CS758: Multicore Programming

Prof. David Wood
Fall 2010

Credits

- Slides based on Milo Martin's CIS 534 slides at Penn, he credits :
  - Intel Academic Community materials and resources
  - UPCRC 2009 Summer School on Multicore Programming
    - Prof. Marc Snir (Illinois)
  - Prof. Katherine Yelick (Berkeley)
  - Prof. David Wood (Wisconsin)
    - Who credits:
      - Prof. Saman Amarasinghe (MIT), Prof. Mark Hill (Wisconsin)
      - Prof. David Patterson (Berkeley), Prof. Marc Snir (Illinois)
  - Prof. Vivek Sarkar (Rice)
    - Who credits:
      - Jack Dongarra (U. Tennessee), John Mellor-Crummey (Rice)
      - Kathy Yelick (Berkeley)
  - David Kirk (NVIDIA) and Wen-mei W. Hwu (Illinois), ECE 498AL

Full Disclosure

- Potential sources of bias or conflict of interest
- I consult for Microsoft Research
- My non-governmental sources of research funding
  - **Google & Microsoft**
- Most of my funding governmental (your tax $$$ at work)
  - Mostly from National Science Foundation (NSF)
  - Also Sandia National Labs
- Collaborators and colleagues
  - Intel, IBM, AMD, Sun, Microsoft, Google, VMWare, etc.
  - (Just about every major computer hardware company)

Why me?

- I'm a computer architect
  - I design hardware, I don't program it
  - I don't know C++ or Java
- Dirty little secret….
  - I used to be a database guy (shh!)
  - Wrote the concurrency control libraries for Synapse Computer Corp.
  - First RDBMS for a microprocessor-based shared memory multiprocessor (back in the early 1980s)
- Dirtier little secret….
  - Not much has changed since then….

Programming Multicores

The Dilbert Approach

Parallel Thinking Exercise: Sorting

- Working in groups (four or more for the class)
  - Develop a method for quickly sorting cards **as a group**
- Think about "communication cost"
  - All cards start face down on table
    - Team members may pick up a card OR put one back
    - Must return to seat after each "communication"
- Think about coordination (aka "synchronization")
  - Team members may coordinate by meeting at end of table
  - No exchange of cards during coordination
- Think about "decomposition"
  - Break problem into smaller pieces

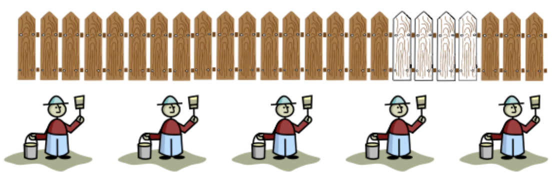
Impediments to Parallelism

Impediments to Parallel Computing

- Identifying "enough" parallelism
  - Problem decomposition (tasks & data)
- Performance
  - Parallel efficiency & scalability
  - Granularity
    - Too small: too much coordination overhead
    - Too large: not enough parallelism, over-stress memory system
  - Load balance
    - Effective distribution of work (statically or dynamically)
  - Memory system: data locality, datasharing, memory bandwidth
  - Synchronization and coordination overheads
- Correctness
  - Incorrect code leads to deadlock, crashes, and/or wrong answers

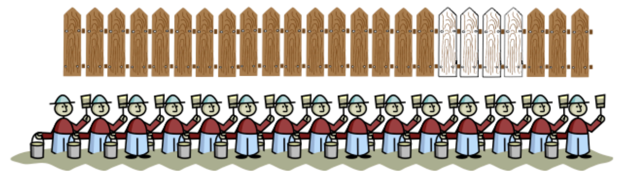
Inherently Parallel

- **Painting a fence:**
  - Work = (picket_painting_time) * (# pickets)
  - Width = (# pickets) / (painter_width)



Granularity

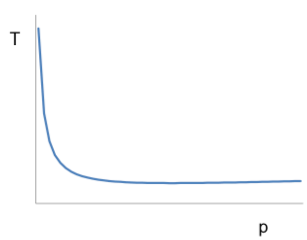
- **Average task size**
  - Cannot be too small



  - Too many painters spoil the fence

More Terminology

- **Running Time** $T_p(N)$ – function of *p*, number of HW threads, and *N*, problem size.
  - Often fix problem size *N*, and look at running time, as function of *p*.



after some point, more processors do not help

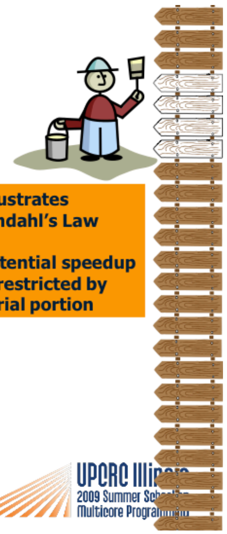
Speedup (Simple)

- Measure of how much faster the computation executes versus the best serial code
  - Serial time divided by parallel time

- Example: *Painting a picket fence*
  - 30 minutes of preparation (serial)
  - One minute to paint a single picket
  - 30 minutes of cleanup (serial)

- Thus, 300 pickets takes 360 minutes (serial time)

Computing Speedup



Number of painters	Time	Speedup
1	30 + 300 + 30 = 360	1.0X
2	30 + 150 + 30 = 210	1.7X
10	30 + 30 + 30 = 90	4.0X
100	30 + 3 + 30 = 63	5.7X
Infinite	*30 + 0 + 30 = 60*	*6.0X*

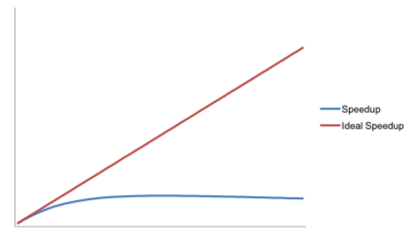
Illustrates Amdahl's Law

Potential speedup is restricted by serial portion

UPCRC Illinois 2009 Summer School on Multicore Programming

Speedup

- $T_1(N)/T_p(N)$



  – Speedup is most often sub-linear

UPCRC Illinois 2009 Summer School on Multicore Programming

Efficiency

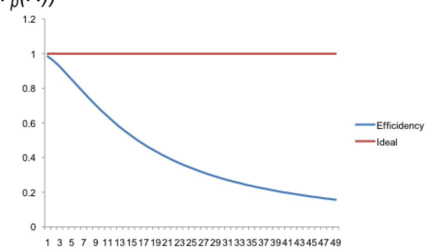
- Measure of how effectively computation resources (threads) are kept busy
  - Speedup divided by number of threads
  - Expressed as average percentage of non-idle time

Number of painters	Time	Speedup	Efficiency
1	360	1.0X	100%
2	30 + 150 + 30 = 210	1.7X	85%
10	30 + 30 + 30 = 90	4.0X	40%
100	30 + 3 + 30 = 63	5.7X	5.7%
Infinite	*30 + 0 + 30 = 60*	*6.0X*	*very low*

UPCRC Illinois 2009 Summer School on Multicore Programming

Efficiency

- $T_1(N)/(pT_p(N))$



  – Efficiency is <1 and decreasing, usually

UPCRC Illinois 2009 Summer School on Multicore Programming

Two Types of "Efficiency"

- Efficiency as "performance per core"
  - Are you capturing the peak efficiency?
- Efficiency as "performance per unit of energy"
  - Is the computation energy efficient
- Examples:
  - Use all cores half the time, one core half the time
    - Assuming unused cores "idle", power efficient
  - Uses all cores all the time, but overheads reduce performance
    - Inefficient in both efficiency metrics

Amdahl's Law

Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors. Two equivalent expressions for Amdahl's Law are given below:

$t_N = (f_p/N + f_s)t_1$ Effect of multiple processors on run time

$S = 1/(f_s + f_p/N)$ Effect of multiple processors on speedup

Where:

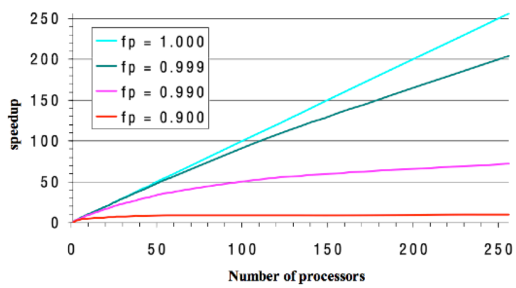
f_s = serial fraction of code

f_p = parallel fraction of code = 1 - f_s

N = number of processors

Illustration of Amdahl's Law

It takes only a small fraction of serial content in a code to degrade the parallel performance. It is essential to determine the scaling behavior of your code before doing production runs using large numbers of processors



Overhead of Parallelism

- **Given enough parallel work, this is the biggest barrier to getting desired speedup**
- **Parallelism overheads include:**
  - **—cost of starting a thread or process**
  - **—cost of communicating shared data**
  - **—cost of synchronizing**
  - **—extra (redundant) computation**
- **Each of these can be in the range of milliseconds (=millions of flops) on some systems**
- **Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (I.e. large granularity), but not so large that there is not enough parallel work**

Load Imbalance

- Load imbalance is the time that some processors in the system are idle due to
  - —insufficient parallelism (during that phase)
  - —unequal size tasks
- Examples of the latter
  - —adapting to "interesting parts of a domain"
  - —tree-structured computations
  - —fundamentally unstructured problems
- Algorithm needs to balance load

Even Parallelism Has Limits

- 1 core. 2 cores. 4 cores. 8 cores. 1024 cores!
  - This is how some multicore researchers count
- Power scaling limitations: "utilization wall"
  - Energy per transistor is decreasing...
  - But, not as rapidly as the number of transistors available
  - Will limit the number of transistors in use at one time
- Memory system: "memory wall"
  - Limited cache capacity, memory bandwidth
- Amount of parallelism in applications: "Amdahl's Law"
  - Few algorithms scale up to 1000s of cores
- Our focus: moderate core counts (walk, then run)
  - Even though limited, parallelism is key to increased performance

The Parallelism Revolution

What is Multicore (Parallel) Computing?

- **Parallel computing**: using multiple processors to...
  - More quickly perform a computation, or...
    - Perform a larger computation in the same amount of time
    - **Programmer expresses and/or coordinates the parallelism**
- Examples:
  - Clusters of computers, coordinate with explicit messages — not covered
  - A shared-memory multiprocessor — course focus
    - Called a **"multicore"** when all on the same chip
  - Graphics processing units (GPUs) — some discussion
    - Perform computations in parallel, increasingly programmable
- The parallel execution motivated by **performance**
  - Different from concurrency in a distributed system or network server

Based on a slide by Katherine Yelick

Aside: This Class is About Three Things

- Performance!
- Performance!!
- Performance!!!
- Ok, not really...
  - Also about correctness, "-abilities", etc.
  - But if you think "computers are fast enough"...
    - And "low power enough"...
  - ...this probably isn't the course for you!
- And not performance "in theory"
  - Physics analogy: not "frictionless surfaces" and "no air resistance"
  - Nitty gritty real-world wall-clock performance

A Trend in Computing Last Few Decades

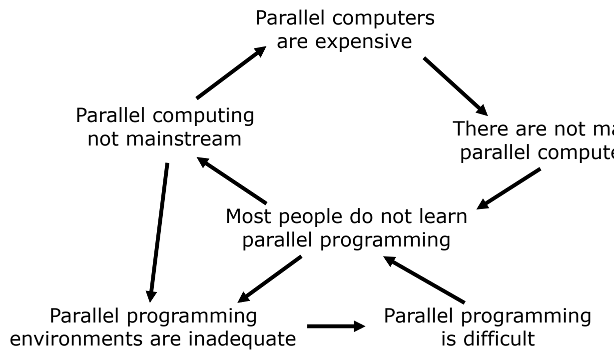
- Old conventional wisdom:
  Trade performance for improved programmer productivity
  - Higher-level languages, interfaces, abstraction layers, frameworks
  - Graphical user interfaces (GUIs)
  - How many hardware instructions to put "hello world" in a window?
  - See: "Spending Moore's Dividend", Jim Larus, CACM 2009
- New conventional wisdom:
  Obtain performance by reducing programming productivity
  - Programmers given additional burden: **writing parallel software**
  - Seems like a really bad idea...
- More conventional wisdom:
  - Parallel programming is intractably difficult

What is Old is New Again

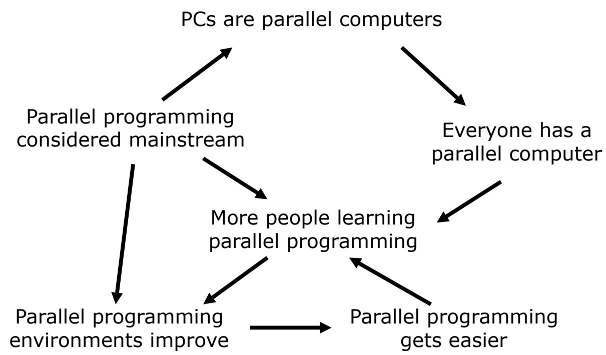
- Parallelism isn't new
  - Commonplace for computational science and engineering
  - To tackle problems too large to solve on any one computer
  - Old-school "supercomputers" were also highly parallel
- Mainstream parallel computing "next big thing" for decades
  - Many companies bet on parallelism and failed
  - Why? One reason: non-parallel computers got faster so quickly
- Ok, so why is parallelism so talked about now?
  - The entire industry has bet on parallelism!
  - Driven to parallelism by technology and architectural realities (next)
    - **Sequential (non-parallel) performance is lagging**
  - Thus, need for parallel programmers & related research

Based on a slide by Katherine Yelick

Old Dynamic of Parallel Computing



New Dynamic of Parallel Computing



Why Explicitly Parallel over Sequential?

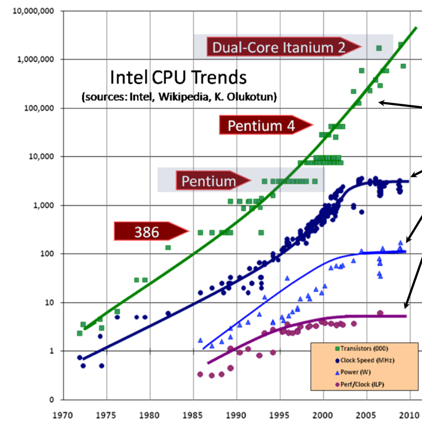
- Today's micro-architectural design realities
  - Pipelining pushed to limits
  - Instruction-level parallelism maxed out
  - Cache misses limit performance
  - Relatively longer wire delays (many cycles to cross the chip)

1. Diminishing returns on single-thread "implicit parallelism"
   - Speedup less than increase in chip area (which is maybe okay)
   - Increasingly, no untapped techniques to accelerate sequential code
2. Power implications
   - Parallelism is power efficient
   - High clock frequency is power inefficient
   - Multiple lower-frequency cores versus single higher-frequency core

Power Implications of Parallelism

- Consider doubling number of cores, same power budget
  - By reducing clock frequency and voltage... but how much?
- First, a few equations (approximate, first order)
  - Frequency ~ Voltage (higher voltage -> transistors switch faster)
  - Dynamic Power ~ Transistors * Frequency * Voltage^2
  - Thus, Dynamic Power ~ Transistors * Frequency^3
- How?
  - Doubling number of cores (transistors) will double the power
  - Reducing frequency & voltage by 20% will cut power in half (0.8^3 is 0.5)... 1.6x the peak performance of original design
- Parallelism has greater performance potential
  - If we can write software for it!

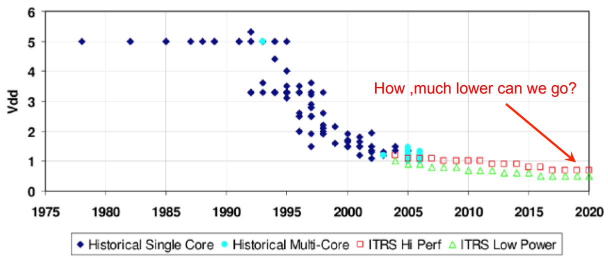
Technology Trend Data



- Performance ~ freq * IPC
- Power ~ Num * Voltage^2 * freq
- # of transistors growing
  - Moore's law
- Clock frequencies flat
- Power budget at limit
- IPC flat
- Solution? Reduce voltage?
  - Yes, but hurts frequency
  - And, see next silde

Source: Herb Sutter's "*The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*"

Voltage Evolution



13

UPCRC Illinois 2009 Summer School on Multicore Programming

Parallel Architectures

Instantiations of Explicit Parallelism

- Multicore
  - More than one processor ("core") on a chip
  - 1990s multi-socket multiprocessor on a chip
  - Provides a "shared memory" abstraction
- Vectors/SIMD
  - Special instructions that operate on multiple data in one instruction
  - Example: four 32-bit adds (pairwise) on 128-bit registers
  - Added by compiler (automatic vectorization or by programmer)
  - 1970s Cray supercomputer on a chip
- Accelerators / Graphics Processing Units (GPUs)
  - 1990s SGI Reality/InfiniteReality Engine on a chip Special-purpose graphics hardware -> general-purpose hardware
    - "Programmable" shaders

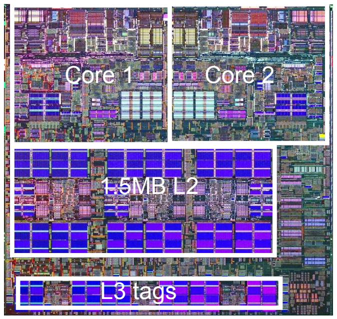
Multicore Everywhere

- Servers
  - Racks of multi-socket multi-core processors
  - Until recently, sweet spot was dual-socket dual-core x86 servers

Name	Year	Sockets	Cores per chip	Threads per core	Total
Sun's Niagara T1	2005	1	8	4	32
Sun's Niagara T2	2007	2	8	8	128
AMD's "Istanbul"	2009	4	6	1	24
Intel's Nehalem-EX	2010	4	8	2	64

- Desktop: Intel's quad-core Core i7
- Game consoles: PS3 (8 cores), XBox 360 (3 cores)
- Laptops: Intel's dual-core Core 2 Duo
- Mobile devices: Atom (x86), Tegra 2 (ARM)

Multicore Examples

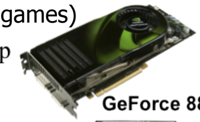


- **Multicore chips**
- **IBM Power5**
  - Two 2+GHz PowerPC cores
  - Shared 1.5 MB L2, L3 tags
- AMD Quad Phenom
  - Four 2+ GHz cores
  - Per-core 512KB L2 cache
  - Shared 2MB L3 cache
- Intel Core 2 Quad
  - Four cores, shared 4 MB L2
  - Two 4MB L2 caches
- Sun Niagara
  - 8 cores, each 4-way threaded
  - Shared 2MB L2, shared FP
  - For servers, not desktop

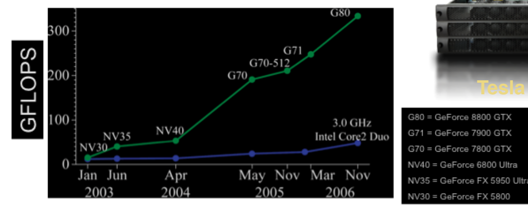
Why multicore? What else would you do with a billion transistors?

Graphics Processing Units (GPU)

- Killer app for parallelism: graphics (3D games)
- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until recently, programmed through graphics API



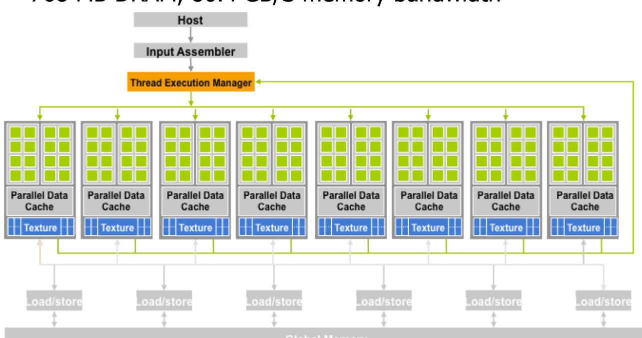
GeForce 8800



  - GPU in every desktop, laptop, mobile device – massive volume and potential impact

GeForce 8800

- 16 highly threaded units, >128 FPU's, 367 GFLOPS 768 MB DRAM, 86.4 GB/S memory bandwidth



Recent Mobile Example: Nvidia's Tegra 2

Mobile example, announced Jan 2010

1080p video playback, 3D touchscreen UI support, and unmatched battery life.

KEY FEATURES

Two next-generation ARM cores, ~500 mWatts

DUAL-CORE ARM® CORTEX -A9 MPCORE™ PROCESSOR
- Symmetric Multi-Processing support for blazing fast web browsing performance, improving load times and rendering of complex pages
- Processing efficiency of Cortex-A9 provides industry leading performance in lowest power envelope

Next Generation NVIDIA Tegra Mobile Web Processor

ULTRA LOW POWER NVIDIA GRAPHICS PROCESSING UNIT (GPU)
- Enhanced NVIDIA graphics technology, enabling full Flash acceleration for an uncompromised HD web browsing experience
- Next generation 3D rendering performance for the most compelling user interfaces and advanced mobile games

Powerful GPU

FULL HIGH DEFINITION MULTIMEDIA
- Up to 1080p video encode/decode and support for HD Web streaming formats, such as YouTube HD
- Complete HW accelerated HD multimedia engine for visually stunning movie playback at lowest possible power

NVIDIA LOW POWER MANAGEMENT ARCHITECTURE
- Effective power management techniques, such as dynamic voltage and frequency scaling, for ultra-efficient power consumption across all use cases
- Low-power design delivers over 140 hours audio and over 16 hours of HD video playback

SPECIFICATIONS

PROCESSOR AND MEMORY SUBSYSTEM
- Dual-core ARM® Cortex-A9 MPCore™ processor, up to 1.0 GHz
- 32-bit LP-DDR2, DDR2

CS 758 Administrivia

CS758: Administrivia

- Instructor: **Prof. David Wood**
- TA: Derek Hower
- Contact email:
  - david@cs.wisc.edu
  - drh5@cs.wisc.edu
- See web for office hours
- Lectures: MWF 1:00-2:15
- WWW: http://www.cs.wisc.edu/~cs758/
  - Keep an eye on: .../schedule.html
- Class e-mail list
  - **If you're not officially registered, see me to get on this list**

CS758: Course Requirements

- Some architecture background
  - Graduate-level (CS/ECE 752) or advanced undergraduate level (CS552)
  - Topics: pipelining, caches, processor performance metrics, etc.
- Substantial programming experience (C/C++/Java)
- Familiarity with C++ programming
  - Least-common denominator language for parallel computing
  - OpenCL/CUDA build upon C
- Instructor's permission

CS758: Course Topics

- Topics
  - Multicore architectures
  - Threads and shared memory
  - Synchronization (barriers and various locks)
  - Task-based runtimes (OpenMP, TBB, Cilk)
  - Data-level parallelism (vectors & SIMD)
  - GPU architectures & programing (OpenCL)
  - Research: serialization sets, transactional memory, multicore map-reduce, etc.
- Non-Topics
  - Cluster computing
  - Message-passing parallelism (MPI)
  - Cloud computing
  - Distributed systems

CS758: Course Outcomes

- Outcomes
  - Knowledge of general concepts in multicore programming
  - Understand performance implications of parallel architectures
    - Difficult to abstract the performance of multicore hardware
  - Hands-on experience writing and tuning multicore software
    - Exposure to several multicore programming approaches
  - Significant parallel programming project
  - Preparation for multicore programming/architectures research
- Non-Outcomes
  - Being an "expert" in any one programming model/language
  - Learning specific tools or development environments in depth

CS758: Warning

- No standard format for such a class
  - No established textbook, canonical assignments, etc.
  - Simultaneously a "new" & "old" topic (stale conventional wisdom)
  - We'll rely mostly on primary sources
- Course format will be some combination of:
  - PhD seminar course (readings, reviews, discussion)
  - Graduate-level project course (programing assignments, project)
  - Lecture course (lectures, exam)
- Plan: primarily in-class discussions, lectures to fill in gaps

CS758: Coursework

- Class participation (10%)
  - Expected to complete assigned readings before class
  - **And actively participate in discussions**
- Paper reviews (10%)
  - Short response to papers we'll discuss in class
  - Turn 9am morning of class period (**must be present**)
  - Grading: Excellent (10 pts), Satisfactory (7 pts), unsatisfactory (3 pts)
- Programming assignments (25%)
  - Various hands-on programming assignments
- Exam (20%) - one exam, not during finals week
  - After spring break, in class, exact date TBD
- In-class paper presentation (5%)
  - Give a ~20 minute presentation of a paper to the class
- Course project (30%)

CS758: Course Project

- Parallel programming project
  - Groups of two (three or one, with advanced approval of instructor)
  - Proposal, presentation (class conference), final report (conference format)
  - More logistics later
- Create substantial parallel program
  - Analyze and tune its parallel performance
  - Focus on parallel aspect (easy or existing serial solution)
- Case study on comparing/contrasting programming models
  - Simpler parallel program...
  - But experimentally compare the performance, discuss ease of development
- Mini-research project
  - Examine modest extension to paper studied in class (default)
    - Runtime system modification, advanced synchronization, etc.
  - Your own idea (more ambitions!)

Academic Honesty

- You're encouraged to discuss the course content and assignments...
- But, anything with your name on it...
  ...must be **YOUR OWN** work
- Possible penalties for dishonesty
  - Zero on assignment (minimum)
  - Fail course
  - Note on permanent record
  - Suspension
  - Expulsion
- See UW Student Code of Conduct

Notes

"Parallelism" versus "Concurrency"

- "Threads and locks"
  - Common idiom for two often-confused, but distinct domains
- "Concurrent" software
  - A property of the "environment" of the program
  - Threads for handling input/output
    - Network packet arrival
    - User interacts with GUI (graphical user interface)
    - Hardware interrupt in O.S.
  - Makes sense even in a single-core system
- "Parallel" software
  - Goal: faster runtime
  - Only for multiple hardware cores or cluster computing
- Some programs are both

For Next Time...

- Read the two papers
  - "The Free Lunch is Over", Herb Sutter
  - "Software and the Concurrency Revolution", Sutter and Larus
- Paper review due at beginning of class (hardcopy)
  - See web page for specifics (posted soon)
- Note: No class Monday (MLK)
- See me now if:
  - You're not officially registered, but want to
  - Any other questions about prerequisites or the course
- Want to receive course emails?
  - Send me email