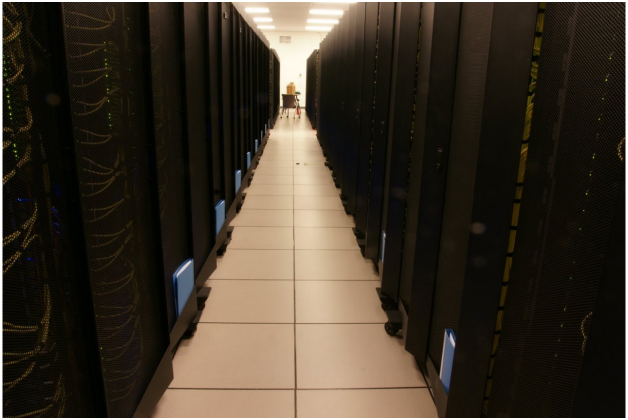
Practical Parallel Programming Intensive

SciNet HPC Consortium

9–13 May 2011

Welcome to the intensive parallel programming course!



Part I

The Course

Main goal of the course

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

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

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

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

Schedule

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

Strongly recommended books

(not provided by us)

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

Part II

Introduction to Parallel Programming

Why Parallel Programming?

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

So use more computers!

Why Parallel Programming?

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

So use more computers!

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?

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

Why is it necessary?

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

However:

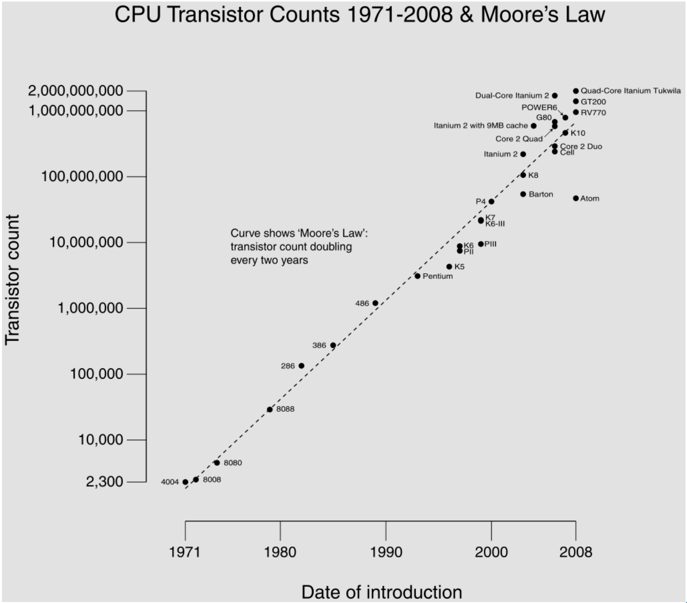
Why is it necessary?

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

However:

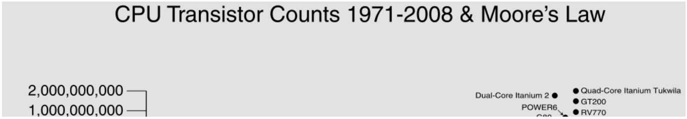
- Advances in clock speeds, bigger and faster memory and disks have been lagging as compared to e.g. 10 years ago.
  Can no longer "just wait a year" and get a better computer.
- So more computing resources here means: more cores running concurrently.
- *Even most laptops now have 2 or more cpus.*
-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

Wait, what about Moore's Law?



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

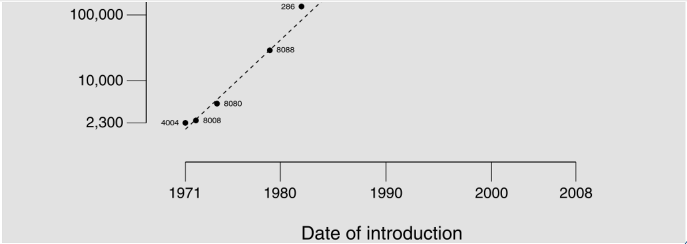
Wait, what about Moore's Law?



Moore's law

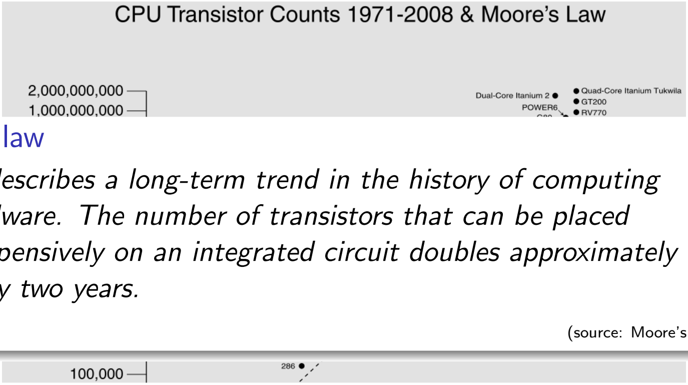
...describes a long-term trend in the history of computing hardware. The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years.

(source: Moore's law, wikipedia)



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

Wait, what about Moore's Law?



Moore's law

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

(source: Moore's law, wikipedia)

But...

- Moores Law didn't promise us clock speed.
- More transistors but getting hard to push clock speed up. Power density is limiting factor.
- So more cores at fixed clock speed.

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

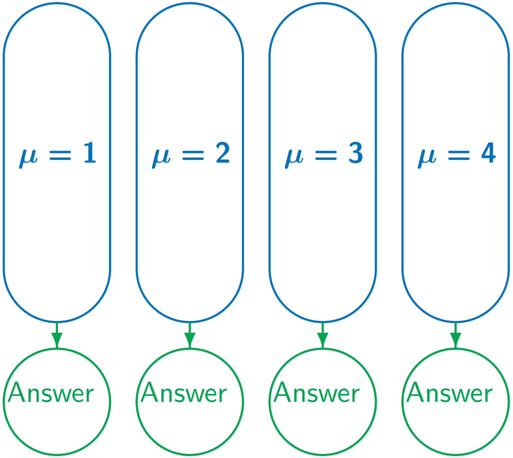
Concurrency

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

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

Parameter study: best case scenario

- Aim is to get results from a model as a parameter varies.
- Can run the serial program on each processor at the same time.
- Get "more" done.

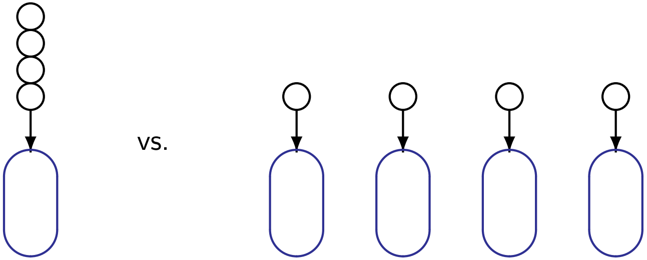


Throughput

- How many tasks can you do per time unit?

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

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



vs.

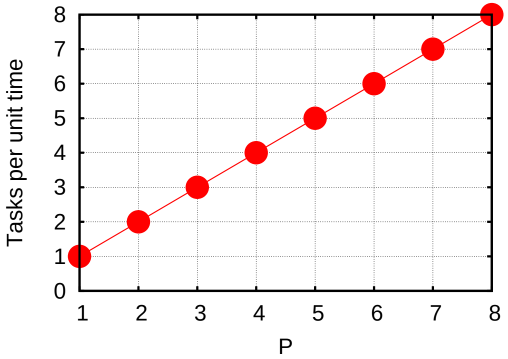
$$\mathbf{T} = \mathbf{N}\mathbf{T}_1 \qquad \mathbf{T} = \mathbf{N}\mathbf{T}_1/\mathbf{P}$$
$$\mathbf{H} = 1/\mathbf{T}_1 \qquad \mathbf{H} = \mathbf{P}/\mathbf{T}_1$$

Scaling — Throughput

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

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

- This is Perfect scaling.

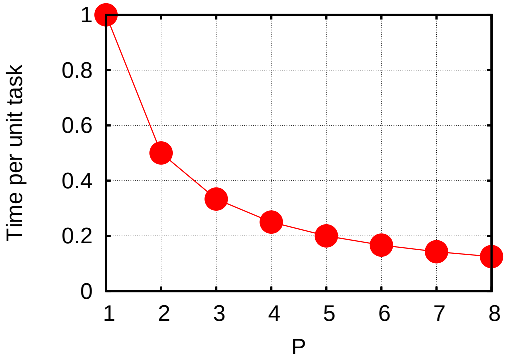


Scaling – Time

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

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

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

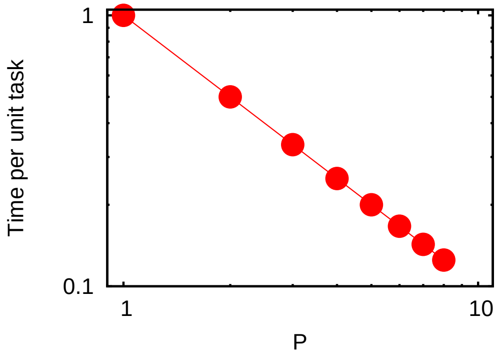


Scaling – Time

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

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

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

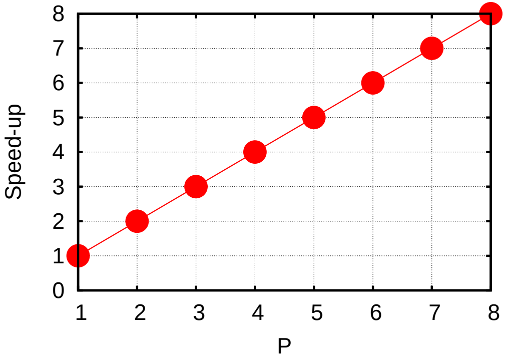


Scaling – Speedup

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

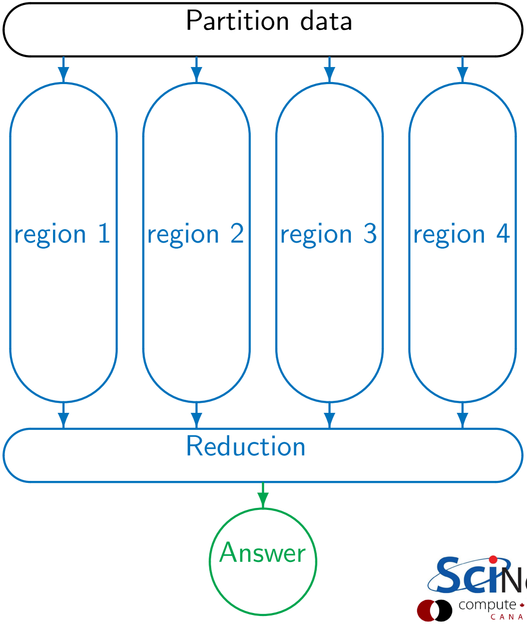
$$S = \frac{T_{\text{serial}}}{T(P)} \propto P$$

- For embarrasingly parallel applications: Linear speed up.

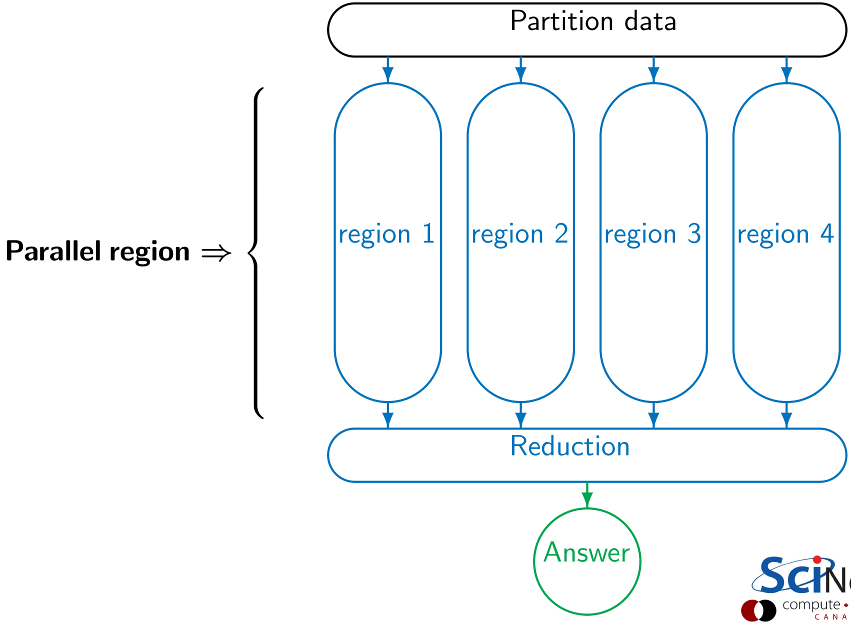


Non-ideal cases

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



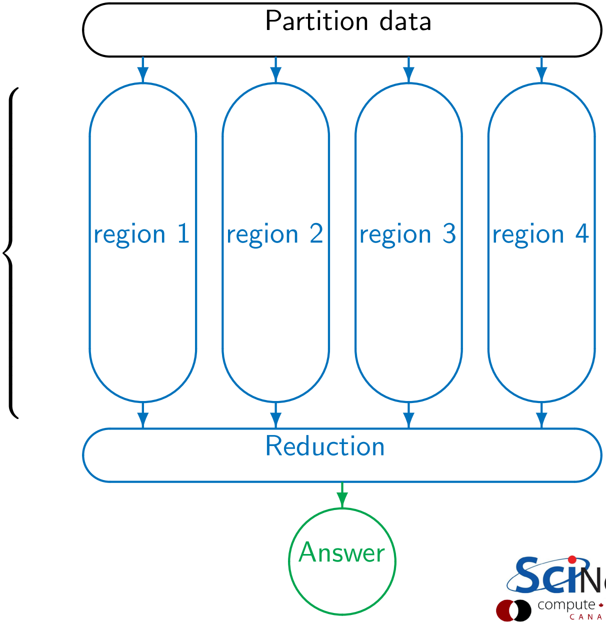
Non-ideal cases



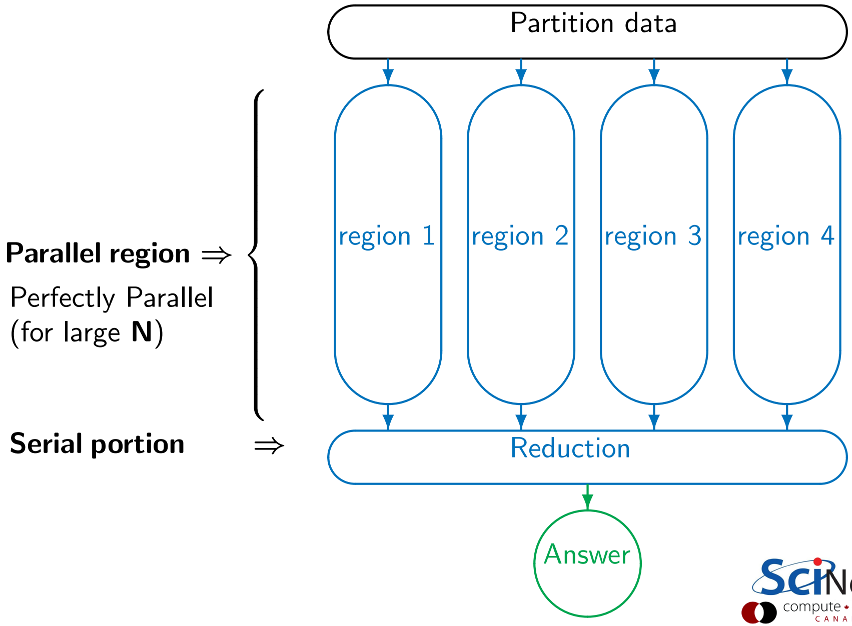
Non-ideal cases

Parallel region ⇒

Perfectly Parallel
(for large **N**)



Non-ideal cases



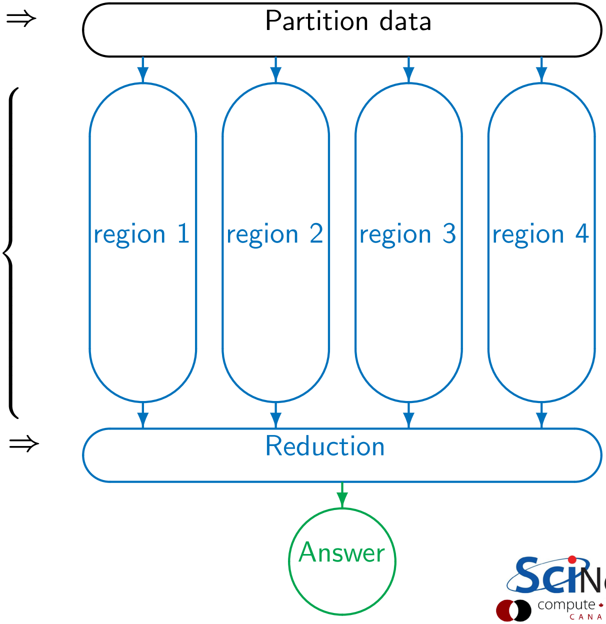
Non-ideal cases

Parallel overhead ⇒

Parallel region ⇒

Perfectly Parallel (for large **N**)

Serial portion ⇒



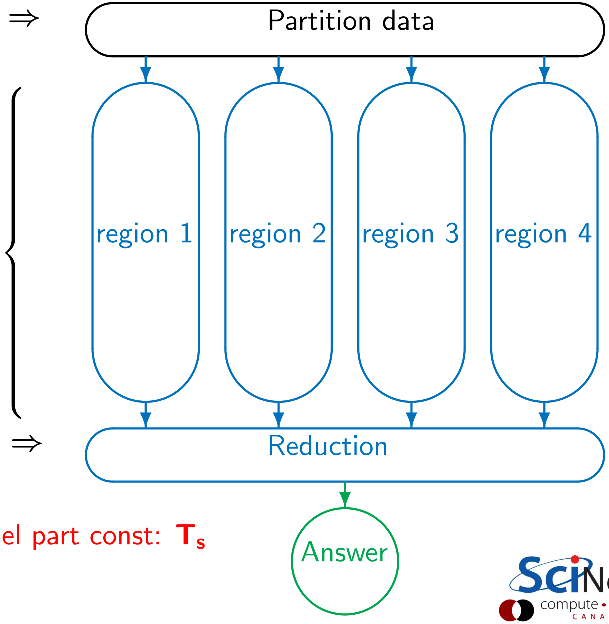
Non-ideal cases

Parallel overhead $\Rightarrow$

Parallel region $\Rightarrow$

Perfectly Parallel (for large **N**)

Serial portion $\Rightarrow$



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

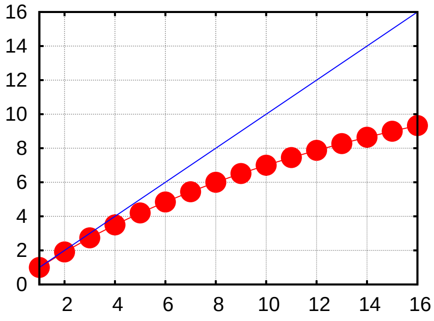
Amdahl's law

Speed-up (without parallel overhead):

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

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

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



(for $f = 5\%$)

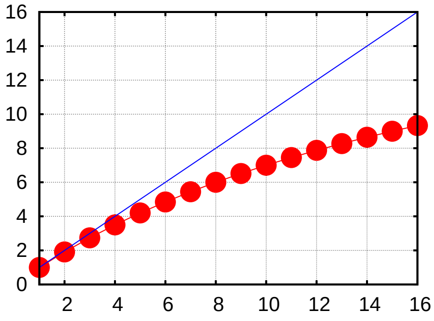
Amdahl's law

Speed-up (without parallel overhead):

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

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

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



(for $f = 5\%$)

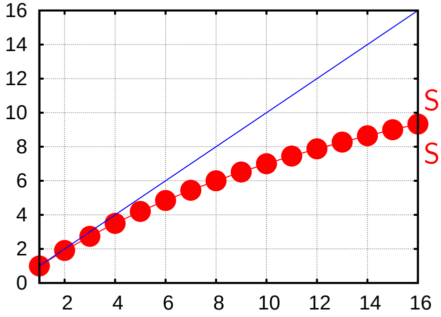
Amdahl's law

Speed-up (without parallel overhead):

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

or, calling $\mathbf{f} = \mathbf{T_s}/(\mathbf{T_s} + \mathbf{NT_1})$ the serial fraction,

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



Serial part dominates asymptotically.

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

(for $\mathbf{f} = \mathbf{5\%}$)

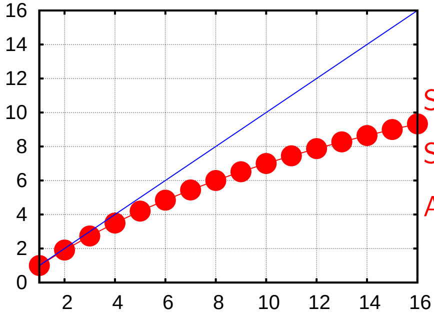
Amdahl's law

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,

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



Serial part dominates asymptotically.

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

And this is the overly optimistic case!

(for $f = 5\%$)

Scaling efficiency

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

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

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

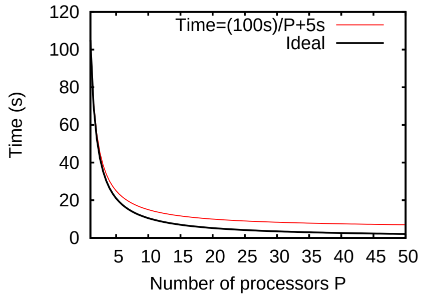
$$\mathbf{Efficiency} \sim \frac{\mathbf{1}}{\mathbf{fP}} \xrightarrow{\mathbf{P}\rightarrow\infty} \mathbf{0}$$

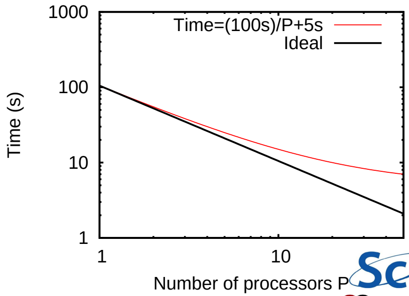
You cannot get 100% efficiency in any non-trivial problem.
All you can aim for here is to make the efficiency as least low as possible.
Sometimes, that can mean running on less processors, but more problems at the same time.

Timing example

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

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





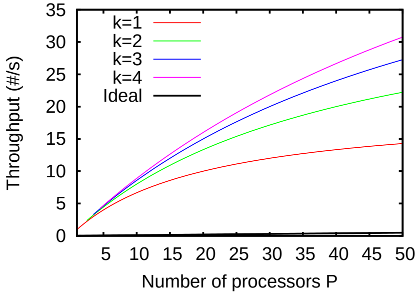
Throughput example

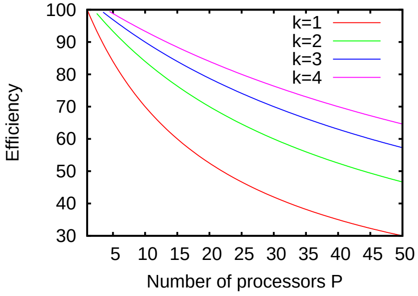
$$H(P) = \frac{N}{\text{Time}(P)}$$

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

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

Say $N = 100$:





Big Lesson #1

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

Less ideal case of Amdahl's law

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

$$\mathbf{T_s \approx P T_1}$$

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

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

Amdahl:

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

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

$$\mathbf{T_s \approx P T_1}$$

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

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

Amdahl:

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

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

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

$$\mathbf{T_s \approx PT_1}$$

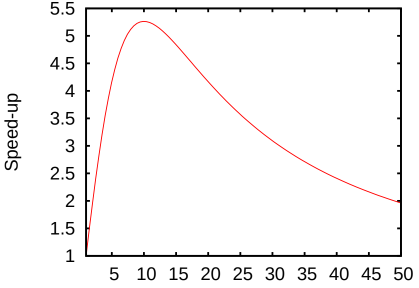
Serial fraction now a function of **P**:

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

Amdahl:

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

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

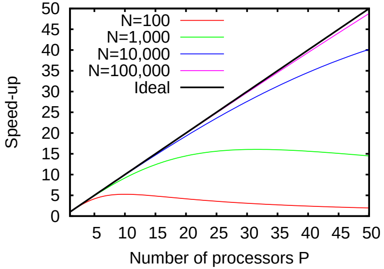


Trying to beat Amdahl's law #1

Scale up!

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

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

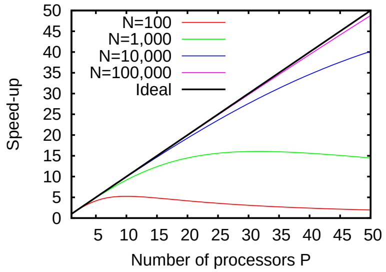


Trying to beat Amdahl's law #1

Scale up!

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

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



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

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

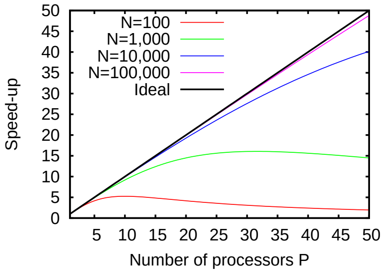
Good weak scaling means this time approaches a constant for large **P**.

Trying to beat Amdahl's law #1

Scale up!

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

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



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

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

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

Gustafson's Law

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

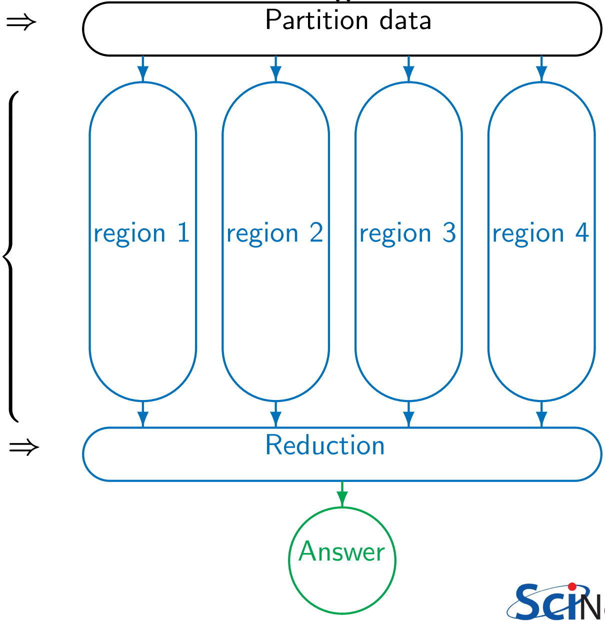
Trying to beat Amdahl's law #2

Parallel overhead ⇒

Parallel region ⇒

Perfectly Parallel (for large **N**)

Serial portion ⇒



Trying to beat Amdahl's law #2

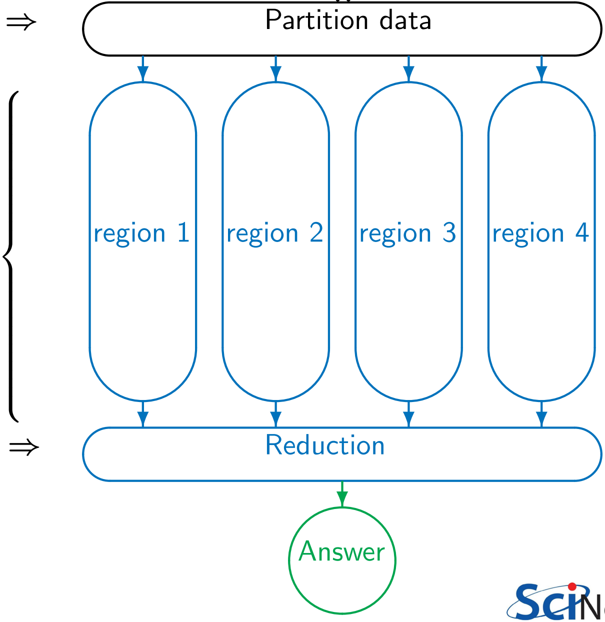
Parallel overhead ⇒

Parallel region ⇒

Perfectly Parallel (for large **N**)

Serial portion ⇒

Rewrite



Trying to beat Amdahl's law #2

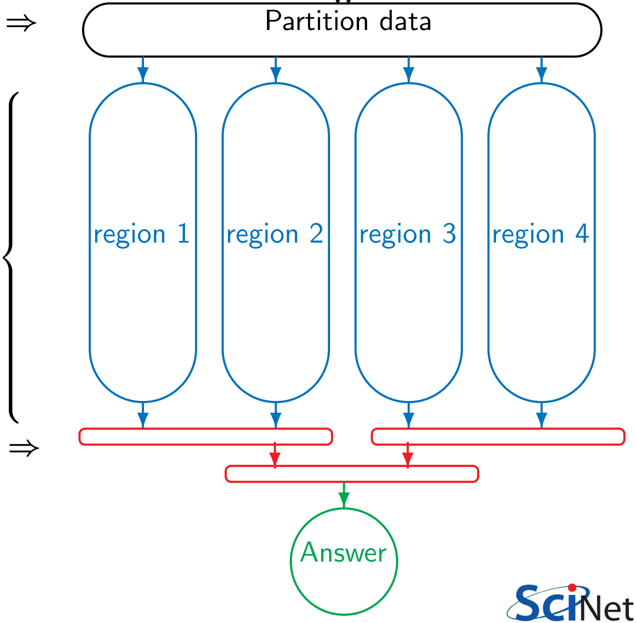
Parallel overhead ⇒

Parallel region ⇒

Perfectly Parallel
(for large **N**)

Serial portion ⇒

Rewrite



Trying to beat Amdahl's law #2

Parallel overhead ⇒

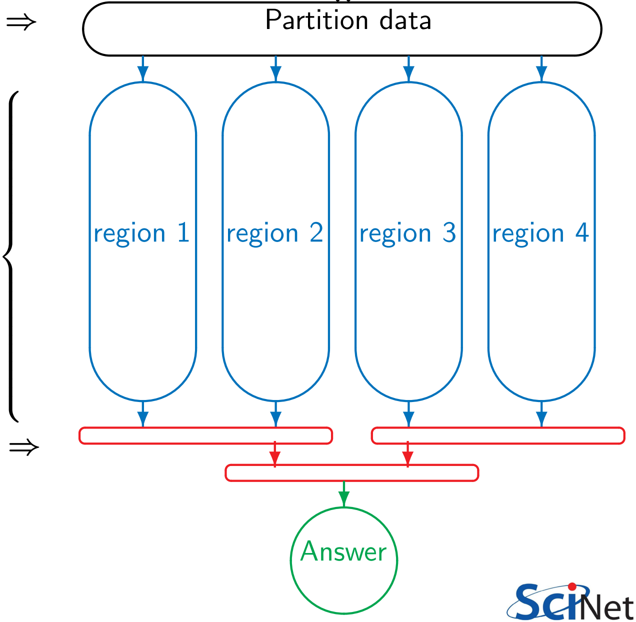
Parallel region ⇒

Perfectly Parallel
(for large **N**)

Serial portion ⇒

Rewrite

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



Trying to beat Amdahl's law #2

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

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

Amdahl:

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

Trying to beat Amdahl's law #2

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

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

Amdahl:

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

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

Trying to beat Amdahl's law #2

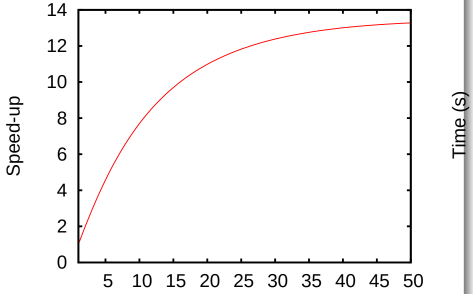
'Serial' fraction now different function of **P**:

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

Amdahl:

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

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



Trying to beat Amdahl's law #2

Weak scaling

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

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

Trying to beat Amdahl's law #2

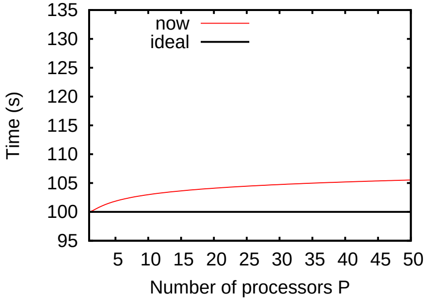
Weak scaling

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

Should approach constant for large **P**.

Let's see. . .

Not quite!



Trying to beat Amdahl's law #2

Weak scaling

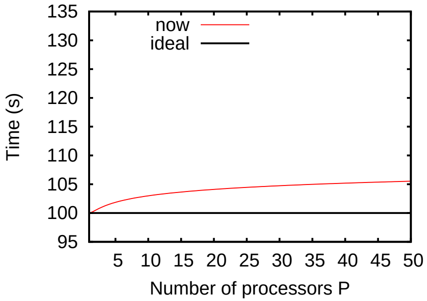
$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

Should approach constant for large **P**.

Let's see. . .

Not quite!

But much better than before.



Trying to beat Amdahl's law #2

Weak scaling

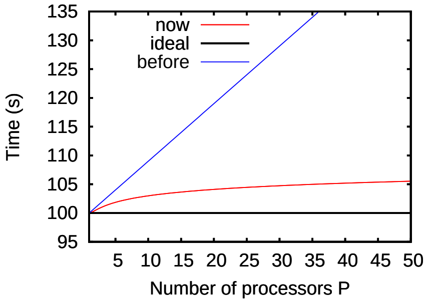
$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

Should approach constant for large **P**.

Let's see. . .

Not quite!

But much better than before.

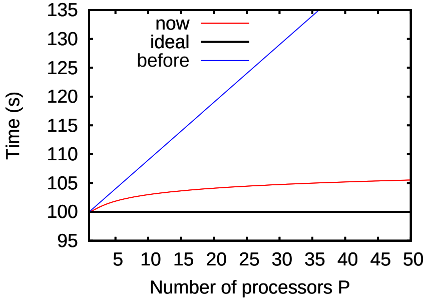


Trying to beat Amdahl's law #2

Weak scaling

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

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



Gustafson?

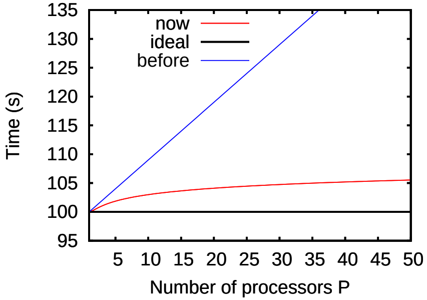
It turns out that Gustafson's law assumes that the serial cost does not change with **P**.
Here that grows logarithmically with **P**, and this is reflected in the weak scaling.

Trying to beat Amdahl's law #2

Weak scaling

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

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



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.

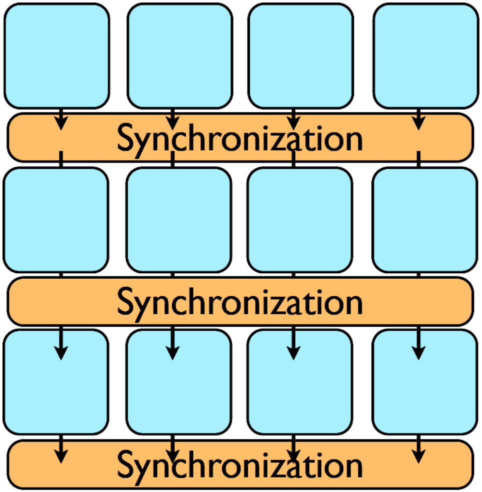
and other algorithms can do better.

Big Lesson #2

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

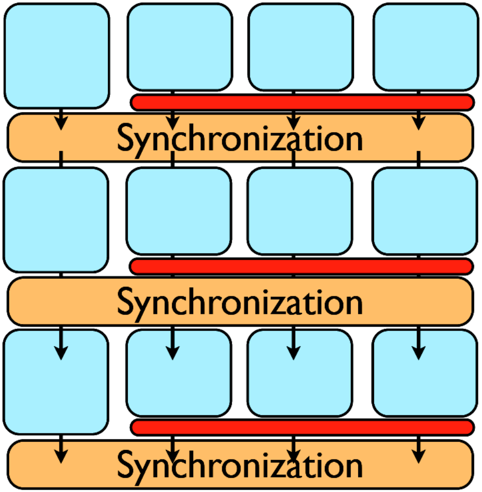
Synchronization

- 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.



Locality

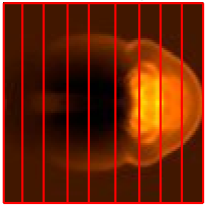
- So far we neglected communication costs.
- But communication costs are more expensive than computation!
- To minimize communication to computation ratio:
    - Keep the data where it is needed.
    - Make sure as little data as possible is to be communicated.
    - Make shared data as local to the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

Locality

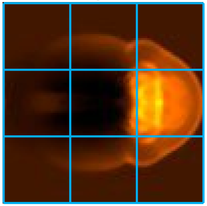
- So far we neglected communication costs.
- But communication costs are more expensive than computation!
- To minimize communication to computation ratio:
  - * Keep the data where it is needed.
  - * Make sure as little data as possible is to be communicated.
  - * Make shared data as local to the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

Example (PDE Domain decomposition)

wrong



right



Big Lesson #3

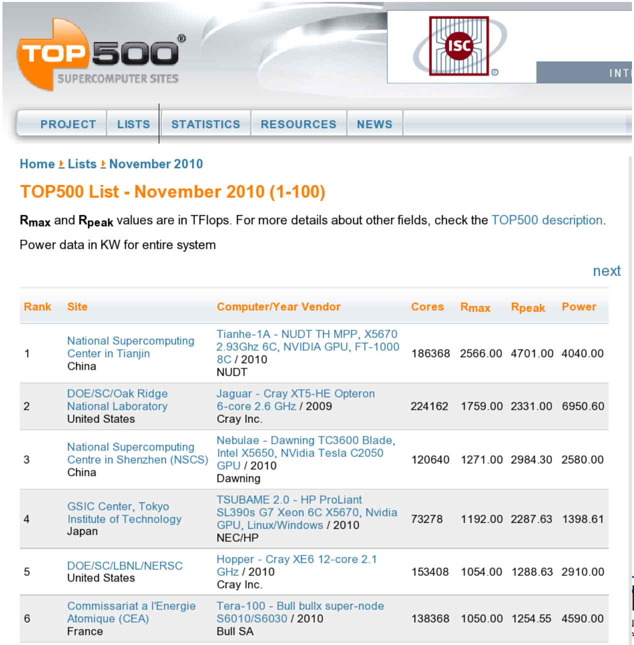
Parallel algorithm design is about finding as much concurrency as possible, and arranging it in a way that maximizes locality.

Parallel Computers

Top500.org:

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

Info on architecture, etc.



Home ▸ Lists ▸ November 2010

TOP500 List - November 2010 (1-100)

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

Power data in KW for entire system

next

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

Supercomputer architectures

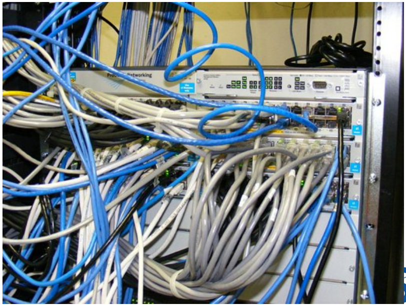
- Clusters, or, distributed memory machines
  In essence a bunch of desktops linked together by a network ("interconnect"). Easy and cheap.
- Multi-core machines, or, shared memory machines
  These can see the same memory. Limited number of cores, typically, and much more $$$.
- Vector machines.
  These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

Most supercomputers are a hybrid combo of these different architectures.

Distributed Memory: Clusters

Simplest type of parallel computer to build

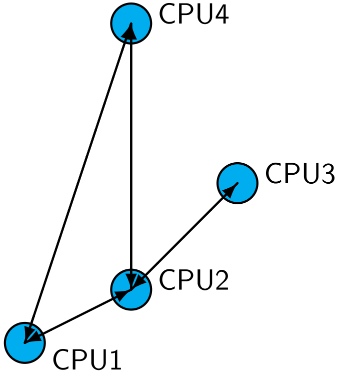
- Take existing powerful standalone computers
- And network them



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

Distributed Memory: Clusters

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

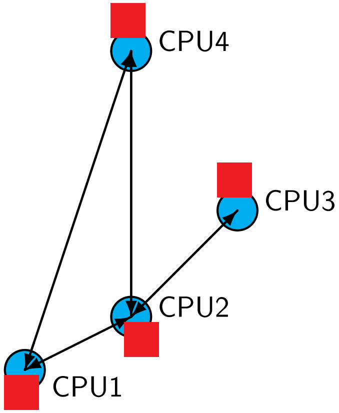


Distributed Memory: Clusters

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

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

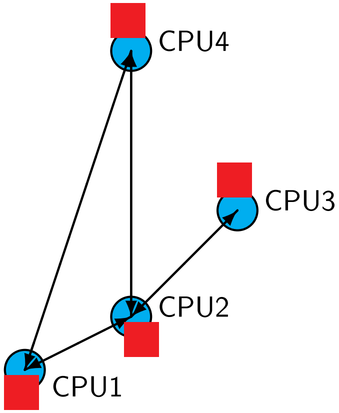
Usual model: "message passing"



Clusters+Message Passing

Hardware:
Easy to build
(Harder to build well)
Can build larger and larger
clusters relatively easily

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



Cluster Communication Cost

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

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

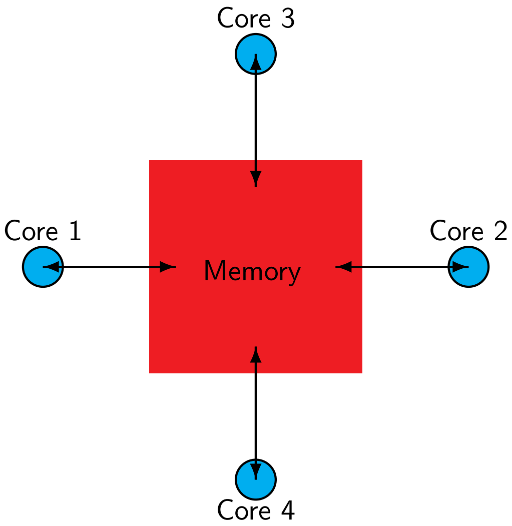
Shared Memory

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

Any coordination done through memory

Could use message passing, but no need.

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



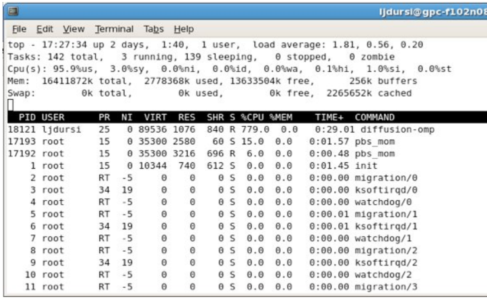
Threads versus Processes

Threads:

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

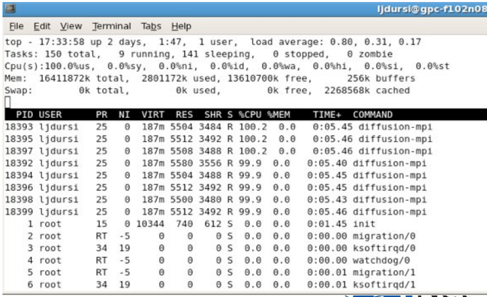
Processes:

Independent tasks with their own memory and resources



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

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



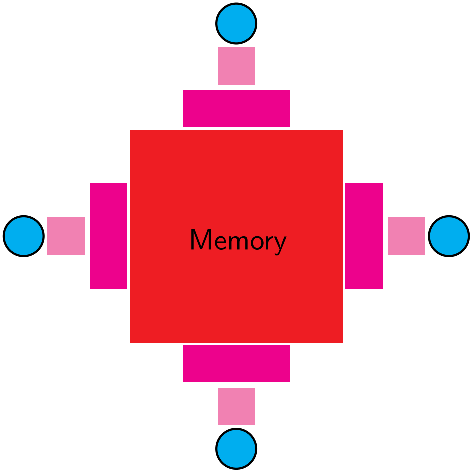
```
top - 17:33:58 up 2 days,  1:47,  1 user,  load average: 0.80, 0.31, 0.17
Tasks: 150 total,   9 running, 141 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  16411872k total,  2801172k used, 13610700k free,      256k buffers
Swap:        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 S  0.0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    4 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.01 migration/1
    6 root      34  19     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/1
```

Shared Memory: NUMA

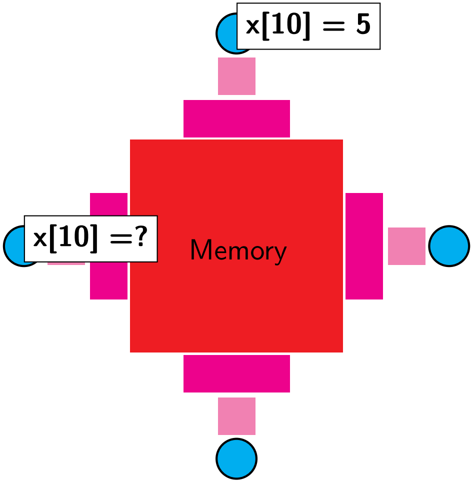
Non-Uniform Memory Access

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



Coherency

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



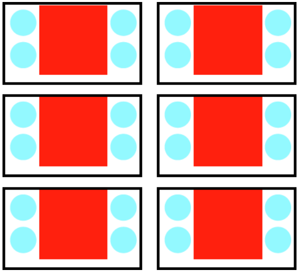
Shared Memory Communication Cost

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

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

Hybrid Architectures

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



We will focus on the aspects that affect the programmer:

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

Big Lesson #4

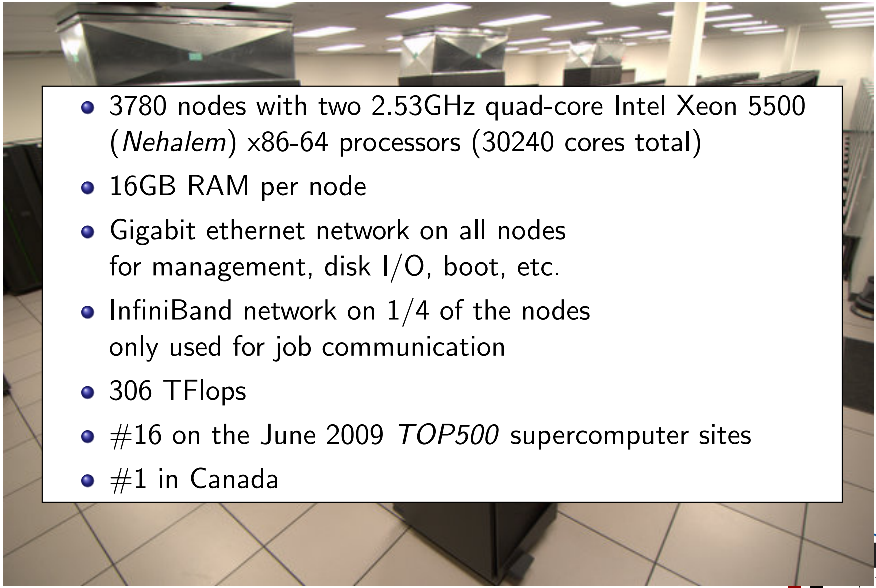
The best approach to parallelizing your problem will depend on both details of your problem and of the hardware available.

Resources at SciNet

1. General Purpose Cluster (GPC)

Resources at SciNet

1. General Purpose Cluster (GPC)



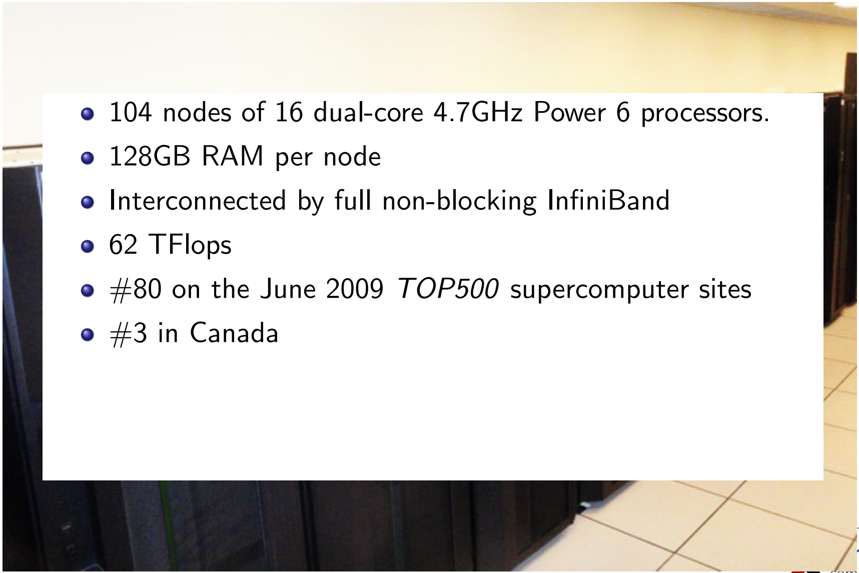
- 3780 nodes with two 2.53GHz quad-core Intel Xeon 5500 (*Nehalem*) x86-64 processors (30240 cores total)
- 16GB RAM per node
- Gigabit ethernet network on all nodes for management, disk I/O, boot, etc.
- InfiniBand network on 1/4 of the nodes only used for job communication
- 306 TFlops
- #16 on the June 2009 *TOP500* supercomputer sites
- #1 in Canada

Resources at SciNet

2. Tightly Coupled System (TCS)

Resources at SciNet

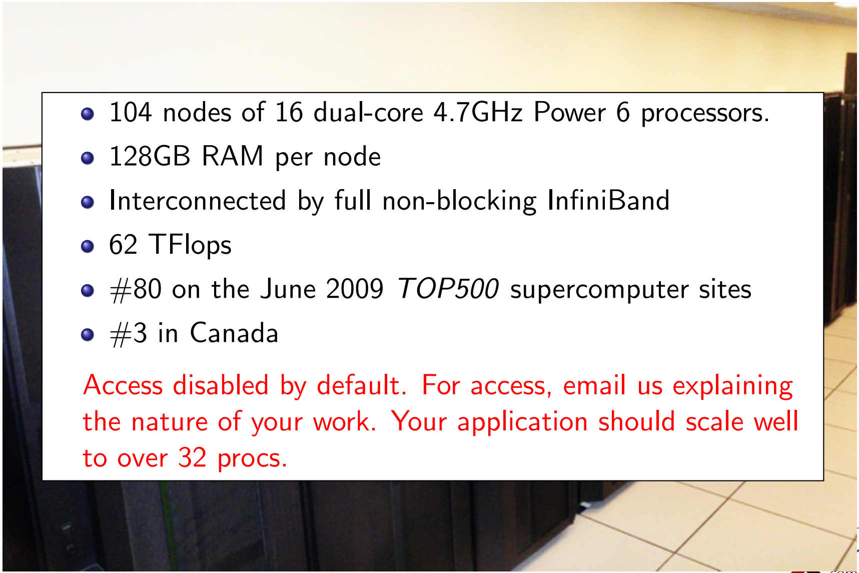
2. Tightly Coupled System (TCS)



- 104 nodes of 16 dual-core 4.7GHz Power 6 processors.
- 128GB RAM per node
- Interconnected by full non-blocking InfiniBand
- 62 TFlops
- #80 on the June 2009 *TOP500* supercomputer sites
- #3 in Canada

Resources at SciNet

2. Tightly Coupled System (TCS)



- 104 nodes of 16 dual-core 4.7GHz Power 6 processors.
- 128GB RAM per node
- Interconnected by full non-blocking InfiniBand
- 62 TFlops
- #80 on the June 2009 *TOP500* supercomputer sites
- #3 in Canada

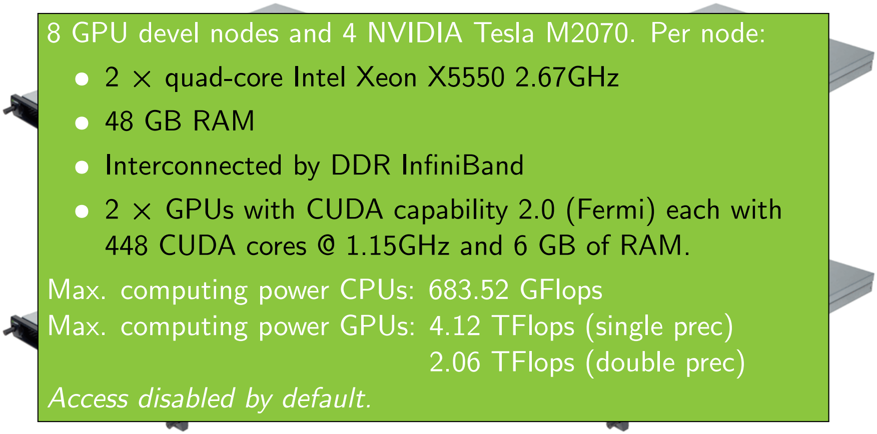
Access disabled by default. For access, email us explaining the nature of your work. Your application should scale well to over 32 procs.

Resources at SciNet

3. Accelerator Research Cluster (ARC)

Resources at SciNet

3. Accelerator Research Cluster (ARC)



8 GPU devel nodes and 4 NVIDIA Tesla M2070. Per node:

- 2 × quad-core Intel Xeon X5550 2.67GHz
- 48 GB RAM
- Interconnected by DDR InfiniBand
- 2 × GPUs with CUDA capability 2.0 (Fermi) each with 448 CUDA cores @ 1.15GHz and 6 GB of RAM.

Max. computing power CPUs: 683.52 GFlops
Max. computing power GPUs: 4.12 TFlops (single prec)
2.06 TFlops (double prec)

Access disabled by default.

Part III

Review of C

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

Example (Basic C program)

```
#include <stdio.h>
// include stdio.h to print
int main() // always called first
{ // braces delimit code block
  printf("Hello world.\n");
  // function call to print
  // line ends with a semicolon
  return 0;
  // optional return value to shell
}
```

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

Example (Basic C program)

```c
#include <stdio.h>
// include stdio.h to print
int main() // always called first
{ // braces delimit code block
  printf("Hello world.\n");
  // function call to print
  // line ends with a semicolon
  return 0;
  // optional return value to shell
}
```

```
$ gcc -o hello hello.c -std=c99 ⎧-O2
                                ⎨-Os
                                ⎨-O3
                                ⎩-Ofast
$
```

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

Example (Basic C program)

```c
#include <stdio.h>
// include stdio.h to print
int main() // always called first
{ // braces delimit code block
  printf("Hello world.\n");
  // function call to print
  // line ends with a semicolon
  return 0;
  // optional return value to shell
}
```

```
                                 ⎧-O2
                                 ⎪-Os
$ gcc -o hello hello.c -std=c99 ⎨-O3
                                 ⎩-Ofast

$ ./hello
Hello world.
$
```

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

Example (Basic C program)

```c
#include <stdio.h>
// include stdio.h to print
int main() // always called first
{ // braces delimit code block
  printf("Hello world.\n");
  // function call to print
  // line ends with a semicolon
  return 0;
  // optional return value to shell
}
```

```
                                        -O2
                                        -Os
$ gcc -o hello hello.c -std=c99  {
                                        -O3
                                        -Ofast

$ ./hello
Hello world.
$ echo $?
0
$ █
```

C review: Functions

Function declaration (prototype)

```
returntype name(argument-spec);
```

Function definition

```
returntype name(argument-spec) {
  statements
}
```

Function call

```
var=name(arguments);
f(name(arguments));
```

Procedures

Procedures are functions with return-type `void` ; called without assignment.

C review: Variables

Define a variable with

```
type name [= value];
```

- *type* may be a
  - * built-in type:
    - floating point type:
      `float`, `double`, `long double`
    - integer type:
      `short`,[`unsigned`] `int`, [`unsigned`] `long int` ,[`unsigned`] `long long int`
    - character or string of characters:
      `char`, `char*`
  - * array, pointer
  - * structure, enumerated type, union
- Variable declarations and code may be mixed in C99.
- Variables can be initialized to a *value* when declared.
  Any non-initialized variable is not set to zero, but has a random value!

C review: Loops

```
for (initialization; condition; increment) {
  statements
}
```

```
while (condition) {
  statements
}
```

You can use `break` to exit the loop.

C review: Loops

```
for (initialization; condition; increment) {
  statements
}
```

```
while (condition) {
  statements
}
```

You can use `break` to exit the loop.

Example

```
#include <stdio.h>
int main() {
  for (int i=1; i<=10; i++)
    printf("%d ",i);
  // note the omitted braces
  printf("\n");
}
```

compute • calcul CANADA

C review: Loops

```
for (initialization; condition; increment) {
  statements
}
```

```
while (condition) {
  statements
}
```

You can use `break` to exit the loop.

Example

```
#include <stdio.h>
int main() {
  for (int i=1; i<=10; i++)
    printf("%d ",i);
  // note the omitted braces
  printf("\n");
}
```

```
$ gcc -o count count.c -O2 -std=c99
$ ./count
1 2 3 4 5 5 6 7 8 9 10
$ █
```

compute • calcul
CANADA

C review: Pointers

```
type *name;
```

C review: Pointers

type **name*;

Example (Pointer assignment)

```
#include <stdio.h>
int main() {
  int a=7,b=5;
  int *ptr=&a;
  a = 13;
  b = *ptr;
  printf("b=%d\n,b);
}
```

```
$ gcc -o ptrex ptrex.c -O2 -std=c99
$ ./ptrex
b=13
$ █
```

C review: Pointers

```
type *name;
```

Example (Pointer assignment)

```c
#include <stdio.h>
int main() {
  int a=7,b=5;
  int *ptr=&a;
  a = 13;
  b = *ptr;
  printf("b=%d\n,b);
}
```

```
$ gcc -o ptrex ptrex.c -O2 -std=c99
$ ./ptrex
b=13
$ █
```

Example (Pass by reference)

```c
void inc(int *i) { (*i)++; }
int main() {
  int j=10;
  inc(&j);
  return j;
}
```

```
$ gcc -o passref passref.c -O2 -std=c99
$ ./passref
$ echo $?
11
$ █
```

C review: Automatic arrays

```
type name[number];
```

- *name* is equivalent to a pointer to the first element.
- Usage `name[i]`. Equivalent to `*(name+i)`.
- C arrays are zero-based.

C review: Automatic arrays

```
type name[number];
```

- *name* is equivalent to a pointer to the first element.
- Usage `name[i]`. Equivalent to `*(name+i)`.
- C arrays are zero-based.

Example

```c
#include <stdio.h>
int main() {
  int a[10]={1,2,3,4,5,6,7,8,9,11};
  int sum=0;
  for (int i=0; i<10; i++)
    sum += a[i];
  printf("sum=%d\n,sum);
}
```

```
$ gcc -o autoarr autoarr.c -O2 -std=c99
$ ./autoarr
56
$ 
```

C review: Automatic arrays

```
type name[number];
```

- *name* is equivalent to a pointer to the first element.
- Usage *name*[i]. Equivalent to *(*name*+i).
- C arrays are zero-based.

Example

```c
#include <stdio.h>
int main() {
  int a[10]={1,2,3,4,5,6,7,8,9,11};
  int sum=0;
  for (int i=0; i<10; i++)
    sum += a[i];
  printf("sum=%d\n,sum);
}
```

```
$ gcc -o autoarr autoarr.c -O2 -std=c99
$ ./autoarr
56
$ █
```

Gotcha:

- There's an compiler dependent limit on *number*.
- C standard only says at least 65535 bytes.

C review: Dynamically allocated arrays

Requires header file:

```
#include <stdlib.h>
```

Defined as a pointer to memory:

```
type *name;
```

Allocated by a function call:

```
name=malloc(number*sizeof(type));
```

Usages:

```
a=name[number];
```

Deallocated by a function call:

```
free(name);
```

- System function call can access all available memory.
- Can check if allocation failed ($name == 0$).
- Can control when memory is given back.

C review: Dynamically allocated arrays

Example

C review: Dynamically allocated arrays

Example

```c
#include <stdlib.h>
#include <stdio.h>
void printarr(int n, int *a) {
  for (int i=0;i<n;i++)
    printf("%d ", a[i]);
  printf("\n");
}
int main(){
  int n=100;
  int *b=malloc(n*sizeof(*b));
  for (int i=0;i<n;i++)
    b[i]=i*i;
  printarr(n,b);
  free(b);
}
```

C review: Dynamically allocated arrays

Example

```c
#include <stdlib.h>
#include <stdio.h>
void printarr(int n, int *a) {
  for (int i=0;i<n;i++)
    printf("%d ", a[i]);
  printf("\n");
}
int main(){
  int n=100;
  int *b=malloc(n*sizeof(*b));
  for (int i=0;i<n;i++)
    b[i]=i*i;
  printarr(n,b);
  free(b);
}
```

```
$ gcc -o dynarr dynarr.c -O2 -std=c99
$ ./dynarr
0 1 4 9 16 25 36 49 64 81 100 121 144
169 196 225 256 289 324 361 400 441
484 529 576 625 676 729 784 841 900
961 1024 1089 1156 1225 1296 1369 1444
1521 1600 1681 1764 1849 1936 2025
2116 2209 2304 2401 2500 2601 2704
2809 2916 3025 3136 3249 3364 3481
3600 3721 3844 3969 4096 4225 4356
4489 4624 4761 4900 5041 5184 5329
5476 5625 5776 5929 6084 6241 6400
6561 6724 6889 7056 7225 7396 7569
7744 7921 8100 8281 8464 8649 8836
9025 9216 9409 9604 9801
$ █
```

C review: Structs = collections of other variables

```
struct name {
  type1 name1;
  type2 name2;
  ...
};
```

C review: Structs = collections of other variables

```
struct name {
  type1 name1;
  type2 name2;
  ...
};
```

Example

```
#include <string.h>
#include <stdio.h>
struct Info {
  char name[100];
  unsigned int age;
};
int main() {
  struct Info my;
  my.age=38;
  strcpy(my.name,"Ramses");
  printf("%d %s\",my.age,my.name);
}
```

C review: Structs = collections of other variables

```
struct name {
  type1 name1;
  type2 name2;
  ...
};
```

Example

```c
#include <string.h>
#include <stdio.h>
struct Info {
  char name[100];
  unsigned int age;
};
int main() {
  struct Info my;
  my.age=38;
  strcpy(my.name,"Ramses");
  printf("%d %s\",my.age,my.name);
}
```

```
$ gcc -o info info.c -O2 -std=c99
$ ./info
Ramses 38
$ █
```

C review: Conditionals

```
if (condition) {
  statements
} else if (other condition) {
  statements
} else {
  statements
}
```

Example

C review: Conditionals

```
if (condition) {
  statements
} else if (other condition) {
  statements
} else {
  statements
}
```

Example

```
int main(){
  int n=20;
  int *b= malloc(n*sizeof(*b));
  if (b==0)
    return 1; //error
  else {
    for (int i=0;i<n;i++)
      b[i]=i*i;
    printarr(n,b);
    free(b);
  }
}
```

C review: Conditionals

```
if (condition) {
  statements
} else if (other condition) {
  statements
} else {
  statements
}
```

Example

```
int main(){
  int n=20;
  int *b= malloc(n*sizeof(*b));
  if (b==0)
    return 1; //error
  else {
    for (int i=0;i<n;i++)
      b[i]=i*i;
    printarr(n,b);
    free(b);
  }
}
```

```
$ gcc -o ifm ifm.c -O2 -std=c99
$ ./ifm
0 1 4 9 16 25 36 49 64 81 100 121
144 169 196 225 256 289 324 361
$ █
```

C review: Multidimensional arrays

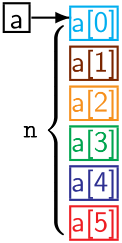
```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```

C review: Multidimensional arrays

```
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```

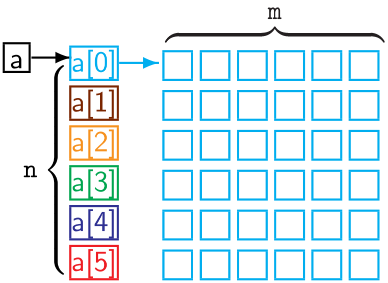
C review: Multidimensional arrays

```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



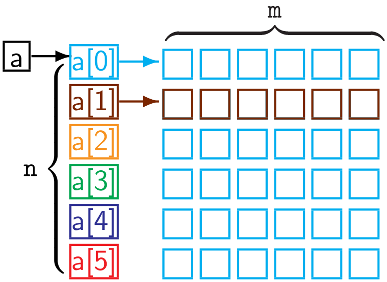
C review: Multidimensional arrays

```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



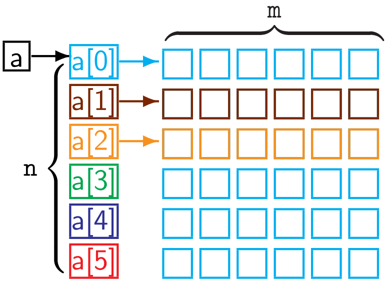
C review: Multidimensional arrays

```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



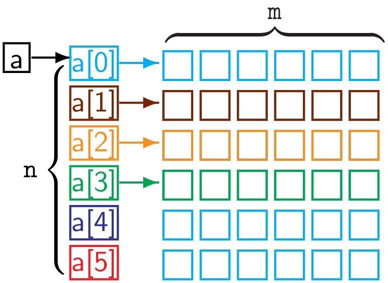
C review: Multidimensional arrays

```
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



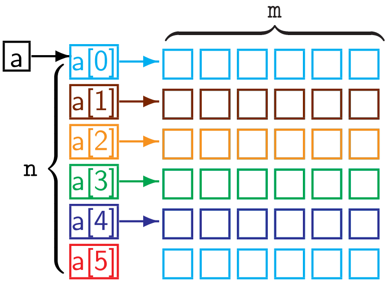
C review: Multidimensional arrays

```
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



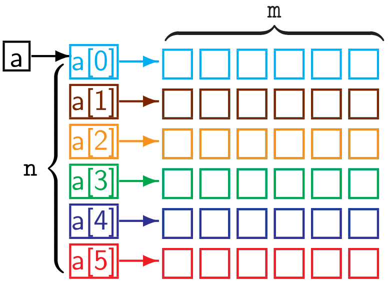
C review: Multidimensional arrays

```
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



C review: Multidimensional arrays

```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



C review: Libraries

Usage

- Put an include line in the source code, e.g.

```
#include <stdio.h>
#include <omp.h>
#include "mpi.h"
```

- Include the libraries at link time using `-l[libname]`.
  Implicit for most standard libraries, with `mpicc` and `gcc -fopenmp`.

C review: Libraries

Usage

- Put an include line in the source code, e.g.

```
#include <stdio.h>
#include <omp.h>
#include "mpi.h"
```

- Include the libraries at link time using `-l[libname]`.
  Implicit for most standard libraries, with `mpicc` and `gcc -fopenmp`.

Common standard libraries

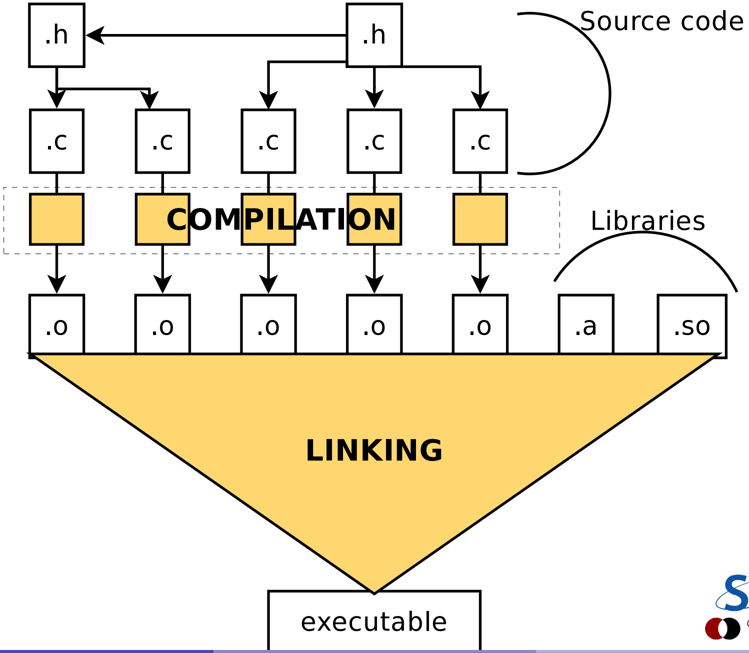
- stdio.h: input/output, e.g., `printf` and `fwrite`
- stdlib.h: memory, e.g. `malloc`
- string.h: strings, memory copies, e.g. `strcpy`
- math.h: special functions, e.g. `sqrt`.
  When using math, you need to link with `-lm`.

CANADA

Compilation:

Building with make

Compilation workflow



Compiling with make

Single source file

```
# This file is called makefile
CC      = gcc
CFLAGS  = -std=c99 -O2
LDFLAGS = -lm
main:  main.c
   $(CC) $(CFLAGS) $(LDFLAGS) $^ -o $@
```

Compiling with make

Single source file

```
# This file is called makefile
CC      = gcc
CFLAGS  = -std=c99 -O2
LDFLAGS = -lm
main:  main.c
   $(CC) $(CFLAGS) $(LDFLAGS) $^ -o $@
```

Multiple source file application

```
CC      = gcc
CFLAGS  = -std=c99 -O2
LDFLAGS = -lm
main:  main.o mylib.o
   $(CC) $(LDFLAGS) $^ -o $@
main.o:  main.c mylib.h
mylib.o:  mylib.h mylib.c
clean:
   rm -f main.o mylib.o
```

Compiling with make

When typing `make` at command line:

- Checks if `main.c` or `mylib.c` or `mylib.h` were changed.
- If so, invokes corresponding rules for object files.
- Only compiles changed code files: faster recompilation.
- Parallel make:

```
$ make -j 3
```

Compiling with make

When typing `make` at command line:

- Checks if `main.c` or `mylib.c` or `mylib.h` were changed.
- If so, invokes corresponding rules for object files.
- Only compiles changed code files: faster recompilation.
- Parallel make:

```
$ make -j 3
```

Gotcha

- Make does not detect changes in compiler, or in system.
- But .o files are system/compiler dependent, so need to be recompiled.
- Always specify a "clean" rule in the makefile, so that moving from one system or compiler to another, you can do a fresh rebuild:

```
$ make clean
$ make
```

Before we start with OpenMP. . .

```
$ ssh -X login.scinet.utoronto.ca
$ ssh -X gpc01
$ qsub -X -I -l nodes=1:ppn=8,walltime=5:00:00,os=centos53develibA
```

to get a dedicated development node (ensure this works).

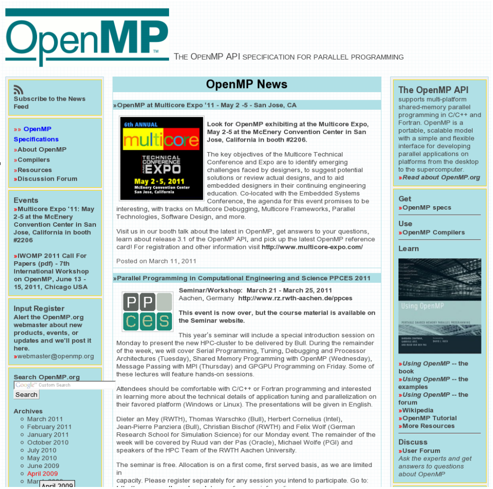
```
$ cp -r ~ljdursi/ppp ~/
$ source ~/ppp/setup
$ cd ~/ppp
$ cd util
$ make
$ cd ~/ppp/omp-intro
$ make mandel
$ mandel
```

Part V

Introduction to OpenMP

OpenMP

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



OpenMP™ The OpenMP API specification for parallel programming

Subscribe to the News Feed

»» OpenMP Specifications
»About OpenMP
»Compilers
»Resources
»Discussion Forum

Events
»Multicore Expo '11: May 2-5 at the McEnery Convention Center in San Jose, California in booth #2206

»IWOMP 2011 Call For Papers (pdf) - 7th International Workshop on OpenMP, June 13 - 15, 2011, Chicago USA

Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Search OpenMP.org
Search

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

OpenMP News

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

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

The key objectives of the Multicore 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 Multicore Debugging, Multicore 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.multicore-expo.com/**

Posted on March 11, 2011

»Parallel Programming in Computational Engineering and Science PPCES 2011

Seminar/Workshop: March 21 - March 25, 2011
Aachen, Germany **http://www.rz.rwth-aachen.de/ppces**

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 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 GPGPU 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.

Dieter an Mey (RWTH), Thomas Warschko (Bull), Herbert Cornelius (Intel), Jean-Pierre Panziera (Bull), Christian Bischof (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 Pas (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:

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

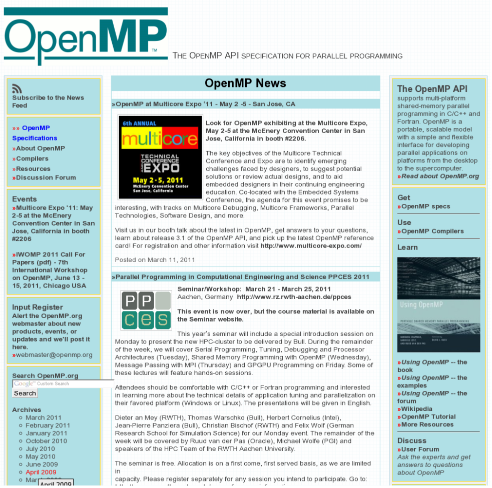
»*Using OpenMP* -- the book
»*Using OpenMP* -- the examples
»*Using OpenMP* -- the forum
»Wikipedia
»OpenMP Tutorial
»More Resources

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

OpenMP

- 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.



THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

Subscribe to the News Feed

»» OpenMP Specifications
»About OpenMP
»Compilers
»Resources
»Discussion Forum

Events
»Multicore Expo '11: May 2-5 at the McEnery Convention Center in San Jose, California in booth #2206

»IWOMP 2011 Call For Papers (pdf) - 7th International Workshop on OpenMP, June 13 - 15, 2011, Chicago USA

Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Search OpenMP.org
Search

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

OpenMP News

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

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

The key objectives of the Multicore 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 Multicore Debugging, Multicore 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.multicore-expo.com/**

Posted on March 11, 2011

»Parallel Programming in Computational Engineering and Science PPCES 2011

Seminar/Workshop: March 21 - March 25, 2011
Aachen, Germany **http://www.rz.rwth-aachen.de/ppces**

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 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 GPGPU 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.

Dieter an Mey (RWTH), Thomas Warschko (Bull), Herbert Cornelius (Intel), Jean-Pierre Panziera (Bull), Christian Bischof (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 Pas (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:

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, you add lines starting with `#pragma omp`.
  This parallelizes the subsequent code block.
- In Fortran, you add lines starting with `!$omp`.
  An `!$omp end ...` is needed to close the parallel region.
- These lines are skipped (for C, sometimes with a warning) by compilers that do not support OpenMP.

When compiling:

- To turn on OpenMP support in gcc and gfortran, 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());
  }
}
```

F90:

```
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

OpenMP example

```
$ gcc -std=c99 -O2 -o omp-hello-world omp-hello-world.c -fopenmp
or
$ gfortran -O2 -o omp-hello-world omp-hello-world.f90 -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 -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.
$ 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?

```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());
  }
}
```

```fortran
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

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());
  }
}
```

Program starts normally (single thread)

```
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

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!

```
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

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.

```
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

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());
  }
}
```

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

```
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

OpenMP functions (from omp.h/omp_lib)

```
#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/omp_lib)

```
#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/omp_lib)

```
#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/omp_lib)

```
#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 it own copy, invisible for others
  Initial and final value undefined!
  (Advanced: firstprivate, lastprivate – copy in/out.)

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, lauches 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, lauches 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, lauches 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 - Fortran version

```
program omp_vars
use omp_lib
implicit none
integer ::  mythread, nthreads
!$omp parallel default(none) private(mythread) shared(nthreads)
  mythread = omp_get_thread_num()
  if (mythread == 0) then
    nthreads = omp_get_num_threads()
  endif
!$omp end parallel
print *,'Number of threads was ', nthreads, '.'
end program omp_vars
```

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);
}
```

```
program omp_vars
use omp_lib
implicit none
integer ::  nthreads
!$omp parallel default(none) shared(nthreads)
!$omp single
  nthreads = omp_get_num_threads()
!$omp end single
!$omp end parallel
print *,'Number of threads was ', nthreads, '.'
end program omp_vars
```

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.

```c
#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);
  }
}
```

```fortran
program omp_loop
use omp_lib
implicit none
integer ::  i, mythread
!$omp parallel default(none) &
!$omp XXXX(i) XXXX(mythread)
  mythread = omp_get_thread_num()
  do i=1,16
    print *, 'thread ', mythread, &
             ' gets i=', i
  enddo
!$omp end parallel
end program omp_loop
```

Loops in OpenMP

Take one of your openmp programs and add a loop.

```c
#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);
  }
}
```

```fortran
program omp_loop
use omp_lib
implicit none
integer ::  i, mythread
!$omp parallel default(none) &
!$omp XXXX(i) XXXX(mythread)
  mythread = omp_get_thread_num()
  do i=1,16
    print *, 'thread ', mythread, &
            ' gets i=', i
  enddo
!$omp end parallel
end program omp_loop
```

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 (or omp do).

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 (or omp do).

```
#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);
  }
}
```

Fortran version

```
program omp_loop
use omp_lib
implicit none
integer ::  i, mythread
!$omp parallel default(none) XXXX(i) XXXX(mythread)
  mythread = omp_get_thread_num()
  !$omp do
  do i=1,16
    print *, 'thread ', mythread, ' gets i=', i
  enddo
  !$omp end do
!$omp end parallel
end program omp_loop
```

Worksharing constructs in OpenMP

- omp for/omp do 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 or daxpy.f90
- make daxpy or make fdaxpy

$$\mathbf{z} = \mathbf{a}\mathbf{x} + \mathbf{y}$$

Warning

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

```c
#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; `double` is a numerical type which can be set to single or double precision

```
#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).

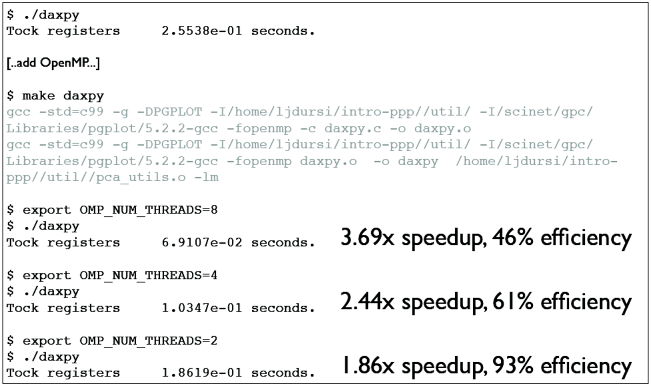
```c
#include <stdio.h>
#include "pca_utils.h"
void daxpy(int n, double a, double *x, double *y, double *z) {
  for (int i=0; i<n; i++) {
    x[i] = (double)i*(double)i;
    y[i] = ((double)i+1.)*((double)i-1.);
  }
  for (int i=0; i<n; i++)
    z[i] += a * x[i] + y[i];
}
int main() {
  int n=1e7;
  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);
}
```

HANDS-ON: Try OpenMPing. . .

HANDS-ON 1:
Parallelize daxpy with OpenMP.
Also do the scaling analysis!

```c
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];
  }
}
```

```fortran
!$omp parallel default(none) private(i) shared(a,x,b,y,z)
!$omp do
do i=1,n
  x(i) = (i)*(i)
  y(i) = (i+1.)*(i-1.)
enddo
!$omp do
do i=1,n
  z(i) = a*x(i) + y(i)
enddo
!$omp end parallel
```



```
$ ./daxpy
Tock registers      2.5538e-01 seconds.

[..add OpenMP...]

$ make daxpy
gcc -std=c99 -g -DPGPLOT -I/home/ljdursi/intro-ppp//util/ -I/scinet/gpc/
Libraries/pgplot/5.2.2-gcc -fopenmp -c daxpy.c -o daxpy.o
gcc -std=c99 -g -DPGPLOT -I/home/ljdursi/intro-ppp//util/ -I/scinet/gpc/
Libraries/pgplot/5.2.2-gcc -fopenmp daxpy.o  -o daxpy  /home/ljdursi/intro-
ppp//util//pca_utils.o -lm

$ export OMP_NUM_THREADS=8
$ ./daxpy
Tock registers      6.9107e-02 seconds.    3.69x speedup, 46% efficiency

$ export OMP_NUM_THREADS=4
$ ./daxpy
Tock registers      1.0347e-01 seconds.    2.44x speedup, 61% efficiency

$ export OMP_NUM_THREADS=2
$ ./daxpy
Tock registers      1.8619e-01 seconds.    1.86x speedup, 93% efficiency
```

```c
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];
  }
}
```

Why is this safe?
Everyone is modifying x,y,z!

```fortran
!$omp parallel default(none) private(i) shared(n,a,x,y,z)
!$omp do
do i=1,n
  x(i) = (i)*(i)
  y(i) = (i+1.)*(i-1.)
enddo
!$omp do
do i=1,n
  z(i) = a*x(i) + y(i)
enddo
!$omp end parallel
```

Dot Product

- Dot product of two vectors
- Implement this, first serially, then with OpenMP
- ndot.c or ndot.f90
- make ndot or make ndotf
- Tells time, answer, correct answer.

$$\mathbf{n} = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} = \sum_{\mathbf{i}} \mathbf{x_i}\, \mathbf{y_i}$$

```
$ ./ndot
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 4.9254e-02 seconds.
```

Dot Product - serial

```c
#include <stdio.h>
#include "pca_utils.h"
double ndot(int n, double *x, double *y){
  double tot=0;
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
int main() {
  int n=1e7;
  double *x = vector(n), *y = vector(n);
  for (int i=0; i<n; i++)
    x[i] = y[i] = i;
  double nn=n-1;
  double ans=nn*(nn+1)*(2*nn+1)/6.0;
  pca_time tt;
  tick(&tt);
  double dot=ndot(n,x,y);
  printf("Dot product is %14.4e (vs %14.4e) for n=%d.  Took %12.4e secs.\n",
    dot, ans, n, tocksilent(&tt));
}
```

Dot Product - serial

```
#include <stdio.h>
#include "pca_utils.h"
double ndot(int n, double *x, double *y){
  double tot=0;
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
int main() {
  int n=1e7;
  double *x = vector(n), *y = vector(n);
  for (int i=0; i<n; i++)
    x[i] = y[i] = i;
  double nn=n-1;
  double ans=nn*(nn+1)*(2*nn+1)/6.0;
  pca_time tt;
  tick(&tt);
  double dot=ndot(n,x,y);
  printf("Dot product is %14.4e (vs %14.4e) for n=%d.  Took %12.4e secs.\n",
    dot, ans, n, tocksilent(&tt));
}
```

```
$ make ndot $ ./ndot
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 4.9254e-02 secs.
```

Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We need the sum from everybody.
- We could make tot shared, then all threads can add to it.

```
double ndot(int n, double *x, double *y){
  double tot=0;
  #pragma omp parallel for default(none) shared(tot,n,x,y)
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_ndot_race
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_race
Dot product is 1.1290e+20
(vs 3.3333e+20) for n=10000000.
Took 5.2628e-02 secs.
```

Not only is the answer wrong, it was slower to compute!

Race Condition - why it's wrong

- Classical parallel bug.
- Multiple writers to some shared resource.
- Can be very subtle, and only appear intermittently.
- Your program can have a bug but not display any symptoms for small runs!
- Primarily a problem with shared memory.

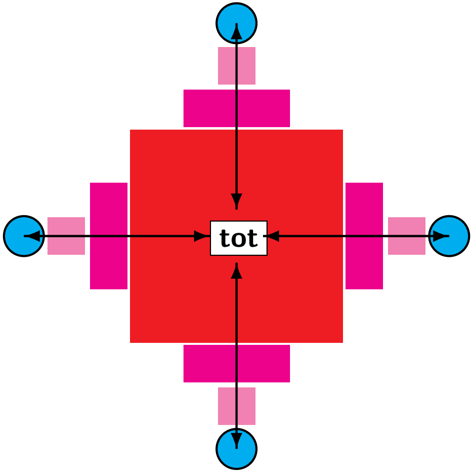
`tot` = 0

Thread 0: add 1	Thread 1: add 2
read `tot`(=0) into register	
reg = reg+1	read `tot`(=0) into register
store reg(=1) into `tot`	reg=reg+2
	store reg(=2) into `tot`

`tot` = 2

Race Condition - why it's slow

- Multiple cores repeatedly trying to read, access, store same variable in memory.
- Not (such) a problem for constants (read only); but a big problem for writing.
- Sections of arrays – better.



OpenMP critical construct

- Defines a critical region.
- Only one thread can be operating within this region at a time.
- Keeps modifications to shared resources saffe.
- `#pragma omp critical`
- `!$omp critical`
  `!$omp end critical`

```
double ndot(int n, double *x, double
*y){
  double tot=0;
  #pragma omp parallel for \
  default(none) shared(tot,n,x,y)
  for (int i=0; i<n; i++)
    #pragma omp critical
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_ndot_critical
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_critical
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 5.1377e+00 secs.
```

Correct, but 100x slower than serial version!

OpenMP atomic construct

- Most hardware has support for atomic instructions (indivisible so cannot get interrupted)
- Small subset, but load/add/stor usually one.
- Not as general as critical
- Much lower overhead.
- `#pragma omp atomic`
- `!$omp atomic`

```
double ndot(int n, double *x, double
*y){
  double tot=0;
  #pragma omp parallel for \
  default(none) shared(tot,n,x,y)
  for (int i=0; i<n; i++)
    #pragma omp atomic
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_ndot_atomic $ export
OMP_NUM_THREADS=8
$ ./omp_ndot_atomic
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 8.5156e-01 secs.
```

Correct, and better – only 16x slower than serial.

How should we fix the slowdown?

- Local sums.
- Each processor sums its local values ($\mathbf{10^7/P}$ additions).
- And then sums to `tot` (only $\mathbf{P}$ additions with critical or atomic. . .
- HANDS-ON: Try it!

$$\mathbf{n} = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} = \sum_{\mathbf{i}} \mathbf{x_i\, y_i} = \sum_{\mathbf{p}} \left(\sum_{\mathbf{i}} \mathbf{x_i\, y_i} \right)$$

HANDS-ON 2:
Parallelize ndot with partial sums.
copy one of the omp_ndot.c's (or fomp_ndot.c's) to omp_ndot_local.c (or fomp_ndot_local.f90).

Local variables

```
tot = 0;
#pragma omp parallel shared(x,y,n,tot)
{
  int mytot = 0;
  #pragma omp for
  for (int i=0; i<n; i++)
    mytot += x[i]*y[i];
  #pragma omp atomic
  tot += mytot;
}
```

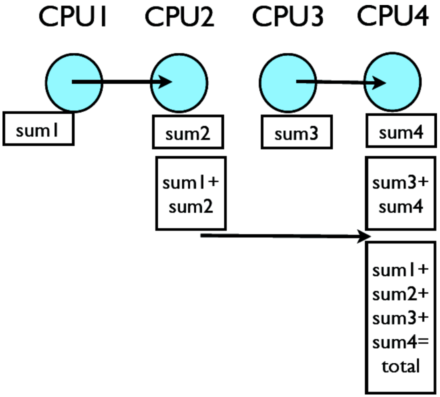
```
ndot = 0.
!$omp parallel shared(x,y,n,ndot) &
!$omp private(i,mytot)
mytot = 0.
!$omp do
do i=1,n
  mytot = mytot + x(i)*y(i)
enddo
!$omp atomic
ndot = ndot + mytot
!$omp end parallel
```

```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 1.7902-02 seconds.
```

Now we're talking! 2.77x faster.

OpenMP Reduction Operations

- This is such a common operation, this is something built into OpenMP to handle it.
- "Reduction" variables - like shared or private.
- Can support several types of operations: - + * ...
- omp_ndot_reduction.c, fomp_ndot_reduction.f90



OpenMP Reduction Operations

```
tot = 0;
#pragma omp parallel \
shared(x,y,n) reduction(+:tot)
{
  #pragma omp for
  for (int i=0; i<n; i++)
    tot += x[i]*y[i];
}
```

```
ndot = 0.
!$omp parallel shared(x,y,n) &
!$omp private(i) reduction(+:ndot)
!$omp do
do i=1,n
  ndot = ndot + x(i)*y(i)
enddo
!$omp end parallel
```

```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 1.8134-02 seconds.
```

Same speed, simpler code!

OpenMP Reduction Operations

```
tot = 0;
#pragma omp parallel for \
shared(x,y,n) reduction(+:tot)
for (int i=0; i<n; i++)
  tot += x[i]*y[i];
```

```
ndot = 0.
!$omp parallel do shared(x,y,n) &
!$omp private(i) reduction(+:ndot)
do i=1,n
  ndot = ndot + x(i)*y(i)
enddo
!$omp end parallel
```

```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 1.8928e-02 seconds.
```

Same speed, simpler code!

Performance

- We threw in 8 cores, got a factor of 3 speedup. Why?
- Often we are limited not by CPU power but by how quickly we can feed CPUs.
- For this problem, we had 107 long vectors, with 2 numbers 8 bytes long flowing through in 0.036 seconds.
- Combined bandwidth from main memory was 4.3 GB/s. Not far off of what we could hope for on this architecture.
- One of the keys to good OpenMP performance is using data when we have it in cache. Complicated functions: easy. Low work-per-element (dot product, FFT): hard.

Load Balancing in OpenMP

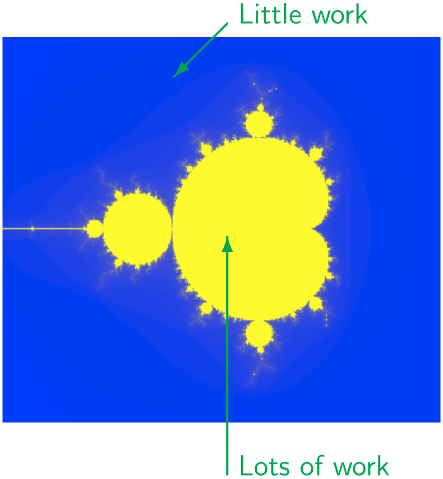
- So far every iteration of the loop had the same amount of work.
- Not always the case
- Sometimes cannot predict beforehand how unbalanced the problem is

OpenMP has work sharing construct that allow you do statically or dynamically balance the load.

Example - Mandelbrot Set

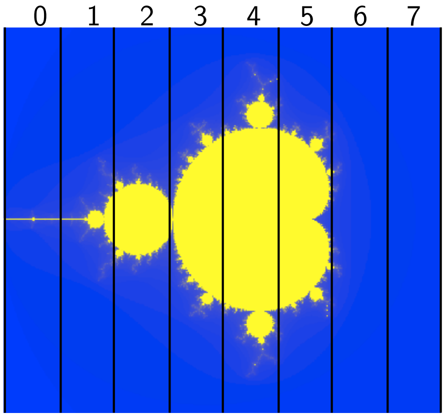
- Mandelbrot set simple example of non-balanced problem.
- Defined as complex points $\mathbf{a}$ where $|\mathbf{b}_\infty|$ finite, with $\mathbf{b_0 = 0}$ and $\mathbf{b_{n+1} = b_n^2 + a}$. If $|\mathbf{b_n}| > \mathbf{2}$, point diverges.
- Calculation:
  - pick some **nmax**
  - iterate for each point **a**, see if crosses 2.
  - Plot **n** or **nmax** as colour.

  Outside of set, points diverge quickly (2-3 steps). Inside, we have to do lots of work (1000s steps).
- make mandel; ./mandel



First OpenMP Mandelbrot Set

- Default work sharing breaks **N** iterations into $\sum$ **N/nthreads** contiguous chunks and assigns them to threads.
- But now threads 7,6,5 will be done and sitting idle while threads 3 and 4 work alone. . .
- Inefficient use of resources.

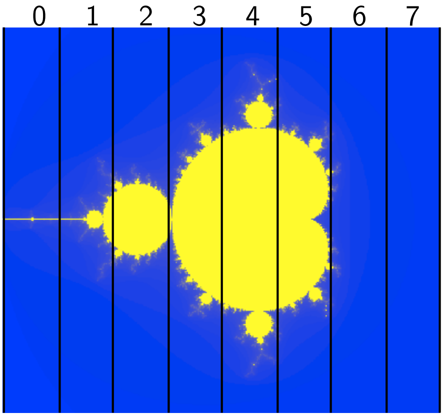


800x800 pix; N/nthreads $\sim$ 100x800

First OpenMP Mandelbrot Set

- Default work sharing breaks **N** iterations into $\sum$ **N/nthreads** contiguous chunks and assigns them to threads.
- But now threads 7,6,5 will be done and sitting idle while threads 3 and 4 work alone. . .
- Inefficient use of resources.

Serial	0.63s
Nthreads=8	0.29s
Speedup	2.2x
Efficiency	27%



800x800 pix; N/nthreads ∼ 100x800

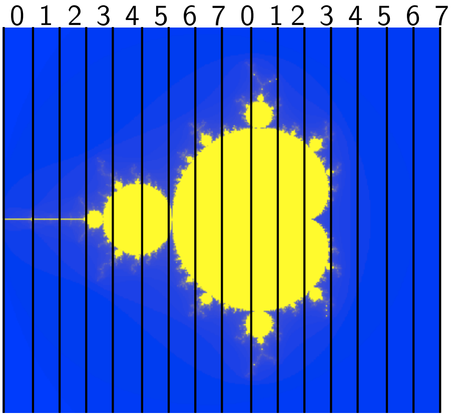
Second Try OpenMP Mandelbrot Set

- Can change the chunk size different from ∼ **N/nthreads**
- In this case, more columns – work distributed a bit better.
- Now, for instance, chunk size 50, and thread 7 gets both a big work chunk and a little one:

```
#pragma omp for schedule(static,50)
```

or

```
!$omp do schedule(static,50)
```



800x800 pix; each threads: 50x800

Second Try OpenMP Mandelbrot Set

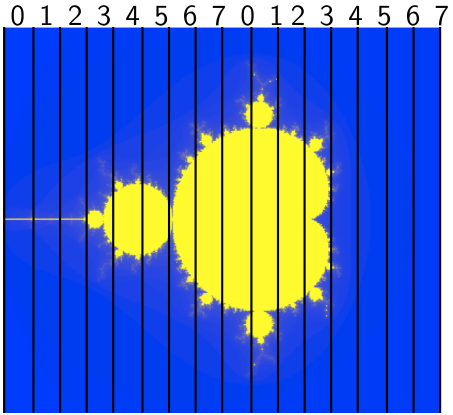
- Can change the chunk size different from $\sim$ **N/nthreads**
- In this case, more columns – work distributed a bit better.
- Now, for instance, chunk size 50, and thread 7 gets both a big work chunk and a little one:

```
#pragma omp for schedule(static,50)
```

or

```
!$omp do schedule(static,50)
```

Serial	0.63s
Nthreads=8	0.15s
Speedup	4.2x
Efficiency	52%



800x800 pix; each threads: 50x800

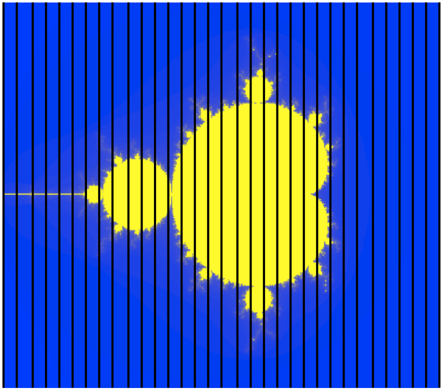
Third Try: Schedule dynamic

- Break up into many pieces and hand them to threads when they are ready.
- Dynamic scheduling.
- Increases overhead, decreases idling threads.
- Can also choose chunk size.

```
#pragma omp for schedule(dynamic)
```

or

```
!$omp do schedule(dynamic)
```



Third Try: Schedule dynamic

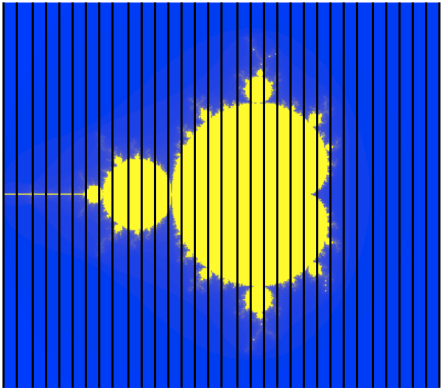
- Break up into many pieces and hand them to threads when they are ready.
- Dynamic scheduling.
- Increases overhead, decreases idling threads.
- Can also choose chunk size.

```
#pragma omp for schedule(dynamic)
```

or

```
!$omp do schedule(dynamic)
```

Serial	0.63s
Nthreads=8	0.10s
Speedup	6.3x
Efficiency	79%



Tuning

- schedule(static) (default) or schedule(dynamic) are good starting points.
- To get best performance in badly imbalanced problems, may have to play with chuck size; depends on your problem and on hardware.

(static,4)	(dynamic,16)
0.084s	0.099s
7/6x	6.4x
95%	79%

Two level loops

In scientific code, we usually have nested loopes were all the work is.

Almost without exception, want the loop on the outside-most loop.
Why?

```
#pragma omp for schedule(static,4)
for (int i=0;i<npix;i++)
  for (int j=0;j<npix;j++){
    double
    x=((double)i)/((double)npix);
    double
    y=((double)j)/((double)npix);
    double complex a=x+I*y;
    mymap[i][j]=how_many_iter_real(a,maxite
  }
```

Summary so far

- Start a parallel region:
  #pragma omp parallel shared() private() default()
- Parallelize a loop:
  #pragma omp for schedule(static/dynamic, chunk)
- Mark off a region only one thread can be in at a time:
  #pragma omp critical
- Safely update a single memory location:
  #pragma omp atomic
- In a parallel region, have only one process do something:
  #pragma omp single

See: http://openmp.org/wp/openmp-specifications/ for more info.
Strongly encouraged - many good sample programs.

A Few More Directives

- #pragma omp ordered - execute the loop in the order it would have run serially. Useful if you want ordered output in a parallel region. Never useful for performance.
- #pragma omp master - a block that only the master thread (thread 0) executes. Usually, #pragma omp single is better.
- #pragma omp sections - execute a list of things in parallel. In OpenMP 3, task directive (later in lecture) is more powerful
- #pragma omp for collapse(n): nested loops scheduled as one big loop.

Summary So Far II

Style Points

- If a variable is a private temporary variable inside a parallel region, try declaring it inside the region.
  Makes parallel region easier to specify, and can prevent bugs.
- OpenMP supports reduction and initialization clauses. These are never necessary to use, but are convenient and can streamline code.
- You have seen how to find out how many threads exist, etc. However, in none of our examples did we use that info.
  If you think you need to know how many threads you have, you may well be doing something wrong (with some notable exceptions such as complex reduction). Using locally declared variables, and critical regions most likely will do everything you need.

Memory Access — a seemingly unrelated intermezzo

Memory Access

- Processors work on local bits of memory in their cache.
- Cache is small and fast. Main memory is big, but slow.
- There is a large latency in getting things from main memory — often hundreds of clock cycles. The fewer times we access main memory, the faster we will go.
- Computers bring in chunks of memory at a time. If you access data in contiguous memory chunks, much of it may already be in cache. Always try to do this - serial or parallel.
- C - last index is rapidly varying. Fortran first index.

Memory Access

- Memory access is important for serial programs, but can become particularly important in OpenMP
- There is typically a limited bandwidth to main memory. If it has to be shared 2, 4, or 8 ways, it becomes especially critical to access it sensibly.
- Note on shared variables in OpenMP: If you aren't changing them, the compiler can copy the shared variable to local cache and no performance hit. Modifying shared variables is expensive - we have already seen this with the dot product.

Example - Matrix Multiplication

Example

- Linear algebra a classic example.
- Matrix multiplication: $\mathbf{C} = \mathbf{A} * \mathbf{B}$, or $\mathbf{c[i][j]} = \sum \mathbf{a[i][k]} * \mathbf{b[k][j]}$.
- Different implementations can take 10-100x longer than optimal. Slowness entirely due to memory access.
- The more you do with stuff youve pulled from main memory, the faster youll run.

Slow Matrix Multiplication

```
void matmult_slowest(double **a,
double **b, double **c, int n) {
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++) {
      c[i][j]=0;
      for (int k=0;k<n;k++)
        c[i][j]=a[i][k]*b[k][j];
    }
}
```

```
int main() {
  pca_time tt;
  int n=500;
  double **a=matrix(n,n);
  double **b=matrix(n,n);
  double **c=matrix(n,n);
  fill_random_matrix(a,n);
  tick(&tt);
  matmult_slowest(a,b,c,n);
  printf("Time to multiply %dx%d ma-
  trices with slow multiplication is
  %f\",n,n,tocksilet(&tt));
  printf("Sum of elements is
  %e\n",matrix_sum(c,n));
}
```

```
$ ./matmult_slow
Time to multiply 1000 x 1000 matrices with slow multiplication is 12.4637
Sum of elements is 2.4997e+08
$
```

Slow Matrix Multiplication

- What happened? For every element in C, we had to pull a fast direction from A, but a slow direction from B.
- Could change the order of the loops, making B fast, but then A would be slow.
- We pulled a slow vector for each element in C, for a total of n2 slow column pulls.
- Could make the transpose of B, then we would always pull from the fast columns. Only have to do n slow pulls this way.
- Drawback: must make a copy of B. If B is large, can take lots of memory.

Transpose Multiplication

```
void matmult_transport(double **x, double **b, double **c, int n) {
  double **bt=matrix(n,n);
  matrix_transpose(b,bt,n);
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++) {
      c[i][j]=0;
      for (int k=0;k<n;k++)
        c[i][j]=a[i][k]*bt[j][k];
    }
}
```

```
$ ./matmul_transporse
Time to multiply 1000 x 1000 matrices with transpose multiplication is 8.8756
Sum of elements is 2.4997e+08
$
```

About 40% faster than slow version.

Blocks

- Multiplication was still kind of slow. Why?
- For every column of C we calculate, we have to process all of B, for a total of n times. That's a lot of memory throughput.
- Recall $\mathbf{c_{ij} = a_{ik} * b_{kj}}$. Nowhere have we said that $\mathbf{c_{ij}}$, $\mathbf{a_{jk}}$, and $\mathbf{b_{kj}}$ are scalars. They could be blocks of the matrices. If we treat them as blocks, then we'll have to go to main memory less often.
- Say blocks are 20x20. Then I have to pull all of B each time I process a column of blocks. Or a total of n/20 times. Much less stress on system memory.

```
void matmult_block(double **a, double
**b, double **c, int n, int bs) {
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
      c[i][j]=0;
  int nb=n/bs;
  assert(nb*bs==n);
  double**myblock_1=matrix(bs,bs);
  double**myblock_2=matrix(bs,bs);
  double**myblock_3=matrix(bs,bs);
  for (int ib=0;ib<nb;ib++)
    for (int jb=0;jb<nb;jb++)
      for (int kb=0;kb<nb;kb++) {
        int ii=ib*bs;
        int jj=jb*bs;
        int kk=kb*bs;
        // Pull blocks from A and B out
        for (int i=0;i<bs;i++)
          for (int j=0;j<bs;j++) {
            myblock_1[i][j]=a[i+ii][j+kk];
            myblock_2[i][j]=b[j+kk][i+jj];
          }
        // Do the block multiplication
        for (int i=0;i<bs;i++)
          for (int j=0;j<bs;j++) {
            myblock_3[i][j]=0;
            for (int k=0;k<bs;k++)
              myblock_3[i][j]+=
                myblock_1[i][k]
                *myblock_2[j][k];
          }
        //Accumulate the product into c
        for (int i=0;i<bs;i++)
          for (int j=0;j<bs;j++)
            c[i+ii][j+jj]+=myblock_3[i][j];
      }
  free_matrix(myblock_1);
  free_matrix(myblock_2);
  free_matrix(myblock_3);
}
```

```
$ ./matmul_block
Time to multiply 1000 x 1000 matrices
with block multiplication is 9.5774
Sum of elements is 2.4997e+08
$
```

Blocks Debrief

- Well, managed to do better in memory, calculation time was similar (slightly larger actually).
- You may gather that writing a fast, parallel matrix multiplier isn't easy. You are right.
- People have spent a long time optimizing matrix multiplication, and gotten to 80-90% of theoretical max, using block-based algorithms (look up goto blas)
- Important corollary: Think you need to code something? Don't! See if someone else has done it. For core routines, they have, and better than you will ever do it.
- For the same problem, Goto runs in 0.1972 – 50x faster. (module load gotoblas)

Big Lesson #5

Make sure serial performance is good before worrying about parallel!

End of intermezzo on memory access

Conditional OpenMP

- There is always overhead associated with starting threads, splitting work, etc. Also, some jobs parallelize better than others.
- Sometimes, overhead takes longer than 1 thread would need to do a job - e.g. very small matrix multiplies.
- OpenMP supports conditional parallelization. Add if(condition) to parallel region beginning. So, for small tasks, overhead low, while large tasks remain parallel.

Conditional OpenMP in Action

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  #pragma omp parallel if (n>10)
  #pragma omp single
    printf("have %d threads with
    n=%d\n", omp_get_num_threads(),n);
}
```

```
$ ./conditional_if 12
have 8 threads with n=12
$ ./conditional_if 9
have 1 threads with n=9
$
```

First, pull an integer from the command line. Check to see if it's bigger than a number (in this case, 10). If so, start a parallel region. Otherwise, execute serially.

Controlling # of Threads

- Sometimes you might want more or fewer threads. May even want to change while running.
- Example - IBM P6 cluster. Matrix multiply runs fast with twice as many program threads as physical cores (hyperthreading). However, matrix factorizations run slower with more threads.
- omp_set_num_threads(int) sets or changes the number of threads during runtime.

omp_set_num_threads() in action

```
#include "stdio.h"
#include "omp.h"
int main(int argc,char *argv[]){
  //find # of physical cores
  //this is an openmp library routine.
  int max_threads=omp_get_num_procs();
  int n=atoi(argv[1]);
  //set # threads equal to input
  //assuming it's less than max_threads
  if (n<max_threads)
    omp_set_num_threads(n);
  else
    omp_set_num_threads(max_threads);
  #pragma omp parallel
  #pragma omp single
  printf("Running with %d threads for
  n=%d.\n", omp_get_num_threads(),n)
}
```

We have changed the # of threads during the program. We could always change the number later on in the same code, if we so desired. Note the use of omp_get_num_procs(), a library call to detect the physical number of available processors.

Tasks

- OpenMP 3.0 supports the #pragma omp task directive.
- A task is a job assigned to a thread. Powerful way of parallelizing non-loop problems.
- Tasks should help omp/mpi hybrid codes - one task can do communications, rest of threads keep working.
- Like all omp, tasks must be called from parallel region.
- Raises complication of nested parallelism (what happens if a parallel loop called from parallel loop?).

Tasks: test_task.c

```
#include <stdio.h>
#include <omp.h>
int main(){
  #pragma omp parallel
  #pragma omp single
  {
    printf("hello");
    #pragma omp task
    {
      printf("hello 1 from
      %d.",omp_get_thread_num());
    }
    #pragma omp task
    printf("hello 2 from
    %d.",omp_get_thread_num());
  }
}
```

Often want to start tasks from as if from serial region. Must be in parallel for tasks to spawn, so #pragma omp parallel followed by #pragma omp single very useful. What would happen w/out #pragma omp single?

Beauty of Tasks

- Some problems naturally fit into tasks that are otherwise hard to parallelize.
- Example (from standard): parallel tree processing.
- Each node has left, right pointers, process each sub- pointer with a task.
- Look how short the parallel tree is!

How would you do this problem without tasks?

```
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node*);
void traverse(struct node* p) {
  if (p->left)
    #pragma omp task
    traverse(p->left);
  if (p->right)
    #pragma omp task
    traverse(p->right);
  process(p);
}
```

HANDS-ON 3:
Parallelize Matrix Multiplications.