CS 61C: Great Ideas in Computer Architecture (Machine Structures) Thread Level Parallelism

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda

- Review
- More OpenMP
- Strong vs. Weak Scaling
- Administrivia
- Technology Break
- Parallel Peer Instruction
- Summary

Review

- Synchronization requires atomic operations
  - Via Load Linked and Store Conditional in MIPS
- Hardware multithreading to get more utilization from processor
- OpenMP is a simple pragma extension to C
  - Threads, Parallel for, private, critical sections, …
- Data races lead to subtle parallel bugs
  - Beware private variables vs. shared variables

Simple Parallelization

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

- For loop must have canonical shape for OpenMP to parallelize it
  - Necessary for run-time system to determine loop iterations
- No premature exits from the loop allowed
  - i.e., No break, return, exit, goto statements

The 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 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.

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

OpenMP Version 3

```
#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

OpenMP Timing

- omp_get_wtime – Elapsed wall clock time

```
double omp_get_wtime(void);
#include <omp.h> // to get function
```

- Elapsed wall clock time in seconds. The 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". On POSIX compliant systems the seconds since the Epoch (00:00:00 UTC, January 1, 1970) are returned.

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 (!) shared L3 cache / chip

Matrix Multiply in OpenMP

```
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++){
            tmp = 0.0;
        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;
```

Note: Outer loop spread across N threads; inner loops inside a 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 SSE3 Instructions to raise floating point computation rate

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 peformance is to solve a lot bigger problem

32 Core: Speed-up vs. Scale-up

Speed-up			Scale-up: Fl. Pt. Ops = 2 x Size^3		
Threads	Time	Speedup	Time	Size (Dim)	Fl. Ops x 10^9
1	13.75	1.00	13.75	1000	2.00
2			13.52	1240	3.81
4			13.79	1430	5.85
8			12.55	1600	8.19
16			13.61	2000	16.00
32			13.92	2500	31.25
64			13.83	2600	35.15

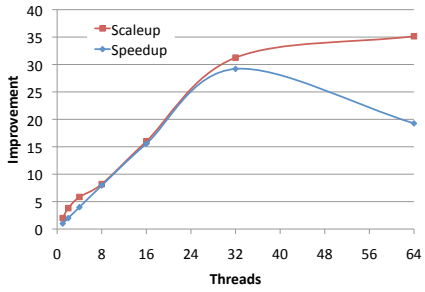
Memory Capacity = f(Size^2), Compute = f(Size^3)

32 Core: Speed-up vs. Scale-up

Speed-up			Scale-up: Fl. Pt. Ops = 2 x Size^3		
Threads	Time (secs)	Speedup	Time (secs)	Size (Dim)	Fl. Ops x 10^9
1	13.75	1.00	13.75	1000	2.00
2	6.88	2.00	13.52	1240	3.81
4	3.45	3.98	13.79	1430	5.85
8	1.73	7.94	12.55	1600	8.19
16	0.88	15.56	13.61	2000	16.00
32	0.47	29.20	13.92	2500	31.25
64	0.71	19.26	13.83	2600	35.15

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

Strong vs. Weak Scaling



Agenda

- Review
- More OpenMP
- Strong vs. Weak Scaling
- Administrivia
- Technology Break
- Parallel Peer Instruction
- Summary

Administrivia

- Turn in your written regrade petitions with your exam to your TA *by tomorrow Tuesday Oct 19*
- Make sure all grades are correct but Project 4, Final Exam by December 1
- Final Exam 8-11AM (TBD) Monday Dec 13

61C in the News

"Why CPUs Aren't Getting Any Faster" MIT Technology Review 10/12

The Central Processing Unit (CPU)--the component that has defined the performance of your computer for many years--has hit a wall.

In fact, the next-generation of CPUs, including Intel's forthcoming Sandy Bridge processor, have to contend with multiple walls--a memory bottleneck (the bandwidth of the channel between the CPU and a computer's memory); the instruction level parallelism (ILP) wall (the availability of enough discrete parallel instructions for a multi-core chip) and the power wall (the chip's overall temperature and power consumption).

Of the three, the power wall is now arguably the defining limit of the power of the modern CPU. As CPUs have become more capable, their energy consumption and heat production has grown rapidly. It's a problem so tenacious that chip manufacturers have been forced to create "systems on a chip"--conurbations of smaller, specialized processors. These systems are so sprawling and diverse that they've caused long-time industry observers question whether the original definition of a CPU even applies to today's chips.

Thanks to Sridatta Thatipamala for suggestion!

Peer Instruction: Why Multicore?

The switch in ~ 2005 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. The "memory wall" meant need to find something else to compute during the long latency to memory
III. OpenMP was a breakthrough in ~2000 that made parallel programming easy
IV. There was no other performance option but replacing 1 ineffecient processor with multiple efficient processors

A)(red)	I only	E)(burgundy)	I and IV
B)(orange)	II only	F)(blue)	II and IV
C)(green)	III only	G)(purple)	III and IV
D)(yellow)	IV only	H)(Olive)	I, III and IV

100s of (dead) Parallel Programming Languages

ActorScript	Concurrent Pascal	JoCaml	Orc
Ada	Concurrent ML	Join	Oz
Afnix	Concurrent Haskell	Java	Pict
Alef	Curry	Joule	Reia
Alice	CUDA	Joyce	SALSA
APL	E	LabVIEW	Scala
Axum	Eiffel	Limbo	SISAL
Chapel	Erlang	Linda	SR
Cilk	Fortan 90	MultiLisp	Stackless Python
Clean	Go	Modula-3	SuperPascal
Clojure	Io	Occam	VHDL
Concurrent C	Janus	occam-π	XC

False Sharing in OpenMP

```
{    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=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 ≥ 16 bytes

Peer Instruction: No False Sharing

```
{    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=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)(red)	`omp_get_num_threads();`
B)(orange)	Constant for number of blocks in cache
C)(green)	Constant for size of block in bytes
D)(yellow)	Constant for size of blocks in doubles

Review Locks, LL/SC

- Lock used to create region ("*critical section*") where only 1 processor can operate
- Processors read lock to see if must wait, or OK to go into critical section (and set to locked)
  0 => lock is free, 1 => lock is taken
- Atomic via Load linked, Store conditional:
  - SC Succeeds if location not changed since the ll
    - Returns 1 in rt (clobbers register value being stored)
      Memory[rs+offset]=Reg[rt]; Reg[rt]= 1
  - SC Fails if location is changed
    - Returns 0 in rt (clobbers register value being stored)
      Memory[rs+offset]=Reg[rt]; Reg[rt]= 0

Peer Instruction: What's Next in Lock?

```
Try: addiu $t0,$zero,1 ;copy locked value
     ll  $t1,0($s1)    ;load linked
```

A)
```
     sc  $t0,0($s1)    ;==0, store conditional
     beq $t0,$zero,try ;branch store fails, try
```

B)
```
     beq $t1,$zero,try ;loop if lock was 1
     sc  $t0,0($s1)    ;==0, store conditional
     beq $t0,$zero,try ;branch store fails, try
```

C)
```
     beq $t1,$zero,try ;loop if lock was 1
     sc  $t0,0($s1)    ;==0, store conditional
     bne $t0,$zero,try ;branch store fails, try
```

D)
```
     bne $t1,$zero,try ;loop if lock was 1
     sc  $t0,0($s1)    ;==0, store conditional
     beq $t0,$zero,try ;branch store fails, try
```

E)
```
     bne $t1,$zero,try ;loop if lock was 1
     sc  $t0,0($s1)    ;==0, store conditional
     bne $t0,$zero,try ;branch store fails, try
```

Review MOESI Cache Coherency

1. *Shared*: up-to-date data, other caches may have copy
2. *Modified*: up-to-date data, changed (dirty), no other cache has copy, OK to write, memory out-of-date
3. *Exclusive*: up-to-date data, no other cache has copy, OK to write, memory up-to-date
4. *Owner*: up-to-date data, other caches may have a copy (they must be in Shared state)

I. If in Exclusive state, processor can write without notifying other caches

II. Owner state is variation of Shared state to let caches supply data instead of going to memory on read miss

III. Exclusive state is variation of Modified state to let caches avoid writing to memory on a miss

A)(red)	I only	D)(yellow)	I and II
B)(orange)	II only	E)(burgundy)	II and III
C)(green)	III only	F)(blue)	I, II and III

Summary

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multiprocessor/Multicore uses Shared Memory
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!
- Data races lead to subtle parallel bugs
- Synchronization via atomic operations:
  - MIPS does it with Load Linked + Store Conditional
- OpenMP as simple parallel extension to C
  - Threads, Parallel for, private, critical sections, …