COMP 790 - 033 - Parallel Computing

Lecture 7
October 5, 2022

CC-NUMA (2)
Memory Consistency and Synchronization Operations

• Reading
  – Patterson & Hennesey, Computer Architecture (2nd Ed.) secn 8.6 – a condensed treatment of consistency models
Coherence and Consistency

• Memory coherence
  – behavior of a single memory location M
  – viewed by one or more processors
  – informally
    • all writes to M are seen in the same order by all processors

• Memory consistency
  – behavior of multiple memory locations read and written by multiple processors
  – viewed by one or more processors
  – informally
    • concerned with the order in which writes on different locations may be seen
Coherence of memory location x

- Defined by three properties (assume x = 0 initially)

\[
\begin{align*}
\text{(a) } & P_1: \ W(x,1) \quad 1 = R(x) \\
& \text{no intervening write of x by } P_1 \text{ or other processor}
\end{align*}
\]

\[
\begin{align*}
\text{(b) } & P_1: \ W(x,1) \\
& P_2: \quad 1 = R(x) \\
& \text{sufficiently large interval and no other write of x}
\end{align*}
\]

\[
\begin{align*}
\text{(c) } & P_1: \ W(x,1) \quad a = R(x) \\
& P_2: \ W(x,2) \quad a = R(x) \quad a \in \{1,2\} \\
& P_3: \quad a = R(x) \quad \text{and has same value at all processors}
\end{align*}
\]

sufficiently large interval and no other writes of x
Consistency Models

• The consistency problem
  – Performance motivates replication
    • Keep data in caches close to processors
  – Replication of read-only blocks is easy
    • No consistency problem
  – Replication of written blocks is hard
    • In what order do we see different write operations?
    • Can we see different orders when viewed from different processors?
  – Fundamental trade-offs
    • Programmer-friendly models perform poorly
Consistency Models

• The importance of a memory consistency model

Initially $A = B = 0$

P1

```
A := 1;
if (B == 0)
  ...
P1 “wins”
```

P2

```
B := 1;
if (A == 0)
  ...
P2 “wins”
```

– P1 and P2 may both win in some consistency models!
  • Violates our (simplistic) mental model of the order of events

• Some consistency models
  • Strict consistency
  • Sequential consistency
  • Processor consistency
  • Release consistency
Strict Consistency

- **Uniprocessor memory semantics**
  - Any read of memory location x returns the value stored by the most recent write operation to x
    - **Natural, simple to program**

\[
P_1: \quad W(x, 1) \\
P_2: \quad 1 = R(x)
\]

Strictly Consistent

\[
P_1: \quad W(x, 1) \\
P_2: \quad 0 = R(x) \quad 1 = R(x)
\]

Non-Strictly Consistent
Strict Consistency

- Implementable in a real system?
  - Requires...
    - absolute measure of time (i.e., global time)
    - slow operation else violation of theory of relativity!
  - Claim: Not what we really wanted (or needed) in the first place!
    - Bad to have correctness depend on relative execution speeds
Sequential Consistency

- Mapping concurrent operations into a single total ordering
  - The result of any execution is the same as if
    - the operations of each processor were performed in sequential order and are interleaved in some fashion to define the total order

\[ P_1: W(x, 1) \]
\[ P_2: 0 = R(x) \quad 1 = R(x) \]

\[ P_1: W(x, 1) \]
\[ P_2: 1 = R(x) \quad 1 = R(x) \]

Both executions are sequentially consistent
Sequential Consistency: Example

- Earlier in time does not imply earlier in the merged sequence
  - is the following sequence of observations sequentially consistent?
  - what is the value of y?

\[
\begin{align*}
P_1: & \quad W(x, 1) \quad \text{?} = R(y) \\
P_2: & \quad W(y, 2) \\
P_3: & \quad 2 = R(y) \quad 0 = R(x) \quad 1 = R(x)
\end{align*}
\]
Processor Consistency

- Concurrent writes by different processors on different variables may be observed in different orders
  - there may not be a single total order of operations observed by all processors
- Writes from a given processor are seen in the same order at all other processors
  - writes on a processor are “pipelined”

<table>
<thead>
<tr>
<th></th>
<th>P_1: W(x, 1)</th>
<th>0 = R(y)</th>
<th>1 = R(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_2:</td>
<td>W(y, 1)</td>
<td>0 = R(x)</td>
<td>1 = R(x)</td>
</tr>
<tr>
<td>P_3:</td>
<td>1 = R(x)</td>
<td>0 = R(y)</td>
<td>1 = R(y)</td>
</tr>
<tr>
<td>P_4:</td>
<td>0 = R(x)</td>
<td>1 = R(y)</td>
<td>1 = R(x)</td>
</tr>
</tbody>
</table>
Processor consistency

- Typical level of consistency found in shared memory multiprocessors
  - insufficient to ensure correct operation of many programs
- Ex: Peterson’s mutual exclusion algorithm

```plaintext
program mutex
var enter1, enter2 : Boolean;
    turn: Integer

process P1
  repeat forever
    enter1 := true
    turn := 2
    while enter2 and turn=2 do skip end
    ... critical section ...
    enter1 := false
    ... non-critical section ...
  end repeat
end P1;

process P2
  repeat forever
    enter2 := true
    turn := 1
    while enter1 and turn=1 do skip end
    ... critical section ...
    enter2 := false
    ... non-critical section ...
  end repeat
end P2;

begin
  enter1, enter2, turn := false, false, 1
  cobegin P1 || P2 coend
end
```
Weak Consistency

- **Observation**
  - memory “fence”
    - if all memory operations up to a checkpoint are known to have completed, the detailed completion order may not be of importance
  - defining a checkpoint
    - a synchronizing operation S issued by processor \( P_i \)
      - e.g. acquiring a lock, passing a barrier, or being released from a condition wait
      - delays \( P_i \) until all outstanding memory operations from \( P_i \) have been completed in other processors

- **Execution rules**
  - synchronizing operations exhibit sequential consistency
  - a synchronizing operation is a memory fence
  - if \( P_i \) and \( P_j \) are synchronized then all memory operations in \( P_i \) complete before any memory operations in \( P_j \) can start
Weak Consistency: Examples

P₁: \[ W(x, 1) \ \ W(y, 2) \ \ S \]

P₂: \[ 1 = R(x) \ \ 0 = R(y) \ \ S \ \ 1 = R(x), \ 2 = R(y) \]

P₃: \[ 0 = R(x) \ \ 2 = R(y) \ \ S \ \ 1 = R(x), \ 2 = R(y) \]

Weakly consistent

P₁: \[ W(x, 1) \ \ W(x, 2) \ \ S \]

P₂: \[ S \ \ 1 = R(x) \]

Not weakly consistent
Memory consistency: processor-centric definition

- A memory consistency model defines which orderings of memory-references made by a processor are preserved for external observers
  - Reference order defined by
    - Instruction order →
    - Reference type \{R,W\} or synchronizing operation (S)
    - location referenced \{a,b\}
  - A memory consistency model preserves some of the reference orders
    - Sequential Consistency (SC), Processor consistency = Total store ordering (TSO), Partial store ordering (PSO), weak consistency

<table>
<thead>
<tr>
<th>reference order</th>
<th>a = b</th>
<th>Consistency Model</th>
<th>a ≠ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>(coherence)</td>
<td></td>
<td>SC</td>
<td>TSO</td>
</tr>
<tr>
<td>Ra → Rb</td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Ra → Wb</td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Wa → Wb</td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Wa → Rb</td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>?a → S → ?b</td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
Consistency models: ordering of “writes”

- **Sequential consistency**
  - all processors see all writes in the same order

- **Processor consistency**
  - All processors see
    - writes from a given processor in the order they were performed (TSO) or in some unknown but fixed order (PSO)
    - writes from different processors may be observed in varying interleavings at different processors

- **Weak consistency**
  - All processors see same state only after explicit synchronization
Example

- OpenMP threads T1, T2
  - variables $a$, $b$, $c$ are shared and initially $a = b = c = 0$
  - $r_1$, $r_2$, $r_3$ are registers
  - which values of $(r_1, r_2, r_3)$ can be observed if printed by the threads?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1: $r_2 := b$</td>
<td>B1: $a := 1$</td>
</tr>
<tr>
<td>A2: $c := 1$</td>
<td>B2: $r_3 := c$</td>
</tr>
<tr>
<td>A3: $r_1 := a$</td>
<td>B3: $b := 1$</td>
</tr>
<tr>
<td>$(r_1, r_2) = ?$</td>
<td>$r_3 = ?$</td>
</tr>
</tbody>
</table>
Memory consistency: Summary

• Memory consistency
  – contract between parallel programmer and parallel processor regarding observable order of memory operations
    • with multiple processors and shared memory, more opportunities to observe behavior
    • therefore more complex contracts

• Where is memory consistency critical?
  – fine-grained parallel programs in a shared memory
    • concurrent garbage collection
    • avoiding race conditions: Java instance constructors
    • constructing high-level synchronization primitives
    • wait-free and lock-free programs
Memory consistency: Summary

• Why memory consistency contracts are difficult to use
  – What memory references does a program perform?
    • Need to understand the output of optimizing compilers
  – In what order may they be observed?
    • Need to understand the memory consistency model
  – How can we construct a correct parallel programs that accommodate these possibilities?
    • Need deep thought and formal methods

• What is a parallel programmer to do, then?
  – Use higher-level concurrency constructs such as loop-level parallelization and synchronized methods (Java)
    • the synchronization inherent in these constructs enables weak consistency models to be used
  – Use machines that provide sequential consistency
    • Increasingly hard to find and invariably “slower”
  – Leave fine-grained unsynchronized memory interaction to the pros
Synchronizing Operations

• Examples
  – *locks* to gain exclusive access for manipulation of shared variables
  – *barrier synchronization* to ensure all processors have reached a program point

• How are these efficiently implemented in a cache-coherent shared memory multiprocessor?
Atomic operations in cc-numa multiprocessors

- **Possible atomic machine operations**

  In the following, `< ... >` refers to atomic execution of action within the brackets, \( m \) is a memory location, and \( r1, r2 \) are processor registers

  - **read and write**
    
    `<r1 := m>`
    
    `<m := r1>`

  - **exchange(m,r1)**
    
    `<r1, m := m, r1>`

  - **test and set(m,r1,r2)**
    
    `<if (m == r1) then m := r2>`

  - **fetch and add(m,r1,r2)**
    
    `<r2 := m + r1; m := r2>`

  - **load-linked(r1,m) and store-conditional(m,r2)**
    
    `<r1 := m>; .... ; <m := r2 or fail>`

    - if \( m \) is updated by another processor between the read and write, the write to \( m \) will not be performed and the condition code \( cc \) will be set to fail
How implemented?

• Atomic read and write
  – simple to implement, difficult to use (recall memory consistency discussion)

• Exchange, test-and-set, fetch-and-add
  – require read-modify-write
    • Involves some hardware-level special coherence protocol

• Load-linked (LL) / Store conditional (SC)
  – LL fetches value into cache line (state = shared)
  – cache-line state is monitored
  – SC fails if cache line has invalid state at time of store
  – Example
    ;; implementation of r2 := fetch-and-add(m,r1) using LL/SC
    try:   ll r3, m
    add   r3, r1, r3   ; r3 := r3 + r1
    sc    r3, m
    bcz   try          ; try again if sc fails
Lock/unlock using atomic operations

- Exchange lock
  - key holds access to the lock
    - key == 0 means lock available
  - to get access, a processor must exchange value 1 with key value 0
    \[
    \begin{align*}
    \{r1 == 1\} \\
    \text{lock: exch r1, key ; spin until zero obtained} \\
    \text{cmpi r1, 0 ;} \\
    \text{bne lock ;} \\
    \{\text{lock obtained}\}
    \end{align*}
    \]
  - to release, exchange with key
    \[
    \begin{align*}
    \{r1 == 0\} \\
    \text{unlock: exch r1, key} \\
    \{\text{lock released}\}
    \end{align*}
    \]
  - what is the effect of spinning on an exchange lock in a CC-NUMA machine?
    - with single processor trying to obtain lock?
      - key is cache-resident in EXCLUSIVE state until released by other processor
    - with multiple processors trying to obtain lock?
      - each exchange brings key into cache and invalidates other copies requiring O(p) cache lines to be refreshed.
Improving cost of contended locks

- “Local” spinning using read-only copy of key
  - avoid coherence traffic while spinning
    ```
    lock:   {r1 == 1}
    try:   lw  r2, key
            cmpi r2,0
            bne try
            {lock observed available}
    exch r1, key
    cmpi r1, 0
    bne try
    {lock obtained}
    ```

- What happens with p processors spinning?
  - No coherence traffic when all processors have key in cache in “shared” state

- What happens when key is released with p processors spinning?
  - key is invalidated and up to p processors observe the lock available
  - up to p processors attempt an exchange
    - one succeeds
    - up to p-1 other processors perform an unsuccessful exch
      - each exch invalidates up to p-2 local copies of key
  - \(O(p^2)\) cache lines moved per lock release
Improving cost of lock release

- LL/SC makes an improvement
  - now $2p$ movements of cache line on release
    
    ```
    lock: {r1 == 1}
    try:  ll  r2, key
    cmpi r2,0
    bne  try
    {lock observed available}
    sc   r1, key
    bz   try
    {lock obtained}
    ```

- basic problem
  - attempt to replicate contended value across caches
  - high cost when $p$ processors contending

- Alternate approaches
  - exponential backoff
    - increase time to re-try with each failure
  - array lock: each process spins on different cache line
Barrier Synchronization

• Delay p processors until all have arrived at barrier
  – simple strategy
    • shared variables: count, release (initially with value 0)
    • in each processor
      lock; count = count + 1; unlock
      if (count == p) then release := 1
      local spinning while release == 0

  – How many cache line moves are required for p processors to pass the barrier?
    • p lock/unlock operations
    • each lock and unlock may have O(p) cache line moves
      – O(p^2) cache line moves in the presence of contention
      – Can we do better?
Barrier synchronization

- Barrier synchronization may have high contention on entry and on release
  - reduce contention on entry using *backoff*
    - exponential backoff in re-attempting lock acquisition
    - random delay in re-attempting lock acquisition
    - both approaches fully serialize entry to the barrier
      - $O(2p)$ cache block movements
  
  - reduce contention on entry and exit using a *combining tree*
    - $O(1)$ contention in lock acquisition
    - $O(p)$ cache line movements
    - $O(lg p)$ lock acquisitions worst case delay
    - more parallelism in scalable shared memory multiprocessors
    - Sometimes implemented in hardware
Dissemination barrier

• Barrier using only atomic reads and writes
  – assume \( p = 2^k \) processors
  – \( \text{arrive}[0 : p-1] \) has initial value zero for all elements.
  – program executed by processor \( i \)

```c
int s = 1;
for (int j = 0; j < k; j++) {
    arrive[i] += 1;
    while (arrive[i] > arrive[(i+s) % p]) { /* spin */ }
    s = 2 * s;
}
/* barrier synchronization achieved */
```

\[ \text{arrive}[i : i+s-1 \mod p] > 0 \]

\[ \text{arrive}[i : i+p-1 \mod p] > 0 \]
Dissemination barrier: example \((p = 4)\)

```c
int s = 1;
for (int j = 0; j < k; j++) {
    arrive[i] += 1;
    while (arrive[i] > arrive[ (i+s) mod p]) { /* spin */ }
    s = 2 * s;
}

s = 4
s = 2
s = 1

0 0 0 0
```