You Are Here!

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- 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 descriptions
  All gates functioning in parallel at same time

Agenda

- Multiprocessor Systems
- Administrivia
- Multiprocessor Cache Consistency
- Synchronization
- Technology Break
- OpenMP Introduction
- Summary

Parallel Processing: Multiprocessor Systems (MIMD)

- Multiprocessor (MIMD): a computer system with at least 2 processors

1. Deliver high throughput for independent jobs via request-level or task-level parallelism
2. Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program

Now use term core for processor (“Multicore”) because “Multiprocessor Microprocessor” is redundant.
### Multiprocessors and You

- Only path to performance is parallelism
  - Clock rates flat or declining
  - SIMD: 2X width every 3-4 years
    - 128b wide now, 256b in 2011, 512b in 2014, 1024b in 2018?
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, ...
- Key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase — i.e., that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication
- **Project #3**: fastest matrix multiply code on 8 processor (8 cores) computers
  - 2 chips (or sockets)/computer, 4 cores/chip

### Potential Parallel Performance
(Assuming SW can use it!)

<table>
<thead>
<tr>
<th>Year</th>
<th>Cores</th>
<th>SIMD bits /Core</th>
<th>Core * SIMD bits</th>
<th>Peak DP GFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>MIMD 2</td>
<td>SIMD 128</td>
<td>256</td>
<td>MIMD 4</td>
</tr>
<tr>
<td>2005</td>
<td>4</td>
<td>SIMD 128</td>
<td>512</td>
<td>SIMD 8</td>
</tr>
<tr>
<td>2007</td>
<td>6</td>
<td>SIMD 128</td>
<td>768</td>
<td>12</td>
</tr>
<tr>
<td>2009</td>
<td>8</td>
<td>SIMD 128</td>
<td>1024</td>
<td>16</td>
</tr>
<tr>
<td>2011</td>
<td>10</td>
<td>SIMD 256</td>
<td>2560</td>
<td>40</td>
</tr>
<tr>
<td>2013</td>
<td>12</td>
<td>SIMD 256</td>
<td>3072</td>
<td>48</td>
</tr>
<tr>
<td>2015</td>
<td>14</td>
<td>SIMD 512</td>
<td>7168</td>
<td>112</td>
</tr>
<tr>
<td>2017</td>
<td>16</td>
<td>SIMD 512</td>
<td>8192</td>
<td>128</td>
</tr>
<tr>
<td>2019</td>
<td>18</td>
<td>SIMD 1024</td>
<td>18432</td>
<td>288</td>
</tr>
<tr>
<td>2021</td>
<td>20</td>
<td>SIMD 1024</td>
<td>20480</td>
<td>320</td>
</tr>
</tbody>
</table>

### Three Key Questions about Multiprocessors

- **Q1** — How do they share data?
  - Single address space shared by all processors/cores
- **Q2** — How do they coordinate?
- **Q3** — How many processors can be supported?

### Three Key Questions about Multiprocessors

- **Q1** — How do they share data?
- **Q2** — How do they coordinate?
- 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
- **Q3** — How many processors can be supported?
- All multicore computers today are Shared Memory Multiprocessors (SMPs)
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor SMP
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor:
    \[ \text{sum}[Pn] = 0; \]
    for \( i = 1000*Pn; i < 1000*(Pn+1); i = i + 1 \)
    \[ \text{sum}[Pn] = \text{sum}[Pn] + A[i]; \]
- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, ...
  - Need to synchronize between reduction steps

An Example with 10 Processors

- \[ \text{sum}[P0] \]
- \[ \text{sum}[P1] \]
- \[ \text{sum}[P2] \]
- \[ \text{sum}[P3] \]
- \[ \text{sum}[P4] \]
- \[ \text{sum}[P5] \]
- \[ \text{sum}[P6] \]
- \[ \text{sum}[P7] \]
- \[ \text{sum}[P8] \]
- \[ \text{sum}[P9] \]
- \( \text{half} = 10 \)

Three Key Questions about Multiprocessors

- Q3 – 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

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

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

![Diagram of memory and cache interaction]

Agenda

- Multiprocessor Systems
- Administrivia
- Multiprocessor Cache Consistency
- Synchronization
- Technology Break
- OpenMP Introduction
- Summary

Course Organization

- Grading
  - Participation and Altruism (5%)
  - Homework (5%) – 4 of 6 HWs Completed
  - Labs (20%) – 7 of 12 Labs Completed
  - Projects (40%)
    1. Data Parallelism (Map-Reduce on Amazon EC2)
    2. Computer Instruction Set Simulator (C)
    3. Performance Tuning of a Parallel Application/Matrix Multiply using cache blocking, SIMD, MIMD (OpenMP, due with partner)
    4. Computer Processor Design (Logisim)
  - Midterm (10%): 6-9 PM Tuesday March 8
  - Final (20%): 11:30-2:30 PM Monday May 9

Midterm Results

- Students with this score

EECS Grading Policy

- [Link to EECS grading policy page]

  "A typical GPA for courses in the lower division is 2.7. This GPA would result, for example, from 17% A’s, 50% B’s, 20% C’s, 10% D’s, and 3% F’s. A class whose GPA falls outside the range 2.5 - 2.9 should be considered atypical."

  - Fall 2010: GPA 2.81
    - 26% A’s, 47% B’s, 17% C’s, 3% D’s, 6% F’s
  - Job/Intern Interviews: They grill you with technical questions, so it’s what you say, not your GPA
    - (New 61c gives good stuff to say)
Administrivia

- Regrade Policy
  - Rubric on-line (soon!)
  - Any questions? Covered in Discussion Section next week
  - Written appeal process
    - Explain rationale for regrade request
    - Attach rationale to exam
    - Submit to your TA in next week’s laboratory

Next Lab and Project

- Lab #8: Data Level Parallelism, posted
- Project #3: Matrix Multiply Performance Improvement
  - Work in groups of two!
  - Part 1: March 27 (end of Spring Break)
  - Part 2: April 3
- HW #5 also due March 27
  - Posted soon

CS 61c in the News

Spring Play

IN THE MATTER OF J. ROBERT OPPENHEIMER
Thursday, March 10 & Friday, March 11
GENIUS? No Doubt
IDEALIST? No Doubt
PATRIOT? Doubt
SPY? To Be Discussed

Resonant in the age of WikiLeaks, In The Matter of J. Robert Oppenheimer explores the ethical and political ramifications of one man’s struggle 60 years ago to Do the Right Thing -- a struggle with which we all contend on a daily basis.

The questions: Did Oppenheimer, who led the team that developed the A-bomb which won the war against Japan, deliberately delay the creation of the H-bomb and thereby jeopardize our security? Did this man prolong the Cold War? Given his Communist past, was he a patriot, a traitor, a spy, a moralizing elitist – or, all of these?

“The hydrogen bomb wasn’t ready.”
– J. Robert Oppenheimer, “The father of the atomic bomb”

“Except for Oppenheimer, we would have had the H-bomb before the Russians”

“Not till the shock of Hiroshima did we scientists grasp the awful consequences of our work.”
– Hans Bethe, Nobel Laureate in physics

“There was a conspiracy of scientists against the bomb, and it was led by Oppenheimer.”
– David Tressel Griggs, Chief Scientist of the Air Force

Join me in the Jinks Theater, and together we will consider the Matter of Dr. J. Robert Oppenheimer.

Richard Muller, Sire

Thursday evening: no reservations required
Friday evening, Ladies Night, reservations required
Member plus 3 guest limit

5:30 p.m. Duck & Cover into the Cartoon Room for cocktails
6:45 p.m. Split some culinary atoms at Dinner in the Dining Room
8:30 p.m. Enter the nuclear world of the Jinks Theater
9:30 p.m. Enjoy a real Afterglow with a radioactive cocktail in the Cartoon Room with Bob Markison & Friends on Thursday and Ron Sfarzo & Bob Sulpizio on Friday.

Next Thursday Show: March 17, Ed Sullivan

Agenda

- Multiprocessor Systems
- Administrivia
- Multiprocessor Cache Consistency
- Synchronization
- Technology Break
- OpenMP Introduction
- Summary

Keeping Multiple Caches Coherent

- Architect’s job: shared memory => keep cache values coherent
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate all other copies
- Shared written result can “ping-pong” between caches
How Does HW Keep $\text{Coherent}$?

• Each cache tracks state of each block in cache:
  1. Shared: up-to-date data, other caches may have a copy
  2. Modified: up-to-date data, changed (dirty), no other cache has a copy, OK to write, memory out-of-date

Name of Common Cache Coherency Protocol: MOESI

• Memory access to cache is either
  - Modified (in cache)
  - Owned (in cache)
  - Exclusive (in cache)
  - Shared (in cache)
  - Invalid (not in cache)

Cache Coherency and Block Size

• Suppose block size is 32 bytes
• Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
• Suppose in X location 4000, Y in 4012
• What will happen?
• Effect called false sharing
• How can you prevent it?

Threads

• Thread of execution: smallest unit of processing scheduled by operating system
• On single/uni-processor, multithreading occurs by time-division multiplexing:
  - Processor switched 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

Data Races and Synchronization

• Two memory accesses form a data race if from different threads to same location, and at least one is a write, and they occur one after another
• If there is a data race, result of 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
Lock and Unlock Synchronization

- Lock used to create region \textit{(critical section)} where only one thread can operate
- Given shared memory, use memory location as synchronization point: \texttt{lock} or \texttt{semaphore}
- Processors read lock to see if must wait, or OK to go into critical section (and set to locked)
  - 0 => lock is free / open / unlocked / lock off
  - 1 => lock is set / closed / locked / lock on

Set the lock

Critical section

\texttt{(only one thread gets to execute this section of code at a time)} e.g., change shared variables

Unset the lock

Possible Lock/Unlock Implementation

- Lock (aka busy wait):
  - \texttt{addiu \$t1,\$zero,1} ; \$t1 = Locked value
  - Loop: \texttt{lw \$t0,lock(\$s0)} ; load lock
  - bne \$t0,\$zero,Loop ; loop if locked
  - \texttt{sw \$t1,lock(\$s0)} ; Unlocked, so lock

- Unlock:
  - \texttt{sw \$zero,lock(\$s0)}

- Any problems with this?

Possible Lock Problem

\begin{itemize}
  \item \texttt{Thread 1}
    \begin{verbatim}
    addiu \$t1,\$zero,1
    Loop: lw \$t0,lock(\$s0)
    bne \$t0,\$zero,Loop
    Lock: sw \$t1,lock(\$s0)
    \end{verbatim}
  \item \texttt{Thread 2}
    \begin{verbatim}
    addiu \$t1,\$zero,1
    Loop: lw \$t0,lock(\$s0)
    bne \$t0,\$zero,Loop
    Lock: sw \$t1,lock(\$s0)
    \end{verbatim}
\end{itemize}

Time

Both threads think they have set the lock

Exclusive access not guaranteed!

Help! Hardware Synchronization

- Hardware support required to prevent interloper (either thread on other core or thread on same core) from changing the value
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register \texttt{$\leftrightarrow$} memory
  - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: \texttt{ll rt,offset(rs)}
- Store conditional: \texttt{sc rt,offset(rs)}
  - Succeeds if location not changed since the \texttt{ll}
  - Returns 1 in rt (clobbers register value being stored)
  - Fails if location has changed
  - Returns 0 in rt (clobbers register value being stored)
- Example: atomic swap (to test/set lock variable)
  - Exchange contents of reg and mem: \$s4 \texttt{$\leftrightarrow$} \(\$s1\)
  - \texttt{try: add \$t0,\$zero,\$s4} ; copy exchange value
  - \texttt{li \$t1,0(\$s1)} ;load linked
  - \texttt{sc \$t0,0(\$s1)} ;store conditional
  - \texttt{beq \$t0,\$zero,try} ;branch store fails
  - \texttt{add \$s4,\$zero,\$t1} ;put load value in \$s4

Test-and-Set

- In a single atomic operation:
  - \texttt{Test} to see if a memory location is set (contains a 1)
  - \texttt{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
  - No other instruction can modify the memory location, including another Test-and-Set instruction
  - 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
  li $t1, 0($s1)
  bne $t1, $zero, Try
  sc $t0, 0($s1)
  beq $t0, $zero, try
  Locked:
  critical section
  sw $zero, 0($s1)
  ```

Agenda

- Multiprocessor Systems
- Administrivia
- Multiprocessor Cache Consistency
- Synchronization
- Technology Break
- OpenMP Introduction
- Summary

Machines in 61C Lab

```bash
/usr/sbin/sysctl -a | grep hw\n
hw.model = MacPro4,1
hw.cachelinesize = 64
hw.cpufrequency = 2,260,000,000
hw.physmem = 2,147,483,648
```

Therefore, should try up to 16 threads to see if performance gain even though only 8 cores
Randy’s Laptop

```
Randy's Laptop

hw.model = MacBookAir3,1
... 
hw.physicalcpu: 2
hw.logicalcpu: 2
... 
hw.cpufrequency: 1,600,000,000
hw.physmem = 2,147,483,648
```

OpenMP

- OpenMP is an API used for multi-threaded, shared memory parallelism
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Portable
- Standardized

OpenMP Programming Model

- Shared Memory, Thread Based Parallelism
  - Multiple threads in the shared memory programming paradigm
  - Shared memory process consists of multiple threads
- Explicit Parallelism
  - Explicit programming model
  - Full programmer control over parallelization

OpenMP Uses the C Extension Pragmas Mechanism

- Pragmas are a mechanism C provides for language extensions
- Many different uses of pragmas: structure packing, symbol aliasing, floating point exception modes, ...
- Good for OpenMP because compilers that don’t recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas
Building Block: the **for** loop

```
for (i=0; i<max; i++) zero[i] = 0;
```

- Break for loop into chunks, and allocate each to a separate thread
  - E.g., if max = 100, with two threads, assign 0-49 to thread 0, 50-99 to thread 1
- Must have relatively simple “shape” for OpenMP to be able to parallelize it simply
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed
  - i.e., No break, return, exit, goto statements

OpenMP: Parallel **for** *pragma*

```
#pragma omp parallel for
for (i=0; i<max; i++) zero[i] = 0;
```

- Master thread creates multiple threads, each with a separate execution context
  - E.g., if max = 100, with two threads, assign 0-49 to thread 0, 50-99 to thread 1
- Must have relatively simple “shape” for OpenMP to be able to parallelize it simply
  - Necessary for the runtime system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed
  - i.e., No break, return, exit, goto statements

Thread Creation

- How many threads will OpenMP create?
- Defined by `OMP_NUM_THREADS` environment variable
- Set this variable to the maximum number of threads you want OpenMP to use
- Usually equals the number of cores in the underlying HW on which the program is run

```c
#include <omp.h>
#include <stdio.h>

main () {
  int nthreads, tid;
  /* Fork team of threads with each having a private tid variable */
  #pragma omp parallel private(tid)
  {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  } /* All threads join master thread and terminate */
}
```
Hello World in OpenMP

localhost:OpenMP randykatz$ ./omp_hello
Hello World from thread = 0
Hello World from thread = 1
Number of threads = 2

OpenMP Directives

OpenMP Critical Section

#include <omp.h>
main()
{
  int x;
  x = 0;
  #pragma omp parallel shared(x)
  {
    #pragma omp critical
    x = x + 1;
  } /* end of parallel section */
}

And In Conclusion, ...

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multiprocessor (Multicore) uses Shared Memory (single address space)
- Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern
- Synchronization via hardware primitives:
  - MIPS does it with Load Linked + Store Conditional
- OpenMP as simple parallel extension to C
- More OpenMP examples next time