Tecnologias Paralelas

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

Murilo Boratto

Índice

1. Resumo

MPI (Message Passing Interface) é uma API padrão para comunicação de dados por meio de mensagens entre processos distribuídos, comumente usada em HPC para criar aplicações que podem ser escalonadas em supercomputadores com múltiplos nós. Como tal, MPI é totalmente compatível com CUDA, que é projetado para computação paralela em um ou mais nós. Existem muitos motivos para querer combinar a programação paralela em MPI e CUDA potencializando os recursos de forma máxima. Um motivo comum é permitir a solução de problemas com um tamanho de dados muito grande para caber na memória de recursos computacionais GPU. Outro motivo é acelerar uma aplicação MPI existente para multi-GPU sobre um único nó existente ou para múltiplos. Com CUDA-AWARE-MPI [2], esses objetivos podem ser alcançados de forma fácil e eficiente. Nesta seção, explicaremos como funciona a compatibilidade entre MPI e CUDA, e o quão eficiente é e como pode ser usada.

2. Benchmarks Ping Pong

Neste guia, veremos um código simples *ping pong* [1] que mede a largura de banda para transferências de dados entre 2 processos MPI. Veremos as seguintes versões:

1. Uma primeira versão utilizando CPU com MPI;
2. Uma segunda versão com MPI + CUDA entre duas GPUs as quais processam dados através da memória da CPU;

3. E uma última que utiliza o CUDA-AWARE-MPI a qual trocam dados diretamente entre GPUs utilizando o GPUdirect ou pelo NVLINK.

2.1. MPI

Começaremos examinando uma versão do código somente para CPU para entendermos a ideia por trás de um programa simples de transferência de dados (*ping pong*). Basicamente, os processos MPI passam dados de um lado para outro e a largura de banda é calculada medindo as transferências, já que se sabe o tamanho que está sendo transferido. Vejamos o código `ping-pong-MPI.c` para ver como ele é implementado. No topo do programa principal, inicializamos o MPI, determinamos o número total de processos, os identificadores dos ranks e nos certificamos de que temos apenas 2 classificações no total para executar o *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;
```

Em seguida, entramos no *loop* principal `for`, onde cada iteração realiza transferências de dados e cálculos de largura de banda para um tamanho de mensagem diferente, variando de 8 bytes a 1 GB:

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

Em seguida, inicializamos o array `A`, definimos alguns rótulos para corresponder aos pares de envio/recebimento MPI.

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

Basicamente, cada iteração do *loop* faz o seguinte:

- ◇ Se tiver rank 0, primeiro envia uma mensagem com os dados da matriz `A` para o rank 1 e, a seguir, espera receber uma mensagem do rank 1.
- ◇ Se o rank 1, primeiro espere receber uma mensagem da rank 0 e, em seguida, envie uma mensagem de volta para a 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();
```

Os dois pontos anteriores descrevem uma transferência de dados da aplicação *ping pong* entre os ranks MPI, enquanto a execução é mensurada. Em seguida, a partir dos resultados da sincronização e do tamanho conhecido das transferências de dados, calculamos a largura de banda e imprimimos os 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);
```

Lembre-se de que, para compilar programas MPI, devemos incluir a opção de compilação apropriada, como:

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

Para executar o programa *ping pong* MPI, impreterivelmente utilizaremos 2 processos:

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

2.2. MPI + CUDA

Agora que estamos familiarizados com o código básico *ping pong* em MPI, vamos dar uma olhada em uma versão que inclui GPUs com CUDA. Neste exemplo, ainda estamos passando dados de um lado para outro entre dois ranks MPI, sendo que desta vez os dados estão na memória da GPU. Mais especificamente, o rank 0 tem um buffer de memória na GPU 0 e o rank 1 tem um buffer de memória na GPU 1, e eles passarão os dados entre as memórias das duas GPUs. Aqui, para obter dados da memória da GPU 0 para a GPU 1, primeiro colocaremos os dados na memória da CPU (*host*). A seguir, conseguimos perceber as diferenças da versão anterior para a nova versão com 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);
      }
}
```

Semelhante à versão somente para CPU, inicializamos MPI e encontramos o identificador de cada rank MPI, mas aqui também atribuímos a cada rank uma GPU diferente (ou seja, a classificação 0 é atribuída à GPU 0 e a classificação 1 é mapeada para a GPU 1), através do comando CUDA `cudaSetDevice`.

```
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 versão, cada iteração do *loop* faz o seguinte:

1. Entramos no *loop* `for` principal que itera sobre os diferentes tamanhos de mensagem, atribuímos e inicializamos o array `A`. No entanto, agora temos uma chamada para `cudaMalloc` para reservar um buffer de memória (`d_A`) nas GPUs e uma chamada para `cudaMemcpy` para transferir os dados inicializados no array `A` para o buffer `d_A`. Faz-se necessário o comando `cudaMemcpy` para levar os dados para a GPU antes de iniciarmos nosso *ping pong*.

2. Os dados devem primeiro ser transferidos da memória GPU 0 para a memória da CPU. Em seguida, uma chamada MPI é usada para passar os dados dos ranks 0 ao 1. Agora que o rank 1 possui os dados (na memória da CPU), ela pode transferir para a memória da GPU 1. Ou seja, o rank 0 deve primeiro transferir os dados de um buffer na memória da GPU 0 para um na memória da CPU. Agora que o rank 1 contém os dados no buffer de memória da CPU, você pode transferi-los para a memória da GPU 1.

Como no caso em que apenas a CPU é utilizada, a partir dos resultados da sincronização e do tamanho conhecido das transferências de dados, calculamos a largura de banda e imprimimos os resultados e finalmente liberamos a memória dos recursos computacionais. Encerramos o MPI e o programa.

De forma similar para compilar programas MPI + CUDA, devemos incluir a opção de compilação apropriada, como:

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

Para executar o *ping pong* MPI+CUDA, impreterivelmente utilizaremos 2 processos:

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

2.3. CUDA-AWARE-MPI

Antes de olhar para este exemplo de código, vamos primeiro descrever CUDA-AWARE-MPI e GPUDirect. CUDA-AWARE-MPI é uma implementação de MPI que permite que buffers de GPU (por exemplo, memória de GPU alocada com `cudaMalloc`) sejam usados diretamente em chamadas MPI. No entanto, o CUDA-AWARE-MPI por si só não especifica se os dados são armazenados em estágios intermédios na memória da CPU ou são passados diretamente de GPU para a GPU, dependerá da estrutura computacional do entorno de execução.

Enquanto que GPUDirect é uma mistura de solução de hardware e software para o CUDA-AWARE-MPI, permitindo transferências de dados diretamente entre GPUs no mesmo nó (*peer-to-peer*) ou diretamente entre GPUs em nós diferentes (suporte RDMA), sem a necessidade de armazenar dados na memória da CPU.

Agora vamos dar uma olhada no código abaixo. É essencialmente o mesmo que a versão testada de MPI+CUDA, mas agora não há chamadas para `cudaMemcpy` durante as etapas de *ping pong*. Em vez disso, usamos nossos buffers de GPU (`d_A`) diretamente nas chamadas 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);
        }
    }
```

Como anteriormente para compilar os códigos com CUDA-AWARE-MPI deveremos incluir as opções de compilação adequada, como por exemplo:

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

E da mesma forma para executar o programa *ping pong* CUDA-AWARE-MPI, impreterivelmente utilizaremos 2 processos:

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

3. Exercícios Propostos

3.1. Multiplicação de Matrizes

Implemente uma multiplicação de matrizes usando a API CUDA-AWARE-MPI onde as matrizes são geradas no processo 0, e todos os processos trabalham em partes da multiplicação. O resultado final será finalmente compilado no processo 0. Faça a implementação com uma distribuição por blocos de linhas e no final compare os tempos de execução variando o tamanho do 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/`