CS 61C: Great Ideas in Computer Architecture

Synchronization, OpenMP

Guest Lecturer: Justin Hsia
Review of Last Lecture

• Multiprocessor systems uses shared memory (single address space)

• Cache coherence implements shared memory even with multiple copies in multiple caches
  – Track state of blocks relative to other caches (e.g. MOESI protocol)
  – False sharing a concern
Great Idea #4: Parallelism

**Software**
- Parallel Requests
  Assigned to computer
  e.g. search “Garcia”
- 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 functioning in parallel at same time

**Hardware**
- Warehouse Scale Computer
  Leverage Parallelism & Achieve High Performance
- Core
- ... Core
- Memory
- Input/Output
- Instruction Unit(s)
- Functional Unit(s)
  \[ A_0 + B_0, A_1 + B_1, A_2 + B_2, A_3 + B_3 \]
- Cache Memory
- Logic Gates

3/15/2013  Spring 2013 -- Lecture #22
Agenda

• Synchronization - A Crash Course
• Administrivia
• OpenMP Introduction
• OpenMP Directives
  – Workshare
  – Synchronization
• Bonus: Common OpenMP Pitfalls
Data Races and Synchronization

• Two memory accesses form a *data race* if different threads access the same location, and at least one is a write, and they occur one after another
  – Means that the result of a program can vary depending on chance (which thread ran 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
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:
  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)
  ```

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:* \( \text{ll } rt, \text{off}(rs) \)

• *Store conditional:* \( \text{sc } rt, \text{off}(rs) \)
  – Returns 1 (success) if location has not changed since the \( \text{ll} \)
  – Returns 0 (failure) if location has changed

• Note that \( \text{sc} \) *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:
  $s4 \leftrightarrow \text{Mem}($s1)

```assembly
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

\textbf{sc} would fail if another threads executes \textbf{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
  ll $t1,0($s1)
  bne $t1,$zero,Try
  sc $t0,0($s1)
  beq $t0,$zero,try

  Locked:

  # critical section

  Unlock:

  sw $zero,0($s1)
Question: 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)
```
Agenda

• Synchronization - A Crash Course
• Administrivia
• OpenMP Introduction
• OpenMP Directives
  – Workshare
  – Synchronization
• Bonus: Common OpenMP Pitfalls
Administrivia

• Midterm re-grade requests due Tuesday (3/19)

• Project 2: MapReduce
  – Work in groups of two!
  – Part 1: Due March 17 (this Sunday)
  – Part 2: Due March 24 (part of Spring Break)

• Homework 4 will be posted before Spring Break
  – If you want to get a head start
Agenda

• Synchronization - A Crash Course
• Administrivia
• OpenMP Introduction
• OpenMP Directives
  – Workshare
  – Synchronization
• Bonus: Common OpenMP Pitfalls
What is OpenMP?

• API used for multi-threaded, shared memory parallelism
  — Compiler Directives
  — Runtime Library Routines
  — Environment Variables

• Portable

• Standardized

• Resources:  http://www.openmp.org/
  and http://computing.llnl.gov/tutorials/openMP/
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:

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

– 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:
    ```c
    #pragma omp parallel private (x)
    ```

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

• OpenMP intrinsic to set number of threads:
  ```c
  omp_set_num_threads(x);
  ```

• OpenMP intrinsic to get number of threads:
  ```c
  num_th = omp_get_num_threads();
  ```

• OpenMP intrinsic to get Thread ID number:
  ```c
  th_ID = omp_get_thread_num();
  ```
#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 */
}
Agenda

• Synchronization - A Crash Course
• Administrivia
• OpenMP Introduction
• OpenMP Directives
  – Workshare
  – Synchronization
• Bonus: Common OpenMP Pitfalls
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

```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
Building Block: \texttt{for} loop

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

- Break \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}

\textit{In general, don’t jump outside of any pragma block}
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 `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*)
OpenMP Directives (Synchronization)

• These are defined *within a parallel section*
  
  • *master*
    – Code block executed only by the master thread (all other threads skip)
  
  • *critical*
    – Code block executed by only one thread at a time
  
  • *atomic*
    – Specific memory location must be updated atomically (like a mini-*critical* section for writing to memory)
    – Applies to single statement, not code block
OpenMP Reduction

- **Reduction** specifies that one or more private variables are the subject of a reduction operation at end of parallel region
  - **Clause** reduction(operation:var)
  - **Operation**: Operator to perform on the variables 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;
```
Summary

• Data races lead to subtle parallel bugs
• Synchronization via hardware primitives:
  – MIPS does it with Load Linked + Store Conditional
• OpenMP as simple parallel extension to C
  – During parallel fork, be aware of which variables should be shared vs. private among threads
  – Work-sharing accomplished with for/sections
  – Synchronization accomplished with critical/atomic/reduction
BONUS SLIDES

You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture. They have been prepared in a way that should be easily readable and the material will be touched upon in the following lecture.
Agenda

• Synchronization - A Crash Course
• Administrivia
• OpenMP Introduction
• OpenMP Directives
  – Workshare
  – Synchronization
• Bonus: Common OpenMP Pitfalls
OpenMP Pitfall #1: Data Dependencies

• Consider the following code:
  
  \[
  \begin{align*}
  a[0] &= 1; \\
  \text{for}(i=1; i<5000; i++) \\
  &\quad a[i] = i + a[i-1];
  \end{align*}
  \]

• There are dependencies between loop iterations!
  
  – Splitting this loop between threads does not guarantee in-order execution
  
  – Out of order loop execution will result in undefined behavior (i.e. likely wrong result)
Open MP Pitfall #2: Sharing Issues

• Consider the following loop:

```c
#pragma omp parallel for
for(i=0; i<n; i++){
    temp = 2.0*a[i];
    a[i] = temp;
    b[i] = c[i]/temp;
}
```

• `temp` is a shared variable!

```c
#pragma omp parallel for private(temp)
for(i=0; i<n; i++){
    temp = 2.0*a[i];
    a[i] = temp;
    b[i] = c[i]/temp;
}
```
OpenMP Pitfall #3: Updating Shared Variables Simultaneously

• Now consider a global sum:

\[
\text{for}(i=0; \ i<n; \ i++) \\
\hspace{1cm} \text{sum} = \text{sum} + a[i];
\]

• This can be done by surrounding the summation by a critical/atomic section or reduction clause:

\[
\text{#pragma omp parallel for reduction(+:sum)} \\
\{ \\
\hspace{1cm} \text{for}(i=0; \ i<n; \ i++) \\
\hspace{2cm} \text{sum} = \text{sum} + a[i]; \\
\}
\]

– Compiler can generate highly efficient code for reduction
OpenMP Pitfall #4: Parallel Overhead

• Spawning and releasing threads results in significant overhead

• Better to have fewer but larger parallel regions
  – Parallelize over the largest loop that you can (even though it will involve more work to declare all of the private variables and eliminate dependencies)
OpenMP Pitfall #4: Parallel Overhead

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

Too much overhead in thread generation to have this statement run this frequently.

Poor choice of loop to parallelize.