Multiprocessors &
Thread Level Parallelism

COE 501
Computer Architecture
Prof. Muhamed Mudawar
Computer Engineering Department
King Fahd University of Petroleum and Minerals
Presentation Outline

- Introduction to Multiprocessors
- Challenges of Parallel Programming
- Cache Coherence
- Directory Cache Coherence
- Synchronization
What is a Multiprocessor?

- Collection of processors, memories, and storage devices
  - That communicate and cooperate to solve large problems fast

- Resource allocation
  - How large is this collection?
  - How powerful are the processing units?

- Data access, communication, and synchronization
  - How do the processing units communicate and cooperate?
  - How data is transmitted?
  - What type of interconnection network?

- Performance, Scalability, Availability, and Power Efficiency
  - How does it all translate into performance? How does it scale?
  - Availability in the presence of faults? Performance per watt?
Flynn’s Taxonomy (1966)

- **SISD**: Single instruction stream, single data stream
  - Uniprocessors

- **SIMD**: Single instruction stream, multiple data streams
  - Same instruction is executed on different data
  - Exploits Data-Level Parallelism (DLP)
  - Vector processors, SIMD instructions, and Graphics Processing Units

- **MISD**: Multiple instruction streams, single data stream
  - No commercial implementation

- **MIMD**: Multiple instruction streams, multiple data streams
  - Most general and flexible architecture for parallel applications
  - Exploits Thread-Level Parallelism (TLP) and Data-Level Parallelism
  - Tightly-coupled versus loosely-coupled MIMD
Major Multiprocessor Organizations

- **Symmetric Multiprocessors (SMP)**
  - Main memory is shared and equally accessible by all processors
  - Called also Uniform Memory Access (UMA)
  - Bus based or interconnection network based

- **Distributed Shared Memory (DSM) multiprocessors**
  - Memory is distributed and shared and accessed by all processors
  - Non-uniform memory access (NUMA)
  - Latency varies between local and remote memory access

- **Message-Passing multiprocessors (Clusters)**
  - Memory is distributed, but NOT shared
  - Each processor can access its own local memory
  - Processors communicate by sending and receiving messages
Single-Chip Multiprocessor (Multicores)

- Multiprocessor on a single chip (called multicores)
- Each core is a processor with private caches
- All processors share a common cache on chip
- Uniform access to shared cache and main memory
Memory is **distributed** among all processors

- Interconnection network connects all the (Multicore MP) nodes
**Distributed Memory Architectures**

- **Distributed Shared Memory (tightly coupled)**
  - Distributed memories are **shared** among all processors
  - Processors can access local and remote memories
  - Remote memory access over interconnection network
  - Non-uniform memory access (NUMA)

- **Message Passing (loosely coupled)**
  - Distributed memories are **NOT shared**
  - Processors cannot access remote memories
  - Multiple private physical address spaces
  - Easier to build and scale than distributed shared memory
  - Message passing communication over network
Multiprocessor Communication Models

- **Shared Memory Architectures**
  - One global physical address space
  - Distributed among all physical memories
  - Any processor can read/write any physical memory
  - Processors communicate using load and store instructions
  - Non-Uniform Memory Access (NUMA) for large-scale multiprocessors

- **Message Passing Architectures (Clusters)**
  - Separate physical address spaces for nodes
  - A compute node consists of one (or few) multicore chips and memory
  - A node cannot directly access the physical memory of another node
  - Nodes communicate via sending and receiving messages
  - Nodes are interconnected via a high-speed network
Next . . .

- Introduction to Multiprocessors
- Challenges of Parallel Programming
- Cache Coherence
- Directory Cache Coherence
- Synchronization
Levels of Parallelism

- Process-level parallelism
  - Processes are scheduled to run in parallel on multiple processors
  - Each process runs in a separate address space

- Thread-level parallelism
  - A running program (or process) creates multiple threads
  - Threads run within the same address space of a process
  - Multiple program counters (MIMD style) and multiple register files
  - Targeted for shared-memory multiprocessors

- Data-level parallelism within a single thread
  - Wide registers store multiple data values (elements of an array)
  - SIMD/Vector instruction executes same operation on multiple data values

- Instruction-level parallelism within a single thread
Parallel Programming Models

- Parallel Task is the unit of parallel computation
- Two major parallel programming models

1. Shared Memory
   - Popular for small machines consisting of at most hundreds of cores
   - Parallel tasks are executed as separate threads on different cores
   - Threads communicate using load and store instructions to shared memory
   - Threads must synchronize explicitly to control their execution order

2. Message Passing
   - Popular for large machines consisting of hundreds of thousands of cores
   - Parallel tasks cannot share memory, each task has its own memory
   - Parallel tasks communicate explicitly using send and receive messages
   - Synchronization is done implicitly using send and receive
Speedup Challenge: Amdahl’s Law

- $F =$ Fraction of the original execution time which is parallelizable
- $P =$ Number of Processors
- $(1 - F) =$ Fraction of the execution time that cannot be parallelized

\[
\text{Speedup} = \frac{1}{\frac{F}{P} + (1 - F)} \quad \text{Speedup} \to \infty = \frac{1}{(1 - F)}
\]

- Parallelism has an overhead
  - Thread communication, synchronization, and load balancing
  - Overhead increases with the number of processors $P$
  - A large group of processors cannot be interconnected with short distances
Amdahl’s Law Sublinear Speedup

\[ \text{Speedup} = \frac{1}{\frac{F}{P} + (1 - F)} \]

- **Superlinear Speedup**
- **Sublinear Speedup**

- \( F = 99\% \)
- \( F = 95\% \)
- \( F = 90\% \)
Amdahl’s Law: Example 1

- Want to achieve 50× speedup on 100 processors

  What fraction of the original execution time can be sequential?

- Solution

\[ F_{\text{parallel}} = \text{Fraction of the execution time, which is parallelizable} \]

\[ F_{\text{sequential}} = (1 - F_{\text{parallel}}) = \text{Fraction of time, which is sequential} \]

\[
\text{Speedup} = \frac{1}{\frac{F_{\text{parallel}}}{100} + (1 - F_{\text{parallel}})} = 50
\]

Solving: \( F_{\text{parallel}} = \frac{98}{99} \approx 0.99 \) and \( F_{\text{sequential}} = 0.01 \)

- Sequential time is at most 1% of original execution time
Amdahl’s Law: Example 2

- Want to achieve 80× speedup on 100 processors

  If 95% of time, we can use 100 processors, how much of the remaining 5% of time we must employ 50 processors, and how much of the remaining time can be sequential?

- Solution

  \[
  \text{Speedup} = \frac{1}{\frac{F_{100}}{100} + \frac{F_{50}}{50} + (1 - F_{100} - F_{50})} = 80
  \]

  If \( F_{100} = 0.95 \) then \( 0.76 + 1.6 \times F_{50} + 4 - 80 \times F_{50} = 1 \)

  Solving: \( F_{50} = \frac{3.76}{78.4} = 0.04796 = 4.796\% \), and

  \[
  F_{\text{sequential}} = 1 - F_{100} - F_{50} = 0.00204 = 0.204\%
  \]
Impact of Inefficient Communication on Speedup

\[ \text{Speedup} = \frac{1}{\frac{0.99}{P} + 0.01} \]

\[ \text{Speedup} = \frac{1}{\frac{0.99}{P} + 0.01 + \frac{P}{10^4}} \]

\[ \text{Speedup} = \frac{1}{\frac{0.99}{P} + 0.01 + \frac{P^2}{10^5}} \]
Strong versus Weak Scaling

- **Strong Scaling**
  - When speedup can be scaled for a fixed-size problem without increasing the data size.

- **Weak Scaling**
  - When speedup cannot be scaled, except by increasing the data size of a given problem.
  - Data size is increased according to the number of processors.

- **Strong scaling is harder than weak scaling**
  - Achieving higher speedup on a multiprocessor while keeping the data size fixed is harder than increasing the input data size according to the number of processors.
Next . . .

- Introduction to Multiprocessors
- Challenges of Parallel Programming
- Cache Coherence
- Directory Cache Coherence
- Synchronization
Caches in a Single-Chip Multiprocessor

- Private Caches are used inside processor cores
- Reduce average latency
  - Data is closer to processor
- Reduce traffic to memory
  - When data is cached
- Caching shared data
  - Shared data is replicated in multiple private caches
  - This requires maintaining copies of the shared data coherent

⇒ Cache coherence problem
Cache Coherence Problem

- Private processor caches create the coherence problem
  - Copies of a shared variable can be present in multiple data caches

- Updating copy of the shared data in one private data cache only
  - Other processors do not see the update!
  - Processors may read different data values through their caches

- Unacceptable to programming and is frequent!

<table>
<thead>
<tr>
<th>Event</th>
<th>Memory variable X</th>
<th>Processor A Data Cache</th>
<th>Processor B Data Cache</th>
<th>Processor C Data Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor A reads X</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor B reads X</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Processor A stores X = 7</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Processor C reads X</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Caches are supposed to be transparent

What would happen if there were no caches?

- All reads and writes would go to the main memory
- Reading a location X should return the last value written to X

What does last value written mean in a multiprocessor?

- All operations on a particular location X would be serialized
- All processors would see the same access order to location X
Formal Definition of Memory Coherence

A memory system is coherent if it satisfies two properties:

1. Write Propagation

   Writes by a processor become visible to other processors

   All reads by any processor to location X must return the most recently written value to location X, if the read and last write are sufficiently separated in time.

2. Write Serialization

   Writes to the same location X are serialized. Two writes to the same location X by any two processors are seen in the same order by all processors.
What to do about Cache Coherence?

- Organize the memory hierarchy to make it go away
  - Remove private caches and use one cache shared by all processors
  - No private caches ➞ No replication of shared data
  - A switch (interconnection network) is needed to access shared cache
  - Increases the access latency of the shared cache

- Mark shared data pages as uncacheable
  - Shared data pages are not cached (must access memory)
  - Private data is cached only
  - We lose performance

- Detect and take actions to eliminate the problem
  - Can be addressed as a basic cache design issue
Hardware Cache Coherence Solutions

- Coherent Caches should provide
  - **Migration**: movement of shared data between processors
  - **Replication**: multiple copies of shared data (simultaneous reading)

- Cache Coherence Protocol
  - Tracking the sharing (replication) of data blocks in private caches

- Snooping Cache
  - Works well with small bus-based multiprocessors
  - Each cache monitors bus to track sharing status of each block

- Directory Based Schemes
  - Keep track of what is being shared in one place, called directory
  - Centralized memory ➔ Centralized directory
  - Distributed shared memory ➔ Distributed directory
Snooping Cache Coherence Protocols

- Cache controller **snoops** all transactions on the shared bus
- Transaction is **relevant** if address matches tag in the cache
- Take action for coherence
  - Invalidate
  - Update
  - Supply data

- Write Invalidate protocol
  - Must invalidate shared copies
- Write Update protocol
  - Must update shared copies
Snooping Cache Coherence Protocols

- Transactions over a shared bus (broadcast medium)
- Cache controller updates the state of blocks in a cache
- Bus transactions
  - Three phases: arbitration, command/address, data transfer
  - One cache issues command/address, all caches observe addresses
- Cache controller receives inputs from two sides
  - Requests from processor (load/store)
  - Bus requests from snooper
- Cache controller takes action
  - Updates state of blocks
  - Responds with data
  - Generates new bus transactions

![Diagram of cache coherence](image-url)

**Processor Requests (Load / Store)**

<table>
<thead>
<tr>
<th>State</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Snoop Bus Transactions**
 MSI Snoopy Cache Coherence Protocol

- Three States for write-back data cache
  1. **Modified**: only this cache has a **modified copy** of this block
  2. **Shared**: block is **read-only** and can be replicated in other caches
  3. **Invalid**: block is invalid

- Four Bus Transactions
  1. **Read Miss**: Service a read miss on bus
  2. **Write Miss**: Service a write miss on bus (obtain exclusive copy)
  3. **Invalidate**: Invalidate copies of this block in other caches
  4. **Write Back**: Write back a modified block on replacement

- On Write, invalidate all other copies
  - Write cannot complete until **invalidate transaction** appears on bus
  - Write serialization: transactions are serialized on bus
**MSI Snoopy Cache Coherence Protocol**

- **Three Cache States:** M (Modified), S (Shared), I (Invalid)
  - Only one cache can have block in Modified state
## MSI Snoopy Cache Coherence Protocol

<table>
<thead>
<tr>
<th>Request</th>
<th>Source</th>
<th>State</th>
<th>Action and Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Hit</td>
<td>Processor</td>
<td>Shared or Modified</td>
<td>Normal Hit: Read data in local data cache (no transaction)</td>
</tr>
<tr>
<td>Read Miss</td>
<td>Processor</td>
<td>Invalid</td>
<td>Normal Miss: <strong>Read miss on bus</strong>, Wait for data, then change state to <strong>Shared</strong></td>
</tr>
<tr>
<td>Read Miss</td>
<td>Processor</td>
<td>Shared</td>
<td>Replace block: Place <strong>Read miss on bus</strong>, Wait for data block, keep <strong>Shared</strong> state</td>
</tr>
<tr>
<td>Read Miss</td>
<td>Processor</td>
<td>Modified</td>
<td>Replace block: Place <strong>Write-Back</strong> block, Place <strong>Read miss on bus</strong>, Wait for data block, then change state to <strong>Shared</strong></td>
</tr>
<tr>
<td>Write Hit</td>
<td>Processor</td>
<td>Modified</td>
<td>Normal Hit: Write data in local data cache (no transaction)</td>
</tr>
<tr>
<td>Write Hit</td>
<td>Processor</td>
<td>Shared</td>
<td>Coherence: Place <strong>Invalidate on bus</strong> (no data), then change state to <strong>Modified</strong></td>
</tr>
<tr>
<td>Write Miss</td>
<td>Processor</td>
<td>Invalid</td>
<td>Normal Miss: Place <strong>Write miss on bus</strong>, wait for data, change state to <strong>Modified</strong></td>
</tr>
<tr>
<td>Write Miss</td>
<td>Processor</td>
<td>Shared</td>
<td>Replace block: Place <strong>Write miss on bus</strong>, wait for data, change state <strong>Modified</strong></td>
</tr>
<tr>
<td>Write Miss</td>
<td>Processor</td>
<td>Modified</td>
<td>Replace block: <strong>Write-Back</strong> block, Place <strong>Write miss on bus</strong>, wait for data block</td>
</tr>
<tr>
<td>Read Miss</td>
<td>Bus</td>
<td>Shared</td>
<td>No action: <strong>Serve read miss</strong> from shared cache or memory</td>
</tr>
<tr>
<td>Read Miss</td>
<td>Bus</td>
<td>Modified</td>
<td>Coherence: <strong>Write-Back &amp; Serve read miss</strong>, then change state to <strong>Shared</strong></td>
</tr>
<tr>
<td>Invalidate</td>
<td>Bus</td>
<td>Shared</td>
<td>Coherence: <strong>Invalidate shared block</strong> in other caches (change state to <strong>Invalid</strong>)</td>
</tr>
<tr>
<td>Write Miss</td>
<td>Bus</td>
<td>Shared</td>
<td>Coherence: <strong>Invalidate shared block</strong> in other caches, <strong>Serve write miss</strong></td>
</tr>
<tr>
<td>Write Miss</td>
<td>Bus</td>
<td>Modified</td>
<td>Coherence: <strong>Serve write miss</strong> (cache-to-cache transfer) and <strong>Invalidate block</strong></td>
</tr>
</tbody>
</table>
### Example on MSI Cache Coherence

<table>
<thead>
<tr>
<th>Request</th>
<th>Processor P1</th>
<th>Processor P2</th>
<th>Bus</th>
<th>Shared Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>State</td>
<td>Addr</td>
<td>Value</td>
<td>State</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>S</td>
<td>A1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>S</td>
<td>A1</td>
<td>15</td>
<td>S</td>
</tr>
<tr>
<td>P2: Write 10 to A1</td>
<td>I</td>
<td>A1</td>
<td>15</td>
<td>M</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>S</td>
<td>A1</td>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>P1: Write 20 to A1</td>
<td>M</td>
<td>A1</td>
<td>20</td>
<td>I</td>
</tr>
<tr>
<td>P2: Write 35 to A1</td>
<td>I</td>
<td>A1</td>
<td>20</td>
<td>M</td>
</tr>
</tbody>
</table>

- **S**: Shared
- **I**: Invalidated
- **M**: Modified
- **Rd Miss**: Read Miss
- **Wr Back**: Write Back
- **Transfer**: Transfer
Coherence Misses

- Caused by writes to a shared block
  - Shared block is replicated in multiple data caches
  - One processor writes shared block → invalidates copies
  - Another processor reads shared block → coherence miss

- True sharing misses
  - Writing/Reading **same variable** by different processors
  - Communication of shared data through cache coherence

- False sharing misses
  - Writing/Reading **different variables in same block**
  - Invalidation mechanism invalidates the entire block
  - Causes cache miss even though word was not modified
Example of True & False Sharing Misses

Variables X and Y belong to the same cache block

Initially, P1 and P2 read shared variable X

Block (X, Y) is in the shared state in P1 and P2

<table>
<thead>
<tr>
<th>Request</th>
<th>P1 Cache State</th>
<th>P2 Cache State</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write X</td>
<td>Shared (X, Y)</td>
<td>Shared (X, Y)</td>
<td>True Sharing Miss (P2 read X)</td>
</tr>
<tr>
<td></td>
<td>Modified (X, Y)</td>
<td>Invalid (X, Y)</td>
<td>P1 invalidates block (X, Y) in P2</td>
</tr>
<tr>
<td>P2: Read Y</td>
<td>Modified (X, Y)</td>
<td>Invalid (X, Y)</td>
<td>False Sharing Miss (P1 did not write Y)</td>
</tr>
<tr>
<td></td>
<td>Shared (X, Y)</td>
<td>Shared (X, Y)</td>
<td>Write-Back &amp; Copy block from P1 to P2</td>
</tr>
<tr>
<td>P1: Write X</td>
<td>Shared (X, Y)</td>
<td>Shared (X, Y)</td>
<td>False Sharing Miss (P2 did not read X)</td>
</tr>
<tr>
<td></td>
<td>Modified (X, Y)</td>
<td>Invalid (X, Y)</td>
<td>P1 invalidates block (X, Y) in P2</td>
</tr>
<tr>
<td>P2: Write Y</td>
<td>Modified (X, Y)</td>
<td>Invalid (X, Y)</td>
<td>False Sharing Miss (P1 did not read Y)</td>
</tr>
<tr>
<td></td>
<td>Invalid (X, Y)</td>
<td>Modified (X, Y)</td>
<td>Write-Back &amp; Copy block from P1 to P2</td>
</tr>
<tr>
<td>P1: Read Y</td>
<td>Invalid (X, Y)</td>
<td>Modified (X, Y)</td>
<td>True Sharing Miss (P2 modified Y)</td>
</tr>
<tr>
<td></td>
<td>Shared (X, Y)</td>
<td>Shared (X, Y)</td>
<td>Write-Back &amp; Copy block from P2 to P1</td>
</tr>
</tbody>
</table>
Alternative Cache Coherence Protocols

- **MESI** Cache Coherence Protocol: Four States
  1. **Modified**: only this cache has a *modified copy* of this block
  2. **Exclusive**: only this cache has a *clean copy* of this block
  3. **Shared**: block may be replicated in more than one cache (*read-only*)
  4. **Invalid**: block is invalid

Exclusive State: prevents invalidate on a write hit (no bus transaction)

- **MOESI** Cache Coherence Protocol: Five States
  1. **Modified**: only this cache has a *modified copy* of this block
  2. **Owned**: this cache is owner, other caches can share block (*read-only*)
  3. **Exclusive**: only this cache has a *clean copy* of this block
  4. **Shared**: block may be replicated in more than one cache (*read-only*)
  5. **Invalid**: block is invalid

Owner must supply data to others on a miss: *cache-to-cache* transfer
Implementing Snooping Cache Coherence

- Bus operation is **NOT atomic** *(cannot be done in 1 cycle)*
  - **Invalidate** bus operation takes multiple cycles
  - **Write miss** bus operation is also not atomic

- One solution: processor that sends invalidate **holds the bus**
  - Until all processors receive the invalidate
  - A single wire signals when all invalidates are completed
  - Then processor that initiated bus invalidate releases the bus

- In a system with multiple buses or network: **races** can happen
  - Two processors want to write to the same block at the same time
  - Must serialize the writes to the same block (**strictly ordered**)
  - Must receive acknowledgement for invalidates before modifying block

- Data for a read or write miss: shared cache or local cache?
  - **Cache-to-Cache** transfer: Data block transfer between two local caches
Next . . .

- Introduction to Multiprocessors
- Challenges of Parallel Programming
- Cache Coherence
- Directory Cache Coherence
- Synchronization
Limitations of Snooping Protocols

- Buses have limitations for scalability
  - Limited number of processor cores that can be attached to a bus
  - Contention on the use of the shared bus
  - Snooping bandwidth is a bottleneck for large number of processor cores

- On-Chip interconnection network ➔ Parallel communication
  - Multiple processor cores can access shared cache banks at the same time
  - Allows chip multiprocessor to scale beyond few processor cores

- Snooping is difficult on network other than bus or ring
  - Must broadcast coherence traffic to all processors, which is inefficient

- How to enforce cache coherence without broadcast?
  - Have a directory that records the state of each cached block
  - Directory entry specifies which private caches have copies of the block
Directory in a Chip Multiprocessor

- Directory in outermost cache (shared by all processor cores)
  - Directory keeps track of copies of each block in local caches

- Outermost cache is split into multiple banks (parallel access)
  - Number of cache banks can vary (not related to number of cores)
Directory in the Shared Cache

- Shared Cache is **inclusive** with respect to all local caches
  - Shared cache contains a superset of the blocks in local caches
  - Example: Intel Core i7

- Directory is implemented in the shared cache
  - Each block in the shared cache is augmented with presence bits
  - If \( k \) processors then \( k \) presence bits + state per block in shared cache
  - Presence bits indicate which cores have a copy of the cache block
  - Each block has **state** information in private and shared cache
  - State = M (Modified), S (Shared), or I (Invalid) in local cache
Terminology

❖ **Local (or Private) Cache**
  ⊳ Where a processor request originates

❖ **Home Directory**
  ⊳ Where information about a cache block is kept
  ⊳ Directory uses presence bits and state to track cache blocks

❖ **Remote Cache**
  ⊳ Has a copy of the cache block

❖ **Cache Coherence ensures Single-Writer, Multiple-Readers**
  ⊳ If a block is **modified** in a local cache then **one valid copy** can exist
    ▪ Shared Cache and memory are not updated

❖ **No bus and don’t want to broadcast to all processor cores**
  ⊳ All messages have explicit responses
States for Local and Shared Cache

- Three states for a **local** (private) cache block:
  1. **Modified**: only this cache has a *modified copy* of this block
  2. **Shared**: block may be replicated in more than one cache (*read-only*)
  3. **Invalid**: block is invalid, not present in this local cache

- Four states for a **shared** cache block (directory):
  1. **Modified**: only one local cache is the owner of this block
     - One local cache (one presence bit) has a *modified copy* of this block
  2. **Owned**: shared cache is the owner of the modified block
     - Modified block was written-back to shared cache, but not to memory
     - A block in the owned state can be shared by multiple local caches
  3. **Shared**: block may be replicated in more than one cache (*read-only*)
  4. **Invalid**: block is invalid in the shared cache and in any local cache
Read Miss by Processor P

- Processor P sends Read Miss message to home directory
- Home Directory: block is Shared or Owned
  - Directory sends data reply message to P, and sets presence bit of P
  - Local cache of processor P changes state of received block to shared
- Home Directory: block is Modified
  - Directory sends Fetch message to remote cache that modified block
  - Remote cache sends Write-Back message to directory (shared cache)
  - Remote cache changes state of block to shared
  - Directory changes state of shared block to owned
  - Directory sends data reply message to P, and sets presence bit of P
  - Local cache of processor P changes state of received block to shared
- Home Directory: block is Invalid ➔ get block from memory
Read Miss to a Block in Modified State

1. Read Miss by P
   Requesting block A

2. Fetch Block A
   Requested by P

3. Write-Back Block A
   Requested by P

4. Data block A
   reply message to P

Total of 4 messages

Processor Q writes-back Block A to shared cache

Block A is shared by P and Q
It is owned by shared cache
Write Miss Message by P to Directory

- Home Directory: block is **Modified**
  - Directory sends **Fetch-Invalidate** message to remote cache of Q
  - Remote cache of processor Q sends **data reply** message directly to P
  - Remote cache changes state of block to **invalid**
  - Local cache of P changes the state of received block to **modified**
  - Directory clears presence bit of Q and sets presence bit of P

- Home Directory: block is **Shared** or **Owned**
  - Directory sends **invalidate** messages to all sharers (presence bits)
  - Directory receives **acknowledge** message and clears presence bits
  - Directory sends **data reply** message to P, sets presence bit of P
  - Local cache of P and directory change state of the block to **modified**

- Home Directory: **Invalid** → get block from memory
Write Miss to a Block in Modified State

1. Write Miss by P
   Requesting block A

2. Fetch-Invalidate
   Block A Requested by P

3. Data block A reply message to P

Total of 3 messages
Processor Q sends data reply to P
Q invalidates its copy of block A
Block A is then modified by P
Write Miss to a Block with Sharers

Directory sends **invalidate** messages to all sharers
Directory receives **acknowledge** message for all invalidates and clears their presence bits
Directory sends data reply message, and changes state of block A to modified
Invalidating a Block with Sharers

Directory sends **invalidate** messages to all sharers

Directory receives **acknowledge** message for all invalidates and clears their presence bits

Directory sends **acknowledge** message to P, and changes state of block A to **modified**
# Directory Protocol Messages

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Source</th>
<th>Destination</th>
<th>Message Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Miss</td>
<td>Local Cache</td>
<td>Home Directory</td>
<td>Processor P has a read miss at address A Request data and make P a read sharer</td>
</tr>
<tr>
<td>Write Miss</td>
<td>Local Cache</td>
<td>Home Directory</td>
<td>Processor P has a write miss at address A Request data and make P the exclusive owner</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Local Cache</td>
<td>Home Directory</td>
<td>Processor P wants to invalidate all copies of the same block at address A in all remote caches</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Home Directory</td>
<td>Remote Caches</td>
<td>Directory sends invalidate message to all remote caches to invalidate shared block at address A</td>
</tr>
<tr>
<td>Acknowledge</td>
<td>Remote Cache</td>
<td>Home Directory</td>
<td>Remote cache sends an acknowledgement message back to home directory after invalidating last shared block A</td>
</tr>
<tr>
<td>Acknowledge</td>
<td>Home Directory</td>
<td>Local Cache</td>
<td>Directory sends acknowledgement message back to local cache of P after invalidating all shared copies of block A</td>
</tr>
<tr>
<td>Fetch</td>
<td>Home Directory</td>
<td>Remote Cache</td>
<td>Directory sends a fetch message to a remote cache to fetch block A and to change its state to Shared</td>
</tr>
<tr>
<td>Fetch &amp; Invalidate</td>
<td>Home Directory</td>
<td>Remote Cache</td>
<td>Directory sends message to a remote cache to fetch block A and to change its state to Invalid</td>
</tr>
<tr>
<td>Data Block Reply</td>
<td>Directory or Cache</td>
<td>Local Cache</td>
<td>Directory or remote cache sends data block reply message to local cache of processor P that requested data block A</td>
</tr>
<tr>
<td>Data Block Write Back</td>
<td>Remote Cache</td>
<td>Home Directory</td>
<td>Remote Cache sends a write-back message to home directory containing data block A</td>
</tr>
</tbody>
</table>
Three states for a cache block in a local (private) cache

Similar to snooping coherence protocol

Requests by processor

- Black arrows

Requests by directory

- Red arrows

**MSI State Diagram for a Local Cache**
MOSI State Diagram for Directory

Invalid
- Write Miss request by P
- Get block from memory
- Data reply to P, Set presence[P] = 1

Shared (read only)
- Read Miss request by P
- Data reply to P, Set presence[P] = 1

Owned by shared cache
- Write Miss or Invalidate request by P
- Invalidate all sharers, clear presence bits
- Data reply to P, Set presence[P] = 1

Modified (Processor P)
- Read Miss request by P
- Get block from memory
- Data reply to P
- Set presence[P] = 1

Read Miss request by Q
- Fetch P
- Data reply to Q
- Write-Back by P
- Presence[P] = 0

Fetch-Invalidate P, Data-reply to Q, Presence[P] = 0, Presence[Q] = 1

Read Miss request by Q
- Fetch P
- Data reply to Q
- Presence[Q] = 1
Next . . .

- Introduction to Multiprocessors
- Challenges of Parallel Programming
- Cache Coherence
- Directory Cache Coherence
- Synchronization
Synchronization

- Cache coherence allows parallel threads to communicate
- However, coherence **does not synchronize** parallel threads
- Synchronization is required to control the execution of threads
- Three types of synchronization are widely used:
  1. **Lock** synchronization
  2. **Event** synchronization
  3. **Barrier** synchronization

- Synchronization **ensures the correctness** of parallel execution
- However, synchronization can be a **performance bottleneck**
  - Reduces the speedup and performance of parallel threads
Lock Synchronization

- Protects a **critical section** accessed by parallel threads

- A critical section is a sequence of instructions that
  - Read – Modify – Write shared data in memory
  - Only **one thread** can be in the critical section at a time

- Two synchronization operations are defined:
  1. Lock(X), just before entering critical section
  2. Unlock(X), just before leaving critical section

- Only **one thread** is allowed to lock variable X at a time

- Other threads **must wait** until the variable X is unlocked

- Access to the critical section is **serialized**
Critical Section

- Critical sections with lock/unlock make threads wait
- Using one lock variable X to lock an array increases contention
- Using many lock variables (fine-grain locking) reduces contention
- Atomic (read-modify-write) instructions can help reducing locks

Each thread:

**Compute1**
lock(X)  
Critical Section
unlock(X)  
Compute2

**Compute1**
lock(X)  
Critical Section  
unlock(X)  
Compute2

**Compute1**
lock(X)  
Critical Section  
unlock(X)  
Compute2

Time
Point-to-Point Event Synchronization

- One thread (**producer**) computes and writes data in memory
- One (or more) thread (**consumer**) reads data in memory
- Consumer thread **must wait** until the data is written in memory
- Two operations are defined:
  1. **Wait(X)**  Causes a thread to wait until variable X is set
  2. **Set(X)**  Set X to true and releases waiting threads (if any)

![Diagram showing producer and consumer threads with Wait and Set operations]
Barrier Synchronization

- Threads **must wait** until **All threads** reach the barrier
- **Last thread** arriving the barrier **releases all waiting** threads
- Total execution time depends on the **slowest thread**
- Threads must be **balanced** to avoid losing performance

Each thread:

```plaintext
loop {
  Compute1
}

Barrier(X,4)

loop {
  Compute2
}
```
Instructions for Synchronization

- Synchronization functions are implemented in a library
- The architecture provides special instructions for synchronization
- **Swap instruction:** \texttt{swap rb, (ra)}
  - Swap a register \texttt{rb} with a memory variable at address \texttt{(ra)}
  - Does a load and store in one \textbf{atomic instruction}
- **Load-Linked (LL) and Store Conditional (SC) instructions:**
  - \texttt{LL rd = (ra)} \quad \text{Load and save address in link register: } \texttt{link = ra}
  - \texttt{SC rd = (ra), rb} \quad \text{If } \texttt{(link==ra)} \{\text{store (ra)=rb; rd=1}\} \text{ else rd=0}
  - If \texttt{SC} succeeds, then the \texttt{LL-SC} sequence has \textbf{executed atomically}
  - If \texttt{SC} fails and returns \textbf{zero} in \texttt{rd} then \texttt{LL-SC} sequence \textbf{must be repeated}
  - The \texttt{LL-SC} sequence can implement many synchronization operations
Implementing Lock and Unlock

- Lock can be implemented with few instructions
- If the lock is acquired then spin (busy-wait)

```assembly
lock: // Address parameter in r1
    ld r2 = (r1) // Load lock value
    bnez r2, lock // Spin if the lock is acquired
    set r2 = 1 // Set lock value
    swap r2, (r1) // Swap r2 with lock variable (r1)
    bnez r2, lock // Make sure it was not locked
    ret // Return to caller
```

- Unlock stores zero to release the lock

```assembly
unlock: // Address parameter in r1
    sd (r1) = r0 // store zero to release the lock
    ret // Return to caller
```
Implementing Atomic Operations

- The LL and SC instructions can implement many atomic operations

- Implementing **Fetch-and-Add** using LL and SC:

  ```
  // Two parameters: r1 = address, r2 = added value
  FetchAdd:
  LL  r3 = (r1)  // Load Linked
  ADD r3 = r3, r2  // Add to r2
  SC  r4 = (r1), r3  // Store conditional
  BEQZ r4, FetchAdd  // Retry if SC failed
  RET
  ```

- If SC succeeds that LL-ADD-SC sequence is executed atomically
- If SC fails that the LL-ADD-SC sequence must be repeated
Concluding Remarks

- Goal: higher performance using multiple processors

- Difficulties
  - Developing parallel software
  - Devising appropriate architectures

- Many reasons for optimism
  - Changing software and application environment
  - Chip-level multiprocessors
    - Lower latency, higher bandwidth interconnect

- An ongoing challenge for computer architects