Speed-Up

Mark Greenstreet

CpSc 418 – Jan. 30, 2017

Outline:

But first, USRA

Summer Undergraduate Research Opportunities

- Natural Sciences and Engineering Research Council (NSERC) Undergraduate Student Research Awards (USRAs)
  - Same process to apply for Science Undergraduate Research Experience (SURE) and Work Learn International Undergraduate Research Awards
- See what academic research really looks like
- Many research areas: ...
  - Google “ubc cs usra” for full list of projects seeking students
- I have several project proposals:
  - Collaborative control of smart wheelchairs for older adults
  - Numerical software for demonstrating correctness of robots and cyber-physical systems
- 16 weeks, flexible schedule
- You get paid!
- Email potential sponsor ASAP (full applications due by Feb 10)

Objectives

- Understand key measures of performance
  - Time: latency vs. throughput
  - Time: wall-clock vs. operation count
  - Speed-up: slide 5
- Understand common observations about parallel performance
  - Amdahl's law: limitations on parallel performance (and how to evade them)
  - The law of modest returns: high complexity problems are bad, and worse on a parallel machine.
  - Superlinear speed-up: more CPUs $\Rightarrow$ more, fast memory – and sometimes you win.
  - Embarrassingly parallel problems: sometimes you win, without even trying.

Measuring Performance

- The main motivation for parallel programming is performance
  - Time: make a program run faster.
  - Space: allow a program to run with more memory.
- To make a program run faster, we need to know how fast it is running.
- There are many possible measures:
  - Latency: time from starting a task until it completes.
  - Throughput: the rate at which tasks are completed.
  - Key observation:

$$
\begin{aligned}
\textit{throughput} &= \frac{1}{\textit{latency}}, \quad \text{sequential programming} \\
\textit{throughput} &\geq \frac{1}{\textit{latency}}, \quad \text{parallel programming}
\end{aligned}
$$

Speed-Up

- Simple definition:

$$speed\_up = \frac{\mathrm{time(sequential\_execution)}}{\mathrm{time(parallel\_execution)}}$$

- We can also describe speed-up as how many percent faster:

$$\%faster = (speed\_up - 1) * 100\%$$

- But beware of the spin:
  - Is “time” latency or throughput?
  - How big is the problem?
  - What is the sequential version:
    - The parallel code run on one processor?
    - The fastest possible sequential implementation?
    - Something else?
- More practically, how do we measure time?

Speed-Up – Example

- Let's say that count 3s of a million items takes 10ms on a single processor.
- If I run count 3s with four processes on a four CPU machine, and it takes 3.2ms, what is the speed-up?
- If I run count 3s with 16 processes on a four CPU machine, and it takes 1.8ms, what is the speed-up?
- If I run count 3s with 128 processes on a 32 CPU machine, and it takes 0.28ms, what is the speed-up?

Time complexity

- What is the time complexity of sorting?
  - What are you counting?
  - Why do you care?
- What is the time complexity of matrix multiplication?
  - What are you counting?
  - Why do you care?

Big-O and Wall-Clock Time

- In our algorithms classes, we count "operations" because we have some belief that they have something to do with how long the actual program will take to execute.
  - Or maybe not. Some would argue that we count "operations" because it allows us to use nifty techniques from discrete math.
  - I'll take the position that the discrete math is nifty because it tells us something useful about what our software will do.
- In our architecture classes, we got the formula:

$$\text{time} \quad = \quad \frac{(\#\text{inst. executed}) * (\text{cycles}/\text{instruction})}{\text{clock frequency}}$$

- The approach in algorithms class of counting comparisons or multiplications, etc., is based on the idea that everything else is done in proportion to these operations.
- BUT, in parallel programming, we can find that a communication between processes can take 1000 times longer than a comparison or multiplication.
  - This may not matter if you're willing to ignore "constant factors."
  - In practice, factors of 1000 are too big to ignore.

Amdahl's Law

- Given a sequential program where
  - fraction s of the execution time is inherently sequential.
  - fraction $1 - s$ of the execution time benefits perfectly from speed-up.
- The run-time on P processors is:

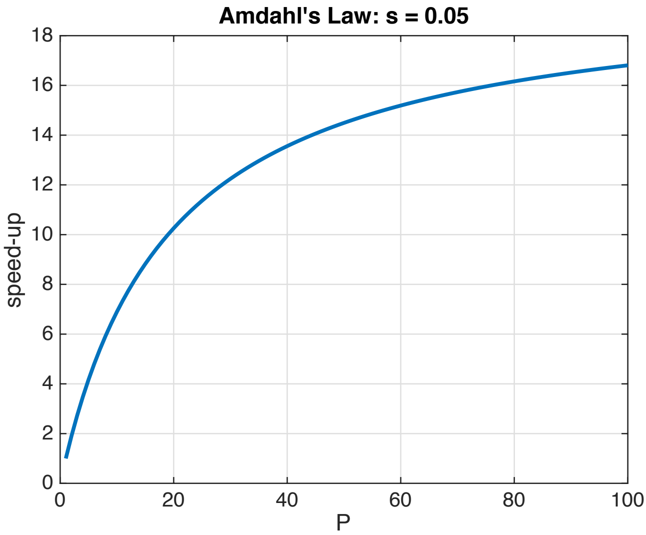
$$T_{parallel} = T_{sequential} * (s + \frac{1-s}{P})$$

- Consequences:
  - Define

$$speed\_up = \frac{T_{sequential}}{T_{parallel}}$$

  - Speed-up on P processors is at most $\frac{1}{s}$.
  - Gene Amdahl argued in 1967 that this limit means that parallel computers are only useful for a few special applications where s is very small.

Amdahl's Law



Amdahl's Law, 49 years later

Amdahl's law is not a physical law.

- Amdahl's law is mathematical theorem:
  - If $T_{parallel}$ is $\left(s + \frac{1-s}{P}\right) T_{sequential}$
  - and $speed\_up = T_{sequential} / T_{parallel}$,
  - then for $0 < s \leq 1$, $speed\_up \leq \frac{1}{s}$.
- Amdahl's law is also an economic law:
  - Amdahl's law was formulated when CPUs were expensive.
  - Today, CPUs are cheap
    - The cost of fabricating eight cores on a die is very little more that the cost of fabricating one.
    - Computer cost is dominated by the rest of the system: memory, disk, network, monitor, ...
- Amdahl's law assumes a fixed problem size.

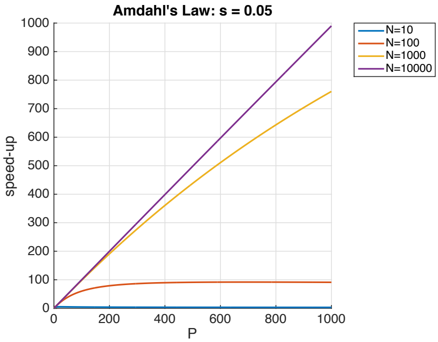
Amdahl's Law, 49 years later

- Amdahl's law is an economic law, not a physical law.
  - Amdahl's law was formulated when CPUs were expensive.
  - Today, CPUs are cheap (see previous slide)
- Amdahl's law assumes a fixed problem size
  - Many computations have s (sequential fraction) that decreases as N (problem size) increases.
  - Having lots of cheap CPUs available will
    - ★ Change our ideas of what computations are easy and which are hard.
    - ★ Determine what the "killer-apps" will be in the next ten years.
      - Ten years from now, people will just take it for granted that most new computer applications will be parallel.
  - Examples: see next slide

Amdahl's Law, 49 years later

- Amdahl's law is an economic law, not a physical law.
- Amdahl's law assumes a fixed problem size
  - Ten years from now, people will just take it for granted that most new computer applications will be parallel.
  - Examples:
    - Managing/searching/mining massive data sets.
    - Scientific computation.
      - Note that most of the computation for animation and rendering resembles scientific computation. Computer games benefit tremendously from parallelism.
      - Likewise for multimedia computing.

Amdahl's Law, one more try



- We can have problems where the parallel work grows faster than the sequential part.
- Example: parallel work grows as $N^{3/2}$ and the sequential part grows as $\log P$.

The Law of Modest Returns

More bad news. ☹

- Let's say we have an algorithm with a sequential run-time $T = (12\text{ns})N^4$.
  - If we're willing to wait for one hour for it to run, what's the largest value of N we can use?
  - If we have 10000 machines, and perfect speed-up (i.e. $speed\_up = 10000$), now what is the largest value of N we can use?
  - What if the run-time is $(5ns)1.2^N$?
- The law of modest returns
  - Parallelism offers modest returns, unless the problem is of fairly low complexity.
  - Sometimes, modest returns are good enough: weather forecasting, climate models.
  - Sometimes, problems have huge N and low complexity: data mining, graphics, machine learning.

Super-Linear Speed-up

Sometimes, $speed\_up > P$. ☺

- How does this happen?
  - Impossibility “proof”: just simulate the P parallel processors with one processor, time-sharing P ways.
- Memory: a common explanation
  - P machines have more main memory (DRAM)
  - and more cache memory and registers (total)
  - and more I/O bandwidth, ...
- Multi-threading: another common explanation
  - The sequential algorithm underutilizes the parallel capabilities of the CPU.
  - A parallel algorithm can make better use.
- Algorithmic advantages: once in a while, you win!
  - Simulation as described above has overhead.
  - If the problem is naturally parallel, the parallel version can be more efficient.
- BUT: be very skeptical of super-linear claims, especially if $speed\_up \gg P$.

Embarrassingly Parallel Problems

Problems that can be solved by a large number of processors with very little communication or coordination.

- Rendering images for computer-animation: each frame is independent of all the others.
- Brute-force searches for cryptography.
- Analyzing large collections of images: astronomy surveys, facial recognition.
- Monte-Carlo simulations: same model, run with different random values.
- **Don't be ashamed if your code is embarrassingly parallel:**
  - Embarrassingly parallel problems are great: you can get excellent performance without heroic efforts.
  - The only thing to be embarrassed about is if you don't take advantage of easy parallelism when it's available.

Lecture Summary

Parallel Performance

- Speed-up: slide 5
- Limits
  - Amdahl's Law, slide 9.
  - Modest gains, slide 15.
- Sometimes, we win
  - Super-linear speedup, slide 16.
  - Embarrassingly Parallel Problems, slide 17.

Preview

February 1: Parallel Performance: Overheads	
Homework:	HW 2 due (11:59pm).
February 3: Parallel Performance: Models	
Mini Assignments	Mini 4 due (10am)
February 6: Parallel Performance: Wrap Up	
February 8: Parallel Sorting – The Zero-One Principle	
Homework (Feb. 15):	HW 3 earlybird (11:59pm), HW 4 goes out.
February 10: Bitonic Sorting (part 1)	
February 15: Family Day – no class	
February 13: Bitonic Sorting (part 2)	
Homework (Feb. 15):	HW 3 earlybird (11:59pm), HW 4 goes out.
February 17: Map-Reduce	
Homework:	HW 3 due (11:59pm).
February 27: TBD	
March 1: Midterm	

- Reading from "Programming Massively Parallel Computers" (D.B. Kirk & W.-M. Hwu) start right after the midterm. Make sure you have a copy.
- You can use either the 2nd or 3rd edition.

Review Questions

- What is speed-up? Give an intuitive, English answer and a mathematical formula.
- Why can it be difficult to determine the sequential time for a program when measuring speed-up?
- What is Amdahl's law? Give a mathematical formula. Why is Amdahl's law a concern when developing parallel applications? Why in many cases is it not a show-stopper?
- Is parallelism an effective solution to problems with high big-O complexity? Why or why not?
- What is super-linear speed-up? Describe two causes.
- What is an embarrassingly parallel problem. Give an example.