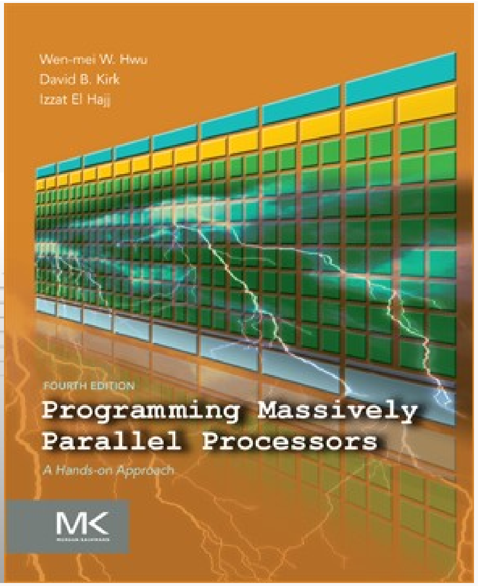
Programming Massively Parallel Processors

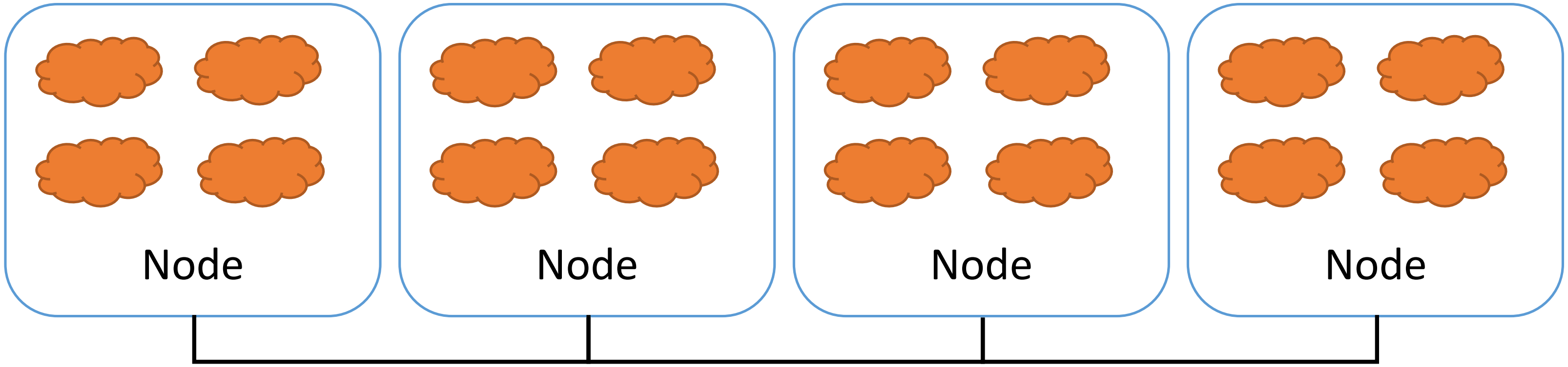
A Hands-on Approach

CHAPTER 20 Programming a Heterogeneous Computing Cluster



MPI 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
  - Also MPI_Finalize...

- `MPI_COMM_WORLD`
  - MPI communicator with all allocated nodes

- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Rank of the calling process in group of *comm*

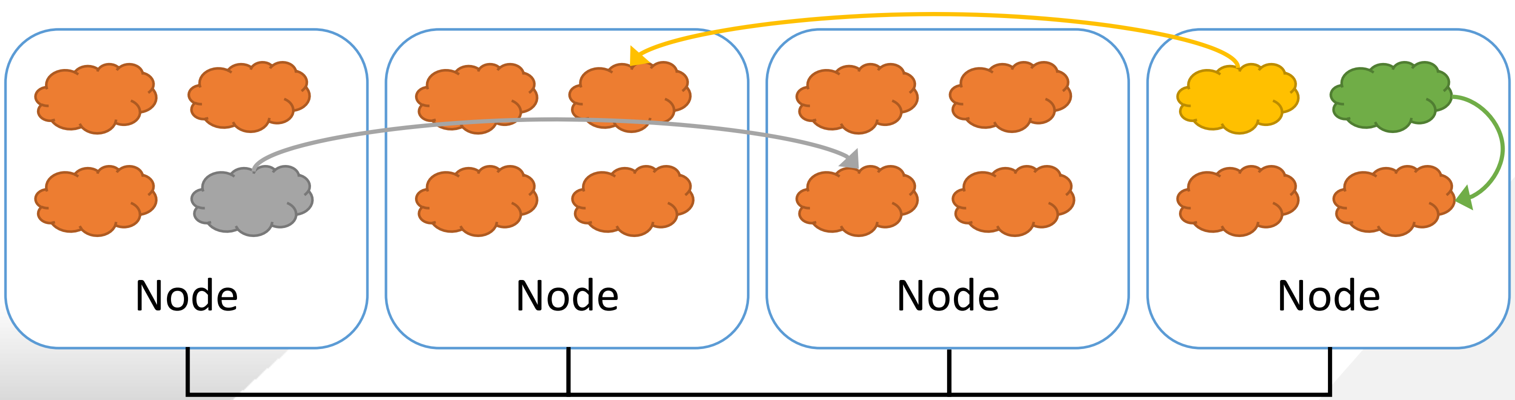
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Number of processes in the group of *comm*

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)

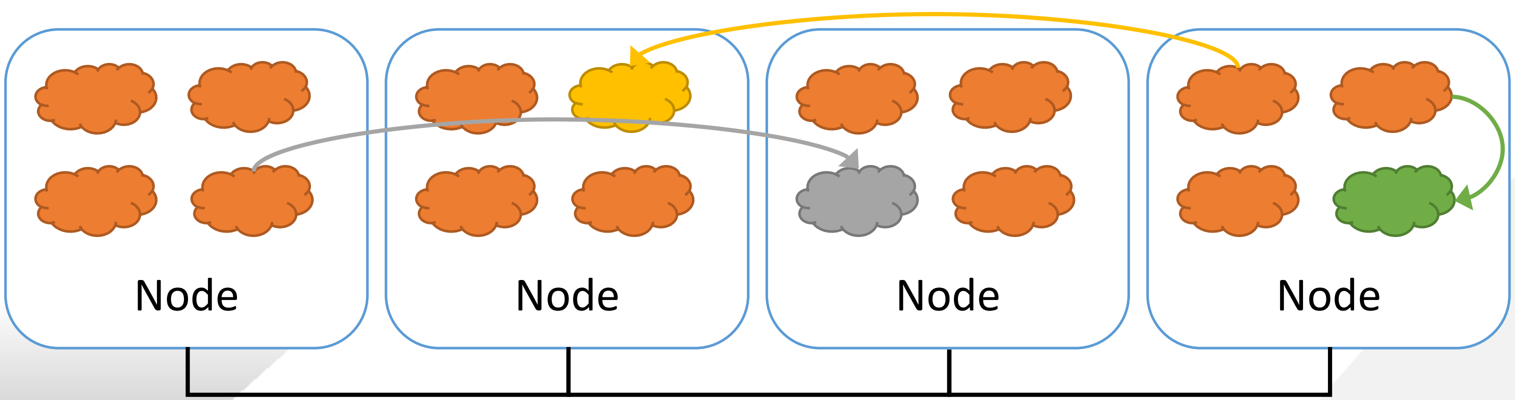


MPI Receiving Data

- `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

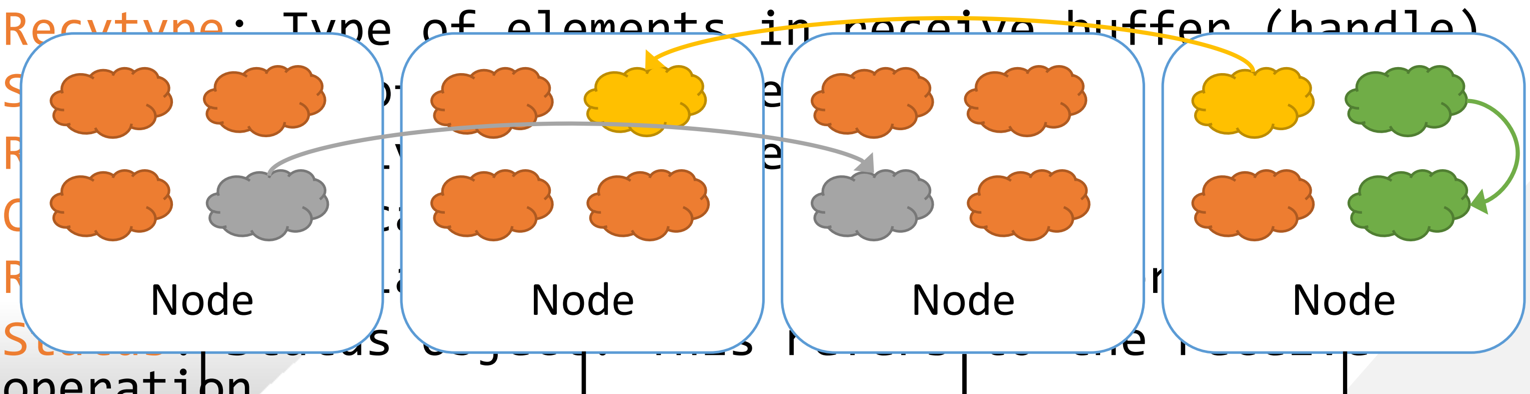
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - Buf: Initial address of receive buffer (choice)
  - Count: Maximum number of elements in receive buffer (integer)
  - Datatype: Datatype of each receive buffer element (handle)
 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



MPI Sending and Receiving Data

- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`
  - Sendbuf: Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  - Recvtype: Type of elements in receive buffer (handle)
  - Source: Rank of source (integer)
  - Recvtag: Receive tag (integer)
  - Comm: Communicator (handle)
  - Recvbuf: Initial address of receive buffer (choice)
  - Status: Status object. This refers to the receive operation

- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`
  - Sendbuf: Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  - Recvtype: Type of elements in receive buffer (handle)
  - S
  - R
  - C
  - R
  - Status: Status object. This refers to the receive operation



MPI Barriers

- `int MPI_Barrier(MPI_Comm comm)`
  - Comm: Communicator (handle)

- Blocks the caller until all group members have called it

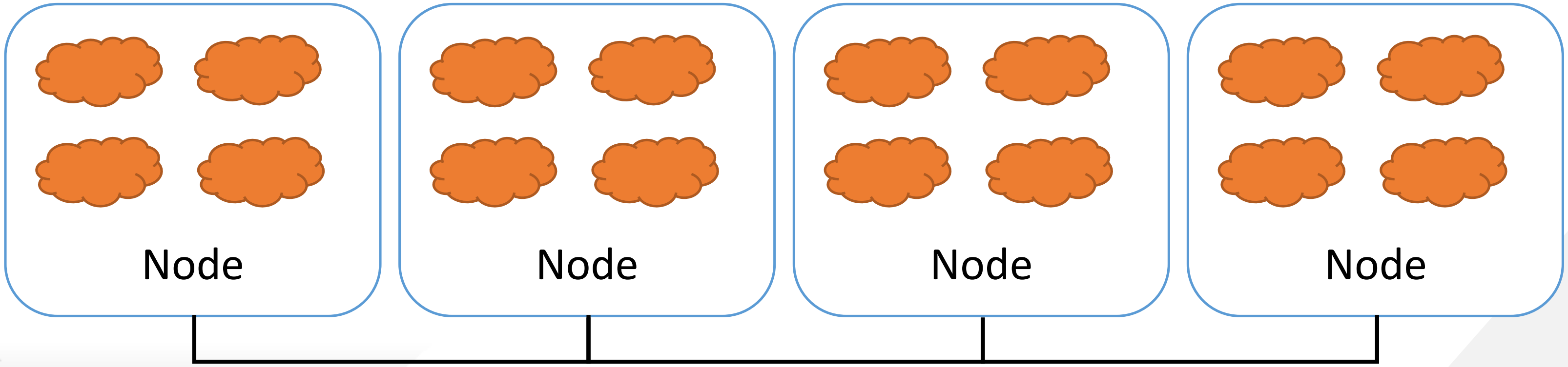
- The call returns at any process only after all group members have entered the call

MPI Barriers

- Wait until all other processes in the MPI *comm* reach the same barrier
  1. All processes are executing Do_Stuff()

Example Code

```
Do_stuff();

MPI_Barrier();

Do_more_stuff();
```

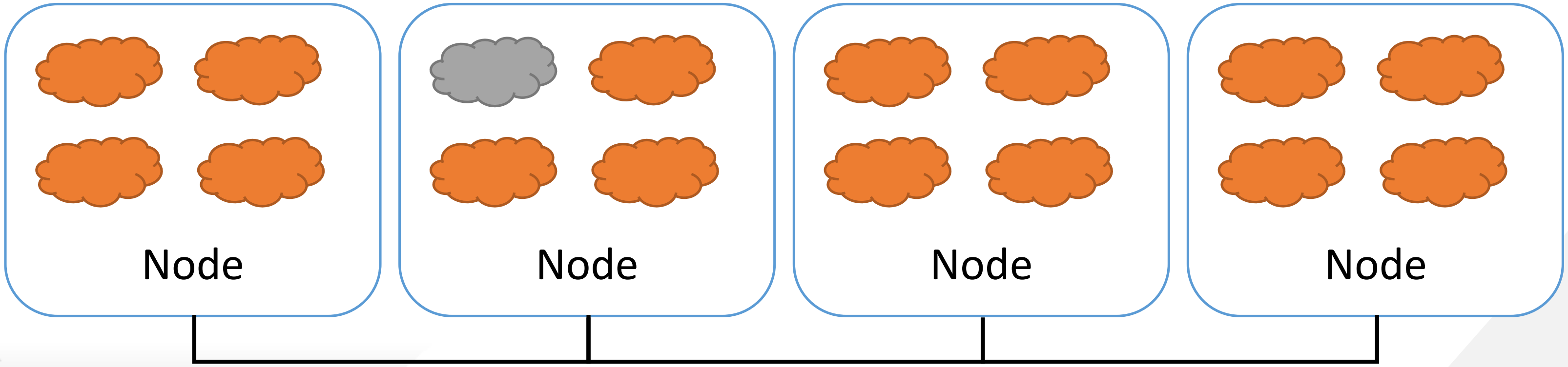


MPI Barriers

- Wait until all other processes in the MPI *comm* reach the same barrier
  1. All processes are executing Do_Stuff()
  2. Some processes reach the barrier

Example Code

```
Do_stuff();

MPI_Barrier();

Do_more_stuff();
```

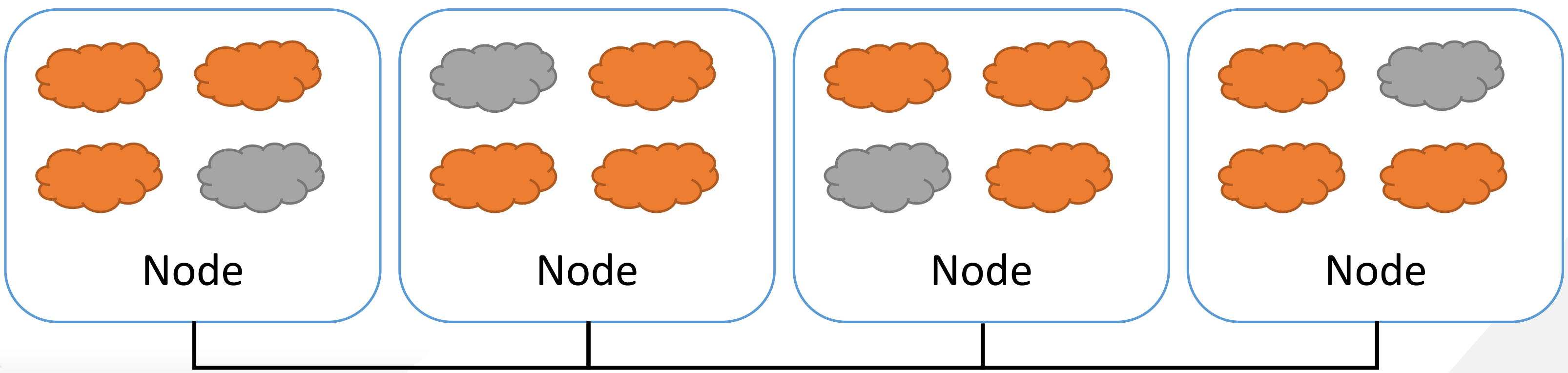


MPI Barriers

- Wait until all other processes in the MPI *comm* reach the same barrier
  1. All processes are executing Do_Stuff()
  2. Some processes reach the barrier and then wait in the barrier

Example Code

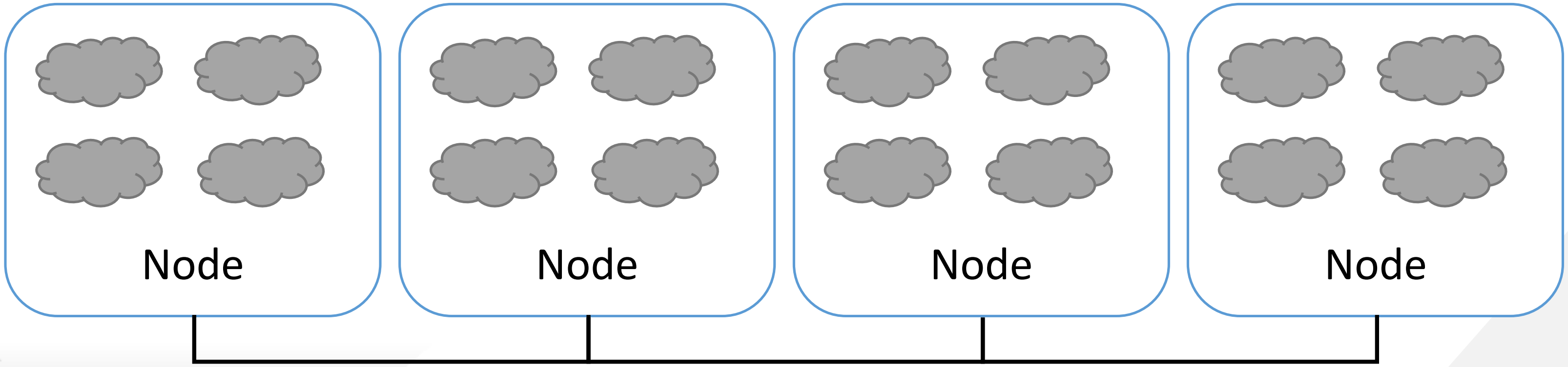
```
Do_stuff();

MPI_Barrier();

Do_more_stuff();
```



- Wait until all other processes in the MPI *comm* reach the same barrier
  1. All processes are executing Do_Stuff()
  2. Some processes reach the barrier and then wait in the barrier until all reach the barrier

Example Code

```
Do_stuff();

MPI_Barrier();

Do_more_stuff();
```

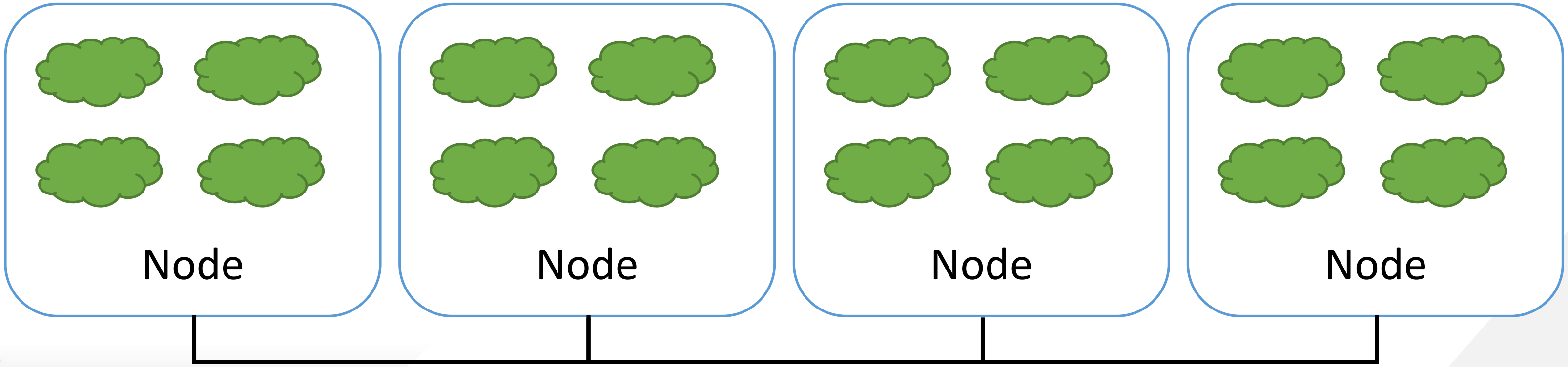


MPI Barriers

- Wait until all other processes in the MPI *comm* reach the same barrier
  1. All processes are executing Do_Stuff()
  2. Some processes reach the barrier and then wait in the barrier until all reach the barrier
  3. All processes execute Do_more_stuff()

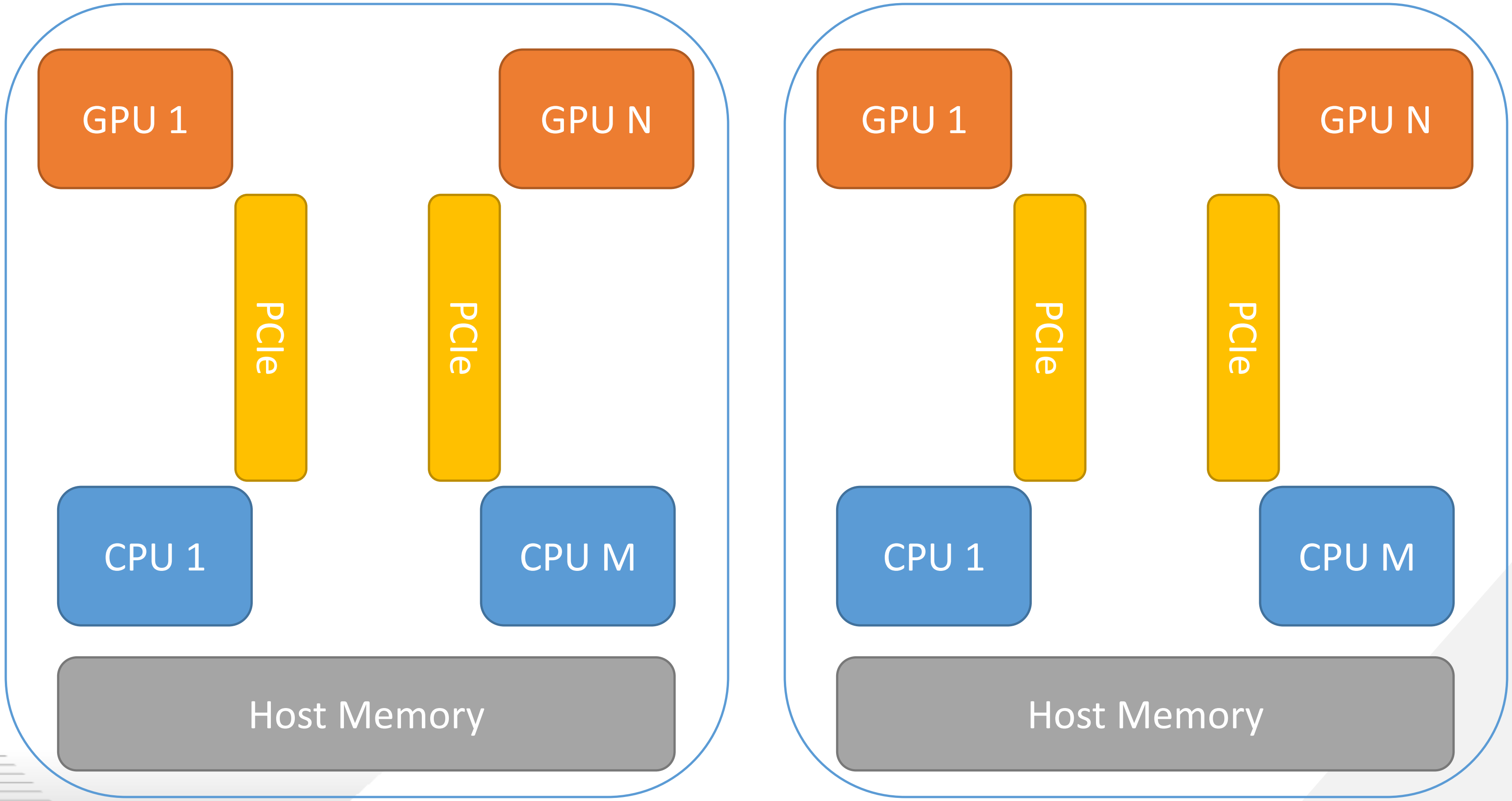
Example Code

```
Do_stuff();

MPI_Barrier();

Do_more_stuff();
```



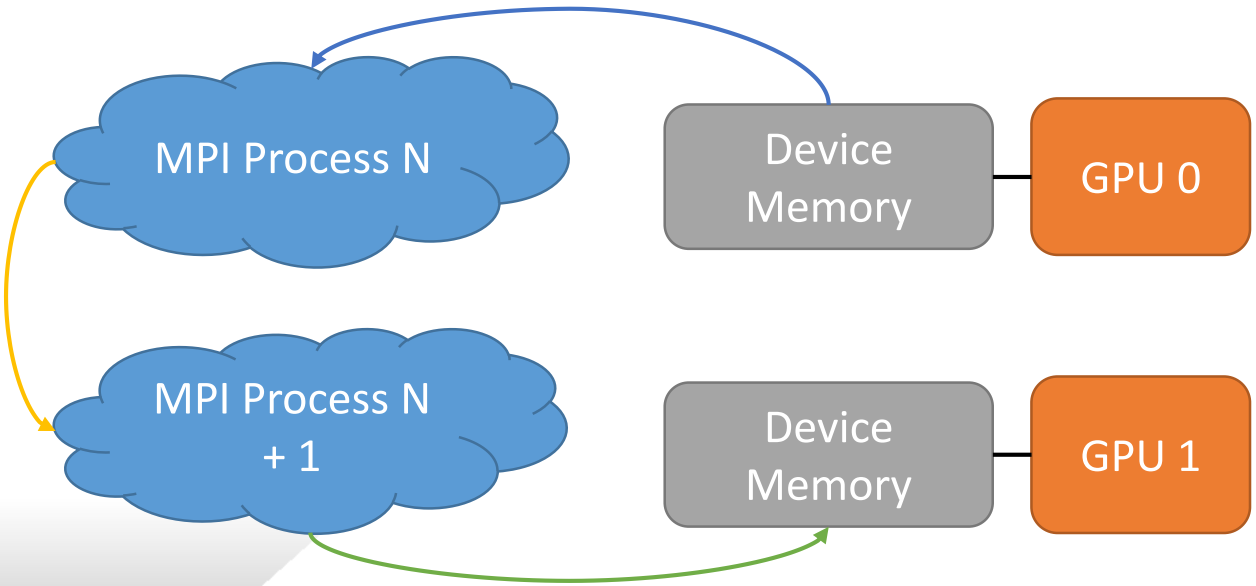
CUDA-based cluster

- Each node contains *N* GPUs



CUDA and MPI Communication

- Source MPI process:
  - `cudaMemcpy(tmp, src, cudaMemcpyDeviceToHost)`
  - `MPI_Send(tmp, ...)`
- Destination MPI process:
  - `MPI_Recv(tmp, ...)`
  - `cudaMemcpy(dst, tmp, cudaMemcpyHostToDevice)`

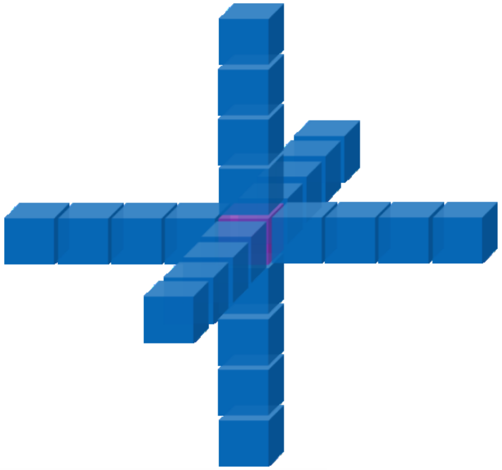


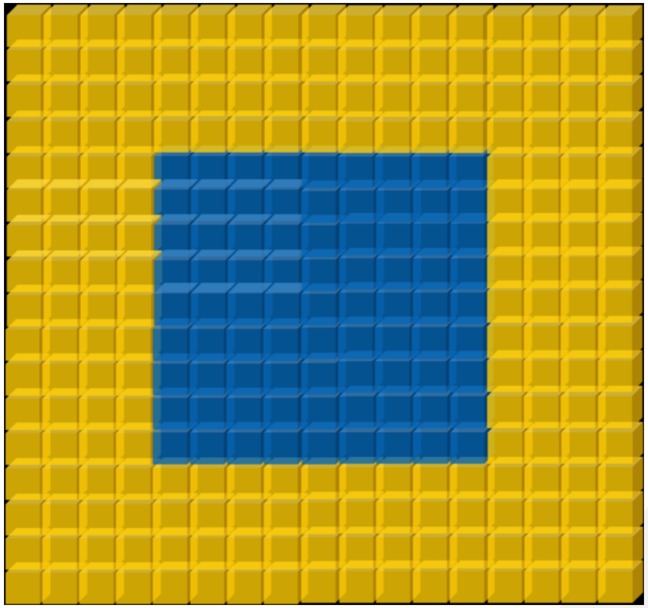
Review of Stencil Computations

- Example: wave propagation modeling

$$\nabla^2 U - \frac{1}{v^2}\frac{\partial U}{\partial t} = 0$$

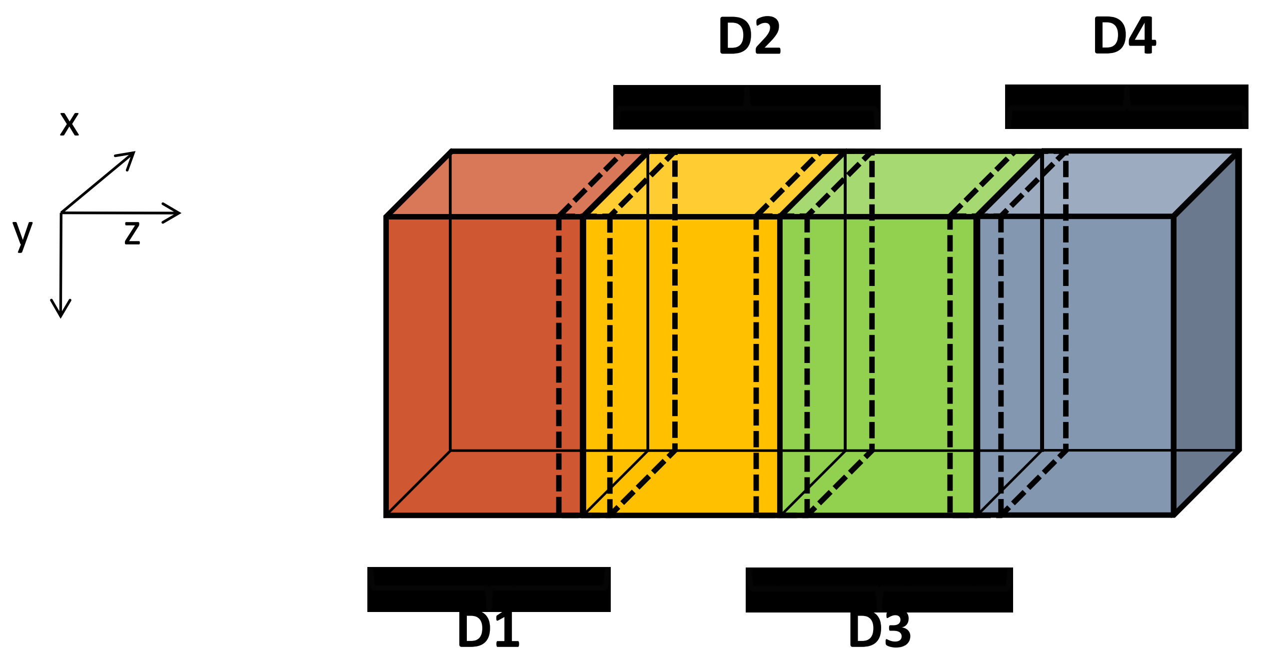
- Approximate Laplacian using finite differences





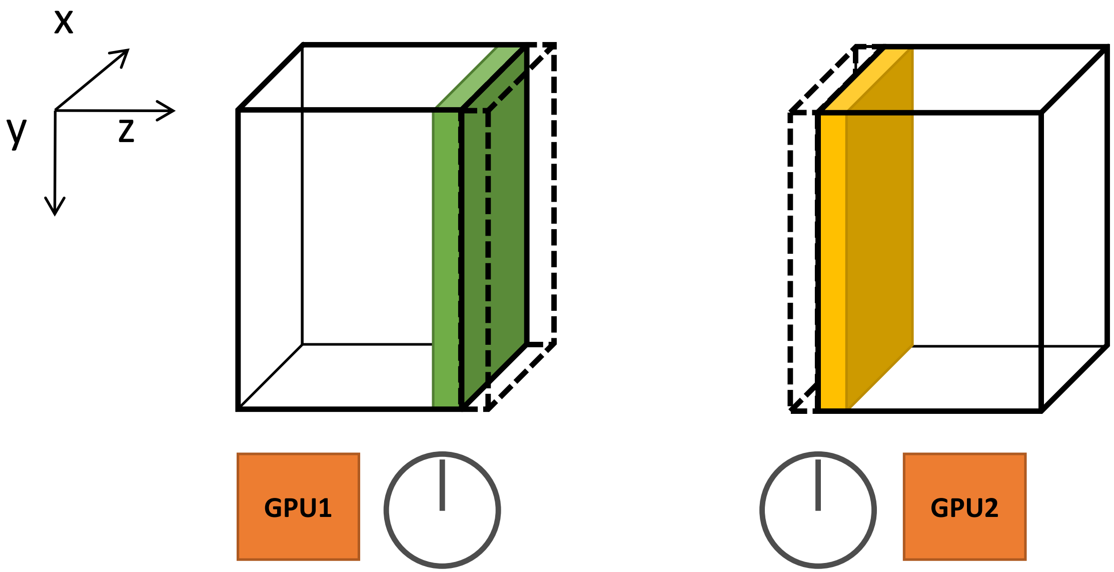
Stencil Domain Decomposition

- Volumes are split into tiles (along the Z-axis)
  - 3D-Stencil introduces data dependencies



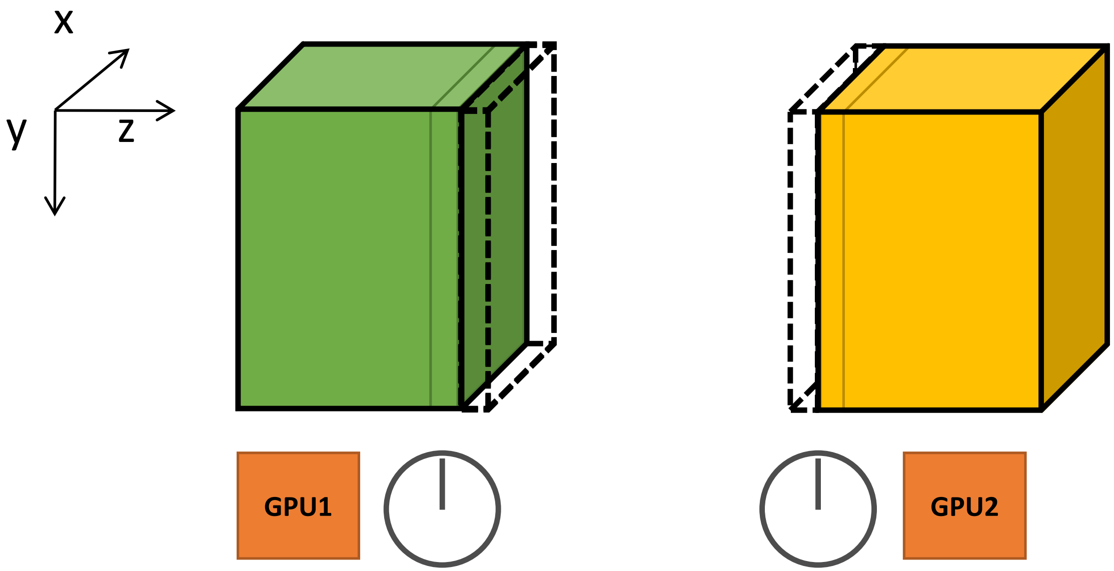
Boundary Exchange Example (I)

- Approach: two-stage execution
  - Stage 1: compute the field points to be exchanged



Boundary Exchange Example (II)

- Approach: two-stage execution
  - Stage 2: Compute the remaining points ***while*** exchanging the boundaries



Stencil Code: Compute Process (IV)

```
for(int i=0; i < nreps; i++) {
    /* Compute values needed by other nodes first */
    launch_kernel(d_output + left_stage1_offset,
        d_input + left_stage1_offset, dimx, dimy, 12, stream1);
    launch_kernel(d_output + right_stage1_offset,
        d_input + right_stage1_offset, dimx, dimy, 12, stream1);

    /* Compute the remaining points */
    launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                 dimx, dimy, dimz, stream2);

    /* Copy the data needed by other nodes to the host */
    cudaMemcpyAsync(h_left_ghost_own,
            d_output + num_ghost_points,
            num_ghost_bytes, cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(h_right_ghost_own,
                d_output + right_stage1_offset + num_ghost_points,
                num_ghost_bytes, cudaMemcpyDeviceToHost, stream1);
    cudaStreamSynchronize(stream1);
```

Stencil Code: Compute Process (V)

```
    /* Send data to left, get data from right */
    MPI_Sendrecv(h_left_ghost_own, num_ghost_points, MPI_REAL,
                 left_neighbor,  i,
                 h_right_ghost, num_ghost_points, MPI_REAL,
                 right_neighbor, i,
                 MPI_COMM_WORLD, &status);
    /* Send data to right, get data from left */
    MPI_Sendrecv(h_right_ghost_own, num_ghost_points, MPI_REAL,
                 right_neighbor, i,
                 h_left_ghost, num_ghost_points, MPI_REAL,
                 left_neighbor, i,
                 MPI_COMM_WORLD, &status);

    cudaMemcpyAsync(d_output+left_ghost_offset,  h_left_ghost,
                 num_ghost_bytes, cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_output+right_ghost_offset, h_right_ghost,
                 num_ghost_bytes, cudaMemcpyHostToDevice, stream1);
    cudaDeviceSynchronize();

    float *temp = d_output;
    d_output = d_input; d_input = temp;
}
```

All streams synced

CUDA-Aware MPI

- Several MPI implementations provide optimized CUDA paths
  - Understand CUDA pointers (no intermediate copies!)
  - Different nodes: Optimized GPU-network pipelined transfers
    - GPUDirect RDMA if available
  - Same node: Use shared memory across processes
    - **`cudaIpc{Open,Get,Close}MemHandle`**
    - No *internal* intermediate copies!

- `Available`
  - OpenMPI >= 1.7
  - MVAPICH2 >= 1.8
  - CRAY MPI >= MPT 5.6.2
  - IBM Platform MPI >= 8.3
  - SGI MPI >= 1.08

Stencil Code: Compute Process (IV) CUDA-aware MPI

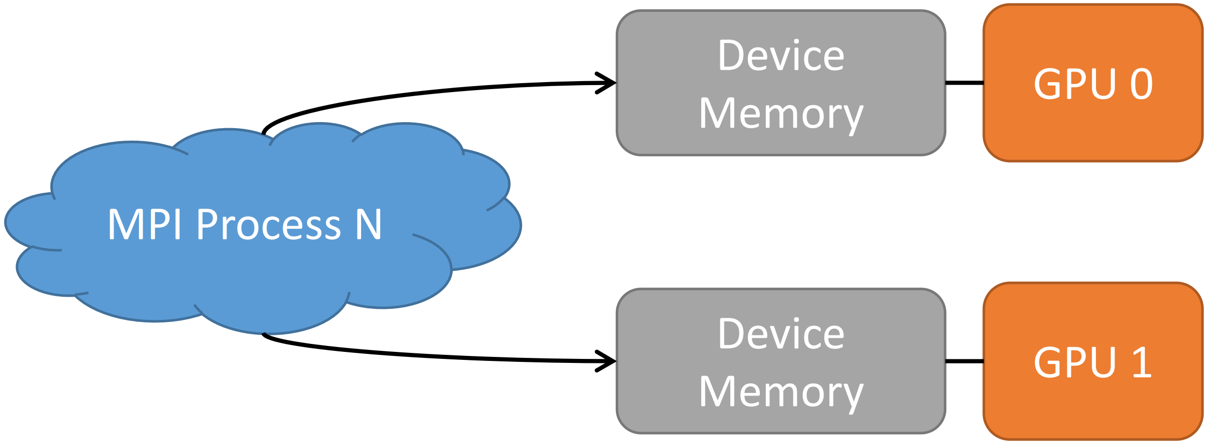
```
for(int i=0; I < nreps; i++) {
    /* Compute values needed by other nodes first */
    launch_kernel(d_output + left_stage1_offset,
        d_input + left_stage1_offset, dimx, dimy, 12, stream1);
    launch_kernel(d_output + right_stage1_offset,
        d_input + right_stage1_offset, dimx, dimy, 12, stream1);

    /* Compute the remaining points */
    launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                dimx, dimy, dimz, stream2);

    /* Wait for points in the boundary to be computed */
    cudaStreamSynchronize(stream1);
```

Stencil Code: Compute Process (V) CUDA-aware MPI

```
    /* Send data to left, get data from right */
    MPI_Sendrecv(d_output+num_ghost_points, num_ghost_points,
                 MPI_REAL, left_neighbor, i,
                  d_output+right_ghost_offset, num_ghost_points,
                 MPI_REAL, right_neighbor, i,
                 MPI_COMM_WORLD, &status);
    /* Send data to right, get data from left */
    MPI_Sendrecv(d_output+right_stage1_offset+num_ghost_points,
                  num_ghost_points, MPI_REAL,
                  right_neighbor, i,
                  d_output+left_ghost_offset, num_ghost_points,
                 MPI_REAL, left_neighbor, i,
                 MPI_COMM_WORLD, &status);

    cudaStreamSynchronize(stream2);

    float *temp = d_output;
    d_output = d_input; d_input = temp;
}-
```

CUDA and MPI

- MPI Processes *can* handle more than one GPU



- Peer GPU-to-GPU communication without need for MPI

- Several MPI processes sharing the same GPU introduce context switch overheads....
- ... but Hyper-Q in Kepler greatly reduces such overheads

References

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.