Batch Scripting for Parallel Systems

Author
Timothy H. Kaiser, Ph.D.
tkaiser@mines.edu

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 M P I 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 M P I E X E C command. As you will see all of the scripts and example programs are available for download.

Purpose:

- To give you some ideas about what is possible
- To give you some examples to follow to help you write your own script
- Be a reference, where we can point people
- Document responses to questions

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 in to 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.

To Cover...

- Multiple jobs on a node
  - Sequential
  - Multiple scripts - one node
  - One Script - different MPI jobs on different cores

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...

- Mapping tasks to nodes
  - Less than N tasks per node
  - Different executables working together
  - Hybrid MPI/OpenMP jobs (MPI and Threading)
  - Running on heterogeneous nodes using all cores
- Job dependencies
  - Chaining jobs
  - Jobs submitting new jobs

Sometimes you might not want to use the standard MPI mapping of tasks to nodes. This can be important form 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

```
[joeuser@mio tests]$ [joeuser@mio tests]$ wget http://geco.mines.edu/scripts/morescripts.tgz
[joeuser@mio tests]$ [joeuser@mio tests]$ tar -xzf morescripts.tgz
[joeuser@mio tests]$ [joeuser@mio tests]$ cd morescripts


[joeuser@mio somescripts]$ make  2> /dev/null
mpicc -o c_ex00 c_ex00.c
mpif90 -o f_ex00 f_ex00.f
rm -rf fmpi.mod
icc info.c -o info_c
ifort info.f90 -o info_f
cp info.py info_p
chmod 700 info_p
ifort -O3 -mkl -openmp pointer.f90 -o fillmem
od -vAn -d -N1048576 < /dev/urandom > segment
tar -czf data.tgz segment
rm -rf segment*
mpicc -DDO_LOCAL_FILE_TEST -c sinkfile.c
mpif90 sinkf.f90 sinkfile.o -o sinkf
mpicc -DDO_LOCAL_FILE_TEST -DDO_C_TEST sinkfile.c -o sinkfile
rm *o *mod
chmod 700 nodes
```


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 p b s. The scripts are in subdirectories for each scheduler.

What we have

- [c_ex00.c, c_ex00]
  - hello world in C and MPI
- [f_ex00.f, f_ex00]
  - hello world in Fortran and MPI
- [info.c, info_c] [info.f90, info_f] [info.py]
  - Serial programs in C, Fortran and Python that print the node name and process id. Creates a node name process id

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.

info.c

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
main() {
    char name[128],fname[128];
    pid_t mypid;
    FILE *f;
    char aline[128];

/* get the process id */
    mypid=getpid();
/* get the host name */
    gethostname(name, 128);
/* make a file name based on these two */
    sprintf(fname,"%s_%8.8d",name,(int)mypid);
/* open and write to the file */
    f=fopen(fname,"w");
    fprintf(f,"C says hello from %d on %s\n",(int)mypid,name);
}
```


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.

info.f90

```
program info
  USE IFPOSIX ! needed by PXFGETPID
  implicit none
  integer ierr,mypid
  character(len=128) :: name,fname
! get the process id
  CALL PXFGETPID (mypid, ierr)
! get the node name
  call mynode(name)
! make a filename based on the two
  write(fname,'(a,"_",i8.8)')trim(name),mypid
! open and write to the file
  open(12,file=fname)
  write(12,*)"Fortran says hello from",mypid," on ",trim(name)
end program
```

```
subroutine mynode(name)
! Intel Fortran subroutine to return
! the name of a node on which you are
! running
  USE IFPOSIX
  implicit none
  integer jhandle
  integer ierr,len
  character(len=128) :: name
  CALL PXFSTRUCTCREATE ("utsname", jhandle, ierr)
  CALL PXFUNAME (jhandle, ierr)
  call PXFSTRGET(jhandle,"nodename",name,len,ierr)
end subroutine
```


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.

info.py

```
#!/usr/bin/env python
import os
# get the process id
mypid=os.getpid()
# get the node name
name=os.uname()[1]
# make a filename based on the two
fname="%s_%8.8d" % (name,mypid)
# open and write to the file
f=open(fname,"w")
f.write("Python says hello from %d on %s\n" %(mypid,name))
```


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.

Example Output From the Serial Programs

```
[joeuser@mio cwp]$ ./info_c
[joeuser@mio cwp]$ ls -lt mio*
-rw-rw-r-- 1 joeuser joeuser 41 Jan 11 13:47
mio.mines.edu_00050205
[joeuser@mio cwp]$ cat mio.mines.edu_00050205
C says hello from 50205 on mio.mines.edu
[joeuser@mio cwp]$
```


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

```
#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 %d on %s\n",myid,numprocs,myname);

    MPI_Finalize();
}
```


This is hello world in MPI. Each process prints its MPI i d, the total number of tasks and the name of the node in which it runs.

Fortran MPI example

```
!********************************************
!  This is a simple hello world program. Each processor
!  prints out its rank and total number of processors
!  in the current MPI run.
!****************************************************************
      program hello
      include "mpif.h"
      character (len=MPI_MAX_PROCESSOR_NAME):: myname
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      call MPI_Get_processor_name(myname,mylen,ierr)
      write(*,*)"Hello from ",myid," of ",numprocs," on ",trim(myname)
      call MPI_FINALIZE(ierr)
      stop
      end
```


This is hello world in Fortran and MPI. Like the C version, each process prints its MPI i d, the total number of tasks and the name of the node in which it runs.

Fortran Matrix Inversion Example

- Fills a number of arrays with random data
- Does a matrix inversion
- Used to test the performance of individual cores of a processor
- Can also be used to test threading

```
      include 'mkl_vsl.fi'
...
      program testinvert
      use numz
...
      call my_clock(cnt1(i))
      CALL DGESV( N, NRHS, twod, LDA, IPIVs(:,i), Bs(:,i), LDB, INFOs(i) )
      call my_clock(cnt2(i))
      write(*,'(i5,i5,3(f12.3))')i,infos(i),cnt2(i),cnt1(i),real(cnt2(i)-cnt1(i),b8)
...
```


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.c

- A somewhat complicated example
- Does a parallel file copy
    - Copies a file seen by MPI task 0 to
    - Each nodes (not task) in an MPI program that does not share the node used by task 0
    - Used in a situation where MPI tasks might not share file systems

MINES MINES MINES MINES MI

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

MINESMINESMINESMINESMI

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

http://www.tldp.org/LDP/Bash-Beginners-Guide/html/

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.

Notes on Commands

- > is used to sent output to a file (date > mylisting)
- >> append output to a file (ls >> mylisting)
- >& send output and error output to a file
- The ; can be used to combine multiline commands on a single line. Thus the following are equivalent

date ; echo “line 2” ; uptime

date
echo “line 2”
date

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.

For loops

```
myf90=`ls *f90`
for f in $myf90 ; do file $f ; done
doint.f90: ASCII program text
fourd.f90: ASCII program text
tintel.f90: ASCII program text
tp.f90: ASCII program text
vect.f90: ASCII program text
```

```
myf90=`ls *f90`
for f in $myf90
  do file $f
done
```

```
for (( c=1; c<=5; c++ )); do echo "Welcome $c times..."; done
Welcome 1 times...
Welcome 2 times...
Welcome 3 times...
Welcome 4 times...
Welcome 5 times...
```

```
for c in 1 2 3 4 5; do echo "Welcome $c times..."; done
Welcome 1 times...
Welcome 2 times...
Welcome 3 times...
Welcome 4 times...
Welcome 5 times...
```

```
for c in `seq 1 2 6`; do  echo "Welcome $c times..."; date; done
Welcome 1 times...
Tue Jul 31 12:17:11 MDT 2012
Welcome 3 times...
Tue Jul 31 12:17:11 MDT 2012
Welcome 5 times...
Tue Jul 31 12:17:11 MDT 2012
```

```
for c in `seq 1 2 6`
  do
  echo "Welcome $c times..."
  date
done
```


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 were 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.

Combing Operations

Operation	Effect
[! EXPR]	True if **EXPR** is false.
[(EXPR)]	Returns the value of **EXPR**. This may be used to override the normal precedence of operators.
[EXPR1 -a EXPR2]	True if both **EXPR1** and **EXPR2** are true.
[EXPR1 -o EXPR2]	True if either **EXPR1** or **EXPR2** is true.

MINESMINESMINESMINESMI

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.

Test Variable Being Set and "if"

We do this loop 3 times.
(1)"var" not set
(2)"var" set but empty
(3)var set and not empty

```
i= 1
var is unset or empty A
var is unset or empty A2

i= 2
var is unset or empty A
var is unset or empty A2
var is set but empty B

i= 3
var is set and not empty C
```

```
for i in 1 2 3 ; do
    echo "i=" $i
    if [ $i == 1 ] ; then unset var ; fi
    if [ $i == 2 ] ; then var="" ; fi
    if [ $i == 3 ] ; then var="abcd" ; fi

    if [ -z "$var" ]  ;     then echo "var is unset or empty A"; fi
    if [ ! -n "$var" ] ;    then echo "var is unset or empty A2"; fi
    if [ -z "${var-x}" ]  ; then echo "var is set but empty B"; fi
    if [ -n "$var" ] ;      then echo "var is set and not empty C"; fi
   echo
done
```


Here we show a few things, including testing to se 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.

String Tests

```
if test "abc" = "def" ;then echo "abc = def" ; else echo "nope 1" ; fi

if test "abc" != "def" ;then echo "abc != def" ; else echo "nope 2" ; fi

if  [ "abc" \< "def" ];then echo "abc < def" ; else echo "nope 3" ; fi

if  [ "abc" \> "def" ]; then echo "abc > def" ; else echo "nope 4" ; fi

if  [ "abc" \> "abc" ]; then echo "abc > abc" ; else echo "nope 5" ; fi
```

```
nope 1
abc != def
abc < def
nope 4
nope 5
```


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.

String Tests

```
if test "abc" = "def" ;then echo "abc = def" ; else echo "nope 1" ; fi
```

```
nope 1
```

```
if test "abc" != "def" ;then echo "abc != def" ; else echo "nope 2" ; fi
```

```
abc != def
```

```
if  [ "abc" \< "def" ];then echo "abc < def" ; else echo "nope 3" ; fi
```

```
abc < def
```

```
if  [ "abc" \> "def" ]; then echo "abc > def" ; else echo "nope 4" ; fi
```

```
nope 4
```

```
if  [ "abc" \> "abc" ]; then echo "abc > abc" ; else echo "nope 5" ; fi
```

```
nope 5
```


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.

File Tests

Test	Meaning
[-a FILE]	True if FILE exists.
[-b FILE]	True if FILE exists and is a block-special file.
[-c FILE]	True if FILE exists and is a character-special file.
[-d FILE]	True if FILE exists and is a directory.
[-e FILE]	True if FILE exists.
[-f FILE]	True if FILE exists and is a regular file.
[-g FILE]	True if FILE exists and its SGID bit is set.
[-h FILE]	True if FILE exists and is a symbolic link.
[-k FILE]	True if FILE exists and its sticky bit is set.
[-p FILE]	True if FILE exists and is a named pipe (FIFO).
[-r FILE]	True if FILE exists and is readable.
[-s FILE]	True if FILE exists and has a size greater than zero.
[-t FD]	True if file descriptor FD is open and refers to a terminal.
[-u FILE]	True if FILE exists and its SUID (set user ID) bit is set.
[-w FILE]	True if FILE exists and is writable.
[-x FILE]	True if FILE exists and is executable.
[-O FILE]	True if FILE exists and is owned by the effective user ID.
[-G FILE]	True if FILE exists and is owned by the effective group ID.
[-L FILE]	True if FILE exists and is a symbolic link.
[-N FILE]	True if FILE exists and has been modified since it was last read.
[-S FILE]	True if FILE exists and is a socket.
[FILE1 -nt FILE2]	True if FILE1 has been changed more recently than FILE2, or if FILE1 exists and FILE2 does not.
[FILE1 -ot FILE2]	True if FILE1 is older than FILE2, or is FILE2 exists and FILE1 does not.
[FILE1 -ef FILE2]	True if FILE1 and FILE2 refer to the same device and inode numbers.

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.

Testing Return Code & /dev/null

- Commands return an exit code
  - 0 = success
  - not 0 = failure
- The exit code from the previous command is stored in $?
- $? can be echoed or tested
- This is often used with piping output into /dev/null "the bit bucket" when you only want to know if a command was successful

```
ls a_dummy_file >& /dev/null

if [ $? -eq 0 ] ; then
    echo "ls of a_dummy_file successful"
fi
```


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 files exists. You do an L S of the file and check the exit code from L S.
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
```

```
ls of a_dummy_file failed
a_dummy_file does not exist

bottom of while loop

ls of a_dummy_file successful
a_dummy_file exists, breaking
```


Finally we show the while command. In this case we're first remove the file a_dummy_file. We then we use L S 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. P O E is an IBM scheduling environment. S G E or sun gridd engine is common on Rocks based systems. L S F is another common environment. Each environment has at lease one command to submit jobs. P O E is the command, for the P O E system. bsub works on L S F. 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 s batch 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

Related Commands SLURM

Command	Description - From http://slurm.schedmd.com/man_index.html
sbatch	Submit a batch script to SLURM.
srun	Run parallel jobs
scancel	Used to signal (cancel) jobs or job steps that are under the control of Slurm.
salloc	Obtain a SLURM job allocation (a set of nodes), Useful for interactive sessions.
sacct	Displays accounting data for all jobs and job steps in the SLURM job accounting log or SLURM database
sacctmgr	Used to view and modify Slurm account information.
sattach	Attach to a SLURM job step.
sdiag	scheduling diagnostic tool.
sinfo	view information about SLURM nodes and partitions.
smap	graphically view information about SLURM jobs, partitions, and set configurations parameters.
sprio	view the factors that comprise a job's scheduling priority
squeue	view information about jobs located in the SLURM scheduling queue.
sreport	Generate reports from the slurm accounting data.
sstat	Display various status information of a running job/step.

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 show queued jobs.

sjobs, slurmjobs, slurmnodes, and inuse are simple wrappers that reformat information from other commands

CSM Unique SLURM Commands

Command /opt/utility/*	**Description**
sjobs	Summary of running and queued jobs
slurmjobs	Show full information for all jobs -h for help
slurmnodes	Show full information for all nodes (-h for help
inuse	Node usage by group
match	Creates an mpiexec "appfile" for MPMD runs and nonstandard mappings of tasks to nodes
match_split	Creates an srun "multi-prog" for MPMD runs and nonstandard mappings of tasks to nodes
phostname	Glorified MPI/OpenMP "hello world"

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.

phostname "help"

```
[joeuser@aun002 ~]$ /opt/utility/phostname -h
phostname arguments:
          -h : Print this help message

no arguments : Print a list of the nodes on which the command is run.

 -f or -1    : Same as no argument but print MPI task id and Thread id
               If run with OpenMP threading enabled OMP_NUM_THREADS > 1
               there will be a line per MPI task and Thread.

 -F or -2    : Add columns to tell first MPI task on a node and and the
               numbering of tasks on a node. (Hint: pipe this output in
               to sort -r

 -a          : Print a listing of the environmental variables passed to
               MPI task. (Hint: use the -l option with SLURM to prepend MPI
               task #.)
[joeuser@aun002 ~]$
```


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.

A Simple Slurm Script for a MPI job

```
#!/bin/bash


#SBATCH --job-name="atest"
#SBATCH --nodes=2
#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 ~/bins/example/mpi
srun -n 8 ./c_ex00
```

Scripts contain comments designated with a # that are interpreted by SLURM and normal shell commands

We go to this directory

Run this MPI program on 8 cores

basic

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	This is a bash script
#SBATCH --job-name="atest"	Give our job a name in the
#SBATCH --nodes=1	We want 1 node
#SBATCH --ntasks-per-node=8	We expect to run 8 tasks/node
#SBATCH --time=00:02:00	We want the node for 2 minutes
#SBATCH --output=stdout	Output will go to a file "stdout"
#SBATCH --error=stderr	Errors will go to a file "stdout"
#SBATCH --export=ALL	**Pass current environment to nodes**
#SBATCH --mail-type=ALL	Send email on abort,begin,end
#SBATCH --mail-user=joeuser@mines.edu	Address for email
#----------------------	Just a normal "comment"
cd /home/joeuser/examples	Go to this directory first
srun -n 8 ./c_ex00	Run c_ex00 on 8 cores

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 minus 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 C D 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 s run or M P I E X E C launch applications in parallel, that is several copies of an application are launched on the nodes you are using. S run is the normal command used under slurm.

s run or M P I E X E C 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

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

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.

Example list of variables

```
SLURM_CHECKPOINT_IMAGE_DIR=/bins/joeuser/examples/mpi
SLURM_NODELIST=node[114-115]
SLURM_JOB_NAME=atest
SLURMD_NODENAME=node114
SLURM_TOPOLOGY_ADDR=node114
SLURM_NTASKS_PER_NODE=8
SLURM_PRIO_PROCESS=0
SLURM_NODE_ALIASES=(null)
SLURM_EXPORT_ENV=ALL
SLURM_TOPOLOGY_ADDR_PATTERN=node
SLURM_NNODES=2
SLURM_JOBID=11160
SLURM_NTASKS=16
SLURM_TASKS_PER_NODE=8(x2)
SLURM_JOB_ID=11160
SLURM_JOB_USER=joeuser
SLURM_JOB_UID=15049
SLURM_NODEID=0
SLURM_SUBMIT_DIR=/bins/joeuser/examples/mpi
SLURM_TASK_PID=14098
SLURM_NPROCS=16
SLURM_CPUS_ON_NODE=8
SLURM_PROCID=0
SLURM_JOB_NODELIST=node[114-115]
SLURM_LOCALID=0
SLURM_JOB_CPUS_PER_NODE=8(x2)
SLURM_GTIDS=0
SLURM_SUBMIT_HOST=aun002.mines.edu
SLURM_JOB_PARTITION=aun
SLURM_JOB_NUM_NODES=2
```


Here is a list of slurm variables that were defined durring 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

docd

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.

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.%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
```

We go to "starting" directory

Run this MPI program on 8 cores

docd

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

```
#!/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
```

The output from the script

Gives the program output as it runs with each run having a unique output file

redirect

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.

Shorten JOBID

- $PBS_JOBID is of the form
  - 45682.mio.mines.edu
- How can we shorten this to just a number?
  - sed -e 's/\..*//'
  - strips everything past the first period

```
$ echo $PBS_JOBID
201665.mio.mines.edu
$ MY_JOBID=`echo $PBS_JOBID | sed -e 's/\..*//'`
$ echo $MY_JOBID
201665
```

On some batch systems the job name might be a combination of text and numbers. For example under P B S JOB I D 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 I D, that can be used in the rest of the script.

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.%j
#SBATCH -e stderr.%j
#SBATCH --export=ALL
#SBATCH --mail-type=ALL
#SBATCH --mail-user=joeuser@mines.edu
#----------------------
JOBID=`echo $SLURM_JOB_ID`
cd $SLURM_SUBMIT_DIR

srun -n 8 ./c_ex00 > my_out.$JOBID
```

The output from the script

myout.#####

number

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 send 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

MINESMINESMINESMINESMI

We want to keep copies of all of the script that were run. We want more that 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.

Records Notifications

```
#!/bin/bash -x
#SBATCH --job-name="hybrid"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --ntasks=16
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=00:02:00
#SBATCH -o stdout.%j
#SBATCH -e stderr.%j

#SBATCH --mail-type=ALL
#SBATCH --mail-user=joeuser@mines.edu

# Go to the directoy 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`

# Save a copy of our environment and script
echo $SLURM_JOB_NODELIST > nodes.$JOBID
cat $0 > script.$JOBID
printenv  > env.$JOBID

#mail us the environment and other "stuff"
####  mail <  env.$JOBID -s $JOBID  $USER@mines.edu
#ssh $SLURM_SUBMIT_HOST "mail <  $MYBASE/$JOBID/env.$JOBID -s $JOBID  $SLURM_JOB_USER@mines.edu"
mkdir -p ~/tmail
cp env.$JOBID ~/tmail
export MAIL_HOST=$SLURM_SUBMIT_HOST
export MAIL_HOST=mindy.mines.edu
ssh mindy.mines.edu "mail <  ~/tmail/env.$JOBID -s $JOBID  $SLURM_JOB_USER@mines.edu"


srun /opt/utility/phostname -F > output.$JOBID
```

How can I record what I did and where?

How can I know when a particular script starts and exactly what is running?

notify

First we set the variable JOB I D to be a shortened version of Slurm Job I D.

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.

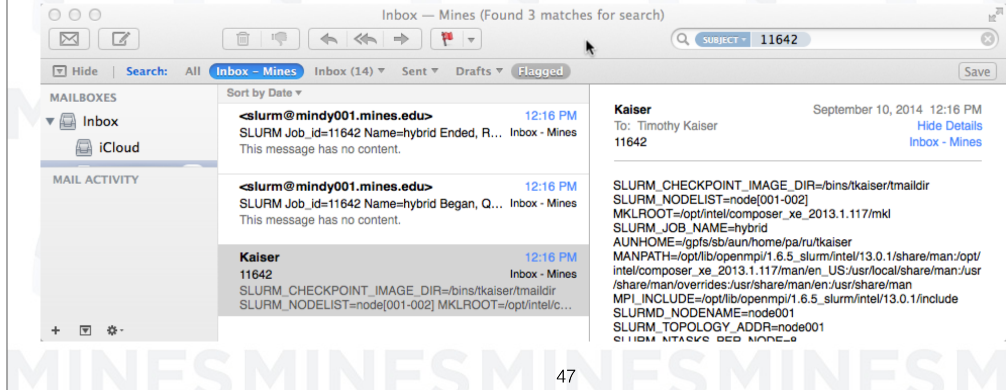
Finally, we mail all of this to us with the subject of our mail being the job I D.

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.

Lots of records...

```
[joeuser@aun002 tmaildir]$ ls -l *11642*
-rw-rw-r-- 1 joeuser joeuser  6108 Sep 10 12:16 env.11642
-rw-rw-r-- 1 joeuser joeuser    14 Sep 10 12:16 nodes.11642
-rw-rw-r-- 1 joeuser joeuser 15934 Sep 10 12:16 output.11642
-rw-rw-r-- 1 joeuser joeuser  1014 Sep 10 12:16 script.11642
-rw-rw-r-- 1 joeuser joeuser   447 Sep 10 12:16 stderr.11642
-rw-rw-r-- 1 joeuser joeuser     0 Sep 10 12:16 stdout.11642
[joeuser@aun002 tmaildir]$
```



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 I D.

We are going to take an additional look at some of the ways we can use variables.

A digression: /opt/utility/expands

The CSM written utility /opt/utility/expands takes a Slurm style compressed node list and creates a full list similar to what is produces by PBS

```
/opt/utility/expands node[001-003,005-007,100] | sort -u
node001
node002
node003
node005
node006
node007
node100
```


The CSM written utility /opt/utility/expands takes a Slurm style compressed node list and creates a full list similar to what is produces 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 -x
#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 directoy 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
```

Not an MPI job but we still use srun to ensure that the computation runs on a compute node

envl

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.

```
#!/bin/bash -x
#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 directoy 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
```


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 reason 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

```
[joeuser@aun001 memory]$ ls -l *11714*
-rw-rw-r-- 1 joeuser joeuser   11 Sep 10 13:31 fillmemc.sinput.input.11714
-rw-rw-r-- 1 joeuser joeuser  128 Sep 10 13:31 fillmemc.sinput.nodes.11714
-rw-rw-r-- 1 joeuser joeuser 3259 Sep 10 13:31 fillmemc.sinput.output.11714
-rw-rw-r-- 1 joeuser joeuser    8 Sep 10 13:31 nodes.11714
-rw-rw-r-- 1 joeuser joeuser  205 Sep 10 13:31 stderr.11714
-rw-rw-r-- 1 joeuser joeuser    0 Sep 10 13:31 stdout.11714
[joeuser@aun001 memory]$

[joeuser@aun001 memory]$ cat fillmemc.sinput.nodes.11714 | sort -u
node001
[joeuser@aun001 memory]$ cat fillmemc.sinput.nodes.11714 | wc
     16      16     128
[joeuser@aun001 memory]$

[joeuser@aun001 memory]$ cat fillmemc.sinput.input.11714
4096 64 1

[joeuser@aun001 memory]$ head fillmemc.sinput.output.11714
 matrix size=        4096
 copies=          64
 bytes=     8589934592 gbytes=     8.000
 using mkl for inverts
 generating data for run           1  of           1
 generating time=       2.114 threads= 16
 starting inverts
   33    0   48675.423   48671.805       3.618
    9    0   48675.423   48671.805       3.618
   57    0   48675.423   48671.805       3.618
```


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.

Case 1: Multiple Scripts

Or the same script several times

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.

```
#!/bin/bash -x
#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 directoy 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
/opt/utility/expands $SLURM_JOB_NODELIST > $APP.$INPUT.nodes.$JOBID

export INPUT=sinput
export APP=fillmemc


cat $INPUT >  $APP.$INPUT.input.$JOBID
srun ./$APP  < $INPUT >> $APP.$INPUT.output.$JOBID
```


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.

```
#!/bin/bash -x
#SBATCH --job-name="hybrid"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks=1
#SBATCH --share
#SBATCH --export=ALL
#SBATCH --time=00:02:00
#SBATCH -o stdout.%j
#SBATCH -e stderr.%j

# Go to the directoy 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.$JOBID
export OMP_NUM_THREADS=2
srun  ./$APP  < $INPUT >> $APP.$INPUT.output.$JOBID
```

We have commented out the line that sets the input file name. We can (must) specify the input before running the script.

inputenv

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..

```
[joeuser@mio test]$ export INPUT=sinput1
[joeuser@mio test]$ qsub from_env
267335.mio.mines.edu
[joeuser@mio test]$ cat mynodes*
n20
[joeuser@mio test]$ export INPUT=sinput2
[joeuser@mio test]$ qsub from_env -l nodes=n20
267336.mio.mines.edu
[joeuser@mio test]$
[joeuser@mio test]$ export INPUT=sinput3
[joeuser@mio test]$ qsub from_env -l nodes=n20
267337.mio.mines.edu
[joeuser@mio test]$
[joeuser@mio test]$ export INPUT=sinput4
[joeuser@mio test]$ qsub from_env -l nodes=n20
267338.mio.mines.edu
[joeuser@mio test]$
```

Which node are we using? It is in the mynodes* file

Force the rest of our jobs to the same node

We have specified the input file here. This is picked up by the script

(If you have a reserved node you can specify it for the first run also.)

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:

```
[joeuser@mio test]$ ls -l fillmemc.sinput*

-rw-rw-r-- 1 joeuser joeuser 3395 Feb 15 11:31 fillmemc.sinput1.26733**5**.mio.mines.edu
-rw-rw-r-- 1 joeuser joeuser 5035 Feb 15 11:32 fillmemc.sinput2.26733**6**.mio.mines.edu
-rw-rw-r-- 1 joeuser joeuser 6675 Feb 15 11:32 fillmemc.sinput3.26733**7**.mio.mines.edu
-rw-rw-r-- 1 joeuser joeuser 1541 Feb 15 11:29 fillmemc.sinput4.26733**4**.mio.mines.edu
-rw-rw-r-- 1 joeuser joeuser 1755 Feb 15 11:31 fillmemc.sinput4.26733**8**.mio.mines.edu
[joeuser@mio test]$
```

Note different job numbers

The input file name becomes part of the output file name

Our execution line

```
./$APP  < $INPUT >> $APP.$INPUT.$PBS_JOBID
```


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..

```
[joeuser@aun001 memory]$ export INPUT=sinput1
[joeuser@aun001 memory]$ sbatch  env2
Submitted batch job 11834
[joeuser@aun001 memory]$ cat nodes.11834
node001

[joeuser@aun001 memory]$ export INPUT=sinput2
[joeuser@aun001 memory]$ sbatch --nodelist=node001 env2
Submitted batch job 11835

[joeuser@aun001 memory]$ export INPUT=sinput3
[joeuser@aun001 memory]$ sbatch --nodelist=node001 env2
Submitted batch job 11836

[joeuser@aun001 memory]$ export INPUT=sinput4
[joeuser@aun001 memory]$ sbatch --nodelist=node001 env2
Submitted batch job 11837
```

Which node are we using? It is in the nodes* file

Force the rest of our jobs to the same node

We have specified the input file here. This is picked up by the script

(If you have a reserved node you can specify it for the first run also.)

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@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
```

Note different job numbers

```
[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 20:35 fillmemc.sinput2.output.11835
-rw-rw-r-- 1 joeuser joeuser 2481 Sep 10 20:34 fillmemc.sinput1.output.11834
[joeuser@mio test]$
```

The input file name becomes part of the output file name

Our execution line

```
./$APP  < $INPUT >> $APP.$INPUT.$PBS_JOBID
```


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.

Case 2: Multiple Executables Same Script

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.

```
#!/bin/bash -x
#SBATCH --job-name="hybrid"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks=1
#SBATCH --share
#SBATCH --export=ALL
#SBATCH --time=00:02:00
#SBATCH -o stdout.%j
#SBATCH -e stderr.%j

# Go to the directoy 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 APP=fillmemc


export OMP_NUM_THREADS=2
for INPUT in  sinput1 sinput2 sinput3 sinput4 ; do

  srun  ./$APP  < $INPUT >> $APP.$INPUT.output.$JOBID &
done
wait
```

We have our application name and input file set as a variable

We launch the application over a list of input files

The wait command "holds" the node until all of your applications are done

Forces the job into the background so we can launch the next

multiwait

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 look 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.

Our output files:

Note common job
numbers with different
input files

```
[joeuser@aun001 memory]$ ls -lt fillmemc*  head -4
-rw-rw-r-- 1 joeuser joeuser    695 Sep 10 20:52 fillmemc.sinput3.output.11839
-rw-rw-r-- 1 joeuser joeuser    695 Sep 10 20:52 fillmemc.sinput1.output.11839
-rw-rw-r-- 1 joeuser joeuser    695 Sep 10 20:52 fillmemc.sinput2.output.11839
-rw-rw-r-- 1 joeuser joeuser    695 Sep 10 20:52 fillmemc.sinput4.output.11839
```

```
srun ./$APP  < $INPUT >> $APP.$INPUT.$JOB &
```


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.

mpiexec/srun and serial applications

- 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 E X E C and MPI run will work with nonMPI programs.
What happens is that MPI E X E 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.

Our Batch file, batch1

```
#!/bin/bash -x
#SBATCH --job-name="hybrid"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --ntasks=16
#SBATCH --share
#SBATCH --export=ALL
#SBATCH --time=00:02:00
#SBATCH -o stdout.%j
#SBATCH -e stderr.%j

# Go to the directoy 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 APP=info.py

  srun  ./$APP
wait
```

info_p is a python program that creates a file based on node name and process id

serial

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

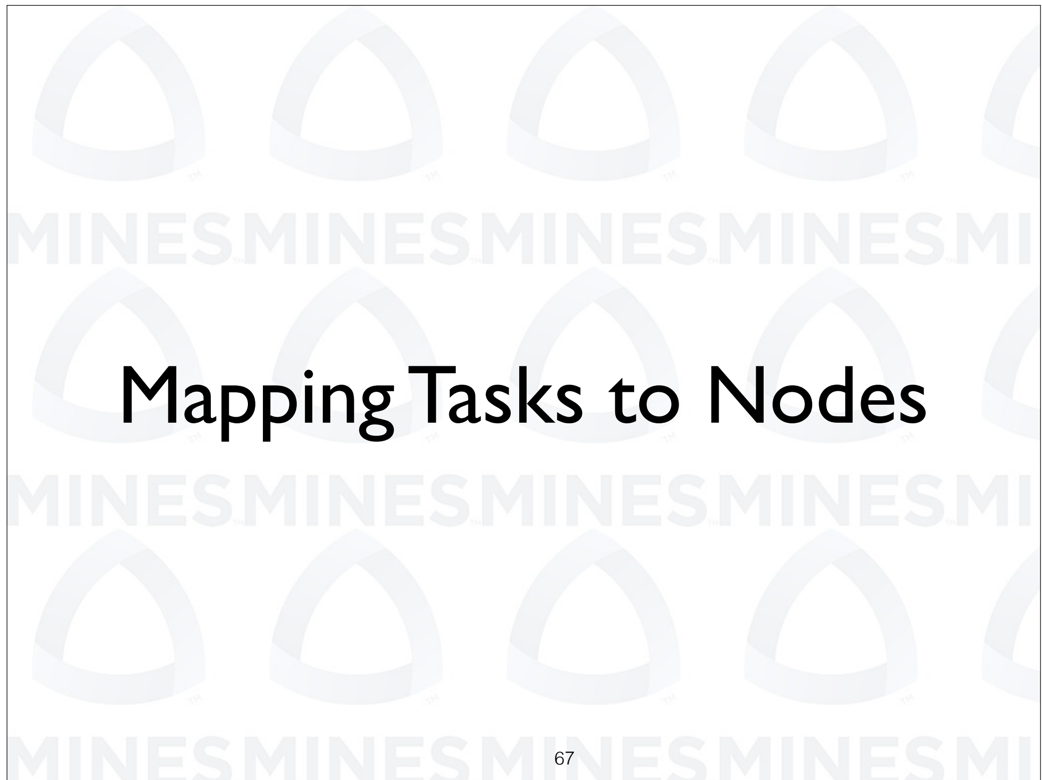
export MYPROGRAM=info_p

```
[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 39 Sep 10 21:05 node002_00007603
-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 M P I E X E C or s run. 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.

Need better than default mappings...

- Want to use less than all of the nodes on a node
  - Large memory/task
  - Hybrid MPI/OpenMPI
- Different executables on various cores (MPMD)
- Heterogeneous environment with different core counts

MINESMINESMINESMINESMI

Why would we want something other than the default mappings? The default mapping is each core gets a M P I task.

We might want less than on 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 M P I and open M P.

We might want to run different executables on various nodes, that is a M P M D 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 M P I E X E X 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 slrum 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 M P I E X E C 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

MINESMINESMINESMINESMI

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 M P M D. You can also list a node and or a program more than once.

Examples

These two are equivalent

Appfile Example 1

```
-host compute-1-1 -np 1 myprogram
-host compute-1-1 -np 1 myprogram
-host compute-1-1 -np 1 myprogram
-host compute-1-1 -np 1 myprogram
```

Appfile Example 2

```
-host compute-1-1 -np 4 myprogram
```

These two are not equivalent

Appfile Example 3

```
-host compute-1-1 -np 2 aya.out
-host compute-2-3 -np 2 bee.out
```

Appfile Example 3

```
-host compute-1-1 -np 1 aya.out
-host compute-2-3 -np 1 aya.out
-host compute-1-1 -np 2 bee.out
-host compute-2-3 -np 2 bee.out
```

Note: You should specify the full path to your program

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.

Difficulty and Solution

- Problem:
  - Names of the nodes that you are using are not known until after the job is submitted
  - You need to create the appfile on the fly from within your PBS script

There is a problem.

The difficultly 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.

Difficulty and Solution

- Solution:
  - Under PBS the variable $PBS_NODEFILE contains the name of a file that has the list of nodes on which your job will run.
  - Under Slrum the variable SLURM_JOB_NODELIST has a compressed list of nodes.
  - We have created a script "match" which takes a list of nodes and a list of applications to run on those nodes and creates an appfile
  - Located on Mio and AuN at /opt/utility/match

MINESMINESMINESMINESMI

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_NODE FILE 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 M P I E X E C command line.

The math script has lots of options we will look at most of them in the next few slides.

Match notes

- Number of applications that get launched
  - Is equal to the length of the longer of the two lists, the node file list, or the application list
  - If the lists are not the same length then multiple copies will be launched
  - Match also takes an optional replication count, the number copies of an application to run on a node
- Feel free to copy and modify the match script for your own needs

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.

Examples

```
#get a copy of all of our nodes, each node will be
#listed 8 times
cat $PBS_NODEFILE > fulllist

#save a nicely sorted short list of nodes, each node only
#listed one time
sort -u $PBS_NODEFILE > shortlist
```

fulllist

```
compute-8-15.local
compute-8-15.local
compute-8-15.local
compute-8-15.local
compute-8-15.local
compute-8-15.local
compute-8-15.local
compute-8-15.local
compute-8-13.local
compute-8-13.local
compute-8-13.local
compute-8-13.local
compute-8-13.local
compute-8-13.local
compute-8-13.local
compute-8-13.local
```

shortlist

```
compute-8-15.local
compute-8-13.local
```

```
/lustre/home/apps/utility/nsort $PBS_NODEFILE
```

Also works to give a sorted list

Here is an example. We are running on two nodes with 8 core each. We first generate two lists. The full list contains 16 entires, 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.

match fulllist twoprograms > appfile1

```
-host  compute-8-15.local  -np  1  c_ex00
-host  compute-8-15.local  -np  1  f_ex00
-host  compute-8-15.local  -np  1  c_ex00
-host  compute-8-15.local  -np  1  f_ex00
-host  compute-8-15.local  -np  1  c_ex00
-host  compute-8-15.local  -np  1  f_ex00
-host  compute-8-15.local  -np  1  c_ex00
-host  compute-8-15.local  -np  1  f_ex00
-host  compute-8-13.local  -np  1  c_ex00
-host  compute-8-13.local  -np  1  f_ex00
-host  compute-8-13.local  -np  1  c_ex00
-host  compute-8-13.local  -np  1  f_ex00
-host  compute-8-13.local  -np  1  c_ex00
-host  compute-8-13.local  -np  1  f_ex00
-host  compute-8-13.local  -np  1  c_ex00
-host  compute-8-13.local  -np  1  f_ex00
```


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.

match shortlist twoprograms > appfile2

```
-host  compute-8-13.local  -np  1  c_ex00
-host  compute-8-15.local  -np  1  f_ex00
```


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.

match shortlist twoprograms 2 > appfile3

```
-host  compute-8-13.local  -np  2  c_ex00
-host  compute-8-15.local  -np  2  f_ex00
```


Here we use a replication count to get two copies on each node.

match shortlist oneprogram 2 > appfile4

```
-host  compute-8-13.local  -np  2  c_ex00
-host  compute-8-15.local  -np  2  c_ex00
```

This will be useful for hybrid MPI OpenMP

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 M P I open M P programs.

Can take names from command line

match <node list file> -p"list of programs" [<number of copies per node>]

```
match shortlist -p"c_ex01 f_ex01" 1 8
```

Run 1 copy of c_ex01 on the first node in shortlist and 8 copies of f_ex01 on the second node

If you don't specify the number of copies then do 1 per core

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

```
match shortlist -p"c_ex01" 8 8
```

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

```
match 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 s run.
Under slurm you can use the multi prog option. This is similar to app file option for mpi e x e c
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.

Slurm Specific mapping

- The hostlist and app list are of the form:

```
HOSTLIST:   APPLIST:
node001     0 helloc
node002     1 hellof
node002     2 hellof
node002     3 hellof
```

- We have the issues creating these files as we do for the mpiexec appfile
- We have a script that converts a mpiexec app file to these separate files

MINES MINES MINES MINES

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 e x e c. So we have created another script that splits an mpi e x e c file to these two files.

match_split

```
Usage:

/opt/utility/match_split [MATCHFILE applist hostlist]

A post processing script for the CSM utility match.

It takes the mpiexec "appfile" output from match and
creates srun style application list and hostfile list
files for use with the srun option "--multi-prog".

You can pipe "|"  match into this program in which
case /opt/utility/match_split will create the files applist
and hostlist.

Of you can specify the file created by match on the command
line in which case the files created will be of the form
MATCHFILE_applist and MATCHFILE_hostlist.

Finally, you can specify all three files on the command line

/opt/utility/match_split MATCHFILE applist hostlist.

To run a slurm job using these files you do two things:

export SLURM_HOSTFILE=hostlist
srun -n12 --multi-prog applist

where -n12 specifies the total number of MPI tasks to start.
```


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.

match_split

```
Examples:

[joeuser@aun001 mpi]$ cat matchfile
-host  node001  -np  1  helloc
-host  node002  -np  3  hellof
[joeuser@aun001 mpi]$
[joeuser@aun001 mpi]$ /opt/utility/match_split matchfile
[joeuser@aun001 mpi]$ cat matchfile_applist
0 helloc
1 hellof
2 hellof
3 hellof
[joeuser@aun001 mpi]$ cat matchfile_hostlist
node001
node002
node002
node002

export SLURM_HOSTFILE=matchfile_hostlist
srun -n4 --multi-prog matchfile_applist
```


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.

match_split

```
[joeuser@aun001 mpi]$ match shortlist -p"c01 f01" 3 2 | /opt/utility/match_split
[joeuser@aun001 mpi]$
[joeuser@aun001 mpi]$ cat applist
0 c01
1 c01
2 c01
3 f01
4 f01
[joeuser@aun001 mpi]$ cat hostlist
node001
node001
node001
node002
node002

export SLURM_HOSTFILE=hostlist
srun -n5 --multi-prog applist
```


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 app list and host list which are again used with s run.

Slurm script for match and match_split

```
#!/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=00:02:00

# Go to the directoy 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 a 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

split

Here is a full example using both an app file with mpi e x e c and the app list and host list with s run. We put the outputs from our two runs in separate files. The two output files are the same.

Output:

```
[joeuser@aun001 mpi]$ cat appfile
-host  node001  -np  4  helloc
-host  node002  -np  8  hellof
```

```
[joeuser@aun001 mpi]$ cat applist
0 helloc
1 helloc
2 helloc
3 helloc
4 hellof
5 hellof
6 hellof
7 hellof
8 hellof
9 hellof
10 hellof
11 hellof
```

```
[joeuser@aun001 mpi]$ cat hostlist
node001
node001
node001
node001
node002
node002
node002
node002
node002
node002
node002
node002
```

```
[joeuser@aun001 mpi]$ cat out2.11895 | sort
C-> Hello from node001 # 0 of 12
C-> Hello from node001 # 1 of 12
C-> Hello from node001 # 2 of 12
C-> Hello from node001 # 3 of 12
F-> Hello from            node002 #    10 of    12
F-> Hello from            node002 #    11 of    12
F-> Hello from            node002 #     4 of    12
F-> Hello from            node002 #     5 of    12
F-> Hello from            node002 #     6 of    12
F-> Hello from            node002 #     7 of    12
F-> Hello from            node002 #     8 of    12
F-> Hello from            node002 #     9 of    12
```


Here we have the three files app file used with mpi e x e c and the files app list and host list used with s run. Again, the outputs produced using s run and mpi e x e c 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 share between all of the tasks of an MPI application. For example we might want to use disk that is local to a node.

A Directory for Each Run

```
#!/bin/bash -x
#SBATCH --job-name="hybrid"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --ntasks=16
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=00:02:00
#SBATCH --mail-type=ALL
#SBATCH --mail-user=joeuser@mines.edu

# Go to the directoy 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`

# Create a "base name" for a directory
# in which our job will run
# For production runs this should be in $SCRATCH
MYBASE=$SLURM_SUBMIT_DIR
#MYBASE=$SCRATCH/mc2_tests

# We could create a directoy for our run based on the date/time
export NEW_DIR=`date +%y%m%d%H%M%S`
# But here we create a directoy for our run based on the $JOBID and go there
mkdir -p $MYBASE/$JOBID
cd $MYBASE/$JOBID
odir=`pwd`
export ODIR=`pwd`
# Create a link back to our starting directory
ln -s $SLURM_SUBMIT_DIR submit
cp $SLURM_SUBMIT_DIR/data.tgz .
tar -xzf data.tgz
```

We could create NEW_DIR based on a time stamp, the year, month, day, hour, minute, and second.

or

Based on JOBID

"Copy" our data to the new directory from our staring directory

dir

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 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 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.

Local Disk Space

- Most parallel machines have some disk space that is local
  - Can only be seen by tasks on the same nodes
  - Can't be seen from the primary compute node
  - Might be faster than shared space
  - Size? Location?
    - Usually a bad idea to use /tmp
    - On "Rocks" it's /state/partition1
- Usage is up to local policy

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 disk

- Figure out where it is
- Create directories
- Copy to the new directory
- Compute
- Copy what you want to shared storage
- Clean up after yourself

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 S S H and S C P to preform 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. S S H logs in to a node then performs a command. S C P does a copy from one node to another.

Set Up

```
#!/bin/bash -x
#SBATCH --job-name="hybrid"
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=4
#SBATCH --ntasks=16
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=00:02:00
#SBATCH --mail-type=ALL
#SBATCH --mail-user=joeuser@mines.edu

# Go to the directoy 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`

# Create a "base name" for a directory
# in which our job will run
# For production runs this should be in $SCRATCH
MYBASE=$SLURM_SUBMIT_DIR
#MYBASE=$SCRATCH/mc2_tests

# We could create a directoy for our run based on the date/time
export NEW_DIR=`date +%y%m%d%H%M%S`
# But here we create a directoy for our run based on the $JOBID and go there
mkdir -p $MYBASE/$JOBID
cd $MYBASE/$JOBID
odir=`pwd`
export ODIR=`pwd`
```

local

Here is the first part of our script. It is the same as the last script that creates a ned 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

```
# Create a link back to our starting directory
ln -s $SLURM_SUBMIT_DIR submit
cp $SLURM_SUBMIT_DIR/data.tgz .
tar -xzf data.tgz

if [ -n "$JOBTMP" ] ; then
  echo using $JOBTMP from environment
else
  export JOBTMP=~/scratch/local_sim
fi


#get a list of nodes...
export nlist=`/opt/utility/expands`
# For each node...
for i in $nlist
do
# Create my temporary directory in /scratch on each node
  ssh $i mkdir -p $JOBTMP/$NEW_DIR
# Copy my data
  echo $USER@$i:$JOBTMP/$NEW_DIR
  scp * $USER@$i:$JOBTMP/$NEW_DIR
done

# save a copy of our nodes
/opt/utility/expands > nlist.$JOBID
```

local

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.

Run

```
export APP=$SLURM_SUBMIT_DIR/sinkfile
match $ODIR/flist.$JOBID -p"$APP" > appfile
mpiexec -app appfile  >& screen.$JOBID
```

local

Here we use the match script to create a list for MPI E X E C and we run the application. MPI E X E C launches the application on each node. But there is a bit of magic happening here. MPI E X E C 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

```
#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
```

local

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.

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

MINESMINESMINESMINESMI

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, -n most cases it is better to use the dependency options within the sceduler. 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.  <depen-
       dency_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 com-
              pletion 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.
```

```
srun --dependency=after:123 /tmp/script
srun --dependency=before:234 /tmp/script
```


This information is from the s run 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.

srun - requesting specific nodes

- srun normally gives you any node
- Can select nodes based on queues
- Can select nodes base on name

```
sbatch -p debug

sbatch -p group_name

sbatch –nodelist=node[001-006]
```


When you run a batch script with s batch you will normally get any available nodes. s batch 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.

A Digression, Not a Batch Script...

```
#!/bin/bash
# $SLURM_NODEFILE defined? (running in batch)
if [ -n "$SLURM_NODELIST" ] ; then
    echo $SLURM_NODELIST
else
# $SLURM_NODELIST not defined
# this implies we are not running in batch
# list all the nodes
  scontrol show nodes -o  | cut -d" " -f1,7 \
                          | sed "s/NodeName=//"
                          | sed "s/CPULoad=//"
  sinfo -a
fi
```

This script prints your nodes if running in batch or all nodes if running on the front end nodes

nodes

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.

Redirection Revisited

```
#!/bin/bash
#PBS -l nodes=1:ppn=8
#PBS -l walltime=00:02:00
#PBS -N testIO
#PBS -o out.$PBS_JOBID
#PBS -e err.$PBS_JOBID
#PBS -V
#PBS -m abe
#PBS -M joeuser@mines.edu
#----------------------
cd $PBS_O_WORKDIR

###############
# http://compgroups.net/comp.unix.shell/bash-changing-stdout/497180
# set up our redirects of stdout and stderr
                            # 1 and 2 are file descriptors for
                            # stdout and stderr
                            # 3 and 4 are descriptors to logfile
                            # we will use 3 for stdout 4 for stderr
exec 3>>logfile.`date +"%y%m%d%H%M%S"`
                            # anything that goes to 4 will go to 3
                            # which is our file we have created
exec 4>&3
exec 5>&1 6>&2              # save "pointers" to stdin and stdout
exec 1>&3 2>&4              # redirect stdin and stdout to file
###############
# normal commands
                            # this line goes to stdout
echo this is a test from stdout
                            # this line goes to stderr
echo this is a test from stderr >&2
                            # error message goes to stderr
ls file_that_does_not_exist
ls
mpiexec -n 8 ./c_ex00 > myout.$PBS_JOBID
mpiexec -n 8 ./c_ex00
###############
exec 1>&5 2>&6              # restore original stdin and stdout
3>&- 4>&-                   # close logfile descriptors
5>&- 6>&-                   # close saved stdin and stdout
```

nodes

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.

Slurm array jobs

```
#!/bin/bash -x
#SBATCH --job-name="array"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks=1
#SBATCH --share
#SBATCH --export=ALL
#SBATCH --time=00:02:00
#SBATCH -o stdout.%j
#SBATCH -e stderr.%j

# Go to the directoy 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${SLURM_ARRAY_TASK_ID}
export APP=fillmemc


/opt/utility/expands $SLURM_JOB_NODELIST >  $APP.$INPUT.nodes.$JOBID
cat $INPUT >  $APP.$INPUT.input.$JOBID
export OMP_NUM_THREADS=2
srun  ./$APP  < $INPUT >> $APP.$INPUT.output.$JOBID.$SLURM_ARRAY_JOB_ID.$SLURM_ARRAY_TASK_ID
```

This script is similar to the last but we use SLURM_ARRAY_TASK_ID and SLURM_ARRAY_JOB_ID to define input and output files

array

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

```
[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 we run 4 jobs on the same node producing:

```
[joeuser@aun001 memory]$ ls -lt | head
total 12768
-rw-rw-r-- 1 joeuser joeuser   1791 Sep 11 14:50 stderr.11968
-rw-rw-r-- 1 joeuser joeuser    803 Sep 11 14:50 stderr.11970
-rw-rw-r-- 1 joeuser joeuser    803 Sep 11 14:50 stderr.11971
-rw-rw-r-- 1 joeuser joeuser    803 Sep 11 14:50 stderr.11969
-rw-rw-r-- 1 joeuser joeuser   2763 Sep 11 14:50 fillmemc.sinput1.output.11968.11968.1
-rw-rw-r-- 1 joeuser joeuser   2763 Sep 11 14:50 fillmemc.sinput4.output.11971.11968.4
-rw-rw-r-- 1 joeuser joeuser   2763 Sep 11 14:50 fillmemc.sinput2.output.11969.11968.2
-rw-rw-r-- 1 joeuser joeuser   2763 Sep 11 14:50 fillmemc.sinput3.output.11970.11968.3
```


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.