Parallel Technologies

Collective Communication Operations with NCCL on Multi-GPU Systems

Murilo Boratto

Contents

1 Abstract **1**

2 Collective Communication Operations **1**
2.1 Broadcast . . . 2
2.2 Reduce . . . 6
2.3 Gather . . . 7
2.4 ReduceScatter . . . 10
2.5 Peer-to-Peer Communications (P2P) . . . 11

3 Proposed Exercises **13**
3.1 Calculation of PI Number using the Riemann Integral . . . 13
3.2 AllReduce on Multi-GPU Systems . . . 14
3.3 Matrix-Vector Multiply on Multi-GPU Systems . . . 14

1 Abstract

It is necessary to carry out communication operations involving multiple computational resources in most parallel applications. These communication operations can be implemented through point-to-point operations. However, this approach is not very efficient for the programmer. Parallel and distributed solutions based on collective operations have long been chosen for these applications. The MPI [4] pattern has a set of very efficient routines that perform collective operations, taking better advantage of the computing power of the available computational resources. Likewise, with the advent of new computational resources, similar routines appear for multi-GPU systems, for example, *NCCL* (NVIDIA Collective Communications Library) [2]. This section will cover the handling of these routines for multi-GPU environments, constantly comparing them with the MPI standard, showing the differences and similarities between the two computational execution environments.

2 Collective Communication Operations

One of the main characteristics of using these types of operations is that communications can have different *symmetries* and *asynchronous*, considering factors such as emission and reception. For collective operations, *symmetry* is defined as the characteristic that all involved resources have to perform the same functions with very similar parameters. Asynchronous is defined as the inherent characteristic that all the involved resources have of not waiting for the others to finish to continue the execution. Considering that collective operations are communication patterns that affect all computational resources of an execution group, it was possible to compare aspects of collective operations using multiprocessor and multi-GPU systems, which are described below.

2.1 Broadcast

`Broadcast` is a collective operation where a computational resource sends the same information to all other elements of an execution group. In *NCCL* the responsible function that performs this operation is `ncclBcast`, being practically equivalent to the `MPI_Bcast` function of the MPI.

```
ncclBcast(void* buff,                             MPI_Bcast(void* buff,
          size_t count,                                     int count,
          ncclDataType_t datatype,                          MPI_DataType datatype,
          int root,                                         int root,
          ncclComm_t comm,                                  MPI_comm comm,
          cudaStream_t stream                               );
          );
```

The table above shows the comparative structure scheme of the functions `ncclBcast` and `MPI_Bcast`. The two functions are practically identical. All resources must invoke both in the `comm` communicator group. Functions send the information stored in `buff` of the `root` resource to everyone else belonging to the execution group. The parameters `count` and `datatype` have, respectively, the functions of specifying the amount of memory that the resource `root` should send to others and the space they should reserve to store the received message. The only difference between the two approaches is found in the last parameter called `stream`, which represents the sending format between the GPUs, so the operator structures are maintained through variable-sized memory spaces in buffer shipping.

You can compare the collective operation of `Broadcast` using MPI with the `MPI_Bcast` function. This function allows disseminating information to all processes of an MPI application, starting from the identifier process 0. As the information to be transmitted is stored in non-consecutive memory locations, it is necessary to invoke the number of processes initialized in the function. The function has the same parameters (`count`, `datatype`, `root` and `comm`) as before. To compile MPI programs, we must include the appropriate compilation option, such as:

```
$ mpicxx mpiCode.c -o object_mpi
```

To execute an MPI program with multiple computational resources (example with 4 processes):

```
$ mpirun -np 4 ./object_mpi
```

Next, a code with the same functionality is demonstrated using the `ncclBcast` function. This function allows you to broadcast information to multiple GPUs that are on the same execution group, which follows the scheme in Figure 1.

Algorithm 1 code that implements a `Broadcast` operation between multiprocessors using MPI. The code copies a vector of 8 positions to all computational resources of the group `comm`, starting from the identifier 0, multiplying each element by 2 at its end.

```
...

int main(int argc, char **argv){

  int i, rank, data_size = 8;
  int *data  = (int*) malloc(data_size * sizeof(int));

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  if(rank == 0){
      for(int i = 0; i < data_size; i++)
         data[i] = 1;
  }

  MPI_Bcast(data, data_size, MPI_INT, 0, MPI_COMM_WORLD);

  for(int i = 0; i < data_size; i++)
    data[i] *= 2;

  MPI_Finalize();

...
```

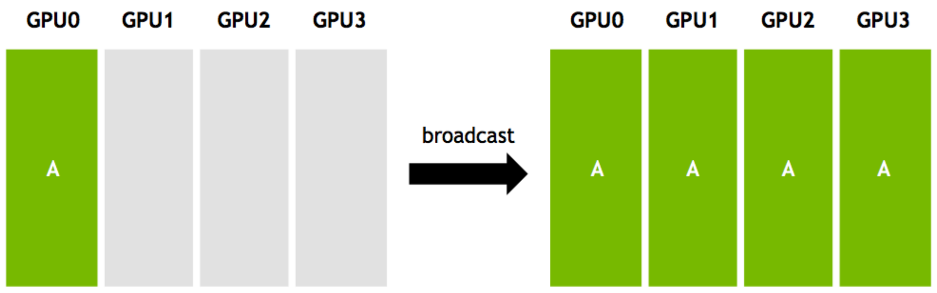

Figure 1: Operating scheme of the `Broadcast` operation in the `comm` group, with 4 resources. The resource `root` is the one with the identifier 0.

To compile *NCCL* programs, we must include the proper compilation option, such as:

```
$ nvcc ncclCode.cu -o object_nccl -lnccl
```

To execute a program with *NCCL* with multiple GPUs, we can use the following example:

```
$ ./object_nccl
```

Algorithm 2 The part of code that implements a `Broadcast` operation between multiple GPUs using *NCCL*. The function of this code is similar to the previous one.

```
...


__global__ void kernel(int *a)
{
  int index = threadIdx.x;
  a[index] *= 2;
}

int main(int argc, char **argv) {
  int data_size   = 8, nGPUs = 0;
  int *data       = (int*) malloc(data_size * sizeof(int));
  int **d_data    = (int**)malloc(nGPUs     * sizeof(int*));
  cudaGetDeviceCount(&nGPUs);
  int *DeviceList = (int *) malloc ( nGPUs * sizeof(int));

  for(int i = 0; i < nGPUs; i++)
      DeviceList[i] = i;

  ncclComm_t* comms = (ncclComm_t*)  malloc(sizeof(ncclComm_t)  * nGPUs);
  cudaStream_t* s   = (cudaStream_t*)malloc(sizeof(cudaStream_t)* nGPUs);
  ncclCommInitAll(comms, nGPUs, DeviceList);

  for(int i = 0; i < data_size; i++)
      data[i] = 1;

  for(int g = 0; g < nGPUs; g++) {
      cudaSetDevice(DeviceList[g]);
      cudaStreamCreate(&s[g]);
      cudaMalloc(&d_data[g], data_size * sizeof(int));

      if(g == 0)
         cudaMemcpy(d_data[g], data, data_size * sizeof(int), cudaMemcpyHostToDevice);
  }

  ncclGroupStart();
          for(int g = 0; g < nGPUs; g++) {
              cudaSetDevice(DeviceList[g]);
               ncclBcast(d_data[g], data_size, ncclInt, 0, comms[g], s[g]);
          }
  ncclGroupEnd();

  for(int g = 0; g < nGPUs; g++) {
      cudaSetDevice(DeviceList[g]);
          kernel <<< 1 , data_size >>> (d_data[g]);
      cudaThreadSynchronize();
  }

...
```

2.2 Reduce

`Reduce` operation is an operation where each computational resource involved contributes an operand to perform the global calculation of an associative or commutative operation (i.e., maximum, minimum, sum, product, etc). In *NCCL*, the responsible function that performs this operation is `ncclReduce`, being equivalent to the `MPI_Reduce` function of the MPI.

```
ncclReduce(const void* sendbuff,          MPI_Reduce(void* sendbuff,
           void* recvbuff,                           void* recvbuff,
           size_t count,                             int count,
           ncclDataType_t datatype,                  MPI_Datatype datatype,
           ncclRedOp_t op,                           MPI_Op op,
           int root,                                 int root,
           ncclComm_t comm,                          MPI_Comm comm
           cudaStream_t stream                      );
          );
```

The table above shows the comparative scheme of the functions `ncclReduce` and `MPI_Reduce`. As before, the two functions are identical. In the `Reduce` operation, the operator is applied to the data located in the send buffer (`sendbuff`) of each computational resource. The result of this function is passed back to all resources in the receive buffer (`recbuff`). The `count` and `datatype` parameters again specify how much memory the `root` resource should send to others and how much space they should reserve to store the incoming message. And again, the difference is in the last parameter called `stream`, which represents the format of the sending buffer between the GPUs.

To show an example of the use of the `Reduce` functions, we will use the example of the *scalar product* of vectors. The code will make the scalar product [1] of two vectors x and y distributed as shown below. First, the partial result $(x * y)$ is calculated, and then the `Reduce` operation is performed. After the operation, the final result will be stored in the source resource, called resource 0. This reduce operation is performed in *NCCL* by the `ncclReduce` function on multi-GPU systems (Figure 2).

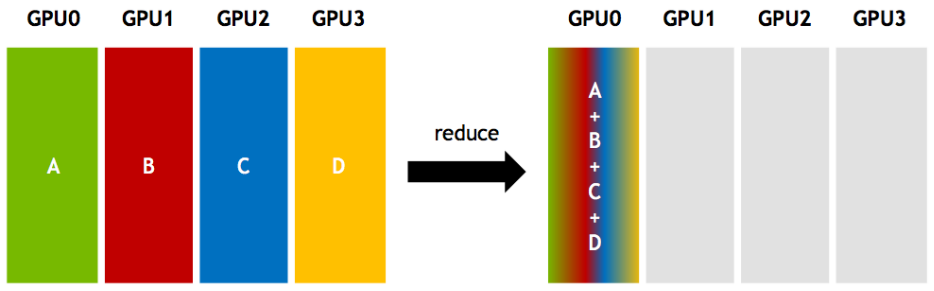

Figure 2: Operating scheme of the `Reduce` operation in the `comm` group, with 4 computational resources.

The part of code in which the `MPI_Reduce` function is used. This function allows the reduction of vectors product in multiprocessor systems using MPI.

Below is a code example using the `ncclReduce` function. This function allows the operation of reducing the product of vectors under multi-GPU using *NCCL*.

Algorithm 3 The part of code that implements a `Reduce` operation between multiprocessors using MPI. The code copies a vector of 4 positions to all computational resources of the `comm` group, and from the resource 0, the dot product of the vectors x and y is calculated, distributed equally between the p parts.

```
...

int main(int argc, char **argv) {
  int int data_size = 4, rank, size;
  double result = 0, result_f;
  double *x  = (double*) malloc(data_size * sizeof(double));
  double *y  = (double*) malloc(data_size * sizeof(double));

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  for(int i = 0; i < data_size; i++) {
      x[i] = 1; y[i] = 2;
      result = result + x[i] * y[i];
  }

  MPI_Reduce(&result, &result_f, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  MPI_Finalize();
...
```

Algorithm 4 The part of code that implements a reduction of operation using *NCCL*. The functionality of this code is similar to the previous one.

```
...
  ncclGroupStart();

     for(int g = 0; g < nGPUs; g++) {
       cudaSetDevice(DeviceList[g]);
       ncclReduce(x_d_data[g], Sx_d_data[g], data_size, ncclDouble, ncclSum, 0, comms[g], s[g]);
       ncclReduce(y_d_data[g], Sy_d_data[g], data_size, ncclDouble, ncclSum, 0, comms[g], s[g]);
     }

  ncclGroupEnd();

  for(int g = 0; g < nGPUs; g++) {
      cudaSetDevice(DeviceList[g]);
      Dev_dot <<< 1, data_size >>> (Sy_d_data[g], Sx_d_data[g], data_size);
      cudaThreadSynchronize();
  }

...
```

2.3 Gather

A `Gather` operation is a collective operation where a computational resource scans information from a set of resources. From one point of view, the `Gather` operation is the inverse of the `Scatter` [3] operation. The difference for this last one resides in a combination of data from a receiving resource, which is solely stored.

The syntax of the `Gather` function for GPUs corresponds to the `ncclAllGather` command which is related to the concept of `AllGather`, which is a routine invocation equivalent to performing p calls to the operation `Gather`, which each time acts as `root`, this is a different computational resource (Figure 3).

```
ncclAllGather(const void* sendbuff,
              void* recvbuff,
              size_t sendcount,
              ncclDataType_t datatype,
              ncclComm_t comm,
              cudaStream_t stream
             );
```

```
MPI_Allgather(void* sendbuff,
              int sendcount,
              MPI_Datatype sendtype,
              void* recvdbuff,
              int recvcount,
              MPI_Datatype recvtype,
              MPI_Comm comm
             );
```

All processes of the communicator `comm` send the content `sendbuff` to the process with identifier `root`. It concatenates all received data ordered by the sender identifier, starting from the position pointed by `recvbuff`. That is, process data with identifier 0 are stored before resources 1, and so on. The receive arguments (`recvdbuff`, `recvcount`, `recvtype`), only have meaning for the `root` resource in MPI. In *NCCL* this information is implicit in the function arguments.

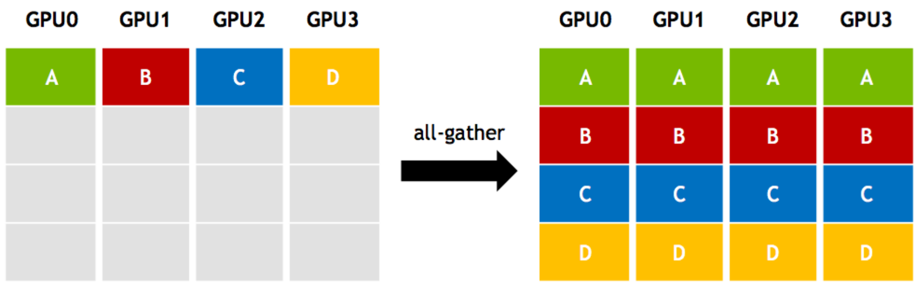

Figure 3: Scheme of the function `AllGather` in the group `comm`, with 4 resources.

The part of the code where the `MPI_Allgather` function is used is shown below. This function allows `AllGather` on multiprocessor systems using MPI.

Algorithm 5 The part of the code that implements a multiprocessor `AllGather` operation using MPI. The code copies a vector of 4 positions with the contents of the identifiers of each resource to everyone in the group.

```
...

int main( int argc, char **argv) {

    int isend, rank, size;
    int *irecv = (int *) calloc (4, sizeof(int));

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    switch(rank) {
      case 0 : isend = rank + 10;  break;
      case 1 : isend = rank + 19;  break;
      case 2 : isend = rank + 28;  break;
      case 3 : isend = rank + 37;  break;
    }

    MPI_Allgather(&isend, 1, MPI_INT, irecv, 1, MPI_INT, MPI_COMM_WORLD);

    MPI_Finalize();
...
```

Algorithm 6 The part of code that implements a `AllGather` operation on multiple GPUs using *NCCL*. The functionality of this code is similar to the previous one.

```
...
  ncclGroupStart();
        for(int g = 0; g < nGPUs; g++) {
           cudaSetDevice(g);
           ncclAllGather(sendbuff[g] + g, recvbuff[g], sendcount, ncclFloat, comms[g], s[g]);
        }
  ncclGroupEnd();
...
```

2.4 ReduceScatter

The `ReduceScatter` operation is a collective operation present in *NCCL* that merges two operations into one. The `Reduce` operation applies to the `Scatter` operation, which involves a reduction operation by distributing operated blocks among computational resources based on their identifying index.

```
ncclReduceScatter(const void* sendbuff,
                  void* recvbuff,
                  size_t recvcount,
                  ncclDataType_t datatype,
                  ncclRedOp_t op,
                  ncclComm_t comm,
                  cudaStream_t stream
                  );
```

The `root` process concatenates all received data sorted by the sender's range, starting from the position pointed to by `recbuff`. From the position pointed to by `recbbuff`, the `root` process concatenates all received data ordered by the receiver's interval. That is, the partial data of the lines of all computational resources are stored in a reduced way in the destination resources.

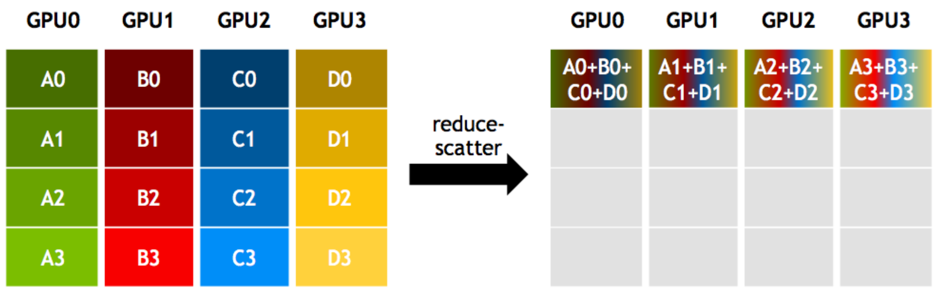

Figure 4: Scheme of the `ReduceScatter` operation in the `comm` group, with 4 resources.

The continuation of the part of the code in which the `ncclReduceScatter` function is used. This function allows you to apply `ReduceScatter` on multi-GPU systems using *NCCL*.

Algorithm 7 The part of code that implements a `ReduceScatter` operation on multiple GPUs using *NCCL*.

```
...

 ncclGroupStart();

  for(int g = 0; g < nGPUs; g++) {
   cudaSetDevice(g);
   ncclReduceScatter(sendbuff[g], recvbuff[g], recvcount, ncclFloat, ncclSum, comms[g], s[g]);
  }

  ncclGroupEnd();

  for(int g = 0; g < nGPUs; g++) {
      cudaSetDevice(g);
      Dev_print <<< 1, size >>> (recvbuff[g]);
      cudaThreadSynchronize();
  }

...
```

2.5 Peer-to-Peer Communications (P2P)

Peer-to-peer communication can be used to express any communication pattern between multiple GPUs. Any peer-to-peer communication needs two *NCCL* calls: one call to send the message (`ncclSend`) and the other to receive it (`ncclRecv`), and every message must have the same count and data typing. Multiple calls to `ncclSend` and `ncclRecv` can be combined with `ncclGroupStart` and `ncclGroupEnd` to form more complex communication patterns, i.e, the *NCCL* semantics allows all variants with different sizes, data types and buffers, by classification, for example: scattering communications, meetings or communication between neighbors in N-dimensional spaces. The syntax of the `ncclSend` and `ncclRecv` routines is shown below, compared to their respective anacronyms in MPI.

```
ncclSend(const void* sendbuff,              MPI_Send(void* sendbuff,
                size_t sendcount,                         int sendcount,
                ncclDataType_t datatype,                  MPI_Datatype sendtype,
                int peer,                                 int peer,
                ncclComm_t comm,                          int tag,
                cudaStream_t stream                       MPI_Comm comm,
               );                                         MPI_Status status
                                                          );

ncclRecv(const void* recvbuff,              MPI_Recv(void* recvbuff,
                size_t recvcount,                         int recvcount,
                ncclDataType_t datatype,                  MPI_Datatype recvtype,
                int peer,                                 int peer,
                ncclComm_t comm,                          int tag,
                cudaStream_t stream                       MPI_Comm comm,
               );                                         MPI_Status status
                                                          );
```

Peer-to-peer communications within a split will be asymmetric and blocking until the group call is completed. Still, calls within a division can be seen as progressing independently, so they should never block each other. Analogous to MPI, a point-to-point operation can be expressed as follows:

Algorithm 8 The part of code that implements a peer-to-peer communication operation using `ncclSend` and `ncclRecv` on multiple GPUs using *NCCL*.

```
...

 ncclGroupStart();

 for(g = 0; g < nGPUs; g++) {
   ncclSend(sendbuff[0], size, ncclInt, g, comms[g], s[g]);
   ncclRecv(recvbuff[g], size, ncclInt, g, comms[g], s[g]);
 }

 ncclGroupEnd();

...
```

3 Proposed Exercises

3.1 Calculation of PI Number using the Riemann Integral

From the following MPI code, write a parallel program using *NCCL* that makes use of the collective communication functions `Broadcast` and `Reduce` to calculate the PI number through the Integration of the $1/(1+x^2)$, where the Riemann sum approximates the integral.

```
...

int main(int argc, char **argv) {

  int master = 0, size, myrank, npoints, npointslocal, i;
  double delta, add, addlocal, x;

  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

  if (myrank == master ){
    printf("Numbers of divide points:");
    scanf("%ld", &npoints);
  }

  MPI_Bcast( &npoints, 1, MPI_INT, master, MPI_COMM_WORLD);

  delta = 1.0/((double) npoints);
  npointslocal = npoints/size;

  printf(" =================== %ld %ld %ld \n", myrank, npoints, npointslocal);

  addlocal = 0;

  x = myrank * npointslocal * delta;

  for (i = 1; i <= npointslocal; ++i){
     addlocal = addlocal + 1.0/(1+x*x);
     x = x + delta;
  }

  MPI_Reduce( &addlocal, &add, 1, MPI_DOUBLE, MPI_SUM, master, MPI_COMM_WORLD );

  if (myrank == master){
     add = 4.0 * delta * add;
     printf("\nPi = %20.16lf\n", add);
   }

   MPI_Finalize();
...
```

3.2 AllReduce on Multi-GPU Systems

From the definition of the `ncclAllReduce` function, write a parallel program using *NCCL* that makes use of the collective communication function of `AllReduce`.

```
    ncclAllReduce(const void* sendbuff,
                  void* recvbuff,
                  size_t count,
                  ncclDataType_t datatype,
                  ncclRedOp_t op,
                  ncclComm_t comm,
                  cudaStream_t stream
                 );
```

3.3 Matrix-Vector Multiply on Multi-GPU Systems

Create a parallel program for multi-GPU systems that executes a matrix-vector product (`y=Ax`) using a data distribution where the matrix `A` and the result vector `y` are distributed in blocks by rows of size `b`, and the vector `x` is spread in its entirety to all computational resources. Once the partial products have been made, the computational resource `P_i` should only have the `y_i` block of `y`. The simplest way to perform this operation is through the `AllGather` operation of the block stored in all resources in the group. Below is the MPI solution for `n` elements of the vector and distributed among all the resources (`y_distr`) of the group.

```
void reune_vector(int n, double *y_distr, double *y) {

  int P;                            // number of resources 'P'

  MPI_Comm_size(MPI_COMM_WORLD, &P); // Getting the number of features 'P'

  send_count =  n / P; // 'send_count' generates the value as a function of 'n' and 'P'

  MPI_Allgather(y_distr, send_count, MPI_DOUBLE, Y, send_count, MPI_DOUBLE, MPI_COMM_WORLD);
}
```

References

[1] Dot Product: Available in: http://mathworld.wolfram.com/DotProduct.html

[2] NVIDIA Collective Communications Library: Available in: https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/index.html

[3] Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of Collective Communication Operations in MPICH. International Journal High Performance Compuing Applications **19**(1), 49–66 (2005)

[4] Vidal, A., Gómez, J.: Introducción a la programación en MPI. Universidad Politècnica de València, Servicio de Publicaciones (2000)