Intermediate Supercomputing
Course Objectives

• Understand basic concepts of parallel programming

• Compile and run code on Pawsey’s supercomputers

• Develop and use advanced job scripts

• Understand how to get good performance out of the filesystem
OVERVIEW OF EXERCISES
Exercise Files

- **Pi Darts program**
  - Computes the value of Pi using Monte Carlo method

\[
A(C) = \pi R^2 \\
A(S) = 4R^2 \\
\pi = 4 \frac{A(C)}{A(S)} \\
\pi = 4 \frac{\# \text{ darts in circle}}{\# \text{ darts in square}}
\]
Exercise Files

• Accuracy of Pi depends on number of darts

• Multiple versions:
  • serial,
  • Python,
  • Intel MKL,
  • OpenMP,
  • MPI and MPI collective,
  • MPI+OpenMP hybrid,
  • CUDA GPU.

• Exercises will involve editing, compiling, and launching job scripts
Downloading exercises

• Log in to Magnus via ssh:

  ssh username@magnus.pawsey.org.au

• Change directory to $MYGROUP

  cd $MYGROUP

• Use git to download the exercise material:

  git clone https://github.com/PawseySupercomputing/Intermediate-Supercomputing.git

Change into the downloaded directory:

  cd Intermediate-Supercomputing

• List the contents of the directory:

  ls
Separate directories for each version of the code e.g.: darts/serial

C and FORTRAN versions

- C version today, but the instructions are transferable to FORTRAN

.c files
- Where the action happens

.h files
- Header files
- Declaration of functions and directives
- Allows us to use outside libraries (modularity)

Makefile
- Recipe for how to compile code in a single step

Queueing script: \texttt{job.slurm}
Submitting exercises

Always use the SLURM reservation ID:

```bash
sbatch --reservation=[RESERVATION_ID] job.slurm
```
PARALLEL PROGRAMMING INTRODUCTION
Parallel architectures - shared memory

• For the memory as a compute resource, a node can be seen as a *shared memory architecture*

• All processors share a global memory address space

• Sharing the same data, performing individual tasks

• Uniform Memory Access (UMA)

• Non-Uniform Memory Access (NUMA)
Parallel architectures – distributed memory

- Each node has its own local memory and data is transferred across the nodes through the network.
Parallel architectures – hybrid systems

- Each node has a hybrid design with accelerators (e.g. GPUs) with their local high bandwidth memory space
- Multiple levels of parallelism
Parallel Programming Models

Depending on the architecture at hand, following are the commonly used programming models:

• Shared memory model
  • Without threads (e.g. using locks and semaphores)
  • Thread model (e.g. pthreads, OpenMP)

• Distributed Memory Model (Message Passing Interface MPI)

• Data parallel model (e.g. Partitioned Global memory address space PGAS)

• Accelerated model (e.g. CUDA, OpenACC)

• Hybrid (e.g. MPI+OpenMP, MPI+CUDA, MPI+OpenACC)
Thread-based parallelism for shared memory systems
Explicit parallelism (parallel regions)
Fork/join model
Based mostly on inserting compiler directives in the code
Parallelism in MPI

- **Message Passing Interface**
  - Standard defining how CPUs send and receive data
  - Vendor specific implementation adhering to the standard
  - Allows CPUs to “talk” to each other
    - i.e. read and write memory
Accelerated computing

- High Bandwidth local memory – data transfers between CPU code and accelerated kernel

- Usage scenarios:
  - CUDA – library for developing and executing kernels
  - OpenACC – compiler directives (similar to OpenMP) for defining accelerated loops or kernels
  - GPU scientific libraries – accelerated versions of common scientific algorithms
PROGRAMMING ENVIRONMENT
Environment Variables

- Hold values to common locations and files on a system

- Dynamic
  - Can change between systems and users (your Magnus variables are different from Zeus)

- Powerful way simplify common tasks
## Environment Variables - examples

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATH</td>
<td>List of directories to search for binaries</td>
</tr>
<tr>
<td>HOME</td>
<td>Path to user’s home directory</td>
</tr>
<tr>
<td>SHELL</td>
<td>Which shell is being used (bash, csh, etc.)</td>
</tr>
<tr>
<td>LD_LIBRARY_PATH</td>
<td>Directories to search for run-time libraries (program execution)</td>
</tr>
<tr>
<td>LIBRARY_PATH</td>
<td>Directories to search for libraries to be linked (program compilation)</td>
</tr>
<tr>
<td>CPATH</td>
<td>Directories to search for header files (.h) – C/C++</td>
</tr>
<tr>
<td>FPATH</td>
<td>Directories to search for header files (.h) – Fortran</td>
</tr>
<tr>
<td>CFLAGS</td>
<td>Compiler flags (optimizations, debug, OpenMP, etc.) - C</td>
</tr>
<tr>
<td>CXXFLAGS</td>
<td>Compiler flags (optimizations, debug, OpenMP, etc.) – C++</td>
</tr>
<tr>
<td>FFLAGS,FCFLAGS</td>
<td>Compiler flags (optimizations, debug, OpenMP, etc.) - Fortran</td>
</tr>
<tr>
<td>MYSCRATCH</td>
<td>Pawsey-specific pointing to user’s scratch space</td>
</tr>
<tr>
<td>MYGROUP</td>
<td>Pawsey-specific pointing to user’s group space</td>
</tr>
</tbody>
</table>
Setting Variables

• Format: KEY = VALUE
  PATH = /usr/local/bin:/usr/bin:/usr/sbin:/opt/bin
  Order is important!

• Accessing and modifying environment variables
  > echo $PATH
  > export PATH=$PATH:/my/new/bin
  > printenv
Pawsey’s Environment

• Multiple applications with multiple builds and multiple compilers for multiple users!

• Module system to manage these variables for each application
  > module list
  > module avail
  > module show python
COMPIRING CODE
Interpreted vs compiled code

C/C++, Fortran programmes are translated to machine code by compilers:

- Very efficient code
- Compilation overhead incurred only once for multiple runs
- Programming skills required

Interpreted languages (like Python or R) are parsed, interpreted and executed each time the program is run:

- Usually much easier to program
- Parsing and interpreting overhead in each run – usually less efficient

Popular scenarios:

- multi-language e.g. interfaces in Python and compute intensive parts in C
- interpreted languages used for prototyping an application
Language performance

<table>
<thead>
<tr>
<th>Function</th>
<th>Fortran</th>
<th>Julia</th>
<th>Python</th>
<th>R</th>
<th>Matlab</th>
<th>Octave</th>
<th>Mathematica</th>
<th>JavaScript</th>
<th>Go</th>
<th>LuaJIT</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcc 4.8.5</td>
<td>0.6.2</td>
<td>3.6.3</td>
<td>3.3.1</td>
<td>R2018a</td>
<td>4.0.3</td>
<td>11.1.1</td>
<td>V8 4.4.103.53</td>
<td>go1.9</td>
<td>Scilua v1.0.0-b12</td>
<td>1.8.0_15</td>
</tr>
<tr>
<td>iteration_pi_sum</td>
<td>1.00</td>
<td>1.00</td>
<td>14.75</td>
<td>8.92</td>
<td>1.01</td>
<td>373.94</td>
<td>1.55</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>recursion_fibonacci</td>
<td>0.58</td>
<td>1.96</td>
<td>100.77</td>
<td>608.8</td>
<td>18.69</td>
<td>13127</td>
<td>139.89</td>
<td>3.84</td>
<td>1.93</td>
<td>1.36</td>
<td>1.73</td>
</tr>
<tr>
<td>recursion_quicksort</td>
<td>1.31</td>
<td>0.94</td>
<td>37.32</td>
<td>264.9</td>
<td>2.35</td>
<td>2559.4</td>
<td>44.77</td>
<td>2.91</td>
<td>1.16</td>
<td>1.51</td>
<td>2.63</td>
</tr>
<tr>
<td>parse_integers</td>
<td>5.29</td>
<td>1.35</td>
<td>19.98</td>
<td>50.90</td>
<td>229.56</td>
<td>3030.1</td>
<td>14.52</td>
<td>6.41</td>
<td>0.97</td>
<td>0.99</td>
<td>2.82</td>
</tr>
<tr>
<td>print_to_file</td>
<td>6.35</td>
<td>0.66</td>
<td>4.49</td>
<td>170.9</td>
<td>110.45</td>
<td>165.4</td>
<td>65.67</td>
<td>--</td>
<td>1.63</td>
<td>0.57</td>
<td>10.60</td>
</tr>
<tr>
<td>matrix_statistics</td>
<td>1.96</td>
<td>1.79</td>
<td>17.93</td>
<td>20.35</td>
<td>8.10</td>
<td>48.22</td>
<td>7.85</td>
<td>11.33</td>
<td>6.10</td>
<td>1.73</td>
<td>4.83</td>
</tr>
<tr>
<td>matrix_multiply</td>
<td>1.27</td>
<td>0.98</td>
<td>1.18</td>
<td>8.74</td>
<td>1.16</td>
<td>1.21</td>
<td>1.20</td>
<td>24.71</td>
<td>1.25</td>
<td>1.11</td>
<td>7.99</td>
</tr>
<tr>
<td>userfunc_mandelbrot</td>
<td>0.75</td>
<td>0.75</td>
<td>132.38</td>
<td>333.0</td>
<td>10.07</td>
<td>6622.4</td>
<td>19.20</td>
<td>1.07</td>
<td>0.79</td>
<td>1.03</td>
<td>1.12</td>
</tr>
</tbody>
</table>

Comparison from [http://julialang.org/](http://julialang.org/). (Serial execution, Intel Core i7-3960X)
Times relative to C (gcc 4.8.5). Lower numbers mean better performance.

These are not optimal results, just how a typical researcher might program in those languages.
Compiling

Source Code
- `func1.c`
- `func2.c`
- `func3.c`

Object Code
- `func1.o`
- `func2.o`
- `func3.o`

Compiler

Linker

Executable
- `myprog.exe`

External Libraries
Compiling Source Code

- Compiling and linking on a Cray (serial and MPI)
  
  ```
  cc -c myprog.c -o myprog.o
  cc myprog.o -o myprog.exe
  ```

- Compiling and linking on Zeus (GNU, Intel and MPI)
  
  ```
  gcc -c myprog.c -o myprog.o
  gcc myprog.o -o myprog.exe
  
  icc -c my_intelprog.c -o my_intelprog.o
  icc my_intelprog.o -o my_intelprog.exe
  
  mpicc -c my_mpiiprog.c -o my_mpiiprog.o
  mpicc my_mpiiprog.o -o my_mpiirog.exe
  ```
There are three compiler suites on Magnus/Galaxy

- PrgEnv-Cray, PrgEnv-Intel and PrgEnv-gnu
  - Fortran, C and C++ compilers
  - Different locations, include paths, libraries

- All support OpenMP/MPI/Libsci/BLAS etc
  - Each PrgEnv handles flags for these libraries

- Default: PrgEnv-cray

- Switch compilers easily
  module swap PrgEnv-cray PrgEnv-intel
  module swap PrgEnv-intel PrgEnv-gnu
Cray Programming Environment

- A cross-compiling environment
  - Compiler runs on the login node
  - Executable runs on the compute nodes
- Modules Environment
  - Allows easy swapping of tools & libraries
- Cray compiler drivers
  - Same command (ftn/cc/CC) regardless of compiler
  - Links in libraries automatically if module is loaded, eg MPI, Libsci, etc.
  - Always use the wrapper!
There are three compiler suites on Zeus:

- GNU, Intel and PGI
  - Fortran, C and C++ compilers
  - Different locations, include paths, libraries

- All support OpenMP

- MPI and numerical libraries modules need to be loaded separately

- Default: gcc/4.8.5

- Switch compilers easily
  
  module load gcc
  module swap gcc intel
  module swap intel pgi
## Compiler Comparison

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCE (Cray)</td>
<td>• Fortran</td>
<td>• C++</td>
</tr>
<tr>
<td></td>
<td>• PGAS (CAF &amp; UPC)</td>
<td>• No inline assembly support</td>
</tr>
<tr>
<td></td>
<td>• Vectorization</td>
<td>• Pedantic</td>
</tr>
<tr>
<td></td>
<td>• OpenMP 4.5 &amp; OpenACC 2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Integration with Cray Tools</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Best bug turnaround time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Pedantic</td>
<td></td>
</tr>
<tr>
<td>GNU</td>
<td>• C++</td>
<td>• Vectorization</td>
</tr>
<tr>
<td></td>
<td>• Scalar optimization</td>
<td>• Fortran</td>
</tr>
<tr>
<td></td>
<td>• Universal availability</td>
<td></td>
</tr>
<tr>
<td>Intel</td>
<td>• Scalar optimization</td>
<td>• Threading/ALPS</td>
</tr>
<tr>
<td></td>
<td>• Vectorization</td>
<td>• PGAS (CAF &amp; UPC)</td>
</tr>
<tr>
<td></td>
<td>• Intel MKL</td>
<td></td>
</tr>
<tr>
<td>PGI</td>
<td>• OpenMP 4.5 &amp; OpenACC 2.5</td>
<td>• Vectorization</td>
</tr>
<tr>
<td></td>
<td>• Scalar optimizations</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Which Compiler?

• No compiler is superior for all codes

• Some codes rely on specific compiler behaviour

• **We recommend to always compile your code against all available compiler suites**
  • Performance differences
  • Good way of catching bugs, improving portability and standard-conformance, find optimization opportunities
  • Don’t try to optimise your code until you know it is portable!
# Compiler Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Cray</th>
<th>Intel</th>
<th>GNU</th>
<th>PGI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization</td>
<td>-03</td>
<td>-03</td>
<td>-03</td>
<td>-03</td>
</tr>
<tr>
<td>OpenMP</td>
<td>-h omp</td>
<td>-qopenmp</td>
<td>-fopenmp</td>
<td>-mp</td>
</tr>
<tr>
<td>Pre-processes</td>
<td>-eZ</td>
<td>-fpp</td>
<td>-cpp</td>
<td>-Mcpp</td>
</tr>
<tr>
<td>Module location</td>
<td>-e m -J &lt;dir&gt;</td>
<td>-module &lt;dir&gt;</td>
<td>-M&lt;dir&gt;</td>
<td>-module &lt;dir&gt;</td>
</tr>
<tr>
<td>Debugging</td>
<td>-g</td>
<td>-G0</td>
<td>-eD</td>
<td>-g</td>
</tr>
<tr>
<td>Floating point</td>
<td>-h fp&lt;n&gt;</td>
<td>-fp-model &lt;key&gt;</td>
<td>-float-store</td>
<td>-Mieee, -Mnoieee, -Mfprelaxed</td>
</tr>
<tr>
<td>IPA</td>
<td>-h ipa&lt;n&gt;</td>
<td>-ipo&lt;n&gt;</td>
<td>-flto=&lt;n&gt;</td>
<td>-Mipa</td>
</tr>
<tr>
<td>Zero uninitialized</td>
<td>-e 0</td>
<td>-zero</td>
<td>-finit-local-zero</td>
<td>Not implemented</td>
</tr>
<tr>
<td>Real promotion (Fortran)</td>
<td>-s real64</td>
<td>-r8</td>
<td>-fdefault-double-8</td>
<td>-r8</td>
</tr>
<tr>
<td>Integer promotion (Fortran)</td>
<td>-s integer64</td>
<td>-i8</td>
<td>-fdefault-integer-8</td>
<td>-i8</td>
</tr>
</tbody>
</table>
## Compiler Flag Suggestions

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Recommended Flags</th>
<th>Compiler Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray</td>
<td>-03  -hfp3 (this is default)</td>
<td>-rm (Fortran) -hlist=m (C/C++)</td>
</tr>
<tr>
<td>GNU</td>
<td>-03  -ffast-math -funroll-loops</td>
<td>-ftree-vectorizer-verbose=2</td>
</tr>
<tr>
<td>Intel</td>
<td>-02  -ipo</td>
<td>-fcode-asm</td>
</tr>
<tr>
<td>PGI</td>
<td>-fast  -Mipa=fast</td>
<td>-Mlist</td>
</tr>
</tbody>
</table>

**Remarks:**
- OpenMP: Enabled by default for Cray, disabled by default with Intel, GNU and PGI
- MPI: Enabled by default for all compilers on Cray systems
Compiling GPU code - CUDA

Compiling CUDA codes on Galaxy

- Setup the environment: `module load craype-accel-nvidia35`
- Compile & link the code (C example):
  - `> nvcc -c -o cuda_code.o cuda_code.cu`
  - `> cc cuda_code.o host_code.c`

Compiling CUDA codes on Zeus:

- Setup the environment: `module load cuda gcc`
- Compile & link the code (C example):
  - `> gcc -c host_code.c`
  - `> nvcc -c cuda_code.cu`
  - `> gcc cuda_code.o host_code.o -lcudart`
Compiling OpenACC codes on Galaxy

• Setup the environment: `module load craype-accel-nvidia35`
• Compile & link the code (C example):
  
  ```
  > cc -h acc acc_code.c
  ```

Compiling OpenACC codes on Zeus:

• Setup the environment: `module load pgi`
• Compile & link the code (C example):
  
  ```
  > pgcc -acc acc_code.c
  ```
A few notes on Makefiles

• Automated build
  • Consistency, reproducible, transportable

• Efficient
  • Only recompile changes
  • Handles dependencies

• Parallel build support

• Easy to use
Makefile rules

- A Makefile tells make what to do
- Makefiles are composed of rules
- Order is not important except for the default target
- The default target is the first in the makefile

```
makefile targets: prerequisites
    command1
    command2
```

- A rule says if the **prerequisite** is out of date or does not exist then create the **target** with this **command**
- The whitespace before a command is a tab
- Special macros can be used for defining general rules for groups of source files
Makefile example

CC = icc
CFLAGS = -O2 -g

all: darts

darts: darts.o
   ${CC} ${CFLAGS} -o darts darts.o

%.o: %.c lcgenerator.h
   $(CC) $(CFLAGS) -c $< -o @$

clean: rm -rf *.o darts
More Options

- Makefiles can do a lot more than this
- Other options also exist
  - autoconf & automake
  - cmake

- Compiling on login nodes
  - Generally safe (CPU intensive rather than memory)
  - Large projects may require batch build
Exercise: serial darts

- Start with the serial version of Darts program: `cd darts/serial/c`

- Adapt Makefile to run on Magnus
  - Open Makefile for editing with nano, vim or any other editor
  - Change icc compilers to Cray wrappers: `CC = cc`

- Compile the code
  - Switch to Intel environment: `module swap PrgEnv-cray PrgEnv-intel`
  - Compile: `make`

- Adapt SLURM script
  - Open job.slurm file for editing with nano, vim or any other editor
  - Things to check: scratch, group, number of tasks, loading appropriate PrgEnv

- Run serial batch job: `sbatch job.slurm`
  - Linux time command can be added to note runtime, e.g.: `time srun --export=all -n 1 ./darts`
LIBRARIES
Scientific Libraries

Scientific libraries available on Pawsey’s supercomputers:

- Cray LibSci
- Intel MKL
- GSL (GNU Scientific Libraries)
- FFTW
- PETSc, Trilinos
- Boost
- NumPy
Cray LibSci

• Available on Cray systems

• The general components of Cray LibSci are:
  • Dense Solvers
    • BLAS, BLACS, LAPACK, ScaLAPACK, IRT
  • Sparse Solvers
    • PETSc, Trilinos
  • FFT
    • FFTW interface

• Tuned for processor
• Tuned for interconnect
• Adaptive and auto-tuned
Cray LibSci - how to use?

• Version provided for all compilers (Cray, Intel, GNU)

• cray-libsci module loaded by default
  • Will link in automatically

• Will select the appropriate serial/threaded version depending on context
  • If OpenMP is enabled or not
  • Inside a parallel region or not

• `OMP_NUM_THREADS` controls threading

• Can force single threaded version
  • `-l sci_cray`
  • `-l sci_intel`
  • `-l sci_gnu`
Intel MKL

- Available on Magnus, Galaxy and Zeus

- The general components of MKL are:
  - Linear Algebra
    - BLAS, LAPACK, Sparse Solvers, ScaLAPACK
  - FFT
    - FFTW, Cluster FFT
  - RNG
    - Congruent, Mersenne Twister
  - Statistics, Data Fitting, Vector Math

- Highly optimized for Intel architectures
Intel MKL - how to use on Cray?

- On Cray systems:
  - Fully compatible with PrgEnv-intel
    
    module swap PrgEnv-cray PrgEnv-intel
  
  - Single threaded MKL compatible with PrgEnv-cray
    
    module load intel
  
  - Must unload cray-libsci
    
    module unload cray-libsci
  
  - Can use -mkl, -mkl=parallel|sequential
    
    * Do not use -mkl=cluster
  
- **Note**: there are different versions of Intel MKL BLACS for different MPI implementations (use libmkl_blacs_intelmpi on Cray)
Intel MKL - how to use on Zeus?

• On Zeus intel-mkl is not loaded with the intel compiler
  module load intel

• Also, the intel-mkl module can be used along with non-Intel compiler suites:
  module load gcc
  module load intel-mkl


• **Note**: there are different versions of Intel MKL BLACS for different MPI implementations
NumPy

- NumPy is the fundamental package for scientific computing with Python.

- The general components of NumPy are:
  - a powerful N-dimensional array object
  - sophisticated (broadcasting) functions
  - tools for integrating C/C++ and Fortran code
  - useful linear algebra, Fourier transform, and random number capabilities

- Using NumPy:
  module load python
  module load numpy
I/O libraries

Application

MPI I/O

POSIX I/O

Lustre Filesystem

HDF5
NetCDF
ADIOS
I/O libraries

- I/O libraries available on Pawsey’s supercomputers:
  - HDF5 - Hierarchical Data Format
  - NetCDF - Network Common Data Format
  - ADIOS - Adaptable IO System

- Key features:
  - manage program I/O for large, complex and heterogeneous data
  - increase performance by using parallel I/O implementation
  - simple, flexible way for scientists to describe the data in their code
I/O libraries – how to use?

<table>
<thead>
<tr>
<th>I/O library</th>
<th>Cray systems</th>
<th>Zeus</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDF5</td>
<td>module load cray-hdf5&lt;br&gt;module load cray-hdf5-parallel&lt;br&gt;module load h5py</td>
<td>module load hdf5&lt;br&gt;module load h5py</td>
</tr>
<tr>
<td>NetCDF</td>
<td>module load cray-netcdf&lt;br&gt;module load cray-parallel-netcdf&lt;br&gt;module load cray-netcdf-hdf5parallel</td>
<td>module load netcdf</td>
</tr>
<tr>
<td>Adios</td>
<td>module load adios</td>
<td>-</td>
</tr>
</tbody>
</table>
Exercise: Intel MKL darts

- Work with the MKL version of Darts program:
  \( \text{cd darts/mkl/c} \)

- Adapt Makefile to run on Magnus
  - Things to check: compiler, flags, linking with MKL libraries

- Compile with MKL
  - Things to check: loading Intel MKL, unloading LibSci

- Adapt SLURM script used for serial Darts program
  - Things to check: loading Intel MKL, unloading LibSci

- Run MKL program (batch or interactive)
  - Note runtime
Exercise: Python darts

• Work with the Python + NumPy version of Darts program:
  cd darts/serial/python

• Adapt SLURM script used for serial Darts program
  • Things to check: load Python and Numpy modules in the queueing script

• Run Python program (batch or interactive)
  • Note runtime
Optimising Throughput (Outcomes)

It is **easier** for the scheduler to fit short jobs in.
- Break jobs up (checkpoint and restart)
- Beware of making jobs *too* small – if the machine is busy you could spend a lot of time sitting in the queue to run each small job.
- Pawsey queues have time limits (typically 12-24 hours).

The queuing system can be used for parallelisation.
- Run small serial simulations independently simultaneously. This minimises parallel overheads. E.g. Parameter sweeps.

Avoid / minimise human interaction.
Job dependencies

- Useful tool for creating advanced workflows
- Supported between jobs, job arrays, array elements
- Not between jobs on different clusters

#SBATCH --dependency=type:jobid,...
## Job dependencies

<table>
<thead>
<tr>
<th>Dependency List</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>after:jobid</td>
<td>Begin after the listed job has begun execution</td>
</tr>
<tr>
<td>afterany:jobid</td>
<td>Begin after the listed job has terminated</td>
</tr>
<tr>
<td>afternotok:jobid</td>
<td>Begin after the listed job has terminated in a failed state</td>
</tr>
<tr>
<td>afterok:jobid</td>
<td>Begin after the listed job has successfully executed</td>
</tr>
<tr>
<td>singleton</td>
<td>Begin after any job of the same job name and user has terminated</td>
</tr>
</tbody>
</table>

- Multiple jobids allowed, eg jobid:jobid
- Job array elements referenced as jobid_index
- Jobs that are requeued after failure treated the same
Chaining Jobs

• Submit the next job from within a batch job at the start or the end of the job

• Useful when running jobs across clusters
Example*: Chaining Jobs

- Example: `slurm/chain/job.slurm`

```bash
#!/bin/bash
#SBATCH --account=courses01
#SBATCH --nodes=1
#SBATCH --time=00:05:00
#SBATCH --export=NONE

: ${job_number}="1"
my_job_number=${job_number}
job_number_max=5

echo "running job ${SLURM_JOB_ID}"

if [[ ${job_number} -lt ${job_number_max} ]]
then
  (( job_number++ ))
  next_jobid=$(sbatch --export=job_number=${job_number} -d afterok:${SLURM_JOB_ID} job.slurm | awk '{print $4}')
  echo "submitted ${next_jobid}"
fi

echo "doing some computations for " ${my_job_number} " seconds"
sleep ${my_job_number}
echo "${SLURM_JOB_ID} done"
```

* Based on NERSC example: http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts/*
Example: Data staging

```bash
#!/bin/bash --login
#SBATCH --partition=workq
#SBATCH --ntasks=1
#SBATCH --account=pawsey0001
#SBATCH --time=00:05:00
#SBATCH --export=NONE

# run simulation
export OMP_NUM_THREADS=24
srun --export=all -n 1 -c 24 hostname

# transfer results
sbatch -M zeus --partition=copyq script-data-copy.sh
```
Job Arrays

- A mechanism for submitting collections of jobs
- Running the same program on many different data sets
- More efficient than submitting lots of individual jobs
- User defined range of elements
  - 0,1,2,3
  - 0-9
  - 0-9,20-29

- \#SBATCH --array=<indexes>

- Maximum number of elements is 1000
- Identify which index: $SLURM_ARRAY_TASK_ID
- Overall job: $SLURM_ARRAY_JOB_ID
Example: Job Arrays

- Example: `slurm/jobarray/job.slurm`

```bash
#!/bin/bash --login

#SBATCH --array=8,16,32
#SBATCH --output=array-%j.out
#SBATCH --nodes=1
#SBATCH --time=00:01:00
#SBATCH --account=pawsey0001
#SBATCH --export=NONE

time srun -n 24 --export=all ./darts-mpi $SLURM_ARRAY_TASK_ID
```
Handling Job Failures

• Control if the batch job restarts after node failure
• Cluster default is to re-queue after failure
  #SBATCH --requeue
  #SBATCH --no-requeue

• Mail notification with --mail-type=REQUEUE
• Can control runtime behaviour by checking SLURM_RESTART_COUNT in batch script
• Job submitted to SLURM (resource request)
• Once allocated, a copy of the job script is executed serially on the lowest-numbered node allocated by SLURM
• The command `srun` is used to launch multiple instances of an executable and run them in parallel
Example: MPI/OpenMP job

- 2 MPI processes/node, 12 OpenMP threads

```bash
#!/bin/bash

#SBATCH --nodes=2
#SBATCH --account=courses01
#SBATCH --time=00:05:00
#SBATCH --export=NONE

export OMP_NUM_THREADS=12
export OMP_PLACES=cores
export OMP_PROC_BIND=close

srun --export=all -n 4 -c $OMP_NUM_THREADS ./a.out
```
Example: GPU job

- 1 MPI processes/node, CUDA

```bash
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --account=courses01
#SBATCH --time=00:05:00
#SBATCH --partition=gpuq
#SBATCH --gres=gpu:4
#SBATCH --constraint=p100
#SBATCH --export=NONE

module load cuda  # on zeus
# module load cudatoolkit  # on galaxy

srun --export=all -n 1 ./a.out
```
Example: job packing

- Multiple serial programs sharing a single node

```bash
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --account=courses01
#SBATCH --job-name=packing
#SBATCH --time=00:05:00
#SBATCH --export=NONE
srun --export=all -N 2 -n 48 ./wrapper.sh
```

- Wrapper script:

```bash
#!/bin/bash
INFILE="input-$SLURM_PROCID"
OUTFILE="output-$SLURM_PROCID.out"
./serial-code.x < $INFILE > $OUTFILE
```
Thread-affinity

- It is important to understand task and thread placement of a given application within a compute node
  - memory access
  - task/thread synchronization

- How to control thread placement?
  - `-c CPUS_PER_TASK` - srun option to set number of threads per each task
  - `OMP_PLACES` - defines a series of places to which the threads are assigned, e.g. sockets, cores, threads
  - `OMP_PROC_BIND` - describes how threads are bound to OpenMP places, e.g. close, spread, master

- The same settings are used on all allocated nodes
Hyper-Threading

- On Cray systems each NUMA domain contains 12 cores (2 hyperthreads each)
- The numbering of the ‘actual cores’ is from 0 to 23

- **Note**: Hyper-Threading is disabled by default

- To use Hyper-Threading on Cray systems: `srun --hint=multithread`
- The numbering of the hyperthreads is from 24 to 47
An example hybrid MPI+OpenMP C program which prints the process/thread affinity information

- Might be used as a test program

Hello from rank 0, thread 0, on nid00010. (core affinity = 0)
Hello from rank 0, thread 1, on nid00010. (core affinity = 1)
Hello from rank 0, thread 2, on nid00010. (core affinity = 2)
Hello from rank 0, thread 3, on nid00010. (core affinity = 3)
Hello from rank 1, thread 0, on nid00010. (core affinity = 12)
Hello from rank 1, thread 1, on nid00010. (core affinity = 13)
Hello from rank 1, thread 2, on nid00010. (core affinity = 14)
Hello from rank 1, thread 3, on nid00010. (core affinity = 15)
Hello from rank 2, thread 0, on nid00010. (core affinity = 4)
Hello from rank 2, thread 1, on nid00010. (core affinity = 5)
Hello from rank 2, thread 2, on nid00010. (core affinity = 6)
Hello from rank 2, thread 3, on nid00010. (core affinity = 7)
Hello from rank 3, thread 0, on nid00010. (core affinity = 16)
Hello from rank 3, thread 1, on nid00010. (core affinity = 17)
Hello from rank 3, thread 2, on nid00010. (core affinity = 18)
Hello from rank 3, thread 3, on nid00010. (core affinity = 19)
...

Xthi code
Exercise: thread-affinity

- Work with the hybrid MPI/OpenMP xthi program
  cd xthi

- Compile the code
  make
Exercise: thread-affinity

- Adapt SLURM script:

  A. Run 6 MPI tasks with 4 OpenMP threads each, no changes to the script – default placement

  B. Run 6 MPI tasks with 4 OpenMP threads each, modify/add following lines:

      ```
      export OMP_NUM_THREADS=4
      export OMP_PLACES=cores
      export OMP_PROC_BIND=close
      srun --export=all -n 6 -c 4 ./xthi
      ```

  C. Run 1 MPI tasks with 4 OpenMP threads, modify/add following lines:

      ```
      export OMP_NUM_THREADS=4
      export OMP_PLACES=cores
      export OMP_PROC_BIND=spread
      srun --export=all -n 1 -c 24 ./xthi
      ```
LUSTRE FILESYSTEM
What is Lustre

- Distributed file system
- Fault tolerant and highly available
- Scalable and high performance
- POSIX compliant
- Supports locking
- Supports quotas
Lustre Architecture

Clients

MDS

OST - Object Storage Server
write(), read(), seek(), flock()

MDT - Meta-Data Server
open(), close(), stat(), unlink()

OST - Object Storage Target
• the backend disks for the OSS
• contain a number of binary objects representing the data for files in Lustre

MDT - Meta-Data Target
• the backend disk for the MDS
• contain information about a Lustre file system’s directory structure and file names, permissions, extended attributes, and file layouts
Lustre File systems

On Magnus and Zeus

<table>
<thead>
<tr>
<th>Filesystem</th>
<th>Type</th>
<th>Size</th>
<th>User quota</th>
<th>Group quota</th>
<th>Flush Policy</th>
<th>Backup</th>
</tr>
</thead>
<tbody>
<tr>
<td>/scratch</td>
<td>Lustre</td>
<td>3 PB</td>
<td>-</td>
<td>-</td>
<td>30 days</td>
<td>No</td>
</tr>
<tr>
<td>/group</td>
<td>Lustre</td>
<td>3 PB</td>
<td>-</td>
<td>1 TB</td>
<td>-</td>
<td>No</td>
</tr>
</tbody>
</table>

- **/scratch**
  - Fast, run your jobs here under /scratch/projectid/userid
  - POSIX compliance and multi-node access
  - **No quota, files will be deleted after 30 days**
  - Data should be moved to other filesystems after job completion

- **/group**
  - Share common files amongst a project group
  - Less performance than /scratch
  - Copy to/from /scratch as required
Striping

- Will distribute files across OSTs
- The default is not to stripe
  - This is the users responsibility
- Striping can improve performance
  - Small files should not be striped
  - For large files, set stripe between 1-4
  - Can tune stripe size – default 1MB
- Can set a default stripe for a file or a directory
  - Files and sub-directories inherit stripe settings from their parent
  - Only effective when a new file is created
Exercise: Lustre filesystem

- Work with the python examples:
  \texttt{cd lustre/}

- Adapt SLURM script to run the example
  - Things to check: path to examples

- Check the performance difference
ADDITIONAL HELP
Pawsey provides extensive documentation:

https://support.pawsey.org.au

- System user guides
- Frequently asked questions
- Pawsey-supported software list
- Maintenance logs
- Policies and terms of use
Assistance

For further assistance, contact the help desk:

•  [https://support.pawsey.org.au/](https://support.pawsey.org.au/)

Help us to help you. Provide details!

E.g.

• Which resource
• Error messages
• Location of files
• SLURM job id
• Your username and IP address if having login issues
• Never tell us (or anyone) your password!
Applications Support Team

Team expertise:

• Access to Pawsey facilities
• High-performance computing
• Parallel programming
• Computational science
• GPU accelerators
• Cloud computing
• Scientific visualisation
• Data-intensive computing

Contact the helpdesk