Running Jobs in Physics & Astronomy

Jay Nietling

August 26, 2014

This is an outline of the facilities for sharing resources, scheduling possibly many jobs for future execution, and programming distributed and shared memory processes on the computers in Physics & Astronomy at UNLV. SLURM (simple Linux utility for resource management) is the resource management system installed on the clusters. Programming models available are OpenMPI (distributed shared memory message passing interface), OpenMP (shared memory parallel programming), Pthreads (shared memory parallel programming), and CUDA (graphics processing unit based parallel programming). Programming distributed and shared memory applications is beyond the scope of this note however, how to compile and run such jobs is outlined.

SLURM (Simple Linux Utility for Resource Management)

Slurm is a light weight system for managing resources (processors, memory, disk) and many possibly cooperating processes within compute clusters in Physics and Astronomy.[1,2,3,4]The processes may be completely independent sequential processes, distributed shared memory processes, multi-core shared memory processes, or processes that utilize graphics processing units. SLURM is simple, open source with a GPL license, portable, fault-tolerant, secure, easy for system administrators to manage, and scalable. It is used on some of the largest computing environments as well as our modest facilities.

Physics Compute Facilities

Clusters of computers in Physics & Astronomy are grouped administratively typically according to grants or research groups that funded the purchase of said cluster. Although SLURM refers to *clusters* in its documentation, we currently map these administratively/financially defined clusters into SLURM *partitions* detailed in Table 1. Access to the partitions is controlled by the software using Unix group membership. To access the resources in a particular partition a user will have to be in the Unix group associated with that partition. Gaining access requires agreement of the manager of the cluster. In addition, there are SLURM partitions identified by *features* indicating type of CPU, availability of GPU or other resources.

[1] **SLURM Nomenclature** – *Node*: the compute resource in SLURM, consisting of a single computer that may include multiple processor sockets, multiple processor cores per socket, physical memory, and temporary disk space.

[2] *Partition*: a grouping of nodes into a logical set. Essentially a job queue with associated properties such as maximum job size, time limits, maximum nodes, and users with access to the queue.

[3] *Job*: a resource allocation assigned to a user for specified amount of time. Jobs are allocated nodes within a partition until the resources within the partition are exhausted. Once a job is assigned a set of nodes, the user is able to initiate parallel work in the form of job steps in any configuration within the allocation. For instance, a single job step may be started that utilizes all nodes allocated to the job, or several job steps may independently use a portion of the allocation.

[4] *Job step*: one of a set of possibly parallel tasks launched at the same time sharing a common communication mechanism, allocated resources within a job's allocation, and steps executing sequentially or concurrently.

SLURM partition	*description*	*host names*	*manager*
cosmo	Cosmological studies	`cosmo-[1-5]`	Nagamine
ld	LD?	`ld-[1-5]`	Proga
open	Open	`h-[0-10]`	Nietling
pes	Potential Energy Surface	`pes-[1-2]`	Robins
phi	Phi	`phi`	Pang
rad	Nuclear studies	`rad-[1-8],moly`	Kim
sn	Super Nova	`sn-[1-11]`	Bing, Nagamine, Proga
solid	HiPSEC cluster	`solid-[1-27]`	Chen

Table 1: Compute clusters in Physics & Astronomy

Getting Started with SLURM

The SLURM system is of classic Unix style design. It is a collection of shell[5] commands that each accomplish a single task. Each command typically has optional features that may be activated with command line arguments. Also typical of Unix, each command has an associated *man page* that describes what the command does and what command-line options and environment variables effect the behavior of the command.

To get started with SLURM, login to a computer that is a member of one of the compute clusters. Host names are listed in Table 1. Then setup your shell's environment for using SLURM as detailed in Figure 1 . After your shell's environment is set up, you'll have access to manual pages such as the excerpt for `sinfo` in Figure 2. All SLURM commands may be run from any node within a cluster.

[5] The Unix command interpreter is referred to as *the shell*.

```
$ # h-0b is a node in the
$ # open cluster
$ ssh h-0b
$ . /share/rc/slurm
$
```

Figure 1: Setup your shell to use SLURM. Typically, our rc scripts modify your shell's `PATH` and `MANPATH` environment variables.

```
$ man sinfo

SINFO(1)                        Slurm components

NAME
       sinfo - view information about SLURM nodes and partitions.

SYNOPSIS
       sinfo [OPTIONS...]

DESCRIPTION
       sinfo  is used to view partition and node information for a
       system running SLURM.
.
.
.
$
```

Figure 2: Viewing the "man page" for the `sinfo` command.

```
$ sinfo --help
Usage: sinfo [OPTIONS]
  -a, --all  show all partiti...
             not accessible)
  -b, --bg   show bgblocks (o...
  -d, --dead show only non-re...
.
.
.
```

Figure 3: sinfo help output.

Create a small shell script as in Figure 4 to submit a job into the SLURM job queue. The shell script is submitted to SLURM with the `sbatch` command as shown in Figure 5. The `#SBATCH` shell comments are also directives to `sbatch` that indicate resource requirements and job steps of the job. These comment/directives have command line equivalents. `--job-name` associates

On modern open source Unix systems, running a command with the `--help` command line option will cause the program to list brief descriptions of all options available with that command. Indeed this is the case with the SLURM commands, an example in Figure 3. The SLURM authors have also consistently included a `--verbose` command line option with the programs that will cause the programs to be more chatty, show more detail, and indicate what the program defaults are.

an arbitrary, hopefully meaningful, name with a job. It is listed by `squeue` and may be used in commands such as `scancel`. Still referring to Figure 4, on line 9, the working directory must be readable and searchable by the user running the job. The output and error arguments direct Unix `stdout` and `stderr` to files. They must be writable by the user running the job. The directives on lines 15-19 of Figure 4 detail the user's perceived resource requirements for the job. The numbers are a little outrageous for a job that is only going to print out a node's hostname but it illustrates how a job is run on multiple nodes. While the shell script itself is a job step, job steps are also created by the `srun` commands on lines 20 and 21.

The script is submitted to SLURM with the `sbatch` command shown in Figure 5. If there are no errors in the script the command will print the *job id* associated with the job. The job enters the queue in the *PENDING* state awaiting the availability of resources requested in the job script. When the resources become available and the job has the highest priority in the queue, a job *allocation* is created for it and it enters the *RUNNING* state. If the job completes successfully it will go into the *COMPLETED* state. If unsuccessful it will go into the *FAILED* state.

The state of the job is printed in the `ST` field in the default output of the `squeue` command as shown in Figure 5. Using the `--verbose` command line argument of `sbatch` will print the full English job states.

```
#!/bin/sh
#
# hello.sh:
#
# a little shell script to
# illustrate SLURM features.
#
#SBATCH --job-name=hello
#SBATCH --workdir=/homes/bozo/test
#SBATCH --output=hello.out
#SBATCH --error=hello.err
#
## resource requirements:
#
#SBATCH --partition=open
#SBATCH --ntasks=5
#SBATCH --nodes=5
#SBATCH --time=00:05:00
#SBATCH --mem-per-cpu=200mb

srun hostname
srun sleep 60

exit 0
```

Figure 4: A simple shell script `hello.sh` to try out SLURM.

```
$ sbatch hello.sh
Submitted batch job 51
$ squeue -j 51
             JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
                51      open    hello     bozo   R       0:16      5 h-[1-3],h-0a,h-0b
$ squeue -j 51
             JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
                51      open    hello     bozo  CG       1:01      1 h-1
$ squeue -j 51
             JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
$ cat hello.out
h-1
h-0b
h-3
h-0a
h-2
$
```

Figure 5: Interaction with SLURM from the shell.

An OpenMPI Job (Distributed Memory)

OpenMPI is an open source implementation of the MPI-2 (message passing interface version 2) specification that is developed and maintained by a consortium of academic, research, and industry partners. MPI is a library of routines and data structures implementing message passing programming on distributed memory computers. This version of programming parallelism is so called *single program, multiple data (SPMD)* a subset of *multiple instruction,*

```
#!/bin/sh
#
# hello-mpi.sh:
#
# a little shell script
# to run an OpenMPI job
#
#SBATCH --job-name=hello-openmpi
#SBATCH --workdir=/homes/bozo/test
#SBATCH --output=hello-openmpi.out
#
## resource requirements:
#
#SBATCH --partition=open
#SBATCH --ntasks=8
#SBATCH --time=00:05:00
#SBATCH --mem-per-cpu=2000mb

. /share/rc/openmpi
srun hello-openmpi

exit 0
```

Figure 6: A shell script to run an OpenMPI Job.

multiple data (MIMD). The distributed memory computer in this case, is a collection of independent computers that communicate with one another using TCP/IP over Ethernet or more exotic hardware. Processes execute simultaneously on different processors passing messages to move memory between processes or synchronize their operations. A simple OpenMPI hello world program is displayed in Figure 14 at the end of the document.

A SLURM shell script to execute the OpenMPI program on 8 cores is represented in Figure 6. Note that line 19 asks the shell to run the file `/share/rc/openmpi`. Much like the script used above in Figure 1 to setup up SLURM environment variables, the `/share/rc/openmpi` script sets up the shell environment to correctly run OpenMPI programs.

The SLURM managed OpenMPI job generates output in file `hello-openmpi.out` that looks something like the output depicted in Figure 7.

```
$ . /share/rc/openmpi
$ mpicc hello-openmpi.c -o hello-...
$ sbatch hello-openmpi.sh
Submitted batch job 56
$ cat hello-openmpi.out
0: We have 8 processors
0: Hello 1! Processor 1 reporting...
0: Hello 2! Processor 2 reporting...
0: Hello 3! Processor 3 reporting...
0: Hello 4! Processor 4 reporting...
0: Hello 5! Processor 5 reporting...
0: Hello 6! Processor 6 reporting...
0: Hello 7! Processor 7 reporting...
$
```

Figure 7: Shell interaction to compile and run Wikipedia OpenMPI hello

A OpenMP Job (Shared Memory)

OpenMP is an application programming interface that supports programming shared memory multiprocessors although it apparently can be extended to non-shared memory systems. This API can be accessed from C, C++, and FORTRAN using the *GCC compiler suite*. This model of parallel programing involves the use of threads of control within a single Unix process. That is multiple threads of control (program counter, stack, registers) can run simultaneously within a process, sharing process memory. Communication between the threads is through the shared memory of the process. The thread and synchronization primitives are specified in the code with compiler directives. Examples of the compiler directives are on lines 7 and 11 of the Wikipedia OpenMP hello world program in Figure 15 at the end of the document. This is in contrast to a Pthread based program where the implementation is accessed through library functions and variables.

```
#!/bin/sh
#
# hello-openmp.sh:
#
# a little shell script
# to run an OPENMP job
#
#SBATCH --job-name=hello-openmp
#SBATCH --workdir=/homes/jay/test
#SBATCH --output=hello-openmp.out
#
#SBATCH --partition=open
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --time=00:05:00
#SBATCH --mem-per-cpu=2000mb
#
export OMP_NUM_THREADS=\
$SLURM_CPUS_PER_TASK
./hello-openmp
exit 0
```

Figure 8: A shell script to run an OpenMP Job. The export of the shell environment variable is on two lines purely for readability in this sidebar.

In this shared memory multi-process(or) type job, it is important to clue SLURM into the organization of processor cores per task (process) so that resources are allocated correctly. Note on lines 15 and 16 of the `sbatch` shell script in Figure 8, SLURM is informed that the OpenMP job is to be a single process essentially with 4 threads, hopefully executing on separate processor cores. Lines 20 and 21 communicate the number of threads allowed to the OpenMP runtime. Note on line 13 of Figure 9 that if we change the number of cpus per task to a number greater than we happen to know is possible SLURM will reject the job.

```
$ gcc -fopenmp hello-openmp.c -o hello-openmp
$ sbatch hello-openmp.sh
Submitted batch job 62
$ squeue -i 4 --format "%.5i %.9P %.8j %.8u %.2t %.10M %.6D"
             JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
                62      open hello-op     bozo   R       0:00      1 h-0a
$  cat hello-openmp.out
Hello World from thread 0
Hello World from thread 1
Hello World from thread 3
Hello World from thread 2
There are 4 threads
$ sbatch hello-openmp.sh # change cpus-per-task=5 and re-submit the job
sbatch: error: Batch job submission failed: Requested node configuration is not available
$
```

Figure 9: Shell interaction showing compilation and execution of OpenMP hello world. Note the `-fopenmp` command line option to Gcc.

A Pthread Job (Shared Memory)

Posix threads (pthreads) are a standard cross platform implementation of threads for C-like languages. Threads are an independent flow of program control within a single process, sharing the address space and other resources of the process. The overhead of launching a new thread is considerably lower than starting up a new process. Communication between threads using common address space can be very efficient. Pthreads provides a fairly low level programming interface with fine-grained control over thread management and synchronization. Although Pthread programs can be difficult to manage, performance gains can be realized with careful programming of selected applications.

```
#!/bin/sh
#
# hello-pthread.sh:
#
# a little shell script
# to run a pthread job
#
#SBATCH --job-name=hello-pthread
#SBATCH --workdir=/homes/jay/test
#SBATCH --output=hello-pthread.out
#
#SBATCH --partition=open
#SBATCH --ntasks=8
#SBATCH --time=00:05:00
#SBATCH --mem-per-cpu=2000mb

hostname
./hello-pthread
exit 0
```

Figure 10: A shell script to run an Pthread job under SLURM.

```
$ gcc -pthread hello-pthread.c -o hello-pthread
$ sbatch hello-pthread.sh
$ squeue
Submitted batch job 60
Thu Aug  1 11:40:20 2013
JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
   60      open hello-pt     bozo   R       0:00      2 h-0a,h-0b
$ cat hello-pthread.out
h-0a
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread 0!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread 1!
Hello World! It's me, thread 2!
Hello World! It's me, thread 3!
Hello World! It's me, thread 4!
In main: creating thread 5
In main: creating thread 6
In main: creating thread 7
Hello World! It's me, thread 6!
Hello World! It's me, thread 7!
Hello World! It's me, thread 5!
```

Figure 11: Shell interaction showing compilation and execution of Pthread hello world.

An Interactive Job

For long running applications where the executing process is large, it may be desirable to locate the program on each individual node rather than a shared file server in order to reduce network traffic, particularly if for some reason the system is over paging. `Sbcast` can be used to distribute data files as well. In this example however, the compiled OpenMP based Wikipedia hello world program is sent to the nodes to be run interactively within the SLURM resource allocation. `salloc` creates a SLURM job allocation on the requested nodes in the selected partition. Lines 6 and 10 of our example illustrate that we can now `srun` commands on each node. On line 14 we broadcast our `hello-openmp` executable program to all our allocated nodes. Then we run the program with `srun` *without* creating a shell script to use with `sbatch`.

```
$ for i in h-3 h-4 h-5 ; do
> ssh $i mkdir /scr1/bozo
> done
$ salloc -p open -w h-3,h-4,h-5
salloc: Granted job allocation 66
$ srun hostname
h-3
h-5
h-4
$ srun date
Mon Aug  5 11:27:48 PDT 2013
Mon Aug  5 11:27:48 PDT 2013
Mon Aug  5 11:27:48 PDT 2013
$ sbcast hello-openmp /scr1/bozo/hello-openmp
$ srun /scr1/bozo/hello-openmp
Hello World from thread 2
Hello World from thread 1
Hello World from thread 3
Hello World from thread 0
There are 4 threads
Hello World from thread 1
Hello World from thread 0
Hello World from thread 3
Hello World from thread 2
There are 4 threads
Hello World from thread 1
Hello World from thread 0
Hello World from thread 3
Hello World from thread 2
There are 4 threads
$ exit
exit
salloc: Relinquishing job allocation 66
```

Figure 12: Interactive allocation of resources, broadcast of executable, and execution of a job.

SLURM Multi-Prog

The multi-prog option to `srun` provides a simple method to specify a list of programs and command-line arguments to those programs to be run within a job allocation. In this example, an interactive job allocation will be requested for 3 nodes. 8 tasks will be run as specified in a file `hello-multiprog.conf`.

Note that multiple command line arguments may be passed to the command such as on lines 4, 6, 8 , 10. If the command to be executed is not on the search path of the shell then a full path name rooted at / should be used.

```
$ cat hello-multiprog.conf
#task        command                command line args
0        hostname
1        echo                task:%t dog
2        hostname
3        echo                task:%t cat
4        hostname
5        echo                task:%t fish
6        hostname
7        echo                task:%t bird
$ salloc -p open -w h-3,h-4,h-5
salloc: Granted job allocation 70
$ srun -n8 --multi-prog hello-multiprog.conf
task:1 dog
h-3
h-3
task:7 bird
task:3 cat
h-5
task:5 fish
h-4
$ exit
salloc: Relinquishing job allocation 70
salloc: Job allocation 70 has been revoked.
$
```

Figure 13: A sample interaction with the shell that allocates SLURM resources and runs a job with a `multiprog` configuration.

Compiling and Running CUDA Jobs

Compiling and Running Φ Jobs

References

[1] Blaise Barney. Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp/, July 2013.

[2] Blaise Barney. Message passing interface (MPI). https://computing.llnl.gov/tutorials/mpi/, July 2013.

[3] Blaise Barney. OpenMP. https://computing.llnl.gov/tutorials/openmp, Jun 2013.

[4] Blaise Barney. Posix threads programming. https://computing.llnl.gov/tutorials/pthreads/, January 2013.

[5] Consortium des Équipments de Calcul Intensif. http://www.ceci-hpc.be/slurm_tutorial.html, 2013.

[6] Joe Landman. OpenMP in 30 minutes. *Linux Magazine*, December 2007.

[7] High Performance Computing Center North. Scripts, job submission files. https://www.hpc2n.umu.se/batchsystem/slurm-scripts, 2013.

[8] SchedMD Team. SLURM tutorials. https://www.schedmd.com/slurmdocs/tutorials.html, January 2013.

```
/*
  "Hello World" MPI Test Program
*/
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[]) {
  char idstr[32];
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;
  /* MPI programs start with MPI_Init; all 'N' processes exist thereafter */
  MPI_Init(&argc,&argv);
  /* find out how big the SPMD world is */
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  /* and this processes' rank is */
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /* At this point, all programs are running equivalently, the rank
     distinguishes the roles of the programs in the SPMD model, with
     rank 0 often used specially... */
  if(myid == 0) {
    printf("%d: We have %d processors\n", myid, numprocs);
    for(i=1;i<numprocs;i++) {
      sprintf(buff, "Hello %d! ", i);
      MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
    }
    for(i=1;i<numprocs;i++) {
      MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
      printf("%d: %s\n", myid, buff);
    }
  } else {
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strncat(buff, idstr, BUFSIZE-1);
    strncat(buff, "reporting for duty\n", BUFSIZE-1);
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
  }

  /* MPI programs end with MPI Finalize; this is a weak synchronization point */
  MPI_Finalize();
  return 0;
}
```

Figure 14: OpenMPI hello world from Wikipedia.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  int th_id, nthreads;
  #pragma omp parallel private(th_id)
  {
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);
    #pragma omp barrier
    if ( th_id == 0 ) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n",nthreads);
    }
  }
  return EXIT_SUCCESS;
}
```

Figure 15: OpenMP hello world.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS     8

void *TaskCode(void *argument)
{
   int tid;

   tid = *((int *) argument);
   printf("Hello World! It's me, thread %d!\n", tid);

   /* optionally: insert more useful stuff here */

   return NULL;
}

int main(void)
{
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int rc, i;

   /* create all threads */
   for (i=0; i<NUM_THREADS; ++i) {
      thread_args[i] = i;
      printf("In main: creating thread %d\n", i);
      rc = pthread_create(&threads[i], NULL, TaskCode, (void *) &thread_args[i]);
      assert(0 == rc);
   }

   /* wait for all threads to complete */
   for (i=0; i<NUM_THREADS; ++i) {
      rc = pthread_join(threads[i], NULL);
      assert(0 == rc);
   }

   exit(EXIT_SUCCESS);
}
```

Figure 16: Pthread hello world from Wikipedia.