Multi-GPU Programming, Part 2

By: Daniel Howard dhoward@ucar.edu (mailto:dhoward@ucar.edu), Consulting Services Group, CISL & NCAR

Date: July 14th, 2022

In this notebook we explore the mini-app MiniWeather (https://github.com/mrnorman/miniWeather) to present techniques and code examples for implementing and assessing performance of various multi-GPU paradigms. We will cover:

1. Interoperability of OpenACC with MPI and NCCL GPU communication libraries
2. Hands-on implementation of MiniWeather with CUDA aware MPI and NCCL

Head to the NCAR JupyterHub portal (https://jupyterhub.hpc.ucar.edu/stable) and **start a JupyterHub session on Casper login** (or batch nodes using 1 CPU, no GPUs) and open the notebook at `11_MultiGPU/11_multiGPU_Part2.ipynb` . Be sure to clone (if needed) and update/pull the NCAR GPU_workshop directory.

```
# Use the JupyterHub GitHub GUI on the left panel or the below shell commands
git clone git@github.com:NCAR/GPU_workshop.git
git pull
```

Workshop Etiquette

- Please mute yourself and turn off video during the session.
- Questions may be submitted in the chat and will be answered when appropriate. You may also raise your hand, unmute, and ask questions during Q&A at the end of the presentation.
- By participating, you are agreeing to UCAR's Code of Conduct (https://www.ucar.edu/who-we-are/ethics-integrity/codes-conduct/participants)
- Recordings & other material will be archived & shared publicly.
- Feel free to follow up with the GPU workshop team via Slack or submit support requests to rchelp.ucar.edu (https://support.ucar.edu)
    - Office Hours: Asynchronous support via Slack (https://ncargpuusers.slack.com) or schedule a time with an organizer

Notebook Setup

Set the `PROJECT` code to a currently active project, ie `UCIS0004` for the GPU workshop, and `QUEUE` to the appropriate routing queue depending on if during a live workshop session (`gpuworkshop`), during weekday 8am to 5:30pm MT (`gpudev`), or all other times (`casper`).

The `GPU_TYPE=gp100` nodes are not configured for multi-GPU computing! Thus, the `gpuworkshop` queue is not useful for this session. Saying as much, please set `GPU_TYPE=v100` and use the `gpudev` both during the workshop and for independent work. See [Casper queue documentation (https://arc.ucar.edu/knowledge_base/72581396#StartingCasperjobswithPBS-Concurrentresourcelimits)](https://arc.ucar.edu/knowledge_base/72581396#StartingCasperjobswithPBS-Concurrentresourcelimits) for more info.

What Have We Learned So Far?

In Part 1 (Multi-GPU_Programming_for_Earth_Scientists_Jiri_Kraus_NVIDIA.pdf), Jiri Kraus from NVIDIA shared many different approaches for Multi-GPU Programming.

- Non-CUDA Aware MPI
- CUDA Aware MPI
- NCCL (pronounced "nickel") - NVIDIA Collective Communication Library
- NVSHMEM - NVIDIA Shared Memory Library
- Other tips and backgrond information

Today, we will focus on hands-on implementation of **CUDA Aware MPI** and **NCCL** within MiniWeather.

Baseline Performance of Non-CUDA Aware MPI

First, let's compile our baseline program `miniWeather_mpi_openacc.F90`. To note, **I/O output has been disabled in all versions**. Also, this version has already been partially modified to offload each MPI task to a distinct GPU.

Casper does not attempt to isolate GPUs between MPI tasks like other HPC centers may choose to do by default. On Casper, **every MPI task can access all GPUs available to the node** it is residing on.

Question: If this baseline code was not modified, why does MiniWeather's performance not improve with increasing the number of MPI tasks that reside on the same node?

Hint: Which GPU device is each MPI task using?

In [ ]:

```
# Compiles the CUDA aware MPI version of MiniWeather using OpenACC
OPENACC_FLAGS="-acc -gpu=cc70,lineinfo"

mpif90 -I${PNETCDF_INC} -Mextend -O0 -DNO_INFORM -c miniWeather_mpi_openacc.F90 -o
miniWeather_mpi_openacc.F90.o \
-D_NX=4096 -D_NZ=2048 -D_SIM_TIME=10.0 -D_OUT_FREQ=10.0 -D_DATA_SPEC=DATA_SPEC_THE
RMAL ${OPENACC_FLAGS}

mpif90 -Mextend -O3 miniWeather_mpi_openacc.F90.o -o openacc -L${PNETCDF_LIB} -lpn
etcdf ${OPENACC_FLAGS}
rm -f miniWeather_mpi_openacc.F90.o
```

Now, submit multi-GPU runs to get some performance benchmarks. We use a validation script to ensure the answer is correct. The validation script has the following usage syntax, where `n_tasks` is optional:

```
./check_output.sh executable mass_relative_tolerance
energy_relative_tolerance [n_tasks]
```

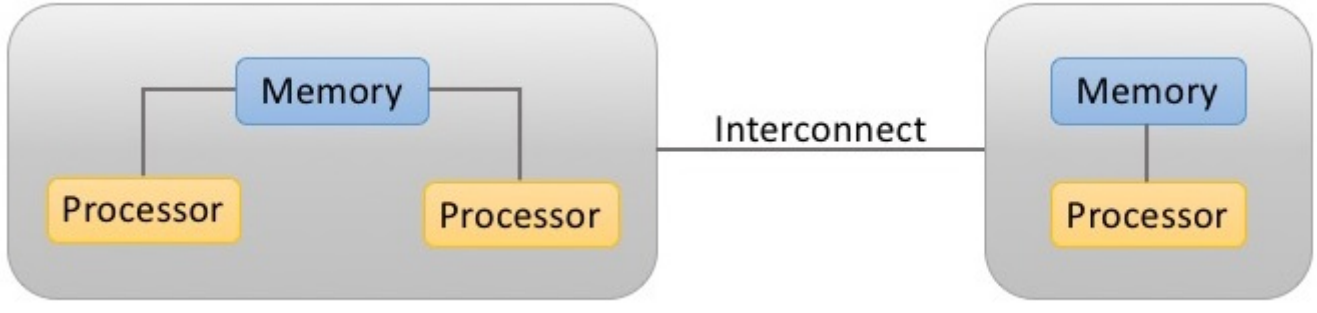
To note, Casper has nodes of 4 or 8 GPUs available and the main `gpgpu` queue allows up to 32 GPUs per job. The `gpudev` queue allows up to 4 GPUs per job.

In [ ]:

```
for S in 1        # Select number of nodes (Casper: gpgpu -> S*N <= 32, gpudev -> S
*N <= 4 )
  do
  for N in 1 2 4   # Number of MPI tasks, CPUs, and GPUs per node (Casper: max(N)=
8, 8-way nodes often busy )
    do
      qcmd -A $PROJECT -q $QUEUE -l select=$S:ncpus=$N:ngpus=$N:mpiprocs=$N -l gpu
_type=$GPU_TYPE -l walltime=30 -- \
      $PWD/check_output.sh $PWD/openacc 1e-13 4.5e-5 $((S*N))
    done
  done
```

Basics of Communication with MPI

The foundation to communicating data and messages for a **distributed-memory** computer is the **Message Passing Interface (MPI)** library. MPI consists of a collection of routines for exchanging data across distributed memory spaces in a parallel program, ie memory from one node to another node or one GPU to another GPU.



The MPI standard was first introduced in 1994 and has evolved many times over the years to a well established level of maturity across multiple library options, such as **OpenMPI**, **Intel MPI**, and **MVAPICH**.

A complete description of the MPI standard can be found on the [MPI Forum's Documentation Page (https://www.mpi-forum.org/docs/)](https://www.mpi-forum.org/docs/).

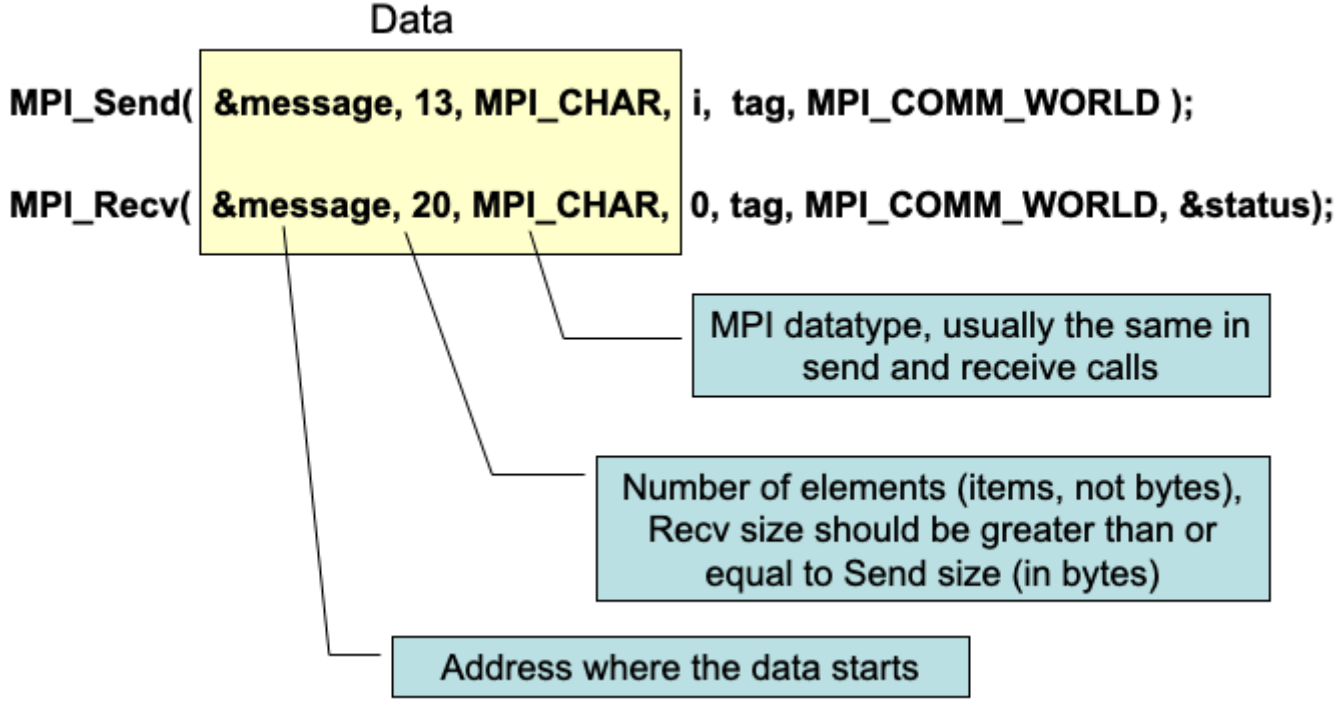
Basic Structure of MPI Programs

1. Initialize communications
    - `MPI_INIT` - initializes the MPI environment
    - `MPI_COMM_SIZE` - returns the number of processes
    - `MPI_COMM_RANK` - returns this process's number (rank)
2. Communicate to share data between processes
    - `MPI_SEND` - sends a blocking message
    - `MPI_RECV` - receives a blocking message
3. Exit from the message-passing system --
    - `MPI_FINALIZE`

There also exists collective operations such as `MPI_Bcast` and `MPI_Allreduce`. A primary concept to understand is **blocking** vs **non-blocking** communication, ie `MPI_Send` vs `MPI_ISend`.

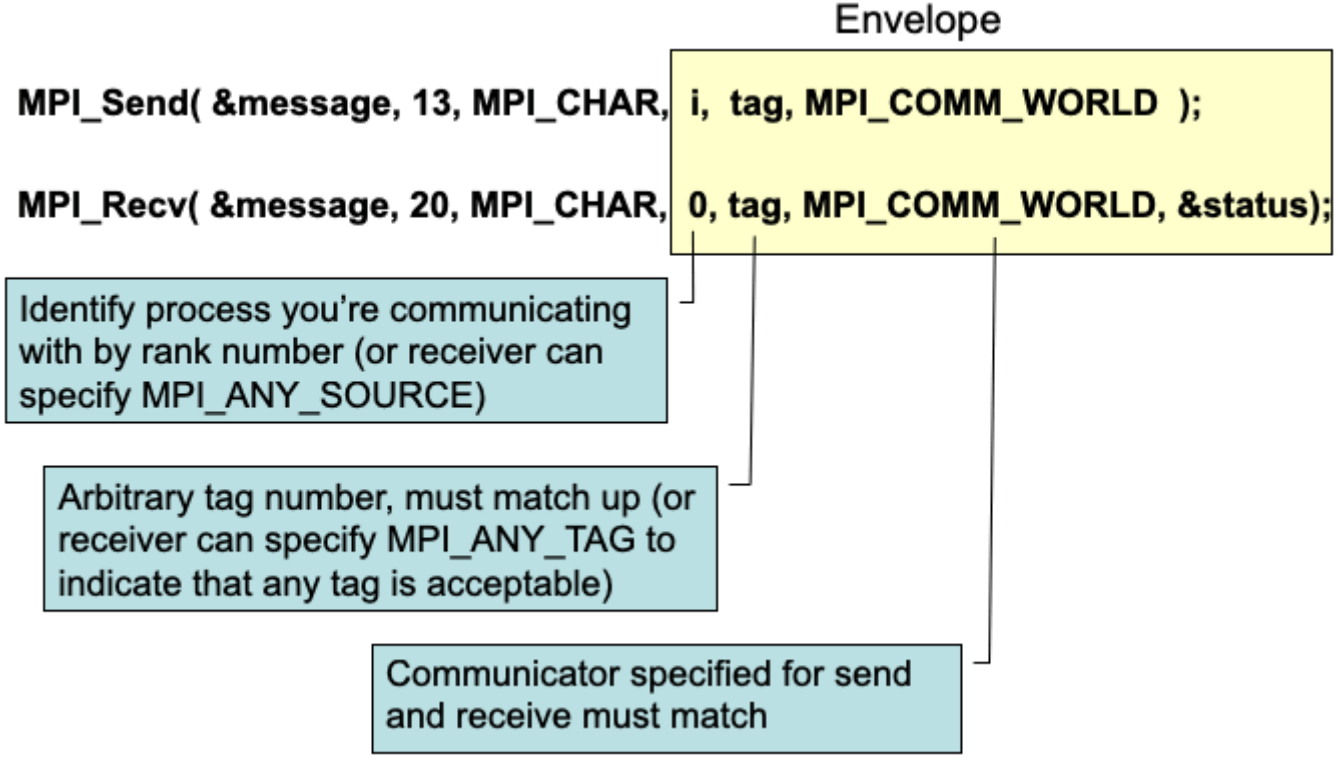
This concept is similar to **synchronous** vs **asynchnronous** operations, discussed in earlier GPU sessions respectively.

Data Parameters - Example `MPI_Send` and `MPI_Recv`



Source: Cornell's [Message Passing Interface Virtual Workshop (https://cvw.cac.cornell.edu/MPI/messages)](https://cvw.cac.cornell.edu/MPI/messages)

Envelope Parameters - Example `MPI_Send` and `MPI_Recv`



Source: Cornell's [Message Passing Interface Virtual Workshop (https://cvw.cac.cornell.edu/MPI/messages)](https://cvw.cac.cornell.edu/MPI/messages)

Extension of MPI to Multi-GPU Communication

There is not time today to go into more details on MPI. Nonetheless, MPI is an important framework to understand and forms the basis of development for similar Multi-GPU communication patterns and developed libraries.

For example, the **NCCL** library utilizes the communication APIs `ncclSend` and `ncclRecv`, which are functionaly equivalent to `MPI_Send` and `MPI_Recv`. Thus, concepts for MPI communication are very useful for understanding related concepts in multi-GPU communication patterns. **The only caveat is the additional separate memory space within the GPU alongside the CPU memory**.

If you are not that familiar with MPI or want to review beginner/advanced concepts, you are encouraged to seek out additional learning material such as:

- Cornell's Virtual Workshop 5-part MPI Series (https://cvw.cac.cornell.edu/topics#MPI)
- NCSA's and UIUC's Introduction to MPI (https://www.hpc-training.org/xsede/moodle/enrol/index.php?id=34) on the hpc-training.org (https://www.hpc-training.org/xsede/moodle/) HPC-Moodle platform
- XSEDE's HPC Workshop: MPI (https://www.psc.edu/resources/training/xsede-hpc-workshop-may-2022-mpi/) May 2022 offering

Using CUDA Aware MPI for Multi-GPU Communication

Fortunately, MiniWeather (https://github.com/mrnorman/miniWeather) already provides an MPI implementation. With OpenACC, calls to MPI had to be surrounded by the directives `!$acc update host()` and `!$acc update device()` (Note: `MPI_ISend` / `MPI_IRecv` are non-blocking MPI calls).

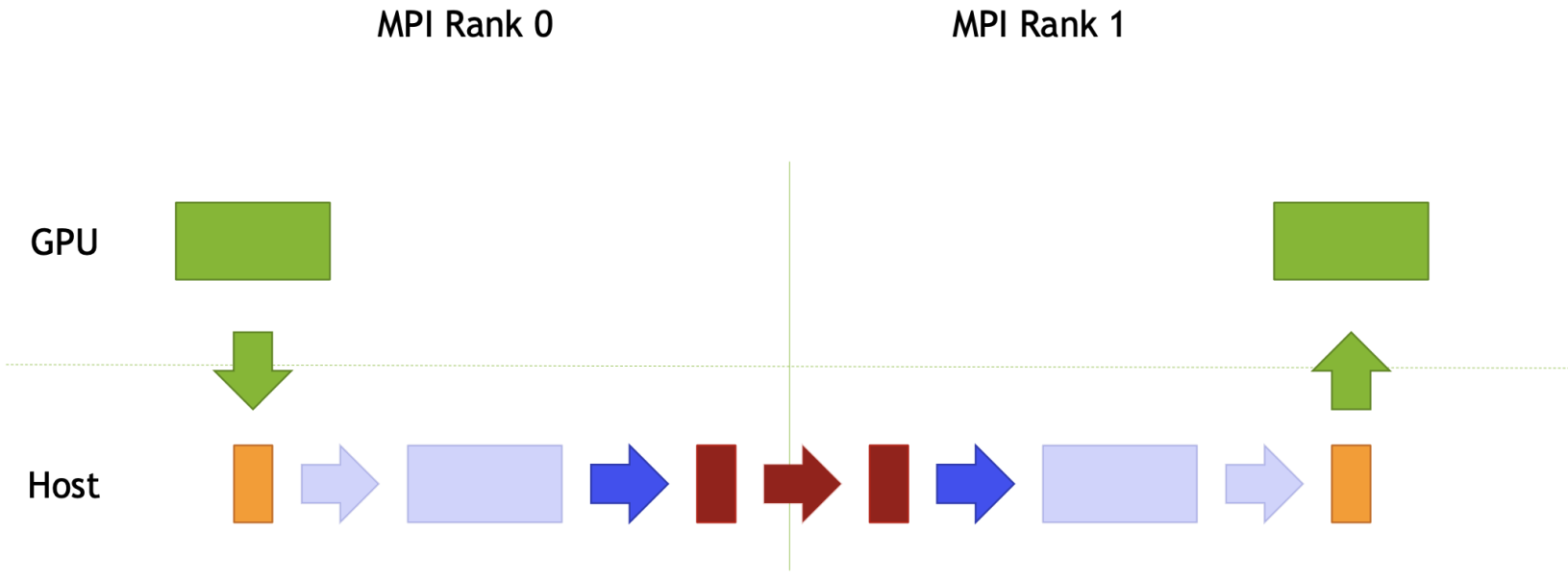
```
!Prepost receives
    call mpi_irecv(recvbuf_l,hs*nz*NUM_VARS,MPI_REAL8, left_rank,0,MPI_COMM_WORLD
,req_r(1),ierr)
    call mpi_irecv(recvbuf_r,hs*nz*NUM_VARS,MPI_REAL8,right_rank,1,MPI_COMM_WORLD
,req_r(2),ierr)

    !OpeanACC GPU Kernel loading send buffers
    !$acc update host(sendbuf_l,sendbuf_r) async
    !$acc wait

    !Fire off the sends
    call mpi_isend(sendbuf_l,hs*nz*NUM_VARS,MPI_REAL8, left_rank,1,MPI_COMM_WORLD
,req_s(1),ierr)
    call mpi_isend(sendbuf_r,hs*nz*NUM_VARS,MPI_REAL8,right_rank,0,MPI_COMM_WORLD
,req_s(2),ierr)
    !Wait for receives to finish
    call mpi_waitall(2,req_r,status,ierr)
    ...
```

This approach works but is not most optimal given available communication routes between nodes.

Recall from the previous session how data must be copied from GPU memory to CPU memory to another node's CPU memory to that node's GPU memory.




```
MPI_Send(s_buf_d, size, MPI_BYTE, 1, tag, MPI_COMM_WORLD);

MPI_Recv(r_buf_d, size, MPI_BYTE, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Some of these memory movement steps can be eliminated with better programming choices.

OpenACC Interoperability and Implemenations for CUDA Aware MPI

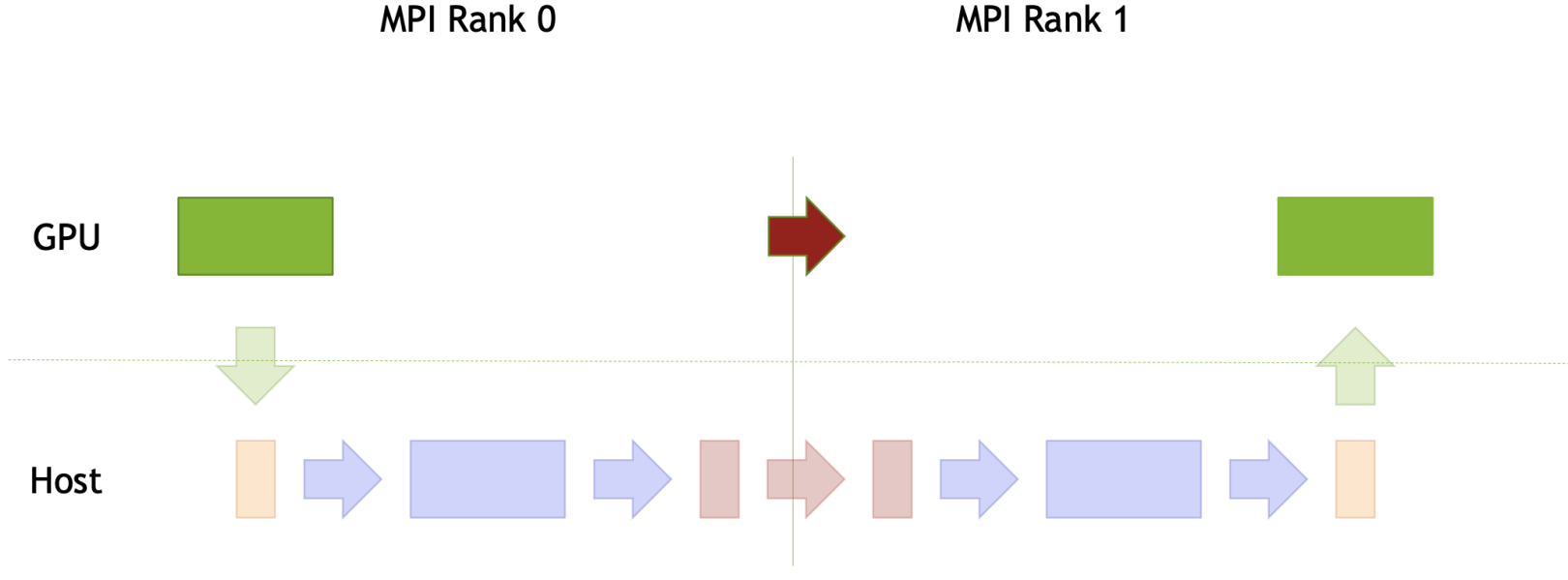
In order to redirect memory movement and avoid unecessary steps between the CPUs and GPUs, **MPI must be provided the GPU device location of data memory**. Here's an example in OpenACC:

```
!$acc host_data use_device(s_buf_d,r_buf_d)
...
!$acc end host_data
```

Essentialy, within any `host_data` on the CPU host, data objects in `use_device()` will no longer reference host/CPU memory but will instead point to device/GPU memory. This simple directive should tightly encapsulate any MPI calls, allowing a CUDA Aware MPI library to directly reference memory on the GPU instead of memory on the CPU. If interested, see more OpenACC interoperability features at [this GitHub (https://github.com/OpenACC/openacc-interoperability-examples)](https://github.com/OpenACC/openacc-interoperability-examples) by Jeff Larkin.

For Casper, the default library `OpenMPI` enables CUDA aware features by default. However, different MPI libraries often require you to specify additional flags or environment variables to use CUDA aware features (slide 22 in [Part 1 (Multi-GPU_Programming_for_Earth_Scientists_Jiri_Kraus_NVIDIA.pdf)](Multi-GPU_Programming_for_Earth_Scientists_Jiri_Kraus_NVIDIA.pdf)).

Once the MPI library directly references GPU device memory when setting up communication, messages communicated across a HPC cluster can more directly travel to their destinations, shortening the time spent in communication.



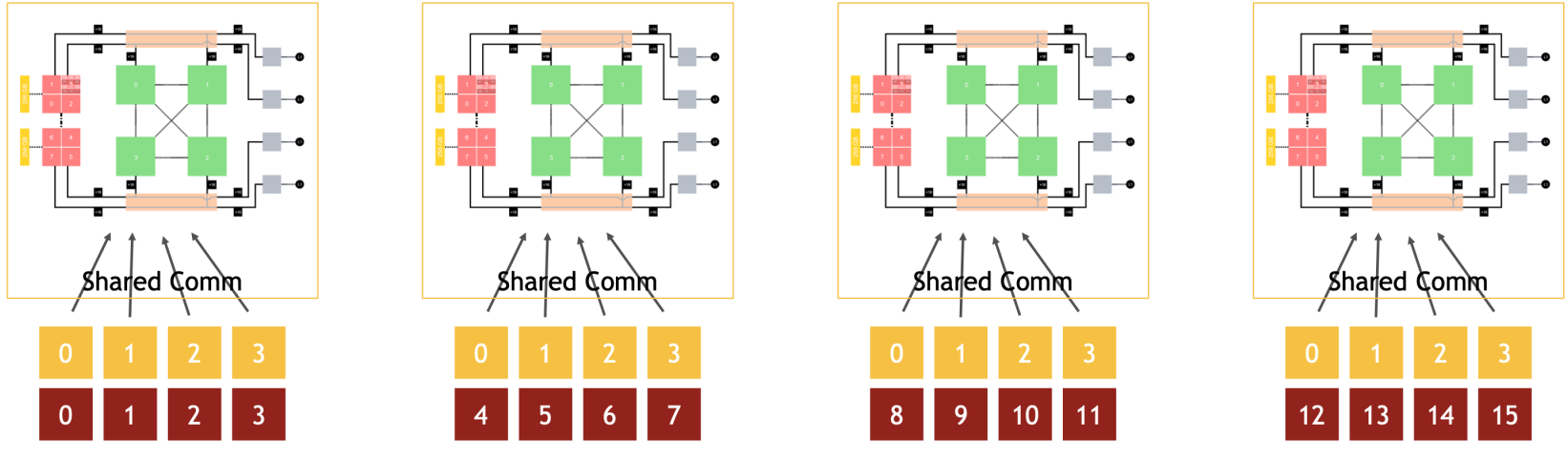

```
MPI_Send(s_buf_d, size, MPI_BYTE, 1, tag, MPI_COMM_WORLD);

MPI_Recv(r_buf_d, size, MPI_BYTE, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Using OpenACC to Assign GPU Devices to MPI Tasks

Parallel programs typically won't automatically know which GPU device it should be assigned to. Managing this is best left to the developer, which allows multiple arrangements, ie ...

- one process per MPI task per GPU
- one process managing multiple GPUs, etc.



A common technique to manage local GPU ranks across global MPI ranks utilizes a split `MPI_COMM_TYPE_SHARED` communicator. This instantiates another MPI communicator that is local to an individual node, indexing a subset of processes across the local node's available MPI tasks.

With a split communicator, you can then use OpenACC Runtime API functions like `acc_get_num_devices()` and `acc_set_device()` (must specify `use openacc`) to assign MPI tasks to specific GPUs given the devices available. An example initialization code snippet is below:

```
call MPI_Init(ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, nranks, ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
    call MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, myrank, MPI_IN
FO_NULL, local_comm, ierr)
    call MPI_Comm_rank(local_comm, local_rank, ierr)

    nGPUs_node = acc_get_num_devices(acc_get_device_type())
    call acc_set_device_num(mod(local_rank,nGPUs_node), acc_get_device_type())
```

EXERCISE - Implement CUDA Aware MPI in `miniWeather_mpiAware_openacc.F90`

Following the direction of the `TODO` sections in miniWeather_mpiAware_openacc.F90 (miniWeather_mpiAware_openacc.F90), create a CUDA aware MPI version of MiniWeather. This will involve the following changes:

1. Line 78 & 80 (miniWeather_mpiAware_openacc.F90#L78) - Instantiate additional variables for the split local communicators and number of GPUs
2. Line 401 & 423 (miniWeather_mpiAware_openacc.F90#L401) - Specify the send and receive buffers in MPI calls to use GPU device memory
3. Line 505 & 525 (miniWeather_mpiAware_openacc.F90#L505) - Add a split communicator and assign GPUs per ranks local to each node

Once finished, use the cells below to compile the CUDA aware version, check for errors, and run it. If you get stuck, solutions are in the solutions (solutions) folder.

In [ ]:

```
# Compiles the CUDA aware MPI version of MiniWeather using OpenACC, exec = ./openaccAware
OPENACC_FLAGS="-acc -gpu=cc70,lineinfo"

mpif90 -I${PNETCDF_INC} -Mextend -O0 -DNO_INFORM -c miniWeather_mpiAware_openacc.F90 -o miniWeather_mpiAware_openacc.F90.o \
-D_NX=4096 -D_NZ=2048 -D_SIM_TIME=10.0 -D_OUT_FREQ=10.0 -D_DATA_SPEC=DATA_SPEC_THERMAL ${OPENACC_FLAGS}

mpif90 -Mextend -O3 miniWeather_mpiAware_openacc.F90.o -o openaccAware -L${PNETCDF_LIB} -lpnetcdf ${OPENACC_FLAGS}
rm -f miniWeather_mpiAware_openacc.F90.o
```

In [ ]:

```
# CUDA Aware MPI runs
S=1; N=4
  qcmd -A $PROJECT -q $QUEUE -l select=$S:ncpus=$N:ngpus=$N:mpiprocs=$N -l gpu_type=$GPU_TYPE -l walltime=30 -- \
  $PWD/check_output.sh $PWD/openaccAware 1e-13 4.5e-5 $((S*N))
```

Using NCCL Library for Multi-GPU Communication

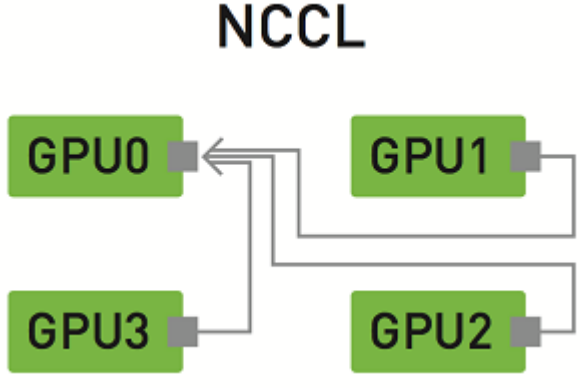
Leveraging the NVIDIA Collective Communication Library (NCCL) with OpenACC is relatively the same as CUDA Aware MPI. Below are some significant points to make:

1. Function prototypes for calling NCCL only slightly differ from MPI functions
2. By default, **all NCCL functions are blocking** within the provided CUDA stream
    - Grouped non-blocking behavior can be achieved with `ncclGroupStart()` and `ncclGroupEnd()` regions
3. **NCCL can directly leverage scheduling and overalapping of communication/compute on the GPU** through effective CUDA stream selection while MPI cannot
4. Error handling is done via returned values from NCCL functions whereas MPI error handling is a passed argument
5. **NCCL arguably should have better collective communication performance** compared to MPI but point-to-point communication will likely not have demonstrable benefit
    - NCCL can achieve improved performance on a well configured system since it automatically can detect and optimize communication across given **node topologies**

Using the NCCL API

- Full documentation for NCCL is here (https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html)
- Listed functional prototypes and API specification is here (https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/api.html)

When using any external library, it is highly encouraged to consult available documentation like the above to learn how to use a library.



In NCCL's case, the same `!$acc host_data use_device()` approach is required alongside OpenACC code (or you can directly pass in a device memory pointer from your own CUDA kernel or other library).

Here are important data and function prototypes for NCCL useful for the next exercise:

1. Data prototypes for initializing NCCL
    - `type(ncclUniqueId) :: nccl_id` - A unique id for each group of communicators
    - `type(ncclResult) :: nccl_result` - A variable to store return values from NCCL functions, used for error handling
    - `type(ncclComm) :: nccl_comm` - The NCCL comunicator object
2. Function prototypes for initializing NCCL
    - `ncclResult_t ncclGetUniqueId(ncclUniqueId* uniqueId)` - Creates a `nccl_id` for the communicators
    - `ncclResult_t ncclCommInitRank(ncclComm_t* comm, int nranks, ncclUniqueId commId, int rank)` - Initializes NCCL, similar to `MPI_Init()`
3. Function prototypes for Send/Recv communication
    - `ncclResult_t ncclSend(const void* sendbuff, size_t count, ncclDataType_t datatype, int peer, ncclComm_t comm, cudaStream_t stream)`
    - `ncclResult_t ncclRecv(void* recvbuff, size_t count, ncclDataType_t datatype, int peer, ncclComm_t comm, cudaStream_t stream)`

To enable runtime NCCL debugging, simply set `NCCL_DEBUG` = `WARN` .

EXERCISE - Implement NCCL Library in `miniWeather_mpiNCCL_openacc.F90`

Extending from the previous exercise, follow the `TODO` sections in miniWeather_mpiNCCL_openacc.F90 (miniWeather_mpiNCCL_openacc.F90) to create a NCCL version of Miniweather. Ideally, make this version portable by using the provided `#ifdef NV_GPU / #endif` preprocessor sections.

1. Line 14 (miniWeather_mpiNCCL_openacc.F90#L14) - Load the NCCL library module
2. Line 93 (miniWeather_mpiNCCL_openacc.F90#L93) - Instantiate NCCL variables needed for communicators
3. Line 425 (miniWeather_mpiNCCL_openacc.F90#L425) - Setup a non-blocking group of sends and receives similar to the MPI sends and receives, also using OpenACC to reference GPU memory
4. Line 560 (miniWeather_mpiNCCL_openacc.F90#L560) - Generate NCCL unique Id and use `MPI_Bcast` to broadcast to all ranks and initialize NCCL communicators

Use the cells below to compile the NCCL version, check for errors, and run it. If you get stuck, solutions are in the solutions (solutions) folder.

In [ ]:

```
# Compiles the NCCL version of MiniWeather using OpenACC, exec = ./openaccNCCL
OPENACC_FLAGS="-acc -gpu=cc70,lineinfo"

mpif90 -I${PNETCDF_INC} -I${NCCL_INC} -Mextend -O0 ${OPENACC_FLAGS} \
-DNV_GPU -DNO_INFORM -D_NX=4096 -D_NZ=2048 -D_SIM_TIME=10.0 -D_OUT_FREQ=10.0 -D_DA
TA_SPEC=DATA_SPEC_THERMAL \
-c miniWeather_mpiNCCL_openacc.F90 -o miniWeather_mpiNCCL_openacc.F90.o

mpif90 -Mextend -O3 ${OPENACC_FLAGS} miniWeather_mpiNCCL_openacc.F90.o -o openaccN
CCL \
-L${PNETCDF_LIB} -lpnetcdf -L${NCCL_LIB} -lnccl
rm -f miniWeather_mpiNCCL_openacc.F90.o
```

In [ ]:

```
# NCCL runs
S=1; N=4
  qcmd -A $PROJECT -q $QUEUE -l select=$S:ncpus=$N:ngpus=$N:mpiprocs=$N -l gpu_typ
e=$GPU_TYPE -l walltime=30 -- \
  $PWD/check_output.sh $PWD/openaccNCCL 1e-13 4.5e-5 $((S*N))
```

Generate `nsys` Profiles of Multi-GPU Jobs

Use the below cells and included `nsysMPI_pbs.sh` (nsysMPI_pbs.sh) script to generate profile reports of each program.

In [ ]:
```
qsub -q gpudev -v EXEC=openacc,N=4 -l select=1:ncpus=4:ngpus=4:mpiprocs=4 nsysMPI_
pbs.sh
```

In [ ]:
```
qsub -q gpudev -v EXEC=openaccAware,N=4 -l select=1:ncpus=4:ngpus=4:mpiprocs=4 nsy
sMPI_pbs.sh
```

In [ ]:
```
qsub -q gpudev -v EXEC=openaccNCCL,N=4 -l select=1:ncpus=4:ngpus=4:mpiprocs=4 nsys
MPI_pbs.sh
```

Additional Considerations

The following `UCX` and `OpenMPI` environment variables are currently recommended for optimal performance of CUDA Aware MPI applications. Future testing and system adjustements may modify these recommendations. Notably, `gdr_copy` is currently not incuded in `UCX_TLS` .

```
export CUDA_LAUNCH_BLOCKING=0
export UCX_TLS=rc,sm,cuda_copy,cuda_ipc
export OMPI_MCA_pml=ucx
export OMPI_MCA_btl=self,vader,tcp,smcuda #openib
export UCX_RNDV_SCHEME=get_zcopy
export UCX_RNDV_THRESH=0
export UCX_MAX_RNDV_RAILS=1
export UCX_MEMTYPE_CACHE=n
```

Add the following to qcmd to try it out.

```
-v
CUDA_LAUNCH_BLOCKING=0,"UCX_TLS='rc,sm,cuda_copy,cuda_ipc'",OMPI_MC

"OMPI_MCA_btl='self,vader,tcp,smcuda'",UCX_RNDV_SCHEME=get_zcopy,UC
 UCX_MAX_RNDV_RAILS=1,UCX_MEMTYPE_CACHE=n
```

Resources

- MPI
    - Cornell's Virtual Workshop 5-part MPI Series (https://cvw.cac.cornell.edu/topics#MPI)
    - NCSA's and UIUC's Introduction to MPI (https://www.hpc-training.org/xsede/moodle/enrol/index.php?id=34) on the hpc-training.org (https://www.hpc-training.org/xsede/moodle/) HPC-Moodle platform
    - XSEDE's HPC Workshop: MPI (https://www.psc.edu/resources/training/xsede-hpc-workshop-may-2022-mpi/) May 2022 offering
- Multi-GPU
    - Condensed NVIDIA documentation on NCCL Fortran API (https://docs.nvidia.com/hpc-sdk/compilers/fortran-cuda-interfaces/index.html#cfnccl-runtime)
    - Full NVIDIA NCCL Documentation (https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html)
    - Julich Multi-GPU Tutorial (https://github.com/FZJ-JSC/tutorial-multi-gpu) material as provided at SC21 and ISC22
    - Cineca's OpenACC Tutorial (https://github.com/romerojosh/cineca-openacc-tutorial), Conjugate Gradient solver using CUDA Aware MPI and NCCL
    - Jiri Kraus' Multi-GPU Programming Model