Running Jobs with PBS (Ocelote and El Gato)

- PBS Overview (Ocelote and El Gato)
- Writing Job Scripts
  - Script Examples
  - Script Builder
  - Scheduler Options
- Monitor Jobs
- Running Multiple Jobs
  - Job Arrays
  - GNU Parallel
- MPI Performance
  - Intel
  - Other MPI Options

PBS Overview (Ocelote and El Gato)

- The Ocelote Quick Start page can provide enough information to run a job. More information is provided on this page. The HPC Supercomputer Overview page emphasizes the differences between the clusters. Our new Github repository has scripts including, but not limited to: Life Sciences, Array Jobs, GNU Parallel, and Singularity examples.

- All the integration on Ocelote is done with "bash" which means to get consistent results bash must be your shell. We recommend all users change their shell to "bash" by using the "uchsh" command.

The HPC systems use a queueing system to manage compute resources and schedule jobs. The batch system constantly monitors all of the queued and running jobs and all the compute resources available to schedule and run the jobs as quickly and efficiently as possible. For Ocelote and Elgato it is PBS Pro.

Jobs are submitted to the batch system using PBS scripts that specify the job's required resources such as number of nodes, cpus, memory, group, wallclock time.

The best source for documentation if you need more detailed information about PBS is directly from Altair:

- PBS Professional User's Guide
- PBS Professional Reference Guide

Writing Job Scripts
To correctly request resources, users must understand something of the configurations of nodes available in order to build the appropriate `select` statements. If the select statement requests hardware configurations that are not available, the job requests will fail or sit in the queue permanently. Refer to the `Compute Resources` page for the configurations paying particular attention to the number of cores and RAM per node. You will also find a table of example resource requests at the bottom of the page.

In most cases you can create a script file on your local machine and transfer it to HPC using `sftp` or `scp` and it should be fine. However that is not true for Windows files which may have hidden characters that produce unexpected behavior when run as a Linux script. Use the "dos2unix" command on a login node to convert them e.g.

```
$ dos2unix -n windowsfile.txt linuxfile.pbs
```

### Script Examples

Examples of PBS scripts can be found here: [https://ua-researchcomputing-hpc.github.io/PBS-Scripts/](https://ua-researchcomputing-hpc.github.io/PBS-Scripts/)

### Script Builder

Knowing what choices to make can be confusing so there is a tool available to help make the choices for you. It will limit you to valid selections and help to build a script that can be cut / pasted into your home directories.

This choice is for Ocelote. All of the Ocelote nodes are configured the same, except for the large memory node. Follow this link: [Script Builder](https://ua-researchcomputing-hpc.github.io/PBS-Scripts/).

### Scheduler Options

- Requesting Resources and Node Types (Standard, GPU, or High Memory)
- Time
- Group List
- Queues
- PVMEM
- Place
- Requesting Email
- Array Jobs
- Interactive Jobs

### Requesting Resources and Node Types (Standard, GPU, or High Memory)

The `select` statement is used to request the compute resources needed to run your job. This includes the number of nodes and cores you need as well as the memory. Additionally, the select statement is used to specify the type of node needed: GPU, High Memory, or Standard.

The basic `select` statement is:

```
#PBS -l select=x:ncpus=Y:mem=Zgb
```

- `X` = the number of nodes or units of the resources required
- `Y` = the number of cores (individual processors) required on each node
- `Z` = the amount of memory (in mb or gb) required on each node

### Standard Nodes

- If you use fewer than 28 cores, consider using a simple fraction of 28 like 7, 14 or 21. More jobs can be run if there are not random core counts chosen. 9 is also quite good since 9 x 3 = 27 with only one core left over.
- On Ocelote, you might ask, why not use more than 168GB RAM if the node has 192GB? We use 6GB per core as a round number memory resource. Multiplied by the core count of 28 gives 168GB. The difference will be used by the operating system for file handling to make your jobs more efficient.
For Ocelote, all of the standard nodes have 28 cores and 6GB per core. `pcmem=6gb` can be added to the line or left off, and it will default to 6gb. The following select statement would request one complete node:

```bash
#PBS -l select=1:ncpus=28:mem=168gb:pcmem=6gb
```

This is an example of a job that uses two nodes.

```bash
#PBS -l select=2:ncpus=28:mem=168gb
```

**ElGato**

On ElGato, all of the standard nodes have 16 cores and 4GB per core. The maximum available memory on a standard ElGato node is 62GB, leaving the difference for the operating system. The following select statement would request one complete node:

```bash
#PBS -l select=1:ncpus=16:mem=62gb:pcmem=4gb
```

**High Memory Node**

One high memory node exists on Ocelote with 2016 GB of available memory and 48 cores. It is designed for jobs that need a lot of memory but are not MPI-enabled. Your `select` statement will include `pcmem=42gb`. If you leave that off, the job will default to 6gb and go somewhere else. Maintain the ratio `ncpus:mem` as `1:42` in order to reserve the required amount of memory for your job. For example:

```bash
#PBS -l select=1:ncpus=30:mem=1260gb:pcmem=42gb
#PBS -l select=1:ncpus=10:mem=420gb:pcmem=42gb
```

**GPU Nodes**

For more information on the GPU nodes available see our page on [GPU Nodes](#).

**Ocelote**

On Ocelote there are 46 Nvidia P100 nodes available with 28 cores and 224 GB of memory. Users of these nodes can use either their standard allocation of cpu hours or windfall time. Jobs that do not need GPU's will not run on them, including windfall jobs that do not need the GPU. Each job will have exclusive access to the node to prevent contention between the CPU cores and the GPU. Each group is limited to ten GPU nodes concurrently.

Half of the nodes have CentOS 7 installed and the other half have CentOS 6.

This requests a CentOS 7 GPU node:

```bash
#PBS -l select=1:ncpus=28:mem=224gb:np100s=1:os7=True
```

or

```bash
#PBS -l select=1:ncpus=28:mem=224gb:ngpus=1:os7=True
```

This requests a CentOS 6 GPU node:

```bash
#PBS -l select=1:ncpus=28:mem=224gb:np100s=1
```

Ocelote has a single node with two GPUs that may be requested with:

```bash
#PBS -l select=1:ncpus=28:mem=224gb:np100s=2
```

Because there is only one on the system, wait times may be longer than for a single-GPU node.

**ElGato**

Unlike Ocelote, ElGato has cgroups enabled which allows for selecting a partial GPU node. Memory requests should be scaled by `ncpusx16gb`. To request a full GPU node:
#PBS -l select=1:ncpus=16:mem=250gb:ngpus=1:pcmem=16gb

To request a single node with two GPUs:

#PBS -l select=1:ncpus=16:mem=250gb:ngpus=2:pcmem=16gb

**Time**

Users must specify the amount of time they'd like to request for their job. Jobs exceeding the amount of time allotted to them will be terminated by the scheduler. Up to 240 hours may be requested by any individual job. For more information, see: Allocation and Limits

#PBS -l walltime=hhh:mm:ss

Optionally, users may also specify CPU time, or cput for their job. CPU time is calculated as time x ncpus. If this option is left unspecified, the job will default to walltime x ncpus.

#PBS -l cput=hhh:mm:ss

**Group List**

Users are required to specify the name of their research group when submitting a job so that their allocation can be used. To find the accounts you have access to, you may either check through the web portal or via a terminal session with the command va

#PBS -W group_list=<YOUR_GROUP>

**Queues**

Users are required to specify a job queue for their job. For more information on job queues, see: Allocation and Limits

#PBS -q <QUEUE>

**PVMEM**

'pvmem' is the maximum amount of virtual memory used by any single process in the job. By default PBS set pvmem=mem/ncpus which is equal memory per process for all processes. Some multi-threaded applications do not have all threads equal. Setting pvmem higher than the default memory per core (usually 6gb) will override the default.

#PBS -l pvmem=Xgb

**Place**

The place statement is used to specify how your job’s chunks are placed.
Place has the form:

```bash
#PBS -l place=arrangement:sharing
```

where:

- arrangement is one of free | pack | scatter
- sharing is one of excl | shared | exclhost

<table>
<thead>
<tr>
<th>Arrangement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>Place job on any node</td>
</tr>
<tr>
<td>pack</td>
<td>All chunks taken from one host</td>
</tr>
<tr>
<td>scatter</td>
<td>Only one chunk from any host</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sharing</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>excl</td>
<td>Only this job uses the node</td>
</tr>
<tr>
<td>shared</td>
<td>Other jobs can run on this node</td>
</tr>
<tr>
<td>exclhost</td>
<td>The entire host is allocated to this job</td>
</tr>
</tbody>
</table>

### Requesting Email

Requesting email notifications with arrays may inundate your inbox. For example, an array with 500 jobs may result in 1500 separate pieces of mail.

If desired, you can request mail to be sent from your job submission. The mail has usefulness for informing you of the resources your job consumed so you can modify resource requests the next time. Mail is specified in PBS with both of the following:

```bash
### Request mail at the beginning, end, and abnormal end of each job
#PBS -m bea
### The address where the mail will be sent. This must be your university account.
#PBS -M netid@email.arizona.edu
```

### Array Jobs

Arrays are a feature of PBS which allows you to submit a series of jobs using a single submission command described by a single submission script. A typical use of this is the need to batch process a large number of very similar jobs, which have similar input and output.

A job array is a single job with a list of sub-jobs. To submit an array job, use the `-J` flag to describe a range of sub-job indices. For example:

```bash
$ qsub -J 1-100 script.pbs
```

The typical submission script for a job array uses the index of each sub-job to define the task specific for each sub-job. E.g. the name of the input file or of the output directory. The sub-job index is given by the PBS variable `PBS_ARRAY_INDEX`. To illustrate its use, consider the application `myApp` processes some files named `input_*.dat` (taken as input), with * ranging from 1 to 100. This processing is described in a single submission script called `submit.sh`, which contains the following line:

```bash
input_\$PBS_ARRAY_INDEX.dat
```
A job array is submitted using this script, with the command `qsub -J 1-100 script.sh`. When a sub-job is executed, the file names in the line above are expanded using the sub-job index, with the result that each sub-job processes a unique input file and outputs the result to a unique output file.

**Interactive Jobs**

Most jobs are submitted to the batch scheduler PBS. You can also use PBS to submit an interactive job meaning that PBS will assign you a job that meets your requirements and put you at the command prompt. This allows you to run work/debug in realtime on a compute node. The directive that distinguishes an interactive job from a batch submission is the `-I` flag. You may submit an interactive job either using a script with the standard PBS directives, or you may submit your job directives directly from the command line. For example:

**On Ocelote:**

```
$ qsub -I -N jobname -W group_list=GROUP-NAME -q windfall -l select=1:ncpus=28:mem=168gb -l walltime=1:0:0
```

**On ElGato:**

```
$ qsub -I -N jobname -W group_list=GROUP-NAME -q windfall -l select=1:ncpus=16:mem=64gb -l walltime=1:0:0
```

**Monitor Jobs**

Once your job(s) have been submitted, you can check their status, output, or delete them:

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qpeek jobid</code></td>
<td>Check the output of a running job</td>
</tr>
<tr>
<td><code>nodes-busy</code></td>
<td>Summary of nodes running / not running jobs</td>
</tr>
<tr>
<td><code>qstat -q</code></td>
<td>List the queues and their run limits</td>
</tr>
<tr>
<td><code>qstat -Q</code></td>
<td>List the status of all the queues</td>
</tr>
<tr>
<td><code>qstat -a</code></td>
<td>List all the jobs on the system and their status</td>
</tr>
<tr>
<td><code>qstat -u &lt;netid&gt;</code></td>
<td>List status of your jobs</td>
</tr>
<tr>
<td><code>qstat -as</code></td>
<td>List all jobs with a brief explanation of their status and</td>
</tr>
<tr>
<td><code>qstat --wantl</code></td>
<td>Show status of all jobs and array elements</td>
</tr>
<tr>
<td><code>qstat -r</code></td>
<td>List all jobs that are running</td>
</tr>
<tr>
<td><code>qstat -i</code></td>
<td>List all jobs that are not running</td>
</tr>
<tr>
<td><code>qstat -f 123456.head1</code></td>
<td>Request detailed information about job 123456</td>
</tr>
<tr>
<td><code>qstat -n 123456.head1</code></td>
<td>Get a list of nodes allocated to job 123456</td>
</tr>
<tr>
<td><code>qstat -T</code></td>
<td>Shows the estimated start time for all jobs in the queue</td>
</tr>
<tr>
<td><code>qdel 123456.head1</code></td>
<td>Delete a job</td>
</tr>
<tr>
<td><code>qdel 123456[][].head1</code></td>
<td>Delete an array job - those are square brackets</td>
</tr>
<tr>
<td><code>qalter &lt;jobid&gt; -l walltime=&lt;time&gt;</code></td>
<td>Alter the walltime of a submitted job. <code>&lt;time&gt;</code> should be hhh:mm:ss and will replace the original walltime requested.</td>
</tr>
</tbody>
</table>
Running Multiple Jobs

*Jobs submissions should not be scripted in loops.* Instead, there are many ways to run multiple analyses with only a single job submission. Some options are detailed below.

Examples of implementing the methods detailed below can be found here: [https://ua-researchcomputing-hpc.github.io/PBS-Scripts/Array-and-Parallel/](https://ua-researchcomputing-hpc.github.io/PBS-Scripts/Array-and-Parallel/)

**Job Arrays**

A job array is a way to easily submit multiple jobs using the same PBS script with a single `qsub`. When an array job is submitted, it will create multiple sub-jobs, each using the same PBS directives and same commands as written in the script. The environment variable `PBS_ARRAY_INDEX` is a unique integer associated with each sub-job in the array and can be used to differentiate tasks.

Job arrays are much more efficient at submitting many tasks than running `qsub` on many individual submission scripts which helps keep the scheduler from being overloaded. Information on the PBS directives used to submit array jobs can be found under Scheduler Options -- Array Jobs.

**GNU Parallel**

GNU Parallel is a shell tool for executing commands in parallel using one or more computers. With GNU Parallel you can run multiple tasks within a single PBS job.

GNU Parallel is useful for running large number of computations. The number of running and queued PBS jobs is limited (see Allocation and Limits). GNU Parallel allows you to bypass those limits and run tens of thousands (and more) of computations.

Examples of running jobs with GNU Parallel can be found in the link at the beginning of this section.

**MPI Performance**

These are guidelines for improving performance of MPI jobs. They will not work in all cases. Google is your friend if you want to find out exactly what they do.

**Intel**

The Intel 2016 Compiler Suite is available and can be accessed via `module load <module name>` from the table below.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>intel/compiler</td>
<td>Adds Intel Compiler Suite (icc, ifort...) to your PATH</td>
</tr>
<tr>
<td>intel/daal</td>
<td>Adds Intel Data Analytics Acceleration Library paths to various environment variables</td>
</tr>
<tr>
<td>intel/gdb</td>
<td>Adds Intel’s variant of gdb called “gdb-ia” to your PATH</td>
</tr>
<tr>
<td>intel/itac</td>
<td>Adds the Intel Trace Analyzer and Collector tools to your PATH</td>
</tr>
<tr>
<td>intel/mkl</td>
<td>Adds the Intel Math Kernel library paths to various environment variables</td>
</tr>
<tr>
<td>intel/mpi</td>
<td>Adds the Intel MPI library path to various environment variables.</td>
</tr>
<tr>
<td>intel/tbb</td>
<td>Adds the Intel Thread Building Blocks paths to various environment variables</td>
</tr>
</tbody>
</table>
All tools default to the 64 bit version; some accept a "/32" suffix if you wish to work with binaries for older Intel architectures.

When you load an Intel compiler module, do NOT include the version. These get updated periodically and your module load will no longer work. module load intel/compiler is sufficient.

Intel compilers icc, ifort, mpicc, and mpifort are available with module load intel/compiler/64

The infiniband network and core placement can be tuned with Intel MPI to improve performance. Here are suggestions to potentially use in your job script:

```bash
export I_MPI_PIN_DOMAIN=auto
export I_MPI_PIN_PROCESSOR_LIST=0-27
export I_MPI_FABRICS=shm:dapl
export I_MPI_DAPL_DIRECT_COPY_THRESHOLD=65536
export I_MPI_DAPL_UD=enable
export I_MPI_DAPL_RDMA_MIXED=enable
export I_MPI_DYNAMIC_CONNECTION=0
export I_MPI_HYDRA_IFACE=ib0
export I_MPI_HYDRA_BOOTSTRAP=ssh
```

The above assumes you are using 28 cores per node; otherwise you would adjust the I_MPI_PIN_PROCESSOR_LIST variable appropriately.

### Other MPI Options

OpenMPI is a popular MPI implementation. Ocelote also has MPICH and MVAPICH

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>openmpi/gcc/1</td>
<td>(gcc only) Adds OpenMPI Version 1 paths to various environment variables</td>
</tr>
<tr>
<td>openmpi/gcc/2</td>
<td>(gcc only) Adds OpenMPI Version 2 paths to various environment variables</td>
</tr>
<tr>
<td>mpich/ge/gcc</td>
<td>(gcc only) Adds MPICH 3 paths to various environment variables</td>
</tr>
<tr>
<td>mpich/ge/intel</td>
<td>(icc only) Adds MPICH 3 paths to various environment variables</td>
</tr>
<tr>
<td>mvapich/gcc</td>
<td>(gcc only) Adds MVAPICH 1 paths to various environment variables</td>
</tr>
<tr>
<td>mvapich/intel</td>
<td>(icc only) Adds MVAPICH 1 paths to various environment variables</td>
</tr>
<tr>
<td>mvapich2/gcc</td>
<td>(gcc only) Adds MVAPICH 2 paths to various environment variables</td>
</tr>
<tr>
<td>mvapich2/intel</td>
<td>(icc only) Adds MVAPICH 2 paths to various environment variables</td>
</tr>
</tbody>
</table>

We suggest that when using OpenMPI, you define the following environment variables in your script:

```bash
export I_MPI_PIN_DOMAIN=auto
export I_MPI_PIN_PROCESSOR_LIST=0-27
export I_MPI_DAPL_DIRECT_COPY_THRESHOLD=65536
export I_MPI_DAPL_UD=enable
export I_MPI_DAPL_RDMA_MIXED=enable
export I_MPI_DYNAMIC_CONNECTION=0
export I_MPI_HYDRA_BOOTSTRAP=ssh
export OMPI_MCA_btl=self,openib
```

The last line is roughly equivalent to this as an alternative:

```bash
mpirun -np 256 --mca btl self,openib ./mpi_hello_world > output.txt
```