CHAPTER

18 Programming a heterogeneous computing cluster

Isaac Gelado and Javier Cabezas

CHAPTER OUTLINE

So far, we have focused on programming a heterogeneous computing system with one host and one device. In high-performance computing (HPC), applications require the aggregate computing power of a cluster of computing nodes. Many of the HPC clusters today have one or more hosts and one or more devices in each node. Historically, these clusters have been programmed predominately with message passing interface (MPI). In this chapter, we will present an introduction to joint MPI/CUDA Programming. The reader should be able to easily extend the material to joint MPI/OpenCL, MPI/OpenACC, and so on. We will only present the MPI concepts that programmers need to understand in order to scale their heterogeneous applications to multiple nodes in a cluster environment. In particular, we will focus on domain partitioning, point-to-point communication, and collective communication in the context of scaling a CUDA kernel into multiple nodes.

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00018-2

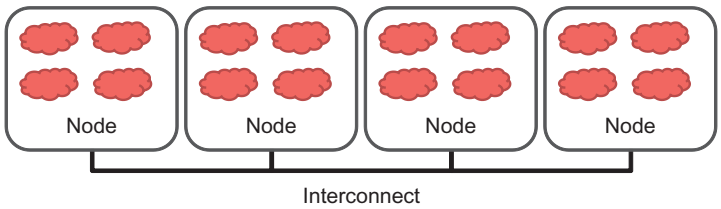

FIGURE 18.1

Programer's view of MPI processes.

18.1 BACKGROUND

While there was practically no top supercomputer using GPUs before 2009, the need for better energy efficiency has led to fast adoption of GPUs in the recent years. Many of the top supercomputers in the world today use both CPUs and GPUs in each node. The effectiveness of this approach is validated by their high rankings in the Green500 list, which reflects their high energy efficiency.

The dominating programing interface for computing clusters today is MPI [Gropp 1999], which is a set of API functions for communication between processes running in a computing cluster. MPI assumes a distributed memory model where processes exchange information by sending messages to each other. When an application uses API communication functions, it does not need to deal with the details of the interconnect network. The MPI implementation allows the processes to address each other using logical numbers, much the same way as using phone numbers in a telephone system: telephone users can dial each other using phone numbers without knowing exactly where the called person is and how the call is routed.

In a typical MPI application, data and work are partitioned among processes. As shown in Fig. 18.1, each node can contain one or more processes, shown as clouds within nodes. As these processes progress, they may need data from each other. This need is satisfied by sending and receiving messages. In some cases, the processes also need to synchronize with each other and generate collective results when collaborating on a large task. This is done with MPI's collective API functions.

18.2 A RUNNING EXAMPLE

We will use a 3D stencil computation similar to that introduced in Chapter 7, Parallel patterns: convolution, as a running example. We assume that the computation calculates heat transfer based on a finite difference method for solving a partial differential equation that describes the physical laws of heat transfer. In particular, we will use the Jacobi Iterative Method where in each iteration or time step, the value of a grid point is calculated as a weighted sum of neighbors (north, east, south, west, up, down) and

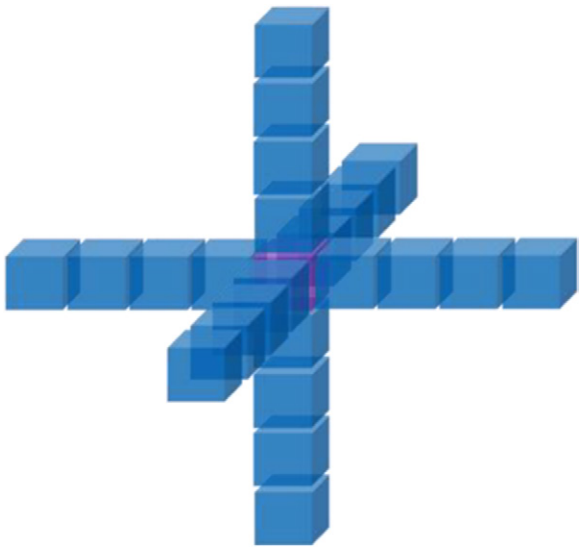

FIGURE 18.2

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

its own value from the previous time step. In order to achieve high numerical stability, multiple indirect neighbors in each direction are also used in the computation of a grid point. This is referred to as a *higher order stencil* computation. For the purpose of this chapter, we assume that four points in each direction will be used.

As shown in Fig. 18.2, there are a total of 24 neighbor points for calculating the next step value of a grid point. In Fig. 18.2, each point in the grid has an x, y, and z coordinate. For a grid point where the coordinate value is $x = i$, $y = j$, and $z = k$, or (i,j,k) its 24 neighbors are $(i-4,j,k)$, $(i-3,j,k)$, $(i-2,j,k)$, $(i-1,j,k)$, $(i+1,j,k)$, $(i+2,j,k)$, $(i+3,j,k)$, $(i+4,j,k)$, $(i,j-4,k)$, $(i,j-3,k)$, $(i,j-2,k)$, $(i,j-1,k)$, $(i,j+1,k)$, $(i,j+2,k)$, $(i,j+3,k)$, $(i,j+4,k)$, $(i,j,k-4)$, $(i,j,k-3)$, $(i,j,k-2)$, $(i,j,k-1)$, $(i,j,k+1)$, $(i,j,k+2)$, $(i,j,k+3)$ and $(i,j,k+4)$. Since the data value of each grid point for the next time step is calculated based on the current data values of 25 points (24 neighbors and itself), the type of computation is often called 25-stencil computation.

We assume that the system is modeled as a structured grid, where spacing between grid points is constant within each direction. This allows us to use a 3D array where each element stores the state of a grid point. The physical distance between adjacent elements in each dimension can be represented by a spacing variable. Note that this grid data structure is similar to that used in the electrostatic potential calculation in Chapter 15, Application case study—molecular visualization and analysis. Fig. 18.3 illustrates a 3D array that represents a rectangular ventilation duct, with x and y dimensions as the cross-sections of the duct and the z dimension the direction of the heat flow along the duct.

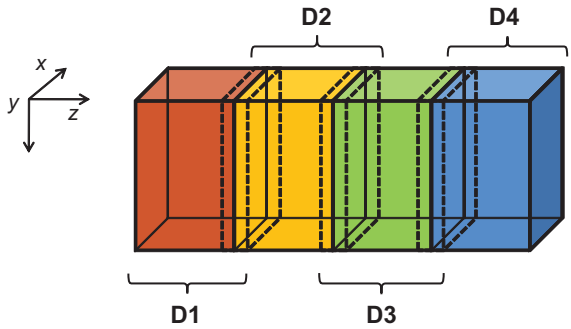

FIGURE 18.3

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

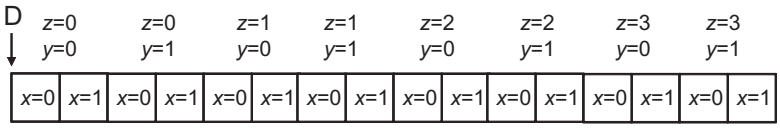

FIGURE 18.4

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

We assume that the data is laid out in the memory space so that x is the lowest dimension, y is the next, and z is the highest. That is, all elements with y=0 and z=0 will be placed in consecutive memory locations according to their x coordinate. Fig. 18.4 shows a small example of the grid data layout. This small example has only 16 data elements in the grid: two elements in the x dimension, two in the y dimension, and four in the z dimension. Both x elements with y=0 and z=0 are placed in memory first. They are followed by all elements with y=1 and z=0. The next group will be elements with y=0 and z=1. The reader should verify that this is simply a 3D generalization of the row-major layout convention of C/C++ discussed in Chapter 3, Scalable parallel execution.

When one uses a computing cluster, it is common to divide the input data into several partitions, called domain partitions, and assign each partition to a node in the cluster. In Fig. 18.3, we show that the 3D array is divided into four domain partitions: D1, D2, D3, and D4. Each of the partitions will be assigned to an MPI compute process.

The domain partitions can be further illustrated with Fig. 18.4. The first section, or slice, of four elements (z=0) in Fig. 18.4 are in the first partition, the second section (z=1) the second partition, the third section (z=2) the third partition, and the fourth section (z=3) the fourth partition. This is obviously a toy example. In a real application, there are typically hundreds or even thousands of elements in each dimension. For the rest of this chapter, it is useful to remember that all elements in a z slice are in consecutive memory locations.

18.3 MESSAGE PASSING INTERFACE BASICS

Like CUDA, MPI programs are based on the SPMD parallel execution model. All MPI processes execute the same program. The MPI system provides a set of API functions to establish communication systems that allow the processes to communicate with each other. Fig. 18.5 shows five essential MPI functions that set up and tear down the communication system for an MPI application. We will use a simple MPI program shown in Fig. 18.6 to illustrate the usage of these API functions. To launch an MPI application in a cluster, a user needs to supply the executable file of the program to the *mpirun* command or the *mpiexec* command in a cluster.

Each process starts by initializing the MPI runtime with an `MPI_Init()` call. This initializes the communication system for all the processes running the application. Once the MPI runtime is initialized, each process calls two functions to prepare for communication. The first function is `MPI_Comm_rank()` that returns a unique number to calling each process, which is called the *MPI rank* or process id for the process. The numbers received by the processes vary from 0 to the number of processes − 1. An MPI rank for a process is equivalent to the expression `blockIdx.x*blockDim.x+threadIdx.x` for a CUDA thread. It uniquely identifies the process in a communication, similar to the phone number in a telephone system.

The `MPI_Comm_rank()` function takes two parameters. The first one is an MPI built-in type `MPI_Comm` that specifies the scope of the request. Each variable of the `MPI_Comm` type is commonly referred to as a communicator. `MPI_Comm` and other MPI built-in types are defined in "`mpi.h`" header file that should be included in all C program files that use MPI. This is similar to the "`cuda.h`" header file for CUDA programs. An MPI application can create one or more *communicators* each of which is a group of MPI processes for the purpose of communication. `MPI_Comm_rank()` assigns a unique id to each process in a communicator. In Fig. 18.6, the parameter value passed is `MPI_COMM_WORLD`, which means that the communicator includes all MPI processes running the application.[1]

- 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.

[1] Interested readers should refer to the MPI reference manual [Gropp 1999] for details on creating and using multiple communicators in an application, in particular the definition and use of intracommunicators and intercommunicators.

```
#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.

The second parameter to the `MPI_Comm_rank()` function is a pointer to an integer variable into which the function will deposit the returned rank value. In Fig. 18.6, a variable `pid` is declared for this purpose. After the `MPI_Comm_rank()` returns, the `pid` variable will contain the unique id for the calling process.

The second API function is `MPI_Comm_size()`, which returns the total number of MPI processes running in the communicator. The `MPI_Comm_size()` function takes two parameters. The first one is of `MPI_Comm` type and gives the scope of the request. In Fig. 18.6, the parameter value passed in is `MPI_COMM_WORLD`, which means the scope of the `MPI_Comm_size()` is all the processes of the application. Since the scope is all MPI processes, the returned value is the total number of MPI processes running the application. This is a value requested by a user when the application is submitted using the mpirun command or the mpiexec command. However, the user may not have requested sufficient number of processes. Also, the system may or may not be able to create all the processes requested. Therefore, it is a good practice for an MPI application program to check the actual number of processes running.

The second parameter is a pointer to an integer variable into which the `MPI_Comm_size()` function will deposit the return value. In Fig. 18.6, a variable `np` is declared for this purpose. After the function returns, the variable `np` contains the number of MPI processes running the application. We assume that the application requires at least three MPI processes. Therefore, it checks if the number of processes is at least three. If not, it calls `MPI_Comm_abort()` function to terminate the communication connections and return with an error flag value 1.

Fig. 18.6 also shows a common pattern for reporting errors or other chores. There are multiple MPI processes but we need to report the error only once. The application code designates the process with `pid=0` to do the reporting. This is similar to the pattern in CUDA kernels where some tasks only need to be done by one of the threads in a thread-block.

As shown in Fig. 18.5, the `MPI_Comm_abort()` function takes two parameters. The first sets the scope of the request. In Fig. 18.6, the scope is set as `MPI_COMM_WORLD`, which means all MPI processes running the application. The second parameter is a code for the type of error that caused the abort. Any number other than 0 indicates that an error has happened.

If the number of processes satisfies the requirement, the application program goes on to perform the calculation. In Fig. 18.6, the application uses `np-1` processes (`pid` from 0 to `np-2`) to perform the calculation and one process (the last one whose `pid` is `np-1`) to perform I/O service for the other processes. We will refer to the process that performs the I/O services as the data server and the processes that perform the calculation as compute processes. If the `pid` of a process is within the range from 0 to `np-2`, it is a compute process and call the `compute_process()` function. If the process `pid` is `np-1`, it is the data server and calls `data_server()` function. This is similar to the pattern where threads perform different actions according to their thread ids.

After the application completes its computation, it notifies the MPI runtime with a call to the `MPI_Finalize()`, which frees all MPI communication resources allocated to the application. The application can then exit with a return value 0, which indicates that no error occurred.

18.4 MESSAGE PASSING INTERFACE POINT-TO-POINT COMMUNICATION

MPI supports two major types of communication. The first is point-to-point type, which involves one source process and one destination process. The source process calls the `MPI_Send()` function and the destination process calls the `MPI_Recv()` function. This is analogous to a caller dialing a call and a receiver answering a call in a telephone system.

Fig. 18.7 shows the syntax for using the `MPI_Send()` function. The first parameter is a pointer to the starting location of the memory area where the data to be sent can be found. The second parameter is an integer that gives that number of data elements to be sent. The third parameter is of an MPI built-in type `MPI_Datatype`. It specifies the type of each data element being sent. The `MPI_Datatype` is defined in `mpi.h` and includes `MPI_DOUBLE` (double precision floating point), `MPI_FLOAT`(single precision floating point), `MPI_INT` (integer), and `MPI_CHAR` (character). The exact sizes of these types depend on the size of the corresponding C types in the host processor. See the MPI reference manual for more sophisticated use of MPI types [Gropp 1999].

- 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.

The fourth parameter for `MPI_Send` is an integer that gives the MPI rank of the destination process. The fifth parameter gives a tag that can be used to classify the messages sent by the same process. The sixth parameter is a communicator that selects the processes to be considered in the communication.

Fig. 18.8 shows the syntax for using the `MPI_Recv()` function. The first parameter is a pointer to the area in memory where the received data should be deposited. The second parameter is an integer that gives the maximal number of elements that the `MPI_Recv()` function is allowed to receive. The third parameter is an `MPI_Datatype` that specifies the type (size) of each element to be received. The fourth parameter is an integer that gives the process id of the source of the message.

The fifth parameter is an integer that specifies the particular tag value expected by the destination process. If the destination process does not want to be limited to a particular tag value, it can use `MPI_ANY_TAG`, which means that the receiver is willing to accept messages of any tag value from the source.

We will first use the data server to illustrate the use of point-to-point communication. In a real application, the data server process would typically perform data input

```
void 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.   unsigned int num_points = dimx * dimy * dimz;
5.   unsigned int num_bytes  = num_points * sizeof(float);
6.   float *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;
```

FIGURE 18.9

Data server process code (Part 1).

and output operations for the compute processes. However, input and output have too much system dependent complexity. Since I/O is not the focus of our discussion, we will avoid the complexity of I/O operations in a cluster environment. That is, instead of reading data from a file system, we will just have the data server to initialize the data with random numbers and distribute the data to the compute processes. The first part of the data server code is shown in Fig. 18.9.

The data server function takes four parameters. The first three parameters specify the size of the 3D grid: number of elements in the *x* dimension `dimx`, the number of elements in the *y* dimension `dimy`, and the number of elements in the *z* dimension `dimz`. The fourth parameter specifies the number of iterations that need to be done for all the data points in the grid.

In Fig. 18.9, Line 1 declares variable np that will contain the number of processes running the application. Line 2 calls `MPI_Comm_size()`, which will deposit the information into `np`. Line 3 declares and initializes several helper variables. The variable `num_comp_procs` contains the number of compute processes. Since we are reserving one process as data server, there are `np-1` compute processes. The variable `first_node` gives the process id of the first compute process, which is 0. The variable `last_node` gives the process id of the last compute process, which is `np-2`. That is, Line 3 designates the first `np-1` processes, 0 through `np-2` as compute processes. This

reflects the design decision and the process with the largest rank serves as the data server. This decision will also be reflected in the compute process code.

Line 4 declares and initializes the `num_points` variable that gives the total number of grid data points to be processed, which is simply the product of the number of elements in each dimension, or `dimx * dimy * dimz`. Line 5 declares and initializes the `num_bytes` variable that gives the total number of bytes needed to store all the grid data points. Since each grid data point is a float, this value is `num_points * sizeof(float)`.

Line 6 declares two pointer variables: `input` and `output`. These two pointers will point to the input data buffer and the output data buffer. Lines 7 and 8 allocate memory for the input and output buffers and assign their addresses to their respective pointers. Line 9 checks if the memory allocations were successful. If either of the memory allocation fails, the corresponding pointer will receive a NULL pointer from the `malloc()` function. In this case, the code aborts the application and reports an error.

Lines 11 and 12 calculate the number of grid point array elements that should be sent to each compute process. As shown in Fig. 18.3, there are two types of compute processes. The first process (Process 0) and the last process (Process 3) compute an "edge" partition that has neighbors only on one side. Partition D1 assigned to the first process has neighbor only on the right side (partition D2). Partition D4 assigned to the last process has neighbor only on the left side (partition D3). We call the compute processes that compute edge partitions the *edge processes*.

Each of the rest of the processes computes an internal partition that has neighbors on both sizes. For example, the second process (Process 1) computes a partition (partition D2) that has a left neighbor (partition D1) and a right neighbor (partition D3). We call the processes that compute internal partitions *internal processes*.

Recall that in the Jacobi Iterative Method, each calculation step for a grid point needs the values of its immediate neighbors from the previous step. This creates a need for halo cells for grid points at the left and right boundaries of a partition, shown as slices defined by dotted lines at the edge of each partition in Fig. 18.3. Note that these halo cells are similar to those in convolution pattern presented in Chapter 7, Parallel patterns: convolution. Therefore, each process also needs to receive four slices of halo cells that contains all neighbors for each side of the boundary grid points of its partition. For example, in Fig. 18.3, partition D2 needs four halo slices from D1 and four halo slices from D3. Note that a halo slice for D2 is a boundary slice for D1 or D3.

Recall that the total number of grid points is `dimx*dimy*dimz`. Since we are partitioning the grid along the z dimension, the number of grid points in each partition should be `dimx*dimy*(dimz/num_comp_procs)`. Recall that we will need four neighbor slices in each direction in order to calculate values within each slice. Because we need to send four slices of grid points for each neighbor, the number of grid points that should be sent to each internal process should be `dimx*dimy*((dimz/num_comp_procs) + 8)`. As for an edge process, there is only one neighbor. Like in the case of convolution, we assume that zero values will be used for the ghost cells and no input data needs to be sent for them. For example,

```
    /* 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);
```

FIGURE 18.10

Data server process code (Part 2).

partition D1 only needs the neighbor slice form D2 on the right side. Therefore, the number of grid points to be sent to an edge process is `dimx*dimy*((dimz/num_comp_procs) + 4)`. That is, each process receives four slices of halo grid points from the neighbor partition on each side.

Line 13 of Fig. 18.9 sets the `send_address` pointer to point to the beginning of the input grid point array. In order to send the appropriate partition to each process, we will need to add the appropriate offset to this beginning address for each `MPI_Send()`. We will come back to this point later.

We are now ready to complete the code for the data server, shown in Fig. 18.10. Line 14 sends Process 0 its partition. Since this is the first partition, its starting address is also the starting address of the entire grid, which was set up in Line 13. Process 0 is an edge process and it does not have a left neighbor. Therefore, the number of grid points to be sent is the value `edge_num_points`, i.e., `dimx*dimy*((dimz/num_comp_procs) + 4)`. The third parameter specifies that the type of each element is an `MPI_FLOAT` which is C `float` (single precision, 4 bytes). The fourth parameter specifies that the value of `first_node`, i.e., 0, is the MPI rank of the destination process. The fifth parameter specifies 0 for the MPI tag. This is because we are not using tags to distinguish between messages sent from the data server. The sixth parameter specifies that the communicator to be used for sending the message should be all MPI processes for the current application.

Line 15 of Fig. 18.10 advances the `send_address` pointer to the beginning of the data to be sent to Process 1. From Fig. 18.3, there are `dimx*dimy*(dimz/num_comp_procs)` elements in partition D1, which means D2 starts at location that is `dimx*dimy*(dimz/num_comp_procs)` elements from the starting location of `input`. Recall that we also need to send the halo cells from D1 as well. Therefore, we adjust the starting address for the `MPI_Send()` back by four slices, which results in the expression for advancing the `send_address` pointer in Line 15: `dimx*dimy*((dimz/num_comp_procs) - 4)`.

Line 16 is a loop that sends out the MPI messages to Process 1 through Process np-3. In our small example for four compute processes, np is 5. The loop sends the MPI messages to Processes 1 and 2. These are internal processes. They need to receive halo grid points for neighbors on both sides. Therefore, the second parameter of the MPI_Send() in Line 17 uses int_num_nodes, i.e., dimx*dimy*((dimz/num_comp_procs) + 8). The rest of the parameters are similar to that for the MPI_Send() in Line 14 with the obvious exception that the destination process is specified by the loop variable process, which is incremented from 1 to np-3 (last_node is np-2).

Line 18 advances the send address for each internal process by the number of grid points in each partition: dimx*dimy*dimz/num_comp_nodes. Note that the starting locations of the halo grid points for internal processes are dimx*dimy*dimz/num_comp_procs points apart. Although we need to pull back the starting address by four slices to accommodate halo grid points, we do so for every internal process so the net distance between the starting locations remains as the number of grid points in each partition.

Line 19 sends the data to the Process np-2, the last compute process that has only one neighbor on the left. The reader should be able to reason through all the parameter values used. Note that we are not quite done with the data server code. We will come back later for the final part of the data server that collects the output values from all compute processes.

We now turn our attention to the compute processes that receive the input from the data server process. In Fig. 18.11, Lines 1 and 2 establish the process id for the

```
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  = (float *)malloc(num_bytes);
     /* Alloc device memory for input and output data */
9.   float *d_input = NULL;
10.  cudaMalloc((void **)&d_input,  num_bytes );
11.  float *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 );
```

FIGURE 18.11

Compute process code (Part 1).

process and the total number of processes for the application. Line 3 establishes that the data server is Process `np-1`. Lines 4 and 5 calculates the number of grid points and the number of bytes that should be processed by each internal process. Lines 6 and 7 calculate the number of grid points and the number of bytes in each halo (four slices).

Lines 8–10 allocate the host memory and device memory for the input data. Although the edge processes need less halo data, they still allocate the same amount of memory for simplicity; part of the allocated memory will not be used by the edge processes. Line 11 sets the starting address of the host memory for receiving the input data from the data server. For all compute processes except Process 0, the starting receiving location is simply the starting location of the allocated memory for the input data. However, we adjust the receiving location by four slices. This is because for simplicity, we assume that the host memory for receiving the input data is arranged the same way for all compute processes: four slices of halo from the left neighbor followed by the partition, followed by four slices of halo from the right neighbor. However, we showed in Line 4 of Fig. 18.10, the data server will not send any halo data from the left neighbor to Process 0. That is, for Process 0, the MPI message from the data server only contains the partition and the halo from the right neighbor. Therefore, Line 10 adjusts the starting host memory location by four slices so that Process 0 will correctly interpret the input data from the data server.

Line 12 receives the MPI message from the data server. Most of the parameters should be familiar. The last parameter reflects any error condition that occurred when the data is received. The second parameter specifies that all compute processes will receive the full amount of data from the data server. However, the data server will send less data to Process 0 and Process `np-2`. This is not reflected in the code because `MPI_Recv()` allows the second parameter to specify a larger number of data points than what is actually received and will only place the actual number of bytes received from the sender into the receiving memory. In the case of Process 0, the input data from the data server contain only the partition and the halo from the right neighbor. The received input will be placed by skipping the first four slices of the allocated memory, which should correspond to the halo for the (non-existent) left neighbor. This effect is achieved with the term `num_halo_points*(pid==0)` in Line 11. In the case of Process `np-2`, the input data contain the halo from the left neighbor and the partition. The received input will be placed from the beginning of the allocated memory, leaving the last four slices of the allocated memory unused.

Line 13 copies the received input data to the device memory. In the case of Process 0, the left halo points are not valid. In the case of Process `np-2`, the right halo points are not valid. However, for simplicity, all compute nodes send the full size to the device memory. The assumption is that the kernels will be launched in such a way that these invalid portions will be correctly ignored. After Line 13, all the input data are in the device memory.

Fig. 18.12 shows Part 2 of the compute process code. Lines 14–16 allocate host memory and device memory for the output data. The output data buffer in the device memory will actually be used as a ping-pong buffer with the input data buffer. That

```
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);
```

FIGURE 18.12

Compute process code (Part 2).

is, they will switch roles in each iteration. Recall that we used a similar scheme in the BFS pattern in Chapter 12, Parallel patterns: graph search. We will return to this point later.

We are now ready to present the code that performs computation steps on the grid points.

18.5 OVERLAPPING COMPUTATION AND COMMUNICATION

A simple way to perform the computation steps is for each compute process to perform a computation step on its entire partition, exchange halo data with the left and right neighbors, and repeat. While this is a very simple strategy, it is not very effective. The reason is that this strategy forces the system to be in one of the two modes. In the first mode, all compute processes are performing computation steps. During this time, the communication network is not used. In the second mode, all compute processes exchange halo data with their left and right neighbors. During this time, the computation hardware is not well utilized. Ideally, we would like to achieve better performance by utilizing both the communication network and computation hardware all the time. This can be achieved by dividing the computation tasks of each compute process into two stages, as illustrated in Fig. 18.13.

During the first stage (Stage 1), each compute process calculates its boundary slices that will be needed as halo cells by its neighbors in the next iteration. Let's continue to assume that we use four slices of halo data. Fig. 18.13 shows that the collection of four halo slices as a dashed transparent piece and the four boundary slices as a colored piece. Note that the colored piece of Process `i` will be copied into the dashed piece of Process `i+1` and vice versa during the next communication. For

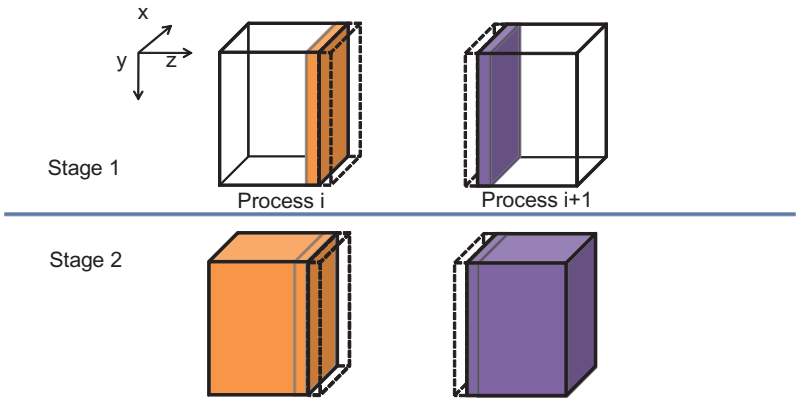

FIGURE 18.13

A two-stage strategy for overlapping computation with communication.

Process 0, the first phase calculates the right four slices of boundary data. For an internal node, it calculates the left four slices and the right four slices of its boundary data. For Process `n-2`, it calculates the left four pieces of its boundary data. The rationale is that these boundary slices are needed by their neighbors for the next iteration. By calculating these boundary slices first, the data can be communicated to the neighbors while the compute processes calculate the rest of its grid points.

During the second stage (Stage 2), each compute process performs two parallel activities. The first is to communicate its new boundary values to its neighbor processes. This is done by first copying the data from the device memory into the host memory, followed by sending MPI messages to the neighbors. As we will discuss later, we need to be careful that the data received from the neighbors are used in the next iteration, not the current iteration. The second activity is to calculate the rest of the data in the partition. If the communication activity takes a shorter amount of time than the calculation activity, we can hide the communication delay and fully utilize the computing hardware all the time. This is usually achieved by having enough slices in the internal part of each partition to allow each compute process to perform computation steps in between communications.

In order to support the parallel activities in Stage 2, we need to use two advanced features of the CUDA Programming model: *pinned memory allocation* and *streams*. A pinned memory allocation requests that the memory allocated will not be paged out by the operating system. This is done with the `cudaHostAlloc()` API call. Lines 19–22, in Fig. 18.12, allocates memory buffers for the left and right boundary slices and the left and right halo slices. The left and right boundary slices need to be sent from the device memory to the left and right neighbor processes. The buffers are used as a host memory staging area for the device to copy data into and then used as the source buffer for `MPI_Send()` to neighbor processes. The left and right halo slices need to be received from neighbor processes. The buffers are used as a host

memory staging area for `MPI_Recv()` to use as destination buffer and then copied to the device memory.

Note that the host memory allocation is done with `cudaHostAlloc()` function rather than the standard `malloc()` function. The difference is that the `cudaHostAlloc()` function allocates a *pinned memory* buffer, sometimes also referred to as *page locked memory* buffer. We need to know a little more background on the memory management in operating systems in order to fully understand the concept of pinned memory buffers.

In a modern computer system, the operating system manages a virtual memory space for applications. Each application has access to a large, consecutive address space. In reality, the system has a limited amount of physical memory that needs to be shared among all running applications. This sharing is performed by partitioning the virtual memory space into pages and mapping only the actively used pages into physical memory. When there is much demand for memory, the operating system needs to "swap out" some of the pages from the physical memory to mass storage such as disks. Therefore, an application may have its data paged out any time during its execution.

The implementation of `cudaMemcpy()` uses a type of hardware called direct memory access (DMA) device. When a `cudaMemcpy()` function is called to copy between the host and device memories, its implementation uses a DMA to complete the task. On the host memory side, the DMA hardware operates on physical addresses. That is, the operating system needs to give a translated physical address to DMA. However, there is a chance that the data may be swapped out before the DMA operation is complete. The physical memory locations for the data may be reassigned to another virtual memory data. In this case, the DMA operation can be potentially corrupted since its data can be overwritten by the paging activity.

A common solution to this data corruption problem is for the CUDA runtime to perform the copy operation in two steps. For a host-to-device copy, the CUDA runtime first copies the source host memory data into a "pinned" memory buffer, which means the memory locations are marked so that the operating paging mechanism will not page out the data. It then uses the DMA device to copy the data from the pinned memory buffer to the device memory. For a device-to-host copy, the CUDA runtime first uses a DMA device to copy the data from the device memory into a pinned memory buffer. It then copies the data from the pinned memory to the destination host memory location. By using an extra pinned memory buffer, the DMA copy will be safe from any paging activities.

There are two problems with this approach. One is that the extra copy adds delay to the `cudaMemcpy()` operation. The second is that the extra complexity involved leads to a synchronous implementation of the `cudaMemcpy()` function. That is, the host program cannot continue to execute until the `cudaMemcpy()` function completes its operation and returns. This serializes all copy operations. In order to support fast copies with more parallelism, CUDA provides a `cudaMemcpyAsync()` function.

In order to use `cudaMemcpyAsync()` function, the host memory buffer must be allocated as a pinned memory buffer. This is done in Lines 19–22 for the host

memory buffers of the left boundary, right boundary, left halo, and right halo slices. These buffers are allocated with the `cudaHostAlloc()` function, which ensures that the allocated memory are pinned or page locked from paging activities. Note that the `cudaHostAlloc()` function takes three parameters. The first two are the same as `cudaMalloc()`. The third specifies some options for more advanced usage. For most basic use cases, we can simply use the default value `cudaHostAllocDefault`.

The second advanced CUDA feature is *streams*, which supports managed concurrent execution of CUDA API functions. A stream is an ordered sequence of operations. When a host code calls a `cudaMemcpyAsync()` function or launches a kernel, it can specify a steam as one of its parameters. All operations in the same stream will be done sequentially. Operations from two different streams can be executed in parallel.

Line 23 of Fig. 18.12 declares two variables that are of CUDA built-in type `cudaStream_t`. Recall that the CUDA built-in types are declared in cuda.h. These variables are then used in calling the `cudaStreamCreate()` function. Each call to the `cudaStreamCreate()` creates a new stream and deposits a pointer to the stream into its parameter. After the calls in Lines 24 and 25, the host code can use either `steram0` or `stream1` in subsequent `cudaMemcpyAsync()` calls and kernel launches.

Fig. 18.14 shows Part 3 of the compute process. Lines 27 and 28 calculate the process id of the left and right neighbors of the compute process. The `left_neighbor` and `right_neighbor` variables will be used by compute processes as parameters when they send message to and receive messages from their neighbors. For Process

```
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,
```

FIGURE 18.14

Compute process code (Part 3).

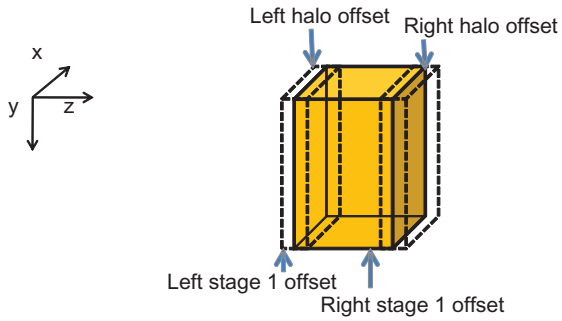

FIGURE 18.15

Device memory offsets used for data exchange with neighbor processes.

0, there is no left neighbor, so Line 27 assigns an MPI constant `MPI_PROC_NULL` to `left_neighbor` to note this fact. For Process `np-2`, there is no right neighbor, so Line 28 assigns `MPI_PROC_NULL` to `right_neighbor`. For all the internal processes, Line 27 assigns `pid-1` to `left_neighbor` and `pid+1` to `right_neighbor`.

Lines 29–33 set up several offsets that will be used to launch kernels and exchange data so that the computation and communication can be overlapped. These offsets define the regions of grid points that will need to be calculated at each stage of Fig. 18.13. They are also visualized in Fig. 18.15.

Note that the total number of slices in each device memory is four slices of left halo points (dashed white), plus four slices of left boundary points, plus `dimx*dimy*(dimz-8)` internal points, plus four slices of boundary points, and four slices of right halo points (dashed white). Variable `left_stage1_offset` defines the starting point of the slices that are needed in order to calculate the left boundary slices. This includes 12 slices of data: 4 slices of left-neighbor halo points, 4 slices of boundary points, and 4 slices of internal points. These slices are the leftmost in the partition so the offset value is set to 0 by Line 31. Variable `right_stage2_offset` defines the starting point of the slices that are needed for calculating the right boundary slices. This also includes 12 slices: 4 slices of internal points, 4 slices of right boundary points, and 4 slices of right halo cells. The beginning point of these 12 slices can be derived by subtracting the total number of slices `dimz+8` by 12. Therefore, the starting offset for these 12 slices is `dimx*dimy*(dimz-4)`.

Line 34 is an MPI barrier synchronization, which is similar to the CUDA `__syncthreads()`. MPI barrier forces all MPI processes specified by the parameter to wait for each other. None of the processes can continue their execution beyond this point until everyone has reached this point. The reason why we want to barrier synchronization here is to ensure that all compute nodes have received their input data and are ready to perform the computation steps. Since they will be exchanging data with each other, we would like to make them all start at about the same time. This way, we will not be in a situation where a few tardy processes delay all other processes during the data exchange. `MPI_Barrier()` is a *collective communication*

function. We will discuss more details about collective communication API functions in the next section.

Line 35 starts a loop that performs the computation steps. For each iteration, each compute process will perform one cycle of the two-stage process in Fig. 18.13.

Line 36 calls a function that will generate the four slices of the left boundary points in Stage 1. We assume that there is a kernel that performs one computation step on a region of grip points. The `launch_kernel()` function takes several parameters. The first parameter is a pointer to the output data area for the kernel. The second parameter is a pointer to the input data area. In both cases, we add the `left_stage1_offset` to the input and output data in the device memory. The next three parameters specify the dimensions of the portion of the grid to be processed, which is 12 slices in this case. Note that we need to have four slices on each side in order to correctly perform four computation steps for all the points in the four left boundary slices. Line 37 does the same for the right boundary points in Stage 1. Note that these kernels will be launched within `stream0` and will be executed sequentially.

Line 38 launches a kernel to generate the `dimx*dimy*(dimz-8)` internal points in Stage 2. Note that this also requires four slices of input boundary values on each side so the total number of input slices is `dimx*dimy*dimz`. The kernel is launched in `stream1` and will be executed in parallel with those launched by Lines 36 and 37.

Fig. 18.16 shows Part 4 of the compute process code. Line 39 copies the four slices of left boundary points to the host memory in preparation for data exchange with the left neighbor process. Line 40 copies the four slices of the right boundary points to the host memory in preparation for data exchange with the right neighbor process. Both are asynchronous copies in `Stream 0` and will wait for the two kernels in `Stream 0` to complete before they copy data. Line 41 is a synchronization that forces the process to wait for all operations in `Stream 0` to complete before it can continue. This makes sure that the left and right boundary points are in the host memory before the process proceeds with data exchange.

During the data exchange phase, we will have all MPI processes to send their boundary points to their left neighbors. That is, all processes will have their right neighbors sending data to them. It is therefore convenient to have an MPI function that sends data to a destination and receives data from a source. This reduces the number of MPI function calls. `MPI_Sendrecv()` function in Fig. 18.17 is such a

```
    /* 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);
```

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 */
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;
    }
```

FIGURE 18.18

Compute process code (Part 5).

function. It is essentially a combination of `MPI_Send()` and `MPI_Recv()` so we will not further elaborate on the meaning of the parameters.

Fig. 18.18 shows Part 5 of the compute process code. Line 42 sends four slices of left boundary points to the left neighbor and receives four slices of right halo points from the right neighbors. Line 43 sends four slices of right boundary points to the right neighbor

and receives four slices of left halo points from the left neighbor. In the case of Process 0, its `left_neighbor` has been set to `MPI_PROC_NULL` in Line 27 so the MPI runtime will not send out the message in Line 42 or receive the message in Line 43 for Process 0. Likewise, the MPI runtime will not receive the message in Line 42 or send out the message in Line 43 for Process `np-2`. Therefore, the conditional assignments in Lines 27 and 28 eliminate the need for special `if-the-else` statements in Lines 42 and 43.

After the MPI messages have been sent and received, Lines 44 and 45 transfer the newly received halo points to the `d_output` buffer of device memory. These copies are done in `stream0` so they will execute in parallel with the kernel launched in Line 38.

Line 46 is a synchronize operation for all device activities. This call forces the process to wait for all device activities, including kernels and data copies to complete. When the `cudaDeviceSynchronize()` function returns, all `d_output` data from the current computation step are in place: left halo data from the left neighbor process, boundary data from the kernel launched in Line 36, internal data form the kernel launched in Line 38, right boundary data from the kernel launched in Line 37, and right halo data from the right neighbor.

Lines 47 and 48 swap the `d_input` and `d_output` pointers. This changes the output of the `d_ouput` data of the current computation step into the `d_input` data of the next computation step. The execution then proceeds to the next computation step by going to the next iteration of the loop of Line 35. This will continue until all compute processes complete the number of computations specified by the parameter `nreps`.

Fig. 18.19 shows Part 6, the final part of the compute process code. Line 49 is a barrier synchronization that forces all processes to wait for each other to finish

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

50.  float *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);
     float *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 );
}
```

FIGURE 18.19

Compute process code (Part 6).

```
      /* 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);
}
```

FIGURE 18.20

Data server code (Part 3).

their computation steps. Lines 50–52 swap `d_output` with `d_input`. This is because Lines 47 and 48, in Fig. 18.18, swapped `d_output` with `d_input` in preparation for the next computation step. However, this is unnecessary for the last computation step. So, we use Lines 50–52 to undo the swap. Line 53 copies the final output to the host memory. Line 54 sends the output to the data server. Line 55 waits for all processes to complete. Lines 56–59 free all the resources before returning to the main program.

Fig. 18.20 shows Part 3, the final part of the data server code, which continues from Fig. 18.10. Line 20 is a barrier synchronization that waits for all compute nodes to complete their computation steps and send their outputs. This barrier corresponds to the barrier at Line 55 of the compute process (Fig. 18.19). Line 22 receives the output data from all the compute processes. Line 23 stores the output into an external storage. Lines 24 and 25 free resources before returning to the main program.

18.6 MESSAGE PASSING INTERFACE COLLECTIVE COMMUNICATION

The second type of MPI communication is collective communication, which involves a group of MPI processes. We have already seen an example of the second type of MPI communication API in the previous section: `MPI_Barrier`. The other commonly used group collective communication types are broadcast, reduction, gather, and scatter [Gropp 1999].

Barrier synchronization `MPI_Barrier()` is perhaps the most commonly used collective communication function. As we have seen the stencil example, barriers are used to ensure that all MPI processes are ready before they begin to interact with

each other. We will not elaborate on the other types of MPI collective communication functions, but encourage the reader to read up on the details of these functions. In general, collective communication functions are highly optimized by the MPI runtime developers and system vendors. Using them usually leads to better performance as well as readability and productivity than trying to achieve the same functionality with combinations of send and receive calls.

18.7 CUDA-AWARE MESSAGE PASSING INTERFACE

Modern MPI implementations are aware of the CUDA Programming model and are designed to minimize the communication latency between GPUs. Currently, direct interaction between CUDA and MPI is supported by MVAPICH2, IBM Platform MPI, and OpenMPI.

CUDA-aware MPI implementations are capable of sending messages from the GPU memory in one node to the GPU memory in a different node. This effectively removes the need of device-to-host data transfers before sending MPI messages, and host-to-device data transfers after receiving an MPI message. This has the potential of simplifying the host code and memory data layout. Following with our stencil example, if we use a CUDA-aware MPI implementation we no longer need host-pinned memory allocations and asynchronous memory copies.

The first simplification is that we no longer need host-pinned memory buffers to transfer the halo points to the host memory. This means that we can safely remove Lines 19–22 in Fig. 18.12. However, we still need to use CUDA streams and two separate GPU kernels to start communicating across nodes as soon as the halo elements have been computed.

The second simplification is that we no longer need to asynchronously copy the halo data from the device to the host memory. As a result, we can also remove Lines 39 and 40 in Fig. 18.16. Since the MPI calls now accept device memory addresses, we need to modify the calls to `MPI_SendRecv` to use them. Note that these memory addresses actually correspond to the device addresses of the asynchronous memory copies in the previous versions (Fig. 18.21).

Since the CUDA-aware MPI implementations will directly update the contents of the GPU memory, we also remove Lines 44 and 45 in Fig. 18.18.

```
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.

Besides removing the data transfers during the halo exchange using `MPI_SendRecv()`, it would be also possible to remove the initial and final memory copies receiving/sending the input/output directly from the GPU memory.

18.8 SUMMARY

We have covered basic patterns of joint CUDA/MPI Programming for HPC clusters with heterogeneous computing nodes. All processes in an MPI application run the same program. However, each process can follow different control flow and function call paths to specialize their roles, as illustrated by the data server and the compute processes in our example. We have also presented a common pattern where compute processes exchange data. We presented the use of CUDA streams and asynchronous data transfers to enable the overlap of computation and communication. We would like to point out that while MPI is a very different Programming system, all major MPI concepts that we covered in this chapter, SPMD, MPI ranks, and barriers have counterparts in the CUDA Programming model. This confirms our belief that by teaching parallel Programming with one model well, our students can quickly pick up other Programming models easily. We would like to encourage the reader to build on the foundation from this chapter and study more advanced MPI features and other important patterns.

18.9 EXERCISES

1. For vector addition, if there are 100,000 elements in each vector and we are using three compute processes, how many elements are we sending to the last compute process?
- **a.** 5
- **b.** 300
- **c.** 333
- **d.** 334

2. If the MPI call `MPI_Send(ptr_a, 1000, MPI_FLOAT, 2000, 4, MPI_COMM_WORLD)` resulted in a data transfer of 40,000 bytes, what is the size of each data element being sent?
- **a.** 1 byte
- **b.** 2 bytes
- **c.** 4 bytes
- **d.** 8 bytes

3. Which of the following statements is true?
- **a.** `MPI_Send()` is blocking by default.
- **b.** `MPI_Recv()` is blocking by default.
- **c.** MPI messages must be at least 128 bytes.
- **d.** MPI processes can access the same variable through shared memory.

4. Use the code base in Appendix A and examples in Chapters 3, 4, 5, and 6, Scalable parallel execution, Memory and data locality, Performance considerations, and Numerical considerations, to develop an OpenCL version of the matrix-matrix multiplication application.

5. Modify the example code to remove the calls to `cudaMemcpy()` on the compute node code by using GPU memory addresses on `MPI_Send` and `MPI_Recv`.

REFERENCE

Gropp, William, Lusk, Ewing, & Skjellum, Anthony (1999a). *Using MPI, 2nd edition: Portable parallel programming with the message passing interface*. Cambridge, MA: MIT Press Scientific And Engineering Computation Series. ISBN 978-0-262-57132-6.

This page intentionally left blank