The University of Queensland

Research Computing Centre

Wiener User Guide

General Information

The name?

Wiener is named in honor of Norbert Wiener, a mathematician who devised an algorithm to remove noise from a signal or image. The technique became known as "Deconvolution". The alternative name for this supercomputer is the "UQ Deconvolution Facility".

Who built Wiener?

- Wiener was the first supercomputer at UQ to be built "collaboratively" by a consortia of the RCC and the research-intensive institutes. The architect of Wiener was Jake Carroll (IMB, QBI, AIBN). The systems engineering and administration behind Wiener was facilitated by Irek Porebski (QBI). The governance and steering group for Wiener was provided by David Abramson (RCC) and Jake Carroll. The original procurement committee for Wiener consisted of seven people, representing the RCC, QBI, IMB and CMM in both operational and scientific interests.

What is it made of?

- Wiener is different from other UQ supercomputing facilities in that it is not a "turn key" delivered platform. Components were provided, hand-selected by the consortium after months of research, development, international field recon and a benchmarking regime, then the system was integrated and installed piece by piece to achieve the desired outcome.

- Wiener is a new breed of deployment at UQ as it uses the OpenHPC platform packages for its distribution and Warewulf for cluster-centric and deployment operations. These tools are increasingly prevalent around the world for new, leadership-class supercomputing facilities. The Wiener compute nodes consist of Dell R740/C4140 series servers, using Intel Xeon Skylake CPU's, nVidia Volta V100 GPU's and Mellanox 100G EDR IB HCA's for storage and compute workload interconnectivity.

Where did it come from?

- Wiener was strategically funded from office of UQ DVC-R via the RCC and a consortium of the university's various cutting-edge microscopy facilities housed within the Centre for Microscopy and Microanalysis (CMM), Institute for Molecular Bioscience (IMB), and Queensland Brain Institute (QBI).

What was it built for?

- Wiener's primary use case (and what it is/was initially built for) is the acceleration of very high throughput imaging instrumentation deskew, deconvolution and manipulation coming from the various centres and institutes, listed above.

Who can use the cluster?

- Wiener has been purchased to fulfil a specific role within the RCC's Advanced Computing Strategy.

- Technically, any researcher within UQ, who has a workload that is a good match for the cluster's capabilities should use Wiener.
  - Wiener has some additional specific requirements which users must demonstrate that they can meet, before access will be granted.
  - It is not a system to run generic CPU workloads on.
  - It is for accelerator-class codes **exclusively**.
  - To be clear: If you "just want to run some CPU jobs" but you are not involved with GPU or accelerator-class supercomputing, nor the imaging sciences, nor deep learning research, this platform is probably not for you. It would be advantageous to explore other resources outlined in the table below.

- If you are unsure about whether or not Wiener is right for you, please do not hesitate to contact rcc-support@uq.edu.au.

- If you are certain about the nature of your computation, accelerated computing and that your use-case fits the Wiener requirements, please contact helpdesk@qbi.uq.edu.au to discuss an account.

- This cluster is an integral part of the RCC Advanced Computing Strategy for UQ.
  The currently available resources, and their primary purpose, are detailed in the following table.

Resource	Primary Purpose
Awoonga	A high throughput computing cluster for loosely coupled, small footprint jobs and job arrays (i.e. single-node or sub-node scale)
FlashLite	A specialized HPC cluster for data-intensive computation (e.g. very large RAM and/or extreme input/output requirements).
Tinaroo	A high performance computing cluster for tightly coupled message passing jobs (i.e. multi-node message passing).
Wiener	A HPC platform targeting image de-convolution using graphical processing units (GPUs).

  This research computing loadout provides a competitive local capability, for UQ, that spans a wide range of requirements.

- Usage accounting is being to used to allow for reporting of usage. This leverages a suite of tools known as Open XDMoD for complete granularity of reporting and graphical day by day analysis of utilisation.

General access to Wiener

Currently, valid users who have been granted access should connect to the cluster by using one of the following methods:

- A command line SSH client to the host wiener.hpc.dc.uq.edu.au, in the form: `ssh -Y yourUqUserName@wiener.hpc.dc.uq.edu.au`
- A SFTP/SCP file transfer tool (eg. linux command line scp, WinSCP, FileZilla, CyberDuck)

Your UQ username and password should always work to access Wiener. These are the "UQ Single Sign On" credentials you are provided by the University Central IT services group.

When you refresh your UQ password, that password will immediately become your password for accessing Tinaroo. The use of SSH public/private key pairs is a convenience, especially when combined with a key agent utility.

Some care must be used and best practices for use of SSH keys are described elsewhere. The most basic level of this is:

Once logged into Wiener,

1. mkdir .ssh
2. `chmod 700 .ssh`
3. cd .ssh
4. `touch authorized_keys`
5. `chmod 600 authorized_keys`
6. Edit your authorized_keys file to add your id_rsa.pub key to this file (paste cleanly!)
7. `cd ..`

If you are using a linux workstation to access Wiener, then this configuration can be achieved by using the command `ssh-copy-id wiener.hpc.dc.uq.edu.au` on the workstation.

More details the command line tools, and file transfer tools, are provided in separate user guides:

- Connecting to HPC
- File Transfers

MFA on Wiener.

The Wiener Supercomputer is an **MFA-enabled system**. This means it uses UQ's Duo multi-factor authentication system. This is the **only** access method provided. Plain usernames, passwords, and keys without MFA have been deprecated.

During the login process to Wiener when using your username and password or Public Key, you will be asked to confirm the authentication using MFA DUO. Depending upon your access method - if you will be using an SSH terminal, Windows SCP, a Web interface of FastX or CVL, it may look slightly different. You will have the option to put the DUO code or enter 1 to send the push verification to your mobile device that you have linked to your MFA sign-up process.

Wiener nodes

Please visit the RCC website for the technical specifications of Wiener nodes.

Login node

Wiener has **one** login node. This is a direct login node. The address is wiener.hpc.dc.uq.edu.au. When you login for the first time, you will be in your own auto-generated home directory. The path is:

```
/clusterdata/uqUserName
```

This is a sub-component of a parallel file system that will be discussed later in the document.

The Wiener login node is accessible from around the UQ network, on campus, or, via UQ VPN if you are off campus. It does not have an external facing IP address and is not visible to the wider internet as it is specifically and explicitly a UQ only resource

To thwart password cracking attempts, a security tool will block repeated connection attempts to Wiener above a threshold. Each authentication attempt is monitored and administrators have heuristics to see in real time when attempts against accounts are being made.

We ask that users be mindful of the nature of a login node. Do not run computational workloads on the login nodes, please. Compiling, yes. Assembly of your code, yes. Editing your scripts and moving things around, yes. Submitting your jobs via "sbatch", yes. **Running "jobs" on the login node and not using the SLURM scheduler...no.**

Visualisation Nodes

There are two visualisation nodes.

Compute Nodes

Please visit the RCC website for the compute node for a list of technical specifications.

Normal CPU related matters

Although a Wiener compute node has 384GB of RAM, your job needs to leave some memory for the operating system. We recommend that you allow at least 2GB. The SLURM scheduler automatically stops a user-job over-allocating, in many cases.

Each node contains 2 physical skylake Intel Xeon 6132 series CPU sockets, containing a total of 28 hardware cores, or 56 threads of (HT) execution. SLURM will let you address as threads or has cores, depending upon the nature of your requests and workloads. There are numerous cases for both hardware core scheduling and software hyper-threading. **Generally speaking, we recommend addressing hardware cores only**.

If you are using pmem, pmix2 or some of the more advanced MPI memory placement modules, then a resource request of ,pmem=5GB will correspond to 140GB if you use all 28 cores (i.e. :ppn=28). **The default user allocation of memory quote on Wiener is 300GB.**

The RCC cluster security model allows you to SSH to compute nodes directly, but only in situations where you have a batch job running there. This prevents accidental harm to valid batch jobs running on the compute nodes. SLURM will enforce this to an instant, with your access to a compute node terminating the instant your job finishes.

Accelerated/GPU related matters

Wiener is UQ's first accelerated "from the ground up" supercomputer. Wiener has been built (so far) in two phases:

Wiener phase 1.0 hardware contains:

2 * nVidia Volta V100 PCI-E connected GPU's per node. Each of these GPU's contains:

- 5120 CUDA cores
- 640 Tensorflow hardware cores
- 16GB of HBM2 class memory.

Wiener phase 2.0 hardware contains:

4 * nVidia Volta V100 SXM2 connected GPU's per node. Each of these GPU's contains:

- 5120 CUDA cores
- 640 Tensorflow hardware cores
- 32GB of HBM2 class memory.

Thus, each node contains either 10240 or 20480 CUDA cores, 1280 or 2560 TensorFlow cores and 32GB or 128GB of HBM2 class memory. The PCI-E root complex of each node is configured such that there is a direct path across a PCI-E switch between both GPU's such that each GPU can signal at low latency should your accelerator code be sufficiently advanced and able to distribute workload between the GPU's appropriately. The phase 2.0 nodes do away with PCI-E switching entirely and use a technology known as SXM2 sockets and nVlink 2.0 to participate in GPU Peer-to-Peer communication for higher performance (900GB/sec) between GPU sockets for thread/process and memory sharing. This discussion is beyond the scope of this document however and can be dealt with by consultation with RCC-support.

The Storage subsystem - how to use it, which to use, performance information.

Wiener has four different storage capabilities, each with a different use case and function. Wiener does not share the same filesystem that is shared on Tinaroo, Awoonga and FlashLite, but it **does** contain a filesystem (MeDiCI) that allows a user to transport data to these other supercomputers.

You home directory - `/clusterdata/uqsomething`

- This is your "landing zone" and where your home directory resides.
- This is a filesystem backed by a Hitachi G200 series disk array. It contains ~120 spindles of NL-SAS 7200 RPM disk and is only to be used as a device to "land in" and keep basic things like your .config's or scripts. Common use cases for your home directory include a place where you have scripts, your source code, and textual outputs, such as log errors or log outputs.
  * It is **not backed up**, nor will it ever be.
- There is a 5GB quota on your home directory. This might seem a little odd and painful - but it is very intentional. It is designed to prevent the accidental use and over-run of a home directory.
- It is capable of writing and reading at around 4GB/sec.
- It is connected over Infiniband using NFSoRDMA (Network File System over Remote Direct Memory Access) using NFSv4.
- Please, keep it clean and don't "park" data here.
- This filesystem is mounted on every node in Wiener. All nodes in Wiener can see this filesystem, by design.
- The NFS protocol is used for communication back and forth in this filesystem over RDMA. It is a native infiniband connected filesystem and as such is capable of very high throughput with very low latency and minimal overhead to CPU or memory subsystem of each node.

The scratch storage filesystem `/scratch/ou_designation`

- This is your traditional scratch space. It is a filesystem **to compute against**.
- This is a filesystem backed by an IBM SpectrumScale GH14S. It contains 384 spindles of NL-SAS 7200 RPM disk. It is used for all computational workloads.
  * It is **not backed up**, nor will it ever be.
- There is a 2TB quota on your scratch area
- It is capable of writing and reading at around 35GB/sec, aggregate read/write IO of ~62GB/sec and 1.1 million input output operations per second.
- Please, keep it clean and don't "park" data here. It is a transient space. Data will be deleted from here after 6 months, in an **automated fashion**.
- This scratch filesystem is a "parallel filesystem" and uses GPFS as the fllesystem technology.
- This filesystem is mounted on every node in Wiener. All nodes in Wiener can see this filesystem, by design.
- The NSD protocol is used for communication back and forth in this filesystem over RDMA. It is a native infiniband connected filesystem and as such is capable of very high throughput with very low latency and minimal overhead to CPU or memory subsystem of each node.

The specialised performance filesystem `/nvmescratch/uqsomething`

- This is special scratch space. It is a filesystem **to compute against**. It is for highly specialised workloads that require the most unusual workload profiles that are as yet only classed as "emerging".
- This is a filesystem backed by a ring of 1.6TB NVME Samsung P1725a flash modules running in an RDMA parallel ring. In entirety it is around 30TB.
  * It is **not backed up**, nor will it ever be.
- There is no quota on this filesystem. It is highly experimental and expected that the users of this filesystem are sufficiently advanced, understand capacity limits and good storage etiquette.
- It is capable of writing and reading at around 180GB/sec and 30,000,000 IOPS.
- Please, keep it clean and don't "park" data here. It is a transient space. Data will be deleted from here after 6 months, in an **automated fashion**.
- This scratch filesystem is a "parallel filesystem" and uses BeeGFS as the fllesystem technology.
- This filesystem is mounted on every node in Wiener. All nodes in Wiener can see this filesystem, by design.
- The BeeGFS protocol is used for communication back and forth in this filesystem over RDMA. It is a native infiniband connected filesystem and as such is capable of extreme throughput with very low latency and minimal overhead to CPU or memory subsystem of each node.
- Careful note: If you don't know why you would need this particular filesystem and you are not aware of the nature of your IO workload, IOskew, IOPS profile or throughput profile, this filesystem should probably not be considered for your work. It is highly specialised, experimental and made for "bleeding edge" compute requirements. Its use should be discussed in consultation with the RCC

The MeDiCI fabric filesets `/afm01/Q0/yourCollectionHere`

- MeDiCI is the topic of an entire knowledge-base article in itself, of which, in part, can be found here:
- In short, MeDiCI is a high speed, parallel filesystem data storage fabric that transports data from instruments to supercomputers using the concept of caching to put the data where it is needed, when it is needed, deterministically.
- It means you can have your instrument generate data, dumped into MeDiCI, then, the namespace allows for the data to be instantly visible on the supercomputer you'd like to process on. This has significant implications for workflow efficiency.
- There is no quota on this filesystem.

- This an archival and parallel filesystem and uses GPFS-AFM as the fllesystem technology.
- This filesystem is mounted on every node in Wiener. All nodes in Wiener can see this filesystem, by design.
- The NSD protocol is used for communication back and forth in this filesystem over RDMA. It is a native infiniband connected filesystem and as such is capable of very fast throughput with very low latency and minimal overhead to CPU or memory subsystem of each node.
- If you do not have a MeDiCI collection but would like to be involved or need one to shunt instrument data or large data sets between your source and the UQ RCC maintained supercomputers, please log an RDM record creation request via the UQ RDM website. Consult the RDM help and the RCC Storage User Guide for tips on successfully completing the RDM record creation request and incorporating Medici storage into your computational workflow.
- Once you have a valid RDM MeDiCI collection
  - It will automatically become accessible (via the autofs mechanism) onto Awoonga/FlashLite/Tinaroo HPC systems.
  - It needs to be mounted onto Wiener by the systems team. You request that by submitting an email to helpdesk@qbi.uq.edu.au

Storage Best Practices

- If you've got many small files, transporting them and archiving them in a large zip or compressed package is best practice (.tar.gz, .tar.bz2 etc). Wiener has the Parallel Bzip2 packages installed, so a user can use many cores to compress and decompress an archive, where required.
- Once you've got your zip file transported to Wiener, you can unzip within scratch or within /nvmescratch, cleaning up when finished.
- Further information about storage is provided in storage user guide (via rcc.uq.edu.au)
- Different storage systems have different latency and IO characteristics. It is important to learn about your job and how it work in order to select the right scratch area to do your work upon.
- The SLURM scheduler can provide hints about IO for you.

A suggested storage-centric workflow

There are many ways to work with a supercomputer. Depending upon your workload and workflow, you will approach the nature of storage IO (input/output), processing and what you do with the resultant outputs in a myriad of different ways. Below is a suggested (one of many possible) way of managing your storage workflow:

1. Use your /clusterdata/uqsomething for small nonvolatile items such as documentation, source code, SBATCH files and the like. Users often compile here, too.
2. Use /afm01/scratch/uqsomething for job outputs and actual read/write heavy computational IO before copying them back to a MeDiCI collection for long term storage (/afm01/Q0xxx).
3. Use /nvmescratch only if you are 100% confident you are IO bound by the traditional GPFS filesystems. Unless you understand how to measure your IOwait state, latency tick timers and disk seek timing semantics, this is something that the RCC can help with.
4. When transferring into MeDiCI after you've finished a computational run, zip or compress first, then move it or copy it into MeDiCI. This is more efficient for MeDiCI and more efficient for you, later on, if you need it back quickly.
5. Unpack the compressed input data onto /afm01/scratch/uqsomething. Not /afm01/Q0xxx

The SLURM scheduler - what it is, how to use it.

Wiener is (once again) a little bit different than the previous UQ supercomputers, as a newer, more feature-rich scheduler was required for very granular access to accelerator (GPU) resources. PBS, TORQUE, SGE et al. did not offer these capabilities, so the newer generation of schedulers (8th generation schedulers, academically) were selected instead. Arriving at SLURM, this is now the industry standard for complicated, accelerated supercomputing workloads. SLURM is not perfect, and there are aspects of PBS that are far more intuitive, but for exclusively accelerator-class supercomputing, SLURM is an ideal fit and community support is growing, by the day.

A full guide is being worked on for SLURM, locally, which can be found here: [[http://www2.rcc.uq.edu.au/hpc/guides/index.html?secure/Batch_SLURM.html]], but at this stage, it is not finalised.

For time being, we recommend familiarising yourself with the SLURM syntax on the SchedMD website in the above links.

A basic SLURM submission script might look like this:

```
[uqjcarr1@wiener ~]$ cat tensorflow_mpi_run.sh

#!/bin/bash
#SBATCH -N 3
#SBATCH --job-name=jake_test_tensor_gpu
```

```
#SBATCH -n 3
#SBATCH -c 1
#SBATCH --mem=50000
#SBATCH -o tensor_out.txt
#SBATCH -e tensor_error.txt
#SBATCH --partition=gpu
#SBATCH --gres=gpu:tesla:2

module load gnu7/7.2.0
module load cuda/9.2.148.1
module load anaconda/3.6
module load mvapich2
module load pmix/1.2.3

srun -n2 --mpi=pmi2 python3.6 benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py --num_gpus=2 --model resnet50
--batch_size 128
```

To run this job, one would run the following, from the command line:

```
sbatch tensorflow_mpi_run.sh
```

To explain what this script is doing, bit by bit:

1. We must use the basic shell hash-bang pragma, to tell the shell interpreter we are using bash.
2. Next, we allocate three nodes.
3. We name our job, so it is identifiable.
4. We say how many tasks we'd like per node.
5. We say how many cores we'd like per task.
6. We ask for our memory per core.
7. We assign some file names for our outputs and error logs.
8. We ask for the GPU partition.
9. We ask for two GPU's per node, using the GRES directives (special to SLURM).
10. Then we use the **modules** system to load gnu basic utilities, CUDA, so we've got access to the nvidia GPU Accelerator libraries, anaconda, so we've got access to the Wiener-compiled TensorFlow accelerated libraries, mvapich2, for our MPI libraries, pmix, for MPI specific memory management.
11. Finally, we "srun" which is similar to the older style "mpirun", but in SLURM-nomenclature, this replaces it.

The job can then be monitored by looking for your outputs in "squeue" as follows:

```
[uqjcarr1@wiener ~]$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
2581 gpu jake_test_tensor_gpu uqjcarr1 R 15:22 1 gpunode-0-0
2579 gpu jake_test_tensor_gpu uqjcarr1 R 3:20:12 1 gpunode-0-2
2580 gpu jake_test_tensor_gpu uqjcarr1 R 4:07:03 1 gpunode-0-2
2578 gpu jake_test_tensor_gpu uqjcarr1 R 6:20:21 1 gpunode-0-3
2582 gpu jake_test_tensor_gpu uqjcarr1 R 1:36:00 1 gpunode-0-0
2583 gpu jake_test_tensor_gpu uqjcarr1 R 14:22 1 gpunode-0-1
```

You can dig further into the jobs to learn more about their execution via:

```
watch -p -n 1 scontrol show jobid -dd 2581
```

A special note on how to use different GPU types [PCI-E vs SXM2] on Wiener.

As mentioned in a previous section, Wiener has two types of nodes (phase 1.0 and phase 2.0). The difference between these nodes is that phase 1.0 nodes used 2 * PCI-E 16GB Volta V100 GPU's in a single node. Phase 2.0 nodes use 4 * 32GB SXM2 connected Volta V100 GPU's. A further difference is the P2P (peer to peer) nVlink 2.0 connectivity of SXM2 GPU's. The phase 2.0 nodes are suited to a further different class of tasks, in addition to being able to perform any of the tasks the phase 1.0 nodes can. Some examples of use cases for phase 2.0 nodes:

a) Where the codes you are running can leverage P2P between GPUs using the nVlink topology.
b) Where the memory footprint of the job is larger than a 16GB GPU HBM2 memory map.
c) Where you require a number of GPUs to share memory and process control across MPI ranks explicitly in a single nodes.
d) You need four GPUs at once in the topology of a single node.

In general, the SXM2 socket version of the nodes run ~12% faster in most machine learning codes - but in P2P workloads they might exhibit as much as a 10x speedup in communication channels between GPU to GPU for memory copy or process copy functions. This is highly dependent upon your code, of course.

Requesting phase 2.0 wiener nodes.

The GRES syntax changes slightly for phase 2.0 nodes. Examples are as follows:

To request four SXM2 based GPU's on phase 2.0 nodes:

```
--gres=gpu:tesla-smx2:4
```

This is in contrast to Wiener phase 1.0 nodes where you'd request two GPUs in a node as follows:

```
--gres=gpu:tesla:2
```

There is a third option for gres which allow you to use any GPU available (effectively a scavenging method to just get any GPU you can!):

```
--gres=gpu:1
```

If your job is not otherwise specified in terms of GPU topology requirements and can simply use a single GPU to do some work we recommend using **`--gres=gpu:1`** if you just need "any old GPU" and want to run a job. It gives you and the SLURM scheduler the absolute best chance of finding the right space and shape for your job. The reality is, the power of a single GPU is in many cases more than enough for many workloads.

How busy is Wiener?

One can gain a rough understanding of how "full" or busy Wiener is via:

```
sinfo
```

As follows:

```
[uqjcarr1@wiener ~]$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
gpu* up infinite 2 mix gpunode-0-[1,3]
gpu* up infinite 2 alloc gpunode-0-[0,2]
gpu* up infinite 13 idle gpunode-0-[4-16]
```

In this case, Wiener is effectively idle, with four nodes doing something and the rest "asleep".

Some explanation on the way SLURM is set up.

- A lot of people who consume supercomputing resources are interested in the way their scheduler is configured.
- Of particular interest to many (with good reason) is "fairness" of the scheduler.
- Wiener uses a FSQ or FairShareQueue policy.
- A fair share queue policy is a complex thing to explain, but in the very most simple terms, the less you run, the more opportunity you have to have a higher chance at gaining access to a resource, compared to a user who may have run lots and consumed many more hours of CPU time. A full treatment of the concept of multi-factor priority, weighting and FSQ can be found here:

https://slurm.schedmd.com/priority_multifactor.html

- It is a complex topic which demands a full section of this wiki on its own, at some point.

How do I cancel or delete jobs?

Easy! Find your job using squeue:

```
[zebra@wiener ~]$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
123544 gpu Class2D uqblahbl R 1:23:51 2 gpunode-0-[0-1]
123334 gpu md01 uqblahbl R 10:59:01 1 gpunode-0-11
123333 gpu md01 uqblahbl R 11:40:19 1 gpunode-0-8
123332 gpu md01 uqblahbl R 12:15:48 1 gpunode-0-9
123474 gpu deepslid uqblahbl R 1-06:04:36 3 gpunode-0-[10,13,16]
123224 gpu 2LCST293 uqblahbl R 1-19:44:16 1 gpunode-0-3
123313 gpu 1LCST323 uqblahbl R 1-08:25:54 1 gpunode-0-7
```

```
123494 gpu 4MSD4 uqblahbl R 3:29:05 1 gpunode-0-15
123545 gpu 016354ms uqblahbl R 2:49:45 8 gpunode-0-[2-9]
123488 gpu md01 uqblahbl R 1-01:55:47 1 gpunode-0-12
123126_0 gpu SegTrain uqblahbl R 3-11:46:37 1 gpunode-0-4
123126_1 gpu SegTrain uqblahbl R 3-11:46:37 1 gpunode-0-5
123126_2 gpu SegTrain uqblahbl R 3-11:46:37 1 gpunode-0-6
123487 gpu md01 uqblahbl R 1-05:13:03 1 gpunode-0-14
123490 gpu ashsbuil uqblahbl R 1-05:15:21 1 gpunode-0-10
```

Let's say 123333 belongs to you.

To cancel it - simply issue `scancel 123333`.

Special cases for advanced scheduling techniques - Preemption and scavenger workloads.

Preemption and scavenger scheduling are advanced techniques in job-schedulers and supercomputing deployment. They are designed as a means to "fill capacity" and to give users the best possible opportunity to use as many resources as possible, whilst at the same time, not taking over resources or blocking otherwise normal workloads from running. This involves some complexity in the way quality of service, weighting and priority assignments are set up inside the scheduler. In short:

1. If you understand what preemption is or what scavenger scheduling is and have used it before, we would recommend you approach us via the support desk to enable it for your account.
2. If you understand the concept of checkpoint'ing workloads, granular workload outputs and how to effectively have your jobs "killed" but retain your job state - you have an appropriate workflow and job type for preemption.

Preemption QOS in SLURM works such that a job can be running in preemption mode - using as much free resources that is reasonably available, but the instant workloads that are "normal" in priority come along, your workload will be killed, "job by job" until another user, who satisfies normal run time semantics (qos=normal) has enough slots free to run their (in-quota) workload. This means that your preemption qos workload will be killed - and the only way to restart it, once resources are available is either via some automation (cron) or some other sbatch automation technique.

To be entirely clear preemption assumes your workloads and your understanding of scheduler automation is advanced and sophisticated enough that you **can** afford your jobs to be killed "mid flight" and the penality to your productivity is low, either because you are constantly checkpointing or your job/task is granularly outputting its results on the fly all the time.

Most importantly - merely having the preemption QOS applied to your account guarantees nothing. It all depends on how busy the rest of the supercomputer is in terms of "normal" qos workload and concurrent user priority weights as to how much or how little of the preemption will allow you to expand into (UEF - user expansion factor). If the system is idle (rare, on Wiener) you may get a lot of resources. If the system is busy, you may potentially get less than you would in using a standard "normal" qos.

If you would like to use preemption and scavenger scheduling QOS on Wiener - please contact the service desk to discuss your needs. It is our expectation that if you are asking for preemption, you understand the implications and the complexity to it, already.

Environmental modules (lmod). How to use modules, what they do, what they provide.

Modules are a long-standing capability and tradition in supercomputing to provide users easy access to all the path detail, libraries and environmental variables they need without the difficulty of setting complex paths in every script.

The modules mechanism is the way to help you find what you need to use installed software.

As an example:

```
module avail
```

Will list all modules available on the system, minus modules that have dependencies (those that need other modules loaded first, to become visible):

```
---------------------------------------------------------------------------- /opt/ohpc/pub/modulefiles
-----------------------------------------------------------------------------
amber/16 bazel/0.11.0 cuda/9.1.85.3 (D) gnu7/7.2.0 openblas/2.2.0_wiener_compile pgi/pgi/2018
```

```
amber/18 (D) bazel/0.12.0 (D) cuda91/fiji/150 imod/4.10.13 pgi/PrgEnv-pgi/18.1 pmix/1.2.3
anaconda/3.6 cmake/3.9.2 cuda91/gromacs/2018.1 intel/mkl_2018_update2 pgi/openmpi/2.1.2/2018 prun/1.2
arpack/0.96 cuda/9.0.176.1 cuda91/imagej/150 llvm4/4.0.1 pgi/pgi-llvm relion/2.0_mvapich_enabled_mpi
arpack/1 (D) cuda/9.1.85.1 gnu/5.4.0 matlab/R2017b pgi/pgi/18.1 (D) singularity/2.4.5

Where:
D: Default Module

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules matching any of the "keys".
```

Some background about modules structure on Wiener is provided by a README module module avail README/Modules

Some other information about the use of modules is available in that same section of the modules.

Not all modules show up when you run module avail. Modules that depend on compiler modules are hidden from view until the compiler module has been loaded.

For example, if you need to the mvapchi2 mpi library that was built with the same compiler as the software you want to use it with, then you need to first load the compiler module that it was built with (gnu7).

```
[uqjcarr1@headnode ~]$ module load gnu7
[uqjcarr1@headnode ~]$ module load mvapich2
[uqjcarr1@headnode ~]$ module list

Currently Loaded Modules:
1) gnu7/7.2.0 2) mvapich2/2.2
```

The "module spider" command is useful for searching for what you are looking for. If it is not there, an email to helpdesk@qbi.uq.edu.au should suffice, such that we can discuss your requirements.

Specialised software stacks and optimised frameworks on Wiener

Wiener is slightly different from a normal supercomputer, in that, beyond good/optimised compilers (Intel compilers, PGI, ifort) that provide an ostensible performance benefit, Wiener also contains several software services and modules that have been designed "accelerated" from the ground up for the specialised tools and technologies that Wiener was built to fulfil the capabilities of. These are:

- Relion
- TensorFlow
- Singularity
- AMBER GPU
- Ansys GPU
- Guppy

Each of these tools or codes was custom-spun on Wiener to leverage the best aspects of the system. A guide for each tool is provided below, such that new users who need to consume these technologies will be able to without having to go through the same low level optimisation process as we (RCC, QBI, UQ) went through in building the systems.

RELION

RELION (for REgularised LIkelihood OptimisatioN, pronounce rely-on) is a stand-alone computer program that employs an empirical Bayesian approach to refinement of (multiple) 3D reconstructions or 2D class averages in electron cryo-microscopy (cryo-EM). It is developed in the group of Sjors Scheres at the MRC Laboratory of Molecular Biology. Briefly, the ill-posed problem of 3D-reconstruction is regularised by incorporating prior knowledge: the fact that macromolecular structures are smooth, i.e. they have limited power in the Fourier domain. In the corresponding Bayesian framework, many parameters of a statistical model are learned from the data, which leads to objective and high-quality results without the need for user expertise. The underlying theory is given in Scheres (2012) JMB. A more detailed description of its implementation is given in Scheres (2012) JSB.

Relion was compiled (and thus requires the following module load) at UQ on Wiener using CUDA 9.1.85.3, mvapich2 MPI, GNU5.4.0 and pmix. The generic configure string to build Relion, using nvidia Volta SM_ARCH acceleration is as follows:

```
cmake3 -DDoublePrec_GPU=ON -DCUDA_ARCH=70 -DFORCE_OWN_FFTW=ON -DFORCE_OWN_FLTK=ON -DCMAKE_INSTALL_PREFIX=/opt/ohpc/pub
/apps/relion/ ..
```

Relion has several sub-modules that needed direct to sm_arch=70 CUDA compilation, within its module. These are:

```
[root@headnode bin]# ls -larth
total 83M
-rwxr-xr-x 1 root root 24M Jul 18 2015 unblur
-rwxr-xr-x 1 root root 25M Jul 18 2015 sum_movie
-rwxr-xr-x 1 root root 4.9K Jan 19 15:21 ctffind_plot_results.sh
-rwxr-xr-x 1 root root 31M Jan 19 15:21 ctffind
-rwxrwxr-x 1 root root 2.5M Apr 12 01:28 motioncor2
```

They are CUDA version dependent and thus, sensitive to the CUDA version which a user loads.

A common example of a well balanced Relion MPI, GPUdirect aware SLURM sbatch job has been provided as a sample, such that users can see an example of an optimal submission:

```
#!/bin/bash
#SBATCH --job-name=Class3D
#SBATCH --nodes=6
#SBATCH --ntasks=24
#SBATCH --mem-per-cpu=10g
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=4
#SBATCH --partition=gpu
#SBATCH --gres=gpu:tesla:2
#SBATCH --error=/nvmescratch/blah/Dataset_1/Class3D/job123/relion_run.err
#SBATCH --output=/nvmescratch/foo/Dataset_1/Class3D/job123/relion_run.out

module load cuda/10.0.130
module load mvapich2-gnu4/2.3
module load fftw-3.3.7-gcc-4.8.5-5igtfs5
module load relion/1.0_mvapich_cuda10

srun --mpi=pmi2 `which relion_refine_mpi` --o Class3D/job123/run --i Select/job120/particles.star --ref Reference.mr
```

ANSYS

The Ansys APDL, mechanical solvers and other tools have been custom-compiled for Wiener to take advantage of it's GPU backend. A sample script has been provided to envoke both CUDA, MPI and distributed workload management internally upon SLURM.

```
#!/bin/bash
#SBATCH --job-name=ansys_mech_example
#SBATCH --output=ansys_mech.out
#SBATCH --error=ansys_mech.err
#SBATCH --mail-user=me@uq.edu.au
#SBATCH --mail-type=END
#SBATCH --nodes=3
#SBATCH --cpus-per-task=1
#SBATCH --mem=48000
#SBATCH --tasks-per-node=28
#SBATCH --hint nomultithread     # Hyperthreading disabled
#SBATCH --gres=gpu:tesla:2
#SBATCH --error=/clusterdata/uqxxxx/ansys.outputs/ansys.err
#SBATCH --output=/clusterdata/uqxxxx/ansys.outputs/ansys.out

INFILE=ansys.input.run.txt
OUTFILE=ansys.outputs/ansys.input.run.output.txt

module load ansys
module load cuda
module load openmpi3
module load gnu7/7.2.0
ulimit -s unlimited

SLURM_NODEFILE=hosts.$SLURM_JOB_ID
srun hostname -s | sort > $SLURM_NODEFILE

export MPI_WORKDIR=ansys.outputs

mech_hosts=""
for host in `sort -u $SLURM_NODEFILE`; do
        n=`grep -c $host $SLURM_NODEFILE`
        mech_hosts=$(printf "%s%s:%d:" "$mech_hosts" "$host" "$n")
done

ansys191 -dis -acc nvidia -na 2 -b -machines ${mech_hosts%%:} -i $INFILE -o $OUTFILE
```

When GPU workloads cannot be containerised, there are further complications and a "fall back" to traditional memory and cores is in place. The complexity is, this will not be known until the job reaches the parallel processing steps of the APDL solver.

```
#!/bin/bash -ex

#SBATCH --job-name=ansys_apdl
#SBATCH --output=ansys.out
#SBATCH --error=ansys.err
#SBATCH --mail-user=someone@uq.edu.au
#SBATCH --hint nomultithread     # Hyperthreading disabled
#SBATCH --mail-type=END
#SBATCH --nodes=13
#SBATCH --ntasks=22
#SBATCH --cpus-per-task=1
#SBATCH --mem=330G
#SBATCH --error=/somewhere/ansys.err
#SBATCH --output=/somewhere/ansys.out

module load ansys

export MPI_WORKDIR=$PWD  #ANSYS will assume the working dir on rank!=0 processes is $HOME !!!!
srun hostname -s > /tmp//hosts.$SLURM_JOB_ID
if [ "x$SLURM_NPROCS" = "x" ]; then
  if [ "x$SLURM_NTASKS_PER_NODE" = "x" ];then
    SLURM_NTASKS_PER_NODE=1
  fi
  SLURM_NPROCS=`expr $SLURM_JOB_NUM_NODES \* $SLURM_NTASKS_PER_NODE`
fi
cat /tmp//hosts.$SLURM_JOB_ID
# format the host list for mechanical
mech_hosts=""
for host in `sort -u /tmp//hosts.$SLURM_JOB_ID`; do
  n=`grep -c $host /tmp//hosts.$SLURM_JOB_ID`
  mech_hosts=$(printf "%s%s:%d:" "$mech_hosts" "$host" "$n")
done
# run the solver
echo "hosts ${mech_hosts%%:}"
ansys191 -dis -b -machines ${mech_hosts%%:} -i file.log -dist_solver_v6.out
# cleanup
rm /tmp/hosts.$SLURM_JOB_ID
```

The further complexity is the balance of memory and CPU's. In this example, there are 13 nodes distributing the workload, due to the fact that the memory allocation per core is determined by the solver. I.e - much of the time, the only way to know how many nodes will distribute the in core memory correctly (without over-subscribing the node and OOM-killer kicking in) is trial and error and tail -f'ing the solver output log. Take note: each run and dataset will be different.

Another further complexity is the nature of the file inputs and outputs that Ansys uses. Typically, the workload will be modelled on a Windows host, using Windows path names. As such, the solver will need to export a ".inp" file, a .db file and a .log file.

The .log file represents the run control data. Paths will need to be updated inside this log file to point to the .db and .inp files for any job to run under linux. A $CWD with the .log, .db and .inp is the minimum required for input for the solver to run effectively (or at all).

Singularity.

Singularity can be thought of containerisation (Docker!) for supercomputing purposes. Wiener runs a Singularity embedded capability, which ships native with the OpenHPC community stack. This is useful for several reasons. Why would a person use containers on a supercomputer?

a) Your workload has so many dependencies to satisfy that only a pre-built container would suffice without wrecking the native "bare metal" installed environment.
b) The base OS running on Wiener for whatever reason, does not support your application (an example is a scenario where CentOS won't suffice, but Ubuntu would!).
c) Your workload is "pre-baked" and can only run in a specific container or node - and will not execute outside of that in a normal sbatch queue.
d) Sensitive codes.
e) Portability and potential scientific reproducibility - where a researcher may find themselves porting the same stack and workload to various supercomputers, but does not want to modify the toolchain (and thus introduce differences) into a workflow or in-silico experimental model. Take the whole container to each supercomputer and run exactly the same code.

For further background information about Singularity, consult the Software Containers User Guide (UQ networks only)

Singularity on Wiener

1. Calling singularity from an sbatch or srun can be easily preceded in scripts with:

```
module load singularity
```

2. Singularity on Wiener has been compiled to be MPI, CUDA and RDMA aware, as with all software stacks.

3. An example of a singularity run-string on Wiener might look like the following.

```
srun -N 1 -p gpu --gres=gpu:tesla:1 --pty bash singularity shell --nv /clusterdata/uqjcarr1/tensorflow_test_osg.simg
```

To explain what this is doing, step by step:

- First, we request one node (-N 1)
- Secondly, we say we will use the gpu partition (-p gpu)
- Third, we use the --gres directive to specify that we'd like to allocate one nvidia voldta GPU
- Next, we invoke a "real" shell with a --pty bash singularity shell command
- finally, we give our path to the singularity image (.simg) we've either downloaded or created ourselves

This will allocate you a piece of a node, direct to hardware passthrough with GPU to run your containerised workload on.

So, where do I get singularity images from - and how do I then run them?

There are several ways to obtain a singularity image.

1. You can make it yourself (advanced).
2. You can ask the admins to make one for you (a sensible route to take!).
3. You can obtain them from the singularity-hub:

http://singularity-hub.org

4. You can obtain them from the docker-hub:

https://hub.docker.com/

Singularity offers some smart integrated methods that allows transparent deployment of singularity native images and docker images (which turn into singularity images when pulled through "shub".

An example of running a singularity image that has been built for a specific Ubuntu distributed version of tensorflow-gpu from the singularity-hub:

```
module load cuda
module load singularity

singularity pull shub://opensciencegrid/osgvo-tensorflow-gpu:latest
Progress |===================================| 100.0%
Done. Container is at: ./osgvo-tensorflow-gpu.img


srun -N 1 -p gpu --gres=gpu:tesla:1 --pty bash singularity shell --nv /clusterdata/uqjcarr1/osgvo-tensorflow-gpu.img
```

One can even pull in Docker images from Docker-hub, and they will run natively inside singularity:

```
singularity pull docker://ubuntu
Initializing Singularity image subsystem
Opening image file: ubuntu.img
Creating 223MiB image
Binding image to loop
Creating file system within image
Image is done: ubuntu.img
Docker image path: index.docker.io/library/ubuntu:latest
Cache folder set to /home/uqjcarr1/.singularity/docker
Importing: base Singularity environment
Importing: /home/uqjcarr1/.singularity/docker
/sha256:b6f892c0043b37bd1834a4a1b7d68fe6421c6acbc7e7e63a4527e1d379f92c1b.tar.gz
Importing: /home/uqjcarr1/.singularity/docker
/sha256:55010f332b047687e081a9639fac04918552c144bc2da4edb3422ce8efcc1fb1.tar.gz
Importing: /home/uqjcarr1/.singularity/docker
/sha256:2955fb827c947b782af190a759805d229cfebc75978dba2d01b4a59e6a333845.tar.gz
```

```
Importing: /home/uqjcarr1/.singularity/docker
/sha256:3deef3fcbd3072b45771bd0d192d4e5ff2b7310b99ea92bce062e01097953505.tar.gz
Importing: /home/uqjcarr1/.singularity/docker
/sha256:cf9722e506aada1109f5c00a9ba542a81c9e109606c01c81f5991b1f93de7b66.tar.gz
Importing: /home/uqjcarr1/.singularity/metadata
/sha256:fe44851d529f465f9aa107b32351c8a0a722fc0619a2a7c22b058084fac068a4.tar.gz
Done. Container is at: ubuntu.img
```

General considerations and warnings around the use of containers.

- You are responsible for understanding the contents of a container. Know how it was built. Know what it does. Read the recipe. Don't assume it is safe or perfect if it has come from the public domain.
- Containers incur a very small amount of overhead. We consider them useful to containerise and "keep enclosed" complex dependencies, builds and workflows, beyond what a python virtual env or modules might. Be that as it may, they are not for everything. Our general recommendation is that one shouldn't use containers when a bare-metal installed module or environment will suffice.
- Containers in the form of Wiener's singularity will work with MPI + RDMA. Knowing how to achieve this in a multi-nodal deployment is another matter entirely that will be context sensitive to the singularity image that you import, or build.
- If you are in doubt about what you need - ask us.
- A common mistake when starting out in scheduling workloads with containers is that users forget to "module load cuda" as well, in the sbatch or srun string. Doing so will mean that, whilst the container is entirely capable of running CUDA code, it has no hardware level access or library level pass through to the physical GPU's in the node.

PyTorch.

Wiener has an nVidia CUDA 10 + Volta optimized implementation and compilation of the popular deep learning and machine vision framework, PyTorch. This includes torch "vision", also known as "torchvision".

To use the framework with your python code, insert these lines into your sbatch script in this order:

```
module load anaconda/3.6
source activate /opt/ohpc/pub/apps/pytorch_1.10
module load cuda/10.0.130
module load gnu/5.4.0
module load mvapich2
```

Some key things about this implementation:

It implements MAGMA-CUDA10 [http://www.icl.utk.edu/files/print/2017/magma-sc17.pdf] - so all the MAGMA linear algebra GPU extensions have been compiled in and are available to you, if you require this level of sophistication in your codes.

- We have compiled in the latest NCCL libraries (CUDA 10 NCCL) and cublas/curand for optimal performance.
- Multi-node/distributed pytorch and multi-GPU pytorch has been enabled. This is an advanced topic beyond the scope of this email to dwell any further upon, however.
- This compliation implements MVAPICH2 for its MPI distributed PyTorch backend.
- If you call Tensors in your matrix/array code - you will automatically offload to the Volta's specialised hardware tensor cores.

PyTorch has an overwhelmingly large range of uses - from basic machine vision tasks, natural language processing, cellular recognition/detection in microscopy through to more advanced artificial intelligence and ML models such as Generative Adversarial Nets and DCGAN's.

TensorFlow.

TensorFlow™ is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

On Wiener, TensorFlow's current baseline version is 1.13, production (stable) running compiled against CUDA 10.0.130. To use TensorFlow on wiener, you must load the following modules/call the following modules in your SBATCH or srun/salloc commands:

```
module load cuda/10.0.130
module load gnu7
module load openmpi3
module load anaconda/3.6
source activate /opt/ohpc/pub/apps/tensorflow_1.13
```

You can then load your tensorflow as follows in python:

```
import tensorflow as tf
```

Horovod.

Horovod is a distributed training framework for TensorFlow, Keras, PyTorch, and MXNet. The goal of Horovod is to make distributed Deep Learning fast and easy to use. Horovod is hosted by the LF AI Foundation (LF AI).

The primary motivation for this project is to make it easy to take a single-GPU TensorFlow program and successfully train it on many GPUs faster. This has two aspects:

1. How much modification does one have to make to a program to make it distributed, and how easy is it to run it?
2. How much faster would it run in distributed mode?

Internally at Uber they found the MPI model to be much more straightforward and require far less code changes than the Distributed TensorFlow with parameter servers.

As an example and so that users can see what we've built:

```
base) [uqjcarr1@wiener ~]$ srun -N 1 --mem=100G -p gpu --gres=gpu:tesla:1 --pty bash
(base) [uqjcarr1@gpunode-0-4 ~]$ module load gnu7
(base) [uqjcarr1@gpunode-0-4 ~]$ module load anaconda/3.7
(base) [uqjcarr1@gpunode-0-4 ~]$ module load cuda/11.1.1
(base) [uqjcarr1@gpunode-0-4 ~]$ conda activate /opt/ohpc/pub/apps/horovod_2021/horovod-gpu-data-science-project/env
(/opt/ohpc/pub/apps/horovod_2021/horovod-gpu-data-science-project/env) [uqjcarr1@gpunode-0-4 ~]$ horovodrun --check-
2021-03-17 09:43:30.790317: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dyna
2021-03-17 09:43:36.871208: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dyna
2021-03-17 09:43:38.272727: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dyna
2021-03-17 09:43:39.714420: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dyna
2021-03-17 09:43:41.015689: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dyna
2021-03-17 09:43:42.331390: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dyna
Horovod v0.21.3:

Available Frameworks:
    [X] TensorFlow
    [X] PyTorch
    [ ] MXNet

Available Controllers:
    [X] MPI
    [X] Gloo

Available Tensor Operations:
    [X] NCCL
    [ ] DDL
    [ ] CCL
    [X] MPI
    [X] Gloo
```

So you can see above what Frameworks we have been able to compile against, the message passing controllers, and the tensor tech we've got available for matrix-mul distribution.

In order to USE Horovod with either a PyTorch or TensorFlow backend in SLURM, an SBATCH script such as this would be required:

```
#!/bin/bash
#SBATCH --job-name=horovod_scaler
#SBATCH --output=horo.out
#SBATCH --error=horo.err
#SBATCH --nodes=3
#SBATCH --ntasks=3
#SBATCH --cpus-per-task=14
#SBATCH --mem=100g
#SBATCH --partition=gpu
#SBATCH --gres=gpu:tesla-smx2:2

export HOROVOD_CUDA_HOME=$CUDA_HOME
export HOROVOD_GPU_OPERATIONS=NCCL

module load gnu7
module load anaconda/3.7
module load cuda/11.1.1
```

```
eval "$(conda shell.bash hook)"
conda activate /opt/ohpc/pub/apps/horovod_2021/horovod-gpu-data-science-project/env
cd /scratch/qbi/uqjcarr1/horovod/examples/tensorflow2
mpiexec -np 6 -env HOROVOD_TIMELINE=/clusterdata/uqjcarr1/3.node.6.gpu.tensorflow2.synthetic.json -env HOROVOD_TIMEL
```

We will go through this part by part.

In this case, you'll see we have asked for three nodes worth of compute. We asked to run 3 tasks per node. This is more or less an MPI rank per task. You'll also note our GRES directive. We asked for 2 GPU's per node of the SXM2 type GPU layout/topology.

We export very specific HOROVOD environmental vars here so Horovod knows what to do with it's GPU operations. In this case we use nvidias NCCL - you need the collective communication library plugged into the edge of all of this or you can't pass your IO across to the GPU direct solvers running in each node that you "wire up" by way of the MPI implementation you invoke. NCCL is "key" to horovod working.

Next, you'll see the modules we load. You will note we DO NOT load openmpi or any other MPI implementation here at all. This is where it usually all goes wrong for people. Leave MPI alone at this juncture. We will explain why in a moment.

Moving along - the eval statement gets you to a point where your conda environments can load correctly.

Next, the conda activate statement. That will drop your shell to the self-enclosed frameworks space to make this easier. It includes the **right** MPI for the job. What this means is that you're not actually using OpenMPI. You're using MPICH3.4.1.

Next, we navigate into a directory where we've got some specific TensorFlow2 distributed Horovod benchmarks and stream training examples

We then execute a run using 6 MPI ranks and we then build out our Horovod profiling timeline for later use so we can see how my code is operating/running on each tensor load/pass point. We want to run in fp16 and an allReduce semantic, so we flag as such.

LAMMPS

The following is a sample job script for running LAMMPS on Wiener.
It may require some modification.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --job-name=lammps_test
#SBATCH -n 1
#SBATCH -c 1
#SBATCH --mem=5000
#SBATCH -o lj.txt
#SBATCH -e lj.txt
#SBATCH --partition=gpu
#SBATCH --gres=gpu:tesla:1
#SBATCH --time=00:01:0

module load gnu/5.4.0
module load cuda/10.0.130
module load ffmpeg-3.2.4-gcc-7.2.0-toyq4wp
module load openmpi/1.10.7
module load fftw/3.3.7
module load lammps/28Jan2019

srun lmp -sf gpu -in in.lj
```

Guppy