Slurm on Frontier

Nick Hagerty
HPC Engineer
System Acceptance & User Environment Group
Oak Ridge Leadership Computing Facility (OLCF)
Oak Ridge National Laboratory (ORNL)

August 23, 2023
# Slurm on Frontier and Crusher

Slurm provides 3 ways of submitting and launching jobs on Frontier/Crusher compute nodes: **batch scripts, interactive, and single-command**.

**NOTE:** Jobs launch on the allocated compute nodes, with the first compute node in the allocation serving as head-node.

---

## Slurm commands associated with the 3 methods:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sbatch</code></td>
<td>Used to submit a batch script to allocate a Slurm job allocation. The script contains options preceded with <code>#SBATCH</code>.</td>
</tr>
<tr>
<td><code>salloc</code></td>
<td>Used to allocate an interactive Slurm job allocation, where one or more job steps (i.e., <code>srun</code> commands) can then be launched on the allocated resources (i.e., nodes).</td>
</tr>
<tr>
<td><code>srun</code></td>
<td>Used to run a parallel job (job step) on the resources allocated with <code>sbatch</code> or <code>salloc</code>, or used to create a resource allocation and run a job step in a single command.</td>
</tr>
</tbody>
</table>
Batch Scripts

A **batch script** can be used to submit a job to run on the compute nodes at a later time. In this case, stdout and stderr will be written to a file(s) that can be opened after the job completes.

Here is an example of a simple batch script:

<table>
<thead>
<tr>
<th>Line</th>
<th>Actual batch script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>#!/bin/bash</code></td>
<td>[optional] shell interpreter line</td>
</tr>
<tr>
<td>2</td>
<td><code>#SBATCH -A &lt;project_id&gt;</code></td>
<td>OLCF project to charge</td>
</tr>
<tr>
<td>3</td>
<td><code>#SBATCH -J &lt;job_name&gt;</code></td>
<td>Job name</td>
</tr>
<tr>
<td>4</td>
<td><code>#SBATCH -o %x-%j.out</code></td>
<td>stdout file name (%x is job name, %j is job id)</td>
</tr>
<tr>
<td>5</td>
<td><code>#SBATCH -t 00:05:00</code></td>
<td>Walltime requested (HH:MM:SS)</td>
</tr>
<tr>
<td>6</td>
<td><code>#SBATCH -p &lt;partition&gt;</code></td>
<td>Batch queue</td>
</tr>
<tr>
<td>7</td>
<td><code>#SBATCH -N 2</code></td>
<td>Number of computed nodes requested</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Blank line</td>
</tr>
<tr>
<td>9</td>
<td><code>srun -N2 -n4 --ntasks-per-node=2 ./a.out</code></td>
<td>srun command to launch parallel job</td>
</tr>
</tbody>
</table>

The script would then be submitted to the queue using `sbatch <slurm_script>`

**You can also do `sbatch --wrap `<executable commands>`' to run a command without a job script**
Interactive Jobs

An **interactive job** allows multiple job steps (i.e., multiple `srun` commands) to be launched (interactively) on compute nodes allocated with the `salloc` command.

Here is an example of such a job:

```
$ salloc -A <project_id> -J <job_name> -t 00:05:00 -p <partition> -N 2
salloc: Granted job allocation 4258
salloc: Waiting for resource configuration
salloc: Nodes frontier[10469-10470] are ready for job

$ srun -n 4 --ntasks-per-node=2 ./a.out
<output printed to terminal>

$ srun -n 2 --ntasks-per-node=1 ./a.out
<output printed to terminal>
```

Here, `salloc` is used to request 2 compute nodes for 5 minutes, and once the compute nodes become available, job steps can be launched on them using `srun`. 
Single Command (non-interactive)

A **single command** job allows job allocation and a job step to be run in the same `srun` command.

Here is an example of such a job:

```
$ srun -A <project_id> -t 00:05:00 -p <partition> -N 2 -n 4 --ntasks-per-node=2 ./a.out
<output printed to terminal>
```

Here, `srun` is used to request 2 compute nodes for 5 minutes and launch a job step with 4 tasks (2 on each node) – all in the same single command.
## Common Slurm Options and Commands

### Common Slurm Submission Options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-A &lt;project_id&gt;</td>
<td>Project ID to charge</td>
</tr>
<tr>
<td>-J &lt;job_name&gt;</td>
<td>Name of job</td>
</tr>
<tr>
<td>-p &lt;partition&gt;</td>
<td>Partition / batch queue</td>
</tr>
<tr>
<td>-t &lt;time&gt;</td>
<td>Wall clock time <a href="">HH:MM:SS</a> (or you can just give minutes)</td>
</tr>
<tr>
<td>-N &lt;number_of_nodes&gt;</td>
<td>Number of compute nodes</td>
</tr>
<tr>
<td>-o &lt;file_name&gt;</td>
<td>Standard output file name</td>
</tr>
<tr>
<td>-e &lt;file_name&gt;</td>
<td>Standard error file name</td>
</tr>
<tr>
<td>--threads-per-core=&lt;threads&gt;</td>
<td>Number of active hardware threads per core [1 (default) or 2]</td>
</tr>
</tbody>
</table>

### Common Slurm Submission Options:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sinfo</td>
<td>Used to view partition and node information.</td>
</tr>
<tr>
<td>squeue</td>
<td>Used to view job and job step information for jobs in the scheduling queue.</td>
</tr>
<tr>
<td>sacct</td>
<td>Used to view accounting data for jobs and job steps in the Slurm database.</td>
</tr>
<tr>
<td>scancel</td>
<td>Used to signal or cancel jobs or job steps.</td>
</tr>
<tr>
<td>scontrol</td>
<td>Used to view or modify active job configuration.</td>
</tr>
</tbody>
</table>

*see man sbatch for more info*

*see individual man pages for more info*
Frontier Compute Node

512 GB (DDR4) (205 GB/s)

[Image of network diagram with labels and connections]

KEY:
- 64 GB HBM 1.6 TB/s
- GPU Mi250X
- 64 GB HBM 1.6 TB/s
- PCI Gen4 ESM (50+50 GB/s)
- Infinity Fabric GPU-GPU (50+50 GB/s)
- Infinity Fabric CPU-GPU (36+36 GB/s)
- PCIe Gen4 (8+8 GB/s)
- Ethernet (25+25 GB/s)
# Frontier Compute Node

## System Overview

### CPU-GPU Configuration

<table>
<thead>
<tr>
<th>Node</th>
<th>GPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>05</td>
<td>64 GB</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>64 GB</td>
</tr>
<tr>
<td>02</td>
<td>07</td>
<td>64 GB</td>
</tr>
<tr>
<td>03</td>
<td>08</td>
<td>64 GB</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>64 GB</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>64 GB</td>
</tr>
<tr>
<td>06</td>
<td>11</td>
<td>64 GB</td>
</tr>
<tr>
<td>07</td>
<td>12</td>
<td>64 GB</td>
</tr>
</tbody>
</table>

### GPU Connectivity

- **Infinity Fabric GPU-GPU**: (50+50 GB/s)
- **PCIe Gen4 ESM**: (50+50 GB/s)
- **Infinity Fabric CPU-GPU**: (36+36 GB/s)
- **PCIe Gen4**: (8+8 GB/s)
- **Ethernet**: (25+25 GB/s)
Low-noise mode is enabled on Frontier
- System processes are constrained to core 0 on every node.
- This means that core 0 will have lower performance than other cores on each node. To address this, core 0 should be isolated from user applications.

Then to provide a symmetric distribution of cores per L3 cache region (and so per GCD), the default behavior is set to isolate the first core from every L3 cache region from users. Orion utilizes these 8 reserved cores to improve I/O stability and performance.

**Default:** Slurm’s `--core-spec (-S)` option is set to 8.
- Set to 0 to make all cores available (at job allocation time).
Frontier Compute Node

Default NIC Binding: `MPICH_OFI_NIC_POLICY=NUMA`

For each MPI process, Cray MPI selects a NIC device that is closest to the CPU NUMA domain being used.

E.g., MPI ranks that land on cores 48-63 will be mapped to NIC 0 and should be mapped to GPUs 0 or 1 (as described previously)
Examples
Slurm GPU Bindings

(Recall: MPI ranks should target the GPU associated with their L3 cache region)

Always check process/thread/GPU bindings: [https://code.ornl.gov/olcf/hello_jobstep](https://code.ornl.gov/olcf/hello_jobstep)

- **GPU_ID** – node-level (or global) GPU ID read from `ROCR_VISIBLE_DEVICES`.
- **RT_GPU_ID** – HIP runtime GPU ID (as reported from `hipGetDevice`).
- **Bus_ID** – physical bus ID associated with the GPUs. Comparing bus IDs shows that different GPUs are being used.
hello_jobstep (https://code.ornl.gov/olcf/hello_jobstep)

Used to test process, thread, and GPU binding

Clone the repo and compile:
$ git clone https://code.ornl.gov/olcf/hello_jobstep.git
$ cd hello_jobstep
$ module load craype-accels-amd-gfx90a
$ make

Usage:

$ OMP_NUM_THREADS=1 srun -N 2 -n 16 -c 7 --gpus-per-task=1 --gpus-bind=closest ./hello_jobstep | sort

MPI 000 - OMP 000 - HWT 001 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 001 - OMP 000 - HWT 009 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 002 - OMP 000 - HWT 017 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 003 - OMP 000 - HWT 025 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 004 - OMP 000 - HWT 033 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 005 - OMP 000 - HWT 041 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 006 - OMP 000 - HWT 049 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 007 - OMP 000 - HWT 057 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 008 - OMP 000 - HWT 001 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 009 - OMP 000 - HWT 009 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 010 - OMP 000 - HWT 017 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 011 - OMP 000 - HWT 025 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 012 - OMP 000 - HWT 033 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 013 - OMP 000 - HWT 041 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 014 - OMP 000 - HWT 049 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 015 - OMP 000 - HWT 057 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
hello_jobstep (https://code.ornl.gov/olcf/hello_jobstep)

Used to test process, thread, and GPU binding

$ OMP_NUM_THREADS=1 srun --N 2 --n 16 --gpu-per-task=1 --gpu-bind=closest ./hello_jobstep | sort

MPI Rank ID

CPU hardware thread ID

HIP runtime GPU (from hipGetDevice)

Node name

node-level GPU ID (read from ROCR_VISIBLE_DEVICES)

Bus ID of GPU

Usage:

$ OMP_NUM_THREADS=1 srun --N 2 --n 16 --gpu-per-task=1 --gpu-bind=closest ./hello_jobstep | sort

MPI 000 - OMP 000 - HWT 001 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 001 - OMP 000 - HWT 009 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 002 - OMP 000 - HWT 017 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 003 - OMP 000 - HWT 025 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 004 - OMP 000 - HWT 033 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 005 - OMP 000 - HWT 041 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 006 - OMP 000 - HWT 049 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 007 - OMP 000 - HWT 057 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 008 - OMP 000 - HWT 001 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 009 - OMP 000 - HWT 009 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 010 - OMP 000 - HWT 017 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 011 - OMP 000 - HWT 025 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 012 - OMP 000 - HWT 033 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 013 - OMP 000 - HWT 041 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 014 - OMP 000 - HWT 049 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 015 - OMP 000 - HWT 057 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
$OMP_NUM_THREADS=1 

srun -N1 -n8 -c7 --gpus-per-node=8 ./hello_jobstep | sort

8 tasks per node, each with 7 CPU cores and 1 GPU

Incorrect mapping!

Bus_ID c1,c6,c9,ce,d1,d6,d9,de
Bus_ID c1,c6,c9,ce,d1,d6,d9,de
Bus_ID c1,c6,c9,ce,d1,d6,d9,de
Bus_ID c1,c6,c9,ce,d1,d6,d9,de
Bus_ID c1,c6,c9,ce,d1,d6,d9,de
Bus_ID c1,c6,c9,ce,d1,d6,d9,de
Bus_ID c1,c6,c9,ce,d1,d6,d9,de
Bus_ID c1,c6,c9,ce,d1,d6,d9,de

Open slide master to edit
$OMP_NUM_THREADS=1\ srun\ -N1\ -n8\ -c7\ --gpu-per-node=8\ --gpu-bind=closest\ ./hello_jobstep\ |sort

8 tasks per node, each with 7 CPU cores and 1 GPU
Incorrect mapping!

8 tasks per node, each with 7 CPU cores and 1 GPU

$OMP_NUM_THREADS=1 srun -N1 -n8 -c7 --gpu-per-task=1 ./hello_jobstep | sort
$OMP_NUM_THREADS=1 srun -N1 -n8 -c7 --gpus-per-task=1 --gpu-bind=closest ./hello_jobstep | sort

8 tasks per node, each with 7 CPU cores and 1 GPU
```
OMP_NUM_THREADS=1 srun -N1 -n1 -c7 --gpu-per-node=8 --gpu-bind=closest ./hello_jobstep | sort
```

```
MP1 000 - OMP 000 - HWT 003 - Node frontier08303 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
```

---

```bash
--gpu-bind={verbose,}<type>
```

Bind tasks to specific GPUs. By default every spawned task can access every GPU allocated to the step. If "verbose," is specified before `<type>`, then print out GPU binding debug information to the stderr of the tasks. **GPU binding is ignored if there is only one task.**

---

**Incorrect mapping!**

1 task per node, with 7 CPU cores and 1 GPU
$ OMP_NUM_THREADS=1 srun --gpu-per-task=1 --gpu-bind=closest ./hello_jobstep | sort

Bind tasks to specific GPUs. By default every spawned task can access every GPU allocated to the step. If "verbose," is specified before <type>, then print out GPU binding debug information to the stderr of the tasks. **GPU binding is ignored if there is only one task.**

1 task per node, with 7 CPU cores and 1 GPU

---

- **GPU binding**
  - By default, every spawned task can access every GPU allocated to the step.
  - If "verbose," is specified before <type>, print out GPU binding debug information to the stderr of the tasks.
  - Ignored if there is only one task.

---

**Diagram:**
- Node layout: 1 task per node, 7 CPU cores, 1 GPU
- GPU allocation:
  - Each node contains 7 CPU cores and 1 GPU
  - GPU allocation is distributed across the nodes
  - GPU binding is ignored in single-task scenarios.
$ OMP_NUM_THREADS=1 srun -N2 -n2 -c7 --gpus-per-node=8 --gpu-bind=closest ./hello_jobstep | sort

Bind tasks to specific GPUs. By default every spawned task can access every GPU allocated to the step. If "verbose," is specified before <type>, then print out GPU binding debug information to the stderr of the tasks. **GPU binding is ignored if there is only one task.**

**Incorrect mapping!**

1 task per node, with 7 CPU cores and 1 GPU (on 2 nodes)
Bind tasks to specific GPUs. By default every spawned task can access every GPU allocated to the step. If "verbose," is specified before <type>, then print out GPU binding debug information to the stderr of the tasks. **GPU binding is ignored if there is only one task.**

```
$ OMP_NUM_THREADS=1 srun -N2 -n2 -c7 --gpu-per-task=1 --gpu-bind=closest ./hello_jobstep | sort
```

1 task per node, with 7 CPU cores and 1 GPU (on 2 nodes)
## Distribution of tasks to node, sockets, and CPUs

| --distribution=<value>[::<value>][:<value>]>({Pack | NoPack}) | Specifies the distribution of MPI ranks across compute nodes, sockets (L3 regions on Crusher), and cores, respectively. The default values are block:cyclic:cyclic |
| --- | --- |
| * Can be used to refer to the default value |

```
-m, --distribution= *
|block|cyclic|arbitrary|plane=<size>|
|block|cyclic|fcyclic|block|cyclic|fcyclic||,
|Pack|NoPack|
```

This option controls the distribution of tasks to the nodes on which resources have been allocated, and the distribution of those resources to tasks for binding (task affinity).

The **first distribution method** (before the first ":") controls the distribution of tasks to nodes.

[default method for distributing tasks to nodes (block)]

The **second distribution method** (after the first ":") controls the distribution of allocated CPUs across sockets for binding to tasks.

[default method for distributing CPUs across sockets (cyclic)]
$ OMP_NUM_THREADS=1 srun -N2 -n16 -c7 --gpus-per-task=1 --gpu-bind=closest ./hello_jobstep | sort

MPI 000 - OMP 000 - HWT 001 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 001 - OMP 000 - HWT 001 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 002 - OMP 000 - HWT 017 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 003 - OMP 000 - HWT 025 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 004 - OMP 000 - HWT 033 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 005 - OMP 000 - HWT 041 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 006 - OMP 000 - HWT 049 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 007 - OMP 000 - HWT 057 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 008 - OMP 000 - HWT 001 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 009 - OMP 000 - HWT 009 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 010 - OMP 000 - HWT 017 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 011 - OMP 000 - HWT 025 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID c9
MPI 012 - OMP 000 - HWT 033 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 013 - OMP 000 - HWT 041 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 014 - OMP 000 - HWT 049 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 015 - OMP 000 - HWT 057 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6

$ OMP_NUM_THREADS=1 srun -N2 -n16 -c7 --gpus-per-task=1 --gpu-bind=closest -m cyclic ./hello_jobstep | sort

MPI 000 - OMP 000 - HWT 001 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 001 - OMP 000 - HWT 001 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 002 - OMP 000 - HWT 009 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 003 - OMP 000 - HWT 009 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 004 - OMP 000 - HWT 017 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 005 - OMP 000 - HWT 017 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 006 - OMP 000 - HWT 025 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 007 - OMP 000 - HWT 025 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 008 - OMP 000 - HWT 033 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 009 - OMP 000 - HWT 033 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 010 - OMP 000 - HWT 041 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 011 - OMP 000 - HWT 041 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 012 - OMP 000 - HWT 049 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 013 - OMP 000 - HWT 049 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 014 - OMP 000 - HWT 057 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 015 - OMP 000 - HWT 057 - Node frontier10228 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6

The first distribution method (before first ":") controls the distribution of tasks to nodes.

[default method for distributing tasks to nodes (block)]
second distribution method (after first ":") controls the distribution of allocated CPUs across sockets for binding to tasks.

[default method for distributing CPUs across sockets (cyclic)]

$ OMP_NUM_THREADS=1 srun -N1 -n16 -c3 --gpu-bind=closest --ntasks-per-gpu=2 ./hello_jobstep | sort

MPI 000 - OMP 000 - HWT 001 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 001 - OMP 000 - HWT 009 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 002 - OMP 000 - HWT 017 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 003 - OMP 000 - HWT 025 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 004 - OMP 000 - HWT 033 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 005 - OMP 000 - HWT 041 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 006 - OMP 000 - HWT 049 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 007 - OMP 000 - HWT 057 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 008 - OMP 000 - HWT 004 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 009 - OMP 000 - HWT 012 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 010 - OMP 000 - HWT 020 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 011 - OMP 000 - HWT 028 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID c9
MPI 012 - OMP 000 - HWT 036 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 013 - OMP 000 - HWT 044 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 014 - OMP 000 - HWT 052 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 015 - OMP 000 - HWT 060 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6

$ OMP_NUM_THREADS=1 srun -N1 -n16 -c3 --gpu-bind=closest --ntasks-per-gpu=2 -m block:block ./hello_jobstep | sort

MPI 000 - OMP 000 - HWT 001 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 001 - OMP 000 - HWT 004 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 002 - OMP 000 - HWT 007 - Node frontier10227 - RT_GPU_ID 0,1 - GPU_ID 4,5 - Bus_ID d1,d6
MPI 003 - OMP 000 - HWT 011 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 004 - OMP 000 - HWT 017 - Node frontier10227 - RT_GPU_ID 0,1 - GPU_ID 2,5 - Bus_ID c9,d6
MPI 005 - OMP 000 - HWT 018 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 006 - OMP 000 - HWT 021 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 007 - OMP 000 - HWT 025 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 008 - OMP 000 - HWT 028 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 009 - OMP 000 - HWT 033 - Node frontier10227 - RT_GPU_ID 0,1 - GPU_ID 3,6 - Bus_ID ce,d9
MPI 010 - OMP 000 - HWT 035 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 011 - OMP 000 - HWT 041 - Node frontier10227 - RT_GPU_ID 0,1 - GPU_ID 6,7 - Bus_ID d9,de
MPI 012 - OMP 000 - HWT 042 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 013 - OMP 000 - HWT 045 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 014 - OMP 000 - HWT 049 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 015 - OMP 000 - HWT 052 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
Another way to check for resources available to each process

Prepend stdout lines with process ID

$ srun -l <srun-options> /bin/bash -c 'echo $(hostname) $(grep Cpus_allowed_list /proc/self/status) GPUS: $ROCR_VISIBLE_DEVICES' | sort

$ srun -l -N1 -n8 -c7 --gpus-per-task=1 --gpu-bind=closest /bin/bash -c 'echo $(hostname) $(grep Cpus_allowed_list /proc/self/status) GPUS: $ROCR_VISIBLE_DEVICES' | sort

0: crusher109 Cpus_allowed_list: 1-7 GPUS: 4
1: crusher109 Cpus_allowed_list: 9-15 GPUS: 5
2: crusher109 Cpus_allowed_list: 17-23 GPUS: 2
3: crusher109 Cpus_allowed_list: 25-31 GPUS: 3
5: crusher109 Cpus_allowed_list: 41-47 GPUS: 7
6: crusher109 Cpus_allowed_list: 49-55 GPUS: 0
7: crusher109 Cpus_allowed_list: 57-63 GPUS: 1

Tells you which CPU cores and GPUs are available to each process, but not where they actually ran.
second distribution method (after first ":") controls the distribution of allocated CPUs across sockets for binding to tasks.

[default method for distributing CPUs across sockets (cyclic)]

$ OMP_NUM_THREADS=1  

srun -N1 -n16 -c3 --gpu-bind=closest --ntasks-per-gpu=2 -m block:block ./hello_jobstep | sort

Due to core isolation:

2 tasks (each with 3 cores) can fit on a node, but there is still 1 more core left over that will be given to the next task.

This can be resolved by either:

1. Setting core isolation to -S16 so there are only 6 cores available per L3
2. Removing the core isolation and using 4 threads per task.
$ OMP_NUM_THREADS=1

second distribution method (after first ":") controls the distribution of allocated CPUs across sockets for binding to tasks.

[default method for distributing CPUs across sockets (cyclic)]

$ OMP_NUM_THREADS=1 srun -N1 -n16 -c3 --gpu-bind=closest --ntasks-per-gpu=2 -m block:block /hello_jobstep | sort

$ OMP_NUM_THREADS=1 srun -l -N1 -n16 -c3 --gpu-bind=closest --ntasks-per-gpu=2 -m block:block /bin/bash -c 'echo $(hostname) $(grep | sort)'
Using all CPU cores on a Crusher/Frontier node

**NOT RECOMMENDED. Why?**
- All system processes are still bound to core 0, so core 0 will provide lower performance
- Orion is configured to use the 1\textsuperscript{st} core from each L3 cache to improve I/O performance

```
$ salloc -Astf016_frontier -N1 -t120 -S0
```

```
$ OMP_NUM_THREADS=8 srun -N1 -n8 -c8 --gpus-per-task=1 --gpu-bind=closest ./hello_jobstep | sort
```

Count of specialized cores per node reserved by the job for system operations and not used by the application.

Remember:
- core 0 will have lower performance than other cores
- Orion performance may be reduced
Multiple job steps on a single node

```bash
$ cat multiple_jobsteps.sh
#!/bin/bash

for idx in {1..8}; do
    # Requires cray-mpich version > 8.1.18
    OMP_NUM_THREADS=1 srun --exact -u -N1 -w <node_name> -n1 -c7 --gpus-per-task=1 --gpu-bind=closest .:/hello_jobstep &
    usleep 50000
done
wait

$ ./multiple_jobsteps.sh
Tue 14 Feb 2023 12:31:59 PM EST
MPI 000 - OMP 000 - HWT 051 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1

Tue 14 Feb 2023 12:32:00 PM EST
MPI 000 - OMP 000 - HWT 059 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6

Tue 14 Feb 2023 12:32:01 PM EST
MPI 000 - OMP 000 - HWT 019 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9

Tue 14 Feb 2023 12:32:02 PM EST
MPI 000 - OMP 000 - HWT 027 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce

Tue 14 Feb 2023 12:32:03 PM EST
MPI 000 - OMP 000 - HWT 003 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1

Tue 14 Feb 2023 12:32:04 PM EST
MPI 000 - OMP 000 - HWT 011 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6

Tue 14 Feb 2023 12:32:05 PM EST
MPI 000 - OMP 000 - HWT 035 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9

Tue 14 Feb 2023 12:32:06 PM EST
MPI 000 - OMP 000 - HWT 042 - Node frontier10227 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
```

Added a `sleep(20)` to the bottom of `hello_jobstep` so it didn’t return so quickly.

For concurrent multi-node steps, same process, but can only run 3 concurrent steps
Using `sbcast` to improve job initialization at scale
Running at scale

By default, every single rank reads from the file system independently:

Number of file system requests to load the binary only: # MPI Ranks

+ cost to link the application: #MPI Ranks x #shared libraries

Shared file systems: Orion/NFS
Running at scale

But, if we can pre-stage binary & libraries in node-local storage:

# MPI Ranks

**frontier00001**

- Rank 0
- Rank 1
- Rank 7

Node-local file system (NVME, /tmp)

**frontier00002**

- Rank 8
- Rank 9
- Rank 15

Node-local file system (NVME, /tmp)

Shared file systems: Orion/NFS
Running at scale

But, if we can pre-stage binary & libraries in node-local storage:

Benefits*:
- Decrease in application launch & link times
- Reduced load & dependence on shared file systems
- Consistent/predictable launch-time behavior
- Reduced occurrence of timeouts at launch time
- Early detection of failed run
## Running at scale - sbcast

### How to use `sbcast` to pre-stage binary & libraries:

<table>
<thead>
<tr>
<th>Line</th>
<th>Bash code in Job Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td># Must use <code>-C nvme</code> in SBATCH headers to use NVME drive</td>
</tr>
<tr>
<td>2</td>
<td>sbcast --send-libs --exclude=NONE -pf ${exe} /mnt/bb/$USER</td>
</tr>
<tr>
<td>3</td>
<td># if SBCAST fails, abort</td>
</tr>
<tr>
<td>4</td>
<td>if [ ! &quot;$?&quot; == 0 ]; then echo &quot;SBCAST failed&quot;; exit 1; fi</td>
</tr>
<tr>
<td>5</td>
<td># This does not work if you use RPATH’s</td>
</tr>
<tr>
<td>6</td>
<td>export LD_LIBRARY_PATH=&quot;/mnt/bb/$USER/${exe}_libs&quot;</td>
</tr>
<tr>
<td>7</td>
<td># libfabric dlopen’s several libraries. Add those paths to end of LD_LIBRARY_PATH:</td>
</tr>
<tr>
<td>8</td>
<td>export LD_LIBRARY_PATH=&quot;${LD_LIBRARY_PATH}:${(pkg-config --variable=libdir libfabric)}&quot;</td>
</tr>
<tr>
<td>9</td>
<td># Patch: libfabric dlopen’s <code>libhsa-runtime64.so</code>, but only <code>libhsa-runtime64.so.1</code> is sbcast</td>
</tr>
<tr>
<td>10</td>
<td># Add sym-link on each node for <code>libhsa-runtime64.so</code></td>
</tr>
<tr>
<td>11</td>
<td>srun -N ${SLURM_NNODES} -n ${SLURM_NNODES} --ntasks-per-node=1 --label \</td>
</tr>
<tr>
<td>12</td>
<td>-D /mnt/bb/$USER/${exe}_libs</td>
</tr>
<tr>
<td>13</td>
<td>if [ -f libhsa-runtime64.so.1 ]; then ln -s libhsa-runtime64.so.1 libhsa-runtime64.so; fi</td>
</tr>
</tbody>
</table>
Running at scale - sbcast

How to use `sbcast` to pre-stage binary & libraries:

<table>
<thead>
<tr>
<th>Line</th>
<th>Bash code in Job Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td># Must use ```-C nvme`` in SBATCH headers to use NVME drive</td>
</tr>
<tr>
<td>2</td>
<td>sbcast --send-libs --exclude=NONE -pf ${exe} /mnt/bb/$USER</td>
</tr>
<tr>
<td>3</td>
<td># if SBCAST fails, abort</td>
</tr>
<tr>
<td>4</td>
<td>if [ ! &quot;$?&quot; == 0 ]; then echo &quot;SBCAST failed&quot;; exit 1; fi</td>
</tr>
<tr>
<td>5</td>
<td>export LD_LIBRARY_PATH=&quot;/mnt/bb/$USER/${exe}_libs&quot;</td>
</tr>
<tr>
<td>6</td>
<td># libfabric dlopen's several libraries. Add those paths to end of LD_LIBRARY_PATH:</td>
</tr>
<tr>
<td>7</td>
<td>export LD_LIBRARY_PATH=&quot;${LD_LIBRARY_PATH}:${(pkg-config --variable=libdir libfabric)}&quot;</td>
</tr>
<tr>
<td>8</td>
<td># Patch: libfabric dlopen's <code>libhsa-runtime64.so</code>, but only <code>libhsa-runtime64.so.1</code> is sbcast</td>
</tr>
<tr>
<td>9</td>
<td># Add sym-link on each node for <code>libhsa-runtime64.so</code></td>
</tr>
<tr>
<td>10</td>
<td>srun -N ${SLURM_NNODES} -n ${SLURM_NNODES} --ntasks-per-node=1 --label \</td>
</tr>
<tr>
<td>11</td>
<td>-D /mnt/bb/$USER/${exe}_libs</td>
</tr>
<tr>
<td>12</td>
<td>if [ -f libhsa-runtime64.so.1 ]; then ln -s libhsa-runtime64.so.1 libhsa-runtime64.so; fi</td>
</tr>
</tbody>
</table>
Running at scale - sbcast

How to use `sbcast` to pre-stage binary & libraries:

<table>
<thead>
<tr>
<th>Line</th>
<th>Bash code in Job Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td># Must use <code>-C nvme</code> in SBATCH headers to</td>
</tr>
<tr>
<td>2</td>
<td>sbcast --send-libs --exclude=NONE -pf ${exe}</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td># if SBCAST fails, abort</td>
</tr>
<tr>
<td>5</td>
<td>if [ ! &quot;$?&quot; == 0 ]; then echo “SBCAST failed”; exit 1; fi</td>
</tr>
<tr>
<td>6</td>
<td>export LD_LIBRARY_PATH=&quot;/mnt/bb/$USER/${exe}_libs&quot;</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td># libfabric dlopen’s several libraries. Add those paths to end of LD_LIBRARY_PATH:</td>
</tr>
<tr>
<td>9</td>
<td>export LD_LIBRARY_PATH=&quot;${LD_LIBRARY_PATH}:$(pkg-config --variable=libdir libfabric)&quot;</td>
</tr>
<tr>
<td>10</td>
<td># Patch: libfabric dlopen’s <code>libhsa-runtime64.so</code>, but only <code>libhsa-runtime64.so.1</code> is sbcast</td>
</tr>
<tr>
<td>11</td>
<td># Add sym-link on each node for <code>libhsa-runtime64.so</code></td>
</tr>
<tr>
<td>12</td>
<td>srun -N ${SLURM_NNODES} -n ${SLURM_NNODES} --ntasks-per-node=1 --label \</td>
</tr>
<tr>
<td>13</td>
<td>-D /mnt/bb/$USER/${exe}_libs</td>
</tr>
<tr>
<td>14</td>
<td>if [ -f libhsa-runtime64.so.1 ]; then ln -s libhsa-runtime64.so.1 libhsa-runtime64.so; fi</td>
</tr>
</tbody>
</table>

SBCAST sends files chunk-by-chunk, so a failed SBCAST may leave partial libraries on some nodes.
Running at scale - sbcast

How to use `sbcast` to pre-stage binary & libraries:

<table>
<thead>
<tr>
<th>Line</th>
<th>Bash code in Job Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td># Must use <code>-C nvme</code> in SBATCH headers to use NVME drive</td>
</tr>
<tr>
<td>2</td>
<td>sbcast --send-libs --exclude=NONE -pf <code>${exe}_libs</code></td>
</tr>
<tr>
<td>3</td>
<td># if SBCAST fails, abort</td>
</tr>
<tr>
<td>4</td>
<td>if [ ! &quot;$?&quot; == 0 ]; then echo &quot;SBCAST failed&quot; fi</td>
</tr>
<tr>
<td>5</td>
<td>export LD_LIBRARY_PATH=&quot;/mnt/bb/USER/${exe}_libs&quot;</td>
</tr>
<tr>
<td>6</td>
<td># libfabric dlopen’s several libraries. Add those paths to end of LD_LIBRARY_PATH:</td>
</tr>
<tr>
<td>7</td>
<td>export LD_LIBRARY_PATH=&quot;${LD_LIBRARY_PATH}:${(pkg-config --variable=libdir libfabric)}&quot;</td>
</tr>
<tr>
<td>8</td>
<td># Patch: libfabric dlopen’s <code>libhsa-runtime64.so</code>, but only <code>libhsa-runtime64.so.1</code> is sbcast</td>
</tr>
<tr>
<td>9</td>
<td># Add sym-link on each node for <code>libhsa-runtime64.so</code></td>
</tr>
</tbody>
</table>
| 10   | srun -N ${SLURM_NNODES} -n ${SLURM_NNODES} --ntasks-per-node=1 --label \
| 11   | -D /mnt/bb/$USER/${exe}_libs \
| 12   | if [ -f libhsa-runtime64.so.1 ]; then ln -s libhsa-runtime64.so.1 libhsa-runtime64.so; fi |

All libraries are placed in an `${exe}_libs` directory in the directory the executable is in.
Running at scale - sbcast

How to use `sbcast` to pre-stage binary & libraries:

<table>
<thead>
<tr>
<th>Line</th>
<th>Bash code in Job Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td># Must use <code>'-C nvme'</code> in SBATCH headers to use NVME drive</td>
</tr>
<tr>
<td>2</td>
<td>sbcast --send-libs --exclude=NONE -pf ${exe} /mnt/bb/$USER</td>
</tr>
<tr>
<td>3</td>
<td># if SBCAST fails, abort</td>
</tr>
<tr>
<td>4</td>
<td>if [ ! &quot;$?&quot; == 0 ]; then echo &quot;SBCAST failed&quot;</td>
</tr>
<tr>
<td>5</td>
<td>export LD_LIBRARY_PATH=&quot;/mnt/bb/$USER/${exe}_libs&quot;</td>
</tr>
<tr>
<td>6</td>
<td># libfabric dlopen's several libraries. Add those paths to end of LD_LIBRARY_PATH:</td>
</tr>
<tr>
<td>7</td>
<td>export LD_LIBRARY_PATH=&quot;${LD_LIBRARY_PATH}:${(pkg-config --variable=libdir libfabric)}&quot;</td>
</tr>
<tr>
<td>8</td>
<td># Patch: libfabric dlopen's <code>libhsa-runtime64.so</code>, but only <code>libhsa-runtime64.so.1</code> is sbcast</td>
</tr>
<tr>
<td>9</td>
<td># Add sym-link on each node for <code>libhsa-runtime64.so</code></td>
</tr>
<tr>
<td>10</td>
<td>srun -N $SLURM_NNODES -n $SLURM_NNODES --ntasks-per-node=1 --label -D /mnt/bb/$USER/${exe}_libs</td>
</tr>
<tr>
<td>11</td>
<td>if [ -f libhsa-runtime64.so.1 ]; then ln -s libhsa-runtime64.so.1 libhsa-runtime64.so; fi</td>
</tr>
</tbody>
</table>

sbcast cannot detect `dlopen` calls, so a few patches are required:
Running at scale - sbcast

Caveats:

• As mentioned, `dlopen` calls are not detected by `sbcast`. This requires additional effort from the user to place `dlopen`'d libraries correctly

• **Any RPATH, RUNPATH paths will still be used**
  • Spack often adds library paths to RPATH
  • make and cmake sometimes do this, check your binary to confirm (see: `readelf`)

• libfabric is not the only libraries that require patches
  • ROCBLAS library requires ROCBLAS_TENSILE_LIBPATH to be set
  • **Please submit a ticket to OLCF Help Desk to receive the most updated list of patches required for your specific application. Include any CrayPE and ROCm libraries that your application links (ie, rocBLAS).**
The benefit of sbcast

- Reduced occurrence of timeouts at launch time
- Early detection of failed initialization
- Consistent/predictable launch behavior
- Lower launch times
- Reduced reliance on Orion/NFS latency
Job Accounting
2 available sources of information: `scontrol` and `sacct`

- **scontrol** shows information about a pending, running, or recently completed (within 10 minutes) job
- **sacct** queries the Slurm database to show detailed information about each step in a job

```bash
> scontrol show job 1413140
JobId=1413140 JobName=lammps_cpu_nodecheck
  UserId=efaccept(15549) GroupId=efaccept(27243) MCS_label=N/A
  Priority=3447 Nice=0 Account=stf016_frontier QOS=normal
  JobState=PENDING Reason=Priority Dependency=(null)
  Requeue=0 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
```

```bash
> sacct -j 1234567
JobID  JobName  Partition  Account  AllocCPUS  State  ExitCode
---------  ---------  ----------  --------  -----------  ------  --------
1234567  orion_85_+  batch  stf002  126  COMPLETED  0:0
1234567.bat+  batch  stf002  63  COMPLETED  0:0
1234567.ext+  extern  stf002  126  COMPLETED  0:0
1234567.0  lcio  stf002  63  CANCELLED  0:15
```

**NOTE:** `sacct` data is not guaranteed to be accurate until a job completes. Running `sacct` during a job is not recommended.
Slurm Tips – job analysis slide 2/3

To show completed jobs in a specific time period, specify a start (-S) and end (-E) time
• The default time window depends on Slurm configuration (see man sacct)

$ sacct --user=tpapathe -S 2022-05-05T00:01 -E 2022-05-05T23:59 -X

<table>
<thead>
<tr>
<th>JobID</th>
<th>JobName</th>
<th>Partition</th>
<th>Account</th>
<th>AllocCPUS</th>
<th>State</th>
<th>ExitCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>107814</td>
<td>interacti+</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107836</td>
<td>interacti+</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107849</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107858</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>256</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107866</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>256</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107867</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>256</td>
<td>CANCELLED+</td>
<td>0:0</td>
</tr>
<tr>
<td>107868</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>256</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107965</td>
<td>MatrixTra+</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107966</td>
<td>rocprof</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>FAILED</td>
<td>6:0</td>
</tr>
<tr>
<td>107969</td>
<td>rocprof</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>FAILED</td>
<td>6:0</td>
</tr>
</tbody>
</table>

Then you can drill into more details about each job with the -j flag and customized output.
Slurm Tips – job analysis slide 3/3

`sacct --env-vars` shows ... all environment variables set at the start of a specific Slurm job.
`sacct --batch-script` ... shows the batch script run by a specific Slurm job.

Slurm provides 100+ fields in the `sacct` command.

```bash
$ sacct --helpformat
Account AdminComment AllocCPUS AllocNodes
AllocTRES AssocID AveCPU AveCPUFreq
AveDiskRead AveDiskWrite AvePages AveRSS
AveVMSize BlockID Cluster Comment
Constraints ConsumedEnergy ConsumedEnergyRaw Container
CPUSTime CPUSTimeRAW DBIndex DerivedExitCode

$ sacct -j 123456 --o jobid,jobname,account,allocnodes
JobID JobName account AllocNodes
--------- ---------- --------- -----------------------
1234567 orion_85+ stf002 1
1234567.bat+ batch stf002 1
1234567.ext+ extern stf002 1
1234567.0 lcio stf002 1
```
Getting Help
Helpful debugging flags

Useful Slurm flags:

• `-u` flag gives unbuffered output
• `-l` flag prepends the task ID to lines of stdout

Shell commands to run immediately before a job step:

```bash
ldd ${exe} &> ldd.log – verifies that your program selects the intended libraries
module -t list &> modules.log – prints list of loaded modules
printenv &> env.log – prints all environment variables
```
Submitting Helpful User Support Tickets

Typical questions we ask users for:

- Job ID
- List of modules loaded when compiling and running
- Batch script used to launch the job
- Job stdout and stderr
- Output from `ldd` run on executable

Notice these are all things that are included from the previous slide 😊

Other helpful things to include in your tickets

- Full errors that are generated
- Has the program run successfully before?
  - If so, did you change anything since then?
- Programming models used in your code

Submit support tickets to help@olcf.ornl.gov