

CS 61C: Great Ideas in Computer  
Architecture (Machine Structures)  
*Thread-Level Parallelism (TLP)*  
*and OpenMP*

Instructors:

Nicholas Weaver & Vladimir Stojanovic

<http://inst.eecs.berkeley.edu/~cs61c/>

# Review

- Sequential software is slow software
  - SIMD and MIMD are paths to higher performance
- MIMD thru: multithreading processor cores (increases utilization), Multicore processors (more cores per chip)
- OpenMP as simple parallel extension to C
  - Pragmas for forking multiple Threads
  - $\approx$  C: small so easy to learn, but not very high level and it's easy to get into trouble

# 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 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
- (more later)

# Analogy: Buying Milk


- Your fridge has no milk. You and your roommate will return from classes at some point and check the fridge
- Whoever gets home first will check the fridge, go and buy milk, and return
- What if the other person gets back while the first person is buying milk?
  - You've just bought twice as much milk as you need!
- It would've helped to have left a note...

# Lock Synchronization (1/2)

- Use a “Lock” to grant access to a region (*critical section*) so that only one thread can operate at a time
  - Need all processors to be able to access the lock, so use a location in shared memory as *the lock*
- Processors read lock and either wait (if locked) or set lock and go into critical section
  - **0** means lock is free / open / unlocked / lock off
  - **1** means lock is set / closed / locked / lock on

# Lock Synchronization (2/2)

- Pseudocode:

Check lock  Can loop/idle here  
if locked

Set the lock

Critical section

(e.g. change shared variables)

Unset the lock

# Possible Lock Implementation

- Lock (a.k.a. busy wait)

```
Get_lock:                               # $s0 -> addr of lock
    addiu $t1,$zero,1                    # t1 = Locked value
Loop:   lw $t0,0($s0)                    # load lock
        bne $t0,$zero,Loop              # loop if locked
Lock:   sw $t1,0($s0)                    # Unlocked, so lock
```

- Unlock

```
Unlock:
    sw $zero,0($s0)
```

- Any problems with this?

# Possible Lock Problem

- Thread 1

```
    addiu $t1,$zero,1
Loop: lw $t0,0($s0)

    bne $t0,$zero,Loop

Lock: sw $t1,0($s0)
```

- Thread 2

```
    addiu $t1,$zero,1
Loop: lw $t0,0($s0)

    bne $t0,$zero,Loop

Lock: sw $t1,0($s0)
```



Time

*Both threads think they have set the lock!  
Exclusive access not guaranteed!*



# Hardware Synchronization

- Hardware support required to prevent an interloper (another thread) from changing the value
  - *Atomic* read/write memory operation
  - No other access to the location allowed between the read and write
- How best to implement in software?
  - Single instr? Atomic swap of register  $\leftrightarrow$  memory
  - Pair of instr? One for read, one for write

# Synchronization in MIPS

- *Load linked:* `ll rt, off(rs)`
- *Store conditional:* `sc rt, off(rs)`
  - Returns **1** (success) if location has not changed since the `ll`
  - Returns **0** (failure) if location has changed
- Note that `sc` *clobbers* the register value being stored (`rt`)!
  - Need to have a copy elsewhere if you plan on repeating on failure or using value later

# Synchronization in MIPS Example

- Atomic swap (to test/set lock variable)

Exchange contents of register and memory:

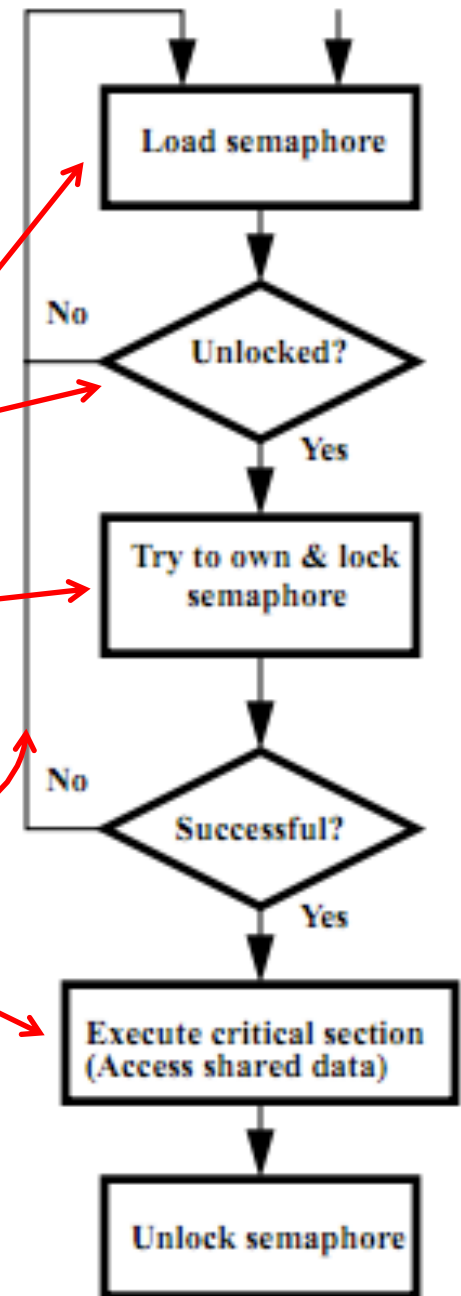
$\$s4 \leftrightarrow \text{Mem}(\$s1)$

```
try: add $t0,$zero,$s4 #copy value
      ll  $t1,0($s1)    #load linked
      sc  $t0,0($s1)    #store conditional
      beq $t0,$zero,try #loop if sc fails
      add $s4,$zero,$t1 #load value in $s4
```

**sc would fail if another threads executes sc here**

# 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
  - While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions
- Useful for implementing lock operations



# Test-and-Set in MIPS

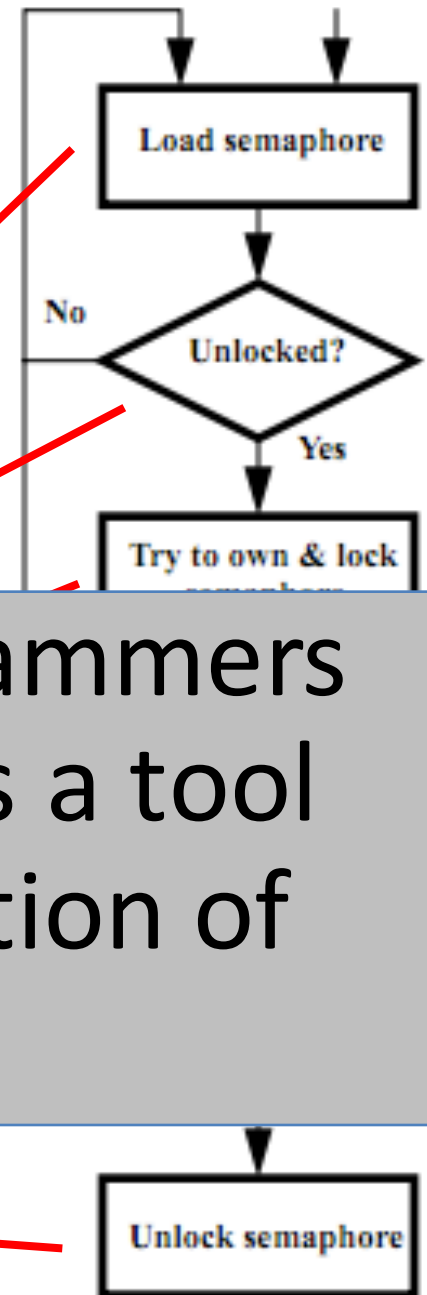
- Example: MIPS sequence for implementing a T&S at (\$s1)

```
Try: addiu $t0,$zero,1  
ll $t0,$s1  
or $t0,$t0,$t0
```

Idea is that not for programmers to use this directly, but as a tool for enabling implementation of parallel libraries

```
UNLOCK:
```

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



**Clickers:** Consider the following code when executed *concurrently* by two threads.

What possible values can result in  $*(\$s0)$ ?

```
# *($s0) = 100
lw    $t0, 0($s0)
addi  $t0, $t0, 1
sw    $t0, 0($s0)
```

**A: 101 or 102**

**B: 100, 101, or 102**

**C: 100 or 101**

**D: 102**

# A related problem: Deadlock

- Consider the following: A dozen lawyers are sitting around a table for dinner
  - Between each lawyer is a chopstick
    - Original version is 'Dining Philosophers' by Dijkstra, but changing to lawyers is a Berkeley innovation...
- Each lawyer grabs the chopstick to the right and then to the left...
  - What if every lawyer only grabs the first chopstick?
- Result is ***deadlock***: each lawyer is waiting on another to release a chopstick

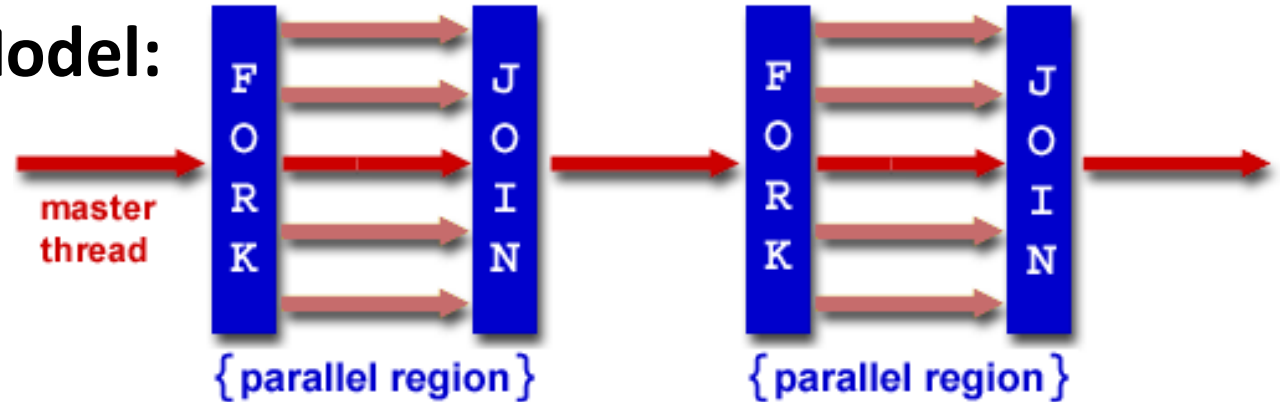
# Solutions for Deadlock...

- Structure your program so it doesn't occur!
  - EG, rather than going “Right then left” go “even then odd”
    - Now the system will always progress
- Or have each lawyer give up after a *random* time, drop the chopstick, *randomly wait* and try again
  - Need randomization to prevent “livelock”
  - Technique used by Ethernet to arbitrate access
- Centrally arbitrate access
  - A waiter tells each lawyer which chopstick to take
  - Which does limit potential parallelism
- Watch for deadlock and respond
  - A waiter is standing by to shoot a lawyer if deadlock occurs...
    - Which is why a long-forgotten Berkeley OS prof changed it to “lawyers”



# OpenMP Programming Model - Review

- **Fork - Join Model:**



- OpenMP programs begin as single process (*master thread*) and executes sequentially until the first parallel region construct is encountered
  - *FORK*: 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 threads
  - *JOIN*: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

# parallel Pragma and Scope - Review

- Basic OpenMP construct for parallelization:

```
#pragma omp parallel
```

```
{ /* code goes here
```

```
    Brackets needed because the  
    pragma applies to a single  
    C statement */
```

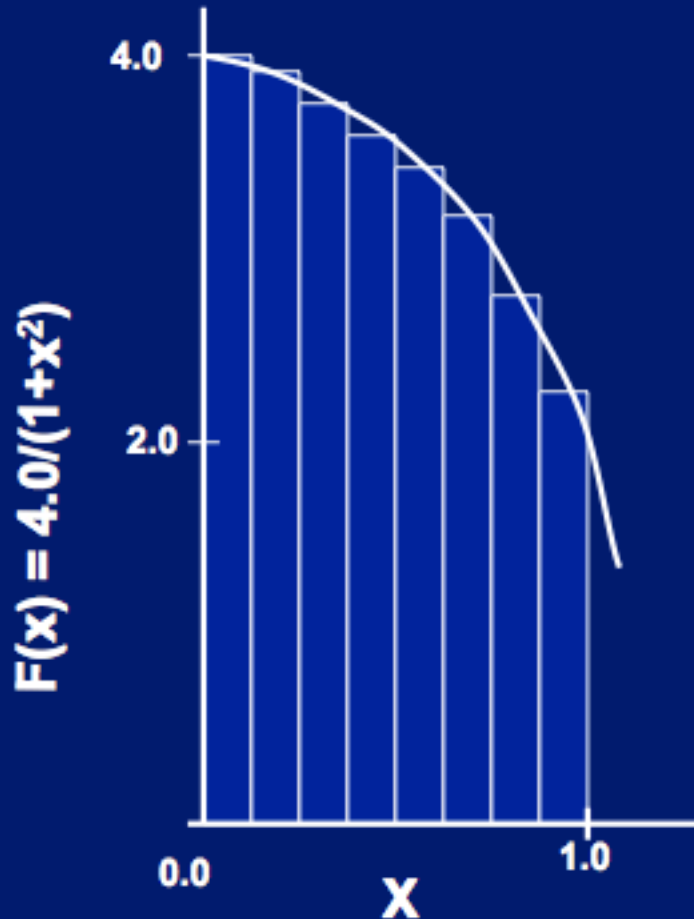
```
}
```

- *Each* thread runs a copy of code within the block
  - Thread scheduling is *non-deterministic*
- OpenMP default is *shared* variables
  - To make private, need to declare with pragma:  

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

# Example: 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 = sum / num_steps;
    printf ("pi = %6.12f\n", pi);
}
```

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

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

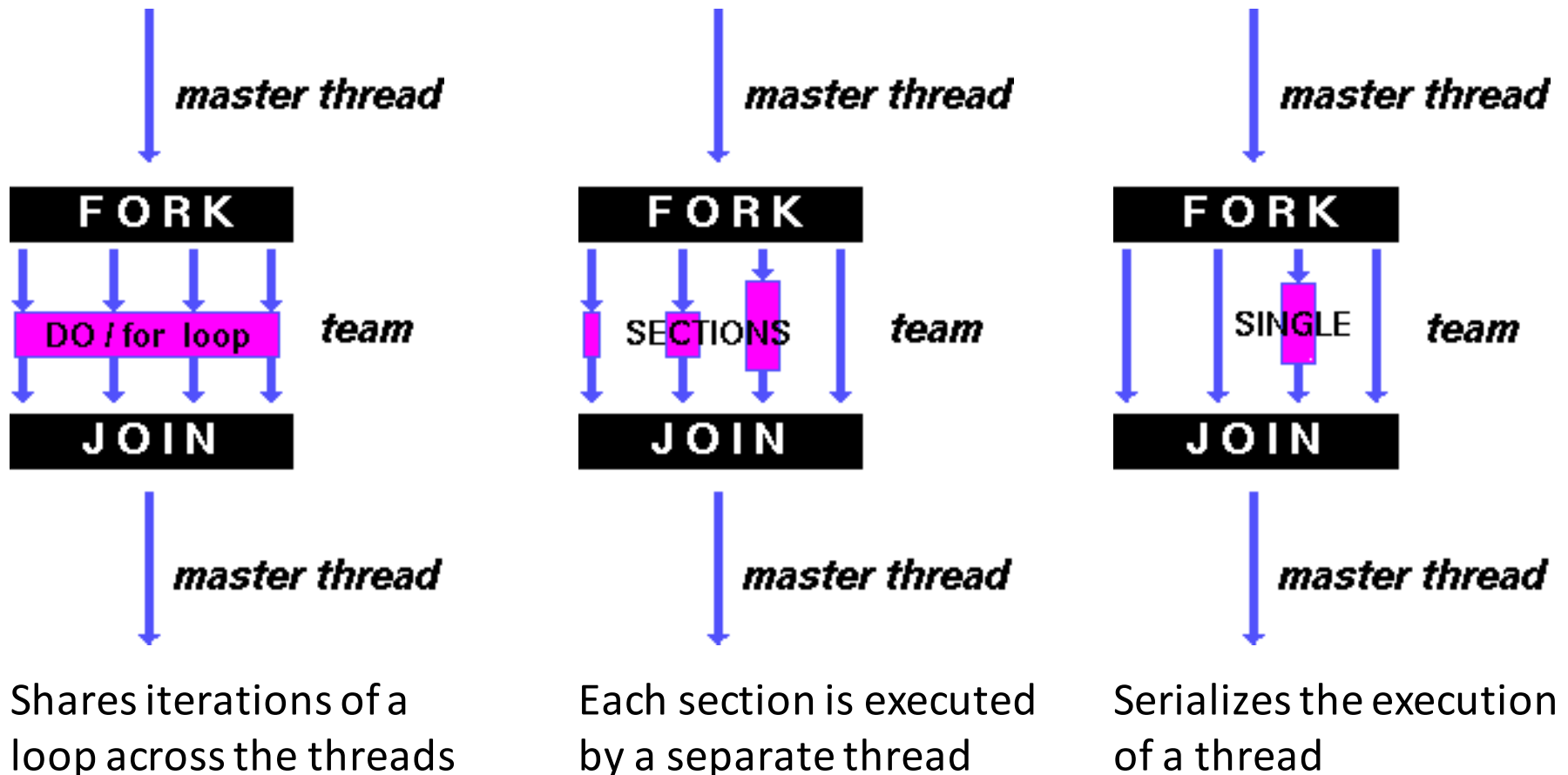
# Parallel OpenMP Version (1)

```
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

void main () {
    int i;          double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #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);
        }
    }
    for(i=1; i<NUM_THREADS; i++)
        sum[0] += sum[i];  pi = sum[0] / num_steps
    printf ("pi = %6.12f\n", pi);
}
```

# OpenMP Directives (Work-Sharing)

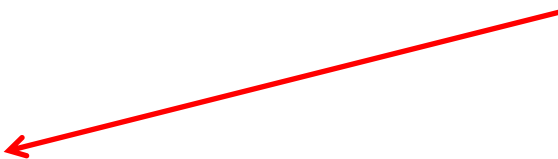
- These are defined *within* a `parallel` section



# Parallel Statement Shorthand

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<len; i++) { ... }
}
```

This is the only  
directive in the  
parallel section




can be shortened to:

```
#pragma omp parallel for
    for (i=0; i<len; i++) { ... }
```

- Also works for sections

# Building Block: `for` loop

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

- Breaks *for loop* into chunks, and allocate each to a separate thread
  - e.g. if `max = 100` with 2 threads:  
assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
  - 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  In general, don't jump outside of any pragma block
  - i.e. No `break`, `return`, `exit`, `goto` statements

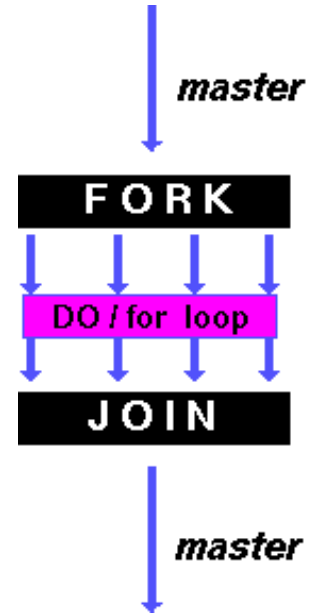


# Parallel `for` *pragma*

```
#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 “barrier” 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?



# OpenMP Timing

- Elapsed wall clock time:

```
double omp_get_wtime(void);
```

- Returns elapsed wall clock time in seconds
- 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,” so subtract results of two calls to `omp_get_wtime` to get elapsed time

# Matrix Multiply in OpenMP

```
// C[M][N] = A[M][P] × B[P][N]
```

```
start_time = omp_get_wtime();
```

```
#pragma omp parallel for private(tmp, j, k)
```

```
for (i=0; i<M; i++){
```

```
    for (j=0; j<N; j++){
```

```
        tmp = 0.0;
```

```
        for( k=0; k<P; k++){
```

```
            /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
```

```
            tmp += A[i][k] * B[k][j];
```

```
        }
```

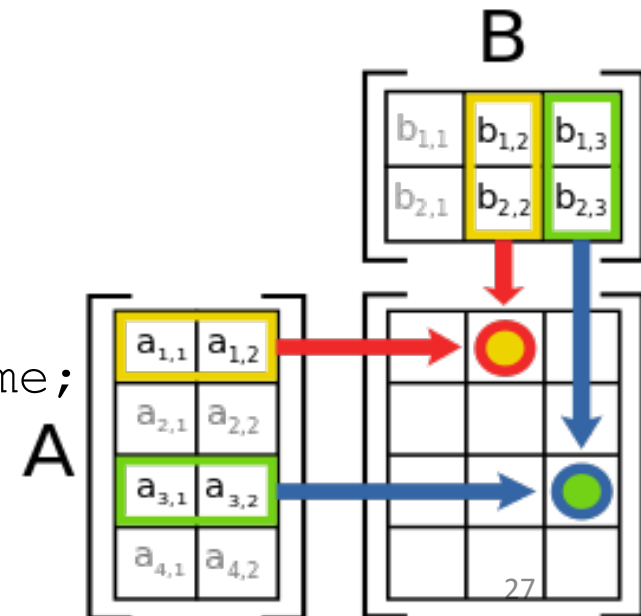
```
        C[i][j] = tmp;
```

```
    }
```

```
}
```

```
run_time = omp_get_wtime() - start_time;
```

← Outer loop spread across N threads;  
inner loops inside a single thread



# Notes on Matrix Multiply Example

- More performance optimizations available:
  - Higher *compiler optimization* (-O2, -O3) to reduce number of instructions executed
  - *Cache blocking* to improve memory performance
    - Take advantage of both spatial and temporal locality
  - Using SIMD SSE instructions to raise floating point computation rate (*DLP*)

# OpenMP Reduction

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp parallel for private ( sum )
for (i = 0; i <= MAX ; i++)
    sum += A[i];
avg = sum/MAX; // bug
```

- *Problem is that we really want sum over all threads!*
- *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.

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp for reduction(+ : sum)
for (i = 0; i <= MAX ; i++)
    sum += A[i];
avg = sum/MAX;
```

# Calculating $\pi$ Version (1) - review

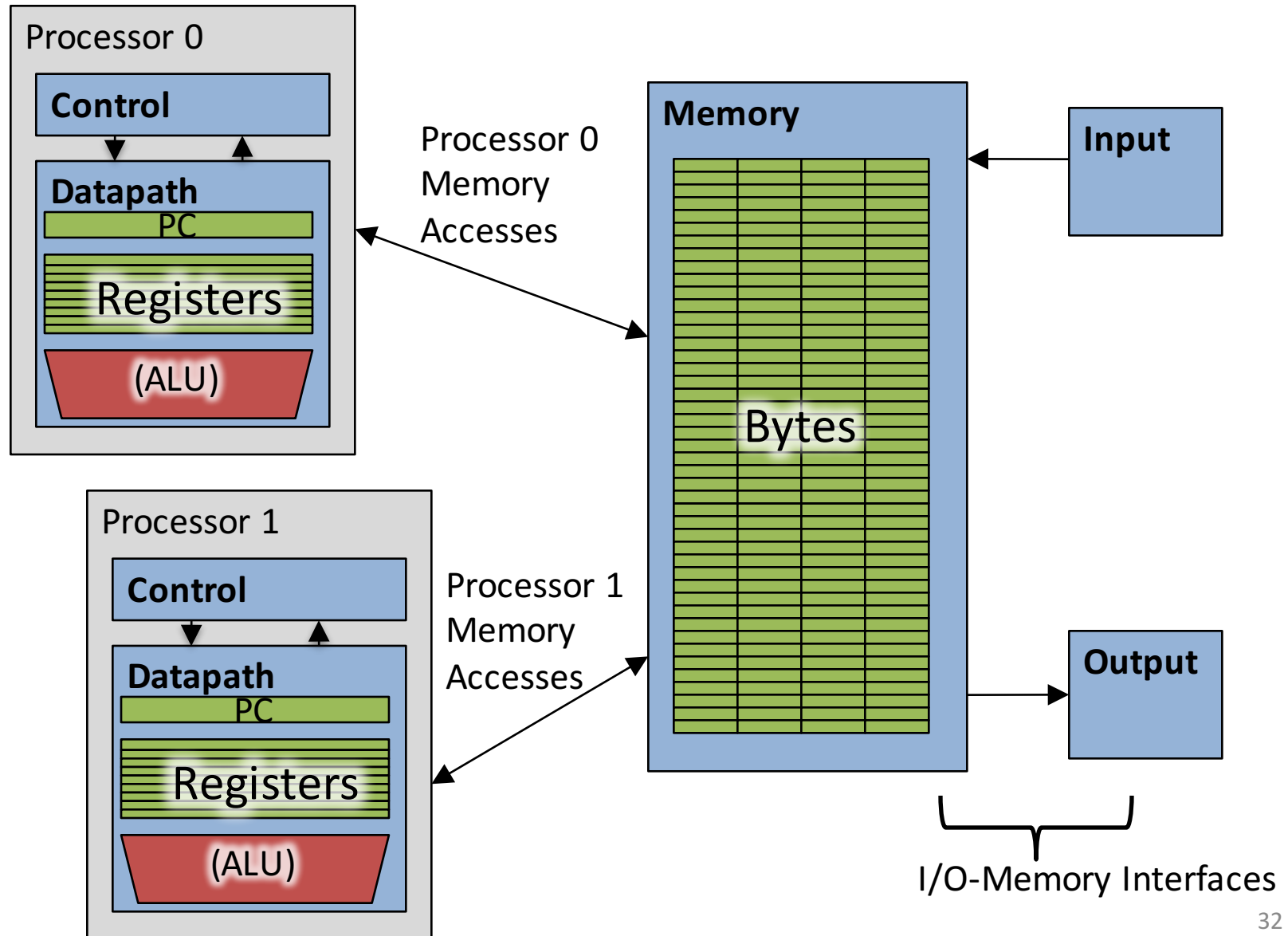
```
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

void main () {
    int i;          double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #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);
        }
    }
    for(i=1; i<NUM_THREADS; i++)
        sum[0] += sum[i];  pi = sum[0] / num_steps
    printf ("pi = %6.12f\n", pi);
}
```

# Version 2: parallel for, reduction

```
#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 = %6.8f\n", pi);
}
```

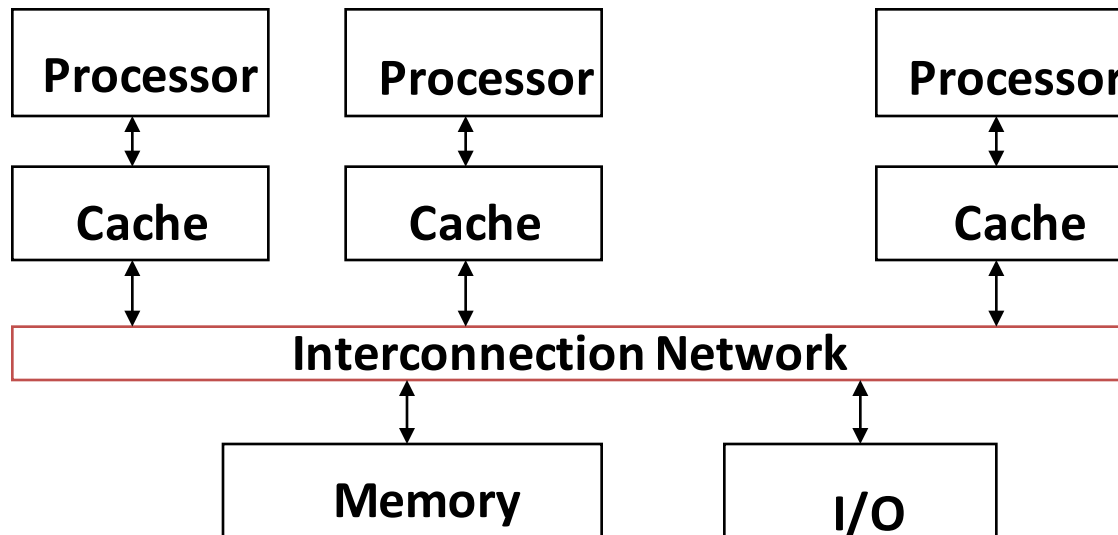
# Simple Multi-core Processor





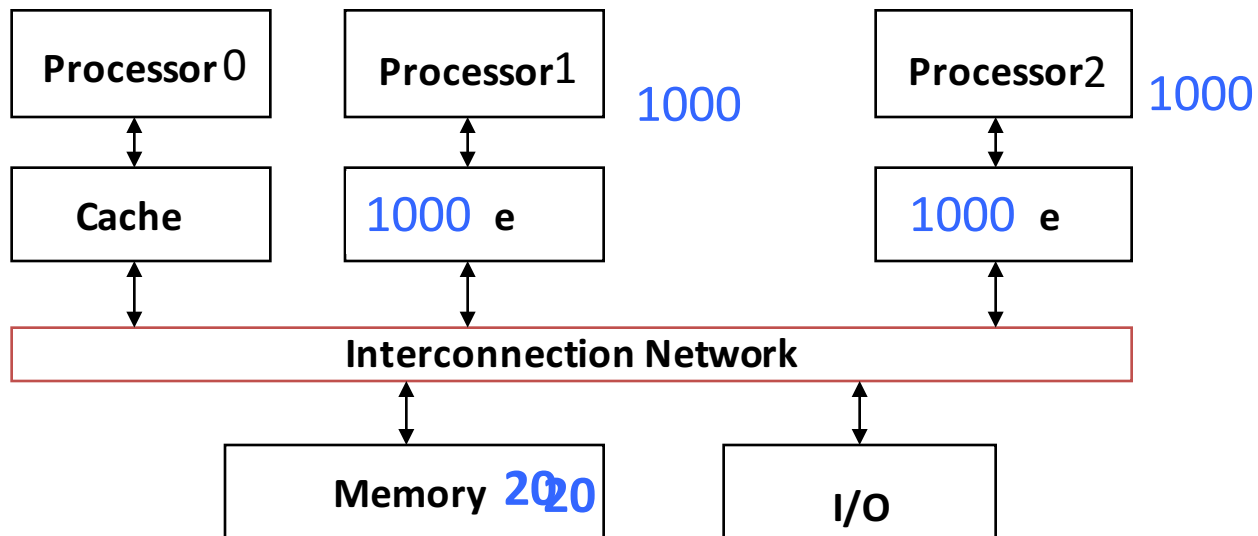
# Multiprocessor Caches

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory



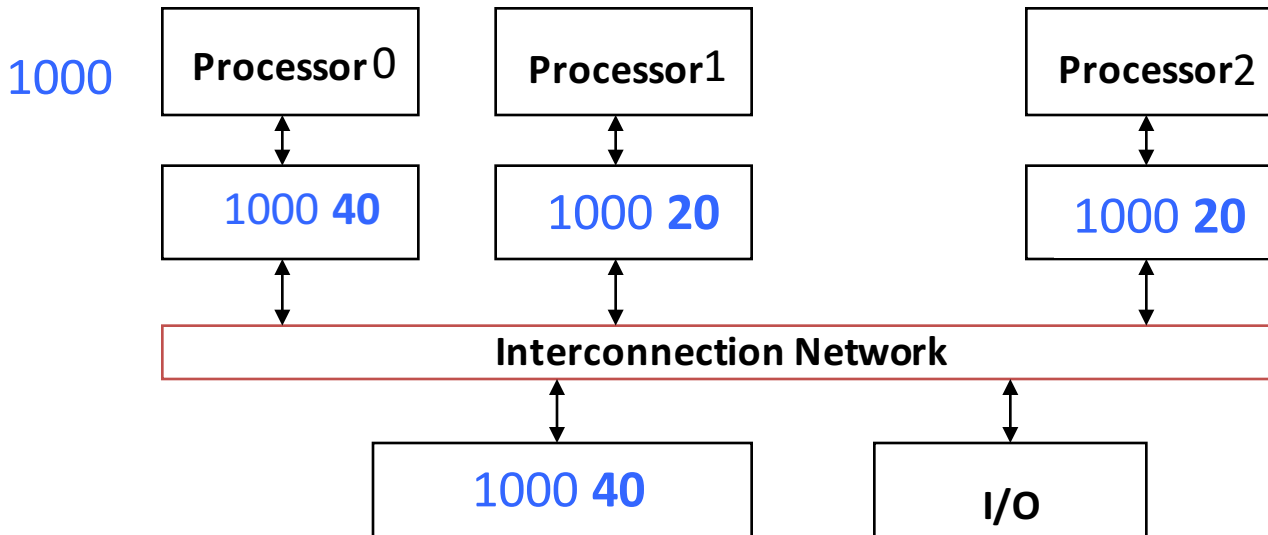
# Shared Memory and Caches

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



# Shared Memory and Caches

- Now:
  - Processor 0 writes Memory[1000] with 40



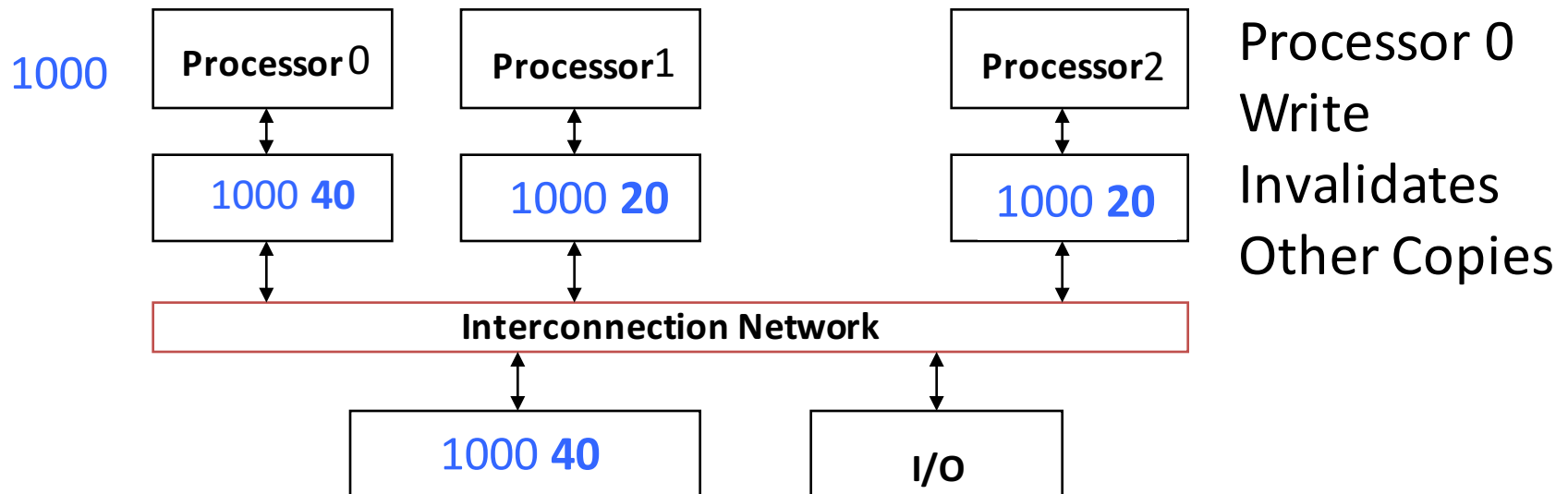
Problem?

# 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 any other copies
- Write transactions from one processor, other caches “snoop” the common interconnect checking for tags they hold
  - Invalidate any copies of same address modified in other cache

# Shared Memory and Caches

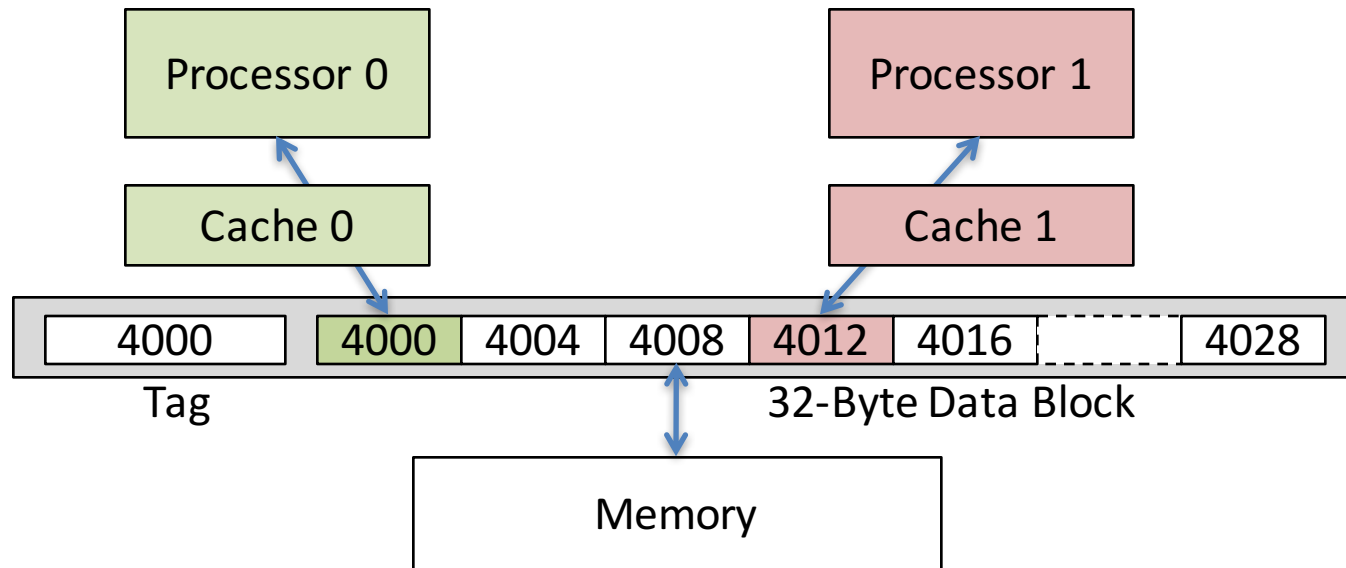
- Example, now with cache coherence
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



# Clickers/Peer Instruction: Which statement is true?

- **A: Using write-through caches removes the need for cache coherence**
- **B: Every processor store instruction must check contents of other caches**
- **C: Most processor load and store accesses only need to check in local private cache**
- **D: Only one processor can cache any memory location at one time**

# Cache Coherency Tracked by Block



- 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?

# Coherency Tracked by Cache Block

- Block ping-pongs between two caches even though processors are accessing disjoint variables
- Effect called *false sharing*
- How can you prevent it?



# Review: Understanding Cache Misses: The 3Cs

- **Compulsory** (cold start or process migration, 1<sup>st</sup> reference):
  - First access to block, impossible to avoid; small effect for long-running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- **Capacity** (not compulsory and...)
  - Cache cannot contain all blocks accessed by the program ***even with perfect replacement policy in fully associative cache***
  - Solution: increase cache size (may increase access time)
- **Conflict** (not compulsory or capacity and...):
  - Multiple memory locations map to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (may increase access time)
  - Solution 3: improve replacement policy, e.g.. LRU

# Fourth “C” of Cache Misses: *Coherence Misses*

- Misses caused by coherence traffic with other processor
- Also known as *communication* misses because represents data moving between processors working together on a parallel program
- For some parallel programs, coherence misses can dominate total misses
  - It gets even more complicated with multithreaded processors: You want separate threads on the same CPU to have common working set, otherwise you get what could be described as *incoherence* misses

# And in Conclusion, ...

- Multiprocessor/Multicore uses Shared Memory
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!
- OpenMP as simple parallel extension to C
  - Threads, Parallel for, private, reductions ...
  - $\approx$  C: small so easy to learn, but not very high level and it's easy to get into trouble
  - Much we didn't cover – including other synchronization mechanisms (locks, etc.)