Tecnologías Paralelas

Operações de Comunicação Coletivas utilizando Sistemas Multi-GPU

Murilo Boratto

Sumário

1 Resumo

Na maioria das aplicações paralelas faz-se necessário realizar operações de comunicação envolvendo múltiplos recursos computacionais. Essas operações de comunicação podem ser implementadas por meio de operações ponto-a-ponto, entretanto, essa abordagem não é muito eficiente para o programador. Soluções paralelas e distribuídos baseadas em operações coletivas foram durante muito tempo a principal escolha para estas aplicações. O padrão MPI [4] possui um conjunto de rotinas bastante eficientes que realizam operações coletivas aproveitando melhor a capacidade de computação dos recursos computacionais disponíveis. Da mesma forma com o advento de novos recursos computacionais surgem rotinas similares para sistemas multi-GPU, por exemplo: *NCCL* (Biblioteca de Comunicações Coletivas da NVIDIA) [2]. Esta seção abordará a manipulação dessas rotinas para ambientes multi-GPU, sempre comparando com o padrão MPI, mostrando as diferenças e semelhanças entre os dois ambientes computacionais de execução.

2 Operações de Comunicação Coletivas

Uma das principais características do uso desses tipos de operações é que as comunicações podem ter *simétrias* e *assincronias* distintas, considerando fatores como emissão e recepção. Para as operações coletivas define-se a *simétria* como a característica que todos os recursos envolvidos tem de executar as mesmas funções com parâmetros muito semelhantes. Enquanto que, define-se a *assíncronia* como a característica inerente que todos os recursos envolvidos tem de não esperarem que os outros terminem para continuar a execução. Consirando que as operações coletivas são padrões de comunicação que afetam todos os recursos computacionais de um grupo de execução, conseguiu-se comparar os aspectos das operações coletivas usando sistemas multiprocessador e multi-GPU, os quais são descritos a seguir.

2.1 Broadcast

O *Broadcast*, ou simplesmente *Difusão*, é uma operação coletiva onde um recurso computacional envia as mesmas informações para todos os outros elementos de um grupo de execução. Em *NCCL* a função responsável que executa esta operação é `ncclBcast`, sendo praticamente equivalente à função `MPI_Bcast` do MPI.

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

```
MPI_Bcast(void* buff,
          int count,
          MPI_DataType datatype,
          int root,
          MPI_comm comm,
          );
```

O quadro acima mostra o esquema de estruturas comparativas das funções `ncclBcast` e `MPI_Bcast`. As duas funções são praticamente idênticas. Ambas devem ser invocados por todos os recursos do grupo de comunicadores `comm`. As funções enviam as informações armazenadas em `buff` do recurso `root` para todos os outros pertencentes ao grupo de execução. Os parâmetros `count` e `datatype` têm, respectivamente, as funções de especificar a quantidade de memória que o recurso `root` deve enviar para outros, e o espaço que eles devem reservar para armazenar a mensagem recebida. A única diferença entre as duas abordagens se encontra no último parâmetro chamado `stream`, o qual representa o formato de envio entre as GPUs, de forma que as estruturas do operador são mantidas através de espaços de memória de tamanho variável no *buffer* de envio.

Pode-se comparar a operação coletiva de *Broadcast* usando MPI com a função `MPI_Bcast`. Esta função permite disseminar informações para todos os processos de uma aplicação MPI, a partir do processo de identificador 0. Como a informação a ser transmitida é armazenada em posições de memória não consecutivas, é necessário invocar o número de processos inicializados na função. A função possuem os mesmos parâmetros

(`count`, `datatype`, `root` e `comm`) que anteriormente. Para compilar o programas MPI, devemos incluir a opção de compilação apropriada, como:

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

Para executar um programa com MPI com múltiplos recursos computacionais (exemplo com 4 processos):

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

Algoritmo 1 Trecho de código que implementa uma operação de `Broadcast` entre multiprocesadores utilizando MPI. O código copia um vetor de 8 posições a todos os recursos computacionais do grupo `comm`, a partir do identificador 0, multiplicando cada elemento por 2 ao seu 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();

...
```

A seguir demonstra-se um código com a mesma funcionalidade utilizando a función `ncclBcast`. Esta função permite difundir uma informação para múltiplas GPUs que estiverem sobre o mesmo grupo de execução, o qual segue o esquema da Figura 1.

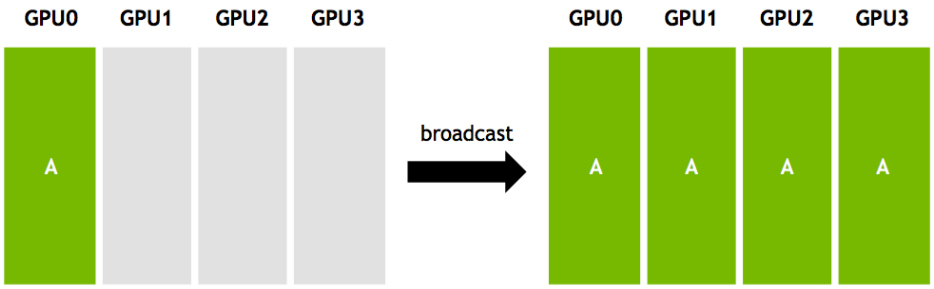

Figura 1: Esquema de funcionamento da operação de `Broadcast` no grupo `comm`, com 4 recursos. O recurso `root` é o que tem o identificador 0.

Para compilar programas *NCCL*, devemos incluir a opção de compilação adequada, como:

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

Para ejecutar um programa com *NCCL* com várias GPUs, podemos utilizar o seguinte exemplo:

```
$ ./object_nccl
```

Algoritmo 2 Trecho de código que implementa uma operação de `Broadcast` entre múltiplas GPUs utilizando *NCCL*. A função deste código é similar a 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 Reduce

Uma operação de *Reduce* ou *Redução* é uma operação onde cada recurso computacional envolvido contribui com um operando para realizar o cálculo global de uma operação associativa ou comutativa (por exemplo: máximo, mínimo, soma, produto, etc). Em *NCCL* a função responsável que executa esta operação é `ncclReduce`, sendo equivalente à função `MPI_Reduce` do 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
          );
```

O quadro acima mostra o esquema comparativo das funções `ncclReduce` e `MPI_Reduce`. Da mesma forma que antes, as duas funções são idênticas. Na operação de *Reduce* o operador é aplicado aos dados localizados no *buffer* de envio (`sendbuff`) de cada recurso computacional. O resultado dessa função é transmitido de volta para todos os recursos no buffer de recebimento (`recbuff`). Os parâmetros `count` e `datatype` especificam novamente a quantidade de memória que o recurso `root` deve enviar para outros, e o espaço que eles devem reservar para armazenar a mensagem recebida. E novamente a diferença está no último parâmetro chamado `stream`, que representa o formato do *buffer* de envío entre as GPUs.

Para mostrar um exemplo do uso das funções de *Reduce*, usaremos o exemplo do *produto escalar* de vetores. O código fará com que o produto escalar [1] de dois vetores x e y distribuídos como mostrado a seguir. Primeiro, o resultado parcial $(x * y)$ é calculado e, em seguida, a operação de *Reduce* é executada. Após a operação o resultado final será armazenado no recurso de origem, chamado recurso 0. Esta operação de redução é executada em *NCCL* pela função `ncclReduce` em sistemas multi-GPU (Figura 2).

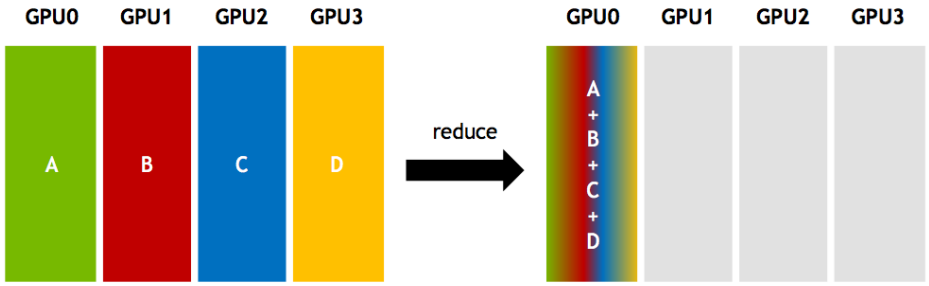

Figura 2: Esquema de funcionamento da operação de `Reduce` no grupo `comm`, com 4 recursos computacionais.

Trecho de código em que se utiliza a función `MPI_Reduce`. Esta função permite que a redução do produto vetores em sistemas multiprocesadores utilizando MPI.

A continuação mostra-se um exemplo de código em que se utiliza a função `ncclReduce`. Esta função permite a operação de redução do produto de vetores sob multi-GPU utilizando *NCCL*.

Algoritmo 3 Trecho de código que implementa uma operação de `Reduce` entre multiprocessadores utilizando MPI. O código copia um vetor de 4 posições a todos os recursos computacionales do grupo `comm`, e a partir do recurso 0, calcula-se o produto escalar dos vetores x e y, distribuídos equitativamente entre as 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 Trecho de código que implementa uma operação de redução entre múltiplas GPUs utilizando *NCCL*. As funcionalidades deste código são similares ao 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 Gather

Uma operação de *Gather* ou *Coleção*, é uma operação coletiva onde um recurso computacional varre as informações a partir de un conjunto de recursos. Desde um ponto de vista, a operação de `Gather` é inversa

a operação de `Scatter` [3]. A diferença com respeito a esta última reside em uma combinação de dados por parte de um recurso receptor, o qual unicamente são armazenados. A sintaxe da função de *Gather* para GPUs corresponde ao comando `ncclAllGather` que está relacionado ao conceito de *AllGather*, que é uma invocação da rotina equivale a realização de p chamadas à operação de *Gather*, em que cada vez atua como `root`, isto é um 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 os processos do comunicador `comm` enviam o conteúdo `sendbuff` ao processo com identificador `root`. O mesmo concatena todos os dados recebidos ordenados pelo identificador do emissor, a partir da posição apontada por `recvbuff`. Isso quer dizer, que os datos do proceso com identificador 0 são armazenados antes que os recurso 1, e assim sucesivamente. Os argumentos de recepção (`recvdbuff`, `recvcount`, `recvtype`), somente tem significado para o recurso `root` em MPI. Em *NCCL* estas informação estão implícitas nos argumentos da função.

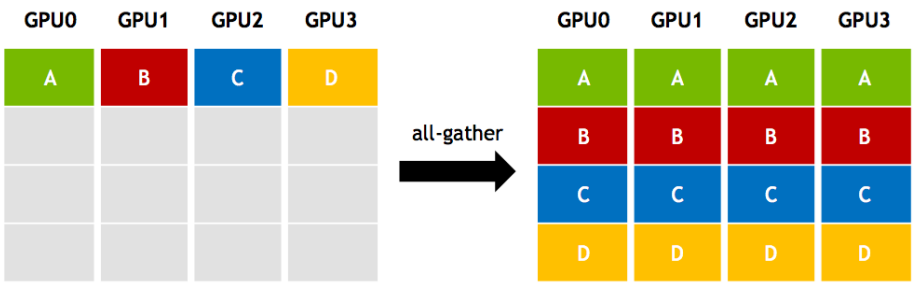

Figura 3: Esquema de funcionamento da função `AllGather` no grupo `comm`, com 4 recursos.

O trecho do código em que a função `MPI_Allgather` é usada é apresentada a seguir. Esta função permite `AllGather` em sistemas multiprocessadores usando MPI.

Algoritmo 5 Trecho de código que implementa uma operação de `AllGather` entre multiprocessadores utilizando MPI. O código copia um vetor de 4 posições com o conteúdo dos identificadores de cada recurso a todos do 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 Trecho de código que implementa uma operação de `AllGather` entre múltiplas GPUs utilizando *NCCL*. As funcionalidades deste código são similares a 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 ReduceScatter

A operação de *ReduceScatter*, é uma operação coletiva presente em *NCCL* que mescla duas operações em uma. A operação de *Reduce* aplicada a operação de *Scatter*, a qual se aplica uma operação de redução distribuindo blocos operados entre os recursos computacionais tendo como base em seu índice identificador.

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

O processo `root` concatena todos os dados recebidos classificados pelo intervalo do remetente, começando da posição apontada por `recbuff`. A partir da posição apontada por `recbbuff`, o processo `root` concatena todos os dados recebidos ordenados pelo intervalo do receptor. Ou seja, os dados parciais das linhas de todos os recursos computacionais são armazenados de forma reduzida nos recursos de destino (Figura **??**).

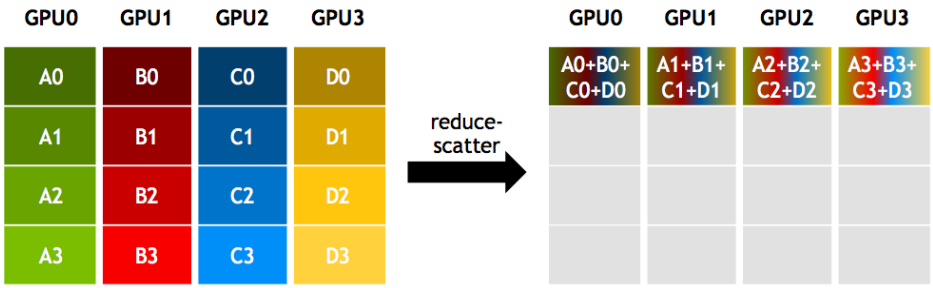

Figura 4: Esquema de funcionamento da operação de *ReduceScatter* no grupo `comm`, com 4 recursos.

Apresenta-se a continuação da parte do código no que se utiliza a função `ncclReduceScatter`. Esta função permite aplicar o *ReduceScatter* em sistemas multi-GPU utilizando *NCCL*.

Algoritmo 7 Trecho de código que implementa uma operação de `ReduceScatter` entre múltiplas 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();
  }

...
```

2.5 Comunicações Ponto a Ponto (P2P)

A comunicação ponto a ponto pode ser usada para expressar qualquer padrão de comunicação entre múltiolas GPUs. Qualquer comunicação ponto a ponto precisa de duas chamadas *NCCL*: uma chamada para envio da mensagem (*ncclSend*) e outra para o seu respectivo recebimento (*ncclRecv*), sendo que toda mensagem tem que ter a mesma contagem e tipificação de dados. Múltiplas chamadas para *ncclSend* e *ncclRecv* podem ser combinadas com *ncclGroupStart* e *ncclGroupEnd* para formar padrões de comunicação mais complexos, isto é, a semântica *NCCL* permite todas as variantes com diferentes tamanhos, tipos de dados e *buffers*, por classificação, como por exemplo: comunições de dispersões, reuniões ou comunicação entre vizinhos em espaços N-dimensionais. A seguir se mostra a sintaxe das rotinas *ncclSend* e *ncclRecv* comparados com os seus respectivos anacrônimos em 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
                                                              );
```

As comunicações ponto a ponto dentro de um grupo serão assimétricas e bloqueantes até que o grupo de chamadas seja concluído, mas as chamadas dentro de um grupo podem ser vistas como progredindo de forma independente, portanto, nunca devem bloquear umas às outras. De forma analóga ao MPI, uma operação ponto a ponto pode ser expressa da seguinte forma:

Algoritmo 8 Trecho de código que implementa uma operação de comunicação ponto a ponto utilizando `ncclSend` e `ncclRecv` entre múltiplas GPUs utilizando *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 Exercícios Propostos

3.1 Cálculo do Número PI através da Integral de Riemann

A partir do código MPI a seguir, escreva um programa paralelo usando *NCCL* que faz uso das funções de comunicação coletiva de *Broadcast* e *Reduce* para o cálculo do número PI através da Integração da função $1/(1+x^2)$, onde a integral é aproximada pela soma 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 AllReduce em Sistemas GPUs

A partir da definição da função `ncclAllReduce`, escreva um programa paralelo usando *NCCL* que faz uso da função de comunicação coletiva de *AllReduce*.

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

3.3 Produto matriz-vetor em multi-GPU

Crie um programa paralelo para sistemas multi-GPU que execute um produto matriz-vetor ($y = Ax$) usando uma distribuição de dados em que a matriz A e o vetor de resultado y são distribuídos por blocos por linhas de tamanho b, e o vetor x é disseminado em sua totalidade para todos os recursos computacionais. Uma vez que os produtos parciais tenham sido feitos, o recurso computacional P_i deve ter apenas o bloco y_i de y. A maneira mais simples de executar esta operação é por meio da operação *AllGather* do bloco armazenado em todos os recursos do grupo. Abaixo está a solução MPI da reunião múltipla dos n elementos do vetor e distribuída entre todos os recursos (y_{distr}) do grupo.

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

  int P;                              // Número de recursos 'P'

  MPI_Comm_size(MPI_COMM_WORLD, &P); // Obtenção do número de recursos 'P'

  send_count =  n / P;                // 'send_count' gera o valor em função de 'n' e 'P'

  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)