جامعة نيويورك أبوظبي

NYU | ABU DHABI

Getting Started with *Dalma*

High Performance Computing NYUAD

November 2017 v6.1

SLURM: Interactive Job

- You can run interactively on either serial or parallel partitions
- For parallel partition the shell will be run on first allocated node
- The same limitations for resource limits apply as for non interactive jobs
- If the system is full you will wait...

```
$> srun --pty -n 1 /bin/bash
```

- Start an interactive bash on a compute node
- Node can be shared by someone elses job
- It will be allocated in the ‘serial’ partition

```
$> srun --pty -n 1 --exclusive  /bin/bash
```

- Start an interactive bash on a compute node
- Node is **NOT** shared with anyone
- It will be allocated in the ‘serial’ partition

```
$> srun --pty -N2 --exclusive  /bin/bash
```

- Start an interactive bash on 2 compute nodes
- Nodes are **NOT** available to anyone else
- It will be allocated in the ‘parallel’ partition

SLURM: Serial Job

```
#!/bin/bash

# Set maximum run time (default depends on partition)
# Here ask for 1 hour and 30 minutes
#SBATCH -t 1:30:00

# Set maximum memory (default depends on partition)
# Here asking for 2GB (default unit is MB, can use GB suffix too)
##SBATCH --mem=2000
#SBATCH --mem=2GB

# Ask for specific node type (sse or avx2)
# serial partition default "sse"
# parallel partition default "avx2"
#SBATCH -C avx2

# Ask SLURM to give you exclusive access to node
# You do this when you need to publish a paper and
# must quote performance / time measurements
# default is shared access
##SBATCH --exclusive

# Number of tasks to run (default is 1)
#SBATCH -n 1

# Number of cores per task (default is 1)
#SBATCH -c 1

# Partition (default=serial)
#SBATCH -p serial

# load modules (as needed by the application)
module load atlas

./serial-application
```

This is what a typical serial job looks like.

The "#SBATCH" lines tell SLURM how many cores to use, which partition to use, etc. Meanwhile the "module load" lines (discussed later) prepare the software environment that is needed to run the "serial-application".

The "#!/bin/bash" indicates that this is a shell script. It is a good practice to test your shell script prior to submitting as many times there may be errors in your script. For this you start an interactive session through SLURM (as seen the previous slide) and simply execute the script manually.

On Dalma we strongly suggest that users submit jobs from their /scratch disk space.

When submitted through SLURM the script will be launched in the directory where the "sbatch" command was used.

By default serial jobs share nodes (eg multiple serial jobs run on the same node concurrently). This can be overriden if you require time measurements for publication purposes.

SLURM: Multi-Threaded Job

```
#!/bin/bash

# Number of tasks to run (default is 1)
#SBATCH -n 1

# Number of cores per task (default is 1)
#SBATCH -c 6

# Partition (default=serial)
#SBATCH -p serial

# load modules (as needed by the application)
#module load XXX

# Use SLURM cores-per-task to set OMP_NUM_THREADS
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

./multi-threaded-application
```

This is what a multi-threaded / OpenMP job looks like.

All parameters, modules, etc. are identical as for serial jobs. Except that you need to define how many cores a task requires. Note that multi-threaded jobs run on the serial partition, and that nodes can be shared with other serial or multi-threaded applications.

We strongly suggest that when submitting OpenMP jobs you set "OMP_NUM_THREADS" to the value of "SLURM_CPUS_PER_TASK" to avoid any discrepancy in your execution – people often forget to change one when they update the other.

SLURM: OpenMPI Job

Method 1 : OpenMPI

```
#!/bin/bash

# Number of MPI threads (default 1 core per MPI thread)
#SBATCH –n 56

# Use parallel partition
# Parallel partition nodes are exclusive
# Parallel partition nodes are only avx2 types
#SBATCH -p parallel

# No need to load the MPI module
# But may need to load other modules here

# Safety always!
module purge

srun ./mpi-application
```

Method 2 : OpenMPI

```
#!/bin/bash

#SBATCH –n 56
#SBATCH -p parallel
module purge
mpiexec ./mpi-application
```

OpenMPI Jobs can be run in 2 ways.

Again all parameters, modules, etc. are identical as for serial jobs. And you also need to define how many cores a task requires. OpenMPI jobs can only be executed on the parallel partition, where all nodes are "avx2" type. The parallel partition will not allow more than 1 job per node, so you run in exclusive mode always.

There's a tight integration between openmpi and SLURM which allows you to use "srun" to launch OpenMPI applications without the need to use "mpiexec" / "mpirun".

The second method is to load the OpenMPI module and to use "mpiexec" to launch the MPI threads on all nodes.

In both cases you don't need to specify the number of MPI threads, or to specify the list of nodes to use.

Moreover, using this method the current working directory and environment variables (and loaded modules) are automatically transmitted to all MPI threads.

SLURM: mvapich2 Job

Method 1 : mvapich2

```
#!/bin/bash

# Number of MPI threads (default 1 core per MPI thread)
#SBATCH –n 56

# Use parallel partition
# Parallel partition nodes are exclusive
# Parallel partition nodes are only avx2 types
#SBATCH -p parallel

# No need to load the MPI module
# But may need to load other modules here

# Safety always!
module purge

srun ./mpi-application
```

Method 2 : OpenMPI

```
#!/bin/bash

#SBATCH –n 56
#SBATCH -p parallel
module purge
mpiexec ./mpi-application
```

OpenMPI Jobs can be run in 2 ways.

Again all parameters, modules, etc. are identical as for serial jobs. And you also need to define how many cores a task requires. OpenMPI jobs can only be executed on the parallel partition, where all nodes are "avx2" type. The parallel partition will not allow more than 1 job per node, so you run in exclusive mode always.

There's a tight integration between openmpi and SLURM which allows you to use "srun" to launch OpenMPI applications without the need to use "mpiexec" / "mpirun". Then you don't need to specify the number of MPI threads, or to specify the list of nodes to use. Using this method the current working directory and environment variables (and loaded modules) are automatically transmitted to all MPI threads.

The second method is to load the OpenMPI module and to use "mpiexec" to launch the MPI threads on all nodes.

TO DO

SLURM: Hybrid MPI Job

Hybrid jobs (MPI+ OpenMP) can be started in 4 different ways.

You can use "srun" or "mpiexec" to launch your jobs, and then you can set the number of cores per task (-c) or the number of tasks-per-node to control the number of MPI threads per node / number of cores per MPI thread.

These methods work for both OpenMPI and MVAPICH2.

Method 1

```
#!/bin/bash
#SBATCH –n 2
#SBATCH -p parallel
#SBATCH –c 28

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
module purge
srun ./mpi-application
```

Method 2

```
#!/bin/bash
#SBATCH –n 2
#SBATCH -p parallel
#SBATCH --tasks-per-node=1

export OMP_NUM_THREADS=28
srun ./mpi-application
```

Method 3

```
#!/bin/bash
#SBATCH –n 2
#SBATCH -p parallel
#SBATCH –c 28

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
module purge
mpiexec./mpi-application
```

Method 4

```
#!/bin/bash
#SBATCH –n 2
#SBATCH -p parallel
#SBATCH --tasks-per-node=1

export OMP_NUM_THREADS=28
mpiexec ./mpi-application
```

SLURM: Job Array

```
#!/bin/bash
# Processing for each experiment requires a single core
# The input files are named file.1, file.2, ..., file.50
#SBATCH -p serial
#SBATCH -n 1
#SBATCH -a 1-50

module purge
./my-code file.${SLURM_ARRAY_TASK_ID}
```

Frequently we need to run the same application / job script several times for either:

- processing multiple independent input files
- looking for an optimal set of input parameters for the same input file (eg parametric analysis)

For instance you may need to process the results of 50 experiments. Then you can (1) prepare a single job script which processes all 50 experiments, one at a time, or (2) prepare 50 job scripts which process concurrently the experiments, one experiment per job.

In either case this is not an optimal use of your time.

The solution is to process your experiments using a job array; the script will be executed 50 times but each time it runs the counter will be different. You can use the counter to refer to each experiment input as in the job script to the left.

جامعة نيويورك أبوظبي
NYU | ABU DHABI

SLURM: Job Array

```
> sbatch job.sh
Submitted batch job 466546
> squeue -u bm102
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
     466546_[1-50]   ser_std   ja1.sh    bm102 PD       0:00      1 (Priority)
> squeue -u bm102
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          466546_1   ser_std   ja1.sh    bm102 CG       0:07      1 compute-21-2
          466546_2   ser_std   ja1.sh    bm102 CG       0:06      1 compute-21-11
          466546_3   ser_std   ja1.sh    bm102 CG       0:07      1 compute-21-11
          466546_4   ser_std   ja1.sh    bm102 CG       0:06      1 compute-21-11
          466546_5   ser_std   ja1.sh    bm102 CG       0:09      1 compute-21-13
          466546_6   ser_std   ja1.sh    bm102 CG       0:10      1 compute-21-13
          466546_7   ser_std   ja1.sh    bm102 CG       0:08      1 compute-21-13
          466546_8   ser_std   ja1.sh    bm102 CG       0:08      1 compute-21-13
          466546_9   ser_std   ja1.sh    bm102 CG       0:10      1 compute-21-13
         466546_10   ser_std   ja1.sh    bm102 CG       0:10      1 compute-21-13
         466546_11   ser_std   ja1.sh    bm102 CG       0:09      1 compute-21-13
         466546_12   ser_std   ja1.sh    bm102 CG       0:07      1 compute-21-13
         466546_13   ser_std   ja1.sh    bm102 CG       0:05      1 compute-21-13
         466546_14   ser_std   ja1.sh    bm102 CG       0:08      1 compute-21-13
         466546_15   ser_std   ja1.sh    bm102 CG       0:07      1 compute-22-3
         466546_16   ser_std   ja1.sh    bm102 CG       0:06      1 compute-22-3
         466546_17   ser_std   ja1.sh    bm102 CG       0:08      1 compute-22-3
         466546_18   ser_std   ja1.sh    bm102 CG       0:09      1 compute-22-3
         466546_19   ser_std   ja1.sh    bm102 CG       0:09      1 compute-22-3
         466546_20   ser_std   ja1.sh    bm102 CG       0:05      1 compute-22-3
         466546_21   ser_std   ja1.sh    bm102 CG       0:08      1 compute-22-3
         466546_22   ser_std   ja1.sh    bm102 CG       0:05      1 compute-1-4
         466546_23   ser_std   ja1.sh    bm102 CG       0:06      1 compute-1-4
         466546_24   ser_std   ja1.sh    bm102 CG       0:06      1 compute-1-4
         466546_25   ser_std   ja1.sh    bm102 CG       0:09      1 compute-1-4
         466546_26   ser_std   ja1.sh    bm102 CG       0:07      1 compute-1-4
         466546_27   ser_std   ja1.sh    bm102 CG       0:06      1 compute-1-4
         466546_28   ser_std   ja1.sh    bm102 CG       0:05      1 compute-8-1
         466546_29   ser_std   ja1.sh    bm102 CG       0:08      1 compute-8-1
         466546_30   ser_std   ja1.sh    bm102 CG       0:09      1 compute-8-1
         466546_31   ser_std   ja1.sh    bm102 CG       0:08      1 compute-8-1
         466546_32   ser_std   ja1.sh    bm102 CG       0:07      1 compute-8-1
         466546_33   ser_std   ja1.sh    bm102 CG       0:07      1 compute-8-1
         466546_34   ser_std   ja1.sh    bm102 CG       0:07      1 compute-9-15
         466546_35   ser_std   ja1.sh    bm102 CG       0:07      1 compute-9-15
         466546_36   ser_std   ja1.sh    bm102 CG       0:06      1 compute-9-15
         466546_37   ser_std   ja1.sh    bm102 CG       0:06      1 compute-9-15
         466546_38   ser_std   ja1.sh    bm102 CG       0:06      1 compute-9-15
         466546_39   ser_std   ja1.sh    bm102 CG       0:08      1 compute-9-15
         466546_40   ser_std   ja1.sh    bm102 CG       0:09      1 compute-9-15
         466546_41   ser_std   ja1.sh    bm102 CG       0:07      1 compute-9-15
         466546_42   ser_std   ja1.sh    bm102 CG       0:08      1 compute-11-13
         466546_43   ser_std   ja1.sh    bm102 CG       0:09      1 compute-11-13
         466546_44   ser_std   ja1.sh    bm102 CG       0:10      1 compute-11-13
         466546_45   ser_std   ja1.sh    bm102 CG       0:09      1 compute-11-13
         466546_46   ser_std   ja1.sh    bm102 CG       0:08      1 compute-11-13
         466546_47   ser_std   ja1.sh    bm102 CG       0:09      1 compute-11-13
         466546_48   ser_std   ja1.sh    bm102 CG       0:06      1 compute-11-13
         466546_49   ser_std   ja1.sh    bm102 CG       0:10      1 compute-11-13
         466546_50   ser_std   ja1.sh    bm102 CG       0:09      1 compute-4-9
```

A job array looks like a normal job, except that the jobid is expended with a taskid – eg 466546_[1-50] and 466565_1.

As you can see in this example we got all 50 jobs to run concurrently as the system was idle during this test. But in practice you will see some jobs running, and some jobs waiting for their turn to run.

Job array processing can be used for serial, multi-threaded and MPI jobs alike.

You can kill all jobs in a job array with "scancel 466546" or an individual job with "scancel 466546_42" for example. You can also refer to a range of taskids such as "scancel 466546_[3-17]".

Each job in the job array will have its own output file (and error file if you are submitting with "-e"). So you have to be careful with not exceeding the 1000 files per directory limit! You may split the outputs in multiple directories or use the parallel job array tool described next.

```
> ls slurm-466546_*
slurm-466546_10.out  slurm-466546_1.out   slurm-466546_29.out  slurm-466546_38.out  slurm-466546_47.out
slurm-466546_11.out  slurm-466546_20.out  slurm-466546_2.out   slurm-466546_39.out  slurm-466546_48.out
slurm-466546_12.out  slurm-466546_21.out  slurm-466546_30.out  slurm-466546_3.out   slurm-466546_49.out
slurm-466546_13.out  slurm-466546_22.out  slurm-466546_31.out  slurm-466546_40.out  slurm-466546_4.out
slurm-466546_14.out  slurm-466546_23.out  slurm-466546_32.out  slurm-466546_41.out  slurm-466546_50.out
slurm-466546_15.out  slurm-466546_24.out  slurm-466546_33.out  slurm-466546_42.out  slurm-466546_5.out
slurm-466546_16.out  slurm-466546_25.out  slurm-466546_34.out  slurm-466546_43.out  slurm-466546_6.out
slurm-466546_17.out  slurm-466546_26.out  slurm-466546_35.out  slurm-466546_44.out  slurm-466546_7.out
slurm-466546_18.out  slurm-466546_27.out  slurm-466546_36.out  slurm-466546_45.out  slurm-466546_8.out
slurm-466546_19.out  slurm-466546_28.out  slurm-466546_37.out  slurm-466546_46.out  slurm-466546_9.out
```

SLURM: Parallel Job Array

```
matlab --nodisplay --nodesktop -r "mycode.m 2.1;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.15;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.17;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.18;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.2;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.4;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.7;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.8;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.81;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.9;exit;"

...

matlab --nodisplay --nodesktop -r "mycode.m 2.91;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.92;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.93;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.94;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.95;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.96;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.97;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.98;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.99;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.991;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.992;exit;"
matlab --nodisplay --nodesktop -r "mycode.m 2.993;exit;"
```

There are times, however, when even a job array doesn't really fit your needs. For instance, you may be looking at running 300,000 very short running jobs. Or maybe, the complexity to search through all the output files orhaving to manually "glue" them back together into a single output file is too demanding. Or maybe it's too difficult to match SLURM's taskids to actual input file names or parametric analysis. Or more probably the time delay between each jobs being scheduled to run is slowing down your processing as your jobs are really short.

For these cases Dalma has the "parallel job array" system– an NYU Abu Dhabi in-house tool – designed to make running large number of jobs even simpler.

```
./process-one-resuls file.1
./process-one-resuls file.2
./process-one-resuls file.3

...

./process-one-result file.50000
```

SLURM: Parallel Job Array

```
> cat list3
echo processing 1
echo processing 2
...
echo processing 100000

> slurm_parallel_ja_submit.sh list3
Input: list3
Actual maximum number of nodes that will be used: 8
Submitting parallel job array using the following modules:
Currently Loaded Modulefiles:
  1) NYUAD/3.0
Submitting parallel job array with OMP_NUM_THREADS= 4
Submitted batch job 466656

> squeue -u bm102
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
      466656_[1-8]   ser_std   sbatch    bm102 PD       0:00      1 (Priority)

> ls list3-466656_*
list3-466656_1.err  list3-466656_2.err  list3-466656_3.err  list3-466656_4.err
list3-466656_5.err  list3-466656_6.err  list3-466656_7.err  list3-466656_8.err
list3-466656_1.out  list3-466656_2.out  list3-466656_3.out  list3-466656_4.out
list3-466656_5.out  list3-466656_6.out  list3-466656_7.out  list3-466656_8.out

> cat list3-466656_1.out
============ job 1 ============
processing 1
============ job 2 ============
processing 2
...
processing 12499
============ job 12500 ============
processing 12500

> cat list3-466656_8.out
============ job 87501 ============
processing 87501
============ job 87502 ============
processing 87502
...
============ job 99999 ============
processing 99999
============ job 100000 ============
processing 100000
```

To submit a parallel job array you need to prepare a file where each line represents a single job to execute. You can have multiple Linux commands on a line as:

```
myprogram1
export V=42; myprogram2 | myprogram3
myprogram4; myprogram5 < inp
myprogram6 > out
```

Then you pass the file to the tool and it automatically submits the job array for you.

The tool splits all jobs into groups and proceeds to run them as a normal job array. During execution all jobs within a group are split across the cores of a compute node and run in parallel.

The output is gathered into files, 1 output file per group and 1 error file per group; the output is ordered to make things easy for you.

You get significant performance gains by eliminating the job scheduling overhead and the queuing delay for each job.

SLURM: Parallel Job Array

```
> slurm_parallel_ja_submit.sh –h
slurm_parallel_ja_submit.sh options:
    -C <constraint> (avx2 or sse)
    -t <time limit HH:MM:SS>
    -p <partition>
    -N <number of nodes>
```

The parallel job array tool allows you to set the time limit for each node – not each job!, the processor type (sse, or avx2), the partition – useful if your research group has its "compute condo", and the number of nodes to use.

The parallel job array tool determines an optimal number of nodes to use (actually job array tasks). It groups all jobs into groups processes them using conventional job array. The number of "nodes" corresponds actually to the number of groups, which is the number of tasks for the job array. The time limit parameter applies to the groups – eg an entire group must complete within the time limit.

جامعة نيويورك أبوظبي
NYU | ABU DHABI

SLURM: Parallel Job Array

```
> module load matlab
> slurm_parallel_job_array_submit.sh matlabjobs


> export OMP_NUM_THREADS=4
> slurm_parallel_job_array_submit.sh jatest
```

The parallel job array tool propagates your environment, and loaded modules, to all jobs. So to execute the previous matlab example you need to load the matlab software module prior to launching the parallel job array.

The tool also support OpenMP jobs, so you can set the number of threads before launching your parallel job array.

SLURM: Parallel Job Array

```
> slurm_parallel_ja_submit.sh list
Input: list
Actual maximum number of nodes that will be used: 8
Submitting parallel job array using the following modules:
Currently Loaded Modulefiles:
  1) NYUAD/3.0                2) all                          3)
gcc/4.9.3                4) mvapich2/gcc_4.9.3/2.2b
Submitting parallel job array with OMP_NUM_THREADS= 1
Submitted batch job 466638



> slurm_parallel_ja_submit.sh -N 12 list
Entered number of nodes to use: 12
Input: list
Actual maximum number of nodes that will be used: 12
Submitting parallel job array using the following modules:
Currently Loaded Modulefiles:
  1) NYUAD/3.0                2) all                          3)
gcc/4.9.3                4) mvapich2/gcc_4.9.3/2.2b
Submitting parallel job array with OMP_NUM_THREADS= 1
Submitted batch job 466644



> export OMP_NUM_THREADS=4
> slurm_parallel_ja_submit.sh -N 12 list
Entered number of nodes to use: 12
Input: list
Actual maximum number of nodes that will be used: 12
Submitting parallel job array using the following modules:
Currently Loaded Modulefiles:
  1) NYUAD/3.0                2) all                          3)
gcc/4.9.3                4) mvapich2/gcc_4.9.3/2.2b
Submitting parallel job array with OMP_NUM_THREADS= 4
Submitted batch job 466646
```

By default the tool will allow up to 8 "nodes" (groups). You can increase the number of nodes when there is a very large number of jobs to process to run faster, or when the groups can't finish within the time limit.

SLURM: Parallel Job Array

```
> cat ja.sh
#!/bin/bash
#SBATCH -p serial
#SBATCH -n 1
#SBATCH -a 1-100
sleep 5
echo -n $SLURM_ARRAY_TASK_ID " "
date

> sbatch ja.sh



> cat list4
./job.sh
./job.sh
...
./job.sh

> slurm_parallel_ja_submit.sh list4
```

For compatibilty with existing job array scripts you can use the SLURM_ARRAY_TASK_ID environment variable in your parallel job array processing.

Here the list4 input file contains the line "./ja.sh" 100 times.

An additional benefit of parallel job array processing is that you are not limited to SLURM's MaxSubmit and MaxJobs account / user limits.

Whereas you can submit a maximum of 200 jobs, and have up to 100 running concurrently using regular job array processing, using parallel job array you have no such limits.