



# GPU Teaching Kit

Accelerated Computing



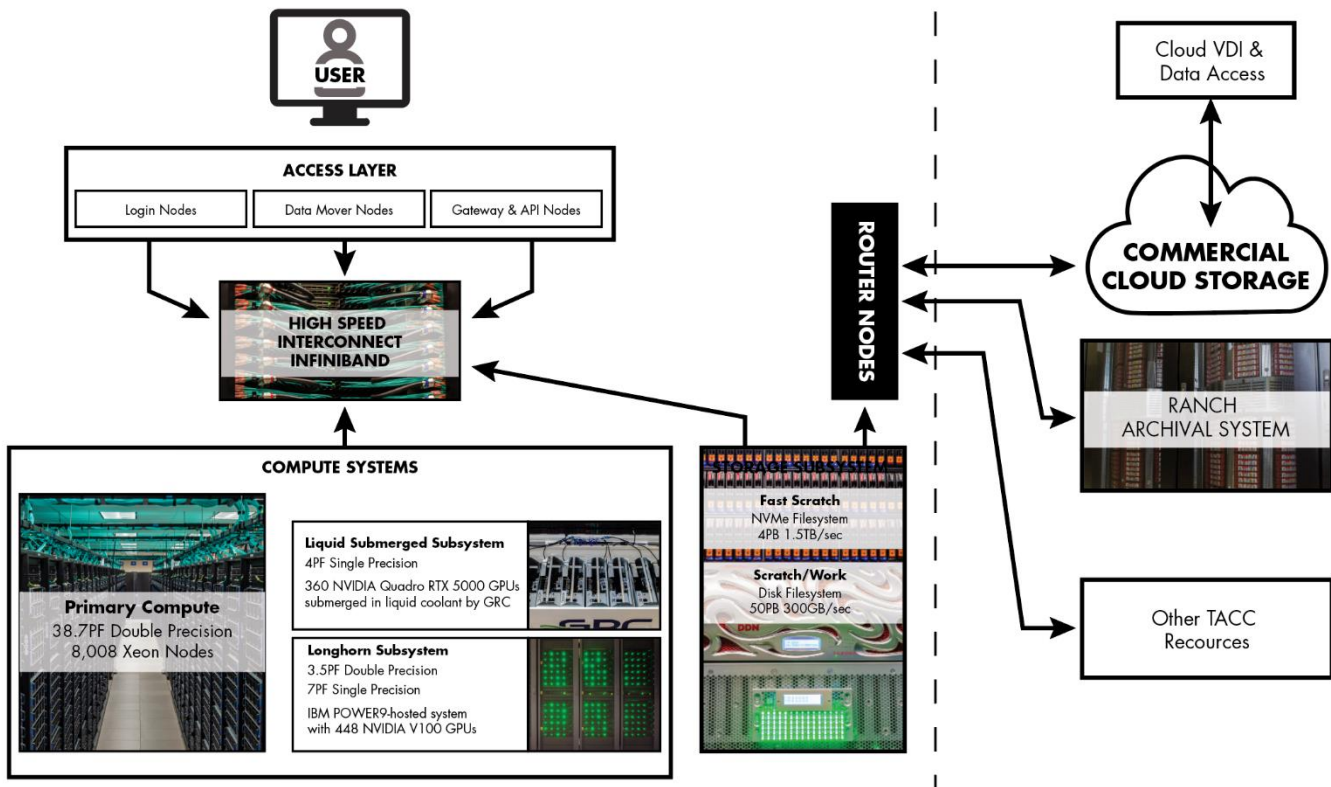
## Module 18 – Related Programming Models: MPI

### Lecture 18.1 - Introduction to Heterogeneous Supercomputing and MPI

# Objective

- To learn the basics of an MPI application
  - Frontera, a supercomputer clusters with heterogeneous CPU-GPU nodes
  - MPI initialization, message passing, and barrier synchronization API functions
  - Vector addition example

# Frontera – Installed at TACC 9/2019



# Frontera: Primary Compute System

8008 Dell EMC PowerEdge C6420 with a Mellanox Infiniband HDR and HDR-100 interconnect.

|                   |   |
|-------------------|---|
| <b>Processors</b> | Intel Xeon Platinum 8280 ("Cascade Lake") <ul style="list-style-type: none"><li>• Number of cores: 28 per socket, 56 per node.</li><li>• Clock rate: 2.7Ghz ("Base Frequency")</li><li>• "Peak" node performance: 4.8TF, double precision</li></ul> |
|-------------------|---|

|               |                          |
|---------------|--------------------------|
| <b>Memory</b> | DDR-4 memory, 192GB/node |
|---------------|--------------------------|

|                   |                  |
|-------------------|------------------|
| <b>Local Disk</b> | 480 GB SSD drive |
|-------------------|------------------|

|                |                              |
|----------------|------------------------------|
| <b>Network</b> | Mellanox InfiniBand, HDR-100 |
|----------------|------------------------------|

# Frontera: Compute Subsystems

Submerged liquid cooled system – 90 nodes

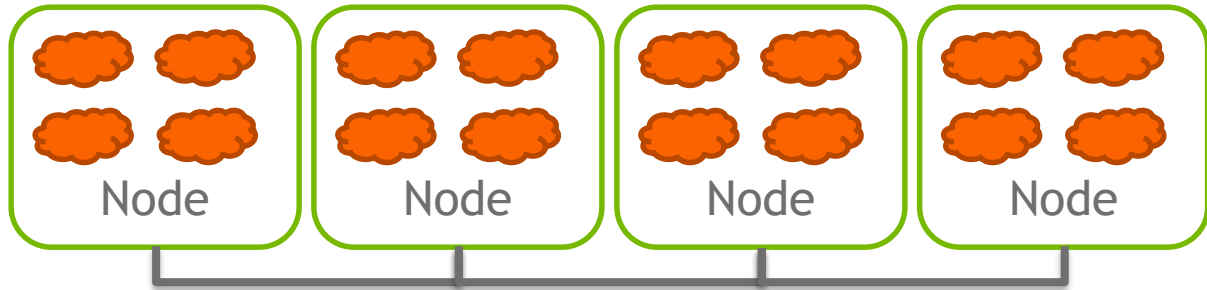
|                         |                                 |
|-------------------------|---------------------------------|
| <b>Processors</b>       | 360 NVIDIA Quadro RTX 5000 GPUs |
| <b>Memory</b>           | 128GB per node                  |
| <b>Cooling</b>          | GRC ICeraQ™ system              |
| <b>Network</b>          | Mellanox InfiniBand, HDR-100    |
| <b>Peak Performance</b> | 4PF single precision            |

“Longhorn”: 108 IBM Power System AC922 nodes

|                         |  |
|-------------------------|--|
| <b>Processors</b>       | IBM POWER9-hosted system with 448 NVIDIA V100 GPUs |
| <b>Memory</b>           | 256GB per node (4 nodes with 512GB per node)       |
| <b>Storage</b>          | 5 petabyte filesystem                              |
| <b>Network</b>          | Infiniband EDR network                             |
| <b>Peak Performance</b> | 3.5PF double precision; 7.0 single precision       |

# MPI – Programming and Execution Model

- Many processes distributed in a cluster



- Each process computes part of the output
- Processes communicate with each other
- Processes can synchronize

# MPI Initialization, Info and Sync

- `int MPI_Init(int *argc, char ***argv)`
  - Initialize MPI
- `MPI_COMM_WORLD`
  - MPI group with all allocated nodes
- `int MPI_Comm_rank (MPI_Comm comm, int *rank)`
  - Rank of the calling process in group of comm
- `int MPI_Comm_size (MPI_Comm comm, int *size)`
  - Number of processes in the group of comm

# Vector Addition: Main Process

```
int main(int argc, char *argv[]) {  
    int vector_size = 1024 * 1024 * 1024;  
    int pid=-1, np=-1;  
  
    MPI_Init(&argc, &argv);  
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);  
    MPI_Comm_size(MPI_COMM_WORLD, &np);  
  
    if(np < 3) {  
        if(0 == pid) printf("Need 3 or more processes.\n");  
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;  
    }
```





# Vector Addition: Main Process

```
if(pid < np - 1)
    compute_node(vector_size / (np - 1));
else
    data_server(vector_size);

MPI_Finalize();
return 0;
}
```

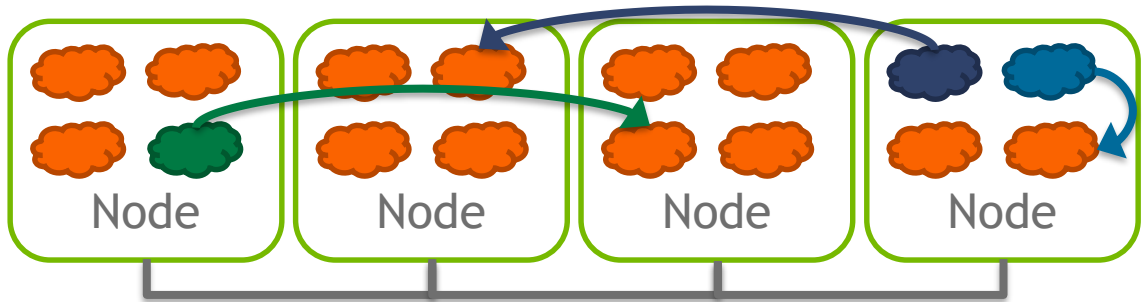


# MPI Sending Data

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - **Buf**: Initial address of send buffer (choice)
  - **Count**: Number of elements in send buffer (nonnegative integer)
  - **Datatype**: Datatype of each send buffer element (handle)
  - **Dest**: Rank of destination (integer)
  - **Tag**: Message tag (integer)
  - **Comm**: Communicator (handle)

# MPI Sending Data

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - **Buf**: Initial address of send buffer (choice)
  - **Count**: Number of elements in send buffer (nonnegative integer)
  - **Datatype**: Datatype of each send buffer element (handle)
  - **Dest**: Rank of destination (integer)
  - **Tag**: Message tag (integer)
  - **Comm**: Communicator (handle)

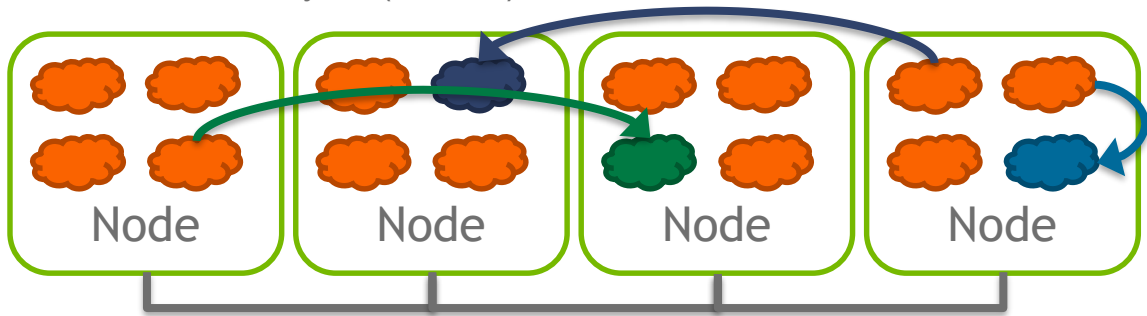


# MPI Receiving Data

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - **Buf**: Initial address of receive buffer (choice)
  - **Count**: Maximum number of elements in receive buffer (integer)
  - **Datatype**: Datatype of each receive buffer element (handle)
  - **Source**: Rank of source (integer)
  - **Tag**: Message tag (integer)
  - **Comm**: Communicator (handle)
  - **Status**: Status object (Status)

# MPI Receiving Data

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - **Buf**: Initial address of receive buffer (choice)
  - **Count**: Maximum number of elements in receive buffer (integer)
  - **Datatype**: Datatype of each receive buffer element (handle)
  - **Source**: Rank of source (integer)
  - **Tag**: Message tag (integer)
  - **Comm**: Communicator (handle)
  - **Status**: Status object (Status)



# Vector Addition: Server Process (I)

```
void data_server(unsigned int vector_size) {
    int np, num_nodes = np - 1, first_node = 0, last_node = np - 2;
    unsigned int num_bytes = vector_size * sizeof(float);
    float *input_a = 0, *input_b = 0, *output = 0;

    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    /* Allocate input data */
    input_a = (float *)malloc(num_bytes);
    input_b = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    if(input_a == NULL || input_b == NULL || output == NULL) {
        printf("Server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data */
    random_data(input_a, vector_size, 1, 10);
    random_data(input_b, vector_size, 1, 10);
}
```

# Vector Addition: Server Process (II)

```
/* Send data to compute nodes */
float *ptr_a = input_a;
float *ptr_b = input_b;

for(int process = 1; process < last_node; process++) {
    MPI_Send(ptr_a, vector_size / num_nodes, MPI_FLOAT,
             process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_a += vector_size / num_nodes;

    MPI_Send(ptr_b, vector_size / num_nodes, MPI_FLOAT,
             process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_b += vector_size / num_nodes;
}

/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);
```

# Vector Addition: Server Process (III)

```
/* Wait for previous communications */
MPI_Barrier(MPI_COMM_WORLD);

/* Collect output data */
MPI_Status status;
for(int process = 0; process < num_nodes; process++) {
    MPI_Recv(output + process * num_points / num_nodes,
             num_points / num_comp_nodes, MPI_REAL, process,
             DATA_COLLECT, MPI_COMM_WORLD, &status );
}

/* Store output data */
store_output(output, dimx, dimy, dimz);

/* Release resources */
free(input_a);
free(input_b);
free(output);
}
```



# Vector Addition: Compute Process (I)

```
void compute_node(unsigned int vector_size ) {
    int np;
    unsigned int num_bytes = vector_size * sizeof(float);
    float *input_a, *input_b, *output;
    MPI_Status status;

    MPI_Comm_size(MPI_COMM_WORLD, &np);
    int server_process = np - 1;

    /* Alloc host memory */
    input_a = (float *)malloc(num_bytes);
    input_b = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);

    /* Get the input data from server process */
    MPI_Recv(input_a, vector_size, MPI_FLOAT, server_process,
             DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
    MPI_Recv(input_b, vector_size, MPI_FLOAT, server_process,
             DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
```

# MPI Barriers

- `int MPI_Barrier (MPI_Comm comm)`
  - Comm: Communicator (handle)
- Blocks the caller until all group members have called it; the call returns at any process only after all group members have entered the call.

# MPI Barriers

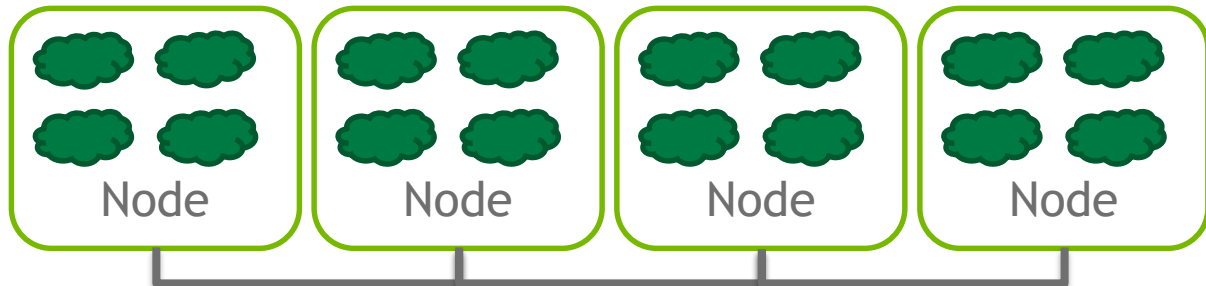
- Wait until all other processes in the MPI group reach the same barrier
  - All processes are executing Do\_Stuff()
  - Some processes reach the barrier and the wait in the barrier until all reach the barrier

## Example Code

```
Do_stuff();
```

```
MPI_Barrier();
```

```
Do_more_stuff();
```



# Vector Addition: Compute Process (II)

```
/* Compute the partial vector addition */
for(int i = 0; i < vector_size; ++i) {
    output[i] = input_a[i] + input_b[i];
}

/* Report to barrier after computation is done*/
MPI_Barrier(MPI_COMM_WORLD);

/* Send the output */
MPI_Send(output, vector_size, MPI_FLOAT,
         server_process, DATA_COLLECT, MPI_COMM_WORLD);

/* Release memory */
free(input_a);
free(input_b);
free(output);
}
```



# GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).