Slurm Survival Guide

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, Fall 2014

Slurm

Slurm (**S**imple **Linux U**tility for **R**esource **M**anagement) is a highly modular open-source software package for utilizing clusters in high-performance computing (HPC):

- Allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time for the user to utilize for their workload,
- Provides a framework for starting, executing, and monitoring work (typically a parallel job, but not necessarily) on a set of allocated nodes,
- Schedules resources by managing a queue of pending workloads submitted by users.

Links

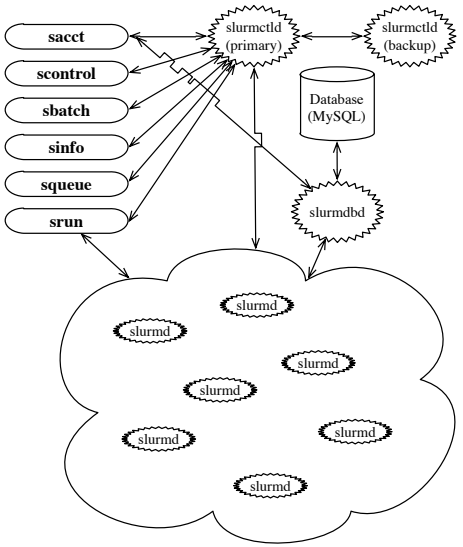
Primary Slurm URLs:

- `computing.llnl.gov/linux/slurm/slurm.html`,
  Slurm's original home at LLNL, now a bit outdated, historical reference
- `www.schedmd.com`,
  Slurm's new primary site

These sites contain a lot of documentation and pointers to others, including various tutorials.

Slurm Architecture

Client commands, `sacct`, `scontrol` (mostly for administrative use), `salloc`, `sbatch`, `sinfo`, `squeue`, `srun` for user submissions and queries. Each compute node runs a **slurmd** daemon that is responsible for command and control on that node. **slurmctld** is the Slurm manager daemon, while **slurmdbd** handles recording the usage data (there is an alternative flat file option).



sinfo

`sinfo` is Slurm's client for querying clusters, nodes, and partitions. Note that partitions can overlap, so a node can belong to multiple partitions. If you run `sinfo` with just default options, on a large cluster you will see a large quantity of displayed information, so let us instead consider a simpler query, and focus on CCR's `debug` partition, which is a small set of nodes set aside for development and debugging work with rapid turnaround:

```
[rush:~]$ sinfo --partition=debug
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug        up    1:00:00      7   idle d07n33s[01-02],d16n[02-03],k05n26,k08n41s[01-02]
```

From this we infer a wall time limit of 1 hour (hence the quicker turnaround for development jobs), and 7 nodes total - but what about more details on the nodes themselves?

There are a host of options that you can specify for `sinfo`, this combination is handy for looking at the details on the nodes:

```
[rush:~]$ sinfo --partition=debug --long --Node
NODELIST        NODES PARTITION       STATE CPUS    S:C:T MEMORY TMP_DISK WEIGHT FEATURES REASON
d07n33s[01-02]      2     debug        idle    8    2:4:1  24000        0     23 IB,CPU-L none
d16n[02-03]         2     debug        idle    8    2:4:1  24000        0     23 IB,CPU-L none
k05n26              1     debug        idle   16    2:8:1 128000        0    102 CPU-E5-2 none
k08n41s[01-02]      2     debug        idle   12    2:6:1  48000        0     31 IB,CPU-E none
```

If you look at the man page for `sinfo`, you can see that the above command is just a handy short cut for using `sinfo --format`, with which we expand the feature set for the nodes:

```
[rush:~]$ sinfo --partition=debug --format="%14N %.5D %9P %6T %.3c %.6z %.6m %.8d %.6w %12f %6G %20E"
NODELIST       NODES PARTITION STATE  CPU  S:C:T MEMORY TMP_DISK WEIGHT FEATURES     GRES   REASON
d07n33s[01-02]     2 debug     idle     8  2:4:1  24000        0     23 IB,CPU-L5630 (null) none
d16n[02-03]        2 debug     idle     8  2:4:1  24000        0     23 IB,CPU-L5520 (null) none
k05n26             1 debug     idle    16  2:8:1 128000        0    102 CPU-E5-2660  gpu:2  none
k08n41s[01-02]     2 debug     idle    12  2:6:1  48000        0     31 IB,CPU-E5645 (null) none
```


squeue

`squeue` provides details on jobs in Slurm. On a very busy cluster, the default output can be overwhelming, much like `sinfo` it can be more helpful to carefully focus on points of interest. More often than not we are interested mainly in our own jobs:

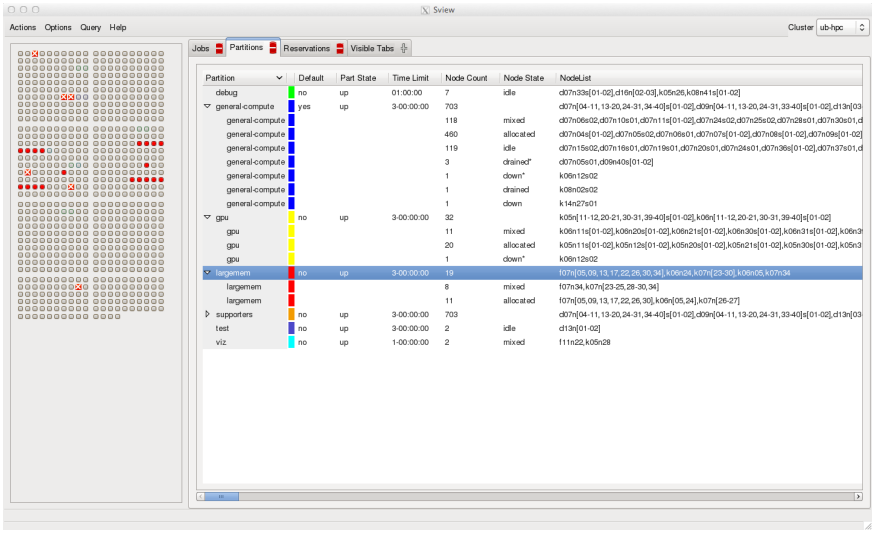
```
[rush:~]$ squeue --users=$USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2482840 general-c  nb-mult   jonesm PD       0:00      2 (Resources)
```

The default output is rather terse, so if you need more detail you can resort to using the `--format` option:

```
[k07n14:~]$ squeue --format="%8i %8u %8a %10P %4D %4C %5m %91 %20S %6h %6Q %6b %12f %R" --users=$USER
JOBID    USER     ACCOUNT  PARTITION  NODE CPUS MIN_M TIMELIMIT START_TIME          SHARED PRIORI GRES   FEATURES     NODELIST(REASON)
2482840  jonesm   ccrtech  general-co 2    32   3000  1:00:00   2014-07-17T11:31:57  no     50490  (null) CPU-E5-2660  (Resources)
```


sview

`sview` is a graphical user interface (GUI) that provides Slurm information on nodes, jobs, partitions, reservations, *etc.* - it provides much the same functionality as `squeue` and `sinfo`. Note that you have to have graphics forwarding enabled (X11 if you are accessing the cluster front end via `ssh`), or alternatively use a remote visualization system (although in this case the graphics acceleration is not really needed). Note that on a busy cluster, `sview` can be rather slow.



sbatch

`sbatch` is the primary way that most users submit jobs to Slurm. The syntax for usage is:

```
sbatch [options] script [args...]
```

where `script` is a conventional shell script.

- If `script` is omitted, then `sbatch` will attempt to read from standard input (very few users go this route)
- Various `options` can be supplied either on the command line, or in the `script` itself as `#SBATCH` directives
- See `man sbatch` for more details on the available `option`, some of which we will cover in the examples

salloc

`salloc` immediately requests an allocation of resources, generally interactively. Here is an example of a simple allocation request on the debug partition:

```
 [rush:~/d_mpi-samples]$ salloc --nodes=1 --ntasks-per-node=8 --partition=debug --time=00:10:00
salloc: Granted job allocation 2524554
[rush:~/d_mpi-samples]$ srun hostname
d07n33s02
d07n33s02
d07n33s02
d07n33s02
d07n33s02
d07n33s02
d07n33s02
d07n33s02
[k07n14:~/d_mpi-samples]$ export | grep SLURM
declare -x SLURM_JOBID="2524554"
declare -x SLURM_JOB_CPUS_PER_NODE="8"
declare -x SLURM_JOB_ID="2524554"
declare -x SLURM_JOB_NODELIST="d07n33s02"
declare -x SLURM_JOB_NUM_NODES="1"
declare -x SLURM_MEM_PER_CPU="3000"
declare -x SLURM_NNODES="1"
declare -x SLURM_NODELIST="d07n33s02"
declare -x SLURM_NODE_ALIASES="(null)"
declare -x SLURM_NPROCS="8"
declare -x SLURM_NTASKS="8"
declare -x SLURM_NTASKS_PER_NODE="8"
declare -x SLURM_SUBMIT_DIR="/ifs/user/jonesm/d_mpi-samples"
declare -x SLURM_SUBMIT_HOST="k07n14"
declare -x SLURM_TASKS_PER_NODE="8"
[rush:~/d_mpi-samples]$ exit
exit
[rush:~/d_mpi-samples]$ export | grep SLURM
[rush:~/d_mpi-samples]$
```


Things to note when using `salloc`:

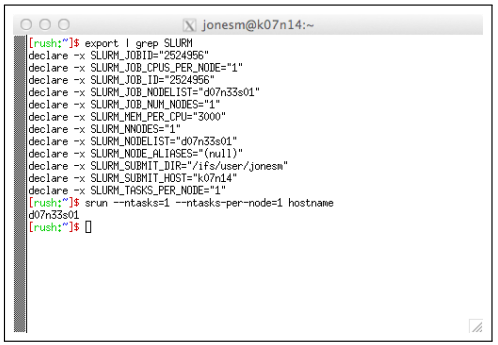
- `salloc` requests will be scheduled like any other Slurm job, which can take some amount of waiting time depending on the amount of resources requested,
- You get a new shell when the allocation succeeds, and you should exit that shell when finished, otherwise your shell environment will be somewhat confused,
- Most Slurm users find it easier to use `sbatch` with a dedicated script, rather than tying up a terminal with an `salloc`
- See `man salloc` for more details on the available options, which are essentially the same as for `sbatch`.

`salloc` can also be used to launch commands directly, either an application:

```
[rush:~/d_mpi-samples]$ salloc --nodes=1 --ntasks-per-node=8 --partition=debug \
                        --time=00:10:00 srun hostname
salloc: Granted job allocation 2524894
d07n33s01
d07n33s01
d07n33s01
d07n33s01
d07n33s01
d07n33s01
d07n33s01
d07n33s01
d07n33s01
salloc: Relinquishing job allocation 2524894
salloc: Job allocation 2524894 has been revoked.
```

or say a handy extra terminal:

```
[rush:~/d_mpi-samples]$ salloc --nodes=1 --ntasks-per-node=8 --partition=debug \
                        --time=00:10:00 xterm
salloc: Granted job allocation 2524915
```

- The `xterm` still gets run on the cluster front-end, just like the previous example,
- You can again utilize Slurm's task launcher, `srun` to use the allocated resources,
- Once you exit the `xterm` the `salloc` will release the allocation.

fisbatch

`fisbatch` is a Slurm community script adapted to CCR - it is designed to automate the process of `salloc` followed by an `ssh` to the allocated node (with X forwarding enabled). To achieve a similar result to what we just did in the previous example:

```
[rush:~/d_mpi-samples]$ fisbatch --partition=debug --nodes=1 --ntasks-per-node=8 \
                                 --time=00:10:00
FISBATCH -- waiting for JOBID 2734481 to start on cluster=ub-hpc and partition=debug
!
FISBATCH -- Connecting to head node (d07n33s02)
[d07n33s02:/ifs/user/jonesm/d_mpi-samples]$
srun hostname
d07n33s02
d07n33s02
d07n33s02
d07n33s02
d07n33s02
d07n33s02
d07n33s02
d07n33s02
[d07n33s02:/ifs/user/jonesm/d_mpi-samples]$ exit
exit
[screen is terminating]
Connection to d07n33s02 closed.
FISBATCH -- exiting job
scancel: error: Kill job error on job id 2734481: Invalid job id specified
```


srun

`srun` is Slurm's parallel task launcher. Note that `srun` can also create the allocation as well, so if you only need to run a single command interactively you can simply execute a single `srun`:

```
[rush:~/d_mpi-samples]$ srun --nodes=1 --ntasks-per-node=8 --partition=debug \
                         --time=00:10:00 hostname
d07n33s01
d07n33s01
d07n33s01
d07n33s01
d07n33s01
d07n33s01
d07n33s01
d07n33s01
[rush:~/d_mpi-samples]$ module load intel-mpi
[rush:~/d_mpi-samples]$ mpicc -o cpi.impi cpi.c
[rush:~/d_mpi-samples]$ export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
[rush:~/d_mpi-samples]$ srun --nodes=1 --ntasks-per-node=8 --partition=debug \
                         --time=00:10:00 cpi.impi
Process 0 on d07n33s01
Process 1 on d07n33s01
Process 2 on d07n33s01
Process 3 on d07n33s01
Process 4 on d07n33s01
Process 5 on d07n33s01
Process 6 on d07n33s01
Process 7 on d07n33s01
pi is approximately 3.1416009869231249, Error is 0.0000083333333318
wall clock time = 0.000337
```

or you can use `srun` in conjunction with `sbatch` and `salloc`.

Important `srun` notes:

- You can use essentially the same Slurm options as with `salloc` and `sbatch`,
- `srun` is a **parallel** task launcher - note from the preceding example that it executed the same binary on all of the allocated cores,
- `srun` supports various MPI implementations, including Intel MPI as in the example above - in this case the extra library (`I_MPI_PMI_LIBRARY`) overrides an existing one in Intel's MPI library to ensure proper task placement, see the `--mpi` option to `srun`,
- Note that options to `srun` can override existing options in a `salloc` or `sbatch` environment (.e.g., you can request all of the cores per node in an `sbatch` script, but then under-subscribe them).

Modules Software Management System

There are a large number of available software packages on the CCR systems, particularly the Linux clusters. To help maintain this often confusing environment, the **modules** package is used to add and remove these packages from your default environment (many of the packages conflict in terms of their names, libraries, etc., so the default is a minimally populated environment).

The module Command

module command syntax:

```
-bash-2.05b$ module help

  Modules Release 3.1.6 (Copyright GNU GPL v2 1991):
  Available Commands and Usage:
        + add|load              modulefile [modulefile ...]
        + rm|unload             modulefile [modulefile ...]
        + switch|swap           modulefile1 modulefile2
        + display|show          modulefile [modulefile ...]
        + avail                 [modulefile [modulefile ...]]
        + use [-a|--append]     dir [dir ...]
        + unuse                 dir [dir ...]
        + update
        + purge
        + list
        + clear
        + help                  [modulefile [modulefile ...]]
        + whatis                [modulefile [modulefile ...]]
        + apropos|keyword       string
        + initadd               modulefile [modulefile ...]
        + initprepend           modulefile [modulefile ...]
        + initrm                modulefile [modulefile ...]
        + initswitch            modulefile1 modulefile2
        + initlist
        + initclear
```


Using module in Batch

If you change shells in your batch script you may need to explicitly load the modules environment:

tcsh :

```
source $MODULESHOME/init/tcsh
```

bash :

```
. ${MODULESHOME}/init/bash
```

but generally you should not need to worry about this step (do a "module list" and if it works ok your environment should already be properly initialized).

Modules Highlights

`module avail [string]`	list available modules (opt. *string*)
`module help name`	help text for *modulename*
`module list`	list modules in your environment
`module load name`	load selected *modulename*
`module unload name`	unload selected *modulename*
`module swap name1 name2`	swaps one module for another

Module Usage Examples

```
[rush:~]$ module list
Currently Loaded Modulefiles:
  1) null      2) modules
[rush:~]$ module avail intel

------------------- /util/Modules/modulefiles --------------------
intel/10.0          intel/13.0          intel-ipp/7.0.1     intel-mpi/4.1.0
intel/10.1          intel/13.1(default) intel-mpi/3.1       intel-mpi/4.1.1
intel/11.0          intel/14.0          intel-mpi/3.2       intel-mpi/4.1.3
intel/11.1          intel/9.0           intel-mpi/4.0       intel-tbb/3.0.3
intel/12.0          intel/9.1           intel-mpi/4.0.1     intel-tbb/4.1
intel/12.1          intel/9.1.040       intel-mpi/4.0.3
[rush:~]$ module avail intel/

------------------- /util/Modules/modulefiles --------------------
intel/10.0    intel/11.1    intel/13.0    intel/9.0
intel/10.1    intel/12.0    intel/13.1    intel/9.1
intel/11.0    intel/12.1    intel/14.0    intel/9.1.040
[rush:~]$ module load intel/12.1
[rush:~]$ which icc
/util/intel/composer_xe_2011_sp1.11.339/bin/intel64/icc
[rush:~]$ module swap intel/12.1 intel/14.0
[rush:~]$ which icc
/util/intel/composer_xe_2013_sp1.2.144/bin/intel64/icc
```


Build Your Own Modules

You can build your own modules, by using the `use.own` module:

```
[rush:~]$ module show use.own
-------------------------------------------
/util/Modules/modulefiles/use.own:

module-whatis   adds your own modulefiles directory to MODULEPATH
module  use --append /user/jonesm/privatemodules
-------------------------------------------
```

- Create your `$HOME/privatemodules` directory
- Create your own module files (`tcl`-based) in this private repository, you can use the existing ones in `$MODULEPATH` as templates

More Module Information

- `http://modules.sourceforge.net`,
  Home site for Modules source code
- `https://www.nersc.gov/users/software/nersc-user-environment/modules`,
  NERSC Modules help page
- `man module`,
  Module command `man` page
- `man modulefile`,
  modulefile syntax `man` page

Simple OpenMP Example

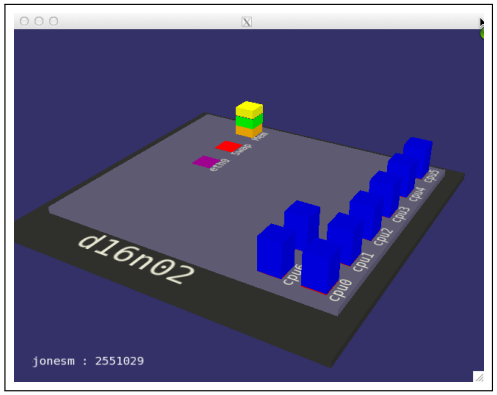
A simple example with an OpenMP based code:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8
#SBATCH --exclusive
#SBATCH --constraint=CPU-L5520
#SBATCH --partition=debug
#SBATCH --time=00:10:00
#SBATCH --mail-type=END
#SBATCH --mail-user=jonesm@buffalo.edu
#SBATCH --output=slurmOMP.out
#SBATCH --job-name=omp
#
module load intel
module list
export | grep SLURM
echo "Running on $SLURM_NNODES node with $SLURM_CPUS_ON_NODE cores."
export OMP_NUM_THREADS=$SLURM_CPUS_ON_NODE # single node only for OpenMP
./lap3-omp<<EOF
1024
1.e-4
EOF
```


Notes on the simple OpenMP Example:

- In this case the `--exclusive` flag is redundant, we are requesting all 8 cores in the L5520 nodes anyway,
- You can also request all 8 cores using `--ntasks-per-node=8`, and omitting `--cpus-per-task` since we run the application directly rather than through srun (which would otherwise interpret `--ntasks-per-node=8` to launch 8 versions of the application, each with 8 OpenMP threads)

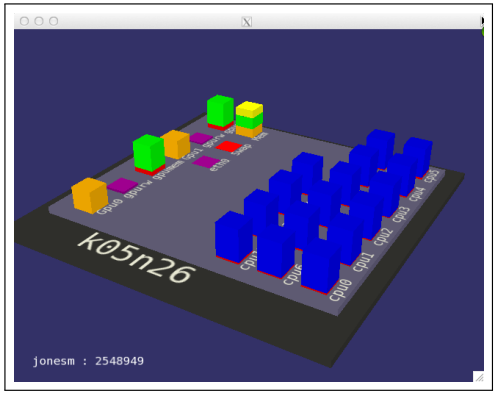
- `slurmjobvis` view of preceding example, showing CPU/network activity and memory used (when you mouse over the columns it should display values)



MPI Parallel Job With GPUs

```
#!/bin/bash
#SBATCH --partition=debug
#SBATCH --time=01:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --gres=gpu:2
#SBATCH --output=slurmNAMD.out
#SBATCH --mail-user=jonesm@buffalo.edu
#SBATCH --mail-type=END
#SBATCH --job-name=stmv
#
module load namd/2.9-MPI-CUDA
#
# CUDA version requires MPI, see /util/slurm-scripts/slurmNAMD for
#   other options
echo "NAMDHOME = "$NAMDHOME
export I_MPI_DEBUG=4
export | grep I_MPI
#
# construct namd nodes file from $PBS_NODEFILE
#
export MY_NODEFILE=tmp.$SLURM_JOBID
srun -l hostname -s | sort -n | awk '{print $2}' > $MY_NODEFILE
NPROCS=`cat $MY_NODEFILE | wc -l`
NNODES=`cat $MY_NODEFILE | uniq | wc -l`
echo "Running $NPROCS processes."
export NAMDNODEFILE=namd.$SLURM_JOBID
echo "group main" > $NAMDNODEFILE
NODES=`cat $MY_NODEFILE`
for node in $NODES ; do
   echo "host "$node >> $NAMDNODEFILE
done
#
# run STMV benchmark
#
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
srun namd2 stmv.namd
[ -e $MY_NODEFILE ] && \rm $MY_NODEFILE
[ -e $NAMDNODEFILE ] && \rm $NAMDNODEFILE
```

- `slurmjobvis` view of preceding example, showing CPU and GPU activity
- Note that the `--partition=debug` currently only has one such GPU node, for more nodes and longer running times, you can instead submit to `--partition=gpu` (32 12-core nodes)



Advanced Example: Multiple Serial Runs

This example runs an ensemble of a serial application, allowing Slurm to map the individual tasks to whatever resources are available:

```
1  #!/bin/bash
2  #SBATCH --tasks=64
3  #SBATCH --time=00:10:00
4  #SBATCH --mail-type=ALL
5  #SBATCH --mail-user=jonesm@buffalo.edu
6  #SBATCH --output=slurmDIST.out
7  #SBATCH --job-name=DIST
8  #
9  module list
10 export | grep SLURM
11 NPROCS=`srun -l hostname -s | wc -l`
12 echo "Running $NPROCS processes."
13 for ((i=0;i<SLURM_NTASKS;i++)); do
14    echo "Task $i"
15    srun --cpus-per-task=1 --exclusive --nodes=1 --ntasks=1 ./rp p11_4.$i > out.$i &
16 done
17 wait
```


Things to note from the ensemble example:

- You need the `wait` command to wait on the jobs sent to the background,
- This example can also be done with an ensemble of individual Slurm jobs, or with a **job array**,
- The individual runs need to take the same amount of time, otherwise some cores will be idled waiting on completion of the slowest run, leading to a poor load balance.

Advanced Example: Multiple Serial Runs With a Job Array

A job array version of running the same ensemble of a serial application:

```
1  #!/bin/bash
2  #SBATCH --nodes=1
3  #SBATCH --ntasks-per-node=1
4  #SBATCH --array=0-31
5  #SBATCH --time=00:10:00
6  #SBATCH --mail-type=ALL
7  #SBATCH --mail-user=jonesm@buffalo.edu
8  #SBATCH --output=slurmARRAY-%j.out
9  #SBATCH --job-name=ARRAY
10 #
11 module list
12 export | grep SLURM
13 NPROCS=`srun -l hostname -s | wc -l`
14 echo "Running $NPROCS processes."
15 srun --nodes=1 --ntasks=1 ./rp p11_4.$SLURM_ARRAY_TASK_ID > out.$SLURM_ARRAY_TASK_ID
```

Notes on Slurm array job example:

- Note that you get multiple jobs (the array) with a single `sbatch` submission, each has its own identifier and output file,
- Load balancing is not an issue in this case, as each job in the array will get its own dedicated resources,
- So when do you use the single job with an embedded ensemble? That depends partially on your preferences, the workload, and queueing policy. Sometimes parallel jobs get higher priority, so bundling the jobs together can actually get higher overall throughput.

Example Using `sbcast` File Staging

You can use Slurm's `sbcast` command to broadcast a file out to remote resources.

```
1  #!/bin/bash
2  #SBATCH --nodes=2
3  #SBATCH --ntasks-per-node=8
4  #SBATCH --constraint=CPU-L5520|CPU-L5630
5  #SBATCH --partition=debug
6  #SBATCH --time=00:10:00
7  #SBATCH --mail-type=END
8  #SBATCH --mail-user=jonesm@buffalo.edu
9  #SBATCH --output=slurmQ.out
10 #SBATCH --job-name=titan-test
11 #
12 module load titan
13 which titan
14 export I_MPI_DEBUG=4
15 export | grep I_MPI
16 export | grep SLURM
17 NNODES=`srun hostname -s | uniq | wc -l`
18 echo "Number of nodes: $NNODES"
19 NPROCS=`srun -l hostname -s | wc -l`
20 echo "Number of cores: $NPROCS"
21 NODELIST=`srun --ntasks-per-node=1 --ntasks=$NNODES hostname -s`
22 echo "Allocated hosts: " $NODELIST
23 # copy all input files out to remote nodes' local scratch
24 for f in *.inp *.data ; do
25   sbcast $f $SLURMTMPDIR/$f
26 done
27 cd $SLURMTMPDIR
28 export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
29 srun titan
30 pwd
31 ls -lrt
32 # unfortunately slurm lacks an inverse to bcast for the output
33 for node in $NODELIST; do
34     ssh $node "cd $SLURMTMPDIR ; cp perf* pile* $SLURM_SUBMIT_DIR/"
35 done
```


Notes on the `sbcast` example:

- The most common use for this is to run a data-intensive application from the local disk on each node (automatically set to `$SLURMTMPDIR`),
- Note the lack of an inverse operation to `sbcast`, hence the loop over the allocated nodes and the `ssh` to manually copy output files back to the submission directory (of course you could also do the same to replace the `sbcast`),
- Are you better off using the local disk or the shared file system? For a data-intensive application, the local disk i/o is more scalable.

Summary

Slurm is a very flexible resource manager, designed to scale to very large installations:

- Flexibility comes with a certain of complexity, there are multiple ways of accomplishing various tasks (e.g., running an interactive job, parallel task launching, etc.)
- Once you have some familiarity with Slurm, it becomes easier to automate your workflows

Options Not Exemplified

Things we have left out:

- There are many additional options to resource allocation requests, for different constraints, accounts, etc.,
  - Memory is one the most important of these resources - our examples utilized little memory, so the default allocated amount (currently 3GB/core) was enough. See the `--mem` option for setting the memory request per node if you need more,
- There are many ways to customize the outputs from the various Slurm client commands (see the various `man` pages),
- Priority and scheduling - a complicated study in its own right, and subject to many site policies and dependencies (see, for example, the Slurm documentation and `man sprio`).

CCR Slurm Specific Details/Notes

- `$SLURMTMPDIR` is a local scratch directory created on each compute node per Slurm job (generally `/scratch/$SLURM_JOBID`)
- Use the **debug** partition for development jobs (less than one hour of walltime, single or two nodes)
- Use the **gpu** partition for production jobs using GPUs
- Use the **largemem** partition when requesting the large memory 32-core nodes (a priority bump shortens the waiting time)