High-Performance Scientific Computing: Parallel Computing Paradigms

Erik Spence

SciNet HPC Consortium

13 March 2014

Today's class

In today's class we will cover:

- The different types of HPC hardware that you might encounter, and their advantages and disadvantages.
- Parallel programming approaches, and how they relate to hardware.
- A mini-introduction to using SciNet's GPC.
- Assignment 9.

Supercomputer architectures

Supercomputer architectures comes in a number of different types:

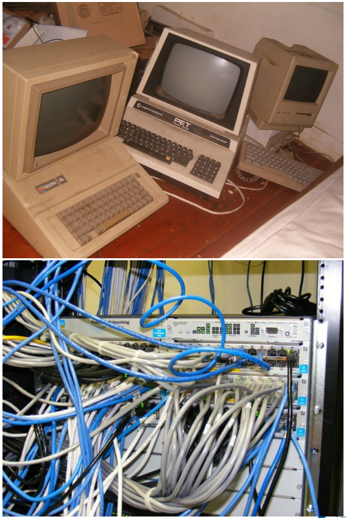
- Clusters, or distributed memory machines, are in essence a bunch of desktops linked together by a network ("interconnect"). Easy and cheap.
- Multi-core machines, or shared-memory machines, are a collection of processors that can see and use the same memory. Limited number of cores, typically, and much more $$$ when the machine is large. Your desktop is such a machine.
- Vector machines were the early supercomputers, and could do the same operation on a large number of numbers at the same time. Very $$$$$$, especially at scale. These days most chips have some low-level vectorization, but you rarely need to worry about it.

Most supercomputers are a hybrid combo of these different architectures.

Distributed Memory: Clusters

Clusters are the simplest type of parallel computer to build:

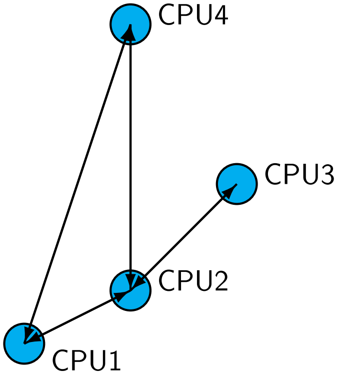
- Take existing powerful standalone computers,
- and network them.



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

Distributed Memory: Clusters

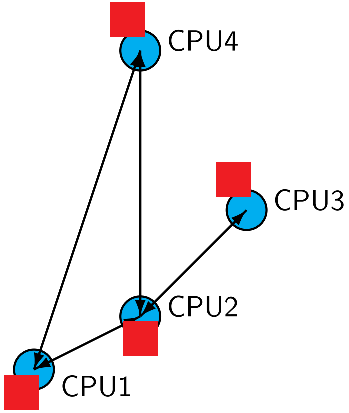
- Each Processor is independent! Parallel code consists of programs running on separate processors, communicating with each other when necessary; could be entirely different programs.



Distributed Memory: Clusters

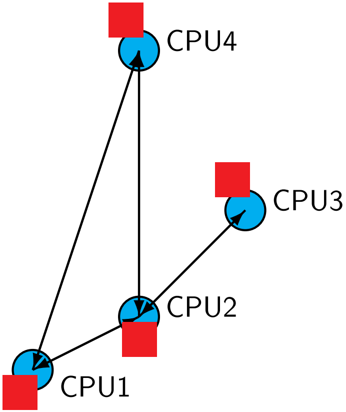
- Each Processor is independent! Parallel code consists of programs running on separate processors, communicating with each other when necessary; could be entirely different programs.
- Each processor has its own memory! Whenever it needs data from another processor, that processor needs to send it.

  The usual model is 'message passing'.



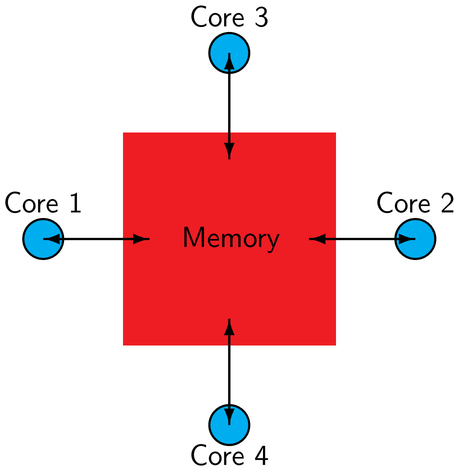
Distributed Memory: Clusters

- **Hardware:**
  Easy to build (harder to build well), easy to expand. One can build larger and larger clusters relatively easily.
- **Software:**
  Every communication needs to be hand-coded: harder to program.



Shared Memory

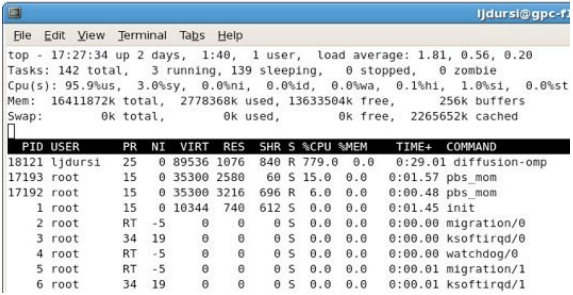
- Different processors acting on one large bank of memory. All processors 'see' the same data.
- All coordination is done through memory.
- Could use message passing, but there is no need.
- Each core is assigned a thread of execution of a single program that acts on the data.
- Your workstation uses this architecture, if it's multi-core.
- Can also use hyper-threading: assigning more than one thread to a given core.



Threads versus Processes

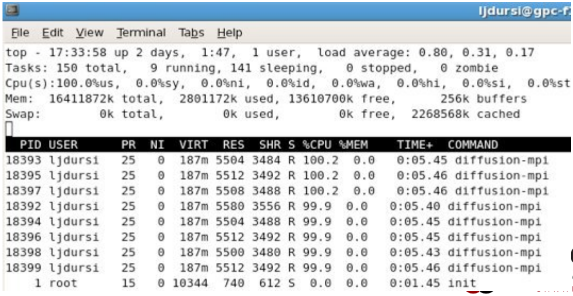
Threads: Threads of execution within one process, with access to the same memory etc.

Processes: Independent tasks with their own memory and resources



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

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



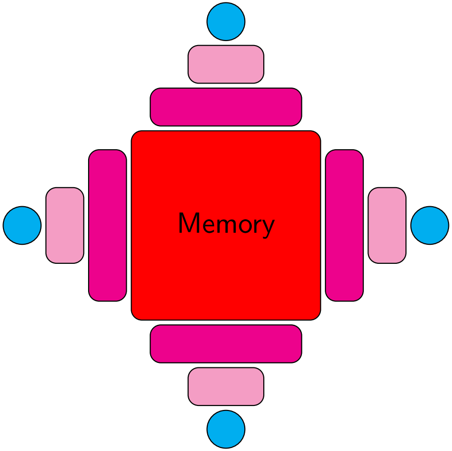
```
top - 17:33:58 up 2 days,  1:47,  1 user,  load average: 0.80, 0.31, 0.17
Tasks: 150 total,   9 running, 141 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  16411872k total,  2801172k used, 13610700k free,      256k buffers
Swap:        0k total,        0k used,        0k free,  2268568k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
18393 ljdursi   25   0  187m 5504 3484 R 100.2  0.0   0:05.45 diffusion-mpi
18395 ljdursi   25   0  187m 5512 3492 R 100.2  0.0   0:05.46 diffusion-mpi
18397 ljdursi   25   0  187m 5508 3488 R 100.2  0.0   0:05.46 diffusion-mpi
18392 ljdursi   25   0  187m 5580 3556 R 99.9  0.0   0:05.40 diffusion-mpi
18394 ljdursi   25   0  187m 5504 3488 R 99.9  0.0   0:05.45 diffusion-mpi
18396 ljdursi   25   0  187m 5512 3492 R 99.9  0.0   0:05.45 diffusion-mpi
18398 ljdursi   25   0  187m 5500 3480 R 99.9  0.0   0:05.43 diffusion-mpi
18399 ljdursi   25   0  187m 5512 3492 R 99.9  0.0   0:05.46 diffusion-mpi
    1 root      15   0 10344  740  612 S  0.0  0.0   0:01.45 init
```

Shared memory: NUMA

Non-uniform memory access:

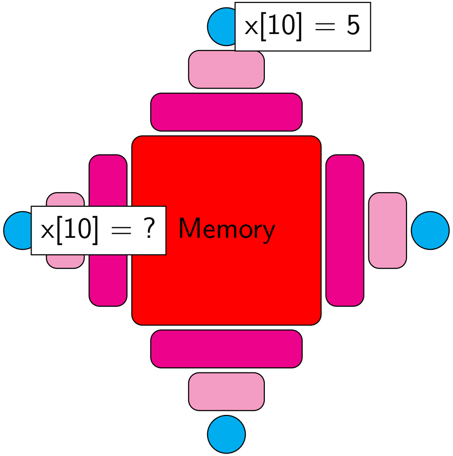
- Each core typically has some memory of its own.
- Cores have cache too.
- Keeping this memory coherent is extremely challenging.



NUMA coherency

Non-uniform memory access:

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



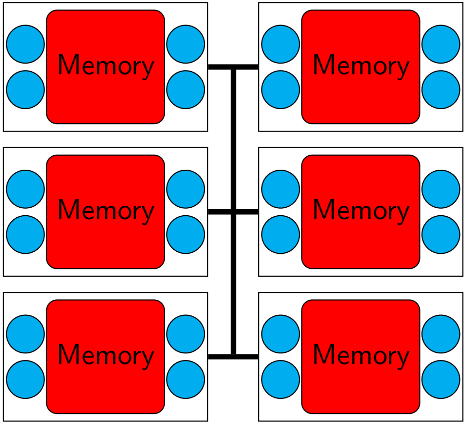
Share memory communication cost

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

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

Hybrid architectures

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



Choosing your programming approach

The programming approach you use depends on the type of problem you have, and the type of machine that you will be using:

- Embarrassingly parallel applications: scripting, GNU Parallel[1].
- Shared memory machine: OpenMP, threads, Automated parallelization.
- Distributed memory machine: MPI, Files.
- Graphics computing: CUDA, OpenCL
- Hybrid combinations.

We will discuss OpenMP, MPI and hybrid coding in this course.

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

Data or computation bound?

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

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

Granularity

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

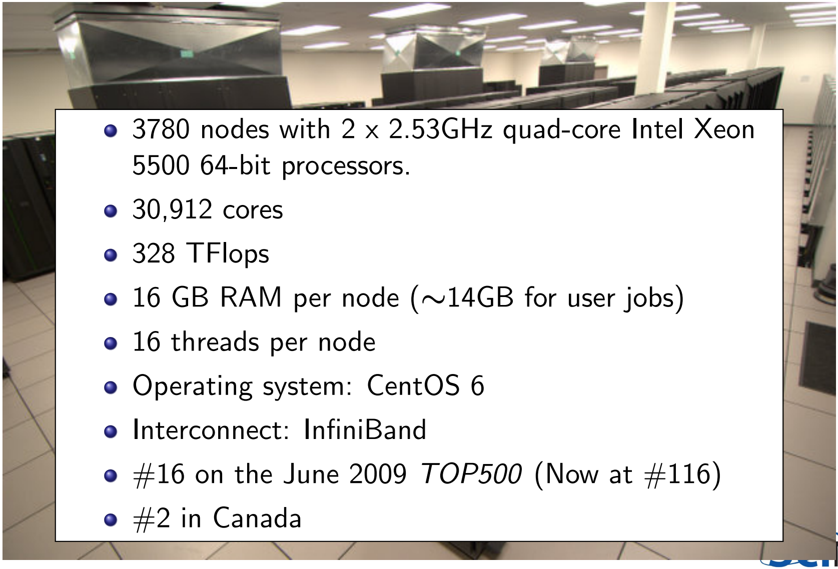
- Fine-grained parallelism
  - Small individual tasks.
  - The data is transferred among processors frequently.
  - OpenMP is often used, to overlap different hardware functions.
- Coarse-grained parallelism
  - Data communicated infrequently, after large amounts of computation.
  - MPI is often used, because of network latency.

Too fine-grained $\longrightarrow$ overhead issues.
Too coarse-grained $\longrightarrow$ load imbalance issues.

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

The General Purpose Cluster

The General Purpose Cluster



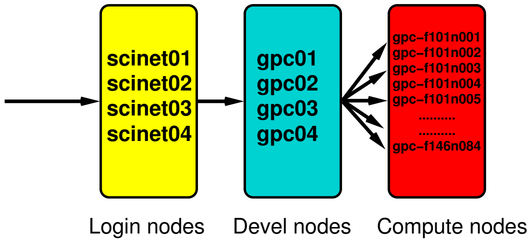
- 3780 nodes with 2 x 2.53GHz quad-core Intel Xeon 5500 64-bit processors.
- 30,912 cores
- 328 TFlops
- 16 GB RAM per node (∼14GB for user jobs)
- 16 threads per node
- Operating system: CentOS 6
- Interconnect: InfiniBand
- #16 on the June 2009 *TOP500* (Now at #116)
- #2 in Canada

Mini-intro to SciNet - getting started

SciNet accounts:

- Need to have an account to access SciNet.
- If you don't have an account, get one: (wiki.scinethpc.ca/wiki/index.php/Essentials)
- If you can't, for whatever reason, email us: (support@scinet.utoronto.ca).
- Before you do anything else, read the SciNet Tutorial and the GPC quick start on the wiki: (wiki.scinethpc.ca/wiki/index.php/GPC_Quickstart).

Mini-intro to SciNet - accessing SciNet



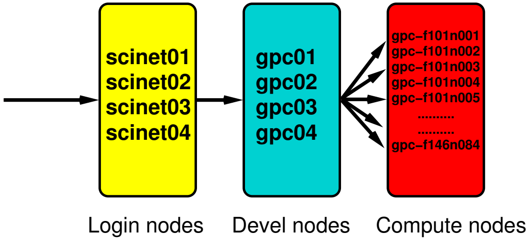
First open a terminal and ssh into a login node (not part of clusters):

```
ejspence@mycomp ~>
ejspence@mycomp ~> ssh -X -l ejspence login.scinet.utoronto.ca
ejspence@scinet01 ~>
```

The login nodes are gateways:

- only to be used for small data transfer.
- and to proceed logging into one of the devel nodes.

Mini-intro to SciNet - working on SciNet



Once logged into a login node, ssh into a devel node. On GPC that means: `gpc01` - `gpc04`. These are aliases for longer node names.

```
ejspence@scinet01 ~>
ejspence@scinet01 ~> ssh -X gpc03
ejspence@gpc-f103n084 ~>
```

`gpc03` is an alias for the node named `gpc-f103n084`. All work (editting, compiling, job submission, *etc.*) is done from the devel nodes.

Mini-intro to SciNet - compiling

- Once you have logged into a devel node you need to load the compiler.
- Other than essentials, all software is loaded using the `module` command.

```
ejspence@gpc-f103n084 ~>
ejspence@gpc-f103n084 ~> module load gcc
ejspence@gpc-f103n084 ~>
```

- You can now compile using g++.
- In general, the Intel compilers are preferred on the GPC, but for the purpose of this course g++ is fine too.

Mini-intro to SciNet - how to run

- You do not run on login nodes, or devel nodes.
- You run on compute nodes.
- Compute nodes are reserved for your use through a queuing system.
- You can get an interactive session on a compute node by making the following request to the queuing system.

```
ejspence@gpc-f103n084 ~>
ejspence@gpc-f103n084 ~> qsub -I -X -qdebug -l nodes=1:ppn=8,walltime=2:00:00
```

- This gives you a dedicated compute node, in the 'debug' queue, for two hours. There is a two hour limit when you use this queue.
- Alternatively, submit a job script.

Mini-intro to SciNet - how to run, continued

- You do not run on login nodes, or devel nodes.
- You run on compute nodes.
- Compute nodes are reserved for your use through a queuing system.
- Compute nodes do not have write access to the /home. All write access is on the /scratch file system.
- You can get to your scratch directory thus:

```
ejspence@gpc-f103n084 ~> pwd
/home/s/scinet/ejspence
ejspence@gpc-f103n084 ~> cd $SCRATCH
ejspence@gpc-f103n084 .../ejspence> pwd
/scratch/s/scinet/ejspence
ejspence@gpc-f103n084 .../ejspence>
```

GNU Parallel

- What if you need to keep all 8 cores on a node busy, but you have a serial code?
- GNU Parallel can help you with that!
- GNU parallel is a really nice tool to run multiple serial jobs in parallel. It allows you to keep the processors on each 8-core node busy, if you provide enough jobs to do.
- GNU parallel is accessible on the GPC in the module gnu-parallel.

```
ejspence@gpc-f103n084 ~> module load gnu-parallel/20130422
```

Note that we recommend the newer version of gnu-parallel over the (default) 2010 one.

GNU Parallel Example

```
ejspence@gpc-f103n084 ~> g++ -O3 mycode.cc -o mycode
```

```
ejspence@gpc-f103n084 ~> cp mycode $SCRATCH/example
```

```
ejspence@gpc-f103n084 ~> cd $SCRATCH/example
```

```
ejspence@gpc-f103n084 .../example> cat joblist.txt
mkdir run1; cd run1; ../mycode 1 > out
mkdir run2; cd run2; ../mycode 2 > out
...
```

```
ejspence@gpc-f103n084 .../example> cat myjob.pbs
#!/bin/bash
#PBS -l nodes=1:ppn=8,walltime=24:00:00
#PBS -N GPJob
cd $PBS_O_WORKDIR
module load gcc gnu-parallel/20130422
parallel -j 8 < joblist.txt
```

```
ejspence@gpc-f103n084 .../example> qsub myjob.pbs
2961985.gpc-sched
```

```
ejspence@gpc-f103n084 .../example> ls
 GPJob.e2961985   GPJob.o2961985   joblist.txt
 mycode           myjob.pbs        run1/
 ...
```

Assignment 9

Please perform the following steps:

1. Make sure you've got a SciNet account!
2. Read the SciNet tutorial, at least the part about using the GPC (support.scinet.utoronto.ca/wiki/images/5/54/SciNet_Tutorial.pdf).
3. Read the GPC Quick Start (support.scinet.utoronto.ca/wiki/index.php/GPC_Quickstart).

Now that you're ready, go get the code for the assignment:

```
ejspence@gpc-f103n084 ~> cd $SCRATCH
ejspence@gpc-f103n084 .../ejspence> git clone \
  /scinet/course/sc3/homework1
Initialized empty Git repository in /scratch/s/scinet/ejspence/homework1/.git/
ejspence@gpc-f103n084 .../ejspence> cd homework1
ejspence@gpc-f103n084 .../homework1> source setup
ejspence@gpc-f103n084 .../homework1> make
```

Assignment 9, continued

The directory homework1 contains a threaded program called 'blurppm' and 266 ppm images to be blurred. It is used thus:

```
blurppm INPUTPPM OUTPUTPPM BLURRADIUS NUMBEROFTHREADS
```

Before you start the rest of the assignment, perform a simple test:

```
ejspence@gpc-f103n084 ~> qsub -I -X -qdebug -l nodes=1:ppn=8,walltime=2:00:00
ejspence@gpc-f109n001 ~> cd $SCRATCH/homework1
ejspence@gpc-f109n001 .../homework1> ./blurppm 001.ppm new001.ppm 30 1
ejspence@gpc-f109n001 .../homework1> display 001.ppm &
ejspence@gpc-f109n001 .../homework1> display new001.ppm &
```

Note that you need an X server running on your local machine to see the results of the 'display' command. Linux and Mac machines should be fine. If you're running Windows, talk to us if you don't know what to do.

Assignment 9, continued

The purpose of this assignment is to do timing tests, and explore how judicious use of threads can improve throughput.

```
ejspence@gpc-f109n001 .../homework1> time ./blurppm 001.ppm new001.ppm 30 1
real 0m52.900s
user 0m52.881s
sys 0m0.008s
ejspence@gpc-f109n001 .../homework1>
```

4. Part 1:
   - Time blurppm with BLURRADIUS ranging from 1 to 41 in steps of 8, and NUMBEROFTHREADS ranging from 1 to 8. Record the (real) duration of each run.
   - Make plots of the duration and speed-up as a function of NUMBEROFTHREADS, for each value of BLURRADIUS.
   - Submit your script and plots.

Assignment 9, continued

5. Part 2:
   - Use GNU parallel to run blurppm on all 266 images with a radius of 41.
   - Investigate different scenarios:
     1. Have GNU parallel run 8 at a time with 1 thread.
     2. Have GNU parallel run 4 at a time with 2 threads.
     3. Have GNU parallel run 2 at a time with 4 threads.
     4. Have GNU parallel run 1 at a time with 8 threads.

     Record the total time it takes to run each of these scenarios. Comment on the results.
   - Repeat this with a BLURRADIUS of 3.
   - Submit scripts, timing data, and comments on results.