# Batch job script examples - Derecho

When using these examples to create your own job scripts to run on Derecho, remember to substitute your own job name and project code, and customize the other directives and commands as necessary. That includes the commands shown for setting your **TMPDIR** environment variable as recommended here: Storing temporary files with TMPDIR.

### Page contents



---

## 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 (Intel compilers and MPI). We use a 2 nodes with 32 MPI ranks on each node and 4 OpenMP threads per MPI rank. Whenever you run a program that compiled with OpenMP support, it is important to provide a value for ompthreads in the select statement; 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
#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

# Load modules to match compile-time environment
module purge
module load ncarenv/22.12 oneapi/2022.2.1 craype/2.7.19 cray-mpich/8.1.21

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

---

## Running an MPI-enabled GPU application

In this example, we run an MPI CUDA program. The application was compiled using the NVIDIA HPC SDK compilers, the CUDA toolkit, and cray-mpich MPI. We request all four GPUs on each of two nodes.

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
#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
```