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：1	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：2025/4/1

实验目的

使用 MPI 点对点通信方式实现并行通用矩阵乘法 (MPI-v1)，并通过实验分析不同进程数量、矩阵规模时该实现的性能。

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. 使用 MPI 点对点通信实现并行矩阵乘法，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。
2. 根据当前实现，在实验报告中讨论两个优化方向:
   a) 在内存有限的情况下，如何进行大规模矩阵乘法计算?
   b) 如何提高大规模稀疏矩阵乘法性能?

实验过程和核心代码

首先定义输出函数和矩阵乘法函数如下:

```
void print_matrix(double *mat, int rows, int cols) {
    int i, j;
    for(i = 0; i < rows; i++){
        for(j = 0; j < cols; j++){
            printf("%8.2f ", mat[i * cols + j]);
        }
        printf("\n");
    }
}
```

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

对于主函数部分，总体而言，在分配好各个进程所需的内存后，由进程 0 负责分配各进程所运算的矩阵部分、汇总运算结果后输出：

1. 初始化 MPI 环境，并获取当前进程的 rank 以及总进程数:

```
    int rank, size;
    int m, n, k; // 矩阵 A 为 m×n，矩阵 B 为 n×k，结果矩阵 C 为 m×k
    int i;

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

    double *A = NULL;       // 仅在进程0中分配
    double *B = NULL;       // 所有进程都需要存储完整的 B 矩阵
    double *C = NULL;       // 仅在进程0中分配完整的 C
    double *local_A = NULL; // 每个进程处理自己的 A 子块
    double *local_C = NULL; // 每个进程计算得到的局部 C
```

2. 仅在进程 0 中通过命令行参数获取矩阵尺寸 m, n, k，并利用 MPI_Send/MPI_Recv 将这些值发送给其他进程:

```
    if(rank == 0) {
        if(argc < 4) {
            fprintf(stderr, "Usage: %s m n k\n", argv[0]);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        m = atoi(argv[1]);
        n = atoi(argv[2]);
        k = atoi(argv[3]);
```

```
    }
    if(rank == 0) {
        for(i = 1; i < size; i++){
            MPI_Send(&m, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            MPI_Send(&k, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(&m, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
        MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
        MPI_Recv(&k, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
    }
```

3. 根据进程数将矩阵 A 按行划分，处理 m 不能整除 size 的情况，计算每个进程需要处理的行数及对应的偏移量:

```
    int rows_per_proc = m / size;
    int remainder = m % size;
    int local_rows = (rank < remainder) ? rows_per_proc + 1 :
       rows_per_proc;
    int offset;
    if(rank < remainder)
        offset = rank * (rows_per_proc + 1);
    else
        offset = remainder * (rows_per_proc + 1) + (rank - remainder)
            * rows_per_proc;
```

4. 分配内存：进程 0 分配完整的矩阵 A、B、C；其他进程分配矩阵 B 以及各自的 A 子块和局部矩阵 C:

```
    if(rank == 0) {
        A = (double*) malloc(m * n * sizeof(double));
        B = (double*) malloc(n * k * sizeof(double));
        C = (double*) malloc(m * k * sizeof(double));
        if(A == NULL || B == NULL || C == NULL) {
            fprintf(stderr, "内存分配失败\n");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        /*
```

```
         * 进程 0 初始化随机数种子，并随机填充矩阵 A（元素为 0-9 的随
           机整数）和矩阵 B。
         */
        srand(time(NULL));
        for(i = 0; i < m * n; i++){
            A[i] = (double)(rand() % 10);  // 随机整数 0-9
        }
        for(i = 0; i < n * k; i++){
            B[i] = (double)(rand() % 10);
        }
    }
    else {
        // 其他进程仅需分配 B 的空间
        B = (double*) malloc(n * k * sizeof(double));
        if(B == NULL) {
            fprintf(stderr, "进程 %d 分配 B 内存失败\n", rank);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
    }
    // 所有进程分配自己 A 子块及局部 C 的内存
    local_A = (double*) malloc(local_rows * n * sizeof(double));
    local_C = (double*) malloc(local_rows * k * sizeof(double));
    if(local_A == NULL || local_C == NULL) {
        fprintf(stderr, "进程 %d 分配局部内存失败\n", rank);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
```

5. 进程 0 将 A 的各子块及完整的矩阵 B 分发给其他进程；其他进程接收对应的 A 子块和矩阵 B:

```
    // 同步所有进程后开始计时
    MPI_Barrier(MPI_COMM_WORLD);
    double start_time = MPI_Wtime();

    if(rank == 0) {
        // 进程 0 将对应的 A 子块和完整的 B 发送给其他进程
        for(i = 1; i < size; i++){
            int proc_rows = (i < remainder) ? rows_per_proc + 1 :
               rows_per_proc;
            int proc_offset;
            if(i < remainder)
                proc_offset = i * (rows_per_proc + 1);
```

```
            else
                proc_offset = remainder * (rows_per_proc + 1) + (i -
                    remainder) * rows_per_proc;
            MPI_Send(&A[proc_offset * n], proc_rows * n, MPI_DOUBLE,
                i, 1, MPI_COMM_WORLD);
            MPI_Send(B, n * k, MPI_DOUBLE, i, 2, MPI_COMM_WORLD);
        }
        // 进程 0 自己拷贝 A 的第一部分到 local_A
        for(i = 0; i < local_rows * n; i++){
            local_A[i] = A[i];
        }
    }
    else {
        // 其他进程接收对应的 A 子块和完整矩阵 B
        MPI_Recv(local_A, local_rows * n, MPI_DOUBLE, 0, 1,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(B, n * k, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
```

6. 每个进程计算自己的局部矩阵乘法：local_C = local_A * B:

```
    matrix_multiply(local_A, B, local_C, local_rows, n, k);
```

7. 进程 0 收集所有进程计算得到的局部矩阵乘法结果，并将其合并到完整的结果矩阵 C 中:

```
    // 收集各进程计算得到的局部结果到进程 0
    if(rank == 0) {
        // 将进程 0 的局部结果拷贝到 C 的相应位置
        for(i = 0; i < local_rows * k; i++){
            C[i] = local_C[i];
        }
        // 接收其他进程计算的结果
        for(i = 1; i < size; i++){
            int proc_rows = (i < remainder) ? rows_per_proc + 1 :
                rows_per_proc;
            int proc_offset;
            if(i < remainder)
                proc_offset = i * (rows_per_proc + 1);
            else
                proc_offset = remainder * (rows_per_proc + 1) + (i -
                    remainder) * rows_per_proc;
```

```
            MPI_Recv(&C[proc_offset * k], proc_rows * k, MPI_DOUBLE,
                i, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    } else {
        // 其他进程将局部结果发送给进程 0
        MPI_Send(local_C, local_rows * k, MPI_DOUBLE, 0, 3,
            MPI_COMM_WORLD);
    }
    //截止并计算时间
    double end_time = MPI_Wtime();
    double elapsed_time = end_time - start_time;
```

8. 进程 0 打印矩阵 A、B、C 以及整个矩阵乘法计算的耗时:

```
    // 进程 0 输出矩阵和计算时间
    if(rank == 0) {
        printf("\n矩阵 A (%d x %d):\n", m, n);
        print_matrix(A, m, n);
        printf("\n矩阵 B (%d x %d):\n", n, k);
        print_matrix(B, n, k);
        printf("\n矩阵 C (%d x %d):\n", m, k);
        print_matrix(C, m, k);
        printf("\n矩阵乘法计算时间: %f 秒\n", elapsed_time);
    }
```

9. 释放所有分配的内存，并结束 MPI 环境:

```
    // 释放内存
    if(A) free(A);
    if(B) free(B);
    if(C) free(C);
    if(local_A) free(local_A);
    if(local_C) free(local_C);

    MPI_Finalize();
    return 0;
```

实验结果

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.004115	0.030273	0.276359	2.851809	51.51469
2	0.005620	0.015893	0.151006	1.395535	29.738463
4	0.002897	0.015424	0.079385	1.500024	17.601987
8	0.003243	0.016818	0.082175	0.863957	12.623343
16	0.002053	0.011770	0.080927	0.839582	12.907987

表 1: 实验结果汇总

128 规模时，1 进程需要 0.004115 s，2 进程时反而变为 0.005620 s，4 进程时出现 0.02897 s 这种更高的数值（可能是一次性调度或测试时负载导致的异常）；到 8、16 进程时时间又降到 0.003243 s、0.002053 s，而总体 8 进程与 16 进程时的运行时间差距并不大，符合“小规模计算通信开销占比较大，测量噪声明显；大规模计算加速效果明显，但加速效果可能随着进程数增大而减缓”的趋势.

优化方向

Q: 在内存有限的情况下，如何进行大规模矩阵乘法计算?

A: 本次实验将矩阵的不同部分分配给不同进程，类似地，内存不足时我们可以将矩阵分块载入内存进行运算，拥有多个内存较小的节点时，可以将大矩阵分散存储在不同节点上，汇总运算结果;

Q: 如何提高大规模稀疏矩阵乘法性能?

A: 可以在运算前改造矩阵，仅保存非零元素及索引，或者对矩阵进行重排.

实验感想

首先是要熟悉 MPI 库函数的调用，其次是要分清进程 0 和其他进程的区别，最后不要忘记给进程 0 也分配一份运算工作。