Lecture 3

Memory Hierarchies

Address Space Organization

Performance and Scalability

Administrivia

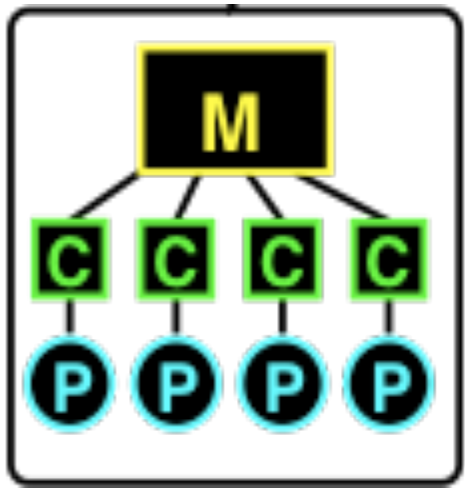
- Make sure you can send mail to, and receive mail from, your ucsd.edu address
- A1 due on Thursday in class: hardcopy
  (electronic turnin: follow instructions)
- Web board
  - Lab partnerships, project partnerships
  - Confirm (by email) that you need a Lincoln account by perusing Moodle
- Office hours
  - Today (after class)
  - Weds: 2:30 to 4:30
- Makeup lecture on Weds10/7 @ 11AM in EBU3B 1202

Address Space Organization

- We classify the address space organization of a parallel computer according to whether or not it provides global memory
- When there is a global memory we have a "shared memory" or "shared address space" architecture
  - ***multiprocessor*** vs ***partitioned global address space***
- Where there is no global memory, we have a "shared nothing" architecture, also known as a ***multicomputer***

Multiprocessor organization

- The address space is global to all processors
- Hardware automatically performs the global to local mapping using address translation mechanisms
- Two types, according to the uniformity of memory access times
- **UMA**: Uniform Memory Access time
  - In the absence of contention all processors observe the same memory access time
  - Also called **Symmetric Multiprocessors**
  - Usually bus based



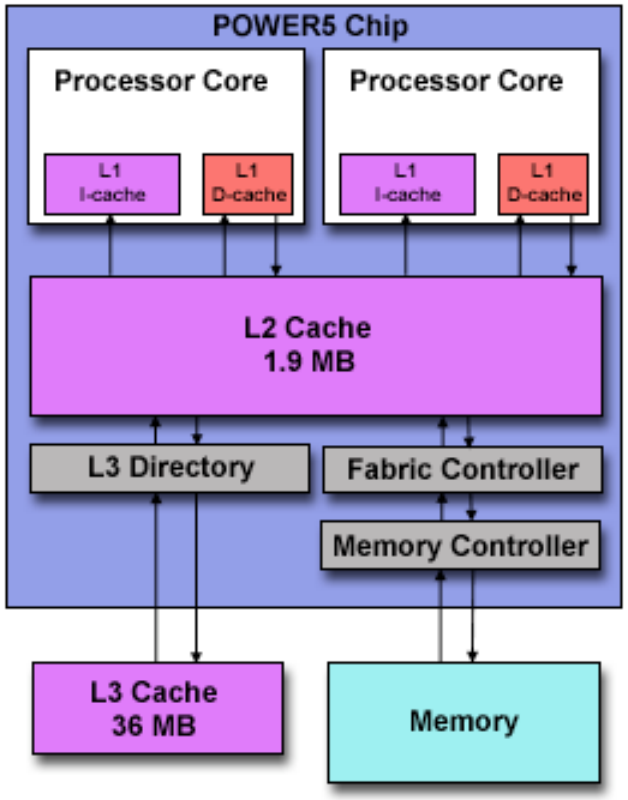
Power 5 memory hierarchy

L1 D-cache: 32KB / proc
128 B line, 4-way assoc

L2: 1.9MB / chip (shared)
128 B line, 10-way assoc

L3: 36 MB / chip (shared)
256B line, 12-way assoc

80 CP access time



L1 I-cache: 64KB / proc
128 B line, 2-way assoc

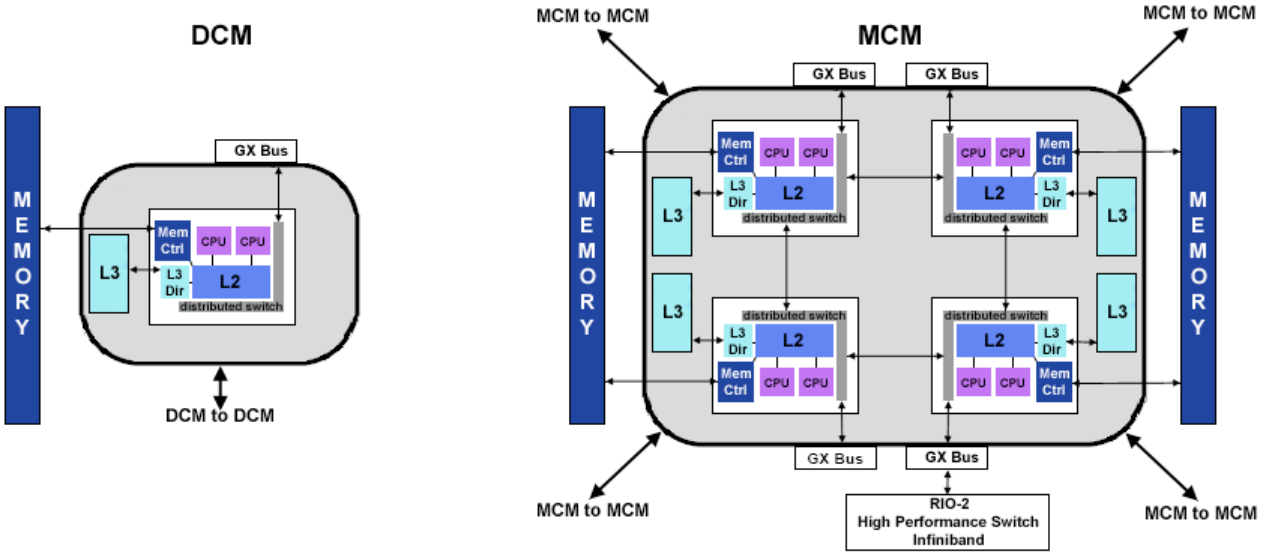
On-chip memory controller &
L3 cache directory

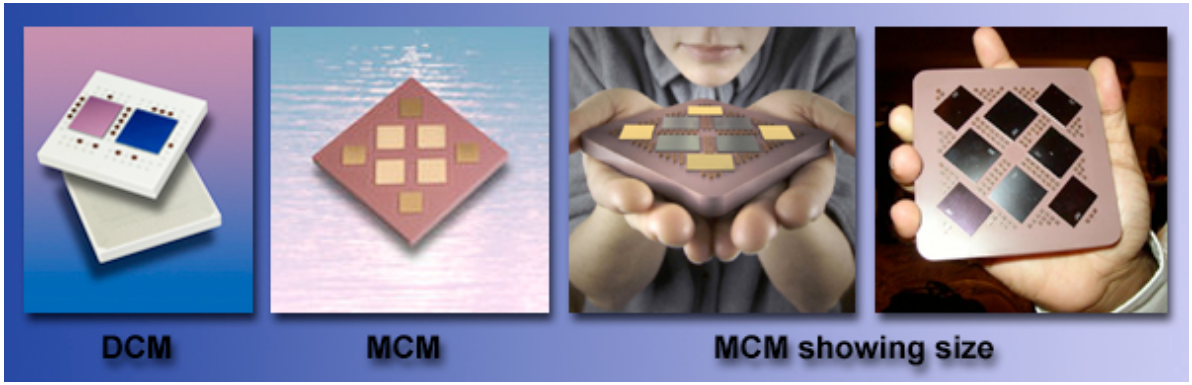
1 GB - 256 GB memory

~16 GB/sec mem-CPU BW

http://computing.llnl.gov/tutorials/ibm_sp (IBM)

Multichip module

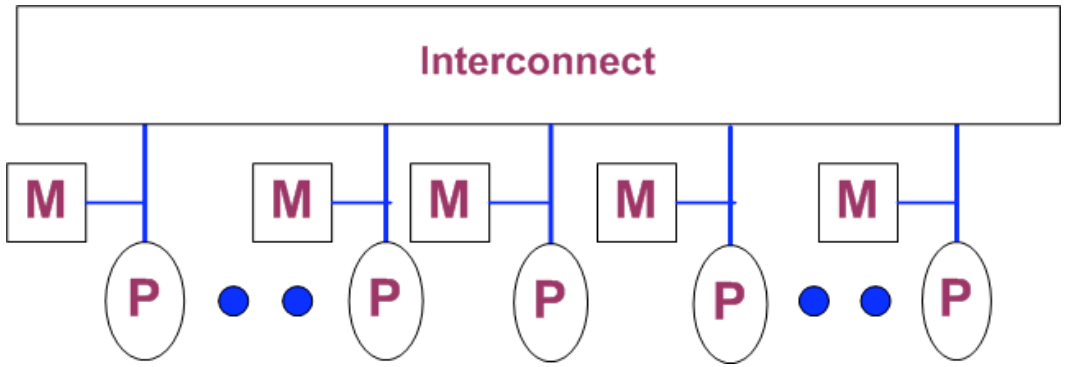




http://computing.llnl.gov/tutorials/ibm_sp (IBM)

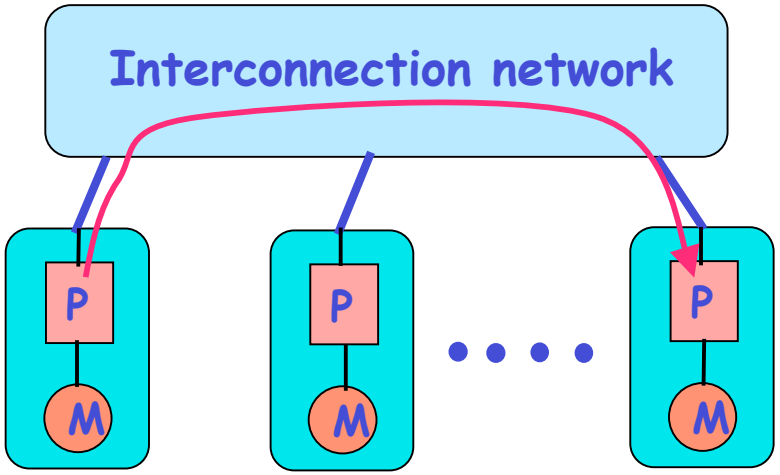
NUMA

- Non-Uniform Memory Access time
  - Processors see distant-dependent access times to memory
  - Implies physically distributed memory
- We often call these ***distributed shared memory architectures***
  - Commercial example: SGI Origin Altix, up to 512 cores
  - Elaborate interconnect with a directory structure to monitor sharers



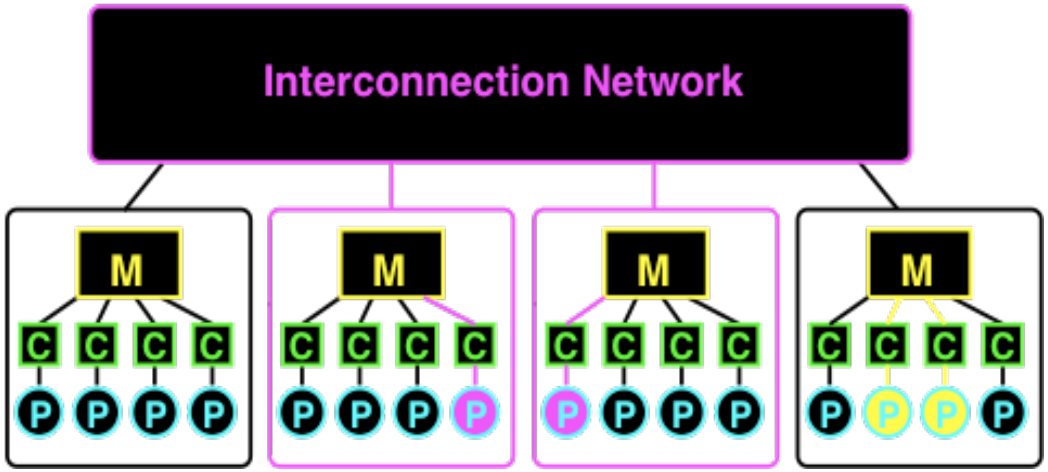
Architectures without shared memory

- Each processor has direct access to local memory only
- Send and receive messages to obtain copies of data from other processors
- We call this a *shared nothing* architecture, or a *multicomputer*



Hybrid organizations

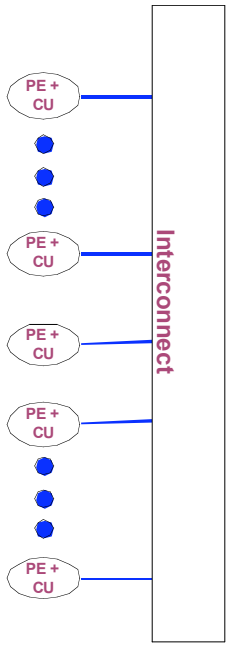
- Multi-tier organizations are hierarchically organized
- Each node is a multiprocessor, usually and SMP
- Nodes communicate by passing messages, processors within a node communicate via shared memory
- All clusters and high end systems today



Flynn's classification (1966)

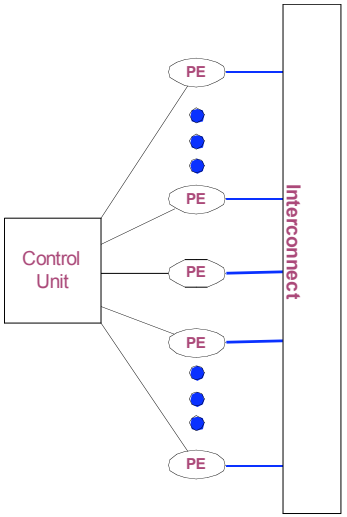
Control mechanism

how do the processors issue instructions?



SIMD: Single Instruction, Multiple Data

Execute a global instruction stream in lock-step

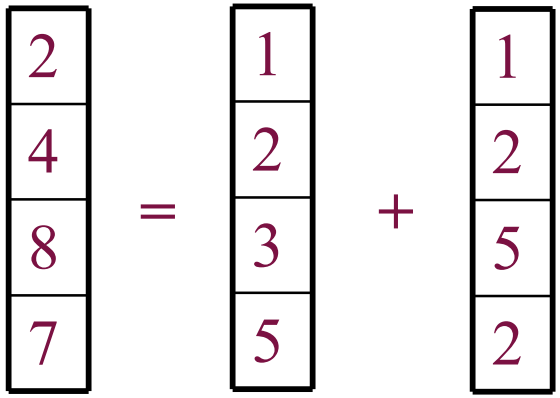


MIMD: Multiple Instruction, Multiple Data

Clusters and servers processors execute instruction streams independently

SIMD (Single Instruction Multiple Data)

- Operate on regular arrays of data
- Two landmark SIMD designs
  - ILIAC IV (1960s)
  - Connection Machine 1 and 2 (1980s)
- Vector computer: Cray-1 (1976)
- Intel and others support SIMD for multimedia and graphics
  - SSE
    Streaming SIMD extensions, Altivec
  - Operations defined on vectors
- GPUs, Cell Broadband Engine
- Reduced performance on data dependent or irregular computations




```
forall  i = 0 : n-1
    x[i] = y[i] + z [ K[i] ]
end forall
```

```
forall  i = 0 : n-1
   if ( x[i] < 0) then
        y[i] = x[i]
   else
        y[i] =  √x[i]
end forall
```

Measures of Performance

- Why do we measure performance?
- Measure of performance
  - Completion time
  - Processor time product
    Completion time × # processors
  - Throughput: amount of work that can be accomplished in a given amount of time
  - Relative performance: given a reference architecture or implementation
    AKA ***Speedup***

Parallel Speedup and Efficiency

- How much of an improvement did our parallel algorithm obtain over the serial algorithm?
- Define the ***parallel speedup, S_P***

$$S_P = \frac{\text{Running time of the best serial program on 1 processor}}{\text{Running time of the parallel program on P processors}}$$

- T_1 is defined as the running time of the "best serial algorithm"
- In general: *not* the running time of the parallel algorithm on 1 processor
- **Definition**: *Parallel efficiency* $E_P = S_P/P$

What can go wrong with speedup?

- Speedup is not always an accurate way to compare different algorithms or machines
- We might be able to obtain a better speedup at the cost of a longer running time
- We have a *super-linear* speedup when

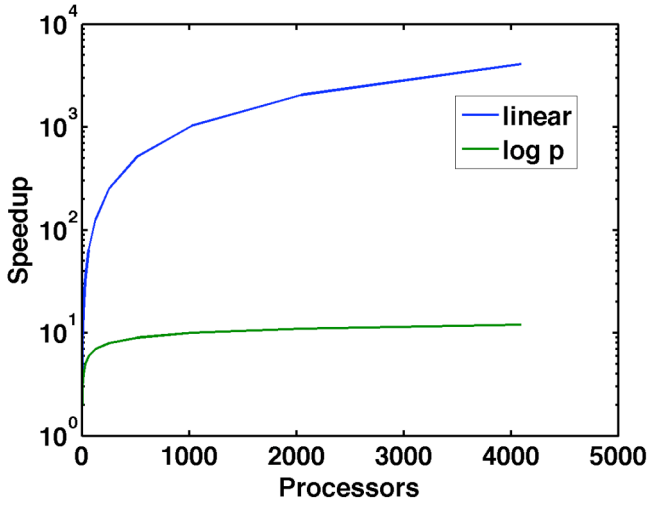
$$E_P > 1 \Rightarrow S_P > P$$

  - Super-linear speedups are often an artifact of inappropriate measurement technique
  - Where there is a super-linear speedup, a better serial algorithm may be lurking

Scalability

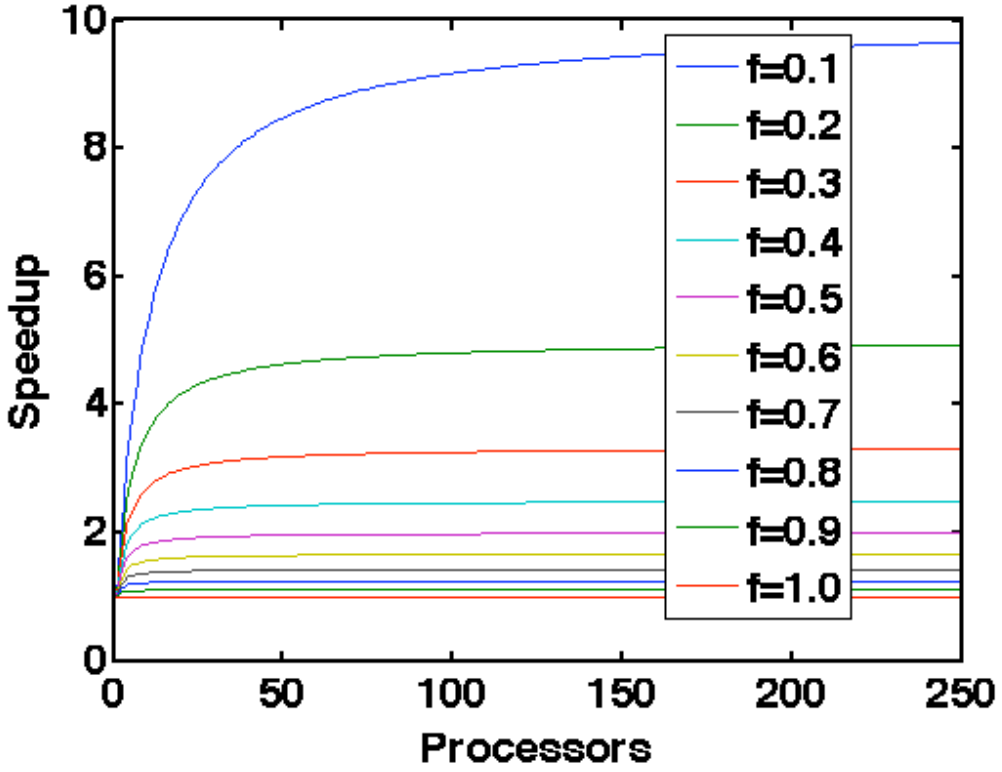
- A computation is **scalable** if performance increases as a "nice function" with the number of processors, e.g. linearly
- In practice scalability can be hard to achieve
  - Serial sections: code that runs on only one processor
  - "Non-productive" work associated with parallel execution, e.g. communication
  - Load imbalance: uneven work assignments over the processors
- Some algorithms present intrinsic barriers to scalability leading to alternatives

```
for  i=0:n-1  sum = sum + x[i]
```



Amdahl's law (1967)

- A serial section limits scalability
- Let f = fraction of T_1 that runs serially
- *Amdahl's Law* (1967) : As $P \rightarrow \infty$, $S_P \rightarrow 1/f$



0.1

0.2

0.3

Scaled Speedup

- Is Amdahl's law pessimistic?
- Observation: Amdahl's law assumes that the workload (W) remains fixed
- But parallel computers are used to tackle more ambitious workloads

  W increases with P

  f often decreases with W

Computing scaled speedup

- Instead of asking what the speedup is, let's ask how long a parallel program would run on a single processor
  [J. Gustafson 1992]
  http://www.scl.ameslab.gov/Publications/Gus/FixedTime/FixedTime.pdf
- Let $T_P = 1$
- f' = fraction of serial time spent on the parallel program
- $T_1 = f' + (1 - f') \times P = S'_P$ = scaled speedup
- Scaled speedup is linear in P

Isoefficiency

- Consequence of Gustafson's observation is that we increase N with P
- Kumar: We can maintain constant efficiency so long as we increase N appropriately
- The *isoefficiency* function specifies the growth of N in terms of P
- If N is linear in P, we have a scalable computation
- More on this later on

Caches

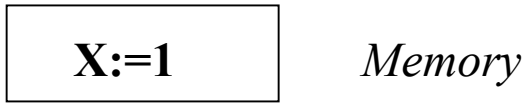
Coherency,
Consistency,
False Sharing

How do cache misses arise?

- The 3 C's
- Cold Start
- Capacity
- Conflict

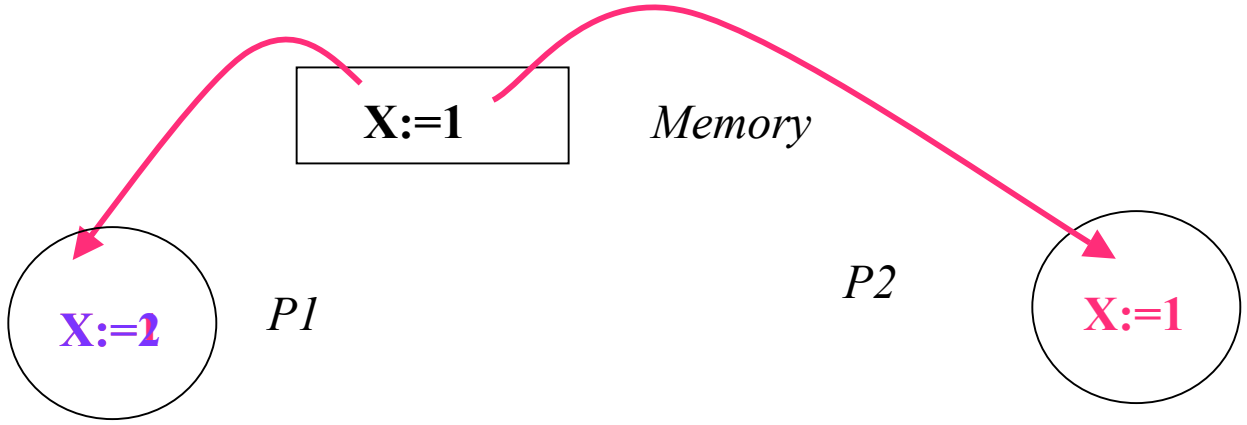
Cache Coherence

- A central design issue in shared memory architectures
- Processors may read and write the same cached memory location
- If one processor writes to the location, all others must *eventually* see the write



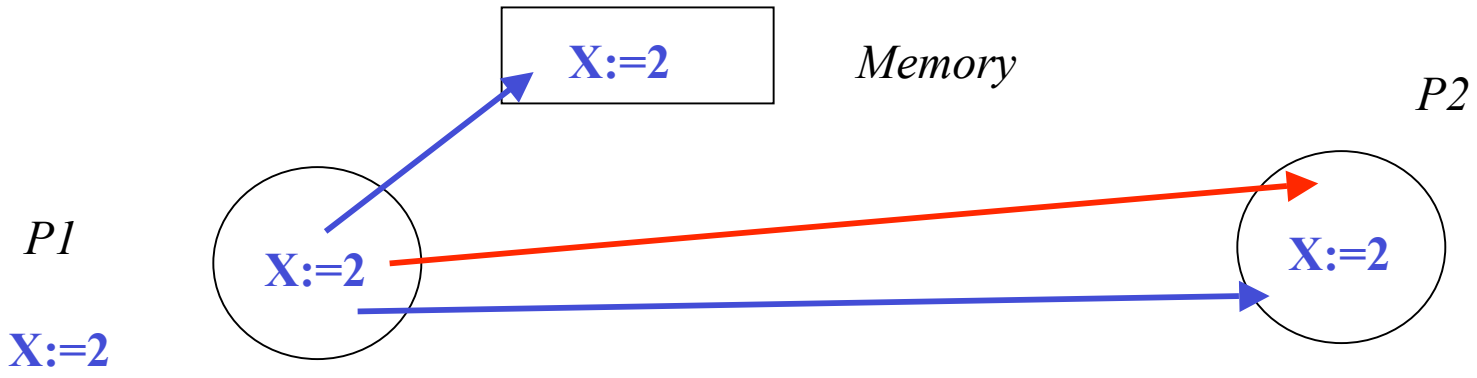
Cache Coherence

- P1 & P2 load X from main memory into cache
- P1 stores 2 into X
- The memory system doesn't have a coherent value for X



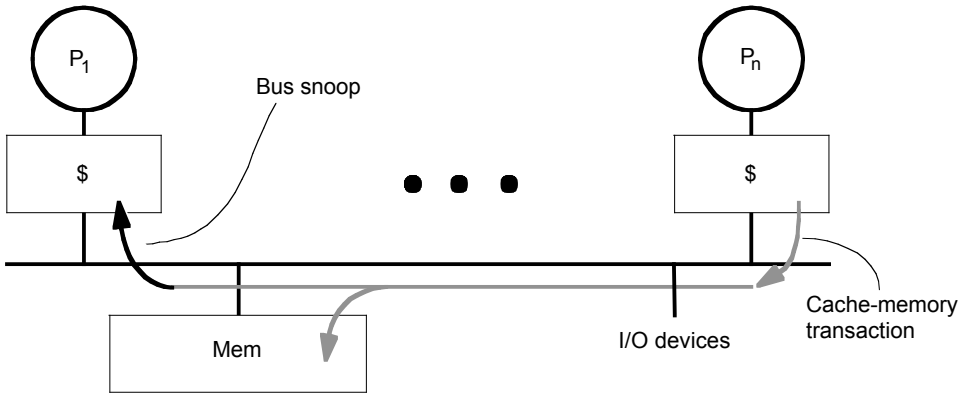
Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write



SMP architectures

- Employ a *snooping protocol* to ensure coherence
- Processors listen to bus activity



Parallel Computer Architecture, Culler, Singh, Gupta

Memory consistency and correctness

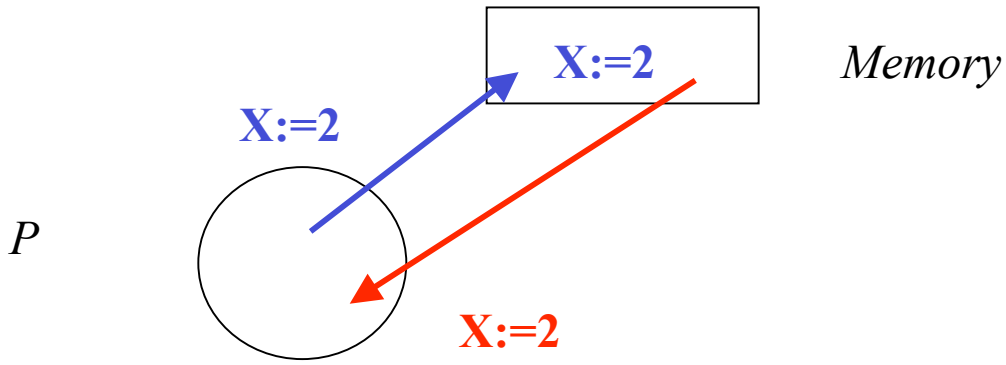
- Cache coherence tells us that memory will *eventually* be consistent
- The memory consistency policy tells us *when* this will happen
- Even if memory is consistent, changes don't propagate instantaneously
- These give rise to correctness issues involving program behavior

Memory consistency

- A memory system is consistent if the following 3 conditions hold
  - Program order
  - Definition of a coherent view of memory
  - Serialization of writes

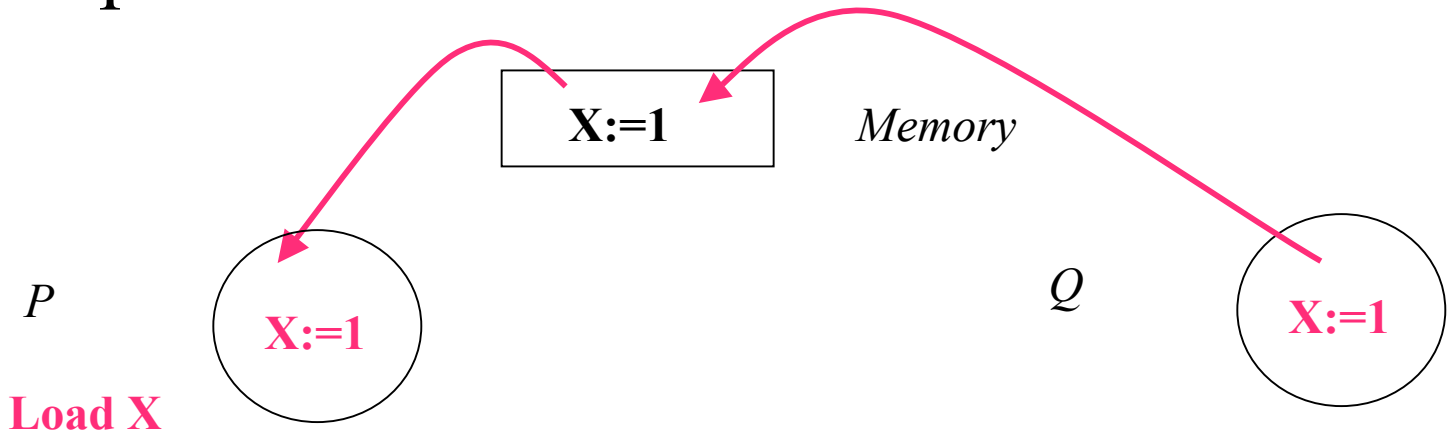
Program order

- If a processor writes and then reads the same location X , and there are no other intervening writes by other processors to X , then the read will always return the value previously written.



Definition of a coherent view of memory

- If a processor P reads from location X that was previously written by a processor Q , then the read will return the value previously written, if a sufficient amount of time has elapsed between the read and the write.



Serialization of writes

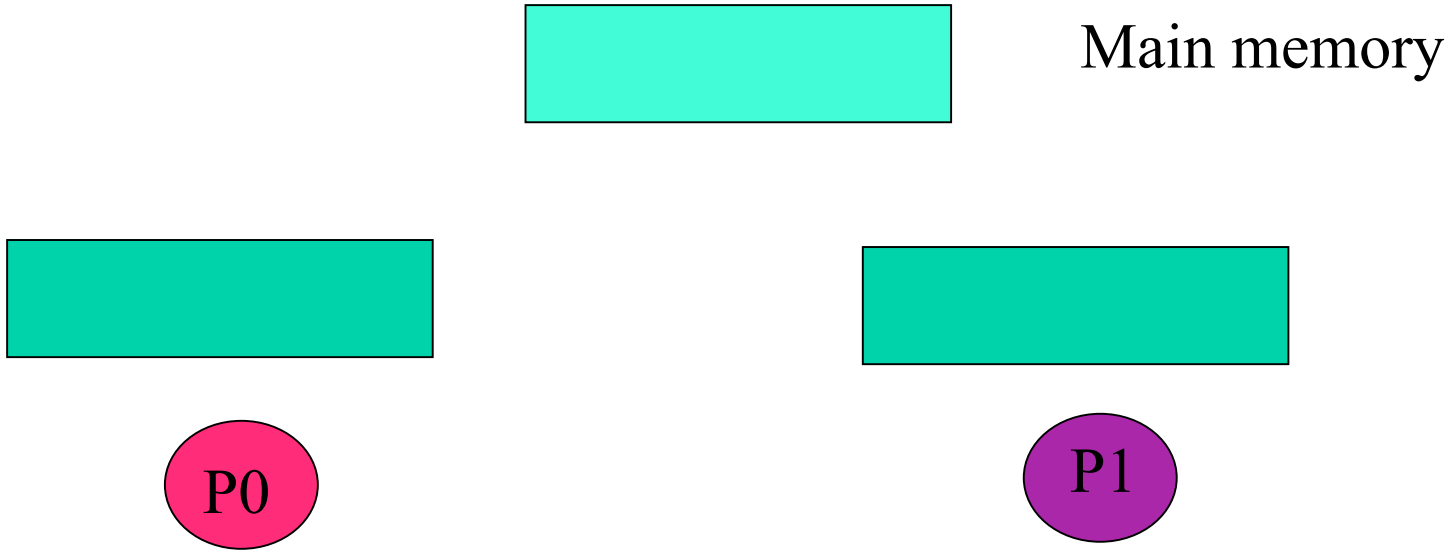
- If two processors write to the same location X, then other processors reading X will observe the same the sequence of values in the order written
- If 10 and then 20 is written into X, then no processor can read 20 and then 10

Memory consistency model

- The memory consistency model determines when a written value will be seen by a reader
- **Sequential Consistency** maintains a linear execution on a parallel architecture that is consistent with the sequential execution of some interleaved arrangement of the separate concurrent instruction streams
- Expensive to implement
- **Relaxed consistency**
  - Enforce consistency only at well defined times
  - Useful in handling false sharing

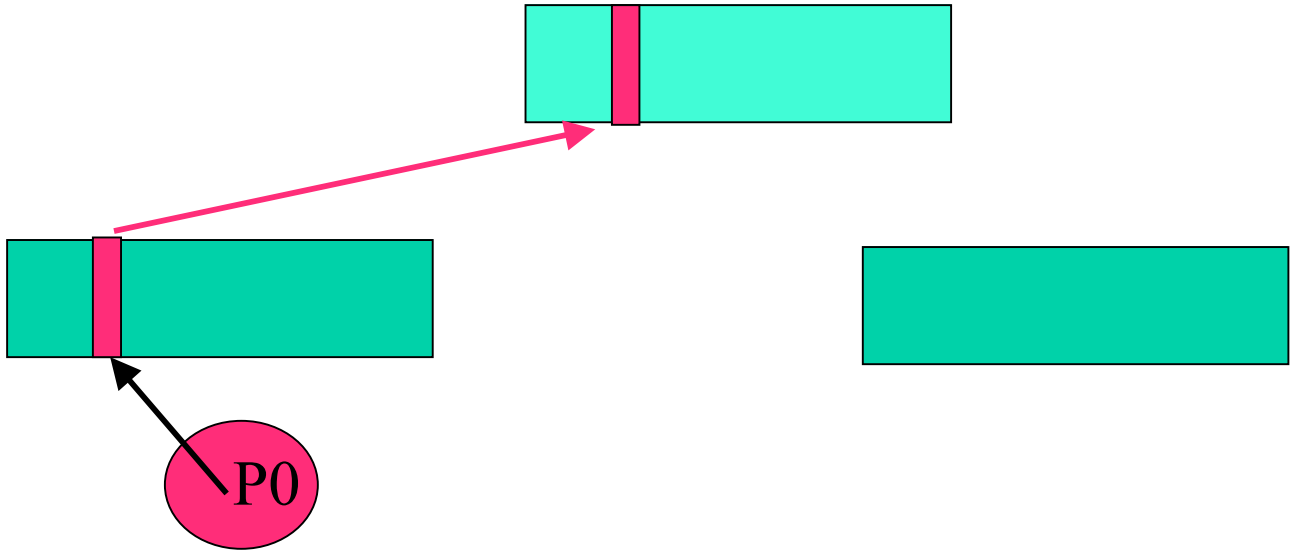
False sharing

- Consider two processors that write to different locations mapping to different parts of the same cache line



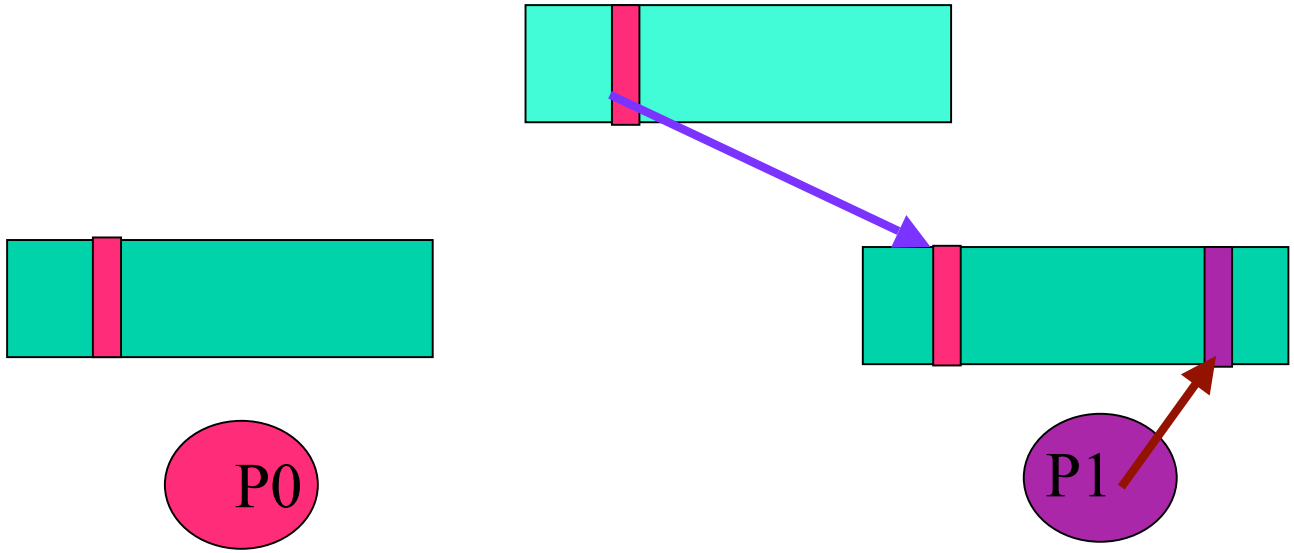
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated



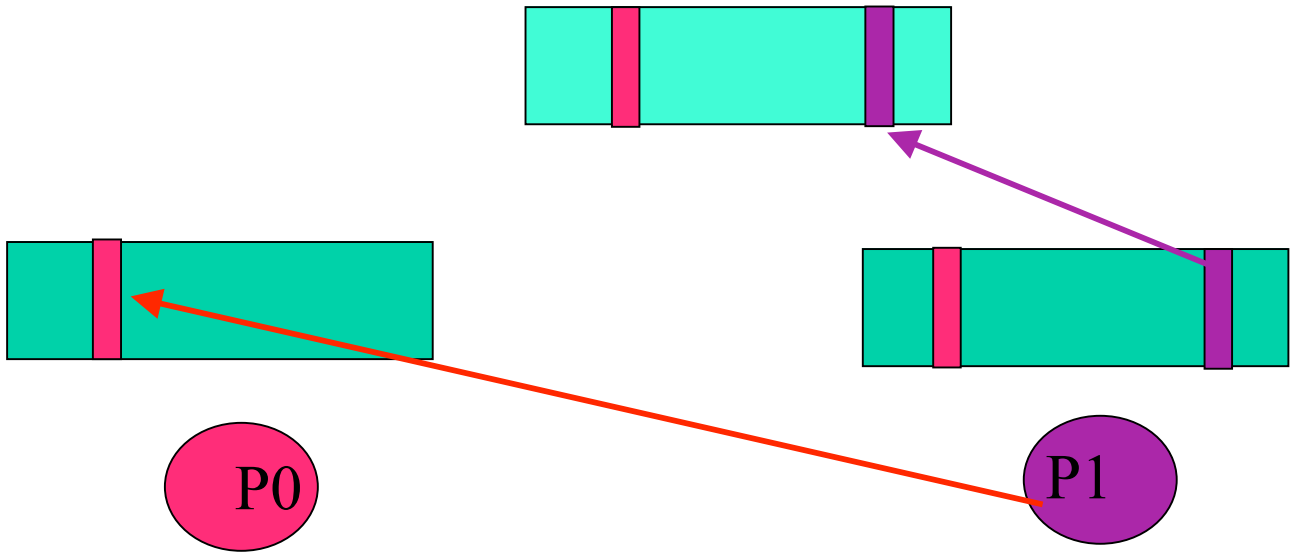
False sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory



False sharing

- P1's write updates main memory
- Snooping protocol invalidates the corresponding block in P0's cache



False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another's cache