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

Instructors:
Krste Asanović and Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa17
New-School Machine Structures

- **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**
Agenda

• Thread Level Parallelism Revisited
• Open MP Part II
• Multiprocessor Cache Coherency
• False Sharing (if time)
• And, in Conclusion, ...
Agenda

- Thread Level Parallelism Revisited
- Open MP Part II
- Multiprocessor Cache Coherency
- False Sharing (if time)
- 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!
SMT (HT): Logical CPUs > Physical CPUs
- Run multiple threads at the same time per core
- Each thread has own architectural state (PC, Registers, etc.)
- 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

- Superscalar
- Fine-Grained
- Coarse-Grained
- Multiprocessing
- Simultaneous Multithreading

Time (processor cycle)

- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Idle slot
Review: Randy’s (Rather Old) Mac Air

- `/usr/sbin/sysctl -a | grep hw\.`
  
  hw.model = Core i7, 4650U (hw.cachelinesize = 64)
  ...
  hw.physicalcpu: 2 (hw.l1icachesize: 32,768)
  hw.logicalcpu: 4 (hw.l1dcachesize: 32,768)
  ...
  hw.cpufreqerfrequency = 1,700,000,000  
  hw.physmem = 8,589,934,592 (8 Gbytes)...

*Every machine is multicore, Even your phone!*
Review: Hive Machines

hw.model = Core i7 4770K
hw.physicalcpu: 4
hw.logicalcpu: 8
...
hw.cpufreq = 3,900,000,000
hw.physmem = 34,359,738,368

hw.cachelinesize = 64
hw.l1icachesize: 32,768
hw.l1dcachesize: 32,768
hw.l2cachesize: 262,144
hw.l3cachesize: 8,388,608

Therefore, should try up to 8 threads to see if performance gain even though only 4 cores
Review: 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 #3: fastest code on 8 processor computer
  – 2 logical CPUs/core, 8 cores/computer
Agenda

• Thread Level Parallelism Revisited
• Open MP Part II
• Multiprocessor Cache Coherency
• False Sharing (if time)
• And, in Conclusion, ...
Review: OpenMP Building Block: \texttt{for} loop

\begin{verbatim}
for (i=0; i<max; i++) zero[i] = 0;
\end{verbatim}

• Breaks \textit{for loop} into chunks, and allocate each to a separate thread
  – e.g. if \texttt{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 \texttt{break, return, exit, goto} statements

In general, don’t jump outside of any \texttt{pragma} block
Review: OpenMP Parallel for pragma

```c
#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?
Example 2: Computing $\pi$

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 $\Delta x$ and height $F(x_i)$ at the middle of interval $i$.

http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 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);
            printf("i =%3d, id =%3d\n", i, id);
        }
    }

    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf("pi = %6.12f\n", pi);
}
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 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);
            printf("i =%3d, id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf("pi = %6.12f\n", pi);
}

\textbf{Trial Run}

\begin{itemize}
  \item \(i = 1, \ id = 1\)
  \item \(i = 0, \ id = 0\)
  \item \(i = 2, \ id = 2\)
  \item \(i = 3, \ id = 3\)
  \item \(i = 5, \ id = 1\)
  \item \(i = 4, \ id = 0\)
  \item \(i = 6, \ id = 2\)
  \item \(i = 7, \ id = 3\)
  \item \(i = 9, \ id = 1\)
  \item \(i = 8, \ id = 0\)
\end{itemize}

\textbf{pi} = 3.142425985001
Scale up: \texttt{num\_steps} = 10^6

\begin{verbatim}
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 1000000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 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);
        // printf("i =%3d, id =%3d\n", i, id);
      }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}  
\end{verbatim}

\pi \approx 3.141592653590

You verify how many digits are correct ...
Can We Parallelize Computing $\text{sum}$?

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

Always looking for ways to beat Amdahl’s Law ...

Summation inside parallel section
- Insignificant speedup in this example, but ...
- $\pi = 3.138450662641$
- Wrong! And value changes between runs?!
- What’s going on?
What’s Going On?

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

- Operation is really $\pi = \pi + \text{sum}[id]$
- What if >1 threads reads current (same) value of $\pi$, computes the sum, stores the result back to $\pi$?
- Each processor reads same intermediate value of $\pi$!
- Result depends on who gets there when
  - A “race” → result is not deterministic
OpenMP Reduction

```c
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.

```c
#pragma omp for reduction(+ : sum)
for (i = 0; i <= MAX ; i++)
    sum += A[i];
avg = sum/MAX;
```
Calculating π Original Version

```c
#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];
    printf ("pi = %6.12f\n", pi);
}
```

11/8/17 Fall 2017 -- Lecture #20
Version 2: parallel for, reduction

```c
#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;
    printf("pi = %6.8f\n", pi);
}
```
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.
Locks

- Computers use locks to control access to shared resources
  - Serves purpose of microphone in example
  - Also referred to as “semaphore”

- Usually implemented with a variable
  - `int lock;`
    - 0 for unlocked
    - 1 for locked
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;

Thread 2

while (lock != 0) ;

lock = 1;

// critical section

lock = 0;

• Thread 2 finds 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 ↔ 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

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]
```
RISC-V 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
```
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)

11/8/17
Fall 2017 - Lecture #20
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?
OpenMP Timing

• Elapsed wall clock time:
  
  ```c
  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, 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;
        }
    }
run_time = omp_get_wtime() - start_time;
```

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*)
Administrivia

• Midterm II graded
  – Regrades will open tomorrow
    • 24hrs to review solutions before requesting regrade
  – Due Friday@11:59:59 PM

• No lab this Friday! (Veterans Day)
  – This week only: attend Thursday or Monday lab for checkoffs

• Project #3 released Wednesday
KEEP CALM
IT'S BREAK TIME
Peer Instruction: Why Multicore?

The switch in ~ 2005 from one processor per chip to multiple processors per chip happened because:

I. The “power wall” meant that no longer get speed via higher clock rates and higher power per chip

II. There was no other performance option but replacing one inefficient processor with multiple efficient processors

III. OpenMP was a breakthrough in ~2000 that made parallel programming easy

<table>
<thead>
<tr>
<th>Color</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>I only</td>
</tr>
<tr>
<td>GREEN</td>
<td>II only</td>
</tr>
<tr>
<td>ORANGE</td>
<td>I &amp; II only</td>
</tr>
<tr>
<td>YELLOW</td>
<td>I, II, &amp; III only</td>
</tr>
</tbody>
</table>
Peer Instruction: Why Multicore?

The switch in ~2005 from one processor per chip to multiple processors per chip happened because:

I. The “power wall” meant that no longer get speed via higher clock rates and higher power per chip

II. There was no other performance option but replacing one inefficient processor with multiple efficient processors

III. OpenMP was a breakthrough in ~2000 that made parallel programming easy

<table>
<thead>
<tr>
<th>Color</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>I only</td>
</tr>
<tr>
<td>GREEN</td>
<td>II only</td>
</tr>
<tr>
<td>ORANGE</td>
<td>I &amp; II only</td>
</tr>
<tr>
<td>YELLOW</td>
<td>I, II, &amp; III only</td>
</tr>
</tbody>
</table>
Agenda

• Thread Level Parallelism Revisited
• Open MP Part II
• Multiprocessor Cache Coherency
• False Sharing (if time)
• And, in Conclusion, ...
(Chip) Multicore Multiprocessor

• SMP: (Shared Memory) Symmetric Multiprocessor
  – Two or more identical CPUs/Cores
  – Single shared *coherent* memory
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
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)
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
Peer Instruction: Which Statement is True?

**RED:** Using write-through caches removes the need for cache coherence

**GREEN:** Every processor store instruction must check contents of other caches

**ORANGE:** Most processor load and store accesses only need to check in local private cache

**YELLOW:** Only one processor can cache any memory location at one time
Peer Instruction: Which Statement is True?

**RED:** Using write-through caches removes the need for cache coherence

**GREEN:** Every processor store instruction must check contents of other caches

**ORANGE:** Most processor load and store accesses only need to check in local private cache

**YELLOW:** Only one processor can cache any memory location at one time
Review MOESI Cache Coherency

1. **Shared**: up-to-date data, other caches may have copy
2. **Modified**: up-to-date data, changed (dirty), no other cache has copy, OK to write, memory out-of-date
3. **Exclusive**: up-to-date data, no other cache has copy, OK to write, memory up-to-date
4. **Owner**: up-to-date data, other caches may have a copy (they must be in Shared state)

I. If in Exclusive state, processor can write without notifying other caches
II. Owner state is variation of Shared state to let caches supply data instead of going to memory on read miss
III. Exclusive state is variation of Modified state to let caches avoid writing to memory on a miss

- RED  I only
- GREEN II only
- ORANGE I and II
- YELLOW I, II and III
Review MOESI Cache Coherency

1. **Shared**: up-to-date data, other caches may have copy
2. **Modified**: up-to-date data, changed (dirty), no other cache has copy, OK to write, memory out-of-date
3. **Exclusive**: up-to-date data, no other cache has copy, OK to write, memory up-to-date
4. **Owner**: up-to-date data, other caches may have a copy (they must be in Shared state)

I. If in Exclusive state, processor can write without notifying other caches
II. Owner state is variation of Shared state to let caches supply data instead of going to memory on read miss
III. Exclusive state is variation of Modified state to let caches avoid writing to memory on a miss

- **RED** I only
- **ORANGE** I and II
- **GREEN** II only
- **YELLOW** I, II and III
Agenda

• Thread Level Parallelism Revisited
• Open MP Part II
• Multiprocessor Cache Coherency
• False Sharing (if time)
• 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

```c
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

```c
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?
  - RED `omp_get_num_threads()`;
  - GREEN Constant for number of blocks in cache
  - ORANGE Constant for size of blocks in bytes
  - YELLOW Constant for size of blocks in doubles
Peer Instruction: Avoid False Sharing

```c
{ 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?
  
  **RED**  omp_get_num_threads();
  **GREEN** Constant for number of blocks in cache
  **ORANGE** Constant for size of blocks in bytes
  **YELLOW** Constant for size of blocks in doubles
Agenda

• Thread Level Parallelism Revisited
• Open MP Part II
• Multiprocessor Cache Coherency
• False Sharing (if time)
• And, in Conclusion, ...
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

• 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!