

Intro to Parallel Computing

Ramses van Zon

PHY1610 Winter 2026



Motivation for Parallel Computing



Why is Parallel Computing necessary?

- **Big Data:**

Modern experiments and observations yield vastly more data to be processed than in the past.

- **Big Science:**

As more computing resources become available, the bar for cutting edge simulations is raised.

- **New Science:**

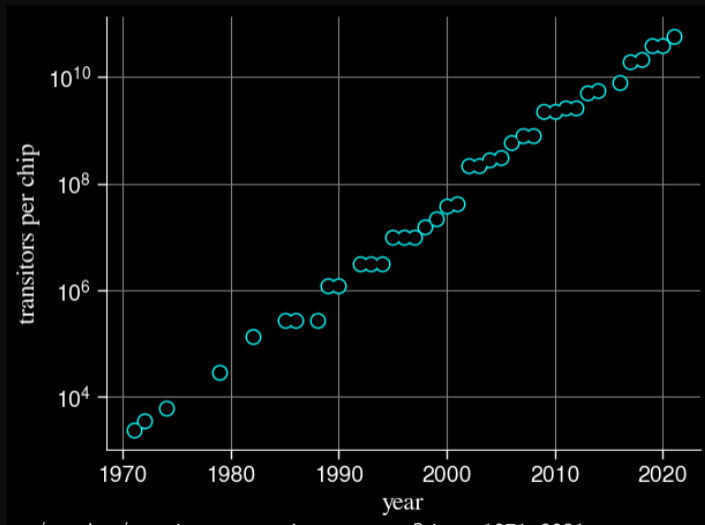
which before could not even be done, now becomes reachable.

However:

- Advances in processor clock speeds, bigger and faster memory and disks have been lagging as compared to fifteen years ago. We can no longer “just wait a year” and get a better computer.
- So more computing resources here means: more cores running *concurrently*.
- Even most laptops now have 2 cpu cores or more.
- So parallel computing is necessary.



Wait, what about Moore's Law?



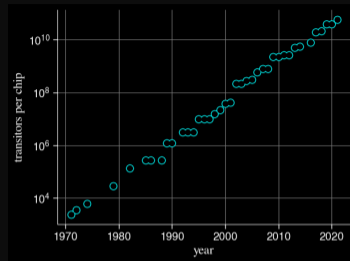
<https://ourworldindata.org/grapher/transistors-per-microprocessor?time=1971..2021>

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.

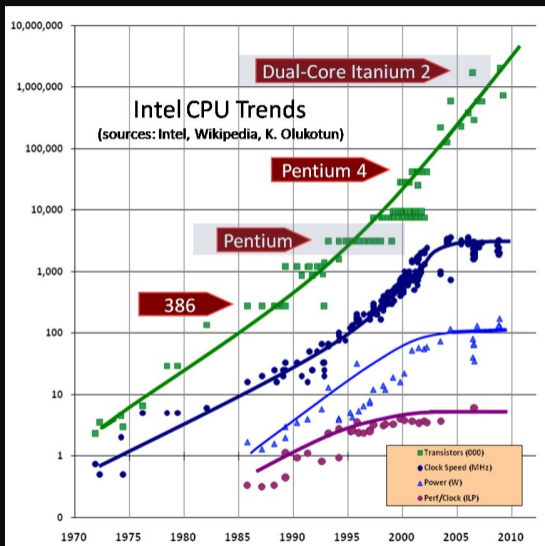
(source: Moore's law, wikipedia)



But...

- Moore's Law doesn't promise us increasing clock speed.
- We've gotten more transistors but it's getting hard to push clock-speed up. Power density is the limiting factor.
- So we've gotten more cores at a fixed clock speed.
- (Also, it is physically reaching its limits)

Wait, what about Moore's Law?



The plot on the left shows not just the number of transistors, which follows Moore's law, but also how clock speeds and power demands have grown.

All curves flatten except the transistor count.

This shows that the continuation of Moore's law is due to the presence of multiple cores, which require parallel programming.

(source: www.extremetech.com)

Concurrency

- All these cores need something to do.
- We need to find parts of the program that can be done independently, and therefore on different cores concurrently.
- We would like there to be many such parts.
- Ideally, the order of execution should not matter either.
- However, data dependencies limit concurrency.



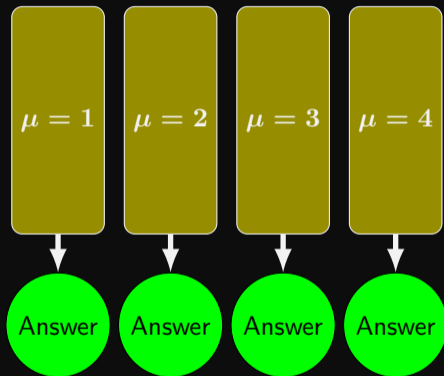
(source: <http://flickr.com/photos/splorp>)

Parallel computing



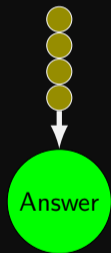
Parameter study: best case scenario

- Suppose the aim is to get results from a model as a parameter varies.
- We can run the serial program on each processor at the same time.
- Thus we get 'more' done.



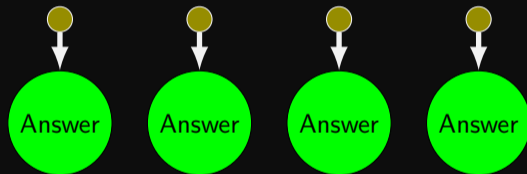
Throughput

- How many tasks can you do per unit time? **throughput** = $H = \frac{N}{T}$
 N is the number of tasks, T is the total time.
- Maximizing H means that you can do as much as possible.
- Independent tasks: using P processors increases H by a factor of P .



$$T = NT_1$$

$$H = 1/T_1$$



$$T = NT_1/P$$

$$H = P/T_1$$

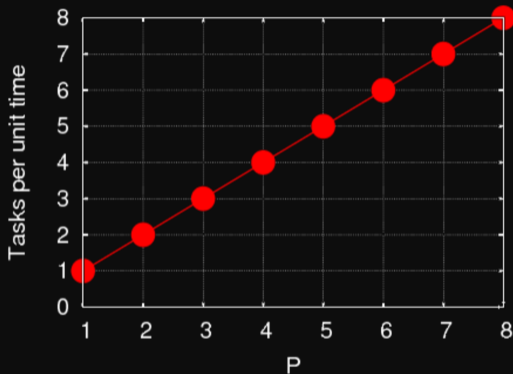
Scaling

Scaling: Throughput

- How a given problem's throughput scales as processor number increases is called **strong scaling**
- In the previous case, linear scaling:

$$H \propto P$$

- This is perfect scaling. These are called “embarrassingly parallel” calculations.

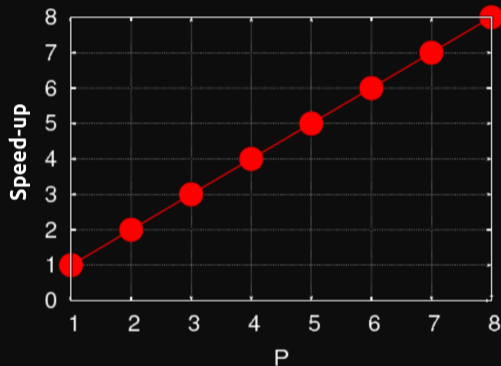


Scaling: Speedup

- Speedup: how much faster the problem is solved as processor number increases.
- This is measured by the serial time divided by the parallel time

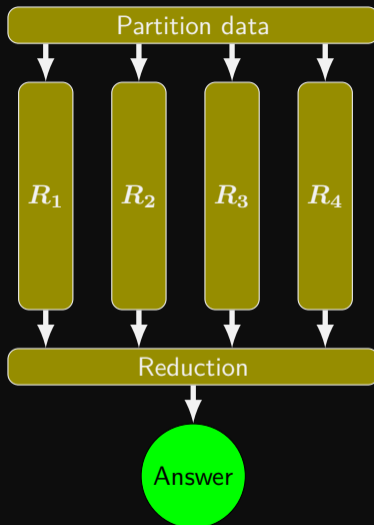
$$S = \frac{T_{\text{serial}}}{T(P)}$$

- For embarrassingly parallel applications, $S \propto P$: linear speed up.

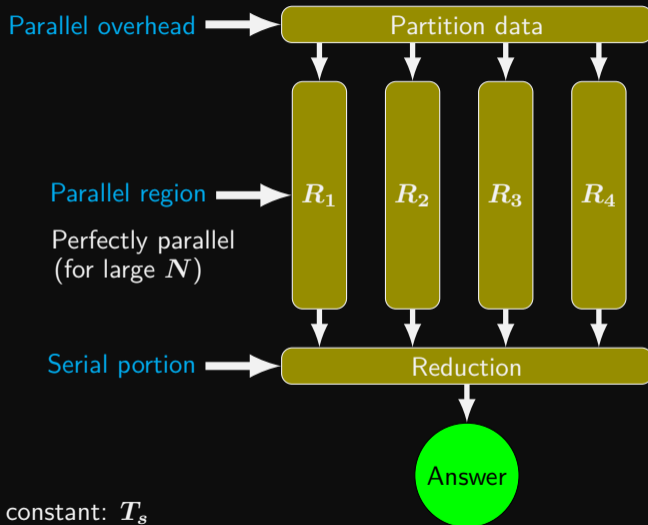


Non-ideal cases

- Say we want to integrate some tabulated experimental data.
- Integration can be split up, so different regions are summed by each processor.
- Non-ideal:
 - ▶ We first need to get data to each processor.
 - ▶ At the end we need to bring together all the sums: *reduction*.



Non-ideal cases



Suppose non-parallel part is constant: T_s

Amdahl's law

Amdahl's law

Speed-up (without parallel overhead):

$$S = \frac{T_{\text{serial}}}{T(P)} = \frac{NT_1 + T_s}{\frac{NT_1}{P} + T_s}$$

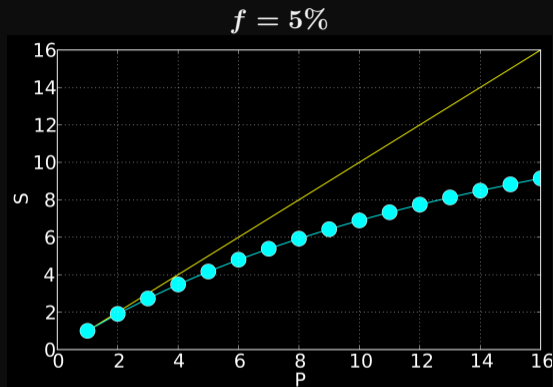
or, calling $f = T_s / (T_s + NT_1)$ the **serial fraction**,

$$S = \frac{1}{f + (1 - f)/P} \quad \xrightarrow{P \rightarrow \infty} \quad \frac{1}{f}$$

The serial part dominates asymptotically.

The speed-up is limited, no matter what size of P .

Aim to structure your program to minimize the serial portions of the code!



Scaling efficiency

Speed-up compared to ideal factor P :

$$\text{Efficiency} = \frac{S}{P}$$

This will invariably fall off for larger P , except for embarrassingly parallel problems.

$$\text{Efficiency} \sim \frac{1}{fP} \xrightarrow{P \rightarrow \infty} 0$$

You cannot get 100% efficiency in any non-trivial problem. \[0.3cm]

All you can aim for here is to make the efficiency as high as possible.

Hardware Architectures



Supercomputer architectures

Supercomputer architectures comes in a number of different types:

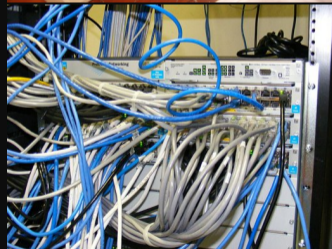
- Clusters, or distributed-memory machines, are in essence a bunch of desktops linked together by a network (“interconnect”). Easy and cheap.
- Multi-core machines, or shared-memory machines, are a collection of processors that can see and use the same memory. Limited number of cores, and much more expensive when the machine is large.
- Accelerator machines, are machines which contain an “off-host” accelerator, such as a GPGPU or Xeon Phi, that is used for computation. Quite fast, but complicated to program.
- Vector machines were the early supercomputers. Very expensive, especially at scale. These days most chips have some low-level vectorization, but you rarely need to worry about it.

Most supercomputers are a hybrid combo of these different architectures.

Distributed Memory: Clusters

Clusters are the simplest type of parallel computer to build:

- Take existing powerful standalone computers,
- and network them.
- Easy to build and easy to expand.
- SciNet's Trillium supercomputer and the Teach cluster are examples.



(source: <http://flickr.com/photos/eurleif>)

Compute Resources at SciNet

Teach Cluster



Number of nodes: 64
Network: EDR Infiniband
RAM/node: 202 GB
Cores/node: 40

Trillium



Number of nodes: 1224
Network: NDR Infiniband
RAM/node: 768 GB
Cores/node: 192

Trillium-GPU



Number of nodes: 63
Network: NDR Infiniband
RAM/node: 768 GB
Cores/node: 96
GPUs/node: 4 H100

Balam



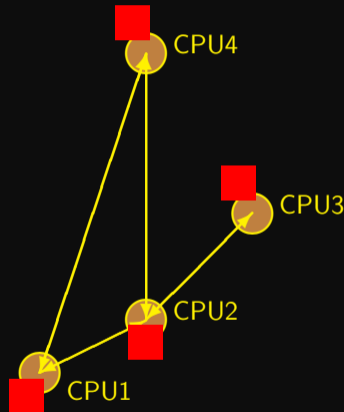
Number of nodes: 10
Network: Infiniband
RAM/node: 1 TB
Cores/node: 64
GPUs/node: 4 A100

*For use by the
'Acceleration
Consortium*



Distributed Memory: Clusters

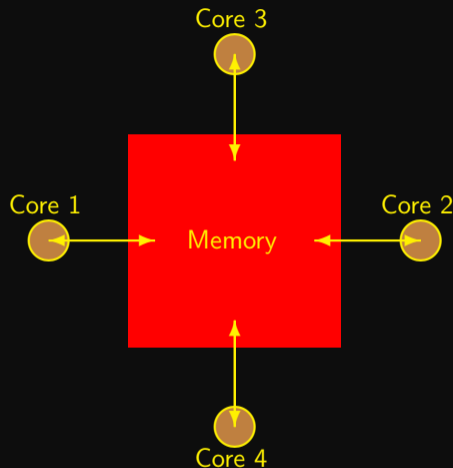
- Each processor is independent! Programs run on separate processors, communicating with each other when necessary. Each processor has its own memory! Whenever it needs data from another processor, that processor needs to send it.
- All communication must be hand-coded:~harder to program.
- MPI programming is used in this scenario.



Shared memory

Shared Memory

- Different processors acting on one large bank of memory. All processors “see” the same data.
- All coordination/communication is done through memory.
- Each core is assigned a thread of execution of a single program that acts on the data.
- Your desktop uses this architecture, if it’s multi-core.
- Can also use hyper-threading: assigning more than one thread to a given core.
- OpenMP is used in this scenario.



Threads versus Processes

Threads Threads of execution within one process, with access to the same memory etc.

Processes Independent tasks with their own memory and resources

```
ljdursi@gpc-f1
File Edit View Terminal Tabs Help
top - 17:27:34 up 2 days, 1:40, 1 user, load average: 1.81, 0.56, 0.20
Tasks: 142 total, 3 running, 139 sleeping, 0 stopped, 0 zombie
Cpu(s): 95.9%us, 3.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.1%hi, 1.0%si, 0.0%st
Mem: 16411872k total, 2778368k used, 13633504k free, 256k buffers
Swap: 0k total, 0k used, 0k free, 2265652k cached

  PID USER      PR  NI  VIRT  RES  SHR  S %CPU %MEM    TIME+  COMMAND
 18121 ljdursi   25   0 89536 1076 840  R 779.0  0.0    0:29.01 diffusion-omp
 17193 root      15   0 35300 2580 60   S 15.0  0.0    0:01.57 pbs_mom
 17192 root      15   0 35300 3216 696  R 6.0  0.0    0:00.48 pbs_mom
    1 root      15   0 10344 740 612  S 0.0  0.0    0:01.45 init
    2 root      RT  -5   0   0   0   0   S 0.0  0.0    0:00.00 migration/0
    3 root      34  19   0   0   0   S 0.0  0.0    0:00.00 ksoftirqd/0
    4 root      RT  -5   0   0   0   0   S 0.0  0.0    0:00.00 watchdog/0
    5 root      RT  -5   0   0   0   0   S 0.0  0.0    0:00.01 migration/1
    6 root      34  19   0   0   0   S 0.0  0.0    0:00.01 ksoftirqd/1

ljdursi@gpc-f1
File Edit View Terminal Tabs Help
top - 17:33:58 up 2 days, 1:47, 1 user, load average: 0.80, 0.31, 0.17
Tasks: 150 total, 9 running, 141 sleeping, 0 stopped, 0 zombie
Cpu(s):100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 16411872k total, 2801172k used, 13610700k free, 256k buffers
Swap: 0k total, 0k used, 0k free, 2268568k cached

  PID USER      PR  NI  VIRT  RES  SHR  S %CPU %MEM    TIME+  COMMAND
 18393 ljdursi   25   0 187m 5504 3484  R 100.2  0.0    0:05.45 diffusion-mpi
 18395 ljdursi   25   0 187m 5512 3492  R 100.2  0.0    0:05.46 diffusion-mpi
 18397 ljdursi   25   0 187m 5508 3488  R 100.2  0.0    0:05.46 diffusion-mpi
 18392 ljdursi   25   0 187m 5580 3556  R 99.9  0.0    0:05.40 diffusion-mpi
 18394 ljdursi   25   0 187m 5504 3488  R 99.9  0.0    0:05.45 diffusion-mpi
 18396 ljdursi   25   0 187m 5512 3492  R 99.9  0.0    0:05.45 diffusion-mpi
 18398 ljdursi   25   0 187m 5500 3480  R 99.9  0.0    0:05.43 diffusion-mpi
 18399 ljdursi   25   0 187m 5512 3492  R 99.9  0.0    0:05.46 diffusion-mpi
    1 root      15   0 10344 740 612  S 0.0  0.0    0:01.45 init
```

Share memory communication cost

Network	Latency	Bandwidth
Gigabit Ethernet	$10\mu\text{s}$ (10,000 ns)	1 Gb/s (60 ns/double)
Infiniband	$2\mu\text{s}$ (2,000 ns)	2-10 Gb/s (10 ns/double)
NUMA (shared memory)	$0.1\mu\text{s}$ (100 ns)	10-20 Gb/s (4 ns/double)

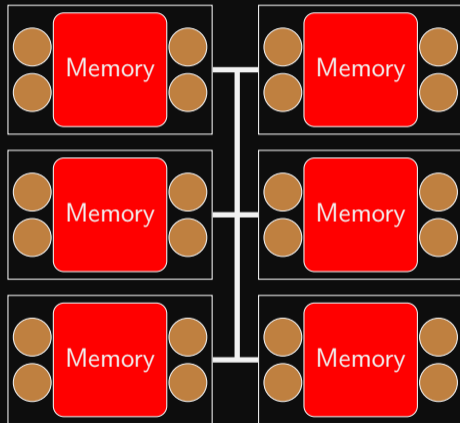
Processor speed: $\mathcal{O}(\text{GFlop}) \sim$ a few ns or less.

Communication is always the slowest part of your calculation!

Hybrid systems

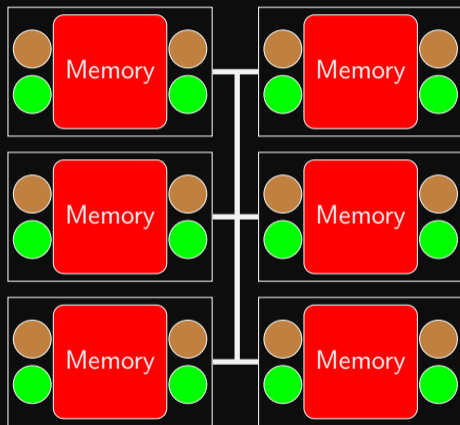
Hybrid architectures

- Multicore nodes linked together with an (high-speed interconnect).
- Many cores have modest vector capabilities.
- Teacup cluster has sixteen cores, and 64 GB of memory, per node.
- Niagara has forty cores, and 202 GB of memory, per node.
- OpenMP + MPI can be used in this scenario.



Hybrid architectures: accelerators

- Multicore nodes linked together with an (high-speed) interconnect.
- Nodes also contain one or more accelerators, e.g. GPUs.
- These are specialized, super-threaded (500-2000+) processors.
- Specialized programming languages, CUDA and OpenCL, are used to program these devices.
- Can be combined with MPI and OpenMP.



Using Supercomputers



Choosing your programming approach

The programming approach you use depends on the type of problem you have, and the type of machine that you will be using:

- Embarrassingly parallel applications: scripting, GNU Parallel¹.
- Shared memory machine: OpenMP, p-threads.
- Distributed memory machine: MPI, PGAS (UPC, Coarray Fortran).
- Graphics computing: CUDA, OpenACC, OpenCL.
- Hybrid combinations.

¹O. Tange (2018): GNU Parallel 2018, March 2018, <https://doi.org/10.5281/zenodo.1146014>.

Using the scheduler

- When you log in, you are on a **login nodes**.
- These are shared.
- You can edit, compile and run quick tests there.
- But the real compute with dedicated resources has to be done on **compute nodes**.

Once you have compiled and tested your code or workflow on a login node, and confirmed that it behaves correctly, you are ready to submit jobs to the cluster.

You submit jobs from a login node to the scheduler queue by passing a script to the sbatch command:

```
teach-login01:~$ sbatch jobscript.sh
```

- The scheduler will run your jobs will run on some of Teach 8 compute nodes.
- When and where your job runs is determined by the scheduler.
- Teach uses SLURM as its job scheduler.



Example

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --time=3:00:00
#SBATCH --job-name serialjobs
#SBATCH --output=serial_output_%j.txt
#SBATCH --mail-type=FAIL

module load gcc/14

parallel -j 4 ./myapp ::: {0..99}
```

```
teach-login01:~$ sbatch serialjobs.sh
```

- First line indicates that this is a bash script.
- Lines starting with #SBATCH go to SLURM.
- `sbatch` reads these lines as a job request
- In this case, SLURM looks for 4 cores to be put on the same one node to be run for 3 hours.
- Once it found such a node, script is run:
 - ▶ Loads modules
 - ▶ Has gnu-parallel load-balance 99 tasks over 4 cores.

Using the Scheduler

Some of the most common sbatch 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
-G	--gpus-per-node	number of gpus per node
	--mem	amount of memory
-A	--account	scheduler account to use
	--reservation	use reserved nodes



Using the Scheduler

Commands to interact with the scheduler

<code>sbatch</code>	submit job
<code>squeue</code>	see queued jobs and their status
<code>scancel</code>	cancel a job
<code>salloc</code>	get short interactive job on a compute node
<code>debugjob -n P</code>	get short interactive job on a compute node

