ECE 454
Computer Systems Programming
Performance Implications of Parallel Architectures

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin
Big Picture

- We know that we need parallelization
- But will more parallelization always yield better performance?

We will study the reasons for this behavior.
Topics

- Cache coherence
- Performance of memory operations
- Implications for software design
- Memory consistency
Cache Coherence
Modern Shared Memory Parallel Architectures

- Provide several processing elements (cores or processors)
- Provide shared memory
  - Any processor can directly reference any memory location
  - Communication occurs implicitly through loads and stores
- Cores have private caches to improve performance
Cache Coherence Problem

- With multiple cores, data is cached in multiple locations, so how do you ensure consistency?
Example 1: Coherence Problem

Thread A: Store X=3

Thread B: Load X

X = ?

Shared Memory (X=2)
Example 2: Coherence Problem

Thread A:  
Thread B:  

Load X

Shared Memory (X=2)
Example 2: Coherence Problem

Thread A:
- Processor
- Cache
  - Tag
  - Data

Thread B:
- Load X
- Processor
- Cache
  - Tag
  - Data

Shared Memory (X=2)
Example 2: Coherence Problem

Thread A:

Thread B:

Load X

Shared Memory (X=2)
Example 2: Coherence Problem

Thread A: Store X=3

Thread B:

Shared Memory (X=2)
Example 2: Coherence Problem

Thread A: Store X=3

Thread B:

Inconsistency!

Shared Memory (X=2)
Cache (or Memory) Coherence

- The behavior of the system is equivalent to there being only a single copy of the data except for the performance benefit of the cache. [Gray and Cheriton 83]

- Cache coherence ensures that all processors have a consistent view of a single memory location (e.g., X)
  - All loads and stores to X can be put on a timeline (total order) that respects the program order of loads and stores of each processor
Why Cache Coherence?

- With non-cache coherent machines, e.g., Intel Rack Scale, The Machine from HP, loads and stores are not synchronized
  - Loads may read stale data, i.e., store is not visible to later load
  - Stores are not sequenced, i.e., stores visible in different orders
  - Really complicates the programming model
MSI Coherence Protocol

• Ensuring coherence requires hardware support
  • Called coherence protocol

• Add three (exclusive) states to each cache line (on each core):
  • Invalid – data is not cached
  • Modified – core has written to the cache line
    • Cache line is inconsistent with primary storage
    • Cache line is not shared with other cores
  • Shared – core has read from the cache line
    • Cache line is consistent with primary storage
    • Cache line may be shared with other cores
MSI Coherence Protocol

Thread A:

Processor

Cache

State: Invalid
Tag: -
Data: -

Thread B:

Processor

Cache

State: Invalid
Tag: -
Data: -

Shared Memory (X=2)
MSI Coherence Protocol

Thread A:

Thread B:

Load X

Read

Shared Memory (X=2)
MSI Coherence Protocol

Thread A: Read

Thread B: Load X

Shared Memory (X=2)

Processor
Cache
State: Invalid
Tag: -
Data: -

Processor
Cache
State: Shared
Tag: X
Data: 2

Read
Fill
MSI Coherence Protocol

Thread A: Store X=3

Thread B: Load X

Shared Memory (X=2)

invalidates all other copies
Thread A: Store X=3
Thread B: Load X

Invalidates all other copies
Thread A: Store X=3
Thread B: Load X

Shared Memory (X,2 (stale))
Problem with MSI

- If a core reads a value that is not cached on any core and then writes to the value, then two cache coherence requests are generated:
  - A request to read the value (required)
  - A request to write the value (unnecessary invalidation request sent because the MSI protocol doesn’t know that no one else has a copy)
MESI (aka Illinois) Protocol

- Four (exclusive) states of each cache line:
  - **Invalid** – data is not cached
  - **Modified** – core has written to the cache line
    - Cache line is inconsistent with primary storage
    - Cache line is not shared with other cores
  - **Shared** – core has read from the cache line
    - Cache line is consistent with primary storage
    - Cache line may be shared with other cores
  - **Exclusive**: core has read from the cache line
    - Cache line is consistent with primary storage
    - Cache line is not shared by other cores
    - Write to Exclusive state does not generate invalidation request
MESI Details: Writing

- An attempt to write to a block that is in Invalid state is called a write miss
  - Must cache the block in Exclusive state before writing to it
  - Generates a read-exclusive (or read for ownership) request
    - A read request with intent to write to the memory address
    - If other caches have copy of data, they send it, invalidate their copy
    - Completes when there are no more valid copies
  - Can then perform the write and enter the modified state
    - This step doesn’t require invalidation request
MESI Examples

- Example 1: load on one core followed by load on another core
- Example 2: load on one core followed by store on another core
- Example 3: store on one core followed by load on another core
Example 1: Mesi Coherence

Thread A:

Thread B:

Shared Memory (X=2)
Example 1: MESI Coherence

Thread A: Load X

Thread B:

Shared Memory (X=2)
Example 1: MESSI Coherence

Thread A:
- Load X
- Cache: State=Invalid, Tag=-, Data=-

Thread B:

Processor
Cache

State | Tag | Data
--- | --- | ---
Invalid | - | -

Read

Shared Memory (X=2)
Example 1: MESI Coherence

Thread A:

- Load X
- Cache: State = Excl, Tag = X, Data = 2

Thread B:

- Load X
- Read
- Fill
- Cache: State = Invalid, Tag = - , Data = -

Shared Memory (X=2)
Example 1: MESI Coherence

Thread A:
- Load X
- Tag: X
- State: Excl
- Data: 2

Thread B:
- Load X
- Tag: -
- State: Invalid
- Data: -

Shared Memory (X=2)
Example 1: MESI Coherence

Thread A:
- Load X
- Processor
- Cache
- State: Excl
- Tag: X
- Data: 2

Thread B:
- Load X
- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Shared Memory (X=2)

Read
Example 1: MESI Coherence

Thread A:

- Load X
- Cache
  - State: Share
  - Tag: X
  - Data: 2

Thread B:

- Load X
- Cache
  - State: Share
  - Tag: X
  - Data: 2

Shared Memory (X=2)

- Notify Shared
- Fill
- Read
Example 2: MESI Coherence

Thread A:

Thread B:

Shared Memory (X=2)
Example 2: MESI Coherence

Thread A:

Thread B:

Load X

Read

Shared Memory (X=2)
Example 2: MESI Coherence

Thread A:

Thread B:

Load X

Read

Fill

Shared Memory (X=2)
Example 2: MESI Coherence

Thread A: Store X=3

Thread B: Load X

Read-Exclusive

Shared Memory (X=2)

read-exclusive invalidates all other copies
Example 2: MESI Coherence

Thread A: Store X=3

Thread B: Load X

read-exclusive invalidates all other copies
Example 2: MESI Coherence

Thread A: Store X=3
Thread B: Load X

the state ‘dirty’ implies exclusiveness
Example 3: MESI Coherence

Thread A:  Thread B:

Shared Memory (X=2)
Example 3: MESI Coherence

Thread A: Store X=5

Thread B:

Shared Memory (X=2)
Example 3: Mesi Coherence

Thread A:

```
Thread A: Store X=5
```

Thread B:

```
Thread B: Read-Exclusive
```

Shared Memory (X=2)
Example 3: MEGI Coherence

Thread A: Store X=5

Thread B: Read-Exclusive

Shared Memory (X=out-of-date)
Example 3: MESI Coherence

Thread A: Store X=5

Thread B: Load X

Shared Memory (X, 2 (out of date))
Example 3: Mesi Coherence

Thread A:
Store $X=5$

Thread B:
Load $X$

Shared Memory ($X=$out-of-date)
Example 3: MESI Coherence

Thread A:

Store X=5

Thread B:

Load X

Shared Memory (X=out-of-date)
Example 3: MESI Coherence

Thread A: Store X=5

Thread B: Load X

Shared Memory (X=5)
Example 3: MESI Coherence

Thread A:
Store X=5

Thread B:
Load X

Shared Memory (X=5)
**MESI Permitted States, Transitions**

<table>
<thead>
<tr>
<th>Local Event</th>
<th>Initial State</th>
<th>Local</th>
<th>Message</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>I</td>
<td>I \rightarrow (S, E)</td>
<td>READ</td>
<td>(S, E) \rightarrow S M \rightarrow S + WB</td>
</tr>
<tr>
<td>Read hit</td>
<td>S, E, M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write miss</td>
<td>I</td>
<td>I \rightarrow M</td>
<td>READEX</td>
<td>(S,E) \rightarrow I M \rightarrow I + WB</td>
</tr>
<tr>
<td>Write hit</td>
<td>S</td>
<td>S \rightarrow M</td>
<td>INVALIDATE</td>
<td>S \rightarrow I</td>
</tr>
<tr>
<td></td>
<td>E, M</td>
<td>E \rightarrow M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance of Memory Operations
Local Caches and Memory Latencies

- Cost of accessing memory
  - Best case
    - Data is cached locally: L1 < 10 cycles (remember this)
  - Worst case
    - Data is accessed from DRAM: 136 – 355 cycles (remember this)

<table>
<thead>
<tr>
<th></th>
<th>Opteron</th>
<th>Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>L2</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>LLC</td>
<td>40</td>
<td>44</td>
</tr>
<tr>
<td>RAM</td>
<td>136</td>
<td>355</td>
</tr>
</tbody>
</table>
Interconnect Between Sockets

Cross-sockets communication can be 2-hops

(a) AMD Opteron
(b) Intel Xeon
## Latency of Remote Access: Read (cycles)

<table>
<thead>
<tr>
<th>System</th>
<th>Opteron</th>
<th>Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hops</td>
<td>one hop</td>
</tr>
<tr>
<td></td>
<td>State</td>
<td>same die</td>
</tr>
<tr>
<td>Modified</td>
<td></td>
<td>81</td>
</tr>
<tr>
<td>Exclusive</td>
<td></td>
<td>83</td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td>83</td>
</tr>
</tbody>
</table>

- Local cache line state is invalid
- **State** is the MESI state of a cache line in a remote cache
- **Cross-socket communication is expensive!**
  - Xeon: cross-socket latency is 4-7.5 larger than within socket
  - Opteron: cross-socket latency even larger than RAM
Latency of Remote Access: Write (cycles)

- Local cache line state is invalid
- **State** is the MESI state of a cache line in a remote cache
- **Cross-socket communication is expensive!**
  - Comparable or more expensive than DRAM accesses

<table>
<thead>
<tr>
<th>State</th>
<th>Hops</th>
<th>Opteron</th>
<th>Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>one hop</td>
<td>two hops</td>
</tr>
<tr>
<td>Modified</td>
<td>same die</td>
<td>83</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>one hop</td>
<td>191</td>
<td>273</td>
</tr>
<tr>
<td></td>
<td>two hops</td>
<td>286</td>
<td>296</td>
</tr>
<tr>
<td>Exclusive</td>
<td>same die</td>
<td>83</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>one hop</td>
<td>191</td>
<td>271</td>
</tr>
<tr>
<td></td>
<td>two hops</td>
<td>286</td>
<td>296</td>
</tr>
<tr>
<td>Shared</td>
<td>same die</td>
<td>246</td>
<td></td>
</tr>
<tr>
<td></td>
<td>one hop</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>two hops</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implications for Software Design
False Sharing

Thread A:

Store X=5

Thread B:

Processors

Cache

State | Tag | Data
---|---|---
-,- | -,- | -,-

Read-Exclusive

Shared Memory (X=0, Y=0)

X and Y are on the same cache line
False Sharing

Thread A:

Store X=5

Thread B:

Processor

Cache

State | Tag | Data
---|---|---
Dirty | X,Y | 5,0

Fill

Read-Exclusive

Shared Memory (X,Y=out-of-date)
False Sharing

Thread A:
Store X=5

Thread B:
Store Y=2

Shared Memory (X,Y=out-of-date)
False Sharing

Thread A:

`Store X=5`

Thread B:

`Store Y=2`

Update

Invalidation

`Shared Memory (X=5,Y=0)`
False Sharing

**Thread A:** Store X=5

**Thread B:** Store Y=2

---

**Processor**

- **Cache**
  - **State**
  - **Tag**
  - **Data**

**State**:
- `-,-` (Out-of-Date)

**Tag**:
- `-,-`

**Data**:
- `-,-`

**Processor**

- **Cache**
  - **State**
  - **Tag**
  - **Data**

**State**:
- `X,Y`

**Tag**:
- `5,2`

**Data**:
- `X,Y`

---

**Shared Memory (X,Y=out-of-date)**
False Sharing

Thread A:

Store X=5

Thread B:

Store Y=2

Shared Memory (X,Y=out-of-date)

Processor

Cache

State  Tag  Data

update

Invalidation

fill

Read-Exclusive

Processor

Cache

State  Tag  Data

Dirty  X,Y  5,2
False Sharing

Thread A:
while(1)
Store X=5

Thread B:
while(1)
Store Y=2

Shared Memory (X, Y = out-of-date)

X, Y cache line will ping-pong back & forth
False Sharing Summary

- False sharing
  - Threads on different cores access unrelated objects
  - Objects are located in same cache block
  - Block will ping-pong between caches on different cores

- Avoid false sharing by careful data arrangement
  - Ensure that unrelated elements are mapped to separate blocks
    - E.g., insert padding (unused data) between shared items
  - Partition allocations by different threads, e.g., jemalloc
Implications for Programmers

- Cache coherence is expensive (more than you thought)
  - Avoid unnecessary sharing (e.g., false sharing)

- Crossing processors/sockets is a killer
  - Can be slower than running the same program on single core!
  - Pthreads provides CPU affinity mask
    - Pin cooperative threads on cores within the same die

- Later, we will see other implications of modern architectures on software design

- Next, we look at another peculiarity of modern parallel architectures
Memory Ordering

With thanks to Anton Burtsy, Paul E. McKenney
Coherence versus Consistency

• Recall *cache coherence* ensures that all processors have a consistent view of a single memory location (e.g., X)
  • All loads and stores to X can be put on a timeline (total order) that respects the program order of loads and stores of each processor
  • Defines memory behavior in the presence of processor caches

• **Memory consistency** defines the behavior of reads and writes by a processor to different locations (as observed by other processors)
  • Defines when writes propagate to other processors, what values reads can return (or cannot return), whether caches exist or not
  • Intuitively, reads should return value of last write
    • But how should last be defined?
A system is **sequentially consistent** if the result of any execution is the same as if all the memory operations were executed in some sequential order, and the memory operations of each processor are executed in program order.

This model is intuitive to programmers, but not implemented by real processors, as we see next.
With sequential consistency, can both reads return 0?

Suppose this is possible (proof by contradiction):
- Add edge between ops X and Y to indicate X happens before Y
  - 2 edges for program order
  - 2 edges for memory ordering dependency, why?
  - Happens-before edges form a cycle!
- Would need time warp for both reads to return 0 😊
- But what happens on real processors?
Pros and Cons of Sequential Consistency

Pros: an intuitive model of parallelism 😊
- Each processor executes memory instructions in order
- Memory ops from all processors appear sequentially ordered

Cons: programs run terribly slowly ☹
- Requires each memory operation to complete (results are visible) before proceeding with next memory operation in program order
- Requires writes be visible in the same order at other processors

```c
P0
void T1(void) {
    A = 1;
    read(B);
}

P1
void T2(void) {
    B = 1;
    read(A);
}
```
Write Completion

• Say write(A, 1) on CPU0 is a write miss
  • Cache coherence protocol sends an invalidate message to other CPUs to invalidate their cached copies of A

• Problem: write completes only after CPU0 receives acknowledgment from CPU1

• Otherwise, another CPU could receive writes out-of-order, perform stale reads, etc.

• Result: writes become slow
Processor Optimization: Store Buffers

- So, let’s not wait for the write completion…
- Record a store in a CPU buffer
- Let CPU proceed immediately
- Send invalidate message, complete the store when invalidate message is acked, i.e., flush the store from the store buffer to the cache
- Causes no issues on uniprocessors
- But what about multiprocessors?
Memory Ordering With Store Buffer

P0
void writer(void) {
    A = 1;
    B = 1;
}

P1
void reader(void) {
    while (B == 0)
        continue;
    assert(A == 1);
}

• Can the assert fail?

• **Assert can fail on some processors 😞, let’s look at why**
Memory Ordering With Store Buffer

P0
A in [invalid], B in [excl]
A = 1;
// save A in store buffer
// send invalidate(A)

B = 1;
// B in [excl],
// so update B in cache

// receive read(B)
// B in [shared]
// send read_reply(B, 1)

How can we fix this problem?

P1
A = [shared], B = [invalid]
while (B == 0)
// read(B)
continue;

assert(A == 1);
// fails
// receive invalidate(A)

// receive read_reply(B, 1)
// exit while loop
assert(A == 1); // fails
// receive invalidate(A)
Memory Ordering With Store Buffer

P0
A in [invalid], B in [excl]
A = 1;
// save A in store buffer
// send invalidate(A)

B = 1;
// B in [excl],
// so update B in cache
// DO NOT UPDATE CACHE UNTIL
// STORE BUFFER IS DRAINED
// receive read(B)
// B in [shared]
// send read_reply(B, 1)

P1
A = [shared], B = [invalid]
while (B == 0)
// read(B)
    continue;

// receive read_reply(B, 1)
// exit while loop
assert(A == 1);
Write Memory Barrier

- **smp_wmb()**
  - Causes the CPU to flush its store buffer before applying subsequent stores to their cache lines
  - The CPU can either
    - Stall until the store buffer is empty before proceeding, or
    - It can use the store buffer to hold subsequent stores until all the prior entries in the buffer had been applied
Memory Ordering With Write Barrier

P0
void writer(void) {
    A = 1;
    smp_wmb();
    B = 1;
}

P1
void reader(void) {
    while (B == 0)
        continue;
    assert(A == 1);
}

• Assert will not fail 😊, let’s look at why
Memory Ordering With Write Barrier

P0

A in [invalid], B in [excl]
A = 1;
// save A in store buffer
// send invalidate(A)
smp_wmb();
// mark store buffer
B = 1;
// store buffer has marked entries,
// so save B in store buffer
// receive read(B)
// B in [shared]
// send read_reply(B, 0)
// receive invalidate_ack(A)
// flush store buffer

P1

A = [shared], B = [invalid]
while (B == 0)
// read(B)
continue;

// receive invalidate(A)
// send invalidate_ack(A)
// receive read_reply(B, 1)
// exit while loop
assert(A == 1); // succeeds
// read(A), reads 1
Invalidate Messages

- Invalidate messages (and their response) can be slow
  - CPU1 cache could be overloaded, so it could respond slowly

- While waiting for invalidate acknowledgements, CPU0 can run out of space in store buffer, stalling execution
Processor Optimization: Invalidate Queues

- So, let’s not wait to invalidate the cache…

- Receive side
  - Stores invalidate request in a queue
  - Acknowledges invalidate right away
  - Applies invalidate later
Invalidate Processing

Write invalidation

CPU0

write(A, 1)

Stall

invalidation

CPU1

apply invalidation

acknowledgement

Write invalidation with invalidate queue

CPU0

write(A, 1)

Stall

invalidation

CPU1

acknowledgement

apply invalidation
### Memory Ordering With Invalidate Queue

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A in [invalid], B in [excl]</strong></td>
<td><strong>A = [shared], B = [invalid]</strong></td>
</tr>
<tr>
<td>A = 1;</td>
<td>while (B == 0)</td>
</tr>
<tr>
<td>// save A in store buffer</td>
<td>// read(B)</td>
</tr>
<tr>
<td>// send invalidate(A)</td>
<td>continue;</td>
</tr>
<tr>
<td>smp_wmb();</td>
<td>// receive invalidate(A)</td>
</tr>
<tr>
<td>// mark store buffer</td>
<td>// queue invalidate(A)</td>
</tr>
<tr>
<td>// receive invalidate_ack(A)</td>
<td>// send invalidate_ack(A)</td>
</tr>
<tr>
<td>// flush store buffer</td>
<td>// receive read_reply(B, 1)</td>
</tr>
<tr>
<td>B = 1;</td>
<td>// exit while loop</td>
</tr>
<tr>
<td>// b in [excl], update B in cache</td>
<td>assert(A == 1); // fails</td>
</tr>
<tr>
<td>// receive read(B)</td>
<td></td>
</tr>
<tr>
<td>// B in [shared]</td>
<td></td>
</tr>
<tr>
<td>// send read_reply(B, 1)</td>
<td></td>
</tr>
</tbody>
</table>

How can we fix this problem?
Memory Ordering With Invalidate Queue

**P0**
A in [invalid], B in [excl]
A = 1;
// save A in store buffer
// send invalidate(A)
smp_wmb();
// mark store buffer

// receive invalidate_ack(A)
// flush store buffer
B = 1;
// b in [excl], update B in cache
// receive read(B)
// B in [shared]
// send read_reply(B, 1)

**P1**
A = [shared], B = [invalid]
while (B == 0)
  // read(B)
    continue;
  // receive invalidate(A)
  // queue invalidate(A)
  // send invalidate_ack(A)

// receive read_reply(B, 1)
// exit while loop
// DRAIN INVALIDATE QUEUE
assert(A == 1);
Read Memory Barrier

- `smp_rmb()`
- Marks all the entries currently in the processor’s invalidate queue
- Forces any subsequent load to wait until all marked entries have been applied to the CPU’s cache
Memory Ordering With Read & Write Barrier

void writer(void) {
    A = 1;
    smp_wmb();
    B = 1;
}

void reader(void) {
    while (B == 0)
        continue;
    smp_rmb();
    assert(A == 1);
}

- Assert will not fail 😊
Memory Ordering Conclusions

- Sequential consistency model makes it easier to write parallel programs since it matches the programmer’s mental model of parallel program execution
  - However, sequential consistency is expensive to implement

- Processors play games by buffering stores and delaying cache invalidations to get good performance
  - Writes may appear to be performed out of order, and reads may return stale data
  - Programmers need to use memory barriers to ensure correct order of cross-CPU memory operations
  - Only programming wizards need apply (as we will see next)!
Memory Consistency and Related Resources

• For an introduction to memory consistency models, see: https://www.cs.utexas.edu/~bornholt/post/memory-models.html

• For an excellent tutorial, see: Shared Memory Consistency Models: A Tutorial Sarita V. Adve, Kourosh Gharachorloo

• For an excellent (online) book, see: A Primer on Memory Consistency and Cache Coherence V. Nagarajan, et al