# CS 61C: Great Ideas in Computer Architecture
## *OpenMP, Part II*

Instructor:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12



## New-School Machine Structures (It's a bit more complicated!)

*Software* | *Hardware*

- 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 @ one time
- Programming Languages

*Harness Parallelism & Achieve High Performance*

Warehouse Scale Computer; Smart Phone; Computer; Core ... Core; Memory (Cache); Input/Output; Core; Instruction Unit(s); Functional Unit(s); $A_0+B_0$ $A_1+B_1$ $A_2+B_2$ $A_3+B_3$; Cache Memory; Logic Gates; Project 3



## Review

- OpenMP as simple parallel extension to C
  - Directives: #pragma parallel for, private, reduction…
  - Runtime Library #include <omp.h>; omp_ funcs
  - Environment variables NUM_THREADS
- OpenMP ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble



## OpenMP Timing

- omp_get_wtime – Elapsed wall-clock time

```
#include <omp.h> // to get function
double omp_get_wtime(void);
```

- Elapsed wall-clock time in seconds. The time is measured per thread, no guarantee can be made that two distinct threads measure the same time.
- Time is measured from some "time in the past". On POSIX-compliant systems the seconds since the Epoch (00:00:00 UTC, January 1, 1970) are returned.



## Matrix Multiply in OpenMP

```
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, i, j, k)
  for (i=0; i<Ndim; i++){
   for (j=0; j<Mdim; j++){
     tmp = 0.0;
     for(k=0;k<Pdim;k++){
        /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
        tmp += *(A+(i*Ndim+k)) *  *(B+(k*Pdim+j));
     }
     *(C+(i*Ndim+j)) = tmp;
   }
  }
run_time = omp_get_wtime() - start_time;
```

*Note: Outer loop index i is private by default. Written explicitly here for clarity*

*Note: Outer loop spread across N threads; inner loops inside a thread*



## Notes on Matrix Multiply Example

More performance optimizations available

- Higher compiler optimization (-O2) to reduce number of instructions executed
- Cache blocking to improve memory performance
- Using SIMD SSE3 Instructions to improve floating-point computation rate

## Description of 32-Core System

- Intel Nehalem Xeon 7550
- HW Multithreading: 2 Threads / core
- 8 cores / chip
- 4 chips / board

⇒ 64 Threads / system

- 2.00 GHz
- 256 KB L2 cache/ core
- 18 MB (!) shared L3 cache / chip



## Experiment

- Try compile and run at NUM_THREADS = 64
- Try compile and run at NUM_THREADS = 64 with –O2
- Try compile and run at NUM_THREADS = 32, 16, 8, … with –O2



## Review: Strong vs Weak Scaling

- Strong scaling: problem size fixed
- Weak scaling: problem size proportional to increase in number of processors
  - Speedup on multiprocessor while keeping problem size fixed is harder than speedup by increasing the size of the problem
  - But a natural use of a lot more performance is to solve a lot bigger problem



## 32 Core: Speed-up vs. Scale-up

Speed-up | Scale-up: Fl. Pt. Ops = 2 x Size$^3$

| Threads | Time | Speedup | Time | Size (Dim) | Fl. Ops x 10^9 |
|---|---|---|---|---|---|
| 1 | 13.75 | 1.00 | 13.75 | 1000 | 2.00 |
| 2 | | | 13.52 | 1240 | 3.81 |
| 4 | | | 13.79 | 1430 | 5.85 |
| 8 | | | 12.55 | 1600 | 8.19 |
| 16 | | | 13.61 | 2000 | 16.00 |
| 32 | | | 13.92 | 2500 | 31.25 |
| 64 | | | 13.83 | 2600 | 35.15 |

Memory Capacity = f(Size$^2$), Compute = f(Size$^3$)



## 32 Core: Speed-up vs. Scale-up

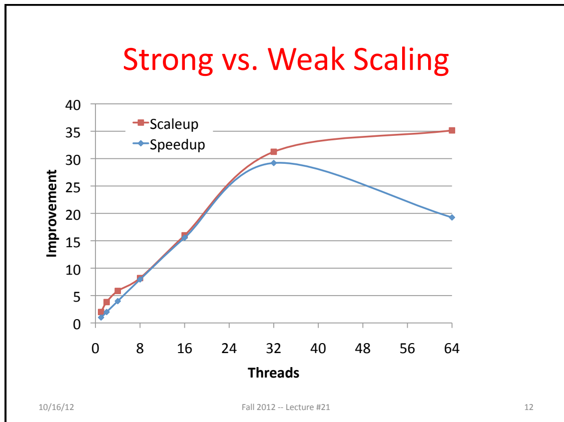
Speed-up | Scale-up: Fl. Pt. Ops = 2 x Size$^3$

| Threads | Time (secs) | Speedup | Time (secs) | Size (Dim) | Fl. Ops x 10^9 |
|---|---|---|---|---|---|
| 1 | 13.75 | 1.00 | 13.75 | 1000 | 2.00 |
| 2 | 6.88 | 2.00 | 13.52 | 1240 | 3.81 |
| 4 | 3.45 | 3.98 | 13.79 | 1430 | 5.85 |
| 8 | 1.73 | 7.94 | 12.55 | 1600 | 8.19 |
| 16 | 0.88 | 15.56 | 13.61 | 2000 | 16.00 |
| 32 | 0.47 | 29.20 | 13.92 | 2500 | 31.25 |
| 64 | 0.71 | 19.26 | 13.83 | 2600 | 35.15 |

Memory Capacity = f(Size$^2$), Compute = f(Size$^3$)



## Strong vs. Weak Scaling



## Peer Instruction: Why Multicore?

The switch in ~ 2004 from 1 processor per chip to multiple processors per chip happened because:

I. The "power wall" meant that no longer get speed via higher clock rates and higher power per chip
II. There was no other performance option but replacing 1 inefficient processor with multiple efficient processors
III. OpenMP was a breakthrough in ~2000 that made parallel programming easy

A)(orange) I only
B)(green) II only
C)(pink) I & II only
D)(yellow) I, II, & III



## Peer Instruction: Why Multicore?

The switch in ~ 2004 from 1 processor per chip to multiple processors per chip happened because:

I. The "power wall" meant that no longer get speed via higher clock rates and higher power per chip
II. There was no other performance option but replacing 1 inefficient processor with multiple efficient processors
III. OpenMP was a breakthrough in ~2000 that made parallel programming easy

A)(orange) I only
B)(green) II only
C)(pink) I & II only
D)(yellow) I, II, & III



## False Sharing in OpenMP

```
{   int i;    double  x, pi, sum[NUM_THREADS];
#pragma omp parallel private (i, 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);
    }
```

- What is problem?
- Sum[0] is 8 bytes in memory, Sum[1] is adjacent 8 bytes in memory => false sharing if block size > 8 bytes



## Peer Instruction: No False Sharing

```
{   int i;    double  x, pi, sum[10000];
#pragma omp parallel private (i, x)
{   int id = omp_get_thread_num(),  fix = __________;
    for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS) {
        x = (i+0.5)*step;
        sum[id*fix] += 4.0/(1.0+x*x);
    }
```

- What is best value to set `fix` to prevent false sharing?

A)(orange) `omp_get_num_threads();`
B)(green) Constant for number of blocks in cache
C)(pink) Constant for size of block in bytes
D)(yellow) Constant for size of blocks in doubles



## Peer Instruction: No False Sharing

```
{   int i;    double  x, pi, sum[10000];
#pragma omp parallel private (i,x)
{   int id = omp_get_thread_num(),  fix = __________;
    for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS) {
        x = (i+0.5)*step;
        sum[id*fix] += 4.0/(1.0+x*x);
    }
```

- What is best value to set `fix` to prevent false sharing?

A)(orange) `omp_get_num_threads();`
B)(green) Constant for number of blocks in cache
C)(pink) Constant for size of block in bytes
D)(yellow) Constant for size of blocks in doubles



## Administrivia

- Lab 8, Data-Level Parallelism
  - Start this early, because you'll need the SSE experience for project 3a
- Project 3a, optimizing matrix manipulation code
  - Part a : just single-thread optimizations – DON'T USE OpenMP
  - (Part b: next week, use OpenMP too)
- HW#5

## Midterm Regrade Request Policy

- NO REQUESTS ACCEPTED UNTIL LECTURE WED OCTOBER 17, i.e., we'll simply delete any that come before
- Must attend discussion section to learn solutions and grading process – TA signoff needed for regrade request!
- Regrade requests must be accompanied by written request explaining rationale for regrade.
- Modifying your copy of exam punishable by F and letter in your University record
- We reserve right to regrade whole exam



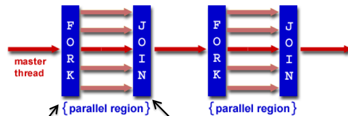
## Types of Synchronization

- Parallel threads run at varying speeds, need to synchronize their execution when accessing shared data.
- Two basic classes of synchronization:
  - Producer-Consumer
    - Consumer thread(s) wait(s) for producer thread(s) to produce needed data
    - Deterministic ordering. Consumer always runs after producer (unless there's a bug!)
  - Mutual Exclusion
    - Any thread can touch the data, but only one at a time.
    - Non-deterministic ordering. Multiple orders of execution are valid.



## Simple OpenMP Parallel Sections

- OpenMP Fork and Join are examples of producer-consumer synchronization




*Master doesn't fork worker threads until data is ready for them*

*At join, have to wait for all workers to finish at a "barrier" before starting following sequential master thread*

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



## Barrier Synchronization

- Barrier waits for all threads to complete a parallel section. Very common in parallel processing.
- How does OpenMP implement this?



## Barrier: First Attempt (pseudo-code)

```
int n_working = NUM_THREADS; /* Shared variable*/
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  foo(ID); /* Do my chunk of work. */

  /* Barrier code. */
  n_working -= 1; /* I'm done */
  if (ID == 0) { /* Master */
    while (n_working != 0)
      ; /* master spins until everyone finished */
  } else {
     /* Put thread to sleep if not master */
  };
}
```



## Flashcard quiz: Implementing Barrier Count decrement

- Thread #1

```
/* n_working -= 1 */
lw $t0, ($s0)
addiu $t0, -1
sw $t0, ($s0)
```

- Thread #2

```
/* n_working -=1 */
lw $t0, ($s0)
addiu $t0, -1
sw $t0, ($s0)
```

If initially **n_working** = 5, what are possible final values after both threads finish above code sequence?

- n_working = 3 only
- n_working = 3, or n_working = 4 only
- n_working = 3, 4, or 5 only
- Undefined

## Review: 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



## Decrement of Barrier Variable is Example of Mutual Exclusion

- Want each thread to *atomically* decrement the **n_working** variable
  - Atomic from Greek "Atomos" meaning indivisible!
- Ideally want:
  - Begin atomic section /*Only one thread at a time*/

```
lw $t0, ($s0)
addiu $t0, -1
sw $t0, ($s0)
```

  - End atomic section/*Allow another thread in */



## New Hardware Instructions

For some common useful cases, some instruction sets have special instructions that atomically read-modify-write a memory location

Example:

```
fetch-and-add r_dest, (r_address), r_val implemented as:
  r_dest = Mem[r_address] //Return old value in register
  t = r_dest + r_val      // Updated value
  Mem[r_address] = t      //Increment value in memory
```

Simple common variant: `test-and-set r_dest, (r_address)`

Atomically reads old value of memory into r_dest, and puts 1 into memory location. Used to implement *locks*



## Use locks for more general atomic sections

Atomic sections commonly called "critical sections"

Acquire(lock) /* Only one thread at a time in section. */
  /* Critical Section Code */
Release(lock) /* Allow other threads into section. */

- A lock is a variable in memory (one word)
- Hardware atomic instruction, e.g., test-and-set, checks and sets lock in memory



## Implementing Barrier Count decrement with locks

```
/* Acquire lock */
spin:
testandset $t0, ($s1) /* $s1 has lock address */
bnez $t0, spin

lw $t0, ($s0)
addiu $t0, -1
sw $t0, ($s0)

/* Release lock */
sw $zero, ($s1)  /*Regular store releases lock*/
```



## MIPS Atomic Instructions

- Splits atomic into two parts:
  - Load Linked `LL rt, offset(base)`
    - Regular load that "reserves" an address
  - Store Conditional `SC rt, offset(base)`
    - Store that only happens if no other hardware thread touched the reserved address
    - Success: rt=1 and memory updated
    - Failure: rt = 0 and memory unchanged
- Can implement test-and-set or fetch-and-add as short code sequence
- Reuses cache snooping hardware to check if other processors touch reserved memory location

## ISA Synchronization Support

- All have some atomic Read-Modify-Write instruction
- Varies greatly – little agreement on "correct" way to do this
- No commercial ISA has direct support for producer-consumer synchronization
  - Use mutual exclusion plus software to get same effect (e.g., barrier in OpenMP)
- This area is still very much "work-in-progress" in computer architecture



## OpenMP Critical Sections

```
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  foo(ID); /* Do my chunk of work. */

#pragma omp critical
  { /* Only one thread at a time */
     /* shared_variable_updates */
  }
}
```



## And in Conclusion, …

- MatrixMultiply speedup versus scaleup
  - Strong versus weak scaling
- Synchronization:
  - Producer-consumer versus mutual-exclusion
- Hardware provides some atomic instructions
  - Software builds up other synchronization using these