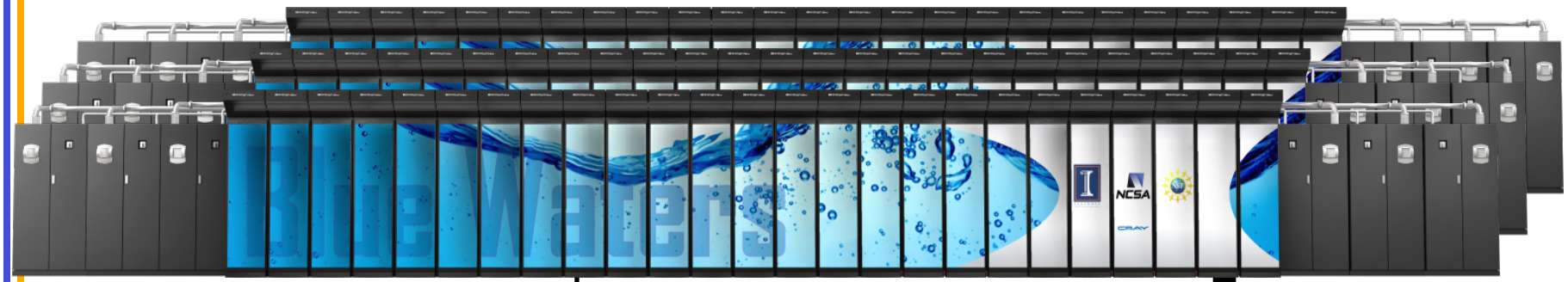
ECE408 / CS483

Applied Parallel Programming

# Lecture 26: Joint CUDA-MPI Programming

*Modified by Karen L. Karavanic*

# Blue Waters Computing System




10/40/100 Gb Ethernet Switch

IB Switch

>1 TB/sec

120+ Gb/sec

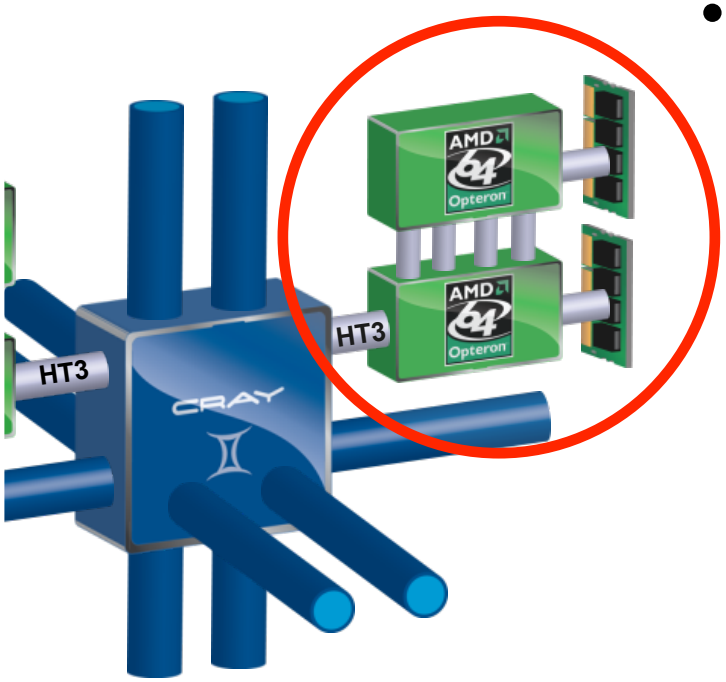
100 GB/sec

**WAN**

**Spectra Logic: 300 PBs**

**Sonexion: 26 PBs**

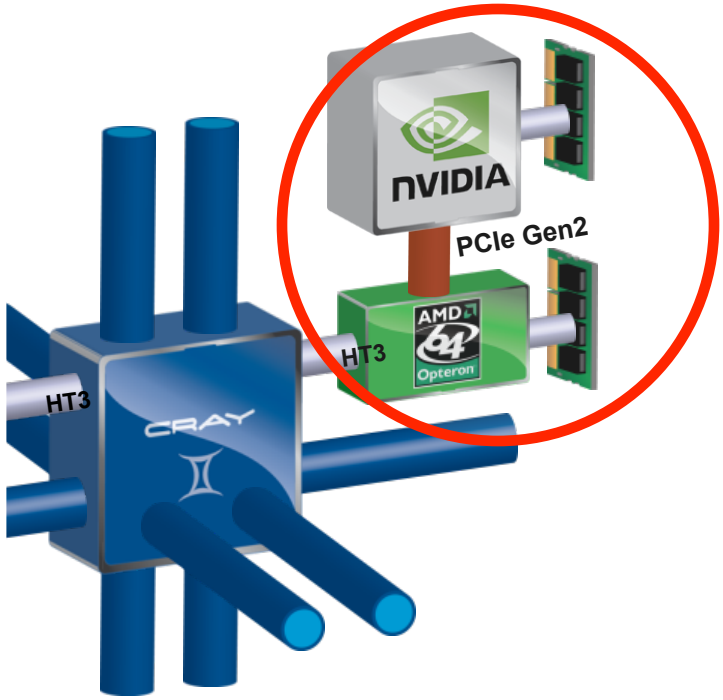
# Cray XE6 Nodes




- Dual-socket Node
  - Two AMD Interlagos chips
    - 16 core modules, 64 threads
    - 313 GFs peak performance
    - 64 GBs memory
      - 102 GB/sec memory bandwidth
  - Gemini Interconnect
    - Router chip & network interface
    - Injection Bandwidth (peak)
      - 9.6 GB/sec per direction

**Blue Waters contains 22,640 Cray XE6 compute nodes.**

# Cray XK7 Nodes

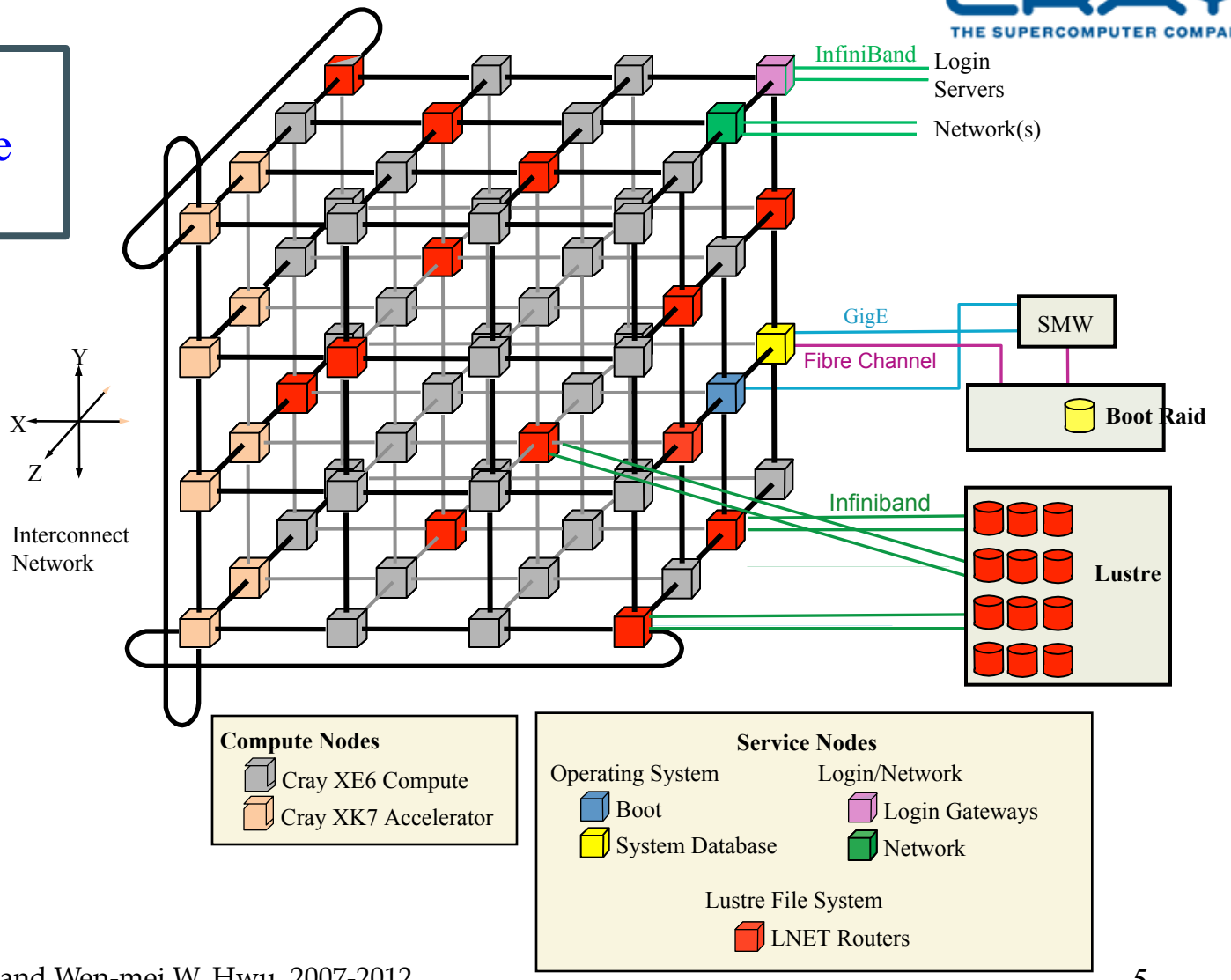



- Dual-socket Node
  - One AMD Interlagos chip
    - 8 core modules, 32 threads
    - 156.5 GFs peak performance
    - 32 GBs memory
      - 51 GB/s bandwidth
  - One NVIDIA Kepler chip
    - 1.3 TFs peak performance
    - 6 GBs GDDR5 memory
      - 250 GB/sec bandwidth
  - Gemini Interconnect
    - Same as XE6 nodes

**Blue Waters contains 3,072 Cray XK7 compute nodes.**

# Gemini Interconnect Network

Blue Waters
3D Torus Size
23 x 24 x 24



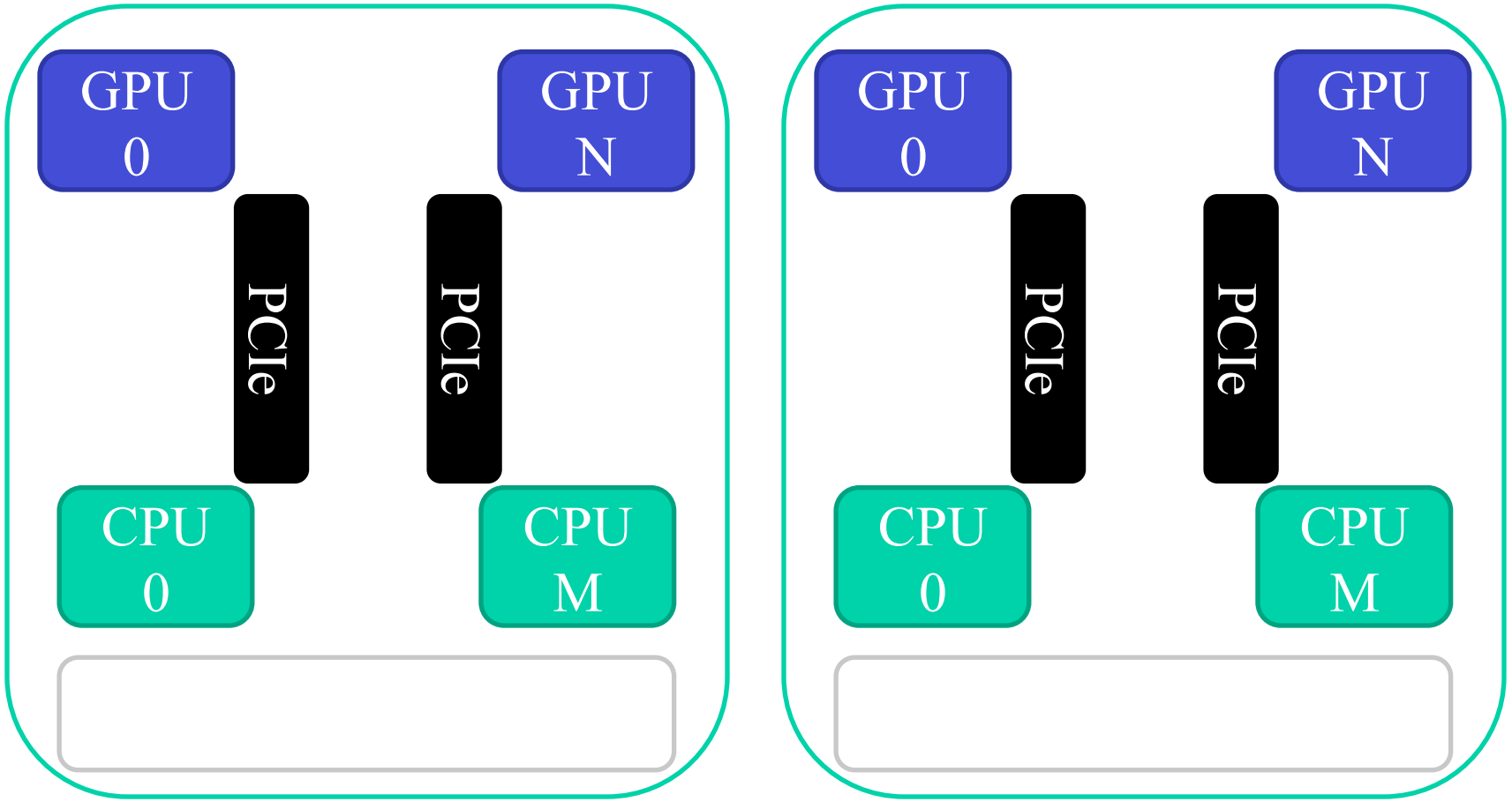
# Blue Waters and Titan Computing Systems

| System Attribute | NCSA Blue Waters | ORNL Titan |
|---|---|---|
| Vendors | Cray/AMD/NVIDIA | Cray/AMD/NVIDIA |
| Processors | Interlagos/Kepler | Interlagos/Kepler |
| Total Peak Performance (PF) | 11.1 | 27.1 |
| Total Peak Performance (CPU/GPU) | 7.1/4 | 2.6/24.5 |
| Number of CPU Chips | 48,352 | 18,688 |
| Number of GPU Chips | 3,072 | 18,688 |
| Amount of CPU Memory (TB) | 1511 | 584 |
| Interconnect | 3D Torus | 3D Torus |
| Amount of On-line Disk Storage (PB) | 26 | 13.6 |
| Sustained Disk Transfer (TB/sec) | >1 | 0.4-0.7 |
| Amount of Archival Storage | 300 | 15-30 |
| Sustained Tape Transfer (GB/sec) | 100 | 7 |

| Science Area | Number of Teams | Codes | Struct Grids | Unstruct Grids | Dense Matrix | Sparse Matrix | N-Body | Monte Carlo | FFT | PIC | Significant I/O |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Climate and Weather | 3 | CESM, GCRM, CM1/WRF, HOMME | X | X | | X | | X | | | X |
| Plasmas/Magnetosphere | 2 | H3D(M),VPIC, OSIRIS, Magtail/ UPIC | X | | | | X | | X | | X |
| Stellar Atmospheres and Supernovae | 5 | PPM, MAESTRO, CASTRO, SEDONA, ChaNGa, MS-FLUKSS | X | | | X | X | X | | X | X |
| Cosmology | 2 | Enzo, pGADGET | X | | | X | X | | | | |
| Combustion/Turbulence | 2 | PSDNS, DISTUF | X | | | | | | X | | |
| General Relativity | 2 | Cactus, Harm3D, LazEV | X | | | X | | | | | |
| Molecular Dynamics | 4 | AMBER, Gromacs, NAMD, LAMMPS | | | | X | X | | X | | |
| Quantum Chemistry | 2 | SIAL, GAMESS, NWChem | | | X | X | X | X | | | X |
| Material Science | 3 | NEMOS, OMEN, GW, QMCPACK | | | X | X | X | X | | | |
| Earthquakes/Seismology | 2 | AWP-ODC, HERCULES, PLSQR, SPECFEM3D | X | X | | | X | | | | X |
| Quantum Chromo Dynamics | 1 | Chroma, MILC, USQCD | X | | X | X | | | | | |
| Social Networks | 1 | EPISIMDEMICS | | | | | | | | | |
| Evolution | 1 | Eve | | | | | | | | | |
| Engineering/System of Systems | 1 | GRIPS,Revisit | | | | | | X | | | |
| Computer Science | 1 | | | X | X | X | | | X | | X |

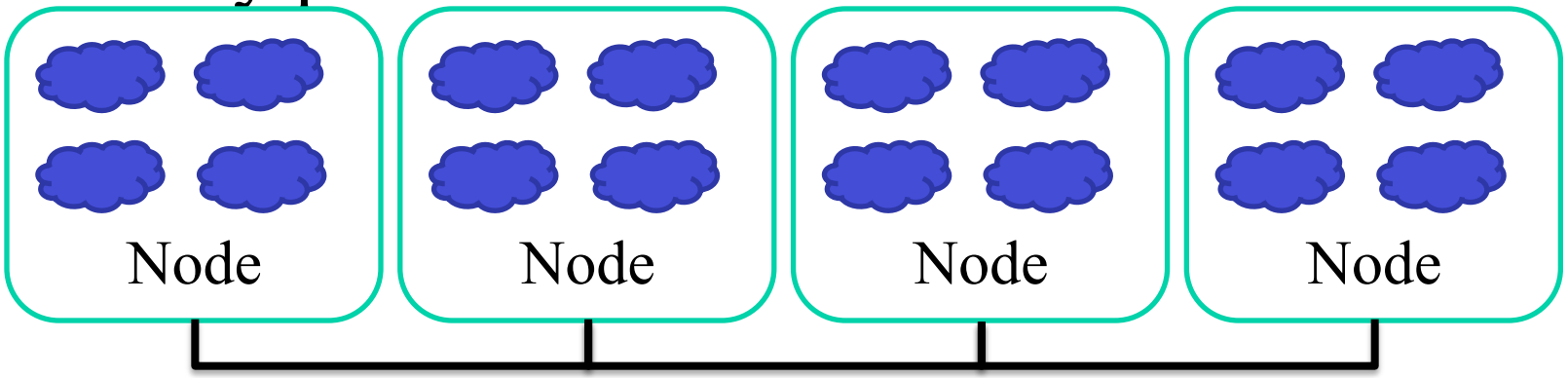
# CUDA-based cluster

- Each node contains *N* GPUs



# 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
- `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

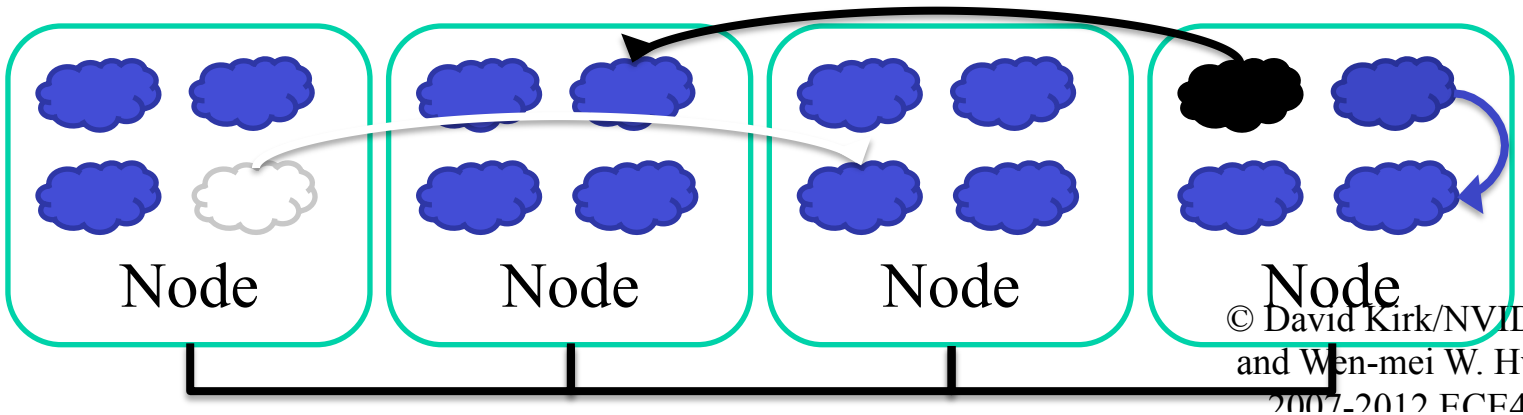
```
int main(int argc, char *argv[]) {
    int vector_size = 1024 * 1024 * 1024;
    int pid=-1, np=-1;

    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("Nedded 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    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)

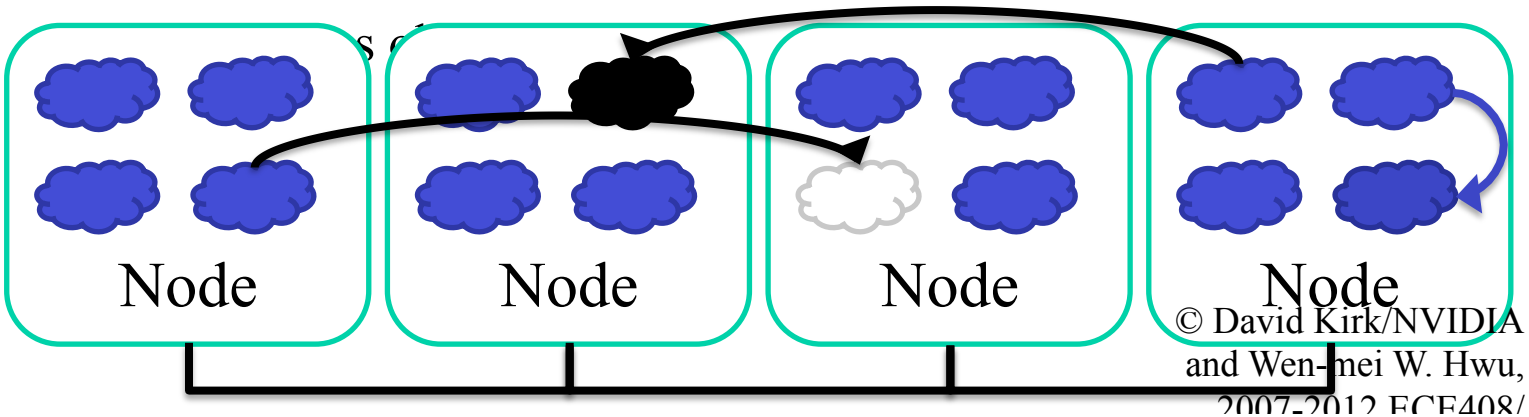


# MPI Receiving Data

- `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)



# 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 */
      *ptr_a = input_a;
      *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);
    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);
```

# 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];
    }

    /* 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);
}
```

# MPI Barriers

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

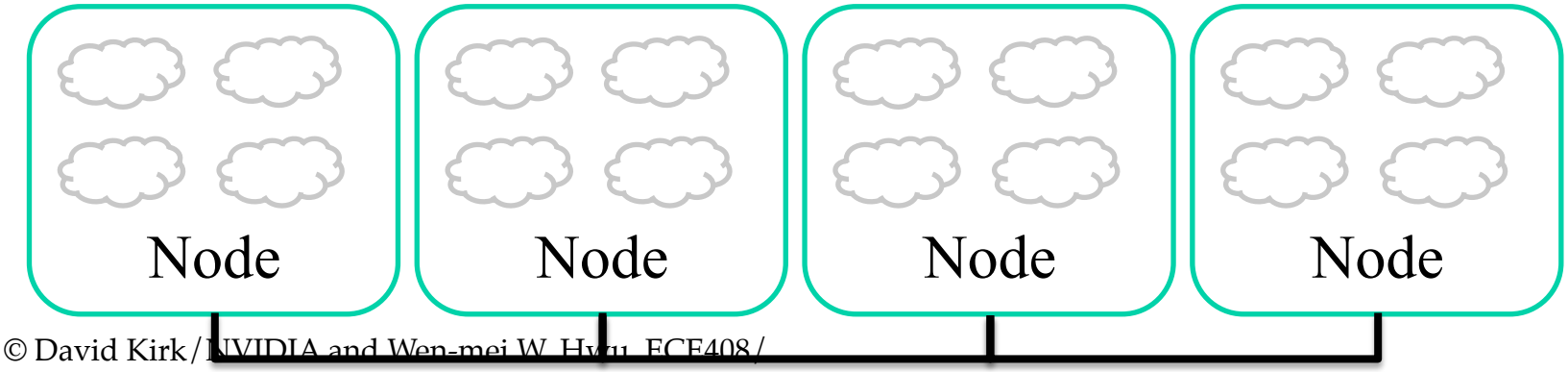
- Blocks the caller until all group members have called it; the call returns at any process only after all group members have entered the call.

# MPI Barriers

- Wait until all other processes in the MPI group reach the same barrier
  1. All processes are executing Do_Stuff()
  2. 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);
}
```

# ADDING CUDA TO MPI

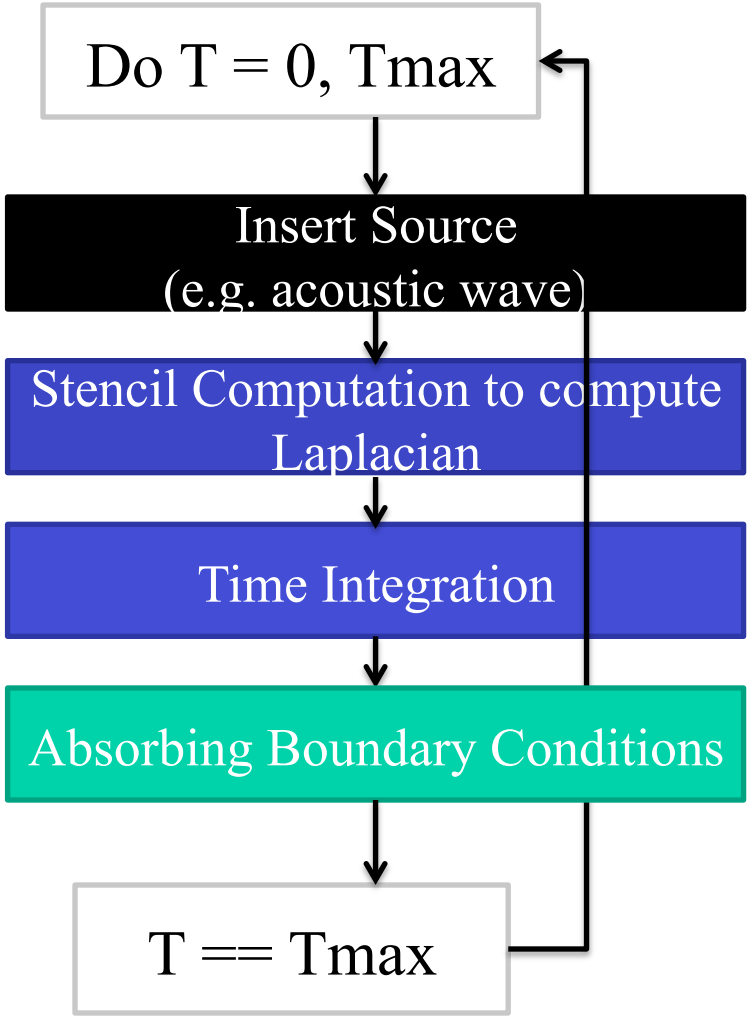
# Vector Addition: CUDA Process (I)

```
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;

    /* Allocate memory */
    gmacMalloc((void **)&input_a, num_bytes);
    gmacMalloc((void **)&input_b, num_bytes);
    gmacMalloc((void **)&output, 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);
```

# Vector Addition: CUDA Process (II)

```
    /* Compute the partial vector addition */
    dim3 Db(BLOCK_SIZE);
    dim3 Dg((vector_size + BLOCK_SIZE - 1) / BLOCK_SIZE);
    vector_add_kernel<<<Dg, Db>>>(gmacPtr(output), gmacPtr(input_a),
                                  gmacPtr(input_b), vector_size);
    gmacThreadSynchronize();

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

    /* Release device memory */
    gmacFree(d_input_a);
    gmacFree(d_input_b);
    gmacFree(d_output);
}
```

# A Typical Wave Propagation Application

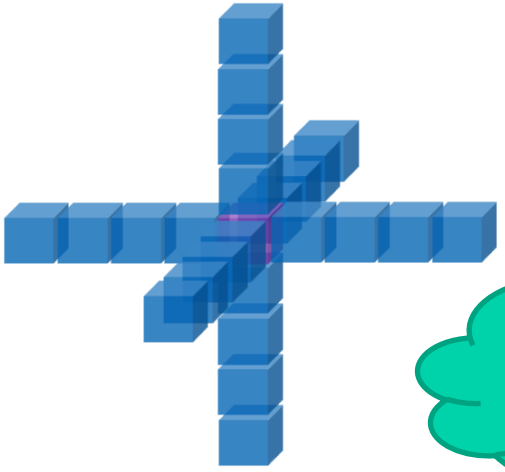


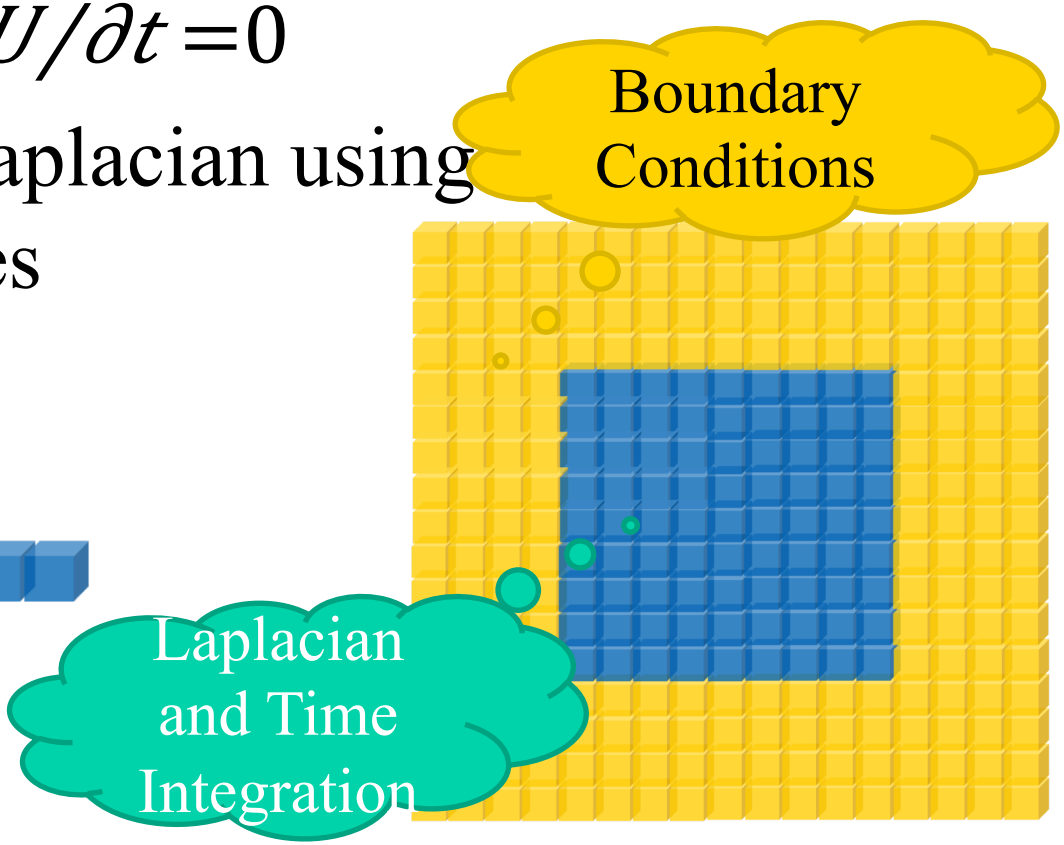
# Review of Stencil Computations

- Example: wave propagation modeling

$\nabla^2 U - 1/v^2 \ \partial U/\partial t = 0$

- Approximate Laplacian using finite differences





# Wave Propagation: Kernel Code

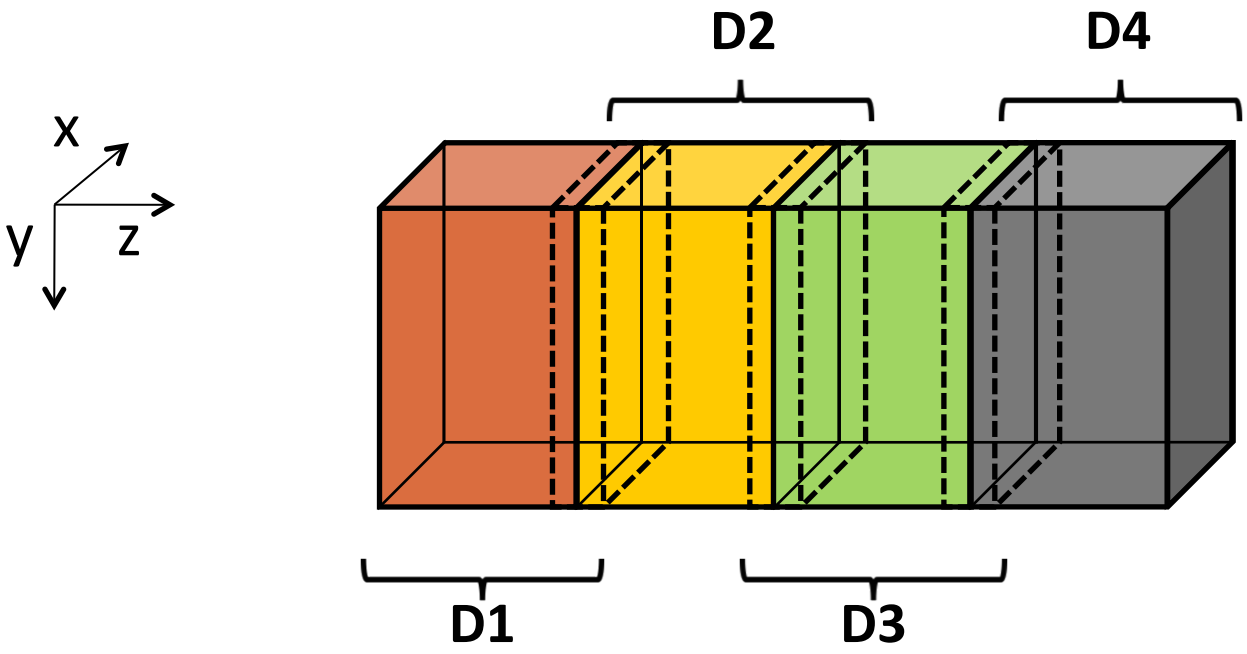
```
/* Coefficients used to calculate the laplacian */
                    coeff[5];

           void wave_propagation(      *next,       *in,
                                           *prev,       *velocity,
dim)
{
              x = threadIdx.x + blockIdx.x * blockDim.x;
              y = threadIdx.y + blockIdx.y * blockDim.y;
              z = threadIdx.z + blockIdx.z * blockDim.z;

    /* Point index in the input and output matrixes */
              n = x + y * dim.z + z * dim.x * dim.y;

    /* Only compute for points within the matrixes */
    if(x < dim.x && y < dim.y && z < dim.z) {

        /* Calculate the contribution of each point to the laplacian */
              laplacian = coeff[0] + in[n];
```

# Wave Propagation: Kernel Code

```
        for(int i = 1; i < 5; ++i) {
            laplacian += coeff[i] *
                                (in[n – i] + /* Left */
                                in[n + i] + /* Right */
                                in[n – i * dim.x] + /* Top */
                                in[n + I * dim.x] + /* Bottom */
                                in[n – i * dim.x * dim.y] + /* Behind */
                                in[n + i * dim.x * dim.y]);  /* Front */
        }

        /* Time integration */
        next[n] = velocity[n] * laplacian + 2 * in[n] – prev[n];
    }
}
```

# Stencil Domain Decomposition

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



# Wave Propagation: Main Process

```
int main(int argc, char *argv[]) {
    int pad = 0, dimx  = 480+pad, dimy  = 480, dimz  = 400, nreps = 100;
    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("Nedded 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_node(dimx, dimy, dimz / (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz, nreps );

    MPI_Finalize();
    return 0;
}
```

# Stencil Code: Server Process (I)

```
void data_server(int dimx, int dimy, int dimz, int nreps) {
    int np, num_comp_nodes = np – 1, first_node = 0, last_node = np - 2;
    unsigned int num_points = dimx * dimy * dimz;
    unsigned int num_bytes  = num_points * sizeof(     );
          *input=0, *output = NULL, *velocity = NULL;
    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    velocity = (float *)malloc(num_bytes);
    if(input == NULL || output == NULL || velocity == NULL) {
        printf(“Server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data and velocity */
    random_data(input, dimx, dimy ,dimz , 1, 10);
    random_data(velocity, dimx, dimy ,dimz , 1, 10);
```

# Stencil Code: Server Process (II)

```
/* Calculate number of shared points */
int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
      *input_send_address = input;

/* Send input data to the first compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, first_node,
        DATA_DISTRIBUTE, MPI_COMM_WORLD );
send_address += dimx * dimy * (dimz / num_comp_nodes - 4);

/* Send input data to "internal" compute nodes */
for(int process = 1; process < last_node; process++) {
    MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
            DATA_DISTRIBUTE, MPI_COMM_WORLD);
    send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Send input data to the last compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, last_node,
        DATA_DISTRIBUTE, MPI_COMM_WORLD);
```

# Stencil Code: Server Process (II)

```
      *velocity_send_address = velocity;

/* Send velocity data to compute nodes */
for(int process = 0; process < last_node + 1; process++) {
    MPI_Send(send_address, edge_num_points, MPI_FLOAT, process,
             DATA_DISTRIBUTE, MPI_COMM_WORLD);
    send_address += dimx * dimy * (dimz / num_comp_nodes);
}

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

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

# Stencil Code: Server Process (III)

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

    /* Release resources */
    free(input);
    free(velocity);
    free(output);
}
```

# Stencil Code: Compute Process (I)

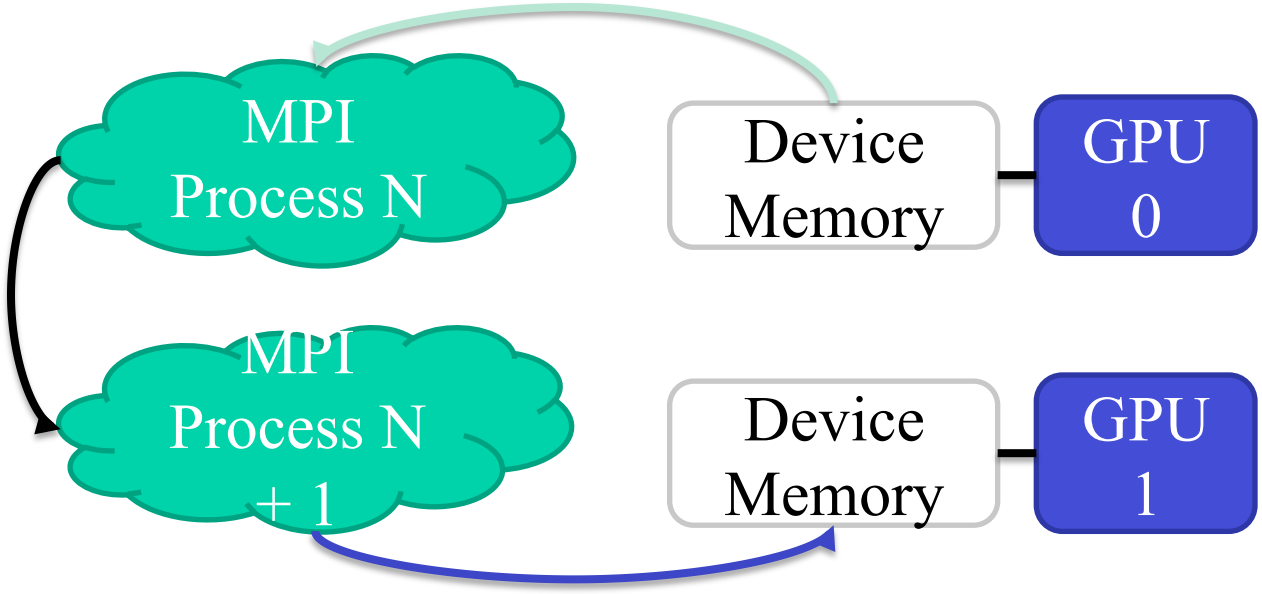
```
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
    int np, pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    unsigned int num_points       = dimx * dimy * (dimz + 8);
    unsigned int num_bytes        = num_points * sizeof(float);
    unsigned int num_ghost_points = 4 * dimx * dimy;
    unsigned int num_ghost_bytes  = num_ghost_points * sizeof(float);

    int left_ghost_offset   = 0;
    int right_ghost_offset  = dimx * dimy * (4 + dimz);

          *input = NULL, *output = NULL, *prev = NULL, *v = NULL;

    /* Allocate device memory for input and output data */
    gmacMalloc((void **)&input,  num_bytes);
    gmacMalloc((void **)&output, num_bytes);
    gmacMalloc((void **)&prev, num_bytes);
    gmacMalloc((void **)&v, num_bytes);
```

# Stencil Code: Compute Process (II)

```
MPI_Status status;
int left_neighbor  = (pid > 0) ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;
int server_process = np - 1;

/* Get the input data from server process */
      *rcv_address = input + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
        DATA_DISTRIBUTE, MPI_COMM_WORLD, &status );

/* Get the velocity data from server process */
rcv_address = h_v + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
        DATA_DISTRIBUTE, MPI_COMM_WORLD, &status );
```

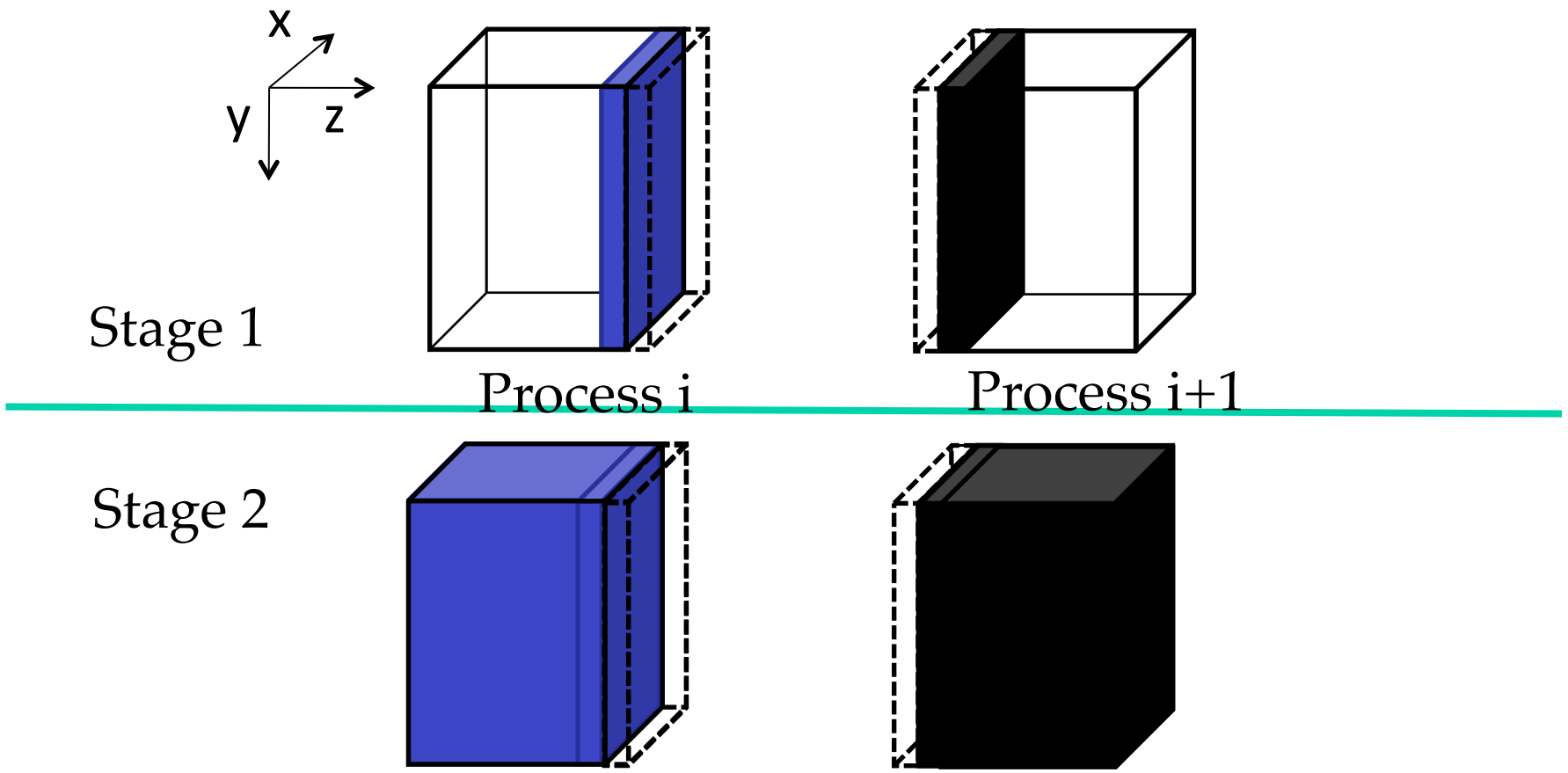
# CUDA and MPI Communication

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



# GETTING SOME OVERLAP

# Overlapping Computation with Communication



# Data Server Process code (I).

```
     data_server(int dimx, int dimy, int dimz, int nreps) {
1.   int np,
/* Set MPI Communication Size */
2.   MPI_Comm_size(MPI_COMM_WORLD, &np);

3.   num_comp_nodes = np – 1, first_node = 0, last_node = np - 2;
4.                num_points = dimx * dimy * dimz;
5.                num_bytes  = num_points * sizeof(     );
6.          *input=0, *output=0;
          /* Allocate input data */
7.   input = (float *)malloc(num_bytes);
8.   output = (float *)malloc(num_bytes);
9.   if(input == NULL || output == NULL) {
         printf("server couldn't allocate memory\n");
         MPI_Abort( MPI_COMM_WORLD, 1 );
     }
     /* Initialize input data */
10.  random_data(input, dimx, dimy ,dimz , 1, 10);
     /* Calculate number of shared points */
11.  int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
12.  int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
13.  float *send_address = input;
```

# Data Server Process Code (II)

```
    /* Send data to the first compute node */
14. MPI_Send(send_address, edge_num_points, MPI_FLOAT, first_node,
             0, MPI_COMM_WORLD );

15. send_address += dimx * dimy * (dimz / num_comp_nodes - 4);
    /* Send data to "internal" compute nodes */
16. for(int process = 1; process < last_node; process++) {
17.     MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
                 0, MPI_COMM_WORLD);
18.     send_address += dimx * dimy * (dimz / num_comp_nodes);
    }

    /* Send data to the last compute node */
19. MPI_Send(send_address, edge_num_points, MPI_FLOAT, last_node,
             0, MPI_COMM_WORLD);
```

# Compute Process Code (I).

```
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
    int np, pid;
1.  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
2.  MPI_Comm_size(MPI_COMM_WORLD, &np);
3.  int server_process = np - 1;


4.  unsigned int num_points       = dimx * dimy * (dimz + 8);
5.  unsigned int num_bytes        = num_points * sizeof(float);
6.  unsigned int num_halo_points = 4 * dimx * dimy;
7.  unsigned int num_halo_bytes  = num_halo_points * sizeof(float);

    /* Alloc host memory */
8.  float *h_input  = (      *)malloc(num_bytes);
        /* Alloca device memory for input and output data */
9.  float *d_input = NULL;
10. cudaMalloc((     **)&d_input,  num_bytes );
11.       *rcv_address = h_input + num_halo_points * (0 == pid);
12. MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status );
13. cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);
}
```

# Stencil Code: Kernel Launch

```
void launch_kernel(float *next, float *in, float *prev, float *velocity,
                   int dimx, int dimy, int dimz)
{
    dim3 Gd, Bd, Vd;

    Vd.x = dimx; Vd.y = dimy; Vd.z = dimz;

    Bd.x = BLOCK_DIM_X; Bd.y = BLOCK_DIM_Y; Bd.z = BLOCK_DIM_Z;

    Gd.x = (dimx + Bd.x - 1) / Bd.x;
    Gd.y = (dimy + Bd.y - 1) / Bd.y;
    Gd.z = (dimz + Bd.z - 1) / Bd.z;

    wave_propagation<<<Gd, Bd>>>(next, in, prev, velocity, Vd);
}
```

# 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 (Status). This refers to the receive operation.

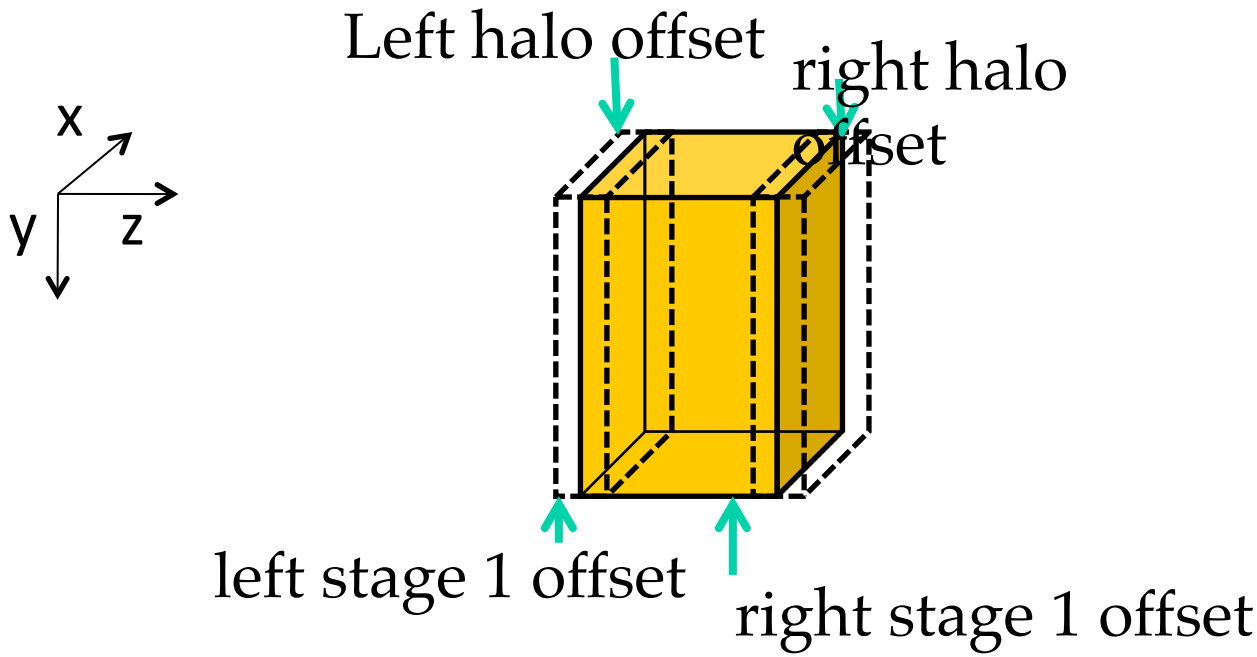
# MPI Sending and Receiving Data

- `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 (Status). This refers to the receive operation.

# Compute Process Code (II)

```
14. float *h_output = NULL, *d_output = NULL, *d_vsq = NULL;
15. float *h_output = (float *)malloc(num_bytes);
16. cudaMalloc((void **)&d_output, num_bytes );

17. float *h_left_boundary = NULL, *h_right_boundary = NULL;
18. float *h_left_halo = NULL, *h_right_halo = NULL;

    /* Alloc host memory for halo data */
19. cudaHostAlloc((void **)&h_left_boundary, num_halo_bytes,
cudaHostAllocDefault);
20. cudaHostAlloc((void **)&h_right_boundary,num_halo_bytes,
cudaHostAllocDefault);
21. cudaHostAlloc((void **)&h_left_halo,     num_halo_bytes,
cudaHostAllocDefault);
22. cudaHostAlloc((void **)&h_right_halo,    num_halo_bytes,
   cudaHostAllocDefault);

    /* Create streams used for stencil computation */
23. cudaStream_t stream0, stream1;
24. cudaStreamCreate(&stream0);
25. cudaStreamCreate(&stream1);
```

# Device Memory Offsets Used for Data Exchange with Neighbors



# Compute Process Code (III)

```
26. MPI_Status status;
27. int left_neighbor  = (pid > 0)    ? (pid - 1) : MPI_PROC_NULL;
28. int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;

    /* Upload stencil cofficients */
    upload_coefficients(coeff, 5);

29. int left_halo_offset   = 0;
30. int right_halo_offset  = dimx * dimy * (4 + dimz);
31. int left_stage1_offset  = 0;
32. int right_stage1_offset = dimx * dimy * (dimz - 4);
33. int stage2_offset       = num_halo_points;

34. MPI_Barrier( MPI_COMM_WORLD );
35. for(int i=0; i < nreps; i++) {
        /* Compute boundary values needed by other nodes first */
36.     launch_kernel(d_output + left_stage1_offset,
            d_input + left_stage1_offset, dimx, dimy, 12, stream0);
37.     launch_kernel(d_output + right_stage1_offset,
            d_input + right_stage1_offset, dimx, dimy, 12, stream0);

        /* Compute the remaining points */
38.     launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                      dimx, dimy, dimz, stream1);
```

# Compute Process Code (IV)

```
        /* Copy the data needed by other nodes to the host */
39.     cudaMemcpyAsync(h_left_boundary, d_output + num_halo_points,
                        num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
40.     cudaMemcpyAsync(h_right_boundary,
                            d_output + right_stage1_offset + num_halo_points,
                            num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
41.     cudaStreamSynchronize(stream0);
```

# Syntax for MPI_Sendrecv()

- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, *recvbuf, int recvcount, MPI_Datatype recvtype, 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 (Status). This refers to the receive operation.

# Compute Process Code (V)

```
        /* Send data to left, get data from right */
42.     MPI_Sendrecv(h_left_boundary, num_halo_points, MPI_FLOAT,
                     left_neighbor,  i, h_right_halo,
                     num_halo_points, MPI_FLOAT, right_neighbor, i,
                     MPI_COMM_WORLD, &status );
        /* Send data to right, get data from left */
43.     MPI_Sendrecv(h_right_boundary, num_halo_points, MPI_FLOAT,
                     right_neighbor, i, h_left_halo,
                     num_halo_points, MPI_FLOAT, left_neighbor,  i,
                     MPI_COMM_WORLD, &status );

44.     cudaMemcpyAsync(d_output+left_halo_offset,  h_left_halo,
                     num_halo_bytes, cudaMemcpyHostToDevice, stream0);
45.     cudaMemcpyAsync(d_output+right_ghost_offset, h_right_ghost,
                     num_halo_bytes, cudaMemcpyHostToDevice, stream0 );
46.     cudaDeviceSynchronize();

47.     float *temp = d_output;
48.     d_output = d_input; d_input = temp;
    }
```

# Compute Process Code (VI)

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

50.       *temp = d_output;
51. d_output = d_input;
52. d_input = temp;

    /* Send the output, skipping halo points */
53. cudaMemcpy(h_output, d_output, num_bytes, cudaMemcpyDeviceToHost);
          *send_address = h_output + num_ghost_points;
54. MPI_Send(send_address, dimx * dimy * dimz, MPI_REAL,
             server_process, DATA_COLLECT, MPI_COMM_WORLD);
55. MPI_Barrier(MPI_COMM_WORLD);

    /* Release resources */
56. free(h_input); free(h_output);
57. cudaFreeHost(h_left_ghost_own); cudaFreeHost(h_right_ghost_own);
58. cudaFreeHost(h_left_ghost); cudaFreeHost(h_right_ghost);
59. cudaFree( d_input ); cudaFree( d_output );
}
```

# Data Server Code (III)

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

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

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

    /* Release resources */
24. free(input);
25. free(output);
}
```

# MPI Message Types

- Point-to-point communication
  - Send and Receive
- Collective communication
  - Barrier
  - Broadcast
  - Reduce
  - Gather and Scatter