Multiprocessors

- Are we reaching performance limits in uniprocessors?
- Performance enhancements are realized thru improvements in:
  - Architecture
  - Technology
- Panel session at VTS'99: "The end of Moore's Law era?"
  - Three say yes (within 10 years), two say no
  - Jury is still out on this one
  - However, it is generally believed that the physics of the process, e.g. the size of an atom, will impose a hard limit
- With reference to Moore's law:
  - "All exponentials in nature eventually saturate."
  - What is the scaling factor of the x-axis?
  - Where are we today on the curve?

Why parallel machines?

- What about improvements in architecture?
  - Uniprocessor improvements reaching a point of diminishing returns!
  - Parallel machines appear to be a natural candidate as a successor to the uniprocessor
- Multiprocessors: cost effective way to improve performance
  - It is unlikely that architectural innovations can be sustained indefinitely
    - ⇒ Analogous to the physical laws that limit technology except in reference to complexity
  - Instead, connect multiple uniprocessors together
  - There has been steady progress on the major obstacle to widespread use of parallel machines => ***software***
- Focus on the mainstream of multiprocessor design
  - Machines with small to medium numbers of processors (<100)
  - Viable architectures with more than 100 CPUs are difficult to predict

Classifying parallel architectures

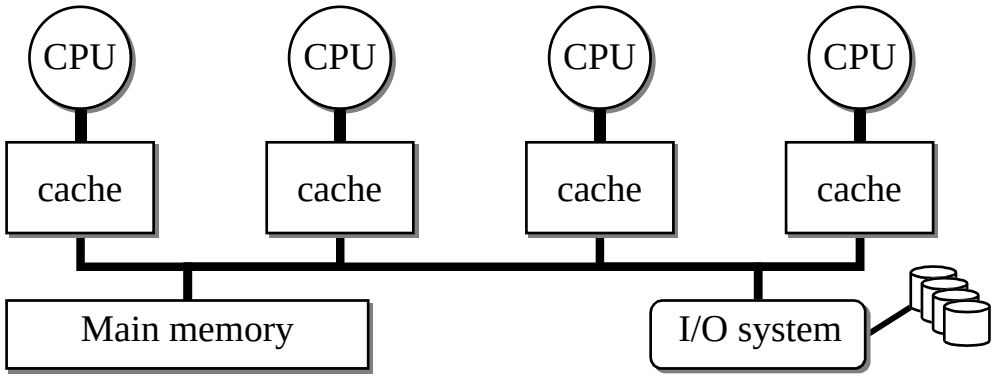
- Flynn's classification => based on parallelism in the instruction and data streams
- SISD (Single instruction & data stream): uniprocessor
- SIMD (Single instruction stream, multiple data streams)
  - Same instruction is executed by many CPUs on different data streams
  - Each processor has its own data memory
  - Only a single instruction memory and control processor which fetches and dispatches instructions
- MISD (Multiple instruction streams, single data stream)
  - No commercial versions built, but perhaps systolic processors?
- MIMD (Multiple instruction streams, multiple data streams)
  - Each CPU fetches its own instructions and operates on its own data
  - Often built using off-the-shelf microprocessors

Classifying parallel architectures

- SIMD model was popular through the 80s
  - Examples include the MasPar and Connection Machine (Thinking Machines)
  - However, less popular today => too expensive to develop
- MIMD model has clearly emerged as the architecture of choice in recent years
  - MIMD offers flexibility
  - Can operate as a single-user machine providing high performance for one application
  - Can operate as multiprogrammed machines running many tasks simultaneously
- MIMDs can build on the cost/performance advantages of off-the-shelf microprocessors & systems

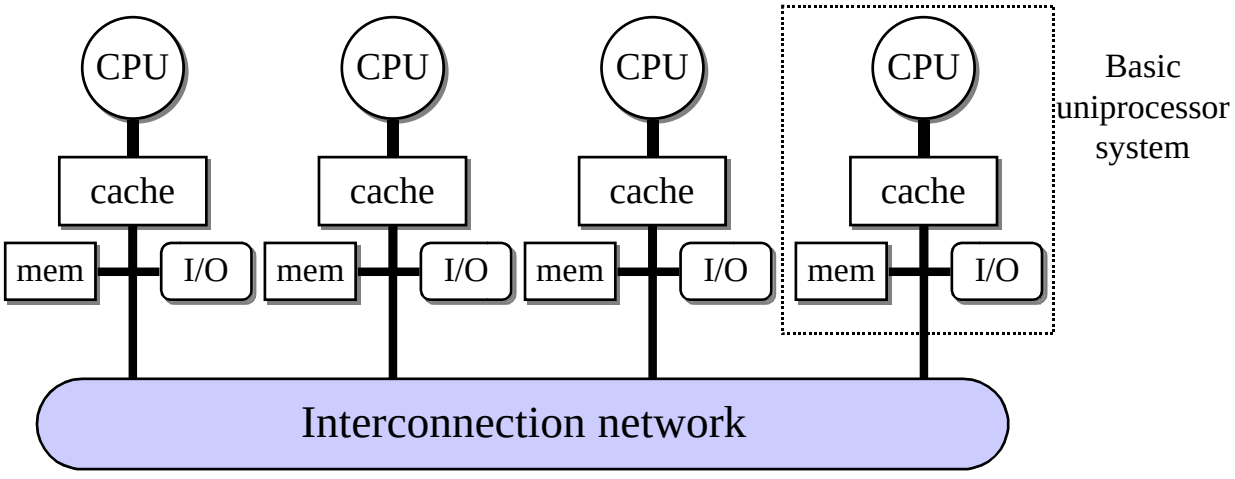
Centralized shared memory architecture

- Two basic types of MIMD architectures:
  - Centralized shared memory (*Uniform Memory Access*, or *UMA*)
  - Distributed shared memory (DSM)
- Centralized shared memory
  - At most a few dozen processors which share a bus and a single main memory
  - Large caches allow the bus and memory organization to satisfy the memory demands of a small number of processors



Distributed memory

- Supports larger processor counts by distributing the memory and allowing multiple memories to work in parallel
- Increases in processor bandwidth requirements => distributed memory beats out centralized shared memory for smaller groups of processors



Distributed vs. centralized shared memory

- Distributing the memory among the nodes has two major advantages
  - It's a cost-effective way to scale the memory bandwidth if most accesses are to local memory in the node
  - It reduces the latency for accesses to local memory, due to less contention
- Distributed memory has some disadvantages as well
  - Communicating data between processors becomes more complex
  - Interprocessor communication has higher latency
- Key characteristics that distinguish among distributed memory machines are
  - How communication is performed.
  - The architecture of the distributed memory

Distributed memory architecture models

- Physically separate memories can be addressed as one logically shared address space
  - The address space is shared—all processors see the same address space
  - These machines are referred to as *NUMA* (*Non-Uniform Memory Access*) in contrast to the centralized UMA machines
- Multicomputer architecture
  - Multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor
  - An associated communication mechanism used for exchanging data
- For DSM, shared memory can be used to communicate data via load and store operations
- For a multicomputer, communication is done by either synchronous (RPC) or asynchronous message passing

Measuring communication bandwidth

- Three performance metrics are critical in any communication mechanism
  - Communication bandwidth
  - Communication latency
  - Communication latency hiding
- Communication bandwidth
  - Bisection bandwidth is the bandwidth across the "narrowest" part of the interconnection network
  - Bandwidth in and out of an individual processor is also important
  - Bandwidth is affected by the architecture within the node and by the communication mechanism
    - When communication occurs, resources are tied up or occupied
    - This can prevent other communication
  - *Occupancy* can limit communication bandwidth

Measuring communication latency

- Lower communication latency is better (of course)
- Communication latency =
  Sender overhead + Time of flight + Transport latency + Receiver overhead
  - Time of flight is preset
  - Transport latency is determined by interconnection network
  - Sender and receiver overhead are determined by communication mechanism
- Complex mechanisms (i.e. for naming and protection) increase latency, particularly those that require the OS
- Latency affects performance either by
  - Causing the processor to wait
  - Tying up processor resources

Hiding communication latency

- How well can the mechanism hide latency by overlapping communication with computation or with other communication?
  - For example, a system that only allows access to a word at a time may have low latency
    - However, it may be unable to hide the latency because each word transferred is treated as a cache miss
  - Another machine may have a higher latency but allow the processor to do other things while waiting for data
- Examples of latency hiding techniques for shared memory will be given later
- Latency hiding is more difficult to measure than the previous two and is application dependent

Performance metrics for communication

- These performance metrics are affected by
  - The size of the data items being communicated by the application
    - ⇒ Size affects the latency and bandwidth in a direct way
  - The effectiveness of the different latency hiding techniques
  - The regularity in the communication patterns.
    - ⇒ These two affect the cost of naming and protection (communication overhead)
- An ideal mechanism would perform well with
  - Large and small data requests.
  - Regular and irregular communication patterns

DSM vs. message passing

- Shared-memory advantages
  - Compatibility with well-understood mechanisms in centralized SM
  - Ease of programming, particularly for systems in which communication patterns are complex or vary dynamically during execution
  - Low overhead for communication: hardware used to enforce protection
  - The ability to use hardware-controlled caching => reduces the frequency of remote communication
- Message-passing advantages:
  - Simpler hardware (especially with respect to building coherent caches)
  - Explicit communication forces programmers and compiler writers to pay attention to what is costly and what is not: is this an advantage?
- Shared-memory communication is more popular today
- Centralized schemes still dominate
  - However, long-term trends favor distributed memory

Challenges of parallel processing

- Amdahl's law applies to parallel processing as well
  - Any program has a parallel portion and a serial portion.
  - The parallel portion is the only part that is sped up by having multiple processors
  - As with uniprocessors, speedup is limited by the fraction of the original program that can be parallelized
- For example, suppose we want to achieve a speedup of **80** with **100** processors
  - What is the fraction of the original computation that can be sequential?
  - With simplifying assumptions (see text):

$$80 = \frac{1}{\frac{Fraction_{parallel}}{100} + (1 - Fraction_{parallel})}$$

$$0.8 \times Fraction_{parallel} + 80 \times (1 - Fraction_{parallel}) = 1$$

$$Fraction_{parallel} = 0.9975$$

Challenges of parallel processing

- The second major challenge involves the large latency of remote memory access
  - This may cost anywhere from 50 clocks to 10,000 clocks!
- The latency is dependent on
  - The communication mechanism
  - The type of interconnection network
  - The scale of the machine
- Insufficient parallelism can be attacked in software with new algorithms that have better parallel performance
- Long communication latency can be attacked by
  - Architecture (caching)
  - Programmer (restructuring the data)
- Focus on techniques for reducing the impact of high latency

Caching in CSM architectures

- The use of large multilevel caches can substantially reduce memory bandwidth demands of a processor
  - This has made it possible for several CPUs to share the same memory through a shared bus
- Caching supports both private and shared data
  - For private data, once cached, its treatment is identical to that of a uniprocessor.
  - For shared data, the shared value may be replicated in many caches
- Replication has several advantages:
  - Reduced latency and memory bandwidth requirements
  - Reduced contention for data items that are read by multiple processors simultaneously
- However, it also introduces a problem: *cache coherence*

Cache coherence

- With multiple caches, one CPU can modify memory at locations that other CPUs have cached
- For example:
  - CPU A reads location *x*, getting the value *N*
  - Later, CPU B reads the same location, getting the value *N*
  - Next, CPU A writes location *x* with the value *N* - 1
  - At this point, any reads from CPU B will get the value *N*, while reads from CPU A will get the value *N* - 1
- This problem occurs both with write-through caches and (more seriously) with write-back caches
- Cache coherence (an informal definition): a memory system is coherent if any read of a data item returns the most recently written value of that data item

Cache coherence definitions

- Coherence defines what values can be returned by a read
- A memory system is coherent if:
  - Read after write works for a single processor
    - If CPU A writes *N* to location *X*, **all** future reads of location *X* will return *N* if no other processor writes location *X* after CPU A
  - Other processors' writes eventually propagate.
    - If CPU A writes value *N* to location *X*, CPU B will eventually be able to read value *N* from location *X*
    - Once it does so, it will continue to read value *N* until location *X* is written again
- This is our intuitive notion of a coherent view of memory

Cache coherence & consistency

- Coherence: writes to a single location are serialized
  - If CPUs A and B both write to location *X*, all processors see the same order of the writes
  - This does not mean that all reads must return the same value
    - If value *N1* is written "first" to location *X*, followed closely by reads of *X* and a write of *X* with value *N2*, some reads may return *N1* and some *N2*
  - However, a processor that reads *N2* will return *N2* for all future reads
- Consistency
  - This indicates when a modification to memory is seen by other processors (i.e. will be returned by a read)
  - Clearly, this ***can't*** be "instantaneous" since it may be that the new value has not even left the processor when a read occurs
  - Issue: when is a write visible to other processors?

Cache consistency

- Consistency issue: when must a written value be seen by a reader?
  - This is defined by a memory consistency model
  - For now, assume that a write is not complete until all processors have "seen" the effect of the write
  - Also, assume that a processor may not reorder memory accesses to move reads before an outstanding write
    - Reads can be reordered, but reads and writes can not be interchanged
- Coherent caches provide both
  - Replication of shared data items (reduces latency and contention)
    - Provide multiple copies of data so that several processors can access a single piece of memory without serialization
  - Migration of data items (reduces latency)
    - Data items are moved from one processor to another as needed

Cache coherence protocols

- Small-scale multiprocessor use hardware mechanisms to track the state of data blocks that are shared
- Two types of protocols
  - Directory based
    - The sharing status of a block of physical memory is kept in one location (the directory)
    - Interprocessor communication is used to maintain coherence
  - Snooping
    - The sharing status is distributed and kept with the block in each cache
    - The caches are usually on a shared memory bus
    - The cache controllers snoop the bus to watch for transactions that occur on data blocks that they hold

Bus snooping protocol: write invalidate

- Write invalidate is the most common protocol, both for snooping and for directory schemes
- The basic ideas behind this protocol:
  - Writes to a location invalidate other caches' copies of the block
  - Reads by other processors on invalidated data cause cache misses
  - If two processors write at the same time, one wins and obtains exclusive access
- Example assumes a write-back cache

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of mem location *X*
CPU A reads *X*	Cache miss	0	-	0
CPU B reads *X*	Cache miss	0	0	0
CPU A writes 1	Invalidate	1	-	0
CPU B reads *X*	Cache miss	1	1	**1**

Bus snooping protocol: write update

- An alternative is to update all cached copies of the modified data item
  - This is called *write update* or *write broadcast*
- To reduce bandwidth requirements, this protocol keeps track of whether or not a word in the cache is shared
  - If not, no broadcast is necessary
- Example again assumes a write-back cache

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of mem location *X*
CPU A reads *X*	Cache miss	0	-	0
CPU B reads *X*	Cache miss	0	0	0
CPU A writes 1	Broadcast	1	1	**1**
CPU B reads *X*	Cache hit	1	1	1

Comparing bus snooping protocols

- Write invalidate is much more popular than write update
- Write update requires more system-wide notifications
  - Multiple writes to the same word with no intervening reads require multiple broadcasts
  - With multiword cache blocks, each word written requires a broadcast
  - Delay between write by one processor and read by another is lower
- Write invalidate uses fewer system-wide notifications
  - The first word written invalidates the entire block
  - Write invalidate works on blocks, while write broadcast works on individual words or bytes
  - Reading an invalidated block causes a miss (somewhat slower)
- Since bus and memory bandwidth is more important in a bus-based multiprocessor, write invalidate performs better

Implementing the write-invalidate protocol

- Write invalidate is simple in bus-based schemes
  - Acquire the bus and broadcast the address to be invalidated
  - Since all processors snoop the bus, they can check the address against items in their cache
- Bus acquisition serializes writes to a memory location
  - Writes to a shared data item cannot complete until the bus is acquired
- How is a data item located when a cache miss occurs?
  - For write-through, it's in memory
  - For write-back, snooping can be used: if a processor finds that it has a dirty copy of the requested cache block, it provides the block instead of memory
- Write-back caches are greatly preferred in a multiprocessor environment since they reduce memory bandwidth

Writes in write-invalidate protocols

- Writes are an issue with cache coherence protocols in general
- The CPU needs to know if any other caches contain the block to be written by a processor.
  - If there are none, then the write need not be placed on the bus, reducing the time to complete the write and reduces memory bandwidth
- This can be tracked by adding an *extra state bit* (in addition to the valid and dirty bits) that indicates if the block is shared
  - If the bit is set (the block is shared), the cache generates an invalidation on the bus and marks the block as private
  - If another processor later requests the block, the miss is snooped and the "owner" sets the state bit to shared

Optimizations for tag checking

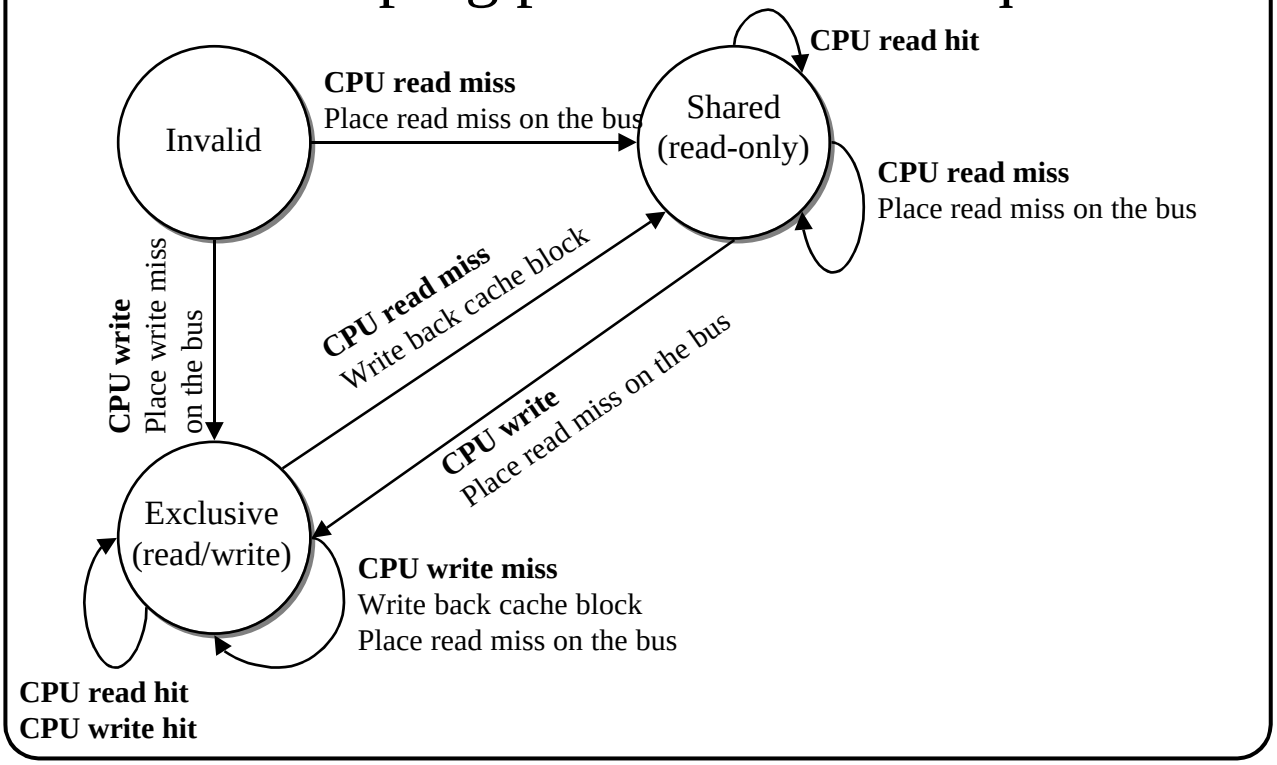
- Note that every bus transaction checks cache-address tags — this could potentially interfere with CPU cache access
- Reduce interference by
- Duplicate the tags: bus access proceeds in parallel with CPU
  - On misses, the processor arbitrates for and updates both sets of tags
  - Snoop also does this to perform an invalidate or to update the shared bit
  - However, a snoop may require fetching a block, thus stalling
- Employing a multilevel cache with inclusion
  - Snooping is directed to L2, where there are fewer processor accesses
  - If a snoop gets a hit, then it must arbitrate for L1 to update state and possibly retrieve data, usually stalling the processor
  - Since it is popular to use multi-level caches in multiprocessors (to reduce memory bandwidth), this solution is usually adopted

Sample bus snooping protocol

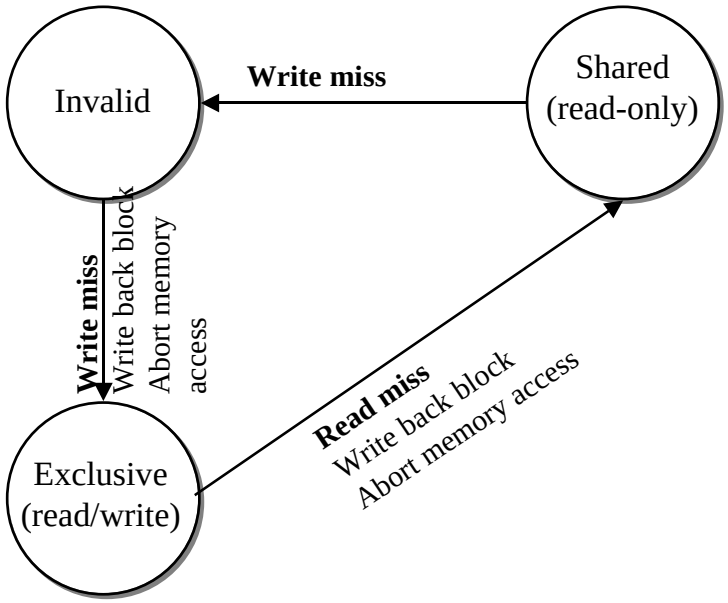
- Implemented by incorporating a finite state controller in each node
  - The controller responds to requests from the processor and bus
- To simplify the controller, write hits and write misses to shared blocks are treated as write misses
  - This causes processors with copies to invalidate them

Request	Source	Function
Read hit	Processor	Read data in cache
Write hit	Processor	Write data in cache
Read miss	Bus	Request data from cache or memory
Write miss	Bus	Request data from cache or memory, and perform any needed invalidates

Bus snooping protocol: CPU requests



Bus snooping protocol: bus requests



Snooping protocols: wrapping up

- Protocol assumes that operations are atomic
  - In reality, a write miss is not atomic — just too much work to do
  - Also, read misses on a split transaction bus are not atomic
  - Nonatomic actions introduce the possibility of deadlock...
- Real protocols distinguish between write hits and write misses
  - From the shared state, a write miss would require the action shown previously
  - However, a write hit does not require that the data be fetched since it is up-to-date — all that's needed is an invalidate operation
- Real protocols distinguish between shared and clean data in exactly one cache
  - A "clean and private" state eliminates the need to generate a bus transaction on a write to a "clean and private" block