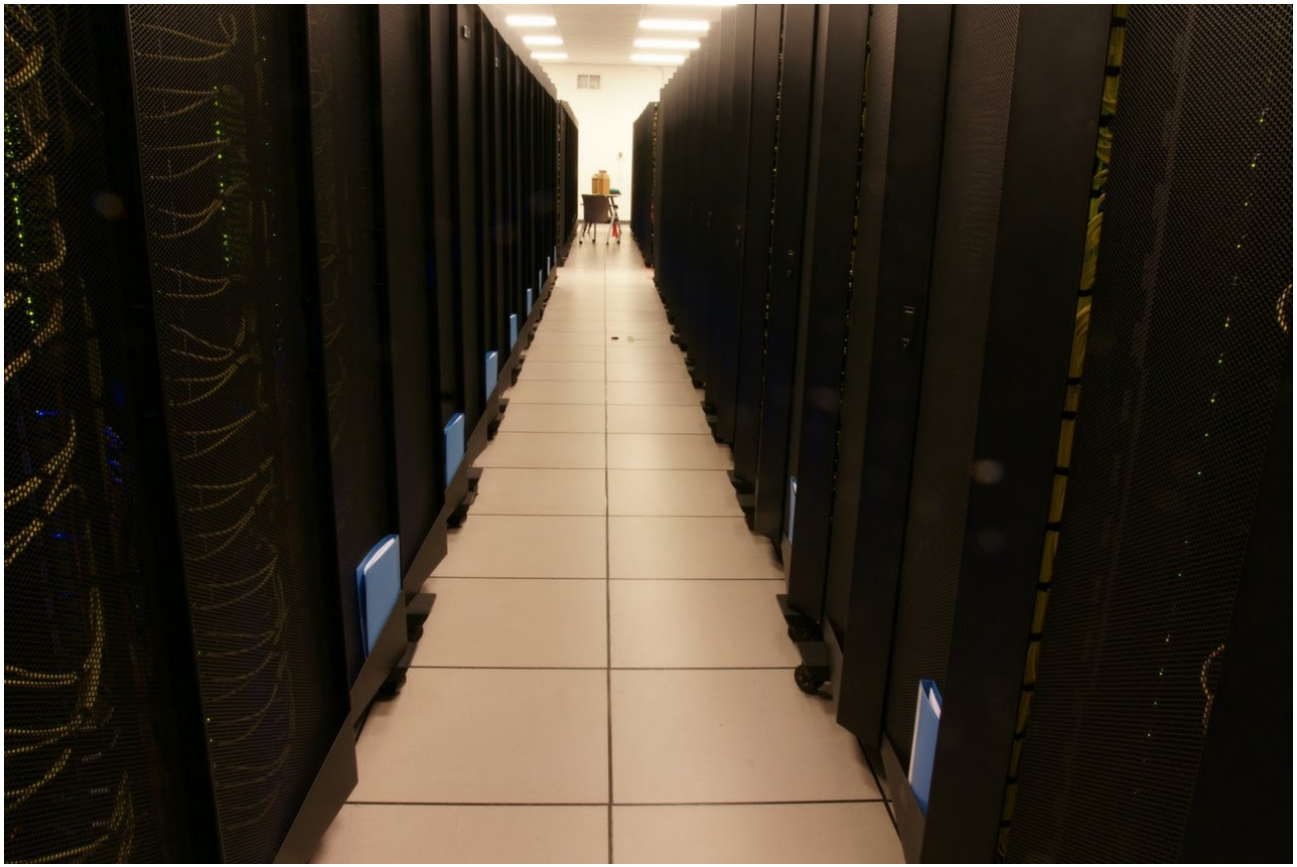
Practical Parallel Programming Intensive

SciNet HPC Consortium

9–13 May 2011

Welcome to the intensive parallel programming course!



SciNet compute • calcul CANADA

Part I

The Course

Main goal of the course

. . . to enable young researchers already experienced in scientific computing to leave with the knowledge necessary to begin writing the parallel codes needed for their research.

The course will be a mix of lectures and immediate feedback on practical assignments, designed to ensure that students leave with significant experience in both OpenMP and MPI, two of the standards for parallel computing today.

So there'll ne a lot of typing and programming to help build skills with OpenMP and MPI.

We will use C and Fortran. We'll assume that you already know one of them, but not both.

SciNet compute • calcul CANADA

Schedule

Mon May 9	AM	Intro to Parallel Computing, SciNet resources
	PM	OpenMP I *+hands on*
Tue May 10	AM	OpenMP II *+hands on*
	PM	MPI I *+hands on*
Wed May 11	AM	MPI II *+hands on*
	PM	Explicit PDEs: Hydrodynamics *+hands on*
Thu May 12	AM	Particle Methods: N-body *+hands on*
	PM	GPU Programming *+hands on*
Fri May 13	AM	Performance tools & Best practices

Strongly recommended books

(not provided by us)

1. B. Chapman, G. Jost and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming* (MIT Press, Cambridge 2008)
2. W. Gropp, E. Lusk, A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, second edition (MIT Press, Cambridge 1999).

SciNet compute • calcul CANADA

Part II

Introduction to Parallel Programming

Why Parallel Programming?

1. **Faster**
   There's a limit to how fast 1 computer can compute.

So use more computers!

Why Parallel Programming?

1. **Faster**
   There's a limit to how fast 1 computer can compute.
2. **Bigger**
   There's a limit to how much memory, disk, etc, can be put on 1 computer.

So use more computers!

SciNet compute • calcul CANADA

Why Parallel Programming?

1. **Faster**
   There's a limit to how fast 1 computer can compute.
2. **Bigger**
   There's a limit to how much memory, disk, etc, can be put on 1 computer.
3. **More**
   Want to do the same thing that was done on 1 computer, but *thousands of times*.

So use more computers!

SciNet compute • calcul CANADA

Why is it necessary?

- Modern experiments and observations yield vastly more data to be processsed than in the past.
- As more computing resources become available (SciNet), the bar for cutting edge simulations is raised.
- Science that before could not even be done becomes reachable.

Why is it necessary?

- Modern experiments and observations yield vastly more data to be processsed than in the past.
- As more computing resources become available (SciNet), the bar for cutting edge simulations is raised.
- Science that before could not even be done becomes reachable.

However:

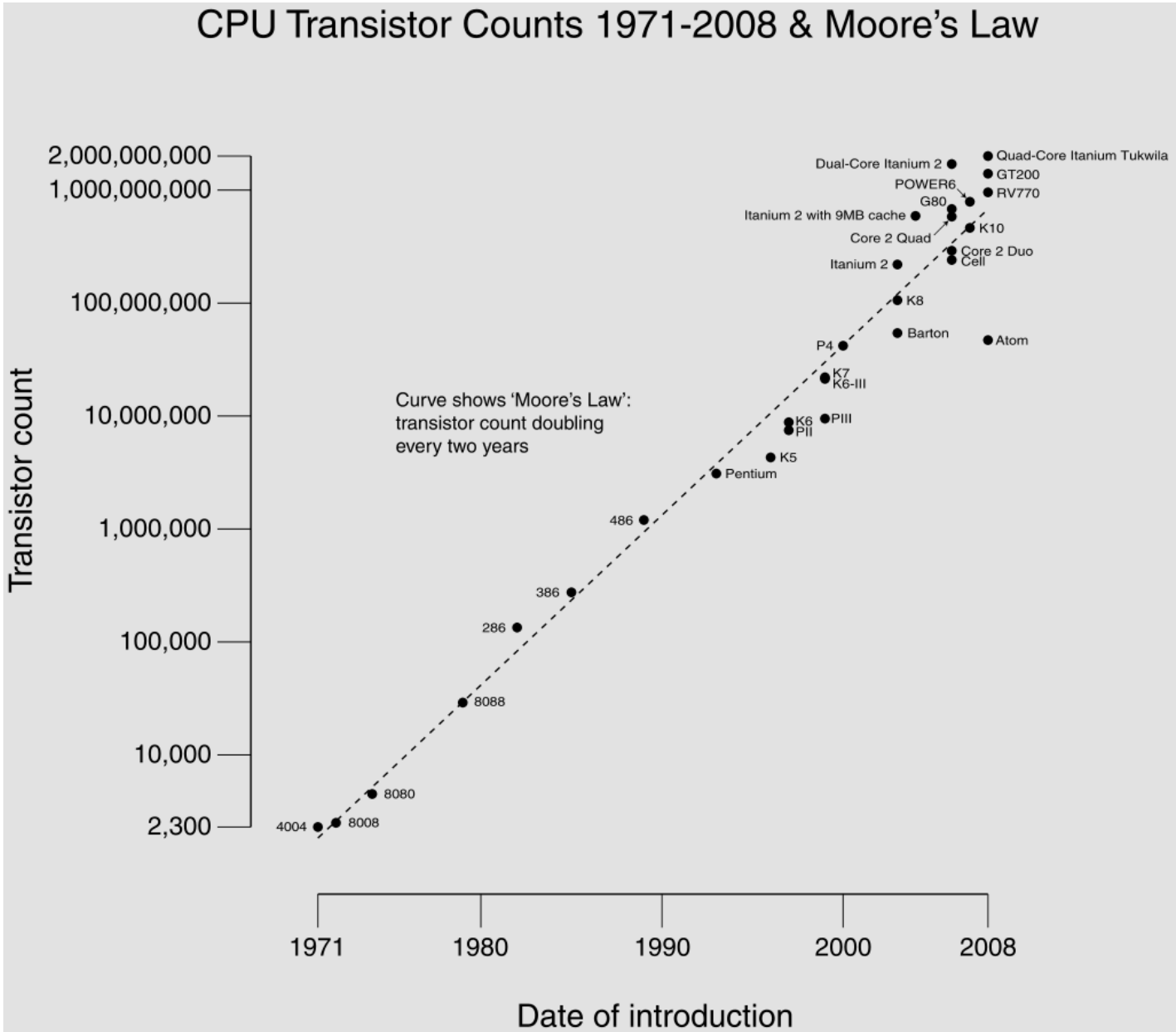
Why is it necessary?

- Modern experiments and observations yield vastly more data to be processsed than in the past.
- As more computing resources become available (SciNet), the bar for cutting edge simulations is raised.
- Science that before could not even be done becomes reachable.

However:

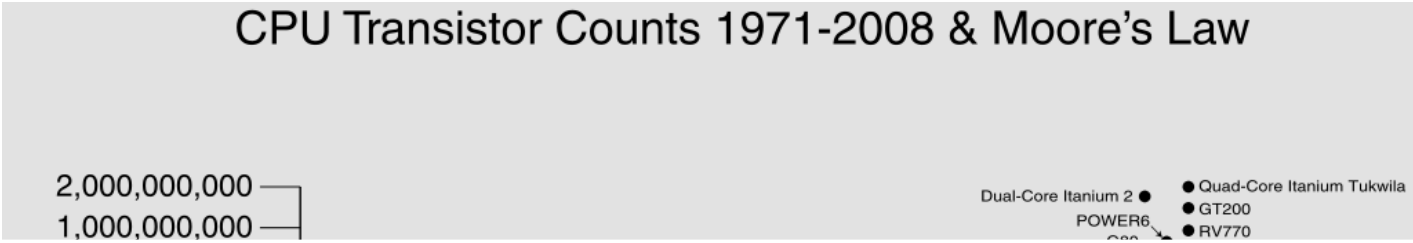
- Advances in clock speeds, bigger and faster memory and disks have been lagging as compared to e.g. 10 years ago.
  Can no longer "just wait a year" and get a better computer.
- So more computing resources here means: more cores running concurrently.
- *Even most laptops now have 2 or more cpus.*
- So parallel computing is necessary.

Wait, what about Moore's Law?



(source: Transistor Count and Moore's Law - 2008.svg, by Wgsimon, wikipedia)

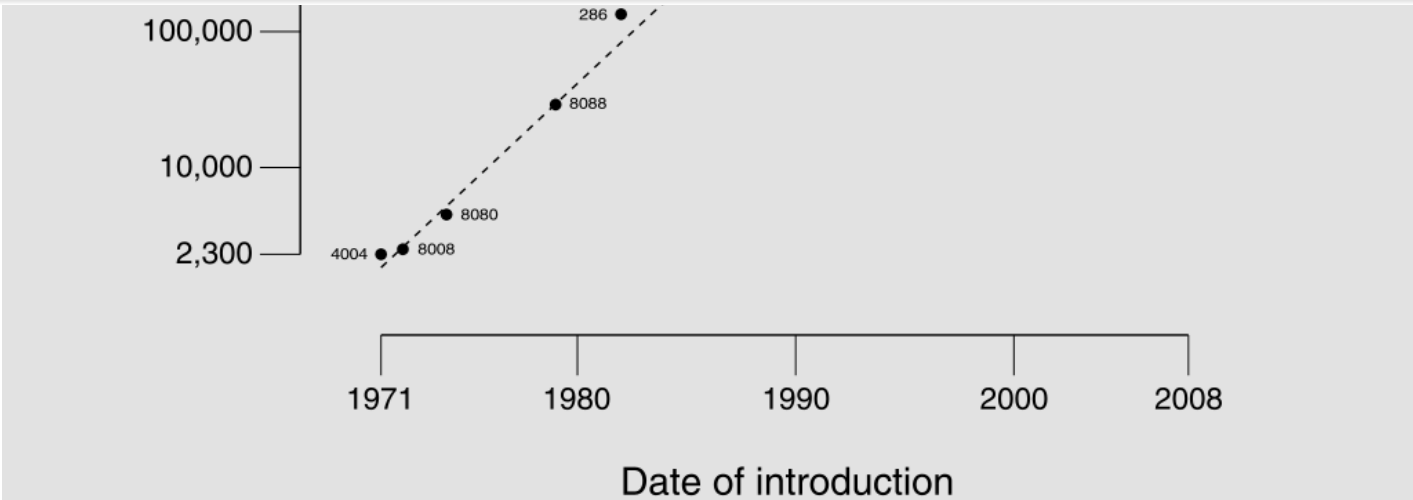
Wait, what about Moore's Law?



Moore's law

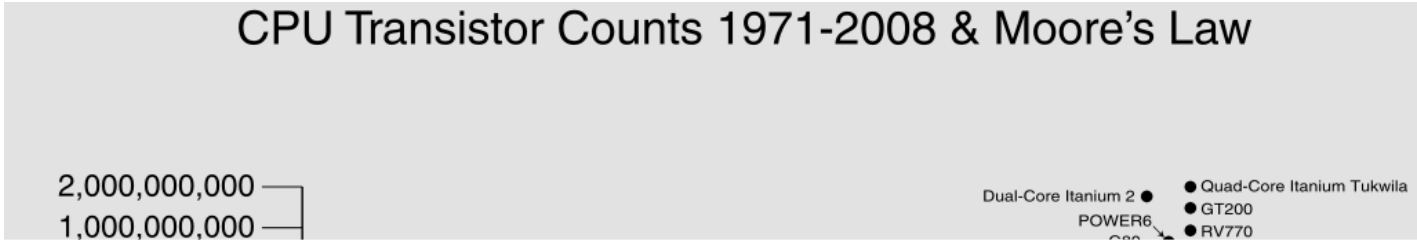
. . . describes a long-term trend in the history of computing hardware. The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years.

(source: Moore's law, wikipedia)



(source: Transistor Count and Moore's Law - 2008.svg, by Wgsimon, wikipedia)

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. . .

- Moores 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.

(source: Transistor Count and Moore's Law - 2008.svg, by Wgsimon, wikipedia)

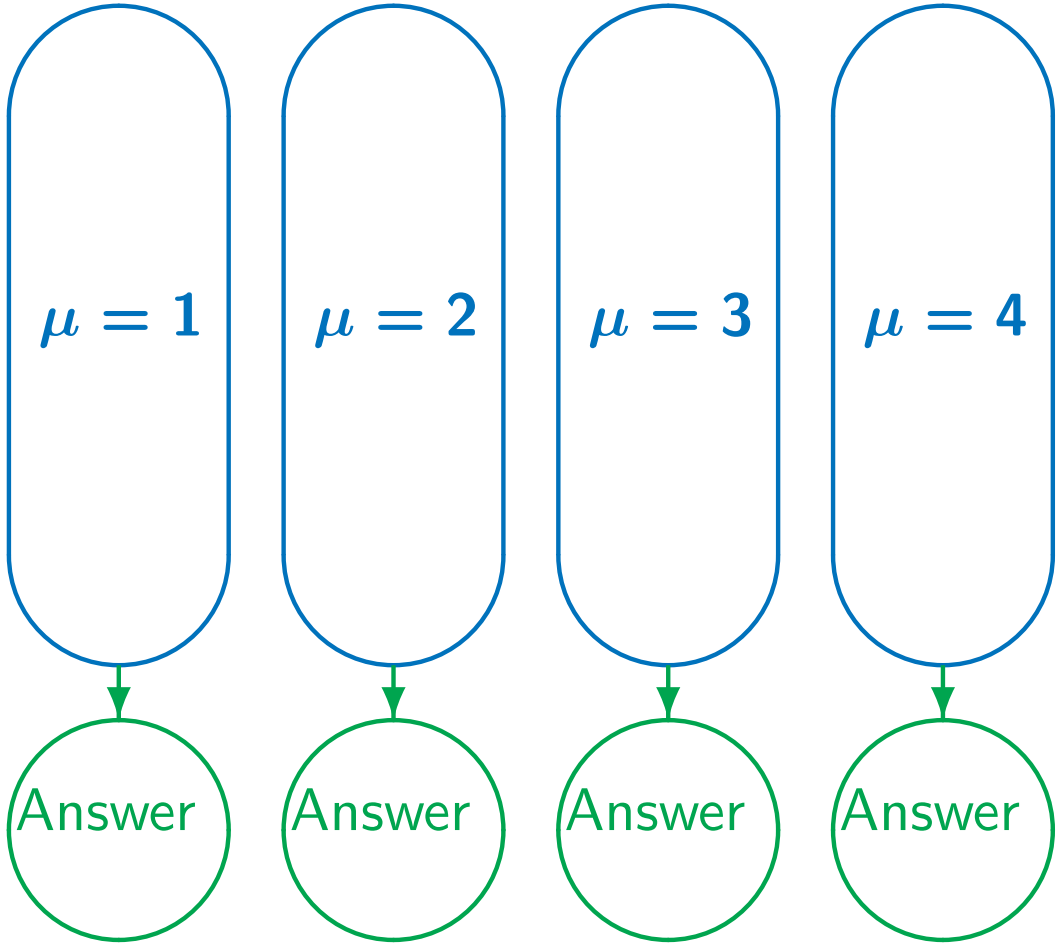
Concurrency

- 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.
- There order of execution should not matter either.
- Data dependencies limit concurrency.

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

Parameter study: 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.



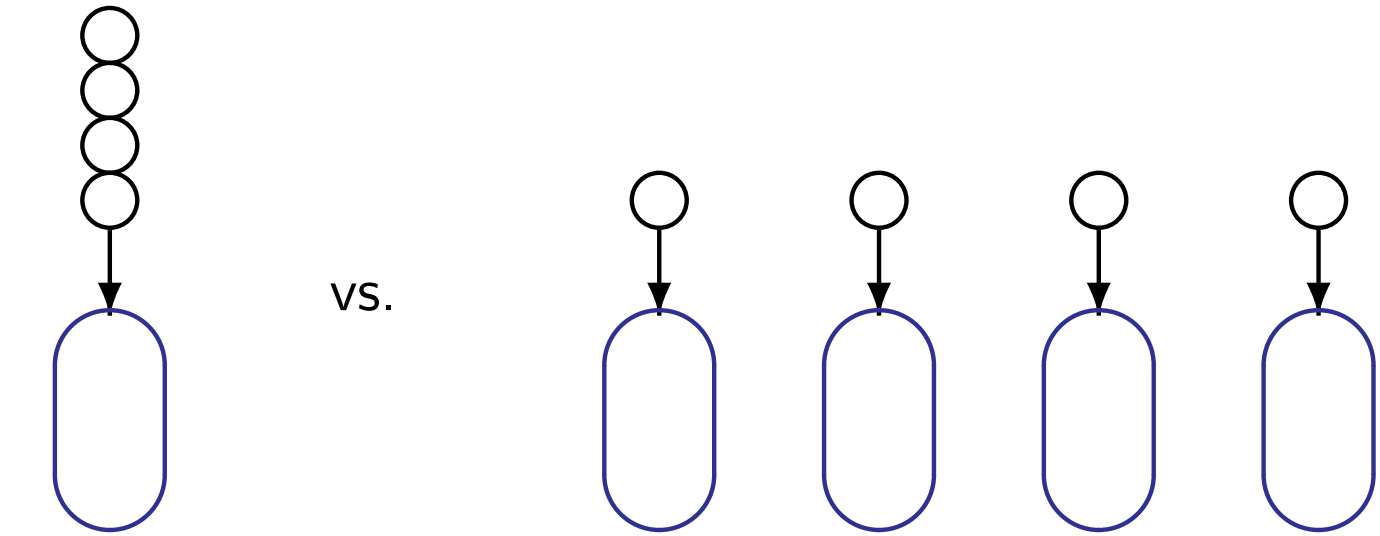
SciNet compute • calcul CANADA

Throughput

- How many tasks can you do per time unit?

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

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



$\mathbf{T = NT_1}$
$\mathbf{H = 1/T_1}$

$\mathbf{T = NT_1/P}$
$\mathbf{H = P/T_1}$

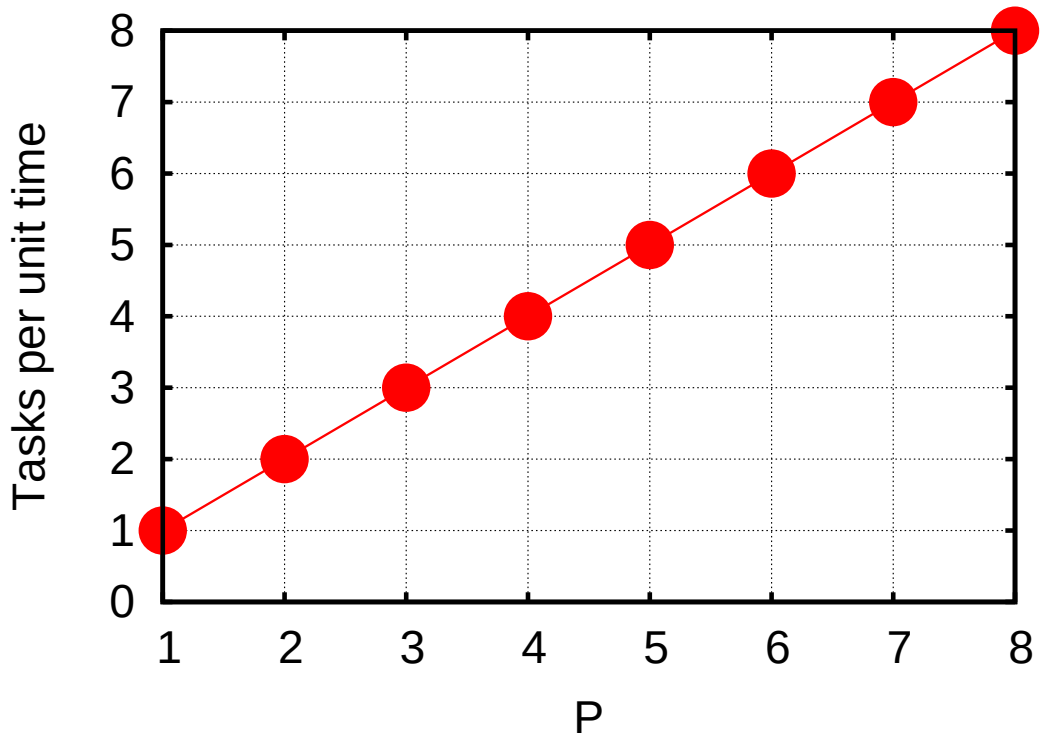
SciNet compute • calcul CANADA

Scaling — Throughput

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

$$\mathbf{H} \propto \mathbf{P}$$

- This is Perfect scaling.

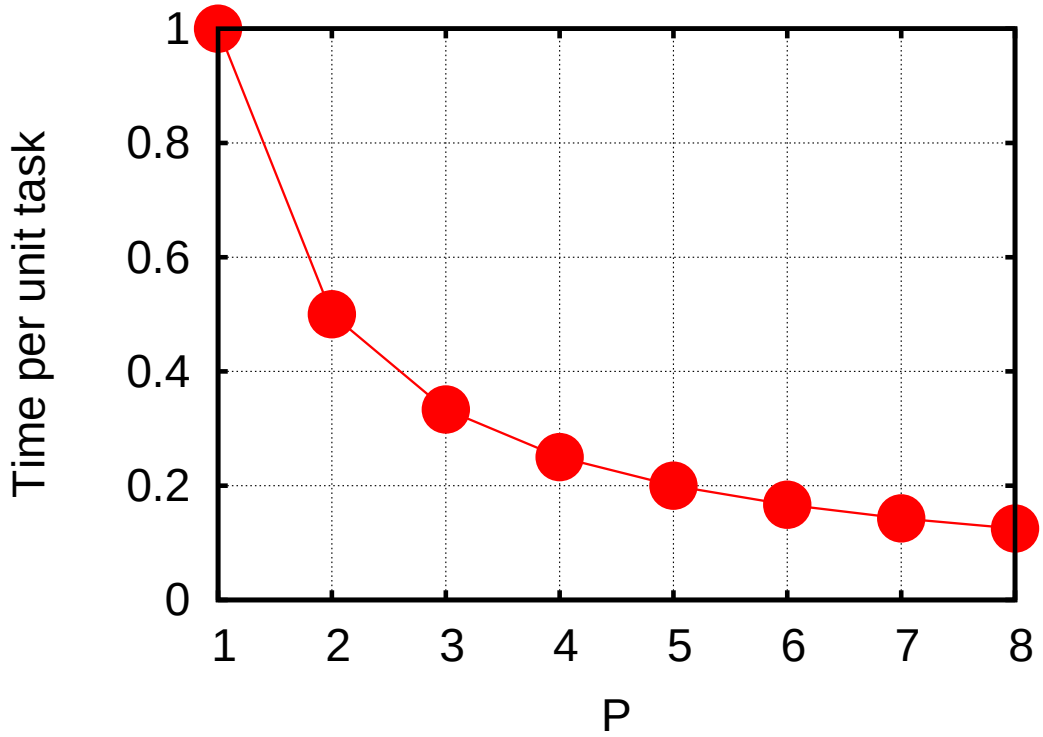


Scaling – Time

- How a problem's timing scales as processor number increases.
- Measured by the time to do one unit. In this case, inverse linear scaling:

$$\mathbf{T \propto 1/P}$$

- Again this is the ideal case, or "embarrasingly parallel".

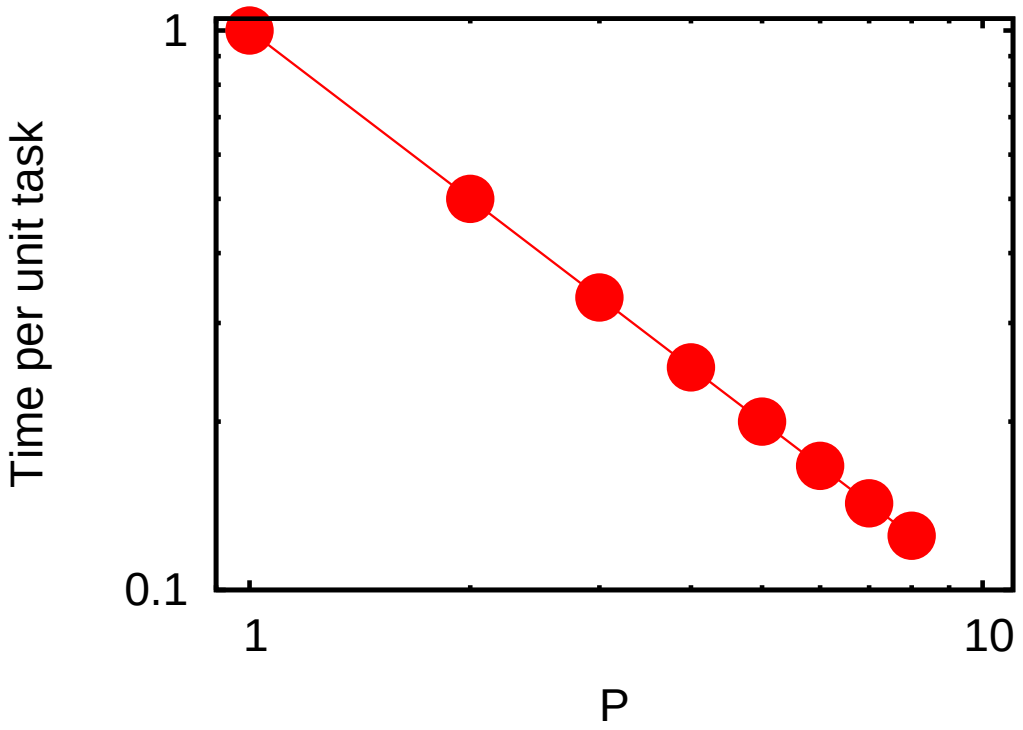


Scaling – Time

- How a problem's timing scales as processor number increases.
- Measured by the time to do one unit. In this case, inverse linear scaling:

$$\mathbf{T \propto 1/P}$$

- Again this is the ideal case, or "embarrasingly parallel".

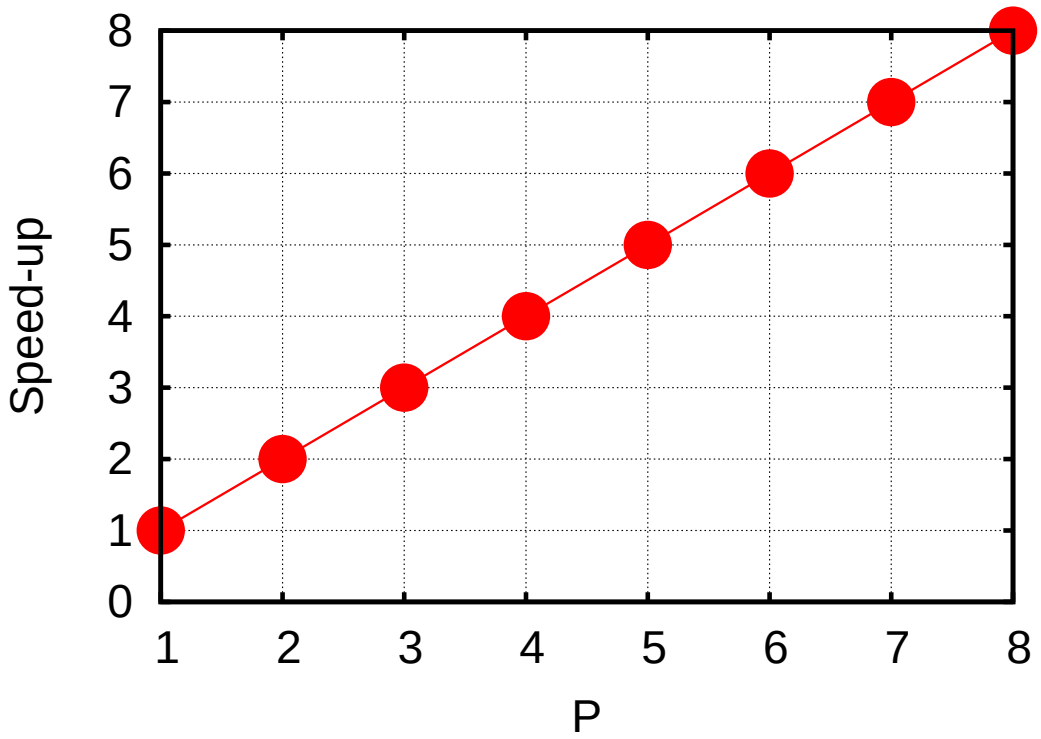


Scaling – Speedup

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

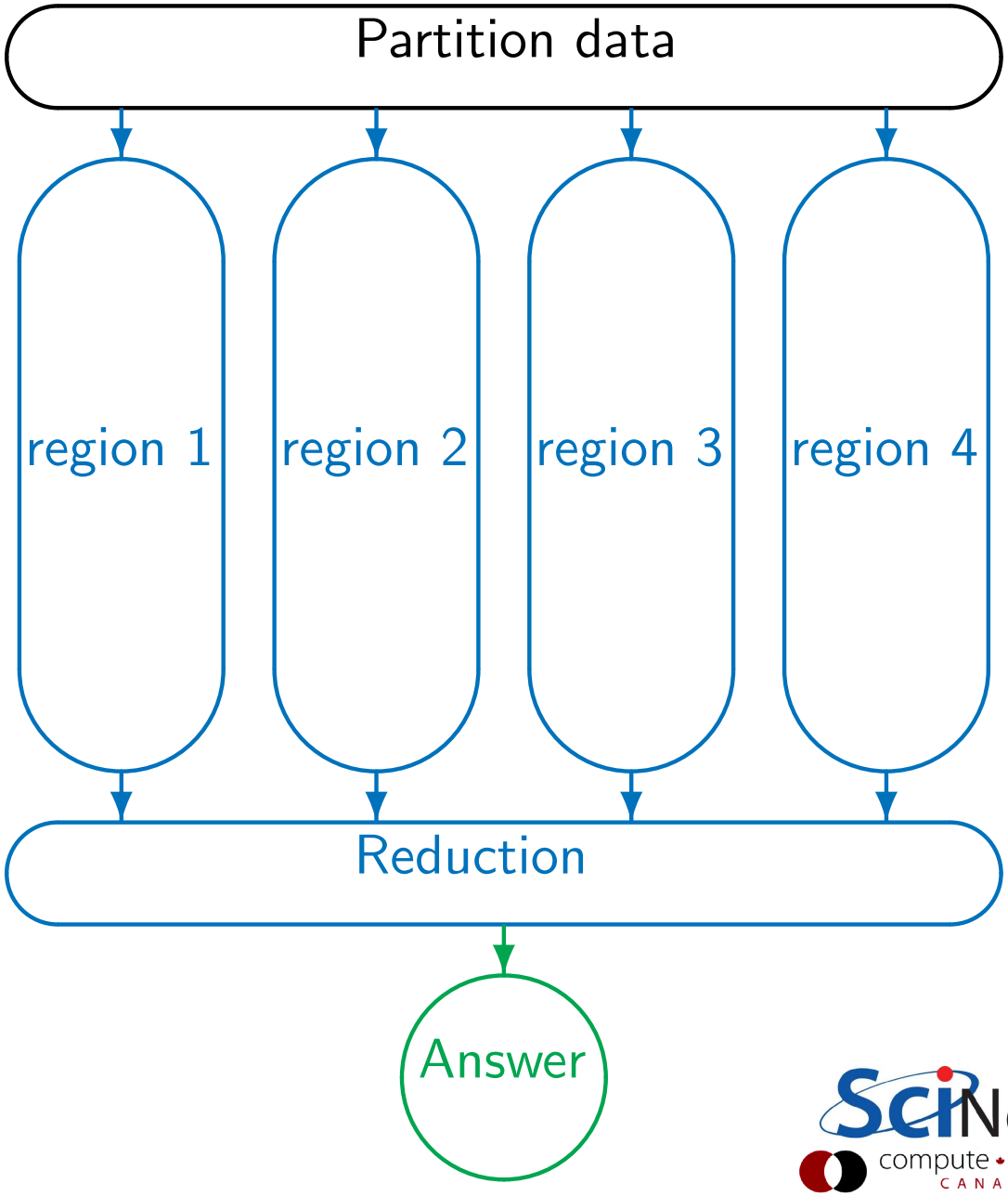
$$S = \frac{T_{serial}}{T(P)} \propto P$$

- For embarrasingly parallel applications: Linear speed up.



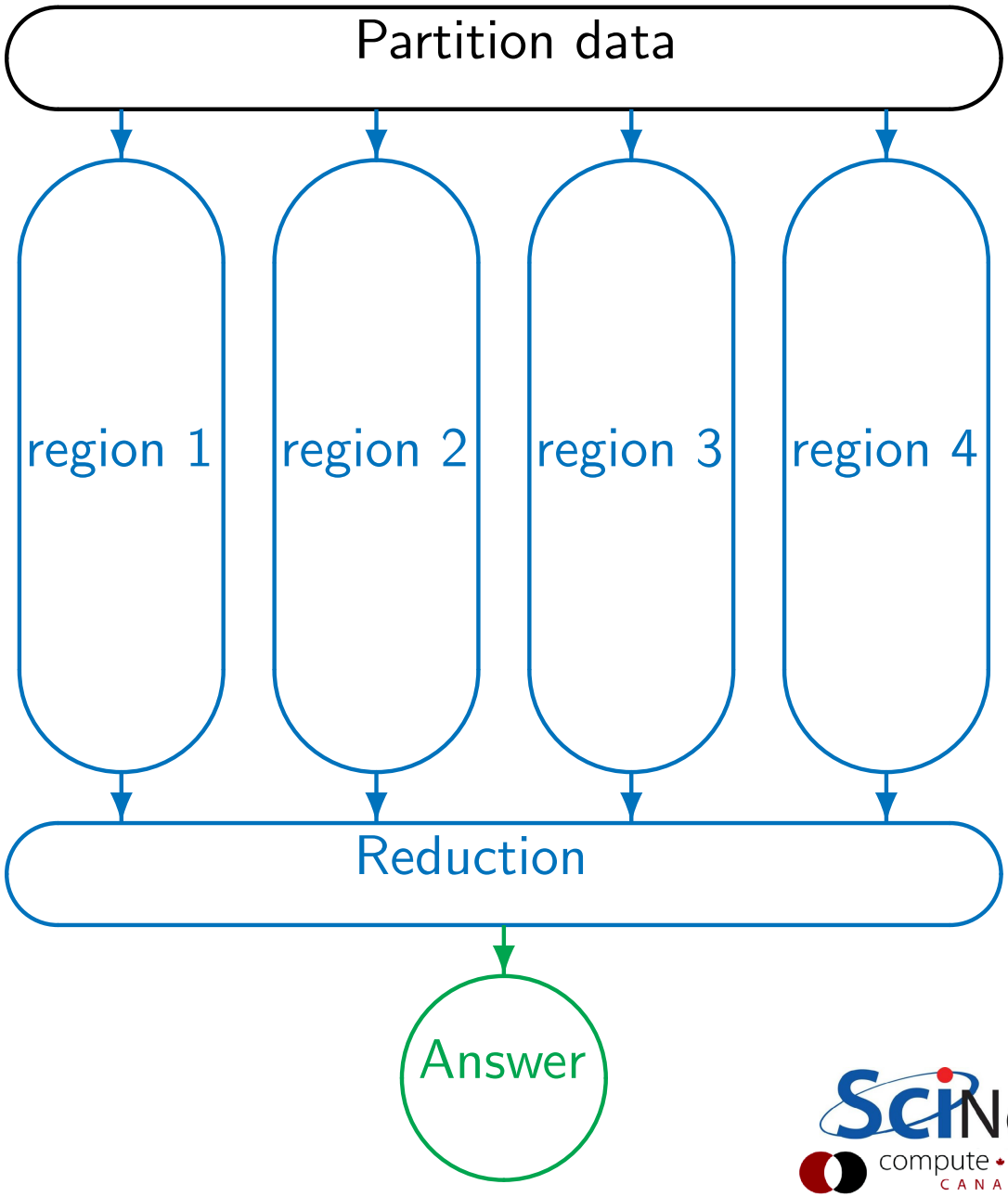
Non-ideal cases

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



Non-ideal cases

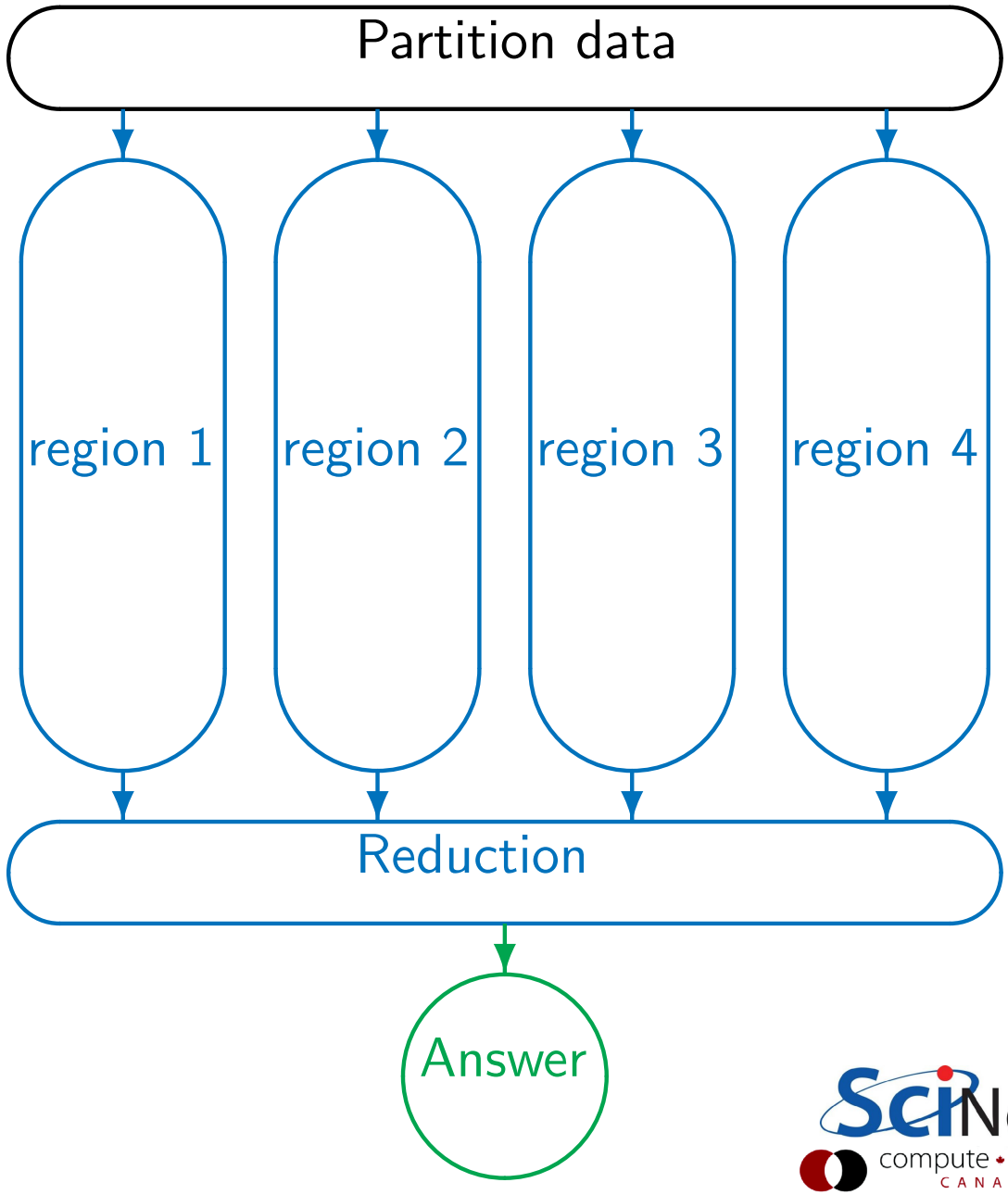
Parallel region $\Rightarrow$



Non-ideal cases

Parallel region $\Rightarrow$

Perfectly Parallel
(for large **N**)

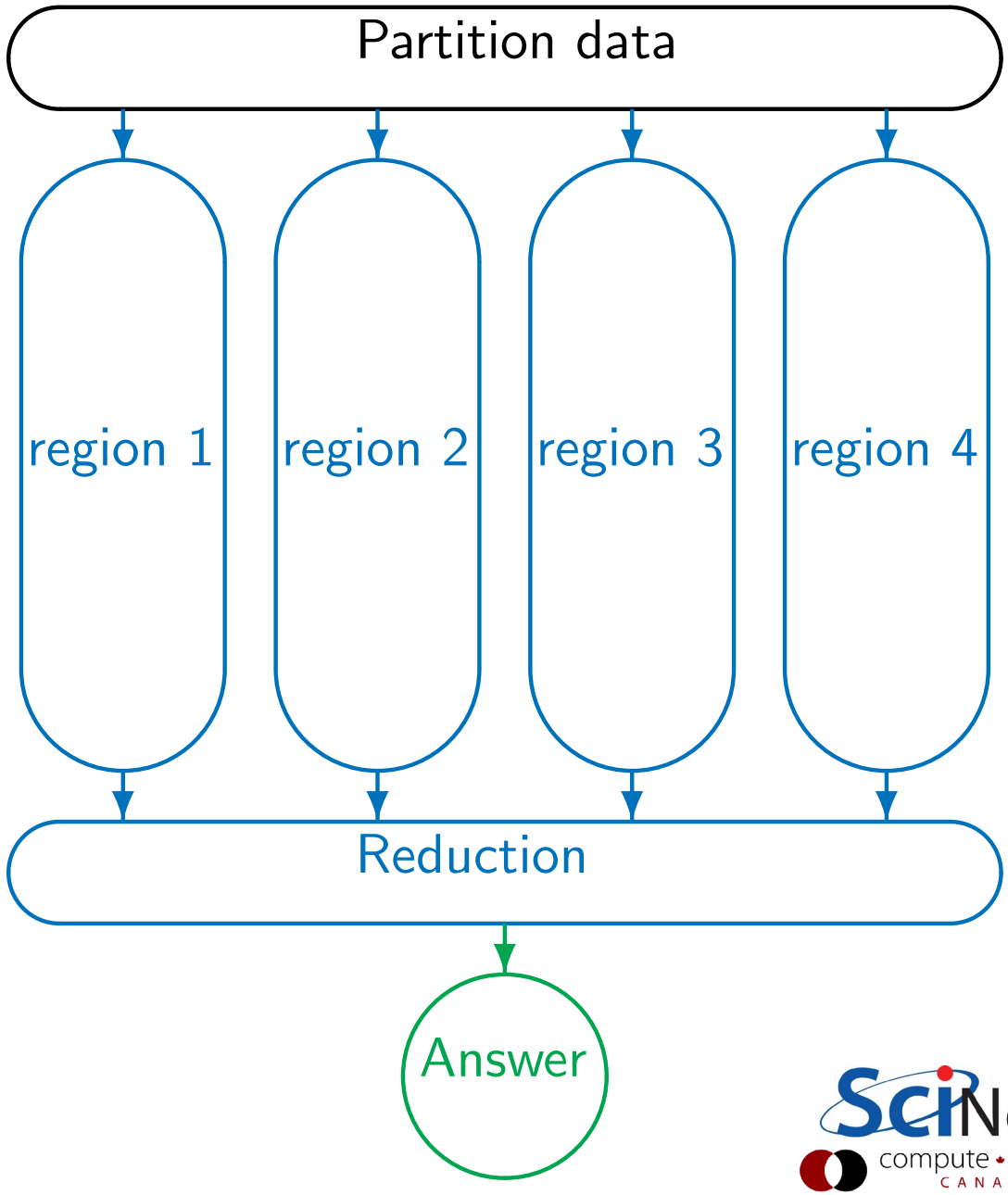


Non-ideal cases

Parallel region $\Rightarrow$

Perfectly Parallel
(for large **N**)

Serial portion $\Rightarrow$



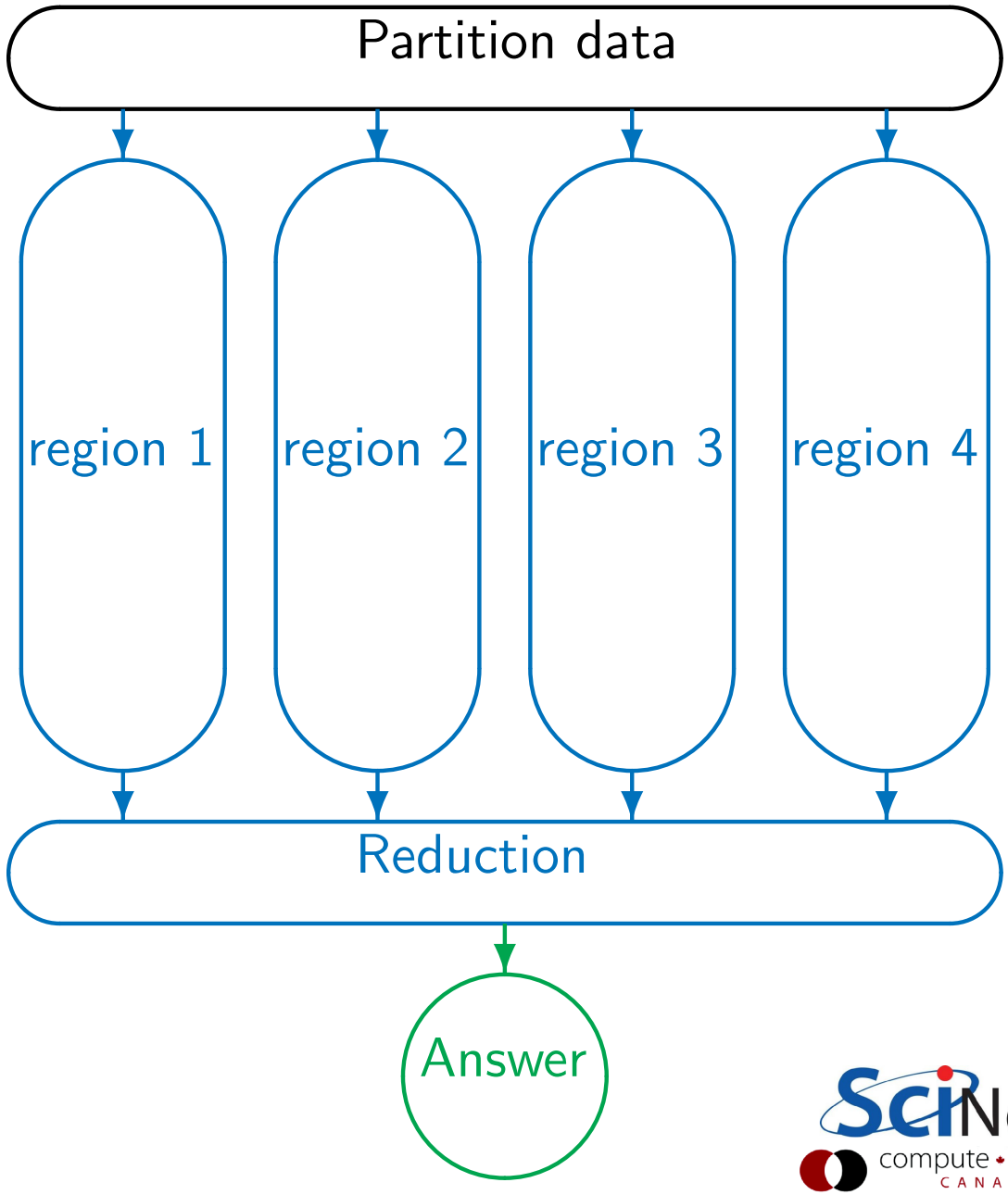
Non-ideal cases

Parallel overhead $\Rightarrow$

Parallel region $\Rightarrow$

Perfectly Parallel
(for large **N**)

Serial portion $\Rightarrow$



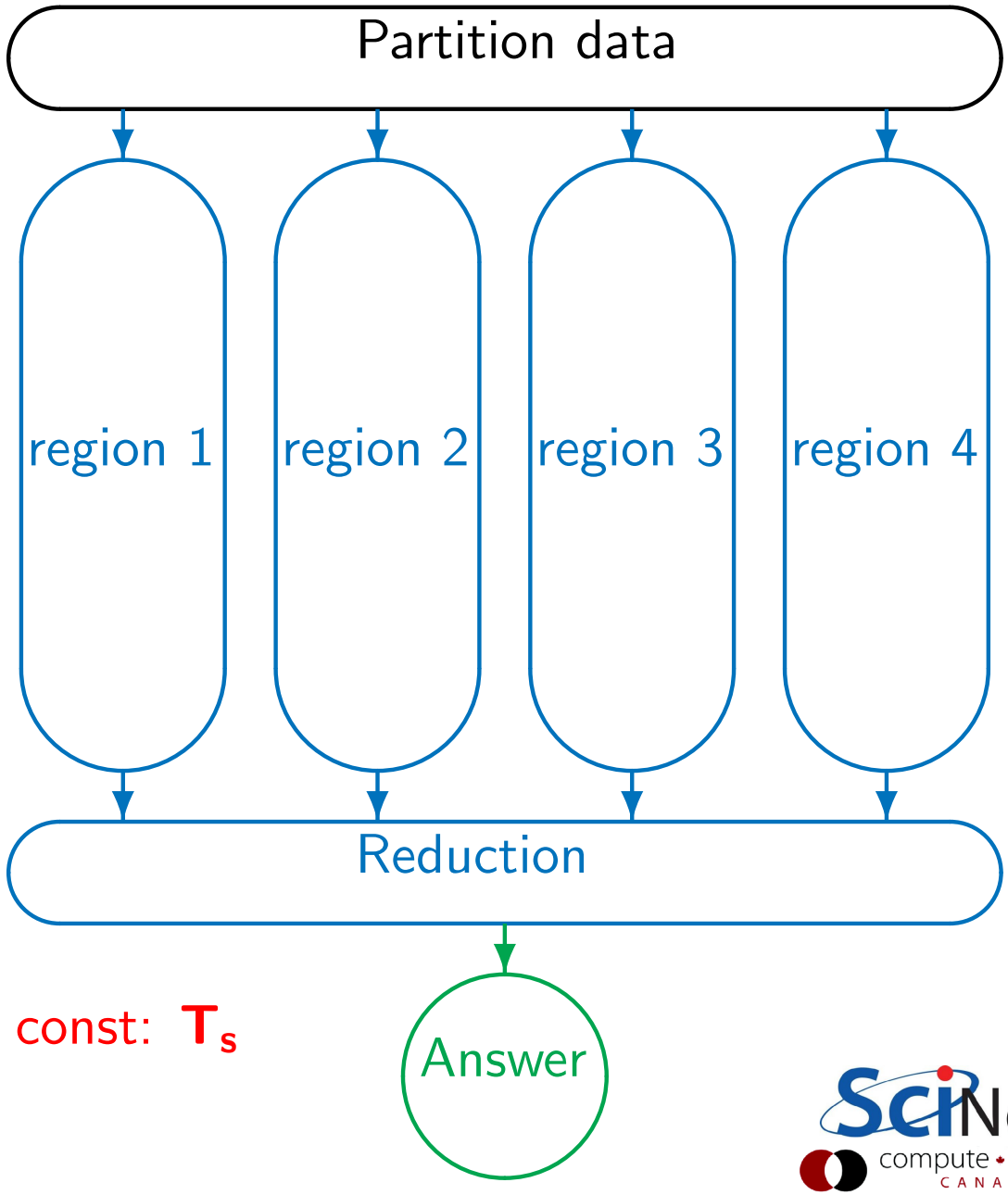
Non-ideal cases

Parallel overhead $\Rightarrow$

Parallel region $\Rightarrow$

Perfectly Parallel
(for large **N**)

Serial portion $\Rightarrow$



Suppose non-parallel part const: $\mathbf{T_s}$

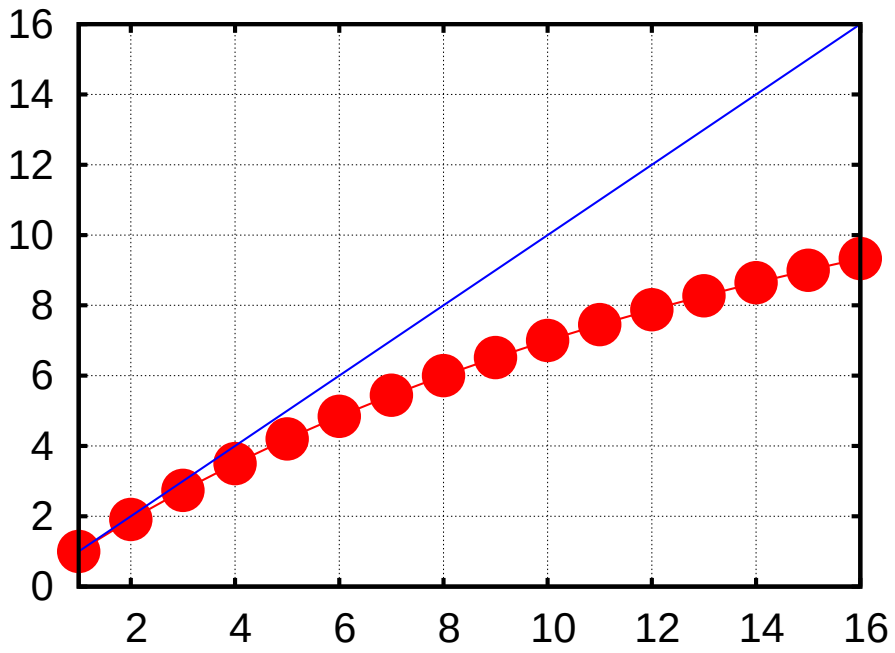
Amdahl's law

Speed-up (without parallel overhead):

$$S = \frac{NT_1 + T_s}{\frac{NT_1}{P} + T_s}$$

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

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



(for $f = 5\%$)

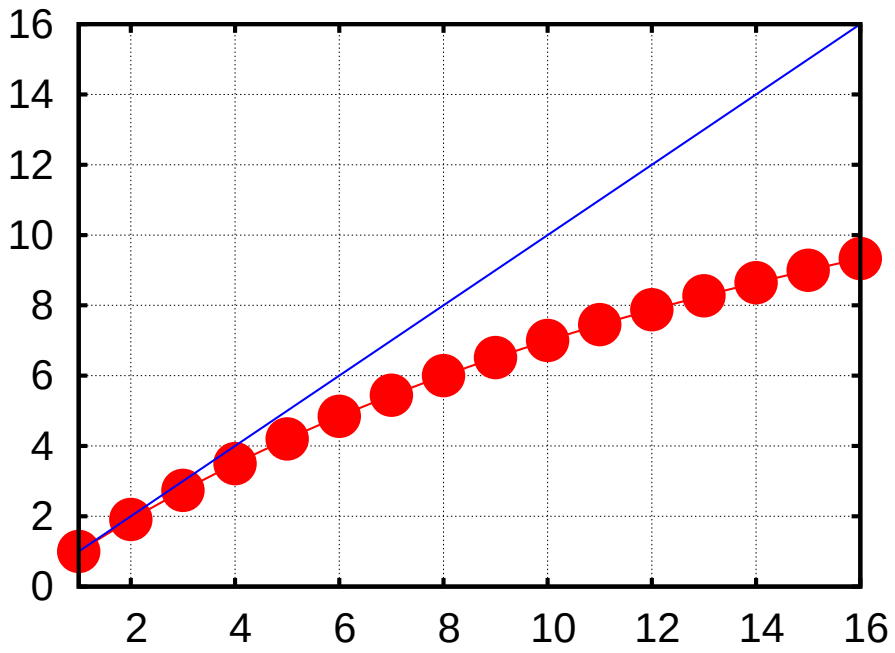
Amdahl's law

Speed-up (without parallel overhead):

$$S = \frac{NT_1 + T_s}{\frac{NT_1}{P} + T_s}$$

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

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



(for $f = 5\%$)

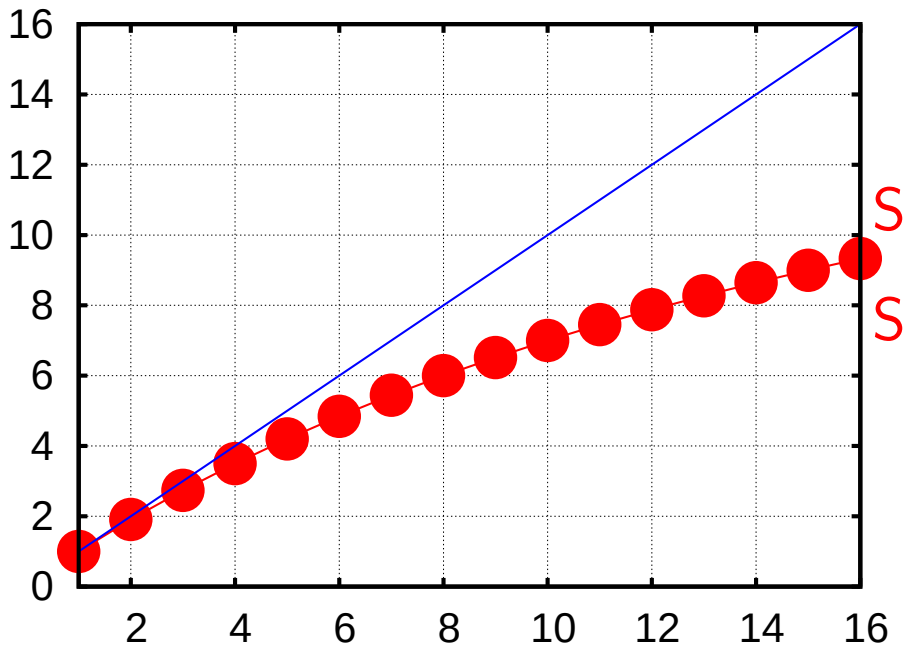
Amdahl's law

Speed-up (without parallel overhead):

$$S = \frac{NT_1 + T_s}{\frac{NT_1}{P} + T_s}$$

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

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



Serial part dominates asymptotically.

Speed-up limited, no matter size of **P**.

(for $f = 5\%$)

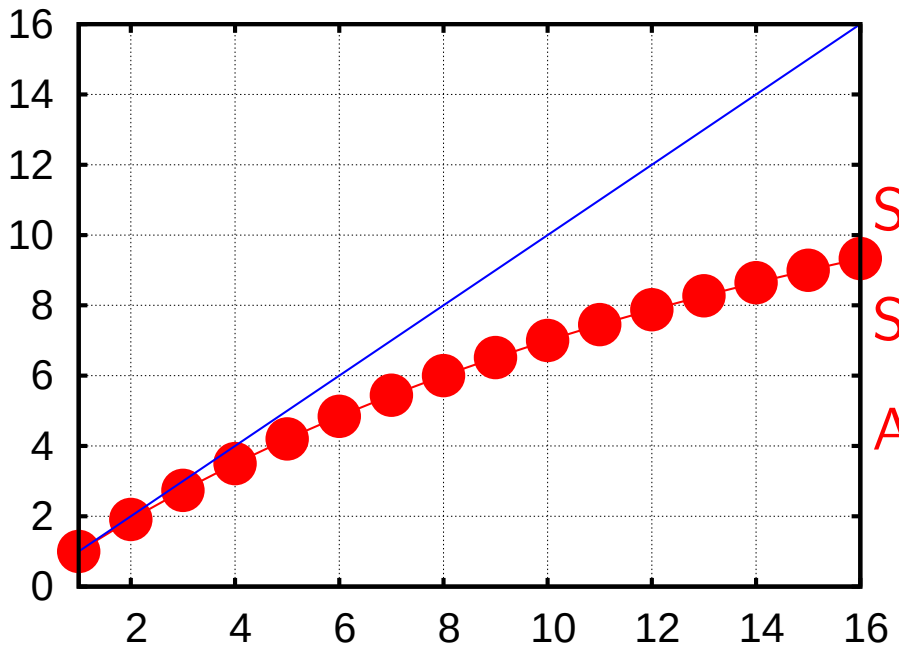
Amdahl's law

Speed-up (without parallel overhead):

$$S = \frac{NT_1 + T_s}{\frac{NT_1}{P} + T_s}$$

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

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



Serial part dominates asymptotically.

Speed-up limited, no matter size of **P**.

And this is the overly optimistic case!

(for $f = 5\%$)

Scaling efficiency

Speed-up compared to ideal factor **P**:

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

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

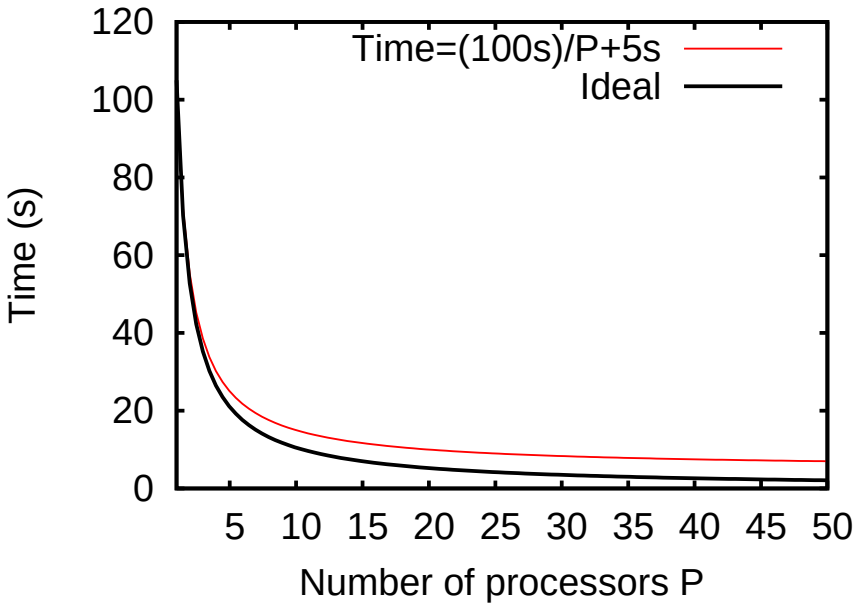
$$\textbf{Efficiency} \sim \frac{\textbf{1}}{\textbf{fP}} \overset{\textbf{P}\to\infty}{\longrightarrow} \textbf{0}$$

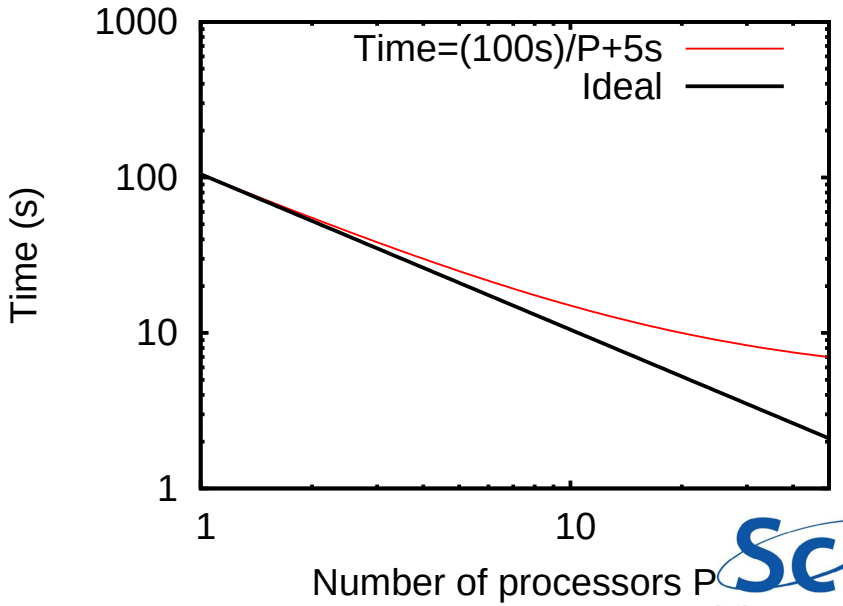
You cannot get 100% efficiency in any non-trivial problem.
All you can aim for here is to make the efficiency as least low as possible.
Sometimes, that can mean running on less processors, but more problems at the same time.

Timing example

- Say 100s in integration cost
- 5s in reduction
- Neglect communication cost
- What happens as we vary number of processors **P**?

$$\textbf{Time} = \textbf{(100s)/P} + \textbf{5}$$





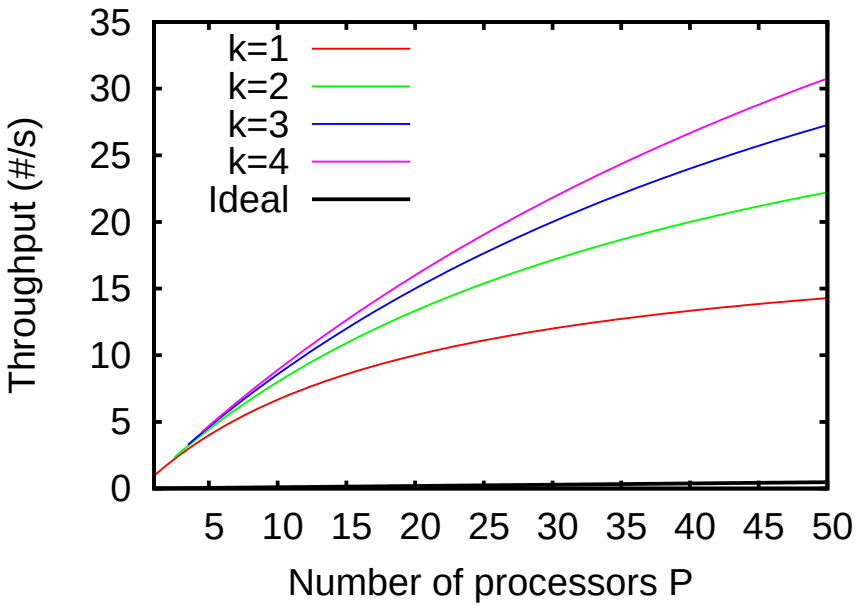
Throughput example

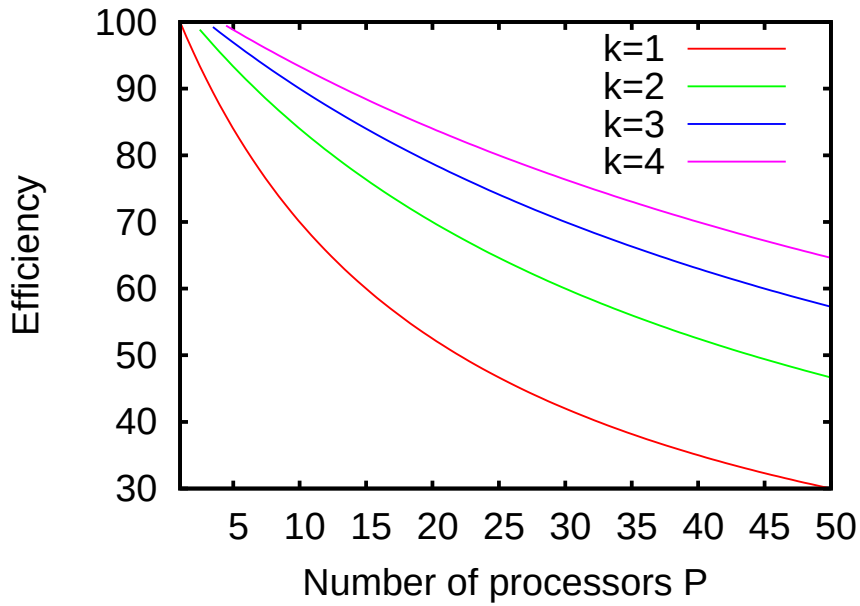
$$H(P) = \frac{N}{Time(P)}$$

- Say we are doing **k** at the same time, on **P** processors total.

$$H_k(P) = \frac{kN}{Time(P/k)}$$

Say $N = 100$:





Big Lesson #1

Always keep throughput in mind: if you have several runs, running more of them at the same time on less processors per run is often advantageous.

Less ideal case of Amdahl's law

We assumed reduction is constant.
But it will in fact increase with P,
from sum of results of all processors

$$T_s \approx PT_1$$

Serial fraction now a function of **P**:

$$f(P) = \frac{P}{N}$$

Amdahl:

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

Less ideal case of Amdahl's law

We assumed reduction is constant.
But it will in fact increase with P,
from sum of results of all processors

$$T_s \approx PT_1$$

Serial fraction now a function of **P**:

$$f(P) = \frac{P}{N}$$

Amdahl:

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

Example: $N = 100$, $T_1 = 1s \ldots$

Less ideal case of Amdahl's law

We assumed reduction is constant.
But it will in fact increase with P,
from sum of results of all processors

$$T_s \approx PT_1$$

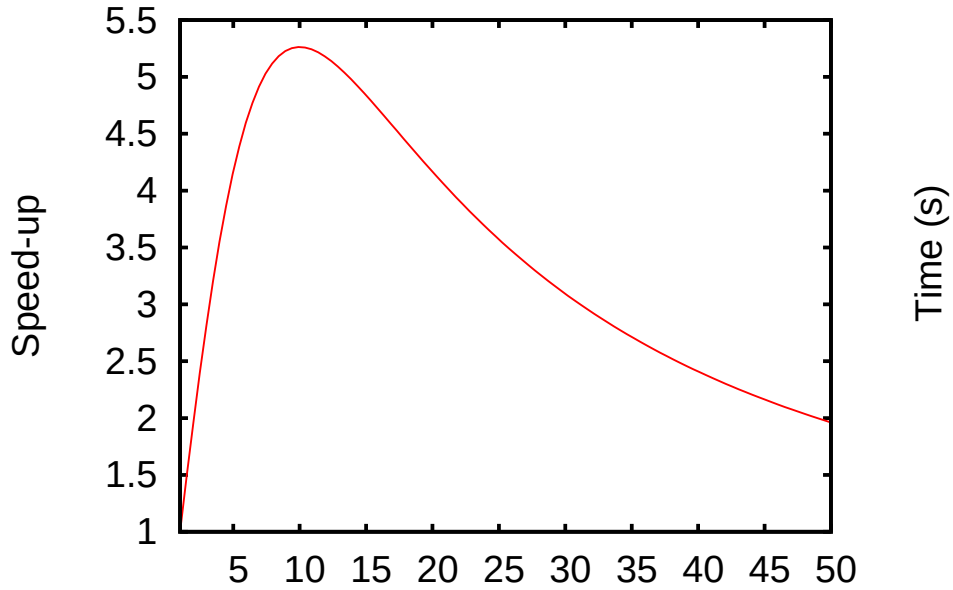
Serial fraction now a function of **P**:

$$f(P) = \frac{P}{N}$$

Amdahl:

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

Example: $N = 100$, $T_1 = 1s$...

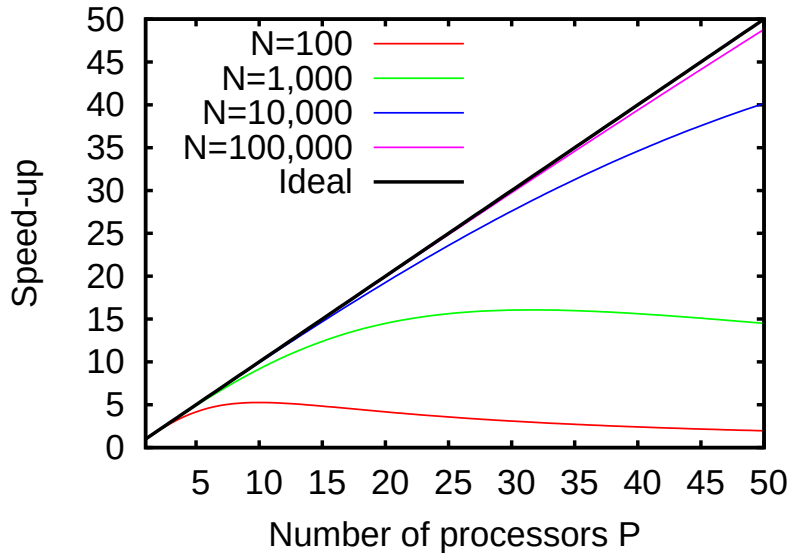


Trying to beat Amdahl's law #1

Scale up!

The larger **N**, the smaller the serial fraction:

$$f(P) = \frac{P}{N}$$

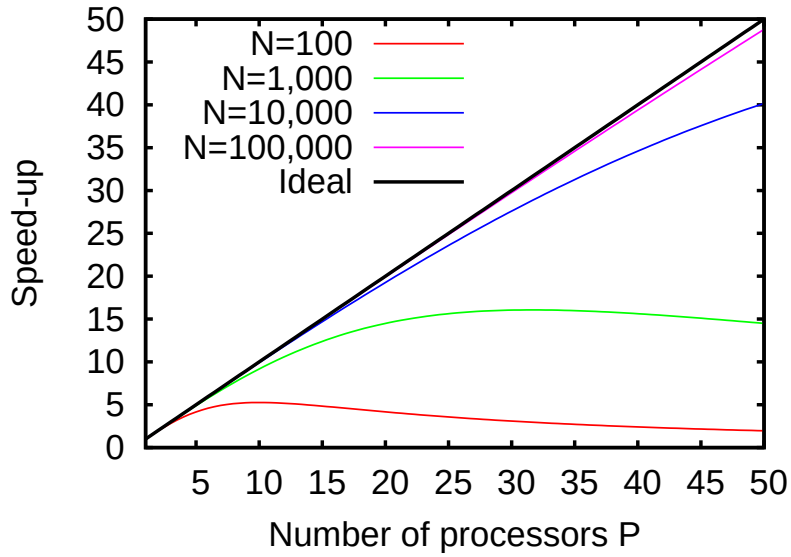


Trying to beat Amdahl's law #1

Scale up!

The larger **N**, the smaller the serial fraction:

$$\mathbf{f(P) = \frac{P}{N}}$$



Weak scaling: Increase problem size while increasing **P**

$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

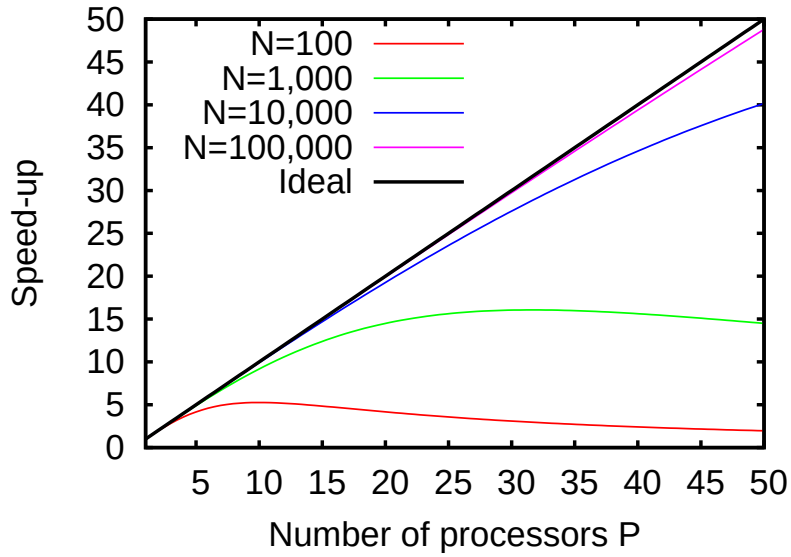
Good weak scaling means this time approaches a constant for large **P**.

Trying to beat Amdahl's law #1

Scale up!

The larger **N**, the smaller the serial fraction:

$$\mathbf{f(P) = \frac{P}{N}}$$



Weak scaling: Increase problem size while increasing **P**

$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

Good weak scaling means this time approaches a constant for large **P**.

Gustafson's Law

Any large enough problem can be efficiently parallelized (Efficiency$\rightarrow$1).

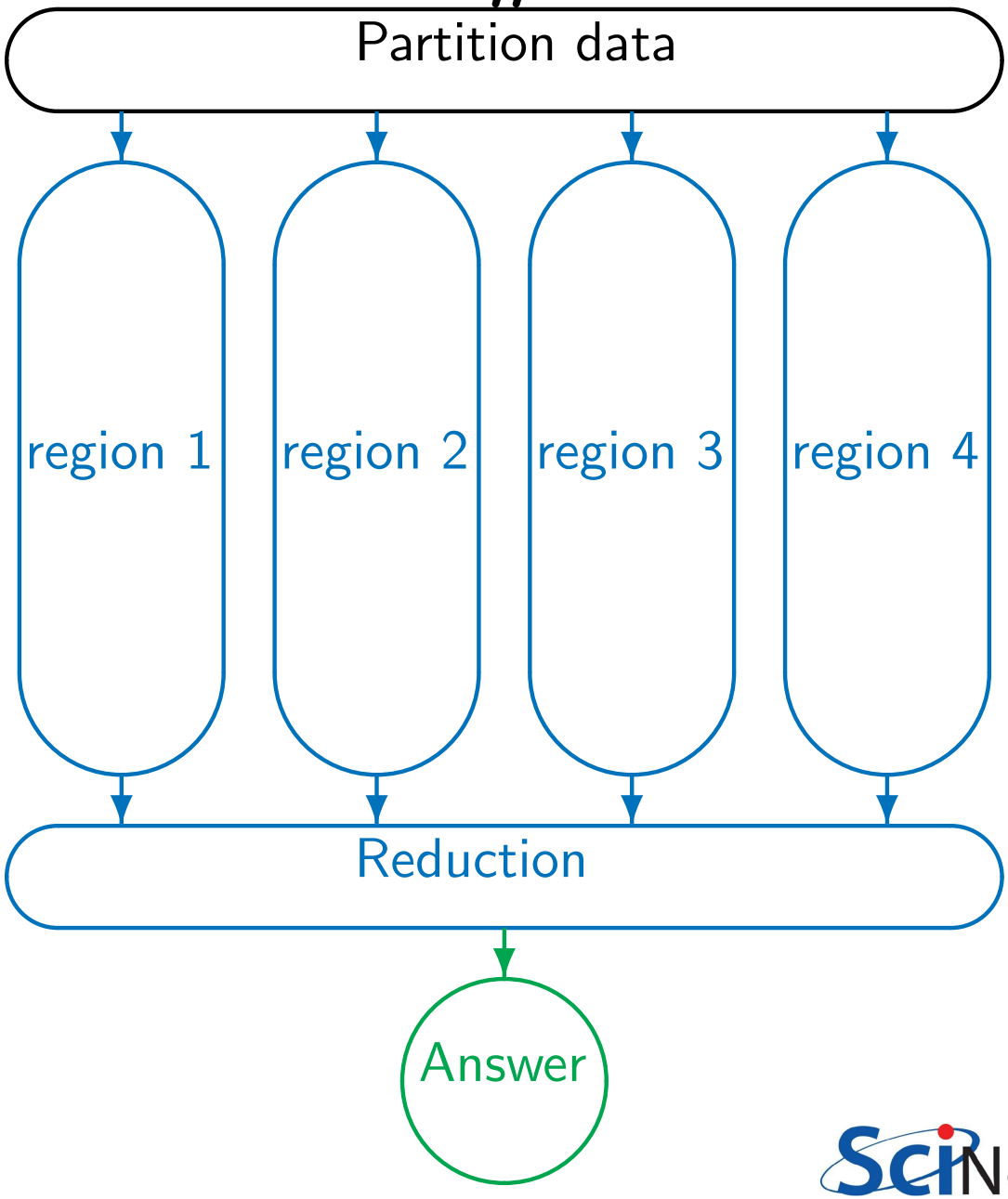
Trying to beat Amdahl's law #2

Parallel overhead $\Rightarrow$

Parallel region $\Rightarrow$

Perfectly Parallel
(for large **N**)

Serial portion $\Rightarrow$



Trying to beat Amdahl's law #2

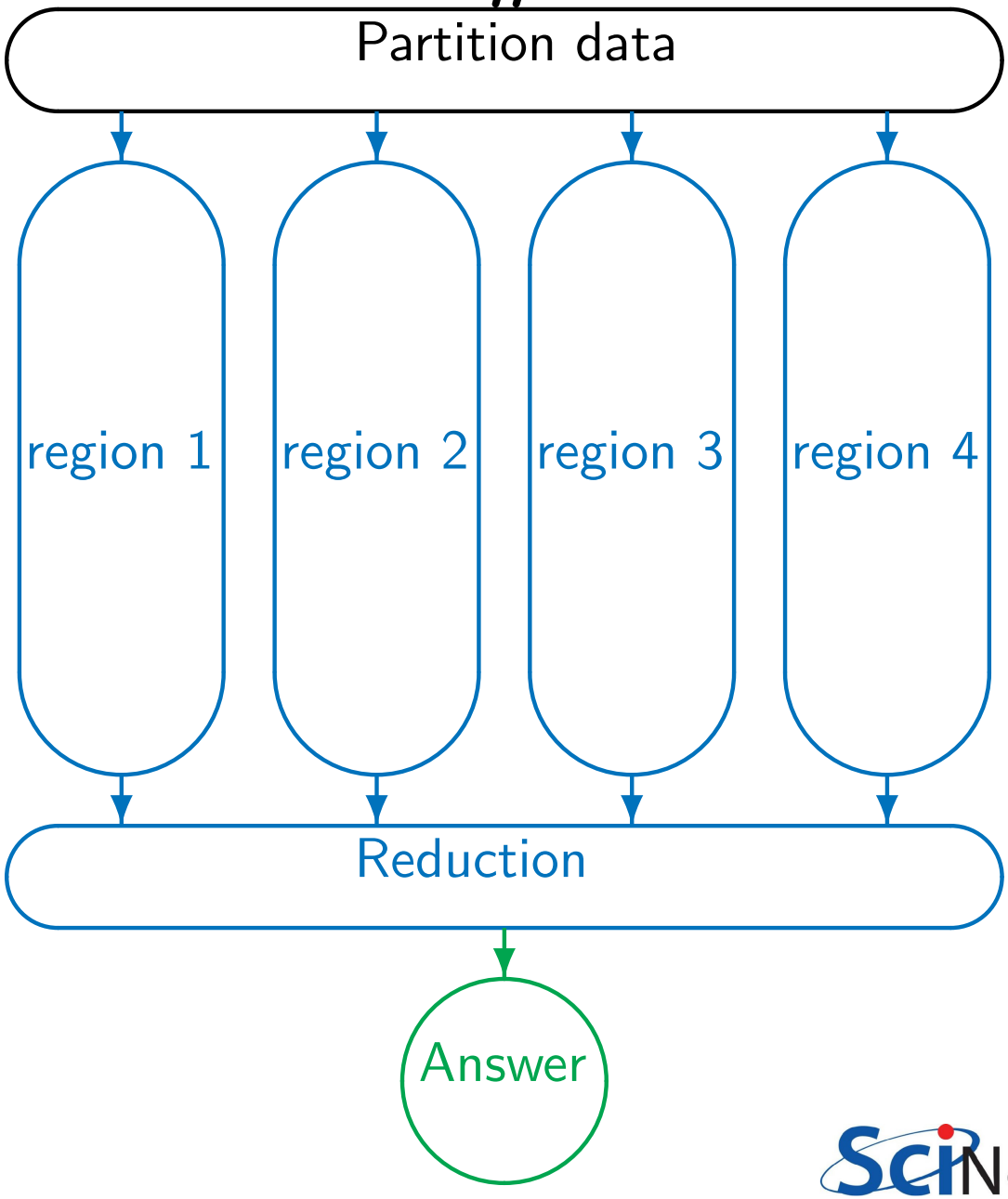
Parallel overhead ⇒

Parallel region ⇒

Perfectly Parallel
(for large **N**)

Serial portion ⇒

Rewrite



Trying to beat Amdahl's law #2

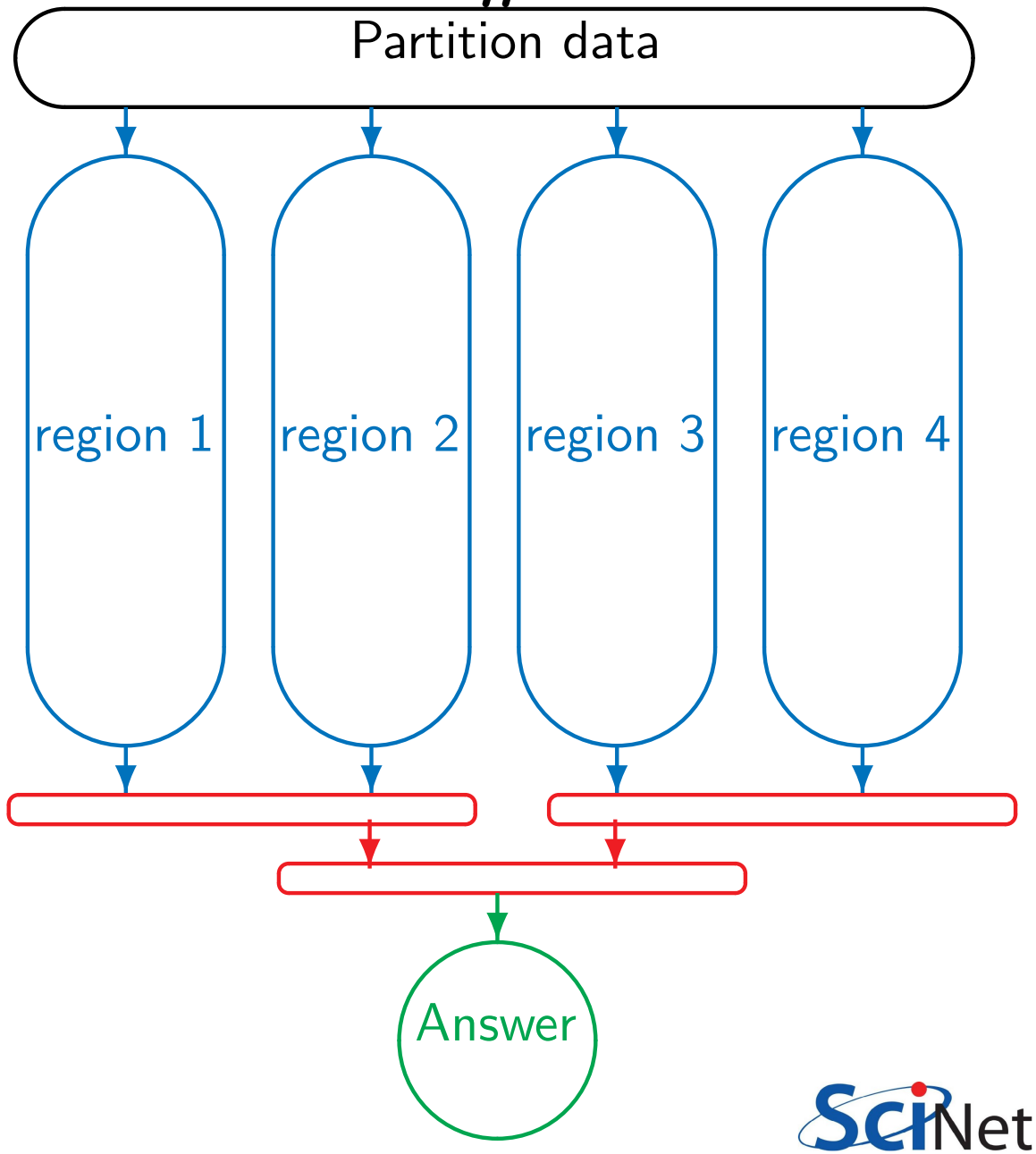
Parallel overhead $\Rightarrow$

Parallel region $\Rightarrow$

Perfectly Parallel
(for large **N**)

Serial portion $\Rightarrow$

Rewrite



Trying to beat Amdahl's law #2

Parallel overhead $\Rightarrow$

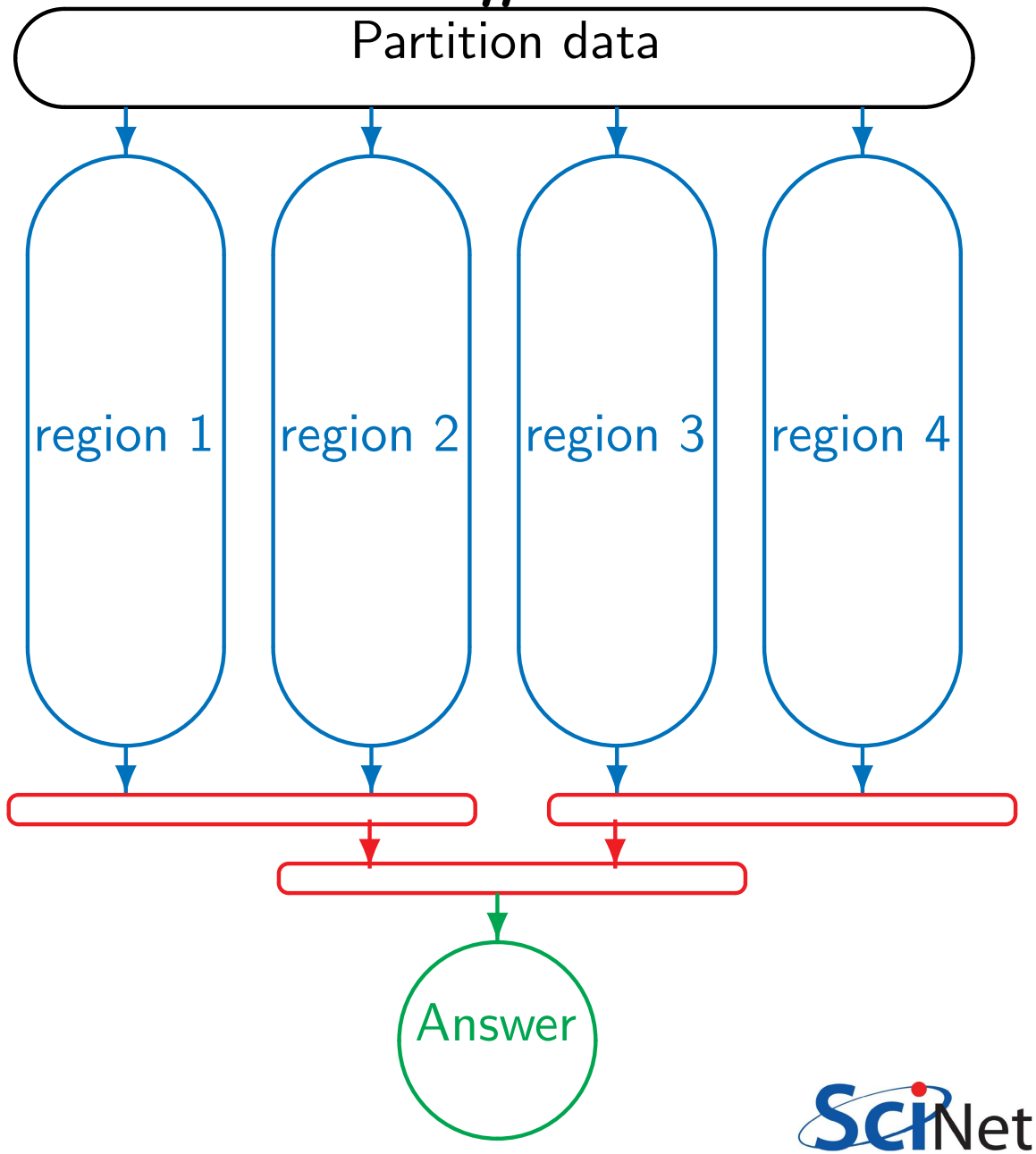
Parallel region $\Rightarrow$

Perfectly Parallel
(for large **N**)

Serial portion $\Rightarrow$

Rewrite

$\propto$ **^{2} log P**



Trying to beat Amdahl's law #2

'Serial' fraction now different function of **P**:

$$\mathbf{f(P) = \frac{^2 \log P}{N}}$$

Amdahl:

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

Trying to beat Amdahl's law #2

'Serial' fraction now different function of $\mathbf{P}$:

$$\mathbf{f(P) = \frac{^2 \log P}{N}}$$

Amdahl:

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

Example: $\mathbf{N = 100}$, $\mathbf{T_1 = 1s}$...

Trying to beat Amdahl's law #2

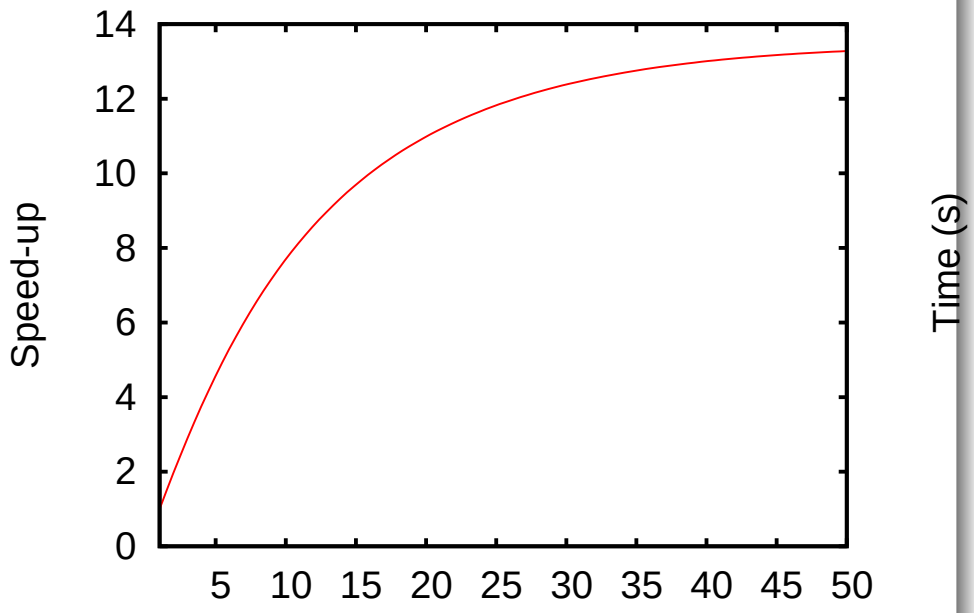
'Serial' fraction now different function of **P**:

$$\mathbf{f(P) = \frac{^2 \log P}{N}}$$

Amdahl:

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

Example: $\mathbf{N = 100}$, $\mathbf{T_1 = 1s}$...



Trying to beat Amdahl's law #2

Weak scaling

$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

Should approach constant for large **P**.
Let's see...

Trying to beat Amdahl's law #2

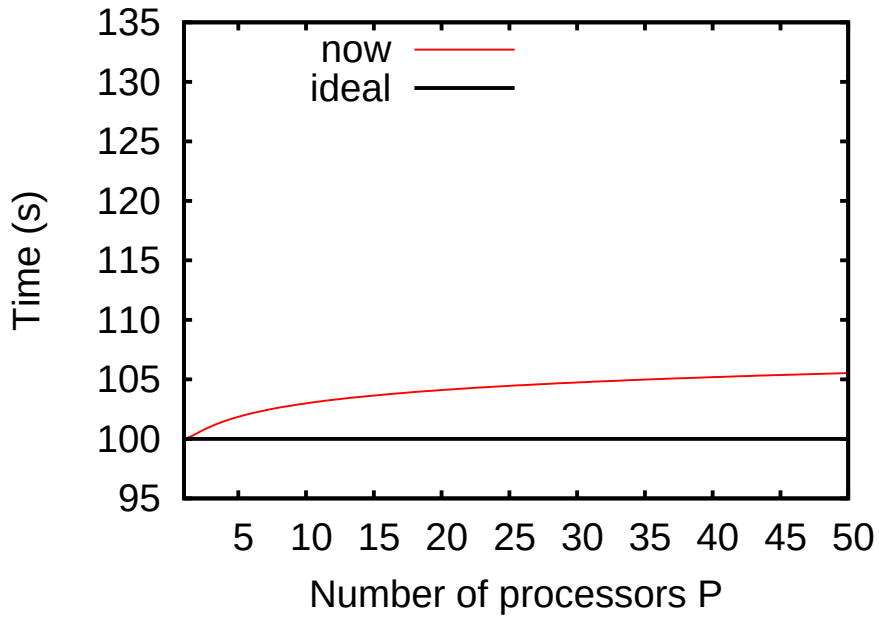
Weak scaling

$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

Should approach constant for large **P**.

Let's see...

Not quite!



Trying to beat Amdahl's law #2

Weak scaling

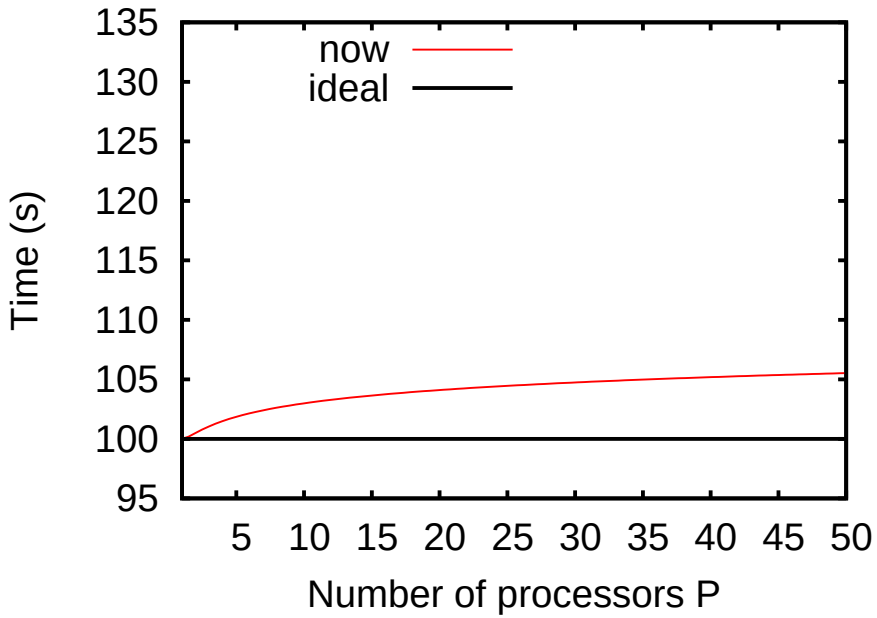
$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

Should approach constant for large **P**.

Let's see. . .

Not quite!

But much better than before.



Trying to beat Amdahl's law #2

Weak scaling

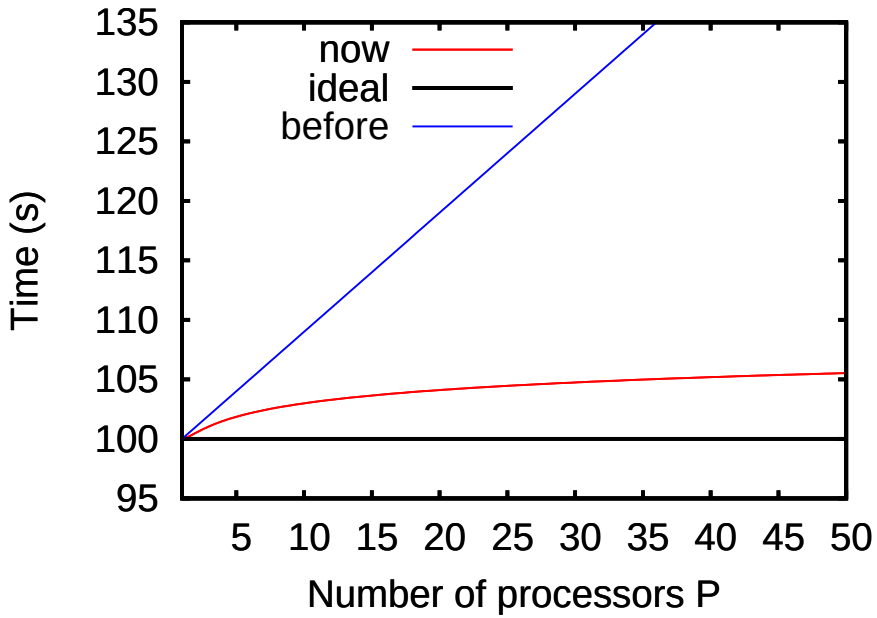
$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

Should approach constant for large **P**.

Let's see...

Not quite!

But much better than before.

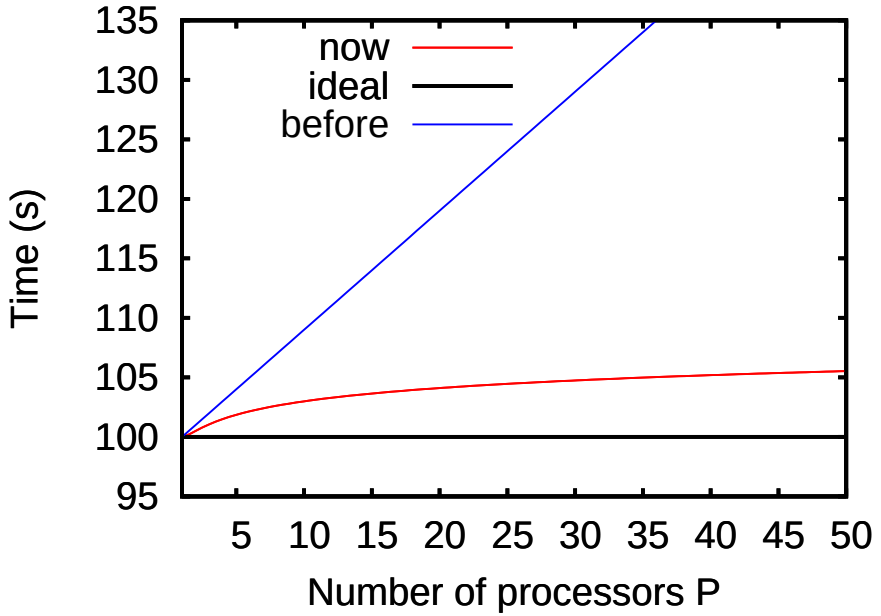


Trying to beat Amdahl's law #2

Weak scaling

$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

Should approach constant for large **P**.
Let's see...
Not quite!
But much better than before.



Gustafson?

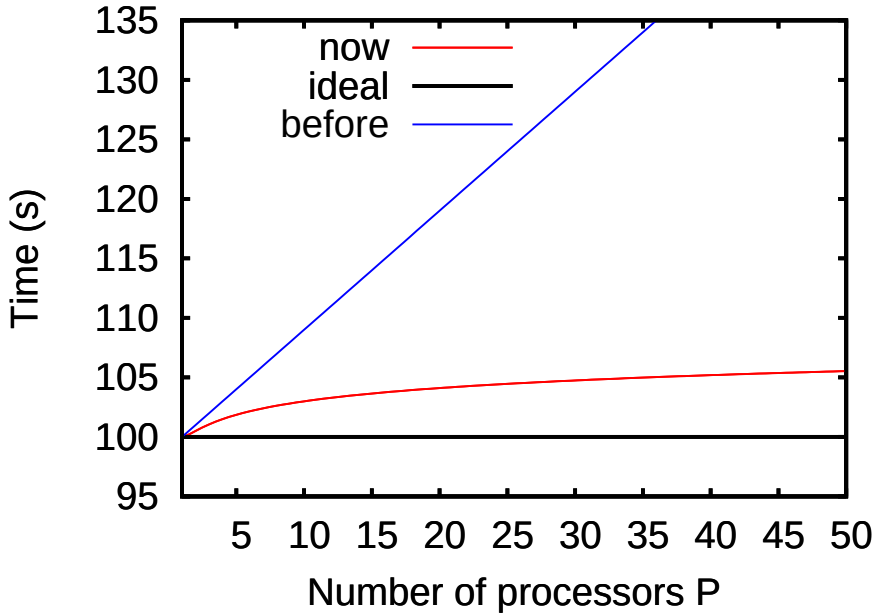
It turns out that Gustafson's law assumes that the serial cost does not change with **P**.
Here that grows logarithmically with **P**, and this is reflected in the weak scaling.

Trying to beat Amdahl's law #2

Weak scaling

$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

Should approach constant for large **P**.
Let's see...
Not quite!
But much better than before.



Really not that bad.
and other algorithms can do better.

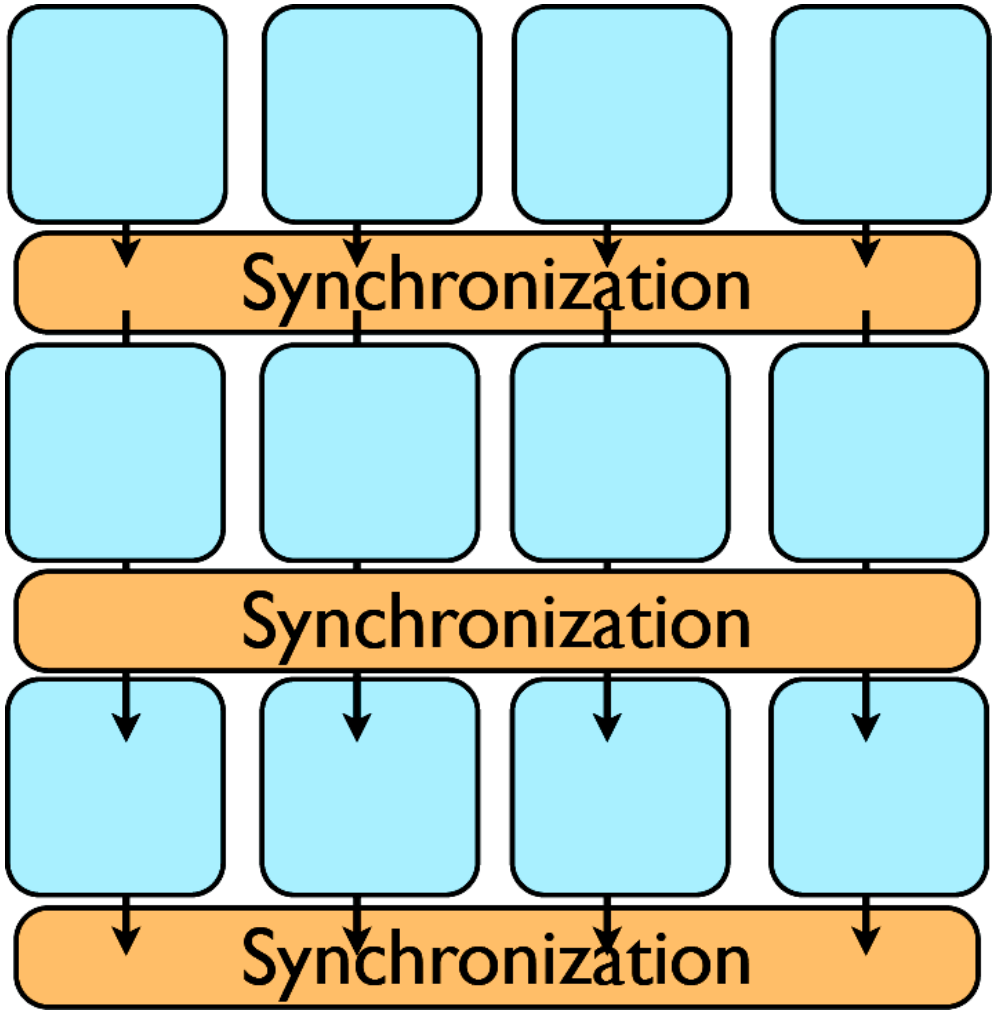
Gustafson?
It turns out that Gustafson's law assumes that the serial cost does not change with **P**.
Here that grows logarithmically with **P**, and this is reflected in the weak scaling.

Big Lesson #2

Optimal Serial Algorithm for your problem may not be the P $\longrightarrow$1 limit of your optimal parallel algorithm.

Synchronization

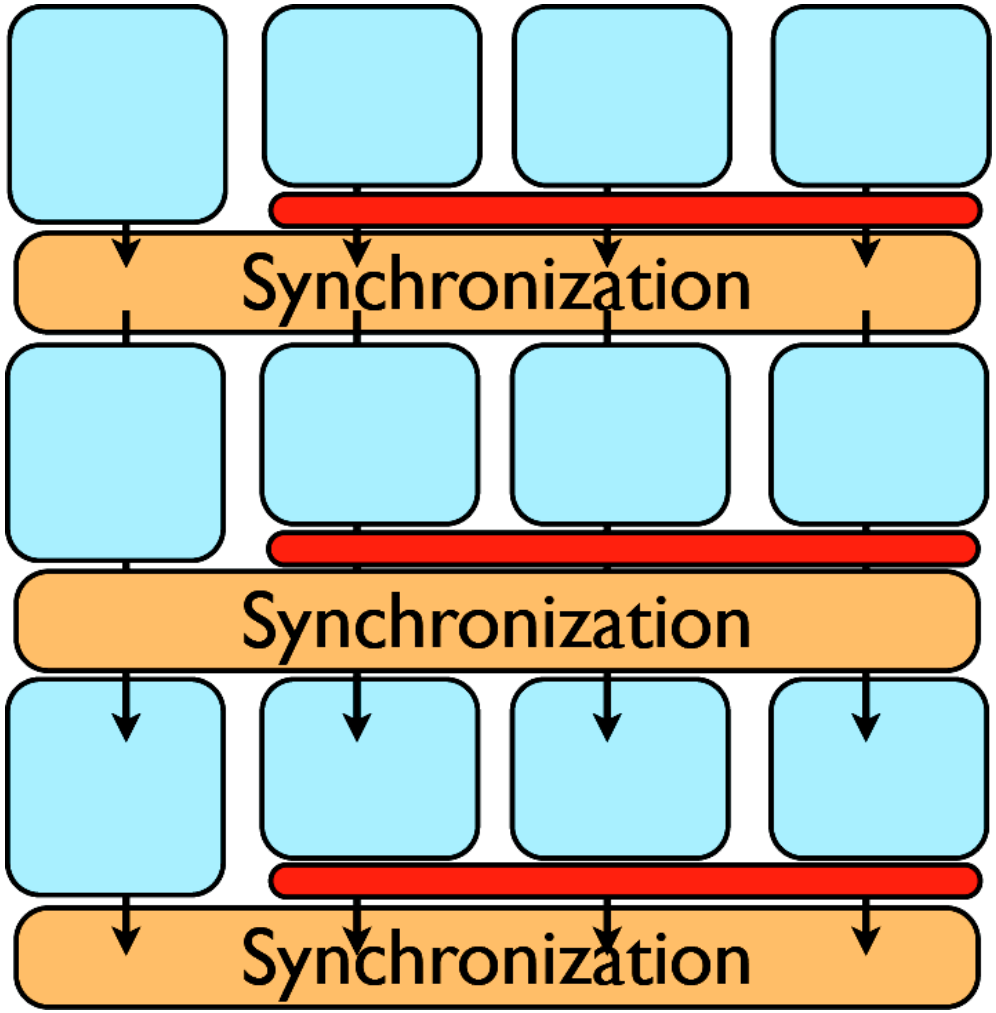
- Most problems are not purely concurrent.
- Some level of synchronization or exchange of information is needed between tasks.
- While synchronizing, nothing else happens: increases Amdahl's **f**.
- And synchronizations are themselves costly.



SciNet compute • calcul CANADA

Load balancing

- The division of calculations among the processors may not be equal.
- Some processors would already be done, while others are still going.
- Effectively using less than **P** processors: This reduces the efficiency.
- Aim for load balanced algorithms.



Locality

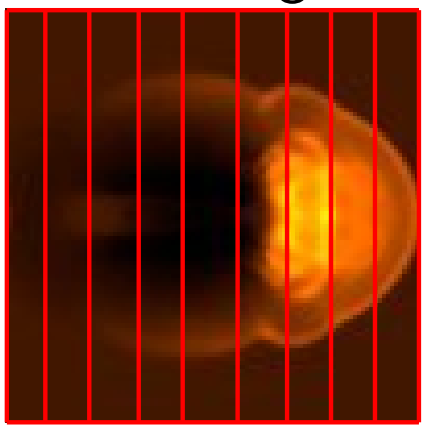
- So far we neglected communication costs.
- But communication costs are more expensive than computation!
- To minimize communication to computation ratio:
  * Keep the data where it is needed.
  * Make sure as little data as possible is to be communicated.
  * Make shared data as local to the right processors as possible.
- Local data means less need for syncs, or smaller-scale syncs.
- Local syncs can alleviate load balancing issues.

Locality

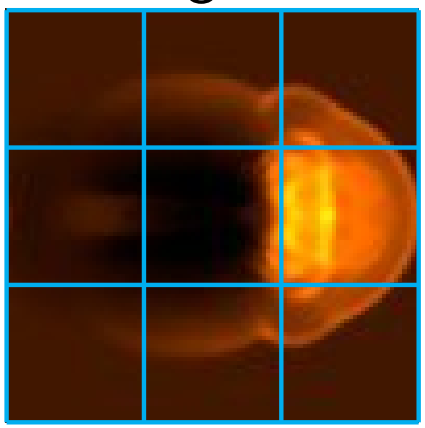
- So far we neglected communication costs.
- But communication costs are more expensive than computation!
- To minimize communication to computation ratio:
  * Keep the data where it is needed.
  * Make sure as little data as possible is to be communicated.
  * Make shared data as local to the right processors as possible.
- Local data means less need for syncs, or smaller-scale syncs.
- Local syncs can alleviate load balancing issues.

Example (PDE Domain decomposition)

wrong



right



Big Lesson #3

Parallel algorithm design is about finding as much concurrency as possible, and arranging it in a way that maximizes locality.

Parallel Computers

Top500.org:

List of the worlds 500 largest supercomputers. Updated every 6 months,

Info on architecture, etc.

Home ▸ Lists ▸ November 2010

TOP500 List - November 2010 (1-100)

R_{max} and R_{peak} values are in TFlops. For more details about other fields, check the TOP500 description.

Power data in KW for entire system

next

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
2	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
3	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	2580.00
4	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010 NEC/HP	73278	1192.00	2287.63	1398.61
5	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz / 2010 Cray Inc.	153408	1054.00	1288.63	2910.00
6	Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bullx super-node S6010/S6030 / 2010 Bull SA	138368	1050.00	1254.55	4590.00

Supercomputer architectures

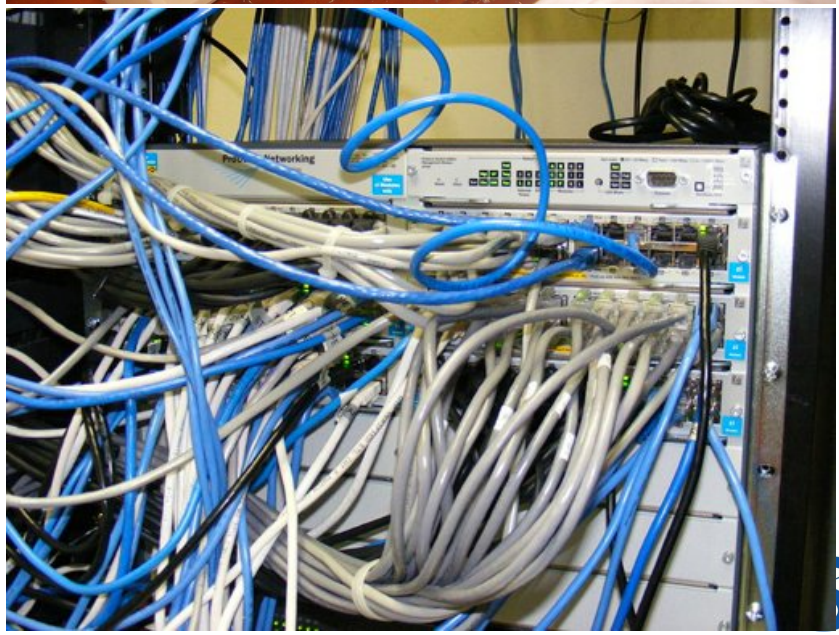
- Clusters, or, distributed memory machines
  In essence a bunch of desktops linked together by a network ("interconnect"). Easy and cheap.
- Multi-core machines, or, shared memory machines
  These can see the same memory. Limited number of cores, typically, and much more $$$.
- Vector machines.
  These were the early supercomputers, and could do the same operation on a large number of numbers at the same time.
  Very $$$$$, especially at scale.
  These days, most chips have some low-level, small size vectorization, but you rarely need to worry about it (compiler should do this).

Most supercomputers are a hybrid combo of these different architectures.

Distributed Memory: Clusters

Simplest type of parallel computer to build

- Take existing powerful standalone computers
- And network them



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

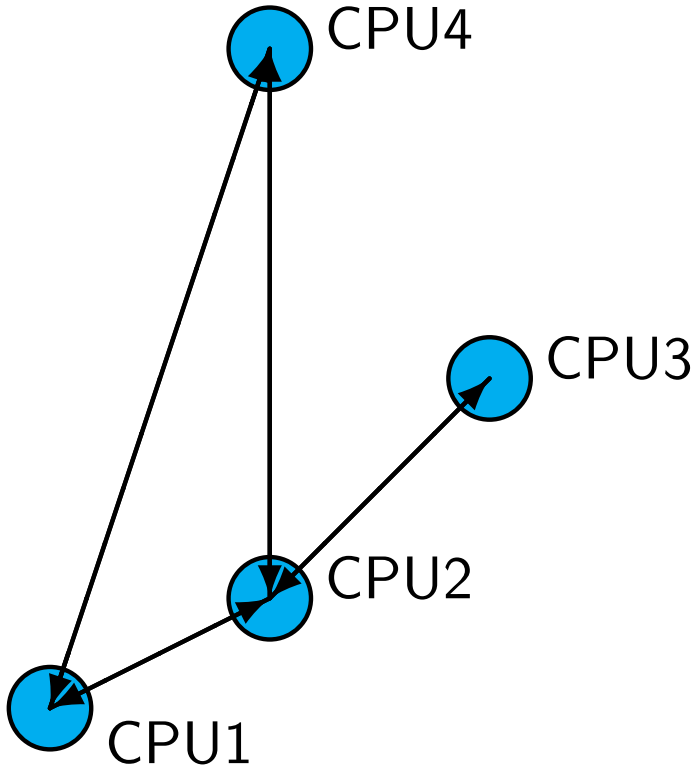
SciNet compute • calcul CANADA

Distributed Memory: Clusters

Each node is independent!

Parallel code consists of programs running on separate computers, communicating with each other.

Could be entirely different programs.

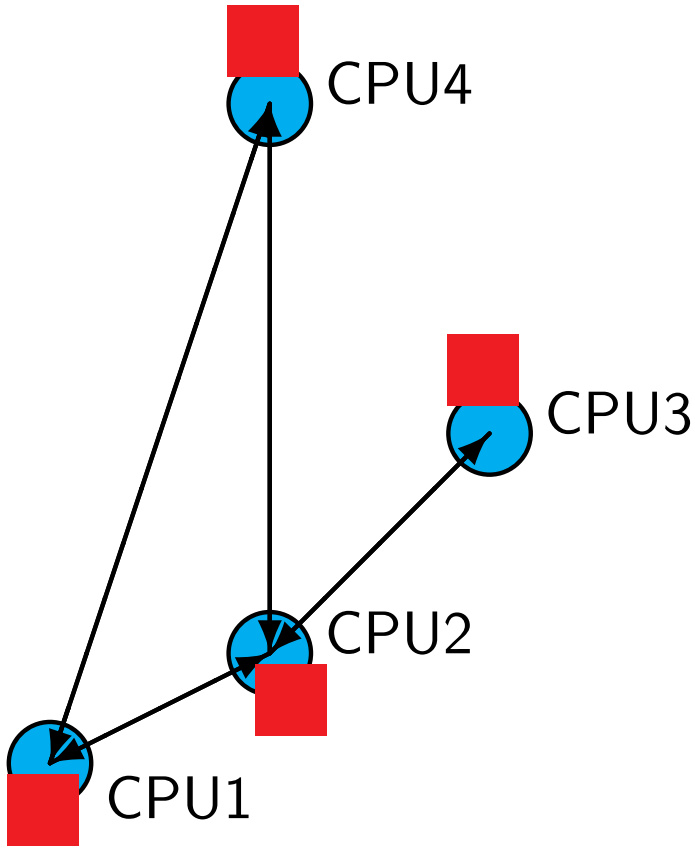


Distributed Memory: Clusters

Each node is independent!
Parallel code consists of programs running on separate computers, communicating with each other.
Could be entirely different programs.

Each node has own memory!
Whenever it needs data from another region, requests it from that CPU.

Usual model: "message passing"



Clusters+Message Passing

Hardware:

Easy to build

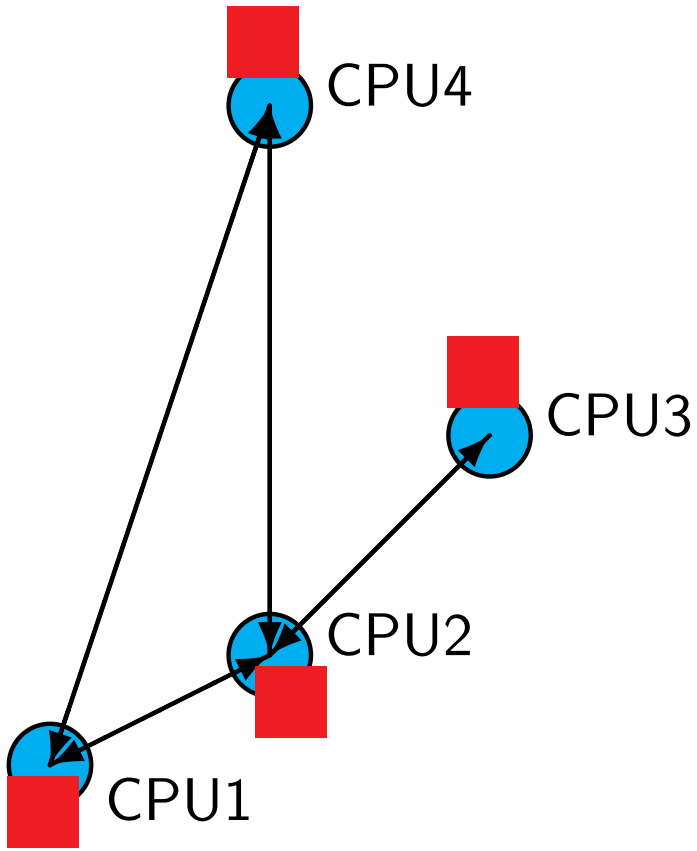
(Harder to build well)

Can build larger and larger clusters relatively easily

Software:

Every communication has to be hand-coded:

hard to program



Cluster Communication Cost

	Latency	Bandwidth
GigE	10 μs (10,000 ns)	1 Gb/s (60 ns/double)
Infiniband	2 μs (2,000 ns)	2-10 Gb/s (10 ns /double)

Processor speed: O(GFLOP) $\sim$ few ns or less.

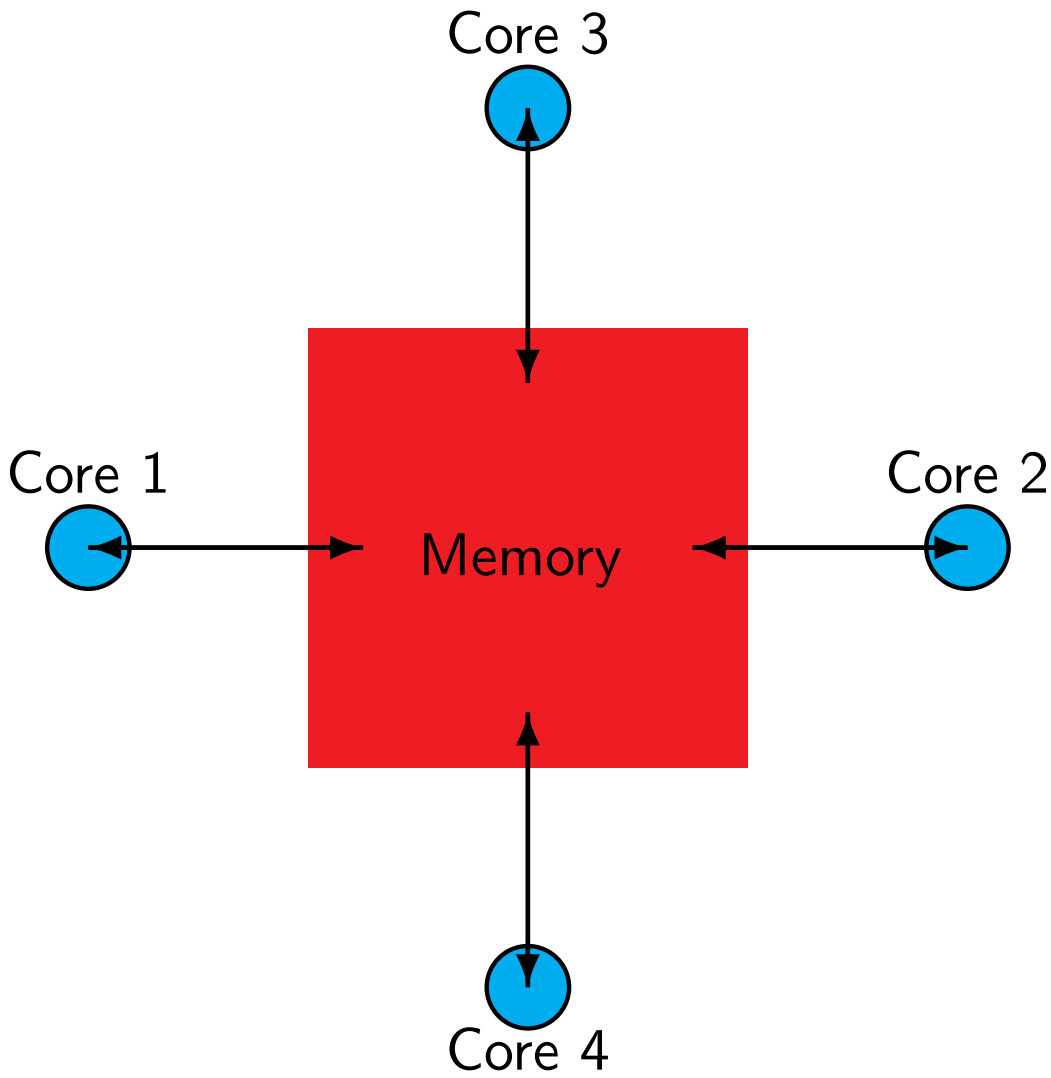
Shared Memory

One large bank of memory, different computing cores acting on it. All 'see' same data.

Any coordination done through memory

Could use message passing, but no need.

Each code is assigned a thread of execution of a single program that acts on the data.



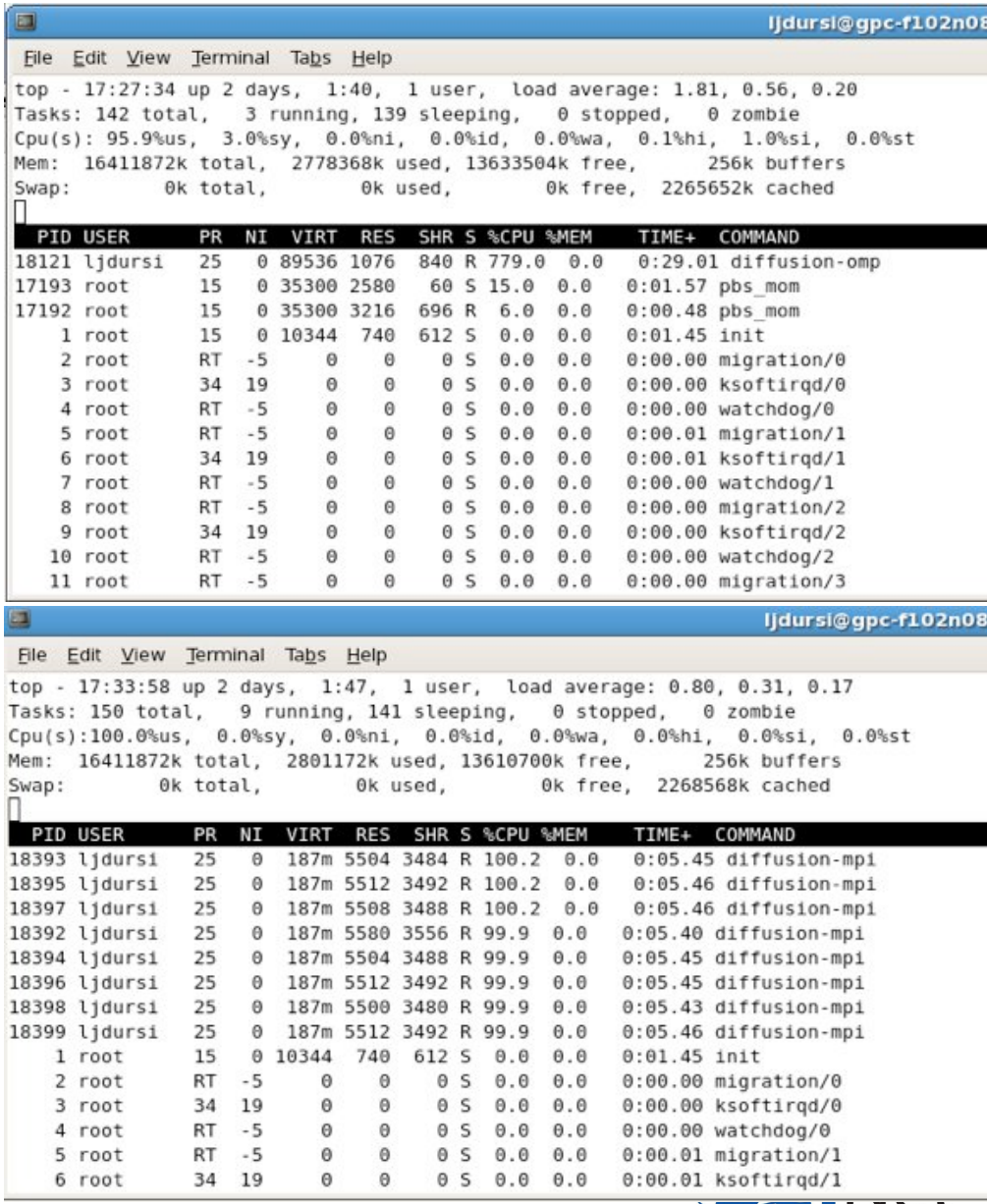
Threads versus Processes

Threads:

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

Processes:

Independent tasks with their own memory and resources

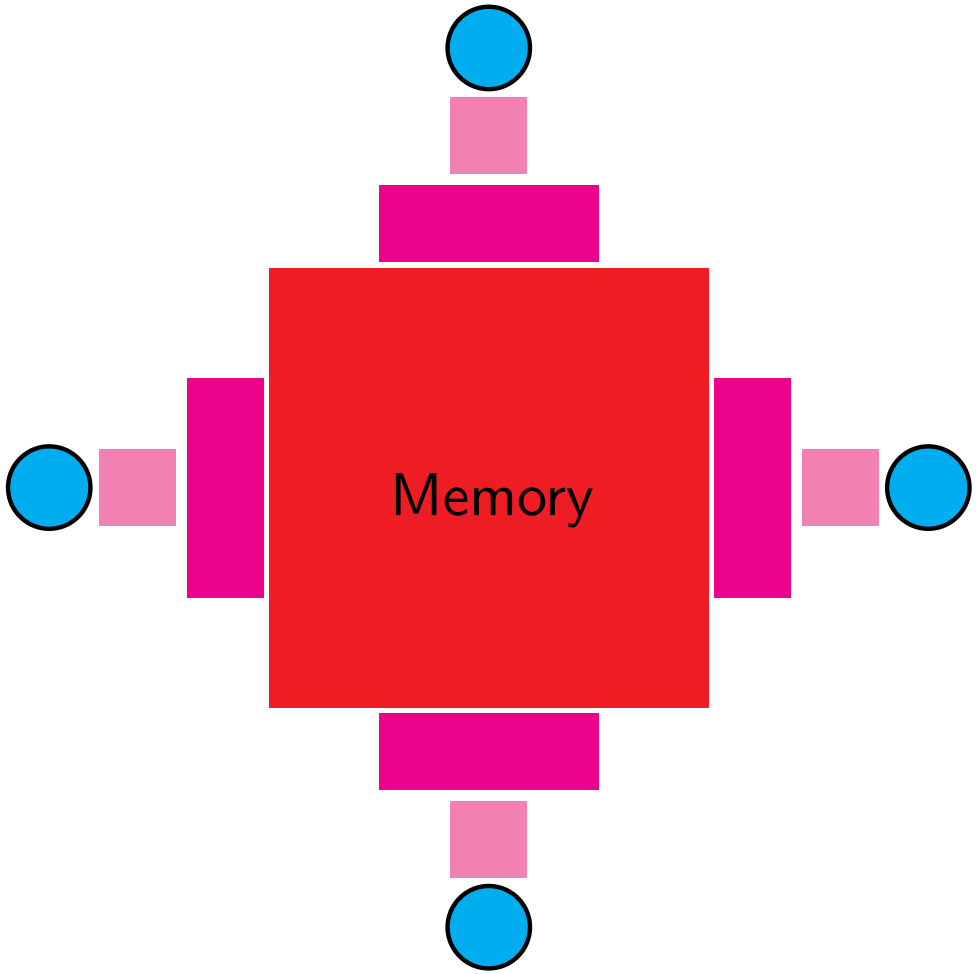


compute • calcul CANADA

Shared Memory: NUMA

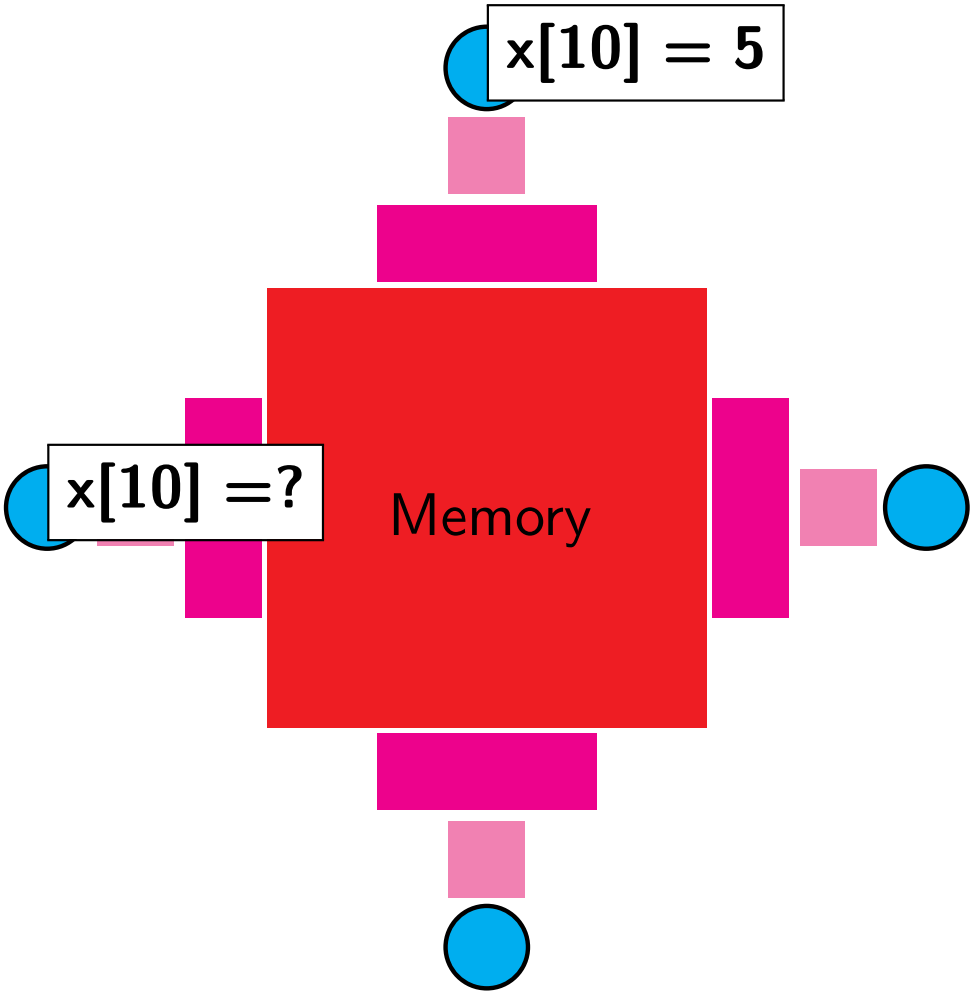
Non-Uniform Memory Access

- Each core typically has some memory of its own.
- Cores have cache too.
- Keeping this memory coherent is extremely challenging.



Coherency

- The different levels of memory imply multiple copies of some regions
- Multiple cores mean can update unpredictably
- Very expensive hardware
- Hard to scale up to lots of processors, very $$$
- Very simple to program!!



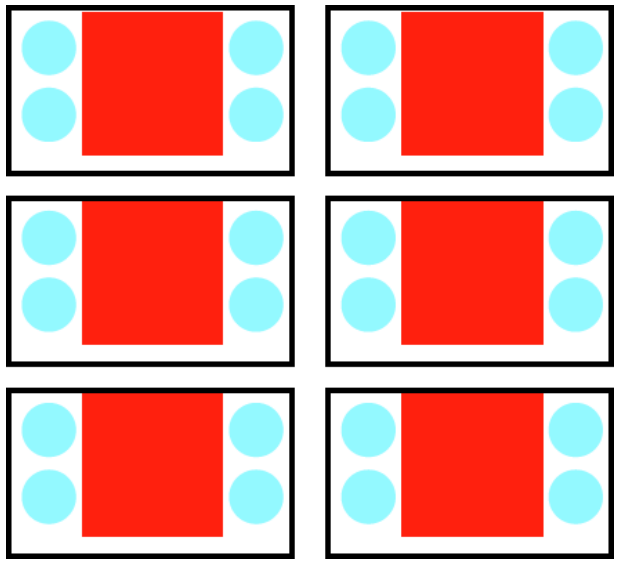
Shared Memory Communication Cost

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

Processor speed: O(GFLOP) $\sim$ few ns or less.

Hybrid Architectures

- Multicore machines linked together with an interconnect
- Many cores have modest vector capabilities.
- Machines with GPU: GPU is multi-core, but the amount of shared memory is limited.



We will focus on the aspects that affect the programmer:

- Shared memory: OpenMP
- Distributed memory: MPI
- Graphics computing: CUDA, OpenCL

SciNet compute • calcul CANADA

Big Lesson #4

The best approach to parallelizing your problem will depend on both details of your problem and of the hardware available.