问题 1

实验思路

单 CPU 方法和 MPI 方法写在一个文件中，会导致程序代码过于复杂，因此分开编写两个程序。两个程序共用的函数放在一个头文件中以便调用，其中 matrixMatrixMul 为计算两个矩阵的乘积，sumArray 为对矩阵的值求和，目的是验证计算结果，代码如下图所示。

lab3-code > C MMM.h > ...

```c
void matrixMatrixMul(float *A, float *B, float *C, int m, int k, int n)
{
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            float temp = 0;
            for (int a = 0; a < k; a++)
            {
                temp += A[i * k + a] * B[a * n + j];
            }
            C[i * n + j] = temp;
        }
    }
}

float sumArray(float *C, int m, int n)
{
    float res = 0;
    for (int i = 0; i < m * n; i++)
    {
        res += C[i];
    }
    return res;
}
```

引入头文件，读取参数后，进行 MPI 初始化，对矩阵 A 按行划分，srow 为每个进程的计算的行数。

```c
#include "MMM.h"

int main(int argc, char **argv)
{
    if (argc != 4)
    {
        printf("Need Three parameters!\n");
        return -1;
    }
    int m = strtol(argv[1], NULL, 10);
    int k = strtol(argv[2], NULL, 10);
    int n = strtol(argv[3], NULL, 10);
    double startTime=MPI_Wtime();
    int pid, np;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    float *A, *B, *C;
    int srow = m / np;
```

对于 master 节点，首先分配矩阵的空间，然后初始化矩阵 A 和 B 的值。接着，把 B 矩阵以广播的形式发送给其他进程，并发送线程需要计算的矩阵 A 的行数据。master 节点只计算自己需要计算的部分，最后，收集其他线程计算的结果到 C 矩阵中。

```
if (pid == 0)
{
    // init
    A = (float *)malloc(m * k * sizeof(float));
    B = (float *)malloc(k * n * sizeof(float));
    C = (float *)malloc(m * n * sizeof(float));
    for (int i = 0; i < m * k; i++)
    {
        A[i] = 1.0 * rand() / RAND_MAX;
    }
    for (int i = 0; i < k * n; i++)
    {
        B[i] = 1.0 * rand() / RAND_MAX;
    }
    // send
    MPI_Bcast(B, k * n, MPI_FLOAT, 0, MPI_COMM_WORLD);
    for (int i = 1; i < np; i++)
    {
        MPI_Send(A + i * srow * k, srow * k, MPI_FLOAT, i,
                 0, MPI_COMM_WORLD);
    }
    // cal
    matrixMatrixMul(A, B, C, srow, k, n);
    // collect
    for (int i = 1; i < np; i++)
    {
        MPI_Recv(C + i * srow * n, srow * n, MPI_FLOAT, i, 1,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
```

对于 slave 节点，B 矩阵需要被分配所有空间，而 A、C 矩阵只被分配需要计算的行。计算后将结果向量发送给 master，这里设置 tag 为 1 便于匹配。

```
else
{
    A = (float *)malloc(srow * k * sizeof(float));
    B = (float *)malloc(k * n * sizeof(float));
    C = (float *)malloc(srow * n * sizeof(float));
    MPI_Bcast(B, k * n, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Recv(A, srow * k, MPI_FLOAT, 0, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    matrixMatrixMul(A, B, C, srow, k, n);
    MPI_Send(C, srow * n, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
}
```

最后，计算 C 矩阵的和来判断两种方法执行的结果是否一致。对于获取时间的函数，单 CPU 中使用了 clock()，MPI 中使用了 MPI_Wtime()。

实验结果

测试 1000*1000 的两个矩阵相乘所需时间。在单 CPU 上计算，用时 4.78s，

而使用了 2 进程的 MPI 方法后，用时 2.51s，提升将近 2 倍。根据 sum 的值也可以看到计算的结果是一致的。

```
xingzm@ubuntu-GPU:~/lab3-code$ ./cpuMMM 1000 1000 1000
sum: 250000304.000000, time: 4.784724s
```

```
xingzm@ubuntu-GPU:~/lab3-code$ mpiexec -n 2 ./mpiMMM 1000 1000 1000
sum: 250000304.000000, time: 2.508024s
```

当把线程数设置为 20 后，计算仅用 0.47s，比单 CPU 方法提升了 10 倍。

```
xingzm@ubuntu-GPU:~/lab3-code$ mpiexec -n 20 ./mpiMMM 1000 1000 1000
sum: 250000304.000000, time: 0.469779s
```

问题 2

实验思路

首先，程序读取命令行参数，输入数组大小和归约函数。这里数字 1 代表求和，数字 2 代表求最大值。读取参数后，为数组随机生成浮点数。

```
if (argc != 3)
{
    printf("Need two parameters!\n");
    return -1;
}
int n = strtol(argv[1], NULL, 10);
int op = strtol(argv[2], NULL, 10);
if (op != 1 && op != 2)
{
    printf("The second parameter must be 1 or 2.\n");
    printf("1: SUM, 2: MAX\n");
    return -1;
}
float *arr = (float *)malloc(n * sizeof(float));
for (int i = 0; i < n; i++)
{
    arr[i] = 1.0 * rand() / RAND_MAX;
}
```

划分进程，每个进程中并求部分和。

```
int pid, np;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
int elements_per_process = ceil(n * 1.0 / np);
int i, init = elements_per_process * pid;
float local_sum = 0;
for (i = init; i < init + elements_per_process && i < n; i++)
{
    local_sum += arr[i];
}
```

首先使用 OpenMPI 提供的 MPI_Allreduce 方法，计算数组的和或者最大值。各个线程均打印结果，线程 0 额外打印运行时间，运行时间计算的是从执行归约操作开始，到所有线程归约结束的时间。

```
{ // MPI_Allreduce
    if (pid == 0)
    {
        printf("MPI_Allreduce:\n");
    }
    float global_sum;
    double startTime = MPI_Wtime();
    MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT,
                  op == 1 ? MPI_SUM : MPI_MAX, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    double endTime = MPI_Wtime();
    printf("pid = %d, %s = %f\n", pid, op == 1 ? "sum" : "max", global_sum);
    if (pid == 0)
    {
        printf("--------------MPI_Allreduce time = %lfs\n",
               endTime - startTime);
    }
}
```

在自己设置的基于环的归约算法中，首先从左到右进行叠加求解，一个进程使用 MPI_Recv 接收前一个线程的数据，进行求和或者最大值的处理，然后使用 MPI_Send 发送给下一个线程。使用取模运算达到环形数组的效果。由于进程 0 不从其他进程接收数据，为了防止死锁，需要对进程 0 进行特殊处理。

```
void customRingBasedAllreduce(float local_sum, float *global_sum, int op, int pid, int np)
{
    float sum = local_sum;
    float tmp;
    int nextPid = (pid + 1) % np;
    int prePid = (pid - 1 + np) % np;
    // scatter
    if (pid == 0)
    {
        MPI_Send(&sum, 1, MPI_FLOAT, 1, 666, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Recv(&tmp, 1, MPI_FLOAT, prePid, 666, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (op == 1)
        {
            sum += tmp;
        }
        else
        {
            sum = fmax(tmp, sum);
        }
        MPI_Send(&sum, 1, MPI_FLOAT, nextPid, 666, MPI_COMM_WORLD);
    }
```

此时，最后一个进程的值为前面归约得到的最终结果。接下来，再次从线程 0 开始，获取前一个进程的最终结果，设置为自己的最终结果，达到了每个线程都归约到相同的值的目的。这里只需要计算到倒数第 2 个线程，而且倒数第 2 个线程中无发送指令，需要特殊处理。

```
// gather
if (pid < np - 2)
{
    MPI_Recv(&tmp, 1, MPI_FLOAT, prePid, 666, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    sum = tmp;
    MPI_Send(&sum, 1, MPI_FLOAT, nextPid, 666, MPI_COMM_WORLD);
}
else if (pid == np - 2)
{
    MPI_Recv(&tmp, 1, MPI_FLOAT, prePid, 666, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    sum = tmp;
}
*global_sum = sum;
```

实验结果

使用 10 个进程来计算 10000 个浮点数的和，可以看到每个线程最后都输出了相同的结果。而且我们编写的基于环的归约操作与 MPI_Allreduce 执行的结果一致，且我们编写的函数执行速度约是官方函数的 2.4 倍，猜测是因为我们编写的方法较为轻量，没有考虑方法的完整性，没有对复杂的业务逻辑进行处理。

```
xingzm@ubuntu-GPU:~/lab3-code$ mpirun -n 10 ./SumArrayCol 10000 1
MPI_Allreduce:
[MPI] pid = 7, sum = 4971.322266
[MPI] pid = 4, sum = 4971.322266
[MPI] pid = 3, sum = 4971.322266
[MPI] pid = 9, sum = 4971.322266
[DIY] pid = 9, sum = 4971.322266
[MPI] pid = 5, sum = 4971.322266
[DIY] pid = 5, sum = 4971.322266
[MPI] pid = 2, sum = 4971.322266
[DIY] pid = 2, sum = 4971.322266
[MPI] pid = 6, sum = 4971.322266
[DIY] pid = 6, sum = 4971.322266
[MPI] pid = 8, sum = 4971.322266
[DIY] pid = 8, sum = 4971.322266
[MPI] pid = 0, sum = 4971.322266
--------------MPI_Allreduce time = 0.000164s

custom ring-based Allreduce:
[DIY] pid = 0, sum = 4971.322266
custom ring-based Allreduce time = 0.000069s
[MPI] pid = 1, sum = 4971.322266
[DIY] pid = 1, sum = 4971.322266
[DIY] pid = 7, sum = 4971.322266
[DIY] pid = 3, sum = 4971.322266
[DIY] pid = 4, sum = 4971.322266
```

对于数组取最大值功能，可以看到程序的执行结果相似。

```
xingzm@ubuntu-GPU:~/lab3-code$ mpirun -n 10 ./SumArrayCol 10000 2
MPI_Allreduce:
[MPI] pid = 3, max = 509.957153
[MPI] pid = 2, max = 509.957153
[MPI] pid = 4, max = 509.957153
[MPI] pid = 1, max = 509.957153
[MPI] pid = 5, max = 509.957153
[MPI] pid = 0, max = 509.957153
--------------MPI_Allreduce time = 0.000194s

custom ring-based Allreduce:
[MPI] pid = 7, max = 509.957153
[MPI] pid = 9, max = 509.957153
[MPI] pid = 6, max = 509.957153
[MPI] pid = 8, max = 509.957153
[DIY] pid = 3, max = 509.957153
[DIY] pid = 1, max = 509.957153
[DIY] pid = 5, max = 509.957153
[DIY] pid = 4, max = 509.957153
[DIY] pid = 2, max = 509.957153
[DIY] pid = 9, max = 509.957153
[DIY] pid = 7, max = 509.957153
[DIY] pid = 0, max = 509.957153
custom ring-based Allreduce time = 0.000089s
[DIY] pid = 6, max = 509.957153
[DIY] pid = 8, max = 509.957153
```