中山大学计算机院本科生实验报告

（2024 学年春季学期）

课程名称：并行程序设计	批改人:
实验：2	专业（方向）：计算机科学与技术
学号：22336226	姓名：王泓沣
Email：wanghf59@mail2.sysu.edu.cn	完成日期：2025/4/6

实验目的

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；
2. 尝试不同数据/任务划分方式（选做）。
3. 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。

实验过程和核心代码

集合通信方式相较于点对点通信方式无需手动显式管理消息匹配，并且能在内部使用优化算法，MPI 点对点通信使用 `MPI_Send` 和 `MPI_Recv` 分别将数据分发给各个进程，再由根进程逐个收集局部结果，而 MPI 集合通信使用 `MPI_Bcast`、`MPI_Scatterv` 和 `MPI_Gatherv` 实现数据的广播、分散和汇总。

辅助函数

首先仍定义打印矩阵和矩阵乘法两个辅助函数：

- 打印矩阵

```
void print_matrix(double *mat, int rows, int cols) {
    for (int i = 0; i < rows; i++){
        for (int j = 0; j < cols; j++){
```

```
            printf("%8.2f ", mat[i * cols + j]);
        }
        printf("\n");
    }
}
```

- 矩阵乘法

```
void matrix_multiply(double *A, double *B, double *C, int local_rows,
    int n, int k) {
    for (int i = 0; i < local_rows; i++){
        for (int j = 0; j < k; j++){
            C[i * k + j] = 0.0;
            for (int l = 0; l < n; l++){
                C[i * k + j] += A[i * n + l] * B[l * k + j];
            }
        }
    }
}
```

主函数

初始化阶段

1. 初始化并解析命令行参数

```
    int rank, size;
    int m, n, k;        // A: m×n, B: n×k, C: m×k
    int method;         // 0: 块划分, 1: 循环划分, 2: 块循环划分
    int block_size = 16; // 仅对循环划分方法 (方法2) 有效, 默认块大小

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        if (argc < 4) {
            fprintf(stderr, "Usage: %s m n k [method] [block_size]\n"
                , argv[0]);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        m = atoi(argv[1]);
        n = atoi(argv[2]);
```

```
        k = atoi(argv[3]);
        method = (argc >= 5) ? atoi(argv[4]) : 0;
        if (method == 2) {
            block_size = (argc >= 6) ? atoi(argv[5]) : 16;
        }
    }
```

2. 使用集合通信方式，广播 m, n, k, method 以及（对块循环）block_size 到所有进程并按照划分方式分配内存

```
    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&method, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if(method == 2)
        MPI_Bcast(&block_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // 所有进程分配 B（全矩阵B每个进程均需保存）
    double *B = (double*) malloc(n * k * sizeof(double));
    // 矩阵 A 和 C 仅在根进程中分配
    double *A = NULL;
    double *C = NULL;
    if (rank == 0) {
        A = (double*) malloc(m * n * sizeof(double));
        C = (double*) malloc(m * k * sizeof(double));
        srand(time(NULL));
        for (int i = 0; i < m * n; i++)
            A[i] = (double)(rand() % 10);
        for (int i = 0; i < n * k; i++)
            B[i] = (double)(rand() % 10);
    }
    // 广播 B 到所有进程
    MPI_Bcast(B, n * k, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // 根据不同划分方式计算每个进程将获得的 A 的行数 local_rows
    int local_rows = 0;
    if (method == 0) { // 块划分
        int rows_per_proc = m / size;
        int remainder = m % size;
        local_rows = (rank < remainder) ? rows_per_proc + 1 :
           rows_per_proc;
    } else if (method == 1) { // 循环划分
```

```
        for (int i = 0; i < m; i++) {
            if (i % size == rank)
                local_rows++;
        }
    } else if (method == 2) { // 块循环划分
        int num_blocks = (m + block_size - 1) / block_size;
        for (int j = 0; j < num_blocks; j++) {
            if (j % size == rank) {
                int start_row = j * block_size;
                int rows_in_block = ((start_row + block_size) <= m) ?
                    block_size : (m - start_row);
                local_rows += rows_in_block;
            }
        }
    }
    // 分配本地 A 和本地结果 C
    double *local_A = (double*) malloc(local_rows * n * sizeof(double
       ));
    double *local_C = (double*) malloc(local_rows * k * sizeof(double
       ));
```

数据分配阶段，按照划分方式选择分发方式

1. 块划分：利用 MPI_Scatterv 实现连续行分发

```
    if (method == 0) {
        int *sendcounts = (int*) malloc(size * sizeof(int));
        int *displs = (int*) malloc(size * sizeof(int));
        int rows_per_proc = m / size;
        int remainder = m % size;
        for (int i = 0; i < size; i++) {
            sendcounts[i] = (i < remainder ? rows_per_proc + 1 :
               rows_per_proc) * n;
        }
        displs[0] = 0;
        for (int i = 1; i < size; i++) {
            displs[i] = displs[i - 1] + sendcounts[i - 1];
        }
        MPI_Scatterv(A, sendcounts, displs, MPI_DOUBLE,
                     local_A, sendcounts[rank], MPI_DOUBLE, 0,
                        MPI_COMM_WORLD);
        free(sendcounts);
```

```
        free(displs);
    }
```

2. 循环划分：根进程按行下标 i % size 分发

```
    else if (method == 1) {
        if (rank == 0) {
            int idx = 0;
            // 根进程复制属于自己的行
            for (int i = 0; i < m; i++) {
                if (i % size == 0) {
                    for (int j = 0; j < n; j++)
                        local_A[idx * n + j] = A[i * n + j];
                    idx++;
                }
            }
            // 对于其他进程，打包并发送其所有行
            for (int p = 1; p < size; p++) {
                int count = 0;
                for (int i = 0; i < m; i++) {
                    if (i % size == p)
                        count++;
                }
                double *temp = (double*) malloc(count * n * sizeof(
                    double));
                int t = 0;
                for (int i = 0; i < m; i++) {
                    if (i % size == p) {
                        for (int j = 0; j < n; j++)
                            temp[t * n + j] = A[i * n + j];
                        t++;
                    }
                }
                MPI_Send(temp, count * n, MPI_DOUBLE, p, 1,
                    MPI_COMM_WORLD);
                free(temp);
            }
        } else {
            MPI_Recv(local_A, local_rows * n, MPI_DOUBLE, 0, 1,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }
```

3. 块循环划分：先按块（每块 block_size 行）划分，再循环分配

```
    else if (method == 2) {
        int num_blocks = (m + block_size - 1) / block_size;
        if (rank == 0) {
            int idx = 0;
            for (int j = 0; j < num_blocks; j++) {
                if (j % size == 0) {
                    int start_row = j * block_size;
                    int rows_in_block = ((start_row + block_size) <=
                       m) ? block_size : (m - start_row);
                    for (int i = 0; i < rows_in_block; i++) {
                        for (int j2 = 0; j2 < n; j2++) {
                            local_A[idx * n + j2] = A[(start_row + i)
                                * n + j2];
                        }
                        idx++;
                    }
                }
            }
            for (int p = 1; p < size; p++) {
                int count = 0;
                for (int j = 0; j < num_blocks; j++) {
                    if (j % size == p) {
                        int start_row = j * block_size;
                        int rows_in_block = ((start_row + block_size)
                            <= m) ? block_size : (m - start_row);
                        count += rows_in_block;
                    }
                }
                double *temp = (double*) malloc(count * n * sizeof(
                   double));
                int idx2 = 0;
                for (int j = 0; j < num_blocks; j++) {
                    if (j % size == p) {
                        int start_row = j * block_size;
                        int rows_in_block = ((start_row + block_size)
                            <= m) ? block_size : (m - start_row);
                        for (int i = 0; i < rows_in_block; i++) {
                            for (int j2 = 0; j2 < n; j2++) {
                                temp[idx2 * n + j2] = A[(start_row +
                                   i) * n + j2];
```

```
                        }
                        idx2++;
                    }
                }
            }
            MPI_Send(temp, count * n, MPI_DOUBLE, p, 1,
                MPI_COMM_WORLD);
            free(temp);
        }
    } else {
        MPI_Recv(local_A, local_rows * n, MPI_DOUBLE, 0, 1,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

计算阶段：各线程同步后开始计时，开始局部矩阵乘法计算

```
MPI_Barrier(MPI_COMM_WORLD);
double t_start = MPI_Wtime();
matrix_multiply(local_A, B, local_C, local_rows, n, k);
double t_end = MPI_Wtime();
double local_time = t_end - t_start;
```

结果收集阶段：将各进程计算得到的局部矩阵 C（尺寸 local_rows×k）汇总成全局矩阵 C

1. 块划分：利用 MPI_Gatherv 收集

```
if (method == 0) {
    int *recvcounts = (int*) malloc(size * sizeof(int));
    int *rdispls = (int*) malloc(size * sizeof(int));
    int rows_per_proc = m / size;
    int remainder = m % size;
    for (int i = 0; i < size; i++) {
        recvcounts[i] = (i < remainder ? rows_per_proc + 1 :
            rows_per_proc) * k;
    }
    rdispls[0] = 0;
    for (int i = 1; i < size; i++) {
        rdispls[i] = rdispls[i - 1] + recvcounts[i - 1];
    }
    MPI_Gatherv(local_C, recvcounts[rank], MPI_DOUBLE,
                C, recvcounts, rdispls, MPI_DOUBLE, 0,
                    MPI_COMM_WORLD);
```

```
        free(recvcounts);
        free(rdispls);
    }
```

2. 循环划分：每个进程发送 local_C，根进程按全局行号组装

```
    else if (method == 1) {
        if (rank == 0) {
            // 先将进程0中属于自己的行写入 C
            for (int i = 0, idx = 0; i < m; i++) {
                if (i % size == 0) {
                    for (int j = 0; j < k; j++)
                        C[i * k + j] = local_C[idx * k + j];
                    idx++;
                }
            }
            // 接收其他进程的数据，并按全局行号放置
            for (int p = 1; p < size; p++) {
                int count = 0;
                for (int i = 0; i < m; i++) {
                    if (i % size == p)
                        count++;
                }
                double *temp = (double*) malloc(count * k * sizeof(
                    double));
                MPI_Recv(temp, count * k, MPI_DOUBLE, p, 2,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                for (int i = 0; i < m; i++) {
                    if (i % size == p) {
                        int index = i / size;
                        for (int j = 0; j < k; j++){
                            C[i * k + j] = temp[index * k + j];
                        }
                    }
                }
                free(temp);
            }
        } else {
            MPI_Send(local_C, local_rows * k, MPI_DOUBLE, 0, 2,
                MPI_COMM_WORLD);
        }
    }
```

3. 块循环划分：根进程按块顺序组装全局 C

```
    else if (method == 2) {
        if (rank == 0) {
            int num_blocks = (m + block_size - 1) / block_size;
            int *proc_block_index = (int*) calloc(size, sizeof(int));
            // 处理根进程自身数据
            for (int j = 0; j < num_blocks; j++) {
                if (j % size == 0) {
                    int start_row = j * block_size;
                    int rows_in_block = ((start_row + block_size) <=
                       m) ? block_size : (m - start_row);
                    int idx = proc_block_index[0];
                    for (int i = 0; i < rows_in_block; i++){
                        for (int j2 = 0; j2 < k; j2++){
                            C[(start_row + i) * k + j2] = local_C[(
                               idx + i) * k + j2];
                        }
                    }
                    proc_block_index[0] += rows_in_block;
                }
            }
            // 接收其他进程数据，并按块组装
            for (int p = 1; p < size; p++){
                int count = 0;
                for (int j = 0; j < num_blocks; j++){
                    if(j % size == p) {
                        int start_row = j * block_size;
                        int rows_in_block = ((start_row + block_size)
                            <= m) ? block_size : (m - start_row);
                        count += rows_in_block;
                    }
                }
                double *temp = (double*) malloc(count * k * sizeof(
                   double));
                MPI_Recv(temp, count * k, MPI_DOUBLE, p, 2,
                   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                proc_block_index[p] = 0;
                for (int j = 0; j < num_blocks; j++){
                    if(j % size == p) {
                        int start_row = j * block_size;
                        int rows_in_block = ((start_row + block_size)
```

```
                        <= m) ? block_size : (m - start_row);
                    int idx = proc_block_index[p];
                    for (int i = 0; i < rows_in_block; i++){
                        for (int j2 = 0; j2 < k; j2++){
                            C[(start_row + i) * k + j2] = temp[(
                                idx + i) * k + j2];
                        }
                    }
                    proc_block_index[p] += rows_in_block;
                }
            }
            free(temp);
        }
        free(proc_block_index);
    } else {
        MPI_Send(local_C, local_rows * k, MPI_DOUBLE, 0, 2,
            MPI_COMM_WORLD);
    }
}
```

收尾阶段：结束计时，打印结果矩阵，释放内存

1. 计算各进程运行时间

```
double *all_times = NULL;
if (rank == 0) {
    all_times = (double*) malloc(size * sizeof(double));
    all_times[0] = local_time;
    for (int p = 1; p < size; p++){
        MPI_Recv(&all_times[p], 1, MPI_DOUBLE, p, 3,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    printf("各进程局部计算时间（秒）：\n");
    for (int p = 0; p < size; p++){
        printf("进程 %d: %f\n", p, all_times[p]);
    }
    free(all_times);
} else {
    MPI_Send(&local_time, 1, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);
}
```

2. 打印结果矩阵，释放内存

```
    if (rank == 0) {
        printf("结果矩阵 C (%d x %d):\n", m, k);
        print_matrix(C, m, k);
    }

    free(B);
    free(local_A);
    free(local_C);
    if (rank == 0) {
        free(A);
        free(C);
    }
    MPI_Finalize();
```

实验结果

取各进程最长的运行时间作为结果

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.003934	0.034690	0.271943	2.469981	51.423618
2	0.002001	0.016876	0.182980	1.328585	29.813501
4	0.000972	0.011387	0.088372	1.210692	21.732647
8	0.001503	0.011106	0.087083	0.884129	12.612047
16	0.000747	0.009533	0.081327	0.881348	12.246024

表 1: 块划分实验结果汇总

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.004196	0.037112	0.269955	2.605436	50.710618
2	0.002048	0.016770	0.137596	1.393587	28.943405
4	0.000963	0.013902	0.089927	1.466483	17.656560
8	0.001510	0.011937	0.076206	0.913765	13.033169
16	0.000743	0.008145	0.073003	0.851440	12.205676

表 2: 循环划分实验结果汇总

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.004204	0.033923	0.269179	2.669471	51.909093
2	0.001982	0.016712	0.140726	1.346181	28.945121
4	0.000957	0.013928	0.108981	1.173554	18.431350
8	0.001083	0.009280	0.083107	0.894761	12.130648
16	0.001283	0.005693	0.066320	0.905268	12.343759

表 3: 块循环划分实验结果汇总

块划分的优点在于实现简单、数据在内存中连续，有利于缓存局部性，在矩阵行间计算量大体均匀的场景下通常表现较好，缺点在于若行间存在负载不均（例如稀疏分布不均），可能出现某些进程负载偏重；循环划分的优点在于能将“繁重行”平均分配到各进程，负载均衡好；适合行间分布差异较大的情况，缺点在于行在内存中分布零散，缓存局部性差，通信打包与拆分开销可能增大。块循环划分优点在于在每个小块（block_size 行）内部保持数据连续，同时通过循环分配实现一定程度的负载均衡；是两者的折中方案，缺点在于实现相对复杂，且在问题非常均匀的情况下，其优势可能并不明显。

总体来看并行效率在中规模矩阵的加速效果最好，小规模矩阵计算加速不明显，大规模矩阵收益递减。

实验感想

首先是要熟悉 MPI 集中通信库函数的调用，其次是要分清根进程（进程 0）和其他进程的区别。另外，与点对点通信方式的一个显著区别在于块划分方式中使用 MPI 集中通信函数进行数据分发与收集，不用特别为进程 0 也显式安排计算工作。