

# Scientific Computing III. High Performance Scientific Computing

(Phys 2109/Ast 3100H)

## Lecture 1: Parallel Programming & OpenMP

SciNet HPC Consortium, University of Toronto

February 10, 2012

# Why Parallel Programming?



# Why Parallel Programming?



## 1. **Faster**

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

# 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.

# 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*.

# 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!**

## Why is it 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, but now becomes reachable.

## Why is it 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, but now becomes reachable.

However:



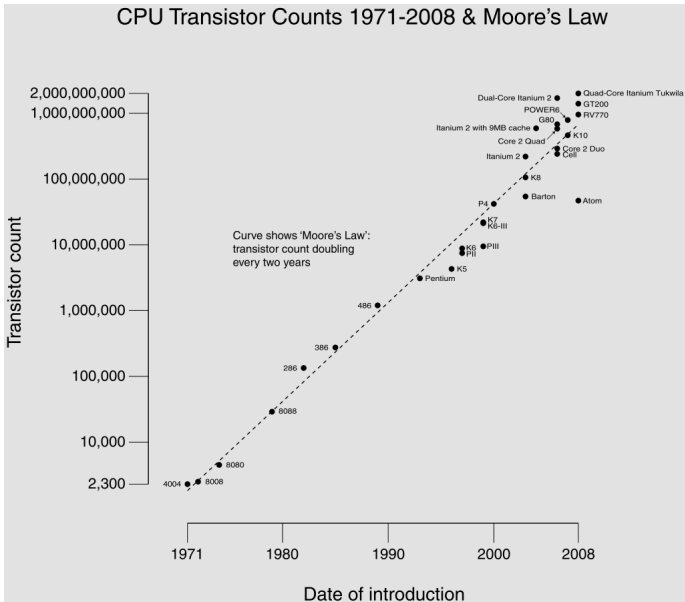
## Why is it necessary?

- ▶ **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, but now 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 Wgmsimon, wikipedia)

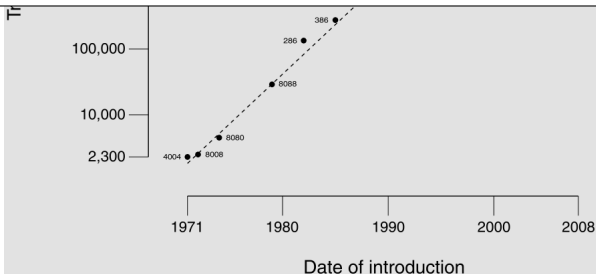
# Wait, what about Moore's Law?

## 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)*



# 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.*

Date of introduction

# Concurrency

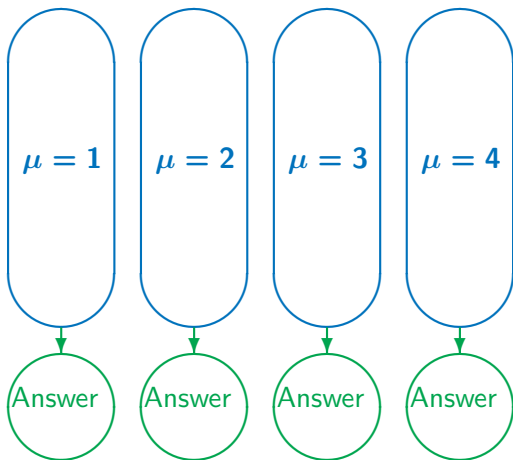
- ▶ Must have something to do for all these cores.
- ▶ Find parts of the program that can be done independently, and therefore concurrently.
- ▶ There must be many such parts.
- ▶ Their order of execution should not matter either.
- ▶ **Data dependencies limit concurrency.**



(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.

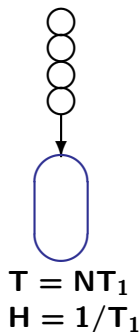


# Throughput

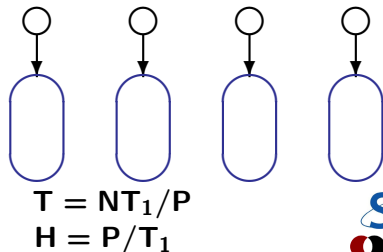
- ▶ How many tasks can you do per time unit?

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

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



vs.

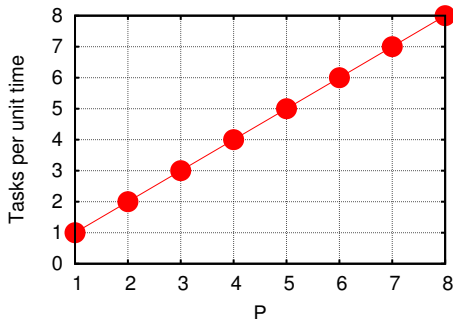


# Scaling — Throughput

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

$$H \propto P$$

- ▶ This is **Perfect scaling**.



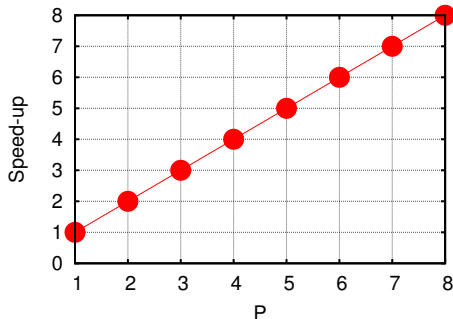


## Scaling – Speedup

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

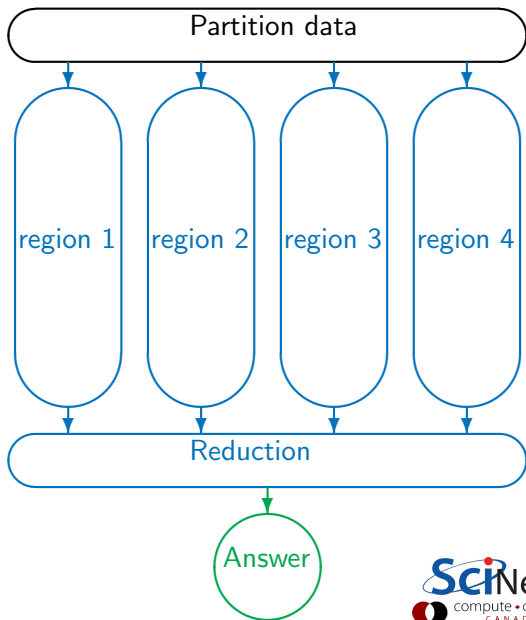
$$S = \frac{T_{\text{serial}}}{T(P)}$$

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

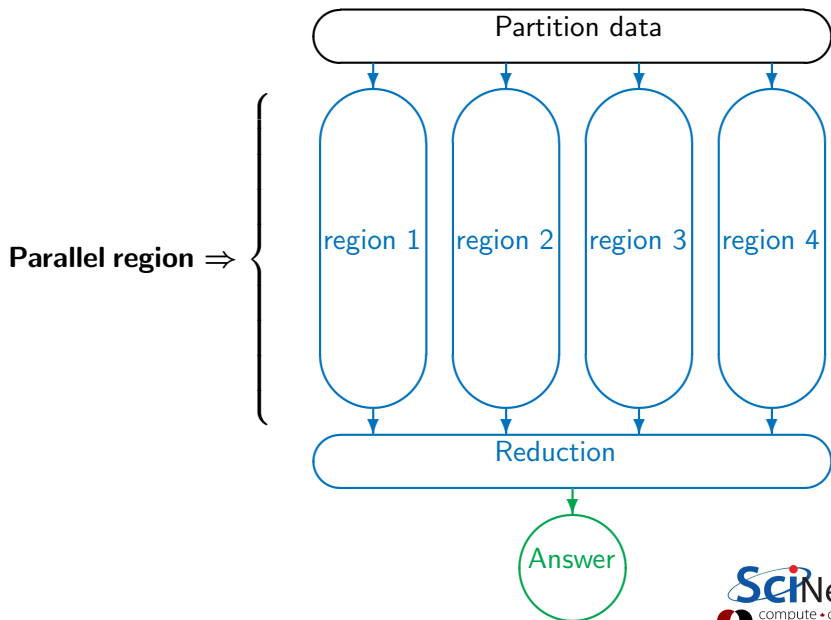


## Non-ideal cases

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

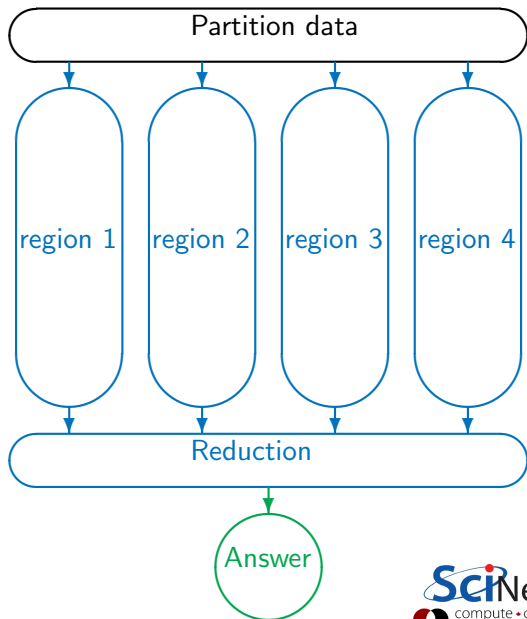


## Non-ideal cases

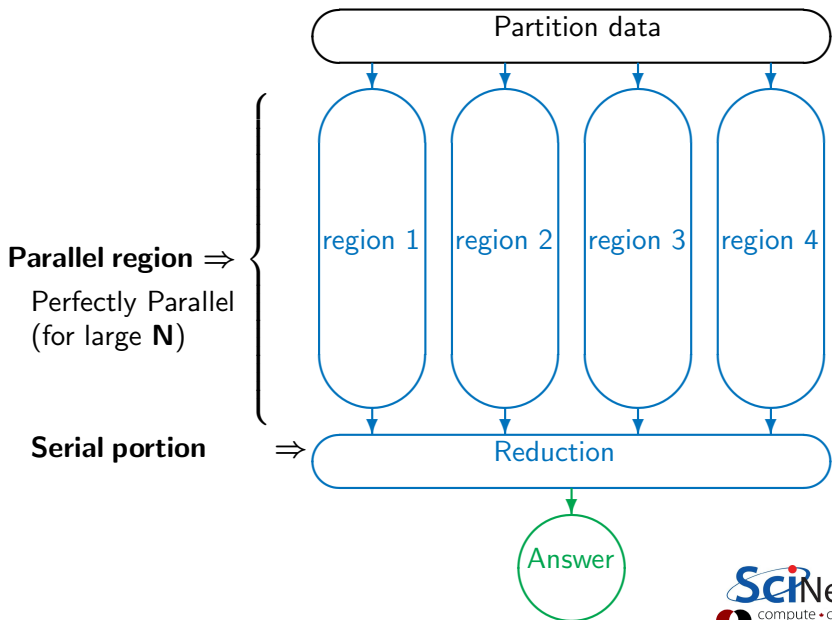


## Non-ideal cases

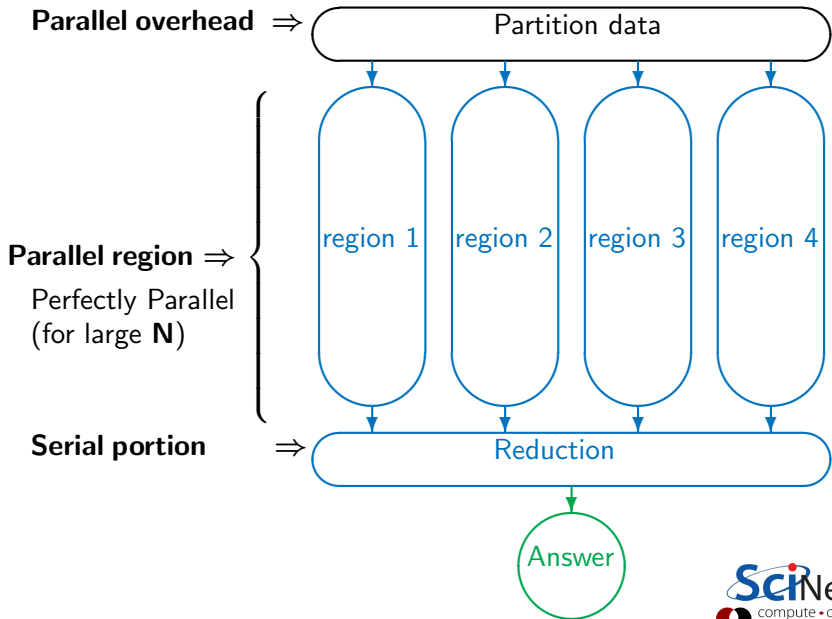
**Parallel region**  $\Rightarrow$   
Perfectly Parallel  
(for large **N**)



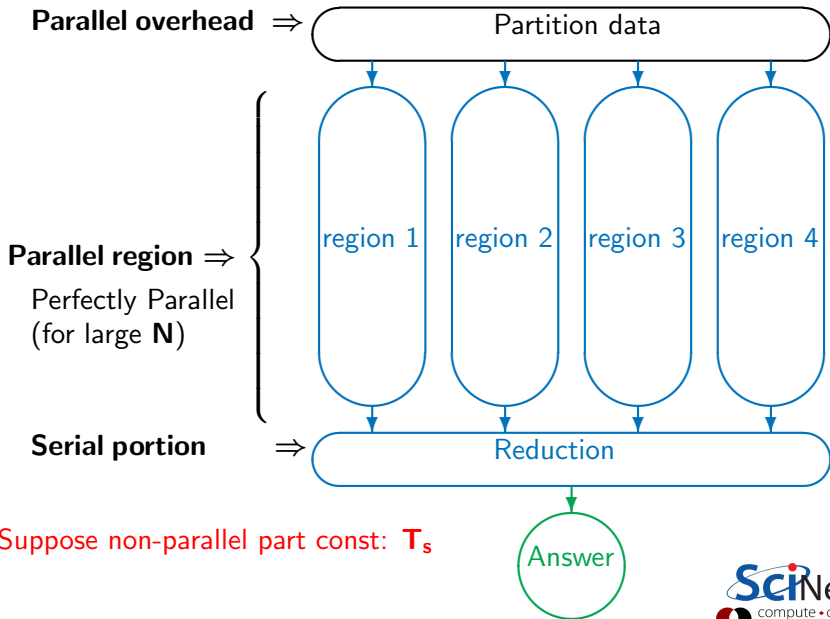
## Non-ideal cases



## Non-ideal cases



## Non-ideal cases



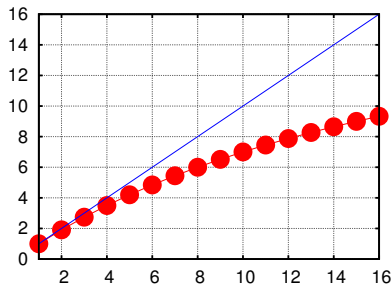
## Amdahl's law

Speed-up (without parallel overhead):

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

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

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



(for  $f = 5\%$ )



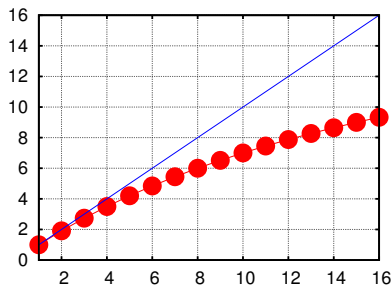
## Amdahl's law

Speed-up (without parallel overhead):

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

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

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



(for  $f = 5\%$ )

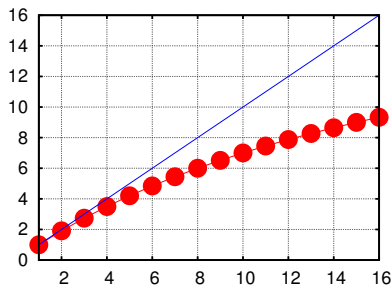
## Amdahl's law

Speed-up (without parallel overhead):

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

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

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



Serial part dominates asymptotically.

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

(for  $f = 5\%$ )

## Scaling efficiency

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

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

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

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

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.

## 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

$$T_s \approx PT_1$$

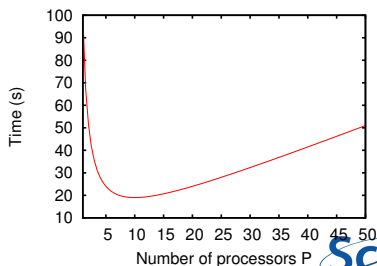
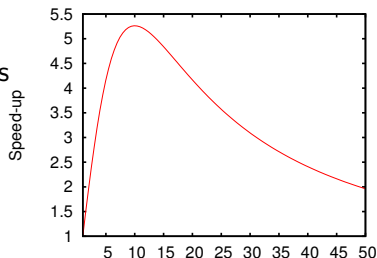
Serial fraction now a function of **P**:

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

Amdahl:

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

Example: **N = 100**, **T<sub>1</sub> = 1s**...

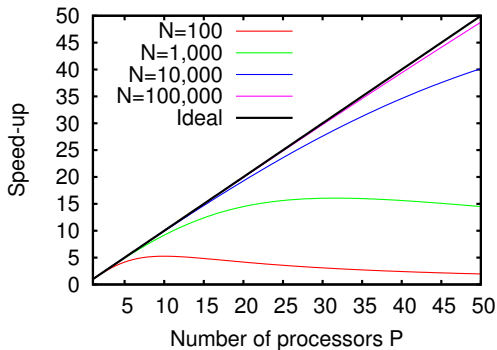


# Trying to beat Amdahl's law #1

## Scale up!

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

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

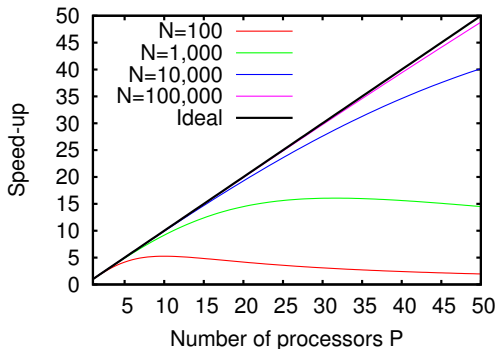


# Trying to beat Amdahl's law #1

## Scale up!

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

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



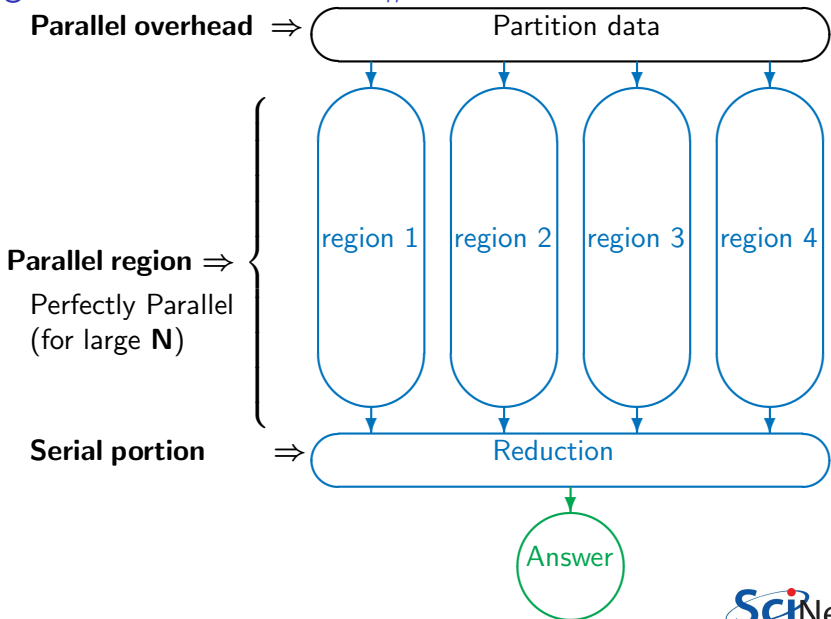
Weak scaling: Increase problem size while increasing **P**

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

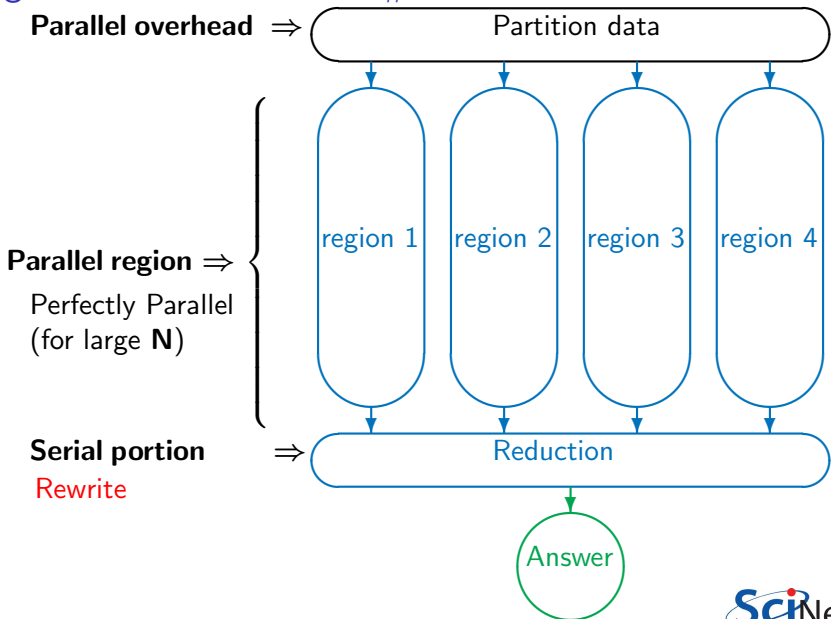
Good weak scaling means this time approaches a constant for large **P**.



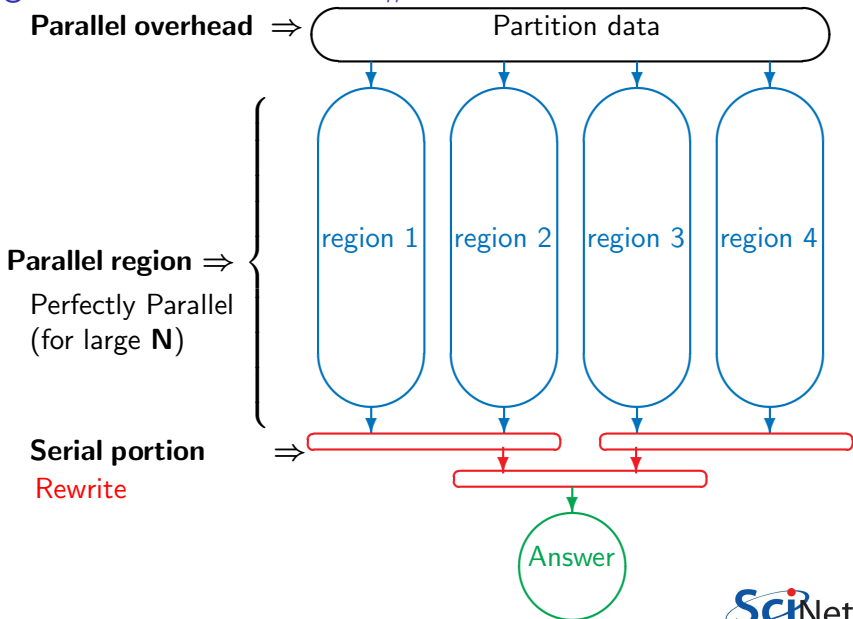
## Trying to beat Amdahl's law #2



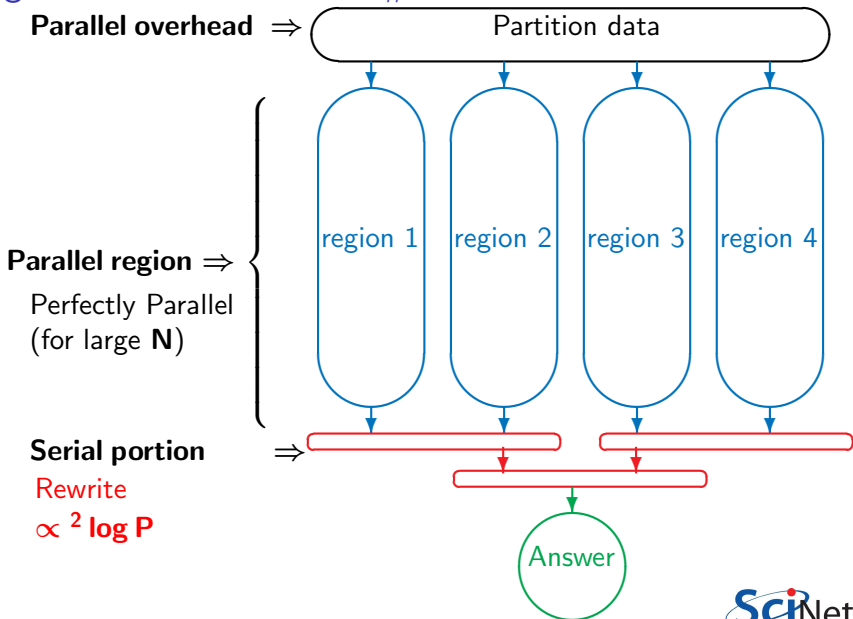
## Trying to beat Amdahl's law #2



## 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**:

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

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**:

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

Amdahl:

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

Example: **N = 100, T<sub>1</sub> = 1s...**

## Trying to beat Amdahl's law #2

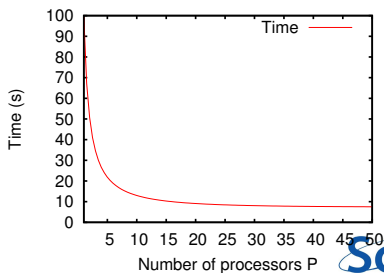
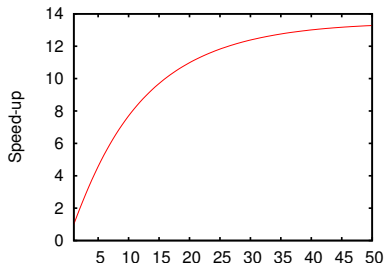
'Serial' fraction now different function of  $P$ :

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

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

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

Should approach constant for large  $\mathbf{P}$ .  
Let's see...



## Trying to beat Amdahl's law #2

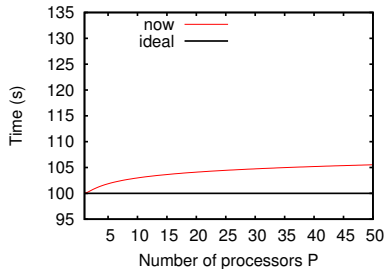
### Weak scaling

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

Should approach constant for large  $\mathbf{P}$ .

Let's see...

Not quite!



## Trying to beat Amdahl's law #2

### Weak scaling

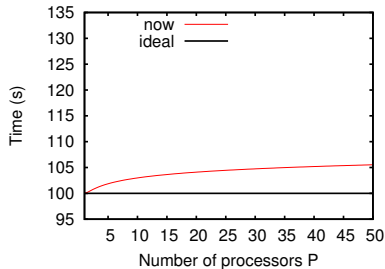
$$\text{Time}_{\text{weak}}(\mathbf{P}) = \text{Time}(\mathbf{N} = n \times \mathbf{P}, \mathbf{P})$$

Should approach constant for large  $\mathbf{P}$ .

Let's see...

Not quite!

But much better than before.



## Trying to beat Amdahl's law #2

### Weak scaling

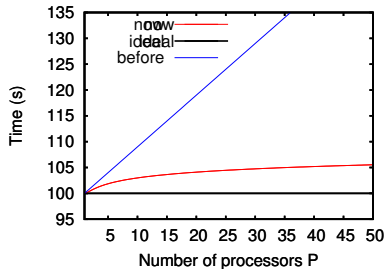
$$\text{Time}_{\text{weak}}(\mathbf{P}) = \text{Time}(\mathbf{N} = n \times \mathbf{P}, \mathbf{P})$$

Should approach constant for large  $\mathbf{P}$ .

Let's see...

Not quite!

But much better than before.



## Trying to beat Amdahl's law #2

### Weak scaling

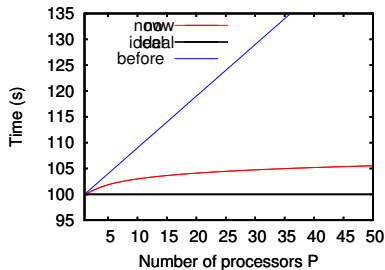
$$\text{Time}_{\text{weak}}(P) = \text{Time}(N = n \times P, P)$$

Should approach constant for large  $P$ .

Let's see...

Not quite!

But much better than before.

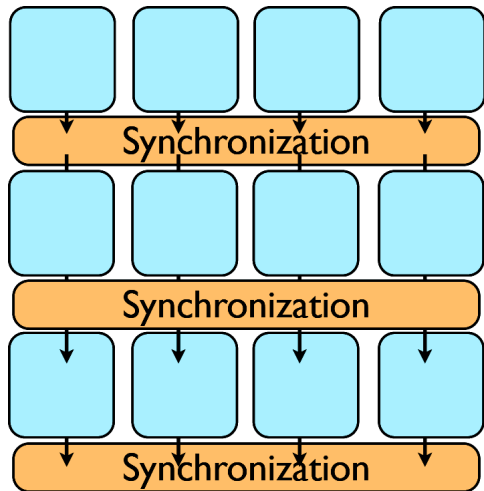


Really not that bad.

& other algorithms can do better.

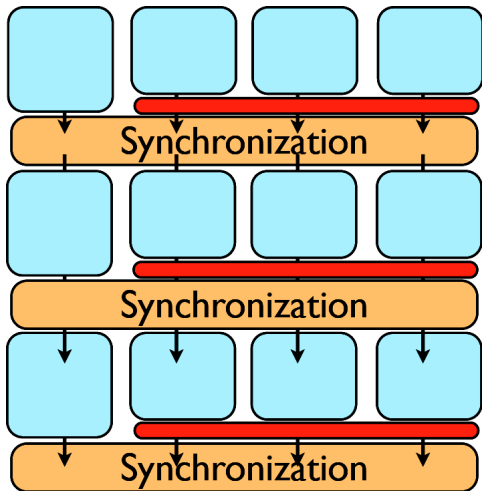
# Synchronization

- ▶ 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.



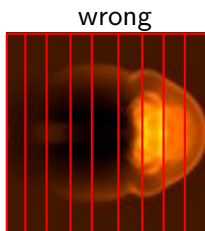
## Locality

- ▶ 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)





## Big Lesson

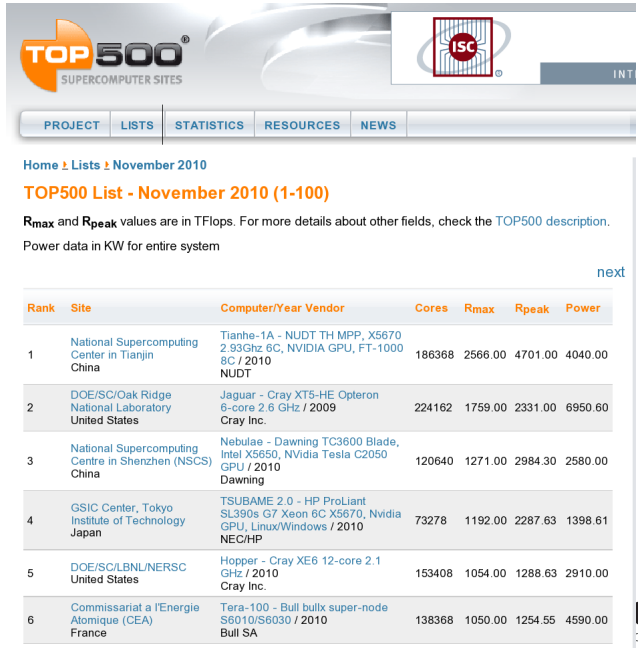
Parallel algorithm design is about finding as much concurrency as possible, and arranging it in a way that maximizes locality.

# Parallel Computers

## Top500.org:

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

Info on  
architecture, etc.



The screenshot shows the Top500.org website interface. At the top left is the 'TOP 500 SUPERCOMPUTER SITES' logo. To the right is the ISC logo. Below these are navigation tabs for 'PROJECT', 'LISTS', 'STATISTICS', 'RESOURCES', and 'NEWS'. The main content area displays the 'TOP500 List - November 2010 (1-100)'. Below the title, it states: 'R<sub>max</sub> and R<sub>peak</sub> values are in TFlops. For more details about other fields, check the TOP500 description. Power data in KW for entire system'. A table lists the top 6 supercomputers with columns for Rank, Site, Computer/Year Vendor, Cores, R<sub>max</sub>, R<sub>peak</sub>, and Power. A 'next' link is visible to the right of the table.

Rank	Site	Computer/Year Vendor	Cores	R <sub>max</sub>	R <sub>peak</sub>	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 bulx super-node S6010/S6030 / 2010 Bull SA	138368	1050.00	1254.55	4590.00

# Supercomputer architectures

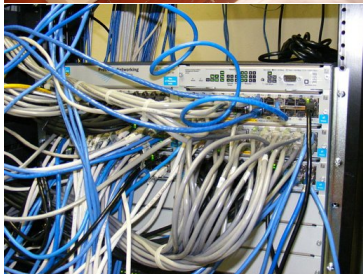
- ▶ Clusters, or, **distributed memory machines**  
In essence a bunch of desktops linked together by a network (“interconnect”). Easy and cheap.
- ▶ Multi-core machines, or, **shared memory machines**  
These can see the same memory. Limited number of cores, typically, and much more \$\$\$.
- ▶ Vector 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

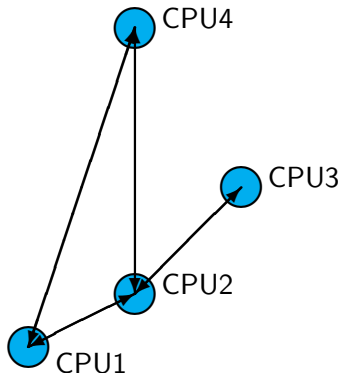


(source: <http://flickr.com/photos/eurleif>)

# 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!

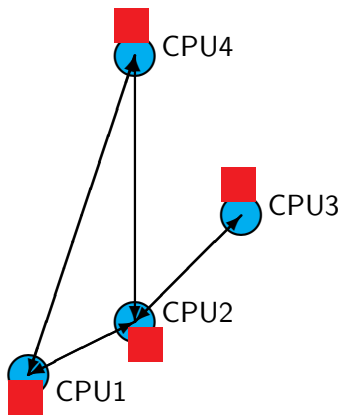
Parallel code consists of programs running on separate computers, communicating with each other.

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:

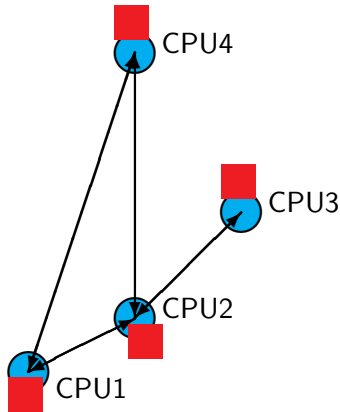
Easy to build

(Harder to build well)

Can build larger and larger clusters relatively easily

## Software:

Every communication has to be hand-coded:  
hard to program



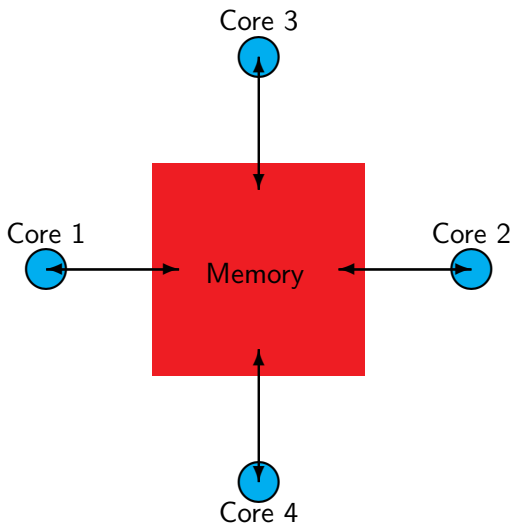
# Shared Memory

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 840  R 779.0  0.0   0:29.01 diffusion-omp
17193 root      15   0 35300 2580  60  S 15.0  0.0   0:01.57 pbs_mom
17192 root      15   0 35300 3216 696  R  6.0  0.0   0:00.48 pbs_mom
  1 root      15   0 10344  740 612  S  0.0  0.0   0:01.45 init
  2 root     RT  -5   0   0   0  0  S  0.0  0.0   0:00.00 migration/0
  3 root     RT  19   0   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     RT  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     RT  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

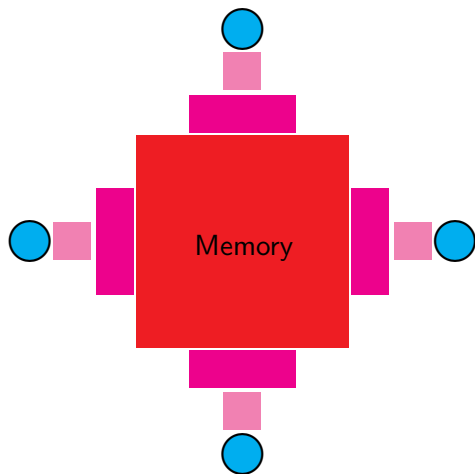
ljdursi@ gpc-f102n081:~
File Edit View Terminal Tabs Help
top - 17:33:58 up 2 days, 1:47, 1 user, load average: 0.80, 0.31, 0.17
Tasks: 150 total, 9 running, 141 sleeping, 0 stopped, 0 zombie
Cpu(s):100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 16411872k total, 2801172k used, 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     RT  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     RT  34  19   0   0  0  S  0.0  0.0   0:00.01 ksoftirqd/1
```

# Shared Memory: NUMA

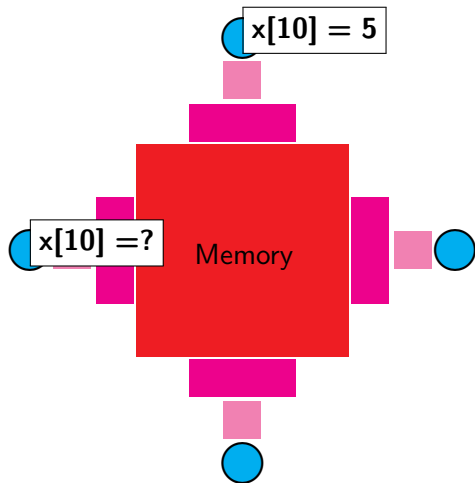
## Non-Uniform Memory Access

- ▶ 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!!



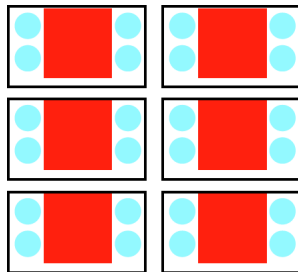
## Shared Memory Communication Cost

	Latency	Bandwidth
GigE	10 $\mu$ 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:  $O(\text{GFLOP}) \sim$  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.
- 
- Shared memory: OpenMP
  - Distributed memory: MPI
  - Graphics computing: CUDA, OpenCL



# Using SciNet

## GPC



# Using SciNet

## GPC

- ▶ 3780 nodes each with  $2 \times$  2.53GHz quad-core Intel Xeon 5500 64-bit processors
- ▶ 30240 cores total
- ▶ 16GB RAM per node
- ▶ No local hard disks
- ▶ Gigabit ethernet network on all nodes  
Used also for management, shared file system, boot, ...
- ▶ InfiniBand network on 1/4 of the nodes  
Only used for job communication
- ▶ 306 TFlops
- ▶ #16 on the June 2009 *TOP500* supercomputer sites

## Before we start with OpenMP: Mini intro to SciNet

- ▶ Need to have an account
- ▶ If you don't: get it  
([wiki.scinethpc.ca/wiki/index.php/Essentials](http://wiki.scinethpc.ca/wiki/index.php/Essentials))
- ▶ If you can't: email us.
- ▶ Read the SciNet Tutorial and the GPC quick start on the wiki.  
([wiki.scinethpc.ca/wiki/index.php/GPC\\_Quickstart](http://wiki.scinethpc.ca/wiki/index.php/GPC_Quickstart))

Access:

```
$ ssh -X login.scinet.utoronto.ca  
$ ssh -X gpc01 (or gpc02, gpc03, gpc04)
```

You compile on `gpc0{1,2,3,4}`.

But to run do:

```
$ qsub -I -l nodes=1:ppn=8,walltime=1:00:00
```

which gets a dedicated compute node for one hour.

Alternatively, submit a job script.



# OpenMP

- ▶ For shared memory systems.
- ▶ Add parallelism to functioning serial code.
- ▶ <http://openmp.org>



THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

<p>Subscribe to the News Feed</p> <p>» OpenMP</p> <ul style="list-style-type: none"><li>» Specifications</li><li>» About OpenMP</li><li>» Compilers</li><li>» Resources</li><li>» Discussion Forum</li></ul> <p>Events</p> <ul style="list-style-type: none"><li>» Multicores Expo '11: May 2-5 at the McHenry Convention Center in San Jose, California in booth #2206</li><li>» IWOMP 2011 Call For Papers (6th - 7th International Workshop on OpenMP, June 13 - 15, 2011, Chicago USA)</li></ul> <p>Input Register</p> <p>Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.</p> <p><a href="mailto:webmaster@openmp.org">webmaster@openmp.org</a></p> <p>Search OpenMP.org</p> <p>» <a href="#">OpenMP</a> - Online Search</p> <p>Search</p> <p>Archives</p> <ul style="list-style-type: none"><li>» March 2011</li><li>» February 2011</li><li>» January 2011</li><li>» October 2010</li><li>» July 2010</li><li>» May 2010</li><li>» June 2009</li><li>» April 2009</li><li>» <a href="#">View more</a></li></ul>	<h2>OpenMP News</h2> <p>» OpenMP at Multicores Expo '11 - May 2-5 - San Jose, CA</p> <p><b>RD ANNUAL multicores</b></p> <p><b>TECHNICAL CONFERENCE EXPO</b></p> <p><b>May 2-5, 2011</b></p> <p>Look for OpenMP exhibiting at the Multicores Expo, May 2-5 at the McHenry Convention Center in San Jose, California in booth #2206.</p> <p>The key objectives of the Multicores Technical Conference and Expo are to identify emerging challenges faced by designers, to suggest potential solutions or review actual designs, and to aid embedded designers in their continuing engineering education. Co-located with the Embedded Systems Conference, the agenda for this event promises to be interesting, with tracks on Multicores Debugging, Multicores Frameworks, Parallel Technologies, Software Design, and more.</p> <p>Visit us in our booth talk about the latest in OpenMP, get answers to your questions, learn about release 3.1 of the OpenMP API, and pick up the latest OpenMP reference card! For registration and other information visit <a href="http://www.multicores-expo.com">http://www.multicores-expo.com</a></p> <p>Posted on March 11, 2011</p> <h2>» Parallel Programming in Computational Engineering and Science PPCEs 2011</h2> <p>Seminar/Workshop: March 21 - March 25, 2011 Aschen, Germany <a href="http://www.rz.neth-aschen.de/ppces">http://www.rz.neth-aschen.de/ppces</a></p> <p>This event is now over, but the course material is available on the Seminar website.</p> <p>This year's seminar will include a special introduction session on Monday to present the new HPC cluster to be delivered by Bull. During the remainder of the week, we will cover Serial Programming, Tuning, Debugging and Processor Architectures (Tuesday), Shared Memory Programming with OpenMP (Wednesday), Message Passing with MPI (Thursday) and C/CPU Programming on Friday. Some of these lectures will feature hands-on sessions.</p> <p>Attendees should be comfortable with C/C++ or Fortran programming and interested in learning more about the technical details of application tuning and parallelization on their favored platform (Windows or Linux). The presentations will be given in English.</p> <p>Dierker An (RWTH), Thomas Worschko (Bull), Herbert Cornibus (Bull), Jean-Pierre Proustere (Bull), Christian Bockel (RWTH) and Felix Wolf (German Research School for Simulation Science) for our Monday event. The remainder of the week will be covered by Ruud van der Plas (Oracle), Michael Wolfe (PGI) and speakers of the HPC Team of the RWTH Aachen University.</p> <p>The seminar is free. Allocation is on a first come, first served basis, as we are limited in capacity. Please register separately for any session you intend to participate. Go to:</p>	<h2>The OpenMP API</h2> <p>supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.</p> <p>» <a href="#">Read about OpenMP.org</a></p> <hr/> <p>Get</p> <ul style="list-style-type: none"><li>» OpenMP specs</li></ul> <hr/> <p>Use</p> <ul style="list-style-type: none"><li>» OpenMP Compilers</li></ul> <hr/> <p>Learn</p>  <p>» <a href="#">Using OpenMP - the book</a></p> <ul style="list-style-type: none"><li>» <a href="#">Using OpenMP - the examples</a></li><li>» <a href="#">Using OpenMP - the forum</a></li><li>» <a href="#">Wikipedia</a></li><li>» <a href="#">OpenMP Tutorial</a></li><li>» <a href="#">More Resources</a></li></ul> <hr/> <p>Discuss</p> <ul style="list-style-type: none"><li>» <a href="#">User Forum</a></li></ul> <p>Ask the experts and get answers to questions about OpenMP.</p>
---	---	---

# OpenMP

- ▶ For shared memory systems.
- ▶ Add parallelism to functioning serial code.
- ▶ <http://openmp.org>

- ▶ Compiler, run-time environment does a lot of work for us
- ▶ Divides up work
- ▶ But we have to tell it how to use variables, where to run in parallel, . . .
- ▶ Mark parallel regions.
- ▶ Works by adding compiler directives to code.

Invisible to non-openmp compilers.



- ▶ OpenMP
- ▶ Specifications
- ▶ About OpenMP
- ▶ Compilers
- ▶ Resources
- ▶ Discussion Forum

Events  
• Multiscore Expo '11: May 2-5 at the McHenry Convention Center in San Jose, California in booth #2206

• NWOMP 2011 Call For Papers (9th - 7th International Workshop on OpenMP), June 13 - 15, 2011, Chicago, USA

Input Register  
Alert the OpenMP.org website about new products, events, or updates and we'll post it here. [www.inputregister.org](http://www.inputregister.org)

Search OpenMP.org  
Search

- Archives
- ◊ March 2011
  - ◊ February 2011
  - ◊ January 2011
  - ◊ October 2010
  - ◊ July 2010
  - ◊ May 2010
  - ◊ June 2009
  - ◊ April 2009
  - ◊ March 2009

## OpenMP News

OpenMP at Multiscore Expo '11 - May 2-5 - San Jose, CA

Look for OpenMP exhibiting at the Multiscore Expo, May 2-5 at the McHenry Convention Center in San Jose, California in booth #2206.

The key objectives of the Multiscore Technical Conference and Expo are to identify emerging challenges faced by designers, to suggest potential solutions or review actual designs, and to aid embedded designers in their continuing engineering education. Co-located with this event are the Embedded Systems Conference, the agenda for this event precedes to be interesting, with tracks on Multiscore Debugging, Multiscore Frameworks, Parallel Technologies, Software Design, and more.

Visit us in our booth talk about the latest in OpenMP, get answers to your questions, learn about release 3.1 of the OpenMP API, and pick up the latest OpenMP reference card! For registration and other information visit <http://www.multiscore-expo.com>

Posted on March 11, 2011

## Parallel Programming in Computational Engineering and Science PPECES 2011

Seminar/Workshop: March 21 - March 25, 2011  
Aachen, Germany <http://www.rz.rwth-aachen.de/ppeces>

This event is now over, but the course material is available on the Seminar website.

This year's seminar will include a special introduction session on Monday to present the new HPC-cluster to be delivered by RWTH. During the remainder of the week, we will cover Serial Programming, Tuning, Debugging and Processor Architectures (Tuesday), Shared Memory Programming with OpenMP (Wednesday), Message Passing with MPI (Thursday) and C/CPU Programming on Friday. Some of these lectures will feature hands-on sessions.

Attendees should be comfortable with C/C++ or Fortran programming and interested in learning more about the technical details of application tuning and parallelization on their favored platform (Windows or Linux). The presentations will be given in English.

Dierker an May (RWTH), Thomas Warschko (BfL), Herbert Cornelius (IWI), Jean-Francois Protaine (BfL), Christian Bockstiegel (RWTH) and Felix Wolf (German Research School for Simulation Science) for our Monday event. The remainder of the week will be covered by Rüdiger van der Plas (Oracle), Michael Wolfe (PGI) and speakers of the HPC Team of the RWTH Aachen University.

The seminar is free. Allocation is on a first come, first served basis, as we are limited in capacity. Please register separately for any session you intend to participate. Go to: <http://www.rz.rwth-aachen.de/ppeces>

**The OpenMP API**  
supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer. [Read about OpenMP.org](#)

Get  
• OpenMP specs

Use  
• OpenMP Compilers

Learn



- Using OpenMP - the book
- Using OpenMP - the examples
- Using OpenMP - the forum
- Wikipedia
- OpenMP Tutorial
- More Resources

Discuss  
• User Forum  
Ask the experts and get answers to questions about OpenMP

# OpenMP basic operations

## In code:

- ▶ In C/C++, you add lines starting with `#pragma omp`. This parallelizes the subsequent code block.
- ▶ These lines are skipped (often with a warning) by compilers that do not support OpenMP.

## When compiling:

- ▶ To turn on OpenMP support in gcc and g++, add the `-fopenmp` flag to the compilation (and link!) commands.

## When running:

- ▶ The environment variable `OMP_NUM_THREADS` determines how many threads will be started in an OpenMP parallel block.

# OpenMP example

C:

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n", omp_get_thread_num());
    }
}
```

# OpenMP example

```
$ gcc -std=c99 -Wall -O2 -o omp-hello-world omp-hello-world.c -fopenmp

$ export OMP_NUM_THREADS=8
$ ./omp-hello-world
...
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
...
$ export OMP_NUM_THREADS=32
$ ./omp-hello-world
...
```

Let's see what happens...

# OpenMP example

```
$ gcc -Wall -O2 -o omp-hello-world omp-hello-world.c -fopenmp
```

```
$ export OMP_NUM_THREADS=8
```

```
$ ./omp-hello-world
```

```
At start of program
```

```
Hello, world, from thread 0!
```

```
Hello, world, from thread 6!
```

```
Hello, world, from thread 5!
```

```
Hello, world, from thread 4!
```

```
Hello, world, from thread 2!
```

```
Hello, world, from thread 1!
```

```
Hello, world, from thread 7!
```

```
Hello, world, from thread 3!
```

```
$ export OMP_NUM_THREADS=1
```

```
$ ./omp-hello-world
```

```
At start of program
```

```
Hello, world, from thread 0!
```

```
$ export OMP_NUM_THREADS=32
```

```
$ ./omp-hello-world
```

```
At start of program
```

```
Hello, world, from thread 11!
```

```
Hello, world, from thread 1!
```

```
Hello, world, from thread 16!
```

```
...
```

# So what happened precisely?

- ▶ OMP\_NUM\_THREADS threads were launched.
- ▶ Each prints “Hello, world ...”;
- ▶ In seemingly random order.
- ▶ Only one “At start of program”.

```
$ gcc -o omp-hello-world omp-hello-world.c
$ export OMP_NUM_THREADS=8
$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
Hello, world, from thread 6!
Hello, world, from thread 5!
Hello, world, from thread 4!
Hello, world, from thread 2!
Hello, world, from thread 1!
Hello, world, from thread 7!
Hello, world, from thread 3!
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
$ export OMP_NUM_THREADS=32
$ ./omp-hello-world
At start of program
Hello, world, from thread 11!
Hello, world, from thread 1!
Hello, world, from thread 16!
```

## So what happened precisely?

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n", omp_get_thread_num());
    }
}
```



## So what happened precisely?

Program starts normally (single thread)

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n", omp_get_thread_num());
    }
}
```

## So what happened precisely?

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n", omp_get_thread_num());
    }
}
```

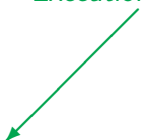
At start of parallel section, launching  
`OMP_NUM_THREADS` threads,  
Each executes the same code!



## So what happened precisely?

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n", omp_get_thread_num());
    }
}
```


At end of parallel section,  
threads join back up,  
Execution continues serially.



## So what happened precisely?

Special function to find number of current thread (first=0).

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n", omp_get_thread_num());
    }
}
```



## OpenMP functions (from omp.h)

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d of %d!\n",
            omp_get_thread_num(),
            omp_get_num_threads());
    }
}
```

omp\_get\_num\_threads() called by all threads.  
Let's see if we can fix that...

## OpenMP functions (from omp.h)

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
            omp_get_thread_num());
    }
    printf("There were %d threads.\n", omp_get_num_threads());
}
```

What do you think, will this work?

## OpenMP functions (from omp.h)

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
            omp_get_thread_num());
    }
    printf("There were %d threads.\n", omp_get_num_threads());
}
```

What do you think, will this work?

No:

Says 1 thread only!

## OpenMP functions (from omp.h)

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
            omp_get_thread_num());
    }
    printf("There were %d threads.\n", omp_get_num_threads());
}
```

What do you think, will this work?

No:

Says 1 thread only!

Why?

Because that is true outside the parallel region!

Need to get the value from the parallel region somehow.



# Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int mythread, nthreads;
    #pragma omp parallel default(none) shared(nthreads) private(mythread)
    {
        mythread = omp_get_thread_num();
        if (mythread == 0)
            nthreads = omp_get_num_threads();
    }
    printf("There were %d threads.\n", nthreads);
}
```

# Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int mythread, nthreads;
    #pragma omp parallel default(none) shared(nthreads) private(mythread)
    {
        mythread = omp_get_thread_num();
        if (mythread == 0)
            nthreads = omp_get_num_threads();
    }
    printf("There were %d threads.\n", nthreads);
}
```

Variable declarations  
How used in parallel region

# Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int mythread, nthreads;
    #pragma omp parallel default(none) shared(nthreads) private(mythread)
    {
        mythread = omp_get_thread_num();
        if (mythread == 0)
            nthreads = omp_get_num_threads();
    }
    printf("There were %d threads.\n", nthreads);
}
```

Variable declarations  
How used in parallel region

- ▶ default(none) can save you hours of debugging!
- ▶ shared: each thread sees it and can modify (be careful!).  
Preserves value.
- ▶ private: each thread gets its own copy, invisible for others  
Initial and final value undefined!

# Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int mythread, nthreads;
    #pragma omp parallel default(none) shared(nthreads) private(mythread)
    {
        mythread = omp_get_thread_num();
        if (mythread == 0)
            nthreads = omp_get_num_threads();
    }
    printf("There were %d threads.\n", nthreads);
}
```

- ▶ Program runs, launches threads.
- ▶ Each thread gets copy of mythread.
- ▶ Only thread 0 writes to nthreads.

# Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int mythread, nthreads;
    #pragma omp parallel default(none) shared(nthreads) private(mythread)
    {
        mythread = omp_get_thread_num();
        if (mythread == 0)
            nthreads = omp_get_num_threads();
    }
    printf("There were %d threads.\n", nthreads);
}
```

- ▶ Program runs, launches threads.
- ▶ Each thread gets copy of mythread.
- ▶ Only thread 0 writes to nthreads.
- ▶ Good idea to declare mythread locally!  
(avoids many bugs)

# Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int nthreads;
    #pragma omp parallel default(none) shared(nthreads)
    {
        int mythread = omp_get_thread_num();
        if (mythread == 0)
            nthreads = omp_get_num_threads();
    }
    printf("There were %d threads.\n", nthreads);
}
```

- ▶ Program runs, launches threads.
- ▶ Each thread gets copy of mythread.
- ▶ Only thread 0 writes to nthreads.
- ▶ Good idea to declare mythread locally!  
(avoids many bugs)

# Single Execution in OpenMP

```
#include <stdio.h>
#include <omp.h>
int main() {
    int nthreads;
    #pragma omp parallel default(none) shared(nthreads)
    {
        int mythread = omp_get_thread_num();
        if (mythread == 0)
            nthreads = omp_get_num_threads();
    }
    printf("There were %d threads.\n", nthreads);
}
```

- ▶ Do we care that it's thread 0 in particular that updates nthreads?
- ▶ Often, we just want the first thread to go through, do not care which one.

# Single Execution in OpenMP

```
#include <stdio.h>
#include <omp.h>
int main() {
    int nthreads;
    #pragma omp parallel default(none) shared(nthreads)
    #pragma omp single
        nthreads = omp_get_num_threads();
    printf("There were %d threads.\n", nthreads);
}
```



## Loops in OpenMP

Take one of your openmp programs and add a loop.

# Loops in OpenMP

Take one of your openmp programs and add a loop.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int i, mythread;
    #pragma omp parallel default(none) XXXX(i) XXXX(mythread)
    {
        mythread = omp_get_thread_num();
        for (i=0; i<16; i++)
            printf("Thread %d gets i=%d\n",
                mythread, i);
    }
}
```

# Loops in OpenMP

Take one of your openmp programs and add a loop.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int i, mythread;
    #pragma omp parallel default(none) XXXX(i) XXXX(mythread)
    {
        mythread = omp_get_thread_num();
        for (i=0; i<16; i++)
            printf("Thread %d gets i=%d\n",
                mythread, i);
    }
}
```

What would you imagine this does when run with e.g.  
OMP\_NUM\_THREADS=8?

## Worksharing constructs in OpenMP

- ▶ We don't generally want tasks to do exactly the same thing.
- ▶ Want to partition a problem into pieces, each thread works on a piece.
- ▶ Most scientific programming full of work-heavy loops.
- ▶ OpenMP has a worksharing construct: `omp for`.

# Worksharing constructs in OpenMP

- ▶ We don't generally want tasks to do exactly the same thing.
- ▶ Want to partition a problem into pieces, each thread works on a piece.
- ▶ Most scientific programming full of work-heavy loops.
- ▶ OpenMP has a worksharing construct: `omp for`.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int i, mythread;
    #pragma omp parallel default(none) XXXX(i) XXXX(mythread)
    {
        mythread = omp_get_thread_num();
        #pragma omp for
        for (i=0; i<16; i++)
            printf("Thread %d gets i=%d\n",mythread,i);
    }
}
```

# Worksharing constructs in OpenMP

- ▶ `omp for` construct breaks up the iterations by thread.
- ▶ If doesn't divide evenly, does the best it can.
- ▶ Allows easy breaking up of work!
- ▶ Advanced: can break up work of arbitrary blocks of code with `omp task` construct.

```
$ ./omp_loop
thread 3 gets i= 6
thread 3 gets i= 7
thread 4 gets i= 8
thread 4 gets i= 9
thread 5 gets i= 10
thread 5 gets i= 11
thread 6 gets i= 12
thread 6 gets i= 13
thread 1 gets i= 2
thread 1 gets i= 3
thread 0 gets i= 0
thread 0 gets i= 1
thread 2 gets i= 4
thread 2 gets i= 5
thread 7 gets i= 14
thread 7 gets i= 15
$
```

## Less trivial example: DAXPY

- ▶ multiply a vector by a scalar, add a vector.
- ▶ (a X plus Y, in double precision)
- ▶ Implement this, first serially, then with OpenMP
- ▶ daxpy.c
- ▶ make daxpy

$$z = ax + y$$

### Warning

This is a common linear algebra construct that you really shouldn't implement yourself. Various BLAS implementations will do a much better job than you. But good for illustration.

```
#include <stdio.h>
#include "pca_utils.h"
void daxpy(int n, double a, double *x, double *y, double *z) {
    for (int i=0; i<n; i++) {
        x[i] = (double)i*(double)i;
        y[i] = ((double)i+1.)*((double)i-1.);
    }
    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}
int main() {
    int n=1e7;
    double *x = vector(n);
    double *y = vector(n);
    double *z = vector(n);
    double a = 5./3.;
    pca_time tt;
    tick(&tt);
    daxpy(n,a,x,y,z);
    tock(&tt);
    free(z);
    free(y);
    free(x);
}
```



```
#include <stdio.h>
#include "pca_utils.h"
void daxpy(int n, double a, double *x, double *y, double *z) {
    for (int i=0; i<n; i++) {
        x[i] = (double)i*(double)i;
        y[i] = ((double)i+1.)*((double)i-1.);
    }
    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}
int main() {
    int n=1e7;
    double *x = vector(n);
    double *y = vector(n);
    double *z = vector(n);
    double a = 5./3.;
    pca_time tt;
    tick(&tt);
    daxpy(n,a,x,y,z);
    tock(&tt);
    free(z);
    free(y);
    free(x);
}
```

Utilities for this course

```
#include <stdio.h>
#include "pca_utils.h"
void daxpy(int n, double a, double *x, double *y, double *z) {
    for (int i=0; i<n; i++) {
        x[i] = (double)i*(double)i;
        y[i] = ((double)i+1.)*((double)i-1.);
    }
    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}
int main() {
    int n=1e7;
    double *x = vector(n);
    double *y = vector(n);
    double *z = vector(n);
    double a = 5./3.;
    pca_time tt;
    tick(&tt);
    daxpy(n,a,x,y,z);
    tock(&tt);
    free(z);
    free(y);
    free(x);
}
```

← Fill arrays with calculated values

```
#include <stdio.h>
#include "pca_utils.h"
void daxpy(int n, double a, double *x, double *y, double *z) {
    for (int i=0; i<n; i++) {
        x[i] = (double)i*(double)i;
        y[i] = ((double)i+1.)*((double)i-1.);
    }
    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}
int main() {
    int n=1e7;
    double *x = vector(n);
    double *y = vector(n);
    double *z = vector(n);
    double a = 5./3.;
    pca_time tt;
    tick(&tt);
    daxpy(n,a,x,y,z);
    tock(&tt);
    free(z);
    free(y);
    free(x);
}
```

Do calculation.



```
#include <stdio.h>
#include "pca_utils.h"
void daxpy(int n, double a, double *x, double *y, double *z) {
    for (int i=0; i<n; i++) {
        x[i] = (double)i*(double)i;
        y[i] = ((double)i+1.)*((double)i-1.);
    }
    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}
int main() {
    int n=1e7;
    double *x = vector(n);
    double *y = vector(n);
    double *z = vector(n);
    double a = 5./3.;
    pca_time tt;
    tick(&tt);
    daxpy(n,a,x,y,z);
    tock(&tt);
    free(z);
    free(y);
    free(x);
}
```

← Driver (setup, call, timing).

# OpenMP version of daxpy

```
void daxpy(int n, double a, double *x, double *y, double *z) {
    #pragma omp parallel default(none) shared(n,x,y,a,z) private(i)
    {
        #pragma omp for
        for (int i=0; i<n; i++) {
            x[i] = (double)i*(double)i;
            y[i] = ((double)i+1.)*((double)i-1.);
        }
        #pragma omp for
        for (int i=0; i<n; i++)
            z[i] += a * x[i] + y[i];
    }
}
```

# Homework

1. Make sure you've got a SciNet account!
2. Read the SciNet tutorial (as it pertains to the GPC)
3. Read the GPC Quick Start.
4. Get the first set of code:

```
$ git clone /scinet/course/sc3/hw1
$ cd hw1
$ . setup
$ make
$ make testrun
```

5. This contains the serial daxpy.
6. Make sure it compiles and runs on the GPC.
7. Create the openmp version as just discussed.
8. Run this version for all values of OMP\_NUM\_THREADS from 1 to 16 on a single node, using a batch script. Make sure to time the duration of these runs.
9. Submit git log, makefile, code, job script(s), and plots of the speedup and efficiency as a function of **P**.