Parallel Technologies

Samples Codes using CUDA-AWARE-MPI on Multi-GPU

Murilo Boratto

Índice

1. Abstract

MPI (Message Passing Interface) is a standard API for communicating data via messages between distributed processes, commonly used in HPC to create applications that can scale on multi-node supercomputers. The MPI is fully compatible with CUDA, designed for parallel computing across one or more nodes. There are many reasons to combine parallel programming in MPI and CUDA to maximize resources. Common sense allows troubleshooting with a data size that is too large to fit in the memory of a single GPU or that would require an excessively long calculation time if only one node were used. Another reason is accelerating an existing MPI application with GPUs to multi-GPU over an existing single or multiple nodes. With CUDA-Aware-MPI [2], these goals can be achieved quickly and efficiently. This chapter will explain how MPI and CUDA compatibility works, how efficient it is, and how it can be used.

2. Benchmarks Ping Pong

In this guide, we will look at a simple *ping pong* [1] code that measures the bandwidth for data transfers between 2 MPI processes. We will look at the following versions:

1. A first version using CPU with MPI;
2. A second version with MPI + CUDA between two GPUs which processes data through CPU memory;
3. And the last one that uses CUDA-Aware-MPI which exchange data directly between GPUs using GPUdirect or by NVLINK.

2.1. MPI

We will start by looking at a CPU-only version of the code to understand the idea behind a simple data transfer program (*ping pong*). MPI processes pass data back and forth, and bandwidth is calculated by measuring the data transfers, as you know how much size is being transferred. Let's look at the `ping-pong-MPI.c` code to see how it is implemented. At the top of the main program, we start the MPI, determine the total number of processes, the rank identifiers, and make sure we only have 2 ranks in total to run the *ping-pong*:

```
int size, rank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Status status;
```

We then enter the main *loop* `for`, where each iteration performs data transfers and bandwidth calculations for different message size, ranging from 8 bytes to 1 GB:

```
for(int i = 0; i <= 27; i++)
   long int N = 1 << i;
```

Next, we initialize the `A` array, define some labels to match the MPI send/receive pairs.

```
double *A = (double*) calloc (N, sizeof(double));
```

Basically, each iteration of the *loop* does the following:

- ◇ If rank is 0, it first sends a message with data from the matrix `A` to rank 1, then expects to receive a message of rank 1.
- ◇ If rank is 1, first expect to receive a message from rank 0 and then send a message back to rank 0.

```
start_time = MPI_Wtime();
    for(int i = 1; i <= loop_count; i++){
        if(rank == 0){
            MPI_Send(A, N, MPI_DOUBLE, 1, tag1, MPI_COMM_WORLD);
            MPI_Recv(A, N, MPI_DOUBLE, 1, tag2, MPI_COMM_WORLD, &stat);
        }
        else if(rank == 1){
            MPI_Recv(A, N, MPI_DOUBLE, 0, tag1, MPI_COMM_WORLD, &stat);
            MPI_Send(A, N, MPI_DOUBLE, 0, tag2, MPI_COMM_WORLD);
        }
    }
stop_time = MPI_Wtime();
```

The previous two points describe an application data transfer *ping pong* between the MPI ranks while the execution is measured. Then, from the synchronization results and the known size of the data transfers, we calculate the bandwidth and print the results:

```
long int num_B = 8 * N;
long int B_in_GB = 1 << 30;
double num_GB = (double)num_B / (double)B_in_GB;
double avg_time_per_transfer = elapsed_time / (2.0*(double)loop_count);
```

Remember that in order to compile MPI programs, we must include the appropriate compilation option, such as:

```
$ mpicc ping-pong-MPI.c -o ping-pong-MPI
```

To execute the *ping pong* MPI program, we will absolutely use 2 processes:

```
$ mpirun -np 2 ./ping-pong-MPI
```

2.2. MPI + CUDA

Now that we are familiar with the basic *ping pong* code in MPI let's take a look at a version that includes GPUs with CUDA. In this example, we are still passing data back and forth between two MPI ratings, but this time the data is in GPU memory. More specifically, rank 0 has a memory buffer on GPU 0, and rank 1 has a memory buffer on GPU 1, and they will pass the data between the memories of the two GPUs. Here, to get data from memory from GPU 0 to GPU 1, we will first put the data into CPU memory (*host*). Next, we can see the differences from the previous version to the new version with MPI+CUDA.

```
for(int i = 1; i <= loop_count; i++){
    if(rank == 0){
              cudaMemcpy(A, d_A, N * sizeof(double), cudaMemcpyDeviceToHost);
              MPI_Send(A, N, MPI_DOUBLE, 1, tag1, MPI_COMM_WORLD);
              MPI_Recv(A, N, MPI_DOUBLE, 1, tag2, MPI_COMM_WORLD, &status);
              cudaMemcpy(d_A, A, N * sizeof(double), cudaMemcpyHostToDevice);
    } else if(rank == 1){
              MPI_Recv(A, N, MPI_DOUBLE, 0, tag1, MPI_COMM_WORLD, &status);
              cudaMemcpy(d_A, A, N * sizeof(double), cudaMemcpyHostToDevice);
              cudaMemcpy(A, d_A, N * sizeof(double), cudaMemcpyDeviceToHost);
              MPI_Send(A, N, MPI_DOUBLE, 0, tag2, MPI_COMM_WORLD);
      }
}
```

Similar to the CPU-only version, we initialize MPI and find the identifier of each MPI rank, but here we also assign each rank a different GPU (i.e., rank 0 is assigned to GPU 0 and rank 1 is mapped to GPU 1).

```
int size, rank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Status status;

cudaSetDevice(rank);
```

For this release, each iteration of the *loop* does the following:

1. We enter the main *loop* `for` which iterates over the different message sizes, assign and initialize the `A` array. However, we now have a call to `cudaMalloc` to reserve a memory buffer (`d_A`) on the GPUs and a call to `cudaMemcpy` to transfer the data initialized in the `A` array to the buffer `d_A`. We need the command `cudaMemcpy` to get the data to the GPU before we start our *ping pong*.

2. Data must first be transferred from GPU memory 0 to CPU memory. Then an MPI call is used to pass the data from ranks 0 to 1. Now that rank 1 has the data (in CPU memory), it can transfer it to GPU memory 1. Rank 0 must first transfer the data from a buffer in GPU 0 memory to one in CPU memory. Now that rank 1 contains the data in the CPU memory buffer, it can transfer it to GPU 1 memory.

As in the case where only the CPU is used, from the synchronization results and the known size of the data transfers, we calculate the bandwidth, print the results, and finally free up the memory of the computational resources. We ended the MPI and the program.

Similarly, to compile MPI + CUDA programs, we must include the appropriate compilation option, such as:

```
$ nvcc -I/usr/include/openmpi -L/usr/lib/openmpi -lmpi -Xcompiler  code.cu -o object
```

To execute *ping pong* MPI+CUDA, we will use 2 processes:

```
$ mpirun -np 2 ./object
```

2.3. CUDA-AWARE-MPI

Before looking at this code example, let's first describe CUDA-AWARE-MPI and GPUDirect. CUDA-AWARE-MPI is an MPI implementation that allows GPU buffers (e.g., GPU memory allocated with `cudaMalloc`) to be used directly in MPI calls. However, CUDA-AWARE-MPI alone does not specify whether data is stored in intermediate stages in CPU memory or is passed directly from GPU to GPU. It will depend on the computational structure of the execution environment.

The GPUDirect is an umbrella name used to refer to several specific technologies. In the context of MPI, the GPUDirect technologies cover all kinds of inter-rank communication: intra-node, inter-node, and RDMA inter-node communication.

Now let is take a look at the code below. It is essentially the same as the tested version of MPI+CUDA, but now there are no calls to `cudaMemcpy` during the *ping pong* steps. Instead, we use our GPU buffers (`d_A`) directly in MPI calls:

```
    for(int i = 1; i <= loop_count; i++){
        if(rank == 0){
            MPI_Send(d_A, N, MPI_DOUBLE, 1, tag1, MPI_COMM_WORLD);
            MPI_Recv(d_A, N, MPI_DOUBLE, 1, tag2, MPI_COMM_WORLD, &status);
        }
        else if(rank == 1){
            MPI_Recv(d_A, N, MPI_DOUBLE, 0, tag1, MPI_COMM_WORLD, &status);
            MPI_Send(d_A, N, MPI_DOUBLE, 0, tag2, MPI_COMM_WORLD);
        }
    }
```

As before, to compile the code with CUDA-AWARE-MPI we must include the proper compilation options, for example:

```
$ nvcc -I/usr/include/openmpi -L/usr/lib/openmpi -lmpi -Xcompiler code.cu -o object
```

And likewise to run the *ping pong* CUDA-AWARE-MPI program, we will absolutely use 2 processes:

```
$ mpirun -np 2 ./object
```

3. Proposed Exercises

3.1. Matrix Multiply

Program a matrix multiplication using the CUDA-AWARE-MPI where matrices are generated in process 0, and all processes work in parts of the multiplication. The final result will finally be compiled in process 0. Make the implementation with distribution by blocks of lines and compare the execution times varying the size of the problem.

Referencias

[1] GitHub: olcf-tutorials. Available in: https://github.com/olcf-tutorials/MPI_ping_pong

[2] NVIDIA: An Introduction to CUDA-Aware MPI. Available in: `https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/`