CS 61C: Great Ideas in Computer Architecture

Lecture 19: *Thread-Level Parallelism and OpenMP Intro*

Instructor: Sagar Karandikar
sagark@eecs.berkeley.edu

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

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Review

- Amdahl's Law: Serial sections limit speedup
- Flynn Taxonomy
- Intel SSE SIMD Instructions
  - Exploit data-level parallelism in loops
  - One instruction fetch that operates on multiple operands simultaneously
  - 128-bit XMM registers
- SSE Instructions in C
  - Embed the SSE machine instructions directly into C programs through use of intrinsics
  - Achieve efficiency beyond that of optimizing compiler

New-School Machine Structures (It's a bit more complicated!)

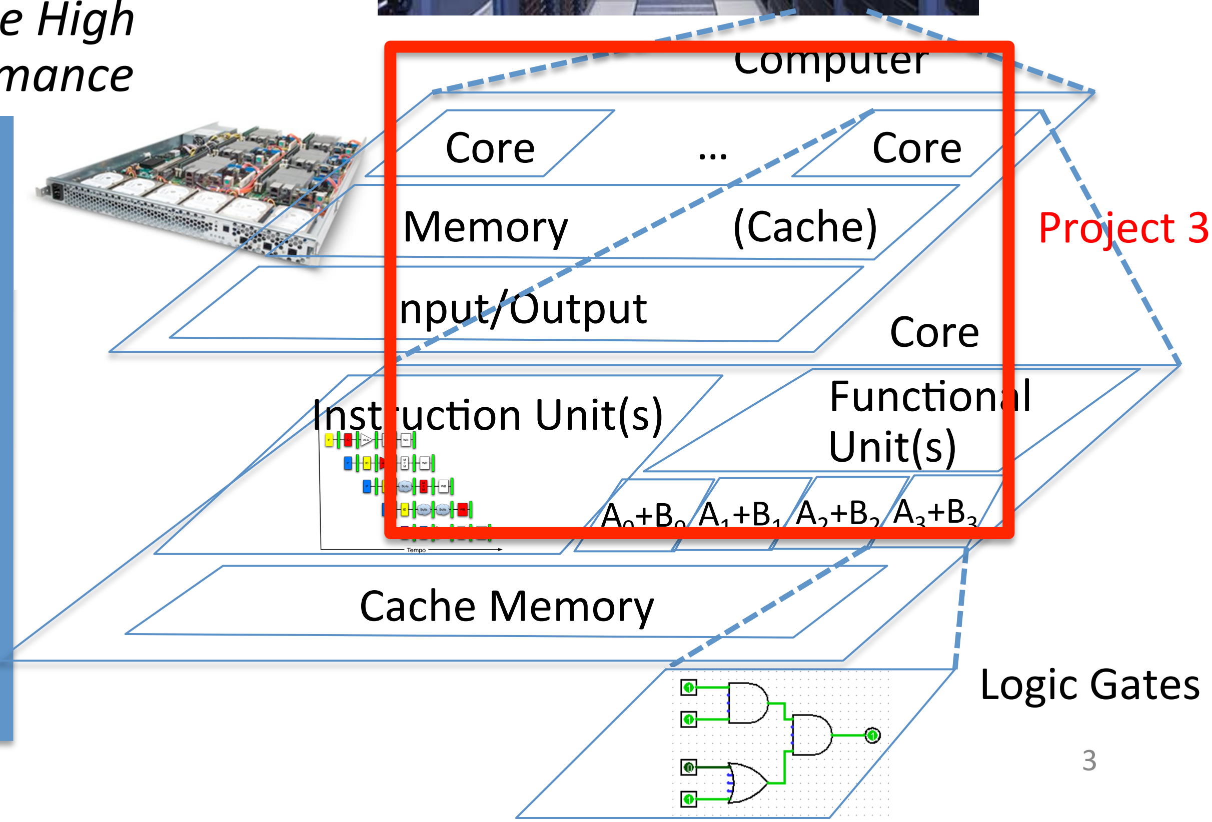
Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

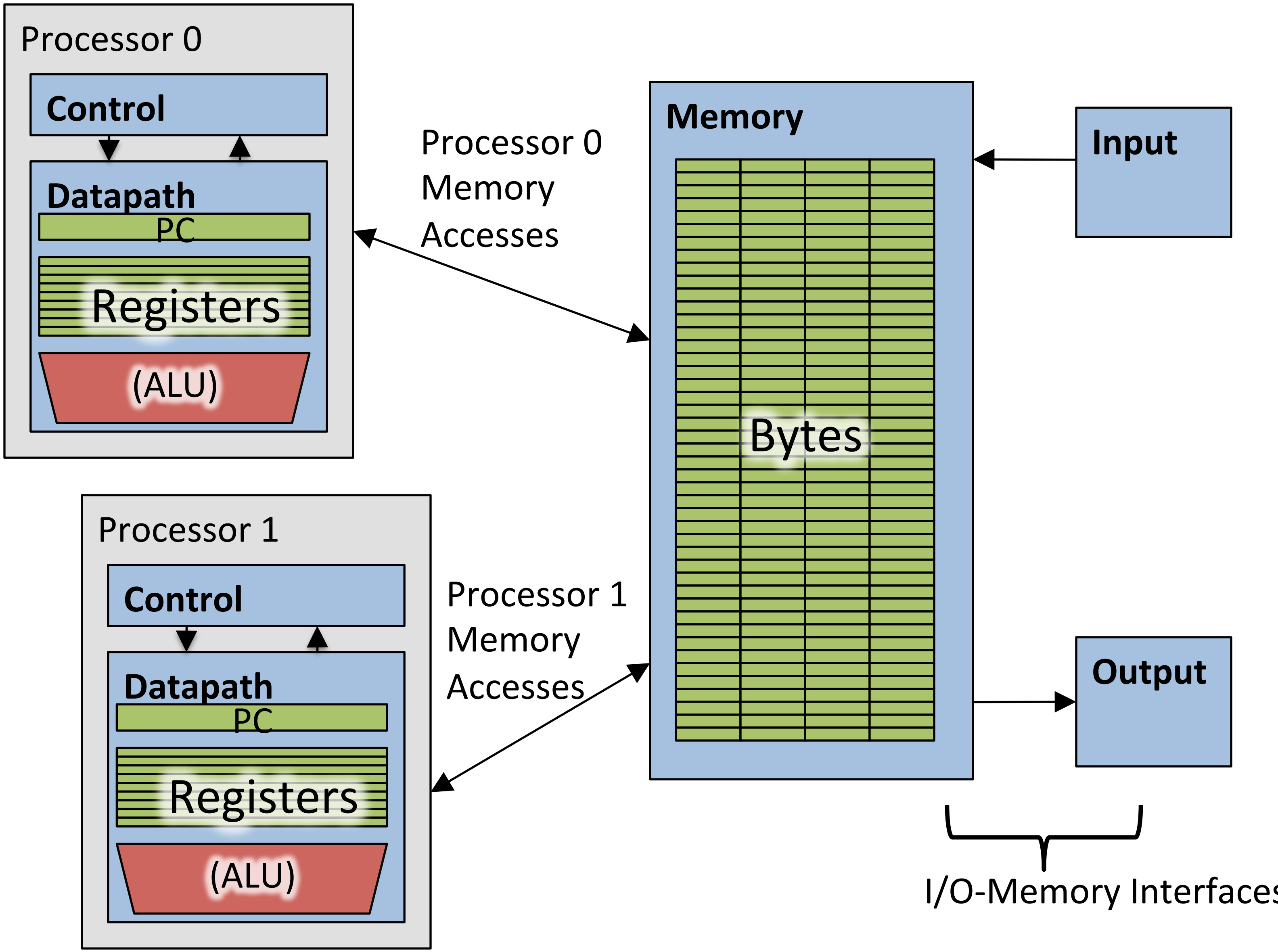
Harness Parallelism & Achieve High Performance

Warehouse Scale Computer

Smart Phone



Simple Multiprocessor

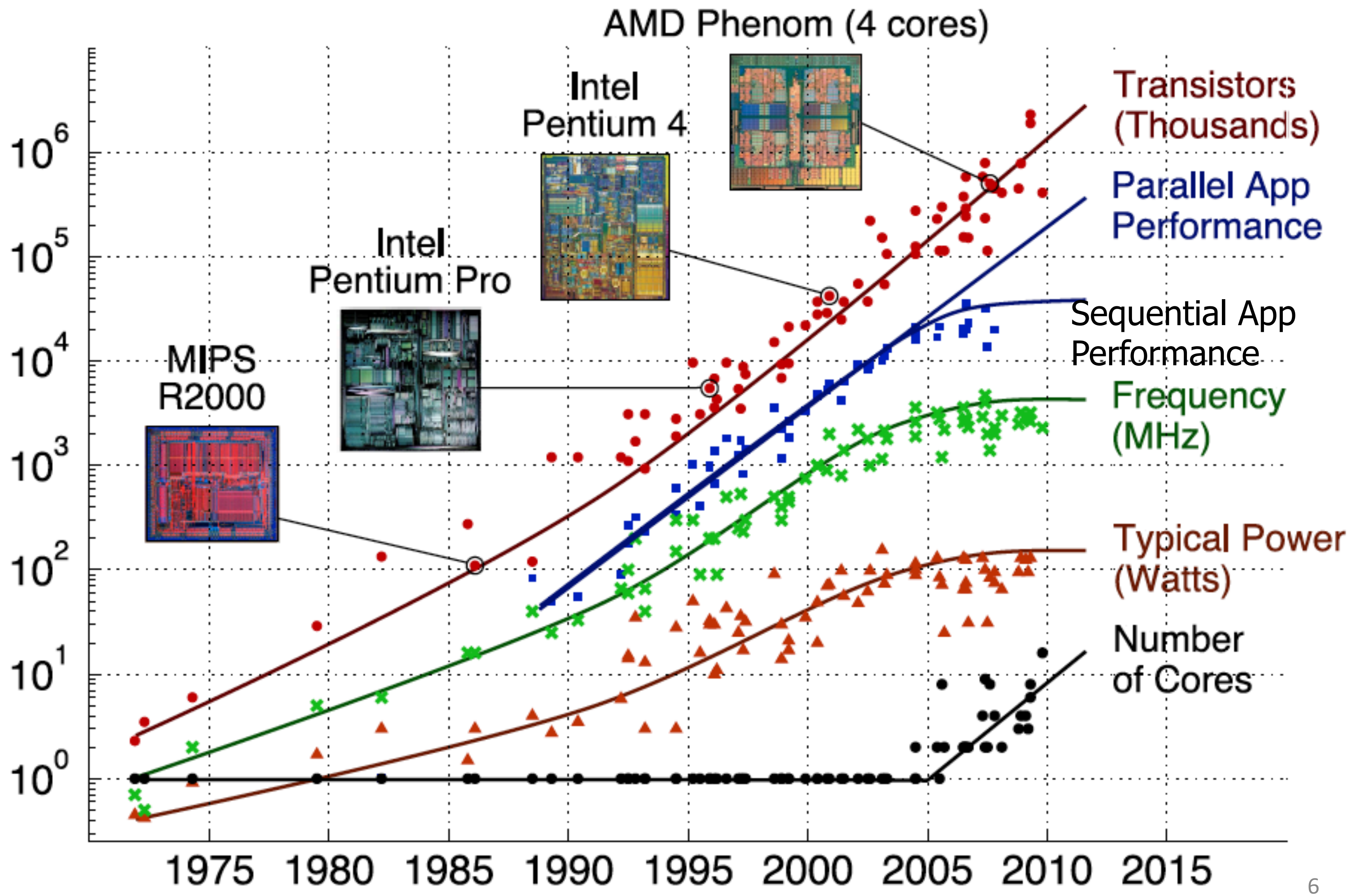


Multiprocessor Execution Model

- Each processor has its own PC and executes an independent stream of instructions (MIMD)
- Different processors can access the same memory space
  - Processors can communicate via shared memory by storing/loading to/from common locations
- Two ways to use a multiprocessor:
  1. Deliver high throughput for independent jobs via job-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**

Use term *core* for processor ("Multicore") because "Multiprocessor Microprocessor" too redundant

Transition to Multicore



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

Parallelism Only Path to Higher Performance

- Sequential processor performance not expected to increase much, and might go down
- If want apps with more capability, have to embrace parallel processing (SIMD and MIMD)
- In mobile systems, use multiple cores and GPUs
- In warehouse-scale computers, use multiple nodes, and all the MIMD/SIMD capability of each node

Multiprocessors and You

- Only path to performance is parallelism
  - Clock rates flat or declining
  - SIMD: 2X width every 3-4 years
    - 128b wide now, 256b 2011, 512b in 2014, 1024b in 2018?
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, …
- Key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication
- Project 3: fastest code on 8-core computers
  - 2 chips/computer, 4 cores/chip

Potential Parallel Performance (assuming SW can use it)

Year	Cores	SIMD bits /Core	Core * SIMD bits	Peak DP FLOPs/Cycle
2003	**MIMD** 2	**SIMD** 128	256	**MIMD** 4
2005	**+2/** 4	**2X/** 128	512	***SIMD** 8
2007	**2yrs** 6	**4yrs** 128	768	12
2009	8	128	1024	16
2011	10	256	2560	40
2013	12	256	3072	48
2015	**2.5X** 14	**8X** 512	7168	**20X** 112
2017	16	512	8192	128
2019	18	1024	18432	288
2021	20	1024	20480	320

Threads

- *Thread:* a sequential flow of instructions that performs some task
- Each thread has a PC + processor registers and accesses the shared memory
- Each processor provides one (or more) *hardware* threads (or *harts*) that actively execute instructions
- Operating system multiplexes multiple *software* threads onto the available hardware threads

Operating System Threads

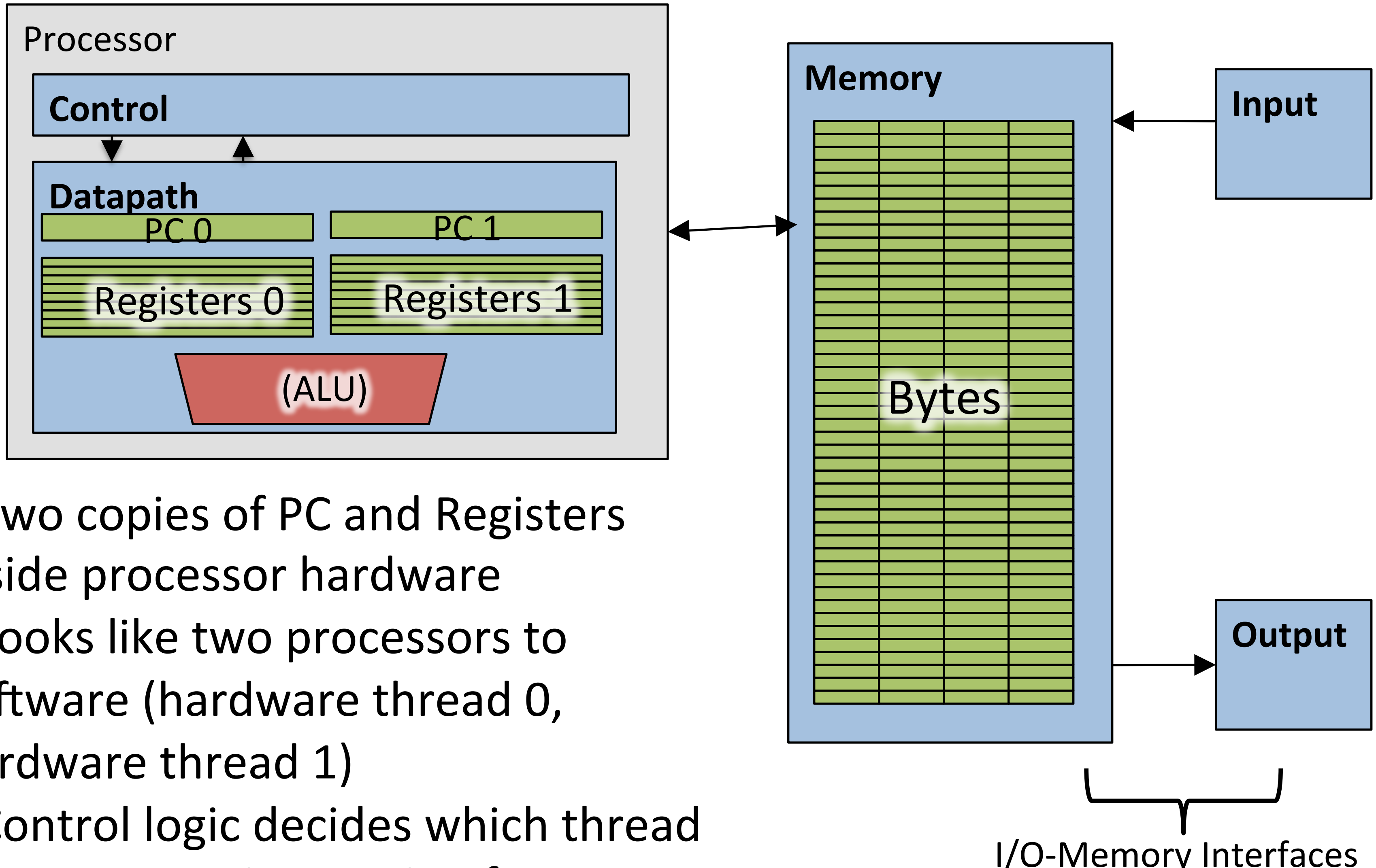
Give the illusion of many active threads by time-multiplexing software threads onto hardware threads

- Remove a software thread from a hardware thread by interrupting its execution and saving its registers and PC into memory
  - Also if one thread is blocked waiting for network access or user input
- Can make a different software thread active by loading its registers into a hardware thread's registers and jumping to its saved PC

Hardware Multithreading

- Basic idea: Processor resources are expensive and should not be left idle
- Long memory latency to memory on cache miss?
- Hardware switches threads to bring in other useful work while waiting for cache miss
- Cost of thread context switch must be much less than cache miss latency
- Put in redundant hardware so don't have to save context on every thread switch:
  - PC, Registers
- Attractive for apps with abundant TLP
  - Commercial multi-user workloads

Hardware 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

Multithreading vs. Multicore

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

Sagar's MacBook Air

```
/usr/sbin/sysctl -a | grep hw\.
```

hw.model = MacBookAir7,2

…

hw.physicalcpu: 2

hw.logicalcpu: 4

…

hw.cpufrequency = 1,600,000,000

hw.memsize = 4,294,967,296

hw.cachelinesize = 64

hw.l1icachesize: 32,768

hw.l1dcachesize: 32,768

hw.l2cachesize: 262,144

hw.l3cachesize: 4,194,304

A Research Machine

```
skarandikar@a8:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
…
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
NUMA node0 CPU(s):     0-7,16-23
NUMA node1 CPU(s):     8-15,24-31
```

Therefore, should try up to 32 threads to see if performance gain even though only 16 real cores

Administrivia

- Project 3-1 Out
  - Last week, we built a CPU together, this week, you start building your own!
- HW4 Out - Caches
- Guerrilla Section on Pipelining, Caches on today, 5-7pm, Woz

Administrivia

- Midterm 2 is Tuesday
  - 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 is Friday, 7/24 from 1-4pm in HP Aud.

Break

100s of (Mostly Dead) Parallel Programming Languages

ActorScript	Concurrent Pascal	JoCaml	Orc
Ada	Concurrent ML	Join	Oz
Afnix	Concurrent Haskell	Java	Pict
Alef	Curry	Joule	Reia
Alice	CUDA	Joyce	SALSA
APL	E	LabVIEW	Scala
Axum	Eiffel	Limbo	SISAL
Chapel	Erlang	Linda	SR
Cilk	Fortan 90	MultiLisp	Stackless Python
Clean	Go	Modula-3	SuperPascal
Clojure	Io	Occam	VHDL
Concurrent C	Janus	occam-π	XC

OpenMP

- OpenMP is a language extension used for multi-threaded, shared-memory parallelism
  - Compiler Directives (inserted into source code)
  - Runtime Library Routines (called from your code)
  - Environment Variables (set in your shell)
- Portable
- Standardized
- Easy to compile: `cc –fopenmp name.c`

Shared Memory Model with Explicit Thread-based Parallelism

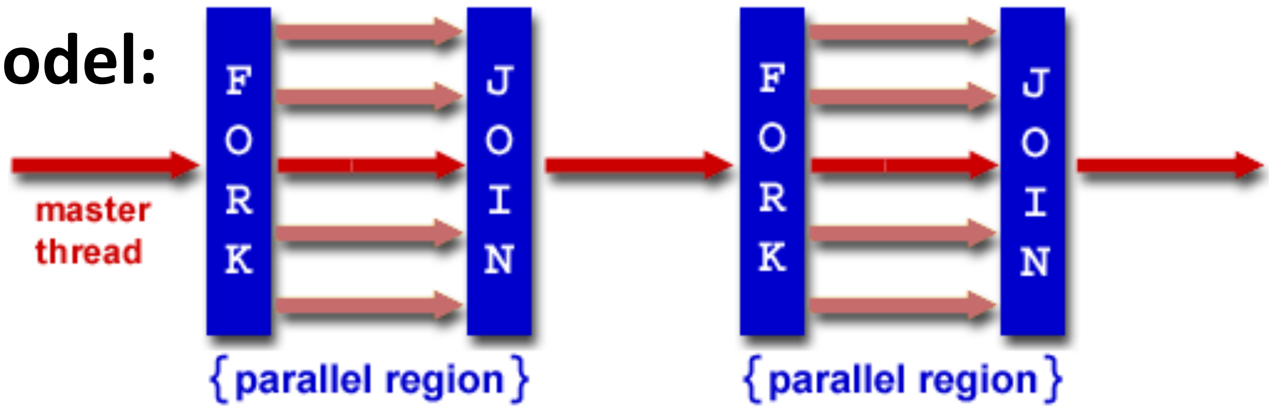
- Multiple threads in a shared memory environment, explicit programming model with full programmer control over parallelization
- **Pros:**
  - Takes advantage of shared memory, programmer need not worry (that much) about data placement
  - Compiler directives are simple and easy to use
  - Legacy serial 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 because compilers that don't recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas

`parallel` Pragma and Scope

- Basic OpenMP construct for parallelization:

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

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

  - *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)
```

Thread Creation

- How many threads will OpenMP create?
- 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

What Kind of Threads?

- OpenMP threads are operating system (software) threads.
- OS will multiplex requested OpenMP threads onto available hardware threads.
- Hopefully each gets a real hardware thread to run on, so no OS-level time-multiplexing.
- But other tasks on machine can also use hardware threads!
- Be careful when timing results for project 3!

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 */
}
```

Data Races and Synchronization

- Two memory accesses form a *data race* if from different threads to same location, and at least one is a write, and they occur one after another
- If there is a data race, result of program can vary 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
- (more later)

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
Set the lock
Critical section
 (e.g. change shared variables)
Unset the lock
```

Can loop/idle here if locked

Possible Lock Implementation

- Lock (a.k.a. busy wait)

```
Get_lock:                    # $s0 -> addr of lock
       addiu $t1,$zero,1     # t1 = Locked value
Loop:  lw $t0,0($s0)         # load lock
       bne $t0,$zero,Loop    # loop if locked
Lock:  sw $t1,0($s0)         # Unlocked, so lock
```

- Unlock

```
Unlock:
    sw $zero,0($s0)
```

- Any problems with this?

Possible Lock Problem

- Thread 1

```
      addiu $t1,$zero,1
Loop: lw $t0,0($s0)

      bne $t0,$zero,Loop

Lock: sw $t1,0($s0)
```

- Thread 2

```
      addiu $t1,$zero,1
Loop: lw $t0,0($s0)

      bne $t0,$zero,Loop

Lock: sw $t1,0($s0)
```

Time

Both threads think they have set the lock!
Exclusive access not guaranteed!

Hardware Synchronization

- Hardware support required to prevent an interloper (another thread) from changing the value
  - *Atomic* read/write memory operation
  - No other access to the location allowed between the read and write
- How best to implement in software?
  - Single instr? Atomic swap of register ↔ memory
  - Pair of instr? One for read, one for write

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

Synchronization in MIPS Example

- Atomic swap (to test/set lock variable)

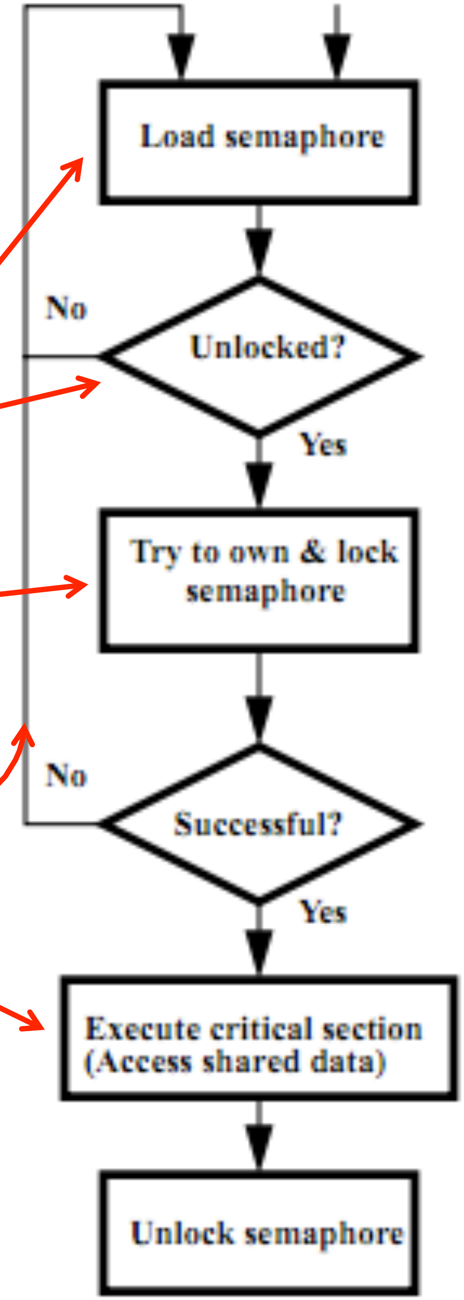
  Exchange contents of register and memory:
  $s4 ↔ Mem($s1)

```
try: add $t0,$zero,$s4 #copy value
     ll  $t1,0($s1)    #load linked
     sc  $t0,0($s1)    #store conditional
     beq $t0,$zero,try #loop if sc fails
     add $s4,$zero,$t1 #load value in $s4
```

`sc` would fail if another threads executes `sc` here

Test-and-Set

- In a single atomic operation:
  - *Test* to see if a memory location is set (contains a 1)
  - *Set* it (to 1) if it isn't (it contained a zero when tested)
  - Otherwise indicate that the Set failed, so the program can try again
  - While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions
- Useful for implementing lock operations



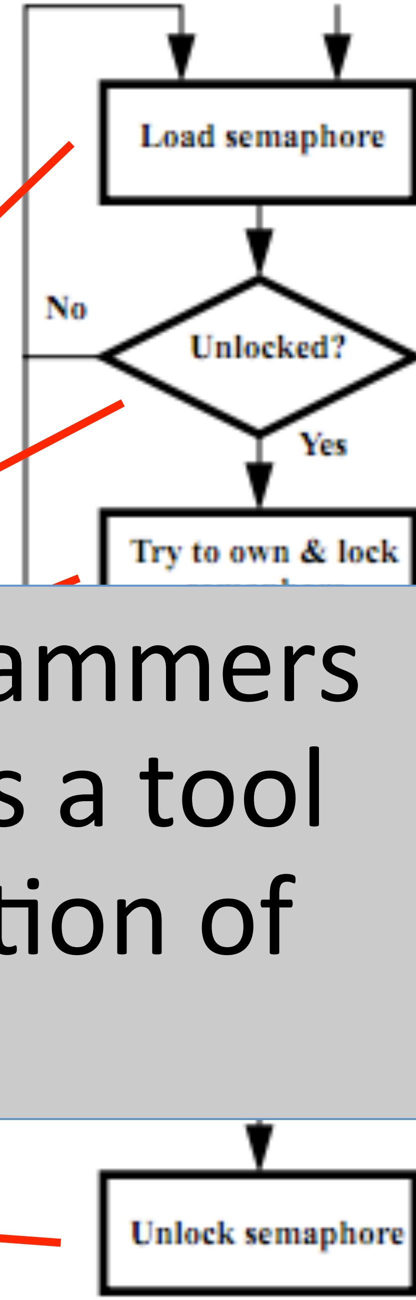
Test-and-Set in MIPS

- Example: MIPS sequence for implementing a T&S at ($s1)

```
Try: addiu $t0,$zero,1
```

Idea is that not for programmers to use this directly, but as a tool for enabling implementation of parallel libraries

```
Unlock:
       sw $zero,0($s1)
```



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

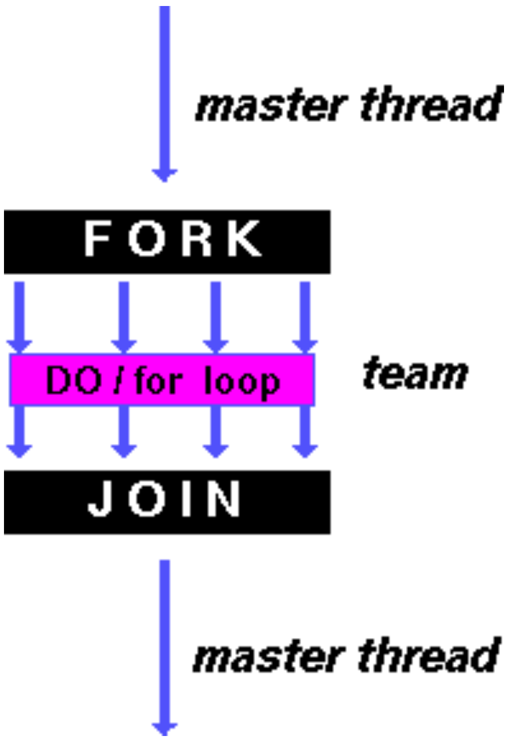
C: 100 or 101

D: 102

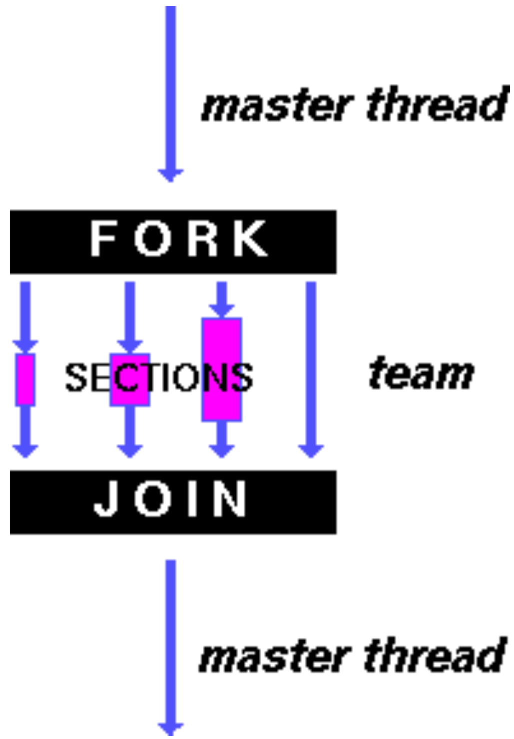
Break

OpenMP Directives (Work-Sharing)

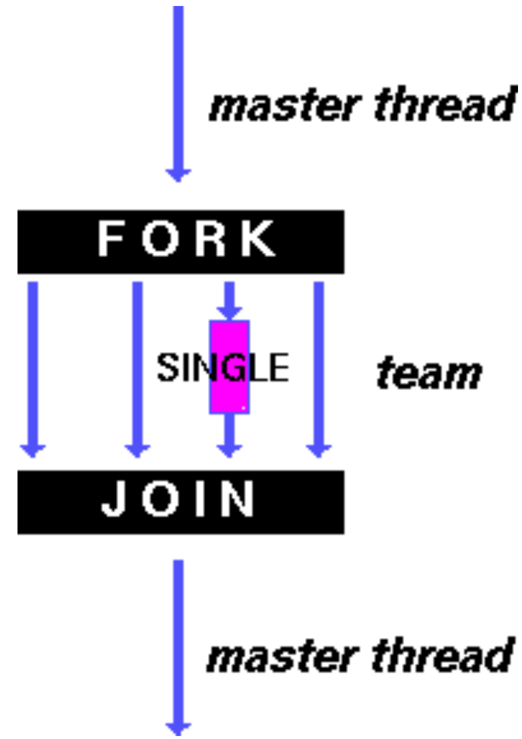
- These are defined *within* a `parallel` section



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

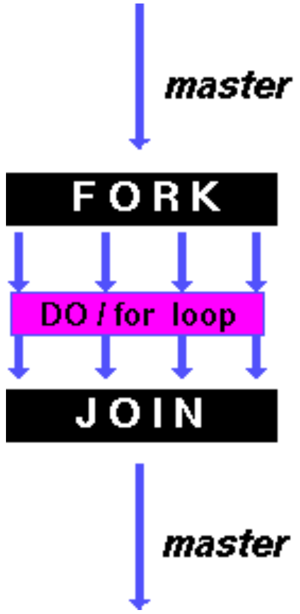
```
for (i=0; i<max; i++) zero[i] = 0;
```

- 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 ← 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 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

```
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

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*)

And in Conclusion, …

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multithreading increases utilization, Multicore more processors (MIMD)
- 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