CS 61C: Great Ideas in Computer Architecture (Machine Structures) Thread Level Parallelism

Instructors:
Randy H. Katz
David A. Patterson

http://inst.eecs.Berkeley.edu/~cs61c/fa10

Review

- Intel SSE SIMD Instructions
  - One instruction fetch that operates on multiple operands simultaneously
- SSE Instructions in C
  - Can embed the SEE machine instructions directly into C programs through the use of intrinsics

Review: Flynn Taxonomy

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

- SISD and MIMD most commonly encountered
- Most common parallel processing programming style: Single Program Multiple Data
  - Single program that runs on all processors of an MIMD

Agenda

- Multiprocessor
- Cache Coherence
- Administrivia
- Technology Break
- Synchronization
- OpenMP (if there is time)

Parallel Processing: Multiprocessor Systems (MIMD)

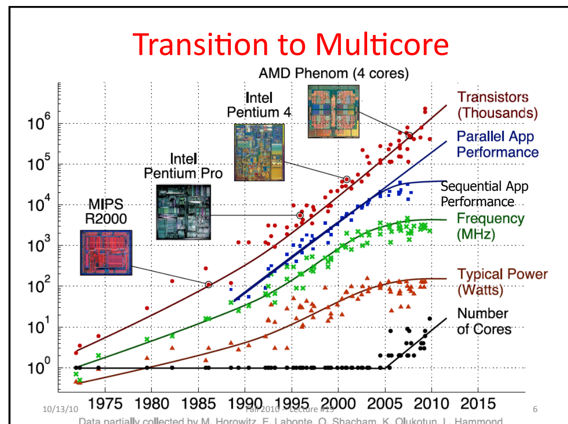
- Multiprocessor (MIMD): a computer system with at least 2 processors

Processor Processor Processor
Cache Cache Cache
Interconnection Network
Memory I/O

1. Deliver high throughput for independent jobs via job-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" too redundant

Transition to Multicore



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 2011, 512b in 2014?, 1024b in 2018?
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, …
- A 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 4: fastest code on 8 processor computers
  - 2 chips/computer, 4 cores/chip

Parallel Performance Over Time

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

Multiprocessor Key Questions

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How many processors can be supported?

Shared Memory Multiprocessor (SMP)

- Q1 – Single address space shared by all processors/cores
- Q2 – 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

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

```
sum[Pn] = 0;
  for (i = 1000*Pn;
      i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = 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

Example: Sum Reduction


```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

An Example with 10 Processors

sum[P0] sum[P1] sum[P2] sum[P3] sum[P4] sum[P5] sum[P6] sum[P7] sum[P8] sum[P9]

P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 half = 10

An Example with 10 Processors

sum[P0] sum[P1] sum[P2] sum[P3] sum[P4] sum[P5] sum[P6] sum[P7] sum[P8] sum[P9]

P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 half = 10

P0 P1 P2 P3 P4 half = 5

P0 P1 half = 2

P0 half = 1

Shared Memory and Caches

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

Processor 0 | Processor 1 | Processor 2
Cache | 1000 e | 1000 e
1000 | 1000
Interconnection Network
Memory 20 | I/O

Shared Memory and Caches

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

1000
Processor 0 | Processor 1 | Processor 2
1000 40 | 1000 20 | 1000 20
Interconnection Network
1000 40 | I/O

Processor 0 Write Invalidates Other Copies

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 $ 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

2 Optional Performance Optimizations of Cache Coherency via new States

- Each cache tracks state of each *block* in cache:

3. Exclusive: up-to-date data, no other cache has a copy, OK to write, memory up-to-date
  - Avoids writing to memory if block replaced
  - Supplies data on read instead of going to memory

4. Owner: up-to-date data, other caches may have a copy (they must be in Shared state)
  - Only cache that supplies data on read instead of going to memory

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)

Agenda

- Multiprocessor
- Cache Coherence
- Administrivia
- Synchronization
- OpenMP (if there is time)

Survey

- Too much reading?
  - To midterm - K&R: 97 pg; WSC: 33; P&H: 143 = 273 total pages
  - Midterm to end - P&H: 186 pg, Handouts: 39 pages = 225 total pages
  - 1st 7 weeks: average is 39 pages / week
  - Last 8 weeks: average is 28 pages / week
  - ~10% don't read book or look at slides
- Highest rated assignment (learned the most): Project 1 42% "I enjoyed it and learned a lot" + 40% "it was a satisfactory assignment and I learned some"

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 1 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

- 2 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 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 ("*critical section*") where only 1 processor can operate
- Given shared memory, use memory location to act synchronization point: "lock"
- Processors read it to see if must wait, or OK to go into critical section (and set to locked)
- 0 => lock is free / open / unlocked
- 1 => lock is taken / closed / locked

Peer Instruction: What Happens?

```
           addiu $t1,$zero, 1       ; t1 = Locked value
Tryagain:  lw $t0, lock($s0)        ; load lock
           beq $t0, $zero, Trygain  ; loop if 0
HaveLock:  sw $t1, lock($s0)        ; Lock must be 1?
```

I. Implements lock correctly
II. Infinite Loop, since no change to lock before beq
III. Doesn't work because another core could read lock in memory before sw changes it to 1, go to critical section
IV. Doesn't work because OS could schedule another thread on this core between lw and sw, and the other thread could go into critical section

A)(red) I only
B)(orange) II only
C)(green) III only
D)(yellow) IV only
E)(burgundy) III and IV

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 ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```


OpenMP

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

Fork/Join Parallelism

- Start out executing the program with one master thread
- Master thread forks worker threads
- Worker threads die or suspend at end of parallel code

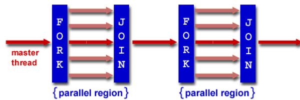


Image courtesy of http://www.llnl.gov/computing/tutorials/openMP/

Simple Parallelization

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

- For loop must have canonical shape for OpenMP to parallelize it
  - Necessary for run-time system to determine loop iterations
- No premature exits from the loop allowed
  - i.e., No break, return, exit, goto statements

OpenMP Extends C with Pragmas

- Pragmas are a mechanism C provides for language extensions
- Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes
- Good mechanism for OpenMP because compilers that don't recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas

The parallel `for` pragma

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

- Master thread creates additional threads, each with a separate execution context
- All variables declared outside for loop are shared by default, except for loop index

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
  - Presumably = number of processors in computer running program

Summary

- 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