

Multiprocessors

- **Two Types:**

- 1. Centrally Shared Memory: most commercial MPs**

- Use small number of processors (most a few dozen)
- A single centralized memory
- Processor interconnected by a BUS
- Uniform Memory Access (UMA) since all processors must go across the bus

- 2. Physically Distributed Memory: much research**

- No shared central memory
- Need high-bandwidth interconnect
- I/O also distributed among the nodes
- Can use off-the-shelf processor boards that contain memory, cache, processor
- Spatial locality: Most accesses are local & fast

- Two types:
 - (a) Distributed shared memory (DSM)
 - (b) Multicomputers/Message-Passing Machines

1. Distributed Shared Memory (DSM)

- Separate memories can be addressed as a single logical address space
- Any processor can access any memory location
- Same address on two processors identifies a single memory element
- “NUMA”: non-uniform memory access
 - Access to local memory is fast
 - Access to remote memory is slow

2. Multicomputers / Message-Passing Machines

- Not a shared address space
- Multiple private address spaces
- Processors cannot address remote data by virtual address
- Must explicitly transfer data among processors using messages

- **Challenges of Parallel Processing**

- 1. Hard to get linear speedup when increasing number of processors**

- Inherently hard to get that much parallelism out of programs
- Certain parts must be done sequentially
- At times many processors will be idle
- Amdahl's Law example:
- 100 processors: want 80X speedup
- $S = 1 / ((1 - Fe) + Fe/Se)$
- Enhanced mode is parallelized mode
- If $S = 80$, $Se = 100$, then must have $Fe = 99.75\%$
- 99.75% of program must be parallelized--unlikely!

- 2. Large latency of remote accesses**

- Remote access involves:
send overhead + time on interconnect + receive O.H.
- from 50 cycles to 10,000 cycles
- Depends on communication mechanism and type of interconnect
- SGI Challenge, Cary T3D: 1 microsecond
- Intel Paragon, IBM SP-2: 30 to 100 microseconds

- **Centralized Shared Memory Architectures**
- Processors share main memory
- Have individual caches
- Private data: used by a single processor
- Shared data: used by multiple processors
 - Shared data may reside in multiple caches
- **Introduces cache coherence problem:**
 - If multiple caches have same item, must be sure they all see the same value
 - What if one processor writes the item?
- **Example: processor A reads X into cache**
 - The processor B reads X
 - Later A writes X
 - How does B get updated value?
- **Requirements for Coherence**
 1. If processor P writes data item X, and later P reads X, and no other processor writes X, then P reads the last value it wrote into X
 2. If processor Q writes X and then processor P reads X, then P will read the value written by Q
 3. Writes to the same location by any P, Q are serialized: seen in the same order by every processor

- **Cache Coherence Schemes**

- Hardware mechanisms for maintaining coherence
- Protocols: track the state of sharing in a data block

- **Two main types of protocols:**

1. Snooping protocols: used for most centralized shared memory machines
2. Directory-based: used for distributed shared memory machines (scales better)

- **Snooping Protocols**

- Each cache that has a copy of a memory block keeps a copy of the sharing status of the block
- No centralized state is kept
- Cache controllers monitor the shared bus (they “snoop” on the bus)
- May invalidate cache entries, send data, alter sharing state based on what passes on the bus

- **Alternatives for Snooping Protocols**

- **1. Write-Invalidate**

- Before writes can occur, writing processor must gain exclusive permission to hold block in its cache
 - Invalidate all copies in other caches
- Most common protocol
- After invalidation, if another processor tries to read data, will get a cache miss--then fetch an updated copy of data
- If two simultaneous attempts to write: one “wins”, other wait
 - Enforced by arbitration on the bus
- Enforces write serialization

- **2. Write Broadcast or Write Update**

- Update all cached copies of a data item when it is updated
- Broadcast changes over shared bus
- To minimize bus traffic, keep track of whether block is actually shared
 - Only broadcast shared values

- **Invalidate is generally chosen: less bus traffic**
- **Comparison:**
 - Multiple writes to the same word by one processor involve a single invalidate but multiple broadcasts
 - If a multi-word cache block, a single invalidate removes entire block from remote caches, but must broadcast every shared word
 - Disadvantage of invalidate: slower
Item is invalidated, processor gets a cache miss, then fetches the update
- **Implementation**
- Use bus to perform invalidates
 - Processor acquires bus: broadcasts invalidate msg & address
 - All processors constantly snoop: watch addresses
 - If address on bus is in cache, invalidate cache entry
- Bus arbitration enforces write serialization: only one processor can have the bus at a time
- When a cache miss later occurs, must find the data
 - Request it on bus: processors snoop, see request
 - **Write back:** if processor has a dirty copy of needed data, it owns the data: supplies it
 - **Write-through:** get block from memory

- **Distributed Shared Memory Architectures**

- Larger processor count
 - No longer a central memory
 - Each node has some part of memory
 - Still a shared address space
 - Local and remote accesses: non-uniform memory access (NUMA)
-
- Snooping cache not scalable to large # of processors
 - Requires communication with all caches on every miss
 - Problem: no shared data structure that shares state of caches

- **Directory Protocols**

- Directory keeps state of every block that may be cached
 - Which caches may have copies?
 - Is block dirty?
- Associate a directory entry with every memory block
- To prevent performance bottleneck on accessing directory, can distribute it along with memory
- Different directory accesses go to different locations
- Sharing status of a block is always known to be in a single location

- **Possible states of directory entries:**
- **Shared:**
 - one or more processors has data cached
 - values in caches and memory are up-to-date
- **Uncached:**
 - no processor has a copy of cache block
- **Exclusive:**
 - one processor has a copy of the block;
 - the block is dirty;
 - the memory copy is out-of-date;
 - the processor is “owner” of the block
- **Must track all processors that have a copy: invalidate on a write!**
 - Keep a bit vector for each memory block
 - One bit per processor tells whether the processor may be caching the block
 - Don’t broadcast an invalidate message: interconnect is no longer a bus
 - Serialize writes based on directory control
- **Attempts to write non-exclusive data generate a miss**

- **Example protocol actions:**
- **Uncached state, memory has current value:**
 1. Read miss: requesting P gets data from M; block status shared, only P in sharing group
 2. Write miss: requesting P gets value from M, block status exclusive, P is owner
- **Block in shared state, memory value up-to-date**
 1. Read miss: P gets data from MEM, P added to share set
 2. Write miss: P gets value from memory, other processors in the sharing set invalidate their copies, status is exclusive and P is owner
- **Block is in Exclusive state**
 1. Read miss: Owner processor Q has exclusive access, another processor P wants to read
Q receives a fetch message to fetch data value
Q sends data to directory location, which forwards it to P
Status becomes shared, sharing set is P & Q
 2. Data write-back: Owner has dirty copy of block, wants to replace it in cache; Writes block to directory, which updates memory; Status becomes uncached
 3. Write miss: another P wants to write data, Q is owner:
Fetch msg to Q; Q sends value to directory, invalidates; directory forwards value to P; P has exclusive access