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

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

Review
- Amdahl’s Law: Serial sections limit speedup
- Flynn Taxonomy
- Intel SSE SIMD Instructions
  - Exploit data-level parallelism in loops
  - One instruction fetch that operates on multiple operands simultaneously
  - 128-bit XMM registers
- SSE Instructions in C
  - Embed the SSE machine instructions directly into C programs through use of intrinsics
  - Achieve efficiency beyond that of optimizing compiler

New-School Machine Structures
(It’s a bit more complicated!)

- 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

Simple Multiprocessor

- Each processor has its own PC and executes an independent stream of instructions (MIMD)
- Different processors can access the same memory space
  - Processors communicate via shared memory by storing/loading to/from common locations
- Two ways to use a multiprocessor:
  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

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

Transition to Multicore

- Intel Pentium 4
- AMD Phenom (4 cores)
- Transistors (Thousands)
- Parallel App Performance
- Frequency (MHz)
- Typical Power (Watts)
- Number of Cores

4/2/15
### Parallelism Only Path to Higher Performance

- Sequential processor performance not expected to increase much, and might go down
- If want apps with more capability, have to embrace parallel processing (SIMD and MIMD)
- In mobile systems, use multiple cores and GPUs
- In warehouse-scale computers, use multiple nodes, and all the MIMD/SIMD capability of each node

### Potential Parallel Performance (assuming SW can use it)

<table>
<thead>
<tr>
<th>Year</th>
<th>Cores</th>
<th>SIMD bits /Core</th>
<th>Core + SIMD bits</th>
<th>Peak GP FLOPs/Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>MIMD 2</td>
<td>128</td>
<td>256</td>
<td>MIMD 4</td>
</tr>
<tr>
<td>2005</td>
<td>4/ 2X/ 2yrs 6</td>
<td>128</td>
<td>512</td>
<td>8</td>
</tr>
<tr>
<td>2007</td>
<td>8/ 4yrs</td>
<td>128</td>
<td>768</td>
<td>12</td>
</tr>
<tr>
<td>2009</td>
<td>10/ 2.5X</td>
<td>256</td>
<td>2560</td>
<td>40</td>
</tr>
<tr>
<td>2011</td>
<td>12/ 8X</td>
<td>256</td>
<td>3072</td>
<td>48</td>
</tr>
<tr>
<td>2013</td>
<td>14/ 16</td>
<td>512</td>
<td>7168</td>
<td>112</td>
</tr>
<tr>
<td>2015</td>
<td>16/ 20X</td>
<td>512</td>
<td>8192</td>
<td>128</td>
</tr>
<tr>
<td>2017</td>
<td>18/ 28X</td>
<td>1024</td>
<td>18432</td>
<td>112</td>
</tr>
<tr>
<td>2019</td>
<td>20/ 32X</td>
<td>1024</td>
<td>20480</td>
<td>320</td>
</tr>
</tbody>
</table>

### Threads

- **Thread:** a sequential flow of instructions that performs some task
- Each thread has a PC + processor registers and accesses the shared memory
- Each processor provides one (or more) hardware threads (or harts) that actively execute instructions
- Operating system multiplexes multiple software threads onto the available hardware threads

### Operating System Threads

Give the illusion of many active threads by time-multiplexing software threads onto hardware threads
- Remove a software thread from a hardware thread by interrupting its execution and saving its registers and PC into memory
  - Also if one thread is blocked waiting for network access or user input
- Can make a different software thread active by loading its registers into a hardware thread’s registers and jumping to its saved PC

### Hardware Multithreading

- Basic idea: Processor resources are expensive and should not be left idle
- Long memory latency to memory on cache miss?
- Hardware switches threads to bring in other useful work while waiting for cache miss
- Cost of thread context switch must be much less than cache miss latency
- Put in redundant hardware so don’t have to save context on every thread switch:
  - PC, Registers
- Attractive for apps with abundant TLP
  - Commercial multi-user workloads
Hardware Multithreading

- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next

Multithreading vs. Multicore

- Multithreading => Better Utilization
  - ~1% more hardware, 1.10X better performance?
  - Share integer adders, floating-point units, all caches (L1 I, L1 D, L2, L3), Memory Controller
- Multicore => Duplicate Processors
  - ~50% more hardware, ~2X better performance?
  - Share outer caches (L2, L3), Memory Controller
- Modern machines do both
  - Multiple cores with multiple threads per core

Krste’s MacBook Air

- /usr/sbin/sysctl -a | grep hw\.
  hw.model = MacBookAir5,1
  hw.cache_linesize = 64
  hw.L1cachesize: 32,768
  hw.L1dcachesize: 32,768
  hw.L2cachesize: 262,144
  hw.L3cachesize: 4,194,304
  hw.cpu_frequency = 2,000,000,000
  hw.memsize = 8,589,934,592

Machines in (old) 61C Lab

- /usr/sbin/sysctl -a | grep hw\.
  hw.model = MacPro4,1
  hw.cache_linesize = 64
  hw.L1cachesize: 32,768
  hw.L1dcachesize: 32,768
  hw.L2cachesize: 262,144
  hw.L3cachesize: 8,388,608
  hw.physicalcpu: 8
  hw.logicalcpu: 16
  hw.cpu_frequency = 2,260,000,000
  hw.physmem = 2,147,483,648

  Therefore, should try up to 16 threads to see if performance gain even though only 8 cores

Administrivia

100s of (Mostly Dead) Parallel Programming Languages

| Language        | ActorScript | Concurrent Pascal | Ada | Concurrent ML | Eiffel | Chapel | Clean | Cilk | Concurrent C | Clojure | Haskell | Java | .NET | Julia | LabVIEW | Modula-3 | Modula-3 | SALSA | SALSA | SALSA | SALSA | SALSA | SALSA | SALSA | SALSA | SALSA |
|-----------------|-------------|------------------|-----|---------------|-------|--------|-------|------|--------------|---------|---------|------|------|------|--------|---------|---------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|---
OpenMP

- OpenMP is a language extension used for multi-threaded, shared-memory parallelism
  - Compiler Directives (inserted into source code)
  - Runtime Library Routines (called from your code)
  - Environment Variables (set in your shell)
- Portable
- Standardized
- Easy to compile: `cc -fopenmp name.c`

Portable Standardized Easy to compile:

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)
    ```
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

What Kind of Threads?

- OpenMP threads are operating system (software) threads.
- OS will multiplex requested OpenMP threads onto available hardware threads.
- Hopefully each gets a real hardware thread to run on, so no OS-level time-multiplexing.
- But other tasks on machine can also use hardware threads!
- Be careful when timing results for project 3!

OMP_NUM_THREADS

- OpenMP intrinsic to set number of threads: `omp_set_num_threads(x)`;
- OpenMP intrinsic to get number of threads: `num_th = omp_get_num_threads()`;
- OpenMP intrinsic to get Thread ID number: `th_ID = omp_get_thread_num()`;

Parallel Hello World

```c
#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 */
}
```

Data Races and Synchronization

- Two 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
- (more later)

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

Possible Lock Implementation

- Lock (a.k.a. busy wait)
  - Get lock:
    ```
    addiu $t1,$zero,1
    Loop: lw $t0,0($s0)
    bne $t0,$zero,Loop
    Lock:
    ```
  - Unlock:
    ```
    sw $zero,0($s0)
    ```

- Any problems with this?

Lock Synchronization (2/2)

- Pseudocode:
  ```
  Check lock
  Set the lock
  Critical section
  (e.g. change shared variables)
  Unset the lock
  ```

Possible Lock Problem

- Thread 1
  ```
  addiu $t1,$zero,1
  Loop: lw $t0,0($s0)
  bne $t0,$zero,Loop
  Lock:
  ```

- Thread 2
  ```
  addiu $t1,$zero,1
  Loop: lw $t0,0($s0)
  bne $t0,$zero,Loop
  Lock:
  ```

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: `ll rt,off(rs)`
- Store conditional: `sc rt,off(rs)`
  - Returns 1 (success) if location has not changed since the `ll`
  - Returns 0 (failure) if location has changed
- Note that `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:
  $s4 \leftrightarrow \text{Mem}(s1)$

try: 
  add $t0,$zero,$s4 # copy value
  li $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
  sc would fail if another thread executes 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
  lw $t0,0($s0)
  addi $t0,$t0,1
  sw $t0,0($s1)

Idea is that not for programmers to use this directly, but as a tool for enabling implementation of parallel libraries

Clickers: Consider the following code when executed concurrently by two threads.

What possible values can result in *($s0)?

A: 101 or 102
B: 100, 101, or 102
C: 100 or 101
D: 102

OpenMP Directives (Work-Sharing)

• These are defined within a parallel section

- #pragma omp parallel
  {            #pragma omp for
    for(i=0;i<len;i++) { ... }  }
  can be shortened to:
  #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<max; i++) zero[i] = 0;
\end{verbatim}
- Break \texttt{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
  - i.e. No \texttt{break, return, exit, goto} statements

Parallel for \texttt{pragma}
\begin{verbatim}
#pragma omp parallel for
for (i=0; i<max; i++) zero[i] = 0;
\end{verbatim}
- 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 \texttt{private} per thread (Why?)
- Implicit synchronization at end of for loop
- Divide index regions sequentially per thread
  - Thread 0 gets 0, 1, …, \(\lfloor \text{max}/n \rfloor \)-1;
  - Thread 1 gets \(\lfloor \text{max}/n \rfloor \), \(\lfloor \text{max}/n \rfloor +1\), …, \(2 \times \lfloor \text{max}/n \rfloor \)-1
  - Why?

OpenMP Timing
- Elapsed wall clock time:
  \begin{verbatim}
  double omp_get_wtime(void);
  \end{verbatim}
  - 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 \texttt{omp_get_wtime} to get elapsed time

Matrix Multiply in OpenMP
\begin{verbatim}
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++)
    for (k=0; k<Pdim; k++)
      tmp = 0.0;
      for ( k=0; k<Pdim; k++)
      \quad /* C(i,j) = sum(over k) A(i,k) * B(k,j)* /
      \quad tmp += \star (A+(i\star Pdim+k)) \times \star (B+(k\star Ndim+j));
      \quad (C+(i\star Ndim+j)) = tmp;
    \}
run_time = omp_get_wtime() - start_time;
\end{verbatim}

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)

And in Conclusion, …
- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multithreading increases utilization, Multicore more processors (MIMD)
- 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