CS 61C: Great Ideas in Computer Architecture

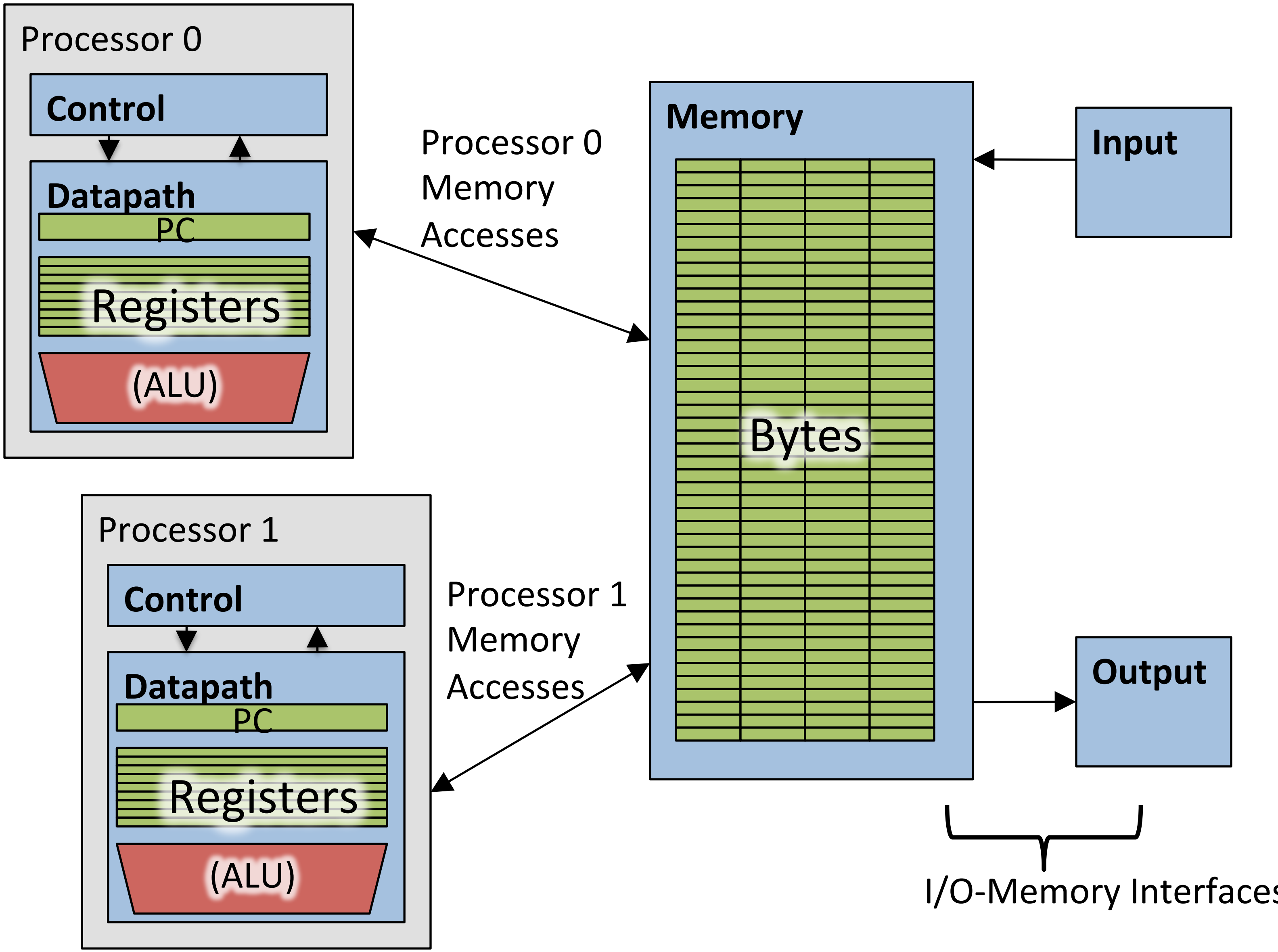
Lecture 20: *Thread-Level Parallelism (TLP) and OpenMP Part 2*

Instructor: Sagar Karandikar
sagark@eecs.berkeley.edu

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

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Review: Symmetric Multiprocessing

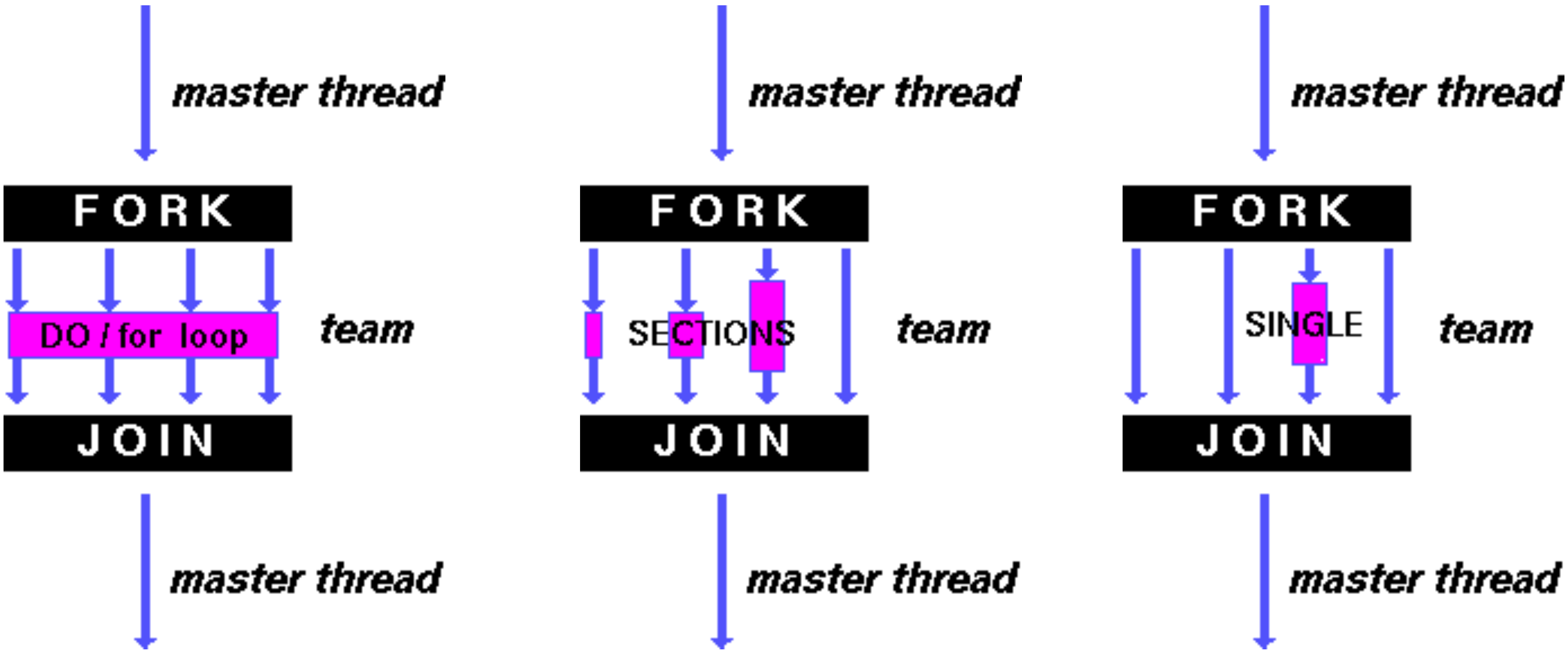


Review

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multithreading increases utilization, Multicore more processors (MIMD)
- Synchronization
  - atomic read-modify-write using load-linked/store-conditional
- OpenMP as simple parallel extension to C
  - Threads, Parallel for, private, critical sections, …
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble

Review: 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

Review: `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++) { … }
```

Review: Building Block: `for` loop

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

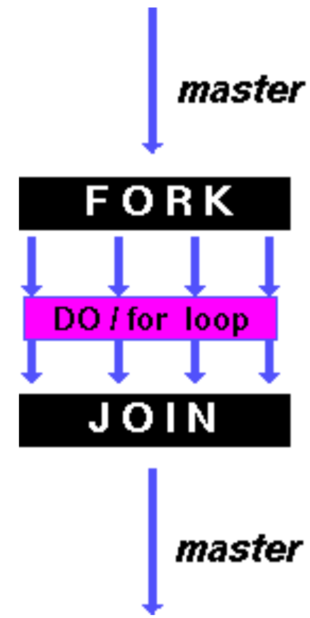
- Break *for loop* into chunks, and allocate each chunk to a separate thread
  - e.g. if `max` = 100 with 2 threads:
    assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple "shape" for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system 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

Review: 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 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?



Review: Matrix Multiply in OpenMP

```
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, i, j, k)
for (i = 0; i < Mdim; i++){
  for (j = 0; j < Ndim; j++){
    tmp = 0.0;
    for(k = 0; k < Pdim; k++){
      /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
      tmp += A[i*Pdim + k] * B[k*Ndim + j];
    }
    C[i*Ndim + j] = tmp;
  }
}
run_time = omp_get_wtime() - start_time;
```

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

Review: 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

New: OpenMP Directives (Synchronization)

- These are defined *within* a `parallel` section
- `master`
  - Code block executed only by the master thread (all other threads skip)
- `critical`
  - Code block executed by only one thread at a time
- `atomic`
  - Specific memory location must be updated atomically (like a mini-`critical` section for writing to memory)
  - Applies to single statement, not code block

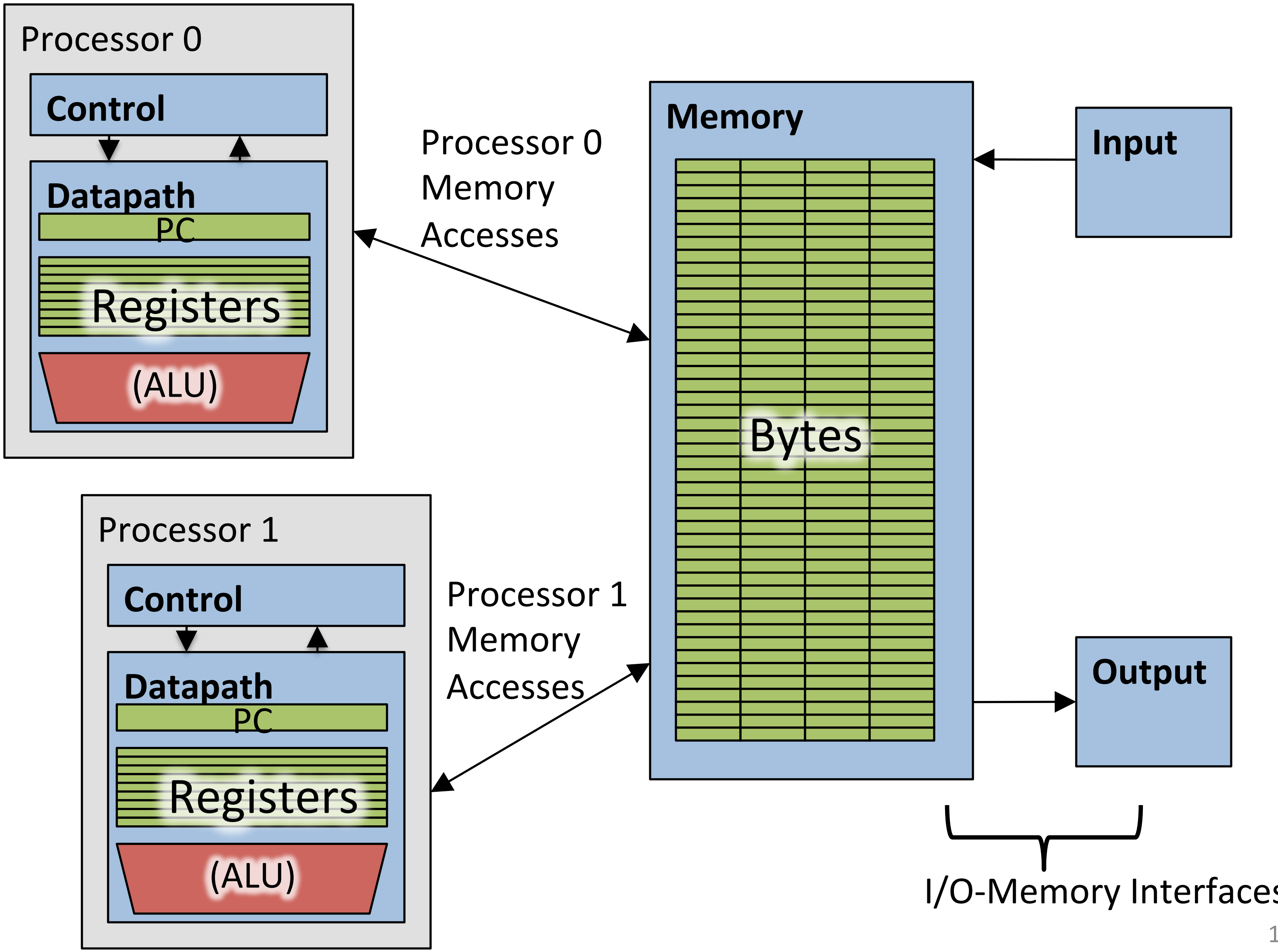
What's wrong with this code?

```
double compute_sum(double *a, int a_len) {
  double sum = 0.0;

  #pragma omp parallel for
  for (int i = 0; i < a_len; i++) {

    sum += a[i];
  }
  return sum;
}
```

Sample use of `critical`

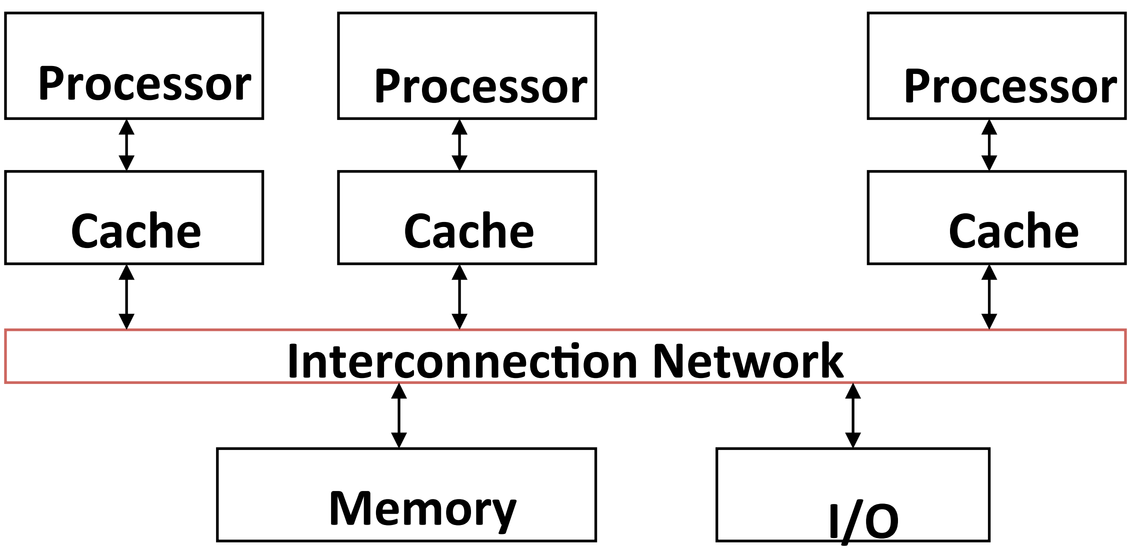
```
double compute_sum(double *a, int a_len) {
  double sum = 0.0;

  #pragma omp parallel for
  for (int i = 0; i < a_len; i++) {
    #pragma omp critical
    sum += a[i];
  }
  return sum;
}
```

Where are the caches?



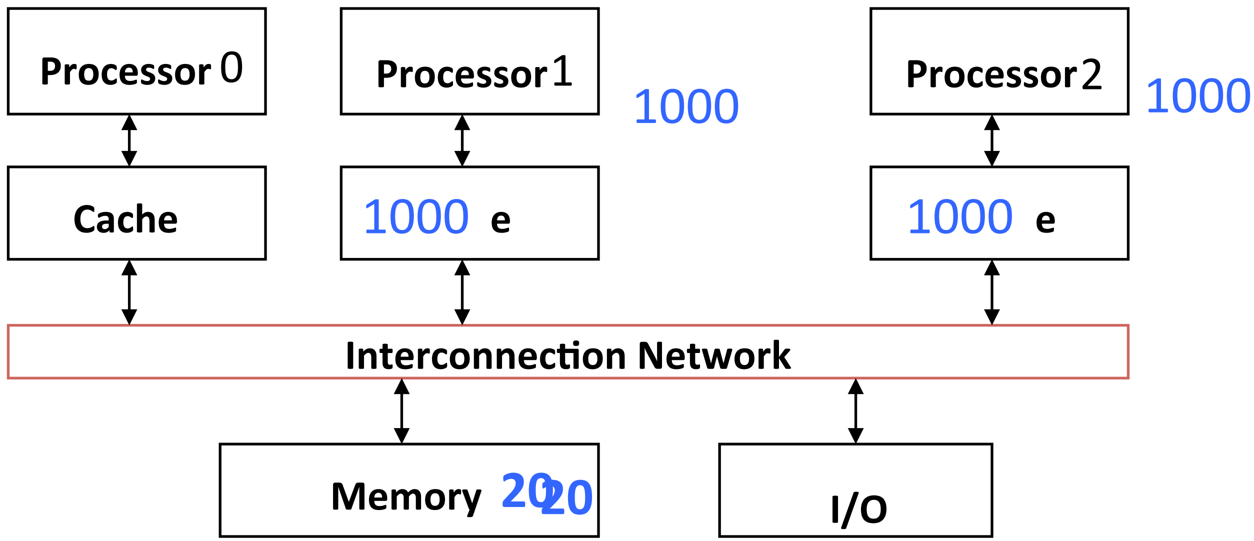
Multiprocessor Caches

- 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



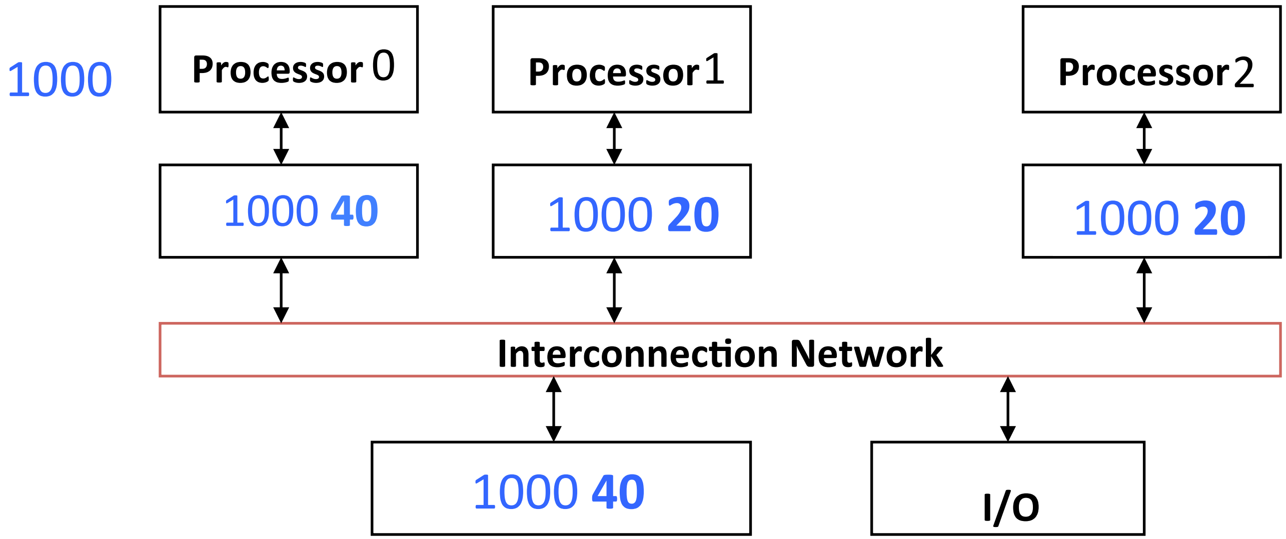
Shared Memory and Caches

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



Shared Memory and Caches

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



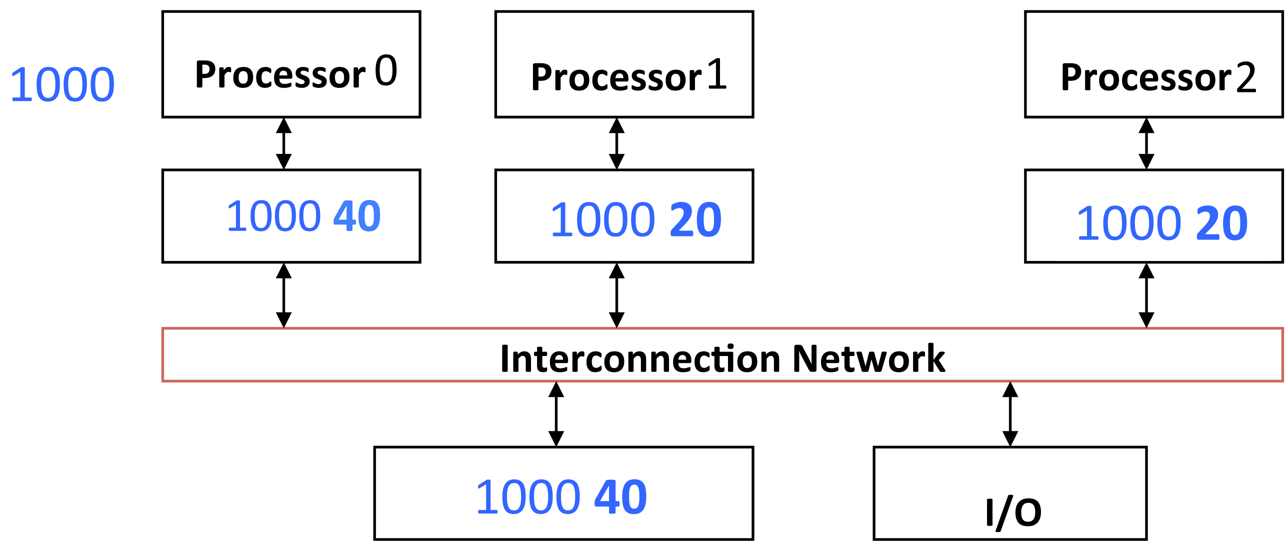
Problem?

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 "snoop" tags of other caches using common interconnect
  - Invalidate any "hits" to same address in other caches
  - If hit is to dirty line, other cache has to write back first!

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

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**

Administrivia

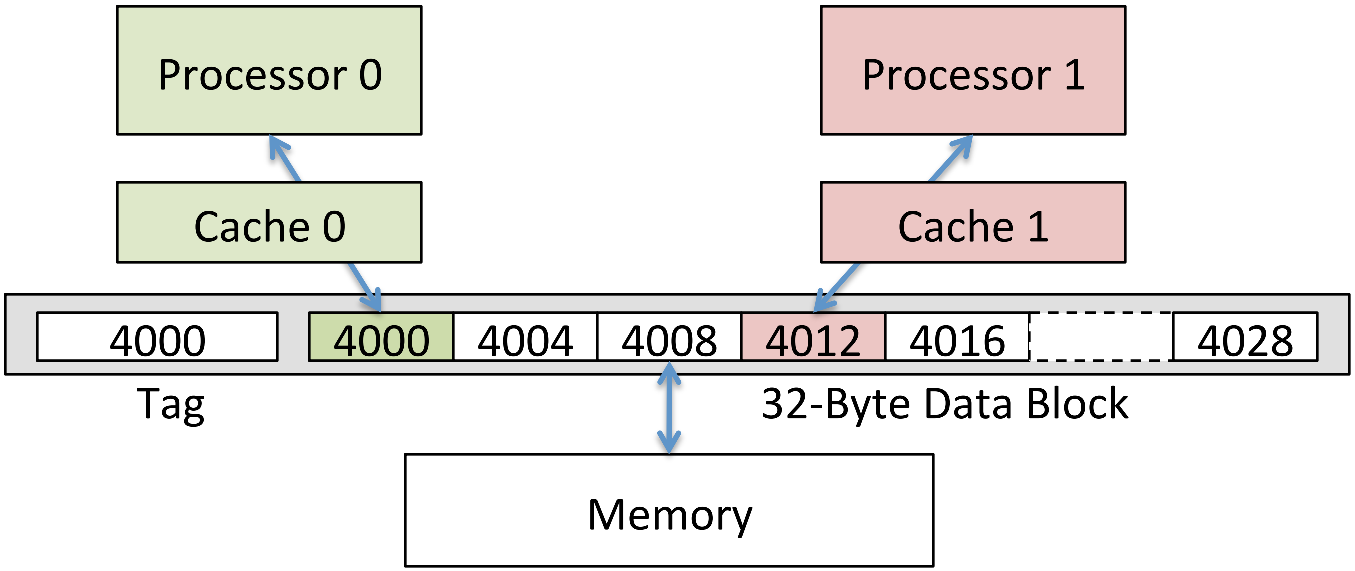
- Project 3-2 Out
- HW5 Out – Performance Programming
- Guerrilla Section on Performance Programming on Thursday, 5-7pm, Woz

Administrivia

- Midterm 2 is **tomorrow**!
  - In this room, at this time
  - Two double-sided 8.5”x11” handwritten cheatsheets
  - We’ll provide a MIPS green sheet
  - No electronics
  - Covers up to and including 07/21 lecture
  - Review session slides on Piazza

Break

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 Line

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

Understanding Cache Misses: The 3Cs

- Compulsory (cold start or process migration, 1st reference):
  - First access to a block in memory impossible to avoid; small effect for long running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- *Conflict (collision):*
  - *Multiple memory locations mapped to the same cache location*
  - *Solution 1: increase cache size*
  - *Solution 2: increase associativity (may increase access time)*

Rules for Determining Miss Type for a Given Access Pattern in 61C

1) Compulsory: A miss is compulsory if and only if it results from accessing the data for the first time. If you have ever brought the data you are accessing into the cache before, it's not a compulsory miss, otherwise it is.

2) Conflict: A conflict miss is a miss that's not compulsory and that would have been avoided if the cache was fully associative. Imagine you had a cache with the same parameters but fully-associative (with LRU). If that would've avoided the miss, it's a conflict miss.

3) Capacity: This is a miss that would not have happened with an infinitely large cache. If your miss is not a compulsory or conflict miss, it's a capacity miss.

Fourth "C" of Cache Misses: *Coherence* Misses

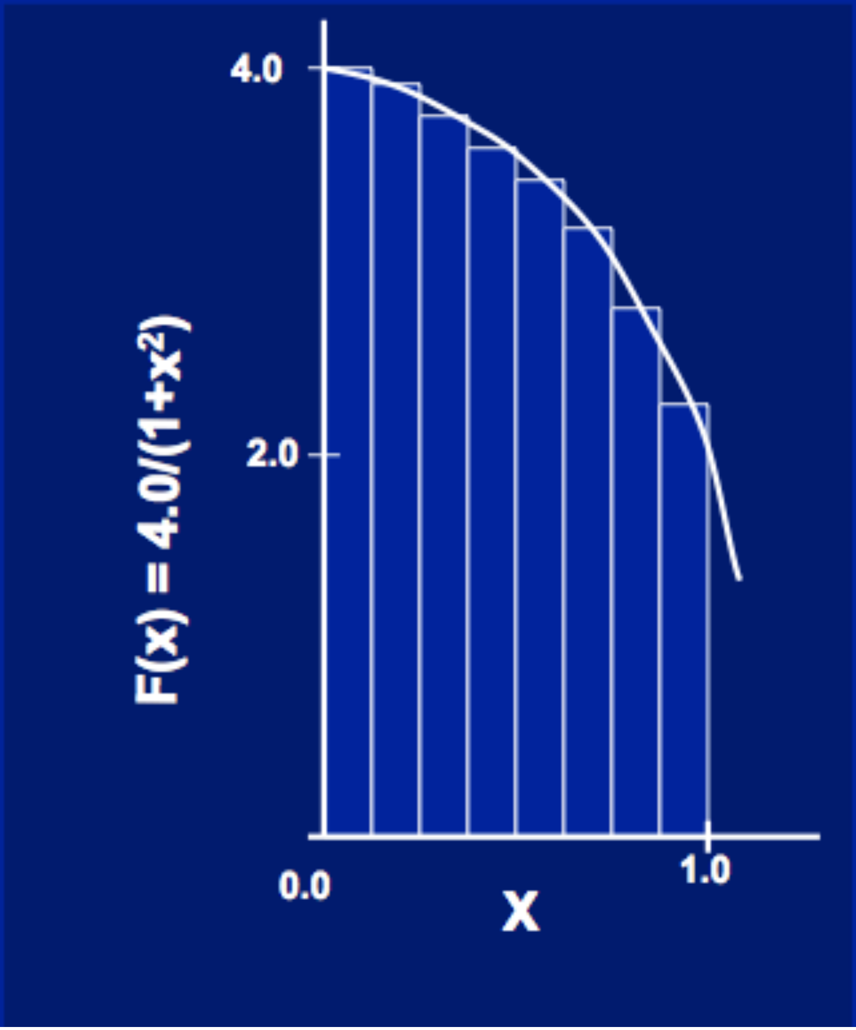
- Misses caused by coherence traffic with other processor
- Also known as *communication* misses because represents data moving between processors working together on a parallel program
- For some parallel programs, coherence misses can dominate total misses

π

3\.
14159265358979323846264338327950 2
88419716939937510582097494459230 7
81640628620899862803482534211706 7
98214808651328230664709384460955 0
58223172535940812848111745028410 2
...

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);
}
```

OpenMP Version (with bug)

```
#include <omp.h>
#define NUM_THREADS 2
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 (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=0, pi=0.0; i<NUM_THREADS; i++)
    pi += sum[i];
  printf ("pi = %6.12f\n", pi  / num_steps);
}
```

Experiment

- Run with NUM_THREADS = 1 multiple times
- Run with NUM_THREADS = 2 multiple times
- What happens?

OpenMP Version (with bug)

```
#include <omp.h>
#define NUM_THREADS 2
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 (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=0, pi=0.0; i<NUM_THREADS; i++)
    pi += sum[i];
  printf ("pi = %6.12f\n", pi  / num_steps);
}
```

Note: loop index variable `i` is shared between threads

Sum Reduction

- Sum 100,000 numbers on 100 processor SMP
  - Each processor has ID: $0 \leq Pn \leq 99$
  - Partition 1000 numbers per processor
  - Initial summation on each processor [Phase I]
  - Aka, "the map phase"

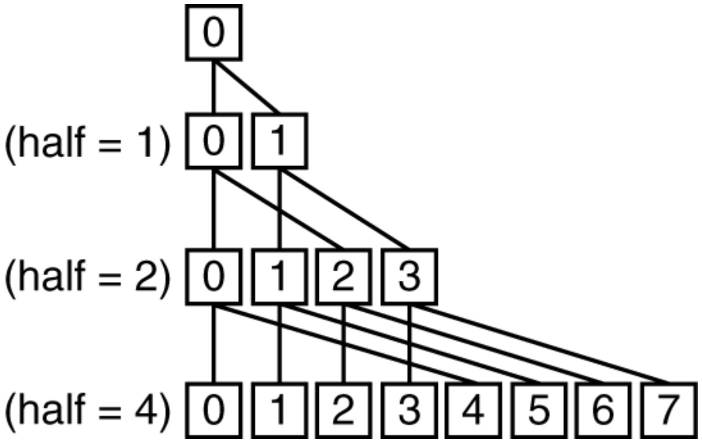
```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
     sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums [Phase II]
  - Reduction: divide and conquer in "the reduce phase"
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

Example: Sum Reduction

Second Phase:
After each processor has computed its "local" sum

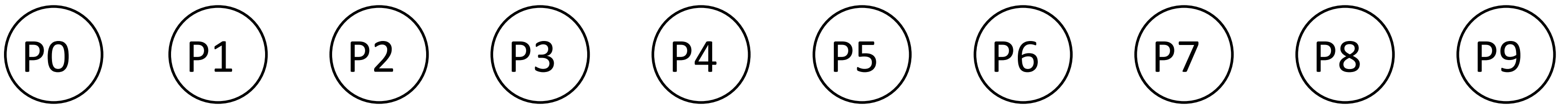
This code runs simultaneously on each core




```
half = 100;
repeat
   synch();
   /* Proc 0 sums extra element if there is one */
   if (half%2 != 0 && Pn == 0)
     sum[0] = sum[0] + sum[half-1];
   half = half/2; /* dividing line on who sums */
   if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

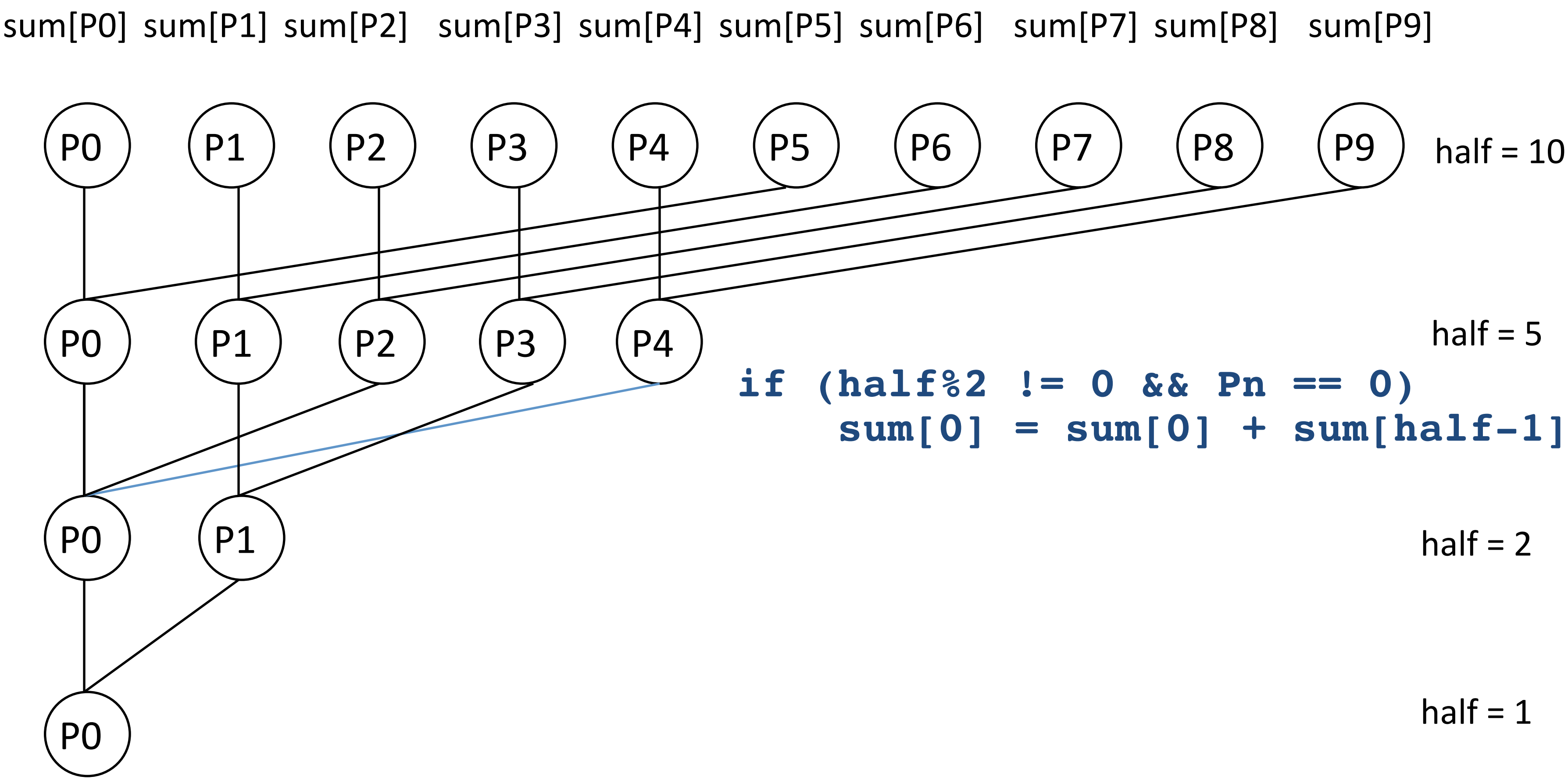
An Example with 10 Processors

sum[P0] sum[P1] sum[P2] sum[P3] sum[P4] sum[P5] sum[P6] sum[P7] sum[P8] sum[P9]



half = 10

An Example with 10 Processors




```
if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
```

OpenMP Reduction

- *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.

```
#pragma omp for reduction(+ : nSum)
for (i = START ; i <= END ; ++i)
  nSum += i;
```

OpenMP Reduction Version

```
#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);
}
```

Note: Don't have to declare for loop index variable `i` private, since that is default

OpenMP Pitfalls

- Unfortunately, we can't just throw pragmas on everything ☹
  - Data dependencies
  - Sharing issues (incorrectly marking private vars as non-private)
  - Updating shared values
  - Parallel Overhead

OpenMP Pitfall #1: Data Dependencies

- Consider the following code:

```
a[0] = 1;
for(i=1; i<5000; i++)
  a[i] = i + a[i-1];
```

- There are dependencies between loop iterations!
  - Splitting this loop between threads does not guarantee in-order execution
  - Out of order loop execution will result in undefined behavior (i.e. likely wrong result)

Open MP Pitfall #2: Sharing Issues

- Consider the following loop:

```
#pragma omp parallel for
for(i=0; i<n; i++){
   temp = 2.0*a[i];
   a[i] = temp;
   b[i] = c[i]/temp;
}
```

- `temp` is a shared variable!

```
#pragma omp parallel for private(temp)
for(i=0; i<n; i++){
   temp = 2.0*a[i];
   a[i] = temp;
   b[i] = c[i]/temp;
}
```

OpenMP Pitfall #3: Updating Shared Variables Simultaneously

- Now consider a global sum:

```
for(i=0; i<n; i++)
    sum = sum + a[i];
```

- This can be done by surrounding the summation by a `critical/atomic` section or `reduction` clause:

```
#pragma omp parallel for reduction(+:sum)
{
   for(i=0; i<n; i++)
       sum = sum + a[i];
}
```

  – Compiler can generate highly efficient code for `reduction`

OpenMP Pitfall #4: Parallel Overhead

- Spawning and releasing threads results in significant overhead
- Better to have fewer but larger parallel regions
  - Parallelize over the largest loop that you can (even though it will involve more work to declare all of the private variables and eliminate dependencies)

OpenMP Pitfall #4: Parallel Overhead

```
start_time = omp_get_wtime();
for (i=0; i<Ndim; i++){
  for (j=0; j<Mdim; j++){
    tmp = 0.0;
    #pragma omp parallel for reduction(+:tmp)
      for( k=0; k<Pdim; k++){
        /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
        tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));
      }
    *(C+(i*Ndim+j)) = tmp;
  }
}
run_time = omp_get_wtime() - start_time;
```

Too much overhead in thread generation to have this statement run this frequently.

Poor choice of loop to parallelize.

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, critical sections, reductions …
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble