CS 61C: Great Ideas in Computer Architecture
OpenMP, Part II

Instructor: Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

Review
- OpenMP as simple parallel extension to C
  - Directives: #pragma parallel for, private, reduction...
  - Runtime Library #include <omp.h>; omp_exec
  - Environment variables NUM_THREADS
- OpenMP = C: small so easy to learn, but not very high level and it’s easy to get into trouble

Matrix Multiply in OpenMP

```c
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, i, j, k)
for (i=0; i<Ndim; i++)
  for (j=0; j<Mdim; j++)
    for (k=0; k<Pdim; k++)
      tmp = 0.0;
      /* 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;
```

Notes on Matrix Multiply Example

More performance optimizations available
- Higher compiler optimization (-O2) to reduce number of instructions executed
- Cache blocking to improve memory performance
- Using SIMD SSE3 Instructions to improve floating-point computation rate
Description of 32-Core System

- Intel Nehalem Xeon 7550
- HW Multithreading: 2 Threads / core
- 8 cores / chip
- 4 chips / board
  ⇒ 64 Threads / system
- 2.00 GHz
- 256 KB L2 cache / core
- 18 MB (1) shared L3 cache / chip

Review: Strong vs Weak Scaling

- Strong scaling: problem size fixed
- Weak scaling: problem size proportional to increase in number of processors
  - Speedup on multiprocessor while keeping problem size fixed is harder than speedup by increasing the size of the problem
  - But a natural use of a lot more performance is to solve a lot bigger problem

32 Core: Speed-up vs. Scale-up

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time (secs)</th>
<th>Speedup</th>
<th>Time (Dim)</th>
<th>Size (Dim)</th>
<th>Fl. Ops x 10^9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13.75</td>
<td>1.00</td>
<td>13.75</td>
<td>1000</td>
<td>2.00</td>
</tr>
<tr>
<td>2</td>
<td>6.88</td>
<td>2.00</td>
<td>13.52</td>
<td>1240</td>
<td>3.81</td>
</tr>
<tr>
<td>4</td>
<td>3.45</td>
<td>3.98</td>
<td>13.79</td>
<td>1430</td>
<td>5.85</td>
</tr>
<tr>
<td>8</td>
<td>1.73</td>
<td>7.94</td>
<td>12.55</td>
<td>1600</td>
<td>8.19</td>
</tr>
<tr>
<td>16</td>
<td>0.88</td>
<td>15.56</td>
<td>13.61</td>
<td>2000</td>
<td>16.00</td>
</tr>
<tr>
<td>32</td>
<td>0.47</td>
<td>29.20</td>
<td>13.92</td>
<td>2500</td>
<td>31.25</td>
</tr>
<tr>
<td>64</td>
<td>0.71</td>
<td>19.26</td>
<td>13.83</td>
<td>2600</td>
<td>35.15</td>
</tr>
</tbody>
</table>

Memory Capacity = f(Size^2), Compute = f(Size^3)

Strong vs. Weak Scaling
The switch in ~2004 from 1 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 1 inefficient processor with multiple efficient processors
III. OpenMP was a breakthrough in ~2000 that made parallel programming easy

**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+=NUM_THREADS) {
    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: Why Multicore?**

The switch in ~2004 from 1 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 1 inefficient processor with multiple efficient processors
III. OpenMP was a breakthrough in ~2000 that made parallel programming easy

**Peer Instruction: No 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+=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{orange} omp_get_num_threads();
  B{green} Constant for number of blocks in cache
  C{pink} Constant for size of block in cache
  D{yellow} Constant for size of blocks in doubles
```

**Administivia**

- Lab 8, Data-Level Parallelism
  - Start this early, because you’ll need the SSE experience for project 3a
- Project 3a, optimizing matrix manipulation code
  - Part a: just single-thread optimizations – DON’T USE OpenMP
  - (Part b: next week, use OpenMP too)
- HW#5
Midterm Regrade Request Policy

• NO REQUESTS ACCEPTED UNTIL LECTURE WED OCTOBER 17, i.e., we’ll simply delete any that come before
• Must attend discussion section to learn solutions and grading process – TA signoff needed for regrade request!
• Regrade requests must be accompanied by written request explaining rationale for regrade.
• Modifying your copy of exam punishable by F and letter in your University record
• We reserve right to regrade whole exam

Types of Synchronization

• Parallel threads run at varying speeds, need to synchronize their execution when accessing shared data.
• Two basic classes of synchronization:
  – Producer-Consumer
    • Consumer thread(s) wait(s) for producer thread(s) to produce needed data
    • Deterministic ordering. Consumer always runs after producer (unless there’s a bug!)
  – Mutual Exclusion
    • Any thread can touch the data, but only one at a time.
    • Non-deterministic ordering. Multiple orders of execution are valid.

Simple OpenMP Parallel Sections

• OpenMP Fork and Join are examples of producer-consumer synchronization

  ![Image](http://www.llnl.gov/computing/tutorials/openMP/)

Barrier Synchronization

• Barrier waits for all threads to complete a parallel section. Very common in parallel processing.
• How does OpenMP implement this?

Barrier: First Attempt (pseudo-code)

```c
int n_working = NUM_THREADS; /* Shared variable*/
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID); /* Do my chunk of work. */
    /* Barrier code. */
    n_working -= 1; /* I’m done */
    if (ID == 0) { /* Master */
        while (n_working != 0) { /* Master spins until everyone finished */
            ; /* master spins until everyone finished */
        } else {
            /* Put thread to sleep if not master */
        }
}
```

Flashcard quiz: Implementing Barrier Count decrement

• Thread #1
  /* n_working -- 1 */
  lw $t0, ($s0)
  addiu $t0, -1
  sw $t0, ($s0)
• Thread #2
  /* n_working -- 1 */
  lw $t0, ($s0)
  addiu $t0, -1
  addiu $t1, -1
  lw $t0, ($s0)
  sw $t0, ($s0)

If initially n_working = 5, what are possible final values after both threads finish above code sequence?

• n_working = 3 only
• n_working = 3, or n_working = 4 only
• n_working = 3, 4, or 5 only
• Undefined
Review: Data Races and Synchronization

- 2 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

Decrement of Barrier Variable is Example of Mutual Exclusion

- Want each thread to *atomically* decrement the *n_working* variable
  - Atomic from Greek “Atomos” meaning indivisible!
- Ideally want:
  - Begin atomic section /*Only one thread at a time*/
    - `lw $t0, ($s0)`
    - `addiu $t0, -1`
    - `sw $t0, ($s0)`
  - End atomic section/*Allow another thread in */

New Hardware Instructions

For some common useful cases, some instruction sets have special instructions that atomically read-modify-write a memory location

Example:

```
fetch-and-add r_dest, (r_address), r_val implemented as:
r_dest = Mem[r_address] //Return old value in register
t = r_dest + r_val     // Updated value
Mem[r_address] = t     //Increment value in memory
```

Simple common variant: *test-and-set* `r_dest`, `(r_address)`

Atomically reads old value of memory into `r_dest`, and puts 1 into memory location. Used to implement locks

Use locks for more general atomic sections

Atomic sections commonly called “critical sections”

```
Acquire(lock) /* Only one thread at a time*/
/* Critical Section Code */
Release(lock) /* Allow other threads into section */
```

- A lock is a variable in memory (one word)
- Hardware atomic instruction, e.g., test-and-set, checks and sets lock in memory

Implementing Barrier Count decrement with locks

```
/* Acquire lock */
spin:
testandset $t0, ($s1) /* $s1 has lock address */
bnez $t0, spin
lw $t0, ($s0)
addiu $t0, -1
sw $t0, ($s0)
/* Release lock */
sw $zero, ($s1)   /*Regular store releases lock*/
```

MIPS Atomic Instructions

- Splits atomic into two parts:
  - Load Linked *LL rt, offset(base)*
    - Regular load that “reserves” an address
  - Store Conditional *SC rt, offset(base)*
    - Store that only happens if no other hardware thread touched the reserved address
    - Success: rt=1 and memory updated
    - Failure: rt = 0 and memory unchanged
- Can implement test-and-set or fetch-and-add as short code sequence
- Reuses cache snooping hardware to check if other processors touch reserved memory location
ISA Synchronization Support

- All have some atomic Read-Modify-Write instruction
- Varies greatly – little agreement on “correct” way to do this
- No commercial ISA has direct support for producer-consumer synchronization
  – Use mutual exclusion plus software to get same effect (e.g., barrier in OpenMP)
- This area is still very much “work-in-progress” in computer architecture

OpenMP Critical Sections

```c
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  foo(ID); /* Do my chunk of work. */
}
#pragma omp critical
{ /* Only one thread at a time */
  /* shared_variable_updates */
}
```

And in Conclusion, ...

- MatrixMultiply speedup versus scaleup
  – Strong versus weak scaling
- Synchronization:
  – Producer-consumer versus mutual-exclusion
- Hardware provides some atomic instructions
  – Software builds up other synchronization using these