Compsci 590.3:
Introduction to Parallel Computing

Alvin R. Lebeck

Slides based on those from the University of Oregon
Admin

Logistics
• Homework 1
  ▪ Due next Tuesday
  ▪ If you added late, get going!
• Use Piazza for questions…
• Posted group info there and machine assignments

Outline
• Architecture review
• Shared memory
• Races & Synchronization
• Data Parallel / SIMD
• Xeon Phi
High Level Language → Inputs to Digital System

```
High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

transistors (switches) turning on and off

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
```

© Alvin R. Lebeck (some slides From U Oregon material)
Memory Partitions

• Text for instructions
  ▪ add res, src1, src2
  ▪ \text{mem}[\text{res}] = \text{mem}[\text{src1}] + \text{mem}[\text{src2}]

• Data
  ▪ static (constants, global variables)
  ▪ dynamic (heap, \text{malloc} / \text{new} allocated)
  ▪ grows up

• Stack
  ▪ local variables
  ▪ grows down

• Variables are names for memory locations
  ▪ int \text{x};
A Simple Program’s Memory Layout

... int result; // global var
int main()
{
    int x;
    ...
    result = x + result;
    ...
}

For now, think of this as performing:
mem[0x208] = mem[0x400] + mem[0x208]
Stored Program Computer

• **Instructions**: a fixed set of built-in operations

• **Instructions and data are stored in the (same) computer memory**
  - It’s all just Bits

• **Fetch-Execute Cycle**
  ```
  while (!done)
      fetch instruction
      execute instruction
  ```

• This is done by the hardware for speed
Instructions

• Fundamental interface between software and hardware
• Specify types of operations, operand types, etc.
• Many types of operations
  ▪ Integer arithmetic: add, sub, mul, div, mod/rem (signed/unsigned)
  ▪ FP arithmetic: add, sub, mul, div, sqrt
  ▪ Integer logical: and, or, xor, not, sll, srl, sra

• If you’re going to add, you need at least 3 operands
  ▪ Two source operands, one destination operand
  ▪ Note: operands don’t have to be unique (e.g., \( A = B + A \))

Two things we need to know
• Question #1: Where can operands come from?
• Question #2: And how are they specified?
MIPS Instruction Set Architecture Categories

- Arithmetic
  - add, sub, mul, etc
- Logical
  - and, or, shift
- Data Transfer
  - load, store
  - MIPS is LOAD/STORE architecture (next slide)
- Conditional Branch
  - implement if, for, while… statements
- Unconditional Jump
  - support method invocation, function call, procedure calls
MIPS is a Load/Store Instruction Set

• Most instructions operate on operands stored in registers

• A register is a place to hold values that can be named within the instruction

• Like memory, but much smaller
  ▪ 32-128 locations
  ▪ MIPS has 32 (need \(\log_2{32} = 5\) bits to specify)

• Instructions to move operands between memory and registers
  ▪ Load word, load byte, store word, store byte, etc.

• But, Professor Lebeck you said that a variable is a name for a memory location!
  ▪ Usually true, but sometimes a variable might live only in a register...so a variable is a name for a memory location or a register.
LOAD / STORE ISA

• Instruction set:
  add, sub, mult, div, … only on operands in registers

ld, st instructions to move data from and to memory, only way to access memory

Example: a*b - (a+c*b)

<table>
<thead>
<tr>
<th></th>
<th>r1, r2, r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r1, c</td>
<td>2, ?, ?</td>
</tr>
<tr>
<td>ld r2, b</td>
<td>2, 3, ?</td>
</tr>
<tr>
<td>mult r1, r1, r2</td>
<td>6, 3, ?</td>
</tr>
<tr>
<td>ld r3, a</td>
<td>6, 3, 4</td>
</tr>
<tr>
<td>add r1, r1, r3</td>
<td>10, 3, 4</td>
</tr>
<tr>
<td>mult r2, r2, r3</td>
<td>10, 12, 4</td>
</tr>
<tr>
<td>sub r3, r2, r1</td>
<td>10, 12, 2</td>
</tr>
</tbody>
</table>

7 instructions

Memory

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
In Class 9.2: 2-address ISA

- Instruction set: Two explicit operands, one implicit
  add, sub, mult, div, …
  one source operand is also destination (one operand can be in memory)
  add reg1, reg2     reg1 <- reg1 + reg2
  add reg1, a        reg1 <- reg1 + mem[a] (this is simplified)

Example: a*b - (a+c*b)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>reg1, reg2, mem[a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>add reg1, b</td>
<td>3,  ?</td>
<td>a = 4</td>
</tr>
<tr>
<td>mult reg1, c</td>
<td>6,  ?</td>
<td>b = 3</td>
</tr>
<tr>
<td>add reg1, a</td>
<td>10,  ?</td>
<td>c = 2</td>
</tr>
<tr>
<td>add reg2, b</td>
<td>10, 3</td>
<td></td>
</tr>
<tr>
<td>mult reg2, a</td>
<td>10, 12</td>
<td>reg1, reg2, reg3</td>
</tr>
<tr>
<td>sub reg2, reg1</td>
<td>10,  2</td>
<td></td>
</tr>
</tbody>
</table>

6 instructions
Memory Hierarchies

- CPU
  - on-chip cache(s)
- Memory (SRAM)
  - off-chip cache
- Memory (DRAM)
  - main memory
- Disk/FLASH

small expensive $/bit  fast
big  cheap $/bit  slow

•so then where is my program and data??
Memory Hierarchies

- CPU memory
- on-chip cache(s)
- off-chip cache
- main memory
- Memory (SRAM)
- Memory (DRAM)
- Disk/FLASH

small expensive $/bit
fast
big cheap $/bit
slow

© Alvin R. Lebeck (some slides From U Oregon material)
Shared Memory Multiprocessors

- Processors connected to one shared Main Memory
- Memory is equidistant (latency) from all processors (UMA)
- Non-uniform memory architecture (NUMA)
- Kernel can run on any processor (symmetric MP)
- Multicore
  - Intel core i7, etc.
- The details of the memory hierarchy matter for performance, but not to get your program to work correctly (mostly)
Shared Memory Programming Abstraction

- Processors connected to one shared memory
- Read from and write to the same variables in memory
- This is good, easy (easier?) to write parallel programs.
- This is bad, what if two processors read and write the same variable?
The Trouble with Concurrency

• One counter (e.g., monitor students at game in Cameron)
• Two threads (T1,T2) in one address space or two processes in the kernel

Example

```c
#pragma omp parallel
{ count++; }
```

• What are possible values for count after two threads execute the following code?

Possible Answer
A. count
B. count + 1
C. count + 2
D. count + 3
The Trouble with Concurrency

- Called a Race Condition or Data Race

```
ld r2, count
add r1, r2, r3
st count, r1

ld r2, count
add r1, r2, r3
st count, r1
```

Private stack per thread for locals

```
ld (count)
add
st (count+1)
```

Shared Data

```
st (count+1)
count+1
```

© Alvin R. Lebeck (some slides From U Oregon material)
Solution: Atomic Sequence of Instructions

• Atomic Sequence
  ▪ Sequence of operations appears to execute to completion without any intervening operations

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin atomic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld (count)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st (count+1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end atomic</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>begin atomic</td>
<td></td>
</tr>
<tr>
<td>wait</td>
<td></td>
<td>count+1</td>
</tr>
<tr>
<td></td>
<td>ld (count)</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st (count+2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end atomic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

© Alvin R. Lebeck (some slides
From U Oregon material)
HW Support for Atomic Operations

• Could provide direct support in Instruction Set
  ▪ Atomic increment
  ▪ Insert node into sorted list??

• Just provide low level primitives to construct atomic sequences
  ▪ called synchronization primitives
    LOCK(counter->lock); // begin atomic
    counter->value = counter->value + 1;
    UNLOCK(counter->lock); // end atomic
  ▪ All updates to counter->value must be “protected” by Lock/Unlock

• test&set (x) instruction: returns previous value of x and sets x to “1”
  LOCK(x) => while (test&set(x));
  UNLOCK(x) => x = 0;

• load linked store conditional pair is lower level and can be used to implement atomic sequences like test&set, compare & swap, etc.
Solution: Atomic Sequence of Instructions

```plaintext
LOCK(x) => while (test&set(x));
UNLOCK(x) => x = 0;
```
Support for Critical Sections

**openMP**

```c
#pragma critical
Can contain general code block
    #pragma omp critical
    counter++;

#pragma atomic
Less general, lower overhead
    #pragma atomic
    counter++;
```

**Lock routines (subset)**

```c
omp_init_lock(omp_lock_t *lock);
omp_set_lock(omp_lock_t *lock);
omp_unset_lock(omp_lock_t *lock);
```

**TBB**

```cpp
tbb::mutex

tbb::mutex mylock;
mylock.lock();
counter++;
mylock.unlock();
```

**CilkPlus**

Use openmp or TBB lock…
Issue for Locks / Critical Sections

• Granularity
  ▪ How much data do you lock at a time?
  ▪ An element of a matrix, a row the entire matrix
  ▪ A node in a graph, a subgraph, the entire graph

• What do you think the tradeoffs are?

• Deadlock…
  T1 Lock(x), T2 Lock(y)
  T1 Lock(y), T2 Lock(x)
Global Synchronization

- Scan example: One algorithm for parallelizing scan is to perform an “up sweep” and a “down sweep”
- For our own parallel implementation need to wait for up sweep to finish before starting down sweep

Up sweep – compute reduction

Down sweep – compute intermediate values
Global Synchronization: Barrier

- Wait for all tasks to enter, then allow all to continue

```c
#pragma omp parallel for
for(int i=0; i< N; i++) {
    up sweep
    #pragma omp barrier
    down sweep
}
cilk pseudocode
  cilk_spawn upworkers();
cilk_sync;
cilk_spawn downsweepers();
cilk_sync;
```

TBB pseudocode
(yes TBB has a full task model)

```c
tbb::spawn upworkers();
wait_for_all();
tbb_spawn downsweepers();
wait_for_all();
```
Data Parallelism

- Perform same operation on different elements (think map)
  - Single Instruction Multiple Data (SIMD)
  - Vector Operations (load, store, add, sub, mul, div, etc.)

```c
#pragma omp parallel for
for (i=0; i<N; i++) {
    A[i] = (A[i] + B[i]) * C[i];
}
```

- Although this can be partitioned across processors to execute in parallel
- Each processor does one operation at a time
  - ```A[i] = (A[i] + B[i]) * C[i];```  
  - Best case is two operations (+ and *) for each element
Data Parallelism

- Another level of parallelism within one thread (SIMD)

```c
for (i=0; i<N; i++) {
    A[i] = (A[i] + B[i]) * C[i];
}
```

```c
for (i=begin; i<end; i+= veclength) {
    // A[i] = (A[i] + B[i]) * C[i];
    ldvec vr1, A[i];
    ldvec vr2, B[i];
    vecadd vr1, vr1, vr2;
    ldvec vr2, C[i];
    vecmul, vr1, vr1, vr2;
    stvec, A[i], vr1;
}
```
Data Parallel

• Compiler extracts this if you use –O2 or higher optimization
• Note this is not in the serial Makefile
  ▪ may want to add –fast option which should give best perf (but not always)
• If needed you can use an openmp pragma
  ▪ #pragma omp simd
  ▪ #pragma parallel for simd
Processor Support for SIMD

• AVX2
  ▪ 256-bit vectors
  ▪ Most general purpose cores have this,

• AVX-512
  ▪ 512-bit vectors
  ▪ Xeon Phis that we have
  ▪ Next generation systems

• Support for some collective operations on vectors
Xeon Phi Accelerator
Xeon Phi Accelerator
Programs on Xeon Phi

• Two modes
• micnativeloadex (we’ve seen this already)
• Offload mode
  ▪ Run part of your program on the host processor
  ▪ Part of your program on the accelerator
  ▪ This is similar to what we will do with GPUs later this semester

#pragma omp offload target(mic:0)
#pragma omp parallel for
for…

Unfortunately, having some performance issues…
Summary

• Shared memory
• Shared Name space for variables
• Races!
  ▪ Non-deterministic
  ▪ Incorrect behavior
• Synchronization
  ▪ Locks
  ▪ Barriers
• Data Parallelism