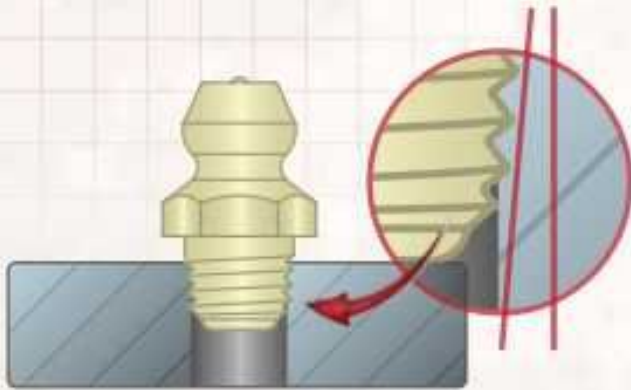


Great Ideas in Computer Architecture

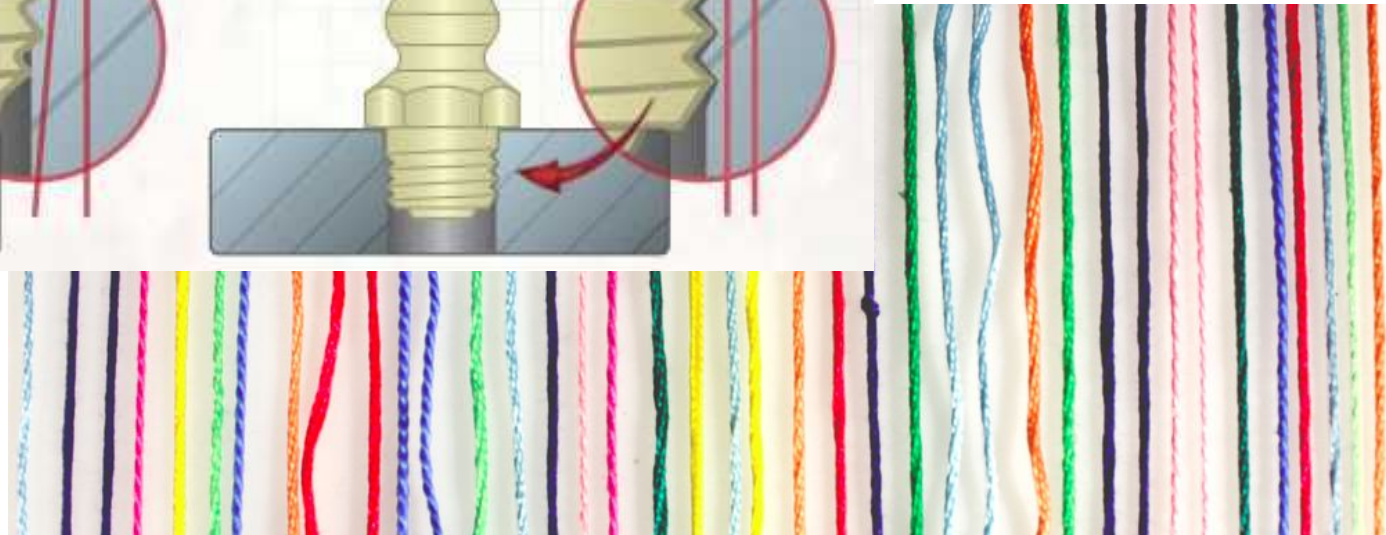
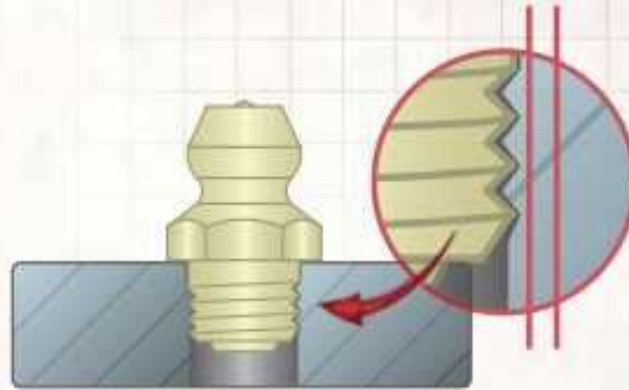
Thread Level Parallelism and OpenMP

Instructor: Steven Ho

TAPER THREADS



PARALLEL THREADS



Review

- Intel SSE SIMD Instructions
 - Embed the SSE machine instructions directly into C programs through use of **intrinsics**
- Loop Unrolling: Access more of array in each iteration of a loop (no hardware support needed)
- Amdahl's Law limits benefits of parallelization
- Multithreading increases utilization
- Multicore more processors (MIMD) -- use shared memory (single address space)

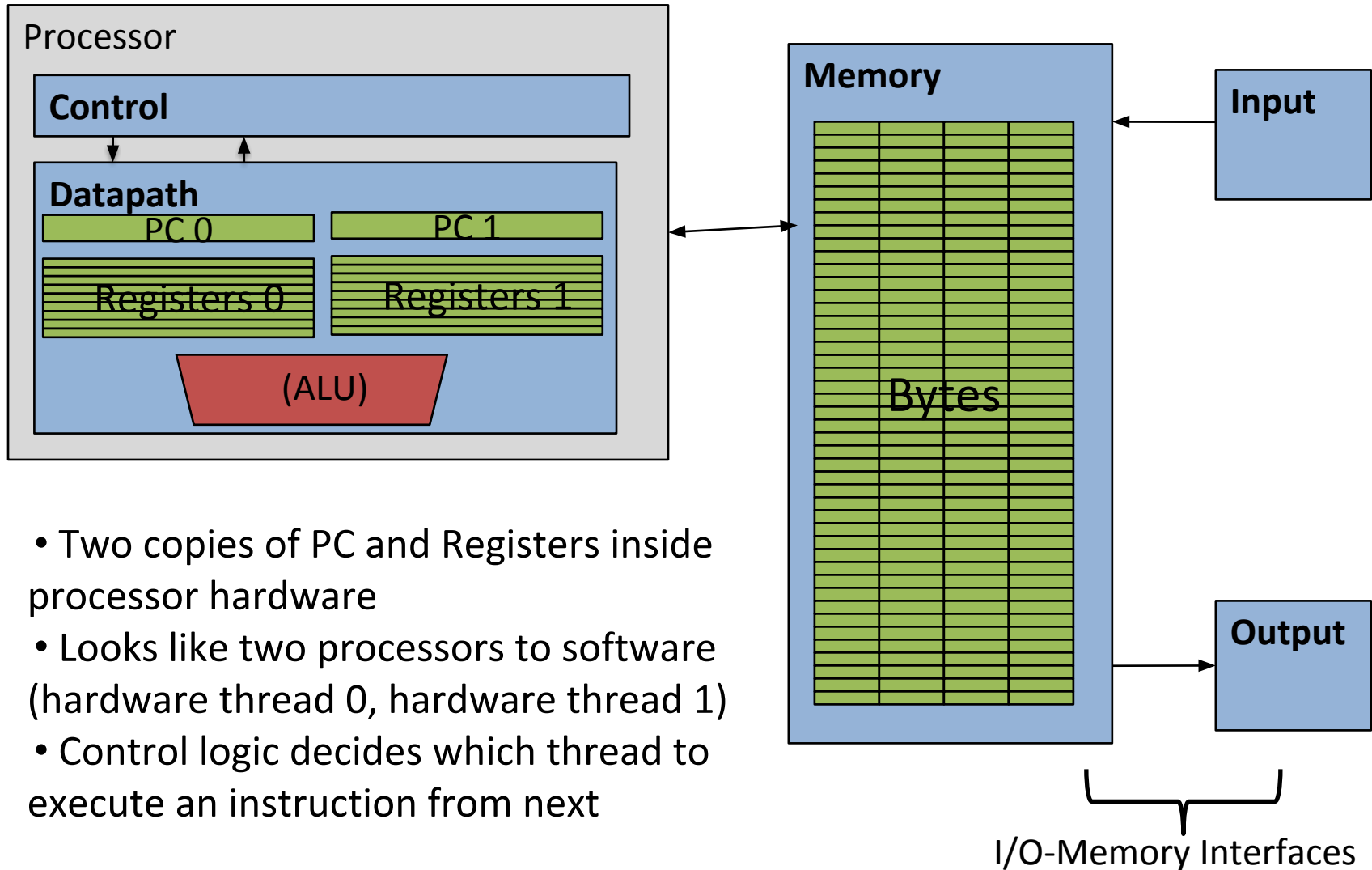
SSE = SIMD; multicore = MIMD

loop unrolling : SIMD :: multithreading : MIMD

Review: Threads

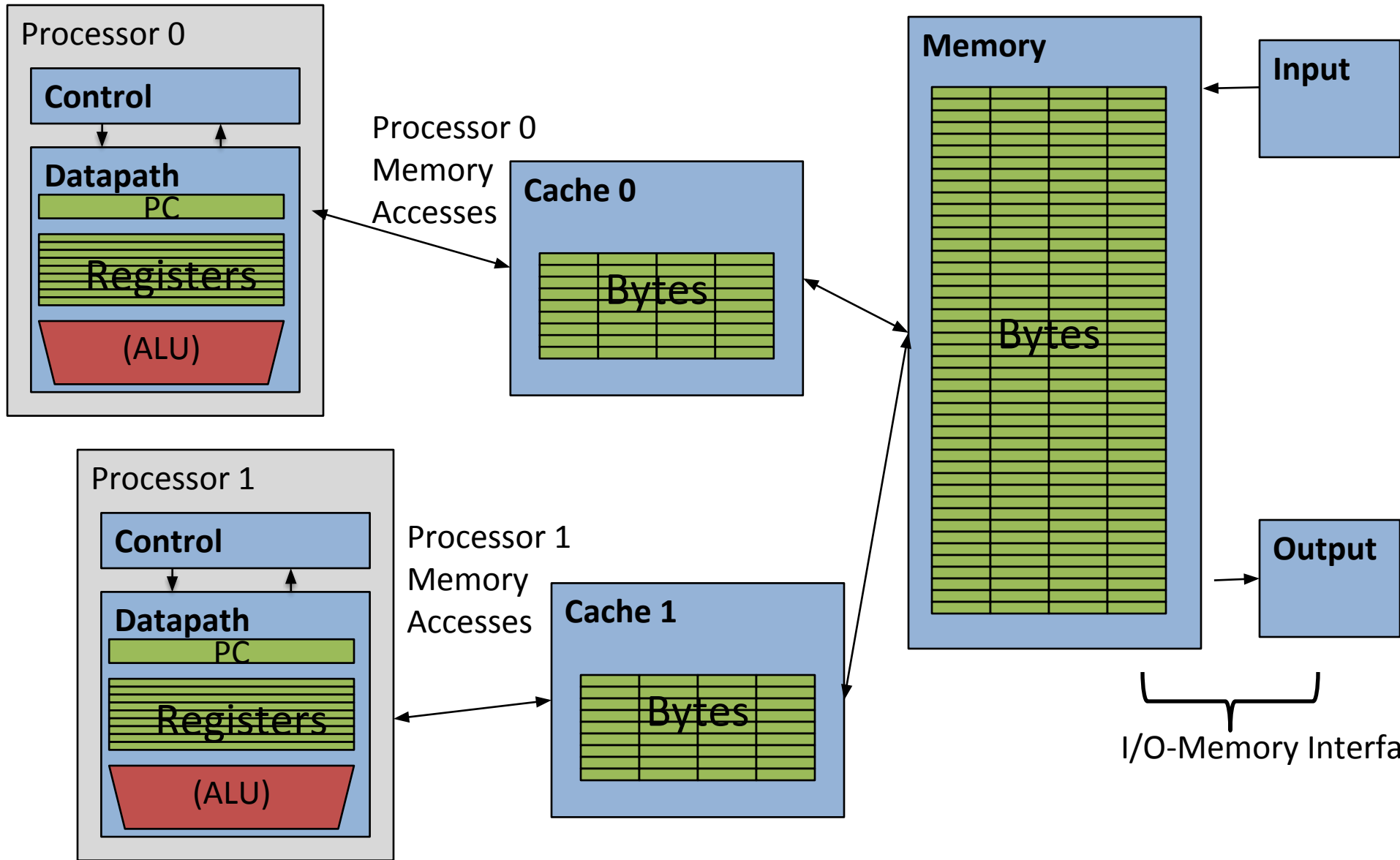
- *Thread of execution*: Smallest unit of processing scheduled by operating system
- On uniprocessor, multithreading occurs by *time-division multiplexing*
 - Processor switches between different threads
 - *Context switching* happens frequently enough user perceives threads as running at the same time
 - OS must share the processor with processes -- need a hardware way of regaining control
- On a multiprocessor, threads run at the same time, with each processor running a thread

Review: Hardware Support for Multithreading



- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next

Review: Symmetric Multiprocessing



Review: Multithreading vs. Multicore

- Multithreading => Better Utilization
 - $\approx 1\%$ more hardware, 1.10X better performance?
 - Share integer adders, floating point adders, caches (L1 I\$, L1 D\$, L2 cache, L3 cache), Memory Controller
- Multicore => Duplicate Processors
 - $\approx 50\%$ more hardware, $\approx 2X$ better performance?
 - Share some caches (L2 cache, L3 cache), Memory Controller
- **Modern machines do both**
 - **Multiple cores with multiples threads per core**

Review: Multiprocessor Systems (MIMD)

- **Multiprocessor (MIMD):** a computer system with at least 2 processors or *cores* (“multicore”)
 - Each core has its own PC and executes an independent instruction stream
 - Processors share the same memory space and can communicate via loads and stores to common locations
- 1. Deliver high throughput for independent jobs via request-level or task-level parallelism
- 2. *Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program*

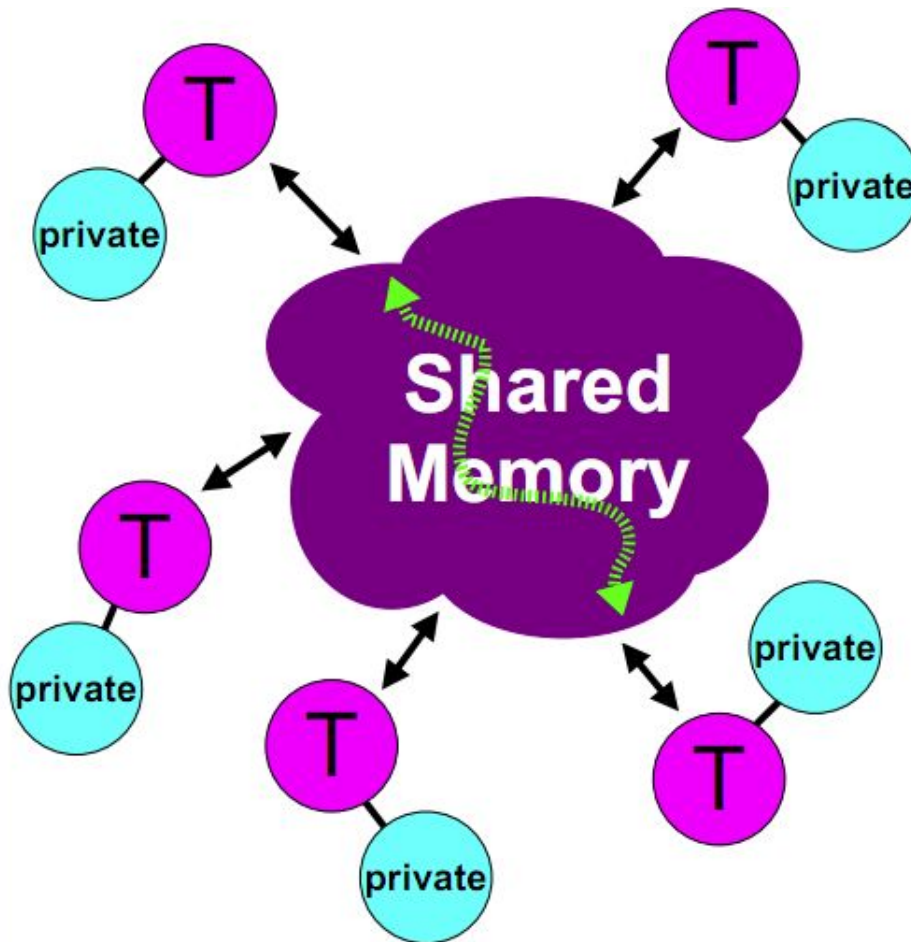
Agenda

- **Multiprocessor Systems**
- Synchronization - A Crash Course
- OpenMP Introduction
- OpenMP Directives
 - Workshare
 - Synchronization
- Bonus: Common OpenMP Pitfalls

Shared Memory Multiprocessor (SMP)

- Single physical address space shared by all processors
- 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*

Memory Model for Multi-threading



- ✓ All threads have access to the same, globally shared, memory
- ✓ Data can be shared or private
- ✓ Shared data is accessible by all threads
- ✓ Private data can only be accessed by the thread that owns it
- ✓ Data transfer is transparent to the programmer
- ✓ Synchronization takes place, but it is mostly implicit

Can be specified in a language with MIMD support – such as **OpenMP**

Data Races and Dependencies

- Thread scheduling is **non-deterministic**
 - Two memory accesses form a *data race* if different threads access the same location, and at least one is a write, and they occur one after another
 - If data dependencies exist between steps that are assigned to different threads, they could be run out of order
- Avoid incorrect results by:
 - 1) *synchronizing* writing and reading to get deterministic behavior
 - 2) not writing to the same memory address

Example: Sum Reduction (1/2)

- Sum 100,000 numbers on 100 processor SMP
 - Each processor has ID: $0 \leq P_n \leq 99$
 - Partition 1000 numbers per processor

- **Step 1:** Initial summation on *each* processor

```
sum[Pn] = 0;
```

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

- no data dependencies so far
- not writing to same address in memory

Example: Sum Reduction (1/2)

- Sum 100,000 numbers on 100 processor SMP
 - Each processor has ID: $0 \leq P_n \leq 99$
 - Partition 1000 numbers per processor
- **Step 1:** Initial summation on *each* processor

```
sum[Pn] = 0;
for (i=1000*Pn; i<1000*(Pn+1); i++)
    sum[Pn] = sum[Pn] + A[i];
```
- **Step 2:** Now need to add these partial sums
 - *Reduction*: divide and conquer approach to sum
 - Half the processors add pairs, then quarter, ...
 - Data dependencies: Need to synchronize between reduction steps

Example: Sum Reduction (2/2)

This is **Step 2**, after all “local” sums computed.
This code runs simultaneously on all processors.

```
half = 100;
```

```
repeat
```

```
  synch();
```

```
  if (half%2 != 0 && Pn == 0)
```

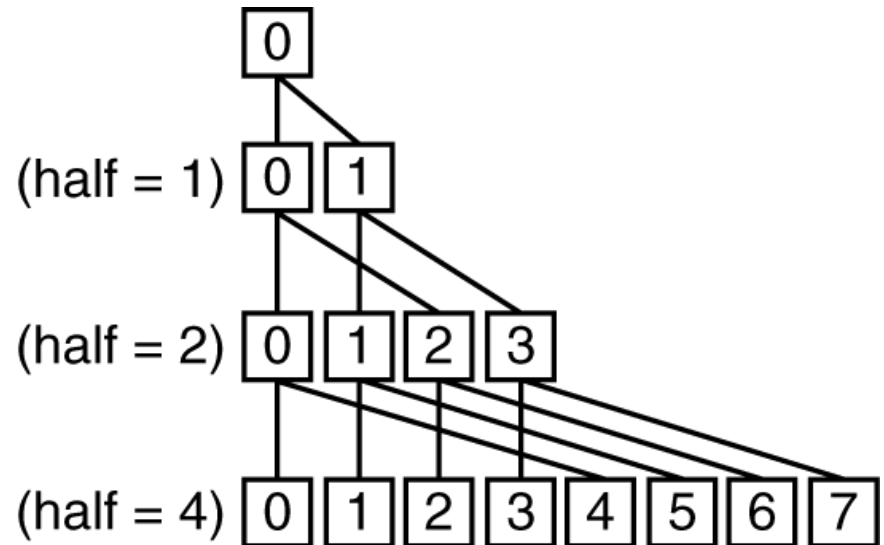
```
    sum[0] = sum[0] + sum[half-1];
```

```
    /* When half is odd, P0 gets extra element */
```

```
    half = half/2; /* dividing line on who sums */
```

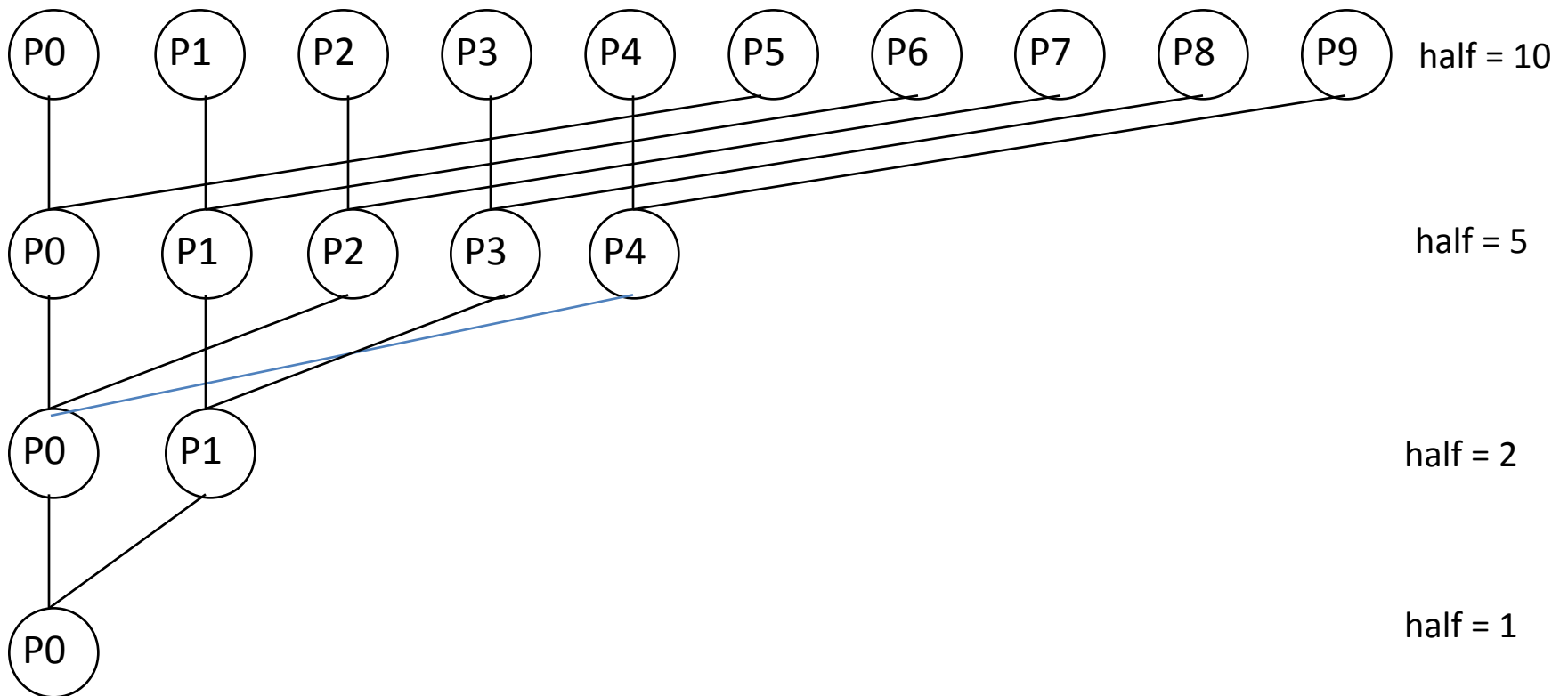
```
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
```

```
until (half == 1);
```



Sum Reduction with 10 Processors

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



Question: Given the Sum Reduction code, are the given variables Shared data or Private data?

```

half = 100;
repeat
    synch ();
    ... /* handle odd elements */
    half = half/2; /* dividing line */
    if (Pn < half)
        sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);

```

	half	sum	Pn
(A)	Shared	Shared	Shared
(B)	Shared	Shared	Private
(C)	Private	Shared	Private
(D)	Private	Private	Shared

Question: Given the Sum Reduction code, are the given variables Shared data or Private data?

```

half = 100;
repeat
    synch ();
    ... /* handle odd elements */
    half = half/2; /* dividing line */
    if (Pn < half)
        sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);

```

	half	sum	Pn
(A)	Shared	Shared	Shared
(B)	Shared	Shared	Private
(C)	Private	Shared	Private
(D)	Private	Private	Shared

Administrivia

- HW5 due tonight, HW6 Released!
- Midterm 2 is coming up! Wed. in lecture
 - Covering up to Performance
 - There will be discussion after MT2 :(
 - Check out Piazza for more logistics
- Proj4 Released soon!
- Guerilla session is probably being moved from Wed., but stay tuned...

Agenda

- Multiprocessor Systems
- **Synchronization - A Crash Course**
- OpenMP Introduction
- OpenMP Directives
 - Workshare
 - Synchronization
- Bonus: Common OpenMP Pitfalls

Data Races and Synchronization

- Two memory accesses form a *data race* if different threads access the same location, and at least one is a write, and they occur one after another
 - Means that the result of a program can vary depending on chance (which thread ran 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

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  if locked Can loop/idle here
Set the lock
Critical section
(e.g. change shared variables)
Unset the lock

Synchronization with Locks

```
// wait for lock released
while (lock != 0) ;
// lock == 0 now (unlocked)

// set lock
lock = 1;

    // access shared resource ...
    // e.g. pi
    // sequential execution! (Amdahl ...)
```

```
// release lock
lock = 0;
```


Lock Synchronization

Thread 1

```
while (lock != 0)
```

```
lock = 1;
```

```
// critical section
```

```
lock = 0;
```



```
lock != 0) ;
```

lock not set,

before thread 1 sets it

- Both threads believe they got and set the lock!

Try as you like, this problem has no solution, not even at the assembly level. Unless we introduce new instructions, that is!

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)

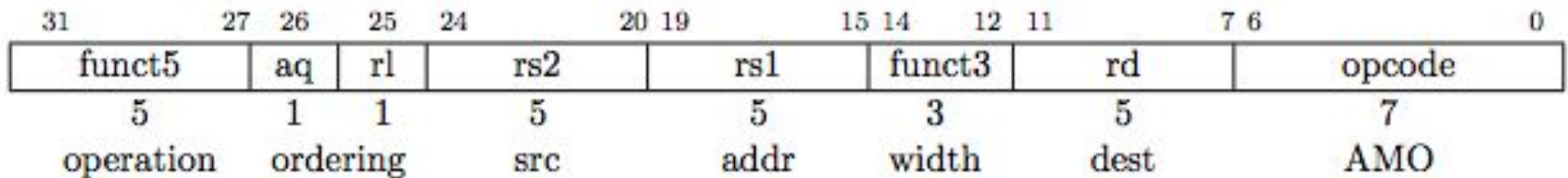
- Common implementations:

- Atomic swap of register \leftrightarrow 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

RISCV 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



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 and rl insure *in order* execution

```
amoadd.w rd,rs2,(rs1):
```

```
t = M[x[rs1]];
```

```
x[rd] = t;
```

```
M[x[rs1]] = t + x[rs2]
```

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 already 1, another
                                     # thread has lock, so
                                     # we must try again
... critical section goes here ...
amoswap.w.rl x0, x0, (a0) # store 0 in lock to
                          # release
```

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)
```

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

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

Meet the Staff



	Damon	Jon
Trash TV Show	Starcraft II Streams	League of Legends Streams
Favorite Plot Twist	Fight Club	
Best Bathroom in Cal	Soda Hall <3	Jacobs Hall
Best Study Spot	Kresge	My Apartment

Agenda

- Multiprocessor Systems
- Synchronization - A Crash Course
- **OpenMP Introduction**
- **OpenMP Directives**
 - Workshare
 - Synchronization
- **Bonus: Common OpenMP Pitfalls**

OpenMP

- API used for multi-threaded, shared memory parallelism
 - Compiler Directives
 - Runtime Library Routines
 - Environment Variables
- Portable
- Standardized
- Resources: <http://www.openmp.org/>
and <http://computing.llnl.gov/tutorials/openMP/>

Summary of
OpenMP 3.0
C/C++ Syntax



Download the full OpenMP API Specification at www.openmp.org.

Directives

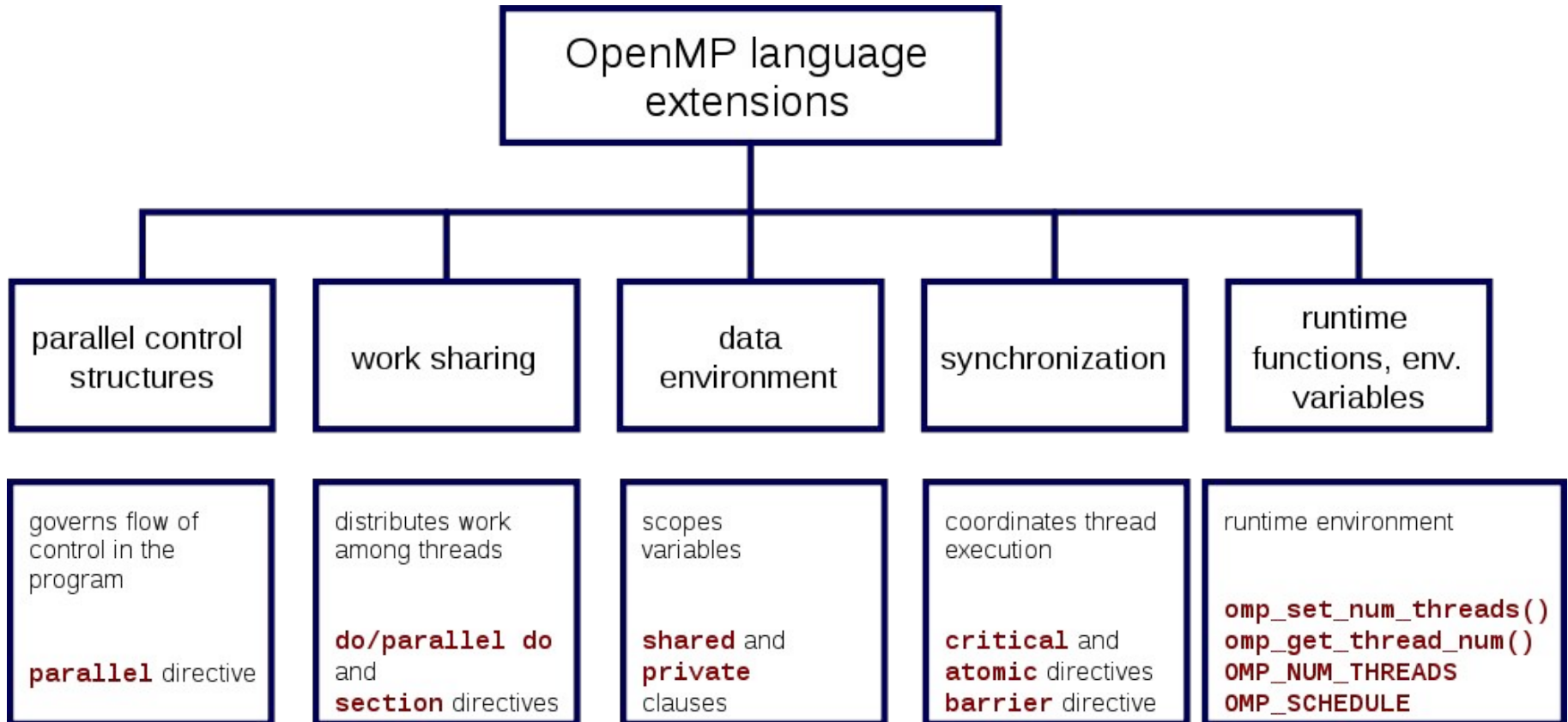
An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

The `parallel` construct forms a team of threads and starts parallel execution.

```
#pragma omp parallel [clause[ [, ]clause]...] new-line  
structured-block
```

```
clause: if (scalar-expression)  
num_threads (integer-expression)  
default (shared | none)  
private (list)  
firstprivate (list)  
shared (list)  
copyin (list)  
reduction (operator: list)
```

OpenMP Specification



Shared Memory Model with Explicit Thread-based Parallelism

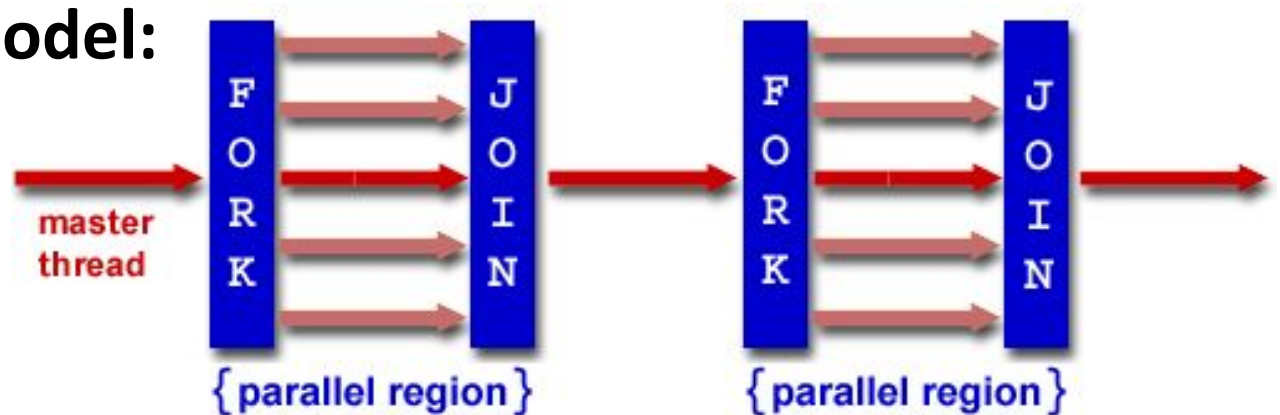
- Multiple threads in a shared memory environment, explicit programming model with full programmer control over parallelization
- **Pros:**
 - Takes advantage of shared memory, programmer need not worry (that much) about data placement
 - Compiler directives are simple and easy to use
 - Legacy serial code does not need to be rewritten
- **Cons:**
 - Code can only be run in shared memory environments
 - Compiler must support OpenMP (e.g. gcc 4.2)

OpenMP in CS61C

- OpenMP is built on top of C, so you don't have to learn a whole new programming language
 - Make sure to add `#include <omp.h>`
 - Compile with flag: `gcc -fopenmp`
 - Mostly just a few lines of code to learn
- You will NOT become experts at OpenMP
 - Use slides as reference, will learn to use in lab
- **Key ideas:**
 - Shared vs. Private variables
 - OpenMP directives for parallelization, work sharing, synchronization

OpenMP Programming Model

- **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

OpenMP Extends C with Pragmas


- *Pragmas* are a preprocessor mechanism C provides for language extensions
- Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes (not covered in 61C)
- Good mechanism for OpenMP -- compilers that don't recognize a pragma just ignore them
 - Runs on sequential computer even with embedded pragmas

parallel Pragma and Scope

- Basic OpenMP construct for parallelization:

```
#pragma omp parallel
```

```
{
```

 This is annoying, but curly brace MUST go on separate line from #pragma

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

```
}
```

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

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


Thread Creation

- Defined by **OMP_NUM_THREADS** environment variable (or code procedure call)
 - Set this variable to the *maximum* number of threads you want OpenMP to use
- Usually equals the number of cores in the underlying hardware on which the program is run
 - But remember thread \neq core

OMP_NUM_THREADS

- OpenMP intrinsic to set number of threads:

```
omp_set_num_threads(x);
```

- OpenMP intrinsic to get number of threads:

```
num_th = omp_get_num_threads();
```

- OpenMP intrinsic to get Thread ID number:

```
th_ID = omp_get_thread_num();
```

Parallel Hello World

```
#include <stdio.h>
#include <omp.h>
int main () {
    int nthreads, tid;

    /* Fork team of threads with private var tid */
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num(); /* get thread id */
        printf("Hello World from thread = %d\n", tid);

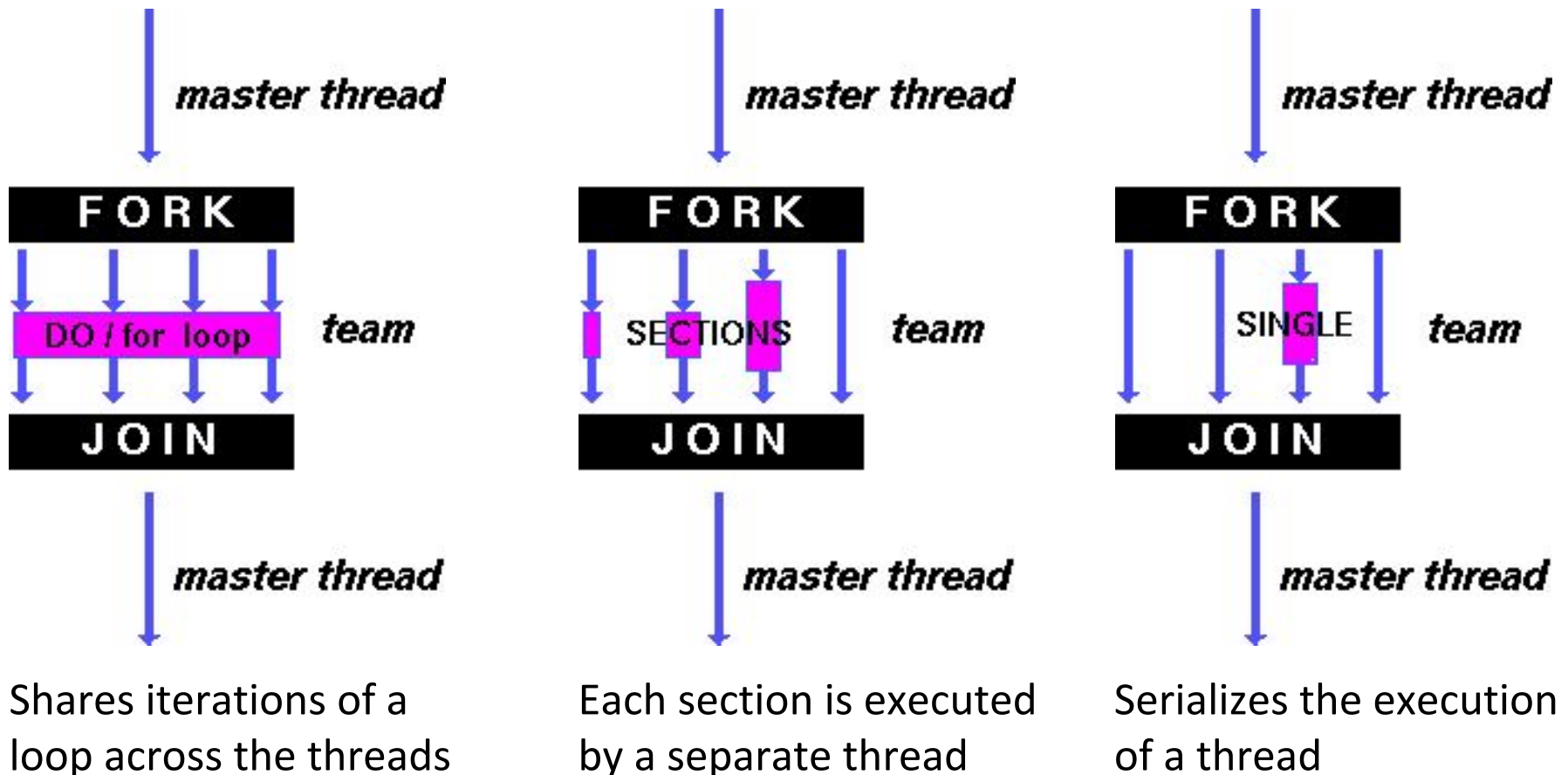
        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master and terminate */
}
```

Agenda

- Multiprocessor Systems
- Synchronization - A Crash Course
- OpenMP Introduction
- **OpenMP Directives**
 - **Workshare**
 - Synchronization
- **Bonus: Common OpenMP Pitfalls**

OpenMP Directives (Work-Sharing)

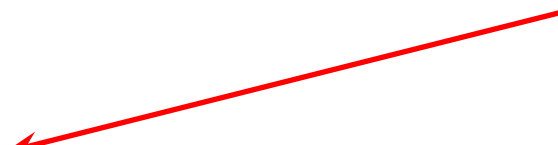
- These are defined *within* a `parallel` section



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++) { ... }
```

Building Block: `for` loop

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

- Break *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

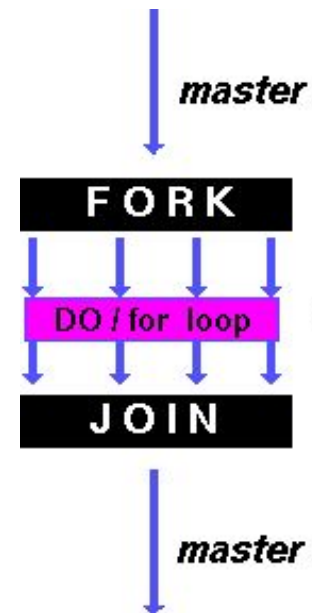
Parallel `for` pragma

```
#pragma omp parallel for
```

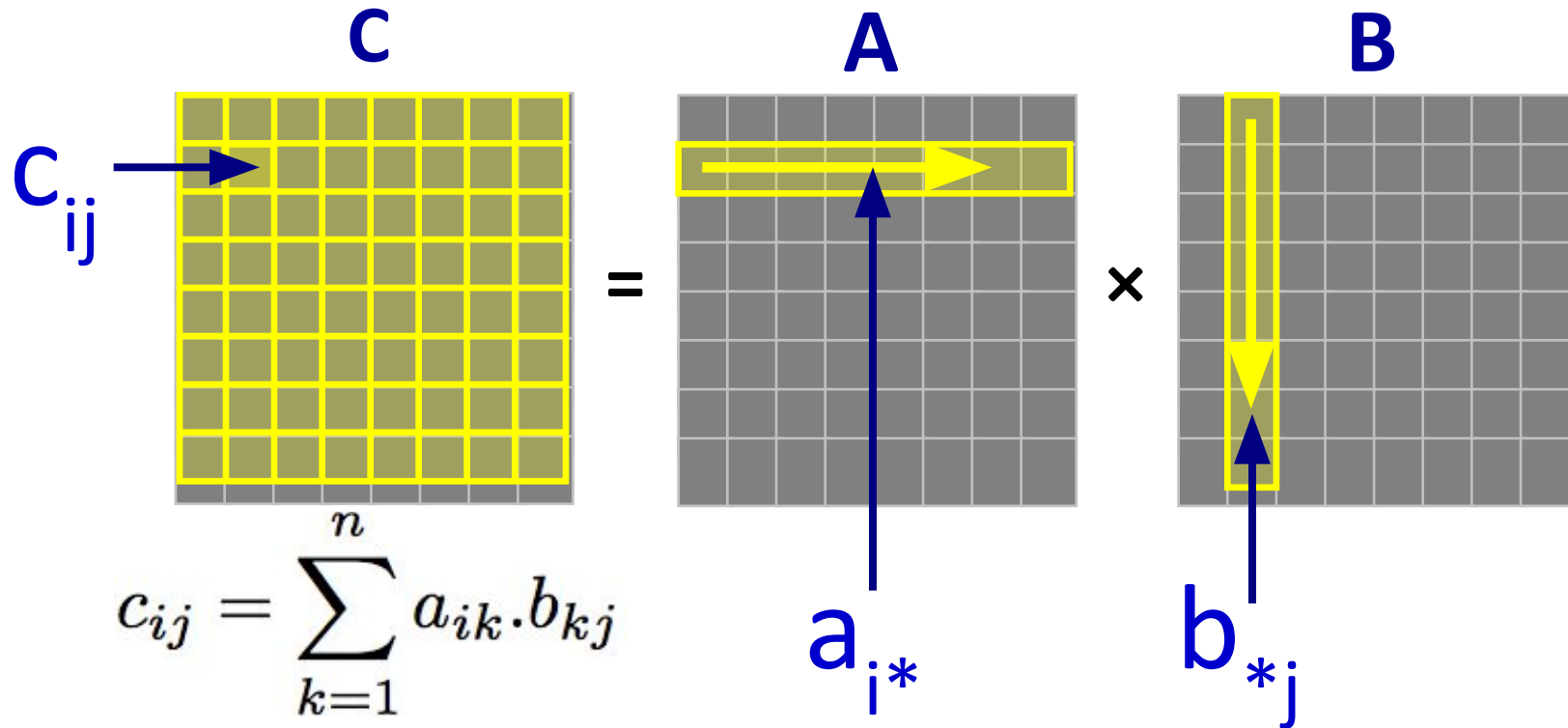
```
for (i=0; i<max; i++) zero[i] = 0;
```

- 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?)
- 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?
- Implicit synchronization at end of for loop

Cache blocking -- better AMAT



Matrix Multiplication



Naïve Matrix Multiply

```
for (i=0; i<N; i++)  
  for (j=0; j<N; j++)  
    for (k=0; k<N; k++)  
      c[i][j] += a[i][k] * b[k][j];
```

Advantage: Code simplicity

Disadvantage: Blindly marches through memory (how does this affect the cache?)

Matrix Multiply in OpenMP

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

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

```
    for (i=0; i<Mdim; i++){ ← Outer loop spread  
        for (j=0; j<Ndim; j++){ across N threads;  
            tmp = 0.0; inner loops inside a  
            for( k=0; k<Pdim; k++){ single thread  
                /* 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;
```

Why is there no data race here?

- Different threads only work on different ranges of i -- inside writing memory access
- Never reducing to a single value.

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*)
 - Improve algorithm by reducing computations and memory accesses in code (what happens in each loop?)

Agenda

- Multiprocessor Systems
- Synchronization - A Crash Course
- OpenMP Introduction
- **OpenMP Directives**
 - Workshare
 - **Synchronization**
- **Bonus: Common OpenMP Pitfalls**

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;  
}
```

OpenMP Synchronization Directives

- 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

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;
}
```


Sample use of critical

```
double compute_sum(double *a, int a_len) {  
    double sum = 0.0;  
    #pragma omp parallel  
    {  
        double local_sum;  
        #pragma omp for  
        for (int i = 0; i < a_len; i++) {  
            local_sum += a[i];  
        }  
        #pragma omp critical  
        sum += local_sum  
    }  
    return sum;  
}
```

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)`
 - **Operation**: perform on the variables (var) at the *end* of the parallel region
 - **Var**: variable(s) on which to perform scalar reduction

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

Agenda

- Multiprocessor Systems
- Synchronization - A Crash Course
- OpenMP Introduction
- OpenMP Directives
 - Workshare
 - Synchronization
- **Bonus: Common OpenMP Pitfalls**

OpenMP Pitfalls

- We can't just throw pragmas on everything and expect performance increase 😞
 - Might not change speed much or break code!
 - Must understand application and use wisely
- Discussed here:
 - 1) Data dependencies
 - 2) Sharing issues (private/non-private variables)
 - 3) Updating shared values
 - 4) 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.

Summary

- Synchronization via hardware primitives:
 - MIPS does it with Load Linked + Store Conditional
- OpenMP as simple parallel extension to C
 - Small, so easy to learn, but not very high level
 - Synchronization accomplished with critical/atomic/reduction