

## 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>

10/15/10 Fall 2010 -- Lecture #19 1

## Agenda

- Review
- Data Races and Synchronization
- Hardware Support for Multithreading
- Atomic Operations in MIPS
- Administrivia
- Technology Break
- OpenMP with Examples
- Summary

10/15/10 Fall 2010 -- Lecture #19 2

## Review

- 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
- Thread: unit of execution that OS schedules
  - Within one core + across multiple cores

10/15/10 Fall 2010 -- Lecture #19 3

## Review: Potential Parallel Performance (assuming SW can use it)

Year	Cores	SIMD bits /Core	Core * SIMD bits	Peak DP FLOPs
2003	MIMD 2	SIMD 128	256	MIMD 4
2005	+2/ 4	2X/ 128	512	*SIMD 8
2007	2yrs 6	4yrs 128	768	12
2009	8	128	1024	16
2011	10	256	2560	40
2013	12	256	3072	48
2015	2.5X 14	8X 512	7168	20X 112
2017	16	512	8192	128
2019	18	1024	18432	288
2021	20	1024	20480	320

10/15/10 Fall 2010 -- Lecture #19 4

## Lock and Unlock Synchronization

- Lock used to create region (“*critical section*”) where only 1 processor can operate
- Given shared memory, use memory location as synchronization point: “*lock*” or “*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 taken / closed / locked / lock on

10/15/10

Fall 2010 -- Lecture #19

5

## Peer Instruction: What Happens?

```

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

```

- Implements lock correctly
- Infinite Loop, since no change to lock before beq
- Doesn't work because another core could read lock in memory before sw changes it to 1, go to critical section
- 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
  - But MIPS does have instructions that both read and write memory
- Or an pair of instructions that acts atomically

10/15/10

Fall 2010 -- Lecture #19

7

## Atomic Operation 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 (clobbers register value being stored)
  - Fails if location is changed
    - Returns 0 in rt (clobbers register value being stored)
- 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

```

Fall 2010 -- Lecture #19

8

## Synchronization in MIPS

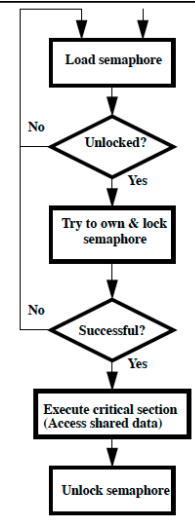
- Example: Check for lock using atomic sequence  
 Try: `addiu $t0,$zero,1 ;copy locked value`  
`ll $t1,0($s1) ;load linked`  
`bne $t1,$zero,try ;loop if lock was 1`  
`sc $t0,0($s1) ;==0, store conditional`  
`beq $t0,$zero,try ;branch store fails,`  
 `;try again`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in `rt` (clobbers register value being stored)
  - Fails if location is changed
    - Returns 0 in `rt` (clobbers register value being stored)

10/15/10 - Lecture #19

9

## Test and Set

- Test-and-set achieves synchronization where only 1 processor is allowed to access a critical section. In general, this technique involves using a variable called the *semaphore* and assigning values to this variable for the “lock-off” or “lock-on” state. Semaphore can be interpreted as a lock to some critical section. Each processor checks if the lock is off; if so, it tries to lock it by modifying the variable appropriately. Once a processor gets the lock, it is then allowed to modify restricted data or access the critical section. After it is done, it gets out of the critical section and modifies the *semaphore* to the “lock-off” state so that other processors can get a chance to access it.



## Multithreading vs. Multicore

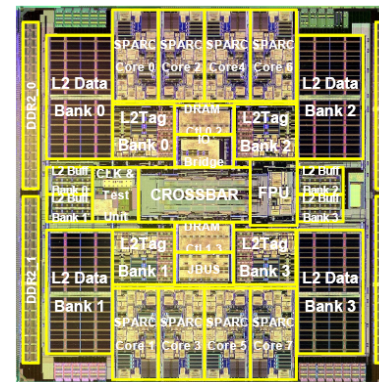
- Basic idea: Processor resources are expensive and should not be left idle
- Long memory latency to memory on cache miss?
- Hardware switches threads to bring in other useful work while waiting for cache miss
- Cost of thread context switch must be much less than cache miss latency
- Put in redundant hardware so don't have to save context on every thread switch:
  - PC, Registers, L1 caches?
- Attractive for apps with abundant TLP
  - Commercial multi-user workloads

10/15/10

Fall 2010 - Lecture #19

11

## Ultrasparc T1 Die Photo



Reuse FPU, L2 caches

Features:

- 8 64-bit Multithreaded SPARC Cores
- Shared 3 MB, 12-way 64B line writeback L2 Cache
- 16 KB, 4-way 32B line ICache per Core
- 8 KB, 4-way 16B line write-through DCache per Core
- 4 144-bit DDR-2 channels
- 3.2 GB/sec JBUS I/O

Technology:

- T1's 90nm CMOS Process
- 9LM Cu Interconnect
- 63 Watts @ 1.2GHz/1.2V
- Die Size: 379mm<sup>2</sup>
- 279M Transistors
- Flip-chip ceramic LGA

## Machines in 61C Lab

```

• /usr/sbin/sysctl -a | grep hw\
hw.model = MacPro4,1    hw.cachelinesize = 64
...
hw.physicalcpu: 8      hw.l1cachesize: 32,768
hw.logicalcpu: 16     hw.l1dcachesize: 32,768
...
hw.l2cachesize: 262,144
hw.l3cachesize: 8,388,608
hw.cpubfrequency =
  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

© Shen, Upastil

13

## Agenda

- Review
- Data Races and Synchronization
- Atomic Operations in MIPS
- Hardware Support for Multithreading
- Administrivia
- Technology Break
- OpenMP with Examples
- Summary

10/15/10

Fall 2010 -- Lecture #19

14

## Administrivia

- Midterm answers and grading rubric online
- Turn in your written regrade petitions with your exam to your TA by next Tuesday Oct 19
- Make sure all grades are correct but Project 4, Final by December 1
- Final Exam 8-11AM (TBD) Monday Dec 13

10/15/10

Fall 2010 -- Lecture #19

15

## 61C In The News

- "Intel CEO Tells Employees That Mobile Effort Will Be 'Marathon'" Bloomberg News/SF Chron, 10/15/10  
Intel Corp. Chief Executive Officer Paul Otellini, working to get his company's chips into tablets and mobile phones, told employees that the effort will be a "marathon, not a sprint."  
"The big question on many people's minds is how will we respond to new computing categories where we currently have little presence, specifically tablets and smartphones," he said yesterday in an e-mail to Intel workers obtained by Bloomberg News. "Winning an architectural contest can take time."

10/15/10

Fall 2010 -- Lecture #19

16

## OpenMP

- OpenMP is an API used for multi-threaded, shared memory parallelism
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Portable, Standardized
- Works for C, C++, Fortran
- gcc -fopenmp file.c

10/15/10

Fall 2010 -- Lecture #19

17

## 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

10/15/10

Fall 2010 -- Lecture #19

18

## 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

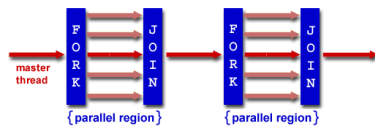


Image courtesy of <http://www.llnl.gov/computing/tutorials/openMP/>

10/15/10

Fall 2010 -- Lecture #19

19

## 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 cores in HW running program

10/15/10

Fall 2010 -- Lecture #19

20

## OMP\_NUM\_THREADS

- Shell command to set number threads:  
`export OMP_NUM_THREADS=x`
- Shell command check number threads:  
`echo $OMP_NUM_THREADS`
- OpenMP intrinsic to get number of threads:  
`int num_th = omp_get_num_threads();`
- OpenMP intrinsic to get Thread ID number:  
`int th_ID = omp_get_thread_num();`

10/15/10 Fall 2010 -- Lecture #19 21

## Invoking Parallel Threads

```
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  foo(ID);
}
```

- Each thread executes a copy of the within the structured structured block

10/15/10 Fall 2010 -- Lecture #19 22

## OpenMP Critical Section

```
float res;
#pragma omp parallel
{ float B;int i, id, nthrds;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads();
  for(i=id;i<niters;i+nthrds){
    B = big_job(i);
    #pragma omp critical
    consume (B, res);
  }
}
```

Threads wait their turn – only one at a time calls consume()

10/15/10 Fall 2010 -- Lecture #19 23

## Shared vs. Private Variables

- OpenMP default is shared variables
- To make private, need to declare with pragma  
`#pragma omp parallel private (x)`
- Now, some examples in OpenMP

10/15/10 Fall 2010 -- Lecture #19 24

## Calculating $\pi$

### Numerical Integration

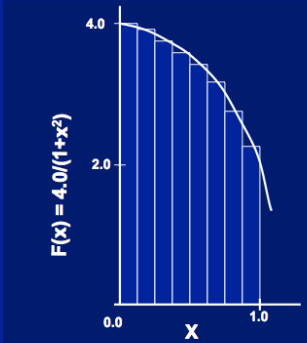
Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^N F(x_i)\Delta x \approx \pi$$

Where each rectangle has width  $\Delta x$  and height  $F(x_i)$  at the middle of interval  $i$ .



## Sequential Calculation of $\pi$ in C

```
#include <stdio.h> /* Serial Code */
static long num_steps = 100000; double step;
void main ()
{ int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=1; i<= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step/num_steps; printf ("pi = %6.12f\n", pi);
}
```

10/15/10

Fall 2010 -- Lecture #19

26

## OpenMP Version (with bug)

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{ int i; double x, pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  #pragma omp parallel private (x)
  { int id = omp_get_thread_num();
    for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS) {
      x = (i+0.5)*step;
      sum[id] += 4.0/(1.0+x*x);
    }
  }
  for(i=0, pi=0.0; i<NUM_THREADS; i++)
    pi += sum[i];
  printf ("pi = %6.12f\n", pi / num_steps);
}
```

10/15/10

Fall 2010 -- Lecture #19

27

## OpenMP Version 2 (with bug)

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{ int i; double x, sum, pi=0.0;
  step = 1.0/(double) num_steps;
  #pragma omp parallel private (x, sum)
  { int id = omp_get_thread_num();
    for (i=id, sum=0.0; i< num_steps; i=i+NUM_THREADS){
      x = (i+0.5)*step;
      sum += 4.0/(1.0+x*x);
    }
  }
  #pragma omp critical
  pi += sum;
  printf ("pi = %6.12f\n", pi/num_steps);
}
```

10/15/10

Fall 2010 -- Lecture #19

28

## 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

10/15/10

Fall 2010 -- Lecture #19

29

## 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 which is *private* per thread (Why?)
- Implicit synchronization at end of for loop
- Divide index regions sequentially per thread
  - Thread 0 gets 0, 1, ..., (max/n)-1;
  - Thread 1 gets max/n, max/n+1, ..., 2\*(max/n)-1
  - Why?

10/15/10

Fall 2010 -- Lecture #19

30

## 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: reduction(operation:var) where
  - *Operation*: operator to perform on the variables (var) at the end of the parallel region
  - *Var*: One or more variables on which to perform scalar reduction.

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

10/15/10

Fall 2010 -- Lecture #19

31

## OpenMP Version 3

```
#include <omp.h>
#include <stdio.h>
/static long num_steps = 100000;
double step;
void main ()
{ int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  #pragma omp parallel for private(x) reduction(+:sum)
  for (i=1; i<= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = sum / num_steps;
  printf ("pi = %6.12f\n", pi);
}
```

10/15/10

Fall 2010 -- Lecture #19

32



## Summary

- Synchronization requires atomic operations
  - Via Load Linked and Store Conditional in MIPS
- Hardware multithreading to get more utilization from processor
- OpenMP is a simple pragma extension to C
  - Threads, Parallel for, private, critical sections, ...
- Data races lead to subtle parallel bugs
  - Beware private variables vs. shared variables