

Gust test system user guide

Gust is a small test system designed to prototype and mimic the hardware, software, user environment, and job execution configuration of the Derecho system, an HPE Cray EX cluster. Gust features both CPU and GPU nodes, a small Lustre scratch file system, the Cray Programming Environment (along with traditional compilers and libraries provided by CISL), and the PBS job scheduler.

Routine Gust maintenance will be conducted between 8 a.m. and 5 p.m. on Tuesdays. If scheduled maintenance downtime is not required, CISL will release the resource reservation and note that in the #gust-users Slack channel.

Page contents

- [Specifications](#)
- [Getting started on Gust](#)
- [Getting help and reporting issues](#)
- [Compiling CPU code](#)
- [Compiling GPU code](#)
- [Running compute jobs](#)

Specifications

16 CPU compute nodes	Dual-socket – 128 CPU cores 256 GB DDR4-3200 memory
2 GPU compute nodes	4 GPUs connected with 600 GB/s NVLink 40 GB High Bandwidth Memory (HBM2) per GPU device Single-socket – 64 CPU cores 512 GB DDR4-3200 memory
2 login nodes	1 CPU-only node and 1 heterogeneous node
CPU architecture	AMD EPYC™ 7763 Milan processors
GPU architecture	NVIDIA 1.41 GHz A100 40GB Tensor Core GPU
Interconnect	HPE Slingshot 11 – single-injection on CPU nodes and quad-injection on GPU nodes
Parallel file system	ClusterStor E1000 Lustre for scratch space
Operating system	Cray Linux Environment / SUSE Linux

Getting started on Gust

Logging in

Log in following this example.

```
ssh -Y username@gust.hpc.ucar.edu
```

Storage spaces

Gust users have access to three globally accessible personal storage spaces by default: home, work, and scratch. The table below provides further detail about each space.

Space	Storage Type	Quota	Policy
Home /glade/u/home/\$USER	Spectrum Scale (GPFS)	50 GB*	Backed up
Work /glade/work/\$USER	Lustre	2 TB**	
Scratch /glade/gust/scratch/\$USER	Lustre	30 TB	Unused data will be purged after 180 days

* *gladequota* will report 100 GB because dual copies of home directories are maintained for data robustness

** On Gust, both work and scratch usage count against the same per-user Lustre quota. On Derecho, work will be located on a GPFS file system with a separate quota from scratch.

The Gust system's home, work, and scratch spaces are separate and distinct from the corresponding production GLADE spaces on Cheyenne, Casper, and (eventually) Derecho. This is intentional in order to isolate the development/test system. To use files/data from production GLADE spaces in the Gust file spaces, you will need to transfer them manually.

Lustre best practices

The work and scratch file spaces on Gust use Lustre technology, unlike other recent disk systems at NCAR that use Spectrum Scale. Lustre systems have tools designed to maximize the usability and performance of both read/write operations and metadata operations (for example, listing directories and files). The **lfs** command is used to access these tools.

Many users are familiar with the GNU tool **find**, which can be used to display all objects (files, directories, and symlinks, for example) that match your search criteria. On Gust's scratch, use **lfs find** instead; it is optimized for the Lustre file system.

```
# Find WRF output files in a directory on Lustre scratch
lfs find run/ -name "wrfout*"
```

The **lfs** utility also provides tools for setting file stripe policies. Lustre file systems use striping to distribute chunks of file data across multiple object storage targets (OSTs). Increasing the stripe count can increase the I/O bandwidth when accessing the file, but performance can also suffer due to increased network communication when accessing the chunks of the file.

The optimal stripe size is sensitive to the data operation(s) being performed, so we recommend using the default stripe settings, which are tuned for typical workflows on the systems at NCAR.

Accessing software using environment modules

Basic utilities like vim, emacs, tar, and more are provided in the operating system environment, as you would see on a personal workstation. A large collection of compilers, development utilities, scientific libraries, and analysis software are also available via *environment modules*. When these "Lmod" modules are loaded, they configure your environment to provide access to executables and simplify compiling and linking your own codes.

Unlike other recent systems at NCAR, Gust's software stack is built on top of the *Cray Programming Environment (CPE)* – a collection of HPC-oriented software that HPE provides. The following modules are available to you by default on Gust:

- **ncarenv** - the current version of the software stack; grants access to software modules and provides basic utilities like *gladequota*, *qinteractive*, *qcmd*, and *qhst*
- **craype** – provides basic components and functionality of the CPE
- **cce** – the Cray compiler collection; includes C/C++/Fortran compilers
- **ncarcompilers** – wrapper scripts that will add library and header/module paths from other loaded library modules (e.g., *netcdf*) to your compiler and linker commands
- **cray-mpich** – the Cray MPI, based on (and ABI-compatible with) MPICH
- **cray-libsci** – numerical routines (BLAS, LAPACK, etc.) tuned for HPE-Cray hardware
- **hdf5** – data storage format (serial version)
- **netcdf** – data storage format built upon HDF5 (serial version)

You can interact with the environment modules system via the **module** command. Using this command, you can switch to alternative compilers and their software stacks (like NVIDIA's HPC SDK and Intel OneAPI). The following commands are commonly used:

- **module avail** – Show all available modules given the current compiler and MPI loaded, if any.
- **module load/unload** – Add or remove modules to/from your current environment. If a requested module conflicts with a currently loaded module, Lmod will report an error.
- **module list** – Show currently loaded modules.
- **module spider** – Search for modules based on a given term.
- **module purge** – Remove all modules from the current environment. This will not remove **craype**, however, as it is required.
- **module reset** – Reload the system-default set of modules.

It is also possible to save your own custom set of default modules. Simply load the desired modules and run *module save [NAME]*. If no name is given, it will be labeled default. When you sign into Gust, that default set will be loaded instead of the system default described above.

In general, if you want to customize your module environment, it is best to create a default module set rather than load modules in your startup *~/.profile* or *~/.tcshrc* file.

Getting help and reporting issues

For Gust user support, we have deployed a three-tiered approach using Slack, Help Desk tickets, and GitHub issues. Each space is detailed below. If you encounter any issues or have how-to questions, consider this guidance:

1. Check the [#gust-users](#) Slack channel first and ask any questions there for a quick response.
2. If indeed something is broken and it is likely related to an environment module, submit a GitHub issue in the [spack-gust](#) repository.
3. All other reports of bugs and problems can be sent to CISL's Consulting Services Group via the [Research Computing help desk](#).

Slack

We would like all early users of Gust to join the [#gust-users](#) Slack channel under the NCAR HPC User Group workspace. (If you are not currently subscribed to this workspace, you can do so through [this link](#).) Since Gust is a dynamic and evolving resource, we encourage Slack engagement as it allows broad user and administrator visibility and enables crowd-sourcing observations, tips, and tricks which we expect to evolve rapidly in the coming months.

Gust software stack GitHub issues

In addition to the above communication methods, software requests and bugs may be reported as GitHub issues in the [spack-gust](#) repository. This repo tracks all CISL software installations performed using Spack, most of which end up accessible via the modules system.

Research Computing help desk

Additional support mechanisms are also available through the [Research Computing help desk](#) ticket system.

Compiling CPU code

Using the Cray compiler collection

The Cray compiler collection includes **cc**, **CC**, and **ftn** for C, C++, and Fortran respectively. If you have the “ncarccompilers” module loaded, building code with dependencies such as netCDF is very easy. For example, compiling a simple Fortran code using netCDF is as follows with the compiler wrappers:

```
ftn -o mybin -lnetcdf mycode.f90
```

Meanwhile, if you did not have the “ncarccompilers” module loaded, you would need to run the following command instead, which includes include-path and linker flags:

```
ftn -I/path/to/netcdf/include -L/path/to/netcdf/lib -lnetcdf -o mybin mycode.f90
```

Using Cray MPICH MPI

Unlike other MPI libraries, Cray MPICH *does not* provide MPI wrapper commands like mpicc, mpicxx, and mpif90. Rather, you are expected to use the same cc, CC, and ftn commands as you would with a serial code. The CPE will add MPI build flags to your commands whenever you have the **cray-mpich** module loaded.

As many application build systems expect the MPI wrappers, our **ncarccompilers** module will translate a call to “mpicc” to “cc” (and likewise for the other languages) as a convenience, typically eliminating the need to alter pre-existing build scripts.

If you are using an MPI application compiled with GPU support, enable CUDA functionality by **loading a cuda module** and setting or exporting this environment variable before calling the MPI launcher in your job:

MPICH_GPU_SUPPORT_ENABLED=1

Also, if your GPU-enabled MPI application makes use of *managed memory*, you also need to set this environment variable:

MPICH_GPU_MANAGED_MEMORY_SUPPORT_ENABLED=1

At runtime, it is also necessary to pass information about job parallelism to the mpiexec (or mpirun / aprun) launcher because this information is not automatically taken from the PBS job script. You can pass this information by setting environment variables or by using mpiexec options. The primary settings you will need are:

- the number of mpi ranks (-n / PALS_NRANKS)
- the number of ranks per node (-ppn / PALS_PPN)
- the number of OpenMP threads or CPUs to associate with each rank (-d / PALS_DEPTH)
- binding options (--cpu-bind / PALS_CPU_BIND)

Cray MPICH has many tunable parameters that can be set through environment variables. Run **man mpi** for a complete listing.

Full details of runtime settings for launching parallel programs can be found by running **man mpiexec**. Example PBS select statements and corresponding mpi launch options for binding a hybrid MPI + OpenMP application (144 MPI ranks, and 4 OpenMP threads per MPI rank, which requires 5 nodes but does not fully subscribe the last node) are shown below. Examples for both methods – setting environment variables and passing options to mpiexec – are provided.

Environment variable example

```
#PBS -l select=5:ncpus=128:mpiprocs=32:ompthreads=4
export PALS_NRANKS=144
export PALS_PPN=32
export PALS_DEPTH=4
export PALS_CPU_BIND=depth
mpiexec ./a.out
```

mpiexec options example

```
#PBS -l select=5:ncpus=128:mpiprocs=32:ompthreads=4
mpiexec --cpu-bind depth -n 144 -ppn 32 -d 4 ./a.out
```

Using Intel compilers

The Intel compiler suite is available on Gust via the **intel** module. It includes compilers for C, C++, and Fortran codes. Note that it is **NOT** loaded by default.

To see which versions are available, use the **module avail** command.

```
module avail intel
```

To load the default Intel compiler, use **module load** without specifying a version.

```
module load intel
```

To load a different version, specify version number when loading the module.

Similarly, you can swap your current compiler module to Intel by using the **module swap** command.

```
module swap cce/14.0.3 intel
```

The table below provides a quick summary of the compile commands or flags needed to compile your C, C++, and Fortran codes.

LANGUAGE	SERIAL PROGRAMS	COMMANDS FOR PROGRAMS	FLAGS TO ENABLE OPENMP
	COMPILE COMMAND	USING MPI	(FOR SERIAL AND MPI)
Fortran	ifort foo.f90	mpif90 foo.f90	-qopenmp
C	icc foo.c	mpicc foo.c	-qopenmp
C++	icpc foo.C	mpicxx foo.C	-qopenmp

Extensive documentation for using the Intel compilers is available [online here](#). To review the manual page for a compiler, run the **man** command for it as in this example:

```
man ifort
```

Optimizing your code with Intel compilers

Intel compilers provide several different optimization and vectorization options. By default, Intel compilers use the **-O2** option, which includes some optimizations.

Using **-O3** instead will provide more aggressive optimizations that may not improve the performance of some programs, while **-O1** enables minimal optimization. A higher level of optimization might increase your compile time significantly.

You can also disable any optimization by using **-O0**.

Be aware that compiling CPU code with the Intel compiler on Gust is significantly different from using the Intel compiler on the Cheyenne system. Flags that are commonly used on Cheyenne might cause Gust jobs to fail or run much more slowly than otherwise possible.

- **DO use on Gust:** -march=core-avx2
- **Do NOT use on Gust:** -xHost , -axHost , -xCORE-AVX2 , -axCORE-AVX2

Examples

To compile and link a single Fortran program and create an executable:

```
ifort filename.f90 -o filename.exe
```

To enable multi-threaded parallelization (OpenMP) include the **-qopenmp** flag as shown here:

```
ifort -qopenmp filename.f90 -o filename.exe
```

Other compilers

In addition to the Cray compilers, we provide these on Gust:

- NVIDIA's HPC SDK
- the GNU Compiler Collection (GCC)

When using non-Cray compilers, you can utilize either the compiler collection's own commands (e.g., ifort, nvfortran) or the equivalent CPE command (ftn) as long as you have your desired compiler module loaded. If you do not have the **ncarccompilers** module loaded and you are using the **cray-mpich** MPI, you will need to use the CPE command.

Compiling GPU code

GPU applications should be built with either the Cray compilers or the NVIDIA HPC SDK compilers and libraries. In the following examples, we demonstrate the use of NVIDIA's tools.

Additional compilation flags for GPU code will depend in large part on which GPU-programming paradigm is being used (e.g., OpenACC, OpenMP, CUDA) and which compiler collection you have loaded. The following examples show basic usage, but note that many customizations and optimizations are possible. You are encouraged to read the relevant man page for the compiler you choose.

OpenACC

To compile with OpenACC directives, simply add the **-acc** flag to your invocation of nvc, nvc++, or nvfortran. A Fortran example:

```
nvfortran -o acc_bin -acc acc_code.f90
```

You can gather more insight into GPU acceleration decisions made by the compiler by adding **-Minfo=accel** to your invocation. Using compiler options, you can also specify which GPU architecture to target. This example will request compilation for both V100 (as on Casper) and A100 GPUs (as on Gust):

```
nvfortran -o acc_bin -acc -gpu=cc70,cc80 acc_code.f90
```

Specifying multiple acceleration targets will increase the size of the binary and the time it takes to compile the code.

OpenMP

Using OpenMP to offload code to the GPU is very similar to using OpenACC. To compile a code with OpenMP offloading, use the **-mp=gpu** flag. The aforementioned diagnostic and target flags also apply to OpenMP offloading.

```
nvfortran -o omp_gpu -mp=gpu omp.f90
```

CUDA

The process for compiling CUDA code depends on whether you are using C++ or Fortran. For C++, the process often involves multiple stages in which you first use **nvcc**, the NVIDIA CUDA compiler, and then your C++ compiler of choice.

```
nvcc -c -arch=sm_80 cuda_code.cu  
g++ -o cuda_bin -lcuda -lcudart main.cpp cuda_code.o
```

Using the **nvcc** compiler driver with a non-NVIDIA C++ compiler requires loading a **cuda** environment module in addition to the compiler of choice.

The compiler handles CUDA code directly, so the compiler you use must support CUDA. This means you should use **nvfortran**. If your source code file ends with the **.cuf** extension, nvfortran will enable CUDA automatically. Otherwise, you can specify the **-Mcuda** flag to the compiler.

```
nvfortran -Mcuda -o cf_bin cf_code.f90
```

Running compute jobs

PBS Pro is used to schedule both interactive and batch compute jobs on Gust. Tasks that are more resource-intensive than coding, compilation, and simple analysis should be executed on the Gust compute nodes rather than on login nodes.

Most jobs are submitted to the “main” *submission* queue, which then routes them to either a CPU or GPU *execution* queue, depending on which resources you request.

Submissions to the “main” queue can request up to 8 CPU nodes and 2 hours of wallclock time. (This can be increased to 12 hours upon request with justification. To request an increase, contact your ASD project’s CSG consultant or submit a request via the [NCAR Research Computing help desk](#).)

CPU-only nodes are scheduled exclusively, meaning a single job from a single user will have exclusive access to the entire node. Meanwhile, the smallest GPU unit you can request is a single A100 GPU. Since there are four (4) A100s per GPU node, your job may share resources on a GPU node depending on your resource requests.

Charging (for Gust only)

Job are charged for used walltime as follows:

CPU jobs	GPU jobs
Elapsed walltime x CPU cores used	Elapsed walltime x GPUs used

Any submission to Gust that requests GPU resources will use GPU core-hours on your project. If no GPUs are requested, your CPU allocation will be used instead.

To check on the status of your allocation for both types of resources, see [sam.ucar.edu](#).

Interactive jobs

Run the **qinteractive** command to start an interactive job. Invoking it without an argument will start an interactive shell on the first available CPU node. The default submission will provide you with a single exclusive CPU node in the “main” queue with 1 hour of wallclock time.

To use another type of node, include a select statement specifying the resources you need. The qinteractive command accepts all PBS flags and resource specifications as detailed by **man qsub**. Some common requests include:

- -A project_code (defaults to PBS_ACCOUNT value that you set in your startup file)
- -l walltime=HH:MM:SS (defaults to 1 hour)
- -l select=1:ncpus=#:mpiprocs=#:ompthreads=#:mem=#GB:ngpus=#

Additionally, qinteractive provides some convenience flags to simplify requesting job resources. If you specify a select statement as described above, it will take precedence over these flags.

- --nchunks=# – number of chunks specified in a select statement (e.g., select=<numchunks>)
- --ntasks=# – number of MPI tasks assigned to each chunk (mpiprocs)
- --nthreads=# – number of SMP threads assigned to each chunk (ompthreads)
- --mem=#GB – amount of memory in gigabytes assigned to each chunk
- --ngpus=# – number of GPUs assigned to each chunk

The following qinteractive invocations, the first with a select statement and the second with a memory flag, are equivalent.

```
qinteractive -A project_code -l select=1:ncpus=1:mem=20GB
qinteractive -A project_code --mem=20GB
```

Batch jobs

For batch jobs, PBS allocates a pool of requested resources and executes a script using those resources. A batch job consists of three primary components:

- the interpreter to be used (typically a shell or programming application like Python),

- directives to PBS specifying the desired compute resources,
- the commands to be executed by the interpreter.

Prepare a batch script by following one of the examples below.

By default, jobs will not import any settings (e.g., environment variables, loaded modules) from your submission environment on the login node. The job will initialize a clean environment using settings from your startup files (`~/.profile` for bash users and `~/.tcshrc` for tcsh users). Any additional configuration should be performed within the job script itself, as this maximizes the portability and reproducibility of the job.

Caution: Avoid using the PBS `-V` option to propagate your environment settings to the batch job; it can cause odd behaviors and job failures when submitting jobs across systems (e.g., from Gust to Casper).

When your job script is ready, use `qsub` to submit it from the login node. PBS settings defined via batch script directives can be overridden using `qsub` command-line arguments. Here is an example in which we define the job at submission time:

```
qsub -A project_code job_script.pbs
```

Submitting jobs with dependencies

It is possible to schedule jobs to run based on the status of other jobs. For example, you might schedule a preprocessing job to run; start a computation job when the preprocessing job is complete; then start a post-processing job when the computation is done. Jobs can be depended on jobs running on other systems (e.g., a Casper job that would only run after a Gust job succeeds).

Let's say you have you have three scripts to submit and run consecutively:

1. `pre.pbs`: a preprocessing job on Casper
2. `main.pbs`: a computation job on Gust
3. `post.pbs`: a post-processing job on Casper

The main job can be run only when the preprocessing job finishes, and the post-processing job can be run only when the computation job finishes. These conditions are assigned using the `"-W depend="` argument to `qsub`. Many conditions are possible, but the most commonly used is *afterok*, which means the specified job must successfully complete before the submitted script will start.

In the following examples, job IDs are stored in environment variables, which are subsequently used to construct the subsequent job's dependency condition.

For bash users

```
JID1=$(qsub -q casper@casper-pbs pre.pbs)
JID2=$(qsub -q main@gusched01 -W depend=afterok:$JID1 main.pbs)
qsub -q casper@casper-pbs -W depend=afterok:$JID2 post.pbs
```

For tcsh users (commands are surrounded by *backticks*, not single quotes)

```
set JID1=`qsub -q casper@casper-pbs pre.pbs`
set JID2=`qsub -q main@gusched01 -W depend=afterok:$JID1 main.pbs`
qsub -q casper@casper-pbs -W depend=afterok:$JID2 post.pbs
```

Jobs are assigned a "held" status by PBS while waiting on a dependency condition to be satisfied.

Specifying job priority

Gust job priority levels can be assigned using a special job resource. Priority determines both the relative importance of the job in the queue as well as the amount charged to the allocation for each compute-hour used. By default, all submissions are assigned "regular" priority. All priority levels are shown in the following table.

Priority	Factor	Description
economy	0.7x	Use if turnaround time is not important and you want to maximize your allocation.
regular	1.0x	Default priority; use for most submissions.
premium	1.5x	Use for jobs you need to run as quickly as possible and don't mind using more allocation.

To change your job's priority, use the `job_priority=` option as either a batch directive or command-line argument to `qsub` or `qinteractive`:

```
#PBS -l job_priority=premium
OR
qinteractive -l job_priority=premium
```

Querying system and job status

Check system status: To see the status of all compute nodes on the system – including used CPUs, GPUs, and memory – as well as a list of jobs running on each node, run the following:

```
pbsnodes -aSJ
```

Check active jobs: Query active PBS jobs (either queued or running) with the **qstat** command. The output from PBS is cached every 10 seconds to improve scheduler performance, so there can be a slight delay in seeing the latest job status. If you provide no arguments to qstat, you will see a default view of all active jobs on Gust. Some useful arguments to qstat include:

qstat jobID	Show a summary of a specified job.
qstat -f jobID	Show verbose information for a specified job.
qstat queue_name	Show all active jobs in a specified queue.
qstat -u \$USER	Show all of your own active jobs.

Check completed jobs: Query historical job information for completed jobs by running the **qhist** command. By default, qhist will show all finished jobs from the current calendar day. Here are some customizations that alter what information is provided:

qhist -u \$USER	Show all of your own finished jobs.
qhist -p YYYYMMDD-YYYYMMDD	Show all finished jobs during the specified period.
qhist -d 5	Show all finished jobs during the past 5 days

Peer scheduling between Gust and Casper

In development.

Using NVIDIA MPS in GPU jobs

Some workflows benefit from processing more than one CUDA kernel on a GPU concurrently, as a single kernel is not sufficient to keep the GPU fully utilized. NVIDIA's Multi-Process Service (MPS) enables this capability on modern NVIDIA GPUs like the A100s on Gust.

Consider using MPS when you are requesting more MPI tasks than physical GPUs. Particularly for jobs with large problem sizes, using multiple MPI tasks with MPS active can sometimes offer a performance boost over using a single task per GPU.

The PBS job scheduler provides MPS support via a chunk-level resource. When you request MPS, PBS will perform the following steps on each specified chunk:

1. Launch the MPS control daemon on each job node.
2. Start the MPS server on each node.
3. Run your GPU application.
4. Terminate the MPS server and daemon.

To enable MPS on job hosts, add **mps=1** to your select statement chunks as follows:

```
#PBS -l select=1:ncpus=8:mpiprocs=8:mem=60GB:ngpus=1:mps=1
```

On each A100 GPU, you may use MPI to launch up to 48 CUDA contexts (GPU kernels launched by MPI tasks) when using MPS. MPS can be used with OpenACC and OpenMP offload codes as well, as the compiler generates CUDA code from your directives at compile time.

Running DDT and MAP jobs

The Arm Forge tools DDT and MAP are provided for debugging, profiling, and optimizing code.

Follow the recommended procedures below to configure the client interface on your local machine, then start your debugging and profiling jobs. (The tools also run from the Gust command line interface.)

Preparing your code

CPU code: Use the **-g** option when you compile your code before debugging or profiling

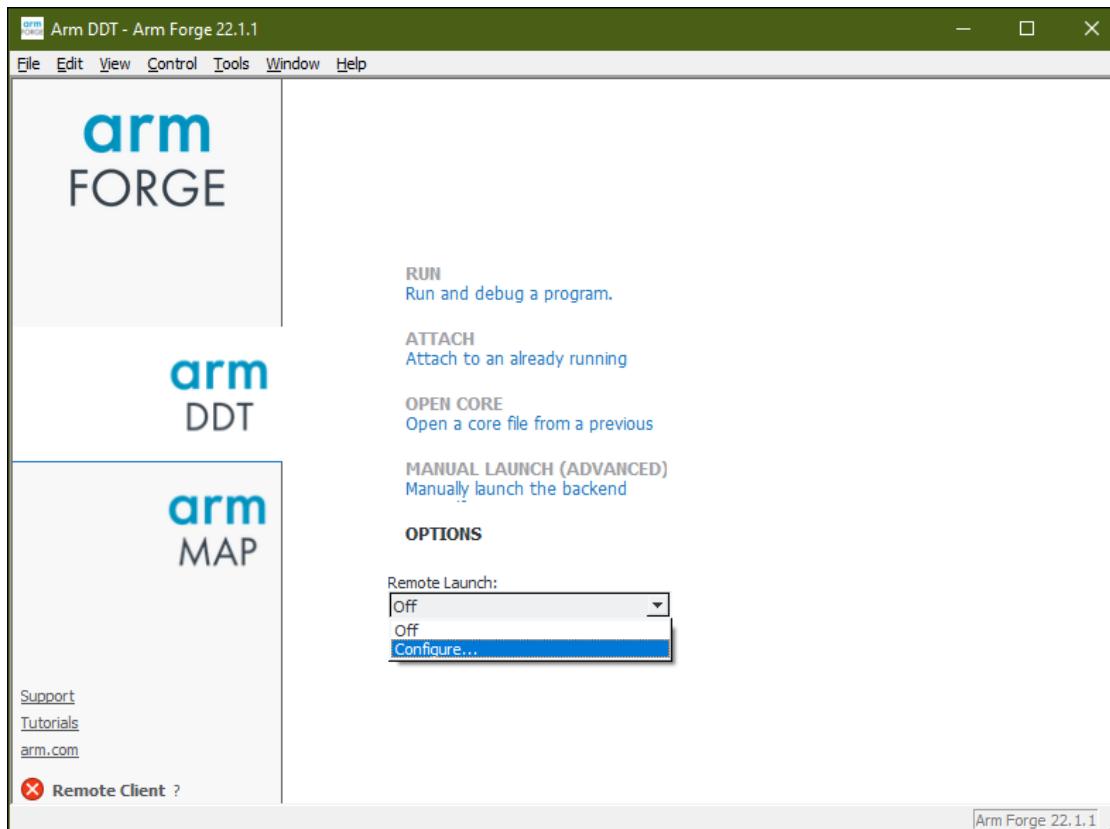
CUDA code: Include both the **-g** and **-G** options for the NVIDIA compilers to debug GPU code.

Do not move or remove the source code, binary, or executable files from the directory or directories in which you compiled them.

Client interface setup

The client software version that you use locally and the server version that you use on Gust must be the same. We recommend using the latest version available. Run **module av arm-forge** to identify the latest version.

- Download the client software from the [Arm site](#).
- Install and start the client on your local machine.
- From the Remote Launch menu (see image), select *Configure*.

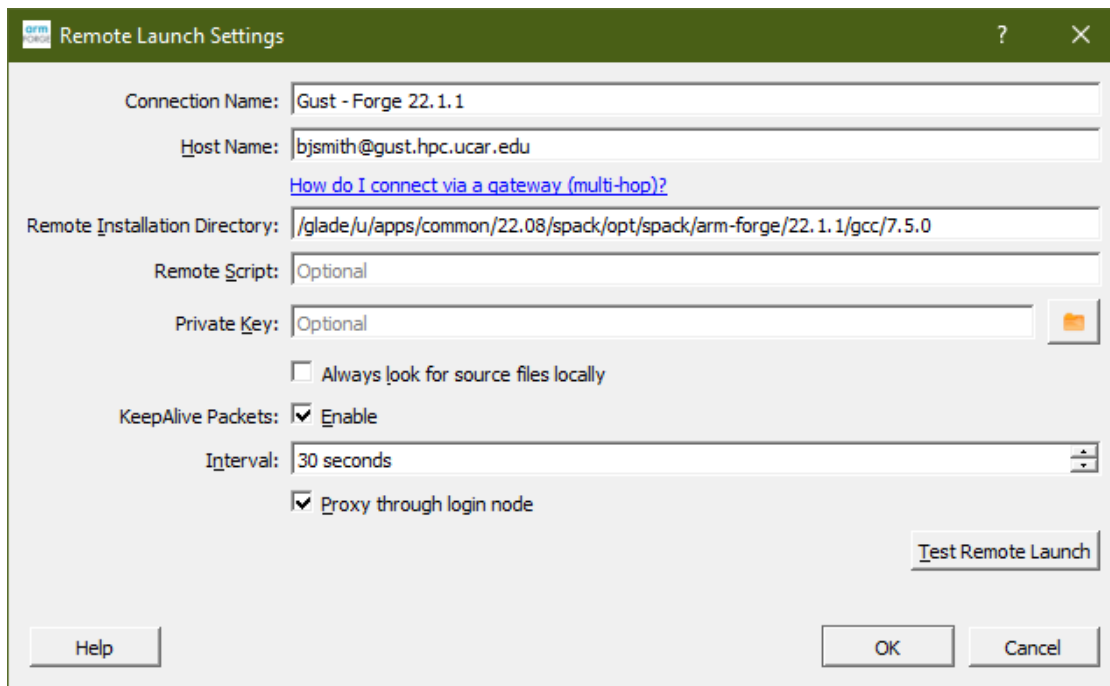


Configure as shown in the following image. The configuration will apply to both DDT and MAP, so you only need to do it once.

- Enter your username followed by @ and the connection name (gust.hpc.ucar.edu) in the "Host Name" field.
- Then, fill in the "Remote Installation Directory" field. You can copy the following text and change the version number to match the version you are using.

```
/glade/u/apps/common/22.08/spack/opt/spack/arm-forge/22.1.1/gcc/7.5.0
```

- Click OK when you're done.

A screenshot of the 'Remote Launch Settings' dialog box. The dialog has a title bar with the 'arm forge' logo, a question mark, and a close button. It contains several input fields: 'Connection Name' with 'Gust - Forge 22.1.1', 'Host Name' with 'bjsmith@gust.hpc.ucar.edu' and a link 'How do I connect via a gateway (multi-hop)?', 'Remote Installation Directory' with '/glade/u/apps/common/22.08/spack/opt/spack/arm-forge/22.1.1/gcc/7.5.0', 'Remote Script' with 'Optional', and 'Private Key' with 'Optional' and a file icon. There are checkboxes for 'Always look for source files locally' (unchecked), 'KeepAlive Packets: Enable' (checked), and 'Proxy through login node' (checked). An 'Interval' spinner is set to '30 seconds'. A 'Test Remote Launch' button is on the right. At the bottom are 'Help', 'OK', and 'Cancel' buttons.

Remote Launch Settings

Connection Name: Gust - Forge 22.1.1

Host Name: bjsmith@gust.hpc.ucar.edu
[How do I connect via a gateway \(multi-hop\)?](#)

Remote Installation Directory: /glade/u/apps/common/22.08/spack/opt/spack/arm-forge/22.1.1/gcc/7.5.0

Remote Script: Optional

Private Key: Optional

☐ Always look for source files locally

KeepAlive Packets: ☒ Enable

Interval: 30 seconds

☒ Proxy through login node

Test Remote Launch

Help OK Cancel

Running a script

Prepare a job script. Specify the "main" *submission* queue and customize the script with your own project code, job name, and so on.

On the last line of your script, use **ddt --connect** (or **map --connect**) instead of mpiexec.

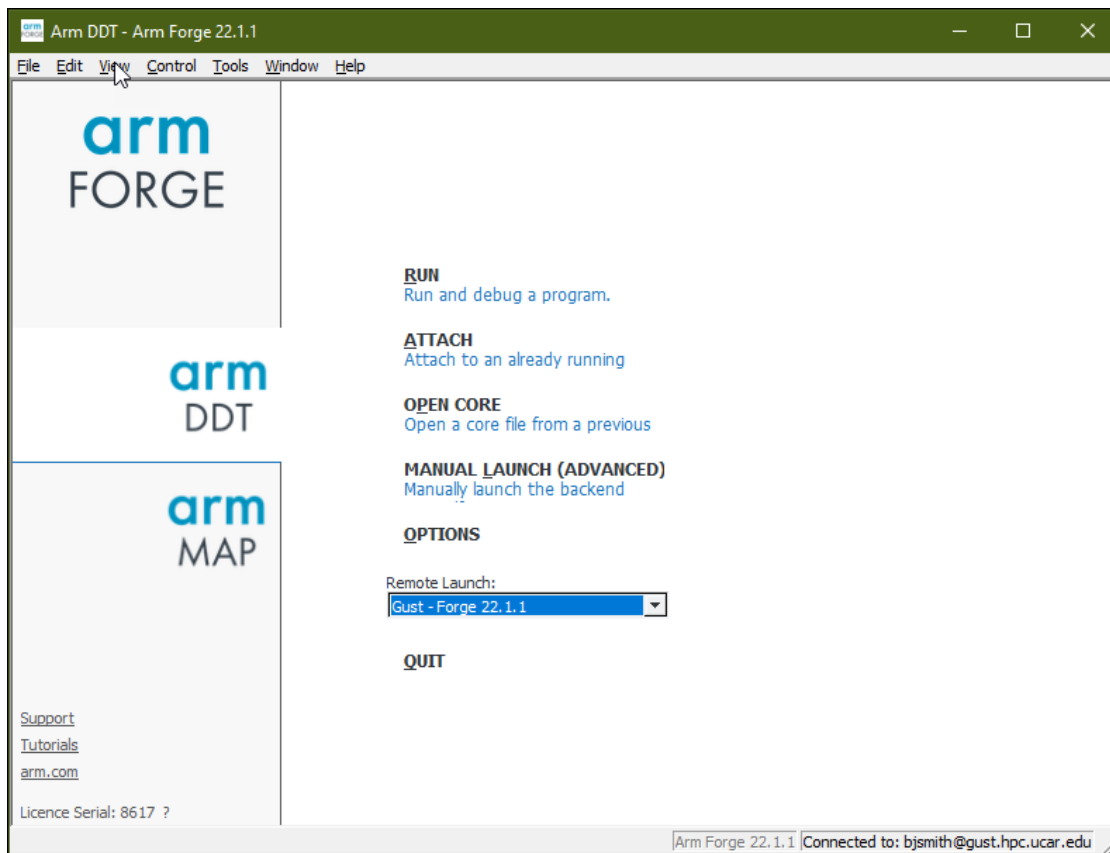
```
ddt --connect ./my_executable
```

Submit your job when indicated below.

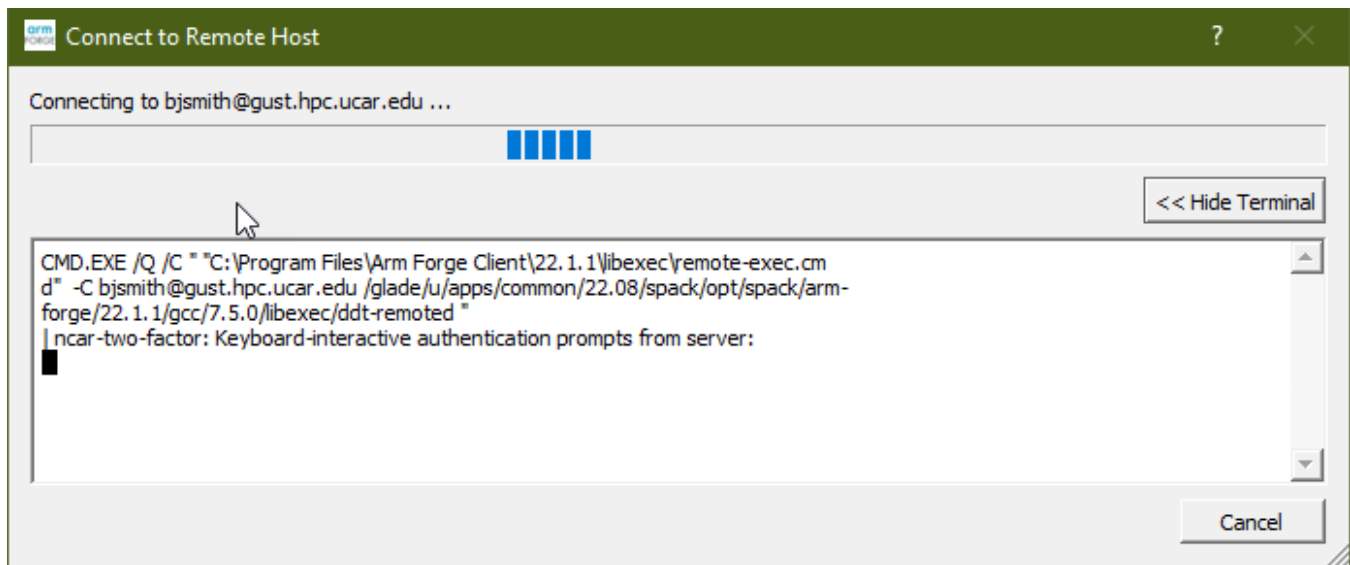
Procedure

Start the client interface on your local machine.

From the Remote Launch menu, select the name of the host configuration you created in the previous step.



When the following dialog box appears, authenticate as usual. (It may be necessary to click *Show Terminal* to see the authentication window).



After you log in, return to your normal terminal window and load the modules you need. Note that in addition to the **arm-forge** module, you will need to load the **cray-cti** module to debug MPI programs. (We recommend including module load commands in your job scripts.)

```
module load arm-forge/22.1.1 cray-cti
```

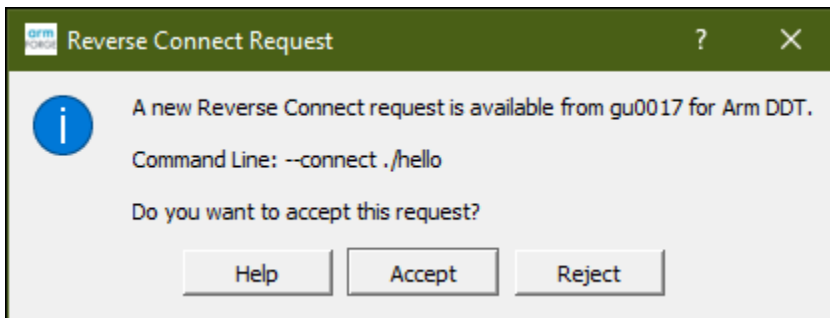
To debug GPU jobs, you may need to set the environment variable `CUDA_DEBUGGER_SOFTWARE_PREEMPTION` to 1

```
export CUDA_DEBUGGER_SOFTWARE_PREEMPTION=1
```

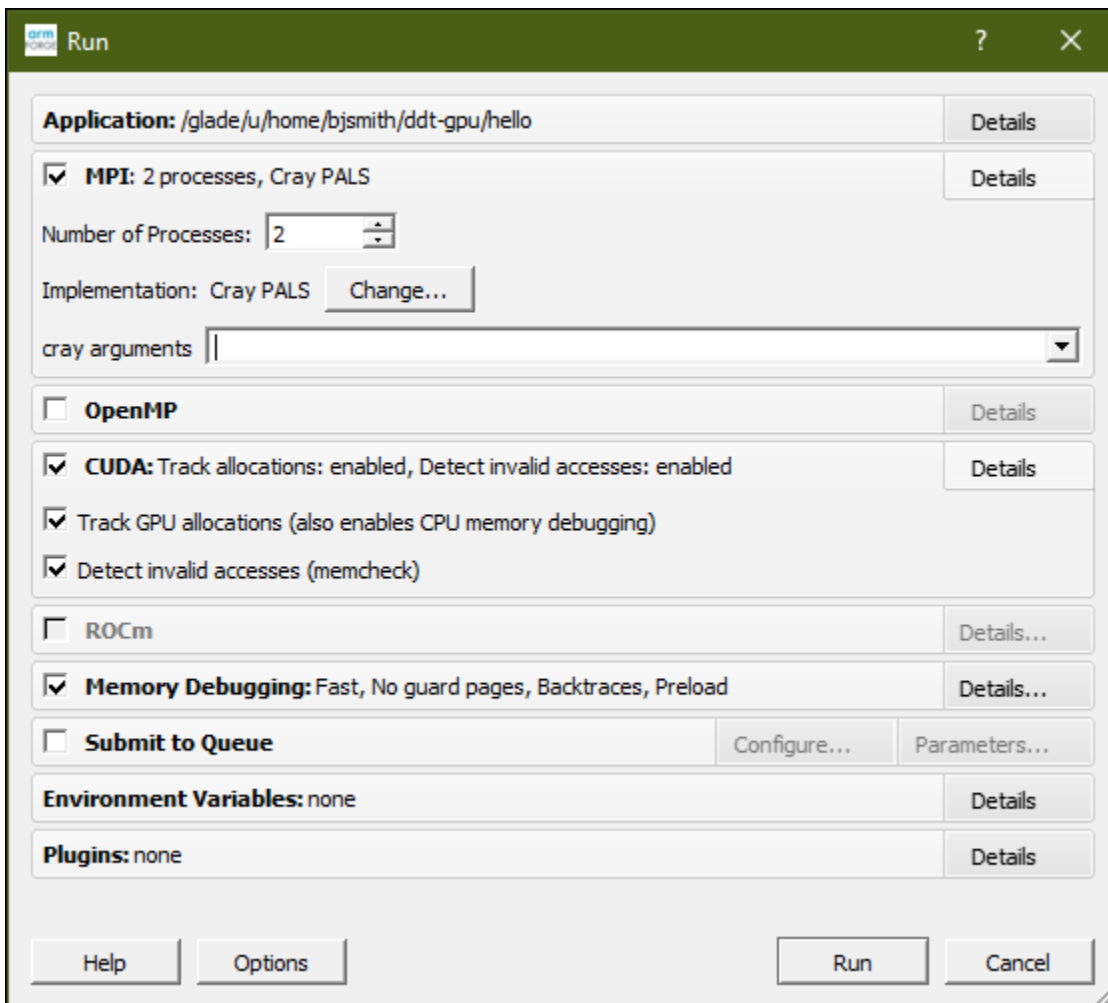
Submit your job script on your command line as in this example:

```
qsub my-debug-script.bash
```

When your job starts, the GUI will show that a *Reverse Connect request* has been made. Accept the request to continue.



A "Run" window will open and display settings imported from your job script. Review the settings. If your program uses Cray MPICH, specify the MPI implementation as Cray PALS where shown in the following image.



Click *Run* after reviewing the settings, and the DDT or MAP window will open.

Quit when you're finished so the license is available to other users.

Batch job script examples

Running a hybrid CPU program with MPI and OpenMP

In this example, we run a hybrid application that uses both MPI tasks and OpenMP threads. The executable was compiled using default modules (Cray compilers and MPI). We use a single MPI process for each CPU and 64 OpenMP threads per MPI process. Whenever you run a program that is thread-aware, it is important to provide a value for `ompthreads`; PBS will use that value to define the **OMP_NUM_THREADS** environment variable.

```
#!/bin/bash
#PBS -A project_code
#PBS -N hybrid_job
#PBS -q main@gusched01
#PBS -l walltime=01:00:00
#PBS -l select=2:ncpus=128:mpiprocs=32:ompthreads=4

# Use scratch for temporary files to avoid space limits in /tmp
export TMPDIR=/glade/scratch/$USER/temp
mkdir -p $TMPDIR

# Run application using the cray-mpich MPI
mpiexec --cpu-bind depth -n 64 -ppn 32 -d 4 ./executable_name
```

Running an MPI-enabled GPU application

In this example, we run a CUDA program that also uses MPI. The application was compiled using the NVIDIA HPC SDK compilers, the CUDA toolkit, and cray-mpich MPI. We request all GPUs on each node.

Please ensure that you have the **cuda** module loaded as shown below when attempting to run GPU applications or nodes may lock up and become unresponsive.

```
#!/bin/bash
#PBS -A project_code
#PBS -N gpu_job
#PBS -q main@gusched01
#PBS -l walltime=01:00:00
#PBS -l select=2:ncpus=64:mpiprocs=4:ngpus=4

# Use scratch for temporary files to avoid space limits in /tmp
export TMPDIR=/glade/scratch/$USER/temp
mkdir -p $TMPDIR

# Load modules to match compile-time environment
module purge
module load nvhpc cuda cray-mpich

# (Optional: Enable GPU managed memory if required.)
# From 'man mpi': This setting will allow MPI to properly
# handle unify memory addresses. This setting has performance
# penalties as MPICH will perform buffer query on each buffer
# that is handled by MPI)
# If you see runtime errors like
# (GTL DEBUG: 0) cuIpcGetMemHandle: invalid argument,
# CUDA_ERROR_INVALID_VALUE
# make sure this variable is set
export MPICH_GPU_MANAGED_MEMORY_SUPPORT_ENABLED=1

# Run application using the cray-mpich MPI
# The 'get_local_rank' command is a script that sets several GPU-
# related environment variables to allow MPI-enabled GPU
# applications to run. The get_local_rank script is detailed
# in the binding section below, and is also made available
# via the ncarenv module.
mpiexec -n 8 -ppn 4 get_local_rank ./executable_name
```

Binding MPI ranks to CPU cores and GPU devices

For some GPU applications, you may need to explicitly control the mapping between MPI ranks and GPU devices (see `man mpi`). One approach is to manually control the `CUDA_VISIBLE_DEVICES` environment variable so a given MPI rank only “sees” a subset of the GPU devices on a node. Consider the following shell script:

get_local_rank

```
#!/bin/bash

export MPICH_GPU_SUPPORT_ENABLED=1
export LOCAL_RANK=${PMI_LOCAL_RANK}
export GLOBAL_RANK=${PMI_RANK}
export CUDA_VISIBLE_DEVICES=$(expr ${LOCAL_RANK} % 4)

echo "Global Rank ${GLOBAL_RANK} / Local Rank ${LOCAL_RANK} / CUDA_VISIBLE_DEVICES=${CUDA_VISIBLE_DEVICES} / $(hostname)"

exec $*
```

It can be used underneath **mpiexec** to bind an MPI process to a particular GPU:

```
#PBS -l select=2:ncpus=64:mpiprocs=4:ngpus=4
...
# Run application using the cray-mpich MPI, binding the local
# mpi rank [0-3] to corresponding GPU index [0-3]:
mpiexec -n 8 -ppn 4 ./get_local_rank ./executable_name
```

The command above will launch a total of 8 MPI ranks across 2 nodes, using 4 MPI ranks per node, and each rank will have dedicated access to one of the 4 GPUs on the node. Again, see **man mpi** for other examples and scenarios.

Binding MPI ranks to CPU cores can also be an important performance consideration for GPU-enabled codes, and can be done with the **--cpu-bind** option to **mpiexec**. For the above example using 2 nodes, 4 MPI ranks per node, and 1 GPU per MPI rank, binding each of the MPI ranks to one of the four separate NUMA domains within a node is likely to be optimal for performance. This could be done as follows:

```
mpiexec -n 8 -ppn 4 --cpu-bind verbose,list:0:16:32:48 ./get_local_rank ./executable_name
```