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

Thread-Level Parallelism (TLP) and OpenMP

Instructors:

John Wawrzynek & 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)
- Synchronization – coordination among threads
  - MIPS: atomic read-modify-write using load-linked/store-conditional
- OpenMP as simple parallel extension to C
  - Pragmas for forking multiple Threads
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble

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

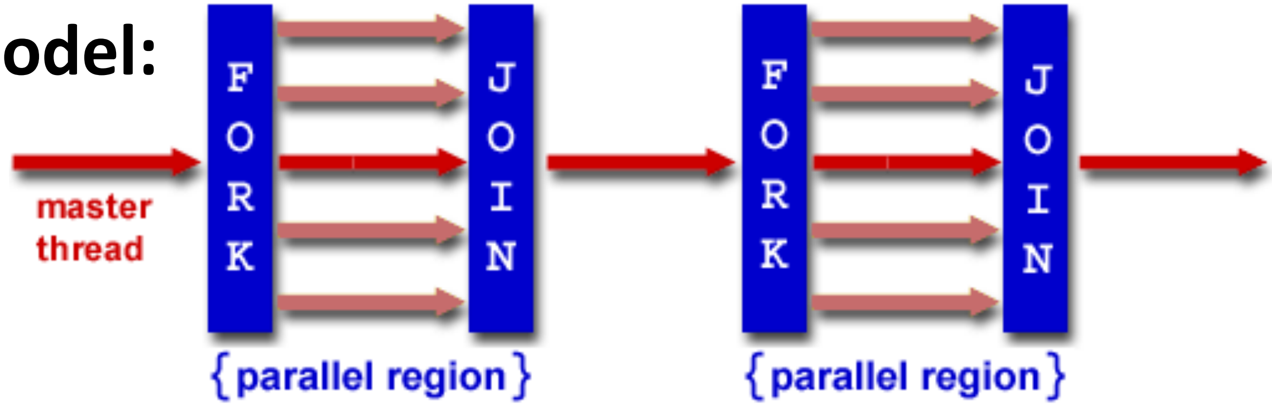
B: 100, 101, or 102

C: 100 or 101

D: 102

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:

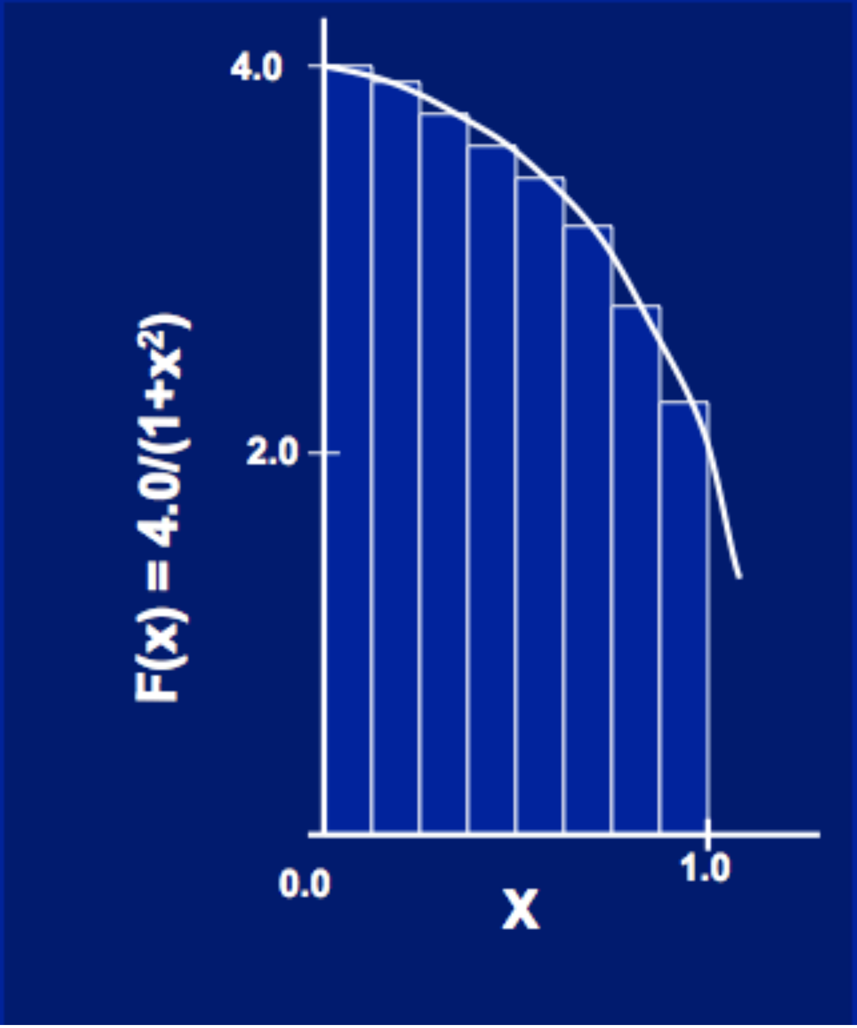
```
#pragma omp parallel
{
   /* code goes here */
}
```

  - *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 π

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 Δx and height $F(x_i)$ at the middle of interval i.

Sequential Calculation of π 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);
}
```

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

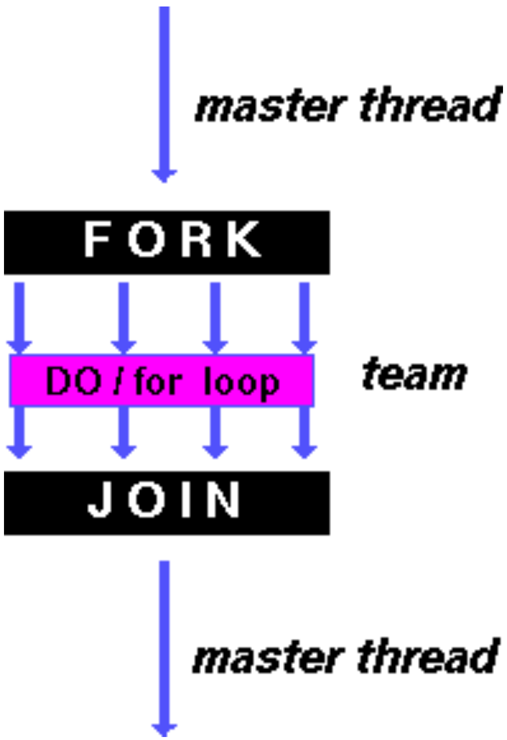
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Parallel OpenMP Version (1)

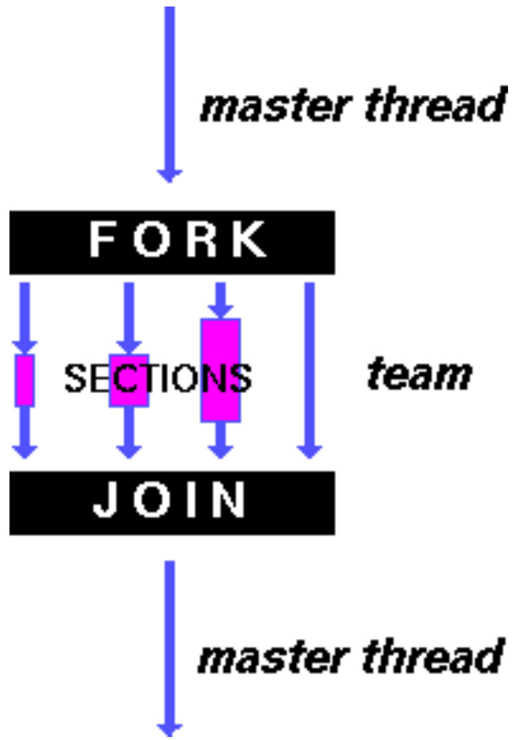
```
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

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)

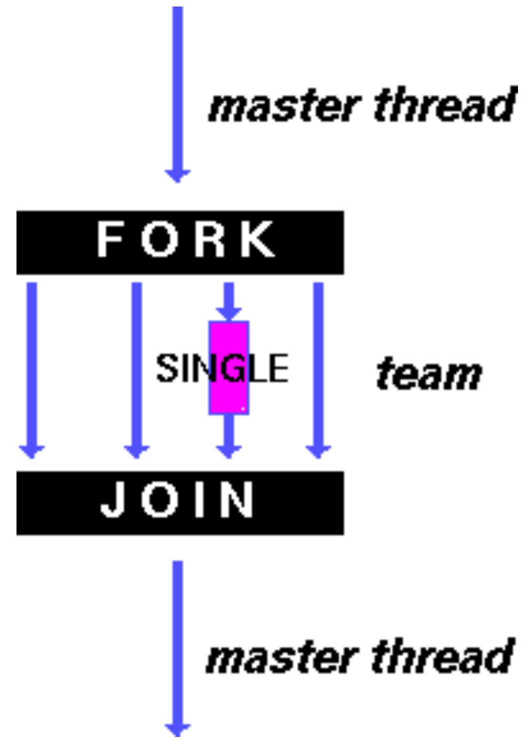
- These are defined *within* a `parallel` section



Shares iterations of a loop across the threads



Each section is executed by a separate thread



Serializes the execution of a thread

`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

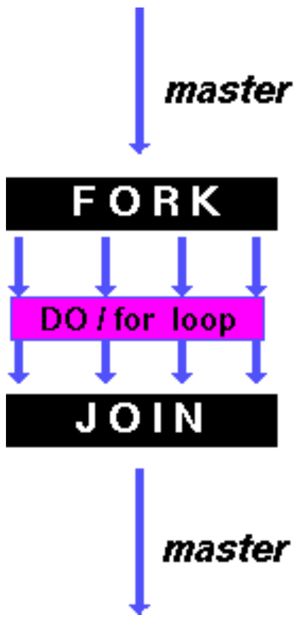
```
for (i=0; i<max; i++) zero[i] = 0;
```

- 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

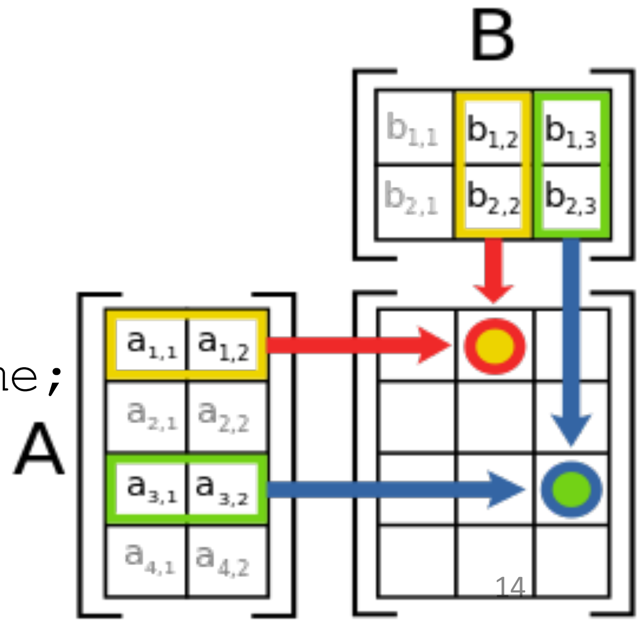
  – Time is measured per thread, no guarantee can be made that two distinct threads measure the same time

  – 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
  - 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 π 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

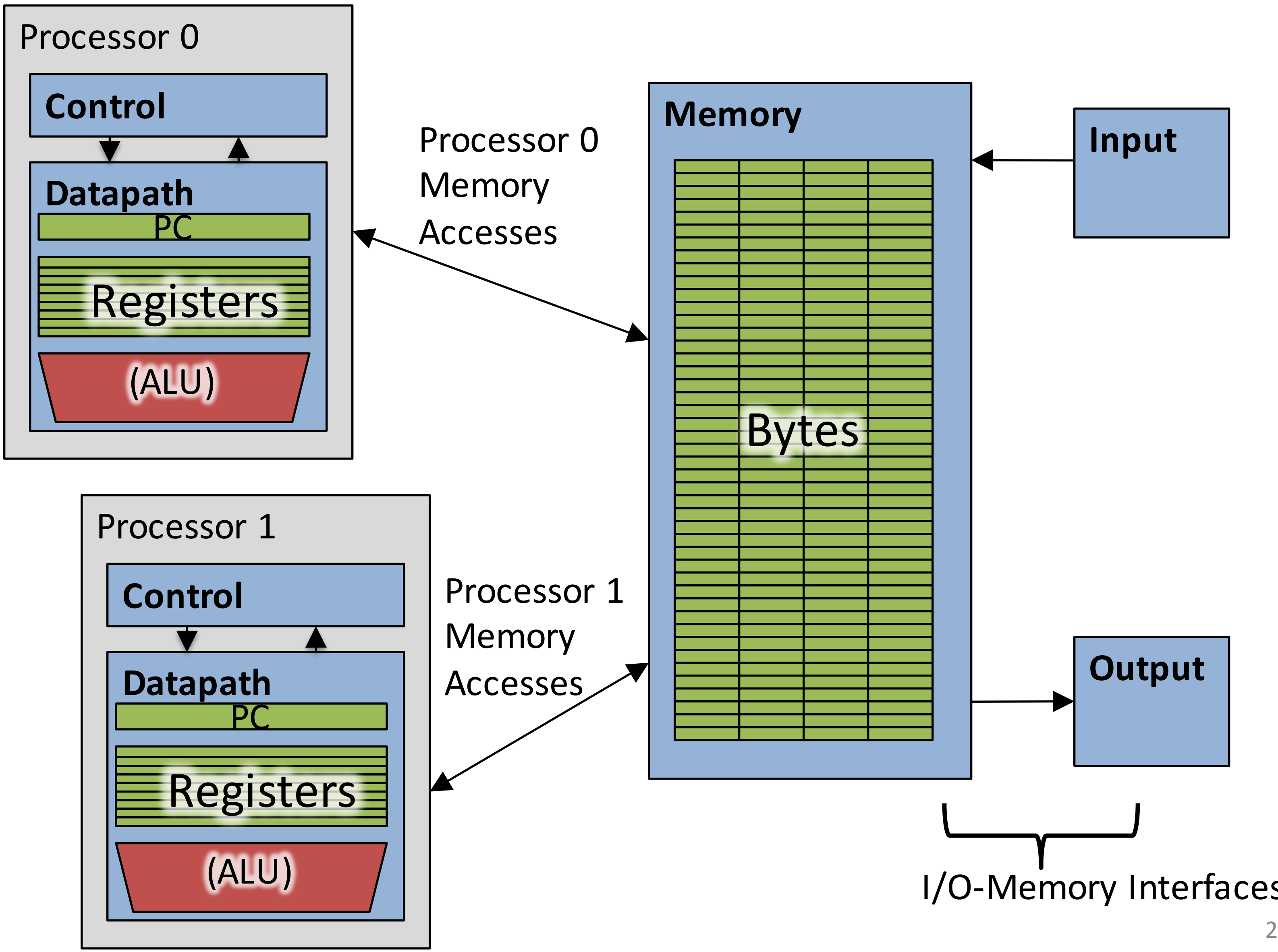
```
#include <omp.h>
#include <stdio.h>
/static long num_steps = 100000;
double step;
void main ()
{    int i;    double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
     for (i=1; i<= num_steps; i++){
      x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
     }
     pi = sum / num_steps;
   printf ("pi = %6.8f\n", pi);
}
```

Administrivia

- MT2 is Tuesday, November 10th:
  - Covers lecture material up till 10/29 lecture
  - TWO cheat sheets, 8.5”x11”
- TA Review Session:

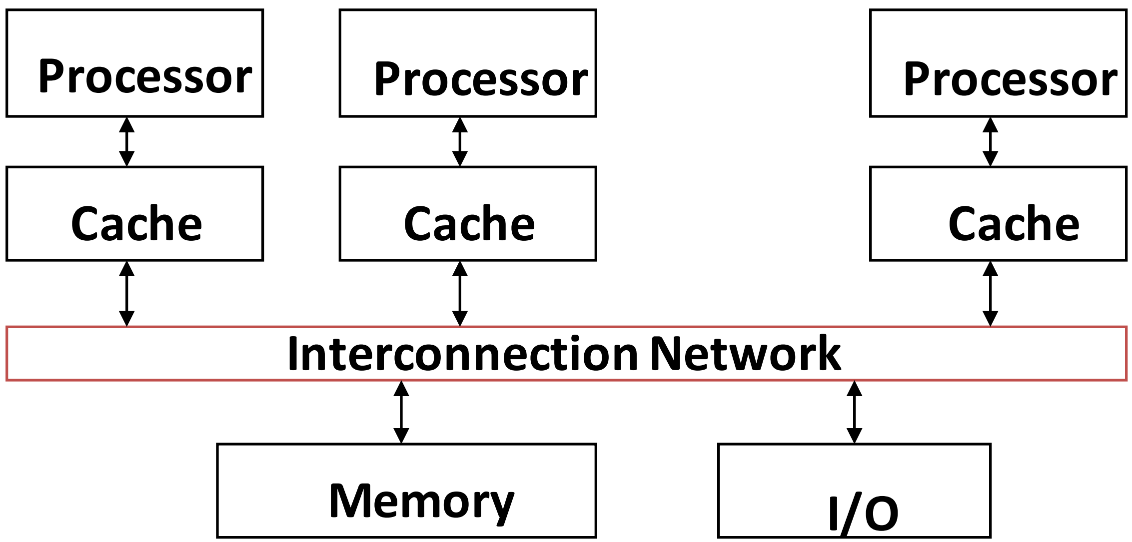
  Sunday 11/08, 5-7PM, 10 Evans
- MT2 Room Assignments
  - If your login is in [aaa-acz], go to 306 Soda
  - Else if you are in DSP, email Fred back
  - Else, go to Wheeler Auditorium

Simple Multi-core Processor



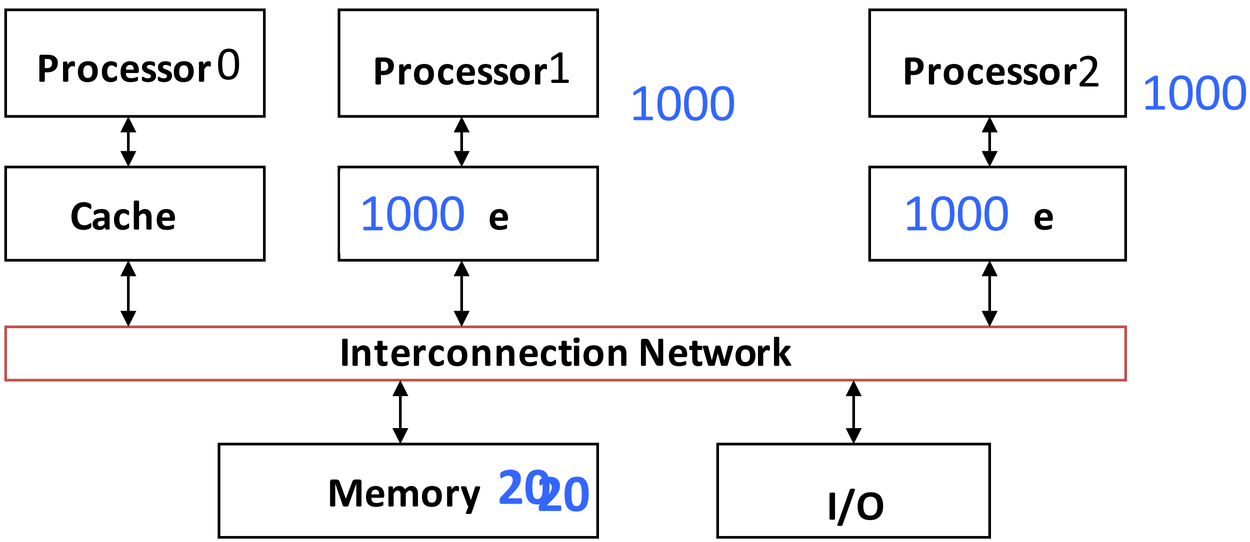
Multiprocessor Caches

- 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



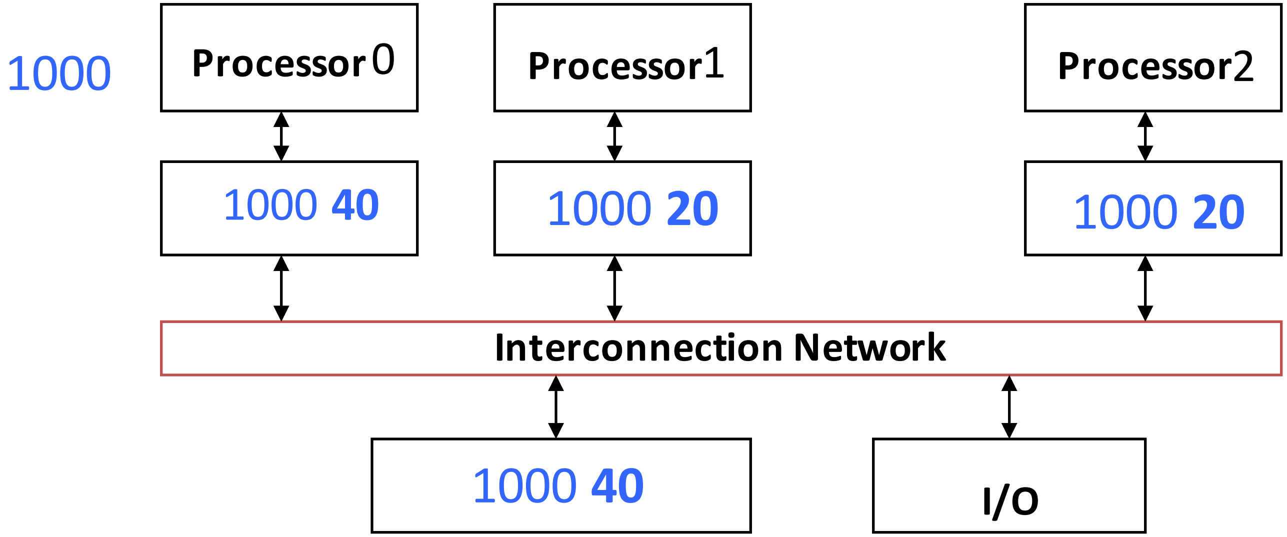
Shared Memory and Caches

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



Shared Memory and Caches

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



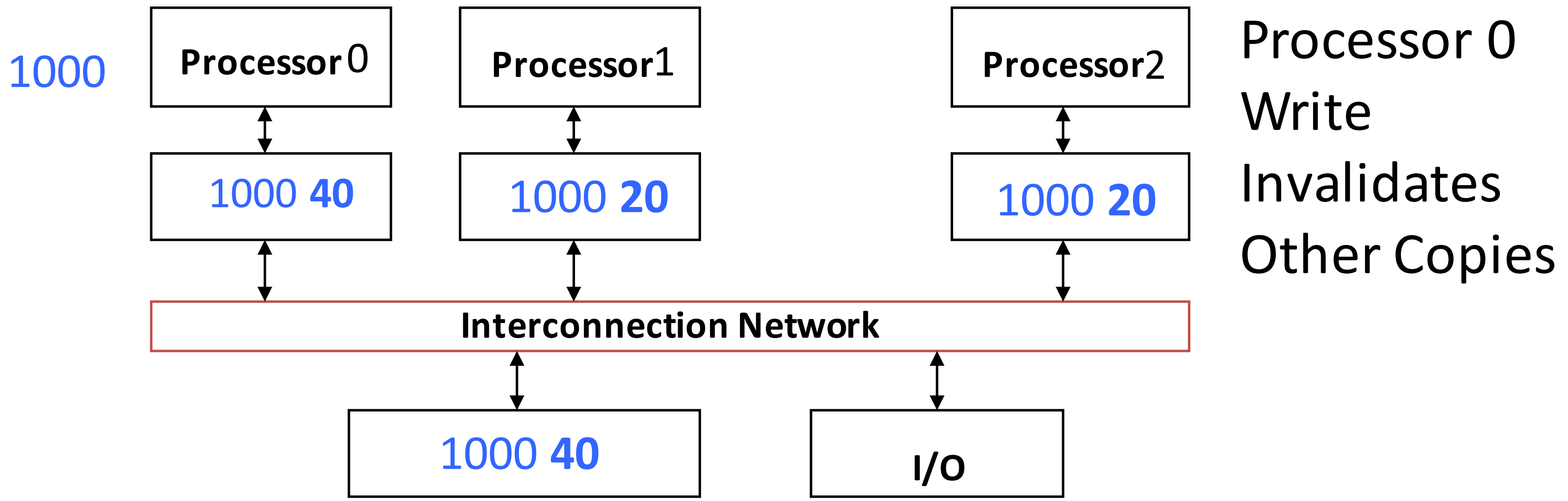
Problem?

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

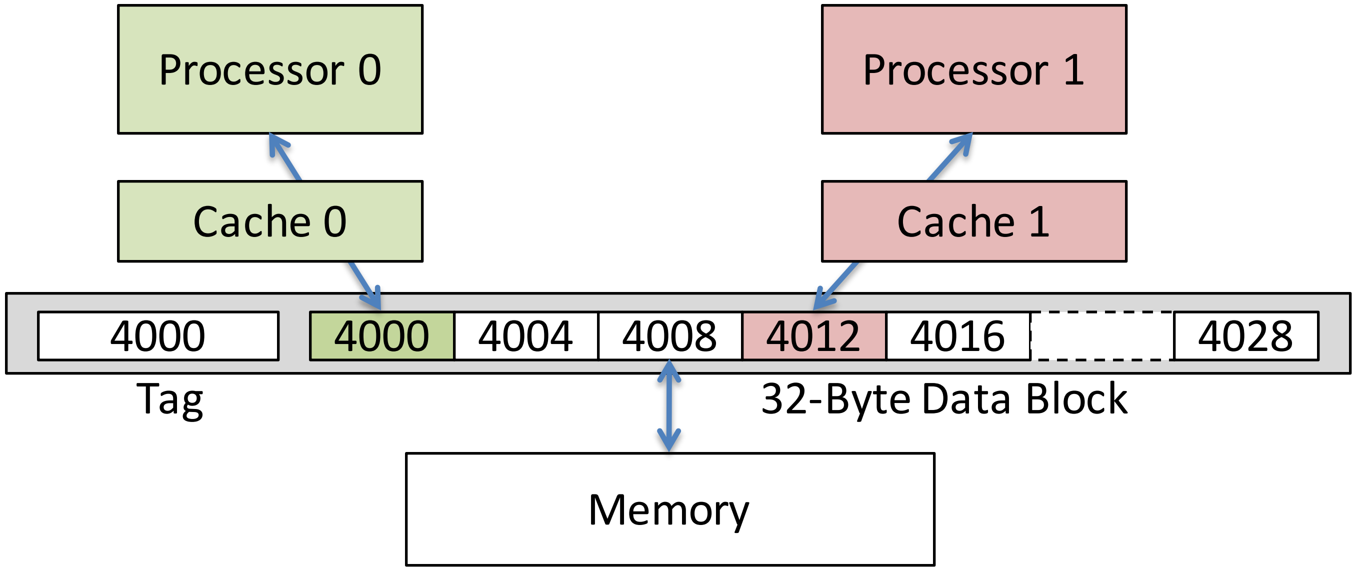
- Example, now with cache coherence
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



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, 1st 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

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 …
  - ≈ 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.)