Tecnologías Paralelas

Códigos utilizando el CUDA-AWARE-MPI sobre Multi-GPU

Murilo Boratto

Índice

1. Resumen

MPI, la interfaz de paso de mensajes, es una API estándar para comunicar datos a través de mensajes entre procesos distribuidos que se usa comúnmente en HPC para crear aplicaciones que pueden escalar en supercomputadores de múltiples nodos. Como tal, MPI es totalmente compatible con CUDA, que está diseñado para la computación paralela en uno o más nodos. Hay muchas razones para querer combinar la programación paralela en MPI y CUDA utilizando las máximas prestaciones. Una razón común es permitir la resolución de problemas con un tamaño de datos demasiado grande como para caber en la memoria de una sola GPU, o que requerirían de un tiempo de cálculo excesivamente largo si solo se utilizase un nodo. Otra razón es la de acelerar una aplicación MPI existente con GPUs o permitir que una aplicación multi-GPU de un solo nodo existente se escale a varios nodos. Con CUDA-Aware-MPI [2], estos objetivos se pueden lograr de manera fácil y eficiente. En este capítulo, se explicará cómo funciona la compatibilidad entre MPI y CUDA, por qué es eficiente y cómo se puede utilizar.

2. Benchmarks Ping Pong

En este guión veremos un código simple de *ping pong* [1] que mide el ancho de banda para transferencias de datos entre 2 procesos MPI. Veremos las siguientes versiones:

1. Solo para CPU utilizando MPI;

2. Una versión MPI + CUDA entre dos GPUs que procesa datos a través de la memoria de la CPU;

3. Una versión CUDA-Aware-MPI que intercambia datos directamente entre GPUs utilizando GPUdirect.

2.1. MPI

Comenzaremos mirando una versión del código solo para CPU para comprender la idea que está detrás de un programa sencillo de transferencia de datos (*ping pong*). Básicamente, 2 procesos MPI pasan datos de un lado a otro y el ancho de banda se calcula midiendo las transferencias de datos y conociendo el tamaño que se transfiere. Veamos el código `ping-pong-MPI.c` para ver cómo se implementa. En la parte superior del programa principal, inicializamos MPI, determinamos el número total de procesos (ranks), el identificador de cada rank y nos aseguramos de que solo tengamos 2 ranks en total para ejecutar el programa de *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;
```

A continuación, entramos en el bucle `for` principal, donde cada iteración realiza transferencias de datos y cálculos de ancho de banda para un tamaño de mensaje diferente, que van de 8 Bytes a 1 GB:

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

Luego inicializamos la matriz `A`, configuramos algunas etiquetas para que coincidan con los pares de envío/recepción de MPI.

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

Básicamente, cada iteración del ciclo hace lo siguiente:

◇ Si tiene el rank 0, primero envía un mensaje con los datos de la matriz `A` al rank 1, y luego espera recibir un mensaje desde el rank 1.

◇ Si tiene el rank 1, primero esperar recibir un mensaje del rank 0, y luego enviar un mensaje de regreso al 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();
```

Los dos puntos anteriores describen una transferencia de datos de *ping pong* entre los ranks MPI, mientras se cronometra la ejecución. Luego, a partir de los resultados de sincronización y el tamaño conocido de las transferencias de datos, calculamos el ancho de banda e imprimimos los resultados:

```
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);
```

Hay que recordar que, para compilar programas MPI, debemos incluir la opción de compilación adecuada, como:

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

Para ejecutar el programa de *ping pong* MPI utilizaremos 2 procesos:

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

2.2. MPI + CUDA

Ahora que estamos familiarizados con un código básico de *ping pong* en MPI, veamos una versión que incluye GPU con CUDA. En este ejemplo, todavía pasamos datos de un lado a otro entre dos ranks MPI, pero esta vez los datos están en la memoria de la GPU. Más específicamente, el rank 0 tiene un buffer de memoria en la GPU 0 y el rank 1 tiene un buffer de memoria en la GPU 1, y pasarán datos de un lado a otro entre las memorias de las dos GPU. Aquí, para obtener datos de la memoria de la GPU 0 a la GPU 1, primero colocaremos los datos en la memoria de la CPU. Ahora, dando un vistazo al código para ver las diferencias con la versión solo para CPU con MPI para la nueva versión con 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 a la versión solo para CPU, inicializamos MPI y encontramos el identificador de cada rank MPI, pero aquí también asignamos el rank MPI a una GPU diferente (es decir, el rank 0 se asigna a la GPU 0 y el rank 1 es mapeado en la 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);
```

Para esta versión, cada iteración del ciclo hace lo siguiente:

1. Al entramos en el bucle `for` principal que itera sobre los diferentes tamaños de mensaje, asignamos e inicializamos la matriz `A`. Sin embargo, ahora tenemos una llamada a `cudaMalloc` para reservar un buffer de memoria (`d_A`) en las GPUs y una llamada a `cudaMemcpy` para transferir los datos inicializados en la matriz `A` al buffer `d_A`. Se necesita `cudaMemcpy` para llevar los datos a la GPU antes de iniciar nuestro *ping pong*.

2. Los datos, primero deben transferirse de la memoria de la GPU 0 a la memoria de la CPU. Luego, se usa una llamada MPI para pasar los datos del rank 0 al 1 (en la memoria de la CPU). Ahora que el rank 1 tiene los datos (en la memoria de la CPU), puede transferir a la memoria de la GPU 1. Esto es, el rank 0 debe transferir primero los datos de un buffer en la memoria de la GPU 0 a uno en la memoria de la CPU. Ahora que el rank 1 tiene los datos en el buffer de memoria de su CPU puede transferirlos a la memoria de la GPU 1.

Al igual que en el caso en el que solo se utiliza la CPU, a partir de los resultados de sincronización y el tamaño conocido de las transferencias de datos, calculamos el ancho de banda e imprimimos los resultados y, finalmente, liberamos la memoria tanto en la CPU como en la GPU. Finalizamos MPI y salimos del programa.

Para compilar programas MPI+CUDA, debemos incluir la opción de compilación adecuada, como por ejemplo:

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

Para ejecutar el *ping pong* MPI+CUDA utilizaremos 2 procesos:

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

2.3. CUDA-AWARE-MPI

Antes de mirar este ejemplo de código, describamos primero CUDA-AWARE-MPI y GPUDirect. CUDA-AWARE-MPI es una implementación de MPI que permite que los buffers de GPU (por ejemplo, memoria de GPU asignada con `cudaMalloc`) se utilicen directamente en llamadas MPI. Sin embargo, CUDA-AWARE-MPI por sí mismo no especifica si los datos se almacenan en etapas a través de la memoria de la CPU o se pasan directamente de la GPU a la GPU.

GPUDirect es una mezcla de solución hardware y software que puede mejorar CUDA-AWARE-MPI al permitir transferencias de datos directamente entre GPU en el mismo nodo (*peer-to-peer*) o directamente entre GPUs en diferentes nodos (soporte RDMA), sin la necesidad de almacenar datos a través de la memoria de la CPU.

Ahora echemos un vistazo al código. Es esencialmente lo mismo que la versión por etapas de CUDA, pero ahora no hay llamadas a `cudaMemcpy` durante los pasos de *ping pong*. En su lugar, usamos nuestros buffers de GPU (`d_A`) directamente en las llamadas MPI:

```
    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);
          }
    }
```

Para compilar programas con CUDA-AWARE-MPI debemos incluir la opción de compilación adecuada, como por ejemplo:

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

Para ejecutar el programa de *ping pong* de CUDA-AWARE-MPI utilizaremos 2 procesos:

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

3. Ejercicio Propuesto

3.1. Multiplicación de Matrices

Programar una multiplicación de matrices utilizando la API CUDA-AWARE-MPI donde las matrices se generan en el proceso 0, todos los procesos trabajan en la multiplicación y el resultado queda finalmente en el proceso 0. Hacer la implementación con una distribución por bloques de filas y al final comparar los tiempos de ejecución variando el tamaño del problema.

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/`