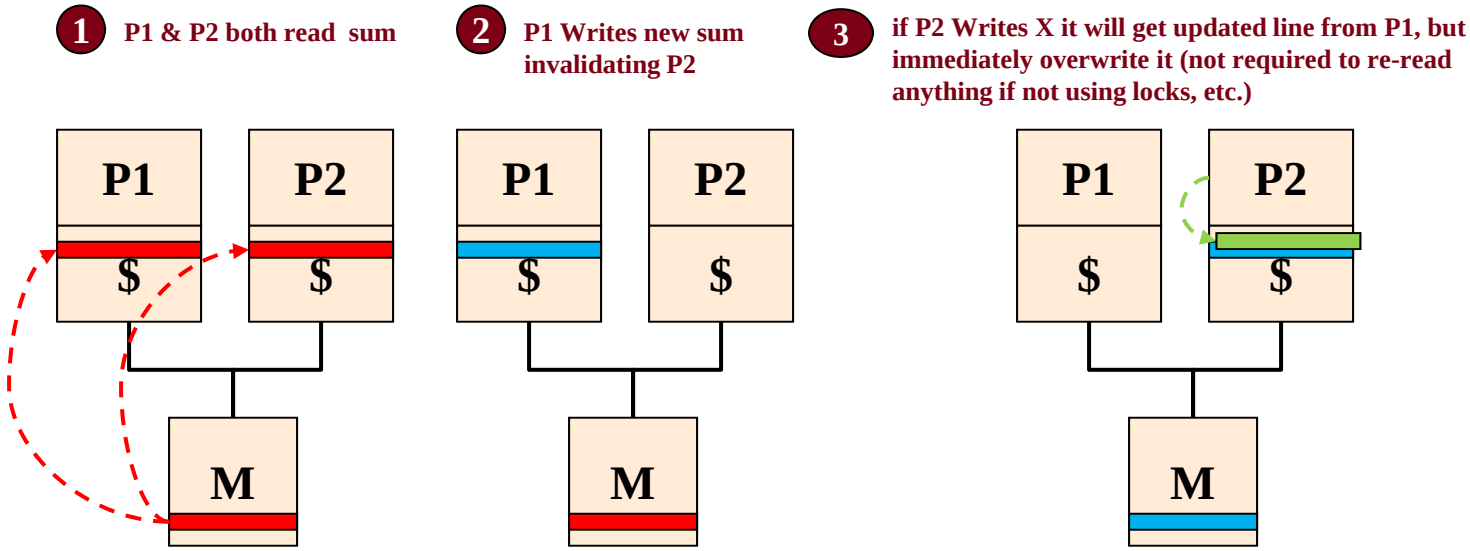
Processor Activity	Bus Activity	P1 $ Content	P1 Block State (E,S,I)	P2 $ Content	P2 Block State (E,S,I)	Memory Contents
		-	-	-	-	A
P1 reads block X	Request X	A	E	-	-	A
P2 reads block X	Request X	A	S	A	S	A
P1 writes block X	Invalidate X	B	E	-	I	A
P2 reads block X	Request X	B	S	B	S	B

Updated Coherency Example

Processor Activity	Bus Activity	P1 $ Content	P1 Block State (E,S,I)	P2 $ Content	P2 Block State (E,S,I)	Memory Contents
		-	-	-	-	A
P1 reads block X	Request X	A	E	-	-	A
P1 writes X		B	E	-	-	A
P2 writes X	Request X	-	I	C	E	B
P1 reads block X	Request X	C	S	C	S	C

Is Cache Coherency = Atomicity?

- No, cache coherence only serializes writes and does not serialize entire read-modify-write sequences
  - Coherence simply ensures two processors don't read two different values of the same memory location
- Consider our sum example

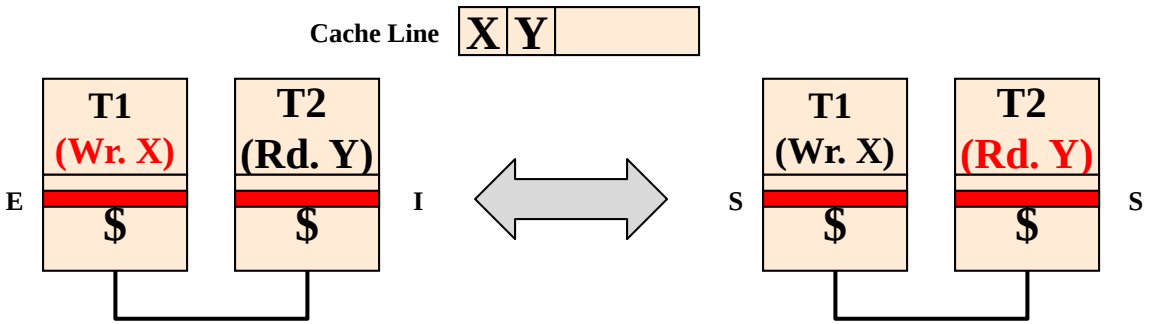


False Sharing

- Thread-independent (i.e. non-shared) variables allocated on the same cache line
- Can cause a large performance degradation due to cache coherence (invalidates, etc.)

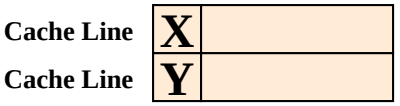
```
int x = 0;
int y = 0;
void t1() {
  for(int x=ITERS; x > 0; x--);
  y = 1;
}
void t2() {
  while( y == 0);
  printf("Y was set to 1\n");
}
```

False Sharing Example




```
int x = 0;
int y __attribute__ ((aligned (64))) = 0;
…
```

One solution: Alignment



Locks and Contention

- The more threads compete for a lock the slower performance will be
  - Continuous sequence of invalidate, get exclusive access for 'tsl' or 'sc', check lock, see it is already taken, repeat
- Options
  - Use special locks (e.g. queuing locks where a thread takes a number and waits until it is called rather than continuously checking if the lock is free)
  - Lock Granularity: Use separate locks for each element in a data structure rather than the one lock for the whole structure
  - Others that you can explore as needed…