Obtaining hardware information and monitoring performance

August 3 – August 7, 2020
SDSC Summer Institute
Robert Sinkovits

Introduction

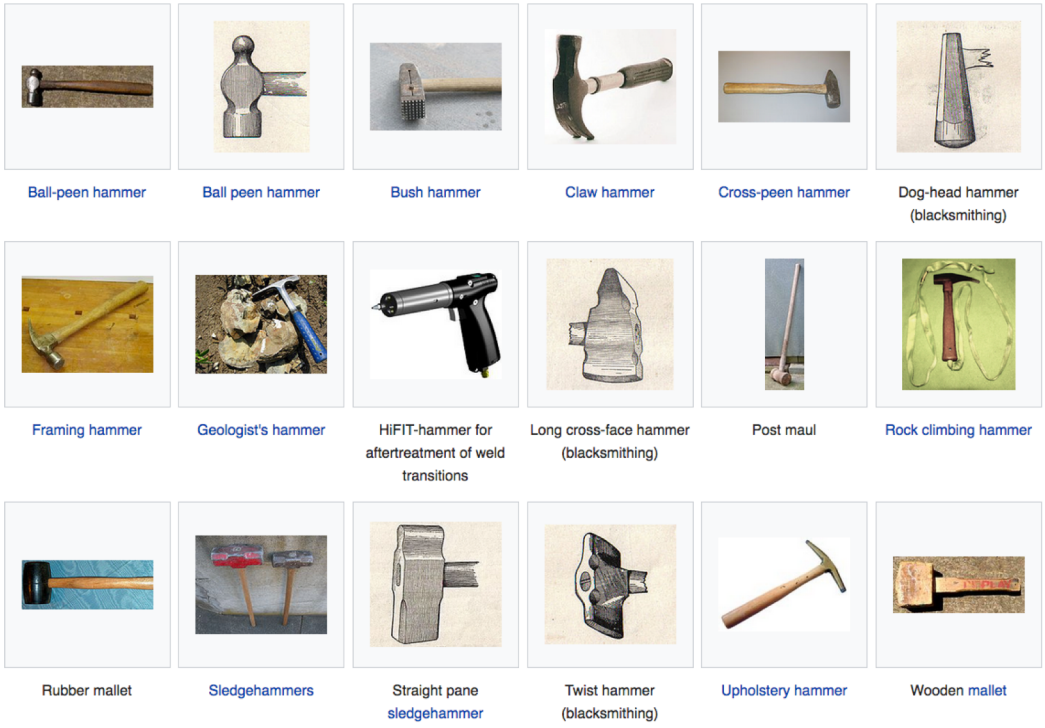
- Most of you are here because you are computational scientists
  - Have a specific scientific problem you're trying to solve
  - Support researchers from a variety of domains
- The actual hardware is probably of secondary interest
  - Hardware is interesting but not *that* interesting
- Nonetheless, it's still helpful to know a bit about hardware
- In this talk we'll learn how to
  - Get information about your system
  - Use some common usage monitoring tools

Getting hardware information – why do I care?

- You may be asked to report details of your hardware in a manuscript, presentation, proposal or request for computer time
- You'll know what you're running on and can answer questions like
  - Is the login node the same as the compute nodes?
  - How does one machine compare to another?
- It will give you a way of estimating performance or at least bounds on performance relative to another system. *All else being equal*, jobs will run at least as fast on hardware with
  - Faster CPU clock speeds
  - Larger caches
  - Faster local drives

Computers are like hammers!

Just like hammers, there is a wide variety of computer hardware. You can probably get away with using the wrong one, but your performance may be suboptimal and you might end up using a bigger tool than you need.



Wikipedia

Processor specifications (using /proc/cpuinfo)

- On Linux systems, the /proc/cpuinfo pseudo-file lists key processor information. Much of this is relatively cryptic, but there are some valuable pieces of data
  - Number of processors (sockets)
  - Processor type or model
  - Nominal clock speed
  - Number of cores per processor
  - Cache size (limited information)
  - Instruction set architecture

Processor specifications (using /proc/cpuinfo)

Top lines of /proc/cpuinfo from Comet login node, with my annotation in red.

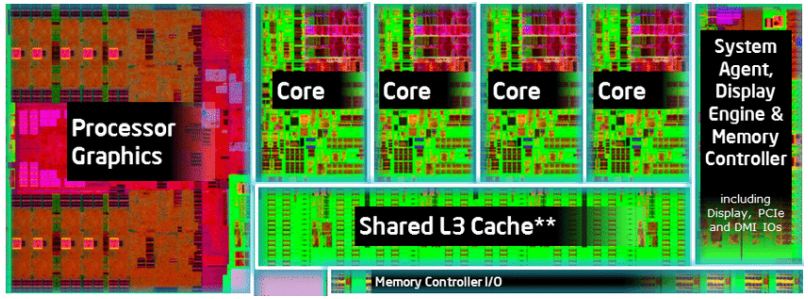
Nomenclature can be a little confusing and we'll address that in the next slide.

```
processor       : 0 (processor number actually means core number)
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz (processor type)
stepping        : 2
Microcode       : 50
cpu MHz         : 2501.000 (nominal clock speed)
cache size      : 30720KB (L3 cache)
physical id     : 0 (physical id actually means processor number)
siblings        : 12
core id         : 0
cpu cores       : 12 (number of cores in processor)
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de … avx … avx2 (AVX/AVX2 capable processor)
```

A brief aside on nomenclature

We normally think of the multicore unit that plugs into the motherboard as the "processor"

The /proc/cpuinfo file uses processor in a different way to mean compute core, counted across all the cores available in a node.




```
processor       : 0
physical id     : 0
cpu cores       : 12
```

Processor/socket and core counts

If you want to find out how many processors or cores you have on the compute node, grep for ‘physical id’ or ‘processor’

```
$ grep processor /proc/cpuinfo
processor : 0
processor : 1
processor : 2
processor : 3
processor : 4
processor : 5
processor : 6
processor : 7
processor : 8
processor : 9
processor : 10
processor : 11
processor : 12
processor : 13
processor : 14
processor : 15
processor : 16
processor : 17
processor : 18
processor : 19
processor : 20
processor : 21
processor : 22
processor : 23
```

```
$ grep 'physical id' /proc/cpuinfo
physical id : 0
physical id : 0
physical id : 0
physical id : 0
physical id : 0
physical id : 0
physical id : 0
physical id : 0
physical id : 0
physical id : 0
physical id : 0
physical id : 0
physical id : 1
physical id : 1
physical id : 1
physical id : 1
physical id : 1
physical id : 1
physical id : 1
physical id : 1
physical id : 1
physical id : 1
physical id : 1
physical id : 1
```

cores 0-11 are on phys. id 0 and 12-23 are on phys. id 1

*Results shown are for **Comet standard compute node***

Processor/socket and core counts

If you want to find out how many processors or cores you have on the compute node, grep for ‘physical id’ or ‘processor’

```
$ grep processor /proc/cpuinfo
processor : 0
processor : 1
processor : 2
processor : 3
processor : 4
processor : 5
processor : 6
processor : 7
processor : 8
processor : 9
processor : 10
processor : 11
processor : 12
processor : 13
processor : 14
processor : 15
processor : 16
processor : 17
processor : 18
processor : 19
processor : 20
processor : 21
processor : 22
processor : 23
```

```
$ grep 'physical id' /proc/cpuinfo
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
physical id : 0
physical id : 1
```

cores are assigned round-robin to phys. ids

*Results shown are for **Comet GPU node***

Processor/socket and core counts

If you want to find out how many processors or cores you have on the compute node, grep for ‘physical id’ or ‘processor’

So why does it matter how the cores are numbered and assigned to sockets (sequentially vs. round-robin)?

Has implications for running hybrid (MPI + OpenMP) codes and the correct mapping of thread to cores. Fortunately, we have mechanisms in place to account for this.

```
$ grep processor /proc/cpuinfo
processor : 0
processor : 1
processor : 2
processor : 3
processor : 4
processor : 5
processor : 6
processor : 7

...

Processor : 60
processor : 61
processor : 62
processor : 63
```

```
$ grep 'physical id' /proc/cpuinfo
physical id : 0
physical id : 1
physical id : 2
physical id : 3
physical id : 0
physical id : 1
physical id : 2
physical id : 3

...

physical id : 0
physical id : 1
physical id : 2
physical id : 3
```

cores are assigned round-robin to phys. ids

Results shown are for ***Comet large memory node***

Quick aside on hyperthreading

Intel® Hyper-Threading Technology (Intel® HT Technology) uses processor resources more efficiently, enabling multiple threads to run on each core. As a performance feature, it also increases processor throughput, improving overall performance on threaded software.

https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

Hyper-Threading is Intel's term for **simultaneous multithreading (SMT).** This is a process where a CPU splits each of its physical **cores** into virtual cores, which are known as threads. For example, most of Intel's CPUs with two cores use hyper-threading to provide four threads, and Intel CPUs with four cores use hyper-threading to provide eight threads.

https://www.tomshardware.com/reviews/hyper-threading-intel-definition,5746.html

SDSC does not enable hyperthreading on its systems. When hyperthreading is enabled, core count will appear to be doubled

A brief aside on pseudo-files

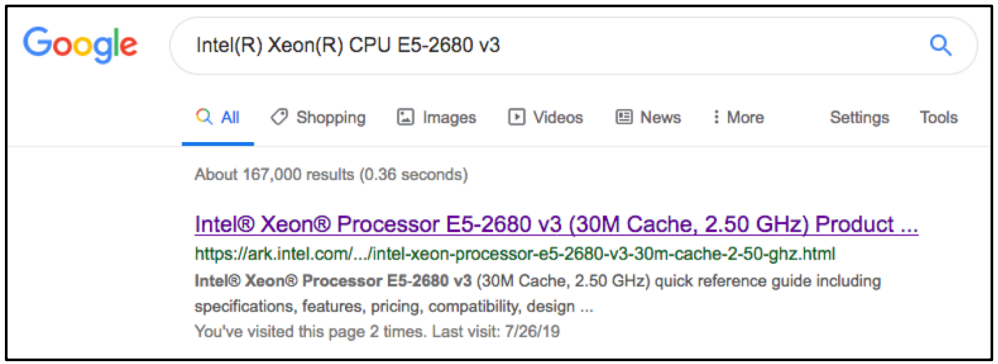
Up to this point, we've been using the term pseudo-file without defining what it is. Recall that in the UNIX/Linux world, everything is treated as a file (files, directories, devices, etc.)

/proc and /sys are just interfaces to the Linux kernel data structures in a convenient and familiar file system format

```
$ ls -ld /proc
dr-xr-xr-x 2307 root root 0 Mar 13 10:45 /proc
$ ls -ld /proc/cpuinfo
-r--r--r-- 1 root root 0 Jul 26 13:49 /proc/cpuinfo

$ head /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family          : 6
model               : 45
model name          : Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
stepping  : 6
cpu MHz             : 2593.861
```

What's in a name?

We usually think of processors in terms of their codenames, such as Sandy Bridge, Haswell or Skylake. Unfortunately, the /proc/cpuinto pseudo-file returns something a little more opaque such as "Intel(R) Xeon(R) CPU E5-2670 v3". A quick Google search helps

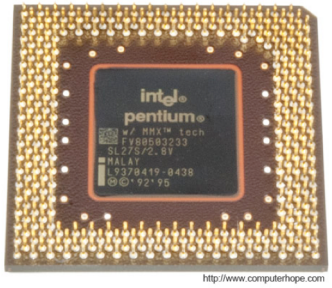


Advanced Vector Extensions (AVX, AVX2, AVX512)

The Advanced Vector Extensions (AVX) are an extension to the x86 microprocessor architecture that allows a compute core to perform up to 8 floating point operations per cycle. Previous limit was 4/core/cycle

- AVX2 improves this to 16 Flops/cycle/core (Comet)
- AVX512 further improves to 32 Flops/cycle/core (Expanse – Oct 2020)

These were developed partially in response to challenges in increasing CPU clock speeds



March 6, 2000 8:00 AM PST

AMD makes move to 1-GHz chip

By Joe Wilcox and Michael Kanellos
Staff Writers, CNET News

Advanced Vector Extensions (AVX, AVX2, AVX512)

- Can theoretically obtain a 2x speedup when going from a non-AVX processor to an AVX capable processor (all else being equal)
  - And another 2x from AVX to AVX2
  - And another 2x from AVX2 to AVX512

- But don't get too excited. It's difficult enough to make good use of AVX and even harder to make good use of AVX2 or AVX512. Need long loops with vectorizable content. Memory bandwidth not keeping up with gains in computing power.

Getting memory information (/proc/meminfo)

On Linux machines, the /proc/meminfo pseudo-file lists key memory specs. More information than you probably want, but at least one bit of useful data

```
MemTotal:       66055696 kB   (total physical memory)
MemFree:         3843116 kB
Buffers:            6856 kB
Cached:         31870056 kB
SwapCached:         1220 kB
Active:          7833904 kB
Inactive:       25583720 kB
Active(anon):     593252 kB
Inactive(anon):   949000 kB
Active(file):    7240652 kB   (pretty good approximation to used memory)
Inactive(file): 24634720 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       2097144 kB
SwapFree:         902104 kB
Dirty:             17772 kB
Writeback:            32 kB
AnonPages:       1540768 kB
...
```

*Results shown are for Gordon **standard node***

For more details, see http://www.redhat.com/advice/tips/meminfo.html

Getting memory information (/proc/meminfo)

On Linux machines, the /proc/meminfo pseudo-file lists key memory specs. More information than you probably want, but at least one bit of useful data

```
MemTotal:       1588229376 kB  (total physical memory)
MemFree:        1575209968 kB
Buffers:            281396   kB
Cached:            1334596   kB
SwapCached:              0   kB
Active:             648320   kB
Inactive:          1027324   kB
Active(anon):        59872   kB  (pretty good approximation to used memory)
Inactive(anon):          4   kB
Active(file):       588448   kB
Inactive(file):    1027320   kB
Unevictable:             0   kB
Mlocked:                 0   kB
SwapTotal:               0   kB
SwapFree:                0   kB
Dirty:                  56   kB
Writeback:               0   kB
AnonPages:           61744   kB
...
```

*Results shown are for Comet **large memory node***

For more details, see http://www.redhat.com/advice/tips/meminfo.html

Getting memory information (/proc/meminfo)

Using a simple script, you can monitor total memory usage for all processes as a function of time. Note that there is a lot of discussion on how to precisely measure memory (http://stackoverflow.com/search?q=measuring+memory+usage). The following should be good enough if you're on a dedicated node.

```
#!/usr/bin/perl
use strict;
use warnings;
my $count = 0;
print (" time(s)    Memory (GB)\n");
while(1) {
    sleep(1);
    $count++;
    open(MI, "/proc/meminfo");
    while(<MI>) {
        if (/Active:/) {
            my (undef, $active, undef) = split();
            $active = $active /         1048576.0;
            printf("%6d       %f\n", $count, $active);
        }
    }
    close(MI);
}
```

More memory information - dmidecode

If you really need to dig deeper and get more details on memory configuration, you can run the dmidecode command. You'll need root privileges to do this.

Output shows the results for one DIMM slot.

dmidecode --type memory

```
Memory Device
        Array Handle: 0x001D
        Error Information Handle: No Error
        Total Width: 72 bits
        Data Width: 64 bits
        Size: 16384 MB
        Form Factor: DIMM
        Set: None
        Locator: DIMM_A1
        Bank Locator: CPU1
        Type: DDR4
        Type Detail: Synchronous Registered (Buffered)
        Speed: 2133 MHz
        Manufacturer: 0xCE00
        Serial Number: 0x394FECDD
        Asset Tag: Unknown
        Part Number: M393A2G40DB0-CPB
        Rank: 1
        Configured Clock Speed: 2133 MHz
        Minimum Voltage:  1.2 V
        Maximum Voltage:  1.2 V
        Configured Voltage:  1.2 V
```

Getting GPU information

If you're using GPU nodes, you can use nvidia-smi (NVIDIA System Management Interface program) to get GPU information (type, count, etc.)

Tesla K80

4 GPUs

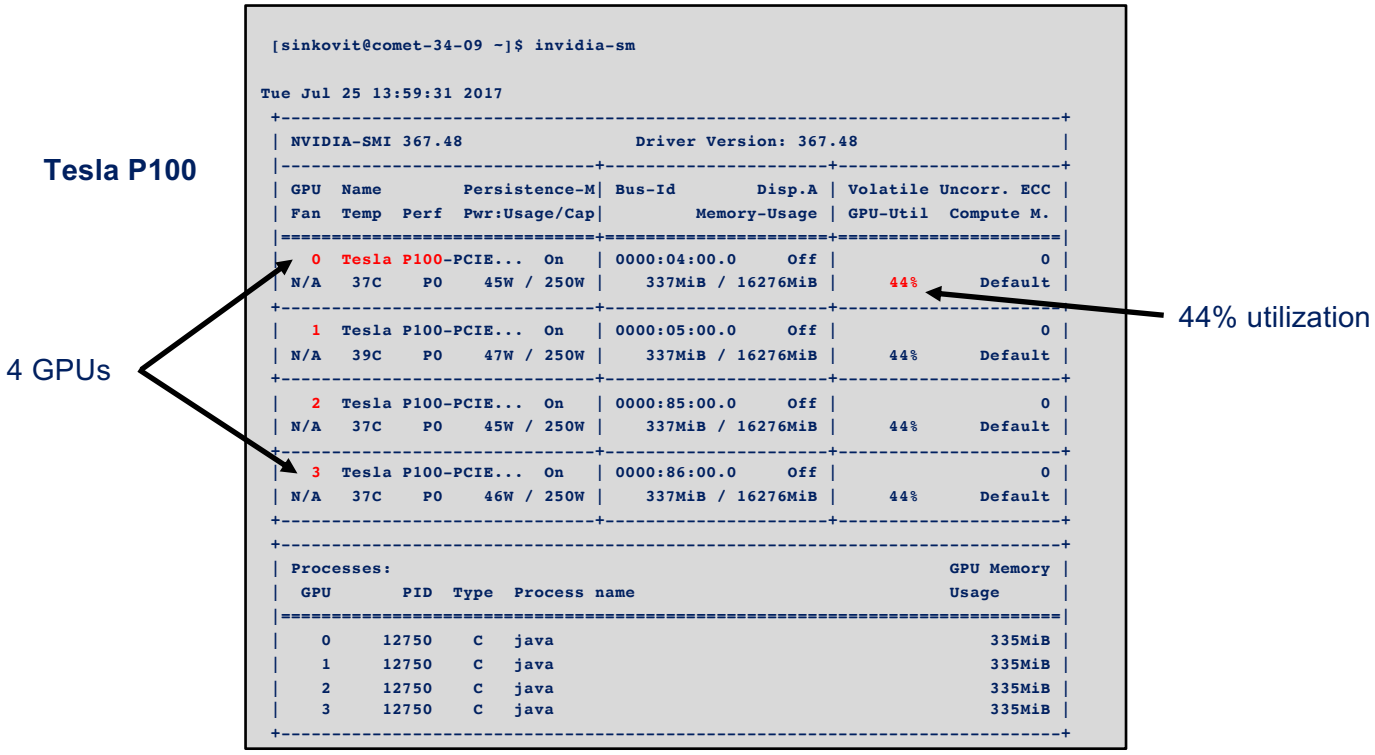
```
[sinkovit@comet-30-04 ~]$ nvidia-smiTue Jul 25 14:10:19 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 367.48                 Driver Version: 367.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           On   | 0000:05:00.0     Off |                  Off |
| N/A   49C    P0    60W / 149W |    905MiB / 12205MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K80           On   | 0000:06:00.0     Off |                  Off |
| N/A   54C    P0   148W / 149W |   1926MiB / 12205MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K80           On   | 0000:85:00.0     Off |                  Off |
| N/A   63C    P0   147W / 149W |   1246MiB / 12205MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla K80           On   | 0000:86:00.0     Off |                  Off |
| N/A   27C    P8    30W / 149W |      0MiB / 12205MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0    192045    C   python                                       905MiB |
|    1     44610    C   /home/sallec/kronos_sarah/kronos_gpu_dp     1924MiB |
|    2    148310    C   /opt/amber/bin/pmemd.cuda                   1244MiB |
+-----------------------------------------------------------------------------+
```

0% utilization

100% utilization

Getting GPU information

If you're using GPU nodes, you can use nvidia-smi (NVIDIA System Management Interface program) to get GPU information (type, count, etc.)



Finding cache information

On Linux systems, can obtain cache properties through the /sys pseudo filesystem. Details may vary slightly by O/S version and vendor, but basic information should be consistent

```
$ pwd
/sys/devices/system/cpu

$ ls
cpu0   cpu12  cpu2  cpu6  cpufreq     online       probe
cpu1   cpu13  cpu3  cpu7  cpuidle     perf_events  release
cpu10  cpu14  cpu4  cpu8  kernel_max  possible     sched_mc_power_savings
cpu11  cpu15  cpu5  cpu9  offline     present      sched_smt_power_savings

$ cd cpu0/cache
$ ls
index0  index1  index2  index3

$ cd index0
$ ls
coherency_line_size  physical_line_partition  size
level                shared_cpu_list          type
number_of_sets       shared_cpu_map           ways_of_associativity
```

Comet Cache properties – Intel Haswell (Intel Xeon E5-2680)

level	type	line size	sets	associativity	size (KB)
L1	data	64	64	8	32
L1	instruction	64	64	8	32
L2	unified	64	512	8	256
L3	unified	64	24576	20	30720

High end processor used in many Top500 supercomputers, including SDSC's Comet system and TACC's Stampede1

L1 and L2 caches are per core

L3 cache shared between all 12 cores in socket

sanity check: line size x sets x associativity = size

L2 cache size = 64 x 512 x 8 = 262144 = 256 K

Gordon Cache properties – Intel Sandy Bridge (Intel Xeon E5-2670)

level	type	line size	sets	associativity	size (KB)
L1	data	64	64	8	32
L1	instruction	64	64	8	32
L2	unified	64	512	8	256
L3	unified	64	16384	20	20480

High end processor used in many Top500 supercomputers, including SDSC's Gordon system

L1 and L2 caches are per core

L3 cache shared between all 8 cores in socket

sanity check: line size x sets x associativity = size

L2 cache size = 64 x 512 x 8 = 262144 = 256 K

Trestles Cache properties – AMD Magny-Cours (AMD Opteron Processor 6136)

level	type	line size	sets	associativity	size (KB)
L1	data	64	512	2	64
L1	instruction	64	512	2	64
L2	unified	64	512	16	512
L3	unified	64	1706	48	5118

Previous generation AMD enterprise level processor, used in SDSC's Trestles system (retired and serving new life at U. Arkansas)

L1 and L2 caches are per core

L3 cache shared between all 8 cores in socket

sanity check: line size x sets x associativity = size

L2 cache size = 64 x 512 x 16 = 524288= 512K

Impact of cache size on performance

Based on the clock speed and instruction set, program run on single core of Gordon should be 2.26x faster than on Trestles. The larger L1 and L2 cache sizes on Trestles mitigate performance impact for very small problems.

DGSEV (Ax=b) wall times as function of problem size

N	t (Trestles)	t (Gordon)	ratio	KB
62	0.000117	0.000086	1.36	30
125	0.000531	0.000384	1.38	122
250	0.002781	0.001542	1.80	488
500	0.016313	0.007258	2.24	1953
1000	0.107222	0.046252	2.31	7812
2000	0.744837	0.331818	2.24	31250
4000	5.489990	2.464218	2.23	125000

Finding SCSI device information

SCSI (Small Computer System Interface) is a common interface for mounting peripheral, such as hard drives and SSDs. The /proc/scsi/scsi file will provide info on SCSI devices

```
[sinkovit@comet-13-65 ~]$ cat /proc/scsi/scsi
Attached devices:
Host: scsi4 Channel: 00 Id: 00 Lun: 00
  Vendor: ATA      Model: INTEL SSDSC2BB16 Rev: D201
  Type:   Direct-Access                    ANSI  SCSI revision: 05
Host: scsi5 Channel: 00 Id: 00 Lun: 00
  Vendor: ATA      Model: INTEL SSDSC2BB16 Rev: D201
  Type:   Direct-Access                    ANSI  SCSI revision: 05
```

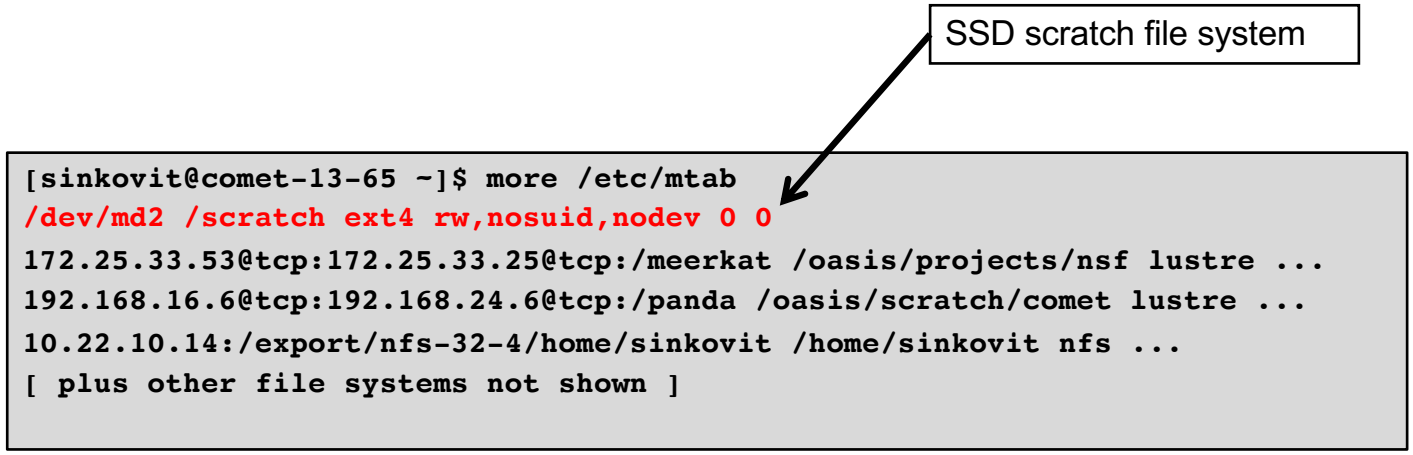
Product name: Intel/Intel SSDSC2BB16.

Hard drive capacity: 160GB

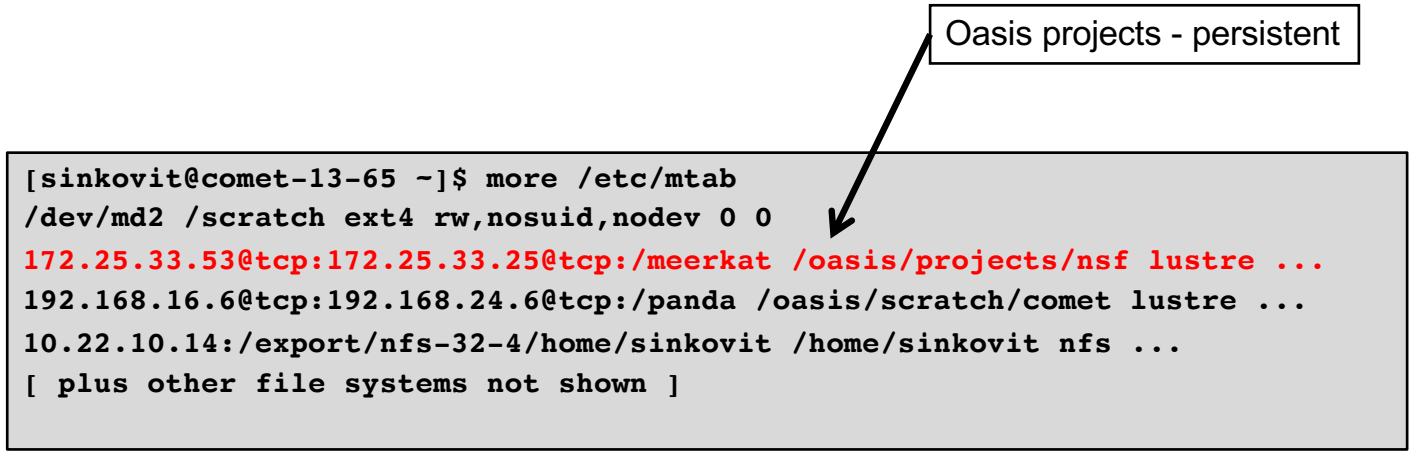
Interface type: SATA 3

Dimensions: 2.5-inch

/etc/mtab lists mounted file systems

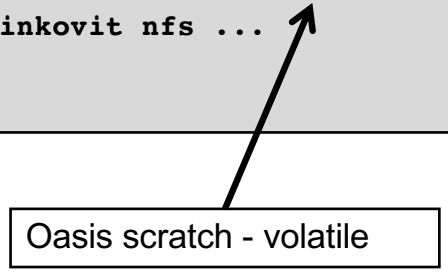


/etc/mtab lists mounted file systems



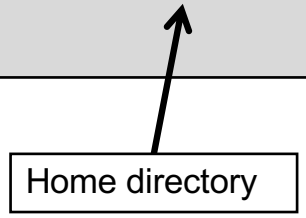
/etc/mtab lists mounted file systems

```
[sinkovit@comet-13-65 ~]$ more /etc/mtab
/dev/md2 /scratch ext4 rw,nosuid,nodev 0 0
172.25.33.53@tcp:172.25.33.25@tcp:/meerkat /oasis/projects/nsf lustre ...
192.168.16.6@tcp:192.168.24.6@tcp:/panda /oasis/scratch/comet lustre ...
10.22.10.14:/export/nfs-32-4/home/sinkovit /home/sinkovit nfs ...
[ plus other file systems not shown ]
```



/etc/mtab lists mounted file systems

```
[sinkovit@comet-13-65 ~]$ more /etc/mtab
/dev/md2 /scratch ext4 rw,nosuid,nodev 0 0
172.25.33.53@tcp:172.25.33.25@tcp:/meerkat /oasis/projects/nsf lustre ...
192.168.16.6@tcp:192.168.24.6@tcp:/panda /oasis/scratch/comet lustre ...
10.22.10.14:/export/nfs-32-4/home/sinkovit /home/sinkovit nfs ...
[ plus other file systems not shown ]
```



df provides information on filesystem usage

```
[sinkovit@comet-13-65 ~]$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/md1               79G   54G   22G  72% /
tmpfs                  63G     0   63G   0% /dev/shm
/dev/md0              239M  106M  117M  48% /boot
/dev/md2              214G   60M  203G   1% /scratch
172.25.33.53@tcp:172.25.33.25@tcp:/meerkat
                      1.4P  798T  530T  61% /oasis/projects/nsf
192.168.16.6@tcp:192.168.24.6@tcp:/panda
                      2.5P  765T  1.8P  31% /oasis/scratch/comet
10.22.10.14:/export/nfs-32-4/home/sinkovit
                       71T   33G   71T   1% /home/sinkovit
```

Finding network information

The ip command (/sbin/ip) is normally used by sys admins, but regular users can use it to learn about networking information

```
$ /sbin/ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 00:1e:67:29:5f:02 brd ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 00:1e:67:29:5f:03 brd ff:ff:ff:ff:ff:ff
4: ib0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
256
    link/infiniband 80:00:00:48:fe:80:00:0a:aa:aa:aa:aa:00:1e:67:03:00:29:5f:07
brd 00:ff:ff:ff:ff:12:40:1b:ff:ff:00:00:00:00:00:00:ff:ff:ff:ff
5: ib1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
256
    link/infiniband 80:00:00:48:fe:8b:bb:bb:bb:bb:bb:b1:00:02:c9:03:00:2f:7b:21
brd 00:ff:ff:ff:ff:12:40:1b:ff:ff:00:00:00:00:00:00:ff:ff:ff:ff
```

Finding OS and kernel information

Use uname to get information on the Linux kernel

```
[sinkovit@comet-ln2 ~]$ uname –r
2.6.32-696.3.2.el6.x86_64
[sinkovit@comet-ln2 ~]$ uname –o
GNU/Linux
[sinkovit@comet-ln2 ~]$ uname –a
Linux comet-ln2.sdsc.edu 2.6.32-696.3.2.el6.x86_64 #1 SMP Tue Jun 20 01:26:55 UTC
2017 x86_64 x86_64 x86_64 GNU/Linux
```

Look in /etc/centos-release to get the Linux distribution (will vary by Linux distro)

```
[sinkovit@comet-ln2 ~]$ cat /etc/centos-release
CentOS release 6.7 (Final)
```

Machine info - overkill?

- We've probably gone a little deeper than is necessary for you to be an effective supercomputer user.
- Think of this as a way to round out your HPC knowledge. You're learning a little bit about the tools of the trade, getting comfortable poking around on a system, acquiring the knowledge that will make it easier to work with your sys admin and picking up the background that will help you to make intelligent decisions in the future.
- Exercise: grab an interactive node (or just the login node) on Comet and experiment with what we've covered. Cheat sheet on the next slide.

Machine info – cheat sheet

File or command	Information provided
less /proc/cpuinfo	CPU specs
less /proc/meminfo	Memory specs and usage
nvidia-smi	GPU specs and usage
cd /sys/devices/system/cpu/cpu0/cache … then look at directory contents	Cache configuration
less /proc/scsi/scsi	Peripherals (e.g. SSDs)
less /etc/mtab	Mounted file systems
df -h	File system usage (readable format)
/sbin/ip link	Networking information
uname -a	OS information
less /etc/centos-release	Centos version

Using the Linux top utility

The top utility is found on all Linux systems and provides a high level view of running processes. Does not give any information at the source code level (profiling), but can still be very useful for answering questions such as

- How many of my processes are running?
- What are the states of the processes (running, sleeping, etc.)?
- Which cores are being utilized?
- Are there any competing processes that may be affecting my performance?
- What fraction of the CPU is each process using?
- How much memory does each process use?
- Is the memory usage growing over time? (Useful for identifying memory leaks)
- How many threads are my processes using?

Customizing top

Top has the following defaults, but is easily customizable

- Processes only (no threads)
- To toggle threads display, type “H” while top is running
- Information for all users
- Can restrict to a single user by launching with “top -u username”
- Process ID, priority, ‘nice’ level, virtual memory, physical memory, shared memory, state, %CPU, %memory, CPU time, command
- To modify, type “f” while top is running and toggle fields using letters
- Update information every 3 seconds
- Change refresh rate by launching with “top -d *n*”
- Ordered by CPU usage
- Type “M” to order by memory usage

Non-threaded code

stivoknis — sinkovit@gcn-17-57:~ — ssh — 94×33

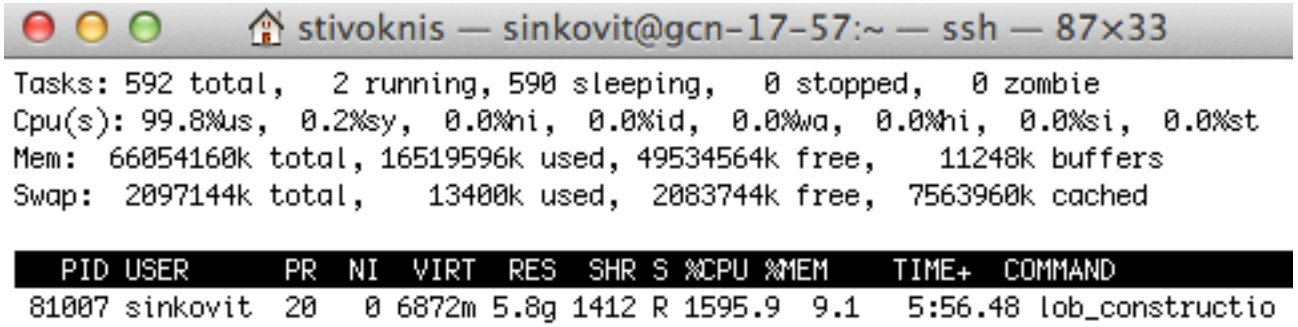
```
top - 08:37:00 up 60 days, 14:23,  1 user,  load average: 15.32, 10.36, 6.12
Tasks: 624 total,  17 running, 607 sleeping,   0 stopped,   0 zombie
Cpu(s): 68.7%us,  1.3%sy,  0.0%ni, 29.9%id,  0.1%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  66054160k total, 37885796k used, 28168364k free,     8808k buffers
Swap:  2097144k total,    13400k used,  2083744k free, 32927192k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
70388 sinkovit  20   0  194m  76m 1612 R 100.0  0.1   1:31.06 lobfaster.pl
72547 sinkovit  20   0  120m 2976 1612 R 100.0  0.0   0:01.49 lobfaster.pl
72516 sinkovit  20   0  127m 9.9m 1608 R 100.0  0.0   0:02.09 lobfaster.pl
72526 sinkovit  20   0  121m 3388 1612 R 100.0  0.0   0:01.84 lobfaster.pl
72535 sinkovit  20   0  121m 4208 1612 R 100.0  0.0   0:01.73 lobfaster.pl
72565 sinkovit  20   0  120m 3212 1612 R 100.0  0.0   0:01.01 lobfaster.pl
72268 sinkovit  20   0  130m  12m 1612 R 98.9  0.0   0:11.96 lobfaster.pl
72359 sinkovit  20   0  123m 5976 1612 R 98.9  0.0   0:09.77 lobfaster.pl
72460 sinkovit  20   0  127m  10m 1612 R 98.9  0.0   0:08.38 lobfaster.pl
72481 sinkovit  20   0  131m  13m 1612 R 98.9  0.0   0:07.44 lobfaster.pl
72529 sinkovit  20   0  122m 4576 1612 R 98.9  0.0   0:01.82 lobfaster.pl
72439 sinkovit  20   0  130m  12m 1612 R 97.0  0.0   0:08.64 lobfaster.pl
72590 sinkovit  20   0  120m 3140 1612 R 71.7  0.0   0:00.37 lobfaster.pl
72602 sinkovit  20   0  120m 2576 1612 R 38.8  0.0   0:00.20 lobfaster.pl
72605 sinkovit  20   0  120m 2528 1600 R 34.9  0.0   0:00.18 lobfaster.pl
72608 sinkovit  20   0  119m 2340 1600 R 21.3  0.0   0:00.11 lobfaster.pl
```

16 processes, each using anywhere from 21.3% to 100% of a compute core.

Memory footprint (RES) is minimal, with each process only using up to 76 MB.

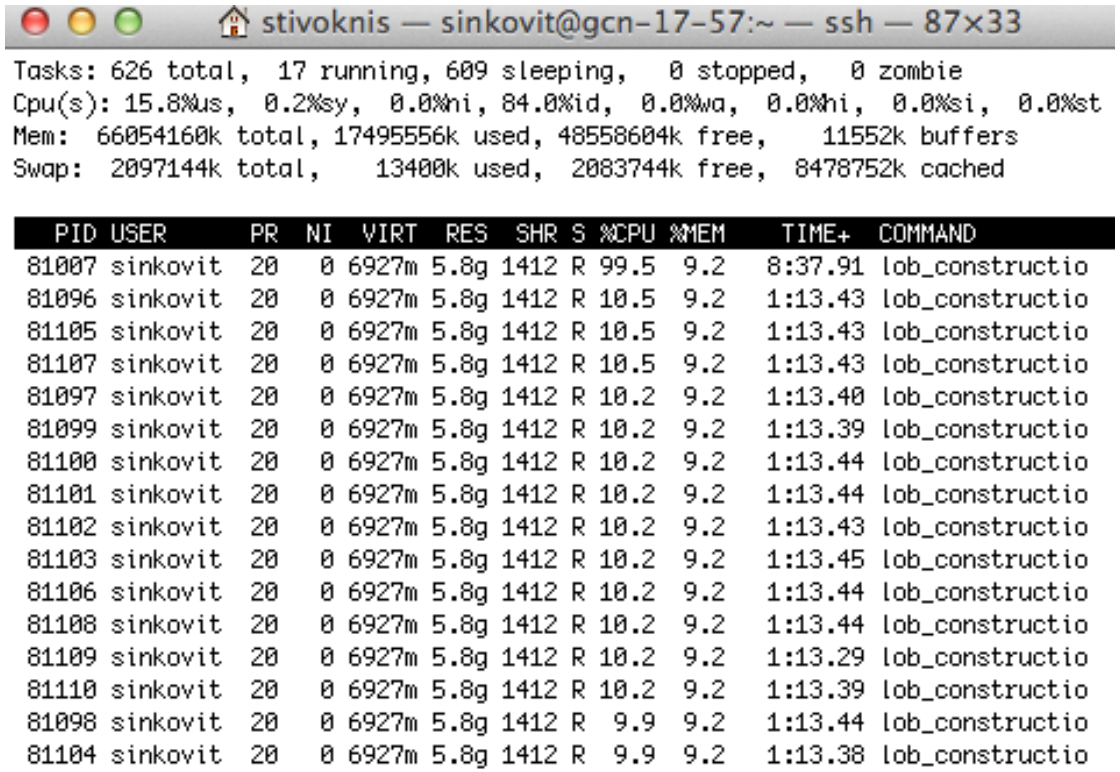
CPU times ranging from 0.11s (just started) to 1:31

Threaded code (thread display off)



Threaded code with thread display toggled to the “off” position. Note the heavy CPU usage, very close to 1600%

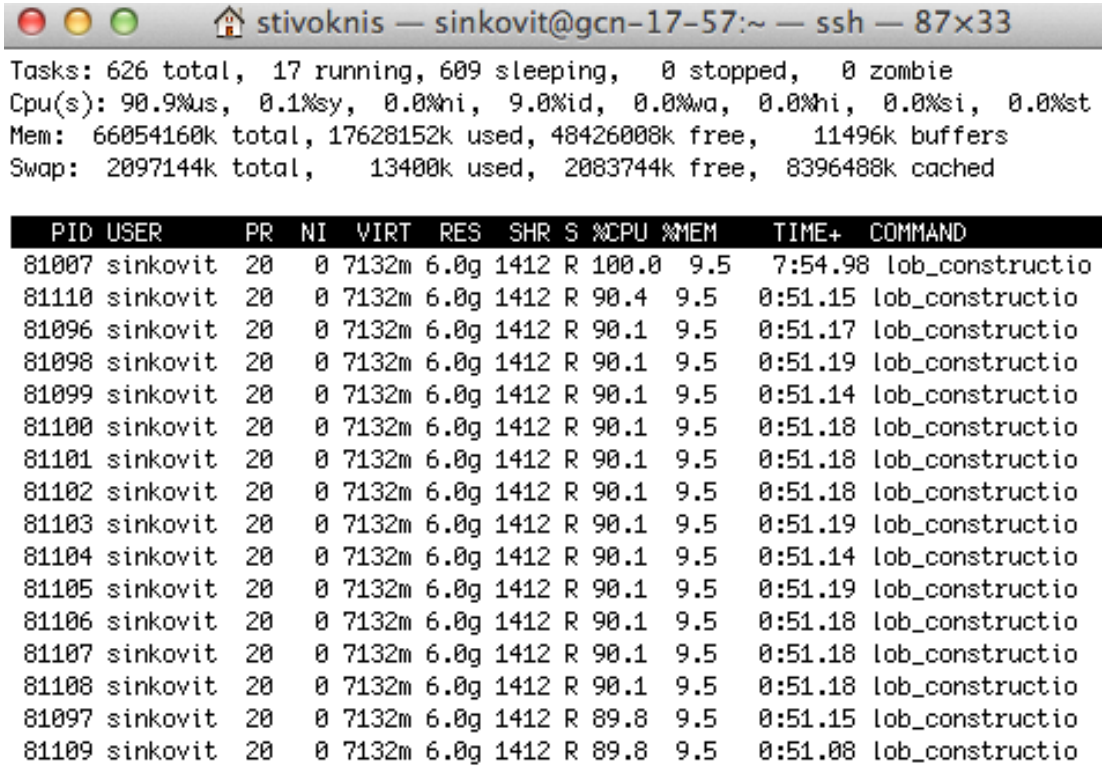
Threaded code (thread display on)



16 threads, with only one thread making good use of CPU

Total memory usage 5.8 GB (9.2% of available)

Threaded code (thread display on)



16 threads, all making good (but not ideal) use of the compute cores

Getting an interactive compute node

- All exercises should be run on the compute nodes, not the login nodes. You will have dedicated access to the former, while the latter are shared by all users connecting to the system.

  $ getcpu1

  $ srun --reservation=SI2019DAY1 --pty --nodes=1 --ntasks-per-node=24 -t 01:00:00 --wait=0 --export=ALL /bin/bash

- Once you have been assigned a compute node, you can access it directly.

  $ ssh comet-11-12 (your node name will be different)

  $ more /etc/security/access.conf
  -:ALL EXCEPT root (wheel) (xsede-admin) sinkovit:ALL

Top example

- The program lineq.c generates a random vector and matrix of rank N, calls the linear solver DGESV (Ax=b) then reports run time. Does this for **10** matrices. Linking with MKL accesses threaded library
- Compile using the following command
  icc -O3 -o lineq lineq.c -mkl
- Open a second terminal, login directly to compute node and launch top
  For example: ssh comet-11-12
- On the first terminal, run program with different matrix sizes
  ./lineq 5000
  ./lineq 10000
- Monitor CPU and memory usage. Do you notice anything funny? If so, see if you can fix the problem.

Supplementary material

Profiling your code with gprof

gprof is a profiling tool for UNIX/Linux applications. First developed in 1982, it is still extremely popular and very widely used. It is always the first tool that I use for my work.

Universally supported by all major C/C++ and Fortran compilers

Extremely easy to use

1. Compile code with -pg option: adds instrumentation to executable
2. Run application: file named gmon.out will be created.
3. Run gprof to generate profile: gprof a.out gmon.out

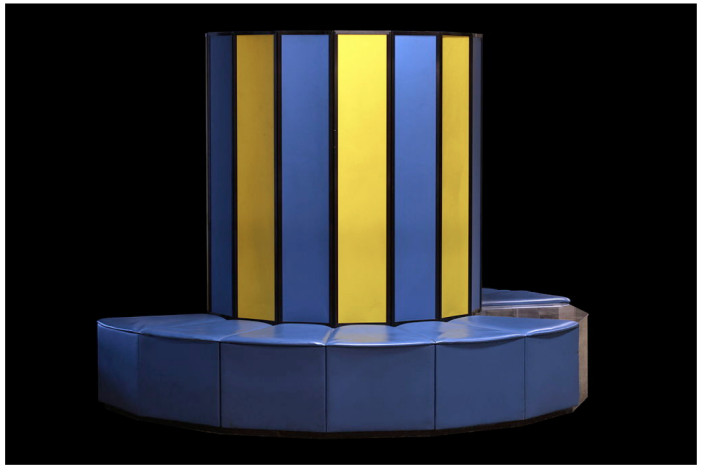
Introduces virtually no overhead

Output is easy to interpret

1982!

Worth reflecting on the fact that gprof goes back to 1982. Amazing when considered in context of the leading technology of the day

Michael Douglas as Gordon Gecko in Wall Street, modeling early 1980s cell phone. List price ~ $3000



Cray X-MP with 105 MHz processor. High end configuration (four CPUs, 64 MB memory) has 800 MFLOP theoretical peak. Cost ~ $15M

gprof flat profile

The gprof flat profile is a simple listing of functions/subroutines ordered by their relative usage. Often a small number of routines will account for a large majority of the run time. Useful for identifying hot spots in your code.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 68.60    574.72   574.72   399587     1.44     1.44  get_number_packed_data
 13.48    687.62   112.90                             main
 11.60    784.81    97.19   182889     0.53     0.53  quickSort_double
  2.15    802.85    18.04   182889     0.10     0.63  get_nearest_events
  1.52    815.56    12.71                             __c_mcopy8
  1.28    826.29    10.73                             _mcount2
  0.96    834.30     8.02    22183     0.36     0.36  pack_arrays
  0.12    835.27     0.97                             __rouexit
  0.08    835.94     0.66                             __rouinit
  0.06    836.45     0.51    22183     0.02     5.58  Is_Hump
  0.05    836.88     0.44        1   436.25   436.25  quickSort
```

gprof call graph

The gprof call graph provides additional levels of detail such as the exclusive time spent in a function, the time spent in all children (functions that are called) and statistics on calls from the parent(s)

```
index % time    self  children    called     name
[1]     96.9  112.90  699.04                 main [1]
              574.72    0.00  399587/399587      get_number_packed_data [2]
                0.51  123.25   22183/22183       Is_Hump [3]
                0.44    0.00       1/1           quickSort [11]
                0.04    0.00       1/1           radixsort_flock [18]
                0.02    0.00       2/2           ID2Center_all [19]
-----------------------------------------------
              574.72    0.00  399587/399587      main [1]
[2]     68.6  574.72    0.00  399587         get_number_packed_data [2]
-----------------------------------------------
                0.51  123.25   22183/22183       main [1]
[3]     14.8    0.51  123.25   22183          Is_Hump [3]
               18.04   97.19  182889/182889      get_nearest_events [4]
                8.02    0.00   22183/22183       pack_arrays [8]
                0.00    0.00   22183/22183       pack_points [24]
```

The value of re-profiling

After optimizing the code, we find that the function main() now accounts for 40% of the run time and would be a likely target for further performance improvements.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 41.58     36.95    36.95                             main
 26.41     60.42    23.47    22183     1.06     1.06  get_number_packed_data
 11.58     70.71    10.29                             __c_mcopy8
 10.98     80.47     9.76   182889     0.05     0.05  get_nearest_events
  8.43     87.96     7.49    22183     0.34     0.34  pack_arrays
  0.57     88.47     0.51    22183     0.02     0.80  Is_Hump
  0.20     88.65     0.18        1   180.00   180.00  quickSort
  0.08     88.72     0.07                             _init
  0.05     88.76     0.04        1    40.00    40.00  radixsort_flock
  0.02     88.78     0.02        1    20.00    20.00  compute_position
  0.02     88.80     0.02        1    20.00    20.00  readsource
```

Limitations of gprof

- grprof only measures time spent in user-space code and does not account for system calls or time waiting for CPU or I/O
- gprof can be used for MPI applications and will generate a gmon.out.id file for each MPI process. But for reasons mentioned above, it will not give an accurate picture of the time spent waiting for communications
- gprof will not report usage for un-instrumented library routines
- In the default mode, gprof only gives function level rather than statement level profile information. Although it can provide the latter by compiling in debug mode (-g) and using the gprof -l option, this introduces a lot of overhead and disables many compiler optimizations.

  In my opinion, I don't think this is such a bad thing. Once a function has been identified as a hotspot, it's usually obvious where the time is being spent (e.g. statements in innermost loop nesting)

gprof for threaded codes

gprof has limited utility for threaded applications (e.g. parallelized with OpenMP, Pthreads) and only reports usage for the main thread

But … there is a workaround
http://sam.zoy.org/writings/programming/gprof.html

“gprof uses the internal ITIMER_PROF timer which makes the kernel deliver a signal to the application whenever it expires. So we just need to pass this timer data to all spawned threads”

http://sam.zoy.org/writings/programming/gprof-helper.c

gcc -shared -fPIC gprof-helper.c -o gprof-helper.so -lpthread -ldl

LD_PRELOAD=./gprof-helper.so

… then run your code

gprof example1

- Compile gprof_ex.f using the following command
  ifort -pg -O3 -fno-inline -o gprof_ex gprof_ex.f
- Run as follows on compute node
  time ./gprof_ex 100000000
- Generate profile and examine results
  gprof gprof_ex gmon.out > profile_gp
- Repeat without function inlining disabled
  ifort -pg -O3 -o gprof_ex gprof_ex.f

gprof example 1 (examining call tree)

```
-----------------------------------------------
                0.85    5.23       1/1           MAIN__ [3]
[6]     12.4    0.85    5.23       1         sub1_ [6]
                4.05    0.00 100000000/100000000     f2_ [7]
                1.18    0.00 100000000/200000000     f1_ [8]
-----------------------------------------------
                4.05    0.00 100000000/100000000     sub1_ [6]
[7]      8.2    4.05    0.00 100000000         f2_ [7]
-----------------------------------------------
                1.18    0.00 100000000/200000000     sub1_ [6]
                1.18    0.00 100000000/200000000     sub2_ [9]
[8]      4.8    2.36    0.00 200000000         f1_ [8]
-----------------------------------------------
                0.39    1.18       1/1           MAIN__ [3]
[9]      3.2    0.39    1.18       1         sub2_ [9]
                1.18    0.00 100000000/200000000     f1_ [8]
-----------------------------------------------
```

- sub1 called by MAIN once
- calls f2 10^8
- calls f1 10^8

- f2 called by sub1 10^8

- f1 called by sub1 10^8
- f1 called by sub2 10^8

- sub2 called by MAIN once
- calls f1 10^8

gprof example 2

- Compile lineq.c using the following command
  icc -pg -O3 -o lineq lineq.c -mkl
- Run as follows on compute node
  export OMP_NUM_THREADS=1; ./lineq 5000
- Generate profile and examine results
  gprof lineq gmon.out > profile_lineq
- How does time reported by program compare to total time in the profile?

Manually instrumenting codes

- Performance analysis tools ranging from the venerable (gprof) to the modern (TAU) are great, but they all have several downsides
  - May not be fully accurate
  - Can introduce overhead
  - Sometimes have steep learning curves

- Once you really know your application, your best option is to add your own instrumentation. Will automatically get a performance report every time you run the code.

- There are many ways to do this and we'll explore portable solutions in C/C++ and Fortran. Note that there are also many heated online discussions arguing over how to properly time codes.

Linux time utility

If you just want to know the overall wall time for your application, can use the Linux time utility. Reports three times

- real – elapsed (wall clock) time for executable
- user – CPU time integrated across all cores
- sys – system CPU time

```
$ export OMP_NUM_THREADS=16 ; time ./lineq_mkl 30000
Times to solve linear sets of equations for n = 30000
t = 70.548615

real     1m10.733s  ← wall time
user     17m23.940s ← CPU time summed across all cores
sys      0m2.225s
```

Manually instrumenting C/C++ codes

The C gettimeofday() function returns time from start of epoch (1/1/1970) with microsecond precision. Call before and after the block of code to be timed and perform math using the tv_sec and tv_usec struct elements

```
struct timeval tv_start, tv_end;

gettimeofday(&tv_start, NULL);
// block of code to be timed
gettimeofday(&tv_end, NULL);

elapsed = (tv_end.tv_sec - tv_start.tv_sec) +
          (tv_end.tv_usec - tv_start.tv_usec) / 1000000.0;
```

Manually instrumenting Fortran codes

The Fortran90 system_clock function returns number of ticks of the processor clock from some unspecified previous time. Call before and after the block of code to be timed and perform math using the elapsed_time function (see next slide)

```
integer clock1, clock2;
double precision elapsed_time

call system_clock(clock1)
// block of code to be timed
call system_clock(clock2)

time = elapsed_time(clock1, clock2)
```

Manually instrumenting Fortran codes (cont.)

Using system_clock can be a little complicated since we need to know the length of a processor cycle and have to be careful about how we handle overflows of counter. Write this once and reuse everywhere.

```
double precision function elapsed_time(c1, c2)
implicit none
integer, intent(in) :: c1, c2
integer ticks, clockrate, clockmax

call system_clock(count_max=clockmax, count_rate=clockrate)
ticks = c2-c1
if(ticks < 0) then
  ticks = clockmax + ticks
endif
elapsed_time = dble(ticks)/dble(clockrate)

return
end function elapsed_time
```

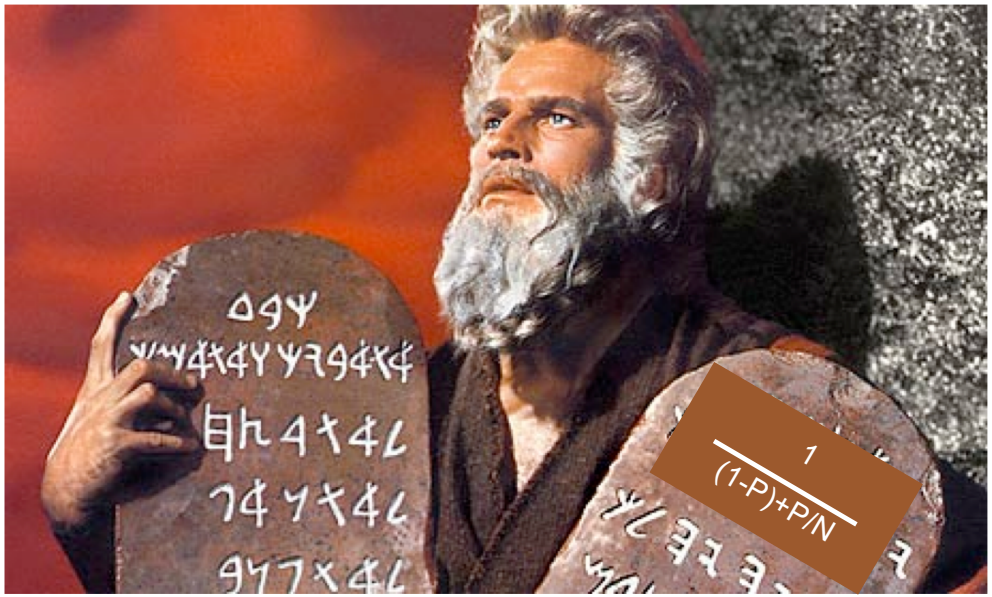
A note on granularity

Don't try to time at too small a level of granularity, such as measuring the time associated with a single statement within a loop

```
elapsed = 0.0;

for (i=0; i<n; i++) {
  w[i] = x[i] * y[i];
  gettimeofday(&tv_start, NULL);
  z[i] = sqrt(w[i]) + x[i];
  gettimeofday(&tv_end, NULL);
  elapsed += (tv_end.tv_sec - tv_start.tv_sec) +
             (tv_end.tv_usec - tv_start.tv_usec) / 1000000.0;
}
```

Although they're pretty lightweight, there is still a cost associated with calls to gettimeofday or system_clock. In addition, the insertion of these calls into loops can impact the flow and hamper optimizations by the compiler.

Amdahl's Law – kind of a big deal



Charlton Heston in *Moses and the Ten Commandments*, delivering a slightly updated version to the SDSC Summer Institute

Amdahl's Law

Amdahl's law sets an upper limit on the speedup of a parallel code based on the serial content.

- Let P be the fraction of the code that can be run in parallel
- Let (1-P) be the serial fraction of the code
- Let N be the number of parallel threads or processes

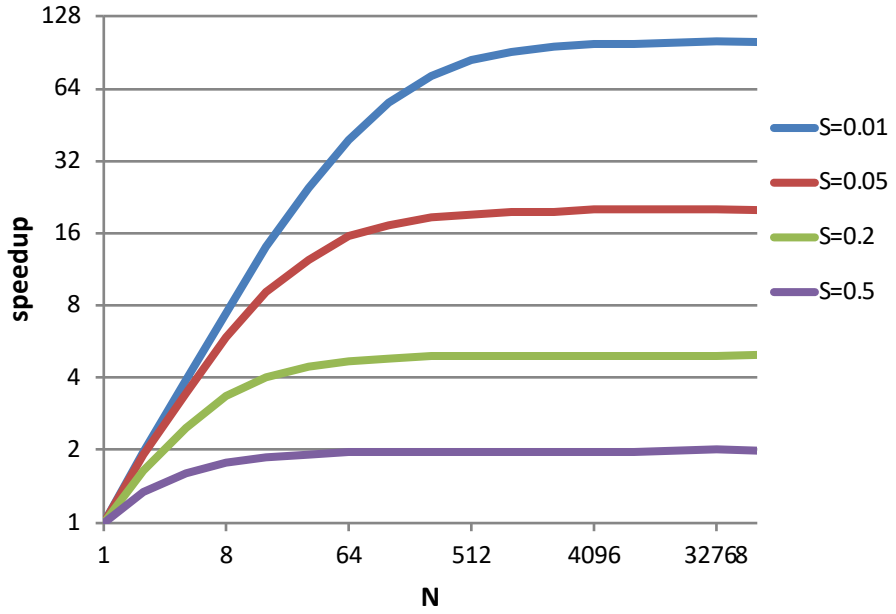
$$S(N) = \frac{1}{(1-P) + P/N} \qquad S(\infty) = \frac{1}{(1-P)}$$

In reality, you will probably do a good bit worse than Amdahl's law due to a number of factors, most importantly

- Load imbalance – processes assigned different amounts of work
- Communications overhead – latency and bandwidth

Amdahl's Law

The theoretical maximum speedup, running on an infinite number of compute cores, is the inverse of the serial content. This places a very stringent bound on the benefits of parallelization



Amdahl's Law

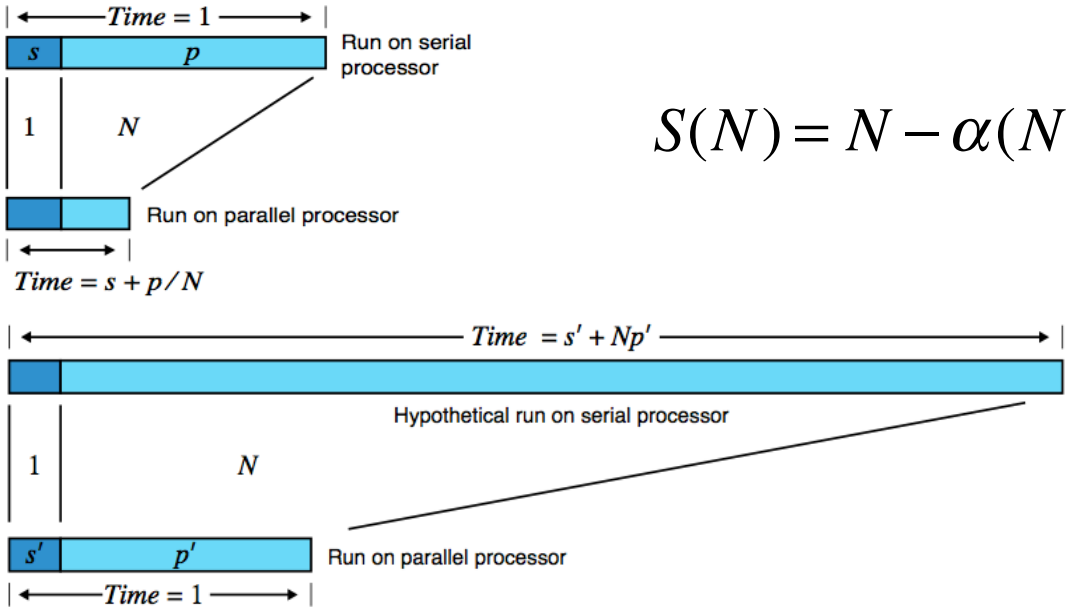
Fortunately, things aren't as bad as they appear. For many applications, there is a lot of work that can be done in parallel and the serial content is rather minimal

- Updating grid cells in discretized solutions of PDEs
- Calculating forces on particles in molecular dynamics, N-body problems
- Ensemble calculations – many repetitions of calculation with different data sets or slightly different input parameters

If your application falls into the last category, there is no need for you to worry about parallelization at the program level. Instead, just develop a workflow that allows you to run the serial instances of your code in parallel.

Gustafson's Law

A limitation of Amdahl's Law is that it assumes you want to run a fixed size problem on an increasing number of processors. In many real problems, you'll want to increase the problem size as the processing power grows



$$S(N) = N - \alpha(N-1)$$

Gustafson, Communications of the ACM 31(5), 1988 532-533

Strong scaling vs. Weak Scaling

The discussion of Amdahl's Law and Gustafson's Law segues into the topic of strong vs. weak scaling

- Strong scaling – how does the run time scale (decline) as the number of processors is increased? Ideally, linear speedup (t ~ 1/N)

- Weak scaling – how does the run time vary as the work per process stays constant while the problem size grows. Ideally, t is unchanged.

Scaling experiment

- Compile lineq.c using the following command
  icc -O3 -o lineq lineq.c -mkl
- On a Comet compute node, run with a variety of problem sizes using 1, 2, 4, 8, 16 and 24 threads. Note the run times and any trends in scalability as the problem size is increased from N=2000 to N=6000

  export OMP_NUM_THREADS=1; ./lineq 4000
  export OMP_NUM_THREADS=2; ./lineq 4000
  export OMP_NUM_THREADS=4; ./lineq 4000
  …
  export OMP_NUM_THREADS=1; ./lineq 6000
  export OMP_NUM_THREADS=2; ./lineq 6000
  export OMP_NUM_THREADS=4; ./lineq 6000
  …