Lab2 - 基于MPI的并行矩阵乘法（进阶）

实验要求

改进上次实验中的MPI并行矩阵乘法(MPI-v1)，并讨论不同通信方式对性能的影响。

输入：m, n, k 三个整数，每个整数取值范围均为[128, 2048]

问题描述：随机生成m×n的A矩阵及n×k的B矩阵，矩阵相乘得到C矩阵

输出：A, B, C三个矩阵，以及矩阵计算时间 t

要求：

1. 采用MPI集合通信实现并行矩阵乘法中的进程间通信；使用mpi_type_create_struct聚合MPI进程内变量后通信。
2. 对于不同实现方式，调整并记录不同进程数量（1-16）及矩阵规模（128-2048）下的时间开销，填写下页表格，并分析其性能及扩展性。

实验过程

1. 实现思路

改动上次实验的代码，先将点对点通信转为集合通信，矩阵A的分发使用`MPI_Scatter`，矩阵B整个分发使用`MPI_Bcast`,矩阵C计算后通过`MPI_Gather`收集到一起。使用mpi_type_create_struct聚合MPI进程内变量后通信，进程之间共享的变量主要是矩阵的规模，因此将m、n、k聚合到一个结构体中使用`MPI_Bcast`通信。

2. 代码实现

定义结构体

```
struct ShareData {
    int m, n, k;
};
```

创建MPI数据型

```
struct ShareData temp;
MPI_Datatype types[3] = {MPI_INT,MPI_INT,MPI_INT };
int block_lengths[3] = { 1, 1, 1 };
MPI_Aint address[3], startaddr;
MPI_Aint displacements[3];

MPI_Get_address(&temp, &startaddr);
MPI_Get_address(&temp.m, &address[0]);
MPI_Get_address(&temp.n, &address[1]);
MPI_Get_address(&temp.k, &address[2]);
displacements[0] = address[0] - startaddr;
displacements[1] = address[1] - startaddr;
displacements[2] = address[2] - startaddr;

MPI_Datatype data_type;
```

```
MPI_Type_create_struct(3, block_lengths, displacements, types, &data_type);
MPI_Type_commit(&data_type);
```

计算分配到每个进程的A矩阵元素个数`send_couts`、收集的C矩阵元素个数`recv_counts`及其偏移量`send_offset`、`recv_offset`.

```
int rows_per_process = m / comm_sz;
int remaining_rows = m % comm_sz;
int soffset = 0, roffset = 0;
for (int i = 0; i < comm_sz; i++) {
    int tmp = (i < remaining_rows) ? rows_per_process + 1 : rows_per_process;
    send_counts[i] = k * tmp;
    recv_counts[i] = n * tmp;
    send_offset[i] = soffset;
    recv_offset[i] = roffset;
    soffset += send_counts[i];
    roffset += recv_counts[i];
}
```

集合通信，使用`MPI_Scatterv`和`MPI_Gatherv`用于划分进程间不均衡情况的分配

```
// process 0
MPI_Scatterv(A, send_counts, send_offset, MPI_DOUBLE, MPI_IN_PLACE, 0,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(B_T, k * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
double* C_partial = (double*)malloc(send_counts[0] * n * sizeof(double));
mpi_multiple(A, B_T, C_partial, send_counts[0]/k, n, k);
MPI_Gatherv(C_partial, recv_counts[0], MPI_DOUBLE, C, recv_counts, recv_offset,
MPI_DOUBLE, 0, MPI_COMM_WORLD);

...
// other processes
MPI_Bcast(&recv_data, 1, data_type, 0, MPI_COMM_WORLD);
m = recv_data.m; n = recv_data.n; k = recv_data.k;
rows = m / comm_sz;
if (my_rank < m % comm_sz) rows++;
double* A_partial = (double*)malloc(rows * k * sizeof(double));
double* C_partial = (double*)malloc(rows * n * sizeof(double));
double* B_T = (double*)malloc(k * n * sizeof(double));
MPI_Scatterv(NULL, NULL, NULL, MPI_DOUBLE, A_partial, rows * k, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
MPI_Bcast(B_T, k * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
mpi_multiple(A_partial, B_T, C_partial, rows, n, k);
MPI_Gatherv(C_partial, rows * n, MPI_DOUBLE, NULL, NULL, NULL, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
```

4. 运行结果

```
// 编译
mpicc -g -Wall -o mpi CollMatMulti.c
// 运行
mpiexec -n 4 ./mpi
```

```
ehpc@61583b2ed2e2:~/data$ mpicc -g -Wall -o mpi CollMatMulti.c
ehpc@61583b2ed2e2:~/data$ mpiexec -n 4 ./mpi
Enter values for m, k, n (128-2048): 128 128 128
Computations completed.
 Top left corner of matrix A:
          83          86          77          15          93          35
           9          27          67          56          97          53
          20          36          44          26          22          65
          21          67           4          13          61          54
          77          34          90          26          24          57
          38          80          18          21          70          62
 Top left corner of matrix B:
          91          24          44          86           5           6
          39           2          76          82          99          77
          27          73          99          92          18          64
          32          13          11           9          76          75
          13          71          81          51          21          69
          48          79          40          72          56          88
Top left corner of matrix C:
  3.0843E+05  3.6042E+05  3.2182E+05  3.3676E+05  2.9544E+05  3.4383E+05
  2.6366E+05  3.0842E+05  2.7344E+05  2.8296E+05   2.539E+05  2.8995E+05
   3.097E+05  3.7485E+05  3.1452E+05  3.2377E+05  3.0331E+05  3.5726E+05
  2.8937E+05  3.4565E+05  2.9142E+05  2.9112E+05   2.761E+05  3.3693E+05
  2.9814E+05  3.5684E+05  3.0174E+05  3.3002E+05   3.026E+05   3.343E+05
  3.0545E+05   3.253E+05  3.0891E+05  3.1408E+05  3.0063E+05  3.2497E+05
Time taken: 0.005599 seconds
```

性能分析

记录不同进程数量（1-16）及矩阵规模（128-2048）下的时间开销，耗时单位毫秒。

进程数\矩阵规模	128	256	512	1024	2048
1	14.721	139.503	1021.753	8138.542	15714.973
2	17.858	130.696	1007.737	7921.316	15561.947
4	98.321	201.612	1890.624	8588.606	16783.314
8	207.186	1205.538	3896.958	12708.551	22302.425
16	1196.118	4900.356	8089.417	59601.979	91402.021

进程数量的性能比较与上一次实验相似，需要衡量进程通信与并行计算之间的消耗，从单进程到多进程的转变可以看到一定的扩展性，但随着进程数量增加，性能提升并不是线性的，因为通信开销、负载不均衡等因素的影响；同时随着进程数量增加，当16个进程时，集合通信的耗时要小于点对点通信，可见集合通信的性能优势体现在进程较多的情况。