Garrawarla Documentation

On this page:

Introduction

The Pawsey Supercomputing Centre has installed a new GPU-enabled system called **Garrawarla,** a **Wajarri** word meaning **"Spider"**, to enable our Murchi son Wide Field Array (MWA) researchers to produce scientific outcomes while the Pawsey Supercomputing System is being procured. This MWA compute cluster provides the latest generation of CPUs and GPUs, high memory bandwidth, and increased memory per node to allow MWA researchers to effectively process large datasets.

System details

System configuration

Garrawarla has the following hardware characteristics:

78 "HPE XL190 Gen10" compute nodes, each with:

- 2x Intel Xeon Gold 6230 20 core 2.1GHz CPU, code name "Cascade Lake".
- 384GB RAM
- 1x 240GB SSD boot drive
- 1x 960GB NVMe drive
- 1x HDR100/Ethernet 100Gb ConnectX-6 with single QSFP56 port
- 1x V100 32GB NVIDIA GPU

Mounted Global File Systems

The **Astronomy** Filesystem (/astro), which is dedicated to MWA, is the only Lustre file system mounted on this cluster. It is provided by HPE, with 3 PB of usable space and capable of reading/writing at 30 GB/s.

Contact the Pawsey Service Desk Portal if you need assistance in migrating your workflows to Garrawarla.

IMPORTANT

Currently, the /group file system has many clients on it, and it is mounted on several locations. As a result it is the most impacted resource at Pawsey due to high metadata load. By **not** mounting /group on Garrawarla, any performance degradation issues with /group will not have an impact on Garrawarla because there will be no Lustre client connections from Garrawarla to the /group Lustre servers.

This approach will cause some disruption to workflows as some steps of a job may include stage in/out of data from/to /group via the copyq on Zeus. Data workflows presents an example to copy results to local storage with the use of copy queue on Zeus at the end of simulation on Magnus. You can modify this script to copy results from/to /group before/after the simulation on Garrawarla.

Available Software

Garrawarla includes the following software characteristics:

- SLES12 with SP5 Operating System
- Slurm 20.02.4 queueing system,
- Compilers: gcc/8.3.0, gcc/5.5.0, gcc/10.1.0 intel/19.0.5, pgi/20.1, clang/10.0.0
- Python/3.8.2
- CUDA/10.2
- MPI: OpenMPI/4.0.3, OpenMPI/4.0.2, IntelMPI/19.0.5
- Singularity/3.5.2
- Profilers: ARM Forge/19.0.3, ARM Forge/20.0.3, Intel VTune/19.0.5, NVIDIA Nsight/2019.5.2

Python/2.7.17 and its associated packages are installed in the /pawsey/mwa/software/mwa_sles12sp4 directory.

IMPORTANT: Run "module use /pawsey/mwa/software/mwa_sles12sp4/modulefiles" before loading these modules.

```
ddeeptimahanti@garrawarla-1:~> module use /pawsey/mwa/software/mwa_sles12sp4/modulefiles
ddeeptimahanti@garrawarla-1:~> module avail

---------------------------- /pawsey/mwa/software/mwa_sles12sp4/modulefiles -----------------------------
   argparse/1.4.0                     (D)    distribute/0.7.3    (D)    pyparsing/2.4.7        (D)
   astropy/2.0.16                            ephem/3.7.7.1       (D)    pytest/4.6.9
   attrs/19.3.0                       (D)    funcsigs/1.0.2      (D)    python-dateutil/2.8.1  (D)
   backports.functools_lru_cache/1.6.1 (D)   functools32/3.2.3-2 (D)    python/2.7.17
   backports_abc/0.5                  (D)    h5py/2.9.0                 pytz/2019.3            (D)
   boost/1.66.0                       (D)    healpy/1.13.0       (D)    scipy/1.2.3
   casacore/2.4.1                            matplotlib/2.1.0           setuptools/38.2.1
   casacore/3.2.1                     (D)    mpi4py/3.0.3        (D)    singledispatch/3.4.0.3 (D)
   certifi/2020.4.5.1                 (D)    numpy/1.13.3               sip/4.19.8
   configparser/4.0.2                        pluggy/0.13.1       (D)    six/1.14.0             (D)
   cycler/0.10.0                      (D)    psycopg2/2.8.5      (D)    subprocess32/3.5.4     (D)
   cython/0.29.14                     (D)    py/1.8.1            (D)    tornado/6.0.4          (D)
   d2to1/0.2.12.post1                 (D)    pyfits/3.5          (D)    zipp/1.2.0
...
```

Log in

Interaction with Garrawarla is done remotely using SSH (Secure Shell version 2, SSH-2):

```
localComputer:~> ssh username@garrawarla.pawsey.org.au
```

More information about SSH-based access: https://support.pawsey.org.au/documentation/display/US/Logging+in

Select and use a compiler

There are two families of supported software compilers on Garrawarla:

- Intel
- GNU: GNU Compiler Collection 8.3.0 is loaded by default.

It is up to you, as the user, to decide which programming environment is most suitable for the task at hand. To know the available gcc versions:

```
ddeeptimahanti@garrawarla-1:~> module avail gcc

------------------------------ /pawsey/mwa_sles12sp4/modulefiles/devel -------------------------------
   gcc/4.8.5    gcc/5.5.0    gcc/8.3.0 (L,D)    gcc/10.1.0
```

In the above in brackets, the L means the module is loaded, and D means it is the default version if no version is specified during the module load.

Compiler executables are named as follows:

Intel		GNU	
Language	**Compiler executable**	**Language**	**Compiler executable**
C	icc	C	gcc
C++	icpc	C++	g++
Fortran	ifort	Fortran	gfortran

Type the "man" command followed by the compiler name to load the corresponding manual page.

Compiling MPI code

MPI libraries can be loaded using the corresponding modules. Use of OpenMPI with Unified Communication X is recommended for normal use cases, and can be achieved by loading the appropriate module:

```
module load openmpi-ucx/4.0.3
```

Once the MPI library is loaded, MPI wrappers are available for the currently selected compiler. Example commands follow:

OpenMPI-UCX				Intel-MPI			
Intel		GNU		Intel		GNU	
Language	**Command**	**Language**	**Command**	**Language**	**Command**	**Language**	**Command**
C	mpicc hello_mpi.c	C	mpicc hello_mpi.c	C	mpiicc hello_mpi.c	C	mpicc hello_mpi.c
C++	mpicxx hello_mpi.cpp	C++	mpicxx hello_mpi.cpp	C++	mpiicpc hello_mpi.cpp	C++	mpicxx hello_mpi.cpp
Fortran	mpif90 hello_mpi.f90	Fortran	mpif90 hello_mpi.f90	Fortran	mpiifort hello_mpi.f90	Fortran	mpif90 hello_mpi.f90

IMPORTANT: Always use srun to launch a MPI executable, regardless of whether it is OpenMPI or Intel-MPI.

In the case of Intel Compilers + Intel MPI, wrapper names, there are differences when compared to three other combinations: mpiicc, mpiicpc and mpiifort (in contrast to the usual mpicc, mpicxx and mpif90).

Codes compiled with OpenMPI will not work properly with libraries compiled with Intel MPI and vice versa. Make sure that all linked libraries are compiled with the same MPI implementation used for your parallel MPI code.

Compiling OpenMP code

To compile code for OpenMP multi-threading, add specific flags at compile time. The syntax differs depending on the selected compiler:

Intel		GNU	
Language	**Command**	**Language**	**Command**
C	icc -qopenmp hello_omp.c	C	gcc -fopenmp hello_omp.c
C++	icpc -qopenmp hello_omp.cpp	C++	g++ -fopenmp hello_omp.cpp
Fortran	ifort -qopenmp hello_omp.f90	Fortran	gfortran -fopenmp hello_omp.f90

Refer to Compiling on Zeus#Usefulcompileroptions for useful compiler options while compiling code on Garrawarla.

Compiling GPU code

GPU code compilation should occur on the compute nodes in the gpuq partition, either interactively for simple programs or via a job script for larger software suites.

Compiler compatibility notice: CUDA versions up to 10.2 are compatible with Intel compilers and GCC compilers.

NVIDIA V100

All the nodes in Garrawarla are equipped with NVIDIA v100 GPUs, based on the NVIDIA Volta architecture and are accessible from the gpuq partition.

Compiling a CUDA application

Compiling interactively

1. To compile interactively, submit a job request with salloc:

```
salloc --partition gpuq --time 1:00:00 --nodes 1 --gres=gpu:1
```

The terminal appears to hang until the job starts.

Once the job has started, your login prompt displays the compute node in the gpuq you are now on.

2. Ensure that the module for the desired compiler is loaded. The current default on Garrawarla is gcc/8.3.0. A different GNU version or the Intel compiler can be loaded with module swap, e.g.:

```
module swap gcc gcc/5.5.0
```

3. Load the CUDA module:

```
module load cuda
```

4. To compile MPI-enabled CUDA code, load the OPENMPI-UCX-GPU module as well:

```
module load openmpi-ucx-gpu
```

5. Execute compile and link stages jointly or separately:

5a. Execute compilation commands jointly, e.g.:

```
srun nvcc -O2 -arch=sm_70 code_host.c code_cuda.cu
```

5b. **Alternatively**, if you require separate compilation and link stages, compile with the "-c" option first, e.g.:

```
srun g++ -O2 -c code_host.cpp
srun nvcc -O2 -arch=sm_70 -c code_cuda.cu
```

5c. Continue with the link stage. Make sure the link stage takes place using the host compiler, and includes the CUDA run time library via "-lcudart":

```
srun g++ code_cuda.o code_host.o -lcudart
```

Compiling using a job script

To compile via a job script:

1. Prepare the script to request a node, load the relevant environment, and execute the compilation commands. For example, create a script file named compile.slurm which contains:

```
#!/bin/bash --login
#SBATCH --nodes=1
#SBATCH --partition=gpuq
#SBATCH --gres=gpu:1
#SBATCH --time=00:10:00
#SBATCH --account=[your-account]

module load cuda

srun nvcc -O2 -arch=sm_70 code_host.c code_cuda.cu
```

2. Submit the script called compile.slurm to the queue:

```
sbatch compile.slurm
```

Compiling an OpenACC application

The PGI compiler (v20.1) is available for compiling code that contains OpenACC directives.

C, C++ and Fortran compilers are invoked using pgcc, pgc++ and pgfortran, respectively.

You can compile either interactively via salloc or in a batch job:.

Compiling interactively

```
salloc --partition gpuq --time 1:00:00 --nodes 1 --gres=gpu:1

module swap gcc pgi
srun pgcc -acc code_openacc.c
```

Compiling using a batch job

```
#!/bin/bash --login
#SBATCH --nodes=1
#SBATCH --partition=gpuq
#SBATCH --gres=gpu:1
#SBATCH --time=00:10:00
#SBATCH --account=[your-account]

module swap gcc pgi

srun pgcc -acc code_openacc.c
```

Review the queuing system configuration

Garrawarla resources are managed by the Slurm queueing system. For detailed information about Slurm, refer to: https://support.pawsey.org.au/documentation/display/US/Job+Scheduling

Garrawarala has two overlapping partitions with all 78 nodes available in both partitions (queues):

- workq- for CPU-only jobs; with only 38 cores available in each node (mwa001-mwa078), max. 24h walltime. The remaining 2 cores are available to support GPU jobs in each node.
- gpuq - for GPU-only jobs; with all 40 cores and single GPU available in each node (mwa001-mwa078), max. 24h walltime. You can either request the entire node (with 40 cores and single GPU) or only 2 cores + single GPU for your GPU jobs. Any non-GPU job requests are automatically rejected from this partition by Slurm.

Submit your job

Garrawarla compute nodes are in both partitions and are configured as a shared resource. This means that it is especially important in a request for the GPU to specify the number of tasks and amount of main memory required by the job. If not specified, by default a job is allocated with a single CPU core, no GPU and around 9GB of RAM.

It is recommended that all jobs request the following:

Option	Purpose
`--account=account`	Set the account to which the job is to be charged. A default account is configured for each user.
`--nodes=nnodes`	Specify the total number of nodes.
`--ntasks=number`	Specify the total number of tasks (processes).
`--gres=gpu:1`	Specify GPUs per node
`--ntasks-per-node=number`	Specify the number of tasks per node.
`--ntasks-per-socket=number`	Specify the number of tasks per socket.
`--cores-per-socket=number`	Specify the number of cores per socket. **Note:** Each node has two CPU sockets with 18 cores and 20 cores, respectively, to support GPU workflows.
`--cpus-per-task=number`	Specify the number of threads per process for multi-threaded jobs.
`--mem=size`	Specify the memory required per node. **Note:** If this option is not used, the scheduler allocates approximately 9gb of memory per process.
`--partition=partition`	Request an allocation on the specified partition. If not specified, jobs are submitted to the default partition.
`--time=hh:mm:ss`	Set the wall-clock time limit for the job.

Refer to the following examples, which demonstrate different job allocation modes, including how to access the local NVMe storage.

Batch Job Examples

Requesting NVMe resources in SLURM

Each node in Garrawarla has an attached NVMe device with 890GB usable space mounted as /nvmetmp.

Request a specific amount of NVMe storage in your job script using `--gres=tmp:<some-value>g` **or** `--tmp=<some-value>g` directives, and request up to 890GB. If both commands are used, only `--gres` is applied. You should not be able to use more NVMe space than what has been allocated to you. By default, without any explicit NVMe request, a job should get allocated 1G of a /nvmetmp on the NVMe device.

The NVMe device (or the portion used by a job) is cleaned up after the job completes. **IMPORTANT:** Migrate any valuable results from the NVMe device before the job completes.

To request 200GB NVMe space	salloc -N 1 --tmp=200g	`salloc: Nodes mwa001 are ready for job`
		`wdavey@mwa001:~> df -h | grep /nvmetmp`
		`/dev/nvme0n1p1  200G  0  200G  0%`
		`/nvmetmp`

Serial job using a single CPU core

In the following example, we assume that the `serial_code` is a serial application that is to be run on a single core in the workq partition. The amount of memory available for the job is adjusted since by default it would be given only about 9GB per process.

```
#!/bin/bash -l
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --mem=380gb
#SBATCH --time=00:01:00
#SBATCH --partition=workq
#SBATCH --account=[your-project]

#load required modules
srun -n 1 ./serial_code
```

OpenMP code using all available CPU cores per node

In the following example, we assume that the `cpu_code` is an OpenMP code using all 20 CPU cores of a socket. **Note:** Each node has 2 CPUs, with 20 cores each. A single process per socket and 20 OpenMP threads (single CPU) is run, leaving the other resources (the other CPU) available for other jobs. The amount of memory is adjusted to 180GB since by default it would be given only 9GB per process. In this example, you are allocated with CPU cores and memory of a single CPU (single NUMA node).

Note: To facilitate GPU workflows, only 38 cores are available on a node in the workq partition, with 18 cores on CPU-1 and 20 on CPU-2. For best performance with OpenMP applications, it is recommend to launch threads in a single CPU/NUMA node.

```
#!/bin/bash -l
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cores-per-socket=[some-value] # up to 18 or 20 to explicitly request CPU socket with 18 or 20 cores
respectively
#SBATCH --cpus-per-task=[some-value] # To run threaded code and should be less than or equal to the above cores-
per-socket value
#SBATCH --mem=180gb
#SBATCH --time=00:01:00
#SBATCH --partition=workq
#SBATCH --account=[your-project]

export OMP_NUM_THREADS=20  # This should be equal to cpus-per-task value
srun -n 1 -c ${OMP_NUM_THREADS} ./cpu_code
```

Non-MPI code using a single GPU

In the following example. we assume that the `gpu_code` is a non-MPI application and can use a single GPU. The amount of memory available for the job since is adjusted by default, and the job is given about 9gb per process.

Note: For best performance with OpenMP applications, it is recommend to launch threads within a single CPU (or NUMA node or socket). Each socket

```
#!/bin/bash -l
#SBATCH --nodes=1
#SBATCH --gres=gpu:1
#SBATCH --ntasks-per-node=1
#SBATCH --cores-per-socket=[some-value] # up to 20 to explicitly request cores from a single CPU socket
#SBATCH --cpus-per-task=[some-value] # To run threaded code and should be less than or equal to the above cores-
per-socket value
#SBATCH --mem=380gb
#SBATCH --time=00:01:00
#SBATCH --partition=gpuq
#SBATCH --account=[your-project]

module load cuda
export OMP_NUM_THREADS=20 # This should be equal to cpus-per-task value
srun -n 1 -c ${OMP_NUM_THREADS} ./gpu_code
```

MPI code using more than one GPU

In the following example, we assume that the `gpu_code` is a MPI application and can use a single GPU per process. Two processes are run, one per node, and we adjust the amount of memory per node since by default the job is given 9gb per process.

```
#!/bin/bash -l
#SBATCH --nodes=2
#SBATCH --gres=gpu:1
#SBATCH --ntasks-per-node=1
#SBATCH --mem=380gb
#SBATCH --time=00:01:00
#SBATCH --partition=gpuq
#SBATCH --account=[your-project]

module load cuda
srun -n 2 -N 2 ./gpu_code
```

OpenMP code using a single GPU and all available CPU cores

In the following example, we assume that the `gpu_code` is an OpenMP code using a single GPU and all 20 CPU cores. **Note:** Each node has 2 CPUs, with 20 cores each. A single process per socket and 20 OpenMP threads (single CPU) is run, leaving the other resources (CPU) available for other jobs. The amount of memory is adjusted to 180gb since by default the job is given 9gb per process. In this example, the job is allocated with CPU cores and memory of a single CPU (single NUMA node).

Note: For best performance with OpenMP applications, it is recommended to launch threads within a single CPU (or NUMA node or socket).

```
#!/bin/bash -l
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cores-per-socket=[some-value] # up to 20 to explicitly request cores from a single CPU socket
#SBATCH --cpus-per-task=[some-value] # To run threaded code and should be less than or equal to the above cores-per-socket value
#SBATCH --gres=gpu:1
#SBATCH --mem=180gb
#SBATCH --time=00:01:00
#SBATCH --partition=gpuq
#SBATCH --account=[your-project]

module load cuda
export OMP_NUM_THREADS=20 # This should be equal to cpus-per-task value
srun -n 1 -c ${OMP_NUM_THREADS} ./gpu_code
```

MPI + OpenMP code using the GPU and all available CPU cores per node

In the following example, we assume that `gpu_code` is an MPI + OpenMP code using a single GPU per process and capable of using OpenMP multi-threading to additionally use all CPU cores in a node. **Note:** There are 2 CPUs per node, with 20 cores each. The code is run on two nodes with one process per node, each using a single GPU. The amount of memory is adjusted to 180gb since by default the job would be given 9gb per process.

```
#!/bin/bash -l
#SBATCH --nodes=2
#SBATCH --gres=gpu:1
#SBATCH --ntasks-per-node=1
#SBATCH --cores-per-socket=[some-value] # up to 20 to explicitly request cores from a single CPU socket
#SBATCH --cpus-per-task=[some-value] # To run threaded code and should be less than or equal to the above cores-per-socket value
#SBATCH --mem=180gb
#SBATCH --time=00:01:00
#SBATCH --partition=gpuq
#SBATCH --account=[your-project]

module load cuda
export OMP_NUM_THREADS=20 # This should be equal to cpus-per-task value
srun -n 2 -c ${OMP_NUM_THREADS} ./gpu_code
```

Run a job using interactive mode

As on other Pawsey systems, you can used the `salloc` command to run interactive sessions. You can use the#SBATCH options mentioned above to specify various interactive job parameters. For example, to run an OpenMP code using 1 GPU, you can open an interactive session with the following command:

```
salloc --nodes=1 --gres=gpu:1 --ntasks-per-socket=1 --cores-per-socket=20 --cpus-per-task=20 --mem=180gb --
time=00:05:00 --partition=gpuq --account=[your-project]
```

For all interactive sessions, after `salloc` has run and you are on a compute node, use the `srun` command to execute your commands. This is valid for all commands. For example, used `srun` to run the `nvidia-smi` command on the interactive node:

```
ddeeptimahanti@mwa041:~> srun nvidia-smi
Sun May 24 16:48:39 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-PCIE...  On   | 00000000:D8:00.0 Off |                    0 |
| N/A   34C    P0    23W / 250W |      0MiB / 32510MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
ddeeptimahanti@mwa041:~>
```

Resource Accounting

Pawsey provides a tailored suite of tools called pawseytools which is already configured to be a default module upon login. The pawseyAccountBalance utility in pawseytools shows the current state of the default group's usage against their allocation and also the /astro storage quota and usage. For example:

```
ddeeptimahanti@garrawarla-1:~> pawseyAccountBalance --cluster=garrawarla -p mwaops -storage
Compute Information
-------------------
          Project ID     Allocation          Usage     % used
          ----------     ----------          -----     ------
              mwaops         100000           1381        1.4

Storage Information
-------------------
/astro usage for mwaops, used = 2.09 TiB, quota = 20.00 TiB
```

Troubleshooting & Good Practices

Using singularity with GPUs

Use the `--nv` option when using Singularity on Garrawarla compute nodes.

Segmentation fault while running CUDA/OpenACC applications with UCX support

A segmentation fault can occur if applications are either statically linked to CUDA libraries or memory is allocated before MPI_Init. As a workaround, disable memory type cache by exporting UCX_MEMTYPE_CACHE=n

Using ramdisk support on the compute nodes

Each node on Garrawarla has up to 50% of the memory (~185GB) mounted in /dev/shm and available as ramdisk, which can be used to speed up large I /O intensive computations. This resource is not trackable in Slurm, so you should cleanup /dev/shm before exiting the job, which otherwise will reduce the memory available for subsequent jobs on that node. Also, to be fair with system usage, request cores according to the ramdisk usage. For example, by default only 9gb is available per core; therefore, to use 90gb of ramdisk you should ask for an additional 10 cores to avoid issues for other jobs running on the same node.

```
ddeeptimahanti@mwa001:~> df -h | grep /dev/shm
tmpfs                                     189G     0  189G   0% /dev/shm
ddeeptimahanti@mwa001:~>
```