Getting Started with Dalma

High Performance Computing NYUAD
• 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...

\$> \texttt{srun \ --pty \ -n \ 1 \ /bin/bash}

• Start an interactive bash on a compute node
  • Node can be shared by someone else's job
  • It will be allocated in the 'serial' partition

\$> \texttt{srun \ --pty \ -n \ 1 \ --exclusive \ /bin/bash}

• Start an interactive bash on a compute node
  • Node is \textbf{NOT} shared with anyone
  • It will be allocated in the 'serial' partition

\$> \texttt{srun \ --pty \ -N2 \ --exclusive \ /bin/bash}

• Start an interactive bash on 2 compute nodes
  • Nodes are \textbf{NOT} available to anyone else
  • It will be allocated in the 'parallel' partition
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.
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.
Method 1 : OpenMPI

```bash
#!/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
```

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.

Method 2 : OpenMPI

```bash
#!/bin/bash

#SBATCH --n 56
#SBATCH -p parallel

module purge

mpiexec ./mpi-application
```
Method 1 : mvapich2

```bash
#!/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
```

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.

Method 2 : OpenMPI

```bash
#!/bin/bash

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

```bash
#!/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

```bash
#!/bin/bash
#SBATCH --n 2
#SBATCH -p parallel
#SBATCH --tasks-per-node=1
export OMP_NUM_THREADS=28
srun ./mpi-application
```

Method 3

```bash
#!/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

```bash
#!/bin/bash
#SBATCH --n 2
#SBATCH -p parallel
#SBATCH --tasks-per-node=1
export OMP_NUM_THREADS=28
mpiexec ./mpi-application
```
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 (e.g., 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.
SLURM: Job Array

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.
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 or having 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.

```bash
./process-one-results file.1
./process-one-results file.2
./process-one-results file.3
...
./process-one-result file.50000
```
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.
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.

> 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 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_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.
For compatibility 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.