HIGH PERFORMANCE COMPUTING INTRODUCTION

Dr. Steve Petruzza

UtahStateUniversity

Reference material for part of this presentation:

An Introduction to Parallel Programming by Peter Pacheco

USU COVID-19 protocol

- **Get vaccinated**, upload proof of vaccination on aggiehealth.usu.edu
- **Wear masks**, at least when you are less than **6 feet** from nearest class member
- Keep track of **your seat** in this classroom, a new link will appear in Canvas starting September 7 called “seating assignment”
- Stay **home** if you are **sick**
- **Green-Red-Orange protocol**
  - *Green, F2F*
  - *Red, remote only (streamed and recorded)*
  - *Orange, hybrid F2F*

What we'll learn in this course

- Computing architectures, from single core to supercomputers
- How to write programs that are explicitly parallel and distributed (using the C/C++ language)
- Learn four different technologies
  - *Posix Threads (Pthreads)*
  - *OpenMP*
  - *Message-Passing Interface (MPI)*
  - *CUDA*
- How to apply parallel programming techniques to real use cases and applications
- Scalability challenges in HPC systems and software
- Use of Python for parallel programming

Important information for the course

- Grading
  - *Homework (30%), 5 assignments*
  - *Mid-term exam (25%)*
  - *Final exam (25%)*
  - *Project (20%)*
- Computing resources
  - *CHPC clusters, will need to request accounts for each student*
- Pre-requisites
  - *C/C++ background assessment, anonymous quiz: https://forms.gle/Fww41vEALbQgfRLU7*
- Communication via Canvas (only)
- Office hours (tentative): Wednesday and Friday 2.00-3.00PM
- TA: Parker Bywater

Homework

- Once every two weeks
- Test knowledge on a specific topic (the last topic we discussed)
- Mix of quiz and coding exercises

Mid-term and finals

- Test knowledge on all the topics discussed
- Mix of quiz and coding exercises (no compilation required)

Project

- Develop different versions of a parallel application using multi threading, MPI and CUDA
- New: team project where you can mix the different technologies in the same application (e.g., simulation and analysis/vis)

What to expect

- “He's a **nice person**, but beware, he's **very punishing** grading. His assignments are cool but **often vague**. He might teach OpenMPI/ OpenMP/Cuda, ask for one, and dock 90% if it wasn't the one he wanted, but it wasn't always labeled. He was nice but hard to understand. You worked 20-70 hrs a week on this class and **most of the class were failing** or Cs.”
- Grading
  - *47% (A/B+), 30% (B,C+), 0.05% (D), 17% (F, dropped early)*

What to expect (some hints)

- Keep up with the material and the homework and you will do great!
- Book and examples can be extremely useful
  - *An Introduction to Parallel Programming. Peter Pacheco*
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface. Anthony Skjellum, William Gropp, and Ewing Lusk*
  - *Programming Massively Parallel Processors, 2nd Edition. David B. Kirk, Wen-mei W. Hwu*

Why High Performance Computing

- Why we need “High performance” computing
- Why we’re building parallel systems
- Why we need to write parallel programs
- How do we write parallel programs
- Concurrent, parallel, distributed!

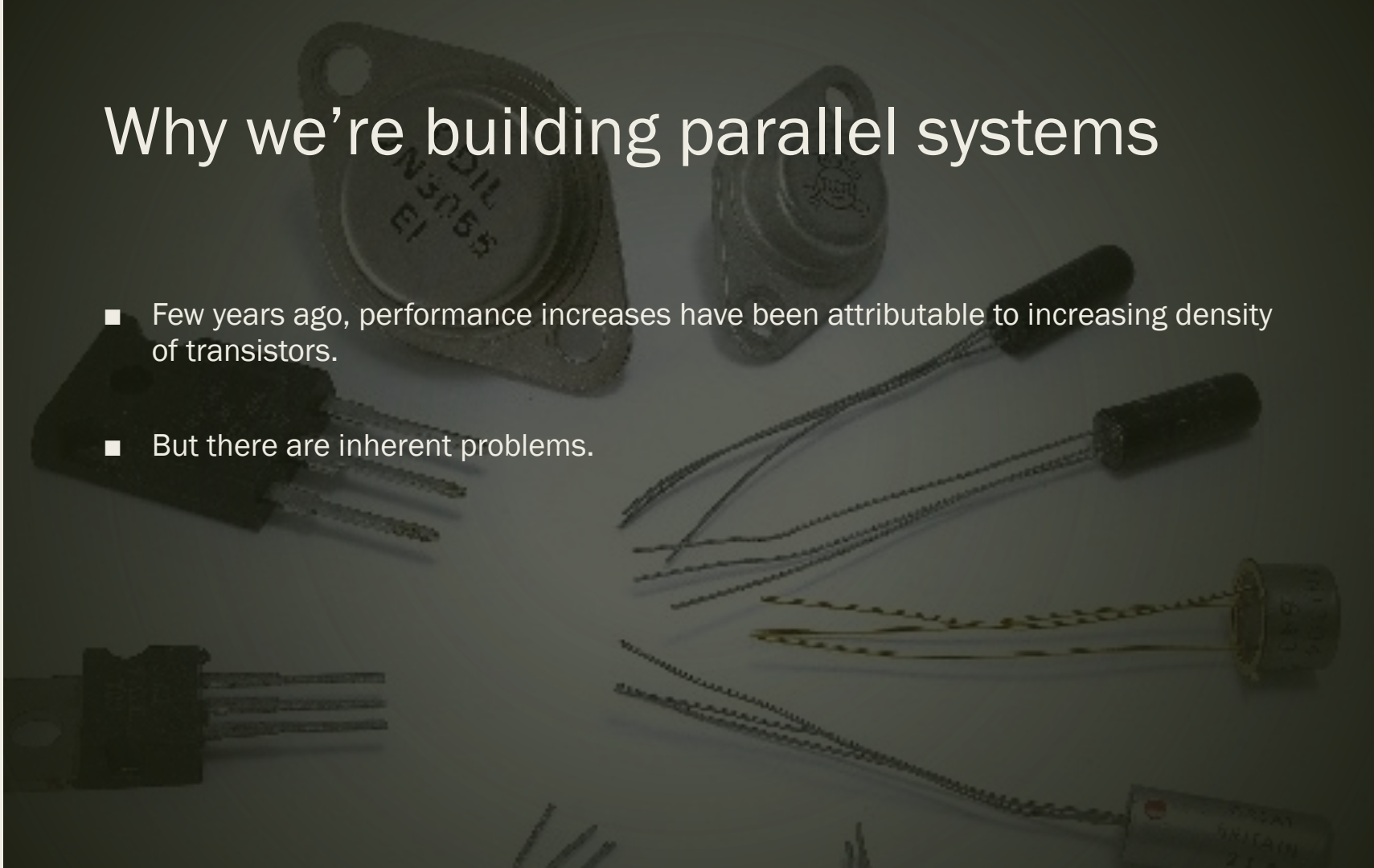
What if computing was just like folding a pizza box?...

Parallelism can make the difference!

Goal: Maximize throughput

1 box every ~3sec = 20 bpm (boxes per minute) !

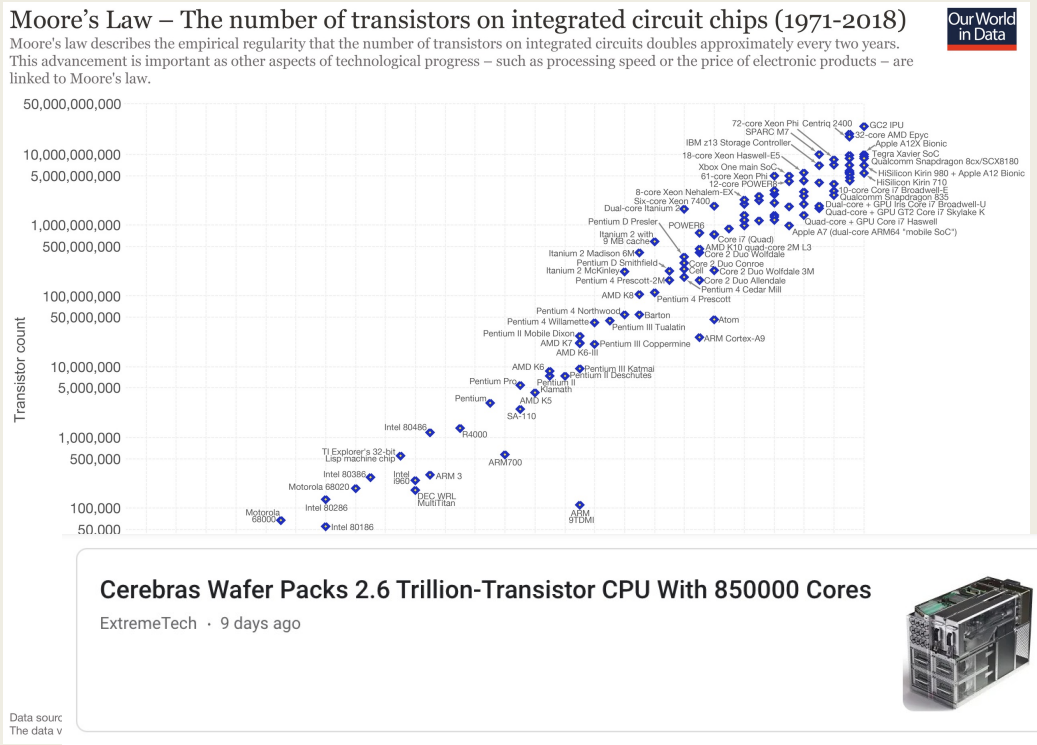
Why we're building parallel systems



- Few years ago, performance increases have been attributable to increasing density of transistors.
- But there are inherent problems.

A little physics lesson: Moore's law

- Moore's Law says that transistors will double every two years
- Smaller transistors = faster processors.
- Faster processors = increased power consumption
- Increased power consumption = increased heat
- Increased heat = unreliable processors



Cerebras Wafer Packs 2.6 Trillion-Transistor CPU With 850000 Cores

ExtremeTech · 9 days ago

The world's largest chip: 2.6 trillion transistors and 850,000 cores

TweakTown · 8 days ago

Aug 2020

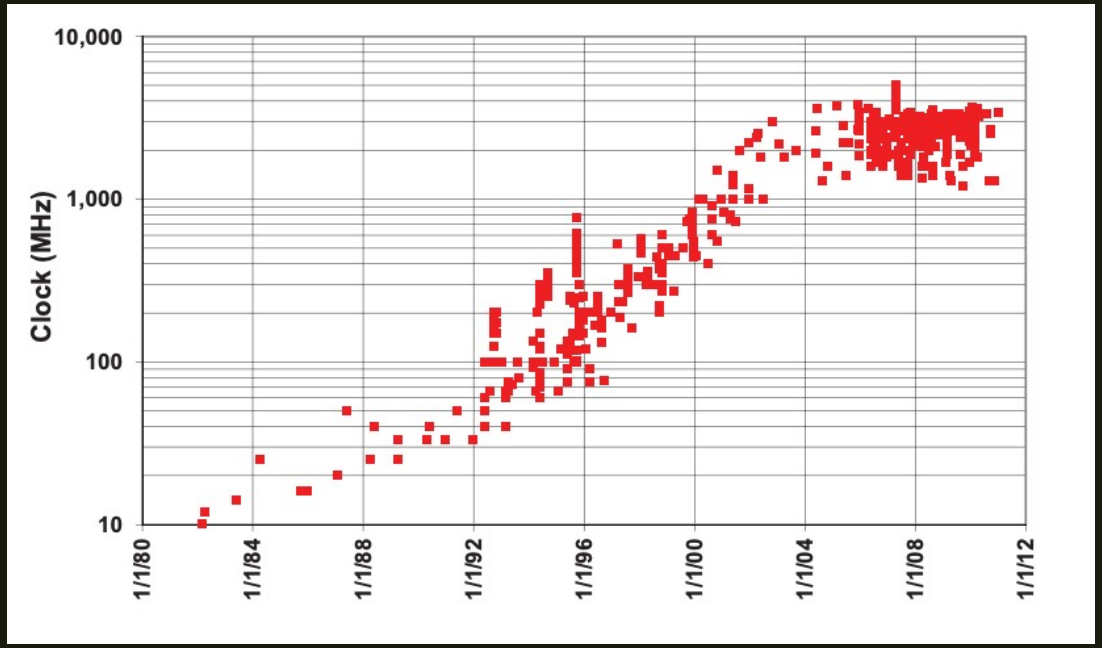
A little physics lesson: Dennard Scaling

- Why haven't clock speeds increased, even though transistors have continued to shrink?
- Dennard (1974) observed that voltage and current should be proportional to the linear dimensions of a transistor
  - *Thus, as transistors shrank, so did necessary voltage and current; power is proportional to the area of the transistor.*

BUT

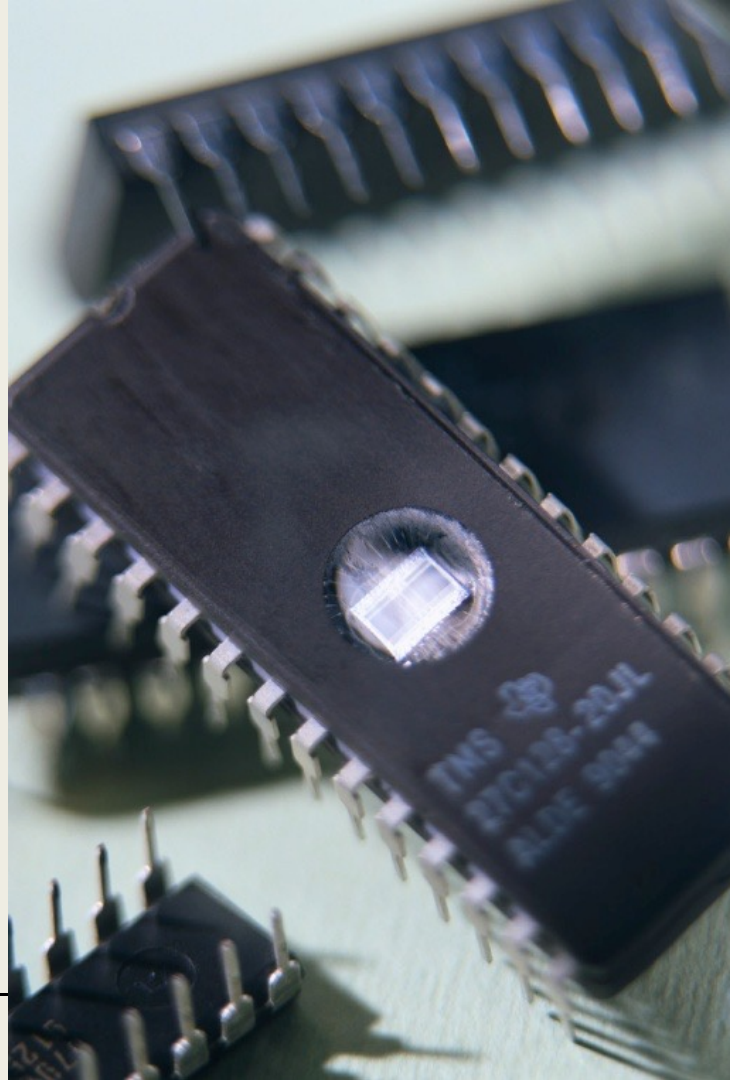
- Dennard scaling ignored the "leakage current" and "threshold voltage", which establish a baseline of power per transistor.
- As transistors get smaller, power density increases because these don't scale with size
- These created a "Power Wall" that has limited practical processor frequency to around 4 GHz since 2006

HISTORIC CLOCK RATES

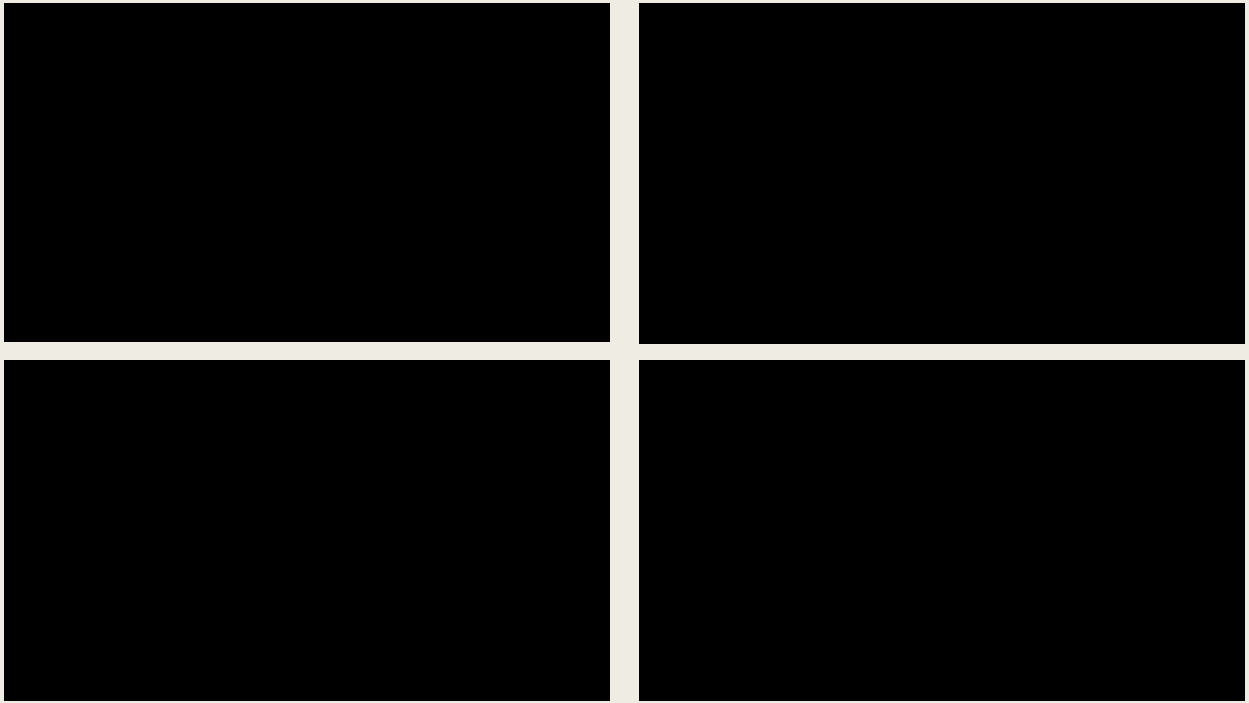


Solution

- Move away from single-core systems to multicore processors.
- “core” = central processing unit (CPU)
- Introducing parallelism!!!



Multi-“Dale” architecture



4 boxes every ~3sec = 80 bpm = 1.33 boxes per second !

HPC systems are more than just a collection of CPUs

- Many other important component affect performance
  - *Network technology and topology*
  - *Disk I/O*
  - *Other shared resources*
- Improving the performance of only a single component can generate bottlenecks and affect the overall performance
  - *if we can make 80 pizza boxes per minute, but my oven can only bake 1 pizza per minute, my overall performance will be bounded by the slowest component (the oven)*
  - *Where do I store the additional pizza boxes?*

New challenges for the programmers

- Adding more processors doesn’t help much if programmers aren’t aware of them…
- … or don’t know how to use them.

- Serial programs don’t benefit from this approach (in most cases).

Why we need ever-increasing performance

Computational power is increasing, but so are our computation problems, data and needs.

Problems we never dreamed of have been solved because of past increases, such as decoding the human genome.

More complex problems are still waiting to be solved.

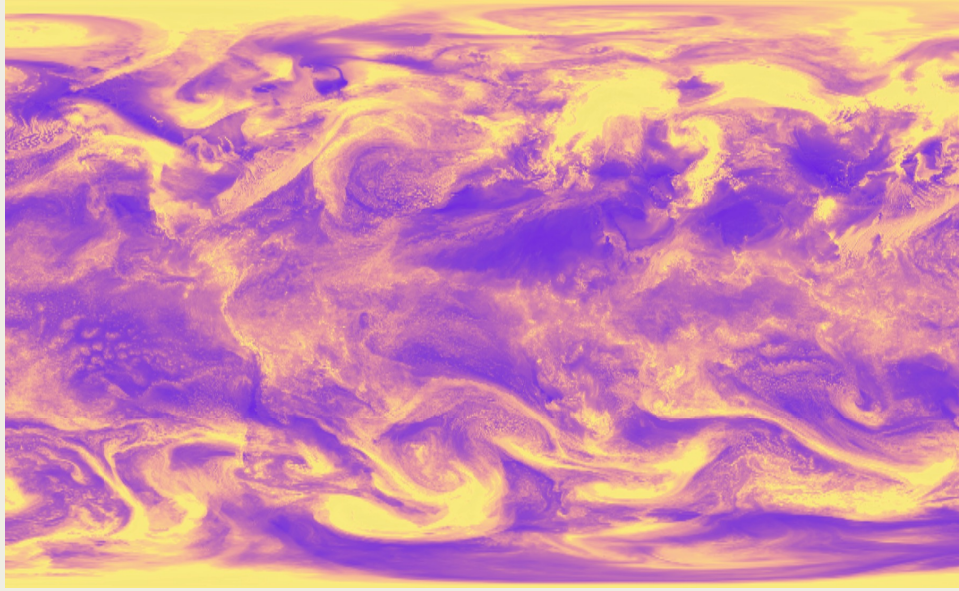
Crisis management

- Fukushima March 2011, a tsunami damages the backup generators of the nuclear power plant
- Cooling systems failed
- Rising heat lead to partial melt down of fuel rods, releasing radiations

©Matt Groening

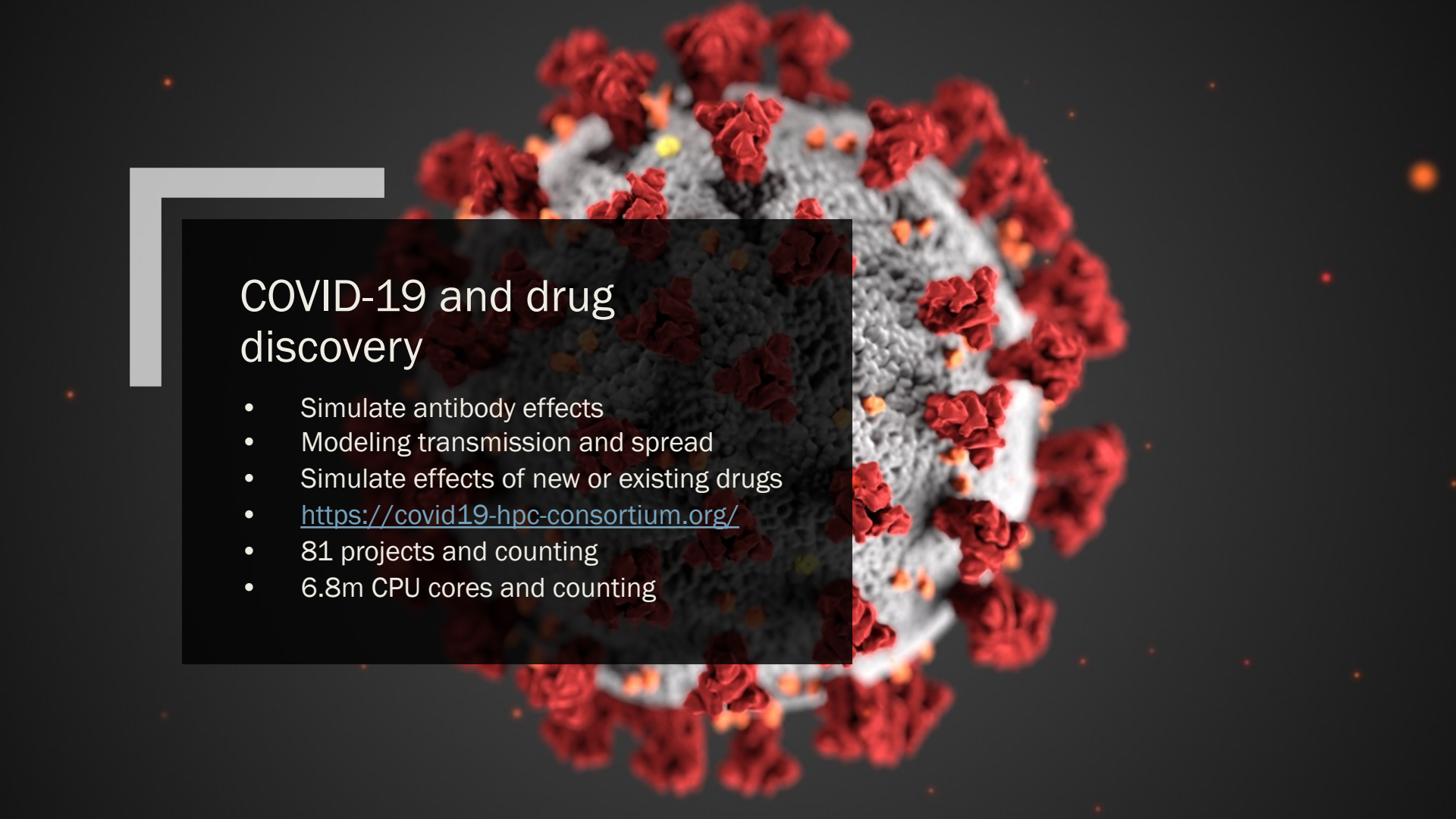
What would you do?

- Evacuation plan (poll)
- Containment plan
- Decontamination plan
- Where, when, how?
- HPC helped simulating many of those complex problems and provided decision maker with data to support their decisions



Climate modeling

- Climate models are built running thousands of simulations
- Understand current trends
- Make more accurate predictions



COVID-19 and drug discovery

- Simulate antibody effects
- Modeling transmission and spread
- Simulate effects of new or existing drugs
- https://covid19-hpc-consortium.org/
- 81 projects and counting
- 6.8m CPU cores and counting

Big Data analysis

- How much data is big data?
- Financial analysis and prediction models
- Machine learning and AI applications

Why we need to write parallel programs

- Running multiple instances of a serial program often isn't very useful.
- Think of running multiple instances of your favorite game.

- What you really want is for it to run faster.

Approaches to the serial problem

- Rewrite serial programs so that they're parallel.
- Write translation programs that automatically convert serial programs into parallel programs.
  - *This is very difficult to do.*
  - *Success has been limited.*

More problems

Some coding constructs can be recognized by an automatic program generator, and converted to a parallel construct.

However, it's likely that the result will be a very inefficient program.

Sometimes the best parallel solution is to step back and devise an entirely new algorithm.

Example

- Compute n values and add them together.
- Serial solution:

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

Example (cont.)

- We have p cores, p much smaller than n.
- Each core performs a partial sum of approximately n/p values.

```
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;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores.

Example (cont.)

- After each core completes execution of the code, is a private variable my_sum contains the sum of the values computed by its calls to Compute_next_value.

- Ex., 8 cores, n = 24, then the calls to Compute_next_value return:

1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

Example (cont.)

- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated “master” core which adds the final result.

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

Example (cont.)

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Global sum

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

Do you see any performance bottleneck in this approach?

Focus on the operations that each core is executing

Do you find any imbalance?

Can we do better?

Analyze the parallel execution

T / Core	0	1	2	3	4	5	6	7
[T0,T2]	next value, sum	next value, sum	next value, sum	next value, sum	next value, sum	next value, sum	next value, sum	next value, sum
[T3]	Rec sum 1, update sum	Send sum 1	Send sum 2	Send sum 3	Send sum 4	Send sum 5	Send sum 6	Send sum 7
[T4]	Rec sum 2, update sum							
[T5]	Rec sum 3, update sum	Multiple cores are idle (i.e., not executing any operations)						
… T9	… update sum							

Better parallel algorithm

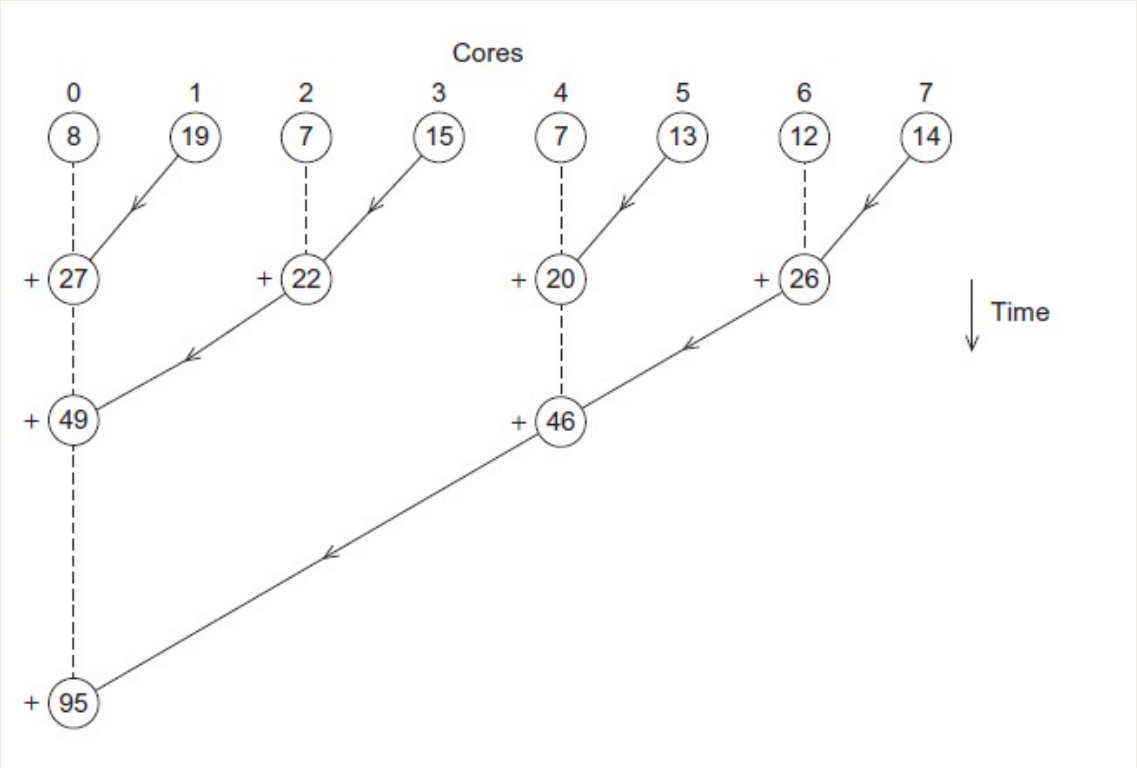
- Don’t make the master core do all the work.
- Share it among the other cores.
- Pair the cores so that core 0 adds its result with core 1’s result.
- Core 2 adds its result with core 3’s result, etc.
- Work with odd and even numbered pairs of cores.

Better parallel algorithm (cont.)

- Repeat the process now with only the evenly ranked cores.
- Core 0 adds result from core 2.
- Core 4 adds the result from core 6, etc.

- Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.

Multiple cores forming a global sum



Analysis

- In the first example, the master core performs 7 receives and 7 additions.
- In the second example, the master core performs 3 receives and 3 additions.
- The improvement is more than a factor of 2!

Analyze the parallel execution

Core	0	1	2	3	4	5	6	7
[T0,T2]	next value, sum	next value, sum	next value, sum	next value, sum	next value, sum	next value, sum	next value, sum	next value, sum
[T3]	Rec sum 1, update sum	Send sum 1	Rec sum 2, update sum	Send sum 3	Rec sum 5, update sum	Send sum 5	Rec sum 7, update sum	Send sum 7
[T4]	Rec sum 2, update sum				Rec sum 7, update sum			
[T5]	Rec sum 4, update sum							

Analysis (cont.)

- The difference is more dramatic with a larger number of cores.
- If we have 1000 cores:
  - *The first example would require the master to perform 999 receives and 999 additions.*
  - *The second example would only require 10 receives and 10 additions.*
- That's an improvement of almost a factor of 100!

How do we write parallel programs?

Task parallelism

- *Partition various tasks carried out solving the problem among the cores.*

Data parallelism

- *Partition the data used in solving the problem among the cores.*
- *Each core carries out similar operations on it's part of the data.*

©Matt Groening

Example: grading assignments

- 15 questions
- 300 exams
- 3 grading assistants

Division of work – data parallelism

TA#1

100 exams

TA#3

100 exams

TA#2

100 exams

Division of work – task parallelism

TA#1

Questions 1 - 5

TA#3

Questions 11 - 15

TA#2

Questions 6 - 10

DIVISION OF WORK – DATA PARALLELISM

```
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;
}
```

Division of work – task parallelism

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

Tasks

1) Receiving
2) Addition
3) Sending

Coordination

- Cores usually need to coordinate their work.
- Communication – one or more cores send their current partial sums to another core.
- Load balancing – share the work evenly among the cores so that one is not heavily loaded.
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

Some Terminology

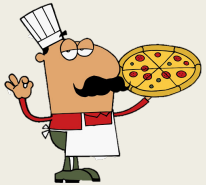
- Sequential computing – a program is one in which only one task can be in progress at any instant
- Concurrent computing – a program is one in which multiple tasks can be in progress at any instant.
- Parallel computing – a program is one in which multiple tasks cooperate closely (simultaneously) to solve a problem
- Distributed computing – a program may need to cooperate with other programs to solve a problem.

An example

You need to prepare **two** pizzas for delivery.

Tasks

- **prepare** the pizza



- **put** it in the oven

- **make** the pizza box

- **place** the pizza into the box

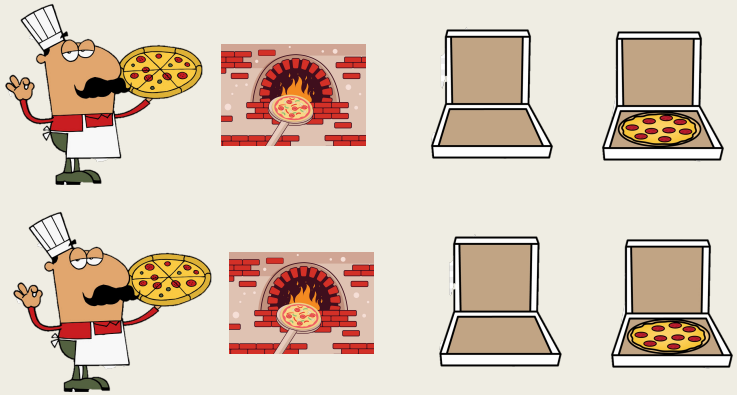
Sequential and concurrent execution

Sequential case

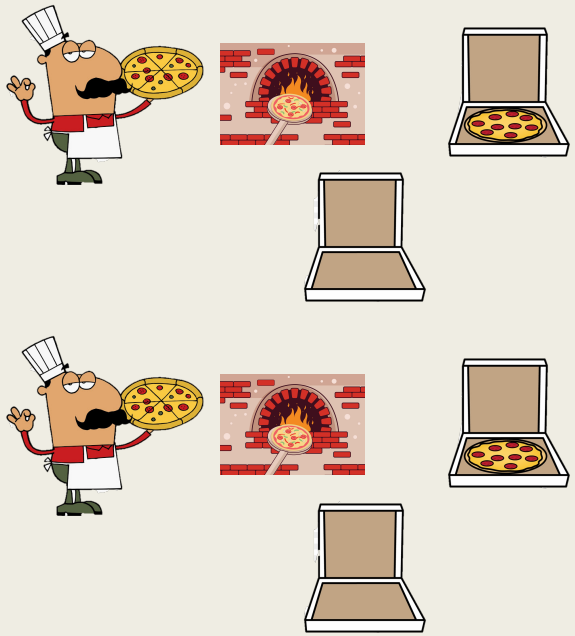
Concurrent case

Parallel execution

Parallel case



Parallel and Concurrent case



What's next?

- Memory, cache, virtual memory
- Computing architectures for parallel execution