



《高性能计算》

课程考核报告

学生姓名： 刘星雨

班 学 号： 11120322

指导教师： 李程俊

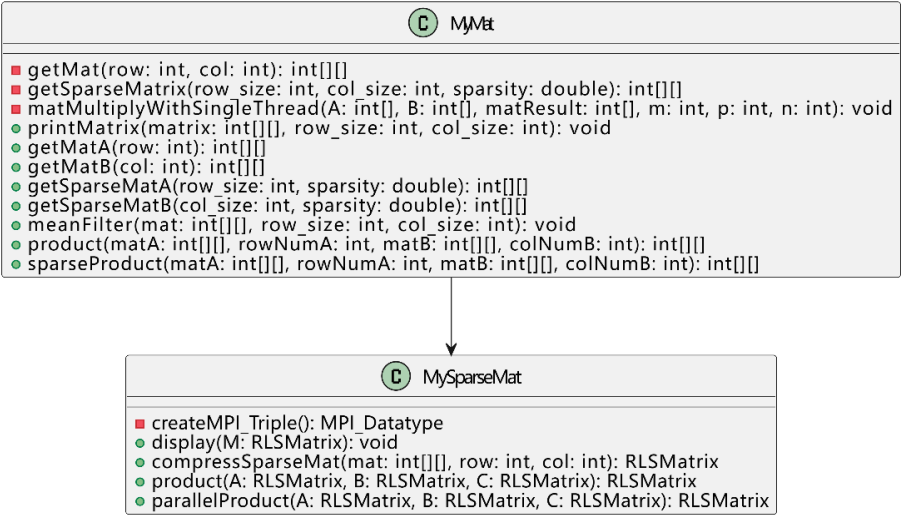
中国地质大学（武汉）计算机学院

2023 年 7 月 1 日

0. 代码结构介绍

注意：本次课程实践的所有代码放在附录。为了方便您的查阅，在报告中讲解算法的部分均由 LaTeX 格式的伪代码、UML 图、流程图、算法逻辑图展示，如果由检查代码的需要，请查看附录。

本次实习充分利用了 C++面向对象的思想，对矩阵的一系列操作封装类单独的类中，旨在与主函数的代码解耦实现可复用性、可维护性、可扩展性。我定义了两种不同类型的类，一个用于封装通用矩阵的方法，包括生成随机通用矩阵、生成学号随机矩阵、生成随机稀疏矩阵（并可以自定义稀疏度）、基于 mpi 的矩阵预处理、基于 mpi 的通用矩阵的乘法、打印 int **二维矩阵；以及一个用于封装稀疏矩阵相关的方法，包括将传统的二维 (int **) 矩阵压缩为行逻辑链接顺序表结构、稀疏矩阵的乘法、基于 mpi 的稀疏矩阵乘法、创建三元表 MPI 数据类型（私有方法）、打印基于行逻辑链接顺序表的稀疏矩阵。文件的 UML 图如下附图 1 所示。



附图 1. 代码的 UML 图

另外，控制台操作逻辑也做了较为完善的处理，利用 *ASCIIFlow Infinity* 提供的 web 服务，生成了字符图形方便用户与程序进行交互和调用。最终控制台界面如下附图 2 所示。

说明：该程序作者是111203刘星雨，用于完成李程俊老师的课程设计题目。内容有三：1 使用类3*3均值滤波对矩阵进行预处理，2 完成通用矩阵乘法，3 完成稀疏矩阵乘法。

注意：矩阵分为 A、B两个，且A矩阵的第一行必须为本人的学号20201003729，这个条件规定了矩阵 A的列数为 11，矩阵 B的行数为 11.综上所述，用户使用该程序时只需要输入矩阵 A的行数和矩阵 B的列数！

输入

矩阵A行数 : -----

矩阵B行数 : -----

矩阵A行数：7

矩阵B列数：7

附图 2. 初始化界面

在输入了 A 矩阵的行数和 B 矩阵的列数以后，会首先打印出两个矩阵的具体内容，随后进入更具体的操作界面，如下附图 3 所示。本次作业的主要功能也集成在该界面中。

```
矩阵A行数： 7
矩阵B列数： 7

matrix A:
第0行: 2 0 2 0 1 0 0 3 7 2 9
第1行: 62 18 28 34 94 5 45 8 9 11 19
第2行: 93 5 85 32 59 85 23 46 56 39 5
第3行: 69 28 80 92 91 2 63 41 71 85 13
第4行: 81 16 73 15 65 99 37 49 65 83 55
第5行: 96 68 44 0 56 20 64 82 39 83 73
第6行: 93 47 94 98 87 33 0 26 6 62 89

matrix B:
第0行: 45 50 10 69 55 24 85
第1行: 74 3 58 61 52 14 78
第2行: 33 10 31 18 33 84 67
第3行: 19 40 14 36 19 19 94
第4行: 20 32 26 98 65 60 54
第5行: 6 86 51 96 41 10 95
第6行: 31 79 84 100 29 100 35
第7行: 11 11 85 56 53 68 56
第8行: 23 76 37 76 64 13 25
第9行: 0 59 13 88 70 77 27
第10行: 49 98 51 3 54 13 98
```

↑

↓

输入数字

1. 重新生成稀疏矩阵A。

2. 重新生成稀疏矩阵B。

3. 对矩阵A进行预处理。

4. 对矩阵B进行预处理。

5. 通用矩阵乘法 $A * B$ 。

6. 稀疏矩阵乘法 $A * B$ 。

7. 退出。

↑

↓

附图 3. 更加具体的操作界面

为了评估程序运行的结果，需要严格使用计数器，以下是我计算运行时长的代码：

```
1. // 记录程序开始时间
2. auto start_time = std::chrono::high_resolution_clock::now();
3. /*
4.     此处运行算法。。。
5. */
6. // 记录程序结束时间
7. auto end_time = std::chrono::high_resolution_clock::now();
8. // 计算程序运行时间
9. auto duration = std::chrono::duration_cast<std::chrono::milliseconds>
    (end_time - start_time).count();
10. // 输出运行时间到日志文件
11. std::ofstream log_file("program_log.txt");
12.
13. if (log_file.is_open()) {
14.     log_file << "程序运行时间: " << duration << " 毫秒\n";
15.     log_file.close();
16. }
17. else {
18.     std::cout << "无法打开日志文件。 \n";
19. }
```

1. 矩阵的预处理

1.1. 算法说明

本次实习所要求的矩阵的预处理，是用每个元素的邻域元素的均值进行异常处理。对于矩阵中任意一个元素 $X_{i,j}$ ，则它的邻域元素的集合可做如下表示：

$$\{X_{m,n} \mid m,n \in Z, -1 \leq m \leq 1, -1 \leq n \leq 1\} \quad (1)$$

可以观察到，邻域元素加上中心元素本身是一个大小为 3×3 的滤波，因此我们的预处理和均值滤波也有一定的相似性。不同点在于，均值滤波是保证所有元素的值都等于其邻域元素的均值，而预处理的作用是消除异常，我们的目标是用均值覆盖存在异常的元素。是否异常的标准将由 σ 判断， $\sigma > 10\%$ 则认为异常，需要被邻域的均值覆盖， σ 的定义如下所示：

$$\sigma = \frac{\sum_{m=-1,n=-1}^{m=1,n=1} X_{m,n}}{Count} \quad (2)$$

其中，Count 表示有效邻域元素的个数，之所以没有用 8 表示，是因为对于边界元素，我们凑不满 8 个邻域元素。从公式 (2) 中可以判断出，我处理边界元素的方法是只计算存在的邻域均值。

1.2. 算法流程图及伪代码

先考虑串行的情况，我们需要对矩阵中的元素进行逐个扫描，通过 3×3 的滤波计算每个元素的邻域均值，随后计算出该元素的异常指数 σ ，如果大于 0.1 则用邻域均值进行替换，反之不变。该方法对应的算法流程图如下附图 4 所示：

该算法的伪代码可以分为三个关键部分：

- 主进程的数据分发
- 子进程的算法实现
- 主进程收集结果。

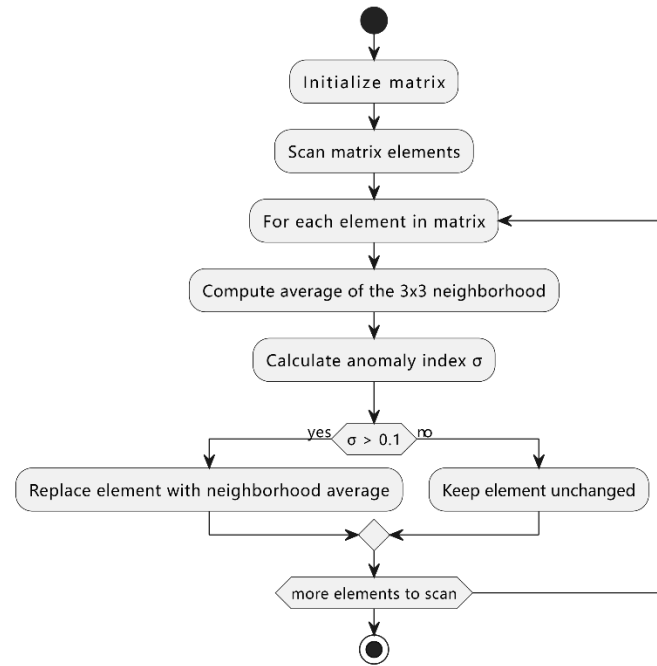
对于即将预处理的矩阵内容，每一行都会依赖上下两行，但边界（如第一行）行只能依赖下面一行，因此对于不同的行数，我们需要做区别对待。有了上述的认知后，就可以进行矩阵拆分并分发的工作了，对于中间部分的矩阵，我们在传递给子进程的时候可以顺带将前后两行一起传递；而对于第一行的矩阵，我们在传递的过程中要避免访问第-1行，这样会导致内存访问异常从而终止进程。我的做法是，对于要传递的一个 batch 的子矩阵，先检查子矩阵的第一行是否为大矩阵的第一行，如果是则不传递前面一行的内容，反之则传递；接着，通过 for 循环将要传递的主体部分一一传给子进程；最后，检查子矩阵的最后一行是否为大矩阵的最后一行，如果是则不传递后面一行的内容，反之则传递。实现该过程的伪代码如下 Algorithm 1 所示。

Algorithm 1 data distribution

```
1: for each rank > 0 do
2:   startRow = (rank - 1) * rowPerProcess.
3:   endRow = rank * rowsPerProcess;
4: end for
5: if startRow > 0 then
6:   endRow += extraRows
7: end if
8: if startRow > 0 then
9:   MPISend(mat[startRow - 1][0], ...).
10: end if
11: for each startRow <= i < endRow do
12:   MPISend(mat[i][0], ...)
13: end for
14: if dest_rank < size - 1 then
15:   MPISend(mat[endRow][0], ...)
16: end if
```

第二部分是子进程处理收到的矩阵，这一部分是算法的主体，也是主要占用算力资源的地方。该部分基本的算法思想是：在接收到主进程传入的子矩阵后，通过父矩阵的维度信息和自己进程的 rank 值推断出哪几行是自己需要进行预处理的，哪几行又是用来辅助预处理工作的。在此基础上，我们要遍历每一个目标元素，按照串行预处理（如附图 4）的办法计算邻域均值，判

断是否需要用邻域均值覆盖掉目标元素（即求取 σ 的值）。该部分的伪代码实现如 **Algorithm 2** 所示。



附图 4. 矩阵与处理流程图（串行时）

Algorithm 2 algorithm 2

Input:

mat: sub-matrix of A;
 cols: number of cloumn of matrix mat;
 rows: number of row of matrix mat;
 startLine, endLine: used to distinguish lines needed calculation from those who don't.

Output:

```

cols == B.col
1: enter into main process
2: for i: startLine → endLine do
3:   for j: 0 → cols do count = 0, sum = 0
4:     for m: i-1 → i+1 do
5:       for n: j-1→j+1 do
6:         if islegalindex then sum += mat[m][n] count++
7:         end if
8:       end for
9:     end for
10:    avg = sum/count;
11:    if abs(mat[i][j] - avg)/avg >= 0.1 then res[i][j] = avg;
12:    end if
13:  end for
14: end for
  
```

第三部分是父进程收集所有子进程的计算结果，并汇总打包成所需要的结果，对传入的原矩阵进行重新赋值，该部分的主要代码如下所示：

```

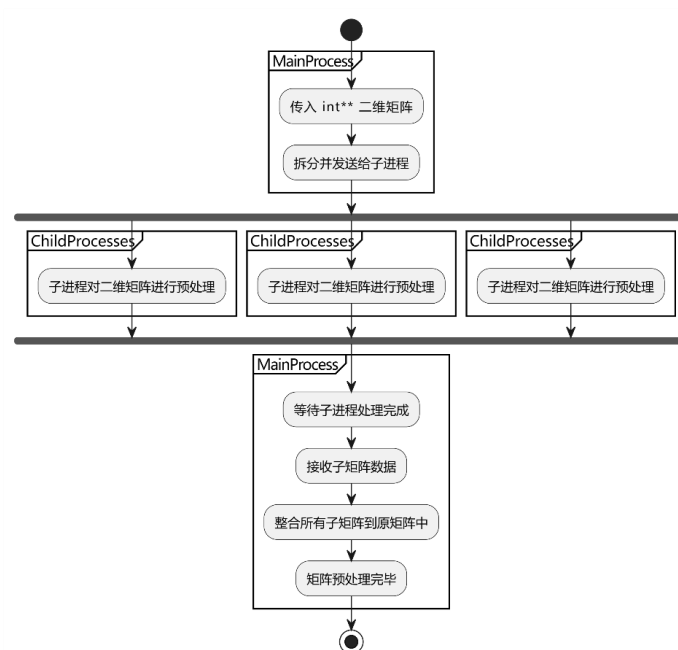
1. for (int dest_rank = 1; dest_rank < size; dest_rank++) {
2.   int dest_start_row = (dest_rank - 1) * rows_per_process;
  
```

```

3.     int dest_end_row = dest_rank * rows_per_process;
4.     if (dest_rank == size - 1) {
5.         dest_end_row += extra_rows;
6.     }
7.
8.     int subrow_size = dest_end_row - dest_start_row;
9.     int** subresultMatrix = new int* [subrow_size];
10.
11.    for (int i = 0; i < subrow_size; i++) {
12.        subresultMatrix[i] = new int[col_size];
13.        MPI_Recv(&subresultMatrix[i][0], col_size, MPI_INT, dest_rank
14.            , 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
15.    }
16.    // Copy the subresultMatrix to the main resultMatrix
17.    for (int i = dest_start_row; i < dest_end_row; i++) {
18.        for (int j = 0; j < col_size; j++) {
19.            resultMatrix[i][j] = subresultMatrix[i - dest_start_row][
20.                j];
21.        }
22.    }
23.    // Free memory
24.    for (int i = 0; i < subrow_size; i++) {
25.        delete[] subresultMatrix[i];
26.    }
27.    delete[] subresultMatrix;
28. }

```

整个算法的流程如下附图 5 所示



附图 5. 矩阵预处理算法流程图

1.3. 算法运行结果

第一步生成矩阵，为了方便截图和检查结果，我将生成 4*11 和 11*4 的 A,B 两个矩阵：

```
矩阵A行数：4
矩阵B列数：4

matrix A:
第0行：2 0 2 0 1 0 0 3 7 2 9
第1行：27 67 95 29 26 32 31 89 75 83 71
第2行：78 2 71 54 56 36 57 20 27 63 74
第3行：39 68 24 57 62 68 57 95 28 55 70

matrix B:
第0行：24 30 28 39
第1行：59 57 54 4
第2行：48 69 25 96
第3行：42 89 1 96
第4行：10 32 38 75
第5行：78 26 56 46
第6行：63 68 56 27
第7行：73 99 39 8
第8行：61 31 42 5
第9行：35 89 0 65
第10行：21 84 16 46
```

附图 6. 生成矩阵（首行为学号 20201003729）

接下来依次对两个矩阵进行矩阵预处理：

```
+-----+
|      |
|  输入数字  |
|      |
|  1. 重新生成稀疏矩阵A。 |
|  2. 重新生成稀疏矩阵B。 |
|  3. 对矩阵A进行预处理。 |
|  4. 对矩阵B进行预处理。 |
|  5. 通用矩阵乘法A * B。 |
|  6. 稀疏矩阵乘法A * B。 |
|  7. 退出。 |
|      |
+-----+

3
对矩阵A进行预处理以后：
第0行：31 38 38 30 17 18 31 40 50 49 52
第1行：27 34 28 38 26 25 31 27 36 41 46
第2行：40 58 49 54 45 48 57 57 63 63 74
第3行：49 42 50 57 54 53 57 37 52 55 70
```

附图 7. 对矩阵 A 进行预处理后的结果

```
+-----+
|      |
|  输入数字  |
|      |
|  1. 重新生成稀疏矩阵A。 |
|  2. 重新生成稀疏矩阵B。 |
|  3. 对矩阵A进行预处理。 |
|  4. 对矩阵B进行预处理。 |
|  5. 通用矩阵乘法A * B。 |
|  6. 稀疏矩阵乘法A * B。 |
|  7. 退出。 |
|      |
+-----+

4
对矩阵B进行预处理以后：
第0行：48 44 36 28
第1行：45 42 43 48
第2行：63 46 58 36
第3行：49 33 65 47
第4行：53 42 52 47
第5行：39 50 46 46
第6行：63 61 46 41
第7行：64 54 39 33
第8行：61 54 42 30
第9行：57 36 47 21
第10行：69 32 56 27
```

附图 8. 对矩阵 B 进行预处理后的结果

使用 4 个线程的性能验证结果如下**表格 1**所示：

表格 1.4 线程性能验证

矩阵维度	程序运行时间开销/ms
10*11	0
1000*11	1
10000*11	13
100000*11	120
200000*11	6821
300000*11	124534

使用 8 个线程的性能

表格 2.8 线程性能验证

矩阵维度	程序运行时间开销/ms
10*11	0
1000*11	1
10000*11	13
100000*11	458
200000*11	1842
300000*11	2945

当矩阵的长度为 1000000 时，预处理在调用 MPI_Send（）函数时显示 out of memory 内存溢出，原因是达到了系统分配给 visual studio 使用的内存上限。

```

矩阵A行数：1000000
矩阵B列数：2
3matrix A:
matrix B:

+-----+
|      |
|  输入数字  |
|      |
|  1. 重新生成稀疏矩阵A。  |
|  2. 重新生成稀疏矩阵B。  |
|  3. 对矩阵A进行预处理。  |
|  4. 对矩阵B进行预处理。  |
|  5. 通用矩阵乘法A * B。  |
|  6. 稀疏矩阵乘法A * B。  |
|  7. 退出。              |
|      |
+-----+

job aborted:
[ranks] message

[0] fatal error
Fatal error in MPI_Send: Other MPI error, error stack:
MPI_Send(buf=0x000001DAF05BE5E0, count=11, MPI_INT, dest=6, tag=0, MPI_COMM_WORLD) failed
Out of memory

[1-7] terminated

---- error analysis ----

[0] on ELWIN
mpi has detected a fatal error and aborted D:\Document\Codes\HPC_Final\x64\Debug\HPC_Final.exe

```

附图 9. 预处理过程内存溢出

1.4. 结果分析说明

结果分析分为两个部分，一个是功能性验证，一个是性能验证。功能性，经过计算检查附图 8，容易发现预处理的结果符合预期。

性能验证，通过对 4 个线程、8 个线程两种并行情况进行分：

1. 对比 4 个进程和 8 个进程的运行时间，我们发现在较小的矩阵维度下（10x11 和 1000x11），4 个进程的运行时间可能比 8 个进程稍快。这是因为在小规模矩阵上，进程间通信开销可能占据了较大的比例，而 8 个进程的通信开销更大。
2. 随着矩阵维度的增加，8 个进程逐渐显示出更好的性能优势。在大规模矩阵（10000x11，100000x11，1000000x11）上，8 个进程明显比 4 个进程快，这是因为 8 个进程可以更好地利用并行计算资源和通信。
3. 对于矩阵预处理操作，可能发现在较小的矩阵上，预处理时间相对于矩阵乘法操作来说较大，但随着矩阵维度的增加，预处理时间在整个程序运行时间中的比例逐渐减小。

2. 通用矩阵的乘法

2.1. 算法说明

要实现基于 mpi 的矩阵乘法，我们需要使用到两个主要的函数。MPI_Scatter 函数把矩阵分块并平均发送给每一个进程（包括 0 进程本身），MPI_Bcast 函数直接将数据广播给所有进程，一般由主进程进行广播，由于无论是哪个进程，要做乘法运算就必须存储整个矩阵 B，因此矩阵 B 就无需使用 MPI_Scatter 来切分；MPI_Gather 函数汇集各个进程的结果到主进程提供的接受对象中，该部分是回收子进程中计算的结果，并拼接到新的 int**数组中。

2.2. 算法流程图及伪代码

附图 10 描述了基于 MPI 的矩阵乘法过程。在主进程中，首先进行初始化，生成矩阵 A 和矩阵 B，并使用 MPI_Bcast 函数将矩阵 B 广播给所有进程。然后，使用 MPI_Scatter 函数将矩阵 A 分块发送给每个子进程。在各个子进程中，对接收到的矩阵分块和矩阵 B 进行乘法运算，并将计算结果存储在本地。接着，主进程使用 MPI_Gather 函数将各个子进程的计算结果汇集到主进程中，并拼接得到结果矩阵。最终，矩阵乘法完成，流程结束。通过并行计算和数据分发与回收，实现了高效的基于 MPI 的矩阵乘法运算。

另外需要注意的是，再将矩阵分块的过程中，由于一开始采用的是根据进程数量的均分策略，有时候行数/进程数除不净，会留下余数。余下的行我们会在主进程接受完所有子进程的计算结果以后由主进程继续计算。该算法的伪代码如下 Algorithm 3 所示。

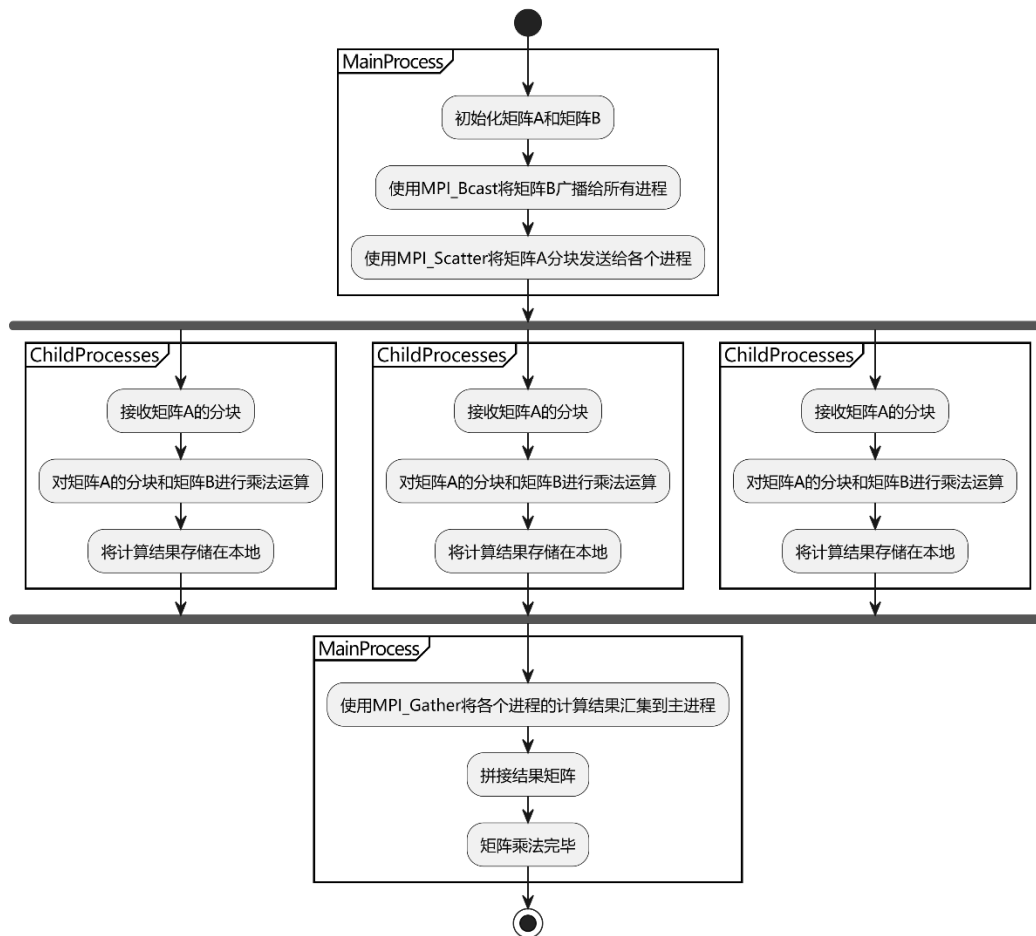
对于每个负责局部计算的进程，通过调用 calProduct 函数进行计算，该函数的伪代码如下 Algorithm 4 所示。

Algorithm 3 Normal Matrixes Product

```
1: MPIScatter(A, rowsPerProcess*11, ...)
2: MPIBcast(B, 11*colNumB,...)
3: calProduct(subA, B, subResult,...) → subResult
4: MPIGather(subResult, rowsPerProcess*colNumB,...)
5: remainedRowsStartId = rowsPerProcess;
6: if rank==0 && remainedRowsStartId < rowNumA then
7:   remainedRows = rowNumA - remainedRowsStartId;
8:   calProduct(A+remainRowsStartId*11, B,
9:   result+remainedRowsStartId*colNumB,...) → result
10: end if
```

Algorithm 4 product for children

```
1: for i = 0 → m do
2:   for j = 0 → n do temp = 0
3:     for k = 0 → p do temp+ = A[i × p + k] × B[k × n + j]
4:   end for
5:   matResult[i × n + j] = temp
6: end for
7: end for
```



附图 10. 通用矩阵乘法流程图

2.3. 算法运行结果

以下是对 5*11 和 11*5 的 A, B 两个矩阵进行运算的结果。

```

矩阵A行数: 5
矩阵B列数: 5

matrix A:
第0行: 2 0 2 0 1 0 0 3 7 2 9
第1行: 11 29 37 98 83 83 63 75 18 2 97
第2行: 23 13 87 31 92 98 37 14 58 97 54
第3行: 53 100 50 17 70 30 78 43 69 74 61
第4行: 10 52 46 70 85 81 28 5 55 86 34

matrix B:
第0行: 69 62 91 23 9
第1行: 91 3 7 32 13
第2行: 2 25 29 36 76
第3行: 73 30 81 19 2
第4行: 0 25 96 59 40
第5行: 74 11 35 24 64
第6行: 6 47 29 49 71
第7行: 3 96 94 21 95
第8行: 9 9 2 58 95
第9行: 90 55 10 44 85
第10行: 42 97 20 21 72
  
```

附图 11. 生成的随机矩阵（首行为学号 20201003729）

```
+-----+
|      |
|  输入数字  |
|      |
|  1. 重新生成稀疏矩阵A。  |
|  2. 重新生成稀疏矩阵B。  |
|  3. 对矩阵A进行预处理。  |
|  4. 对矩阵B进行预处理。  |
|  5. 通用矩阵乘法A * B。  |
|  6. 稀疏矩阵乘法A * B。  |
|  7. 退出。  |
|      |
+-----+

5
(通用) 矩阵点积结果:
第0行: 772 1533 832 923 1978
第1行: 21787 27464 31961 19095 32578
第2行: 24243 22126 24035 23319 38602
第3行: 26758 25828 24522 24656 37191
第4行: 26464 17361 22205 21004 30432
```

附图 12. 通用矩阵乘法运行结果

在 4 进程的前提下进行通用矩阵乘法

表格 3.4 线程矩阵乘法的结果

矩阵 A 维度	矩阵 B 维度	程序运行时间开销/ms
10*11	11*10	5
1000*11	11*1000	15
10000*11	11*10000	916
100000*11	11*100000	4879

在 8 进程的前提下进行通用矩阵乘法

表格 4.8 线程矩阵乘法的结果

矩阵 A 维度	矩阵 B 维度	程序运行时间开销/ms
10*11	11*10	12
1000*11	11*1000	18
10000*11	11*10000	876
100000*11	11*100000	2443

同样的在 200000*11 的矩阵乘法计算时出现了内存溢出的情况:

```
矩阵A行数: 200000
矩阵B列数: 200000
matrix A:
matrix B:

+-----+
|      |
|  输入数字  |
|      |
|  1. 重新生成稀疏矩阵A。  |
|  2. 重新生成稀疏矩阵B。  |
|  3. 对矩阵A进行预处理。  |
|  4. 对矩阵B进行预处理。  |
|  5. 通用矩阵乘法A * B。  |
|  6. 稀疏矩阵乘法A * B。  |
|  7. 退出。  |
|      |
+-----+

5

job aborted:
[ranks] message

[0] terminated

[1] process exited without calling finalize

[2-3] terminated

---- error analysis ----

[1] on ELWIN
D:\Document\Codes\HPC_Final\x64\Debug\HPC_Final.exe ended prematurely and may have crashed. exit code 0xc0000005

---- error analysis ----
```

附图 13. 通用矩阵乘法内存溢出

2.4. 结果分析说明

从上述的结果可以看见，由于 VS 内存大小受限制，我们可以进行测试的矩阵大小并不能太大，可以确定性能瓶颈就在内存大小上。在内存受限的前提下，我们对矩阵的每次测试结果每次都有较大的波动，为了去到较为可信的值，每一行测试用例我都进行了至少 20 次的计算，最终取 20 次结果的平均值。从平均值来看，可以分析出在矩阵较小的时候，8 线程并没有什么优势，反而由于要分配更多数据而导致计算速度更慢，但在矩阵逐渐变大后，8 线程的优势在运算时间上就逐渐体现。

3. 稀疏矩阵的乘法

3.1. 算法说明

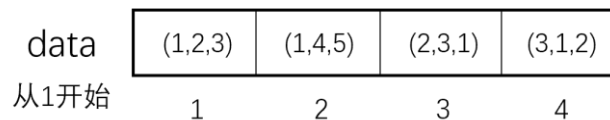
要使用稀疏矩阵算法，就需要先调研稀疏矩阵的表示方式。经过调查，我找到了 3 中存储方式，分别为三元组顺序表存储、行逻辑链接的顺序表、十字链表。本次课程实践我选择使用行逻辑链接顺序表结构来存储稀疏矩阵。

按照行逻辑链接顺序表来存储稀疏矩阵的过程也可以成为矩阵压缩，对于 0 的数量较多的中大型矩阵，我们可以通过稀疏矩阵进行存储，从而降低一个矩阵在计算机中的空间开销。接下来我将介绍一下稀疏矩阵算法的基本实现原理。

行逻辑链接的顺序表和三元组顺序表的实现过程类似，它们存储矩阵的过程完全相同，都是将矩阵中非 0 元素的三元组（行标、列标和元素值）存储在一维数组中。但为了提高提取查找指定行非 0 元素的效率，前者在存储矩阵时比后者多使用了一个数组，专门记录矩阵中每行第一个非 0 元素在一维数组中的位置。

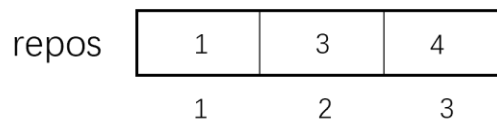
```
0 3 0 5
0 0 1 0
2 0 0 0
```

上式是一个稀疏矩阵，当使用行逻辑链接的顺序表对其进行压缩存储时，需要做以下两个工作：1. 将矩阵中的非 0 元素采用三元组的形式存储到一维数组 data 中，如附图 14 所示（和三元组顺序表一样）2. 使用数组 rpos 记录矩阵中每行第一个非 0 元素在一维数组中的存储位置。如图 3 所示。



附图 14

通过以上两步操作，即实现了使用行逻辑链接的顺序表存储稀疏矩阵。



附图 15

行逻辑链接顺序表的结构体定义为以下结构：

```
1. /* 三元组 */
2. struct Triple {
3.     int i, j;    // 表示元素的行列值
4.     int e;      // 表示元素的值
5. };
6.
7. /* 行逻辑链接顺序表 */
```

```

8. struct RLSMatrix {
9.     Triple data[1000];           // 非零元素的三元组表
10.    int rowOffset[1000];         // 各行第一个非零元素在 data 中的索引表
11.    int mu, nu, tu;              // m -> 行数, n -> 列数, t -> 非零元素的个数
12. };

```

3.2. 算法流程图及伪代码

该部分算法流程分为 3 个部分，首先我需要完成生成稀疏矩阵的算法，该算法需要能够根据调用者提供的行、列数以及稀疏度来生成随机的稀疏矩阵，按照要求首行为我的学号 20201003729。具体实现的伪代码如 **Algorithm 5** 所示：

Algorithm 5 Random Sparse Matrix Generation

```

1: function GENERATESPARSEMATRIX(rows, cols, sparsity)
2:   matrix  $\leftarrow$  empty 2D array of size rows  $\times$  cols
3:   for i = 1 to rows do
4:     for j = 1 to cols do
5:       randomNumber  $\leftarrow$  random number between 0 and 1
6:       if randomNumber  $\leq$  sparsity then
7:         matrix[i][j]  $\leftarrow$  random value
8:       end if
9:     end for
10:  end for
11:  return matrix
12: end function

```

在有了稀疏矩阵以后，我需要稀疏矩阵转换为行逻辑链接顺序表形式，也就是压缩矩阵，采用占用内存更小的稀疏矩阵存储方式。由于其中涉及的下标变量过多不便于展示，我将只展示核心部分。

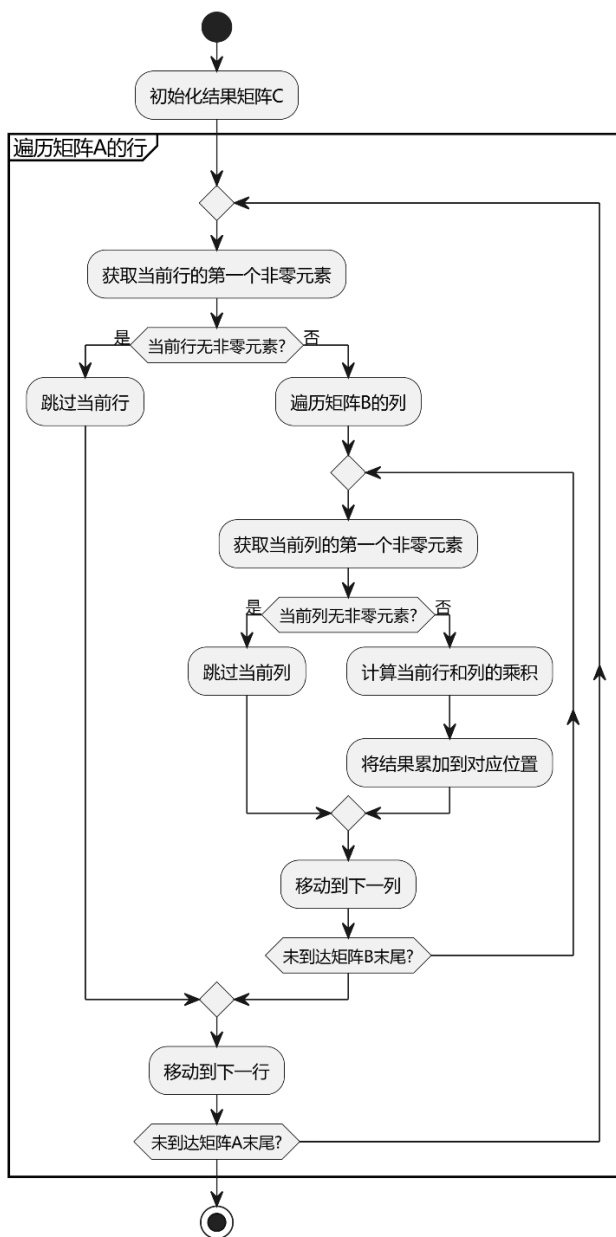
```

1. for (int i = 0; i < row; i++) {
2.     bool flag = true; // 如果找到了该行第一个元素，则设置为 false
3.     for (int j = 0; j < col; j++) {
4.         if (mat[i][j] != 0) { // 记录非零元素
5.             tu++;
6.             data[tu].e = mat[i][j];
7.             data[tu].i = i + 1;
8.             data[tu].j = j + 1;
9.
10.            if (flag) { // 记录第一个非零元素在 data 数组里的位置
11.                flag = false;
12.                rowOffset[i + 1] = tu;
13.            }
14.        }
15.    }
16.    if (flag == true) { // 遍历整行没有找到非零元素，则将 offset 值设置为 tu+1
17.        rowOffset[i + 1] = -1;
18.    }

```

19. }

最后就是实现稀疏矩阵在行逻辑链接顺序表前提下的矩阵乘法，该部分同样十分复杂，我将做粗略介绍。对矩阵的乘积进行深度剖析，矩阵 A 和矩阵 B 相乘的流程图如下附图 16 所示：

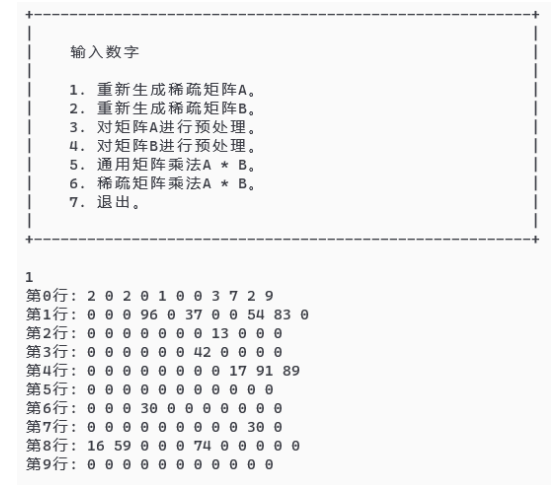


附图 16 稀疏矩阵乘法流程图

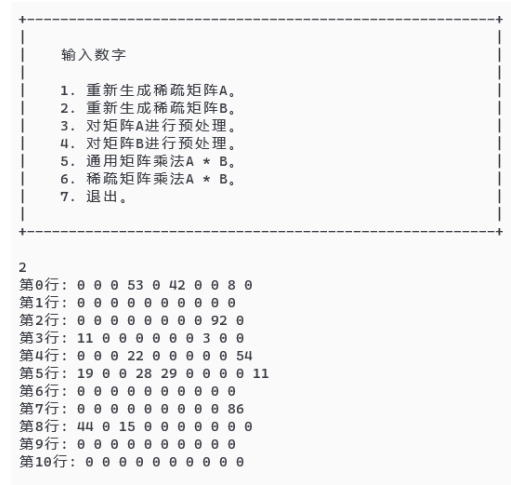
在知道了稀疏矩阵的乘法应该怎么进行后，我们可以定义一个 `product(Mat subA, Mat B, Mat C)` 方法，对 AB 矩阵进行乘法运算。为了用到并行计算，我们可以用主进程将矩阵 A 切分为多个部分，由子进程对子矩阵 subA 进行计算，最后由主进程收集各个进程传递的结果。同样的，由于矩阵 B 在 subA 矩阵每一行都要完整地使用，因此矩阵 B 可以直接通过 `MPI_Bcast()` 广播给所有子进程。

3.3. 算法运行结果

生成稀疏矩阵（10*11 和 11*10，稀疏度 0.3）：

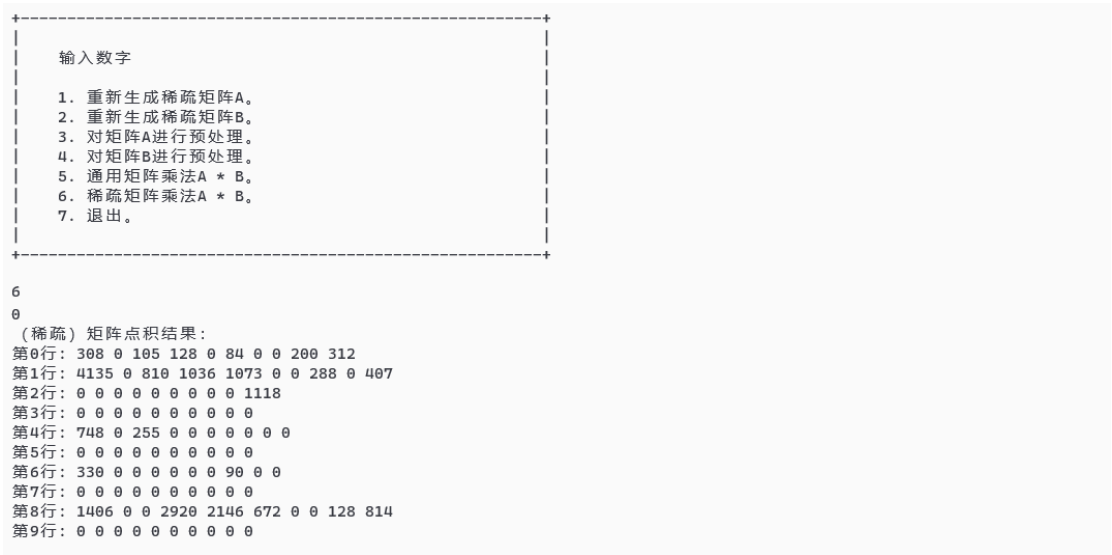


附图 17. 稀疏矩阵 A



附图 18. 稀疏矩阵 B

对稀疏矩阵 A，B 做乘法运算：



附图 19. 稀疏矩阵乘法的功能性验证

在 4 进程的前提下进行稀疏矩阵乘法

表格 5. 4 线程稀疏矩阵乘法的结果

矩阵 A 维度	矩阵 B 维度	程序运行时间开销/ms
10*11	11*10	3
1000*11	11*1000	9
10000*11	11*10000	325
100000*11	11*100000	2543

在 8 进程的前提下进行稀疏矩阵乘法

表格 6. 8 线程稀疏矩阵乘法的结果

矩阵 A 维度	矩阵 B 维度	程序运行时间开销/ms
10*11	11*10	5
1000*11	11*1000	16
10000*11	11*10000	155
100000*11	11*100000	2012

3.4. 结果分析说明

根据稀疏矩阵乘法（见表格 5.4 线程稀疏矩阵乘法的结果表格 6.8 线程稀疏矩阵乘法的结果）的开销和通用矩阵乘法（见表格 2.8 线程性能验证表格 3.4 线程矩阵乘法的结果）的程序运行时间，我们可以发现稀疏矩阵在使用行逻辑链接顺序表的时候会有更好的效果。

由于通用矩阵乘法的算法是僵硬的遍历所有计算可能，所以无论是稀疏矩阵还是通用矩阵在使用这套算法的时候，只要规模相同，就可以认为所需要的时间开销相同。因此，不需要重新使用新生成的稀疏矩阵再次测算通用矩阵的算法时间开销。

横向对比之外，我们也可以纵向对比，分析 4 线程和 8 线程下分别对稀疏矩阵算法的影响。从表格中我们可以发现基本规律和通用矩阵相同，即随着矩阵的规模扩大，线程带来的时间优势也在提升。但同时也出现了提升不明显的情况，出现该情况的主要原因是稀疏矩阵的稀疏度被我设置为了 0.3，也就是说矩阵中的 0 元素相比于非 0 元素非常多，即使我们大幅度的扩大矩阵规模（到了 10 的 5 次方），但实际上在对矩阵进行压缩以后，需要存储的内容并没有很多。考虑到线程的优势和矩阵的规模有正相关性，同时压缩后的稀疏矩阵规模受限，因此线程的优势在这里并不明显。该应用场景的局限也解答了我的一个疑问：为什么网上关于稀疏矩阵的 mpi 代码这么少[^]。

附件

作业一

```
1. #pragma warning(disable : 4996)
2. #include <iostream>
3. #include <mpi.h>
4. #include <cstdlib>
5.
6. #define MAX_SIZE 10000
7.
8. void Merge(int* a, int* b, int start, int middle, int end)
9. {
10.     int na1, na2, nb, i;
11.     na1 = start;
12.     nb = start;
13.     na2 = middle + 1;
14.
15.     while ((na1 <= middle) && (na2 <= end)) {
16.         if (a[na1] <= a[na2]) {
17.             b[nb++] = a[na1++];
18.         }
19.         else {
20.             b[nb++] = a[na2++];
21.         }
22.     }
23.
24.     if (na1 <= middle) {
25.         for (i = na1; i <= middle; i++) {
26.             b[nb++] = a[i];
27.         }
```



```

28.     }
29.     if (na2 <= end) {
30.         for (i = na2; i <= end; i++) {
31.             b[nb++] = a[i];
32.         }
33.     }
34.
35.     for (i = start; i <= end; i++) {
36.         a[i] = b[i];
37.     }
38. }
39.
40. void Merge_Sort(int* a, int* b, int start, int end)
41. {
42.     int middle;
43.
44.     if (start < end) {
45.         middle = (start + end) / 2;
46.         Merge_Sort(a, b, start, middle);
47.         Merge_Sort(a, b, middle + 1, end);
48.         Merge(a, b, start, middle, end);
49.     }
50. }
51.
52. int main(int argc, char* argv[])
53. {
54.     int arr[MAX_SIZE];
55.     int comm_sz;
56.     int my_rank;
57.     int length = 0;
58.     int size;
59.     int* sub;
60.     int* tmp;
61.     int* result = nullptr;
62.     int* tmp_for_last;
63.     int i;
64.     FILE* fpread;
65.
66.     // 读文件
67.     for (i = 0; i < MAX_SIZE; i++) {
68.         arr[i] = -1;
69.     }
70.
71.     fpread = fopen("input.txt", "r");

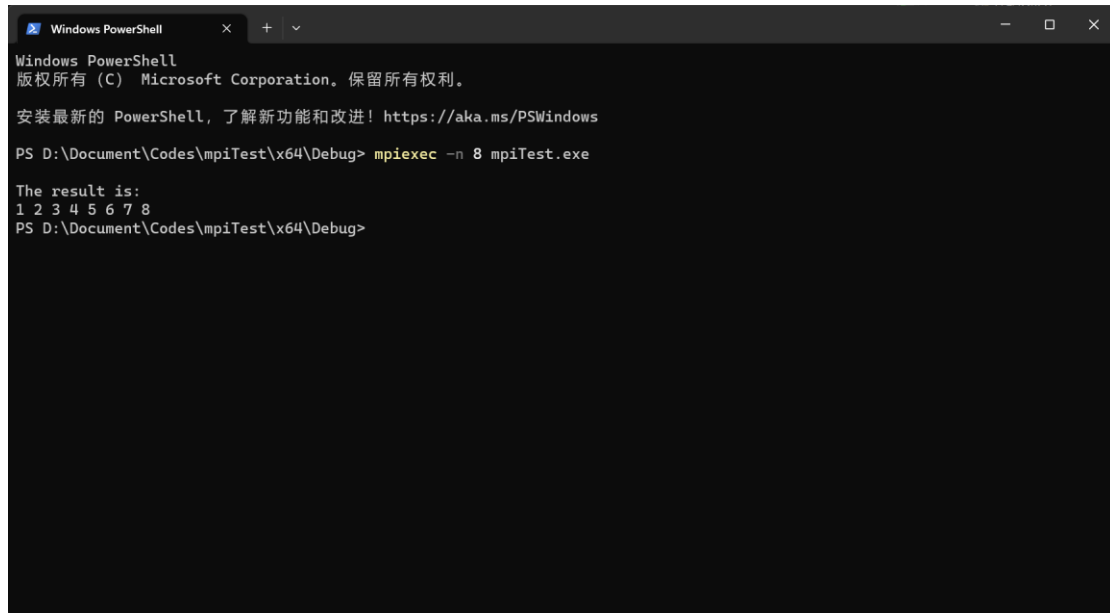
```

```

72.     if (fpread == nullptr) {
73.         std::cout << "Read error!" << std::endl;
74.         return 0;
75.     }
76.     for (i = 0; !feof(fpread); i++) {
77.         fscanf(fpread, "%d ", &arr[i]);
78.         length++;
79.     }
80.
81.     MPI_Init(NULL, NULL);
82.     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
83.     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
84.
85.     size = length / comm_sz; // 计算每个进程所需数组的大小
86.     sub = new int[size]; // 为进程所用数组开辟空间
87.
88.     MPI_Scatter(arr, size, MPI_INT, sub, size, MPI_INT, 0,
        MPI_COMM_WORLD); // 将数组元素分发给每个进程
89.
90.     tmp = new int[size]; // 开辟归并排序时需要的临时数组
91.     Merge_Sort(sub, tmp, 0, size - 1);
92.
93.     if (my_rank == 0) {
94.         result = new int[length]; // 开辟结果所需的数组空间
            (与输入数组一样大)
95.     }
96.
97.     MPI_Gather(sub, size, MPI_INT, result, size, MPI_INT, 0
        , MPI_COMM_WORLD); // 将各进程排序的数组收集到0号进程
98.
99.     if (my_rank == 0) {
100.         tmp_for_last = new int[length];
101.         Merge_Sort(result, tmp_for_last, 0, length - 1);
102.
103.         // 打印结果
104.         std::cout << "\nThe result is: " << std::endl;
105.         for (i = 0; i < length; i++) {
106.             std::cout << result[i] << " ";
107.         }
108.         std::cout << std::endl;
109.
110.         delete[] result;
111.         delete[] tmp_for_last;
112.     }

```

```
113.     delete[] sub;
114.     delete[] tmp;
115.
116.     MPI_Finalize();
117.     return 0;
118. }
```



```
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

安装最新的 PowerShell，了解新功能和改进！https://aka.ms/PSWindows

PS D:\Document\Codes\mpiTest\x64\Debug> mpiexec -n 8 mpiTest.exe

The result is:
1 2 3 4 5 6 7 8
PS D:\Document\Codes\mpiTest\x64\Debug>
```

附图 20 作业一运行结果

作业二

```
1. #include <iostream>
2. #include <cstdlib>
3. #include <ctime>
4. #include <algorithm>
5. #include <cmath>
6. #include <mpi.h>
7.
8. using namespace std;
9.
10. struct Pair {
11.     int left;
12.     int right;
13. };
14.
15. const int MAX_PROCESS = 128;
16. const int NUM = 100;
17. const int MAX = 1000000;
18. const int MIN = 0;
19.
20. int arr[NUM];
21. int temp[NUM];
```

```

22.
23. Pair pairs[MAX_PROCESS];
24.
25. int counter = -1;
26.
27. void swap(int A[], int i, int j) {
28.     int temp = A[i];
29.     A[i] = A[j];
30.     A[j] = temp;
31. }
32.
33. int findpivot(int i, int j) {
34.     return (i + j) / 2;
35. }
36.
37. int partition(int A[], int l, int r, int pivot) {
38.     do {
39.         while (A[++l] < pivot);
40.         while ((r != 0 && (A[--r] > pivot)));
41.         swap(A, l, r);
42.     } while (l < r);
43.     swap(A, l, r);
44.     return l;
45. }
46.
47. void quicksort(int A[], int i, int j, int currentdepth, int
    targetdepth) {
48.     if (currentdepth == targetdepth) {
49.         int rank = ++counter;
50.         pairs[rank].left = i;
51.         pairs[rank].right = j;
52.         cout << pairs[rank].left << " and " << pairs[rank].right
            << " : rank " << rank << endl;
53.         return;
54.     }
55.     if (j <= i) return;
56.     int pivotindex = findpivot(i, j);
57.     swap(A, pivotindex, j);
58.     int k = partition(A, i - 1, j, A[j]);
59.     swap(A, k, j);
60.     quicksort(A, i, k - 1, currentdepth + 1, targetdepth);
61.     quicksort(A, k + 1, j, currentdepth + 1, targetdepth);
62. }
63.

```

```

64. void quicksort(int A[], int i, int j) {
65.     if (j <= i) return;
66.     int pivotindex = findpivot(i, j);
67.     swap(A, pivotindex, j);
68.     int k = partition(A, i - 1, j, A[j]);
69.     swap(A, k, j);
70.     quicksort(A, i, k - 1);
71.     quicksort(A, k + 1, j);
72. }
73.
74. int main(int argc, char* argv[]) {
75.     MPI_Init(&argc, &argv);
76.     int RANK, SIZE, targetdepth, left, right, REAL_SIZE;
77.
78.     MPI_Comm_rank(MPI_COMM_WORLD, &RANK);
79.     MPI_Comm_size(MPI_COMM_WORLD, &SIZE);
80.     REAL_SIZE = SIZE;
81.     if (RANK == 0) {
82.         cout << "Quick sort start..." << endl;
83.         cout << "Generate random data... ";
84.
85.         memset(arr, 0, NUM * sizeof(arr[0]));
86.         srand(time(NULL));
87.         for (int i = 0; i < NUM; i++) {
88.             arr[i] = MIN + rand() % (MAX - MIN);
89.         }
90.         cout << "Done." << endl;
91.         targetdepth = log2(SIZE);
92.         cout << "Rank: " << RANK << endl;
93.         cout << "Sorting... ";
94.         quicksort(arr, 0, NUM - 1, 0, targetdepth);
95.         REAL_SIZE = counter + 1;
96.         for (int i = 1; i < SIZE; i++) {
97.             int left = pairs[i].left;
98.             int right = pairs[i].right;
99.             MPI_Send(&REAL_SIZE, 1, MPI_INT, i, 99, MPI_COMM_WORLD);
100.             MPI_Send(&left, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
101.             MPI_Send(&right, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
102.             MPI_Send(&arr, NUM, MPI_INT, i, 2, MPI_COMM_WORLD);
103.         }
104.
105.         left = pairs[0].left;
106.         right = pairs[0].right;
107.         quicksort(arr, left, right);

```

```

108.     cout << "Process " << RANK << " done." << endl;
109. }
110.
111. for (int process = 1; process < REAL_SIZE; process++) {
112.     if (RANK == process) {
113.         MPI_Status status;
114.         MPI_Recv(&REAL_SIZE, 1, MPI_INT, 0, 99, MPI_COMM_WORLD
, &status);
115.         MPI_Recv(&left, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &sta
tus);
116.         MPI_Recv(&right, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &st
atus);
117.         MPI_Recv(&arr, NUM, MPI_INT, 0, 2, MPI_COMM_WORLD, &st
atus);
118.         if (process < REAL_SIZE) {
119.             quicksort(arr, left, right);
120.             MPI_Send(&arr, NUM, MPI_INT, 0, 0, MPI_COMM_WORLD);
121.             cout << "Process " << RANK << " done." << endl;
122.         }
123.     }
124. }
125.
126. if (RANK == 0) {
127.     for (int i = 1; i < REAL_SIZE; i++) {
128.         //cout << "Master is ready to receive data from proces
s " << i << endl;
129.         MPI_Status status;
130.         MPI_Recv(&temp, NUM, MPI_INT, i, 0, MPI_COMM_WORLD, &s
tatus);
131.         for (int j = pairs[i].left; j <= pairs[i].right; j++)
        {
132.             arr[j] = temp[j];
133.         }
134.         //cout << "Master has combined data from process " <<
i << endl;
135.     }
136.     cout << "Done." << endl;
137.     cout << "Result:" << endl;
138.     int counter = 1;
139.     int row = 20;
140.
141.     for (int i = 0; i < NUM; i++, counter++) {
142.         cout << arr[i] << " ";
143.         if (arr[i] < arr[max(i - 1, 0)]) {

```

```

144.         cerr << "Invalid! " << arr[i] << " > " << arr[max(i -
           1, 0)] << " i is " << i << endl;
145.     }
146.     if (counter % row == 0) cout << endl;
147. }
148. }
149. MPI_Finalize();
150.
151. }

```

```

Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

安装最新的 PowerShell，了解新功能和改进！https://aka.ms/PSWindows

PS D:\Document\Codes\mpiTest\x64\Debug> mpiexec -n 8 QuickSort.exe
Quick sort start...
Generate random data... Done.
Rank: 0
Sorting... 0 and 511 : rank 0
513 and 1060 : rank 1
1062 and 1401 : rank 2
1403 and 1614 : rank 3
1616 and 6997 : rank 4
6999 and 7741 : rank 5
7743 and 7877 : rank 6
7879 and 7999 : rank 7
Process 1 done.
Process 2 done.
Process 3 done.
Process 5 done.
Process 0 done.
Process 6 done.
Done.
Result:
Process 4 done.
0 6 11 13 18 19 23 30 47 60 62 89 97 98 101 101 107 113 116 122
125 132 133 138 145 147 160 167 169 175 175 182 202 206 210 210 214 219 219 224
Process 7 done.
234 239 240 246 264 273 274 287 289 292 295 299 300 305 306 307 314 321 324 324
329 332 334 339 340 350 355 359 360 362 366 366 374 380 384 384 394 395 399 400
401 402 410 414 424 427 428 437 443 444 445 447 459 460 460 464 473 475 487 490
490 490 490 491 499 499 516 516 520 527 530 537 537 538 541 545 552 553 558 561
561 568 573 575 575 589 596 599 605 611 623 630 631 632 632 645 647 658 666 670
670 680 683 683 684 686 687 689 693 694 694 695 703 703 706 711 731 734 754 755
758 759 771 774 774 775 781 788 790 804 805 810 813 820 824 831 835 835 838 852
855 858 861 862 862 863 863 866 866 866 877 888 892 898 899 907 908 908 912 913
916 917 917 917 918 921 923 928 934 936 937 940 941 956 961 964 972 972 977 979
983 986 986 990 994 997 997 999 1004 1006 1008 1009 1016 1022 1026 1030 1035 1043 1045 1048
1049 1051 1053 1055 1055 1056 1073 1073 1077 1083 1094 1095 1097 1100 1107 1107 1111 1115 1126 1141
1158 1168 1168 1173 1180 1181 1183 1191 1208 1209 1211 1214 1219 1223 1229 1231 1245 1247 1247 1251
1252 1252 1254 1262 1262 1270 1275 1278 1281 1281 1291 1293 1297 1297 1299 1305 1312 1314 1316 1318
1320 1320 1324 1327 1331 1344 1346 1346 1363 1373 1376 1390 1391 1395 1397 1400 1404 1406 1408 1415
1419 1422 1425 1426 1432 1434 1434 1439 1445 1446 1455 1457 1465 1469 1469 1471 1474 1476 1477 1479
1488 1488 1489 1490 1498 1498 1504 1505 1507 1525 1528 1532 1539 1540 1541 1543 1544 1544 1548 1550
1555 1557 1571 1576 1576 1577 1600 1602 1613 1617 1621 1621 1634 1640 1644 1644 1647 1653 1654 1655
1656 1661 1664 1679 1689 1696 1701 1705 1711 1719 1727 1729 1735 1738 1748 1753 1756 1762 1771 1773
1773 1775 1781 1783 1785 1785 1786 1788 1788 1791 1793 1795 1801 1809 1810 1814 1816 1825 1831 1835
1837 1837 1839 1840 1843 1847 1848 1854 1859 1874 1876 1876 1879 1881 1881 1882 1886 1891 1894 1897

```

附图 21 作业二运行结果

课程实践

1. /*
2. 作者：刘星雨
3. 学号：20201003729
4. 文件名："myMat.h"
5. */
- 6.
7. #pragma once

```
8. #include <iostream>
9. #include <vector>
10. #include <random>
11. #include <mpi.h>
12. #include <cstdlib>
13. using namespace std;
14.
15. class MyMat {
16. private:
17.     /*
18.         随机矩阵生成器
19.         1. 输入行列数
20.     */
21.     int** getMat(int row, int col);
22.
23.     /*
24.         获取稀疏矩阵
25.         1. 输入行数
26.         2. 输入列数
27.         3. 输入稀疏度
28.     */
29.     int** getSparseMatrix(int row_size, int col_size, double sparsity
30. );
31.     void matMultiplyWithSingleThread(int* A, int* B, int* matResult,
32. int m, int p, int n);
33. public:
34.     /*
35.         打印矩阵
36.     */
37.     void printMatrix(int** matrix, int row_size, int col_size);
38.
39.     /*
40.         矩阵 A 生成器
41.         1. 输入行数（列数必须为 11），输出随机矩阵
42.         2. 矩阵的第一行为学号
43.     */
44.     int** getMatA(int row);
45.
46.     /*
47.         矩阵 B 生成器
48.         1. 输入列数（行数必须为 11，对应矩阵 A），输出随机矩阵
49.     */
```



```

50.     int** getMatB(int col);
51.
52.     /*
53.         稀疏矩阵 A 生成器
54.         1. 输入行数（列数必须为 11）
55.         2. 输入稀疏度
56.     */
57.     int** getSparseMatA(int row_size, double sparsity);
58.
59.     /*
60.         稀疏矩阵 B 生成器
61.         1. 输入列数（行数必须为 11）
62.         2. 输入稀疏度
63.     */
64.     int** getSparseMatB(int col_size, double sparsity);
65.
66.     /*
67.         对矩阵进行预处理
68.         1. 对于  $[x_0 - \text{average}(x_1, \dots, x_8)] / \text{average}(x_1, \dots, x_8) < 10\%$ , 成立则
           不改变  $M(x_0, y_0)$  的值, 反之则用均值覆盖
69.     */
70.     void meanFilter(int** mat, int row_size, int col_size);
71.
72.     /*
73.         通用矩阵乘法
74.         1. 输入矩阵 A
75.         2. 输入矩阵 B
76.     */
77.     int** product(int** matA, int rowNumA, int** matB, int colNumB);
78.
79.     /*
80.         稀疏矩阵乘法
81.         1. 输入矩阵 A
82.         2. 输入矩阵 B
83.     */
84.     int** sparseProduct(int** matA, int rowNumA, int** matB, int colNumB);
85.
86. };

```



```

1.  /*
2.     作者: 刘星雨
3.     学号: 20201003729
4.     文件名: "mySparseMat.h"

```

```

5.  */
6.
7. #pragma once
8. #include "sparseMatbody.h"
9. #include <iostream>
10. #include <mpi.h>
11. using namespace std;
12.
13. class MySparseMat {
14. private:
15.     /*
16.         获取稀疏矩阵
17.         1. 输入行数
18.         2. 输入列数
19.         3. 输入稀疏度
20.     */
21.     int** getSparseMatrix(int row_size, int col_size, double sparsity
22. );
23.     // 封装 MPI 数据类型的创建
24.     MPI_Datatype createMPI_Triple() {
25.         MPI_Datatype MPI_TRIPLE;
26.         int blocklengths[3] = { 1, 1, 1 };
27.         MPI_Aint displacements[3];
28.         MPI_Datatype types[3] = { MPI_INT, MPI_INT, MPI_INT };
29.         MPI_Aint baseaddr, addr1, addr2;
30.         MPI_Get_address(&((Triple*)0)->i, &baseaddr);
31.         MPI_Get_address(&((Triple*)0)->j, &addr1);
32.         MPI_Get_address(&((Triple*)0)->e, &addr2);
33.         displacements[0] = 0;
34.         displacements[1] = addr1 - baseaddr;
35.         displacements[2] = addr2 - baseaddr;
36.         MPI_Type_create_struct(3, blocklengths, displacements, types,
37.             &MPI_TRIPLE);
38.         MPI_Type_commit(&MPI_TRIPLE);
39.         return MPI_TRIPLE;
40.     }
41. public:
42.     /*
43.         打印矩阵
44.     */
45.     void display(RLSMatrix M);
46.

```

```

47.      /*
48.          把二维矩阵压缩行逻辑链接顺序表形式
49.      */
50.      RLSMatrix compressSparseMat(int** mat, int row, int col);
51.
52.      /*
53.          串行计算稀疏矩阵的点积
54.      */
55.      RLSMatrix product(RLSMatrix A, RLSMatrix B, RLSMatrix C);
56.
57.      /*
58.          并行计算稀疏矩阵的点积
59.      */
60.      RLSMatrix parallelProduct(RLSMatrix A, RLSMatrix B, RLSMatrix C);
61. };

```

```

1.  /*
2.      作者: 刘星雨
3.      学号: 20201003729
4.      文件名: "sparseMatBody.h"
5.  */
6.
7.  #pragma once
8.
9.  /* 三元组 */
10. struct Triple {
11.     int i, j;    // 表示元素的行列值
12.     int e;       // 表示元素的值
13. };
14.
15. /* 行逻辑链接顺序表 */
16. struct RLSMatrix {
17.     Triple data[1000];    // 非零元素的三元组表
18.     int rowOffset[1000]; // 各行第一个非零元素在 data 中的索引表
19.     int mu, nu, tu;       // m -> 行数, n -> 列数, t -> 非零元素的个
        数
20. };

```

```

1.  /*
2.      作者: 刘星雨
3.      学号: 20201003729
4.      文件名: "myMat.cpp"
5.  */
6.

```

```

7. #include "myMat.h"
8.
9. void MyMat::meanFilter(int** mat, int row_size, int col_size)
10. {
11.     int rank, size;
12.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13.     MPI_Comm_size(MPI_COMM_WORLD, &size);
14.
15.     // 串行则跳出
16.     if (size < 2) return;
17.
18.     int rows_per_process = row_size / (size - 1);
19.     int extra_rows = row_size % (size - 1);
20.
21.     if (rank == 0) { // 主进程分发数据
22.         for (int dest_rank = 1; dest_rank < size; dest_rank++) {
23.             int dest_start_row = (dest_rank - 1) * rows_per_process;
24.
25.             int dest_end_row = dest_rank * rows_per_process;
26.
27.             if (dest_rank == size - 1) {
28.                 dest_end_row += extra_rows;
29.             }
30.
31.             // 把矩阵数据发送给目标进程
32.             if (dest_start_row > 0) { // 多发上面一行
33.                 MPI_Send(&mat[dest_start_row - 1][0], col_size, MPI_INT, dest_rank, 0, MPI_COMM_WORLD);
34.             }
35.             for (int i = dest_start_row; i < dest_end_row; i++) {
36.                 MPI_Send(&mat[i][0], col_size, MPI_INT, dest_rank, 0, MPI_COMM_WORLD);
37.             }
38.             if (dest_rank < size - 1) { // 多发下面一行
39.                 MPI_Send(&mat[dest_end_row][0], col_size, MPI_INT, dest_rank, 0, MPI_COMM_WORLD);
40.             }
41.         }
42.     } else { // 子进程获取并处理数据
43.         int start_row = (rank - 1) * rows_per_process;
44.         int end_row = rank * rows_per_process;
45.         if (rank == size - 1) {
46.             end_row += extra_rows;

```

```

47.     }
48.     int sub_row_size = rows_per_process;
49.     if (rank == size - 1) {
50.         sub_row_size = end_row - start_row;
51.     }
52.     if (start_row > 0) sub_row_size++;
53.     if (end_row < row_size) sub_row_size++;
54.
55.     mat = new int* [sub_row_size];
56.     int** resultMat = new int* [sub_row_size];
57.
58.     for (int i = 0; i < sub_row_size; i++) {
59.         mat[i] = new int[col_size];
60.         resultMat[i] = new int[col_size];
61.         MPI_Recv(&mat[i][0], col_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
62.         for (int j = 0; j < col_size; j++) {
63.             resultMat[i][j] = mat[i][j];
64.         }
65.     }
66.
67.     int start_line = 1;
68.     if (start_row == 0) start_line = 0;
69.     int end_line = sub_row_size - 2;
70.     if (end_row == row_size) end_line = sub_row_size - 1;
71.
72.     //if (rank == 2) {
73.     //    cout << "\n start_row = " << start_row << endl;
74.     //    cout << "\n end_row = " << end_row << endl;
75.     //    cout << "\n start_line = " << start_line << endl;
76.     //    cout << "\n end_line = " << end_line << endl;
77.     //    cout << "\n sub_row_size = " << sub_row_size << endl;
78.     //}
79.
80.     for (int i = start_line; i <= end_line; i++) {
81.         for (int j = 0; j < col_size; j++) {
82.             int count = 0;
83.             int sum = 0;
84.             for (int m = i - 1; m <= i + 1; m++) {
85.                 for (int n = j - 1; n <= j + 1; n++) {
86.                     if (m >= 0 && m < sub_row_size && n >= 0 && n
                        < col_size && !(m == i && n == j)) {
87.                         // if (rank == 1 && i == 0 && j == 0) cout
                        t << "!!! " << mat[m][n];

```

```

88.                sum += mat[m][n];
89.                count++;
90.            }
91.        }
92.    }
93.    float average = static_cast<float>(sum) / count;
94.    int centerElement = mat[i][j];
95.    // if (rank == 1 && i == 0 && j == 0) cout << "count:
    " << count << " avg: " << average << endl;
96.    if (abs(centerElement - average) / average >= 0.1) {
97.        resultMat[i][j] = static_cast<int>(average);
98.    }
99.    }
100.    }
101.
102.    for (int i = start_line; i <= end_line; i++) {
103.        MPI_Send(&resultMat[i][0], col_size, MPI_INT, 0, 0, MP
    I_COMM_WORLD);
104.    }
105.
106.    //if (rank == 2) {
107.    //    printMatrix(resultMat, sub_row_size, col_size);
108.    //    cout << endl;
109.    //}
110.
111.    // Free memory
112.    for (int i = 0; i < sub_row_size; i++) {
113.        delete[] mat[i];
114.    }
115.    delete[] mat;
116.
117.    for (int i = 0; i < sub_row_size; i++) {
118.        delete[] resultMat[i];
119.    }
120.    delete[] resultMat;
121.    }
122.
123.    if (rank == 0) {
124.        int** resultMatrix = new int* [row_size];
125.        for (int i = 0; i < row_size; i++) {
126.            resultMatrix[i] = new int[col_size];
127.            for (int j = 0; j < col_size; j++) {
128.                resultMatrix[i][j] = mat[i][j];

```

```

129.         }
130.     }
131.
132.     for (int dest_rank = 1; dest_rank < size; dest_rank++) {
133.         int dest_start_row = (dest_rank - 1) * rows_per_process;
134.         int dest_end_row = dest_rank * rows_per_process;
135.         if (dest_rank == size - 1) {
136.             dest_end_row += extra_rows;
137.         }
138.
139.         int subrow_size = dest_end_row - dest_start_row;
140.         int** subresultMatrix = new int* [subrow_size];
141.         /*cout << "receiving process:
142.         " << dest_rank << "/" << subrow_size << endl;*/
143.         for (int i = 0; i < subrow_size; i++) {
144.             subresultMatrix[i] = new int[col_size];
145.             MPI_Recv(&subresultMatrix[i][0], col_size, MPI_INT
146.             , dest_rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
147.             /*cout << "debug:
148.             " << dest_rank << "/" << i << endl;*/
149.             /*if (rank == 1) for (int k = 0; k < col_size; k++)
150.             ) cout << "*** " << subresultMatrix[i][k] << " " << endl;*/
151.         }
152.
153.         // Copy the subresultMatrix to the main resultMatrix
154.         for (int i = dest_start_row; i < dest_end_row; i++) {
155.             for (int j = 0; j < col_size; j++) {
156.                 /*cout << "receiving process:
157.                 " << i << "/" << j << endl;*/
158.                 resultMatrix[i][j] = subresultMatrix[i - dest_
159.                 start_row][j];
160.             }
161.         }
162.
163.         // Free memory
164.         for (int i = 0; i < subrow_size; i++) {
165.             delete[] subresultMatrix[i];
166.         }
167.         delete[] subresultMatrix;
168.     }
169.
170.     // printMatrix(resultMatrix, row_size, col_size);

```

```

165.
166.         // Copy the resultMatrix to the original mat array
167.         for (int i = 0; i < row_size; i++) {
168.             for (int j = 0; j < col_size; j++) {
169.                 mat[i][j] = resultMatrix[i][j];
170.             }
171.         }
172.
173.         // Free memory
174.         for (int i = 0; i < row_size; i++) {
175.             delete[] resultMatrix[i];
176.         }
177.         delete[] resultMatrix;
178.     }
179. }
180.
181. int** MyMat::product(int** matA, int rowNumA, int** matB, int colN
    umB)
182. {
183.     int rank, size;
184.     int common_size = 11;
185.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
186.     MPI_Comm_size(MPI_COMM_WORLD, &size);
187.
188.     int rows_per_process = rowNumA / size;
189.
190.     int* A = nullptr, *B = new int[common_size * colNumB], *C = n
        ullptr;
191.     int* subA = new int[rows_per_process * common_size];
192.     int* subC = new int[rows_per_process * colNumB];
193.
194.     // 1. 初始化一维矩阵
195.     if (rank == 0) {
196.         A = new int[rowNumA * common_size];
197.         int index = 0;
198.         for (int i = 0; i < rowNumA; i++) {
199.             for (int j = 0; j < common_size; j++) {
200.                 A[index++] = matA[i][j];
201.             }
202.         }
203.
204.         B = new int[common_size * colNumB];
205.         index = 0;
206.         for (int i = 0; i < common_size; i++) {

```



```

207.         for (int j = 0; j < colNumB; j++) {
208.             B[index++] = matB[i][j];
209.         }
210.     }
211.     C = new int[rowNumA * colNumB];
212. }
213. // 2. 阻塞所有进程，保证所有进程所需变量初始化成功
214. MPI_Barrier(MPI_COMM_WORLD);
215.
216. // 3. 将矩阵 A 拆分发送给各个进程，Scatter()方法发送对象包括父进程本
    身
217. MPI_Scatter(A, rows_per_process * common_size, MPI_INT, subA,
    rows_per_process * common_size, MPI_INT, 0, MPI_COMM_WORLD);
218. // 4. 将矩阵 B 完整地广播出去
219. MPI_Bcast(B, common_size * colNumB, MPI_INT, 0, MPI_COMM_WORLD
    );
220.
221. //cout << "this is process:" << rank << endl;
222.
223. // 5. 单个进程对矩阵的处理
224. matMultiplyWithSingleThread(subA, B, subC, rows_per_process, c
    ommon_size, colNumB);
225.
226. // 6. 汇集各个进程的结果
227. MPI_Gather(subC, rows_per_process * colNumB, MPI_INT, C, rows_
    per_process * colNumB, MPI_INT, 0, MPI_COMM_WORLD);
228.
229. // 7. 用进程 0 处理剩余的行
230. int remainRowsStartId = rows_per_process * size;
231. if (rank == 0 && remainRowsStartId < rowNumA) {
232.     int remainRows = rowNumA - remainRowsStartId;
233.     matMultiplyWithSingleThread(A + remainRowsStartId * common
        _size, B, C + remainRowsStartId * colNumB, remainRows, common_size, c
        olNumB);
234. }
235.
236. delete[] subA;
237. delete[] subC;
238. delete[] B;
239.
240. int** matC = nullptr;
241. if (rank == 0) {
242.     // 主进程获取结果矩阵
243.     matC = new int* [rowNumA];

```

```

244.         for (int i = 0; i < rowNumA; i++) {
245.             matC[i] = new int[colNumB];
246.             for (int j = 0; j < colNumB; j++) {
247.                 matC[i][j] = C[i * colNumB + j];
248.             }
249.         }
250.         delete[] A;
251.         delete[] C;
252.     }
253.
254.     return matC;
255. }
256.
257. int** MyMat::sparseProduct(int** matA, int rowNumA, int** matB, in
    t colNumB)
258. {
259.     return nullptr;
260. }
261.
262. int** MyMat::getMat(int row, int col)
263. {
264.     // 初始化随机数生成器
265.     random_device rd;
266.     mt19937 gen(rd());
267.     uniform_int_distribution<int> dis(0, 100);
268.
269.     int** inputMatrix = new int* [row];
270.     for (int i = 0; i < row; ++i) {
271.         inputMatrix[i] = new int[col];
272.         for (int j = 0; j < col; ++j) {
273.             // 随机生成数值
274.             inputMatrix[i][j] = dis(gen);
275.         }
276.     }
277.
278.     return inputMatrix;
279. }
280.
281. int** MyMat::getMatA(int row)
282. {
283.     int** ret = getMat(row, 11);
284.     int digits[] = { 2, 0, 2, 0, 1, 0, 0, 3, 7, 2, 9 };
285.     // 把第一行设置为学号
286.     for (int i = 0; i < 11; i++) {

```

```
287.         ret[0][i] = digits[i];
288.     }
289.     return ret;
290. }
291.
292. int** MyMat::getMatB(int col)
293. {
294.     return getMat(11, col);
295. }
296.
297. void MyMat::printMatrix(int** matrix, int row_size, int col_size)
298. {
299.     for (int i = 0; i < row_size; ++i) {
300.         cout << "第" << i << "行: ";
301.         for (int j = 0; j < col_size; ++j) {
302.             cout << matrix[i][j] << " ";
303.         }
304.         cout << endl;
305.     }
306.
307. int** MyMat::getSparseMatrix(int row_size, int col_size, double sp
arsity)
308. {
309.     int** sparseMatrix = new int* [row_size];
310.
311.     for (int row = 0; row < row_size; row++) {
312.         sparseMatrix[row] = new int[col_size];
313.
314.         for (int col = 0; col < col_size; col++) {
315.             // 生成随机值, 根据给定的稀疏度确定是否为零
316.             if (static_cast<double>(rand()) / RAND_MAX > sparsity)
317.             {
318.                 sparseMatrix[row][col] = 0;
319.             }
320.             else {
321.                 // 生成非零随机值, 范围在 1 到 9 之间
322.                 sparseMatrix[row][col] = rand() % 100 + 1;
323.             }
324.         }
325.
326.         return sparseMatrix;
327.     }
```

```

328.
329. void MyMat::matMultiplyWithSingleThread(int* A, int* B, int* matRe
    sult, int m, int p, int n)
330. {
331.     for (int i = 0; i < m; i++) {
332.         for (int j = 0; j < n; j++) {
333.             float temp = 0;
334.             for (int k = 0; k < p; k++) {
335.                 temp += A[i * p + k] * B[k * n + j];
336.             }
337.             matResult[i * n + j] = temp;
338.         }
339.     }
340. }
341.
342. int** MyMat::getSparseMatA(int row_size, double sparsity)
343. {
344.     int** ret = getSparseMatrix(row_size, 11, sparsity);
345.     int digits[] = { 2, 0, 2, 0, 1, 0, 0, 3, 7, 2, 9 };
346.     // 把第一行设置为学号
347.     for (int i = 0; i < 11; i++) {
348.         ret[0][i] = digits[i];
349.     }
350.     return ret;
351. }
352.
353. int** MyMat::getSparseMatB(int col_size, double sparsity)
354. {
355.     return getSparseMatrix(11, col_size, sparsity);
356. }

```

```

1.  /*
2.     作者: 刘星雨
3.     学号: 20201003729
4.     文件名: "mySparseMat.cpp"
5.  */
6.
7.  #include "mySparseMat.h"
8.
9.  void MySparseMat::display(RLSMatrix M)
10. {
11.     cout << "test::::" << M.mu << " && " << M.nu << endl;
12.     int i, j, k;
13.     for (i = 1; i <= M.mu; i++) {
14.         for (j = 1; j <= M.nu; j++) {

```

```

15.         int value = 0;
16.         //输出前 mu - 1 行矩阵
17.         if (i + 1 <= M.mu) {
18.             for (k = M.rowOffset[i]; k < M.rowOffset[i + 1]; k++)
19.             {
20.                 if (i == M.data[k].i && j == M.data[k].j) {
21.                     printf("%d ", M.data[k].e);
22.                     value = 1;
23.                     break;
24.                 }
25.             }
26.             if (value == 0) {
27.                 printf("0 ");
28.             }
29.             //输出矩阵最后一行的数据
30.             else {
31.                 for (k = M.rowOffset[i]; k <= M.tu; k++) {
32.                     if (i == M.data[k].i && j == M.data[k].j) {
33.                         printf("%d ", M.data[k].e);
34.                         value = 1;
35.                         break;
36.                     }
37.                 }
38.                 if (value == 0) {
39.                     printf("0 ");
40.                 }
41.             }
42.         }
43.         printf("\n");
44.     }
45. }
46.
47. RLSMatrix MySparseMat::compressSparseMat(int** mat, int row, int col)
48. {
49.     RLSMatrix comMat;
50.     comMat.mu = row;
51.     comMat.nu = col;
52.     Triple data[1000]; // 从1开始存储
53.     int rowOffset[1000] = { 0 }; // 从1开始存储
54.     int tu = 0; // 表示非零元素的个数
55.
56.

```

```
57.     for (int i = 0; i < row; i++) {
58.         bool flag = true; // 如果找到了该行第一个元素，则设置为 false
59.         for (int j = 0; j < col; j++) {
60.             if (mat[i][j] != 0) { // 记录非零元素
61.                 tu++;
62.                 data[tu].e = mat[i][j];
63.                 data[tu].i = i + 1;
64.                 data[tu].j = j + 1;
65.
66.                 if (flag) { // 记录第一个非零元素在 data 数组里的位置
67.                     flag = false;
68.                     rowOffset[i + 1] = tu;
69.                 }
70.             }
71.         }
72.         if (flag == true) { //遍历整行没有找到非零元素，则将 offset 值设置
           为 tu+1
73.             rowOffset[i + 1] = -1;
74.         }
75.     }
76.
77.     for (int i = 0; i < 1000; i++) {
78.         if (rowOffset[i] == -1) {
79.             rowOffset[i] = tu + 1;
80.         }
81.     }
82.
83.     comMat.mu = row;
84.     comMat.nu = col;
85.     comMat.tu = tu;
86.
87.     // 将 data 数组复制到 comMat 的 data 数组中
88.     for (int i = 1; i <= tu; i++) {
89.         comMat.data[i] = data[i];
90.     }
91.
92.     // 将 rowOffset 数组复制到 comMat 的 rowOffset 数组中
93.     for (int i = 1; i <= row; i++) {
94.         comMat.rowOffset[i] = rowOffset[i];
95.     }
96.
97.     return comMat;
98. }
99.
```

```

100.  RLSMatrix MySparseMat::product(RLSMatrix A, RLSMatrix B, RLSMatrix
    C)
101.  {
102.      int rank, size;
103.      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
104.
105.      if (rank != 0) {
106.          return RLSMatrix();
107.      }
108.
109.      // 初始化矩阵 C 的各个参数
110.      C.mu = A.mu;
111.      C.nu = B.nu;
112.      C.tu = 0;
113.
114.      /*
115.          判断矩阵点积的合法性
116.          要求 1. 矩阵 A 的列数=矩阵 B 的行数
117.          要求 2. 两个矩阵必须均存在非零值，否则乘法运算没有意义
118.      */
119.      if (A.nu != B.mu || A.tu * B.tu == 0) {
120.          return C;
121.      }
122.
123.      int arow, ccol;
124.
125.      //遍历矩阵 A 的每一行
126.      for (arow = 1; arow <= A.mu; arow++)
127.      {
128.          //创建一个临时存储乘积结果的数组，且初始化为 0，遍历每次都需要清
            空
129.          int ctemp[1000] = {};
130.          C.rowOffset[arow] = C.tu + 1;
131.          //根据行数，在三元组表中找到该行所有的非 0 元素的位置
132.          int tp;
133.          if (arow < A.mu)
134.              tp = A.rowOffset[arow + 1]; //获取矩阵 A 的下一行第一个非零
                元素在 data 数组中位置
135.          else
136.              tp = A.tu + 1; //若当前行是最后一行，则取最后一个元素+1
137.
138.          int p;
139.          int brow;
140.          //遍历当前行的所有的非 0 元素

```

```

141.         for (p = A.rowOffset[arow]; p < tp; p++)
142.         {
143.             brow = A.data[p].j; //取该非 0 元素的列数，便于去 B 中找对应的
                做乘积的非 0 元素
144.             int t;
145.             // 判断如果对于 A 中非 0 元素，找到矩阵 B 中做乘法的那一行中的
                所有的非 0 元素
146.             if (brow < B.mu)
147.                 t = B.rowOffset[brow + 1];
148.             else
149.                 t = B.tu + 1;
150.             int q;
151.             //遍历找到的对应的非 0 元素，开始做乘积运算
152.             for (q = B.rowOffset[brow]; q < t; q++)
153.             {
154.                 //得到的乘积结果，每次和 ctemp 数组中相应位置的数值做加
                和运算
155.                 ccol = B.data[q].j;
156.                 ctemp[ccol] += A.data[p].e * B.data[q].e;
157.             }
158.         }
159.         //矩阵 C 的行数等于矩阵 A 的行数，列数等于矩阵 B 的列数，所以，得到
                的 ctemp 存储的结果，也会在 C 的列数的范围内
160.         for (ccol = 1; ccol <= C.nu; ccol++)
161.         {
162.             //由于结果可以是 0，而 0 不需要存储，所以在这里需要判断
163.             if (ctemp[ccol])
164.             {
165.                 //不为 0，则记录矩阵中非 0 元素的个数的变量 tu 要+1；且该
                值不能超过存放三元素数组的空间大小
166.                 if (++C.tu > 1000)
167.                     return C;
168.                 else {
169.                     C.data[C.tu].e = ctemp[ccol];
170.                     C.data[C.tu].i = arow;
171.                     C.data[C.tu].j = ccol;
172.                 }
173.             }
174.         }
175.     }
176.     return C;
177. }
178.

```



```

179.  RLSMatrix MySparseMat::parallelProduct(RLSMatrix A, RLSMatrix B, R
    LMatrix C)
180.  {
181.      int rank, size;
182.      int rowA, colA, tA, rowB, colB, tB;
183.
184.      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
185.      MPI_Comm_size(MPI_COMM_WORLD, &size);
186.
187.      // 最终回收的数据
188.      Triple* dataC = nullptr;
189.      int* rowOffsetC = nullptr;
190.      int rowC(0), colC(0), tC(0);
191.      // 子进程进行处理的数据
192.      Triple* sub_dataC = nullptr;
193.      int* sub_rowOffsetC = nullptr;
194.      int sub_tC(0);
195.
196.      if (rank == 0) {
197.          rowA = A.mu;
198.          colA = A.nu;
199.          tA = A.tu;
200.          rowB = B.mu;
201.          colB = B.nu;
202.          tB = B.tu;
203.      }
204.
205.      // 将矩阵公用部分的参数广播出去
206.      MPI_Bcast(&rowA, 1, MPI_INT, 0, MPI_COMM_WORLD);
207.      MPI_Bcast(&colA, 1, MPI_INT, 0, MPI_COMM_WORLD);
208.      MPI_Bcast(&rowB, 1, MPI_INT, 0, MPI_COMM_WORLD);
209.      MPI_Bcast(&colB, 1, MPI_INT, 0, MPI_COMM_WORLD);
210.
211.      // 创建 MPI 的三元组数据类型
212.      MPI_Datatype MPI_TRIPLE = createMPI_Triple();
213.
214.      int rows_per_process = A.mu / (size - 1);
215.      int extra_rows = A.mu % (size - 1);
216.
217.      /*
218.          主进程分发数据
219.          1. data 数据
220.          2. rowOffset 数据
221.          3. 其他数据在子进程中重新计算获取（不占用主进程的计算资源）

```

```

222.     */
223.     if (rank == 0) {
224.         for (int dest_rank = 1; dest_rank < size; dest_rank++) {
225.             // 计算起止行数 (从 1 开始)
226.             int start_row = (dest_rank - 1) * rows_per_process + 1
                ;
227.             int end_row = dest_rank * rows_per_process;
228.             if (dest_rank == size - 1) { // 最后一个进程包揽多余的
                行
229.                 end_row += extra_rows;
230.             }
231.
232.             // 计算对应的非零元素在表中的范围
233.             int start_data = A.rowOffset[start_row];
234.             int len_data = A.rowOffset[end_row] - start_data;
235.
236.             // 分发矩阵 A 的数据
237.             MPI_Send(&A.rowOffset[start_row], end_row - start_row
                + 1, MPI_INT, dest_rank, 0, MPI_COMM_WORLD);
238.             MPI_Send(&A.data[start_data], len_data, MPI_TRIPLE, de
                st_rank, 0, MPI_COMM_WORLD);
239.
240.
241.             // 分发矩阵 B 的数据
242.             MPI_Send(&B.rowOffset[1], rowB, MPI_INT, dest_rank, 0,
                MPI_COMM_WORLD);
243.             MPI_Send(&B.data[1], tB, MPI_TRIPLE, dest_rank, 0, MPI
                _COMM_WORLD);
244.
245.         }
246.     }
247.     else {
248.         int rows = rows_per_process;
249.         // 最后一个进程包揽多余的行
250.         if (rank == size - 1) {
251.             rows += extra_rows;
252.         }
253.         // 计算起止行数
254.         int start_row = (rank - 1) * rows_per_process + 1;
255.         int end_row = rank * rows_per_process;
256.         int len_data; // data 数组中有效的非零元素个数
257.
258.         // 按照行数分配 Offset
259.         int* rowOffset = new int[end_row - start_row + 1];

```

```

260.          // 按比例分配 data 的长度
261.          Triple* data = new Triple[1000 * rows / rowA];
262.
263.          MPI_Recv(&rowOffset[1], end_row - start_row + 1, MPI_INT,
264.                  0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
265.          len_data = rowOffset[end_row - start_row + 1] - rowOffset[
266.                  1];
267.          MPI_Recv(&data[1], len_data, MPI_TRIPLE, 0, 0, MPI_COMM_WO
268.                  RLD, MPI_STATUS_IGNORE);
269.
270.          // 处理 rowOffset 数组
271.          int delta = start_row - 1; // 另起始行数为第一行
272.          for (int i = 0; i < end_row - start_row + 1; i++) {
273.              rowOffset[i + 1] -= delta;
274.          }
275.
276.          // 定义分配到的子矩阵 A
277.          RLSMatrix subA;
278.          for (int i = 1; i <= end_row - start_row + 1; i++) {
279.              subA.data[i] = data[i];
280.          }
281.          for (int i = 1; i <= len_data; i++) {
282.              subA.rowOffset[i] = rowOffset[i];
283.          }
284.          subA.mu = rows;
285.          subA.nu = colA;
286.          subA.tu = len_data;
287.
288.          // 定义分到的完整矩阵 B
289.          RLSMatrix B;
290.          MPI_Recv(&B.rowOffset[1], rowB, MPI_INT, 0, 0, MPI_COMM_WO
291.                  RLD, MPI_STATUS_IGNORE);
292.          MPI_Recv(&B.data[1], tB, MPI_TRIPLE, 0, 0, MPI_COMM_WORLD,
293.                  MPI_STATUS_IGNORE);
294.          B.mu = rowB;
295.          B.nu = colB;
296.          B.tu = tB;
297.
298.          // 获取子矩阵
299.          RLSMatrix res;
300.          res = product(subA, B, res);
301.
302.          dataC = new Triple[res.tu + 1];
303.          for (int i = 1; i <= res.tu; i++) {

```

```

299.             sub_dataC[i] = res.data[i];
300.         }
301.
302.         rowOffsetC = new int[res.mu + 1];
303.         for (int i = 1; i <= res.mu; i++) {
304.             sub_rowOffsetC[i] = res.rowOffset[i];
305.         }
306.
307.         sub_tC = res.tu;
308.     }
309.
310.     // 主进程收集各子进程的计算结果
311.     if (rank == 0) {
312.         MPI_Gather(dataC, tC, MPI_TRIPLE, sub_dataC, tC, MPI_INT,
313.             0, MPI_COMM_WORLD);
314.         MPI_Gather(rowOffsetC, rowA, MPI_TRIPLE, sub_rowOffsetC, r
315.             owA, MPI_INT, 0, MPI_COMM_WORLD);
316.         rowC = rowA;
317.         colC = colB;
318.         RLSMatrix resC;
319.         for (int i = 0; i < tC; i++) {
320.             resC.data[i] = dataC[i];
321.         }
322.         for (int i = 0; i < rowA; i++) {
323.             resC.rowOffset[i] = rowOffsetC[i];
324.         }
325.         resC.mu = rowC;
326.         resC.nu = colC;
327.         resC.tu = tC;
328.
329.         return resC;
330.     }
331.     return RLSMatrix();
332. }
333.
334. int** MySparseMat::getSparseMatrix(int row_size, int col_size, dou
335.     ble sparsity)
336. {
337.     int** sparseMatrix = new int* [row_size];
338.     for (int row = 0; row < row_size; row++) {
339.         sparseMatrix[row] = new int[col_size];

```

```

340.
341.         for (int col = 0; col < col_size; col++) {
342.             // 生成随机值，根据给定的稀疏度确定是否为零
343.             if (static_cast<double>(rand()) / RAND_MAX > sparsity)
344.             {
345.                 sparseMatrix[row][col] = 0;
346.             }
347.             else {
348.                 // 生成非零随机值，范围在 1 到 9 之间
349.                 sparseMatrix[row][col] = rand() % 100 + 1;
350.             }
351.         }
352.
353.     return sparseMatrix;
354. }
```

```

1.  /*
2.     作者: 刘星雨
3.     学号: 20201003729
4.     文件名: "mySparseMat.cpp"
5.  */
6.
7.  #include "myMat.h"
8.  #include "mySparseMat.h"
9.  #include <chrono>
10. #include <fstream>
11.
12. int main(int argc, char* argv[]) {
13.     MPI_Init(&argc, &argv);
14.     int rank;
15.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16.     int rowNumA;
17.     int colNumB;
18.     MyMat myMat;
19.     int** matA = nullptr;
20.     int** matB = nullptr;
21.
22.     if (rank == 0) {
23.         cout <<
24.             "+-----+
25.             |

```

```

26.          "|
              |\n"
27.          "| 说明：该程序作者是 111203 刘星雨，用于完成李程俊老师的课程
设计题目    。内容有    |\n"
28.          "| 三：1 使用类 3*3 均值滤波对矩阵进行预处理 ， 2 完成通用矩阵
乘法 ， 3 完成稀疏矩阵    |\n"
29.          "| 乘
法。
              |\n"
30.          "|
              |\n"
31.          "| 注意：矩阵分为 A、B 两个 ， 且 A 矩阵的第一行必须为本人的学号
20201003729，这个    |\n"
32.          "| 条件规定了矩阵 A 的列数为 11，矩阵 B 的行数为 11.综上所述，
用户使用该程序时只    |\n"
33.          "| 需要输入矩阵 A 的行数和矩阵 B 的列
数！                                |\n"
34.          "|
              |\n"
35.          "|
              |\n"
36.          "|
              |\n"
37.          "|
              +-----
+                                |\n"
38.          "|                                输
入                                |\n"
39.          "|                                |
              |\n"
40.          "|                                | 矩阵 A 行数：-----
-                                |\n"
41.          "|                                |
              |\n"
42.          "|                                | 矩阵 B 行数：-----
-                                |\n"
43.          "|                                |
              |\n"
44.          "|                                +-----
+                                |\n"
45.          "|
              |\n"
46.          "|
              |\n"

```

```

47.         "+-----+\n"
         -----+\n"
48.         << endl;
49.         cout << "矩阵 A 行数: ";
50.         cin >> rowNumA;
51.         cout << endl;
52.         cout << "矩阵 B 列数: ";
53.         cin >> colNumB;
54.         cout << endl;
55.
56.         // 生成矩阵 A
57.         matA = myMat.getMatA(rowNumA);
58.         cout << "matrix A:" << endl;
59.         // myMat.printMatrix(matA, rowNumA, 11);
60.         cout << endl;
61.
62.         // 生成矩阵 B
63.         matB = myMat.getMatB(colNumB);
64.         cout << "matrix B:" << endl;
65.         // myMat.printMatrix(matB, 11, colNumB);
66.         cout << endl;
67.
68.
69.
70.     }
71.
72.     // 把获取的矩阵大小信息广播给所有线程
73.     MPI_Bcast(&rowNumA, 1, MPI_INT, 0, MPI_COMM_WORLD);
74.     MPI_Bcast(&colNumB, 1, MPI_INT, 0, MPI_COMM_WORLD);
75.
76.     int input = 0;
77.     while (input != 7) {
78.         if (rank == 0) {
79.             cout <<
80.                 "+-----+\n"
81.                 -----+\n"
82.                 "|
83.                 |\n"
84.                 "|    输入数
85.                 字
86.                 |\n"
87.                 "|    1. 重新生成稀疏矩阵
88.                 A。
89.                 |\n"

```

```

85.          "|    2. 重新生成稀疏矩阵
    B。          |\n"
86.          "|    3. 对矩阵 A 进行预处
    理。          |\n"
87.          "|    4. 对矩阵 B 进行预处
    理。          |\n"
88.          "|    5. 通用矩阵乘法
    A * B。          |\n"
89.          "|    6. 稀疏矩阵乘法
    A * B。          |\n"
90.          "|    7. 退
    出。          |\n"
91.          "|
    |\n"
92.          "+-----+
    -----+|\n"
93.          << endl;
94.          cin >> input;
95.          }
96.
97.          MPI_Bcast(&input, 1, MPI_INT, 0, MPI_COMM_WORLD);
98.
99.          switch (input) {
100.             case 1: {
101.                 if (rank == 0) { // 只用父进程生成稀疏矩阵
102.                     for (int i = 0; i < rowNumA; i++) {
103.                         delete[] matA[i];
104.                     }
105.                     delete[] matA;
106.                     matA = myMat.getSparseMatA(rowNumA, 0.15);
107.                     myMat.printMatrix(matA, rowNumA, 11);
108.                     cout << endl;
109.                 }
110.                 break;
111.             }
112.             case 2: {
113.                 if (rank == 0) { // 只用父进程生成稀疏矩阵
114.                     for (int i = 0; i < 11; i++) {
115.                         delete[] matB[i];
116.                     }
117.                     delete[] matB;
118.                     matB = myMat.getSparseMatB(colNumB, 0.15);
119.                     myMat.printMatrix(matB, 11, colNumB);
120.                     cout << endl;

```



```
121.         }
122.         break;
123.     }
124.     case 3: {
125.         // 记录程序开始时间
126.         auto start_time = std::chrono::high_resolution_clock::
            now();
127.         myMat.meanFilter(matA, rowNumA, 11);
128.         MPI_Barrier(MPI_COMM_WORLD);
129.         if (rank == 0) {
130.             // 记录程序结束时间
131.             auto end_time = std::chrono::high_resolution_clock
                ::now();
132.             // 计算程序运行时间
133.             auto duration = std::chrono::duration_cast<std::ch
                rono::milliseconds>(end_time - start_time).count();
134.             // 输出运行时间
135.             cout << duration << endl;
136.             //cout << "对矩阵 A 进行预处理以后: " << endl;
137.             //myMat.printMatrix(matA, rowNumA, 11);
138.             //cout << endl;
139.         }
140.         break;
141.     }
142.     case 4: {
143.         // 记录程序开始时间
144.         auto start_time = std::chrono::high_resolution_clock::
            now();
145.         myMat.meanFilter(matB, 11, colNumB);
146.         MPI_Barrier(MPI_COMM_WORLD);
147.         if (rank == 0) {
148.             // 记录程序结束时间
149.             auto end_time = std::chrono::high_resolution_clock
                ::now();
150.             // 计算程序运行时间
151.             auto duration = std::chrono::duration_cast<std::ch
                rono::milliseconds>(end_time - start_time).count();
152.             // 输出运行时间
153.             cout << duration << endl;
154.             //cout << "对矩阵 B 进行预处理以后: " << endl;
155.             //myMat.printMatrix(matB, 11, colNumB);
156.             //cout << endl;
157.         }
158.         break;
```

```

159.         }
160.         case 5: {
161.             // 记录程序开始时间
162.             auto start_time = std::chrono::high_resolution_clock::
                now();
163.             int** C = myMat.product(matA, rowNumA, matB, colNumB);

164.             MPI_Barrier(MPI_COMM_WORLD);
165.             if (rank == 0) {
166.                 // 记录程序结束时间
167.                 auto end_time = std::chrono::high_resolution_clock
                    ::now();
168.                 // 计算程序运行时间
169.                 auto duration = std::chrono::duration_cast<std::ch
                    rono::milliseconds>(end_time - start_time).count();
170.                 cout << duration << endl;
171.                 cout << "（稀疏）矩阵点积结果: " << endl;
172.                 myMat.printMatrix(C, rowNumA, colNumB);
173.                 cout << endl;
174.                 for (int i = 0; i < rowNumA; i++) {
175.                     delete[] C[i];
176.                 }
177.                 delete[] C;
178.             }
179.             break;
180.         }
181.         case 6: {
182.             MySparseMat sparseMat;
183.             // 使用行逻辑顺序表压缩稀疏矩阵
184.             RLMatrix compressedMatA;
185.             RLMatrix compressedMatB;
186.             RLMatrix compressedMatC;
187.             // 主进程对矩阵进行压缩处理
188.             if (rank == 0) {
189.                 RLMatrix compressedMatA = sparseMat.compressSpars
                    eMat(matA, rowNumA, 11);
190.                 RLMatrix compressedMatB = sparseMat.compressSpars
                    eMat(matB, 11, colNumB);
191.                 compressedMatC = sparseMat.product(compressedMatA,
                    compressedMatB, compressedMatC);
192.             }
193.
194.             // 打印乘法结果
195.             if (rank == 0) {

```

```
196.             sparseMat.display(compressedMatC);
197.         }
198.     }
199.     case 7: {break; }
200.     default: {break; }
201. }
202. }
203.
204.
205.
206.
207.     MPI_Finalize();
208. }
```