

# High Performance Computing (HPC) Introduction

Ontario Summer School on  
High Performance Computing

Scott Northrup  
SciNet HPC Consortium  
Compute Canada

May 7th, 2013

# Outline

- 1 HPC Overview
- 2 Parallel Computing
  - Amdahl's law
  - HPC Lesson #1
  - Beating Amdahl's law
  - HPC Lesson #2
  - Load balancing
  - Locality
  - HPC Lesson #3
- 3 HPC Hardware
  - Distributed Memory
  - Shared Memory
  - Hybrid Architectures
  - HPC Lesson #4
- 4 HPC Programming Models
- 5 GNU Parallel
- 6 HPC System Software

## Contributing Material

- SciNet Parallel Scientific Computing Course  
- L. J. Dursi & R. V. Zon, SciNet
- Parallel Computing Models - D. McCaughan, SHARCNET
- High Performance Computing - D. McCaughan, SHARCNET
- HPC Architecture Overview - H. Merez, SHARCNET
- Intro to HPC - T. Whitehead, SHARCNET

# Scientific High Performance Computing

## What is it?

HPC is essentially leveraging larger and/or multiple computers to solve computations in parallel.

# Scientific High Performance Computing

## What is it?

HPC is essentially 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

# Scientific High Performance Computing

## What is it?

HPC is essentially 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 HPC?

- My problem takes to long → more/faster computation
- My problem is to big → more memory
- My data is to big → more storage

## Why is it necessary?

- Modern experiments and observations yield vastly more data to be processed than in the past.
- As more computing resources become available, the bar for cutting edge simulations is raised.
- Science that could not have been done before becomes tractable.

# Scientific High Performance Computing

## Why is it necessary?

- Modern experiments and observations yield vastly more data to be processed than in the past.
- As more computing resources become available, the bar for cutting edge simulations is raised.
- Science that could not have been done before becomes tractable.

## However

- Advances in clock speeds, bigger and faster memory and storage have been lagging as compared to e.g. 10 years ago.  
*Can no longer “just wait a year” and get a better computer.*
- So modern HPC means more hardware, not faster hardware.
- Thus parallel programming/computing is required.



## HR Dilemma

- Problem: job needs to get done faster

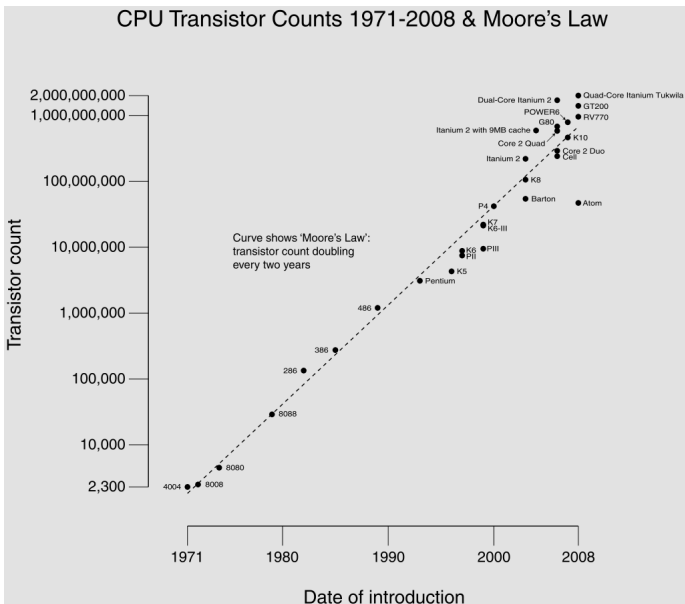
## HR Dilemma

- Problem: job needs to get done faster
  - can't hire substantially faster people
  - can hire more people

## HR Dilemma

- Problem: job needs to get done faster
  - can't hire substantially faster people
  - can hire more people
- Solution:
  - split work up between people (divide and conquer)
  - requires rethinking the work flow process
  - requires administration overhead
  - eventually administration larger than actual work

# Wait, what about Moore's Law?



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

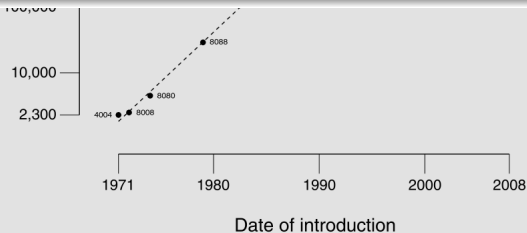
# Wait, what about Moore's Law?

## CPU Transistor Counts 1971-2008 & 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?

CPU Transistor Counts 1971-2008 & 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 clock speed.*
- *More transistors but getting hard to push clockspeed up. Power density is limiting factor.*
- *So more cores at fixed clock speed.*

## Thinking Parallel

The general idea is if one processor is good, many processors will be better

- Parallel programming is not generally trivial
- Tools for automated parallelism are either highly specialized or absent
- serial algorithms/mathematics don't always work well in parallel without modification

## Thinking Parallel

The general idea is if one processor is good, many processors will be better

- Parallel programming is not generally trivial
- Tools for automated parallelism are either highly specialized or absent
- serial algorithms/mathematics don't always work well in parallel without modification

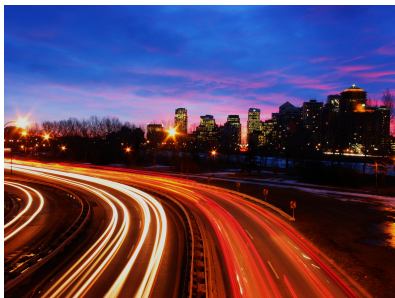
## Parallel Programming

- its Necessary (serial performance has peaked)
- its Everywhere (cellphones, tablets, laptops, etc)
- its Only getting worse (Sequoia has 1.5 Million cores)



# Concurrency

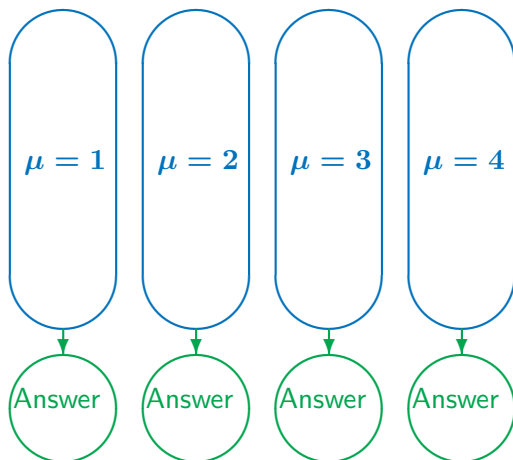
- Must have something to do for all these cores.
- Find parts of the program that can be done independently, and therefore concurrently.
- There must be many such parts.
- Their order of execution should not matter either.
- **Data dependencies limit concurrency.**



(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.

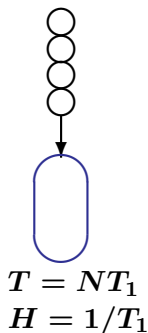


# Throughput

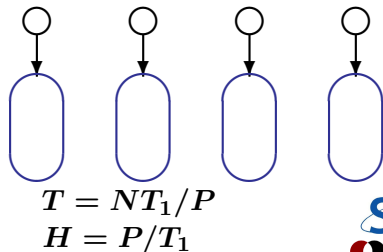
- How many tasks can you do per time unit?

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

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



vs.

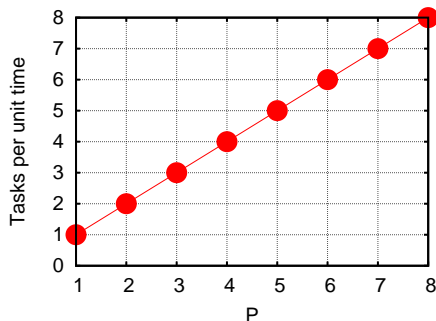


# Scaling — Throughput

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

$$H \propto P$$

- This is **Perfect scaling**.

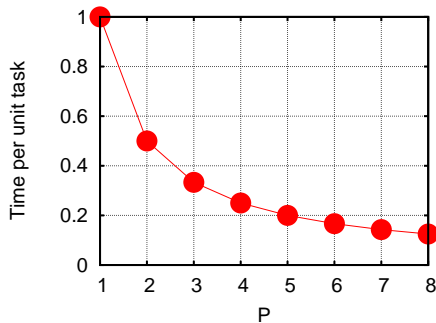


# Scaling – Time

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

$$T \propto 1/P$$

- Again this is the ideal case, or “embarrassingly parallel”.

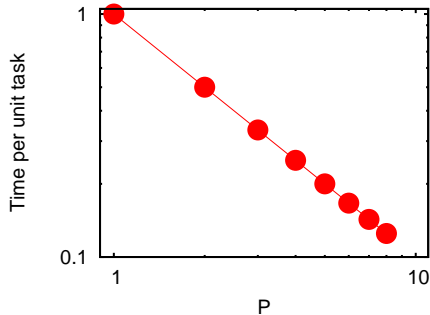


# Scaling – Time

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

$$T \propto 1/P$$

- Again this is the ideal case, or “embarrassingly parallel”.

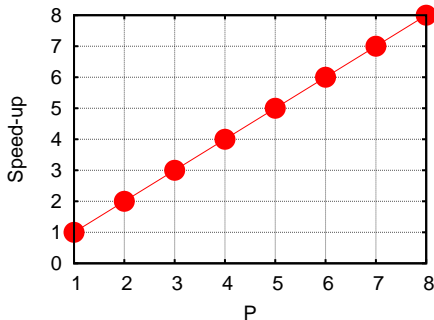


# Scaling – Speedup

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

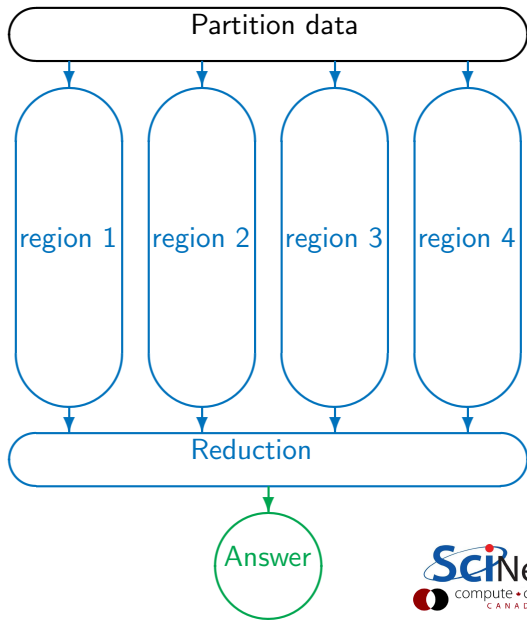
$$S = \frac{T_{serial}}{T(P)} \propto P$$

- For embarrassingly parallel applications: Linear speed up.



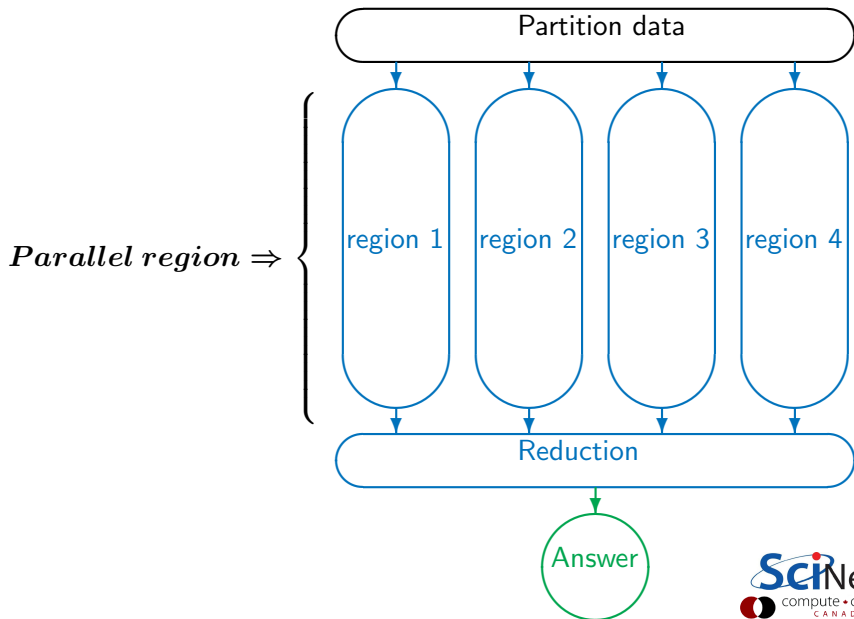
# Non-ideal cases

- 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”

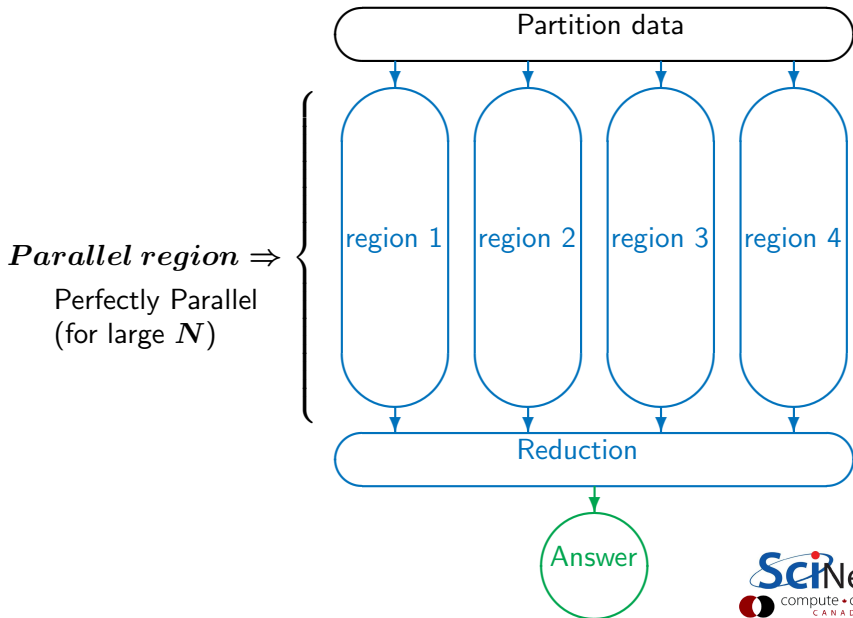




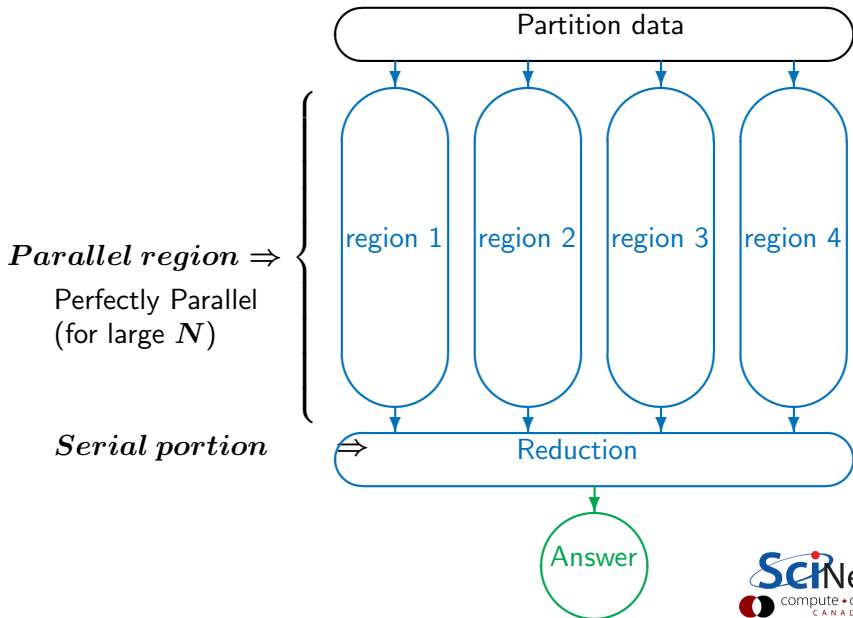
# Non-ideal cases



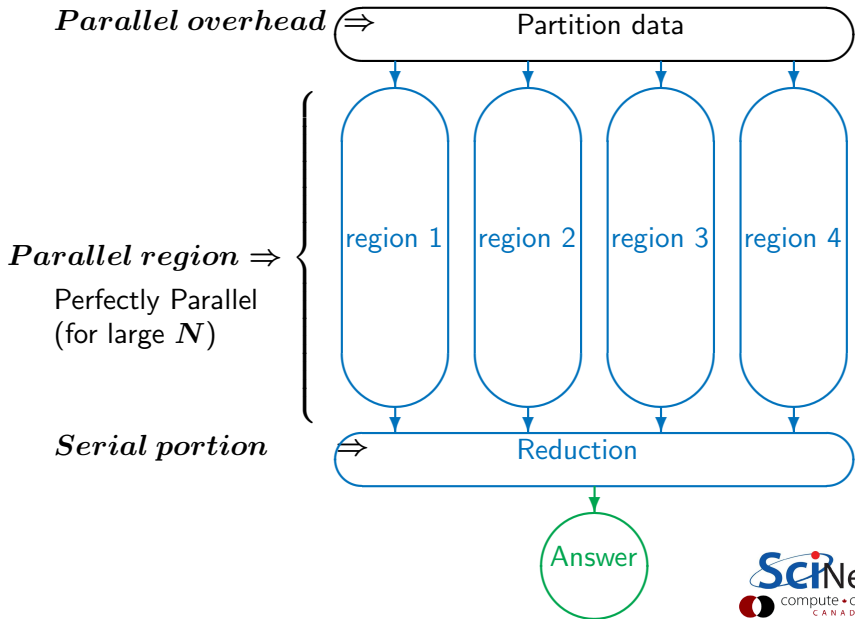
# Non-ideal cases



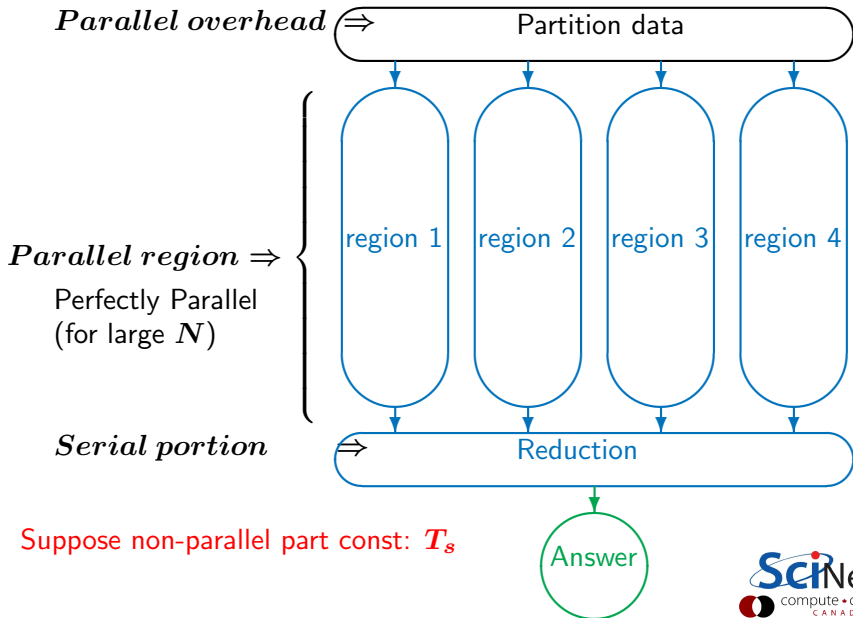
# Non-ideal cases



# Non-ideal cases



# Non-ideal cases



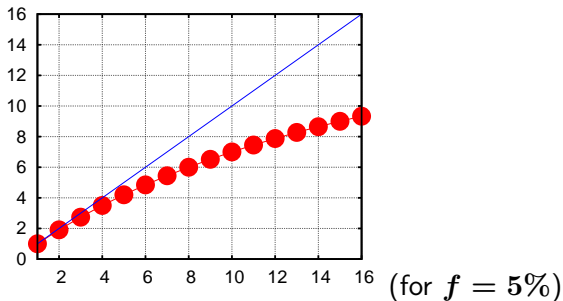
# Amdahl's law

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,

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



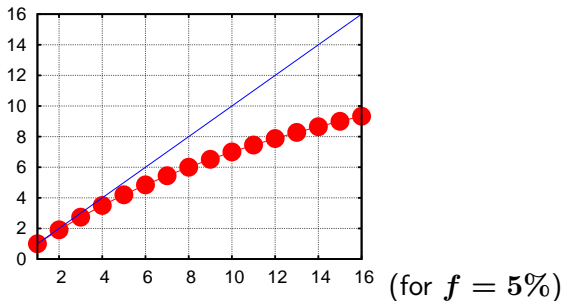
# Amdahl's law

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,

$$S = \frac{1}{f + (1 - f)/P} \quad \begin{matrix} P \rightarrow \infty \\ \longrightarrow \end{matrix} \quad \frac{1}{f}$$



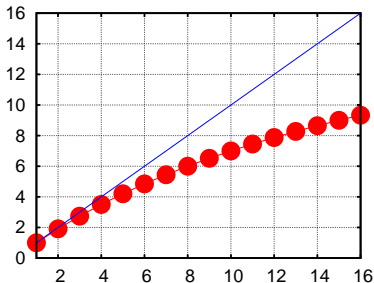
# Amdahl's law

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,

$$S = \frac{1}{f + (1 - f)/P} \quad \begin{matrix} P \rightarrow \infty \\ \longrightarrow \end{matrix} \quad \frac{1}{f}$$



Serial part dominates asymptotically.

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

(for  $f = 5\%$ )



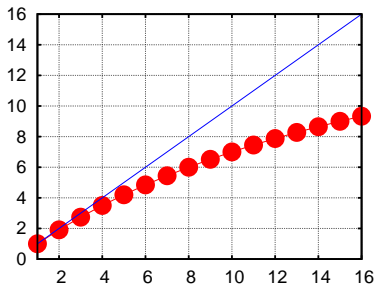
# Amdahl's law

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,

$$S = \frac{1}{f + (1 - f)/P} \quad \begin{matrix} P \rightarrow \infty \\ \longrightarrow \end{matrix} \quad \frac{1}{f}$$



Serial part dominates asymptotically.

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

And this is the overly optimistic case!

(for  $f = 5\%$ )

Speed-up compared to ideal factor  $P$ :

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

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

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

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

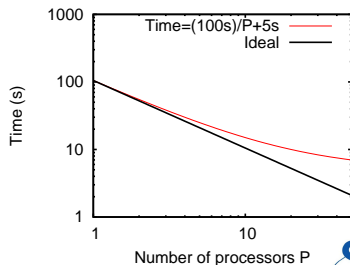
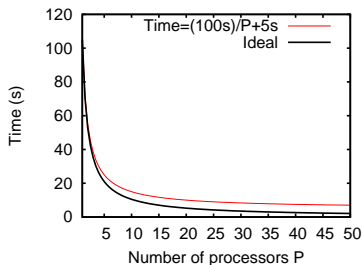
All you can aim for here is to make the efficiency as least low as possible.

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  $P$ ?

$$Time = (100s)/P + 5$$



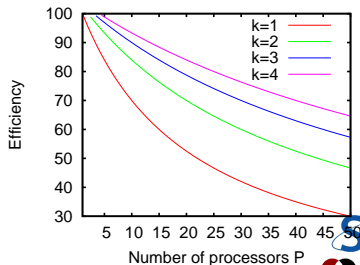
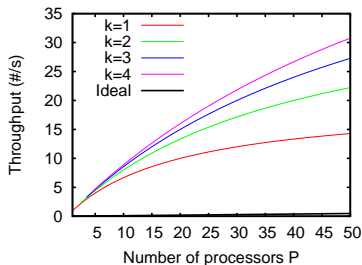
# Throughput example

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

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

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

Say  $N = 100$ :



## HPC 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:  $N = 100$ ,  $T_1 = 1s \dots$

## 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

$$T_s \approx PT_1$$

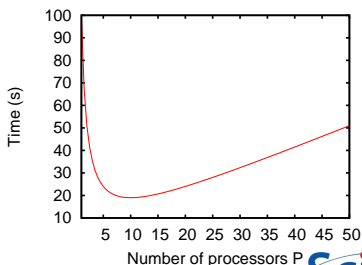
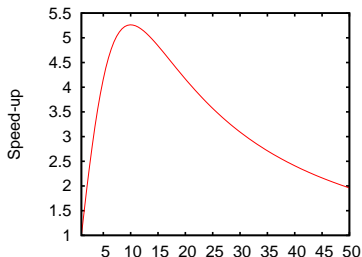
Serial fraction now a function of  $P$ :

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

Amdahl:

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

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

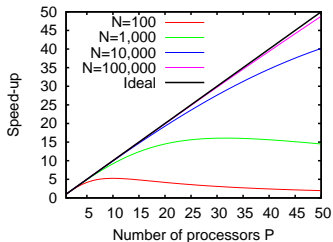




## Scale up!

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

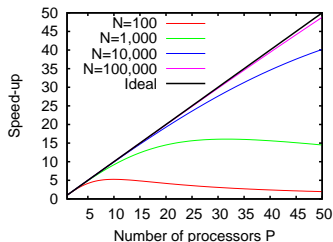
$$f(P) = \frac{P}{N}$$



## Scale up!

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

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



Weak scaling: Increase problem size while increasing  $P$

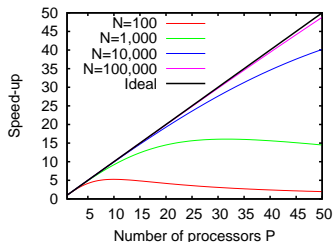
$$Time_{weak}(P) = Time(N = n \times P, P)$$

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

## Scale up!

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

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



Weak scaling: Increase problem size while increasing  $P$

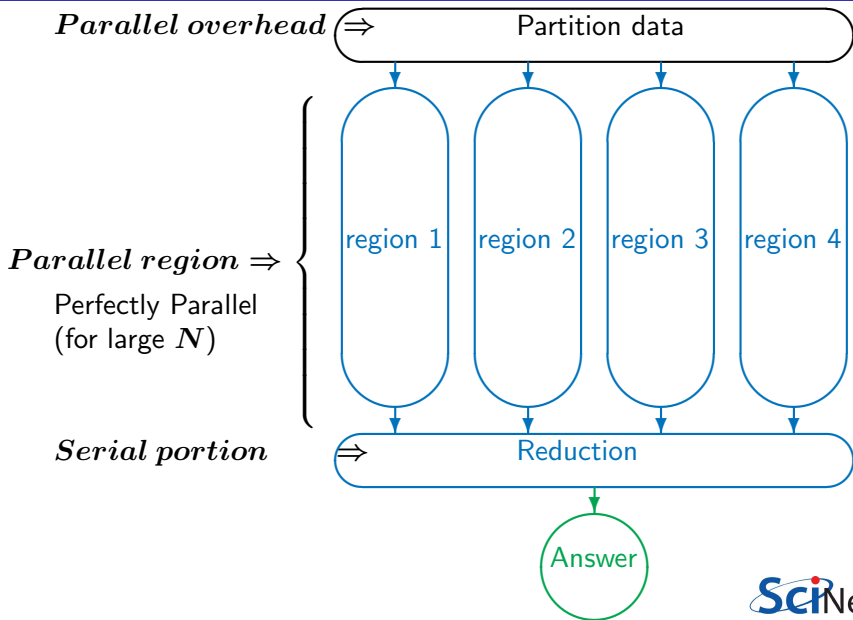
$$Time_{weak}(P) = Time(N = n \times P, P)$$

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

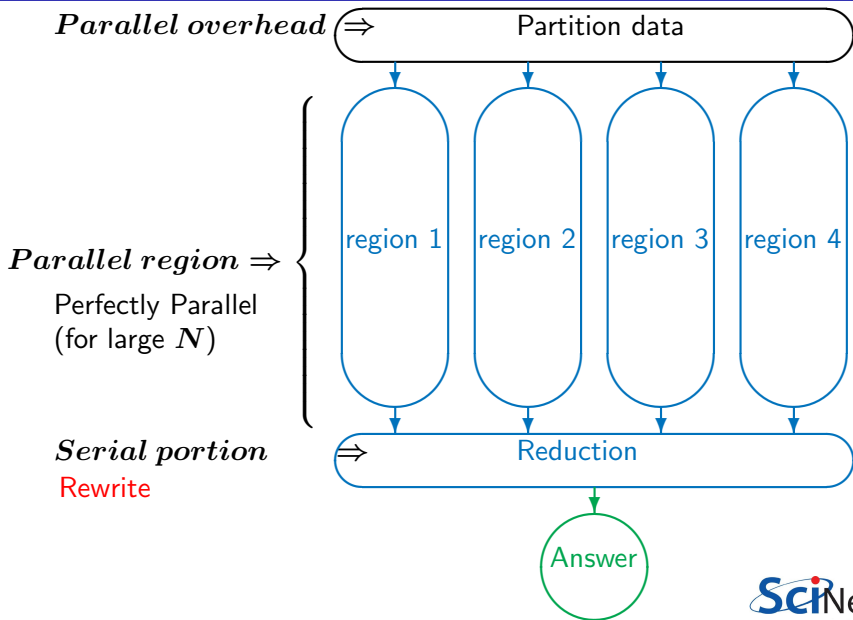
## Gustafson's Law

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

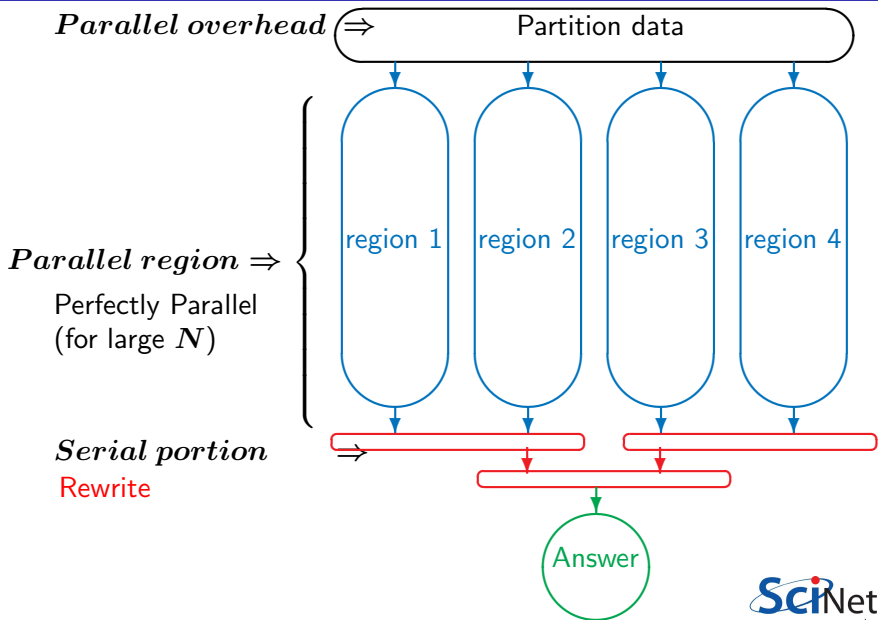
# Trying to beat Amdahl's law #2



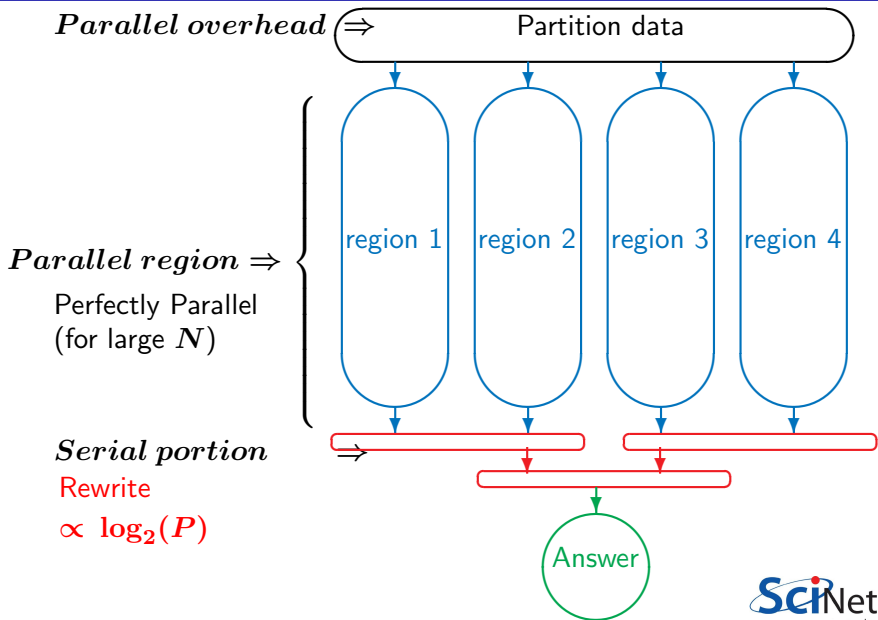
# Trying to beat Amdahl's law #2



# Trying to beat Amdahl's law #2



# Trying to beat Amdahl's law #2



## Trying to beat Amdahl's law #2

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

$$f(P) = \frac{\log_2(P)}{N}$$

Amdahl:

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



## Trying to beat Amdahl's law #2

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

$$f(P) = \frac{\log_2(P)}{N}$$

Amdahl:

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

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

# Trying to beat Amdahl's law #2

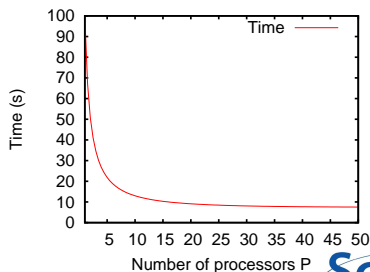
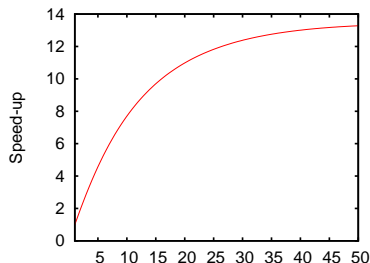
'Serial' fraction now different function of  $P$ :

$$f(P) = \frac{\log_2(P)}{N}$$

Amdahl:

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

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



# Trying to beat Amdahl's law #2

## Weak Scaling

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

Should approach constant for  
large  $P$ .

Let's see...

# Trying to beat Amdahl's law #2

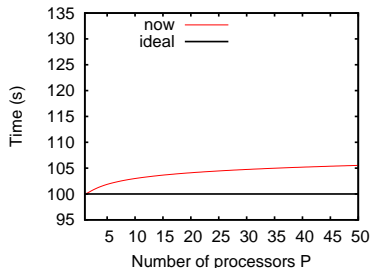
## Weak Scaling

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

Should approach constant for large  $P$ .

Let's see...

Not quite!



# Trying to beat Amdahl's law #2

## Weak Scaling

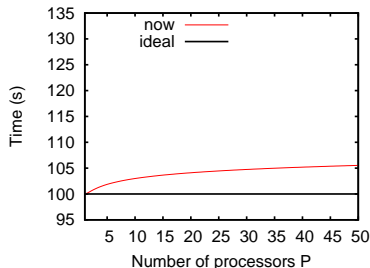
$$Time_{weak}(P) = Time(N = n \times P, P)$$

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

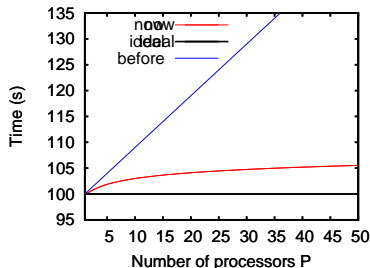
$$Time_{weak}(P) = Time(N = n \times P, P)$$

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

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

Should approach constant for large  $P$ .

Let's see...

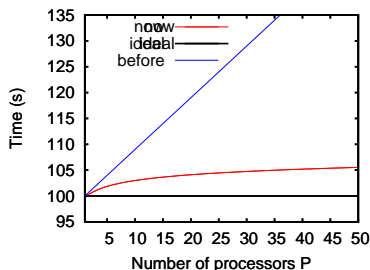
Not quite!

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

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

Should approach constant for large  $P$ .

Let's see...

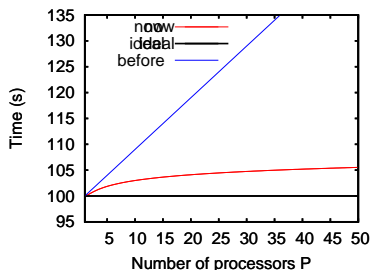
Not quite!

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.



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

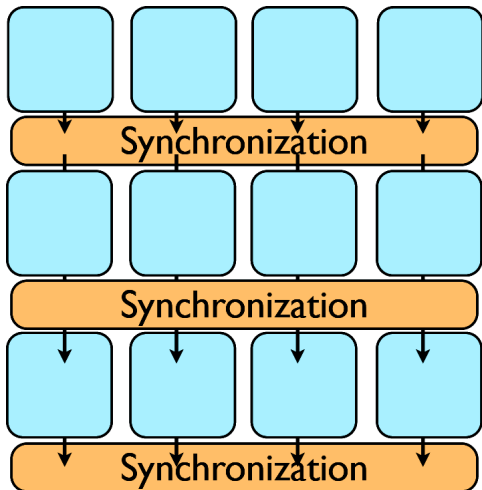


## HPC Lesson #2

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

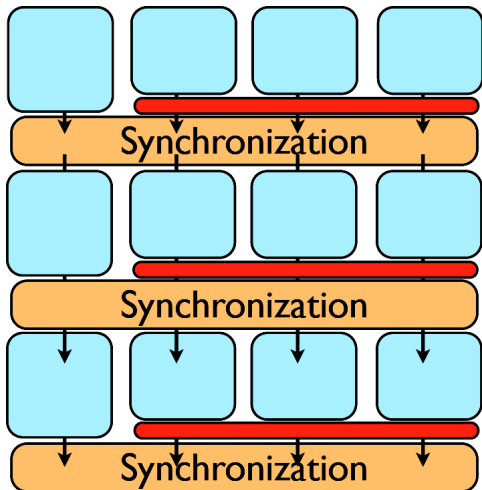
# Synchronization

- 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.



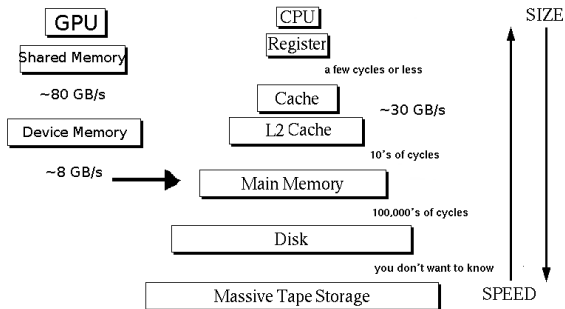
# Load balancing

- 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.



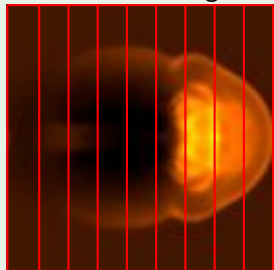
- So far we neglected communication costs.
- But communication costs are more expensive than computation!
- To minimize communication to computation ratio:
  - \* Keep the data where it is needed.
  - \* Make sure as little data as possible is to be communicated.
  - \* Make shared data as local to the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

# Locality

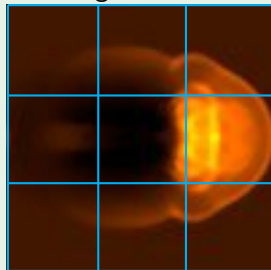


## Example (PDE Domain decomposition)

wrong



right



## HPC Lesson #3

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

## Top500.org:

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

Info on  
architecture, etc.

PROJECT	LISTS	STATISTICS	RESOURCES	NEWS
---------	-------	------------	-----------	------

[Home](#) » [Lists](#) » [June 2012](#)

### TOP500 List - June 2012 (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	DOE/INSAILLNL United States	<b>Sequoia</b> - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom / 2011 IBM	1572864	16324.75	20132.66	7890.0
2	RIKEN Advanced Institute for Computational Science (AICS) Japan	<b>K computer</b> , SPARC64 VIIIlx 2.0GHz, Tofu interconnect / 2011 Fujitsu	705024	10510.00	11280.38	12659.9
3	DOE/SC/Argonne National Laboratory United States	<b>Mira</b> - BlueGene/Q, Power BQC 16C 1.90GHz, Custom / 2012 IBM	786432	8162.38	10066.33	3945.0
4	Leibniz Rechenzentrum Germany	<b>SuperMUC</b> - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR / 2012 IBM	147456	2897.00	3185.05	3422.7
5	National Supercomputing Center in Tianjin China	<b>Tianhe-1A</b> - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 / 2010 NUDT	186368	2566.00	4701.00	4040.0
6	DOE/SC/Oak Ridge National Laboratory United States	<b>Jaguar</b> - Cray XK6, Opleron 6274 19C 2.20GHz, Cray Gemini interconnect, NVIDIA 2090 / 2009 Cray Inc.	298592	1941.00	2627.61	5142.0
7	CINECA Italy	<b>Fermi</b> - BlueGene/Q, Power BQC 16C 1.90GHz, Custom / 2012 IBM	163840	1725.49	2097.15	821.9
8	Forschungszentrum Juelich (FZJ) Germany	<b>JuQUEEN</b> - BlueGene/Q, Power BQC 16C 1.90GHz, Custom / 2012 IBM	131072	1380.39	1677.72	657.5
9	CEA/TGCC-GENCI France	<b>Curie thin nodes</b> - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR / 2012 Bull	77184	1359.00	1667.17	2251.0
10	National Supercomputing Centre in Shenzhen (NSCS) China	<b>Nebulae</b> - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 / 2010 Dawning	120640	1271.00	2984.30	2580.0



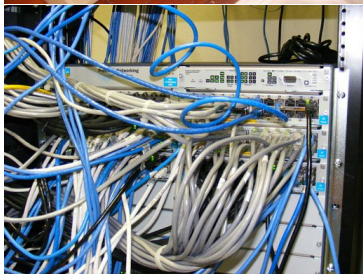
## Architectures

- Clusters, or, **distributed memory machines**
  - A bunch of servers linked together by a network (“interconnect”).
  - commodity x86 with gigE, Cray XK, IBM BGQ
- Symmetric Multiprocessor (SMP) machines, or, **shared memory machines**
  - These can see the same memory, typically Limited number of cores.
  - IBM Pseries, SGI Altix and Ultraviolet
- Vector machines.
  - No longer dominant in HPC anymore.
  - Cray, NEC
- Accelerator (GPU, Cell, MIC, FPGA)
  - Heterogeneous use of standard CPU’s with a specialized accelerator.
  - NVIDIA, AMD, Intel, Xilinx

# Distributed Memory: Clusters

Simplest type of parallel computer to build

- Take existing powerful standalone computers
- And network them



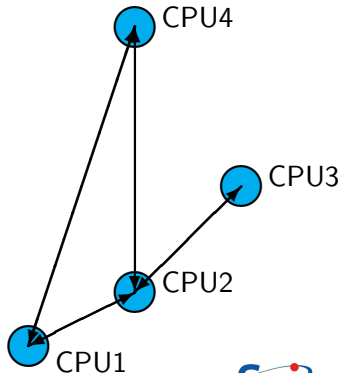
(source: <http://flickr.com/photos/eurleif>)

# 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!

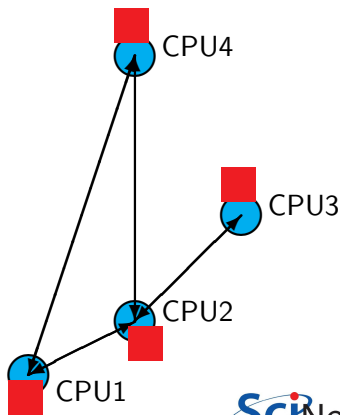
Parallel code consists of programs running on separate computers, communicating with each other.

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”



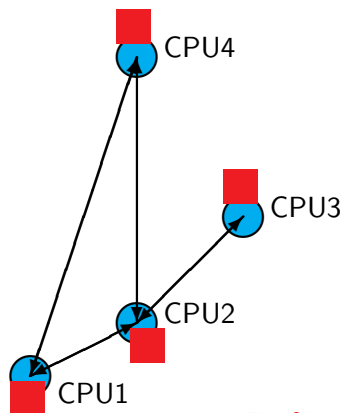
# Clusters+Message Passing

## 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



# Task (function, control) Parallelism

Work to be done is decomposed across processors

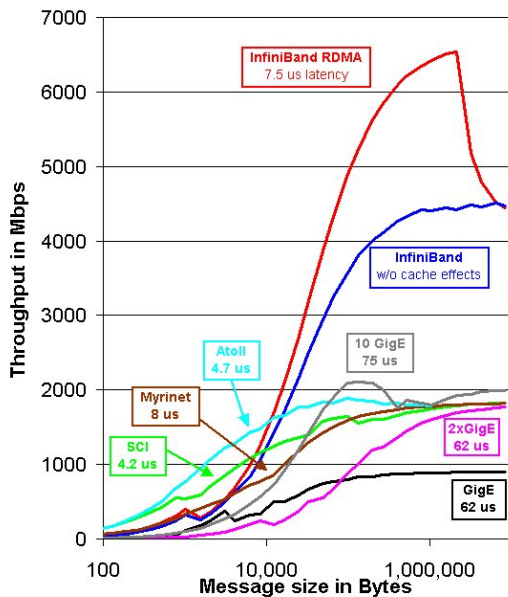
- e.g. divide and conquer
- each processor responsible for some part of the algorithm
- communication mechanism is significant
- must be possible for different processors to be performing different tasks

# Cluster Communication Cost

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

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

# Cluster Communication Cost





# SciNet General Purpose Cluster (GPC)



# SciNet General Purpose Cluster (GPC)

- 3864 nodes with two 2.53GHz quad-core Intel Xeon 5500 (*Nehalem*) x86-64 processors (30240 cores total)
- 16GB RAM per node
- Gigabit ethernet network on all nodes for management and boot
- DDR and QDR InfiniBand network on the nodes for job communication and file I/O
- 306 TFlops
- #16 on the June 2009 *TOP500* supercomputer sites (current #94)
- #1 in Canada

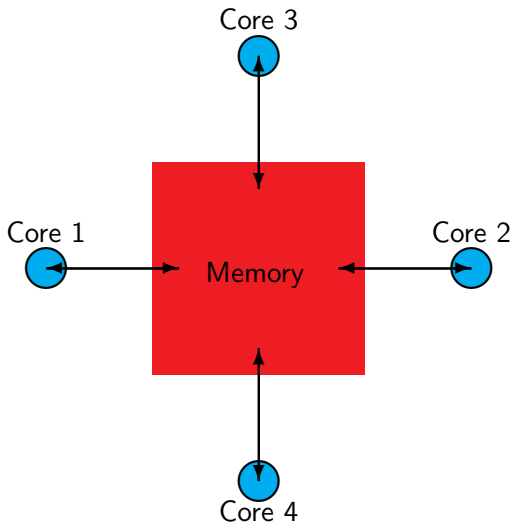
# Shared Memory

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

```
ljdursi@ gpc-f102n081:~
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

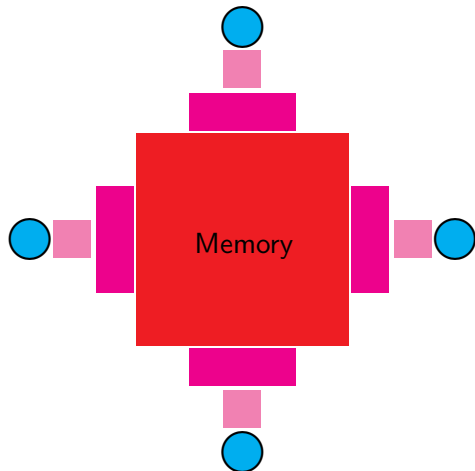
  PID USER      PR  NI  VIRT  RES  SHR  S %CPU  %MEM    TIME+  COMMAND
18121 ljdursi  25   0 89536 1076 848  R  779.0  0.0   0:29.01 diffusion-omp
17193 root     15   0 35300 2580  60  S  15.0  0.0   0:01.57 pbs_mom
17192 root     15   0 35300 3216 696  R  6.0  0.0   0:00.48 pbs_mom
  1 root     15   0 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
  7 root     RT -5   0   0   0  0  S  0.0  0.0   0:00.00 watchdog/1
  8 root     RT -5   0   0   0  0  S  0.0  0.0   0:00.00 migration/2
  9 root     34  19   0   0   0  S  0.0  0.0   0:00.00 ksoftirqd/2
 10 root     RT -5   0   0   0  0  S  0.0  0.0   0:00.00 watchdog/2
 11 root     RT -5   0   0   0  0  S  0.0  0.0   0:00.00 migration/3
```

```
ljdursi@ gpc-f102n081:~
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
  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
```

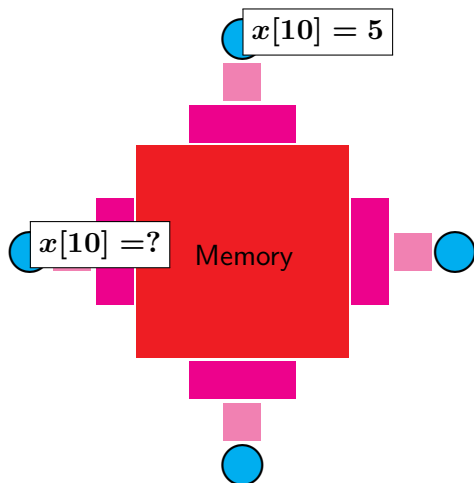
## Non-Uniform Memory Access

- 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 simple to program!!



# Data (Loop) Parallelism

Data is distributed across processors

- easier to program, compiler optimization
- code otherwise looks fairly sequential
- benefits from minimal communication overhead
- scale limitations

# Shared Memory Communication Cost

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

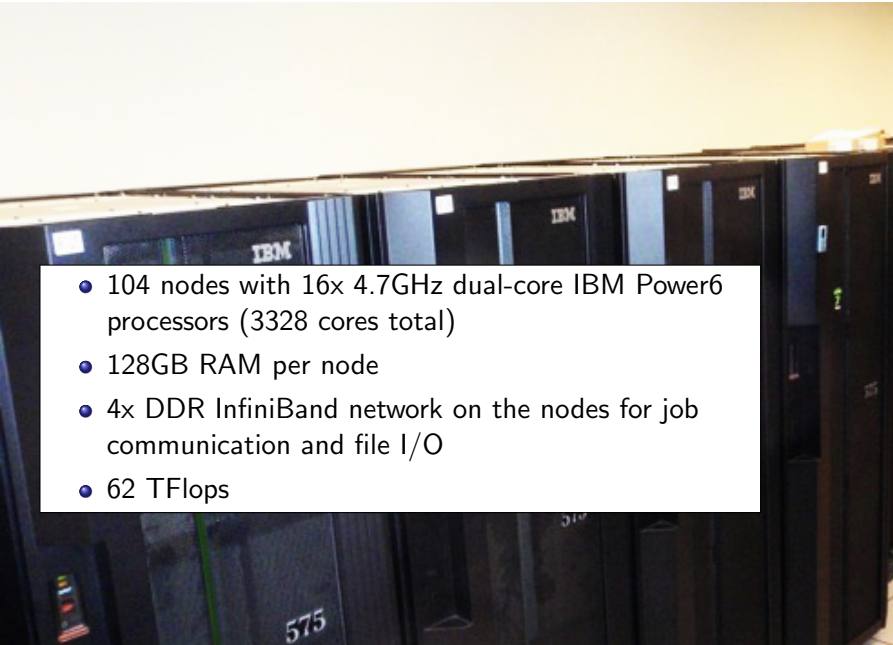
Processor speed:  $O(\text{GFLOP}) \sim \text{few ns or less.}$



# SciNet Tightly Coupled System (TCS)

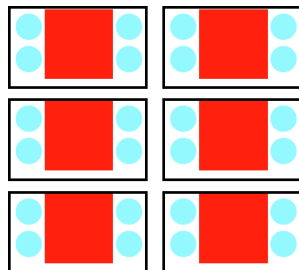


# SciNet Tightly Coupled System (TCS)

- 
- 104 nodes with 16x 4.7GHz dual-core IBM Power6 processors (3328 cores total)
  - 128GB RAM per node
  - 4x DDR InfiniBand network on the nodes for job communication and file I/O
  - 62 TFlops

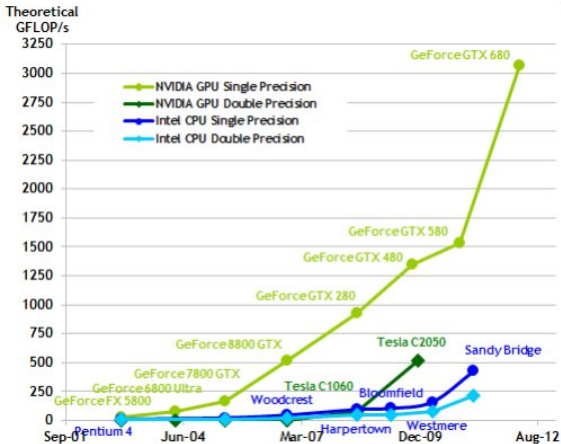
# Hybrid Architectures

- 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.



## Accelerators

- CPUs are not optimal for all algorithms and workloads, have to address many use cases.
- Special purpose co-processors (accelerators) can be connected to the CPU to handle particular tasks more efficiently.
- Long history of different accelerator architectures (and processors themselves) focusing on SIMD (single instruction, multiple data; vector) operations.
- Graphics processors (GPUs) have been extended such that they are flexible enough to handle these workloads (and they're relatively cheap).

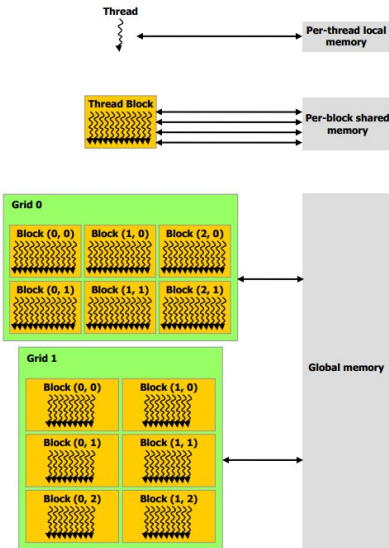


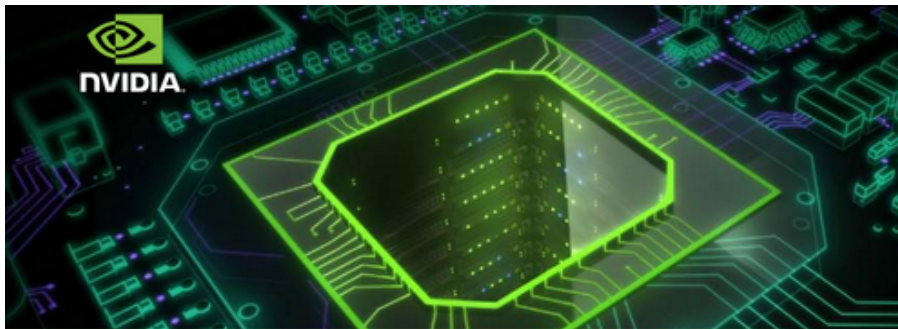
GPUs trade-off single thread performance and memory caching for SIMD parallel performance.

- GPUs communicate with the CPU via PCI Express bus.
- GPUs are driven by a host process running on the CPU, which invokes computational kernels.
- Extreme parallelism to get good speedup and hide latency, high arithmetic intensity.
- Data has to be explicitly sent back and forth to the GPU
- Fine-grained parallelism complements other parallel methods (MPI, threads, etc.)
- Programming is typically more involved (but tools and frameworks are constantly improving)

# GPGPU

- An array of streaming multiprocessors attached to a global memory.
- Computational kernels are executed using many threads, which are organized in a grid of thread blocks.
- Threads within a block have a pool of shared memory, as well as local, private memory.
- Focus on leveraging massive parallelism in algorithms while working within the constraints of limited shared-memory.





TESLA™ M2050 / M2070  
GPU COMPUTING MODULE  
SUPERCOMPUTING AT 1/10<sup>TH</sup> THE COST





- 54 nodes with two 2.26 GHz quad-core Intel Xeon E5607 (432 cores total)
- 48GB RAM per node
- 2x Nvidia M2070 Tesla GPU's per node
- QDR Infiniband interconnect

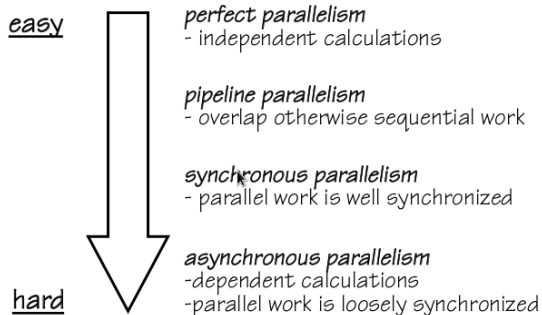
TESLA™ M2050 / M2070  
GPU COMPUTING MODULE  
SUPERCOMPUTING AT 1/10<sup>TH</sup> THE COST

## HPC Lesson #4

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

# Program Structure

Structure of the problem dictates the ease with which we can implement parallel solutions easy



## Granularity

A measure of the amount of processing performed before communication between processes is required.

## Parallelism

- Fine Grained
  - constant communication necessary
  - best suited to shared memory environments
- Coarse Grained
  - significant computation performed before communication is necessary
  - ideally suited to message-passing environments
- Perfect
  - no communication necessary

# Batching Serial Jobs

- SciNet is primarily a parallel computing resource. (Parallel here means OpenMP and MPI, not many serial jobs.)

# Batching Serial Jobs

- SciNet is primarily a parallel computing resource. (Parallel here means OpenMP and MPI, not many serial jobs.)
- You should never submit purely serial jobs to the GPC queue. The scheduling queue gives you a full 8-core node. Per-node scheduling of serial jobs would mean wasting 7 cpus.

# Batching Serial Jobs

- SciNet is primarily a parallel computing resource. (Parallel here means OpenMP and MPI, not many serial jobs.)
- You should never submit purely serial jobs to the GPC queue. The scheduling queue gives you a full 8-core node. Per-node scheduling of serial jobs would mean wasting 7 cpus.
- Nonetheless, if you can make efficient use of the resources using serial runs and get good science done, that's good too.

# Batching Serial Jobs

- SciNet is primarily a parallel computing resource. (Parallel here means OpenMP and MPI, not many serial jobs.)
- You should never submit purely serial jobs to the GPC queue. The scheduling queue gives you a full 8-core node. Per-node scheduling of serial jobs would mean wasting 7 cpus.
- Nonetheless, if you can make efficient use of the resources using serial runs and get good science done, that's good too.
- Users need to utilize whole nodes by running at least 8 serial runs at once.

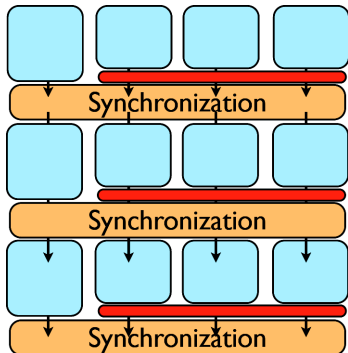


## Easy case: serial runs of equal duration

```
#PBS -l nodes=1:ppn=8,walltime=1:00:00
cd $PBS_O_WORKDIR
(cd rundir1; ./dorun1) &
(cd rundir2; ./dorun2) &
(cd rundir3; ./dorun3) &
(cd rundir4; ./dorun4) &
(cd rundir5; ./dorun5) &
(cd rundir6; ./dorun6) &
(cd rundir7; ./dorun7) &
(cd rundir8; ./dorun8) &
wait # or all runs get killed immediately
```

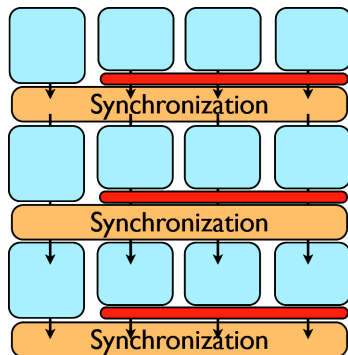
# Hard case: serial runs of unequal duration

Different runs may not take the same time: **load imbalance**.



# Hard case: serial runs of unequal duration

Different runs may not take the same time: **load imbalance**.



- Want to keep all 8 cores on a node busy.
- Or even 16 virtual cores on a node (HyperThreading).
- ⇒ GNU Parallel can do this

- GNU parallel is a tool to run multiple (serial) jobs in parallel. *As parallel is used within a GPC job, we'll call these **subjobs**.*
- It allows you to keep the processors on each 8-core node busy, if you provide enough subjobs.
- GNU Parallel can use multiple nodes as well.

On the GPC cluster:

- GNU parallel is accessible on the GPC in the module gnu-parallel, which you can load in your .bashrc.

```
$ module load gnu-parallel/20121022
```

- There are currently (Nov 2012) three gnu-parallel modules on the GPC. Although for compatibility gnu-parallel/2010 is the default, we recommend using gnu-parallel/20121022.

## SETUP

- A serial c++ code 'mycode.cc' needs to be compiled.

## SETUP

- A serial c++ code 'mycode.cc' needs to be compiled.
- It needs to be run 32 times with different parameters, 1 through 32.

## SETUP

- A serial c++ code 'mycode.cc' needs to be compiled.
- It needs to be run 32 times with different parameters, 1 through 32.
- The parameters are given as a command line argument.

## SETUP

- A serial c++ code 'mycode.cc' needs to be compiled.
- It needs to be run 32 times with different parameters, 1 through 32.
- The parameters are given as a command line argument.
- 8 subjobs of this code fit into the GPC compute nodes's memory.



## SETUP

- A serial c++ code 'mycode.cc' needs to be compiled.
- It needs to be run 32 times with different parameters, 1 through 32.
- The parameters are given as a command line argument.
- 8 subjobs of this code fit into the GPC compute nodes's memory.
- Each serial run on average takes  $\sim 2$  hour.

# GNU Parallel Example

\$

# GNU Parallel Example

```
$ cd $SCRATCH/example  
$
```

# GNU Parallel Example

```
$ cd $SCRATCH/example  
$ module load intel  
$
```

# GNU Parallel Example

```
$ cd $SCRATCH/example  
$ module load intel  
$ icpc -O3 -xhost mycode.cc -o myapp  
$
```

# GNU Parallel Example

```
$ cd $SCRATCH/example
$ module load intel
$ icpc -O3 -xhost mycode.cc -o myapp
$ cat > subjob.lst
  mkdir run01; cd run01; ../myapp 1 > out
  mkdir run02; cd run02; ../myapp 2 > out
  ...
  mkdir run32; cd run32; ../myapp 32 > out
$
```

# GNU Parallel Example

```
$ cd $SCRATCH/example
$ module load intel
$ icpc -O3 -xhost mycode.cc -o myapp
$ cat > subjob.lst
  mkdir run01; cd run01; ../myapp 1 > out
  mkdir run02; cd run02; ../myapp 2 > out
  ...
  mkdir run32; cd run32; ../myapp 32 > out
$ cat > GPJob
  #PBS -l nodes=1:ppn=8,walltime=12:00:00
  cd $SCRATCH/example
  module load intel gnu-parallel/20121022
  parallel --jobs 8 < subjob.lst
$
```

# GNU Parallel Example

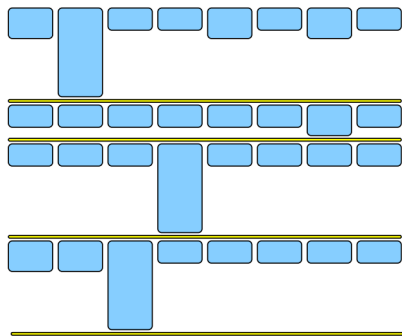
```
$ cd $SCRATCH/example
$ module load intel
$ icpc -O3 -xhost mycode.cc -o myapp
$ cat > subjob.lst
  mkdir run01; cd run01; ../myapp 1 > out
  mkdir run02; cd run02; ../myapp 2 > out
  ...
  mkdir run32; cd run32; ../myapp 32 > out
$ cat > GPJob
  #PBS -l nodes=1:ppn=8,walltime=12:00:00
  cd $SCRATCH/example
  module load intel gnu-parallel/20121022
  parallel --jobs 8 < subjob.lst
$ qsub GPJob
  2961985.gpc-sched
$
```



# GNU Parallel Example

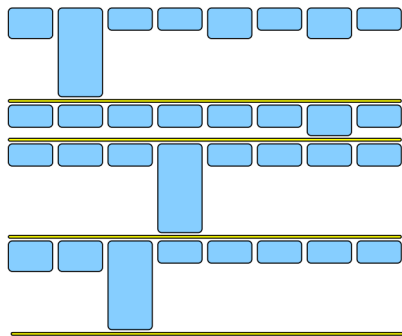
```
$ cd $SCRATCH/example
$ module load intel
$ icpc -O3 -xhost mycode.cc -o myapp
$ cat > subjob.lst
  mkdir run01; cd run01; ../myapp 1 > out
  mkdir run02; cd run02; ../myapp 2 > out
  ...
  mkdir run32; cd run32; ../myapp 32 > out
$ cat > GPJob
  #PBS -l nodes=1:ppn=8,walltime=12:00:00
  cd $SCRATCH/example
  module load intel gnu-parallel/20121022
  parallel --jobs 8 < subjob.lst
$ qsub GPJob
2961985.gpc-sched
$ ls
  GPJob  GPJob.e2961985  GPJob.o2961985  subjob.lst
  myapp  run01           run02           run03
  ...
```

# GNU Parallel Example

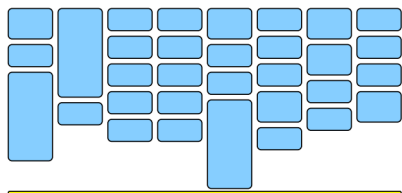


17 hours  
42% utilization

# GNU Parallel Example



17 hours  
42% utilization



10 hours  
72% utilization

## What else can it do?

- Recover from crashes (joblog/resume options)
- Span multiple nodes

## Using GNU Parallel

- [wiki.scinethpc.ca/wiki/index.php/User\\_Serial](http://wiki.scinethpc.ca/wiki/index.php/User_Serial)
- [wiki.scinethpc.ca/wiki/images/7/7b/Tech-talk-gnu-parallel.pdf](http://wiki.scinethpc.ca/wiki/images/7/7b/Tech-talk-gnu-parallel.pdf)
- [www.gnu.org/software/parallel](http://www.gnu.org/software/parallel)
- [www.youtube.com/playlist?list=PL284C9FF2488BC6D1](http://www.youtube.com/playlist?list=PL284C9FF2488BC6D1)
- O. Tange, GNU Parallel – The Command-Line Power Tool, ;login: The USENIX Magazine, February 2011:42-47.

## Languages

- serial
  - C, C++, Fortran
- threaded (shared memory)
  - OpenMP, pthreads
- message passing (distributed memory)
  - MPI, PGAS (UPC, Coarray Fortran)
- accelerator (GPU, Cell, MIC, FPGA)
  - CUDA, OpenCL, OpenACC

## HPC Software Stack

- Typically GNU/Linux
- non-interactive batch processing using a queuing system scheduler
- software packages and versions usually available as “modules”
- Parallel filesystem (GPFS,Lustre)