CS 110
Computer Architecture
Lecture 21:
OpenMP & Cache Coherence

Instructors:

Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/21s/

School of Information Science and Technology

ShanghaiTech University

Slides based on UC Berkeley's 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
  - RISC-V: atomic read-modify-write using load-reserve/store-conditional & AMOs
- 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

OpenMP Programming Model - Review

- **Fork - Join Model:**
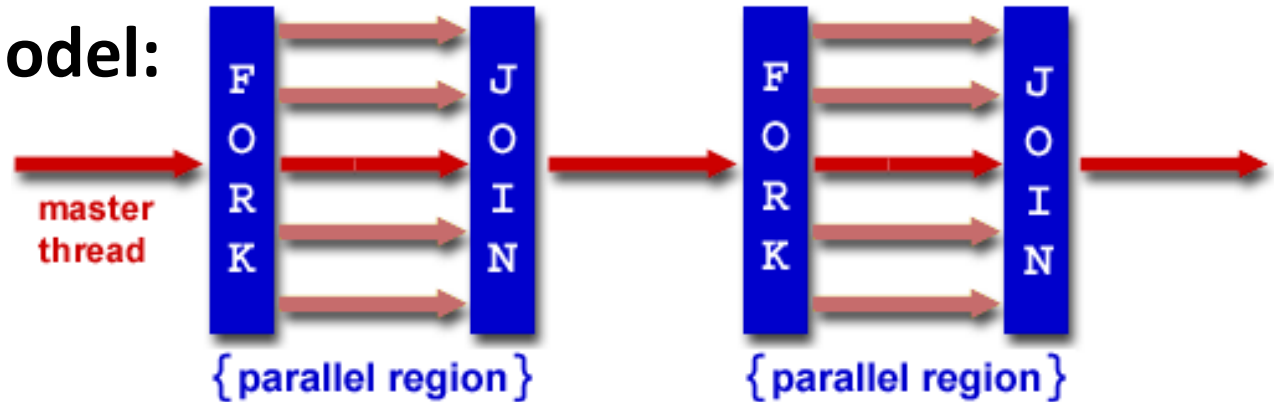
- OpenMP programs begin as single process (*master thread*) and executes sequentially until the first parallel region construct is encountered
  - *FORK:* Master thread then creates a team of parallel threads
  - Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads
  - *JOIN:* When the team of 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
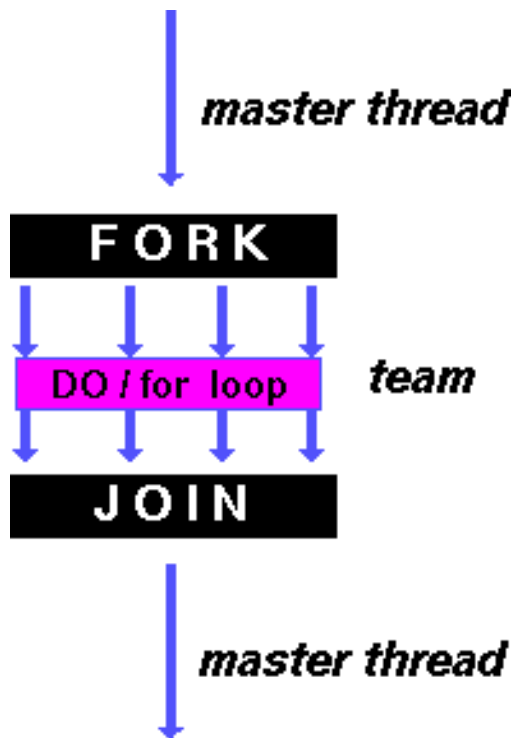   – Thread scheduling is *non-deterministic*

- OpenMP default is *shared* variables

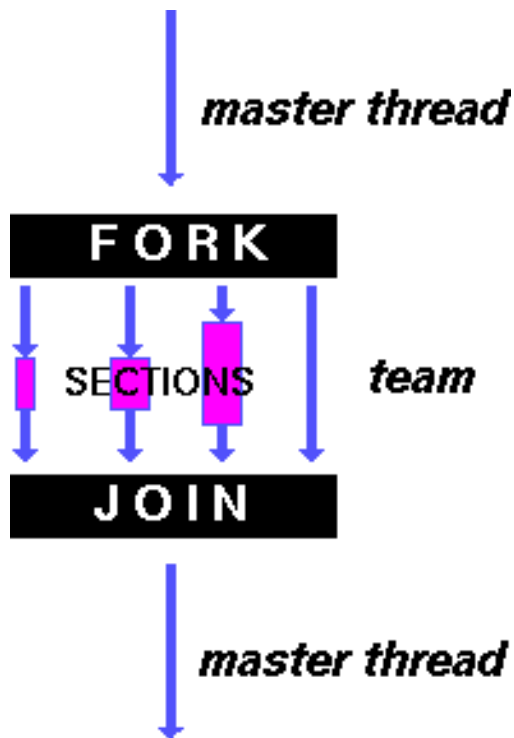   – To make private, need to declare with pragma:

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

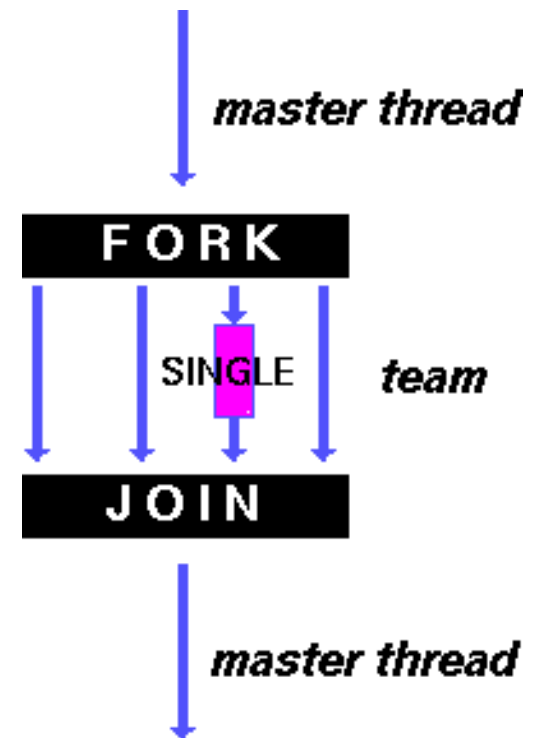# OpenMP Directives (Work-Sharing)

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

Shares iterations of a loop across the threads

Each section is executed by a separate thread
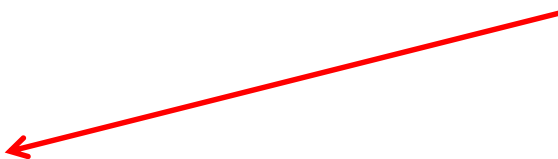
Serializes the execution of a thread

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?

8
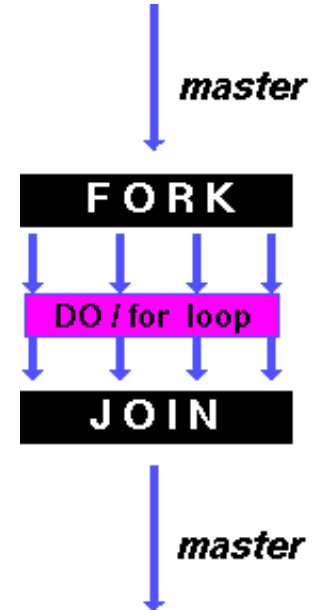
OpenMP Example

```
1  /* clang –Xpreprocessor –fopenmp –lomp –o for for.c */
2
3  #include <stdio.h>
4  #include <omp.h>
5  int main()
6  {
7      omp_set_num_threads(4);
8      int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
9      int N = sizeof(a)/sizeof(int);
10
11     #pragma omp parallel for
12     for (int i=0; i<N; i++) {
13         printf("thread %d, i = %2d\n",
14             omp_get_thread_num(), i);
15         a[i] = a[i] + 10 * omp_get_thread_num();
16     }
17
18     for (int i=0; i<N; i++) printf("%02d ", a[i]);
19     printf("\n");
20 }
```

```
$ gcc-5 -fopenmp for.c;./a.out
% clang -Xpreprocessor -fopenmp -
lomp -o for for.c;  ./for
thread 0, i =  0
thread 1, i =  3
thread 2, i =  6
thread 3, i =  8
thread 0, i =  1
thread 1, i =  4
thread 2, i =  7
thread 3, i =  9
thread 0, i =  2
thread 1, i =  5
00 01 02 13 14 15 26 27 38 39
```

The call to find the maximum number of threads that are available to do work is `omp_get_max_threads()` (from omp.h).

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
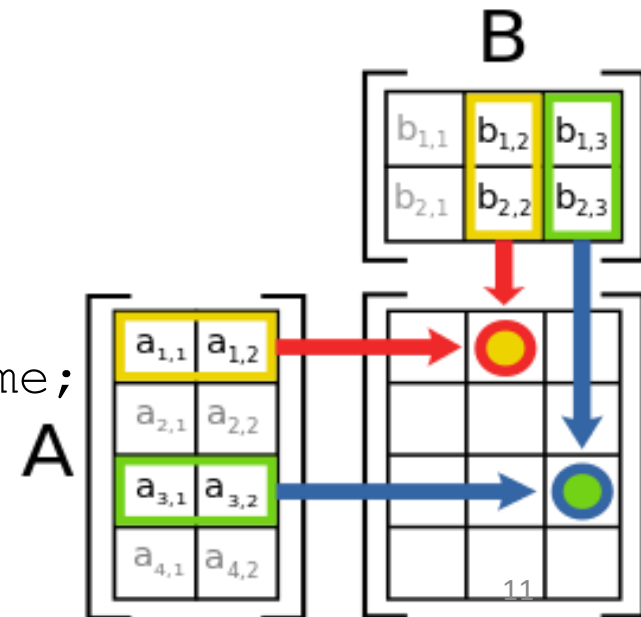
Notes on Matrix Multiply Example

- 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*)

Example: Calculating π

Numerical Integration
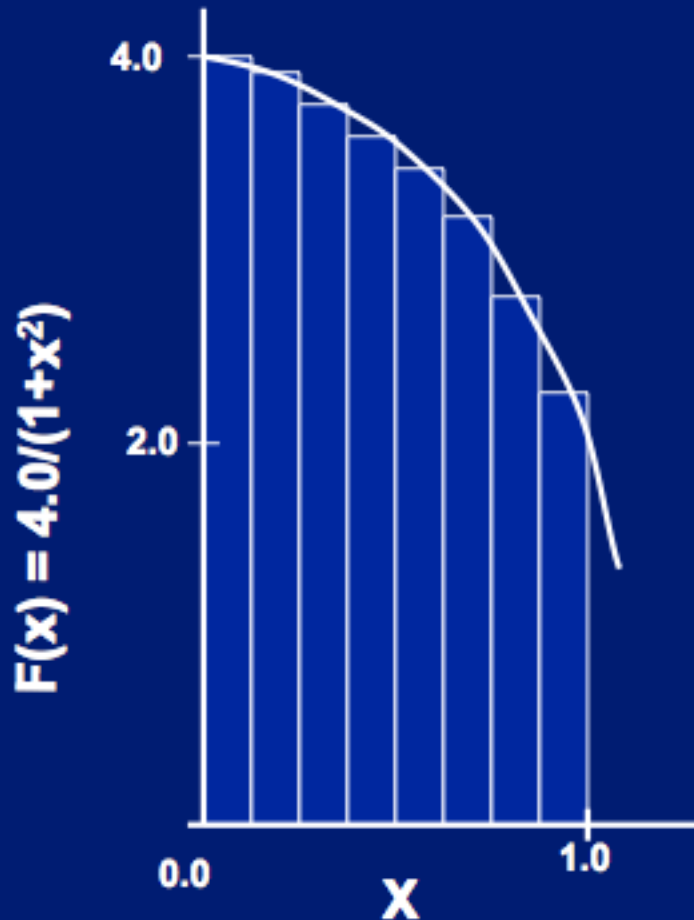
Mathematically, we know that:

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

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 π

```c
#include <stdio.h>

void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5) *step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

pi = 3.142425985001

- Resembles π, but not very accurate
- Let's increase **num_steps** and parallelize

Parallelize (1) …

```c
#include <stdio.h>
```
#include <omp.h>

```c
void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
#pragma parallel for
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5) *step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

- Problem: each thread needs access to the shared variable **sum**
- Code runs sequentially …

Parallelize (2) …
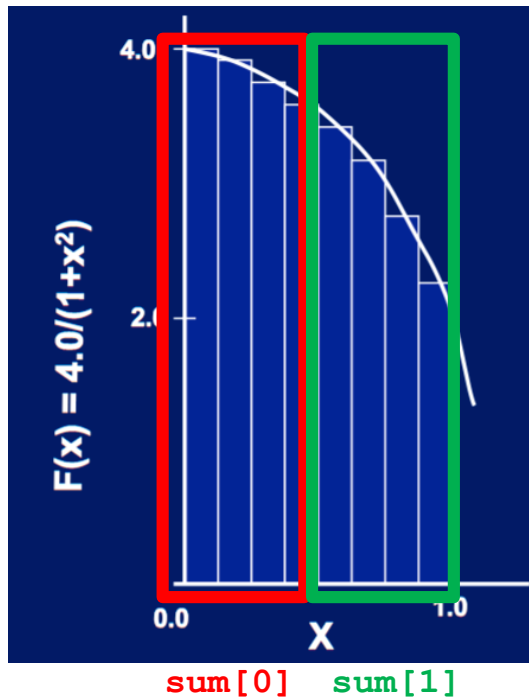
1. Compute
   `sum[0]` and `sum[1]`
   in parallel

2. Compute
   `sum = sum[0] + sum[1]`
   sequentially

Parallel π——Trial Run

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

```
i =  1,  id =  1
i =  0,  id =  0
i =  2,  id =  2
i =  3,  id =  3
i =  5,  id =  1
i =  4,  id =  0
i =  6,  id =  2
i =  7,  id =  3
i =  9,  id =  1
i =  8,  id =  0
pi = 3.142425985001
```

Scale up: `num_steps = ` 10^6

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 1000000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            // printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

`pi =`
`3.141592653590`

You verify how many digits are correct …

Can We Parallelize Computing `sum`?

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

Always looking for ways to beat Amdahl's Law …

Summation inside parallel section
- Insignificant speedup in this example, but …
- `pi = 3.138450662641`
- Wrong! And value changes between runs?!
- What's going on?

Question

What are the possible values of ***(x11)** after executing this code by two *concurrent* threads?

```
# *(x11) = 100
lw    x12,0(x11)
addi  x12,x12,1
sw    x12,0(x11)
```

Values of *(x1) ?
A: None of these
B: 100
C: 101
D: 102
E: 100 or 101
F: 101 or 102
G: 100 or 102
H: 100 or 101 or 102

Case 0

Thread 0	Thread 1
x12←100	
x12←101	
*(x11)←101	
	x12←101
	x12←102
	*(x11)←102

Case 1

Thread 0	Thread 1
x12←100	
	x12←100
x12←101	
	x12←101
*(x11)←101	
	*(x11)←101

What's Going On?

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

- Operation is really

    pi = pi + sum[id]

- What if >1 threads reads current (same) value of **pi**, computes the sum, stores the result back to **pi**?
- Each processor reads same intermediate value of **pi**!
- Result depends on who gets there when
    - A "race" → result is **not deterministic**

21

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 <u>all</u> 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 :   +, *, -, &, ^, |, &&, or ||.
 - *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;
```

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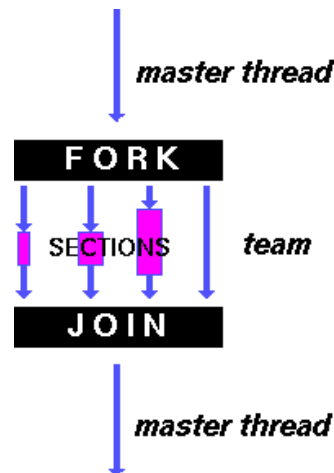
```c
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000;
double step;
void main (){
    int i;    double x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
    for (i=1; i<= num_steps; i++){
      x = (i - 0.5) * step;
      sum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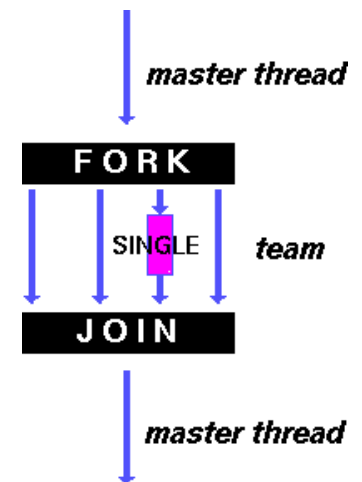
More on OpenMP

- These are defined *within* a `parallel` section
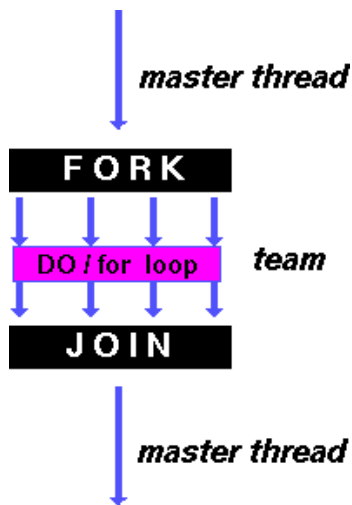
Shares iterations of a loop across the threads

Each section is executed by a separate thread

Serializes the execution of a thread

There are more, like `critical`, `barrier`, `atomic`, `master`, … Try them by yourself.

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
        int i = 0;
        omp_set_num_threads(4); // Maximum 4 threads
        #pragma omp parallel private(i)
        {
                printf("thread %d start\n", omp_get_thread_num());

                #pragma omp single
                {
                        for (i = 0; i < 6; i++)
                        {
                                printf("Single, thread %d execute i = %d\n",
                                        omp_get_thread_num(), i);
                        }
                }

        }
}
```

ection

master thread

R K

INGLE   *team*

```
wangc@HP:~/TT$ gcc single.c -o s -fopenmp
wangc@HP:~/TT$ ./s
thread 3 start
Single, thread 3 execute i = 0
Single, thread 3 execute i = 1
Single, thread 3 execute i = 2
Single, thread 3 execute i = 3
Single, thread 3 execute i = 4
Single, thread 3 execute i = 5
thread 1 start
thread 2 start
thread 0 start
wangc@HP:~/TT$
```

master thread   *er thread*

Shares iterations of a   ecution
loop across the threads

There are more, like cri   ourself.

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
        int i = 0;
        omp_set_num_threads(4); // Maximum 4 threads
        #pragma omp parallel private(i)
        {
                printf("thread %d start\n", omp_get_thread_num());

                #pragma omp master
                {
                        for (i = 0; i < 6; i++)
                        {
                                printf("Master, thread %d execute i = %d\n",
                                    omp_get_thread_num(), i);
                        }
                }
                printf("Outside master,
                        omp_get_thread
        }
}
```

section

master thread

```
wangc@HP:~/TT$ gcc master.c -o m -fopenmp
wangc@HP:~/TT$ ./m
thread 2 start
Outside master, thread 2 execute i = 0
thread 1 start
Outside master, thread 1 execute i = 0
thread 3 start
Outside master, thread 3 execute i = 0
thread 0 start
Master, thread 0 execute i = 0
Master, thread 0 execute i = 1
Master, thread 0 execute i = 2
Master, thread 0 execute i = 3
Master, thread 0 execute i = 4
Master, thread 0 execute i = 5
Outside master, thread 0 execute i = 6
wangc@HP:~/TT$
```
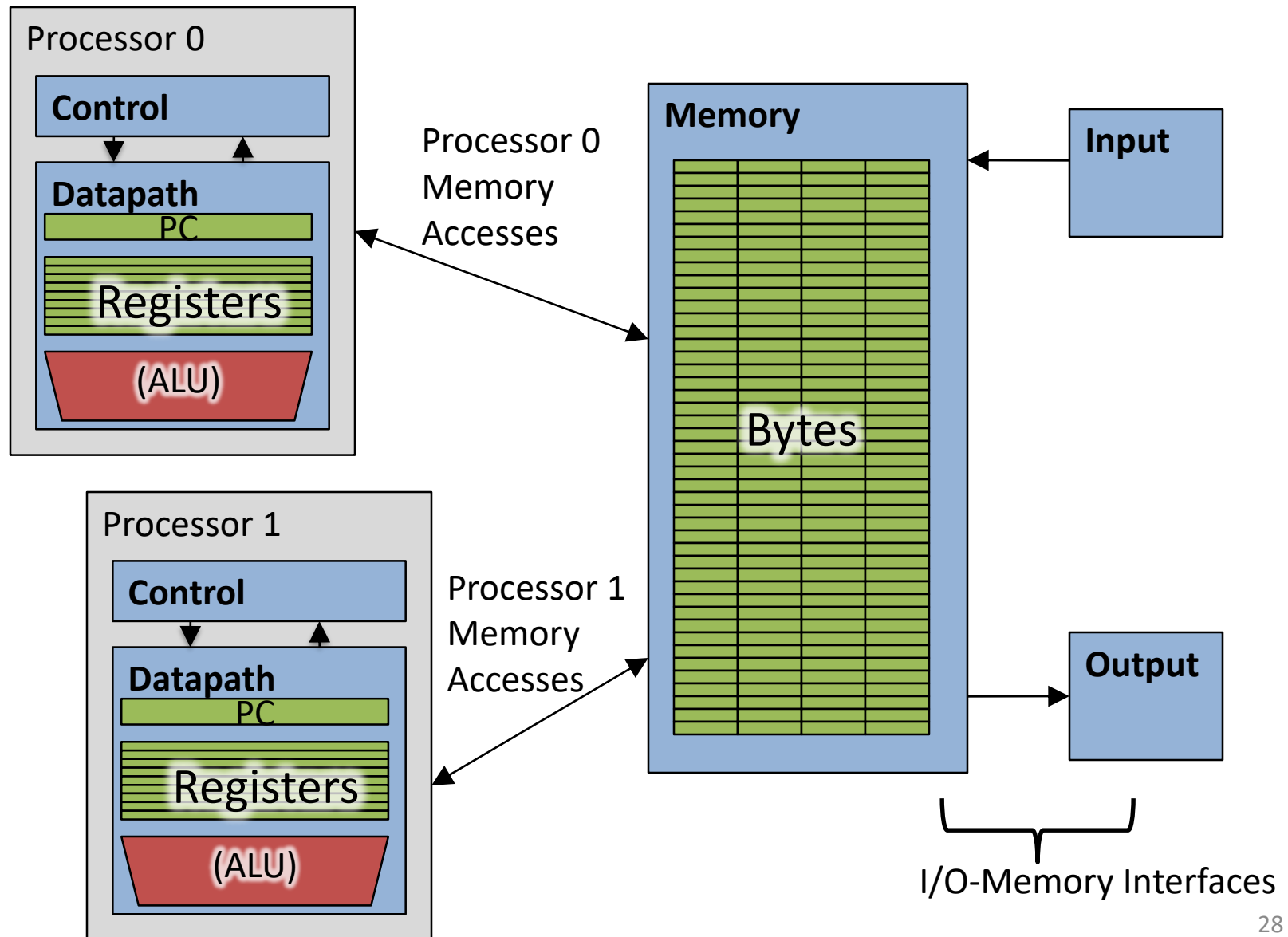
Shares iterations of a   Eac
loop across the threads   by a

There are more, like `critical`, ba

CACHE COHERENCE

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

Processor	Processor	Processor
Cache	Cache	Cache

Interconnection Network
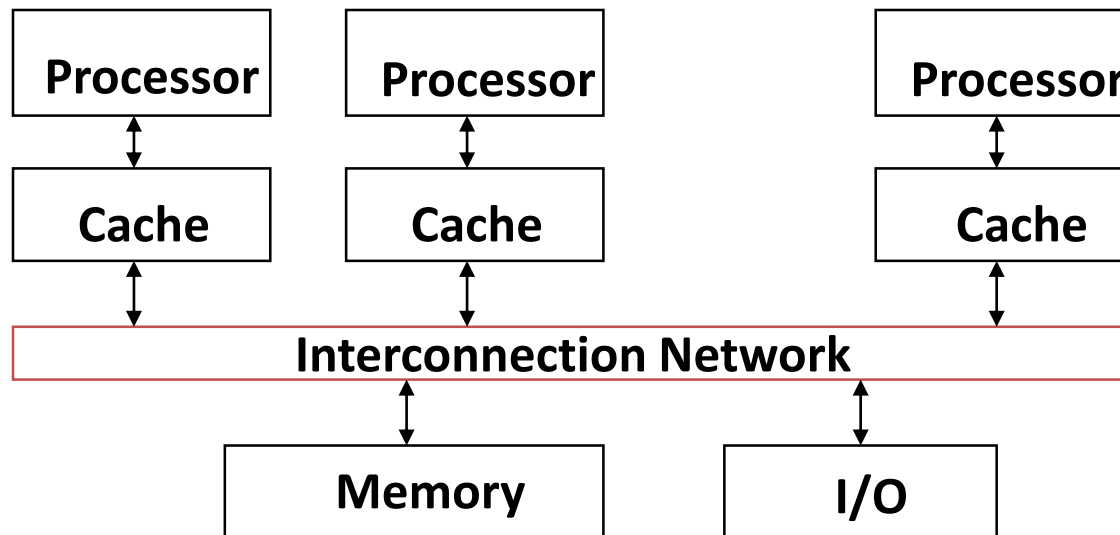
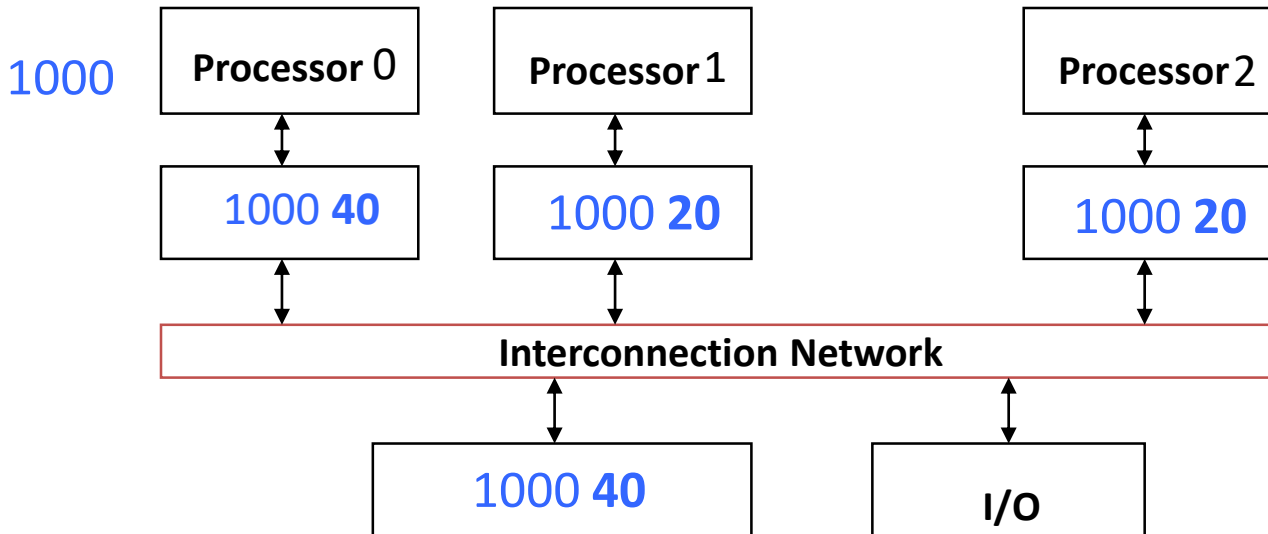| Memory	I/O

Shared Memory and Caches

- ## What if?

  – Processors 1 and 2 read Memory[1000] (value 20)
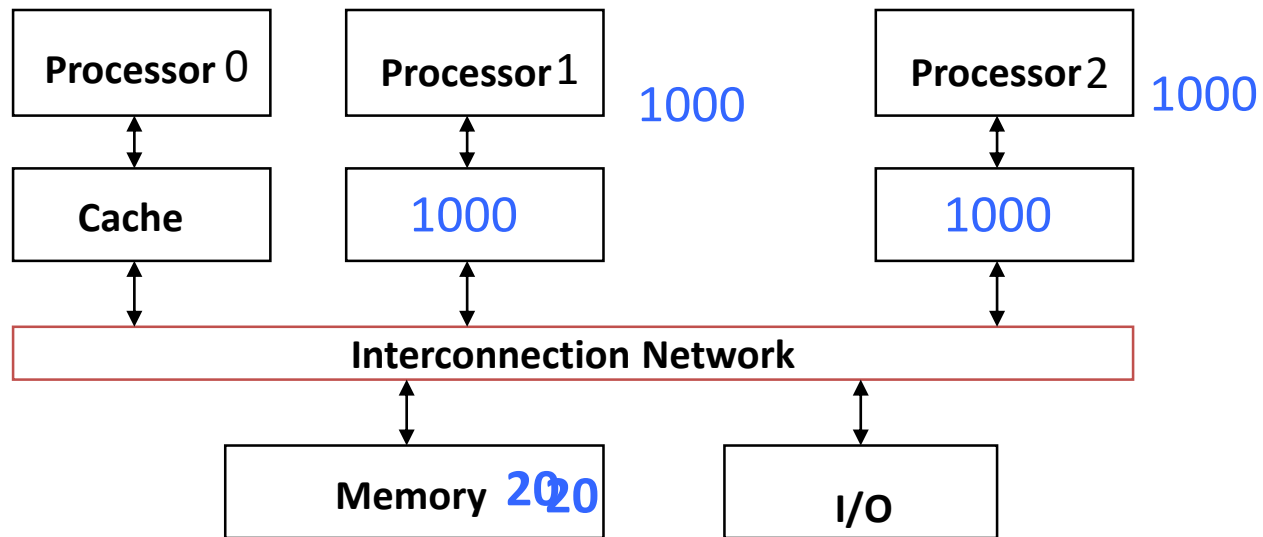
Shared Memory and Caches

- Now:

  – Processor 0 writes Memory[1000] with 40

1000

Processor 0	Processor 1	Processor 2
1000 **40**	1000 **20**	1000 **20**

Interconnection Network

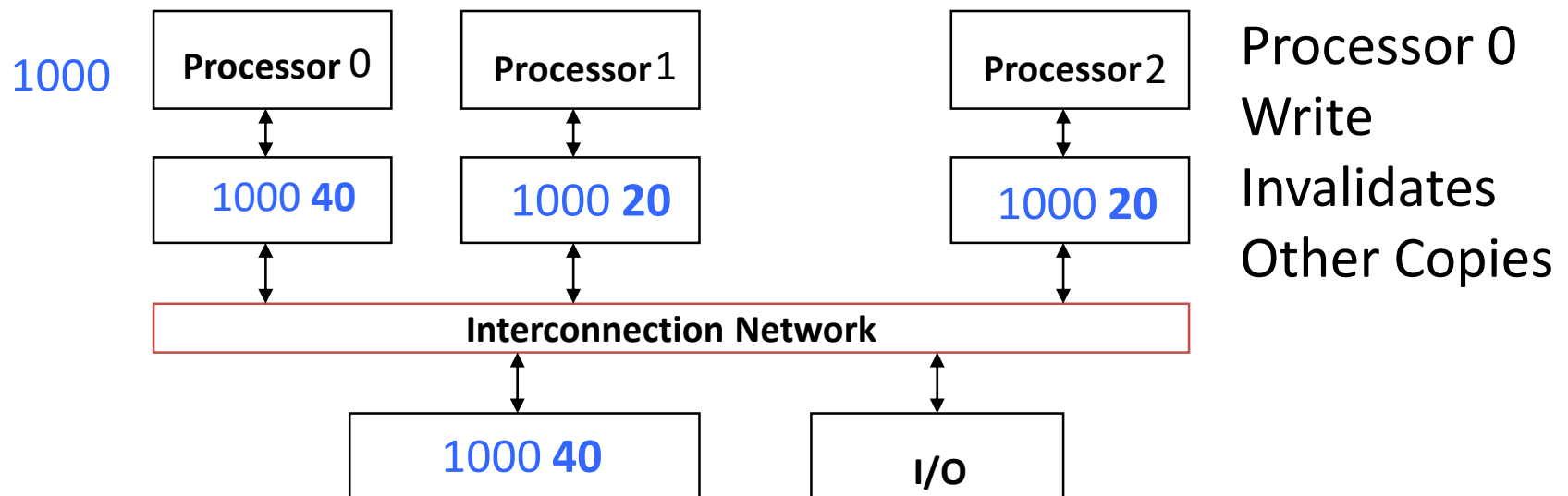| 1000 **40** | **I/O** |

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
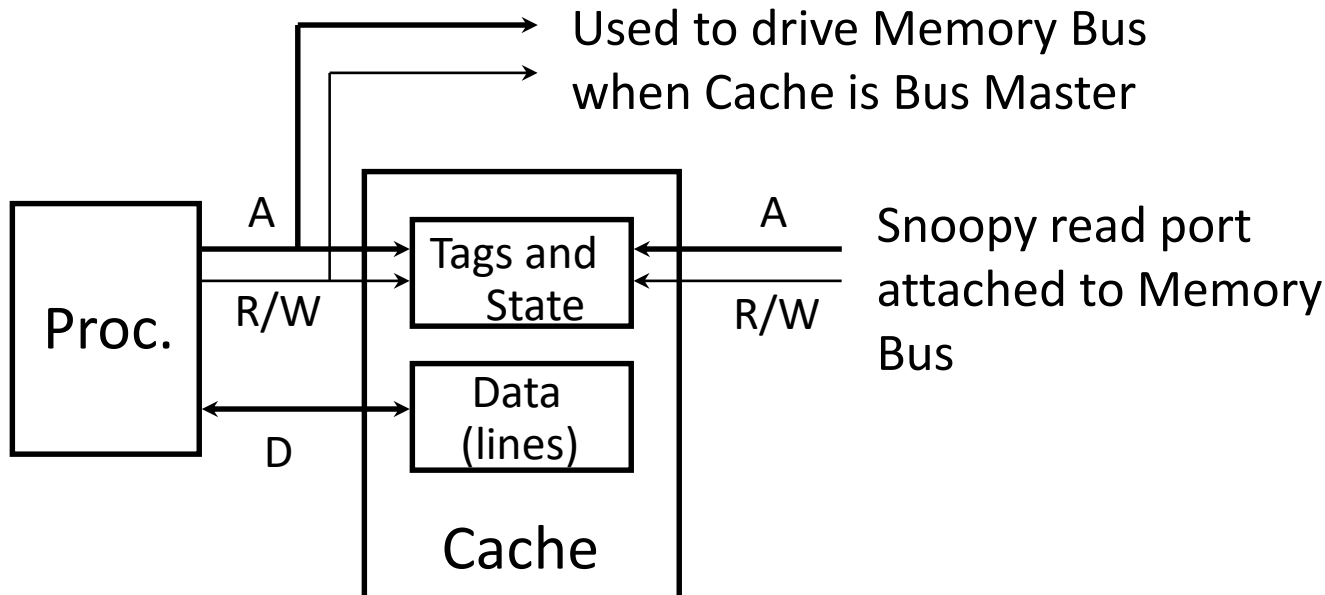 - Invalidate any copies of same address modified in other cache

Shared Memory and Caches

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

1000

Processor 0	Processor 1	Processor 2
1000 **40**	1000 **20**	1000 **20**

Processor 0 Write Invalidates Other Copies

Interconnection Network

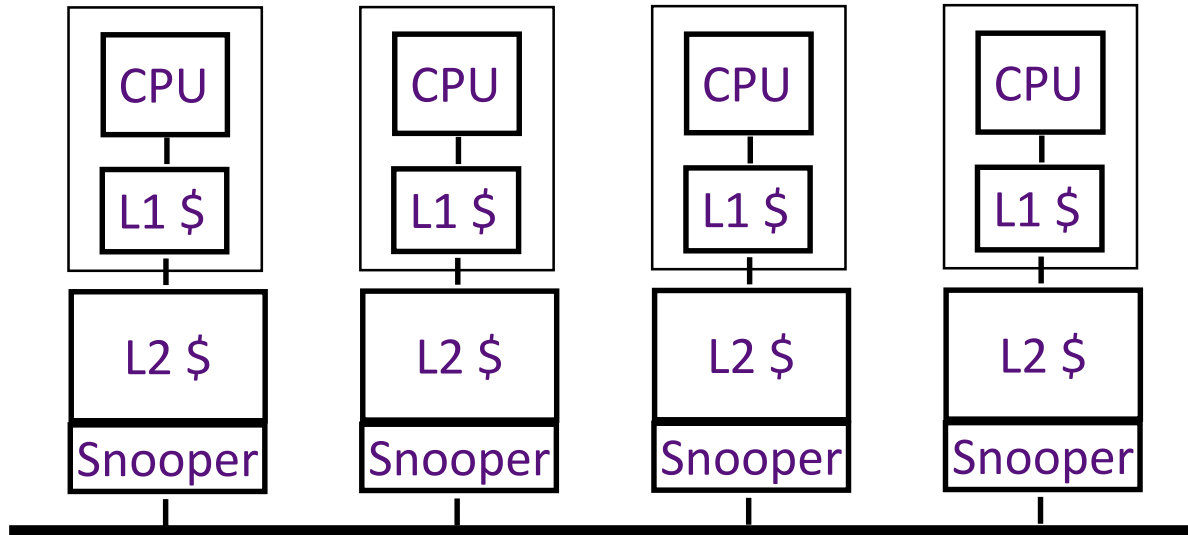| 1000 **40**	**I/O**

Snoopy Cache, *Goodman 1983*

- Idea: Have cache watch (or snoop upon) other memory transactions, and then "do the right thing"
- Snoopy cache tags are dual-ported

Used to drive Memory Bus
when Cache is Bus Master

Proc.

A

R/W

D

Tags and State

Data (lines)

Cache

A

R/W

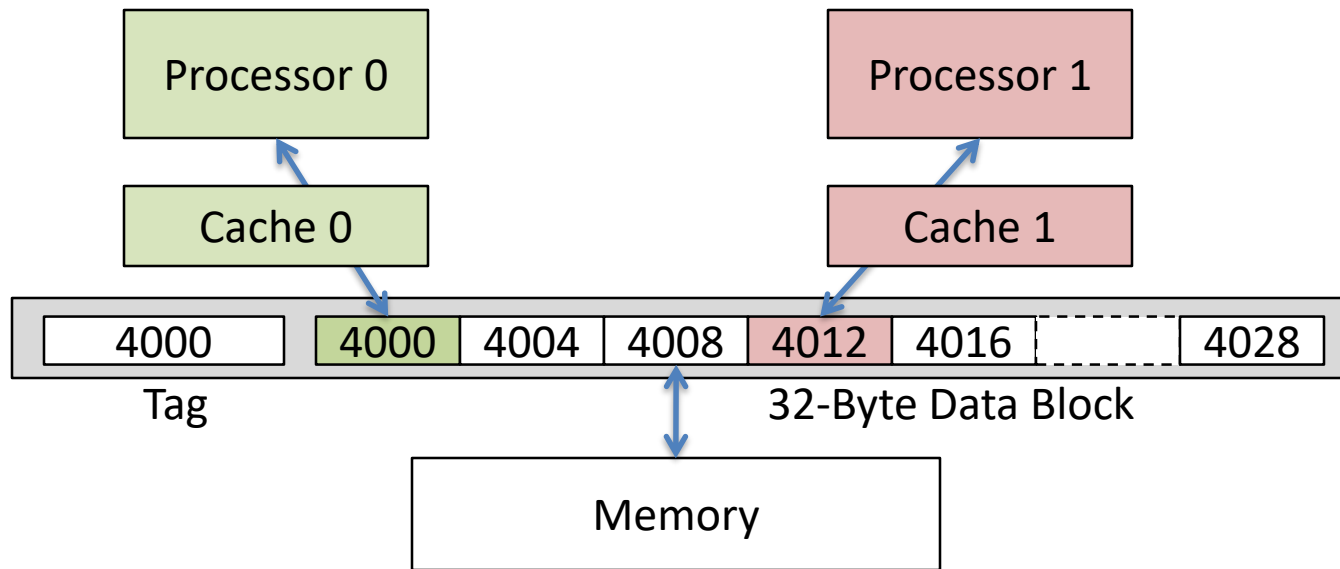Snoopy read port attached to Memory Bus

Optimized Snoop with Level-2 Caches

- Processors often have two-level caches
  - small L1, large L2 (usually both on chip now)
- Inclusion property: entries in L1 must be in L2
  - invalidation in L2 => invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

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?
  - Keep variables far apart (at least block size (64 byte))

Shared Memory and Caches

- Use valid bit to "unload" cache lines (in Processors 1 and 2)
- Dirty bit tells me: "I am the only one using this cache line"! => no need to announce on Network!
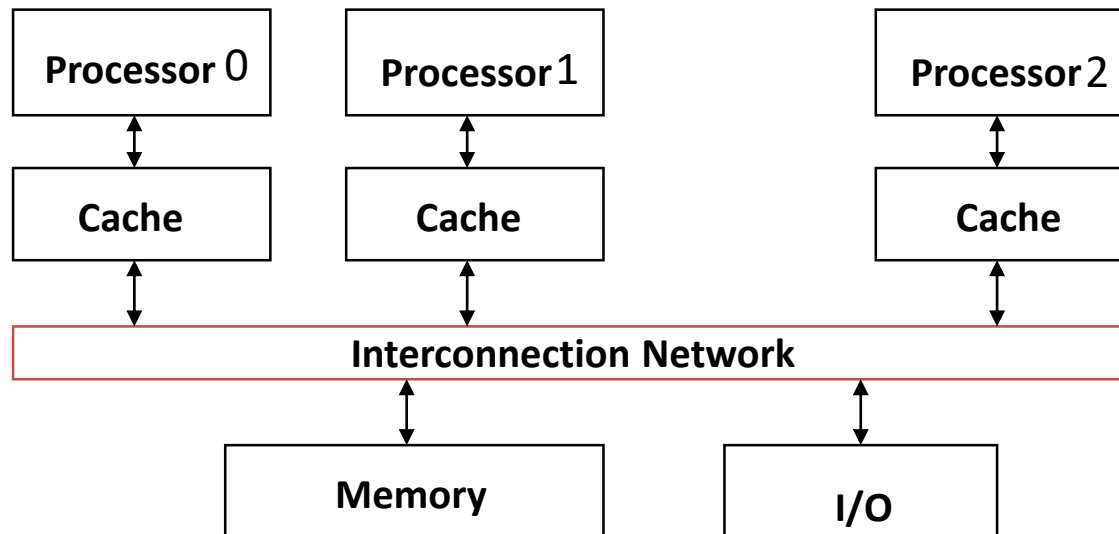
Review: Understanding Cache Misses: The 3Cs

- 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.)