

HPC and Scientific Computing at OIST

II: Using the Clusters

Jan Moren, SCDA



OKINAWA INSTITUTE OF SCIENCE AND TECHNOLOGY GRADUATE UNIVERSITY

沖縄科学技術大学院大学

Set Up SSH

- Any OIST member can use the HPC resources. Apply here:
 - <https://scda-guide.oist.jp/request-access.html>
Select “Open Resources”
- OSX Users
 - You should already have SSH
 - Install “XQuartz” for graphics (reboot after installation)
- Windows Users
 - Install free “MobaXTerm”. Can use SSH and graphical applications.
- Linux and BSD Users
 - You already have everything.

<https://scda-guide.oist.jp/connect-clusters-redirect.html>

Let's Log In

Log in to Deigo:

```
$ ssh -X <your-id>@deigo.oist.jp
```

copy the slides, example scripts and programs to your home:

```
$ cp -r /apps/share/training/Intro .
```

Handy Tip: Avoid typing with *tab completion*:

```
$ cp -r /a<tab>/sh<tab>/t<tab>/I<tab> .
```



Press the tab key to fill in the name

Press **once** to fill in unique parts. Press **twice** to see matching alternatives. This works with directories, files, programs and parameters.

Later, **please go through** "Getting Started" and "Connecting to the Clusters" pages in the documentation

HighSci

All your jobs and storage on one page:
<https://highsci.oist.jp>

Open Hours

Ask about anything, online or in person:
[https://scda-answers.oist.jp/
books/scda-open-hours](https://scda-answers.oist.jp/books/scda-open-hours)

ask-scda

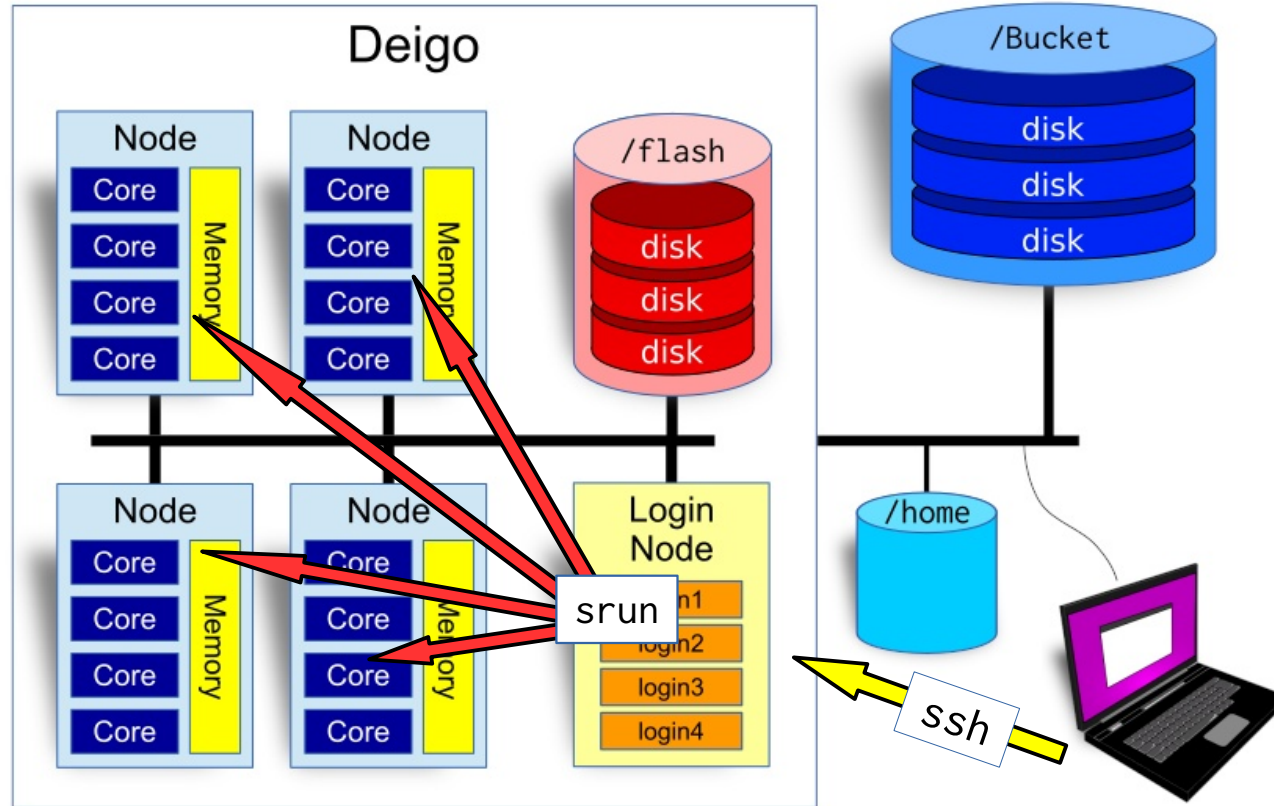
Ask questions, report problems:
ask-scda@oist.jp

SCDA

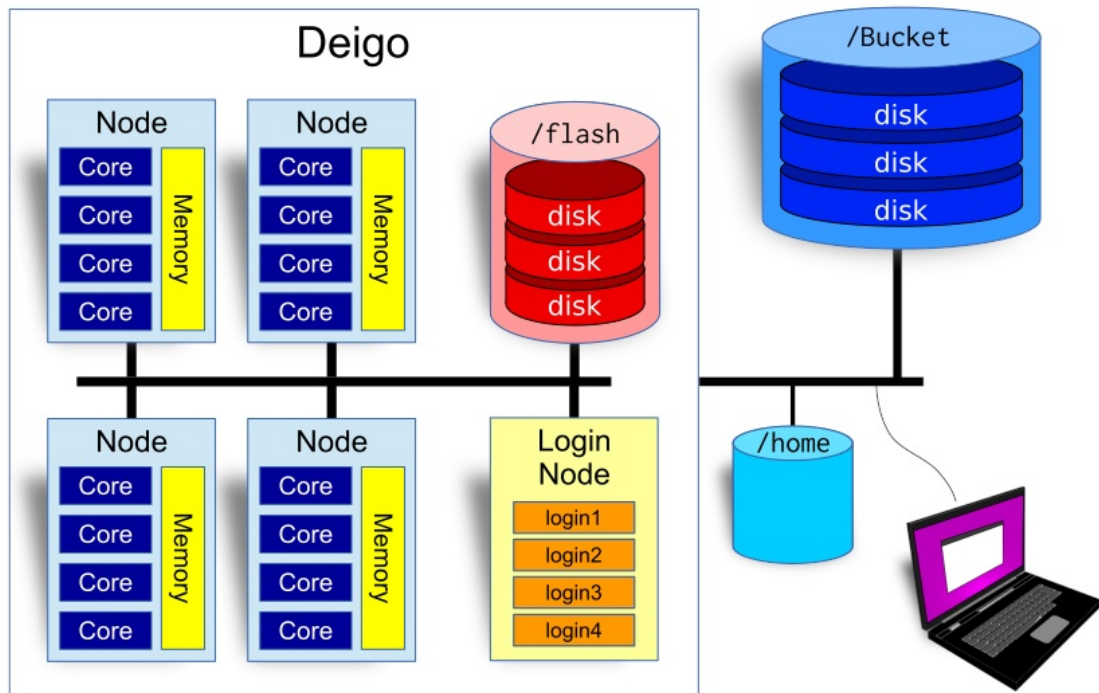
Our documentation start page:
<https://scda-guide.oist.jp/>

Using the Cluster

The Deigo cluster

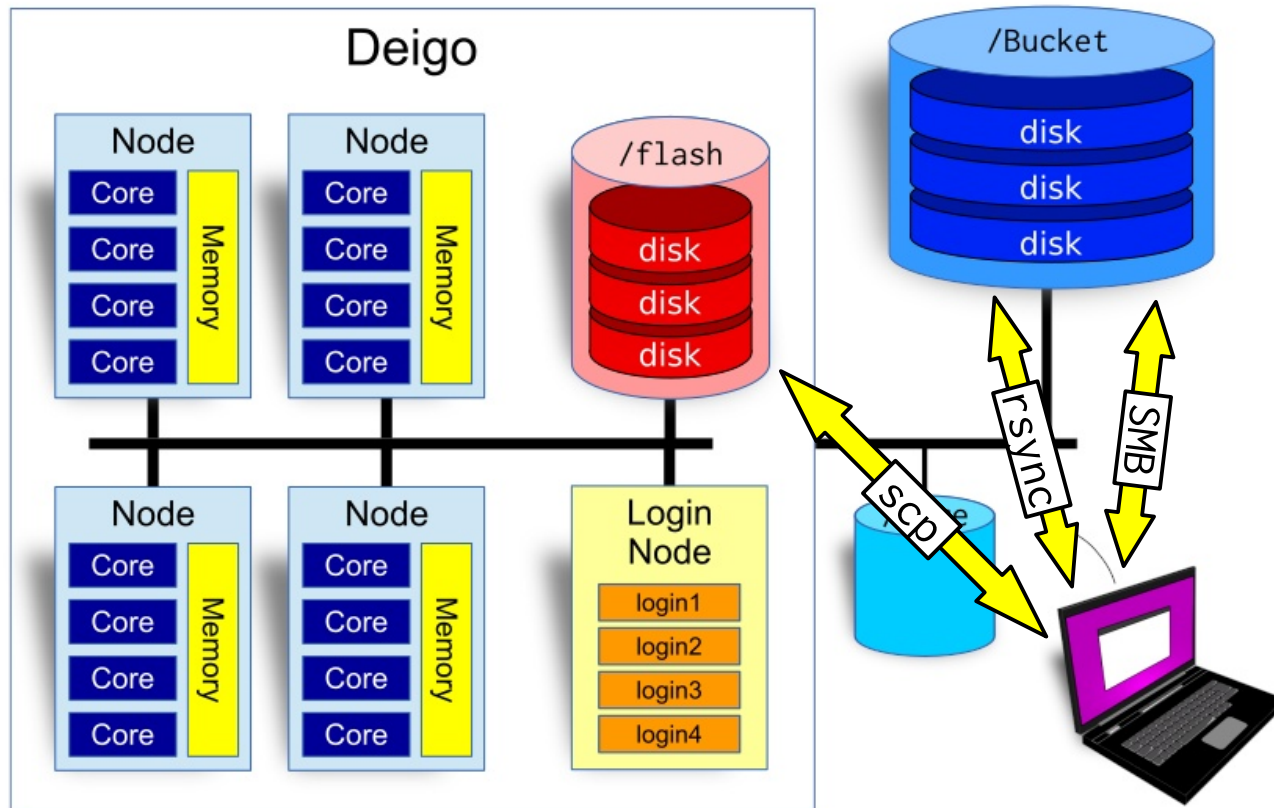


Deigo Storage



- /home Your home.
Small (<50GB), slow.
Use for: configuration files, source
- /flash In-cluster file system
Not big (10TB/unit), fast
Use for: running jobs
- /bucket Long-term storage file system
Big, backed up*
read-only from computing nodes
Use for: storing data

Moving Data



Modules

We provide most applications and versions through *modules*.

With modules we can offer you multiple versions of applications and libraries.

The 'module' command takes *subcommands* to search, list and use the modules that are installed.

module av List available modules.

```
$ module av                                # list available modules. Also 'ml av'
```

----- /apps/.metamodules81 -----		
amd-modules (L)	intel-modules	user-modules

----- /apps/.modulefiles81 -----		
BUSCO/4.1.2 (D)	cmake/3.18.1	matlab/R2019a (D)
Gaussian/09RE01R2	comsol/52	metabat/2.12.1
...	...	...

Modules

module load Load a module. Either a specific version or the latest one:

```
$ module load intel  
$ module load julia/1.3.1
```

```
$ ml intel  
$ ml julia/1.3.1
```

module list Show loaded modules:

```
$ module li  
Currently Loaded Modules:  
  1) intel/2019_update5  2) julia/1.3.1
```

```
$ ml  
Currently Loaded Modules:  
  1) intel/2019_update5  2) julia/1.3.1
```

module unload Remove one module
module purge Remove all loaded modules

```
$ module unload julia  
$ module purge  
$ module li  
No modules loaded
```

```
$ ml -julia  
$ ml purge  
$ ml  
No modules loaded
```

Modules

```
$ ml help          # command help for the 'ml' and 'module' commands
```

```
Usage: module [options] sub-command [args ...]
```

Options:

```
  -h -? -H --help          This help message
```

```
...
```

```
$ ml help julia      # find out what a module does
```

```
----- Module Specific Help for "julia/1.4.1"-----
```

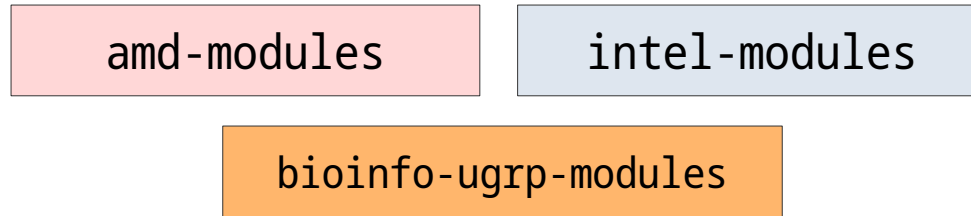
A high-level, high-performance dynamic programming language for technical computing. You can consider it a modern, high-performance replacement to MATLAB or a numerical-focused alternative to the more general Python ecosystem.

```
$ ml spider julia    # search for a module
```

```
...
```

```
$ ml key compiler    # free-form keyword search
```

Modules



Separate module collections:

- ◆ The default modules
most user software
- ◆ “amd-modules” and “intel-modules”
modules built specifically for one CPU
 - Intel compiler is in “intel-modules”
but useful for all CPUs
- ◆ “bioinfo-ugrp-modules”
user-managed
bioinformatics modules

Modules

amd-modules

intel-modules

bioinfo-ugrp-modules

use module collections by loading and unloading:

```
$ module load intel-modules
```

```
$ module av
```

```
----- /apps/.intel-modulefiles81 -----  
fftw.gcc/3.3.5          intel.mpi/2019_update5
```

```
----- /apps/.metamodules81 -----  
amd-modules          intel-modules          (L)          user-modules
```

```
----- /apps/.modulefiles81 -----  
BUSCO/3.0.2          hmmer/3.1b2  
BUSCO/4.0.6          igv/2.3.82
```

```
$ module load intel.mpi
```

Slurm

- Slurm manages all cluster hardware resources:
 - **partitions** Sets of nodes that belong together
 - **nodes** Computing nodes in the cluster, including type, size and architecture
 - **cores** CPU cores in a node
 - **memory** All memory in all nodes
 - **GPU** GPUs and other special hardware
- Slurm accepts job submissions and queues them.
- Slurm allocates resources, starts jobs and keeps track of resource usage.



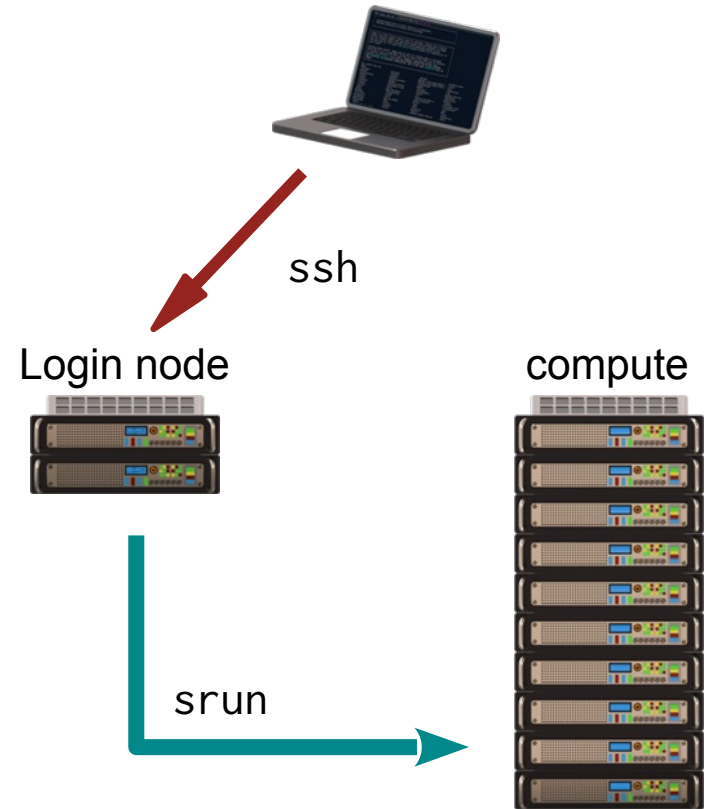
srun

Let's run a simple job:

```
$ cd Intro/code  
$ srun -p compute -t 1:00 --mem=10G -c 1 ./pi_serial
```

We give srun these options:

-p	The partition we want to use (compute)
-t	Amount of time we wanted (one minute)
--mem	Amount of memory (10 gigabytes)
-c	The number of cores (1)
./pi_serial	The program (pi_serial) in the current directory (./)



Use srun for interactive, quick jobs

srun

Let's run a simple job:

```
$ cd Intro/code  
$ srun -p compute -t 1:00 --mem=10G -c 1 ./pi_serial
```

Handy tip: If we add a '&' to the end of a command, we don't have to wait for it to finish (but we need to stay logged in):

```
$ srun -t 1:00 --mem=10G -c 1 ./pi_serial &
```

We give srun these options:

-p	The partition we want to use (compute)
-t	Amount of time we wanted (one minute)
--mem	Amount of memory (10 gigabytes)
-c	The number of cores (1)
./pi_serial	The program (pi_serial) in the current directory (./)

Use srun for interactive, quick jobs

Slurm

We can specify time and memory in a number of different ways.

Specify time:

10	10 minutes
10:00	10 minutes
5:30:00	5 hours, 30 minutes
3-12	3 days, 12 hours
1-6:30:00	1 day, 6 hours, 30 minutes

Specify memory:

500K	500 KB
1M	1 MB or 1024 KB
1G	1 GB or 1024 MB
1T	1 TB or 1024 GB

You can specify seconds, but Slurm will ignore any time resolution smaller than one minute.

Let's try it!

Try this yourself:

- Start a job with srun
- Find our latest version of the Python module
- Load the module
- Run this to test:

```
$ python3 --version  
Python 3.11.4
```

Let's try it!

*the values are
up to you*

```
# start a job:
```

```
$ srun -p compute -c 1 --mem=8G -t 0-1 --pty bash
```

```
# look at available modules:
```

```
$ module av
```

```
# load the version we want:
```

```
$ module load python/3.11.4
```

```
# see which version we have now:
```

```
$ python3 --version
```

Interactive Jobs

You can run interactive applications right on the cluster

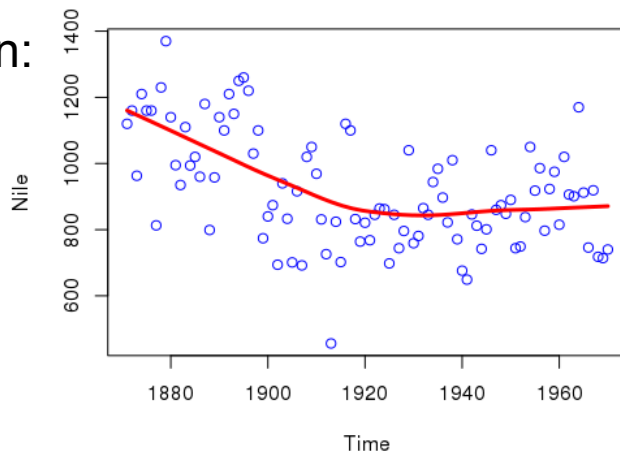
Two new options:

- x11 Allows applications to display graphics
- pty Connects keyboard and terminal directly to the application

```
$ srun -p compute -t 1:00:00 -c 8 --mem=10G --x11 --pty bash
```

Now treat your nodes and cores as your own virtual workstation:

```
$ module load R
$ R
...
> summary(Nile)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  456.0   798.5   893.5   919.4  1032.0  1370.0
> scatter.smooth(Nile, col="blue", lpars=list(col="red", lwd=3))
```



Interactive Jobs

You can run interactive applications right on the cluster

Two new options:

--x11 Allows applications to display graphics

--pty Connects

```
$ srun -t 1:00:00 -c 8 --mem=
```

Now treat your nodes and

```
$ module load R
$ R
```

```
> summary(Nile)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  456.0   798.5   893.5   919.4  1032.0  1370.0

> scatter.smooth(Nile, col="blue", lpars=list(col="red", lwd=3))
```

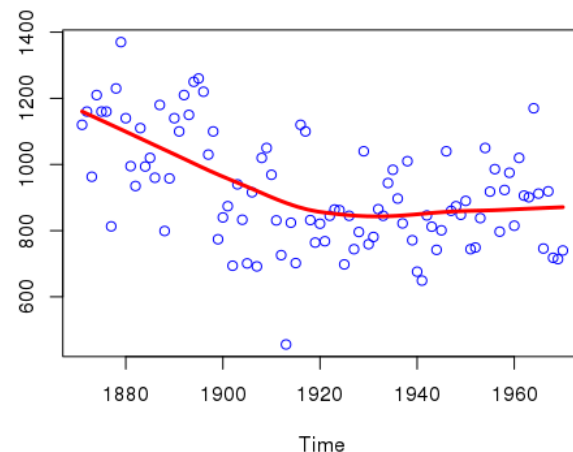
The '-X' parameter for ssh:

```
$ ssh -X name@deigo.oist.jp
```

allows remote apps to display
graphics on your local machine

On some Macs, use '-Y' instead

ation



queue and scancel

Let's see what a regular job looks like with queue:

```
$ srun -p compute -t 5:00 -c 1 -n 1 sleep 2m &  
$ queue  
      JOBID PARTITION      NAME      USER ST      TIME  NODES NODELIST(REASON)  
      637797   compute   sleep  jan-more  R      0:03     1 deigo012010
```

You can format the output in many ways — see the manpage!

Cancel the job by job ID:

```
$ scancel 637797
```

Cancel all jobs with name “sleep”:

```
$ scancel -n sleep
```

Cancel all your jobs:

```
$ scancel -u $USER      # these do the  
$ scancel -u jan-moren  # same thing
```

Combine: cancel your pending jobs:

```
$ scancel -t PENDING -u $USER
```

Seff and Ruse

measure memory, cores and time

seff

- Measures the entire job
- Ask for info after the job ends

```
$ seff 45676223
Job ID: 45676223
Cluster: deigo
User/Group: jan-moren/scicomsec
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 2
CPU Utilized: 00:00:16
CPU Efficiency: 22.22% of 00:01:12 core-walltime
Job Wall-clock time: 00:00:36
Memory Utilized: 914.26 MB
Memory Efficiency: 5.58% of 16.00 GB (16.00 GB/node)
```

ruse

- Measures a single command
- samples data every 10s
- saves info in a “.ruse” file

```
$ module load ruse
$ srun -t 0-1 --mem=16G -c 2 -p compute ruse myprog
$ cat myprog-30511.ruse
Time:          00:00:35
Memory:        914.4 MB
Cores:         2
Total_procs:   41
Active_procs:   9
Proc(%): 85.9 24.2 5.5 3.7 1.6 0.4 0.3 0.2 0.2
```

Seff and Ruse

measure memory, cores and time

seff

- Measures the entire job
- Ask for info after the job ends

```
$ seff 45676223
Job ID: 45676223
Cluster: deigo
User/Group: jan-moren/scicomsec
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 2
CPU Utilized: 00:00:16
CPU Efficiency: 22.22% of 00:01:12 core-walltime
Job Wall-clock time: 00:00:36
Memory Utilized: 914.26 MB
Memory Efficiency: 5.58% of 16.00 GB (16.00 GB/node)
```

ruse

change the sampling rate and
record change over time:

```
$ ruse --help
```

```
...
-s, --steps          Print each sample step
-t, --time=SECONDS   Sample every SECONDS
```

```
$ srun -t 5:00 -c 4 -p compute ruse -st1 ./svd.py
```

time (secs)	mem (MB)	processes tot	actv	process (sorted, %CPU)	usage			
1	545.5	7	7	33 16 15 13	2	2	2	
2	549.1	7	4	99 98 98 96				
...								
19	791.8	7	4	99 97 95 92				
20	1083.6	7	4	88 88 88 88				

```
Time:          00:00:23
Memory:        1.1 GB
Cores:         4
Total_procs:   7
Active_procs:  7
Proc(%): 95.9  94.4  93.6  91.7  0.1  0.1  0.1
```



Serial, Shared memory, and MPI jobs

We need three kind of resources:

Cores and nodes

Memory

Time

We have four kinds of jobs:

Serial

A single process using a single thread.
Part of an “embarassingly parallel” job.

Shared memory

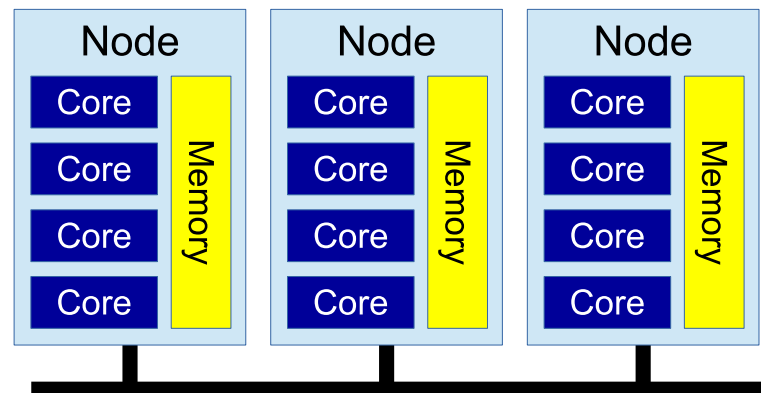
A single process with multiple threads.
Also called multithreaded. Fine-grained parallel.

MPI

Multiple processes with a single thread. Coarse-grained parallel, or distributed.

Hybrid

Multiple processes using multiple threads. A combination of shared memory and MPI.



Serial Job

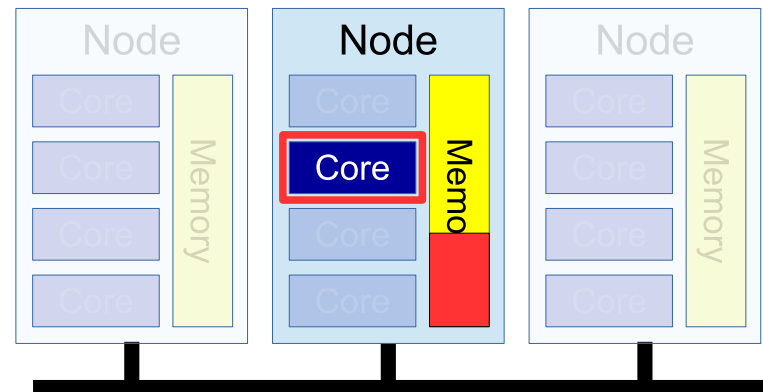
— single process, single thread

Simple data processing tasks,
copying files, some scripts

Simple Python, Matlab programs

Allocate:

- One process (called “task”)
- One core (called “cpu”)
- Per-*node* memory



--ntasks	-n	Number of tasks (processes). Set to 1
--cpus-per-task	-c	Number of cores per task. Set to 1
--mem		Memory per <i>node</i> .

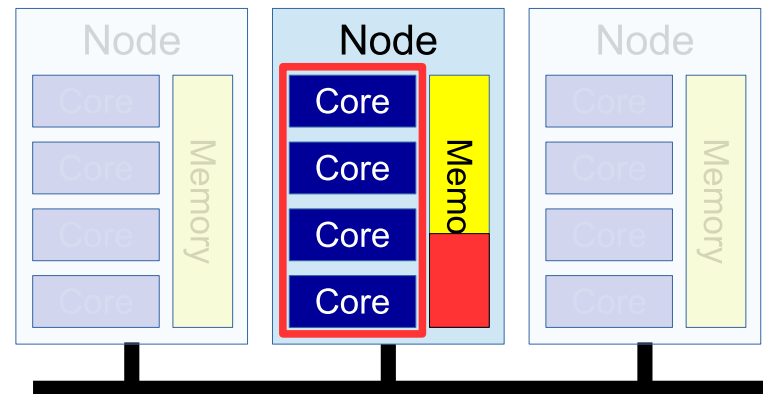
```
$ srun -p compute -t 1:00 --mem=1G -c 1 -n 1 ./pi_serial
```

Shared memory Job

— single process, multiple threads

Most scientific software.

Numerical Python, Matlab and R programs



Allocate:

- One process
- Multiple cores, up to the maximum for a node
- Per-*node* memory

--ntasks	-n	Number of tasks (processes). Set to 1
--cpus-per-task	-c	Number of cores per task.
--mem		Memory per <i>node</i> .

```
$ srun -p compute -t 1:00 --mem=1G -c 4 -n 1 ./pi_omp
```

MPI Job

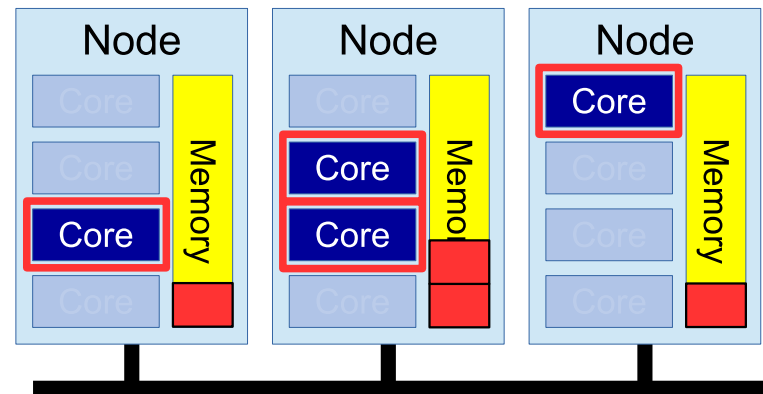
— Multiple processes, single thread

Scalable scientific software.

Simulators, MD and weather modeling, etc.

Allocate:

- Many processes
- Single core per process
- Per-core memory



<code>--ntasks</code>	<code>-n</code>	Number of tasks (processes).
<code>--cpus-per-task</code>	<code>-c</code>	Number of cores per task. Set to 1
<code>--mem-per-cpu</code>		Memory per <i>core</i> .
<code>--mpi=pmix</code>		Make Slurm start the MPI job for us

```
$ srun -p compute --mpi=pmix -t 1:00 --mem-per-cpu=1G -c 1 -n 3 ./pi_mpi
```

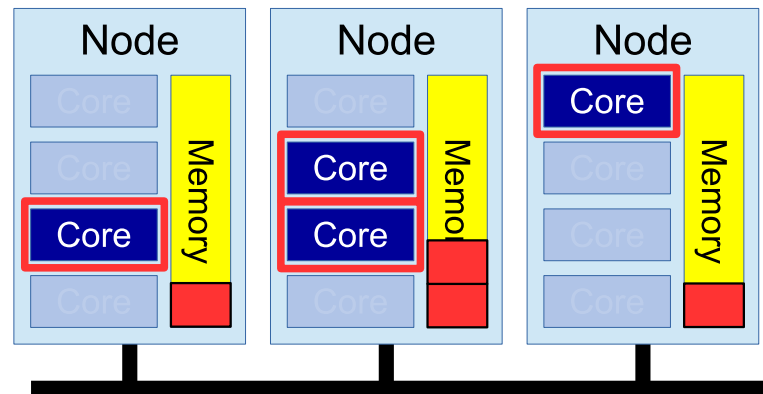
MPI Job

NOTE:

Instructions may tell you to use 'mpirun', 'mpiexec' or similar

You *can* start MPI programs that way, but there are problems scheduling them with SLURM.

→ Use '--mpi=pmix' instead if you can!



--ntasks	-n	Number of tasks (processes).
--cpus-per-task	-c	Number of cores per task. Set to 1
--mem-per-cpu		Memory per <i>core</i> .
--mpi=pmix		Make Slurm start the MPI job for us

```
$ srun -p compute --mpi=pmix -t 1:00 --mem-per-cpu=1G -c 1 -n 3 ./pi_mpi
```

Batch Jobs

A **batch job** is a command *script* that runs a job.
It's a text file with commands.
Compare with srun:

```
$ srun -p compute -t 1:00 --mem=1G -c 1 -n 1 ./pi_serial
```

batch_example.slurm

Tells the system this is a Bash script

'#' are **comments** in the bash shell

'#SBATCH' tells Slurm these are option lines

The commands we want to run

You run the script with '**sbatch**':

```
#!/bin/bash
```

```
#SBATCH -p compute
```

```
#SBATCH -t 1:00
```

```
#SBATCH --mem=1G
```

```
#SBATCH -c 1
```

```
#SBATCH -n 1
```

```
./pi_serial
```

```
$ sbatch batch_example.slurm
```



Batch Jobs

A batch script is just a text file.

- lines beginning with “#SBATCH” are job settings, **read by SLURM**
- When your job starts, each line is run as if you typed it on the command line
- Everything after “#” are comments and ignored when your job runs
- Create and edit with **nano** (simple) or **gedit** (graphical):

```
$ nano batch_example.slurm
```

```
#!/bin/bash
```

```
#SBATCH -p compute
```

```
#SBATCH -t 1:00
```

```
#SBATCH --mem=1G
```

```
#SBATCH -c 1
```

```
#SBATCH -n 1
```

```
./pi_serial
```

Tells the system this is a Bash script

- ‘#’ are **comments** in the bash shell
- ‘#SBATCH’ tells Slurm these are option lines

The shell commands we want to run

Batch Jobs

Batch jobs are the very best way to run jobs.

Benefits:

- **You can submit the job, then log out**
 - Slurm queues the job, and will start it when there are resources.
 - Slurm can email you when it's done

```
#!/bin/bash

#SBATCH -p compute
#SBATCH -t 1:00
#SBATCH --mem=1G
#SBATCH -c 1
#SBATCH -n 1

./pi_serial
```

Batch Jobs

Batch jobs are the very best way to run jobs.

Benefits:

- You can submit the job, then log out
 - Slurm queues the job, and will start it when there are resources.
 - Slurm can email you when it's done
- **The script is reliable**
 - Reuse and resubmit without forgetting any steps or typing errors
 - keep handy scripts around

```
#!/bin/bash

#SBATCH -p compute
#SBATCH -t 1:00
#SBATCH --mem=1G
#SBATCH -c 1
#SBATCH -n 1

./pi_serial
```

Batch Jobs

Batch jobs are the very best way to run jobs.

Benefits:

- You can submit the job, then log out
 - Slurm queues the job, and will start it when there are resources.
 - Slurm can email you when it's done
- The script is reliable
 - Reuse and resubmit without forgetting any steps or typing errors
 - keep handy scripts around
- **It's a record of your process**
 - Keep it in git, or store a copy with your results to keep a record of how you produced it.

```
#!/bin/bash

#SBATCH -p compute
#SBATCH -t 1:00
#SBATCH --mem=1G
#SBATCH -c 1
#SBATCH -n 1

./pi_serial
```

Batch Jobs

We have some new useful options:

--job-name	-J	Give your job a name
--output	-o	File for job output
(--error	-e	<i>File for job error output)</i>

```
#SBATCH --job-name=Myjob  
#SBATCH --output=Myjob-%j.out  
  
#SBATCH --error=Myjob-%j.err
```

- --job-name is useful to manage your job
- --output sets the output file name.
 - The default output file name is “slurm-<job ID number>.out”.
Use ‘%j’ for the job ID, and ‘%u’ for your user name.
- Normally errors go into the output file. If you set ‘--error’, they go into a separate error file instead.

Note: Normally **you should not use a separate error file**. it makes finding errors more difficult.

Batch Jobs

Get an email when your job is done:

--mail-type		send email about your job
--mail-user		your email address

```
#SBATCH --mail-type=BEGIN,FAIL  
#SBATCH --mail-user=name@oist.jp
```

- --mail-type sets when Slurm will email you:
 - BEGIN Your job began running
 - END Your job finished
 - FAIL Your job failed for some reason
 - ALL Any of different events
- --mail-user sets the email address (Only OIST email).

Batch Jobs

You can put almost any commands in the script:

Rscript.slurm

```
#!/bin/bash
#SBATCH --partition=compute
#SBATCH --job-name=R_job
#SBATCH --output=R_%j.out
#SBATCH --time=10:00
#SBATCH --mem=2G
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
```

```
module load R
```

```
echo "stats for 1e8 normal random values:"
```

```
Rscript -e "summary(rnorm(1e8,0,1))"
```

Submit with 'sbatch'

```
$ sbatch Rscript.slurm
```

Check progress with "**squeue**":

```
$ squeue
```

Tip: always put module commands in the script. That way you never forget to load the modules you need

```
module load R
```

Batch Jobs

Shared memory batch job script

pi_omp.slurm

```
#!/bin/bash
#SBATCH --partition=compute
#SBATCH --job-name=omp_job
#SBATCH --output=omp_%j.out
#SBATCH --time=10:00
#SBATCH --mem=1G
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8

echo "cores:" $OMP_NUM_THREADS
./pi_omp
```

Shared memory jobs run on a single node.

- Set the *total* memory with --mem.
- One node, one task (= one process), but we will use 8 cores.
- The OMP_NUM_THREADS variable contains the number of cores.

Handy Tip: We can set options on the command line too. They *override* the settings in the script file. So, set default values in the script, then change for each job.

```
$ sbatch pi_omp.slurm
$ sbatch -c 1 pi_omp.slurm
$ sbatch -c 16 pi_omp.slurm
```

Batch Jobs

MPI batch job script

pi_mpi.slurm

```
#!/bin/bash
#SBATCH --job-name=mpi_job
#SBATCH --partition=compute
#SBATCH --output=mpi_%j.out
#SBATCH --time=10:00
#SBATCH --mem-per-cpu=1G
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=1

module load openmpi.gcc/5.0.3
srun --mpi=pmix ./pi_mpi
```

```
$ sbatch pi_mpi.slurm
```

MPI jobs run N copies of the program, each doing part of the job.

- eight tasks, one core per task.
- Set the memory *per core*.
- With “--mpi=pmix” we let slurm start and manage our MPI task.
- **You can use srun** inside a batch job. This is called a *job step*.
We can't use the “--mpi” option in sbatch itself so we need to use a job step for mpi jobs.
Job steps are really useful for lots of things.



Batch Jobs

MPI batch job script

pi_mpi.slurm

```
#!/bin/bash
#SBATCH --job-name=mpi_job
#SBATCH --partition=compute
#SBATCH --output=mpi_%j.out
#SBATCH --time=10:00
#SBATCH --mem-per-cpu=1G
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=1

module load openmpi.gcc/5.0.3
mpirun -np $SLURM_NPROCS ./pi_mpi
```

```
$ sbatch pi_mpi.slurm
```

MPI jobs run N copies of the program, each doing part of the job.

- eight tasks, one core per task.
- Set the memory *per core*.
- With “--mpi=pmix” we let slurm start and manage our MPI task.
- **Sometimes** MPI apps won't start with Slurm
 - currently, this includes Intel MPI programs
 - You can still use “mpirun” instead



Exercise!

We have a Python program:

svd.py

Make a Slurm batch script that runs this program

1. Use an editor: nano or gedit
2. Set at least:
cores
memory
running time
partition
3. Load the python module

Start Nano like this:

```
$ nano svd.slurm
```

no mouse support, shift+arrows marks areas

^k = “control” + “k” = cut

^u = paste

^o = save

^x = exit

M-u = “alt” key plus “u” = undo

Template batch script:

```
#!/bin/bash
#SBATCH -p ???
#SBATCH -t ???
#SBATCH -c ???
#SBATCH --mem= ???
```

```
# load python module:
???
# run svd.py in current folder:
???
```

One solution:

Slurm file:

```
#!/bin/bash
#SBATCH -p compute
#SBATCH -t 10:00
#SBATCH -c 1
#SBATCH --mem=10G

# load python module
module load python/3.11.4

# run svd.py in current folder
./svd.py
```

- Time, cores and memory are *suggestions*.
 - your best guess depends on what you will be running – use “ruse” to measure it.
- Always specify the version when you load a module

How to find the optimal number of cores

1. Let's change our script

- Load the Ruse module
- Add ruse in front of your program:
`ruse ./svd.py`
- svd.py is fast, so set the sample time to 1 second:
`ruse -t 1 ./svd.py`
- Add "-C epyc" to Slurm settings
`#SBATCH -C epyc`

- *ruse* measures memory and core use every 10 seconds by default.
- Deigo has two kinds of nodes (Intel or AMD cpus). When you measure performance, always use the same type.

The script

Slurm file:

```
#!/bin/bash
#SBATCH -p compute
#SBATCH -C epyc
#SBATCH -t 10:00
#SBATCH -c 1
#SBATCH --mem=10G
```

```
module load python/3.7.3 ruse/2.0
```

```
ruse -t 1 ./svd.py
```

Quickly change the core setting on the command line:

```
$ sbatch -c 16 svd.slurm
```

-C epyc makes sure you use the AMD CPUs (use "-C xeon" for intel).

See output with "less" or "cat":

```
$ cat svd.py-12345.ruse
```

How to find the optimal number of cores

2. test the run time:

1. Run the job with one core
 - see how much memory and time it takes

2. Try with more cores

NOTE: run multiple times for each core setting

What seems to be the best core setting for this program?

With multiple runs:

- Do we measure the average?
- Or the median?
- Or pick the fastest time?

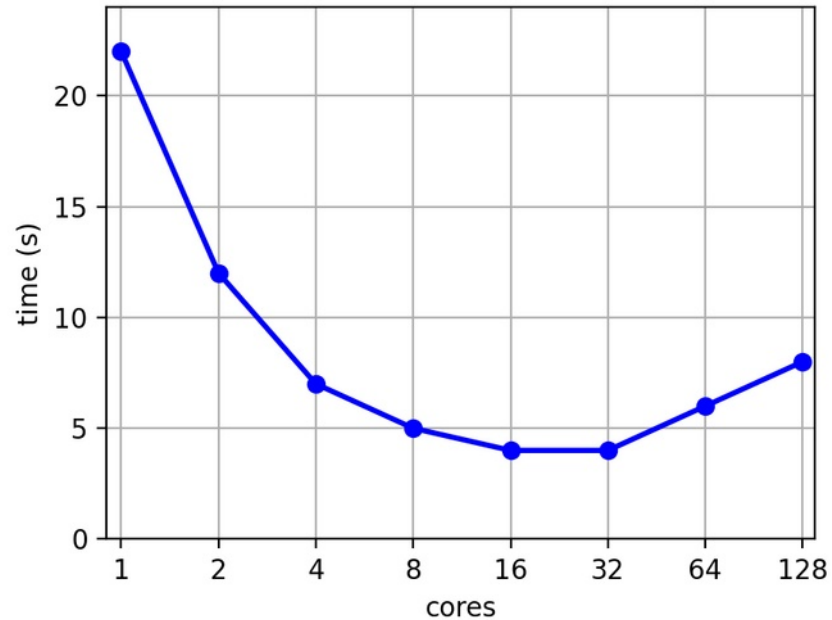
Pick best core setting:

- Pick the peak?
- The closest “nice” number?
- A bit before the peak?

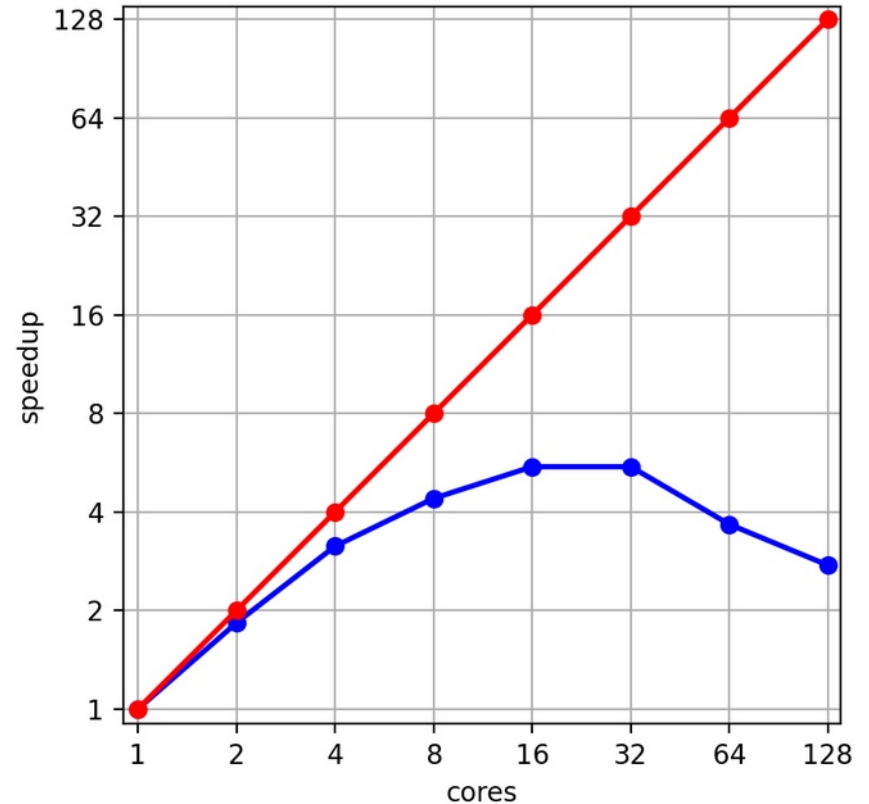
Scaling of svd.py

Min value, ten runs each

Execution time over cores



Speedup by cores (red is ideal)



Anti-patterns

The Slurm scheduler is *very* busy. Putting a high load on it is not good:

- Very short jobs (less than ~10 minutes)
- Lots of jobs in the queue (many thousands)
- Running a lot of slurm commands:
 - Running "squeue" in a loop
 - Write your own shell script to submit lots of jobs (use "array jobs" instead – ask us!)

Combine Short jobs

Combine multiple short jobs into one

- You are more efficient!
The shorter your jobs, the more you save
- Fewer Slurm activity bursts
Spend time computing, not starting or stopping
- fewer jobs in total
less strain on Slurm

```
my-program in-01.dat
```



```
my-program in-01.dat  
my-program in-02.dat  
my-program in-03.dat  
my-program in-04.dat
```

Four 4min jobs



~24 min

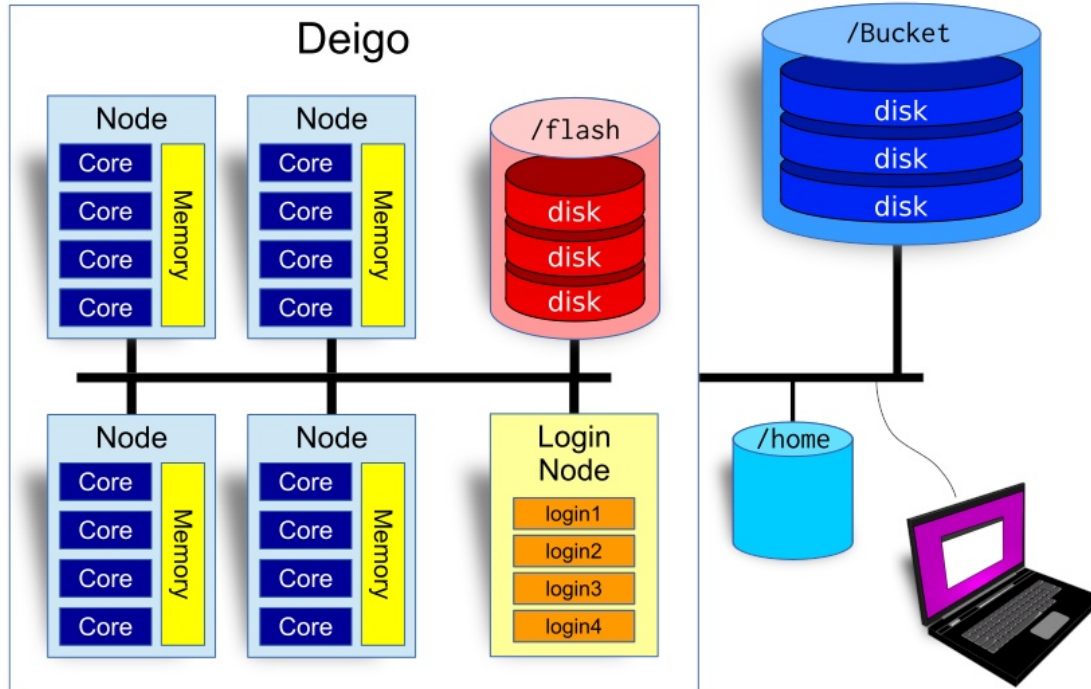
One 16min job



~18 min



Data Workflow



workflow

- Keep your research data on Bucket
- Start your job from Home or Flash:
 1. Read data from Bucket
 2. Write data to Flash
 3. Move results back to Bucket
- **Clean up:** *delete all* from Flash

Batch Job template for Deigo

job_template.slurm

```
#!/bin/bash
#SBATCH --partition compute
#SBATCH --time=0-1
#SBATCH --mem=20G

#create a temporary directory in Flash
tempdir=$(mktemp -d /flash/MyunitU/myprog.XXXX)
cd $tempdir

#run the job
./pi_serial >output.txt

#copy results to bucket, delete temp dir
scp output.txt deigo:/bucket/MyunitU/
rm -r $tempdir
```

We read data from Bucket, and save output to Flash. Finally we save the output and clean up

- Create a temporary directory in /flash
 - `mktemp -d` creates a unique directory. XXXX is replaced with random characters.
 - Save the directory name in `tempdir`
- Go to `tempdir`, then run our code there
- We use `scp` to copy our results to Bucket
 - /bucket is not writeable on compute nodes
 - `scp` copies files over the network
- Delete `tempdir` and everything inside.

Array Jobs

Array jobs run a set of jobs with different parameters.

- option ‘--array’ specifies a list of numbers:

1-10 from 1 to 10 (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

1,4-6,10 1, 4 to 6, then 10 (1, 4, 5, 6, 10)

0-16:5 every 5th value from 0 to 16 (0, 5, 10, 15)

- Slurm submits your script as one job for each index
 - Shell variable `SLURM_ARRAY_TASK_ID` is set to the current index.
 - For --output filenames, %A = job ID; %a = array index.

```
#!/bin/bash
#SBATCH --partition compute
#SBATCH --time=1:00
#SBATCH --array=1-5
#SBATCH --output=arr_%A-%a.out

echo ${SLURM_ARRAY_TASK_ID}
```

Array Jobs

We run 5 array jobs.

array_example.slurm

```
#!/bin/bash
#SBATCH --job-name=array_ex
#SBATCH --partition compute
#SBATCH --time=10:00
#SBATCH --array=1-5
#SBATCH --output=arr_%A-%a.out

echo ${SLURM_ARRAY_TASK_ID}
```

array_ex 1

```
#!/bin/bash
#SBATCH --job-name ...
...

echo 1
```

array_ex 2

```
#!/bin/bash
#SBATCH --job-name ...
...

echo 2
```

...

array_ex 5

```
#!/bin/bash
#SBATCH --job-name ...
...

echo 5
```