Overview

This article will cover the essential topics to run jobs on Dalma. To know more about Dalma hardware, check this page Cluster - Dalma.

Prerequisite

The operating system on Dalma is Linux. Make sure you know the basics. Useful links:

- [http://linuxcommand.org/](http://linuxcommand.org/)
- [https://www.edx.org/course/introduction-linux-linuxfoundationx-lfs101x-0](https://www.edx.org/course/introduction-linux-linuxfoundationx-lfs101x-0)

Usage Model

This is the structure of our HPC cluster, Dalma.

Your typical workflow on Dalma:

1. (One time only) Let us know your computational requirement.
2. (One time only) Apply an HPC account and pass our quiz.
3. If needed, transfer your input data to Dalma.
4. Log on to Dalma login nodes.
5. Submit jobs on login nodes.
6. Your jobs will queue for execution.
7. Once done, examine the output.

These steps are explained below.

Getting or Renewing an Account

Get a new account and pass the quiz as instructed here: Accounts.

The yearly renewal instructions are on the same page

It takes 2 business day to activate your account.

Access

Once your account is ready, you can access Dalma. With Linux or Mac in NYU AD/NY network, simply ssh in your local terminal:

```
ssh <your-NetID>@dalma.abudhabi.nyu.edu
```
If you use Windows or outside NYU AD/NY network, follow the instructions here: Access Dalma.

Storage

We have 4 storage systems for you: $HOME (/home/<NetID>), $SCRATCH (/scratch/<NetID>), $WORK (/work/<NetID> and $ARCHIVE (/archive/<NetID>).

In short, you should put all your data to $SCRATCH and run your jobs from there. Only a small persistent fraction to $HOME (e.g., source code, executables, Python / R packages...). For long-term storage, archive them to $ARCHIVE. $WORK is not visible on compute nodes but mountable on your local workstation, best suited to quick post-processing, analysis and visualization, without moving your data.

Backing up is a user's own responsibility. E.g., if a user deleted something accidentally, we can not recover, unfortunately.

$HOME and $SCRATCH

$HOME and $SCRATCH can be accessed as follows:

```bash
# to check where you are
pwd
# to change the current working directory to $HOME
cd $HOME
# is equivalent to
cd /home/<your-NetID>
# or
cd
# for $SCRATCH
cd $SCRATCH
```

We urge our users to clean up their storage regularly.

Retention Policy Applies
Files older than 90 days at $SCRATCH will be deleted.

Running jobs from /home is a serious violation of HPC policy. Any users who intentionally violate this policy will get their account suspended. /home SSDs are not designed for scratch disks, it will kill the SSDs quickly.

$WORK

$WORK can be accessed on login nodes as follows:

```bash
cd $WORK
# is equivalent to
cd /work/<your-NetID>
```

$WORK can also be mounted on your workstation, Linux and Mac only. Instructions are in this page. Mount $WORK with SSHFS

We urge our users to clean up their storage regularly.

Retention Policy Applies
$ARCHIVE

The $ARCHIVE can be accessed as The guide to Archive on Dalma.

Summary

The usages are summarized below.

<table>
<thead>
<tr>
<th></th>
<th>/home</th>
<th>/scratch</th>
<th>/work</th>
<th>/archive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use for storing</td>
<td>source code / executable / perl-python-R packages</td>
<td>data</td>
<td>anything</td>
<td>anything</td>
</tr>
<tr>
<td>Accessible From</td>
<td>login / compute</td>
<td>login / compute</td>
<td>login</td>
<td>login</td>
</tr>
<tr>
<td>Use to Run Jobs</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Retention Time (Days)</td>
<td>No Limit</td>
<td>90</td>
<td>120</td>
<td>No Limit</td>
</tr>
<tr>
<td>Mountable</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Default Quota</td>
<td>5GB, 100K Files</td>
<td>5TB, 500K Files</td>
<td>5TB, 500K Files</td>
<td>5TB, 125K Files</td>
</tr>
</tbody>
</table>

Default block quota and inode quota, more precisely.

Run myquota in the terminal on Dalma to check your current usage and quota. Example output:

```
<table>
<thead>
<tr>
<th>DISK SPACE</th>
<th>filesystem</th>
<th>size</th>
<th>quota</th>
<th># FILES (1000's)</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>number</td>
<td>size</td>
<td>quota</td>
<td>number</td>
</tr>
</tbody>
</table>

-----------------------------------------------
//home 131KB 6442MB ( 0%)
0 100 ( 0%)
//scratch 220GB 5242GB ( 4%)
4 500 ( 1%)
//work 4KB 5242GB ( 0%)
0 512 ( 0%)
//archive 418GB 5242GB ( 8%)
3 125 ( 3%)
```

Data Transfer

You can use either Terminal or FileZilla (with a GUI) to transfer your data from/to Dalma: File Transfer using rsync and File Transfer using FileZilla.

To this point, you should be able to log in Dalma and upload/download data.

Hardware

Files older than 120 days at $WORK will be deleted.
Dalma consists of more than 12K CPU cores. But it is very unlikely that your code can scale up to use them all (contact us directly if you are confident). From the user perspective, here are the important specifications for most nodes:

- The CPU mode is Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, supporting AVX2.
- 28 CPU cores per node. Implications:
  - If your code doesn't support MPI, or you don't know what MPI is, use maximum 28 cores per job.
  - For MPI jobs using more than one node, always use a number of cores dividable by 28, to utilize the full nodes.
- 4 GB memory per core by default.

Contact us if you need special configuration (extra large memory, GPU, etc...)

Software

On HPC:

1. The Operating System is CentOS 7. Windows / Mac software is not supported.
2. No GUI. No display.

You can compile / install your own software, and/or use our Module system. For the latter, first check what applications are available.

```bash
# Run the following commands after logging in Dalma
module avail
```

Then you could select the desired software to load. The following example shows how to load a self-sufficient-single-application environment for gromacs.

```bash
# Run the following commands after logging in Dalma
module load gromacs
# or use the full module name with version
module load gromacs/5.0.4
```

The following example shows how to load an environment for compiling source code from scratch.

```bash
# Run the following commands after logging in Dalma
module load gcc
# multiple modules could be loaded in one line
module load openmpi fftw3
```

At this point, compilers like 'gcc', 'gfortran' and 'g++' are available, in a sense that the paths to those executables are prepended to $PATH. Also, paths to libraries files from FFTW3 will be prepended to $LD_LIBRARY_PATH.

If you cannot find a certain version of the software (for example, you are looking for Python 3, but only to find Python 2 is available), try running the following command to make all modules visible first.

```bash
# Run the following commands after logging in Dalma
module load all
module avail python
```

```
/share/apps/NYUAD/modules/ALL
----------------------------------
python/2.7.11 python/3.5.1
```
As you can see, Python 3 is available then. You could load Python 3 by loading the specific module.

```
module load python/3.5.1
```

At this point, you should be able to invoke the executable, e.g., `python`.

Alternatively, install Conda in your $HOME for hassle-free, independent Python environment. Follow this page: [Python - Create Your Own Environment using Anaconda](#)

---

**Running Jobs**

Now it is the exciting part. With input data and software ready, you can run your computational tasks now.

On HPC, you don't run it directly on the login nodes. Instead, you submit jobs on login nodes. These jobs will be queued to the system and executed eventually. Conceptually, each job is a 2-step process:

1. You request certain resources from the system. The most common resources are CPU cores.
2. With the assigned resources, you run your computational tasks.

There are two ways, interactive sessions or batch jobs.

**Interactive Sessions**

You could get an interactive session directly from your terminal, on compute nodes. Only short interactive jobs should be used (e.g., experimenting with new modifications to your Matlab code).

To start an interactive session, use `srun` command:

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

Then you can run your applications on the terminal directly. E.g.,

```
[gh50@login-0-1 ~]$ srun --pty -n 1 /bin/bash
srun: job 775175 queued and waiting for resources
srun: job 775175 has been allocated resources
[gh50@compute-21-1 ~]$
```

In a real scenario, the system might be exhausted with no available resources to you. You need to wait in this circumstance.

In this example, user gh50 requested 1 CPU core (-n 1) on login node (login-0-1). The system responded, assigned a job id (775175), queued the job and assigned 1 CPU core from one of the compute nodes (compute-21-1) to the user.

To exit the interactive session, type **Ctrl+d**, or

```
exit
```
Batch Job

Besides interactive sessions, a user can submit batch jobs to the system. For production jobs, batch jobs should be used.

A complete batch job workflow:

1. Write a job script, which consists of 2 parts:
   a. Resources requirement.
   b. Commands to be executed.
2. Submit the job.
3. Relax, have a coffee, log off if you wish. The computer will do the work.
4. Come back to examine the result.

Batch Job Script

A job script is a text file describing the job. As discussed, the first part tells how much resources you want. The second part is what you want to run. Choose one of the following examples to start with. If you are not sure, contact us.

Resources Limit

The cluster is shared among the whole university. The HPC steering committee decides each year on resources limit for each department. We at NYUAD HPC center implement these limits.

Typically, a user can ask for 48 hours, 700 CPU cores maximum per job.

If you ask for more resources than you can use, your job will stay in the queue forever. (e.g., you specify 10000 hours walltime in your job script)

If you have multiple jobs (which is very normal), your jobs will start either immediately if the system is free and the quotas for you and your department have not been exhausted.

A Job with 1 CPU Core

This is a very basic example, using only one CPU core.

```
#!/bin/bash
# Set number of tasks to run
#SBATCH --ntasks=1
# Walltime format hh:mm:ss
#SBATCH --time=00:30:00
# Output and error files
#SBATCH -o job.%J.out
#SBATCH -e job.%J.err

# **** Put all #SBATCH directives above this line! ****
# **** Otherwise they will not be effective! ****

# **** Actual commands start here ****
# Load modules here (safety measure)
module purge
# You may need to load gcc here .. This is application specific
# module load gcc
# Replace this with your actual command.
'serial-hello-world' for example
hostname
```

As you can see, it is a simple bash script, plus some lines on the top, starting with #SBATCH,
which are the Slurm directives.

Those Slurm directives specify resources required. E.g., ‘-ntasks=1’ is 1 CPU core. ‘--time=00:30:00’ means the maximum walltime is 30 mins. ‘-o job.%J.out’ is redirecting the stdout, usually your screen output, to a file called ‘job.%JOBID.out’. Why? Because the system will run your job in the background, hence no display.

Everything under the Slurm directives is normal Linux command. This example runs ‘hostname’, which will print the hostname. In reality, you should load your desired modules, and execute whatever you want to run.

Multithreading Job

Multithreading enables a process to spawn multiple threads to accelerate its execution. The most common multithreading model in HPC is OpenMP. If your application supports this (not sure? contact us to find out), you could use the below example.

```
#!/bin/bash
# Set number of tasks to run
#SBATCH --ntasks=1
# Set the number of CPU cores for each task
#SBATCH --cpus-per-task=4
# Walltime format hh:mm:ss
#SBATCH --time=00:30:00
# Output and error files
#SBATCH -o job.%J.out
#SBATCH -e job.%J.err

# **** Put all #SBATCH directives above this line! ****
# **** Otherwise they will not be effective! ****

# **** Actual commands start here ****
# Load modules here (safety measure)
module purge
# You may need to load gcc here .. This is application specific
# module load gcc

# If you are using OpenMP application, keep this line.
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# Replace this with your actual command. In this example, you should run a multithreading supported application
hostname
```

Comparing to the previous examples, there are 2 extra lines:

1. ‘#SBATCH --cpus-per-task=4’: this asks the system to assign 4 CPU cores per tasks. This number should be no larger than and a divisor of 28 (e.g., 2, 4, 7, 14, 28) because the majority of our nodes comes with 28 cores.
2. ‘export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK’: this tells your applications, if OpenMP supported, to use all the CPU cores assigned to your job, by spawning an exact number of OpenMP threads.

Remember, running a job is 2 steps process: 1. Request the resources. 2. Use the resources. This example is a perfect illustration. Run with what you requested, no more, no less.

Pure MPI Job

Now comes the pure MPI Jobs.
#!/bin/bash
# Set number of tasks to run
# This number should be divisible by 28. E.g., 56, 84, 112...
#SBATCH --ntasks=56
# Walltime format hh:mm:ss
#SBATCH --time=00:30:00
# Output and error files
#SBATCH -o job.%J.out
#SBATCH -e job.%J.err

# **** Put all #SBATCH directives above this line! ****
# **** Otherwise they will not be effective! ****

# **** Actual commands start here ****
# Load modules here (safety measure)
module purge
# You may need to load gcc here .. This is application specific
# module load gcc
# Replace this with your actual command.
'serial-hello-world' for example
srun hostname

Comparing to the 1 core example, there are 2 different lines:

1. `#SBATCH --ntasks=56`: This line requests 56 cores. **This number should be divisible by 28. E.g., 56, 84, 112...**
2. `srun hostname`: This tells your application to run with MPI support, utilizing all CPU cores requested.
   The old school 'mpiexec' or 'mpirun' are supported as well. But you need to load 'openmpi' module in this case.

Hybrid MPI Job

If your application support MPI + OpenMP hybrid parallelization, you could follow this example to submit a hybrid job.
#!/bin/bash
# Set number of tasks to run
#SBATCH --ntasks=56
# Set the number of CPU cores for each task
#SBATCH --cpus-per-task=4
# Walltime format hh:mm:ss
#SBATCH --time=00:30:00
# Output and error files
#SBATCH -o job.%J.out
#SBATCH -e job.%J.err

# **** Put all #SBATCH directives above this line! ****
# **** Otherwise they will not be effective! ****

# **** Actual commands start here ****
# Load modules here (safety measure)
module purge
# You may need to load gcc here .. This is application specific
# module load gcc

# If you are using Hybrid MPI + OpenMP application, keep this line.
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

# Replace this with your actual command.
'serial-hello-world' for example
srun hostname

In this case, the number of CPU cores requested is 56 (ntasks) * 4 (cpus-per-task) = 224. This number should be divisible by 28 to use all the cores on the nodes. As in the multithreading job example, make sure ‘cpus-per-task’ is a divisor of 28.

Job Array

This example shows how to submit a job array, consist of 100 jobs, with environmental variable SLURM_ARRAY_TASK_ID varies from 1 to 100.

#!/bin/bash
# Set number of tasks to run
#SBATCH --ntasks=1
# Walltime format hh:mm:ss
#SBATCH --time=00:30:00
# Output and error files
#SBATCH -o job.%J.out
#SBATCH -e job.%J.err
#SBATCH -a 1-100

# **** Put all #SBATCH directives above this line! ****
# **** Otherwise they will not be effective! ****

echo "I am running job $SLURM_ARRAY_TASK_ID"

Or you can varies SLURM_ARRAY_TASK_ID from 51 to 100.
#SBATCH -a 50-100

Or set the maximum number of simultaneously running tasks from the job array to 10.

#SBATCH -a 1-100%10

We only allow a maximum of 200 jobs in queue for any given user.

**Submitting a Job**

Once you have your job script prepared, you could use the command `sbatch` to submit your job.

```
sbatch <jobscript>
```

Let say if you saved your job script into a file called `job.sh`. Then you should run the following.

```
sbatch job.sh
```

After the submission, it will return the corresponding **job id**. E.g.,

```
[gh50@login-0-1 overview]$ sbatch threads-job.sh
Submitted batch job 775602
```

In this case, the job id is 775602. You can safely log off Dalma at this point. Once the system can accommodate your request, the script will be executed. The screen output will be saved to the files you specified in the job script.

**Checking Job Status**

**Before and During Job Execution**

This command shows all your current jobs.

```
squeue
```

Example output:

```
[gh50@login-0-1 ~]$ squeue -j 31408
JOBID PARTITION     NAME     USER ST TIME  NODES Nodelist(REASON)
31408 ser_std  job1.sh     gh50 R 0:02      1 compute-21-4
```

It means the job with Job ID 31408, has been running (ST: R) for 2 minutes on compute-21-4.
For more verbose information, use `scontrol show job`.

```
scontrol show job <jobid>
```

After Job Execution

Once the job is finished, the job can’t be inspected by `squeue` or `scontrol show job`. At this point, you could inspect the job by `sacct`.

```
sacct -j <jobid>
```

The following commands give you extremely verbose information on a job.

```
sacct -j <jobid> -l
```

Canceling a Job

If you decide to end a job prematurely, use `scancel`.

```
scancel <jobid>
```

**Use with Cautions**

To cancel all jobs from your account. Run this on Dalma terminal.

```
scancel -u <NetID>
```

That is. Up to this point, you should be able to run your computational tasks on Dalma. If there is any question, don’t hesitate to contact us (contacts on the right)!