CS 61C: Great Ideas in Computer Architecture *Thread Level Parallelism (TLP)*

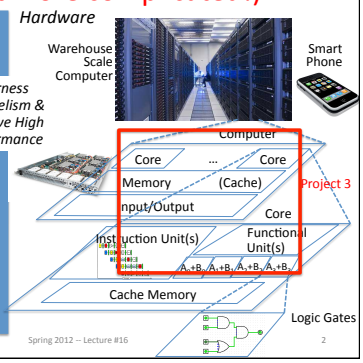
Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

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

Software | *Hardware*

- 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

Harness Parallelism & Achieve High Performance



Review

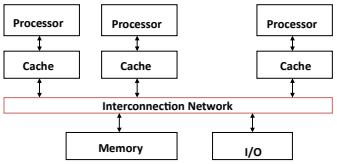
- Amdhal's Cruel Law: Law of Diminishing Returns
- Loop Unrolling to Expose Parallelism
- Optimize Miss Penalty via Memory system
- As the field changes, cs61c has to change too!
- Still about the software-hardware interface
  - Programming for performance via measurement!
  - Understanding the memory hierarchy and its impact on application performance
  - Unlocking the capabilities of the architecture for performance: SIMD

Agenda

- MIMD vs. SIMD
- Cache Coherency
- Threads
- Administrivia
- OpenMP
- Strong vs. Weak Scaling
- Parallel Peer Instruction
- Summary

Parallel Processing: Multiprocessor Systems (MIMD)

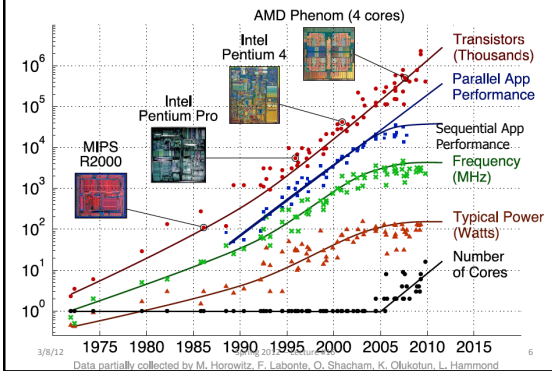
- Multiprocessor (MIMD): a computer system with at least 2 processors



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**

Now Use term *core* for processor ("Multicore") because "Multiprocessor Microprocessor" too redundant

Transition to Multicore



Multiprocessors and You

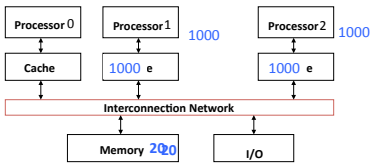
- Only path to performance is parallelism
  - Clock rates flat or declining
  - SIMD: 2X width every 3-4 years
    - 128b wide now, 256b 2011, 512b in 2014?, 1024b in 2018?
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, …
- A key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication
- Project 3: fastest code on 8 processor computers
  - 2 chips/computer, 4 cores/chip

Potential Parallel Performance (assuming SW can use it)

Year	Cores	SIMD bits /Core	Core * SIMD bits	Peak DP FLOPs
2003	MIMD 2	SIMD 128	256	MIMD 4
2005	+2/ 4	2X/ 128	512	*SIMD 8
2007	2yrs 6	4yrs 128	768	12
2009	8	128	1024	16
2011	10	256	2560	40
2013	12	256	3072	48
2015	2.5X 14	8X 512	7168	20X 112
2017	16	512	8192	128
2019	18	1024	18432	288
2021	20	1024	20480	320

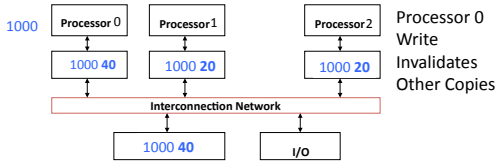
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



Keeping Multiple Caches Coherent

- Architect's job: shared memory => keep cache values coherent
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate all other copies
- Shared written result can "ping-pong" between caches

Cache Coherency and Block Size

- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?
- Effect called *false sharing*
- How can you prevent it?

Threads

- *thread of execution*: smallest unit of processing scheduled by operating system
- On 1 processor, multithreading occurs *by time-division multiplexing*:
  - Processor switched between different threads
  - *Context switching* happens frequently enough user perceives threads as running at the same time
- On a multiprocessor, threads run at the same time, with each processor running a thread

Multithreading vs. Multicore

- 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, L1 caches?
- Attractive for apps with abundant TLP
  - Commercial multi-user workloads

Multithreading vs. Multicore

- Multithreading => Better Utilization
  - ≈1% more hardware, 1.10X better performance?
  - Share integer adders, floating point adders, caches (L1 I $, L1 D$, L2 cache, L3 cache), Memory Controller
- Multicore => Duplicate Processors
  - ≈50% more hardware, ≈2X better performance?
  - Share lower caches (L2 cache, L3 cache), Memory Controller

Machines in (old) 61C Lab

- `/usr/sbin/sysctl -a | grep hw\.`

hw.model = MacPro4,1
…
hw.physicalcpu: 8
hw.logicalcpu: 16
…
hw.cpufrequency = 2,260,000,000
hw.physmem = 2,147,483,648

hw.cachelinesize = 64
hw.l1icachesize: 32,768
hw.l1dcachesize: 32,768
hw.l2cachesize: 262,144
hw.l3cachesize: 8,388,608

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

Administrivia

- Midterm answers and grading ruberic online
- Turn in your written regrade petitions with your exam to your TA by next discussion section
- Long term administrivia: Make sure all grades are correct but Project 4 by Friday April 20
- Final Exam 11:30-2:30 (TBD) Wed May 9

Did Well on Midterm!

- Mean: 77.1
- Mode: 75.0
- Standard deviation: 14.7
- Minimum: 27.0
- 1st quartile: 68.0
- 2nd quartile (median): 79.5
- 3rd quartile: 88.5
- Maximum: 99.0
- Max possible: 100.0

```
25.0 - 30.0:  1 *
30.0 - 35.0:  0
35.0 - 40.0:  4 ***
40.0 - 45.0:  4 ***
45.0 - 50.0:  3 **
50.0 - 55.0: 12 *******
55.0 - 60.0:  8 *****
60.0 - 65.0: 16 *********
65.0 - 70.0: 23 *************
70.0 - 75.0: 21 ************
75.0 - 80.0: 37 ********************
80.0 - 85.0: 34 *******************
85.0 - 90.0: 33 ******************
90.0 - 95.0: 31 *****************
95.0 -100.0:25 **************
```

Survey

- What is going well?
  - Projects, interesting concepts, course revision, labs, discussions, lecture organization
- What would you like to see more of?
  - More examples, clearer project descriptions
- What is the most important thing you have learned?
  - Study habits (read before lecture), MapReduce, MIPS/ assembly language, memory hierarchy, how it all fits together (C->MIPS->Machine)
  - "MapReduce and pointer manipulation. I've already used them in interviews."

Survey Cont'd

- Which topics that you feel unsure about?
  - Caches, MapReduce, pointers
- Topics you understand really well?
  - Numbers, MIPS, MapReduce, caches
- 40% read book before lecture (!)
- 65% like or OK with student roulette
- 75% rarely miss a lecture
- 82% like or OK with Peer Instruction
- 88% like or OK with 61C in the News
- 88% like or OK with Get to know your Prof

61c in the News

- Today is International Women's Day
- Grace Murray Hopper(1906–1992) one of the first programmers, developed the first compiler, and was referred to as "Amazing Grace."
- Conference in her name
- She became a rear admiral in the US Navy, and in 1997 a warship was named for her: the USS Hopper.

OpenMP

- OpenMP is an API used for multi-threaded, shared memory parallelism
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Portable
- Standardized
- Easy to compile: `cc –fopenmp name.c`

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

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

OpenMP Extends C with Pragmas

- Pragmas are a mechanism C provides for language extensions
- Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes
- 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

Fork/Join Parallelism

- Start out executing the program with one master thread
- Master thread *forks* worker threads as enter parallel code
- Worker threads *join* (die or suspend) at end of parallel code

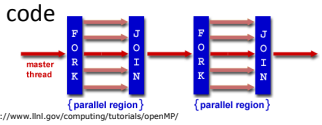


Image courtesy of http://www.llnl.gov/computing/tutorials/openMP/

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

Thread Creation

- How many threads will OpenMP create?
  - Can set via `omp_set_num_threads(NUM_THREADS);`
  - Presumably = number of processors in computer running program

Invoking Parallel Threads

```
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  foo(ID);
}
```

- Each thread executes a copy of the within the structured structured block
- OpenMP intrinsic to get Thread ID number: `omp_get_thread_num()`

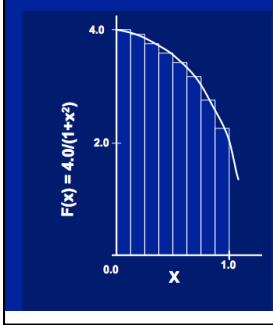
π

3.
14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410 2
...

- Pi Day is 3-14 (started at SF Exploratorium)

Calculating π

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Sequential Calculation of π in C

```
#include <stdio.h>/* Serial Code */
static long num_steps = 100000; double step;
void main ()
{   int i;      double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    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 = %6.12f\n", pi);
}
```


OpenMP Version (with bug)

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{   int i;    double  x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
#pragma omp parallel private (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);
    }
}
    for(i=0, pi=0.0; i<NUM_THREADS; i++)
      pi += sum[i] ;
  printf ("pi = %6.12f\n", pi  / num_steps);
}
```


Experiment

- Run with NUM_THREADS = 1 multiple times
- Run with NUM_THREADS = 2 multiple times
- What happens?

OpenMP Version (with bug)

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{   int i;    double  x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
#pragma omp parallel private (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);
    }
}
    for(i=0, pi=0.0; i<NUM_THREADS; i++)
      pi += sum[i] ;
  printf ("pi = %6.12f\n", pi  / num_steps);
}
```

Note: loop index variable `i` is shared between threads

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 Reduction Version

```
#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.

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) 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 (!) shared L3 cache / chip

Experiment

- Try compile and run at NUM_THREADS = 64
- Try compile and run at NUM_THREADS = 64 with –O2
- Try compile and run at NUM_THREADS = 32, 16, 8, … with –O2

Review: Strong vs Weak Scaling

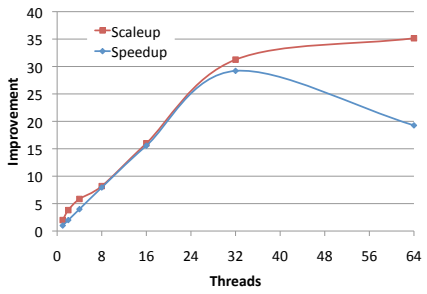
- Strong scaling: problem size fixed
- Weak scaling: problem size proportional to increase in number of processors
  - Speedup on multiprocessor while keeping problem size fixed is harder than speedup by increasing the size of the problem
  - But a natural use of a lot more performance is to solve a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

32 Core: Speed-up vs. Scale-up

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



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 ineffecient processor with multiple efficient processors
III. OpenMP was a breakthrough in ~2000 that made parallel programming easy

A)(orange) I only
B)(green) II only
C)(pink) I & II only
D)(yellow) I, II, & III

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)(orange) `omp_get_num_threads();`

B)(green) **Constant for number of blocks in cache**

C)(pink) **Constant for size of block in bytes**

D)(yellow) **Constant for size of blocks in doubles**

And in Conclusion, …

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multithreading increases utilization, Multicore more processors (MIMD)
- Multiprocessor/Multicore uses Shared Memory
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!
- 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 its easy to get into trouble