实验2-基于MPI的并行矩阵乘法（进阶）

学号	姓名
20319045	刘冠麟

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，每个整数的取值范围均为$[128, 2048]$

问题描述:

随机生成$m \times n$的矩阵A及$n \times k$的矩阵B，并对这两个矩阵进行矩阵乘法运算，得到矩阵C.

输出:

A, B, C三个矩阵，及矩阵计算所消耗的时间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;尝试不同数据/任务划分方式(选做)。
2. 对于不同实现方式，调整并记录不同线程数量$(1-16)$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。

关键代码解释

由于实验代码的整体框架与实验1一致，只是将点对点通信改为了集合通信，所以这里解释改动的关键部分。

实验要求进行集合通信，主要由以下代码实现

对于master进程

判断进程的rank是否为0，如果为0则为master进程。

- **广播矩阵B**:
    - master进程拥有矩阵B的完整数据，需要把这些数据通过调用`MPI_Bcast`函数广播给所有其他进程。

```
        MPI_Bcast(B.data(), n * k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- **将矩阵A分发**:

  master进程将其持有的矩阵A分发到所有进程，每个进程接收到对应行数的数据：

```
        vector<int> sendcounts(size);
        vector<int> displs(size);

        // 计算每个进程应该接收的行数和起始位置
```

```
        for (int i = 0; i < size; ++i) {
            sendcounts[i] = (i < remaining_rows) ? (rows_per_process + 1)
: rows_per_process;
            displs[i] = (i == 0) ? 0 : (displs[i - 1] + sendcounts[i -
1]);
        }

        // 创建表示一行的数据
        MPI_Datatype row_type;
        MPI_Type_contiguous(n, MPI_DOUBLE, &row_type);
        MPI_Type_commit(&row_type);

        // 通过Scatterv操作将A矩阵分发到各个进程
        MPI_Scatterv(A.data(), sendcounts.data(), displs.data(),
row_type, MPI_IN_PLACE, sendcounts[rank] * n, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
```

- 定义两个整数向量 `sendcounts` 和 `displs`，用于存储每个进程应该接收的A矩阵的行数和每个进程接收数据的起始位置。
- 然后通过循环计算每个进程应该接收的A矩阵的行数和每个进程接收数据的起始位置，这样可以确保矩阵A的行在各个进程之间均匀分配。
- 然后为了让 `MPI_Scatterv` 知道如何打包和解包数据，定义一个MPI数据类型 `row_type` 表示矩阵A的一行。
- 最后用 `MPI_Scatterv` 函数将矩阵A分发到所有进程，每个进程根据 `sendcounts` 和 `displs` 接收相应部分的A矩阵数据。

- **从子进程收集结果**:

  每个进程都计算了矩阵C的一部分并将这些部分发送回master进程。主进程通过调用 `MPI_Gatherv` 函数从各个子进程中收集计算结果。

```
MPI_Gatherv(C.data(), my_rows * k, MPI_DOUBLE, nullptr, nullptr, nullptr,
row_type, 0, MPI_COMM_WORLD);
```

对于子进程

对于rank！=0即子进程的情况：

- **接收矩阵B：**

  首先从主进程接受矩阵B，创建一个动态数组B并通过调用 `MPI_Bcast` 函数从主进程接收矩阵B的全部数据：

```
        vector<double> B(n * k);

        MPI_Bcast(B.data(), n * k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- **接受矩阵A的一部分**

  首先计算出每个进程理论上应该处理的行数和剩余行数，每个进程根据其进程ID确定自己应该处理的A矩阵的行数。如果进程ID小于剩余行数，则该进程将处理额外的一行。

```
        int rows_per_process = m / size;
        int remaining_rows = m % size;
        int my_rows = (rank < remaining_rows) ? (rows_per_process + 1) :
rows_per_process;
```

再通过Scatterv操作接收A矩阵的一部分

```
    MPI_Datatype row_type;
    MPI_Type_contiguous(n, MPI_DOUBLE, &row_type);
    MPI_Type_commit(&row_type);
    // 通过Scatterv操作接收A矩阵的一部分
    vector<double> A(my_rows * n);

    MPI_Scatterv(nullptr, nullptr, nullptr, row_type, A.data(), my_rows *
n,        MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- **计算矩阵乘法并将操作结果发送回master进程**

```
        matrix_multiplication(my_rows, n, k, A, B, C);  // 计算矩阵乘法
        // 通过Gatherv操作将结果发送回master进程
        MPI_Gatherv(C.data(), my_rows * k, MPI_DOUBLE, nullptr, nullptr,
nullptr, row_type, 0, MPI_COMM_WORLD);
```

不同的数据划分方式

改为分发B的列进行计算：

```
void subMatrixMultiplyTranspose(double* matrixA, double* matrixB, double*
matrixC, int rowsA, int colsA, int colsB) {
        for(int i=0;i < rowsA; i++){
            for(int j=0;j < colsB; j++){
                for(int k=0; k < colsA; k++){
                    matrixC[j * colsB + i] += matrixA[i * colsA + k] * matrixB[j
* colsB +i];
                }
            }
        }
void Transpose(double* matrix, int nRows, int nCols){
    int max_row_col = nRows > nCols ? nRows :nCols;
    int t=0;
    for(int i = 0; i < max_row_col - 1; i++){
        for(int j = i + 1; j < max_row_col; j++){
            t = matrix[i * nCols + j];
            matrix[i * nCols + j] = matrix[j * nCols + i];
            matrix[j * nCols + i]= t;
        }
    }
}
```

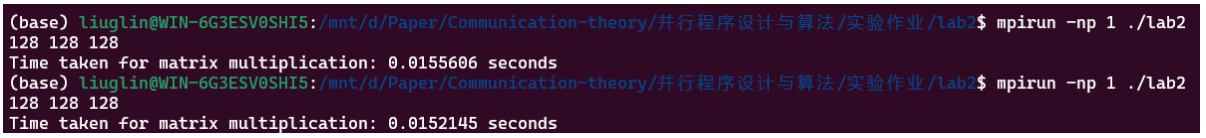
C的每列由A的所有行和B的列计算得到。

编译运行

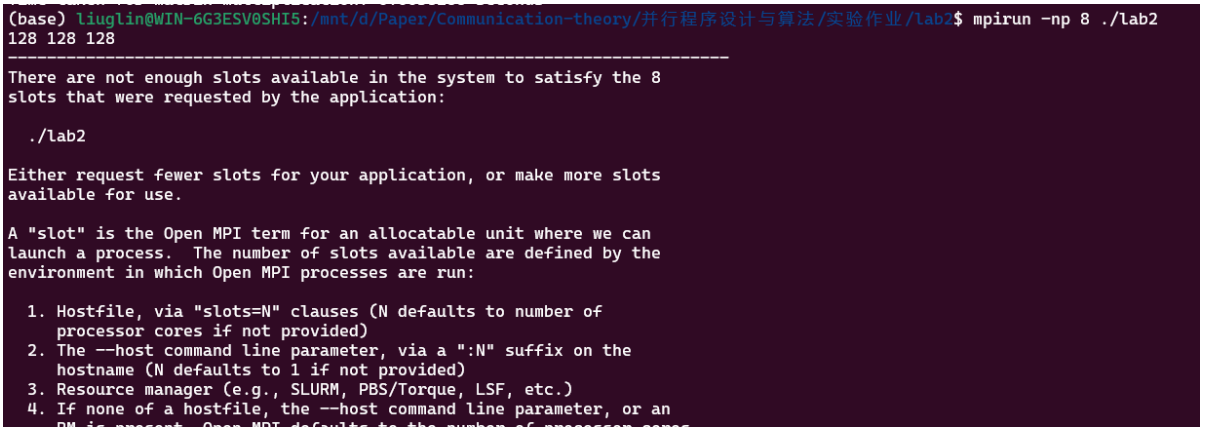
编译运行：

```
mpic++ lab2.cpp -o lab2
mpirun -np 1 ./lab2 128 128 128
```

运行结果如下：



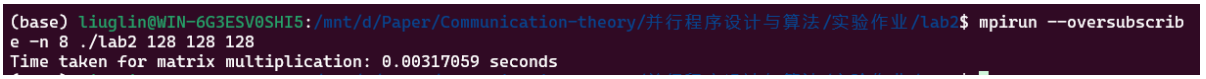
如果指定的进程数过多，数量超过了Open MPI在系统中分配的slots数目，会有如下报错：



这个时候调整运行指令，使用`--oversubscribe`选项来告诉Open MPI允许进程数量超过可用的槽位数。这样就可以让Open MPI忽略槽位数的限制：

```
mpirun --oversubscribe -n 8 ./lab2 128 128 128
```

此时可以运行：



改变进程数和矩阵规模后可得如下表格：

进程数\矩阵规模	128	256	512	1024	2048
1	0.015140	0.1594	1.554	19.75	337.5
2	0.007529	0.0820	0.8221	9.863	182.9
4	0.003826	0.05868	0.5496	5.419	103.9
8	0.003171	0.0320	0.3730	3.890	61.19
16	0.007584	0.0668	0.4729	3.885	63.58

根据表格计算得出加速比和效率如下：

加速比

矩阵大小\进程数	1	2	4	8	16
128	1	2.010891	3.957135	4.774519	1.996308
256	1	1.943902	2.716428	4.981250	2.386228
512	1	1.890281	2.827511	4.166220	3.286107
1024	1	2.002433	3.644584	5.077121	5.083655
2048	1	2.002967	3.248316	5.515607	5.308273

效率

矩阵大小\进程数	1	2	4	8	16
128	1	1.005446	0.989284	0.596815	0.124769
256	1	0.971951	0.679107	0.622656	0.149139
512	1	0.945140	0.706878	0.520777	0.205382
1024	1	1.001217	0.911146	0.634640	0.317728
2048	1	1.001484	0.812079	0.689451	0.331767

根据运行时间，分析程序并行性能及扩展性

- **并行性能**
    - **对于较小的矩阵（如128和256）**，随着进程数的增加加速比的提升并没有达到理想状态（即加速比应等于或接近于进程数）。
    - **对于较大的矩阵（如1024和2048）**，加速比的提升更接近理想状态，尤其是在使用更多进程时。例如，矩阵大小为1024和2048时，使用16个进程的加速比都超过了5倍，接近于理想的加速比16。
- **效率**
    - 效率随着进程数的增加而降低。对于较小的矩阵，效率显著下降，可能是因为程序是**弱拓展性**的，数据量不足以使所有进程都充分利用。
- **扩展性**
    - 该程序对于较大矩阵显示出较好的扩展性，尤其是在增加进程数时，加速比和效率都保持在相对较高的水平。表面该程序是**弱拓展性的。**
    - 对于较小矩阵，程序的扩展性较差。
- **对比点对点通信**
  在数据量较小时集合通信明显慢于点对点，但随着数据量的增大，集合通信的速度将明显快于点对点。而且集合通信的效率也普遍较高。
- **对于16个进程比8个进程的情况更慢**

  虚拟机最多只有8个核，超过8个核后再增加只能是共享核的线程，切换内存反而会导致更慢。