The Spartan HPC System at the University of Melbourne

COMP90024 Cluster and Cloud Computing

University of Melbourne, March 25, 2020

lev.lafayette@unimelb.edu.au

Who Is This Guy?

Currently Senior HPC DevOps Engineer at University of Melbourne since 2015

Has also worked for:
Victorian Partnership for Advanced Computing as HPC Systems Administrator and Quality Assurance Coordinator
Ministry of Foreign Affairs and Cooperation of Timor-Leste, Information and Communications Technology Consultant
Parliament of Victoria as an Electorate Officer, specialising in various database services etc.

Has done some teaching to some researchers around Australia, written a few books and few more journal articles. Have given a few presentations at various local, national, and international conferences.

Degrees in politics, philosophy, sociology, technology management, project management, adult and tertiary education, information systems etc.

http://levlafayette.com
https://www.linkedin.com/in/levlafayette/

Outline of Lecture

"This is an advanced course but we get mixed bag: students that have 5+ years of MPI programming on supercomputers, to students that have only done Java on Windows."

- Some background on supercomputing, high performance computing, parallel computing, research computing (they're not the same thing!).
- Why performance and scale matters, and why it should matter to *you*.
- An introduction to Spartan, University of Melbourne's HPC/cloud hybrid system
- Logging in, help, and environment modules.
- Job submission with Slurm workload manager; simple submissions, multi-core, multi-node, job arrays, job dependencies, interactive jobs.
- Parallel programming with shared memory and threads (OpenMP) and distributed memory and message passing (OpenMPI)
- Tantalising hints about more advanced material on message passing routines.

Supercomputers

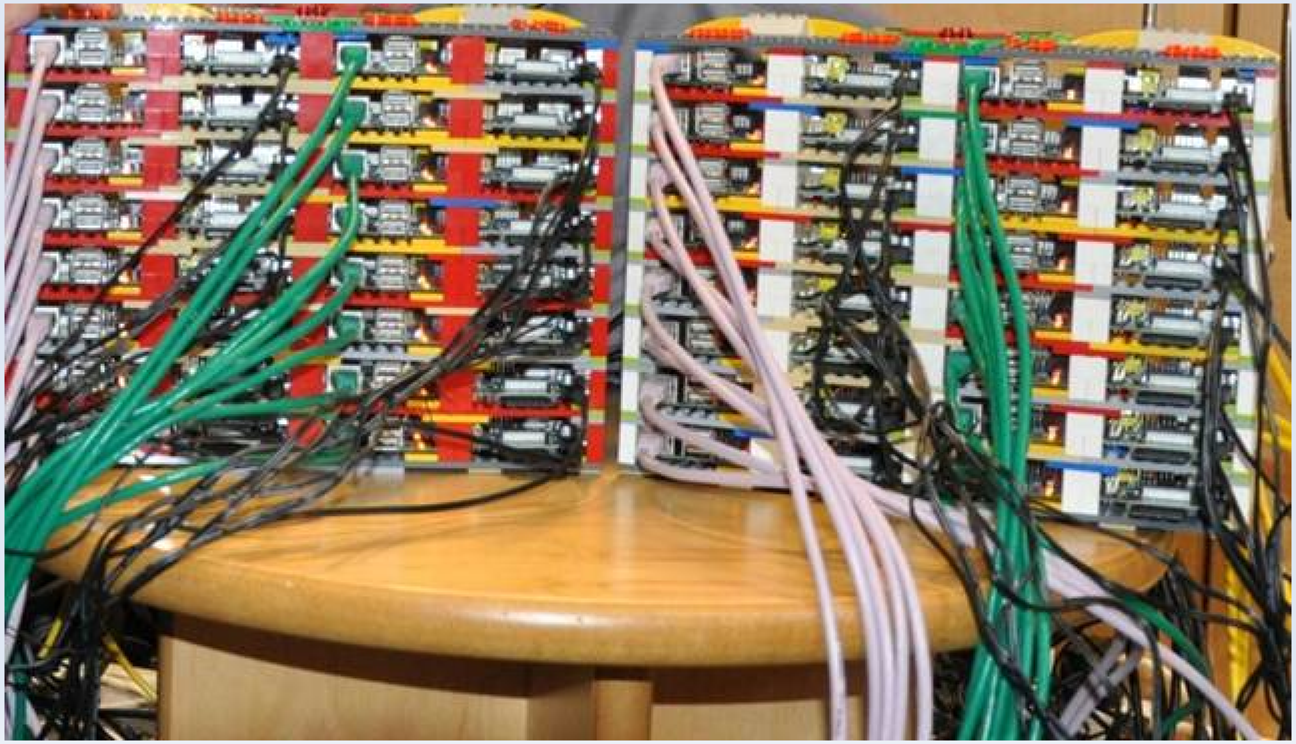
'Supercomputer" arbitrary term. In general use it means any single computer system (itself a contested term) that has exceptional processing power for its time. One metric is the number of floating-point operations per second (FLOPS) such a system can carry out. The Top500 list is based on FLOPS using LINPACK - HPC Challenge is a broader, more interesting metric. The current number #1 system is Summit from the Oak Ridge National Laboratory in the United States.

1994: 170.40 GFLOPS
1996: 368.20 GFLOPS
1997: 1.338 TFLOPS
1999: 2.3796 TFLOPS
2000: 7.226 TFLOPS
2004; 70.72 TFLOPS
2005: 280.6 TFLOPS
2007: 478.2 TFLOPS
2008: 1.105 PFLOPS
2009: 1.759 PFLOPS
2010: 2.566 PFLOPS
2011: 10.51 PFLOPS
2012: 17.59 PFLOPS
2013: 33.86 PFLOPS
2014: 33.86 PFLOPS
2015: 33.86 PFLOPS
2016: 93.01 PFLOPS
2017: 93.01 PFLOPS
2018: 143.00 PFLOPS (200.795 Rpeak)
2019: 148.60 PFLOS (200,795 Rpeak)

High Performance Computing

High-performance computing (HPC) is any computer system whose architecture allows for above average performance. A system that is one of the most powerful in the world, but is poorly designed, could be a "supercomputer".

Clustered computing is when two or more computers serve a single resource. This improves performance and provides redundancy; typically a collection of smaller computers strapped together with a high-speed local network (e.g., Myrinet, InfiniBand, 10 Gigabit Ethernet). Even a cluster of Raspberry Pi with a Lego chassis (University of Southampton, 2012)!



Horse and cart as a computer system and the load as the computing tasks. Efficient arrangement, bigger horse and cart, or a teamster?

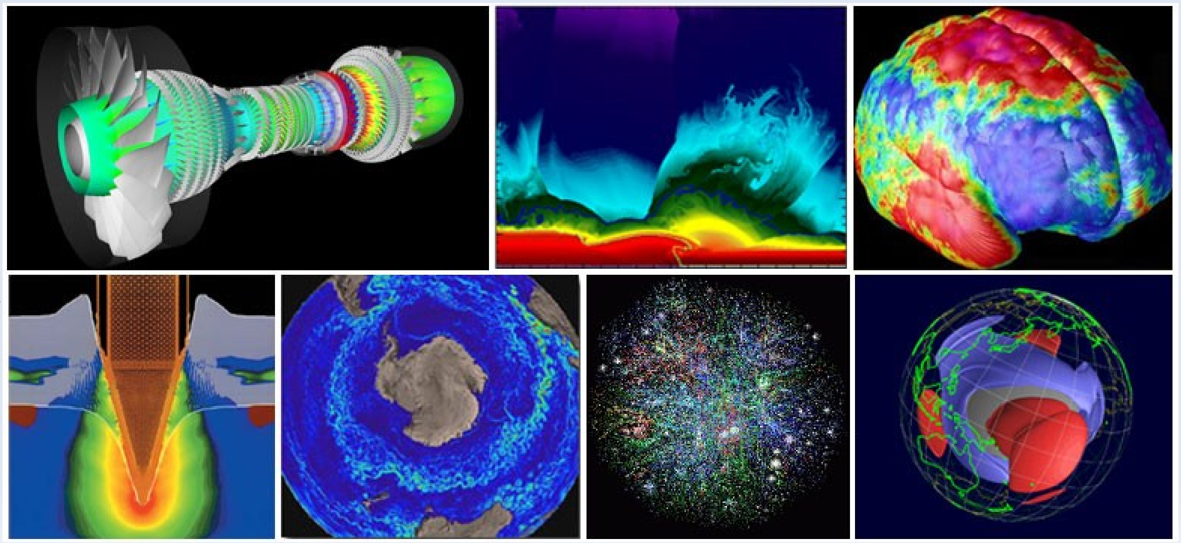
The clustered HPC is the most efficient, economical, and scalable method, and for that reason it dominates supercomputing.

Parallel and Research Programming

With a cluster architecture, applications can be more easily parallelised across them. Parallel computing refers to the submission of jobs or processes over multiple processors and by splitting up the data or tasks between them (random number generation as data parallel, driving a vehicle as task parallel).

Research computing is the software applications used by a research community to aid research. This skills gap is a major problem and must be addressed because as the volume, velocity, and variety of datasets increases then researchers will need to be able to process this data.

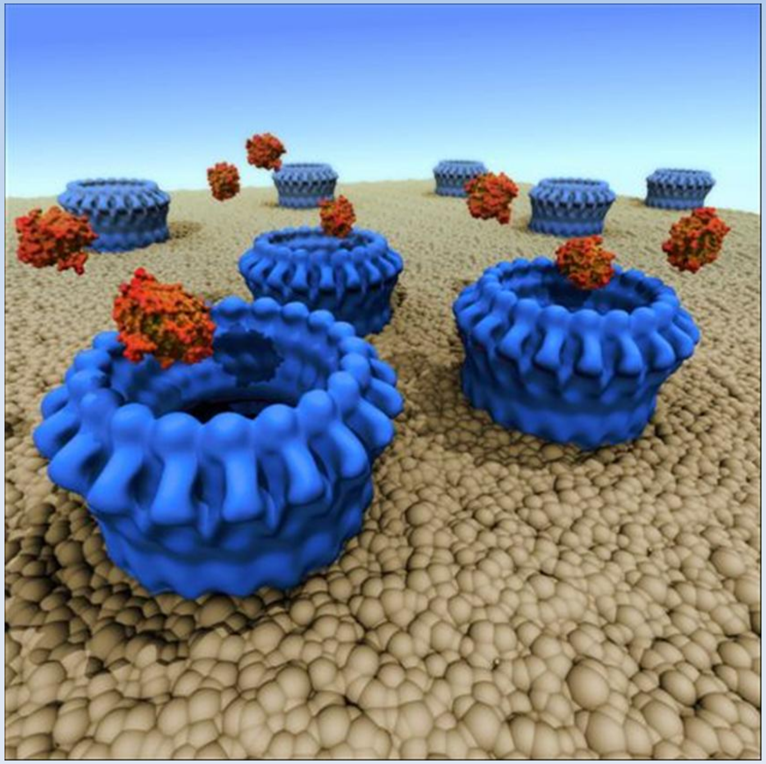
Computational capacity does have a priority (the system must exist prior to use), in order for that capacity to realised in terms of usage a skill-set competence must also exist. The the core issue is that high performance compute clusters is just speed and power but also usage, productivity, correctness, and reproducibility.
(image from Lawrence Livermore National Laboratory)



There is nascent research that show a strong correlation between research output and availability of HPC facilities. (Apon et al 2010)

Some Local Examples

Researchers from Monash University, the Peter MacCallum Cancer Institute in Melbourne, the Birkbeck College in London, and VPAC in 2010 unravelled the structure the protein perforin to determine how pathogenic cells are attacked by white blood cells.



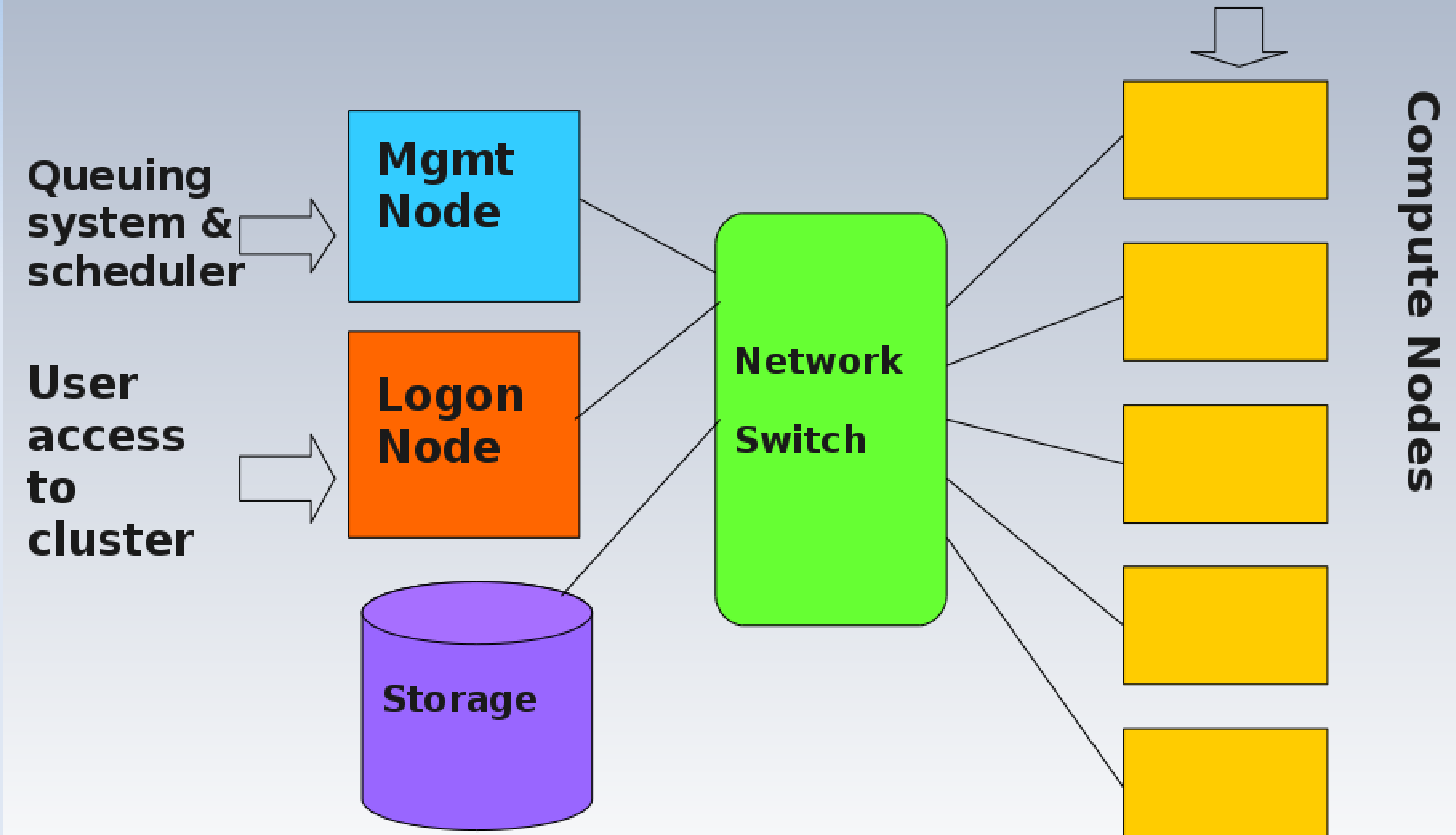
In 2015 researchers from VLSCI announced how natural antifreeze proteins bind to ice to prevent it growing which has important implications for extending donated organs and protecting crops from frost damage.

In 2016 CSIRO researchers successfully manipulated the behaviour of Metallic Organic Frameworks to control their structure and alignment which provides opportunities for real-time and implantable medical electric devices.

In 2019 University of Melbourne researchers used Spartan to automatically detect, classify and count unique classes of trucks and trailers.

Most research these days that involves large datasets and/or complex computations is made on HPC systems.

HPC Cluster Design



It's A GNU/Linux World

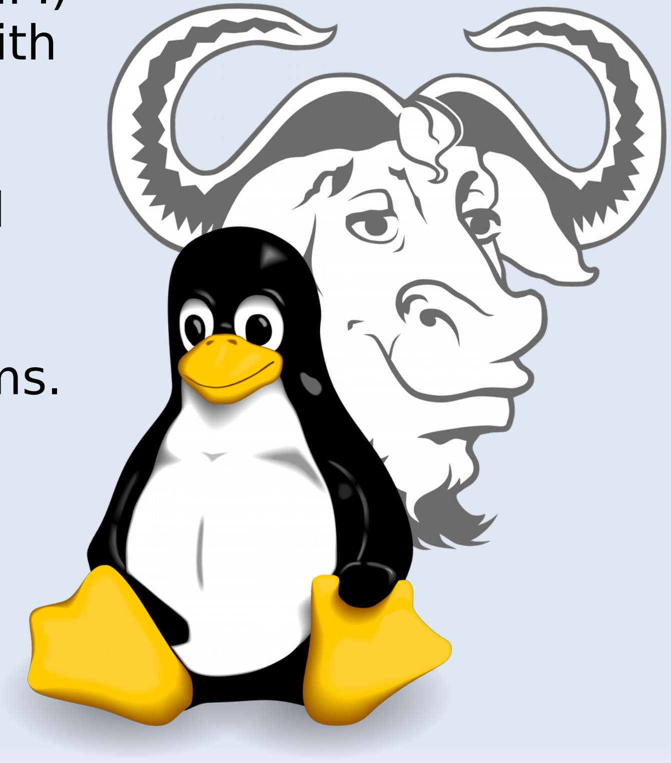
From November 2017 onwards of the Top 500 Supercomputers worldwide, *every single machine* used Linux.

The command-line interface provides a great deal more power and is very resource efficient.

GNU/Linux scales and does so with stability and efficiency. Critical software such as the Message Passing Interface (MPI) and nearly all scientific programs are designed to work with GNU/Linux.

The operating system and many applications are provided as "free and open source", which means that not only are there are some financial savings, were also much better placed to improve, optimize and maintain specific programs.

Free or open source software (not always the same thing) can be can be compiled from source for the specific hardware and operating system configuration, and can be optimised according to compiler flags. There is necessary where every clock cycle is important.



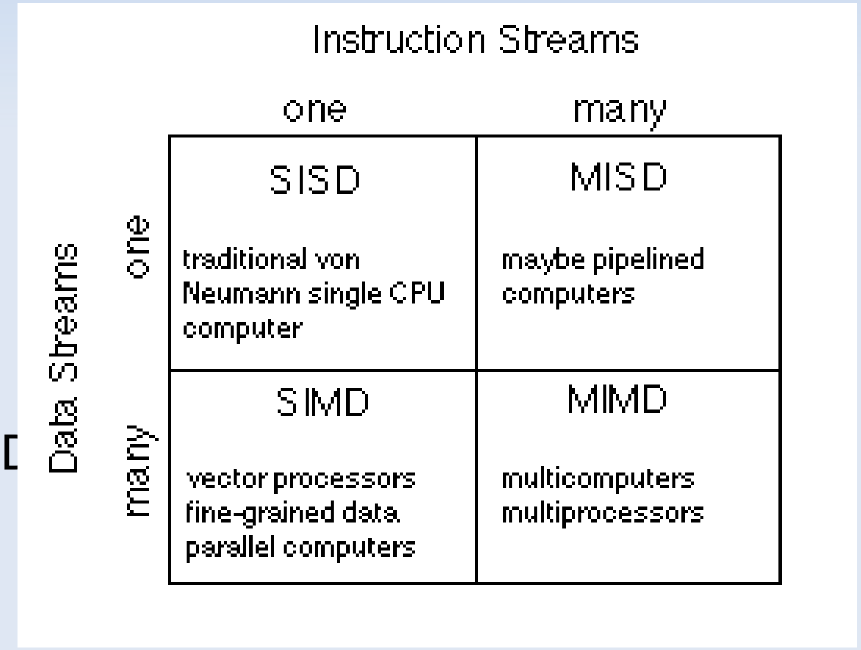
Flynn's Taxonomy and Multicore Systems

It is possible to illustrate the degree of parallelisation by using Flynn's Taxonomy of Computer Systems (1966), where each process is considered as the execution of a pool of instructions (instruction stream) on a pool of data (data stream).

Over time computing systems have moved towards multi-processor, multi-core, and often multi-threaded and multi-node systems.

As computing technology has moved increasingly to the MIMD taxonomic classification additional categories have been added:

Single program, multiple data streams (SPMD)
Multiple program, multiple data streams (MPM



Data Streams	Instruction Streams: one	Instruction Streams: many
one	SISD traditional von Neumann single CPU computer	MISD maybe pipelined computers
many	SIMD vector processors fine-grained data parallel computers	MIMD multicomputers multiprocessors

Some trends include GPGPU development, massive multicore systems (e.g., The Angstrom Project, the Tile CPU with 1000 cores) and massive network connectivity and shared resources (e.g., Plan9 Operating System).

(Image from Dr. Mark Meyer, Canisius College)

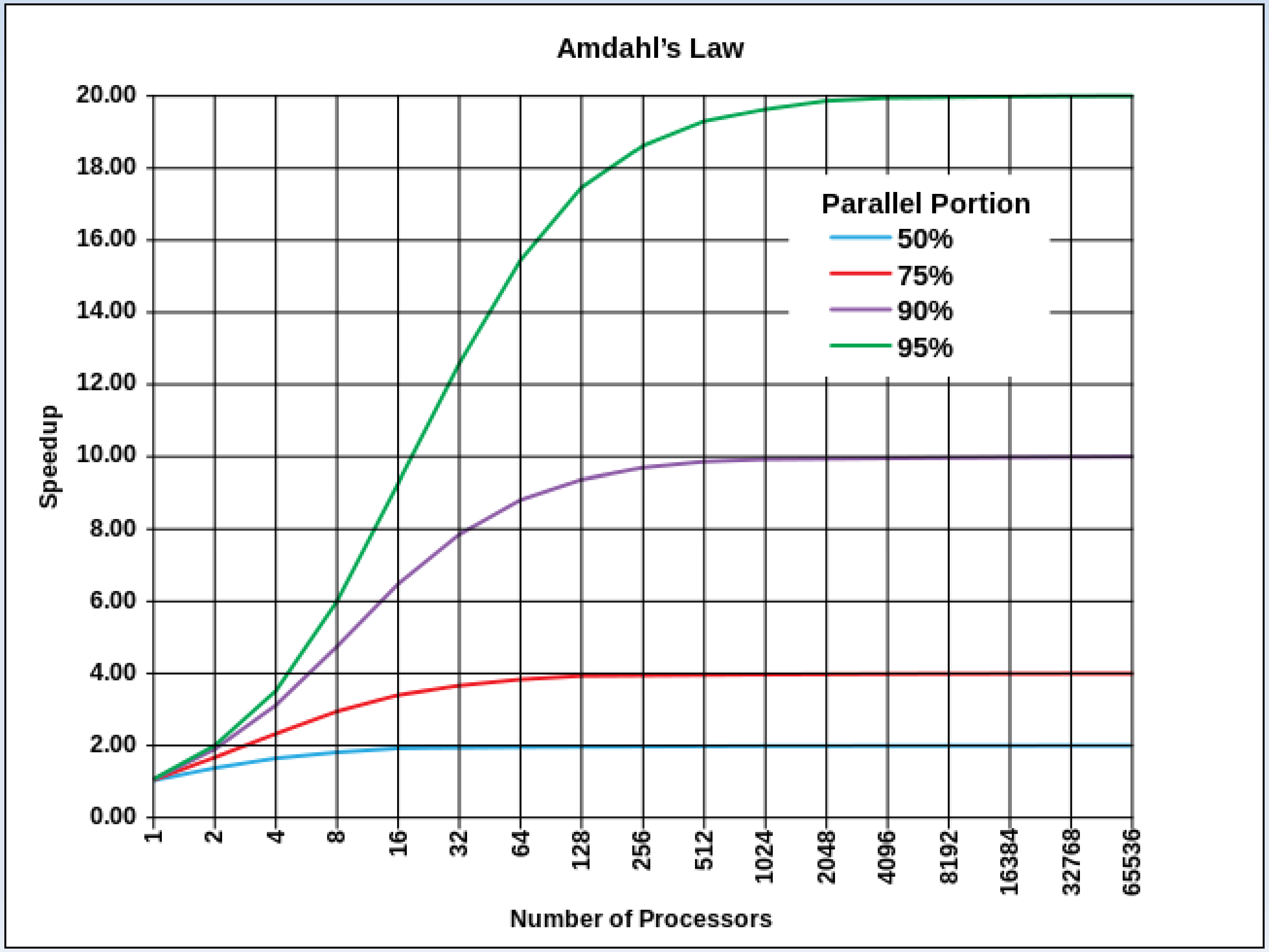
Limitations of Parallel Computation

Parallel programming and multicore systems should mean better performance. This can be expressed a ratio called speedup

Speedup (p) = Time (serial)/ Time (parallel)

Correctness in parallelisation requires synchronisation. Synchronisation and atomic operations causes loss of performance, communication latency.

Amdahl's law, establishes the maximum improvement to a system when only part of the system has been improved. Gustafson and Barsis noted that Amadahl's Law assumed a computation problem of fixed data set size.



What's More Important Than Performance?

There are many things in software and hardware that are more important than performance.

Correctness of code and signal
Clarity of code and architecture
Reliability of code and equipment
Modularity of code and components
Readability of code and hardware documentation
Compatibility of code and hardware

All this must be done before engaging in performance.

Common rule in parallel programming: *develop a working serial version of your code first and then work out what parts can be made parallel.*

Do We Need Performance?

Early systems needed performance engineering by necessity. Software development was more advanced that hardware capacity. However, hardware developments increased rapidly and for many years performance engineering was not a priority as a result.

This led to many major software engineers to argue that optimisation was something to avoid, or at very least, defer in preference to working, readable code, and that does not have difficult levels of complexity.

”More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.” (William Wulf, 1972)

”Premature optimization is the root of all evil.” (Donald Knuth, 1974)

”The First Rule of Program Optimization: Don’t do it. The Second Rule of Program Optimization - For experts only: Don’t do it yet.” (Michael Jackson, 1988?)

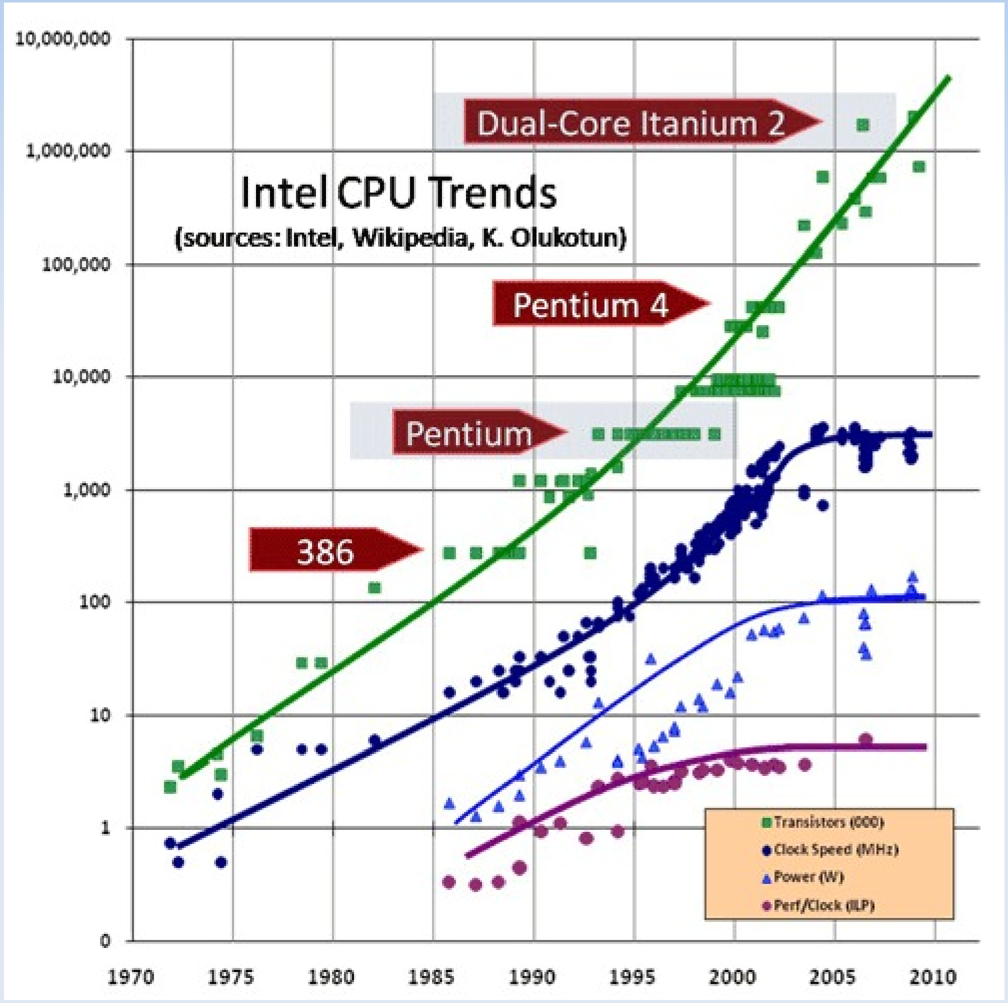
Nevertheless, optimisation did occur - especially at the design, build and compile level. Identification of critical locations in code (profiling) aids identification on what parts of source-code can be optimised.

Hardware Performance Meets Physics

Technological scaling from the 1960s to the early 2000s followed the propositions of Moore's Law (transistor density), Dennard Scaling (voltage and current scale down), and Koomey's Law (computations per joule), and most importantly, processor clock frequency.

At that point it become obvious that hardware improvements was facing an encounter with basic physics. *Heat.*

Clock speed flattened from the early 2000s onwards, but transistor count could continue to increase by the adoption of multi-core and multi-processor systems. More recently, for *certain types* of parallel processing, the adoption of general purpose use for graphics processing units (GPUs)



Processing with Performance and Scale

The increase in data volume, velocity, and variety is well-studied, with a 23% increase in storage during the 1980s to 2000s (Hilbert, López, 2007). Expectation that global data volume will have increased by an order of magnitude (4.4 zettabytes to 44 zettabytes) from 2013 to 2020 (Reinsel et al 2017).

Storage issues are only part of the problem of the deluge of data. Datasets are stored for a purpose, which means that they must be manipulated or modified in some way, i.e. processed.

Distributed or loosely coupled computational systems, such as various grid computing architectures (e.g., SETI@home, folding@home) is not the solution for large datasets except in cases where the dataset can be broken up, in which case it is a large collection of smaller datasets.

Couple the problem with large datasets, increasingly complex problems, and the bottlenecks in processing capacity per processor and it becomes very clear on why more tasks are moving to high performance computing.

Programming with Performance and Scale

To process data with performance and scale one needs hardware which can operate with performance and scale and software that is written with performance and scale in mind. The hardware side is covered by high performance compute clusters, and schedulers. The software side requires making the right programming choices (e.g., what sort of problem am I trying to solve?, c.f., John Gustafson) and adopting the right tools and the right techniques.

Using an example from Charles Leiserson's MIT's course "Performance Engineering of Software Systems" a square matrix multiplication, where C_ij = Sum A_ik*B_jk, assume n=2^k, using AWS c4.8x Large Machines (Intel Xeon E5-2666 v3, 2 processors, 9 cores per processor, 2.9GHz), peak 836 GFLOPS. The Python example (nested loops) takes 21,042 seconds; Java takes 2,738 seconds (i.e., 8.8x faster than Python), and CLang takes 1,156 seconds (i.e., 18x faster than Python). Why? An interpreted language versus a byte-code with JiT compilation, versus compilation to machine-code.

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

UniMelb's HPC System: Spartan

A detailed review was conducted in 2016 looking at the infrastructure of the Melbourne Research Cloud, High Performance Computing, and Research Data Storage Services. University desired a 'more unified experience to access compute services'

Recommended solution, based on technology and usage, is to make use of existing NeCTAR Research cloud with an expansion of general cloud compute provisioning and use of a smaller "true HPC" system on bare metal nodes.

Heterogeneous hardware:
Physical/BigMem partition: 28 nodes, 1204 cores, more powerful processors, more memory faster interconnect
Cloud/Longcloud partition: 165 nodes, 1980 cores, less powerful processors, less memory, slower interconnect
GPGPU partition: 73 nodes, 1752 cores, 292 P100 GPUs
Several specialist and project specific partitions.
Full range available at https://dashboard.hpc.unimelb.edu.au

Spartan is ~~Small but~~ Important

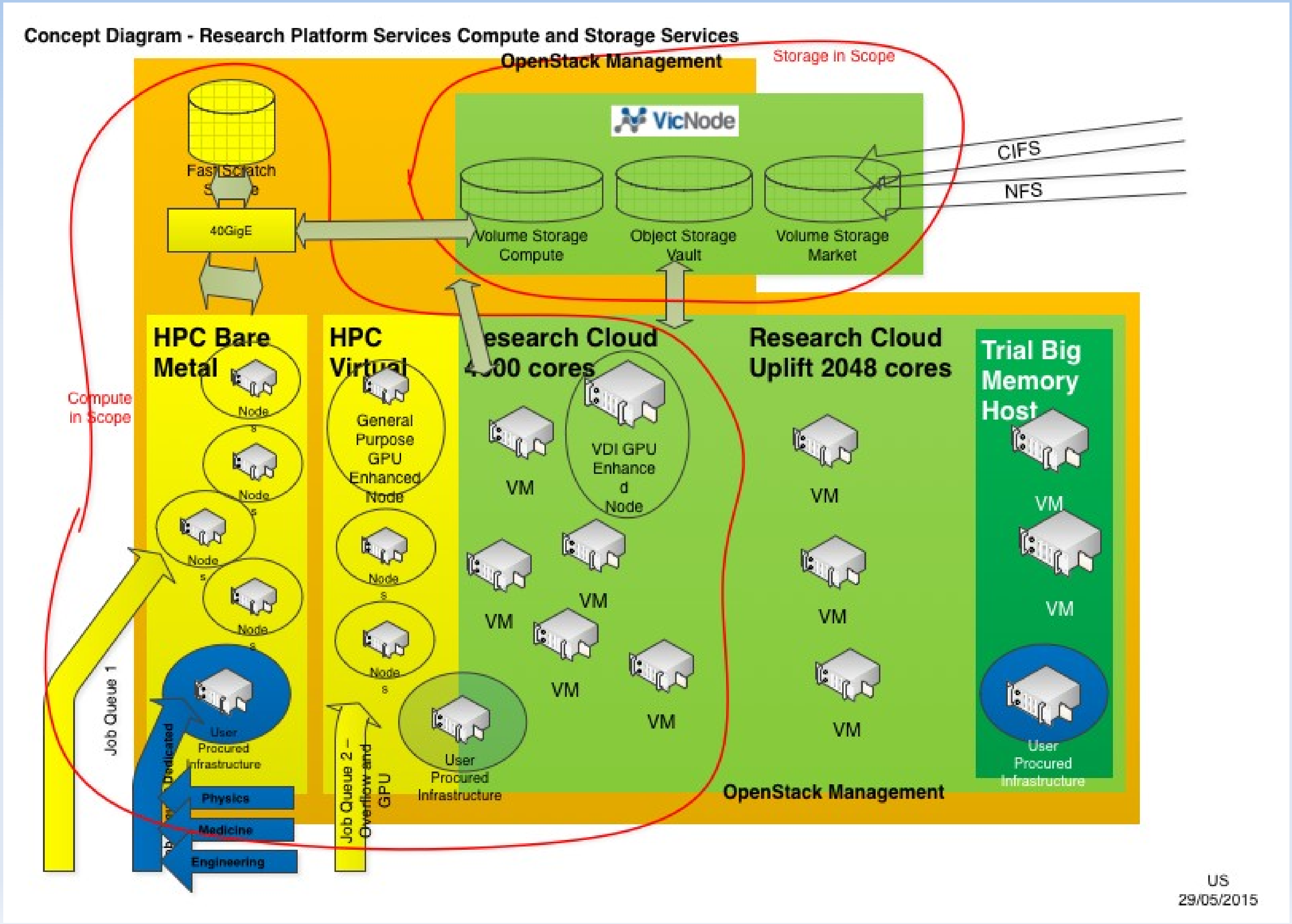
Spartan as a model of an HPC-Cloud Hybrid has been featured at Multicore World, Wellington, 2016, 2017; eResearchAustralasia 2016, Center for Scientific Computing (CSC) Goethe University Frankfurt, 2016, High Performance Computing Center (HLRS) University of Stuttgart, 2016, High Performance Computing Centre Albert-Ludwigs-University Freiburg, 2016; European Organization for Nuclear Research (CERN), 2016, Centre Informatique National de l'Enseignement Supérieur, Montpellier, 2016; Centro Nacional de Supercomputación, Barcelona, 2016, and the OpenStack Summit, Barcelona 2016.
https://www.youtube.com/watch?v=6D1lobuCZqE

Also featured in OpenStack and HPC Workload Management in Stig Telfer (ed), The Crossroads of Cloud and HPC: OpenStack for Scientific Research, Open Stack, 2016
http://openstack.org/assets/science/OpenStack-CloudandHPC6x9Booklet-v4-online.pdf

Architecture also featured in:
Spartan and NEMO: Two HPC-Cloud Hybrid Implementations. 2017 IEEE 13th International Conference on e-Science, DOI: 10.1109/eScience.2017.70

The Chimera and the Cyborg, Hybrid Compute Advances in Science, Technology and Engineering Systems Journal, Vol. 4, No. 2, 01-07, 2019

The Original Architecture



Spartan's Early Performance Tests

Service	**Network Device**	**Network**	**Protocol**	**Latency (usecs)**
UoM HPC Traditional	Mellanox	56Gb	Infiniband FDR	1.17
Legacy Edward HPC	Cisco Nexus	10Gbe	TCP/IP	19
Spartan Cloud nodes	Cisco Nexus	10Gbe	TCP/IP	60
Spartan Bare Metal	Mellanox	40Gbe	TCP/IP	6.85
Spartan Bare Metal	Mellanox	25Gbe	RDMA Ethernet	1.84
Spartan Bare Metal	Mellanox	40Gbe	RDMA Ethernet	1.15
Spartan Bare Metal	Mellanox	56Gbe	RDMA Ethernet	1.68
Spartan Bare Metal	Mellanox	100Gbe	RDMA Ethernet	1.3

Job	**Task**	**Resources**	**Control HPC**	**Spartan Cloud**	
BWA	Disk	8 core Single Node	1:18:49	1:02:56	1:40:21
GROMACS	Compute	128 core Multinode	0:30:02	0:30:10	0:30:32
NAMD	Compute, I/O	128 core Multinode	1:11:41	1:00:46	1:55:54

These are figures from 2016; recently we've had a significant upgrade to network and processors - about 25% improvement.

Setting Up An Account and Training

Spartan uses its own authentication that is tied to the university Security Assertion Markup Language (SAML). The login URL is `https://dashboard.hpc.unimelb.edu.au/karaage`

Users on Spartan must belong to a project. Projects must be led by a University of Melbourne researcher (the "Principal Investigator") and are subject to approval by the Head of Research Compute Services.

Participants in a project can be researchers or research support staff from anywhere.

The University, through Research Platforms, has an extensive training programme for researchers who wish to use Spartan. This includes day-long courses in “Introduction to Linux and HPC Using Spartan”, “Advanced Linux and Shell Scripting for High Performance Computing”, “Parallel Programming On Spartan”, “GPU Programming with OpenACC and CUDA”, and “Regular Expressions Using Linux”.

University of Melbourne is a key contributor to the International HPC Certification Program.

Logging In and Help

To log on to a HPC system, you will need a user account and password and a Secure Shell (ssh) client. Most HPC cluster administrators do not allow connections with protocols such as Telnet, FTP or RSH as they insecurely send passwords in plain-text over the network, which is easily captured by packet analyser tools (e.g., Wireshark). Linux distributions almost always include SSH as part of the default installation as does Mac OS 10.x, although you may also wish to use the Fugu SSH client. For MS-Windows users, the free PuTTY client is recommended. To transfer files use scp, WinSCP, Filezilla, and especially rsync.

Logins to Spartan are based on POSIX identity for the system

`ssh your-username@spartan.hpc.unimelb.edu.au` or

For help go to http://dashboard.hpc.unimelb.edu.au or check man spartan. Lots of example scripts at /usr/local/common

Need more help? Problems with submitting a job, need a new application or extension to an existing application installed, if job generated unexpected errors etc., an email can be sent to: hpc-support@unimelb.edu.au

Don't email individual sysadmins. Provide as much information as possible (e.g., location of job script, error message etc)

Things You Should Not Do

Do not run jobs on the login node. This is a shared location and if you use up the cores, memory or other resources there, you will make life miserable for everyone. Use the batch system instead. This lecture will show you how!

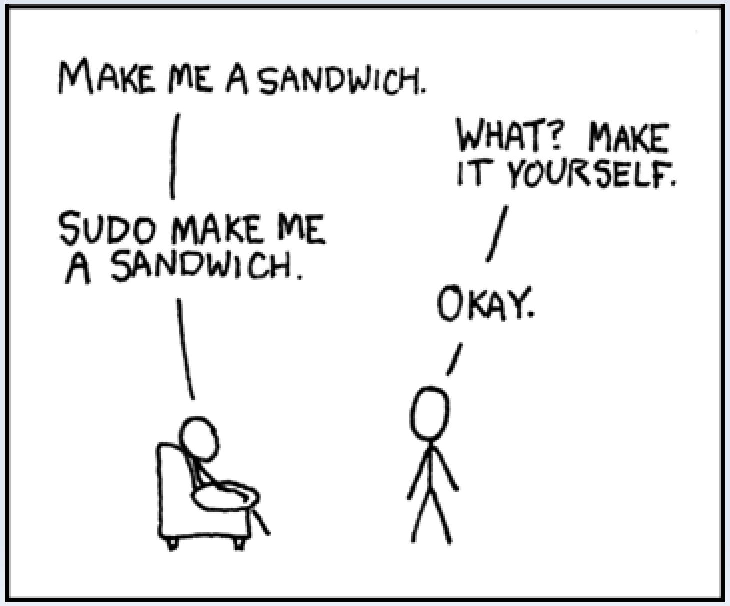
If you need to do compute-memory tasks with immediate feedback, run an interactive job. This lecture will show you how!

Do not use sudo to run commands. Giving users root access on a shared system would be a security nightmare! You don't have sudo rights and you'll get a stern email (probably from me) if you try.

Do not try to install software with apt, yum or some other package manager. It will not work. We typically compile software from source and add extensions to particular versions.

Exception: we do allow some local extensions to be installed for Python, R, Perl etc. Check the individual application information on Spartan for this.

Exception: we do allow users to compiler their own software for specific projects which is usually stored in their project directory.



The Linux Environment and Modules

Assumption here is that everyone has had exposure to the Linux command line. If not, you'd better get some! At least learn the twenty or so basic environment commands to navigate the environment, manipulate files, manage processes. Plenty of good online material available (e.g., my book "Supercomputing with Linux", https://github.com/VPAC/superlinux)

Environment modules provide for the dynamic modification of the user's environment (e.g., paths) via module files. Each module contains the necessary configuration information for the user's session to operate according according to the modules loaded, such as the location of the application's executables, its manual path, the library path, and so forth.

Modulefiles also have the advantages of being shared with many users on a system and easily allowing multiple installations of the same application but with different versions and compilation options. Sometimes users want the latest and greatest of a particular version of an application for the feature-set they offer. In other cases, such as someone who is participating in a research project, a consistent version of an application is desired. In both cases consistency and therefore *reproducibility* is attained.

Having multiple version of applications available on a system is essential in research computing!

Modules Commands

Some basic module commands include the following:

```
module help
```
The command module help , by itself, provides a list of the switches, subcommands, and subcommand arguments that are available through the environment modules package.

```
module avail
```
This option lists all the modules which are available to be loaded.

```
module whatis <modulefile>
```
This option provides a description of the module listed.

```
module display <modulefile>
```
Use this command to see exactly what a given modulefile will do to your environment, such as what will be added to the PATH, MANPATH, etc. environment variables.

More Modules Commands

`module load <modulefile>`
This adds one or more modulefiles to the user's current environment (some modulefiles load other modulefiles.

module unload <modulefile>
This removes any listed modules from the user's current environment.

module switch <modulefile1> <modulefile2>
This unloads one modulefile (modulefile1) and loads another (modulefile2).

module purge
This removes all modules from the user's environment.

In the lmod system as used on Spartan there is also “module spider” which will search for all possible modules and not just those in the existing module path and provide descriptions.

(Image from NASA, Apollo 9 “spider module”)

Batch Systems and Workload Managers

The Portable Batch System (or simply PBS) is a utility software that performs job scheduling by assigning unattended background tasks expressed as batch jobs among the available resources.

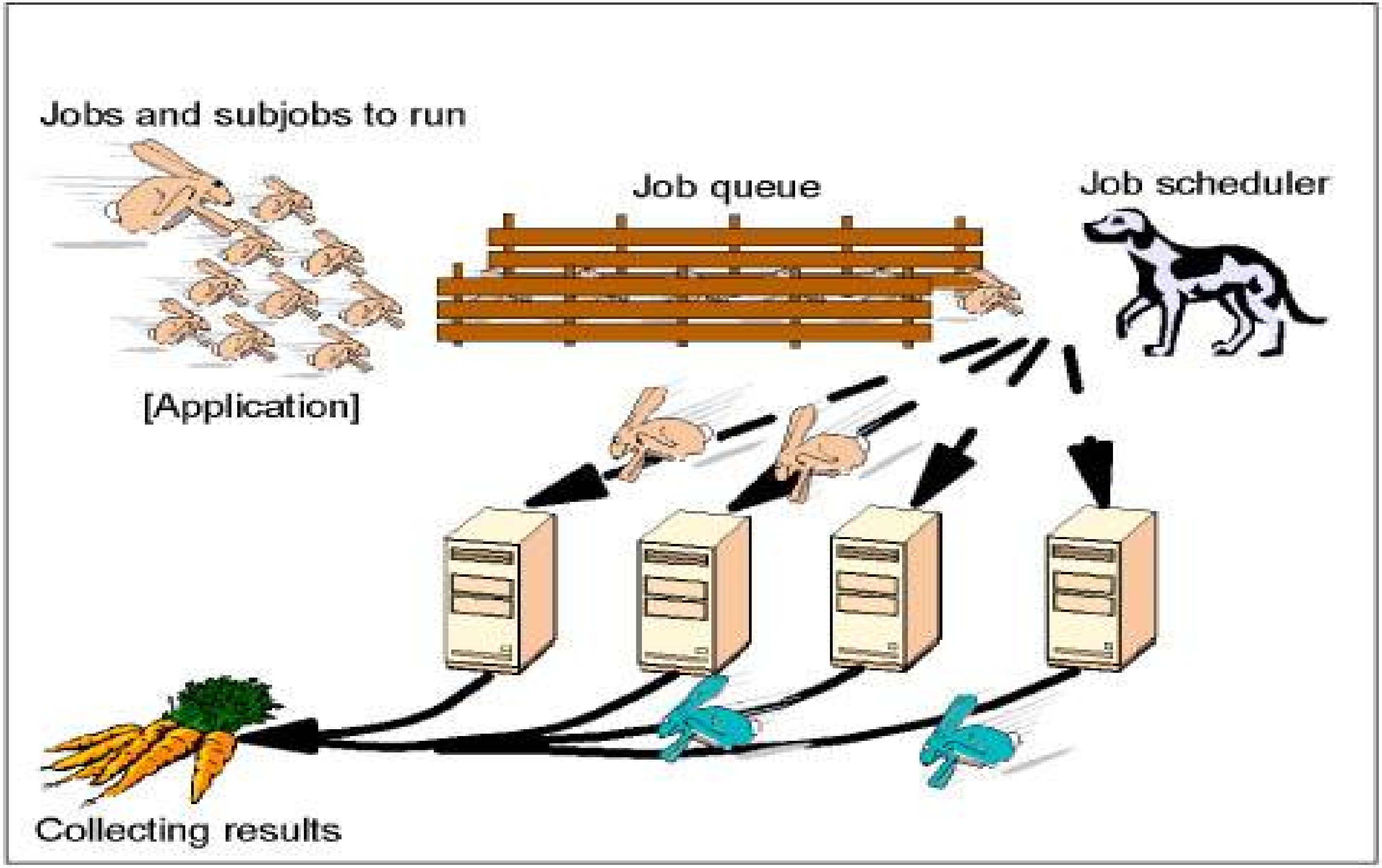
The original Portable Batch System was developed by MRJ Technology Solutions under contract to NASA in the early 1990s. In 1998 the original version of PBS was released as an open-source product as OpenPBS. This was forked by Adaptive Computing (formally, Cluster Resources) who developed TORQUE (Terascale Open-source Resource and QUEue Manager). Many of the original engineering team is now part of Altair Engineering who have their own version, PBSPro.

In addition to this the popular job scheduler Slurm (originally “Simple Linux Utility for Resource Management”, SLURM), now simply called Slurm Workload Manager, also uses batch script where are very similar in intent and style to PBS scripts.

Spartan uses the Slurm Workload Manager. A job script written on one needs to be translated to another (handy script available pbs2slurm https://github.com/bjpop/pbs2slurm)

In addition to this variety of implementations of PBS different institutions may also make further elaborations and specifications to their submission filters (e.g., site-specific queues, user projects for accounting). (Image from the otherwise dry IBM 'Red Book' on Queue Management)

Submitting and Running Jobs



Submitting and Running Jobs

Submitting and running jobs is a relatively straight-forward process consisting of:

1) Setup and launch
2) Job Control, Monitor results
3) Retrieve results and analyse.

Don't run jobs on the login node! Use the queuing system to submit jobs.

1. Setup and launch consists of writing a short script that initially makes resource requests and then commands, and optionally checking queueing system.

Core command for checking queue:	squeue | less
Alternative command for checking queue:	showq -p cloud | less
Core command for job submission:	sbatch [jobscript]

2. Check job status (by ID or user), cancel job.

Core command for checking job in Slurm:	squeue -j [jobid]
Detailed command in Slurm:	scontrol show job [jobid]
Core command for deleting job in Slurm:	scancel [jobid]

3.Slurm provides an error and output files They may also have files for post-job processing. Graphic visualisation is best done on the desktop.

Simple Script Example

```
#!/bin/bash
#SBATCH --partition=cloud
#SBATCH --time=01:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
module load my-app-compiler/version
my-app data
```

The script first invokes a shell environment, followed by the partition the job will run on (the default is 'cloud' for Spartan). The next four lines are resource requests, specifically for one compute node, one task.

After these requests are allocated, the script loads a module and then runs the executable against the dataset specified. Slurm also automatically exports your environment variables when you launch your job, including the directory where you launched the job from. If your data is a different location this has to be specified in the path!

After the script is written it can be submitted to the scheduler. See examples in /usr/local/common/IntroLinux

```
[lev@spartan]$ sbatch myfirstjob.slurm
```

Multi-threaded, Multi-core, and Multi-node Examples

Modifying resource allocation requests can improve job efficiency if the applications has been designed to take advantage of this. The scheduler simply allocates resources, it does not make your application parallel!

For example shared-memory multithreaded jobs on Spartan (e.g., OpenMP), modify the --cpus-per-task to a maximum of 8, which is the maximum number of cores on a single instance.

```
#SBATCH --cpus-per-task=8
```

For distributed-memory multicore job using message passing, the multinode partition has to be invoked and the resource requests altered e.g.,

```
#!/bin/bash
#SBATCH -partition=physical
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
module load my-app-compiler/version
srun my-mpi-app
```

Note that multithreaded jobs *cannot* be used in a distributed memory model across nodes. They can however exist be conducted on distributed memory jobs which include a shared memory component (hybrid OpenMP-MPI jobs).

Arrays : Many Jobs, One Script

Job arrays allow the same batch script, and therefore the same resource requests, and launches multiple jobs simultaneously. A typical example is to apply the same task across multiple datasets

The array directive sets the index values which can be a range (e.g., 0-31), comma-separated values, (e.g., 1,3,9,11) or a range with a step value (e.g., 1-7:2)

The following example submits 10 batch jobs with myapp running against datasets dataset1.csv, dataset2.csv, ... dataset10.csv

```
#SBATCH --array=1-10
myapp ${SLURM_ARRAY_TASK_ID}.csv
```

Examples are in /usr/local/common/array and /usr/local/common/Octave

When a job array is launched each subjob gets its own job number that is a subset of the main job e.g., 14950773_1, 14950773_2, 14950773_3 etc. These can be managed by the normal schedule commands (e.g., scancel 14950773_3)

Dependencies : Job Pipelines

A dependency condition is established on which the launching of a batch script depends, creating a conditional pipeline. A typical use case is where the output of one job is required as the input of the next job.

```
#SBATCH --dependency=afterok:myfirstjobid mysecondjob
```

Several conditional directives to be placed on a job which are tested prior to the job being intiatied, which are summarised as after, afterany, afterok, afternotok, singleton (e.g., sbatch --dependency=afterok:jobid1 job2.slurm). Multiple jobs can be listed as dependencies with colon separated values (e.g., `sbatch --dependency=afterok:jobid1:jobid2 job3.slurm`).

Some dependency types (not these differ from those used PBSPro and TORQUE)

after:jobid[:jobid...]	job can begin after the specified jobs have started
afterany:jobid[:jobid...]	job can begin after the specified jobs have terminated
afternotok:jobid[:jobid...]	job can begin after the specified jobs have failed
afterok:jobid[:jobid...]	job can begin after the specified jobs have run to completion with an exit code of zero (see the user guide for caveats).
singleton	job can begin execution after all previously launched jobs with the same name and user have ended.

See example in /usr/local/common/depend

Interactive Jobs

For real-time interaction, with resource requests made on the command line, an interactive job is called. This puts the user on to a compute node.

This is typically done if they user wants to run a large script (and shouldn't do it on the login node), or wants to test or debug a job. The following command would launch one node with two processors for ten minutes.

Once an interactive job is launched, the user can conduct whatever computationally intensive task they wish with the resources that they have allocated. For example, a multi-threaded application, tested in normal and multi-threaded computation.

```
[lev@spartan interact]$ sinteractive --ntasks=2 --cpus-per-task=2
srun: job 15489392 queued and waiting for resources
srun: job 15489392 has been allocated resources
[lev@spartan-rc002 interact]$ gcc iterate.c -o iterate
[lev@spartan-rc002 interact]$ time ./iterate
..
[lev@spartan-rc002 interact]$ export OMP_NUM_THREADS=2
[lev@spartan-rc002 interact]$ gcc -fopenmp iterate.c -o iterate
[lev@spartan-rc002 interact]$ time ./iterate
```

X-Windows Forwarding

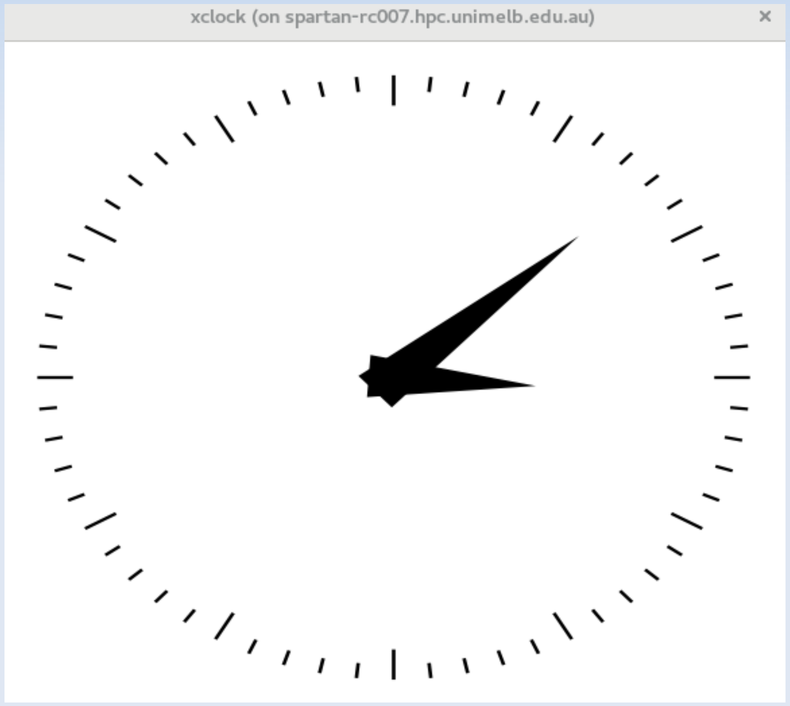
In almost all cases it is much better to do computation on the cluster and visualisation on a local system. In some cases however it is unavoidable to require x-windows forwarding.

It is best to login with the -X option for security and then to login with -X again to the login node. The compute node with then pass through the graphics via the login node to the desktop system.

Please note that you will need an x-windows client on your desktop for the visualisation.

```
[lev@cricetomys ~]$ ssh
lev@spartan.hpc.unimelb.edu.au -X

[lev@spartan]$ sinteractive --nodes=1 --ntasks-per-
node=2 -X
srun: job 602795 queued and waiting for resources
srun: job 602795 has been allocated resources
[lev@spartan-rc002 ~]$ xclock
```



Job Scripts are Shell Scripts

Because the script first invokes a shell environment all job scripts are shell scripts. This means that any of the tools and techniques used in bash shell programming can also be used in job submission scripts, which is very handy managing files and directories within the environment.

It also means that the scheduler directives (#SBATCH) are interpreted as comments by the shell environment, but are understood by the Slurm scheduler.

It also means that scheduler directives must come first! Do all your scheduler directives, then include your shell commands (including loading your modules)

This includes assigning variables, shell substitutions, loops, conditional statements, case statements, shell functions, heredocs and so forth.

There are are number of shell scripting examples in /usr/local/common/HPCshells, and example of a Slurm job with extensive shell commands in /usr/local/common/HPCshells/NAMD and the use of a heredoc and a loop in /usr/local/common/HPCshells/Gaussian

Job Staging for Network Performance

Where you are on the system matters for job performance; this applies to all systems, and HPC systems are no different - and it is a well known problem. See Grace Hopper, “Mind Your Nanoseconds” (https://www.youtube.com/watch?v=9eyFDBPk4Yw)

For small test jobs (such as the examples gives in this lecture) launching with a dataset in one's home directory or project directory is sufficient. For larger datasets, one should use the /scratch or a local disk directory which have faster disk, interconnect and/or shorter distance.

Sorry! There is no /scratch directory for this class!

In general:

/home, /data/projects/$projectID is slower.
/scratch/$projectID is faster
/var/local/tmp is the fastest

But /var/local/tmp is the local disk of the compute node! So you must copy the results back to one's project directory (for example) when as part of the job script.

PBS, SLURM Comparison

User Commands	**PBS/Torque**	**SLURM**
Job submission	qsub [script_file]	sbatch [script_file]
Job submission	qdel [job_id]	scancel [job_id]
Job status (by job)	qstat [job_id]	squeue [job_id]
Job status (by user)	qstat -u [user_name]	squeue -u [user_name]
Node list	pbsnodes -a	sinfo -N
Queue list	qstat -Q	squeue
Cluster status	showq, qstatus -a	squeue -p [partition]
Environment		
Job ID	$PBS_JOBID	$SLURM_JOBID
Submit Directory	$PBS_O_WORKDIR	$SLURM_SUBMIT_DIR
Submit Host	$PBS_O_HOST	$SLURM_SUBMIT_HOST
Node List	$PBS_NODEFILE	$SLURM_JOB_NODELIST
Job Array Index	$PBS_ARRAYID	$SLURM_ARRAY_TASK_ID

PBS and SLURM Comparison

Job Specification	**PBS**	**SLURM**
Script directive	#PBS	#SBATCH
Queue	-q [queue]	-p [queue]
Job Name	-N [name]	--job-name=[name]
Nodes	-l nodes=[count]	-N [min[-max]]
CPU Count	-l ppn=[count]	-n [count]
Wall Clock Limit	-l walltime=[hh:mm:ss]	-t [days-hh:mm:ss]
Event Address	-M [address]	--mail-user=[address]
Event Notification	-m abe	--mail-type=[events]
Memory Size	-l mem=[MB]	--mem=[mem][M|G|T]
Proc Memory Size	-l pmem=[MB]	--mem-per-cpu=[mem][M|G|T]

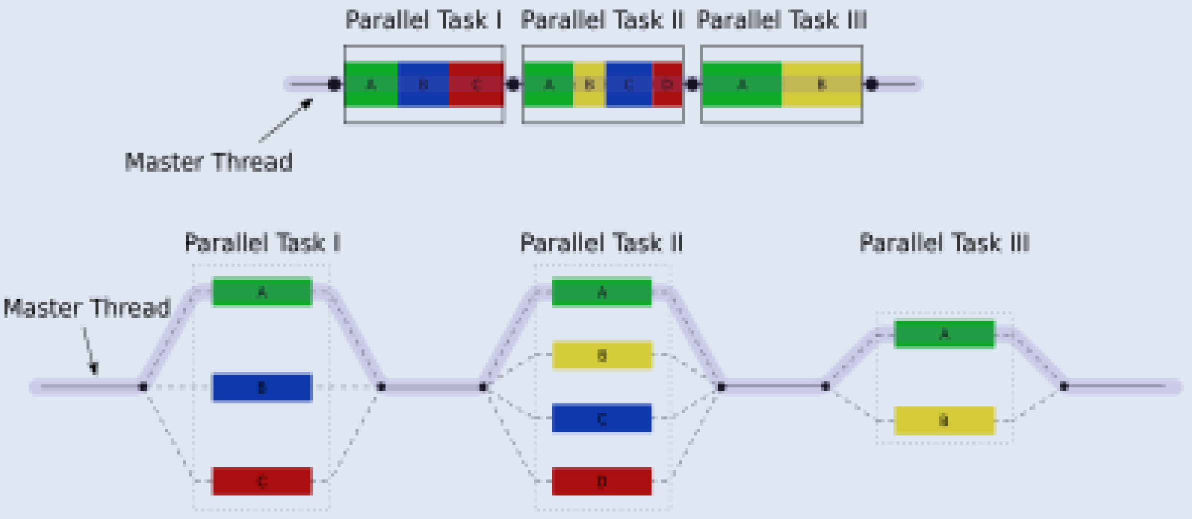
Shared Memory Parallel Programming

One form of parallel programming is multithreading, whereby a master thread forks a number of sub-threads and divides tasks between them. The threads will then run concurrently and are then joined at a subsequent point to resume normal serial application.

One implementation of multithreading is OpenMP (Open Multi-Processing). It is an Application Program Interface that includes directives for multi-threaded, shared memory parallel programming. The directives are included in the C or Fortran source code and in a system where OpenMP is not implemented, they would be interpreted as comments.

There is no doubt that OpenMP is an easier form of parallel programming, however it is limited to a single system unit (no distributed memory) and is thread-based rather than using message passing. Many examples in `/usr/local/common/OpenMP`.

(image from: User A1, Wikipedia)



Shared Memory Parallel Programming

```
#include <stdio.h>
#include  "omp.h"
int main(void)
{
   int id;
   #pragma omp parallel num_threads(8) private(id)
   {
   int id = omp_get_thread_num();
   printf("Hello world %d\n", id);
   }
return 0;
}
```

```
program hello2omp
   include "omp_lib.h"
   integer :: id
   !$omp parallel num_threads(8) private(id)
     id = omp_get_thread_num()
          print *, "Hello world", id
   !$omp end parallel
end program hello2omp
```

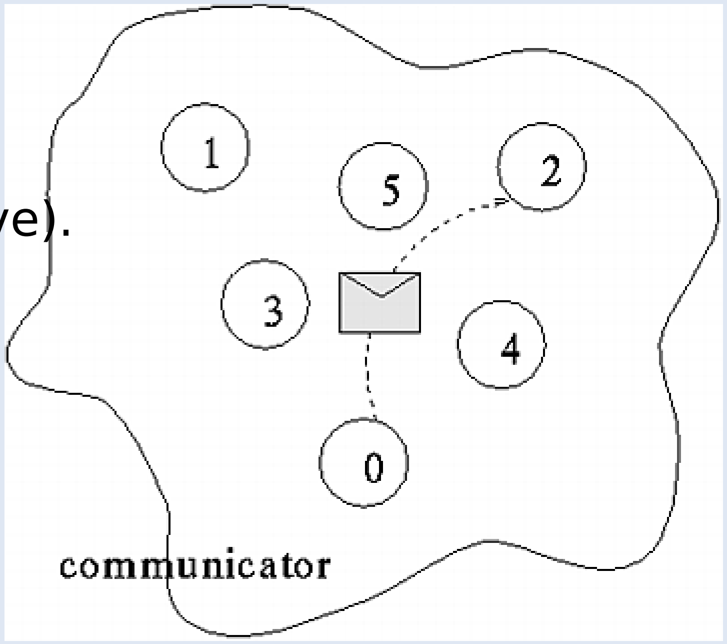
Distributed Memory Parallel Programming

Moving from shared memory to parallel programming involves a conceptual change from multi-threaded programming to a message passing paradigm. In this case, MPI (Message Passing Interface) is one of the most well popular standards and is used here, along with a popular implementation as OpenMPI.

The core principle is that many processors should be able cooperate to solve a problem by passing messages to each through a common communications network.

The flexible architecture does overcome serial bottlenecks, but it also does require explicit programmer effort (the "questing beast" of automatic parallelisation remains somewhat elusive).

The programmer is responsible for identifying opportunities for parallelism and implementing algorithms for parallelisation using MPI.

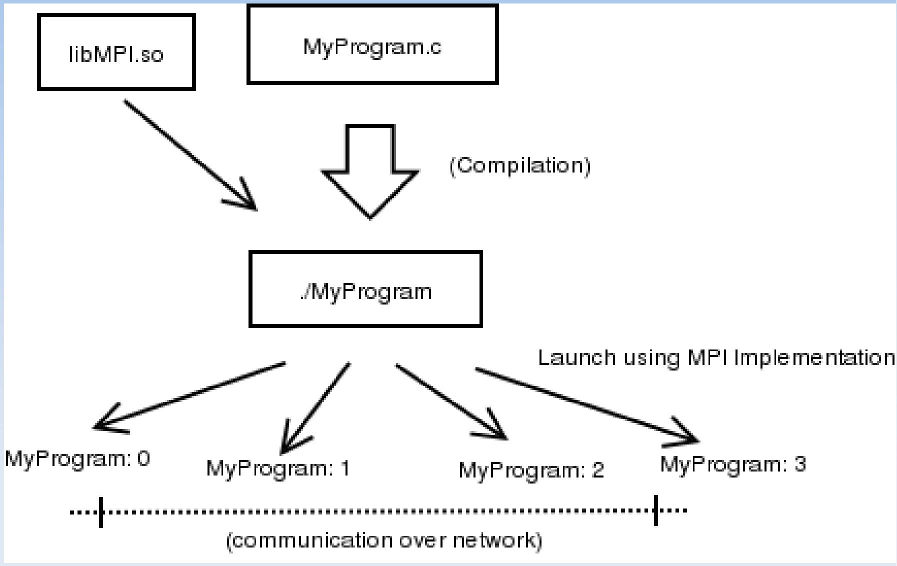


Distributed Memory Parallel Programming

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int  argc;
char **argv;
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}

!       Fortran MPI Hello World
        program hello
        include 'mpif.h'
        integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)

        call MPI_INIT(ierror)
        call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
        print*, 'node', rank, ': Hello world'
        call MPI_FINALIZE(ierror)
        end
```



MPI Compilation and Job Scripts

The OpenMP example needs to be compiled with OpenMP directives. The OpenMP example cannot run across compute nodes; therefore it is best run on the “cloud” partition. The OpenMPI compilation needs to call the MPI wrappers.

```
module load OpenMPI/1.10.0-GCC-4.9.2
gcc -fopenmp helloomp.c -o helloompc
mpigcc mpihelloworld.c -o mpihelloworld
```

```
#!/bin/bash
#SBATCH -p cloud
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=16
export OMP_NUM_THREADS=16
module load GCC/4.9.2
mpiexec helloompc
```

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --nodes=2
#SBATCH --ntasks=16
module load OpenMPI/1.10.2-GCC-4.9.2
mpiexec mpi-helloworld
```

MPJ Express for Java

Java can be compiled with MPI bindings; we have done this with OpenMPI/3.0.0 only. A more common option is to use MPJ-Express. The following “HellowWorld.java” program is compiled and executed. See /usr/local/common/Java

```
import mpi.*;
public class HelloWorld {
public static void main(String args[]) throws Exception {
        MPI.Init(args);
        int me = MPI.COMM_WORLD.Rank();
        int size = MPI.COMM_WORLD.Size();
        System.out.println("Hi from <"+me+">");
        MPI.Finalize();
        }
}
```

```
sinteractive --time=1:00:00 --nodes=1 --ntasks=2
module load MPJ-Express
javac -cp .:/usr/local/easybuild/software/MPJ-Express/0.44-goolf-
2015a-Java-9.0.4/lib/mpj.jar HelloWorld.java
mpjrun.sh -np 2 HelloWorld
```

MPI4Py Express for Python

Python too has various MPI bindings available. The most common used is MPI4Py. Sample assignment data is provided from the 2016 and 2015 in the Spartan directory, `/usr/local/common`. As a package it can be simply imported (e.g., `from mpi4py import MPI`).

But remember! With environment modules with extensions you do not necessarily get all the packages/libraries/extensions that you might expect. See the README file for an explanation of how to review the extensions already installed.

Examples provided in the directory of various Python jobs with MPI bindings with Slurm submissions scripts.

Usually we have examples for for single-core, dual core, eight-core, sixteen-core (different partition), and two-nodes with four cores each.

But that's for your assignment!

MPI Communication: A Game of Ping-Pong

A very popular and basic use of MPI Send and Recv routines is a ping-ping program. Why? Because it can be used to test latency within and between nodes and partitions if they have different interconnect (like on Spartan).

An example is given in `/usr/local/common/OpenMPI` as `mpi-pingpong.c` with a job submission script that can be modified accord to the test case `mpi-pingpong.slurm`. There are some routines here which manage the communication in the ping-pong activity.

`MPI_Status()` MPI_Status is not a routine, but rather a data structure and is typically attached to an MPI_Recv() routine.

`MPI_Request()` A wrapper for MPI Requests such as wait, waitany, waitall, waitsome, start, cancel, startall.

`MPI_Barrier()` Enforces synchronisation between MPI processes in a group by placing a barrier on communication between groups. An MPI barrier completes after all group members have entered the barrier.

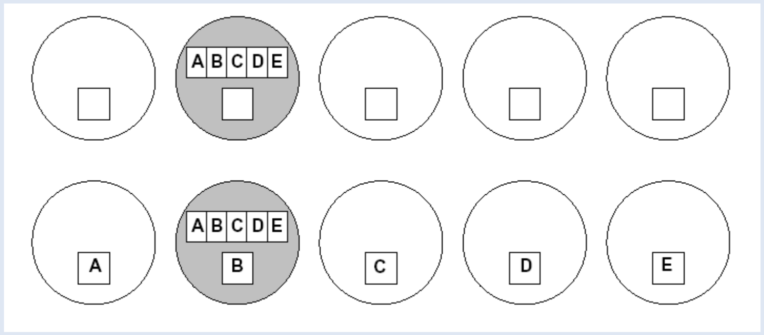
`MPI_Wtime()` - Returns an elapsed time as a floating-point number of seconds on the calling processor from an arbitrary time in the past.

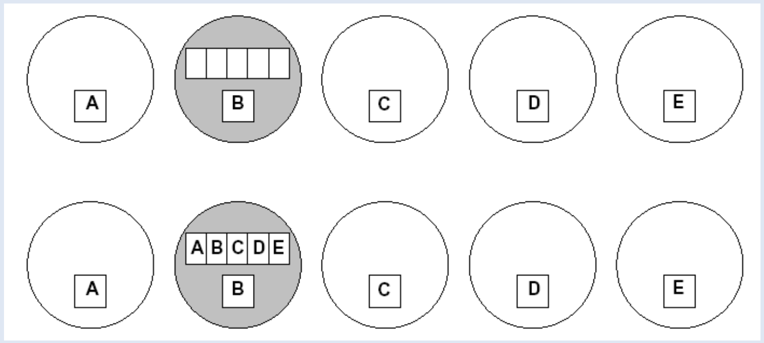
MPI Collective Communications

There are *many* other MPI routines which are *not* going to explored here! However just as a little taste one of the most popular is MPI collective communications and reduction operations. Collective communications include `MPI_Broadcast`, `MPI_Scatter`, `MPI_Gather`, `MPI_Reduce`, and `MPI_Allreduce`.

`MPI_Bcast` Broadcasts a message from the process with rank "root" to all other processes of the communicator, including itself. It is significantly more preferable than using a loop.

`MPI_Scatter` sends data from one task to all tasks in a group; the inverse operation of `MPI_Gather.` The outcome is as if the root executed n send operations and each process executed a receive. `MPI_Scatterv` scatters a buffer in parts to all tasks in a group.

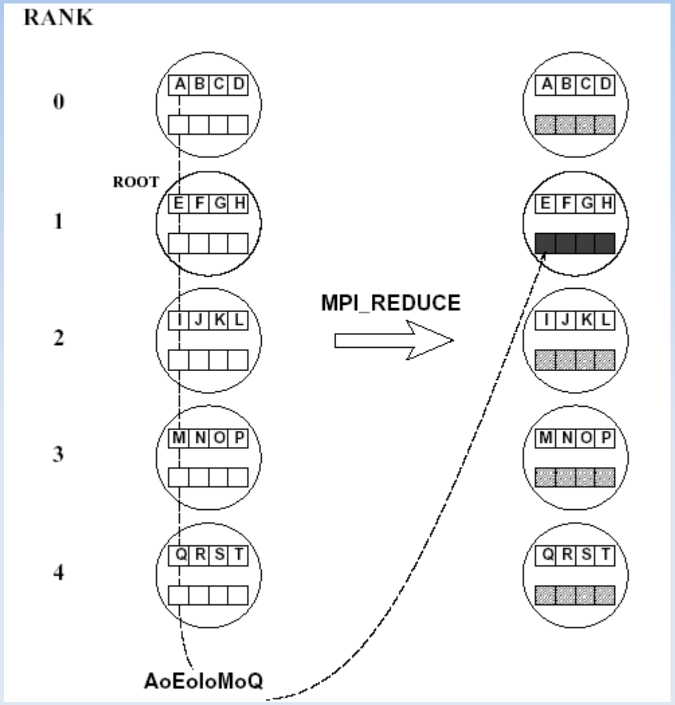


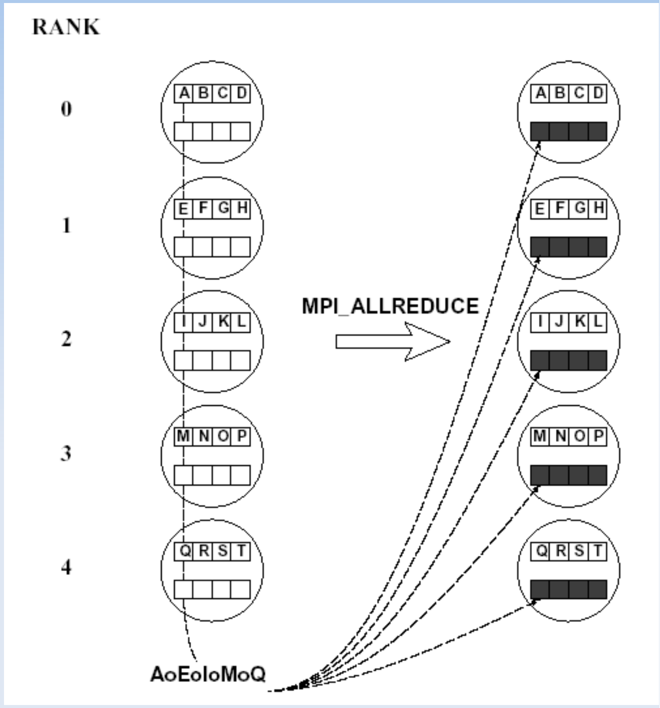


MPI Reduction Communications

`MPI_Reduce` performs a reduce operation (such as sum, max, logical AND, etc.) across all the members of a communication group.

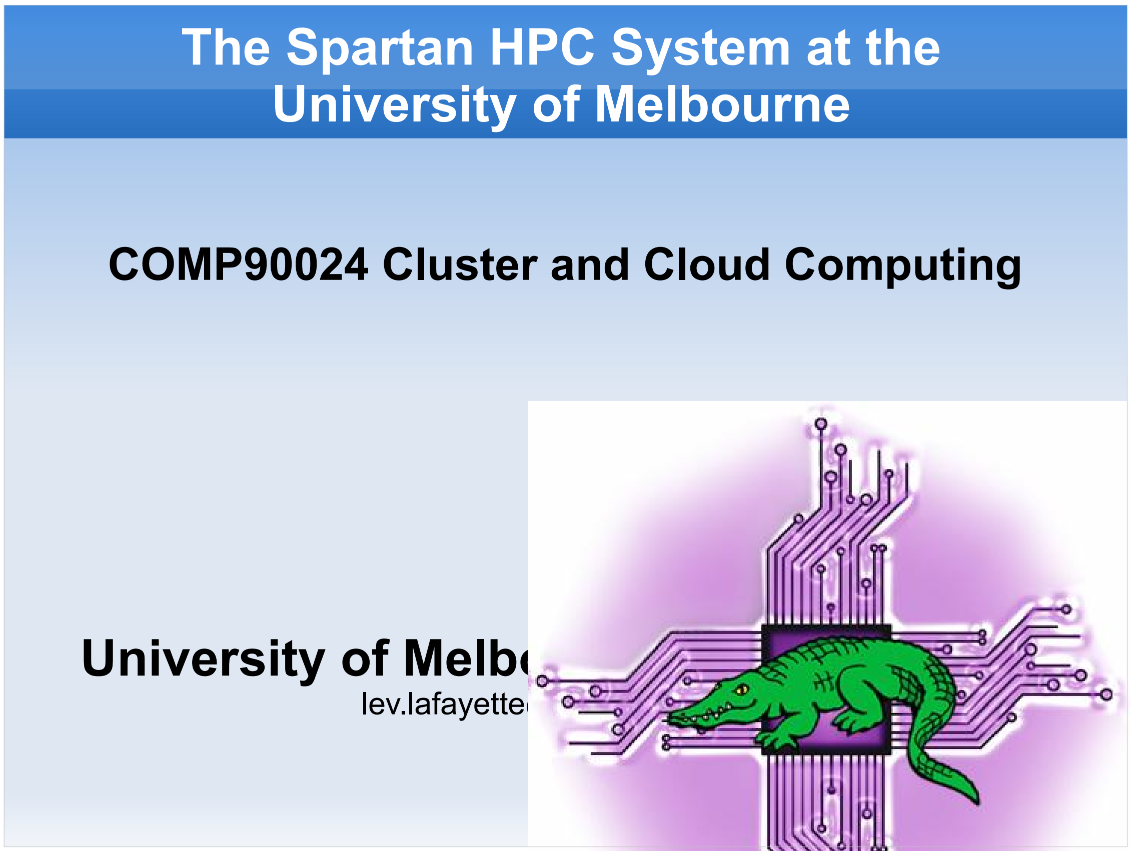
`MPI_Allreduce` conducts the same operation but returns the reduced result to all processors.





The general principle in Reduce and All Reduce is the idea of reducing a set of numbers to a small set via a function. If you have a set of numbers (e.g., [1,2,3,4,5]) a reduce function (e.g., sum) can convert that set to a reduced set (e.g., 15). MPI_Reduce takes in an array of values as that set and outputs the result to the root process. MPI_AllReduce outputs the result to all processes.

THANKS FOR WATCHING
& LISTENING PATIENTLY
makeameme.org

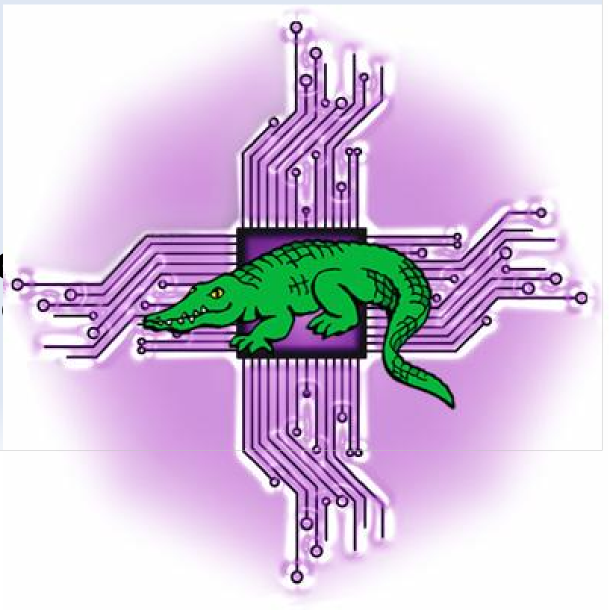


The Spartan HPC System at the University of Melbourne

COMP90024 Cluster and Cloud Computing

University of Melb

lev.lafayette



Who Is This Guy?

Currently Senior HPC DevOps Engineer at University of Melbourne since 2015

Has also worked for:
Victorian Partnership for Advanced Computing as HPC Systems Administrator and Quality Assurance Coordinator
Ministry of Foreign Affairs and Cooperation of Timor-Leste, Information and Communications Technology Consultant
Parliament of Victoria as an Electorate Officer, specialising in various database services etc.

Has done some teaching to some researchers around Australia, written a few books and few more journal articles. Have given a few presentations at various local, national, and international conferences.

Degrees in politics, philosophy, sociology, technology management, project management, adult and tertiary education, information systems etc.

http://levlafayette.com
https://www.linkedin.com/in/levlafayette/

Outline of Lecture

"This is an advanced course but we get mixed bag: students that have 5+ years of MPI programming on supercomputers, to students that have only done Java on Windows."

- Some background on supercomputing, high performance computing, parallel computing, research computing (they're not the same thing!).
- Why performance and scale matters, and why it should matter to *you*.
- An introduction to Spartan, University of Melbourne's HPC/cloud hybrid system
- Logging in, help, and environment modules.
- Job submission with Slurm workload manager; simple submissions, multi-core, multi-node, job arrays, job dependencies, interactive jobs.
- Parallel programming with shared memory and threads (OpenMP) and distributed memory and message passing (OpenMPI)
- Tantalising hints about more advanced material on message passing routines.

Supercomputers

'Supercomputer” arbitrary term. In general use it means any single computer system (itself a contested term) that has exceptional processing power for its time. One metric is the number of floating-point operations per second (FLOPS) such a system can carry out. The Top500 list is based on FLOPS using LINPACK - HPC Challenge is a broader, more interesting metric. The current number #1 system is Summit from the Oak Ridge National Laboratory in the United States.

1994:	170.40 GFLOPS
1996:	368.20 GFLOPS
1997:	1.338 TFLOPS
1999:	2.3796 TFLOPS
2000:	7.226 TFLOPS
2004;	70.72 TFLOPS
2005:	280.6 TFLOPS
2007:	478.2 TFLOPS
2008:	1.105 PFLOPS
2009:	1.759 PFLOPS
2010:	2.566 PFLOPS
2011:	10.51 PFLOPS
2012:	17.59 PFLOPS
2013:	33.86 PFLOPS
2014:	33.86 PFLOPS
2015:	33.86 PFLOPS
2016:	93.01 PFLOPS
2017:	93.01 PFLOPS
2018:	143.00 PFLOPS (200.795 Rpeak)
2019:	148.60 PFLOS (200,795 Rpeak)

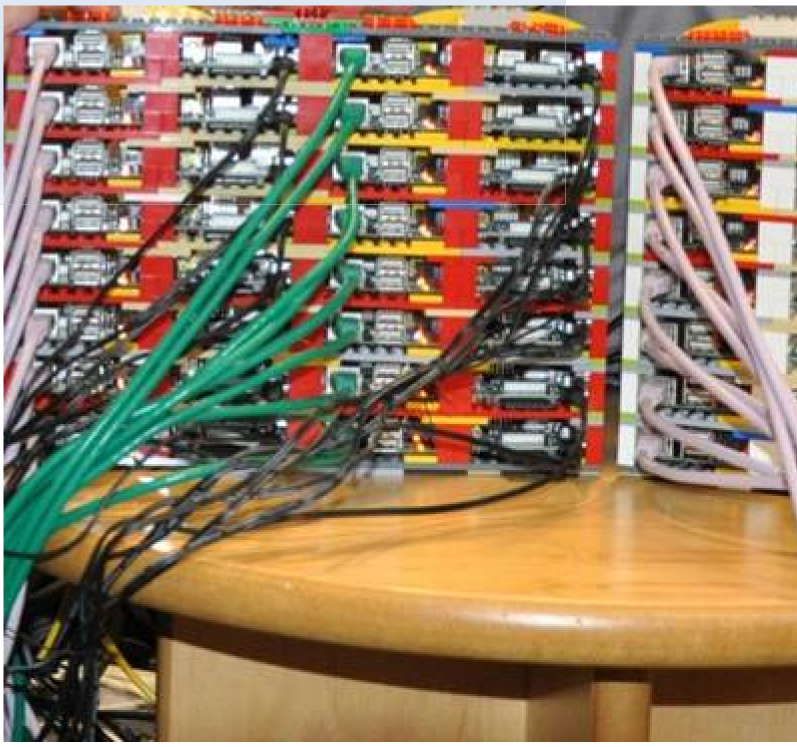
High Performance Computing

High-performance computing (HPC) is any computer system whose architecture allows for above average performance. A system that is one of the most powerful in the world, but is poorly designed, could be a "supercomputer".

Clustered computing is when two or more computers serve a single resource. This improves performance and provides redundancy; typically a collection of smaller computers strapped together with a high-speed local network (e.g., Myrinet, InfiniBand, 10 Gigabit Ethernet). Even a cluster of Raspberry Pi with a Lego chassis (University of Southampton, 2012)!

Horse and cart as a computer system and the load as the computing tasks. Efficient arrangement, bigger horse and cart, or a teamster?

The clustered HPC is the most efficient, economical, and scalable method, and for that reason it dominates supercomputing.

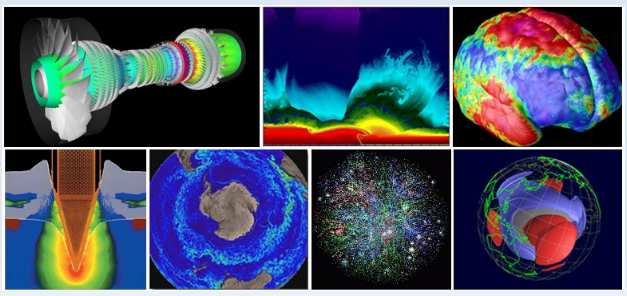


Parallel and Research Programming

With a cluster architecture, applications can be more easily parallelised across them. Parallel computing refers to the submission of jobs or processes over multiple processors and by splitting up the data or tasks between them (random number generation as data parallel, driving a vehicle as task parallel).

Research computing is the software applications used by a research community to aid research. This skills gap is a major problem and must be addressed because as the volume, velocity, and variety of datasets increases then researchers will need to be able to process this data.

Computational capacity does have a priority (the system must exist prior to use), in order for that capacity to realised in terms of usage a skill-set competence must also exist. The the core issue is that high performance compute clusters is just speed and power but also usage, productivity, correctness, and reproducibility.
(image from Lawrence Livermore National Laboratory)



There is nascent research that show a strong correlation between research output and availability of HPC facilities. (Apon et al 2010)

Some Local Examples

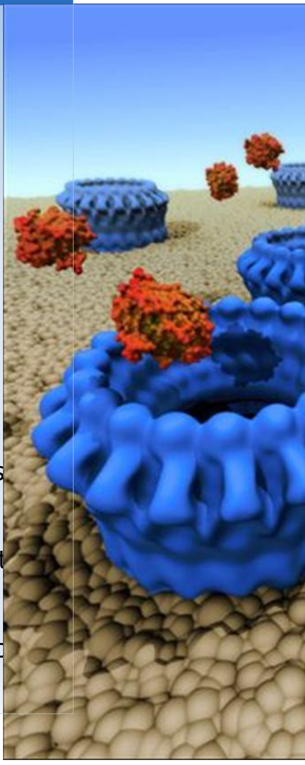
Researchers from Monash University, the Peter MacCallum Cancer Institute in Melbourne, the Birkbeck College in London, and VPAC in 2010 unravelled the structure the protein perforin to determine how pathogenic cells are attacked by white blood cells.

In 2015 researchers from VLSCI announced how natural antifreeze proteins bind to ice to prevent it growing which has important implications for extending donated organs and protecting crops from frost damage.

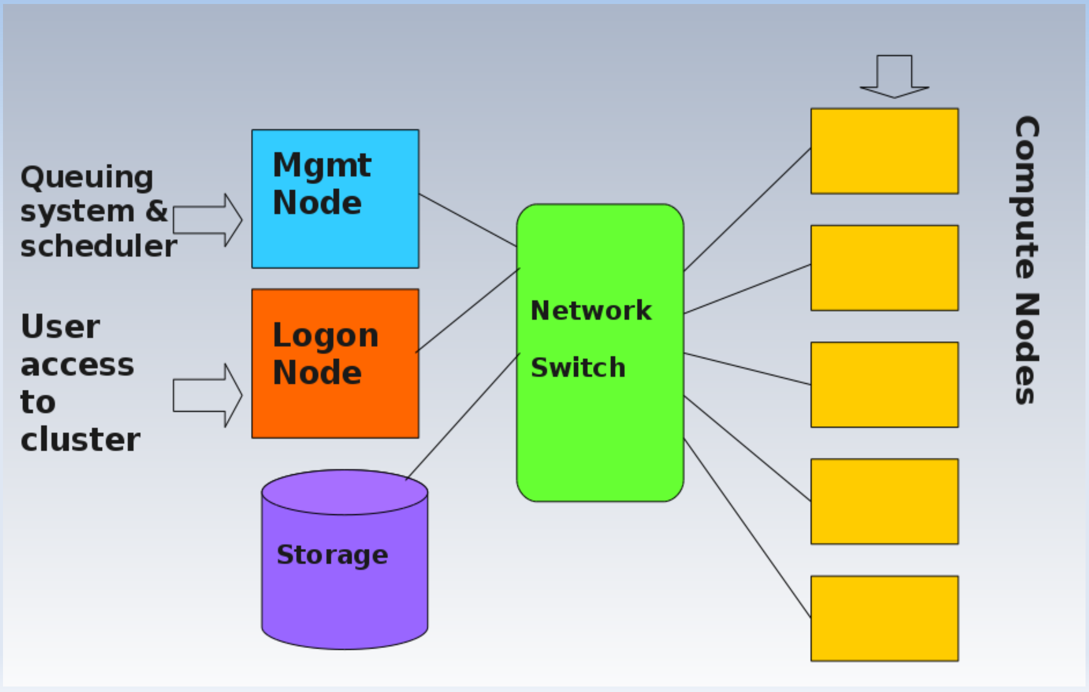
In 2016 CSIRO researchers successfully manipulated the behaviour of Metallic Organic Frameworks to control their structure and alignment which provides opportunities real-time and implantable medical electric devices.

In 2019 University of Melbourne researchers used Spartan to automatically detect classify and count unique classes of trucks and trailers.

Most research these days that involves large datasets and/or complex computatic is made on HPC systems.



HPC Cluster Design

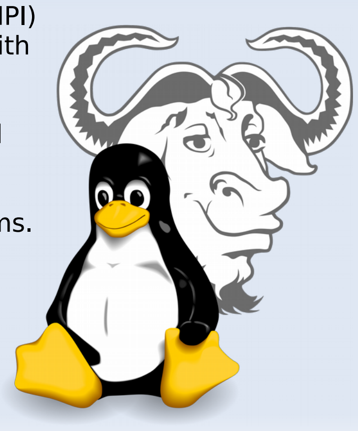


It's A GNU/Linux World

From November 2017 onwards of the Top 500 Supercomputers worldwide, *every single machine* used Linux.

The command-line interface provides a great deal more power and is very resource efficient.

GNU/Linux scales and does so with stability and efficiency. Critical software such as the Message Passing Interface (MPI) and nearly all scientific programs are designed to work with GNU/Linux.



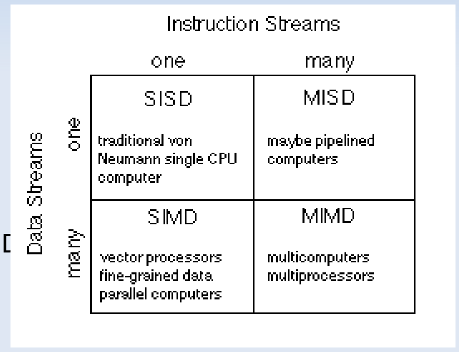
The operating system and many applications are provided as "free and open source", which means that not only are there are some financial savings, were also much better placed to improve, optimize and maintain specific programs.

Free or open source software (not always the same thing) can be can be compiled from source for the specific hardware and operating system configuration, and can be optimised according to compiler flags. There is necessary where every clock cycle is important.

Flynn's Taxonomy and Multicore Systems

It is possible to illustrate the degree of parallelisation by using Flynn's Taxonomy of Computer Systems (1966), where each process is considered as the execution of a pool of instructions (instruction stream) on a pool of data (data stream).

Over time computing systems have moved towards multi-processor, multi-core, and often multi-threaded and multi-node systems.



Data Streams	Instruction Streams: one	Instruction Streams: many
one	SISD traditional von Neumann single CPU computer	MISD maybe pipelined computers
many	SIMD vector processors fine-grained data parallel computers	MIMD multicomputers multiprocessors

As computing technology has moved increasingly to the MIMD taxonomic classification additional categories have been added:

Single program, multiple data streams (SPMD)
Multiple program, multiple data streams (MPMD

Some trends include GPGPU development, massive multicore systems (e.g., The Angstrom Project, the Tile CPU with 1000 cores) and massive network connectivity and shared resources (e.g., Plan9 Operating System).

(Image from Dr. Mark Meyer, Canisius College)

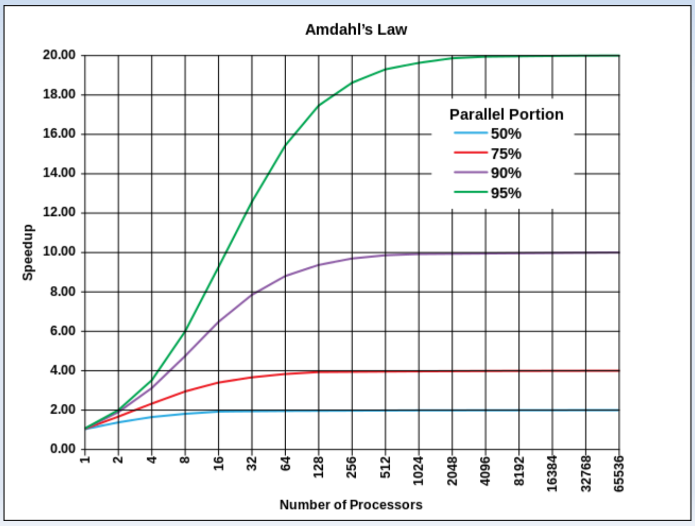
Limitations of Parallel Computation

Parallel programming and multicore systems should mean better performance. This can be expressed a ratio called speedup

Speedup (p) = Time (serial)/ Time (parallel)

Correctness in parallelisation requires synchronisation. Synchronisation and atomic operations causes loss of performance, communication latency.

Amdahl's law, establishes the maximum improvement to a system when only part of the system has been improved. Gustafson and Barsis noted that Amadahl's Law assumed a computation problem of fixed data set size.



What's More Important Than Performance?

There are many things in software and hardware that are more important than performance.

Correctness of code and signal
Clarity of code and architecture
Reliability of code and equipment
Modularity of code and components
Readability of code and hardware documentation
Compatibility of code and hardware

All this must be done before engaging in performance.

Common rule in parallel programming:
develop a working serial version of your code first and then work out what parts can be made parallel.

Do We Need Performance?

Early systems needed performance engineering by necessity. Software development was more advanced that hardware capacity. However, hardware developments increased rapidly and for many years performance engineering was not a priority as a result.

This led to many major software engineers to argue that optimisation was something to avoid, or at very least, defer in preference to working, readable code, and that does not have difficult levels of complexity.

”More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.” (William Wulf, 1972)

”Premature optimization is the root of all evil.” (Donald Knuth, 1974)

”The First Rule of Program Optimization: Don’t do it. The Second Rule of Program Optimization - For experts only: Don’t do it yet.” (Michael Jackson, 1988?)

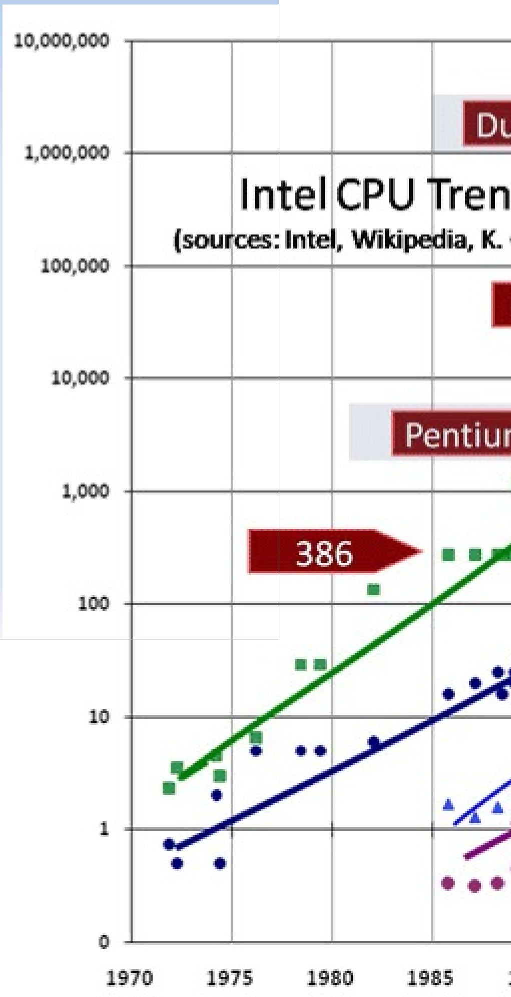
Nevertheless, optimisation did occur - especially at the design, build and compile level. Identification of critical locations in code (profiling) aids identification on what parts of source-code can be optimised.

Hardware Performance Meets Physics

Technological scaling from the 1960s to the early 2000s followed the propositions of Moore's Law (transistor density), Dennard Scaling (voltage and current scale down), and Koomey's Law (computations per joule), and most importantly, processor clock frequency.

At that point it become obvious that hardware improvements was facing an encounter with basic physics. *Heat.*

Clock speed flattened from the early 2000s onwards, but transistor count could continue to increase by the adoption of multi-core and multi-processor systems. More recently, for *certain types* of parallel processing, the adoption of general purpose use for graphics processing units (GPUs)



Processing with Performance and Scale

The increase in data volume, velocity, and variety is well-studied, with a 23% increase in storage during the 1980s to 2000s (Hilbert, López, 2007). Expectation that global data volume will have increased by an order of magnitude (4.4 zettabytes to 44 zettabytes) from 2013 to 2020 (Reinsel et al 2017).

Storage issues are only part of the problem of the deluge of data. Datasets are stored for a purpose, which means that they must be manipulated or modified in some way, i.e. processed.

Distributed or loosely coupled computational systems, such as various grid computing architectures (e.g., SETI@home, folding@home) is not the solution for large datasets except in cases where the dataset can be broken up, in which case it is a large collection of smaller datasets.

Couple the problem with large datasets, increasingly complex problems, and the bottlenecks in processing capacity per processor and it becomes very clear on why more tasks are moving to high performance computing.

Programming with Performance and Scale

To process data with performance and scale one needs hardware which can operate with performance and scale and software that is written with performance and scale in mind. The hardware side is covered by high performance compute clusters, and schedulers. The software side requires making the right programming choices (e.g., what sort of problem am I trying to solve?, c.f., John Gustafson) and adopting the right tools and the right techniques.

Using an example from Charles Leiserson's MIT's course "Performance Engineering of Software Systems" a square matrix multiplication, where C_ij = Sum A_ik*B_jk, assume n=2^k, using AWS c4.8x Large Machines (Intel Xeon E5-2666 v3, 2 processors, 9 cores per processor, 2.9GHz), peak 836 GFLOPS. The Python example (nested loops) takes 21,042 seconds; Java takes 2,738 seconds (i.e., 8.8x faster than Python), and CLang takes 1,156 seconds (i.e., 18x faster than Python). Why? An interpreted language versus a byte-code with JiT compilation, versus compilation to machine-code.

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

UniMelb's HPC System: Spartan

A detailed review was conducted in 2016 looking at the infrastructure of the Melbourne Research Cloud, High Performance Computing, and Research Data Storage Services. University desired a 'more unified experience to access compute services'

Recommended solution, based on technology and usage, is to make use of existing NeCTAR Research cloud with an expansion of general cloud compute provisioning and use of a smaller "true HPC" system on bare metal nodes.

Heterogeneous hardware:
Physical/BigMem partition: 28 nodes, 1204 cores, more powerful processors, more memory faster interconnect
Cloud/Longcloud partition: 165 nodes, 1980 cores, less powerful processors, less memory, slower interconnect
GPGPU partition: 73 nodes, 1752 cores, 292 P100 GPUs
Several specialist and project specific partitions.
Full range available at https://dashboard.hpc.unimelb.edu.au

Spartan is ~~Small but~~ Important

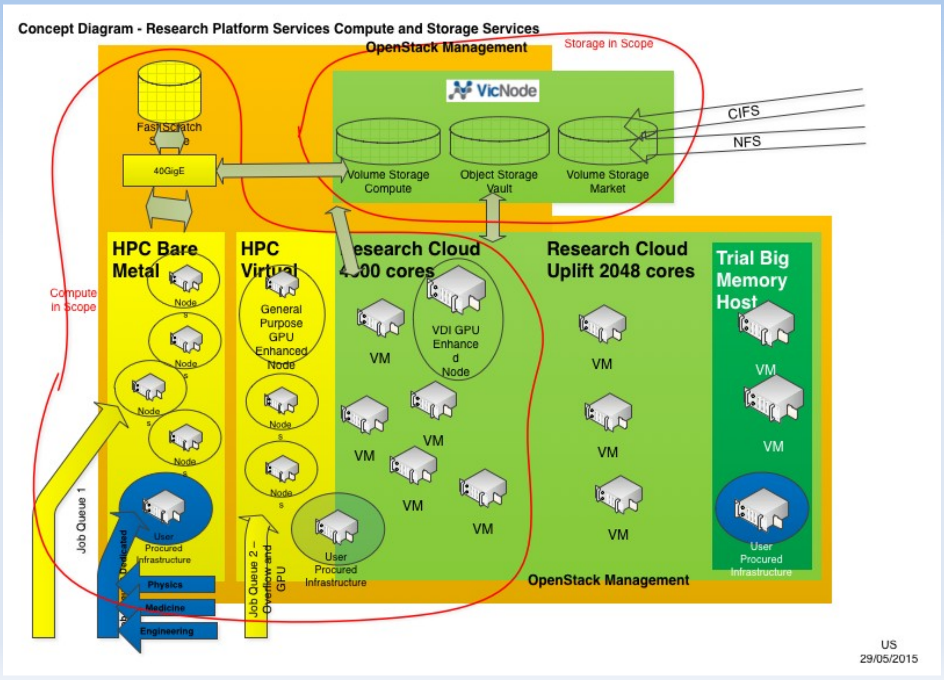
Spartan as a model of an HPC-Cloud Hybrid has been featured at Multicore World, Wellington, 2016, 2017; eResearchAustralasia 2016, Center for Scientific Computing (CSC) Goethe University Frankfurt, 2016, High Performance Computing Center (HLRS) University of Stuttgart, 2016, High Performance Computing Centre Albert-Ludwigs-University Freiburg, 2016; European Organization for Nuclear Research (CERN), 2016, Centre Informatique National de l'Enseignement Supérieur, Montpellier, 2016; Centro Nacional de Supercomputación, Barcelona, 2016, and the OpenStack Summit, Barcelona 2016.
https://www.youtube.com/watch?v=6D1IobuCZqE

Also featured in OpenStack and HPC Workload Management in Stig Telfer (ed), The Crossroads of Cloud and HPC: OpenStack for Scientific Research, Open Stack, 2016
http://openstack.org/assets/science/OpenStack-CloudandHPC6x9Booklet-v4-online.pdf

Architecture also featured in:
Spartan and NEMO: Two HPC-Cloud Hybrid Implementations.
2017 IEEE 13th International Conference on e-Science, DOI: 10.1109/eScience.2017.70

The Chimera and the Cyborg, Hybrid Compute Advances in Science, Technology and Engineering Systems Journal, Vol. 4, No. 2, 01-07, 2019

The Original Architecture



Spartan's Early Performance Tests

Service (usecs)	Network Device	Network	Protocol	Latency
UoM HPC Traditional	Mellanox	56Gb	Infiniband FDR	1.17
Legacy Edward HPC	Cisco Nexus	10Gbe	TCP/IP	19
Spartan Cloud nodes	Cisco Nexus	10Gbe	TCP/IP	60
Spartan Bare Metal	Mellanox	40Gbe	TCP/IP	6.85
Spartan Bare Metal	Mellanox	25Gbe	RDMA Ethernet	1.84
Spartan Bare Metal	Mellanox	40Gbe	RDMA Ethernet	1.15
Spartan Bare Metal	Mellanox	56Gbe	RDMA Ethernet	1.68
Spartan Bare Metal	Mellanox	100Gbe	RDMA Ethernet	1.3

Job	Task	Resources	Control HPC	Spartan Cloud
BWA	Disk	8 core Single Node	1:18:49	1:02:56	1:40:21
GROMACS	Compute	128 core Multinode	0:30:02	0:30:10	0:30:32
NAMD	Compute, I/O	128 core Multinode	1:11:41	1:00:46	1:55:54

These are figures from 2016; recently we've had a significant upgrade to network and processors - about 25% improvement.

Setting Up An Account and Training

Spartan uses its own authentication that is tied to the university Security Assertion Markup Language (SAML). The login URL is `https://dashboard.hpc.unimelb.edu.au/karaage`

Users on Spartan must belong to a project. Projects must be led by a University of Melbourne researcher (the "Principal Investigator") and are subject to approval by the Head of Research Compute Services.

Participants in a project can be researchers or research support staff from anywhere.

The University, through Research Platforms, has an extensive training programme for researchers who wish to use Spartan. This includes day-long courses in “Introduction to Linux and HPC Using Spartan”, “Advanced Linux and Shell Scripting for High Performance Computing”, “Parallel Programming On Spartan”, “GPU Programming with OpenACC and CUDA”, and “Regular Expressions Using Linux”.

University of Melbourne is a key contributor to the International HPC Certification Program.

Logging In and Help

To log on to a HPC system, you will need a user account and password and a Secure Shell (ssh) client. Most HPC cluster administrators do not allow connections with protocols such as Telnet, FTP or RSH as they insecurely send passwords in plain-text over the network, which is easily captured by packet analyser tools (e.g., Wireshark). Linux distributions almost always include SSH as part of the default installation as does Mac OS 10.x, although you may also wish to use the Fugu SSH client. For MS-Windows users, the free PuTTY client is recommended. To transfer files use scp, WinSCP, Filezilla, and especially rsync.

Logins to Spartan are based on POSIX identity for the system

`ssh your-username@spartan.hpc.unimelb.edu.au` or

For help go to http://dashboard.hpc.unimelb.edu.au or check man spartan. Lots of example scripts at /usr/local/common

Need more help? Problems with submitting a job, need a new application or extension to an existing application installed, if job generated unexpected errors etc., an email can be sent to: hpc-support@unimelb.edu.au

Don't email individual sysadmins. Provide as much information as possible (e.g., location of job script, error message etc)

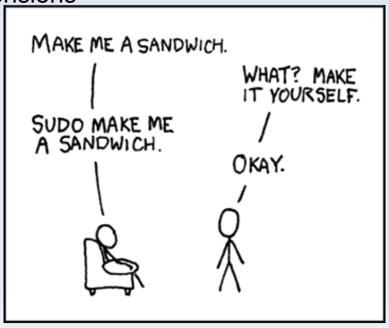
Things You Should Not Do

Do not run jobs on the login node. This is a shared location and if you use up the cores, memory or other resources there, you will make life miserable for everyone. Use the batch system instead. This lecture will show you how!

If you need to do compute-memory tasks with immediate feedback, run an interactive job. This lecture will show you how!

Do not use sudo to run commands. Giving users root access on a shared system would be a security nightmare! You don't have sudo rights and you'll get a stern email (probably from me) if you try.

Do not try to install software with apt, yum or some other package manager. It will not work. We typically compile software from source and add extensions to particular versions.



Exception: we do allow some local extensions to be installed for Python, R, Perl etc. Check the individual application information on Spartan for this.

Exception: we do allow users to compiler their own software for specific projects which is usually stored in their project directory.

The Linux Environment and Modules

Assumption here is that everyone has had exposure to the Linux command line. If not, you'd better get some! At least learn the twenty or so basic environment commands to navigate the environment, manipulate files, manage processes. Plenty of good online material available (e.g., my book "Supercomputing with Linux", https://github.com/VPAC/superlinux)

Environment modules provide for the dynamic modification of the user's environment (e.g., paths) via module files. Each module contains the necessary configuration information for the user's session to operate according according to the modules loaded, such as the location of the application's executables, its manual path, the library path, and so forth.

Modulefiles also have the advantages of being shared with many users on a system and easily allowing multiple installations of the same application but with different versions and compilation options. Sometimes users want the latest and greatest of a particular version of an application for the feature-set they offer. In other cases, such as someone who is participating in a research project, a consistent version of an application is desired. In both cases consistency and therefore *reproducibility* is attained.

Having multiple version of applications available on a system is essential in research computing!

Modules Commands

Some basic module commands include the following:

```
module help
```
The command module help , by itself, provides a list of the switches, subcommands, and subcommand arguments that are available through the environment modules package.

```
module avail
```
This option lists all the modules which are available to be loaded.

```
module whatis <modulefile>
```
This option provides a description of the module listed.

```
module display <modulefile>
```
Use this command to see exactly what a given modulefile will do to your environment, such as what will be added to the PATH, MANPATH, etc. environment variables.

More Modules Commands

`module load <modulefile>`
This adds one or more modulefiles to the user's current environment (some modulefiles load other modulefiles.

module unload <modulefile>
This removes any listed modules from the user's current environment.

module switch <modulefile1> <modulefile2>
This unloads one modulefile (modulefile1) and loads another (modulefile2).

module purge
This removes all modules from the user's environment.

In the lmod system as used on Spartan there is also “module spider” which will search for all possible modules and not just those in the existing module path and provide descriptions.

(Image from NASA, Apollo 9 “spider module”)

Batch Systems and Workload Managers

The Portable Batch System (or simply PBS) is a utility software that performs job scheduling by assigning unattended background tasks expressed as batch jobs among the available resources.

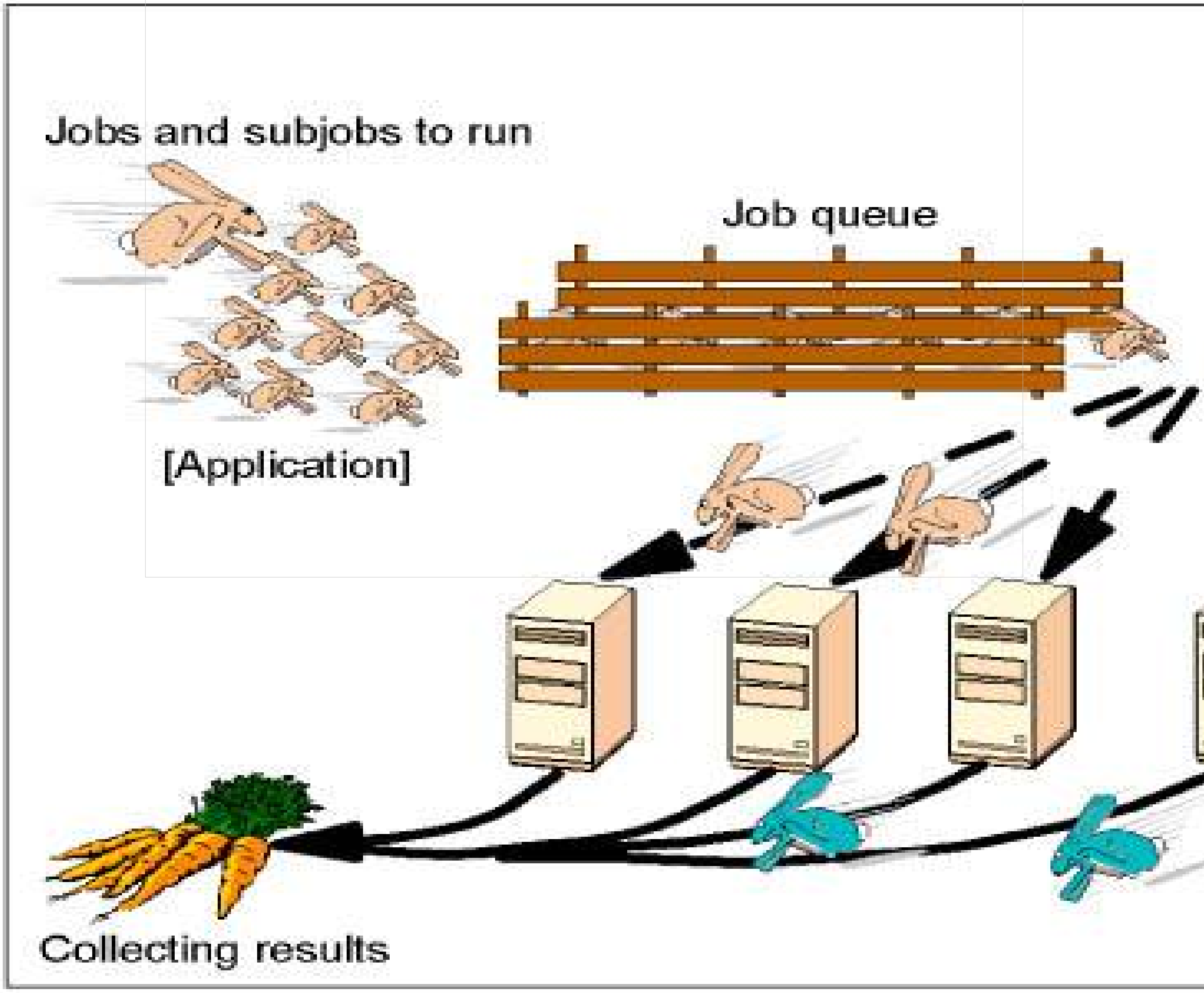
The original Portable Batch System was developed by MRJ Technology Solutions under contract to NASA in the early 1990s. In 1998 the original version of PBS was released as an open-source product as OpenPBS. This was forked by Adaptive Computing (formally, Cluster Resources) who developed TORQUE (Terascale Open-source Resource and QUEue Manager). Many of the original engineering team is now part of Altair Engineering who have their own version, PBSPro.

In addition to this the popular job scheduler Slurm (originally "Simple Linux Utility for Resource Management", SLURM), now simply called Slurm Workload Manager, also uses batch script where are very similar in intent and style to PBS scripts.

Spartan uses the Slurm Workload Manager. A job script written on one needs to be translated to another (handy script available pbs2slurm https://github.com/bjpop/pbs2slurm)

In addition to this variety of implementations of PBS different institutions may also make further elaborations and specifications to their submission filters (e.g., site-specific queues, user projects for accounting). (Image from the otherwise dry IBM 'Red Book' on Queue Management)

Submitting and Running Jobs



Submitting and Running Jobs

Submitting and running jobs is a relatively straight-forward process consisting of:

1) Setup and launch
2) Job Control, Monitor results
3) Retrieve results and analyse.

Don't run jobs on the login node! Use the queuing system to submit jobs.

1. Setup and launch consists of writing a short script that initially makes resource requests and then commands, and optionally checking queueing system.

Core command for checking queue:	squeue | less
Alternative command for checking queue:	showq -p cloud | less
Core command for job submission:	sbatch [jobscript]

2. Check job status (by ID or user), cancel job.

Core command for checking job in Slurm:	squeue -j [jobid]
Detailed command in Slurm:	scontrol show job [jobid]
Core command for deleting job in Slurm:	scancel [jobid]

3.Slurm provides an error and output files They may also have files for post-job processing. Graphic visualisation is best done on the desktop.

Simple Script Example

```
#!/bin/bash
#SBATCH --partition=cloud
#SBATCH --time=01:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
module load my-app-compiler/version
my-app data
```

The script first invokes a shell environment, followed by the partition the job will run on (the default is 'cloud' for Spartan). The next four lines are resource requests, specifically for one compute node, one task.

After these requests are allocated, the script loads a module and then runs the executable against the dataset specified. Slurm also automatically exports your environment variables when you launch your job, including the directory where you launched the job from. If your data is a different location this has to be specified in the path!

After the script is written it can be submitted to the scheduler. See examples in /usr/local/common/IntroLinux

```
[lev@spartan]$ sbatch myfirstjob.slurm
```

Multi-threaded, Multi-core, and Multi-node Examples

Modifying resource allocation requests can improve job efficiency if the applications has been designed to take advantage of this. The scheduler simply allocates resources, it does not make your application parallel!

For example shared-memory multithreaded jobs on Spartan (e.g., OpenMP), modify the --cpus-per-task to a maximum of 8, which is the maximum number of cores on a single instance.

```
#SBATCH --cpus-per-task=8
```

For distributed-memory multicore job using message passing, the multinode partition has to be invoked and the resource requests altered e.g.,

```
#!/bin/bash
#SBATCH -partition=physical
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
module load my-app-compiler/version
srun my-mpi-app
```

Note that multithreaded jobs *cannot* be used in a distributed memory model across nodes. They can however exist be conducted on distributed memory jobs which include a shared memory component (hybrid OpenMP-MPI jobs).

Arrays : Many Jobs, One Script

Job arrays allow the same batch script, and therefore the same resource requests, and launches multiple jobs simultaneously. A typical example is to apply the same task across multiple datasets

The array directive sets the index values which can be a range (e.g., 0-31), comma-separated values, (e.g., 1,3,9,11) or a range with a step value (e.g., 1-7:2)

The following example submits 10 batch jobs with myapp running against datasets dataset1.csv, dataset2.csv, ... dataset10.csv

```
#SBATCH --array=1-10
myapp ${SLURM_ARRAY_TASK_ID}.csv
```

Examples are in /usr/local/common/array and /usr/local/common/Octave

When a job array is launched each subjob gets its own job number that is a subset of the main job e.g., 14950773_1, 14950773_2, 14950773_3 etc. These can be managed by the normal schedule commands (e.g., scancel 14950773_3)

Dependencies : Job Pipelines

A dependency condition is established on which the launching of a batch script depends, creating a conditional pipeline. A typical use case is where the output of one job is required as the input of the next job.

```
#SBATCH --dependency=afterok:myfirstjobid mysecondjob
```

Several conditional directives to be placed on a job which are tested prior to the job being intiatied, which are summarised as after, afterany, afterok, afternotok, singleton (e.g., sbatch --dependency=afterok:jobid1 job2.slurm). Multiple jobs can be listed as dependencies with colon separated values (e.g., `sbatch --dependency=afterok:jobid1:jobid2 job3.slurm`).

Some dependency types (not these differ from those used PBSPro and TORQUE)

after:jobid[:jobid...]	job can begin after the specified jobs have started
afterany:jobid[:jobid...]	job can begin after the specified jobs have terminated
afternotok:jobid[:jobid...]	job can begin after the specified jobs have failed
afterok:jobid[:jobid...]	job can begin after the specified jobs have run to completion with an exit code of zero (see the user guide for caveats).
singleton	job can begin execution after all previously launched jobs with the same name and user have ended.

See example in /usr/local/common/depend

Interactive Jobs

For real-time interaction, with resource requests made on the command line, an interactive job is called. This puts the user on to a compute node.

This is typically done if they user wants to run a large script (and shouldn't do it on the login node), or wants to test or debug a job. The following command would launch one node with two processors for ten minutes.

Once an interactive job is launched, the user can conduct whatever computationally intensive task they wish with the resources that they have allocated. For example, a multi-threaded application, tested in normal and multi-threaded computation.

```
[lev@spartan interact]$ sinteractive --ntasks=2 --cpus-per-task=2
srun: job 15489392 queued and waiting for resources
srun: job 15489392 has been allocated resources
[lev@spartan-rc002 interact]$ gcc iterate.c -o iterate
[lev@spartan-rc002 interact]$ time ./iterate
..
[lev@spartan-rc002 interact]$ export OMP_NUM_THREADS=2
[lev@spartan-rc002 interact]$ gcc -fopenmp iterate.c -o iterate
[lev@spartan-rc002 interact]$ time ./iterate
```

X-Windows Forwarding

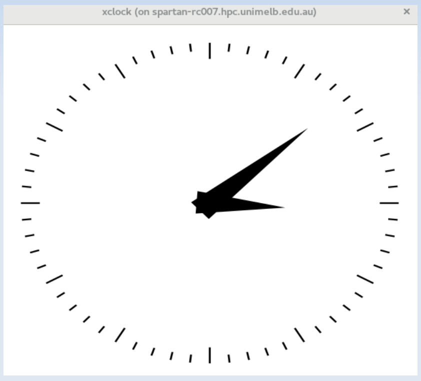
In almost all cases it is much better to do computation on the cluster and visualisation on a local system. In some cases however it is unavoidable to require x-windows forwarding.

It is best to login with the -X option for security and then to login with -X again to the login node. The compute node with then pass through the graphics via the login node to the desktop system.

Please note that you will need an x-windows client on your desktop for the visualisation.

```
[lev@cricetomys ~]$ ssh
lev@spartan.hpc.unimelb.edu.au -X

[lev@spartan]$ sinteractive --nodes=1 --ntasks-per-
node=2 -X
srun: job 602795 queued and waiting for resources
srun: job 602795 has been allocated resources
[lev@spartan-rc002 ~]$ xclock
```



Job Scripts are Shell Scripts

Because the script first invokes a shell environment all job scripts are shell scripts. This means that any of the tools and techniques used in bash shell programming can also be used in job submission scripts, which is very handy managing files and directories within the environment.

It also means that the scheduler directives (#SBATCH) are interpreted as comments by the shell environment, but are understood by the Slurm scheduler.

It also means that scheduler directives must come first! Do all your scheduler directives, then include your shell commands (including loading your modules)

This includes assigning variables, shell substitutions, loops, conditional statements, case statements, shell functions, heredocs and so forth.

There are are number of shell scripting examples in /usr/local/common/HPCshells, and example of a Slurm job with extensive shell commands in /usr/local/common/HPCshells/NAMD and the use of a heredoc and a loop in /usr/local/common/HPCshells/Gaussian

Job Staging for Network Performance

Where you are on the system matters for job performance; this applies to all systems, and HPC systems are no different - and it is a well known problem. See Grace Hopper, “Mind Your Nanoseconds” (https://www.youtube.com/watch?v=9eyFDBPk4Yw)

For small test jobs (such as the examples gives in this lecture) launching with a dataset in one's home directory or project directory is sufficient. For larger datasets, one should use the /scratch or a local disk directory which have faster disk, interconnect and/or shorter distance.

Sorry! There is no /scratch directory for this class!

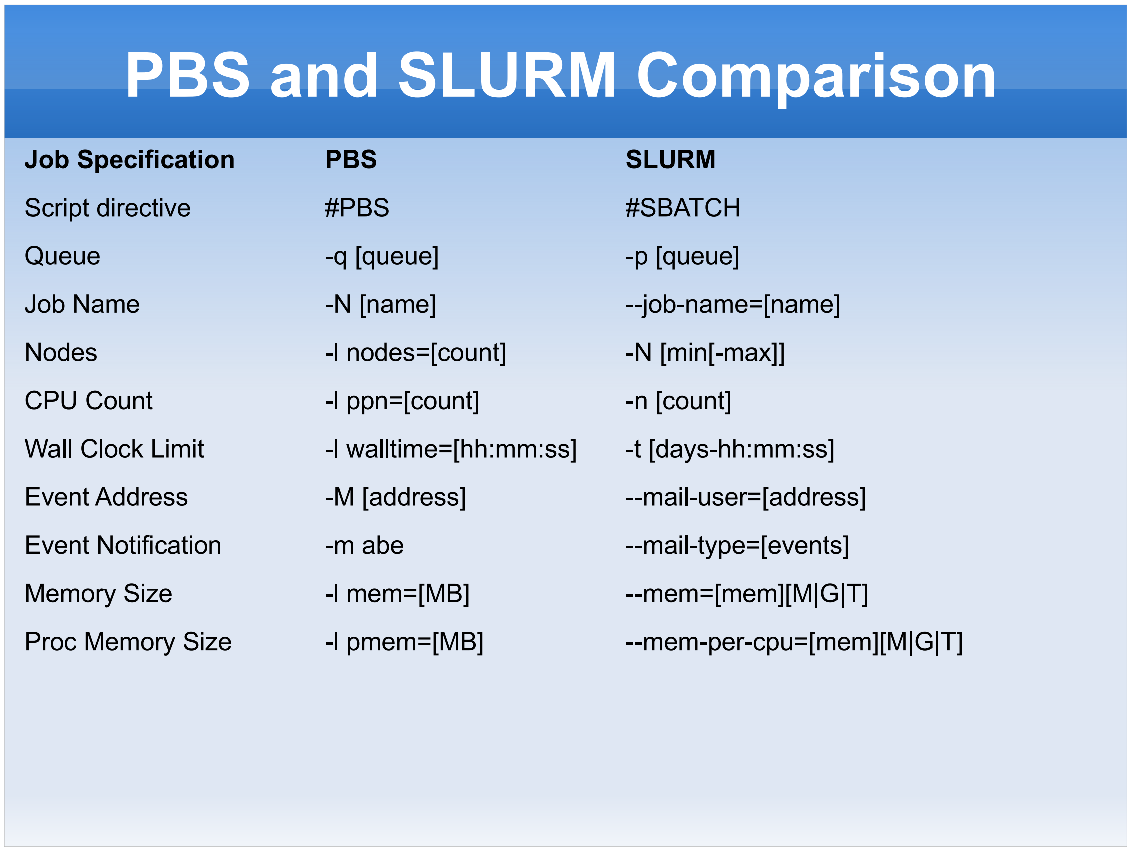
In general:

/home, /data/projects/$projectID is slower.
/scratch/$projectID is faster
/var/local/tmp is the fastest

But /var/local/tmp is the local disk of the compute node! So you must copy the results back to one's project directory (for example) when as part of the job script.

PBS, SLURM Comparison

User Commands	**PBS/Torque**	**SLURM**
Job submission	qsub [script_file]	sbatch [script_file]
Job submission	qdel [job_id]	scancel [job_id]
Job status (by job)	qstat [job_id]	squeue [job_id]
Job status (by user)	qstat -u [user_name]	squeue -u [user_name]
Node list	pbsnodes -a	sinfo -N
Queue list	qstat -Q	squeue
Cluster status	showq, qstatus -a	squeue -p [partition]
Environment		
Job ID	$PBS_JOBID	$SLURM_JOBID
Submit Directory	$PBS_O_WORKDIR	$SLURM_SUBMIT_DIR
Submit Host	$PBS_O_HOST	$SLURM_SUBMIT_HOST
Node List	$PBS_NODEFILE	$SLURM_JOB_NODELIST
Job Array Index	$PBS_ARRAYID	$SLURM_ARRAY_TASK_ID



PBS and SLURM Comparison

Job Specification	PBS	SLURM
Script directive	#PBS	#SBATCH
Queue	-q [queue]	-p [queue]
Job Name	-N [name]	--job-name=[name]
Nodes	-l nodes=[count]	-N [min[-max]]
CPU Count	-l ppn=[count]	-n [count]
Wall Clock Limit	-l walltime=[hh:mm:ss]	-t [days-hh:mm:ss]
Event Address	-M [address]	--mail-user=[address]
Event Notification	-m abe	--mail-type=[events]
Memory Size	-l mem=[MB]	--mem=[mem][M|G|T]
Proc Memory Size	-l pmem=[MB]	--mem-per-cpu=[mem][M|G|T]

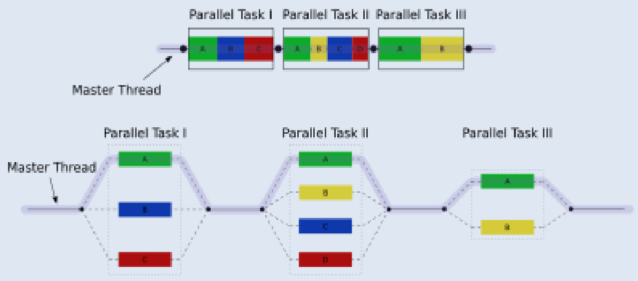
Shared Memory Parallel Programming

One form of parallel programming is multithreading, whereby a master thread forks a number of sub-threads and divides tasks between them. The threads will then run concurrently and are then joined at a subsequent point to resume normal serial application.

One implementation of multithreading is OpenMP (Open Multi-Processing). It is an Application Program Interface that includes directives for multi-threaded, shared memory parallel programming. The directives are included in the C or Fortran source code and in a system where OpenMP is not implemented, they would be interpreted as comments.

There is no doubt that OpenMP is an easier form of parallel programming, however it is limited to a single system unit (no distributed memory) and is thread-based rather than using message passing. Many examples in `/usr/local/common/OpenMP`.

(image from: User A1, Wikipedia)



Shared Memory Parallel Programming

```
#include <stdio.h>
#include  "omp.h"
int main(void)
{
   int id;
   #pragma omp parallel num_threads(8) private(id)
   {
   int id = omp_get_thread_num();
   printf("Hello world %d\n", id);
   }
return 0;
}
```

```
program hello2omp
   include "omp_lib.h"
   integer :: id
   !$omp parallel num_threads(8) private(id)
     id = omp_get_thread_num()
          print *, "Hello world", id
   !$omp end parallel
end program hello2omp
```

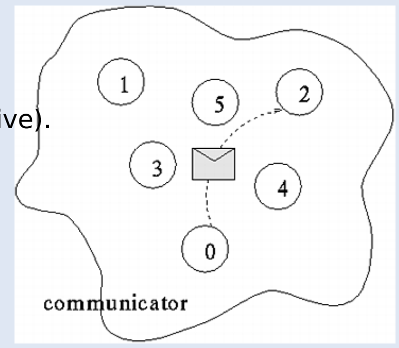
Distributed Memory Parallel Programming

Moving from shared memory to parallel programming involves a conceptual change from multi-threaded programming to a message passing paradigm. In this case, MPI (Message Passing Interface) is one of the most well popular standards and is used here, along with a popular implementation as OpenMPI.

The core principle is that many processors should be able cooperate to solve a problem by passing messages to each through a common communications network.

The flexible architecture does overcome serial bottlenecks, but it also does require explicit programmer effort (the "questing beast" of automatic parallelisation remains somewhat elusive).

The programmer is responsible for identifying opportunities for parallelism and implementing algorithms for parallelisation using MPI.

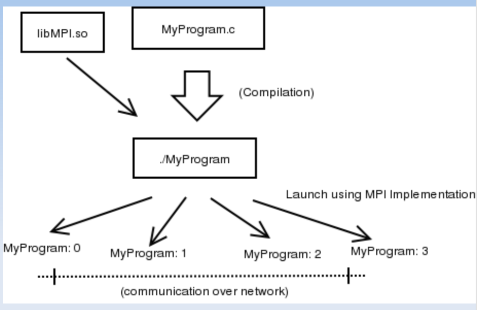


Distributed Memory Parallel Programming

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int  argc;
char **argv;
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}

!       Fortran MPI Hello World
        program hello
        include 'mpif.h'
        integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)

        call MPI_INIT(ierror)
        call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
        print*, 'node', rank, ': Hello world'
        call MPI_FINALIZE(ierror)
        end
```



MPI Compilation and Job Scripts

The OpenMP example needs to be compiled with OpenMP directives. The OpenMP example cannot run across compute nodes; therefore it is best run on the “cloud” partition. The OpenMPI compilation needs to call the MPI wrappers.

```
module load OpenMPI/1.10.0-GCC-4.9.2
gcc -fopenmp helloomp.c -o helloompc
mpigcc mpihelloworld.c -o mpihelloworld
```

```
#!/bin/bash
#SBATCH -p cloud
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=16
export OMP_NUM_THREADS=16
module load GCC/4.9.2
mpiexec helloompc
```

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --nodes=2
#SBATCH --ntasks=16
module load OpenMPI/1.10.2-GCC-4.9.2
mpiexec mpi-helloworld
```

MPJ Express for Java

Java can be compiled with MPI bindings; we have done this with OpenMPI/3.0.0 only. A more common option is to use MPJ-Express. The following “HellowWorld.java” program is compiled and executed. See /usr/local/common/Java

```
import mpi.*;
public class HelloWorld {
public static void main(String args[]) throws Exception {
        MPI.Init(args);
        int me = MPI.COMM_WORLD.Rank();
        int size = MPI.COMM_WORLD.Size();
        System.out.println("Hi from <"+me+">");
        MPI.Finalize();
        }
}
```

```
sinteractive --time=1:00:00 --nodes=1 --ntasks=2
module load MPJ-Express
javac -cp .:/usr/local/easybuild/software/MPJ-Express/0.44-goolf-
2015a-Java-9.0.4/lib/mpj.jar HelloWorld.java
mpjrun.sh -np 2 HelloWorld
```

MPI4Py Express for Python

Python too has various MPI bindings available. The most common used is MPI4Py. Sample assignment data is provided from the 2016 and 2015 in the Spartan directory, `/usr/local/common`. As a package it can be simply imported (e.g., `from mpi4py import MPI`).

But remember! With environment modules with extensions you do not necessarily get all the packages/libraries/extensions that you might expect. See the README file for an explanation of how to review the extensions already installed.

Examples provided in the directory of various Python jobs with MPI bindings with Slurm submissions scripts.

Usually we have examples for for single-core, dual core, eight-core, sixteen-core (different partition), and two-nodes with four cores each.

But that's for your assignment!

MPI Communication: A Game of Ping-Pong

A very popular and basic use of MPI Send and Recv routines is a ping-ping program. Why? Because it can be used to test latency within and between nodes and partitions if they have different interconnect (like on Spartan).

An example is given in `/usr/local/common/OpenMPI` as `mpi-pingpong.c` with a job submission script that can be modified accord to the test case `mpi-pingpong.slurm`. There are some routines here which manage the communication in the ping-pong activity.

`MPI_Status()` MPI_Status is not a routine, but rather a data structure and is typically attached to an MPI_Recv() routine.

`MPI_Request()` A wrapper for MPI Requests such as wait, waitany, waitall, waitsome, start, cancel, startall.

`MPI_Barrier()` Enforces synchronisation between MPI processes in a group by placing a barrier on communication between groups. An MPI barrier completes after all group members have entered the barrier.

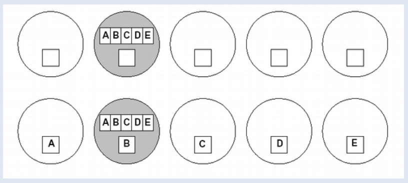
`MPI_Wtime()` - Returns an elapsed time as a floating-point number of seconds on the calling processor from an arbitrary time in the past.

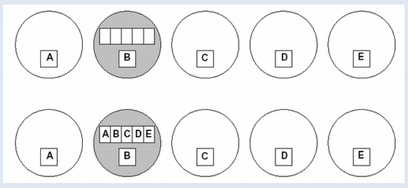
MPI Collective Communications

There are *many* other MPI routines which are *not* going to explored here! However just as a little taste one of the most popular is MPI collective communications and reduction operations. Collective communications include `MPI_Broadcast`, `MPI_Scatter`, `MPI_Gather`, `MPI_Reduce`, and `MPI_Allreduce`.

`MPI_Bcast` Broadcasts a message from the process with rank "root" to all other processes of the communicator, including itself. It is significantly more preferable than using a loop.

`MPI_Scatter` sends data from one task to all tasks in a group; the inverse operation of `MPI_Gather.` The outcome is as if the root executed n send operations and each process executed a receive. `MPI_Scatterv` scatters a buffer in parts to all tasks in a group.

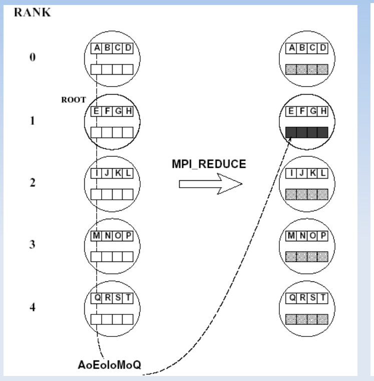


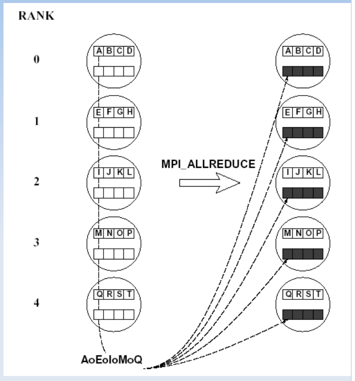


MPI Reduction Communications

`MPI_Reduce` performs a reduce operation (such as sum, max, logical AND, etc.) across all the members of a communication group.

`MPI_Allreduce` conducts the same operation but returns the reduced result to all processors.





The general principle in Reduce and All Reduce is the idea of reducing a set of numbers to a small set via a function. If you have a set of numbers (e.g., [1,2,3,4,5]) a reduce function (e.g., sum) can convert that set to a reduced set (e.g., 15). MPI_Reduce takes in an array of values as that set and outputs the result to the root process. MPI_AllReduce outputs the result to all processes.

THANKS FOR WATCH
& LISTENING PATIEN