CPE 631 Lecture 21: Multiprocessors

Aleksandar Milenković, milenka@ece.uah.edu
Electrical and Computer Engineering
University of Alabama in Huntsville

Review: Small-Scale—Shared Memory

- Caches serve to:
  - Increase bandwidth versus bus/memory
  - Reduce latency of access
  - Valuable for both private data and shared data

What about cache consistency?

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>$A$</th>
<th>$B$</th>
<th>X (memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CPU A: R x</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CPU B: R x</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>CPU A: W x,0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Snoopy-Cache State Machine-III

MESI: CPU Requests
Fundamental Issues

- 3 Issues to characterize parallel machines
  - 1) Naming
  - 2) Synchronization
  - 3) Performance: Latency and Bandwidth (covered earlier)

Fundamental Issue #1: Naming

- Naming: how to solve large problem fast
  - what data is shared
  - how it is addressed
  - what operations can access data
  - how processes refer to each other

- Choice of naming affects code produced by a compiler; via load where just remember address or keep track of processor number and local virtual address for msg. passing

- Choice of naming affects replication of data; via load in cache memory hierarchy or via SW replication and consistency

Global physical address space:
any processor can generate, address and access it in a single operation
- memory can be anywhere: virtual addr. translation handles it

Global virtual address space: if the address space of each process can be configured to contain all shared data of the parallel program

Segmented shared address space: locations are named
<process number, address> uniformly for all processes of the parallel program

MESI: Bus Requests

Invalid

Exclusive

Modified (read/write)

Shared

BusRd /=>BusWB

BusRdEx /=>BusWB

BusRdEx /=> Sh

BusRdEx /=>BusWB

BusRd /=>BusWB
Fundamental Issue #2: Synchronization

- To cooperate, processes must coordinate
- Message passing is implicit coordination with transmission or arrival of data
- Shared address
  => additional operations to explicitly coordinate:
  e.g., write a flag, awaken a thread, interrupt a processor

Summary: Parallel Framework

- Layers:
  - Programming Model:
    • Multiprogramming:
      lots of jobs, no communication
    • Shared address space:
      communicate via memory
    • Message passing:
      send and receive messages
    • Data Parallel:
      several agents operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)
  - Communication Abstraction:
    • Shared address space: e.g., load, store, atomic swap
    • Message passing: e.g., send, receive library calls
    • Debate over this topic (ease of programming, scaling)
      => many hardware designs 1:1 programming model

Larger MPs

- Separate Memory per Processor
- Local or Remote access via memory controller
- One Cache Coherency solution: non-cached pages
- Alternative: directory per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean,...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/one location)
  - MINUS: In memory => directory is \( f(\text{memory size}) \) vs. \( f(\text{cache size}) \)
- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which Procs have copies of their blocks

Distributed Directory MPs

C - Cache
M - Memory
IO - Input/Output

Interconnection Network
Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date

- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)

- Keep it simple(r):
  - Writes to non-exclusive data => write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

Directory Protocol

- No bus and don’t want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses

- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether exclusive or shared

- Example messages on next slide:
P = processor number, A = address

### Directory Protocol Messages

<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Msg</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
<td>Processor P reads data at address A; make P a read sharer and arrange to send data back</td>
</tr>
<tr>
<td>Write miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
<td>Processor P writes data at address A; make P the exclusive owner and arrange to send data back</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Home directory</td>
<td>Remote caches</td>
<td>A</td>
<td>Invalidate a shared copy at address A. Fetch the block at address A and send it to its home directory.</td>
</tr>
<tr>
<td>Fetch/Invalidate</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
<td>Fetch the block at address A and send it to its home directory; invalidate the block in the cache</td>
</tr>
<tr>
<td>Data value reply</td>
<td>Home directory</td>
<td>Local cache</td>
<td>Data</td>
<td>Return a data value from the home memory (read miss response)</td>
</tr>
<tr>
<td>Data write-back</td>
<td>Remote cache</td>
<td>Home directory</td>
<td>A, Data</td>
<td>Write-back a data value for address A (invalidate response)</td>
</tr>
</tbody>
</table>

### State Transition Diagram for an Individual Cache Block in a Directory Based System

- States identical to snoopy case; transactions very similar
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss msg to home directory
- Write misses that were broadcast on the bus for snooping => explicit invalidate & data fetch requests
- Note: on a write, a cache block is bigger, so need to read the full cache block
CPU - Cache State Machine

- State machine for CPU requests for each memory block.
- Invalid state if in memory.

Invalid state:
- Send Data Write Back message to home directory.
- Send Read Miss message to home directory.

Shared (read only):
- Send Data Write Back message to home directory.
- Send Read Miss message to home directory.

Exclusive (read/write):
- Send Data Write Back message to home directory.
- Send Read Miss message to home directory.

CPU Read hit:
- Send Data Write Back message to home directory.
- Send Read Miss message to home directory.

CPU Write:
- Send Data Write Back message to home directory.
- Send Read Miss message to home directory.

Fetch:
- Send Data Write Back message to home directory.
- Send Read Miss message to home directory.

State Transition Diagram for the Directory

- Same states & structure as the transition diagram for an individual cache.
- 2 actions: update of directory state & send msgs to satisfy requests.
- Tracks all copies of memory block.
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message.

Example Directory Protocol

- Message sent to directory causes two actions:
  - Update the directory
  - More messages to satisfy request
- Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  - Read miss: requesting processor sent data from memory & requestor made only sharing node; state of block made Shared.
  - Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.
- Block is Shared => the memory value is up-to-date:
  - Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  - Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.
Example Directory Protocol

- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:
  - Read miss: owner processor sent data fetch message, causing state of block in owner’s cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor.
    - Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.
  - Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
  - Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write 10 to A1</td>
<td>Read A1</td>
<td>Write A1 to A2</td>
<td>Read A2</td>
<td>A1 and A2 map to the same cache block</td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block
Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block

Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block

Implementing a Directory

- We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network (see Appendix I) – *The devil is in the details*
- Optimizations:
  - read miss or write miss in Exclusive: send data directly to requestor from owner vs. 1st to memory and then from memory to requestor
Parallel Program: An Example

/* Title: Matrix multiplication kernel
 * Author: Aleksandar Milenkovic, milenkovic@computer.org
 * Date: November, 1997
 */

/* Command Line Options
 * -pP: P = number of processors; must be a power of 2
 * -nN: N = number of columns (even integers)
 * -h : Print out command line options.
 */

void main(int argc, char*argv[]) {
    /* Define shared matrix */
    ma = (double **) G_MALLOC(N*sizeof(double *));
    mb = (double **) G_MALLOC(N*sizeof(double *));
    for(i=0; i<N; i++) {
        ma[i] = (double *) G_MALLOC(N*sizeof(double));
        mb[i] = (double *) G_MALLOC(N*sizeof(double));
    }
    /* Initialize the Index */
    Index = 0;
    /* Initialize the barriers and the lock */
    LOCKINIT(indexLock)
    BARINIT(bar_fin)
    /*  read/initialize data   */
    ...
    /* do matrix multiplication in parallel a=a*b */
    /* Create the slave processes. */
    for (i = 0; i < numProcs-1; i++)
        CREATE(SlaveStart)
    /* Make the master do slave work so we
     * don't waste a processor */
    SlaveStart();
    ...
}

---

Parallel Program: An Example

/*====== SlaveStart ================*/
/* This is the routine that each processor will be
 * executing in parallel */
void SlaveStart() {
    int myIndex, i, j, k, begin, end;
    double tmp;
    LOCK(indexLock); /* enter the critical section */
    myIndex = Index; /* read your ID */
    ++Index; /* increment it, so the next
     * will operate on ID+1 */
    UNLOCK(indexLock); /* leave the critical
     * section */
    /* Initialize begin and end */
    begin = (N/numProcs)*myIndex;
    end = (N/numProcs)*(myIndex+1);
    /* the main body of a thread */
    for(i=begin; i<end; i++) {
        for(j=0; j<N; j++) {
            tmp=0.0;
            for(k=0; k<N; k++) {
                tmp = tmp + ma[i][k]*mb[k][j];
            }
            ma[i][j] = tmp;
        }
    }
    BARRIER(bar_fin, numProcs);
}

---

Synchronization

- Why Synchronize? Need to know when it is
  safe for different processes to use shared data
- Issues for Synchronization:
  - Uninterruptable instruction to fetch and update
    memory (atomic operation);
  - User level synchronization operation using this
    primitive;
  - For large scale MPs, synchronization can be a bottleneck;
  - techniques to reduce contention and latency of
    synchronization

Uninterruptable Instruction to Fetch and Update Memory

- Atomic exchange: interchange a value in a register for
  a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    0 if you succeeded in setting the lock (you were first)
    1 if other processor had already claimed access
  - Key is that exchange operation is indivisible
- Test-and-set: tests a value and sets if the value
  passes the test
- Fetch-and-increment: it returns the value of a memory
  location and atomically increments it
  - 0 => synchronization variable is free
Lock&Unlock: Test&Set

/* Test&Set */
===============
loadi R2, #1
lockit: exch R2, location /* atomic operation*/
        bnez R2, lockit /* test*/
unlock: store location, #0 /* free the lock (write 0) */

Lock&Unlock: Test and Test&Set

/* Test and Test&Set */
=======================
lockit: load R2, location /* read lock variable */
        bnz R2, lockit /* check value */
        loadi R2, #1
        exch R2, location /* atomic operation */
        bnez reg, lockit /* if lock is not acquired, repeat */
unlock: store location, #0 /* free the lock (write 0) */

Lock&Unlock: Test and Test&Set

/* Load-linked and Store-Conditional */
=======================================
lockit: ll R2, location /* load-linked read */
        bnz R2, lockit /* if busy, try again */
        load R2, #1
        sc location, R2 /* conditional store */
        beqz R2, lockit /* if sc unsuccessful, try again */
unlock: store location, #0 /* free the lock 0 */

Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
    - Store conditional returns 1 if it succeeds (no other store to same memory location since preceeding load) and 0 otherwise
- Example doing atomic swap with LL & SC:

  try:  mov R3,R4 ; mov exchange value
        li R2,0(R1) ; load linked
        sc R3,0(R1) ; store conditional (returns 1, if Ok)
        beqz R3,try ; branch store fails (R3 = 0)
        mov R4,R2 ; put load value in R4

- Example doing fetch & increment with LL & SC:

  try:  li R2,0(R1) ; load linked
        addi R2,R2,#1 ; increment (Ok if reg-reg)
        sc R2,0(R1) ; store conditional
        beqz R2,try ; branch store fails (R2 = 0)
User Level Synchronization—Operation Using this Primitive

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock
  ```
  li R2,#1
  lockit:  exch R2,0(R1)  ;atomic exchange
  bnez R2,lockit  ;already locked?
  ```

- What about MP with cache coherency?
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables

- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):
  ```
  try:  li R2,#1
  lockit:  lw R3,0(R1)  ;load var
  bnez R3,lockit  ;not free=>spin
  exch R2,0(R1)  ;atomic exchange
  bnez R2,try  ;already locked?
  ```

Barrier Implementation

```c
struct BarrierStruct {
  LOCKDEC(counterlock);
  LOCKDEC(sleeplock);
  int sleepers;
};
#define BARDEC(B) struct BarrierStruct B;
#define BARINIT(B) sys_barrier_init(&B);
#define BARRIER(B,N) sys_barrier(&B, N);
```

Another MP Issue: Memory Consistency Models

- What is consistency? When must a processor see the new value? e.g., seems that
  ```
  P1: A = 0;  P2: B = 0;
  ....  ....
  A = 1;  B = 1;
  L1: if (B == 0) ...  L2: if (A == 0) ...
  ```
  Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?

- Memory consistency models: what are the rules for such cases?
- Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
  ```
  P3: delay all memory operations until all invalidates done
  ```
Memory Consistency Model

- Schemes faster execution to sequential consistency
- Not really an issue for most programs; they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations
    - write (x)
    - release (s) (unlock)
    - acquire (s) (lock)
    - read(x)
- Only those programs willing to be nondeterministic are not synchronized: "data race": outcome f(proc. speed)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses