CMPEN 431
Computer Architecture
Fall 2018

Intro to Multiprocessor/Multicore Systems

Jack Sampson(www.cse.psu.edu/~sampson)

[Slides adapted from work by Mary Jane Irwin, in turn adapted from *Computer Organization and Design, Revised 4th Edition*,

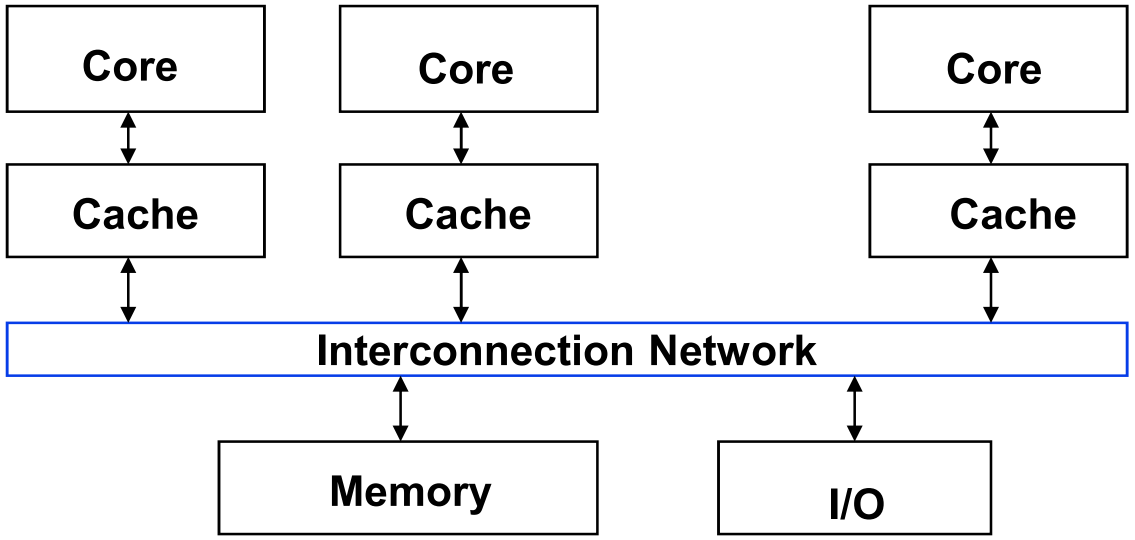
Patterson & Hennessy, © 2011, Morgan Kaufmann and *5th Edition*,

Patterson & Hennessy, © 2014, MK

With additional thanks/credits to Amir Roth, Milo Martin, CIS/UPenn]

The Big Picture: Where are We Now?

- Multiprocessor – a computer system with at least 2 cores
- Multicore – a chip (processor) with at least 2 cores



- Can deliver high throughput for multiple independent jobs – multiprogramming – via task-level or process-level parallelism
- Can improve the run time of a *single* program that has been specially crafted to run on a multiprocessor – a multithreaded (parallel) program

Multicores (aka CMPs) Now the Norm

- Power constraints have stalled clock rate improvements
  - If you want better performance, have to get it by effectively using multiple cores on the same chip !

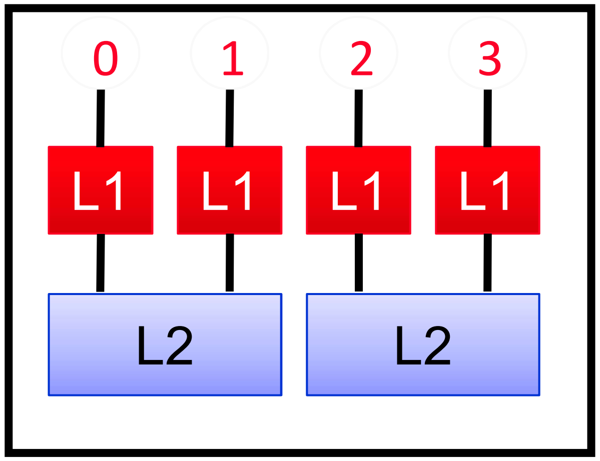
- Today's processors contain more than one core – Chip Multi(micro)Processors (CMPs) – in a single IC (modern ICs are actually SoCs that contain more than just processors! – more on that later)

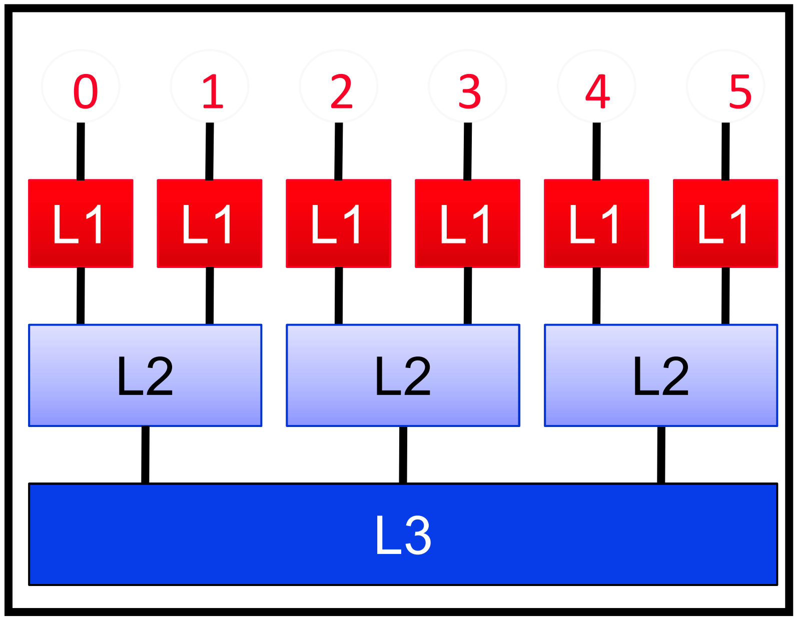
	AMD Opteron 6174	Intel Xeon 7500	IBM POWER7	Sun Niagara 2
Cores/chip	6	8	8	8
Clock rate	2.2 GHz	2.3GHz	4.14 GHz	1.4 GHz
Chip power	115 W	130W	~100 W?	94 W

Multicore = Platform Variety

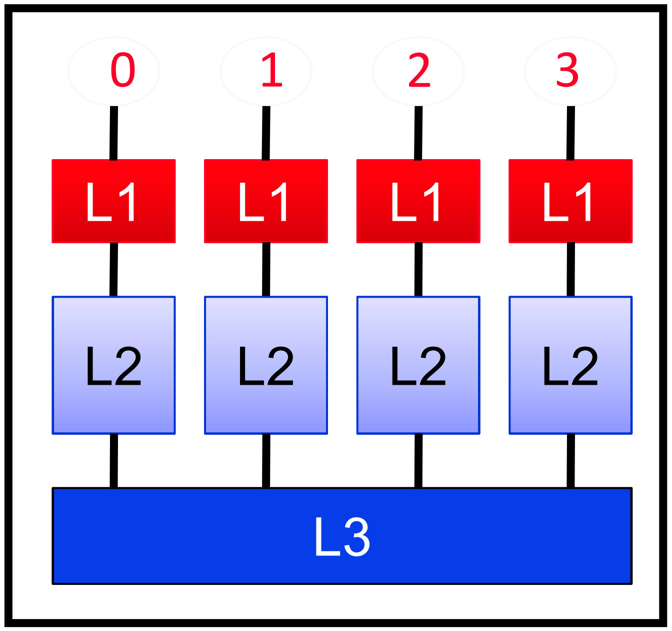
- Number/type of cores
- Levels and structure of the cache hierarchy
- Number of memory controllers (MCs) on-chip
- Type of on-chip interconnect

Harpertown





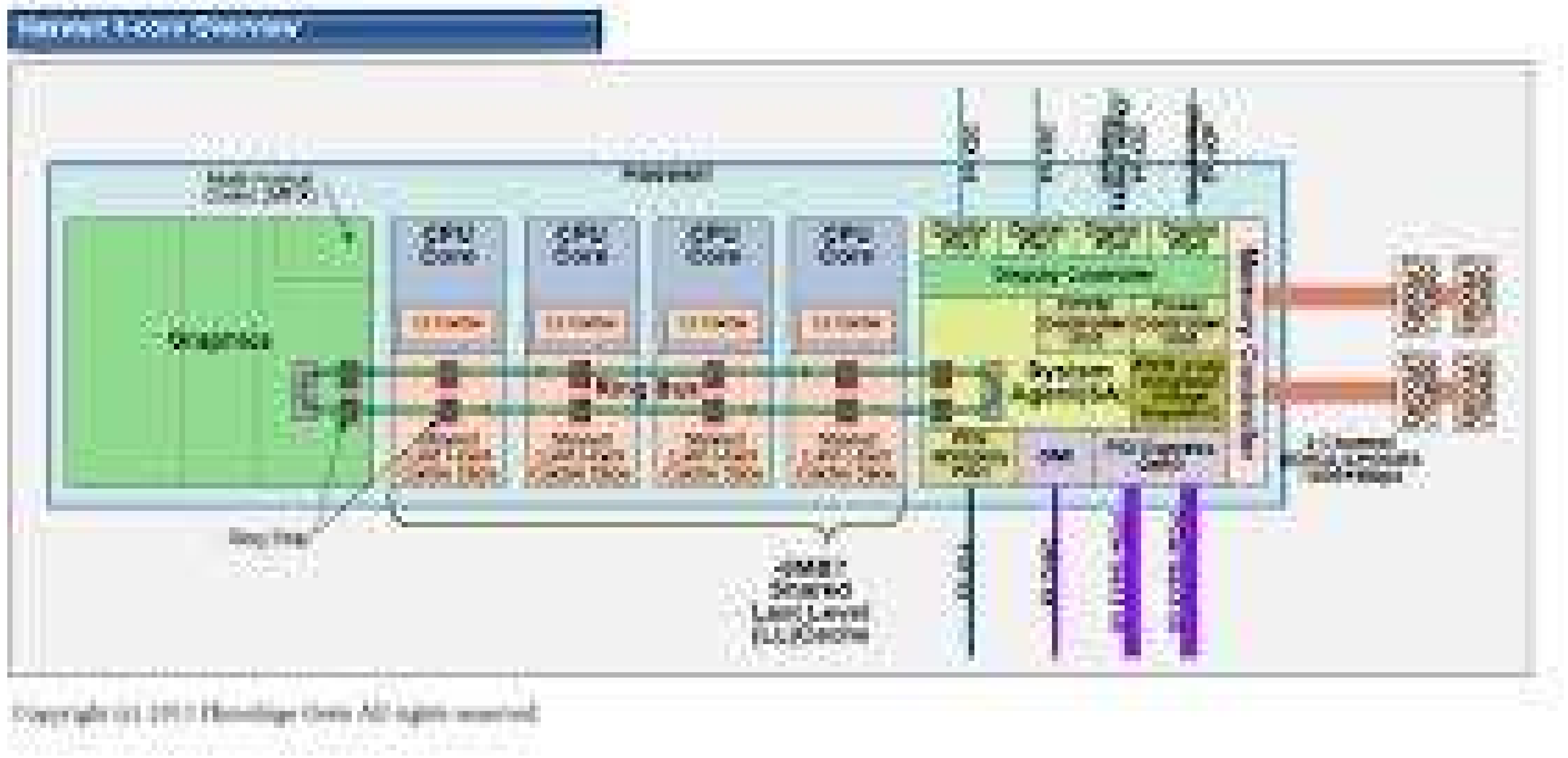
Dunnington



Nehalem

Intel’s Haswell Multicore

- 4 cores + graphics co-processor, shared LLC, 2 MCs



Quad core die shown above | Transistor count: 1.4 Billion | Die size: 177mm^2

Key Multiprocessor Design Questions

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How scalable is the architecture? How many cores can be supported?

Shared Memory multiProcessor (SMP)

- Q1 – Single physical address space shared by all cores
- Q2 – Threads on cores coordinate/communicate through shared variables in memory (via loads and stores)
  1. Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one core at a time (used to implement critical sections)
  2. Caches must be kept coherent (read of a data item returns the most recently written value of that data item)
- SMPs come in two styles
  1. Uniform memory access (UMA) multiprocessors
  2. Nonuniform memory access (NUMA) multiprocessors
- Programming NUMAs is the harder of the two
- But NUMAs can scale to larger sizes and have lower latency to local memory

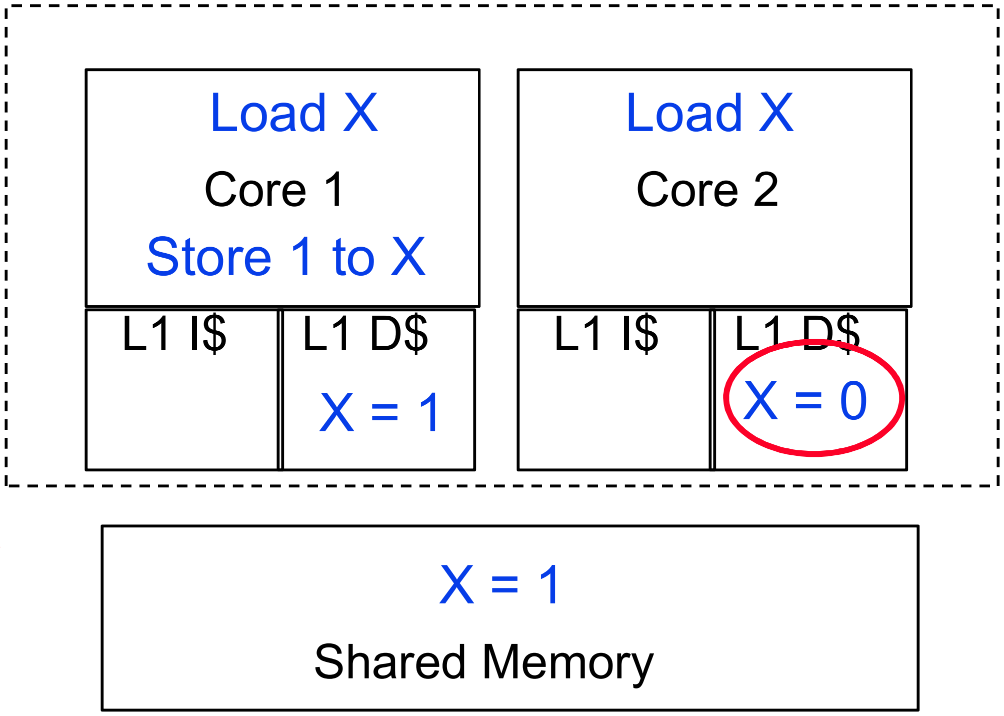
SMP Multithreaded Programming Model

- Creating multithreaded programs for multicores ?
  - Need a well-defined set of concurrency abstractions
- Programmer explicitly creates multiple software threads
  - Java has thread support built in, C/C++ supports P-threads library
  - Gives parallel speedups via Thread-Level Parallelism (TLP)
- All loads & stores are to a single **shared memory** space
  - Each thread has a private stack frame for its local variables
- A “thread switch” can occur at any time
  - Pre-emptive multithreading by OS
    - Hardware timer interrupt occasionally triggers OS
    - Quickly swapping threads gives illusion of concurrent execution
  - FGMT (cycle-level HW switching)
  - SMT (logically sub-cycle HW switching)

A Potential Problem in Parallel Processors

- Illustration of the cache coherence problem in a two core processor with private L1I$ and L1D$ caches and a (common) shared memory containing a variable **X**.

1. Core 1: Load X
2. Core 2: Load X
3. Core 1: Store 1 to X
4. …<other insts>
5. Core 1 does WB of X
6. Core 2: **Load X**



A Coherent Memory System

- Any read of a data item should return the most recently written value of the data item
  - Coherence – defines what values can be returned by a read
    - Stores (writes) to the same location are serialized (two stores to the same location must be seen in the same order by all cores)
  - Consistency – determines when a written value will be returned by a read

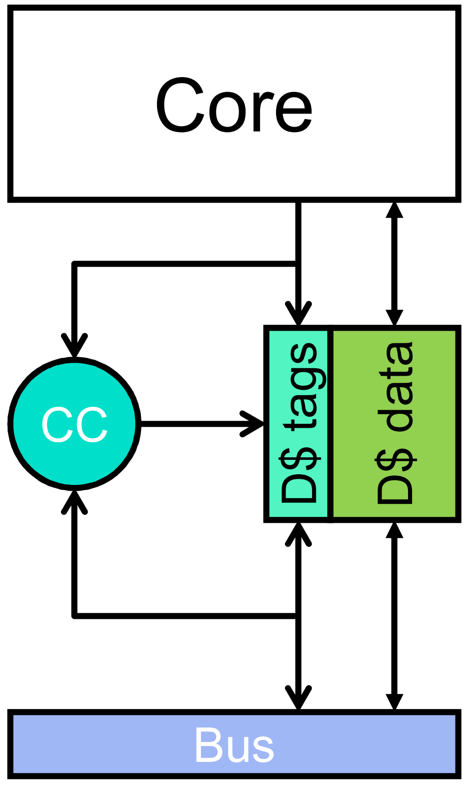
Critical for performance

- To enforce coherence, caches must provide:
  - Replication of **current** shared data items in multiple cores' caches
    - Replication reduces both latency and contention for a read shared data item
  - Migration of **updated** shared data items to a core's local cache
    - Migration reduces the latency to access the data and the bandwidth demand on the shared memory

Cache Coherence Protocols

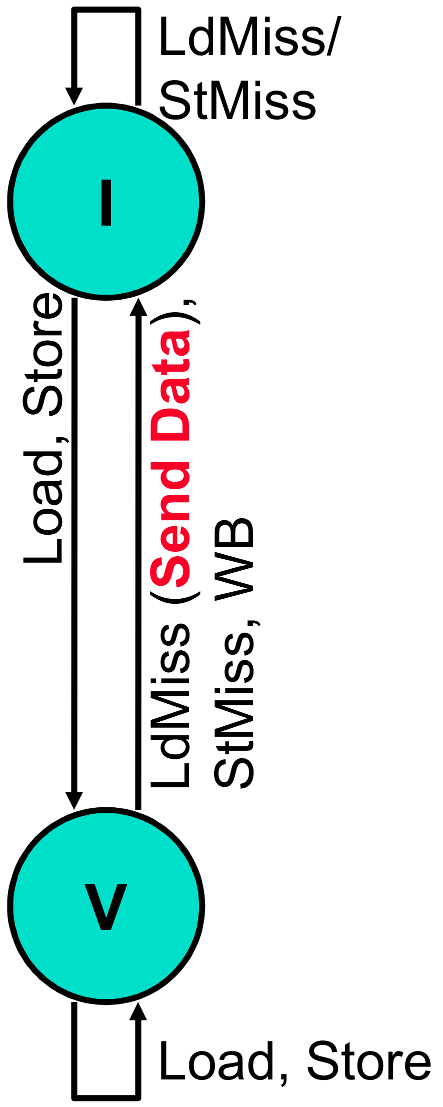
- Need a hardware mechanism to ensure cache coherence the most popular of which (for small SMPs) is snooping
  - The cache controllers monitor (snoop) on the broadcast medium (e.g., bus) with duplicate address tag hardware (so they don’t interfere with core’s access to the cache) to determine if their cache has a copy of a block that is requested
- Write invalidate protocols – writes require exclusive access and invalidate *all* other existing copies
  - Exclusive access ensure that no other readable or writable copies of an item exists in other caches
  - If two processors attempt to write the same data at the same time, one of them wins the race causing the other core’s copy to be invalidated. For the other core to complete, it must first obtain the new data from the first core’s cache – thus enforcing write serialization
- Other protocols (write update) possible, but not common

Hardware Cache Coherence



- Write-back caches (rather than write-through) for performance reasons
- Coherence Controller (CC)
  - Monitors ("snoops") bus traffic (addresses and data)
  - Executes the **coherence protocol**
    - What to do the with local copy when you see different things happening on bus
- Three core-initiated events
  - **Ld**: load **St**: store **WB**: write-back
- Two remote-initiated (bus) events
  - **LdMiss**: read miss from ***another*** core
  - **StMiss**: write miss from ***another*** core

VI (MI) coherence protocol



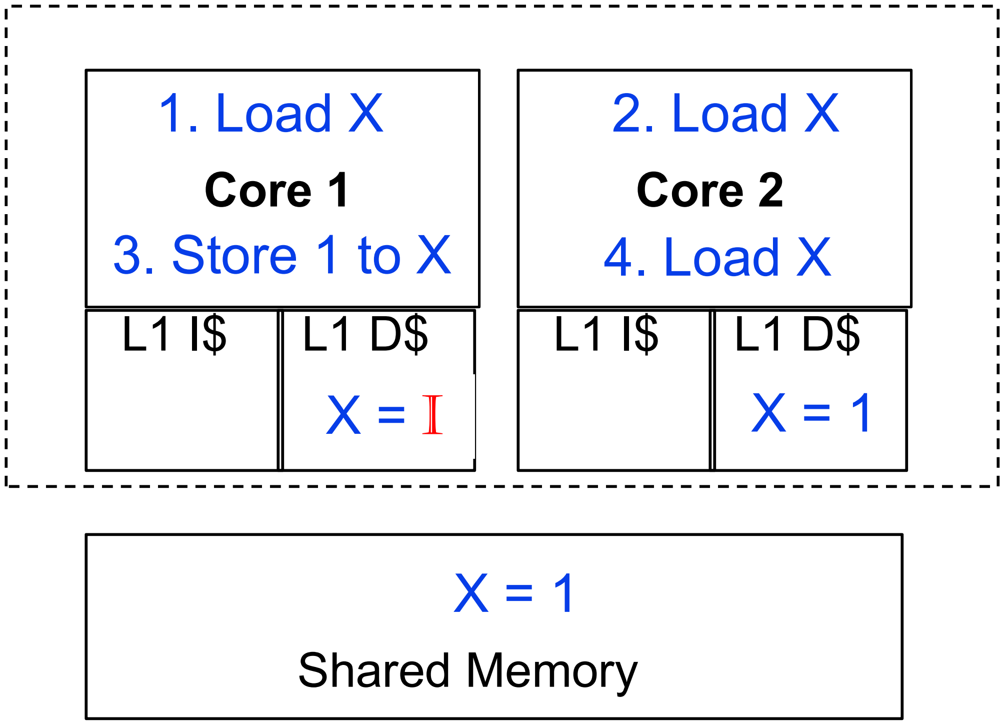
- **VI (valid-invalid) protocol**: aka MI
  - Two states (per block in cache)
    - **V (valid)**: have block
    - **I (invalid)**: don't have block
    - + Can implement with the valid bit
- Protocol diagram (shown on left)
  - If anyone **else** wants to read/write block
    - Give it up: transition to **I** state
  - WriteBack (WB) if your own copy is dirty
- This is an **invalidate protocol**
- **Update protocol**: copy data (write-through), don't invalidate
  - Sounds good, but wastes a lot of bus bandwidth

VI Protocol State Transition Table

	From This Core		*From Other Cores*	
State	Load	Store	LdMiss	StMiss
Invalid (I)	Miss $\rightarrow$ V	Miss $\rightarrow$ V	---	---
Valid (V)	Hit	Hit	**Send Data** $\rightarrow$ I	$\rightarrow$ I

- Rows are "states" (**I** vs **V)** of data blocks in the caches
- Columns are "events" in the cores
  - WB (WriteBack) events not shown
  - **V $\rightarrow$ I** on a LdMiss (**Send Data**) also updates memory **if** block is Dirty
- Memory sends data when no core responds

Example of VI Snooping Protocol



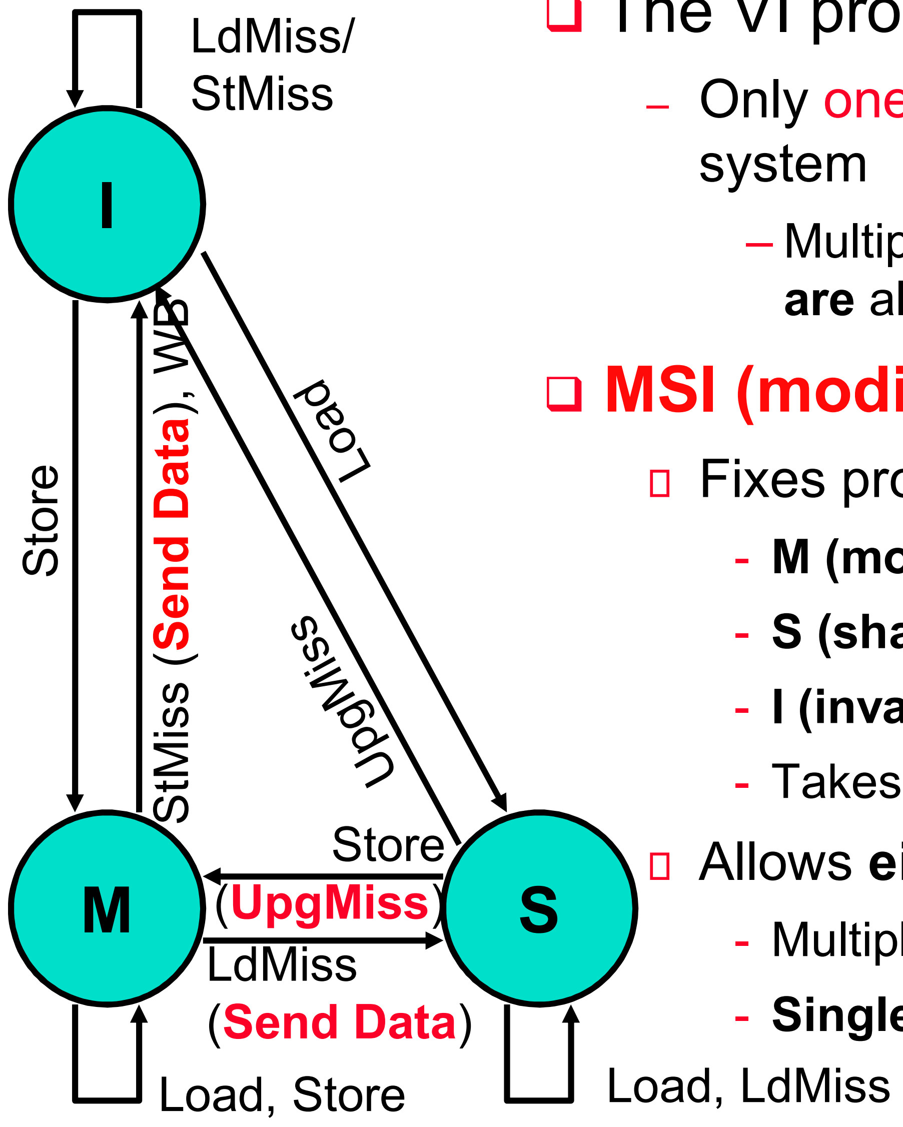
2. LdMiss seen by Core 1, so Core 1 does a **Send Data** and then invalidates its copy

3. StMiss seen by Core 2, so Core 2 invalidates its copy

4. LdMiss seen by Core 1, so Core 1 does a **Send Data** and then invalidates its copy

- When the second load miss by Core 2 occurs, Core 1 responds with the data value for Core 2 (canceling the response from the shared memory), updating the value of X in the shared memory, and invalidating its own copy

VI → MSI



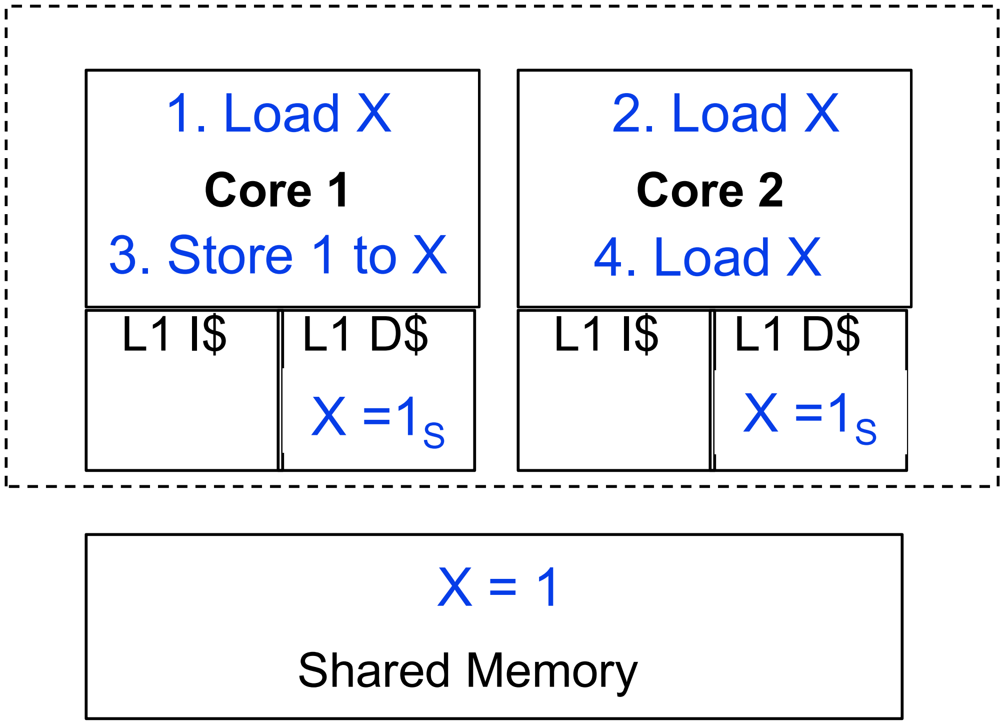
- The VI protocol is **inefficient**
  - Only one cached copy allowed in entire system
    - Multiple caches copies can't exist even if they **are** all read-only
- **MSI (modified-shared-invalid)**
  - Fixes problem: splits "V" state into two states
    - **M (modified)**: local dirty copy
    - **S (shared)**: local clean copy
    - **I (invalid)**
    - Takes 2 bits to implement
  - Allows **either**
    - Multiple read-only copies (**S**-state) **--OR--**
    - **Single** read/write copy (**M**-state)

MSI protocol state transition table

	From This Core		*From Other Cores*	
State	Load	Store	LdMiss	UpgMiss StMiss
Invalid (I)	Miss ➔ S	Miss ➔ M	---	---
Shared (S)	Hit	**UpgMiss** ➔ M	---	➔ I
Modified (M)	Hit	Hit	**Send Data** ➔ S	**Send Data** ➔ I

- M ➔S transition on a LdMiss also updates memory (**Send Data**) since the block is Dirty
  - After which memory will respond (as all cores will be in S)

Example of MSI Snooping Protocol



2. LdMiss seen by Core 1 (no action required)

3. Core 1 marks X as modified and sends out an **UpgMiss**

UpgMiss seen by Core 2 so Core 2 invalidates its copy

4. LdMiss seen by Core 1, so Core 1 does a **Send Data** of X and marks its copy as shared

- When the second load miss by Core 2 occurs, Core 1 responds with the value canceling the response from the shared memory (and also updating the value of X in the shared memory and marking its copy as shared)

Other Coherence Protocols

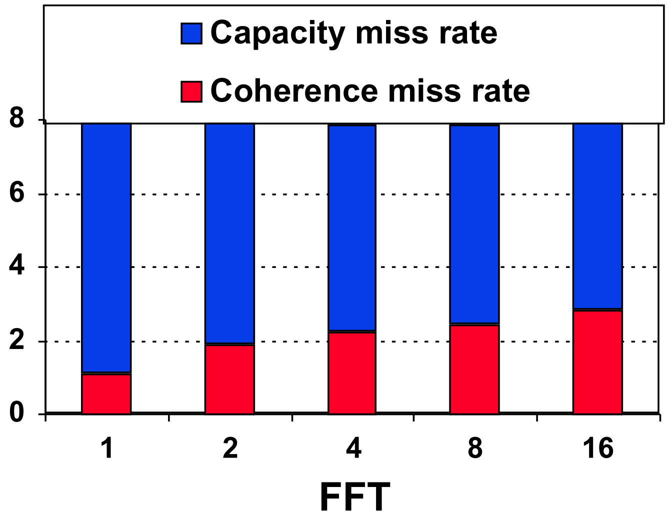
- Another write-invalidate protocol used in the Pentium 4 (and many other processors) is MESI with four states:
  - Modified – same
  - Exclusive – only one copy of the shared data is allowed to be cached
    - Since there is only one copy of the data, write hits don't need to send UpgMiss broadcast
  - Shared – same (multiple copies of the shared data may be cached (i.e., data permitted to be cached with more than one core))
  - Invalid – load miss becomes E if no other core is caching the data, else it becomes S on load
- Snooping protocols (and buses) don't scale well to many-cores
  - They use directory based, non-broadcast coherence protocols where a directory in memory keeps track of data ownership

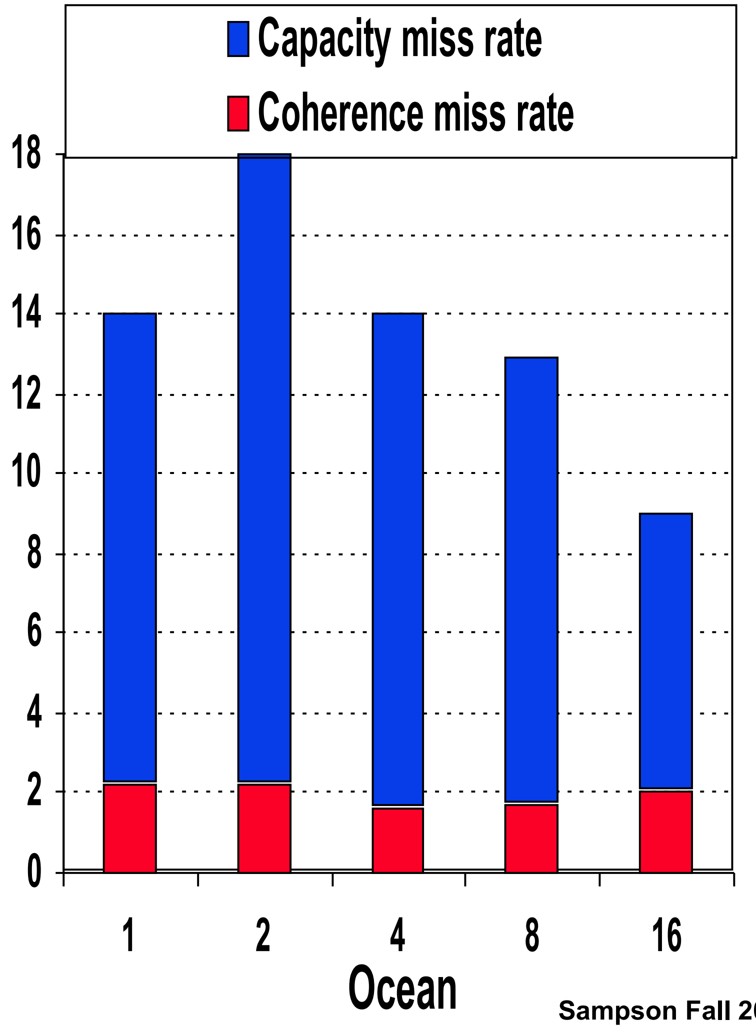
The 4th C: Cache Coherence Misses

- Shared data has lower spatial and temporal locality
  - Share data misses often dominate cache behavior even though they may only be 10% to 40% of the data accesses

64KB 2-way set associative data cache with 32B blocks

Hennessy & Patterson, *Computer Architecture: A Quantitative Approach*

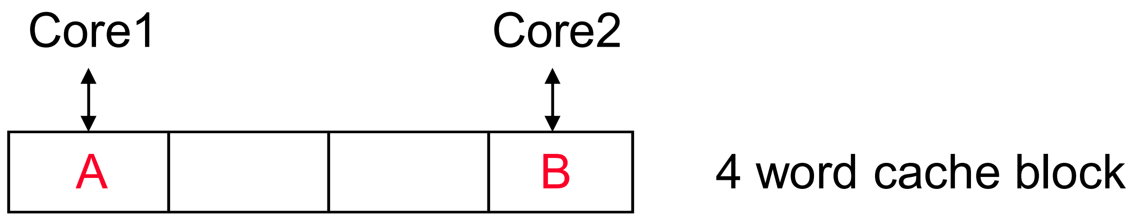




Block Size Effects

- Writes to one word in a multi-word block mean that the full block is invalidated

- Multi-word blocks can also result in false sharing: when two cores are writing to two different variables that happen to fall in the same cache block
  - With VI false sharing increases cache miss rates



- Compilers can help reduce false sharing by allocating highly correlated data to the same cache block

Memory Consistency

- When are stores seen by other cores ?
  - "Seen" means a load returns the written value
  - Can't be instantaneously
- Assumptions: Write Serialization
  - A store completes only when all cores have seen it
  - A core does not reorder its own stores with other memory accesses (true on in-order commit datapaths)
- Consequences
  - C1 stores to location X then C2 stores to location X (and in MSI, C1 invalidates its copy of X)
  - Serializing the stores ensures that **every** core will first see the store by C1 and then, eventually, the store by C2
  - Cores can reorder loads, but not stores

Memory Consistency Models

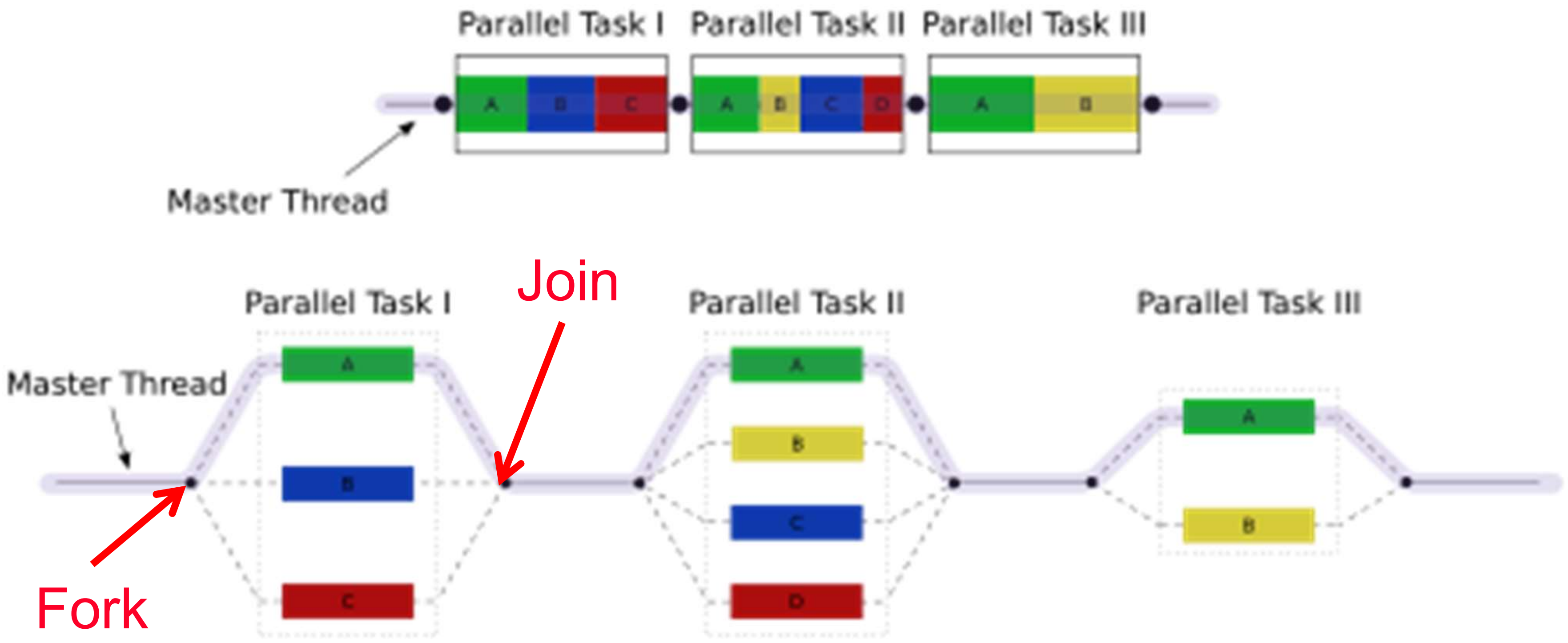
- **Sequential consistency (SC)** (MIPS, PA-RISC)
  - **Formal definition of memory view programmers expect**
  - Cores see their own loads and stores in program order
    - + Provided naturally, even with out-of-order execution
  - But also: Cores see others' loads and stores in program order
  - And finally: All cores see the same global load/store ordering
    - – Last two conditions not naturally enforced by coherence
  - **Indistinguishable from multi-programmed uniprocessor**
- **Processor consistency (PC)** (x86, SPARC)
  - Allows an in-order Store Buffer
    - - Stores can be deferred, but must be put into the cache **in order**
- **Release consistency (RC)** (ARM, Itanium, PowerPC)
  - Allows an un-ordered Store Buffer
    - - Stores can be put into cache **in any order**

Shared Memory multiProcessor (SMP)

- Q1 – Single memory address space shared by all cores
- Q2 – Cores coordinate/communicate through shared variables in memory (via loads and stores)
  1. Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one core at a time (used to implement critical sections)
  2. Caches must be kept coherent (cores have the same shared data)
- SMPs come in two styles
  1. Uniform memory access (UMA) multiprocessors
  2. Nonuniform memory access (NUMA) multiprocessors
- Programming NUMAs is the harder of the two
- But NUMAs can scale to larger sizes and have lower latency to local memory

Fork-Join Computation Model

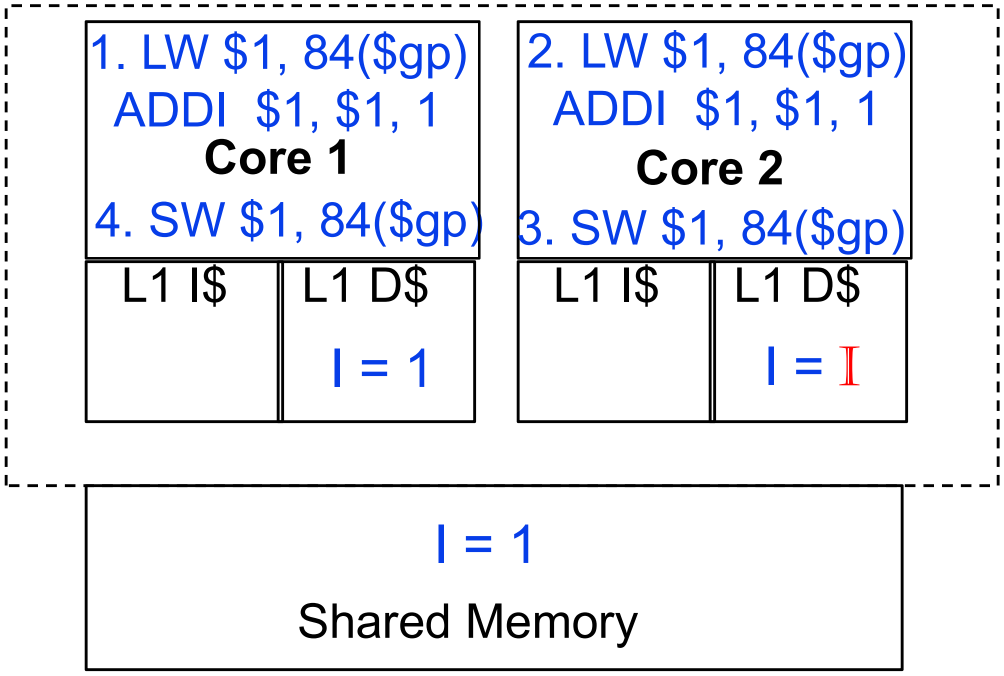
- The master thread "forks" into a number of threads which execute blocks of code in parallel and then "join" back into the master thread when done.



http://en.wikipedia.org/wiki/OpenMP

Coherence Alone Isn’t Enough

Assume variable “I” at address 84($gp) and identical $gp



2. LdMiss seen by Core 1, so Core 1 does a **Send Data** and then invalidates its copy

4. StMiss seen by Core 2, so Core 2 does a WB of I (since its dirty) updating shared memory, and then invalidates its copy. Core 1 then does its write of I to its cache

- When the SW by Core 2 occurs, shared memory is updated. The problem is that Core 1’s register file contains stale data for I so what gets stored in Core 1’s cache is also stale.
  - Need another way to fix this kind of data sharing

Process Synchronization

- Need to be able to coordinate threads working on a common task; sharing data must be coordinated carefully
  - Lock variables (semaphores) are used to coordinate or synchronize threads
    - mutual exclusion – restrict data access to one core at a time
    - sequential ordering – must complete the first operation before the second operation is allowed to begin

1. Need an architecture-supported arbitration mechanism to decide which core gets access to the lock variable
   - The single bus provides an arbitration mechanism, since the bus is the only path to memory – the core that gets the bus wins
2. Need architecture-supported operation(s) for atomicity
   - Locking can be done via an ***atomic*** swap operation which allows a core to both read a location *and* set it to the locked state – test-and-set or READ-MODIFY-WRITE – in the *same* bus operation

MIPS Atomic Exchange Support

- Atomic exchange (atomic swap) – interchanges a value in a register for a value in memory atomically, i.e., as one operation (instruction)
  - Implementing an atomic exchange would require **both** a memory read and a memory write in a single, uninterruptable instruction
- MIPS provides `ll/sc`: load-linked / store-conditional
  - Atomic load/store pair

```
ll r2,0(&lock)       #load linked
// potentially other load/store's by other cores
sc r1,0(&lock)       #store conditional
```

  - On `ll`, the SRAM cache controller (or DRAM memory controller) remembers the core id and the load address (in a link register) …
    - And watches for **stores** by other cores (or for any exceptions)
  - If a store by another core to the same address is detected or a context switch occurs, the `sc` fails (i.e., the store is not performed)
    - the `sc` returns a 1 to the core if the store was successful, 0 on fail

Atomic Exchange with `ll` and `sc`

- If the contents of the memory location specified by the `ll` are changed (by some other core) before the `sc` to the same address executes, the `sc` fails (returns a zero)

```
lock: add $t0, $zero, $s4      #$t0=$s4 (exchange value)
      ll  $t1, 0($s1)          #load memory value to $t1
```

// potentially other load/store instr's executed by other cores

```
      sc  $t0, 0($s1)          #try to store the exchange
                               #value to memory, if fail
                               #due to intervening store
                               #$t0 = 0, else $t0 = 1
```

If the `sc` fails (does not succeed in storing the exchange value into memory) and returns a 0 in `$t0`, the code sequence tries again

```
      beq $t0, $zero, lock     #try again on failure
      add $s4, $zero, $t1      #put exchange value in $s4
```

On success, the contents of `$s4` and the memory location specified by `$s1` have been atomically exchanged

Atomic Exchange Example

- Assume all loads and stores (including `ll` and `sc`) hit in the L1 cache

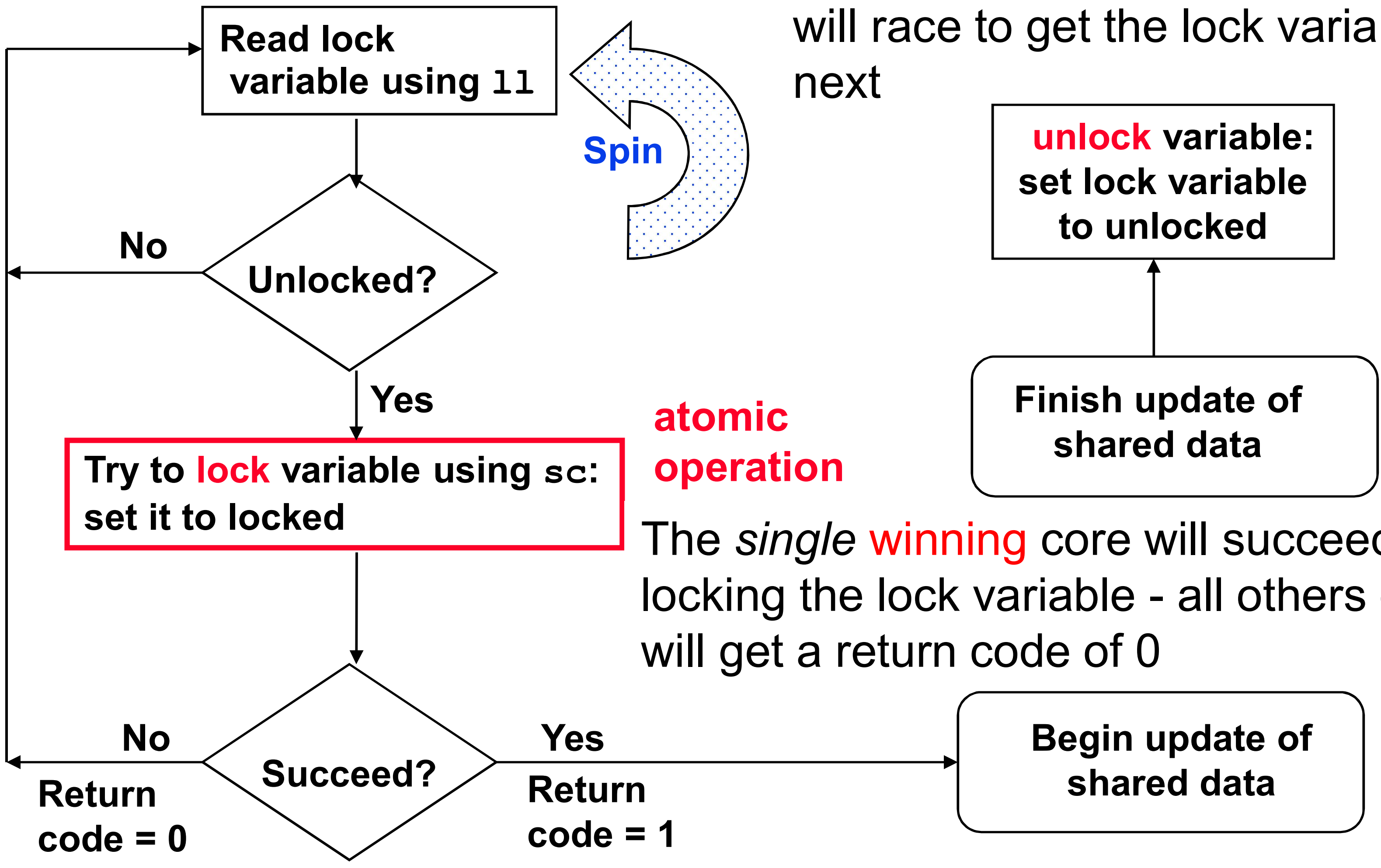
Core0	Core1	Cycle	Core 0 $t0	Core 0 $t1	Mem ($s1)	Core1 $t0	Core1 $t1
		0	30	40	55	10	11
	`ll $t1,0($s1)`	1					
`ll $t1,0($s1)`		2					
	`sc $t0,0($s1)`	3					
`sc $t0,0($s1)`		4					

Atomic Exchange Example

- Assume all loads and stores (including `ll` and `sc`) hit in the L1 cache

			Core 0		Mem ($s1)	Core1	
Core0	Core1	Cycle	$t0	$t1		$t0	$t1
		0	30	40	55	10	11
	`ll $t1,0($s1)`	1					**55**
`ll $t1,0($s1)`		2		**55**			
	`sc $t0,0($s1)`	3			**10**	**1**	
`sc $t0,0($s1)`		4	**0**		**10**		

Spin-Lock Synchronization


Read lock variable using `ll`
Spin
Unlocked?
No
Yes
Try to lock variable using `sc`: set it to locked
atomic operation
Succeed?
No
Return code = 0
Yes
Return code = 1
Begin update of shared data
Finish update of shared data
unlock variable: set lock variable to unlocked

On unlock all the cores spinning will race to get the lock variable next

The *single* winning core will succeed in locking the lock variable - all others cores will get a return code of 0

Spin-Locks on Bus Connected ccUMAs

- ccUMA = cache-coherent UMA
- With a bus based cache coherency protocol (e.g., MSI, MESI), joins are done via spin-locks which allow cores to wait on a local copy of the lock variable in their caches
  - Reduces bus traffic – once the core with the lock releases the lock (e.g., writes a 0) all other caches see that write and invalidate their old copy of the lock variable. Unlocking restarts the `ll-sc` race to get the lock. The winning core gets the bus and writes the lock back to 1. The other caches then invalidate their copy of the lock and on the next lock read fetch the new lock value (1) from memory.
- This scheme has problems scaling up to many cores because of the communication traffic when the lock is released and contested

Cache Coherence Bus Traffic

	C0	C1	C2	Bus activity	Memory
1	Has lock	Spins	Spins	None	
2	Releases lock (0)	Spins	Spins	Bus services C0's invalidate	
3		Cache miss	Cache miss	Bus services C2's cache miss	
4		Waits	Reads lock (0)	Response to C2's cache miss	Update lock in memory from C0
5		Reads lock (0)	Swaps lock (`ll,sc` of 1)	Bus services C1's cache miss	
6		Swaps lock (`ll,sc` of 1)	Swap succeeds	Response to C1's cache miss	Sends lock variable to C1
7		Swap fails	Has lock	Bus services C2's invalidate	
8		Spins	Has lock	Bus services C1's cache miss	

Summing 100,000 Numbers on 100 Core SMP

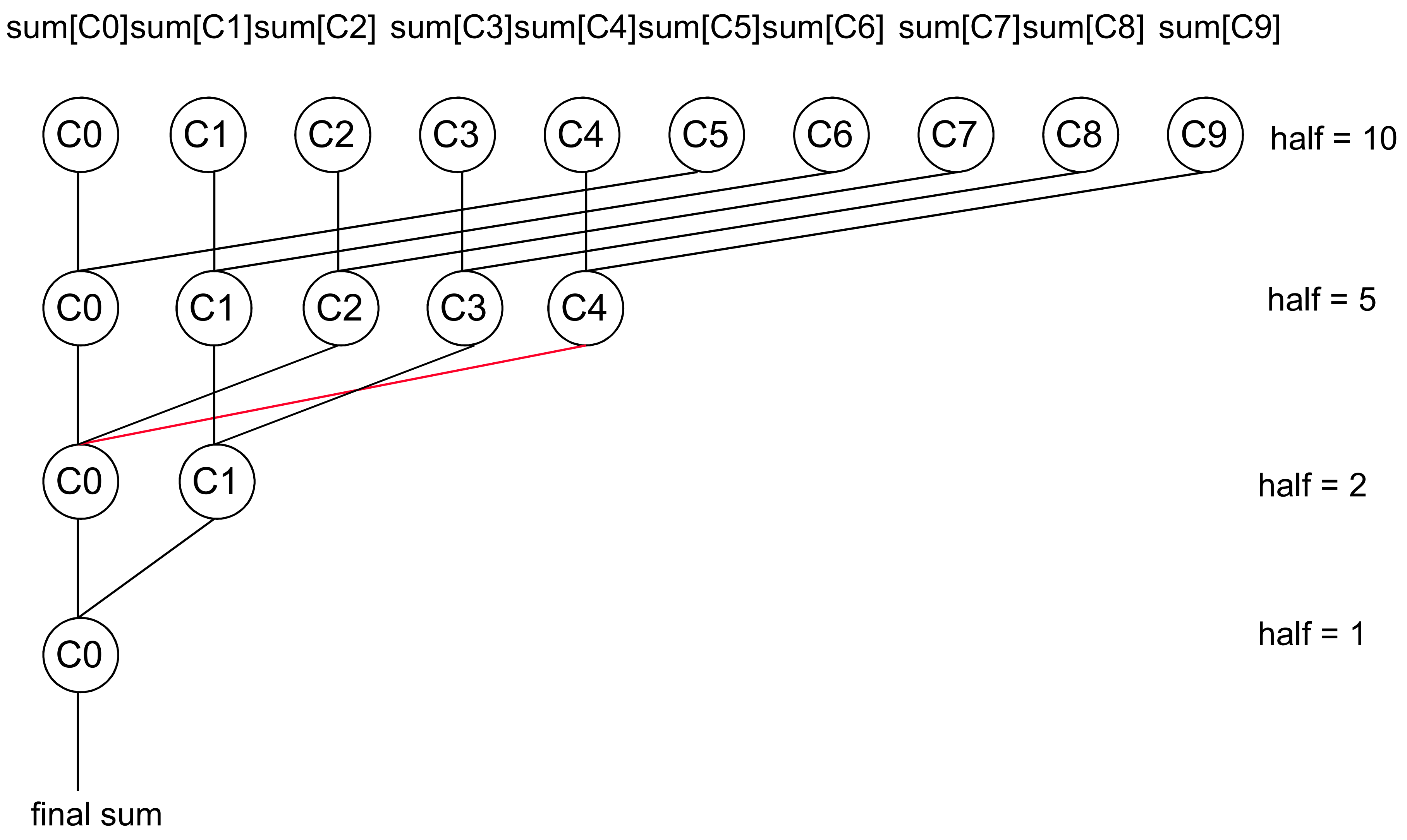
- Cores start by running a thread loop that sums their subset of vector `A` numbers (vectors `A` and `sum` are shared variables, `Cn` is the core's number, `i` is a private variable)

```
sum[Cn] = 0;
for (i = 1000*Cn; i< 1000*(Cn+1); i = i + 1)
  sum[Cn] = sum[Cn] + A[i];
```

- The cores' threads then coordinate in adding together the partial sums (`half` is a private variable initialized to `100` (the number of cores)) – reduction

```
repeat
  synch();                /*synchronize first
  if (half%2 != 0 && Cn == 0)
       sum[0] = sum[0] + sum[half-1];
  half = half/2
  if (Cn<half) sum[Cn] = sum[Cn] + sum[Cn+half]
until (half == 1);        /*final sum in sum[0]
```

An Example with 10 Cores (10 Threads)



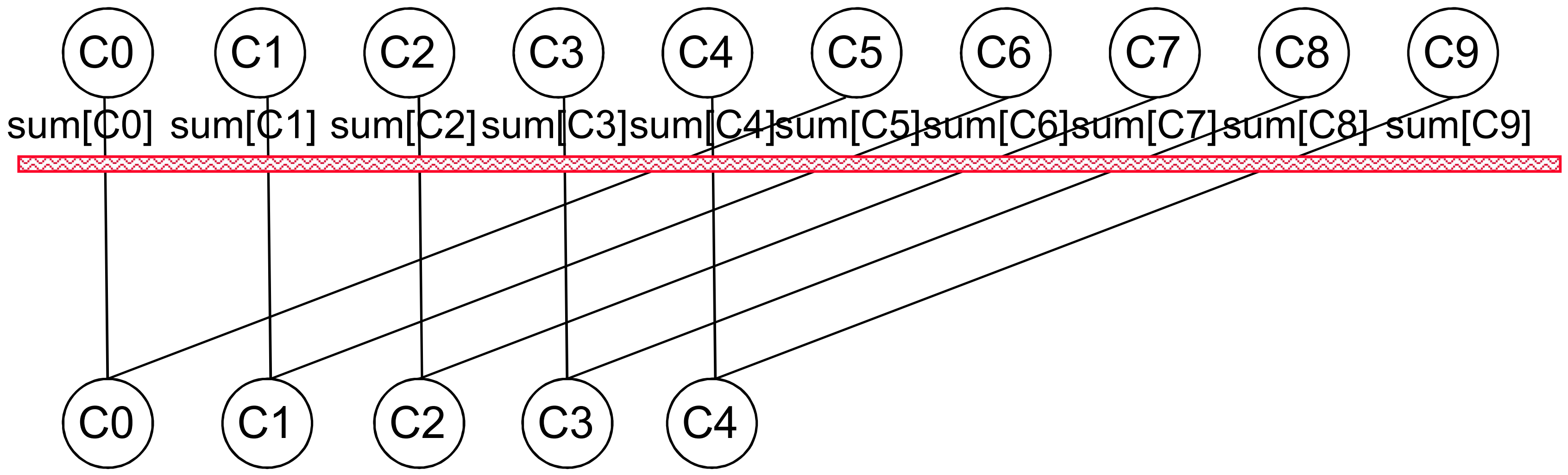
What was in Synch() routine?

- `Cn` is the core's number, vectors `A` and `sum` are shared variables, `i` is a private variable, `half` is a private variable initialized to the number of cores

```
 sum[Cn] = 0;
 for (i = 1000*Cn; i< 1000*(Cn+1); i = i + 1)
   sum[Cn] = sum[Cn] + A[i];
                                               /* each core sums its
                                               /* subset of vector A
repeat                                        /* adding together the
                                              /* partial sums
   synch();                                   /*synchronize first
   if (half%2 != 0 && Cn == 0)
        sum[0] = sum[0] + sum[half-1];
   half = half/2
   if (Cn<half) sum[Cn] = sum[Cn] + sum[Cn+half];
until (half == 1);                            /*final sum in sum[0]
```

An Example with 10 Cores (10 Threads)

- `synch()`: Cores (theads) must synchronize before the "consumer" core (thread) tries to read the results from the memory location written by the "producer" core (thread)
  - Barrier synchronization – cores wait at the barrier, not proceeding until every core has reached it



Barrier Implemented with Spin-Locks

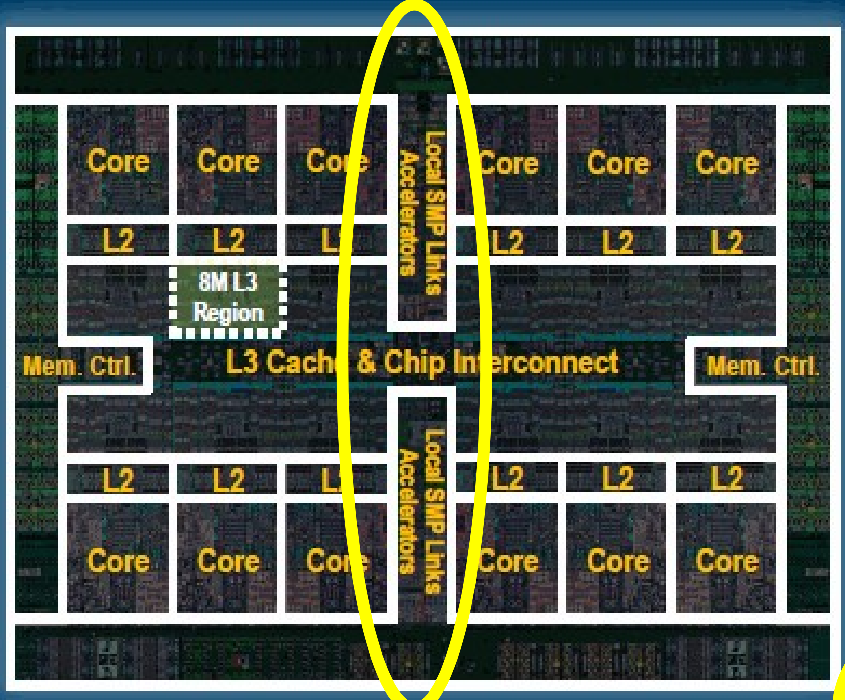
- `n` is a shared variable initialized to the number of cores, `count` is a shared variable initialized to 0, `arrive` and `depart` are shared spin-lock variables where `arrive` is initially unlocked and `depart` is initially locked

```
procedure synch()
  lock(arrive);
    count := count + 1;    /* count the cores/threads
    if count < n           /* as they arrive at
                           /* the barrier
        then unlock(arrive)
        else unlock(depart);
  lock(depart);
    count := count - 1;    /* count the cores/threads
    if count > 0           /* as they leave barrier
        then unlock(depart)
        else unlock(arrive);
```

IBM

POWER8 Processor

Technology
- 22nm SOI, eDRAM, 15 ML 650mm2



Cores
- 12 cores (SMT8)
- 8 dispatch, 10 issue, 16 exec pipe
- 2X internal data flows/queues
- Enhanced prefetching
- 64K data cache, 32K instruction cache

Accelerators
- Crypto & memory expansion
- Transactional Memory
- VMM assist
- Data Move / VM Mobility

Energy Management
- On-chip Power Management Micro-controller
- Integrated Per-core VRM
- Critical Path Monitors

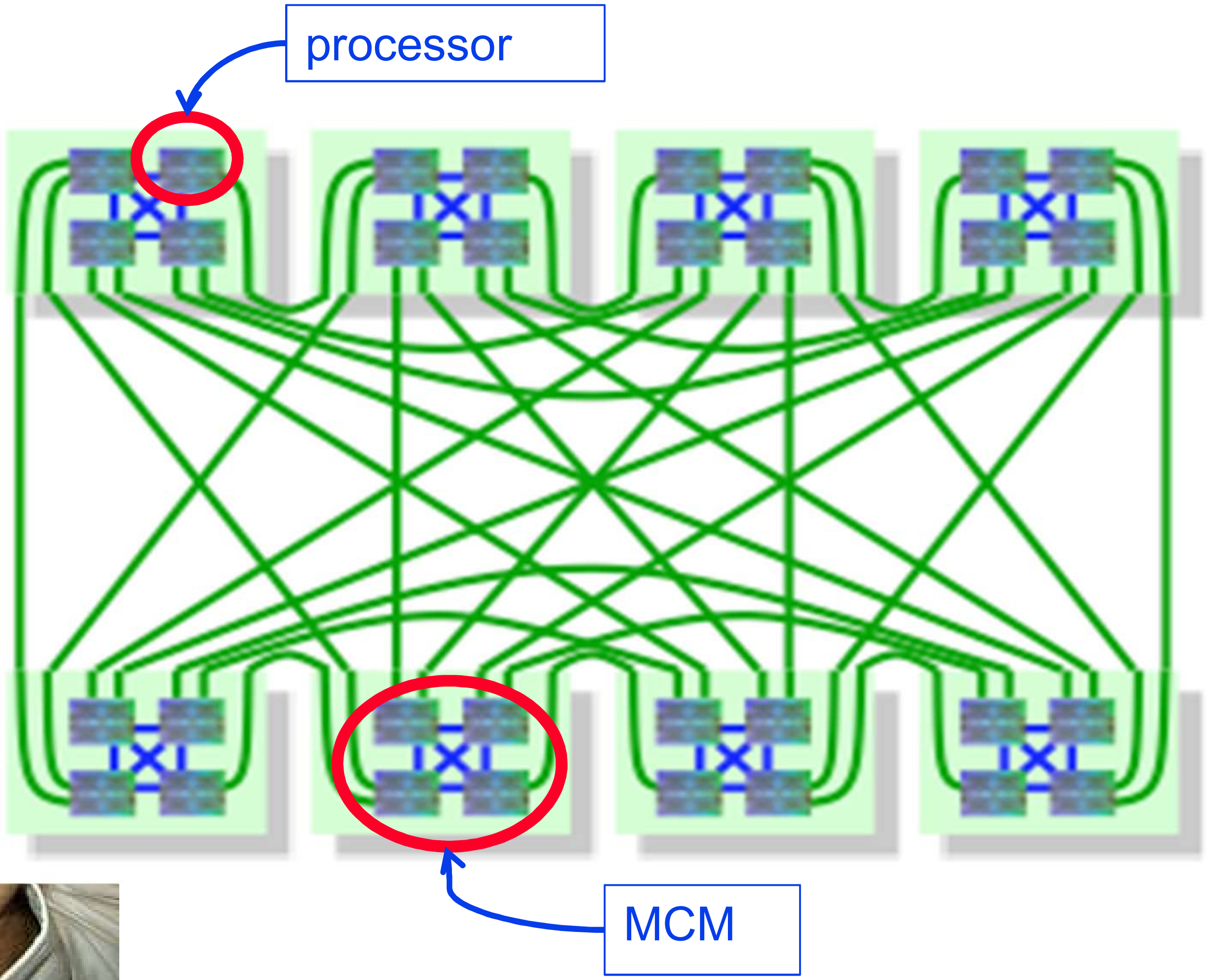
Caches
- 512 KB SRAM L2 / core
- 96 MB eDRAM shared L3
- Up to 128 MB eDRAM L4 (off-chip)

Memory
- Up to 230 GB/s sustained bandwidth

Bus Interfaces
- Durable open memory attach interface
- Integrated PCIe Gen3
- SMP Interconnect
- CAPI (Coherent Accelerator Processor Interface)

SMP Support "Guess" (based on Power7)

- Up to 360 GB/s SMP bandwidth per processor (8 cores in the Power 7, 12 in the Power 8)
  - 12*4=48 cores per MCM
  - 48*8=384 cores per system



Key SMP Multiprocessor Design Questions

- Q1 – How do they share data?

  A single physical address space shared by all cores

- Q2 – How do they coordinate?

  Program threads on cores coordinate/ communicate through atomic operations on shared variables in memory (via loads and stores)

- Q3 – Scalability - How many cores can be supported?

			# of Cores
Communication model	SMP	NUMA	8 to 256 +
		UMA	2 to 32
Physical interconnect	Network		8 to 256 +
	Bus		2 to 8