Uniform Memory Access (UMA) architecture (aka SMP)

Parallel and Distributed Computing

Department of Computer Science and Engineering (DEI)
Instituto Superior Técnico

September 21, 2011
Outline

- Shared vs Distributed Memory architectures
- Memory coherence and consistency in UMA architectures
  - Snooping protocols
  - Directory-based protocols
- Synchronization
MIMD Architectures

Memory organizations for multiple instruction, multiple data (MIMD) architectures:

- shared memory
- distributed memory

![Diagram of MIMD Architectures]

- \( P_1 \) with input \( I_1 \) and output \( Dout_1 \)
- \( P_2 \) with input \( I_2 \) and output \( Dout_2 \)
- ... (for \( P_n \)) with input \( I_n \) and output \( Dout_n \)
Uniform Memory Access (UMA) architecture

also known as

Symmetric Shared-Memory Multiprocessors (SMP)
Distributed-Memory Systems

Non-Uniform Memory Access (NUMA) architecture or Multicomputers
Contention in memory access?

Diagram:

- P
- P
- P
- P

Main Memory

I / O
Contestation in memory access?

P

Cache

Main Memory

P

Cache

I / O

P

Cache

P

Cache
Shared-Memory Systems

Contention in memory access?

Caches write-back or write-through?

Typically write-back: to reduce number of accesses to main memory.
Shared-Memory Systems

Contestion in memory access?

Caches write-back or write-through?
Typically write-back: to reduce number of accesses to main memory.
Naturally, in distributed-memory systems caches are present and are local to each processor:
Shared vs Distributed Memory Systems

- data-sharing much easier in shared-memory architectures
  - data is divided in private data and shared data
  - in distributed-memory systems, data needs to migrate using messages
    \[ \implies \text{communication overhead higher in distributed-memory architectures} \]
Shared vs Distributed Memory Systems

- data-sharing much easier in shared-memory architectures
  - data is divided in private data and shared data
  - in distributed-memory systems, data needs to migrate using messages
    ⇒ communication overhead higher in distributed-memory architectures

- tasks simpler in shared-memory architectures since they share context
  - threads in shared-memory architectures
  - processes in distributed-memory architectures
    ⇒ task startup / finalize overhead is lower in shared-memory architectures
    ⇒ allow for a finer-grain parallelism
Shared vs Distributed Memory Systems

- data-sharing much easier in shared-memory architectures
  - data is divided in private data and shared data
  - in distributed-memory systems, data needs to migrate using messages
    ⇒ communication overhead higher in distributed-memory architectures

- tasks simpler in shared-memory architectures since they share context
  - threads in shared-memory architectures
  - processes in distributed-memory architectures
    ⇒ task startup / finalize overhead is lower in shared-memory architectures
    ⇒ allow for a finer-grain parallelism

- uniform data-access time in shared-memory architectures
data-sharing much easier in shared-memory architectures
- data is divided in private data and shared data
- in distributed-memory systems, data needs to migrate using messages
  ⇒ communication overhead higher in distributed-memory architectures

tasks simpler in shared-memory architectures since they share context
- threads in shared-memory architectures
- processes in distributed-memory architectures
  ⇒ task startup / finalize overhead is lower in shared-memory architectures
  ⇒ allow for a finer-grain parallelism

uniform data-access time in shared-memory architectures

easier programming of shared-memory architectures
Shared vs Distributed Memory Systems

However:
However:

- contention to main memory limits the scalability of shared-memory architectures
However:

- contention to main memory limits the scalability of shared-memory architectures

- distributed-memory is an effective way to increase memory bandwidth
Shared vs Distributed Memory Systems

However:

- contention to main memory limits the scalability of shared-memory architectures
- distributed-memory is an effective way to increase memory bandwidth
- communication is explicit, meaning it is easier to understand and control
Shared vs Distributed Memory Systems

However:

- contention to main memory limits the scalability of shared-memory architectures

- distributed-memory is an effective way to increase memory bandwidth

- communication is explicit, meaning it is easier to understand and control

- hardware can be simpler
Coherence in Shared-Memory Systems

In parallel systems, data migration and replication are normal and desirable events.

However, since shared data may be present in several caches, a new problem arises: cache coherence.
Coherence in Shared-Memory Systems

In parallel systems, data migration and replication are normal and desirable events.

However, since shared data may be present in several caches, a new problem arises: cache coherence.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache uP A</th>
<th>Cache uP B</th>
<th>Memory Address X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>uP A reads $M[X]$</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
In parallel systems, data migration and replication are normal and desirable events.

However, since shared data may be present in several caches, a new problem arises: cache coherence.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache uP A</th>
<th>Cache uP B</th>
<th>Memory Address X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>uP A reads M[X]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>uP B reads M[X]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
In parallel systems, data migration and replication are normal and desirable events.

However, since shared data may be present in several caches, a new problem arises: cache coherence.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache uP A</th>
<th>Cache uP B</th>
<th>Memory Address X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>uP A reads \texttt{M[X]}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>uP B reads \texttt{M[X]}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>uP A 1→\texttt{M[X]}</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Memory Coherence

A memory system is coherent if it always reads the most recent value written to the memory position being read.

**Coherence:** defines what values can be returned in a read operation

**Consistency:** defines when a value that is written to memory is returned in a read operation
A memory system is coherent if it always reads the most recent value written to the memory position being read.

**Coherence:** defines what values can be returned in a read operation

**Consistency:** defines when a value that is written to memory is returned in a read operation

Complementary concepts:

- **Coherence:** access behavior to one memory location
- **Consistency:** interaction between accesses to different locations
A memory system is **coherent** if:

- A read of \( M[X] \) by processor \( P \) that follows a write by \( P \) to \( M[X] \), and with no write to \( M[X] \) by another processor in between, always returns the value written by \( P \).

- Two writes to the same location are serialized; two writes to the same location by any two processors are seen in the same order by all processors.
A memory system is **coherent** if:

- A read of $M[X]$ by processor $P$ that follows a write by $P$ to $M[X]$, and with no write to $M[X]$ by another processor in between, always returns the value written by $P$.

- A read of $M[X]$ by $P_i$ that follows a write of $M[X]$ by $P_j$ always returns the value written by $P_j$ if the read and write are sufficiently separated in time and no other writes to $M[X]$ occur between the two accesses.
A memory system is **coherent** if:

- a read of $M[X]$ by processor $P$ that follows a write by $P$ to $M[X]$, and with no write to $M[X]$ by another processor in between, always returns the value written by $P$.

- a read of $M[X]$ by $P_i$ that follows a write of $M[X]$ by $P_j$ always returns the value written by $P_j$ if the read and write are sufficiently separated in time and no other writes to $M[X]$ occur between the two accesses.

- write to the same location are serialized; two writes to the same location by any two processors are seen in the same order by all processors.
The **consistency model** of a memory system defines when must a processor observe the writes of another processor.

P1: \[ A = 0; \]
\[ \vdots \]
\[ A = 1; \]
\[ \text{if}(B == 0) \]
L1: \[ \vdots \]

P2: \[ B = 0; \]
\[ \vdots \]
\[ B = 1; \]
\[ \text{if}(A == 0) \]
L2: \[ \vdots \]
Memory Consistency

The consistency model of a memory system defines when must a processor observe the writes of another processor.

P1: A = 0;
    
    A = 1;
    if(B == 0)

L1: 

P2: B = 0;
    
    B = 1;
    if(A == 0)

L2: 

Sequential Consistency: memory accesses executed by each processor kept in order, and memory accesses among different processors arbitrarily interleaved
⇒ each write causes a processor to wait until all processors have been notified of the memory change
Cache Coherence Protocols

Cache coherence protocols keep track of the state of the shared blocks of data.

Directory based: the state of each shared block of data is kept in a centralized location, called the directory.

Snooping: each cache has the sharing status of each data block that it stores. All caches monitor (or snoop) the bus to determine if the status of the data blocks should be modified.

- Write invalidate protocol
- Write update protocol
### Snooping: Invalidate Protocols

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
<th>Cache uP A</th>
<th>Cache uP B</th>
<th>Address Memory X</th>
</tr>
</thead>
<tbody>
<tr>
<td>uP</td>
<td>Bus</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>uP A</td>
<td>reads M[X]</td>
<td>Miss X</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Snooping: Invalidate Protocols

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
<th>Cache uP A</th>
<th>Cache uP B</th>
<th>Address Memory X</th>
</tr>
</thead>
<tbody>
<tr>
<td>uP A reads M[X]</td>
<td>Miss X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uP B reads M[X]</td>
<td>Miss X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Action</td>
<td>Action</td>
<td>Cache</td>
<td>Cache</td>
<td>Address</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>uP</td>
<td>Bus</td>
<td>uP A</td>
<td>uP B</td>
<td>Memory X</td>
</tr>
<tr>
<td>uP A reads ( M[X] )</td>
<td>Miss X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uP B reads ( M[X] )</td>
<td>Miss X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uP A ( \rightarrow ) ( M[X] )</td>
<td>Invalidate X</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
### Snooping: Invalidate Protocols

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
<th>Cache uP A</th>
<th>Cache uP B</th>
<th>Address Memory X</th>
</tr>
</thead>
<tbody>
<tr>
<td>uP A reads M[X]</td>
<td>Miss X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>uP B reads M[X]</td>
<td>Miss X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uP A 1→M[X]</td>
<td>Invalidate X</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>uP B reads M[X]</td>
<td>Miss X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Write update protocol is also known as Broadcast Protocol.

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
<th>Cache uP A</th>
<th>Cache uP B</th>
<th>Address Memory X</th>
</tr>
</thead>
<tbody>
<tr>
<td>uP A reads $M[X]$</td>
<td>Miss $X$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bus</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>0</th>
</tr>
</thead>
</table>
Write update protocol is also known as Broadcast Protocol.

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
<th>Cache A</th>
<th>Cache B</th>
<th>Address Memory</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>uP A reads $M[X]$</td>
<td>Miss $X$</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uP B reads $M[X]$</td>
<td>Miss $X$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Write update protocol is also known as Broadcast Protocol.

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
<th>Cache uP A</th>
<th>Cache uP B</th>
<th>Address Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>uP A reads M[X]</td>
<td>Miss X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>uP B reads M[X]</td>
<td>Miss X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uP A 1→M[X]</td>
<td>Broadcast X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Write update protocol is also known as Broadcast Protocol.

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
<th>Cache uP A</th>
<th>Cache uP B</th>
<th>Address Memory X</th>
</tr>
</thead>
<tbody>
<tr>
<td>uP A reads M[X]</td>
<td>Miss X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uP B reads M[X]</td>
<td>Miss X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uP A 1→M[X]</td>
<td>Broadcast X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>uP B reads M[X]</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
multiple writes to the same address cause multiple broadcasts, however only one invalidation
Comparison Invalidate/Update Protocols

- multiple writes to the same address cause multiple broadcasts, however only one invalidation

- each word written in a given cache block causes a broadcast, but only the first word written in that block cause an invalidation
multiple writes to the same address cause multiple broadcasts, however only one invalidation

each word written in a given cache block causes a broadcast, but only the first word written in that block cause an invalidation

the broadcast protocol leads to a smaller delay when reading a word written by another processor
Comparison Invalidate/Update Protocols

- multiple writes to the same address cause multiple broadcasts, however only one invalidation

- each word written in a given cache block causes a broadcast, but only the first word written in that block cause an invalidation

- the broadcast protocol leads to a smaller delay when reading a word written by another processor

⇒ write invalidate protocols are by far the most used because of the lower bandwidth requirements
Directory-based Protocols

The sharing state of each data block is kept in centralized directory.

Operations (same before):
- handle read-misses
- handle writes to shared blocks
- (write-misses are a sequence of these two)
Definition of block states:

**Uncached:** no processor has a copy of the data block

**Shared:** one or more processors have a copy of the data block and the value stored in memory (and in the caches) is up-to-date

**Exclusive:** only one processor has a copy of the data block and it has been modified; value stored in memory is outdated (the processor *owns* the data block)

Directory access can become a bottleneck: distributed directory!
Synchronization

In many situations, synchronization is required between tasks in a parallel application: mutual exclusion areas, synchronization barriers, etc.

Software synchronization is only possible through hardware primitives.

Typically, processors provide read-modify-write assembly instructions that read and modify atomically a memory address:

- non-interruptible
- without relinquishing the memory buses

Examples:

TAS (test-and-set)
*Fetch-and-increment*
Exchange Ri,M[semaphore]
Synchronization Example: Mutual Exclusion

Let \( \text{sem} \) be a semaphore that controls the access to a mutual exclusion memory region.

- \( \text{sem}=0 \) indicates region is free
- \( \text{sem}=1 \) indicates region is occupied

Two processes try simultaneously to have access to the mutual exclusive zone. To this end, they both execute

\[
\begin{align*}
\text{Move} & \quad \text{R1, 1} \\
\text{Exchange} & \quad \text{R1, M[sem]}
\end{align*}
\]

Since the exchange of values between R1 and M[sem] is atomic, only one of the processes will get R1=0.
Spin locks:

```
DADDUI  R1, R0, #1

lock:  EXCH  R1, 0(R2) ; atomic exchange
       BNEZ  R1, lock  ; locked?

       EXCH  R1, 0(R2) ; unlock
```

Active wait:

- expected short unavailability period
- minimizes latency time to access resource
Mutual Exclusion with Spin Locks

Should semaphore be stored in the cache?

Advantages:
- reduces accesses to main memory or interconnection network
- high probability of reuse in near future

Problem... With read-modify-write operations, each test makes a write. If more than one processor waiting for lock, too many invalidate/update messages!

⇒ loop with simple read operation and only attempts the atomic lock when read as free!
Mutual Exclusion with Spin Locks

Should semaphore be stored in the cache?

Advantages:

- reduces accesses to main memory or interconnection network
- high probability of reuse in near future

Problem...
Mutual Exclusion with Spin Locks

Should semaphore be stored in the cache?

Advantages:

- reduces accesses to main memory or interconnection network
- high probability of reuse in near future

Problem...

With read-modify-write operations, each test makes a write. If more than one processor waiting for lock, too many invalidate/update messages!
Mutual Exclusion with Spin Locks

Should semaphore be stored in the cache?

Advantages:

- reduces accesses to main memory or interconnection network
- high probability of reuse in near future

Problem...

With read-modify-write operations, each test makes a write. If more than one processor waiting for lock, too many invalidate/update messages!

⇒ loop with simple read operation and only attempts the atomic lock when read as free!
Mutual Exclusion with Spin Locks

More efficient spin lock:

```
lock:    LD      R1, 0(R2) ; read lock
        BNEZ    R1, lock ; locked?
        DADDUI  R1, R0, #1
        EXCH    R1, 0(R2) ; atomic exchange
        BNEZ    R1, lock ; locked?
        :        
        EXCH    R1, 0(R2) ; unlock
```

Mostly *reads* during active wait.
Review

- Shared vs Distributed Memory architectures
- Memory coherence and consistency in UMA architectures
  - Snooping protocols
  - Directory-based protocols
- Synchronization
Non-Uniform Memory Access (NUMA) architecture

and

Multicomputers