CS 470 warm-up activity

- Introduce yourself to nearby classmates
- Work together as a group to answer the following questions:
  - Assume each computer in the room has at least four CPU cores (a reasonable assumption for computers <5 years old). How many cores do we have in this room total?
  - What is the world's largest and fastest supercomputer? Where is it located, and how many cores does it contain?
- Also, answer this one-question survey:

Visit **gosocrative.com** and enter room name LAMJMU

World's fastest supercomputer (2024)

- **Frontier** *(at least according to the Top500 list)*
  - **Oak Ridge National Laboratory** (Tennessee)
  - 74 cabinets w/ 8,000 lbs of equipment
  - 9,400+ EPYC 64C 2GHz CPUs
  - 37,000+ AMD Instinct 250X GPUs
  - Total of **8,699,904 cores**!
  - 700 PB storage w/ 75 TBps read bandwidth
  - HPE Cray OS (based on SUSE Linux)
  - 1.1 Eflops max Linpack performance
  - 23 MW power consumption

Sources:

- `top500.org`
- `ornl.gov`

CS 470
Spring 2024

Mike Lam, Professor

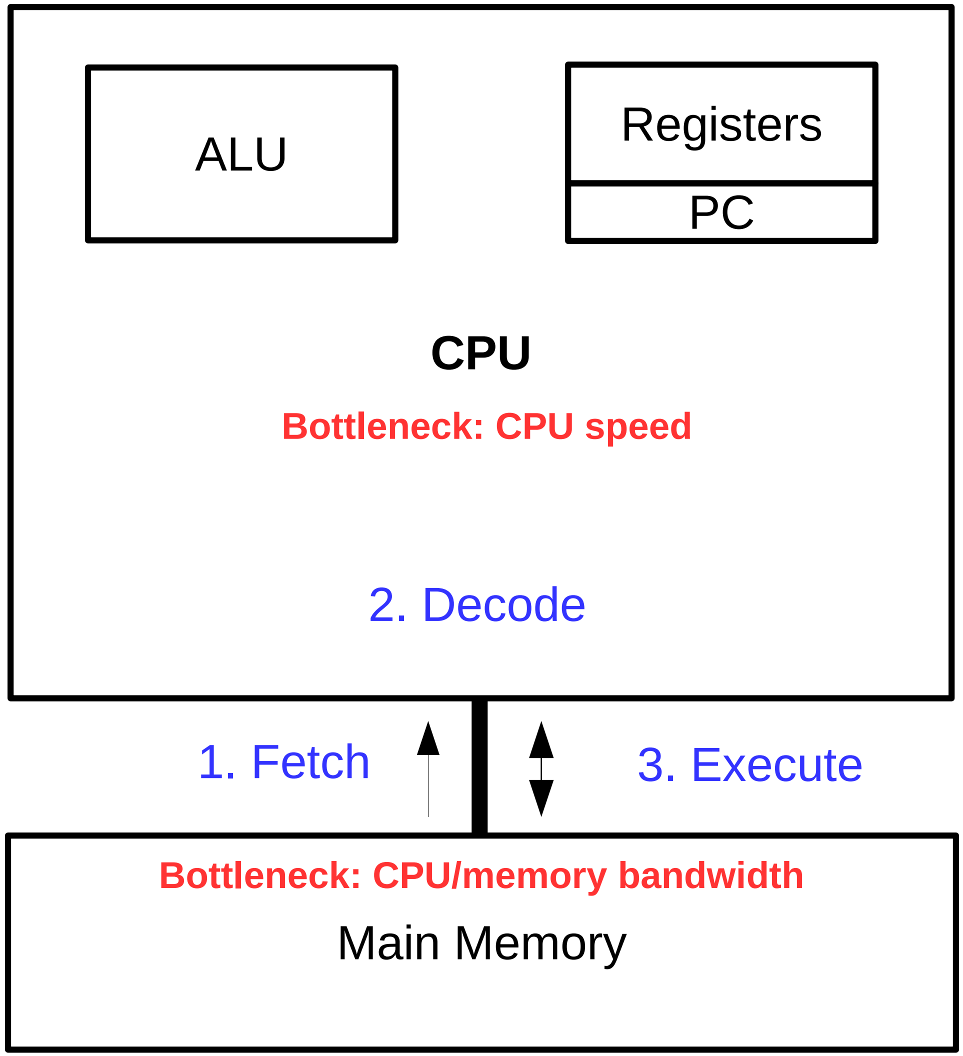
Parallel and Distributed Systems

Advanced System Elective

Motivation

- Why do we have (and why should we study) parallel and distributed systems?
- Let's go back to CS 261 …

von Neumann (CS 261)



History of parallelism

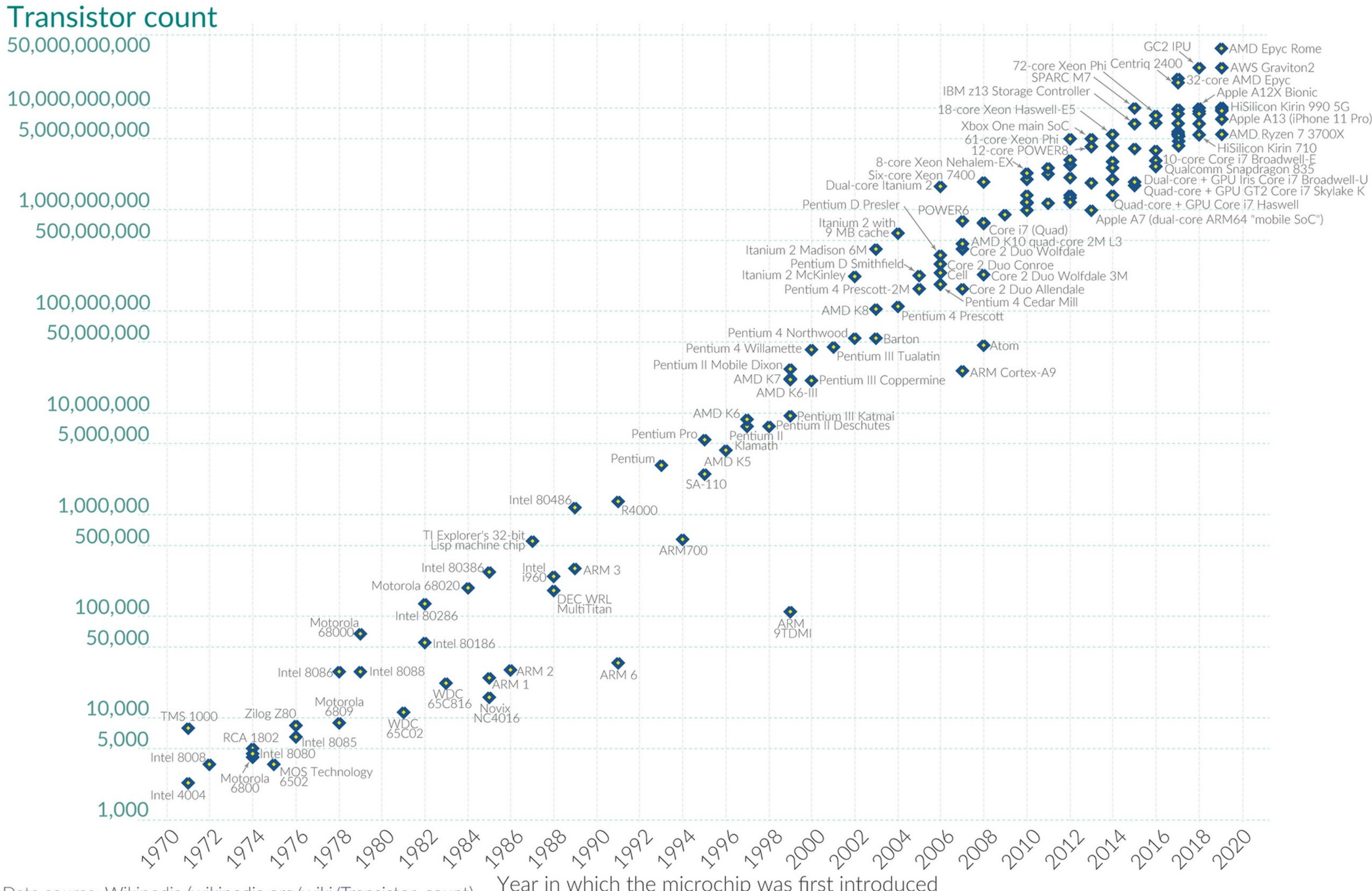
- Uniprogramming / batch (1950s)
  - Traditional von Neumann, no parallelism
- Multiprogramming / time sharing (1960s)
  - Increased utilization, lower response time
- Multiprocessing / shared memory (1970s)
  - Increased throughput, strong scaling
- Distributed computing / distributed memory (1980s)
  - Larger problems, weak scaling
- Hybrid computing / heterogeneous (2000s onward)
  - Energy-efficient strong/weak scaling

Moore’s Law

Moore’s Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world’s largest problems.

Semiconductor device fabrication

MOSFET scaling (process nodes)

- 10 µm – 1971
- 6 µm – 1974
- 3 µm – 1977
- 1.5 µm – 1981
- 1 µm – 1984
- 800 nm – 1987
- 600 nm – 1990
- 350 nm – 1993
- 250 nm – 1996
- 180 nm – 1999
- 130 nm – 2001
- 90 nm – 2003
- 65 nm – 2005
- 45 nm – 2007
- 32 nm – 2009
- 22 nm – 2012
- 14 nm – 2014
- 10 nm – 2016
- 7 nm – 2018
- 5 nm – 2020

Future

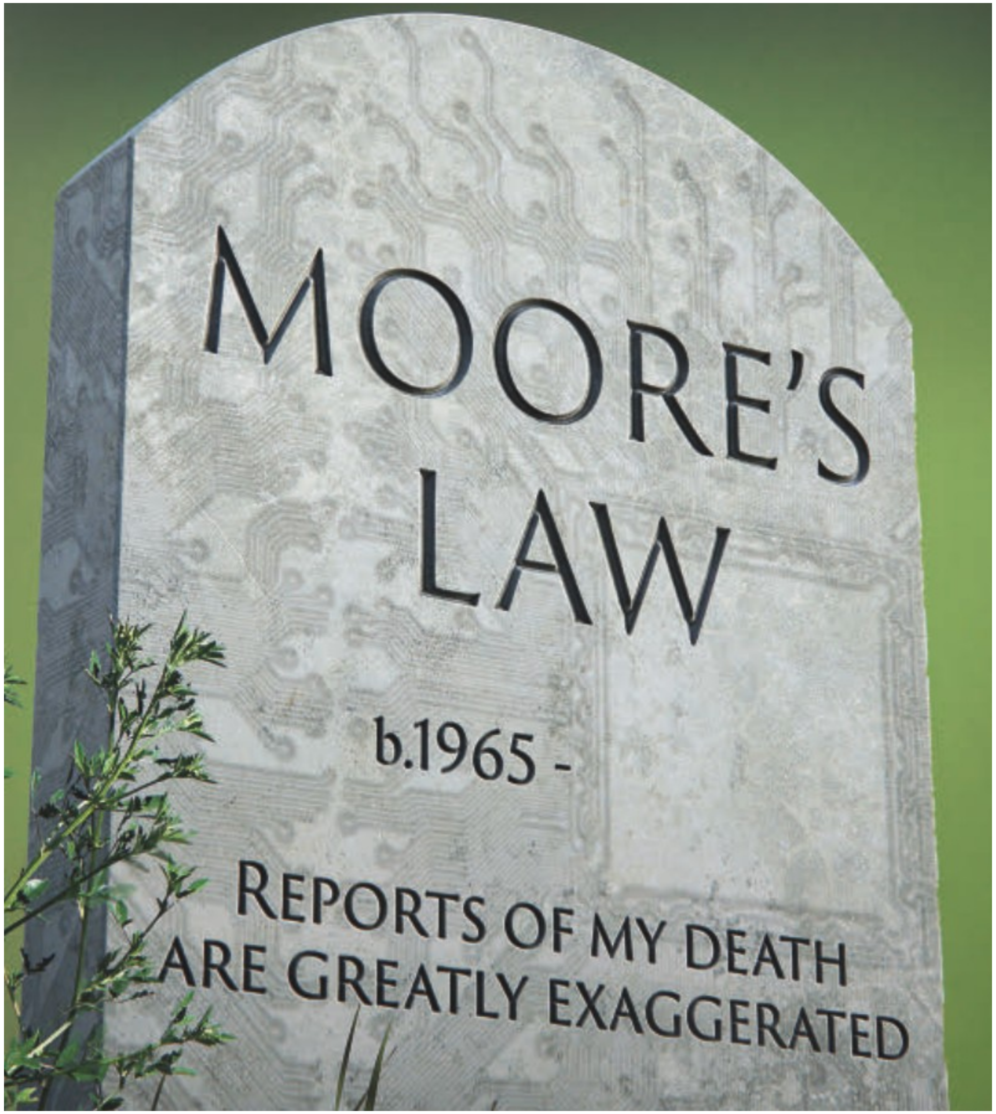
- 3 nm ~ 2022
- 2 nm ~ 2024

Issue: CPU Physics

- More transistors → higher energy use
- Higher energy use → higher heat
- Higher heat → lower reliability (e.g., signal leakage)
- Manufacturing limitations
- Quantum effects at sub-nanometer resolution
- Related observation: Dennard scaling (i.e., power consumption per area remains constant) failed in 2000s

Will Moore's Law eventually fail?

Moore's Law



Cover of the January 2017 edition of *Communications of the ACM*

Alternative to Moore's Law

- Scale out, not up
  - **More** processors rather than **faster** processors
    - (Remember Frontier's 2GHz processors?)
  - Requires parallelism at higher levels than instruction-level parallelism (e.g., pipelining)

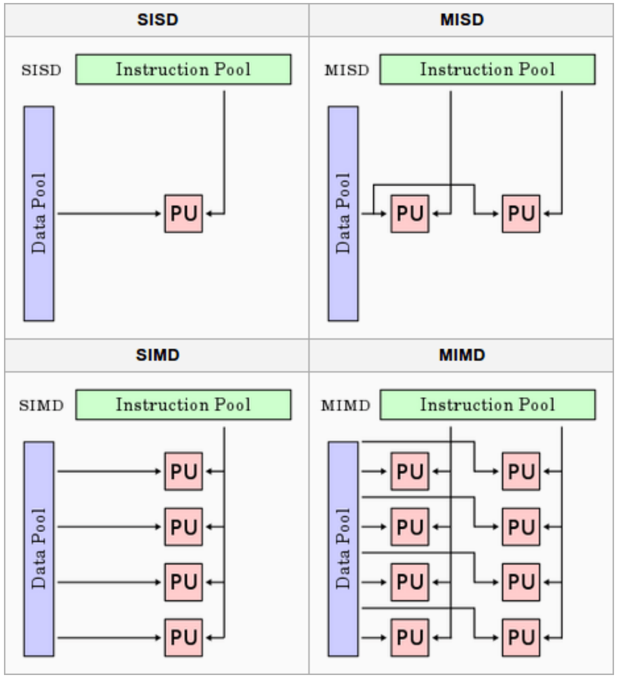
"Post-Moore's Law Era"

System architectures

- However, there's also a limit to how many cores we can put in a single computer
  - Energy consumption, heat emission, memory saturation
- Solution: more computers!
  - Communicate via network
  - This is called a distributed system
- There are so many kinds of parallelism
  - We need ways to concisely describe them and discuss their tradeoffs for particular applications

System architectures

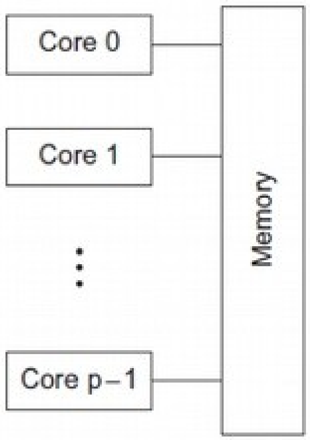
- Flynn's Taxonomy
  - Single Instruction, Single Data (SISD)
    - Traditional von Neumann
    - Increasingly insufficient!
  - Single Instruction, Multiple Data (SIMD)
    - Vector instructions (SSE/AVX) – remember from CS 261?
    - GPUs and other accelerators
  - Multiple Instruction, Multiple Data (MIMD)
    - Single Program, Multiple Data (SPMD)
    - Shared memory and distributed memory
  - Single Instruction, Multiple Threads (SIMT)
    - New term gaining prominence in past few years
    - Alternative way of describing GPUs



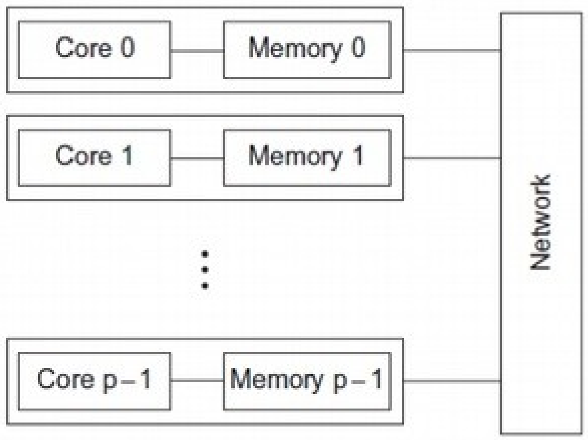
Trend: higher number of slower,
more energy-efficient processors

System architectures

- Shared memory
  - Idea: add more **CPUs**
  - Paradigm: threads
  - Technologies: Pthreads, OpenMP
  - Issue: synchronization



- Distributed memory
  - Idea: add more **computers**
  - Paradigm: message passing
  - Technologies: MPI, PGAS
  - Issue: data movement



Potential tradeoff between **simplicity** and **scalability**

Shared memory software

- Threading libraries
  - Low-level explicit multiprocessing programming
  - Independent threads of execution; shared variables
  - Synchronization mechanisms (locks, semaphores, conditions, barriers)
    - Prevents data races and enforces thread safety
  - Libraries: **Pthreads**, Java Threads, Boost Threads
- Language extensions
  - Write one program that is both serial and (implicitly) parallel
  - Use pragmas to annotate the program with parallelism guidelines
  - Threading and synchronization added automatically (usually by compiler)
  - Languages: **OpenMP**, OpenACC

Distributed memory software

- Message-Passing Interface (MPI)
  - Low-level explicit message-passing programming
  - Point-to-point operations (Send / Receive)
  - Collective operations (Broadcast / Reduce)
    - Allow MPI implementations to optimize data movement
  - Libraries: **OpenMPI**, MPICH, MVAPICH

- Partitioned Global Address Space (PGAS)
  - Make distributed memory look and act “like” shared memory
  - Split address space among all processes
  - Message passing is added automatically (usually by compiler)
  - Languages: Chapel

Hybrid architectures

- Shared memory on the node
  - Hardware: many-core CPU and/or coprocessor (e.g., GPU)
  - Enables energy-efficient strong scaling
  - Technologies: OpenMP, **CUDA**, OpenACC, OpenCL

- Distributed memory between nodes
  - Hardware: interconnect and distributed FS
  - Enables weak scaling w/ efficient I/O
  - Technologies: Infiniband, Lustre, HDFS, **MPI**

"**Summit**" supercomputer
Oak Ridge National Lab
Image from hpcwire.com

Shared memory summary

- Shared memory systems can be very efficient
  - Low overhead for thread creation/switching
  - Uniform memory access times (symmetric multiprocessing)
- They also have significant issues
  - Limited scaling (# of cores) due to interconnect costs
  - Requires explicit thread management and synchronization
  - Caching problems can be difficult to diagnose
- Core design tradeoff: synchronization granularity
  - Higher granularity: simpler but slower
  - Lower granularity: more complex but faster
  - Paradigm: synchronization is expensive

Distributed memory summary

- Distributed systems can scale massively
  - Hundreds or thousands of nodes, petabytes of memory
  - Millions of cores, petaflops of computation capacity
- They also have significant issues
  - Non-uniform memory access (NUMA) costs
  - Requires explicit data movement between nodes
  - More difficult debugging and optimization
- Core design tradeoff: data distribution
  - How to partition and arrange the data; is any of it duplicated?
  - Goal: minimize data movement
  - Paradigm: computation is “free” but communication is not

CS 470 technology summary

	Explicit	**Implicit**
Shared memory	Pthreads, CUDA	OpenMP
Distributed memory	MPI	*(none)*

Parallel systems are ubiquitous

- “New” problem: writing parallel software
  - Running a program in parallel is not always easy
  - Sometimes the **problem** is not easily parallelizable
  - Sometimes communication overwhelms computation
  - But the stakes are too high to ignore parallelism!

Core issue: parallelization

- As humans, we usually think sequentially
  - *"Do this, then that"* w/ deterministic execution

- Parallel programming requires a different approach
  - *"Do this and that in any order (or at the same time)"*
  - Introduction of non-determinism
  - Requires sophisticated understanding of dependencies

- Sometimes, the best parallel solution is to discard the serial solution and revisit the problem

Example from IPP

- Compute n values and calculate their sum
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

How should we parallelize this?

What problems will we encounter?

Example from IPP

- Initial parallel solution:

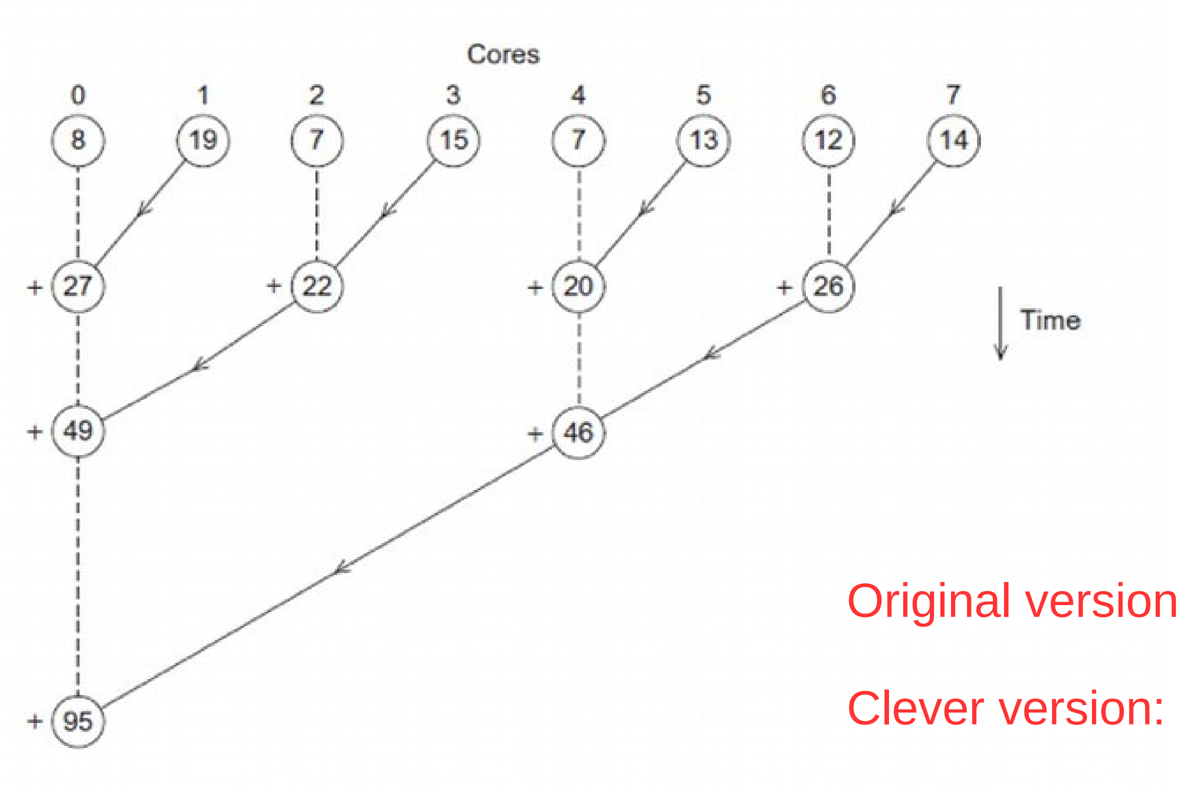
```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value( . . .);
   my_sum += my_x;
}

if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

Insight: split up the compute work, then have the master core aggregate the results

Shared-mem alternative: use a mutex!

Example from IPP

- There's a better way to compute the final sum
  - Distribute the work; don't do all the additions serially
  - Fewer computations on the critical path (longest chain of work)



Original version: 7 messages and 7 additions

Clever version: 3 messages and 3 additions

Example from IPP

- Improvement is even greater w/ higher # of cores
- For 1000 cores:
  - Original version: 999 messages and 999 additions
  - Clever version: 10 messages and 10 additions

This is an asymptotic improvement!

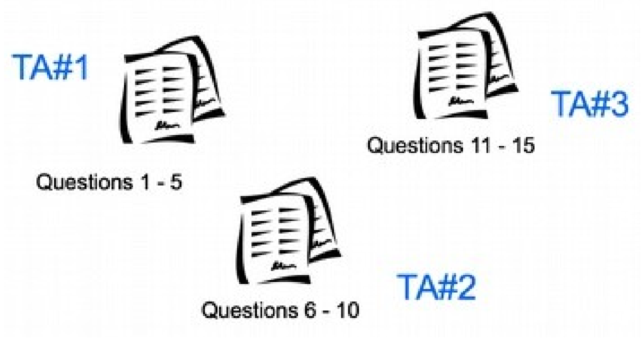
(*why*?)

Discussion

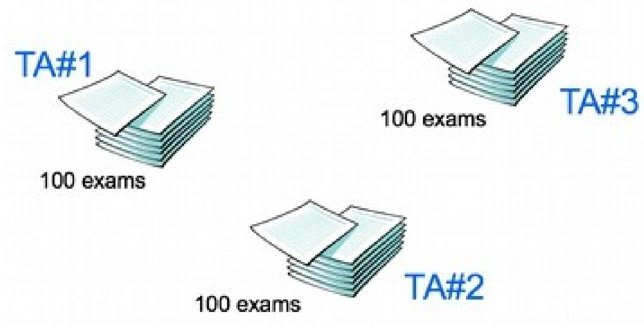
- Assume we have three graduate TAs to grade a 15-question exam for roughly 300 students. How do we finish the grading as quickly as possible?
  - Are there multiple valid approaches?

Kinds of parallelism

- **Task** parallelism / decomposition
  - Partition **tasks** among processes
  - Pass data between processes
  - Processes can be highly optimized



- **Data** parallelism / decomposition
  - Partition **data** among processes
  - Each process performs all tasks
  - Lower latency for individual outputs



Potential tradeoff between **throughput** and **latency**

Our goals this semester

- Learn some parallel & distributed programming technologies
  - Pthreads, OpenMP, CUDA, MPI
- Study parallel & distributed system architectures
  - Shared memory, distributed cluster, hybrid, cloud
- Study general parallel computing approaches
  - Parallel algorithms, message passing, task/data decomposition
- Analyze application performance
  - Speedup, weak/strong scaling, locality, communication overhead
- Explore parallel & distributed issues
  - Networks, synchronization & consistency, fault tolerance, security

Parallel & distributed systems

- Hardware architectures
- Software patterns & frameworks

  (w/ standard projects P1-P3)

First half of CS 470

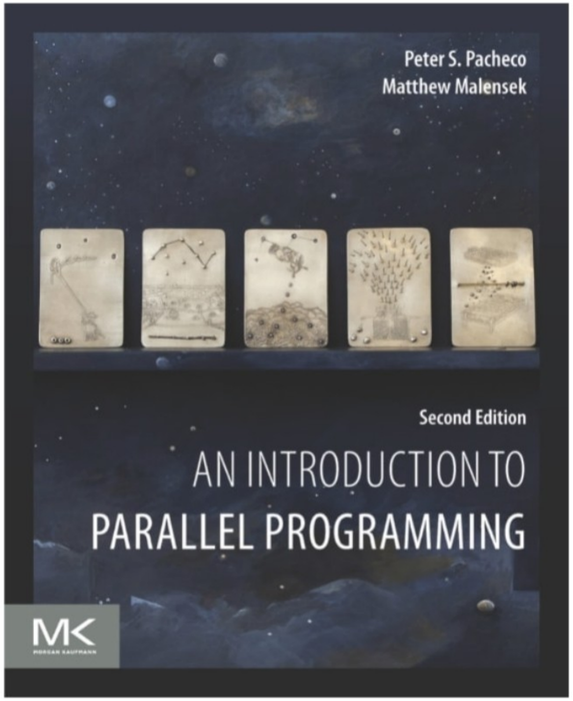
- Interconnects and naming
- Synchronization and consistency
- Fault tolerance
- Cloud computing
- Security
- Applications: Web & File Systems

  (w/ final project)

Second half of CS 470

Course textbook

- **An Introduction to Parallel Programming, 2nd Edition**
  - Peter S. Pacheco and Matthew Malensek
  - New edition!



- Sources:
  - JMU Bookstore ($70)
  - Amazon ($50)
  - Elsevier ScienceDirect (**free!**)
    - (electronic, link on syllabus)
  - Rose library (on reserve)

Course notes

- The course slides are the course notes, and they are quite comprehensive
  - Especially during the second half
  - Not all topics will be covered explicitly in class
  - **Most** topics will not be covered extensively in lectures
  - We'll focus on the most useful / difficult topics during class
  - You are responsible for reviewing the slides for all material not fully covered in class
  - Ask clarification questions on Discord

Course format

- Public files and calendar on website (bookmark it!)
- Private files and grades on Canvas

- Canvas quizzes (usually 1-2 per week)
  - Two attempts; goal is to prompt re-reading if needed
- In-class labs (usually ~1 per week) w/ Canvas submission
  - Groups of two or three (submit one copy with everyone's names)
- Standard projects (every 2-3 weeks in 1st half) w/ Canvas submission
  - Groups of up to two, individual code reviews
- Research project (entire semester, starting now!)
  - Groups of up to four (**three HIGHLY recommended**)
- In-class exams (midterm & final)

Course grades

Quizzes and Labs	20%
Projects	30%
Exams	50%

- Quizzes and labs are **formative**
  - Designed to help you learn
- Exams are **summative**
  - Designed to assess what you have learned
- Projects are **both**
  - Designed to give you experience writing parallel and distributed programs
  - Intended as both a learning experience but also to measure progress

Class policies

- If you test positive for COVID-19 or are consistently coughing and/or sneezing, **please stay home**
  - Contact me ASAP regarding missed class
  - If you feel a bit ill but well enough to attend class (and are NOT consistently coughing and/or sneezing), please consider wearing a surgical or N95/KN95 mask to protect others
  - Feel free to wear a mask in class or office hours for any reason
- Feel free to bring laptops to class
  - Please do not cause distractions for others
- These policies may change
  - Changes will be announced via Canvas message

Discord

- Synchronous and quicker than Canvas or email
- A link to the server is on Canvas
- Public channels
  - `#general`
  - `#random`
  - `#standard-projects`
  - `#research-projects`
- Private channels
  - Individual project groups
  - Formed later in the semester as needed

Assumed skills

- All material in CS 261 and CS 361
  - (we will review Pthreads a bit)
- Some other things you should be able to do:
  - Login to a remote Linux server via SSH in a terminal
  - Copy files and folders on the command line (cp)
  - Edit files from the command line (e.g., nano or vim)
  - Download files using the command line (e.g., curl or wget)
  - Implement a singly-linked list
  - Use GDB to find segfault sources
  - Use GDB or logs to trace execution
  - Use Valgrind to locate memory problems

Standard projects

- Practice using parallel and distributed technologies
- Practice good software engineering and code analysis
- Submission: code + analysis / review / response
  - Code can be written individually or in teams of two
    - Benefits vs. costs of working in a team
    - AI-assist technologies are **allowed** – statement required re: use
  - Analysis must be included as comments at top
    - Requirements will vary by assignment
  - Graded code reviews after project submission
    - Review two other submissions; must be done individually
  - Response to assess the reviews you receive

Research

- In addition to standard projects, we will be doing a research project this semester

Research project

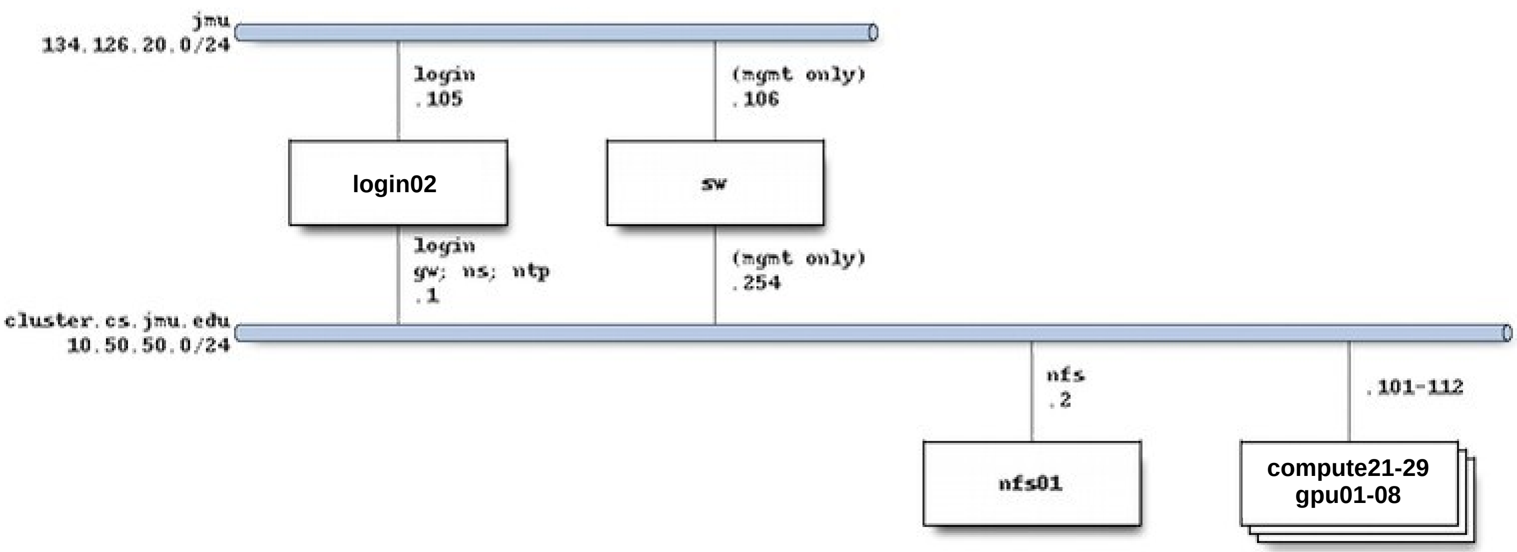
- Semester-long project
  - Teams of 2-4 people (**three HIGHLY recommended**)
  - Personalized topic; largely open-ended
  - Must involve parallel/distributed systems or software
  - Must include significant programming or analysis
  - Preferably uses Pthreads, OpenMP, MPI, or CUDA
  - Multiple submissions:
    - Overview + idea, groups, proposal, draft, poster, final
    - Use LaTeX for proposal and draft+final reports
  - Graded on progress and application of course concepts
  - Goal: **significant, open-ended, novel** “capstone” experience

Research Project

- Most importantly: **DON’T STRESS!**
- Read about a lot of previous projects
- Find a topic you’re excited about
- Keep project teams to 2-3 members
- If you **start early**, schedule time weekly to meet and work with your group, anticipate and leave time to deal with setbacks, and **communicate regularly** with me, your project *will* be successful.

Our distributed cluster

- **Compute nodes**:
  - 9x Dell PowerEdge R6525 w/ 2x AMD EPYC 7252 (8C, 3.1 Ghz, HT) 64 GB
  - 8x Dell PowerEdge R6525 w/ 2x AMD EPYC 7252 (8C, 3.1 Ghz, HT) 64 GB and NVIDIA A2 GPU
- **Login node**: Dell PowerEdge R6525 w/ 2x AMD EPYC 7252 (8C, 3.1 Ghz, HT) 64 GB
- **File server**: Dell PowerEdge R730 w/ Xeon E5-2640v3 (8C, 2.6Ghz, HT) 32 GB
  - **Storage**: 8x 1.2TB 10K SAS HDD w/ RAID
- **Interconnect**: Dell N3024 Switch 24x1GbE, 2x10GbE SFP+ (212Gbps duplex)



Cluster access

- Detailed instructions online:
  `w3.cs.jmu.edu/lam2mo/cs470/cluster.html`
- Connect to login node via SSH
  - Hostname: `login02.cluster.cs.jmu.edu`
  - User/password: *(your e-ID and password)*
- Recommended conveniences
  - Set up public/private key access from `stu`
  - Set up `.ssh/config` entries
    - w/ `stu` as jump host if you want off-campus access

Cluster access

Hostname: login02.cluster.cs.jmu.edu

- Things to play with:
  - "squeue" or "watch squeue" to see jobs
  - "srun <command>" to run an interactive job
    - Use "-n <p>" to launch *p* processes
    - Use "-N <n>" to request *n* nodes (defaults to *p*/16)
    - The given "<command>" will run in every process
    - Use "--gres=gpu" to request one of the GPU nodes
  - "srun -n <p> <command>" to run an interactive MPI job
    - Will launch *p* MPI processes

```
srun hostname
srun -n 4 hostname
srun -n 16 hostname
srun -N 4 hostname
srun sleep 5
srun -N 2 sleep 5
```

```
module load mpi
srun -n 1 /shared/cs470/mpi-hello/hello
srun -n 2 /shared/cs470/mpi-hello/hello
srun -n 4 /shared/cs470/mpi-hello/hello
srun -n 8 /shared/cs470/mpi-hello/hello
srun -n 16 /shared/cs470/mpi-hello/hello
(etc.)
```

What's the max *n*?

TODO items for this week

- Take course welcome survey if you haven't already
- Research project overview quiz due next Friday
  - Start talking with others about research topics!
- Make sure you can access Discord
- Make sure you can SSH into `login02.cluster.cs.jmu.edu`
  - Must be on JMU network (e.g., proxy jump through `stu`)
  - Email me **BEFORE** the next class if you encounter issues

Closing exhortations

- Take care of yourself
  - And if you can, someone else
  - Build (or reconnect with) a support network
  - Protect your boundaries
  - Carve out time to disconnect and rest
  - Talk to someone if things start getting overwhelming
- Have a great semester!