PHY1610H - Scientific Computing: Introduction to Parallel Programming

Ramses van Zon and Marcelo Ponce

SciNet HPC Consortium/Physics Department
University of Toronto

March 2021

Lecture 17: Today's class

Today we will discuss the following topics:

- Motivation for parallel programming.
- Limits to parallel programming.
- Supercomputing achitectures.
- Different parallel programming approaches.

Why is HPC 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 (SciNet), the bar for cutting edge simulations is raised.
- **New Science:** which before could not even be done, now becomes reachable.

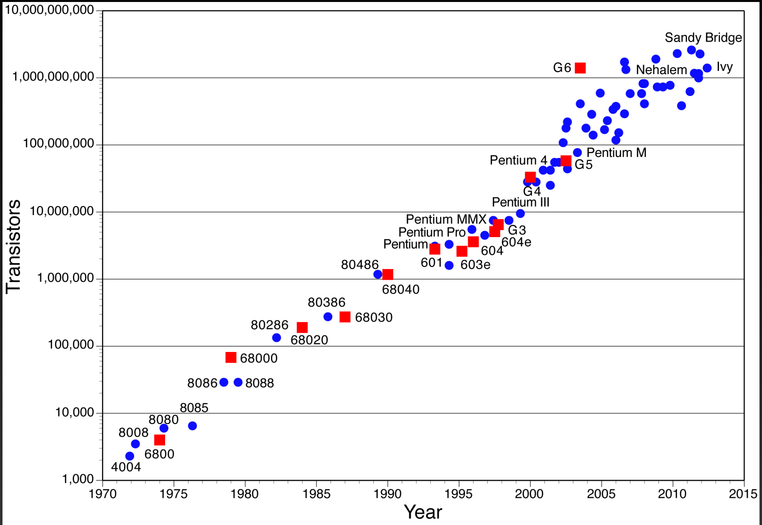
However:

- Advances in processor clock speeds, bigger and faster memory and disks have been lagging as compared to ten 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 or more cpus.
- So parallel computing is necessary.

Why Parallel Programming?

- **Faster**
  There's a limit to how fast one computer can compute.
- **Bigger**
  There's a limit to how much memory, disk, *etc.*, can be put on one computer.
- **More**
  We want to do the same thing that was done on one computer, but *thousands of times*.
- So use more computers!

Wait, what about Moore's Law?



(source: www.overlock.net)

Wait, what about Moore's Law?

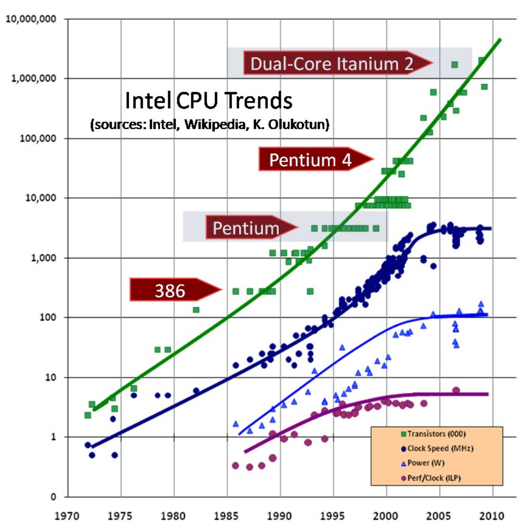
Moore's Law:

> *. . . 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 didn'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.

Wait, what about Moore's Law?



(source: www.extremetech.com)

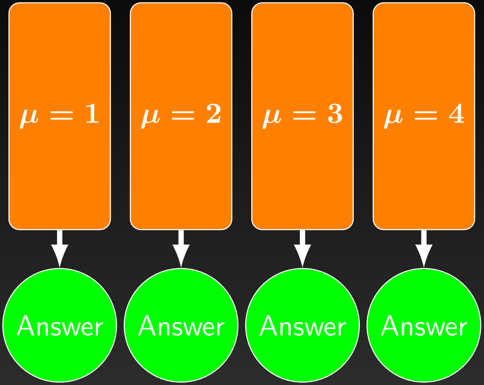
Concurrency

- All these cores need something to do.
- We need to find parts of the program that can 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)

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.



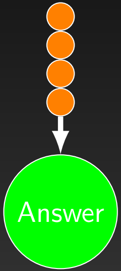
Throughput

- How many tasks can you do per unit time?

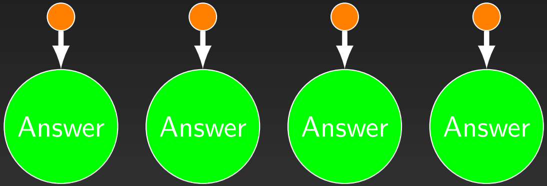
$$\text{throughput} = H = \frac{N}{T}$$

  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.



$$\begin{aligned} T &= NT_1 \\ H &= 1/T_1 \end{aligned}$$



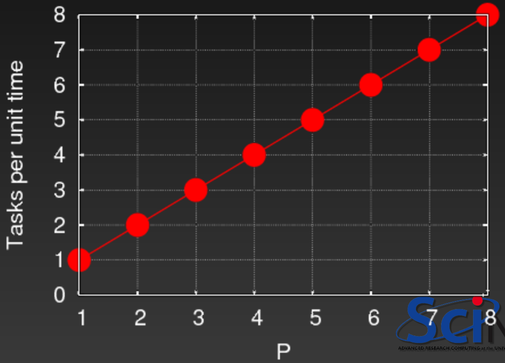
$$\begin{aligned} T &= NT_1/P \\ H &= P/T_1 \end{aligned}$$

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" jobs.

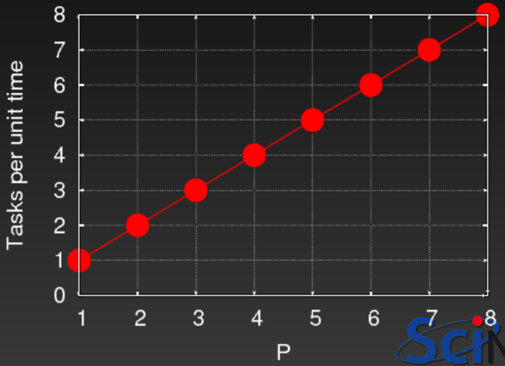


Scaling — Speedup

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

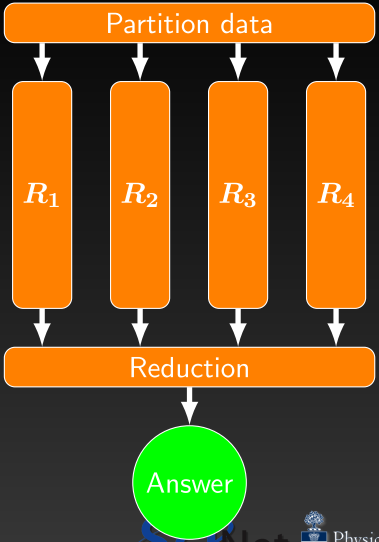
$$S = \frac{T_{\text{serial}}}{T(P)}$$

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

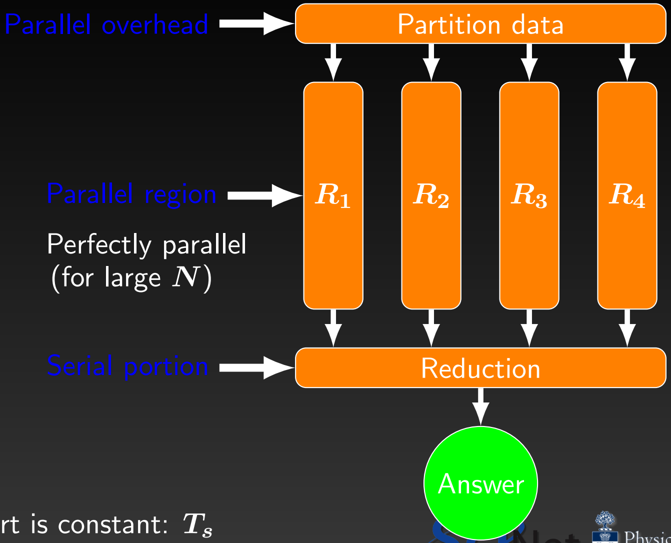


Non-ideal cases

- 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



Parallel overhead

Partition data

R_1 R_2 R_3 R_4

Parallel region

Perfectly parallel
(for large N)

Serial portion

Reduction

Answer

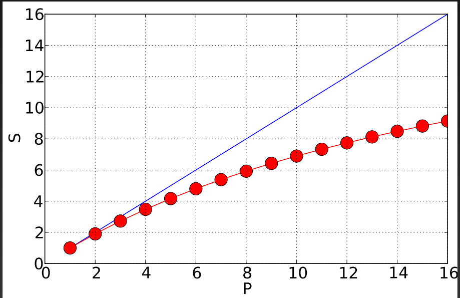
Suppose non-parallel part is constant: T_s

Amdahl's law

Speed-up (without parallel overhead): $S = \dfrac{T_{\text{serial}}}{T(P)} = \dfrac{NT_1 + T_s}{\frac{NT_1}{P} + T_s}$

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

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



The serial part dominates asymptotically. The speed-up is limited, no matter what size of P. $f = 5\%$ above.

Amdahl's law, example

An example of Amdahl's law:

- Suppose your code consists of a portion which is serial, and a portion that can be parallelized.
- Suppose further that, when run on a single processor,
  - the serial portion takes one hour to run.
  - the parallel porition takes nineteen hours to run.
- Even if you throw an infinite number of processors at the parallel part of the problem, the code will never run faster than 1 hour, since that is the amount of time the serial part needs to complete.

The goal is to structure your program to minimize the serial portions of the code.

Scaling efficiency

Speed-up compared to ideal factor P:

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

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

$$\textbf{Efficiency} \sim \frac{1}{fP} \xrightarrow{P\to\infty} 0$$

You cannot get 100% efficiency in any non-trivial problem.

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

Supercomputer architectures

Supercomputer architectures comes in a number of different types:

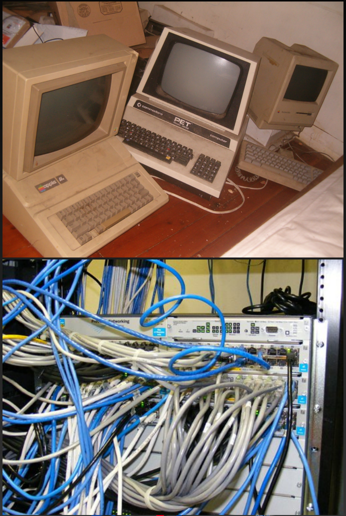
- Clusters, or distributed-memory machines, are in essence a bunch of desktops linked together by a network ("interconnect"). Easy and cheap.
- Multi-core machines, or shared-memory machines, are a collection of processors that can see and use the same memory. Limited number of cores, and much more expensive 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 Niagara supercomputer and the teach cluster are examples.



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

Compute Resources at SciNet

Teach Cluster (part of old GPC)

Number of nodes: 42

interconnect: Infiniband

RAM/node: 64 GB

Cores/node: 16

Niagara

Number of nodes: 1500 (60000 cores)

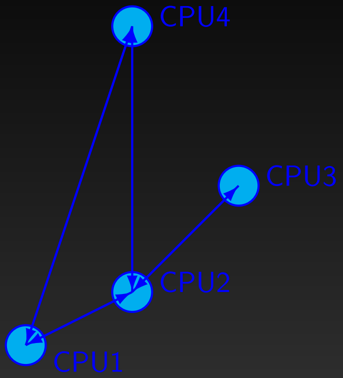
interconnect: Dragonfly+

RAM/node: 202GB

Cores/node: 40 (80 hyperthreads)

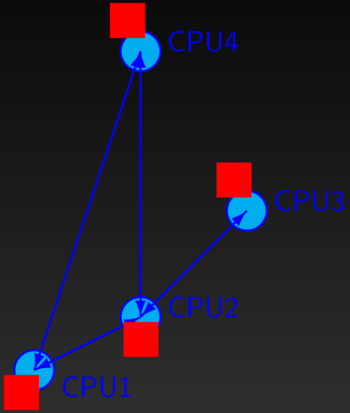
Distributed Memory: Clusters

- Each Processor is independent! Programs run on separate processors, communicating with each other when necessary.



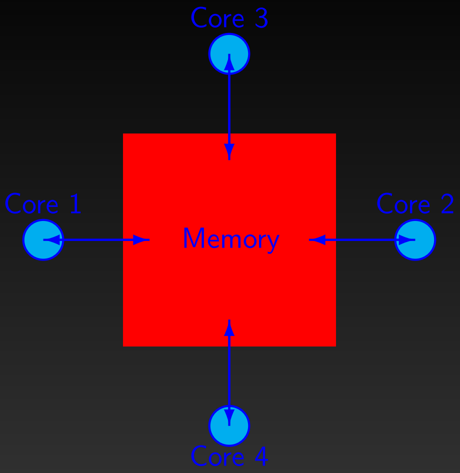
Distributed Memory: Clusters

- 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

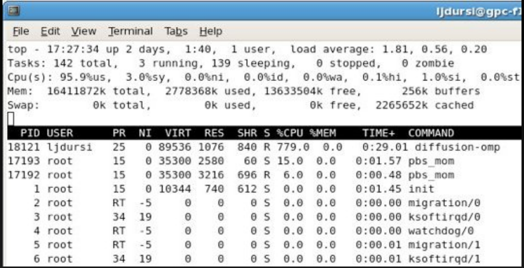
- Different processors acting on one large bank of memory. All processors 'see' the same data.
- All coordination/communication is done through memory.
- Each core is assigned a thread of execution of a single program that acts on the data.
- Your desktop uses this architecture, if it's multi-core.
-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



Threads versus Processes

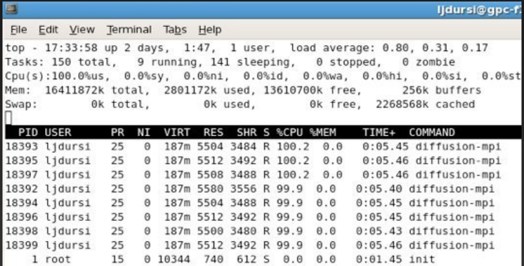
Threads: Threads of execution within one process, with access to the same memory etc.

Processes: Independent tasks with their own memory and resources



```
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 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 S  0.0  0.0   0:00.00 watchdog/0
    5 root      RT  -5     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
```



```
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

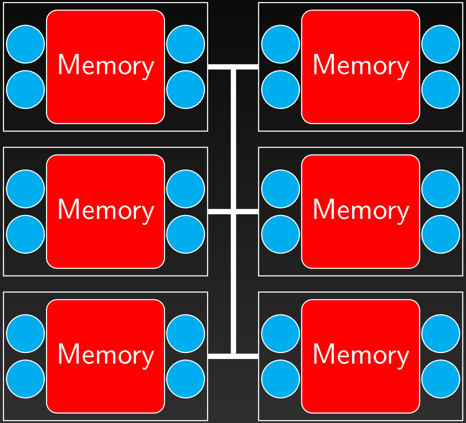
	Latency	Bandwidth
Gigabit Ethernet	10μs (10,000 ns)	1 Gb/s (60 ns/double)
Infiniband	2μs (2,000 ns)	2-10 Gb/s (10 ns/double)
NUMA (shared memory)	0.1μs (100 ns)	10-20 Gb/s (4 ns/double)

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

Communication is always the slowest part of your calculation!

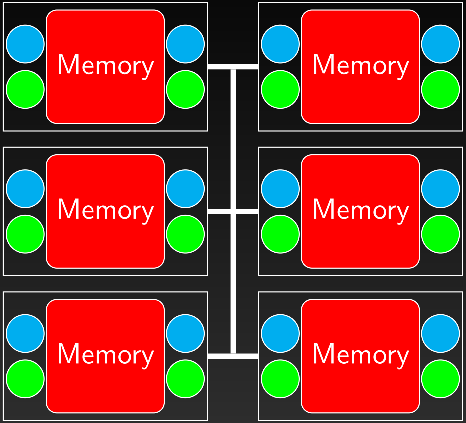
Hybrid architectures

- Multicore nodes linked together with an (high-speed) interconnect.
- Many cores have modest vector capabilities.
- Teach (old-GPC) 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, GPGPUs (General Purpose Graphics Processing Units) or Xeon Phis.
- These are specialized, super-threaded (500-2000+) processors.
- Specialized programming languages, CUDA and OpenCL, are used to program these devices.



- MPI and OpenMP can also be used in combination with accelerators.

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 Parallel[1].
- Shared memory machine: OpenMP, p-threads.
- Distributed memory machine: MPI, PGAS (UPC, Coarray Fortran).
- Graphics computing: CUDA, OpenACC, OpenCL.
- Hybrid combinations.

We focus on OpenMP, MPI programming in this course.

[1] O. Tange (2011): GNU Parallel - The Command-Line Power Tool, ;login; The USENIX Magazine, February 2011:42-47.

Data or computation bound?

The programming approach you should use also depends upon the type of problem that is being solved:

- Computation bound, requires task parallelism
  - Need to focus on parallel processes/threads.
  - These processes may have very different computations to do.
  - Bring the data to the computation.
- Data bound, requires data parallelism
  - There focus here is the operations on a large dataset.
  - The dataset is often an array, partitioned and tasks act on separate partitions.
  - Bring the computation to the data.

Granularity

The degree to which parallelizing your algorithm makes sense affects the approach used:

- Fine-grained (loop) parallelism
  - Smaller individual tasks.
  - The data is transferred among processors frequently.
  - Shared Memory Model, OpenMP.
  - Scale Limitations
- Coarse-grained (task) parallelism
  - Divide and conquer.
  - Data communicated infrequently, after large amounts of computation.
  - Distributed memory, MPI.

Too fine-grained → overhead issues.

Too coarse-grained → load imbalance issues.

The balance depends upon the architecture, access patterns and the computation.

Summary

- You need to learn parallel programming to truly use the hardware that you have at your disposal.
- The serial only portions of your code will truly reduce the effectiveness of the parallelism of your algorithm. Minimize them.
- There are many different hardware types available: distributed-memory cluster, shared-memory, hybrid.
- The programming approach you need to use depends on the nature of your problem.

References

- `https://docs.scinet.utoronto.ca/index.php/Teach`
- `https://docs.scinet.utoronto.ca/index.php/Niagara_Quickstart`