CS 61C: Great Ideas in Computer Architecture (Machine Structures)  
*Thread-Level Parallelism (TLP)* and *OpenMP*

**Instructors:**  
Krste Asanovic & Vladimir Stojanovic  
http://inst.eecs.berkeley.edu/~cs61c/
Review

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

- These are defined within a parallel section

**FORK**
- 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

```c
#pragma omp parallel
{
    #pragma omp for
    for(i=0; i<len; i++) { ... }
}
```

can be shortened to:

```c
#pragma omp parallel for
    for(i=0; i<len; i++) { ... }
```

• Also works for sections

This is the only directive in the parallel section
Building Block: *for* loop

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

- Break *for loop* into chunks, and allocate each chunk to a separate thread
  - e.g. if `max = 100` with 2 threads:
    assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed
  - i.e. No `break`, `return`, `exit`, `goto` statements

In general, don’t jump outside of any pragma block
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:

```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

```c
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*)
Simple Multiprocessor
Multiprocessor Caches

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory
Shared Memory and Caches

• What if?
  – Processors 1 and 2 read Memory[1000] (value 20)
Shared Memory and Caches

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

Problem?
Keeping Multiple Caches Coherent

• Architect’s job: shared memory => keep cache values coherent
• Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  – If only reading, many processors can have copies
  – If a processor writes, invalidate any other copies
• Write transactions from one processor “snoop” tags of other caches using common interconnect
  – Invalidate any “hits” to same address in other caches
  – If hit is to dirty line, other cache has to write back first!
Shared Memory and Caches

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

Processor 0

Write
Invalidates
Other Copies

Processor 0

Processor 1

Processor 2

Interconnection Network

I/O
Clickers/Peer Instruction: Which statement is true?

• A: Using write-through caches removes the need for cache coherence
• B: Every processor store instruction must check contents of other caches
• C: Most processor load and store accesses only need to check in local private cache
• D: Only one processor can cache any memory location at one time
Administrivia

• MT2 is Thursday, April 9th:
  – Covers lecture material up till 3/31 lecture
  – TWO cheat sheets, 8.5”x11”
• Suppose block size is 32 bytes
• Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
• Suppose in X location 4000, Y in 4012
• What will happen?
Coherency Tracked by Cache Line

- Block ping-pongs between two caches even though processors are accessing disjoint variables
- Effect called *false sharing*
- How can you prevent it?
Review: Understanding Cache Misses: 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
How to Calculate 3C’s using Cache Simulator

1. **Compulsory**: set cache size to infinity and fully associative, and count number of misses

2. **Capacity**: reduce cache size from infinity, usually in powers of 2, *implement optimal replacement*, and count misses for each reduction in size
   - 16 MB, 8 MB, 4 MB, ... 128 KB, 64 KB, 16 KB

3. **Conflict**: Change from fully associative to n-way set associative while counting misses
   - Fully associative, 16-way, 8-way, 4-way, 2-way, 1-way
• Three sources of misses (SPEC2000 integer and floating-point benchmarks)
  – Compulsory misses 0.006%; not visible
  – Capacity misses, function of cache size
  – Conflict portion depends on associativity and cache size
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
3.

\[ 141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067982148086513282306647093844609550582231725359408128481117450284102... \]
Calculating $\pi$

Numerical Integration

Mathematically, we know that:

$$\int_{0}^{1} \frac{4.0}{(1+x^2)} \, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width $\Delta x$ and height $F(x_i)$ at the middle of interval $i$. 

$$F(x) = \frac{4.0}{(1+x^2)}$$
Sequential Calculation of $\pi$ in C

```c
#include <stdio.h> /* Serial Code */
static long num_steps = 100000; double step;
void main ()
{
    int i;       double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=1; i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = sum/num_steps;
    printf("pi = %6.12f\n", pi);
}
```
OpenMP Version (with bug)

#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
  int i; double x, pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
#pragma omp parallel private (x)
{
  int id = omp_get_thread_num();
  for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS)
  {
    x = (i+0.5)*step;
    sum[id] += 4.0/(1.0+x*x);
  }
}
  for (i=0, pi=0.0; i<NUM_THREADS; i++)
  {
    pi += sum[i];
  }
  printf ("pi = %6.12f\n", pi / num_steps);
}
Experiment

• Run with NUM_THREADS = 1 multiple times
• Run with NUM_THREADS = 2 multiple times
• What happens?
#include <omp.h>

static long num_steps = 100000; double step;

#define NUM_THREADS 2

void main ()
{
  int i;   double x, pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;

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

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

static long num_steps = 100000; double step;
#define NUM_THREADS 2

void main ()
{
    int i;     double x, sum, pi=0.0;
    step = 1.0/(double) num_steps;

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

#pragma omp critical
    pi += sum;
}

    printf ("pi = %6.12f\n",pi/num_steps);
}
OpenMP Reduction

- **Reduction**: specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region: reduction(operation:var) where
  - **Operation**: operator to perform on the variables (var) at the end of the parallel region
  - **Var**: One or more variables on which to perform scalar reduction.

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

```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 / num_steps;
    printf ("pi = %6.8f\n", pi);
}
```

Note: Don’t have to declare for loop index variable i private, since that is default
And in Conclusion, ...

• Multiprocessor/Multicore uses Shared Memory
  – Cache coherency implements shared memory even with multiple copies in multiple caches
  – False sharing a concern; watch block size!
• OpenMP as simple parallel extension to C
  – Threads, Parallel for, private, critical sections, reductions ...
  – \( \approx \) C: small so easy to learn, but not very high level and it’s easy to get into trouble