

CS 61C:
Great Ideas in Computer Architecture
TLP and Cache Coherency

Instructor:

Bernhard Boser and Randy H. Katz

<http://inst.eecs.Berkeley.edu/~cs61c/fa16>

New-School Machine Structures

Software

Hardware

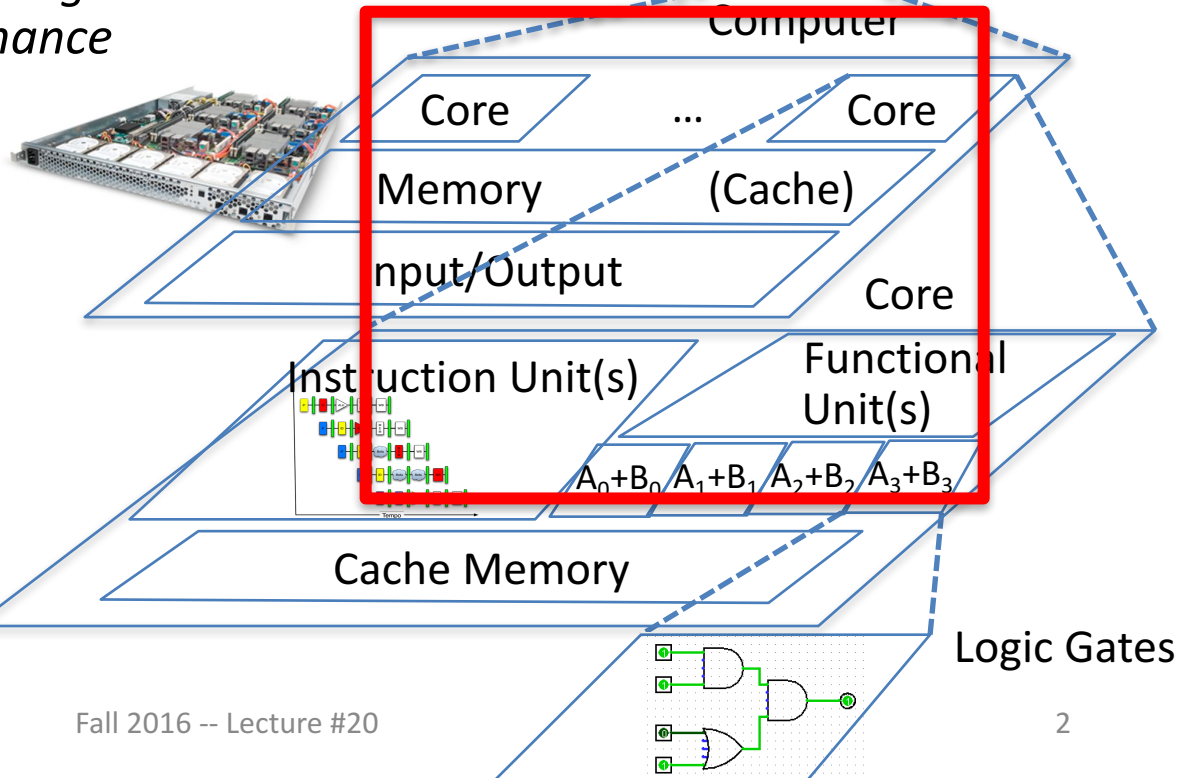
- Parallel Requests
Assigned to computer
e.g., Search “Katz”
- Parallel Threads
Assigned to core
e.g., Lookup, Ads
- Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions
- Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words
- Hardware descriptions
All gates @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

*Harness
Parallelism &
Achieve High
Performance*

Warehouse
Scale
Computer



Smart
Phone



Agenda

- Review: Thread Level Parallelism
- Open MP Reduction
- Multiprocessor Cache Coherency
- False Sharing
- And in Conclusion, ...

Agenda

- Review: Thread Level Parallelism
- Open MP Reduction
- Multiprocessor Cache Coherency
- False Sharing
- And in Conclusion, ...

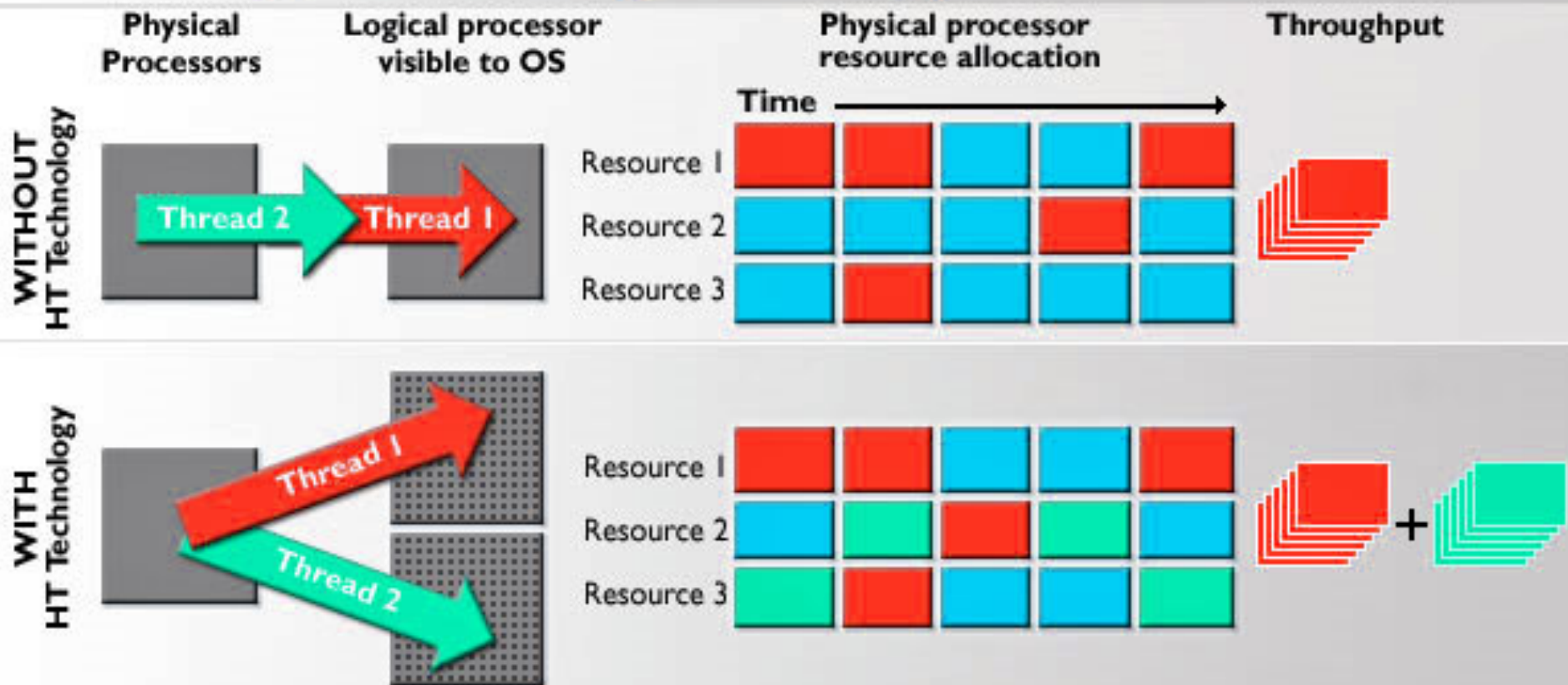
ILP vs. TLP

- Instruction Level Parallelism
 - Multiple instructions in execution at the same time, e.g., *instruction pipelining*
 - *Superscalar*: launch more than one instruction at a time, typically from one instruction stream
 - ILP limited because of pipeline hazards

ILP vs. TLP

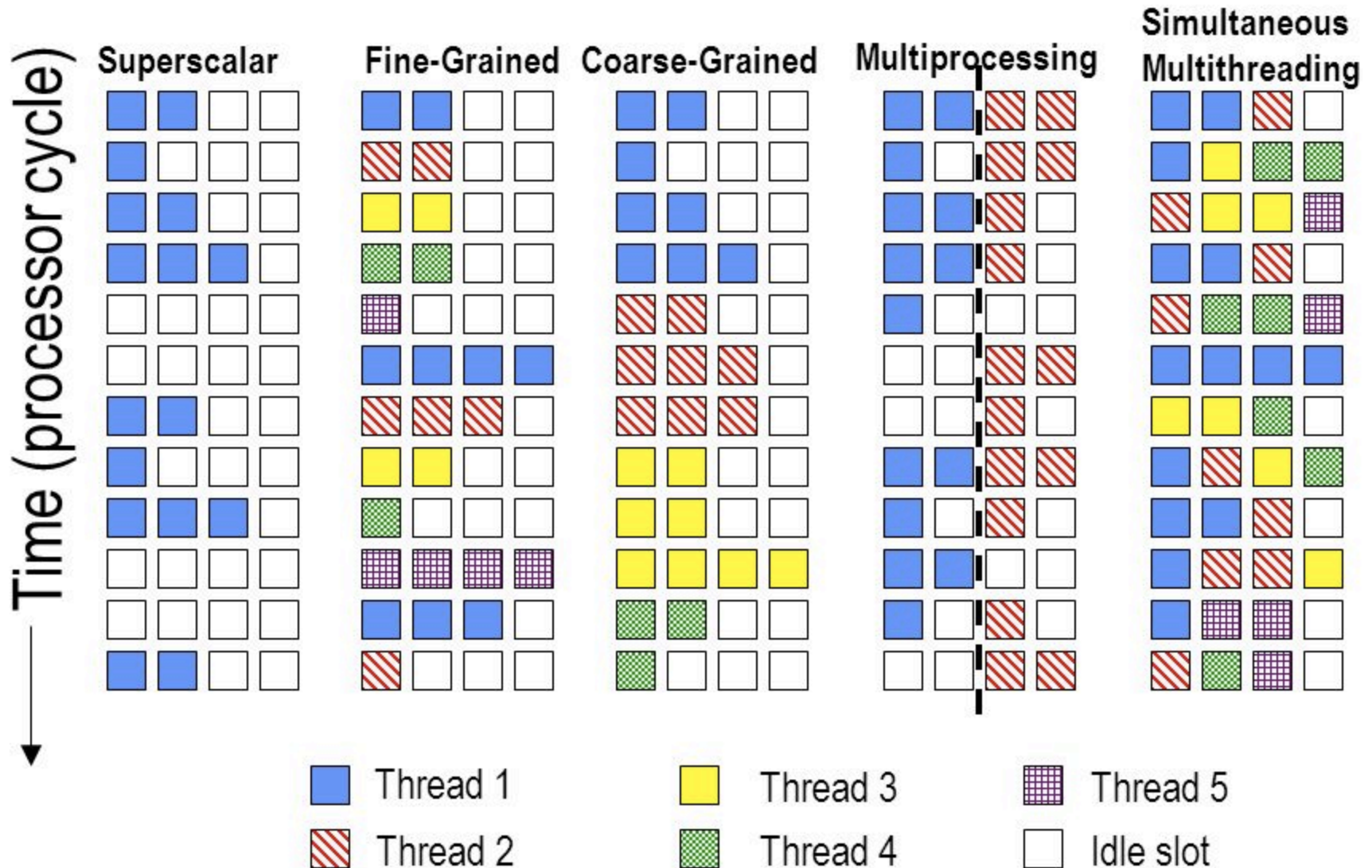
- Thread Level Parallelism
 - *Thread*: sequence of instructions, with own program counter and processor state (e.g., register file)
 - *Multicore*:
 - **Physical CPU**: One thread (at a time) per CPU, in software OS switches threads typically in response to I/O events like disk read/write
 - **Logical CPU**: Fine-grain thread switching, in hardware, when thread blocks due to cache miss/memory access
 - **Hyperthreading** (aka **Simultaneous Multithreading**--SMT): Exploit superscalar architecture to launch instructions from different threads *at the same time*!

How Hyper-Threading Technology Works



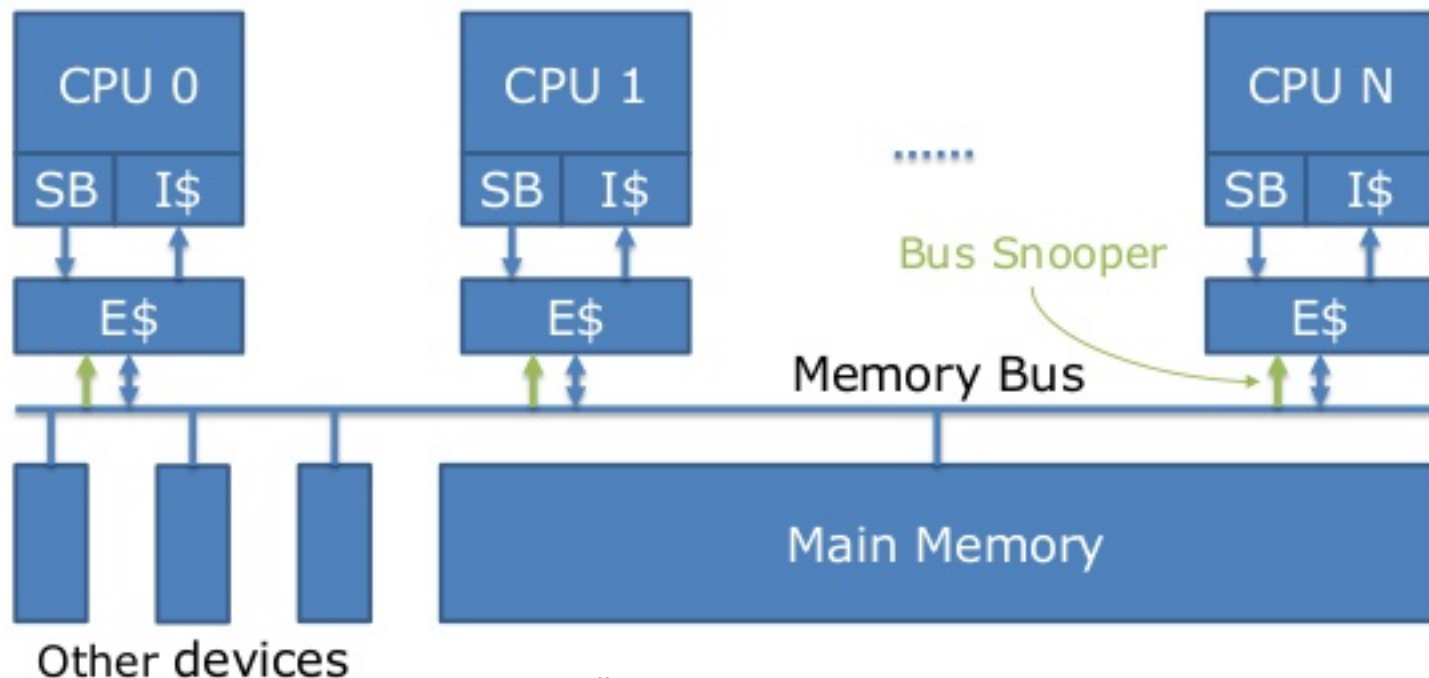
- SMT (HT): Logical CPUs > Physical CPUs
 - Run multiple threads at the same time per core
 - Each thread has own architectural state (PC, Registers, Conditions)
 - Share resources (cache, instruction unit, execution units)
 - Improves Core CPI (clock ticks per instruction)
 - May degrade Thread CPI (Utilization/Bandwidth v. Latency)
 - See <http://dada.cs.washington.edu/smt/>

Summary: Multithreaded Categories



(Chip) Multicore Multiprocessor

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



Randy's (Old) Mac Air

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

hw.model = Core i7, 4650U hw.cachelinesize = 64

... hw.l1icachesize: 32,768

hw.physicalcpu: 2 hw.l1dcachesize: 32,768

hw.logicalcpu: 4 hw.l2cachesize: 262,144

... hw.l3cachesize: 4,194,304

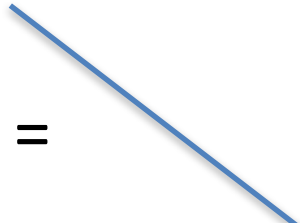
hw.cpufrequency =
1,700,000,000

hw.physmem =
8,589,934,592 (8 Gbytes) ...

Every machine is multicore,
Even your phone!

Hive Machines

hw.model = Core i7 4770K hw.cachelinesize = 64
hw.physicalcpu: 4 hw.l1icachesize: 32,768
hw.logicalcpu: 8 hw.l1dcachesize: 32,768
... hw.l2cachesize: 262,144
hw.cpufrequency = hw.l3cachesize: 8,388,608
3,900,000,000
hw.physmem =
34,359,738,368



Therefore, should try up to 8 threads to see if performance gain even though only 4 cores

Why Parallelism?


- Only path to performance is parallelism
 - Clock rates flat or declining
 - SIMD: 2X width every 3-4 years
 - AVX-512 2015, 1024b in 2018? 2019?
 - MIMD: Add 2 cores every 2 years (2, 4, 6, 8, 10, ...)
Intel Broadwell-Extreme (2Q16): 10 Physical CPUs, 20 Logical CPUs
- Key challenge: craft parallel programs with high performance on multiprocessors as # of processors increase – i.e., that scale
 - Scheduling, load balancing, time for synchronization, overhead for communication
- Project #4: fastest code on 8 processor computer
 - 2 logical CPUs/core, 8 cores/computer

Agenda

- Review: Thread Level Parallelism
- **Open MP Reduction**
- Multiprocessor Cache Coherency
- False Sharing
- And in Conclusion, ...

Review: OpenMP Building Block: `for` loop

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

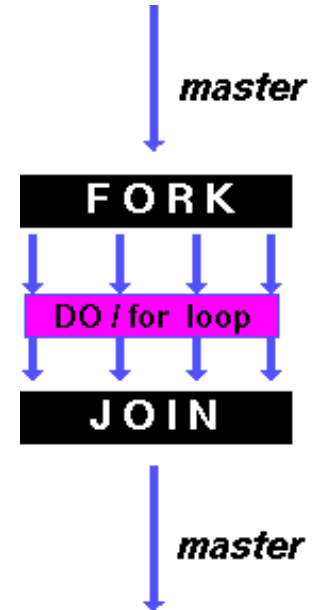
- Breaks *for loop* into chunks, and allocate each to a separate thread
 - e.g. if `max = 100` with 2 threads:
assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
 - Necessary for the run-time system to be able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

Review: OpenMP 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 implicitly *private* per thread
- 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?



OpenMP Timing

- Elapsed wall clock time:

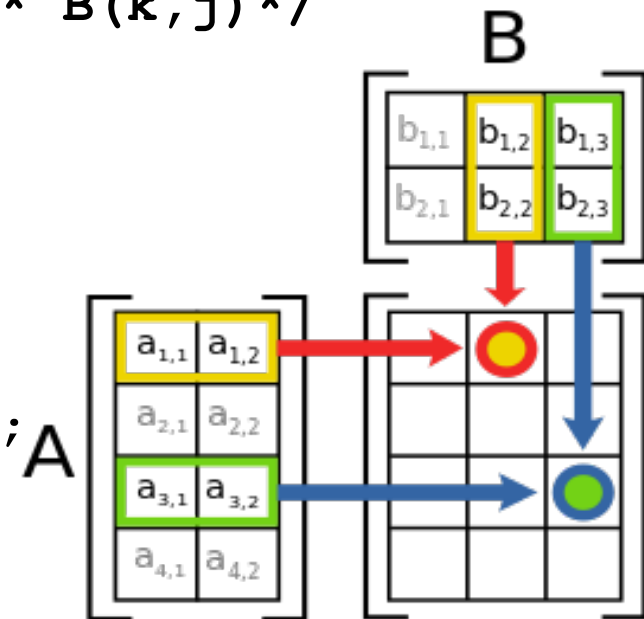
```
double omp_get_wtime(void);
```

- 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;
    }
  }
```

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



Matrix Multiply in Open MP

- More performance optimizations available:
 - Higher *compiler optimization* (-O2, -O3) to reduce number of instructions executed
 - *Cache blocking* to improve memory performance
 - Using SIMD AVX instructions to raise floating point computation rate (*DLP*)

New: 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 all 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
 - *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;
```

Review: Calculating π Original Version

```
#include <omp.h>
#define NUM_THREADS 4
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 ( i, 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=1; i<NUM_THREADS; i++)
        sum[0] += sum[i];  pi = sum[0] / num_steps
    printf ("pi = %6.12f\n", pi);
}
```

Version 2: parallel for, reduction

```
#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);
}
```

Administrivia

- Midterm II graded and regrade period open until Monday@11:59:59 PM
 - Mean similar to Midterm I
 - Approximately 60% of points
- What do about Friday labs and Veterans Day? W before Thanksgiving? R&R Week?
 - Make ups for Friday labs posted on Piazza
- Project #4-1 released



Agenda

- Review: Thread Level Parallelism
- Open MP Reduction
- **Multiprocessor Cache Coherency**
- False Sharing
- And in Conclusion, ...

Multiprocessor Key Questions

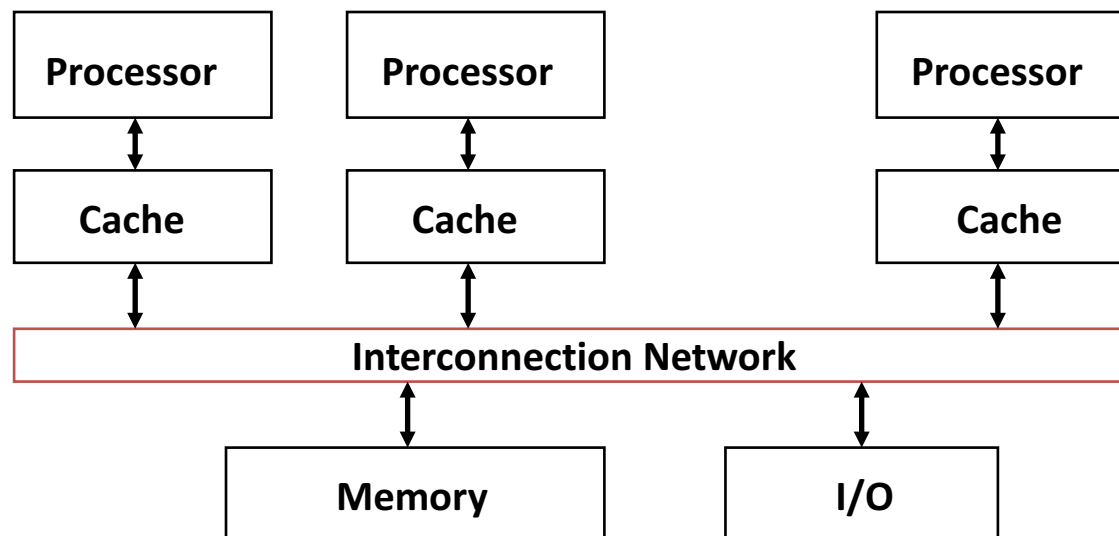
- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How many processors can be supported?

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

Parallel Processing: Multiprocessor Systems (MIMD)

- **Multiprocessor (MIMD)**: a computer system with at least 2 processors

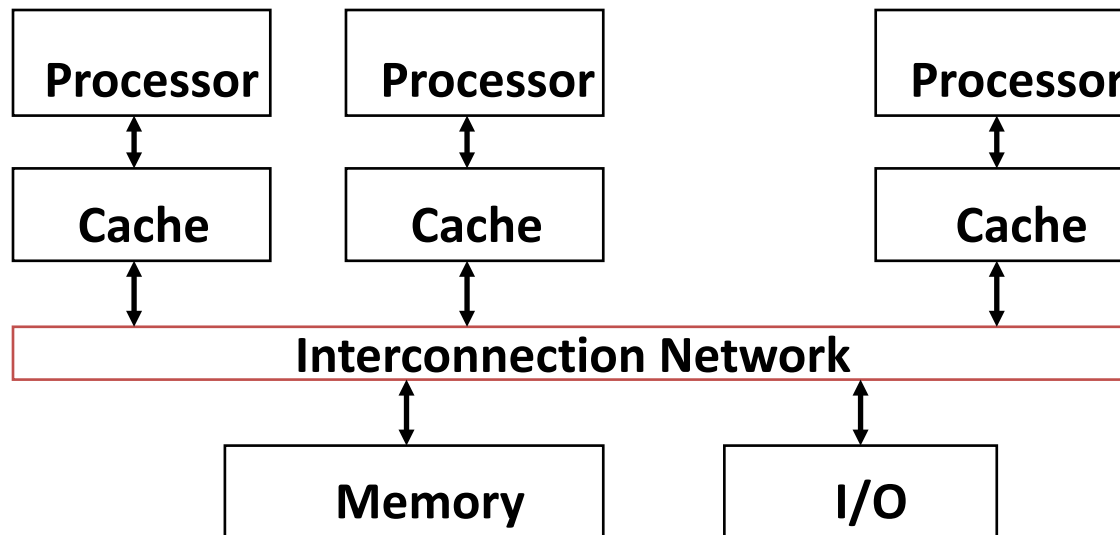


1. Deliver high throughput for independent jobs via job-level parallelism
2. **Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program**

Now Use term **core** for processor (“Multicore”) because
“Multiprocessor Microprocessor” too redundant

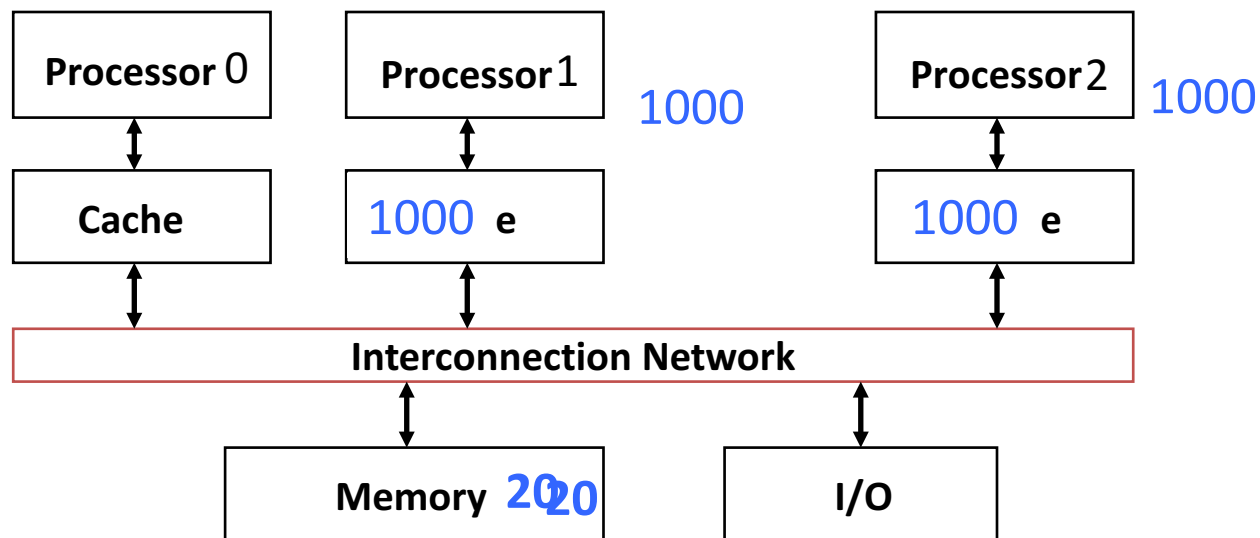
Multiprocessor Caches

- 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



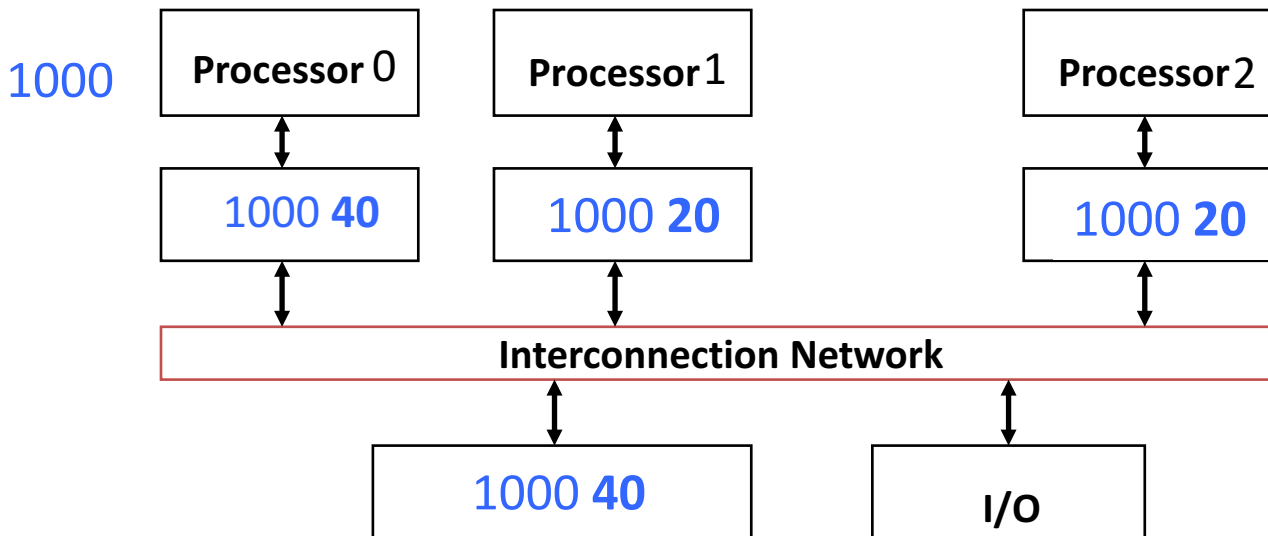
Shared Memory and Caches

- 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, other caches “snoop” the common interconnect checking for tags they hold
 - Invalidate any copies of same address modified in other cache

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

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)
 - Only cache that supplies data on read instead of going to memory

Name of 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)



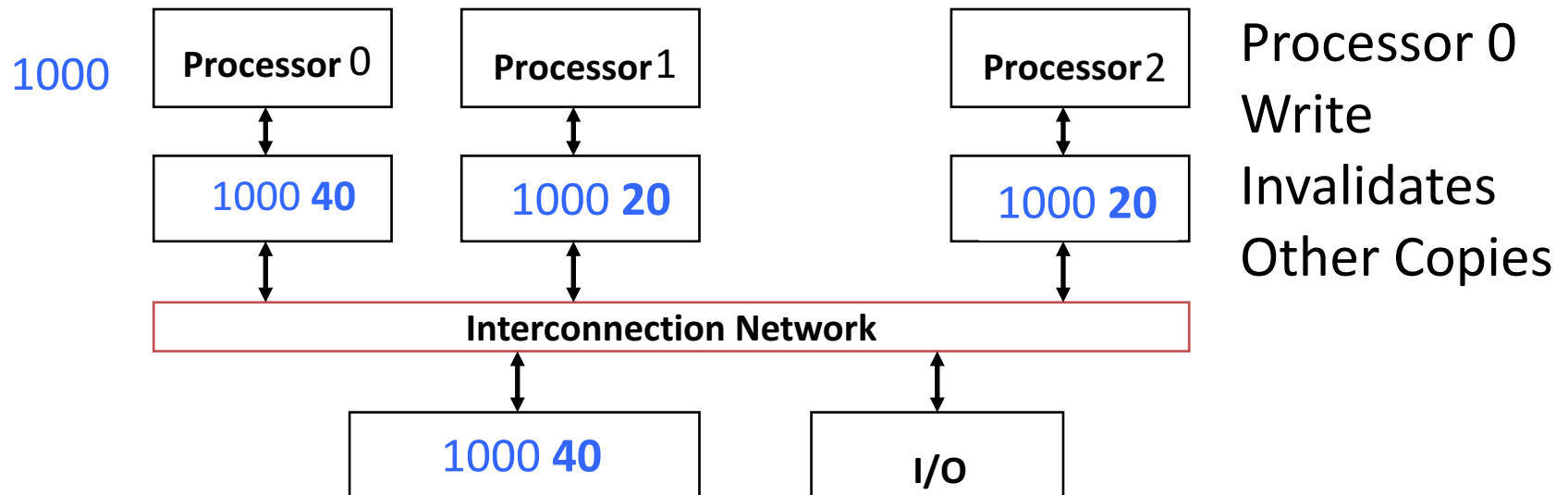
Snooping/Snoopy Protocols
e.g., the Berkeley Ownership Protocol

See http://en.wikipedia.org/wiki/Cache_coherence

Berkeley Protocol is a wikipedia stub!

Shared Memory and Caches

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



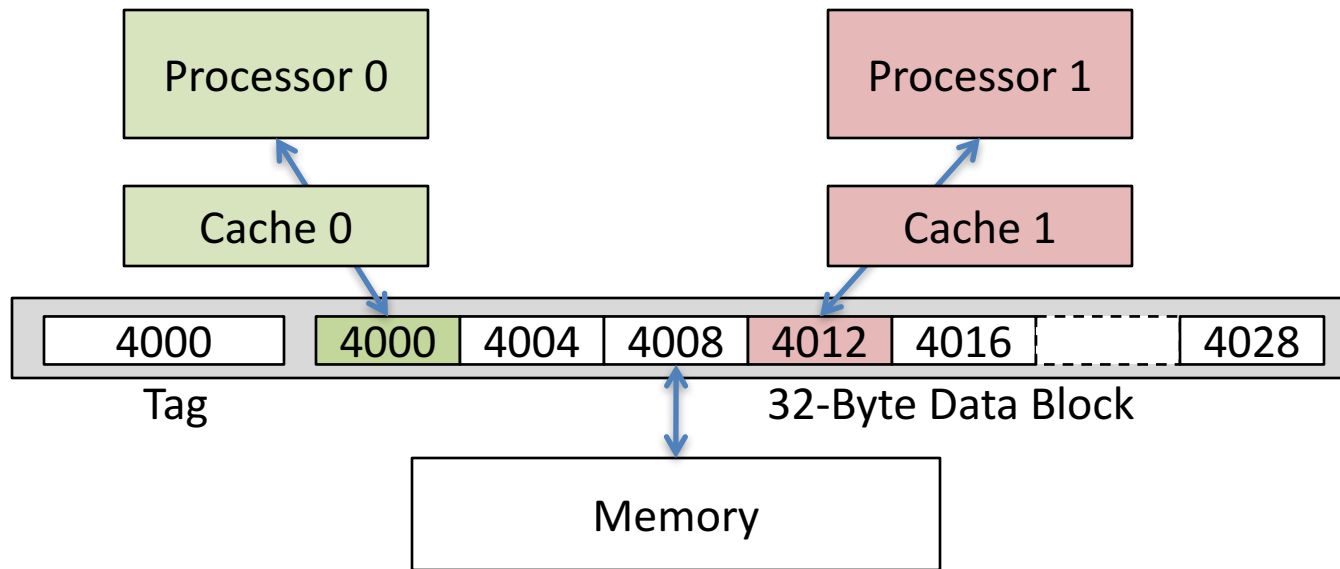


**KEEP
CALM
IT'S
BREAK
TIME**

Agenda

- Review: Thread Level Parallelism
- Open MP Reduction
- Multiprocessor Cache Coherency
- **False Sharing**
- And in Conclusion, ...

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?

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

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

False Sharing in OpenMP

```
int i;      double  x, pi, sum[NUM_THREADS];
#pragma omp parallel private (i, x)
{
    int id = omp_get_thread_num();
    for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREAD)
    {
        x = (i+0.5)*step;
        sum[id] += 4.0/(1.0+x*x);
    }
}
```

- What is problem?
- Sum[0] is 8 bytes in memory, Sum[1] is adjacent 8 bytes in memory => false sharing if block size > 8 bytes

Peer Instruction: Avoid False Sharing

```
{  int i;  double x, pi, sum[10000];  
#pragma omp parallel private (i, x)  
{  int id = omp_get_thread_num(), fix = _____;  
    for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS)  
    {  
        x = (i+0.5)*step;  
        sum[id*fix] += 4.0/(1.0+x*x);  
    }
```

- What is best value to set `fix` to prevent false sharing?
 - (A) `omp_get_num_threads()` ;
 - (B) Constant for number of blocks in cache
 - (C) Constant for size of blocks in bytes
 - (D) Constant for size of blocks in doubles

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

Review: Synchronization in MIPS

- *Load linked:* **ll \$rt, off(\$rs)**
 - Reads memory location (like **lw**)
 - Also sets (hidden) “link bit”
 - Link bit is reset if memory location (**off(\$rs)**) is accessed
- *Store conditional:* **sc \$rt, off(\$rs)**
 - Stores **off(\$rs) = \$rt** (like **sw**)
 - Sets **\$rt=1** (success) if link bit is set
 - i.e. no (other) process accessed **off(\$rs)** since **ll**
 - Sets **\$rt=0** (failure) otherwise
 - Note: **sc clobbers \$rt**, i.e. changes its value

Review: Lock Synchronization

$\$t0 = 1$ before calling **ll**: minimize time between **ll** and **sc**

Broken Synchronization

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

```
lock = 1;
```

```
// critical section
```

```
lock = 0;
```

Fix (lock is at location $\$s1$)

```
Try: addiu $t0,$zero,1
      ll    $t1,0($s1)
      bne   $t1,$zero,Try
      sc    $t0,0($s1)
      beq   $t0,$zero,Try
```


Locked:

```
# critical
section
```

Unlock:

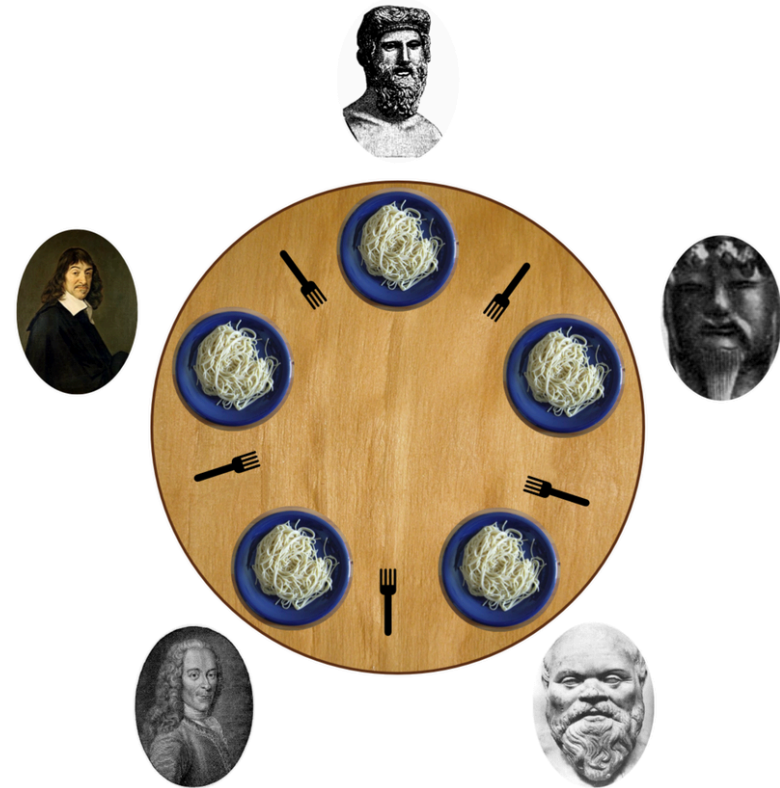
```
sw $zero,0($s1)
```

Try again if **sc** failed
(another thread executed
sc since above **ll**)



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?



Agenda

- Review: Thread Level Parallelism
- Open MP Reduction
- Multiprocessor Cache Coherency
- False Sharing
- **And in Conclusion, ...**

And in Conclusion, ...

- OpenMP as simple parallel extension to C
 - Threads level programming with `parallel` for `pragma`, `private` variables, `reductions`, ...
 - \approx C: small so easy to learn, but not very high level and it's easy to get into trouble
- ILP vs. TLP
 - CMP (Chip Multiprocessor aka Symmetric Multiprocessor) vs. SMT (Simultaneous Multithreading)
 - Cache coherency implements shared memory even with multiple copies in multiple caches
 - False sharing a concern; watch block size!