Hi. Welcome to this workshop on batch scripting for parallel systems. We are going to talk about writing scripts for running high performance computing applications, primarily MPI programs but there will be some scripts for threaded applications and even serial codes. In particular, we will be discussing how you might do things a little out of the ordinary, past just a simple MPI EXEC command. As you will see all of the scripts and example programs are available for download.
The purpose of this talk is primarily to help you write your own scripts. The scripts shown here can be a reference and a starting point for your scripts. All of the scripts shown here are from real life. That is they are scripts that I have used or scripts that have been written to answer a question that has been posed to us.
To Cover...

- Our test codes
- Bash useful concepts
- Basic Scripts
- Using Variables in Scripts
- Redirecting Output, getting output before a job finishes
- Getting Notifications
- Keeping a record of what you did
- Creating directories on the fly for each job
- Using local disk space

We have a lot to cover. We will start by explaining our example programs. We will concentrate on using the bash scripting language so we will have an overview of some useful concepts. Then we go into basic scripts and show how variables can be used in the scripts. The variables can be used to do output redirection. The notifications and record keeping are related and are important in research. We will discuss how to create directories on the fly which can be used even on local file systems.
People often ask if it is possible to have multiple executables within a script. We will talk about several ways that can be done.

To Cover...

- Multiple jobs on a node
- Sequential
- Multiple scripts - one node
- One Script - different MPI jobs on different cores
Sometimes you might not want to use the standard MPI mapping of tasks to nodes. This can be important for large memory per task jobs, MPMD programming, where you have different executables for various MPI tasks. This is also important for Hybrid MPI/OpenMP jobs. You will find this is sometimes required if you have heterogeneous environments. So we will discuss a method for mapping tasks to nodes that we have developed at CSM that has actually been exported to other countries. Finally, we will discuss the process of chaining jobs, having one job run after a first finishes.
The Source and Scripts

These programs vary from a glorified “Hello World” to being very complex.

We also include copies of all our scripts

http://geco.mines.edu/guide/scripts

All of the scripts that we are showing today, along with the Fortran, C, and Python source codes are available from the link shown. After you download the scripts you can use make to create the executables. The makefile assumes that the Intel Fortran, and C compilers, along with the Intel MKL library are available. The Intel compiler produces a warning on some of the programs so we use the two greater than pipe to dev null to suppress the warning. There are two versions of this talk. One for the slurm scheduler and one for PBS. The scripts are in subdirectories for each scheduler.
So we have hello world in C and Fortran MPI. Next we have serial programs that replicate MPI’s ability to get the node name and task ID. These will be used for scripts that run serial applications across multiple nodes, to show where each task ends up running.
This is the C version of info. It gets the process ID and name of the nodes. It then creates a file with a name based on the process id and writes the information to the file.
The fortran program is the same except it does not get a line from standard input. We have a subroutine that handles request to returns the name of the node we are running on.
This is the python version of info. It gets the process ID and name of the nodes. It then creates a file with a name based on the process id and writes the information to the file.
This is example output from our C program. It creates a file with a name that is a combination of the name of the machine on which it was run and the process i d. It then puts a message in the file that echoes the file name.
C MPI example

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

/************************************************************
!
This is a simple hello world program. Each processor prints out
it's rank and the size of the current MPI run (Total number of
processors).
************************************************************/

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs,mylen;
    char myname[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(myname,&mylen);
    /* print out my rank and this run's PE size*/
    printf("Hello from \\
d of \\
 on \\

\n",myid,numprocs,myname);
    MPI_Finalize();
}
```

This is hello world in MPI. Each process prints its MPI id, the total number of tasks and the name of the node in which it runs.
This is hello world in Fortran and MPI. Like the C version, each process prints its MPI id, the total number of tasks and the name of the node in which it runs.
This is a small segment of a matrix inversion program that we developed to test single core and threading performance for one of our machines. It is designed to simply do a number of inversions in a single run. It can either single or multiple threads for each inversion and/or do multiple inversion in parallel. It fills a large matrix with data then adjusts the pointer 2 d to point to a sub matrix within the matrix to invert. This particular version of the program uses the Intel MKL library for the inversion and the random number generator to fill the matrix.
Sinkfile does a file synchronization. It was designed to move files between nodes of a parallel system where file systems or the working directories not shared. This was actually developed to enable running programs out of local scratch space. It uses MPI to move data.
Batch Scripts

- Batch scripts are just that - scripts
- Run with some “shell”, bash, csh, tsh, python
- Most of the things you can do in a normal script can be done in a batch script
- Some lines in the script are comments to the shell
- Comments can have meaning to the parallel environment
- The parallel environment can define/use variables

Finally we can start to talk about batch scripting. Batch scripts, for our purposes, are just scripts that run on a machine capable of parallel applications. Scripts are interpreted programs that are run by a shell. Bash, Csh, Tsh and python are interpreted languages and shells to run the languages. We will be concentrating on bash for the rest of this talk.

So we can do most of what we can do in a batch script that we can do in a normal script. Batch scripting environments have some extensions. In particular, in batch scripting environments you can have lines that look like comments to the scripting shell but they have meaning to the parallel system. Also, there can be extra environmental variables that are defined by the system.
Bash

- Default shell on CSM machines
- Used to interact with the machine, run commands
- Bash commands can be run interactively or put in a script file
- A script file is really a “simple” Program
- List of commands
- We will use bash in our examples but other shells and scripting languages have similar capabilities
- First we discuss some features of bash


As we said, we will use bash in our examples but other shells and scripting languages have similar capabilities.

Bash is the default shell on CSM machines. The shell is used to interact with the machine in particular to run commands.

The shell runs "normal" commands and bash commands.

Bash commands can be run interactively or put in a script file that is then run.

A script file is really a “simple” List of commands or program. It can the structure of a normal program with tests, branches and loops.
We want to talk a bit about some of the syntax features in bash that we will be using. The “greater than” symbol can be used to send output from a command to a file. A double “greater than” will append output to an existing file. If you use “greater than” along with an “ampersand” then output and error information are sent to a file. The “semicolon” can be used to combine a collection of lines into a single line.
Notes on Commands

- Putting commands in ` ` returns the output of a command into a variable
- Can be use create a list with other commands such as “for loops”

```
myf90=`ls *f90`
echo $myf90
doint.f90 fourd.f90 tintel.f90 tp.f90 vect.f90
```

```
np=`expr 3 + 4`
np=`expr $PBS_NUM_NODES * 4`
np=`expr $PBS_NUM_NODES / 4`
```

The command expr with “```” can be used to do integer math.

Putting commands in back tick marks returns the output of a command into a variable. We can use this along with the expr command to do integer math.

One way we used this is to create a list with other commands. Here for example, we get a list of Fortran 90 programs. You will see in the next slide this can be used to do loops.
Here we have four types of for loops. In the first case we are looping over a list of files that was created with the `L S` command.

In the second case we are using C style indexing. For the third case we are actually incrementing over a list of items given on the command line. Note, these do not need to be numbers. Finally, we use the `SEQ` or sequence command to generate the values for our loop.
We are going to look at “if” statements and testing variables next. Before that we note that expressions can be logically grouped.

Minus A is the and operation.
Minus O allows the or operation.
The explanation mark negates the value of the expression.
The brackets do precedence changes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ! EXPR ]</td>
<td>True if EXPR is false.</td>
</tr>
<tr>
<td>[ ( EXPR ) ]</td>
<td>Returns the value of EXPR. This may be used to override the normal precedence of operators.</td>
</tr>
<tr>
<td>[ EXPR1 -a EXPR2 ]</td>
<td>True if both EXPR1 and EXPR2 are true.</td>
</tr>
<tr>
<td>[ EXPR1 -o EXPR2 ]</td>
<td>True if either EXPR1 or EXPR2 is true.</td>
</tr>
</tbody>
</table>
Here we show a few things, including testing to see if a variable is set. In this for loop we have the values 1, 2, and 3. In the first part of the loop we test i. On the iteration we unset the variable var. In the second iteration we set var to an empty string. Finally in the last iteration we set var to the string A B C D.

In the second part of the loop we do our tests. A minus n test is true if the variable is set to a nonempty string. Minus Z is true if the variable is unset or empty.

We have the results of the tests on the left.
Here we have some string tests. We can test to see if they are the same, different, greater than or less than. Note the back slash in front of the greater than and less that symbols. This is required because these are normally reserved for input and output redirection.
Here we have some string tests. We can test to see if they are the same, different, greater than or less than. Note the back slash in front of the greater than and less than symbols. This is required because these are normally reserved for input and output redirection.
### File Tests

<table>
<thead>
<tr>
<th>Test</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a FILE</td>
<td>True if FILE exists.</td>
</tr>
<tr>
<td>!b FILE</td>
<td>True if FILE exists and is a block-special file.</td>
</tr>
<tr>
<td>!e FILE</td>
<td>True if FILE exists and is a character-special file.</td>
</tr>
<tr>
<td>!f FILE</td>
<td>True if FILE exists and is a directory.</td>
</tr>
<tr>
<td>!g FILE</td>
<td>True if FILE exists and its SGIID bit is set.</td>
</tr>
<tr>
<td>!h FILE</td>
<td>True if FILE exists and its sticky bit is set.</td>
</tr>
<tr>
<td>!i FILE</td>
<td>True if FILE exists and is a regular file.</td>
</tr>
<tr>
<td>!k FILE</td>
<td>True if FILE exists and has a size greater than zero.</td>
</tr>
<tr>
<td>!m FILE</td>
<td>True if file descriptor FD is open and refers to a terminal.</td>
</tr>
<tr>
<td>!n FILE</td>
<td>True if FILE exists and its SUID (set user ID) bit is set.</td>
</tr>
<tr>
<td>!o FILE</td>
<td>True if FILE exists and is readable.</td>
</tr>
<tr>
<td>!p FILE</td>
<td>True if FILE exists and is writable.</td>
</tr>
<tr>
<td>!q FILE</td>
<td>True if FILE exists and is executable.</td>
</tr>
<tr>
<td>!r FILE</td>
<td>True if FILE exists and is owned by the effective user ID.</td>
</tr>
<tr>
<td>!s FILE</td>
<td>True if FILE exists and is owned by the effective group ID.</td>
</tr>
<tr>
<td>!t FILE</td>
<td>True if FILE exists and has been modified since it was last read.</td>
</tr>
<tr>
<td>!u FILE</td>
<td>True if FILE exists and is a symbolic link.</td>
</tr>
<tr>
<td>!v FILE</td>
<td>True if FILE exists and is a socket.</td>
</tr>
</tbody>
</table>

We can also do file tests some of the important ones are to see, if the file exists, see if it’s readable or writable, we can also test using the minus N T O T or E F options to see the relative age of files or to see if two files are actually identical.
Checking Terminal Input

```
echo "Do you want to proceed?"
echo -n "Y/N: "
read yn
if [ $yn = "y" ] || [ $yn = "Y" ]; then
    echo "You said yes"
else
    echo "You said no"
fi

Note spacing in the if statement. It is important!
```

If you have an interactive script and you want to check input this is how it is done.
Unix commands return an exit code. A normal exit or success is 0

The exit code from the previous command is stored in the variable dollar question. Dollar question can be echoed or tested with an if statement. This is often used with piping output into /dev/null “the bit bucket” when you only want to know if a command was successful. This is actually another way to determine if a file exists. You do an LS of the file and check the exit code from LS. If it is 0 the file exists.
While and with a Test and break

```
rm -f a_dummy_file
while true ; do
  ls a_dummy_file >& /dev/null
  if [ $? -eq 0 ] ; then
    echo "ls of a_dummy_file successful"
  else
    echo "ls of a_dummy_file failed"
  fi
  if [ -a a_dummy_file ] ; then
    echo "a_dummy_file exists, breaking"
    break
  else
    echo "a_dummy_file does not exist"
  fi
  touch a_dummy_file
  echo ; echo "bottom of while loop" ; echo
done
```

Finally we show the while command. In this case we’re first remove the file `a_dummy_file`. We then we use `LS` command to see if the file exist. If not, we create it. The next time through we break out of the while loop using the `break` coming
Running Batch Scripts

- A batch script is submitted to a scheduler
- PBS/torque/moab, sge, lsf, poe, slurm
- Commands to submit scripts
  - qsub, msub, bsub, poe, sbatch
- The scheduler decides where and when to run your script
  - Wait for nodes to become available
  - Wait for other jobs to finish
  - Jobs are given a name so that you can track them in the system

Finally we talk about running parallel jobs. Scheduling environments have several layers. Here we lump them all together at the highest level. To run a parallel job we submit our script to a scheduler. POE is an IBM scheduling environment. SGE or sun gridd engine is common on Rocks based systems. LSF is another common environment. Each environment has at lease one command to submit jobs. POE is the command, for the POE system. bsub works on LSF. Msub is specific to moab but has much of the same functionality as qsub. On the CSM systems we use the slurm scheduler with its command sbatch to run jobs.

The scheduler decides where and when to run your script
It waits for nodes to become available, that is it wait
for other jobs to finish. Some other user might get the
nodes before you because of policy issues or they might
have a jobs that is guaranteed to run quickly.

Another purpose of the the scheduler is to give a job a name.

Jobs are given a name so that you can track them in the system
There are a number of commands that are useful in a batch system. These are specific to Slurm but similar commands are available in other systems.

- `sbatch` is used to submit jobs.
- `scancel` cancels jobs.
- `sstat` provides detailed status report for specified job.
- `squeue` shows queued jobs.

`sjobs`, `slurmjobs`, `slurmnodes`, and `inuse` are simple wrappers that reformat information from other commands.
These are some commands which C S M has written to supplement the standard slurm commands.

S jobs, slurm jobs, slurm nodes, and in use are simple wrappers that reformat information from other commands.

Match and match split are designed to make it easy to do nonstandard mapping of MPI tasks to nodes.

phostname is designed to show the mapping of MPI tasks to nodes. The source for phostname is included in the examples distribution.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sjobs</td>
<td>Summary of running and queued jobs</td>
</tr>
<tr>
<td>slurmjobs</td>
<td>Show full information for all jobs -h for help</td>
</tr>
<tr>
<td>slurmnodes</td>
<td>Show full information for all nodes (h for help</td>
</tr>
<tr>
<td>inuse</td>
<td>Node usage by group</td>
</tr>
<tr>
<td>match</td>
<td>Creates an mpiexec &quot;appfile&quot; for MPMD runs and nonstandard mappings of tasks to nodes</td>
</tr>
<tr>
<td>match_split</td>
<td>Creates an srun &quot;multi-prog&quot; for MPMD runs and nonstandard mappings of tasks to nodes</td>
</tr>
<tr>
<td>phostname</td>
<td>Glorified MPI/OpenMP &quot;hello world&quot;</td>
</tr>
</tbody>
</table>
Running phostname with the –h option will show its input options. phostname is a good test program to determine if your script is mapping MPI tasks to nodes as you expect and to determine the environment under which your program is running.
From now on we will be talking about Slurm script with the understanding that scripts for other systems are similar.

This is a simple slurm script. We will talk about each line on the next slide. Here we want to point out that we have two types of lines in our script. We have colored them blue and black. The black lines start with a pound S BATCH. Bash sees these lines as comments. They do, however have meaning to slurm.
A Simple Slurm Script for a MPI job

```
#!/bin/bash
#SBATCH --job-name="atest"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --time=00:02:00
#SBATCH --output=stdout
#SBATCH --error=stderr
#SBATCH --export=ALL
#SBATCH --mail-type=ALL
#SBATCH --mail-user=joeuser@mines.edu

cd /home/joeuser/examples
srun -n 8 ./c_ex00
```

Our first line is telling the world that this is a bash script and it will be run using the interpreter `bin bash`. The next two lines tell Slurm that we want 1 node that we expect to have 8 cores and we want it for 2 minutes. Any output from scripts will, by default be put in a file `standard out` and error information will go to a file `standard err`. As we will see, these files unfortunately, may not become visible to the user until the job finishes. The `--export` option is important. It tells slurm that you want to run the script using the environmental variables that are defined at the time the job is submitted. Without this unexpected things can happen, usually bad.

The next two lines are used together. We have a line that gives us an email address. The other lines says we want email on actions. That is, when a job aborts, begins, and ends.

The final two lines are bash commands. When we start our we are in our home directory. So we do a `cd` to get to the directory that contains our program. Finally we use the `s run` command to run a parallel application.
What happens when you run a script?

- You are given a collection of nodes
- You are logged on to one of the nodes, the primary compute node
- Any “normal” script command only run on the primary compute node
- Extra effort must be taken to run on all nodes
- `srun` or `mpiexec` (srun for Slurm, mpiexec is used for PBS)
- Also Run only on the primary compute node
- Makes the effort to launch MPI jobs on all nodes

So, what happens when a job runs?

The scheduler gives you a collection on nodes on which to run. These are yours until the job finishes.

Next, in logs you into one of the nodes. Let's call this the job’s primary compute node.

All of your scripts run on the job’s primary compute node. Unless you make a special effort to do so, none of the other nodes in your collection will have any thing run on the.

The command `srun` or `mpiexec` launch applications in parallel, that is several copies of an application are launched on the nodes you are using. `srun` is the normal command used under slurm.

`srun` or `mpiexec` actually even runs on the job’s primary compute node. It is an extra effort command in that it launches executables on all of the nodes in your collection.
So, we have looked at basic scripts and some features of bash. We are now ready to talk about putting things together. We will start with some usage of variables in scripts.
### Slurm “script” Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
<th>Typical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLURM_SUBMIT_DIR</td>
<td>Directory for the script</td>
<td>/panfs/storage/scratch/joeuser</td>
</tr>
<tr>
<td>SLURM_JOB_USER</td>
<td>Who are you</td>
<td>joeuser</td>
</tr>
<tr>
<td>SLURM_EXPORT_ENV</td>
<td>Variables to export</td>
<td>ALL</td>
</tr>
<tr>
<td>SLURM_NNODES</td>
<td># nodes for the job</td>
<td>2</td>
</tr>
<tr>
<td>SLURM_JOBID</td>
<td>Job ID</td>
<td>11160</td>
</tr>
<tr>
<td>SLURM_NODELIST</td>
<td>Compressed list of nodes</td>
<td>node[114-115]</td>
</tr>
<tr>
<td>SLURM_SUBMIT_HOST</td>
<td>Host used to launch job</td>
<td>aun002.mines.edu</td>
</tr>
</tbody>
</table>

You can also use variables you define before you submit your script and variables defined in your environment.

When you run a script and your job starts on a compute node there are a number of new environmental variables defined. These are some of the slurm variables. SLURM_SUBMIT_DIR is set to the directory which contains the script that you are running. Typically your first command in your script will be to CD to the directory. This gets you back where you started. SLURM_JOBID is a unique name for each job that runs. It is often used to create output files that are unique to the run. That way you can rerun the same script without overwriting your previous files. SLURM_NODELIST contains a list of nodes on which your job will run. It is often useful to save this list for various purposes.

Finally, I want to point out that any environmental variable that you define before running the script can be used within the script when the job is running. We will see one use for this shortly.
Here is a list of slurm variables that were defined during a two node mpi job on the CSM platform A U N or golden. Note the variable SLURM_JOB_NODELIST. The compute node on golden are named node and SLURM_JOB_NODELIST is defined as node followed by a compressed numerical list, here 114 dash 115.
A Simple Slurm Script for a MPI job

#!/bin/bash
#SBATCH --job-name="atest"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --time=00:02:00
#SBATCH -o stdout
#SBATCH -e stderr
#SBATCH --export=ALL
#SBATCH --mail-type=ALL
#SBATCH --mail-user=joeuser@mines.edu

#---------------------

cd $SLURM_SUBMIT_DIR
srun -n 8 ./c_ex00

We go to “starting” directory
Run this MPI program on 8 cores

Recall, when a job starts it starts in your home directory not the directory that contains your script.

SLURM_SUBMIT_DIR is set to the directory which contains the script that you are running. Very often you have your data files in this directory so it is a good place to start MPI jobs.

A common first line in a script is:

C D $ SLURM_SUBMIT_DIR

This effectively gets you back where you started.
For slurm you can specify a job specific output file using the extension %j on the end of the output and error file names.

SLURM_SUBMIT_DIR is set to the directory which contains the script that you are running. Very often you have your data files in this directory so it is a good place to start MPI jobs.

A common first line in a script is

C D $ SLURM_SUBMIT_DIR

This effectively gets you back where you started.
A Simple Slurm Script for a MPI job

```bash
#!/bin/bash
#SBATCH --job-name="atest"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --time=00:02:00
#SBATCH -o stdout.%j
#SBATCH -e stderr.%j
#SBATCH --export=ALL
#SBATCH --mail-type=ALL
#SBATCH --mail-user=joeuser@mines.edu

#----------------------

cd $SLURM_SUBMIT_DIR
srun -n 8 ./c_ex00 >& myout.$SLURM_JOB_ID
```

For this script the output from the MPI program `c underscore E 0 0` will be put in a file in the working directory as the program runs. Output that is not manually redirected still will go to out and stdout `SLURM_JOB_ID`. It is possible to redirect everything to a local file but the method is very geeky so we will wait on that.
On some batch systems the job name might be a combination of text and numbers. For example under PBS JOB ID is a concatenation of a number and machine name. It is possible to create a shortened version that just contains the number. We use the command sed or, stream editor, to strip off the machine name. The string show here as the input to sed strips off everything past the first period. This is then assigned to a new variable, MY_JOB ID, that can be used in the rest of the script.
Here we have this show in a script. The output goes to a file with a name of the form my out followed by a number.
People often forget that when they do computer runs they are doing research. If they were doing a lab experiment they would be keeping records. That is just how research is done.

Here we will show some methods for keeping detailed records of jobs run automatically. We will also show how you can improve on the runtime notifications that are normally sent by the batch system.
We want to keep records of what scripts we have run.

- To be notified when a script runs
  - Under PBS
    - #PBS -M with -abe
    - Will send a notice at start and stop
  - Under Slurm
    - #SBATCH —mail-type=ALL
    - Produces information emails
  - We want more than the job number
  - We want everything

We want to keep copies of all of the script that were run. We want more than just start and end times and job names. If you have multiple jobs running this might not be enough to be informative. We want everything, copies of the script, nodes in use and so forth.
First we set the variable JOB ID to be a shortened version of Slurm Job ID.

The Slurm node list node contains a list of nodes for the run. We save our list.

The next with the, cat command, creates a copy of our run script.

We next save our environment.

Finally, we mail all of this to us with the subject of our mail being the job ID.

We can use the, SLURM JOB USER, variable to give us the address. More likely this would just be hard coded.

There is a trick required here. On most machines compute nodes can not do email. What we are doing here is using ssh to run mail on the machine management node, mindy, piping in the file we have created to be the body of the email. We actually copy the environment to a subdirectory off of our home directory. We do this because on some machines the directory from which we run a job might not be available on the management node.
This is the output from running this script. We have our environment, our node list, the output, and a copy of the run script. We have received the normal email generated from the run along with the information put in environment file. This is in the email with the subject of the job ID.
More on variables and a few other details

We are going to take an additional look at some of the ways we can use variables.
The CSM written utility /opt/utility/expands takes a Slurm style compressed node list and creates a full list similar to what is produced by PBS.

Here we specify the node list on the command line. If this command is run during a parallel job it can get the compressed node list from the environment.

Finally we pipe the output through sort with the minus u option giving us a sort list of unique node names.
#!/bin/bash
#SBATCH --job-name="hybrid"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks=1
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=00:02:00
#SBATCH -o stdout.%j
#SBATCH -e stderr.%j

# Go to the directory from which our job was launched
cd $SLURM_SUBMIT_DIR

# Create a short JOBID base on the one provided by the scheduler
JOBID=`echo $SLURM_JOBID`

echo $SLURM_JOB_NODELIST > nodes.$JOBID

export INPUT=sinput
export APP=fillmemc

/opt/utility/expands $SLURM_JOB_NODELIST > APP.$INPUT.nodes.$JOBID
cat $INPUT > APP.$INPUT.input.$JOBID
srun ./$APP < $INPUT >> APP.$INPUT.output.$JOBID

There is a lot going on in this script with environmental variables. We will talk about that next. First we talk about the green lines and the two bold ones after it. We are asking for a single node and we are going to only run a single task on it. The bold line tells the scheduler to not allow any other user access to this node, even if it thinks cores are available.
As before, we create a file containing our node list. We actually recommend people do this for debugging purposes. Sometimes there are problems with particular nodes. This helps track them down.

The other new thing here is that have our input file and the name of the application stored in a variable. There are several reasons for doing this. We will see how this makes multiple runs easier in later slides.

For now it has some advantages. It can make our script easier to read. Also, in this example, we have output files that have embedded in their name the name of our input file and application.

Just for kicks, we save our node list in this new output file. We then save a copy of our input and put the output in its own file.
What we get

We get outputs from our script the map back to the input and the application name.
Multiple Executables Tricks

• Case 1: Multiple jobs running on the same node at the same time
  • Independent
  • Launched from different scripts

• Case 2: Multiple executables running on the same node at the same time
  • Independent
  • Launched from a single script

• Either case could be serial or MPI

• Case 3: Using mpiexec or srun to launch several serial programs

It the first case we are going to look at multiple jobs running on the same node at the same time.

In the second case we have multiple executables running on the same node at the same time.

For case one, we will have a bunch of small core count jobs that we want to run on the same node and each job is launched using a different script.

In case two we have a single script but that single script launches several independent programs on the same node.

We assume here that you are somewhat familiar with basic batch scripting.

We are first going to review some of the variables that are available in your scripts. and talk about how you might use them.
In our next example we are going to show how to run multiple jobs on a single node using multiple batch submission commands and multiple scripts. More likely, we will be reusing the same script. We will show how to do this using different data sets without needing to edit the script.
Now, let’s look at some of the ways we are using variables in this script. The echo command saves a compressed list of nodes. We are using an input file and application file that are also specified in variables, INPUT and APP.

The expands command gives us the fill list of nodes which is put in a file which has an name made up of the application we are running, the input file name, and the job ID.

The final line runs our program with the input file defined as “s input” and appends the output to the file that contained our list of nodes. If we want to run this example several times on the same node we would need to edit the script and change the input file.
With the previous slide we had a script that we could run multiple times on the same node but there is a problem. If we ran the script without any changes each run would be identical, with the same input file “s input” and the same executable “fillmemc”.

In this example, we have commented out the line that sets our input file. What we are going to do is read the value for the input file variable, “INPUT” from the environment when we submit the run. So we can have multiple runs going at the same time with different input files. Before each submission we set the value on “INPUT” using an export command.
Assume we have 4 data sets and we are willing to run on any one node.

Here is what it looks like in an session where we submit multiple jobs to the same node with different inputs.

In the first line we set the input file to “s input 1”. When we submit the script with the sbatch command this is picked up and we run with this input.

The next thing we do is to “cat” or “type” the mynodes file. This tells us which node our job is running on.

Then we use export again to set our input file to “s input 2”.

Then we submit the script again. This time we add the option “–l nodes=n20” to our submission command. This will force the script to the same node.

We then repeat these steps for two more input files, “s input 3” and “s input 4”. Again we force the script to run on the same node.

Now if you have a node reserved you can use the same “–l” option to force your script to run on your reserved node.
Our output files:

Here is a list of the output files produce from this series of runs. Our execution line within our script specifies output files which are made up of the application name, the input file name and the job ID. So we get a series of output files with each one having a different input file name and a different job number. Very cool.

That’s it for this example.
Assume we have 4 data sets and we are willing to run on any one node.

Here is what it looks like in an session where we submit multiple jobs to the same node with different inputs.

In the first line we set the input file to “s input 1”. When we submit the script with the sbatch command this is picked up and we run with this input.

The next thing we do is to “cat” or “type” the mynodes file. This tells us which node our job is running on.

Then we use export again to set our input file to “s input 2”.

Then we submit the script again. This time we add the option “--nodelist=node001” to our submission command. This will force the script to the same node.

We then repeat these steps for two more input files, “s input 3” and “s input 4”. Again we force the script to run on the same node.
Our output files:

```
[joeuser@mio test]$ ls -l fillmemc.sinput*
-rw-rw-r-- 1 joeuser joeuser 2763 Sep 10 20:35 fillmemc.sinput3.output.11836
-rw-rw-r-- 1 joeuser joeuser 2716 Sep 10 20:35 fillmemc.sinput4.output.11837
-rw-rw-r-- 1 joeuser joeuser 2481 Sep 10 20:34 fillmemc.sinput2.output.11835
-rw-rw-r-- 1 joeuser joeuser 2481 Sep 10 20:34 fillmemc.sinput1.output.11834
```

Note different job numbers

```
[joeuser@aun001 memory]$ squeue -u joeuser

JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
11836 debug hybrid joeuser  R       0:10      1 node001
11837 debug hybrid joeuser  R       0:10      1 node001
11835 debug hybrid joeuser  R       0:24      1 node001
11834 debug hybrid joeuser  R       1:24      1 node001
```

Here is a list of the output files produced from this series of runs. Our execution line within our script specifies output files which are made up of the application name, the input file name and the job ID. So we get a series of output files with each one having a different input file name and a different job number. Very cool.

That’s it for this example.
Now let's look at the second case.

Here we are going to launch a collection of executable on a single node in parallel. We are going to do this by starting the programs in a loop and putting them in the background.

There can be a problem when you do this. When you put the programs in the background the script will continue to run past the all the program start points. The script will run to the end with the programs still running and exit. When the script exits you will not longer have access to the node on which you are running your job. Your programs will be killed, will hang, or the could become zombie tasks. Zombie tasks can adversely effect other programs.

So there are a number of tricks to prevent this from happening. We will show you, most likely, the simplest here.
Let’s first look at how we launch multiple programs with a for loop. We have the same program execution line from the previous example. However, in this case the “INPUT” file is set with a for statement. The loop will first set “INPUT” to “s input 1” and then start the program “fillmemc”. Next the program will be started with the input “s input 2” and so on.

The ampersand at the end to the execution line forces the program to run in the background. Without this, we would have a situation where the first program would start and the second would not start until the first had completed.

Now here is the trick to get this all to work properly. We put a “wait” command at the end of the script. This causes the run script to pause at that point until all of your programs that you launched have finished.

Very simple.
Here again is a list of our output files. In this case, since we launch all of the programs as part of the same submit command each has the same job number. We do have different input files and thus different input file names.
Some versions of mpiexec and srun will work with nonMPI programs.

- Creates specified number of copies of the program, all independent.

Some versions of MPI EXE C and MPI run will work with nonMPI programs. What happens is that MPI EXE C creates the specified number of copies of the program, all independent. The next slide has the simple script for doing this. However, you might want to add a wait at the end of this script in case the instances of the application don't finish at the same time.
This is a simple script. One of the things we have changed that we request 2 nodes and will be running 8 tasks per node. The nodes file will contain the compressed list of our two nodes. Our executable name is held in in a variable. In this case it is info p. Recall that this program creates a file which has a name that is a concatenation of the node name and the process I D.

By the way, we have C and Fortran versions of the program. We use the python version to show that we can also run scripts from within scripts.

Our version of srun supports launching serial programs. So we expect that we will get 16 copies of info p running and thus 16 output files.
Running a serial program with mpiexec

export MYPROGRAM=info_p

[joeuser@aun001 memory]$ sbatch -p debug serial
Submitted batch job 11841

[joeuser@aun001 memory]$ cat nodes.11841
node[001-002]

[joeuser@aun001 memory]$ ls -lt node00*
-rw-rw-r-- 1 joeuser joeuser 39 Sep 10 21:05 node002_00007602
-rw-rw-r-- 1 joeuser joeuser 39 Sep 10 21:05 node002_00007604
-rw-rw-r-- 1 joeuser joeuser 39 Sep 10 21:05 node002_00007607
-rw-rw-r-- 1 joeuser joeuser 39 Sep 10 21:05 node002_00007606
-rw-rw-r-- 1 joeuser joeuser 39 Sep 10 21:05 node002_00007605
-rw-rw-r-- 1 joeuser joeuser 39 Sep 10 21:05 node002_00007608
-rw-rw-r-- 1 joeuser joeuser 39 Sep 10 21:05 node002_00007609
-rw-rw-r-- 1 joeuser joeuser 40 Sep 10 21:05 node001_00026256
-rw-rw-r-- 1 joeuser joeuser 40 Sep 10 21:05 node001_00026257
-rw-rw-r-- 1 joeuser joeuser 40 Sep 10 21:05 node001_00026253
-rw-rw-r-- 1 joeuser joeuser 40 Sep 10 21:05 node001_00026255
-rw-rw-r-- 1 joeuser joeuser 40 Sep 10 21:05 node001_00026254
-rw-rw-r-- 1 joeuser joeuser 40 Sep 10 21:05 node001_00026259
-rw-rw-r-- 1 joeuser joeuser 40 Sep 10 21:05 node001_00026260
-rw-rw-r-- 1 joeuser joeuser 40 Sep 10 21:05 node001_00026258

[joeuser@aun001 memory]$ cat node001_00026256
Python says hello from 26256 on node001

We do get 16 output files from the 16 instances of the program. 8 are from one node and 8 from the other and each is given a unique name.
We have been concerned with scripting commands up to the point we start a parallel application. The parallel application is started with `MPIEXEC` or `srun`. We have not cared much about the mapping of tasks to nodes. We are going to look at that in more detail now. We are going to look at other than the default mappings of tasks to nodes.
Why would we want something other than the default mappings? The default mapping is each core gets a MPI task.

We might want less than one task per core. This would occur if we require a large memory per task or if we are doing hybrid programming with a combination of MPI and open MPI.

We might want to run different executables on various nodes, that is a MPMD program.

We might need 8 tasks on one node and 12 on another.
Method for OpenMPI and MVAPICH2

- Same method works for both versions of MPI
- Create a description of your job on the fly from inside your script
- The description is a mapping of programs to cores
- Tell mpiexec/mpirun to use your description to launch your job
- We created a utility script to make it easy

The OpenMPI and MVAPICH2 version of MPI have the ability to run programs based on a detailed description file that maps tasks to nodes. We can create this description file on the fly that maps tasks to cores. We then tell MPIEXE to use our description files.

As you will see there are some difficulties in creating the description file so we have created a script to make it easier.
Alternate syntax for mpiexec

- Normally you specify the number of MPI tasks on the mpiexec line
- The alternate syntax is to provide an “appfile”
  - `mpiexec -app appfile`
  - The appfile is a mapping of executables to nodes

Note: The normal “parallel run” command under slurm is `srun`. CSM machines AuN and Mio also support `mpiexec` under slurm. The method discussed here will work on these machines. However, there is also a slurm specific method for doing mappings that will be discussed below. It splits the node list and the application list into two separate files. An easy way to use the slurm specific method is to first create the appfile as discussed here. There is a CSM written utility to split the appfile into the two separate files.

The normal syntax for a `MPIEXEC` command is to provide the executable and the number of instances to run. There is an alternative syntax where you provide an appfile. The appfile contains our detailed mapping of tasks to nodes.
Appfile format

- Collection of lines of the form
  -host <host name> -np <number of copies to run on host> <program name>
- Specify different application names in your appfile for MPMD
- You can specify a node or program more than once

The appfile is just a collection of lines. Each line contains the name of the node on which to run, the number of tasks to run, and the path to the program to run. You can list different programs, thus run MPMD. You can also list a node and or a program more than once.
We have some examples. Examples 1 and 2 are the same. They both specify running 4 copies of my program on node compute 1 1. Examples 3 and 4 are not the same. For example 3 we have 2 copies of A or compute 1 1 and two copies of B on 2 3. In example 4 we have a copy of each program running on each of the two nodes.
There is a problem.

The difficulty is that the names of the nodes that you are assigned by the scheduler are not known until after the job is submitted. So you need to create the appfile on the fly from within your PBS script.
We have created a script "match" located at /opt/utility/match which takes a list of nodes and a list of applications to run on those nodes and creates an appfile. The match script is also included as part of the examples for this talk.

The PBS variable $PBS_NODEFILE contains the name of a file that has the list of nodes on which your job will run.

SLURM_JOB_NODELIST has a compressed list of nodes.
Solution

Given your \$PBS\_NODEFILE and a list of programs in a file app\_list the simplest usage of match is:

```
match \$PBS\_NODEFILE app\_list > appfile
mpiexec --app appfile
```

For mvapich2 replace --app with --configfile

If you have a list of applications to run in a file, say app\_list, then the one usage of match would be as shown here. We list the node file, the applications list and run match using these two as input. The output is piped into a file app file which is then used on the MPI EXEC command line.

The math script has lots of options we will look at most of them in the next few slides.
We supply match with two lists, the list of applications to run and the list of nodes on which to run. The number of instances of the application to run is determined by the length of the longer of the two lists. That is, if the lists are not the same length then multiple copies will be launched.

By the way, as you will see match can take replication counts.
Here is an example. We are running on two nodes with 8 core each. We first generate two lists. The full list contains 16 entries, 8 per node and the short list contains just the two node names.
Examples

• We have two programs we are going to play with: f_ex00 and c_ex00

• We have two program lists that we are going to use:
  • oneprogram
    • c_ex00
    • twoprograms
    • c_ex00
    • f_ex00

We also create two program lists. One contains a single program name and the other contains two programs in the list. We call these two lists one program and two programs.
Here we use the full list of nodes and the list of the two programs. This generates an app file that has 16 tasks with 4 copies of each program on each of the two nodes.
Our short list has the same number of entries and the number of programs in the file two programs. If we use these two together we get one copy of each program on a single node.
Here we use a replication count to get two copies on each node.
With a single program in our list we can specify the number of copies to put on each node. This might be useful for hybrid MPI OpenMP programs.
Match can take the names of program to run from the command line using the \(-p\) option. We note that the names of the applications must be in quotes. We then can put the number of tasks to launch on each node after the application list.
Running on Heterogeneous Nodes

- Mixed numbers of cores (8, 12, 16)
- Want to use all of the cores
- The number of cores expected on a node is ppn=N
- Could use match with a fixed core count but this might leave some open or over subscribed

```
machine shortlist -p"c_ex01" 8 8
```

- If you don’t specify the number of copies then you will be given 1 per core

```
machine shortlist -p"c_ex01"
```

Mio currently has nodes with 8 cores and some with 12, 16, or 20 cores. If you run it without giving a number of tasks to run on a node it will put one per core. This is actually difficult to do otherwise.
Slurm Specific mapping

- Slurm has several ways to specify application to node mapping
- mpiexec works on Mio and AuN
- Another way:
  - Node list and application list go in separate files, say hostlist and applist
  - To Run on 12 cores:
    
    ```
    export SLURM_HOSTFILE=hostlist
    srun -n12 --multi-prog applist
    ```

While Mio and Aun support mpiexec for Open MPI MPI jobs, the normal command used to launch jobs under slurm is srun.
Under slurm you can use the multi prog option. This is similar to app file option for mpiexec but the description is split into two files. We have a host list file and an app list file. The host list file is specified using the environmental variable slurm host file. The app list file is specified with the command line option multi prog.
The host list file lists the hosts and the app list file lists the programs. One difference is that you list the MPI task ID in the app list file.

The problem is that we have the same issues creating these files as we do with the app file under MPI exec. So we have created another script that splits an mpiexec app file to these two files.
Match split is the script that does the split. These instructions, along with usage examples, are what you get if you specify `-h` on the command line for match split.

One way to use this script is to “pipe” the output from match in to match_split. This will create the files app list and host list. You can also specify the file output from match on the command line.
Here is an example of using match split with the file match file that was previously created by the script match. We create the files match file app list and match file host list. These are then used with the s run command.
Here we have a shortlist of nodes in our jobs. In this case we pipe the output of match directly into match\_split producing the files applist and hostlist which are again used with s run.
Slurm script for match and match_split

```bash
#!/bin/bash -x
#SBATCH --job-name="match"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=16
#SBATCH --ntasks=32
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=00:00:00

# Go to the directory from which our job was launched
cd $SLURM_SUBMIT_DIR

# Create a short JOBID base on the one provided by the scheduler
JOBID=`echo $SLURM_JOBID`

echo $SLURM_JOB_NODELIST > nodes.$JOBID

# Create a shortlist of nodes
/opt/utility/expands $SLURM_JOB_NODELIST | sort -u > shortlist

# Run match to create an mpiexec appfile
/opt/utility/match shortlist -p "helloc hellof" 4 8 > appfile

# Run the job using mpiexec
mpiexec --app appfile > outone.$JOBID

# Run match_split to create a srun applist and hostlist
/opt/utility/match_split appfile applist hostlist

# Run the job using srun
export SLURM_HOSTFILE=hostlist
srun -n12 --multi-prog applist > out2.$JOBID
```

First run we use match to create and appfile and run using mpiexec

Then we create separate app list and hostlist files and run using srun

Here is a full example using both an app file with mpiexec and the app list and host list with srun. We put the outputs from our two runs in separate files. The two output files are the same.
Here we have the three files app file used with mpi exe and the files app list and host list used with s run. Again, the outputs produced using s run and mpi exe are the same.
We talked earlier about creating copies of input files, scripts, and then environment as a form of documenting your research. Next we will look at a method of creating directories to further segment your research runs.

This is actually a prerequisite for looking at how you might use disk space that is not shared between all of the tasks of an MPI application. For example we might want to use disk that is local to a node.
WE show here another way to create a unique name. We can use the date command with a format statement. The one shown here gives us a directory name based on date of the from $Y \ Y \ M \ M \ D \ D \ H \ H \ M \ M \ S \ S$, for year, month, day, hour, minute, second.

So we create the directory and $\text{C D}$ to it. We save the path to this new directory. We will use is in a future version of this script.

We then copy our data from the starting directory to the current one. Here, we assume the data is in a $\text{T G Z}$ file. After coping it we unpack it.

Finally we run our application. We give the full path to the application because it is not in our local directory.
That was easy. Now for the next part, creating directories on local space and using it. Most parallel machines, but not all have some local disk space. What do we mean by local space. It is disk space that can only be seen by some tasks, usually tasks on the same node. Local space can not be seen from the primary compute node. Local space may be faster than shared space, but not on all machines. The size and location of local space is machine dependent. On most unix based systems each node normally has a slash temp space. It is usually not a good idea to use slash temp because it is used by the system. If it fills the node can hang. If you are running on a rocks based system you will have a local disk, state partition one.
Using local directories can be a multistep process. The exact steps will change depending on what you want to do but here we:

1. Figure out where it is.
2. Create directories.
3. Copy to the new directory.
4. Compute in the local directory.
5. Copy what you want to shared storage.
6. Clean up after yourself.

One of the difficulties is, as you recall, all batch commands are actually run on the primary compute node.
Here it is...

- Figure out where local disk is
- Create a shared directory where all of the results will be copied in the end
- Get a list of nodes
- Use ssh to create a directory on each node
- Do all the "normal" saves done in other examples
- Go to the new directory (This only happens on master node.)
- Use "match" to create an appfile
- Run the application (All tasks will get launched in the same named directory)
- Use scp to copy the files to shared storage
- Clean up

Here is a slightly more detailed outline of what we will be doing in our script. We will use most of what we have talked about so far. We will be using SSH and SCP to perform remote operations from the primary compute node. Understanding the reason why we use these commands is the key to understanding this script. We use these commands because they allow us to perform operations on disks via a remote connection. SSH logs in to a node then performs a command. SCP does a copy from one node to another.
Here is the first part of our script. It is the same as the last script that creates a new directory for the run except for the lines in red.

These lines `job temp`. `job temp` is the variable that holds the path to our local disk space. These red lines test to see if `job temp` is defined in our environment. If so, it will be used. If not, we use a default value of scratch.

As we said before, and as shown in blue, we will save the path to our newly created directory for future use.
Set Up 2

Here we first get a list of nodes. In this script we will actually use this list not just print it. After we get the list of nodes we use it in out for loop. For every node run the S S H command. It logs on to the node $ I and runs the command mkdir, make directory. We give the full path to the directory to create as the concatenation of the path to our local space and the new directory name.

After the ssh command completes we are back on the primary compute node. We use S C P to copy the local files to each compute node.

The next few lines do the normal things of saving information about our job.

Finally we C D to our local disk space. Again, this happens only on the primary compute node.
Here we use the match script to create a list for MPI EXEC and we run the application. MPI EXEC launches the application on each node. But there is a bit of magic happening here. MPI EXEC knows from which directory it is run, that is it knows the full path to the directory. When it launches an application it will go to the directory with the same name on a remote machine, even though it is in reality a completely different file system.
Clean Up

```bash
for each node...
for i in $nlist
do
    # Copy files from our local space on each node back to
    # my working directory creating a subdirectory for each node.
    mkdir -p $ODIR/$i
    scp -r $USER@$i:$JOBTMP/$NEW_DIR/* $USER@aun.mines.edu:$ODIR/$i
    ### or ####
    # ssh -r $USER@$i cp -r $JOBTMP/$NEW_DIR/* $SLURM_SUBMIT_DIR/$i

    # Remove the temporary directory
    ssh $i rm -r $JOBTMP/$NEW_DIR
done
```

After our M P I job completes we again loop over each node in our list. We first create a subdirectory for each node in directory O D I R. Here is where we use this variable. Then we can use either scp to copy from each compute node to the subdirectory. Or we can in this case use ssh with a normal copy command. Why does a normal copy work? We are assuming that the directory O D I R is visible from the compute nodes.

The last ssh is very important. It is used to delete the local directories we have created.
Chaining Jobs

There are times when you might think you want to run a series of jobs one after another. We will look at that next.
Running jobs in sequence

- In theory a batch script can submit another script
- One script creates a second then runs it
- Most systems don’t support submission from compute nodes
  - Must run from primary compute node
  - ssh mio.mines.edu “cd rundir ; qsub next_run”
- In most cases it is better to use the batch dependency option

In theory a batch script can submit another script. That is, one script creates a second then runs it or it just knows which job to run next. This is a bit more difficult than it sounds. For one reason, most systems don’t support submission from compute nodes. You must submit from the front end node not the primary compute node. Here is an example ssh command to do the submission.

However, in most cases it is better to use the dependency options within the scheduler. We’ll look at that next.
Depend section of the srun man page

```
-d, --dependency=<dependency_list>
  Defer the start of this job until the specified dependencies have been satisfied/completed. <dependency_list> is of the form "<type:job_id[:job_id],type:job_id[:job_id]>". Many jobs can share the same dependency and these jobs may even belong to different users. The value may be changed after job submission using the scontrol command.

  after:job_id[:jobid...]  
  This job can begin execution after the specified jobs have begun execution.

  afterany:job_id[:jobid...]  
  This job can begin execution after the specified jobs have terminated.

  afternotok:job_id[:jobid...]  
  This job can begin execution after the specified jobs have terminated in some failed state (non-zero exit code, node failure, timed out, etc).

  afterok:job_id[:jobid...]  
  This job can begin execution after the specified jobs have successfully executed (ran to completion with an exit code of zero).

  expand:job_id  
  Resources allocated to this job should be used to expand the specified job. The job to expand must share the same QOS (Quality of Service) and partition. Gang scheduling of resources in the partition is also not supported.

  singleton  
  This job can begin execution after any previously launched jobs sharing the same job name and user have terminated.
```

This information is from the srun man page, describing the dependency option. It shows that there are several sub options. The usage syntax is show at the bottom of the page. We see that we can request that a job not start until after another starts, or after it finishes. There are several other options but these are the most important.
When you run a batch script with `sbatch` you will normally get any available nodes. `sbatch` have the ability to be more selective of nodes. Here we show how to select nodes by partition name. On Mio each group has its own partition for which it gets priority access. On AuN we have a debug partition which is designed for very short small jobs. We can also select nodes by name. Here we select nodes 1 to 6 on AuN. On Mio the nodes are named compute followed by a number.
There are two more things I want to talk about. This is a script that I wrote recently that I have as part of my environment. It is not a batch script but it actually works in two modes. What it does is it checks to see if the variable Slurm Node File is defined. This will happen if you are running a batch script or running interactively. If it is defined the script then just prints a list of our nodes for the current job. If we are not running on compute nodes then Slurm Node File is not defined and we use the command Slurm nodes to print a list of all of the nodes in our system.
Slurm array jobs

- Slurm allows array jobs:

  Job arrays offer a mechanism for submitting and managing collections of similar jobs quickly and easily. All jobs must have the same initial options (e.g. size, time limit, etc.), however it is possible to change some of these options after the job has begun execution using the command specifying the JobID of the array or individual ArrayJobID.

  Job arrays will have two additional environment variable set. `SLURM_ARRAY_JOB_ID` will be set to the first job ID of the array. `SLURM_ARRAY_TASK_ID` will be set to the job array index value. For example a job submission of this sort:

  ```
  sbatch --array=1-3 -N1 some_script
  ```

  will generate a job array containing three jobs. If the `sbatch` command responds

  Submitted batch job 36

  then the environment variables will be set as follows:

  ```
  SLURM_JOBID=36
  SLURM_ARRAY_JOB_ID=36
  SLURM_ARRAY_TASK_ID=1
  SLURM_JOBID=37
  SLURM_ARRAY_JOB_ID=36
  SLURM_ARRAY_TASK_ID=2
  SLURM_JOBID=38
  SLURM_ARRAY_JOB_ID=36
  SLURM_ARRAY_TASK_ID=3
  ```

You can create what is called array jobs in slurm. For an array job you are effectively submitting the same script N number of times but it can be done using a single command along with the “array” option. For array jobs there are two additional environmental variables defined, `SLURM_ARRAY_JOB_ID` and `SLURM_ARRAY_TASK_ID`. Each instance in the array gets defined a successive values for `SLURM_ARRAY_TASK_ID` starting at 1. They also each get successive values for `SLURM_JOB_ID`. `SLURM_ARRAY_JOB_ID` is the same for all of the instances in the array job. It is equal to the first `SLURM_JOB_ID`.

Here is an example where we have requested 3 instances of the script some script to be run.
This is the ultimate bash geek script. I got the base script from the web page listed here. What it does is a complete redirection of all standard out and standard error to files. In bash 1 and 2 are file descriptors for standard out and standard error.

What we do is set up two new file descriptors 3 and 4 to point to a logfile. Actually, we set up 3 to point to the log file and then have 4 point to 3.

Next we save 1 and 2 by having two other descriptors 5 and 6 point to them. Then we have 1 point to 3 and 2 point to 4, both of which point to the log file.

After that everything that goes to standard out and standard error goes to the log file.

One of the interesting commands is `ls file_that_does_not_exist`. This actually produces an error message that is sent to the log file.

The last three lines of the script restore standard out and standard error and close the log file.
As we have seen before we can use the environmental variables internally to set input and output files. Here we are doing both using `SLURM_ARRAY_TASK_ID` and `SLURM_ARRAY_JOB_ID`.

Note that we have specified the share option in our script. This will allow all sub jobs to run on the same node.
Slurm array jobs

Here we run 4 jobs on the same node producing:

```
[joeuser@aun001 memory]$ sbatch -p debug --nodelist=node002 --array=1-4 array
Submitted batch job 11968

[joeuser@aun001 memory]$ squeue -u joeuser

JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
11968_1     debug   hybrid  joeuser  R       0:05      1 node002
11968_2     debug   hybrid  joeuser  R       0:05      1 node002
11968_3     debug   hybrid  joeuser  R       0:05      1 node002
11968_4     debug   hybrid  joeuser  R       0:05      1 node002
```

Here is the output of squeue showing 4 instances of our script running from using the array option on the command line. Note that we have also specified a particular node on which to run. Finally we show the files created by running this array job with file names created from the environmental variables.
With that, we say good day.