# Intro to Supercomputing

Ramses van Zon

June 9, 2020

# What is Supercomputing?

**Supercomputing, a.k.a. High Performance Computing, is leveraging larger and/or multiple computers to solve computations in parallel.**

## What does it involve?

- hardware - pipelining, instruction sets, multi-processors, inter-connects
- algorithms - concurrency, efficiency, communications
- software - parallel approaches, compilers, optimization, libraries

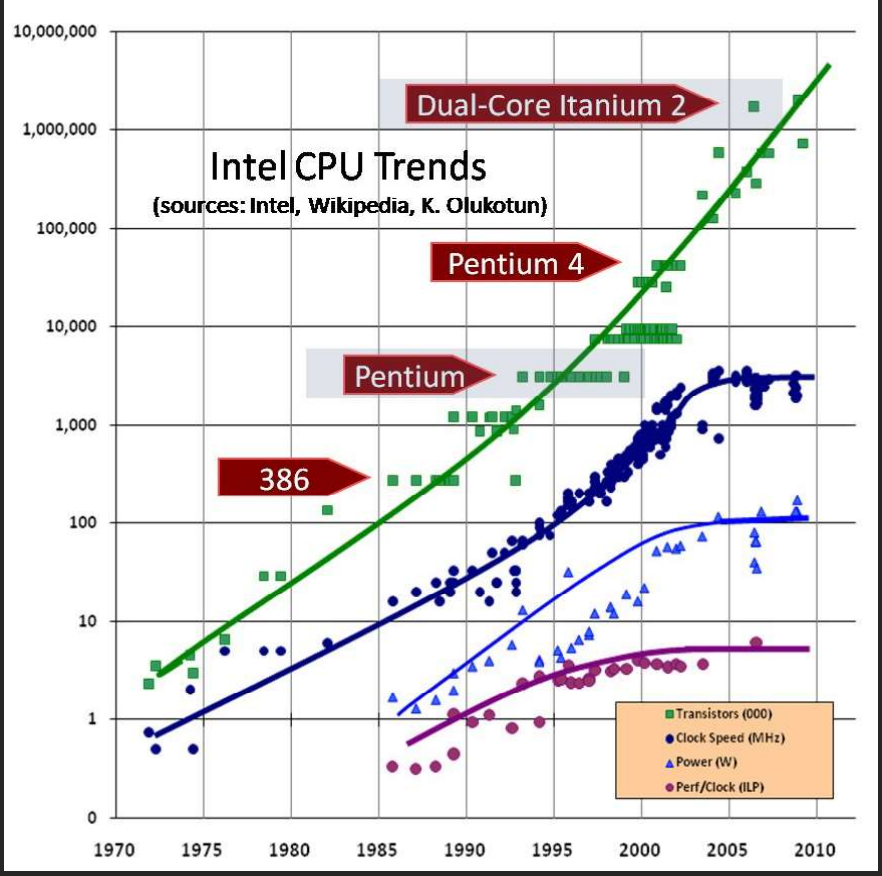
## When do I need Supercomputing/HPC?

- My problem takes too long $\rightarrow$ more/faster computation
- My problem is too big $\rightarrow$ more memory
- My data is too big $\rightarrow$ more storage

# Examples where supercomputing is needed

- Computational Fluid Dynamics
- Molecular Dynamics and N-Body Simulations
- Smooth Particle Hydrodynamics
- Monte Carlo Simulations
- Computational Quantum Chemistry
- Bioinformatics
- Data Science and Machine Learning

# The “free lunch” is over

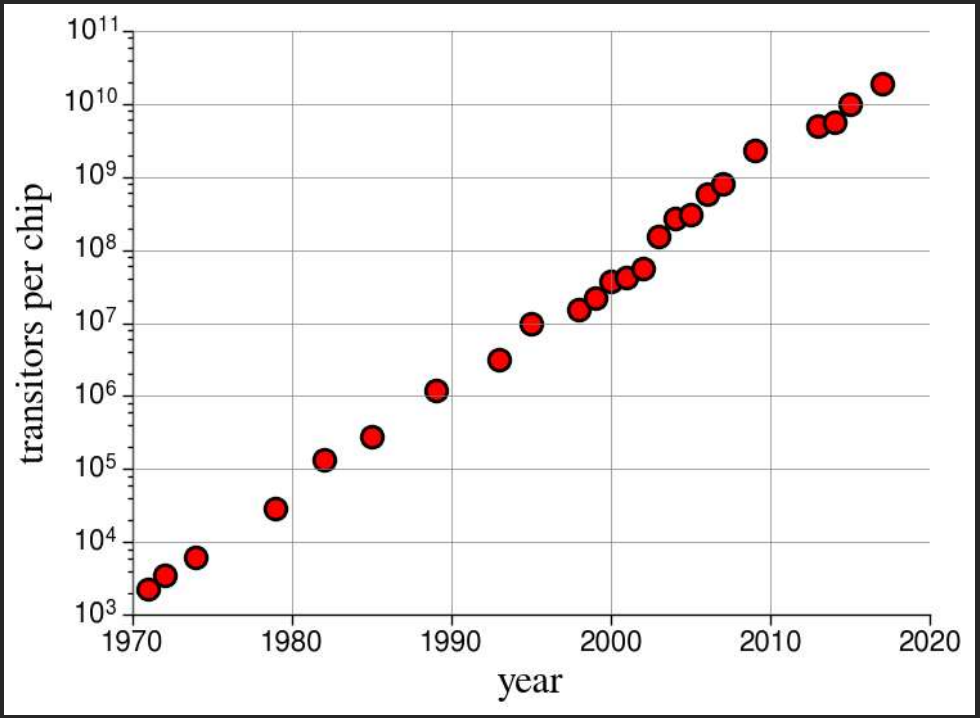



- There once was a time in which computer processor speeds steadily increased in newer generations.
- Due to physical limitations, this trend stopped around 2005, and advances in the speed of processors, memory, and storage, have plateaued.

So:

- Modern HPC means **more** hardware, not faster hardware.
- Thus **parallel** programming and computing is required.

# Wait, what about Moore's Law?



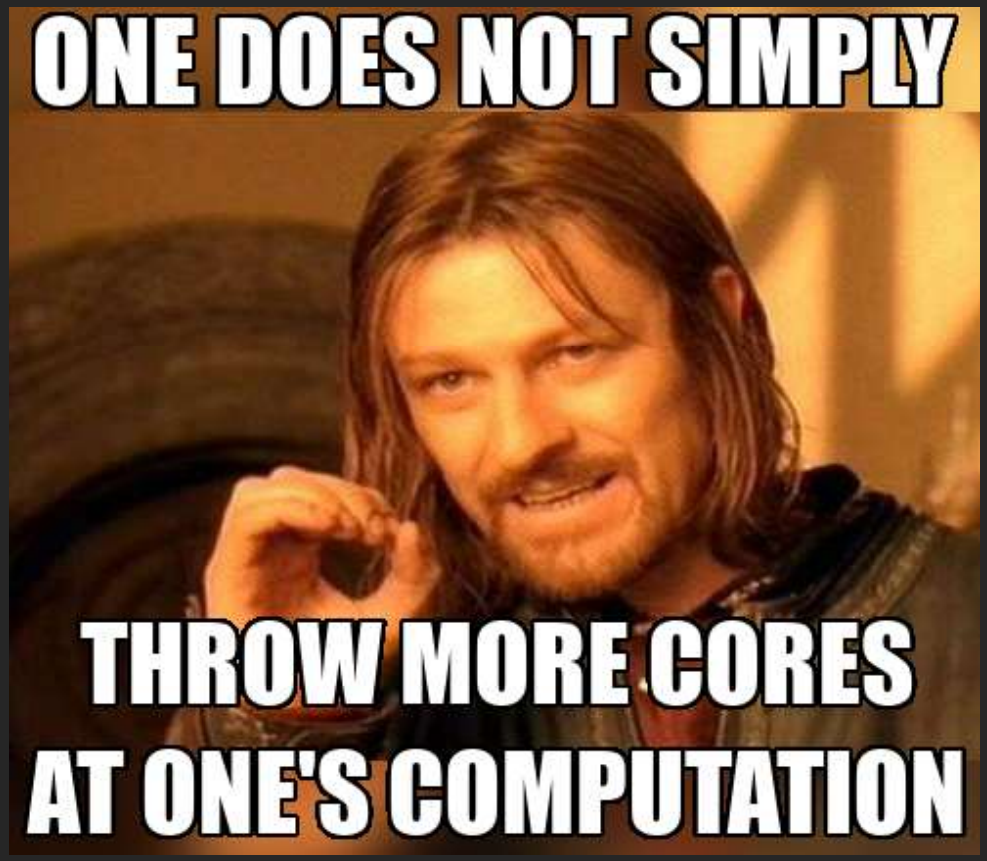

***Moore's law...***

describes a long-term trend in the history of computing hardware. The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years.

***But...***

- Moore's Law didn't promise us clock speed.
- More transistors but getting hard to push clock speed up.
- Power density is limiting factor.
- So more cores at fixed clock speed.

# All good, more cores = faster, right?




More cores is like having more workers.

***HR Dilemma***

Problem: job needs to get done faster

- can't hire substantially faster people
- can hire more people
- must alter workflow from a one-person job

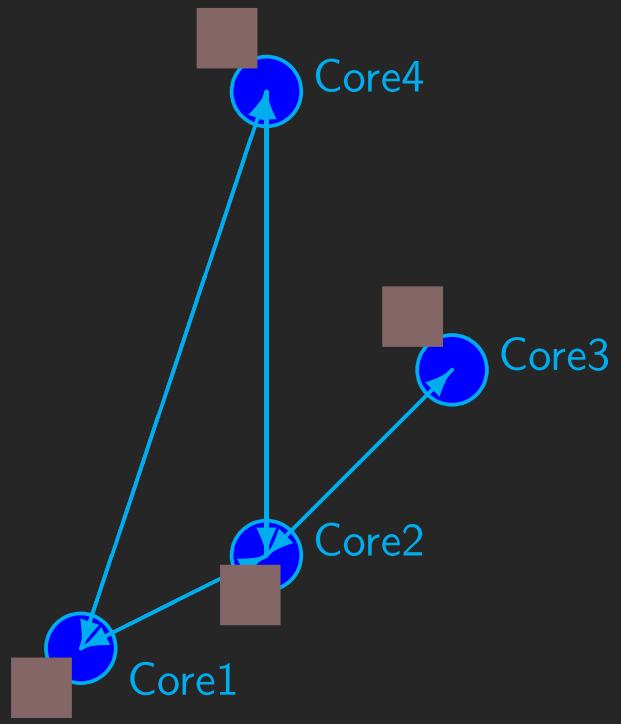
Solution:

- split work up between people (divide and conquer)
- requires rethinking the workflow process
- requires administration overhead
- eventually administration larger than actual work

# Supercomputer Architectures

# Clusters

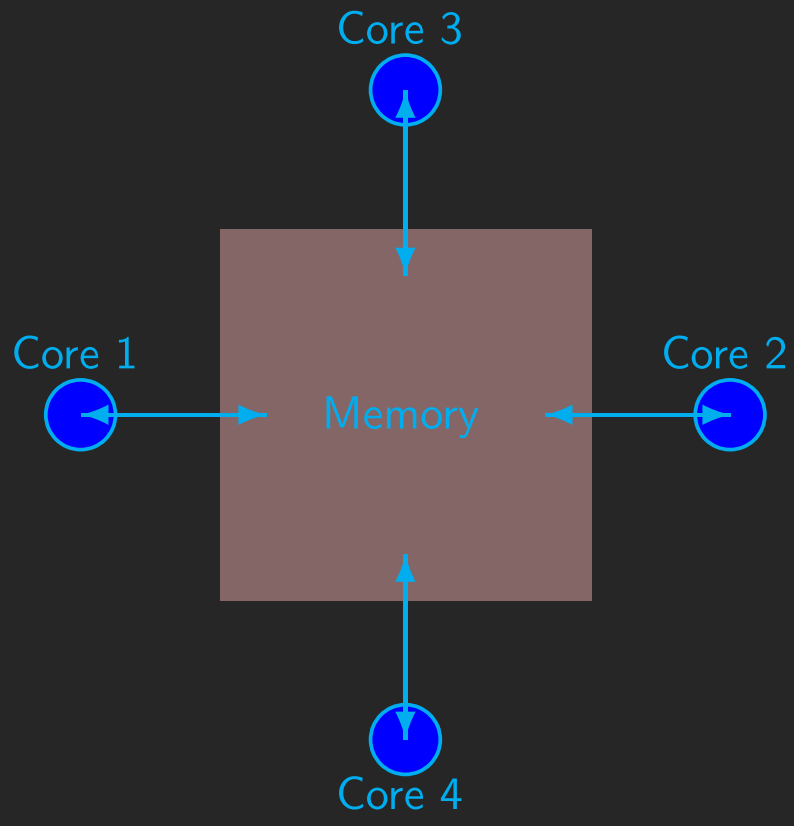
- Take existing powerful standalone computers (called a "node"),
- Link them together through a network (or "*interconnect*").
- Easy to build and easy to expand.
- Because each nodes has its own memory that the other nodes cannot see, these are called distributed memory systems.
- Nodes communicate and transfer data through messages.
- *Programming Model:* Message Passing Interface (MPI)



# Multi-core Computers

- A collection of processors that can see and use the same memory.
- Limited number of cores, and much more expensive when the number of cores is large.
- Coordination/communication done through memory.
- Also known as shared-memory systems.
- *Programming model:* Threads (e.g. OpenMP)

Your desktop, laptop and cell phone likely use this kind of architecture.



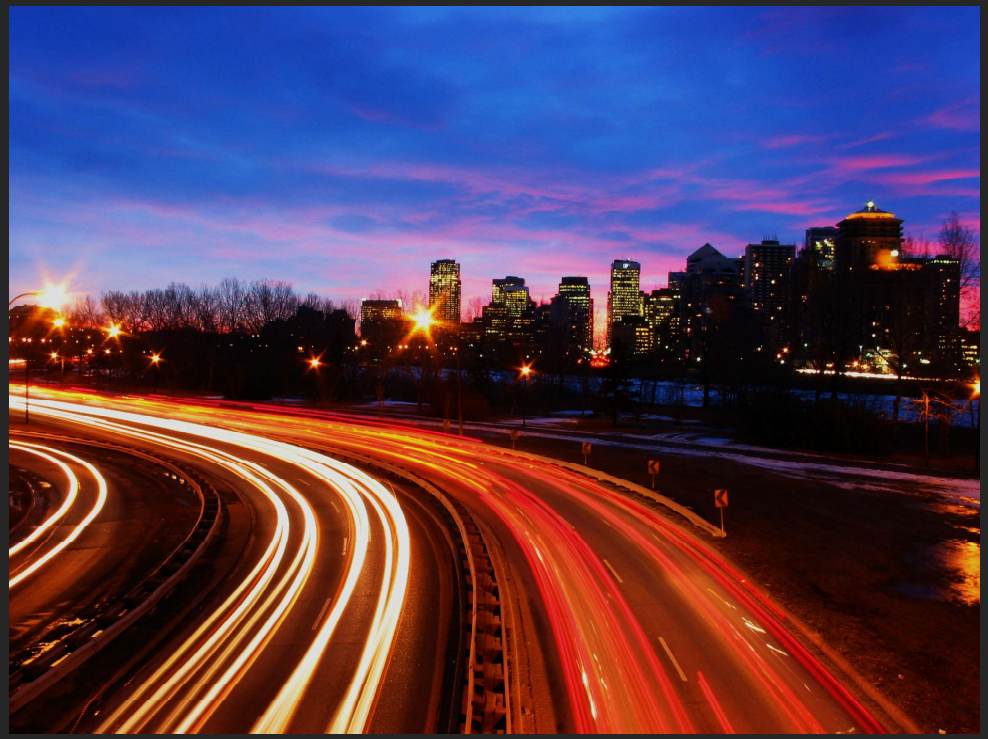
# Accelerators

- Systems with accelerators are machines which contain an "off-host" accelerator, such as a GPU or Xeon Phi.
- These accelerator devices are very fast and good at massively parallel processing (having 500-2000+ cores).
- Complicated to program.
- Programming model: CUDA, OpenACC, and OpenCL.
- Needs to be combine with at least some 'host' code:
  heterogeous computing

# Parallel processing

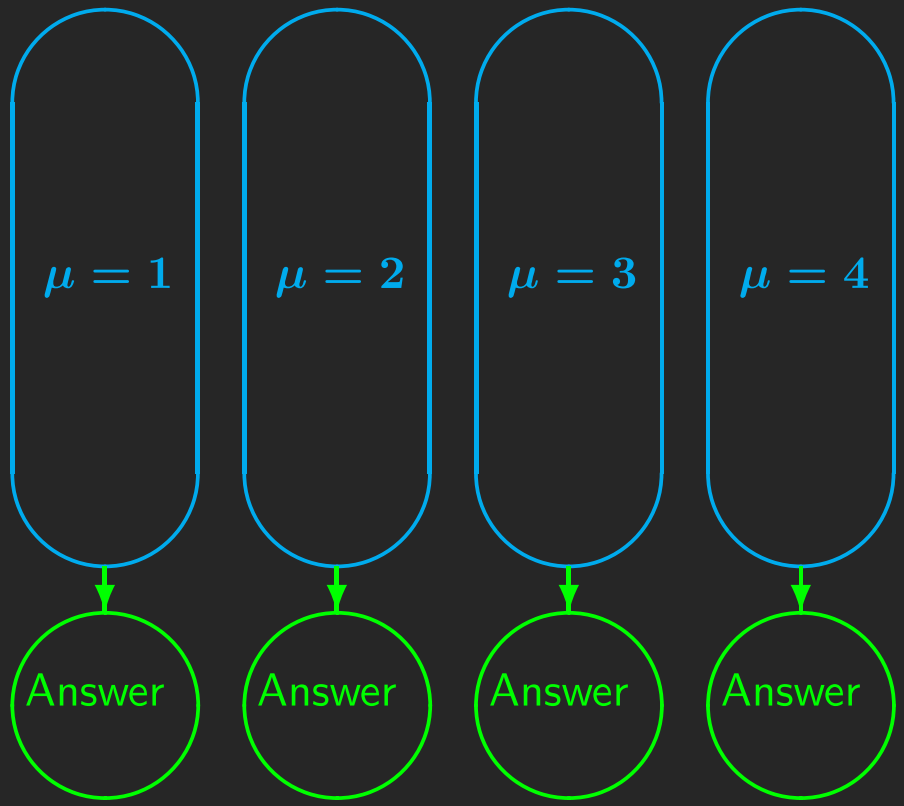
# Concurrency

- Must have something to do for all these cores.
- Find parts of the program that can done independently, and therefore concurrently.
- There must be many such parts.
- Their order of execution should not matter either.
- Data dependencies limit concurrency.



# Parameter "sweep": best case scenario

- Aim is to get results from a model as a parameter varies.
- Can run the serial program on each processor at the same time.
- Get "more" done.

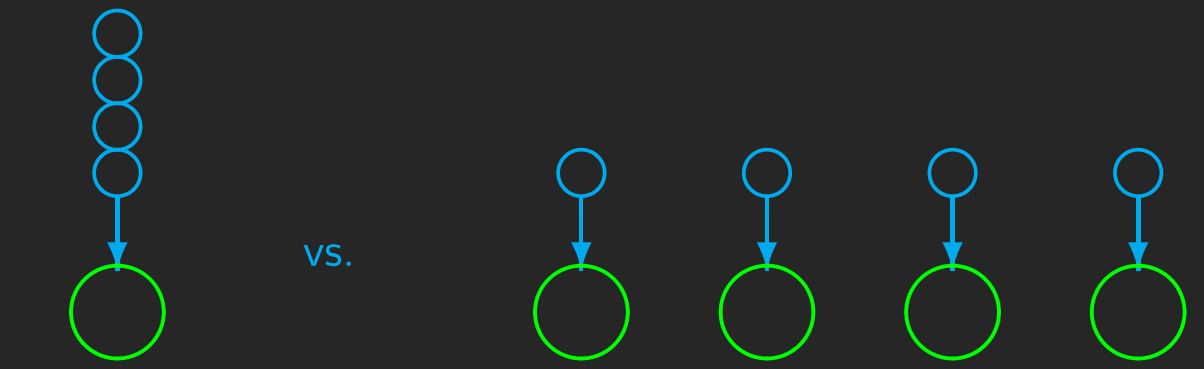


# Throughput

- How many tasks can you do per unit time?

$$\text{throughput} = H = \frac{N}{T}$$

- Maximizing $H$ means that you can do as much as possible.
- Independent tasks: using $P$ processors increases $H$ by a factor $P$




$T = NT_1$
$H = 1/T_1$

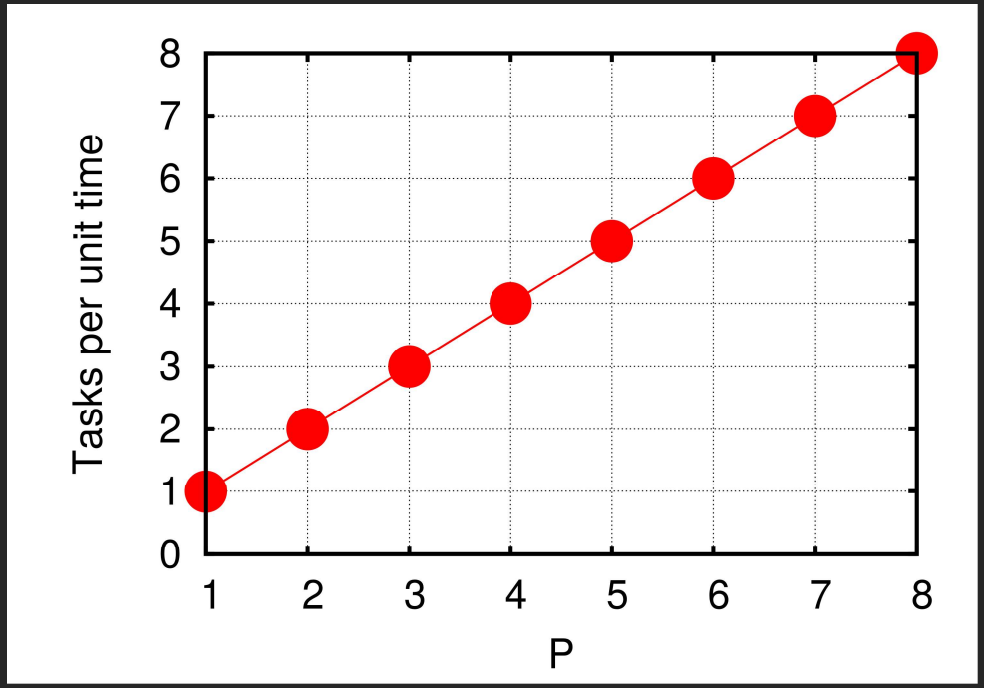
$T = NT_1/P$
$H = P/T_1$

# Scaling — Throughput

- How a problem's throughput scales as processor number increases ("strong scaling").
- In this case, linear scaling:

$$H \propto P$$

- This is Perfect scaling.

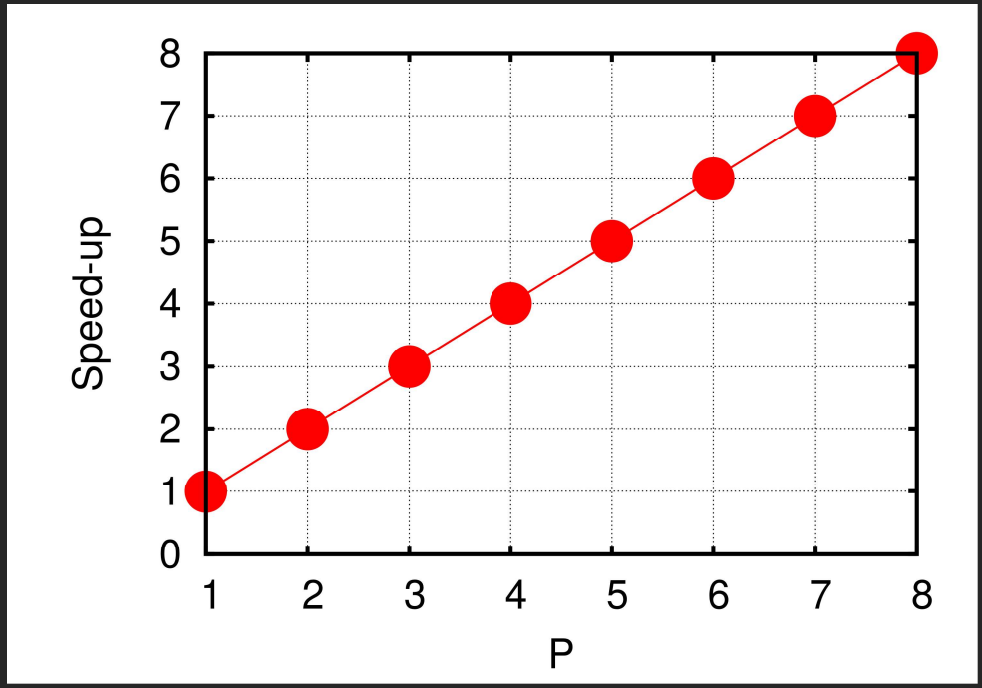


# Scaling – Speedup

- How much faster the problem is solved as processor number increases.
- Measured by the serial time divided by the parallel time

$$S = \frac{T_{serial}}{T(P)} \propto P$$

- For embarrassingly parallel applications: Linear speed up.

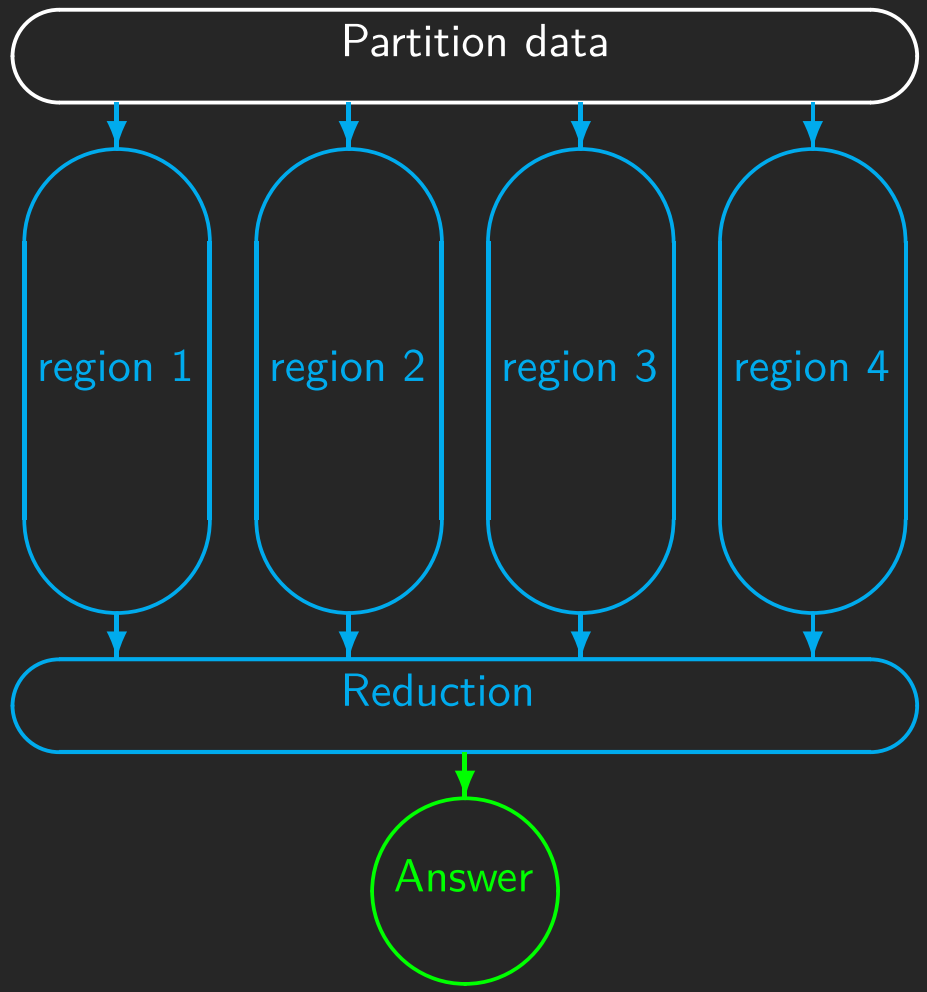


# Non-Ideal Parallel Computations

# Non-ideal case #1: Non-parallelizable algorithms

- Say we want to integrate some tabulated experimental data.
- Integration can be split up, so different regions are summed by each processor.
- Non-parallelizable parts of the algorithm:
  - First need to get data to processor
  - And at the end bring together all the sums: reduction

$T_s \equiv$ time for serial part (Partition+Reduction)
$T_p \equiv$ time for parallelizable part (for $P = 1$, so, the sum of all the regions on the right)



# Deriving Amdahl's law

Speed-up (without parallel overhead):

$$S = \frac{T_p + T_s}{T_p/P + T_s}$$

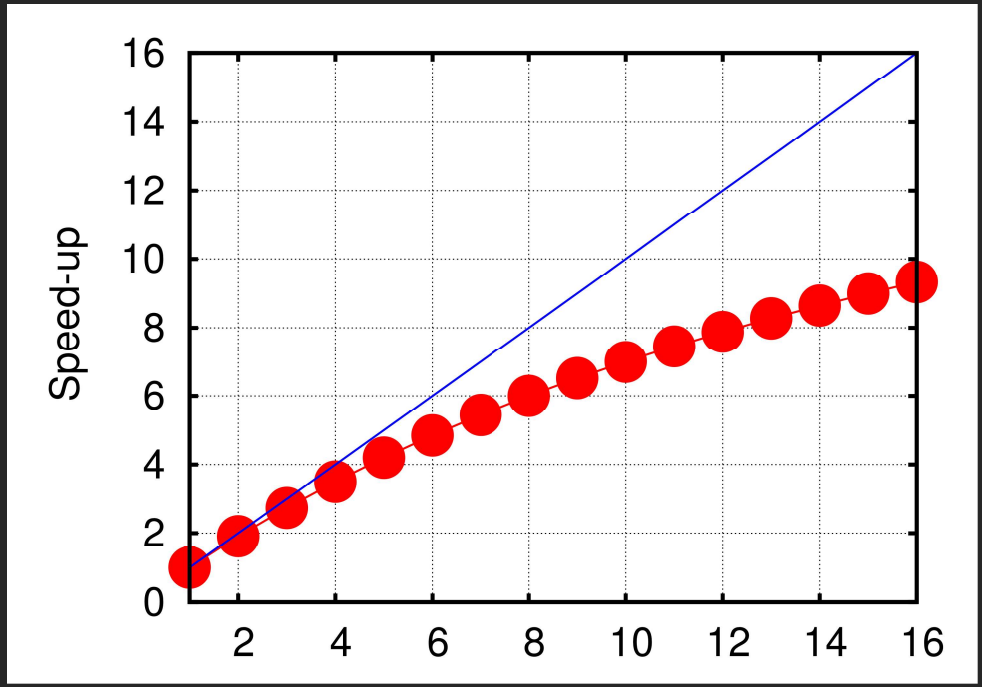
or, with $f \equiv T_s/(T_s + T_p)$ the serial fraction,

$$S = \frac{1}{f + (1-f)/P}$$

Note that

$$\lim_{P\to\infty} S = \frac{1}{f}$$

- Serial part dominates asymptotically.
- Speed-up limited, no matter size of $P$.




(example for $f = 5\%$)

# Non-ideal case #2: Non-locality

- Moving data around slows things down because communication is slower than computing.
- Not computing where the data resides or was generated, requires data movement and wastes time.
- Many memory and storage systems hide locality, using caches or pulling data automatically.
- To influence the locality, you need to change the data access pattern.

**Communication and data motion can rarely be completely avoided, but can be minimized.**

- Shared memory systems: having data processed by the core that generated it improves locality.
- Distributed systems: not having data in the process that needs it, means more communications.
- File systems and memory: accessing data contiguously helps.

  E.g. using many separately files is not contiguous, and also requires additional I/O operations.
- Reusing data helps.

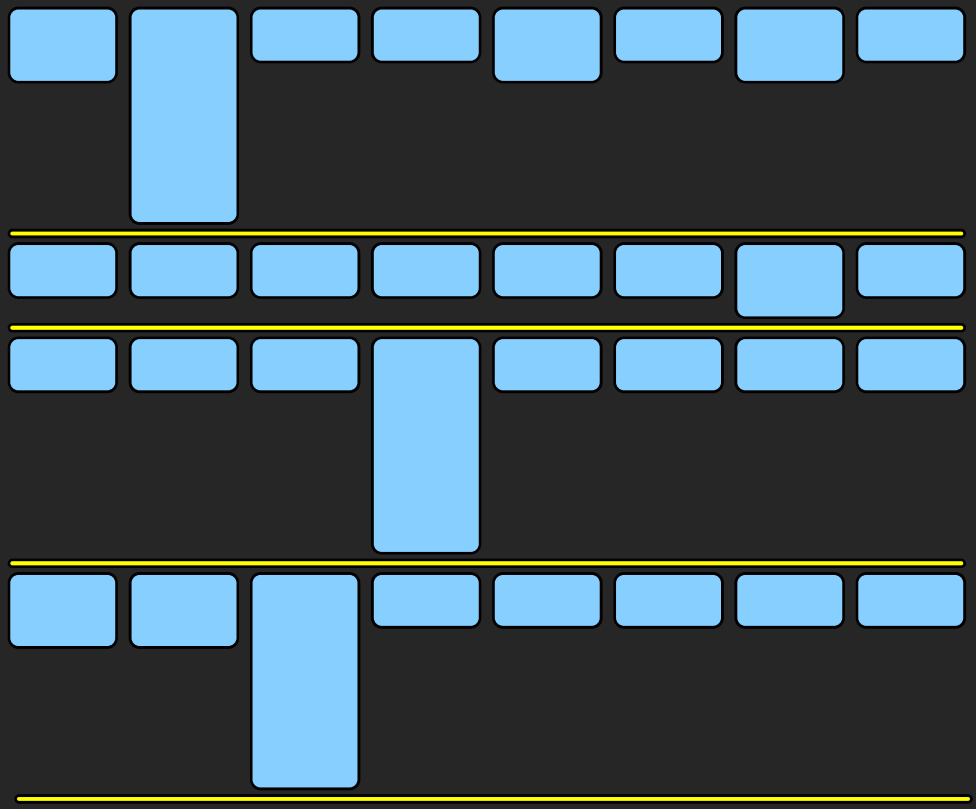
# Non-ideal case #3: Load imbalance

- Suppose you have 32 computations to do, and they are all independent.
- That would scale perfectly, but this time there is a catch:

  **The different computations takes very different times.**

  **And we can't know how long a computation will take before we run it.**
- Let's say we want to run these computations on 8 cores.

Easy, right? We'll just run 4 sets of 8!



# May I have a supercomputer, please?

- If you need a supercomputer, your computation has outgrown your computer or laptop.
- You (and very few people) can afford their own supercomputer: you will need to use a supercomputer that you share with potentially hundreds or thousands of other users.
- Due to the internet, resources can be used remotely, allowing for more sharing and larger systems.
- Sharing is good. Shared resources get better utilization.

  No cores need to wait e.g. because the code isn't ready or the new postdoc hasn't arrived. Someone else can use the computing time in the meantime.

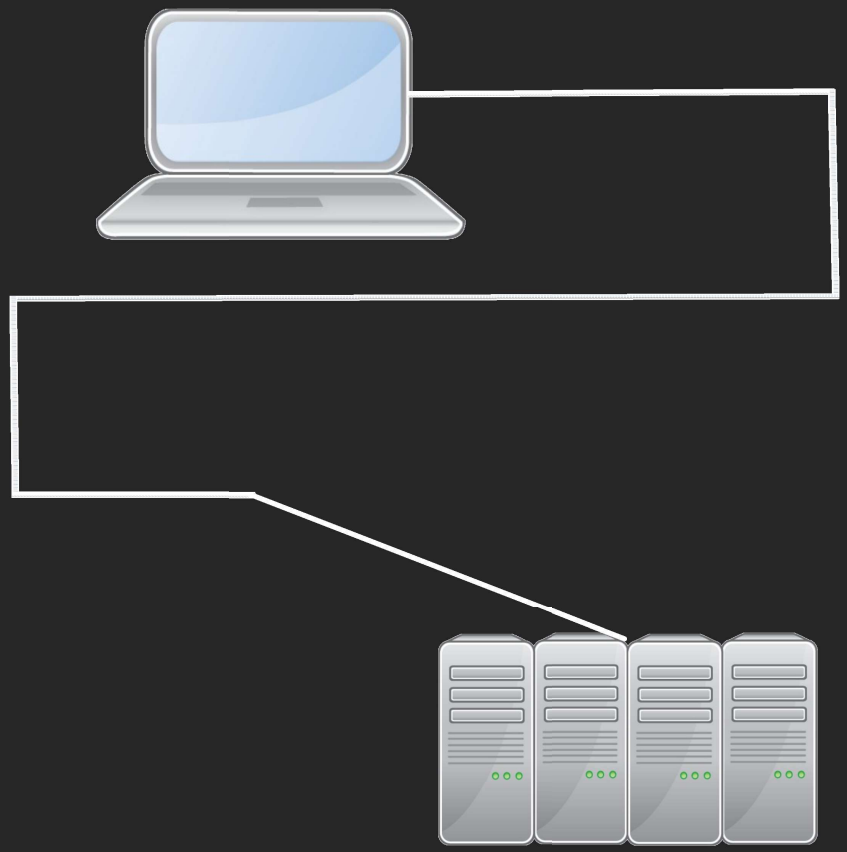
  *E.g., Niagara's utilization typically lies between 94% and 98%, with jobs always in the queue.*

**Fact**

Because supercomputers are remote and shared resources, these machines need to be used quite differently from how you use your own computer.

# It's Remote

- You're at your computer ("terminal")
- The supercomputer is in a data centre somewhere ("server").
- You must connect remotely using `ssh` ("secure shell").
- You must interact with the supercomputer using the command line.
- Yes, you read that right, ***the command line!***
- Yes, you read that right, ***the command line!***
- Nobody uses a GUI in HPC. Nobody.



# It's shared

- You're on the login node **together with all other folks** in this course.
- You have a home directory, called $HOME, and a scratch directory $SCRATCH (in fact, these are variables containing their true location).
- All other nodes of the Teach Cluster are compute nodes.
- To run on compute nodes, you need to create a job script that contains a request for specific resources for a specific time.
- You pass this job script to the scheduler using the sbatch command.
  The scheduler used on Teach, and on many other supercomputers, is called SLURM.
- The scheduler allocates compute resources to your job.

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --time=00:20:00
#SBATCH --output=sweep_bondbreak_output.txt

# This is the sweep_bondbreak.sh script

module load python # bondbreak needs python

# Run multiple cases with different random seeds
X=1.2      # initial bond extension
T=2.2      # temperature
DT=0.0003  # timestep
MT=400.0   # max.time to simulate
ODT=1.0    # interval at which write data
NREPS=96   # how many repeats
for ((S=1; S<=NREPS; S++)) ; do
 echo "Repetition $S/$NREPS"
 DFN=out/output-$T-$S.data
 LFN=out/output-$T-$S.log
 time ./bondbreak $X $T $DT $MT $S $DFN $ODT $LFN
done

# Extract the breakage times from the logs
awk '/BREAKAGE DETECTED/{print $8}' out/*.log
```

⟵ First line makes this a bash shell script

⟵ #SBATCH lines request 1 core for 20 minutes

*The rest of the script is run on the compute node that the scheduler allocates for your job.*

⟵ Most software requires `module` commands

⟵ Setup up parameters

⟵ One way to loop in bash scripts

⟵ Construct filenames depending on `T` and `S`

⟵ Pass parameter to bondbreak app (also time how long each takes)

⟵ Collect breakage times (`awk` out of scope)

# Why a scheduler?

The compute nodes/cores need to be fairly shared among all users.

- You can't just reserve cores for particular users, or at least some of them wouldn't be utilized all the time (which is a waste, as other users could have used them).
- So instead of having fixed reservations, users must submit jobs.
- Each job must specify the resources it needs (time/cpus/gpus).
- A program called the scheduler takes those resource requests and finds a time slot and (set of) compute node(s) to allocate for the job.

On a busy system, the allocated time is usually in the future, and often unknown.

**I.e., you have to wait.**

*Scheduling for a whole cluster is hard and takes time, therefore there are limits to how many jobs you can submit as well as a minimum size. If you have many small jobs to do, bunch them up and use GNU Parallel.*

# Using the Scheduler

There are different schedulers, but the SciNet clusters use SLURM (as do all Compute Canada clusters).

Some of the most common parameters are:

| | | |
|---|---|---|
| `-t` | `--time` | amount of time |
| `-N` | `--nodes` | number of nodes |
| `-n` | `--ntasks` | number of tasks |
| | `--ntasks-per-node` | number of tasks per node |
| `-c` | `--cpus-per-task` | number of threads per task |
| | `--gres=...` | special requests, e.g. GPUs |
| | `--mem=...` | amount of memory |


Commands to interact with the scheduler

| | |
|---|---|
| `sbatch` | submit job |
| `squeue` | see queued jobs and their status |
| `scancel` | cancel a job |
| `seff` | see job stats after completion |
| `salloc/debugjob` | get short interactive job on a compute node |