CS 61C: Great Ideas in Computer Architecture

Amdahl’s Law, Thread Level Parallelism

Instructor: Justin Hsia
1st Half in Review

• Write bigger, talk slower
• Students afraid/too lost to respond
  – Don’t be afraid/ashamed of being lost/wrong
• Peer instruction grading
  – Answers will be added to slides after lecture
• 2nd half material is “clean slate”
• Found copied code! Will be dealt with at end.
  – Don’t dig yourself into a deeper hole...
Review of Last Lecture

• Flynn Taxonomy of Parallel Architectures
  – SIMD: Single Instruction Multiple Data
  – MIMD: Multiple Instruction Multiple Data

• Intel SSE SIMD Instructions
  – One instruction fetch that operates on multiple operands simultaneously
  – 128/64 bit XMM registers
  – Embed the SSE machine instructions directly into C programs through use of intrinsics

• Loop Unrolling: Access more of array in each iteration of a loop
Agenda

• Amdahl’s Law
• Administrivia
• Multiprocessor Systems
• Multiprocessor Cache Coherence
• Synchronization - A Crash Course
Amdahl’s (Heartbreaking) Law

- Speedup due to enhancement E:
  \[
  \text{Speedup w/E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/E}}
  \]

- **Example:** Suppose that enhancement E accelerates a fraction \(F\) (\(F<1\)) of the task by a factor \(S\) (\(S>1\)) and the remainder of the task is unaffected.

  \[
  \text{Exec time w/E} = \text{Exec Time w/o E} \times \left[ (1-F) + \frac{F}{S} \right]
  \]

  \[
  \text{Speedup w/E} = \frac{1}{\left[ (1-F) + \frac{F}{S} \right]}
  \]
Amdahl’s Law

• Speedup = \( \frac{1}{(1 - F) + \frac{F}{S}} \)

  Non-speed-up part \( (1 - F) \)
  Speed-up part \( \frac{F}{S} \)

• Example: the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

\[
\frac{1}{0.5 + 0.5} = \frac{1}{0.5 + 0.25} = 1.33
\]
Consequence of Amdahl’s Law

- The amount of speedup that can be achieved through parallelism is limited by the non-parallel portion of your program!
Parallel Speed-up Examples (1/3)

Speedup w/ \( E = \frac{1}{(1-F) + \frac{F}{S}} \)

- Consider an enhancement which runs 20 times faster but which is only usable 15% of the time
  \[
  \text{Speedup} = \frac{1}{0.85 + 0.15/20} = 1.166
  \]
- What if it’s usable 25% of the time?
  \[
  \text{Speedup} = \frac{1}{0.75 + 0.25/20} = 1.311
  \]
- Amdahl’s Law tells us that to achieve linear speedup with more processors, none of the original computation can be scalar (non-parallelizable)
- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less
  \[
  \text{Speedup} = \frac{1}{0.001 + 0.999/100} = 90.99
  \]
Parallel Speed-up Examples (2/3)

- 10 “scalar” operations (non-parallelizable)
- 100 parallelizable operations
  - Say, element-wise addition of two 10x10 matrices.
- 110 operations
  - $100/110 = 0.909$ Parallelizable, $10/110 = 0.091$ Scalar
Parallel Speed-up Examples (3/3)

Speedup w/ \( E = \frac{1}{(1-F) + \frac{F}{S}} \)

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors
  \[ \text{Speedup} = \frac{1}{0.091 + 0.909/10} = 5.5 \]

- What if there are 100 processors?
  \[ \text{Speedup} = \frac{1}{0.091 + 0.909/100} = 10.0 \]

- What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?
  \[ \text{Speedup} = \frac{1}{0.001 + 0.999/10} = 9.9 \]

- What if there are 100 processors?
  \[ \text{Speedup} = \frac{1}{0.001 + 0.999/100} = 91 \]
Strong and Weak Scaling

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem
  - **Strong scaling:** When speedup is achieved on a parallel processor without increasing the size of the problem
  - **Weak scaling:** When speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors

- **Load balancing** is another important factor: every processor doing same amount of work
  - Just 1 unit with twice the load of others cuts speedup almost in half (bottleneck!)
Question: Suppose a program spends 80% of its time in a square root routine. How much must you speed up square root to make the program run 5 times faster?

\[
\text{Speedup w/ } E = \frac{1}{(1-F) + F/S}
\]

(A) 10
(B) 20
(C) 100
(D) None of the above
Agenda

• Amdahl’s Law
• Administrivia
• Multiprocessor Systems
• Multiprocessor Cache Coherence
• Synchronization - A Crash Course
Midterm Re-grade Policy

- Rubric posted
- Any questions? Ask in discussion section
- *Written* appeal process
  - Explain rationale for re-grade request
  - Attach rationale to exam & submit to your TA by the end of Thursday’s lab
  - ENTIRE test will be re-graded (is possible to end up with a *lower* grade)
Administrivia (2/2)

• Project 2: Matrix Multiply Performance Improvement
  – Work in groups of two!
  – Part 1: Due July 28 (this Sunday)
  – Part 2: Due August 4 (posted by Thu)
• HW 5 also due July 31 (posted Thu)
• Closely packed due dates, try to get ahead of schedule for the project
Agenda

- Amdahl’s Law
- Administrivia
- Multiprocessor Systems
- Multiprocessor Cache Coherence
- Synchronization - A Crash Course
Great Idea #4: Parallelism

**Software**

- **Parallel Requests**
  - Assigned to computer
  - e.g. search “Garcia”

- **Parallel Threads**
  - Assigned to core
  - e.g. lookup, ads

- **Parallel Instructions**
  - > 1 instruction @ one time
  - e.g. 5 pipelined instructions

- **Parallel Data**
  - > 1 data item @ one time
  - e.g. add of 4 pairs of words

**Hardware**

- **Warehouse Scale Computer**
- **Leverage Parallelism & Achieve High Performance**

**Computer**

- Core
- ... Core
- Memory
- Input/Output

**Functional Unit(s)**

- $A_0 + B_0$
- $A_1 + B_1$
- $A_2 + B_2$
- $A_3 + B_3$

**Cache Memory**

**Logic Gates**

- 17

7/22/2013 Summer 2013 -- Lecture #16
Parallel Processing: Multiprocessor Systems (MIMD)

- **Multiprocessor (MIMD):** a computer system with at least 2 processors
  - Use term *core* for processor (“multicore”)

1. Deliver high throughput for independent jobs via request-level or task-level parallelism
Parallel Processing: Multiprocessor Systems (MIMD)

- **Multiprocessor (MIMD):** a computer system with at least 2 processors
  - Use term *core* for processor (“multicore”)

2. Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program
Shared Memory Multiprocessor (SMP)

- Single address space shared by all processors
- Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time

- All multicore computers today are SMP
Memory Model for Multi-threading

- All threads have access to the same, globally shared, memory
- Data can be shared or private
- Shared data is accessible by all threads
- Private data can only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- Synchronization takes place, but it is mostly implicit

Can be specified in a language with MIMD support – such as **OpenMP**
Example: Sum Reduction (1/2)

- Sum 100,000 numbers on 100 processor SMP
  - Each processor has ID: $0 \leq P_n \leq 99$
  - Partition 1000 numbers per processor

- **Step 1:** Initial summation on *each* processor
  
  ```
  sum[P_n] = 0;
  for (i=1000*P_n; i<1000*(P_n+1); i++)
    sum[P_n] = sum[P_n] + A[i];
  ```

- **Step 2:** Now need to add these partial sums
  - *Reduction:* divide and conquer approach to sum
  - Half the processors add pairs, then quarter, ...
  - Need to synchronize between reduction steps
This is **Step 2**, after all “local” sums computed. *This code runs simultaneously on all processors.*

```c
half = 100;
repeat
    synch();
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
        /* When half is odd, P0 gets extra element */
    half = half/2; /* dividing line on who sums */
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```
Sum Reduction with 10 Processors

Question: Given the Sum Reduction code, are the given variables Shared data or Private data?

```
half = 100;
repeat
    synch();
    ... /* handle odd elements */
    half = half/2; /* dividing line */
    if (Pn < half)
        sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

<table>
<thead>
<tr>
<th>half</th>
<th>sum</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>Shared</td>
<td>Shared Shared</td>
</tr>
<tr>
<td>(B)</td>
<td>Shared</td>
<td>Shared Private</td>
</tr>
<tr>
<td><strong>(C)</strong></td>
<td><strong>Private</strong></td>
<td><strong>Shared</strong> Private</td>
</tr>
<tr>
<td>(D)</td>
<td><strong>Private</strong></td>
<td><strong>Private</strong> Shared</td>
</tr>
</tbody>
</table>
Agenda

• Amdahl’s Law
• Administrivia
• Multiprocessor Systems
• Multiprocessor Cache Coherence
• Synchronization - A Crash Course
Shared Memory and Caches

• How many processors can be supported?
  – Key bottleneck in an SMP is the memory system
  – Caches can effectively increase memory bandwidth/open the bottleneck

• But what happens to the memory being actively shared among the processors through the caches?
Shared Memory and Caches (1/2)

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

• What if?
  – Processors 1 and 2 read Memory[1000]
  – Processor 0 writes Memory[1000] with 40

Processor 0
Write
Invalidates
Other Copies
Cache Coherence (1/2)

• **Cache Coherence:** When multiple caches access the same memory, ensure data is consistent in each local cache
  – Main goal is to ensure no cache incorrectly uses an outdated value of memory
• Many different implementations
  – We will focus on **snooping**, where every cache monitors a bus (the interconnection network) that is used to broadcast activity among the caches
Cache Coherence (2/2)

• **Snooping Idea:** When any processor has a cache miss or writes to memory, notify the other processors via the bus
  – If reading, multiple processors are allowed to have the most up-to-date copy
  – If a processor writes, **invalidate** all other copies (those copies are now old)

• What if many processors use the same block?
  – Block can “ping-pong” between caches
Implementing Cache Coherence

• Cache write policies still apply
  – For coherence between cache and memory
  – Write-through/write-back
  – Write allocate/no-write allocate

• What does each cache need?
  – Valid bit? Definitely!
  – Dirty bit? Depends...
  – **New:** Assign each *block* of a cache a *state*, indicating its status relative to the other caches
Cache Coherence States (1/3)

• The following state should be familiar to you:

• **Invalid**: Data is not in cache
  – Valid bit is set to 0
  – Could still be empty or invalidated by another cache; data is not up-to-date
  – This is the only state that indicates that data is NOT up-to-date
Cache Coherence States (2/3)

• The following states indicate sole ownership
  — No other cache has a copy of the data
• Modified: Data is dirty (mem is out-of-date)
  — Can write to block without updating memory
• Exclusive: Cache and memory both have up-to-date copy
Cache Coherence States (3/3)

• The following states indicate that multiple caches have a copy of the data

• *Shared:* One of multiple copies in caches
  – Not necessarily consistent with memory
  – Does not have to write to memory if block replaced (other copies in other caches)

• *Owned:* “Main” copy of multiple in caches
  – Same as Shared, but can supply data on a read instead of going to memory (can only be 1 owner)
Cache Coherence Protocols

• Common protocols called by the subset of caches states that they use:
  - Modified
  - Owned
  - Exclusive
  - Shared
  - Invalid

• e.g. MOESI, MESI, MSI, MOSI, etc.

Snooping/Snoopy Protocols
  e.g. the Berkeley Ownership Protocol
Example: MOESI Transitions

You’ll see more in discussion.
False Sharing

• One block contains both variables $x$ and $y$
• What happens if Processor 0 reads and writes to $x$ and Processor 1 reads and writes to $y$?
  – Cache invalidations even though not using the same data!
• This effect is known as false sharing
• How can you prevent it?
  – e.g. different block size, ordering of variables in memory, processor access patterns
Get To Know Your Staff

• Category: Television
Agenda

• Amdahl’s Law
• Administrivia
• Multiprocessor Systems
• Multiprocessor Cache Consistency
• Synchronization - A Crash Course
Threads

• **Thread of execution**: Smallest unit of processing scheduled by operating system

• On uniprocessor, multithreading occurs by *time-division multiplexing*
  – Processor switches between different threads
  – *Context switching* happens frequently enough user perceives threads as running at the same time

• On a multiprocessor, threads run at the same time, with each processor running a thread
Multithreading vs. Multicore (1/2)

- **Basic idea:** Processor resources are expensive and should not be left idle
- Long memory latency to memory on cache miss?
  - Hardware switches threads to bring in other useful work while waiting for cache miss
  - Cost of thread context switch must be much less than cache miss latency
- Put in redundant hardware so don’t have to save context on every thread switch:
  - PC, Registers, L1 caches?
Multithreading vs. Multicore (2/2)

• Multithreading => Better Utilization
  – ≈1% more hardware, 1.10X better performance?
  – Share integer adders, floating point adders, caches (L1 I $, L1 D$, L2 cache, L3 cache), Memory Controller

• Multicore => Duplicate Processors
  – ≈50% more hardware, ≈2X better performance?
  – Share lower caches (L2 cache, L3 cache), Memory Controller
Data Races and Synchronization

• Two memory accesses form a data race if different threads access the same location, and at least one is a write, and they occur one after another
  – Means that the result of a program can vary depending on chance (which thread ran first?)
  – Avoid data races by synchronizing writing and reading to get deterministic behavior

• Synchronization done by user-level routines that rely on hardware synchronization instructions
Analogy: Buying Milk

• Your fridge has no milk. You and your roommate will return from classes at some point and check the fridge
• Whoever gets home first will check the fridge, go and buy milk, and return
• What if the other person gets back while the first person is buying milk?
  – You’ve just bought twice as much milk as you need!
• It would’ve helped to have left a note...
Lock Synchronization (1/2)

• Use a “Lock” to grant access to a region (critical section) so that only one thread can operate at a time
  – Need all processors to be able to access the lock, so use a location in shared memory as the lock

• Processors read lock and either wait (if locked) or set lock and go into critical section
  – 0 means lock is free / open / unlocked / lock off
  – 1 means lock is set / closed / locked / lock on
Lock Synchronization (2/2)

• Pseudocode:

  Check lock
  Set the lock
  Critical section
  (e.g. change shared variables)
  Unset the lock

Can loop/idle here if locked
Possible Lock Implementation

• **Lock (a.k.a. busy wait)**
  
  Get\_lock:    # $s0 -> address of lock
  addiu $t1,$zero,1    # t1 = Locked value
  Loop:  lw $t0,0($s0)    # load lock
         bne $t0,$zero,Loop  # loop if locked
  Lock:  sw $t1,0($s0)    # Unlocked, so lock

• **Unlock**
  
  Unlock:
  sw $zero,0($s0)

• **Any problems with this?**
Possible Lock Problem

• Thread 1
  addiu $t1,$zero,1
  Loop: lw $t0,0($s0)
  bne $t0,$zero,Loop
  Lock: sw $t1,0($s0)

• Thread 2
  addiu $t1,$zero,1
  Loop: lw $t0,0($s0)
  bne $t0,$zero,Loop
  Lock: sw $t1,0($s0)

Both threads think they have set the lock! Exclusive access not guaranteed!
Hardware Synchronization

• Hardware support required to prevent an interloper (another thread) from changing the value
  – *Atomic* read/write memory operation
  – No other access to the location allowed between the read and write

• How best to implement?
  – Single instr? Atomic swap of register ↔ memory
  – Pair of instr? One for read, one for write
Synchronization in MIPS

• **Load linked:** \( \text{ll } rt, \text{off}(rs) \)

• **Store conditional:** \( \text{sc } rt, \text{off}(rs) \)
  
  – Returns \( 1 \) (success) if location has not changed since the \( \text{ll} \)
  
  – Returns \( 0 \) (failure) if location has changed

• Note that \( \text{sc} \) **clobbers** the register value being stored \((rt)\)!
  
  – Need to have a copy elsewhere if you plan on repeating on failure or using value later
Synchronization in MIPS Example

- Atomic swap (to test/set lock variable)
  Exchange contents of register and memory: $s4 ←→ Mem($s1)

  try: add $t0,$zero,$s4  #copy value
       ll $t1,0($s1)     #load linked
       sc $t0,0($s1)    #store conditional
       beq $t0,$zero,try #loop if sc fails
       add $s4,$zero,$t1 #load value in $s4

  sc would fail if another threads executes sc here
Test-and-Set

• In a single atomic operation:
  – *Test* to see if a memory location is set (contains a 1)
  – *Set* it (to 1) if it isn’t (it contained a zero when tested)
  – Otherwise indicate that the Set failed, so the program can try again
  – While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions

• Useful for implementing lock operations
Test-and-Set in MIPS

- Example: MIPS sequence for implementing a T&S at ($s1)
  
  Try: addiu $t0, $zero, 1
  ll $t1, 0($s1)
  bne $t1, $zero, Try
  sc $t0, 0($s1)
  beq $t0, $zero, try

  Locked:

  # critical section

  Unlock:
  sw $zero, 0($s1)
Summary

• Amdahl’s Law limits benefits of parallelization
• Multiprocessor systems uses shared memory (single address space)
• Cache coherence implements shared memory even with multiple copies in multiple caches
  – Track state of blocks relative to other caches (e.g. MOESI protocol)
  – False sharing a concern
• Synchronization via hardware primitives:
  – MIPS does it with Load Linked + Store Conditional