ECE408/CS483/CSE408 Spring 2025

Applied Parallel Programming

Lecture 27: Generalizing Parallelism and Course Retrospective

Course Reminders

- Labs
  - All are done, all graded
- Project
  - PM3 is due **this Friday**
  - Final competition is due next Friday
- Midterm Exam 2
  - Tuesday, May 6th 7:00pm-10:00pm US Central time
  - In-person, similar to MT1
  - HKN review session is this Saturday at 3pm
  - **See Canvas for details**

Objective

- To learn terminology and concepts from the broader high-performance computing community
- To generalize some of the techniques illustrated in class for use with future codes
- To see the impact parallelism is making on the world around us

ECE 408 Retrospective: What did we do this semester?

- Elementary Computational Patterns
  - Matrix Multiplication, Convolution, Reduction, Scan, Histogram, Sparse Representations
- Parallel Optimization
  - Threading, Memory Management, Coalescing, Thread Divergence, Data Privatization, Profiling
- Programming Systems
  - CUDA, (OpenACC, OpenCL, Hip, OpenMP,…)

Speedup Measures the Success of Parallelization

- Let's start by defining **parallel speedup** (usually just called speedup).
- Let's say that
  - when I run my program in parallel
  - it finishes **X** times faster
  - than when I run it sequentially.
- Specifically,
  - **X = T(sequential) / T(parallel)**, and
  - **X is the speedup** of my parallel code.
- Note that speedup assumes a fixed problem size.

Speedup Depends on the Best Sequential Code

- We have T(sequential) / T(parallel).
- **But how do we find T(sequential)?**

- T(sequential) **should measure** the
  - **best algorithm** for a sequential machine (may/may not be the algorithm parallelized),
  - **optimized** for a sequential machine, with
  - **no parallelism support** remnants (no parallel overhead).

Find (Don’t Write) a Competitive Baseline

- **Sequential code is** what we in Engineering call
  - the **baseline design**,
  - the alternative against which
  - we demonstrate improvements.
- As Prof. Hwu once pointed out,
  - **no one will believe** that **you** worked hard
  - to **optimize your baseline**…
  - even if you did!
- If possible, **compare someone else’s best work**.

Efficiency Measures Effective Use of Resources

- Next is **parallel efficiency** (or just efficiency).
- Efficiency measures how well a code uses parallel resources.
- When executing **on P processors**,
  - **efficiency = speedup on P processors / P**.

Efficiency is Often Below 1, But Should Not be Tiny

What value should efficiency have?

- According to those paying for the machines, 1.
- According to most real applications,
  - **something non-negligible, near 1**
  - but not 1,
  - as other bottlenecks come into play.

Efficiency is Rarely Above 1

Can efficiency be >1?

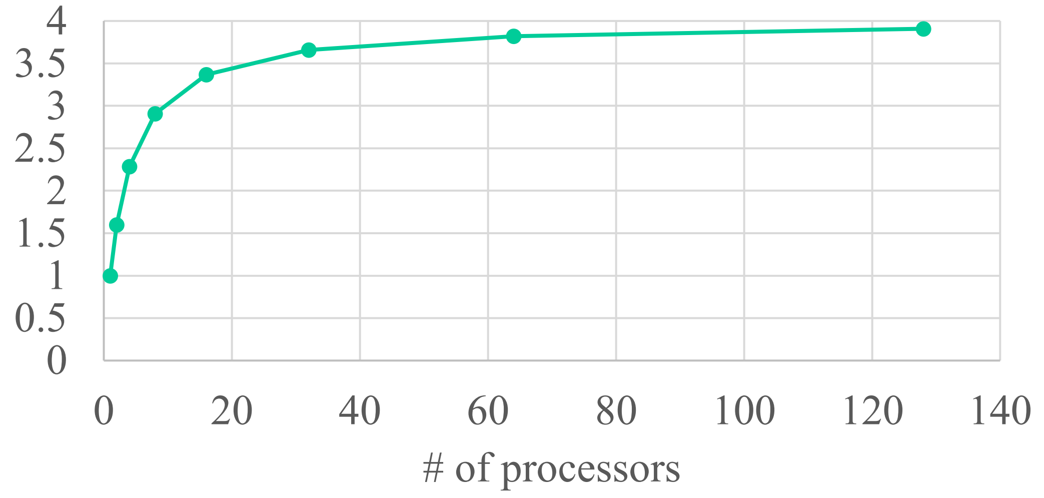
- Rarely—called **superlinear speedup**.
- possible causes:
  - certain types of extra resources (such as caches)
  - luck (parallel search happens to find an answer more quickly).

Scalability Measures Effect of Parallel Overheads

- Next, **scalability**:
  - **for how many processors is**
  - **speedup linear**, or is efficiency flat?

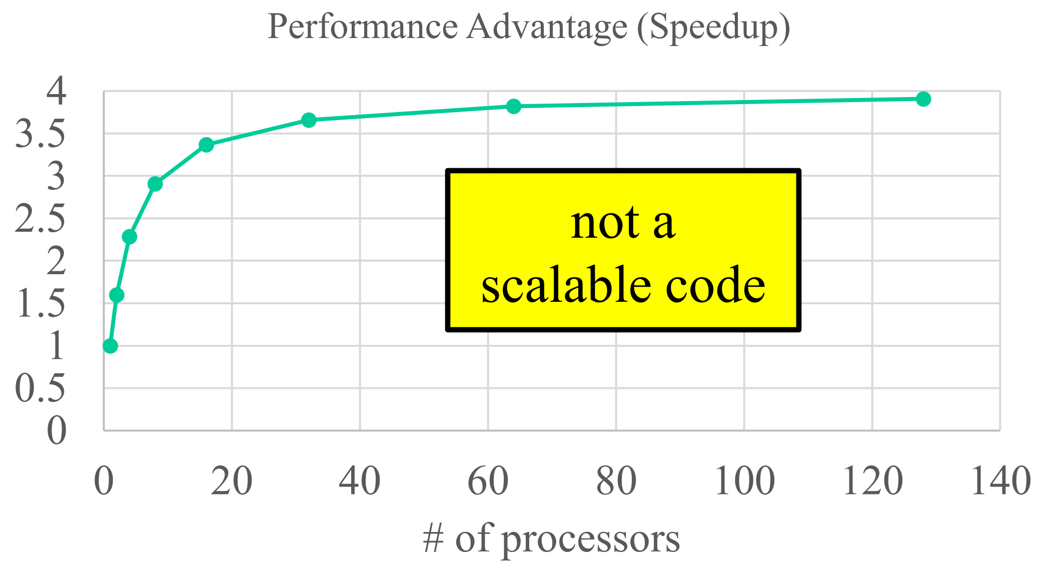
- At some **P**, with fixed problem size, speedup will flatten out.

Performance Advantage (Speedup)



Good Scalability Requires Minimal Parallel Overhead

- For larger values of **P**, speedup starts to drop (unless one leaves processors idle).
- **Good scalability** means
  - **no falloff** on your machine
  - **for maximum** measurable **value of P**.



Efficiency Not So Meaningful When Cores Vary Widely

- **But what is P for a single GPU?**
    - 1?
    - Number of SMs?
    - Number of PEs (total)?
- We can still measure speedup,
    - but for a single GPU,
    - we **estimate efficiency**
    - **by comparing** resource **use**
    - **with** the GPU’s **peak values**.

(As we’ve done in our class already.)

Speedup Measures Improvement for an Input Set

- Again, **speedup assumes** a **fixed problem size**.
  - For many applications, that's reasonable.
  - Users care about their input sets, not about hypothetical inputs.
- But that's **not always the best assumption**.

For Other Situations, We Need Different Metrics

- **Sometimes we care about throughput**:
  - frames per second for video / game quality,
  - transactions per second for databases, or
  - user operations per second for datacenters.
- And **sometimes input size**
  - **is limited** by memory
  - or by feasible runtime,
  - as in many supercomputing applications.

Scaling Problem Size with P Good for Science Apps

- Other **variants of speedup on P processors**:*
- **scaled speedup**:
  - problem size is linear in **P**
  - (good scaled speedup is **1**)
- **memory-constrained speedup**:
  - biggest problem that fits in memory (which scales with **P**)
  - only works for **O(N)** algorithms

*J. P. Singh, J. L. Hennessy, A. Gupta, "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *IEEE Computer*, 26(7):42-50, July 1993.

Problem Size Sometimes Chosen Through Practical Means

- Other **variants of speedup on P processors**:*
- **time-constrained speedup**:
  - biggest problem that finishes by the time I return from lunch
  - sometimes reasonable…
  - …but we could wait overnight for a grand challenge application?

*J. P. Singh, J. L. Hennessy, A. Gupta, "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *IEEE Computer*, 26(7):42-50, July 1993.

Parallel Grain Size is the Work Done per Thread

- **Parallel grain size** is work per thread (task).
  - Remember discussing what to parallelize?
  - Output elements, input elements, …
- **Each source** of parallelism has **a natural grain size**:
  - loop body,
  - objects in a container,
  - rows/columns/blocks/elements in a matrix,
  - graph nodes/connected components.

Consider Different Sources of Parallelism

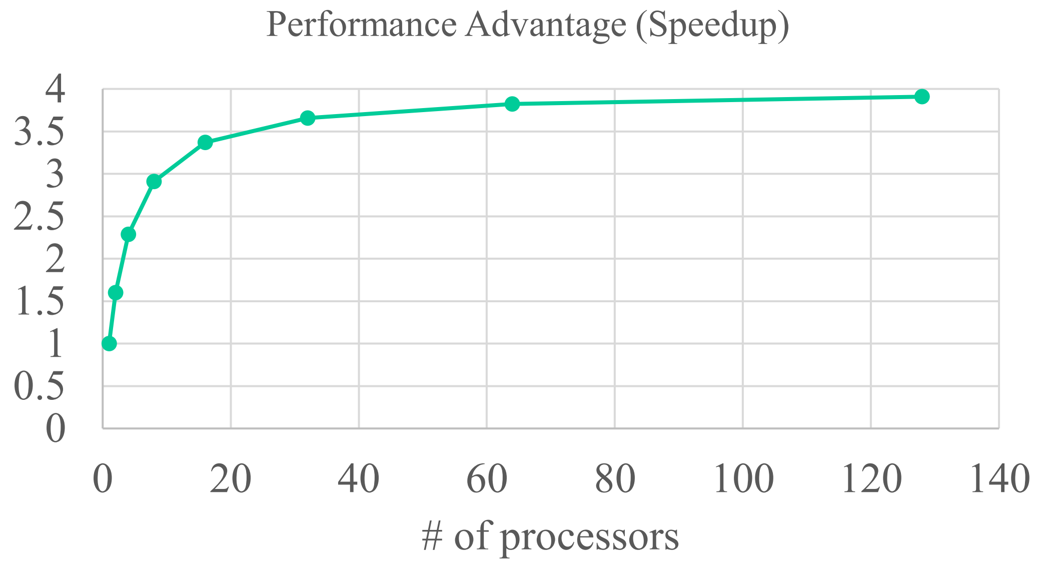
- Some sources exhibit higher work variance (and branch divergence) than others
  - conditionals/inner loops in loop body
  - complex per-object methods
  - rows in upper/lower diagonal matrix
  - matrix elements usually roughly constant
  - degree of nodes, size of connected components.
    - **Be sure to consider the alternatives!**

Amdahl's Law Helps Set Expectations

- **Amdahl's Law** says
  - **speedup** is **bounded** above
  - **by 1 / (sequential fraction)**.

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

- For example, if you parallelize code that takes 75% of the time, you can't get more than 4 × speedup.



Evaluate Your Work Intelligently and Meaningfully

- But, again, for fixed input.
- There are other 'laws' as well that view the problem differently.
- **So what matters most?**
  - Some apps today are missing/simplified due to resource limits.
  - Some apps become possible/more useful with bigger problem sizes.

Fit evaluation of utility to your app,
not your app to an evaluation metric.

A Few Useful Concepts

- Now, I’d like to go over a few useful ideas from high-performance computing.
- Most you’ve seen before, so I’ll tie them into what you’ve seen and done in our class.

Bulk Synchronous Execution Dominates Fast Computing

- The **bulk synchronous** style
  - dominates HPC and CUDA applications.
  - **Barriers separate** temporal **regions of code**
    - usually O(100) lines long
    - interleaving / data **sharing occurs only within regions** (called phases).
- Why?
  - Simpler to debug regions than whole programs.
  - (similar to Stroustrup's view of classes' value).
- Bulk synchronous execution **does tend to correlate resource usage**, which is bad.

Necessary/Good Sources of Parallel Overhead

- Good ways to waste time in parallel;
  - push bits around (**communicate**)—a necessary overhead in most parallel codes
  - **do some extra work** (to avoid communicating)
    - for example, do pooling after convolution in a CNN kernel to reduce shared-to-global memory traffic
    - another: do extra adds to reduce the number of barriers, as in a Kogge-Stone scan
  - bicker about priority (**contend for shared resources**)

Bad Sources of Parallel Overhead

- Bad ways to waste time in parallel;
  - twiddle your thumbs (**wait for long-latency events**)
  - **watch others work**
    - example: branch divergence in a GPU
    - example: poor scheduling decisions
  - line up single file (**unnecessary serialization**)
    - example: coarse synchronization, lack of privatization
    - example: temporally correlated accesses to shared hardware resource
    - example: use one CUDA stream

Dynamic Load Balancing Sometimes Needed

- In our class, we have generally
  - assigned fixed work per thread.
  - Usually, this is the simplest approach, but it may lead to load imbalance.
- One common solution—**load balancing**:
  - dynamic mapping of work to threads using
  - one or more queues of work
    - pull chunk of work from a queue, do it, repeat
    - start with bigger chunks, later grab smaller
    - if queue is empty, **steal work** from another.

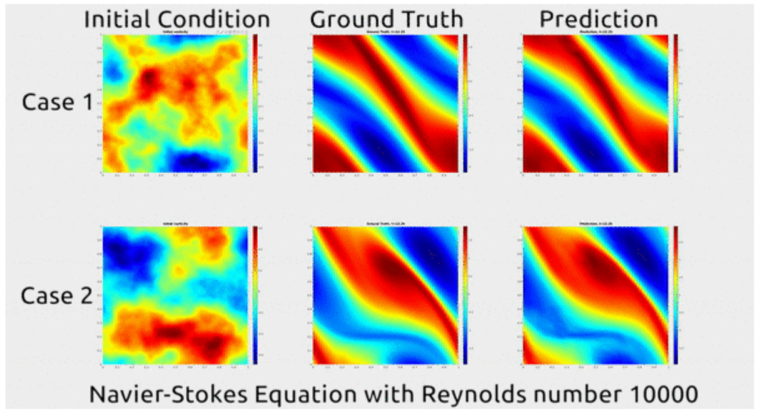
CUDA Scheduling May Need to Become More Expressive

- One last question: kernel/block scheduling.
- Most OS schedulers use **time-sharing**:
  try to be fair to all the running programs.
- But if you have many processors, why pay parallel overhead?
- Use **space-sharing** instead!
- Lots of supercomputers and datacenters do.
- How are thread blocks within CUDA kernels scheduled?

Recent news from the real world (1/3)

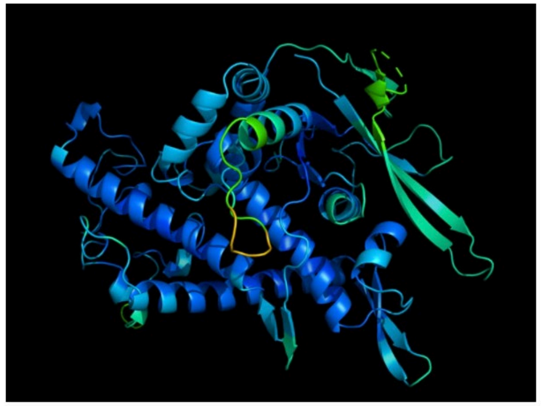
AI has cracked a key mathematical puzzle for understanding our world

Partial differential equations can describe everything from planetary motion to plate tectonics, but they're notoriously hard to solve.

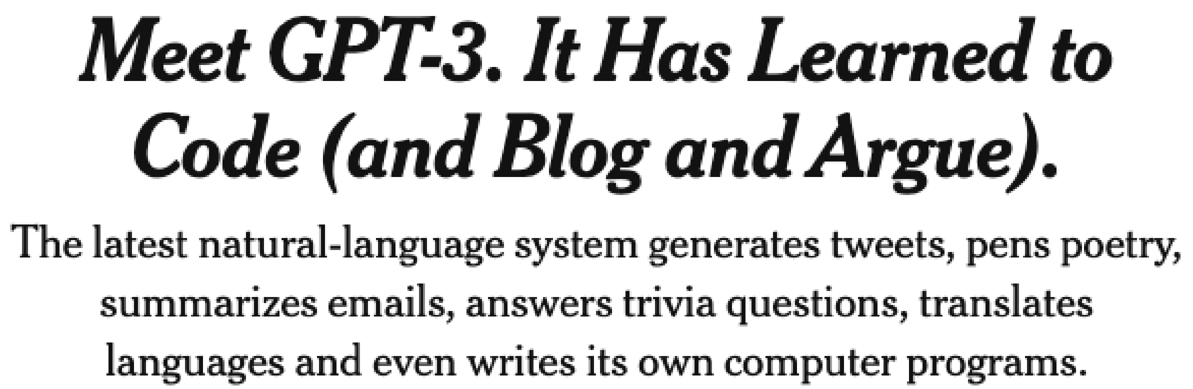


'It will change everything': DeepMind's AI makes gigantic leap in solving protein structures

Google's deep-learning program for determining the 3D shapes of proteins stands to transform biology, say scientists.

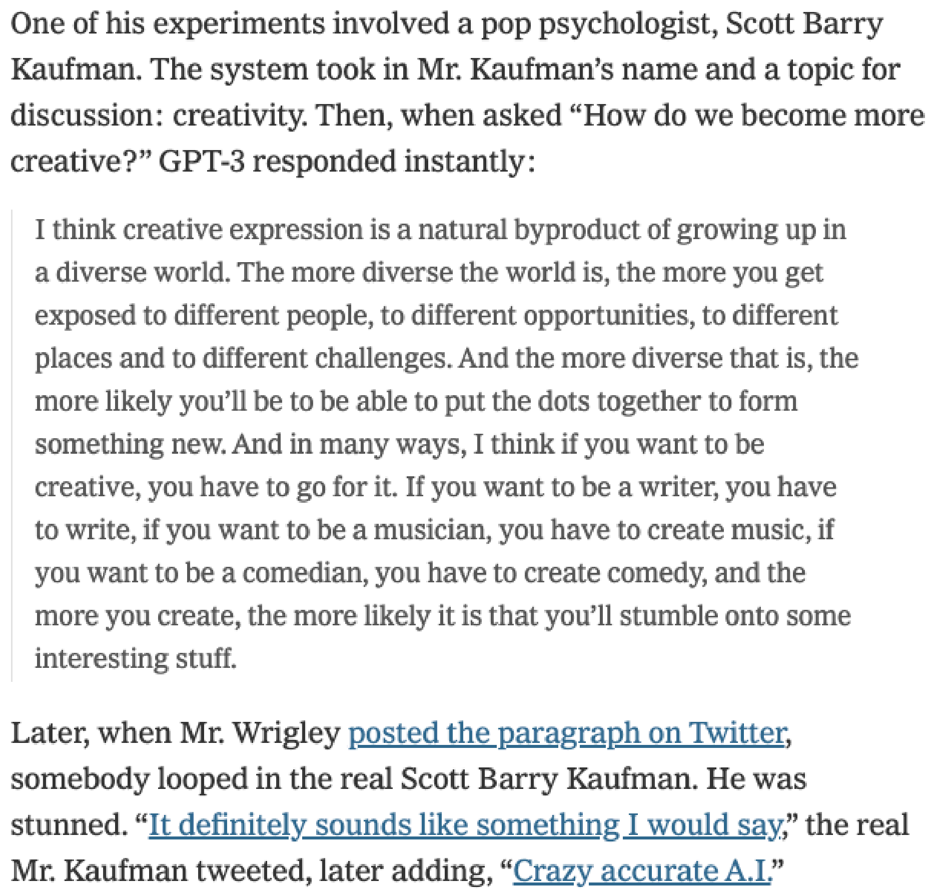

A protein's function is determined by its 3D shape. Credit: DeepMind

Recent news from the real world (2/3)



Meet GPT-3. It Has Learned to Code (and Blog and Argue).

The latest natural-language system generates tweets, pens poetry, summarizes emails, answers trivia questions, translates languages and even writes its own computer programs.



One of his experiments involved a pop psychologist, Scott Barry Kaufman. The system took in Mr. Kaufman's name and a topic for discussion: creativity. Then, when asked "How do we become more creative?" GPT-3 responded instantly:

> I think creative expression is a natural byproduct of growing up in a diverse world. The more diverse the world is, the more you get exposed to different people, to different opportunities, to different places and to different challenges. And the more diverse that is, the more likely you'll be to be able to put the dots together to form something new. And in many ways, I think if you want to be creative, you have to go for it. If you want to be a writer, you have to write, if you want to be a musician, you have to create music, if you want to be a comedian, you have to create comedy, and the more you create, the more likely it is that you'll stumble onto some interesting stuff.

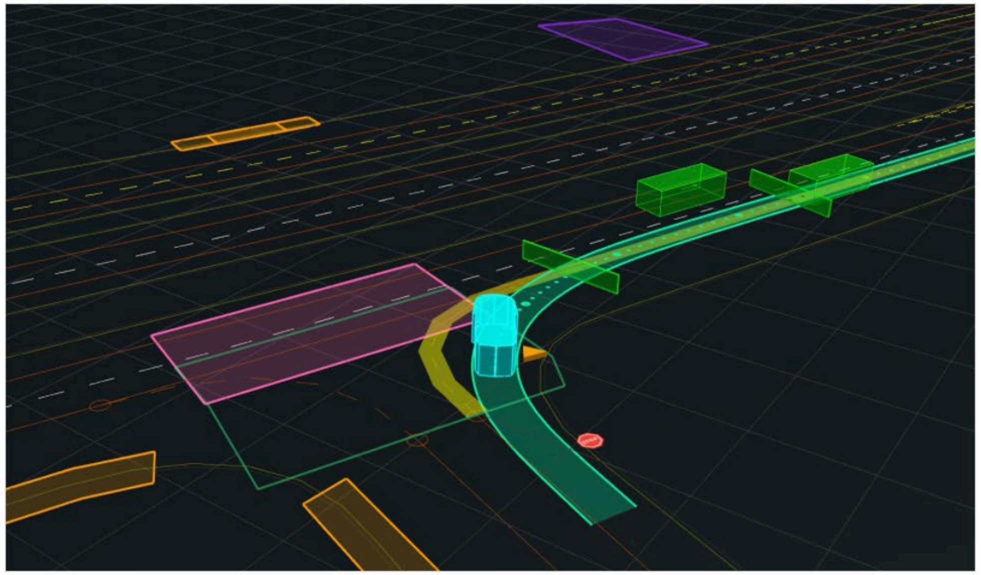
Later, when Mr. Wrigley posted the paragraph on Twitter, somebody looped in the real Scott Barry Kaufman. He was stunned. "It definitely sounds like something I would say," the real Mr. Kaufman tweeted, later adding, "Crazy accurate A.I."

Recent news from the real world (3/3)

April 28, 2020

Off road, but not offline: How simulation helps advance our Waymo Driver

TECHNOLOGY



Gaining 100+ years of experience in one day

Simulation is vital in the advancement of self-driving technology. At Waymo, one day in simulation is like driving more than 100 years in the real world. In simulation, we drive around 20 million miles a day, expanding the scale and complexity of our experience. True to our Alphabet heritage, our team has world-class expertise in applying cloud technology at massive scale to advance and grow our simulation efforts. To date, we have driven over 15 billion miles in simulation, and we continue to increase the velocity of our learning. In simulation, we can keep learning from each of our 20 million autonomous miles on public roads, prepare for rare edge cases, explore new ideas, validate and test new software, and continue improving our rider experience.

If this is exciting to you…

- Courses in Advanced Computing: ECE 508, ECE 511, CS 533
- Computational Science: CSE 401
- Topical Courses: Bioinformatics, Machine Learning / AI, Scientific Computing, Material Science

THAT'S ALL, FOLKS!

GOOD LUCK ON THE EXAM!

Please fill in end-of-semester course evaluation at
https://ices.citl.illinois.edu