Chapter-18
Programming a heterogeneous computing cluster

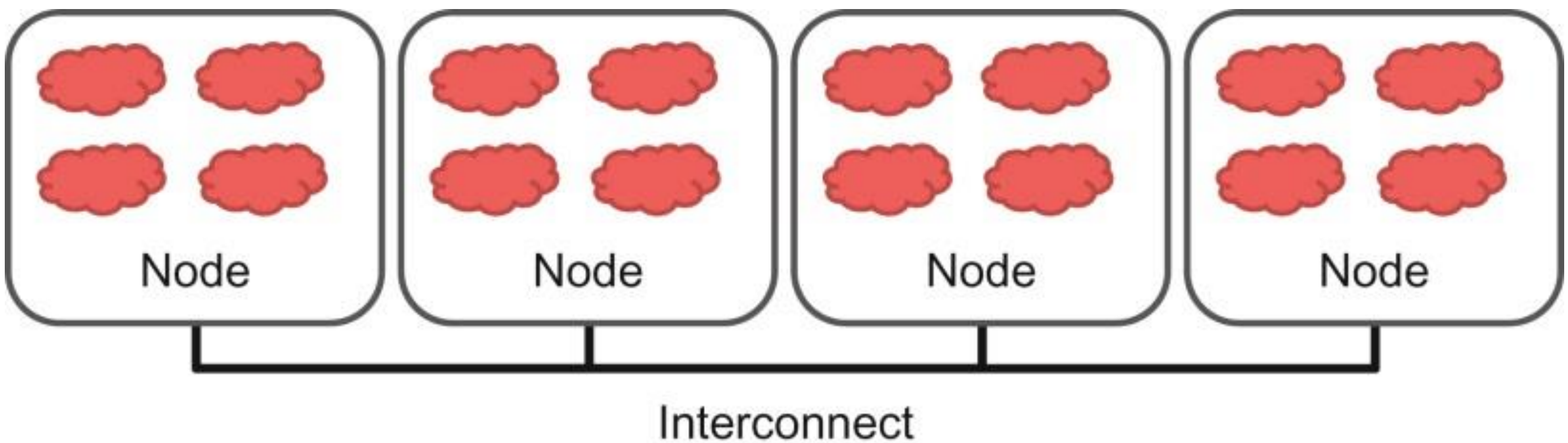

FIGURE 18.1:Programer's view of MPI processes.

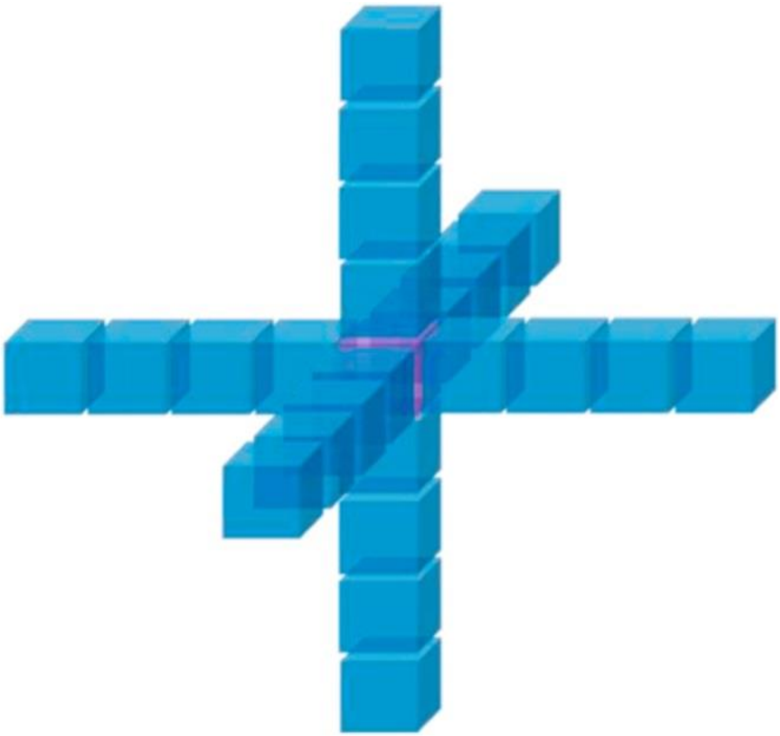

FIGURE 18.2: A 25-stencil computation example, with neighbors in the *x*, *y*, *z* directions.

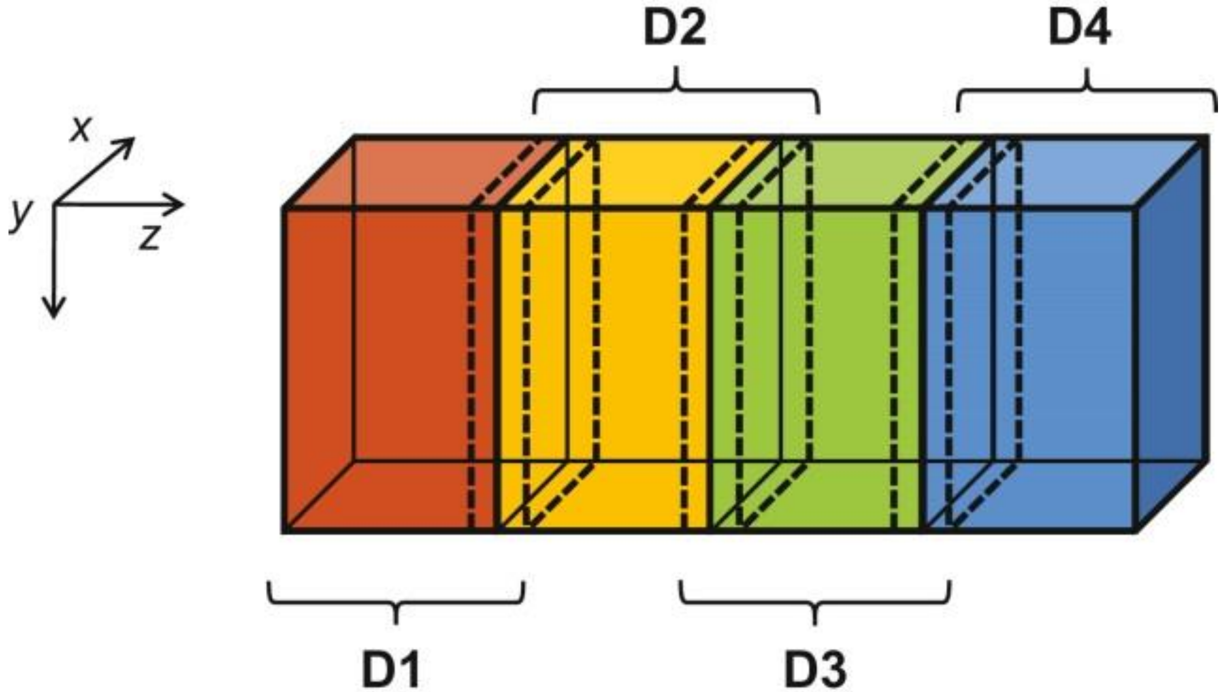

FIGURE 18.3: 3D grid array for the modeling heat transfer in a duct.

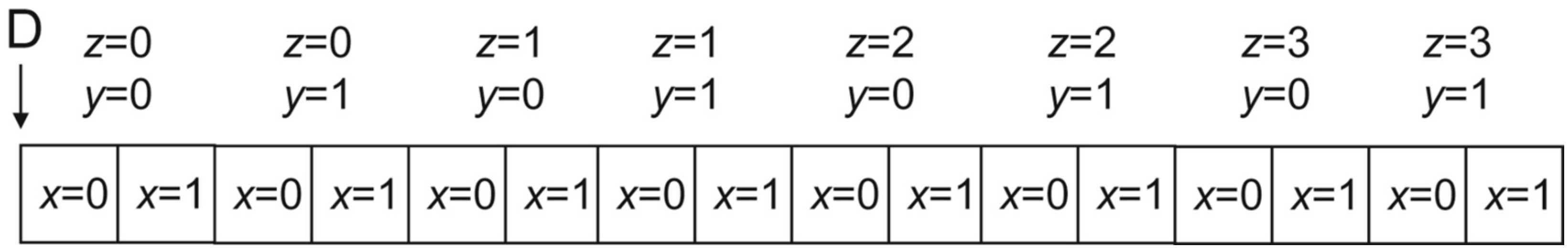

FIGURE 18.4: A small example of memory layout for the 3D grid.

- int MPI_Init (int*argc, char***argv)
  - Initialize MPI
- 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
- int MPI_Comm_abort (MPI_Comm comm)
  - Terminate MPI comminication connection with an error flag
- int MPI_Finalize ( )
  - Ending an MPI application, close all resources

FIGURE 18.5: Five basic MPI functions for establishing and closing a communication system.

```
#include "mpi.h"

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("Needed 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_process(dimx, dimy, dimz/ (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz, nreps);

    MPI_Finalize();
    return 0;
}
```

FIGURE 18.6: A simple MPI main program.

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

FIGURE 18.7: Syntax for the MPI_Send() function.

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

FIGURE 18.8: Syntax for the MPI_Recv() function.

```c
void data_server(int dimx, int dimy, int dimz, int nreps) {
    int np,
    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &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(float);
    float *input=0, *output=0;
    /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    if(input == NULL || output == NULL) {
         printf("server couldn't allocate memory\n");
         MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data */
    random_data(input, dimx, dimy ,dimz , 1, 10);
    /* 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);
    float *send_address = input;
```

FIGURE 18.9: Data server process code (Part 1).

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

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

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

FIGURE 18.10: Data server process code (Part 2).

```
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);
     int server_process = np - 1;


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

     /* Alloc host memory */
     float *h_input  = (float *)malloc(num_bytes);
     /* Alloc device memory for input and output data */
     float *d_input = NULL;
     cudaMalloc((void **)&d_input,  num_bytes );
     float *rcv_address = h_input + num_halo_points * (0 == pid);
     MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status );
```

FIGURE 18.11: Compute process code (Part 1).

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

float *h_left_boundary = NULL, *h_right_boundary = NULL;
float *h_left_halo = NULL, *h_right_halo = NULL;

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

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

FIGURE 18.12: Compute process code (Part 2).

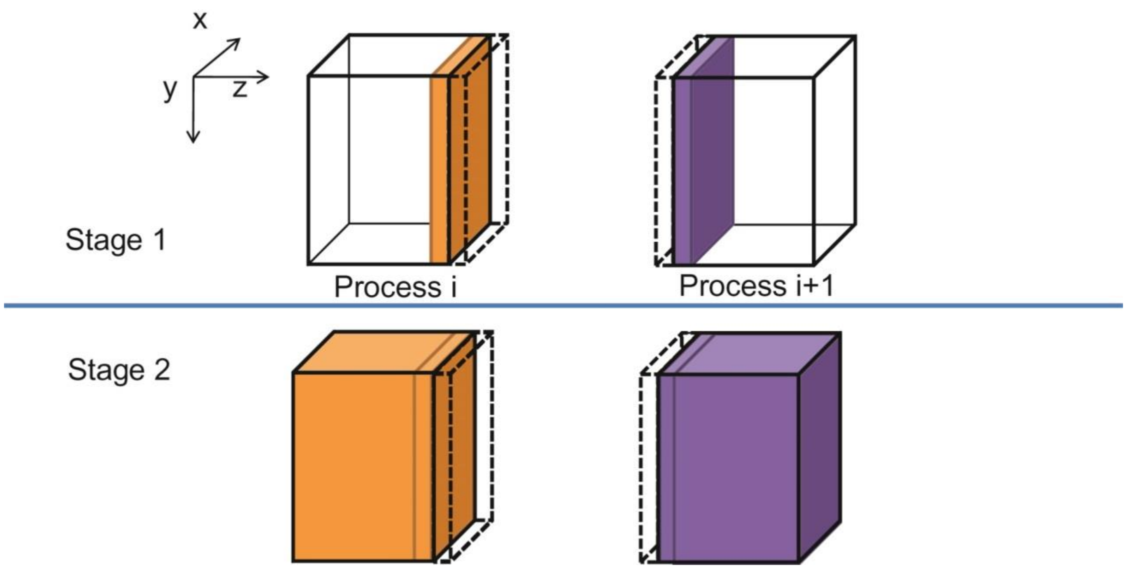

FIGURE 18.13: A two-stage strategy for overlapping computation with communication.

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

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

int left_halo_offset    = 0;
int right_halo_offset   = dimx * dimy * (4 + dimz);
int left_stage1_offset  = 0;
int right_stage1_offset = dimx * dimy * (dimz - 4);
int stage2_offset       = num_halo_points;

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

    /* Compute the remaining points */
    launch_kernel(d_output + stage2_offset, d_input +
stage2_offset,
```

FIGURE 18.14: Compute process code (Part 3).

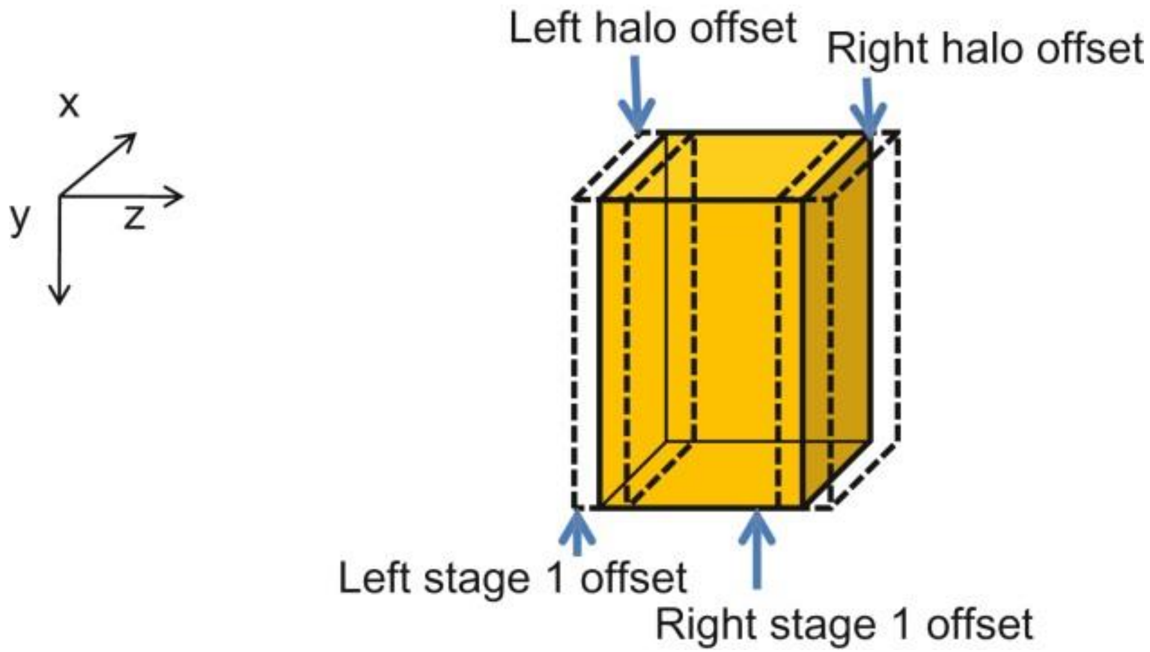

FIGURE 18.15: Device memory offsets used for data exchange with neighbor processes.

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

FIGURE 18.16: Compute process code (Part 4).

- 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.

FIGURE 18.17: Syntax for the MPI_Sendrecv() function.

```
/* Send data to left, get data from right */
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 */
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 );

cudaMemcpyAsync(d_output+left_halo_offset,  h_left_halo,
             num_halo_bytes, cudaMemcpyHostToDevice, stream0);
cudaMemcpyAsync(d_output+right_ghost_offset, h_right_ghost,
             num_halo_bytes, cudaMemcpyHostToDevice, stream0 );
cudaDeviceSynchronize();

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

FIGURE 18.18: Compute process code (Part 5).

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

    float *temp = d_output;
    d_output = d_input;
    d_input = temp;

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

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

FIGURE 18.19: Compute process code (Part 6).

```
    /* 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_REAL, process,
           DATA_COLLECT, MPI_COMM_WORLD, &status );

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

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

FIGURE 18.20: Data server code (Part 3).


```
MPI_SendRecv(d_output + num_halo_points, num_halo_points, MPI_FLOAT,
          left_neighbor, i, d_output + left_halo_offset, num_halo_points,
          MPI_FLOAT, right_neighbor, i, MPI_COMM_WORLD, &status);
MPI_SendRecv(d_output + right_stage1_offset, num_halo_points,
          num_halo_points, MPI_FLOAT, right_neighbor, i,
          d_output + right_halo_offset, num_halo_points,
          MPI_FLOAT, left_neighbor, i, MPI_COMM_WORLD, &status);
```

FIGURE 18.21: Revised MPI SendRec calls when using CUDA-aware MPI.