Tecnologías Paralelas

Operaciones de Comunicación Colectivas utilizando Sistemas Multi-GPU

Murilo Boratto

Sumário

1 Resumen

En la mayoría de aplicaciones paralelas es necesario realizar operaciones de comunicación donde se involucran múltiples recursos computacionales. Estas operaciones de comunicación podrían implementarse mediante operaciones punto a punto. Sin embargo, esta aproximación no es muy eficiente para el programador y para las prestaciones de las aplicaciones paralelas. MPI [4] dispone de un conjunto de rutinas bastante eficientes que realizan operaciones colectivas con el objetivo de aprovechar mejor los recursos computacionales concurrentes. A partir de esto surgen rutinas similares para sistemas multi-GPU, por ejemplo: *NCCL* (NVIDIA Collective Communications Library) [2]. Luego, en esta sección se abordará la manipulación de estas rutinas para ambientes multi-GPU, siempre comparando con el estándar MPI, mostrando las diferencias y similitudes entre los dos ambientes computacionales de ejecución.

2 Operaciones de Comunicación Colectivas

Una de las principales características del uso de estos tipos de operaciones es que las comunicaciones son *simétricas* y *asíncronas*. Se dice *simétrica* porque todos los recursos involucrados ejecutan las mismas funciones con parámetros muy similares. La comunicación es *asíncrona* porque todos los recursos involucrados no deben esperar a que todos los demás finalicen para continuar la ejecución. Las operaciones colectivas son patrones de comunicación que afectan a todos los recursos computacionales de un grupo de ejecución. A continuación se describen los aspectos de operaciones colectivas utilizando sistemas multipocesadores y multi-GPU.

2.1 Difusión

La *Difusión*, o simplemente *Broadcast*, es una operación colectiva donde un recurso computacional envía la misma información a todos los otros elementos de un grupo de ejecución. En *NCCL* la función responsable que realiza esta operación es `ncclBcast`, siendo prácticamente equivalente a la función `MPI_Bcast` de 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                            );
          );
```

La tabla anterior muestra el esquema comparativo de funcionamiento de las funciones `ncclBcast` y `MPI_Bcast`. Las dos funciones son prácticamente idénticas. Las dos deben ser invocadas por todos los recursos del grupo `comm` con el mismo parámetro `root`. Las funciones envían la información almacenada en `buff` para el recurso `root` a todos del grupo de ejecución. Los parámetros `count` y `datatype` tienen, respectivamente, las funciones de especificar la cantidad de memoria que debe enviar el recurso `root` a los demás, y el espacio que éstos deben reservar para almacenar el mensaje recibido. Lo único diferente es último parámetro llamado `stream`, lo cual representa el formato de envío entre las GPUs, donde se sostienen las estructuras de los operadores a través de espacios de memoria de tamaño variable del búfer de envío.

Se puede comparar la operación colectiva de *Difusión* utilizando MPI con la función `MPI_Bcast`. Esta función permite difundir una información a todos los procesos de una aplicación MPI, a partir del proceso de rango 0. Puesto que la información a difundir se halla almacenada en posiciones no consecutivas de memoria, es necesario invocar el número de procesos inicializados en la función de Difusión. Es importante destacar que todos los procesos invocan la función con los mismos parámetros (`count`, `datatype`, `root` y `comm`), pudiendo ser distinto en cada proceso el valor del puntero a la información (`data`). Hay que recordar que para compilar programas MPI, debemos incluir la opción de compilación adecuada, como:

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

Para ejecutar un programa con MPI con varios recursos computacionales, se puede utilizar el siguiente ejemplo con 4 procesos:

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

Algoritmo 1 Parte del código que implementa una operación de `Difusión` entre multiprocesadores utilizando MPI. El código copia un vector de 8 posiciones a todos los recursos computacionales del grupo, a partir del recurso 0, multiplicando cada elemento por 2 a su final.

```
...

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();

...
```

Seguidamente se muestra el código con la misma funcionalidad utilizando la función `ncclBcast`. Esta función permite difundir una información para múltiples GPUs que estuvieran sobre el mismo grupo de ejecución, que sigue el esquema de la Figura 1.

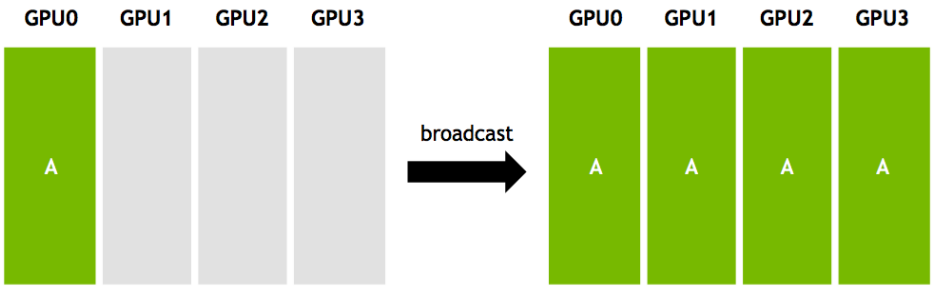

Figura 1: Esquema de funcionamiento de una `Difusión` en el grupo `comm`, con 4 recursos. El recurso `root` es el que tiene el rango 0.

Para compilar programas *NCCL*, debemos incluir la opción de compilación adecuada, como:

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

Para ejecutar un programa con *NCCL* con varias GPUs, puede usar el siguiente ejemplo:

```
$ ./object_nccl
```

Algoritmo 2 Parte del código que implementa una operación de `Difusión` entre múltiples GPUs utilizando *NCCL*. La funcionalidad de este código es similar al anterior.

```
...


__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 Reducción

Una operación de *Reducción* o *Reduce* es una operación donde cada recurso computacional involucrado contribuye con un operando para realizar el cálculo global de una operación asociativa y conmutativa (por ejemplo: máximo, mínimo, suma, producto, etc.). En *NCCL* la función responsable que realiza esta operación es `ncclReduce`, siendo equivalente a la función `MPI_Reduce` de MPI.

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

```
MPI_Reduce(void* sendbuff,
           void* recvbuff,
           int count,
           MPI_Datatype datatype,
           MPI_Op op,
           int root,
           MPI_Comm comm
          );
```

La tabla anterior muestra el esquema comparativo de funcionamiento de las funciones `ncclReduce` y `MPI_Reduce`. De la misma forma que antes, las dos funciones son idénticas. En la operación de *Reducción* el operador se aplica sobre los datos ubicados en el búfer de envío (`sendbuff`) de cada recurso computacional. El resultado de esta función se difunde de vuelta a todos los recursos en el búfer de recepción (`recvbuff`). Los parámetros `count` y `datatype` otra vez especifican la cantidad de memoria que debe enviar el recurso `root` a los demás, y el espacio que éstos deben reservar para almacenar el mensaje recibido. Y de nuevo la diferencia se encuentra en el último parámetro llamado `stream`, lo cual representa el formato del búfer de envío entre las GPUs.

Para mostrar un ejemplo de uso de las funciones de Reducción se utilizará el ejemplo del *producto escalar* de vectores. El código realizará el producto escalar [1] de dos vectores x e y distribuidos tal como se muestra a continuación. En primer lugar, se calcula el resultado parcial $(x * y)$, y después se realiza la operación de reducción. Después de la operación de reducción, el resultado final se almacenará en el recurso de origen, llamado recurso 0. Esta operación de reducción se realiza en *NCCL* mediante la función `ncclReduce` en sistemas multi-GPU (Figura 2).

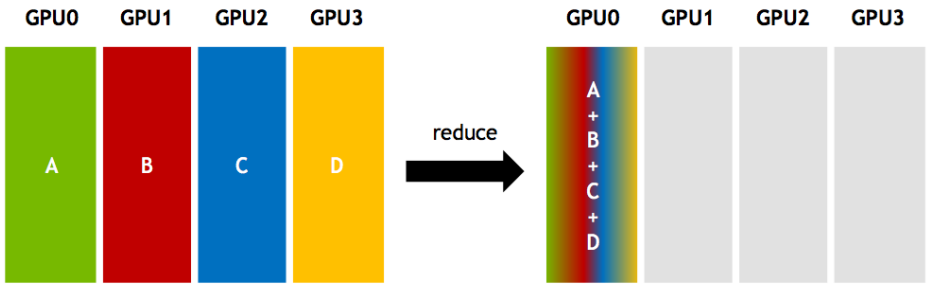

Figura 2: Esquema de funcionamiento de una `Reducción` en el grupo `comm`, con 4 recursos.

Fragmento de código en el que se utiliza la función `MPI_Reduce`. Esta función permite la reducción del producto vectores en sistemas multiprocesadores utilizando MPI.

A continuación se muestra un ejemplo de código en el que se utiliza la función `ncclReduce`. Esta función permite la reducción del producto vectores en multi-GPU utilizando *NCCL*.

Algoritmo 3 Parte del código que implementa una operación de `Reducción` entre multiprocesadores utilizando MPI. El código copia un vector de 4 posiciones a todos los recursos computacionales del grupo, a partir del recurso 0, calculando el producto escalar de los vectores x y y, distribuido por bloques de elementos equitativamente entre p partes.

```
...

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();
...
```

Algoritmo 4 Parte del código que implementa una operación de Reducción entre múltiples GPUs utilizando *NCCL*. Las funcionalidades de este código es similar al anterior.

```
...

  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 Recogida

Una operación de *Recogida* o *Gather*, es una operación colectiva donde un recurso computacional recoge información a partir de un conjunto de recursos. Desde este punto de vista, la operación de `Recogida` es la inversa de la operación de `Reparto` [3]. La diferencia con respecto de esta última reside en que no se realiza una combinación de los datos por parte del recurso receptor, únicamente se almacenan. La sintaxis de la rutina de *Recogida* es la función `ncclAllGather` que está relacionada con el concepto de Multi-Recogida, ya que una invocación a la rutina equivale a la realización de p llamadas a la operación de *Recogida*, en la que cada vez actúa como `root` un recurso computacional diferente (Figura 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
             );
```

Todos los procesos del comunicador `comm` envían el contenido `sendbuff` al proceso con rango `root`. El proceso `root` concatena todos los datos recibidos ordenados por el rango del emisor, a partir de posición apuntada por `recvbuff`. Es decir, los datos del proceso con rango 0 son almacenados antes que los del recurso 1, y así sucesivamente. Los argumentos de recepción (`recvdbuff`, `recvcount`, `recvtype`), solo tienen significado para el recurso `rrot` en MPI. En *NCCL* estas informaciones están implícitas en los argumentos de la función.

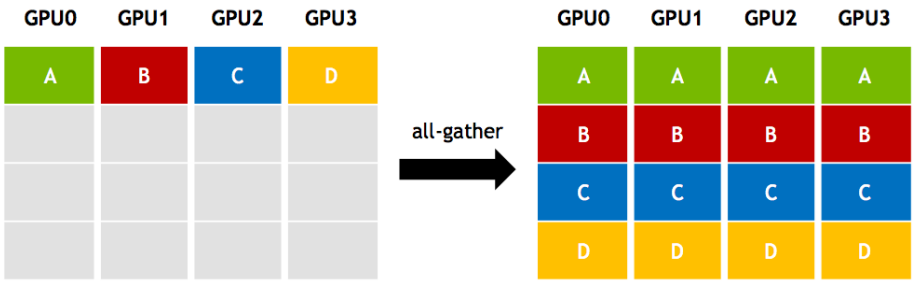

Figura 3: Esquema de funcionamiento de una `Multi-Recogida` en el grupo `comm`, con 4 recursos.

Se presenta a continuación la parte del código en el que se utiliza la función `MPI_Allgather`. Esta función permite la multi-recogida en sistemas multiprocesadores utilizando MPI.

Algoritmo 5 Parte del código que implementa una operación de `Multi-Recogida` entre multiprocesadores utilizando MPI. El código copia un vector de 4 posiciones con el contenido de los identificadores de cada recurso del grupo a todos del grupo.

```
...

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();
...
```

Algoritmo 6 Parte del código que implementa una operación de `Multi-Recogida` entre múltiples GPUs utilizando *NCCL*. La funcionalidad de este código es similar al anterior.

```
...
  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 Reducción y Reparto

La operación de *Reducción y Reparto*, o *ReduceScatter*, es una operación colectiva presente en *NCCL* que mezcla dos operaciones en una. La operación *Reducción y Reparto* realiza la misma operación que la operación de *Reducción*, excepto que el resultado se *Reparte* a bloques iguales entre el rango de recursos computacionales, y para cada recurso se obtiene una parte de los datos en función de su índice.

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

El proceso `root` concatena todos los datos recibidos ordenados por rango del emisor, a partir de la posición apuntada por `recvbuff`. A partir de la posición apuntada por `recvbuff`, el proceso `root` concatena todos los datos recibidos ordenados por el rango del receptor. Es decir, los datos parciales de las filas de todos recursos computacionales son almacenados de forma reducida en los recursos de destino (Figura 4).

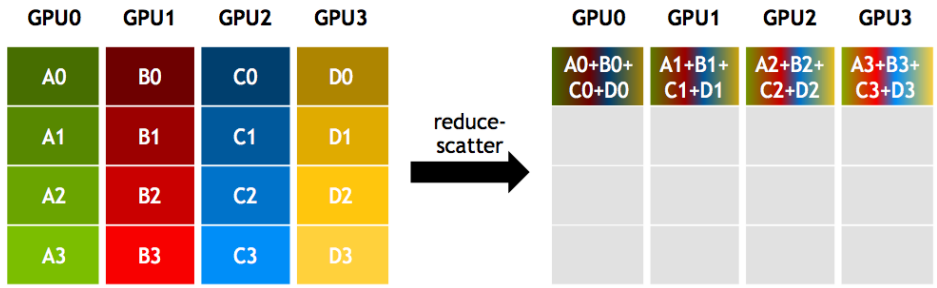

Figura 4: Esquema de funcionamiento de una operación del *Reducción y Reparto* en el grupo `comm`, con 4 recursos.

Se presenta a continuación la parte del código en el que se utiliza la función `ncclReduceScatter`. Esta función permite la *Reducción y Reparto* en sistemas multi-GPU utilizando *NCCL*.

Algoritmo 7 Parte de código que implementa una operación de `Reducción y Reparto` entre múltiples GPUs utilizando *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();
  }

...
```

3 Ejercicios Propuestos

3.1 Calculo del Número PI mediante la Integral de Riemann

A partir del código MPI a continuación escriba un programa paralelo utilizando *NCCL* que haga uso de funciones de comunicación colectivas de *Difusión* y *Reducción* para el cálculo del número PI mediante la Integración de la función $1/(1+x^2)$, donde la integral se aproxima mediante la Suma de Riemann.

```
...

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 Multi-Reducción en Sistemas GPUs

A partir de la definición de la función `ncclAllReduce` escriba una aplicación paralelo utilizando *NCCL* que haga uso de la función de comunicación colectiva de *Multi-Reducción*.

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

3.3 Reunión de los elementos de un vector

Construya una aplicación paralela para sistemas multi-GPU que realice un producto matriz por vector ($y = Ax$) utilizando una distribución de datos donde la matriz A y el vector resultado y se hallan distribuidos por bloques de filas de tamaño b, y el vector x se difunda en su totalidad a todos los recursos computacionales. Una vez realizados los productos parciales, el recurso computacional P_i debe poseer únicamente el bloque y_i de y. La forma más sencilla de realizar esta operación es mediante una *Multi-reunión* del bloque almacenado en todos los recursos del grupo. Mas abajo se muestra la solución MPI de la Multi-reunión de los n elementos del vector y distribuidos entre todos los recursos (y_{distr}), sobre todos los recursos del grupo.

```
void reune_vector(int n, double *y_distr, double *y) {
  int P;                              // Número de recursos

  MPI_Comm_size(MPI_COMM_WORLD, &P); // Obtner el número de recursos

  send_count =  n / P;                // 'send_count' toma valor en función de n y P

  // Se realiza la multi-reunión
  MPI_Allgather(y_distr, send_count, MPI_DOUBLE, Y, send_count, MPI_DOUBLE, MPI_COMM_WORLD);
}
```

Referências

[1] Dot Product: Disponible en: http://mathworld.wolfram.com/DotProduct.html

[2] NVIDIA Collective Communications Library: Disponible en: 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)