GPGPU Programming

Donato D'Ambrosio

Department of Mathematics and Computer Science
Cubo 30B, University of Calabria, Rende 87036, Italy

Lecturer's email: donato.dambrosio [at] unical.it
Lecturer's homepage: http://www.mat.unical.it/~donato

Course's homepage: https://www.mat.unical.it/~donato/gpgpu.html
Course team page: https://bit.ly/3CU9xiR
Code to join the team: **3d98qzr**

Academic Year 2021/22

Table of contents

MPI/CUDA Programming

MPI/CUDA Programming

MPI/CUDA Programming

- We here consider how to run CUDA application on HPC computer clusters with one or more GPUs per node.
- We will consider MPI (Message Passing Interface) and its integration with CUDA.

Figure: Marconi 100 HPC cluster with Nvidia Volta V100 GPUs (CINECA).

Background

- MPI assumes a distributed memory model where processes exchange information 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.
- In a MPI application, data and work are partitioned among processes. Each node can contain one or more processes

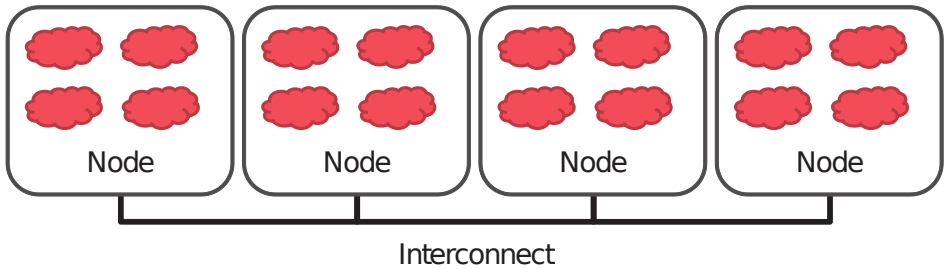


Figure: Example of processes (clouds) distributed among computing nodes interconnected to each other.

A Running Example

- We consider a heat transfer 3D stencil model based on a finite difference numerical scheme. In particular, we consider the Jacobi Iterative Method where, in each iteration (or time step), the new value of a grid point is calculated as a weighted sum of neighbors and its own value from the previous time step. A high order stencil computation (i.e., with indirect neighbors) is considered for numerical stability purpose.

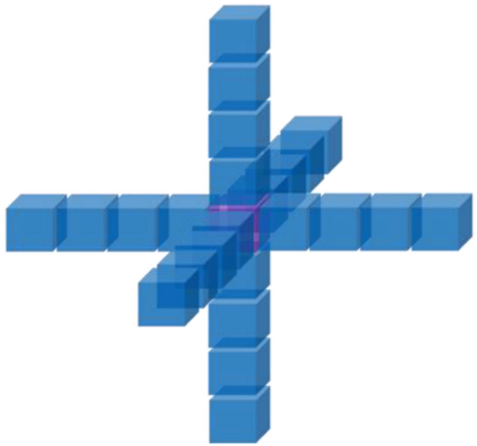
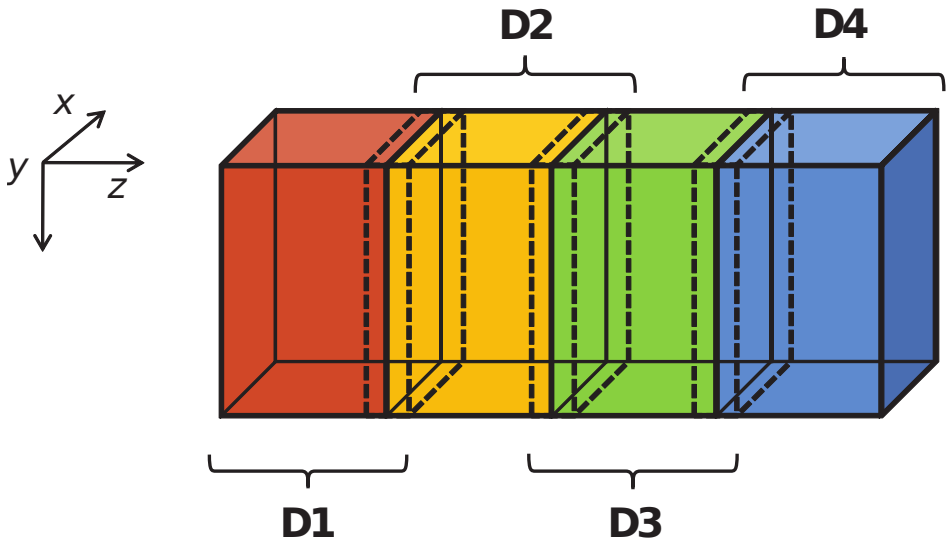


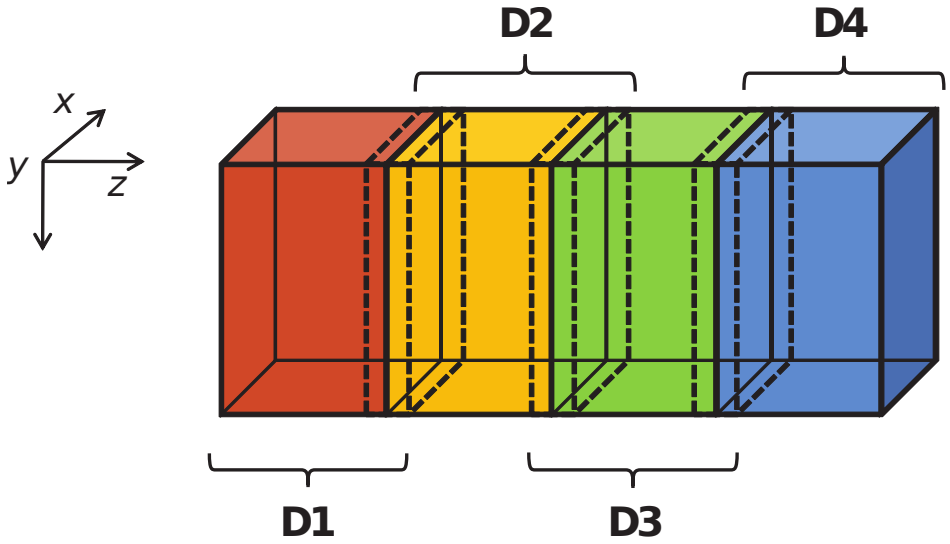
Figure: High-order 25-stencil computation neighborhood with for cells in each direction. Central cell is included.

Case of Study

- The system is modeled as a structured grid. The physical distance between adjacent elements in each dimension can be represented by a spacing variable. The Figure below shows the modeled domain, representing a rectangular ventilation duct.
- Slice-major, and then row-major order is adopted, i.e., the first dimension is *x* (the column), then y (the row), and eventually, z (the slice).



Case of Study



- The domain is subdivided in partitions (D1, D2, D3, and D4 in the example), each one assigned to an MPI compute process.
- This technique is quite common in Parallel Computing and is known as domain decomposition.
- We will ignore the CUDA kernel implementation to focus the CUDA/MPI integration.

MPI Basics

- Like CUDA, MPI is based on the SPMD parallel execution model. All MPI processes execute the same program.
- MPI provides a set of API functions that allow the processes to communicate with each other.

```
// Initialize MPI
int MPI_Init (int*argc, char***argv)
// Rank of the calling process in group of comm
int MPI_Comm_rank (MPI_Comm comm, int *rank)
// Number of processes in the group of comm
int MPI_Comm_size (MPI_Comm comm, int *size)
// Terminate MPI comminication connection with an error flag
int MPI_Comm_abort (MPI_Comm comm)
// Ending an MPI application, close all resources
int MPI_Finalize ( )
```

- `MPI_Comm` is an MPI type used to define communicator objects. A communicator is a group of MPI processes for the purpose of communication. `MPI_COMM_WORLD` is a predefined communicator including each running process.

MPI Basics

- In the example, we consider one compute process per GPU.
- We define a set of `np` processes where:
  - processes `0,1,...,np-2` compute the stencil algorithm;
  - process `np-1` performs I/O and data initialization (data server).
- Therefore, in the case the system has 32 GPUs, we run the program as:

```
mpirun -np 33 ./heat_multiGPU
```

- In the `main()` function, if `pid == np-1` (`pid` is the process id), the `data_server()` function is called, otherwise is the `compute_process()` function to be called.
- The `main()` function, together with the basic point-to-point communication functions used in the example considered, are shown in the next slides.

MPI Basics

```
#include <mpi.h> // <- This is the MPI header file

int main(int argc, char *argv[])
{
  int pid=-1, np=-1; dimx=dimy=480, dimz=400, nsteps=100;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  if(np < 3) {
    if (pid == 0) printf("At least 3 processes are required\n");
    MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
  }
  if(pid < np - 1)
    compute_process(dimx, dimy, dimz/(np - 1), nsteps);
  else
    data_server(dimx, dimy, dimz, nsteps);

  MPI_Finalize();
  return 0;
}
```

MPI Point-to-Point Communication

- Point-to-point communication happens between a source and a destination process.

```
int MPI_Send (
      void *buf,              //Starting address the source buffer
      int count,              //Number of elements in send buffer
      MPI_Datatype datatype,//Datatype of each send buffer element
      int dest,               //Rank of destination process
      int tag,                //Message tag
      MPI_Comm comm )         //MPI Communicator

int MPI_Recv (
      void *buf,              //Starting address the dest. buffer
      int count,              //Number of elements to be received
      MPI_Datatype datatype,//Datatype of receiving elements
      int source,             //Rank of source process
      int tag,                //Message tag
      MPI_Comm comm,          //MPI Communicator
      MPI_Status *status )    //Status object
```

MPI Point-to-Point Communication

- The `MPI_Sendrecv()` function is essentially a combination of `MPI_Send()` and `MPI_Recv()`.

```
int MPI_Sendrecv(
      void *sendbuf,          //Initial address of send buffer
      int sendcount,          //Number of elements in send buffer
      MPI_Datatype sendtype,//Type of elements in send buffer
      int dest,               //Rank of destination
      int sendtag,            //Send tag
      void *recvbuf,          //Initial address of dest. buffer
      int recvcount,          //Number of elements in receive buffer
      MPI_Datatype recvtype,//Type of elements in receive buffer
      int source,             //Rank of source
      int recvtag,            //Receive tag
      MPI_Comm comm,          //Communicator
      MPI_Status *status )   //Status of the receive operation
```

- We will see an example of application later in these slides.

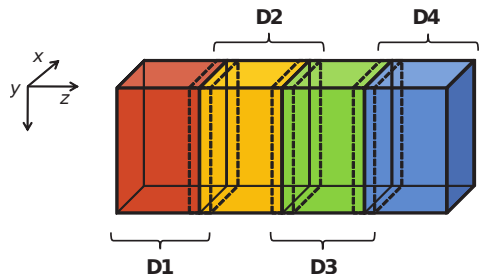
MPI Point-to-Point Communication

- `MPI_Send()` and `MPI_Recv` are blocking communication functions (the only once we consider in this lecture), meaning that the calling process waits until the function terminates.
- Pay attention to properly couple send/recv blocking calls between processes, otherwise a deadlock condition can occur!
- `MPI_Datatype` specifies the type of each data element being sent/received. It includes: `MPI_DOUBLE`, `MPI_FLOAT`, `MPI_INT`, and `MPI_CHAR`. The sizes of these types depend on the size of the corresponding C types in the host.
- 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.

MPI Point-to-Point Communication - Data Server 1/3

```
void data_server(int dimx, int dimy, int dimz, int nsteps)
{
  int np,
  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;
  float *input=0, *output=0;

  input = (float *)malloc(num_points * sizeof(float));
  output = (float *)malloc(num_points * sizeof(float));
  if(input == NULL || output == NULL) MPI_Abort( MPI_COMM_WORLD, 1 );

  //initialize data randomly
  random_data(input, dimx, dimy ,dimz , 1, 10);

  //define the number of points to be sent to the compute processes
  int edge_num_points = dimx * dimy * ((dimz / num_comp_nodes) + 4);
  int int_num_points  = dimx * dimy * ((dimz / num_comp_nodes) + 8);

  // continue...
```

MPI Point-to-Point Communication - Data Server 2/3



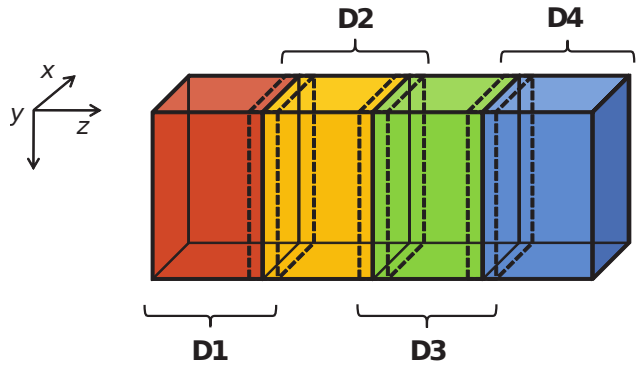

```
float *send_address = input;
MPI_Send(send_address,edge_num_points,MPI_FLOAT,first_node,0,
  MPI_COMM_WORLD);

send_address += dimx * dimy * ((dimz / num_comp_nodes) - 4);
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);
}

MPI_Send(send_address, edge_num_points, MPI_FLOAT, last_node, 0,
  MPI_COMM_WORLD);

//continue...
```

MPI Point-to-Point Communication - Data Server 3/3

```
  // Wait for nodes to compute
  // Actually, this call is not strictly necessary since we
  // have used blocking poin-to-point communication functions
  // to send the data to the computing processes
  MPI_Barrier(MPI_COMM_WORLD);

  // Collect output data from the compute processes
  // Even in this case, point-to-point blocking calls are considered
  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 );

   // Perform I/O and release memory
   store_output(output, dimx, dimy, dimz);
   free(input);
   free(output);
}
```

MPI Point-to-Point Communication - Compute Process

- The compute processes receive the input from the data server process, compute the stencil algorithm, and eventually send back the updated chunks to the data server process.



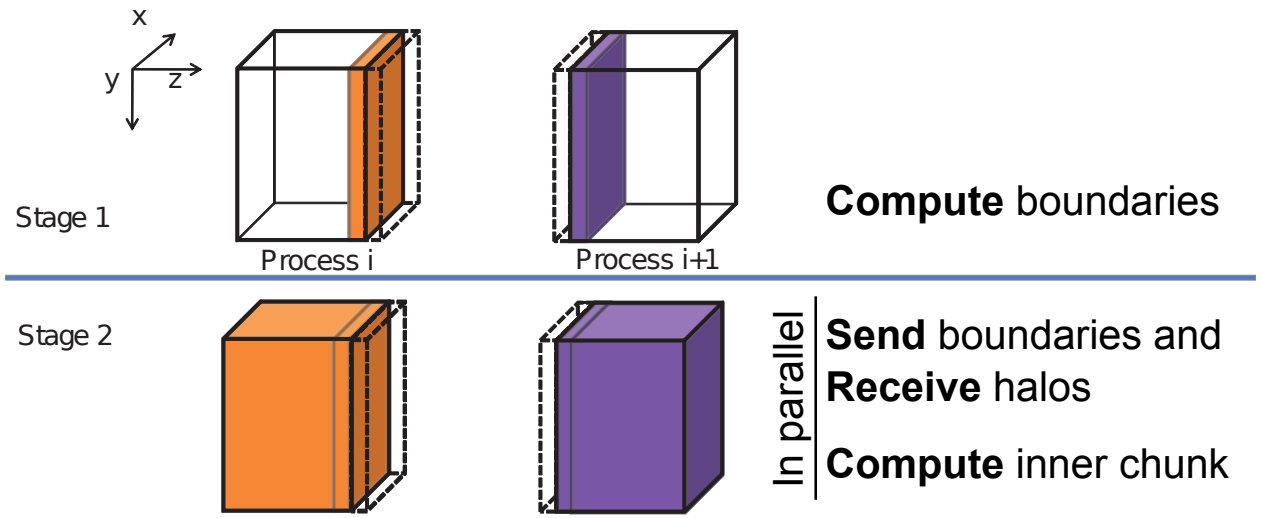
- First of all, the compute processes allocate both host an GPU memory needed to store the domain chunk, halos included (as a simplification, the edge processes allocate the same amount of memory of the other processes, although they need less halo data).
- Then, it receive the chunk from the data server and copy it to the GPU.

MPI Point-to-Point Communication - Compute Proc 1/6

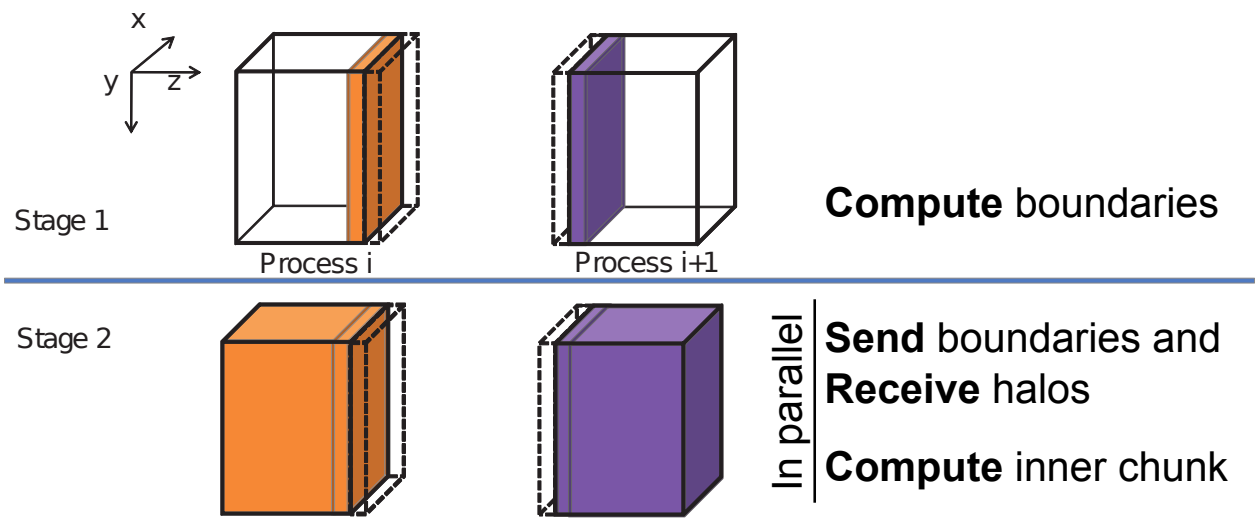
```
void compute_node_stencil(int dimx, int dimy, int dimz, int nsteps )
{
  int np, pid;
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Status status;
  int server_process = np - 1;

  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);

  float *h_input  = (float *)malloc(num_bytes);
  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);
  cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);

  //continue...
```

MPI Point-to-Point Communication - Compute Process

- The compute process then allocates memory for the output data. Note that, `d_input` and `d_output` will switch roles in each iteration in order to avoid copy operations.
- In order to hide the communication overhead, we overlap computation and communication (i.e., halos exchange), by using both the communication network and computation hardware in parallel. The computation tasks is subdivided into two stages

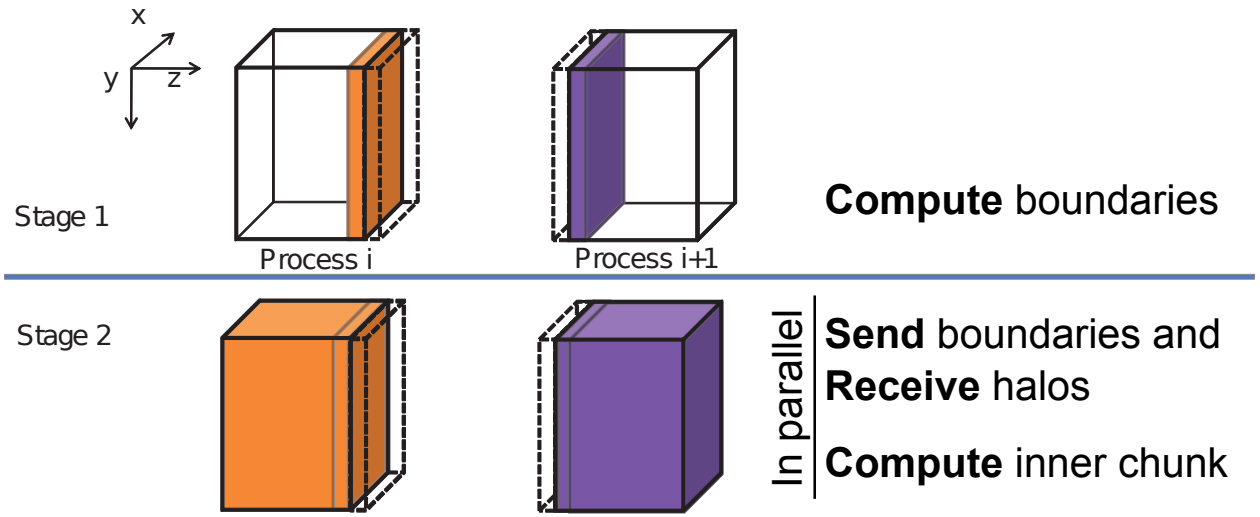


MPI Point-to-Point Communication - Compute Process



- Stage 1. All the compute processes update their boundary cells (both on the left and on the right).
- Stage 2, Parallel Activity 1.
  - Copy updated boundaries to the host.
  - Send boundary to adjacent processes/receive boundaries from adjacent processes and store them into the corresponding halos.
  - Compute the inner part of the domain chunk.

MPI Point-to-Point Communication - Compute Process



- If the communication takes a shorter amount of time than the calculation, the communication delay is hided.
- 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.

MPI Point-to-Point Communication - Compute Process

- Wath we have to do is to replace the blocking CUDA copy calls with non-blocking alternatives that permit the host application to continue its execution without the need to wait for the copy operation to complete.
- To this purpose, we use the `cudaMemcpyAsync()` function.
- In order to use the asynchronous copy function, the host memory buffer must be allocated as a pinned memory buffer by means of the `cudaHostAlloc()` function[1].

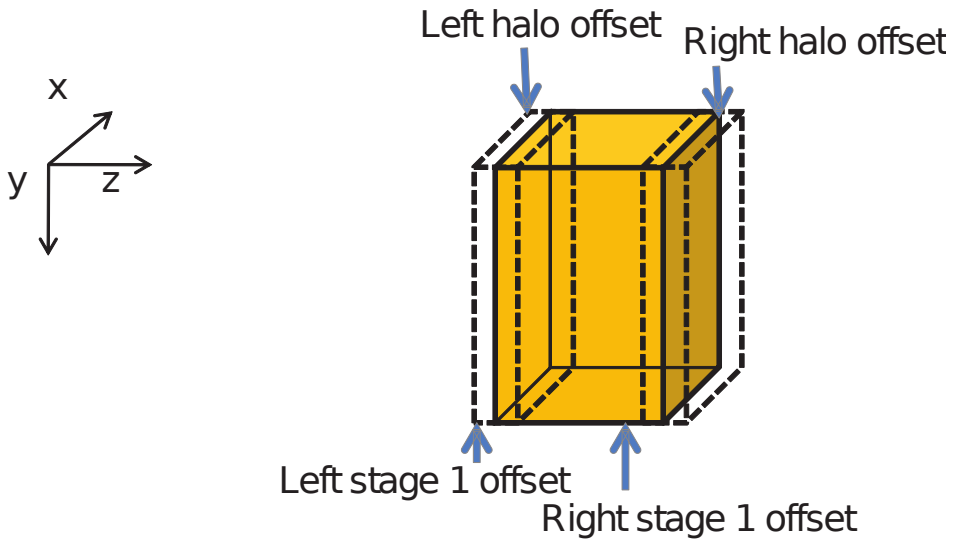
[1] Using a pinned buffer prevent the operating system to page out the buffer from the physical memory and reassign that area to another program. Without pinned memory, the asyncronous CUDA copy function could write over a memory area that the operating sytem could have reassigned to another application, resulting in a data corruption operation.

MPI Point-to-Point Communication - Compute Proc 2/6

```
float *h_output = NULL, *d_output = NULL, *d_vsq = NULL;
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 pinned 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);

//continue...
```

MPI Point-to-Point Communication - Compute Process

- The second advanced CUDA feature we need is streams, which supports managed concurrent execution of CUDA API functions.
- A stream is an ordered sequence of operations (queue). Operations from different streams will be executed in parallel.
- When the host calls `cudaMemcpyAsync()` or launches a kernel, it can specify a steam as one of its parameters.
- In the following we will refer the offsets below



MPI Point-to-Point Communication - Compute Proc 3/6

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

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

cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

MPI_Barrier( MPI_COMM_WORLD );
for(int i=0; i < nsteps; i++) {
  // Compute boundaries enqueuing kernels in stream0 (serial exec.)
  launch_kernel(d_output + left_stage1_offset,
                d_input + left_stage1_offset,
                dimx, dimy, 12, stream0); // 12 slices required...
  launch_kernel(d_output + right_stage1_offset,
                d_input + right_stage1_offset,
                dimx, dimy, 12, stream0); // ... to update the halos

  // continue...
```

MPI Point-to-Point Communication - Compute Proc 4/6

```
// Compute the remaining internal points
// Since the kernel is enqueued in the stream1 queue,
// internal points are computed in parallel with the
// boundary ones, which were enqueued in the stream0 queue
launch_kernel(d_output + stage2_offset,
              d_input + stage2_offset,
              dimx, dimy, dimz, stream1);

// Copy the data needed by other processes to the host
// This operation does not need to wait for the kernel to complete
// since kernels are non-blocking calls to the host
// Moreover, the following copy calls are asynchronous
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);

// like __syncthreads() but for stream0
cudaStreamSynchronize(stream0);

// continue...
```

MPI Point-to-Point Communication - Compute Process

- A possible slow-down in the above code could be due to the `cudaMemcpyAsync()` statements:

```
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);
```

- In fact, using the same stream (stream0), the operations are serialized GPU side (this also holds for the kernels executions).
- A possible improvement (if needed) could consist in using two streams for the halos computation and tranfer (one stream per halo).
- Using two different threads CPU side to parallelize the write operations in the host application, could further improve the performance.

MPI Point-to-Point Communication - Compute Proc 5/6

```
    // Since we are sure that stream0 has complete its execution
    // we can exchange the boundaries.
    // 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(); // Blocks until the device has completed
                             // all preceding requested tasks
    swap(d_input, d_output); // swaps buffers to avoid copy
}
// continue...
```

MPI Point-to-Point Communication - Compute Proc 6/6

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

  swap(d_input, d_output); // One more swap

  // 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_FLOAT, 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 );
}
```

Single Node Multi GPU

- The heat transfer example only work in the case of a single GPU per computing node.
- In the case of a single workstation with more GPUs (like JPDM2 for instance), or even a cluster with more GPUs per node, some modifications must be accounted in order to assign a different GPU to each MPI preocess running on the workstation/node.
- CUDA applications runs on the current GPU, which is the GPU attached to the first PCIx slot (usually 0) by default.
- A 0-based ID is assigned to each GPU attached to the host.
- The `cudaGetDeviceCount()` function can be used to query the number of GPUs attached to the host.
- The `cudaSetDevice()` function can be used to change the current GPU. Passing the MPI pid to the function will assign the GPU to the porcess.

Single Node Multi GPU: How To Complile on JPDM2

- In order to compile a CUDA/MPI application on JPDM2 you need to specify the MPI include directory and link the MPI library. To this purpose, you can issue something like the following command:

```
nvcc heat_multiGPU.cu -L/usr/lib/openmpi/ -lmpi -o heat_multiGPU
```

- The application can be therefore executed by defining a shell script for srun lite the following:

```
#Bash script runMPI.sh
#!/bin/bash
srun mpirun -np 33 ./heat_multiGPU
```

- And eventually submit the script to slurm by issuing the following command:

```
sbatch runMPI.sh
```

Slurm will redirect the output to a file like to `slurm-816.out`, where 816 can be any other number.