Great Ideas in Computer Architecture

OpenMP, Cache Coherence

Instructor: Sruthi Veeragandham
Review of Last Lecture (1/3)

• Sequential software is slow software
  – SIMD and MIMD only path to higher performance
• Multithreading increases utilization, Multicore more processors (MIMD)
• OpenMP as simple parallel extension to C
  – Small, so easy to learn, but not very high level
  – It’s easy to get into trouble (more today!)
• Synchronization in RISC-V:

• **Atomic Swap:**

  amoswap.w.aq rd,rs2,(rs1)
  amoswap.w.rl rd,rs2,(rs1)

  – swaps the memory value at M[R[rs1]] with the register value in R[rs2]

  – **atomic** because this is done in one instruction

• Another option: lr (load reserve) & sc (store conditional)
Review of Last Lecture (3/3)

- These are defined within a parallel section

- Shares iterations of a loop across the threads
- Each section is executed by a separate thread
- Serializes the execution of a thread
Agenda

• OpenMP Directives
  – Workshare for Matrix Multiplication
  – Synchronization

• Administrivia

• Common OpenMP Pitfalls

• Multiprocessor Cache Coherence

• Break

• Coherence Protocol: MOESI
Matrix Multiplication

\[ C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \]
Naïve Matrix Multiply

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      C[i*N+j] += A[i*N+k] * B[k*N+j];

**Advantage:** Code simplicity

**Disadvantage:** Blindly marches through memory (how does this affect the cache?)
Matrix Multiply in OpenMP

```c
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, i, j, k)
    for (i=0; i<Mdim; i++){
        for (j=0; j<Ndim; j++){
            tmp = 0.0;
            for( k=0; k<Pdim; k++){
                /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
                tmp += *(A+(i*Pdim+k)) * *(B+(k*Ndim+j));
            }
            *(C+(i*Ndim+j)) = tmp;
        }
    }
run_time = omp_get_wtime() - start_time;
```

**Why is there no data race here?**

- Different threads only work on different ranges of i -- inside writing memory access
- Never reducing to a single value (because every write is unique).
Naïve Matrix Multiply

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            C[i*N + j] += A[i*N + k] * B[N*k + j];

Question: What if cache block size > N?
Block Size > N

\[ C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \]

Won’t use last half of the block!
Naïve Matrix Multiply

```c
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i*N+j] += A[i*N+k] * B[N*k+j];
```

**Question:** What if cache block size > N?

— We wouldn’t be using all the data in the blocks that were put in the cache for matrix C and A!

What about if cache block size < N?
Block Size < N

\[ C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \]

Cache Block

Must pull in two blocks instead of one!
Cache Blocking

• Increase the number of cache hits you get by using up as much of the cache block as possible
  – For an N x N matrix multiplication:
    • Instead of *striding* by the dimensions of the matrix, stride by the **blocksize**
    • When N is not perfect divisible by the blocksize, chunk up data as much as possible into block sizes and handle the remainder as a tailcase
• You’ve already done this in lab 7—really try to understand it!
Agenda

• OpenMP Directives
  – Workshare for Matrix Multiplication
  – Synchronization

• Administrivia

• Common OpenMP Pitfalls

• Multiprocessor Cache Coherence

• Break

• Coherence Protocol: MOESI
OpenMP Reduction

• **Reduction**: specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region:
  \[
  \text{reduction} (\text{operation} : \text{var})
  \]
  – **Operation**: perform on the variables (var) at the end of the parallel region
  – **Var**: variable(s) on which to perform scalar reduction

```c
#pragma omp for reduction(+ : nSum)
for (i = START ; i <= END ; ++i)
  nSum += i;
```
Sample use of `reduction`

double compute_sum(double *a, int a_len) {
    double sum = 0.0;
    #pragma omp parallel for reduction(+ : sum)
    for (int i = 0; i < a_len; i++) {
        sum += a[i];
    }
    return sum;
}
Administrivia

• HW6 Released! Due 7/30
• Midterm 2 is tomorrow in lecture!
  – Covering up to Performance
  – There will be discussion after MT2 :(  
  – Check out Piazza for more logistics
• Proj4 Released soon!
• Guerilla session is now Sunday 2-4pm, @Cory 540AB
Agenda

• OpenMP Directives
  – Workshare for Matrix Multiplication
  – Synchronization
• Administrivia
• Common OpenMP Pitfalls
• Multiprocessor Cache Coherence
• Meet the Staff
• Coherence Protocol: MOESI
OpenMP Pitfalls

• We can’t just throw pragmas on everything and expect performance increase 😞
  – Might not change speed much or break code!
  – Must understand application and use wisely

• Discussed here:
  1) Data dependencies
  2) Sharing issues (private/non-private variables)
  3) Updating shared values
  4) Parallel overhead
OpenMP Pitfall #1: Data Dependencies

• Consider the following code:
  
  ```c
  a[0] = 1;
  for(i=1; i<5000; i++)
      a[i] = i + a[i-1];
  ```

• There are dependencies between loop iterations!
  
  – Splitting this loop between threads does not guarantee in-order execution
  
  – Out of order loop execution will result in undefined behavior (i.e. likely wrong result)
Open MP Pitfall #2: Sharing Issues

• Consider the following loop:

```c
#pragma omp parallel for
for (i=0; i<n; i++){
    temp = 2.0*a[i];
    a[i] = temp;
    b[i] = c[i]/temp;
}
```

• `temp` is a shared variable!

```c
#pragma omp parallel for private(temp)
for (i=0; i<n; i++){
    temp = 2.0*a[i];
    a[i] = temp;
    b[i] = c[i]/temp;
}
```
OpenMP Pitfall #3: Updating Shared Variables Simultaneously

- Now consider a global sum:
  
  ```c
  for (i=0; i<n; i++)
      sum = sum + a[i];
  ```

- This can be done by surrounding the summation by a critical/atomic section or reduction clause:
  
  ```c
  #pragma omp parallel for reduction(+:sum)
  {
      for (i=0; i<n; i++)
          sum = sum + a[i];
  } 
  ```

  - Compiler can generate highly efficient code for reduction
OpenMP Pitfall #4: Parallel Overhead

• Spawning and releasing threads results in significant overhead

• Better to have fewer but larger parallel regions
  – Parallelize over the largest loop that you can (even though it will involve more work to declare all of the private variables and eliminate dependencies)
OpenMP Pitfall #4: Parallel Overhead

```c
start_time = omp_get_wtime();
for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
        tmp = 0.0;
        #pragma omp parallel for reduction(+:tmp)
        for (k=0; k<Pdim; k++){
            /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
            tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));
        }
        *(C+(i*Ndim+j)) = tmp;
    }
}
run_time = omp_get_wtime() - start_time;
```

Too much overhead in thread generation to have this statement run this frequently.
Poor choice of loop to parallelize.
Agenda

- OpenMP Directives
  - Workshare for Matrix Multiplication
  - Synchronization
- Administrivia
- Common OpenMP Pitfalls
- Multiprocessor Cache Coherence
- Break
- Coherence Protocol: MOESI
Where are the caches?

Processor 0
- Control
- Datapath
  - PC
  - Registers
  - (ALU)
- Memory Accesses

Memory
- Bytes
- I/O-Memory Interfaces

Processor 1
- Control
- Datapath
  - PC
  - Registers
  - (ALU)
- Memory Accesses

Input

Output
Multiprocessor Caches

• Memory is a performance bottleneck
  – Even with just one processor
  – Caches reduce bandwidth demands on memory

• Each core has a *local* private cache
  – Cache misses access shared common memory
Shared Memory and Caches

• What if?
  – Processors 1 and 2 read Memory[1000] (value 20)
Shared Memory and Caches

• Now:
  – Processor 0 writes Memory[1000] with 40

Problem?
Keeping Multiple Caches Coherent

• Architect’s job: keep cache values **coherent** with shared memory

• Idea: on cache miss or write, notify other processors via interconnection network
  – If **reading**, many processors can have copies
  – If **writing**, **invalidate** all other copies

• Write transactions from one processor “snoop” tags of other caches using common interconnect
  – Invalidate any “hits” to same address in other caches
Shared Memory and Caches

• Example, now with cache coherence
  – Processors 1 and 2 read Memory[1000]
  – Processor 0 writes Memory[1000] with 40
Question: Which statement is TRUE about multiprocessor cache coherence?

(A) Using write-through caches removes the need for cache coherence

(B) Every processor store instruction must check the contents of other caches

(C) Most processor load and store accesses only need to check in the local private cache

(D) Only one processor can cache any memory location at one time
Break
Agenda

• OpenMP Directives
  – Workshare for Matrix Multiplication
  – Synchronization
• Administrivia
• Common OpenMP Pitfalls
• Multiprocessor Cache Coherence
• Break
• Coherence Protocol: MOESI
How Does HW Keep $ Coherent?

- Simple protocol: **MSI**
- Each cache tracks state of each **block** in cache:
  - **Modified**: up-to-date, changed (**dirty**), OK to write
    - no other cache has a copy
    - copy in memory is out-of-date
    - must respond to read request by other processors
  - **Shared**: up-to-date data, not allowed to write
    - other caches may have a copy
    - copy in memory is up-to-date
  - **Invalid**: data in this block is “garbage”
MSI Protocol: Current Processor

Invalid

Read Miss (get block from memory) -> Shared

Write Miss (get block from memory (WB/Write Alloc)) -> Modified

Write Hit

Read Hit

Modified

Read Hit

Write Hit
MSI Protocol: Response to Other Processors

Invalid

Probe Write Hit

Shared

Probe Write Miss (write back to memory)

A probe is another processor

Probe Read Miss (write back to memory)

Modified

Probe Read Hit

Probe Write Hit
How to keep track of state block is in?

- Already have valid bit + dirty bit
- Introduce a new bit called “shared” bit

<table>
<thead>
<tr>
<th></th>
<th>Valid Bit</th>
<th>Dirty Bit</th>
<th>Shared Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Shared</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Invalid</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

X = doesn’t matter
Here there’s only one other processor, but we would have to check & invalidate the data in *every* other processor.
Compatibility Matrix

- Each block in each cache is in one of the following states:
  - **Modified** (in cache)
  - **Shared** (in cache)
  - **Invalid** (not in cache)

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>M</strong></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>I</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Compatibility Matrix**: Allowed states for a given cache block in any pair of caches.
Problem: Writing to Shared is Expensive

• If block is in shared, need to check if other caches have data (so we can invalidate) if we want to write.
• If block is in modified, don’t need to check other caches if we want to write.
  – Why? Only one cache can have data if modified.
Performance Enhancement 1: Exclusive State

- New state: **exclusive**
- **Exclusive**: up-to-date data, OK to write (change to modified)
  - no other cache has a copy
  - copy in memory up-to-date
  - no write to memory if block replaced
  - supplies data on read instead of going to memory
- Now, if block is in shared, at least 1 other cache must contain it:
  - **Shared**: up-to-date data, not allowed to write
    - other caches may definitely have a copy
    - copy in memory is up-to-date
MESI Protocol: Current Processor

- **Invalid**: Read Miss, first to cache data
- **Exclusive**: Read Hit, Write Hit
- **Shared**: Read Hit, other cache(s) have data
- **Modified**: Write Miss (WB/Write Alloc), Read Hit, Write Hit

The diagram illustrates the state transitions based on cache operations.
MESI Protocol: Response to Other Processors

- **Invalid**:
  - Probe Write Hit (write back to memory)

- **Exclusive**:
  - Probe Write Miss

- **Shared**:
  - Probe Read Hit
  - Probe Write Hit

- **Modified**:
  - Probe Read Hit

The diagram illustrates the transitions between the states based on the events (Probe Write Hit, Probe Read Hit).
How to keep track of state block is in?

- New entry in truth table: Exclusive

<table>
<thead>
<tr>
<th></th>
<th>Valid Bit</th>
<th>Dirty Bit</th>
<th>Shared Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Exclusive</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Shared</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Invalid</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

X = doesn’t matter
Problem: Expensive to Share Modified

• In MSI and MESI, if we want to share block in modified:
  1. Modified data written back to memory
  2. Modified block → shared
  3. Block that wants data → shared
• Writing to memory is expensive! Can we avoid it?
Performance Enhancement 2: Owned State

- **Owner**: up-to-date data, read-only (like shared, you can write if you invalidate shared copies first and your state changes to modified)
  - Other caches have a shared copy (**Shared** state)
  - Data in memory not up-to-date
  - **Owner supplies data on probe read** instead of going to memory

- **Shared**: up-to-date data, not allowed to write
  - other caches definitely have a copy
  - copy in memory is *may* be up-to-date
Common Cache Coherency Protocol: MOESI (snoopy protocol)

• Each block in each cache is in one of the following states:
  – Modified (in cache)
  – Owned (in cache)
  – Exclusive (in cache)
  – Shared (in cache)
  – Invalid (not in cache)

Compatibility Matrix: Allowed states for a given cache block in any pair of caches
MOESI Protocol: Current Processor

- **Invalid**
  - Read Miss Exclusive
    - **Exclusive**
      - Read Hit
      - Write Hit
    - Write Miss (WB/Write Alloc)
      - **Modified**
        - Read Hit
        - Write Hit
    - Read Miss
      - **Shared**
        - Read Hit
      - **Owned**
        - Read Hit

- **Shared**
  - Read Hit

- **Owned**
  - Read Hit
MOESI Protocol: Response to Other Processors

- **Invalid**
  - Probe Write Miss
  - Probe Write Hit
  - Probe Read Hit

- **Exclusive**
  - Probe Read Hit

- **Shared**
  - Probe Write Hit
  - Probe Read Hit

- **Owned**
  - Probe Write Hit
  - Probe Read Hit

- **Modified**
  - Probe Read Hit
  - Probe Write Hit
How to keep track of state block is in?

- New entry in truth table: Owned

<table>
<thead>
<tr>
<th>State</th>
<th>Valid Bit</th>
<th>Dirty Bit</th>
<th>Shared Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Owned</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Exclusive</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Shared</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Invalid</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

X = doesn’t matter
MOESI Example

Block 0 is in the exclusive state, so we don’t have to check if the block is in other caches and needs to be invalidated.

Block 0 is never unnecessarily evicted & we don’t waste bandwidth writing to memory.

Processor 0 -- Block 0 Read
Processor 0 -- Block 0 Write
Processor 1 -- Block 0 Read
Processor 0 -- Block 0 Write
Processor 1 -- Block 0 Read
Cache Coherency (MOESI protocol)

Invalid → Exclusive:
- Probe Write Hit
- Read Miss Exclusive

Exclusive → Invalid:
- Read Miss Exclusive

Exclusive → Shared:
- Probe Read Hit
- Write Miss (WB memory)

Shared → Exclusive:
- Read Hit
- Probe Read Hit

Shared → Invalid:
- Read Hit
- Probe Write Hit

Shared → Modified:
- Write Hit
- Read Hit
- Write Hit

Modified → Invalid:
- Write Hit
- Read Hit

Modified → Exclusive:
- Read Hit
- Write Hit

Modified → Shared:
- Write Hit

Modified → Owned:
- Write Hit

Owned → Modified:
- Write Hit

Owned → Shared:
- Read Hit
- Probe Read Hit

(May consider Owned as special case of Shared)

"Read" and "Write" are by this core.
"Probe Read" and "Probe Write" are reads and writes by others, that must probe this core's caches.
• Suppose:
  – Block size is 32 bytes
  – P0 reading and writing variable X, P1 reading and writing variable Y
  – X in location 4000, Y in 4012
• What will happen?
False Sharing

• Block ping-pongs between two caches even though processors are accessing disjoint variables
  – Effect called false sharing

• How can you prevent it?
  – Want to “place” data on different blocks
  – Reduce block size
False Sharing vs. Real Sharing

- If same piece of data being used by 2 caches, ping-ponging is inevitable
- This is **not** false sharing
- Would miss occur if block size was only 1 word?
  - Yes: true sharing
  - No: false sharing
Understanding Cache Misses: The 3Cs

• **Compulsory** (cold start or process migration, 1st reference):
  – First access to a block in memory impossible to avoid
  – Solution: block size ↑ (MP ↑; very large blocks could cause MR ↑)

• **Capacity**:
  – Cache cannot hold all blocks accessed by the program
  – Solution: cache size ↑ (may cause access/HT ↑)

• **Conflict (collision)**:
  – Multiple memory locations map to same cache location
  – Solutions: cache size ↑, associativity ↑ (may cause access/HT ↑)
“Fourth C”: Coherence Misses

• Misses caused by **coherence** traffic with other processor
• Also known as **communication misses** because represents data moving between processors working together on a parallel program
• For some parallel programs, coherence misses can dominate total misses
Summary

• **Synchronization** via hardware primitives:
  – RISCV does it with load reserve + store conditional or amoswap

• **OpenMP** as simple parallel extension to C
  – Synchronization accomplished with critical/atomic/reduction
  – Pitfalls can reduce speedup or break program logic

• **Cache coherence** implements shared memory even with multiple copies in multiple caches
  – **False sharing** a concern