Lecture 1: Introduction to CS 5220

David Bindel

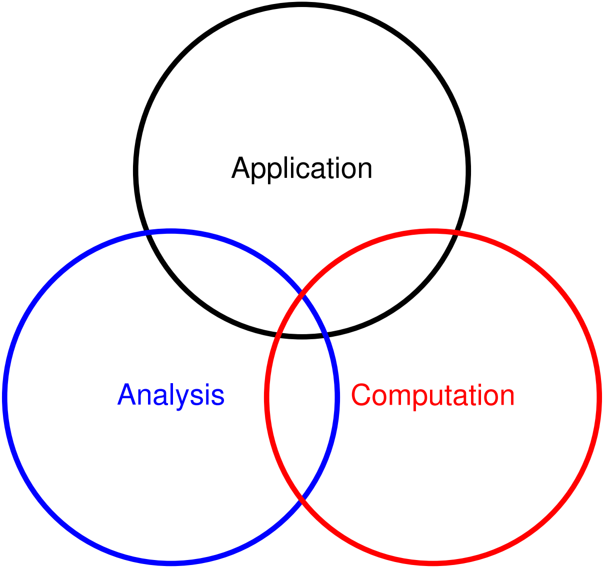
24 Aug 2011

CS 5220: Applications of Parallel Computers

`http://www.cs.cornell.edu/~bindel/class/cs5220-f11/`
`http://www.piazza.com/cornell/cs5220`

Time:	TR 8:40–9:55
Location:	110 Hollister
Instructor:	David Bindel (`bindel@cs`)
Office:	5137 Upson Hall
Office hours:	M 4–5, Th 10–11, or by appt.

The Computational Science & Engineering Picture



Applications Everywhere!

These tools are used in more places than you might think:

- Climate modeling
- CAD tools (computers, buildings, airplanes, ...)
- Control systems
- Computational biology
- Computational finance
- Machine learning and statistical models
- Game physics and movie special effects
- Medical imaging
- Information retrieval
- ...

Parallel computing shows up in all of these.

Why Parallel Computing?

1. Scientific computing went parallel long ago
   - Want an answer that is right enough, fast enough
   - Either of those might imply a lot of work!
   - ... and we like to ask for more as machines get bigger
   - ... and we have a lot of data, too
2. Now everyone else is going the same way!
   - Moore's law continues (double density every 18 months)
   - But clock speeds stopped increasing around 2005
     - ... otherwise we'd have power densities associated with the sun's surface on our chips!
     - But no more free speed-up with new hardware generations
   - Maybe double number of cores every two years instead?
   - Consequence: We all become parallel programmers?

Lecture Plan

Roughly three parts:

1. **Basics:** architecture, parallel concepts, locality and parallelism in scientific codes
2. **Technology:** OpenMP, MPI, CUDA/OpenCL, UPC, cloud systems, profiling tools, computational steering
3. **Patterns:** Monte Carlo, dense and sparse linear algebra and PDEs, graph partitioning and load balancing, fast multipole, fast transforms

Goals for the Class

You will learn:

- Basic parallel concepts and vocabulary
- Several parallel platforms (HW and SW)
- Performance analysis and tuning
- Some nuts-and-bolts of parallel programming
- Patterns for parallel computing in computational science

You might also learn things about

- C and UNIX programming
- Software carpentry
- Creative debugging (or swearing at broken code)

Workload

CSE usually requires teams with different backgrounds.

- Most class work will be done in small groups (1–3)
- Three assigned programming projects (20% each)
- One final project (30%)
  - Should involve some performance analysis
  - Best projects are attached to interesting applications
  - Final presentation in lieu of final exam

Prerequisites

You should have:

- Basic familiarity with C programming
  - See CS 4411: Intro to C and practice questions.
  - Might want Kernighan-Ritchie if you don't have it already
- Basic numerical methods
  - See CS 3220 from last semester.
  - Shouldn't panic when I write an ODE or a matrix!
- Some engineering or physics is nice, but not required

How Fast Can We Go?

Speed records for the Linpack benchmark:

`http://www.top500.org`

Speed measured in flop/s (floating point ops / second):

- Giga (10^9) – a single core
- Tera (10^{12}) – a big machine
- Peta (10^{15}) – current top 10 machines (5 in US)
- Exa (10^{18}) – favorite of funding agencies

Current record-holder: Japan's K computer (8.2 Petaflop/s).

Peak Speed of the K Computer

$(2 \times 10^9$ cycles / second$) \times$
(8 flops / cycle / core) =
  16 GFlop/s / node

(16 GFlop/s / node) $\times$ (8 cores / node) =
  128 GFlop/s / node

(128 GFlop/s / node) $\times$
(68544 nodes) =
  8.77 GFlop/s

Linpack performance is about 93% of peak.

Current US Record-Holder

DOE Jaguar at ORNL

- Cray XT5-HE with
  - 6-core AMD x86_64 Opteron 2.6 GHz (10.4 GFlop/s/core)
  - 224162 cores
  - Custom interconnect
- 2.33 Petaflop/s theoretical peak
- 1.76 Petaflop/s Linpack benchmark (75% peak)
- 0.7 Petaflop/s in a blood flow simulation (30% peak)
  (Highly tuned – this code won the 2010 Gordon Bell Prize)
- Performance on a more standard code?
  - 10% is probably very good!

Parallel Performance in Practice

So how fast can I make my computation?

- Peak $>$ Linpack $>$ Gordon Bell $>$ Typical
- Measuring performance of real applications is hard
  - Typically a few bottlenecks slow things down
  - And figuring out why they slow down can be tricky!
- And we *really* care about time-to-solution
  - Sophisticated methods get answer in fewer flops
  - ... but may look bad in benchmarks (lower flop rates!)

See also David Bailey's comments:

- Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers (1991)
- Twelve Ways to Fool the Masses: Fast Forward to 2011 (2011)

Quantifying Parallel Performance

- Starting point: good *serial* performance
- Strong scaling: compare parallel to serial time on the same problem instance as a function of number of processors (p)

$$\text{Speedup} = \frac{\text{Serial time}}{\text{Parallel time}}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{p}$$

- Ideally, speedup = p. Usually, speedup $< p$.
- Barriers to perfect speedup
  - Serial work (Amdahl's law)
  - Parallel overheads (communication, synchronization)

Amdahl's Law

Parallel scaling study where some serial code remains:

$$\begin{aligned} p &= \text{number of processors} \\ s &= \text{fraction of work that is serial} \\ t_s &= \text{serial time} \\ t_p &= \text{parallel time} \geq st_s + (1-s)t_s/p \end{aligned}$$

Amdahl's law:

$$\text{Speedup} = \frac{t_s}{t_p} = \frac{1}{s + (1-s)/p} > \frac{1}{s}$$

So 1% serial work $\implies$ max speedup < 100×, regardless of p.

A Little Experiment

Let's try a simple parallel attendance count:

- **Parallel computation:** Rightmost person in each row counts number in row.
- **Synchronization:** Raise your hand when you have a count
- **Communication:** When all hands are raised, each row representative adds their count to a tally and says the sum (going front to back).

(Somebody please time this.)

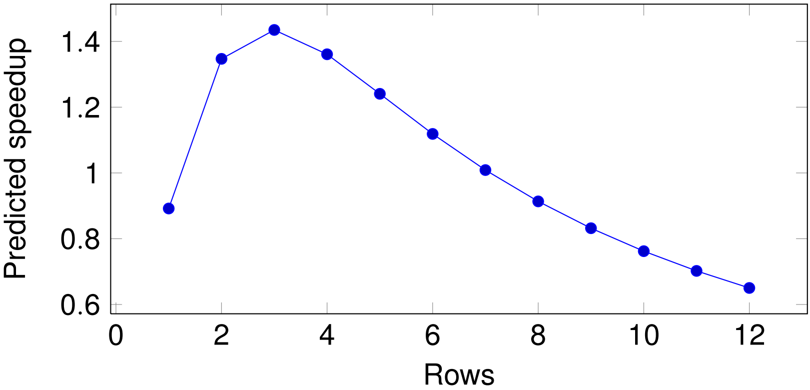
A Toy Analysis

Parameters:

$$\begin{aligned} n &= \text{number of students} \\ r &= \text{number of rows} \\ t_c &= \text{time to count one student} \\ t_t &= \text{time to say tally} \\ t_s &\approx n t_c \\ t_p &\approx n t_c / r + r t_t \end{aligned}$$

How much could I possibly speed up?

Modeling Speedup



(Parameters: $n = 55$, $t_c = 0.3$, $t_t = 2$.)

Modeling Speedup

The bound

$$\text{speedup} < \frac{1}{2}\sqrt{\frac{nt_c}{t_t}}$$

is usually tight (for previous slide: $1.435 < 1.436$).

Poor speed-up occurs because:

- The problem size n is small
- The communication cost is relatively large
- The serial computation cost is relatively large

Some of the usual suspects for parallel performance problems!

Things would look better if I allowed both n and r to grow — that would be a *weak* scaling study.

Summary: Thinking about Parallel Performance

Today:

- We're approaching machines with peak *exaflop* rates
- But codes rarely get peak performance
- Better comparison: tuned serial performance
- Common measures: *speedup* and *efficiency*
- Strong scaling: study speedup with increasing p
- Weak scaling: increase both p and n
- Serial overheads and communication costs kill speedup
- Simple analytical models help us understand scaling

Next time: Computer architecture and serial performance.

And in case you arrived late

`http://www.cs.cornell.edu/~bindel/class/cs5220-f11/`

`http://www.piazza.com/cornell/cs5220`