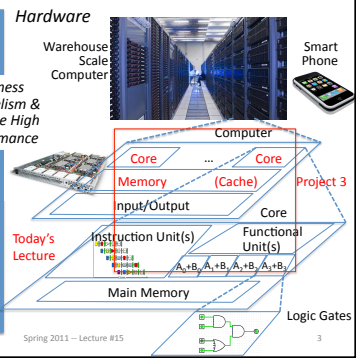
CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Thread Level Parallelism

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp11

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

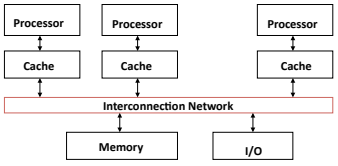
- Multiprocessor Systems
- Administrivia
- Multiprocessor Cache Consistency
- Synchronization
- Technology Break
- OpenMP Introduction
- Summary

Agenda

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

Parallel Processing: Multiprocessor Systems (MIMD)

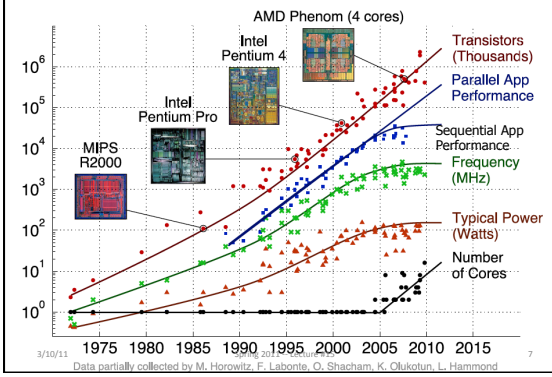
- Multiprocessor (MIMD): a computer system with at least 2 processors



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

Transition to Multicore



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

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, …
- 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!)

Year	Cores	SIMD bits /Core	Core * SIMD bits	Peak DP GFLOPs
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

Three Key Questions about Multiprocessors

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

Three Key Questions about Multiprocessors

- Q1 – How do they share data?
- Single address space shared by all processors/ cores

Three Key Questions about Multiprocessors

- 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
- *All* multicore computers today are Shared Memory Multiprocessors (SMPs)

Example: Sum Reduction

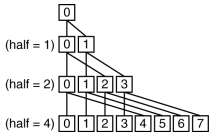
- Sum 100,000 numbers on 100 processor SMP
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor:

```
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

This code executes simultaneously in P0, P1, …, P7




```
half = 8;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets extra element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

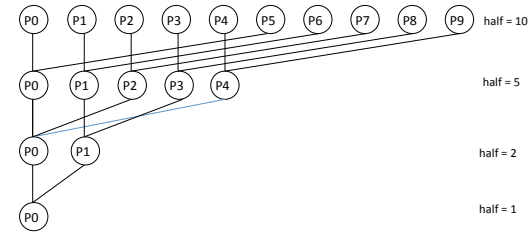

Student Roulette?

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]

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]

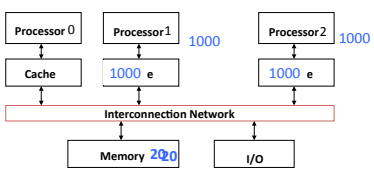


Three Key Questions about Multiprocessors

- 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

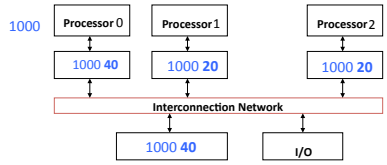
- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



Student Roulette?

Shared Memory and Caches

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



Processor 0 *Write Invalidates* Other Copies

Student Roulette?

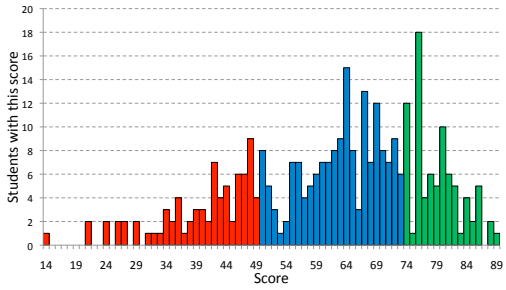
Agenda

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

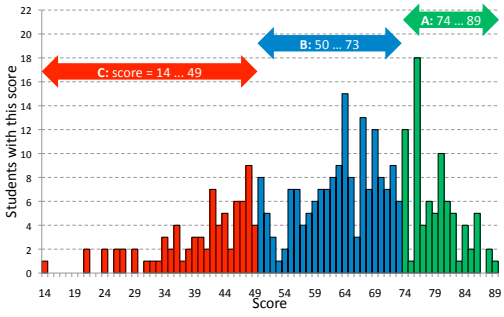
Course Organization

- 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)
  - Extra Credit: Matrix Multiply Competition, anything goes
  - *Midterm (10%): 6-9 PM Tuesday March 8*
  - Final (20%): 11:30-2:30 PM Monday May 9

Midterm Results



Midterm Results



EECS Grading Policy

- http://www.eecs.berkeley.edu/Policies/ugrad.grading.shtml
  "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)

	Fall	Spring
2010	2.81	2.81
2009	2.71	2.81
2008	2.95	2.74
2007	2.67	2.76

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

Administrivia

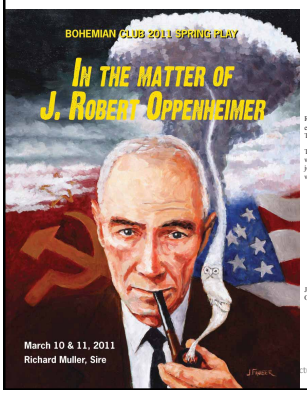
- Next Lab and Project
  - Lab #8: Data Level Parallelism, posted
  - Project #3: Matrix Multiply Performance Improvement
    - Work in groups of two!
    - Part 1: March 27 (end of Spring Break)
    - Part 2: April 3
  - HW #5 also due March 27
    - Posted soon

CS 61c in the News

TOP UNIVERSITIES BY REPUTATION 2011

REPUTATION RANK	INSTITUTION	COUNTRY / REGION	REPUTATION change
1	Harvard University	United States	100.0
2	Massachusetts Institute of Technology	United States	85.0
3	University of Cambridge	United Kingdom	80.7
4	University of California Berkeley	United States	74.7
5	Stanford University	United States	71.5
6	University of Oxford	United Kingdom	68.6
7	Princeton University	United States	36.6
8	University of Tokyo	Japan	33.2
9	Yale University	United States	28.3
10	California Institute of Technology	United States	23.5

See http://www.timeshighereducation.co.uk/world-university-rankings/2010-2011/reputation-rankings.html



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"
-- Edward Teller, "The father of the hydrogen bomb."

"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

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 $ 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

Two 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)

Snooping/Snoopy Protocols
e.g., the Berkeley Ownership Protocol
See http://en.wikipedia.org/wiki/Cache_coherence
Berkeley Protocol is a wikipedia stub!

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 (*critical section*) where only one thread 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 set / closed / locked / lock on

```
Set the lock
Critical section
  (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):

```
        addiu $t1,$zero,1  ; t1 = Locked value
Loop:   lw $t0,lock($s0)   ; load lock
        bne $t0,$zero,Loop ; loop if locked
Lock:   sw $t1,lock($s0)   ; Unlocked, so lock
```

- Unlock:

```
sw $zero,lock($s0)
```

- Any problems with this?

Student Roulette?

Possible Lock Problem

- Thread 1

```
      addiu $t1,$zero,1
Loop: lw $t0,lock($s0)

      bne $t0,$zero,Loop

Lock: sw $t1,lock($s0)
```

- Thread 2

```
      addiu $t1,$zero,1
Loop: lw $t0,lock($s0)

      bne $t0,$zero,Loop

Lock: sw $t1,lock($s0)
```

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 ↔ memory
  - Or an atomic pair of instructions

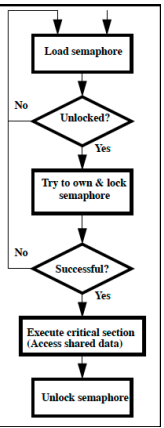
Synchronization in MIPS

- Load linked: `ll rt,offset(rs)`
- Store conditional: `sc rt,offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt (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: $s4 ←→ ($s1)

```
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
```


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
  - No other instruction can modify the memory location, including another Test-and-Set instruction
- Useful for implementing lock operations



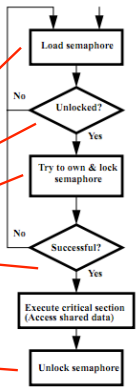
Test-and-Set in MIPS

- Example: MIPS sequence for implementing a T&S at ($s1)

```
Try: addiu $t0,$zero,1
     ll  $t1,0($s1)
     bne $t1,$zero,Try
     sc  $t0,0($s1)
     beq $t0,$zero,try
Locked:

     critical section

     sw $zero,0($s1)
```



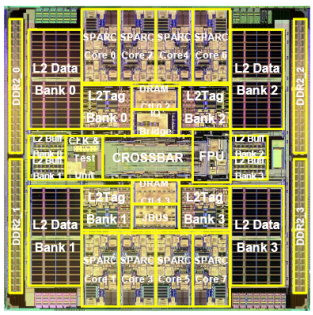
Agenda

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

Agenda

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

Ultrasparc T1 Die Photo



Reuse FPUs, 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:

- TI's 90nm CMOS Process
- 9LM Cu Interconnect
- 63 Watts @ 1.2GHz/1.2V
- Die Size: 379mm²
- 279M Transistors
- Flip-chip ceramic LGA

Machines in 61C Lab

- `/usr/sbin/sysctl -a | grep hw\.`

hw.model = MacPro4,1
…
hw.physicalcpu: 8
hw.logicalcpu: 16
…
hw.cpufrequency = 2,260,000,000
hw.physmem = 2,147,483,648

hw.cachelinesize = 64
hw.l1icachesize: 32,768
hw.l1dcachesize: 32,768
hw.l2cachesize: 262,144
hw.l3cachesize: 8,388,608

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

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

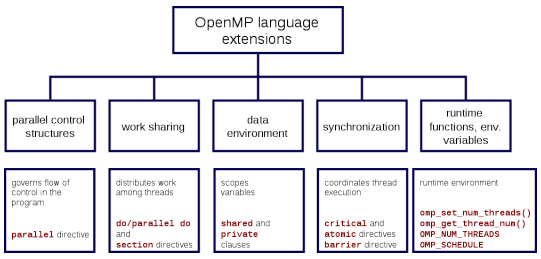
hw.cachelinesize = 64
hw.l1icachesize = 32768
hw.l1dcachesize = 32768
hw.l2cachesize = 3,145,728

No l3 cache
Dual core
One hw context per core

OpenMP

- OpenMP is an API used for multi-threaded, shared memory parallelism
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Portable
- Standardized
- See http://computing.llnl.gov/tutorials/openMP/

OpenMP

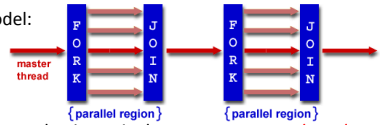


OpenMP Programming Model

- 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 Programming Model

- Fork - Join Model:



- OpenMP programs begin as single process: *master thread*; Executes sequentially until the first parallel region construct is encountered
  - FORK: the master thread then creates a team of parallel threads
  - Statements in program that are enclosed by the parallel region construct are executed in parallel among the various team threads
  - JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

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
- 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
- Usually equals the number of cores in the underlying HW on which the program is run

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:
  `num_th = omp_get_num_threads();`
- OpenMP intrinsic to get Thread ID number:
  `th_ID = omp_get_thread_num();`

Parallel Threads and Scope

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

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

- OpenMP default is shared variables
- To make private, need to declare with pragma

```
#pragma omp parallel private (x)
```

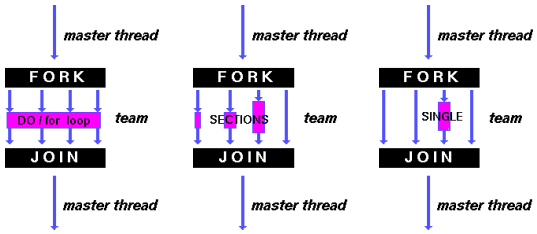

Hello World in OpenMP

```
#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



shares iterations of a loop across the team

each section executed by a separate thread

serializes the execution of a thread

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 */
}
```

Only one thread executes the following code block at a time

Compiler generates necessary lock/unlock code around the increment of x

Student Roulette?

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