UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Professor
Bora Nikolić

Thread-Level Parallelism III

Berkeley
UNIVERSITY OF CALIFORNIA

cs61c.org

Hardware Synchronization

Review: OpenMP Building Block: `for` loop

- **OpenMP as simple parallel extension to C**
  - Threads level programming with `parallel for` pragma
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble

```
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**
  - i.e. No `break`, `return`, `exit`, `goto` statements ← In general, don't jump outside of any `pragma` block

Garcia, Nikolić

Review: Data Races and Synchronization

- Two memory accesses form a data race if from different threads access same location, at least one is a write, and they occur one after another
- If there is a data race, result of program varies 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

Garcia, Nikolić

Hardware Synchronization

- **Solution:**
  - Atomic read/write
  - Read & write in single instruction
    - No other access permitted between read and write
  - Note:
    - Must use *shared memory* (multiprocessing)
- **Common implementations:**
  - Atomic swap of register ↔ memory
  - Pair of instructions for "linked" read and write
    - write fails if memory location has been "tampered" with after linked read
- **RISC-V has variations of both, but for simplicity we will focus on the former**

RISC-V Atomic Memory Operations (AMOs)

- **AMOs atomically perform an operation on an operand in memory and set the destination register to the original memory value**
- **R-Type Instruction Format:** `Add, And, Or, Swap, Xor, Max, Max Unsigned, Min, Min Unsigned`

31–27	26	25	24–20	19–15	14–12	11–7	6–0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
operation	ordering		src	addr	width	dest	AMO

Load from address in rs1 to “t”
rd = ”t”, i.e., the value in memory
Store at address in rs1 the calculation “t” <operation> rs2
aq(acquire) and rl(release) to insure in order execution

```
amoadd.w rd,rs2,(rs1):
  t = M[x[rs1]];
  x[rd] = t;
  M[x[rs1]] = t + x[rs2]
```


Garcia, Nikolić

RISCV Critical Section

- Assume that the lock is in memory location stored in register a0
- The lock is "set" if it is 1; it is "free" if it is 0 (it's initial value)

```
        li           t0, 1          # Get 1 to set lock
Try:    amoswap.w.aq t1, t0, (a0)   # t1 gets old lock value
                                    # while we set it to 1
        bnez         t1, Try        # if it was already 1, another
                                    # thread has the lock,
                                    # so we need to try again
         … critical section goes here …
        amoswap.w.rl x0, x0, (a0)   # store 0 in lock to release
```


Garcia, Nikolić

Lock Synchronization

Broken Synchronization

```
while (lock != 0) ;

lock = 1;

// critical section

lock = 0;
```

Fix (lock is at location (a0))

```
       li              t0, 1
Try:   amoswap.w.aq    t1, t0, (a0)
       bnez            t1, Try
Locked:

        # critical section

Unlock:
        amoswap.w.rl  x0, x0, (a0)
```


Garcia, Nikolić

OpenMP Locks

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

int main(void) {
    omp_lock_t lock;
    omp_init_lock(&lock);

#pragma omp parallel
    {
        int id = omp_get_thread_num();

        // parallel section
        // ...

        omp_set_lock(&lock);
        // start sequential section
        // ...
        printf("id = %d\n", id);

        // end sequential section
        omp_unset_lock(&lock);

        // parallel section
        // ...

    }
    omp_destroy_lock(&lock);
}
```


Garcia, Nikolić

Synchronization in OpenMP

- **Typically are used in libraries of higher level parallel programming constructs**
- **E.g. OpenMP offers `#pragma`s for common cases:**
  - critical
  - atomic
  - barrier
  - ordered
- **OpenMP offers many more features**
  - E.g., private variables, reductions
  - See online documentation
  - Or tutorial at
    - http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

Garcia, Nikolić

OpenMP Critical Section

```
#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);
        }
#pragma omp critical
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

Mutual exclusion

- Only one thread at a time can enter a **`critical`** region. (Threads wait their turn)

Berkeley
UNIVERSITY OF CALIFORNIA

Garcia, Nikolić

Deadlock

- **Deadlock: a system state in which no progress is possible**
- **Dining Philosopher's Problem:**
  - Think until the left fork is available; when it is, pick it up
  - Think until the right fork is available; when it is, pick it up
  - When both forks are held, eat for a fixed amount of time
  - Then, put the right fork down
  - Then, put the left fork down
  - Repeat from the beginning
- **Solution?**

Garcia, Nikolić

OpenMP Timing

- **Elapsed wall clock time:**

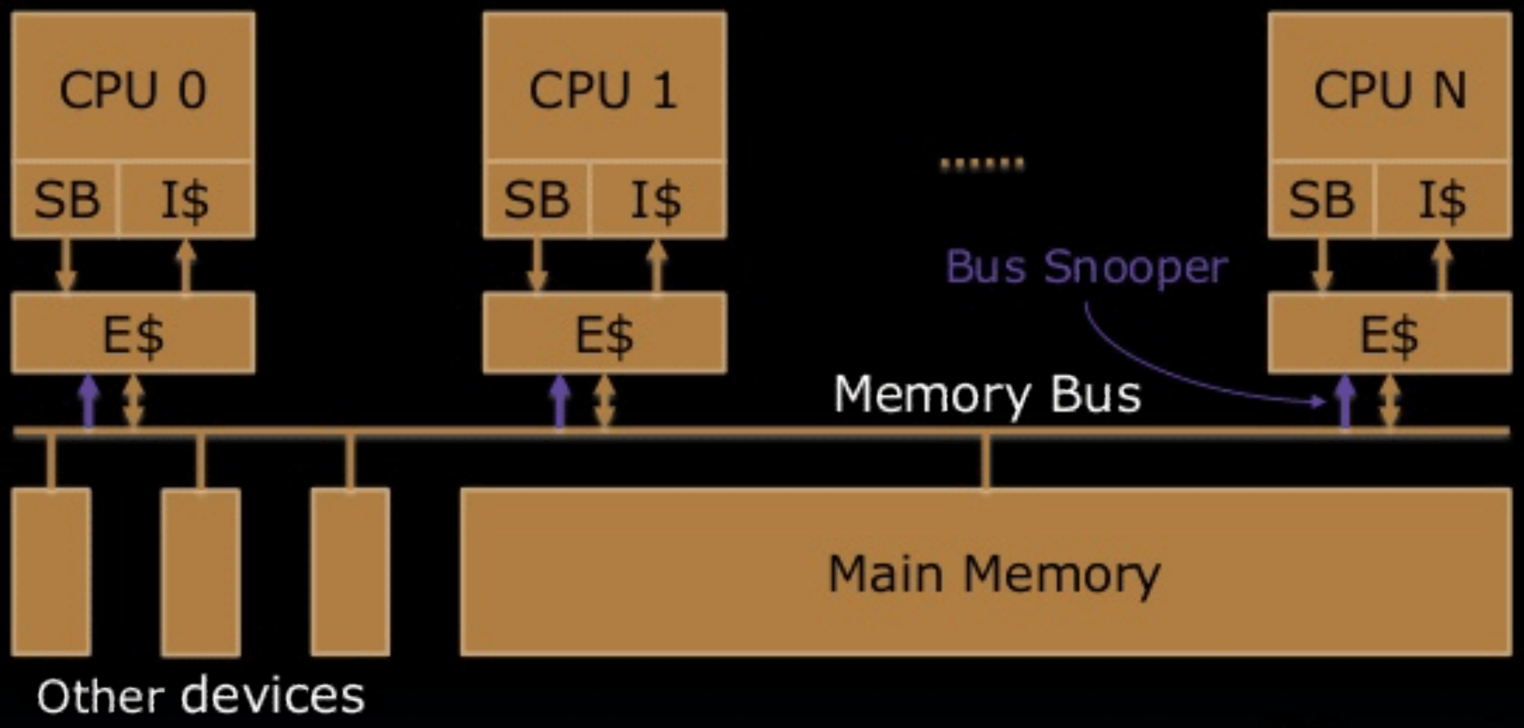
  `double omp_get_wtime(void);`

  - 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

Garcia, Nikolić

Shared Memory and Caches

(Chip) Multicore Multiprocessor



- **SMP: (Shared Memory) Symmetric Multiprocessor**
  - Two or more identical CPUs/Cores
  - Single shared coherent memory

Garcia, Nikolić

Multiprocessor Key Questions

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

Garcia, Nikolić

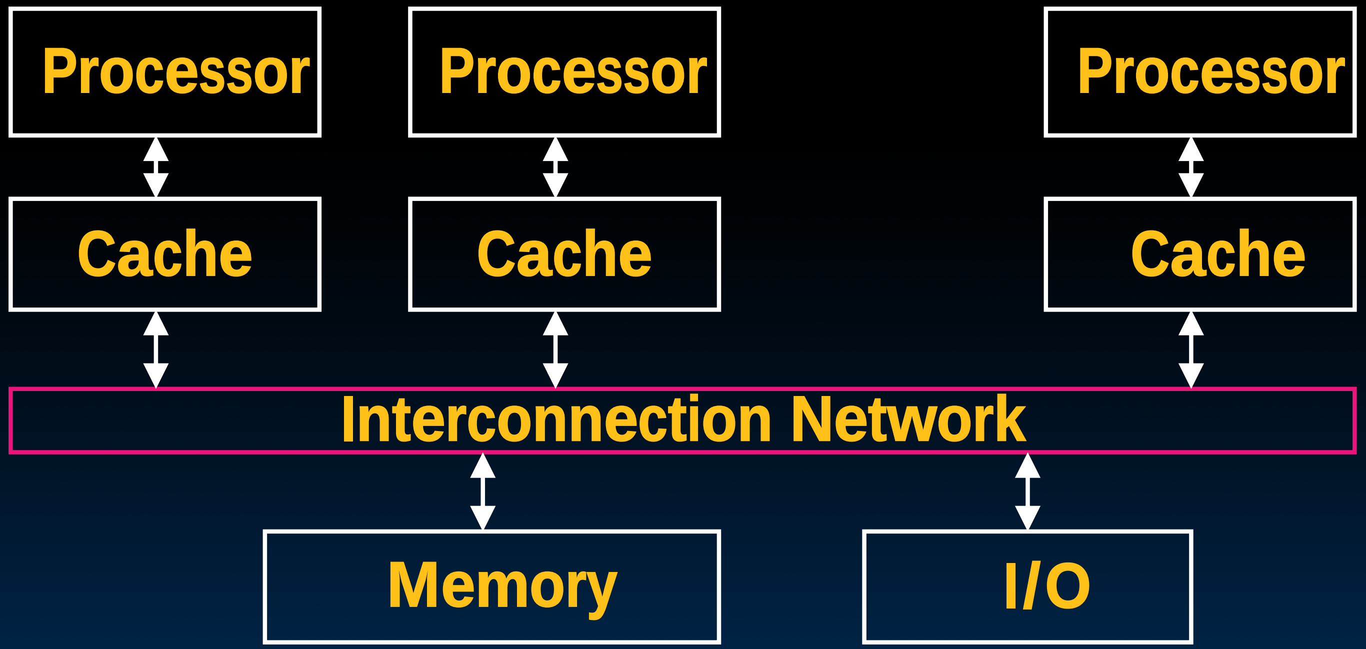
Shared Memory Multiprocessor (SMP)

- **Q1 – Single address space shared by all processors/cores**
- **Q2 – 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 SMP**

Garcia, Nikolić

Multiprocessor Caches

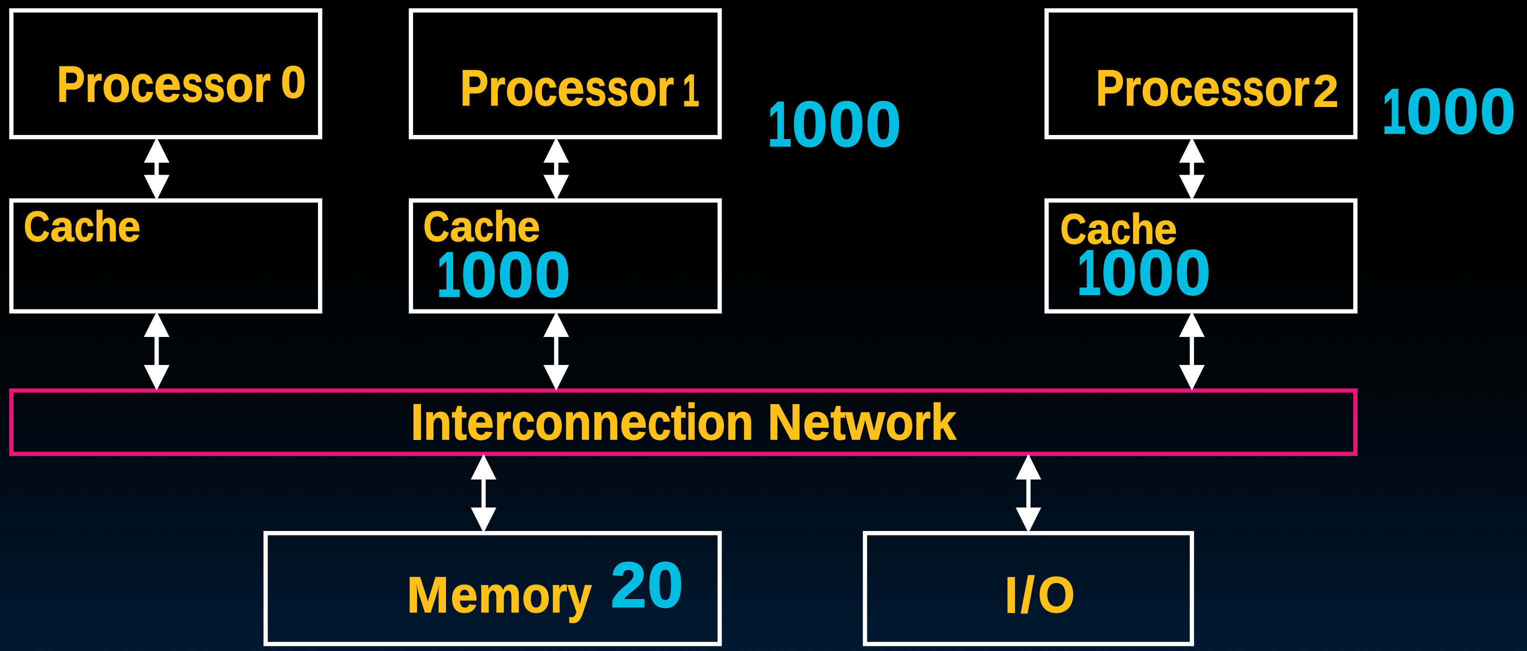
- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory



Garcia, Nikolić

Shared Memory and Caches

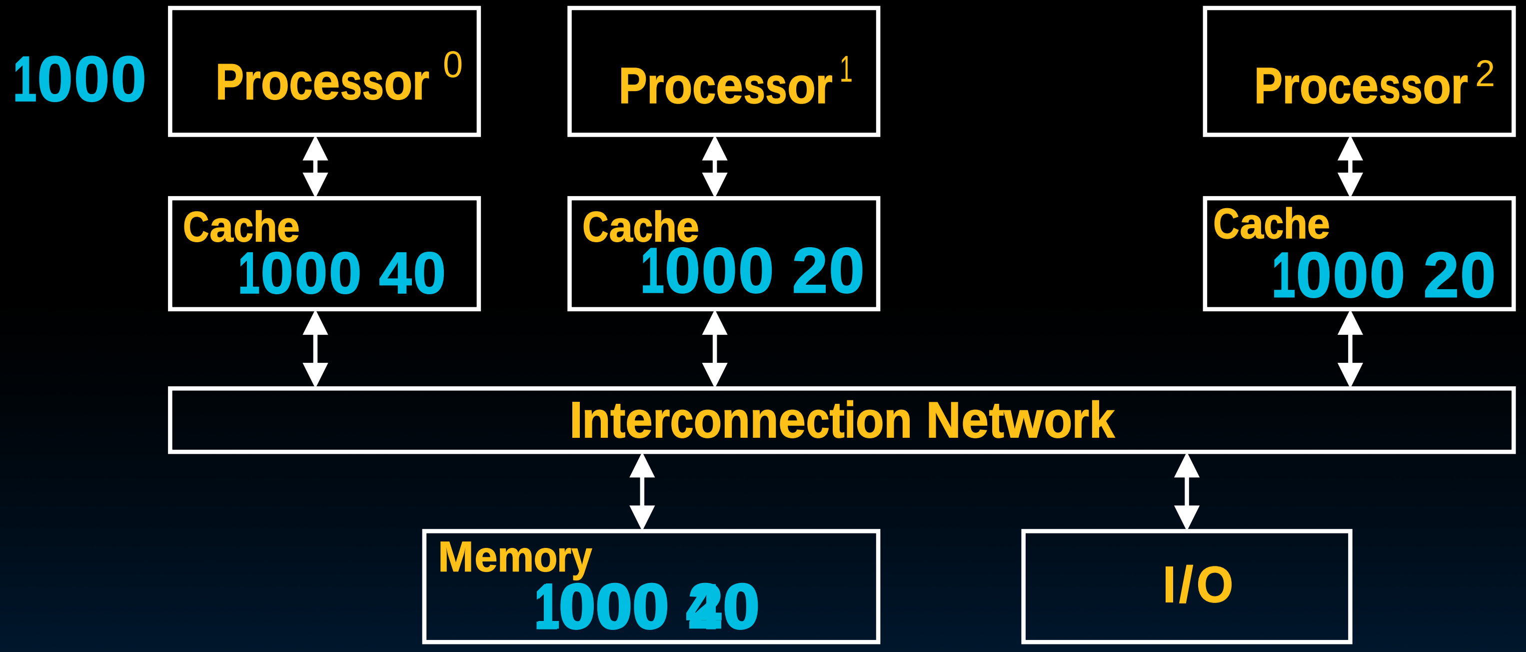
- **What if?**
  - Processors 1 and 2 read Memory[1000] (value 20)



Berkeley UNIVERSITY OF CALIFORNIA

Shared Memory and Caches

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



Problem?

Garcia, Nikolić

Cache Coherency

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

Garcia, Nikolić

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 (i.e., write back)

Garcia, Nikolić

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)**
   - This cache is one of several with a valid copy of the cache line, but has the exclusive right to make changes to it. It must broadcast those changes to all other caches sharing the line. The introduction of owned state allows dirty sharing of data, i.e., a modified cache block can be moved around various caches without updating main memory. The cache line may be changed to the Modified state after invalidating all shared copies, or changed to the Shared state by writing the modifications back to main memory. Owned cache lines must respond to a snoop request with data.

Garcia, Nikolić

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)

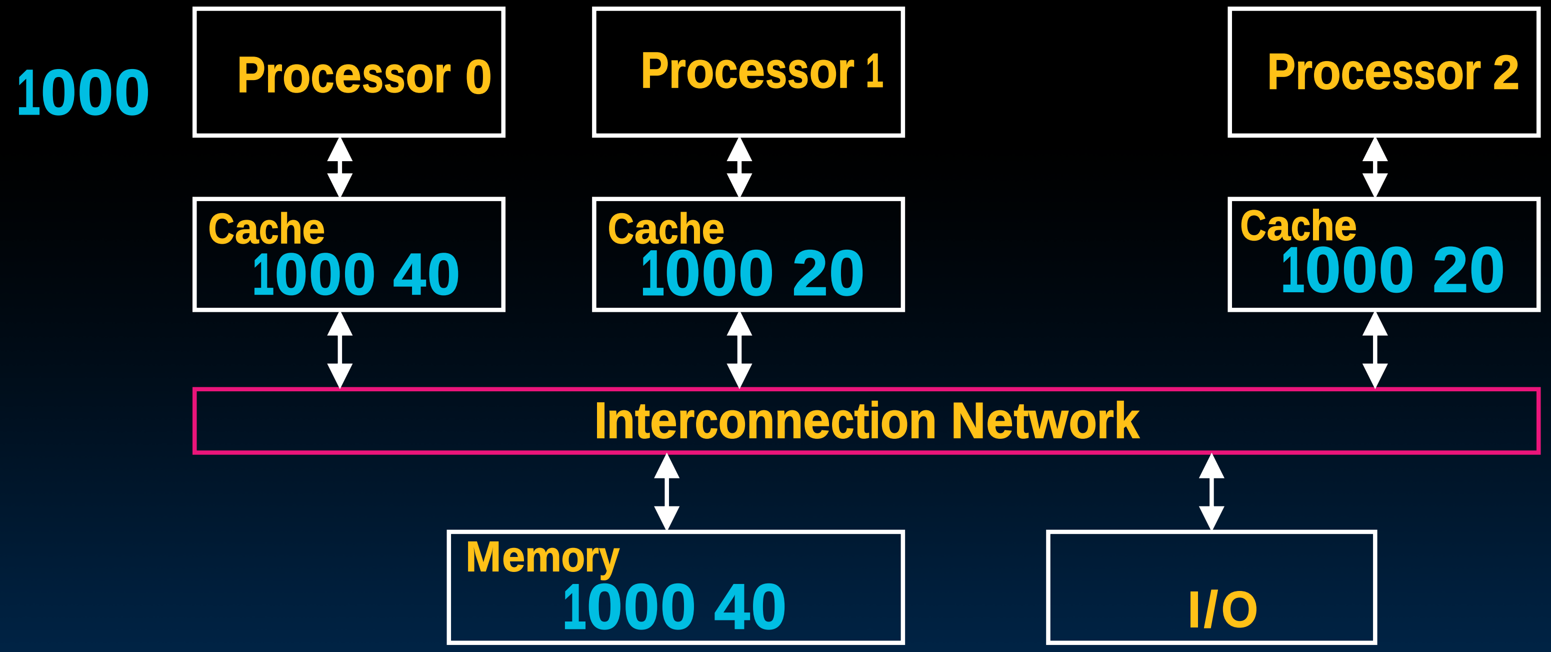
	M	O	E	S	I
M	✗	✗	✗	✗	✓
O	✗	✗	✗	✓	✓
E	✗	✗	✗	✗	✓
S	✗	✓	✗	✓	✓
I	✓	✓	✓	✓	✓

Snooping/Snoopy Protocols
e.g., the Berkeley Ownership Protocol
See `https://en.wikipedia.org/wiki/MOESI_protocol`

Garcia, Nikolić

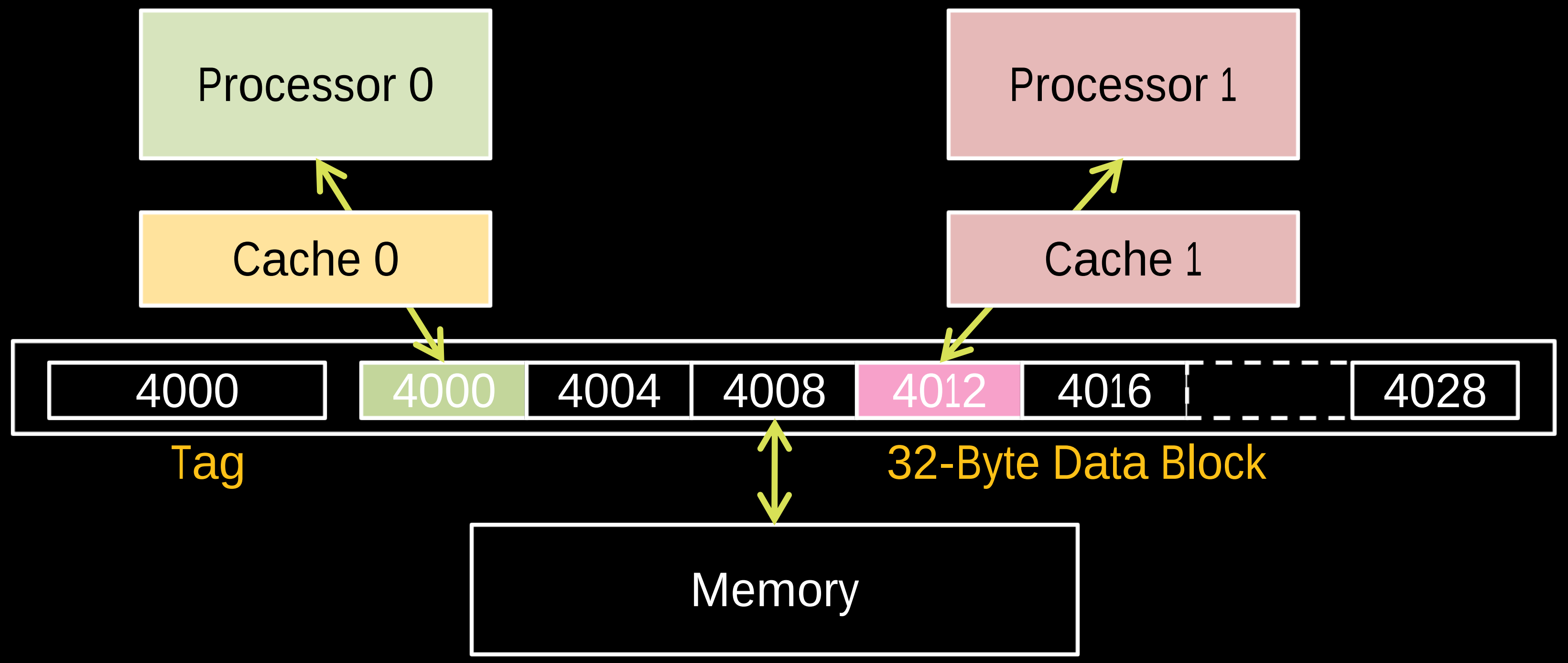
Shared Memory and Caches

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



Processor 0 Write Invalidates Other Copies

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?

Garcia, Nikolić

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?

Garcia, Nikolić

Remember 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

Garcia, Nikolić

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

Garcia, Nikolić

And, in Conclusion, …

- **OpenMP as simple parallel extension to C**
  - Threads level programming with `parallel for` pragma, `private` variables, `reductions`, …
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble
- **TLP**
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!

☑

Garcia, Nikolić